По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-137-0, название «Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

The Rails Way

Obie Fernandez

Addison-Wesley

HIGH TECH

Путь Rails

Подробное руководство по созданию приложений в среде Ruby on Rails

Оби Фернандес

Санкт-Петербург — Москва
2009

Серия «High tech»

Оби Фернандес

Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails

Перевод А. Слинкина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*Л. Пискунова*
Редактор	*Е. Бекназарова*
Корректор	*Т. Золотова*
Верстка	*Н. Пискунова*
Художник	*В. Гренда*

Фернандес О.
Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails. – Пер. с англ. – СПб: Символ-Плюс, 2009. – 768 с., ил.

ISBN13: 978-5-93286-137-0
ISBN10: 5-93286-137-1

Среда Ruby on Rails стремительно занимает ведущее место в ряду наиболее популярных платформ для разработки веб-приложений. Она основана на одном из самых элегантных языков программирования, Ruby, и доставляет истинное удовольствие своим приверженцам. Хотите оказаться в первых рядах? Тогда эта книга для вас! Ее автор, Оби Фернандес, и целая группа экспертов подробно описывают основные возможности и подсистемы Rails: контроллеры, маршрутизацию, поддержку стиля REST, объектно-реляционное отображение с помощью библиотеки ActiveRecord, применение технологии AJAX в Rails-приложениях и многое другое. Отталкиваясь от своего уникального опыта и приводя подробные примеры кода, Оби демонстрирует, как с помощью инструментов и рекомендованных методик Rails добиться максимальной продуктивности и получать наслаждение от создания совершенных приложений.

ISBN13: 978-5-93286-137-0
ISBN10: 5-93286-137-1
ISBN 0-321-44561-9 (англ)

© Издательство Символ-Плюс, 2009

Authorized translation from the English language edition, entitled RAILS WAY, THE, 1st Edition, ISBN 0321445619, by FERNANDEZ, OBIE, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2008 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Russian language edition published by SYMBOL-PLUS PUBLISHING LTD, Copyright © 2009.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 3245353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 23.10.2008. Формат 70x100 $^{1}/_{16}$. Печать офсетная.
Объем 48 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Дези – моей любимой, подруге, музе.

Оглавление

Предисловие 22
Благодарности 22
Об авторе 26
Введение 27

1. Среда и конфигурирование Rails 38

Запуск 39
Параметры среды по умолчанию 39
Начальная загрузка 40
Пакеты RubyGem 42
Инициализатор 42
Подразумеваемые пути загрузки 42
Rails, модули и код автозагрузки 43
Встройка Rails Info 44
Конфигурирование 45
Дополнительные конфигурационные параметры 49
Режим разработки 49
Динамическая перезагрузка классов 50
Загрузчик классов в Rails 50
Режим тестирования 52
Режим эксплуатации 52
Протоколирование 53
Протоколы Rails 55
Анализ протоколов 56
Syslog 58
Заключение 59

2. Работа с контроллерами 60

Диспетчер: с чего все начинается 61
Обработка запроса 61
Познакомимся с диспетчером поближе 62

Рендеринг представления ... 64
Если сомневаетесь, рисуйте ... 64
Явный рендеринг ... 65
Рендеринг шаблона другого действия ... 65
Рендеринг совершенно постороннего шаблона ... 66
Рендеринг подшаблона ... 67
Рендеринг встроенного шаблона ... 67
Рендеринг текста ... 67
Рендеринг структурированных данных других типов ... 68
Пустой рендеринг ... 68
Параметры рендеринга ... 68
Переадресация ... 71
Коммуникация между контроллером и представлением ... 74
Фильтры ... 75
Наследование фильтров ... 76
Типы фильтров ... 77
Упорядочение цепочки фильтров ... 78
Around-фильтры ... 78
Пропуск цепочки фильтров ... 80
Условная фильтрация ... 80
Прерывание цепочки фильтров ... 81
Потоковая отправка ... 81
send_data(data, options = {}) ... 81
send_file(path, options = {}) ... 82
Как заставить сам веб-сервер отправлять файлы ... 85
Заключение ... 86

3. Маршрутизация ... 87

Две задачи маршрутизации ... 88
Связанные параметры ... 90
Метапараметры («приемники») ... 91
Статические строки ... 91
Файл routes.rb ... 93
Маршрут по умолчанию ... 94
О поле :id ... 95
Генерация маршрута по умолчанию ... 96
Модификация маршрута по умолчанию ... 97
Предпоследний маршрут и метод respond_to ... 97
Метод respond_to и заголовок HTTP-Accept ... 98
Пустой маршрут ... 99
Самостоятельное создание маршрутов ... 100
Использование статических строк ... 100
Использование собственных «приемников» ... 101

Замечание о порядке маршрутов 102
Применение регулярных выражений в маршрутах 103
Параметры по умолчанию и метод url_for 104
Что случилось с :id 105
Использование литеральных URL 106
Маскирование маршрутов 106
Маскирование пар ключ/значение 107
Именованные маршруты 108
Создание именованного маршрута 108
Что лучше: name_path или name_url? 108
Замечания 109
Как выбирать имена для маршрутов 109
Синтаксическая глазурь 111
Еще немного глазури? 111
Метод организации контекста with_options 112
Заключение 113

4. REST, ресурсы и Rails 114

О REST в двух словах 115
REST в Rails 116
Маршрутизация и CRUD 117
Ресурсы и представления 118
Ресурсы REST и Rails 118
От именованных маршрутов к поддержке REST 119
И снова о глаголах HTTP 120
Стандартные REST-совместимые действия контроллеров 121
Хитрость для методов PUT и DELETE 122
Одиночные и множественные REST-совместимые маршруты 123
Специальные пары: new/create и edit/update 123
Одиночные маршруты к ресурсам 124
Вложенные ресурсы 125
Явное задание :path_prefix 127
Явное задание :name_prefix 127
Явное задание REST-совместимых контроллеров 129
А теперь все вместе 129
Замечания 131
О глубокой вложенности 131
Настройка REST-совместимых маршрутов 133
Маршруты к дополнительным действиям 133
Дополнительные маршруты к наборам 134
Замечания 134
Ресурсы, ассоциированные только с контроллером 136

Различные представления ресурсов 138
Метод respond_to 138
Форматированные именованные маршруты 139
Набор действий в Rails для REST 139
index 140
show 143
destroy 143
new и create 144
edit и update 146
Заключение 146

5. Размышления о маршрутизации в Rails 147

Исследование маршрутов в консоли приложения 147
Распечатка маршрутов 148
Анатомия объекта Route 149
Распознавание и генерация с консоли 151
Консоль и именованные маршруты 153
Тестирование маршрутов 153
Подключаемый модуль Routing Navigator 155
Заключение 156

6. Работа с ActiveRecord 157

Основы 158
Миграции 160
Создание миграций 161
Migration API 164
Определение колонок 166
Методы в стиле макросов 171
Объявление отношений 172
Примат соглашения над конфигурацией 173
Приведение к множественному числу 173
Задание имен вручную 175
Унаследованные схемы именования 175
Определение атрибутов 176
Значения атрибутов по умолчанию 177
Сериализованные атрибуты 179
CRUD: создание, чтение, обновление, удаление 179
Создание новых экземпляров ActiveRecord 179
Чтение объектов ActiveRecord 180
Чтение и запись атрибутов 182
Доступ к атрибутам и манипулирование ими до приведения типов 184
Перезагрузка 185

Динамический поиск по атрибутам ... 185
Специальные SQL-запросы ... 186
Кэш запросов ... 187
Обновление ... 189
Обновление с условием ... 190
Обновление конкретного экземпляра ... 191
Обновление конкретных атрибутов ... 191
Вспомогательные методы обновления ... 192
Контроль доступа к атрибутам ... 192
Удаление и уничтожение ... 193
Блокировка базы данных ... 194
Оптимистическая блокировка ... 194
Пессимистическая блокировка ... 196
Замечание ... 197
Дополнительные средства поиска ... 197
Условия ... 198
Упорядочение результатов поиска ... 199
Параметры limit и offset ... 200
Параметр select ... 201
Параметр from ... 201
Группировка ... 202
Параметры блокировки ... 202
Соединение и включение ассоциаций ... 202
Параметр readonly ... 203
Соединение с несколькими базами данных в разных моделях ... 203
Прямое использование соединений с базой данных ... 204
Модуль DatabaseStatements ... 204
Другие методы объекта connection ... 206
Другие конфигурационные параметры ... 208
Заключение ... 209

7. Ассоциации в ActiveRecord ... 211

Иерархия ассоциаций ... 211
Отношения один-ко-многим ... 213
Добавление ассоциированных объектов в набор ... 215
Методы класса AssociationCollection ... 215
Ассоциация belongs_to ... 218
Перезагрузка ассоциации ... 218
Построение и создание связанных объектов через ассоциацию ... 219
Параметры метода belongs_to ... 220
Ассоциация has_many ... 225
Параметры метода has_many ... 225
Методы прокси-классов ... 232

Отношения многие-ко-многим ... 233
Метод has_and_belongs_to_many ... 233
Конструкция has_many :through ... 240
Параметры ассоциации has_many :through ... 244
Отношения один-к-одному ... 247
Ассоциация has_one ... 247
Параметры ассоциации has_one ... 249
Несохраненные объекты и ассоциации ... 251
Ассоциации один-к-одному ... 251
Наборы ... 252
Расширения ассоциаций ... 252
Класс AssociationProxy ... 253
Методы reload и reset ... 253
Методы proxy_owner, proxy_reflection и proxy_target ... 253
Заключение ... 255

8. Валидаторы в ActiveRecord ... 256

Нахождение ошибок ... 256
Простые декларативные валидаторы ... 257
validates_acceptance_of ... 257
validates_associated ... 258
validates_confirmation_of ... 258
validates_each ... 259
validates_inclusion_of и validates_exclusion_of ... 259
validates_existence_of ... 260
validates_format_of ... 261
validates_length_of ... 262
validates_numericality_of ... 262
validates_presence_of ... 262
validates_uniqueness_of ... 263
Исключение RecordInvalid ... 264
Общие параметры валидаторов ... 264
:allow_nil ... 265
:if ... 265
:message ... 265
:on ... 265
Условная проверка ... 266
Замечания по поводу применения ... 266
Работа с объектом Errors ... 267
Манипулирование набором Errors ... 268
Проверка наличия ошибок ... 268
Нестандартный контроль ... 268
Отказ от контроля ... 270
Заключение ... 271

9. Дополнительные возможности ActiveRecord 272

Обратные вызовы 272
Регистрация обратного вызова 273
Парные обратные вызовы before/after 274
Прерывание выполнения 275
Примеры применения обратных вызовов 275
Особые обратные вызовы: after_initialize и after_find 278
Классы обратных вызовов 279
Наблюдатели 282
Соглашения об именовании 282
Регистрация наблюдателей 283
Момент оповещения 283
Наследование с одной таблицей 283
Отображение наследования на базу данных 285
Замечания об STI 287
STI и ассоциации 288
Абстрактные базовые классы моделей 290
Полиморфные отношения has_many 291
Случай модели с комментариями 291
Замечание об ассоциации has_many 294
Модули как средство повторного использования общего поведения 294
Несколько слов об области видимости класса и контекстах 297
Обратный вызов included 298
Модификация классов ActiveRecord во время выполнения 299
Замечания 300
Ruby и предметно-ориентированные языки 301
Заключение 302

10. ActionView 303

Основы ERb 304
Практикум по ERb 304
Удаление пустых строк из вывода ERb 306
Закомментирование ограничителей ERb 306
Условный вывод 306
RHTML? RXML? RJS? 307
Макеты и шаблоны 307
Подстановка содержимого 308
Переменные шаблона 310
Защита целостности представления от данных, введенных пользователем 313
Подшаблоны 314
Простые примеры 314
Повторное использование подшаблонов 316

Разделяемые подшаблоны ... 316
Передача переменных подшаблонам ... 317
Рендеринг наборов ... 319
Протоколирование ... 320
Кэширование ... 320
Кэширование в режиме разработки? ... 321
Кэширование страниц ... 321
Кэширование действий ... 321
Кэширование фрагментов ... 323
Истечение срока хранения кэшированного содержимого ... 326
Автоматическая очистка кэша с помощью дворников ... 328
Протоколирование работы кэша ... 329
Подключаемый модуль Action Cache ... 329
Хранилища для кэша ... 330
Заключение ... 332

11. Все о помощниках ... 333

Модуль ActiveRecordHelper ... 333
Отчет об ошибках контроля ... 334
Автоматическое создание формы ... 335
Настройка выделения ошибочных полей ... 338
Модуль AssetTagHelper ... 339
Помощники для формирования заголовка ... 339
Только для подключаемых модулей: добавление включаемых по умолчанию JavaScript-сценариев ... 343
Модуль BenchmarkHelper ... 343
Модуль CacheHelper ... 343
Модуль CaptureHelper ... 344
Модуль DateHelper ... 345
Помощники для выбора даты и времени ... 345
Помощники для задания отдельных элементов даты и времени ... 346
Параметры, общие для всех помощников, связанных с датами ... 349
Методы distance_in_time со сложными именами ... 349
Модуль DebugHelper ... 351
Модуль FormHelper ... 351
Создание форм для моделей ActiveRecord ... 351
Как помощники формы получают свои значения ... 358
Модуль FormOptionsHelper ... 359
Помощники select ... 359
Другие помощники ... 361
Модуль FormTagHelper ... 365

Модуль JavaScriptHelper ... 368
Модуль NumberHelper ... 370
Модуль PaginationHelper ... 372
will_paginate ... 372
paginator ... 373
Paginating Find ... 374
Модуль RecordIdentificationHelper ... 374
Модуль RecordTagHelper ... 375
Модуль TagHelper ... 376
Модуль TextHelper ... 378
Модуль UrlHelper ... 384
Написание собственных модулей ... 390
Мелкие оптимизации: помощник Title ... 390
Инкапсуляция логики представления: помощник photo_for ... 391
Более сложное представление: помощник breadcrumbs ... 392
Обертывание и обобщение подшаблонов ... 393
Помощник tiles ... 393
Обобщение подшаблонов ... 396
Заключение ... 399

12. Ajax on Rails ... 400

Библиотека Prototype ... 401
Подключаемый модуль FireBug ... 402
Prototype API ... 403
Функции верхнего уровня ... 403
Объект Class ... 405
Расширения класса JavaScript Object ... 406
Расширения класса JavaScript Array ... 407
Расширения объекта document ... 408
Расширения класса Event ... 409
Расширения класса JavaScript Function ... 410
Расширения класса JavaScript Number ... 412
Расширения класса JavaScript String ... 413
Объект Ajax ... 415
Объект Ajax.Responders ... 415
Объект Enumerable ... 416
Класс Hash ... 421
Объект ObjectRange ... 422
Объект Prototype ... 422
Модуль PrototypeHelper ... 422
link_to_remote ... 422
remote_form_for ... 426
periodically_call_remote ... 427

observe_field ... 428
observe_form ... 429
RJS – пишем Javascript на Ruby ... 429
RJS-шаблоны ... 431
<<(javascript) ... 432
[](id) ... 432
alert(message) ... 432
call(function, *arguments, &block) ... 432
delay(seconds = 1) { ... } ... 433
draggable(id, options = {}) ... 433
drop_receiving(id, options = {}) ... 433
hide(*ids) ... 433
insert_html(position, id, *options_for_render) ... 433
literal (code) ... 434
redirect_to(location) ... 434
remove(*ids) ... 434
replace(id, *options_for_render) ... 434
replace_html(id, *options_for_render) ... 434
select(pattern) ... 435
show(*ids) ... 435
sortable(id, options = {}) ... 435
toggle(*ids) ... 435
visual_effect(name, id = nil, options = {}) ... 435
JSON ... 435
Перетаскивание мышью ... 437
Сортируемые списки ... 439
Автозавершение ... 439
Редактирование на месте ... 440
Заключение ... 441

13. Управление сеансами ... 442

Что хранить в сеансе ... 443
Текущий пользователь ... 443
Рекомендации по работе с сеансами ... 443
Способы организации сеансов ... 444
Отключение сеансов для роботов ... 445
Избирательное включение сеансов ... 445
Безопасные сеансы ... 446
Хранилища ... 446
ActiveRecord SessionStore ... 446
PStore (на базе файлов) ... 447
Хранилище DRb ... 447
Хранилище memcache ... 448
Спорное хранилище CookieStore ... 449

Жизненный цикл сеанса и истечение срока хранения ... 451
Подключаемый модуль Session Timeout ... 451
Отслеживание активных сеансов ... 452
Повышенная безопасность сеанса ... 453
Удаление старых сеансов ... 453
Cookies ... 453
Чтение и запись cookies ... 454
Заключение ... 455

14. Регистрация и аутентификация ... 456

Подключаемый модуль Acts as Authenticated ... 457
Установка и настройка ... 457
Модель User ... 458
Класс AccountController ... 466
Получение имени пользователя из cookies ... 468
Текущий пользователь ... 469
Протоколирование в ходе тестирования ... 470
Заключение ... 471

15. XML и ActiveResource ... 472

Метод to_xml ... 472
Настройка результата работы to_xml ... 473
Ассоциации и метод to_xml ... 475
Продвинутое применение метода to_xml ... 476
Динамические атрибуты ... 477
Переопределение метода to_xml ... 478
Уроки реализации метода to_xml в классе Array ... 478
Класс XML Builder ... 480
Разбор XML ... 482
Преобразование XML в хеши ... 482
Библиотека XmlSimple ... 483
Приведение типов ... 484
Библиотека ActiveResource ... 485
Метод find ... 486
Метод create ... 487
Метод update ... 489
Метод delete ... 489
Заголовки ... 490
Настройка ... 491
Хешированные формы ... 492
Заключение ... 493

16. ActionMailer 494

Конфигурирование 494
Модели почтальона 495
Подготовка исходящего почтового сообщения 496
Почтовые сообщения в формате HTML 499
Многочастные сообщения 499
Вложение файлов 501
Отправка почтового сообщения 502
Получение почты 502
Справка по TMail::Mail API 503
Обработка вложений 504
Конфигурирование 505
Заключение 505

17. Тестирование 506

Терминология Rails, относящаяся к тестированию 507
К вопросу об изоляции... 508
Mock-объекты в Rails 509
Настоящие Mock-объекты и заглушки 510
Тесты сопряжения 511
О путанице в терминологии 512
Класс Test::Unit 513
Прогон тестов 514
Фикстуры 515
Фикстуры в формате CSV 516
Доступ к записям фикстуры из тестов 516
Динамические данные в фикстурах 517
Использование данных из фикстур в режиме разработки 518
Генерация фикстур из данных, используемых в режиме разработки 518
Параметры фикстур 520
Никто не любит фикстуры 520
Не все так плохо с фикстурами 522
Утверждения 523
Простые утверждения 523
Утверждения Rails 525
По одному утверждению в каждом тестовом методе 526
Тестирование моделей с помощью автономных тестов 527
Основы тестирования моделей 528
Что тестировать 529

Тестирование контроллеров
с помощью функциональных тестов 529
Структура и подготовка 530
Методы функциональных тестов 531
Типичные утверждения 531
Тестирование представлений
с помощью функциональных тестов 535
Тестирование поведения RJS 539
Другие методы выборки 540
Тестирование правил маршрутизации 541
Тесты сопряжения в Rails 542
Основы 543
API тестов сопряжения 543
Работа с сеансами 544
Задания Rake, относящиеся к тестированию 544
Приемочные тесты 545
Приемочные тесты с самого начала 546
Система Selenium 547
Основы 547
Приступая к работе 548
RSelenese 550
Заключение 551

18. RSpec on Rails 552

Введение в RSpec 553
Обязанности и ожидания 554
Предикаты 555
Нестандартные верификаторы ожиданий 556
Несколько примеров для одного поведения 557
Разделяемые поведения 558
Mock-объекты и заглушки в RSpec 561
Прогон «спеков» 564
Установка RSpec и подключаемого модуля RSpec on Rails 566
Подключаемый модуль RSpec on Rails 566
Генераторы 566
Спецификации модели 567
Спецификации контроллеров 570
Спецификации представлений 573
Спецификации помощников 574
Обстраивание 575
Инструменты RSpec 575
Autotest 575
RCov 576
Заключение 577

19. Расширение Rails с помощью подключаемых модулей 578

Управление подключаемыми модулями 579
Повторное использование кода 579
Сценарий plugin 580
Система Subversion и сценарий script/plugin 584
Использование Piston 586
Установка 586
Импорт внешней библиотеки 587
Конвертация существующих внешних библиотек 588
Обновление 588
Блокировка и разблокировка 588
Свойства Piston 589
Написание собственных подключаемых модулей 589
Точка расширения init.rb 590
Каталог lib 591
Расширение классов Rails 592
Файлы README и MIT-LICENSE 593
Файлы install.rb и uninstall.rb 594
Специальные задания Rake 595
Rakefile подключаемого модуля 596
Тестирование подключаемых модулей 597
Заключение 598

20. Конфигурации Rails в режиме эксплуатации 599

Краткая история промышленной эксплуатации Rails 600
Предварительные условия 601
Контрольный перечень 602
Серверное и сетевое окружение 603
Ярус веб-сервера 604
Ярус сервера приложений 604
Ярус базы данных 605
Мониторинг 605
Управление версиями 605
Установка 605
Ruby 606
Система RubyGems 606
Rails 607
Mongrel 607
Mongrel Cluster 607
Nginx 607
Subversion 608
MySQL 608

Monit 608
Capistrano 609
Конфигурация 609
Конфигурирование Mongrel Cluster 609
Конфигурирование Nginx 610
Конфигурирование Monit 614
Конфигурирование Capistrano 616
Конфигурирование сценариев init 616
Сценарий init для Nginx 616
Сценарий init для Mongrel 618
Сценарий init для Monit 619
Развертывание и запуск 620
Другие замечания по поводу промышленной системы 621
Избыточность и перехват управления при отказе 621
Кэширование 621
Производительность и масштабируемость 621
Безопасность 623
Удобство сопровождения 623
Заключение 623

21. Capistrano 625

Обзор системы Capistrano 626
Терминология 626
Основы 627
Что Capistrano сделала, а что – нет 628
Приступаем к работе 628
Установка 629
Готовим приложение Rails для работы с Capistrano 629
Конфигурирование развертывания 631
О сценарии spin 632
Подготовка машины развертывания 632
Развертываем! 634
Переопределение предположений Capistrano 634
Использование удаленной учетной записи пользователя 635
Изменение системы управления версиями, используемой Capistrano 635
Работа без доступа к системе управления версиями с машины развертывания 635
Если файл database.yml не хранится в репозитории СУВ 636
Если миграции не проходят без ошибок 638
Полезные рецепты Capistrano 639
Переменные и их область видимости 639
Упражнение 1. Промежуточный сервер 641
Упражнение 2. Управление другими службами 643

Развертывание на нескольких серверах ... 645
Транзакции ... 646
Доступ к машинам развертывания через прокси-серверы ... 648
Заключение ... 648

22. Фоновая обработка ... 649

Сценарий script/runner ... 650
Приступаем к работе ... 650
Несколько слов об использовании ... 651
Замечания по поводу script/runner ... 652
DRb ... 652
Простой DRb-сервер ... 652
Использование DRb из Rails ... 653
Замечания о DRb ... 654
Ресурсы ... 654
BackgrounDRb ... 654
Приступаем к работе ... 655
Конфигурирование ... 656
Знакомство с принципами работы BackgrounDRb ... 656
Использование класса MiddleMan ... 657
Подводные камни ... 658
Замечания о BackGrounDRb ... 659
Daemons ... 659
Порядок применения ... 659
Введение в потоки ... 660
Замечания о библиотеке Daemons ... 662
Заключение ... 662

A. Справочник по ActiveSupport API ... 663

B. Предметы первой необходимости для Rails ... 731

Послесловие ... 740

Алфавитный указатель ... 754

Предисловие

Rails – больше, чем среда для программирования веб-приложений. Это еще и тема для размышлений о веб-приложениях. В отличие от чистого листа бумаги, готового стерпеть выражение любых мыслей, она отдает предпочтение не столько гибкости, сколько удобству «делать то, что большинству людей необходимо в большинстве случаев для решения большинства задач». Для проектировщика она служит смирительной рубашкой, позволяющей не думать о том, что не имеет значения, и сосредоточиться на действительно важных вещах.

Чтобы принять такой компромисс, вы должны понимать не только, как нечто делается в Rails, но и почему это делается именно так. Лишь поняв причину, вы сможете обратить платформу себе на пользу, а не бороться с ней. Это не означает, что вы всегда будете вынуждены соглашаться с определенным решением, но вам придется уживаться с пронизывающим среду принципом примата соглашений. Вы должны будете научиться жертвовать личными пристрастиями и смирять свой нрав, а наградой будет повышение производительности собственного труда.

Эта книга поможет вам. Она не только станет гидом при изучении возможностей Rails, но и позволит вам проникнуть в его мысли и душу. Почему мы решили поступить так, а не иначе, почему мы осуждаем некоторые широко распространенные подходы? Включены даже дискуссии и рассказы о том, как мы пришли к определенному мнению, прямо со слов членов сообщества, которые принимали участие в выработке решения.

Научиться писать приложение «Здравствуй, мир» в Rails нетрудно и самостоятельно, но вот осознать целостную структуру Rails гораздо сложнее. Я аплодирую Оби, который взялся сопровождать вас в этом путешествии. Наслаждайтесь!

Дэвид Хейнемейер Хэнссон, создатель Ruby on Rails

Благодарности

Особая сердечная благодарность моему редактору Дебре Уильямс Коули (Debra Williams Cauley), которая разглядела мой потенциал и полтора года терпеливо вела меня по тернистому пути к завершению этой

книги. Это одна из самых милых и умных женщин, которых я знаю, и я надеюсь на долгую и продуктивную дружбу с ней. Все остальные члены команды издательства Addison-Wesley тоже проявляли недюжинное терпение, всегда были готовы помочь и поддержать меня: Сан Ди Филлипс (San Dee Phillips), Сон Линь Ки (Songlin Qiu), Мэнди Фрэнк (Mandie Frank), Мари Мак-Кинли (Marie McKinley) и Хэзер Фокс (Heather Fox).

Разумеется, моей семье пришлось потерпеть, пока я был поглощен работой над книгой, особенно на протяжении последних шести месяцев (когда сроки сдачи начали неумолимо приближаться). Тэйлор, Лиам и Дэзи, я вас очень люблю. Спасибо, что вы предоставили мне время и создали комфортные условия для работы. У меня нет слов, чтобы выразить благодарность моей давней подруге (и любимой разработчице на Rails), Дэзи Мак-Адам (Desi McAdam), которая сняла с меня на время все домашние обязанности. Также хочу сказать спасибо своему отцу Марко, который прозорливо предположил, что я задумал не одну книгу, а целую серию, посвященную языку Ruby.

Написание книги такого охвата и объема не может быть трудом одиночки – я в неоплатном долгу перед многими людьми, предлагавшими свои идеи и критические замечания. Дэвид Блэк (David Black) на ранних этапах дал мне толчок, поделившись материалами о маршрутизации и контроллерах. Пишущие для этой серии авторы – Джеймс Адам (James Adam), Тротер Кэшн (Trotter Cashion) и Мэтт Пеллетье (Matt Pelletier) – тоже не остались в стороне, подкинув мне материал о подключаемых модулях, пакете `ActiveRecord` и развертывании в промышленной среде соответственно. Мэтт Бауэр (Matt Bauer) помог мне закончить главы, посвященные Ajax и XML, а Джоди Шоуэрс (Jodi Showers) написал главу о Capistrano. Пэт Мэддокс (Pat Maddox) помог разобраться с особо неприятной ситуацией, касающейся метода изменения атрибута, и посодействовал в завершении главы об Rspec, на которую Дэвид Челимски (David Chelimsky) позже написал квалифицированную рецензию. Чарлз Брайан Куинн (Charles Brian Quinn) и Пратик Наик (Pratik Naik) очень вовремя предложили консультацию и материалы по фоновой обработке. Диего Скатаглини (Diego Scataglini) также рецензировал некоторые из последующих глав.

Фрэнсис Хуанг (Francis Hwang) и Себастьян Делмонт (Sebastian Delmont) занимались техническим рецензированием, начиная с самых ранних этапов работы над книгой. Позже к ним присоединились Уилсон Билкович (Wilson Bilkovich) и Кортенэ Гаскинг (Courtenay Gasking), которые, помимо написания оригинального основного текста и врезок, здорово помогли мне уложиться в сроки. В частности, Уилсон со своим неподражаемым юмором постоянно развлекал меня, а заодно переписал главу о фоновой обработке, обогатив ее глубоким знанием предмета. Из других ценных рецензентов хочу упомянуть Сэма Аарона (Sam Aaron), Нола Стоу (Nola Stowe) и Сьюзан Поттер (Susan Potter). В последние несколько недель мой новый друг и коллега Джон «Lark» Лар-

ковски (Jon «Lark» Larkowski) обнаружил еще несколько ошибок, которые остальные рецензенты почему-то проглядели.

Пакостник Зед Шоу (Zed Shaw) делил со мной кров на протяжении почти всего времени работы над книгой и являлся постоянным источником вдохновения. Также огромное спасибо всем моим друзьям по IRC-каналу #caboose за квалифицированные советы и мнения. Кого-то я уже упомянул, а ниже привожу список остальных: Джош Сассер (Josh Susser – hasmanyjosh), Рик Олсон (Rick Olson – technoweenie), Эзра Зигмунтович (Ezra Zygmuntovich – ezmobius), Джеффри Грозенбах (Geoffrey Grosenbach – topfunky), Робби Рассел (Robby Russel – robbyonrails), Джереми Хуберт (Jeremy Hubert), Дэйв Фейрам (Dave Fayram – kirindave), Мэтт Лайон (Matt Lyon – mattly), Джошуа Сирлс (Joshua Sierles – corp), Дэн Петерсон (Dan Peterson – danp), Дэйв Эстелс (Dave Astels – dastels), Тревор Сквайрс (Trevor Squires – protocool), Дэвид Гудлэд (David Goodlad – dgoodlad), Эми Хой (Amy Hoy – eriberri), Джош Гебел (Josh Goebel – Dreamer3), Эван Феникс (Evan Phoenix – evan), Райан Дэвис (Ryan Davis – zenspider), Майкл Шуберт (Michael Schubert), Кристи Балан (Cristi Balan – evilchelu), Джеми ван Дайк (Jamie van Dyke – fearoffish), Ник Вильямс (Nic Williams – drnick), Эрик Ходель (Eric Hodel – drbrain), Джеймс Кокс (James Cox – imajes), Кевин Кларк (Kevin Clark – kevinclark), Томас Фукс (Thomas Fuchs – madrobby), Манфред Стинстра (Manfred Stienstra – manfred-s), Пейсти Пейст Бот (Pastie Paste Bot – pastie), Эван Хэншоу-Плат (Evan Henshaw-Plath – rabble), Роб Орсини (Rob Orsini – rorsini), Адам Кейс (Adam Keys – therealadam), Джон Этейд (John Athayde – boborishi), Роберт Буске (Robert Bousquet), Брайан Хеллкамп (Bryan Helkamp – brynary) и Чед Фоулер (Chad Fowler). Еще я просто обязан включить самого Дэвида Хейнемейера Хэнссона (nextangler), который всегда оказывал мне поддержку и даже ответил на несколько головоломных вопросов.

Целый ряд моих бывших коллег по компании ThoughtWorks косвенно помогли родиться этой книге. Первый, и самый главный, Мартин Фаулер (Martin Fowler) – не только источник вдохновения, он помог мне наладить отношения с издательством Addison-Wesley и помогал советом и поддержкой на ранних стадиях проекта, когда я еще не понимал, во что ввязался. Исполнительный директор ThoughtWorks Рой Сингэм (Roy Singham) осознал заложенный в Rails потенциал еще в 2005 году, когда я познакомил его с Дэвидом Хэнссоном, и с тех пор твердо поддерживал мою деятельность по пропаганде Ruby (и даже сам иногда агитировал за него в разных уголках мира). Особенно важно это было на начальных этапах, когда многие уважаемые сотрудники ThoughtWorks считали, что Ruby – очередная дешевка, которая скоро умрет своей смертью.

В компании ThoughtWorks немало других людей, которых я должен поблагодарить. Прежде всего, горжусь тем, что был первым наставником Джея Филдса (Jay Fields) в Ruby и могу сослаться на него как на живое доказательство того, что ученик может превзойти учителя. Карлосу Виллела (Carlos Villela) я отчасти обязан интересом к языку Ruby,

который поначалу мне не понравился. Со своим приятелем Упендра Канда (Upendra Kanda) я провел долгие месяцы, работая над первыми платными заказами на Rails. Выдающийся хакер Майкл Грейнджер (Michael Granger) поведал мне, как применять Ruby нетрадиционными способами. Фред Джордж (Fred George) – мой наставник и источник вдохновения. Стив Уилкинс (Steve Wilkins) и Ферн Шифф (Fern Schiff) одними из первых поддержали Rails как среду для разработки коммерческих приложений. Среди других энтузиастов Rails в ThoughtWorks, которые так или иначе помогали мне, хочу упомянуть Бадри Янакириман (Badri Janakiraman), Рохана Кини (Rohan Kini), Картика Си (Kartik C), Пола Хамманта (Paul Hammant), Робина Гибсона (Robin Gibson), Ника Дрю (Nick Drew), Шрихари Сринивасан (Srihari Srinivasan), Джулиан Бут (Julian Boot), Йона Тирсена (Jon Tirsen), Крис Стивенсон (Chris Stevenson), Алекса Верховски (Alex Verkhovsky), Патрика Фарли (Patrick Farley), Нила Форда (Neal Ford), Рона Кэмпбелла (Ron Campbell), Джулиас Шоу (Julias Shaw), Дэвида Райса (David Rice), Джеффа Паттона (Jeff Patton), Кента Спиллнера (Kent Spillner), Джона Хьюма (John Hume), Джейка Скраггса (Jake Scruggs) и Стивена Чу (Stephen Chu).

В своей оплачиваемой работе с Rails я сталкивался с прогрессивно мыслящими и готовыми идти на риск заказчиками, хорошими людьми, которые доверили мне и моей команде разработку решений на неопробованной платформе. Всех попадающих в эту категорию я, наверное, не назову поименно, но некоторые мне особенно запомнились. Благодарю Хэнка Роарка (Hank Roark), Тома Хорсли (Tom Horsley), Говарда Тодда (Howard Todd), Джеффа Элама (Jeff Elam) и Кэрол Ринауро (Carol Rinauro) из компании Deere за решительную поддержку и доверие. Также спасибо Роберту Брауну (Robert Brown) из Barclays за то, что он умеет заглядывать вперед, а также за дружбу и поддержку. С Дирком и Бреттом Элмендорфами (Dirk и Brett Elmendorf) из Rackspace было очень приятно работать.

Сотрудники компании InfoQ.com всегда были готовы к сотрудничеству и с пониманием относились к нехватке у меня времени на завершающих стадиях работы над книгой. Особая благодарность Флойду Маринеску (Floyd Marinescu) и Диане Плеза (Diana Plesa) за терпение. Также благодарю всю мою команду разработчиков на Ruby, в том числе Вернера Шустера (Werner Schuster) и Себастьена Оврея (Sebastien Auvray).

В начале января 2007 года мой друг Марк Смит (Mark Smith) подарил мне счастливую возможность работать с коллективом блестящих специалистов над интересными Web 2.0-проектами в солнечном городке Атлантик Бич во Флориде. Команда First Street Live в любой момент была готова прийти на помощь: Мэриан Филан (Marian Phelan), Райан Роданд (Ryan Poland), Ник Стрейт (Nick Strate), Джо Хант (Joe Hunt), Клэй Кромберг (Clay Kromberg), Дена Фриман (Dena Freeman) и все остальные… (Марк, ты просто умница, мы навсегда останемся друзьями. Спасибо за прекрасную обстановку для работы, за книжные вечеринки

и за постоянную стимуляцию мозгов. Честное слово, я вряд ли сумел бы закончить эту книгу, если бы не приехал сюда поработать для тебя.)

И напоследок я выражаю самую сердечную признательность ПрагДэйву Томасу (PragDave Thomas), который, как говорят, заметил: «Оби, пишущий книгу о Rails – все равно что маркиз де Сад, пишущий книгу о том, как вести себя за столом».

Об авторе

Оби Фернандес – признанный технический специалист и независимый консультант. Он возился с компьютерами еще в 80-е годы, когда приобрел свой первый Commodore VIC-20, и, оказавшись в нужном месте в нужное время, в середине 90-х стал работать программистом на Java над некоторыми из самых первых проектов масштаба предприятия. В 1998 году Оби переехал в Атланту, штат Джорджия, и стал широко известен как ведущий архитектор успешной местной, вновь образованной компании MediaOcean. Он также основал группу пользователей экстремального программирования (Extreme Programming User Group), позже переименованную в Agile Atlanta, и в течение нескольких лет был ее президентом и организатором. В 2004 году Оби Фернандес вернулся к корпоративным программам и начал работу над несколькими рискованными, но прогрессивными проектами во всемирно известной консалтинговой компании ThoughtWorks.

С начала 2005 года Оби пропагандирует в Интернете Ruby и Rails посредством блогов и онлайновых публикаций, вызывая некоторую неприязнь (и грубости) со стороны старых приятелей по сообществу разработчиков программ с открытыми исходными текстами на Java. Он регулярно выступает с докладами на различных конференциях и встречах пользователей, а иногда даже проводит учебные семинары для корпораций и групп, желающих приобщиться к разработке на платформе Rails.

В настоящее время Оби специализируется на разработке и продвижении на рынок крупномасштабных веб-приложений. Он по-прежнему почти ежедневно пишет заметки на разные темы в своем блоге по адресу *http://obiefernandez.com*.

Введение

В конце 2004 года мы с моим добрым приятелем Аслаком Хеллесоем (Aslak Hellesoy)[1] консультировали одну из крупных американских компаний по производству автомобилей. Работа была интересной, изобиловала сложными политическими ситуациями, техническими неудачами и сорванными сроками. Речь шла не только о наших обычных сроках – клиенту грозили штрафы на миллионы долларов в день, если бы мы не справились вовремя. Давление было то еще!

Как-то, находясь на перепутье, команда решила положить в основу нашей системы непрерывной интеграции любимый проект Аслака под названием DamageControl. Это был написанный на Ruby вариант достопочтенного сервера CruiseControl, созданного компанией ThoughtWorks, в которой мы работали. Проблема состояла в том, что DamageControl – как бы помягче сказать? – не был готовым продуктом. И, как и многое, связанное с Ruby, не слишком хорошо работал в Windows. Тем не менее, уж не помню по какой причине, мы были вынуждены развернуть его на старом сервере Windows 2000, где попутно хранился репозиторий исходных текстов StarTeam (бывает же такое!).

Аслаку нужна была помощь, на протяжении нескольких недель мы с ним на пару занимались программированием прикладного кода DamageControl и правкой написанных на C библиотек управления процессами на платформе Win32 для Ruby. К тому времени у меня за спиной был восьмилетний опыт серьезного программирования корпоративных систем на Java и глубокая привязанность к великолепной интегрированной среде разработки IntelliJ IDE. Не могу передать, как сильно я ненавидел Ruby в тот момент своей карьеры.

Так что же изменилось? Сразу скажу, что из той передряги я выбрался живым и взялся за сравнительно легкий проект за границей, покинув на время лондонское отделение ThoughtWorks. Примерно через месяц Ruby снова привлек мое внимание, на этот раз в связи с оживленным обсуждением в блогосфере касательно скорого появления новой плат-

[1] Аслак хорошо известен в кругах разработчиков программ с открытыми исходными кодами на Java, главным образом как автор XDoclet.

формы для разработки веб-приложений под названием Ruby on Rails. Я решил дать Ruby еще один шанс. Кто знает, вдруг он не так уж плох? На новой платформе я быстро написал социальную сеть для внутреннего использования в компании ThoughtWorks.

И этот первый эксперимент с Rails, занявший несколько недель в феврале 2005 года, изменил мою жизнь. Весь опыт, накопленный мной за годы создания веб-приложений, был аккумулирован в единой среде, написанной так элегантно и лаконично, как мне еще не приходилось видеть. Мой интерес к Java мгновенно угас (хотя на то, чтобы прекратить пользоваться IntelliJ, ушел еще почти год). Я начал активно писать в блогах о Ruby и Rails и настойчиво пропагандировать его внутри и за пределами ThoughtWorks. Все остальное, как говорится, история.

В 2007 году основанные на использовании Rails проекты, которым я положил начало в ThoughtWorks, приносят компании половину годового дохода. Организован большой отдел, выпускающий коммерческое ПО на базе Ruby. В том числе и программу CruiseControl.rb, которую, как я подозреваю, Аслак хотел написать с самого начала. Она удостоилась чести стать официальным сервером непрерывной интеграции, используемым командой разработчиков ядра Ruby on Rails.

Ruby и Rails

Почему опытные разработчики корпоративных приложений вроде меня влюбляются в Ruby и Rails? Для удовлетворения предъявляемых требований сложность решений, создаваемых с применением технологий Java и Microsoft, просто неприемлема. Излишняя сложность не позволяет отдельному человеку понять проект в целом и сильно усложняет коммуникацию внутри команды. Из-за упора на следование паттернам проектирования и зацикленности на производительности на этих платформах пропадает удовольствие от работы над приложением.

> *В сообществе Rails нет места принуждению. DHH (David Heinemeier Hansson) выбрал язык, который доставлял ему радость. Платформа Rails родилась из кода, который представлялся ему красивым. Это и задало тон общения в сообществе Rails. Все в мире Rails субъективно. Человек либо приемлет что-то, либо нет. Но между теми, кто приемлет, и теми, кто не приемлет, нет злобы, а лишь кроткая попытка убедить.*
>
> Пэт Мэддокс

Ruby – красивый язык. Кодировать на Ruby приятно. Все мои знакомые, перешедшие на Ruby, говорят, что стали счастливее. Главным образом по этой причине Ruby и Rails изменяют статус кво, особенно в области разработки корпоративных приложений. Прежде чем стать

приверженцем Rails, я привык работать над проектами с нечеткими требованиями, не имеющими отношения к реальным потребностям. Я устал выбирать между конкурирующими платформами и интегрировать их между собой, устал писать уродливый код.

А Ruby – динамический, высокоуровневый язык. Код на Ruby проще читать и писать, поскольку он более естественно отображается на конкретную предметную область и по стилю ближе к естественному человеческому языку. Удобство восприятия имеет массу преимуществ не только в краткосрочной, но и в долгосрочной перспективе, поскольку программа передается в промышленную эксплуатацию и должна быть понятна программистам сопровождения.

Мой опыт показывает, что в программах, написанных на Ruby, меньше строк, чем в аналогичных программах на языках Java и C#. А чем меньше кода, тем проще его сопровождать, что немаловажно, так как затраты на долгосрочное сопровождение считаются самой крупной стоимостной составляющей успешных программных проектов. Отладка небольших программ занимает меньше времени даже без «навороченных» инструментов отладки.

Восхождение Rails и принятие его в качестве одной из основных платформ

Как и вся идеология гибкой (agile) разработки, породившая эту платформу, Rails нацелен прежде всего на удовлетворение нужд разработчиков приложений – не архитекторов ПО и, уж конечно, не компьютерных теоретиков. Агрессивно борясь с ненужной сложностью, Rails проявляет свои лучшие черты в ориентированных на человека аспектах разработки, которые и определяют успешность наших проектов. Программирование в Rails доставляет нам удовольствие – в этом и есть залог успеха!

Rails предоставляет исчерпывающие инструменты и технологическую инфраструктуру, поощряя нас к разработке того, что действительно ценно для бизнеса. Заложенный в Ruby *Принцип Наименьшего Удивления* воплощен в простом и элегантном дизайне Rails, а самое главное то, что исходные тексты Rails полностью открыты и бесплатны. Следовательно, если больше ничего не помогает, можно заглянуть в исходный код и найти ответ даже на самый сложный вопрос.

Дэвид как-то заметил, что он не очень рад тому, что Rails стал одной из основных платформ, так как конкурентное преимущество, которое было у первых адептов, теперь сойдет на нет. Ранние приверженцы были в основном индивидуалами и небольшими группами веб-дизайнеров и программистов, пришедших по большей части из мира PHP.

Rails и корпоративные приложения

Называйте меня идеалистом, но я верю, что даже разработчики корпоративных систем в больших консервативных компаниях стремились бы стать более эффективными и восприимчивыми к инновациям, будь у них соответствующие инструменты и стимулы. Мне кажется, что именно поэтому среди них с каждым годом становится все больше тех, кто запрыгивает в вагон Rails.

Быть может, разработчики корпоративных приложений станут наиболее громогласными и восторженными приверженцами Ruby и Rails, поскольку сейчас именно эта группа больше всего теряет от сложившегося положения вещей. Они становятся жертвами массовых увольнений и неправильно понимаемого аутсорсинга, основанного на предположениях о том, что «спецификация важнее реализации» и «реализация должна быть механическим и тривиальным делом».

Правда ли, что спецификация важнее реализации? Для большинства проектов это не так. Действительно ли реализация тривиальна для всех проектов, а не только совсем простеньких? Конечно, нет! Существуют фундаментальные сложности, свойственные разработке ПО, особенно масштаба предприятия[1]:

- трудные для понимания унаследованные системы;
- очень сложные предметные области, например банковские инвестиции;
- заинтересованные стороны и бизнес-аналитики, которые на самом деле не знают, что им нужно;
- менеджеры, противящиеся повышению производительности труда, так как это урезает их годовые бюджеты;
- конечные пользователи, активно саботирующие ваш проект;
- политика! Не высовывайся, а то укоротят ровно на голову.

Консультируя компании, входящие в список Fortune 1000, я ежедневно наблюдал такие ситуации в течение десяти лет и видел, как хоронили здоровые идеи. Но есть и альтернатива стремлению угодить и нашим и вашим, альтернатива настолько привлекательная, что она способна преодолеть политические соображения и гарантированно заслужить вам одобрение и распахнуть двери для новых возможностей.

Альтернатива состоит в том, чтобы быть незаурядным! Начинается все с продуктивности. Я хочу сказать, что вы должны быть в своем деле настолько эффективны, что никому в голову не придет делать из вас козла отпущения; даже сама попытка должна быть равнозначна политическому самоубийству. Я говорю о культивировании практики, при

[1] Не хочу сказать, что компаниям-стартапам проще, но их проблемы не так драматичны.

которой ваши результаты настолько блестящи, что вызывают слезы радости даже у самых циничных и жестокосердых заказчиков проекта. Я намекаю на то, что у вас остается время на непрекращающееся доведение своих приложений до состояния совершенства, привлекающего на вашу сторону восторженных конечных пользователей.

Будьте незаурядны, и у вас появятся счастливые клиенты, вовремя платящие по счетам, а вы не подпадете под ежегодное сокращение штатов, поскольку тот, кто принимает решения, скажет: «Нет, такими людьми мы разбрасываться не можем».

Одну секундочку. Не стану порицать вас, если вы отнесетесь к моим словам скептически, но это вовсе не пустая болтовня. Я описываю собственную жизнь после перехода на платформу Ruby on Rails. Цель этой книги – помочь вам сделать Ruby on Rails своим секретным (или не таким уж секретным) оружием в борьбе за выживание в предательском мире разработки ПО.

Получение результатов

Мы хотели на собственном опыте и знании индустрии показать вам, как получать практические результаты от своих проектов на платформе Ruby on Rails, вооружить вас доводами в пользу выбора этой технологии и аргументами в спорах, которые неизбежно будут возникать. Понимая, что серебряных пуль не бывает, мы хотим также рассказать о ситуациях, когда выбор Rails был бы ошибкой.

Следуя выбранному пути, мы глубоко проанализируем все компоненты Rails и обсудим, как их можно расширить, когда такая необходимость возникнет. Ruby – исключительно гибкий язык, а значит, существуют мириады способов самостоятельно настроить поведение Rails. Вы поймете, что путь Ruby заключается в том, чтобы предоставить вам свободу в выборе оптимального решения поставленной задачи.

Эту книгу можно использовать и как справочник – вы найдете в ней руководство по Rails API, описание богатого набора идиом Ruby, подходов к проектированию, библиотек и подключаемых модулей, полезных разработчику корпоративных приложений на платформе Ruby on Rails.

О пристрастном ПО

Прежде чем продолжить, скажу, что среда Rails получилась столь незаурядной в частности потому, что это «пристрастное» ПО, написанное пристрастными программистами. И эта книга, и ее авторы тоже пристрастны.

Ниже приведен перечень некоторых пристрастных мнений о разработке, нашедших отражение на страницах этой книги. Вы можете с чем-то не соглашаться, просто имейте в виду:

- мотивация и продуктивность разработчика для успешности проекта важнее всего остального;
- лучший способ обеспечить мотивацию и продуктивность – сосредоточиться на получении результата, повышающего ценность бизнеса;
- производительность означает «максимально быстрое исполнение при наличии имеющегося набора ресурсов»;
- масштабируемость означает «максимально быстрое исполнение при наличии необходимого набора ресурсов»;
- производительность не имеет значения, если вы не можете масштабировать систему;
- если масштабируемость обходится дешево, то стремление выжать из процессоров всю производительность до последней капли ни в коем случае не должно стоять на первом месте;
- связывание масштабируемости с выбором инструментов разработки – это распространенная в индустрии ошибка; к большинству программных систем требование экстремальной масштабируемости не предъявляется;
- для производительности небезразличен выбор языка и инструментов, поскольку чем выше уровень языка, тем проще писать и понимать написанные на нем программы. Все согласны, что в большинстве приложений проблемы с производительностью вызваны плохо написанным кодом;
- примат соглашения над конфигурацией – наилучший принцип написания ПО. Гигантские конфигурационные XML-файлы должны быть изжиты;
- переносимость кода, то есть возможность запустить имеющуюся программу на другой платформе без изменений, не так уж важна;
- лучше решить задачу хорошо, даже если это решение будет работать только на одной платформе;
- переносимости грош цена, если проект проваливается;
- переносимость относительно базы данных, то есть работоспособность одной и той же программы для разных СУБД, редко бывает важна и почти никогда не достигается;
- внешний вид важен даже для небольших проектов. Если ваше приложение плохо выглядит, все будут считать, что оно и написано плохо;
- как правило, технология не должна диктовать подход к решению задачи; однако этот совет не надо воспринимать как оправдание для продолжения работы с негодной технологией;
- достоинства использования в приложении обобщенных компонентов сомнительны. Конкретные проекты обычно создаются для решения весьма специфичных задач бизнеса, и требования к инфраструктуре резко отличаются, поэтому на практике добиться параметризованного повторного использования очень трудно.

Немало получилось пристрастных мнений. Но не пугайтесь, наша книга – это прежде всего справочник, и больше такого рода перечней вы не встретите.

Об этой книге

Эта книга не учебник и не введение в язык Ruby или платформу Rails. Она задумана как справочник, который профессиональный разработчик на платформе Rails будет использовать в повседневной работе. Иногда мы залезаем внутрь Rails и приводим фрагменты кода, чтобы показать, почему Rails ведет себя так, а не иначе. Уверенный в себе читатель, возможно, сумеет изучить Rails, пользуясь только этой книгой, обширными сетевыми ресурсами и собственным умом, но существуют другие публикации, более подходящие для начинающих.

Я занимаюсь разработкой на платформе Rails на постоянной основе, как и все прочие авторы, участвовавшие в работе над этой книгой. Мы не тратим все время на написание книг или обучение других, хотя с удовольствием занимаемся этим для дополнительного заработка.

Я начал писать эту книгу для себя, потому что терпеть не могу пользоваться онлайновой документацией, особенно справкой по API, к которой приходится обращаться снова и снова. Поскольку лицензия на использование документации по API весьма либеральна (как и все в Rails), в нескольких разделах книги я привожу цитаты из нее. Но практически во всех случаях оригинальная документация дополнена или исправлена, снабжена дополнительными примерами и комментариями, основанными на практическом опыте.

Надеюсь, что вы, как и я, любите, чтобы книга лежала рядом с клавиатурой, на полях можно было писать свои заметки, оставлять закладки и загибать уголки. Когда я пишу программу, мне нужно, чтобы под рукой всегда была и документация по API, и подробные объяснения, и подходящие примеры.

Организация книги

Я стремился организовать материал естественным образом, но при этом сделать книгу самым лучшим справочником по Rails. Поэтому я уделял много внимания целостности описания всех подсистем Rails, приводя там, где необходимо, детальную информацию по API. По охвату материала главы несколько различаются, и у меня есть подозрение, что в одной книге уже невозможно охватить Rails целиком.

Поверьте, было очень непросто выбрать такую структуру книги, которая удовлетворила бы всех и каждого. В частности, некоторые читатели недоумевали, почему подсистема `ActiveRecord` не рассматривается в самом начале. Rails – это прежде всего платформа для разработки веб-приложений, и, на мой взгляд, реализация контроллеров и марш-

рутизации – наиболее уникальная, мощная и эффективная ее особенность, а ActiveRecord стоит на втором месте, хотя и совсем близко.

Таким образом, последовательность изложения материала выглядит следующим образом:

- среда Rails, инициализация, конфигурирование и протоколирование;
- диспетчер Rails, контроллеры, рендеринг и маршрутизация;
- архитектурный стиль REST, ресурсы и Rails;
- основы ActiveRecord, ассоциации, контроль и продвинутые приемы;
- система шаблонов ActionView, кэширование и помощники;
- Ajax, библиотеки Prototype и Scriptaculous на языке JavaScript и RJS;
- управление сеансами, регистрация и аутентификация;
- XML и ActiveResource;
- фоновая обработка и ActionMailer;
- тестирование и спецификации (включая описание RSpec on Rails и Selenium);
- установка, администрирование и написание собственных подключаемых модулей;
- развертывание Rails в промышленной среде, конфигурирование и Capistrano.

Примеры кода и листинги

Предметные области в примерах кода должны быть знакомы большинству профессиональных разработчиков. Это приложения для отслеживания времени и затрат, управления региональными данными и ведения блогов. Я не буду на многих страницах объяснять тонкие нюансы бизнес-логики или обосновывать выбор проектных решений, не имеющих прямого отношения к рассматриваемой теме. Следуя стилю Хэла Фултона, автора книги *The Ruby Way*[1], вышедшей в этой же серии, большинство примеров не являются законченными программами – приводится только относящийся к делу код. Части кода, опущенные для краткости, обозначаются многоточием.

Если фрагмент кода длинный и значимый, и я считаю, что у вас может возникнуть желание вставить его в собственную программу буквально, он снабжается отдельным заголовком «Листинг». Таких листингов в книге не так уж много. Собранные вместе листинги не составляют

[1] Хэл Фултон «Программирование на языке Ruby», ДМК-Пресс, 2007. – *Прим. перев.*

готовую работающую систему. Листинги призваны лишь послужить источником вдохновения для создания промышленных программ, но не забывайте, что в них зачастую отсутствуют детали, необходимые в реальном приложении. Например, в примерах контроллеров обычно нет разбиения на страницы и логики контроля доступа, поскольку это отвлекало бы от основной темы.

Подключаемые модули

Всякий раз, как вам приходится писать инфраструктурный код, совершенно не относящийся к предметной области приложения, вы, вероятно, занимаетесь лишней работой. Надеюсь, что при возникновении такого ощущения у вас будет под рукой эта книга. Почти всегда имеется какой-то новый метод в Rails API или сторонний подключаемый модуль, который выполняет необходимую вам работу.

На самом деле, одной из отличительных черт этой книги является то, что я, не колеблясь, ссылаюсь на имеющиеся сторонние модули и даже документирую образцы, которые, на мой взгляд, особенно важны для эффективной работы в Rails. Если подключаемый модуль лучше встроенной в Rails функциональности, мы вообще не рассматриваем последнюю (примером может служить разбиение на страницы).

Для среднего разработчика Rails может удвоить продуктивность, но я встречал серьезных программистов на платформе Rails, которым удавалось достигать куда более впечатляющих результатов. Связано это с тем, что мы с религиозной почтительностью относимся к принципу «Не повторяйся» (Don't Repeat Yourself – DRY), следствием которого является принцип «Не изобретай колесо» (Don't Reinvent The Wheel – DRTW). Повторная реализация того, для чего уже существует хорошая реализация, – ненужная трата времени, хотя иногда это очень соблазнительно, поскольку программирование на Ruby – наслаждение.

Ruby on Rails на самом деле является обширной экосистемой, состоящей из кода ядра, официальных и сторонних подключаемых модулей. Эта экосистема развивается с бешеной скоростью и предоставляет вам все технологии, необходимые для построения даже самых сложных веб-приложений масштаба предприятия. Моя цель – вооружить вас достаточными знаниями, чтобы вы не пытались раз за разом изобретать колесо.

Рекомендуемые печатные издания и онлайновые ресурсы

Читателю этой книги могут быть полезны некоторые источники, упомянутые в данном разделе.

У большинства программистов на языке Ruby всегда под рукой «Киркомотыга» – книга *Programming Ruby*, поскольку это хороший спра-

вочник по языку. Читателям, которые хотят глубоко разобраться во всех нюансах программирования на Ruby, рекомендую второе издание упомянутой выше книги Хэла Фултона *The Ruby Way, Second Edition*.

Я настоятельно рекомендую глубокие видеопрезентации по различным аспектам Rails *Peepcode Screencasts* несравненного Джеффри Грозенбаха (Geoffrey Grosenbach), выложенные на сайте *http://peepcode.com*.

О Дэвиде Хейнемейере Хэнссоне

Я имел удовольствие подружиться с Дэвидом Хейнемейером Хэнссоном[1], создателем Rails, в начале 2005 года, еще до того как Rails стал общепризнанной платформой, а он превратился в *всемирную суперзвезду Web 2.0*. Дружба с Дэвидом во многом объясняет то, почему я сегодня пишу эту книгу. Мнения и публичные высказывания Дэвида формируют мир Rails, поэтому его часто цитируют при обсуждении природы платформы Rails и способов ее эффективного применения.

Дэвид пару раз говорил мне, что ненавидит кличку DHH, которой люди предпочитают пользоваться вместо его длинного и труднопроизносимого имени. Поэтому в этой книге я всегда называю его Дэвид, а не DHH. Встретив имя Дэвид без каких бы то ни было пояснений, вы должны понимать, что я говорю о единственном и неповторимом Дэвиде Хейнемейере Хэнссоне.

Rails в общем и целом все еще остается небольшим сообществом, и в некоторых случаях я называю разработчиков ядра и знаменитостей по имени. Идеальным примером может служить удивительный человек, участник команды разработчиков ядра, Рик Олсон (Rick Olson), написавший столько полезных подключаемых модулей, что его имя встречается в самых разных местах книги.

Цели

Я уже говорил, что льщу себя надеждой сделать эту книгу вашим основным справочником по Ruby on Rails. Не думаю, что найдется много людей, которые прочтут ее от корки до корки, если только они не ставят себе целью расширить базовые знания о платформе Rails. Но как бы то ни было, надеюсь, что со временем эта книга вселит в вас (как разработчика приложений и программиста) бóльшую уверенность при принятии решений, касающихся проектирования и реализации повседневных задач. Потратив время на чтение книги, вы станете лучше понимать идеи, положенные в основу Rails, а вкупе с практическим опытом это сделает работу над реальными проектами в Rails более комфортной.

Если вы выступаете в роли архитектора или руководителя проекта, наша книга не для вас, однако она поможет вам чувствовать себя более

[1] Известен также как DHH.

уверенно при обсуждении плюсов и минусов перехода на платформу Ruby on Rails и способов расширения Rails для достижения целей, стоящих перед возглавляемым вами проектом.

Наконец, если вы менеджер, то вам будут особенно интересны оценки практических перспектив и рассмотрение методик тестирования и инструментальных средств. Я надеюсь, что вы поймете, почему разработчики сходят с ума от языка Ruby и платформы Rails.

Предварительные условия

Предполагается, что читатель обладает следующими знаниями:

- основы синтаксиса Ruby и такие языковые конструкции, как блоки;
- уверенное владение принципами объектно-ориентированной разработки и паттернами проектирования;
- базовые знания в области реляционных баз данных и языка SQL;
- знакомство со структурой и работой приложений для Rails;
- базовые знания о таких сетевых протоколах, как HTTP и SMTP;
- базовые знания о XML-документах и веб-службах;
- знакомство с транзакциями, в частности, понимание того, что такое ACID (атомарность, непротиворечивость, изолированность, долговечность).

В разделе «Организация книги» уже отмечалось, что книга не построена по принципу «от простого к сложному». Некоторые главы действительно начинаются с изложения базового, можно даже сказать, вводного материала, после чего рассматриваются более сложные темы. Конечно, имеются куски, которые опытный разработчик на платформе Rails пропустит. Но мне кажется, что в каждой главе читатель любого уровня найдет для себя что-то новое и интересное.

Необходимое технологическое обеспечение

Последняя модель Apple *MacBookPro* с 4 Гб памяти и операционной системой OS X 10.4. Шучу, конечно. Linux тоже вполне годится для разработки в Rails. Microsoft Windows? Скажу так – у вас может быть другое мнение. Обратите внимание, как изящно и дипломатично я выразил эту мысль. В этой книге мы *не будем* обсуждать разработку для Rails на платформах Microsoft[1]. Насколько мне известно, большинство профессионалов в области Rails занимаются разработкой и развертыванием на других платформах.

[1] По этому поводу см. блог Softies on Rails по адресу *http://softiesonrails.com*.

1

Среда и конфигурирование Rails

Rails вызывает такой интерес не в последнюю очередь потому, что я не пытался угодить людям, не испытывающим тех же проблем, что я сам. Отделение эксплуатационной среды от среды разработки составляло для меня серьезную проблему, и я решил ее наилучшим способом, который смог придумать.

Дэвид Хейнемейер Хэнссон

Приложения для Rails заранее конфигурируются в трех стандартных режимах: *разработка*, *тестирование* и *эксплуатация*. Каждый из них соответствует некоторой среде исполнения и ассоциирован с набором параметров, которые определяют, например, к какой базе данных подключаться и надо ли перезагружать составляющие приложение классы при каждом запросе. При желании совсем нетрудно создать собственную среду.

Текущая среда задается переменной окружения `RAILS_ENV`, которая содержит имя требуемого режима работы и соответствует файлу с параметрами среды в папке `config/environment`. Поскольку параметры среды управляют рядом фундаментальных аспектов поведения Rails, в частности загрузкой классов, то для понимания пути Rails вы должны хорошо представлять себе назначение параметров среды.

В этой главе мы познакомимся с тем, как Rails запускается и обрабатывает запросы, для чего рассмотрим сценарии `boot.rb` и `environments.rb`, а также параметры трех стандартных режимов. Кроме того, мы затронем тему определения собственной среды и поговорим о причинах, по которым это может понадобиться.

Запуск

Одна из первых задач любого процесса, обрабатывающего запросы к Rails (например, сервера Webrick), — загрузка файла `config/environment.rb`. Взгляните, например, на строку в начале файла `public/dispatch.rb`:

```
require File.dirname(__FILE__) + "/../config/environment"
```

Другим процессам, которым необходима *вся* среда Rails (например, консоли и вашим тестам) также требуется файл `config/environment.rb`, поэтому вы обнаружите предложение `require` в начале таких файлов, как `test/test_helper.rb`.

Параметры среды по умолчанию

Последовательно рассмотрим параметры в подразумеваемом по умолчанию файле `environment.rb`, который читается в процессе начальной загрузки приложения для Rails 1.2.

Переопределение режима

Первый параметр касается только тех, кто развертывает приложение в среде виртуального хостинга (shared hosting). Он закомментирован, так как устанавливать режим работы Rails в этом месте приходится редко:

```
# Раскомментируйте следующую строку, чтобы принудительно перевести Rails
# в режим эксплуатации, если вы не контролируете веб-сервер или сервер
# приложений и не можете настроить его правильно
# ENV['RAILS_ENV'] ||= 'production'
```

Предостережение. Если здесь вы присвоите переменной окружения `RAILS_ENV` или, если на то пошло, константной переменной `RAILS_ENV` значение `production`, то *все* в Rails будет работать в режиме эксплуатации. Например, ваш набор тестов будет функционировать неправильно, поскольку сценарий `test_helper.rb` устанавливает `RAILS_ENV` в `test` непосредственно перед загрузкой среды.

Версия Rails Gem

Напомним, что в этот момент среда Rails еще не загружена. На самом деле, сценарий должен найти и загрузить платформу до совершения других действий. Этот параметр говорит сценарию, какую версию Rails загружать:

```
# Задает используемую версию gem-пакета Rails, когда каталог
# vendor/rails отсутствует
RAILS_GEM_VERSION = '1.2.3'
```

Как видите, на момент написания этой главы последняя версия Rails (которую я на самом деле установил в виде gem-пакета) имела номер 1.2.3.

Этот параметр не принимается во внимание, если вы «сидите на острие» (running *edge*). Под этим в сообществе понимают, что вы запускаете замороженный образ Rails, хранящийся в подкаталоге `vendor/rails` вашего проекта. Чтобы скопировать последнюю версию Rails из репозитория в свой каталог `vendor/rails`, включите в проект команду `rake rails:freeze:edge`.

Начальная загрузка

Следующие строки в файле `environment.rb` — это место, где колеса начинают крутиться после загрузки файла `config/boot.rb`:

```
# Начальная загрузка среды Rails, платформа и конфигурации по умолчанию
require File.join(File.dirname(__FILE__), 'boot')
```

Обратите внимание, что этот сценарий загрузки генерируется как часть вашего Rails-приложения, но редактировать его не следует. Я вкратце опишу, что он делает, так как это может оказаться полезно для поиска причин ошибок при установке Rails.

Сначала сценарий начальной загрузки убеждается, что установлена переменная окружения `RAILS_ROOT`; она содержит путь к корню текущего проекта Rails. Переменная `RAILS_ROOT` повсеместно используется в коде Rails для поиска различных файлов, в том числе шаблонов представления. Если она не установлена, то в качестве значения задается путь к каталогу на один уровень выше текущего (напомню, что мы находимся в каталоге `config`).

«Острие Rails»

Есть такое выражение: «Если ты не сидишь на острие, то занимаешь слишком много места». Для большинства разработчиков на платформе Rails «сидеть на острие» означает пользоваться последней (и, хочется надеяться, самой лучшей) версией Ruby on Rails.

Разработчики ядра Rails выкладывают свои файлы в открытый репозиторий системы управления версиями Subversion. С давних пор тех, кто запускает свои приложения в официально не выпущенной версии Rails, чтобы воспользоваться новыми функциями и не сталкиваться с уже исправленными ошибками, принято называть *сидящими на острие Rails*. Для этого вам нужно лишь извлечь код Rails из хранилища в каталог `vendor/rails` своего приложения.

В октябре 2005 года в стандартный rakefile Rails были добавлены задания для автоматизации процесса *заморозки* и *разморозки* приложения, то есть привязки его к конкретной версии «острия»: `rails:freeze:edge` и `rails:unfreeze`.

Если вы еще не знакомы с этим феноменом, то может возникнуть вопрос, зачем кому-то в здравом уме необходимо разрабатывать приложение на основе заведомо нестабильной версии ПО. Для большинства из нас мотивом служит желание иметь в своем распоряжении самые свежие возможности, и, к счастью, такое решение оказывается не столь уж безумным. История убеждает, что основная ветвь Rails остается вполне стабильной. В 2005 году, когда осуществлялся переход к версии Rails 1.0, я разрабатывал, «сидя на острие», проект для большого корпоративного клиента. Мы осознавали риск, но в острие был ряд критически важных нововведений и исправлений ошибок, без которых мы не могли работать. На протяжении нескольких месяцев наша версия Rails обновлялась до двух раз в неделю, и был лишь один случай, когда неожиданная ошибка оказалась связана с проблемой в ядре Rails.

Своей стабильностью острие обязано прежде всего широкому тестовому покрытию Rails. Регрессионные тесты эффективно предотвращают проникновение ошибок в код ядра. Любая заплата, предлагаемая разработчикам ядра, должна включать адекватные автономные и функциональные тесты, иначе она даже не будет рассматриваться. Принципы гибкого программирования в Rails – это не просто общий курс, а религиозный догмат.

Но после выхода Rails 1.2 работа на острие перестала быть насущной необходимостью, поэтому разработчики ядра настоятельно отговаривают от такой практики. Официально выпущенные версии достаточно хороши для большинства разработчиков. И все же иногда острие Rails бывает полезно. Например, это позволило нам обсудить в данной книге многие особенности Rails 2.0 еще до выхода официальной версии 2.0.

На платформах, отличных от Windows, используется стандартная библиотека Ruby `pathname`, чтобы удалить из пути соседние символы косой черты и бесполезные точки:

```
unless RUBY_PLATFORM =~ /mswin32/
  require 'pathname'
  root_path = Pathname.new(root_path).cleanpath(true).to_s
end
```

Теперь сценарий загрузки должен определить, какую версию Rails использовать. Первым делом он проверяет, есть ли в вашем проекте подкаталог `vendor/rails`. Это означало бы, что вы «сидите на острие»:

```
File.directory?("#{RAILS_ROOT}/vendor/rails")
```

Это самый простой случай. Если же вы не «сидите на острие», а я подозреваю, что большинство разработчиков Rails не сидят, то сценарию придется еще поработать и найти пакет *RubyGem*, содержащий Rails.

Пакеты RubyGem

Прежде всего отметим, что сценарий начальной загрузки затребует (с помощью `require`) библиотеку `rubygems`. Скорее всего, вы знаете, что такое система RubyGems, поскольку устанавливали с ее помощью Rails, выполнив команду `gem install rails`.

По причинам, которые выходят за рамки настоящего обсуждения, Rails иногда загружает *только сценарий начальной загрузки*. Поэтому далее этот сценарий считывает файл `config/environment.rb` как текст (пропуская закомментированные строки) и с помощью модуля `regexp` ищет и разбирает строку `RAILS_GEM_VERSION`.

Определив, какую версию Rails загружать, сценарий затребует gem-пакет Rails. Если в этот момент окажется, что в файле `environment.rb` указана версия Rails, отсутствующая на вашей рабочей станции, то при попытке запустить сервер или консоль Rails будет выдано примерно такое сообщение об ошибке:

```
Cannot find gem for Rails =1.1.5:
Install the missing gem with 'gem install -v=1.1.5 rails', or
change environment.rb to define RAILS_GEM_VERSION with your
desired version.

Не могу найти gem для Rails =1.1.5:
Установите отсутствующий gem командой 'gem install -v=1.1.5 rails' или
измените файл environment.rb, указав в параметре RAILS_GEM_VERSION нужную
вам версию.
```

Инициализатор

Далее сценарий начальной загрузки затребует сценарий `initializer.rb` (инициализатор), который отвечает за конфигурирование и позже будет использован сценарием `environment`.

И наконец, сценарий начальной загрузки передает инициализатору подразумеваемые по умолчанию пути загрузки (иными словами, строит `classpath`). Путь загрузки собирается из списка компонентных сред, составляющих саму Rails, и папок вашего приложения, в которых хранится код. Напомню, что в Ruby путь загрузки определяет, где метод `require` должен искать вызываемый код. До этой точки путь загрузки содержал лишь текущий рабочий каталог.

Подразумеваемые пути загрузки

Код, с помощью которого Rails задает пути загрузки, легко читается и хорошо прокомментирован, поэтому я просто процитирую его без пояснений (листинг 1.1).

Листинг 1.1. Некоторые методы в файле railties/lib/initializer.rb

```
def default_frameworks
  [ :active_record, :action_controller, :action_view,
    :action_mailer, :action_web_service ]
end

def default_load_paths
  paths = ["#{root_path}/test/mocks/#{environment}"]

  # Добавить каталог контроллера приложения
  paths.concat(Dir["#{root_path}/app/controllers/"])

  # Затем подкаталоги компонентов.
  paths.concat(Dir["#{root_path}/components/[_a-z]*"])

  # Затем стандартные каталоги для включаемых файлов.
  paths.concat %w(
    app
    app/models
    app/controllers
    app/helpers
    app/services
    app/apis
    components
    config
    lib
    vendor
  ).map { |dir| "#{root_path}/#{dir}" }.select { |dir|
  File.directory?(dir) }

  paths.concat Dir["#{root_path}/vendor/plugins/*/lib/"]
  paths.concat builtin_directories
end

def builtin_directories
  # Каталоги builtin включаются только для среды разработки.
  (environment == 'development') ?
  Dir["#{RAILTIES_PATH}/builtin/*/"] : []
end
```

Rails, модули и код автозагрузки

Обычно в Ruby для включения в приложение кода из другого файла вы употребляете предложение require. Но в Rails это стандартное поведение расширено за счет простого соглашения, позволяющего в большинстве случаев загружать ваш код автоматически.

Если вам доводилось пользоваться консолью Rails, то вы уже знаете, как это соглашение проявляется: никогда ничего не приходится загружать явно!

Вот как это работает. Встречая в вашем коде класс или модуль, который еще не был определен, Rails применяет следующие правила для определения файлов, которые нужно затребовать, чтобы загрузить этот класс или модуль:

- если класс или модуль не является вложенным, вставить подчерк между именами констант и затребовать файл с получившимся именем, например:
  - EstimationCalculator преобразуется в require 'estimation_calculator';
  - KittTurboBoost преобразуется в require 'kitt_turbo_boost';
- если класс или модуль является вложенным, Rails вставляет подчерк между именами констант в каждом объемлющем модуле и требует файл из соответствующего подкаталога, например:
  - MacGyver::SwissArmyKnife преобразуется в require 'mac_gyver/swiss_army_knife';
- `Some::ReallyRatherDeeply::NestedClass` преобразуется в `'some/ really_rather_deeply/nested_class'` и, если этот класс еще не загружен, Rails ожидает найти его в файле `nested_class.rb`, который находится в подкаталоге `really_rather_deeply` каталога some, расположенного где-то в пути загрузки Ruby (например, в одном из подкаталогов `app` или в каталоге `lib` подключаемого модуля).

Таким образом, вам редко придется явно загружать Ruby-код в приложение для Rails (с помощью `require`), если вы будете придерживаться соглашений об именовании.

Встройка Rails Info

Зачем метод `default_load_paths` в листинге 1.1 вызывает метод `builtin_directories`? Что вообще такое «каталоги встроек»? Это то место, в котором Rails ищет поведение приложения (то есть модели, помощников и контроллеры). Можете называть это механизмом подключения дополнительных модулей, который предлагается платформой.

В настоящее время существует единственная «встройка» (builtin) `railties/builtin/rails_info`. Известна она, главным образом, как цель, на которую ведет ссылка «About your application's environment» (О среде вашего приложения), отображаемая на странице `index.html` («Welcome Aboard») в любом новом приложении Rails.

Чтобы проверить, как она работает, запустите любое приложение Rails в режиме разработки и укажите в броузере адрес *http://localhost:3000/rails/info/properties*. Вы увидите на экране диагностическую информацию, в том числе номера версий и параметры:

```
Ruby version                    1.8.5 (i686-darwin8.8.1)
RubyGems version                0.9.0
```

```
Rails version                   1.2.0
Active Record version           1.14.4
Action Pack version             1.12.5
Action Web Service version      1.1.6
Action Mailer version           1.2.5
Active Support version          1.3.1
Edge Rails revision             33
Application root                /Users/obie/prorails/time_and_expenses
Environment                     development
Database adapter                mysql
Database schema version         8
```

Конфигурирование

Вернемся к сценарию `environment`, где нас ожидают параметры, определяемые разработчиком. Следующие две строки выглядят так:

```
Rails::Initializer.run do |config|
  # Параметры в файлах config/environments/* более приоритетны, чем
  # заданные здесь
```

Комментарий напоминает, что параметры, заданные в файлах описания среды для конкретного режима, имеют больший приоритет, чем параметры в файле `environment.rb`. Связано это с тем, что такие файлы загружаются позже, поэтому ранее загруженные значения одноименных параметров перезаписываются.

Пропуск частей среды

Первый параметр позволяет *не* загружать ненужные вам части Rails (Ruby – интерпретируемый язык, поэтому если есть возможность уменьшить объем кода, анализируемого интерпретатором, ею надо воспользоваться просто из соображений производительности). Во многих приложениях для Rails не используются ни электронная почта, ни веб-службы, поэтому мы и взяли их в качестве примеров:

```
# Пропустить среды, которые вы не собираетесь использовать (работает только #
совместно с vendor/rails)
config.frameworks -= [ :action_web_service, :action_mailer ]
```

Дополнительные пути загрузки

В тех редких случаях, когда в состав умалчиваемых путей загрузки необходимо добавить дополнительные, можете воспользоваться следующим параметром:

```
# Включить в состав путей загрузки свои каталоги
config.load_paths += %W( #{RAILS_ROOT}/extras )
```

Если вы не знаете, скажу, что `%W` преобразует список разделенных пробелами значений в массив-литерал и довольно часто используется в коде

Rails для удобства (признаюсь, что поначалу меня, привыкшего к синтаксису Java, это сбивало с толку).

По-моему, механизм дополнительных путей загрузки на самом деле не нужен, так как Rails обычно расширяется с помощью подключаемых модулей, а для них существует отдельное соглашение о путях загрузки. В главе 19 «Расширение Rails с помощью подключаемых модулей» эта тема рассматривается более подробно, а книга Джеймса Адама (James Adam) *Rails Plugins: Extending Rails Beyond the Core*, также вышедшая в серии Addison-Wesley Professional Ruby Series, – исчерпывающее справочное руководство по написанию подключаемых модулей.

Отметим, что два последних параметра не являются параметрами в традиционном смысле этого слова, то есть результатом присваивания значения некоторому свойству. При работе с объектом конфигурации Rails в полной мере использует предоставляемую Ruby возможность добавлять и удалять элементы из массива.

Переопределение уровня протоколирования

По умолчанию принимается уровень протоколирования `:debug`, но это можно переопределить.

```
# Использовать во всех средах один и тот же уровень протоколирования
# (по умолчанию в режиме эксплуатации используется :info, в остальных :debug)
config.log_level = :debug
```

Ниже в этой главе мы детально рассмотрим механизм протоколирования в Rails.

Хранилище сеансов ActiveRecord

Если вы хотите сохранять сеансы пользователей в базе данных (а в режиме эксплуатации так оно, как правило, и бывает), воспользуйтесь параметром, который позволяет определить соответствующие настройки:

```
# Хранить сеансы в базе данных, а не в файловой системе
# (создайте таблицу сеансов командой 'rake db:sessions:create')
config.action_controller.session_store = :active_record_store
```

Дублирование схемы

При каждом запуске тестов Rails сохраняет схему базы данных, используемой в режиме разработки, в базе данных для тестирования, вызывая автоматически сгенерированный сценарий `schema.rb`. Он очень похож на сценарий миграции `ActiveRecord` и в действительности основан на том же самом API.

Если вы выполняете действия, не совместимые с кодом дубликатора схемы (см. комментарий ниже), возникает необходимость вернуться к старому способу сохранения схемы с помощью SQL:

```
# Использовать SQL вместо дубликатора схемы на основе Active Record при
# создании тестовой базы. Это необходимо, если дубликатор не может
# сбросить всю схему, например, потому что в ней определены ограничения
# или специфичные для конкретной СУБД типы столбцов.
config.active_record.schema_format = :sql
```

Наблюдатели

Наблюдатели (observer) `ActiveRecord` — это полноценные объекты в приложениях Rails, решающие такие задачи, как очистка кэшей и управление денормализованными данными. В стандартном файле `environment.rb` приведены примеры классов, которые могут выступать в вашем приложении в роли наблюдателей (в Rails на самом деле нет наблюдателей `cacher` и `garbage_collector`, но это вовсе не означает, что Ruby не выполняет уборку мусора).

```
# Активировать наблюдателей, которые должны работать постоянно
# config.active_record.observers = :cacher, :garbage_collector
```

Мы будем детально рассматривать наблюдателей `ActiveRecord` в главе 9 «Дополнительные вопросы `ActiveRecord`».

Часовые пояса

По умолчанию в Rails используется локальное время (с помощью модуля `Time.local`), то есть тот же часовой пояс, что и на сервере. Чтобы указать Rails другой часовой пояс, следует модифицировать переменную окружения `TZ`.

Список часовых поясов обычно находится в папке `/usr/share/zoneinfo`. На платформе Mac перечень допустимых имен поясов хранится в файле `/usr/share/zoneinfo/zone.tab`. Не забывайте, что это решение (а в особенности конкретные значения) зависит от операционной системы, а сама операционная система, возможно, уже установила подходящее значение часового пояса от вашего имени.

Задавать значение `TZ` можно в любом месте программы, но если вы хотите изменить его для всего веб-приложения, лучше поместить соответствующий код в файл `environment.rb` рядом с другими настройками часового пояса.

```
ENV['TZ'] = 'US/Eastern'
```

А что если вы хотите поддерживать разные часовые пояса для разных пользователей? Прежде всего следует сказать библиотеке `ActiveRecord`, чтобы она хранила время в базе данных в виде UTC (воспользовавшись модулем `Time.utc`).

```
# Потребовать, чтобы Active Record использовала время UTC, а не локальное
config.active_record.default_timezone = :utc
```

Затем понадобится способ переключения между часовыми поясами в зависимости от конкретного пользователя. К сожалению, входящий

в состав Rails стандартный класс TimeZone не умеет обрабатывать переход на летнее время, что, откровенно говоря, делает его совершенно бесполезным.

Существует gem-пакет TZInfo[1], написанный на чистом Ruby, в который входит класс TimeZone, корректно работающий с летним временем. Его можно подставить вместо одноименного класса в Rails. Чтобы воспользоваться им в своем приложении, установите пакет TZInfo и подключаемый модуль tzinfo_timezone. Однако, поскольку это решение написано целиком на Ruby, оно работает медленно (хотя не исключено, что для вашей задачи его быстродействия хватит).

Но постойте – разве класс Time, входящий в стандартный дистрибутив Ruby, не умеет корректно работать с летним временем?

```
>> Time.now
=> Mon Nov 27 16:32:51 -0500 2006
>> Time.now.getgm
=> Mon Nov 27 21:32:56 UTC 2006
```

На консоль все выводится правильно. Значит, написать собственный метод преобразования будет несложно, правда?

В сообщении, отправленном в список рассылки rails-mailing-list в августе 2006 года[2], Гжегош Данилюк (Grzegorz Daniluk) привел пример, показывающий, как это сделать (и продемонстрировал, что он работает от 7 до 9 раз быстрее TZInfo). Добавьте следующий код в любой модуль-помощник своего приложения или оформите его в виде отдельного класса в папке lib:

```
# Чтобы преобразовать полученное дату и время в UTC и сохранить в БД
def user2utc(t)
  ENV["TZ"] = current_user.time_zone_name
  res = Time.local(t.year, t.month, t.day, t.hour, t.min, t.sec).utc
  ENV["TZ"] = "UTC"
  res
end

# Чтобы отобразить дату и время
def utc2user(t)
  ENV["TZ"] = current_user.time_zone_name
  res = t.getlocal
  ENV["TZ"] = "UTC"
  res
end
```

Филип Росс (Philip Ross), автор TZInfo, в том же списке рассылки поместил подробный ответ, показывающий, что это решение не работает для пользователей на платформе Windows[3]. Он также привел комментарии

[1] *http://tzinfo.rubyforge.org/*

[2] *www.ruby-forum.com/topic/79431*

[3] *http://article.gmane.org/gmane.comp.lang.ruby.rails/75790*

по поводу обработки некорректно заданного времени: «Еще один аспект, в котором TZInfo лучше, чем использование переменной окружения TZ, связан с обработкой некорректно и неоднозначно заданного локального времени (например, при переходе на летнее время и обратно). Time.local всегда возвращает время, пусть даже оно некорректно или неоднозначно. TZInfo сообщает о некорректно заданном времени и позволяет разрешить неоднозначность, указав, следует ли использовать летнее или обычное время, или выполнив блок, в котором производится выбор».

Короче говоря, не пользуйтесь Windows. Шучу, шучу. Из всего это нужно извлечь урок: корректная обработка времени – не такое простое дело, и подходить к решению этой задачи следует очень аккуратно.

Дополнительные конфигурационные параметры

Мы рассмотрели все конфигурационные параметры, для которых в стандартном файле environment.rb имеются примеры. Существуют и другие параметры, но я подозреваю, что вы о них не знаете, и вряд ли они когда-нибудь понадобятся. Если хотите ознакомиться со всем списком, загляните в исходный текст или в документацию по классу Configuration, которая начинается примерно со строки 400 файла railties/lib/initializer.rb.

Помните, мы говорили, что переменная окружения RAILS_ENV, определяет, какие параметры среды загружать дальше? Теперь самое время рассмотреть параметры, принимаемые по умолчанию для каждого из стандартных режимов Rails.

Режим разработки

Режим разработки принимается в Rails по умолчанию, именно в нем вы будете проводить бóльшую часть времени:

```
# Определенные здесь параметры имеют больший приоритет, чем параметры
# в файле config/environment.rb

# В режиме разработки код приложения перезагружается при каждом запросе.
# Это увеличивает время реакции, но идеально подходит для разработки,
# так как вам не приходится перезагружать веб-сервер после внесения каждого
# изменения в код.
config.cache_classes = false

# Записывать в протокол сообщения об ошибках при случайном вызове метода
# для объекта nil.
config.whiny_nils = true

# Активировать сервер точек останова, с которыми соединяется
# script/breakpointer
config.breakpoint_server = true
```

```
# Показывать полные отчеты об ошибках и запретить кэширование
config.action_controller.consider_all_requests_local = true
config.action_controller.perform_caching             = false
config.action_view.cache_template_extensions         = false
config.action_view.debug_rjs                         = true

# Не обращать внимание, если почтовый клиент не может отправить сообщение
config.action_mailer.raise_delivery_errors = false
```

В следующих разделах я подробно рассмотрю наиболее важные параметры, начав с кэширования классов; этот параметр отвечает за динамическую перезагрузку классов в Rails.

Динамическая перезагрузка классов

Одна из отличительных особенностей Rails – скорость цикла внесения изменений в режиме разработки. Правите код, щелкаете по кнопке Обновить в броузере – и все! Изменения волшебным образом отражаются на приложении. Такое поведение управляется параметром `config.cache_classes`, который, как видите, установлен в `false` в самом начале сценария `config/environments/development.rb`.

Не вдаваясь в технические детали, скажу, что если параметр `config.cache_classes` равен `true`, то Rails загружает классы с помощью предложения `require`, а если `false` – то с помощью предложения `load`.

Когда вы затребуете файл с кодом на Ruby с помощью `require`, интерпретатор исполнит и кэширует его. Если файл затребуется снова (при последующих запросах), интерпретатор пропустит предложение `require` и пойдет дальше. Если же файл загрузится предложением `load`, то интерпретатор считает и разберет его снова вне зависимости от того, сколько раз файл загружался раньше.

Теперь рассмотрим загрузку классов в Rails более пристально, поскольку иногда вам не удается заставить код перезагружаться автоматически, и это может довести до белого каления, *если не* понимаешь, как на самом деле работает механизм загрузки классов!

Загрузчик классов в Rails

В старом добром Ruby нет никаких соглашений об именовании файла в соответствии с содержимым. Однако в Rails, как легко заметить, почти всегда имеется прямая связь между именем Ruby-файла и содержащимся в нем классом. В Rails используется тот факт, что Ruby предоставляет механизм обратных вызовов для отсутствующих констант. Когда Rails встречает в коде неопределенную константу, он вызывает подпрограмму загрузки класса, которая, полагаясь на соглашения об именовании файлов, пытается найти и затребовать нужный сценарий.

Откуда загрузчик классов знает, где искать файл? Мы уже затрагивали этот вопрос выше при обсуждении роли сценария `initializer.rb` в процессе запуска Rails. В Rails имеется концепция путей загрузки, и по умолчанию множество путей включает практически все каталоги, куда вам может прийти в голову поместить код приложения.

Метод `default_load_paths` определяет, в каком порядке Rails просматривает каталоги в пути загрузки. Мы разберем код этого метода и объясним назначение каждой части пути загрузки.

Каталог `test/mocks` (подробно рассматривается в главе 17 «Тестирование») дает возможность переопределить поведение стандартных классов Rails:

```
paths = ["#{root_path}/test/mocks/#{environment}"]

# Добавить каталог контроллера приложения.
paths.concat(Dir["#{root_path}/app/controllers/"])

# Затем подкаталоги компонентов.
paths.concat(Dir["#{root_path}/components/[_a-z]*"])

# Затем стандартные каталоги для включаемых файлов.
paths.concat %w(
  app
  app/models
  app/controllers
  app/helpers
  app/services
  app/apis
  components
  config
  lib
  vendor
).map { |dir| "#{root_path}/#{dir}" }.select { |dir|
  File.directory?(dir) }

  paths.concat Dir["#{root_path}/vendor/plugins/*/lib/"]
  paths.concat builtin_directories
end
```

Хотите посмотреть содержимое пути загрузки для своего проекта? Запустите консоль и распечатайте переменную `$:`. Вот так:

```
$ console
Loading development environment.
>> $:
=> ["/usr/local/lib/ruby/gems/1.8/gems/ ... # выводятся примерно 20 строк
```

Для экономии места я опустил часть выведенного на консоль текста. В пути загрузки типичного проекта Rails обычно бывает 30 и более каталогов. Убедитесь сами.

Режим тестирования

Если Rails запускается в режиме тестирования (то есть значение переменной окружения `RAILS_ENV` равно `test`), то действуют следующие параметры:

```
# Определенные здесь параметры имеют больший приоритет, чем параметры
# в файле config/environment.rb

# Среда тестирования служит исключительно для прогона набора тестов
# вашего приложения. Ни для чего другого она не предназначена. Помните,
# что тестовая база данных – это "рабочая область", она уничтожается
# и заново создается при каждом прогоне. Не полагайтесь на хранящиеся
# в ней данные!
config.cache_classes = true

# Записывать в протокол сообщение об ошибке при случайном вызове метода
# для объекта nil.
config.whiny_nils = true

# Показывать полные отчеты об ошибках и запретить кэширование
config.action_controller.consider_all_requests_local = true
config.action_controller.perform_caching = false

# Не разрешать объекту ActionMailer отправлять почтовые сообщения
# реальным адресатам. Метод доставки :test сохраняет отправленные
# сообщения в массиве ActionMailer::Base.deliveries.
config.action_mailer.delivery_method = :test
```

Как правило, вообще не возникает необходимости модифицировать параметры среды тестирования.

Режим эксплуатации

Режим эксплуатации предназначен для запуска приложений Rails, развернутых в среде хостинга для обслуживания пользовательских запросов. Между режимом эксплуатации и другими режимами есть ряд существенных отличий, и на одном из первых мест стоит повышение быстродействия, поскольку классы приложения не перезагружаются при каждом запросе.

```
# Определенные здесь параметры имеют больший приоритет, чем параметры
# в файле config/environment.rb
# Среда эксплуатации предназначена для готовых, "живых" приложений.
# Код не перезагружается при каждом запросе.
config.cache_classes = true

# Для распределенной среды использовать другой протокол.
# config.logger = SyslogLogger.new
```

```
# Полные отчеты об ошибках запрещены, кэширование включено.
config.action_controller.consider_all_requests_local = false
config.action_controller.perform_caching = true

# Разрешить отправку изображений, таблиц стилей и javascript-сценариев
# с сервера статического контента.
# config.action_controller.asset_host = "http://assets.example.com"

# Отключить ошибки доставки почты, недопустимые электронные адреса
# игнорируются.
# config.action_mailer.raise_delivery_errors = false
```

Нестандартные среды

При необходимости для приложения Rails можно создать нестандартную среду исполнения, скопировав и изменив один из существующих файлов в каталоге `config/environments`. Чаще всего это делают ради формирования дополнительных вариантов среды эксплуатации, например, для постадийной работы (staging) или контроля качества.

У вас есть доступ к промышленной базе данных с рабочей станции, на которой ведется разработка? Тогда имеет смысл организовать *смешанную* (triage) среду. Для этого клонируйте обычные параметры режима разработки, но в описании соединения с базой данных укажите на промышленный сервер. Такая комбинация может оказаться полезной для быстрой диагностики ошибок в процессе промышленной эксплуатации.

Протоколирование

В большинстве программных контекстов в Rails (моделях, контроллерах, шаблонах представлениях) присутствует атрибут `logger`, в котором хранится ссылка на объект протоколирования, согласованный с интерфейсом `Log4r` или с применяемым по умолчанию в Ruby 1.8+ классом `Logger`. Чтобы получить ссылку на объект `logger` из любого места программы, воспользуйтесь константой `RAILS_DEFAULT_LOGGER`. Для нее даже есть специальная комбинация клавиш в редакторе TextMate (rdb →).

В Ruby совсем нетрудно создать новый объект `Logger`:

```
$ irb
> require 'logger'
=> true

irb(main):002:0> logger = Logger.new STDOUT
=> #<Logger:0x32db4c @level=0, @progname=nil, @logdev=
#<Logger::LogDevice:0x32d9bc ... >
```

```
> logger.warn "do not want!!!"
W, [2007-06-06T17:25:35.666927 #7303] WARN -- : do not want!!!
=> true

> logger.info "in your logger, giving info"
I, [2007-06-06T17:25:50.787598 #7303] INFO -- : in your logger, giving
your info
=> true
```

Обычно сообщение в протокол добавляется путем вызова того или иного метода объекта `logger`, в зависимости от серьезности ситуации. Определены следующие стандартные уровни серьезности (в порядке возрастания):

- **debug** – указывайте этот уровень для вывода данных, полезных в будущем для отладки. В режиме эксплуатации сообщения такого уровня обычно не пишутся;
- **info** – этот уровень служит для вывода информационных сообщений. Я обычно использую его, чтобы запротоколировать, снабдив временными штампами, необычные события, которые все же укладываются в рамки корректного поведения приложения;
- **warn** – данный уровень служит для вывода информации о необычных ситуациях, которые имеет смысл расследовать подробнее. Иногда я вывожу в протокол предупреждающие сообщения, когда в программе срабатывает сторожевой код, препятствующий клиенту выполнить недопустимое действие. Цель при этом – уведомить лицо, ответственное за сопровождение, о злонамеренном пользователе или об ошибке в пользовательском интерфейсе, например:

```
def create
  begin
    @group.add_member(current_user)
    flash[:notice] = "Вы успешно присоединились к #{@scene.display_name}"
  rescue ActiveRecord::RecordInvalid
    flash[:error] = "Вы уже входите в группу #{@group.name}"
    logger.warn "Пользователь пытался дважды присоединиться к одной и
                 той же группе. Пользовательский интерфейс не должен
                 это разрешать."
  end

  redirect_to :back
end
```

- **error** – этот уровень служит для вывода информации об ошибках, не требующих перезагрузки сервера;
- **fatal** – самое страшное, что может случиться. Ваше приложение теперь неработоспособно, для его перезапуска необходимо ручное вмешательство.

Протоколы Rails

В папке `log` приложения Rails хранится три файла-протокола, соответствующих трем стандартным средам, а также протокол и pid-файл сервера Mongrel. Файлы-протоколы могут расти очень быстро. Чтобы упростить очистку протоколов, предусмотрено задание `rake`:

```
rake log:clear # Усекает все файлы *.log files в log/ до нулевой длины
```

На этапе разработке очень полезным может оказаться содержимое файла `log/development.log`. Разработчики часто открывают окно терминала, в котором запущена команда `tail -f`, чтобы следить, что пишется в этот файл:

```
$ tail -f log/development.log

  User Load (0.000522) SELECT * FROM users WHERE (users.'id' = 1)
  CACHE (0.000000) SELECT * FROM users WHERE (users.'id' = 1)
```

В протокол разработки выводится много интересной информации, например при каждом запросе в протокол – полезные сведения о нем. Ниже приведен пример, взятый из одного моего проекта, и описание выводимой информации:

```
Processing UserPhotosController#show (for 127.0.0.1 at 2007-06-06
17:43:13) [GET]
  Session ID: b362cf038810bb8dec076fcdaec3c009
  Parameters: {"/users/8-Obie-Fernandez/photos/406"=>nil,
  "action"=>"show", "id"=>"406", "controller"=>"user_photos",
  "user_id"=>"8-Obie-Fernandez"}
  User Load (0.000477) SELECT * FROM users WHERE (users.'id' = 8)
  Photo Columns (0.003182) SHOW FIELDS FROM photos
  Photo Load (0.000949) SELECT * FROM photos WHERE (photos.'id' = 406
  AND (photos.resource_id = 8 AND photos.resource_type = 'User'))
Rendering template within layouts/application
Rendering photos/show
  CACHE (0.000000) SELECT * FROM users WHERE (users.'id' = 8)
Rendered adsense/_medium_rectangle (0.00155)
  User Load (0.000541) SELECT * FROM users WHERE (users.'id' = 8)
  LIMIT 1
  Message Columns (0.002225) SHOW FIELDS FROM messages
  SQL (0.000439) SELECT count(*) AS count_all FROM messages WHERE
  (messages.receiver_id = 8 AND (messages.'read' = 0))
Rendered layouts/_header (0.02535)
Rendered adsense/_leaderboard (0.00043)
Rendered layouts/_footer (0.00085)
Completed in 0.09895 (10 reqs/sec) | Rendering: 0.03740 (37%) | DB:
0.01233 (12%) | 200 OK [http://localhost/users/8-Obie-
Fernandez/photos/406]
  User Columns (0.004578) SHOW FIELDS FROM users
```

- контроллер и вызванное действие;
- IP-адрес компьютера, отправившего запрос;
- временной штамп, показывающий, когда поступил запрос;
- идентификатор сеанса, ассоциированного с этим запросом;
- хеш параметров запроса;
- информация о запросе к базе данных, включая время и текст предложения SQL;
- информация о попадании в кэш, включая время и текст SQL-запроса, ответ на который был получен из кэша, а не путем обращения к базе данных;
- информация о рендеринге каждого шаблона, использованного при выводе представления, и время, затраченное на обработку шаблона;
- общее время выполнения запроса и вычисленное по нему число запросов в секунду;
- сравнение времени, потраченного на операции с базой данных и на рендеринг;
- код состояния HTTP и URL для ответа, отправленного клиенту.

Анализ протоколов

Используя протокол разработки и толику здравого смысла, можно легко выполнить различные виды неформального анализа.

Производительность. Изучение производительности приложения – один из напрашивающихся видов анализа. Чем быстрее выполняется запрос, тем больше запросов сможет обслужить данный процесс Rails. Поэтому производительность часто выражается в *запросах в секунду*. Найдите, для каких запросов обращение к базе данных и рендеринг выполняются долго, и разберитесь в причинах.

Важно понимать, что время, сохраняемое в протоколе, не отличается *повышенной точностью*. Оно, скорее, даже неправильно просто потому, что очень трудно замерить временные характеристики процесса, находясь внутри него. Сложив процентные доли времени, затраченного на обращение к базе данных и на рендеринг, вы далеко не всегда получите величину, близкую к 100%.

Однако пусть объективно цифры не точны, зато они дают прекрасную основу для субъективных сравнений в контексте одного и того же приложения. С их помощью мы можете понять, стало ли некоторое действие занимать больше времени, чем раньше, как его время соотносится с временем выполнения другого действия и т. д.

SQL-запросы. ActiveRecord ведет себя не так, как вы ожидали? Протоколирование текста SQL-запроса, сгенерированного ActiveRecord, часто помогает отладить ошибки, связанные со сложными запросами.

Выявление ошибок вида N+1 select. При отображении некоторой записи вместе с ассоциированным с ней набором записей есть шанс допустить так называемую ошибку вида *N+1 select*. Ее признак – наличие серии из многих предложений SELECT, отличающихся только значением первичного ключа.

Вот, например, фрагмент протокола реального приложения Rails, демонстрирующий ошибку N+1 select в том, как загружаются экземпляры класса FlickrPhoto:

```
FlickrPhoto Load (0.001395) SELECT * FROM flickr_photos WHERE
(flickr_photos.resource_id = 15749 AND flickr_photos.resource_type =
'Place' AND (flickr_photos.'profile' = 1)) ORDER BY updated_at desc
LIMIT 1
FlickrPhoto Load (0.001734) SELECT * FROM flickr_photos WHERE
(flickr_photos.resource_id = 15785 AND flickr_photos.resource_type =
'Place' AND (flickr_photos.'profile' = 1)) ORDER BY updated_at desc
LIMIT 1
FlickrPhoto Load (0.001440) SELECT * FROM flickr_photos WHERE
(flickr_photos.resource_id = 15831 AND flickr_photos.resource_type =
'Place' AND (flickr_photos.'profile' = 1)) ORDER BY updated_at desc
LIMIT 1
```

... и так далее, и так далее на протяжении многих и многих страниц протокола. Знакомо?

К счастью, на каждый из этих запросов к базе уходит очень небольшое время – примерно 0,0015 с. Это объясняется тем, что:

1) MySQL исключительно быстро выполняет простые предложения SELECT;
2) мой процесс Rails работал на машине, где находилась база данных.

И тем не менее суммарно эти N запросов способны свести производительность на нет. Если бы не вышеупомянутые компенсирующие факторы, я столкнулся бы с серьезной проблемой, причем наиболее явственно она проявлялась бы при расположении базы данных на отдельной машине, поскольку ко времени выполнения каждого запроса добавлялись бы еще и сетевые задержки.

Проблема N+1 select – это еще не конец света. В большинстве случаев для ее решения достаточно правильно пользоваться параметром :include в конкретном вызове метода find.

Разделение ответственности. Правильно спроектированное приложение на основе паттерна модель-вид-контроллер следует определенным

протоколам, описывающим распределение по логическим ярусам операций с базой данных (которая выступает в роли модели) и рендеринга (вид). Вообще говоря, желательно, чтобы контроллер взял на себя загрузку из базы всех данных, которые понадобятся для рендеринга. В Rails это достигается за счет того, что код контроллера запрашивает у модели необходимые данные и сохраняет их в переменных экземпляра, доступных виду (представлению).

Доступ к базе данных на этапе рендеринга обычно считается дурной практикой. Вызов методов `find` напрямую из кода шаблона нарушает принцип разделения ответственности и способен стать причиной ночных кошмаров у персонала службы сопровождения[1].

Однако существует немало возможностей для ползучего проникновения в ваш код неявных операций доступа к базе данных на этапе рендеринга. Иногда эти операции инкапсулируются в модели, а иногда выполняются в ходе отложенной загрузки ассоциаций. Можем ли мы решительно осудить такую практику? Трудно дать определенный ответ. Бывают случаи (например, при кэшировании фрагментов), когда обращение к базе на этапе рендеринга имеет смысл.

Использование альтернативных схем протоколирования

Легко! Достаточно присвоить одной из переменных класса `logger`, например `ActiveRecord::Base.logger`, объект класса, совместимого с классом `Logger` из стандартного дистрибутива Ruby.

Простой прием, основанный на возможности подмены объектов протоколирования, демонстрировался Дэвидом на различных встречах, в том числе в основном докладе на конференции Railsconf 2007. Открыв консоль, присвойте `ActiveRecord::Base.logger` новый экземпляр класса `Logger`, указывающий на `STDOUT`. Это позволит вам просматривать генерируемые SQL-запросы прямо на консоли. Джемис подробно рассматривает эту технику и другие возможности на странице *http://weblog.jamisbuck.org/2007/1/31/more-on-watchingactiverecord*.

Syslog

В различных вариантах ОС UNIX имеется системная служба `syslog`. Есть ряд причин, по которым она может оказаться более удобным средством протоколирования работы Rails-приложения в режиме эксплуатации:

[1] Практически все когда-либо написанные приложения для PHP страдают от этой проблемы.

- более точный контроль над уровнями протоколирования и содержимым сообщений;
- консолидация протоколов нескольких приложений Rails;
- при использовании дистанционных средств syslog возможна консолидация протоколов приложений Rails, работающих на разных серверах. Конечно, это удобнее, чем обрабатывать разрозненные протоколы, хранящиеся на каждом сервере приложений в отдельности.

Можно воспользоваться написанной Эриком Ходелем (Eric Hodel) библиотекой SyslogLogger[1], чтобы организовать интерфейс приложения Rails с `syslog`. Для этого придется загрузить библиотеку, затребовать ее с помощью `require` в сценарии `environment.rb` и подменить экземпляр `RAILS_DEFAULT_LOGGER`.

Заключение

Мы начали путешествие в мир Rails с обзора различных сред исполнения Rails и механизма загрузки зависимостей, в том числе и кода вашего приложения. Подробно рассмотрев сценарий `environment.rb` и его варианты, зависящие от режима, мы узнали, как настроить поведение Rails под свои нужды. В процессе обсуждения различных версий библиотек Rails, используемых в конкретном проекте, мы попутно затронули вопрос о «сидении на острие» и о том, когда оно имеет смысл.

Мы также изучили процедуру начальной загрузки Rails, для чего потребовалось заглянуть в исходные тексты (мы и дальше будем при необходимости совершать такие погружения в исходный код Rails).

В главе 2 «Работа с контроллерами» мы продолжим путешествие и рассмотрим диспетчер Rails и `ActionController`.

[1] *http://seattlerb.rubyforge.org/SyslogLogger/*.

2

Работа с контроллерами

Уберите всю бизнес-логику из контроллеров и переместите ее в модель. Контроллеры должны отвечать только за отображение URL (включая и данные из других HTTP-запросов), координацию между моделями и видами и отправку результатов в виде HTTP-ответа. Попутно контроллеры могут заниматься контролем доступа, но больше почти ничем. Эти указания очень определенны, но для того чтобы им следовать, необходима интуиция и тонкий расчет.

Ник Каллен, Pivotal Labs
http://www.pivotalblabs.com/articles/2007/07/16/the-controller-formula

В Rails, как и в любом другом приложении, имеется поток передачи управления между частями программы. Но в Rails этот поток довольно сложен. Среда состоит из многих компонентов, которые вызывают друг друга. Среди прочего платформа должна на лету определить, какие файлы приложения вызываются и что в них находится. Разумеется, решение этой задачи зависит от конкретного приложения.

Сердце данного механизма – *контроллер*. Клиент, обращающийся к вашему приложению, просит его выполнить некое *действие контроллера*. Происходить это может разными способами, и в некоторых граничных случаях не происходит вовсе... но, если вы знаете, какое место контроллер занимает в жизненном цикле приложения, то сможете разобраться, как с ним сопрягается все остальное. Вот почему мы рассматриваем контроллеры ранее других частей части Rails API.

В аббревиатуре MVC (модель-вид-контроллер) контроллер обозначается буквой «С». После диспетчера контроллер является первым из компонентов, обрабатывающих входящий запрос. Контроллер отвечает за поток управления в программе; он извлекает информацию из базы данных (обычно с помощью интерфейса `ActiveRecord`) и предоставляет ее видам (представлениям).

Контроллеры очень тесно связаны с видами – более тесно, чем с моделями. Можно написать весь слой приложения, относящийся к модели, не создав ни единого контроллера, или поручить работу над контроллером и моделью разным людям, которые никогда не общаются между собой. С другой стороны, виды и контроллеры сцеплены гораздо сильнее. Они разделяют много общей информации, представленной, главным образом, в форме переменных экземпляра. Это означает, что имена переменных, выбранные в контроллере, влияют на действия в представлении.

В этой главе мы рассмотрим, что происходит на пути к исполнению действия контроллера и что получается в результате. По ходу мы обратим пристальное внимание на настройку самих классов контроллеров, особенно в том, что касается многообразных способов рендеринга представлений. Завершим мы эту главу обсуждением еще двух тем, относящихся к контроллерам: фильтров и потоковой отправки.

Диспетчер: с чего все начинается

Среда Rails используется для создания веб-приложений, поэтому сначала *запрос* обрабатывается веб-сервером: Apache, Lighttpd, Nginx и т. д. Затем сервер переправляет запрос приложению Rails, где он попадает к *диспетчеру*.

Обработка запроса

Выполнив свою часть обработки запроса, сервер передает диспетчеру различную информацию:

- URI запроса (например, *http://localhost:3000/timesheets/show/3* или что-то в этом роде);
- окружение CGI (список имен параметров CGI и соответствующих им значений).

В задачу диспетчера входит:

- выяснить, какой контроллер должен обработать запрос;
- определить, какое действие следует выполнить;
- загрузить файл нужного контроллера, который содержит определение класса контроллера на языке Ruby (например, TimesheetsController);
- создать экземпляр класса контроллера;
- сказать этому экземпляру, какое действие нужно выполнить.

Все это происходит быстро и незаметно для вас. Маловероятно, что вам когда-нибудь придется копаться в исходном коде диспетчера; можете просто полагаться на то, что эта штука работает, и работает правильно. Но чтобы по-настоящему осмыслить путь Rails, важно понимать, что происходит внутри диспетчера. В частности, необходимо помнить, что различные части вашего приложения – это просто фрагменты (иногда весьма объемные) написанного на Ruby кода, которые загружаются в работающий интерпретатор Ruby.

Познакомимся с диспетчером поближе

В педагогических целях выполним функции диспетчера вручную. Так вы сможете лучше почувствовать поток управления в приложениях Rails.

Для этого небольшого упражнения запустим новое приложение Rails:

```
$ rails dispatch_me
```

Теперь создадим простой контроллер с действием `index`:

```
$ cd dispatch_me/
$ ruby ./script/generate controller demo index
```

Заглянув в код только что сгенерированного контроллера в файле `app/controllers/demo_controller.rb`, вы обнаружите в нем действие `index`:

```
class DemoController < ApplicationController
def index
end
end
```

Сценарий `generate` также автоматически создал файл `app/views/demo/index.rhtml`, который содержит шаблон представления, соответствующего этому действию. Шаблон включает некоторые подстановочные переменные. Чтобы не усложнять задачу, заменим его более простым файлом, который сможем опознать с первого взгляда. Сотрите все содержимое файла `index.rhtml` и введите такую строку:

```
Hello!
```

Ее не назовешь дизайнерским шедевром, но для наших целей сойдет. Итак, мы выстроили косточки домино в ряд, теперь пора толкнуть переднюю: диспетчер. Для этого запустим консоль Rails, находясь в каталоге приложения. Введите команду `ruby script/console`:

```
$ ruby script/console
Loading development environment.
>>
```

Теперь мы находимся в самом сердце приложения Rails, которое ожидает инструкций.

Обычно при передаче запроса диспетчеру Rails веб-сервер устанавливает две переменные окружения. Поскольку мы собираемся вызвать диспетчер вручную, эти переменные придется установить самостоятельно:

```
>> ENV['REQUEST_URI'] = "/demo/index"
=> "/demo/index"
>> ENV['REQUEST_METHOD'] = "get"
=> "get"
```

Теперь мы готовы обмануть диспетчер, заставив его думать, будто он получил запрос. На самом деле, диспетчер *действительно* получает запрос только не от сервера, а от человека, сидящего за консолью.

Вот как выглядит команда:

```
>> Dispatcher.dispatch
```

А вот и ответ от приложения Rails:

```
Content-Type: text/html; charset=utf-8
Set-Cookie: _dispatch_me_session_id=336c1302296ab4fa1b0d838d; path=/
Status: 200 OK
Cache-Control: no-cache
Content-Length: 7

Hello!
```

Мы вызвали метод `dispatch` класса `Dispatcher`, и в результате было выполнено действие `index` и рендеринг соответствующего шаблона (в том виде, в каком мы его оставили), к результатам рендеринга добавлены HTTP-заголовки, и все вместе возвращено нам.

Теперь представьте: если бы вы были не человеком, а веб-сервером, и проделали все то же самое, то сейчас могли бы вернуть документ, состоящий из заголовков и строки Hello!, клиенту. Именно так все и происходит. Загляните в подкаталог `public` приложения `dispatch_me` (или любого другого приложения Rails). Среди прочего вы найдете там следующие файлы диспетчера:

```
$ ls dispatch.*
dispatch.cgi dispatch.fcgi dispatch.rb
```

Каждый раз, когда Rails получает запрос, веб-сервер передает управление одному из этих файлов. Какому *именно*, зависит от конфигурации сервера. В конечном итоге, все они делают одно и то же: вызывают метод класса `Dispatcher.dispatch`, как мы перед этим сделали с консоли.

Вы можете пройти по следу и дальше, ознакомившись с файлом `public/.htaccess` и конфигурацией своего сервера. Но для понимания цепочки событий, возникающих при получении запроса к Rails, и той роли, которую играет в них контроллер, сказанного выше достаточно.

Рендеринг представления

Цель типичного действия контроллера – выполнить рендеринг шаблона представления, то есть заполнить шаблон и передать результаты (обычно в форме HTML-документа) серверу, чтобы тот доставил их клиенту.

Как ни странно (по крайней мере, это может показаться немного странным, хотя и не лишенным логики), можно не определять действие контроллера, *если существует шаблон с таким же именем, как у действия*.

Можете проверить это в ручном режиме. Откройте файл `app/controller/demo_controller.rb` и удалите действие `index`, после чего файл будет выглядеть так:

```
class DemoController < ApplicationController
end
```

Не удаляя файл `app/views/demo/index.rhtml`, попробуйте выполнить с консоли то же упражнение, что и выше (вызвать метод `Dispatcher.dispatch` и т. д.). Результат получится точно таким же, как и раньше.

Кстати, не забывайте перезагружать консоль после внесения изменений – автоматически она не распознает, что код изменился. Самый простой способ перезагрузить консоль – просто ввести команду `reload!`. Но имейте в виду, что все существующие экземпляры `ActiveRecord`, на которые вы храните ссылки, также придется перезагрузить (с помощью их собственных методов `reload`). Иногда проще выйти из консоли и запустить ее заново.

Если сомневаетесь, рисуйте

Rails знает, что, получив запрос к действию `index` демонстрационного контроллера, он должен любой ценой вернуть что-то серверу. Раз действия `index` в файле контроллера нет, Rails пожимает плечами и говорит: «Что ж, если бы действие `index` было, оно все равно оказалось бы пустым и я бы выполнил рендеринг шаблона `index.rhtml`. Так и сделаю это».

Однако даже на примере пустого действия контроллера кое-чему можно научиться. Вернемся к исходной версии демонстрационного контроллера:

```
class DemoController < ApplicationController
  def index
  end
end
```

Можно сделать вывод, что если в действии контроллера *ничего другого не задано*, то по умолчанию выполняется рендеринг шаблона, имя ко-

торого соответствует имени контроллера и действия. В данном случае это шаблон `views/demo/index.rhtml`.

Иными словами, в каждом действии контроллера имеется неявная команда `render`. Причем `render` – это самый настоящий метод. Предыдущий пример можно было бы переписать следующим образом:

```
def index
render :template => "demo/index"
end
```

Но это необязательно, поскольку и так подразумевается, что именно данное действие нужно вам. Это частный случай принципа *примата соглашения над конфигурацией*, часто упоминаемого в разговорах разработчиков для Rails. Не заставляйте разработчика писать код, который можно было бы добавить по соглашению.

Однако команда `render` – это не просто способ попросить Rails выполнить то, что он и так собирался сделать.

Явный рендеринг

Рендеринг шаблона – это как выбор рубашки. Если вам не нравится самая ближняя из висящих в шкафу – назовем ее рубашкой по умолчанию, – можно протянуть руку и достать другую.

Если действие контроллера не хочет рисовать шаблон по умолчанию, то может нарисовать любой другой, вызвав метод `render` явно. Доступен любой шаблон, находящийся в поддереве с корнем `app/views` (на самом деле, это не совсем точно – Доступен вообще любой шаблон в системе). Но зачем контроллеру может понадобиться выполнять рендеринг шаблона, отличного от шаблона по умолчанию? Причин несколько, и, познакомившись с некоторыми, мы сможем узнать о многих полезных возможностях метода контроллера `render`.

Рендеринг шаблона другого действия

Типичный случай, когда нужно выполнять рендеринг совсем другого шаблона, – это повторный вывод формы, в которую пользователь ввел недопустимые данные. Обычно в такой ситуации выводят ту же самую форму с данными, введенными пользователем, а также дополнительную информацию об ошибках, чтобы пользователь мог их исправить и отправить форму еще раз.

Причина, по которой в этом случае приходится рисовать не шаблон по умолчанию, заключается в том, что действие *обработки* формы и действие *вывода* формы – чаще всего разные действия. Поэтому действию, которое обрабатывает форму, необходим способ повторно вывести исходный шаблон (форму), а не переходить к следующему экрану, как в случае корректно заполненной формы.

Ух, какое многословное объяснение получилось. Вот практический пример:

```
class EventController < ActionController::Base
  def new
    # Это (пустое) действие выполняет рендеринг шаблона new.rhtml, который
    # содержит форму для ввода информации о новом событии, оно по существу
    # не нужно.
  end

  def create
    # Этот метод обрабатывает данные из формы. Данные доступны через
    # вложенный хеш params, являющийся значением ключа :event.
    @event = Event.new(params[:event])
    if @event.save
      flash[:notice] = "Событие создано!"
      redirect_to :controller => "main" # пока не обращайте внимания на
                                        # эту строку
    else
      render :action => "new" # не выполняет метод new!
    end
  end
end
```

В случае ошибки – когда вызов `@event.save` не возвращает `true` – мы снова выполняем рендеринг шаблона `new` то есть файла `new.rhtml`. В предположении, что шаблон `new.rhtml` написан правильно, будет автоматически включена информация об ошибке, хранящаяся в новом (но не сохраненном) объекте `@event` класса `Event`.

Отметим, что сам шаблон `new.rhtml` не «знает», что он рисуется действием `create`, а не `new`. Он просто делает то, что требуется: выполняет подстановки на основе содержащихся в нем инструкций и переданных контроллером данных (в данном случае – объекта `@event`).

Рендеринг совершенно постороннего шаблона

Точно так же, как рисуется шаблон другого действия, выполняется и рендеринг любого хранящегося в системе шаблона. Для этого следует вызвать метод `render`, передав в параметре `:template` или `:file` путь к нужному файлу шаблона.

Параметр `:template` должен содержать путь относительно корня дерева шаблонов (`app/views`, если вы ничего не меняли, что было бы весьма необычно), а параметр `:file` – абсолютный путь в файловой системе.

Честно говоря, параметр `:template` редко используется при разработке приложений Rails.

```
render :template => "abuse/report" # рендеринг app/views/abuse/report.rhtml
render :file => "/railsapps/myweb/app/views/templates/common.rhtml"
```

Рендеринг подшаблона

Еще один случай – рендеринг *подшаблона* (partial template или просто partial). Вообще говоря, подшаблоны позволяют представить всю совокупность шаблонов в виде небольших файлов, избежав громоздкого кода и выделив модули, допускающие повторное использование.

Контроллер прибегает к рендерингу подшаблонов чаще всего для AJAX-вызовов, когда необходимо динамически обновлять участки уже выведенной страницы. Эта техника, равно как и вообще рассмотрение подшаблонов, более подробно излагается в главе 10 «Компонент ActionView».

Рендеринг встроенного шаблона

Иногда броузеру нужно послать результат трансляции какого-нибудь фрагмента шаблона, который слишком мал, чтобы оформлять его в виде отдельной части. Признаю, что такая практика спорна, так как является вопиющим нарушением принципа разделения ответственности между различными слоями MVC.

Один из часто встречающихся случаев употребления встроенного рендеринга и, пожалуй, единственная причина, по которой такая возможность вообще включена, – это использование помощников при обработке AJAX-запросов, например `auto_complete_result` (см. главу 12 «Ajax on Rails»).

```
render :inline => „<%= auto_complete_result(@headings, 'name') %>"
```

Rails обрабатывает такой встроенный код точно так же, как если бы это был шаблон представления.

> **Говорит Кортенэ...**
>
> Будь вы моим подчиненным, я отругал бы вас за использование в контроллере кода, относящегося к представлениям, даже если это всего одна строка.
>
> То, что относится к представлениям, должно там и находиться!

Рендеринг текста

Что если нужно отправить броузеру всего лишь простой текст, особенно когда речь идет об ответах на AJAX-запросы и некоторые запросы к веб-службам?

```
render :text => 'Данные приняты'
```

Рендеринг структурированных данных других типов

Команда `render` принимает ряд параметров, облегчающих возврат структурированных данных в таких форматах, как JSON или XML. При этом в ответе правильно выставляется заголовок `content-type` и другие характеристики.

:json

JSON[1] – это небольшое подмножество языка JavaScript, применяемое в качестве простого формата обмена данными. Чаще всего он используется для отправки данных JavaScript-сценарию, который работает на стороне клиента в обогащенном веб-приложении и посылает серверу AJAX-запросы. В библиотеку `ActiveRecord` встроена поддержка для преобразования в формат JSON, что делает Rails идеальной платформой для возврата данных в этом формате, например:

```
render :json => @record.to_json
```

:xml

В `ActiveRecord` встроена также поддержка для преобразования в формат XML, например:

```
render :xml => @record.to_xml
```

Вопросы, связанные с XML, мы будем подробно рассматривать в главе 15 «XML и ActiveResource».

Пустой рендеринг

Редко, но бывает, что не нужно рисовать вообще ничего (для обхода ошибки: в броузере Safari «ничего» на самом деле означает отправку броузеру одного пробела).

```
render :nothing => true, :status => 401 # Не авторизован
```

Стоит отметить, что, как показано в этом примере, `render :nothing => true` часто используется в сочетании с некоторым кодом состояния HTTP (см. раздел «Параметры рендеринга»).

Параметры рендеринга

В большинстве обращений к методу `render` задаются дополнительные параметры. Ниже они перечислены в алфавитном порядке.

:content_type

С любым контентом, который циркулирует в Сети, ассоциирован тип MIME[2]. Например, HTML-контенту соответствует тип `text/html`. Но

---

[1] Дополнительную информацию о JSON см. на сайте *http://www.json.org/*.

[2] Спецификация MIME занимает пять документов RFC, поэтому удобнее ознакомиться с вполне приличным описанием в «Википедии» на странице *http://en.wikipedia.org/wiki/MIME*.

иногда необходимо отправить клиенту данные в формате, отличном от HTML. Rails не проверяет формат переданного идентификатора MIME, поэтому вы должны сами позаботиться о том, чтобы значение параметра `:content_type` было допустимым.

:layout

По умолчанию Rails придерживается определенных соглашений о шаблоне размещения, в который обертывается ваш ответ. Эти соглашения подробно рассматриваются в главе 10 «Компонент ActionView». Параметр `:layout` позволяет указать, нужен вам шаблон размещения или нет.

:status

В протоколе HTTP определено много стандартных кодов состояния[1], описывающих вид ответа на запрос клиента. В большинстве типичных случаев Rails выбирает подходящий код автоматически, например `200 OK` для успешно обработанного запроса.

Для изложения теории и практики использования всех возможных кодов состояния HTTP потребовалась бы отдельная глава, а то и целая книга. Для удобства в табл. 2.1 приведено несколько кодов, которые, по моему опыту, полезны при повседневном программировании для Rails.

Таблица 2.1. Общеупотребительные коды состояния HTTP

Код состояния	Описание

307 Temporary Redirect Запрошенному ресурсу временно присвоен другой URI	Иногда необходимо временно переадресовать пользователя на другое действие, например, потому что работает какой-то длительный процесс или учетная запись владельца конкретного ресурса приостановлена.
	Этот код состояния говорит, что текущий URI запрошенного ресурса указан в HTTP-заголовке `Location`. Поскольку методу `render` не передается хеш заголовков ответа, вы должны установить их самостоятельно перед вызовом `render`. К счастью, хеш `response` находится в области видимости методов контроллера, как в примере:
	`def paid_resource`
	`  if current_user.account_expired?`
	`  response.headers['Location'] =`
	`  account_url(current_user)`
	
	`render :text => "Account expired", :status =>`
	`307`
	`  end`
	`end`

[1] Полный перечень кодов состояния HTTP см. на странице *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*

Таблица 2.1. Общеупотребительные коды состояния HTTP (окончание)

Код состояния	Описание
401 Unauthorized	Иногда пользователь не предоставляет верительных грамот, необходимых для просмотра ресурса с ограниченным доступом, или процедура аутентификации/авторизации завершается неудачно. Если применяется базовая (Basic) схема аутентификации или аутентификация дайджестом (Digest Authentication), вы, скорее всего, должны вернуть код `401`

403 Forbidden
Сервер понял запрос, но отказывается его выполнять
	Я предпочитаю использовать код `403` в сочетании с коротким сообщением (`render :text`) в ситуации, когда клиент запросил ресурс, который в обычных обстоятельствах недоступен через интерфейс веб-приложения. Иными словами, запрос, скорее всего, был сформирован искусственно. Человек или робот с добрыми или дурными намерениями (это неважно) пытается заставить сервер делать то, что он делать не должен.
	Например, приложение Rails, над которым я сейчас работаю, открыто для всех, и ежедневно на него заходит робот GoogleBot. Возможно, из-за когда-то существовавшей ошибки был проиндексирован URL `/favorites`.
	Однако каталог `/favorites` должен быть доступен только зарегистрированным пользователям. Но, коль скоро Google знает об этом URL, он будет заходить туда снова и снова.
	Вот как я его торможу:
	`def index`
	`  return render :nothing => true,`
	`  :status => 403 unless logged_in?`
	`  @favorites = current_user.favorites.find(:all)`
	`end`

404 Not Found Сервер не может найти запрошенный ресурс	Код 404 можно использовать, например, когда ресурс с указанным идентификатором отсутствует в базе данных (то ли потому что идентификатор указан неверно, то ли потому что ресурс удален). Например, ресурса, соответствующего запросу «GET /people/2349594934896107», в нашей базе данных нет вовсе, так что же мы должны показать? Сообщение о том, что человека с таким идентификатором не существует? Нет, в соответствии с архитектурным стилем REST правильно вернуть ответ с кодом 404. А если у вас параноидальные наклонности и вы знаете, что такой ресурс существовал в прошлом, то можете послать в ответ код `410 Gone`

Код состояния	Описание
503 Service Unavailable Сервер временно недоступен	Код 503 удобен, когда сайт на некоторое время закрывается на техническое обслуживание, особенно при обновлении веб-служб, написанных в соответствии с архитектурным стилем REST. Один из рецензентов книги, Сьюзан Поттер, поделилась таким советом: *В своих проектах для Rails я создаю приложение-заглушку, которое возвращает код 503 в ответ на любой запрос. Клиентами моих служб обычно выступают другие службы, поэтому разработчики клиентов, потребляющих мои веб-службы, знают, что это временный перерыв, вызванный, скорее всего, плановым обслуживанием (заодно это служит напоминанием о необходимости просматривать, а не игнорировать, электронную почту, которую я отправляю по выходным).*

Переадресация

Жизненный цикл приложения Rails разбит на запросы. При поступлении каждого нового запроса все начинается сначала.

Рендеринг шаблона, который подразумевается по умолчанию, является альтернативным, частичным, просто текстом или чем-нибудь еще, – это последний шаг обработки запроса. *Переадресация* же означает, что обработка текущего запроса завершается, и начинается обработка нового.

Рассмотрим еще раз пример метода `create` для обработки формы:

```
def create
  @event = Event.new(params[:event])
  if @event.save
    flash[:notice] = "Событие создано!"
    redirect_to :controller => "main"
  else
    render :action => "new"
  end
end
```

Если операция сохранения завершается успешно, мы записываем сообщение в хеш `flash` и вызываем метод `redirect_to` для перехода к совершенно другому действию. В данном случае это действие `index` (оно не задано явно, но принимается по умолчанию) контроллера *`main`*.

Смысл в том, что при сохранении записи о новом событии `Event` надлежит вернуть пользователя к представлению верхнего уровня. Так почему просто не выполнить рендеринг шаблона `main/index.rhtml`?

Говорит Кортенэ...

Помните, что код, следующий за вызовом метода `redirect` или `render`, исполняется и приложение отправит данные броузеру не раньше, чем он завершится.

В случае сложной логики часто бывает желательно вернуться сразу после обращения к `redirect` или `render`, находящегося глубоко внутри последовательности предложений `if`, чтобы предотвратить ошибку `DoubleRenderError`:

```
def show
  @user = User.find(params[:id])
  if @user.activated?
    render :action => 'activated' and return
  end
  case @user.info
  ...
  end
end
```

```
if @event.save
  flash[:notice] = "Событие создано!"
  render :controller => "main", :action => "index"
  ...
```

В результате действительно будет выполнен рендеринг шаблона `main/index.rhtml`. Но тут есть подводные камни. Предположим, например, что действие `main/index` выглядит следующим образом:

```
def index
  @events = Event.find(:all)
end
```

Если выполнить рендеринг `index.rhtml` из действия `event/create`, то действие `main/index` *не будет выполнено*. Поэтому переменная `@events` останется неинициализированной. Следовательно, при рендеринге `index.rhtml` возникнет ошибка, так как в этом шаблоне (предположительно) используется `@events`:

```
<h1>Schedule Manager</h1>
<p>Текущий список ваших событий:</p>
<% @events.each do |event| %>
  здесь какая-то HTML-разметка
<% end %>
```

Вот почему мы должны выполнить переадресацию на действие `main/index`, а не просто позаимствовать его шаблон. Команда `redirect_to` начнет все с чистого листа: создаст новый запрос, инициирует новое действие и решит, по какому шаблону выводить ответ.

Говорит Себастьян...

Какая переадресация правильна?

Используя метод `redirect_to`, вы говорите пользовательскому агенту (то есть броузеру), что необходимо выполнить новый запрос с другим URL. Такой ответ может интерпретироваться по-разному, поэтому в современной спецификации протокола HTTP определены четыре разных кода состояния для переадресации.

В старой версии HTTP 1.0 было два кода: 301 Moved Permanently (Перемещен постоянно) и 302 Moved Temporarily (Перемещен временно). Постоянная переадресация означает, что пользовательский агент должен забыть о старом URL и использовать новый, обновив все хранящиеся ссылки (например, закладку или в Google запись в поисковой базе данных). Временная переадресация – это одноразовое действие. Исходный URL все еще действителен, но для данного конкретного запроса агент должен запросить ресурс с указанным URL.

Однако тут кроется проблема: какой метод использовать для переадресованного запроса, если первоначально был выполнен POST-запрос? В случае постоянной переадресации безопасно предположить, что новый запрос должен выполняться методом GET, так как это справедливо для всех сценариев применения. Но временная переадресация используется как для переадресации на представление ресурса, только что модифицированного в ходе обработки исходного POST-запроса (наиболее часто встречающийся случай), так и для переадресации всего исходного POST-запроса на новый URL, который и должен позаботиться об его обработке.

В HTTP 1.1 это проблема решена путем определение двух новых кодов состояния: 303 See other (См. в другом месте) и 307 Temporary Redirect (Временная переадресация). Код 303 говорит пользовательскому агенту, что нужно выполнить GET-запрос вне зависимости от того, каким методом выполнялся исходный запрос, а 307 – что нужно обязательно использовать *тот же самый* метод, что и для исходного запроса.

Большинство современных броузеров трактует код 302 так же, как 303, то есть посылают GET-запрос. Именно поэтому метод `redirect_to` в Rails по-прежнему отправляет код 302. Код 303 был бы лучше, поскольку при этом не остается места для интерпретации (а, стало быть, и путаницы), но я подозреваю, что никому эта проблема не показалась достаточно серьезной, чтобы поместить запрос на исправление.

Если вам когда-нибудь потребуется переадресация с кодом 307, например, чтобы продолжить обработку POST-запроса в другом действии, всегда можно организовать это самостоятельно, для чего достаточно записать путь в заголовок `response.header["Location"]`, а затем выполнить рендеринг, вызвав метод `render :status => 307`.

Коммуникация между контроллером и представлением

При выполнении рендеринга шаблона обычно используются данные, которые контроллер извлек из базы. Иными словами, контроллер получает то, что ему нужно, от модели и передает представлению.

В Rails передача данных от контроллера представлению осуществляется с помощью переменных экземпляра. Обычно действие контроллера инициализирует одну или несколько переменных. Затем они могут использоваться представлением.

В выборе переменных, через которые осуществляется обмен данными, кроется некая ирония (и возможный источник путаницы для новичков). Основная причина, по которой эти переменные вообще существуют, заключается в том, чтобы объекты (будь то объекты `Controller`, `String` или какие-то другие) могли хранить ссылки на данные, которые они *не* разделяют с другими объектами. При выполнении действия контроллера все происходит в контексте объекта контроллера – скажем, экземпляра класса `DemoController` или `EventController`. Говоря «контекст», мы имеем в виду и то, что любая переменная экземпляра в коде принадлежит экземпляру контроллера.

Но рендеринг шаблона выполняется в контексте другого объекта – экземпляра класса `ActionView::Base`. У этого объекта есть собственные переменные экземпляра, и он не имеет доступа к переменным экземпляра контроллера.

Поэтому, на первый взгляд, переменные экземпляра – это самый плохой способ организовать совместный доступ двух объектов к общим данным. Однако это возможно; по крайней мере, можно сделать так, что будет казаться, будто общий доступ имеется. В действительности Rails в цикле обходит все переменные объекта контроллера и для каждой из них создает переменную экземпляра в объекте представления с тем же именем и данными.

Для среды это довольно тяжелая работа – все равно что вручную копировать список вещей, которые нужно купить в бакалейной лавке. Зато жизнь программиста упрощается. Если вы сторонник концептуальной чистоты Ruby, то можете скривиться при мысли о том, что переменные экземпляра служат для связывания объектов, а не их отделения друг от друга. Но, с другой стороны, поборник чистоты Ruby должен понимать, что в Ruby можно делать массу самых разных вещей, в том числе и копировать переменные экземпляра в цикле. Ничего противоречащего идеологии Ruby в этом нет. А с точки зрения программиста это дает возможность организовать прозрачную связь между контроллером и шаблоном, который он рисует.

Фильтры

Фильтры позволяют контроллеру выполнять пред- и постобработку своих действий. Фильтры можно использовать для аутентификации, кэширования или аудита, перед тем как выполнять требуемое действие. Методы фильтрации реализованы как *макросы*, то есть находятся в начале метода контроллера, в контексте класса, перед определением других методов. Мы опускаем скобки вокруг аргументов метода, чтобы подчеркнуть декларативную природу фильтров:

```
before_filter :require_authentication
```

Как и большинству других *макро*методов в Rails, методу фильтрации можно передать произвольное число символов:

```
before_filter :security_scan, :audit, :compress
```

или расположить их в отдельных строках:

```
before_filter :security_scan
before_filter :audit
before_filter :compress
```

В отличие от чем-то похожих методов обратного вызова в `ActiveRecord`, невозможно реализовать метод фильтрации в контроллере, просто добавив метод с именем `before_filter` или `after_filter`.

Говорит Кортенэ...

Некоторые любят использовать фильтры для загрузки записи, когда операция ожидает всего одну запись, а логика относительно сложна. Разумеется, переменные экземпляра, установленные фильтром, доступны любым действиям. Но это спорный подход; некоторые разработчики считают, что все обращения к базе данных должны быть вынесены из фильтра и помещены в метод `action`.

```
before_filter :load_product, :only => [ :show,
:edit, :update, :destroy ]
def load_product
  @product =
current_user.products.find_by_permalink(params[:id]
)
  redirect_to :action => 'index' and return false
unless @product.active?

end
```

Методы фильтрации следует объявлять как `protected` или `private`, в противном случае их можно будет вызывать как открытые действия контроллера (с помощью маршрута, принимаемого по умолчанию).

Важно, что фильтры имеют доступ к запросу, ответу и всем переменным экземпляра, которые устанавливаются другими фильтрами в цепочке или самим действием (в случае фильтров `after`). Фильтры могут устанавливать переменные экземпляра, которые будут использоваться в запрошенном действии; очень часто так и поступают.

Наследование фильтров

Фильтры распространяются вниз по иерархии наследования контроллеров. В типичном приложении Rails имеется класс `ApplicationController`, которому наследуют все остальные контроллеры, поэтому, если вы хотите, чтобы некий фильтр выполнялся в любом случае, поместите его именно в этот класс.

```
class ApplicationController < ActionController::Base
after_filter :compress
```

Подклассы могут добавлять и/или пропускать ранее определенные фильтры, не оказывая влияния на суперкласс. Рассмотрим, например, взаимодействие двух связанных отношением наследования классов (листинг 2.1).

Листинг 2.1. Два кооперативных фильтра before

```
class BankController < ActionController::Base

  before_filter :audit

  private

    def audit
      # Записать действия и параметры этого контроллера в контрольный журнал
    end

end

class VaultController < BankController

  before_filter :verify_credentials

  private

    def verify_credentials
      # проверить, что пользователю разрешен доступ в хранилище
    end

end
```

Перед выполнением любого действия контроллера `BankController` (или его подкласса) будет вызван метод `audit`. Для действий же контроллера `VaultController` сначала вызывается метод `audit`, а потом `verify_credentials`, поскольку фильтры заданы именно в таком порядке (фильтры исполняются в контексте класса, в котором объявлены, а класс `BankController` должен быть загружен раньше, чем `VaultController`, так как является родителем последнего).

Если метод `audit` по какой-то причине вернет `false`, то ни метод `verify_credentials`, ни запрошенное действие не выполняются. Это называется *прерыванием цепочки фильтров* (halting the filter chain), и, заглянув в протокол режима обработки, вы обнаружите в нем запись о том, что фильтр такой-то прервал обработку запроса.

Типы фильтров

Фильтры можно реализовать одним из трех способов: ссылкой на метод (символ), внешним классом или встроенным методом (Proc-объектом). Первый способ встречается чаще всего; в этом случае фильтр ссылается на какой-нибудь защищенный или закрытый метод где-то в иерархии наследования контроллера. В листинге 2.1 так реализованы фильтры в обоих классах `BankController` и `VaultController`.

Классы фильтров

С помощью внешних классов проще реализовать повторно используемые фильтры, например, для сжатия выходной информации. Для этого в любом классе определяется статический метод фильтрации, и этот класс передается фильтру, как показано в листинге 2.2.

Листинг 2.2. Фильтр сжатия выходной информации

```
class OutputCompressionFilter
  def self.filter(controller)
    controller.response.body = compress(controller.response.body)
  end
end

class NewspaperController < ActionController::Base
  after_filter OutputCompressionFilter
end
```

Метод `self.filter` класса `Filter` передается экземпляру фильтруемого контроллера, при этом фильтр получает доступ ко всем аспектам контроллера и может манипулировать последним по своему усмотрению.

Встроенная фильтрация

Встраивание (путем передачи параметра-блока методу фильтрации) можно применять для выполнения короткой операции, не требующей

пояснений, или просто в качестве теста на скорую руку. Работает этот способ следующим образом:

```
class WeblogController < ActionController::Base
 before_filter {|controller| false if controller.params["stop"]}
end
```

Как видите, блок ожидает, что ему будет передан контроллер, после того как последний запишет ссылку-запрос во внутренние переменные. Это означает, что блок имеет доступ к объектам запроса и ответа вместе со всеми вспомогательными методами для доступа к параметрам, сеансу, шаблону и т. д. Отметим, что встроенный метод не обязан быть блоком – любой объект, отвечающий на вызов метода `call`, например `Proc` или `Method`, тоже подойдет.

Фильтры `around` ведут себя несколько иначе, чем обычные фильтры `before` и `after` (подробнее об этом см. раздел, посвященный `around`-фильтрам).

Упорядочение цепочки фильтров

Методы `before_filter` и `after_filter` добавляют указанные фильтры в конец цепочки существующих. Обычно именно это и требуется, но иногда порядок выполнения фильтров важен. В таких случаях можно воспользоваться методами `prepend_before_filter` и `prepend_after_filter`. Фильтр помещается в начало соответствующей цепочки и выполняется раньше всех остальных (листинг 2.3).

Листинг 2.3. Пример добавления фильтров before в начало цепочки

```
class ShoppingController < ActionController::Base
  before_filter :verify_open_shop

class CheckoutController < ShoppingController
  prepend_before_filter :ensure_items_in_cart, :ensure_items_in_stock
```

Теперь цепочка фильтров для контроллера `CheckoutController` выглядит так: `:ensure_items_in_cart`, `:ensure_items_in_stock`, `:verify_open_shop`. Если хотя бы один из фильтров `ensure` вернет `false`, мы так и не узнаем, открыт магазин или нет, – цепочка фильтров будет прервана.

Можно передавать несколько аргументов-фильтров любого типа, а также фильтр-блок. Если передан блок, он трактуется как последний аргумент.

Around-фильтры

`Around`-фильтры обертывают действие, то есть выполняют некий код до и после действия. Их можно объявлять в виде ссылок на методы, блоков или объектов, отвечающих на вызов метода `filter` или оба метода `before` и `after`.

Чтобы использовать в качестве around-фильтра метод, передайте символ, именующий некоторый метод. Для выполнения этого метода внутри блока воспользуйтесь предложением yield (или block.call). В листинге 2.4 показан around-фильтр, протоколирующий исключения (я не хочу сказать, что вы должны делать нечто подобное в своем приложении; это просто пример).

Листинг 2.4. Around-фильтр для протоколирования исключений

```
around_filter :catch_exceptions

private

  def catch_exceptions
    yield
  rescue => exception
    logger.debug "Перехвачено исключение! #{exception}"
    raise
  end
```

Чтобы использовать в качестве around-фильтра блок, передайте блок, который в виде аргументов принимает контроллер и блок действия. Вызывать yield из блока around-фильтра напрямую нельзя – вместо этого явно вызовите блок действия:

```
around_filter do |controller, action|
  logger.debug "перед #{controller.action_name}"
  action.call
  logger.debug "после #{controller.action_name}"
end
```

Чтобы совместно с around-фильтром использовать фильтрующий объект, передайте объект, отвечающий на вызов метода :filter или на вызовы :before и :after. Из метода фильтрации передайте управление блоку следующим образом:

```
around_filter BenchmarkingFilter

class BenchmarkingFilter
  def self.filter(controller, &block)
    Benchmark.measure(&block)
  end
end
```

Фильтрующий объект с методами before и after обладает одной особенностью – вы должны явно вернуть true из метода before, если хотите вызвать метод after.

```
around_filter Authorizer

class Authorizer
  # Этот метод вызывается до действия. Возврат false отменяет действие.
  def before(controller)
    if user.authorized?
```

```
      return true
    else
      redirect_to login_url
      return false
    end
  end

  def after(controller)
    # Выполняется после действия, только если before вернул true
  end
end
```

Пропуск цепочки фильтров

Фильтр, объявленный в базовом классе, применяется ко всем подклассам. Это удобно, но иногда в подклассе необходимо пропустить фильтры, унаследованные от суперкласса:

```
class ApplicationController < ActionController::Base
  before_filter :authenticate
  around_filter :catch_exceptions
end

class SignupController < ApplicationController
  skip_before_filter :authenticate
end

class ProjectsController < ApplicationController
  skip_filter :catch_exceptions
end
```

Условная фильтрация

Применение фильтров можно ограничить определенными действиями. Для этого достаточно указать, какие действия включаются или исключаются. В обоих случаях можно задать как одиночное действие (например, `:only => :index`), так и массив действий (`:except => [:foo, :bar]`).

```
class Journal < ActionController::Base
  before_filter :authorize, :only => [:edit, :delete]

  around_filter :except => :index do |controller, action_block|
    results = Profiler.run(&action_block)
    controller.response.sub! "</body>", "#{results}</body>"
  end

  private
    def authorize
      # Переадресовать на login, если не аутентифицирован.
  end

end
```

Прерывание цепочки фильтров

Методы before_filter и around_filter могут прервать обработку запроса до выполнения действия контроллера. Это полезно, например, чтобы отказать в доступе неаутентифицированным пользователям.

Как уже отмечалось выше, для прерывания цепочки фильтров достаточно, чтобы фильтр вернул значение false. Вызов метода render или redirect_to также прерывает цепочку фильтров. Если цепочка фильтров прервана, то after-фильтры не выполняются. Around-фильтры прекращают обработку запроса, если не был вызван блок действия.

Если around-фильтр возвращает управление до вызова блока, то цепочка прерывается, и after-фильтры не вызываются.

Если before-фильтр возвращает false, вторая часть любого around-фильтра все равно выполняется, но сам метод действия не вызывается, равно как не вызываются и after-фильтры.

Потоковая отправка

Мало кто знает, что в Rails, помимо рендеринга шаблонов, встроена определенная поддержка потоковой отправки броузеру двоичного контента. Потоковая отправка удобна, когда необходимо послать броузеру динамически сгенерированный файл (например, изображение или PDF-файл). В модуле ActionController::Streaming module для этого предусмотрено два метода: send_data и send_file. Один из них полезен, вторым почти никогда не следует пользоваться. Сначала рассмотрим полезный метод.

send_data(data, options = {})

Метод send_data позволяет отправить пользователю текстовые или двоичные данные в виде именованного файла. Можно задать параметры, определяющие тип контента и видимое имя файла; указать, надо ли пытаться отобразить данные в броузере вместе с другим содержимым, или предложить пользователю загрузить их в виде вложения.

Параметры метода send_data

У метода send_data есть следующие параметры:

- :filename – задает имя файла, видимое броузеру;
- :type – задает тип контента HTTP. По умолчанию подразумевается 'application/octetstream';
- :disposition – определяет, следует ли отправлять файл в одном потоке с другими данными или загружать отдельно;
- :status – задает код состояния, сопровождающий ответ. По умолчанию принимается '200 OK'.

Примеры использования

Для загрузки динамически сгенерированного tgz-архива можно поступить следующим образом:

```
send_data generate_tgz('dir'), :filename => 'dir.tgz'
```

В листинге 2.5 приведен пример отправки броузеру динамически сгенерированного изображения; это часть реализации системы *captcha*, которая мешает злонамеренным роботам использовать ваше веб-приложение нежелательным образом.

Листинг 2.5. Контроллер Captcha, в котором используется библиотека RMagick и метод send_data

```
require 'RMagick'

class CaptchaController < ApplicationController

  def image
    # Создать холст RMagic и нарисовать на нем трудночитаемый текст
    ...
    image = canvas.flatten_images
    image.format = "JPG"

    # отправить броузеру
    send_data(image.to_blob, :disposition => 'inline',
                             :type => 'image/jpg')
  end
end
```

send_file(path, options = {})

Метод `send_file` отправляет клиенту файл порциями по 4096 байтов. В документации по API говорится: «Это позволяет не читать сразу весь

> **Замечание по поводу безопасности**
>
> Заметим, что метод `send_file` применим для чтения любого файла, доступного пользователю, от имени которого работает серверный процесс Rails. Поэтому очень тщательно проверяйте[1] параметр `path`, если его источником может быть не заслуживающая доверия веб-страница.

[1] Хейко Веберс (Heiko Webers) написал прекрасную статью о проверке имен файлов, которая доступна по адресу *http://www.rorsecurity.info/2007/03/27/working-with-files-in-rails/*.

файл в память и, следовательно, дает возможность отправлять очень большие файлы».

К сожалению, *это неправда*. Когда метод `send_file` используется в приложении Rails, работающем под управлением сервера Mongrel (а именно так большинство приложений сегодня и запускается), *весь файл считывается-таки в память*! Поэтому использование `send_file` для отправки больших файлов приведет только к большой головной боли. В следующем разделе обсуждается, как заставить веб-сервер отправлять файлы напрямую.

Параметры метода send_file

На случай, если вы все-таки решите воспользоваться методом `send_file` (и не говорите потом, что вас не предупреждали), приведу список его параметров:

- `:filename` – имя файла, видимое броузеру. По умолчанию принимается `File.basename(path)`;
- `:type` – тип контента HTTP. По умолчанию `'application/octet-stream'`;
- `:disposition` – отправлять ли файл в общем потоке или загружать отдельно;
- `:stream` – посылать ли файл пользовательскому агенту по мере считывания (true) или предварительно прочитать весь файл в память (`false`). По умолчанию `true`;
- `:buffer_size` – размер буфера (в байтах), используемого для потоковой отправки файла. По умолчанию `4096`;
- `:status` – код состояния, сопровождающий ответ. По умолчанию `'200 OK'`;
- `:url_based_filename` – должно быть true, если вы хотите, чтобы броузер вывел имя файла из URL; это необходимо для некоторых броузеров при использовании имен файлов, содержащих не-ASCII-символы (задание параметра :filename отменяет этот режим).

Большинство этих параметров обрабатывается закрытым методом `send_file_headers!` из модуля `ActionController::Streaming`, который и устанавливает соответствующие заголовки ответа. Поэтому если для отправки файлов вы используете веб-сервер, то, возможно, захотите взглянуть на исходный текст этого метода. Если вы пожелаете предоставить пользователю дополнительную информацию, которую Rails не поддерживает (например `Content-Description`), придется кое-что почитать о других HTTP заголовках Content-*[1].

[1] См. официальную спецификацию по адресу *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html*.

Говорит Кортенэ...

Мало найдется разумных причин обслуживать статические файлы с помощью Rails.

Очень, очень мало.

Если вам абсолютно необходимо воспользоваться одним из методов `send_data` или `send_file`, настоятельно рекомендую закэшировать файл перед отправкой. Сделать это можно несколькими способами (не забывайте, что правильно сконфигурированный веб-сервер сам обслуживает файлы в каталоге `public/` и не заходит в каталог `rails`).

Можно, например, просто скопировать файл в каталог `public`:

```
public_dir = File.join(RAILS_ROOT, 'public',
controller_path)
FileUtils.mkdir_p(public_dir)
FileUtils.cp(filename, File.join(public_dir,
filename))
```

Все последующие обращения к этому ресурсу будут обслуживаться самим веб-сервером.

Можно вместо этого попробовать воспользоваться директивой `caches_page`, которая автоматически сделает нечто подобное (кэширование рассматривается в главе 10).

Наконец, имейте в виду, что документ может кэшироваться прокси-сервером или броузером. Способом кэширования на промежуточных узлах управляют заголовки `Pragma` и `Cache-Control`. По умолчанию требуется, чтобы клиент запросил у сервера, можно ли возвращать кэшированный ответ[1].

Еще одна причина ненавидеть Internet Explorer

По умолчанию заголовки `Content-Type` и `Content-Disposition` устанавливаются так, чтобы поддержать загрузку произвольных двоичных файлов для максимально возможного числа броузеров. Но как будто специально чтобы подогреть ненависть к Internet Explorer, версии 4, 5, 5.5 и 6 этого богом проклятого броузера весьма специфически обрабатывают загрузку файлов, особенно по протоколу HTTPS.

[1] Обзор кэширования в Сети см. на странице *http://www.mnot.net/cache_docs/*.

Примеры использования

Для начала простейший пример загрузки ZIP-файла:

```
send_file '/path/to.zip'
```

Для отправки JPG-файла в потоке с другими данными требуется указать MIME-тип контента:

```
send_file '/path/to.jpg',
          :type => 'image/jpeg',
          :disposition => 'inline'
```

Следующий пример выведет в броузере HTML-страницу с кодом 404. Мы добавили в описание типа объявление кодировки с помощью параметра `charset`:

```
send_file '/path/to/404.html,
          :type => 'text/html; charset=utf-8',
          :status => 404
```

А как насчет потокой отправки FLV-файла флэш-плееру внутри броузера?

```
send_file @video_file.path,
          :filename => video_file.title + '.flv',
          :type => 'video/x-flv',
          :disposition => 'inline'
```

Как заставить сам веб-сервер отправлять файлы

Решение проблемы переполнения памяти, возникающей в связи с использованием метода `send_file`, заключается в том, чтобы воспользоваться средствами, которые такие веб-серверы, как Apache, Lighttpd и Nginx предлагают для прямой отправки файлов, даже если они не находятся в каталоге общедоступных документов. Для этого нужно задать в ответе специальный HTTP-заголовок, указав в нем путь к файлу, который веб-сервер должен отправить клиенту.

Вот как это делается для Apache и Lighttpd:

```
response.headers['X-Sendfile'] = path
```

А вот так – для Nginx:

```
response.headers['X-Accel-Redirect'] = path
```

В обоих случаях вы должны завершить действие контроллера, попросив Rails ничего посылать, поскольку этим займется веб-сервер.

```
render :nothing => true
```

В любом случае возникает вопрос, зачем *вообще* нужен механизм отправки файлов броузеру, если уже имеется готовый – запрашивать файлы из каталога `public`. Например, потому что часто встречаются

веб-приложения, которым необходимо возвращать файлы, защищенные от публичного доступа[1] (к числу таких приложений относятся практически все существующие порносайты).

Заключение

В этой главе мы рассмотрели некоторые основополагающие аспекты работы Rails: диспетчер и механизм рендеринга представлений контроллерами. Заодно мы обсудили фильтры действий контроллера; вы часто будете применять их для самых разных целей. API `ActionController` относится к числу наиболее фундаментальных концепций, которыми вы должны овладеть на пути к становлению экспертом по Rails.

Далее мы перейдем еще к одной теме, тесно связанной с диспетчерами и контроллерами: вопросу о том, как Rails решает, каким образом отвечать на запрос. То есть к системе маршрутизации.

[1] Бен Кэртис (Ben Curtis) описал замечательный подход к защищенной загрузке файлов в статье по адресу *http://www.bencurtis.com/archives/2006/11/serving-protected-downloads-with-rails/*.

3

Маршрутизация

Во сне я видел тысячи новых дорог....
Но проснулся и пошел по старой.

Китайская пословица

Под маршрутизацией в Rails понимается механизм, который на основе URL входящего запроса решает, какое действие должно предпринять приложение. Однако этим его функции отнюдь не ограничиваются. Система маршрутизации в Rails – крепкий орешек. Но за кажущейся сложностью не так уж много концепций. Стоит их усвоить – и все кусочки головоломки встают на свои места.

В этой главе мы познакомимся с основными приемами определения маршрутов и манипулирования ими, а в следующей рассмотрим предлагаемые Rails механизмы для поддержки написания приложений, согласующихся с принципами архитектурного стиля Representational State Transfer (REST). Эти механизмы могут оказаться исключительно полезными, даже если вы не собираетесь углубляться в теоретические дебри REST.

Многие примеры, приведенные в этих двух главах, основаны на небольшом аукционном приложении. Они достаточно просты и понятны. Идея такова: есть аукционы, на каждом из которых торгуется один лот. Кроме того, есть пользователи, предлагающие заявки со своими ценами. Вот по существу и все.

Главное событие в жизненном цикле соединения с приложением Rails – срабатывание какого-либо действия контроллера. Следовательно, кри-

тически важна процедура определения того, *какой* контроллер и *какое* действие выбрать. Вот эта-то процедура и составляет сущность системы маршрутизации.

Система маршрутизации отображает URL на действия. Для этого применяются правила, которые вы задаете с помощью команд на языке Ruby в конфигурационном файле `config/routes.rb`. Если не переопределять правила, прописанные в этом файле по умолчанию, вы получите некое разумное поведение. Но не так уж трудно написать собственные правила и обратить гибкость системы маршрутизации себе во благо.

На самом деле у системы маршрутизации две задачи. Она отображает запросы на действия и конструирует URL, которые вы можете передавать в качестве аргументов таким методам, как `link_to`, `redirect_to` и `form_tag`. Система знает, как преобразовать URL, заданный посетителем, в последовательность контроллер/действие. Кроме того, она знает, как изготовить представляющие URL строки по вашим спецификациям.

Когда вы пишете такой код:

```
<%= link_to "Лоты", :controller => "items", :action => "list" %>
```

система маршрутизации передает помощнику `link_to` следующий URL:

```
http://localhost:3000/items/list
```

Таким образом, система маршрутизации – мощный двусторонний механизм. Она *распознает* URL и маршрутизирует их соответствующим образом, а также генерирует URL, используя в качестве шаблона правила маршрутизации. По мере изложения мы будем обращать внимание на обе эти одинаково важные цели.

Две задачи маршрутизации

Механизм распознавания URL важен, потому что именно он позволяет приложению решить, что делать с поступившим запросом:

```
http://localhost:3000/myrecipes/apples Что нам делать?!
```

Генерация URL полезна, так как позволяет в шаблонах представлений и контроллерах применять синтаксис сравнительно высокого уровня для вставки URL. Вам не придется писать такой код:

```
<a href="http://localhost:3000/myrecipes/apples">Мои рецепты блюд из яблок</
a>
Не хочется писать такое вручную!
```

Система маршрутизации решает обе задачи: интерпретацию (распознавание) URL запроса и запись (генерирование) URL. Они основаны на формулируемых вами правилах. Эти правила вставляются в файл `config/routes.rb` с применением особого синтаксиса (это обычный код на Ruby, но в нем используются специальные методы и параметры).

Каждое правило – или, применяя общеупотребительный термин, *маршрут* – включает строку-образец, которая служит шаблоном как для сопоставления с URL, так и для их порождения. Образец состоит из статических подстрок, символов косой черты (имитирующих синтаксис URL) и позиционных метапараметров, служащих «приемниками» для отдельных компонентов URL как при распознавании, так и при генерации.

Маршрут может также включать один или несколько связанных параметров в форме пар «ключ/значение» из некоторого хеша. Судьба этих пар зависит от того, что собой представляет ключ. Есть два «волшебных» ключа (`:controller` и `:action`), которые определяют, что нужно сделать. Остальные ключи (`:blah`, `:whatever` и т. д.) сохраняются для ссылок в будущем. Вот конкретный маршрут, связанный с предыдущими примерами:

```
map.connect 'myrecipes/:ingredient',
            :controller => "recipes",
            :action => "show"
```

В этом примере вы видите:

- статическую строку (`myrecipes`);
- метапараметр, соответствующий компоненту URL (`:ingredient`);
- связанные параметры (`:controller => "recipes", :action => "show"`).

Синтаксис маршрутов достаточно развит – этот пример вовсе не является самым сложным (как, впрочем, и самым простым), – поскольку на них возлагается много обязанностей. Один-единственный маршрут типа приведенного выше должен содержать достаточно информации как для сопоставления с существующим URL, так и для изготовления нового. Синтаксис маршрутов разработан с учетом обеих процедур.

Разобраться в нем не так уж сложно, если рассмотреть разные типы полей по очереди. Мы займемся этим на примере маршрута для «ингредиентов». Не расстраивайтесь, если сначала не все будет понятно. На протяжении этой главы мы обсудим различные приемы и раскроем все тайны.

Изучая анатомию маршрутов, мы познакомимся с ролью, которую каждая часть играет в распознавании и генерации URL. Не забывайте, что это всего лишь демонстрационный пример. Маршруты позволяют многое, но, чтобы понять, как все это работает, лучше начать с простого.

Связанные параметры

На этапе распознавания связанные параметры – пары «ключ/значение» в хеше, задаваемом в конце списка аргументов маршрута, – определяют, что должно происходить, если данный маршрут соответствует поступившему URL. Предположим, что в броузере был введен такой URL:

```
http://localhost:3000/myrecipes/apples
```

Этот URL соответствует маршруту для ингредиентов. В результате будет выполнено действие `show` контроллера `recipes`. Чтобы понять причину, снова рассмотрим наш маршрут:

```
map.connect 'myrecipes/:ingredient',
:controller => "recipes",
:action => "show"
```

Ключи `:controller` и `:action` – ***связанные***: если URL сопоставился с этим маршрутом, то запрос всегда будет обрабатываться именно данным контроллером и данным действием. Ниже вы познакомитесь с техникой определения контроллера и действия на основе сопоставления с *метапараметрами*. Однако в этом примере метапараметры не участвуют. Контроллер и действие «зашиты» в код.

Желая сгенерировать URL, вы должны предоставить значения всех необходимых связанных параметров. Тогда система маршрутизации сможет отыскать нужный маршрут (если передано недостаточно информации для формирования маршрута, Rails возбудит исключение).

Параметры обычно представляются в виде хеша. Например, чтобы сгенерировать URL из маршрута для ингредиентов, нужно написать примерно такой код:

```
<%= link_to " Мои рецепты блюд из яблок",
     :controller => "recipes",
     :action => "show",
     :ingredient => "apples" %>
```

Значения `"recipes"` и `"show"` для параметров `:controller` и `:action` сопоставляются с маршрутом для ингредиентов, в котором значения этих параметров постоянны. Значит строка-образец, указанная в данном маршруте, может служить шаблоном генерируемого URL.

Применение хеша для задания компонентов URL – общая техника всех методов порождения URL (`link_to`, `redirect_to`, `form_for` и т. д.). Внутри они обращаются к низкоуровневому методу `url_for`, о котором мы поговорим чуть ниже.

Мы пока ни слова не сказали о компоненте `:ingredient`. Это метапараметр в строке-образце.

Метапараметры («приемники»)

Символ :ingredient в рассматриваемом маршруте называется метапараметром (wildcard parameter), или переменной. Можете считать его *приемником*; он должен быть заменен неким значением. Значение, подставляемое вместо метапараметра, определяется позиционно в ходе сопоставления URL с образцом:

```
http://localhost:3000/myrecipes/apples   Кто-то запрашивает данный URL...
                      'myrecipes/:ingredient'   который соответствует
                                                этому образцу
```

В данном случае приемник :ingredient получает из URL значение apples. Следовательно, в элемент хеша params[:ingredient] будет записана строка "apples". К этому элементу можно обратиться из действия recipes/show. При генерации URL необходимо предоставить значения для всех приемников – метапараметров в строке-образце. Для этого применяется синтаксис «ключ => значение». В этом и состоит смысл последней строки в предшествующем примере:

```
<%= link_to "Мои рецепты блюд из яблок",
      :controller => "recipes",
      :action => "show",
      :ingredient => "apples" %>
```

В данном обращении к методу link_to мы задали значения трех параметров. Два из них должны соответствовать зашитым в код связанным параметрам маршрута; третий, :ingredient, будет подставлен в образец вместо метапараметра :ingredient.

Но все они – не более чем пары «ключ/значение». Из обращения к link_to не видно, передаются ли «зашитые» или подставляемые значения. Известно лишь, что есть три значения, связанные с тремя ключами, и этого должно быть достаточно для идентификации маршрута, а, стало быть, строки-образца, а, стало быть, шаблона URL.

Статические строки

В нашем примере маршрута образец содержит статическую строку myrecipes.

```
map.connect 'myrecipes/:ingredient',
            :controller => "myrecipes",
            :action => "show"
```

Эта строка служит отправной точкой для процедуры распознавания. Когда система маршрутизации видит URL, начинающийся с /myrecipes, она сопоставляет его со статической строкой в маршруте для ингредиентов. Любой URL, который не содержит строку myrecipes в начале, не будет сопоставлен с этим маршрутом.

При генерации URL статические строки просто копируются в URL, формируемый системой маршрутизации. Следовательно, в рассматриваемом примере такой вызов `link_to`:

```
<%= link_to "Мои рецепты блюд из яблок",
      :controller => "recipes",
      :action => "show",
      :ingredient => "apples" %>
```

породит следующий HTML-код:

```
<a href="http://localhost:3000/myrecipes/apples">Мои рецепты блюд из яблок</
a>
```

Строка `myrecipes` при вызове `link_to` не указывается. Сопоставление с маршрутом основывается на *параметрах*, переданных `link_to`. Затем генератор URL использует заданный в маршруте образец как шаблон для порождения URL. А уже в этом образце присутствует подстрока `myrecipes`.

Распознавание и *генерация* URL – две задачи, решаемые системой маршрутизации. Можно провести аналогию с адресной книгой, хранящейся в мобильном телефоне. Когда вы выбираете из списка контактов имя Гэвин, телефон находит соответствующий номер. А когда Гэвин звонит вам, телефон просматривает все номера в адресной книге и определяет, что вызывающий номер принадлежит именно Гэвину; в результате на экране высвечивается имя Гэвин.

Маршрутизация в Rails несколько сложнее поиска в адресной книге, поскольку в ней участвуют переменные. Отображение не взаимно однозначно, но идея та же самая: распознать, что пришло в запросе, и сгенерировать выходную HTML-разметку.

Теперь обратимся к правилам маршрутизации. Читая текст, вы должны все время держать в уме двойственную природу распознавания/генерации. Вот два принципа, которые особенно полезно запомнить:

- и распознавание, и генерация управляются *одним и тем же правилом*. Вся система построена так, чтобы вам не приходилось записывать правила дважды. Каждое правило пишется один раз и применяется в обоих направлениях;
- URL, генерируемые системой маршрутизации (с помощью метода `link_to` и родственных ему), *имеют смысл только для самой системы маршрутизации*. Путь `recipes/apples`, который генерирует система, не содержит никакой информации о том, как будет происходить обработка, – он лишь отображается на некоторое правило маршрутизации. Именно правило предоставляет информацию, необходимую для вызова определенного действия контроллера. Не зная правил маршрутизации, невозможно понять, что означает данный URL.

Как это выглядит на практике, мы детально рассмотрим по ходу обсуждения.

Файл routes.rb

Маршруты определяются в файле `config/routes.rb`, как показано в листинге 3.1 (с некоторыми дополнительными комментариями). Этот файл создается в момент первоначального создания приложения Rails. В нем уже прописано несколько маршрутов, и в большинстве случаев вам не придется ни изменять их, ни добавлять новые.

Листинг 3.1. Файл routes.rb, подразумеваемый по умолчанию

```
ActionController::Routing::Routes.draw do |map|
  # Приоритет зависит от порядка следования.
  # Чем раньше определен маршрут, тем выше его приоритет.
  # Пример простого маршрута:
  # map.connect 'products/:id', :controller => 'catalog',
                                :action => 'view'

  # Помните, что значения можно присваивать не только параметрам
  # :controller и :action
  # Пример именованного маршрута:
  # map.purchase 'products/:id/purchase', :controller => 'catalog',
                                          :action => 'purchase'

  # Этот маршрут можно вызвать как purchase_url(:id => product.id)
  # Вы можете задать маршрут к корню сайта, указав значение ''
  # -- не забудьте только удалить файл public/index.html.
  # map.connect '', :controller => "welcome"

  # Разрешить загрузку WSDL-документа веб-службы в виде файла
  # с расширением 'wsdl', а не фиксированного файла с именем 'wsdl'
  map.connect ':controller/service.wsdl', :action => 'wsdl'

  # Установить маршрут по умолчанию с самым низким приоритетом.
  map.connect ':controller/:action/:id.:format'
  map.connect ':controller/:action/:id'
end
```

Код состоит из единственного вызова метода `ActionController::Routing::Routes.draw`, который принимает блок. Все, начиная со второй и кончая предпоследней строкой, тело этого блока.

Внутри блока есть доступ к переменной `map`. Это экземпляр класса `ActionController::Routing::RouteSet::Mapper`. С его помощью конфигурируется вся система маршрутизации в Rails – правила маршрутизации определяются вызовом методов объекта `Mapper`. В подразумеваемом по умолчанию файле `routes.rb` встречается несколько обращений к методу `map.connect`. Каждое обращение (по крайней мере, незакомментированное) создает новый маршрут и регистрирует его в системе маршрутизации.

Система маршрутизации должна найти, какому образцу соответствует распознаваемый URL, или провести сопоставление с параметрами ге-

нерируемого URL. Для этого все правила – маршруты – обходятся в порядке их определения, то есть следования в файле routes.rb. Если сопоставить с очередным маршрутом не удалось, процедура сопоставления переходит к следующему. Как только будет найден подходящий маршрут, поиск завершается.

> **Говорит Кортенэ...**
>
> Маршрутизация – пожалуй, один из самых сложных аспектов Rails. Долгое время вообще был только один человек, способный вносить изменения в исходный код этой подсистемы, настолько она запутана. Поэтому не расстраивайтесь, если не сможете ухватить ее суть с первого раза. Так было с большинством из нас.
>
> Но при этом синтаксис файла routes.rb довольно прямолинеен. На задание маршрутов в типичном проекте Rails у вас вряд ли уйдет больше пяти минут.

Маршрут по умолчанию

В самом конце файла routes.rb находится *маршрут по умолчанию*:

```
map.connect ':controller/:action/:id'
```

Маршрут по умолчанию – это в некотором роде конец пути; он определяет, что должно произойти, когда больше ничего не происходит. Однако это еще и неплохая отправная точка. Если вы понимаете, что такое маршрут по умолчанию, то сможете разобраться и в более сложных примерах.

Маршрут по умолчанию состоит из строки-образца, содержащей три метапараметра-приемника. Два из них называются :controller и :action. Следовательно, действие, определяемое этим маршрутом, зависит исключительно от метапараметров; нет ни связанных параметров, ни параметров, зашитых в код контроллера и действия.

Рассмотрим следующий сценарий. Поступает запрос на такой URL:

```
http://localhost:3000/auctions/show/1
```

Предположим, что он не соответствует никакому другому образцу. Тогда поиск доходит до последнего маршрута в файле – маршрута по умолчанию. В этом маршруте есть три приемника, а в URL – три значения, поэтому складывается три позиционных соответствия:

```
:controller/:action/:id
 auctions / show / 1
```

Таким образом, мы получили контроллер auctions, действие show и значение «1» для параметра id (которое должно быть сохранено в params[:id]). Теперь диспетчер знает, что делать.

Поведение маршрута по умолчанию иллюстрирует некоторые особенности системы маршрутизации. Например, по умолчанию для любого запроса в качестве действия подразумевается `index`. Другой пример: если в образце есть метапараметр, например `:id`, то система маршрутизации предпочитает найти для него значение, но если такового в URL не оказывается, она присвоит ему значение *nil*, а не придет к выводу, что соответствие не найдено.

В табл. 3.1 приведены примеры нескольких URL и показаны результаты применения к ним этого правила.

Таблица 3.1. Примеры применения маршрута по умолчанию

URL	Результат		Значение id
	Контроллер	**Действие**	
`/auctions/show/3`	`auctions`	`show`	**3**
`/auctions/index`	`auctions`	`index`	***nil***
`/auctions`	`auctions`	`index` **(по умолчанию)**	*`nil`*
`/auctions/show`	`auctions`	`show`	*`nil`* – возможно, ошибка!

В последнем случае *nil*, вероятно, ошибка, так как действия `show` без идентификатора, скорее всего, быть не должно!

О поле :id

Отметим, что в обработке поля `:id` этого URL нет ничего магического; оно трактуется просто как значение с именем. При желании можно было бы изменить правило, изменив `:id` на `:blah`, но тогда надо не забыть внести соответствующее изменение в действие контроллера:

```
@auction = Auction.find(params[:blah])
```

Имя `:id` выбрано исходя из общепринятого соглашения. Оно отражает тот факт, что действию часто нужно получать конкретную запись из базы данных. Основная задача маршрутизатора – определить контроллер и действие, которые надо вызвать. Поле `id` – это дополнение, которое позволяет действиям передавать друг другу данные.

Поле `id` в конечном итоге попадает в хеш `params`, доступный всем действиям контроллера. В типичном, классическом случае его значение используется для выборки записи из базы данных:

```
class ItemsController < ApplicationController
  def show
    @item = Item.find(params[:id])
  end
end
```

Генерация маршрута по умолчанию

Маршрут по умолчанию не только лежит в основе распознавания URL и выбора правильного поведения, но и играет определенную роль при генерации URL. Вот пример обращения к методу `link_to`, в котором для генерации URL применяется маршрут по умолчанию:

```
<%= link_to item.description,
    :controller => "item",
    :action => "show",
    :id => item.id %>
```

Здесь предполагается, что существует локальная переменная `item`, содержащая (опять же предположительно) объект `Item`. Идея в том, чтобы создать гиперссылку на действие `show` контроллера `item` и включить в нее идентификатор `id` данного лота. Иными словами, гиперссылка должна выглядеть следующим образом:

```
<a href="localhost:3000/item/show/3">Фотография Гудини с автографом</a>
```

Именно такой URL любезно создает механизм генерации маршрутов. Взгляните еще раз на маршрут по умолчанию:

```
map.connect ':controller/:action/:id'
```

При вызове метода `link_to` мы задали значения всех трех полей, присутствующих в образце. Системе маршрутизации осталось лишь подставить эти значения и включить результирующую строку в URL:

```
item/show/3
```

При щелчке по этой ссылке ее URL будет распознан благодаря второй половине системы маршрутизации, что вызовется нужное действие подходящего контроллера, которому в элементе `params[:id]` будет передано значение 3.

В данном примере при генерации URL используется логика подстановки метапараметров: в образце указываются три символа — `:controller`, `:action`,`:id`, вместо них в генерируемый URL подставляются значения, которые мы передали. Сравните с предыдущим примером:

```
map.connect 'recipes/:ingredient',
            :controller => "recipes",
            :action => "show"
```

Чтобы заставить генератор выбрать именно этот маршрут, вы должны при обращении к `link_to` передать строки `recipes` и `show` в качестве значений параметров `:controller` и `:action`. В случае маршрута по умолчанию, да и вообще любого маршрута, образец которого содержит символы, соответствие все равно должно быть найдено, но значения могут быть любыми.

Модификация маршрута по умолчанию

Чтобы прочувствовать систему маршрутизации, лучше всего попробовать что-то изменить и посмотреть, что получится. Проделаем это с маршрутом по умолчанию. Потом надо будет вернуть все в исходное состояние, но модификация кое-чему вас научит.

Давайте поменяем местами символы `:controller` и `:action` в образце:

```
# Установить маршрут по умолчанию с самым низким приоритетом.
map.connect ':action/:controller/:id'
```

Теперь в маршруте по умолчанию первым указывается действие. Это означает, что URL, который раньше записывался в виде *http://localhost:3000/auctions/show/3*, теперь должен выглядеть как *http://localhost:3000/show/auctions/3*. А при генерации URL из этого маршрута мы будем получать результат в порядке /show/auctions/3.

Это не очень логично – предыдущий маршрут по умолчанию был лучше. Зато вы стали лучше понимать, что происходит, особенно в части магических символов `:controller` и `:action`. Попробуйте еще какие-нибудь изменения и посмотрите, какой эффект они дадут (и не забудьте вернуть все назад).

Предпоследний маршрут и метод respond_to

Сразу перед маршрутом по умолчанию находится маршрут:

```
map.connect ':controller/:action/:id.:format'
```

Строка `.:format` в конце сопоставляется с точкой и значением метапараметра `format` после поля `id`. Следовательно, этот маршрут соответствует, например, такому URL:

```
http://localhost:3000/recipe/show/3.xml
```

Здесь в элемент хеша `params[:format]` будет записано значение `xml`. Поле `:format` – особое; оно интепретируется специальным образом в действии контроллера. И связано это с методом `respond_to`.

Метод `respond_to` позволяет закодировать действие так, что оно будет возвращать разные результаты в зависимости от запрошенного формата. Вот пример действия `show` в контроллере `items`, которое возвращает результат в формате HTML или XML:

```
def show
  @item = Item.find(params[:id])
  respond_to do |format|
    format.html
```

```
      format.xml { render :xml => @item.to_xml }
    end
  end
```

Здесь в блоке `respond_to` есть две ветви. Ветвь HTML состоит из предложения `format.html`. Запрос HTML-данных будет обработан путем обычного рендеринга представления RHTML. Ветвь XML включает блок кода. При запросе XML-данных этот блок будет выполнен, а результаты выполнения возвратятся клиенту.

Проиллюстрируем это, вызвав программу `wget` из командной строки (выдача слегка сокращена):

```
$ wget http://localhost:3000/items/show/3.xml -O -
Resolving localhost... 127.0.0.1, ::1
Connecting to localhost|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 295 [application/xml]
<item>
 <created-at type="datetime">2007-02-16T04:33:00-05:00</created-at>
 <description>Violin treatise</description>
 <id type="integer">3</id>
 <maker>Leopold Mozart</maker>
 <medium>paper</medium>
 <modified-at type="datetime"></modified-at>
 <year type="integer">1744</year>
</item>
```

Суффикс `.xml` в конце URL заставляет метод `respond_to` пойти по ветви xml и вернуть XML-представление лота.

Метод respond_to и заголовок HTTP-Accept

Вызвать ветвление в методе `respond_to` может также заголовок `HTTP-Accept` в запросе. В этом случае нет необходимости добавлять в URL часть `.:format`.

В следующем примере мы не задаем в `wget` суффикс `.xml`, а устанавливаем заголовок `Accept`:

```
wget http://localhost:3000/items/show/3 -O - –header="Accept:
text/xml"
Resolving localhost... 127.0.0.1, ::1
Connecting to localhost|127.0.0.1|:3000... connected.
HTTP request sent, awaiting response...
200 OK
Length: 295 [application/xml]
<item>
 <created-at type="datetime">2007-02-16T04:33:00-05:00</created-at>
 <description>Violin treatise</description>
 <id type="integer">3</id>
 <maker>Leopold Mozart</maker>
 <medium>paper</medium>
```

```
  <modified-at type="datetime"></modified-at>
  <year type="integer">1744</year>
</item>
```

Результат получился точно такой же, как в предыдущем примере.

Пустой маршрут

Если нет желания учиться на собственном опыте, можно вообще не трогать маршрут по умолчанию. Но в файле `routes.rb` есть еще один маршрут, который тоже в какой-то мере считается умалчиваемым, и вот его-то вы, скорее всего, захотите изменить. Речь идет о пустом маршруте.

В нескольких строках от маршрута по умолчанию (см. листинг 3.1) вы обнаружите такой фрагмент:

```
# Вы можете задать маршрут к корню сайта, указав значение ''
# -- не забудьте только удалить файл public/index.html.
# map.connect '', :controller => "welcome"
```

То, что вы видите, и называется пустым маршрутом; это правило говорит о том, что должно произойти, если кто-то наберет следующий URL:

```
http://localhost:3000   Отметьте отсутствие "/anything" в конце!
```

Пустой маршрут – в определенном смысле противоположность маршруту по умолчанию. Если маршрут по умолчанию говорит: «Мне нужно три значения, и я буду интерпретировать их как контроллер, действие и идентификатор», то пустой маршрут говорит: «Мне не нужны *никакие* значения; я вообще *ничего* не хочу, я уже знаю, какой контроллер и действие вызывать!»

В только что сгенерированном файле `routes.rb` пустой маршрут закомментирован, поскольку для него нет универсального или хотя бы разумного значения по умолчанию. Вы сами должны решить, что в вашем приложении означает «пустой» URL.

Ниже приведено несколько типичных примеров правил для пустых маршрутов:

```
map.connect '', :controller => "main", :action => "welcome"
map.connect '', :controller => "top", :action => "login"
map.connect '', :controller => "main"
```

Последний маршрут ведет к действию `main/index`, поскольку действие `index` подразумевается по умолчанию, когда никакое другое не указано.

Отметим, что в Rails 2.0 в объект `map` добавлен метод `root`, поэтому теперь определять пустой маршрут для приложения Rails рекомендуется так:

```
map.root :controller => "homepage"
```

Наличие пустого маршрута позволяет пользователям что-то увидеть, когда они заходят на ваш сайт, указав лишь его доменное имя.

Самостоятельное создание маршрутов

Маршрут по умолчанию является общим. Он предназначен для перехвата всех маршрутов, которым не нашлось более точного соответствия выше. А теперь займемся этим самым «*выше*», то есть маршрутами, которые в файле `routes.rb` предшествуют маршруту по умолчанию.

Вы уже знакомы с основными компонентами маршрута: статическими строками, связанными параметрами (как правило, в их число входит `:controller`, а часто еще и `:action`) и метапараметрами-приемниками, которым значения присваиваются позиционно из URL или на основе ключей, заданных в хеше, определяющем URL.

При создании новых маршрутов вы должны рассуждать так же, как система маршрутизации:

- для распознавания необходимо, чтобы в маршруте было достаточно информации – либо зашитой в код, либо готовой для приема значений из URL, – чтобы состоялся выбор контроллера и действия (по крайней мере, контроллера – по умолчанию будет выбрано действие `index`, если вас это устраивает);
- для генерации необходимо, чтобы зашитых параметров и метапараметров было достаточно для создания нужного маршрута.

Коль скоро эти условия соблюдены и маршруты перечислены в порядке убывания приоритетов (в порядке «проваливания»), система будет работать должным образом.

Использование статических строк

Помните – тот факт, что для выполнения любого запроса нужен контроллер и действие, еще не означает, что необходимо точное соответствия между количеством полей в строке-образце и количеством связанных параметров.

Например, *допустимо* написать такой маршрут:

```
map.connect ":id", :controller => "auctions", :action => "show"
```

и он будет распознавать следующий URL:

```
http://localhost:3000/8
```

Система маршрутизации запишет 8 в `params[:id]` (исходя из позиции приемника `:id`, который соответствует позиции «8» в URL) и выполнит действие `show` контроллера `auctions`. Разумеется, визуально такой маршрут воспринимается странно. Лучше поступить примерно так, как в листинге 2.2, где семантика выражена более отчетливо:

```
map.connect "auctions/:id", :controller => "auctions", :action => "show"
```

Такой маршрут распознал бы следующий URL:

```
http://localhost:3000/auctions/8
```

Здесь `auctions` – статическая строка. Система будет искать ее в распознаваемом URL и вставлять в URL, который генерируется следующим кодом:

```
<%= link_to "Информация об аукционе",
    :controller => "auctions",
    :action => "show",
    :id => auction.id %>
```

Использование собственных «приемников»

До сих пор нам встречались магические параметры `:controller` и `:action` и хоть и не магический, но стандартный параметр `:id`. Можно также завести собственные параметры – зашитые или мета. Тогда и ваши маршруты и код приложения окажутся более выразительными и самодокументированными.

Основная причина для заведения собственных параметров состоит в том, чтобы ссылаться на них из программы. Например, вы хотите, чтобы действие контроллера выглядело следующим образом:

```
def show
  @auction = Auction.find(params[:id])
  @user = User.find(params[:user_id])
end
```

Здесь символ `:user_id`, как и `:id`, выступает в роли ключа хеша. Но, значит, он должен как-то туда попасть. А попадает он точно так же, как параметр `:id` – вследствие указания в маршруте, по которому мы добрались до действия `show`.

Вот как выглядит этот маршрут:

```
map.connect 'auctions/:user_id/:id',
    :controller => "auctions",
    :action => "show"
```

При распознавании URL

```
/auctions/3/1
```

этот маршрут вызовет действие `auctions/show` и установит в хеше `params` оба ключа – `:user_id` и `:id` (при позиционном сопоставлении `:user_id` получает значение 3, а `:id` – значение 1).

Для генерации URL достаточно добавить ключ `:user_id` в спецификацию URL:

```
<%= link_to "Аукцион",
```

```
:controller => "auctions",
:action => "show",
:user_id => current_user.id,
:id => ts.id %>
```

Ключ `:user_id` в хеше сопоставится с приемником `:user_id` в образце маршрута. Ключ `:id` также сопоставится, равно как и параметры `:controller` и `:action`. Результатом будет URL, сконструированный по шаблону `auctions/:user_id/:id`.

В хеш, описывающий URL, при вызове `link_to` и родственных методов можно поместить много спецификаторов. Если какой-то параметр не найдется в правиле маршрутизации, он будет добавлен в строку запроса генерируемого URL. Например, если добавить

```
:some_other_thing => "blah"
```

в хеш, передаваемый методу `link_to` в примере выше, то получится такой URL:

```
http://localhost:3000/auctions/3/1?some_other_thing=blah
```

Замечание о порядке маршрутов

И при распознавании, и при генерации маршруты перебираются в том порядке, в котором они определены в файле `routes.rb`. Перебор завершается при обнаружении первого соответствия, поэтому следует остерегаться ложных срабатываний.

Предположим, например, что в файле `routes.rb` есть два следующих маршрута:

```
map.connect "users/help", :controller => "users"
map.connect ":controller/help", :controller => "main"
```

Если пользователь зайдет на URL `/users/help`, то справку выдаст действие `users/help`, а если на URL `/any_other_controller/help`, — сработает действие `help` контроллера `main`. Согласен, нетривиально.

А теперь посмотрим, что случится, если поменять эти маршруты местами:

```
map.connect ":controller/help", :controller => "main"
map.connect "users/help", :controller => "users"
```

Если пользователь заходит на `/users/help`, то сопоставляется первый маршрут, поскольку более специализированный маршрут, в котором часть `users` обрабатывается по-другому, определен в файле ниже.

Тут есть прямая аналогия с другими операциями сопоставления, например с предложением `case`:

```
case string
when /./
  puts "Сопоставляется с любым символом!"
```

```
when /x/
  puts "Сопоставляется с 'x'!"
end
```

Во вторую ветвь `when` мы никогда не попадем, потому что строка `'x'` будет сопоставлена в первой ветви. Необходимо всегда сначала располагать частные, а потом – общие случаи:

```
case string
when /x/
  puts "Сопоставляется с 'x'!"
when /./
  puts "Сопоставляется с любым символом!"
end
```

В примерах предложений `case` мы воспользовались регулярными выражениями – `/x/` и т. д. – для записи образцов, с которыми сопоставляется строка. Регулярные выражения встречаются и в синтаксисе маршрутов.

Применение регулярных выражений в маршрутах

Иногда требуется не просто распознать маршрут, а извлечь из него более детальную информацию, чем позволяют компоненты и поля. Можно воспользоваться регулярными выражениями[1].

Например, можно маршрутизировать все запросы show на действие `error`, если поле `id` не числовое. Для этого следует создать два маршрута: один – для числовых идентификаторов, а другой – для всех остальных:

```
map.connect ':controller/show/:id',
  :id => /\d+/, :action => "show"

map.connect ':controller/show/:id',
  :action => "alt_show"
```

Если хотите (в основном ради понятности), можете обернуть ограничения, записанные в терминах регулярных выражений, в специальный хеш параметров с именем `:requirements`:

```
map.connect ':controller/show/:id',
  :action => "show", :requirements => { :id => /\d+/ }
```

Регулярные выражения в маршрутах могут быть полезны, особенно если существуют маршруты, отличающиеся *только* образцами компонентов. Но это нельзя считать полноценной заменой контролю целостности данных. URL, сопоставившийся с маршрутом, в котором

[1] Дополнительную информацию о регулярных выражениях см. в книге *Хэла Фултона* (Hal Fulton) *The Ruby Way*, опубликованной в этой же серии.

есть регулярные выражения, можно уподобить кандидату, прошедшему интервью первой ступени. Необходимо еще убедиться, что поступившее значение допустимо с точки зрения предметной области приложения.

Параметры по умолчанию и метод url_for

Методы генерации URL, которыми вы, скорее всего, будете пользоваться, – `link_to`, `redirect_to` и им подобные – на самом деле являются обертками низкоуровневого метода `url_for`. Но метод `url_for` заслуживает рассмотрения и сам по себе, поскольку это позволит кое-что узнать о генерировании URL в Rails (а, возможно, вам когда-нибудь захочется вызвать `url_for` напрямую).

Метод `url_for` предназначен для генерации URL по вашим спецификациям с учетом правил в сопоставившемся маршруте. Этот метод не выносит пустоты: при генерации URL он пытается заполнить максимально возможное число полей, а если не может найти для конкретного поля значение в переданном вами хеше, то ищет его в параметрах текущего запроса.

Другими словами, столкнувшись с отсутствием значений для каких-то частей URL, `url_for` по умолчанию использует текущие значения `:controller`, `:action` и, если нужно, других параметров, необходимых маршруту.

Это означает, что можно не повторять задание одной и той же информации, если вы находитесь в пределах одного контроллера. Например, внутри представления `show` для шаблона, принадлежащего контроллеру `auctions`, можно было бы создать ссылку на действие `edit` следующим образом:

```
<%= link_to "Редактировать аукцион", :action => "edit", :id => @auction.id %>
```

В предположении, что рендеринг этого представления выполняют только действия контроллера `auctions`, текущим контроллером на этапе рендеринга всегда будет `auctions`. Поскольку в хеше для построения URL нет ключа `:controller`, генератор автоматически выберет `auctions`, и после подстановки в маршрут по умолчанию (`:controller/:action/:id`) получится следующий URL (для аукциона 5):

```
<a href="http://localhost:3000/auctions/edit/5">Редактировать аукцион</a>
```

То же справедливо и в отношении действий. Если не передавать ключ `:action`, будет подставлено текущее действие. Однако имейте в виду, что довольно часто одно действие выполняет рендеринг шаблона, принадлежащего другому действию. Поэтому выбор текущего действия по умолчанию встречается реже, чем выбор текущего контроллера.

Что случилось с :id

Отметим, что в предыдущем примере мы выбрали `:controller` по умолчанию, но были вынуждены явно задать значение для `:id`. Объясняется это тем, как работает механизм выбора умолчаний в методе `url_for`. Генератор маршрутов просматривает сегменты шаблона URL слева направо, а шаблон в нашем случае выглядит так:

```
:controller/:action/:id
```

Поля заполняются параметрами из текущего запроса до тех пор, пока не встретится поле, для которого явно задано значение:

```
:controller/:action/:id
по умолчанию!       задано!
```

Встретив поле, для которого вы задали значение, генератор проверяет, совпадает ли это значение с тем, которое он все равно использовал бы по умолчанию. Поскольку в нашем примере задействуется шаблон `show`, а ссылка ведет на действие `edit`, то для поля `:action` передано не то значение, которое было бы выбрано по умолчанию.

Обнаружив значение, отличающееся от умалчиваемого, метод `url_for` вообще прекращает использовать умолчания. Он решает, что раз уж вы один раз отошли от умолчаний, то и в дальнейшем к ним не вернетесь, – первое поле со значением, отличным от умалчиваемого, и *все поля справа от него* не получают значений из текущего запроса.

Именно поэтому мы задали для `:id` конкретное значение, хотя оно вполне могло бы совпадать со значением `params[:id]`, оставшемся от предыдущего запроса.

Контрольный вопрос: что произойдет, если данный маршрут сделать маршрутом по умолчанию

```
map.connect ':controller/:id/:action'
```

а потом произвести следующие изменения в шаблоне `show.rhtml`:

```
<%= link_to "Редактировать аукцион", :action => "edit" %>
```

Ответ: поскольку `:id` теперь находится не справа, а слева от `:action`, генератор с радостью заполнит поля `:controller` и `:id` значениями из текущего запроса. Затем вместо `:action` он подставит строку `"edit"`, поскольку мы зашили ее в шаблон. Справа от `:action` ничего не осталось, следовательно, все уже сделано.

Поэтому, если это представление `show` для аукциона 5, то мы получим ту же гиперссылку, что и раньше. *Почти*. Так как маршрут по умолчанию изменился, поменяется и порядок полей в URL:

```
<a href="http://localhost:3000/auctions/5/edit">Редактировать аукцион</a>
```

Никаких преимуществ это не дает. Но у нас и цель другая – понять, как работает система маршрутизации, наблюдая за тем, что происходит при небольших изменениях.

Использование литеральных URL

Если угодно, можете зашить пути и URL в виде строковых аргументов метода `link_to`, `redirect_to` и им подобных. Например, вместо:

```
<%= link_to "Справка", :controller => "main", :action => "help" %>
```

можно было бы написать:

```
<%= link_to "Справка", "/main/help" %>
```

Однако при использовании литерального пути или URL вы полностью обходите систему маршрутизации. Применяя литеральные URL, вы берете на себя ответственность за их сопровождение. (Можете, конечно, для вставки значений пользоваться механизмом интерполяции строк, который предоставляет Ruby, но прежде чем становиться на этот путь, подумайте, надо ли вам заново реализовывать функциональность Rails.)

Маскирование маршрутов

В некоторых случаях желательно выделить из маршрута один или несколько компонентов, не проводя поочередное сопоставление с конкретными позиционными параметрами. Например, ваши URL могут отражать структуру дерева каталогов. Если пользователь заходит на URL

```
files/list/base/books/fiction/dickens
```

то вы хотите, чтобы действие `files/list` получило доступ ко всем четырем оставшимся полям. Однако иногда полей может быть всего три:

```
/files/list/base/books/fiction
```

или пять:

```
/files/list/base/books/fiction/dickens/little_dorrit
```

Следовательно, необходим маршрут, который сопоставлялся бы со *всем после второй компоненты URI* (для данного примера).

Добиться этого можно с помощью *маскирования маршрутов* (route globbing). Для маскирования употребляется звездочка:

```
map.connect 'files/list/*specs'
```

Теперь действие `files/list` будет иметь доступ к массиву полей URL через элемент `params[:specs]`:

```
def list
  specs = params[:specs] # например, ["base", "books", "fiction", "dickens"]
end
```

Маска может встречаться только в конце строки-образца. Такая конструкция *недопустима*:

```
map.connect 'files/list/*specs/dickens' # Не работает!
```

Маска проглатывает все оставшиеся компоненты URL, поэтому из самой ее семантики вытекает, что она должна находиться в конце образца.

Маскирование пар ключ/значение

Маскирование маршрутов могло бы составить основу более общего механизма составления запросов о лотах, выставленных на аукцион. Предположим, что нужно придумать схему для представления URL следующего вида:

```
http://localhost:3000/items/field1/value1/field2/value2/...
```

В ответ на такой запрос необходимо вернуть список всех лотов, для которых указанные поля имеют указанные значения, причем количество пар «поле-значение» в URL неограниченно.

Иными словами, URL `http://localhost:3000/items/year/1939/medium/wood` должен генерировать список всех деревянных изделий, произведенных в 1939 году.

Эту задачу решает следующий маршрут:

```
map.connect 'items/*specs', :controller => "items", :action => "specify"
```

Разумеется, для поддержки такого маршрута необходимо соответствующим образом написать действие `specify`, например так, как показано в листинге 3.2.

Листинг 3.2. Действие specify

```
def specify
  @items = Item.find(:all, :conditions => Hash[params[:specs]])
  if @items.any?
    render :action => "index"
  else
    flash[:error] = "Не могу найти лоты с такими свойствами"
   redirect_to :action => "index"
  end
end
```

А что делает метод «квадратные скобки» класса `Hash`? Он преобразует одномерный массив пар «ключ/значение» в хеш! Еще одно свидетельство в пользу того, что без глубокого знания Ruby не стать экспертом в Rails.

Следующая остановка: именованные маршруты – способ инкапсуляции логики маршрутизации в специализированных методах-помощниках.

Именованные маршруты

Тема именованных маршрутов заслуживает отдельной главы. То, что вы сейчас узнаете, найдет непосредственное продолжение при изучении связанных с REST аспектов маршрутизации в главе 4.

Идея именованных маршрутов призвана главным образом облегчить жизнь программисту. С точки зрения приложения никаких видимых эффектов это не дает. Когда вы присваиваете маршруту имя, в контроллерах и представлениях определяется новый метод с именем `name_url` (где *name* – имя, присвоенное маршруту). При вызове этого метода с подходящими аргументами генерируется URL для маршрута. Кроме того, создается еще и метод `name_path`, который генерирует только путевую часть URL без протокола и имени хоста.

Создание именованного маршрута

Чтобы присвоить имя маршруту, вызывается особый метод объекта `map`, которому передается имя, а не обычный метод `connect`:

```
map.help 'help',
  :controller => "main",
  :action => "show_help"
```

В данном случае вы получаете методы `help_url` и `help_path`, которые можно использовать всюду, где Rails ожидает URL или его компонент:

```
<%= link_to "Справка!", help_path %>
```

И, разумеется, обычные правила распознавания и генерации остаются в силе. Образец включает только статическую строку `"help"`. Поэтому в гиперссылке вы увидите путь

```
/help
```

При щелчке по этой ссылке будет вызвано действие `show_help` контроллера `main`.

Что лучше: name_path или name_url?

При создании именованного маршрута в действительности создаются по крайней мере два метода-помощника. В предшествующем примере они назывались `help_url` и `help_path`. Разница между этими методами в том, что метод `_url` генерирует полный URL, включая протокол и доменное имя, а `_path` – только путь (иногда говорят *относительный путь*).

Согласно спецификации HTTP, при переадресации следует задавать URI, и некоторые считают, что речь идет о полностью квалифицированном URL.[1] Поэтому, если вы хотите быть педантом, то, вероятно, *следует* пользоваться методом `_url` при передаче именованного маршрута в качестве аргумента методу `redirect_to` в коде контроллера.

Метод `redirect_to`, похоже, отлично работает и с относительными путями, которые генерирует помощник `_path`, поэтому споры на эту тему более-менее бессмысленны. На самом деле, если не считать переадресации, перманентных ссылок и еще некоторых граничных случаев, *путь Rails* состоит в том, чтобы использовать `_path`, а не `_url`. Первый метод порождает более короткую строку, а пользовательский агент (броузер или еще что-то) должен уметь выводить полностью квалифицированный URL, зная HTTP-заголовки запроса, базовый адрес документа и URL запроса.

Читая эту книгу и изучая код и примеры из других источников, помните, что `help_url` и `help_path` делают по существу одно и то же. Я предпочитаю употреблять метод `_url` при обсуждении техники именованных маршрутов, но пользоваться методом `_path` внутри шаблонов представлений (например, при передаче параметров методам `link_to` и `form_for`). В общем, это вопрос стиля, базирующийся на теории о том, что URL – общая вещь, а путь – специализированная. В любом случае имейте в виду оба варианта и запомните, что они очень тесно связаны между собой.

Замечания

Именованные маршруты позволяют немного сэкономить на генерации URL. Именованный маршрут сразу выводит вас на тот маршрут, который вам нужен, минуя процедуру сопоставления. Это означает, что можно предоставлять меньше деталей, чем пришлось бы в противном случае. Задавать значения всех метапараметров в образце маршрута все равно необходимо, но про зашитые в код связанные параметры можно забыть. Единственная причина, по которой последние задаются, состоит в том, чтобы при генерации URL направить систему маршрутизации к нужному маршруту. Однако, когда вы используете именованный маршрут, система уже знает, какое правило вы хотите применить, поэтому налицо (небольшое) повышение производительности.

Как выбирать имена для маршрутов

Самый лучший способ понять, какие именованные маршруты вам нужны, – применить подход «сверху вниз». Подумайте, что вы хотите

[1] Зед Шоу (Zed Shaw), автор веб-сервера Mongrel и эксперт во всем, что относится к протоколу HTTP, не смог дать мне исчерпывающий ответ на этот вопрос, а это кое о чем говорит (о нестрогости спецификации HTTP, а не о некомпетентности Зеда).

написать в коде приложения, а потом создайте маршруты, которые помогут решить стоящую перед вами задачу.

Рассмотрим, например, следующее обращение к `link_to`:

```
<%= link_to "Аукцион по продаже #{h(auction.item.description)}",
    :controller => "auctions",
    :action => "show",
    :id => auction.id %>
```

Правило маршрутизации для сопоставления с этим путем (обобщенный маршрут) выглядит так:

```
map.connect "auctions/:id",
  :controller => "auctions",
  :action => "show"
```

Как-то не хочется еще раз перечислять все параметры маршрутизации только для того, чтобы система поняла, какой маршрут нам нужен. И было бы очень неплохо сократить код вызова `link_to`. В конце концов, в правиле маршрутизации контроллер и действие уже определены.

Вот вам и неплохой кандидат на роль именованного маршрута. Мы можем улучшить ситуацию, заведя маршрут `auction_path`:

```
<%= link_to "Аукцион по продаже #{h(auction.item.description)}",
     auction_path(:id => auction.id) %>
```

Присваивание маршруту имени – это срезание пути; имя выводит нас прямо на нужный маршрут без утомительного поиска, избавляя заодно от необходимости задавать длинные описания параметров, зашитых в маршрут.

> **Говорит Кортенэ...**
>
> Не забывайте экранировать описания лотов!
>
> Такие ссылки, как `#{auction.item.description}`, всегда следует заключать в метод `h()` во избежание атак с использованием *кросс-сайтовых сценариев* (XSS). Если, конечно, вы не реализовали какой-нибудь хитроумный способ контроля входных данных.

Именованный маршрут выглядит так же, как обычный, – мы лишь заменяем слово `connect` именем маршрута:

```
map.auction "auctions/:id",
  :controller => "auctions",
  :action => "show"
```

В представлении теперь можно использовать более компактный вариант `link_to`, а гиперссылка (для аукциона 3) содержит следующий URL:

```
http://localhost:3000/auctions/show/3
```

Синтаксическая глазурь

Аргумент, передаваемый методу `auction_path`, можно еще сократить. Если в качестве аргумента именованному маршруту нужно передать идентификатор, достаточно указать лишь число, опустив ключ `:id`:

```
<%= link_to "Аукцион по продаже #{h(auction.item.description)}",
     auction_path(auction.id) %>
```

Но можно пойти еще дальше – достаточно передать объекты, и Rails извлечет идентификатор автоматически:

```
<%= link_to "Аукцион по продаже #{h(auction.item.description)}",
     auction_path(auction) %>
```

Этот принцип распространяется и на другие метапараметры в строке-образце именованного маршрута. Например, если имеется такой маршрут:

```
map.item 'auction/:auction_id/item/:id',
  :controller => "items",
  :action => "show"
```

то, вызвав метод `link_to` следующим образом:

```
<%= link_to item.description, item_path(auction, item) %>
```

вы получите следующий путь (который зависит от конкретного значения идентификатора):

```
/auction/5/item/11
```

Здесь мы дали Rails возможность вывести идентификаторы объектов аукциона и лота. При условии что аргументы передаются в порядке расположения их идентификаторов в образце, в сгенерированный путь будут подставлены правильные значения.

Еще немного глазури?

Вовсе необязательно, чтобы генератор маршрутов вставлял в URL именно значение идентификатора. Можно подменить значение, определив в модели метод `to_param`. Предположим, вы хотите, чтобы в URL аукциона по продаже некоторого лота фигурировало название этого лота. В файле модели `item.rb` переопределим метод `to_param`. В данном случае сделаем это так, чтобы он возвращал «нормализованное» название (знаки препинания убраны, а между словами вставлены знаки подчеркивания):

```
def to_param
  description.gsub(/\s/, "-").gsub([^\w-], '').downcase
end
```

Тогда вызов метода `item_path(@item)` вернет нечто подобное:

```
/auction/3/item/cello-bow
```

Разумеется, если в поле `:id` вы помещаете строку типа cello-bow, то должны как-то научиться снова получать из нее объект. Приложения для ведения блогов, в которых эта техника используется с целью создания «жетонов» (slugs) в перманентных ссылках, часто заводят в базе данных отдельный столбец для хранения «нормализованной» версии названия, выступающего как часть пути. В результате для восстановления исходного объекта можно сделать что-то типа:

```
Item.find_by_munged_description(params[:id])
```

И, конечно, в маршруте можно назвать этот параметр не `:id`, а как-то более осмысленно!

Говорит Кортенэ...

Почему не следует употреблять в URL числовые идентификаторы?

Во-первых, потому что конкуренты могут увидеть, сколько вы создали аукционов. Во-вторых, если идентификаторы – последовательные числа, можно написать автоматизированного паука, который будет воровать ваш контент. В-третьих, это открывает дверь в вашу базу данных. И наконец, слова просто приятнее выглядят.

Метод организации контекста with_options

Иногда полезно создать несколько именованных маршрутов, относящихся к одному и тому же контроллеру. Эту задачу можно решить с помощью метода `with_options` объекта `map`.

Предположим, что имеются такие именованные маршруты:

```
map.help '/help', :controller => "main", :action => "help"
map.contact '/contact', :controller => "main", :action => "contact"
map.about '/about', :controller => "main", :action => "about"
```

Все три маршрута можно консолидировать следующим образом:

```
map.with_options :controller => "main" do |main|
  main.help '/help', :action => "help"
  main.contact '/contact', :action => "contact"
  main.about '/about', :action => "about"
end
```

Три внутренних вызова создают именованные маршруты с ограниченным контекстом, в котором значением параметра `:controller` является строка `"main"`, поэтому трижды повторять это не нужно.

Отметим, что внутренние методы выполняются от имени объекта `main`, а не `map`. После организации контекста вызовы методов вложенного объекта `main` выполняют всю грязную работу.

Говорит Кортенэ...

Квалифицированный программист Rails, измеряя производительность приложения под нагрузкой, обратит внимание, что маршрутизация, распознавание маршрутов, а также методы `url_for`, `link_to` и родственные им часто оказываются самой медленной частью цикла обработки запросов. (Примечание: это несущественно, пока количество просмотров страниц не достигнет тысяч в час, поэтому не занимайтесь преждевременной оптимизацией.)

Распознавание маршрутов работает медленно, потому что на время вычисления маршрута все остальное приостанавливается. Чем больше маршрутов, тем медленнее все крутится. В некоторых проектах количество маршрутов исчисляется сотнями.

Генерация URL работает медленно, потому что часто в странице встречается много обращений к `link_to`.

Что в этом случае делать разработчику? Первое, что следует предпринять, когда приложение кряхтит и стонет от непосильной нагрузки (вот ведь повезло кому-то!), – кэшировать сгенерированные URL или заменить их текстом. Каждый такой шаг позволяет выиграть какие-то миллисекунды, но все они суммируются.

Заключение

Первая половина этой главы помогла вам разобраться в общих принципах маршрутизации, основанной на правилах `map.connect`, и понять, что у системы маршрутизации двоякое назначение:

- распознавание входящих запросов и отображение их на действия контроллера с попутной инициализацией дополнительных переменных-приемников;
- распознавание параметров URL в методе `link_to` и родственных ему, а также поиск соответствующего маршрута, по которому можно сгенерировать HTML-ссылки.

Знания об общей природе маршрутизации мы дополнили более продвинутыми приемами, например использованием регулярных выражений и маскированием маршрутов.

Наконец, перед тем как двигаться дальше, удостоверьтесь, что понимаете, как работают именованные маршруты и почему они упрощают жизнь разработчика, позволяя сократить код представлений. В следующей главе мы будем определять группы взаимосвязанных именованных маршрутов, и вы поймете, что сейчас мы взобрались на вышку, с которой удобно прыгать в пучины REST.

4

REST, ресурсы и Rails

Пока не появился REST, я (как и многие другие) по-настоящему не понимал, куда помещать свое барахло.

Йонас Никлас, сообщение в списке рассылки по Ruby on Rails

В версии 1.2 в Rails была добавлена поддержка проектирования в соответствии с архитектурным стилем REST. Спецификация REST (Representational State Transfer – передача представляемых состояний) – сложная тема из области теории информации, ее полное рассмотрение выходит далеко за рамки этой главы[1]. Однако некоторые краеугольные положения мы все же осветим. В любом случае средства REST, предоставляемые Rails, могут быть вам полезны, даже если вы не эксперт и не горячий приверженец REST.

Основная причина состоит в том, что все разработчики сталкиваются с задачей названия и организации ресурсов и действий в своих приложениях. Типичные операции для всех приложений с хранением в базе данных прекрасно укладываются в парадигму REST, и очень скоро вы в этом убедитесь.

---

[1] Для тех, кто интересуется стилем Rest, каноническим текстом является диссертация Роя Филдинга, которую можно найти по адресу *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*. Особенно интересны главы 5 и 6, в которых REST рассматривается во взаимосвязи с HTTP. Кроме того, массу информации и ссылки на дополнительные ресурсы можно найти на вики-сайте, посвященном REST, по адресу *http://rest.blueoxen.net/cgi-bin/wiki.pl*.

О REST в двух словах

Рой Томас Филдинг (Roy T. Fielding), создатель REST, называет свое детище сетевым «архитектурным стилем», точнее, стилем, проявляющимся в архитектуре World Wide Web. На самом деле Филдинг является не только создателем REST, но и одним из авторов самого протокола HTTP, – REST и Сеть очень тесно связаны друг с другом.

Филдинг определяет REST как ряд ограничений, налагаемых на взаимодействие между компонентами системы. По существу, вначале имеется просто множество компьютеров, способных общаться между собой, а затем мы постепенно налагаем ограничения, разрешающие одни способы общения и запрещающие другие.

В число ограничений REST (среди прочих) входят:

- применение архитектуры клиент-сервер
- коммуникация без сохранения состояния
- явное извещение о возможности кэширования ответа

Сеть World Wide Web допускает коммуникацию, отвечающую требованиям REST. Но она также допускает и нарушения принципов REST – ограничения не будут соблюдаться, если вы специально об этом не позаботитесь. Однако Филдинг – один из авторов протокола HTTP и, хотя у него есть некоторые претензии к этому протоколу с точки зрения REST (как и критические замечания по поводу широкого распространения практики, не согласующейся с принципами REST, например использования cookies), общее соответствие между REST и веб – не случайное совпадение.

REST проектировался с целью помочь вам предоставлять службы, причем не произвольным образом, а в согласии с идиомами и конструкциями, присущими HTTP. Если поищете, то легко найдете многочисленные дискуссии, в которых REST сравнивается, например, с SOAP. Смысл аргументов в защиту REST сводится к тому, что HTTP уже позволяет предоставлять службы, поэтому дополнительный семантический уровень поверх него излишен. Просто надо уметь пользоваться тем, что дает HTTP.

Одно из достоинств стиля REST заключается в том, что он хорошо масштабируется для больших систем, например Сети. Кроме того, он поощряет, даже требует, использовать стабильные долгоживущие идентификаторы ресурсов (URI). Компьютеры общаются между собой, посылая запросы и ответы, помеченные этими идентификаторами. Эти запросы и ответы содержат также *представления* (в виде текста, XML, графики и т. д.) ресурсов (высокоуровневое, концептуальное описание содержимого). В идеале, запрашивая у компьютера XML-представление ресурса, скажем «Ромео и Джульетта», вы каждый раз указываете в запросе один и тот же идентификатор и метаданные, описывающие,

что требуется получить именно XML, и получаете один и тот же ответ. Если возвращаются разные ответы, должна быть причина, например запрашиваемый ресурс является изменяемым («Текущая интерпретация для студента №3994»).

Ниже мы еще вернемся к ресурсам и представлениям. А пока посмотрим, какое место в этой картине занимает Rails.

REST в Rails

Поддержка REST в Rails складывается из методов-помощников и дополнений к системе маршрутизации, спроектированных так, чтобы придать определенный стиль, логику и порядок контроллерам, а стало быть, и восприятию приложения внешним миром. Это больше чем просто набор соглашений об именовании (хотя и это тоже). Чуть ниже мы поговорим о деталях, а пока отметим, что по большому счету преимущества от использования REST в Rails можно разбить на две категории:

- для вас – удобство и автоматическое следование методикам, доказавшим свою состоятельность на практике;
- для всех остальных – согласованный с REST интерфейс к службам вашего приложения.

Извлечь пользу из первого преимущества можно даже в том случае, когда второе вас не волнует. На самом деле именно на этом аспекте мы и сконцентрируем внимание: как поддержка REST в Rails может облегчить вам жизнь и помочь сделать код элегантнее.

Мы не хотим преуменьшать важность стиля REST как такового или подвергать сомнению стремление предоставлять согласованные с REST службы. Просто невозможно рассказать обо всем на свете, а этот раздел книги посвящен маршрутизации, поэтому взглянем на REST именно под этим углом зрения.

Заметим еще, что взаимоотношения между Rails и REST, хотя и плодотворные, не свободны от сложностей. Многие подходы, применяемые в Rails, изначально не согласуются с предпосылками REST. Стиль REST предполагает коммуникацию без сохранения состояния – каждый запрос должен содержать все необходимое получателю для выработки правильного ответа. Но практически все сколько-нибудь сложные программы для Rails нуждаются в сохранении состояния на сервере для отслеживания сеансов. И эта практика несовместима с идеологией REST. Cookies на стороне клиента – тоже используемые во многих приложениях Rails – Филдинг отметает как не согласующуюся с REST практику.

Распутывать все узлы и разрешать все дилеммы мы не станем. Повторюсь: наша цель – показать, как устроена поддержка REST, и распах-

нуть двери для дальнейшего исследования и применения на практике – включая изучение диссертации Филдинга и теоретических постулатов REST. Мы не сможем рассмотреть все, но то, о чем мы будем говорить, совместимо с более широкой трактовкой темы.

История взаимоотношений REST и Rails начинается с CRUD...

Маршрутизация и CRUD

Акроним CRUD (Create Read Update Delete – Создание Чтение Обновление Удаление) – это классическая сводка операций с базой данных. Заодно это призывный клич разработчиков на платформе Rails. Поскольку мы обращаемся к базам данных с помощью абстракций, то склонны забывать, как на самом деле все просто. Проявляется это в придумывании слишком уж креативных имен для действий контроллеров. Возникает искушение называть действия как-то вроде `add_item`, `replace_email_address` и т. д. Но в этом нет никакой необходимости.

Да, контроллер не отображается на базу данных в отличие от модели. Но жизнь будет проще, если называть действия в соответствии с операциями CRUD или настолько близко к ним, насколько это возможно.

Система маршрутизации не «заточена» под CRUD. Можно создать маршрут, ведущий к любому действию, как бы оно ни называлось. Выбор CRUD-имен – это вопрос дисциплины. Но... при использовании средств поддержки REST, предлагаемых Rails, это происходит автоматически.

Поддержка REST в Rails подразумевает и стандартизацию имен действий. В основе этой поддержки лежит техника автоматического создания групп именованных маршрутов, которые жестко запрограммированы для указания на конкретный предопределенный набор действий.

В этом есть своя логика. Присваивать действиям CRUD-имена – это хорошо. Использовать именованные маршруты – удобно и элегантно. Поэтому применение механизмов REST – короткий путь к проверенным практикой подходам.

Слова «короткий путь» не передают, насколько мало от вас требуется для получения большой отдачи. Стоит поместить такое предложение:

```
map.resources :auctions
```

в файл `routes.rb`, как вы уже создадите четыре именованных маршрута, которые фактически позволяют соединиться с *семью* действиями контроллера, – как именно, будет описано в этой главе. И у этих действий будут симпатичные CRUD-совместимые имена.

Слово «resources» в выражении `map.resources` заслуживает особого внимания.

Ресурсы и представления

Стиль REST характеризует коммуникацию между *компонентами* системы (здесь компонентом может быть, скажем, веб-броузер или сервер) как последовательность запросов, ответами на которые являются *представления ресурсов*.

В данном контексте ресурс – это «концептуальное отображение» (Филдинг). Сами ресурсы не привязаны ни к базе данных, ни к модели, ни к контроллеру. Вот некоторые примеры ресурсов:

- текущее время дня
- история выдачи книги библиотекой
- полный текст романа «Крошка Доррит»
- карта города Остин
- инвентарная ведомость склада

Ресурс может быть одиночным или множественным, изменяемым (как время дня) или фиксированным (как текст «Крошки Доррит»). По существу это высокоуровневая абстракция того, что вы хотите получить, отправляя запрос.

Но получаете вы не сам ресурс, а его *представление*. Именно здесь REST распадается на мириады циркулирующих в Сети типов контента и фактически доставляемых данных. В любой момент времени для ресурса существует несколько представлений (в том числе 0). Так, ваш сайт может предлагать как текстовую, так и аудиоверсию «Крошки Доррит». Обе версии будут считаться одним и тем же ресурсом, на который указывает один и тот же *идентификатор* (URI). Нужный тип контента – то или иное представление – задается в запросе дополнительно.

Ресурсы REST и Rails

Как почти все в Rails, поддержка REST-совместимых приложений «пристрастна», то есть предлагается конкретный способ проектирования REST-интерфейса, и чем выше ваша готовность принять его, тем больший урожай удобств вы пожнете. Данные приложений Rails хранятся в базе, поэтому подход Rails к REST заключается в том, чтобы как можно теснее ассоциировать ресурс с моделью `ActiveRecord` или парой модель/контроллер.

Терминология используется довольно свободно, например, часто можно услышать выражение «ресурс Book». На самом деле, обычно подразумевается модель `Book`, контроллер `book` с набором CRUD-действий и ряд именованных маршрутов, относящихся к этому контроллеру (благодаря предложению `map.resources :books`). Но пусть даже модель `Book` и соответствующий контроллер существуют, ресурсы в смысле

REST, которые видны внешнему миру, обитают на более высоком уровне абстракции – «Крошка Доррит», история выдачи и т. д.

Лучший способ понять, как устроена поддержка REST в Rails, – двигаться от известного к новому, в данном случае от общей идеи именованных маршрутов к их специализации для целей REST.

От именованных маршрутов к поддержке REST

В начале разговора об именованных маршрутах мы приводили примеры консолидации различных сущностей в имени маршрута. Создав маршрут вида

```
map.auction 'auctions/:id',
  :controller => "auction",
  :action => "show"
```

вы получаете возможность воспользоваться удобными методами-помощниками в следующих ситуациях:

```
<%= link_to h(item.description), auction_path(item.auction) %>
```

Этот маршрут гарантирует, что будет сгенерирован путь, активирующий действие `show` контроллера `auctions`. Такие именованные маршруты хороши краткостью и легкостью зрительного восприятия.

Ассоциировав метод `auction_path` с действием `auction/show`, мы сделали все необходимое в терминах стандартных операций с базой данных. А теперь взглянем на это с точки зрения CRUD. Именованный маршрут `auction_path` хорошо согласуется с именем `show` (буквой R в акрониме CRUD). А что, если нам нужны хорошие имена маршрутов для действий `create`, `update` и `delete`.

Имя `auction_path` мы уже использовали для действия `show`. Можно было бы предложить имена `auction_delete_path`, `auction_create_path`... но они выглядят как-то громоздко. В действительности нам хотелось бы, чтобы вызов `auction_path` означал разные вещи в зависимости от того, на какое действие указывает URL.

Поэтому нужен способ отличить один вызов `auction_path` от другого. Можно было бы различать единственное (`auction_path`) и множественное (`auctions_path`) число. URL в единственном числе семантически означает, что мы хотим что-то сделать с одним существующим объектом аукциона. Если же операция производится над множеством аукционов, то больше подходит множественное число.

К числу множественных операций над аукционами относится и создание. Обычно действие `create` встречается в таком контексте:

```
<% form_tag auctions_path do |f| %>
```

Здесь употребляется множественное число, поскольку мы говорим не «выполнить действие применительно к конкретному аукциону», а «при-

менительно ко всему множеству аукционов выполнить действие создания». Да, мы создаем один аукцион, а не много. Но в момент обращения к именованному маршруту auctions_path мы имеем в виду все аукционы вообще.

Другой случай, когда имеет смысл маршрут с именем во множественном числе, – получение списка всех объектов определенного вида или просто какое-то общее представление, не ограниченное отображением одного объекта. Такое представление обычно реализуется действием index. Подобные действия, как правило, загружают много данных в одну или несколько переменных, а соответствующее представление выводит их в виде списка или таблицы (возможно, не одной).

И в этом случае хорошо бы иметь возможность написать так:

```
<%= link_to "Щелкните здесь для просмотра всех аукционов", auctions_path %>
```

Но тут идея о создании вариантов маршрута auction_path в единственном и множественном числе упирается в потолок: уже есть два места, где требуется множественное число. Одно из них create, другое – index. Однако выглядят они одинаково:

```
http://localhost:3000/auctions
```

Как система маршрутизации узнает, что в одном случае мы имели в виду действие create, а в другом – index? Нужен еще какой-то флаг – переменная, по которой можно было бы организовать ветвление.

К счастью, такая переменная есть.

И снова о глаголах HTTP

Формы отправляются методом POST. Действия index запрашиваются методом GET. Следовательно, нам нужно, чтобы система маршрутизации понимала, что

```
/auctions в GET-запросе
```

и

```
/auctions в POST-запросе
```

это разные вещи. Кроме того, мы хотим, чтобы она генерировала один и тот же URL – /auctions – но разными HTTP-методами в зависимости от обстоятельств.

Именно это и делает механизм поддержки REST в Rails. Он позволяет сказать, что маршрут /auctions должен вести в разные места в зависимости от метода HTTP-запроса. Он дает возможность определить маршруты с одинаковыми именами, учитывающие, какой глагол HTTP употреблен. Короче говоря, глаголы HTTP используются в нем в качестве тех самых дополнительных данных, которые необходимы для лаконичного решения поставленной задачи.

Это достигается за счет применения специальной команды маршрутизации: `map.resources`. Вот как она выглядит для аукционов:

```
map.resources :auctions
```

Это все. Одна такая строка в файле `routes.rb` эквивалентна определению четырех именованных маршрутов (как вы вскоре убедитесь). А в результате комбинирования четырех маршрутов с различными методами HTTP-запросов вы получаете семь полезных – очень полезных – перестановок.

Стандартные REST-совместимые действия контроллеров

Вызов `map.resources :auctions` означает заключение некоей сделки с системой маршрутизации. Система отдает вам четыре именованных маршрута. Они могут вести на одно из семи действий контроллера в зависимости от метода HTTP-запроса. В обмен вы соглашаетесь использовать строго определенные имена действий контроллера: index, create, show, update, destroy, new, edit.

Ей-богу, это неплохая сделка, так как система проделывает за вас большую работу, а навязываемые имена действий очень близки к CRUD.

В табл. 4.1 суммированы все действия. Она устроена, как таблица умножения, – на пересечении строки, содержащей именованный маршрут, и столбца с методом HTTP-запроса находится то, что вы получаете от системы. В каждой клетке (кроме незаполненных) показан, во-первых, генерируемый маршрутом URL, а во-вторых, действие, вызываемое при распознавании этого маршрута (в таблице упоминается только метод `_url`, но вы получаете и метод `_path`).

Таблица 4.1. REST-совместимые маршруты, а также помощники, пути и результирующие действия контроллера

Метод-помощник	GET	POST	PUT	DELETE
client_url(@client)	/clients/1 *show*		/clients/1 *update*	/clients/1 *destroy*
clients_url	/clients *index*	/clients *create*		
edit_client_url (@client)	/clients/1/edit *edit*			
new_client_url	/clients/new *new*			

Для действий `edit` и `new` имена маршрутов уникальны, а в URL применяется специальный синтаксис. К этим особым случаям мы еще вернемся ниже.

Поскольку именованные маршруты комбинируются с HTTP-методами, необходимо знать, как при генерации URL задать метод запроса,

чтобы маршруты clients_url для методов GET и POST не выполняли одно и то же действие контроллера. Бóльшая часть того, что надлежит сделать, можно свести к нескольким правилам:

1. По умолчанию подразумевается HTTP-метод GET.
2. При обращении к методам form_tag и form_for автоматически используется HTTP-метод POST.
3. При необходимости (а она возникает в основном для операций PUT и DELETE) вы можете явно указать метод запроса в дополнение к URL, сгенерированному именованным маршрутом.

   Необходимость задавать операцию DELETE возникает, например, в ситуации, когда вы хотите с помощью ссылки активировать действие destroy:

```
<%= link_to "Удалить этот аукцион", :url => auction(@auction),
        :method => :delete %>
```

В зависимости от использованного метода-помощника (типа form_for) можно поместить название HTTP-метода во вложенный хеш:

```
<% form_for "auction", :url => auction(@auction),
       :html => { :method => :put } do |f| %>
```

В этом примере маршрут с именем в единственном числе комбинируется с методом PUT, что приводит к вызову действия update (см. пересечение строки 2 со столбцом 4 в табл 4.1).

Хитрость для методов PUT и DELETE

Вообще говоря, веб-браузеры отправляют запросы только методами GET и POST. Чтобы заставить их посылать запросы PUT и DELETE, Rails необходимо проявить некую «ловкость рук». Вам об этом беспокоиться не надо, но полезно знать, что происходит за кулисами.

Запрос методом PUT или DELETE в контексте REST в Rails – это на самом деле POST-запрос со скрытым полем _method, которому присваивается значение put или delete. Приложение Rails, обрабатывающее запрос, замечает это и маршрутизирует запрос на действие update или destroy соответственно.

Таким образом, можно сказать, что поддержка REST в Rails опережает время. Компоненты REST, применяющие протокол HTTP, *обязаны* понимать все методы запроса. Но не понимают, поэтому Rails приходится вмешаться. Разработчику, который пытается понять, как именованные маршруты отображаются на имена действий, необязательно задумываться об этом мелком мошенничестве. И хочется надеяться, что со временем нужда в нем отпадет.

Одиночные и множественные REST-совместимые маршруты

Имена некоторых REST-совместимых маршрутов записываются в единственном числе (одиночные маршруты), других – во множественном (множественные маршруты). Логика такова:

1. Маршруты для действий `show`, `new`, `edit` и `destroy` одиночные, так как они применяются к конкретному ресурсу.
2. Остальные маршруты множественные. Они относятся к множествам взаимосвязанных ресурсов.

Одиночным REST-совместимым маршрутам требуется аргумент, так как они должны знать идентификатор элемента множества, к которому применяется действие. Синтаксически допускается как простой список аргументов:

```
item_url(@item) # show, update или destroy в зависимости от глагола HTTP
```

так и хеш:

```
item_url(:id => @item)
```

Не требуется (хотя и не возбраняется) вызывать метод `id` объекта `@item`, так как Rails понимает, что вы хотите сделать именно это.

Специальные пары: new/create и edit/update

Как видно из табл. 4.1, действия `new` и `edit` подчиняются специальным соглашениям о REST-совместимых именах. Причина связана с действиями `create` и `update` и с тем, как они связаны с действиями `new` и `edit`.

Обычно операции `create` и `update` вызываются путем отправки формы. Это означает, что с каждой из них на самом деле ассоциировано два действия – два запроса:

1. Действие, приводящее к отображению формы.
2. Действие, заключающееся в обработке данных отправленной формы.

С точки зрения REST-совместимой маршрутизации это означает, что действие `create` тесно связано с предшествующим ему действием `new`, а действие `update` – с действием `edit`. Действия `new` и `edit` играют роль ассистентов; их единственное назначение – показать пользователю форму, необходимую для создания или обновления ресурса.

Для включения этих двухшаговых сценариев в общую картину ресурсов требуется небольшой трюк. Форма для редактирования ресурса сама по себе ресурсом не является. Это скорее «предресурс». Форму для создания нового ресурса можно рассматривать как некий вид ресурса, если допустить, что «быть новым», то есть не существовать, – нечто такое, что ресурс может сделать, не переставая быть ресурсом…

Да, подпустил я философии. Но вот как все это реализовано в Rails для REST.

Считается, что действие `new` создает новый одиночный ресурс (а не множество ресурсов). Однако, так как логически эта транзакция описывается глаголом GET, а GET для одиночного ресурса уже соответствует действию `show`, то для `new` необходим маршрут с отдельным именем.

Вот почему мы вынуждены писать

```
<%= link_to "Создать новый лот", new_item_path %>
```

чтобы получить ссылку на действие `items/new`.

Что касается действия `edit`, то оно не должно давать полноценный ресурс, а скорее быть некоей «разновидностью для редактирования» действия `show`. Поэтому для него применяется тот же URL, что для `show`, но с модификатором в виде суффикса `/edit`, что совместимо с форматом URL для действия `new`:

```
/items/5/edit
```

Стоит отметить, что до выхода версии Rails 2.0 действие `edit` отделялось точкой с запятой: `/items/5;edit`. Это решение было продиктовано скорее ограничениями системы маршрутизации, нежели более возвышенными мотивами. Однако подобная схема создавала больше проблем, чем решала[1], и была исключена из «острия Rails» сразу после выхода версии Rails 1.2.3.

Соответствующий именованный маршрут называется `edit_item_url(@item)`. Как и в случае `new`, имя маршрута для действия `edit` содержит дополнительные слова, чтобы отличить его от маршрута к действию `show`, предназначенного для получения существующего одиночного ресурса методом `GET`.

Одиночные маршруты к ресурсам

Помимо метода `map.resources`, существует одиночная (или «синглетная») форма маршрутизации ресурса: `map.resource`. Она используется для представления ресурса, который в данном контексте существует в единственном числе.

---

[1] Точка с запятой не только выглядит *странно*, но и создает ряд более существенных проблем. Например, она очень мешает кэшированию. Пользователи броузера Safari не могли аутентифицировать URL, содержащие точку с запятой. Кроме того, некоторые веб-серверы (и прежде всего Mongrel) справедливо считают, что точки с запятой являются частью строки запроса, так как этот символ зарезервирован, чтобы обозначать начало параметров пути (относящихся к элементу пути между символами косой черты, в отличие от параметров запроса, следующих за символом «?»).

Одиночный маршрут к ресурсу, расположенный в начале списка маршрутов, может быть полезен, когда во всем приложении или, быть может, в сеансе пользователя существует только один ресурс такого типа.

Например, приложение для ведения адресных книг предоставляет каждому зарегистрированному пользователю собственную адресную книгу, поэтому можно было бы написать:

```
map.resource :address_book
```

В результате из всего множества маршрутов к ресурсу вы получите только одиночные: `address_book_url` для `GET/PUT`, `edit_address_book_url` для `GET` и `update_address_book_url` для `PUT`.

Отметим, что имя метода `resource`, аргумент этого метода и имена всех именованных маршрутов записываются в единственном числе. Предполагается, что вы работаете в контексте, где имеет смысл говорить «адресная книга» – одна и только одна, поскольку для текущего пользователя есть лишь одна адресная книга. Контекст не устанавливается автоматически – вы должны аутентифицировать пользователя и извлечь его адресную книгу из базы данных (а равно и сохранить ее) явно. В этом отношении никакой «магии» или чтения мыслей не предусмотрено; этого всего лишь еще одна техника маршрутизации, которой вы при желании можете воспользоваться.

Вложенные ресурсы

Предположим, что нужно выполнять операции над заявками: создание, редактирование и т. д. Вы знаете, что каждая заявка ассоциирована с некоторым аукционом. Это означает, что, выполняя операцию над заявкой, вы на самом деле оперируете парой заявка/аукцион или, если взглянуть под другим углом зрения, вложенной структурой аукцион/заявка. Заявки всегда встречаются в конце пути, проходящего через аукцион.

Таким образом, в данном случае необходим URL вида

```
/auctions/3/bids/5
```

Действия при получении запроса к такому URL зависят, разумеется, от употребленного глагола HTTP. Но семантика самого URL такова: ресурс, который можно идентифицировать как заявку 5 на аукционе 3.

Почему бы просто не перейти на URL `bids/5`, минуя аукцион? На то есть две причины. Во-первых, первоначальный URL более информативен. Согласен, он длиннее, но увеличение длины служит для того, чтобы сообщить дополнительную информацию о ресурсе. Во-вторых, механизм реализации REST-совместимых маршрутов в Rails при таком URL дает вам непосредственный доступ к идентификатору аукциона через `params[:auction_id]`.

Для создания маршрутов к вложенным ресурсам поместите в файл routes.rb такие строки:

```
map.resources :auctions do |auction|
  auction.resources :bids
end
```

Отметим, что внутренний метод resources вызывается для объекта auction, а не map. Об этом часто забывают.

Смысл данной конструкции состоит в том, чтобы сообщить объекту map, что вам нужны REST-совместимые маршруты к ресурсам аукциона, то есть вы хотите получить auctions_url, edit_auction_url и все остальное. Кроме того, вам необходимы REST-совместимые маршруты к заявкам: auction_bids_url, new_auction_bid_url и т. д.

Однако, применяя команду для получения вложенных ресурсов, вы обещаете, что при любом использовании именованного маршрута к заявке будете указывать аукцион, в который она вложена. В коде приложения это выглядит как аргумент метода именованного маршрута:

```
<%= link_to "Смотреть все заявки", auction_bids_path(@auction) %>
```

Такой вызов позволяет системе маршрутизации добавить часть /auctions/3 перед /bids. А на принимающем конце – в данном случае в действии bids/index, на которое этот URL указывает, – вы сможете найти идентификатор аукциона @auction в элементе params[:auction_id] (это множественный REST-совместимый маршрут для метода GET; если забыли, справьтесь с табл. 4.1).

Глубина вложенности может быть произвольна. Каждый уровень вложенности на единицу увеличивает количество аргументов, передаваемых вложенным маршрутам. Следовательно, для одиночных маршрутов (show, edit, destroy) требуются по меньшей мере два аргумента, как показано в листинге 4.1.

Листинг 4.1. Передача двух параметров для идентификации вложенного ресурса с помощью link_to

```
<%= link_to "Удалить эту заявку",
    auction_bid_path(@auction, @bid), :method => :delete %>
```

Это позволяет системе маршрутизации получить информацию (@auction.id и @bid.id), необходимую ей для генерации маршрута.

Если хотите, можете добиться того же результата, передавая аргументы в хеше, но обычно так не делают, потому что код получается длиннее:

```
auction_bid_path(:auction => @auction, :bid => @bid)
```

Явное задание :path_prefix

Добиться эффекта вложенности маршрутов можно также, явно указав параметр `:path_prefix` при обращении к методу отображения ресурсов. Вот как это можно сделать для заявок, вложенных в аукционы:

```
map.resources :auctions
map.resources :bids, :path_prefix => "auctions/:auction_id"
```

В данном случае вы говорите, что все URL заявок должны включать статическую строку `auctions` и значение `auction_id`, то есть контекстную информацию, необходимую для ассоциирования заявок с конкретным аукционом.

Основное отличие этого подхода от настоящего вкладывания ресурсов связано с именами генерируемых методов-помощников. Вложенные ресурсы автоматически получают префикс имени, соответствующий родительскому ресурсу (см. `auction_bid_path` в листинге 4.1.)

Скорее всего, техника вкладывания будет встречаться вам чаще, чем явное задание `:path_prefix`, потому что обычно проще позволить системе маршрутизации самостоятельно вычислить префикс, исходя из пути вложения ресурсов. Плюс, как мы скоро увидим, при желании нетрудно избавиться от лишних префиксов.

Явное задание :name_prefix

Иногда некий ресурс требуется вложить в несколько других ресурсов. Или в одном случае обращаться к ресурсу по вложенному маршруту, а в другом – напрямую. Может даже возникнуть желание, чтобы помощники именованных маршрутов указывали на разные ресурсы в зависимости от контекста, в котором исполняются[1]. Все это возможно с помощью префикса имени `:name_prefix`, поскольку он позволяет управлять процедурой, генерирующей методы-помощники для именованных маршрутов.

Предположим, что вы хотите обращаться к заявкам не только через аукционы, как в предыдущих примерах, но и указывая лишь номер заявки. Иными словами, требуется, чтобы распознавались и генерировались маршруты обоих видов:

`/auctions/2/bids/5` и `/bids/5`

Первое, что приходит в голову, – задать в качестве первого помощника `bid_path(@auction, @bid)`, а в качестве второго – `bid_path(@bid)`. Кажется

[1] Тревор Сквайрс (Trevor Squires) написал замечательный подключаемый модуль ResourceFu, позволяющий реализовать такую технику. Вы можете загрузить его со страницы *http://agilewebdevelopment.com/plugins/resource_fu*.

логичным предположить, что если необходим маршрут к заявке, не проходящий через объемлющий ее аукцион, можно просто опустить параметр, определяющий аукцион.

Принимая во внимание, что система маршрутизации автоматически задает префиксы имен, вы должны переопределить `name_prefix` для заявок, чтобы все работало, как задумано (листинг 4.2).

Листинг 4.2. Переопределение name_prefix во вложенном маршруте

```
map.resources :auctions do |auction|
  auction.resources :bids, :name_prefix => nil
end
```

Я широко применял такую технику в реальных приложениях и хочу заранее предупредить вас, что после исключения механизма префиксации имен отлаживать ошибки маршрутизации становится на порядок труднее. Но у вас может быть и другое мнение.

В качестве примера рассмотрим, что нужно сделать, захоти мы получать доступ к заявкам по другому маршруту – через того, кто из разместил, а не через аукцион (листинг 4.3).

Листинг 4.3. Переопределение name_prefix во вложенном маршруте

```
map.resources :auctions do |auction|
  auction.resources :bids, :name_prefix => nil
end

map.resource :people do |people|
  people.resources :bids, :name_prefix => nil # вы уверены?
end
```

Поразительно, но код в листинге 4.3 должен бы[1] работать правильно и генерировать следующих помощников:

```
bid_path(@auction, @bid) # /auctions/1/bids/1
bid_path(@person, @bid) # /people/1/bids/1
```

Но, если идти по этому пути, код контроллера и представления может усложниться.

Сначала контроллер должен будет проверить, какой из элементов `params[:auction_id]` и `params[:person_id]` существует, и загрузить соответствующий контекст. В шаблонах представлений, вероятно, придется выполнить аналогичную проверку, чтобы сформировать правильное отображение. В худшем случае появится куча предложений `if/else`, загромождающих код!

Решая заняться программированием подобного дуализма, вы, скорее всего, идете по ложному пути. К счастью, мы можем явно указать,

[1] Могу лишь сказать должен бы, поскольку маршрутизация исторически является наиболее изменчивой частью Rails, так что работа этого кода зависит от конкретной версии. Я точно знаю, что в версии Rails 1.2.3 он не работает.

какой контроллер следует ассоциировать с каждым из наших маршрутов.

Явное задание REST-совместимых контроллеров

Мы пока еще не говорили о том, как REST-совместимые маршруты отображаются на контроллеры. Из всего вышесказанного могло сложиться впечатление, что это происходит автоматически. На самом деле так оно и есть, и основой является имя ресурса.

Вернемся к нашему примеру и рассмотрим следующий вложенный маршрут:

```
map.resources :auctions do |auction|
  auction.resources :bids
end
```

Здесь участвуют два контроллера: `AuctionsController` и `BidsController`.

Можно явно указать, какой из них использовать, задействовав параметр `:controller` метода `resources`. Он позволяет присвоить ресурсу произвольное имя (видимое пользователю), сохранив согласованность имени контроллера с различными стандартами именования, например:

```
map.resources :my_auctions, :controller => :auctions do |auction|
  auction.resources :my_bids, :controller => :bids
end
```

А теперь все вместе

Теперь, познакомившись с параметрами `:name_prefix`, `:path_prefix`, и `:controller`, мы можем собрать все воедино и показать, когда точный контроль над REST-совместимыми маршрутами может быть полезен.

Например, сделанное в листинге 4.3 можно усовершенствовать, воспользовавшись параметром `:controller` (листинг 4.4).

Листинг 4.4. Несколько вложенных ресурсов заявок с явно заданным контроллером

```
map.resources :auctions do |auction|
  auction.resources :bids, :name_prefix => nil,
                           :controller => :auction_bids
end

map.resource :people do |people|
  people.resources :bids, :name_prefix => nil,
                          :controller => :person_bids
end
```

На практике классы `AuctionBidsController` и `PersonBidsController`, вероятно, будут расширять один и тот же родительский класс, как показа-

но в листинге 4.5, и пользоваться фильтрами before для загрузки правильного контекста.

Листинг 4.5. Определение подклассов контроллеров для работы с вложенными маршрутами

```
class BidsController < ApplicationController
  before_filter :load_parent
  before_filter :load_bid

  protected

    def load_parent
      # переопределяется в подклассах
    end

    def load_bid
      @bids = @parent.bids
    end
end

class AuctionBidsController < BidsController

  protected

    def load_parent
      @parent = @auction = Auction.find(params[:auction_id])
    end
end

class PersonBidsController < BidsController

  protected

    def load_parent
      @parent = @person = Person.find(params[:person_id])
    end
end
```

Отметим, что обычно именованные параметры задаются в виде символов, но параметр :controller понимает и строки, поскольку это необходимо, когда класс контроллера находится в пространстве имен, как в следующем примере, где задается административный маршрут для аукционов:

```
map.resources :auctions,
         :controller => 'admin/auctions', # Admin::AuctionsController
         :name_prefix => 'admin_',
         :path_prefix => 'admin'
```

Замечания

Нужна ли вложенность? В случае одиночных маршрутов вложенность обычно не дает ничего такого, что без нее получить нельзя. В конце концов любая заявка принадлежит какому-то аукциону. Это означает, что доступ к `bid.auction_id` ничуть не сложнее, чем к `params[:auction_id]`, в предположении, что объект заявки у вас уже есть.

Более того, объект заявки не зависит от вложенности. Элемент `params[:id]` получит значение 5, и соответствующую запись можно извлечь из базы данных напрямую. Совсем необязательно знать, какому аукциону эта заявка принадлежит.

```
Bid.find(params[:id])
```

Стандартное обоснование разумного применения вложенных ресурсов, которое чаще всего приводит Дэвид, – простота контроля над разрешениями и контекстными ограничениями. Как правило, доступ к вложенному ресурсу должен быть разрешен только в контексте родительского ресурса, и проконтролировать это в программе несложно, если помнить, что вложенный ресурс загружается с помощью ассоциации `ActiveRecord` родителя (листинг 4.6).

Листинг 4.6. Загрузка вложенного ресурса с помощью ассоциации has_many родителя

```
@auction = Auction.find(params[:auction_id])
@bid = @auction.bids.find(params[:id]) # предотвращает несоответствие между
                                       # аукционом и заявкой
```

Если вы хотите добавить к аукциону заявку, то URL вложенного ресурса будет выглядеть так:

```
http://localhost:3000/auctions/5/bids/new
```

Аукцион идентифицируется в URL, в результате чего отпадает необходимость загромождать форму скрытыми полями, называть действие `add_bid`, сохранять идентификатор пользователя в `:id` и прибегать к другим не согласующимся с REST хитростям.

О глубокой вложенности

Джеймис Бак (Jamis Buck) – очень влиятельная фигура в сообществе пользователей Rails, почти такая же влиятельная, как сам Дэвид. В феврале 2007 года в своем блоге[1] он поделился мыслью о том, что глубокая вложенность – это *плохо*, и предложил следующее эвристическое правило: «Уровень вложенности ресурса никогда не должен превышать единицу».

[1] *http://weblog.jamisbuck.org/2007/2/5/nesting-resources.*

Этот совет продиктован опытом и практическими соображениями. Методы-помощники для маршрутов, вложенных более чем на два уровня, становятся слишком длинными и неуклюжими. При работе с ними легко допустить ошибку, а понять, что не так, довольно сложно.

Предположим, что в нашем приложении с заявкой может быть связано несколько комментариев. Можно было бы следующим образом определить маршрут, в котором комментарии вложены в заявки:

```
map.resources :auctions do |auctions|
  auctions.resources :bids do |bids|
    bids.resources :comments
  end
end
```

Но тогда пришлось бы прибегать к различным параметрам, чтобы избежать появления помощника с именем `auction_bid_comments_path` (это еще не так плохо, мне встречались куда более уродливые имена).

Вместо этого Джеймис предлагает поступать следующим образом:

```
map.resources :auctions do |auctions|
  auctions.resources :bids
end

map.resources :bids do |bids|
  bids.resources :comments
end

map.resources :comments
```

Обратите внимание, что каждый ресурс (за исключением аукциона) определен дважды: один раз – в пространстве имен верхнего уровня, а другой – в своем собственном контексте. Обоснование? Для работы с отношением родитель-потомок вам в действительности нужны только два уровня. В результате URL становятся короче, а с методами-помощниками проще иметь дело.

```
auctions_path        # /auctions
auctions_path(1)     # /auctions/1
auction_bids_path(1) # /auctions/1/bids
bid_path(2)          # /bids/2
bid_comments_path(3) # /bids/3/comments
comment_path(4)      # /comments/4
```

Говорит Кортенэ...

Многие из нас не согласны с уважаемым Джеймисом. Хотите устроить потасовку на конференции по Rails? Задайте вопрос: «Кто считает, что более одного уровня вложенности в маршруте – это хорошо?»

Лично я не всегда следую рекомендации Джеймиса в своих проектах, но обратил внимание на одну вещь, относящуюся к ограничению глубины вложенности ресурсов: в этом случае можно оставлять префиксы имен на своих местах, а не обрубать их с помощью `:name_prefix => nil`. И поверьте мне, префиксы имен очень здорово упростят сопровождение вашего кода в будущем.

Настройка REST-совместимых маршрутов

REST-совместимые маршруты дают группу именованных маршрутов, заточенных для вызова ряда весьма полезных и общеупотребительных действий контроллеров – надмножества CRUD, о котором мы уже говорили. Но иногда хочется выполнить добавочную настройку, не отказываясь от преимуществ, которые дает соглашение об именовании REST-совместимых маршрутов и «таблица умножения», описывающая комбинации именованных маршрутов с методами HTTP-запросов. Например, это было бы полезно при наличии нескольких вариантов просмотра ресурса, которые можно назвать «показами». Вы не можете (и не должны) применять действие `show` более чем для одного такого варианта. Лучше представлять это как разные взгляды на ресурс и создать URL для каждого такого взгляда.

Маршруты к дополнительным действиям

Пусть, например, мы хотим реализовать возможность отзыва заявки. Основной вложенный маршрут для заявок выглядит так:

```
map.resources :auctions do |a|
  a.resources :bids
end
```

Мы хотели бы иметь действие `retract`, которое показывает форму (и, быть может, выполняет проверки допустимости отзыва). Действие `retract` – не то же самое, что `destroy`; оно – скорее, предтеча `destroy`. В этом смысле данное действие аналогично действию `edit`, которое выводит форму для последующего действия `update`. Проводя параллель с парой `edit`/`update`, мы хотели бы, чтобы URL выглядел так:

```
/auctions/3/bids/5/retract
```

а метод-помощник назывался `retract_bid_url`. Достигается это путем задания дополнительного маршрута `:member` для `bids`, как показано в листинге 4.7.

Листинг 4.7. Добавление маршрута к дополнительному действию

```
map.resources :auctions do |a|
  a.resources :bids, :member => { :retract => :get }
end
```

Если затем следующим образом добавить ссылку на операцию отзыва в представление:

```
<%= link_to "Отозвать", retract_bid_path(auction, bid) %>
```

то сгенерированный URL будет содержать модификатор `/retract`. Но такая ссылка, вероятно, должна выводить форму отзыва, а не выполнять саму процедуру отзыва! Я это говорю, потому что, согласно базовым принципам HTTP, GET-запрос не должен изменять состояние сервера – для этого предназначены POST-запросы. Достаточно ли добавить параметр `:method` в вызов `link_to`?

```
<%= link_to "Отозвать", retract_bid_path(auction,bid), :method=>:post
%>
```

Не совсем. Напомню, что в листинге 4.7 мы определили маршрут к операции отзыва как `:get`, потому система маршрутизации не распознает POST-запрос. Но решение есть – надо лишь определить маршрут так, чтобы на него отображался любой глагол HTTP:

```
map.resources :auctions do |a|
  a.resources :bids, :member => { :retract => :any }
end
```

Дополнительные маршруты к наборам

Описанной техникой можно воспользоваться для добавления маршрутов, которые концептуально применимы ко всему набору ресурсов:

```
map.resources :auctions, :collection => { :terminate => :any }
```

Этот пример дает метод `terminate_auctions_path`, который порождает URL, отображаемый на действие `terminate` контроллера `auctions` (пример, пожалуй, несколько странный, но идея в том, что он позволяет завершить сразу все аукционы).

Таким образом, вы можете, оставаясь в рамках совместимости с REST, точно настраивать поведение маршрутизации в своем приложении и включать особые случаи, не переставая рассуждать в терминах ресурсов.

Замечания

При обсуждении REST-совместимой маршрутизации в списке рассылки Rails[1], Джош Сассер (Josh Susser) предложил *инвертировать* синтаксис записи нестандартных действий, чтобы ключом был глагол HTTP, а значением – массив имен действий:

```
map.resources :comments,
              :member => { :get => :reply,
                           :post => [:reply, :spawn, :split] }
```

[1] Полностью с обсуждением можно познакомиться по адресу *http://www.ruby-forum.com/topic/75356*.

Среди других причин Джош отметил, что это здорово упростило бы написание так называемых *возвратов* (post-back), то есть действий контроллера двойного назначения, которые умеют обрабатывать GET и POST-запросы в одном методе.

Дэвид отозвался негативно. Возразив против возвратов в принципе, он сказал: «Я начинаю думать, что явное игнорирование [возвратов] в `map.resources` – это запроектированная особенность».

Ниже в том же обсуждении, продолжая защищать API, Дэвид добавил: «Если вы пишете так много дополнительных методов, что повторение начинает надоедать, следует пересмотреть исходные позиции. *Возможно, вы не так уж хорошо следуете REST, как могли бы*» (курсив мой).

Ключевой является последняя фраза. Включение дополнительных действий портит элегантность общего дизайна REST-совместимых приложений, поскольку уводит в сторону от выявления всех ресурсов, характерных для вашей предметной области.

Памятуя, что *реальные* приложения сложнее примеров в справочном руководстве, посмотрим все же, нельзя ли смоделировать отзывы строго, с использованием ресурсов. Вместо того чтобы включать действие `retract` в контроллер `BidsController`, может быть, стоит ввести отдельный ресурс «отзыв», ассоциированный с заявками, и написать для его обработки контроллер `RetractionController`.

```
map.resources :bids do |bids|
  bids.resource :retraction
end
```

Теперь `RetractionController` можно сделать ответственным за все операции, касающиеся отзывов, а не мешать эту функциональность с `BidsController`. Если вдуматься, отзыв заявок – достаточно сложное дело, которое в конце концов, все равно обросло бы громоздкой логикой. Пожалуй, выделение для него отдельного контроллера можно назвать *надлежащим разделением обязанностей* и даже *правильным объектно-ориентированным подходом*.

Не могу не продолжить рассказ об этом знаменательном обсуждении в списке рассылки, потому что с ним связан бесценный момент в истории сообщества Rails, укрепивший нашу репутацию как «пристрастной шайки»!

Джош ответил: «Хочу уточнить... Вы считаете, что код, который трудно читать и утомительно писать, – это достоинство? Пожалуй, с позиций сравнения макро- и микрооптимизации я бы не стал с вами спорить, но полагаю, что это спорный способ побудить людей писать правильно. Если совместимость с REST сводится только к этому, то не надо думать, что *синтаксический уксус* заставит народ поступать как надо. Однако если вы хотели сказать, что организация хеша действий в виде `{:action => method, ...}` желательна, так как гарантирует, что каждое

действие будет использоваться ровно один раз, то в этом, конечно, есть смысл» (курсив мой).

Дэвид действительно считал, что менее понятный и более трудоемкий код в данном конкретном случае является преимуществом, и с энтузиазмом ухватился за термин *синтаксический уксус*. Спустя примерно два месяца он поместил в свой блог одно из самых знаменитых рассуждений о концепции (ниже приводится выдержка):

> *В спорах о проектировании языков и сред уже давно прижился термин «синтаксическая глазурь». Речь идет о превращении идиом в соглашения, о пропаганде единого стиля, поскольку он обладает несомненными достоинствами: красотой, краткостью и простотой использования. Все мы любим синтаксическую глазурь и приветствуем в ней все: вселяющие ужас пропасти, головокружительные вершины и кремовую серединку. Именно это делает языки, подобные Ruby, такими сладкими по сравнению с более прямолинейными альтернативами.*
>
> *Но мы нуждаемся не в одном лишь сахаре. Хороший дизайн не только поощряет правильное использование, но и препятствует неправильному. Если мы можем украшать какой-то стиль или подход синтаксической глазурью, чтобы поспособствовать его использованию, то почему бы не сдобрить кое-что синтаксическим уксусом, дабы воспрепятствовать неразумному применению. Это делается реже, но оттого не становится менее важным...*
>
> *http://www.loudthinking.com/arc/2006_10.html*

Ресурсы, ассоциированные только с контроллером

Слово «ресурс», будучи существительным, наводит на мысль о таблицах и записях в базе данных. Однако в REST ресурс не обязательно должен один в один отображаться на модель `ActiveRecord`. Ресурсы – это высокоуровневые абстракции сущностей, доступных через веб-приложение. Операции базы данных – лишь один из способов сохранять и извлекать данные, необходимые для генерации представлений ресурсов.

Ресурс в REST необязательно также напрямую отображать на контроллер, по крайней мере теоретически. В ходе обсуждения параметров `:path_prefix` и `:controller` метода `map.resources` вы видели, что при желании можно предоставлять REST-службы, для которых публичные идентификаторы (URI) вообще не соответствуют именам контроллеров.

А веду я к тому, что иногда возникает необходимость создать набор маршрутов к ресурсам и связанный с ними контроллер, которые не соответствуют никакой модели в приложении. Нет ничего плохого в пол-

ном комплекте ресурс/контроллер/модель, где все имена соответствуют друг другу. Но бывают случаи, когда представляемые ресурсы можно инкапсулировать в контроллер, но не в модель.

Для аукционного приложения примером может служить контроллер сеансов. Предположим, что в файле `routes.rb` есть такая строка:

```
map.resource :session
```

Она отображает URL `/session` на контроллер `SessionController` как одиночный ресурс, тем не менее модели `Session` не существует (кстати, ресурс правильно определен как одиночный, потому что с точки зрения пользователя существует только *один* сеанс).

Зачем идти по пути REST при аутентификации? Немного подумав, вы осознаете, что сеансы пользователей можно создавать и уничтожать. Сеанс создается, когда пользователь регистрируется, и уничтожается, когда он выходит. Значит принятый в Rails REST-совместимый подход сопоставления действия и представления `new` с действием `create` годится! Форму регистрации пользователя можно рассматривать как форму создания сеанса, находящуюся в файле шаблона `session/new.rhtml` (листинг 4.8).

Листинг 4.8. REST-совместимая форма регистрации

```
<h1>Регистрация</h1>
<% form_for :user, :url => session_path do |f| %>
<p>Имя: <%= f.text_field :login %></p>
<p>Пароль: <%= f.password_field :password %></p>
<%= submit_tag "Log in" %>
<% end %>
```

Когда эта форма отправляется, данные обрабатываются методом `create` контроллера сеансов, который показан в листинге 4.9.

Листинг 4.9. REST-совместимое действие регистрации

```
def create
  @user = User.find_by_login(params[:user][:login])
  if @user and @user.authorize(params[:user][:password])
    flash[:notice] = "Добро пожаловать, #{@user.first_name}!"
    redirect_to home_url
  else
    flash[:notice] = "Неправильное имя или пароль."
    redirect_to :action => "new"
  end
end
```

Это действие ничего не пишет в базу данных, но имя `create` адекватно, поскольку оно создает сеанс. Более того, если позже вы решите, что сеансы следует-таки хранить в базе данных, то уже будет готово подходящее место для обработки.

Надо ясно понимать, что CRUD как философия именования действий и CRUD как набор фактических операций с базой данных иногда могут существовать независимо друг от друга и что средства обработки ресурсов в Rails полезно ассоциировать с контроллером, для которого нет соответствующей модели. Создание сеанса – не самый убедительный пример REST-совместимой практики, поскольку REST требует, чтобы передача представлений ресурсов осуществлялась без сохранения состояния. Но это неплохая иллюстрация того, как и почему принимаются проектные решения, касающиеся маршрутов и ресурсов, но не затрагивающие приложение в целом.

Присваивать действиям имена, согласованные с CRUD, – в общем случае правильная идея. Если вы выполняете много операций создания и удаления, то регистрацию пользователя проще представлять себе как создание сеанса, чем изобретать для нее отдельную семантическую категорию. Вместо того чтобы вводить новую концепцию «пользователь регистрируется», просто считайте, что речь идет о частном случае старой концепции «создается сеанс».

Различные представления ресурсов

Одна из заповедей REST состоит в том, что компоненты построенной на принципах REST системы обмениваются между собой *представлениями* ресурсов. Различие между ресурсами и их представлениями весьма существенно.

Как клиент или потребитель служб REST, вы получаете от сервера не сам ресурс, а его представление. Вы также передаете представления серверу. Так, операция отправки формы посылает серверу представление ресурса, а вместе с ним запрос, например PUT, диктующий, что это представление следует использовать как основу для обновления ресурса. Представления можно назвать единой валютой в сфере управления ресурсами.

Метод respond_to

В Rails возможность возвращать различные представления основана на использовании метода `respond_to` в контроллере, который, как вы видели, позволяет формировать разные ответы в зависимости от желания пользователя. Кроме того, при создании маршрутов к ресурсам вы автоматически получаете механизм распознавания URL, оканчивающихся точкой и параметром `:format`.

Предположим, например, что в файле маршрутов есть маршрут `map.resources :auctions`, а логика контроллера `AuctionsController` выглядит примерно так:

```
def index
  @auctions = Auction.find(:all)
  respond_to do |format|
    format.html
    format.xml { render :xml => @auctions.to_xml }
  end
end
```

Теперь появилась возможность соединиться с таким URL: `http://localhost:3000/auctions.xml`.

Система маршрутизации обеспечит выполнение действия `index`. Она также распознает суффикс `.xml` в конце маршрута и пойдет по ветви `respond_to`, которая возвращает XML-представление.

Разумеется, все это относится к этапу распознавания URL. А как быть, если вы хотите сгенерировать URL, заканчивающийся суффиксом `.xml`?

Форматированные именованные маршруты

Система маршрутизации дает также варианты именованных маршрутов к ресурсу с модификатором `.:format`. Пусть нужно получить ссылку на XML-представление ресурса. Этого можно достичь с помощью варианта REST-совместимого именованного маршрута с префиксом `formatted_`:

```
<%= link_to "XML-версия этого аукциона",
     formatted_auction_path(@auction, "xml") %>
```

В результате генерируется следующая HTML-разметка:

```
<a href="/auctions/1.xml">XML-версия этого аукциона</a>
```

При щелчке по этой ссылке срабатывает относящаяся к XML ветвь блока `respond_to` в действии `show` контроллера `auctions`. Возвращаемая XML-разметка может выглядеть в броузере не очень эстетично, но сам именованный маршрут к вашим услугам.

Круг замкнулся: вы можете генерировать URL, соответствующие конкретному типу ответу, и обрабатывать запросы на получение различных типов ответа с помощью метода `respond_to`. А можно вместо этого указать тип желаемого ответа с помощью заголовка `Accept` в запросе. Таким образом, система маршрутизации и надстроенные над ней средства построения маршрутов к ресурсам дают набор мощных и лаконичных инструментов для дифференциации запросов и, следовательно, для генерирования различных представлений.

Набор действий в Rails для REST

Встроенные в Rails средства поддержки REST сводятся, в конечном итоге, к именованным маршрутам и действиям контроллеров, на которые они указывают. Чем больше вы работаете со средствами Rails для

REST, тем лучше понимаете назначение каждого из семи REST-совместимых действий. Разумеется, в разных контроллерах (и приложениях) они используются по-разному. Тем не менее, поскольку количество действий конечно, а их роли довольно четко определены, у каждого действия есть ряд более-менее постоянных свойств и присущих только ему характеристик.

Ниже мы рассмотрим каждое из семи действий, приведя примеры и комментарии. Мы уже встречались с ними ранее, особенно в главе 2 «Работа с контроллерами», но сейчас вы познакомитесь с предысторией, почувствуете характеристические особенности каждого действия и поймете, какие вопросы возникают при выборе любого из них.

index

Обычно действие `index` дает представление множественной формы ресурса (или *набора*). Как правило, представление, порождаемое этим действием, общедоступно и достаточно обще. Действие `index` предъявляет миру наиболее нейтральное представление.

Типичное действие `index` выглядит примерно так:

```
class AuctionsController < ApplicationController

  def index
    @auctions = Auction.find(:all)
  end

  ...

end
```

Шаблон представления отображает общедоступные сведения о каждом аукционе со ссылками на детальную информацию о самом аукционе и профиле продавца.

Хотя `index` лучше всего считать открытым действием, иногда возникают ситуации, когда необходимо отобразить представление набора, недоступное широкой публике. Например, у пользователя должна быть возможность посмотреть список всех своих заявок, но при этом видеть чужие списки запрещено.

В этом случае наилучшая стратегия состоит в том, чтобы «закрыть дверь» как можно позже. Вам на помощь придет REST-совместимая маршрутизация.

Пусть нужно сделать так, чтобы каждый пользователь мог видеть историю своих заявок. Можно было бы профильтровать результат работы действия `index` контроллера `bids` с учетом текущего пользователя (`@user`). Проблема, однако, в том, что тем самым мы исключаем использование этого действия для более широкой аудитории. Что если нужно

получить открытое представление текущего набора заявок с наивысшей ценой предложения? А, быть может, даже переадресовать на представление index аукционов? Идея в том, чтобы сохранять максимально открытый доступ настолько долго, насколько это возможно.

Сделать это можно двумя способами. Один из них – проверить, зарегистрировался ли пользователь, и соответственно решить, что показывать. Но такой подход здесь не пройдет. Во-первых, зарегистрировавшийся пользователь может захотеть увидеть более общедоступное представление. Во-вторых, чем больше зависимостей от состояния на стороне сервера мы сможем устранить или консолидировать, тем лучше.

Поэтому будем рассматривать два списка заявок не как открытую и закрытую версию одного и того же ресурса, а как два разных ресурса. Это различие можно инкапсулировать прямо в маршрутах:

```
map.resources :auctions do |auctions|
  auctions.resources :bids, :collection => { :manage => :get }
end
```

Теперь контроллер заявок bids можно организовать так, что доступ будет изящно разбит на уровни, задействуя при необходимости фильтры и устранив ветвление в самих действиях:

```
class BidsController < ApplicationController

  before_filter :load_auction
  before_filter :check_authorization, :only => :manage

  def index
    @bids = Bid.find(:all)
  end

  def manage
    @bids = @auction.bids
  end

  ...

  protected

    def load_auction
      @auction = Auction.find(params[:auction_id])
    end

    def check_authorization
      @auction.authorized?(current_user)
    end
end
```

Мы четко разделили ресурсы /bids и /bids/manage, а также роли, которые они играют в приложении.

Говорит Кортенэ...

Некоторые разработчики полагают, что использование фильтров для загрузки данных – *преступление против всего хорошего и чистого*. Если ваш коллега или начальник пребывает в этом убеждении, включите поиск в действие, устроенное примерно так:

```
def show
  @auction = Auction.find(params[:id])
  unless auction.authorized?(current_user)
    ... # доступ запрещен
  end
end
```

Альтернативный способ – добавить метод в класс `User`, поскольку за авторизацию должен отвечать объект, представляющий пользователя:

```
class User < ActiveRecord::Base
  def find_authorized_auction(auction_id)
    auction = Auction.find(auction_id)
    return auction.authorized?(self) && auction
  end
end
```

И вызовите его из действия контроллера `AuctionController`:

```
def show
  @auction = current_user.find_authorized_auction
(params[:id]) else
  raise ActiveRecord::RecordNotFound
  end
end
```

Можно даже добавить метод в модель `Auction`, поскольку именно эта модель управляет доступом к данным.

```
def self.find_authorized(id, user)
  auction = find(id)
  return auction.authorized?(user) && auction
end
```

С точки зрения именованных маршрутов, мы теперь имеем ресурсы `bids_url` и `manage_bids_url`. Таким образом, мы сохранили общедоступную, лишенную состояния грань ресурса `/bids` и инкапсулировали зависящее от состояния поведение в отдельный подресурс `/bids/manage`. Не пугайтесь, если такой образ мыслей с первого раза не показался вам естественным, – это нормально при освоении REST.

Если бы я занимал в отношении REST догматическую позицию, то счел бы странным и даже отвратительным включать в обсуждение касающихся REST приемов саму идею инкапсуляции поведения, зависяще-

го от состояния, поскольку REST-совместимые запросы по определению не должны зависеть от состояния. Однако мы показали, что имеющиеся в Rails средства поддержки REST могут, так сказать, постепенно убавляться в ситуациях, когда необходимо отойти от строго следующего принципам REST интерфейса.

show

REST-совместимое действие `show` относится к одиночному ресурсу. Обычно оно интерпретируется как представление информации об одном объекте, одном элементе набора. Как и `index`, действие `show` активируется при получении GET-запроса.

Типичное, можно сказать классическое, действие `show` выглядит примерно так:

```
class AuctionController < ApplicationController
  def show
    @auction = Auction.find(params[:id])
  end
end
```

Конечно, действие `show` может полагаться на фильтры `before` как на способ, позволяющий не загружать показываемый ресурс явно в самом действии. Возможно, вам необходимо отличать (скажем, исходя из маршрута) общедоступные профили от профиля текущего пользователя, который может допускать модификацию и содержать дополнительную информацию.

Как и в случае действия `index`, полезно делать действие `show` максимально открытым, а административные и привилегированные представления переносить либо в отдельный контроллер, либо в отдельное действие.

destroy

Доступ к действиям `destroy` обычно предполагает наличие административных привилегий, хотя, конечно, все зависит от того, что именно вы удаляете. Для защиты действия `destroy` можно написать код, представленный в листинге 4.10.

Листинг 4.10. Защита действия destroy

```
class UsersController < ApplicationController
  before_filter :admin_required, :only => :destroy
```

Типичное действие `destroy` могло бы выглядеть следующим образом (предполагая, что пользователь `@user` уже загружен в фильтре `before`):

```
def destroy
  @user.destroy
  flash[:notice] = "Пользователь удален!"
  redirect_to users_url
end
```

Такой подход можно отразить в простом административном интерфейсе:

```
<h1>Пользователи</h1>
<% @users.each do |user| %>
  <p><%= link_to h(user.whole_name), user_path(user) %>
  <%= link_to("delete", user_path(user), :method => :delete) if
  current_user.admin? %></p>
<% end %>
```

Ссылка `delete` присутствует, только если текущий пользователь является администратором.

На самом деле самое интересное в последовательности REST-совместимого удаления в Rails происходит в представлении, которое содержит ссылки `to` на действие. Ниже приведена HTML-разметка одной итерации цикла. Предупреждение: она длиннее, чем можно было бы ожидать.

```
<p><a href="http://localhost:3000/users/2">Emma Knight Peel</a>
  <a href="http://localhost:3000/users/2" onclick="var f =
document.createElement('form'); f.style.display = 'none';
this.parentNode.appendChild(f); f.method = 'POST'; f.action =
this.href;var m = document.createElement('input');
m.setAttribute('type', 'hidden'); m.setAttribute('name', '_method');
m.setAttribute('value', 'delete'); f.appendChild(m);f.submit();return
false;">Удалить</a>)</p>
```

Почему так много кода – да еще на JavaScript! – для двух маленьких ссылочек? Первая ссылка обрабатывается быстро – она просто ведет к действию `show` для данного пользователя. А причина, по которой вторая ссылка получилась такой длинной, состоит в том, что отправка методом `DELETE` потенциально опасна. Rails хочет максимально затруднить подлог таких ссылок и сделать все, чтобы эти действия нельзя было выполнить случайно, например в результате обхода вашего сайта пауком или роботом. Поэтому, когда вы задаете метод `DELETE`, в HTML-документ встраивается целый JavaScript-сценарий, который обертывает ссылку в форму. Поскольку роботы не отправляют форм, это обеспечивает коду дополнительный уровень защиты.

new и create

Вы уже видели, что в Rails для REST действия `new` и `create` идут рука об руку. «Новый ресурс» – это просто виртуальная сущность, которую еще нужно создать. Соответственно, действие `new` обычно выводит форму, а действие `create` создает новую запись исходя из данных формы.

Пусть требуется, чтобы пользователь мог создавать (то есть открывать) аукцион. Тогда вам понадобится:

1. Действие `new` для отображения формы.
2. Действие `create`, которое создаст новый объект `Auction` из данных, введенных в форму, и затем выведет представление (результат действия `show`) этого аукциона.

У действия new работы немного. На самом деле ему вообще ничего не надо делать. Как и всякое пустое действие, его можно опустить. При этом Rails все равно поймет, что вы хотите выполнить рендеринг представления new.html.erb.

Шаблон представления new.html.erb мог бы выглядеть, как показано в листинге 4.11. Обратите внимание, что некоторые поля ввода отнесены к пространству имен :item (благодаря методу-помощнику fields_for), а другие – к пространству имен :auction (из-за метода-помощника form_for). Объясняется это тем, что лот и аукцион в действительности создаются в тандеме.

Листинг 4.11. Форма для создания нового аукциона

```
<h1>Создать новый аукцион</h1>
<%= error_messages_for :auction %>

<% form_for :auction, :url => auctions_path do |f| %>
  <% fields_for :item do |i| %>
    <p>Описание лота: <%= i.text_field "description" %></p>
    <p>Производитель лота: <%= i.text_field "maker" %></p>
    <p>Материал лота: <%= i.text_field "medium" %></p>
    <p>Год выпуска лота: <%= i.text_field "year" %></p>
  <% end %>
  <p>Резервировать: <%= f.text_field "reserve" %></p>
  <p>Шаг торгов: <%= f.text_field "incr" %></p>
  <p>Начальная цена: <%= f.text_field "starting_bid" %></p>
  <p>Время окончания: <%= f.datetime_select "end_time" %>
  <%= submit_tag "Создать" %>
<% end %>
```

Действие формы выражено с помощью именованного маршрута auctions в сочетании с тем фактом, что для формы автоматически генерируется POST-запрос.

После отправки заполненной формы наступает время главного события: действия create. В отличие от new, у этого действия есть работа:

```
def create
  @auction = current_user.auctions.build(params[:auction])
  @item = @auction.build_item(params[:item])
  if @auction.save
    flash[:notice] = "Аукцион открыт!"
    redirect_to auction_url(@auction)
  else
    render :action => "new"
  end
end
```

Наличие пространств имен "auction" и "item" для полей ввода позволяет нам воспользоваться обоими с помощью хеша params, чтобы создать новый объект Auction из ассоциации текущего пользователя с аукционами и одновременно объект Item методом build_item. Это удобный спо-

соб работы сразу с двумя ассоциированными объектами. Если по какой-то причине метод `@auction.save` завершится с ошибкой, ассоциированный объект не будет создан, поэтому беспокоиться об очистке нет нужды.

Если же операция сохранения завершается успешно, будут созданы и аукцион, и лот.

edit и update

Подобно операциям `new` и `create`, операции `edit` и `update` выступают в паре: `edit` отображает форму, а `update` обрабатывает введенные в нее данные.

Формы для редактирования и ввода новой записи очень похожи (можно поместить большую часть кода в частичный шаблон и включить его в обе формы; оставляем это в качестве упражнения для читателя). Вот как мог бы выглядеть шаблон `edit.html.erb` для редактирования лота:

```
<h1>Редактировать лот</h1>
<% form_for :item, :url => item_path(@item),
            :html => { :method => :put } do |item| %>
  <p>Описание: <%= item.text_field "description" %></p>
  <p>Производитель: <%= item.text_field "maker" %></p>
  <p>Материал: <%= item.text_field "medium" %></p>
  <p>Год выпуска: <%= item.text_field "year" %></p>
  <p><%= submit_tag "Сохранить изменения" %></p>
<% end %>
```

Основное отличие от формы для задания нового лота (листинг 4.11) заключается в используемом именованном маршруте и в том, что для действия `update` следует задать метод запроса PUT. Это послужит диспетчеру указанием на необходимость вызвать метод обновления.

Заключение

В этой главе мы рассмотрели непростую тему, касающуюся применения принципов REST к проектированию приложений Rails, в основном с точки зрения системы маршрутизации и действий контроллера. Мы узнали, что основой Rails для REST является метод `map.resources` в файле маршрутов, а также о многочисленных параметрах, позволяющих структурировать приложение именно так, как должно. Как выяснилось, в ряде мест Дэвид и команда разработчиков ядра Rails разлили синтаксический уксус, чтобы помешать нам пойти по ложному пути.

Один из сложных аспектов написания и сопровождения серьезных приложений Rails – понимание системы маршрутизации и умение находить ошибки, которые вы, без сомнения, будете допускать в ходе повседневной работы. Эта тема настолько важна для любого разработчика на платформе Rails, что мы посвятили ей целую главу.

5

Размышления о маршрутизации в Rails

Вы находитесь в лабиринте, состоящем из похожих
друг на друга извилистых коридоров.

Adventure (компьютерная игра,
популярная в конце 1970-х годов)

В данном контексте «размышления» означает не «раздумья», а исследование, тестирование и поиск причин ошибок. В системе маршрутизации есть немало мест, над которыми можно поразмыслить подобным образом, а также удобные подключаемые модули, которые могут помочь в этом.

Мы не собираемся рассматривать все существующие приемы и подключаемые модули, но познакомим вас с некоторыми важными инструментами, позволяющими убедиться, что ваши маршруты делают именно то, на что вы рассчитывали, а в противном случае понять, в чем ошибка.

Мы также заглянем в исходный код системы маршрутизации, в том числе и средств, ориентированных на поддержку REST-совместимой маршрутизации.

Исследование маршрутов в консоли приложения

Описав полный круг, мы возвращаемся к тому, с чего начали эту книгу, когда говорили о диспетчере, – на этот раз мы воспользуемся консолью приложения для исследования внутренних механизмов работы

уже знакомой нам системы маршрутизации. Это поможет в поиске причин ошибок и позволит лучше понять саму систему.

Распечатка маршрутов

Начнем с перечисления всех имеющихся маршрутов. Для этого необходимо получить текущий объект `RouteSet`:

```
$ ruby script/console
Loading development environment.
>> rs = ActionController::Routing::Routes
```

В ответ будет выведено довольно много информации – распечатка всех определенных в системе маршрутов. Но можно представить эту распечатку в более удобном для восприятия виде:

```
>> puts rs.routes
```

В результате перечень маршрутов будет выведен в виде таблицы:

```
GET    /bids/              {:controller=>"bids", :action=>"index"}
GET    /bids.:format/      {:controller=>"bids", :action=>"index"}
POST   /bids/              {:controller=>"bids", :action=>"create"}
POST   /bids.:format/      {:controller=>"bids", :action=>"create"}
GET    /bids/new/          {:controller=>"bids", :action=>"new"}
GET    /bids/new.:format/ {:controller=>"bids", :action=>"new"}
GET    /bids/:id;edit/     {:controller=>"bids", :action=>"edit"}
GET    /bids/:id.:format;edit {:controller=>"bids", :action=>"edit"}

# и т. д.
```

Возможно, столько информации вам не нужно, но представление ее в подобном виде помогает разобраться. Вы получаете зримое подтверждение факта, что у каждого маршрута есть метод запроса, образец URL и параметры, определяющие пару контроллер/действие.

В таком же формате можно получить и список именованных маршрутов. Но в этом случае имеет смысл немного подправить формат. При переборе вы получаете имя и назначение каждого маршрута. Данную информацию можно использовать для форматирования в виде таблицы:

```
rs.named_routes.each {|name, r| printf("%-30s %s\n", name, r) }; nil
```

`nil` в конце необходимо, чтобы программа `irb` не выводила реальное возвращаемое значение при каждом обращении к `each`, поскольку это выдвинуло бы интересную информацию за пределы экрана.

Результат выглядит примерно так (слегка «причесан» для представления на печатной странице):

```
history    ANY     /history/:id/  {:controller=>"auctions",
                                    :action=>"history"}
new_us     GET     /users/new/    {:controller=>"users", :action=>"new"}
new_auction GET    /auctions/new/ {:controller=>"auctions",
```

```
:action=>"new"}

# и т. д.
```

Идея в том, что можно вывести на консоль разнообразную информацию о маршрутизации, отформатировав ее по собственному разумению. Ну а что насчет «необработанной» информации? Исходная распечатка также содержала несколько важных элементов:

```
#<ActionController::Routing::Route:0x275bb7c
@requirements={:controller=>"bids", :action=>"create"},

@to_s="POST /bids.:format/ {:controller=>\"bids\",
:action=>\"create\"}",
@significant_keys=[:format, :controller, :action],
@conditions={:method=>:post},
@segments=[#<ActionController::Routing::DividerSegment:0x275d65c
@raw=true, @is_optional=false, @value="/">,
#<ActionController::Routing::StaticSegment:0x275d274
@is_optional=false, @value="bids">,
#<ActionController::Routing::DividerSegment:0x275ce78 @raw=true,
@is_optional=false, @value=".">,
#<ActionController::Routing::DynamicSegment:0x275cdc4
@is_optional=false, @key=:format>,
#<ActionController::Routing::DividerSegment:0x275c798 @raw=true,
@is_optional=true, @value="/">]>
```

Анатомия объекта Route

Самый лучший способ понять, что здесь происходит, – взглянуть на представление маршрута в формате YAML (Yet Another Markup Language). Ниже приведен результат работы операции `to_yaml`, снабженный комментариями. Объем информации велик, но, изучив ее, вы узнаете много нового. Можно сказать, рентгеном просветите способ конструирования маршрута.

```
# Все это - объект Route

-- !ruby/object:actionController::Routing::Route

# Этот маршрут распознает только PUT-запросы.
conditions:
  :method: :put

# Главная цепочка событий в процедуре распознавания и точки для подключения
# механизма сопоставления в процедуре генерации.
requirements:
  :controller: bids
  :action: update

# Сегменты. Это формальное определение строки-образца.
# Учитывается все, включая разделители (символы косой черты).
```

```
# Отметим, что каждый сегмент – это экземпляр того или иного класса:
# DividerSegment, StaticSegment или DynamicSegment.
# Читая дальше, вы сможете реконструировать возможные значения образца.

# Обратите внимание на автоматически вставленное поле regexp, которое
# ограничивает множество допустимых значений сегмента :id.
segments:
- !ruby/object:actionController::Routing::DividerSegment
  is_optional: false
  raw: true
  value: /
- !ruby/object:actionController::Routing::StaticSegment
  is_optional: false
  value: auctions
- !ruby/object:actionController::Routing::DividerSegment
  is_optional: false
  raw: true
  value: /
- !ruby/object:actionController::Routing::DynamicSegment
  is_optional: false
  key: :auction_id
- !ruby/object:actionController::Routing::DividerSegment
  is_optional: false
  raw: true
  value: /
- !ruby/object:actionController::Routing::StaticSegment
  is_optional: false
  value: bids
- !ruby/object:actionController::Routing::DividerSegment
  is_optional: false
  raw: true
  value: /
- !ruby/object:actionController::Routing::DynamicSegment
  is_optional: false
   key: :id
  regexp: !ruby/regexp /[^\/;.,?]+/
- !ruby/object:actionController::Routing::DividerSegment
  is_optional: true
  raw: true
  value: /
significant_keys:
- :auction_id
- :id
- :controller
- :action

# (Это должно находиться в одной строке; разбито на две только для
# удобства форматирования.)
to_s: PUT /auctions/:auction_id/bids/:id/
          {:controller=>"bids" :action=>"update"}
```

Хранение сегментов в виде набора позволяет системе маршрутизации выполнять распознавание и генерацию, поскольку сегменты можно перебирать как для сопоставления с поступившим URL, так и для вывода URL (в последнем случае сегменты используются в качестве трафарета).

Конечно, об устройстве механизма работы с маршрутами можно узнать еще больше, заглянув в исходный код. Очень далеко мы заходить не будем, но познакомимся с файлами `routing.rb` и `resources.rb` в дереве `ActionController`. Там вы найдете определения классов `Routing`, `RouteSet`, различных классов `Segment` и многое другое. Если хотите подробнее узнать, как это все работает, обратитесь к серии статей в блоге Джеймиса Бака, одного из членов команды разработчиков ядра Rails[1].

Но этим использование консоли не ограничивается – вы можете непосредственно выполнять операции распознавания и генерации URL.

Распознавание и генерация с консоли

Чтобы вручную выполнить с консоли распознавание и генерацию, сначала зададим в качестве контекста текущего сеанса объект `RouteSet` (если вы никогда не встречались с таким приемом, предоставляем случай познакомиться с интересным применением IRB):

```
$ ./script/console
Loading development environment.
>> irb ActionController::Routing::Routes
>>
```

Вызывая команду `irb` в текущем сеансе работы с IRB, мы говорим, что объектом по умолчанию – `self` – будет выступать набор маршрутов. Это позволит меньше печатать в дальнейшем при вводе команд.

Чтобы узнать, какой маршрут генерируется при заданных параметрах, достаточно передать эти параметры методу `generate`. Ниже приведено несколько примеров с комментариями.

Начнем с вложенных маршрутов к ресурсам-заявкам. Действие `create` генерирует URL набора; поле `:id` в нем отсутствует. Но для организации вложенности имеется поле `:auction_id`.

```
>> generate(:controller => "bids", :auction_id => 3, :action =>
 "create")
=> "/auctions/3/bids"
```

Далее следует два маршрута к заявкам `bids`, вложенных в `users`. В обоих случаях (`retract` и `manage`) указания подходящего имени действия достаточно для включения в путь URL дополнительного сегмента.

```
>> generate(:controller => "bids", :user_id => 3, :id => 4, :action =>
 "retract")
```

[1] *http://weblog.jamisbuck.org/2006/10/4/under-the-hood-route-recognition-in-rails.*

```
=> "/users/3/bids/4/retract"
>> generate(:controller => "bids", :user_id => 3, :action => "manage")
=> "/users/3/bids/manage"
```

Не забыли про действие `history` контроллера `auctions`, которое выводит историю заявок? Вот как сгенерировать URL для него:

```
>> generate(:controller => "auctions", :action => "history", :id => 3)
=> "/history/3"
```

В следующих двух примерах иллюстрируется маршрут `item_year`, которому в качестве параметра нужно передать год, записанный четырьмя цифрами. Отметим, что генерация выполняется неправильно, если год не соответствует образцу, – значение года *добавляется в виде строки запроса*, а не включается в URL в виде сегмента пути:

```
>> generate(:controller => "items", :action => "item_year", :year =>
1939)
=> "/item_year/1939"
>> generate(:controller => "items", :action => "item_year", :year =>
19393)
=> "/items/item_year?year=19393"
```

Можно поступить и наоборот, то есть начать с путей и посмотреть, как система распознавания маршрутов преобразует их в контроллер, действие и параметры.

Вот что происходит для маршрута верхнего уровня, определенного в файле `routes.rb`:

```
>> recognize_path("/")
=> {:controller=>"auctions", :action=>"index"}
```

Аналогичный результат получается для маршрута к ресурсу `auctions`, который записан во множественном числе и отправлен методом `GET`:

```
>> recognize_path("/auctions", :method => :get)
=> {:controller=>"auctions", :action=>"index"}
```

Для запроса методом `POST` результат будет иным – он маршрутизируется к действию `create`:

```
>> recognize_path("/auctions", :method => :post)
=> {:controller=>"auctions", :action=>"create"}
```

Та же логика применима к множественному `POST`-запросу во вложенном маршруте:

```
>> recognize_path("/auctions/3/bids", :method => :post)
=> {:controller=>"bids", :action=>"create", :auction_id=>"3"}
```

Нестандартные действия тоже распознаются и преобразуются в нужный контроллер, действие и параметры:

```
>> recognize_path("/users/3/bids/1/retract", :method => :get)
=> {:controller=>"bids", :user_id=>"3", :action=>"retract", :id=>"1"}
```

Маршрут к истории заявок ведет на контроллер auctions:

```
>> recognize_path("/history/3")
=> {:controller=>"auctions", :action=>"history", :id=>"3"}
```

Маршрут item_year распознает только пути с четырехзначными числами в позиции :year. Во втором из показанных ниже примеров система сообщает об ошибке – подходящего маршрута не существует.

```
>> recognize_path("/item_year/1939")
=> {:controller=>"items", :action=>"item_year", :year=>"1939"}
>> recognize_path("/item_year/19393")
ActionController::RoutingError: no route found to match
"/item_year/19393" with {}
```

Консоль и именованные маршруты

С консоли можно выполнять и именованные маршруты. Проще всего это сделать, включив модуль ActionController::UrlWriter и задав произвольное значение для принимаемого по умолчанию хоста (просто чтобы подавить ошибки):

```
>> include ActionController::UrlWriter
=> Object
>> default_url_options[:host] = "example.com"
=> "example.com"
```

Теперь можно вызвать именованный маршрут и посмотреть, что вернет система, то есть какой URL будет сгенерирован:

```
>> auction_url(1)
=> "http://example.com/auctions/1"
>> formatted_auction_url(1,"xml")
=> "http://example.com/auctions/1.xml"
>> formatted_auctions_url("xml")
=> "http://example.com/auctions.xml"
```

Как всегда, консоль приложения помогает и учиться, и отлаживать программу. Если в одном окне открыть файл routes.rb, а в другом консоль, то вы сможете очень быстро разобраться во взаимосвязи механизмов распознавания и генерации маршрутов. К сожалению, консоль не умеет автоматически перезагружать таблицу маршрутов. Даже метод reload!, похоже, не заставляет ее перечитать файл.

Для обучения и быстрого тестирования консоль хороша, но существуют инструменты для более систематического подхода к тестированию маршрутов.

Тестирование маршрутов

Система предоставляет следующие средства для тестирования маршрутов:

- assert_generates
- assert_recognizes
- assert_routing

Третий метод – assert_routing – представляет собой комбинацию первых двух. Вы передаете ему путь и параметры, а он проверяет, чтобы в результате распознавания пути эти параметры действительно получали указанные значения, и наоборот – чтобы из указанных параметров генерировался заданный путь. Тема тестирования и задания маршрутов подробно рассматривается в главах 17 «Тестирование» и 18 «RSpec on Rails».

Вы сами решаете, сколько и каких тестов написать для своего приложения. В идеале комплект тестов должен включать по меньшей мере, все комбинации, встречающиеся в приложении. Если хотите посмотреть на достаточно полный набор тестов маршрутизации, загляните в файл routing.rb в подкаталоге test установленной библиотеки ActionPack. При последнем подсчете в нем была 1881 строка. Эти строки предназначены для тестирования самой среды, так что от вас не требуется (и не рекомендуется!) дублировать тесты в своем приложении. Однако их изучение (как, впрочем, и других файлов для тестирования Rails) может навести на полезные мысли и уж точно послужит иллюстрацией процедуры разработки, управляемой тестами.

Замечание о синтаксисе аргументов

Не забывайте действующее в Ruby правило, касающееся использования хешей в списках аргументов: если последний аргумент в списке – хеш, то фигурные скобки можно опускать.

Поэтому так писать *можно*:

```
assert_generates(user_retract_bid_path(3,1),
                 :controller => "bids",
                 :action => "retract",
                 :id => "1", :user_id => "3")
```

Если хеш встречается не в последней позиции, то фигурные скобки обязательны. Следовательно, так писать *должно*:

```
assert_recognizes({:controller => "auctions",
                   :action => "show",
                   :id => auction.id.to_s },
                  auction_path(auction))
```

Здесь последним аргументом является auction_path(auction), поэтому фигурные скобки вокруг хеша необходимы.

Если вы получаете загадочные сообщения о синтаксических ошибках в списке аргументов, проверьте, не нарушено ли это правило.

Подключаемый модуль Routing Navigator

Рик Олсон (Rick Olson), один из разработчиков ядра Rails, написал подключаемый модуль Routing Navigator, который позволяет прямо в броузере получить ту же информацию, что при исследовании маршрутов с консоли, только еще лучше.

Для установки модуля Routing Navigator, находясь в каталоге верхнего уровня приложения Rails, выполните следующую команду:

```
./script/plugin install
http://svn.techno-weenie.net/projects/plugins/routing_navigator/
```

Теперь нужно сообщить одному или нескольким контроллерам, что они должны показывать искомую информацию о маршрутизации. Например, можно добавить такую строку:

```
routing_navigator :on
```

в начало определения класса контроллера в файле `auction_controller.rb` (куда вы поместили фильтры `before` и другие методы класса). Разумеется, это следует делать только на этапе разработки – оставлять информацию о маршрутизации в промышленно эксплуатируемом приложении не стоит.

При щелчке по кнопке Recognize (Распознать) или Generate (Генерировать) вы увидите поля, в которые можно ввести путь (в случае распознавания) или параметры (в случае генерации), а затем выполнить соответствующую операцию. Идея та же, что при работе с консолью, но оформление более элегантное.

Еще одна кнопка называется Routing Navigator. Щелкнув по ней, вы попадете на новую страницу со списком всех маршрутов (как обычных, так и именованных), которые определены в вашем приложении. Над списком маршрутов расположены уже описанные выше кнопки, позволяющие ввести путь или параметры и выполнить распознавание или генерацию.

Список всех имеющихся маршрутов может оказаться весьма длинным. Но его можно отфильтровать, воспользовавшись еще одним полем – YAML to filter routes by requirements (YAML для фильтрации маршрутов по требованию). Например, если ввести в него строку `controller: bids` и щелкнуть по кнопке Filter, то в нижней части появится список маршрутов, которые относятся к контроллеру `bids`.

Модуль Routing Navigator – это великолепный инструмент для отладки, поэтому имеет смысл потратить некоторое время на его изучение.

Заключение

Вот и подошло к концу наше путешествие в мир маршрутизации Rails, как с поддержкой REST, так и без оной. Разрабатывая приложения для Rails, вы, конечно, выберете наиболее приемлемые для себя идиомы и приемы работы; а, если понадобятся примеры, к вашим услугам огромный массив уже написанного кода. Если не забывать о фундаментальных принципах, то умение будет возрастать, что не замедлит сказаться на элегантности и логичности ваших программ.

Счастливо выбрать маршрут!

6

Работа с ActiveRecord

Объект, обертывающий строку таблицы или представления базы данных, инкапсулирует доступ к базе и добавляет к данным логику предметной области.

Мартин Фаулер,
«Архитектура корпоративных программных приложений»

Паттерн ActiveRecord, выявленный Мартином Фаулером в основополагающей книге *Patterns of Enterprise Architecture*[1], отображает один класс предметной области на одну таблицу базы данных, а один экземпляр класса – на одну строку таблицы. Хотя этот простой подход применим и не во всех случаях, он обеспечивает удобную среду для доступа к базе данных и сохранения в ней объектов.

Среда ActiveRecord в Rails включает механизмы для представления моделей и их взаимосвязей, операций CRUD (Create, Read, Update, Delete), сложных поисков, контроля данных, обратных вызовов и многого другого. Она опирается на принцип «примата соглашения над конфигурацией», поэтому проще всего ее применять, когда уже на этапе создания схемы новой базы данных вы следуете определенным соглашениям. Однако ActiveRecord предоставляет и средства конфигурирования, позволяющие адаптировать его к унаследованным базам данных, в которых соглашения Rails не применялись.

---

[1] Мартин Фаулер «Архитектура корпоративных программных приложений», Вильямс, 2007 год.

В основном докладе на конференции, посвященной рождению Rails, в 2006 году, Мартин Фаулер сказал, что в Ruby on Rails паттерн Active Record внедрен настолько глубоко, насколько никто не мог и предполагать. На этом примере показано, чего можно добиться, если всецело посвятить себя достижению идеала, в качестве которого в Rails выступает простота.

Основы

Для полноты начнем с изложения самых основ работы ActiveRecord. Первое, что нужно сделать при создании нового класса модели, – объявить его как подкласс `ActiveRecord::Base`, применив синтаксис расширения Ruby:

```
class Client < ActiveRecord::Base
end
```

По принятому в ActiveRecord соглашению класс `Client` отображается на таблицу `clients`. О том, как Rails понимает, что такое множественное число, см. раздел «Приведение к множественному числу» ниже. По тому же соглашению, ActiveRecord ожидает, что первичным ключом таблицы будет колонка с именем `id`. Она должна иметь целочисленный тип, а сервер должен автоматически инкрементировать ключ при создании новой записи. Отметим, что в самом классе нет никаких упоминаний об имени таблицы, а также об именах и типах данных колонок.

Каждый экземпляр класса ActiveRecord обеспечивает доступ к данным одной строки соответствующей таблицы в объектно-ориентированном стиле. Колонки строки представляются в виде атрибутов объекта; при этом применяются простейшие преобразования типов (то есть типу varchar соответствует строка Ruby, типам даты/времени – даты Ruby и т. д.). Проверка наличия значения по умолчанию не производится. Типы атрибутов выводятся из определения колонок в таблице, ассоциированной с классом. Добавление, удаление и изменение самих атрибутов или их типов реализуется изменением описания таблицы в схеме базы данных.

Если сервер Rails запущен в *режиме разработки*, то изменения в схеме базы данных отражаются на объектах ActiveRecord немедленно, и это видно в веб-броузере. Если же изменения внесены в схему в режиме работы с консолью Rails, то они автоматически *не* подхватываются, но это можно сделать вручную, набрав на консоли команду `reload!`.

Путь Rails состоит в том, чтобы генерировать, а не писать трафаретный код. Поэтому вам почти никогда не придется ни создавать файл для своего класса модели, ни вводить его объявление. Гораздо проще воспользоваться для этой цели встроенным в Rails генератором моделей.

Например, позволим генератору моделей создать класс `Client` и посмотрим, какие в результате появятся файлы:

```
$ script/generate model client
      exists app/models/
      exists test/unit/
      exists test/fixtures/
      create app/models/client.rb
      create test/unit/client_test.rb
      create test/fixtures/clients.yml
      exists db/migrate
      create db/migrate/002_create_clients.rb
```

Файл, в котором находится новый класс модели, называется client.rb:

```
class Client < ActiveRecord::Base
end
```

Просто и красиво. Посмотрим, что еще было создано. Файл **client_test.rb** содержит заготовку для автономных тестов:

```
require File.dirname(__FILE__) + '/../test_helper'

class ClientTest < Test::Unit::TestCase
  fixtures :clients

  # Заменить настоящими тестами.
  def test_truth
    assert true
  end
end
```

Комментарий предлагает заменить метод `test_truth` настоящими тестами. Но пока мы просто знакомимся со сгенерированным кодом, поэтому пойдем дальше. Отметим, что класс `ClientTest` ссылается на файл *фикстуры* (fixture) **clients.yml**:

```
# О фикстурах см. http://ar.rubyonrails.org/classes/Fixtures.html
one:
  id: 1
two:
  id: 2
```

Какие-то идентификаторы... Автономные тесты и фикстуры рассматриваются в главе 17 «Тестирование».

И наконец, имеется файл миграции с именем **002_create_clients.rb**:

```
class CreateClients < ActiveRecord::Migration
  def self.up
    create_table :clients do |t|
      # t.column :name, :string
    end
  end
```

```
  def self.down
    drop_table :clients
  end
end
```

Механизм миграций в Rails позволяет создавать и развивать схему базы данных, без него вы не получили бы никаких моделей ActiveRecord (точнее, они были бы очень скучными). Коли так, рассмотрим миграции более подробно.

Говорит Кортенэ...

ActiveRecord – прекрасный пример «Золотого пути» Rails. Это означает, что, оставаясь в рамках наложенных ограничений, можно зайти очень далеко. Но стоит свернуть в сторону, и вы, скорее всего, завязнете в грязи. Золотой путь подразумевает соблюдение ряда соглашений, в частности, присваивание таблицам имен во множественном числе (users).

Разработчики, недавно открывшие для себя Rails, а также приверженцы конкурирующих веб-платформ ругаются, что их заставляют именовать таблицы определенным образом, на уровне базы данных нет никаких ограничений, обработка внешних ключей абсолютно неправильна, в системах масштаба предприятия первичные ключи должны быть составными, и т. д. и т. п.

Но перестаньте хныкать – все это не более чем умолчания, которые можно переопределить в одной строке кода или с помощью подключаемого модуля.

Миграции

Никуда не уйти от того факта, что со временем схема базы данных эволюционирует. Добавляются новые таблицы, изменяются имена колонок, что-то удаляется – в общем, вы понимаете, о чем я. Если разработчики не готовы строго соблюдать соглашения и придерживаться определенной дисциплины, то синхронизация схемы базы данных с кодом приложений традиционно становится очень трудоемкой задачей.

Миграции в Rails помогают развивать схему базы данных приложения (ее еще называют DDL-описанием[1]) без уничтожения и нового создания базы после каждого изменения. А это означает, что вы не теряете данные, появившиеся в процессе разработки. Быть может, иногда это

[1] DDL (Data Definition Language) – язык определения данных. – *Примеч. перев.*

и не так важно, но обычно очень удобно. При выполнении миграции достаточно описать изменения, необходимые для перехода от одной версии схемы к другой – следующей или *предыдущей*.

Создание миграций

Rails предлагает генератор для создания миграций. Вот текст справки по нему[1]:

```
$ script/generate migration
Порядок вызова: script/generate migration MigrationName [флаги]
Информация о Rails:
-v, --version          Вывести номер версии Rails и завершиться.
-h, --help             Вывести это сообщение и завершиться.
Общие параметры:
-p, --pretend          Выполнять без внесения изменений.
-f, --force            Перезаписывать существующие файлы.
-s, --skip             Пропускать существующие файлы.
-q, --quiet            Подавить нормальную печать.
-t, --backtrace        Отладка: выводить трассировку стека в случае ошибок.
-c, --svn              Модифицировать файлы в системе subversion.
                       (Примечание: команда svn должна находиться по одному
                       из просматриваемых путей.)
Описание:
   Генератор миграций создает заглушку для новой миграции базы данных.
   В качестве аргумента передается имя миграции. Имя может быть задано
   в ВерблюжьейНотации или с_подчерками.
   Генератор создает класс миграции в каталоге db/migrate, указывая в начале
   имени порядковый номер.
Пример:
   ./script/generate migration AddSslFlag
   Если уже существуют 4 миграции, то для миграция AddSslFlag будет создан
файл db/migrate/005_add_ssl_flag.rb.
```

Как видите, от вас требуется только задать понятное имя миграции в нотации CamelCase[2], а генератор сделает все остальное. Напрямую мы вызываем генератор только при необходимости изменить атрибуты колонок в существующей таблице.

---

[1] Оригинальный генератор печатает справку на английском языке. Для удобства читателя она переведена. – *Примеч. перев.*

[2] Нотация CamelCase или camelCase (вариативность написания заглавных букв существенна) получила название ВерблюжьейНотации. Имена идентификаторов, состоящие из нескольких слов, записываются так, что слова не разделяются никакими знаками, но каждое слово начинается с заглавной буквы, например dateOfBirth или DateOfBirth. При этом первое слово может начинаться с заглавной или строчной буквы – в зависимости от соглашения. Визуально имеется ряд «горбов», как у верблюда. Отсюда и английское название. – *Примеч. перев.*

Как уже отмечалось выше, другие генераторы, в частности генератор моделей, тоже создают сценарии миграции, если только не указан флаг `--skip-migration`.

Именование миграций

Последовательность миграций определяется простой схемой нумерации, отраженной в именах файлов; генератор миграций формирует порядковые номера автоматически.

По соглашению, имя файла начинается с трехзначного номера версии (дополненного слева нулями), за которым следует имя класса миграции, отделенное знаком подчерка. (Примечание: имя файла *обязано* точно соответствовать имени класса, иначе процедура миграции закончится с ошибкой.)

Генератор миграций определяет порядковый номер следующей миграции, справляясь со специальной таблицей в базе данных, за которую отвечает Rails. Она называется `schema_info` и имеет очень простую структуру:

```
mysql> desc schema_info;
+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| version | int(11) | YES  |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
1 row in set (0.00 sec)
```

Таблица содержит всего одну колонку и одну строку, в которой хранится текущий номер миграции приложения.

Содержательная часть имени миграции оставлена на ваше усмотрение, но многие известные мне разработчики Rails стараются отражать в нем операцию над схемой (в простых случаях) или хотя бы намекнуть, для чего миграция предназначена (в более сложных).

Подвохи миграций

Если вы пишете свои программы для Rails в одиночку, то у схемы порядковой нумерации имен никаких подвохов нет, так что можете смело пропустить этот раздел. Проблемы начинаются, когда над одним проектом работают несколько программистов, особенно большие коллективы. Речь идет не о проблемах миграции API самой среды Rails – именно сопровождение изменяющейся схемы базы данных представляет собой сложную и пока не до конца решенную задачу.

Вот что пишет о проблемах миграции, с которыми приходилось сталкиваться, мой старый приятель по компании ThoughtWorks Джей Филдс (Jay Fields), не раз возглавлявший большие коллективы разработчиков на платформе Rails, в своем блоге:

Миграции – это прекрасно, но за них приходится расплачиваться. При работе в большом коллективе (а моя теперешняя команда состоит из 14 человек и продолжает расти) случаются конфликты миграций. Проблему можно отчасти решить с помощью договоренностей, но нет сомнений, что миграции могут стать узким местом. Кроме того, сам процесс создания миграции в большой команде может протекать болезненно. Прежде чем создавать новую миграцию, вы должны убедиться, что коллеги сохранили свои миграции в системе управления версиями. Далее наилучшее развитие событий – создать миграцию, которая ничего не изменяет, и сразу же поставить ее на учет. Это гарантирует, что вы не помешаете создавать миграции другим членам команды, однако не всегда можно ограничиться одним лишь добавлением таблицы. Если миграция изменяет структуру базы данных, то часто некоторые тесты перестают работать. Очевидно, не следует ставить миграцию на учет, пока все тесты не заработают нормально, но на это может потребоваться время. В течение всего этого времени больше никто не сможет вносить изменения в базу данных[1].

По-другому та же проблема проявляется, когда требуется внести изменения сразу в несколько ветвей программы. Это вообще оказывается кошмаром (дополнительную информацию на эту тему см. в комментариях к процитированной записи в блоге Джея).

К сожалению, у данной проблемы нет простого решения, разве что спроектировать базу от начала до конца еще до реализации приложения. Но на этом пути куча своих проблем (настолько много, что мы даже затрагивать их не будем). Могу лишь по собственному опыту сказать, что заранее продумать пусть грубую, но достаточно полную схему базы данных, а уж потом нырять с головой в кодирование весьма полезно.

Также весьма желательно известить коллег о том, что вы собираетесь приступить к созданию новой миграции, чтобы потом два человека не

Говорит Себастьян...

Мы применяем подключаемый к `svn` сценарий (идею подал Конор Хант), который контролирует добавление новых миграций в репозиторий и предотвращает появление одинаковых номеров.

Еще один подход – назначить человека, отвечающего за миграции. Тогда разработчики могли бы локально создавать и тестировать миграции, а потом отправлять их по электронной почте «координатору», который проверит результат и присвоит правильные номера.

[1] *http://jayfields.blogspot.com/2006/12/rails-migrations-with-large-team-part.html.*

Говорит Кортенэ...

Выгоды от хранения схемы базы данных в системе управления версиями намного перевешивают трудности, которые возникают, когда члены команды спонтанно изменяют схему. Хранение всех версий кода снимает неприятный вопрос: «Кто добавил это поле?».

Как обычно, на эту тему написано несколько подключаемых к Rails модулей. Один из них написал я сам и назвал IndependentMigrations. Проще говоря, он позволяет иметь несколько миграций с одним и тем же номером. Другие модули допускают идентификацию миграций по временному штампу. Подробнее о моем модуле и альтернативах можно прочитать по адресу *http://blog.caboo.se/articles/2007/3/27/independent-migrations-plugin*.

попытались поставить на учет миграцию с одним и тем же номером, что приведет к печальным последствиям.

Migration API

Но вернемся к самому Migration API. Вот как будет выглядеть созданный ранее файл 002_create_clients.rb после добавления определений четырех колонок в таблицу `clients`:

```
class CreateClients < ActiveRecord::Migration
  def self.up
    create_table :clients do |t|
      t.column :name, :string
      t.column :code, :string
      t.column :created_at, :datetime
      t.column :updated_at, :datetime
    end
  end

  def self.down
    drop_table :clients
  end
end
```

Как видно из этого примера, директивы миграции находятся в определениях двух методов класса: `self.up` и `self.down`. Если перейти в каталог проекта и набрать команду `rake db:migrate`, будет создана таблица `clients`. По ходу миграции Rails печатает информативные сообщения, чтобы было видно, что происходит:

```
$ rake db:migrate
(in /Users/obie/prorails/time_and_expenses)
```

```
== 2 CreateClients: migrating
=============================================
-- create_table(:clients)
   -> 0.0448s
== 2 CreateClients: migrated (0.0450s)
================================
```

Обычно выполняется только код метода `up`, но, если вы захотите *откатиться* к предыдущей версии схемы, то метод `down` опишет, что нужно сделать, чтобы отменить действия, произведенные в методе `up`.

Для отката необходимо выполнить то же самое задание `migrate`, но передать в качестве параметра номер версии, на которую необходимо откатиться: `rake db:migrate VERSION=1`.

create_table(name, options)

Методу `create_table` необходимы по меньшей мере имя таблицы и блок, содержащий определения колонок. Почему мы задаем идентификаторы символами, а не просто в виде строк? Работать будет и то, и другое, но для ввода символа нужно на одно нажатие меньше[1].

В методе `create_table` сделано серьезное и обычно оказывающееся истинным предположение: необходим автоинкрементный целочисленный первичный ключ. Именно поэтому вы не видите его объявления в списке колонок. Если это допущение не выполняется, придется передать методу `create_table` некоторые параметры в виде хеша.

Например, как определить простую связующую таблицу, в которой есть два внешних ключа, но ни одного первичного? Просто задайте параметр `:id` равным `false` – в виде булева значения, а не символа! Тогда миграция не будет автоматически генерировать первичный ключ:

```
create_table :ingredients_recipes, :id => false do |t|
  t.column :ingredient_id, :integer
  t.column :recipe_id, :integer
end
```

Если же вы хотите, чтобы колонка, содержащая первичный ключ, называлась не id, передайте в параметре `:id` какой-нибудь символ. Пусть, например, корпоративный стандарт требует, чтобы первичные ключи именовались с учетом объемлющей таблицы: *tablename_id*. Тогда ранее приведенный пример нужно записать так:

```
create_table :clients, :id => :clients_id do |t|
  t.column :name, :string
  t.column :code, :string
  t.column :created_at, :datetime
```

[1] Если вы находите, что взаимозаменяемость символов и строк в Rails несколько раздражает, то не одиноки.

```
    t.column :updated_at, :datetime
  end
```

Параметр `:force => true` говорит миграции, что нужно *предварительно удалить определяемую таблицу, если она существует*. Но будьте осторожны, поскольку при запуске в режиме эксплуатации это может привести к потере данных (чего вы, возможно, не хотели). Насколько я знаю, параметр `:force` наиболее полезен, когда нужно привести базу данных в известное состояние, но при повседневной работе он редко бывает нужен.

Параметр `:options` позволяет включить дополнительные инструкции в SQL-предложение CREATE и полезен для учета специфики конкретной СУБД. В зависимости от используемой СУБД можно задать, например, кодировку, схему сортировки, комментарии, минимальный и максимальный размер и многие другие свойства.

Параметр `:temporary => true` сообщает, что нужно создать таблицу, существующую только во время выполнения миграции. В сложных случаях это может оказаться полезным для переноса больших наборов данных из одной таблицы в другую, но вообще-то применяется нечасто.

> **Говорит Себастьян...**
>
> Малоизвестно, что можно удалять файлы из каталогов миграции (сохраняя самые свежие), чтобы размер каталога `db/migrate` оставался на приемлемом уровне. Можно, скажем, переместить старые миграции в каталог `db/archived_migrations` или сделать еще что-то в этом роде.
>
> Если вы хотите быть абсолютно уверены, что ваш код допускает развертывание с нуля, можете заменить миграцию с наименьшим номером миграцией «создать заново все», основанной на текущем содержимом файла `schema.rb`.

Определение колонок

Добавить в таблицу колонки можно либо с помощью метода `column` внутри блока, ассоциированного с предложением `create_table`, либо методом `add_column`. Второй метод отличается от первого только тем, что принимает в качестве первого аргумента имя таблицы, куда добавляется колонка.

```
create_table :clients do |t|
  t.column :name, :string
end

add_column :clients, :code, :string
add_column :clients, :created_at, :datetime
```

Первый (или второй) параметр – имя колонки, а второй (или третий) – ее тип. В стандарте SQL92 определены фундаментальные типы данных, но в каждой конкретной СУБД имеются свойственные только ей расширения стандарта.

Если вы знакомы с типами данных в СУБД, то предыдущий пример мог вызвать недоумение: почему колонка имеет тип `string`, хотя в базах данных такого типа нет, а есть типы `char` и `varchar`?

Отображение типов колонок

Причина, по которой для колонки базы данных объявлен тип `string`, заключается в том, что миграции в Rails по идее должны быть независимыми от СУБД. Вот почему можно (и я это делал) вести разработку на СУБД Postgres, а развертывать систему на Oracle.

Полное обсуждение вопроса о том, как выбирать правильные типы данных, выходит за рамки этой книги. Но полезно иметь под рукой справку от отображении обобщенных типов в миграциях на конкретные типы для различных СУБД. В табл. 6.1 такое отображение приведено для СУБД, которые наиболее часто встречаются в приложениях Rails.

Таблица 6.1. Отображение типов данных для СУБД, которые наиболее часто встречаются в приложениях Rails

Тип миграции Класс Ruby	MySQL	Postgres	SQLite	Oracle
:binary String	blob	bytea	blob	blob
:boolean Boolean	tinyint(1)	boolean	Boolean	number(1)
:date Date	date	date	date	date
:datetime Time	datetime	timestamp	datetime	date
:decimal BigDecimal	decimal	decimal	decimal	decimal
:float Float	float	float	float	number
:integer Fixnum	int(11)	integer	integer	number(38)
:string String	varchar(255)	character varying(255)	varchar(255)	varchar(255)
:text String	text	clob(32768)	text	clob
:time Time	time	time	time	date
:timestamp Time	datetime	timestamp	datetime	date

Для каждого класса-адаптера соединения существует хеш native_database_types, устанавливающий описанное в табл. 6.1 отображение. Если вас заинтересуют отображения для других СУБД, можете открыть код соответствующего адаптера и найти в нем вышеупомянутый хеш. Так, в классе SQLServerAdapter из файла sqlserver_adapter.rb хеш native_database_types выглядит следующим образом:

```
def native_database_types
  {
    :primary_key => "int NOT NULL IDENTITY(1, 1) PRIMARY KEY",
    :string      => { :name => "varchar", :limit => 255 },
    :text        => { :name => "text" },
    :integer     => { :name => "int" },
    :float       => { :name => "float", :limit => 8 },
    :decimal     => { :name => "decimal" },
    :datetime    => { :name => "datetime" },
    :timestamp   => { :name => "datetime" },
    :time        => { :name => "datetime" },
    :date        => { :name => "datetime" },
    :binary      => { :name => "image"},
    :boolean     => { :name => "bit"}
  }
end
```

Дополнительные характеристики колонок

Во многих случаях одного лишь указания типа данных недостаточно. Все объявления колонок принимают еще и следующие параметры:

```
:default => value
```

Задает значение по умолчанию, которое записывается в данную колонку вновь созданной строки. Явно указывать null необязательно, достаточно просто опустить этот параметр.

```
:limit => size
```

Задает размер для колонок типа string, text, binary и integer. Семантика зависит от конкретного типа данных. В общем случае ограничение на строковые типы относится к числу символов, а для других типов речь идет о количестве байтов, выделяемых в базе для хранения значения.

```
:null => true
```

Делает колонку обязательной, добавляя ограничение not null, которое проверяется на уровне СУБД.

Количество знаков в десятичной записи

Для колонок, объявленных как :decimal, можно задавать следующие характеристики:

```
:precision => number
```

Здесь `precision` (точность) – общее число цифр в десятичной записи числа.

```
:scale => number
```

Здесь `scale` (масштаб) – число цифр *справа* от десятичного знака. Например, для числа 123,45 точность равна 5, а масштаб – 2. Очевидно, масштаб не может быть больше точности.

Примечание

Для десятичных типов велика опасность потери данных при переносе с одной СУБД на другую Например, поскольку в Oracle и SQL Server принимаемая по умолчанию точность различается, в процессе переноса может происходить отбрасывание знаков и, как следствие, изменение числового значения. Поэтому разумно всегда задавать точность и масштаб явно.

Подводные камни при выборе типов колонок

Выбор типа колонки не всегда очевиден и зависит как от используемой СУБД, так и от требований, предъявляемых приложением:

- **:binary.** В зависимости от способа использования хранение в базе двоичных данных может сильно снизить производительность. Rails загружает объекты из базы данных целиком, поэтому присутствие больших двоичных атрибутов в часто употребляемых моделях заметно увеличивает нагрузку на сервер базы данных;
- **:boolean.** Булевы значения в разных СУБД хранятся по-разному. Иногда для представления true и false используются целые значения 1 и 0, а иногда – символы T и F. Rails прекрасно справляется с задачей отображения таких значений на «родные» для Ruby объекты `true` и `false`, поэтому задумываться о реальной схеме хранения не нужно. Прямое присваивание атрибутам значений 1 или F, в конкретном случае, может быть, и сработает, но такая практика считается антипаттерном;
- **:date**, **:datetime** и **:time**. Сохранение дат в СУБД, где нет встроенного типа даты, например в Microsoft SQL Server, может стать проблемой. Rails отображает тип `datetime` на класс Ruby `Time`, который не позволяет представить даты ранее 1 января 1970 года. Но ведь имеющийся в Ruby класс `DateTime` умеет работать с более ранними датами, так почему же он не используется в Rails? Дело в том, что класс `Time` реализован на C и потому работает очень быстро, тогда как `DateTime` написан на чистом Ruby и, следовательно, медленнее.

Чтобы заставить ActiveRecord отображать тип даты на `DateTime` вместо `Time`, поместите код из листинга 6.1 в какой-нибудь файл, находящийся в каталоге **lib/** и затребуйте его из сценария **config/environment.rb** с помощью `require`.

Листинг 6.1. Отображение даты на тип DateTime вместо Time

```
require 'date'
# Это необходимо сделать, потому что класс Time не поддерживает
# даты ранее 1970 года…

class ActiveRecord::ConnectionAdapters::Column
  def self.string_to_time(string)
    return string unless string.is_a?(String)
    time_array = ParseDate.parsedate(string)[0..5]
    begin
      Time.send(Base.default_timezone, *time_array)
    rescue
      DateTime.new(*time_array) rescue nil
    end
  end
end
```

- **:decimal.** В старых версиях Rails (до 1.2) тип `:decimal` с фиксированной точкой не поддерживался, поэтому во многих ранних приложениях Rails некорректно использовался тип `:float`. Числа с плавающей точкой по природе своей неточны, поэтому для большинства бизнес-приложений следует выбирать тип **:decimal**, а не **:float**;
- **:float.** Не пользуйтесь типом `:float` для хранения денежных сумм[1] и вообще любых данных, для которых необходима фиксированная точность. Поскольку числа с плавающей точкой дают хорошую аппроксимацию, простое хранение данных в таком формате, наверное, приемлемо. Проблемы начинаются, когда вы пытаетесь выполнять над числами математические действия или операции сравнения, поскольку внести таким образом ошибку в приложение до смешного просто, а найти ее ох как тяжело;
- **:integer** и **:string**. Есть не так уж много неприятностей, с которыми можно столкнуться при использовании целых и строковых типов. Это основные кирпичики любого приложения, и многие разработчики опускают задание размера, получая по умолчанию 11 цифр и 255 знаков соответственно.

Следует помнить, что при попытке сохранить значение, которое не помещается в отведенную для него колонку (для строк – по умолчанию 255 знаков), вы не получите никакого уведомления об ошибке. Строка будет просто молча обрезана. Убеждайтесь, что длина введенных пользователем данных не превосходит максимально допустимой.

- **:text.** Есть сообщения о том, что текстовые поля снижают производительность запросов настолько, что в сильно нагруженных приложениях это может превратиться в проблему. Если вам абсолютно необходимы текстовые данные в приложениях, для которых быстродействие критично, помещайте их в отдельную таблицу;

[1] По адресу *http://dist.leetsoft.com/api/money/* размещен рекомендуемый класс Money с открытым исходным текстом.

- **:timestamp.** В версии Rails 1.2 при создании новых записей ActiveRecord может не очень хорошо работать, когда значение колонки по умолчанию генерируется функцией, как для данных типа timestamp в Postgres. Проблема в том, что Rails не исключает такие колонки из предложения `insert`, как следовало бы, а задает для них «значение» `null`, что может приводить к игнорированию значения по умолчанию.

Нестандартные типы данных

Если в вашем приложении необходимы типы данных, специфичные для конкретной СУБД (например, тип `:double`, обеспечивающий более высокую точность, чем `:float`), включите в файл **config/environment.rb** директиву `config.active_record.schema_format = :sql`, чтобы заставить Rails сохранять информацию о схеме в «родном» для данной СУБД формате DDL, а не в виде кросс-платформенного кода на Ruby, записываемого в файл **schema.rb**.

«Магические» колонки с временными штампами

К колонкам типа timestamp Rails применяет *магию*, если они названы определенным образом. Active Record автоматически снабжает операции *создания* временным штампом, если в таблице есть колонка с именем `created_at` или `created_on`. То же самое относится к операциям *обновления*, если в таблице есть колонка с именем `updated_at` или `updated_on`.

Отметим, что в файле миграции тип колонок `created_at` и `updated_at` должен быть задан как `datetime`, а не `timestamp`.

Автоматическую проштамповку можно глобально отключить, задав в файле **config/environment.rb** следующую переменную:

```
ActiveRecord::Base.record_timestamps = false
```

За счет наследования данный код отключает временные штампы для всех моделей, но можно сделать это и избирательно в конкретной модели, если установить переменную `record_timestamps` в `false` только для нее. По умолчанию временные штампы выражены в местном поясном времени, но, если задать переменную `ActiveRecord::Base.default_timezone = :utc`, будет использовано время UTC.

Методы в стиле макросов

Большинство существенных классов, которые вы пишете, программируя приложение для Rails, сконфигурированы для вызова *в стиле макросов* (в определенных кругах это также называется *предметно-ориентированным языком*, или *DSL* – domain-specific language). Основная идея заключается в том, что в начале класса размещается максимально понятный блок кода, конфигурация которого сразу видна.

Для размещения вызовов в стиле макросов именно в начале файла есть веская причина. Эти методы *декларативно* сообщают Rails, как управлять экземплярами, выполнять контроль данных, делать обратные вызовы и взаимодействовать с другими моделями. Часто в таких методах используется *метапрограммирование*, то есть на этапе выполнения они добавляют в класс то или иное поведение в форме дополнительных переменных экземпляра или методов.

Объявление отношений

Рассмотрим, например, класс `Client`, в котором объявлены некоторые отношения. Не пугайтесь, если смысл этих объявлений вам не ясен; мы подробно поговорим об этом в главе 7 «Ассоциации в ActiveRecord». Сейчас я хочу лишь показать, что имею в виду, говоря о *стиле макросов*:

```
class Client < ActiveRecord::Base
  has_many :billing_codes
  has_many :billable_weeks
  has_many :timesheets, :through => :billable_weeks

end
```

Благодаря трем объявлениям `has_many` класс `Client` получает по меньшей мере три новых атрибута – прокси-объекты, позволяющие интерактивно манипулировать ассоциированными наборами.

Я припоминаю, как когда-то в первый раз обучал своего друга – опытного программиста на Java – основам Ruby и Rails. Несколько минут он пребывал в сильном замешательстве, а потом я буквально увидел, как в его голове зажглась лампочка, и он провозгласил: «О! Так это же методы!».

Ну конечно, это самые обычные вызовы методов в контексте объекта класса. Мы опустили скобки, чтобы подчеркнуть декларативность. Это не более чем вопрос стиля, но лично мне скобки в таком фрагменте кажутся неуместными:

```
class Client < ActiveRecord::Base
  has_many(:billing_codes)
  has_many(:billable_weeks)
  has_many(:timesheets, :through => :billable_weeks)
end
```

Когда интерпретатор Ruby загружает файл **client.rb**, он выполняет методы `has_many`, которые, еще раз подчеркну, определены как *методы класса* `ActiveRecord::Base`. Они выполняются в контексте *класса* `Client` и добавляют в него атрибуты, которые в дальнейшем становятся доступны *экземплярам* класса `Client`. Новичку такая модель программирования может показаться странной, но очень скоро она становится «вторым я» любого программиста Rails.

Примат соглашения над конфигурацией

Примат соглашения над конфигурацией – один из руководящих принципов Rails. Если вы следуете принятым в Rails соглашениям, то почти ничего не придется явно конфигурировать. И здесь мы наблюдаем разительный контраст с тем, сколько приходится конфигурировать для запуска даже простейшего приложения в других технологиях.

Дело не в том, что всякое новое приложение Rails уже создается с готовой *умалчиваемой* конфигурацией, отражающей применяемые соглашения. Нет – соглашения уже *впечатаны* в среду, жестко зашиты в ее поведение, и это-то подразумеваемое по умолчанию поведение вы и переопределяете с помощью явного конфигурирования, когда возникает необходимость.

Стоит также отметить, что, как правило, конфигурирование производится очень близко к тому, что конфигурируется. Объявления *ассоциаций*, *проверок* и *обратных вызовов* располагаются в начале большинства моделей ActiveRecord.

Подозреваю, что многие из нас впервые занялись конфигурированием (в противовес соглашению), чтобы изменить соответствие между именем класса и таблицы базы данных, поскольку по умолчанию Rails предполагает, что имя таблицы образуется как множественное число от имени класса. И, поскольку такое соглашение застает врасплох многих начинающих разработчиков Rails, рассмотрим эту тему, перед тем как двигаться дальше.

Приведение к множественному числу

В Rails есть класс `Inflector`, в обязанность которого входит преобразование строк (слов) из единственного числа в множественное, имен классов – в имена таблиц, имен классов с указанием модуля – в имена без модуля, имен классов – во внешние ключи и т. д. (некоторые операции имеют довольно смешные имена, например `dasherize`).

Принимаемые по умолчанию окончания для формирования единственного и множественного числа неисчисляемых имен существительных хранятся в файле **inflections.rb** в каталоге установки Rails. Как правило, класс `Inflector` успешно находит имя таблицы, образуемое приведением имени класса к множественному числу, но иногда случаются оплошности. Для многих новых пользователей Rails это становится первым камнем преткновения, но причин для паники нет. Можно заранее проверить, как `Inflector` будет реагировать на те или иные слова. Для этого понадобится лишь консоль Rails, которая, кстати, является одним из лучших инструментов при работе с Rails.

Чтобы запустить консоль, выполните из командной строки сценарий **script/console**, который находится в каталоге вашего проекта.

```
$ script/console
>> Inflector.pluralize "project"
=> "projects"
>> Inflector.pluralize "virus"
=> "viri"
>> Inflector.pluralize "pensum"
=> "pensums"
```

Как видите, `Inflector` достаточно умен – в качестве множественного числа от virus он правильно выбрал viri. Но, если вы знаете латынь, то, наверное, обратили внимание, что pensum во множественном числе на самом деле пишется как pensa. Понятно, что инфлектор латыни не обучен.

Однако вы *можете* расширить познания инфлектора одним из трех способов:

- добавить новое правило-образец
- описать исключение
- объявить, что некое слово не имеет множественного числа

Лучше всего делать это в файле **config/environment.rb**, где уже имеется прокомментированный пример:

```
Inflector.inflections do |inflect|
  inflect.plural /^(.*)um$/i, '\1a'
  inflect.singular /^(.*)a/i, '\1um'
  inflect.irregular 'album', 'albums'
  inflect.uncountable %w( valium )
end
```

Кстати, в версии Rails 1.2 инфлектор принимает слова, уже записанные *во множественном числе*, и... ничего с ними не делает, что, наверное, самое правильное. Прежние версии Rails вели себя в этом отношении не так разумно.

```
>> "territories".pluralize
=> "territories"
>> "queries".pluralize
=> "queries"
```

Если хотите посмотреть на длинный список существительных, которые `Inflector` правильно приводит к множественному числу, загляните в файл **activesupport/test/inflector_test.rb**. Я нашел в нем немало интересного, например:

```
"datum" => "data",
"medium" => "media",
"analysis" => "analyses"
```

Надо ли сообщать разработчикам ядра об ошибках в работе Inflector

Майкл Козярский (Michael Koziarski), один из разработчиков ядра Rails, пишет, что сообщать о проблемах с классом `Inflector` не следует: «Инфлектор практически заморожен; до выхода версии 1.0 мы добавляли в него много правил для исправления ошибок, чем привели в ярость тех, кто называл свои таблицы согласно старым вариантам. Если необходимо, добавляйте исключения в файл environment.rb самостоятельно».

Задание имен вручную

Разобравшись с инфлектором, вернемся к конфигурированию классов моделей ActiveRecord. Методы `set_table_name` и `set_primary_key` позволяют обойти соглашения Rails и явно задать имя таблицы и имя колонки, содержащей первичный ключ.

Предположим, к примеру (и только к примеру!), что я вынужден использовать для именования таблиц какое-то мерзкое соглашение, отличающееся от принятого в ActiveRecord. Тогда я мог бы поступить следующим образом:

```
class Client < ActiveRecord::Base
  set_table_name "CLIENT"
  set_primary_key "CLIENT_ID"
end
```

Методы `set_table_name` и `set_primary_key` позволяют выбрать для таблиц и первичных ключей произвольные имена, но тогда вы должны явно указать их в классе модели. Для одной модели это всего лишь пара лишних строчек, но в большом приложении оказывается ненужным усложнением, поэтому не делайте этого без острой необходимости.

Если вы не можете сами диктовать соглашения о выборе имен, например, когда схемами управляет отдельная группа администрирования базы данных, то выбора, наверное, нет. Но при наличии свободы действий лучше придерживаться соглашений Rails. Возможно, это покажется непривычным, зато позволит сэкономить время и избежать ненужных сложностей.

Унаследованные схемы именования

Если вы работаете с унаследованными схемами, может возникнуть искушение вставлять метод `set_table_name` повсюду, нужен он или не нужен. Но прежде чем у вас появится такая привычка, познакомьтесь

еще с некоторыми возможностями, которые позволят избежать повторений и облегчить себе жизнь.

Чтобы полностью отключить механизм приведения имен таблиц к множественному числу, добавьте в конец файла **config/environment.rb** следующую строку:

```
ActiveRecord::Base.pluralize_table_names = false
```

В классе `ActiveRecord::Base` есть и другие полезные атрибуты, позволяющие сконфигурировать Rails для работы с унаследованными схемами именования:

- `primary_key_prefix_type`. Акцессор для задания префикса, который добавляется в начало имени любого первичного ключа. Если задан параметр `:table_name`, то ActiveRecord будет считать, что первичный ключ называется tableid, а не id. Если же задан параметр `:table_name_with_underscore`, то предполагается, что первичный ключ называется table_id;
- `table_name_prefix`. Иногда к имени таблицы добавляют имя базы данных. Установите этот атрибут, чтобы не включать префикс в имена всех классов модели вручную;
- `table_name_suffix`. Помимо префикса, можно добавить к именам всех таблиц еще и суффикс;
- `underscore_table_names`. Установите в `false`, если не хотите, чтобы ActiveRecord вставляла подчерки между отдельными частями составного имени таблицы.

Определение атрибутов

Список атрибутов, ассоциированных с классов модели ActiveRecord, явно не кодируется. На этапе выполнения модель ActiveRecord получает схему базы данных непосредственно от сервера. Добавление, удаление и изменение атрибутов или их типов производится путем манипулирования самой базой данных – с помощью команд SQL или графических инструментов. Но в идеале для этого следует применять миграции ActiveRecord.

Практическое следствие паттерна ActiveRecord состоит в том, что вы должны определить структуру таблицы базы данных и убедиться, что эта таблица существует, перед тем как начинать работу с моделью. У некоторых программистов такая философия проектирования может вызывать трудности, особенно если они привыкли к проектированию «сверху вниз».

Без сомнения, путь Rails подразумевает, что классы модели имеют тесную связь со схемой базой данных. Но, с другой стороны, помните, что модели могут быть обычными классами Ruby, необязательно расширя-

ющими ActiveRecord::Base. В частности, очень часто классы моделей, не имеющие отношения к ActiveRecord, применяются с целью инкапсуляции данных и логики для уровня представления.

Значения атрибутов по умолчанию

Миграции позволяют задавать значения атрибутов по умолчанию путем передачи параметра :default методу column, но, как правило, это следует делать на уровне модели, а не на уровне базы данных. Значения по умолчанию – часть логики предметной области, поэтому их место – рядом со всей прочей предметной логикой приложения, то есть на уровне модели.

Типичный пример – модель должна возвращать строку "n/a" вместо nil (или пустой строки), если атрибуту еще не присвоено значение. Этот пример достаточно прост, поэтому может служить отправной точкой для разговора о том, как атрибуты возникают на этапе выполнения.

Для начала соорудим простенький тест, описывающий желаемое поведение:

```
class SpecificationTest < Test::Unit::TestCase
  def test_default_string_for_tolerance_should_be_na
    spec = Specification.new
    assert_equal 'n/a', spec.tolerance
  end
end
```

Этот тест, как и следовало ожидать, не проходит. ActiveRecord не предоставляет в модели никаких методов класса для декларативного определения значений по умолчанию. Похоже, придется явно создать акцессор для данного атрибута, который будет возвращать значение по умолчанию.

Обычно с акцессорами атрибутов ActiveRecord разбирается самостоятельно, но в данном случае нам предстоит вмешаться и подставить свой *метод чтения*. Для этого достаточно определить метод с таким же именем, как у атрибута, и воспользоваться оператором or, котовый вернет альтернативу, если @tolerance равно nil:

```
class Specification < ActiveRecord::Base
  def tolerance
    @tolerance or 'n/a'
  end
end
```

Теперь тест проходит. Замечательно. Все сделали? Не совсем. Надо еще протестировать случай, при котором должно возвращаться истинное значение @tolerance. Добавим спецификацию теста с непустым значением @tolerance и изменим имена методов тестирования, сделав их более содержательными.

```
class SpecificationTest < Test::Unit::TestCase
  def test_default_string_for_tolerance_should_return_na_when_nil
    spec = Specification.new
    assert_equal 'n/a', spec.tolerance
  end

  def test_tolerance_value_should_be_returned_when_not_nil
    spec = Specification.new(:tolerance => '0.01mm')
    assert_equal '0.01mm', spec.tolerance
  end
end
```

Опаньки! Второй тест не проходит. Похоже, в любом случае возвращается строка "n/a". Это означает, что в атрибут @tolerance ничего не было записано. Должны ли мы знать о том, что запись в атрибут не производится? Это деталь реализации ActiveRecord или нет?

Тот факт, что в Rails переменные экземпляра наподобие @tolerance не используются для хранения атрибутов модели, – действительно деталь реализации. Но в экземплярах моделей есть два метода write_attribute и read_attribute, которые ActiveRecord предлагает для переопределения акцессоров, подразумеваемых по умолчанию, а это как раз то, что мы пытаемся сделать. Давайте исправим класс Specification:

```
class Specification < ActiveRecord::Base
  def tolerance
    read_attribute(:tolerance) or 'n/a'
  end
end
```

Теперь тест проходит. А как насчет простого примера использования write_attribute?

```
class SillyFortuneCookie < ActiveRecord::Base
  def message=(txt)
    write_attribute(:message, txt + ' in bed')
  end
end
```

Оба примера можно было бы написать и по-другому, воспользовавшись более короткой формой чтения и записи атрибутов с помощью квадратных скобок:

```
class Specification < ActiveRecord::Base
  def tolerance
    self[:tolerance] or 'n/a'
  end
end

class SillyFortuneCookie < ActiveRecord::Base
  def message=(txt)
    self[:message] = txt + ' in bed'
  end
end
```

Сериализованные атрибуты

Одна из самых «крутых» (на мой взгляд) особенностей ActiveRecord – возможность помечать колонки типа text как *сериализованные*. Любой объект (точнее, граф объектов), записываемый в такой атрибут, будет храниться в базе данных в формате YAML – стандартном для Ruby формате сериализации.

> **Говорит Себастьян...**
>
> Максимальный размер колонок типа TEXT составляет 64K. Если сериализованный атрибут оказывается длиннее, не миновать многочисленных ошибок.
>
> С другой стороны, если ваши сериализованные атрибуты оказываются настолько длинными, то стоит еще раз подумать, что вы хотите сделать. Как минимум перенесите такие атрибуты в отдельную таблицу и выберите для них более длинный тип данных, если СУБД позволяет.

CRUD: создание, чтение, обновление, удаление

Акроним CRUD обозначает четыре стандартные операции любой СУБД.

У него несколько негативная окраска, поскольку в английском языке слово crud означает «ненужное барахло». Однако в кругах, связанных с Rails, использование слова CRUD всецело одобряется. Как мы увидим в следующих главах, проектирование функциональности приложений в виде набора CRUD-операций считается самым правильным подходом!

Создание новых экземпляров ActiveRecord

Самый прямолинейный способ создать новый экземпляр модели ActiveRecord – воспользоваться обычным механизмом конструирования в Ruby – методом класса `new`. Вновь созданные объекты могут быть пустыми (если опустить параметры) или с уже установленными, но еще не сохраненными атрибутами. Достаточно передать конструктору хеш, в котором имена ключей соответствуют именам колонок в ассоциированной таблице. В обоих случаях допустимые ключи определяются именами колонок в таблице, поэтому нельзя задать атрибут, которому не соответствуют никакая колонка.

В только что созданном, но еще не сохраненном объекте ActiveRecord имеется атрибут `@new_record`, который можно опросить методом `new_record?`:

```
>> c = Client.new
=> #<Client:0x2515584 @new_record=true, @attributes={"name"=>nil,
"code"=>nil}>
>> c.new_record?
=> true
```

Конструкторы ActiveRecord принимают необязательный блок и используют его для дополнительной инициализации. Этот блок выполняется, когда в экземпляре уже установлены значения всех переданных конструктору атрибутов:

```
>> c = Client.new do |client|
?> client.name = "Nile River Co."
>> client.code = "NRC"
>> end
=> #<Client:0x24e8764 @new_record=true, @attributes={"name"=>"Nile
River Co.", "code"=>"NRC"}>
```

В ActiveRecord имеется также удобный метод класса `create`, который создает новый экземпляр, записывает его в базу данных и возвращает – все в одной операции:

```
>> c = Client.create(:name => "Nile River, Co.", :code => "NRC")
=> #<Client:0x4229490 @new_record_before_save=true, @new_record=false,
@errors=#<ActiveRecord::Errors:0x42287ac @errors={},
@base=#<Client:0x4229490 ...>>, @attributes={"name"=>"Nile River,
Co.", "updated_at"=>Mon Jun 04 22:24:27 UTC 2007, "code"=>"NRC",
"id"=>1, "created_at"=>Mon Jun 04 22:24:27 UTC 2007}>
```

Метод `create` не принимает блок. Должен бы, поскольку это самое естественное место для инициализации объекта перед сохранением, но, увы, не принимает.

Чтение объектов ActiveRecord

Считывать данные из базы в экземпляр объекта ActiveRecord очень легко и удобно. Основной механизм – метод `find`, который скрывает операцию SQL SELECT от разработчика.

Метод find

Искать существующий объект по первичному ключу очень просто. Пожалуй, это одна из первых вещей, которые мы узнаем о Rails, только начиная изучать эту среду. Достаточно вызвать метод `find`, указав ключ искомого экземпляра. Но помните: если такой экземпляр отсутствует, возникнет исключение `RecordNotFound`.

```
>> first_project = Project.find(1)
>> boom_client = Client.find(99)
ActiveRecord::RecordNotFound: Couldn't find Client with ID=99
    from
/vendor/rails/activerecord/lib/active_record/base.rb:1028:in
```

```
'find_one'
    from
/vendor/rails/activerecord/lib/active_record/base.rb:1011:in
'find_from_ids'
    from
/vendor/rails/activerecord/lib/active_record/base.rb:416:in 'find'
    from (irb):
```

Метод `find` понимает также два специальных символа Ruby: `:first` и `:all`:

```
>> all_clients = Client.find(:all)
=> [#<Client:0x250e004 @attributes={"name"=>"Paper Jam Printers",
"code"=>"PJP", "id"=>"1"}>, #<Client:0x250de88
@attributes={"name"=>"Goodness Steaks", "code"=>"GOOD_STEAKS",
"id"=>"2"}>]

>> first_client = Client.find(:first)
=> #<Client:0x2508244 @attributes={"name"=>"Paper Jam Printers",
"code"=>"PJP", "id"=>"1"}>
```

Мне странно, что не существует параметра `:last`, но получить последнюю запись несложно, воспользовавшись параметром `:order`:

```
>> all_clients = Client.find(:first, :order => 'id desc')
=> #<Client:0x2508244 @attributes={"name"=>"Paper Jam Printers",
"code"=>"PJP", "id"=>"1"}>
```

Кстати, для методов Ruby совершенно естественно возвращать значения разных типов в зависимости от параметров, что и иллюстрирует предыдущий пример. В зависимости от того, как вызван метод `find`, вы получаете либо единственный объект ActiveRecord, либо массив таких объектов.

Наконец, метод `find` понимает также массив ключей и возбуждает исключение `RecordNotFound`, если не может найти хотя бы один из них:

```
>> first_couple_of_clients = Client.find(1, 2)
[#<Client:0x24d667c @attributes={"name"=>"Paper Jam Printers",
"code"=>"PJP", "id"=>"1"}>, #<Client:0x24d65b4 @attributes={"name"=>
"Goodness Steaks", "code"=>"GOOD_STEAKS", "id"=>"2"}>]

>> first_few_clients = Client.find(1, 2, 3)
ActiveRecord::RecordNotFound: Couldn't find all Clients with IDs
(1,2,3)
   from /vendor/rails/activerecord/lib/active_record/base.rb:1042:in
 'find_some'
   from /vendor/rails/activerecord/lib/active_record/base.rb:1014:in
 'find_from_ids'
   from /vendor/rails/activerecord/lib/active_record/base.rb:416:in
 'find'
   from (irb):9
```

Чтение и запись атрибутов

Выбрав из базы данных экземпляр модели, вы можете получить доступ к колонкам несколькими способами. Самый простой (и понятный) – воспользоваться оператором «точка» для доступа к атрибуту:

```
>> first_client.name
=> "Paper Jam Printers"
>> first_client.code
=> "PJP"
```

Полезно знать о закрытом методе `read_attribute`, которого мы вскользь коснулись выше. Он удобен, если нужно переопределить акцессор атрибута, подразумеваемый по умолчанию. Для иллюстрации, не выходя из консоли Rails, я *заново открою* класс и переопределю акцессор `name`, чтобы он инвертировал прочитанное из базы значение:

```
>> class Client
>>   def name
>>     read_attribute(:name).reverse
>>   end
>> end
=> nil
>> first_client.name
=> "sretnirP maJ repaP"
```

Мне не составит труда продемонстрировать, *почему* в этом случае необходимо переопределять `read_attribute`:

```
>> class Client
>>   def name
>>     self.name.reverse
>>   end
>> end
=> nil
>> first_client.name
SystemStackError: stack level too deep
       from (irb):21:in 'name'
       from (irb):21:in 'name'
       from (irb):24
```

Как и следовало ожидать, в дополнение к методу `read_attribute` существует метод `write_attribute`, который позволяет изменить значение атрибута:

```
project = Project.new
project.write_attribute(:name, "A New Project")
```

Переопределять методы установки атрибутов для задания нестандартного поведения так же просто, как методы чтения:

```
class Project
  # Описанию проекта нельзя присваивать пустую строку
  def description=(new_value)
```

```
    self[:description] = new_value unless new_value.blank?
  end
end
```

Предыдущий пример иллюстрирует простой способ контроля данных – перед тем как разрешить присваивание, проверяется, пусто ли новое значение. Ниже вы познакомитесь с более удобными подходами к решению этой задачи.

Нотация хеша

Еще один способ доступа к атрибутам заключается в использовании оператора «квадратные скобки», который позволяет обращаться к атрибутам так, как будто это обычный хеш:

```
>> first_client['name']
=> "Paper Jam Printers"
>> first_client[:name]
=> "Paper Jam Printers"
```

Строка или символ

Многие методы в Rails принимают в качестве параметров как символы, так и строки, и это может вносить путаницу. Что правильнее?

Общее правило состоит в том, чтобы использовать символы, когда строка выступает в роли имени, и стро́ки, когда речь идет о значении. Пожалуй, символы правильнее употреблять в качестве ключей хеша и для иных подобных целей.

Здравый смысл подсказывает, что нужно выбрать какое-то одно соглашение и следовать ему во всем приложении, но большинство разработчиков для Rails всюду, где можно, употребляют символы.

Метод attributes

Существует также метод `attributes`, возвращающий хеш, где каждый атрибут представлен своим именем и значением, которое возвращает `read_attribute`. Если вы переопределяете методы чтения и записи атрибутов, следует помнить, что метод `attributes` *не* вызывает переопределенные версии акцессоров чтения, тогда как метод `attributes=` (позволяющий выполнять множественное присваивание) *вызывает* переопределенные версии акцессоров записи.

```
>> first_client.attributes
=> {"name"=>"Paper Jam Printers", "code"=>"PJP", "id"=>1}
```

Возможность получить сразу весь хеш атрибутов бывает полезна, когда требуется перебрать их или передать весь набор другой функции. Отметим, что хеш, который возвращает метод `attributes`, *не является* ссылкой на внутреннюю структуру в объекте ActiveRecord. Это копия, поэтому изменения не отразятся на объекте, от которого копия получена.

```
>> atts = first_client.attributes
=> {"name"=>"Paper Jam Printers", "code"=>"PJP", "id"=>1}
>> atts["name"] = "Def Jam Printers"
=> "Def Jam Printers"
>> first_client.attributes
=> {"name"=>"Paper Jam Printers", "code"=>"PJP", "id"=>1}
```

Для внесения групповых изменений в атрибуты объекта ActiveRecord можно передать хеш методу `attributes`.

Доступ к атрибутам и манипулирование ими до приведения типов

Адаптеры соединений в ActiveRecord извлекают результаты в виде строк, а Rails при необходимости преобразует их в другие типы данных, исходя из типа колонки таблицы. Например, целые типы преобразуются в экземпляры класса `Fixnum` и т. д.

Даже при работе с новым экземпляром объекта ActiveRecord, конструктору которого были переданы строки, при попытке доступа к соответствующим атрибутам их значения будут приведены к нужному типу.

Но иногда хочется прочитать (или изменить) непреобразованные значения атрибутов – это позволяют создать акцессоры `<attribute>_before_type_cast`, которые формируются в модели автоматически.

Пусть, например, требуется получать денежные суммы в виде строк, введенных конечными пользователями. Если вы инкапсулировали такие значения в класс для представления денежных сумм (кстати, настоятельно рекомендую), придется иметь дело с докучливыми знаками доллара и запятыми. В предположении, что в модели `Timesheet` определен атрибут `rate` типа `:decimal`, следующий код уберет ненужные символы, перед тем как выполнять приведение типа для операции сохранения:

```
class Timesheet < ActiveRecord::Base
  before_save :fix_rate

  def fix_rate
    rate_before_type_cast.tr!('$,','')
  end
end
```

Перезагрузка

Метод `reload` выполняет запрос к базе данных и переустанавливает атрибуты объекта ActiveRecord. Ему передается необязательный аргумент `options`, так что можно, например, написать `record.reload(:lock => true)`, чтобы запись перечитывалась под защитой исключительной блокировки (см. раздел «Блокировка базы данных» ниже в этой главе).

Динамический поиск по атрибутам

Поскольку одна из самых распространенных операций в приложениях, работающих с базой данных, – простой поиск по одной или нескольким колонкам, в Rails есть эффективный способ решить эту задачу, не прибегая к параметру `conditions` метода `find`. Работает он благодаря применению имеющегося в Ruby обратного вызова `method_missing`, который выполняется, когда запрошенный метод еще не определен.

Имена методов динамического поиска начинаются с префиксов `find_by_` или `find_all_by_`, обозначающих, что вы хотите получить одно значение или массив соответственно. Семантика аналогична вызову метода `find` с параметром `:first` или `:all`.

```
>> City.find_by_name("Hackensack")
=> #<City:0x3205244 @attributes={"name" => "Hackensack", "latitude" =>
"40.8858330000", "id" => "15942", "longitude" => "-74.0438890000",
"state" => "NJ" }>

>> City.find_all_by_name("Atlanta").collect(&:state)
=> ["GA", "MI", "TX"]
```

Допускается также задавать в имени поискового метода несколько атрибутов, разделяя их союзом and, так что возможно имя `Person.find_by_user_name_and_password` или даже `Payment.find_by_purchaser_and_state_and_country`.

Достоинство динамических методов поиска в том, что запись получается короче и проще для восприятия. Вместо `Person.find(:first, ["user_name = ? AND password = ?", user_name, password])` попробуйте написать просто `Person.find_by_user_name_and_password(user_name, password)`:

```
>> City.find_by_name_and_state("Atlanta", "TX")
=> #<City:0x31faeac @attributes={ "name" => "Atlanta", "latitude" =>
 "33.1136110000", "id" => "25269", "longitude" => "-94.1641670000",
 "state" => "TX"}>
```

Можно даже вызывать динамический метод с параметрами, как обычный метод. `Payment.find_all_by_amount` – не что иное, как `Payment.find_all_by_amount(amount, options)`. А полный интерфейс метода `Person.find_by_user_name` выглядит как `Person.find_by_user_name(user_name, options)`. Поэтому вызывают его так: `Payment.find_all_by_amount(50, :order => "created_on")`.

Тот же самый динамический стиль можно применять для создания объекта, если последний еще не существует. Такой динамический метод называется `find_or_create_by_` и возвращает найденный объект, если он существует; в противном случае создает объект, а потом возвращает его. Если же вы хотите вернуть новую запись без предварительного сохранения, воспользуйтесь методом `find_or_initialize_by_`.

Специальные SQL-запросы

Метод класса `find_by_sql` принимает SQL-запрос на выборку и возвращает массив объектов ActiveRecord, соответствующих найденным строкам. Вот набросок примера, но использовать его в реальных приложениях ни в коем случае нельзя:

```
>> Client.find_by_sql("select * from clients")
=> [#<Client:0x4217024 @attributes={"name"=>"Nile River, Co.",
"updated_at"=>"2007-06-04 22:24:27", "code"=>"NRC", "id"=>"1",
"created_at"=>"2007-06-04 22:24:27"}>, #<Client:0x4216ffc
@attributes={"name"=>"Amazon, Co.", "updated_at"=>"2007-06-04
22:26:22",
"code"=>"AMZ", "id"=>"2", "created_at"=>"2007-06-04 22:26:22"}>]
```

Еще и еще раз подчеркиваю: пользоваться методом `find_by_sql` следует, *только когда без него не обойтись*! Прежде всего таким способом вы снижаете степень переносимости между различными СУБД – при использовании стандартных операций поиска, предоставляемых ActiveRecord, Rails автоматически учитывает различия между СУБД.

Кроме того, в ActiveRecord уже встроена весьма развитая функциональность для абстрагирования предложений `SELECT`, и изобретать ее заново было бы неразумно. Есть много случаев, когда кажется, что без `find_by_sql` не обойтись, однако на самом деле это не так. Типичная ситуация – запрос с предикатом `LIKE`:

```
>> Client.find_by_sql("select * from clients where code like 'A%'")
=> [#<Client:0x4206b34 @attributes={"name"=>"Amazon, Inc.", ...}>]
```

Но оказывается, что оператор `LIKE` легко включить в параметр `conditions`:

```
>> param = "A"
>> Client.find(:all, :conditions => ["code like ?", "#{param}%"])
=> [#<Client:0x41e3594 @attributes={"name"=>"Amazon, Inc...}>] #
Правильно!
```

Rails незаметно для вас *обезвреживает*[1] ваш SQL-запрос при условии, что он параметризован. ActiveRecord выполняет SQL-запросы с помощью метода `connection.select_all`, затем обходит результирующий мас-

[1] Обезвреживание предотвращает атаки внедрением SQL. Дополнительную информацию о таких атаках применительно к Rails см. в статье по адресу *http://www.rorsecurity.info/2007/05/19/sql-injection/*.

сив хешей и для каждой строки вызывает метод `initialize`. Так выглядел бы предыдущий запрос, будь он *непараметризован*:

```
>> param = "A"
>> Client.find(:all, :conditions => ["code like '#{param}%'"])
=> [#<Client:0x41e3594 @attributes={"name"=>"Amazon, Inc...}>] #
Только не это!
```

Обратите внимание на отсутствующий знак вопроса, играющий роль подставляемого символа. Никогда не забывайте, что интерполяция поступивших от пользователя значений в любое предложение SQL – крайне опасное дело! Подумайте, что произойдет, если злонамеренный пользователь инициирует это небезопасное обращение к `find` с таким значением в `param`:

```
"Amazon'; DELETE FROM users;'
```

Как это ни печально, очень немногие хорошо понимают, что такое внедрение SQL. В данном случае лучшим другом вам будет Google.

Кэш запросов

По умолчанию Rails пытается оптимизировать производительность, включая простой *кэш запросов*: хеши, хранящиеся в памяти текущего потока, – по одному на каждое соединение с базой данных (в большинстве приложений Rails будет всего один такой хеш).

Если кэш запросов включен, то при каждом вызове `find` (или любой другой операции выборки) результирующий набор сохраняется в хеше, причем в качестве ключа выступает предложение SQL. Если то же самое предложение встретится еще раз, то для порождения нового набора объектов модели будет использован кэшированный результирующий набор без повторного обращения к базе данных.

Кэширование запросов можно включить вручную, обернув операции в блок `cache`, как в следующем примере:

```
User.cache do
puts User.find(:first)
puts User.find(:first)
puts User.find(:first)
end
```

Заглянув в файл `development.log`, вы найдете такие записи:

```
Person Load (0.000821) SELECT * FROM people LIMIT 1
CACHE (0.000000) SELECT * FROM people LIMIT 1
CACHE (0.000000) SELECT * FROM people LIMIT 1
```

К базе данных было только одно обращение. Проведя такой же эксперимент на консоли без блока `cache`, вы увидите, что будут запротоколированы три разных события `Person Load`.

Операции сохранения и удаления приводят к очистке кэша, чтобы не распространялись экземпляры с уже недействительным состоянием. При необходимости вы можете вручную очистить кэш запросов, вызвав метод класса `clear_query_cache`.

Подключаемый модуль ActiveRecord Context

Рик Олсон вычленил из своего популярного приложения Lighthouse этот подключаемый модуль, позволяющий *инициализировать* кэш запросов набором объектов, который заведомо понадобится. Это очень полезное дополнение к встроенной в ActiveRecord поддержке кэширования.

Дополнительную информацию см. на странице *http://activereload.net/2007/5/23/spend-less-time-in-thedatabase-and-more-time-outdoors*.

Протоколирование

В файле протокола отмечается, когда данные читались из кэша запросов, а не из базы. Поищите строки, начинающиеся со слова `CACHE`, а не `Model Load`.

```
Place Load (0.000420) SELECT * FROM places WHERE (places.'id' =
15749)
CACHE (0.000000) SELECT * FROM places WHERE (places.'id' = 15749)
CACHE (0.000000) SELECT * FROM places WHERE (places.'id' = 15749)
```

Кэширование запросов по умолчанию в контроллерах

Из соображений производительности механизм кэширования запросов ActiveRecord по умолчанию включается при обработке действий контроллеров. Модуль `SqlCache`, определенный в файле **caching.rb** библиотеки ActionController, примешан к классу `ActionController::Base` и обертывает метод `perform_action` с помощью `alias_method_chain`:

```
module SqlCache
  def self.included(base) #:nodoc:
    base.alias_method_chain :perform_action, :caching
  end

  def perform_action_with_caching
    ActiveRecord::Base.cache do
      perform_action_without_caching
    end
  end
end
```

Ограничения

Кэш запросов ActiveRecord намеренно сделан чрезвычайно простым. Поскольку в качестве ключей хеша буквально используются предложения SQL, с помощью которых были выбраны данные, то невозможно опознать различные вызовы `find`, отличающиеся формулировкой, но семантически эквивалентные и дающие одинаковые результаты.

Например, предложения select foo from bar where id = 1 и select foo from bar where id = 1 limit 1 считаются разными запросами и занимают две записи в хеше. Подключаемый модуль **active_record_context** Рика Олсона – пример более интеллектуальной реализации кэша, поскольку результаты индексируются первичными ключами, а не текстами предложений SQL.

Обновление

Простейший способ манипулирования значениями атрибутов заключается в том, чтобы трактовать объект ActiveRecord как обычный объект Ruby, то есть выполнять присваивание напрямую с помощью метода `myprop=(some_value)`.

Есть также целый ряд других способов обновления объектов ActiveRecord, о них и пойдет речь в этом разделе. Посмотрите, как используется метод `update` класса `ActiveRecord::Base`:

```
class ProjectController < ApplicationController
  def update
    Project.update(params[:id], params[:project])
    redirect_to :action=>'settings', :id => project.id
  end

  def mass_update
    Project.update(params[:projects].keys, params[:projects].values])
    redirect_to :action=>'index'
  end
end
```

Первый вариант `update` принимает числовой идентификатор и хеш значений атрибутов, а второй – список идентификаторов и список значений. Второй вариант полезен при обработке формы, содержащей несколько допускающих обновление строк.

Метод класса `update` сначала вызывает процедуру проверки и не сохраняет запись, если проверка не проходит. Однако объект он возвращает вне зависимости от того, проверены данные успешно или нет. Следовательно, если вы хотите узнать результат проверки, то должны после обращения к `update` вызвать метод `valid?`:

```
class ProjectController < ApplicationController
  def update
    @project = Project.update(params[:id], params[:project])
```

```
    if @project.valid? # а надо ли выполнять контроль еще раз?
      redirect_to :action=>'settings', :id => project.id
    else
      render :action => 'edit'
    end
  end
end
```

Проблема в том, что в этом случае метод `valid?` вызывается дважды, поскольку один раз его уже вызывал метод `update`. Быть может, более правильно было бы воспользоваться методом экземпляра `update_attributes`:

```
class ProjectController < ApplicationController
  def update
    @project = Project.find(params[:id]
    if @project.update_attributes(params[:project])
      redirect_to :action=>'settings', :id => project.id
    else
      render :action => 'edit'
    end
  end
end
```

И, конечно, если вы хоть немного программировали для Rails, то сразу распознаете здесь идиому, поскольку она применяется в генерируемом коде обстраивания (scaffolding). Метод `update_attributes` принимает хеш со значениями атрибутов и возвращает true или false в зависимости от того, завершилась ли операция сохранения успешно или нет, что, в свою очередь, определяется успешностью проверки.

Обновление с условием

В ActiveRecord есть еще один метод, полезный для обновления сразу нескольких записей: `update_all`. Он тесно связан с предложением SQL `update…where`. Метод `update_all` принимает два параметра: часть `set` предложения SQL и условия, включаемые в часть `where`. Возвращается количество обновленных записей[1].

Мне кажется, что это один из методов, которые более уместны в контексте сценария, а не в методе контроллера, но у вас может быть иное мнение. Вот пример, показывающий, как я передал бы ответственность за все проекты Rails в системе новому менеджеру проектов:

```
Project.update_all("manager = 'Ron Campbell'", "technology =
'Rails'")
```

[1] Библиотека Microsoft ADO не сообщает, сколько было обновлено записей, поэтому метод `update_all` не работает с адаптером для SQL Server.

Обновление конкретного экземпляра

Самый простой способ обновить объект ActiveRecord состоит в том, чтобы изменить его атрибуты напрямую, а потом вызвать метод `save`. Стоит отметить, что метод `save` либо вставляет запись в базу данных, *либо* – если запись с таким первичным ключом уже есть – обновляет ее:

```
project = Project.find(1)
project.manager = 'Brett M.'
assert_equal true, project.save
```

Метод `save` возвращает true, если сохранение завершилось успешно, и false в противном случае. Существует также метод `save!`, который в случае ошибки возбуждает исключение. Каким из них пользоваться, зависит от того, хотите ли вы обрабатывать ошибки немедленно или поручить это какому-то другому методу выше по цепочке вызовов.

В общем-то, это вопрос стиля, хотя методы сохранения и обновления без восклицательного знака, то есть возвращающие булево значение, чаще используются в действиях контроллеров, где фигурируют в качестве условия, проверяемого в предложении `if`:

```
class StoryController < ApplicationController

  def points
    @story = Story.find(params[:id])
    if @story.update_attribute(:points, params[:value])
      render :text => "#{@story.name} updated"
    else
      render :text => "Error updating story points"
    end
  end
end
```

Обновление конкретных атрибутов

Методы экземпляра `update_attribute` и `update_attributes` принимают либо одну пару ключ/значение, либо хеш атрибутов соответственно. В указанные атрибуты записываются переданные значения, и новые данные сохраняются в базе – все в рамках одной операции.

Метод `update_attribute` обновляет единственный атрибут и сохраняет запись. Обновления, выполняемые этим методом, *не подвергаются проверке*! Другими словами, данный метод позволяет сохранить модель в базе данных, даже если состояние объекта некорректно. По заявлению разработчиков ядра Rails, это сделано намеренно. Внутри метод эквивалентен присваиванию `model.attribute = some_value`, за которым следует `model.save(false)`.

Говорит Кортенэ...

Если в модели определены ассоциации, то ActiveRecord автоматически создает вспомогательные методы для массового присваивания. Иными словами, если в модели `Project` имеется ассоциация `has_many :users`, то появится метод записи атрибутов `user_ids`, который будет вызван из `update_attributes`.

Это удобно, если вы обновляете ассоциации с помощью флажков в интерфейсе пользователя, поскольку достаточно назвать флажки `project[user_ids][]` – и все остальное Rails проделает самостоятельно.

В некоторых случаях небезопасно разрешать пользователю устанавливать ассоциации таким способом. Подумайте, не стоит ли прибегнуть к методу `attr_accessible`, чтобы предотвратить массовое присваивание, когда есть шанс, что какой-нибудь злоумышленник попробует воспользоваться вашим приложением некорректно.

Напротив, метод `update_attributes` *выполняет все проверки*, поэтому часто используется в действиях по обновлению, где в качестве параметра ему передается хеш `params`, содержащий новые значения.

Вспомогательные методы обновления

Rails предлагает ряд вспомогательных методов обновления вида `increment` (увеличить на единицу), `decrement` (уменьшить на единицу) и `toggle` (изменить состояние на противоположное), которые выполняют соответствующие действия для числовых и булевых атрибутов. У каждого из них имеется вариант с восклицательным знаком (например, `toggle!`), который после модификации атрибута вызывает еще и метод `save`.

Контроль доступа к атрибутам

Конструкторы и методы обновления, принимающие хеши для выполнения массового присваивания значений атрибутам, уязвимы для хакерских атак, если используются в сочетании с хешами параметров, доступными в методах контроллера.

Если в классе ActiveRecord есть атрибуты, которые вы хотели бы защитить от случайного или массового присваивания, воспользуйтесь следующими методами класса, позволяющими контролировать доступ к атрибутам.

Метод `attr_accessible` принимает список атрибутов, для которых разрешено массовое присваивание. Это наиболее осторожный способ защиты от массового присваивания.

Если же вы предпочитаете сначала разрешить все и вводить ограничения по мере необходимости, то к вашим услугам метод attr_protected. Атрибуты, переданные этому методу, будут защищены от массового присваивания. Попытка присвоить им значения просто игнорируется. Чтобы изменить значение такого атрибута, необходимо вызвать метод прямого присваивания, как показано в примере ниже:

```
class Customer < ActiveRecord::Base
  attr_protected :credit_rating
end

customer = Customer.new(:name => "Abe", :credit_rating => "Excellent")
customer.credit_rating # => nil

customer.attributes = { "credit_rating" => "Excellent" }
customer.credit_rating # => nil

# а теперь разрешенный способ задать ставку кредита
customer.credit_rating = "Average"
customer.credit_rating # => "Average"
```

Удаление и уничтожение

Наконец, есть два способа удалить запись из базы данных. Если уже имеется экземпляр модели, можно уничтожить его методом destroy:

```
>> bad_timesheet = Timesheet.find(1)
>> bad_timesheet.destroy

=> #<Timesheet:0x2481d70 @attributes={"updated_at"=>"2006-11-21
05:40:27", "id"=>"1", "user_id"=>"1", "submitted"=>nil, "created_at"=>
"2006-11-21 05:40:27"}>
```

Метод destroy удаляет данные из базы и замораживает экземпляр (делает доступным только для чтения), чтобы нельзя было сохранить его еще раз:

```
>> bad_timesheet.save
TypeError: can't modify frozen hash
from activerecord/lib/active_record/base.rb:1965:in `[]='
```

Альтернативно можно вызывать методы destroy и delete как методы класса, передавая один или несколько идентификаторов записей, подлежащих удалению. Оба варианта принимают либо единственный идентификатор, либо массив:

```
Timesheet.delete(1)
Timesheet.destroy([2, 3])
```

Схема именования может показаться несогласованной, однако это не так. Метод delete непосредственно выполняет предложение SQL, не загружая экземпляры предварительно (так быстрее). Метод destroy сначала загружает экземпляр объекта ActiveRecord, а потом вызывает

для него метод экземпляра `destroy`. Семантическое различие трудноуловимо, но становится понятным, когда задаются обратные вызовы `before_destroy` или возникают *зависимые* ассоциации, то есть дочерние объекты, которых нужно автоматически удалить вместе с родителем.

Блокировка базы данных

Термином *блокировка* обозначается техника, позволяющая предотвратить обновление одних и тех же записей несколькими одновременно работающими пользователями. При загрузке строк таблицы в модель ActiveRecord по умолчанию вообще не применяет блокировку. Если в некотором приложении Rails в любой момент времени обновлять данные может только один пользователь, то беспокоиться о блокировках не нужно.

Если же есть шанс, что чтение и обновление данных могут одновременно выполнять несколько пользователей, то вы обязаны озаботиться *конкурентностью*. Спросите себя, какие *коллизии* или *гонки* (race conditions) могут иметь место, если два пользователя попытаются обновить модель в один и тот же момент?

К учету конкурентности в приложениях, работающих с базой данных, есть несколько подходов. В ActiveRecord встроена поддержка двух из них: *оптимистической* и *пессимистической* блокировки. Есть и другие варианты, например блокирование таблиц целиком. У каждого подхода много сильных и слабых сторон, поэтому для максимально надежной работы приложения имеет смысл их комбинировать.

Оптимистическая блокировка

Оптимистическая стратегия блокировки заключается в обнаружении и разрешении конфликтов по мере их возникновения. Обычно ее рекомендуют применять в тех случаях, когда коллизии случаются нечасто. При оптимистической блокировке записи базы данных вообще не блокируются, так что название только сбивает с толку.

Оптимистическая блокировка – довольно распространенная стратегия, поскольку многие приложения проектируются так, что любой пользователь изменяет лишь данные, которые концептуально принадлежат только ему. Поэтому конкуренция за обновление одной и той же записи маловероятна. Идея оптимистической блокировки в том, что, коль скоро коллизии редки, то обрабатывать их нужно лишь в случае реального возникновения.

Если вы контролируете схему базы данных, то реализовать оптимистическую блокировку совсем просто. Достаточно добавить в таблицу целочисленную колонку с именем **lock_version** и значением по умолчанию 0:

```
class AddLockVersionToTimesheets < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :lock_version, :integer, :default => 0
  end

  def self.down
    remove_column :timesheets, :lock_version
  end
end
```

Само наличие такой колонки изменяет поведение ActiveRecord. Если некоторая запись загружена в два экземпляра модели и сохранена с разными значениями атрибутов, то успешно завершится обновление первого экземпляра, а при обновлении второго будет возбуждено исключение `ActiveRecord::StaleObjectError`.

Для иллюстрации оптимистической блокировки напишем простой автономный тест:

```
class TimesheetTest < Test::Unit::TestCase

  fixtures :timesheets, :users

  def test_optimistic_locking_behavior
    first_instance = Timesheet.find(1)
    second_instance = Timesheet.find(1)

    first_instance.approver = users(:approver)
    second_instance.approver = users(:approver2)

    assert first_instance.save, "Успешно сохранен первый экземпляр"

    assert_raises ActiveRecord::StaleObjectError do
      second_instance.save
    end
  end
end
```

Тест проходит, потому что при вызове `save` для второго экземпляра мы ожидаем исключения `ActiveRecord::StaleObjectError`. Отметим, что метод `save` (без восклицательного знака) возвращает false и не возбуждает исключений, если сохранение не выполнено *из-за ошибки контроля данных*. Другие проблемы, например блокировка записи, могут приводить к исключению. Если вы хотите, чтобы колонка, содержащая номер версии, называлась не **lock_version**, а как-то иначе, измените эту настройку с помощью метода `set_locking_column`. Чтобы это изменение действовало глобально, добавьте в файл **environment.rb** такую строку:

```
ActiveRecord::Base.set_locking_column 'alternate_lock_version'
```

Как и другие настройки ActiveRecord, эту можно задать на уровне модели, если включить в класс модели такое объявление:

```
class Timesheet < ActiveRecord::Base
  set_locking_column 'alternate_lock_version'
end
```

Обработка исключения StaleObjectError

Добавив оптимистическую блокировку, вы, конечно, не захотите остановиться на этом, поскольку иначе пользователь, оказавшийся проигравшей стороной в разрешении коллизии, просто увидит на экране сообщение об ошибке. Надо постараться обработать исключение `StaleObjectError` с наименьшими потерями.

Если обновляемые данные очень важны, вы, возможно, захотите потратить время на то, чтобы смастерить дружелюбную к пользователю систему, каким-то образом сохраняющую изменения, которые тот пытался внести. Если же данные легко ввести заново, то как минимум сообщите пользователю, что обновление не состоялось. Ниже приведен код контроллера, в котором реализован такой подход:

```
def update
  begin
    @timesheet = Timesheet.find(params[:id])
    @timesheet.update_attributes(params[:timesheet])
    # куда-нибудь переадресовать
  rescue ActiveRecord::StaleObjectError
    flash[:error] = "Табель был модифицирован, пока вы его редактировали."
    redirect_to :action => 'edit', :id => @timesheet
  end
end
```

У оптимистической блокировки есть ряд преимуществ. Для нее не нужны специальные механизмы СУБД, и реализуется она сравнительно просто. Как видно из примера, для обработки исключения `StaleObjectError` потребовалось очень немного кода.

Недостатки связаны, главным образом, с тем, что операции обновления занимают чуть больше времени, так как необходимо проверить версию блокировки. Кроме того, пользователи могут быть недовольны, так как узнают об ошибке только *после* отправки данных, терять которые было бы крайне нежелательно.

Пессимистическая блокировка

Для пессимистической блокировки требуется специальная поддержка со стороны СУБД (впрочем, в наиболее распространенных СУБД она есть). На время операции обновления блокируются некоторые строки в таблицах. Это не дает другим пользователям читать записи, которые будут обновлены, и тем самым предотвращает потенциальную возможность работы с устаревшими данными.

Пессимистическая блокировка появилась в Rails относительно недавно и работает в сочетании с транзакциями, как показано в примере ниже:

```
Timesheet.transaction do
  t = Timesheet.find(1, :lock=> true)
  t.approved = true
  t.save!
end
```

Можно также вызвать метод `lock!` для существующего экземпляра модели, а он уже внутри вызовет `reload(:lock => true)`. Вряд ли стоит это делать после изменения атрибутов экземпляра, поскольку при перезагрузке изменения будут потеряны.

Пессимистическая блокировка производится на уровне базы данных. В сгенерированное предложение `SELECT` ActiveRecord добавит модификатор `FOR UPDATE` (или его аналог), в результате чего всем остальным соединениям будет заблокирован доступ к строкам, возвращенным этим предложением. Блокировка снимается после фиксации транзакции. Теоретически возможны ситуации (скажем, Rails «грохается» в середине транзакции?!), когда блокировка не будет снята до тех пор, пока соединение не завершится или не будет закрыто по тайм-ауту.

Замечание

Веб-приложения лучше масштабируются при оптимистической блокировке, поскольку, как мы уже говорили, в этом случае вообще никакие записи не блокируются. Однако приходится добавлять логику обработки ошибок. Пессимистическую блокировку реализовать несколько проще, но при этом могут возникать ситуации, когда один процесс Rails вынужден ждать, пока остальные освободят блокировки, то есть *не сможет обслуживать никакие поступающие запросы*. Напомним, что процессы Rails однопоточные.

На мой взгляд, пессимистическая блокировка не должна быть такой опасной, как на некоторых других платформах, поскольку Rails не держит транзакции дольше, чем на протяжении обработки одного HTTP-запроса. Собственно, я почти уверен, что в архитектуре, где ничего не разделяется, такое вообще невозможно.

Опасаться нужно ситуации, когда многие пользователи конкурируют за доступ к одной записи, для обновления которой нужно ощутимое время. Лучше всего, чтобы транзакции с пессимистической блокировкой были короткими и исполнялись быстро.

Дополнительные средства поиска

При первом знакомстве с методом `find` мы рассматривали только поиск по первичному ключу и параметры `:first` и `:all`. Но этим доступные возможности отнюдь не исчерпываются.

Условия

Очень часто возникает необходимость отфильтровать результирующий набор, возвращенный операцией поиска (которая сводится к предложению SQL SELECT), добавив условия (в часть WHERE). ActiveRecord позволяет сделать это различными способами с помощью хеша параметров, который может быть передан методу find.

Условия задаются в параметре :conditions в виде строки, массива или хеша, представляющего часть WHERE предложения SQL. Массив следует использовать, когда исходные данные поступают из внешнего мира, например из веб-формы, и перед записью в базу должны быть *обезврежены*. Небезопасные данные, поступающие извне, называются *подозрительными* (tainted).

Условия можно задавать в виде простой строки, когда данные не вызывают подозрений. Наконец, хеш работает примерно так же, как массив, с тем отличием, что допустимо только сравнение на равенство. Если это вас устраивает (то есть условие не содержит, например, оператора LIKE), то я рекомендую пользоваться хешем, так как это безопасный и, пожалуй, наиболее удобный для восприятия способ.

В документации по Rails API есть много примеров, иллюстрирующих параметр :conditions:

```
class User < ActiveRecord::Base
  def self.authenticate_unsafely(login, password)
    find(:first,
         :conditions => "login='#{login}' AND password='#{password}'")
  end

  def self.authenticate_safely(login, password)
    find(:first,
         :conditions => ["login= ? AND password= ?", login, password])
    end

  def self.authenticate_safely_simply(login, password)
    find(:first,
         :conditions => {:login => login, :password => password})
  end
end
```

Метод authenticate_unsafely вставляет параметры прямо в запрос и потому уязвим к *атакам с внедрением SQL*, если имя и пароль пользователя берутся непосредственно из HTTP-запроса. Злоумышленник может просто включить SQL-запрос в строку, которая должна была бы содержать имя или пароль.

Методы authenticate_safely и authenticate_safely_simply *обезвреживают* имя и пароль перед вставкой в текст запроса, гарантируя тем самым, что противник не сможет экранировать запрос и войти в систему в обход аутентификации (или сделать еще что-нибудь похуже).

При использовании нескольких полей в условии бывает трудно понять, к чему относится, скажем, четвертый или пятый вопросительный знак. В таких случаях можно прибегнуть к именованным связанным переменным. Для этого нужно заменить вопросительные знаки символами, а в хеше задать значения для соответствующих символам ключей.

В документации есть хороший пример на эту тему (для краткости мы его немного изменили):

```
Company.find(:first, [
    " name = :name AND division = :div AND created_at > :date",
    {:name => "37signals", :div => "First", :date => '2005-01-01' }
])
```

Во время краткого обсуждения последней формы представления в IRC-чате Робби Рассел (Robby Russell) предложил мне такой фрагмент:

```
:conditions => ['subject LIKE :foo OR body LIKE :foo', {:foo =>
'woah'}]
```

Иными словами, при использовании именованных связанных переменных (вместо вопросительных знаков) к одной и той же переменной можно привязываться несколько раз. Здорово!

Простые условия в виде хеша также встречаются часто и бывают полезны:

```
:conditions => {:login => login, :password => password})
```

В результате генерируются условия, содержащие только сравнение на равенство, и оператор SQL `AND`. Если вам нужно что-то, кроме `AND`, придется воспользоваться другими формами.

Булевы условия

Очень важно проявлять осторожность при задании условий, содержащих булевы значения. Различные СУБД по-разному представляют их в колонках. В некоторых имеется встроенный булев тип данных, а в других применяется тот или иной символ, часто «1» / «0» или «T» / «F» (и даже «Y» / «N»).

Rails незаметно для вас производит необходимые преобразования, если условия заданы в виде массива или хеша, а в качестве значения параметра задана булева величина в смысле Ruby:

```
Timesheet.find(:all, :conditions => ['submitted=?', true])
```

Упорядочение результатов поиска

Значением параметра `:order` является фрагмент SQL, определяющий сортировку по колонкам:

```
Timesheet.find(:all, :order => 'created_at desc')
```

В SQL по умолчанию предполагается сортировка по возрастанию, если спецификация asc/desc опущена.

Говорит Уилсон...

Стандарт SQL не определяет никакой сортировки, если в запросе отсутствует часть order by. Некоторых разработчиков это застает врасплох, поскольку они считают, что по умолчанию предполагается сортировка ORDER BY id ASC.

Сортировка в случайном порядке

Rails не проверяет значение параметра `:order`, следовательно, вы можете передать любую строку, которую понимает СУБД, а не только пары колонка/порядок сортировки. Например, это может быть полезно, когда нужно выбрать случайную запись:

```
# MySQL
Timesheet.find(:first, :order => 'RAND()')

# Postgres
Timesheet.find(:first, :order => 'RANDOM()')

# Microsoft SQL Server
Timesheet.find(:first, :order => 'NEWID()')

# Oracle
Timesheet.find(:first, :order => 'dbms_random.value')
```

Отметим, что сортировка больших наборов данных в случайном порядке может работать очень медленно, особенно в случае MySQL.

Параметры limit и offset

Значением параметра `:limit` должно быть целое число, указывающее максимальное число отбираемых запросом строк. Параметр `:offset` указывает номер первой из возвращаемых строк результирующего набора, при этом самая первая строка имеет номер 1. В сочетании эти два параметра используются для разбиения результирующего набора на страницы.

Например, следующее обращение к методу `find` вернет вторую страницу списка табелей, содержащую 10 записей:

```
Timesheet.find(:all, :limit => 10, :offset => 11)
```

В зависимости от особенностей модели данных в приложении может иметь смысл всегда налагать *некоторое* ограничение на максимальное количество объектов ActiveRecord, отбираемых одним запросом. Если позволить пользователю выполнять запросы, загружающие в Rails тысячи объектов ActiveRecord, то крах неминуем.

Параметр select

По умолчанию параметр :select принимает значение «*», то есть соответствует предложению SELECT * FROM. Но это можно изменить, если, например, вы хотите выполнить соединение, но не включать в результат колонки, по которым соединение производилось. Или включить в результирующий набор вычисляемые колонки:

```
>> b = BillableWeek.find(:first, :select => "monday_hours +
   tuesday_hours + wednesday_hours as three_day_total")
=> #<BillableWeek:0x2345fd8 @attributes={"three_day_total"=>"24"}>
```

Применяя параметр :select, как показано в предыдущем примере, имейте в виду, что колонки, не заданные в запросе, – неважно, явно или с помощью «*», – *не попадают в результирующие объекты*! Так, если в том же примере обратиться к атрибуту monday_hours объекта b, результат будет неожиданным:

```
>> b.monday_hours
NoMethodError: undefined method 'monday_hours' for
 #<BillableWeek:0x2336f74 @attributes={"three_day_total"=>"24"}>
   from activerecord/lib/active_record/base.rb:1850:in
'method_missing'
   from (irb):38
```

Чтобы получить сами колонки, а *также* вычисляемую колонку, добавьте «*» в параметр :select:

```
:select => '*, monday_hours + tuesday_hours + wednesday_hours as
three_day_total'
```

Параметр from

Параметр :from определяет часть генерируемого предложения SQL, в которой перечисляются имена таблиц. Если необходимо указать дополнительные таблицы для соединения или обратиться к представлению базы данных, можете задать значение явно.

Следующий пример взят из приложения, в котором используются признаки (теги):

```
def find_tagged_with(list)
  find(:all,
       :select => "#{table_name}.*",
       :from => "#{table_name}, tags, taggings",
       :conditions =>
["#{table_name}.#{primary_key}=taggings.taggable_id
       and taggings.taggable_type = ?
       and taggings.tag_id = tags.id and tags.name IN (?)",
       name, Tag.parse(list)])
end
```

Если вам интересно, почему вместо непосредственного задания имени таблицы интерполируется переменная table_name, то скажу, что этот код подмешан в конечный класс из модулей Ruby. Данная тема подробно обсуждается в главе 9 «Дополнительные возможности ActiveRecord».

Группировка

Параметр :group задает имя колонки, по которой следует сгруппировать результаты, и отображается на часть GROUP BY предложения SQL. Вообще говоря, параметр :group используется в сочетании с :select, так как при вычислении в SELECT агрегатных функций SQL требует, чтобы все колонки, по которым агрегирование не производится, были перечислены в части GROUP BY.

```
>> users = Account.find(:all,
                        :select => 'name, SUM(cash) as money',
                        :group => 'name')
=> [#<User:0x26a744 @attributes={"name"=>"Joe", "money"=>"3500"}>,
    #<User:0xaf33aa @attributes={"name"=>"Jane", "money"=>"9245"}>]
```

Имейте в виду, что эти дополнительные колонки возвращаются в виде строк – ActiveRecord не пытается выполнить для них приведение типов. Для преобразования в числовые типы вы должны явно обратиться к методу to_i или to_f.

```
>> users.first.money > 1_000_000
ArgumentError: comparison of String with Fixnum failed
  from (irb):8:in '>'
```

Параметры блокировки

Задание параметра :lock => true для операции поиска *в контексте транзакции* устанавливает исключительную блокировку на отбираемые строки. Эта тема рассматривалась выше в разделе «Блокировка базы данных».

Соединение и включение ассоциаций

Параметр :joins полезен, когда вы выполняете группировку (GROUP BY) и агрегирование данных из других таблиц, но не хотите загружать сами ассоциированные объекты.

```
Buyer.find(:all,
           :select => 'buyers.id, count(carts.id) as cart_count',
           :joins => 'left join carts on carts.buyer_id=buyers.id',
           :group => 'buyers.id')
```

Однако чаще всего параметры :joins и :include применяются для *попутной выборки* (eager-fetch) дополнительных объектов в одном предложении SELECT. Эту тему мы рассмотрим в главе 7.

Параметр readonly

Если задать параметр `:readonly => true`, то все возвращенные объекты помечаются как доступные только для чтения. Изменить их атрибуты вы можете, а сохранить в базе – нет.

```
>> c = Comment.find(:first, :readonly => true)
=> #<Comment id: 1, body: "Hey beeyotch!">
>> c.body = "Keep it clean!"
=> "Keep it clean!"
>> c.save
ActiveRecord::ReadOnlyRecord: ActiveRecord::ReadOnlyRecord
from /vendor/rails/activerecord/lib/active_record/base.rb:1958
```

Соединение с несколькими базами данных в разных моделях

Обычно для создания соединения применяется метод `ActiveRecord::Base.establish_connection`, а для его получения – метод `ActiveRecord::Base.connection`. Все производные от `ActiveRecord::Base` классы будут работать по этому соединению. Но что если в каких-то моделях необходимо воспользоваться другим соединением? ActiveRecord позволяет задавать соединение на уровне класса.

Предположим, что имеется подкласс `ActiveRecord::Base` с именем `LegacyProject`, для которого данные хранятся не в той же базе, что для всего приложения Rails, а в какой-то другой. Для начала опишите свойства этой базы данных в отдельном разделе файла `database.yml`. Затем вызовите метод `LegacyProject.establish_connection`, чтобы для класса `LegacyProject` и *всех его подклассов* использовалось альтернативное соединение.

Кстати, чтобы этот пример работал, необходимо выполнить в контексте класса предложение `self.abstract_class = true`. В противном случае Rails будет считать, что в подклассах `LegacyProject` используется наследование с одной таблицей (single-table inheritance – STI), о котором речь пойдет в главе 9.

```
class LegacyProject < ActiveRecord::Base
  establish_connection :legacy_database
  self.abstract_class = true
  ...
end
```

Метод `establish_connection` принимает строку (или символ), указывающую на конфигурационный раздел в файле `database.yml`. Можно вместо этого передать параметры в хеше-литерале, хотя такое включение конфигурационных данных прямо в модель вместо `database.yml` может лишь привести к ненужной путанице.

```
class TempProject < ActiveRecord::Base
establish_connection(:adapter => 'sqlite3', :database =>
':memory:')
...
end
```

Rails хранит соединения с базами данных в пуле соединений внутри экземпляра класса `ActiveRecord::Base`. Пул соединений – это просто объект `Hash`, индексированный классами ActiveRecord. Когда во время выполнения возникает необходимость установить соединение, метод `retrieve_connection` просматривает иерархию классов, пока не найдет подходящее соединение.

Прямое использование соединений с базой данных

Есть возможность напрямую использовать соединения ActiveRecord с базой данных. Иногда это полезно при написании специализированных сценариев или для тестирования на скорую руку. Доступ к соединению дает атрибут `connection` класса ActiveRecord. Если во всех ваших моделях используется одно и то же соединение, получайте этот атрибут от класса `ActiveRecord::Base`.

Самая простая операция, которую можно выполнить при наличии соединения, – вызов метода `execute` из модуля `DatabaseStatements` (подробно рассматривается в следующем разделе). Например, в листинге 6.2 показан метод, который одно за другим выполняет предложения SQL, записанные в некотором файле.

Листинг 6.2. Поочередное выполнение SQL-команд из файла на одном соединении ActiveRecord

```
def execute_sql_file(path)
  File.read(path).split(';').each do |sql|
    begin
      ActiveRecord::Base.connection.execute(#{sql}\n") unless
sql.blank?
    rescue ActiveRecord::StatementInvalid
      $stderr.puts "предупреждение: #{$!}"
    end
  end
end
```

Модуль DatabaseStatements

Модуль `ActiveRecord::ConnectionAdapters::DatabaseStatements` примешивает к объекту соединения ряд полезных методов, позволяющих работать с базой данных напрямую, не используя модели ActiveRecord.

Я сознательно не включил в рассмотрение некоторые методы из этого модуля (например, `add_limit!` и `add_lock`), поскольку они нужны самой среде Rails для динамического построения SQL-предложений, и я не думаю, что разработчикам приложений от них много пользы.

```
begin_db_transaction()
```

Вручную начинает транзакцию в базе данных (и отключает принятый в ActiveRecord по умолчанию механизм автоматической фиксации).

```
commit_db_transaction()
```

Фиксирует транзакцию (и снова включает механизм автоматической фиксации).

```
delete(sql_statement)
```

Выполняет заданное SQL-предложение `DELETE` и возвращает количество удаленных строк.

```
execute(sql_statement)
```

Выполняет заданное SQL-предложение в контексте текущего соединения. В модуле `DatabaseStatements` этот метод является абстрактным и переопределяется в реализации адаптера к конкретной СУБД. Поэтому тип возвращаемого объекта, представляющего результирующий набор, зависит от использованного адаптера.

```
insert(sql_statement)
```

Выполняет SQL-предложение `INSERT` и возвращает значение последнего автоматически сгенерированного идентификатора для той таблицы, в которую произведена вставка.

```
reset_sequence!(table, column, sequence = nil)
```

Используется только для Oracle и Postgres; записывает с поименованную последовательность максимальное значение, найденное в колонке `column` таблицы `table`.

```
rollback_db_transaction()
```

Откатывает текущую транзакцию (и включает механизм автоматической фиксации). Вызывается автоматически, если блок транзакции возбуждает исключение, или возвращает false.

```
select_all(sql_statement)
```

Возвращает *массив* хешей, в которых ключами служат имена колонок, а значениями – прочитанные из них значения.

```
ActiveRecord::Base.connection.select_all("select name from businesses
order by rand() limit 5")
=> [{"name"=>"Hopkins Painting"}, {"name"=>"Whelan & Scherr"},
{"name"=>"American Top Security Svc"}, {"name"=>"Life Style Homes"},
{"name"=>"378 Liquor Wine & Beer"}]
```

```
select_one(sql_statement)
```

Аналогичен select_all, но возвращает только первую строку результирующего набора в виде объекта Hash, в котором ключами служат имена колонок, а значениями – прочитанные из них значения. Отметим, что этот метод автоматически не добавляет в заданное вами SQL-предложение модификатор limit, поэтому, если набор данных велик, не забудьте включить его самостоятельно.

```
>> ActiveRecord::Base.connection.select_one("select name from
businesses
order by rand() limit 1")
=> {"name"=>"New York New York Salon"}

select_value(sql_statement)
```

Работает, как select_one, но возвращает *единственное* значение: то, что находится в первой колонке первой строки результирующего набора.

```
>> ActiveRecord::Base.connection.select_value("select * from
businesses
order by rand() limit 1")
=> "Cimino's Pizza"

select_values(sql_statement)
```

Работает, как select_value, но возвращает массив значений из первой колонки каждой строки результирующего набора.

```
>> ActiveRecord::Base.connection.select_values("select * from
businesses
order by rand() limit 5")
=> ["Ottersberg Christine E Dds", "Bally Total Fitness", "Behboodikah,
Mahnaz Md", "Preferred Personnel Solutions", "Thoroughbred Carpets"]

update(sql_statement)
```

Выполняет заданное SQL-предложение UPDATE и возвращает количество измененных строк. В этом отношении не отличается от метода delete.

Другие методы объекта connection

Полный перечень методов объекта connection, возвращаемого экземпляром адаптера с конкретной СУБД, довольно длинный. В реализациях большинства адаптеров Rails определены специализированные варианты этих методов, и это разумно, так как все СУБД обрабатывают SQL-запросы немного по-разному, а различия между синтаксисом нестандартных команд, например извлечения метаданных, очень велики.

Заглянув в файл **abstract_adapter.rb**, вы найдете реализации всех методов, предлагаемые по умолчанию:

```
...
```

```
# Возвращает понятное человеку имя адаптера. Записывайте имя в разных
# регистрах; кто захочет, сможет воспользоваться методом downcase.
def adapter_name
  'Abstract'
end

# Поддерживает ли этот адаптер миграции? Зависит от СУБД, поэтому
# абстрактный адаптер всегда возвращает +false+.
def supports_migrations?
  false
end

# Поддерживает ли этот адаптер использование DISTINCT в COUNT? +true+
# для всех адаптеров, кроме sqlite.
def supports_count_distinct?
  true
end

...
```

В следующих описаниях и примерах я обращаюсь к объекту соединения для приложения **time_and_expenses** с консоли Rails, а ссылка на объект `connection` для удобства присвоена переменной `conn`.

```
active?
```

Показывает, является ли соединение активным и готовым для выполнения запросов.

```
adapter_name
```

Возвращает понятное человеку имя адаптера:

```
>> conn.adapter_name
=> "SQLite"
```

disconnect! и reconnect!

Закрывает активное соединение или закрывает и открывает вместо него новое соответственно.

```
raw_connection
```

Предоставляет доступ к настоящему соединению с базой данных. Полезен, когда нужно выполнить нестандартное предложение или воспользоваться теми средствами реализованного в Ruby драйвера базы данных, которые ActiveRecord не раскрывает (при попытке написать пример для этого метода я с легкостью «повалил» консоль Rails – исключения, возникающие при работе с `raw_connection`, практически не обрабатываются).

```
supports_count_distinct?
```

Показывает, поддерживает ли адаптер использование `DISTINCT` в агрегатной функции `COUNT`. Это так (возвращается `true`) для всех адаптеров,

кроме SQLite, а для этой СУБД приходится искать обходные пути выполнения подобных запросов.

```
supports_migrations?
```

Показывает, поддерживает ли адаптер миграции.

```
tables
```

Возвращает список всех таблиц, определенных в схеме базы данных. Включены также таблицы, которые обычно не раскрываются с помощью моделей ActiveRecord, в частности `schema_info` и `sessions`.

```
>> conn.tables
=> ["schema_info", "users", "timesheets", "expense_reports",
"billable_weeks", "clients", "billing_codes", "sessions"]
```

```
verify!(timeout)
```

Отложенная проверка соединения; метод `active?` Вызывается, только если он не вызывался в течение `timeout` секунд.

Другие конфигурационные параметры

Помимо параметров, говорящих ActiveRecord, как обрабатывать имена таблиц и первичных ключей, существует еще ряд настроек, управляющих различными функциями. Все они задаются в файле **config/environment.rb**.

Параметр `ActiveRecord::Base.colorize_logging` говорит Rails, нужно ли использовать ANSI-коды для раскрашивания сообщений, записываемых в протокол адаптером соединения ActiveRecord. Раскраска (доступная всюду, кроме Windows), заметно упрощает восприятие протоколов, но, если вы пользуетесь такими программами, как syslog, могут возникнуть проблемы. По умолчанию равен `true`. Измените на `false`, если просматриваете протоколы в программах, не понимающих ANSI-коды цветов.

Вот фрагмент протокола с ANSI-кодами:

```
^[[4;36;1mSQL (0.000000)^[[0m ^[[0;1mMysql::Error: Unknown table
'expense_reports': DROP TABLE expense_reports^[[0m
^[[4;35;1mSQL (0.003266)^[[0m ^[[0mCREATE TABLE expense_reports
(`id`
int(11) DEFAULT NULL auto_increment PRIMARY KEY, `user_id` int(11))
ENGINE=InnoDB^[[0m
```

`ActiveRecord::Base.default_timezone` говорит Rails, надо ли использовать `Time.local` (значение `:local`) или `Time.utc` (значение `:utc`) при интерпретации даты и времени, полученных из базы данных. По умолчанию `:local`.

Говорит Уилсон...

Почти никто из моих знакомых не знает, как просматривать раскрашенные протоколы в программах постраничного вывода. В случае программы `less` флаг `-R` включает режим вывода на экран «необработанных» управляющих символов.

`ActiveRecord::Base.allow_concurrency` определяет, надо ли создавать соединение с базой данных в каждом потоке или можно использовать одно соединение для всех потоков. По умолчанию равен `false` и, принимая во внимание количество предупреждений и страшных историй[1], сопровождающих любое упоминание этого параметра в Сети, будет разумно не трогать его. Известно, что при задании `true` количество соединений с базой резко возрастает.

`ActiveRecord::Base.generate_read_methods` определяет, нужно ли пытаться ускорить доступ за счет генерации оптимизированных методов чтения, чтобы избежать накладных обращений к методу `method_missing` при доступе к атрибутам по имени. По умолчанию равен `true`.

`ActiveRecord::Base.schema_format` задает формат вывода схемы базы данных для некоторых заданий rake. Значение `:sql` означает, что схема выводится в виде последовательности SQL-предложений, которые могут зависеть от конкретной СУБД. Помните о возможных несовместимостях при выводе в этом режиме, когда работаете с разными базами данных на этапе разработки и тестирования.

По умолчанию принимается значение `:ruby`, и тогда схема выводится в виде файла в формате `ActiveRecord::Schema`, который можно загрузить в любую базу данных, поддерживающую миграции.

Заключение

В этой главе мы рассмотрели основы работы с `ActiveRecord`, включенным в Ruby on Rails для создания классов моделей, привязанных к базе данных. Мы узнали, что философия ActiveRecord основана на *примате соглашения над конфигурацией*, который является важнейшим принципом пути Rails. Но в то же время можно вручную задать параметры, которые отменяют действующие соглашения.

---

[1] Читайте сообщение Зеда Шоу в списке рассылки о сервере Mongrel по адресу *http://permalink.gmane.org/gmane.comp.lang.ruby.mongrel.general/245*, в котором объясняются опасности, связанные с параметром `allow_concurrency`.

Мы также познакомились с методами класса `ActiveRecord::Base`, которому наследуют все сохраняемые модели в Rails. Этот класс включает все необходимое для выполнения основных CRUD-операций: создания, чтения, обновления и удаления. Наконец, мы рассмотрели, как при необходимости можно работать с объектами соединений ActiveRecord напрямую. В следующей главе мы продолжим изучение ActiveRecord и поговорим о том, как объекты во взаимосвязанных моделях могут взаимодействовать посредством ассоциаций.

7

Ассоциации в ActiveRecord

Если вы можете что-то материализовать, создать нечто, воплощающее концепцию, то затем это позволит вам работать с абстрактной идеей более эффективно. Именно так обстоит дело с конструкцией has_many :through

Джош Сассер

Ассоциации в ActiveRecord позволяют декларативно выражать отношения между классами моделей. Мощь и понятный стиль Associations API немало способствует тому, что с Rails так приятно работать.

В этой главе мы рассмотрим различные виды ассоциаций `ActiveRecord`, сделав особый упор на то, когда какие ассоциации применять и как можно их настраивать. Мы также поговорим о классах, дающих доступ к самим отношениям.

Иерархия ассоциаций

Как правило, ассоциации выглядят как методы объектов моделей ActiveRecord. Например, метод `timesheets` может представлять табели, ассоциированные с данным объектом `user`.

```
>> user.timesheets
```

Однако может быть непонятно, объекты какого типа возвращают такие методы. Связано это с тем, что такие объекты маскируются под обычные объекты или массивы Ruby (в зависимости от типа конкрет-

ной ассоциации). В показанном выше фрагменте метод `timesheet` может возвращать нечто, похожее на массив объектов, представляющих проекты.

Консоль даже подтвердит эту мысль. Спросите, какой тип имеет любой ассоциированный набор, и консоль ответит, что это `Array`:

```
>> obie.timesheets.class
=> Array
```

Но она лжет, пусть даже неосознанно. Методы для ассоциации `has_many` – на самом деле экземпляры класса `HasManyAssociation`, иерархия которого показана на рис. 7.1.

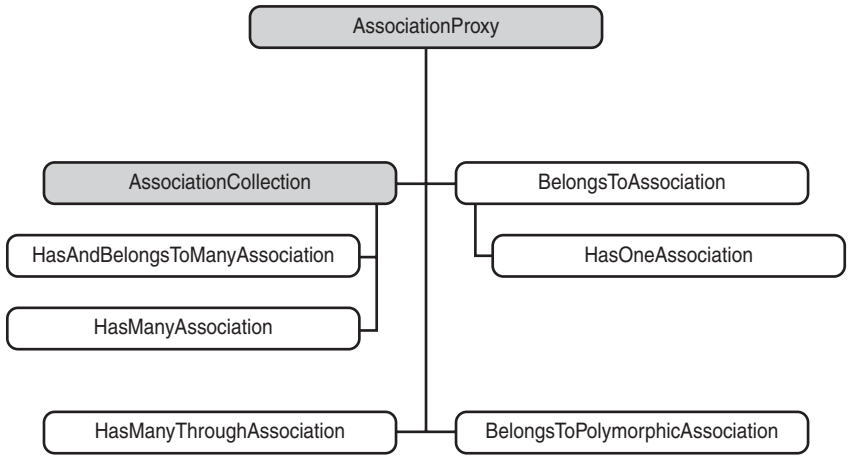

***Рис. 7.1.** Иерархия прокси-классов Association*

Предком всех ассоциаций является класс `AssociationProxy`. Он определяет базовую структуру и функциональность прокси-классов любой ассоциации. Если посмотреть в начало его исходного текста (листинг 7.1), то обнаружится, что он уничтожает определения целого ряда методов.

Листинг 7.1. Фрагмент кода из файла lib/active_record/associations/association_proxy.rb

```
instance_methods.each { |m|
undef_method m unless m =~ /(^__|^nil\?$|^send$|proxy_)/ }
```

В результате значительная часть обычных методов экземпляроа в прокси-объекте отсутствует, а их функциональность делегируется объекту, который прокси замещает, с помощью механизма `method_missing`. Это означает, что обращение к методу `timesheets.class` возвращает класс скрытого массива, а не класс самого прокси-объекта. Убедиться в том, что `timesheet` действительно является прокси-объектом, можно, спросив, отвечает ли он на какой-нибудь из открытых методов класса `AssociationProxy`, например `proxy_owner`:

```
>> obie.timesheets.respond_to? :proxy_owner
=> true
```

К счастью, в Ruby не принято интересоваться фактическим классом объекта. Гораздо важнее, на какие сообщения объект отвечает. Поэтому я думаю, что было бы ошибкой писать код, который зависит от того, с чем работает: с массивом или с прокси-объектом ассоциации. Если без этого никак не обойтись, вы всегда можете вызвать метод `to_a` и получить фактический объект `Array`:

```
>> obie.timesheets.to_a # make absolutely sure we're working with an
Array
=> []
```

Предком всех ассоциаций `has_many` является класс `AssociationCollection`, и бóльшая часть определенных в нем методов работает независимо от объявления параметров, определяющих отношение. Прежде чем переходить к деталям прокси-классов ассоциаций, познакомимся с самым фундаментальным типом ассоциации, который встречается в приложениях Rails чаще всего: парой `has_many` / `belongs_to`.

Отношения один-ко-многим

В нашем демонстрационном приложении примером отношения один-ко-многим служит ассоциация между классами `User`, `Timesheet` и `ExpenseReport`:

```
class User < ActiveRecord::Base
  has_many :timesheets
  has_many :expense_reports
end
```

Табели и отчеты о расходах необходимо связывать и в обратном направлении, чтобы можно было получить пользователя `user`, которому принадлежит отчет или табель:

```
class Timesheet < ActiveRecord::Base
  belongs_to :user
end

class ExpenseReport < ActiveRecord::Base
  belongs_to :user
end
```

При выполнении этих объявлений Rails применяет *метапрограммирование*, чтобы добавить в модели новый код. В частности, создаются объекты *прокси-наборов*, позволяющие легко манипулировать отношением.

Для демонстрации поэкспериментируем с объявленными отношениями на консоли. Для начала я создам пользователя:

```
>> obie = User.create :login => 'obie', :password => '1234',
:password_confirmation => '1234', :email => 'obiefernandez@gmail.com'
=> #<User:0x2995278 ...}>
```

Теперь проверю, появились ли наборы для табелей и отчетов о расходах:

```
>> obie.timesheets
ActiveRecord::StatementInvalid:
SQLite3::SQLException: no such column: timesheets.user_id:
SELECT * FROM timesheets WHERE (timesheets.user_id = 1)
from /.../connection_adapters/abstract_adapter.rb:128:in `log'
```

Тут Дэвид мог бы воскликнуть: «Фу-у-у!» Я забыл добавить внешние ключи в таблицы `timesheets` и `expense_reports`, поэтому, прежде чем идти дальше, сгенерирую миграцию для внесения изменений:

```
$ script/generate migration add_user_foreign_keys
exists db/migrate
create db/migrate/004_add_user_foreign_keys.rb
```

Теперь открываю файл `db/migrate/004_add_user_foreign_keys.rb` и добавляю недостающие колонки:

```
class AddUserForeignKeys < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :user_id, :integer
    add_column :expense_reports, :user_id, :integer
  end

  def self.down
    remove_column :timesheets, :user_id
    remove_column :expense_reports, :user_id
  end
end
```

Запускаю `rake db:migrate` для внесения изменений:

```
$ rake db:migrate
(in /Users/obie/prorails/time_and_expenses)
== AddUserForeignKeys: migrating
=============================================
-- add_column(:timesheets, :user_id, :integer)
   -> 0.0253s
-- add_column(:expense_reports, :user_id, :integer)
   -> 0.0101s
== AddUserForeignKeys: migrated (0.0357s)
=============================================
```

Теперь можно добавить новый пустой табель для только что созданного пользователя и убедиться, что он попал в набор `timesheets`:

```
>> obie = User.find(1)
=> #<User:0x29cc91c ... >
>> obie.timesheets << Timesheet.new
```

```
=> [#<Timesheet:0x2147524 @new_record=true, @attributes={}>]
>> obie.timesheets
=> [#<Timesheet:0x2147524 @new_record=true, @attributes={}>]
```

Добавление ассоциированных объектов в набор

Согласно документации по Rails, добавление объекта в набор has_many приводит к автоматическому его сохранению при условии, что родительский объект (владелец набора) уже сохранен в базе. Проверим, что это действительно так, для чего вызовем метод reload, который повторно считывает значения атрибутов объекта из базы данных:

```
>> obie.timesheets.reload
=> [#<Timesheet:0x29b3804 @attributes={"id"=>"1", "user_id"=>"1"}>]
```

Все на месте. Метод << автоматически установил внешний ключ user_id.

Метод << принимает один или несколько объектов ассоциаций, подлежащих добавлению в набор, и, поскольку он линеаризует (flatten) свой список аргументов и вставляет по одной записи, операции push и concat ведут себя одинаково.

В примере с пустым табелем я мог бы вызвать метод create прокси-объекта ассоциации, и он работал бы точно так же:

```
>> obie.timesheets.create
=> #<Timesheet:0x248d378 @new_record=false ... >
```

Однако будьте осторожны, выбирая между << и create! Хотя на первый взгляд оба метода делают одно и то же, существует ряд очень существенных различий в способах их реализации, и вы должны об этих различиях знать (дополнительную информацию см. в следующем подразделе «Методы класса AssociationCollection»).

Методы класса AssociationCollection

Как показано на рис. 7.1, у класса AssociationCollection имеются следующие подклассы: HasManyAssociation и HasAndBelongsToManyAssociation. Оба подкласса наследуют от своего родителя методы, описанные ниже (в классе HasManyThroughAssociation определен очень похожий набор методов, которые мы рассмотрим чуть позже).

<<(*records) и create(attributes = {})

В версии Rails 1.2.3 и предшествующих ей метод << сначала загружал *весь набор* из базы данных; потенциально это очень дорогая операция! Однако метод create просто вызывал одноименный метод класса модели ассоциации, передавая ему значение внешнего ключа, так что в базе данных устанавливалась связь. К счастью, в Rails 2.0 поведение << исправлено – он больше не загружает набор целиком и, стало быть, ведет себя очень похоже на create.

Однако эта та область Rails, в которой можно очень сильно навредить себе, если забыть об осторожности. Например, оба метода добавляют один или несколько ассоциированных объектов в зависимости от того, что именно передано: одиночный объект или массив. Но при этом метод << транзакционный, а create – нет.

Еще одно различие касается обратных вызовов ассоциаций (они рассматриваются в разделе этой главы, посвященном параметрам метода has_many). Метод << инициирует обратные вызовы :before_add и :after_add, а метод create – нет.

Наконец, оба метода существенно отличаются в плане возвращаемого значения. Метод create возвращает вновь созданный экземпляр, и это как раз то, что вы ожидаете, исходя из аналогии с одноименным методом класса ActiveRecord::Base. Метод же << возвращает прокси-объект ассоциации (хотя он и маскируется под массив), что позволяет выполнять сцепление и является естественным поведением массивов в Ruby.

Однако << вернет false, а не ссылку на себя самого, *если при добавлении хотя бы одной записи произойдет ошибка*. Поэтому вы не должны полагаться на то, что возвращаемое значение – массив, который допускает сцепление операций.

clear

Удаляет все записи из ассоциации, очищая поле внешнего ключа (см. также delete). Если при конфигурировании ассоциации параметр :dependent был задан равным :delete_all, то метод clear обходит все ассоциированные объекты и для каждого вызывает destroy.

Метод clear является *транзакционным*.

delete(*records) и delete_all

Методы delete и delete_all позволяют разорвать только заданные или все ассоциации соответственно. Оба метода *транзакционные*.

Говоря о производительности, стоит отметить, что delete_all сначала загружает весь набор ассоциированных объектов в память, чтобы получить их идентификаторы. Затем он выполняет SQL-предложение UPDATE, сбрасывая внешние ключи всех ассоциированных в данный момент объектов в null и разрывая их связь с родителем. Поскольку весь набор загружается в память, неразумно применять этот метод, когда набор ассоциированных объектов очень велик.

Примечание

Имена методов delete и delete_all могут ввести в заблуждение. По умолчанию они ничего не удаляют из базы данных, а лишь разрывают ассоциации, очищая поле внешнего ключа в ассоциированной записи. Это поведение соответствует значению :nullify параметра :dependent. Если ассоциация сконфигурирована так, что :dependent равен :delete_all или :destroy, то ассоциированные записи действительно удаляются из базы.

destroy_all

Метод `destroy_all` не принимает никаких параметров, работая по принципу «все или ничего». Он начинает транзакцию и вызывает метод `destroy` для каждого объекта ассоциации, что приводит к удалению из базы соответствующих им записей с помощью отдельных SQL-предложений `DELETE`. В этом случае возможны проблемы с производительностью, если применять данный метод к большим наборам ассоциированных объектов, так как все они предварительно загружаются в память.

length

Возвращает размер набора, загружая его в память и вызывая метод `size` для массива. Если вы собираетесь воспользоваться этим методом, чтобы проверить, пуст ли набор, то лучше вызвать `length.zero?`, а не просто `empty?`. Это более эффективно.

replace(other_array)

Заменяет текущий набор набором `other_array`. Для этого удаляет объекты, которые входят в текущий набор, но не входят в `other_array`, и вставляет (с помощью `concat`) объекты, отсутствующие в текущем наборе, но присутствующие в `other_array`.

size

Если набор уже загружен, то метод `size` возвращает его размер. В противном случае для получения размера набора ассоциированных объектов без загрузки его в память выполняется запрос `SELECT COUNT(*)`.

Если в начале работы набор не загружен и есть вероятность, что он не пуст, а загружать его все равно придется, то использование метода `length` позволит сэкономить один `SELECT`.

Параметр `:uniq`, который удаляет дубликаты из ассоциированных наборов, влияет на способ вычисления размера. По существу, он принудительно загружает все объекты из базы, чтобы Rails мог удалить дубликаты программно.

sum(column, *options)

Вычисляет сумму значений с помощью SQL-запроса. Первый параметр – символ, определяющий колонку, по которой производится суммирование. Чтобы СУБД могла подсчитать сумму, вы должны задать также параметр `:group`.

```
total = person.accounts.sum(:credit_limit, :group => 'accounts.firm_id')
```

В зависимости от структурирования ассоциации, возможно, придется устранить неоднозначности в запросе, задав префикс в имени таблицы, передаваемой `:group`.

uniq

Обходит текущий набор и формирует объект `Set`, помещая в него найденные уникальные значения. Имейте в виду, что равенство объектов `ActiveRecord` определяется на основе их идентификаторов, то есть объекты считаются равными, если значения атрибутов `id` одинаковы.

> **Предостережение относительно имен ассоциаций**
>
> Не создавайте ассоциации с такими же именами, как у методов экземпляров класса `ActiveRecord::Base`. Поскольку ассоциация добавляет метод с таким же, как у нее, именем в модель, то унаследованный метод будет переопределен, что приведет к катастрофе. Так, не стоит присваивать ассоциации имя `attributes` или `connection`.

Ассоциация belongs_to

Метод класса `belongs_to` выражает отношение между одним объектом `ActiveRecord` и одним ассоциированным объектом, для которого в первом хранится внешний ключ. «Принадлежность» определяется местоположением внешнего ключа.

Присваивание объекта ассоциации `belongs_to` заносит в атрибут внешнего ключа идентификатор объекта-владельца, но автоматически не сохраняет запись в базе данных, как видно из следующего примера:

```
>> timesheet = Timesheet.create
=> #<Timesheet:0x248f18c ... @attributes={"id"=>1409, "user_id"=>nil,
"submitted"=>nil} ...>
>> timesheet.user = obie
=> #<User:0x24f96a4 ...>
>> timesheet.user.login
=> "obie"
>> timesheet.reload
=> #<Timesheet:0x248f18c @billable_weeks=nil, @new_record=false,
@user=nil...>
```

Определение отношения `belongs_to` в классе создает атрибут с тем же именем во всех экземплярах этого класса. Как уже отмечалось, на самом деле данный атрибут является прокси для соответствующего объекта `ActiveRecord` и добавляет средства удобного манипулирования отношением.

Перезагрузка ассоциации

Простое обращение к атрибуту влечет за собой выполнение запроса к базе данных (если необходимо), в результате чего возвращается эк-

земпляр ассоциированного объекта. Акцессор принимает параметр force_reload, который указывает ActiveRecord, надо ли перезагружать объект, если в настоящий момент он кэширован в результате предыдущей операции доступа.

В следующей взятой с консоли распечатке показано, как я получаю object_id объекта user, ассоциированного с табелем. Обратите внимание, что при втором вызове ассоциации через user значение object_id не изменилось. Ассоциированный объект кэширован. Однако, если передать акцессору параметр true, будет выполнена повторная загрузка, и я получу другой экземпляр.

```
>> ts = Timesheet.find :first
=> #<Timesheet:0x3454554 @attributes={"updated_at"=>"2006-11-21
05:44:09", "id"=>"3", "user_id"=>"1", "submitted"=>nil,
"created_at"=>"2006-11-21 05:44:09"}>
>> ts.user.object_id
=> 27421330
>> ts.user.object_id
=> 27421330
>> ts.user(true).object_id
=> 27396270
```

Построение и создание связанных объектов через ассоциацию

Метод belongs_to с помощью техники метапрограммирования добавляет фабричные методы для автоматического создания новых экземпляров связанного класса и присоединения их к родительскому объекту с помощью внешнего ключа.

Метод build_association не сохраняет новый объект, тогда как create_association сохраняет. Обоим методам можно передать необязательный хеш, содержащий значения атрибутов, которыми инициализируется вновь созданный объект. Оба метода – не более чем однострочные утилиты, которые, на мой взгляд, не очень полезны, так как создавать экземпляры в этом направлении обычно не имеет смысла!

Для иллюстрации я просто приведу код построения объекта User по объекту Timesheet или объекта Client по BillingCode. В реальной программе они, скорее всего, никогда не встретятся в силу бессмысленности подобной операции:

```
>> ts = Timesheet.find :first
=> #<Timesheet:0x3437260 @attributes={"updated_at"=>"2006-11-21
05:44:09", "id"=>"3", "user_id"=>"1", "submitted"=>nil, "created_at"
=>"2006-11-21 05:44:09"}>

>> ts.build_user
=> #<User:0x3435578 @attributes={"salt"=>nil, "updated_at"=>nil,
"crypted_password"=>nil, "remember_token_expires_at"=>nil,
```

```
"remember_token"=>nil, "login"=>nil, "created_at"=>nil, "email"=>nil},
@new_record=true>

>> bc = BillingCode.find :first
=> #<BillingCode:0x33b65e8 @attributes={"code"=>"TRAVEL", "client_id"
=>nil, "id"=>"1", "description"=>"Travel expenses of all sorts"}>

>> bc.create_client
=> #<Client:0x33a3074 @new_record_before_save=true,
@errors=#<ActiveRecord::Errors:0x339f3e8 @errors={},
@base=#<Client:0x33a3074 ...>>, @attributes={"name"=>nil, "code"=>nil,
"id"=>1}, @new_record=false>
```

Гораздо чаще вам придется создавать экземпляры объектов, принадлежащих владельцу, со стороны отношения `has_many`.

Параметры метода belongs_to

Методу `belongs_to` можно передать следующие параметры в хеше.

:class_name

Представим на секунду, что нам нужно установить еще одно отношение `belongs_to` между классами `Timesheet` и `User`, чтобы промоделировать лицо, утверждающее табель. Для начала вы, наверное, добавили бы колонку `approver_id` в таблицу `timesheets` и колонку `authorized_approver` в таблицу `users`:

```
class AddApproverInfo < ActiveRecord::Migration

  def self.up
    add_column :timesheets, :approver_id, :integer
    add_column :users, :authorized_approver, :boolean
  end

  def self.down
    remove_column :timesheets, :approver_id
    remove_column :users, :authorized_approver
  end
end
```

А затем – такую ассоциацию `belongs_to`:

```
class Timesheet < ActiveRecord::Base
  belongs_to :approver
  ...
```

Проблема в том, что Rails не может определить, с каким классом вы пытаетесь установить связь, на основе только этой информации, потому что вы (вполне законно) нарушили соглашение об именовании отношения по связываемому классу. Тут-то и приходит на помощь параметр `:class_name`.

```
class Timesheet < ActiveRecord::Base
  belongs_to :approver, :class_name => 'User'
  ...
```

:conditions

А как насчет добавления условий к ассоциации belongs_to? Rails позволяет добавлять к отношению условия, которые должны выполняться, чтобы отношение считалось допустимым. Для этого предназначен параметр :conditions с тем же синтаксисом, что для добавления условий при обращении к методу find.

В последней миграции я добавил в таблицу users колонку authorized_approver, и сейчас мы ею воспользуемся:

```
class Timesheet < ActiveRecord::Base
  belongs_to :approver,
             :class_name => 'User',
             :conditions => ['authorized_approver = ?', true]
  ...
end
```

Теперь присваивание объекта user полю approver допустимо, только если пользователь имеет право утверждать табели. Я добавлю тест, который одновременно документирует мои намерения и демонстрирует их в действии.

Сначала нужно позаботиться о том, чтобы в фикстуре, описывающей пользователей (users.yml) был представлен пользователь, имеющий право утверждения. Для полноты картины я добавлю еще и пользователя, не имеющего такого права. В конец файла test/fixtures/users.yml я вставил такую разметку:

```
approver:
  id: 4
  login: "manager"
  authorized_approver: true
joe:
  id: 5
  login: "joe"
  authorized_approver: false
```

Теперь обратимся к файлу test/unit/timesheet_test.rb, в который я добавил тест, проверяющий работу приложения:

```
require File.dirname(__FILE__) + '/../test_helper'

class TimesheetTest < Test::Unit::TestCase

  fixtures :users

  def test_only_authorized_user_may_be_associated_as_approver
    sheet = Timesheet.create
```

```
    sheet.approver = users(:approver)
    assert_not_nil sheet.approver, "approver assignment failed"
  end
end
```

Для начала неплохо, но я хочу быть уверен, что система не даст записать в поле `approver` объект, представляющий неуполномоченного пользователя. Поэтому добавлю еще один тест:

```
def test_non_authorized_user_cannot_be_associated_as_approver
  sheet = Timesheet.create
  sheet.approver = users(:joe)
  assert sheet.approver.nil?, "approver assignment should have failed"
end
```

Однако у меня есть некоторые подозрения по поводу корректности этого теста, и, как я и опасался, он не работает должным образом:

```
1) Failure:
test_non_authorized_user_cannot_be_associated_as_approver(TimesheetTest)
[./test/unit/timesheet_test.rb:16]:
approver assignment should have failed.
<false> is not true.
```

Проблема в том, что `ActiveRecord` (хорошо это или плохо, но, скорее, все-таки плохо) позволяет мне выполнить недопустимое присваивание. Параметр `:conditions` применяется только во время запроса, когда ассоциация считывается из базы данных. Мне еще предстоит потрудиться, чтобы получить желаемое поведение, но пока я просто смирюсь с тем, что делает Rails, и исправлю свои тесты:

```
def test_only_authorized_user_may_be_associated_as_approver
  sheet = Timesheet.create
  sheet.approver = users(:approver)
  assert sheet.save
  assert_not_nil sheet.approver(true), "approver assignment failed"
end

def test_non_authorized_user_cannot_be_associated_as_approver
  sheet = Timesheet.create
  sheet.approver = users(:joe)
  assert sheet.save
  assert sheet.approver(true).nil?, "approver assignment should fail"
end
```

Эти тесты проходят. Я сохранил объект `sheet`, поскольку одного лишь присваивания атрибуту недостаточно для сохранения записи. Затем я воспользовался параметром `force_reload`, чтобы заставить Rails перечитать `approver` из базы данных, а не просто вернуть мне тот же экземпляр, который я только что сам присвоил в качестве значения атрибута.

Запомните, что параметр `:conditions` для отношений не влияет на присваивание ассоциированных объектов. Он влияет лишь на то, как эти

объекты считываются из базы данных. Для гарантии того, что пользователь, утверждающий табель, имеет на это право, придется добавить обратный вызов `before_save` в сам класс `Timesheet`. Подробно об обратных вызовах мы будем говорить в начале главы 9 «Дополнительные возможности ActiveRecord», а пока вернемся к параметрам ассоциации `belongs_to`.

:foreign_key

Задает имя внешнего ключа, с помощью которого следует искать ассоциированный объект. Обычно Rails самостоятельно выводит эту информацию из имени ассоциации, добавляя суффикс `_id`. Но этот параметр позволяет при необходимости переопределить данное соглашение.

```
# без явного указания Rails предполагает, в какой колонке находится
# идентификатор администратора
belongs_to :administrator, :foreign_key => 'admin_user_id'
```

:counter_cache

Этот параметр заставляет Rails автоматически обновлять счетчик ассоциированных объектов, принадлежащих данному объекту. Если параметр равен `true`, предполагается, что имя колонки составлено из множественного числа имени подчиненного класса и суффикса `_count`, но можно задать имя колонки и самостоятельно:

```
:counter_cache => true
:counter_cache => 'number_of_children'
```

Если в любой момент времени значительная доля наборов ассоциаций пуста, можно оптимизировать производительность ценой небольшого увеличения базы данных за счет кэширования счетчиков. Причина состоит в том, что, когда кэшированный счетчик равен 0, Rails *даже не пытается* запрашивать ассоциированные записи у базы данных!

Примечание

В колонку, где хранится кэшированный счетчик, СУБД *должна записывать по умолчанию значение 0*! В противном случае кэширование счетчиков *вообще не будет работать*. Связано это с тем, что Rails реализует данный механизм, добавляя простой обратный вызов, который выполняет предложение `UPDATE`, увеличивающее счетчик на единицу.

Если вы проявите беспечность и не зададите для колонки счетчика значение 0 по умолчанию или неправильно укажете имя колонки, то все равно будет казаться, что механизм кэширования счетчика работает! Во всех классах с ассоциацией `has_many` имеется магический метод `collection_count`. Он возвращает правильное значение, если параметр `:counter_cache` не задан или значение в колонке для кэширования счетчика равно `null`!

:include

Принимает список имен ассоциаций второго порядка, которые должны быть загружены одновременно с загрузкой объекта. На лету конструируется предложение SELECT с необходимыми частями LEFT OUTER JOIN, так что одним запросом к базе мы получаем весь граф объектов.

При условии здравого использования :include и тщательного замера временных характеристик иногда удается весьма значительно повысить производительность приложения, главным образом за счет устранения запросов N+1. С другой стороны, сложные запросы со многими соединениями и создание больших деревьев объектов обходятся очень дорого, поэтому иногда использование :include может весьма существенно замедлить работу приложения. Как говорится, раз на раз не приходится.

> **Говорит Уилсон...**
>
> Если :include ускоряет приложение, значит оно слишком сложное и нуждается в перепроектировании.

:polymorphic => true

Параметр :polymorphic говорит, что объект связан с ассоциацией *полиморфно*. Так в Rails говорят о ситуации, когда в базе данных вместе с внешним ключом хранится также тип связанного объекта. Сделав отношение belongs_to полиморфным, вы абстрагируете ассоциацию таким образом, что ее может заполнить любая другая модель в системе.

Полиморфные ассоциации позволяют слегка поступиться ссылочной целостностью ради удобства реализации отношений родитель-потомок, допускающих повторное использование. Типичные примеры дают такие модели, как прикрепленные фотографии, комментарии, примечания и т. д.

Проиллюстрируем эту идею, написав класс Comment, который прикрепляется к аннотируемым объектам полиморфно. Мы ассоциируем его с отчетами о расходах и табелями. В листинге 7.2 приведен код миграции, содержащий информацию об измененной схеме БД и соответствующие классы. Обратите внимание на колонку :subject_type, в которой хранится имя ассоциированного класса.

Листинг 7.2. Класс Comment, в котором используется полиморфное отношение belongs_to

```
create_table :comments do |t|
  t.column :subject, :string
  t.column :body, :text
```

```
    t.column :subject_id, :integer
    t.column :subject_type, :string
    t.column :created_at, :datetime
  end

  class Comment < ActiveRecord::Base
    belongs_to :subject, :polymorphic => true
  end

  class ExpenseReport < ActiveRecord::Base
    belongs_to :user
    has_many :comments, :as => :subject
  end

  class Timesheet < ActiveRecord::Base
    belongs_to :user
    has_many :comments, :as => :subject
  end
```

Как видно из кода классов `ExpenseReport` и `Timesheet` в листинге 7.2, имеется парный синтаксис, с помощью которого вы сообщаете `ActiveRecord`, что отношение полиморфно – `:as => :subject`. Мы пока даже не обсуждали отношения `has_many`, а полиморфным отношениям посвящен отдельный раздел в главе 9. Поэтому не будем забегать вперед, а обратимся к отношениям `has_many`.

Ассоциация has_many

Ассоциация `has_many` позволяет определить отношение, при котором для одной модели существуют *много* других *принадлежащих ей* моделей. Непревзойденная понятность таких программных конструкций, как `has_many`, – одна из причин, по которым люди влюбляются в Rails.

Метод класса `has_many` часто применяется без дополнительных параметров. Если Rails может вывести тип класса, участвующего в отношении, из имени ассоциации, то никакого добавочного конфигурирования не требуется. Следующий фрагмент кода должен быть вам уже привычен:

```
class User
  has_many :timesheets
  has_many :expense_reports
```

Имена ассоциаций можно привести к единственному числу и сопоставить с именами моделей в приложении, поэтому все работает, как и ожидается.

Параметры метода has_many

Несмотря на простоту использования метода `has_many`, для знакомых с его параметрами открывается широкое поле для настройки.

:after_add

Этот обратный вызов выполняется после добавления записи в набор методом <<. Метод create его не активирует, поэтому, прежде чем полагаться на обратные вызовы ассоциаций, надо хорошенько все продумать.

Параметры обратного вызова передаются в has_many с помощью одного или нескольких символов, соответствующих именам методов, или Proc-объектов. В листинге 7.3 приведен пример для :before_add.

:after_remove

Вызывается после того, как запись была удалена из набора методом delete. Параметры обратного вызова передаются в has_many с помощью одного или нескольких символов, соответствующих именам методов, или Proc-объектов. В листинге 7.3 приведен пример для :before_add.

:as

Определяет, какую полиморфную ассоциацию belongs_to использовать для связанного класса (о полиморфных отношениях см. главу 9).

:before_add

Вызывается перед добавлением записи в набор методом << (напомним, что concat и push – псевдонимы <<). Если обратный вызов возбудит исключение, то объект не будет добавлен в набор (главным образом, потому что обратный вызов делается сразу после проверки соответствия типов и внутри << нет предложения rescue).

Параметры обратного вызова передаются в has_many с помощью одного или нескольких символов, соответствующих именам методов, или Proc-объектов. Можно задать параметр для единственного обратного вызова (в виде Symbol или Proc) или для массива таких вызовов.

Листинг 7.3. Простой пример использования обратного вызова :before_add

```
has_many :unchangable_posts,
  :class_name => "Post",
  :before_add => :raise_exception

private
  def raise_exception(object)
    raise "Вы не можете добавить сообщение"
  end
```

Разумеется, при использовании Proc-объекта код получился бы гораздо короче – всего одна строка. Параметр owner – это объект, владеющий ассоциацией, а параметр record – добавляемый объект.

```
has_many :unchangable_posts,
  :class_name => "Post",
  :before_add => Proc.new {|owner, record| raise "Вы не можете это сделать!"}
```

И еще один вариант – с помощью лямбда-выражения, которое не проверяет *арность* параметров блока:

```
has_many :unchangable_posts,
  :class_name => "Post",
  :before_add => lamda {raise "You can't add a post"}
```

:before_remove

Вызывается перед удалением записи из набора методом `delete`. Дополнительную информацию см. в описании параметра `:before_add`.

:class_name

Параметр `:class_name` – общий для всех ассоциаций. Он позволяет задать строку, представляющую имя класса ассоциации, и необходим, когда это имя невозможно вывести из имени самой ассоциации.

:conditions

Параметр `:conditions` – общий для всех ассоциаций. Он позволяет добавить дополнительные условия в генерируемый ActiveRecord SQL-запрос, который загружает в ассоциацию объекты из базы данных.

Для указания дополнительных условий в параметре `:conditions` могут быть различные причины. Вот пример, отбирающий только утвержденные комментарии:

```
has_many :comments, :conditions => ['approved = ?', true]
```

Кроме того, никто не запрещает иметь несколько ассоциаций `has_many` для представления одной и той же пары таблиц разными способами. Не забывайте только задавать еще и имя класса:

```
has_many :pending_comments, :conditions => ['approved = ?', true],
                            :class_name => 'Comment'
```

:counter_sql

Переопределяет генерируемый ActiveRecord SQL-запрос для подсчета числа записей, принадлежащих данной ассоциации. Использовать этот параметр совместно с `:finder_sql` необязательно, так как ActiveRecord автоматически генерирует SQL-запрос для получения количества записей по переданному SQL-предложению выборки.

Как и для любых других SQL-предложений в ActiveRecord, необходимо заключать всю строку в одиночные кавычки во избежание преждевременной интерполяции (вы же не хотите, чтобы эта строка была ин-

терполирована в контексте данного класса в момент объявления ассоциации, – необходимо, чтобы интерполяция произошла во время выполнения).

```
has_many :things, :finder_sql => 'select * from t where id = #{id}'
```

:delete_sql

Переопределяет генерируемый ActiveRecord SQL-запрос для разрыва ассоциации. Доступ к ассоциированной модели предоставляет метод `record`.

:dependent => :delete_all

Все ассоциированные объекты удаляются одним махом с помощью единственной SQL-команды. Примечание: хотя этот режим *гораздо* быстрее, чем задаваемый параметром `:destroy_all`, в нем не активируются обратные вызовы при удалении ассоциированных объектов. Поэтому пользоваться им необходимо с осторожностью и только для ассоциаций, зависящих исключительно от родительского объекта.

:dependent => :destroy_all

Все ассоциированные объекты уничтожаются вместе с уничтожением родительского объекта путем вызова метода `destroy` каждого объекта.

:dependent => :nullify

По умолчанию при удалении ассоциированных записей внешние ключи, соединяющие их с родительской записью, просто очищаются (в них записывается `null`). Поэтому явно задавать этот параметр совершенно необязательно, он приведен только для справки.

:exclusively_dependent

Объявлен устаревшим; эквивалентен `:dependent => :delete_all`.

:extend => ExtensionModule

Задает модуль, методы которого расширяют класс прокси-набора ассоциации. Применяется в качестве альтернативы определению дополнительных методов в блоке, передаваемом самому методу `has_many`. Обсуждается в разделе «Расширение ассоциаций».

:finder_sql

Задает полное SQL-предложение выборки данных для ассоциаций. Это удобный способ загрузки сложных ассоциаций, зависящих от нескольких таблиц. Но применять его приходится сравнительно редко.

Операции подсчета количества записей выполняются с помощью SQL-предложения, конструируемого на основе запроса, который передает-

ся в параметре :finder_sql. Если ActiveRecord не справляется с преобразованием, необходимо также явно передать запрос в параметре :counter_sql.

:foreign_key

Переопределяет принимаемое по соглашению имя внешнего ключа, который используется в предложении SQL для загрузки ассоциации.

:group

Имя атрибута, по которому следует группировать результаты. Используется в части GROUP BY SQL-запроса.

:include

Принимает массив имен ассоциаций второго порядка, которые следует загрузить одновременно с загрузкой данного набора. Как и в случае параметра :include для ассоциаций типа belongs_to, иногда это позволяет заметно повысить производительность приложения, но нуждается в тщательном тестировании.

Для иллюстрации проанализируем, как параметр :include отражается на SQL-запросе, генерируемом для навигации по отношениям. Воспользуемся следующими упрощенными версиями классов Timesheet, BillableWeek и BillingCode:

```
class Timesheet < ActiveRecord::Base
  has_many :billable_weeks
end

class BillableWeek < ActiveRecord::Base
  belongs_to :timesheet
  belongs_to :billing_code
end

class BillingCode < ActiveRecord::Base
  belongs_to :client
  has_many :billable_weeks
end
```

Сначала необходимо подготовить тестовые данные, поэтому я создаю экземпляр timesheet и добавляю в него две оплачиваемые недели. Затем каждой оплачиваемой неделе назначаю код оплаты, что приводит к появлению графа объектов (включающего четыре объекта, связанных между собой ассоциациями).

Далее выполняю метод, записанный в одну строчку collect, который возвращает массив кодов оплаты, ассоциированный с табелем:

```
>> Timesheet.find(3).billable_weeks.collect{ |w| w.billing_code.code }
=> ["TRAVEL", "DEVELOPMENT"]
```

Если не задавать для ассоциации `billable_weeks` параметр `:include`, потребуется четыре обращения к базе данных (скопированы из протокола `log/development.log` и немного «причесаны»):

```
Timesheet Load (0.000656)     SELECT * FROM timesheets
                              WHERE (timesheets.id = 3)
BillableWeek Load (0.001156)  SELECT * FROM billable_weeks
                              WHERE (billable_weeks.timesheet_id = 3)
BillingCode Load (0.000485)   SELECT * FROM billing_codes
                              WHERE (billing_codes.id = 1)
BillingCode Load (0.000439)   SELECT * FROM billing_codes
                              WHERE (billing_codes.id = 2)
```

Это пример так называемой проблемы N+1 select, от которой страдают многие системы. Для загрузки *одной* оплачиваемой недели требуется *N* предложений `SELECT`, отбирающих ассоциированные с ней записи.

А теперь добавим в ассоциацию `billable_weeks` параметр `:include`, после чего класс `Timesheet` будет выглядеть следующим образом:

```
class Timesheet < ActiveRecord::Base
  has_many :billable_weeks, :include => [:billing_code]
end
```

Как просто! Запустив тот же тестовый запрос в консоли, мы получим те же самые результаты:

```
>> Timesheet.find(3).billable_weeks.collect{ |w| w.billing_code.code }
=> ["TRAVEL", "DEVELOPMENT"]
```

Но посмотрите, как изменилось сгенерированное SQL-предложение:

```
Timesheet Load (0.002926) SELECT * FROM timesheets LIMIT 1
BillableWeek Load Including Associations (0.001168) SELECT
billable_weeks."id" AS t0_r0, billable_weeks."timesheet_id" AS t0_r1,
billable_weeks."client_id" AS t0_r2, billable_weeks."start_date" AS
t0_r3, billable_weeks."billing_code_id" AS t0_r4,
billable_weeks."monday_hours" AS t0_r5, billable_weeks."tuesday_hours"
AS t0_r6, billable_weeks."wednesday_hours" AS t0_r7,
billable_weeks."thursday_hours" AS t0_r8,
billable_weeks."friday_hours"
AS t0_r9, billable_weeks."saturday_hours" AS t0_r10,
billable_weeks."sunday_hours" AS t0_r11, billing_codes."id" AS t1_r0,
billing_codes."client_id" AS t1_r1, billing_codes."code" AS t1_r2,
billing_codes."description" AS t1_r3 FROM billable_weeks LEFT OUTER
JOIN
billing_codes ON billing_codes.id = billable_weeks.billing_code_id
WHERE
(billable_weeks.timesheet_id = 3)
```

Rails добавил часть `LEFT OUTER JOIN`, так что данные о кодах оплаты загружаются вместе с оплачиваемыми неделями. Для больших наборов данных производительность может возрасти очень заметно!

Выявить проблему N+1 select проще всего, наблюдая, что записывается в протокол при выполнении различных операций приложения (конечно, работать надо с реальными данными, иначе это упражнение будет пустой тратой времени). Операции, для которых попутная загрузка может обернуться выгодой, характеризуются наличием многочисленных однострочных предложений SELECT – по одному для каждой ассоциированной записи.

Если у вас «зудит» (быть может, «склонны к мазохизму» – более правильное выражение в данном случае), можете попробовать активировать глубокую иерархию ассоциаций, включив хеши в массив :include:

```
Post.find(:all, :include=>[:author, {:comments=>{:author=>:gravatar }}])
```

Здесь мы отбираем не только все комментарии для сообщения Post, но и их авторов и аватары. Для описания загружаемых ассоциаций символы, массивы и хеши можно употреблять в любых сочетаниях.

Честно говоря, глубокая вложенность параметров :include плохо документирована, и, пожалуй, сопряженные с этим проблемы перевешивают потенциальные достоинства. Самая серьезная неприятность заключается в том, что выборка чрезмерно большого количества данных в одном запросе может «посадить» производительность. Начинать всегда надо с простейшего работающего решения, затем выполнять замеры и анализировать, способна ли оптимизация с применением попутной загрузки улучшить быстродействие.

> **Говорит Уилсон...**
>
> Учиться применению попутной загрузки надо, ходя по тонкому льду, как мы. Это закаляет характер!

:insert_sql

Переопределяет генерируемое ActiveRecord предложение SQL для создания ассоциаций. Для доступа к ассоциированной модели применяется метод record.

:limit

Добавляет часть LIMIT к сгенерированному SQL-предложению для загрузки данной ассоциации.

:offset

Целое число, задающее номер первой из отобранных строк, которые следует вернуть.

:order

Задает порядок сортировки ассоциированных объектов с помощью части `ORDER BY` предложения `SELECT`, например: `"last_name, first_name DESC"`.

:select

По умолчанию *, как в предложении `SELECT * FROM`. Но это можно изменить, если, например, вы хотите включить дополнительные вычисляемые колонки или добавить в ассоциированный объект колонки из соединяемых таблиц.

:source и :source_type

Применяются исключительно как дополнительные параметры при использовании ассоциации `has_many :through` с полиморфной `belongs_to`; более подробно рассматриваются ниже.

:table_name

Параметр `:table_name` позволяет переопределять имена таблиц (в части `FROM`), используемых в SQL-предложениях, которые генерируются для загрузки ассоциаций.

:through

Создает набор ассоциации посредством другой ассоциации. См. ниже раздел «Конструкция has_many :through».

:uniq => true

Убирает объекты-дубликаты из набора. Полезно в сочетании с `has_many :through`.

Методы прокси-классов

Метод класса `has_many` создает прокси-набор ассоциации, обладающий всеми методами класса `AssociationCollection`, а также несколькими методами, определенными в классе `HasManyAssociation`.

build(attributes = {})

Создает новый объект в ассоциированном наборе и связывает его с владельцем, задавая внешний ключ. Не сохраняет объект в базе данных и не добавляет его в набор ассоциации. Из следующего примера ясно, что если не сохранить значение, возвращенное методом `build`, новый объект будет потерян:

```
>> obie.timesheets
=> <timesheets not loaded yet>
```

```
>> obie.timesheets.build
=> #<Timesheet:0x24c6b8c @new_record=true, @attributes={"user_id"=>1,
"submitted"=>nil}>
>> obie.timesheets
=> <timesheets not loaded yet>
```

В онлайновой документации по API подчеркивается, что метод `build` делает в точности то же самое, что конструирование нового объекта с передачей значения внешнего ключа в виде атрибута:

```
>> Timesheet.new(:user_id => 1)
=> #<Timesheet:0x24a52fc @new_record=true, @attributes={"user_id"=>1,
"submitted"=>nil}>
```

count(*args)

Вычисляет количество ассоциированных записей в базе данных с использованием SQL.

find(*args)

Отличается от обычного метода `find` тем, что область видимости ограничена ассоциированными записями и дополнительными условиями, заданными в объявлении отношения.

Припомните пример ассоциации `has_one` в начале этой главы. Он был несколько искусственным, поскольку гораздо проще было бы найти последний модифицированный табель с помощью `find:`.

Отношения многие-ко-многим

Ассоциирование хранимых объектов с помощью связующей таблицы – один из самых сложных аспектов объектно-реляционного отображения, правильно реализовать его в среде отнюдь не тривиально. В Rails есть два способа представить в модели отношения многие-ко-многим. Мы начнем со старого и простого метода `has_and_belongs_to_many`, а затем рассмотрим более современную конструкцию `has_many :through`.

Метод has_and_belongs_to_many

Метод `has_and_belongs_to_many` устанавливает связь между двумя ассоциированными моделями `ActiveRecord` с помощью промежуточной связующей таблицы. Если связующая таблица явно не указана в параметрах, то Rails строит ее имя, конкатенируя имя таблиц в соединяемых классах в алфавитном порядке с разделяющим подчерком.

Например, если бы метод `has_and_belongs_to_many` (для краткости `habtm`) применялся для организации отношения между классами `Timesheet` и `BillingCode`, то связующая таблица называлась бы `billing_codes_`

timesheets, и это отношение было бы определено в моделях. Ниже приведены классы миграции и моделей:

```
class CreateBillingCodesTimesheets < ActiveRecord::Migration
  def self.up
    create_table :billing_codes_timesheets, :id => false do |t|
      t.column :billing_code_id, :integer, :null => false
      t.column :timesheet_id, :integer, :null => false
    end
  end

  def self.down
    drop_table :billing_codes_timesheets
  end
end

class Timesheet < ActiveRecord::Base
  has_and_belongs_to_many :billing_codes
end

class BillingCode < ActiveRecord::Base
  has_and_belongs_to_many :timesheets
end
```

Отметим, что первичный ключ id здесь не нужен, поэтому методу create_table был передан параметр :id => false. Кроме того, поскольку необходимы обе колонки с внешними ключами, мы задали параметр :null => false (в реальной программе нужно было бы позаботиться о построении индексов по обоим внешним ключам).

Отношение, ссылающееся само на себя

Что можно сказать об отношениях, ссылающихся на себя (self-referential)? Связывание модели с самой собой с помощью отношения habtm реализуется без труда – достаточно явно задать некоторые параметры.

В листинге 7.4 я создал связующую таблицу и установил связи между ассоциированными объектами BillingCode. Как и раньше, показаны классы миграции и моделей.

Листинг 7.4. Связанные коды оплаты

```
class CreateRelatedBillingCodes < ActiveRecord::Migration
  def self.up
    create_table :related_billing_codes, :id => false do |t|
      t.column :first_billing_code_id, :integer, :null => false
      t.column :second_billing_code_id, :integer, :null => false
    end
  end

  def self.down
```

```
    drop_table :related_billing_codes
  end
end

class BillingCode < ActiveRecord::Base
  has_and_belongs_to_many :related,
    :join_table => 'related_billing_codes',
    :foreign_key => 'first_billing_code_id',
    :association_foreign_key => 'second_billing_code_id',
    :class_name => 'BillingCode'
end
```

Двусторонние отношения

Стоит отметить, что отношение `related` в классе `BillingCode` из листинга 7.4 не является *двусторонним*. Тот факт, что между двумя объектами существует ассоциация в одном направлении, еще не означает, что и в другом направлении тоже есть ассоциация. Но как быть, если требуется автоматически установить двустороннее отношение?

Для начала напишем тест для класса `BillingCode`, который подтвердит работоспособность нашего решения. Начнем с добавления в файл `test/fixtures/billing_codes.yml` двух записей, с которыми будем работать далее:

```
travel:
  code: TRAVEL
  client_id:
  id: 1
  description: Разнообразные транспортные расходы
development:
  code: DEVELOPMENT
  client_id:
  id: 2
  description: Кодирование и т. д.
```

Мы не хотим, чтобы после добавления двустороннего отношения нормальная работа программы была нарушена, поэтому сначала мой тестовый метод проверяет, работает ли обычное отношение `habtm`:

```
require File.dirname(__FILE__) + '/../test_helper'

class BillingCodeTest < Test::Unit::TestCase
  fixtures :billing_codes

  def test_self_referential_habtm_association
    billing_codes(:travel).related << billing_codes(:development)
    assert BillingCode.find(1).related.include?(BillingCode.find(2))
  end
end
```

Этот тест проходит. Теперь я могу модифицировать тест, чтобы доказать работоспособность двустороннего поведения, которое мы собира-

емся добавить. Новый вариант очень похож на исходный (обычно я предпочитаю включать только одно утверждение в каждый метод, но в данном случае более уместно поместить оба утверждения вместе). Второе предложение assert проверяет, имеется ли в наборе related вновь ассоциированного класса связанный с ним объект BillingCode:

```
require File.dirname(__FILE__) + '/../test_helper'

class BillingCodeTest < Test::Unit::TestCase

  fixtures :billing_codes

  def setup
    @travel = billing_codes(:travel)
    @development = billing_codes(:development)
  end

  def test_self_referential_bidirectional_habtm_association
    @travel.related << @development
    assert @travel.related.include?(@development)
    assert @development.related.include?(@travel)
  end
end
```

Разумеется, тест не проходит, поскольку мы еще не добавили новое поведение. Мне этот подход не очень нравится, поскольку приходится «зашивать» SQL-запрос в замечательный во всех остальных отношениях код Ruby. Однако путь Rails допускает использование SQL, когда это оправдано, а в данном случае – как раз такая ситуация.

> **Говорит Уилсон...**
>
> Если бы методы << и create активировали обратные вызовы ассоциаций, мы могли бы реализовать двусторонние отношение, не привлекая код на языке SQL. К сожалению, это не так. Сделать-то все равно можно... только при использовании create вторая сторона отношения будет не видна.

Специальные параметры для SQL

Чтобы получить двустороннее отношение, воспользуемся параметром :insert_sql метода has_and_belongs_to_many для переопределения предложения INSERT, которое Rails по умолчанию применяет для ассоциирования объектов друг с другом.

Есть мелкая хитрость, позволяющая не вспоминать, как выглядит синтаксис предложения INSERT. Просто скопируйте предложение, ко-

торое генерирует Rails. Его нетрудно найти в конце файла `log/test.log` после запуска автономного теста из предыдущего раздела:

```
INSERT INTO related_billing_codes (`first_billing_code_id`,
`second_billing_code_id`) VALUES (1, 2)
```

Теперь осталось лишь подправить предложение `INSERT`, чтобы оно вставляло не одну, а две строки. Есть искушение просто добавить точку с запятой и подставить второе предложение `INSERT`. Но это работать не будет, поскольку не разрешается объединять два предложения в одно, разделяя их точкой с запятой. Если мучает любопытство, попробуйте и посмотрите, что получится.

В Google я обнаружил способ вставить несколько строк в одном SQL-предложении, который работает в Postgres, MySQL и DB2[1]. Он описан в стандарте SQL-92, а не просто поддерживается разными производителями:

```
:insert_sql => 'INSERT INTO related_billing_codes
                (`first_billing_code_id`, `second_billing_code_id`)
                VALUES (#{id}, #{record.id}), (#{record.id}, #{id})'
```

Пытаясь заставить работать специальные параметры для SQL, надо помнить о нескольких важных моментах. Во-первых, вся строка, содержащая SQL-предложение, должна быть заключена в *одиночные кавычки*. Если вы воспользуетесь двойными кавычками, то интерполяция в строку произойдет в контексте класса, в котором она объявлена, а не во время выполнения запроса.

Кроме того, раз уж мы заговорили о кавычках, обратите внимание, что после копирования предложения `INSERT` из протокола имена колонок оказались заключены в обратные апострофы, а не в одиночные кавычки. Попытка использовать одиночные кавычки в этом случае обречена на провал, так как адаптер базы данных экранирует кавычки, что приводит к неверному синтаксису. Согласен, это раздражает, но, к счастью, самостоятельно задавать SQL-запросы приходится не так часто.

И еще не забывайте, что интерполяция в строку, содержащую ваше SQL-предложение, происходит в контексте объекта, владеющего ассоциацией. Ассоциированный же объект доступен с помощью метода `record`. Если вы внимательно читали листинг, то заметили, что для установки двусторонней связи мы просто добавили в таблицу `related_billing_codes` две строки – по одной для каждого направления.

Прогон теста подтверждает, что решение на основе `:insert_sql` работает. Следует также задать параметр `:delete_sql`, чтобы можно было корректно разорвать двустороннее отношение. Как и раньше, я следую методике разработки, управляемой тестами, поэтому добавлю в класс `BillingCodeTest` такой тест:

[1] *http://en.wikipedia.org/wiki/Insert_(SQL)#Multirow_inserts.*

```
def test_that_deletion_is_bidirectional_too
  billing_codes(:travel).related << billing_codes(:development)
  billing_codes(:travel).related.delete(billing_codes(:development))
  assert !BillingCode.find(1).related.include?(BillingCode.find(2))
  assert !BillingCode.find(2).related.include?(BillingCode.find(1))
end
```

Он очень похож на предыдущий, только после установки отношения сразу же разрывает его. Думаю, что первое утверждение выполнится успешно, а второе – с ошибкой:

```
$ ruby test/unit/billing_code_test.rb
Loaded suite test/unit/billing_code_test
Started
.F
Finished in 0.159424 seconds.

  1) Failure:
test_that_deletion_is_bidirectional_too(BillingCodeTest)
[test/unit/billing_code_test.rb:16]:
<false> is not true.

2 tests, 4 assertions, 1 failures, 0 errors
```

Ну что ж, все работает, как и ожидалось. Вытащим из протокола `log/test.log` SQL-предложение `DELETE`, которое предстоит подправить:

```
DELETE FROM related_billing_codes WHERE first_billing_code_id = 1 AND
second_billing_code_id IN (2)
```

Да-а... Тут придется повозиться подольше, чем с `INSERT`. Озадаченный оператором `IN`, я решил заглянуть в файл `active_record/associations/has_and_belongs_to_many_ association.rb` и посмотреть на реализацию соответствующего метода:

```
def delete_records(records)
  if sql = @reflection.options[:delete_sql]
    records.each { |record|
      @owner.connection.execute(interpolate_sql(sql, record))
    }
  else
    ids = quoted_record_ids(records)
    sql = "DELETE FROM #{@reflection.options[:join_table]}
           WHERE #{@reflection.primary_key_name} = #{@owner.quoted_id}
           AND #{@reflection.association_foreign_key} IN (#{ids})"
    @owner.connection.execute(sql)
  end
end
```

Окончательная версия класса `BillingCode` выглядит так:

```
class BillingCode < ActiveRecord::Base
  has_and_belongs_to_many :related,
    :join_table => 'related_billing_codes',
    :foreign_key => 'first_billing_code_id',
```

```
    :association_foreign_key => 'second_billing_code_id',
    :class_name => 'BillingCode',
    :insert_sql => 'INSERT INTO related_billing_codes
                    (`first_billing_code_id`,
                     `second_billing_code_id`)
                    VALUES (#{id}, #{record.id}), (#{record.id},
                            #{id})'
end
```

Эффективное связывание двух существующих объектов

До выхода версии Rails 2.0 метод << загружает *все содержимое ассоциированного набора* из базы данных в память. Если ассоциированных записей много, на эту операция может уйти немало времени!

Дополнительные колонки в связующих таблицах для ассоциации has_and_belongs_to_many

Rails не будет возражать, если вы добавите в связующую таблицу произвольные дополнительные колонки. Добавочные атрибуты будут прочитаны из базы и включены в объекты модели, к которым дает доступ ассоциация `habtm`. Однако практический опыт подсказывает, что сложности, с которыми придется столкнуться в приложении, если пойти по этому пути, делают его малопривлекательным.

И что же это за сложности? Во-первых, записи, прочитанные из связующих таблиц с дополнительными атрибутами, помечаются как доступные только для чтения, потому что сохранить изменения, внесенные в эти атрибуты, невозможно.

Во-вторых, надо учитывать, что способо, которым Rails делает дополнительные колонки связующей таблицы доступными вам, может приводить к проблемам в других частях программы. Конечно, магическое появление добавочных атрибутов в объекте – это «круто», но что произойдет, если вы попытаетесь обратиться к этим атрибутам для объекта, который *не был* сконструирован посредством выборки из базы по ассоциации `habtm`? Вот так! Готовьтесь провести немало времени в обществе отладчика.

Если не считать объявленного устаревшим метода `push_with_attributes`, прочие методы прокси-класса `habtm` работают так же, как для отношения `has_many`. Кроме того, `habtm` имеет те же параметры, что `has_many`; уникален лишь параметр `:join_table`, который позволяет задавать имя связующей таблицы.

Подводя итог, отметим, что ассоциация `habtm` – простой способ организовать отношение многие-ко-многим с помощью связующей таблицы. Если вам не требуются дополнительные данные, описывающие это отношение, все будет хорошо. Проблемы с `habtm` начинаются, когда в связующей таблице появляются еще какие-то колонки, и в этом случае лучше перейти к использованию конструкции `has_many :through`.

«Модели настоящего соединения» и habtm

Документация по Rails 1.2 предупреждает: «Настоятельно рекомендуется перейти от ассоциаций habtm с дополнительными атрибутами к модели настоящего соединения». Использование habtm, которое считалось одной из инноваций Rails, вышло из моды, когда появилась возможность создавать модели *настоящего* соединения посредством ассоциации has_many :through.

Впрочем, habtm никуда не денется по двум прагматическим причинам. Во-первых, этот механизм необходим многим унаследованным приложениям Rails. А, во-вторых, habtm дает способ соединять классы без определения первичного ключа в связующей таблице, что полезно. Однако в большинстве случаев отношения многие-ко-многим лучше моделировать с помощью конструкции has_many :through.

Конструкция has_many :through

Хорошо известный в сообществе Rails человек и мой друг Джош Сассер считается экспертом по ассоциациям в ActiveRecord, даже его блог называется *has_many :through*. Джош описал ассоциацию :through[1] (еще в те дни, когда эта функция только появилась в версии Rails 1.1) настолько лаконично и хорошо, что я не смогу придумать ничего лучшего:

> *Ассоциация has_many :through позволяет косвенно описать отношение один-ко-многим с помощью промежуточной связующей таблицы. Посредством одной такой таблицы можно описать несколько отношений, а, значит, этот механизм может быть использован вместо has_and_belongs_to_many. Самое существенное преимущество заключается в том, что связующая таблица содержит полноценные объекты модели вместе с первичными ключами и вспомогательными данными. Отпадает необходимость в методе push_with_attributes; модели соединения работают точно так же, как все остальные модели ActiveRecord.*

Модели соединения

Чтобы продемонстрировать использование ассоциации has_many :through на практике, подготовим модель Client, чтобы в ней было несколько объектов Timesheet, связанных с помощью обычной ассоциации has_many, которую назовем billable_weeks.

```
class Client < ActiveRecord::Base
  has_many :billable_weeks
  has_many :timesheets, :through => :billable_weeks
end
```

Класс BillableWeek уже встречался в нашем приложении и пригоден для использования в качестве модели соединения:

[1] *http://blog.hasmanythrough.com/articles/2006/02/28/association-goodness.*

```
class BillableWeek < ActiveRecord::Base
  belongs_to :client
  belongs_to :timesheet
end
```

Мы можем также подготовить обратное отношение – от табелей к клиентам:

```
class Timesheet < ActiveRecord::Base
  has_many :billable_weeks
  has_many :clients, :through => :billable_weeks
end
```

Обратите внимание, что ассоциация `has_many :through` всегда используется в сочетании с обычной ассоциацией `has_many`. Еще отметим, что обычная ассоциация `has_many` часто называется одинаково в обоих соединяемых классах, следовательно, параметр `:through` будет выглядеть идентично на обеих сторонах:

```
:through => :billable_weeks
```

А как насчет модели соединения: должна ли она всегда иметь две ассоциации `belongs_to`? *Нет*.

Вы можете использовать `has_many :through` для агрегирования ассоциаций `has_many` или `has_one` в модели соединения. Простите, что в качестве примера я возьму далекую от практики предметную область, я просто хочу как можно понятнее изложить то, что пытаюсь описать:

```
class Grandparent < ActiveRecord::Base
  has_many :parents
  has_many :grand_children, :through => :parents, :source => :childs
end

class Parent < ActiveRecord::Base
  belongs_to :grandparent
  has_many :childs
end
```

Чтобы избежать многословия в следующих главах, я буду называть такое использование ассоциации `has_many :through` *агрегированием*.

Говорит Кортенэ...

Мы постоянно применяем ассоциацию `has_many :through`! Она практически полностью вытеснила старый механизм `has_and_belongs_to_many`, поскольку позволяет преобразовать модели соединения в полноценные объекты.

Представьте, что вы пригласили девушку на свидание, а она заводит разговор об Отношениях (в конечном итоге, о Нашей Свадьбе). Это пример ассоциации, которая означает нечто более важное, чем индивидуальные объекты по обе стороны.

Замечания о применении и примеры

Использовать неагрегирующие ассоциации has_many :through можно почти так же, как любые другие ассоциации has_many. Ограничения касаются работы с несохраненными записями.

```
>> c = Client.create(:name => "Trotter's Tomahawks", :code => "ttom")
=> #<Client:0x2228410...>
>> c.timesheets << Timesheet.new
ActiveRecord::HasManyThroughCantAssociateNewRecords: Cannot associate
new records through 'Client#billable_weeks' on '#'. Both records must
have an id in order to create the has_many :through record associating
them.
```

Гм, какая неприятность! В отличие от обычной ассоциации has_many, ActiveRecord не позволяет добавить объект в ассоциацию has_many :through, если хотя бы на одной стороне отношения находится несохраненная запись.

Метод create сохраняет запись, перед тем как добавить ее, поэтому работает должным образом, если родительский объект не является несохраненным:

```
>> c.save
=> true

>> c.timesheets.create
=> [#<Timesheet:0x2212354 @new_record=false, @new_record_before_save=
true, @attributes={"updated_at"=>Sun Mar 18 15:37:18 UTC 2007,
"id"=>2,
"user_id"=>nil, "submitted"=>nil, "created_at"=>Sun Mar 18 15:37:18
UTC
2007}, @errors=#<ActiveRecord::Errors:0x2211940 @base=
#<Timesheet:0x2212354 ...>, @errors={}>> ]
```

Основное достоинство ассоциации has_many :through заключается в том, что ActiveRecord снимает с вас заботу об управлении экземплярами модели соединения. Вызвав метод reload для ассоциации billable_weeks, мы увидим, что объект, представляющий оплачиваемую неделю, создан автоматически:

```
>> c.billable_weeks.reload
=> [#<BillableWeek:0x139329c @attributes={"tuesday_hours"=>nil,
"start_date"=>nil, "timesheet_id"=>"2", "billing_code_id"=>nil,
"sunday_hours"=>nil, "friday_hours"=>nil, "monday_hours"=>nil,
"client_id"=>"2", "id"=>"2", "wednesday_hours"=>nil,
"saturday_hours"=>nil, "thursday_hours"=>nil}> ]
```

Созданный объект BillableWeek правильно ассоциирован как с Client, так и с Timesheet. К сожалению, ряд прочих атрибутов (например, колонки start_date и hours) не заполнен.

Одно из возможных решений – вызвать вместо этого метод `create` ассоциации `billable_weeks` и включить новый объект `Timesheet` как одно из предоставляемых свойств.

```
>> bw = c.billable_weeks.create(:start_date => Time.now,
                                :timesheet => Timesheet.new)
=> #<BillableWeek:0x250fe08 @timesheet=#<Timesheet:0x2510100
@new_record=false, ...>
```

Агрегирующие ассоциации

Когда конструкция `has_many :through` применяется для агрегирования ассоциаций с несколькими потомками, возникают более существенные ограничения – вы можете сколько угодно предъявлять запросы с помощью метода `find` и родственных ему, но не можете ни добавлять, ни создавать новые записи.

Давайте, например, добавим ассоциацию `billable_weeks` в класс `User`:

```
class User < ActiveRecord::Base
  has_many :timesheets
  has_many :billable_weeks, :through => :timesheets
  ...
```

Ассоциация `billable_weeks` агрегирует все объекты, представляющие оплачиваемые недели, которые принадлежат всем табелям данного пользователя:

```
class Timesheet < ActiveRecord::Base
  belongs_to :user
  has_many :billable_weeks, :include => [:billing_code]
  ...
```

А теперь зайдем в консоль Rails и подготовим данные, чтобы можно было воспользоваться новым набором `billable_weeks` (для `User`):

```
>> quentin = User.find :first
#<User id: 1, login: "quentin" ...>
>> quentin.timesheets
=> []

>> ts1 = quentin.timesheets.create
=> #<Timesheet id: 1 ...>

>> ts2 = quentin.timesheets.create
=> #<Timesheet id: 2 ...>

>> ts1.billable_weeks.create(:start_date => 1.week.ago)
=> #<BillableWeek id: 1, timesheet_id: 1 ...>

>> ts2.billable_weeks.create :start_date => 2.week.ago
=> #<BillableWeek id: 2, timesheet_id: 2 ...>
```

```
>> quentin.billable_weeks
=> [#<BillableWeek id: 1, timesheet_id: 1 ...>, #<BillableWeek id: 2,
timesheet_id: 2 ...>]
```

Просто ради смеха посмотрим, что получится, если мы попытаемся создать объект `BillableWeek` от имени экземпляра `User`:

```
>> quentin.billable_weeks.create(:start_date => 3.weeks.ago)
NoMethodError: undefined method `user_id=' for
#<BillableWeek:0x3f84424>
```

Вот так-то... `BillableWeek` принадлежит не пользователю, а табелю, поэтому в нем нет поля `user_id`.

Модели соединения и валидаторы

При добавлении в конец не-агрегирующей ассоциации `has_many :through` с помощью метода << ActiveRecord всегда создает новую модель соединения, даже если для двух соединяемых записей она уже существует. Чтобы избежать дублирования, можете добавить в модель соединения ограничение `validates_uniqueness_of`.

Вот как могло бы выглядеть такое ограничение для нашей модели соединения `BillableWeek`:

```
validates_uniqueness_of :client_id, :scope => :timesheet_id
```

Здесь мы говорим: «Для каждого табеля может существовать только один экземпляр конкретного клиента».

Если в модели соединения имеются дополнительные атрибуты с собственной логикой контроля, то следует помнить еще об одной детали. При добавлении записей непосредственно в ассоциацию `has_many :through` автоматически создается новая модель соединения *с пустым набором атрибутов*. Контроль дополнительных колонок, скорее всего, завершится неудачно. Если такое имеет место, то для добавления новой записи придется создать объект модели соединения и ассоциировать его с помощью его же собственного прокси-объекта ассоциации:

```
timesheet.billable_weeks.create(:start_date => 1.week.ago)
```

Параметры ассоциации has_many :through

У ассоциации `has_many :through` такие же параметры, что и у `has_many`. Напомним, что `:through` – сам по себе не более чем параметр `has_many`! Однако при наличии `:through` некоторые параметры `has_many` изменяют семантику или становятся более существенными. Прежде всего параметры `:class_name` и `:foreign_key` теперь недопустимы, так как они выводятся из целевой ассоциации модели соединения.

Ниже приведен перечень других параметров, которые в случае `has_many :through` интерпретируются иначе.

:source

Параметр :source определяет, *какую* ассоциацию использовать. Обычно задавать его необязательно, поскольку по умолчанию ActiveRecord предполагает, что имеется в виду единственное (или множественное) число имени ассоциации has_many. Если имена ассоциаций не соответствуют друг другу, требуется указать параметр :source явно.

Например, в следующей строке для заполнения timesheets используется ассоциация sheet в классе BillableWeek.

```
has_many :timesheets, :through => :billable_weeks, :source => :sheet
```

:source_type

Параметр :source_type необходим, когда вы устанавливаете ассоциацию has_many :through с полиморфной ассоциацией belongs_to в модели соединения.

Рассмотрим следующий пример с клиентами и контактами:

```
class Client < ActiveRecord::Base
  has_many :contact_cards
  has_many :contacts, :through => :contact_cards
end

class ContactCard < ActiveRecord::Base
  belongs_to :client
  belongs_to :contacts, :polymorphic => true
end
```

Самое важное здесь то, что у клиента Client есть много контактов contacts, которые могут описываться любой моделью, а в модели соединения ContactCard объявлены как полиморфные. Для примера ассоциируем физических и юридических лиц с контактными карточками:

```
class Person < ActiveRecord::Base
  has_many :contact_cards, :as => :contact
end

class Business < ActiveRecord::Base
  has_many :contact_cards, :as => :contact
end
```

Теперь подумайте, какое сальто предстоит сделать ActiveRecord, чтобы понять, к каким таблицам обращаться в поисках контактов клиента. Теоретически необходимо знать обо всех классах моделей, связанных с другой стороной полиморфной ассоциации contacts.

На самом деле, на такие сальто ActiveRecord не способна, что и хорошо с точки зрения производительности:

```
>> Client.find(:first).contacts
ArgumentError: /.../active_support/core_ext/hash/keys.rb:48:
in `assert_valid_keys': Unknown key(s): polymorphic
```

Единственный способ добиться в этом сценарии хоть какого-то результата – немного помочь ActiveRecord, подсказав, в какой таблице нужно искать, когда запрашивается набор `contacts`, и сделать это позволяет параметр `source_type`. Его значением является символ, представляющий имя конечного класса:

```
class Client < ActiveRecord::Base
  has_many :people_contacts, :through => :contact_cards,
           :source => :contacts, :source_type => :person
  has_many :business_contacts, :through => :contact_cards,
           :source => :contacts, :source_type => :business
end
```

После указания `:source_type` ассоциация работает должным образом.

```
>> Client.find(:first).people_contacts.create!
[#<Person:0x223e788 @attributes={"id"=>1}, @errors=
#<ActiveRecord::Errors:0x223dc0c @errors={}, @base=
#<Person: 0x...>>, @new_record_before_save=true, @new_record=false>]
```

Код получился несколько длиннее, и магии в нем нет, но он работает. Если вас расстроил тот факт, что нельзя ассоциировать `people_contacts` и `business_contacts` в одной ассоциации `contacts`, можете попробовать написать собственный метод-акцессор для контактов клиента:

```
class Client < ActiveRecord::Base
  def contacts
    people_contacts + business_contacts
  end
end
```

Разумеется, вы должны понимать, что вызов метода `contacts` означает по меньшей мере два запроса к базе данных, а вернет он объект `Array` без методов прокси-класса ассоциации, на которые вы, возможно, рассчитывали.

:uniq

Параметр `:uniq` говорит, что ассоциация должна включать только уникальные объекты. Это особенно полезно при работе с `has_many :through`, так как два разных объекта `BillableWeek` могут ссылаться на один и тот же объект `Timesheet`.

```
>> client.find(:first).timesheets.reload
[#<Timesheet:0x13e79dc @attributes={"id"=>"1", ...}>,
#<Timesheet:0x13e79b4 @attributes={"id"=>"1", ...}>]
```

Ничего экстраординарного в одновременном присутствии в памяти двух разных экземпляров модели для одной и той же записи нет, просто обычно это нежелательно.

```
class Client < ActiveRecord::Base
  has_many :timesheets, :through => :billable_weeks, :uniq => true
end
```

Если задан параметр :uniq, возвращается только один экземпляр для одной записи.

```
>> client.find(:first).timesheets.reload
[#<Timesheet:0x22332ac ...>]
```

Реализация метода uniq в классе AssociationCollection – поучительный пример того, как в Ruby создать набор, содержащий уникальные значения, пользуясь классом Set и методом inject. Она доказывает также, что уникальность определяется только первичным ключом записи и ничем больше.

```
def uniq(collection = self)
  seen = Set.new
  collection.inject([]) do |kept, record|
    unless seen.include?(record.id)
      kept << record
      seen << record.id
    end
    kept
  end
end
```

Отношения один-к-одному

Пожалуй, самым простым типом отношений являются отношения один к одному. В ActiveRecord такое отношение объявляется путем совместного использования методов has_one и belongs_to. Как и в случае отношения has_many, belongs_to вызывается для модели, которой соответствует таблица базы данных, содержащая внешний ключ, связывающий записи.

Ассоциация has_one

Концептуально метод has_one работает почти так же, как has_many, только в запросе на выборку ассоциированного объекта указывается часть LIMIT 1, чтобы отбиралось не более одной записи.

Для именования отношения типа has_one следует выбирать существительное в единственном числе, чтобы код читался более естественно, например: has one :last_timesheet, has one :primary_account, has one :profile_photo и так далее.

Посмотрим, как ассоциация has_one употребляется на практике, добавив пользователям еще и аватары:

```
class Avatar < ActiveRecord::Base
  belongs_to :user
end

class User < ActiveRecord::Base
```

```
  has_one :avatar
  # ... продолжение класса User ...
end
```

Здесь все достаточно просто. Запустив консоль, мы сможем увидеть некоторые новые методы, добавленные в класс User ассоциацией has_one:

```
>> u = User.find(:first)
>> u.avatar
=> nil

>> u.build_avatar(:url => '/avatars/smiling')
#<Avatar:0x2266bac @new_record=true, @attributes={"url"=>
"/avatars/smiling", "user_id"=>1}>

>> u.avatar.save
=> true
```

Как видите, для создания нового аватара и ассоциирования его с пользователем можно воспользоваться методом build_avatar. Хотя тот факт, что has_one ассоциирует аватар с пользователем, радует, в нем ничего, что не умеет делать has_many. Поэтому посмотрим, что происходит, когда мы присваиваем пользователю новый аватар:

```
>> u = User.find(:first)
>> u.avatar
=> #<Avatar:0x2266bac @attributes={"url"=>"/avatars/smiling",
"user_id"=>1}>

>> u.create_avatar(:url => '/avatars/frowning')

=> #<Avatar:0x225071c @new_record=false, @attributes={"url"=>
"/avatars/4567", "id"=>2, "user_id"=>1}, @errors=
#<ActiveRecord::Errors:0x224fc40 @base=#<Avatar:0x225071c ...>,
@errors={}>>

>> Avatar.find(:all)
=> [#<Avatar:0x22426f8 @attributes={"url"=>"/avatars/smiling",
"id"=>"1", "user_id"=>nil}>, #<Avatar:0x22426d0
@attributes={"url"=>"/avatars/frowning", "id"=>"2", "user_id"=>"1"}>]
```

Последняя строка – самая интересная, из нее следует, что первоначальный аватар больше не ассоциирован с пользователем. Правда, старый аватар не удален из базы данных, а нам бы этого хотелось. Поэтому зададим параметр :dependent => :destroy, чтобы аватары, более не ассоциированные ни с каким пользователем, уничтожались:

```
class User
  has_one :avatar, :dependent => :destroy
end
```

Немного поиграв с консолью, мы можем убедиться, что все работает, как задумано:

```
>> u = User.find(:first)
>> u.avatar
=> #<Avatar:0x22426d0 @attributes={"url"=>"/avatars/frowning",
"id"=>"2", "user_id"=>"1"}>

>> u.avatar = Avatar.create(:url => "/avatars/jumping")
=> #<Avatar:0x22512ac @new_record=false,
@attributes={"url"=>"avatars/jumping", "id"=>3, "user_id"=>1},
@errors=#<ActiveRecord::Errors:0x22508e8 @base=#<Avatar:0x22512ac
...>,
@errors={}>>

>> Avatar.find(:all)
=> [#<Avatar:0x22426f8 @attributes={"url"=>"/avatars/smiling", "id"
=>"1", "user_id"=>nil}>, #<Avatar:0x2245920 @attributes={"url"=>
"avatars/jumping","id"=>"3", "user_id"=>"1"}>]
```

Как видите, параметр `:dependent => :destroy` помог нам избавиться от сердитого аватара (frowning), но оставил улыбчивого (smiling). Rails уничтожает лишь аватар, связь которого с пользователем была разорвана только что, а недействительные данные, которые находились в базе до этого, там и остаются. Имейте это в виду, когда решите добавить параметр `:dependent => :destroy`, и не забудьте предварительно удалить плохие данные вручную.

Как я уже упоминал выше, ассоциация `has_one` часто используется, чтобы выделить одну интересную запись из уже имеющегося отношения `has_many`. Предположим, например, что необходимо получить доступ к последнему табелю пользователя:

```
class User < ActiveRecord::Base
  has_many :timesheets
  has_one :latest_timesheet, :class_name => 'Timesheet'
end
```

Мне пришлось задать параметр `:class_name`, чтобы ActiveRecord знала, какой объект ассоциировать (она не может вывести имя класса из имени ассоциации `:latest_timesheet`).

При добавлении отношения `has_one` в модель, где уже имеется отношение `has_many` с той же самой моделью, *необязательно* добавлять еще один вызов метода `belongs_to` только ради нового отношения `has_one`. На первый взгляд, это противоречит интуиции, но ведь для чтения данных из базы используется тот же самый внешний ключ, не так ли?

Что произойдет при замене существующего конечного объекта `has_one` другим? Это зависит от того, был ли новый связанный объект создан до или после заменяемого, поскольку ActiveRecord не добавляет никакой сортировки в запрос, генерируемый для отношения `has_one`.

Параметры ассоциации has_one

У ассоциации `has_one` практически те же параметры, что и у `has_many`.

:as

Позволяет организовать полиморфную ассоциацию (см. главу 9).

:class_name

Позволяет задать имя класса, используемого в этой ассоциации. Написав `has_one :latest_timesheet, :class_name => 'Timesheet'`, `:class_name => 'Timesheet'`, вы говорите, что `latest_timesheet` — последний объект `Timesheet` из всех ассоциированных с данным пользователем. Обычно этот параметр Rails выводит из имени ассоциации.

:conditions

Позволяет задать условия, которым должен отвечать объект, чтобы быть включенным в ассоциацию. Условия задаются так же, как при вызове метода `ActiveRecord#find`:

```
class User
  has_one :manager,
          :class_name => 'Person',
          :conditions => ["type = ?", "manager"]
end
```

Здесь `manager` определен как объект класса `Person`, для которого поле `type = "manager"`. Я почти всегда применяю параметр `:conditions` в сочетании с отношением `has_one`. Когда ActiveRecord загружает ассоциацию, она потенциально может найти много строк с подходящим внешним ключом. В отсутствие условий (или, быть может, задания сортировки) вы оставляете выбор конкретной записи на усмотрение базы данных.

:dependent

Параметр `:dependent` определяет, как `ActiveRecord` должна поступать с ассоциированными объектами, когда удаляется их родитель. Этот параметр может принимать несколько значений, которые работают точно так же, как в ассоциации `has_many`.

Если задать значение `:destroy`, Rails уничтожит ассоциированный объект, который не связан ни с одним родителем. Значение `:delete` указывает, что ассоциированный объект следует уничтожить, не активируя обычных обратных вызовов. Наконец, подразумеваемое по умолчанию значение `:nullify` записывает во внешний ключ `null`, разрывая тем самым связь между объектами.

:foreign_key

Задает имя внешнего ключа в таблице ассоциации.

:include

Разрешает «попутную загрузку» дополнительных объектов вместе с загрузкой ассоциированного объекта. Дополнительную информацию см. в описании параметра `:include` для ассоциаций `has_many` и `belongs_to`.

:order

Позволяет задать фрагмент предложения SQL для сортировки результатов. В отношениях `has_one` это особенно полезно, если нужно ассоциировать последнюю запись или что-то в этом роде:

```
class User
  has_one :latest_timesheet,
          :class_name => 'Timesheet',
          :order => 'created_at desc'
end
```

Несохраненные объекты и ассоциации

Разрешается манипулировать объектами и ассоциациями до сохранения их в базе данных, но имеется специальное поведение, о котором вы должны помнить, и связано оно главным образом с сохранением ассоциированных объектов. Считается ли объект несохраненным, зависит от того, что возвращает метод `new_record?`.

Ассоциации один-к-одному

Присваивание объекта ассоциации типа `has_one` автоматически сохраняет как сам этот объект, *так и* объект, который он заместил (если таковой был), чтобы обновить значения в поле внешнего ключа. Исключением из этого правила является случай, когда родительский объект еще не сохранен, поскольку в данной ситуации значение внешнего ключа еще неизвестно.

Если сохранение невозможно хотя бы для одного из обновленных объектов (поскольку его состояние недопустимо), операция присваивания возвращает false, *и присваивание не выполняется*. Это поведение разумно, но может служить источником недоразумений, если вы про него не знаете. Если кажется, что ассоциация не работает должным образом, проверьте правила контроля связанных объектов.

Если по какой-то причине необходимо выполнить присваивание объекта ассоциации `has_one` без сохранения, можно воспользоваться методом `build` ассоциации:

```
user.profile_photo.build(params[:photo])
```

Присваивание объекта ассоциации belongs_to не приводит к сохранению родительского или ассоциированного объекта.

Наборы

Добавление объекта в наборы has_many и has_and_belongs_to_many приводит к автоматическому сохранению при условии, что родительский объект (владелец набора) уже сохранен в базе данных.

Если объект, добавленный в набор (методом << или подобными ему средствами), не удалось сохранить, операция добавления возвращает false. Если вы хотите написать более явный код или добавить объект в набор без автоматического сохранения, можете воспользоваться методом набора build. Работает он так же, как create, но не вызывает save.

Элементы набора автоматически сохраняются (или обновляются) вместе с сохранением (или обновлением) его родителя.

Расширения ассоциаций

Прокси-объекты, управляющие доступом к ассоциациям, можно расширить, написав собственный код. Вы можете добавить нестандартные методы поиска и фабричные методы, которые будут использоваться для одной конкретной ассоциации.

Пусть, например, вам нужен лаконичный способ сослаться на лиц, связанных с учетной записью, по имени. Можно обернуть метод find_or_create_by_first_name_and_last_name набора people в такой аккуратный пакетик:

Листинг 7.5. Расширение ассоциации для набора people

```
class Account < ActiveRecord::Base
  has_many :people do
    def named(name)
      first_name, last_name = name.split(" ", 2)
      find_or_create_by_first_name_and_last_name(first_name,
last_name)
    end
  end
end
```

Теперь у набора people появился метод named.

```
person = Account.find(:first).people.named("David Heinemeier Hansson")
person.first_name # => "David"
person.last_name # => "Heinemeier Hansson"
```

Если один и тот же комплект расширений желательно применить к нескольким ассоциациям, то можно написать модуль расширения вместо блока с определениями методов.

Ниже реализована та же функциональность, что в листинге 7.5, только помещена она в отдельный модуль Ruby:

```
module ByNameExtension
  def named(name)
    first_name, last_name = name.split(" ", 2)
    find_or_create_by_first_name_and_last_name(first_name, last_name)
  end
end
```

Теперь этот модуль можно использовать для расширения различных ассоциаций, лишь бы они были совместимы (для этого примера контракт состоит в том, что должен существовать метод с именем `find_or_create_by_first_name_and_last_name`).

```
class Account < ActiveRecord::Base
  has_many :people, :extend => ByNameExtension
end

class Company < ActiveRecord::Base
  has_many :people, :extend => ByNameExtension
end
```

Если вы хотите использовать несколько модулей расширения, то можете передать в параметре `:extend` не один модуль, а целый массив:

```
has_many :people, :extend => [ByNameExtension, ByRecentExtension]
```

В случае конфликта имена методы, определенные в модулях, которые добавляются позднее, замещают методы, определенные в предшествующих модулях.

Класс AssociationProxy

Класс `AssociationProxy`, предок прокси-классов всех ассоциаций (если забыли, обратитесь к рис. 7.1), предоставляет ряд полезных методов, которые применимы к большинству ассоциаций и бывают необходимы при написании расширений.

Методы reload и reset

Метод `reset` восстанавливает начальное состояние прокси-объекта выгруженной ассоциации (кэшированные объекты ассоциаций очищаются). Метод `reload` сначала вызывает `reset`, а потом загружает ассоциированные объекты из базы данных.

Методы proxy_owner, proxy_reflection и proxy_target

Возвращают ссылки на внутренние атрибуты `owner`, `reflection` и `target` прокси-объект ассоциации.

Метод proxy_owner возвращает ссылку на объект, владеющий ассоциацией.

Объект proxy_reflection является экземпляром класса ActiveRecord::Reflection::AssociationReflection и содержит все конфигурационные параметры ассоциации. В их число входят как параметры, имеющие значения по умолчанию, так и параметры, которые были явно переданы в момент объявления ассоциации[1].

proxy_target — это ассоциированный массив (или сам ассоциированный объект в случае ассоциаций типа belongs_to и has_one).

На первый взгляд кажется неразумным предоставлять открытый доступ к этим атрибутам и разрешать манипулирование ими. Однако без доступа к ним писать нетривиальные расширения ассоциаций было бы гораздо труднее. Методы loaded?, loaded, target и target= объявлены открытыми по той же причине.

В следующем примере демонстрируется использование proxy_owner в методе расширения published_prior_to, который предложил Уилсон Билкович:

```
class ArticleCategory < ActiveRecord::Base

  acts_as_tree

  has_many :articles do

    def published_prior_to(date, options = {})
      if proxy_owner.top_level?
         Article.find_all_published_prior_to(date, :category =>
proxy_owner)
      else
         # здесь self - это ассоциация 'articles', поэтому унаследуем
         # ее контекст
         self.find(:all, options)
      end
    end
  end # расширение has_many :articles

  def top_level?
    # есть ли у нас родитель и является ли он корневым узлом дерева?
    self.parent && self.parent.parent.nil?
  end
end
```

[1] Дополнительную информацию о том, когда может быть полезен отражающий объект, а также рассказ об установке ассоциации типа has_many :through посредством других ассоциаций того же типа см. в статье по адресу *http://www.pivotalblabs.com/articles/2007/08/26/ten-things-ihate-about-proxy-objects-part-i*, которую обязательно должен прочесть каждый разработчик на платформе Rails.

Подключаемый к ActiveRecord модуль расширения `acts_as_tree` создает ассоциацию, ссылающуюся на себя по колонке `parent_id`. Ссылка `proxy_owner` используется, чтобы проверить, является ли родитель этой ассоциации узлом дерева «верхнего уровня».

Заключение

Способность моделировать ассоциации – это то, что превращает ActiveRecord в нечто большее, чем просто уровень доступа к базе данных. Простота и элегантность объявления ассоциаций – причина того, что ActiveRecord больше, чем обычный механизм объектно-реляционного отображения.

В этой главе были изложены основные принципы работы ассоциаций в ActiveRecord. Мы начали с рассмотрения иерархии классов ассоциаций с корнем в `AssociationProxy`. Хочется надеяться, что знакомство с внутренними механизмами работы помогло вам проникнуться их мощью и гибкостью. Ну а руководство по параметрам и методам всех типов ассоциаций должно стать хорошим подспорьем в повседневной работе.

8

Валидаторы в ActiveRecord

Компьютеры подобны ветхозаветным богам –
множество правил и никакой пощады.

Джозеф Кэмпбелл

Validations API в ActiveRecord позволяет декларативно объявлять допустимые состояния объектов модели. Методы контроля включены в разные точки жизненного цикла объекта модели и могут инспектировать объект на предмет того, установлены ли определенные атрибуты, находятся ли их значения в заданном диапазоне и удовлетворяют ли другим заданным логическим условиям.

В этой главе мы опишем имеющиеся методы контроля (валидаторы) и способы их эффективного применения, Мы изучим, как валидаторы взаимодействуют с атрибутами модели и как можно применить встроенный механизм выдачи сообщений об ошибках в пользовательском интерфейсе.

Наконец, мы обсудим важный RubyGem-пакет `Validatable`, который позволяет выйти за пределы встроенных в Rails возможностей и определить собственные критерии контроля для данного объекта модели в зависимости от того, какую роль он играет в системе в данный момент.

Нахождение ошибок

Проблемы, обнаруживаемые в ходе контроля данных, еще называются (маэстро, туш...) ошибками! Любой объект модели `ActiveRecord` содержит

набор ошибок, к которому можно получить доступ с помощью атрибута с именем (каким бы вы думали?) errors. Это экземпляр класса ActiveRecord::Errors, определенного в файле lib/active_record/validations.rb наряду с прочим кодом, относящимся к контролю.

Если объект модели не содержит ошибок, набор errors пуст. Когда вы вызываете для объекта модели метод valid?, выполняется целая последовательность шагов для нахождения ошибок. В слегка упрощенном виде она выглядит так:

1. Очистить набор error
2. Выполнить валидаторы
3. Вернуть признак, показывающий, пуст набор errors или нет

Если набор errors пуст, объект считается корректным. Вот так все просто. Если вы сами пишете валидатор, реализующий логику контроля (такие примеры имеются в этой главе), то помечаете объект как некорректный, добавляя элементы в набор errors с помощью метода add.

Позже мы рассмотрим класс Errors подробнее. Но сначала имеет смысл познакомиться собственно с методами контроля.

Простые декларативные валидаторы

Всюду, где возможно, рекомендуется задавать валидаторы модели декларативно с помощью одного или сразу нескольких методов класса, доступных всем экземплярам ActiveRecord. Если явно не оговорено противное, все методы validates принимают переменное число атрибутов плюс необязательные параметры. Существуют параметры, общие для всех валидаторов, их мы рассмотрим в конце раздела.

validates_acceptance_of

Во многих веб-приложениях есть окно, в котором пользователю предлагают согласиться с условиями обслуживания или сделать другой выбор. Обычно в окне присутствует флажок. Атрибуту, объявленному в этом валидаторе, не соответствует никакая колонка в базе данных; при вызове данного метода он автоматически создает виртуальные атрибуты для каждого заданного вами именованного атрибута. Я считаю такой тип контроля *синтаксической глазурью*, поскольку он специфичен именно для веб-приложений.

```
class Account < ActiveRecord::Base
  validates_acceptance_of :privacy_policy, :terms_of_service
end
```

Сообщение об ошибке

Если валидатор validates_acceptance_of обнаруживает ошибку, то в объекте модели сохраняется сообщение *attribute* must be accepted (*атрибут* необходимо принять).

Параметр accept

Необязательный параметр `:accept` позволяет изменить значение, иллюстрирующее согласие. По умолчанию оно равно `"1"`, что соответствует значению, генерируемому для флажков методами-помощниками Rails.

```
class Cancellation < ActiveRecord::Base
  validates_acceptance_of :account_cancellation, :accept => 'YES'
end
```

Если в предыдущем примере вы воспользуетесь текстовым полем, связав его с атрибутом `account_cancellation`, пользователь должен будет ввести *YES*, иначе валидатор сообщит об ошибке.

validates_associated

Если с данной моделью ассоциированы другие объекты модели, корректность которых должна проверяться при сохранении, можно применить метод `validates_associated`, работающий для всех типов ассоциаций. При вызове этого валидатора (по умолчанию в момент сохранения) будет вызван метод `valid?` каждого ассоциированного объекта.

```
class Invoice < ActiveRecord::Base
  has_many :line_items
  validates_associated :line_items
end
```

Стоит отметить, что безответственное использование метода `validates_associated` может привести к циклическим зависимостям и бесконечной рекурсии. Не *бесконечной*, конечно, просто программа рухнет. Если взять предыдущий пример, то не следует делать нечто подобное в классе `LineItem`:

```
class LineItem < ActiveRecord::Base
  belongs_to :invoice
  validates_associated :invoice
end
```

Этот валидатор не сообщит об ошибке, если ассоциация равна `nil`, так как в этот момент ее еще просто не существует. Если вы хотите убедиться, что ассоциация заполнена *и* корректна, то должны будете использовать `validates_associated` в сочетании с `validates_presence_of` (рассматривается ниже).

validates_confirmation_of

Метод `validates_confirmation_of` дает еще один пример синтаксической глазури для веб-приложений, где часто встречаются пары дублирующих текстовых полей, в которые пользователь должен ввести одно и то же значение, например пароль или адрес электронной почты. Этот валидатор создает виртуальный атрибут для значения в поле подтверж-

дения и сравнивает оба атрибута. Чтобы объект считался корректным, значения атрибутов должны совпадать.

Вот пример все для той же фиктивной модели `Account`:

```
class Account < ActiveRecord::Base
  validates_confirmation_of :email, :password
end
```

В пользовательский интерфейс для модели `Account` необходимо включить дополнительные текстовые поля, имена которых заканчиваются суффиксом `_confirmation`. В отправленной форме значения в этих полях должны совпадать со значениями в парных им полях без суффикса `_confirmation`.

validates_each

Метод `validates_each` отличается от своих собратьев тем, что для него не определена конкретная функция контроля. Вы просто передаете ему массив имен атрибутов и блок, в котором проверяется допустимость значения каждого из них.

Блок может пометить объект модели как некорректный, поместив информацию об ошибках в набор `errors`. *Значение, возвращаемое блоком, игнорируется.*

Ситуаций, в которых этот метод может пригодиться, не так уж много; примером может служить контроль с помощью внешних служб. Вы можете представить такой внешний валидатор в виде *фасада*, специфичного для своего приложения, и вызвать его с помощью блока `validates_each`:

```
class Invoice < ActiveRecord::Base
  validates_each :supplier_id, :purchase_order do |record, attr, value|
    record.errors.add(attr) unless PurchasingSystem.validate(attr, value)
  end
end
```

Отметим, что параметры для экземпляра модели (`record`), имя атрибута и проверяемое значение передаются в виде параметров блока:

validates_inclusion_of и validates_exclusion_of

Метод `validates_inclusion_of` и парный ему `validates_exclusion_of` очень полезны, но, если вы не благоговейно относитесь к требованиям, предъявляемым к приложению, то готов поспорить на небольшую сумму, что не понимаете, зачем они нужны.

Эти методы принимают переменное число имен атрибутов и необязательный параметр `:in`. Затем они проверяют, что значение атрибута входит (или соответственно не входит) в перечисляемый объект, переданный в `:in`.

Примеры, приведенные в документации по Rails, наверное, лучше всего иллюстрируют применение этих методов, поэтому я возьму их за основу:

```
class Person < ActiveRecord::Base
  validates_inclusion_of :gender, :in => ['m','f'],
                         :message => 'Среднего рода?'

class Account
  validates_exclusion_of :login,
                         :in => ['admin', 'root', 'superuser'],
                         :message => 'Борат говорит: "Уйди, противный!"'
end
```

Обратите внимание, что в этих примерах впервые встретился параметр `:message`, общий для всех методов контроля. Он служит для задания сообщения, которое помещается в набор `Errors` в случае ошибки. О сообщениях, подразумеваемых по умолчанию, и о том, как их эффективно настраивать, мы поговорим ниже.

validates_existence_of

Этот валидатор реализован в подключаемом модуле, но я считаю его настолько полезным в повседневной работе, что решил включить в свой перечень. Он проверяет, что внешний ключ в ассоциации `belongs_to` ссылается на существующую запись в базе данных. Можете считать это ограничением внешнего ключа, реализованным на уровне Rails. Валидатор также хорошо работает с полиморфными ассоциациями `belongs_to`.

```
class Person < ActiveRecord::Base
  belongs_to :address
  validates_existence_of :address
end
```

Саму идею и реализующий ее подключаемый модуль предложил Джош Сассер, который написал в своем блоге следующее:

> *Меня всегда раздражало отсутствие валидатора, проверяющего, ссылается ли внешний ключ на существующую запись. Есть валидатор validates_presence_of, который проверяет, что внешний ключ не равен nil. А validates_associated сообщит, если для записи, на которую ссылается этот ключ, не проходят ее собственные проверки. Но это либо слишком мало, либо слишком много, а мне нужно нечто среднее. Поэтому я решил, что пора написать собственный валидатор.*
>
> *http://blog.hasmanythrough.com/2007/7/14/validate-your-existence*

Чтобы установить этот дополнительный модуль, зайдите в каталог своего проекта и выполните следующую команду:

```
$ script/plugin install
http://svn.hasmanythrough.com/public/plugins/validates_existence/
```

Если параметр `:allow_nil => true`, то сам ключ может быть равен `nil`, и никакой контроль тогда не выполняется. Если же ключ отличен от `nil`, посылается запрос с целью удостовериться, что запись с таким внешним ключом существует в базе данных. По умолчанию в случае ошибки выдается сообщение does not exist (не существует), но, как и в других валидаторах, его можно переопределить с помощью параметра `:message`.

validates_format_of

Чтобы применять валидатор `validates_format_of`, вы должны уметь пользоваться регулярными выражениями в Ruby. Передайте этому методу один или несколько подлежащих проверке атрибутов и регулярное выражение в параметре `:with` (обязательном). В документации по Rails приведен хороший пример – проверка формата адреса электронной почты:

```
class Person < ActiveRecord::Base
  validates_format_of :email,
    :with => /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i
end
```

Кстати, этот валидатор не имеет *ничего общего* со спецификацией электронных почтовых адресов в RFC[1].

Говорит Кортенэ...

Регулярные выражения – замечательный инструмент, но иногда они бывают очень сложными, особенно если нужно проверять доменные имена или адреса электронной почты.

Чтобы разбить длинное регулярное выражение на обозримые куски, можно воспользоваться механизмом интерполяции `#{ }`:

```
validates_format_of :name, :with =>
/^((localhost)|#{DOMAIN}|#{NUMERIC_IP})#{PORT}$/
```

Это выражение понять довольно легко.

Сами же подставляемые константы сложнее, но разобраться в них проще, чем если бы все было свалено в кучу:

```
PORT = /(([:]\d+)?)/
DOMAIN = /([a-z0-9\-]+\.?)*([a-z0-9]{2,})\.[a-z]{2,}/
NUMERIC_IP = /(?>(?:1?\d?\d|2[0-4]\d|25[0-
5])\.){3}(?:1?\d?\d|2[0-4]\d|25[0-
5])(?:\/(?:[12]?\d|3[012])|-(?> (?:1?\d?\d|2[0-
4]\d|25[0-5])\.){3}(?:1?\d?\d|2[0-4]\d|25[0-5]))?/
```

[1] Если вам нужно контролировать электронные адреса, попробуйте подключаемый модуль по адресу *http://code.dunae.ca/validates_email_format_of*.

validates_length_of

Метод validates_length_of принимает различные параметры, позволяющие точно задать ограничения на длину одного из атрибутов модели.

```
class Account < ActiveRecord::Base
  validates_length_of :login, :minimum => 5
end
```

Параметры, задающие ограничения

В параметрах :minimum и :maximum нет никаких неожиданностей, только не надо использовать их вместе. Чтобы задать диапазон, воспользуйтесь параметром :within и передайте в нем диапазон Ruby, как показано в следующем примере:

```
class Account < ActiveRecord::Base
  validates_length_of :login, :within => 5..20
end
```

Чтобы задать точную длину атрибута, воспользуйтесь параметром :is:

```
class Account < ActiveRecord::Base
  validates_length_of :account_number, :is => 16
end
```

Параметры, управляющие сообщениями об ошибках

Rails позволяет задать детальное сообщение об ошибке, обнаруженной валидатором validates_length_of с помощью параметров :too_long, :too_short и :wrong_length. Включите в текст сообщения спецификатор %d, который будет замещен числом, соответствующим ограничению:

```
class Account < ActiveRecord::Base
  validates_length_of :account_number, :is => 16,
                      :wrong_length => "длина должна составлять %d знаков"
end
```

validates_numericality_of

Несколько коряво названный метод validates_numericality_of служит для проверки того, что атрибут содержит числовое значение. Параметр :integer_only позволяет дополнительно указать, что значение должно быть целым, и по умолчанию равен false.

```
class Account < ActiveRecord::Base
  validates_numericality_of :account_number, :integer_only => true
end
```

validates_presence_of

Один из наиболее употребительных валидаторов :validates_presence_of гарантирует, что обязательный атрибут задан. Для проверки значения

используется метод `blank?`, определенный в классе `Object`, который возвращает `true`, если значение равно `nil` или пустой строке `""`.

```
class Account < ActiveRecord::Base
  validates_presence_of :login, :email, :account_number
end
```

Проверка наличия ассоциированных объектов

Желая проверить наличие ассоциации, применяйте валидатор к атрибуту, содержащему внешний ключ, а не к самой ассоциации. Отметим, что контроль завершится неудачей, если оба объекта – родитель и потомок – еще не сохранены (поскольку в этом случае внешний ключ будет пуст).

validates_uniqueness_of

Метод `validates_uniqueness_of` проверяет, что значение атрибута уникально среди всех моделей одного и того же типа. При этом *не* добавляется ограничение уникальности на уровне базы данных. Вместо этого строится и выполняется запрос на поиск подходящей записи в базе. Если запись будет найдена, валидатор сообщит об ошибке.

```
class Account < ActiveRecord::Base
  validates_uniqueness_of :login
end
```

Параметр `:scope` позволяет использовать дополнительные атрибуты для проверки уникальности. С его помощью можно передать одно или несколько имен атрибутов в виде символов (несколько символов помещаются в массив).

```
class Address < ActiveRecord::Base
  validates_uniqueness_of :line_two, :scope => [:line_one, :city, :zip]
end
```

Можно также указать, должны ли строки сравниваться с учетом регистра; для этого служит параметр `:case_sensitive` (для не-текстовых атрибутов он игнорируется).

Гарантии уникальности модели соединения

При использовании моделей соединения (посредством `has_many :through`) довольно часто возникает необходимость сделать отношение уникальным. Рассмотрим пример, в котором моделируется запись студентов на посещение курсов:

```
class Student < ActiveRecord::Base
  has_many :registrations
  has_many :courses, :through => :registrations
end
```

```
class Registration < ActiveRecord::Base
  belongs_to :student
  belongs_to :course
end

class Course < ActiveRecord::Base
  has_many :registrations
  has_many :students, :through => :registrations
end
```

Как гарантировать, что студент не запишется на один и тот же курс более одного раза? Самый короткий способ – воспользовать валидатором `validates_uniqueness_of` с ограничением `:scope`. Но не забывайте, что указывать необходимо внешние ключи, а не имена самих ассоциаций:

```
class Registration < ActiveRecord::Base
  belongs_to :student
  belongs_to :course

  validates_uniqueness_of :student_id, :scope => :course_id,
            :message => "может записаться на каждый курс только один раз"
end
```

Поскольку в случае неудачи этой проверки сообщение об ошибке, принимаемое по умолчанию, бессмысленно, я задал собственное сообщение, которое после подстановки будет выглядеть так: «`Student` может записаться на каждый курс только один раз».

Исключение RecordInvalid

Если вы выполняете операции с восклицательным знаком (например, `save!`) или Rails самостоятельно пытается выполнить сохранение, и при этом валидатор обнаруживает ошибку, будьте готовы к обработке исключения `ActiveRecord::RecordInvalid`. Оно возникает в случае ошибки при контроле, а в сопроводительном сообщении описывается причина ошибки.

Вот простой пример, взятый из одного моего приложения, в котором модель `User` подвергается довольно строгим проверкам:

```
>> u = User.new
=> #<User ...>
>> u.save!
ActiveRecord::RecordInvalid: Validation failed: Name can't be blank,
Password confirmation can't be blank, Password is too short (minimum
is 5 characters), Email can't be blank, Email address format is bad
```

Общие параметры валидаторов

Перечисленные ниже параметры применимы ко всем методам контроля.

:allow_nil

Во многих случаях контроль необходим, только если значение задано, поскольку пустое значение допустимо. Параметр `:allow_nil` позволяет пропустить проверку, если значение атрибута равно `nil`. Помните, что значение сравнивается только с `nil`, на пустые строки этот параметр не распространяется.

:if

Параметр `:if` рассматривается в следующем разделе «Условная проверка».

:message

Выше мы уже говорили, что ошибки, обнаруженные в процессе контроля, запоминаются в наборе `Errors` проверяемого объекта модели. Частью любого элемента этого набора является сообщение, описывающее причину ошибки. Все валидаторы принимают параметр `:message`, позволяющий переопределить сообщение, подразумеваемое по умолчанию.

```
class Account < ActiveRecord::Base
  validates_uniqueness_of :login, :message => "is already taken"
end
```

:on

По умолчанию валидаторы запускаются при любом сохранении (в операциях создания и обновления). При необходимости можно ограничиться только одной из этих операций, передав в параметре `:on` значение `:create` или `:update`.

Например, режим `:on => :create` удобно использовать в сочетании с валидатором `validates_uniqueness_of`, поскольку проверка уникальности при больших наборах данных может занимать много времени.

```
class Account < ActiveRecord::Base
  validates_uniqueness_of :login, :on => :create
end
```

Минуточку – а не приведет ли это к проблемам, если при обновлении модели в будущем будет задано неуникальное значение? Вот тут-то на сцену выходит метод `attr_protected`. С его помощью важные атрибуты модели можно защитить от массового присваивания. В действии контроллера, которое создает новые учетные записи, вы должны будете установить значение параметра `login` вручную.

Условная проверка

Все методы контроля принимают параметр :if, который позволяет во время выполнения (а не на стадии определения класса) решить, нужна ли проверка.

При вызове показанного ниже метода evaluate_condition из класса ActiveRecord::Validations ему передается значение параметра :if в качестве параметра condition и проверяемый объект модели в качестве параметра record:

```
# Определить по заданному условию condition, нужно ли проверять запись
# record (условие может быть задано в виде блока, метода или строки)
def evaluate_condition(condition, record)
  case condition
    when Symbol: record.send(condition)
    when String: eval(condition, binding)
    else
      if condition_block?(condition)
        condition.call(record)
      else
        raise ActiveRecordError,
          "Должен быть символ, строка (передаваемая eval) или Proc-объект"
      end
    end
end
```

Анализ предложения case в реализации этого метода показывает, что параметр :if можно задать тремя способами:

- Symbol – имя вызываемого метода передается в виде символа. Пожалуй, это самый распространенный вариант, обеспечивающий наивысшую производительность;
- String – задавать кусок кода на Ruby, интерпретируемый с помощью eval, может быть удобно, если условие совсем короткое. Но помните, что динамическая интерпретация кода работает довольно медленно;
- блок – Proc-объект, передаваемый методу call. Наверное, самый элегантный выбор для однострочных условий.

Замечания по поводу применения

Когда имеет смысл применять условные проверки? Ответ такой: всякий раз, когда сохраняемый объект может находиться в одном из нескольких допустимых состояний.

В качестве типичного примера (используется в подключаемом модуле acts_as_authenticated) приведем модель User (или Person), применяемую при регистрации и аутентификации:

```
validates_presence_of :password, :if => :password_required?
```

```
validates_presence_of :password_confirmation, :if =>
:password_required?
validates_length_of :password, :within => 4..40,
:if=>:password_required?
validates_confirmation_of :password, :if => :password_required?
```

Этот код не отвечает принципу DRY (то есть в нем встречаются повторения). О его переработке с помощью метода `with_options` см. в главе 14 «Регистрация и аутентификация». Там же подробно рассматривается применение и реализация подключаемого модуля `acts_as_authenticated`.

Существует лишь два случая, когда для корректности модели необходимо наличие поля пароля (в открытом виде):

```
protected

  def password_required?
    crypted_password.blank? || !password.blank?
  end
```

Первый случай – когда атрибут `crypted_password` пуст, поскольку это означает, что мы имеем дело с новым экземпляром класса `User`, еще не имеющим пароля. Второй случай – когда сам атрибут `password` не пуст; быть может, это свидетельствует об операции обновления и о том, что пользователь пытается переустановить свой пароль.

Работа с объектом Errors

Ниже приведен перечень стандартных текстов сообщений об ошибках, взятый непосредственно из кода Rails:

```
@@default_error_messages = {
  :inclusion => "is not included in the list",
  :exclusion => "is reserved",
  :invalid => "is invalid",
  :confirmation => "doesn't match confirmation",
  :accepted => "must be accepted",
  :empty => "can't be empty",
  :blank => "can't be blank",
  :too_long => "is too long (maximum is %d characters)",
  :too_short => "is too short (minimum is %d characters)",
  :wrong_length => "is the wrong length (should be %d characters)",
  :taken => "has already been taken",
  :not_a_number => "is not a number"
}
```

Как мы уже отмечали, для формирования окончательного сообщения об ошибке контроля в начало любой из этих строк дописывается имя атрибута с заглавной буквы. Не забывайте, что стандартное сообщение можно переопределить с помощью параметра `:message`.

Манипулирование набором Errors

Есть ряд методов, позволяющих вручную добавлять информацию об ошибках в набор `Errors` и изменять его состояние.

add_to_base(msg)

Добавляет сообщение об ошибке, относящееся к состоянию объекта *в целом*, а не к значению конкретного атрибута. Сообщение должно быть законченным предложением, поскольку Rails не применяет к нему никакой дополнительной обработки.

add(attribute, msg)

Добавляет сообщение об ошибке, относящееся к конкретному атрибуту. Сообщение должно быть фрагментом предложения, которое приобретает смысл после дописывания в начало имени атрибута с заглавной буквы.

clear

Как и следовало ожидать, метод `clear` очищает набор `Errors`.

Проверка наличия ошибок

Есть также два метода, позволяющих узнать, есть ли в объекте `Errors` ошибки, относящиеся к конкретным атрибутам.

invalid?(attribute)

Возвращает true или false в зависимости от наличия ошибок, относящихся к атрибуту `attribute`.

on(attribute)

Возвращает значения разных типов в зависимости от состояния набора ошибок, относящихся к атрибуту `attribute`. Возвращает `nil`, если для данного атрибута нет ни одной ошибки. Возвращает строку с сообщением об ошибке, если с данным атрибутом ассоциирована ровно одна ошибка. Возвращает массив строк с сообщениями об ошибках, если с данным атрибутом ассоциировано несколько ошибок.

Нестандартный контроль

Вот мы и подошли к вопросу о нестандартных методах контроля, к которым вы можете прибегнуть, если нормальная декларативная форма вас не устраивает.

Выше в этой главе я описал процедуру нахождения ошибок во время контроля, оговорившись, что объяснение было несколько упрощенным. Ниже приведена фактическая реализация, поскольку, на мой взгляд, она чрезвычайно элегантна, легко читается и помогает понять, куда именно можно поместить нестандартную логику контроля.

```
def valid?
  errors.clear

  run_validations(:validate)
  validate

  if new_record?
    run_validations(:validate_on_create)
    validate_on_create
  else
    run_validations(:validate_on_update)
    validate_on_update
  end

  errors.empty?
end
```

Здесь есть три обращения к методу `run_validations`, который и запускает декларативные валидаторы, если таковые были определены. Кроме того, имеется три метода обратного вызова (абстрактных?), которые специально оставлены в модуле `Validations` без реализации. При необходимости вы можете переопределить их в своей модели `ActiveRecord`.

Нестандартные методы контроля полезны для проверки состояния объекта *в целом*, а не его отдельных атрибутов. За неимением лучшего примера предположим, что вы работаете с объектом модели с тремя целочисленными атрибутами (`:attr1`, `:attr2` и `:attr3`), а также заранее вычисленной суммой (`:total`). Атрибут `:total` всегда должен быть равен сумме трех других атрибутов:

```
class CompletelyLameTotalExample < ActiveRecord::Base
  def validate
    if total != (attr1 + attr2 + attr3)
      errors.add_to_base("Сумма не сходится!")
    end
  end
end
```

Помните: чтобы пометить объект как некорректный, необходимо добавить информацию об ошибках в набор `Errors`. Значение, возвращаемое валидатором, не используется.

Отказ от контроля

Модуль Validations, примешанный к классу ActiveRecord::Base, влияет на три метода экземпляра, как видно из следующего фрагмента (взят из файла activerecord/lib/active_record/ validations.rb, входящего в дистрибутив Rails):

```
def self.included(base) # :nodoc:
  base.extend ClassMethods
  base.class_eval do
    alias_method_chain :save, :validation
    alias_method_chain :save!, :validation
    alias_method_chain :update_attribute, :validation_skipping
  end
end
```

Затрагиваются методы save, save! и update_attribute. Процедуру контроля для методов save и save! можно опустить, передав методу параметр false.

Впервые наткнувшись на вызов save(false) в коде Rails, я был слегка ошарашен. Я подумал: «Не припомню, чтобы у метода save был параметр», потом заглянул в документацию по API, и оказалось, что память меня не подводит! Заподозрив, что документация врет, я полез смотреть реализацию этого метода в классе ActiveRecord::Base. Нет никакого параметра. «Что за черт! Добро пожаловать в чудесный мир Ruby, – сказал я себе. – Как же выходит, что я не получаю ошибку о лишнем аргументе?»

В конечном итоге то ли я сам догадался, то ли кто-то подсказал: нормальный метод Base#save подменяется, когда примешивается модуль Validations, а по умолчанию так оно и есть. Из-за наличия alias_method_chain вы получаете открытый, хотя и недокументированный метод save_without_validation, и, как мне кажется, с точки зрения сопровождения, это куда понятнее, чем save(false).

А что насчет метода update_attribute? Модуль Validations переопределяет принимаемую по умолчанию реализацию, заставляя ее вызвать save(false). Это короткий фрагмент, поэтому я приведу его целиком:

```
def update_attribute_with_validation_skipping(name, value)
  send(name.to_s + '=', value)
  save(false)
end
```

Вот почему update_attribute не вызывает валидаторов, хотя родственный ему метод update_attributes вызывает; этот вопрос очень часто задают в списках рассылки. Тот, кто писал документацию по API, полагает, что это поведение «особенно полезно для булевых флагов в существующих записях».

Не знаю, правда это или нет, зато точно знаю, что это источник постоянных споров в сообществе. К сожалению, я мало что могу добавить, кроме простого совета, продиктованного здравым смыслом: «Будьте очень осторожны, применяя метод `update_attribute`. Он легко может сохранить объекты ваших моделей в противоречивом состоянии».

Заключение

В этой относительно короткой главе мы подробно рассмотрели API валидаторов в ActiveRecord. Один из самых притягательных аспектов Rails – возможность декларативно задавать критерии корректности объектов моделей.

9

Дополнительные возможности ActiveRecord

ActiveRecord – это простая структура объектно-реляционного отображения (ORM), если сравнивать ее с другими подобными модулями, например Hibernate в Java. Но пусть это не вводит вас в заблуждение – несмотря на скромный экстерьер, в ActiveRecord немало весьма продвинутых функций. Для максимально эффективной работы с Rails вы должны освоить не только основы ActiveRecord, но и понимать, например, когда имеет смысл выйти за пределы паттерна «одна таблица – один класс» или как пользоваться модулями Ruby, чтобы избавиться от дублирования и сделать свой код чище.

В этой главе мы завершим рассмотрение ActiveRecord, уделив внимание обратным вызовам, наблюдателям, наследованию с одной таблицей (single-table inheritance – STI) и полиморфным моделям. Мы также немного поговорим о метапрограммировании и основанных на Ruby предметно-ориентированных языках (domain-specific language – DSL) применительно к ActiveRecord.

Обратные вызовы

Эта функция ActiveRecord позволяет умелому разработчику подключить то или иное поведение в различные точки жизненного цикла модели, например после инициализации, перед вставкой, обновлением или удалением записи из базы данных и т. д.

С помощью обратных вызовов можно решать самые разные задачи – от простого протоколирования и модификации атрибутов перед выполнением контроля до сложных вычислений. Обратный вызов может прервать жизненный цикл. Некоторые обратные вызовы даже позволяют на лету изменять поведение класса модели. В этом разделе мы изучим все упомянутые сценарии, но сначала посмотрим, как выглядит обратный вызов. Взгляните на следующий незамысловатый пример:

```
class Beethoven < ActiveRecord::Base

  before_destroy :last_words
  ...

  protected

  def last_words
    logger.info "Рукоплещите, друзья, комедия окончена"
  end
end
```

Итак, перед смертью (пардон, уничтожением методом `destroy`) класс `Beethoven` произносит прощальные слова, которые будут запротоколированы навечно. Как мы скоро увидим, существует 14 разных способов добавить подобное поведение в модель. Но прежде, чем огласить весь список, поговорим о механизме регистрации обратного вызова.

Регистрация обратного вызова

Вообще-то, самый распространенный способ зарегистрировать обратный вызов – поместить его в начало класса, воспользовавшись типичным для Rails методом класса в стиле макроса. Но есть и более многословный путь к той же цели. Просто реализуйте обратный вызов как метод в своем классе. Иными словами, предыдущий пример можно было бы записать и так:

```
class Beethoven < ActiveRecord::Base
  ...

  protected

    def before_destroy
      logger.info "Рукоплещите, друзья, комедия окончена"
    end

end
```

Это тот редкий случай, когда более лаконичное решение оказывается хуже. На самом деле, почти всегда предпочтительнее – осмелюсь даже сказать, что в этом состоит путь Rails, – использовать обратные вызовы в виде макросов, а не реализовывать их в виде методов, и вот почему:

- объявления обратных вызовов в виде макросов размещаются в начале определения класса, то есть само наличие обратного вызова становится более очевидным по сравнению с размещением тела метода где-то в середине файла;
- обратные вызовы в виде макросов ставятся в очередь. Это означает, что к одной и той же точке в жизненном цикле можно подключить более одного метода. Обратные вызовы выполняются в том порядке, в котором были помещены в очередь;
- обратные вызовы для одной и той же точки расширения можно поставить в очередь на разных уровнях иерархии наследования, и они все равно будут работать, не замещая друг друга, как в случае методов;
- обратные вызовы, реализованные как методы модели, всегда вызываются в последнюю очередь.

Однострочные обратные вызовы

Если (и только если) процедура обратного вызова совсем коротенькая[1], вы можете добавить ее, передав блок макросу обратного вызова. Коротенькая – значит состоящая из одной строки!

```
class Napoleon < ActiveRecord::Base
  before_destroy {|r| logger.info "Josephine..." }
  ...
end
```

Защищенный или закрытый

За исключением случаев, в которых используется блок, методы обратных вызовов должны быть защищенными или закрытыми. Только не открытыми, поскольку метод обратного вызова никогда не должен вызываться из кода вне модели.

Хотите верьте, хотите нет, но есть и другие способы реализации обратных вызовов, однако их мы рассмотрим ниже. А пока приведем перечень точек расширения, к которым можно подключить обратные вызовы.

Парные обратные вызовы before/after

Всего существует 14 типов обратных вызовов, которые можно зарегистрировать в моделях! Двенадцать из них – это пары `before/after`, напри-

[1] Если вы заглядывали в исходный код старых версий Rails, то, возможно, встречали обратные вызовы в виде макросов, получающие короткую строку кода на Ruby, которую предстояло интерпретировать (`eval`) в контексте объекта модели. Начиная с Rails 1.2 такой способ добавления обратных вызовов объявлен устаревшим, потому что в подобных ситуациях всегда лучше использовать блоки.

мер before_validation и after_validation (оставшиеся два, after_initialize и after_find – особые случаи, которые мы обсудим позже).

Перечень обратных вызовов

Ниже приведен перечень точек расширения, вызываемых в ходе операции save (этот перечень немного различен для сохранения новой и существующей записи):

- before_validation;
- before_validation_on_create;
- after_validation;
- after_validation_on_create;
- before_save;
- before_create (для новых записей) и before_update (для существующих записей);
- ActiveRecord обращается к базе данных и выполняет INSERT или UPDATE;
- after_create (для новых записей) и before_update (для существующих записей);
- after_save.

Для операций удаления определены еще два обратных вызова:

- before_destroy;
- ActiveRecord обращается к базе данных и выполняет DELETE;
- after_destroy вызывается после замораживания всех атрибутов (они делаются доступными только для чтения).

Прерывание выполнения

Если вы вернете из метода обратного вызова булево значение false (не nil), то ActiveRecord прервет цепочку выполнения. Больше никакие обратные вызовы не активируются. Метод save возвращает false, а save! возбуждает исключение RecordNotSaved.

Имейте в виду, что в Ruby метод неявно возвращает значение последнего вычисленного выражения, поэтому при написании обратных вызовов часто допускают ошибку, которая приводит к непреднамеренному прерыванию выполнения. Если объект, содержащий обратные вызовы, по какой-то таинственной причине не хочет сохраняться, проверьте, не возвращает ли обратный вызов false.

Примеры применения обратных вызовов

Разумеется, решение о том, какой обратный вызов использовать в данной ситуации, зависит от целей, которых вы хотите добиться. Я не мо-

гу предложить ничего лучшего, чем ряд примеров, которые могут навести вас на полезные мысли при написании собственных программ.

Сброс форматирования атрибутов с помощью before_validate_on_create

Типичный пример использования обратных вызовов `before_validate` касается очистки введенных пользователем атрибутов. Например, в следующем классе `CreditCard` (цитирую документацию по Rails API) атрибут `number` предварительно нормализуется, чтобы предотвратить ложные срабатывания валидатора:

```
class CreditCard < ActiveRecord::Base
  ...
  private

    def before_validation_on_create
      # Убрать из номера кредитной карты все, кроме цифр
      self.number = number.gsub(/[^0-9]/, "")
    end
end
```

Геокодирование с помощью before_save

Предположим, что ваше приложение хранит адреса и умеет наносить их на карту. *Перед сохранением* для адреса необходимо выполнить геокодирование (нахождение координат), чтобы потом можно было легко поместить точку на карту[1].

Как часто бывает, сама формулировка требования наводит на мысль об использовании обратного вызова `before_save`:

```
class Address < ActiveRecord::Base
  include GeoKit::Geocoders

  before_save :geolocate
  validates_presence_of :line_one, :state, :zip
  ...

  private

    def geolocate
      res = GoogleGeocoder.geocode(to_s)
      self.latitude = res.lat
      self.longitude = res.lng
    end
end
```

[1] Рекомендую отличный подключаемый модуль GeoKit for Rails, который находится по адресу *http://geokit.rubyforge.org/*.

Прежде чем двигаться дальше, сделаем парочку замечаний. Предыдущий код прекрасно работает, если геокодирование завершилось успешно. А если нет? Надо ли в этом случае сохранять запись? Если не надо, цепочку выполнения следует прервать:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
  return false if not res.success # прервать выполнение

  self.latitude = res.lat
  self.longitude = res.lng
end
```

Но остается еще одна проблема – вызывающая программа (а, стало быть, и конечный пользователь) ничего не знает о том, что цепочка была прервана. Хотя мы сейчас не находимся в валидаторе, я думаю, что было бы уместно поместить информацию об ошибке в набор `errors`:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
  if res.success
    self.latitude = res.lat
    self.longitude = res.lng
  else
    errors.add_to_base("Ошибка геокодирования. Проверьте адрес.")
    return false
  end
end
```

Если выполнить геокодирование не удалось, мы добавляем сообщение об ошибке для объекта в целом, прерываем выполнение и не сохраняем запись.

Перестраховка с помощью before_destroy

Что если приложение обрабатывает очень важные данные, которые, будучи раз введены, уже не должны удаляться? Может быть, имеет смысл вклиниться в механизм удаления ActiveRecord и вместо того, чтобы реально удалять запись, просто пометить ее как удаленную?

В следующем примере предполагается, что в таблице `accounts` имеется колонка `deleted_at` типа `datetime`.

```
class Account < ActiveRecord::Base
  ...
  def before_destroy
    update_attribute(:deleted_at, Time.now) and return false
  end

end
```

Я решил реализовать этот обратный вызов в виде метода, гарантируя тем самым, что он будет выполнен последним в очереди, связанной

с точкой расширения `before_destroy`. Он возвращает `false`, поэтому выполнение прерывается и запись не удаляется из базы данных[1].

Пожалуй, стоит отметить, что при определенных обстоятельствах Rails позволяет случайно обойти обратные вызовы `before_destroy`:

- методы `delete` и `delete_all` класса `ActiveRecord::Base` почти идентичны. Они напрямую удаляют строки из базы данных, не создавая экземпляров моделей, а это означает, что никаких обратных вызовов не будет;
- объекты моделей в ассоциациях, заданных с параметром `:dependent => :delete_all`, удаляются напрямую из базы данных одновременно с удалением из набора с помощью методов ассоциации `clear` или `delete`.

Стирание ассоциированных файлов с помощью after_destroy

Если с объектом модели ассоциированы какие-то файлы, например вложения или загруженные картинки, то при удалении объекта можно стереть и эти файлы с помощью обратного вызова `after_destroy`. Хорошим примером может служить следующий метод из великолепного подключаемого модуля AttachmentFu[2] Рика Олсона:

```
# Стирает файл. Вызывается из обратного вызова after_destroy
def destroy_file
  FileUtils.rm(full_filename)
  ...
rescue
  logger.info "Исключение при стирании #{full_filename ... }"
  logger.warn $!.backtrace.collect { |b| " > #{b}" }.join("\n")
end
```

Особые обратные вызовы: after_initialize и after_find

Обратный вызов `after_initialize` активируется при создании новой модели `ActiveRecord` (с нуля или из базы данных). Его наличие позволяет обойтись без переопределения самого метода `initialize`.

Обратный вызов `after_find` активируется, когда `ActiveRecord` загружает объект модели из базы данных и *предшествует* `after_initialize`, если реализованы оба. Поскольку методы поиска вызывают `after_find` и `after_`

---

[1] В реальной программе надо было бы модифицировать еще все методы поиска, так чтобы в часть `WHERE` добавлялось условие `deleted_at is NULL`; в противном случае записи, помеченные как удаленные, будут по-прежнему видны. Это нетривиальная задача, но, к счастью, вам не придется решать ее самостоятельно. Рик Олсон написал подключаемый модуль ActsAsParanoid, который именно это и делает; вы можете найти его по адресу *http://svn.techno-weenie.net/projects/plugins/acts_as_paranoid*.

[2] Скачать AttachmentFu можно по адресу *http://svn.techno-weenie.net/projects/plugins/attachment_fu*.

initialize для каждого найденного объекта, то из соображений производительности реализовывать их следует как методы, а не как макросы.

Что если нужно выполнить некоторый код только при первом создании экземпляра модели, а не после каждой его загрузки из базы? Такой обратный вызов не предсумотрен, но это можно сделать с помощью after_initialize. Просто добавьте проверку на новую запись:

```
def after_initialize
  if new_record?
  ...
  end
end
```

Написав много приложений Rails, я обнаружил, что удобно хранить предпочтения пользователя в сериализованном хеше, ассоциированном с объектом User. Реализовать эту идею позволяет метод serialize моделей ActiveRecord, который прозрачно сохраняет граф объектов Ruby в текстовой колонке таблицы в базе данных. К сожалению, ему нельзя передать значение по умолчанию, поэтому я вынужден задавать его самостоятельно:

```
class User < ActiveRecord::Base
  serialize :preferences # по умолчанию nil
  ...

  private

    def after_initialize
      self.preferences ||= Hash.new
    end
end
```

В обратном вызове after_initialize я могу автоматически заполнить атрибут preferences модели пользователя, записав в него пустой хеш, поэтому мне не придется проверять его на nil при таком способе доступа: user.preferences [:show_help_text] = false. Конечно, хранить в сериализованном виде имеет смысл только данные, которые не будут фигурировать в SQL-запросах.

Средства метапрограммирования Ruby в сочетании с возможностью выполнять код в момент загрузки модели с помощью обратного вызова after_find – это поистине «гремучая смесь». Поскольку мы еще не закончили изучение обратных вызовов, я вернусь к вопросу об использовании after_find ниже в разделе «Модификация классов ActiveRecord во время выполнения».

Классы обратных вызовов

Достаточно часто возникает желание повторно использовать код обратного вызова для нескольких объектов. Поэтому Rails позволяет писать так называемые *классы* обратных вызовов. Вам нужно лишь пере-

дать в очередь данного обратного вызова объект, который отвечает на имя этого обратного вызова и принимает объект модели в качестве параметра.

Вот пример из раздела о перестраховке, записанный в виде класса обратного вызова:

```
class MarkDeleted
  def self.before_destroy(model)
    model.update_attribute(:deleted_at, Time.now) and return false
  end
end
```

Поскольку класс `MarkDeleted` не обладает состоянием, я реализовал обратный вызов в виде метода *класса*. Поэтому не придется создавать объекты `MarkDeleted` только ради вызова этого метода. Достаточно просто передать класс в очередь обратного вызова моделей, которые должны обладать поведением «пометить вместо удаления»:

```
class Account < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end

class Invoice < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end
```

Несколько методов обратных вызовов в одном классе

Нет такого закона, который запрещал бы иметь более одного метода обратного вызова в классе обратного вызова. Например, можно реализовать специальные требования к контрольному журналу:

```
class Auditor
  def initialize(audit_log)
    @audit_log = audit_log
  end

  def after_create(model)
    @audit_log.created(model.inspect)
  end

  def after_update(model)
    @audit_log.updated(model.inspect)
  end

  def after_destroy(model)
    @audit_log.destroyed(model.inspect)
  end
end
```

Чтобы добавить к классу ActiveRecord механизм записи в контрольный журнал, нужно сделать вот что:

```
class Account < ActiveRecord::Base
  after_create Auditor.new(DEFAULT_AUDIT_LOG)
  after_update Auditor.new(DEFAULT_AUDIT_LOG)
  after_destroy Auditor.new(DEFAULT_AUDIT_LOG)
  ...
end
```

Но это же коряво – добавлять три объекта Auditor в трех строчках. Можно было бы завести локальную переменную auditor, но так повторения все равно не избежать. Вот удобный случай воспользоваться механизмом *открытости классов* в Ruby, который позволяет модифицировать классы, не являющиеся частью вашего приложения.

Не лучше было бы просто поместить предложение acts_as_audited в начало модели, нуждающейся в контрольном журнале? Можно добавить его даже в класс ActiveRecord::Base, и тогда средства ведения журнала будут доступны всем моделям.

В своих проектах я помещаю сляпанный «на скорую руку» код, подобный приведенному в листинге 9.1, в файл lib/core_ext/active_record_base.rb, но вы можете решить по-другому. Можно даже оформить его в виде подключаемого модуля (детали см. в главе 19 «Расширение Rails с помощью подключаемых модулей»). Не забудьте только затребовать его в файле config/environment.rb, а то он никогда не загрузится.

Листинг 9.1. Сляпанный на скорую руку метод acts as audited

```
class ActiveRecord::Base
  def self.acts_as_audited(audit_log=DEFAULT_AUDIT_LOG)
    auditor = Auditor.new(audit_log)
    after_create auditor
    after_update auditor
    after_destroy auditor
  end
end
```

Теперь код класса Account уже не выглядит таким загроможденным:

```
class Account < ActiveRecord::Base
  acts_as_audited
  ...
end
```

Тестопригодность

После добавления методов обратного вызова в класс модели необходимо проверить, что они корректно работают в сочетании с моделью, в которую добавлены. Иногда это проблематично, иногда нет. Классы обратных вызовов, напротив, очень легко тестировать автономно.

Следующий тестовый метод проверяет правильность функционирования класса обратного вызова `Auditor` (с помощью библиотеки Mocha, которую можно загрузить с сайта *http://mocha.rubyforge.org/*):

```
def test_auditor_logs_created
  (model = mock).expects(:inspect).returns('foo')
  (log = mock).expects(:created).with('foo')

  Auditor.new(log).after_create(model)
end
```

В главе 17 «Тестирование» и в главе 18 «RSpec on Rails» рассматриваются методики тестирования с помощью библиотек `Test::Unit` и `RSpec` соответственно.

Наблюдатели

Принцип одной функции (single responsibility principle) – один из столпов объектно-ориентированного программирования. Его смысл в том, что у каждого класса должна быть единственная функция. В предыдущем разделе вы узнали об обратных вызовах – полезной возможности моделей ActiveRecord, которая позволяет подключать новое поведение к разным точкам жизненного цикла объекта модели. Даже если поместить дополнительное поведение в классы обратных вызовов, их наличие все равно требует вносить изменения в определение класса самой модели. С другой стороны, Rails предоставляет механизм расширения, полностью прозрачный для класса модели, – это наблюдатели (`Observer`).

Вот как можно реализовать функциональность класса обратного вызова `Auditor` в виде наблюдателя за объектами `Account`:

```
class AccountObserver < ActiveRecord::Observer
  def after_create(model)
    DEFAULT_AUDIT_LOG.created(model.inspect)
  end

  def after_update(model)
    DEFAULT_AUDIT_LOG.updated(model.inspect)
  end

  def after_destroy(model)
    DEFAULT_AUDIT_LOG.destroyed(model.inspect)
  end
end
```

Соглашения об именовании

При создании подкласса, наследующего классу `ActiveRecord::Observer`, часть Observer в имени подкласса отщепляется. В случае класса `AccountObserver` из предыдущего примера ActiveRecord знает, что наблюдать нужно за классом `Account`. Однако не всегда такое поведение же-

лательно. На самом деле для такого универсального класса, как Auditor, это было бы даже шагом в неверном направлении, поэтому предоставляется возможность переопределить указанное соглашение с помощью метода-макроса observe. Мы по-прежнему расширяем класс ActiveRecord::Observer, но свободны в выборе имени подкласса и можем явно сообщить ему, за чем наблюдать:

```
class Auditor < ActiveRecord::Observer
  observe Account, Invoice, Payment

  def after_create(model)
    DEFAULT_AUDIT_LOG.created(model.inspect)
  end

  def after_update(model)
    DEFAULT_AUDIT_LOG.updated(model.inspect)
  end

  def after_destroy(model)
    DEFAULT_AUDIT_LOG.destroyed(model.inspect)
  end
end
```

Регистрация наблюдателей

Если бы не существовало места, где Rails мог бы найти зарегистрированных наблюдателей, они вообще никогда не загрузились бы, потому что никаких ссылок на них из кода приложения нет. В главе 1 «Среда и конфигурирование Rails» мы упоминали, что в сгенерированном для вашего приложения файле config/environment.rb есть закомментированная строка, в которой можно определить подлежащих загрузке наблюдателей:

```
# Активировать наблюдателей, которые должны работать постоянно
config.active_record.observers = [:auditor]
```

Момент оповещения

Структура извещает наблюдателей о событиях перед срабатыванием добавленных в объект обратных вызовов. В противном случае было бы невозможно воздействовать на объект в целом, например в наблюдателе before_destroy, до того как выполнились собственные обратные вызовы объекта.

Наследование с одной таблицей

Очень многие приложения начинаются с разработки вариации на тему класса User. Со временем появляются различные виды пользователей, и между ними надо как-то проводить различие. Так возникают классы

Admin и Guest, являющиеся подклассами User. Теперь общее поведение можно оставить в User, а поведение подтипа перенести в подкласс. При этом все данные о пользователях можно по-прежнему хранить в таблице users – нужно лишь завести колонку type, где будет находиться имя класса, объект которого нужно создать для представления данной строки.

Вернемся к упоминавшемуся выше классу Timesheet и продолжим рассмотрение наследования с одной таблицей на его примере. Нам нужно знать, сколько оплачиваемых часов billable_hours еще не оплачено для данного пользователя. Подойти к вычислению этой величины можно разными способами, мы решили добавить метод экземпляра в класс Timesheet:

```
class Timesheet < ActiveRecord::Base
  ...

  def billable_hours_outstanding
    if submitted?
      billable_weeks.map(&:total_hours).sum
    else
      0
    end
  end

  def self.billable_hours_outstanding_for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
  end
end
```

Я вовсе не хочу сказать, что это хороший код. Он работает, но неэффективен, а предложение if/else не вызывает восторга. Недостатки становятся очевидны, когда появляется требование помечать табель Timesheet как оплаченный. Нам приходится снова модифицировать метод billable_hours_outstanding:

```
def billable_hours_outstanding
  if submitted? and not paid?
    billable_weeks.map(&:total_hours).sum
  else
    0
  end
end
```

Это изменение – вопиющее нарушение приниципа *открытости-закрытости*[1], который понуждает нас писать код так, чтобы он был открыт для расширения, но закрыт для модификации. Принцип нарушен, потому что нам пришлось изменить метод billable_hours_outstanding, чтобы учесть новое состояние объекта Timesheet. В таком простом

[1] Хорошее краткое изложение имеется на странице *http://en.wikipedia.org/wiki/Open/closed_principle*.

примере это, возможно, не кажется серьезной проблемой, но подумайте, сколько ветвей пришлось бы добавить в класс `Timesheet`, чтобы реализовать такую функциональность, как `paid_hours` (оплаченные часы) и `unsubmitted_hours` (не представленные к оплате часы).

И каково же решение проблемы с постоянно изменяющимся условным предложением? Поскольку вы читаете раздел о наследовании с одной таблицей, то, надо думать, не удивитесь, узнав, что мы рекомендуем объектно-ориентированное наследование. Для этого разобьем исходный класс `Timesheet` на четыре:

```
class Timesheet < ActiveRecord::Base
  # код, не имеющий отношения к делу, опущен

  def self.billable_hours_outstanding_for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
  end
end

class DraftTimesheet < Timesheet
  def billable_hours_outstanding
    0
  end
end

class SubmittedTimesheet < Timesheet
  def billable_hours_outstanding
    billable_weeks.map(&:total_hours).sum
  end
end
```

Если позже потребуется обсчитывать частично оплаченные табели, то нужно будет просто добавить новое поведение в виде класса `PaidTimesheet`. И никаких условных предложений!

```
class PaidTimesheet < Timesheet
  def billable_hours_outstanding
    billable_weeks.map(&:total_hours).sum - paid_hours
  end
end
```

Отображение наследования на базу данных

Задача эффективного отображения наследования объектов на реляционную базу данных не имеет универсального решения. Мы затронем лишь одну стратегию, которую Rails поддерживает изначально. Называется она *наследование с одной таблицей* (single-table inheritance), или (для краткости) *STI*.

В случае STI вы заводите в базе данных одну таблицу, в которой хранятся все объекты, принадлежащие данной иерархии наследования.

В ActiveRecord для этой таблицы выбирается имя, основанное на корневом классе иерархии. В нашем примере она будет называться timesheets.

Но ведь именно так она и была названа раньше, разве нет? Да, однако для поддержки STI нам придется добавить в нее колонку type, где будет находиться строковое представление типа хранимого объекта. Для модификации базы данных подойдет следующая миграция:

```
class AddTypeToTimesheet < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :type, :string
  end

  def self.down
    remove_column :timesheets, :type
  end
end
```

Значение по умолчанию не нужно. Коль скоро в модель ActiveRecord добавлена колонка type, Rails автоматически будет помещать в нее правильное значение. Полюбоваться этим поведением мы можем в консоли:

```
>> d = DraftTimesheet.create
>> d.type
=> 'DraftTimesheet'
```

Когда вы пытаетесь найти объект с помощью любого из методов find, определенных в базовом классе с поддержкой STI, Rails автоматически создает объекты подходящего подкласса. Особенно это полезно в таких случаях, как рассматриваемый пример с табелем, когда мы извлекаем все записи, относящиеся к конкретному пользователю, а затем вызываем методы, которые по-разному ведут себя в зависимости от класса объекта:

```
>> Timesheet.find(:first)
=> #<DraftTimesheet:0x2212354...>
```

Говорит Себастьян...

Слово type употребляется для именования колонок очень часто, в том числе для целей, не имеющих никакого отношения к STI. Именно поэтому вам, скорее всего, доводилось сталкиваться с ошибкой ActiveRecord::SubclassNotFound. Rails видит колонку type в классе Car и пытается найти класс SUV, которого не существует.

Решение простое: скажите Rails, что для STI нужно использовать другую колонку:

```
set_inheritance_column "not_sti"
```

Rails не станет жаловать на отсутствие такой колонки, а просто проигнорирует ее.

Недавно текст сообщения был изменен, чтобы оно лучше объясняло причину, но существует немало разработчиков, которые лишь мельком смотрят на сообщение, а потом тратят долгие часы, пытаясь понять, что не так с моделью (есть также немало читателей, пропускающих врезки в книгах, но я, по крайней мере, удвоил вероятность того, что они заметят эту проблему).

Замечания об STI

Хотя Rails существенно упрощает наследование с одной таблицей, стоит помнить о нескольких подводных камнях.

Начнем с того, что *запрещается заводить в двух подклассах атрибуты с одинаковым именем, но разного типа*. Поскольку все подклассы хранятся в одной таблице, такие атрибуты должны храниться в одной колонке таблицы. Честно говоря, проблемой это может стать лишь тогда, когда вы неправильно подошли к моделированию данных.

Гораздо важнее другое: *в любом подклассе на каждый атрибут должна отводиться только одна колонка, и любой атрибут, не являющийся общим для всех подклассов, должен допускать значение nil*. В подклассе `PaidTimesheet` есть колонка `paid_hours`, не встречающаяся больше ни в каких подклассах. Подклассы `DraftTimesheet` и `SubmittedTimesheet` не используют эту колонку и оставляют ее равной `null` в базе данных. Для контроля данных в колонках, не являющихся общими для всех подклассов, необходимо пользоваться валидаторами `ActiveRecord`, а не средствами СУБД.

Кроме того, *не стоит заводить подклассы со слишком большим количеством уникальных атрибутов*. Иначе в таблице базы данных будет много колонок, содержащих `null`. Обычно, появление в дереве наследования подклассов с большим числом уникальных атрибутов свидетельствует о том, что вы допустили ошибку при проектировании и должны переработать проект. Если STI-таблица выходит из-под контроля, то, быть может, для решения вашей задачи наследование непригодно. А, может быть, базовый класс слишком абстрактный?

Наконец, при работе с унаследованными базами данных может случиться так, что вместо `type` для колонки придется выбрать другое имя. В таком случае задайте имя колонки в своем базовом классе с помощью метода класса `set_inheritance_column`. Для класса `Timesheet` можно поступить следующим образом:

```
class Timesheet < ActiveRecord::Base
  set_inheritance_column 'object_type'
end
```

Теперь Rails будет автоматически помещать тип объекта в колонку `object_type`.

STI и ассоциации

Во многих приложениях, особенно для управления данными, можно встретить модели, очень похожие с точки зрения данных, но различающиеся поведением и ассоциациями. Если до перехода на Rails вы работали с другими объектно-ориентированными языками, то, наверное, привыкли разбивать задачу на иерархические структуры.

Взять, например, приложение Rails, занимающиеся учетом населения в штатах, графствах, городах и пригородах. Все это – местности, поэтому возникает желание определить STI-класс `Place`, как показано в листинге 9.2. Для ясности я включил также схему базы данных[1]:

Листинг 9.2. Схема базы данных Places и класс Place

```
# == Schema Information
#
# Table name: places
#
# id          :integer(11) not null, primary key
# region_id   :integer(11)
# type        :string(255)
# name        :string(255)
# description :string(255)
# latitude    :decimal(20, 1)
# longitude   :decimal(20, 1)
# population  :integer(11)
# created_at  :datetime
# updated_at  :datetime

class Place < ActiveRecord::Base
end
```

`Place` – это квинтэссенция абстрактного базового класса. Его не следует инстанцировать, но в Ruby нет механизма, позволяющего гарантированно предотвратить это (ну и не страшно, это же не Java!). А теперь определим конкретные подклассы `Place`:

```
class State < Place
  has_many :counties, :foreign_key => 'region_id'
  has_many :cities, :through => :counties
end
```

[1] Если вы хотите включать автоматически сгенерированную информацию о схеме в начало классов моделей, познакомьтесь с написанным Дэйвом Томасом подключаемым модулем annotate_models, который можно скачать со страницы *http://svn.pragprog.com/Public/plugins/annotate_models*.

```
class County < Place
  belongs_to :state, :foreign_key => 'region _id'
  has_many :cities, :foreign_key => 'region _id'
end

class City < Place
  belongs_to :county, :foreign_key => 'region _id'
end
```

У вас может возникнуть искушение добавить ассоциацию `cities` в класс `State`, поскольку известно, что конструкция `has_many :through` работает как с `belongs_to`, так и с `has_many`. Тогда класс `State` принял бы такой вид:

```
class State < Place
  has_many :counties, :foreign_key => 'region_id'
  has_many :cities, :through => :counties
end
```

Оно бы и замечательно, если бы только это работало. К сожалению, в данном случае, поскольку мы опрашиваем только одну таблицу, невозможно различить разные типы объектов в таком запросе:

```
Mysql::Error: Not unique table/alias: 'places': SELECT places.* FROM
places INNER JOIN places ON places.region_id = places.id WHERE
((places.region_id = 187912) AND ((places.type = 'County'))) AND
((places.`type` = 'City' ))
```

Как заставить это работать? Лучше всего было бы использовать специфические внешние ключи, а не пытаться перегрузить семантику `region_id` во всех подклассах. Для начала изменим определение таблицы `places`, как показано в листинге 9.3.

Листинг 9.3. Пересмотренная схема базы данных Places

```
# == Schema Information
#
# Table name: places
#
# id              :integer(11) not null, primary key
# state_id        :integer(11)
# county_id       :integer(11)
# type            :string(255)
# name            :string(255)
# description     :string(255)
# latitude        :decimal(20, 1)
# longitude       :decimal(20, 1)
# population      :integer(11)
# created_at      :datetime
# updated_at      :datetime
```

Без параметра `:foreign_key` в ассоциациях подклассы упростились. Плюс, можно воспользоваться обычным отношением `has_many` от `State` к `City`, а не более сложной конструкцией `has_many :through`.

```
class State < Place
  has_many :counties
  has_many :cities
end

class County < Place
  belongs_to :state
  has_many :cities
end

class City < Place
  belongs_to :county
end
```

Разумеется, многочисленные колонки, допускающие `null`, не принесут вам признания в среде сторонников чистоты реляционных баз данных. Но это еще цветочки. Чуть ниже в этой главе мы будем более подробно заниматься полиморфными отношениями `has_many`, и уж тогда-то вы точно заслужите ненависть всех пуристов.

Абстрактные базовые классы моделей

В моделях ActiveRecord допустимо не только наследование с одной таблицей. Можно организовать обобществление кода с помощью наследования, но сохранять объекты в разных таблицах базы данных. Для этого требуется создать абстрактный базовый класс модели, который будут расширять подклассы, представляющие сохраняемые объекты. По существу, из всех рассматриваемых в настоящей главе приемов это один из самых простых.

Возьмем класс `Place` из предыдущего раздела (см. листинг 9.3) и превратим его в абстрактный базовый класс, показанный в листинге 9.4. Это совсем просто – достаточно добавить всего одну строчку:

Листинг 9.4. Абстрактный класс Place

```
class Place < ActiveRecord::Base
  self.abstract = true
end
```

Я же говорил – просто. Помечая модель `ActiveRecord` как абстрактную, вы делаете нечто противоположное созданию STI-класса с колонкой `type`. Вы говорите Rails: «Я *не* хочу предполагать, что существует таблица с именем `places`».

В нашем примере это означает, что надо будет создать отдельные таблицы для штатов, графств и городов. Возможно, это именно то, что надо. Но помните, что теперь мы уже не сможем запрашивать все подтипы с помощью такого кода:

```
Place.find(:all)
```

Абстрактные классы – это область Rails, где почти не существует авторитетных правил – помочь может только опыт и интуиция.

Если вы еще не обратили внимания, отмечу, что в иерархии моделей `ActiveRecord` наследуются как методы класса, так и методы экземпляра. А равно константы и другие члены классов, появляющиеся в результате включения модулей. Это означает, что в класс `Place` можно поместить код, полезный всем его подклассам.

Полиморфные отношения has_many

Rails позволяет определить класс, связанный отношением `belong_to` с классами разных типов. Об этом красноречиво рассказал в своем блоге Майк Байер (Mike Bayer):

> *«Полиморфная ассоциация», хотя и имеет некоторое сходство с обычным полиморфным объединением в иерархии классов, в действительности таковым не является, поскольку вы имеете дело с конкретной ассоциацией между одним конечным классом и произвольным числом исходных классов, а исходные классы не имеют между собой ничего общего. Иными словами, они не связаны отношением наследования и, возможно, хранятся в совершенно разных таблицах. Таким образом, полиморфная ассоциация относится, скорее, не к объектному наследованию, а к аспектно-ориентированному программированию (AOP); некая концепция применяется к набору различных сущностей, которые больше никак не связаны между собой. Такая концепция называется сквозной задачей (cross-cutting concern); например, все сущности предметной области должны поддерживать протокол изменений в одной таблице базы данных. А в нашем примере для ActiveRecord объекты `Order` и `User` должны иметь ссылки на объект Address*[1].

Другими словами, это не полиморфизм в объектно-ориентированном смысле слова, а некая своеобразная особенность Rails.

Случай модели с комментариями

Возвращаясь к нашему примеру о временных затратах и расходах, предположим, что с каждым из объектов классов `BillableWeek` и `Timesheet` может быть связано много комментариев (общий класс `Comment`). Наивный подход к решению этой задачи – сделать класс `Comment` принадлежащим одновременно `BillableWeek` и `Timesheet` и завести в таблице базы данных колонки `billable_week_id` и `timesheet_id`:

```
class Comment < ActiveRecord::Base
  belongs_to :timesheet
  belongs_to :expense_report
end
```

[1] *http://techspot.zzzeek.org/?p=13.*

Этот подход наивен, потому что с ним было бы трудно работать и нелегко обобщить. Помимо всего прочего, необходимо было бы включить в приложение код, гарантирующий, что никакой объект Comment не принадлежит *одновременно* объектам BillableWeek и Timesheet. Написать код для определения того, к чему присоединен данный комментарий, было бы затруднительно. Хуже того – если понадобится ассоциировать комментарии еще с каким-то классом, придется добавлять в таблицу comments еще один внешний ключ, допускающий null.

В Rails эта проблема имеет элегантное решение с помощью так называемых *полиморфных ассоциаций*, которые мы уже затрагивали, когда обсуждали параметр :polymorphic => true ассоциации belongs_to в главе 7 «Ассоциации в ActiveRecord».

Интерфейс

Для использования полиморфной ассоциации нужно определить лишь одну ассоциацию belongs_to и добавить в таблицу базы данных две взаимосвязанных колонки. И после этого к любому классу в системе можно будет присоединять комментарии (что наделяет его свойством *commentable*), не изменяя ни схему базы данных, ни саму модель Comment.

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

В нашем приложении нет класса (или модуля) Commentable. Мы назвали ассоциацию :commentable, потому что это слово точно описывает интерфейс объектов, ассоциируемых подобным способом. Имя :commentable фигурирует и на другом конце ассоциации:

```
class Timesheet < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class BillableWeek < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

Здесь мы видим ассоциацию has_many с параметром :as. Этот параметр помечает ассоциацию как полиморфную и указывает, какой интерфейс используется на другом конце. Раз уж мы заговорили об этом, отметим, что на другом конце полиморфной ассоциации belongs_to может быть ассоциация has_many или has_one, порядок работы при этом не изменяется.

Колонки базы данных

Ниже приведена миграция, создающая таблицу comments:

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
```

```
      t.column :text, :text
      t.column :commentable_id, :integer
      t.column :commentable_type, :string
    end
  end
end
```

Как видите, имеется колонка commentable_type, в которой хранится имя класса ассоциированного объекта. Принцип работы можно наблюдать в консоли Rails:

```
>> c = Comment.create(:text => "I could be commenting anything.")
>> t = TimeSheet.create
>> b = BillableWeek.create
>> c.update_attribute(:commentable, t)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "Timesheet: 1"
>> c.update_attribute(:commentable, b)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "BillableWeek: 1"
```

Видно, что оба объекта Timesheet и BillableWeek имеют один и тот же идентификатор id (1). Но благодаря атрибуту commentable_type, хранящемуся в виде строки, Rails может понять, с каким объектом связан комментарий.

Конструкция has_many :through и полиморфизм

При работе с полиморфными ассоциациями имеются некоторые логические ограничения. Например, поскольку Rails не может понять, какие таблицы необходимо соединять при наличии полиморфной ассоциации, следующий гипотетический код работать не будет:

```
class Comment < ActiveRecord::Base
  belongs_to :user
  belongs_to :commentable, :polymorphic => true
end

class User < ActiveRecord::Base
  has_many :comments
  has_many :commentables, :through => :comments
end

>> User.find(:first).comments
ActiveRecord::HasManyThroughAssociationPolymorphicError: Cannot have
a has_many :through association 'User#commentables' on the polymorphic
object 'Comment#commentable'.
```

Если вам действительно необходимо нечто подобное, то использование has_many :through с полиморфными ассоциациями все же возможно, толь-

ко надо точно указать, какой из возможных типов вам нужен. Для этого служит параметр :source_type. В большинстве случаев придется еще указать параметр :source, так как имя ассоциации не будет соответствовать имени интерфейса, заданного для полиморфной ассоциации:

```
class User < ActiveRecord::Base
  has_many :comments
  has_many :commented_timesheets, :through => :comments,
           :source => :commentable, :source_type => 'Timesheet'
  has_many :commented_billable_weeks, :through => :comments,
           :source => :commentable, :source_type => 'BillableWeek'
end
```

Это многословно, и вообще при таком подходе элегантность решения начинает теряться, но все же он работает:

```
>> User.find(:first).commented_timesheets
=> [#<Timesheet:0x575b98 @attributes={}> ]
```

Замечание об ассоциации has_many

Мы уже близимся к завершению рассмотрения ActiveRecord, а, как вы, возможно, заметили, еще не был затронут вопрос, представляющийся очень важным многим программистам: ограничения внешнего ключа в базе данных. Объясняется это тем, что путь Rails не подразумевает использования таких ограничений для обеспечения ссылочной целостности. Данный аспект вызывает, мягко говоря, противоречивые мнения, и некоторые разработчики вообще сбросили со счетов Rails (и его авторов) именно по этой причине.

Никто не мешает вам включить в схему ограничения внешнего ключа, хотя вы поступите мудро, подождав с этим до написания большей части приложения. Разумеется, полиморфные ассоциации представляют собой исключение. Это, наверное, самое яркое проявление неприятия ограничений внешних ключей со стороны Rails. Если вы не готовы идти на бой, то, наверное, не стоит привлекать внимание администратора базы данных к этой теме.

Модули как средство повторного использования общего поведения

В этом разделе мы обсудим одну из стратегий выделения функциональности, общей для не связанных между собой классов моделей. Вместо наследования мы поместим общий код в модули.

В разделе «Полиморфные отношения has_many» мы описали, как добавить комментарии, в примере, касающемся временных затрат и расходов. Продолжим работу с этим примером, так как он отлично подходит для разложения на модули.

Мы реализуем следующее требование: как пользователи, так и утверждающие должны иметь возможность добавлять комментарии к объектам `Timesheet` и `ExpenseReport`. Кроме того, поскольку наличие комментариев служит индикатором того, что табель или отчет о расходах потребовал дополнительного рассмотрения и времени на обработку, администратор приложения должен иметь возможность быстро просмотреть список недавних комментариев. Но такова уж человеческая природа, что администратор время от времени просто проглядывает комментарии, не читая их, поэтому в требованиях оговорено, что должен быть предоставлен механизм пометки комментария как прочитанного сначала утверждающим, а потом администратором.

Снова воспользуемся полиморфной ассоциацией `has_many :as`, которая положена в основу этой функциональности:

```
class Timesheet < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class ExpenseReport < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end
```

Затем создадим для администратора контроллер и действие, которое будет выводить список из 10 последних комментариев, причем каждый элемент будет ссылкой на комментируемый объект.

```
class RecentCommentsController < ApplicationController
  def show
    @recent_comments = Comment.find( :all, :limit => 10,
    :order => 'created_at DESC' )
  end
end
```

Вот простой шаблон представления для вывода недавних комментариев:

```
<ul>
  <% @recent_comments.each do |comment| %>
    <li>
      <h4><%= comment.created_at -%></h4>
      <%= comment.text %>
      <div class="meta">
        Комментарий о:
        <%= link_to comment.commentable.title,
           content_url( comment.commentable ) -%>
      </div>
    </li>
  <% end %>
</ul>
```

Пока все хорошо. Полиморфная ассоциация позволяет легко свести в один список комментарии всех типов. Но напомним, что каждый комментарий должен быть помечен «OK» утверждающим и/или администратором. Помеченный комментарий не должен появляться в списке.

Не станем здесь описывать интерфейс для одобрения комментариев. Достаточно сказать, что в классе `Comment` есть атрибут `reviewed`, который возвращает `true`, если комментарий помечен «OK».

Чтобы найти все непрочитанные комментарии к некоторому объекту, мы можем воспользоваться расширением ассоциации, изменив определения классов моделей следующим образом:

```
class Timesheet < ActiveRecord::Base
  has_many :comments, :as => :commentable do
    def approved
      find(:all, :conditions => {:reviewed => false })
    end
  end
end

class ExpenseReport < ActiveRecord::Base
  has_many :comments, :as => :commentable do
    def approved
      find(:all, :conditions => {:reviewed => false })
    end
  end
end
```

Мне этот код не нравится, и я надеюсь, что теперь вы уже понимаете почему. Он нарушает принцип DRY! В классах `Timesheet` и `ExpenseReport` имеются идентичные методы поиска непрочитанных комментариев. По сути дела, они обладают общим интерфейсом – *commentable*!

В Ruby имеется механизм определения общих интерфейсов – необходимо включить в каждый класс модуль, который содержит код, разделяемый всеми реализациями общего интерфейса.

Определим модуль `Commentable` и включим его в наши классы моделей:

```
module Commentable
  has_many :comments, :as => :commentable do
    def approved
     find( :all,
           :conditions => ['approved = ?', true ] )
    end
  end
end

class Timesheet < ActiveRecord::Base
  include Commentable
end
```

```
class ExpenseReport < ActiveRecord::Base
  include Commentable
end
```

Не работает! Чтобы исправить ошибку, необходимо понять, как Ruby интерпретирует код, в котором используются открытые классы.

Несколько слов об области видимости класса и контекстах

Во многих других интерпретируемых объектно-ориентированных языках программирования есть две фазы выполнения – сначала интерпретатор загружает определения классов и говорит «вот определение, с которым я должен работать», а потом исполняет загруженный код. Но при таком подходе очень трудно (хотя и возможно) добавлять в класс новые методы на этапе выполнения.

Напротив, Ruby позволяет добавлять методы в класс в любой момент. В Ruby, написав `class MyClass`, вы не просто сообщаете интерпретатору, что нужно определить класс, но еще и говорите: «выполни следующий код в области видимости этого класса».

Пусть имеется такой сценарий на Ruby:

```
1 class Foo < ActiveRecord::Base
2   has_many :bars
3 end

4 class Foo
5   belongs_to :spam
6 end
```

Когда интерпретатор видит строку 1, он понимает, что нужно выполнить следующий далее код (до завершающего `end`) в контексте объекта класса `Foo`. Поскольку объекта класса `Foo` еще не существует, интерпретатор создает этот класс. В строке 2 предложение `has_many :bars` выполняется в контексте объекта класса `Foo`. Что бы ни делало сообщение `has_many`, это делается сейчас.

Когда в строке 2 еще раз встретится объявление `class Foo`, мы снова попросим интерпретатор выполнить следующий далее код в контексте объекта класса `Foo`, но на этот раз интерпретатор уже будет знать о классе `Foo` и больше не создаст его. Поэтому в строке 5 мы просто говорим интерпретатору, что нужно выполнить предложение `belongs_to :spam` в контексте того же самого объекта класса `Foo`.

Чтобы можно было выполнить предложения `has_many` и `belongs_to`, эти методы должны существовать в том контексте, в котором вызваны. Поскольку они определены как методы класса `ActiveRecord::Base`, а выше мы объявили, что класс `Foo` расширяет `ActiveRecord::Base`, то этот код выполняется без ошибок.

Однако, определив модуль Commentable следующим образом:

```
module Commentable
  has_many :comments, :as => :commentable do
    def approved
      find( :all,
            :conditions => ['approved = ?', true ] )
    end
  end
end
```

мы получим ошибку при попытке выполнить в нем предложение has_many. Дело в том, что метод has_many не определен в контексте объекта модуля Commentable.

Теперь, разобравшись с тем, как Ruby интерпретирует код, мы понимаем, что предложение has_many на самом деле должно быть выполнено в контексте включающего класса.

Обратный вызов included

К счастью, в классе Module определен удобный обратный вызов, который позволит нам достичь желаемой цели. Если в объекте Module определен метод included, то он будет вызываться при каждом включении этого модуля в другой модуль или класс. В качестве аргумента ему передается объект модуля/класса, в который включен данный модуль.

Мы можем определить метод included в объекте модуля Commentable, так чтобы он выполнял предложение has_many в контексте включающего класса (Timesheet, ExpenseReport и т. д.):

```
module Commentable
  def self.included(base)
    base.class_eval do
```

Говорит Кортенэ...

Тут надо соблюсти тонкий баланс. Такие магические трюки, как include Commentable, конечно, позволяют вводить меньше текста, и модель выглядит проще, но это может также означать, что код вашей ассоциации делает такие вещи, о которых вы не подозреваете. Можно легко запутаться и потратить долгие часы, пока трассировка программы не заведет совсем в другой модуль.

Лично я предпочитаю оставлять все ассоциации в модели и расширять их с помощью модуля. Тогда весь перечень ассоциаций виден в коде модели как на ладони:

```
has_many :comments, :as => :commentable, :extend =>
Commentable
```

```
      has_many :comments, :as => :commentable do
        def approved
          find(:all, :conditions => ['approved = ?', true ])
        end
      end
    end
  end
end
```

Теперь при включении модуля `Commentable` в классы наших моделей предложение `has_many` будет выполняться так, будто мы написали его в теле соответствующего класса.

Модификация классов ActiveRecord во время выполнения

Средства метапрограммирования Ruby в сочетании с обратным вызовом `after_find` открывают двери для ряда интересных возможностей, особенно если вы готовы к размыванию границ между кодом и данными. Я говорю о модификации поведения классов модели *на лету*, в момент, когда они загружаются в приложение.

В листинге 9.5 приведен сильно упрощенный пример такой техники в предположении, что в модели существует колонка `config`. При выполнении обратного вызова `after_find` мы получаем описатель уникального *синглетного* (singleton) класса[1] экземпляра загруженной модели. Затем с помощью метода `class_eval` выполняется содержимое атрибута `config`, принадлежащего данному экземпляру класса `Account`. Так как мы делаем это с помощью синглетного класса, относящегося только к данному экземпляру, а не с помощью глобального класса `Account`, то на остальных экземплярах учетных записей в системе это ровным счетом никак не сказывается.

Листинг 9.5. Метапрограммирование во время выполнения в обратном вызове after_find

```
class Account < ActiveRecord::Base
  ...
  private

    def after_find
      singleton = class << self; self; end
```

[1] Не думаю, что эти слова что-то значат для вас, если вы не знакомы с идеей синглетных классов в Ruby и возможностью интерпретировать произвольные строки как Ruby-код во время выполнения. Неплохой отправной точкой может служить статья *http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html*.

```
        singleton.class_eval(config)
      end
    end
```

Я применял подобные приемы в приложении для управления цепочкой поставщиков, которое писал для большой промышленной компании. В промышленности для описания одной отгрузки товара употребляется термин *партия*. Атрибуты и бизнес-логика обработки данной партии зависят непосредственно от поставщика и товара. Поскольку множество поставщиков и товаров изменяется еженедельно (а то и ежедневно), система должна была допускать переконфигурацию без повторного развертывания.

Не вдаваясь в подробности, скажу, что приложение было написано так, что сопровождающие его программисты могли легко настраивать поведение системы, манипулируя хранящимся в базе данных Ruby-кодом, ассоциированным с товаром, входящим в партию.

Например, с партиями масла, отгружаемыми в адрес компании Acme Dairy Co., могло быть ассоциировано бизнес-правило, требующее, чтобы код товара состоял ровно из 10 цифр. Тогда в базе данных для масла, предназначенного для Acme Dairy, хранились бы такие строчки:

```
validates_numericality_of :product_code, :only_integer => true
validates_length_of       :product_code, :is => 10
```

Замечания

Сколько-нибудь полное описание всего того, что можно достичь за счет метапрограммирования в Ruby, и способов правильно это делать составило бы целую книгу. Например, вы бы поняли, что выполнение произвольного Ruby-кода, хранящегося в базе, – штука опасная. Поэтому я еще раз подчеркиваю, что все примеры сильно упрощены. Я лишь хочу познакомить вас с имеющимися возможностями.

Если вы решите применять такие приемы в реальных приложениях, нужно принять во внимание безопасность, порядок утверждения и многие другие аспекты. Быть может, вы захотите выполнять не произвольный Ruby-код, а ограничиться небольшим подмножеством языка, достаточным для решаемой задачи. Вы можете спроектировать компактный API или даже разработать предметно-ориентированный язык (DSL), предназначенный специально для выражения бизнес-правил и поведений, которые должны загружаться динамически. Все глубже проваливаясь в кроличью нору, вы, возможно, загоритесь идеей написать специализированные анализаторы своего языка, которые могли бы исполнять его в различных контекстах: для обнаружения ошибок, формирования отчетов и т. п. В этой области возможности безграничны.

Ruby и предметно-ориентированные языки

Мой бывший коллега Джей Филдс и я были первыми, кто применил сочетание метапрограммирования на Ruby и *внутренних*[1] предметно-ориентированных языках в ходе разработки приложений Rails для клиентов компании ThoughtWorks. Я все еще иногда выступаю на конференциях с докладами о создании DSL на Ruby и пишу об этом в своем блоге.

Джей тоже продолжал писать и рассказывать об эволюции техники разработки Ruby DSL, которую он называет *естественными языками бизнеса* (Business Natural Languages, сокращенно BNL[2]). При разработке BNL вы проектируете предметно-ориентированный язык, который синтаксически может отличаться от Ruby, но достаточно близок к нему, чтобы программу можно было легко преобразовать в корректный Ruby-код и выполнить на этапе выполнения, как показано в листинге 9.6.

Листинг 9.6. Пример естественного языка бизнеса

```
employee John Doe
compensate 500 dollars for each deal closed in the past 30 days
compensate 100 dollars for each active deal that closed more than
365 days ago
compensate 5 percent of gross profits if gross profits are greater
than
1,000,000 dollars
compensate 3 percent of gross profits if gross profits are greater
than
2,000,000 dollars
compensate 1 percent of gross profits if gross profits are greater
than
3,000,000 dollars
```

Возможность использовать такие продвинутые приемы, как DSL, – еще один могучий инструмент в руках опытного разработчика для Rails.

---

[1] Уточнение «внутренний» служит, чтобы отличить предметно-ориентированный язык, полностью погруженный в некий язык общего назначения, например Ruby, от языка, для которого требуется специально написанный анализатор.

[2] Если поискать слово BNL в Google, то вы получите кучу ссылок на сайт Barenaked Ladies, находящийся в Торонто, поэтому лучше уж сразу отправляйтесь к первоисточнику по адресу *http://bnl.jayfields.com*.

Говорит Кортенэ...

Все DSL – отстой! За исключением, конечно, написанных Оби. Читать и писать на DSL-языке может только его автор. Когда проект переходит к другому разработчику, часто проще сделать все заново, чем разбираться в разных хитростях и заучивать слова, которые допустимо употреблять в имеющемся DSL-языке.

На самом деле, метапрограммирование в Ruby – тоже отстой. Люди, которым дали в руки этот новенький инструмент, часто не знают меры. Я считаю, что метапрограммирование – `self.included`, `class_eval` и им подобные – лишь портят код в большинстве проектов.

Если вы пишете веб-приложение, то программисты, которые присоединятся к разработке и сопровождению проекта в будущем, скажут спасибо, если вы будете использовать простые, ясные, четко очерченные и хорошо протестированные методы, а не залезать в существующие классы или прятать ассоциации в модулях.

Ну а если, прочитав все это, вы все-таки решите попробовать и справитесь с задачей... что ж, ваш код может оказаться мощнее и выразительнее, чем вы можете себе представить.

Заключение

Этой главой мы завершаем рассмотрение `ActiveRecord` – одного из самых важных и мощных структур, встроенных в Rails. Мы видели, как обратные вызовы и наблюдатели помогают элегантно структурировать код в объектно-ориентированном духе. Мы также пополнили свой арсенал моделирования техникой наследования с одной таблицей и уникальными для `ActiveRecord` полиморфными отношениями.

К этому моменту мы рассмотрели две составных части паттерна MVC: модель и контроллер. Настало время приступить к третьей и последней части: видам, или представлениям.

10

ActionView

У самого великого и самого тупого есть две общие черты. Вместо того чтобы изменять свои представления в соответствии с фактами, они пытаются подогнать факты под свои представления... и это может оказаться очень неудобно, если одним из фактов, нуждающихся в изменении, являетесь вы сами.

Doctor Who[1]

Контроллеры – это скелет и мускулатура приложения Rails. Продолжая аналогию, можно сказать, что модели – это ум и сердце приложения, а шаблоны представлений (основанные на библиотеке `ActionView`, третьем из основных компонентов Rails) – его кожа, то есть то, что видно внешнему миру.

`ActionView` – это часть Rails API, предназначенная для сборки визуального компонента приложения, то есть HTML-разметки и связанного с ней контента, который отображается в броузере, когда кто-нибудь обращается к приложению. На самом деле, в прекрасном новом мире REST-ресурсов `ActionView` отвечает за генерацию любой информации, исходящей из приложения.

`ActionView` включает полноценную систему шаблонов, основанную на библиотеке ERb для Ruby. Она получает от контроллера данные и объединяет их с кодом представления, образуя презентационный уровень для конечного пользователя.

---

[1] Главный герой научно-фантастического сериала компании BBC. – *Примеч. перев.*

В этой главе мы рассмотрим фундаментальные принципы структуры ActionView, начиная с основ шаблонов и заканчивая эффективным использованием подшаблонов (partial), а также значительным повышением быстродействия за счет кэширования.

Основы ERb

Стандартные файлы шаблонов в Rails пишутся на диалекте Ruby, который называется *Embedded Ruby*, или ERb. Библиотека ERb входит в дистрибутив Ruby и не является уникальной особенностью Rails.

ERb-документ обычно содержит статическую HTML-разметку, перемежающуюся кодом на Ruby, который динамически исполняется во время рендеринга шаблона. Коль скоро вы вообще занимаетесь программированием для Rails, то, конечно, знаете, как Ruby-код, вставляемый в ERb-документ, обрамляется парой ограничителей.

Существует два вида ограничителей, которые служат разным целям и работают точно так же, как их аналоги в технологиях JSP и ASP, с которыми, вы, возможно, знакомы:

```
<% %> и <%= %>
```

Код между ограничителями исполняется всегда. Разница в том, что в первом случае возвращаемое значение отбрасывается, а во втором – подставляется в текст, формируемый шаблоном.

Очень распространенная ошибка – *случайное использование отбрасывающего ограничителя при необходимости подставляющего*. Вы будете рвать на себе волосы, пытаясь понять, почему нужное значение не выводится на экран, хотя никаких ошибок не выдается.

Практикум по ERb

Поэкспериментировать с ERb можно и вне Rails, так как интерпретатор ERb – стандартная часть Ruby. Поэтому можно попрактиковаться в написании и обработке ERb-шаблонов с помощью этого интерпретатора. Вызывается он командной утилитой erb.

Например, введите в файл (скажем, demo.erb) такой код:

```
Перечислим все методы класса string в Ruby.

Для начала нам понадобится строка.

<% str = "Я - строка!" %>

Итак, строка у нас есть: вот она:

<%= str %>

А теперь посмотрим на ее методы:
```

```
<% str.methods.sort[0...10].each_with_index do |m,i| %>
  <%= i+1 %>. <%= m %>
<% end %>
```

Теперь подайте этот файл на вход `erb`:

```
$ erb demo.erb
```

Вот что вы увидите:

```
Перечислим все методы класса string в Ruby.

Для начала нам понадобится строка.


Итак, строка у нас есть: вот она:

Я - строка!

А теперь посмотрим на ее методы: -- может быть, не все, чтобы не уходить за
пределы экрана


1. %

2. *

3. +

4. <

5. <<

6. <=

7. <=>

8. ==

9. ===

10. =~
```

Как видите, весь Ruby-код внутри ограничителей выполнился, включая присваивание переменной `str` и итерации в цикле `each`. Но только код внутри ограничителей со знаком равенства дал вклад в выходную информацию, сформированную в результате выполнения шаблона.

Возможно, вы обратили внимание на пустые строки. Наличие заключенного в ограничители кода никак не отражается на подсчете строчек. Строчка – она и есть строчка, поэтому следующий код выводится как пустая строка:

```
<% end %>
```

Удаление пустых строк из вывода ERb

Rails позволяет избавиться хотя бы от части ненужных пустых строк за счет использования модифицированных ограничителей:

```
<%- str.methods.sort[0...10].each_with_index do |m,i| -%>
  <%= i+1 %>. <%= m %>
<%- end -%>
```

Обратите внимание на знаки минус в ограничителях – они подавляют *начальные пробелы* и *избыточные переходы на новую строку* в выводе шаблона. При правильном использовании можно заметно улучшить внешний вид выводимой по шаблону информации. Конечному пользователю это безразлично, но может облегчить изучение HTML-разметки, формируемой приложением.

Закомментирование ограничителей ERb

Символ комментария `#`, принятый в Ruby, работает и для ограничителей ERb. Просто поместите его после знака процента в открывающем теге ограничителя.

```
<%#= str %>
```

Содержимое закомментированного ERb-тега игнорируется. Оставлять закомментированные фрагменты, замусоривая шаблон, не стоит, но для временного отключения они полезны.

Условный вывод

Одна из самых распространенных идиом при написании представлений в Rails – условный вывод содержимого. Простейший способ условного управления выводом заключается в использовании предложений `if/else` в сочетании с тегами `<% %>`:

```
<% if @show_subtitle %>
<h2><%= @subtitle %></h2>
<% end %>
```

Часто можно воспользоваться постфиксными условиями `if` и тем самым сократить код, поскольку тег `<%=` игнорирует значение `nil`:

```
<h2><%= @subtitle if @show_subtitle %></h2>
```

Конечно, в предыдущем примере таится потенциальная проблема. Первая, более длинная, форма условного вывода полностью исключает теги `<h2>`, вторая – нет.

Есть два способа решить эту проблему, не отказываясь от однострочной формы.

Уродливое решение мне доводилось встречать в некоторых приложениях Rails, и только поэтому я его здесь привожу:

```
<%= "<h2>#{@subtitle}</h2>" if @show_subtitle %>
```

Ну отвратительно же! В более элегантном решении применяется метод-помощник Rails `content_tag`:

```
<%= content_tag('h2', @subtitle) if @show_subtitle %>
```

Методы-помощники – как включенные в Rails, так и написанные вами самостоятельно, – это основной инструмент для построения элегантных шаблонов представлений. Подробно они рассматриваются в главе 11 «Все о помощниках».

RHTML? RXML? RJS?

В версии Rails 2.0 принято называть файлы с ERb-шаблонами, добавляя расширение `.erb`, а в более ранних версиях использовалось расширение `.rhtml`.

Для шаблонов существует еще два стандартных формата и два суффикса:

- `.builder` (ранее `.rxml`) – говорит Rails, что шаблон нужно выполнять с помощью библиотеки `Builder::XmlMarkup`, написанной Джимом Вайрихом (Jim Weirich). Она позволяет легко генерировать XML-документы. Работа с библиотекой Builder рассматривается в главе 15 «XML и ActiveResource».
- `.rjs` (не менялся) – запускает встроенные в Rails средства генерации JavaScript, об этом речь пойдет в главе 12 «Ajax on Rails».

В конце этой главы мы дадим краткий обзор дополнительных языков шаблонов, которые можно без труда интегрировать с Rails, и расскажем, когда они могут пригодиться. А пока продолжим разговор о макетах и шаблонах.

Макеты и шаблоны

В Rails приняты простые соглашения об использовании шаблонов, относящиеся к их размещению в структуре каталогов проекта Rails.

Каталог `app/views`, изображенный на рис. 10.1, содержит подкаталоги, названные по именам контроллеров, применяемых в приложении. В каталог, соответствующий контроллеру, помещаются шаблоны, имена которых соответствуют именам действий.

В специальном каталоге `app/views/layout` находятся шаблоны макетов, служащие в качестве повторно используемых контейнеров для представлений. И снова для определения того, какой шаблон выполнять, применяются соглашения; только на этот раз сопоставление производится с именем контроллера.

В случае макетов важную роль играет иерархия наследования контроллеров. Для большинства приложений Rails в каталоге макетов име-

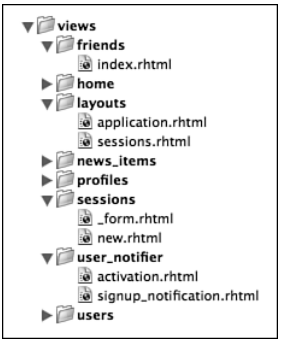

***Рис. 10.1.** Типичная структура каталога app/views*

ется файл `application.rhtml`. Его имя происходит от имени контроллера `ApplicationController`, которому обычно наследуют все прочие контроллеры приложения; поэтому он и выбран в качестве макета по умолчанию для всех представлений.

Разумеется, этот макет выбирается лишь в отсутствие специфического, но в большинстве случаев имеет смысл использовать только один шаблон уровня приложения, например показанный в листинге 10.1.

Листинг 10.1. Простой универсальный макет application.rhtml

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>My Rails Application</title>
  <%= stylesheet_link_tag 'scaffold', :media => "all" %>
  <%= javascript_include_tag :defaults %>
 </head>
<body>
 <%= yield :layout %>
</body>
</html>
```

Методы `stylesheet_link_tag` и `javascript_include_tag` — это помощники, которые автоматически вставляют в документ стандартные теги `LINK` и `SCRIPT`, необходимые для включения CSS и JavaScript-файлов. Кроме них, интерес в этом шаблоне представляет только обращение к методу `yield :layout`, которое мы сейчас и обсудим.

Подстановка содержимого

Встроенное в Ruby ключевое слово `yield` нашло элегантное применение в организации совместной работы шаблонов макета и действий.

Обратите внимание, как это слово употребляется в середине шаблона макета:

```
<body>
 <%= yield :layout %>
</body>
```

В данном случае символ `:layout` означает специальное сообщение, посылаемое системе рендеринга. Он отмечает место, в которое нужно вставить генерируемую действием выходную информацию; обычно это результат рендеринга шаблона, соответствующего действию.

В макете может быть несколько мест, в которые подставляется содержимое. Для этого достаточно несколько раз обратиться к `yield` с различными идентификаторами. В качестве примера приведем макет (конечно, упрощенный), в котором предусмотрены места для боковых колонок слева и справа:

```
<body>
  <div class="left sidebar">
    <%= yield :left %>
  </div>
  <div id="main_content">
    <%= yield :layout %>
  </div>
  <div class="right sidebar">
    <%= yield :right %>
  </div>
</body>
```

В центральный элемент `DIV` подставляется содержимое, порожденное главным шаблоном. Но как передать Rails содержимое двух боковых колонок? Легко – воспользуйтесь в коде шаблона методом `content_for`:

```
<% content_for(:left) do %>
<h2>Навигация</h2>
<ul>
  <li>...
</ul>
<% end %>

<% content_for(:right) do %>
<h2>Справка</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
   sed do eiusmod tempor incididunt ut labore et dolore magna
   aliqua. Ut enim ad minim veniam, quis nostrud ...
<% end %>

<h1>Заголовок страницы</h1>
<p>Обычное содержимое шаблона, подставляемое вместо символа
   :layout</p>
...
```

Помимо боковых колонок и иных видимых блоков, я рекомендую использовать `yield` для вставки дополнительного содержимого в элемент страницы `HEAD`, как показано в листинге 10.2. Это исключительно полезная техника, поскольку Internet Explorer иногда ведет себе непредсказуемо, если теги `SCRIPT` встречаются вне элемента `HEAD`.

Листинг 10.2. Подстановка дополнительного содержимого в заголовок

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Мое приложение Rails</title>
  <%= stylesheet_link_tag 'scaffold', :media => "all" %>
  <%= javascript_include_tag :defaults %>
  <%= yield :head %>
 </head>
 <body>
  <%= yield :layout %>
 </body>
</html>
```

Переменные шаблона

Мы видели, как работает механизм подстановки блоков содержимого в макет, но есть ли другие способы передачи данных от контроллера представлению? В ходе подготовки шаблона переменные экземпляра, которым были присвоены значения на этапе выполнения действия контроллера, копируются в одноименные переменные экземпляра в контексте шаблона.

Переменные экземпляра

Копирование переменных экземпляра – основная форма взаимодействия между контроллером и представлением, и, честно говоря, это поведение – одна из фундаментальные особенностей Rails, поэтому вы о нем, конечно, знаете:

```
class HelloWorldController < ActionController::Base
  def index
    @msg = "Здравствуй, мир!"
  end
end

# template file /app/views/hello_world/index.html.erb
<%= @msg %>
```

А вот что вы, возможно, не знаете, так это то, что из контроллера в шаблон копируются не только переменные экземпляра. Однако вклю-

чать в свой код явные зависимости от некоторых из перечисленных ниже объектов не стоит. Особенно остерегайтесь их использования в операциях с данными. Помните, что стандартное применение паттерна MVC подразумевает, что на уровне контроллера готовятся данные для рендеринга, а не само представление!

assigns

Хотите увидеть все, что пересекает границу между контроллером и представлением? Включите в шаблон строку `<%= debug(assigns) %>` и посмотрите, что она выведет. Атрибут `assigns` – часть внутреннего устройства Rails, поэтому пользоваться им напрямую в промышленном коде не следует.

base_path

Путь в локальной файловой системе к каталогу приложения, начиная с которого хранятся шаблоны:

controller

С помощью этой переменной можно получить доступ к экземпляру текущего контроллера до того, как он выйдет за пределы области видимости в конце обработки запроса. Вы можете воспользоваться тем, что контроллер знает свое собственное имя (атрибут `controller_name`) и имя только что выполненного действия (атрибут `action_name`); это позволит более эффективно структурировать CSS-стили (см. листинг 10.3).

Листинг 10.3. Классы для тега BODY образованы из имени контроллера и действия

```
<html>
  ...
  <body class="<%= controller.controller_name %>
               <%= controller.action_name %>">
  ...
  </body>
</html>
```

В результате тег `BODY` будет выглядеть примерно так (в зависимости от выполненного действия):

```
<body class="timesheets index">
```

Надеюсь, вы знаете, что буква C в аббревиатуре CSS расшифровывается, как *cascading* (каскадные), а означает это, что имена классов каскадно распространяются вниз по дереву элементов, встречающихся в разметке, и могут употребляться при создании правил. Трюк, примененный в листинге 10.3, заключается в том, что мы автоматически включили имена контроллера и действия в качестве имен классов для элемента `BODY`, поэтому в дальнейшем их можно очень гибко использовать для настройки внешнего вида страницы.

Вот как этот прием позволяет варьировать фоновую картинку для элементов класса header в зависимости от пути к контроллеру:

```
body.timesheets .header {
  background: url(../images/timesheet-bg.png) no-repeat left top
}

body.expense_reports .header {
  background: url(../images/expense-reports-bg.png) no-repeat left top
}
```

flash

flash – это переменная представления, которой вы, несомненно, будете регулярно пользоваться. Она уже проскальзывала в примерах и применяется, когда нужно отправить пользователю сообщение с уровня контроллера, но только на время следующего запроса.

В Rails часто употребляется конструкция flash[:notice] для отправки информационных сообщений и flash[:error], если сообщение более серьезно. Лично я люблю заключать их в элементы DIV, располагаемые в начале макета и позиционируемые с помощью CSS-стилей, как показано в листинге 10.4.

Листинг 10.4. Стандартизованное место для информационного сообщения и сообщения об ошибке в файле application.html.erb

```
<html>
  ...
  <body>
    <%= content_tag 'div', h(flash[:notice]),
          :class => 'notice', :id => 'notice' if flash[:notice] %>
    <%= content_tag 'div', h(flash[:error]),
          :class => 'notice error', :id => 'error' if flash[:error] %>

    <%= yield :layout %>
  </body>
</html>
```

Метод-помощник content_tag позволяет выводить все это содержимое условно. Без него мне пришлось бы заключать HTML-разметку в блок if, что сделало бы описанную схему довольно громоздкой.

headers

В переменной headers хранятся значения HTTP-заголовков, сопровождающих обрабатываемый запрос. В представлении они могут понадобиться разве что для того, чтобы посмотреть на них в целях отладки. Поместите в любое место макета строку <%= debug(headers) %>, и вы увидите в броузере (после обновления страницы, конечно) что-то вроде:

```
--
Status: 200 OK
cookie:
```

```
- - adf69ed8dd86204d1685b6635adae0d9ea8740a0
Cache-Control: no-cache
```

logger

Хотите записать что-то в протоколы во время рендеринга представления? Воспользуйтесь методом `logger`, чтобы получить экземпляр класса `Logger`. Если вы ничего не меняли, то по умолчанию это будет `RAILS_DEFAULT_LOGGER`.

params

Это тот же самый хеш `params`, который доступен контроллеру; он содержит пары имя/значение, указанные в запросе. Иногда я напрямую использую значения из хеша `params` в представлении, особенно когда речь идет о страницах с фильтрацией и сортировкой строк:

```
<p>Фильтр по месяцу:
<%= select_tag(:month_filter,
options_for_select(@month_options, params[:month_filter])) %>
```

С точки зрения безопасности крайне нежелательно помещать неочищенные данные из запроса в выходной поток, формируемый шаблоном. В следующем разделе «Защита целостности представления от данных, введенных пользователем» мы рассмотрим эту тему более подробно.

request и *response*

Представлению доступны объекты `request` и `response`, соответствующие запросу и ответу HTTP, но, помимо отладки, я не вижу для них других применений в шаблоне.

session

В переменной `session` хранится хеш, представляющий сеанс пользователя. Возможно, бывают ситуации, когда значения из него можно использовать для изменения рендеринга, но меня в дрожь бросает от мысли, что вы захотите устанавливать параметры сеанса из уровня представления. Применяйте с осторожностью и, в основном, для отладки, как `request` и `response`.

Защита целостности представления от данных, введенных пользователем

Если данные, необходимые приложению, вводятся пользователем или поступают из иного не заслуживающего доверия источника, то не забывайте о необходимости экранировать и обезопасить содержимое шаблона. В противном случае вы распахнете двери для самых разнообразных хакерских атак.

Рассмотрим, например, следующий фрагмент шаблона, который всего лишь копирует значение `params[:page_number]` в выходной поток:

```
<h1>Результаты поиска</h1>
<h2>Страница <%= params[:page_number] %></h2>
```

Простой способ включить номер страницы, не так ли? Но подумайте, что произойдет, если кто-нибудь отправит этой странице запрос, содержащий вложенный тег SCRIPT и некоторое вредоносное значение в качестве параметра page_number. Бах! Злонамеренный код попал прямо в ваш шаблон!

К счастью, существует совсем несложный способ предотвратить такие атаки, и, поскольку разработчики Rails ожидают, что вы будете пользоваться им часто, они присвоили соответствующему методу однобуквенное имя h:

```
<h1>Результаты поиска</h1>
<h2>Страница <%=h(params[:page_number]) %></h2>
```

Метод h *экранирует* HTML-содержимое – вместо того, чтобы включать его напрямую в разметку, знаки < и > заменяются соответствующими компонентами, в результате чего попытки внедрения вредоносного кода терпят неудачу. Разумеется, на содержимое, в котором нет разметки, это не оказывает никакого влияния.

Но что, если необходимо отобразить введенную пользователем HTML-разметку, как часто бывает в блогах, где допустимы форматированные комментарии? В таком случае попробуйте воспользоваться методом sanitize из класса ActionView::Helpers::TextHelper. Он уберет теги, которые наиболее часто применяются для атак: FORM и SCRIPT, а все прочие оставит без изменения. Метод sanitize подробно рассматривается в главе 11.

Подшаблоны

Подшаблоном (partial) называется фрагмент кода шаблона. В Rails подшаблоны применяются для разбиения кода представления на отдельные блоки, из которых можно собирать макеты с минимумом дублирования. Синтаксически подшаблон начинается со строки render :partial => "name". Перед именем шаблона должен стоять подчерк, что позволяет визуально отличить его от других файлов в каталоге шаблонов. *Однако при ссылке на подшаблон знак подчерка опускается.*

Простые примеры

В простейшем случае подшаблон используется для оформления части кода шаблона. Некоторые разработчики таким образом разбивают шаблоны на логически независимые части. Иногда понять структуру страницы легче, если разложить ее на существенные фрагменты. Например, в листинге 10.5 приведена простая страница регистрации, разбитая на ряд подшаблонов.

Листинг 10.5. Простая форма регистрации пользователя с подшаблонами

```
<h1>Регистрация пользователя</h1>
<%= error_messages_for :user %>
<% form_for :user, :url => users_path do -%>
  <table class="registration">
   <tr>
    <td class="details demographics">
     <%= render :partial => 'details' %>
     <%= render :partial => 'demographics' %>
    </td>
    <td class="location">
     <%= render :partial => 'location' %>
    </td>
   </tr>
   <tr>
    <td colspan="2"><%= render :partial => 'opt_in' %></td>
   </tr>
   <tr>
    <td colspan="2"><%= render :partial => 'terms' %></td>
   </tr>
  </table>
  <p><%= submit_tag 'Зарегистрироваться' %></p>
<% end -%>
```

И давайте сразу посмотрим на один из подшаблонов. Для экономии места возьмем самый маленький – содержащий флажки, которые описывают настройки данного приложения. Его код приведен в листинге 10.6; обратите внимание, что имя файла начинается с подчерка.

Листинг 10.6. Подшаблон с настройками в файле app/views/users/_opt_in.html.erb

```
<fieldset id="opt_in">
  <legend>Spam Opt In</legend>
  <p><%= check_box :user, :send_event_updates %>
    Посылать извещения о новых событиях!<br/>
  <%= check_box :user, :send_site_updates%>
    Уведомлять меня о новых службах</p>
</fieldset>
```

Лично я предпочитаю заключать подшаблоны в семантически значимые контейнеры в разметке. В случае подшаблона из листинга 10.6 оба флажка помещены внутрь элемента `<fieldset>`, которому присвоен атрибут `id`. Следование этому неформальному правилу помогает мне мысленно представлять, как содержимое данного подшаблона соотносится с родительским шаблоном. Если бы речь шла о другой разметке, например вне формы, возможно, вместо `<fieldset>` я выбрал бы контейнер `<div>`.

Почему не оформить в виде подшаблонов содержимое тегов `<td>`? Это вопрос стиля – я люблю, когда весь скелет разметки находится в одном

месте. В данном случае скелетом является табличная структура, показанная в листинге 10.5. Если бы части данной таблицы находились в подшаблонах, это лишь затемнило бы общую верстку страницу. Признаю, что это область личных предпочтений, и могу лишь поделиться, как удобнее работать лично мне.

Повторное использование подшаблонов

Поскольку форма регистрации аккуратно разбита на компоненты, легко создать простую форму редактирования, в которой некоторые из этих компонентов используются повторно (листинг 10.7).

Листинг 10.7. Простая форма редактирования данных о пользователе, повторно использующая некоторые подшаблоны

```
<h1>Редактирование пользователя</h1>
<%= error_messages_for :user %>
<% form_for :user, :url => user_path(@user),
                   :html => { :method => :put } do -%>
  <table class="settings">
   <tr>
    <td class="details">
     <%= render :partial => 'details' %>
    </td>
    <td class="demographics">
     <%= render :partial => 'demographics' %>
    </td>
   </tr>
   <tr>
    <td colspan="2" class="opt_in">
     <%= render :partial => 'opt_in' %>
    </td>
   </tr>
  </table>
  <p><%= submit_tag 'Сохранить настройки' %></p>
<% end -%>
```

Сравнив листинги 10.5 и 10.7, вы увидите, что в форме редактирования структура таблицы несколько изменилась, и содержимого в ней меньше, чем в форме регистрации. Возможно, адресная информация более подробно вводится на отдельной странице, и уж конечно вы не хотите заставлять пользователя принимать соглашение при каждом изменении настроек.

Разделяемые подшаблоны

До сих пор мы рассматривали работу с подшаблонами, находящимися в том же каталоге, что и их родительский шаблон. Однако никто не мешает ссылаться на подшаблоны, хранящиеся в других каталогах, –

достаточно просто написать в начале имя каталога. Но знак подчерка все равно следует опускать, что может показаться немного странным.

Добавим в конец формы регистрации листинга 10.5 подшаблон captcha, мешающий спамерам автоматически регистрироваться:

```
    ...
    <tr>
     <td colspan="2"><%= render :partial => 'terms' %></td>
    </tr>
    <tr>
     <td colspan="2"><%= render :partial => 'shared/captcha' %></td>
    </tr>
   </table>
   <p><%= submit_tag 'Зарегистрироваться' %></p>
  <% end -%>
```

Так как подшаблон captcha используется в разных частях приложения, имеет смысл поместить его в общую папку, а не в папку конкретного представления. Однако, преобразуя существующий код шаблона в разделяемый подшаблон, нужно проявлять осторожность. Очень легко неосознанно создать подшаблон, который будет неявно зависеть от того, откуда выполняется его рендеринг.

Рассмотрим, например, участника списка рассылки Rails-talk с некорректным подшаблоном в файле login/_login.rhtml:

```
<% form_tag do %>
 <label>Имя:</label>
  <%= text_field_tag :username, params[:username] %>
  <br />
  <label>Пароль:</label>
  <%= password_field_tag :password, params[:password] %>
  <br />
  <%= submit_tag "Войти" %>
<% end %>
```

Отправка формы работает, если подшаблон выводится как часть действия контроллера login (на «странице входа»), но перестает работать, когда этот же подшаблон включается в шаблон представления любой другой части сайта. Проблема в том, что метод form_tag (рассматривается в следующей главе) обычно принимает необязательный параметр action, который говорит, куда отправлять информацию. Если этот параметр опущен, форма отправляется странице с текущим URL, а он зависит от страницы, в которую был включен разделяемый подшаблон.

Передача переменных подшаблонам

Подшаблоны наследуют переменные экземпляра, доступные родительскому шаблону. Именно поэтому работают методы-помощники (text, password и т. п.), встречающиеся в листингах 10.5 и 10.7, – они неявно

полагаются на то, что в области видимости есть переменная `@user`. Мне кажется, что в некоторых случаях таким неявным разделением переменных вполне можно пользоваться, особенно когда подшаблон тесно связан со своим родителем. Сомнений вообще не возникало бы, если бы единственной причиной разбиения на подшаблоны было стремление уменьшить размер и сложность особенно больших шаблонов.

Однако, когда у вас появляется привычка выделять подшаблоны с целью повторного использования в разных местах приложения, вводить зависимости от неявно передаваемых переменных становится рискованно. Поэтому Rails поддерживает передачу подшаблону переменных с локальной областью видимости с помощью параметра-хеша `:locals`:

```
render :partial => 'shared/address', :locals => { :form => form }
```

Имена и значения, переданные в хеше `:locals`, преобразуются в локальные переменные (без префикса @) подшаблона. В листинге 10.8 приведена вариация на тему шаблона страницы регистрации. На этот раз мы воспользовались вариантом метода `form_for`, который передает блоку параметр `form`, представляющий саму форму. Этот параметр далее передается подшаблонам.

Листинг 10.8. Простой шаблон страницы регистрации пользователя, в котором форма передается как локальная переменная

```
<h1>Регистрация пользователя</h1>
<%= error_messages_for :user %>
<% form_for :user, :url => users_path do |form| -%>
  <table class="registration">
   <tr>
    <td class="details address demographics">
     <%= render :partial => 'details',
                :locals => {:form => form } %>
     <%= render :partial => 'shared/address',
                :locals => {:form => form } %>
    </td>
   </tr>
  </table>
  <p><%= submit_tag 'Зарегистрироваться' %></p>
<% end -%>
```

И наконец, в листинге 10.9 приведена разделяемая форма для ввода адреса.

Листинг 10.9. Простой разделяемый подшаблон для ввода адреса с использованием локальной переменной

```
<fieldset class="address">
 <legend>Адрес</legend>
 <p><label>Улица</label><br/>
   <%= form.text_area :street, :rows => 2, :cols => 40 %></p>
 <p><label>Город</label><br/>
```

```
    <%= form.text_field :city %></p>
  <p><label>Штат</label><br/>
    <%= form.text_field :state, :size => 2 %></p>
  <p><label>Почтовый индекс</label><br/>
    <%= form.text_field :zip, :size => 15 %></p>
</fieldset>
```

У методов-помощников для обработки форм, которые мы будем рассматривать в главе 11, есть варианты для вызова с переменной `form`, поставляемой методом `form_for`. Именно ее мы и передали в подшаблоны с помощью хеша `:locals`.

Хеш local_assigns

При необходимости проверить наличие некоторой локальной переменной искать ее надо в хеше `local_assigns`, который является частью любого шаблона. Конструкция `defined? variable` работать не будет в силу ограничений, присущих системе рендеринга.

```
<% if local_assigns.has_key? :special %>
<%= special %>
<% end %>
```

Рендеринг наборов

Одно из самых удачных применений подшаблонов – рендеринг наборов. Привыкнув оформлять рендеринг с помощью подшаблонов, вы уже не захотите вновь загромождать свои шаблоны уродливыми циклами `for` и `each`.

```
render :partial => 'entry', :collection => @entries
```

Коротко, ясно и опирается на соглашение об именовании. Самое существенное тут – способ, которым подшаблон узнает отображаемый объект. Объект записывается в локальную переменную с тем же именем, что у самого подшаблона. Подшаблон, соответствующий предыдущему фрагменту, должен был бы сослаться на локальную переменную `entry`.

```
<%= div_for(entry) do %>
  <%= h(entry.description) %>
  <%= distance_of_time_in_words_to_now entry.created_at %> ago
<% end %>
```

Переменная partial_counter

Существует еще одна малоизвестная переменная, устанавливаемая в подшаблонах для рендеринга наборов. Это счетчик с начальным значением 0, отслеживающий, сколько раз уже был выведен подшаблон. Он полезен для вывода нумерованных списков. Имя переменной образуется из имени подшаблона и суффикса `_counter`.

```
<%= div_for(entry) do %>
  <%= entry_counter %>:
  <%= h(entry.description) %>
  <%= distance_of_time_in_words_to_now entry.created_at %> ago
<% end %>
```

Разделяемые подшаблоны для рендеринга наборов

Если бы вы захотели использовать подшаблон для наборов при выводе единственного объекта, то этот объект следовало бы передать в хеше `:locals`, описанном в предыдущем разделе:

```
render :partial => 'entry', :locals => {:entry => @entry }
```

Мне встречался следующий прием, позволяющий избежать передачи параметра `locals`:

```
<% entry = @entry if @entry %>
<% div_for(entry) do %>
  <%= h(entry.description) %>
  <%= distance_of_time_in_words_to_now entry.created_at %> ago
<% end %>
```

Это работает, но код некрасивый, содержит повторения и неявно зависит от наличия необязательной переменной `@entry`. Не делайте так. Пользуйтесь параметром `:locals`, который и предназначен для подобных задач.

Протоколирование

Заглянув в журнал разработки, вы увидите в нем записи о том, какие подшаблоны выводились и сколько на это потребовалось времени:

```
Rendering template within layouts/application
Rendering listings/index
Rendered listings/_listing (0.00663)
Rendered listings/_listing (0.00296)
Rendered listings/_listing (0.00173)
Rendered listings/_listing (0.00159)
Rendered listings/_listing (0.00154)
Rendered layouts/_login (0.02415)
Rendered layouts/_header (0.03263)
Rendered layouts/_footer (0.00114)
```

Кэширование

Механизм кэширования в Rails позволяет создавать приложения, которые очень быстро отвечают при развертывании в режиме эксплуатации. Кэшировать можно все: от целых страниц до их фрагментов. Кэшированные части хранятся на диске в виде HTML-файлов и отправляются веб-сервером в ответ на последующие запросы при минимуме

участия со стороны Rails. В Rails поддерживается три способа кэширования:

- Кэширование страниц. Вся информация, сгенерированная действием контроллера, записывается на диск, и дальнейшая обработка происходит без участия диспетчера Rails;
- Кэширование действий. Вся информация, сгенерированная действием контроллера, записывается на диск, но диспетчер Rails все же участвует в обработке последующих запросов, и выполняются фильтры контроллера;
- Кэширование фрагментов. Можно кэшировать на диске произвольные части сгенерированной страницы, чтобы не тратить время на их рендеринг в будущем.

Кэширование в режиме разработки?

Я хотел с самого начала упомянуть, что в режиме разработки кэширование отключено. Если вы хотите поэкспериментировать, измените следующую строку в файле `config/environments/development.rb`:

```
config.action_controller.perform_caching = false
```

Но, конечно, не забудьте восстановить ее в исходном виде, перед тем как сохранять проект в репозитории, если не хотите столкнуться с некоторыми сбивающими с толку ошибками[1].

Кэширование страниц

Кэширование страниц – это простейший вид кэширования. Для его включения служит метод-макрос `caches_page` в контроллере. Он говорит Rails о том, что нужно записать на диск весь ответ на запрос, чтобы в дальнейшем его мог отдавать сам веб-сервер без вмешательства со стороны диспетчера. При этом не будет производиться запись в протокол Rails, не будут срабатывать фильтры контроллера – вообще Rails никак себя не проявит, как будто речь идет о статическом HTML-файле в каталоге `public` проекта.

Кэширование действий

По определению, если нечто может изменяться при каждом запросе или в зависимости от того, кто просматривает страницу, то кэширование не допускается. С другой стороны, если нам нужно лишь выполнить фильтры, которые проверят некоторые условия перед отображе-

[1] В замечательном ролике на эту тему Джеффри Грозенбэк предлагает ввести в проект еще один режим с названием `development_with_caching`, где кэширование отключено в экспериментальных целях (*http://peepcode.com/products/page-action-and-fragment-caching*).

нием запрошенной страницы, подойдет метод caches_action. Он делает почти то же самое, что и кэширование страниц, только перед возвратом кэшированного HTML-файла выполняются фильтры контроллера. Это позволяет осуществить дополнительную обработку или даже переадресовать пользователя на другую страницу.

Кэширование действий реализовано на базе кэширования фрагментов (рассматривается ниже) и around-фильтра (см. главу 2 «Работа с контроллерами»). Содержимое кэшированного действия индексируется текущим хостом и путем, то есть механизм работает, даже если приложение Rails обслуживает несколько субдоменов с помощью маскирования DNS. Кроме того, различные представления одного и того же ресурса, например HTML и XML, трактуются как разные запросы и кэшируются отдельно.

В этом разделе примеры кода будут относится к демонстрационному приложению lil_journal[1]. В приложении есть как открытые, так и закрытые для публичного просмотра записи, поэтому в умалчиваемом режиме следует выполнить фильтр, который проверит, зарегистрирован ли пользователь, и при необходимости переадресует его на действие public. В листинге 10.10 приведен код контроллера EntriesController.

Листинг 10.10. Контроллер EntriesController в приложении lil_journal

```
class EntriesController < ApplicationController

  before_filter :check_logged_in, :only => [:index]

  caches_page :public
  caches_action :index

  def public
    @entries = Entry.find(:all,
                          :limit => 10,
                          :conditions => {:private => false})
    render :action => 'index'
  end

  def index
    @entries = Entry.find(:all, :limit => 10)
  end

  private

    def check_logged_in
      redirect_to :action => 'public' unless logged_in?
    end

end
```

[1] Subversion URL: *http://obiefernandez.com/svn/projects/awruby/prorails/lil_journal*.

Действие public отображает только открытые записи и доступно всем, поэтому является кандидатом на кэширование страницы. Но, поскольку у него нет собственного шаблона, мы явно вызываем в конце действия метод render :action => 'index'.

Замечания о проектировании

Априорная информация о том, что приложению потребуется кэширование, должна учитываться при проектировании. В проектах с необязательной аутентификацией часто имеются такие действия контроллеров, для которых кэширование страниц или действий невозможно, поскольку они обрабатывают оба состояния «зарегистрированности» внутри себя. Так произошло бы и в программе из листинга 10.10, если бы действие index обрабатывало вывод открытых и закрытых записей:

```
def index
  opts = {}
  opts[:limit] = 10
  opts[:conditions] = {:private => false } unless logged_in?
  @posts = Entry.find(:all, opts)
end
```

Как правило, бывает не слишком много полностью статических страниц, которые можно кэшировать с помощью методов cache_page или cache_action, поэтому наступает черед кэширования фрагментов.

Кэширование фрагментов

Пользователи уже привыкли к динамичным страницам, и в макете вашего приложения наверняка будут области для приветствия или счетчика уведомлений. Механизм кэширования фрагментов позволяет сохранить на диске части выводимой страницы и при последующих запросах отправлять их пользователям, не выполняя рендеринг заново. Повышение производительности не так ощутимо, как при кэшировании страниц или действий, поскольку диспетчеру Rails все-таки приходится вмешиваться. Тем не менее скорость работы приложения можно существенно увеличить.

Метод cache

По своей природе кэширование фрагментов определяется в шаблоне представления, а не на уровне контроллера. Для этого служит метод cache класса ActionView. Он принимает блок, позволяющий обернуть подлежащее кэшированию содержимое.

После того как пользователь регистрируется в приложении Lil' Journal, в заголовке должна отображаться информация о нем, поэтому вопрос о кэшировании действий для индексной страницы даже не стоит. Мы уберем директиву action_cache из EntriesController, но оставим директиву cache_page для действия public. Затем откроем шаблон entries/

index.html.erb и добавим кэширование фрагментов, как показано в листинге 10.11.

Листинг 10.11. Шаблон entries/index.html.erb с кэшированием фрагментов в приложении Lil' Journal

```
<%= content_tag :h1, "#{@user.name}'s Journal" %>
<% cache do %>
  <%= render :partial => 'entry', :collection => @entries %>
<% end %>
```

Вот, собственно, и все. Теперь HTML-разметка, сгенерированная в результате рендеринга набора записей, сохранена в кэше фрагментов, ассоциированном со страницей entries/index. Этого достаточно, если вы кэшируете только один фрагмент на странице, но обычно необходимо назначить фрагменту дополнительный идентификатор.

Именованные фрагменты

Метод cache принимает необязательный параметр name:

```
<% cache "my_fragment" do %>
```

Если не задавать его, как мы и поступили в листинге 10.11, то в качестве ключа доступа к кэшу используется URL объемлющей страницы. Такой способ подходит, когда на странице кэшируется только один фрагмент.

Если же кэшируется несколько фрагментов страницы, необходим дополнительный идентификатор, чтобы не возникло конфликтов имен. В листинге 10.12 приведен улучшенный вариант страницы со списком записей, где мы добавили боковую колонку с перечнем недавних комментариев.

Листинг 10.12. Страница со списком записей, в которой присутствуют две директивы кэширования фрагментов

```
<%= content_tag :h1, "#{@user.name}'s Journal" %>

<% cache(:fragment => 'entries') do %>
  <%= render :partial => 'entry', :collection => @entries %>
<% end %>

<%- content_for :sidebar -%>

  <% cache(:fragment => 'recent_comments') do %>
  <%= render :partial => 'comment', :collection => @recent_comments
%>
  <% end %>

<% end %>
```

После рендеринга этого кода в кэше будут храниться два фрагмента с такими ключами:

- `/entries/index?fragment=entries`
- `/entries/index?fragment=recent_comments`

Индексация фрагментов по URL страницы представляет собой изящное решение довольно трудной задачи. Представьте, например, что произошло бы, если бы мы добавили в приложение Lil' Journal разбиение на страницы и захотели получить записи на второй странице. Без каких бы то ни было усилий с нашей стороны дополнительные фрагменты попали бы в кэш с такими ключами:

- `/entries/index?page=2&fragment=entries`
- `/entries/index?page=2&fragment=recent_comments`

Примечание

В Rails для конструирования уникальных идентификаторов фрагментов применяется вспомогательный метод `url_for`. *Не требуется, чтобы ключи фрагментов соответствовали реальным URL, встречающимся в приложении.*

Глобальные фрагменты

Иногда требуется кэшировать фрагменты, связанные не только с URL единственной страницы приложения. Чтобы добавить в кэш фрагменты с глобальными ключами, мы снова воспользуемся параметром `name` метода-помощника `cache`, но на этот раз передадим в качестве его значения строку, а не хеш.

Для демонстрации данной методики потребуем, чтобы приложение Lil' Journal отображало на каждой странице статистические сведения о пользователе. В листинге 10.13 подшаблон `stats` кэшируется для каждого пользователя, причем в качестве ключа выступает имя пользователя с суффиксом `"_stats"`.

Листинг 10.13. Страница со списком записей, на которой отображается глобальная статистика пользователя

```
<%= content_tag :h1, "#{@user.name}'s Journal" %>

<% cache(:fragment => 'entries') do %>
  <%= render :partial => 'entry', :collection => @entries %>
<% end %>

<%- content_for :sidebar -%>

  <% cache(@user.name + "_stats") do %>
    <%= render :partial => 'stats' %>
  <% end %>
```

```
    <% cache(:fragment => 'recent_comments') do %>
      <%= render :partial => 'comment', :collection => @recent_comments %>
    <% end %>

<% end %>
```

При обсуждении кэширования мы упустили из виду один вопрос – истечение срока хранения *устаревшего* кэшированного содержимого, которое уже не соответствует реальным данным.

Устранение ненужных обращений к базе данных

Поместив фрагменты представления в кэш, бессмысленно обращаться к базе за получением данных для этих фрагментов. Ведь результаты запросов все равно не будут использоваться, пока не истечет срока хранения фрагментов в кэше. Метод `read_fragment` позволяет проверить, существует ли кэшированное содержимое; он принимает те же параметры, что и ассоциированный с ним метод `cache`.

Вот как следует модифицировать действие `index`:

```
def index
  unless read_fragment(:fragment => 'entries')
    @entries = Entry.find(:all, :limit => 10)
  end
end
```

Теперь метод поиска будет выполняться, только если кэш необходимо обновить.

Истечение срока хранения кэшированного содержимого

При применении кэширования необходимо рассмотреть все ситуации, которые могут привести к устареванию кэша. А затем написать код, который удалит старое содержимое, освободив, так сказать, место для более свежего.

Очистка кэша страниц и действий

Методы `expire_page` и `expire_action` позволяют явно удалить содержимое из кэша так, чтобы при следующем запросе оно сгенерировалось заново. Для идентификации удаляемого содержимого используется тот же метод `url_for`, что и в других местах Rails. В листинге 10.14 показано, как включить очистку в метод `create` контроллера `entries`.

Листинг 10.14. Действие create контроллера entries

```
1 def create
2   @entry = @user.entries.build(params[:entry])
3   if @entry.save
4     expire_page :action => 'public'
5     redirect_to entries_path(@entry)
```

```
6   else
7     render :action => 'new'
8   end
9 end
```

Обратите внимание на строку 4, где из кэша явно удаляется страница, ассоциированная с действием `public`. Но, если подумать, обнаружится, что не только действие `create` делает кэш недействительным. То же самое относится к действиям `update` и `destroy`.

Разрабатывая свои приложения, особенно в стиле REST, помните, что различные представления одного и того же ресурса считаются разными запросами и кэшируются отдельно. Если вы кэшировали XML-представление действия, то для стирания его из кэша необходимо добавить параметр `:format => :xml` в спецификацию действия.

Очистка кэша фрагментов

Ой! Я почти забыл (честное слово), что надо очищать еще и кэшированные фрагменты, для чего предназначен метод `expire_fragment`. С учетом этого действие `create` будет выглядеть так:

```
def create
  @entry = @user.entries.build(params[:entry])
  if @entry.save
    expire_page :action => 'public'
    expire_fragment(:fragment => 'entries')
    expire_fragment(:fragment => (@user.name + "_stats"))
    redirect_to entries_path(@entry)
  else
    render :action => 'new'
  end
end
```

Использование регулярных выражений в методах очистки кэша

В процедуре очистки, которую мы добавили в действие `create`, все еще (!) осталась серьезная проблема. Если помните, мы говорили, что механизм кэширования фрагментов будет работать и в случае разбиения списка записей на страницы, причем ключи фрагментов при этом выглядят так:

```
'/entries/index?page=2&fragment=entries'
```

Но если ограничиться лишь вызовом `expire_fragment(:fragment => 'entries')`, из кэша будут удалены только фрагменты для первой страницы. Поэтому метод `expire_fragment` понимает также регулярные выражения, и мы должны этим воспользоваться:

```
expire_fragment(r%{entries/.*})
```

Однако должен же быть более удобный способ сделать кэш недействительным. Очень не хочется помнить о необходимости вставлять кучу сложных предложений удаления из кэша во все действия. Кроме того, кэширование – это отдельная задача, и, похоже, подходить к ней нужно с позиций аспектно-ориентированного программирования.

Автоматическая очистка кэша с помощью дворников

Класс `Sweeper` (дворник) во многом напоминает объект `Observer` из библиотеки `ActiveRecord`, но предназначен специально для очистки кэшированного содержимого. Вы говорите дворнику, за изменением каких моделей он должен наблюдать, как делаете это для классов обратных вызовов и наблюдателей.

В листинге 10.15 показан дворник, который следит за надлежащим кэшированием страниц со списком записей в приложении Lil' Journal.

Листинг 10.15. Дворник для страниц со списком записей в приложении Lil' Journal

```
class EntrySweeper < ActionController::Caching::Sweeper
  observe Entry

  def expire_cached_content(entry)
    expire_page :controller => 'entries', :action => 'public'
    expire_fragment(r%{entries/.*})
    expire_fragment(:fragment => (entry.user.name + "_stats"))
  end

  alias_method :after_save, :expire_cached_content
  alias_method :after_destroy, :expire_cached_content
end
```

Написав класс-дворник (который должен находиться в каталоге `app/models`), нужно приказать контроллеру использовать этот класс в сочетании с действиями. Вот как выглядит начало переработанного контроллера `entries` для приложения Lil' Journal:

```
class EntriesController < ApplicationController

  before_filter :check_logged_in, :only => [:index]
  caches_page :public
  cache_sweeper :entry_sweeper, :only => [:create, :update, :destroy]

  ...
```

Как и многие другие макросы контроллеров, метод `cache_sweeper` принимает необязательные параметры `:only` и `:except`. Не имеет смысла беспокоить дворника действиями, которые не могут изменить состояние приложения, поэтому в нашем примере мы добавим параметр `:only`.

Как и наблюдатели, дворники могут наблюдать не только за одной моделью. Но, если вы соберетесь пойти по этому пути, помните, что методы

обратных вызовов должны знать, как работать с каждой моделью. В этом случае может пригодиться предложение `case`, как показано в листинге 10.16 – полностью переработанной версии класса `EntrySweeper`, который теперь способен наблюдать не только за объектами `Entry`, но и `Comment`.

Листинг 10.16. Класс EntrySweeper, переработанный для наблюдения за объектами Entries и Comments

```
class EntrySweeper < ActionController::Caching::Sweeper
  observe Entry, Comment

  def expire_cached_content(record)
    expire_page :controller => 'entries', :action => 'public'
    expire_fragment(r%{entries/.*})

    user = case entry
      when Entry then record.user
      when Comment then record.entry.user
    end

    expire_fragment(:fragment => (user.name + "_stats"))
  end

  alias_method :after_save, :expire_cached_content
  alias_method :after_destroy, :expire_cached_content
end
```

Протоколирование работы кэша

Если вы включите кэширование в режиме разработки, то увидите в протоколе записи о кэшировании и очистке кэша:

```
Processing Entries#index (for 127.0.0.1 at 2007-07-20 23:07:09) [GET]
 ...
Cached page: /entries.html (0.03949)


Processing Entries#create (for 127.0.0.1 at 2007-07-20 23:10:50)
[POST]
 ...
Expired page: /entries.html (0.00085)
```

Это неплохой способ убедиться, что кэширование работает так, как вы хотели.

Подключаемый модуль Action Cache

Том Фейкс (Tom Fakes) и Скотт Лэрд (Scott Laird) написали подключаемый модуль Action Cache, который рекомендуется использовать вместо встроенных в Rails средств кэширования. Он не изменяет API кэширования, но подставляет другую реализацию.

```
script/plugin install http://craz8.com/svn/trunk/plugins/action_cache
```

Этот модуль обладает следующими особенностями:

- записи в кэше хранятся в виде YAML-потоков (а не в виде HTML-разметки), поэтому в ответе вместе с кэшированным содержимым можно также возвращать заголовки;
- добавляет в ответ заголовок `last-modified`, поэтому клиенты могут посылать `GET`-запрос `If-modified`. Если у клиента уже есть копия кэшированного содержимого, он получит ответ `304 Not Modified`;
- гарантирует, что кэшируются только ответы с кодом `200 OK`. В противном случае в кэше могли бы застрять страницы с сообщениями об ошибках без полезного содержимого (что приводит к труднодиагностируемым проблемам);
- позволяет разработчику подменить используемую в Rails реализацию механизма, генерирующего ключи кэша;
- разрешает задавать в действии необязательный срок хранения записи в кэше, по истечении которого она будет автоматически удалена;
- позволяет управлять кэшированием действия во время выполнения в зависимости от параметров запроса (например, никогда не кэшировать содержимое для администраторов сайта).
- новый метод `expire_all_actions` очищает весь кэш действий;
- реализация метода `expire_action` изменена, так что из кэша удаляются все элементы, отвечающие регулярному выражению. Если вы следуете стилю REST и для одного и того же действия можете возвращать представления в форматах HTML, JS и XML, то при очистке любого из них методом `expire_action :controller => 'foo', :action => 'bar'` удаляются и все остальные.

Хранилища для кэша

В отличие от сеансовых данных, кэш фрагментов может занимать очень много места. Rails предоставляет четыре варианта организации хранилища для кэша:

- `FileStore` – фрагменты хранятся на диске в каталоге, определяемом параметром `cache_path`. Этот способ хорошо работает в любом режиме, и фрагменты доступны всем процессам веб-сервера, запущенным из одного и того же каталога приложения;
- `MemoryStore` – фрагменты хранятся в памяти и могут занимать недопустимо большой объем в памяти процесса;
- `DRbStore` – фрагменты хранятся в отдельном разделяемом процессе DRb. Только в этом случае имеется один кэш для всех процессов, но в ходе развертывания приходится иметь дело с дополнительным DRb-процессом;
- `MemCacheStore` – работает аналогично DRbStore, но использует проверенный сервер кэширования `memcached`. Я провел неформальный опрос

нескольких программистов, профессионально работающих с Rails, и все они согласились, что memcache – наилучший вариант[1].

Пример конфигурации

По умолчанию подразумевается режим :memory_store.

```
ActionController::Base.fragment_cache_store = :memory_store

ActionController::Base.fragment_cache_store = :file_store,
"/path/to/cache/directory"
ActionController::Base.fragment_cache_store = :drb_store,
"druby://localhost:9192"
ActionController::Base.fragment_cache_store = :mem_cache_store,
"localhost"
```

Ограничения на файловое хранилище

Если ваше приложение Rails размещено на одном сервере, настроить кэширование довольно просто (разумеется, кодирование – совсем другое дело).

Но когда приложение работает на кластере физических серверов, очистка кэша может стать трудновыполнимой задачей. Если не разместить файловое хранилище на разделяемой файловой системе, например NFS или GFS, ничего работать не будет.

Ручная очистка с помощью rake-задания

Если вы остановитесь на файловом хранилище, то, вероятно, захотите иметь способ ручной очистки всего кэша приложения. Это несложно сделать с помощью системы Rake. Просто добавьте файл cache.rake в папку lib/tasks. В нем должно содержаться примерно такое задание:

Листинг 10.17. Rake-задание cache_sweeper

```
desc "Ручная очистка кэша"
task :cache_sweeper do
  FileUtils.rm_rf Dir[File.join(RAILS_ROOT, "public", "entries*")]
#pages
  FileUtils.rm_rf Dir[File.join(RAILS_ROOT, "tmp", "cache*")]
#fragments
end
```

В этом примере я указал каталог entries, а вы не забудьте включить одно или несколько предложений FileUtils.rm_rf, соответствующих кэшируемым страницам *вашего* приложения.

---

[1] Если вы остановитесь на режиме memcache, обязательно познакомьтесь с подключаемым модулем CacheFu (автор Err the Blog), который можно скачать со страницы *http://require.errtheblog.com/plugins/browser/cache_fu*.

И последнее – часто подобный грубый подход с применением `FileUtils.rm_rf` предпочитают методам `expire_*`, потому что иногда проще стереть весь кэш и заново заполнять его по мере необходимости.

Заключение

В этой главе мы рассмотрели структуру `ActionView`, подробно остановившись на системе ERb-шаблонов и механизме работы рендеринга в Rails. Мы также уделили много внимания подшаблонам, поскольку они принципиально важны для эффективного программирования в Rails. От сравнительно простых принципов работы шаблонов мы перешли к более сложной теме – кэшированию. Зная, как реализовать кэширование, вы сможете сэкономить день работы над приложением, от которого требуется высокая производительность. Разработчики нагруженных сайтов склонны считать Rails весьма своеобразным генератором HTML, который помогает создавать кэшированное содержимое.

А теперь пришло время поговорить о механизме, с помощью которого можно наделить уровень представления различными хитроумными возможностями, не загромождая шаблоны. Я имею в виду помощников.

11

Все о помощниках

Благодарю за помощь помощнику
в оказании помощи беспомощному.
Ваша помощь очень... помогла!

Миссис Дуонг в фильме *Weekenders*

Мы уже встречались с несколькими методами-помощниками, которые Rails предоставляет для организации пользовательского интерфейса веб-приложения. В этой главе описаны все модули-помощники и содержащиеся в них методы, а также даны инструкции по созданию собственных помощников.

Библиотеки `PrototypeHelper` и `ScriptaculousHelper` вынесены из ядра Rails 2.0 и теперь распространяются как подключаемые модули. Они позволяют легко добавить в приложение Rails функциональность Ajax и подробно рассматриваются в главе 12 «Ajax on Rails».

Примечание

Эта глава представляет собой справочник. Хотя я приложил максимум усилий к тому, чтобы ее можно было читать подряд, обратите внимание, что модули-помощники `ActionView` расположены в алфавитном порядке, начиная с `ActiveRecordHelper` и заканчивая `UrlHelper`. В разделе о каждом модуле методы собраны в логические группы там, где это оправдано.

Модуль ActiveRecordHelper

Модуль `ActiveRecordHelper` содержит методы-помощники для быстрого создания форм из моделей `ActiveRecord`. Метод `form` может создать всю форму, сгенерировав поля для простых типов данных, содержащихся

в записи. Однако он не знает, как собирать компоненты пользовательского интерфейса для манипулирования ассоциациями. Как правило, разработчики составляют формы с нуля, пользуясь методами из модуля `FormHelper`, а не из этого модуля.

Отчет об ошибках контроля

Методы `error_message_on` и `error_messages_for` помогают добавить в шаблон единообразно отформатированную информацию об ошибках, обнаруженных валидаторами.

error_message_on(object, method, prepend_text = "", append_text = "", css_class = "formError")

Возвращает тег `DIV`, внутри которого находится сообщение об ошибке, присоединенное к указанному методу `method` объекта `object`, если таковое существует. Содержимое может быть специализировано с помощью параметров, содержащих текст, который предшествует сообщению, и следующий за ним, а также имя CSS-класса.

Этот метод обычно применяется, когда в пользовательском интерфейсе необходимы отдельные сообщения для некоторых полей формы (как в следующем примере, взятом из жизни):

```
<div class="form_field">
  <div class="field_label">
    <span class="required">*</span>
    <label>Имя</label>
   </div>
  <div class="textual">
    <%= form.text_field :first_name, :maxlength => 34, :tabindex => 1
%>
    <%= error_message_on :user, :first_name %>
  </div>
</div>
```

error_messages_for(*params)

Возвращает тег `DIV`, содержащий все сообщения об ошибках для всех объектов, которые хранятся в переменных экземпляра, переданных в качестве параметров. Этот метод применяется для обстраивания (scaffolding) в Rails, но редко встречается в реальных приложениях. В документации по Rails API рекомендуется использовать реализацию данного метода как образец для решения собственных задач:

> *Это готовый фрагмент для представления ошибок, в который встроены некоторые строки и HTML-разметка. Если вам нужно нечто, сильно отличающееся от готового представления, имеет смысл самостоятельно работать с объектом `object.errors`. Загляните в исходный код и убедитесь, насколько это просто.*

Мы последуем совету и воспроизведем здесь исходный код данного метода, но предупреждаем, что использовать его в качестве образца следует, *только* если вы хорошо владеете языком Ruby! Если же у вас есть время изучить реализацию, то вы, безусловно, узнаете много нового о реализации среды Rails, в котором весьма своеобразно преломляются возможности Ruby.

```
def error_messages_for(*params)
  options = params.last.is_a?(Hash) ? params.pop.symbolize_keys : {}
  objects = params.collect { |object_name|
              instance_variable_get("@#{object_name}")
            }.compact
  count = objects.inject(0) {|sum, object| sum + object.errors.count }
  unless count.zero?
    html = {}
    [:id, :class].each do |key|
      if options.include?(key)
        value = options[key]
        html[key] = value unless value.blank?
      else
        html[key] = 'errorExplanation'
      end
    end

    header_message = "#{pluralize(count, 'error')} prohibited this
    #{(options[:object_name] || params.first).to_s.gsub('_', ' ')}
    from being saved"

    error_messages = objects.map {|object|
      object.errors.full_messages.map {|msg| content_tag(:li, msg)}
    }

    content_tag(:div,
    content_tag(options[:header_tag] || :h2, header_message) <<
    content_tag(:p, 'There were problems with the following fields:') <<
    content_tag(:ul, error_messages), html )
  else
    ''
  end
end
```

Ниже в этой главе мы подробно поговорим о написании собственных методов-помощников.

Автоматическое создание формы

Следующие методы используются для автоматического создания полей в коде обстраивания. Вы тоже можете ими пользоваться, но подозреваю, что в реальных приложениях от них толку мало.

form(name, options)

Возвращает форму целиком с тегами `input` и всем прочим для поименованного объекта модели `ActiveRecord`. Ниже приведены примеры кода из документации по Rails API, в котором используется гипотетический объект `Post` из приложения для организации доски объявлений:

```
> form("post")

=> <form action='/post/create' method='post'>
    <p>
      <label for="post_title">Заголовок</label><br/>
      <input id="post_title" name="post[title]"
             size="30" type="text" value="Здравствуй, мир" />
    </p>
    <p>
      <label for="post_body">Тело</label><br/>
      <textarea cols="40" id="post_body" name="post[body]"
rows="20">
      И снова на штурм горы!
      </textarea>
   </p>
   <input type='submit' value='Создать' />
</form>
```

Внутри этот метод вызывает `record.new_record?`, чтобы понять, какое действие необходимо для формы: `create` или `update`. С помощью параметра `:action` можно задать действие и явно.

Если вам нужно, чтобы атрибут `enctype` формы был равен `multipart` (необходимо для загрузки файлов), задайте параметр `options[:multipart]` равным `true`.

Можно также передать параметр `:input_block`, воспользовавшись идиомой Ruby `Proc.new` для создания анонимного блока кода. Заданный блок будет вызван для каждой *контентной колонки* модели, а возвращенное им значение подставлено в форму.

```
> form("entry", :action => "sign",
    :input_block => Proc.new { |record, column|
      "#{column.human_name}: #{input(record, column.name)}<br/>" })

=> <form action='/post/sign' method='post'>
   Сообщение:
   <input id="post_title" name="post[title]" size="30"
          type="text" value="Здравствуй, мир" /><br />
  <input type='submit' value='Подписаться' />
</form>
```

Приведенный в этом примере *блок построителя* (builder block), как он называется в документации по Rails API, пользуется методом-помощ-

ником input, который также является частью модуля и подробно рассматривается в следующем разделе.

Наконец, можно включить в форму дополнительное содержимое, передав при вызове метода form блок, как показано ниже:

```
form("entry", :action => "sign") do |s|
  s << content_tag("b", "Отдел")
  s << collection_select("department", "id", @departments, "id",
"name")
end
```

Блоку передается строковый аккумулятор (в примере он назван s), куда дописывается произвольное содержимое, которое должно располагаться между основными полями ввода и тегом submit.

input(name, method, options)

Метод input с подходящим параметром name принимает идентифицирующую информацию и автоматически генерирует HTML-тег input, исходя из атрибута модели ActiveRecord. Возвращаясь к примеру объекта Post, приведем фрагмент, взятый из документации по Rails API:

```
input("post", "title")

=> <input id="post_title" name="post[title]" size="30"
          type="text" value="Здравствуй, мир" />
```

Для демонстрации типов полей ввода, генерируемых этим методом, я просто приведу кусок кода из самого модуля ActiveRecordHelper:

```
def to_tag(options = {})
  case column_type
    when :string
      field_type = @method_name.include?("password") ? "password" :
"text"
      to_input_field_tag(field_type, options)
    when :text
      to_text_area_tag(options)
    when :integer, :float, :decimal
      to_input_field_tag("text", options)
    when :date
      to_date_select_tag(options)
    when :datetime, :timestamp
      to_datetime_select_tag(options)
    when :time
      to_time_select_tag(options)
    when :boolean
      to_boolean_select_tag(options)
  end
end
```

Настройка выделения ошибочных полей

По умолчанию, чтобы выделить поле формы, не прошедшее проверку, Rails обертывает его в элемент DIV с именем класса fieldWithErrors. Это поведение можно настраивать, поскольку реализовано оно с помощью Proc-объекта, хранящегося как конфигурационное свойство класса ActionView::Base:

```
module ActionView
  class Base
    @@field_error_proc = Proc.new { |html_tag, instance|
      "<div class=\"fieldWithErrors\">#{html_tag}</div>"
    }
    cattr_accessor :field_error_proc
  end

  ...
```

Зная это, для изменения способа обработки ошибок в контроле достаточно переопределить атрибут field_error_proc в модуле ActionView, подставив собственный Proc-объект. Я рекомендую делать это либо в файле config/environment.rb, либо в классе контроллера вашего приложения ApplicationController.

В листинге 11.1 я изменил этот параметр, так что полям ввода, содержащим ошибки, предшествует красное слово ERR.

Листинг 11.1. Измененный способ индикации ошибочно введенных полей

```
ActionView::Base.field_error_proc =
  Proc.new do |html_tag,instance|
    %(<div style="color:red">ERR</div>) + html_tag
  end
```

Многие считают, что было бы гораздо лучше по умолчанию помечать CSS-классом fieldWithErrors сам тег input, а не обертывать его в дополнительный DIV. Действительно, это упростило бы нам жизнь, поскольку лишний DIV часто искажает выверенную с точностью до пикселя верстку. Однако, поскольку к моменту вызова field_error_proc строка html_tag уже построена, модифицировать ее содержимое – задача нетривиальная.

Существуют решения, которые с помощью регулярных выражений все же модифицируют строку html_tag, например такое, взятое со страницы *http://snippets.dzone.com/tag/field_error_proc*:

```
ActionView::Base.field_error_proc = Proc.new do |html_tag, instance|
  error_style = "background-color: #ffff80"
  if html_tag =~ /<(input|textarea|select)[^>]+style=/
    style_attribute = html_tag =~ /style=['"]/
    html_tag.insert(style_attribute + 7, "#{error_style}; ")
  elsif html_tag =~ /<(input|textarea|select)/
    first_whitespace = html_tag =~ /\s/
```

```
      html_tag[first_whitespace] = " style='#{error_style}' "
    end
    html_tag
  end
```

Но как уродливо! Безусловно, эта часть ActionView заслуживает усовершенствования.

Модуль AssetTagHelper

Согласно документации по Rails API, этот модуль:

> *Предоставляет методы для связывания HTML-страницы с другими ресурсами, в том числе с изображениями, JavaScript-сценариями, таблицами стилей и каналами. Можно сказать Rails, что такие ресурсы следует брать с выделенного сервера, задав параметр* `ActionController::Base.asset_host` *в файле environment.rb. Эти методы не проверяют факт существования связываемого ресурса.*

Модуль AssetTagHelper включает ряд методов, которыми вы часто будете пользоваться при разработке приложений Rails, и прежде всего image_tag.

Помощники для формирования заголовка

Большинство помощников из этого модуля служит для добавления информации в тег HEAD HTML-документа.

auto_discovery_link_tag(type = :rss, url_options = {}, tag_options = {})

Возвращает тег link, который броузеры и считыватели новостей могут использовать для автоматического распознавания канала в формате RSS или ATOM. Параметр type может принимать значения :rss (по умолчанию) или :atom. Дополнительные атрибуты тега link управляются параметром url_options, задаваемым в том же формате, что для url_for.

Модифицировать сам тег LINK позволяет параметр tag_options:

- :rel задает атрибут rel; по умолчанию "alternate";
- :type переопределяет тип MIME (например, "application/atom+xml"), который в противном случае Rails сгенерировал бы самостоятельно;
- :title задает заголовок ссылки; по умолчанию принимается значение type, записанное заглавными буквами.

Ниже приведены примеры использования метода auto_discovery_link_tag, взятые из документации по Rails API:

```
auto_discovery_link_tag # =>
  <link rel="alternate" type="application/rss+xml" title="RSS"
  href="http://www.curenthost.com/controller/action" />
```

```
auto_discovery_link_tag(:atom) # =>
  <link rel="alternate" type="application/atom+xml" title="ATOM"
  href="http://www.curenthost.com/controller/action" />

auto_discovery_link_tag(:rss, {:action => "feed"}) # =>
  <link rel="alternate" type="application/rss+xml" title="RSS"
  href="http://www.curenthost.com/controller/feed" />

auto_discovery_link_tag(:rss, {:action => "feed"}, {:title => "My
RSS"}) # =>
  <link rel="alternate" type="application/rss+xml" title="My RSS"
  href="http://www.curenthost.com/controller/feed" />
```

О предпочтительном значке

Поскольку разрешается задавать дополнительные параметры, теоретически можно было бы использовать метод `auto_discovery_link_tag` для генерации тега `LINK`, описывающего предпочтительный значок (небольшое изображение, которое показывается в адресной строке броузера и в закладках):

```
auto_discovery_link_tag('image/x-icon', 'favicon.ico',
  :rel => 'shortcut icon', :title => '')

<link rel="shortcut icon" href="favicon.ico"
type="image/x-icon" title="">
```

Но, честно говоря, не вижу причин использовать для этой цели метод `auto_discovery_link_tag`, поскольку гораздо проще написать HTML-код напрямую! В этом теге нет никакой динамичности, которая требовала бы привлечения помощников.

Я специально включил такой пример, чтобы преподать вам урок: не нужно пользоваться помощниками, когда требуемую разметку можно НАПИСАТЬ самостоятельно.

image_path(source)

Вычисляет путь к изображению в каталоге `public/images`. Полные пути от корня (начинающиеся со знака «/») передаются без изменения. Этот метод вызывается из метода `image_tag` для построения пути к изображению. Передача имени файла без расширения, практиковавшаяся в ранних версиях Rails, более не поддерживается.

```
image_path("edit.png") # => /images/edit.png
image_path("icons/edit.png") # => /images/icons/edit.png
image_path("/icons/edit.png") # => /icons/edit.png
```

image_tag(source, options = {})

Возвращает тег IMAGE, который можно подставлять в шаблон. Параметр source может содержать полный или относительный путь к файлу в каталоге public/images. С помощью параметра options можно включить в тег IMAGE произвольные дополнительные атрибуты.

Два следующих элемента в хеше options трактуются особым образом:

- :alt – если альтернативный текст не задан, используется базовое имя файла из параметра source без расширения и преобразованное в верхний регистр;
- :size – будучи представлен в виде *width* x *height*, например "30x45", преобразуется в пару атрибутов width="30" и height="45". Если формат некорректен, этот параметр молчаливо игнорируется.

```
image_tag("icon.png") # =>
  <img src="/images/icon.png" alt="Icon" />

image_tag("icon.png", :size => "16x10", :alt => "Edit Entry") # =>
  <img src="/images/icon.png" width="16" height="10" alt="Edit Entry" />

image_tag("/photos/dog.jpg", :class => 'icon') # =>
  <img src="/photos/icon.gif" alt="Dog" class="icon"/>
```

> **Говорит Кортенэ...**
>
> Метод image_tag пользуется внутренним методом image_path, определяющим путь, указываемый в теге. К сожалению, это означает, что нельзя присваивать имя image контроллеру, который должен работать как ресурс, поскольку возникнет конфликт имен.

javascript_include_tag(*sources)

Возвращает тег SCRIPT для каждого из переданных в параметре sources путей. Можно передавать имя JavaScript-файла (расширение .js допустимо опускать) из каталога public/javascripts или полный путь, начиная от корня сайта.

Для включения в свое приложение JavaScript-библиотек Prototype и Scriptaculous передайте в качестве параметра символ :defaults. Если при этом в каталоге public/javascripts существует файл application.js, он тоже будет включен. Чтобы модифицировать атрибуты тега SCRIPT, передайте в *последнем аргументе* хеш.

```
javascript_include_tag "xmlhr", :defer => 'defer' # =>
  <script type="text/javascript" src="/javascripts/xmlhr.js"
  defer="defer"></script>

javascript_include_tag "common.javascript", "/elsewhere/cools" # =>
  <script type="text/javascript"
src="/javascripts/common.javascript"></script>
  <script type="text/javascript" src="/elsewhere/cools.js"></script>

javascript_include_tag :defaults # =>
  <script type="text/javascript"
src="/javascripts/prototype.js"></script>
  <script type="text/javascript"
src="/javascripts/effects.js"></script>
  ...
  <script type="text/javascript"
src="/javascripts/application.js"></script>
```

javascript_path(source)

Вычисляет путь к JavaScript-ресурсу в каталоге `public/javascripts`. Если в имени файла нет расширения, дописывается `.js`. Полные пути от корня сайта передаются без изменения. Вызывается из метода `javascript_include_tag` для построения пути к сценарию.

stylesheet_link_tag(*sources)

Возвращает по одному тегу `LINK`, ссылающемуся на таблицу стилей, для каждого из переданных в параметре `sources` путей. Если не указано расширение, автоматически добавляется `.css`. Как и прочие помощники, этот метод принимает переменное число аргументов и хеш с дополнительными параметрами, который должен быть последним аргументом. Все перечисленные в этом хеше параметры преобразуются в атрибуты тега.

```
stylesheet_link_tag "style" # =>
  <link href="/stylesheets/style.css" media="screen"
        rel="Stylesheet" type="text/css" />

stylesheet_link_tag "style", :media => "all" # =>
  <link href="/stylesheets/style.css" media="all"
        rel="Stylesheet" type="text/css" />

stylesheet_link_tag "random.styles", "/css/stylish" # =>
  <link href="/stylesheets/random.styles" media="screen"
        rel="Stylesheet" type="text/css" />
  <link href="/css/stylish.css" media="screen"
        rel="Stylesheet" type="text/css" />
```

stylesheet_path(source)

Вычисляет путь к CSS-ресурсу в каталоге `public/stylesheets`. Если в имени файла нет расширения, дописывается `.css`. Полные пути от корня сайта передаются без изменения. Вызывается из метода `stylesheet_link_tag` для построения пути к таблице стилей.

Только для подключаемых модулей: добавление включаемых по умолчанию JavaScript-сценариев

Метод класса `ActionView::Helpers::AssetTagHelper` с именем `register_javascript_include_default` позволяет авторам подключаемых модулей включить один или несколько дополнительных JavaScript-файлов при обращении к `javascript_include_tag :defaults`. Этот метод предназначен только для вызова из процедуры инициализации подключаемого модуля с целью зарегистрировать дополнительные js-файлы, которые модуль установил в каталог `public/javascripts`. Подробнее об этом методе см. главу 19 «Расширение Rails с помощью подключаемых модулей».

Модуль BenchmarkHelper

Одна из наиболее часто упоминаемых инноваций в Rails – встроенные средства протоколирования. Модуль `BenchmarkHelper` добавляет возможность измерять время выполнения произвольных участков кода и полезен при поиске узких мест, снижающих быстродействие приложения.

benchmark(message = "Benchmarking", level = :info)

Измеряет время выполнения блока в шаблоне и записывает результат в протокол.

```
<% benchmark "Notes section" do %>
  <%= expensive_notes_operation %>
<% end %>
```

Этот код добавит в протокол сообщение вида Notes section (0.34523) при выполнении данного шаблона. В качестве необязательного второго аргумента можно передать уровень протоколирования (`:debug`, `:info`, `:warn`, `:error`); по умолчанию подразумевается `:info`.

Модуль CacheHelper

В этом модуле есть всего один метод `cache`. Он применяется для кэширования фрагментов шаблона без кэширования всего действия. В Rails

имеется также механизм кэширования страниц, реализованный с помощью метода caches_page контроллеров; в этом случае вся информация, сгенерированная действием, записывается в HTML-файл, который веб-сервер затем будет возвращать пользователям без участия Rails.

Напротив, кэширование фрагментов полезно, когда некоторые элементы разметки, формируемой действием, часто изменяются или зависят от сложно вычисляемого состояния, тогда как другие изменяются редко и могут показываться любому пользователю. Границы кэшируемого фрагмента определяются в шаблоне с помощью метода-помощника cache. Эта тема подробно обсуждалась в главе 2 «Работа с контроллерами».

Модуль CaptureHelper

Одна из самых замечательных особенностей представлений в Rails состоит в том, что вы можете не ограничиваться рендерингом в единственный «поток» содержимого. Попутно можно определять в шаблоне блоки, которые следует вставить в другие части страницы. Это реализуется с помощью двух методов из модуля CaptureHelper.

capture(&block)

Метод capture позволяет запомнить часть генерируемой по шаблону информации (заключенную внутри блока) и присвоить ее переменной экземпляра. Позже значение этой переменной можно будет использовать в другом месте шаблона.

```
<% @message_html = capture do %>
<div>Это сообщение</div>
<% end %>
```

Не думаю, что метод полезен в шаблоне сам по себе. Гораздо интереснее использовать его в собственных методах-помощниках. Он дает возможность писать помощники, которые захватывают содержимое шаблона, обернутое в блок. Мы рассмотрим эту технику подробнее в разделе «Написание помощников».

content_for(name, &block)

Мы уже упоминали метод content_for в разделе «Подстановка содержимого» главы 10. Он позволяет обозначить часть шаблона как содержимое для другой части страницы. Работает данный метод аналогично родственному методу capture (на самом деле, просто вызывает его). Вместо того чтобы возвращать содержимое переданного ему блока, он запоминает содержимое и позволяет позднее извлечь его с помощью yield в каком-то другом месте шаблона.

Типичный случай – вставка содержимого в боковую колонку макета. В примере ниже ссылка появляется не в основном «потоке» шаблона представления, а в месте, где встретится конструкция `<%= yield :navigation_sidebar %>`.

```
<% content_for :navigation_sidebar do %>
  <%= link_to 'Detail Page', item_detail_path(item) %>
<% end %>
```

Модуль DateHelper

Модуль `DateHelper` применяется для создания HTML-тегов `select`, содержащих календарные данные. Кроме того, в нем находится помощник с одним из самых длинных в Rails названий: `distance_of_time_in_words_to_now`.

Помощники для выбора даты и времени

Следующие методы помогают создавать в форме поля ввода, относящиеся к дате и времени. Все они рассчитаны на многопараметрическое присваивание объекту `ActiveRecord`. Это означает, что хотя в HTML-форме генерируется несколько полей ввода, при отправке на сервер программа понимает, что эти поля относятся к одному атрибуту модели. Магия Rails к вашим услугам!

date_select(object_name, method, options = {})

Возвращает набор тегов `select` (по одному для года, месяца и дня), уже настроенных с учетом заданного атрибута типа «дата» (определяется параметром `method`), принадлежащего назначенному шаблону объекта (определяется параметром `object_name`).

Теги `select` можно дополнительно настроить с помощью хеша `options`, который может содержать любые ключи, распознаваемые построителями отдельных тегов (например, `:use_month_numbers` для `select_month`).

Метод `date_select` понимает также задаваемые в хеше параметры `:discard_year`, `:discard_month` и `:discard_day`, которые позволяют исключить из генерируемого набора соответствующие теги. Исходя из здравого смысла, исключение месяца приводит также к исключению дня. Если же день исключается, а месяц – нет, Rails считает, что речь идет о первом дне месяца.

Можно также явно задать порядок тегов с помощью параметра `:order`, который содержит массив символов `:year`, `:month` и `:day` в желаемом порядке. Некоторые символы можно опускать, и тогда соответствующий тег `select` не будет включен.

Если задать в хеше параметр `:disabled => true`, то пользователь не сможет изменять элементы (листинг 11.2).

Листинг 11.2. Примеры использования метода date_select

```
date_select("post", "written_on")

date_select("post", "written_on", :start_year => 1995,
                                  :use_month_numbers => true,
                                  :discard_day => true,
                                  :include_blank => true)

date_select("post", "written_on", :order => [:day, :month, :year])

date_select("user", "birthday", :order => [:month, :day])
```

datetime_select(object_name, method, options = {})

Работает так же, как `date_select` с тем отличием, что добавляются еще теги `select` для ввода часа и минуты. Тег для ввода секунды можно добавить с помощью необязательного параметра `:include_seconds`. Вместе с возможностью вводить информацию о времени предоставляются и новые параметры для исключения ненужных тегов: `:discard_hour`, `:discard_minute` и `:discard_seconds`.

time_select(object_name, method, options = {})

Возвращает набор тегов `select` (по одному для часа, минуты и – по специальному запросу – секунды), уже настроенных с учетом заданного атрибута типа «время» (определяется параметром `method`), принадлежащего назначенному шаблону объекта (определяется параметром `object_name`). Чтобы включить тег для секунд, нужно задать параметр `:include_seconds`.

```
time_select("post", "sunrise")
time_select("post", "start_time", :include_seconds => true)
```

Помощники для задания отдельных элементов даты и времени

Иногда нужен только один элемент даты или времени, и Rails предоставляет в ваше распоряжение полный набор соответствующих помощников. В отличие от рассмотренных выше, следующие помощники не привязываются к переменной экземпляра на странице. Они просто принимают в качестве первого параметра объект Ruby, представляющий дату или время (у всех этих методов один и тот же набор необязательных параметров, который мы рассмотрим ниже)

select_date(date = Date.today, options = {})

Возвращает набор тегов `select` (по одному для года, месяца и дня), уже настроенных с учетом заданной (или текущей) даты. Можно явно за-

дать порядок тегов с помощью параметра :order, который может содержать массив символов :year, :month и :day в желаемом порядке.

select_datetime(datetime = Time.now, options = {})

Возвращает набор тегов select (по одному для года, месяца, дня, часа и минуты), уже настроенных с учетом заданных даты и времени. Дополнительно можно включить тег для секунд, указав параметр :include_seconds => true. Можно явно задать порядок тегов с помощью параметра :order, который содержит массив символов :year, :month, :day, :hour, :minute и :seconds в желаемом порядке. Кроме того, параметры :date_separator и :time_separator позволяют задать знаки-разделители элементов даты и времени соответственно (по умолчанию подразумеваются «/» и «:»).

select_day(date, options = {})

Возвращает тег select (с элементами для каждого дня – от 1 до 31), в котором уже выбран день, соответствующий текущей дате или дате, переданной в параметре date. По умолчанию тег называется day, но с помощью параметра :field_name можно указать и другое имя поля. Вместо даты в параметре date можно передать число от 1 до 31.

select_hour(datetime, options = {})

Возвращает тег select (с элементами для каждого часа – от 0 до 23), в котором уже выбран час, соответствующий текущему времени или времени, переданному в параметре datetime. В параметре datetime можно также передать число от 0 до 23.

select_minute(datetime, options = {})

Возвращает тег select (с элементами для каждой минуты – от 0 до 59), в котором уже выбрана минута, соответствующая текущему времени или времени, переданному в параметре datetime. Может также возвращать список с шагом minute_step (от 0 до 59), в котором выбрана минута 0. В параметре datetime также передаются числа от 0 до 59.

select_month(date, options = {})

Возвращает тег select (с элементами для каждого месяца – с января по декабрь), в котором уже выбран месяц, соответствующий текущей дате или дате, переданной в параметре date. По умолчанию в списке показываются названия месяцев, а в качестве значений, отправляемых серверу, фигурируют их номера (от 1 до 12).

Но можно и в визуальном представлении использовать номера, а не названия месяцев; для этого следует задать параметр :use_month_numbers

=> true. Если вам одновременно нужны номера и названия, задайте параметр :add_month_numbers => true. Если вы предпочитаете показывать сокращенные названия месяцев, укажите :use_short_month key => true. Наконец, если вы собираетесь использовать собственные названия месяцев, задайте к хеше ключ :use_month_names key, значением которого должен быть массив из 12 названий месяцев.

```
# В качестве ключей выступают "January", "March" и т. п.
select_month(Date.today)

# В качестве ключей выступают "1", "3" и т. п.
select_month(Date.today, :use_month_numbers => true)

# В качестве ключей выступают "1 - January", "3 - March" и т. п.
select_month(Date.today, :add_month_numbers => true)

# В качестве ключей выступают "Jan", "Mar" и т. п.
select_month(Date.today, :use_short_month => true)

# В качестве ключей выступают "Januar", "Marts" и т. п.
select_month(Date.today, :use_month_names => %w(Januar Februar
Marts ...))
```

Параметр :field_name позволяет переопределить имя поля. По умолчанию оно называется month.

select_second(datetime, options = {})

Возвращает тег select (с элементами для каждой секунды – от 0 до 59), в котором уже выбрана секунда, соответствующая текущему времени или времени, переданному в параметре datetime. В качестве параметра datetime можно передавать объект типа DateTime или числовое значение секунды.

select_time(datetime, options = {})

Возвращает набор тегов select (по одному для часа и минуты). Ключ :add_separator позволяет задать формат вывода.

select_year(date, options = {})

Возвращает тег select с элементами для каждого из пяти годов по обе стороны от текущего, который является выбранным. Начальный и конечный годы можно изменить с помощью параметров :start_year и :end_year. Если :start_year меньше :end_year, то года располагаются в списке в порядке убывания. Параметр date может быть объектом типа Date или числовым значением года.

```
# года в порядке возрастания
select_year(Date.today, :start_year => 1992, :end_year => 2007)
```

```
# года в порядке убывания
select_year(Date.today, :start_year => 2005, :end_year => 1900)
```

Параметры, общие для всех помощников, связанных с датами

Все методы, возвращающие теги `select`, принимают одни и те же перечисленные ниже дополнительные параметры в хеше `options`.

- `:discard_type`. Задавайте `true`, если не хотите включать в имя select часть, соответствующую типу. В этом случае метод `select_month` ограничится просто `date` (это можно переопределить с помощью параметра `:prefix`) вместо `date[month]`;
- `:field_name`. Позволяет переопределить естественное имя тега `select` (равное day, minute и т. д.);
- `:include_blank`. Задавайте `true`, если разрешается вводить пустую дату;
- `:prefix`. Переопределяет принимаемый по умолчанию префикс `date`, включаемый в имена тегов `select`. Если для метода `select_month` задать префикс `birthday`, то он сформирует имя `birthday[month]`, а не `date[month]`;
- `:use_hidden`. Задавайте `true`, если значение даты и время нужно оформить в виде скрытого поля, а не тега `select`.

Методы distance_in_time со сложными именами

Некоторые методы семейства `distance_in_time` имеют длинные и сложные описательные имена, которые никто не может запомнить. По крайней мере, пока не примелькаются.

Я считаю, что следующие методы служат идеальной иллюстрацией пути Rails в подходе к проектированию API. Вместо того чтобы остановиться на коротком и по необходимости более загадочном имени, автор платформы решил сделать имя длинным и описывающим назначение метода. Это один из тех случаев, когда даже не-программист способен читать код и понимать, что он делает (может быть).

Я также нахожу эти методы примечательными и потому, что они позволяют ответить на вопрос: почему Rails иногда считают частью феномена, именуемого Web 2.0. Какая еще среда включает способы *гуманистического* отображения временных штампов?

distance_of_time_in_words(from_time, to_time = 0, include_seconds = false)

Возвращает приблизительную разность в секундах между двумя моментами времени, представленными объектами типа `Time`, `Date` или целыми числами. Задайте параметр `include_seconds` равным `true`, если хо-

тите получить более точное приближение, когда разность меньше одной минуты.

Хотите посмотреть, как время преобразуется в словесные выражения? Вот полная реализация, которая, на мой взгляд, дает образчик хорошего программирования на Ruby:

```
def distance_of_time_in_words(from_time, to_time = 0, include_seconds =
false)

  from_time = from_time.to_time if from_time.respond_to?(:to_time)
  to_time = to_time.to_time if to_time.respond_to?(:to_time)

  d_minutes = (((to_time - from_time).abs)/60).round
  d_seconds = ((to_time - from_time).abs).round

  case d_minutes
    when 0..1
      unless include_seconds
        return (d_minutes==0) ? 'less than a minute' : '1 minute'
      end

    case d_seconds
      when 0..4   then 'less than 5 seconds'
      when 5..9   then 'less than 10 seconds'
      when 10..19 then 'less than 20 seconds'
      when 20..39 then 'half a minute'
      when 40..59 then 'less than a minute'
      else             '1 minute'
    end

    when 2..44      then "#{d_minutes} minutes"
    when 45..89     then 'about 1 hour'
    when 90..1439   then "about #{(d_minutes.to_f/60.0).round} hours"
    when 1440..2879      then '1 day'
    when 2880..43199     then "#{(d_minutes / 1440).round} days"
    when 43200..86399    then 'about 1 month'
    when 86400..525599   then "#{(d_minutes / 43200).round} months"
    when 525600..1051199 then 'about 1 year'
    else                      "over #{(d_minutes / 525600).round}
years"
  end
end
```

В документации по Rails API предлагается обратить внимание на то, что в Rails один год равен 365,25 дням.

distance_of_time_in_words_to_now(from_time, include_seconds = false)

Работает аналогично `distance_of_times_in_words`, только параметр `to_time` равен текущему времени. Обычно вызывается применительно

к атрибутам модели created_at или updated_at, и далее в шаблоне следует строка ago (назад), как в этом примере:

```
<strong><%= comment.user.name %></strong><br/>
<small><%= distance_of_time_in_words_to_now review.created_at %>
ago</small>
```

Модуль DebugHelper

В модуле DebugHelper есть всего один метод debug. При включении в шаблон передавайте ему объект, который нужно вывести в формате YAML и окружить тегами PRE. Полезно для отладки в режиме разработки, но и только.

Модуль FormHelper

Модуль FormHelper содержит ряд методов для работы с HTML-формами, особенно в части, касающейся объектов модели ActiveRecord, назначенных шаблону. В нем есть методы, соответствующие каждому виду полей ввода, определенному в HTML (text, password, select и т. д.). При отправке формы и ее получении значения полей ввода собираются в хеш params, который передается контроллеру.

Существуют две разновидности методов-помощников, относящихся к формам. Методы из этого модуля предназначены специально для работы с атрибутами модели, а аналогично названные методы из модуля FormTagHelper — нет.

Создание форм для моделей ActiveRecord

Основной метод в этом модуле называется form_for, в какой-то мере мы уже рассматривали его в главе 4 «REST, ресурсы и Rails». Вы передаете методу form_for имя модели, для которой хотите создать форму (или сам экземпляр модели), в качестве первого параметра, а вторым параметром должен быть URL, подставляемый в атрибут формы action. Помощник отдает управление блоку, передавая ему объект form, а для этого объекта в блоке вызываются методы формирования полей ввода уже без первого аргумента.

Предположим, что нам нужна форма, которую пользователь должен заполнить для создания новой записи типа Person, а вызов @person = Person.new встречается в действии контроллера, которое выполняет рендеринг такого шаблона:

```
<% form_for :person, @person, :url => { :action => "create" } do
|form| %>
  <%= form.text_field :first_name %>
  <%= form.text_field :last_name %>
```

```
  <%= submit_tag 'Создать' %>
<% end %>
```

Это эквивалентно следующему (устаревшему) варианту `form_for`, в котором объект `form` не передается с помощью `yield`, а имя объекта, на основе которого заполняются поля ввода, указывается явно:

```
<% form_for :person, @person, :url => { :action => "create" } do %>
  <%= text_field :person, :first_name %>
  <%= text_field :person, :last_name %>
  <%= submit_tag 'Создать' %>
<% end %>
```

В первой версии меньше повторов (вспомните о принципе DRY), но о более многословной версии совсем забывать не стоит – иногда она бывает необходима.

Переменные необязательны

Если вы явно указываете имя объекта для полей ввода, а не получаете его из объекта `form`, то имейте в виду, что оно не обязано соответствовать «живому» экземпляру объекта в области видимости шаблона. Rails не станет ругаться, если не найдет там указанный объект, а просто подставит в форму пустые поля.

Соглашения о формах, генерируемых Rails

HTML-разметка, генерируемая методом `form_for` в предыдущем примере, характерна для всех форм Rails и следует определенным соглашениям об именовании:

```
<form action="/persons/create" method="post">
  <input id="person_first_name" name="person[first_name]" size="30"
type="text" />
  <input id="person_last_name" name="person[last_name]" size="30"
type="text" />
  <input name="commit" type="submit" value="Создать" />
</form>
```

После отправки этой формы хеш `params` будет выглядеть следующим образом (привожу формат, в котором он представлен в протоколе режима разработки):

```
Parameters: {"commit"=>"Create", "action"=>"create",
"controller"=>"persons",
"person"=> {"first_name"=>"William", "last_name"=>"Smith"}}
```

Как видите, в хеше `params` имеется вложенный хеш `"person"`, доступный контроллеру как `params[:person]`. Это одна из основополагающих особенностей Rails, и я удивлюсь, если вы до сих пор не знали о ней. Обещаю, что в дальнейшем не буду уделять слишком много внимания тривиальным вещам.

Отображение существующих значений

Если вы редактируете существующий экземпляр класса `Person`, то поля формы заполнятся текущими значениями его атрибутов, поэтому результирующая HTML-разметка выглядит так:

```
<form action="/persons/create" method="post">
<input id="person_first_name" name="person[first_name]" size="30"
type="text" value="Obie"/>
<input id="person_last_name" name="person[last_name]" size="30"
type="text" value="Fernandez"/>
<input name="commit" type="submit" value="Создать" />
</form>
```

Ладно, это основы Rails. Но что если вы хотите заполнить поля формы для нового экземпляра объекта модели, записав в них некоторые значения? Надо ли передавать значения в хеше `options` помощникам, формирующим поля ввода? Нет. Поскольку помощники формы отображают значения атрибутов модели, достаточно просто инициализировать объект в контроллере:

```
def new
  @person = Person.new(:first_name => 'First', :last_name => 'Last')
end
```

Поскольку вы вызываете метод `new`, в базу данных ничего не записывается, а выбранные по умолчанию значения волшебным образом появляются в полях ввода.

Одновременное обновление нескольких объектов

Все это замечательно, если редактируется один объект. Но как быть, если нужно одновременно редактировать несколько записей? Когда имя атрибута, переданное методу `form_for` или одному из помощников для конкретных полей ввода, содержит пару квадратных скобок, в автоматически генерируемые атрибуты `name` и `id` тега `input` включается идентификатор объекта.

Мне эта техника кажется сомнительной на нескольких уровнях. Обычно мы идентифицируем имена атрибутов символами, но наличие квадратных скобок после символа (`:name[]`) недопустимо. Приходится вместо этого именовать объект с помощью строки:

```
<% form_for "person[]" do |form| %>
 <% for @person in @people %>
  <%= form.text_field :name %>
  ...
```

Кроме того, HTML-разметка, сгенерированная для тегов `input`, выглядит примерно так:

```
<input type="text" id="person_8_name" name="person[8][name]"
 value="Obie Fernandez"/>
```

Вот тебе и раз! Структура хеша, переданного контроллеру, существенно отличается от привычной. Теперь уровень вложенности в хеше `params` вырос до трех в области `"person"`, и (как будто этого мало) идентификаторы объектов используются в качестве ключей:

```
Parameters: {"person"=>{"8"=>{"name"=>"Obie Fernandez"},
"9"=>{"name"=>"Jodi Showers"}, ...}, ... }
```

Придется изменить код контроллера, обрабатывающего форму, если вы не хотите столкнуться с такой трассировкой стека:

```
NoMethodError (undefined method `8=' for #<User:0x8762174>):
/vendor/rails/activerecord/lib/active_record/base.rb:2032:in
`method_missing'
```

Хорошая новость заключается в том, что способ обработки вложенных хешей при таком методе обновления в вашем контроллере, пожалуй, можно назвать одним из самых элегантных примеров продуманной интеграции Rails на разных уровнях паттерна MVC:

```
Person.update(params[:person].keys, params[:person].values)
```

Красиво! Вот из-за подобных вещей программировать в Rails так приятно.

Квадратные скобки в новых записях?

Если вы вставляете HTML-разметку в документ динамически – с помощью JavaScript или AJAX, то можете обратить себе на пользу поведение Rails в части пустых квадратных скобок.

Когда используется соглашение об именах с квадратными скобками, Rails с удовольствием генерирует для новых объектов модели следующую разметку:

```
<input type="text" id="person__name" name="person[][name]"/>
```

Если вы захотите динамически добавлять в родительскую форму строки, соответствующие дочерним записям, то сможете легко воспроизвести это соглашение. Только не забудьте включить в имена полей ввода пару пустых квадратных скобок.

Получив отправленную форму, диспетчер Rails предполагает, что значением ключа `:person` в хеше `params` является объект `Array`, поэтому и вы, программируя действие контроллера, должны помнить, что `params[:person]` – массив!

Если же учесть, что для выполнения вставки нескольких записей методу класса `create` в моделях `ActiveRecord` передается массив хешей, то перед нами возникает еще один красивый пример межуровневой интеграции в Rails:

```
def add_people
  Person.create(params[:person])
  ...
end
```

Однако у этой техники есть и недостатки, поскольку для ее правильной работы необходимо, чтобы имена *всех* полей ввода в пространстве имен `person` содержали пару пустых квадратных скобок. Попробовав включить для того же объекта какое-нибудь поле без квадратных скобок, вы очень расстроите диспетчер Rails:

```
DISPATCHER FAILSAFE RESPONSE (has cgi) Sun Jul 15 14:36:35 -0400 2007
Status: 500 Internal Server Error
Conflicting types for parameter containers. Expected an instance of
Hash but found an instance of Array. This can be caused by colliding
Array and Hash parameters like qs[]=value&qs[key]=value.
```

Индексированные поля ввода

Пойдем дальше. Существует чуть более многословный и не такой магический способ определить несколько наборов полей ввода – воспользоваться параметром `:index`, который понимают методы самих полей ввода. Этот параметр позволяет явно задать идентификатор, вводимый в имена полей, что открывает ряд интересных возможностей.

Можно реплицировать прием с квадратными скобками, который мы обсуждали в предыдущем разделе. Пусть, например, имеются имена полей для набора физических лиц:

```
<% for @person in @people %>
<%= text_field :person, :name, :index => @person.id %>
...
```

Атрибут `id` отдельного человека будет вставлен в хеш `params` так же, как описывалось выше для имен с квадратными скобками, и мы получим точно такую же вложенность.

Чтобы перейти к более интересным вещам, заметим, что параметру `:index` безразличен тип «подсовываемого» ему идентификатора, что позволяет определить перечисляемые наборы записей. Именно в этом состоит отличие от техники квадратных скобок, и, думается мне, эта тема заслуживает более подробного объяснения.

Рассмотрим показанный в листинге 11.3 шаблон, относящийся к приложению для баскетбольного турнира (или к любому приложению, в котором четко определены роли людей).

Листинг 11.3. Форма ввода данных о баскетбольной команде

```
<% form_for :team do |f|%>
  <h2>Название команды</h2>
  Название: <%= f.text_field :name %><br/>
  Тренер: <%= f.text_field :coach %>
  <% ["guard_1", "guard_2", "forward_1", "forward_2", "center"].each
{|role| %>
    <h3><%= role.humanize %></h3>
    Имя: <%= text_field :players, :name, :index => role %>
  <% } %>
<% end %>
```

В ходе выполнения этот код порождает следующую HTML-разметку:

```
<form method="post" action="/homepage/team">
  <h2>Название команды</h2>
  Название: <input id="team_name" type="text" size="30"
name="team[name]"/><br/>
  Тренер: <input id="team_coach" type="text" size="30"
name="team[coach]"/>
  <h3>Guard 1</h3>
  Имя: <input id="players_guard_1_name" type="text" size="30"
  name="players[guard_1][name]"/>
  <h3>Guard 2</h3>
  Имя: <input id="players_guard_2_name" type="text" size="30"
  name="players[guard_2][name]"/>
  <h3>Forward 1</h3>
  Имя: <input id="players_forward_1_name" type="text" size="30"
  name="players[forward_1][name]"/>
  <h3>Forward 2</h3>
  Имя: <input id="players_forward_2_name" type="text" size="30"
  name="players[forward_2][name]"/>
  <h3>Center</h3>
  Имя: <input id="players_center_name" type="text" size="30"
  name="players[center][name]"/>
</form>
```

Если теперь отправить эту форму (что я и сделал, описав мою любимую команду), то действие контроллера получит следующий хеш с параметрами (я немного отформатировал взятую из протокола запись, чтобы четче проявилась структура):

```
Parameters: {"team"=>{
              "name"=>"Chicago Bulls",
              "coach"=>"Phil Jackson"},
            {"players"=> {
              "forward_1"=>{"name"=>"Scottie Pippen"},
              "forward_2"=>{"name"=>"Horace Grant"},
              "center"=>{"name"=>"Bill Cartwright"},
              "guard_1"=>{"name"=>"Michael Jordan"},
              "guard_2"=>{"name"=>"John Paxson"}}, ... }
```

Я специально включил в текстовые поля для ввода имени и возраста игроков идентификатор `:players`, а не стал привязывать их к объекту `team` формы. Вам даже не нужно заботиться об инициализации переменной `@players` – форма все равно будет работать. Помощники формы не ругаются, если переменная, для отражения которой они предназначены, равна `nil`, *если вы идентифицируете ее с помощью символа, а не передаете переменную экземпляра напрямую методу.*

Для полноты картины привожу в листинге 11.4 несколько упрощенный код действия контроллера, которое способно обработать отправленную форму.

Листинг 11.4. Действие контроллера, создающее описание команды

```
def create
  @team = Team.create(params[:team])
  params[:players].keys.each do |role|
    @team.add_player(role, Player.new(params[:players][role]))
  end
  ...
end
```

Принимая во внимание вложенность хеша параметров, мы можем разобрать его в цикле по ключам `params[:players].keys` и выполнить операции для каждой роли в отдельности. Разумеется, в этом коде предполагается, что у объекта `team` есть метод экземпляра `add_player(role, player)`, но думаю, что идею вы поняли.

Фиктивные акцессоры

Тут вы, наверное, думаете про себя: «Постой, постой...». Если бы модель `Team` знала, как отобразить хеш `players` на свои атрибуты, код контроллера был бы не в пример проще. Его вообще можно было бы свести всего к одной строке (не считая проверки ошибок и переадресации):

```
def create
  @team = Team.create(params[:team])
  ...
end
```

Впечатляет? (Меня точно!) Нам нужно лишь совершить мелкий подлог, воспользовавшись тем, что Джош Сассер называет фиктивными акцессорами[1], – то есть методами, позволяющими инициализировать части модели, не являющиеся атрибутами, которые соответствуют колонкам таблицы в базе данных. Нашей модели `Team` необходим метод изменения `players`, который знает, как добавить игроков в набор. Одна из возможных реализаций приведена в листинге 11.5.

Листинг 11.5. Добавление методов изменения, умеющих работать с хешем params

```
class Team < ActiveRecord::Base
  has_many :positions
  has_many :players, :through => :positions

  def players=(players_hash)
    players_hash.keys.each do |role|
      positions.create(:role => role,
                       :player => Player.new(players_hash[role]))
```

[1] Джош Сассер рассказывает, как задать значения по умолчанию для атрибутов модели, не соответствующих колонкам таблицы, в статье по адресу *http://blog.hasmanythrough.com/2007/1/22/using-faux-accessors-to-initialize-values*.

```
    end
  end
end

class Position < ActiveRecord::Base
  belongs_to :player
  belongs_to :team
end

class Player < ActiveRecord::Base
  has_many :positions
  has_many :teams, :through => :positions
end
```

Итак, метод `players=` вызывается при обращении к методу `Team.create`, и ему передается заполненный хеш `params`, который включает вложенный хеш `:players`. Предупреждаю, что эта техника годится не во всех случаях. В рассмотренном примере, где имеется отношение `has_many :through`, связывающее классы `Team`, `Position` и `Player`, она идеально подходит, но модель вашей предметной области может быть устроена иначе. Важна самая идея – стремиться по возможности писать столь же *чистый* код. Это и есть путь Rails.

> **Говорит Кортенэ...**
>
> Подобное сокрытие кода внутри метода делает код и проще, и мощнее. Теперь можно тестировать данный метод автономно, а для тестирования контроллера подставить вместо него заглушку. В этом случае заглушка позволяет сосредоточиться на тестировании логики действия контроллера, а не поведения базы данных.
>
> Кроме того, вы сами или ваш коллега сможете изменить реализацию, не нарушая работу других тестов, а код для работы с базой данных останется там, где ему и место, – в модели.

Я немного отклонился от темы. Мы ведь говорили о помощниках формы, поэтому рассмотрим еще один важный вопрос, связанный с ними, а потом двинемся дальше.

Как помощники формы получают свои значения

Для понимания работы помощников форм в Rails важно иметь в виду, что отображаемые ими значения берутся непосредственно из базы данных еще до того, как к ним приложит руку разработчик. Если вы не знаете, что делаете, то можете получить неожиданные результаты, пытаясь переопределить значения, отображаемые в форме.

Проиллюстрируем это на модели `LineItem`, в которой есть атрибут `rate` типа `decimal` (соответствующий колонке `rate` в таблице базы данных). Мы подменим его *неявный* акцессор своим:

```
class LineItem < ActiveRecord::Base
  def rate
    "A RATE"
  end
end
```

В обычной ситуации переопределенный акцессор скрывает доступ к настоящему атрибуту `rate`, в чем легко убедиться с помощью консоли:

```
>> li = LineItem.new
=> #<LineItem:0x34b5d18>
>> li.rate
=> "A RATE"
```

Но предположим, что вы собираете форму для редактирования объектов `LineItem` с помощью помощников:

```
<%- form_for :line_item do |f| -%>
  <%= f.text_field :rate %>
<%- end -%>
```

Выясняется, что она работает, как будто никакого переопределения акцессора `rate` не было. Это сделано специально, но настолько странно выглядит, что было подано несколько извещений об ошибке[1].

Суть дела в том, что помощники формы в Rails пользуются специальными методами с именами вида *attribute*`_before_type_cast` (они рассматривались в главе 6). В предыдущем примере вызывается метод `rate_before_type_cast` в обход переопределенного нами метода.

Модуль FormOptionsHelper

Методы из модуля `FormOptionsHelper` помогают работать с HTML-тегами `select`, предоставляя средства для преобразования наборов объектов в последовательность тегов `option`.

Помощники select

Следующие методы помогают создавать теги `select` с известными идентификаторами объекта (`object`) и атрибута (`attribute`).

[1] Читайте сагу о том, почему помощники формы не пользуются акцессорами объектов, по адресу *http://dev.rubyonrails.org/ticket/2322*.

collection_select(object, attribute, collection, value_method, text_method, options = {}, html_options = {})

Возвращает теги `select` и `option` для заданной пары `object` и `attribute`, обращаясь к методу `options_from_collection_for_select` (также в этом модуле) для создания списка тегов `option` исходя из параметра `collection`.

country_select(object, attribute, priority_countries = nil, options = {}, html_options = {})

Возвращает теги `select` и `option` для заданного объекта и метода, обращаясь к методу `country_options_for_select` при создании списка тегов `option`. Примечание: названия стран, вставляемые Rails, не содержат стандартного двузначного кода страны.

Аргумент `priority_countries` позволяет задать массив названий стран, которые для удобства пользователя должны показываться в начале раскрывающегося списка:

```
<%= form.country_select :billing_country, ["United States"] %>
```

select(object, attribute, choices, options = {}, html_options = {})

Создает тег `select` и последовательность вложенных в него тегов `option` для заданного объекта и атрибута. Если объект не равен `nil` и указанный атрибут в нем имеет какое-то значение, то это значение окажется выбранным в списке (о формате параметра `choices` см. описание метода `options_for_select`).

В примере ниже значение атрибута `@person.person_id` равно 1:

```
select("post", "person_id", Person.find(:all).collect {|p|
[ p.name, p.id ] }, { :include_blank => true })
```

В результате выполнения этого кода будет сгенерирована следующая HTML-разметка:

```
<select name="post[person_id]">
<option value=""></option>
<option value="1" selected="selected">David</option>
<option value="2">Sam</option>
<option value="3">Tobias</option>
</select>
```

Параметр `:selected => value` позволяет явно задать выбранный элемент, а `:selected => nil` — не выбирать никакой элемент. Параметр `:include_blank => true` вставляет пустой тег `option` в начало списка, чтобы заранее выбранных значений не было.

time_zone_select(object, method, priority_zones = nil, options = {}, html_options = {})

Возвращает теги `select` и `option` для заданного объекта и метода, обращаясь к методу `time_zone_options_for_select` для генерации списка тегов `option`.

Помимо параметра `:include_blank`, описанного в предыдущем разделе, этот метод поддерживает также параметр `:model`, по умолчанию равный `TimeZone`. С его помощью пользователь может задать для объекта модели другой часовой пояс (дополнительную информацию см. в разделе о методе `time_zone_options_for_select`).

Другие помощники

Все описываемые далее методы возвращают только список тегов `option`, поэтому вызывать их нужно из помощника, возвращающего тег `select`, или каким-то иным способом обернуть в тег `select`.

country_options_for_select(selected = nil, priority_countries = nil)

Возвращает строку, содержащую теги `option` практически для всех стран. Если хотите, чтобы один из тегов `option` был помечен атрибутом `selected`, укажите название страны в параметре `selected`. Можете также передать в параметре `priority_countries` массив стран, которые должны оказаться в начале (длинного) списка.

option_groups_from_collection_for_select(collection, group_method, group_label_method, option_key_method, option_value_method, selected_key = nil)

Возвращает строку, содержащую теги `option`, подобно `options_from_collection_for_select`, но дополнительно окружает их тегами `OPTGROUP`. Объект `collection` должен возвращать подмассив элементов, когда вызывается его метод `group_method`. Каждая группа в `collection` должна возвращать собственное имя при вызове метода `group_label_method`. Параметры `option_key_method` и `option_value_method` служат для вычисления атрибутов тега `option`.

Пожалуй, все это гораздо проще показать на примере, чем объяснить словами:

```
>> html_option_groups_from_collection(@continents, "countries",
"continent_name", "country_id", "country_name",
@selected_country.id)
```

Данный код мог бы вывести примерно такую HTML-разметку:

```
<optgroup label="Africa">
 <select>Egypt</select>
 <select>Rwanda</select>
 ...
</optgroup>
<optgroup label="Asia">
 <select>China</select>
 <select>India</select>
 <select>Japan</select>
...
</optgroup>
```

Для ясности приведем также соответствующие классы модели:

```
class Continent
  def initialize(p_name, p_countries)
    @continent_name = p_name; @countries = p_countries
  end

  def continent_name
    @continent_name
  end

  def countries
    @countries
  end
end

class Country
  def initialize(id, name)
    @id, @name = id, name
  end

  def country_id
    @id
  end

  def country_name
    @name
  end
end
```

options_for_select(container, selected = nil)

Принимает контейнер (хеш, массив или иной перечисляемый объект) и возвращает строку тегов `option`.

Если элементы контейнера отвечают на сообщения `first` и `last` (например, каждый элемент является массивом из двух элементов), то «первый» принимается в качестве значения тега `option`, а «последний» — в качестве текста. Не слишком сложно написать выражение, которое конструирует двухэлементный массив с помощью итераторов `map` и `collect`.

Пусть, например, имеется набор предприятий, и вы хотите, чтобы пользователь мог отфильтровать этот набор по категории бизнеса. Категория – это не просто строка; в примере ниже она представлена полноценной моделью, связанной с предприятием ассоциацией `belongs_to`:

```
class Business < ActiveRecord::Base
  belongs_to :category
end

class Category < ActiveRecord::Base
  has_many :businesses

  def <=>(other)
end
```

В упрощенном виде шаблон для отображения такого набора предприятий мог бы выглядеть следующим образом:

```
<% opts = @businesses.map(&:category).sort.collect {|c| [[c.name],
[c.id]]} %>
<% select_tag(:filter, options_for_select(opts, params[:filter])) %>
```

В первой строке формируется контейнер, удовлетворяющий требованиям метода `options_for_select`, – сначала производится агрегирование по атрибуту `category` набора `businesses` с помощью метода `map` и поддерживаемой Ruby синтаксической конструкции `&:method`. Во второй строке из созданных опций генерируется тег `select` (см. ниже в этой главе). На практике следовало бы еще немного подправить список категорий, чтобы он был отсортирован в нужном порядке и не содержал дубликатов:

```
... @businesses.map(&:category).uniq.sort.collect {...
```

Для небольших наборов данных такие манипуляции вполне можно производить на уровне кода, написанного на Ruby. Поскольку вряд ли кто-то станет «запихивать» в тег `select` сотни, а то и тысячи строк, эта техника весьма полезна. Не забудьте реализовать в свой модели метод `<=>`, если хотите, чтобы ее можно было сортировать методом `sort`.

Кроме того, в подобных случаях имеет смысл поэкспериментировать с попутной загрузкой, чтобы не выполнять по одному запросу к базе для каждого объекта, представленного в теге `select`. В нашем примере контроллер мог бы заполнить набор предприятий с помощью такого кода:

```
@businesses = Business.find(:conditions => ..., :include => :category)
```

Хеши автоматически преобразуются в форму, приемлемую для метода `options_for_select`, – ключи становятся «первыми» элементами, а значения – «последними».

Если задан параметр `selected` (содержащий одно значение или массив при необходимости выбрать несколько элементов), то выбранными становятся элементы, для которых «последний» соответствует значению (или значениям) этого параметра:

```
>> options_for_select([["Dollar", "$"], ["Kroner", "DKK"]])
   <option value="$">Dollar</option>
   <option value="DKK">Kroner</option>

>> options_for_select([ "VISA", "MasterCard" ], "MasterCard")
   <option>VISA</option>
   <option selected="selected">MasterCard</option>

>> options_for_select({ "Basic" => "$20", "Plus" => "$40" }, "$40")
   <option value="$20">Basic</option>
   <option value="$40" selected="selected">Plus</option>

>> options_for_select([ "VISA", "MasterCard", "Discover" ],
                      ["VISA", "Discover"])
   <option selected="selected">VISA</option>
   <option>MasterCard</option>
   <option selected="selected">Discover</option>
```

Если данный метод неправильно отображает выбранный элемент, убедитесь, что значение параметра `selected` соответствует типу объектов, хранящихся в наборе `collection`, иначе программа работать не будет. В следующем примере, если `price` – числовое значение, не отвечающее на метод `to_s`, будет выбран неправильный элемент, так как опции переданы в виде строк:

```
>> options_for_select({ "Basic" => "20", "Plus" => "40" }, price.to_s)
   <option value="20">Basic</option>
   <option value="40" selected="selected">Plus</option>
```

options_from_collection_for_select(collection, value_method, text_method, selected=nil)

Возвращает строку, содержащую теги `option`, построенную обходом всего набора `collection` и присваиванием результата вызова `value_method` атрибуту `value`, а результата вызова `text_method` – тексту тега `option`. Если задан параметр `selected`, то элемент, для которого `value_method` возвращает совпадающее значение, оказывается выбранным.

time_zone_options_for_select(selected = nil, priority_zones = nil, model = TimeZone)

Возвращает строку, содержащую теги `option` практически для всех часовых поясов. Укажите в качестве `TimeZone` название пояса, который должен быть выбран. Можно также передать в параметре `priority_zones` массив объектов `TimeZone`, которые должны оказаться в начале (длинного) списка. Вспомогательный метод `TimeZone.us_zones` позволяет получить список часовых поясов США.

Параметр `selected` должен быть либо равен `nil`, либо содержать строку с именем объекта `TimeZone` (см. приложение А «Справочник по Active-Support API»).

По умолчанию model – константа TimeZone (ее можно получить от ActiveRecord в виде объекта-значения). Единственное требование, предъявляемое к параметру model, заключается в том, что он должен отвечать на метод all и возвращать при этом массив объектов, представляющих часовые пояса.

Модуль FormTagHelper

Перечисленные ниже методы генерируют HTML-форму и теги input, исходя из явно заданных имен и значений, и этим отличаются от аналогичных методов из модуля FormHelper, для которых необходима ассоциация с экземпляром модели ActiveRecord. Все эти методы-помощники принимают параметр options, содержащий специальные опции или просто значения дополнительных атрибутов, которые нужно добавить в генерируемый HTML-тег.

check_box_tag(name, value = "1", checked = false, options = {})

Создает HTML-разметку для флажка (тега input типа checkbox). В отличие от более заковыристого метода check_box из модуля FormHelper, не создает дополнительного скрытого поля, гарантирующего возврат значения false, даже когда флажок не отмечен:

```
>> check_box_tag('remember_me')
=> <input id="remember_me" name="remember_me" type="checkbox"
value="1"/>

>> check_box_tag('remember_me', 1, true)
=> <input checked="checked" id="remember_me" name="remember_me"
   type="checkbox" value="1" />
```

end_form_tag

До версии Rails 2.0 метод end_form_tag выводил HTML-строку </form> и применялся в сочетании с методом start_form_tag. Но теперь мы пользуемся блоком для обертывания содержимого формы, поэтому данный метод больше не нужен.

file_field_tag(name, options = {})

Создает поле для загрузки файла. Напомню, что для HTML-формы необходимо указать тип кодирования *multipart*, иначе загрузка файла работать не будет:

```
<%= form_tag { :action => "post" }, { :multipart => true } %>
  <label for="file">Загружаемый файл</label>
  <%= file_field_tag :uploaded_data %>
  <%= submit_tag %>
<%= end_form_tag %>
```

Действие контроллера получит объект File, указывающий на загруженный файл, который в данный момент существует в виде временного файла в вашей системе. Вопрос об обработке загруженного файла в этой книге не рассматривается. Умный человек не станет сочинять собственный обработчик, а воспользуется великолепным подключаемым модулем AttachmentFu[1] Рика Олсона.

form_tag(url_for_options = {}, options = {}, *parameters_for_url, &block)

Открывает тег FORM, записывая в атрибут action URL, переданный в параметре url_for_options. У этого метода есть синоним start_form_tag.

Параметр :method по умолчанию равен POST. HTTP-методы GET и POST броузеры умеют обрабатывать самостоятельно; если же вы укажете глагол «put», «delete» или иной, добавится скрытое поле с именем _method и значением, соответствующим указанному в параметре :method. Диспетчер Rails понимает параметр _method, который лежит в основе REST-совместимых механизмов, обсуждавшихся в главе 4.

Параметр :multipart позволяет указать, что в форме есть поля для загрузки файлов, на которые сервер должен соответственно реагировать:

```
>> form_tag('/posts')
=> <form action="/posts" method="post">

>> form_tag('/posts/1', :method => :put)
=> <form action="/posts/1" method="put">

>> form_tag('/upload', :multipart => true)
=> <form action="/upload" method="post" enctype="multipart/form-data">
```

Следует отметить, что все параметры метода form_tag необязательны. Если ничего не задавать, то форма будет отправлена на тот же URL, с которого поступила, это решение на скорую руку, которое я часто применяю при разработке прототипов или во время экспериментов. Чтобы организовать действие контроллера, способное обрабатывать такие возвраты, просто включите предложение if/else, в котором проверяется метод запроса:

```
def add
  if request.post?
    # обработать параметры, отправленные методом POST
    redirect_to :back
  end
end
```

Обратите внимание, что если поступил POST-запрос, я обрабатываю хеш params, а затем переадресую клиента на исходный URL (с помощью

[1] Модуль Attachment Fu можно найти по адресу *http://svn.techno-weenie.net/projects/plugins/attachment_fu*.

redirect_to :back). В противном случае обработка продолжится и будет выполнен рендеринг шаблона, ассоциированного с действием.

hidden_field_tag(name, value = nil, options = {})

Создает скрытое поле; параметры такие же, как у метода text_field_tag.

image_submit_tag(source, options = {})

Выводит изображение, при щелчке по которому форма будет отправлена. Интерфейс у этого метода такой же, как у аналогичного ему метода image_tag из модуля AssetTagHelper.

Картинки представляют собой популярную замену стандартным тегам submit, поскольку разнообразят внешний вид приложения. Кроме того, они позволяют определить местоположение курсора мыши – хеш params включает параметры x и y.

password_field_tag(name = "password", value = nil, options = {})

Создает поле для ввода пароля. Во всех остальных отношениях этот метод ничем не отличается от text_field_tag.

radio_button_tag(name, value, checked = false, options = {})

Создает переключатель. Не забудьте, что все элементы переключателя должны иметь одно и то же имя name, иначе броузер не будет считать, что они принадлежат одному переключателю.

select_tag(name, option_tags = nil, options = {})

Создает раскрывающийся список или (если параметр :multiple равен true) список, допускающий выбор нескольких элементов. Параметр option_tags представляет собой строку тегов option, подставляемую внутрь тега select. Необязательно строить эту строку самостоятельно – можно воспользоваться помощниками из модуля FormOptions (рассмотрен в предыдущем разделе), которые умеют генерировать часто употребляемые списки стран, часовых поясов или ассоциированных записей.

start_form_tag

Синоним метода form_tag.

submit_tag(value = "Save changes", options = {})

Создает кнопку типа submit с надписью, заданной в параметре value. С помощью параметра :disable_with в хеше options можно задать надпись, появляющуюся, когда кнопка неактивна.

text_area_tag(name, content = nil, options = {})

Создает многострочное поле для ввода текста (тег TEXTAREA). Параметр :size позволяет указывать размеры текстовой области, не прибегая к явному заданию параметров :rows и :cols.

```
>> <%= text_area_tag "body", nil, :size => "25x10" %>
=> <textarea name="body" id="body" cols="25" rows="10"></textarea>
```

text_field_tag(name, value = nil, options = {})

Создает стандартное поле для ввода текста.

> **Говорит Кортенэ...**
>
> Многие из перечисленных выше функций будут удобны начинающему программисту Rails, но настоящие асы пишут HTML-теги и JavaScript-код вручную.

Модуль JavaScriptHelper

Предоставляет помощников для включения в шаблоны JavaScript-кода.

button_to_function(name, function, html_options={}, &block)

Возвращает тег button; при щелчке по соответствующей кнопке вызывается JavaScript-функция с помощью обработчика события onclick. Аргумент function можно опускать, если задан блок update_page, содержащий код в стиле RJS. В хеше options задаются дополнительные атрибуты тега button:

```
button_to_function "Привет", "alert('Hello world!')"

button_to_function "Удалить", "if (confirm('Вы уверены?')) do_delete()"

button_to_function "Подробнее" do |page|
  page[:details].visual_effect :toggle_slide
end

button_to_function "Подробнее", :class => "details_button" do |page|
  page[:details].visual_effect :toggle_slide
end
```

define_javascript_functions()

Включает исходный код всех используемых в проекте JavaScript-библиотек, помещая его в единственный тег SCRIPT. Первым включается файл prototype.js, затем – его расширения, входящие в состав ядра (они

находятся в файлах с именами, начинающимися со слова prototype). После этого в произвольном порядке включаются все файлы из каталога `public/javascripts`.

Предпочтительнее пользоваться помощником `javascript_include_tag` из модуля `AssetTagHelper`, который создает теги `SCRIPT` со ссылками на удаленные ресурсы.

escape_javascript(javascript)

Экранирует переходы на новую строку, одиночные и двойные кавычки во фрагментах JavaScript-кода.

javascript_tag(content, html_options={})

Выводит тег `SCRIPT`, внутрь которого помещено значение параметра `content`. Параметры, заданные в хеше `html_options`, трансформируются в дополнительные атрибуты тега.

link_to_function(name, function, html_options={}, &block)

Возвращает ссылку, при щелчке по которой будет вызвана JavaScript-функция, обернутая в обработчик события `onclick`. В конце добавляется предложение `return false;`, чтобы броузер завершил на этом обработку щелчка по ссылке:

```
>> link_to_function "Привет", "alert('Hello world!')"
=> <a onclick="alert('Hello world!'); return false;"
href="#">Привет</a>

>> link_to_function(image_tag("delete"), "if (confirm('Вы уверены?'))
do_delete()")

>> <a onclick="if (confirm('Вы уверены?')) do_delete(); return false;"
href="#">
   <img src="/images/delete.png?" alt="Delete"/>
</a>
```

Как и в случае `button_to_function`, можно опускать параметр `function`, если задан блок с JavaScript-кодом в стиле RJS:

```
>> link_to_function("Показать подробности", nil, :id => "more_link") do |page|
     page[:details].visual_effect :toggle_blind
     page[:more_link].replace_html "Убрать подробности"
   end

>> <a href="#" id="more_link" onclick="try {
        $('details').visualEffect('toggle_blind');
        $('more_link').update('Убрать подробности');
}...
```

Модуль NumberHelper

Этот модуль содержит методы, которые помогают преобразовывать числовые данные в отформатированные строки, отвечающие потребностям шаблона. Имеются методы для форматирования номеров телефонов, денежных сумм, процентов и размеров файлов, представления с заданной точностью и позиционирования.

human_size(size, precision=1)

Синоним `number_to_human_size`.

number_to_currency(number, options = {})

Преобразует число в денежную сумму. Параметры формата можно задать в хеше `options`:

- `:precision` задает количество цифр после десятичной точки, по умолчанию 2;
- `:unit` задает символ валюты, по умолчанию `"$"`;
- `:separator` задает разделитель между целой и дробной частью, по умолчанию `"."`;
- `:delimiter` задает разделитель между группами из трех цифр, по умолчанию `","`.

```
>> number_to_currency(1234567890.50)
=> $1,234,567,890.50

>> number_to_currency(1234567890.506)
=> $1,234,567,890.51

>> number_to_currency(1234567890.506, :precision => 3)
=> $1,234,567,890.506

>> number_to_currency(1234567890.50, :unit => "&pound;",
=> :separator => ",", :delimiter => ""
=> &pound;1234567890,50
```

number_to_human_size(size, precision=1)

В предположении, что `size` – это размер в байтах, возвращает наиболее понятное его представление. Полезно, когда нужно сообщить размер файла пользователю. Если `size` невозможно преобразовать в число, возвращает `nil`. По умолчанию принимается один знак после запятой, но параметр `precision` позволяет изменить это соглашение:

```
number_to_human_size(123) => 123 Bytes
number_to_human_size(1234) => 1.2 KB
```

```
number_to_human_size(12345) => 12.1 KB
number_to_human_size(1234567) => 1.2 MB
number_to_human_size(1234567890) => 1.1 GB
number_to_human_size(1234567890123) => 1.1 TB
number_to_human_size(1234567, 2) => 1.18 MB
```

У данного метода есть синоним `human_size`.

number_to_percentage(number, options = {})

Форматирует число в виде строки со знаком процента. Формат настраивается с помощью параметров в хеше `options`:

- `:precision` – задает количество знаков после запятой, по умолчанию 3;
- `:separator` – задает разделитель между целой и дробной частью, по умолчанию ".".

```
number_to_percentage(100) => 100.000%
number_to_percentage(100, {:precision => 0}) => 100%
number_to_percentage(302.0574, {:precision => 2}) => 302.06%
```

number_to_phone(number, options = {})

Представляет число в формате номера телефона, принятого в США. С помощью параметров в хеше `options` формат можно настроить:

- `:area_code` – заключает код региона в скобки;
- `:delimiter` – задает разделитель, по умолчанию "-";
- `:extension` – добавляет в конец сгенерированной строки указанный добавочный номер;
- `:country_code` – задает код страны.

```
number_to_phone(1235551234) => 123-555-1234
number_to_phone(1235551234, :area_code => true) => (123) 555-1234
number_to_phone(1235551234, :delimiter => " ") => 123 555 1234
```

number_with_delimiter(number, delimiter=",", separator=".")

Выделяет в числе группы по три цифры, вставляя между ними заданный разделитель `delimiter`. Формат можно настроить, задавая разделители между группами цифр, а также между целой и дробной частью (параметр `separator`):

- `delimiter` – задает разделитель между группами из трех цифр, по умолчанию ",";
- `separator` – задает разделитель между целой и дробной частью, по умолчанию ".".

```
number_with_delimiter(12345678) => 12,345,678
number_with_delimiter(12345678.05) => 12,345,678.05
number_with_delimiter(12345678, ".") => 12.345.678
```

number_with_precision(number, precision=3)

Форматирует число с заданным количеством знаков после запятой. По умолчанию 3.

```
number_with_precision(111.2345) => 111.235
number_with_precision(111.2345, 2) => 111.24
```

Модуль PaginationHelper

Из версии Rails 2.0 модуль `PaginationHelper` исключен. Работал он плохо, и все его терпеть не могли. Никто не помнит, чтобы когда-нибудь пользовался этим модулем для разбиения на страницы, – все применяли для этой цели что-то другое.

К счастью, изобретать собственный механизм разбиения вам не придется, поскольку есть парочка готовых подключаемых модулей, содержащих все необходимое. Мы не будем здесь подробно их обсуждать, но я по крайней мере укажу вам правильное направление.

> **Говорит Кортенэ...**
>
> Предлагаемый по умолчанию разбиватель на страницы был неплох, но совершенно не масштабировался.

will_paginate

Этот подключаемый модуль[1], написанный авторами популярного блога ERR the blog, применяют практически все опытные разработчики для Rails. Установите его командой:

```
$ script/plugin install svn://errtheblog.com/svn/plugins/will_pagin
```

Если коротко, то принцип работы `will_paginate` заключается в добавлении к модели `ActiveRecord` метода `paginate`, который можно использовать вместо `find`. Помимо обычных для `find` соглашений об именовании, аргументов и необязательных параметров, метод `paginate` принимает еще и параметр `:page` (что вполне естественно).

В контроллере запрос может выглядеть так:

```
@posts = Post.paginate_by_board_id @board.id, :page => params[:page]
```

Код шаблона представления почти не изменяется, просто он получает для вывода меньше записей. Для отображения самого элемента управ-

[1] Модуль `will_paginate` находится по адресу *http://require.errtheblog.com/plugins/browser/will_paginate*, но ведь вы это знаете, поскольку все разработчики для Rails подписываются на рассылку Err the Blog.

ления разбиением на страницы необходимо вызвать метод `will_paginate`, передав подлежащий выводу набор:

```
<%= will_paginate @posts %>
```

Авторы этого модуля предлагают даже CSS-стили, чтобы вы могли сделать элемент управления более симпатичным (рис. 11.1).

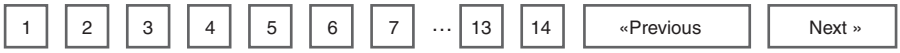

***Рис. 11.1.** Элемент управления разбиением на страницы, отформатированный с помощью CSS-стилей*

Если вы озабочены оптимизацией сайта для поисковых машин (SEO), то модуль `will_paginate`, скорее всего, вас устроит, так как он генерирует ссылки для каждой страницы в отдельности. Поисковые роботы будут видеть их как уникальные ссылки, переходить по каждой из них и, таким образом, проиндексируют все страницы. Традиционно считается, что робот переходит по ссылке «Следующая» всего несколько раз, а потом это ему надоедает, и он отправляется на более интересные сайты.

paginator

Этот проект[1] задуман известным экспертом по Rails Брюсом Уильямсом (Bruce Williams), размещен на сайте Rubyforge и кому-то может понравиться из-за своей простоты. Он представляет собой пакет Rubygem, а не подключаемый модуль, поэтому для установки нужно выполнить команду `sudo gem install paginator`.

Библиотека `paginator` не интегрируется с `ActiveRecord`, как `will_paginate`, а предоставляет лаконичный API для обертывания обращений к методу `find` в контроллере:

```
def index
  @pager = ::Paginator.new(Foo.count, PER_PAGE) do |offset, per_page|
    Foo.find(:all, :limit => per_page, :offset => offset)
  end
  @page = @pager.page(params[:page])
  ...
end
```

В код представления не нужно включать специальные методы-помощники, вместо этого в вашем распоряжении оказывается объект `@page`:

```
<% @page.items.each do |foo| %>
<%# Показать что-то для каждого элемента на странице %>
<% end %>
<%= @page.number %>
```

[1] Домашняя страница проекта Paginator находится по адресу *http://paginator.rubyforge.org/*.

```
<%= link_to("Prev", foos_url(:page => @page.prev.number)) if @page.prev? %>
<%= link_to("Next", foos_url(:page => @page.next.number)) if@page.next? %>
```

Paginating Find

Если у вас есть тяга к приключениям, можете скачать с сайта `http://svn.cardboardrocket.com/paginating_find` простую библиотеку для разбиения на страницы, которая расширяет метод `find` из `ActiveRecord`, а не пытается подменить его. Я думаю, что этот проект пока не такой зрелый, как `will_paginate` или `paginator`, но выглядит многообещающе.

Модуль RecordIdentificationHelper

Этот модуль обертывает методы библиотеки `ActionController::RecordIdentifier` и инкапсулирует ряд соглашений об именовании для работы с записями, например с моделями `ActiveRecord`, `ActiveResource` и вообще любыми объектами моделей, которые желательно представить в виде разметки (типа HTML) и у которых есть атрибут `id`. Описанные паттерны затем используются в попытке поднять действия представлений на более высокий логический уровень.

Пусть, например, в файле маршрутов определен маршрут `map.resources :posts`, а в представлении имеется такой код:

```
<% div_for(post) do %>
  <%= post.body %>
<% end %>
```

Для элемента `DIV` в этом случае генерируется следующая разметка:

```
<div id="post_45" class="post">
  Как прекрасен этот мир!
</div>
```

Обратите внимание на соглашение, отраженное в атрибуте `id`. Теперь перейдем к контроллеру, в котором есть рассчитанный на применение AJAX метод `destroy`. Идея в том, чтобы его можно было вызвать для удаления записи, которая должна исчезнуть со страницы без перезагрузки последней:

```
def destroy
  post = Post.find(params[:id])
  post.destroy

  respond_to do |format|
    format.html { redirect_to :back }
    format.js do
      # Вызывает: new Effect.fade('post_45');
      render(:update) { |page| page[post].visual_effect(:fade) }
    end
  end
end
```

Как видно из предыдущего примера, можно почти не думать, что на самом деле представляет собой идентификатор модели (то есть элемент DIV, в котором хранится информация из модели). Вы лишь знаете, что какой-то идентификатор у нее есть и что последующие обращения к RJS ожидают следования одному и тому же соглашению об именовании, так что, придерживаясь его, можно будет писать меньше кода. Более подробно об этой технике рассказано в главе 12.

dom_class(record_or_class, prefix = nil)

При именовании классов DOM принято соглашение, согласно которому имя объекта или класса записывается в единственном числе.

```
dom_class(post) # => "post"
dom_class(Person) # => "person"
```

Если вам нужно адресовать несколько экземпляров одного и того же класса в одном представлении, можно задать префикс класса DOM:

```
dom_class(post, :edit) # => "edit_post"
dom_class(Person, :edit) # => "edit_person"
```

dom_id(record, prefix = nil)

При именовании классов DOM принято соглашение, согласно которому имя объекта или класса записывается в единственном числе и после подчерка указывается идентификатор. Если идентификатор не найден, употребляется префикс new_:

```
dom_class(Post.new(:id => 45)) # => "post_45"
dom_class(Post.new) # => "new_post"
```

Если вам нужно адресовать несколько экземпляров одного и того же класса в одном представлении, можно задать для метода dom_id префикс. Например, обращение dom_class(Post.new(:id => 45), :edit) порождает строку edit_post_45.

partial_path(record_or_class)

Возвращает строку, содержащую имя записи или класса во множественном и единственном числе, что очень полезно для автоматического рендеринга подшаблонов в соответствии с соглашениями:

```
partial_path(post) # => "posts/post"
partial_path(Person) # => "people/person"
```

Модуль RecordTagHelper

Этот модуль тесно связан с модулем RecordIdentificationHelper, поскольку помогает в создании HTML-разметки, следующей четким и продуманным соглашениям о наименовании.

content_tag_for(tag_name, record, *args, &block)

Этот помощник создает HTML-элемент, для которого атрибуты `id` и `class` соотносятся с заданным объектом `ActiveRecord`.

Пусть, например, `@person` — экземпляр класса `Person`, в котором поле `id` равно 123. Тогда следующий шаблон:

```
<% content_tag_for(:tr, @person) do %>
  <td><%=h @person.first_name %></td>
  <td><%=h @person.last_name %></td>
<% end %>
```

порождает такую разметку:

```
<tr id="person_123" class="person">
  ...
</tr>
```

Если необходимо, чтобы атрибут `id` в HTML-разметке имел определенный префикс, вы можете задать его с помощью третьего аргумента:

```
>> content_tag_for(:tr, @person, :foo) do ...
=> <tr id="foo_person_123" class="person">...
```

Помощник `content_tag_for` принимает также хеш с дополнительными параметрами, которые преобразуются в HTML-атрибуты тега. Если задать в нем ключ `:class`, то его значение будет объединено с именем класса по умолчанию, а не заменит полностью (поскольку это свело бы на нет весь смысл метода!):

```
>> content_tag_for(:tr, @person, :foo, :class => 'highlight') do ...
=> <tr id="foo_person_123" class="person highlight">...
```

div_for(record, *args, &block)

Порождает элемент `DIV`, для которого атрибуты `id` и `class` соотносятся с заданным объектом `ActiveRecord`. Этот метод ведет себя аналогично `content_tag_for`, только в нем уже «зашит» вывод элементов `DIV`.

Модуль TagHelper

В этом модуле собраны помощники для программной генерации HTML-тегов.

cdata_section(content)

Возвращает раздел `CDATA`, обертывающий заданный текст `content`. Разделы `CDATA` используются для экранирования участков текста, содержащих символы, которые иначе распознавались бы как разметка. Каждый раздел `CDATA` начинается строкой `<![CDATA[` и заканчивается строкой `]]>`.

content_tag(name, content = nil, options = nil, &block)

Возвращает блочный HTML-тег `name`, окружающий заданный текст `content`. Дополнительные HTML-атрибуты можно задать в хеше `options`. Вместо того чтобы передавать текст в виде аргумента, вы можете предоставить блок, содержащий дополнительную разметку (и/или дополнительные обращения к методу `content_tag`), – тогда хеш `options` будет вторым, а не третьим параметром. Вот несколько простых примеров вызова `content_tag` без блока:

```
>> content_tag(:p, "Hello world!")
=> <p>Hello world!</p>

>> content_tag(:div, content_tag(:p, "Hello!"), :class => "message")
=> <div class="message"><p>Hello!</p></div>

>> content_tag("select", options, :multiple => true)
=> <select multiple="multiple">...options...</select>
```

А вот пример задания текста внутри блока (показано, как это выглядит в шаблоне, а не на консоли):

```
<% content_tag :div, :class => "strong" do -%>
  Hello world!
<% end -%>
```

Этот код порождает следующую HTML-разметку:

```
<div class="strong"><p>Hello world!</p></div>
```

escape_once(html)

Возвращает экранированный вариант HTML-разметки, не затрагивая уже имеющихся в ней экранированных компонентов:

```
>> escape_once("1 > 2 &amp; 3")
=> "1 &lt; 2 &amp; 3"

>> escape_once("&lt;&lt; Accept & Checkout")
=> "&lt;&lt; Accept &amp; Checkout"
```

tag(name, options = nil, open = false)

Возвращает пустой HTML-тег с именем `name`, по умолчанию в виде, отвечающем спецификации XHTML. Если параметр `open` равен `true`, создается открывающий тег, совместимый со стандартом HTML 4.0 и более ранними версиями. Дополнительные HTML-атрибуты можно передать в хеше `options`.

Если для некоторого атрибута не подразумевается наличие значения (например, `disabled` и `readonly`), то в хеше `options` можно указать для него значение `true`. Имена атрибутов могут задаваться в виде строк или символов:

```
>> tag("br")
=> <br />

>> tag(„br", nil, true)
=> <br>

>> tag(„input", { :type => ‚text', :disabled => true })
=> <input type="text" disabled="disabled" />

>> tag(„img", { :src => „open.png" })
=> <img src="open.png" />
```

Модуль TextHelper

Методы из этого модуля предназначены для фильтрации, форматирования и преобразования строк.

auto_link(text, link = :all, href_options = {}, &block)

Преобразует все URL и электронные почтовые адреса внутри строки `text` в гиперссылки. Параметр `link` позволяет уточнить, что именно следует преобразовывать; он может принимать значения `:email_addresses` или `:urls`. В порождаемые теги `a` можно добавить атрибуты, задав их в хеше `href_options`.

Если по какой-то причине вас не устраивает способ, которым Rails превращает почтовые адреса и URL в ссылки, можете указать в этом методе блок. Блоку передается каждый обнаруженный адрес, а возвращенное блоком значение подставляется в генерируемый текст в виде ссылки:

```
>> auto_link("Go to http://obiefernandez.com and say hello to
obie@obiefernandez.com")
=> "Зайдите на <a
href="http://www.rubyonrails.org">http://www.rubyonrails.org</a>           и
поздоровайтесь с <a
href="mailto:obie@obiefernandez.com">obie@obiefernandez.com</a>"

>> auto_link("Заходите на мой блог по адресу http://www.myblog.com/. Пишите
мне на адрес me@email.com.", :all, :target => '_blank') do |text|
truncate(text, 15)
end

=> "Заходите на мой блог по адресу <a href=\"http://www.myblog.com/\"
target=\"_blank\">http://www.m...</a>.
Пишите мне на адрес <a
href=\"mailto:me@email.com\">me@email.com</a>."
```

concat(string, binding)

Предпочтительный способ вывода текста в представлении – применение ERB-конструкции `<%= expression %>`. Обычные методы `puts` и `print` рабо-

тают в коде, написанном на eRuby, не так, как ожидается (если, конечно, вы ожидаете, что печатаемый с их помощью текст появится в броузере). Если вам необходимо выводить код в обход блока `<% expression %>`, можете воспользоваться методом `concat`. В моей практике он оказывался особенно полезен при написании собственных помощников.

cycle(first_value, *values)

Создает объект `Cycle`, в котором метод `to_s` при каждом обращении возвращает следующий элемент массива `values` (переходя от последнего снова к первому). Это можно использовать, например, для попеременного назначения CSS-класса строкам таблицы.

В следующем примере CSS-класс меняется для четных и нечетных строк (предполагается, что в переменной `@items` находится массив чисел от 1 до 4):

```
<table>
 <% @items.each do |item| %>
  <tr class="<%= cycle("even", "odd") -%>">
   <td>item</td>
  </tr>
 <% end %>
</table>
```

Как видно из этого примера, необязательно сохранять ссылку на объект `cycle` в локальной переменной или еще где-то – достаточно просто повторно вызывать метод `cycle`. Это удобно, но означает, что для вложенных циклов необходимы идентификаторы. Решение – передать методу `cycle` последним параметром имя `:name => cycle_name`. Кроме того, можно вручную перевести цикл в исходное состояние методом `reset_cycle`, которому передается имя цикла.

Ниже приведен набор данных, который надо обойти:

```
# Меняем CSS-классы соседних строк и цвет текста для значений в каждой
строке
@items = [{:first => 'Robert', :middle => 'Daniel', :last => 'James'},
          {:first => 'Emily', :last => 'Hicks'},
          {:first => 'June', :middle => 'Dae', :last => 'Jones'}]
```

А вот код шаблона. Поскольку количество выводимых ячеек может быть различно, перед началом обхода мы устанавливаем цикл в исходное состояние:

```
<% @items.each do |item| %>
<tr class="<%= cycle("even", "odd", :name => "row_class")
  <% item.values.each do |value| %>
    <td style="color:<%= cycle("red", "green", :name => "colors") -%>">
    <%= value %>
    </td>
  <% end %>
```

```
  <% reset_cycle("colors") %>
</tr>
<% end %>
```

excerpt(text, phrase, radius = 100, excerpt_string = "...")

Извлекает из текста фрагмент, соответствующий первому вхождению фразы `phrase`. Параметр `radius` задает количество дополнительных символов по обе стороны выдержки (по умолчанию равен 100). Если при этом достигается начало или конец текста, с соответствующей стороны добавляется строка `excerpt_string`. Если искомая фраза не найдена, метод возвращает `nil`:

```
>> excerpt('This is an example', 'an', 5)
=> "...s is an examp..."

>> excerpt('This is an example', 'is', 5)
=> "This is an..."

>> excerpt('This is an example', 'is')
=> "This is an example"

>> excerpt('This next thing is an example', 'ex', 2)
=> "...next t..."

>> excerpt('This is also an example', 'an', 8, '<chop> ')
=> "<chop> is also an example"
```

highlight(text, phrases, highlighter = '\1')

Выделяет во всем тексте одну или несколько фраз, вставляя заданную строку-выделитель `highlighter`. Выделитель может быть задан в виде заключенной в одиночные кавычки строки, где встречается последовательность `\1`. Вместо нее будет поставлена выделяемая фраза:

```
>> highlight('You searched for: rails', 'rails')
=> You searched for: <strong class="highlight">rails</strong>

>> highlight('You searched for: ruby, rails, dhh', 'actionpack')
=> You searched for: ruby, rails, dhh

>> highlight('You searched for: rails', ['for', 'rails'],
'<em>\1</em>')
=> You searched <em>for</em>: <em>rails</em>

>> highlight('You searched for: rails', 'rails', "<a
href='search?q=\1'>\1</a>")
=> You searched for: <a href='search?q=rails>rails</a>
```

markdown(text)

Возвращает текст, в котором все коды Markdown преобразованы в HTML-теги. Этот метод доступен, только если установлен gem-пакет BlueCloth:

```
>> markdown("We are using __Markdown__ now!")
=> "<p>We are using <strong>Markdown</strong> now!</p>"

>> markdown("We like to _write_ `code`, not just _read_ it!")
=> "<p>We like to <em>write</em> <code>code</code>, not just
<em>read</em> it!</p>"

>> markdown("The [Markdown
website](http://daringfireball.net/projects/markdown/) has more
information.")
=> "<p>The <a
href="http://daringfireball.net/projects/markdown/">Markdown
website</a> has more information.</p>"

>> markdown('![The ROR logo](http://rubyonrails.com/images/rails.png
"Ruby on Rails")')
=> '<p><img src="http://rubyonrails.com/images/rails.png" alt="The ROR
logo" title="Ruby on Rails" /></p>'
```

pluralize(count, singular, plural = nil)

Пытается найти множественную форму слова `singular`, написанного в единственном числе, если параметр `count` не равен 1. Если в параметре `plural` уже задана множественная форма, то при `count` > 1 она и будет использована. Если загружен класс `Inflector` из библиотеки `ActiveSupport`, он применяется для определения множественной формы, в противном случае к `singular` просто дописывается в конце буква s:

```
>> pluralize(1, 'person')
=> "1 person"

>> pluralize(2, 'person')
=> "2 people"

>> pluralize(3, 'person', 'users')
=> "3 users"

>> pluralize(0, 'person')
=> "0 people"
```

reset_cycle(name = "default")

Возвращает цикл в исходное состояние (см. описание метода `cycle` из этого модуля), так что при следующем вызове перебор начнется с первого элемента. В параметре `name` можно указать имя цикла.

sanitize(html)

Обезвреживает HTML-код, преобразуя теги `<form>` и `<script>` в обычный текст и удаляя все атрибуты, начинающиеся со строки `on` (например, `onClick`), чтобы нельзя было выполнить произвольный JavaScript-код. Кроме того, удаляются все атрибуты `href` и `src`, начинающиеся со строки `"javascript:"`. Можно уточнить, что именно следует обезвреживать, определив переменные `VERBOTEN_TAGS` и `VERBOTEN_ATTRS` перед загрузкой этого модуля:

```
>> sanitize('<script> do_nasty_stuff() </script>')
=> &lt;script> do_nasty_stuff() &lt;/script>

>> sanitize('<a href="javascript: sucker();">Click here for $100</a>')
=> <a>Click here for $100</a>

>> sanitize('<a href="#" onClick="kill_all_humans();">Click
here!!!</a>')
=> <a href="#">Click here!!!</a>

>> sanitize('<img src="javascript:suckers_run_this();" />')
=> <img />
```

> **Говорит Кортенэ...**
>
> Более качественное обезвреживание обеспечивает подключаемый модуль Whitelist Рика Олсона, который можно найти по адресу *http://svn.techno-weenie.net/projects/plugins/white_list*.

simple_format(text)

Возвращает текст, преобразованный в HTML с помощью простых правил форматирования. Два и более последовательных знака перехода на новую строку (`\n\n`) обозначают новый абзац, то есть текст заключается в теги `P`. Одиночный знак перехода обозначает разрыв строки, поэтому вставляется тег `BR`. Данный метод не удаляет символы `\n` из текста.

strip_links(text)

Удаляет из текста все теги гиперссылок, оставляя только сам текст:

```
>> strip_links('<a href="http://www.rubyonrails.org">Ruby on
Rails</a>')
=> Ruby on Rails

>> strip_links('Пишите мне на адрес <a
href="mailto:me@email.com">me@email.com</a>.')
=> Пишите мне на адрес me@email.com.
```

```
>> strip_links('Блог: <a href="http://www.myblog.com/" class="nav"
target="_blank">Заходите</a>.')
=> Блог: Заходите
```

strip_tags(html)

Удаляет из текста все HTML-теги, включая комментарии. Этот метод пользуется лексическим анализатором `html-scanner`, который и ограничивает его возможности по разбору HTML-кода.

```
>> strip_tags("Удалить <i>эти</i> теги!")
=> Удалить эти теги!

>> strip_tags("Нет <b>жирному шрифту</b>! <a href='more.html'>См. далее
здесь</a>...")

=> Нет жирному шрифту! См. далее здесь...
>> strip_tags("<div id='top-bar'>Добро пожаловать на мой сайт!</div>")
=> Добро пожаловать на мой сайт!
```

textilize(text)

Этот метод доступен, только если установлен gem-пакет RedCloth. Он возвращает `text`, в котором все коды Textile преобразованы в HTML-теги (подробнее о синтаксисе языка разметки Textile см. на сайте `http://hobix.com/textile/`):

```
>> textilize("*Это Textile!* Ликуйте!")
=> "<p><strong>Это Textile!</strong> Ликуйте!</p>"

>> textilize("Я _люблю_ ROR (Ruby on Rails)!")
=> "<p>Я <em>люблю</em> <acronym title="Ruby on
Rails">ROR</acronym>!</p>"

>> textilize("h2. Textile -упрощает- разметку!")
=> "<h2>Textile <del>упрощает</del> разметку!</h2>"

>> textilize("На сайт Rails "сюда":http://www.rubyonrails.org/.)
=> "<p>На сайт Rails <a
href="http://www.rubyonrails.org/">сюда</a>.</p>"
```

textilize_without_paragraph(text)

Возвращает `text`, в котором все коды Textile преобразованы в HTML-теги, но без обрамляющих тегов `<p>`, добавляемых RedCloth.

truncate(text, length = 30, truncate_string = "...")

Если длина текста `text` превышает `length`, то `text` усекается до заданной длины, а вслед за ним добавляется строка `truncate_string`:

```
>> truncate("Когда-то давным-давно в тридесятом государстве", 8)
=> "Когда-то..."
```

```
>> truncate("Когда-то давным-давно в тридесятом государстве")
=> "Когда-то давным-давно в тридес..."

>> truncate("И обнаружилось, что многие люди стали спать лучше.",
14, "... (далее)")
=> "И обнаружилось... (continued)"
```

word_wrap(text, line_width = 80)

Разбивает текст на строки длиной не более `line_width`. Строка разбивается в месте последнего пробельного символа, при котором длина еще не превышает `line_width` (по умолчанию 80):

```
>> word_wrap('Once upon a time', 4)
=> "Once\nupon\na\ntime"

>> word_wrap('Once upon a time', 8)
=> "Once upon\na time"

>> word_wrap('Once upon a time')
=> "Once upon a time"

>> word_wrap('Once upon a time', 1)
=> "Once\nupon\na\ntime"
```

Модуль UrlHelper

Этот модуль содержит ряд методов для генерации ссылок и порождения URL, зависящих от системы маршрутизации, которая подробно рассматривалась в главах 3–5.

button_to(name, options = {}, html_options = {})

Генерирует форму, содержащую единственную кнопку отправки на URL, определяемый параметрами в хеше `options`. Это самый надежный способ гарантировать, что ссылки, которые изменяют данные, не будут «щелкаться» роботами или ускорителями. Если наличие HTML-кнопки противоречит стилю вашего макета, можно воспользоваться методом `link_to` (также из этого модуля) с модификатором `:method`.

В хеше `options` могут находиться те же параметры, что для метода `url_for` (из этого же модуля).

Для сгенерированного элемента `FORM` указывается CSS-класс с именем `button-to`, позволяющий стилизовать саму форму и ее потомков. Параметры `:method` и `:confirm` работают так же, как для помощника `link_to`. Если модификатор `:method` не задан, по умолчанию подразумевается метод `POST`. Кнопку можно сделать неактивной, задав параметр `:disabled => true`.

```
>> button_to "Создать", :action => "new"
=> "<form method="post" action="/controller/new" class="button-to">
      <div><input value="Создать" type="submit" /></div>
    </form>"

>> button_to "Удалить картинку", { :action => "delete", :id => @image.id },
:confirm => "Вы уверены?", :method => :delete

=> "<form method="post" action="/images/delete/1" class="button-to">
      <div>
        <input type="hidden" name="_method" value="delete" />
        <input onclick="return confirm(Вы уверены?');"
value="Удалить"
type="submit" />
</div>
</form>"
```

current_page?(options)

Возвращает true, если URL текущего запроса был сгенерирован с помощью указанных параметров. Пусть, например, сейчас выполняется рендеринг для действия /shop/checkout:

```
>> current_page?(:action => 'process')
=> false

>>current_page?(:action => 'checkout') # контроллер неявно подразумевается
=> true

>> current_page?(:controller => 'shop', :action => 'checkout')
=> true
```

link_to(name, options = {}, html_options = nil)

Один из самых важных методов-помощников. Создает тег ссылки с заданным текстом `name`, который ведет на URL, порождаемый с помощью параметров в хеше `options`. Допустимые параметры описаны в разделе, посвященном методу `url_for`. Вместо хеша `options` можно передать строку, которая станет значением атрибута `href`. Если `name = nil`, текстом ссылки становится сам URL:

- `:confirm => 'question?'` добавляет JavaScript-сценарий с предложением ответить на указанный вопрос. Если пользователь отвечает утвердительно, ссылка обрабатывается нормально, в противном случае не предпринимается никаких действий;
- `:popup => true` открывает ссылку во всплывающем окне. Можно также задать строку параметров, передаваемых методу JavaScript `window.open`;
- `:method => symbol` задает альтернативный глагол HTTP для данного запроса (отличный от GET). Этот модификатор приводит к динами-

ческому созданию HTML-формы и отправке ее серверу указанным методом (`:post`, `:put`, `:delete` или специализированным, заданным в виде строки, например `"HEAD"`).

Вообще говоря, GET-запросы должны быть идемпотентными, то есть не модифицировать состояние ресурса на сервере. Поэтому их можно вызывать многократно без негативных последствий. Запросы, которые модифицируют ресурсы сервера или приводят к выполнению таких опасных действий, как удаление записи, не должны ассоциироваться с нормальной гиперссылкой, поскольку поисковые роботы и так называемые акселераторы броузеров могут переходить по таким ссылкам во время посещения вашего сайта, оставляя за собой хаос.

Если пользователь отключил JavaScript, запрос будет выполнен методом GET вне зависимости от того, что указано в параметре `:method`. Достигается это путем включения корректного атрибута `href`. Если приложению необходимо, чтобы запрос был отправлен каким-то конкретным методом, контроллер должен проверить этот факт, пользуясь методами `post?`, `delete?` или `put?` объекта `request`.

Как обычно, в хеше `html_options` можно передать HTML-атрибуты тега `a`:

```
>> link_to "Перейти на другой сайт", "http://www.rubyonrails.org/",
:confirm => "Вы уверены?"
=> "<a href="http://www.rubyonrails.org/" onclick="return confirm('Вы
уверены?');">Перейти на другой сайт</a>"

>> link_to "Справка", { :action => "help" }, :popup => true
=> "<a href="/testing/help/" onclick="window.open(this.href);return
false;">Справка</a>"

>> link_to "Показать картинку", { :action => "view" }, :popup =>
['new_window_name', 'height=300,width=600']
=> "<a href="/testing/view/" onclick="window.open(this.href,
'new_window_name','height=300,width=600');return false;">Показать
картинку</a>"

>> link_to "Удалить картинку", { :action => "delete", :id => @image.id },
:confirm => "Вы уверены?", :method => :delete
=> <a href="/testing/delete/9/" onclick="if (confirm('Вы уверены?'))
{
var f = document.createElement('form');
       f.style.display = 'none'; this.parentNode.appendChild(f);
f.method = 'POST'; f.action = this.href;
       var m = document.createElement('input'); m.setAttribute('type',
'hidden'); m.setAttribute('name', '_method');
        m.setAttribute('value', 'delete'); f.appendChild(m);f.submit();
};return false;">Удалить картинку</a>
```

link_to_if(condition, name, options = {}, html_options = {}, &block)

Создает тег ссылки с теми же параметрами, что метод `link_to`, если условие `condition` равно `true`. В противном случае выводит только значение `name` (или значение, вычисленное блоком `block`, если он задан).

link_to_unless(condition, name, options = {}, html_options = {}, &block)

Создает тег ссылки с теми же параметрами, что метод `link_to`, если условие `condition` не равно `true`. В противном случае выводит только значение `name` (или значение, вычисленное блоком `block`, если он задан).

link_to_unless_current(name, options = {}, html_options = {}, &block)

Создает тег ссылки с теми же параметрами, что метод `link_to`, если URI текущего запроса совпадает с URI этой ссылки. В противном случае выводит только значение `name` (или значение, вычисленное блоком `block`, если он задан).

Этот метод иногда оказывается весьма кстати. Подчеркнем, что переданный ему блок вычисляется, если текущее действие совпадает с указанным. Поэтому, если бы на странице комментариев мы захотели вывести ссылку «Назад» вместо ссылки на ту же самую страницу комментариев, то могли бы поступить следующим образом:

```
<%= link_to_unless_current("Comment", { :controller => 'comments',
    :action => 'new'}) do
      link_to("Назад", { :controller => 'posts', :action => 'index'
})
  end %>
```

mail_to(email_address, name = nil, html_options = {})

Создает тег ссылки `mailto`, ведущий на указанный почтовый адрес `email_address`, который одновременно является текстом ссылки, если не указан параметр `name`. В хеше `html_options` можно передать дополнительные параметры тега.

Помощник `mail_to` поддерживает несколько способов противодействия сборщикам постовых адресов и модификации самого почтового адреса. Все они управляются параметрами в хеше `html_options`:

- `:encode`. Этот ключ может принимать в качестве значений строки `"javascript"` или `"hex"`. В случае строки `"javascript"` динамически создается и кодируется ссылка `mailto:`, а затем вызывает метод `eval` для вставки ее в DOM страницы. Если пользователь отключил JavaScript, то созданная таким способом ссылка не показывается вовсе.

В случае строки "hex" адрес email_address перед выводом в ссылку mailto: представляется в шестнадцатеричном виде;

- :replace_at. Если параметр name не задан, в качестве текста ссылки фигурирует email_address. Эта опция позволяет замаскировать email_address путем подстановки указанной строки вместо знака @;
- :replace_dot. Если параметр name не задан, в качестве текста ссылки фигурирует email_address. Эта опция позволяет замаскировать email_address путем подстановки указанной строки вместо точки в почтовом адресе;
- :subject. Тема почтового сообщения;
- :body. Тело почтового сообщения;
- :cc. Получатели копии почтового сообщения;
- :bcc. Получатели слепой копии почтового сообщения.

Ниже приведены примеры использования метода:

```
>> mail_to "me@domain.com"
=> <a href="mailto:me@domain.com">me@domain.com</a>

>> mail_to "me@domain.com", "My email", :encode => "javascript"
=> <script type="text/javascript">eval(unescape('%64%6f%63...%6d%65'))
</script>

>> mail_to "me@domain.com", "My email", :encode => "hex"
=> <a href="mailto:%6d%65@%64%6f%6d%61%69%6e.%63%6f%6d">My email</a>

>> mail_to "me@domain.com", nil, :replace_at => "_at_", :replace_dot =>
"_dot_", :class => "email"
=> <a href="mailto:me@domain.com"
class="email">me_at_domain_dot_com</a>

>> mail_to "me@domain.com", "My email", :cc => "ccaddress@domain.com",
:subject => "This is an example email"
=> <a
href="mailto:me@domain.com?cc=ccaddress@domain.com&subject=This%20i
s%20an%20example%20email">My email</a>
```

url_for(options = {})

Метод url_for возвращает URL, построенный по указанным в хеше options параметрам. Сами параметры такие же, как для метода url_for из класса ActionController (он подробно обсуждался в главе 3 «Маршрутизация»).

Отметим, что по умолчанию параметр :only_path равен true, так что получается относительный путь /controller/action, а не полностью квалифицированный URL вида http://example.com/controller/action.

Если метод url_for вызван из представления, то он возвращает URL, к которому применено экранирование HTML. Если вам необходим не-

экранированный URL, передайте в хеше `options` параметр `:escape => false`.

Ниже приведен полный список параметров, которые можно задать в хеше `options` для метода `url_for`:

- `:anchor`. Задает якорь (`#anchor`), добавляемый в конец пути;
- `:only_path`. Говорит, что надо генерировать относительный URL (опустив протокол, имя хоста и номер порта);
- `:trailing_slash`. Добавляет завершающую косую черту, например `"/archive/2005/"`. Отметим, что задавать этот параметр не рекомендуется, так как он вступает в конфликт с кэшированием;
- `:host`. Переопределяет подразумеваемое по умолчанию (текущее) имя хоста;
- `:protocol`. Переопределяет подразумеваемый по умолчанию (текущий) протокол;
- `:user`. Встроенная аутентификация HTTP (необходимо задать также параметр `:password`);
- `:password`. Встроенная аутентификация HTTP (необходимо задать также параметр `:user`);
- `:escape`. Определяет, надо ли применять к возвращаемому URL экранирование HTML.

```
>> url_for(:action => 'index')
=> /blog/

>> url_for(:action => 'find', :controller => 'books')
=> /books/find

>> url_for(:action => 'login', :controller => 'members', :only_path =>
false, :protocol => 'https')
=> https://www.railsapplication.com/members/login/

>> url_for(:action => 'play', :anchor => 'player')
=> /messages/play/#player

>> url_for(:action => 'checkout', :anchor => 'tax&ship')
=> /testing/jump/#tax&amp;ship

>> url_for(:action => 'checkout', :anchor => 'tax&ship', :escape =>
false)
=> /testing/jump/#tax&ship
```

Связь с именованными маршрутами

Если любому методу из модуля `UrlModule`, который принимает те же параметры, что `url_for`, передать не хеш, а экземпляр модели `ActiveRecord` или `ActiveResource`, будет сгенерирован путь для маршрута к этой запи-

си (если таковой существует). Поиск производится по имени класса, причем алгоритм достаточно «умен», чтобы, вызвав метод `new_record?` для переданной модели, определить, нужна ли ссылка на маршрут к *набору* или *отдельному члену*.

Например, при передаче объекта `Timesheet` будет предпринята попытка использовать маршрут `timesheet_path`. Если маршрут к этому объекту вложен в другой маршрут, придется вызывать помощник маршрута явно, так как Rails не в состоянии определить это автоматически:

```
>> url_for(Workshop.new)
=> /workshops

>> url_for(@workshop) # existing record
=> /workshops/5
```

Написание собственных модулей

При разработке приложения для Rails нужно пользоваться любой возможностью вынести дублирующийся код в методы-помощники. Специализированные помощники помещаются в один из модулей, находящихся в папке `app/helpers` вашего приложения.

Написание эффективных методов-помощников – искусство сродни разработке эффективных API. По существу, помощник – это специализированный API для кода представления, существующий на уровне одного приложения. Очень трудно научить проектированию API в книге. Это знание приобретается в результате обучения у более опытных программистов и многочисленных проб и ошибок. Тем не менее в данном разделе мы рассмотрим несколько сценариев и стилей реализации в надежде, что вы найдете что-то подходящее для вашего приложения.

Мелкие оптимизации: помощник Title

Здесь рассматривается простой метод-помощник, который я использовал во многих своих проектах. Называется он `page_title` и объединяет две простые функции, необходимые в любом добротном HTML-документе:

- установку заголовка страницы `title` в элементе head документа
- краткое описание назначения страницы в элементе `h1`

Предполагается, что вы хотите сделать элементы `title` и `h1` одинаковыми и увязать метод с шаблоном приложения. В листинге 11.6 приведен код помощника, который следует включить в файл `app/helpers/application_helper.rb`, поскольку он должен быть доступен всем представлениям.

Листинг 11.6. Помощник title

```
def page_title(name)
  @title = name
  content_tag("h1", name)
end
```

Сначала метод устанавливает значение переменной `@title`, а затем выводит элемент `h1`, содержащий тот же самый текст. Во второй строке можно было бы применить интерполяцию в строку вида `"<h1>#{name}</h1>"`, но мне кажется, что это хуже, чем воспользоваться встроенным в Rails помощником `content_tag`.

Мой шаблон приложения ищет переменную `@page_title` и, если находит, выводит ее перед названием сайта:

```
<html>
  <head>
    <title><%= "#@page_title - " if @page_title %>Название сайта</title>
```

Понятно, что метод `page_title` следует вызывать в месте шаблона представления, где должен появиться элемент `h1`:

```
<%= page_title "Новый пользователь" %>
<%= error_messages_for :user %>
<% form_for(:user, :url => users_path) do |f| %>
  ...
```

Инкапсуляция логики представления: помощник photo_for

Вот еще один сравнительно простой пример. На этот раз мы не просто будем выводить данные, а инкапсулируем некую логику для определения выводимых данных: фотографии из профиля пользователя или изображения-заглушки. В противном случае этот кусок пришлось бы повторять в разных местах приложения.

Контракт с данным помощником заключается в том, что с объектом, представляющим пользователя, ассоциирован объект `profile_photo`, который является экземпляром модели вложения, основанной на подключаемом модуле `attachment_fu` Рика Олсона. Код в листинге 11.7 понятен и без детального описания этого модуля. Чтобы не усложнять пример, я решил присваивать значение переменной `src` в предложении `if/else`; а вообще-то здесь просто напрашивается *тернарный оператор* Ruby.

Листинг 11.7. Помощник photo_for, инкапсулирующий логику, общую для разных представлений

```
def photo_for(user, size=:thumb)
  if user.profile_photo
```

```
    src = user.profile_photo.public_filename(size)
  else
    src = 'user_placeholder.png'
  end
  link_to(image_tag(src), user_path(user))
end
```

Более сложное представление: помощник breadcrumbs

Во многих веб-приложениях встречается идея *хлебных крошек* (breadcrumbs). Под этим понимается расположенный в верхней части страницы список ссылок, показывающий, насколько далеко пользователь углубился в иерархически организованный сайт. Мне кажется, что имеет смысл вынести логику построения «хлебных крошек» в отдельный метод-помощник, а не оставлять ее в шаблоне макета.

Идея нашей реализации (листинг 11.8) заключается в том, чтобы воспользоваться наличием переменных экземпляра (зависящих от принятых в вашем приложении соглашений) для определения того, нужно ли добавлять элементы в массив ссылок-крошек.

Листинг 11.8. Помощник breadcrumbs для корпоративного справочника

```
1 def breadcrumbs
2   return if controller.controller_name == 'homepage'

3   html = [link_to('Home', home_path)]

4   # первый уровень
5   html << link_to('Компании', companies_path) if @companies ||
    @company
6   html << link_to(@company, company_path(@company)) if @company

7   # второй уровень
8   html << link_to('Отделы', departments_path) if @depts || @dept
9   html << link_to(@dept, department_path(@dept)) if @dept

10  # третий и последний уровень
11  html << link_to('Сотрудники', employees_path) if @employees ||
    @employee
12  html << link_to(@employee.name, employee_path(@employee)) if
    @employee

13  html.join(' &gt; ')
14 end
```

Приведем построчные пояснения к этому коду, отмечая предположения, сделанные при проектировании приложения.

В строке 2 мы завершаем выполнение, если находимся в контексте контроллера `homepage`, поскольку на начальной странице «хлебные крошки» не нужны. Предложение `return` без указания значения неяв-

но возвращает `nil`, что нас вполне устраивает – в шаблон ничего не выводится.

В строке 3 мы начинаем строить в локальной переменной `html` массив HTML-ссылок. В конечном итоге он будет содержать все «хлебные крошки». Первая ссылка указывает на начальную страницу приложения; она, конечно, зависит от приложения, но всегда есть, поэтому мы сразу же помещаем ее в массив. В данном примере предполагается, что на начальную страницу ведет *именованный маршрут* `home_path`.

Инициализировав массив `html`, нам остается проверить наличие переменных, образующих иерархию (строки 4–12). Предполагается, что если некоторый отдел выведен, то компания, к которой он относится, тоже находится в текущем контексте. Это не произвольное решение, а типичный паттерн для приложений Rails, построенных на принципах стиля REST, в которых применяются вложенные ресурсы.

Наконец, в строке 13 массив HTML-ссылок с помощью метода `join` преобразуется в строку с разделителем `>`, так что мы получаем традиционное представление «хлебных крошек».

Обертывание и обобщение подшаблонов

Я не думаю, что сами по себе подшаблоны гарантируют какой-то особенно элегантный или лаконичный код шаблона. Если в моем приложении постоянно встречается один и тот же подшаблон, я обертываю его в специализированный метод-помощник, который четко описывает назначение подшаблона и формализует параметры. Если это имеет смысл, я обобщаю реализацию с целью получить облегченный, повторно используемый компонент.

О, ересь, ересь! Слово *компонент* – ругательство в среде приверженцев Rails. Компоненты в Rails настолько презираемы, что я их даже рассматривать в этой книге не буду, а в версии Rails 2.0 они вообще изъяты. Говоря о компонентах, я всего лишь имею в виду фрагмент пользовательского интерфейса, допускающий простое повторное использование.

Помощник tiles

Рассмотрим шаги, необходимые для написания метода-помощника, обертывающего подшаблон общего назначения. В листинге 11.9 приведен код подшаблона, соответствующего фрагменту пользовательского интерфейса, который встречается во многих приложениях и обычно называется *изразцом* (tile). Состоит он из двух частей: уменьшенной фотографии слева и названия-ссылки и краткого описания справа.

С помощью изразцов можно представить и другие модели, например пользователей и файлы. Как уже было сказано, изразцы очень распро-

странены в современных пользовательских интерфейсах и операционных системах. Поэтому превратим подшаблон изразца `cities` в нечто пригодное для отображения других типов данных.

Примечание

Я понимаю, что на HTML-таблицы теперь поглядывают косо, и готов согласиться с тем, что верстка на базе элементов `DIV` в сочетании с CSS обеспечивает бóльшую гибкость. Но, поскольку в данном примере структура пользовательского интерфейса все-таки табличная, я решил не усложнять и остановился на таблицах.

Листинг 11.9. Подшаблон изразца до обертывания и обобщения

```
1  <table class="cities tiles">
2   <% cities.in_groups_of(columns) do |row| -%>
3    <tr>
4    <% row.each do |city| -%>
5     <td id="<%= dom_id(city) %>">
6      <div class="left">
7       <%= image_tag city.main_photo.public_filename(:thumb) -%>
8      </div>
9      <div class="right">
10      <div class="title"><%= city.name %></div>
11      <div class="description"><%= city.description %></div>
12     </div>
13    </td>
14   <% end # row.each -%>
15   </tr>
16  <% end # in_groups_of -%>
17 </table>
```

Пояснения к коду подшаблона изразца

Поскольку мы собираемся превратить этот специализированный для городов подшаблон в обобщенный компонент пользовательского интерфейса, я хочу, чтобы вы для начала хорошо поняли исходный код. Поэтому объясню все, что делается в листинге 11.9.

В строке 1 открывается элемент `table`, и ему приписываются семантически значимые CSS-классы, чтобы саму таблицу и ее содержимое можно было стилизовать.

В строке 2 используется полезное расширение `in_groups_of` класса `Array`, предоставляемое Rails. Здесь встречаются обе наши локальные переменные: `cities` и `columns`. Их необходимо передать в подшаблон с помощью хеша `:locals` при вызове метода `render :partial`. В переменной `cities` находится список отображаемых городов, а в переменной `columns` – целое число, показывающее, сколько изразцов должно содержаться в каждой строке таблицы. В цикле строятся все строки.

В строке 3 открывается строка таблицы – элемент `<tr>`.

В строке 4 начинается цикл по изразцам в одной строке, причем блоку передается текущий город `city`.

В строке 5 открывается элемент `<td>` и используется помощник `dom_id`, который автоматически генерирует идентификатор ячейки таблицы в виде `city_98`, `city_99` и так далее.

В строке 6 открывается элемент `<div>`, соответствующий левой стороне изразца, и ему назначается подходящий CSS-класс, чтобы в дальнейшем его можно было стилизовать.

В строке 7 вызывается помощник `image_tag`, который вставляет уменьшенную фотографию города.

В строке 10 вставляется содержимое в `DIV` с классом `title`. В данном случае это название города и штат, где он находится.

В строке 11 напрямую вызывается метод `description`, а в оставшихся строках закрываются циклы и контейнерные элементы.

Вызов подшаблона Tiles

Чтобы воспользоваться этим подшаблоном, мы должны вызвать метод `render :partial`, передав ему в хеше `:locals` два обязательных параметра:

```
render :partial => "cities/tiles",
       :locals => { :collection => @user.cities, :columns => 3 }
```

Догадываюсь, что многие опытные программисты писали подобные подшаблоны и думали, как включить значения по умолчанию для некоторых параметров. В данном случае было бы хорошо не задавать каждый раз значение `:columns`, так как в большинстве случаев мы хотим иметь три колонки.

Проблема в том, что, поскольку параметры передаются в хеше `:locals` и становятся локальными переменными, не существует простого способа вставить значение по умолчанию в самом подшаблоне. Если опустить часть `:columns => n` при вызове подшаблона, то Rails возбудит исключение, сообщив, что не существует ни локальной переменной, ни метода `columns`. Это вам не переменная экземпляра, с которой можно обращаться беззаботно, потому что по умолчанию она равна `nil`.

Опытные «рубисты», вероятно, знают о методе `defined?`, который позволяет выяснить, есть ли в текущей области видимости указанная локальная переменная, но при его использовании код получается уж очень уродливым. Следующий вариант можно было бы счесть элегантным, *но он не работает!*[1]

```
<% columns = 3 unless defined? columns %>
```

[1] Если вы знакомы с Ruby, то, наверное, в курсе, что метод Proc.new и его синоним proc тоже позволяют создавать анонимные блоки кода. Из-за некоторых тонких различий я предпочитаю лямбда-выражения. В блоке лямбда-выражения проверяется *арность* переданного при вызове списка аргументов, а явный возврат из блока работает корректно.

Вместо того чтобы рассказывать о непростых идиомах Ruby, я просто покажу, как данная проблема решается в Rails, и именно с этого места мы начнем обсуждать технику обертывания помощников.

Написание метода-помощника

Во-первых, я добавлю в модуль `CitiesHelper` приложения новый метод-помощник (листинг 11.10). Поначалу он будет довольно простым. Когда я размышлял, как этот метод назвать, мне пришла в голову мысль, что `tiled(@cities)` читается лучше, чем `tiles(@cities)`, поэтому я и выбрал имя `tiled`.

Листинг 11.10. Метод Tiled из модуля CitiesHelper

```
module CitiesHelper

  def tiled(cities, columns=3)
    render :partial => "cities/tiles",
           :locals => { :collection => cities, :columns => columns }
    end
end
```

О значении параметра `columns` по умолчанию я подумал с самого начала, и задал его в параметрах помощника. Это стандартная возможность языка Ruby.

Теперь, вместо того чтобы вызывать в шаблоне представления метод `render :partial`, я могу просто написать `<%= tiled(@cities) %>`, что намного элегантнее и короче. Кроме того, я тем самым разорвал связь между реализацией таблицы изразцов и представлением. Если в будущем мне потребуется изменить способ рендеринга таблицы, делать это придется только в одном месте – в методе-помощнике.

Обобщение подшаблонов

Подготовив сцену, можем начинать шоу. Сначала перенесем помощник в модуль `ApplicationHelper`, чтобы он стал доступен всем шаблонам представлений. Файл `_tiled_table.html.erb` с подшаблоном также перенесем – в каталог `app/views/shared/`, чтобы подчеркнуть, что он не связан ни с каким конкретным представлением. В интересах хорошего стиля я еще пройдусь по реализации и дам идентификаторам более общие имена. Ссылка на массив `cities` теперь будет называться `collection`, а переменная блока `city` – `item`. Новый код подшаблона представлен в листинге 11.11.

Листинг 11.11. Код подшаблона изразца с измененными именами

```
1  <table class="tiles">
2   <% collection.in_groups_of(columns) do |row| -%>
3    <tr>
```

```
4    <% row.each do |item| -%>
5     <td id="<%= dom_id(item) %>">
6      <div class="left">
7       <%= image_tag(item.main_photo.public_filename(:thumb)) %>
8      </div>
9      <div class="right">
10      <div class="title"><%= item.name %></div>
11      <div class="description"><%= item.description %></div>
12     </div>
13    </td>
14   <% end # row.each -%>
15   </tr>
16  <% end # in_groups_of -%>
17 </table>
```

Еще остался вопрос о контракте между этим кодом подшаблона и объектами, которые он выводит. Объекты обязаны отвечать на сообщения `main_photo`, `name` и `description`. Критический анализ других моделей в приложении показал, что мне необходима бóльшая гибкость. У одних сущностей есть имена, у других – заголовки. Иногда необходимо, чтобы под именем представленного объекта располагалось его описание, а иногда требуется вставить дополнительные данные об объекте плюс некоторые ссылки.

Последний штрих: лямбда-выражение

Ruby позволяет сохранять ссылки на анонимные методы (они называются также *Proc-объектами*, или *лямбда-выражениями*) и вызывать их в любой момент времени[7]. И что нам дает эта возможность? Для начала можно воспользоваться лямбда-выражением, чтобы передать блок кода, который динамически сформирует части подшаблона.

Например, сейчас написать код показа уменьшенных изображений проблематично. Этот код сильно зависит от обрабатываемого объекта, а я хотел бы передавать инструкции о получении миниатюры, не прибегая к огромному предложению `if/else` и не внося в модели логику, относящуюся к уровню представления. Сделайте небольшую паузу и осмыслите только что описанную проблему, а потом посмотрите, как она решена в листинге 11.12. Подсказка: в переменных `thumbnail`, `link`, `title` и `description` хранятся лямбда-выражения!

Листинг 11.12. Подшаблон изразца, переработанный с использованием лямбда-выражений

```
1 <div class="left">
2   <%= link_to thumbnail.call(item), link.call(item) %>
3 </div>
4 <div class="right">
5   <div class="title">
6     <%= link_to title.call(item), link.call(item) %>
7   </div>
```

```
8   <div class="description"><%= description.call(item) %></div>
9 </div>
```

Отметим, что содержимое левого и правого DIV берется из переменных, содержащих лямбда-выражения. В строке 2 мы вызываем метод link_to, причем оба его аргумента вычисляются динамически. Аналогичная конструкция в строке 6 обеспечивает порождение ссылки title. В обоих случаях первое лямбда-выражение должно вернуть результат обращения к методу image_tag, а второе – URL. Переменная item всюду содержит текущий выводимый объект, который передается лямбда-выражению как переменная блока.

> **Говорит Уилсон...**
>
> Вместо link.call(item) можно было бы написать link[item]. Это выглядит еще круче, только есть опасность свихнуться. (Proc#[] – синоним Proc#call.)

Новый метод-помощник Tiled

Если теперь вы посмотрите на листинг 11.13, то увидите, что метод tiled претерпел существенные изменения. Чтобы список позиционных аргументов не слишком разрастался, я решил передавать последним параметром хеш опций options. Это полезный подход, аналогичный принятому во всех стандартных помощниках Rails.

Одна из опций, :link, уникальна для каждого передаваемого объекта, и значения по умолчанию у нее нет, поэтому я проверяю наличие данной опции в строке 3. Для всех остальных параметров имеются значения по умолчанию, и они передаются методу render :partial в хеше :locals.

Листинг 11.13. Метод-помощник Tiled с лямбда-выражениями в качестве параметров

```
1 module ApplicationHelper

2 def tiled(collection, opts={})
3   raise 'link option is required' unless opts[:link]

4   opts[:columns] ||= 3

5   opts[:thumbnail] ||= lambda do |item|
6    image_tag(item.photo.public_filename(:thumb))
7   end

8   opts[:title] ||= lambda {|item| item.to_s }
9   opts[:description] ||= lambda {|item| item.description }
```

```
10  render :partial => "shared/tiled_table",
11         :locals => { :collection => collection,
12                      :columns => opts[:columns] || 3,
13                      :thumbnail => opts[:thumbnail],
14                      :title => opts[:title],
15                      :description => opts[:description] }
16  end
17 end
```

И, чтобы завершить пример, покажем, как вызывается новый помощник `tiled` из шаблона:

```
<%= tiled(@cities, :link => lambda {|city| city_path(city)}) %>
```

Метод `city_path` доступен блоку лямбда-выражения, поскольку это *замыкание*, наследующее контекст исполнения, в котором создано.

Заключение

Это была длинная глава, которую можно использовать в качестве подробного справочника по методам-помощникам, предоставляемым Rails, а также как источник идей для написания собственных. Эффективное применение помощников позволяет создавать элегантные шаблоны, удобные для сопровождения.

Прежде чем завершить рассмотрение ActionPack (это название объединяет `ActionController` и `ActionView`), мы отправимся в путешествие по миру Ajax и Javascript. Пожалуй, одной из основных причин популярности Rails является поддержка двух этих ключевых для Web 2.0 технологий.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-137-0, название «Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

12

Ajax on Rails

Ajax – это не технология, а сплав нескольких технологий, которые полезны сами по себе, но в сочетании открывают совершенно новые возможности.

Джесси Дж. Гарретт, изобретатель термина

Акроним Ajax расшифровывается как *Asynchronous JavaScript and XML* (асинхронный JavaScript и XML). Он охватывает технологии, которые позволяют оживить веб-страницы за счет действий, происходящих вне нормального жизненного цикла HTTP-запроса (без полного обновления страницы).

Вот некоторые применения техники Ajax:

- асинхронная отправка данных формы;
- непрерывная навигация по представленным в веб картам, как, например, на сайте Google Maps;
- динамическое обновление списков и таблиц, как в Gmail и других почтовых веб-сервисах;
- электронные таблицы в Сети;
- формы, допускающие редактирование «на месте»;
- немедленный просмотр отформатированного текста.

Идея Ajax стала возможной благодаря API XMLHttpRequestObject (XHR), реализованному во всех современных броузерах. Он позволяет JavaScript-сценарию на стороне броузера обмениваться данными с сер-

вером и использовать полученные данные для изменения пользовательского интерфейса приложения «на лету», без полного обновления страницы. Написать код, который напрямую работает с XHR во всех броузерах, мягко говоря, нелегко. Именно поэтому развелось так много библиотек с открытыми исходными текстами, которые поддерживают Ajax.

Кстати, Ajax, особенно в Rails, имеет очень мало общего с XML, несмотря на присутствие буквы X в акрониме. Полезная нагрузка асинхронных запросов и ответов от сервера может быть произвольной. Часто серверу отправляются просто параметры формы, а в ответ возвращаются фрагменты HTML, вставляемые в `DOM` страницы. Нередко сервер посылает данные, представленные в формате JavaScript Object Notation (JSON) – упрощенной разновидности языка JavaScript.

В задачу этой книги не входит изучение основ JavaScript или Ajax. Мы не будем также вдаваться в вопросы проектирования, касающиеся добавления Ajax в приложение. Это долгая и противоречивая история. Для надлежащего рассмотрения этих тем потребовалась бы целая книга, и такие книги на рынке есть. Поэтому в оставшейся части главы просто предполагается, что вы понимаете смысл технологии Ajax и причины использования ее в своих приложениях.

Ruby on Rails до предела упрощает включение Ajax в приложение благодаря изобретательной интеграции с библиотеками Prototype и Scriptaculous. В начале этой главы мы поговорим об идеологии и реализации этих JavaScript-библиотек, а потом перейдем к справочному разделу, в котором описаны методы-помощники из `ActionPack`, поддерживающие Ajax on Rails. Мы рассмотрим также имеющийся в Rails механизм RJS, позволяющий вызывать JavaScript с помощью серверного кода на Ruby.

Чтобы извлечь из этой главы максимум пользы, вы должны быть хотя бы немного знакомы с программированием на языке JavaScript.

Библиотека Prototype

Библиотеку Prototype (находится по адресу *http://prototype.conio.net*) написал и активно сопровождает Сэм Стефенсон (Sam Stephenson) – участник команды разработчиков ядра Rails. Автор описывает эту библиотеку как «уникальный, простой в употреблении инструментарий для разработки на базе классов» и «самую лучшую библиотеку для поддержки Ajax».

Библиотека Prototype входит в дистрибутив Ruby on Rails и копируется во все вновь создаваемые проекты под именем `public/javascripts/prototype.js`. Она насчитывает примерно 2000 строк кода на языке JavaScript и закладывает фундамент для организации любых видов Ajax-

взаимодействий с сервером и программирования визуальных эффектов на стороне клиента. Фактически, несмотря на тесную связь с Ruby on Rails, библиотека Prototype исключительно полезна и сама по себе.

Подключаемый модуль FireBug

FireBug[1] – это чрезвычайно мощное расширение для броузера Firefox, которое обязательно должны поставить все желающие разрабатывать Ajax-приложения. Оно позволяет инспектировать Ajax-запросы, детально исследовать DOM страницы и даже изменять на лету элементы и CSS-стили, причем эффект изменений сразу же показывается в окне броузера. Кроме того, это еще и «могучий» отладчик JavaScript, кото-

Рис. 12.1. *FireBug – необходимая вещь для разработчиков Ajax-приложений*

рый дает возможность задавать наблюдаемые выражения и устанавливать точки прерывания (рис. 12.1).

Встроенный в FireBug инспектор DOM можно использовать для исследования поведения библиотеки Prototype во время выполнения на странице броузера. FireBug обладает также интерактивной консолью, позволяющей экспериментировать с JavaScript в броузере точно так же, как irb делает это с Ruby.

Некоторые примеры кода данной главы скопированы из консоли FireBug, которая выдает приглашение >>>. Так, при инспектировании объекта Prototype в консоли получается следующий результат:

```
>>> Prototype
Object Version=1.5.0_rc2 BrowserFeatures=Object
```

Рассказывая об Ajax on Rails, я в шутку наставлял своих студентов: «Даже если вы не слышите больше ничего из того, что я говорю, используйте FireBug! Повышение вашей продуктивности очень быстро окупит затраты на мой гонорар».

---

[1] Для установки подключаемого к Firefox модуля FireBug необходимо зайти на сайт *http://www.getfirebug.com/*.

Prototype API

Понимать принципы устройства и функционирования Prototype API для работы с Ajax on Rails необязательно, но это будет весьма полезно, когда вы захотите выйти за пределы простых примеров и начнете писать собственные функции на JavaScript.

Значительная часть кода в файле `prototype.js` посвящена определению нетривиальных объектно-ориентированных языковых конструкций сверх уже имеющихся в JavaScript. Например, функция `extend` открывает дорогу к наследованию. Многие части библиотеки Prototype покажутся программистам на Ruby удивительно знакомыми, например метод `inspect` класса `Object` и метод `gsub` класса `String`. Поскольку в JavaScript функции работают как *замыкания* аналогично блокам Ruby, то для работы с массивами, манипуляций с итераторами и во многих других аспектах API Prototype берет Ruby за образец.

В общем, по духу код Prototype очень близок к Ruby, что позволяет знатокам Ruby и Rails комфортно себя чувствовать и продуктивно работать. Вы даже можете полюбить язык JavaScript (если этого еще не произошло), который, несмотря на изначальную непритязательность и дурную репутацию, на самом деле является чрезвычайно мощным и выразительным языком программирования. Не обращайте внимания на повсеместно встречающееся ключевое слово function, в конце концов оно просто станет неразличимой деталью фона.

Функции верхнего уровня

Следующие функции определены в контексте Prototype верхнего уровня.

$(id[, id2...])

Функция `$` – это одновременно сокращенная запись *и* расширение одной из самых употребительных функций при программировании на JavaScript в броузере: `document.getElementByID`. Поскольку данная функция используется очень часто, для нее выбрано предельно короткое имя, что согласуется с одним из основных принципов проектирования эффективного API.

Функции `$()` можно передать одну или несколько строк, а в ответ она вернет либо один соответствующий элемент, либо массив элементов в предположении, что на странице есть элементы с указанными атрибутами ID. Для удобства функция `$` не возбуждает исключения, когда ей передан экземпляр элемента, а не строка. Она просто возвращает этот же элемент или добавляет его в результирующий массив.

Если элемента с указанным ID не существует, для него возвращается значение `undefined`, что согласуется с поведением исходной функции `document.getElementByID`. Попытка получить более одного элемента с од-

ним и тем же ID, скорее всего, успеха иметь не будет, хотя это зависит от реализации конкретного броузера. Лучше формировать разметку правильно, чтобы элементов с одинаковыми идентификаторами не было.

$$(expr[, expr2...])

Функция `$$` принимает один или несколько CSS-селекторов и возвращает массив соответствующих элементов `DOM`. Способность искать элементы по CSS-селекторам – одна из наиболее важных особенностей Prototype.

$A(var)

Функция `$A` – синоним `Array.from`. Она преобразует свой параметр в объект `Array`, включающий функции объекта `Enumerable` (см. раздел «Объект Enumerable» ниже в этой главе).

Внутри Prototype функция `$A` используется главным образом для преобразования списков аргументов и узлов `DOM` в массивы. Отметим, что в последних версиях Prototype функции объекта Enumerable примешиваются прямо к встроенному в JavaScript объекту `Array`, поэтому пользы от функции уже немного.

$F(id)

Функция `$F` – синоним `Form.Element.getValue`. Она возвращает значение поля формы с заданным ID. Это полезный вспомогательный метод, поскольку работает он вне зависимости о того, является ли запрошенное поле текстовым, списком select или текстовой областью (TEXTAREA).

$H(obj)

Функция `$H` расширяет простой объект JavaScript из объектов `Enumerable` и `Hash`, делая его похожим на хеш в понимании Ruby (см. раздел «Объект Hash» ниже в этой главе).

$R(start, end, exclusive)

Функция `$R` – сокращенная запись конструктора `ObjectRange` (см. раздел «Объект ObjectRange» ниже в этой главе).

Try.these(func1, func2[, func3...]

Строго говоря, это не функция верхнего уровня, но мне показалось удобным включить ее в этот раздел, так как `these` – единственная функция объекта `Try`.

При выполнении операций, которые по-разному реализованы в разных броузерах, часто приходится пробовать различные способы, пока не найдется подходящий. В объекте `Try` определена функция `these`; ей пе-

редается список функций, которые выполняются поочередно, пока не найдется функция, *не возбуждающая* исключение.

Классический пример, взятый из кода самой библиотеки Prototype, – способ получения ссылки на объект `XMLHttpRequest`, который существенно различен для Firefox и Internet Explorer:

```
var Ajax = {
  getTransport: function() {
    return Try.these(
      function() {return new XMLHttpRequest()},
      function() {return new ActiveXObject('Msxml2.XMLHTTP')},
      function() {return new ActiveXObject('Microsoft.XMLHTTP')}
    ) || false;
  },

  activeRequestCount: 0
}
```

Объект Class

В объекте `Class` определена функция `create`, применяемая для объявления новых экземпляров Ruby-подобных классов. Далее эти классы могут объявить функцию `initialize`, которая будет выступать в роли конструктора при вызове `new` для создания нового экземпляра.

Ниже в качестве примера приведена реализация объекта `ObjectRange`:

```
ObjectRange = Class.create();
Object.extend(ObjectRange.prototype, Enumerable);
Object.extend(ObjectRange.prototype, {
  initialize: function(start, end, exclusive) {
    this.start = start;
    this.end = end;
    this.exclusive = exclusive;
  },

  ...
});

var $R = function(start, end, exclusive) {
  return new ObjectRange(start, end, exclusive);
}
```

Сначала для `ObjectRange` создается класс (который будет вести себя похоже на `class ObjectRange` в Ruby). Затем объект `prototype` объекта `ObjectRange` расширяется с целью добавить методы экземпляра. К нему подмешиваются функции объекта `Enumerable`, а затем – функции анонимного JavaScript-объекта, определенного с помощью фигурных скобок, внутри которых находится функция `initialize` и прочие методы экземпляра.

Расширения класса JavaScript Object

Одна из причин, по которым код, написанный с применением Prototype, может выглядеть так чисто и кратко, – тот факт, что функции подмешиваются непосредственно в базовые классы JavaScript (в частности `Object`).

Object.clone(object)

Возвращает копию объекта `object`, переданного в качестве параметра. Делается это путем использования объекта-параметра для расширения нового экземпляра `Object`:

```
clone: function(object) {
  return Object.extend({}, object);
}
```

Object.extend(destination, source)

Статическая функция `extend` в цикле перебирает все свойства переданного объекта `source`, включая функции, и копирует их в объект `destination`. Тем самым она служит основой механизмов наследования и клонирования (в языке JavaScript нет встроенной поддержки наследования).

Исходный код настолько поучителен и прост, что я решил включить его целиком:

```
Object.extend = function(destination, source) {
  for (var property in source) {
    destination[property] = source[property];
  }
  return destination;
}
```

Object.keys(obj) и Object.values(obj)

Объекты в JavaScript ведут себя почти так же, как ассоциативные массивы (или хеши) в других языках, и повсеместно используются в таком духе. Статическая функция `keys` возвращает список свойств, определенных в объекте. Статическая функция `values` возвращает список значений свойств.

Object.inspect(param)

Если параметр не определен (в JavaScript это не соответствует равенству `null`), статическая функция `inspect` возвращает строку `'undefined'`. Если же параметр равен `null`, возвращается строка `'null'`. Если в объекте-параметре определена функция `inspect()`, она вызывается и возвращается ее результат. В противном случае вызывается функция `toString()`.

Расширения класса JavaScript Array

Помимо определенных в объекте `Enumerable`, для массивов доступны также методы, перечисленные ниже.

array.clear()

Удаляет из массива все элементы и возвращает его. Интересно, что реализация этого метода просто устанавливает длину массива равной нулю:

```
clear: function() {
  this.length = 0;
  return this;
}
```

array.compact()

Удаляет из массива все элементы, равные `null` и `undefined`, и возвращает его. Обратите внимание на употребление функции `select` в реализации:

```
compact: function() {
  return this.select(function(value) {
    return value != undefined || value != null;
  });
}
```

array.first() и array.last()

Возвращают первый и последний элементы массива соответственно.

array.flatten()

Принимает массив и рекурсивно «разглаживает» его, копируя элементы в новый массив, который и возвращает. Иными словами, функция перебирает все элементы исходного массива и для каждого элемента, который сам является массивом, копирует его элементы в возвращаемый массив. Обратите внимание на употребление функции `inject` в реализации:

```
flatten: function() {
  return this.inject([], function(array, value) {
    return array.concat(value && value.constructor == Array ?
      value.flatten() : [value]);
  });
}
```

array.indexOf(object)

Возвращает индекс элемента `object` в массиве или `-1`, если элемент не найден:

```
indexOf: function(object) {
  for (var i = 0; i < this.length; i++)
    if (this[i] == object) return i;
    return -1;
  });
}
```

array.inspect()

Переопределяет функцию `inspect` из объекта `Object`, так что она печатает элементы массива через запятую:

```
indexOf: function() {
  return '[' + this.map(Object.inspect).join(', ') + ']';
}
```

array.reverse(inline)

Изменяет порядок элементов в массиве на противоположный. Если аргумент `inline` равен `true` (это значение подразумевается по умолчанию), модифицируется исходный массив, в противном случае он остается неизменным, и возвращается копия.

array.shift()

Удаляет из массива последний элемент и возвращает его. В результате размер массива уменьшается на 1.

array.without(obj1[, obj2, ...])

Удаляет из массива элементы, перечисленные в аргументах. Принимает множество удаляемых элементов в виде массива или списка, причем единообразие достигается за счет применения к аргументам функции `$A`. Обратите внимание на употребление функции `select` в реализации:

```
without: function() {
  var values = $A(arguments);
  return this.select(function(value) {
    return !values.include(value);
  });
}
```

Расширения объекта document

Метод `document.getElementsByClassName(className [, parentElement])` возвращает список элементов `DOM`, для которых имя CSS-класса равно `className`. Необязательный параметр `parentElement` позволяет ограничить поиск конкретной ветвью `DOM`, а не просматривать весь документ, начиная с элемента `body` (режим по умолчанию).

Расширения класса Event

Для удобства в класс Event добавлены следующие константы:

```
Object.extend(Event, {
  KEY_BACKSPACE: 8,
  KEY_TAB:       9,
  KEY_RETURN:   13,
  KEY_ESC:      27,
  KEY_LEFT:     37,
  KEY_UP:       38,
  KEY_RIGHT:    39,
  KEY_DOWN:     40,
  KEY_DELETE:   46
});
```

Их наличие упрощает программирование обработчиков событий клавиатуры. В следующем примере мы применяем в обработчике события onKeyPress предложение switch, чтобы проверить, не нажал ли пользователь клавишу Escape.

```
onKeyPress: function(event) {
  switch(event.keyCode) {
    case Event.KEY_ESC:
      alert('Отменено');
      Event.stop(event);
  }
}
```

Event.element()

Возвращает элемент, являющийся источником события.

Event.findElement(event, tagName)

Обходит дерево DOM снизу вверх, начиная с элемента – источника события. Возвращает первый встретившийся элемент, для которого имя тега равно tagName (без учета регистра). Если подходящих элементов не найдено, функция возвращает сам элемент-источник, а не завершается с ошибкой, что может приводить к некоторой путанице.

Event.isLeftClick(event)

Возвращает true, если событие вызвано щелчком левой кнопкой мыши.

Event.observe(element, name, observer, useCapture) и Event.stopObserving(element, name, observer, useCapture)

Функция observe обертывает встроенную в броузер функцию addEventListener, включенную в спецификацию DOM Level 2. Она устанавлива-

ет отношение «наблюдаемый-наблюдатель» между указанным элементом element и функцией observer. Параметр element может быть как строковым идентификатором ID, так и самим элементом. В случае событий мыши и клавиатуры в этом качестве часто выступает элемент document.

Функция stopObserving, обертывающая встроенный метод DOM removeEventListener, разрывает связь между элементом и обработчиком события.

Параметр name должен быть названием события (в виде строки), определенного в спецификации DOM для броузеров (blur, click и т. д.)[1].

Параметр observer должен быть *ссылкой* на функцию, то есть именем функции без скобок (частый источник путаницы!). Почти всегда в сочетании с observe используется функция bindAsEventListener, чтобы обработчик события исполнялся в правильном контексте (см. ниже раздел «Расширения класса JavaScript Function»).

Необязательный параметр useCapture позволяет указать, что обработчик следует вызывать на фазе погружения (capture), а не всплытия (bubbling), по умолчанию он равен false.

Следующий пример взят из реализации объекта AutoCompleter, входящего в библиотеку Scriptaculous:

```
addObservers: function(element) {
  Event.observe(element, "mouseover",
this.onHover.bindAsEventListener(this));
  Event.observe(element, "click",
this.onClick.bindAsEventListener(this));
}
```

Event.pointerX(event) and Event.pointerY(event)

Возвращает координаты *x* и *y* курсора мыши в момент возникновения события.

Event.stop(event)

Останавливает распространение события и отменяет поведение по умолчанию, как бы оно ни было определено.

Расширения класса JavaScript Function

Следующие две функции примешиваются к встроенному классу Function.

[1] Исчерпывающее описание событий DOM и способов работы с ними см. на странице *http://www.quirksmode.org/dom/w3c_events.html*.

function.bind(obj)

Используется для привязывания функции к контексту объекта, переданного в качестве параметра. Почти всегда этот параметр равен this, то есть привязка происходит к контексту текущего объекта, поскольку основное назначение bind – гарантировать, что функция, определенная в каком-то другом месте, будет исполняться в том контексте, где вы сейчас находитесь.

Вот, например, как реализована функция registerCallback в объекте PeriodicalExecuter:

```
registerCallback: function() {
  setInterval(this.onTimerEvent.bind(this), this.frequency * 1000);
}
```

Необходимо привязать функцию onTimerEvent к контексту объекта, для которого регистрируется обратный вызов, а не к объекту-прототипу самого PeriodicalExecuter.

Концепцию привязки нелегко усвоить, если вы не являетесь опытным программистом на JavaScript и не имеете склонности к функциональному программированию, поэтому не переживайте, если с первого раза вам не удалось ее понять.

function.bindAsEventListener(obj)

Применяется для *присоединения* данной функции в качестве обработчика события DOM таким образом, что объект event будет передан ей как параметр. Используйте так же, как bind, если хотите гарантировать, что некоторый метод будет выполняться в контексте конкретного экземпляра, а не прототипа класса, в котором определен. В самой библиотеке Prototype этот метод не используется, но широко применяется в библиотеке Scriptaculous и в JavaScript-приложениях, когда необходимо создать класс-наблюдатель, содержащий обработчики событий, которые привязаны к элементам страницы.

В следующем примере приведен код класса, призванного извещать об изменениях в некотором поле ввода, выдавая настраиваемое сообщение:

```
var InputObserver = Class.create();
InputObserver.prototype = {
  initialize: function(input, message) {
    this.input = $(input);
    this.message = message;
    this.input.onchange = this.alertMessage.bindAsEventListener(this);
  },

  alertMessage: function(e) {
    alert(this.message + ' (' + e.type + ')');
  }
};

var o = new InputObserver('id_поля ввода', 'Поле ввода');
```

Расширения класса JavaScript Number

Следующие функции примешаны к встроенному классу `Number`.

number.toColorPart()

Возвращает шестнадцатеричное представление целочисленного RGB-кода цвета:

```
toColorPart: function() {
  var digits = this.toString(16);
  if (this < 16) return '0' + digits;
  return digits;
},
```

Напомню, что числа в JavaScript автоматически не «*обертываются в объект*». Вы должны присвоить числовое значение переменной, самостоятельно обернуть число в экземпляр класса `Number` или просто заключить его в круглые скобки – только тогда можно будет вызывать для него методы. Не убедил? Можете проверить сами в консоли FireBug:

```
>>> 12.toColorPart();
missing ; before statement 12.toColorPart();
>>> n = new Number(12)
12
>>> n.toColorPart();
"0c"
>>> n = 12
12
>>> n.toColorPart();
"0c"
>>> (12).toColorPart();
"0c"
>>> 12.toColorPart
missing ; before statement 12.toColorPart
>>> (12).toColorPart
function()
```

number.succ()

Возвращает следующее по порядку число:

```
succ: function() {
  return this + 1;
},
```

number.times()

Подобно методу `times`, который существует в Ruby для числовых объектов, метод `number.times()` принимает блок кода и вызывает его `number` раз (соответствующих значению числа, для которого вызван метод). Обратите внимание на использование функции `$R`, которая позволяет

легко создать диапазон, и функции each для вызова указанной функции iterator:

```
times: function(iterator) {
  $R(0, this, true).each(iterator);
  return this;
}
```

Ниже приведен простой пример, в котором пять раз вызывается функция alert. Напомню, что вызывать JavaScript-функцию напрямую для *необернутого* числа нельзя, поскольку синтаксический анализатор ожидает скобки:

```
>>> (5).times(new Function("alert('yeah')"))
5
```

Расширения класса JavaScript String

Следующие функции примешаны к встроенному классу String.

string.camelize()

Преобразует строки с разделителями-дефисами к виду lowerCamelCase:

```
>>> "about-to-be-camelized".camelize()
"aboutToBeCamelized"
```

string.dasherize()

Преобразует строки с разделителями-подчерками в строки с разделителями-дефисами:

```
>>> "about_to_be_dasherized".dasherize()
"about-to-be-dasherized"
```

string.escapeHTML() и string.unescapeHTML()

Функция экземпляра escapeHTML экранирует HTML и XML-разметку в строке, преобразуя угловые скобки в соответствующие компоненты:

```
>>> '<script src="http://evil.org/bad.js"/>'.escapeHTML()
"&lt;script src="http://evil.org/bad.js"/&gt;"
```

Функция unescapeHTML выполняет обратную операцию.

string.evalScripts() и string.extractScripts()

Функция экземпляра evalScripts выполняет содержимое всех тегов <script>, встречающихся в строке.

Функция экземпляра extractScripts возвращает массив строк с содержимым тегов <script>, встречающихся в строке. Отметим, что сами открывающие и закрывающие теги <script> не включаются, извлекается только JavaScript-код.

string.gsub(pattern, replacement) и string.sub (pattern, replacement, count)

Функция экземпляра `gsub` возвращает *копию* строки, в которой все вхождения образца `pattern` заменены строкой `replacement`. Исходная строка не модифицируется. Образец должен быть литеральным регулярным выражением JavaScript, заключенным между символами «/».

Функция `sub` аналогична `gsub`, но выполняет не более `count` замен, причем по умолчанию `count` равно 1.

string.scan(pattern, iterator)

Функция экземпляра `scan` очень похожа на `gsub`, но вместо строки замены принимает итератор.

string.strip()

Функция экземпляра `strip` удаляет начальные и хвостовые пробелы. Обратите внимание на сцепленные вызовы `replace` в реализации:

```
strip: function() {
  return this.replace(/^\s+/, '').replace(/\s+$/, '');
}
```

string.stripScripts() и string.stripTags()

Функция экземпляра `stripScripts` удаляет из строки все теги `<script>` (вместе с содержимым), а функция экземпляра `stripTags` – все HTML- и XML-теги.

string.parseQuery() и string.toQueryParams()

Обе функции преобразуют строку запроса (в формате, принятом в URL) в объект JavaScript:

```
>>> "?foo=bar&da=da+do+la".toQueryParams()
Object foo=bar da=da+do+la
```

string.toArray()

Возвращает массив символов, составляющих строку.

string.truncate(length, truncationString)

Работает, как метод `truncate`, который Rails подмешивает к строкам. Если длина строки больше `length`, она усекается, и в конец добавляется строка `truncationString` (по умолчанию равная `"..."`).

```
>>> "Mary had a little lamb".truncate(14)
"Mary had a ..."
```

string.underscore()

Прямой перенос метода `underscore`, который Rails подмешивает к строкам. Преобразует строки, записанные в верблюжьейНотации в строки с разделителями-подчерками. Изменяет `::` на `/` с целью преобразования пространств имен Ruby в пути:

```
>>> "ActiveRecord::Foo::BarCamp".underscore()
"active_record/foo/bar_camp"
```

Объект Ajax

Объект `Ajax` сам по себе обладает полезным поведением, а также служит корневым пространством имен для других относящихся к Ajax объектов в библиотеке Prototype.

Ajax.activeRequestCount

Содержит количество исполняемых в данный момент Ajax-запросов. Поскольку они выполняются асинхронно, это значение может быть больше единицы. Используется для реализации индикаторов активности – маленьких анимированных картинок, извещающих пользователя о том, что сейчас происходит обращение к серверу:

```
Ajax.Responders.register({
  onCreate: function() {
    if($('busy') && Ajax.activeRequestCount > 0)
      Effect.Appear('busy', { duration:0.5, queue:'end' });
  },

  onComplete: function() {
    if($('busy') && Ajax.activeRequestCount == 0)
    Effect.Fade('busy', {duration:0.5, queue:'end' });
  }
});
```

Ajax.getTransport()

Возвращает ссылку на объект `XMLHttpRequestObject`, предоставляемый броузером. Обычно вам не нужно обращаться к этой функции самостоятельно – ее вызывают другие функции, относящиеся к Ajax.

Объект Ajax.Responders

Объект `Responders` управляет списком обработчиков, заинтересованных в получении извещений о событиях, которые касаются Ajax. В предыдущем примере показано, как `Ajax.Responders` используется для регистрации двух функций обратного вызова, `onCreate` и `onComplete`, которые занимаются показом и сокрытием графического изображения, инди-

цирующего активность Ajax. Помимо статических функций, описанных в последующих разделах, объект Ajax.Responders еще и заимствует функции из Enumerable.

Ajax.Responders.register(responder)

Добавляет объекты responder в список зарегистрированных обработчиков, заинтересованных в получении событий, касающихся Ajax. Обработчики вызываются в порядке регистрации и должны реализовывать хотя бы один из следующих обратных вызовов Ajax: onCreate, onComplete или onException.

Ajax.Responders.unregister(responder)

Удаляет объект responder из списка зарегистрированных обработчиков.

Объект Enumerable

Объект Enumerable используется так же, как модуль Ruby Enumerable, применяемый в качестве примеси. Необходимо, чтобы в объекте, к которому примешивается Enumerable, была определена функция each. Библиотека Prototype подмешивает Enumerable во многие объекты, в том числе Array, Hash, ObjectRange, Ajax.Responders и Element.ClassNames.

Точно так же, как в Ruby, можно подмешивать Enumerable в собственные JavaScript-классы. Достаточно лишь предоставить реализацию функции _each:

```
// Класс, в котором реализована функция _each
var MyCustomClass = Class.create();
MyCustomClass.prototype = {
  _each: function(iterator) {
     for (var i = 0, length = this.length; i < length; i++) {
       iterator(this[i]);
     }
  }
}

// Подмешиваем функции Enumerable
Object.extend(MyCustomClass.prototype, Enumerable);
```

Язык JavaScript не поддерживает закрытых и защищенных функций, поэтому в библиотеке Prototype имена функций, не предназначенных для открытого использования, начинаются со знака подчерка.

Дизайн объекта Enumerable отличается от принятого в Ruby тем, что предоставляет вам открытую функцию each, которая внутри пользуется функцией _each. Но, отвлекшись от этого, вы заметите, что большинство итераторных функций, определенных в Enumerable, очень похожи на свои аналоги в Ruby.

enumerable.each(iterator)

Функция each принимает ссылку на функцию в параметре iterator и вызывает ее для каждого перебираемого элемента. Текущий элемент передается итераторной функции в качестве параметра.

В следующем простом примере три раза вызывается встроенная функция alert:

```
function alerter(msg) {
  alert(msg);
}

["foo", "bar", "baz"].each(alerter)
```

Чтобы в JavaScript передать ссылку на функцию, достаточно указать ее имя, опустив круглые скобки.

Ниже перечислены остальные итераторные функции. Большинство итераторов вызывается с двумя параметрами: value и index.

enumerable.all(iterator)

Функция all передает каждый элемент объекта Enumerable итератору и возвращает true, если функция iterator ни разу не вернула false. Если параметр iterator опущен, то каждый элемент просто вычисляется в булевом контексте. Можно считать, что функция all – это одна большая булева операция AND.

enumerable.any(iterator)

Функция any передает каждый элемент объекта Enumerable итератору и возвращает true, если функция iterator хотя бы один раз вернула true. Если параметр iterator опущен, каждый элемент просто вычисляется в булевом контексте. Можно считать, что функция any – это одна большая булева операция OR.

enumerable.collect(iterator) и enumerable.map(iterator)

Функция collect (у нее имеется синоним map) возвращает результаты применения функции iterator к каждому элементу объекта Enumerable:

```
>>> $R(1,4).collect(Prototype.K) // K возвращает переданный ей аргумент
[1, 2, 3, 4]
>>> $R(1,4).collect(function(){return "cat"})
["cat", "cat", "cat", "cat"]
```

enumerable.detect(iterator) и enumerable.find(iterator)

Функция detect (у нее есть синоним find) используется для нахождения первого элемента объекта Enumerable, который отвечает критерию, определяемому функцией iterator:

```
>>> $R(1,100).detect(function(i){ return i % 5 == 0 && i % 6 == 0 })
30
```

enumerable.eachSlice(number[, iterator])

Функция `eachSlice` разбивает элементы массива на `number` отрезков. Затем она возвращает результаты применения функции `collect` с необязательной функцией `iterator` в качестве параметра к получившемуся списку отрезков, что по сути дела «разглаживает» результат в одномерный массив:

```
>>> $R(1,10).eachSlice(5)
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]

>>> $R(1,10).eachSlice(2, function(slice) { return slice.first() })
[1, 3, 5, 7, 9]
```

enumerable.findAll(iterator) и enumerable.select (iterator)

Функция `findAll` (у нее есть синоним `select`) используется для нахождения всех элементов объекта `Enumerable`, которые отвечают критерию, определяемому функцией `iterator`:

```
>>> $R(1,100).findAll(function(i){ return i % 5 == 0 && i % 6 == 0 })
[30, 60, 90]
```

enumerable.grep(pattern[, iterator])

Функция `grep` возвращает все элементы объекта `Enumerable`, которые сопоставляются с регулярным выражением, заданным в параметре `pattern`.

Необязательная функция `iterator` вызывается для всех сопоставившихся элементов:

```
>>> quote = "The truth does not change according to our ability to
stomach it"
"The truth does not change according to our ability to stomach it"

>>> quote.split(' ').grep(/\w{5}/)
["truth", "change", "according", "ability", "stomach"]

>>> quote.split(' ').grep(/\w{5}/, function(val, i){ return i + ":" +
val })
["1:truth", "4:change", "5:according", "8:ability", "10:stomach"]
```

enumerable.include(obj) и enumerable.member(obj)

Функция `include` (у нее есть синоним `member`) возвращает `true`, если хотя бы один элемент объекта `Enumerable` равен параметру `obj`. Сравнение производится с помощью оператора `==`:

```
>>> ['a','b','c'].include('a')
true
```

```
>>> ['a','b','c'].include('x')
false
```

enumerable.inGroupsOf(num[, filler])

Функция `inGroupsOf` похожа на `eachSlice`, но не принимает итератор. Она всегда возвращает двухмерный массив, содержащий группы одинакового размера, которые состоят из элементов объекта `Enumerable`. Необязательный параметр `filler` позволяет задать значение для заполнения недостающих элементов в последней группе и по умолчанию равен `null`:

```
>>> $R(1,10).inGroupsOf(3)
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, null, null]]
>>> $R(1,10).inGroupsOf(3, 0) // дополнить нулями
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 0, 0]]
```

enumerable.inject(accumulator, iterator)

Функция `inject` объединяет элементы объекта `Enumerable`, применяя функцию `iterator` к объекту `accumulator` и каждому перебираемому элементу по очереди. На каждой итерации `accumulator` присваивается значение, возвращенное функцией `iterator`. В отличие от аналогичной функции Ruby вариант `inject` из библиотеки Prototype требует, чтобы объекту `accumulator` было присвоено начальное значение:

```
>>> $R(1,5).inject(0, function(acc, e) { return acc + e })
15
```

enumerable.invoke(functionName[, arg1, arg2...])

Функция `invoke` вызывает функцию с именем `functionName` для каждого элемента объекта `Enumerable`. Вызываемой функции передаются необязательные параметры `arg1`, `arg2` и т. д.:

```
>>> $$('.details li').invoke('hide')
```

enumerable.max([iterator]) and enumerable.min([iterator])

Функции `max` и `min` очень похожи. Они возвращают элементы объекта `Enumerable` с максимальным и минимальным значением соответственно. Если задана необязательная функция `iterator`, то она применяется для преобразования значения каждого элемента перед сравнением:

```
>>> $R(1,5).min()
1
>>> ["1","2","3"].max(function(val) { return Number(val) })
3
```

enumerable.partition([iterator])

Функция `partition` возвращает массив из двух элементов. Первый элемент – это массив, содержащий элементы объекта `Enumerable`, для которых необязательная функция `iterator` вернула `true`; второй – массив,

содержащий элементы объекта Enumerable, для которых iterator вернула false. Если функция iterator не задана, используются значения самих элементов, вычисленные в булевом контексте:

```
>>> ["1",null,"2",null,null].partition()
[["1", "2"], [null, null, null]]
```

enumerable.pluck(propertyName)

Функция pluck «выдирает» значения указанного свойства из всех элементов объекта Enumerable и возвращает их в виде массива. Это просто удобный вспомогательный метод, аналогичный collect:

```
>>> $$('script').pluck('src')
["http://localhost:3000/javascripts/prototype.js?1165877878",
"http://localhost:3000/javascripts/effects.js?1161572695",
"http://localhost:3000/javascripts/dragdrop.js?1161572695",
"http://localhost:3000/javascripts/controls.js?1161572695",
"http://localhost:3000/javascripts/application.js?1161572695", ""]
```

enumerable.reject(iterator)

Функция reject возвращает элементы объекта Enumerable, для которых обязательная функция iterator вернула false.

enumerable.sortBy(iterator)

Функция sortBy возвращает элементы объекта Enumerable, отсортированные по критерию, который определяется обязательной функцией iterator.

Кстати, составляя пример для этой функции, я понял, что по выработанной в Ruby привычке часто забываю писать return в теле функции iterator. К сожалению, не всегда это приводит к фатальной ошибке в сценарии и потому является источником проблем. Не забывайте, что в языке JavaScript возврат из функции следует производить явно с помощью return!

```
>>> linusQuote = "Software is like sex: It's better when it's free."
"Software is like sex: It's better when it's free."
>>> linusQuote.split(' ').sortBy(function(s,index) { return s.length })
["is", "it's", "sex:", "It's", "like", "when", "free.", "better",
"Software"]
```

enumerable.toArray() и enumerable.entries()

Функция toArray (у нее есть синоним entries) возвращает элементы объекта Enumerable в виде массива.

enumerable.zip(enum1, enum2[, enum3...][, iterator])

Интересная функция zip построена по образцу одноименного итератора в Ruby. К сжатию она не имеет никакого отношения; скорее, подой-

дет метафора молнии (zipper) на брюках. Функция `zip` объединяет элементы каждого из переданных объектов `Enumerable`, так что возвращаемый список имеет столько же элементов, сколько имеется в объекте, в контексте которого функция вызвана.

Если последний (необязательный) параметр является функцией, то она выступает в роли итератора и применяется к каждому элементу возвращаемого массива. Проще всего проиллюстрировать поведение функции на примере, аналогичном приведенному в книге *Programming Ruby*:

```
>>> a = [4, 5, 6]
[4, 5, 6]
>>> b = [7, 8, 9]
[7, 8, 9]
>>> [1, 2, 3].zip(a, b)
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Класс Hash

Класс `Object` в JavaScript, объекты которого можно создавать на лету с помощью фигурных скобок, очень похож на ассоциативный массив (он же хеш). Безо всяких модификаций он поддерживает синтаксис квадратных скобок для присваивания и выборки.

Библиотека Prototype предоставляет класс `Hash`, который расширяет `Enumerable` и добавляет знакомые по хешам Ruby функции.

hash.keys() и hash.values()

Функции `keys` и `values` возвращают соответственно списки ключей и значений.

hash.merge(another)

Функция `merge` объединяет хеши `another` и `hash`. Если некоторый ключ встречается в обоих хешах, значение в `hash` перезаписывается:

```
>>> $H({foo:'foo', bar:'bar'}).merge({foo:'FOO', baz:'baz'})
Object foo=FOO bar=bar baz=baz
```

hash.toQueryString()

Функция `toQueryString` форматирует пары ключ/значение из хеша в виде строки запроса для добавления в конец URL. По сравнению с ручным конструированием строки запроса это оказывается гораздо проще:

```
>>> $H(Prototype).toQueryString()
"Version=1.5.0_rc2&BrowserFeatures=%5Bobject%20Object%5D&ScriptFragment
=(%3F%3A%3Cscript.*%3F%3E)((%0A%7C%0D%7C.)*%3F)(%3F%3A%3C%2Fscript%3E)"
```

Объект ObjectRange

Объект ObjectRange обеспечивает простой способ создания диапазонов в JavaScript. У него есть конструктор, но чаще используется функция $R. В библиотеке Prototype для нахождения следующего значения в диапазоне применяется метод succ, и такой метод Prototype подмешивает в классы Number и String. Кроме того, Prototype подмешивает объект Enumerable, что делает диапазоны намного полезнее:

```
>>> $A($R(1, 5)).join(', ')
'1, 2, 3, 4, 5'
>>> $R(1, 3).zip(['Option A', 'Option B', 'Option C'], function(tuple) {
  return tuple.join(' = ');
})
['1 = Option A', '2 = Option B', '3 = Option C']
```

При использовании строковых диапазонов необходима осторожность, поскольку ObjectRange не проверяет выход за границы алфавита, а перебирает всю таблицу символов. В результате может быть создан поистине гигантский массив:

```
>>> $A($R('a', 'c'))
['a', 'b', 'c']
>>> $A($R('aa', 'ab'))
['...', 'ax', 'ay', 'az', 'a{', 'a|', 'a}', ...] // Очень большой массив
```

Объект Prototype

Объект Prototype содержит номер версии библиотеки в свойстве Version, небольшое регулярное выражение ScriptFragment для сопоставления с тегом script в HTML-разметке и две очень простые функции.

Функция emptyFunction, как и следует из названия, пуста. Функция K не имеет никакого отношения к гиперфакториалу или комплексным числам – она просто возвращает значение, которое ей было передано, и используется в Prototype для внутренних надобностей.

Модуль PrototypeHelper

Рассматривая в главе 11 помощников, мы сознательно опустили модуль PrototypeHelper и тесно связанный с ним Scriptaculous. Они обеспечивают простой способ работы с библиотеками Prototype и Scriptaculous соответственно, позволяя снабдить свое приложение функциональностью Ajax.

link_to_remote

Обсудив функциональность, предоставляемую библиотекой Prototype, мы можем теперь выполнить простой Ajax-вызов. Rails минимизирует

объем JavaScript-кода, который придется писать вручную. Мы воспользуемся одним из самых употребительных методов-помощников link_to_remote для получения случайного числа от контроллера (листинг 12.1). Контроллер написан в соответствии с принципами REST, в нем есть метод respond_to для формирования ответа клиентам в виде JavaScript-кода.

Листинг 12.1. Метод контроллера, вызываемый с применением Ajax

```
Class RandomsController < ApplicationController
  def index
  end

  def new
    respond_to do |wants|
      wants.js { render :text => rand(1_000_000) }
    end
  end
end
```

Мы создадим только представление index, поскольку метод new умеет выводить лишь текст. В этом представлении мы воспользуемся помощником link_to_remote для генерации Ajax-ссылки на метод new. Результат выполнения запроса будет помещен в тег div с идентификатором result. Параметр url определяет, куда должна вести ссылка. Метод link_to_remote принимает те же значения, что стандартный метод link_to.

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <%= link_to_remote 'Случайное число, пожалуйста', :url =>
new_random_path,
:update => 'result' %>
    <br/><br/>
    <div id="result"></div>
  </body>
</html>
```

Сгенерированная страница выглядит следующим образом:

```
<html>
  <head>
    <script src="/javascripts/prototype.js?1184547490"
            type="text/javascript"></script>
    <script src="/javascripts/effects.js?1184547490"
            type="text/javascript"></script>
    <script src="/javascripts/dragdrop.js?1184547490"
            type="text/javascript"></script>
    <script src="/javascripts/controls.js?1184547490"
            type="text/javascript"></script>
```

```
    <script src="/javascripts/application.js?1184547490"
            type="text/javascript"></script>
  </head>
  <body>
    <a href="#" onclick="new Ajax.Updater('result',
'http://localhost:3000/randoms/new', {asynchronous:true,
evalScripts:true}); return false;">Случайное число, пожалуйста</a>
    <br/>
    <br/>
    <div id="result"></div>
</body>
</html>
```

В результате обращения к методу `javascript_include_tag :defaults` были добавлены необходимые теги `script`. Rails дописывает в конец URL уникальное число, чтобы избежать проблем, связанных с кэшированием старых версий JavaScript-файлов броузером.

Помощник `link_to_remote` настраивается в широких пределах, в частности, мы можем сохранить все полученные случайные числа. Сначала необходимо изменить представление, включив в него маркированный список, и не заменять содержимое тега `ul` при каждом щелчке по Ajax-ссылке. Вместо этого мы будем добавлять результат в конец списка:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <%= link_to_remote 'Случайное число, пожалуйста', :url =>
new_random_path,
:update => 'result', :position => :bottom %>
<br/><br/>
    <ul id="result"></ul>
  </body>
</html>
```

Затем изменим контроллер, чтобы он выводил тег элемента списка:

```
Class RandomsController < ApplicationController
  ...

  def new
    respond_to do |wants|
      wants.js { render :text => "<li>#{rand(1_000_000)}</li>" }
    end
  end
end
```

Теперь при каждом щелчке по ссылке новый результат будет размещаться под последним из ранее полученных. Параметр `position` может принимать одно из четырех значений: `:before`, `:after`, `:top` и `:bottom`. Значения `:before` и `:after` относятся к элементу, а `:top` и `:bottom` – к по-

томкам элемента. Чтобы помещать последний результат в начало списка, достаточно было бы заменить :bottom на :top. Но если бы мы указали параметр :before, новые элементы оказались бы вне маркированного списка:

```
...
    <br/><br/>
    <li>15416</li>
    <li>9871</li>
    <ul id="result"></ul>
...
```

Что случится, если во время выполнения Ajax-запроса возникнет ошибка? В методе link_to_remote предусмотрен обратный вызов для обработки таких ситуаций. Для этого достаточно задать параметр :failure:

```
...
<%= link_to_remote 'Случайное число, пожалуйста', :url => new_random_path,
:update => 'result', :position => :bottom, :failure => "alert('Ошибка
HTTP ' + request.status + '!')" %>
...
```

Значением параметра :failure должна быть JavaScript-функция или фрагмент кода. Код обратного вызова имеет доступ к объекту XMLHttpRequest. В данном случае выводится код состояния, но можно было бы показать и ответ целиком, воспользовавшись свойством request.responseText. Есть и другие обратные вызовы, все они описаны в табл. 12.1.

Таблица 12.1. Обратные вызовы, доступные методу link_to_remote

Параметр	Описание
:before	Вызывается перед началом запроса
:after	Вызывается сразу после отправки запроса, но до :loading
:loading	Вызывается, когда броузер начинает загружать данные в документ
:interactive	Вызывается, когда броузер закончил загружать документ
:success	Вызывается, когда запрос XMLHttpRequest полностью обработан, и код состояния HTTP находится в диапазоне 2XX
:failure	Вызывается, когда запрос XMLHttpRequest полностью обработан, и код состояния HTTP находится не в диапазоне 2XX
:complete	Вызывается, когда запрос XMLHttpRequest окончательно обработан (после :success или :failure, если эти обратные вызовы заданы)

Если вам необходим еще более точный контроль, можете задать обратные вызовы для конкретных кодов состояния. В следующем примере

задается обратный вызов для кода 404. Отметим, что код состояния – не символ, а целое число:

```
...
<%= link_to_remote 'Случайное число, пожалуйста', :url => new_random_path,
:update => 'result', :position => :bottom, :failure => "alert('Ошибка
HTTP ' + request.status + '!')", 404 => "alert('Не найден')" %>
...
```

У метода `link_to_remote` есть ряд параметров для настройки поведения на стороне броузера. Хотя, как правило, с помощью Ajax посылаются асинхронные запросы, можно затребовать и синхронное поведение, задав параметр `:type => :synchronous`. Тогда броузер будет заблокирован до тех пор, пока не закончится обработка запроса. Можно также добавить диалоговое окно подтверждения, задав `:confirm => true`. Если необходимо выполнять Ajax-запрос условно, передайте в параметре `:condition` JavaScript-выражение, истинность которого будет вычислять броузер.

remote_form_for

Как методу `link_to` соответствует `link_to_remote`, так и у метода `form_for` есть аналог `remote_form_for`. Он принимает те же параметры и обратные вызовы, что `link_to_remote`, но возвращает тег `form` формы, асинхронно отправляемой серверу с помощью объекта `XMLHttpRequest`. Как и `form_for`, метод `remote_form_for` представляет значения в виде стандартного объекта `params`. Мы можем проиллюстрировать это на примере Ajax-формы, которая создаст новый экземпляр модели `Addition`, сложит оба атрибута и вернет результат. Контроллер при этом выглядит так:

```
Class AdditionsController < ApplicationController
  def new
    @addition = Addition.new
  end

  def create
    @addition = Addition.new(params[:addition])
    respond_to do |wants|
      wants.js { render :text => @addition.sum_x_and_y}
    end
  end
end
```

Новое представление запишем в следующем виде:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <% remote_form_for :addition, @addition,
```

```
        :url => additions_path,
        :update => 'result' do |f| %>
        X: <%= f.text_field :x %>
        Y: <%= f.text_field :y %>
        <%= submit_tag 'Create' %>
    <% end %>
    <div id="result"></div>
  </body>
</html>
```

Результат его рендеринга – следующий код:

```
<html>
  <head>
    ...
  </head>
  <body>
    <form action="/additions/non_ajax_create" method="post"
          onsubmit="new Ajax.Updater('result', '/additions',
          {asynchronous:true,
evalScripts:true, parameters:Form.serialize(this)}); return false;">
X: <input id="addition_x" name="addition[x]" size="30" type="text" />
Y: <input id="addition_y" name="addition[y]" size="30" type="text" />
<input name="commit" type="submit" value="Create" />
    </form>
    <div id="result"></div>
  </body>
</html>
```

Методу `remote_form_for` можно также передать «аварийный» URL для броузеров, не поддерживающих JavaScript. По умолчанию подразумевается то же действие, которое задано в параметре `url`. Чтобы указать что-то другое, задайте параметр `:action` в хеше `html`:

```
...
<% remote_form_for :addition, @addition, :url => additions_path,
 :update => 'result', :html => { :action => url_for(:controller =>
 'additions', :action => 'non_ajax_create') } do |f| %>
...
```

Еще один способ отправить форму через Ajax – воспользоваться обычным методом `form_for`, но вместо стандартной отправки вызвать метод `submit_to_remote`, который принимает те же параметры, что и `remote_form_for`.

periodically_call_remote

В Rails Ajax часто применяется метод `periodically_call_remote`, который обращается к указанному URL каждые *n* секунд. Он принимает те же параметры и поддерживает такие же обратные вызовы, как метод `link_to_remote`. По умолчанию интервал между обращениями равен де-

сяти секундам, но параметр :frequency позволяет задать иное значение. Можно изменить генератор случайных чисел из предыдущего примера, так чтобы он запрашивал новое число каждые пять секунд. Контроллер для этого модифицировать не потребуется, хватит и представления:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <%= periodically_call_remote :url => new_random_path, :update =>
'result', :frequency => 5 %>
    <br/><br/>
    <div id="result"></div>
  </body>
</html>
```

observe_field

Если метод periodically_call_remote обращается к серверу каждые *n* секунд, то observe_field – только при любом изменении некоторого поля формы. Этим можно воспользоваться, например, для отображения списка возможных кодов городов, когда пользователь вводит число в текстовое поле. Возьмем такое представление index:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    Код города: <%= text_field_tag 'number' %>
    <br/>
    <span id="area_code_results"></span>
    <%= observe_field 'number', :url => { :controller => 'area_codes',
 :action => 'show'}, :frequency => 0.25, :update => '
area_code_results', :with => 'number' %>
  </body>
</html>
```

В написанном ранее контроллере изменим метод show:

```
Class AreaCodesController < ApplicationController
  def show
    respond_to do |wants|
      wants.js {
        @area_codes = AreaCode.find_like(params[:number])
`if @area_codes.empty? || params[:number].blank?
          render :text => ' '
        else
          render :text => @area_codes.map(&:to_s).join('<br/>')
        end
```

```
        }
      end
    end
  end
```

Метод `observe_field` каждые 0,25 секунды проверяет, произошли ли изменения в элементе `DOM` с идентификатором *number*. Если поле изменилось, то есть в него были введены или, наоборот, удалены какие-то данные, посылается запрос `XMLHttpRequest`. Адресатом запроса является действие, заданное в параметре `url`; в данном случае – действие `show` контроллера `area_codes`. Этот метод ищет все коды городов, начинающиеся с уже введенных цифр.

В контроллере можно пользоваться стандартным объектом `params`, поскольку мы задали в методе `observe_field` параметр `with`. Иначе пришлось бы анализировать само тело запроса с помощью метода `request.body.read`. Можно было бы также задать дополнительные параметры, записав в параметр `with` примерно такую строку: `'number='+ escape($('number').value) + '&other_value=-1'`. Значения `number` и `other_value` можно было бы получить из того же объекта `params`.

По умолчанию метод `observe_field` срабатывает по событию `change` для текстовых полей и областей и по событию `click` для переключателей и флажков. Если вы хотите отслеживать другое событие, просто передайте в параметре `on` его имя, например `blur` или `focus`. В примере выше мы обращались к URL, но могли бы с тем же успехом вызвать и JavaScript-функцию. Для этого достаточно было бы указать в параметре `:function` ее имя: `:function => 'update_results'`. Помимо перечисленных, метод `observe_field` принимает еще и все параметры, распознаваемые методом `link_to_remote`.

observe_form

Если вы хотите наблюдать за всей формой, метод `observe_form` может оказаться удобнее, нежели `observe_field`. Он принимает DOM-идентификатор формы и следит за всеми ее элементами. Параметры и поведение у него такие же, как у метода `observe_field`, только по умолчанию в параметр `with` заносится сериализованное представление формы (строка запроса).

RJS – пишем Javascript на Ruby

В Rails включена возможность, называемая RJS. Расшифровывается этот акроним, по всей видимости, как Ruby JavaScript. RJS API генерирует код на языке JavaScript, исходя из кода на Ruby, и позволяет тем самым манипулировать представлением или его частями со стороны сервера.

В примере с кодами городов из предыдущего раздела мы выводили результат поиска кода с помощью метода `render :text`:

```
render :text => @area_codes.map(&:to_s).join('<br/>')
```

А если хочется поместить в начало списка информацию о том, сколько кодов было найдено? Можно было бы просто добавить в возвращаемый результат такую строку:

```
render :text => "Найдено кодов: #{area_codes.size}<br/>
#{@area_codes.map(&:to_s).join('<br/>')}"
```

Этот способ годится, но что если количество найденных результатов необходимо показать еще и в другом месте формы, где простая конкатенация строк работать не будет?

Для начала знакомства с RJS изменим структуру шаблона представления следующим образом:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    Код города: <%= text_field_tag 'number' %>
    <span id="area_code_results_message"></span>
    <br/>
    <hr/>
    <span id="area_code_results"></span>
    <%= observe_field 'number', :url => { :controller => 'area_codes',
  :action => 'show'}, :frequency => 0.25, :with => 'number' %>
  </body>
</html>
```

Теперь перепишем блок в методе `respond_to`, воспользовавшись RJS:

```
wants.js {
  @area_codes = AreaCode.find_like(params[:number])
  if @area_codes.empty?
     render :update do |page|
       page.replace_html 'area_code_results_message',
                         'Ничего не найдено'
       page.replace_html 'area_code_results', ''
     end
  else
     render :update do |page|
       page.replace_html 'area_code_results_message',
                         "Найдено кодов: #{@area_codes.size}"
       page.replace_html 'area_code_results',
                         @area_codes.map(&:to_s).join('<br/>')
     end
  end
}
```

Поскольку мы пользуемся RJS, уже нет необходимости передавать методу `observe_field` параметр `update`. Связано это с тем, что и `observe_field`, и все остальные методы, которые мы до сих пор обсуждали, исполняют любой полученный JavaScript-код.

В контроллере мы больше не будем выводить текст, а вызовем метод `render :update`, который попросит `ActionController` сгенерировать блок кода на языке JavaScript. Rails предлагает целый ряд помощников для создания JavaScript-кода.

В примере выше используется помощник `replace_html`, заменяющий значением второго аргумента HTML-содержимое элемента, идентификатор которого задан первым аргументом.

С помощью FireBug можно посмотреть, какой JavaScript-код был послан броузеру в теле ответа:

```
Element.update("area_code_results_message", "Found 41 Results");
Element.update("area_code_results", "301 - MD, W Maryland: Silver
Spring, Frederick, Camp Springs, Prince George's County (see
240)\074br/\076302 - DE, Delaware\074br/\076303 - CO, Central
Colorado:Denver (see 970, also 720 overlay)\074br/\076...
```

RJS–шаблоны

Объединив логику контроллера и представления в одном месте, мы отклонились от рекомендуемой методики. Это можно исправить, вынеся RJS-код из контроллера в шаблоны.

Сначала создадим следующий файл представления, назвав его `show.js.rjs`:

```
if @area_codes.empty? || params[:number].blank?
  page.replace_html 'area_code_results_message',
                    'Ничего не найдено'
  page.replace_html 'area_code_results', ''
else
  page.replace_html 'area_code_results_message',
                    "Найдено результатов: #{@area_codes.size}"
  page.replace_html 'area_code_results',
                    @area_codes.map(&:to_s).join('<br/>')
end
```

Теперь подправим контроллер:

```
class AreaCodesController < ApplicationController
  def show
     @area_codes = AreaCode.find(:all,
       :conditions => ['number like ?', "#{params[:number]}%"])
  end
end
```

Конструкция `respond_to` исчезла, и мы полагаемся на то, что Rails по умолчанию выбирает представление, соответствующее запросу. Иными словами, Rails выберет JavaScript-представление, если запрос был

отправлен с помощью XMLHttpRequest. RJS также можно использовать в помощниках.

В состав Rails входит несколько методов, имеющих касательство к RJS, и мы опишем их в следующих разделах.

<<(javascript)

Данный метод вывод на страницу готовый JavaScript-код. Это полезно, когда вы включили в файл application.js собственный метод и хотите его вызвать. Например:

```
// application.js
function my_method() {
  alert('вызван мой метод');
}

// my_controllers.rb
class MyControllers < Application
  def show
    ...
    render :update do |page|
      page << 'my_method();'
    end
    ...
  end
end
```

[](id)

Возвращает ссылку на элемент с указанным идентификатором DOM. Впоследствии для этого элемента можно вызывать такие методы, как hide, show и т. д. Поведение такое же, как у конструкции $(id) в библиотеке Prototype:

```
render :update do |page|
  page['my_div'].hide # то же, что $('my_div').hide
end
```

alert(message)

Выводит сообщение методом JavaScript alert:

```
render :update do |page|
  page.alert('Тут что-то не в порядке')
end
```

call(function, *arguments, &block)

Вызывает JavaScript-функцию с заданными аргументами. Если задан еще и блок, создается новый генератор, и весь сгенерированный JavaScript-код погружается в анонимную функцию function() { ... } и передается в качестве последнего аргумента класса:

```
// application.js
function my_method() {
  alert('Вызван мой метод');
}

// my_controllers.rb
class MyControllers < Application
  def show
    ...
    render :update do |page|
      page.call('my_method')
    end
    ...
  end
end
```

delay(seconds = 1) { ... }

Заданный блок выполняется по прошествии указанного количества секунд:

```
render :update do |page|
  page.delay(5) {
    page.visual_effect :highlight, 'results_div', :duration => 1.5
  }
end
```

draggable(id, options = {})

Создается элемент, допускающий перетаскивание (перетаскиваемые элементы обсуждаются в разделе «Перетаскивание мышью»).

drop_receiving(id, options = {})

Создается элемент, допускающий бросание (обсуждается в разделе «Перетаскивание мышью»).

hide(*ids)

Скрывает элементы с заданными идентификаторами DOM:

```
render :update do |page|
  page.hide('options_div')
  page.hide('options_form', 'options_message')
end
```

insert_html(position, id, *options_for_render)

Вставляет HTML-код в заданное место относительно элемента с указанным идентификатором DOM. Значения параметра position перечислены в табл. 12.2.

Таблица 12.2. Допустимые значения параметра position метода insert_html

Параметр	Описание
:top	HTML вставляется внутрь элемента, перед текущим содержимым
:bottom	HTML вставляется внутрь элемента, после текущего содержимого
:before	HTML вставляется непосредственно перед элементом
:after	HTML вставляется непосредственно после элемента

Параметр options_for_render может содержать либо строку HTML-кода, либо опции, передаваемые методу render:

```
render :update do |page|
  page.insert_html :after, 'my_div', '<br/><p>Мой текст</p>'
  page.insert_html :before, 'my_other_div', :partial => 'list_items'
end
```

literal (code)

Применяется для передачи литерального JavaScript-выражения в качестве аргумента другого метода-генератора JavaScript. У возвращаемого объекта есть метод to_json, который порождает код.

redirect_to(location)

Заставляет броузер перейти по указанному адресу:

```
render :update do |page|
  page.redirect_to 'http://www.berlin.de'
end
```

remove(*ids)

Удаляет элементы с указанными идентификаторами DOM.

replace(id, *options_for_render)

Заменяет элемент с указанным идентификатором DOM целиком (а не только содержащийся внутри HTML), подставляя вместо него строку или содержимое, определяемое параметром options_for_render:

```
render :update do |page|
  page.replace 'my_div', '<div>Сообщение</div>'
  page.replace 'my_div', :partial => 'entry'
end
```

replace_html(id, *options_for_render)

Заменяет внутреннее HTML-содержимое элемента с указанным идентификатором DOM, подставляя вместо него строку или содержимое, определяемое параметром options_for_render.

select(pattern)

Получает набор ссылок на элементы, отвечающие указанному CSS-образцу. К возвращенному набору можно применять стандартные методы объекта `Enumerable` из библиотеки Prototype:

```
render :update do |page|
  page.select('div.header p').first
  page.select('div.body ul li').each do |value|
    value.hide
  end
end
```

show(*ids)

Показывает ранее скрытые элементы с заданными идентификаторами `DOM`.

sortable(id, options = {})

Создает допускающий сортировку элемент (эта тема обсуждается в разделе «Сортируемые списки».

toggle(*ids)

Меняет видимость элемента с заданными идентификаторами `DOM` на противоположную. Другими словами, видимые элементы становятся скрытыми, и наоборот.

visual_effect(name, id = nil, options = {})

Запускает визуальный эффект с указанным именем для элемента с заданным идентификатором `DOM`. Из RJS можно вызывать эффекты `appear`, `fade`, `slidedown`, `slideup`, `blinddown` и `blindup`. Для смены эффекта на противоположный существуют также методы `toggle_appear`, `toggle_slide` и `toggle_blind` (полный перечень визуальных эффектов, позволяющих не только отображать элементы, см. в документации по библиотеке Scriptaculous). Например, для постепенного растворения изображения надо было бы написать:

```
render :update do |page|
  page.visual_effect :fade, 'my_div'
end
```

JSON

JavaScript Object Notation (JSON) – это способ простого кодирования JavaScript-объектов. Rails предоставляет метод `to_json` для любого

объекта. Одного и того же результата можно достичь как с помощью RJS, так и с помощью JSON. Основное различие заключается в том, где находится логика обработки результата. В RJS обработкой занимается Rails, а в JSON – JavaScript-код на стороне броузера.

Для иллюстрации изменим написанный ранее контроллер, чтобы он возвращал JSON-представление:

```
class AreaCodesController < ApplicationController
  def show
    respond_to do |wants|
      wants.json {
        @area_codes=
          AreaCode.find_all_by_number(params[:area_code][:number])
        render :json => @area_codes.to_json
      }
    end
  end
end
```

Этот код вернет такой результат:

```
[{attributes: {updated_at: "2007-07-22 20:47:18", number: "340", id:
"81", description: "US Virgin Islands (see also 809)", created_at:
"2007-07-22 20:47:18", state: "VI"}}, {attributes: {updated_at: "2007-
07-22 20:47:18", number: "341", id: "82", description: "(overlay on
510; SUSPENDED)", created_at: "2007-07-22 20:47:18", state: "CA"}},
{attributes: {updated_at: "2007-07-22 20:47:18", number: "345", id:
"83", description: "Cayman Islands", created_at: "2007-07-22
20:47:18", state: "-"}}, {attributes: {updated_at: "2007-07-22
20:47:18", number: "347", id: "84", description: "New York (overlay
for 718: NYC area, except Manhattan)", created_at: "2007-07-22
20:47:18", state: "NY"}}]
```

Теперь надо изменить представление, чтобы оно правильно обрабатывало возвращенное содержимое в формате JSON:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    Код города: <%= text_field_tag 'number' %>
    <span id="area_code_results_message"></span>
    <br/>
    <hr/>
    <span id="area_code_results"></span>
    <%= observe_field 'number',
          :url => { :controller => 'area_codes',:action => 'show'},
          :frequency => 0.25,
          :with => 'number',
          :complete => "process_area_codes(request)" %>
  </body>
</html>
```

Единственное изменение – это добавление обратного вызова JavaScript-функции process_area_codes, которое мы определим в файле application.js:

```
function process_area_codes(request) {
  area_codes = request.responseText.evalJSON();
  $('area_code_results').innerHTML = ' ';
  area_codes.each(function(area_code, index) {
    new Insertion.Bottom("area_code_results", "<li>" +
      area_code.attributes.number + " - " +
      area_code.attributes.state + ", " +
      area_code.attributes.description + "</li>");
  });
}
```

Перетаскивание мышью

Библиотека Scriptaculous и так упрощает процедуру перетаскивания, а Rails добавляет несколько методов-помощников, упрощающих ее еще сильнее. Проиллюстрируем это, сделав каждый элемент списка кодов перетаскиваемым на область бросания, где он становится выбранным. Сначала изменим представление:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    Код города: <%= text_field_tag 'number' %>
    <span id="area_code_results_message"></span>
    <hr/>
    Выбран: <span id="selected" style="padding: 0 100px; width:
200px; height: 200px; background-color: lightblue;"></span>
<%= drop_receiving_element 'selected', :onDrop =>
"function(element) { $('selected').innerHTML = element.innerHTML; }",
:accept=>'area_code' %>
    <hr/>
    <span id="area_code_results"></span>
    <%= observe_field 'number', :url => { :controller => 'area_codes',
:action => 'show'}, :frequency => 0.25, :with => 'number' %>
  </body>
</html>
```

Мы воспользовались JavaScript-помощником drop_receiving_element, чтобы на элемент с идентификатором 'selected' можно было бросать элементы с CSS-классом 'area_code'. Мы также присвоили его параметру onDrop ссылку на JavaScript-функцию, которая будет копировать внутреннюю HTML-разметку перетащенного элемента. Теперь изменим представление show.js.rjs, чтобы все возвращенные коды городов допускали перетаскивание:

```
if @area_codes.empty? || params[:number].blank?
  page.replace_html 'area_code_results_message',
                    'Ничего не найдено'
  page.replace_html 'area_code_results', ''
else
  page.replace_html 'area_code_results_message',
                    "Найдено результатов #{@area_codes.size}"
  page.replace_html 'area_code_results', ''

  @area_codes.each do |area_code|
    id = area_code.number.to_s
    page.insert_html :bottom,
                     'area_code_results',
                     content_tag(:div,
                                 area_code,
                                 :id => id, :class => 'area_code')
    page.draggable id, :revert => true
  end
end
```

Здесь мы обходим все коды городов и делаем каждый из них перетаскиваемым элементом `div`. В идентификатор элемента записывается номер кода города и устанавливается CSS-класс `'area_code'`. Последнее существенно, поскольку на элемент-приемник, созданный в предыдущем разделе, можно бросать только элементы с классом `'area_code'`.

Теперь можно обновить страницу и перетащить код города на покрашенный прямоугольник «Выбран». Это симпатично, но хотелось бы еще отправлять серверу выбранный код города. Для этого изменим описание элемента-приемника:

```
<%= drop_receiving_element 'selected',
        :onDrop => "function(element) {$('selected').innerHTML =
                                        element.innerHTML; }",
        :accept => 'area_code',
        :url => { :controller => 'area_codes',
                  :action => 'area_code_selected' } %>
```

Теперь при бросании элемента будет отправлен запрос `XMLHttpRequest` методу `area_code_selected`. По умолчанию серверу посылается идентификатор `id` брошенного элемента, то есть номер выбранного кода города:

```
Class AreaCodesController < ApplicationController
  def area_code_selected
    area_code = AreaCode.find_by_number(params[:id])
    # сделать что-то с кодом города
    render :nothing => true
  end
end
```

Сортируемые списки

Поверх механизма перетаскивания библиотека Scriptaculous и Rails реализуют возможность создания сортируемых списков. Мы можем воспользоваться ею для сортировки возвращенного списка кодов городов. Сначала изменим представление, организовав набор кодов в виде маркированного списка, поскольку именно этого ожидает JavaScript-функция `sortable`:

```
<html>
  ...
  <ul id="area_code_results"></ul>
  ...
</html>
```

В файле `show.js.rjs` тоже нужно перейти к тегам `li` и сделать список сортируемым. Кроме того, мы удалим объявление `draggable`, поскольку это поведение подразумевается по умолчанию, когда список объявляется сортируемым:

```
if @area_codes.empty? || params[:number].blank?
  ...
else
  ...
  @area_codes.each do |area_code|
    id = area_code.number.to_s
    page.insert_html :bottom, 'area_code_results', content_tag(:li,
area_code, :id => id, :class => 'area_code')
  end
  page.sortable 'area_code_results', :url => { :controller =>
    'area_codes', :action => 'sorted_area_codes' }
end
```

После этих изменений мы можем сортировать возвращенные коды городов путем перетаскивания внутри списка. Каждый раз, как элемент с кодом города бросается, методу `sorted_area_codes` отправляется запрос `XMLHttpRequest`. Доступ к списку мы получаем с помощью элемента `params[:area_code_results]`, который содержит массив кодов городов в отсортированном виде, причем каждый код представлен своим идентификатором в `DOM`. Если в идентификаторе присутствует знак подчерка, то сериализуется и посылается серверу только последняя часть. Например, для элемента с идентификатором `'string_1'` будет отправлено `'1'`.

Автозавершение

Хотя эта функциональность в версии Rails 2.0 перенесена в подключаемый модуль, она все равно остается весьма полезной. Автозавершите-

ли часто используются, чтобы предлагать список вариантов при вводе в текстовое поле.

Возвращаясь к нашему примеру, можно применить автозавершитель для вывода подходящих кодов городов по мере ввода. Иначе говоря, после того как я введу цифру 6, автозавершитель покажет мне все коды, начинающиеся с шестерки. Для начала укажем специальное текстовое поле:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    Код города: <%= text_field_with_auto_complete :area_code, :number
%>
  </body>
</html>
```

Здесь создается обычное текстовое поле, с которым ассоциирован объект `Ajax.Autocompleter`; при каждом нажатии клавиши он будет отправлять запрос `XMLHttpRequest` методу `auto_complete_for_area_code_number`. По умолчанию `text_field_with_auto_complete` ищет метод с именем `auto_complete_for_#{object_name}_#{field}`. Такой метод в нашем контроллере выглядит следующим образом:

```
class AreaCodesController < ApplicationController
  ...
  def auto_complete_for_area_code_number
    @area_codes = AreaCode.find_by_number(params[:area_code][:number])
render :inline => "<%=auto_complete_result(@area_codes, :number)%>"
  end
  ...
end
```

Мы передаем методу `render` параметр `:inline`, поскольку `auto_complete_result` — это помощник `ActionView`, который напрямую в контроллере недоступен.

Редактирование на месте

С помощью комбинации Rails и Scriptaculous можно организовать редактирование на месте. В Rails 2.0 эта функциональность также будет вынесена из ядра в подключаемый модуль. Чтобы таким способом редактировать описание кода города, надо изменить наше представление следующим образом:

```
<html>
  <head>
    <%= javascript_include_tag :defaults %>
  </head>
```

```
  <body>
    Номер: <%= in_place_editor_field :area_code, :number %><br/>
    Штат: <%= in_place_editor_field :area_code, :state%><br/>
    Описание: <%= in_place_editor_field :area_code, :description %>
  </body>
</html>
```

Поля, допускающие редактирование на месте, по умолчанию ищут метод с именем `set_#{object_name}_#{field}`. Так, полю `number` (Номер) соответствует метод `set_area_code_number`:

```
Class AreaCodesController < ApplicationController
  ...
  def set_area_code_number
    @area_code = AreaCode.find(params[:id])
    render :text => @area_code.number
  end
  ...
end
```

Заключение

Успех Rails часто соотносят с наступлением Web 2.0, и один из факторов, связывающих Rails с этим явлением, – встроенная поддержка Ajax. О программировании Ajax написаны десятки книг, в том числе об использовании Ajax совместно с Rails. Это обширная тема, но она занимает настолько значительное место в Rails, что мы сочли необходимым включить ее в нашу книгу.

В этой главе я настоятельно рекомендовал установить подключаемый к броузеру Firefox модуль FireBug, если вы этого еще не сделали, и пользоваться им. Затем было приведено полное справочное руководство по JavaScript-библиотеке Prototype – неотъемлемой части программирования Ajax, а также обзор функциональности, предоставляемой модулем Rails `PrototypeHelper`.

Из разделов, посвященных RJS, вы узнали о технологии написания JavaScript-кода на Ruby; в некоторых случаях она бывает очень кстати.

Наконец, вы познакомились со встроенными в Rails помощниками для создания визуальных эффектов и элементов управления на базе библиотеки Scriptaculous.

13

Управление сеансами

Мне бы совсем не понравилось проснуться однажды утром и обнаружить, что ты – это не ты!

Д-р Майлз Дж. Биннелл (Кэвин Маккарти)
в фильме «Нашествие похитителей тел» (*Allied Artists*, 1956)

HTTP – протокол без состояния. В отсутствие понятия сеанса (эта идея встречается не только в Rails) было бы невозможно узнать, как один HTTP-запрос соотносится с другим. Вы не смогли бы понять, кто работает с вашим приложением! Идентификацию (и, возможно, аутентификацию) пользователя пришлось бы выполнять для каждого запроса, обрабатываемого сервером[1].

К счастью, когда с приложением, написанным для Rails, начинает работать новый пользователь, для него автоматически создается новый сеанс. С помощью сеансов мы можем сохранять на сервере достаточно информации, что сильно облегчает программистскую задачу.

Слово *сеанс* относится как к промежутку времени, в течение которого пользователь активно работает с приложением, так и к сохраняемой для этого пользователя структуре данных. Эта структура представляет собой хеш, снабженный уникальным *идентификатором сеанса* – 32-значной строкой из случайных шестнадцатеричных цифр. При созда-

[1] Если вы только начали заниматься программированием для Сети и хотите получить очень подробное объяснение принципов работы сеансов, будет полезно ознакомиться со статьей по адресу *http://www.technicalinfo.net/papers/WebBasedSessionManagement.html*.

нии нового сеанса Rails автоматически посылает броузеру cookie, содержащий идентификатор данного сеанса. С этого момента в каждом запросе, который броузер отправляет серверу, будет содержаться тот же самый идентификатор сеанса, что позволяет связать запросы между собой.

Подход Rails к проектированию веб-приложений подразумевает минимальное использование сеансов для хранения данных о состоянии. Согласно философии «ничего не разделяй», которая пронизывает Rails, подходящим местом для хранения данных является база. Точка. Резюме таково: чем дольше вы храните объекты в сеансовом хеше пользователя, тем больше проблем создаете себе, пытаясь предотвратить устаревание объектов (то есть рассинхронизацию с содержимым базы данных).

Эта глава посвящена вопросам, касающимся работы с сеансами. И первый из них – что помещать в сеанс.

Что хранить в сеансе

Решение о том, что именно хранить в сеансовом хеше, не будет сверхтрудным, если вы просто согласитесь хранить как можно меньше. В общем случае целые числа (значения ключей) и короткие строки – это нормально. Объекты – нет.

Текущий пользователь

Есть одно важное целое число, которое хранится в сеансе большинства приложений Rails. Это идентификатор текущего пользователя `current_user_id`. Не объект, представляющий текущего пользователя, а только его идентификатор. Даже если вы придумаете собственный код регистрации и аутентификации (чего делать не следует), не сохраняйте весь объект `User` (или `Person`) в сеансе на время работы пользователя (дополнительную информацию об отслеживании текущего пользователя см. в главе 14 «Регистрация и аутентификация»). Система аутентификации должна загружать экземпляр класса пользователя из базы данных при каждом запросе и поддерживать его в актуальном состоянии с помощью метода в подклассе `ApplicationController` вашего приложения. Среди прочего, следование этому совету позволит вам запрещать доступ тому или иному пользователю, не дожидаясь, пока истечет срок действия его сеанса.

Рекомендации по работе с сеансами

Вот несколько общих рекомендаций по хранению объектов в сеансе:

- они должны быть сериализуемы с помощью Marshal API, являющегося частью Ruby. Это исключает, в частности, соединения с базой данных и другие объекты ввода/вывода;

- большие графы объектов могут не удовлетворять ограничениям на размер сенсового хранилища. Наличие таких ограничений зависит от выбранного хранилища; этот вопрос будет рассмотрен ниже;
- критически важные данные не следует хранить в сеансе, поскольку они могут быть утрачены в результате неожиданного исчезновения сеанса (например, если пользователь закрыл броузер или стер cookies);
- не следует хранить в сеансе объекты с часто изменяющимися атрибутами. Подумайте, что произойдет, когда в модели появятся кэшируемые атрибуты-счетчики, а вы будете постоянно обращаться к ее копии в сеансе, но не извлекать из базы данных. Счетчики навсегда потеряют свою актуальность;
- изменение структуры объекта и сохранение старых версий в сеансе – прямая дорога к неприятностям. Сценарии развертывания должны очищать все старые сеансы, чтобы такие проблемы не возникали, но для некоторых типов сеансовых хранилищ, например в cookies, избежать их трудно. Самое простое решение (я повторяюсь) – вообще не хранить объекты в сеансе.

Способы организации сеансов

Метод класса `session` предоставляет несколько способов организации сеансов в контроллерах. Обращения к нему можно поместить прямо в класс `ApplicationController` или в начало конкретных контроллеров приложения.

Например, некоторым приложениям вообще не нужно отслеживать сеансы пользователей, и тогда можно резко повысить производительность, отключив соответствующую часть обработки запросов в Rails:

```
# отключить управление сеансами для всех действий.
session :off
```

Как и для других методов класса, имеющих отношение к конфигурированию, поддерживаются параметры `:except` и `:only`:

```
# отключить управление сеансами для всех действий, кроме foo и bar.
session :off, :except => %w(foo bar)
# отключить управление сеансами только для действий atom и rss.
session :off, :only => %w(atom rss)
```

Параметр `:if` также поддерживается и бывает полезен, когда нужно по атрибутам конкретного запроса решить, заслуживает ли он создания сеанса:

```
# сеанс будет отключен только для действия'foo', но лишь в том случае,
# когда оно запрошено как веб-служба
session :off, :only => :foo, :if => lambda { |req| req.parameters[:ws]
}
```

Отключение сеансов для роботов

Если вы разместили публичный веб-сайт, то агенты-пауки (их еще называют *роботами*) рано или поздно его найдут. Поскольку они не поддерживают cookies, каждый запрос будет приводить к созданию нового сеанса, что создает совершенно излишнюю нагрузку на сервер.

Отключить сеансы специально для роботов довольно просто, так как они идентифицируют себя в заголовке User-Agent HTTP-запроса. Нужно лишь добавить в класс `ApplicationController` вызов метода `session :off` с динамическим условием `:if`, как показано в листинге 13.1.

Листинг 13.1. Отключение сеансов для роботов с помощью анализа строки User-Agent

```
class ApplicationController < ActionController::Base
session :off, :if => lambda {|req| req.user_agent =~
/(Google|Slurp)/i}
```

Типичный робот Googlebot идентифицирует себя как Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html). Робот Yahoo называет себя Mozilla/5.0 (compatible; Yahoo! Slurp; *http://help.yahoo.com/help/us/ysearch/slurp*). Стоит провести небольшое исследование с привлечением протоколов доступа веб-сервера и выяснить, какие роботы заходят на ваш сайт и с какими пользовательскими агентами надо проводить сопоставление. Выявленные строки добавьте в регулярное выражение внутри блока `lambda`.

У данной техники имеется еще один аспект, связанный с тестированием. На момент работы над этой книгой в классе `TestRequest` нет метода `user_agent`. Следовательно, тесты и спецификации контроллера обрушатся после добавления вызова `req.user_agent`, показанного в листинге 13.1. К счастью, классы в Ruby открыты, поэтому проблему легко решить. Достаточно добавить показанный в листинге 13.2 код в файл `test_helper.rb` или `spec_helper.rb`.

Листинг 13.2. Заплата на класс TestRequest для включения метода user_agent

```
class ActionController::TestRequest
  def user_agent
    "Mozilla/5.0"
  end
end
```

Избирательное включение сеансов

Предположим, что ваше приложение не нуждается в сохранении состояния и вы глобально отключили сеансы в классе `ApplicationController`:

```
class ApplicationController < ActionController::Base
  session :off
```

Но в некоторых контроллерах все же хотелось бы сеансы включить, например, для консоли администратора. Писать `session :on` в подклассе `ApplicationController` бессмысленно – не будет работать, но, как ни странно, можно написать `session :disable => false`:

```
class AdminController < ApplicationController
  session :disable => false
```

Безопасные сеансы

Иногда необходимо настроить приложение Rails так, чтобы сеансы работали только для протокола HTTPS:

```
# сеансы будут работать только по протоколу HTTPS
session :session_secure => true
```

Параметр `:session_secure` можно использовать также в сочетании с `:only`, `:except` и `:if`, чтобы обезопасить лишь определенные части приложения. Имейте в виду, что для корректной работы в этом режиме необходимо правильно настроить веб-сервер.

Хранилища

Существуют различные механизмы сохранения сеансов на Rails-сервере, и вы должны выбрать механизм, который лучше всего отвечает потребностям вашего приложения. По моим оценкам, 80% (а то и больше) приложений, развернутых в режиме эксплуатации, пользуются хранилищем `ActiveRecord SessionStore`.

ActiveRecord SessionStore

По умолчанию Rails сохраняет данные сеансов в файлах папки `/tmp/sessions` проекта. Это приемлемо для экспериментов и совсем небольших приложений. Если приложение сколько-нибудь крупное, рекомендуется минимизировать взаимодействие Rails с файловой системой. К сеансам это тоже относится.

Существует ряд настроек для оптимизации хранилища сеансов, но чаще всего используют `ActiveRecord` и сохраняют данные сеансов в базе. На самом деле, это настолько распространенный способ, что в Rails встроены даже инструменты для перехода к такой конфигурации. Первым делом необходимо создать миграцию, пользуясь заданием `rake`, предназначенным специально для данной цели, и запустить эту миграцию для создания новой таблицы:

```
$ rake db:sessions:create
exists db/migrate
create db/migrate/009_add_sessions.rb
$ rake db:migrate
```

```
(in /Users/obie/prorails/time_and_expenses)
== AddSessions: migrating ====================================
-- create_table(:sessions)
-> 0.0049s
-- add_index(:sessions, :session_id)
-> 0.0033s
-- add_index(:sessions, :updated_at)
-> 0.0032s
== AddSessions: migrated (0.0122s) ===========================
```

Второй (и последний) шаг – сообщить Rails о том, что отныне хранить сеансы следует в таблице `sessions`. Для этого предусмотрена настройка в файле `config/environment.rb`:

```
config.action_controller.session_store = :active_record_store
```

Вот и все.

PStore (на базе файлов)

По умолчанию для хранения сеансов в Rails применяются файлы в формате PStore, находящиеся в каталоге `tmp`. В них содержимое сеансовых хешей представлено в «родном» сериализованном виде. Для работы в этом режиме вам не придется менять никакие настройки.

Если нагрузка на ваш сайт велика, то этот режим не для вас! Его производительность низка, поскольку маршалинг и демаршалинг структур данных в Ruby работает медленно. Кроме того, серверу будет тяжело управляться с тысячами файлов сеансов в одном каталоге; возможны и отказы из-за переполнения таблицы файловых дескрипторов. Я видел (и писал об этом в блоге) реальный сайт под управлением Rails, который «падал», поскольку в разделе диска, где хранились файлы сеансов, заканчивалось место!

Хранилище DRb

DRb – это служба распределенных вычислений, написанная на Ruby. Она позволяет процессам Ruby легко обмениваться объектами по сети. Чтобы воспользоваться хранилищем DRb, необходимо запустить отдельный DRb-сервер, который будет играть роль репозитория сеансов. Но с появлением дополнительных процессов усложняется процедура развертывания и сопровождения. Поэтому, если соображения производительности не заставляют вас выбрать именно этот режим, лучше остановиться на хранилище `ActiveRecord Session Store`.

В настоящий момент DRb-хранилище совсем непопулярно и, как утверждают авторы некоторых блогов, не всегда даже правильно работает. Если вы хотите поэкспериментировать с DRb в качестве хранилища сеансов Rails, поищите сценарий `drb_server.rb` в дистрибутиве Rails:

```
config.action_controller.session_store = :drb_store
```

Если вам кажется, что возможностей ActiveRecord Session Store уже не хватает, попробуйте экстра-супероптимизированную версию, которую написал Стефан Каес (Stefan Kaes)[1] или подумайте насчет хранилища memcache.

Хранилище memcache

Если ваш сайт очень сильно нагружен, вероятно, вы уже в том или ином виде пользуетесь хранилищем memcache. В этом решении применяется кэш в памяти удаленного процесса – на нем основано большинство высоконагруженных Rails-сайтов в Интернете.

Для организации репозитория сеансов необходим отдельный memcache-сервер, который работает поразительно быстро. Удобно и то, что в хранилище уже встроен механизм отслеживания срока хранения сеансов, поэтому вам не придется удалять устаревшие сеансы самостоятельно. Однако настроить и сопровождать это хранилище гораздо сложнее[2]:

```
config.action_controller.session_store = :mem_cache_store
```

Чтобы хранилище memcache заработало, необходимо включить его настройки в файл environment.rb:

```
require 'memcache'
memcache_options = {
  :c_threshold => 10_000,
  :compression => true,
  :debug => false,
  :namespace => ":app-#{RAILS_ENV}",
  :readonly => false,
  :urlencode => false
}

CACHE = MemCache.new memcache_options
CACHE.servers = 'localhost:11211'

ActionController::Base.session_options[:expires] = 1800
ActionController::Base.session_options[:cache] = CACHE
```

---

[1] Версия ActiveRecord SessionStore Стефана Каеса находится по адресу *http://railsexpress.de/blog/articles/2005/12/19/roll-your-own-sql-session-store*.

[2] Джеффри Грозенбах написал блестящее руководство по работе с memcache. Его можно найти по адресу *http://nubyonrails.com/articles/2006/08/17/memcached-basics-for-rails*.

Спорное хранилище CookieStore

В феврале 2007 года один из разработчиков ядра, Джереми Кемпер (Jeremy Kemper), положил в основную ветвь довольно смелое дополнение к Rails. Он изменил принимаемый по умолчанию механизм хранения сеансов с достопочтенного `PStore` на новый, основанный на хранилище `CookieStore`. И так прокомментировал свое решение:

> *По умолчанию в Rails вводится хранилище сеансов на базе cookies. Обычно в сеансе хранится всего лишь идентификатор пользователя и короткое сообщение; оба укладываются в ограничение 4 Кб на размер cookie. Для обеспечения целостности данных в cookie включена безопасно вычисляемая свертка (пользователь, не знающий секретного ключа, использованного для ее генерации, не сможет изменить свой user_id). Если вам необходимо хранить в сеансе более 4 Кб данных или вы не хотите, чтобы пользователь видел хранящиеся в сеансе данные, выберите другое хранилище. Хранение сеанса в cookie существенно быстрее альтернативных вариантов.*

Для использования `CookieStore` необходимо установить версию Rails 2.0 (в которой этот механизм принят по умолчанию) или добавить в файл `environment.rb` следующий конфигурационный раздел:

```
config.action_controller.session = {
  :session_key => '_my_app_session',
  :secret => 'длинный ключ для вычисления свертки'
}
```

Я назвал хранилище `CookieStore` спорным из-за негативных последствий решения сделать его умалчиваемым. Прежде всего оно налагает строгие ограничения на размер сеанса – не более 4 Кб. С таким ограничением можно смириться, если вы следуете Пути Rails и не храните в сеансе ничего, кроме целых чисел и коротких строк. Если же вы презрели рекомендации, то можете столкнуться с проблемами.

Кроме того, многие разработчики отмечали небезопасность хранения данных сеанса, в том числе идентификатора текущего пользователя, на компьютере, где находится броузер. Впрочем, приняты меры, серьезно затрудняющие взлом сеансового cookiea. Для такой цели пришлось бы скомпрометировать алгоритм SHA512, а это непростая задача.

Если вам нужна еще более сильная защита[1], можете подменить применяемый алгоритм вычисления свертки:

```
class CGI::Session::CookieStore
  def generate_digest(data)
```

[1] Мой старый приятель Кортенэ поместил в своем блоге яркое сообщение по поводу хранения сеансов в cookiee. См. *http://blog.caboo.se/articles/2007/2/21/new-controversial-default-rails-session-storage-cookies*.

```
    # подставить в эту строку иной криптографический алгоритм
    Digest::SHA512.hexdigest "#{data}#{@secret}"
  end
end
```

Еще одна проблема – уязвимость к *атакам повторным воспроизведением*. На эту тему в списке рассылки по ядру Rails заведена огромная нить сообщений. Начал ее С. Роберт Джеймс[1] (S. Robert James), описав возможную атаку:

> Пример:
>
> 1. Пользователь получает cookie, в котором хранится его кредитный лимит.
> 2. Пользователь совершает покупку.
> 3. В сеанс записывается новый – уменьшенный – кредитный лимит.
>
> Злобный хакер берет cookie, сохраненный на шаге 1, и подсовывает его в хранилище cookies броузера. В результате восстановлен кредитный лимит до покупки.
>
> Обычно такие проблемы решаются с помощью одноразовых маркеров (nonce). В каждое сообщение включается такой маркер, при этом отправитель отслеживает все использованные маркеры и отвергает сообщения с повторяющимися маркерами. Но в данном случае сделать это очень трудно, так как может существовать несколько серверов приложений (Mongrel).
>
> Разумеется, можно было бы хранить маркеры в базе данных, но это сводит на нет исходное намерение!

Краткий ответ таков: не храните в сеансе секретные данные. Никогда. Если хотите более развернутый ответ, то имейте в виду, что для координирования отправки одноразовых маркеров несколькими серверами пришлось бы организовывать дистанционное взаимодействие при каждом запросе, а это аннулирует все преимущества от хранения сеансов в cookies.

У механизма хранения сеансов в cookies есть потенциальная уязвимость к атакам воспроизведением, которая позволяет злонамеренным пользователям, работающим на общих компьютерах, применять украденные cookies, чтобы войти в приложение, из которого законный владелец вроде бы вышел. Подведу итог: если вы решите воспользоваться хранилищем сеансов в cookies, то как следует подумайте о возможных последствиях.

---

[1] Если вы хотите прочитать всю нить (все 83 сообщения), задайте Google запрос Replay attacks with cookie session. Среди результатов поиска должна быть ссылка на эту тему в группе Ruby on Rails: Core Google Group.

Жизненный цикл сеанса и истечение срока хранения

Довольно часто возникает необходимость уничтожить сеанс пользователя, который ничего не делал в течение определенного времени. Как ни странно, по умолчанию эта важнейшая функциональность в Rails не включена. С помощью встроенных параметров сеанса можно задать конкретный срок хранения, но если приложение Rails работает в режиме эксплуатации, срок устанавливается только один раз. Это не страшно, если срок хранения истекает в далеком будущем (после ожидаемого момента перезапуска серверных процессов).

Но что если срок хранения нужно сделать небольшим? Скажем, 20 минут с момент создания сеанса? Будет ли работать следующий подход:

```
class ApplicationController < ActionController::Base
  session :session_expires => 20.minutes.from_now
end
```

Проблема в том, что 20 минут, отсчитываемых от текущего момента (`from now`), то есть с момента запуска серверного процесса, очень скоро останутся в прошлом. Но если дата устаревания сеанса уже прошла, то при каждом запросе будет создаваться новый сеанс, что приведет к хаосу и многим печалям.

Подключаемый модуль Session Timeout

К счастью, существует проверенный практикой подключаемый модуль, решающий эту проблему. Написал его Luke Redpath, а скачать можно со страницы *http://opensource.agileevolved.com/trac/wiki/SessionTimeout*.

После установки этого модуля в приложение в классе `ApplicationController` появится метод `session_times_out_in`. Первый его параметр – интервал в секундах, по истечении которого сеанс устаревает. С помощью вспомогательных методов Rails вы сможете сделать свой код очень простым для восприятия.

Реализуем, например, 20-минутный тайм-аут, который только что обсуждали:

```
class ApplicationController < ActionController::Base
  session_times_out_in 20.minutes
end
```

Второй (необязательный) параметр позволяет указать метод обратного вызова, срабатывающий, когда приходит запрос, при обработке которого обнаруживается, что сеанс устарел. Иногда необходимо произвести какую-то очистку, переадресовать пользователя на другую страницу или сделать что-то подобное.

Как с любыми обратными вызовами в Rails, второй параметр может быть символом, ссылающимся на метод:

```
class ApplicationController < ActionController::Base
  session_times_out_in 20.minutes, :after_timeout => :log_timeout_msg

  private

  def log_timeout_msg
    logger.info "Срок хранения сеанса истек"
  end
end
```

Можно также задать Proc-объект или лямбда-выражение. Ему будет передан экземпляр текущего контроллера, который я в следующем примере проигнорировал:

```
class ApplicationController < ActionController::Base
  session_times_out_in 20.minutes,
    :after_timeout => proc {
      |controller| logger.info "Срок хранения сеанса истек"
    }
end
```

Элегантно, не правда ли?

Отслеживание активных сеансов

Часто необходимо показать, сколько пользователей работают с приложением в текущий момент. При использовании хранилища `ActiveRecord Session Store` сделать это очень легко. По умолчанию таблица `sessions` содержит колонку `updated_at`. В предположении, что *активный* означает «выполнивший какое-то действие в течение последнего часа», количество активных пользователей можно подсчитать следующим образом:

```
CGI::Session::ActiveRecordStore::Session.count :conditions =>
["updated_at > ?", 1.hour.ago ]
```

Объекты, найденные с помощью класса `Session`, можно использовать точно так же, как любые другие экземпляры `ActiveRecord`. Содержимое сеанса хранится в атрибуте `data`, в сериализованном виде (не предназначенном для чтения человеком) в колонке типа `text`, поэтому (по умолчанию) невозможно запрашивать сеансы по содержимому. Хотите посмотреть, как это выглядит?

```
>> CGI::Session::ActiveRecordStore::Session.find:first
=> #<CGI::Session::ActiveRecordStore::Session:0x26fe65c

@attributes={"updated_at"=>"2006-11-29 02:06:01",
"session_id"=>"73bb9cd7fd19a5c1cae8cd0fda0cb6bb", "id"=>"1",
"data"=>"BAh7BiIKZmxhc2hJQzonQWNOaW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo\nSG
FzaHsABjoKQHVzZWR7AA==\n"}>
```

Поэтому, если вы желаете опрашивать данные, хранящиеся в сеансе, например, чтобы вывести список имен активных пользователей, придется немного поработать. Вкратце, вам предстоит добавить еще одну колонку в таблицу `sessions` и фильтр `after_filter` в класс `ApplicationController`, который сохраняет в ней интересующие вас данные при каждом запросе (полную реализацию оставляем в качестве упражнения для читателя).

Повышенная безопасность сеанса

Эрик Элмор (Erik Elmore) в своем блоге приводит длинное и подробное повествование о том, как написать «параноидальное» хранилище сеансов[1]. Среди прочего, его реализация обеспечивает защиту от атак путем фиксации сеанса (сохраняя в составе данных сеанса IP-адрес), возможность узнать, какие пользователи сейчас «в онлайне», и функцию административного прерывания сеансов злоумышленников. Кроме того, она работает «сверхбыстро», поскольку обращается к базе данных напрямую, не создавая объектов `ActiveRecord`. Возможно, его статья не имеет к вам прямого отношения, прежде всего поскольку код ориентирован на СУБД MySQL, но с ней точно стоит ознакомиться, если вы намерены делать с сеансами что-то нестандартное.

Удаление старых сеансов

Если вы пользуетесь подключаемым модулем `session_timeout` и хранилищем `ActiveRecordStore`, то удалить старые сеансы совсем просто. Помните, что у этого модуля есть параметр `:after_timeout`.

Можете также написать собственный небольшой набор утилит для управления сеансами. В листинге 13.3 приведен класс, который можно поместить в папку `/lib` и при необходимости вызывать из консоли или сценария в режиме эксплуатации.

Листинг 13.3. Класс SessionMaintenance для удаления старых сеансов

```
class SessionMaintenance

  def self.cleanup(period=24.hours.ago)
    session_store = CGI::Session::ActiveRecordStore::Session
    session_store.destroy_all( ['updated_at < ?', period] )
  end
end
```

Cookies

Этот раздел посвящен самим cookies, в не хранилищу сеансов в них. *Контейнер cookies* выглядит как хеш, а доступ к нему дает метод

[1] *http://burningtimes.net/articles/2006/10/15/paranoid-rails-session-storage.*

cookies, вызываемый в контексте контроллера. Многие разработчики в Rails используют cookies для хранения пользовательских настроек и других несекретных данных небольшого объема. Но никогда не следует хранить в cookies секретные данные, поскольку злоумышленник легко может прочитать и модифицировать их. Для хранения секретной информации больше подходит база данных.

Вразрез с ожиданиями некоторых разработчиков, контейнер cookies по умолчанию недоступен в шаблонах представлений и помощниках. При необходимости, в согласии с рекомендуемой практикой применения паттерна модель-вид-контроллер, значение cookiea следует помещать в переменную экземпляра в коде контроллера, тогда его можно будет использовать в представлении:

```
@list_mode = cookies[:list_mode] || 'expanded'
```

Если вам действительно нужен доступ к cookies в помощниках или представлениях, существует простое решение – просто объявите cookies как метод-помощник:

```
class MyController < ActionController::Base
  helper_method :cookies
```

Чтение и запись cookies

В контейнер cookies помещаются cookies, полученные вместе с запросом. А в составе ответа отправляются все находящиеся в нем cookies. Отметим, что cookies читаются по значению, поэтому вы не получите сам объект cookiea, а только содержащееся в нем значение в виде строки (или массива строк, если значений несколько). Это ограничение, но не думаю, что на практике оно так уж серьезно.

Чтобы создать или обновить cookie, вы просто присваиваете значения с помощью оператора []. В правой части оператора присваивания может быть либо одна строка, либо хеш опций. Например, опция :expires определяет, через сколько секунд броузер должен удалить cookie. Здесь могут быть полезны различные вспомогательные методы Rails для работы со временем:

```
# запись в простой сеансовый cookie
cookies[:list_mode] = params[:list_mode]

# при задании опций фигурные скобки обязательны, иначе возникнет
# синтаксическая ошибка
cookies[:recheck] = {:value => "false", :expires => Time.now +
5.minutes}
```

Опция :path полезна, когда нужно задать параметры, действующие для конкретных разделов или даже отдельных записей приложения. По умолчанию :path равно '/', что соответствует корню приложения.

Опция `:domain` позволяет задать доменное имя и чаще всего применяется, когда приложение работает на конкретном хосте, но cookies должны распространяться на весь домен.

```
cookies[:login] = {:value => @user.security_token,
                   :domain => '.domain.com',
                   :expires => Time.now.next_year }
```

Можно также задавать опцию `:secure`, тогда Rails будет передавать cookie только по соединению, защищенному протоколом HTTPS:

```
# запись в простой сеансовый cookie
cookies[:account_number] = { :value => @account.number, :secure =>
true }
```

Наконец, метод `delete` позволяет удалять cookies:

```
cookies.delete :list_mode
```

Заключение

Решение о том, как применять сеансы, относится к числу самых сложных для разработчика веб-приложений. Именно поэтому мы посвятили этой теме два раздела в начале главы. Кроме того, мы рассмотрели различные варианты конфигурирования сеансов, включая механизмы хранения и методы управления жизненным циклом сеансов.

Далее мы переходим к тесно связанной теме: регистрации и аутентификации.

14

Регистрация и аутентификация

Слава богу, их всего-то не больше миллиарда, поскольку DHH не считает, что аутентификации и авторизации место в ядре.

Замечание на странице
http://del.icio.us/revgeorge/authentication

Держу пари, что в каждом веб-приложении, над которым вам доводилось работать, есть какая-то форма обеспечения безопасности, и некоторые считают, что имело бы смысл включить тот или иной стандартный механизм аутентификации в такую «повседневную» платформу, как Rails.

Однако так уж получается, что безопасность – один из аспектов проектирования приложений, где бизнес-логики куда больше, чем кажется на первый взгляд. Дэвид четко сформулировал свое мнение по этому поводу, объяснив, почему в Rails не включен стандартный механизм аутентификации:

> *Контекст важнее единообразия. Повторное использование только тогда работает хорошо, когда различные экземпляры настолько похожи, что вы готовы пожертвовать небольшими различиями ради повышения продуктивности. В случае инфраструктуры, каковой является среда Rails, так бывает часто, а в случае бизнес-логики, к каковой относится аутентификация, модули и компоненты, вообще редко.*

Плохо ли, хорошо ли, но мы вынуждены либо сами писать код аутентификации либо искать подходящее решение вне ядра Rails. Написать

код аутентификации нетрудно, как нетрудно вообще *все* в Rails. Но зачем изобретать велосипед? Это не Путь Rails!

Как я намекнул в эпиграфе к этой главе, выбирать есть из чего. Складывается впечатление, что коль скоро аутентификация – одна из функций, которые включаются в приложение в первую очередь, то это также и один из первых проектов, за который принимаются многие честолюбивые авторы подключаемых модулей.

Богатство выбора – вещь хорошая, но в данном случае я склонен считать его скорее злом. По запросу rails authentication Google выдает *свыше пяти миллионов ссылок*! На первой странице результатов поиска я насчитал по меньшей мере десять различных подходов к решению задачи – и что еще за *salted password generator* (генератор паролей с затравкой)?

Но расстраиваться не надо. Все профессионалы, работающие на платформе Rails, согласны, что два самых лучших подключаемых модуля аутентификации написал один из разработчиков ядра Рик Олсон, известный также как *techno weenie*[1]. В этой главе мы рассмотрим один из них – Acts as Authenticated.

Подключаемый модуль Acts as Authenticated

Рик описывает Acts as Authenticated как «простой подключаемый модуль для генерации форм аутентификации». Он позволяет без труда добавить в приложение механизм *аутентификации на основе форм*. Кроме того, он предоставляет стандартный API для получения различной информации, например, о том, зашел ли обычный пользователь или администратор, а также дает доступ к самому объекту `User`.

> **Говорит Кортенэ...**
>
> Acts as Authenticated также умеет работать с базовой HTTP-аутентификацией (она же – *мерзкое окно входа*), поэтому вам ничего не придется делать, чтобы получить защищенный API для своих REST-совместимых приложений. При необходимости модуль даже правильно возвращает код 401 Unauthorized.

Установка и настройка

Чтобы установить библиотеку `acts_as_authenticated` как подключаемый модуль, выполните команду `script/plugin install acts_as_authenticated`. Сначала вы генерируете заголовок кода аутентификации с помощью

[1] Сайт Рика называется *http://techno-weenie.net/*.

входящего в состав модуля генератора, а затем настраиваете базовую реализацию под нужды своего приложения.

Генератор кода вызывается командой `script/generate` и принимает два параметра: имя модели и имя контроллера. Ниже мы вызвали генератор с именами `user` и `account`:

```
$ script/generate authenticated user account
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/account
exists test/functional/
exists test/unit/
create app/models/user.rb
create app/controllers/account_controller.rb
create lib/authenticated_system.rb
create lib/authenticated_test_helper.rb
create test/functional/account_controller_test.rb
create app/helpers/account_helper.rb
create test/unit/user_test.rb
create test/fixtures/users.yml
create app/views/account/index.rhtml
create app/views/account/login.rhtml
create app/views/account/signup.rhtml
create db/migrate
create db/migrate/001_create_users.rb
```

Как и при генерации кода обстраивания (scaffolding), мы видим, что созданы файлы модели, контроллера, миграции и соответствующие тесты. Я подробно расскажу об использовании этого модуля и отмечу, какой урок можно извлечь из него для написания собственного кода.

Модель User

Начнем с файла миграции, который автоматически создал генератор: `db/migrate/001_create_users.rb`:

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table "users", :force => true do |t|
      t.column :login,                            :string
      t.column :email,                            :string
      t.column :crypted_password,                 :string, :limit => 40
      t.column :salt,                             :string, :limit => 40
      t.column :created_at,                       :datetime
      t.column :updated_at,                       :datetime
      t.column :remember_token,                   :string
      t.column :remember_token_expires_at,        :datetime
    end
  end
```

```
  def self.down
    drop_table "users"
  end
end
```

Стандартные колонки мало чем отличаются от тех, что вы включили бы в модель `User` **сами. О колонках** `crypted_password` **и** `salt` **мы поговорим чуть ниже. Если вам нужны дополнительные колонки (скажем, имя и фамилия), добавьте их в файл миграции.**

Теперь откроем файл `app/models/user.rb` **и посмотрим, как выглядит новенькая, с пылу с жару, модель** `User` **(листинг 14.1).**

Листинг 14.1. Модель User, сгенерированная подключаемым модулем Acts As Authenticated

```
require 'digest/sha1'

class User < ActiveRecord::Base
  # Виртуальный атрибут для незашифрованного пароля
  attr_accessor :password

  validates_presence_of            :login, :email
  validates_presence_of            :password,
                                   :if => :password_required?
  validates_presence_of            :password_confirmation,
                                   :if => :password_required?
  validates_length_of              :password, :within => 4..40,
                                   :if => :password_required?
  validates_confirmation_of        :password,
                                   :if => :password_required?
  validates_length_of              :login, :within => 3..40
  validates_length_of              :email, :within => 3..100
  validates_uniqueness_of          :login, :email, :case_sensitive => false

  before_save :encrypt_password

  # Аутентифицирует пользователя по имени и незашифрованному паролю,
  # возвращая объект User или nil
  def self.authenticate(login, password)
    u = find_by_login(login) # необходимо получить затравку
    u && u.authenticated?(password) ? u : nil
  end

  # Шифрует произвольные данные, применяя затравку.
  def self.encrypt(password, salt)
    Digest::SHA1.hexdigest("--#{salt}--#{password}--")
  end

  # Шифрует пароль, применяя затравку.
  def encrypt(password)
    self.class.encrypt(password, salt)
  end
```

```
  def authenticated?(password)
    crypted_password == encrypt(password)
  end

  def remember_token?
    remember_token_expires_at &&
   (Time.now.utc < remember_token_expires_at)
  end

  # Следующие методы создают и сбрасывают поля, необходимые,
  # чтобы вспомнить пользователя при следующем открытии броузера
  def remember_me
    self.remember_token_expires_at =
      2.weeks.from_now.utc
    self.remember_token =
      encrypt("#{email}--#{remember_token_expires_at}")
    save(false)
  end

  def forget_me
   self.remember_token_expires_at = nil
   self.remember_token = nil
   save(false)
  end

  protected
    def encrypt_password
      return if password.blank?
      self.salt =
        Digest::SHA1.hexdigest("--#{Time.now}--#{login}--") if
new_record?
        self.crypted_password = encrypt(password)
    end

    def password_required?
      crypted_password.blank? || !password.blank?
    end
end
```

Ничего себе! Да тут точно больше кода, чем мы привыкли видеть в генерируемых Rails классах. Давайте проанализируем модель `User` и посмотрим, чему можно научиться на ее примере.

Атрибуты, не хранящиеся в базе данных

Иногда имеет смысл включить в модель `ActiveRecord` атрибуты, не хранящиеся в базе данных. Добавляются они с помощью макросов `attr_*`, о которых написано практически в любом руководстве по языку Ruby для начинающих.

Обратите внимание на атрибут `:password` в самом начале модели `User`. Как-то странно он выглядит, да? Разве можно хранить пароль пользователя в базе данных?

Но все встает на свои места, если принять во внимание, что нехранимые атрибуты часто используются в моделях `ActiveRecord` для *недолговечных* данных. Пароль в открытом тексте существует только на протяжении выполнения запроса в составе отправленной HTML-формы. Перед сохранением пароль необходимо зашифровать методом `encrypt_password`.

```
def encrypt_password
  return if password.blank?
  if new_record?
    self.salt = Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}--")
  end
  self.crypted_password = encrypt(password)
end
```

Отметим, что свойство `password` встречается в обращениях `password.blank?` и `encrypt(password)`. Как же атрибут `password` получает значение, если `ActiveRecord` ничего не знает о нехранимых атрибутах? Явно?

Но, имея покрытие автономными тестами, мы можем увидеть, где устанавливается наш атрибут `password`. Внутренний голос мне подсказывает, что в большинстве случаев такие дополнительные атрибуты должны устанавливаться в конструкторах `ActiveRecord`, но с помощью автономного теста для модели `User` я могу это убедительно доказать вам.

Какой еще автономный тест? Текст, который сгенерировал подключаемый модуль. Прежде чем что-нибудь изменять, давайте запустим `rake test` и убедимся, что в самом начале все тесты проходят:

```
$ rake
(in /Users/obie/time_and_expense)
/opt/local/bin/ruby -Ilib:test
"/opt/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader.rb" "test/unit/user_test.rb"
Loaded suite /opt/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader

Started
..........
Finished in 0.312914 seconds.

10 tests, 17 assertions, 0 failures, 0 errors

/opt/local/bin/ruby -Ilib:test
"/opt/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader.rb"
"test/functional/account_controller_test.rb"
Loaded suite /opt/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader
```

```
Started
..............
Finished in 0.479761 seconds.

14 tests, 26 assertions, 0 failures, 0 errors

/opt/local/bin/ruby -Ilib:test
"/opt/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader.rb"
```

Зеленый свет – все тесты прошли! Следовательно, можно спокойно заняться экспериментами. Нас вроде бы интересовало, как используется атрибут `password`. Что сломается, если мы изменим `attribute_accessor` на `attribute_reader`, сделав тем самым `password` доступным только для чтения?

```
class User < ActiveRecord::Base
# Нехранимый атрибут для незашифрованного пароля
attr_reader :password
```

Если снова прогнать тесты, выяснится, что многие перестали работать, причем сообщения об ошибках точно указывают место, где устанавливается значение `password`. И неизменно это оказываются конструкторы `ActiveRecord`.

Например, вот какую ошибку выдает тест метода `signup` из класса `AccountController`:

```
4) Error:
test_should_require_pwd_confirmation_on_signup(AccountControllerTest):
NoMethodError: undefined method `password=' for #<User:0x2a45068>
    active_record/base.rb:1842:in `method_missing'
    active_record/base.rb:1657:in `attributes='
    active_record/base.rb:1656:in `attributes='
    active_record/base.rb:1490:in `initialize_without_callbacks'
    active_record/callbacks.rb:225:in `initialize'
    app/controllers/account_controller.rb:23:in `signup'
```

Стоит взглянуть на метод `signup`, как сразу же обнаружится, что атрибут `password`, передаваемый конструктору `User`, погребен в хеше параметров, которые отправлены в составе формы регистрации, сгенерированной в представлении:

```
def signup
  @user = User.new(params[:user])
  return unless request.post?
  ...
end
```

Нехранимые атрибуты важны, поскольку позволяют скрыть детали реализации от представления. В данном случае было бы небезопасно раскрывать представлению свойства `salt` и `crypted_password`.

Валидаторы

Возвращаясь к модели User, мы видим, что Рик включил разумные умолчания для проверки пароля. Разумеется, никто не мешает изменить их с учетом собственных требований и намерений:

```
validates_presence_of      :password
                           :if => :password_required?
validates_presence_of      :password_confirmation,
                           :if => :password_required?
validates_length_of        :password, :within => 4..40,
                           :if => :password_required?
validates_confirmation_of  :password,
                           :if => :password_required?
```

Условные обратные вызовы

Обратите внимание на использование параметров :if в методах контроля для задания условного обратного вызова. Символ :password_required? определяет, какой метод определяет необходимость контроля.

На первый взгляд кажется, что код не в полной мере соответствует принципу DRY, поскольку строка :if => :password_required? повторяется четыре раза. Можно ли написать лаконичнее?

```
# вроде бы все DRY, но будет ли работать?
if password_required?
  validates_presence_of      :password
  validates_presence_of      :password_confirmation
  validates_length_of        :password, :within => 4..40,
  validates_confirmation_of  :password
end
```

Нет! Подумайте, когда выполняетя это предложение if. Нам нужно проверить условие password_required? на том шаге жизненного цикла модели, где производится контроль, а не в момент определения класса User. Соблюсти принцип DRY можно было бы с помощью метода Rails with_options, но, пожалуй, это тот случай, когда результат не оправдывает усилий.

```
with_options :if => :password_required? do |u|
  u.validates_presence_of       :password
  u.validates_presence_of       :password_confirmation
  u.validates_length_of         :password, :within => 4..40
  u.validates_confirmation_of   :password
end
```

Все эти правила контроля применяются при условии, что атрибут crypted_password пуст, а атрибут password не пуст. В противном случае при каждом обновлении модели User пароль переустанавливается:

```
def password_required?
  crypted_password.blank? || !password.blank?
end
```

Обратный вызов before_save

В главе 2 «Работа с контроллерами» мы говорили, что обратные вызовы позволяют указать метод, который следует вызывать в некоторой точке жизненного цикла объекта ActiveRecord. Например, так:

```
before_save :encrypt_password
```

Возвращаясь к модели User, отметим, что перед добавлением записи о новом пользователе в базу данных пароль необходимо зашифровать. Символ :encrypt_password указывает на одноименный защищенный метод, расположенный в конце класса:

```
def encrypt_password
  return if password.blank?
  if new_record?
    self.salt = Digest::SHA1.hexdigest("--#{Time.now}--#{login}--")
  end
  self.crypted_password = encrypt(password)
end
```

Здесь мы говорим: «Если пароль пуст, не пытаться шифровать его. В противном случае вычислить и запомнить атрибуты salt и crypted_password для последующего сохранения».

В колонке salt хранится одноразовый ключ хеширования. Это делает систему аутентификации более безопасной, чем если бы мы в качестве ключа использовали системную константу.

Метод authenticate

Следующий метод authenticate из двух строчек – отличный пример метода класса. Его логика не связана ни с каким конкретным экземпляром. Однако если вы плохо знаете Ruby, реализация может показаться излишне лаконичной и загадочной. Я помогу вам разобраться:

```
# Аутентифицирует пользователя по имени и незашифрованному паролю,
# возвращая объект User или nil
def self.authenticate(login, password)
  u = find_by_login(login) # необходимо получить затравку
  u && u.authenticated?(password) ? u : nil
end
```

В первой строке мы пытаемся найти запись о пользователе с данным значением login. Если запись не найдена, метод поиска возвращает

значение nil, которое присваивается переменной u, в результате чего выражение && во второй строке возвращает false.

Если же запись найдена, надо проверить пароль. В комментарии Рика просто говорится «необходимо получить затравку», поскольку если бы значение затравки не было уникально для каждого пользователя, то для аутентификации достаточно было бы указать в запросе имя пользователя *и* зашифрованный пароль. В предположении, что обращение к find_by_login возвращает экземпляр класса User, тернарное выражение во второй строке вызывает его метод authenticated? и в зависимости от полученного результата возвращает либо этот экземпляр, либо nil.

Маркер remember_token

Вы, конечно, знаете, что во многих веб-приложениях имеется флажок под полями Имя и Пароль, который позволяет не аутентифицироваться каждый раз вручную. Достигается это с помощью *разделяемого секрета* в виде маркера remember_token. При вызове метода remember_me экземпляра User создается зашифрованная строка, которую броузер пользователя сохранит в виде cookies. В реализации по умолчанию cookies хранится две недели, но вы можете задать другое значение.

Метод forget_me просто стирает соответствующие атрибуты:

```
def remember_me
  self.remember_token_expires_at = 2.weeks.from_now.utc
  self.remember_token =
      encrypt("#{email}--#{remember_token_expires_at}")
  save(false)
end

def forget_me
  self.remember_token_expires_at = nil
  self.remember_token = nil
  save(false)
end
```

Примечание

В последних двух встречается вызов метода save с булевым аргументом. В Rails передача ему значения false означает *сохранение без контроля*. Я отметил это, потому что в документации по Rails API не упоминается, что метод save может принимать булев аргумент. И это неслучайно – в штатной реализации данного аргумента нет, а появляется он, *только когда вы добавляете в модель валидаторы*, то есть сигнатура метода динамически изменяется во время выполнения.

Метод remember_token? проверяет, есть ли маркер с не истекшим сроком действия. Отметим, что по умолчанию принимается время UTC, а не местное:

```
def remember_token?
  remember_token_expires_at && (Time.now.utc <
  remember_token_expires_at)
end
```

На этом мы завершаем рассмотрение модели `User`**, сгенерированной подключаемым модулем** `acts_as_authenticated`**. Разумеется, вы можете добавить в класс** `User` **код, специфичный для своего приложения.**

Класс AccountController

Теперь откроем файл `app/controllers/account_controller.rb` **и посмотрим, какие действия сгенерировал для нас подключаемый модуль (листинг 14.2).**

Листинг 14.2. Класс AccountController

```
class AccountController < ApplicationController

  # Не забудьте, что AuthenticationSystem надо включать в класс Application
  # Controller, а не сюда
  include AuthenticatedSystem

  # Если необходима функциональность "запомни меня", добавьте этот
  # before-фильтр в Application Controller
  before_filter :login_from_cookie

  # Поприветствуйте как-нибудь пользователя.
  def index
    redirect_to(:action => 'signup') unless logged_in? || User.count > 0
  end

  def login
    return unless request.post?
    self.current_user =
      User.authenticate(params[:login], params[:password])
    if current_user
      if params[:remember_me] == "1"
        self.current_user.remember_me
        cookies[:auth_token] =
          { :value => self.current_user.remember_token,
            :expires => self.current_user.remember_token_expires_at }
      end

      redirect_back_or_default(:controller => '/account',
                               :action => 'index')

      flash[:notice] = "Logged in successfully"
    end
  end

  def signup
    @user = User.new(params[:user])
    return unless request.post?

    @user.save!
    self.current_user = @user
```

```
    redirect_back_or_default(:controller => '/account',
                             :action => 'index')
    flash[:notice] = "Thanks for signing up!"
    rescue ActiveRecord::RecordInvalid
      render :action => 'signup'
  end

  def logout
    self.current_user.forget_me if logged_in?
    cookies.delete :auth_token
    reset_session
    flash[:notice] = "You have been logged out."
    redirect_back_or_default(:controller => '/account',
                             :action => 'index')
  end
end
```

Обратите внимание на важное напоминание в комментарии в начале файла. Две строчки кода необходимо переместить из этого файла в код контроллера приложения:

```
class ApplicationController < ActionController::Base
  include AuthenticatedSystem
```

Сначала необходимо включить в класс `ApplicationController` модуль `AuthenticatedSystem`, чтобы содержащиеся в нем методы были доступны всем контроллерам вашей системы. Модуль `AuthenticatedSystem` предоставляет методы чтения и изменения атрибута `current_user` (хранящегося в сеансе). Кроме того, в нем имеется очень полезный метод `logged_in?`. Эти методы можно вызвать в заголовке макета приложения, например, чтобы показывать ссылку Вход или Выход в зависимости от того, аутентифицирован пользователь или нет:

```
<div id="login_message">
 <% if logged_in? -%>
  <%= "Вошел как <strong>#{h(current_user.name)}</strong>" %> |
  <%= link_to "Logout", :controller => 'account', :action => 'logout' %>
 <% else -%>
  <%= link_to "Logout", :controller => 'account', :action => 'login' %>
|
  <%= link_to "Signup", :controller => 'account', :action => 'signup' %>
 <% end -%>
</div>
```

Это ставит перед нами интересный вопрос, ответ на который сразу не очевиден: «Как получается, что `current_user` и `logged_in?` доступны в представлении?» Методы класса `Controller` (а они таковыми являются, поскольку примешаны из модуля `AuthenticatedSystem`) доступны в представлении, только если объявлены как помощники. Но при беглом взгляде на файл `application.rb` мы не видим ни одного обращения к `helper_method`.

Ответ кроется в методе `self.included`, который находится примерно в середине файла `authenticated_system.rb`:

```
# Реализация точки расширения, делающая #current_user и #logged_in?
# доступными в качестве методов-помощников ActionView.
def self.included(base)
  base.send :helper_method, :current_user, :logged_in?
end
```

Вот оно! Лично я предпочитаю помещать точки расширения `included` исключительно в начало модулей, поскольку, если они сразу не бросаются в глаза, потом возможны всяческие недоразумения.

Поясню, что здесь происходит. Когда модуль `AuthenticatedSystem` включается в какой-то класс, эта точка расширения вызывается в контексте класса объекта, выполняющего включение, – вызов `helper_method` происходит так, будто он был встроен в исходный класс. Магия метапрограммирования? Нет, просто надлежащее использование модулей Ruby, очень характерное для Пути Rails.

Получение имени пользователя из cookies

Далее следует необязательный фильтр, который позволяет входить в приложение, пользуясь присланным броузером cookies; в комментарии это названо «функциональностью "запомни меня"». Предположим, что мы хотим предоставить пользователям такую возможность, и перенесем в класс `ApplicationController` также строчку `before_filter :login_from_cookie`. Полюбопытствуем, что в действительности делает метод `login_from_cookie` из модуля `AuthenticatedSystem`:

```
# При вызове с включенным фильтром before_filter :login_from_cookie
# метод проверяет наличие cookies before_filter :login_from_cookie и
# и, если возможно, аутентифицирует пользователя.
def login_from_cookie
  return unless cookies[:auth_token] && !logged_in?
  user = User.find_by_remember_token(cookies[:auth_token])
  if user && user.remember_token?
    user.remember_me
    self.current_user = user
    cookies[:auth_token] = {
      :value => self.current_user.remember_token,
      :expires => self.current_user.remember_token_expires_at
    }
    flash[:notice] = "Logged in successfully"
  end
end
```

Этот код неплохо воспринимается, но все же разберем его подробнее, обратив внимание еще на один пример идиоматического использования Ruby и Rails, а также на работу с объектом `cookies`.

В первой строчке демонстрируется правильное использование необязательного ключевого слова `return` в Ruby: мы сразу же выходим из функции, если маркера `:auth_token` нет или пользователь уже вошел в систему. Важно не тратить время впустую, поскольку эта строка выполняется при каждом запросе.

Следующий шаг – получение маркера `:auth_token` из cookies, чтобы можно было поискать пользователя в базе данных. Способ структурирования этого предложения `if` – широко распространенная идиома. Оператор `&&` выполняет «закорачивание», то есть в данном случае возвращает `false`, если `user` равно `nil`. В результате мы избегаем исключения `NoMethodError` при попытке вызвать метод `remember_token`, когда объекта нет. Вы будете часто встречать такую идиому в программах для Rails и должны научиться пользоваться ею в собственном коде.

Если условие выполнено, мы вызываем метод `remember_me` для объекта `user` и делаем его текущим пользователем (`current_user`), что, собственно, и составляет смысл действия «вход в систему». И наконец, обновляются атрибуты cookies и в хеш `flash` заносится сообщение «Logged in successfully» (Вы успешно вошли в систему), которое будет показано пользователю.

Текущий пользователь

Мои повернутые на Java мозги иногда сбиваются, когда я вижу строки вида `self.current_user = user`. «С какой стати мы должны делать текущего пользователя *переменной экземпляра контроллера*?!!» Дело в том, что реализация методов чтения и изменения атрибута `current_user` *не совсем прямолинейна*. В них присутствует некая логика, хотя и выраженная в виде очень лаконичного кода на Ruby.

```
# Получает текущего пользователя из сеанса.
def current_user
  @current_user ||=
  (session[:user] && User.find_by_id(session[:user])) || :false
end

# Сохраняет заданного пользователя в сеансе.
def current_user=(new_user)
  session[:user]=
    (new_user.nil? || new_user.is_a?(Symbol)) ? nil : new_user.id
  @current_user = new_user
end
```

Ваша реализация `GuestUser` может зависеть от специфики приложения. Возможно, вы захотите повторить интерфейс реального объекта `User`, оставив атрибуты незаполненными.

Другой подход – определить в классе `GuestUser` обратный вызов `method_missing`, который будет возбуждать исключение `LoginRequiredError`, пе-

Говорит Уилсон...

Видимо, текстовый редактор Рика берет с него плату за каждую строчку. Читая код, вы с удивлением замечаете, что в сеансе сохраняется идентификатор текущего пользователя, а вовсе не сам объект `user`. Сохранение идентификатора и поиск по нему в базе данных считается в Rails рекомендованной практикой из-за проблем, связанных с сериализацией объектов и синхронизации данных.

Кроме того, идиоматическое использование оператора `||=` гарантирует, что текущий пользователь считывается из базы лишь один раз при каждом запросе – будучи прочитан, он кэшируется в переменной экземпляра контроллера. Кстати, диспетчер Rails создает новый экземпляр контроллера при поступлении каждого запроса, поэтому не нужно беспокоиться о том, что хранение `current_user` в переменной экземпляра небезопасно.

По своему опыту могу сказать, что возврат `:false` из метода `current_user`, когда пользователь не зарегистрирован, – сущее наказание. Думаете, я преувеличиваю? Тогда попробуйте сами в большом приложении повсюду расставить проверки `if logged_in?`, чтобы избежать ошибок `NoMethodErrors`, когда пользователь не аутентифицирован.

Имеет смысл переопределить в своем классе `ApplicationController` версию `current_user`, полученную от модуля `AuthenticatedSystem`. Реализация может остаться почти такой же, но вместо `:false` будет возвращать *фиктивный объект* `GuestUser`.

```
# Получает текущего пользователя из сеанса.
def current_user
  @current_user ||=
  (session[:user] && User.find_by_id(session
:user])) || GuestUser.new
end
```

рехватываемое системой аутентификации. Идея в том, чтобы автоматически предложить зарегистрироваться перед предоставлением доступа к запрошенному ресурсу, а не кодировать в каждом случае явную проверку или, еще того хуже, сообщать об ошибке сервера.

Протоколирование в ходе тестирования

Рик предоставляет нам также Ruby-модуль `AuthenticatedTestHelper`, который можно подмешать в класс `TestUnit` из файла `test_helper.rb`, чтобы содержащиеся в нем методы были доступны всем тестам:

```
class Test::Unit::TestCase
  include AuthenticatedTestHelper
```

Самый важный метод в этом модуле называется `login_as`. Его можно вызывать из метода `setup` или отдельной ветви теста для установления сеанса. Передайте методу объект `User`, которому должен принадлежать сеанс. А если вы вместо этого передадите символ, то `login_as` будет обращаться к фикстурам, описывающим пользователей (`test/fixtures/users.yml`) – вот где фикстуры могут здорово пригодиться:

```
def setup
  login_as(:quentin) # quentin was added by acts_as_authenticated
```

Модуль `AuthenticatedTestHelper` содержит также метод `authorize_as`, который имитирует базовую HTTP-аутентификацию, а не просто записывает переданный объект пользователя в атрибут `current_user` для текущего сеанса. Метод `authorize_as` можно применять для тестирования действий контроллера, которые будут играть роль веб-служб, а также пользователей, аутентифицирующихся с помощью механизма HTTP, а не через форму.

Наконец, тестовые методы-помощники `assert_requires_login` и `assert_http_authentication_required` принимают блок и позволяют проверить, что указанные действия контроллера действительно заставляют пользователя выполнить вход через форму или пройти базовую HTTP-аутентификацию.

Заключение

Почти во всех приложениях Rails необходима какая-то форма регистрации и контроля доступа. Именно поэтому так полезно научиться работать с подключаемым модулем Acts as Authenticated, которому в основном и посвящена эта глава. Помимо входящего в состав модуля генератора кода и класса `User`, мы также познакомились с генерируемым контроллером `Account`, узнали, как войти в систему на основе полученного cookies и получить доступ к текущему пользователю из любого места приложения.

15

XML и ActiveResource

Структура – ничто, если кроме нее ничего нет.
Скелеты только пугают людей, когда разгуливают сами по себе.
Мне странно, почему это не относится к XML.

Эрик Наггем

XML не пользуется уважением в сообществе Rails. Он слишком отдает «корпоративностью». В мире Ruby гораздо большего внимания удостоился другой язык разметки, YAML (Yet Another Markup Language). Однако во многих приложениях применение XML – объективная реальность, особенно если речь заходит об интероперабельности с другими системами. К счастью, Ruby on Rails предоставляет достаточную функциональность, относящуюся к XML.

В этой главе обсуждается, как генерировать и разбирать XML-документы в приложениях Rails. Начнем мы с детального рассмотрения метода `to_xml`, которым обладают многие объекты в Rails.

Метод to_xml

Иногда нужно просто получить XML-представление объекта, и модель `ActiveRecord` дает простой автоматический способ генерации XML с помощью метода `to_xml`. Поэкспериментируем с этим методом в консоли и посмотрим, что он умеет делать.

Запускаю консоль для демонстрационного приложения, относящегося к работе над книгами, и нахожу объект `ActiveRecord`, которым буду манипулировать:

```
>> Book.find(:first)
=> #<Book:0x264ebf4 @attributes={"name"=>"Professional Ruby on Rails
Developer's Guide", "uri"=>nil, "updated_at"=>2007-07-02T13:58:19- 05:00,
"text"=>nil, "created_by"=>nil, "type"=>"Book", "id"=>"1",
"updated_by"=>nil, "version"=>nil, "parent_id"=>nil, "position"=>nil,
"state"=>nil, "created_at"=>2007-07-02T13:58:19-05:00}>
```

Ага, вот он – экземпляр класса `Book`. Посмотрим на его обобщенное XML-представление:

```
>> Book.find(:first).to_xml
=> "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<book>\n <created-at
type=\"datetime\">2007-07-02T13:58:19-05:00</created-at>\n <created-by
type=\"integer\">\n </created-by>\n <id type=\"integer\">\n1 </id>\n
<name>Professional Ruby on Rails Developer's Guide</name>\n <parent-id
type=\"integer\">\n </parent-id>\n <position type=\"integer\">\n
</position>\n <state></state>\n <text>Empty</text>\n <updated-at
type=\"datetime\">2007-07-02T13:58:19-05:00</updated-at>\n <updated-by
type=\"integer\">\n </updated-by>\n <uri></uri>\n <version
type=\"integer\">\n </version>\n</book>\n"
```

Н-да, не фонтан. Нам поможет имеющаяся в Ruby функция `print`:

```
>> print Book.find(:first).to_xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
  <created-by type="integer"></created-by>
  <id type="integer">1</id>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <parent-id type="integer"></parent-id>
  <position type="integer"></position>
  <state></state>
  <text>Empty</text>
  <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
  <updated-by type="integer"></updated-by>
  <uri></uri>
  <version type="integer">
  </version>
</book>
```

Намного лучше! Так что же мы здесь имеем? Похоже на довольно прямолинейную сериализацию экземпляра `Book` в формате XML.

Настройка результата работы to_xml

В начале находится стандартная команда обработки, за ней – тег, соответствующий имени класса объекта. Свойства представлены в виде под-

элементов, а для нестроковых полей включен атрибут type. Но это всего лишь поведение, принимаемое по умолчанию, – мы можем настроить его с помощью дополнительных параметров метода to_xml.

Оставим в XML-представлении только название и URI книги, воспользовавшись параметром only. Передается он в составе уже хорошо знакомого хеша параметров, причем значением параметра :only является массив:

```
>> print Book.find(:first).to_xml(:only => [:name,:uri])
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <uri></uri>
</book>
```

Следуя принятым в Rails соглашениям, имеется и противоположный only параметр except, который оставляет все свойства, кроме указанных.

Что, если фрагмент XML, содержащий название и URI книги, требуется включить в другой документ? В этом случае воспользуемся параметром skip_instruct, который позволяет избавиться от команды обработки:

```
>> print Book.find(:first).to_xml(:skip_instruct => true, :only =>
[:name,:uri])
<book>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <uri></uri>
</book>
```

Мы можем изменить корневой элемент XML-представления Book и сделать отступ шириной не два, а четыре пробела. В этом нам помогут параметры root и indent соответственно:

```
>> print Book.find(:first).to_xml(:root => 'textbook', :indent => 4)
<?xml version="1.0" encoding="UTF-8"?>
<textbook>
    <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
    <created-by type="integer"></created-by>
    <id type="integer">1</id>
    <name>Professional Ruby on Rails Developer's Guide</name>
    <parent-id type="integer"></parent-id>
    <position type="integer"></position>
    <state></state>
    <text>Empty</text>
    <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
    <updated-by type="integer"></updated-by>
    <uri></uri>
    <version type="integer">
    </version>
</textbook>
```

По умолчанию Rails преобразует имена атрибутов в ВерблюжьейНотации и с подчерками в строки, где слова разделены дефисами, напри-

мер: created-at и parent-id. Но можно задать режим разделения слов подчерками, если присвоить параметру dasherize значение false:

```
>> print Book.find(:first).to_xml(:dasherize => false, :only =>
[:created_at,:created_by])
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created_at type="datetime">2007-07-02T13:58:19-05:00</created_at>
  <created_by type="integer"></created_by>
</book>
```

В примере выше включен атрибут type. Параметр skip_types позволяет опустить его:

```
>> print Book.find(:first).to_xml(:skip_types => true, :only =>
[:created_at,:created_by])
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created-at>2007-07-02T13:58:19-05:00</created-at>
  <created-by></created-by>
</book>
```

Ассоциации и метод to_xml

До сих пор мы ограничивались только базовой функциональностью ActiveRecord, игнорируя ассоциации. Ну а если нужно включить в XML-представление не только саму книгу, но и описание ее глав? Как раз для этого Rails предлагает параметр :include. Его значением является массив ассоциаций:

```
>> print Book.find(:first).to_xml(:include => :chapters)
<?xml version="1.0" encoding="UTF-8"?>
  <book>
  <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
  <created-by type="integer"></created-by>
  <id type="integer">1</id>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <parent-id type="integer"></parent-id>
  <position type="integer"></position>
  <state></state>
  <text>Empty</text>
  <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
  <updated-by type="integer"></updated-by>
  <uri></uri>
  <version type="integer">
  </version>
  <chapters>
    <chapter>
      <name>Introduction</name>
      <uri></uri>
    </chapter>
    <chapter>
```

```
      <name>Your Rails Decision</name>
      <uri></uri>
    </chapter>
  </chapters>
</book>
```

Метод `to_xml` способен работать с любым массивом, коль скоро каждый его элемент отвечает на сообщение `to_xml`. Если же попытаться вызвать его для массива, элементы которого не отвечают на это сообщение, мы получим такой результат:

```
>> [:cat, :dog, :ferret].to_xml
RuntimeError: Not all elements respond to to_xml
    from /activesupport/lib/active_support/core_ext/array/
    conversions.rb:48:in `to_xml'
    from (irb):6
```

В отличие от массивов, хеши Ruby имеют естественное представление в формате XML – ключи соответствуют именам тегов, а значение – содержимому. Rails автоматически вызывает метод `to_s`, чтобы получить строковое представление значения:

```
>> print ({:pet => 'cat'}.to_xml)
<?xml version="1.0" encoding="UTF-8"?>
<hash>
  <pet>cat</pet>
</hash>
```

Метод `to_xml` для объектов `Array` и `Hash` принимает одни и те же аргументы, за исключением `:include`.

Продвинутое применение метода to_xml

По умолчанию метод `to_xml` в `ActiveRecord` сериализует только хранимые атрибуты. Однако бывают случаи, когда нужно включить в XML-представление также нехранимые, выводимые или вычисляемые атрибуты. Например, в модели `Book` мог бы быть метод, возвращающий среднее количество страниц в одной главе:

```
class Book < ActiveRecord::Base

  def pages_per_chapter
    self.pages / self.chapters.length
  end
end
```

Для учета этого метода в ходе XML-сериализации следует задать параметр `:methods`:

```
>> print Book.find(:first).to_xml(:methods => :pages_per_chapter)
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
```

```
  <created-by type="integer"></created-by>
  <id type="integer">1</id>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <parent-id type="integer"></parent-id>
  <position type="integer"></position>
  <state></state>
  <text>Empty</text>
  <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
  <updated-by type="integer"></updated-by>
  <uri></uri>
  <version type="integer"></version>
  <pages-per-chapter>45</pages-per-chapter>
</book>
```

Можно также передать в параметре `methods` массив имен методов, которые следует вызывать.

Динамические атрибуты

При необходимости включить дополнительные элементы, не входящие в состав сериализуемого объекта, можно задать параметр `:procs` и передать в нем один или несколько Proc-объектов. При вызове им будет передан хеш параметров метода `to_xml`, с помощью которого можно получить доступ к объекту `XmlBuilder` (класс `XmlBuilder` — это основное средство генерации XML в Rails, мы рассмотрим его ниже в этой главе):

```
>> copyright = Proc.new {|opts|
opts[:builder].tag!('copyright','2007')}

>> print Book.find(:first).to_xml(:procs => [copyright])
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
  <created-by type="integer"></created-by>
  <id type="integer">1</id>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <parent-id type="integer"></parent-id>
  <position type="integer"></position>
  <state></state>
  <text>Empty</text>
  <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
  <updated-by type="integer"></updated-by>
  <uri></uri>
  <version type="integer"></version>
  <color>blue</color >
</book>
```

К сожалению, у этой техники есть одно странное ограничение: Proc-объектам не передается сама сериализуемая запись, поэтому таким путем можно добавить лишь данные, внешние по отношению к объекту.

Переопределение метода to_xml

Иногда для представления данных в формате XML требуется сделать что-то совсем необычное. В таких случаях можно создать XML-разметку вручную:

```
class Book < ActiveRecord::Base

  def to_xml(options = {})
    xml = options[:builder] || Builder::XmlMarkup.new(options)
    xml.instruct! unless options[:skip_instruct]
    xml.book do
      xml.tag!(:color, 'red')
    end
  end
  ...
end
```

Получается следующий результат:

```
>> print Book.find(:first).to_xml
<?xml version="1.0" encoding="UTF-8"?><book><color>red</color></book>
```

Уроки реализации метода to_xml в классе Array

Метод `to_xml` в классе `Array` дает отличный пример выразительности и элегантности, которых можно достичь при программировании на Ruby. Разберем код, который является составной частью добавленных Rails расширений класса `Array` и находится в файле `core_ext/array/conversions.rb` подсистемы `ActiveSupport`:

```
def to_xml(options = {})
  raise "Not all elements respond to to_xml" unless all? { |e|
    e.respond_to? :to_xml }
```

Видите, насколько текст первой строки близок к обычному английскому языку? Проверка элементов массива в методе `to_xml` показывает, каким удобным для восприятия и элегантным может быть код на Ruby. Вы должны стремиться к тому же и в собственных программах.

Далее мы видим, как Rails определяет имя объемлющего тега:

```
options[:root]||= all? { |e|
  e.is_a?(first.class) && first.class.to_s != "Hash" } ?
  first.class.to_s.underscore.pluralize : "records"
```

Обратите внимание на укороченный оператор присваивания `||=`. В нем используется либо переданное значение `options[:root]`, либо имя корневого элемента вычисляется. Такой стиль условного присваивания – широко распространенная идиома в Ruby, вы должны к ней привыкнуть. Если `options[:root]` равно `nil`, сначала проверяется, все ли элементы являются экземплярами одного и того же класса (и при этом не являются хешами). Если это условие истинно, то есть тип всех элемен-

тов совпадает с типом первого элемента массива, то имя корневого элемента генерируется с помощью следующего выражения: `first.class.to_s.underscore.pluralize`.

В противном случае корневой тег по умолчанию будет называться `"records"`, хотя в документации по Rails API этот факт не отражен. Просматривая данный код, я задался вопросом: «На что ссылается переменная `first`?»

Потом я вспомнил, что код выполняется в контексте экземпляра `Array`, поэтому `first` – просто вызов метода, возвращающего первый элемент массива.

Изящно. Но перейдем к следующей строчке метода `to_xml`, которая управляет выбором имени для тегов элементов массива: `options[:children] ||= options[:root].singularize`.

Тут все просто. Если конфигурация не была явно изменена, то Rails просто использует имя корневого элемента в единственном числе. Одной из первых вещей, которые мы узнаем о Rails, является тот факт, что `ActiveRecord` автоматически определяет формы единственного и множественного числа имен классов и таблиц баз данных. Но многие лишь гораздо позднее осознают, насколько широкое применение класс `Inflector` находит в других частях Rails. Надеюсь, что этот пример убедил вас, как важно действовать заодно с инфлектором, а не идти ему наперекор, задавая имена вручную.

Что можно сказать о величине отступа? По умолчанию она равна двум пробелам: `options[:indent] ||= 2`.

Дальше становится интереснее. В следующей строке мы видим, что `to_xml` пользуется классом `Builder::XmlMarkup` для генерации XML-разметки.

```
options[:builder] ||=
  Builder::XmlMarkup.new(:indent => options[:indent])
```

Параметр `:builder` позволяет передать существующий экземпляр `Builder`, а не создавать новый. Важность этого параметра станет очевидной позже, когда мы перейдем к обсуждению интеграции метода `to_xml` в специализированные процедуры генерации XML:

```
root = options.delete(:root).to_s
children = options.delete(:children)
```

Эти значения понадобятся нам для построения имен тегов корневого и дочерних элементов, поэтому запомним их и одновременно удалим из хеша `options`. Это первый признак того, что хеш `options` будет повторно использован в другом вызове (когда придет время генерировать XML-разметку дочерних элементов):

```
if !options.has_key?(:dasherize) || options[:dasherize]
  root = root.dasherize
end
```

Параметр :dasherize по умолчанию равен true, и это разумно, так как по соглашению, принятому в XML, составные имена тегов разделяются дефисами. Своей элегантностью код Rails в огромной степени обязан тому, что одни библиотеки строятся на базе других. В данном случае это демонстрируется на примере метода со смешным названием dasherize.

Следуя дальше, мы встречаем параметр :instruct, который уже обсуждался выше. В классе Builder имеется метод instruct!, который вставляет команду обработки XML. Разумеется, второй раз вставлять ее не следует, поэтому сразу после первой вставки параметр :skip_instruct в хеше options принудительно устанавливается в true, так как этот хеш будет и далее использоваться при рекурсивных вызовах:

```
options[:builder].instruct! unless options.delete(:skip_instruct)
opts = options.merge({:skip_instruct => true, :root => children })
```

Наконец, мы вызываем метод tag! объекта Builder, он выводит тег корневого элемента и сразу же рекурсивно вызывает to_xml (в итераторе each) для вывода разметки дочерних элементов:

```
  options[:builder].tag!(root) { each { |e| e.to_xml(opts) } }
end
```

Класс XML Builder

В предыдущем разделе мы говорили, что для генерации XML-разметки Rails пользуется внутренним классом Builder::XmlMarkup. Если метода to_xml недостаточно, и вы хотите сгенерировать специальную разметку, можете пользоваться экземплярами класса Builder напрямую. К счастью, Builder API – одна из самых мощных библиотек Ruby и очень простая в применении, стоит только освоиться.

В документации по API говорится: «Все (ну почти все) методы объекта XmlMarkup транслируются в эквивалентную XML-разметку. Любой метод с блоком трактуется как тег XML с вложенной разметкой, генерируемой блоком». Это очень сжатое описание принципа работы Builder, но примеры, тоже взятые из документации по API класса Builder, помогут нам разобраться.

Пусть переменная xm – экземпляр класса Builder::XmlMarkup:

```
xm.em("emphasized")                  # => <em>emphasized</em>
xm.em { xm.b("emp & bold") }         # => <em><b>emph &amp; bold</b></em>

xm.a("foo", "href"=>"http://foo.org")
                                     # => <a href="http://foo.org">foo</a>

xm.div { br }                        # => <div><br/></div>

xm.target("name"=>"foo", "option"=>"bar")
                                     # => <target option="foo" name="bar"/>
```

```
xm.instruct!                          # <?xml version="1.0" encoding="UTF-8"?>

xm.html {                             # <html>
  xm.head {                           #   <head>
    xm.title("History")               #     <title>History</title>
  }                                   #   </head>

  xm.body {                           #   <body>
    xm.comment! "HI"                  #     <!-- HI -->
    xm.h1("Header")                   #     <h1>Header</h1>
    xm.p("paragraph")                 #     <p>paragraph</p>
  }                                   #   </body>
}                                     # </html>
```

Чаще всего объект `Builder::XmlBuilder` применяется для вывода XML-документа в ответ на запрос. Мы уже говорили о переопределении метода `to_xml` в классе `ActiveRecord` для генерации собственной разметки. Другой способ (хотя и не рекомендуемый) заключается в использовании XML-шаблона.

Мы могли бы изменить метод `show` в классе `BooksController`, чтобы он работал с XML-шаблоном. Для этого вместо:

```
def BooksController < ApplicationController
  ...
  def show
    @book = Book.find(params[:id])
    respond_to do |format|
    format.html
    format.xml { render :xml => @book.to_xml }
  end
  ...
end
```

нужно было бы написать:

```
def BooksController < ApplicationController
  ...
  def show
    @book = Book.find(params[:id])
    respond_to do |format|
    format.html
    format.xml
  end
  ...
end
```

Теперь Rails будет искать файл `show.xml.builder` в каталоге `RAILS_ROOT/views/books`. Этот файл содержит следующий код:

```
xml.book {
  xml.title @book.title
  xml.chapters {
```

```
    @book.chapters.each { |chapter|
      xml.chapter {
        xml.title chapter.title
      }
    }
  }
}
```

В данном шаблоне `xml` – экземпляр `Builder::XmlMarkup`. Как и в случае ERb-представлений, у нас есть доступ к переменным экземпляра, установленным в контроллере, в данном случае – к переменной `@book`. Использование класса `Builder` в представлении дает удобный способ генерации XML.

Разбор XML

В дистрибутив Ruby включена полнофункциональная библиотека REXML для работы с XML. Рассмотрение ее во всех деталях выходит за рамки этой книги. Но если ваши потребности не слишком велики, например ограничиваются интерпретацией ответов от веб-служб, можно воспользоваться простыми средствами разбора XML, встроенными в Rails.

Преобразование XML в хеши

Rails позволяет превратить произвольный фрагмент XML-документа в хеши Ruby с помощью метода `from_xml`, добавленного в класс `Hash`.

Для демонстрации я сформирую простенькую строку в формате XML и преобразую ее в хеш:

```
>> xml = <<-XML
<pets>
  <cat>Franzi</cat>
  <dog>Susie</dog>
  <horse>Red</horse>
</pets>
XML

>> Hash.from_xml(xml)
=> {"pets"=>{"horse"=>"Red", "cat"=>"Franzi", "dog"=>"Susie"}}
```

У метода `from_xml` нет необязательных параметров. Вы можете не передавать ему никакого аргумента, передать строку в формате XML или объект `IO`. Если вообще ничего не передано, метод `from_xml` ищет файл с именем `scriptname.xml` (или, более точно, `$0.xml`). В Rails это не имеет особого смысла, но может пригодиться, если вы пользуетесь данной функциональностью в сценариях, не связанных с обработкой HTTP-запросов.

Чаще передается строка, как в предыдущем примере, или объект IO. Последнее особенно полезно при необходимости разобрать XML-файл:

```
>> Hash.from_xml(File.new('pets.xml')
=> {"pets"=>{"horse"=>"Red", "cat"=>"Franzi", "dog"=>"Susie"}}
```

Библиотека XmlSimple

Для преобразования XML в объект Hash Rails пользуется библиотекой XmlSimple:

```
class Hash

  ...

  def from_xml(xml)
    typecast_xml_value(undasherize_keys(XmlSimple.xml_in(xml,
          'keeproot' => true,
          'forcearray' => false,
          'forcecontent' => true,
          'contentkey' => '__content__')
        ))
  end
  ...
end
```

При обращении к XmlSimple Rails задает четыре параметра. Первый, :keeproot, говорит XmlSimple, что не нужно отбрасывать корневой элемент, в противном случае это было бы сделано по умолчанию:

```
>> XmlSimple.xml_in('<book title="The Rails Way" />', :keeproot =>
true)
=> { 'book' => [{'title' => 'The Rails Way'}]

>> XmlSimple.xml_in('<book title="The Rails Way" />', :keeproot =>
false)
=> {'title' => 'The Rails Way'}
```

Второму параметру присваивается значение :forcearray, в результате вложенные элементы представляются массивами, даже если существует всего один вложенный элемент. По умолчанию XmlSimple предполагает, что этот параметр равен true. Разница иллюстрируется в следующем примере:

```
>> XmlSimple.xml_in('<book><chapter index="1"/></book>', :forcearray =>
true)
=> {"chapter"=>[{"index"=>"1"}]}

>> XmlSimple.xml_in('<book><chapter index="1"/></book>', :forcearray =>
false)
=> {"chapter" => {"index"=> "1"}}
```

Третий параметр, :forcecontent, тоже устанавливается в true, и это означает, что в результирующий хеш будет добавлена пара ключ/значение с ключом content, даже если у разбираемого элемента нет ни содержимого, ни атрибутов. В результате элементы-братья представляются единообразно, что обеспечивает гораздо более удобную работу с хешем, как можно понять из следующего примера:

```
>> XmlSimple.xml_in('<book>
                      <chapter index="1">Слова</chapter>
                      <chapter>Числа</chapter>
                    </book>', :forcecontent => true)

=> {"chapter" => [{"content"=>"Слова", "index"=>"1"},
{"content"=>"Числа"}]}

>> XmlSimple.xml_in('<book>
                      <chapter index="1">Слова</chapter>
                      <chapter>Числа</chapter>
                    </book>', :forcecontent => false)

=> {"chapter" => [{"content"=>"Слова", "index"=>"1"}, "Числа"]}
```

Последний параметр – :contentkey. – XmlSimple по умолчанию использует для представления данных, содержащихся внутри элемента, строку "content". Rails изменяет ее на "__content__", чтобы уменьшить вероятность конфликта с именем настоящего XML-тега "content".

Приведение типов

Когда мы вызываем метод Hash.from_xml, в результирующем хеше не оказывается ключей "__content__". Куда же они делись? Rails не передает результат разбора с помощью XmlSimple напрямую программе, вызвавшей метод from_xml, а отправляет его методу typecast_xml_value, который преобразует строковые значения в правильные типы. Делается это с помощью атрибута type в элементах XML-документа. Например, вот как выглядит XML-документ, автоматически сгенерированный для объекта Book:

```
>> print Book.find(:first).to_xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <created-at type="datetime">2007-07-02T13:58:19-05:00</created-at>
  <created-by type="integer"></created-by>
  <id type="integer">1</id>
  <name>Professional Ruby on Rails Developer's Guide</name>
  <parent-id type="integer"></parent-id>
  <position type="integer"></position>
  <state></state>
  <text>Empty</text>
```

```
  <updated-at type="datetime">2007-07-02T13:58:19-05:00</updated-at>
  <updated-by type="integer"></updated-by>
  <uri></uri>
  <version type="integer">
  </version>
</book>
```

Среди прочего, метод `to_xml` устанавливает атрибуты `type`, которые определяют класс сериализуемого значения. Если подать этот XML-документ на вход методу `from_xml`, то Rails приведет строки к типам соответствующих объектов Ruby:

```
>> Hash.from_xml(Book.find(:first).to_xml)
=> {"book"=>{"name"=>"Professional Ruby on Rails Developer's Guide",
"uri"=>nil, "updated_at"=>Mon Jul 02 18:58:19 UTC 2007,
"text"=>"Empty", "created_by"=>nil, "id"=>1, "updated_by"=>nil,
"version"=>0, "parent_id"=>nil, "position"=>nil, "created_at"=>Mon Jul
02 18:58:19 UTC 2007, "state"=>nil}}
```

Библиотека ActiveResource

Веб-приложения часто должны обслуживать как пользователей, представленных броузерами, так и иные системы, предлагая некоторый API. В других языках эта задача решается с помощью протокола SOAP или формы XML-RPC, но в Rails применяется более простой подход. В главе 4 «REST, ресурсы и Rails» мы говорили о REST-совместимых контроллерах и использовании метода `respond_to` для возврата различных представлений ресурса. Это дает возможность обратиться по URL *http://localhost:3000/auctions.xml* и получить в ответ XML-представление всех аукционов в системе.

Мы можем написать клиента, потребляющего эти данные, с помощью библиотеки `ActiveResource`. Данная библиотека – стандартная часть дистрибутива Rails, заменившая `ActionWebService` (последняя по-прежнему доступна в виде gem-пакета). `ActiveResource` понимает все особенности REST-совместимой маршрутизации и XML-представлений. Чтобы воспользоваться ею в примере с аукционами, необходимо как минимум написать:

```
class Auction < ActiveResource::Base
self.site = 'http://localhost:3000'
end
```

Для получения списка аукционов вызовем метод `find`:

```
>> auctions = Auction.find(:all)
```

Библиотека `ActiveResource` по духу и интерфейсу очень напоминает `ActiveRecord`.

Метод find

В ActiveResource имеются такие же методы find, как в ActiveRecord (табл. 15.1). Отличаются они только тем, что вместо параметра :conditions используется :params.

Таблица 15.1. Методы find в библиотеке ActiveResource

ActiveRecord	ActiveResource	URL
`Auction.find(:all)`	`Auction.find(:all)`	GET http://localhost:3000/auctions.xml
`Auction.find(1)`	`Auction.find(1)`	GET http://localhost:3000/auctions/1.xml
`Auction.find(:first)`	`Auction.find(:first)`	GET http://localhost:3000/auctions.xml *получает весь список, затем вызывает для него метод first
`Auction.find(:all, :conditions => { :first_name => 'Matt')`	`Auction.find(:all, :params => { :first_name => 'Matt')`	GET http://localhost:3000/auctions.xml?first_name=Matt
`Item.find(:all, :conditions => { :auction_id => 6 })`	`Item.find(:all, :params => { :auction_id => 6 })`	GET http://localhosn :3000/auctions/6/items.xml
`Item.find(:all, :conditions => { :auction_id => 6, :used => true })`	`Item.find(:all, :params => { :auction_id => 6, :used => true })`	GET http://localhost:3000/auctions/6/items.xml?used=true

В двух последних примерах табл. 15.1 показывается, как использовать ActiveResource при вложенном ресурсе. Можно было бы также создать специальный метод used в контроллере items:

```
class ItemController < ActiveResource::Base

  def used
    @items = Item.find(:all,
              :conditions => {:auction_id => params[:auction_id],
                              :used => true })

    respond_to do |format|
      format.html
      format.xml { render :xml => @items.to_xml }
    end
  end
end
```

И добавить в файл routes.rb такой ресурс items:

```
map.resources :items, :member => {:used => :get }
```

Выполнив эти подготовительные шаги, мы получаем такой URL:

```
http://localhost:3000/auctions/6/items/used.xml
```

Обратиться к этому URL и получить стоящие за ним данные с помощью ActiveResource можно следующим образом:

```
>> used_items = Item.find(:all, :from => :used)
```

Написанный нами метод возвращает набор лотов – отсюда и параметр :all. Предположим, что у нас есть метод, возвращающий только самый последний размещенный лот, например:

```
class ItemController < ActiveResource::Base

  def newest
    @item = Item.find(:first,
              :conditions => {:auction_id => params[:auction_id]},
              :order => 'created_at DESC',
              :limit => 1)
    respond_to do |format|
      format.html
      format.xml { render :xml => @items.to_xml }
    end
  end
end
```

Тогда можно было бы написать следующий вызов:

```
>> used_items = Item.find(:one, :from => :newest)
```

Важно обратить внимание на то, как обрабатывается запрос к несуществующему лоту. Если мы попытаемся обратиться к лоту с id, равным –1 (такого нет), то получим HTTP-ответ с кодом 404. Получив такой ответ, ActiveResource возбудит исключение ResourceNotFound. Как мы увидим далее, ActiveResource очень активно пользуется кодами состояния HTTP.

Метод create

Библиотека ActiveResource может не только производить выборку данных, но и создавать их. Если бы мы хотели разместить с помощью ActiveResource новую заявку на лот, то должны были бы написать:

```
>> Bid.create(:username => 'me', :auction_id => 3, :item_id => 6,
:amount => 34.50)
```

В результате был бы отправлен POST-запрос на URL *http://localhost:3000/auctions/6/items/6.xml*, содержащий данные заявки. В контроллер следовало бы добавить такой код:

```
class BidController < ActiveResource::Base
  ...
  def create
    @bid = Bid.new(params[:bid])
    respond_to do |format|
```

```
      if @bid.save
        flash[:notice] = 'Заявка успешно создана.'
        format.html { redirect_to(@bid) }
        format.xml { render :xml => @bid, :status => :created,
:location => @bid }
      else
        format.html { render :action => "new" }
        format.xml { render :xml => @bid.errors, :status =>
:unprocessable_entity}
      end
    end
  end
  ...
end
```

Если заявка создана успешно, будет возвращен ее идентификатор и код состояния 201, а заголовок Location будет указывать на URL только что созданной заявки. Имея заголовок Location, мы можем узнать `id` новой заявки, например:

```
>> bid = Bid.create(:username => 'me', :auction_id => 3, :item_id =>
6, :amount => 34.50)
>> bid.id # => 12
>> bid.new? # => false
```

Если бы мы попытались создать еще одну заявку, не указав сумму предложения, то могли бы ознакомиться с сообщениями об ошибках:

```
>> bid = Bid.create(:username => 'me', :auction_id => 3, :item_id => 6)
>> bid.valid? # => false
>> bid.id # => nil
>> bid.new? # => true
>> bid.errors.class # => ActiveResource::Errors
>> bid.errors.size # => 1
>> bid.errors.on_base # => "Сумма не может быть пустой"
>> bid.errors.full_messages # => "Сумма не может быть пустой"
>> bid.errors.on(:amount) # => nil
```

В данном случае метод `create` вернул объект `Bid`, но в некорректном состоянии. Попытавшись узнать его `id`, мы получим `nil`. Обратившись к методу `ActiveResources.errors`, мы увидим, что привело к ошибке. Этот метод ведет себя, как `ActiveRecord.error`, но с одним важным отличием. В `ActiveRecord`, вызвав метод `Errors.on`, мы получим ошибку для данного атрибута. А в предыдущем примере вместо этого получается `nil`. Причина в том, что `ActiveResource`, в отличие от `ActiveRecord`, ничего не знает о существующих атрибутах. Чтобы получить эту информацию, `ActiveRecord` обращается к СУБД с запросом `SHOW FIELDS FROM <table>`, но в `ActiveResource` никакого эквивалента этому нет. `ActiveResource` может узнать, что атрибут существует, одним-единственным способом – от нас. Например:

```
>> bid = Bid.create(:username => 'me', :auction_id => 3, :item_id =>
6, :amount => nil)
```

```
>> bid.valid? # => false
>> bid.id # => nil
>> bid.new? # => true
>> bid.errors.class # => ActiveResource::Errors
>> bid.errors.size # => 1
>> bid.errors.on_base # => "Amount can't be blank"
>> bid.errors.full_messages # => "Сумма не может быть пустой"
>> bid.errors.on(:amount) # => "не может быть пустой"
```

В данном случае мы сообщили ActiveResource о существовании атрибута amount в методе create. Теперь можно вызывать Errors.on без опасений.

Метод update

Редактирование в ActiveResource устроено по аналогии с ActiveRecord:

```
>> bid = Bid.find(1)
>> bid.amount # => 10.50
>> bid.amount = 15.00
>> bid.save # => true
>> bid.reload
>> bid.amount # => 15.00
```

Если мы присвоим атрибуту amount значение nil, метод ActiveResource.save вернет false. В данном случае мы могли бы опросить набор ActiveResource::Errors о причинах ошибки точно так же, как в методе create. Существенное отличие между библиотеками ActiveResource и ActiveRecord — отсутствие в первой методов save! и update!.

Метод delete

Удаление в ActiveResource может происходить двумя способами. В первом не требуется создавать экземпляр ActiveResource:

```
>> Bid.delete(1)
```

При другом способе мы сначала создаем экземпляр ActiveResource:

```
>> bid = Bid.find(1)
>> bid.destroyAuthorization
```

В ActiveResource включена поддержка базовой HTTP-аутентификации. Напомню, что базовая аутентификация реализуется с помощью специального HTTP-заголовка, который легко подсмотреть. Поэтому следует работать на HTTPS-соединении. Если безопасное соединение установлено, то для аутентификации ActiveResource необходимо только передать имя и пароль пользователя:

```
Class MoneyTransfer < ActiveResource::Base
  self.site = 'https://localhost:3000'
  self.username = 'administrator'
  self.password = 'secret'
end
```

Теперь ActiveResource будет аутентифицироваться при каждом соединении. Если имя или пароль неверны, возбуждается исключение ActiveResource::ClientError. Реализовать базовую аутентификацию в контроллере можно с помощью подключаемого модуля:

```
$ ./script/plugin install http_authentication
```

Далее нужно настроить контроллер:

```
class MoneyTransferController < ApplicationController
  USERNAME, PASSWORD = "administrator", "secret"

  before_filter :authenticate

  ...
  def create
    @money_transfer = Bid.new(params[:money_transfer])
    respond_to do |format|
      if @money_transfer.save
        flash[:notice] = 'Перевод денег успешно осуществлен.'
        format.html { redirect_to(@money_transfer) }
        format.xml { render :xml => @money_transfer, :status =>
:created, :location => @money_transfer }
      else
        format.html { render :action => "new" }
        format.xml { render :xml => @money_transfer.errors, :status =>
:unprocessable_entity}
      end
    end
  end
  ...

  private
    def authenticate
      authenticate_or_request_with_http_basic do |username, password|
        username == USERNAME && password == PASSWORD
      end
    end
end
```

Заголовки

ActiveResource позволяет при каждом запросе задавать HTTP-заголовки. Это можно сделать двумя способами. Первый – подготовить заголовок в виде переменной:

```
Class Auctions< ActiveResource::Base
  self.site = 'http://localhost:3000'

  @headers = { 'x-flavor' => 'orange' }
end
```

В результате при каждом обращении к сайту будет посылаться заголовок HTTPX-FLAVOR: orange. В нашем контроллере мы могли использовать значение из заголовка:

```
class AuctionController < ActiveResource::Base
  ...
  def show
    @auction = Auction.find_by_id_and_flavor(params[:bid],
request.headers['HTTP_X_FLAVOR']) respond_to do |format|
      format.html
      format.xml { render :xml => @auction.to_xml }
    end
  end
  ...
end
```

Второй способ задать заголовки для `ActiveResource` — переопределить метод `headers`:

```
Class Auctions< ActiveResource::Base
  self.site = 'http://localhost:3000'

  def headers
    { 'x-flavor' => 'orange' }
  end
end
```

Настройка

`ActiveResource` предполагает, что URL совместимы с REST, но так бывает не всегда. К счастью, можно настроить префикс URL и атрибут `collection_name`. Пусть мы имеем дело с подклассом `ActiveResource`:

```
Class OldAuctionSystem < ActiveResource::Base
  self.site = 'http://s60:3270'

  self.prefix = '/cics/'
  self.collection_name = 'auction_pool'
end
```

Тогда будут использованы следующие URL:

`OldAuctionSystem.find(:all)` GET http://s60:3270/cics/auction_pool.xml
`OldAuctionSystem.find(1)` GET http://s60:3270/cics/auction_pool/1.xml
`OldAuctionSystem.find(1).save` PUT http://s60:3270/cics/auction_pool/1.xml
`OldAuctionSystem.delete(1)` DELETE http://s60:3270/cics/auction_pool/1.xml
`OldAuctionSystem.create(...)` POST http://s60:3270/cics/auction_pool.xml

Можно также изменить имя элемента, применяемое для генерации XML. В предыдущем примере метод `create` класса `OldAuctionSystem` в виде XML выглядел бы так:

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<OldAuctionSystem>
  <title>Auction A</title>
  ...
</OldAuctionSystem>
```

Для изменения имени элемента следовало бы написать:

```
Class OldAuctionSystem < ActiveResource::Base
  self.site = 'http://s60:3270'

  self.prefix = '/cics/'
  self.element_name = 'auction'
end
```

что привело бы к следующему результату:

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<Auction>
  <title>Auction A</title>
  ...
</Auction>
```

У задания атрибута `element_name` есть одно последствие: для генерации URL `ActiveResource` использует имя элемента во множественном числе. В данном случае это `'auctions'`, а не `'OldAuctionSystems'`. Если вы хотите сохранить старый URL, следует также задать атрибут `collection_name`.

Можно указать `ActiveResource` и поле первичного ключа:

```
Class OldAuctionSystem < ActiveResource::Base
  self.site = 'http://s60:3270'

  self.primary_key = 'guid'
end
```

Хешированные формы

Методы `Find`, `Create`, `Save` и `Delete` соответствуют глаголам HTTP `GET`, `POST`, `PUT` и `DELETE`. В `ActiveResource` тоже есть методы для каждого из этих глаголов. Они принимают те же аргументы, что `Find`, `Create`, `Save` и `Delete`, но возвращают хеш, в который преобразован полученный XML-документ. Например:

```
>> bid = Bid.find(1)
>> bid.class # => ActiveRecord::Base
>> bid_hash = Bid.get(1)
>> bid_hash.class # => Hash
```

Заключение

Методов `to_xml` и `from_xml` достаточно для обработки XML в большинстве практических ситуаций, с которыми сталкивается средний разработчик на платформе Rails. За их простотой скрывается немалая гибкость и мощь, и в этой главе мы постарались объяснить эти методы достаточно подробно, что вдохновить вас на дальнейшее изучение методов работы с XML в мире Ruby.

Методы `to_xml` и `from_xml` позволяют безо всякого труда создать среду, связывающую разные приложения Rails с помощью веб-служб. Такая среда называется `ActiveResource`, и вы прослушали краткий, но интенсивный курс по этому предмету.

16

ActionMailer

Это замечательный способ отправлять
электронную почту при минимальном объеме кода.

Джейк Скраггс[1]

Интеграция с электронной почтой – неотъемлемая часть большинства современных веб-приложений. Это и восстановление забытых паролей, и управление своими учетными записями посредством почтовых сообщений. Как бы то ни было, вы будете рады узнать, что благодаря `ActionMailer` Rails прекрасно поддерживает как отправку, так и получение электронных писем. В этой главе мы расскажем, что необходимо сделать на этапе развертывания, чтобы можно было отправлять и получать почту, и познакомимся с тем, как пишутся *модели почтальона* – так в Rails называются сущности, инкапсулирующие код работы с электронной почтой.

Конфигурирование

По умолчанию Rails пытается отправить почту по протоколу SMTP (порт 25) через сервер localhost. Если у вас Rails работает на машине, где запущен демон SMTP, и он локально принимает SMTP-почту, то для отправки электронных писем больше ничего делать не надо. Если

[1] *http://jakescruggs.blogspot.com/2007/02/actionmailer-tips.html.*

же на локальной машине SMTP не работает, следует решить, как система будет отправлять исходящую почту.

Когда SMTP не используются напрямую, остается два основных способа: воспользоваться программой *sendmail* или сообщить Rails, как соединиться с внешним почтовым сервером. В большинстве организаций имеются SMTP-серверы, предназначенные как раз для такого случая, хотя стоит отметить, что из-за спама многие провайдеры хостинга прекратили предоставлять разделяемую SMTP-службу.

Модели почтальона

Сконфигурировав почтовую систему, мы можем перейти к следующему шагу: создать модель почтальона (mailer model), в которой будет инкапсулирован код, относящийся в отправке и получению электронной почты. Rails предлагает для этой цели специальный генератор.

Для демонстрации создадим почтальона, который будет отправлять уведомления о нарушении сроков пользователям нашего приложения для управления временем и затратами:

```
$ script/generate mailer LateNotice
      exists app/models/
      create app/views/late_notice
      exists test/unit/
      create test/fixtures/late_notice
      create app/models/late_notice.rb
      create test/unit/late_notice_test.rb
```

Папка представлений для почтальона создается в каталоге `app/views/late_notice`, а заглушка самого почтальона находится в файле `app/models/late_notice.rb`:

```
class LateNotice < ActionMailer::Base
end
```

Как и в создаваемом по умолчанию подклассе `ActiveRecord`, начинки не так уж много. Ну а что насчет теста? Взгляните на листинг 16.1.

Листинг 16.1. Тест ActionMailer

```
require File.dirname(__FILE__) + '/../test_helper'

class LateNoticeTest < Test::Unit::TestCase

  FIXTURES_PATH = File.dirname(__FILE__) + '/../fixtures'
  CHARSET = "utf-8"

  include ActionMailer::Quoting

  def setup
    ActionMailer::Base.delivery_method = :test
```

```
    ActionMailer::Base.perform_deliveries = true
    ActionMailer::Base.deliveries = []
    @expected = TMail::Mail.new
    @expected.set_content_type "text", "plain", { "charset" => CHARSET }
    @expected.mime_version = '1.0'
  end

  private
    def read_fixture(action)
      IO.readlines("#{FIXTURES_PATH}/late_notice/#{action}")
    end

    def encode(subject)
      quoted_printable(subject, CHARSET)
    end
end
```

Однако! Код настройки куда объемнее, чем мы привыкли видеть, это отражает внутреннюю сложность работы с почтовой подсистемой.

Подготовка исходящего почтового сообщения

При работе с подклассами `ActionMailer` вы определяете открытые методы *почтальона*, соответствующие различным типам сообщений, которые собираетесь посылать. Внутри открытого метода вы устанавливаете параметры сообщения и присваиваете значения переменным, которые понадобятся шаблону сообщения.

Продолжая пример, напишем метод `late_timesheet`, принимающий параметры `user` и `week_of`. Отметим, что в нем задается основная информация, необходимая для отправки уведомления (листинг 16.2).

Листинг 16.2. Метод почтальона

```
def late_timesheet(user, week_of)
  recipients user.email
  subject "[Время и затраты] Уведомление о просроченном табеле"
  from "system@timeandexpenses.com"
  body :recipient => user.name, :week => week_of
end
```

Ниже приведен список всех относящихся к почте параметров, которые можно установить в методе почтальона.

attachment

Задает файл вложения. Можно вызывать несколько раз, чтобы включить более одного вложения.

bcc

Адреса получателей слепых копий данного сообщения (заголовок `Bcc:`) в виде строки (если адрес один) или массива строк.

body

Тело сообщения. Значением является хеш (в котором задаются значения переменных, передаваемых в шаблон) или строка (сам текст сообщения).

`ActionMailer` автоматически *нормализует* строки, когда содержимое письма является простым текстом, то есть ставит в конце строки знак `\n` вместо платформенно зависимого символа.

cc

Адреса получателей копии сообщения (заголовок `Cc:`) в виде строки (если адрес один) или массива строк.

charset

Набор символов для сообщения. По умолчанию совпадает со значением атрибута `default_charset` в классе `ActionMailer::Base`.

content_type

Тип содержимого сообщения. По умолчанию – `text/plain`.

from

Адрес отправителя сообщения в виде строки (обязательный параметр).

headers

Дополнительные заголовки, включаемые в сообщение, в виде хеша.

implicit_parts_order

Массив, задающий порядок сортировки частей многочастного сообщения. Выражен в терминах MIME-типа. По умолчанию совпадает со значением атрибута `default_implicit_parts_order` в классе `ActionMailer::Base`: `[ "text/html", "text/enriched", "text/plain" ]`.

mailer_name

Переопределяет имя почтальона, которое по умолчанию образуется как имя класса почтальона в единственном числе. Это имя используется для поиска шаблонов почтальона. Если вы хотите разместить шаблон в нестандартном месте, задайте этот параметр.

mime_version

По умолчанию `"1.0"`, но может быть задана явно.

part

Позволяет отправлять многочастные сообщения, определив для каждого тип содержимого, шаблон и переменные, подставляемые

в тело. Отметим, что задавать этот параметр необязательно, так как `ActionMailer` автоматически находит и использует многочастные шаблоны в предположении, что имя шаблона строится как имя действия, за которым следует тип содержимого.

Этот параметр также необходим, если вы пытаетесь отправить HTML-сообщение со встроенными вложениями (обычно графическими файлами). Дополнительную информацию см. в разделе «Многочастные сообщения» ниже, где также описан небольшой специальный API метода `part`.

recipients

Адреса получательна сообщения в виде строки (если адрес один) или массива строк. Напомним, что этому методу необходимо передавать именно *строковые* адреса, а не объекты приложения, представляющие пользователей.

```
recipients users.map(&:email)
```

sent_on

Необязателен дата отправки сообщения, обычно равная `Time.now`. Если данный параметр явно не указан, механизм отправки автоматически подставит значение.

subject

Тема сообщения.

template

Имя шаблона сообщения. Поскольку по умолчанию имя шаблона совпадает с именем метода почтальона, этот параметр можно использовать, когда общий шаблон разделяется несколькими методами почтальона.

Тело сообщения создается с помощью шаблона `ActionView` (обычный ERb-файл), для которого переменными экземпляра являются параметры, заданные в хеше `body`. Следовательно шаблон для метода почтальона из листинга 16.2 мог бы выглядеть так:

```
Уважаемый(ая) <%= @recipient %>,
Ваш табель на неделю с <%= @week %> просрочен.
```

И если бы получателем был Дэвид, сгенерировалось бы такое сообщение:

```
Date: Sun, 12 Dec 2004 00:00:00 +0100
From: system@timeandexpenses.com
To: david@loudthinking.com
```

```
Subject: [Time and Expenses] Late timesheet notice
Уважаемый(ая) Дэвид Хэнссон,
Ваш табель на неделю с Aug 15th просрочен.
```

Почтовые сообщения в формате HTML

Чтобы отправить сообщение в формате HTML, необходимо подготовить шаблон представления, порождающий HTML-разметку, и указать в методе почтальона MIME-тип `text/html`, как показано в листинге 16.3.

Листинг 16.3. Метод почтальона для отправки сообщения в формате HTML

```
class MyMailer < ActionMailer::Base

  def signup_notification(recipient)
    recipients   recipient.email_address_with_name
    subject      "Информация о новой учетной записи"
    body         "account" => recipient
    from         "system@example.com"
    content_type "text/html"
  end
end
```

Если не считать специального значения `content_type`, процедура точно такая же, как при отправке простого текстового сообщения. Хотите *внедрить* графику в HTML, чтобы она отправлялась в том же письме (в виде встроенных вложений) и показывалась получателю? На данный момент в `ActionMailer` имеется нерешенная проблема, которая затрудняет эту задачу. Дополнительную информацию и предлагаемую в качестве временного решения заплату см. на странице *http://dev.rubyonrails.org/ticket/2179*[1].

Многочастные сообщения

Метод `part` предлагает небольшой самостоятельный API для создания многочастных сообщений. С его помощью можно создавать сообщения, составленные из разнородных частей. Широко распространена практика (листинг 16.4) отправки простого текста и сообщения в формате HTML, чтобы получатели, которые могут читать только простой текст, не оставались обделенными.

---

[1] Отметим, что поиск в Google информации о встроенных вложениях графики обычно указывает на страницу *http://blog.caboo.se/articles/2006/02/19/how-to-send-multipart-alternative-e-mail-with-inlineattachments*, где приведено простое, но не работающее решение.

Листинг 16.4. Метод почтальона, отправляющий многочастное уведомление о регистрации

```
class ApplicationMailer < ActionMailer::Base

  def signup_notification(recipient)
    recipients recipient.email_address_with_name
    subject "Информация о новой учетной записи"
    from "system@example.com"

    part :content_type => "text/html",
    :body => render_message("signup_as_html", :account =>
recipient)

    part "text/plain" do |p|
      p.body = render_message("signup_as_plain", :account =>
recipient)
      p.transfer_encoding = "base64"
    end

  end
end
```

Параметры метода part

Метод `part` принимает различные параметры в виде хеша или блока (в листинге 16.4 демонстрируются оба способа инициализации):

- `:body` – представляет тело части *в виде строки*. *Хеш передавать нельзя* (в отличие от `ActionMailer::Base`). Если вы хотите, чтобы часть генерировалась по шаблону, можете вызвать метод `render` или `render_template` почтальона и передать возвращенный им результат (см. листинг 16.4);
- `:charset` – задает набор символов для части. По умолчанию совпадает с набором символов объемлющей части или самого почтальона (например, UTF8);
- `:content_type` – MIME-тип содержимого части;
- `:disposition` – диспозиция содержимого части, обычно inline или attachment;
- `:filename` – имя файла для данной части, обычно вложения. Указанное значение получатель увидит, когда попытается сохранить вложение; оно не имеет никакого отношения к именам файлов на вашем сервере;
- `:headers` – дополнительные заголовки, включаемые в часть, в виде хеша;
- `:transfer_encoding` – метод кодирования данной части, например `"base64"` `"quoted-printable"`.

Неявные многочастные сообщения

Выше уже упоминалось, что многочастные сообщения можно формировать и неявно, не вызывая метод `part`, поскольку `ActionMailer` автоматически находит и использует шаблоны частей, предполагая, что имя шаблона строится как имя действия, за которым следует тип содержимого. Каждый обнаруженный шаблон добавляется в виде отдельной части сообщения.

Например, если существуют перечисленные ниже шаблоны, будет выполнен рендеринг каждого, и результаты включатся в виде частей сообщения с соответствующим MIME-типом. Каждому шаблону передается один и тот же хеш `body`:

- signup_notification.text.plain.erb
- signup_notification.text.html.erb
- signup_notification.text.xml.builder
- signup_notification.text.x-yaml.erb

Вложение файлов

Присоединять вложения позволяет метод `attachment` в сочетании со стандартным методом Ruby `File.read` или блоком, генерирующим содержимое файла (листинг 16.5).

Листинг 16.5. Присоединение вложений к сообщению

```
class ApplicationMailer < ActionMailer::Base
  def signup_notification(recipient)
    recipients   recipient.email_address_with_name
    subject      "Информация о новой учетной записи"
    from         "system@example.com"

    attachment :content_type => "image/jpeg",
      :body => File.read("an-image.jpg")

    attachment "application/pdf" do |a|
      a.body = generate_your_pdf_here()
    end
  end
end
```

Метод `attachment` – не более чем удобная обертка вокруг API метода `part`. Первое вложение, показанное в листинге 16.5, можно было бы добавить (чуть менее элегантно) и так:

```
part :content_type => "image/jpeg",
     :disposition => "inline",
     :filename => "an-image.jpg",
```

```
    :transfer_encoding => "base64" do |attachment|
       attachment.body = File.read("an-image.jpg")
    end
```

Итак, мы подробно поговорили о подготовке почтового сообщения, но как его теперь отправить получателю?

Отправка почтового сообщения

Даже не пытайтесь вызывать методы экземпляра, например `signed_up`, напрямую. Вместо этого вызовите один из двух методов класса, сгенерированных на основе определенных в классе почтальона методов экземпляра. Эти методы имеют префиксы `deliver_` и `create_` соответственно. На самом деле, наибольший интерес для вас представляет метод `deliver_`.

Например, если бы вы написали в классе `ApplicationMailer` метод экземпляра с именем `signed_up_notification`, то использовать его нужно было бы следующим образом:

```
# создать объект tmail для тестирования
ApplicationMailer.create_signed_up_notification("david@loudthinking.com")

# послать уведомление о регистрации
ApplicationMailer.deliver_signed_up("david@loudthinking.com")

# А вот так неправильно!
ApplicationMailer.new.signed_up("david@loudthinking.com")
```

Получение почты

В состав дистрибутива Ruby еще с 2003 года входит библиотека `TMail` для работы с почтой. Она включена и в дистрибутив Rails в виде зависимости для `ActionMailer`. Вам как разработчику для Rails из библиотеки `TMail` интересен лишь класс `TMail::Mail`.

Для получения почты вы должны написать открытый метод `receive` в одном из подклассов `ActionMailer::Base`, входящих в ваше приложение. Он будет принимать единственный параметр – объект `Tmail`. Чтобы принять входящую почту, вы вызываете *метод класса* `receive`, определенный в вашем классе почтальона. Строковое представление полученного сообщения автоматически преобразуется в объект `Tmail` и передается методу `receive` для дальнейшей обработки. Метод *класса* `receive` реализовывать не нужно – он наследуется от `ActionMailer::Base`.

Объяснение получилось довольно путаное, проще показать на примере. Такой пример приведен в листинге 16.6.

Листинг 16.6. Простой класс почтальона MessageArchiver с методом receive

```
class MessageArchiver < ActionMailer::Base

  def receive(email)
    person = Person.find_by_email(email.to.first)
    person.emails.create(:subject => email.subject, :body =>
email.body)
  end

end
```

Метод *класса* `receive` может быть адресатом для агента передачи почты Postfix или иного процесса, способного передавать почту по конвейеру другому процессу. Входящий в состав Rails сценарий `runner` упрощает обработку входящей почты:

```
./script/runner 'MessageArchiver.receive(STDIN.read)'
```

При таком подходе метод *класса* `receive` получит строковое представление сообщения из стандартного входа `STDIN`.

Справка по TMail::Mail API

Поскольку входящее сообщение представляется в виде экземпляра `TMail::Message`, думаю, что будет уместно привести справочный материал хотя бы по основным атрибутам этого класса. Онлайновая документация по библиотеке `TMail` находится по адресу *http://i.loveruby.net/en/projects/tmail/doc/*, но приведенной ниже информации достаточно практически для всех надобностей.

attachments

Массив объектов `TMail::Attachment`, ассоциированных с объектом сообщения. Класс `TMail::Attachment` расширяет входящий в Ruby класс `StringIO`, добавляя в него атрибуты `original_filename` и `content_type`. Во всех остальных отношениях можете работать с ними точно так же, как с любым другим экземпляром `StringIO` (см. пример ниже в листинге 16.7).

body

Текст тела сообщения в предположении, что это одночастное сообщение, содержащее простой текст. При вызове метода `body` для многочастного сообщения будет возвращена *преамбула*.

date

Объект `Time`, соответствующий значению из заголовка `Date:`.

has_attachments?

Возвращает `true` или `false` в зависимости от того, содержит ли сообщение вложения.

multipart?

Возвращает `true`, если это многочастное сообщение.

parts

Массив объектов `TMail::Mail`, по одному для каждой части многочастного сообщения.

subject

Тема сообщения.

to

Массив строк, представляющих адреса получателей из заголовка `To:` сообщения. Атрибуты `cc`, `bcc` и `from` работают аналогично для остальных адресных полей.

Обработка вложений

Обработка файлов, присоединенных к входящему почтовому сообщению, сводится к чтению атрибута `attachments` объекта `TMail`, как показано в листинге 16.7. В примере предполагается, что есть класс `Person`, в котором определена ассоциация `has_many` с объектом `attachment_fu`, названная `photos`:

```
class PhotoByEmail < ActionMailer::Base

  def self.receive(email)
    from = email.from.first
    person = Person.find_by_email(from)
    logger.warn("Person not found [#{from}]") and return unless person

    if email.has_attachments?
      email.attachments.each do |file|
        person.photos.create(:uploaded_data => file)
      end
    end
  end
end
```

Больше по этому поводу сказать особо нечего, если, конечно, не считать вопроса о конфигурировании процессора почты (не являющегося частью Rails). А их, как известно, конфигурировать очень не-

просто[1]. Добившись того, что процессор почты будет правильно вызывать сценарий `runner`, добавьте в `crontab` задание, которое будет проверять входящую почту, скажем, раз в пять минут. Впрочем, интервал зависит от конкретного приложения.

Конфигурирование

Как правило, для отправки почты специально конфигурировать ничего не надо, поскольку на промышленном сервере уже установлена программа `sendmail`, и `ActionMailer` будет только рад воспользоваться ею для отправки сообщений.

Если же `sendmail` не установлена, можете попробовать настроить Rails так, чтобы отправлять почту напрямую по протоколу SMTP. В классе `ActionMailer::Base` имеется хеш `smtp_settings` (в версиях, предшествующих Rails 2.0, он назывался `server_settings`), в котором хранится конфигурационная информация. Настройки зависят от используемого SMTP-сервера. В примере (листинг 16.7) показаны все имеющиеся настройки и их значения по умолчанию. Вам следует добавить аналогичный код в свой файл `config/environment.rb`.

Листинг 16.7. Настройки SMTP для ActionMailer

```
ActionMailer::Base.smtp_settings = {
  :address => 'smtp.yourserver.com',     # по умолчанию localhost
  :port => '25',                         # по умолчанию 25
  :domain => 'yourserver.com',           # по умолчанию
localhost.localdomain
  :user_name => 'user',                  # умолчания нет
  :password => 'password',               # умолчания нет
  :authentication => :plain              # :plain, :login или :cram_md5
}
```

Заключение

В этой главе мы видели, как легко в Rails получать и отправлять почту. Написав сравнительно немного кода, вы можете наделить свое приложение возможностью отправлять почту даже в формате HTML со встроенными графическими вложениями. Получать почту еще проще, если отвлечься от настройки сценариев обработки почты и заданий `cron`. Мы также кратко рассмотрели относящиеся к почте конфигурационные параметры в файле `config/environment.rb`.

---

[1] Роб Орсини (Rob Orsini), автор книги *Rails Cookbook*, вышедшей в издательстве O'Reilly, рекомендует программу getmail, которую можно скачать по адресу *http://pyropus.ca/software/getmail*.

17

Тестирование

Не хочу сказать, что Rails заставляет вас вести разработку, управляемую тестами. Просто по-другому работать в Rails сложно.

Брайан Энг, из интервью в подкасте, посвященном Rails

Автоматизированные тесты позволяют проверять функциональность приложения, предотвращать регрессию (появление новых и ранее исправленных ошибок) и поддерживать гибкость кода. Под тестовым покрытием понимают количество и качество автоматизированных тестов, относящихся к эксплуатируемой программе. Если тестовое покрытие недостаточно, то нет уверенности в том, что система работает должным образом. Мы можем внести какое-то изменение, нарушающее правильную работу приложения, и не заметить ошибки до того момента, как диагностировать и исправить ее окажется гораздо труднее.

Невозможно переоценить важность тестирования кода на Ruby. В отсутствие компилятора нет уверенности даже в том, что в программе отсутствуют синтаксические ошибки! Вы обязаны предполагать, что код ошибочен, до тех пор пока он не подвергнется суровому испытанию. Предпочитаете, чтобы ошибка обнаружилась на машине для разработки, которая находится под вашим контролем и где диагностировать проблему легко? Или хотите узнать о ней уже после того, как развернутое приложение обрушит сервер и приведет в ярость начальников, коллег и конечных пользователей? Это серьезный вопрос.

Дэвид и все остальные разработчики ядра Rails искренне верят в достоинства высококачественного автоматизированного тестирования и подают пример сообществу: тестовое покрытие самоой платформы Rails необычайно широко. Заплаты, даже с мелкими исправлениями, не принимаются, если они не сопровождаются работающими тестами. С самого начала тестирование было неотъемлемой частью Пути Rails, что отличает ее от большинства других платформ.

Когда у вас войдет в привычку вести разработку под управлением тестов, вы станете тщательно продумывать и уточнять требования к проекту в самом начале, а не после того, как работа будет несколько раз забракована отделом контроля качества. Таким образом, разработка тестов одновременно с кодированием – это по существу один из видов специфицирования. В составе Ruby имеется библиотека RSpec, в которой процедура управления разработкой с помощью спецификаций понимается буквально. Вы можете использовать ее вместе с Rails *вместо* тестирования. К сожалению, RSpec пока не стала основным инструментом разработчиков на платформе Rails, а, поскольку эта глава называется «Тестирование», мы отложим разговор о RSpec до главы 18.

Честно говоря, темы, рассматриваемые в данной главе, могли бы составить содержание отдельной (и большой) книги. Было отнюдь не просто организовать материал так, чтобы он оказался полезен большинству читателей. Нелегким стал и выбор подходящего уровня детализации.

Поскольку эта книга – прежде всего справочник, я пытался не слишком отвлекаться на философские аспекты тестирования и ограничился обсуждением следующих вопросов:

- принятая в Rails терминология, относящаяся к тестированию;
- обзор структуры автономного тестирования `Test::Unit`, входящего в дистрибутив Ruby, и его связи с Rails;
- фикстуры – средство управления тестовыми данными, и причины, по которым все так их не любят;
- автономное тестирование, функциональное тестирование и тестирование сопряжений с помощью `Test::Unit`;
- задания Rake, относящиеся к тестированию;
- приемочное тестирование и система Selenium.

Терминология Rails, относящаяся к тестированию

Прежде чем двигаться дальше, я хотел бы внести ясность в некоторые вопросы, смущающие недавно пришедших к Rails программистов, знакомых с ортодоксальным подходом к автономному тестированию.

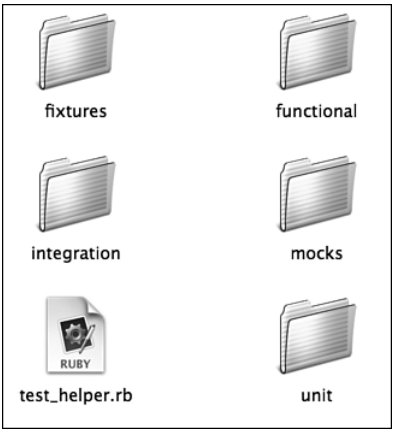

***Рис. 17.1.** Стандартные подкаталоги папки `test` приложения Rails*

На рис. 17.1 представлены стандартные подкаталоги папки `test` вашего приложения.

Термины *fixtures* (фикстуры), *integration* (сопряжение), *mocks* (mock-объекты, объекты-подделки) и *unit* (автономные тесты) имеют в Rails особый смысл, иногда слегка, а иногда кардинально отличающийся от трактовки, общепринятой в мире программотехники (software engineering). Возникающий когнитивный диссонанс делает тестирование одним из моих любимых аспектов среды Rails в целом.

К вопросу об изоляции...

В автономных и функциональных тестах Rails используется Fixtures API, взаимодействующий с базой данных. На самом деле надо еще постараться сделать так, чтобы автономные тесты не задействовали базу данных. Учитывая, что нормальные тесты в Rails никогда не тестируют изолированные блоки кода на Ruby, их невозможно считать истинно автономными; в соответствии с традиционным определением терминов их следовало бы называть функциональными тестами. Минуточку... Если все это функциональные тесты, то причем тут папка `unit`? Ага, вот мы и наткнулись на противоречие.

Боюсь, что у выбранной системы терминов нет убедительного оправдания. В папке `unit` в действительности находятся функциональные тесты моделей, ориентированные на тестирование отдельных методов, как и положено автономным тестам. В папку `functional` помещены функциональные тесты контроллеров, ориентированные на тестирование отдельных действий контроллеров. Тем и другим мы посвятим разделы в настоящей главе.

Mock-объекты в Rails

Папка `mocks` трактуется во время прогона тестов особым образом. Находящиеся в ней классы загружаются в последнюю очередь и, следовательно, могут переопределять поведение других классов в системе. Эта возможность особенно полезна при наличии классов, выполняющих действия, которые не должны выполняться на этапе тестирования:

- взаимодействие с внешними системами, например шлюзами платежных систем, системами геокодирования и другими веб-службами;
- порождение новых процессов или выполнение затяжной обработки;
- безвозвратное изменение состояния системы.

Пользуясь тем, что в Ruby классы открыты, вы можете написать *поддельные* версии методов вместо тех, которые при прогоне тестов выполняться не должны. Иначе говоря, *mock*-методы. Предположим, например, что некий код отправляет транзакции для обработки классу `PaymentGateway`. В режиме эксплуатации программа вызывает его метод `process`. Если этот метод не подделывать, то при прогоне теста транзакция будет послана *реальному* шлюзу платежной системы, что, скорее всего, недопустимо.

Однако проблему можно решить, подменив реализацию метода `process` в классе `PaymentGateway` собственной, которая не общается с настоящим шлюзом. Эта процедура называется *monkeypatching* (подмена)[1]. Подобный класс можно было бы поместить в каталог `mocks`:

```
class PaymentGateway
  def process(transaction)
    # Это наш шлюз, подделаем транзакции!
  end
end
```

Сама идея отдаленно напоминает традиционные mock-объекты, применяемые при тестировании, но на самом деле не имеет с ней ничего общего (в действительности это некая форма *заглушек*, то есть вы заглушаете истинную функциональность класса). Решив подделать шлюз под платежную систему, вы, вероятно, захотите наделить его ка-

[1] Высокомерные «Python-исты» терпеть не могут термин monkeypatching, описывающий изменение (возможное благодаря открытой природе классов в Ruby) реализации классов, которые не вы писали. Убежденные «Ruby-сты» тоже не любят его, так как не считают такую практику настолько необычной, чтобы придумывать для нее специальный термин. Адам Кейс (Adam Keys) считает, что сейчас в моде термин duck-punching. Мне кажется, что одно слово все же лучше длинной фразы: «изменение поведения существующего класса или объекта, возможное благодаря открытой природе классов в Ruby».

ким-то состоянием, дабы позже можно было проверить, что метод `process` действительно вызывался:

```
class PaymentGateway
  attr :processed_transaction

  def process(transaction)
    # Это наш шлюз, подделаем транзакции!
    @processed_transaction = transaction
  end
end
```

Если бы мы не проверяли, вызывался ли метод `process`, то никогда бы и не узнали, правильно ли работает программа. Я не стану далее развивать этот пример и вообще углубляться в тему mock-объектов. На мой взгляд, каталог `mocks` следовало бы вообще удалить. Никто им не пользуется. И многие из нас считают эту особенность Rails мерзостью, которая вообще не должна была бы появляться на свет[1].

Настоящие Mock-объекты и заглушки

Для включения механизма подделки в тесты Rails следует пользоваться библиотекой `Mocha`. На самом деле, она включена в дистрибутив Rails как зависимость для собственного комплекта тестов Rails. Но Mocha поставляется в виде RubyGem-пакета, поэтому вы сами должны скачать последнюю версию с помощью стандартной команды установки таких пакетов:

```
sudo gem install mocha
```

Библиотека Mocha предоставляет унифицированный, простой и легко воспринимаемый синтаксис как для традиционных Mock-объектов, так и для подделок реальных объектов. *Традиционным* Mock-объектом называется объект, который можно включать в сценарий, ожидая, что при прогоне теста будут вызваны определенные его методы. Если эти ожидания не оправдываются, Mock-объект приводит к завершению теста с ошибкой.

Mocha также предлагает механизм *заглушек*, то есть переопределения существующих методов, чтобы они возвращали известные значения. Она позволяет даже подделывать и заглушать методы реальных (не-Mock) классов и экземпляров. Так, можно заглушить методы `find` или акцессоры ассоциаций в экземпляре `ActiveRecord`, уменьшив тем самым зависимость от сложных настроек фикстур.

Ниже приведен реальный пример использования библиотеки Mocha для подделки действия, которое *не* должно выполняться во время про-

[1] Признаю, это преувеличение. Когда я показал точко что прочитанный вами абзац Рику Олсону, он завуалированно высказался в защиту разработчиков ядра, заметив: «Mocha тогда не было».

гона тестов, – обращения к службе Google Geocoder. Мало того что это медленно, так мы еще не сумеем успешно прогнать тест, не будучи подключенными к Сети:

```
class SearchControllerTest < Test::Unit::TestCase

  def test_should_geolocate_zip_codes_to_return_cities_result
    res = mock('response')
    res.stubs(:lat => 40, :lng => 50)
    GoogleGeocoder.expects(:geocode).with('07601').returns(res)

    place = mock('place')
    Place.expects(:new).with(:lat => 40, :lng => 50).returns(place)

    hackensack = mock('city')
    City.expects(:near).with(place, 10.miles).returns([hackensack])
    post :index, :text => '07601'

    assert_include hackensack, assigns(:places)
  end
end
```

Если непонятно, поясняю – этот тест проверяет, вызывает ли контроллер поиска службу геокодирования для поиска городов поблизости от региона с заданным почтовым индексом. Метод `mock` создает новый Mock-объект ответа. Благодаря *динамической типизации* (duck typing) реальный тип объекта не имеет значения. Метод `stubs` говорит Mock-объекту, что при вызове методов, указанных в качестве ключей хеша, должны возвращаться заданные значения. На этом я пока закончу разговор о библиотеке Mocha, чтобы не отклоняться слишком далеко от основной темы данной главы – тестирования. Однако я включил в текст несколько примеров, показывающих, как применение Mocha упрощает тестирование в Rails.

Тесты сопряжения

Тестирование сопряжений появилось в версии Rails 1.1 и, пожалуй, оно больше всего соответствует классическому пониманию этого термина в программотехнике. Тест позволяет подавать на вход несколько запросов и проверять совместную работу различных компонентов приложения:

```
class AdvancedTest < ActionController::IntegrationTest
  fixtures :people, :forums

  def test_login_and_create_forum
    login_as :admin
    get '/forums'
    assert_response :success

    post '/forums', :forum => {:name => 'a new forum'}
```

```
      assert_response :redirect
      follow_redirect!
      assert_select 'div#topics div.title:last-of-type', 'a new forum'
    end
  end
```

Тесты сопряжений можно считать «накачанными» функциональными тестами. Не стоит включать в проект слишком много таких тестов, поскольку они работают медленно. Тем не менее они помогают выявить трудные для обнаружения ошибки во взаимодействиях. Ниже мы посвятим тестированию сопряжений целый раздел.

О путанице в терминологии

Имеет ли место теорминологическая путаница в том, что касается тестирования в Rails? *Да, безусловно*. Будет ли она устранена в обозримом будущем? *Нет*[1].

Существуют ли альтернативы? Разумеется! Иначе это был бы не Ruby. Однако сразу предупреждаю, что, пойдя по любой из упомянутых ниже дорог, вы свернете с пути Rails, так что решайте сами. Вот краткий обзор имеющихся вариантов, как это мне представляется:

- мой любимый подход: вообще отказаться от инфраструктуры тестирования Rails и пользоваться RSpec (хотя бы бегло ознакомьтесь с разделами о фикстурах и тестировании сопряжений в этой главе и можете переходить к следующей);
- переименовать подкаталоги в папке `test`, назвав их `models`, `controllers`, `views` и т. д. Необходимые изменения в заданиях для Rake оставляю в качестве упражнения для читателя;
- запретить доступ к базе данных из автономных тестов, воспользовавшись советом Джея Филдса, опубликованным по адресу *http://blog.jayfields.com/2006/06/ruby-on-rails-unit-tests.html*.

> **Говорит Уилсон...**
>
> Тот, кто пишет приложения без тестов, – плохой человек, не способный любить.

[1] Я просил Дэвида пояснить, почему выбраны именно такие категории, и он ответил: «Термин *автономные тесты* был выбран, чтобы подчеркнуть, что они относятся к отдельным методам моделей или служб, тогда как функциональные тесты имеют дело с композицией нескольких элементов (контроллеров и моделей). На практике произошло смешение понятий, и сегодня их можно было бы назвать тестами моделей и контроллеров. Но для нас это не слишком существенно».

Единственное, чего вы никогда не должны делать, – это отказываться от тестирования вовсе. Поставлять приложения Rails без автоматизированного тестового покрытия безответственно и непрофессионально.

Класс Test::Unit

Все стандартные тесты в Rails расширяют встроенный в Ruby класс `Test::Unit`, который считается частью *семейства xUnit*. Аналогичные платформы существуют для большинства основных языков программирования. Если вам доводилось работать с программой JUnit для Java или NUnit для .NET, то с принципиальными концепциями вы уже знакомы. Если нет, вам поможет следующий далее обзор идей, положенных в основу xUnit.

Отдельный тестовый сценарий представляется подклассом `Test::Unit::TestCase`. В него включается набор методов, тестирующих тот или иной аспект приложения. Вот как тестовый сценарий выглядит в Rails:

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase

  fixtures :models

  def setup
    # код, выполняемый до запуска всех остальных тестовых методов
  end

  def test_some_bit_of_functionality
    # логика и утверждения
  end

  def teardown
    # код, выполняемый после завершения всех тестовых методов
  end
end
```

Тестом называется один открытый метод тестового сценария, имя которого начинается с префикса `test_`, а тело содержит код, доказывающий правильность работы небольшого фрагмента программы. Имена тестовых методов должны пояснять, для чего тест предназначен. Длинные имена, например `test_only_authorized_user_may_be_associated_as_approver`, приветствуются, а такие короткие, как `test_works`, порицаются. Текст теста должен быть удобочитаемым, и из него должно быть понятно назначение тестируемого кода. Поэтому тела тестовых методов должны быть краткими и целенаправленными.

Утверждением (assertion) называется сравнение ожидаемого значения с результатом вычисления выражения. Например, в тесте вы можете при-

своить атрибуту модели значение, отвергаемое валидатором, и в предложении `assert` проверить, что получено ожидаемое сообщение об ошибке.

Метод *setup* выполняется до запуска всех остальных тестовых методов. Метод *teardown* выполняется уже после окончания работы всех тестовых методов. Методы подготовки (setup) встречаются в тестах Rails очень часто, а методы очистки (teardown) – гораздо реже.

Тест считается успешным, если при его прогоне все утверждения дали ожидаемые результаты, и не возникло исключений. Различают два вида не-успешного исхода: *отказ* и *ошибка*. Отказ теста означает, что некоторое утверждение несправедливо, а ошибка – что из-за исключения или ошибки исполняющей среды выполнение теста было прервано.

Иногда успешный прогон теста называют *зеленым индикатором* (green bar), а неуспешный – *красным индикатором* (red bar). Метафора основана на традиционном представлении в графических интерфейсах к системам автономного тестирования. Когда тесты запускаются из командной строки, вы видите последовательность точек, свидетельствующих о прогоне отдельных тестовых методов. Символы `F` и `E` в этой последовательности означают *отказ* (failure) и *ошибку* (error) соответственно.

Комплектом тестов (test suite) называется набор тестовых сценариев; в других языках они часто определяются разработчиком. В стандартных проектах Rails нет понятия комплекта тестов, определенного разработчиком. Вместо этого наборы тестов хранятся в трех подкаталогах папки test – functional, integration и unit – и запускаются с помощью заданий Rake.

Прогон тестов

Классы `Test::Unit` устроены так, что при запуске тестового файла из командной строки в интерпретаторе Ruby будут выполнены все содержащиеся в нем тестовые методы. Чтобы прогнать только один тестовый метод, следует указать его имя после флага `-n`:

```
$ ruby test/unit/timesheet_test.rb -n
  test_only_authorized_user_may_be_associated_as_approver

Loaded suite test/unit/timesheet_test
Started
.
Finished in 1.093507 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

Как правило, мы запускаем весь комплект тестов приложения командой `rake test` из каталога проекта. Достаточно даже набрать просто `rake`, поскольку задание `test` подразумевается по умолчанию. Ниже в этой главе мы рассмотрим все относящиеся к тестированию задания Rake.

Фикстуры

Возможность получить именованную ссылку на конкретный кусок известного графа объектов – это поистине круто.

Майкл Козярски (Micael Koziarski), один из разработчиков ядра Rails

В промышленности стенды (fixture) применяются для производства и тестирования самых разных предметов: от печатных плат до электронных устройств, от исходного сырья до готовой продукции. В контексте программотехники фикстурой называется механизм определения исходного состояния системы до прогона тестов.

Генераторы кода в Rails создают принимаемые по умолчанию файлы фикстур для новых моделей в каталоге `test/fixtures` внутри проекта. Устроены они примерно так, как показано в листинге 17.1.

Файл имеет расширение `.yml`, показывающее, что он представлен в формате YAML (Yet Another Markup Language)[1]. Язык YAML применяется для сериализации данных и не имеет прямого отношения к XML. Принятая в нем организация областей видимости на основе отступов напоминает язык Python.

Листинг 17.1. Пример простого YAML-файла фикстуры: test/fixtures/users.yml

```
quentin:
  id: 1
  email: quentin@example.com
  created_at: <%= 6.months.ago.to_s(:db) %>
newbie:
  id: 2
  email: newbie@domain.com
  crypted_password: <%= Digest::SHA1.hexdigest("password") %>
  created_at: <%= 1.minute.ago.to_s(:db) %>
```

Отступы полей в фикстуре должны быть одинаковыми, иначе синтаксический анализатор YAML собьется. Кроме того, в отступе *не должно быть знаков табуляции*. Согласно спецификации YAML, ширина отступа – деталь представления, предназначенная исключительно для обозначения структуры, и в остальном игнорируется. Я рекомендую делать отступы в два пробела, как это принято в мире Ruby[2].

Взгляните еще раз на фикстуру в листинге 17.1. Файл содержит данные, которые должны находиться в таблице `users` тестовой базы на мо-

[1] Дополнительную информацию о YAML см. на сайте *http://yaml.org*.

[2] Нередко по неосторожности отступы в файлах фикстуры сбиваются. К счастью, сообщение, которое выдает Rails, встретив некорректно записанную фикстуру, обычно вполне информативно.

мент запуска тестов. Поясню, что отображение файла фикстуры на таблицу базы данных производится по имени файла.

По существу фикстура представляет собой двухмерный сериализованный хеш. В нашем примере тег `quentin:` – это ключ (имя), позволяющий однозначно идентифицировать хеш свойств, соответствующий первой записи в таблице `users` из тестовой базы данных. Возьмите за правило по возможности выбирать для записей в фикстурах информативные и осмысленные имена. Ключ используется для быстрого получения из теста ссылки на объект фикстуры без его загрузки из базы с помощью `ActiveRecord`.

Ключи второго уровня (`:id`, `:email`, `:created_at` и т. д.) должны соответствовать именам колонок в таблице. Как видно из данного примера, необязательно задавать ключи для каждой колонки; более того, старайтесь ограничиваться минимальным количеством данных, необходимым для целей тестирования. Поле `:id` должно присутствовать всегда, и зачастую возникают проблемы из-за перепутанной последовательности идентификаторов, загоняющей СУБД в ступор. Мы поговорим о том, как можно избежать подобных неприятностей, когда ниже будем рассматривать динамические данные в фикстурах.

Фикстуры в формате CSV

Фикстуры можно форматировать также в формате CSV (с полями, разделенными запятыми). Если раньше вы с ним не сталкивались, скажу, что CSV – старый переносимый текстовый формат, который в наши дни чаще всего применяется для платформенно независимого экспорта/импорта данных в электронные таблицы Microsoft Excel.

Вот та же самая (хотя воспринимаемая с куда большим трудом) фикстура, записанная в формате CSV:

```
id, email, crypted_password, created_at
1, quentin@example.com, , <%= 6.months.ago.to_s(:db) %>
2, newbie@domain.com, <%= Digest::SHA1.hexdigest("password") %>,
<%= 1.minute.ago.to_s(:db) %>
```

Фу, как некрасиво! Иногда тестовые данные удобно хранить в какой-нибудь электронной таблице, например Excel. Но, вообще-то, YAML гораздо популярнее, так как его проще читать и редактировать вручную.

Доступ к записям фикстуры из тестов

Тот факт, что записи в YAML-файле фикстуры снабжены осмысленными именами, служит еще одним доводом в пользу формата YAML. Для доступа к записи в файле фикстуры необходимо указать используемую фикстуру в контексте класса вашего теста. Затем в методах класса вызывается метод с тем же именем, что у файла фикстуры, а в качестве параметра ему передается имя записи, например `timesheets(:first)`. Это

простой и быстрый способ обращения к тестовым данным, а поскольку имена выбираете вы сами, им можно придать разумную семантику:

```
fixtures :users

def test_can_access_quentin_fixture
  assert(users(:quentin) != nil)
end
```

С другой стороны, имена для записей в CSV-фикстурах генерируются автоматически: сначала указывается имя файла фикстуры, затем подчерк и порядковый номер. В нашем примере первая фикстура будет называться `:user_1`, следующая – `:user_2` и т. д. У этих имен нет никакой семантики, можно лишь заключить, что речь идет об экземплярах класса `user`.

Динамические данные в фикстурах

Перед тем как загружать записи из фикстуры в тестовую базу данных, Rails пропускает их через анализатор ERb. Следовательно, в фикстурах может присутствовать внедренный код, заключенный в скобки `<% %>` или `<%= %>`, как и в шаблонах представлений. Этим динамическим поведением можно воспользоваться для повышения эффективности работы с фикстурами.

В предыдущих примерах встречалось динамическое содержимое. Так, конструкция `<%= Digest::SHA1.hexdigest ("password") %>` избавляет нас от необходимости вручную выполнять шифрование и копировать такие непонятные строки, как `00742970dc9e6319f8019fd54864d3ea740f04b1`, рискуя допустить при этом ошибку.

Другой пример – задание значений дат. Мы можем попросить Rails отформатировать даты в соответствии с требованиями СУБД: `<%= 5.days.ago.to_s(:db) %>`.

Использование динамического содержимого в фикстурах не ограничивается преобразованием и форматированием значений. Можно включать циклы для порождения большого объема данных. В документации по Rails приведен пример, демонстрирующий такую технику:

```
<% for i in 1..1000 %>
fix_<%= i %>:
  id: <%= i %>
  name: guy_<%= 1 %>
<% end %>
```

Не забывайте, что внутри тегов ERb может находиться любой корректный код на Ruby, в том числе определение новых методов-помощников и включение модулей.

При работе с фикстурами меня всегда раздражала необходимость вручную следить за правильной последовательностью идентификаторов.

Потом я понял, что можно просто включить в файл фикстуры вызов метода `auto_increment` (сами записи я для простоты сократил):

```
<%
  def auto_increment
    @id ||= 0; @id += 1
  end
%>
quentin:
  id: <%= auto_increment %>
  login: quentin
aaron:
  id: <%= auto_increment %>
  login: aaron
```

Если вы хотите сделать методы, подобные `auto_increment`, доступными в *любом* файле фикстуры, определите их в файле `fixture_helpers.rb`. Поместите этот файл в папку проекта `/lib` и добавьте строчку `require 'fixture_helpers'` в начало файла `test/test_helper.rb`.

Использование данных из фикстур в режиме разработки

Потратив немало времени на тщательную подготовку тестовых данных, вы, наверное, захотите сохранить записи из фикстуры в базе данных, используемой в режиме разработки, чтобы можно было обращаться к ним интерактивно из браузера или консоли.

В конфигурации Rake, принимаемой в Rails по умолчанию, имеется специально предназначенная для этого цель. Наберите в командной строке `rake db:fixtures:load`, чтобы импортировать записи фикстуры в текущую среду. Эта цель `rake` позволяет выбрать подмножество загружаемых фикстур, для чего нужно лишь добавить в конец команды выражение вида `FIXTURE=table1,table2`.

Примечание

Если вы пользуетесь в фикстурах методами-помощниками, как описано в предыдущем разделе, то имейте в виду, что задание `db:fixtures:load` не затребует файл `test_helper.rb`, поэтому вам, возможно, придется поместить строку `require 'fixture_helpers'` в конец файла `config/environments/development.rb`.

Генерация фикстур из данных, используемых в режиме разработки

Если в вашем приложении имеются сложные отношения или вы просто ленитесь[1] вручную создавать файлы фикстур, возможно, вы захоти-

[1] Назвать программиста ленивым – это комплимент, а не оскорбление.

те автоматически сгенерировать фикстуры из данных, хранящихся в базе, которая используется в режиме разработки. Предполагая, что уже имеется работающее приложение, вам нужен лишь способ экспорта данных из базы в формате YAML.

По какой-то причине экспорт данных в фикстуры *не* является частью ядра Rails[1]. Но, учитывая, что Rails предоставляет метод `to_yaml` для моделей `ActiveRecord` и объектов `Hash`, не слишком трудно написать задание для Rake самостоятельно. Однако и это не обязательно, потому что Джеффри Грозенбах уже написал подключаемый модуль, ценность которого проверена на практике. Он называется `ar_fixtures` и устанавливается командой:

```
$ script/plugin install http://topfunky.net/svn/plugins/ar_fixtures
```

После установки модуля для экспорта фикстуры достаточно выполнить команду `rake db:fixtures:dump`. В отличие от встроенного задания Rake для загрузки фикстур, это принимает параметр `MODEL`, передающий имя подкласса `ActiveRecord`, из которого экспортируются данные:

```
$ rake db:fixtures:dump MODEL=BillingCode
```

Никаких подтверждений не выдается, поэтому проверяйте, что было сделано:

```
--
billing_code_00001:
  code: TRAVEL
  client_id:
  id: 1
  description: Разнообразные транспортные расходы
billing_code_00002:
  code: DEVELOPMENT
  client_id:
  id: 2
  description: Кодирование и т. д.
```

Честно говоря, мне не очень нравится, как этот подключаемый модуль выводит данные, но это просто потому, что мне бы хотелось видеть колонку `id` в начале и исправить другие подобные мелочи. Кроме того, я хотел бы, чтобы записи со значениями `nil` не выводились вовсе. Но в любом случае в качестве отправной точки этот модуль вполне годится.

---

[1] Когда я задал этот вопрос в блоге #caboose, Кортенэ ответил: «Наверное, ни у кого из разработчиков не возникало желания экспортировать свои фикстуры в файл». Может быть, и правда все объясняется так просто.

Кстати, пусть вас не пугает строка в начале файла. Она просто обозначает начало YAML-документа и может быть удалена безо всяких последствий.

Параметры фикстур

Параметр `use_transactional_fixtures`, присутствующий вместе с пояснениями в стандартном файле `test/test_helper.rb` вашего проекта, определяет, должен ли Rails пытаться ускорить выполнение тестов, погружая каждый тестовый метод в транзакцию, которая откатывается при завершении. Откат ускоряет прогон комплекта тестов, поскольку отпадает необходимость загружать данные из фикстур в базу перед запуском каждого теста. По умолчанию этот параметр равен `true`.

Транзакционные фикстуры важны не только ради производительности. Откат изменений после завершения каждого тестового метода означает, что можно избежать сцепленности между методами, то есть зависимости от изменений, имевших место вне контекста одного метода. Взаимосвязанные тестовые методы гораздо труднее отлаживать, поэтому не делайте такой ужасной ошибки.

В стандартном файле `test_helper.rb` есть также параметр `use_instantiated_fixtures`, по умолчанию равный `false`. Если его включить, то в каждой записи фикстуры автоматически появятся переменные экземпляра, но ценой заметного снижения производительности тестов. Этот параметр восходит к самым ранним версиям Rails и, насколько я знаю, теперь не применяется.

Никто не любит фикстуры

На конференции Railsconf 2006 один из выступавших спросил: «Кто из сидящих здесь *любит* фикстуры?»

Я стоял в конце зала и немного отвлекся. Вообще-то я принадлежу к людям, которые не колеблясь поднимают руку на публике. И потому оказался в нелепом положении – единственный из всех ответил на вопрос утвердительно. Так почему же опытные разработчики Rails столь непримиримы к фикстурам? На этот вопрос есть разные ответы, и все они многогранны. Я познакомлю вас с собственной интерпретацией этой нелюбви, а потом поделюсь своими соображениями о том, как следует пользоваться фикстурам, не сбиваясь с пути Rails.

Фикстуры замедляют выполнение тестов

Это правда. Фикстуры обращаются к базе данных, и в результате большие комплекты тестов увязают в густой жиже бешеной активности ввода/вывода. Стоит убрать фикстуры, и комплекты, состоящие из тысяч тестовых сценариев, начинают выполняться за считанные секунды. Эту причину часто называют сторонники чистоты автономного тестирования.

В общем случае не включайте фикстуры, которые тестам не нужны. Если удается провести тестирование, пользуясь только объектами, созданными «с нуля», или только Mock-объектами и заглушками, создаваемыми библиотеками типа Mocha, так даже лучше.

Фикстуры позволяют работать с некорректными данными

Если вы принадлежит к людям, для которых корректность данных в базе превыше всего, то, наверное, фикстуры для вас – несомненное зло. Почему? Потому что во время загрузки данных из фикстур в базу не производится никаких проверок. Rails наплевать, задали вы в фикстуре только одну колонку данных или все. Он с радостью поместит в строку данные, которые вы предоставили. Чуть ли не единственное требование – совместимость типа колонки с записываемыми в нее данными. Это, конечно, плохо.

Но если вы так сильно обеспокоены корректностью данных в базе, *не пишите фикстуры с некорректными данными*. И проблема будет решена.

Попутно отмечу, что если эта причина ненависти к фикстурам задела вас за живое, то вы, наверное, еще и верите в полезность ограничений внешнего ключа. Фикстуры плохо уживаются с такими ограничениями, поскольку не так-то просто правильно задать порядок загрузки данных в базу.

Проблемы с поддержкой

Эта причина относится к разряду более глобальных. Если вы храните в фикстурах много данных, то в конце концов столкнетесь с проблемой их сопровождения. Управлять большими наборами данных, представленными в формате фикстур, слишком сложно... именно поэтому мы храним данные в базе!

Трудно также поддерживать согласованность внешних ключей между объектами, хранящимися в фикстурах. Наши мозги просто не предназначены для запоминания отношений, выраженных одними лишь целыми числами.

За все время работы с Rails я так и не нашел ни одной убедительной причины хранить в файлах фикстур более десятка представительных экземпляров моделей. Хотите хранить больше – потом не жалуйтесь.

Фикстуры хрупки

Еще одна причина не любить фикстуры – их *хрупкость*. Под словом «хрупкость» понимают способность (или неспособность) кода сохранять работоспособность при изменениях. Если мелкое изменение в фикстурах «ломает» сразу группу тестов, хрупкость налицо. Если хрупких тестов становится слишком много, вы вообще перестаете тестировать, и тогда после смерти с гарантией попадете в ад. Сами видите, фикстуры – зло!

Но если серьезно, существуют стратегии, позволяющие избежать хрупкости фикстур. Еще раз повторю свой совет – объем фикстур должен быть не слишком велик. Тогда реже придется вносить в них изменения. Существуют также подключаемые модули, позволяющие определить дискретные наборы фикстур[1], представляющие различные сценарии тестирования, а не сваливать все в одну кучу.

Кроме того, подключаемый модуль `fixtures_references`[2] дает интересное решение проблемы хрупкости, разрешая ссылаться на одни фикстуры из других. Да-да, вам больше не придется держать в памяти перекрестные ссылки, выраженные в терминах внешних ключей. Просто сошлитесь на связанную запись фикстуры, пользуясь тем же синтаксисом, что и в самом тесте:

```
<% fixtures :employees %>

packaging:
  id: 1
  name: Packaging
  manager_id: <%= employees(:bob)['id'] %>
```

Не все так плохо с фикстурами

Работая над своими проектами, я убедился, что при использовании такой структуры для создания Mock-объектов, как Mocha, можно свести применение фикстур к минимуму. И тогда они не причинят неприятностей, описанных в предыдущем разделе.

Иногда удается обойтись вовсе без фикстур, и это замечательно. В других случаях они оказываются удобны. Я считаю, что главное тут – оставлять в фикстурах лишь немногие репрезентативные случаи, которые вы способны относительно легко поддерживать и при необходимости объединять их заглушками. Кстати, иногда без фикстур просто никуда не деться (если, конечно, вы не хотите создавать внутри теста объекты с нуля и записывать их в базу данных, чего я лично терпеть не могу). На ум сразу приходит ситуация с тестированием метода `find_by_sql` в моделях. Коль скоро вы пишете SQL-запрос, то определенно хотите предъявить его настоящей базе данных и убедиться, что он работает. Написать для этого случая чистый автономный тест невозможно, если только не разбирать SQL самостоятельно!

Резюме: *ненавидеть* фикстуры необязательно, надо лишь пользоваться ими с умом.

---

[1] Обогатите свой опыт работы с фикстурами, прочитав статью на странице *http://thatswhatimtalkingabout.org/news/2006/8/31/fixture-sets-for-rails*.

[2] Обогатите свой опыт еще сильнее, познакомившись со статьей *http://www.pluginaweek.org/2007/04/07/14-fix-your-fixtures-with-fewer-foreign-key-ids/*.

Утверждения

Методы-утверждения – это механизм верификации, применяемый внутри тестовых методов. Любой метод-утверждение принимает необязательный (последний) параметр, в котором задается сообщение об отказе. Пользуйтесь ими, если не хотите, чтобы кто-то (а то и вы сами) долго чесал в затылке, пытаясь понять, в чем причина отказа теста.

Простые утверждения

В модуле `Test::Unit::Assertions` (это часть самого языка Ruby) определены разнообразные простые методы-утверждения, которые можно использовать во всех тестах Rails. Для экономии места в приведенном ниже перечне методы, содержащие и не содержащие слово `not`, описываются в одном разделе. Кроме того, я опустил парочку методов, не имеющих отношения к программированию в Rails.

assert и deny

Если бы утверждения были ручными инструментами, то `assert` и `deny` оказались бы кувалдами – тупыми, целенаправленными и мало пригодными для тонкой работы. Дайте им булево выражение, и они будут счастливы. Эти утверждение больше любых других заслуживают явного задания сообщений об отказе. В противном случае вы увидите несколько обескураживающее заявление типа false should be true (ложь должна быть истиной).

```
assert @user.valid?, "user был неправильным"

deny @user.valid?, "user был правильным"
```

Применяя `assert`, не забывайте, что передавать ему необходимо именно булево значение. И помните, что `nil` в булевом контексте вычисляется как `false`. Если ваша цель – проверить, равен ли результат вычисления выражения `nil`, пользуйтесь утверждениями `assert_nil` или `assert_not_nil`, чтобы ясно выразить свое намерение.

assert_block

Проверяет, что блок возвращает значение `true`:

```
assert_block "Отказ из-за невозможности выполнить do" do
  # какая-то обработка в блоке do
  do_the_thing
end
```

assert_empty

Проверяет, пуст ли набор (метод `empty?` возвращает `true`).

```
assert_empty @collection
```

assert_equal и assert_not_equal

Принимают два параметра и сравнивают их на равенство (или неравенство) методом `equal?`. Первый параметр – ожидаемое значение, второй – проверяемое выражение.

```
assert_equal "passed", @status
```

assert_in_delta и assert_in_epsilon

Убеждается, что число с плавающей точкой не слишком далеко отклоняется от заданного значения. В документации по Ruby говорится, что утверждение `assert_in_epsilon` «похоже на `assert_in_delta`, но лучше обрабатывает погрешность пропорционально величине параметров»:

```
assert_in_delta(expected, actual, 0.01, message="exceeded tolerance")
```

assert_include

Проверяет, что элемент входит в набор:

```
assert_include item, @collection
```

assert_instance_of

Проверяет, является ли объект экземпляром заданного класса (не модуля). Частица `of` в имени утверждения призвана напомнить, что первым параметром является класс:

```
assert_instance_of Time, @timestamp
```

assert_kind_of

Убеждается, что объект связан с классом *или* модулем отношением `kind_of?`. Частица `of` в имени утверждения означает, что первым параметром является класс или модуль:

```
assert_kind_of Enumerable, @my_collection
```

assert_match и assert_no_match

Проверяет, сопоставляется ли заданное значение с регулярным выражением. Регулярное выражение является первым параметром:

```
assert_match /\d{5}(-\d{4})?/, "78430-9594"
```

assert_nil и assert_not_nil

Убеждается, что ссылка равна (или не равна) `nil`:

```
assert_not_nil User.find(:first)
```

assert_same и assert_not_same

Убеждается, что оба параметра ссылаются на один и тот же (или разные) объект(ы):

```
assert_same "foo".intern, "foo".intern
```

assert_raise и assert_nothing_raised

Проверяет, возбуждается ли исключение внутри переданного блока. Мне нравится пример в документации по Ruby:

```
assert_raise RuntimeError, LoadError do
   raise 'Бум!!!'
end
```

assert_respond_to

Проверяет, отвечает ли объект на заданное сообщение (метод `respond_to?` возвращает `true`). Первым параметром должен быть объект, вторым — сообщение:

```
assert_respond_to @playlist, :shuffle
```

flunk

Вызывает отказ теста с указанным сообщением:

```
flunk "REDO FROM START"
```

Легко допустить ошибку, употребив метод `fail` (принадлежащий классу `Kernel` и потому доступный из любого места) вместо `flunk`. Метод `fail` приводит к возбуждению исключения исполняющей средой и, стало быть, к прекращению теста, но с ошибкой, а не с отказом.

Утверждения Rails

Rails добавляет еще ряд утверждений помимо предоставляемых структурой `Test::Unit`:

- `assert_difference` и `assert_no_difference`
- `assert_generates`, `assert_recognizes` и `assert_routing`
- `assert_response` и `assert_redirected_to`
- `assert_tag` (устарело) и `assert_select`
- `assert_template`
- `assert_valid`

Универсальными в этом списке являются только утверждения `assert_difference` и `assert_no_difference`. Все утверждения будут рассмотрены

в последующих разделах с примерами, когда мы будем говорить о тестировании конкретных частей Rails.

По одному утверждению в каждом тестовом методе

Я бы предпочел при первом прогоне комплекта тестов узнать, что все 10 тестов не прошли, чем увидеть, что не прошли 5, а остальные то ли прошли, то ли нет.

Джей Филдс

Я люблю, чтобы тесты были максимально элегантными и выразительными. Идею ограничиться только одним утверждением в каждом тестовом методе обычно приписывают гуру методики разработки на основе тестов Дейву Эстелсу (Dave Astels). В сообществе Rails ее популяризировали Джей Филдс[1] и др.

Если вы последуете этому совету, то вместо одного такого тестового метода:

```
def test_email_address_validation
  u = users(:sam)
  u.email = "sam"
  assert ! u.valid?
  u.email = "johh.doe@google.com"
  assert u.valid?
  u.email = "johh_doe@mail.mx.1.google.com"
  assert u.valid?
  u.email = "johh_doe+crazy-iness@mail.mx.1.google.com"
  assert u.valid?
  u.email = "sam@@colgate.com"
  assert ! u.valid?
end
```

напишете, по крайней мере, пять методов, каждый из которых будет лучше описывать тестируемое поведение. Для удобства я бы еще переместил присваивание значения `users(:sam)` переменной в метод `setup`:

```
def setup
  @sam = users(:sam)
end

def test_email_validation_fails_with_simple_string_name
  @sam.email = "sam"
  assert! @sam.valid? # prefer not over ! for readability
end

def test_email_validation_should_succeed_for_valid_email_containing_dot
  @sam.email = "johh.doe@google.com"
```

[1] Джей дал одно из лучших объяснений причины, по которой не следует в одном тесте употреблять несколько утверждений, в своем блоге по адресу *http://blog.jayfields.com/2007/06/testing-one-assertion-per-test.html*.

```
  assert @sam.valid?
end

... # надеюсь, вы поняли идею
```

Основное достоинство такой тактики состоит в том, что тесты становятся удобнее для сопровождения, но есть и ряд других важных преимуществ:

- вы вынуждены более четко определять назначение теста в его имени, которое выводится в результаты тестирования в случае отказа. В противном случае смысл каждого утверждения будет упрятан в комментарий или вообще останется темен;
- вы получаете более точную картину *причины*, по которой тест не прошел. Если в одном тестовом методе имеется несколько утверждений (возможно, не связанных между собой), то первое же ложное приводит к отказу теста, а остальные даже не проверяются;
- Джей также говорит, что следование принципу одного утверждения помогает ему критически осмысливать проект модели предметной области: «И часто в итоге мне удается структурировать модель так, что каждый метод отвечает только за одну функцию».

Давайте еще немного поговорим о тестировании модели предметной области приложения Rails, точнее о моделях `ActiveRecord`.

Тестирование моделей с помощью автономных тестов

Мы уже отмечали выше, что автономный тест в Rails – это на самом деле функциональный тест модели `ActiveRecord`. Необязательно тестировать работу встроенных методов, например `find`, поскольку они уже в достаточной мере покрыты тестами самой среды Rails.

Например, следующий тест может быть полезен в образовательных целях, но для вашего проекта Rails совершенно бесполезен:

```
def test_find
  @user = User.create(:name => "John Foster")
  assert_equal @user, User.find_by_name("John Foster")
end
```

Почему бесполезен? Потому что в нем тестируется функциональность Rails, уже подтвержденная тестовым покрытием. Если вы не писали метод `find_by_name` вручную, то ровным счетом ничего не доказали относительно *своего* кода, а лишь удостоверились в том, что Rails работает, как и ожидалось.

Стоит отметить, что, помимо методов экземпляров, в наших моделях `ActiveRecord` немало декларативных *методов-макросов*, размещаемых в начале файла. Не забывайте создавать тесты и для них тоже.

Основы тестирования моделей

Когда генератор создает модель `ActiveRecord`, Rails автоматически помещает в каталог `test/unit` проекта заготовку автономного теста. Выглядит она примерно так:

```
require File.dirname(__FILE__) + '/../test_helper'

class BillableWeekTest < Test::Unit::TestCase
  fixtures :billable_weeks

  # Заменить реальными тестами.
  def test_truth
    assert true
  end
end
```

> **Говорит Уилсон...**
>
> Среднее приложение Rails было бы на 120% лучше, если бы генерируемый по умолчанию тест модели содержал предложение `flunk "напиши меня"`, а не `assert true`.

Традиционно автономный тест ограничивается тестированием одного открытого метода объекта, и больше ничем не занимается[1]. Тестируя модель `ActiveRecord` в Rails, мы обычно включаем несколько тестовых методов для каждого открытого метода, чтобы тщательнее проверить поведение.

В листинге 17.2 приведены два тестовых метода из класса `TimesheetTest` для демонстрационного приложения.

Листинг 17.2. Тест модели Timesheet из приложения для управления временем и затратами

```
 1 class TimesheetTest < Test::Unit::TestCase
 2
 3   fixtures :users
 4
 5    def test_authorized_user_may_be_associated_as_approver
 6     sheet = Timesheet.create
```

[1] Если для автономного теста необходимы какие-то зависимости, внешние по отношению к тестируемому объекту, их следует моделировать с помощью Mock-объектов или заглушек, чтобы поведение теста не определялось корректной работой этих зависимостей. О Mock-объектах или заглушках мы поговорим ниже.

```
 7     sheet.approver = users(:approver)
 8     assert sheet.save
 9    end
10
11    def test_non_authorized_user_cannot_be_associated_as_approver
12      sheet = Timesheet.create
13      begin
14        sheet.approver = users(:joe)
15        flunk "присваивание approver должно завершиться ошибкой"
16      rescue UnauthorizedApproverException
17        # ожидается
18      end
29    end
30 end
```

Выше в этой главе мы уже говорили, как использовать систему фикстур, чтобы получить объекты, готовые для тестирования (в строке 3 вызывается метод `fixtures :users`). Можно иметь сколько угодно фикстур, имена которых всегда должны задаваться во множественном числе. Я не загружаю фикстуры табелей, поскольку в них нет необходимости. Простейшего, созданного с нуля экземпляра `Timesheet` мне вполне достаточно (строки 6 и 12).

Если бы я хотел написать еще более идиоматичный Ruby-код в этом примере, то, пожалуй, изменил бы строки 13–18, воспользовавшись утверждением `assert_raises`.

Что тестировать

В этом тесте скрыто присутствует реализация решения о том, разрешено ли пользователю одобрять табели. В вашем случае это может оказаться как хорошо, так и плохо. Я считаю, что хорошо. В конце концов, это тест табеля, а не пользователя или системы авторизации. Когда я писал этот тестовый сценарий, то хотел лишь проверить, что одно присваивание завершается успешно, а другое – нет. Логика авторизации в данном случае несущественна.

Тестирование контроллеров с помощью функциональных тестов

Когда вы пользуетесь генератором для создания контроллера `ActiveRecord`, Rails автоматически создает функциональный тест. Функциональные тесты позволяют проверить, что контроллер правильно обрабатывает поступающие ему запросы. Поскольку контроллер также вызывает код представления для запрошенных действий, то в функциональный тест можно включать утверждения о содержимом ответа.

Структура и подготовка

Функциональные тесты следуют определенным соглашениям, с которыми большинство начинающих разработчиков Rails знакомятся, впервые открывая функциональный тест, сгенерированный средой обстраивания. В примерах ниже мы воспользуемся функциональным тестом из открытого пакета Beast для создания досок объявлений[1].

Первая строка похожа на автономный тест, поскольку требует общий файл `test_helper`. Затем требуется тестируемый контроллер:

```
require File.dirname(__FILE__) + '/../test_helper'
```

Обычно все исключения, возбуждаемые во время выполнения контроллера, перехватываются в предложении `rescue`, чтобы пользователю можно было показать страницу с сообщением об ошибке. Но во время тестирования желательно, чтобы исключения дошли до верхнего уровня, где платформа их перехватит и сможет зарегистрировать. Вот отличный случай показать, насколько полезны открытые классы в Ruby, – в функциональных тестах мы просто переопределяем метод `rescue_action` в классе `ActionController`:

```
require 'forums_controller'
# Повторно возбудить то же исключение, что и сам контроллер.
class ForumsController; def rescue_action(e); raise e; end; end
```

Точки с запятой, которые в Ruby служат необязательными ограничителями предложений, здесь используются, чтобы уменьшить длину файла.

Далее мы открываем класс самого функционального теста, имя которого, по принятому соглашению, начинается с имени тестируемого контроллера. Если в тестовом сценарии используются фикстуры, они обычно указываются в следующей же строчке:

```
class ForumsControllerTest < Test::Unit::TestCase
  fixtures :forums, :posts, :users
```

В функциональных тестах всегда имеется метод `setup`. В нем инициализируются три обязательных переменных экземпляра, и выполняется он перед вызовом тестовых методов:

```
def setup
  @controller = ForumsController.new
  @request = ActionController::TestRequest.new
  @response = ActionController::TestResponse.new
end
```

Вы можете использовать объекты `@request` и `@response` напрямую, но для удобства предлагаются еще и дополнительные акцессоры для ат-

[1] Beast – это «небольшой легкий форум на платформе Rails с пугающим названием (в переводе – чудовище), состоящий примерно из 500 строк кода». Дополнительную информацию см. на сайте *http://beast.caboo.se/*.

рибутов session, flash и cookies. К этим структурам можно в любой момент обращаться из тестового метода (как для установки значения, так и в утверждении об ожидаемом изменении состояния после выполнения действия контроллера).

Методы функциональных тестов

Обычно для каждого действия контроллера пишут по крайней мере два тестовых метода: для нормальной обработки и для обработки ошибочных ситуаций. Как правило, любой метод функционального теста сначала обеспечивает необходимые предусловия, а затем вызывает метод (GET, POST, PUT, DELETE или HEAD), соответствующий типу HTTP-запроса. Предоставляется также метод xhr, позволяющий имитировать взаимодействие с объектом XMLHttpRequest броузера, посылающего Ajax-запрос.

Первый параметр метода запроса – символ, соответствующий имени тестируемого метода действия, далее передается хеш параметров.

В следующем примере проверяется, успешно ли выполнился GET-запрос к URL /forums/edit/1^:

```
def test_should_get_edit_form
  login_as :aaron
  get :edit, :id => 1
  assert_response :success
end
```

Типичные утверждения

Ниже описаны наиболее употребительные в функциональных тестах утверждения. Вы можете придумать и свои собственные, но не забывайте, что основная задача контроллера – вызов бизнес-логики моделей и подготовка окружения для правильного рендеринга представления.

Проверка правильности присваивания значений переменным, используемым в шаблоне

Одна из главных задач действий контроллера, которые выполняют рендеринг шаблона, – инициализация используемых в нем переменных экземпляра. Написать такого рода утверждения можно по-разному, но идея всегда одна и та же – воспользоваться тем, что после вызова действия контроллера метод assigns возвращает хеш переменных экземпляра, подготовленных для шаблона.

Из-за наличия кода обстраивания у многих появилась привычка ограничиваться простым утверждением, что переменным присвоены значения, отличные от nil. Это менее безопасно, чем утверждения о реальных значениях переменных:

```
def test_that_users_are_assigned_properly

  # недостаточно сильное утверждение
  assert_not_nil assigns(:users)

  # уже лучше
  assert_equal 3, assigns(:users).size

  # самое лучшее, проверяет содержимое users и получает осмысленное
  # сообщение в случае неудачи
  assert_equal %w(Bobby Sammy Jonathan), assigns(:users).map(&:login)
end
```

То самое «осмысленное сообщение», о котором говорится в комментарии, мы получаем за счет небольшого трюка с методом `map`, позволяющего производить сравнение с массивом имен, а не объектов. Проблема в том, что когда метод `assert_equal` завершается неудачно, на выходе мы получаем выданную `inspect` распечатку состояния не совпавших объектов, а для больших моделей `ActiveRecord` это будет малопонятный набор строчек кода.

Проверка кода состояния HTTP в ответе

Утверждение `assert_response` в функциональных тестах – удобный способ проверить, что ответ имеет один из следующих типов:

- `:success` – код состояния равен 200
- `:redirect` – код состояния в диапазоне 300–399
- `:missing` – код состояния равен 404
- `:error` – код состояния в диапазоне 500–599

```
def test_should_get_index_successfully
  get :index
  assert_response :success
end
```

Можно также явно передать числовое значение кода, например `assert_response(501)`, или его символьный эквивалент[1] – `assert_response(:not_implemented)`.

Проверка MIME-типа содержимого ответа (и значений из других заголовков)

В объекте `@response` хранится много информации (если хотите в этом убедиться, попробуйте добавить строчку `puts @response.inspect` после обработки запроса). Среди прочих атрибутов имеется хеш `headers`, ко-

[1] Полный перечень символов, описывающих коды ответа, см. в исходном тексте класса `ActionController::StatusCodes`.

торым можно воспользоваться, например, для проверки MIME-типа содержимого и кодировки символов:

```
XML_UTF8 = "application/xml; charset=utf-8"

def test_should_get_index_with_xml
  request.env['Content-Type'] = 'application/xml'
  get :index, :format => 'xml'
  assert_response :success
  assert_equal XML_UTF8, @response.headers['Content-Type']
end
```

Проверка рендеринга конкретного шаблона

Метод `assert_template` позволяет без труда проверить, выполнило ли тестируемое действие рендеринг указанного шаблона:

```
def test_get_index_should_render_index_template
  get :index
  assert_template 'index'
end
```

Проверка переадресации на заданный URL

Аналогично метод `assert_redirected_to` проверяет, что последнее выполненное действие переадресовало клиента именно на адрес, указанный в переданных параметрах. Совпадение может быть частичным, например `assert_redirected_to(:controller => "weblog")` будет также соответствовать результату выполнения метода `redirect_to(:controller => "weblog", :action => "show")` и т. д.:

```
def test_accessing_protected_content_redirects_to_login
  get :protected
  assert_redirected_to :controller => "session", :action => "new"
end
```

Метод `follow_redirect` предназначен, чтобы можно было сформулировать утверждения относительно второго действия того же самого контроллера[1]. Попытка последовать по пути переадресации, ведущему на другой контроллер, приведет к исключению (см. ниже раздел «Тесты сопряжения в Rails» о том, как в одном тестовом методе выполнять несколько запросов).

Проверка задания кратких сообщений

Вспомогательный метод `flash` дает прямой доступ к хешу `flash` в сеансе пользователя:

---

[1] На момент написания книги метод `follow_redirect` в функциональных тестах работал неправильно. Дополнительную информацию о самой ошибке и ее текущем статусе см. на странице *http://dev.rubyonrails.org/ticket/7997*.

```
def test_accessing_protected_content_redirects_to_login
  post :create ... # плохие атрибуты
  assert_equal "Ошибка при сохранении. Проверьте форму и попробуйте еще раз.",
  flash[:error]
end
```

Проверка изменений в базе данных после выполнения действия

Метод `assert_difference` позволяет легко проверить результат работы действий контроллера, добавляющих в базу данных новые записи или удаляющих существующие:

```
assert_difference 'Article.count' do
  post :create, :article => {...}
end
```

Можно указать как положительную, так и отрицательную «дельту». По умолчанию принимается +1, но если задать −1, получится удобный способ тестирования операций удаления:

```
assert_difference 'Article.count', -1 do
  delete :destroy, :id => ...
end
```

Можно также передать массив выражений, которые будут вычислены:

```
assert_difference [ 'Report.count', 'AuditLog.entries.size' ], +2 do
  post :create, :article => {...}
end
```

Кстати, `+2` — допустимый (хотя и редко применяемый) синтаксис в Ruby. В данном случае это излишне, но очень красноречиво выражает идею данного утверждения. Пусть удобство чтения кода будет одной из важнейших ваших целей при повседневном кодировании, а особенно в тестах.

Если вы хотите проверить, что количество записей в базе данных не изменилось, можете воспользоваться методом `assert_no_difference`:

```
assert_no_difference 'Article.count' do
  post :create, :article => {...} # недопустимые атрибуты
end
```

Проверка корректности состояния модели

Метод `assert_valid` проверяет, что, по мнению `ActiveRecord`, переданная запись корректна, и выводит сообщение об ошибке, если это не так:

```
def test_create_should_assign_a_valid_model
  post :create ... # допустимые атрибуты
  assert_valid assigns(:post)
end
```

Проверка результата вывода представления

Путь Rails предполагает, что тестирование результатов рендеринга производится в контексте тестовых методов контроллера. Срабатывание действий контроллера инициирует всю цепочку рендеринга, и результирующая HTML-разметка (или иная) оказывается в атрибуте `response.body`. В Rails имеется развитый API для формулирования утверждений о полученном результате: `assert_select`. Если заглянуть в код, написанный для предыдущих версий Rails, можно наткнуться на утверждение `assert_tag`; оно объявлено устаревшим, и теперь следует пользоваться методом `assert_select`.

Тестирование представлений с помощью функциональных тестов

В методе `assert_select` для формулирования утверждений о HTML- или XML-разметке используется синтаксис CSS-селекторов. Это необычайно гибкий и мощный механизм. Если вы понимаете синтаксис CSS-селекторов (а, будучи разработчиком веб-приложений, должны бы), то, безусловно, предпочтете метод `assert_select` для проверки содержимого представлений[1].

Ниже приведено несколько примеров, заимствованных из документации, которые показывают, как можно проверить, что указанные элементы существуют, имеют заданное текстовое содержимое, содержат заданное число дочерних элементов, следуют в заданном порядке и т. д.:

```
def test_page_has_right_title
  get :index
  assert_select "title", "Добро пожаловать"
end

def test_form_includes_four_input_fields
  get :new
  assert_select "form input", 4
end

def test_show_does_not_have_any_forms_in_it
  get :show, :id => 1
  assert_select "form", false, "На странице show не должно быть форм"
end

def page_has_one_link_back_to_user_page
  get :show, :id => 1
```

[1] Кстати, метод `assert_select` – иллюстрация философии подключаемых модулей в Rails. Поначалу Ассаф Аркин (Assaf Arkin) написал его как подключаемый модуль, но он оказался настолько полезным, что разработчики ядра включили данный метод в версию Rails 1.2 в январе 2007 года.

```
    assert_select "a[href=?]",
                  url_for(:controller=>"user", :id=>user_id),
                  :count => 1, :text => "Назад"
end
```

У метода `assert_select` есть два варианта, а его API – один из самых сложных в Rails. Необычность данного метода в том, что первый параметр необязателен. В обоих вариантах параметры `value`, `equality` и `message`, равно как и `block`, необязательны.

assert_select(selector, [*values, equality, message, &block])

В первом (наиболее употребительном) варианте для выборки из ответа элементов в контекст функционального теста используется строка CSS-селектора `selector`.

assert_select(element, selector, [*values, equality, message, &block])

Во втором варианте (который, как мне кажется, большинство разработчиков применяют редко) в качестве параметра `element` выступает явно заданный экземпляр класса `HTML::Node`, а отбираются все соответствующие селектору элементы, начиная с данного элемента (включительно) и вглубь.

Необязательный параметр block

Если вызвать `assert_select` внутри блока, переданного внешнему `assert_select`, утверждение будет автоматически проверяться для каждого элемента, отбираемого объемлющим утверждением, как показано в следующих примерах, дающих абсолютно идентичные результаты:

```
assert_select "ol > li" do |elements|
  elements.each do |element|
    assert_select element, "li"
  end
end

# более короткая версия
assert_select "ol > li" do
  assert_select "li"
end
```

Справочник по селекторам

Метод `assert_select` понимает форматы CSS-селекторов, перечисленных в табл. 17.1. В утверждении их можно сочетать для отбора одного или нескольких элементов.

Иногда нужно проверить существование экземпляров или подсчитать их число, пользуясь CSS-селектором, а еще и подставить значение. В та-

Таблица 17.1. Справочник по CSS-селекторам

Селектор	Поведение
E	Элемент с именем тега *E* (например, "DIV" соответствует первому найденному элементу DIV$[1]). Ниже *E* может обозначать как имя тега, так и любое другое допустимое CSS-выражение, служащее для выборки одного или нескольких элементов
E F	Элемент *F* является потомком элемента *E* (необязательно непосредственным)
E > F	Элемент *F* — непосредственный потомок элемента *E*. Пример: проверить заголовок на странице регистрации: `Assert_select "html:root > head > title", "Login"`
E ~ F	Элемент *E* предшествует элементу *F*. Пример: проверить, что спонсируемые элементы списка предшествуют бесплатным: `assert_select 'LI.sponsored ~ LI:not(.sponsored)'`
E + F	Элемент *E* непосредственно предшествует элементу *F*. Пример: проверить, что DIV с идентификатором content следует сразу же за DIV header без промежуточных элементов
E.class *E.class.otherclass*	Элементы, которым назначен указанный CSS-класс
E#myid	Элемент с идентификатором *myid*
E[attribute	Элементы, имеющие атрибут с указанным именем
E[attribute="value"]	Элементы, для которых значение указанного атрибута точно совпадает с заданной строкой *value*
E[attribute~="value"]	Элементы, для которых значение указанного атрибута содержит разделенные пробелами строки, одна из которых точно совпадает с *value*
E[attribute^="start"]	Элементы, для которых значение указанного атрибута начинается с заданной строки *start*
E[attribute$="end"]	Элементы, для которых значение указанного атрибута оканчивается заданной строкой *end*
E[attribute="str"]*	Элементы, для которых значение указанного атрибута содержит подстроку *str*
E[attribute|="value"]	Элементы, для которых значение указанного атрибута содержит разделенный дефисами список строк, начинающийся с *value*

[1] CSS-селекторы тегов нечувствительны к регистру. В этой книге мы обычно записываем HTML-теги заглавными буквами просто для удобства восприятия.

Таблица 17.1. Справочник по CSS-селекторам (окончание)

Селектор	Поведение
E:root	Корневой элемент документа
E:first-child	Первый дочерний элемент *E*. Примечание: Этот и другие подобные селекторы отбирают несколько элементов, если *E* отбирает несколько элементов
E:last-child	Последний дочерний элемент *E*
E:nth-child(n) E:nth-last-child(*n*)	n-й дочерний элемент *E*, начиная с первого (в CSS элементы нумеруются с 1). Для варианта last-child отсчет начинается от последнего дочернего элемента
E:first-of-type	Первый брат типа *E*. Пример: проверить, что все раскрывающиеся списки в данном документе включают в начале пустую строку: assert_select('SELECT OPTION:first-of-type','')
E:last-of-type	Последний брат типа *E*
E:nth-of-type(*n*) *E*:nth-last-of-type(*n*)	n-й брат типа *E*. Для варианта last-of-type отсчет начинается от последнего брата. Пример: проверить порядок и содержимое тегов OPTION для следующего элемента SELECT: <SELECT id="filter"> <OPTION>None</OPTION> <OPTION>Businesses</OPTION> <OPTION>People</OPTION> </SELECT> assert_select('SELECT#filter option:nth-of-type(1)', 'None') assert_select('SELECT#filter option:nth-of-type(2)', 'Businesses') assert_select('SELECT#filter option:nth-of-type(3)', 'People')
E:only-child	Элементы, являющиеся единственным потомком своего родителя, сопоставленного с *E*
E:only-of-type	Элементы – единственные братья типа, сопоставленного с *E*. Пример: проверить, что на странице присутствует только одна внешняя ссылка на JavaScript-файл (иногда это правило постулируется для повышения производительности). assert_select('HEAD SCRIPT[src]:only-of-type')
E:empty	Элементы, не имеющие потомков(в том числе и текстовых узлов)
E:not(*selector*)	Элементы, не соответствующие селектору *selector*

ком случае можно воспользоваться замещаемым символом ?. Он применим к именам классов (.?, classname), идентификаторам (.?, id) и обычным атрибутам ([attr=?], string, или regexp):

```
assert_select "form[action=?]", url_for(:action=>"login") do
  assert_select "input[type=text][name=username]"
  assert_select "input[type=password][name=password]"
end
```

Виды сравнения

Параметр equality необязателен и может принимать значения, перечисленные в табл. 17.2. По умолчанию он равен true, и это означает, что был найден хотя бы один элемент, соответствующий CSS-селектору. Если вы хотите задать только один критерий для сопоставленных элементов, пользуйтесь единичной формой. В противном случае передайте в качестве критерия хеш.

Таблица 17.2. Возможные значения параметра equality в методе assert_select

Единичная форма	В виде хеша	Пояснение
true	:minimum => 1	Сопоставлен по крайней мере один элемент
false	:count => 0	Не сопоставлен ни один элемент
"something"	:text => "something"	Все сопоставленные элементы имеют заданное текстовое содержимое
/^[a-z]{2}$/i	:text => /^[a-z]{2}$/i	Все сопоставленные элементы соответствуют регулярному выражению
n	:count => *n* :minimum => *n* :maximum => *n*	Сопоставлено ровно *n* элементов. Сопоставлено по меньшей мере *n* элементов. Сопоставлено не более *n* элементов
n..m	:minimum => *n*, :maximum => *n*	Количество сопоставленных элементов находится в заданном диапазоне

Тестирование поведения RJS

В функциональных тестах для проверки манипуляций представлением в стиле RJS в контроллерах используются варианты утверждения assert_select_rjs.

assert_select_rjs(*args, &block)

Если опустить `args`, то утверждается просто, что с помощью RJS было обновлено или вставлено один или несколько элементов. Во всех вариантах метода `assert_select_rjs` можно использовать вложенные утверждения `assert_select` для проверки HTML-разметки, генерируемой в результате обновления или вставки:

```
# Проверить, что RJS вставляет или обновляет список из четырех элементов
assert_select_rjs 'my_list' do
  assert_select "ol > li", 4
end
```

assert_select_rjs(id)

То же, что `assert_select_rjs(*args, &block)`, но задает конкретный идентификатор *id* элемента, подлежащего обновлению или вставке.

assert_select_rjs(operation, id)

То же, что `assert_select_rjs(*args, &block)`, но задает *операцию*, применяемую к элементу с заданным идентификатором *id*. Параметр `operation` может принимать значения `replace`, `chained_replace`, `replace_html`, `chained_replace_html` или `insert_html`.

assert_select_rjs(:insert, position, id)

То же, что `assert_select_rjs(*args, &block)`, но говорит, что операция RJS равна `:insert`. Параметр `position` может принимать значения `:top`, `:bottom`, `:before` или `:after`.

Другие методы выборки

В дополнение к методу `assert_select` существуют методы для проверки электронной почты и кодированного HTML-кода, а также метод `css_select`, полезный в сочетании с версией `assert_select`, которая принимает экземпляр `HTML::Node` в качестве первого параметра.

assert_select_email(*args, &block)

Утверждение относительно тела доставленного почтового сообщения (в формате HTML).

assert_select_encoded(*args, &block)

Для операций с кодированным HTML, например, в описаниях статей в RSS-каналах.

css_select(selector, *values) и css_select(element, selector, *values)

Возвращают массивы выбранных элементов, которые будут пусты, если ни один элемент не выбран.

Тестирование правил маршрутизации

В Rails имеется набор применяемых в функциональных тестах утверждений для проверки того, что файл маршрутов сконфигурирован должным образом.

assert_generates(expected_path, options, defaults={}, extras = {}, message=nil)

Проверяет, что переданные в хеше `options` параметры подходят для генерации заданного пути. Это утверждение противоположно `assert_recognizes`:

```
assert_generates("/items", :controller => "items",
                 :action => "index")

assert_generates("/items/list", :controller => "items",
                 :action => "list")

assert_generates("/items/list/1", :controller => "items",
                 :action => "list", :id => "1")
```

assert_recognizes(expected_options, path, extras={}, message=nil)

Проверяет, что заданный путь корректно маршрутизирован и соответствует параметрам, переданным в хеше `options`. Это утверждение противоположно `assert_generates`:

```
# проверить действие по умолчанию
assert_recognizes({:controller => 'items', :action => 'index'},
'items')

# проверить конкретное действие
assert_recognizes({:controller => 'items', :action => 'list'},
'items/list')

# проверить действие с параметром
assert_recognizes({:controller => 'items', :action => 'list',
:id => '1'}, 'items/list/1')
```

Во втором параметре можно передать хеш, определяющий метод запроса. Это полезно для маршрутов, требующих конкретного HTTP-

метода. Хеш должен содержать ключ :path, значением которого является путь во входящем запросе, и ключ :method, описывающий требуемый глагол HTTP:

```
# проверить, что запрос POST с путем /items вызывает действие create
# контроллера ItemsController
assert_recognizes({:controller => 'items', :action => 'create'},
                  {:path => 'items', :method => :post})
```

Кроме того, в хеше extras можно передать параметры, которые обычно должны присутствовать в строке запроса. Это полезно для проверки того, что параметры из строки запроса правильно помещаются в хеш params. Для тестирования строки запроса следует пользоваться только аргументом extras; дописывание строки в конец пути (что обычно и делается в коде приложения) работать не будет:

```
# проверить, что путь '/items/list/1?view=print' возвращает правильные
# параметры
assert_recognizes({:controller => 'items', :action => 'list',
                   :id => '1', :view => 'print'},
                  'items/list/1', { :view => "print" })
```

assert_routing(path, options, defaults={}, extras={}, message=nil)

Проверяет, что путь и параметры соответствуют друг другу в обоих направлениях. Иными словами, URL, сгенерированный из параметров в хеше options, совпадает с path, а параметры, извлеченные из пути path, идентичны параметрам в хеше options. По существу это сочетание assert_recognizes и assert_generates в одном утверждении.

Тесты сопряжения в Rails

В Rails 1.1 появился API для *тестирования сопряжений* как «логическое продолжение уже имеющихся видов тестирования»[1].

Тесты сопряжений отличаются от функциональных тем, что проверяют взаимодействия для последовательности из нескольких запросов от броузера, адресованных различным контроллерам и действиям. Помимо проверки функциональности приложения хорошо написанные тесты сопряжений должны выявлять ошибки при использовании маршрутизации и пользовательских сеансов. При таком подходе тесты сопряжений в Rails *можно было бы* считать подходящей заменой приемочных тестов для вашего проекта.

---

[1] Хорошее введение написал Джеймис, см. *http://jamis.jamisbuck.org/articles/2006/03/09/integration-testing-in-rails-1-1*.

Основы

Тесты сопряжений хранятся в виде файлов в каталоге `test/integration`. Они запускаются с помощью Rake-задания `test:integration` или, как и все прочие тесты Rails, путем исполнения файла непосредственно в интерпретаторе Ruby.

Вам не придется писать трафаретный код теста сопряжений самостоятельно, поскольку имеется генератор. Укажите в качестве аргумента генератора имя теста, записанное в ВерблюжьейНотации или с_подчерками:

```
$ script/generate integration_test user_groups
     exists test/integration/
     create test/integration/user_groups_test.rb
```

Генератор создаст шаблон теста сопряжений, пригодный для редактирования, например:

```
require "#{File.dirname(__FILE__)}/../test_helper"

  class UserGroupsTest < ActionController::IntegrationTest
    # fixtures :your, :models

    # Заменить реальными тестами.
    def test_truth
      assert true
    end
  end
end
```

До этого момента все очень напоминает написание тестов для моделей и контроллеров, но дальше нужно решить, как мы собираемся реализовать подлежащую тестированию функцию. В этом смысле написание теста сопряжения до реализации функции можно рассматривать как инструмент проектирования и специфицирования. Прежде чем продолжить пример, давайте познакомимся с методами API, применяемыми при кодировании тестов сопряжения.

API тестов сопряжения

Методы `get` и `post` в качестве первого аргумента принимают строку-путь, а в качестве второго – параметры запроса в виде хеша. Если хотите передать в первом параметре хорошо вам знакомый хеш «контроллер/действие», воспользуйтесь методом `url_for`. Метод `follow_redirect!` говорит, что тест должен следовать по адресу, сформированному в последнем запросе. Метод `status` возвращает код состояния HTTP для последнего запроса. В утверждении о переадресации метод `redirect?` проверяет, что код состояния для последнего запроса равен 300.

Модуль `ActionController::Assertions::ResponseAssertions` содержит утверждения, которые мы будем использовать в тестах сопряжений. Напомню, что у всех методов-утверждений в Rails имеется необязательный параметр – сообщение, отображаемое, когда при прогоне теста утверждение оказывается ложным.

assert_redirected_to(options = {}, message = nil)

Проверяет, соответствуют ли переданные в хеше `options` параметры переадресации вызову метода переадресации в последнем действии. Соответствие может быть неполным, например `assert_redirected_to (:controller => "weblog")` также будет соответствовать переадресации `redirect_to(:controller => "weblog", :action => "show")` и т. д.

assert_response(type, message = nil)

Убеждается, что код состояния в HTTP-ответе соответствует заданному критерию. Вместо целочисленного значения (например, `assert_response(501)`) можно передавать символы из следующего списка:

- `:success` – код состояния равен 200
- `:redirect` – код состояния в диапазоне 300–399
- `:missing` – код состояния равен 404
- `:error` – код состояния в диапазоне 500–599

assert_template(expected = nil, message = nil)

Проверяет, что для ответа на запрос был выполнен рендеринг подходящего шаблона.

Работа с сеансами

Экземпляр `Session` в тесте сопряжения представляет набор запросов и ответов, выполненных последовательно неким виртуальным пользователем. Поскольку можно создать несколько таких сеансов и запускать их параллельно, то удается (до некоторой степени) смоделировать одновременно работающих пользователей.

Обычно новый сеанс создается методом `IntegrationTest#open_session`, а не в результате прямого инстанцирования экземпляра класса `Integration::Session`.

Задания Rake, относящиеся к тестированию

По умолчанию Rakefile для проектов Rails включает 10 заданий, относящихся к тестированию. Все они перечислены в табл. 17.3.

Таблица 17.3. Задания Rake, относящиеся к тестированию

Цель	Описание
`rake db:test:clone`	Задание `clone` заново создает тестовую базу данных согласно схеме, применяемой в текущем режиме
`rake db:test:clone_structure`	Задание `clone_structure` создает тестовую базу данных, повторяя структуру базы данных в режиме разработки. Аналогично `db:test:clone`, но копирует лишь схему базы, а не ее содержимое. В ходе повседневной работы вряд ли возникнет необходимость в запуске любого из этих заданий, поскольку они являются зависимостями для других заданий тестирования
`rake db:test:prepare`	Подготавливает тестовую базу данных к прогону тестов и загружает в нее схему текущей базы данных режима разработки. Если вы прогоняете тест сразу после внесения изменений в схему базы данных, то должны сначала выполнить это задание, иначе тест не пройдет
`rake db:test:purge`	Очищает тестовую базу данных
`rake test`	Цель `test` является целью по умолчанию в стандартных Rakefile, то есть для ее выполнения достаточно набрать просто команду `rake` без параметров. При этом прогоняются все тесты, находящиеся в папках `test/units` и `test/functionals`
`rake test:functionals`	Прогоняет только тесты в папке `test/functionals`
`rake test:integration`	Прогоняет только тесты в папке `test/integration`
`rake test:units`	Прогоняет только тесты в папке `test/units`
`rake test:recent`	Прогоняет только тесты, которые были модифицированы в течение последних 10 мин.
`rake test:uncommitted`	Прогоняет только тесты, которые были модифицированы по сравнению с версиями, хранящимися в системе Subversion

Приемочные тесты

Хорошо написанный комплект приемочных тестов – необходимая составная часть успеха любого сложного программного проекта, особенно разрабатываемого в соответствии с принципами гибкого (agile) программирования и такими методиками, как экстремальное программирование. Одно из лучших определений *приемочного теста* я видел на официальном сайте Extreme Programming:

> *Заказчик описывает подлежащие тестированию сценарии после того, как история пользователя (user story) была корректно реализована. История может содержать один или несколько приемочных тестов – столько, сколько необходимо для проверки работоспособности системы*[1].

Проще говоря, приемочные тесты позволяют убедиться в том, что мы реализовали данную функцию, или – на жаргоне экстремального программирования – историю пользователя. Традиционно приемочные тесты состоят из ряда тестовых сценариев – последовательности действий, выполняемых тестировщиком (или поданных на вход инструмента тестирования), которые проверяют правильность работы приложения. Сегодня полагаться на ручное приемочное тестирование вебприложений считается дурным тоном. Это медленно, чревато ошибками и дорого.

Индивидуальный приемочный тест может быть полезен разработчику, который пишет его как инструмент проектирования и организации прохождения задач. Полный комплект автоматизированных приемочных тестов показывает, в какой степени реализована требуемая функциональность системы, и может служить непрерывным индикатором близости проекта к завершению.

Приемочные тесты с самого начала

Для меня как разработчика, занимающегося данной историей пользователя или функцией, наличие автоматизированных приемочных тестов позволяет сосредоточиться на порученной задаче и поддерживать продуктивность на оптимальном уровне. В них применимы те же принципы разработки, управляемой тестами, но на более высоком концептуальном уровне, чем позволяют автономные тесты. В Rails многие вещи делаются настолько просто, что может оказаться трудно ограничить себя работой одновременно только над одной задачей, а это опасная привычка.

Кроме того, опыт показывает, что в большинстве приложений Rails (и приложений для Web 2.0 вообще) наибольшее внимание уделяется работе с данными и пользовательскому интерфейсу. В них просто недостаточно бизнес-логики, оправдывающей надобность большого числа автономных тестов. Однако оставлять какие-то части приложения без тестового покрытия нельзя – в интерпретируемом языке, каковым является Ruby, это значило бы искать приключений на свою голову.

Тут-то и приходят на помощь автоматизированные приемочные тесты: закодируйте критерии приемлемости для данной истории пользователя или функции до того, как приступать к ее реализации. Требования неясны? Уточняйте. Не знаете, как следует проектировать модели и конт-

[1] Определение приемочного тестирования см. на странице *http://www.extremeprogramming.org/rules/functionaltests.html*.

роллеры? Займитесь этим. Примите какие-то отправные решения и положите их в основу теста.

Пока работа над реализацией по-настоящему не начата, приемочный тест должен отказывать в самом начале. По мере продвижения вперед снова и снова прогоняйте тест, наблюдая как красный индикатор сменяется зеленым. Когда весь тест позеленеет, можете праздновать победу! В процессе этой работы вы создали автоматизированный комплект регрессионных тестов, а в этом есть масса положительных сторон.

К счастью, поскольку очень многие программисты Rails – фанаты гибкого программирования, существуют отличные инструменты, позволяющие легко и быстро создавать комплекты приемочных тестов.

Здесь мы рассмотрим самые известные из них, начав с системы Selenium on Rails – продукта с открытыми исходными текстами. Затем познакомимся со средствами приемочного тестирования, встроенными в саму среду Rails, и закончим описанием еще некоторых полезных инструментов.

Система Selenium

Selenium – это название семейства инструментов тестирования, которые работают прямо в броузере и точно повторяют действия реального пользователя. Разрабатывалась она группой программистов и тестировщиков в компании ThoughtWorks и проектировалась специально под требования к приемочному тестированию, выдвигаемые адептами гибкого программирования.

Основы

Сценарий в Selenium состоит из последовательности команд, которые изменяют и проверяют состояние броузера. Есть три вида команд: действия, акцессоры и утверждения. Поначалу вы будете иметь дело преимущественно с действиями и утверждениями. Большинство команд принимают два параметра: *target* и *value*.

Некоторые команды ожидают, пока не станет истинным заданное условие. Это позволяет системе Selenium тестировать переходы между страницами и функциональность Ajax. Такие команды сразу же завершаются успешно, если проверяемое условие уже истинно. Если же условие так и не стало истинным до истечения тайм-аута, они завершаются неудачно и останавливают тест.

Действия и утверждения

Действиями называются команды, которые манипулируют состоянием приложения, например «открыть URL» или «щелкнуть по указанной гиперссылке». Если действие завершается отказом или ошибкой, выполнение теста прекращается.

Действия с суффиксом AndWait, например `click_and_wait`, говорят Selenium, что броузер сейчас обратится к серверу и надо подождать, пока загрузится новая страница или завершится вызов Ajax.

Утверждения служат для проверки того, что состояние приложения отвечает ожиданиям. Например, «убедиться, что заголовок страницы равен X» или «проверить, что флажок отмечен». Все утверждения Selenium можно использовать в трех режимах: `assert`, `verify` и `wait_for`. Так, можно написать `assert_text`, `verify_text` или `wait_for_text`.

Если утверждение оказывается ложным, выполнение теста прекращается. В случае `verify` тест протоколирует отказ, но продолжает выполняться. Поэтому используйте `assert`, если надо убедиться, что текущее состояние допускает продолжение тестирования, и `verify`, если необходимо проверить, скажем, значения в полях формы, а завершать тест в случае обнаружения отличий не обязательно.

Локаторы

Цель действий и утверждений в Selenium – некоторый HTML-элемент на странице, который задается с помощью локатора. Существуют различные виды строк-локаторов с собственными соглашениями, но наиболее употребительные из них ядро системы понимает автоматически. Локаторы, начинающиеся со строки `document.`, считаются обычным выражением, обозначающим путь в DOM, как в JavaScript. Локаторы, начинающиеся с //, трактуются как путь в DOM в смысле XPath. Все остальные локаторы (без префикса) считаются идентификаторами (атрибут `id` элемента).

Образцы

Образцы служат для задания ожидаемого значения произвольного текста на странице, поля формы, конкретного узла в разметке, атрибута элемента и т. д. Selenium поддерживает различные виды образцов, в том числе регулярные выражения, но в большинстве случаев образец является обычной строкой.

Справочные материалы по Selenium

Полное справочное руководство по Selenium имеется в Сети[1]. В этой главе мы кратко опишем порядок использования команд, но это введение нельзя считать исчерпывающим руководством.

Приступая к работе

Selenium on Rails[2] – это название продукта из семейства Selenium, специально предназначенного для разработчиков на платформе Rails. Он

[1] Справочное руководство по Selenium находится по адресу *http://release.openqa.org/selenium-core/nightly/reference.html*.

[2] Подключаемый модуль Selenium on Rails находится по адресу *http://openqa.org/selenium-on-rails/*.

распространяется в виде подключаемого модуля к Rails и отлично интегрируется со структурой тестирования и фикстурами.

Установить Selenium on Rails и начать с ним работать очень просто:

1. Установить основные файлы Selenium, необходимые подключаемому модулю: `gem install selenium`.
2. Установить подключаемый модуль Selenium on Rails для данного проекта: `script/plugin install http://svn.openqa.org/svn/selenium-on-rails/selenium-on-rails/`.
3. Сгенерировать тестовый каталог и тестовый сценарий: `script/generate selenium first.rsel`.
4. Запустить сервер Rails в тестовом режиме: `server -e test`.
5. Открыть в броузере приложение Selenium Test Runner: *http://localhost:3000/selenium*.

Если установка прошла успешно, вы увидите в броузере приложение Selenium Test Runner. Вы уже должны понимать, что тесты Selenium можно запускать из командной строки и интегрировать в автоматизированный комплект тестов, но для демонстрации воспользуемся для их выполнения встроенным веб-интерфейсом.

Тест First

Щелкните по кнопке All (Все) на панели управления Execute Tests (Выполнить тесты). Selenium выполнит тестовый сценарий First, отображаемый в средней верхней панели интерфейса. Если заголовок начальной страницы вашего приложения Rails отличен от Home, тест завершится неудачно.

Первые две команды теста отображаются на светло-зеленом фоне. Это означает, что они выполнились правильно. Однако фон ячейки заголовка таблицы и строки, содержащей команду `assertTitle`, светло-красный, что означает отказ. На панели управлении в правом верхнем углу интерфейса показывается статус тестового прогона.

Чтобы тест прошел успешно, откройте файл `test/selenium/first.rsel` и измените команду `assert_title`, чтобы она выполняла сравнение с настоящим значением элемента `title` вашего сайта. У себя я изменил его на Ruby on Rails: Welcome aboard:

```
setup
open '/'
assert_title 'Ruby on Rails: Welcome aboard'
```

Вернемся к броузеру и обновим страницу, чтобы изменения вступили в силу. Теперь тест должен пройти успешно и окраситься в зеленый цвет – цвет удачи!

RSelenese

Сценарии Selenium on Rails написаны на Ruby с применением API, который авторы назвали RSelenese. Это прямой перенос языка команд Selenium в стиле Ruby – идентификаторы написаны строчными буквами с подчерками, а не в ВерблюжьейНотации. Ядро Selenium on Rails ищет тесты RSelenese в дереве, начинающемся от каталога `test/selenium`. Это файлы с расширением `.rsel`.

Selenium on Rails также понимает и исполняет сценарии в формате HTML, находящиеся в файле с расширением `.sel`. Однако на практике лучше писать бóльшую часть приложения на Ruby, и тесты в этом смысле – не исключение. Кроме того, поскольку язык RSelenese – просто Ruby, вы можете пользоваться обычными языковыми конструкциями, например условными предложениями и циклами, чтобы повысить выразительность.

В примере ниже показано использование итератора в RSelenese для последовательного открытия десяти страниц:

```
(1..10).each do |num|

  open :controller => 'user', :action => 'list', :page => num
end
```

Частичные сценарии

Нередко бывает, что в комплекте приемочных тестов имеются части, общие для всех тестовых сценариев. Чтобы не отступать от принципа DRY, можно определить частичные сценарии и включать их в другие файлы. При наличии нескольких общих действий, выполняемых в различных тестах, можно поместить их в отдельный включаемый тестовый сценарий.

Частичный тестовый сценарий похож на обычный, только имя содержащего его файла начинается с подчерка:

```
#_login.rsel
open '/login'
type 'name', 'john'
type 'password', 'password'
click 'submit'
```

Для включения частичного тестового сценария служит метод `include_partial`:

```
include_partial 'login'
```

Возможность передавать в частичные сценарии переменные делает их еще полезнее. Пусть, например, мы хотим воспользоваться частичным сценарием `login` не только для пользователя `john`. Заведем в нем локальные переменные, играющие роль формальных параметров...

```
#_login.rsel with params
open '/login'
type 'name', name
type 'password', password
click 'submit'
```

... и затем передадим в хеше фактические параметры при вызове `include_partial`:

```
include_partial 'login', :name => 'jane', :password => 'foo'
```

Заключение

Эта глава оказалась для меня одной из самых трудных – наверное, потому что обилие материала тянет на отдельную книгу. Мы немного поговорили о тестировании в Rails и упомянули, что во избежание ненужных проблем следует разумно пользоваться фикстурами и применять сторонние библиотеки для создания Mock-объектов, например Mocha. Мы также обсудили различия между автономными и функциональными тестами, тестами сопряжений и приемочными тестами в Rails. Попутно были представлены справочные материалы, которые помогут вам в повседневной работе.

Так получилось, что многие разработчики решили вовсе отказаться от тестирования в Rails и примкнуть к несколько иной школе мышления в части верификации проектов. Эта философия называется *разработкой, управляемой поведением* (behavior-driven development), а поддерживающая ее библиотека – *RSpec*.

18

RSpec on Rails

Не думаю, что человеческое сердце может быть охвачено бóльшим волнением, чем сердце изобретателя, который видит, как творение его разума воплощается в жизнь.

Никола Тесла

RSpec – это предметно-ориентированный язык, написанный на Ruby и предназначенный для специфицирования требуемого поведения Ruby-приложения. Его сильная сторона заключается в том, что RSpec-сценарии очень легко читаются и позволяют авторам выражать свои намерения более ясно и элегантно, чем с помощью методов и утверждений из класса `Test::Unit`.

Продукт *RSpec on Rails*, призванный заменить подсистемы тестирования Rails, был выпущен в конце 2006 года. В нем есть функции верификации, подделывания (mocking) и создания заглушек, адаптированные для использования с моделями, контроллерами и представлениями Rails. Рад сообщить, что я пользуюсь в своих проектах подключаемым модулем *RSpec* с момента его появления, и с тех пор ни разу не притрагивался к `Test::Unit` для решения сколько-нибудь важных задач[1].

[1] Если не считать написания главы 17, в которой рассматривается класс `Test::Unit`.

Введение в RSpec

Поскольку RSpec-сценарии (или просто «спеки») так легко читаются, я не могу придумать лучшего способа познакомить вас с этой структурой, чем просто представить реальные спецификации. Попутно отмечу ключевые идеи, о которых необходимо знать.

В листинге 18.1 приведен фрагмент реального RSpec-сценария, определяющего поведение класса модели `CreditCard`.

Листинг 18.1. Блок описания из спецификации Spec2 модели CreditCard в системе Monkeycharger[1]

```
 1 describe "A valid credit card" do
 2    before(:each) do
 3      @credit_card = generate_credit_card
 4    end
 5
 6    it "should be valid after saving" do
 7      @credit_card.save
 8      @credit_card.should be_valid
 9    end
10 end
```

RSpec-сценарии – это наборы *поведений*, которые в свою очередь являются наборами примеров. В строке 1 метод `describe` незримо создает объект `Behavior`. Поведение задает контекст для набора примеров, и вы должны передать фрагмент предложения, точно описывающий контекст, который собираетесь специфицировать.

Метод `before` в строке 2 (и парный ему метод `after`) сродни методам `setup` и `teardown` в xUnit-структурах типа `Test::Unit`. Они позволяют установить определенное состояние до прогона примера и, если необходимо, произвести очистку после прогона. Данному поведению блок `after` не нужен. (Для краткости исходный текст метода `generate_credit_card`, вызываемого в строке 3, в листинг не включен. Это просто фабричный метод, возвращающий экземпляры `CreditCard` с известными, но допускающими переопределение атрибутами. Ниже в этой главе мы еще будем говорить об использовании методов-помощников для написания удобного для восприятия кода «спеков».)

Метод `it` в строке 6 создает объект `Example`, и вы должны предоставить его описание. Идея в том, чтобы завершить предложение, начатое в методе `describe`. В нашем случае получится A valid credit card should be

[1] Скачать проект Monkeycharger можно с сайта *http://monkeycharger.googlecode.com/*.

valid after saving (Допустимая кредитная карта должна оставаться допустимой после сохранения). Смысл понятен?

Обязанности и ожидания

Идем дальше. В строке 7 листинга 18.1 для объекта кредитной карты вызывается метод `save`, а поскольку это модель `ActiveRecord`, будет произведен контроль данных. Нам осталось лишь убедиться, что валидатор признал экземпляр `CreditCard` допустимым. Но вместо утверждений в стиле xUnit в RSpec применяется «прикольный» синтаксис в духе DSL, в основе которого лежат методы с именами `should` и `should_not`.

Во время выполнения RSpec подмешивает методы `should` и `should_not` в базовый класс Ruby `Object`, поэтому они оказываются доступны всем объектам. Оба метода ожидают получить объект `Matcher`, который вы порождаете с помощью синтаксиса «ожиданий».

```
@credit_card.should be_valid
```

Есть несколько способов сгенерировать такие объекты и передать их методу `should` (или `should_not`):

```
receiver.should(matcher)                    # простейший пример
# Проходит, если matcher.matches?(receiver)

receiver.should == expected # произвольное значение
# Проходит, если (receiver == expected)

receiver.should === expected # произвольное значение
# Проходит, если (receiver === expected)

receiver.should =~ regexp
# Проходит, если (receiver =~ regexp)
```

Обучение тому, как писать «ожидания», наверное, самая содержательная часть RSpec. Одна из наиболее употребительных идиом – should equal (должно быть равно) – близка к утверждению `assert_equal` в `Test::Unit`. Вот как можно было бы переписать утверждение credit card should be valid (кредитная карта должна быть допустимой) с применением синтаксиса `should equal`:

```
@credit_card.valid?.should == true
```

Метод `valid?` возвращает `true` или `false` и, согласно нашему «спеку», должен возвращать `true`. Но почему мы с самого начала не записали ожидание в таком виде? Просто потому, что стоящие подряд вопросительный знак и точка выглядят коряво. Работает, конечно, но не так элегантно и читаемо, как `should be_valid`. В RSpec не существует предопределенного метода `be_valid` – вместо него следует подставить *произвольный предикат*.

Предикаты

Благодаря магии механизма method_missing RSpec способна поддержать произвольные предикаты, то есть понимает, что если вы вызываете какой-то метод, начинающийся с be_, то следующая далее часть имени должна считаться указателем на булев атрибут целевого объекта.

В простейших предикатах целью являются булевы значения и nil:

```
target.should be_true
target.should be_false
target.should be_nil
target.should_not be_nil
```

Но возможны и произвольные булевы предикаты, даже с параметрами!

```
collection.should be_empty # проходит, если target.empty?
target.should_not be_empty # проходит, если не target.empty?
target.should_not be_under_age(13) # проходит, если не
                                   # target.under_age?(13)
```

Вместо префикса be_ в предикатах можно также употреблять неопределенный артикль, если при этом «спек» звучит более естественно:

```
"a string".should be_an_instance_of(String)
3.should be_a_kind_of(Fixnum)
3.should be_a_kind_of(Numeric)
3.should be_an_instance_of(Fixnum)
3.should_not be_instance_of(Numeric) # не проходит
```

Однако изобретательность (сумасшествие?) на этом не заканчивается. RSpec понимает даже префикс have_, например, в предикатах типа has_key?:

```
{:foo => "foo"}.should have_key(:foo)
{:bar => "bar"}.should_not have_key(:foo)
```

В RSpec есть ряд «верификаторов» (matcher) ожиданий для работы с классами, реализующими модуль Enumerable. Можно проверить, содержит ли массив конкретный элемент или входит ли в строку некая подстрока:

```
[1, 2, 3].should include(1)
[1, 2, 3].should_not include(4)
"foobar".should include("bar")
"foobar".should_not include("baz")
```

На десерт подается толика синтаксической глазури для проверки длины набора:

```
[1, 2, 3].should have(3).items
```

А что если вы хотите специфицировать длину набора has_many? Конструкция Schedule.days.should have(3).items, пожалуй, выглядит малосимпатично. RSpec и тут готова подсластить пилюлю:

```
schedule.should have(3).days # проходит, если schedule.days.length == 3
```

Нестандартные верификаторы ожиданий

Если ни один из готовых верификаторов не позволяет выразить ваше ожидание естественным образом, можно без труда написать собственный. Требуется лишь реализовать класс Ruby со следующими четырьмя методами (обязательны из них только два):

```
matches?(actual)
failure_message
negative_failure_message # необязателен
description # необязателен
```

В документации по RSpec API в качестве примера приводится игра, участники которой могут находиться в различных зонах виртуальной доски. Следующий «спек» говорит, что игрок `bob` должен быть в зоне 4:

```
bob.current_zone.should eql(Zone.new("4"))
```

Однако более выразительно выглядит такая запись с применением нестандартного верификатора, представленного в листинге 18.2:

```
bob.should be_in_zone("4") and bob.should_not be_in_zone("3")
```

Листинг 18.2. Класс нестандартного верификатора ожиданий BeInZone

```
class BeInZone
  def initialize(expected)
    @expected = expected
  end
  def matches?(target)
    @target = target
    @target.current_zone.eql?(Zone.new(@expected))
  end
  def failure_message
    "expected #{@target.inspect} to be in Zone #{@expected}"
  end
  def negative_failure_message
    "expected #{@target.inspect} not to be in Zone #{@expected}"
  end
end
```

Помимо класса верификатора, необходимо еще написать следующий метод, который должен находиться в области видимости «спека»:

```
def be_in_zone(expected)
  BeInZone.new(expected)
end
```

Обычно это делается путем включения и метода, и класса в модуль, который затем включается в «спек»:

```
describe "Player behaviour" do
  include CustomGameMatchers
  ...
end
```

Можно включить помощников глобально в файл `spec_helper.rb`, который в файле «спека» затребуется предложением `require`:

```
Spec::Runner.configure do |config|
  config.include(CustomGameMatchers)
end
```

Отметим, что думать об экземплярах классов `Behavior` и `Example` при написании RSpec-сценариев не нужно (они используются только внутри самой структуры).

Несколько примеров для одного поведения

Представленное в листинге 18.1 описание из системы Monkeycharger довольно простое, в нем есть только один пример. Но это лишь потому, что я не хотел усложнять введение в основные концепции RSpec.

Обычно в поведении бывает больше одного примера. Проще всего показать дополнительный код из спецификации Monkeycharger. В листинге 18.3 приведен следующий блок `describe` из «спека» `CreditCard`, в котором уже целых пять примеров.

Листинг 18.3. Еще один блок описания из спецификации CreditCard в системе Monkeycharger

```
describe CreditCard do
  # должен быть правильный месяц
  it "should have a valid month" do
    card = generate_credit_card(:month => 'f')
    card.errors.on(:month).should == "is not a valid month"
  end

  # должен быть правильный год
  it "should have a valid year" do
    card = generate_credit_card(:year => 'asdf')
    card.errors.on(:year).should == "is not a valid year"
  end

  # дата не должна быть в прошлом
  it "date should not be in the past" do
    past_month = (Date.today << 2)
    card = generate_credit_card(:year => past_month.year,
                                :month => past_month.month)
    card.should_not be_valid
  end

  # полное имя должно состоять из двух слов
  it "should have two words in the name" do
    card = generate_credit_card(:name => "Sam")
    card.errors.on(:name).should == "must be two words long."
  end
```

```
    # фамилия должна состоять из двух слов, если в полном имени три слова
    it "should have two word last_name if name is three words long" do
      card = generate_credit_card(:name => "Sam Van Dyk")
      card.last_name.should == "Van Dyk"
    end

    # имя должно состоять из двух слов, если в полном имени три слова
    it "should have one word first_name if name is three words long" do
      card = generate_credit_card(:name => "Sam Van Dyk")
      card.first_name.should == "Sam"
    end
  end
```

Даже если вы почти ничего не знаете о кредитных картах (или об RSpec), то все равно без труда прочитаете эту спецификацию.

В традиционных RSpec-сценариях методу `describe` обычно необходимо передать строку. Но в листинге 18.3 показан «спек» для модели `ActiveRecord`, написанный в контексте RSpec on Rails, поэтому в описании можно передать не строку, а класс модели (мы еще вернемся к этому вопросу, когда дойдем до специфики Rails; а пока обсуждаем основы RSpec).

Разделяемые поведения

Часто необходимо специфицировать несколько поведений, у которых есть нечто общее. Было бы глупо писать один и тот же код снова и снова. Программисты предпочитают выносить общий код в отдельные методы. Однако проблема в том, что поведение RSpec состоит из многих кусочков:

- `before(:all)`
- `before(:each)`
- `after(:each)`
- `after(:all)`
- все ожидания
- все включаемые модули

Даже если выполнить рефакторинг добротно, останется много дублирования. К счастью, RSpec предоставляет так называемые *разделяемые поведения*. Разделяемое поведение не запускается само по себе, а включается в другие поведения. Для этого мы передаем методу `describe` параметр `:shared => true`.

Пусть нужно специфицировать два класса: `Teacher` и `Student`. Помимо уникального у них есть и некое общее поведение, которое и представляет для нас интерес. Вместо того чтобы описывать его дважды, мы можем создать разделяемое поведение и включить его в спецификацию каждого класса:

```
describe "people in general", :shared => true do
  it "should have a name" do
  @person.name.should_not be_nil
end

it "should have an age" do
  @person.age.should_not be_nil
  end
end
```

Откуда берется переменная экземпляра `@person`? Мы же нигде не присваивали ей значение. Оказывается, что этот «спек» не запустится, поскольку и запускать-то еще нечего. Назначение разделяемого поведения – вынести общий код из спецификаций разных поведений. Нам еще предстоит написать «спек», в котором это разделяемое поведение будет использоваться:

```
describe Teacher do
  before(:each) do
    @person = Teacher.new("Ms. Smith", 30, 50000)
  end

  it_should_behave_like "people in general"

  it "should have a salary" do
    @person.salary.should == 50000
  end
end
```

Метод `it_should_behave_like` принимает в качестве аргумента строку. Затем RSpec находит разделяемое поведение с таким именем и включает его в спецификацию `Teacher`.

То же самое делается и для класса `Student`:

```
describe Student do
  before(:each) do
    @person = Student.new("Susie", 8, "pink")
  end

  it_should_behave_like "people in general"

  it "should have a favorite color" do
    @person.favorite_color.should == "pink"
  end
end
```

Вызвав команду `spec` с параметром `–format specdoc` (или в сокращенной форме `-f s`), мы увидим, что разделяемое поведение действительно включено в спецификации обоих классов:

```
Teacher
- should have a name
```

```
- should have an age
- should have a salary

Student
- should have a name
- should have an age
- should have a favorite color
```

Важно отметить, что на момент написания этой книги RSpec вызывает методы `before` и `after` в том порядке, в котором они определены в спецификации. Убедиться в этом можно, добавив в каждом случае отладочную печать:

```
describe "people in general"
  before(:each) do
    puts "shared before()"
  end

  after(:each) do
    puts "shared after()"
  end
  ...
end

describe Teacher do
  before(:each) do
    puts "teacher before()"
    @person = Teacher.new("Ms. Smith", 30, 50000)
  end

  after(:each) do
    puts "teacher after()"
  end

  it_should_behave_like "people in general"
  ...
end
```

В результате будет выведено:

```
teacher before()
shared before()
teacher after()
shared after()
.
```

Перенесите предложение `it_should_behave_like` в начало «спека» и убедитесь, что теперь первым вызывается метод `before` разделяемого поведения.

Mock-объекты и заглушки в RSpec

В главе 17 мы познакомились с понятием Mock-объекта и заглушки в контексте библиотеки Mocha. В RSpec эти идеи также интенсивно применяются[1]. Библиотеки Mocha и RSpec можно использовать совместно, но в наших примерах мы ограничимся только механизмами RSpec, которые ни в чем не уступают Mocha. В общем-то, они практически одинаковы, разве что имена методов немного различаются.

Mock-объекты

Для создания Mock-объекта достаточно в любом месте «спека» вызвать метод `mock`, передав ему имя в качестве необязательного параметра. Если в «спеке» используется несколько Mock-объектов, настоятельно рекомендуется давать им имена. При наличии нескольких анонимных Mock-объектов будет трудно отличить один от другого в случае ошибки:

```
echo = mock('echo')
```

Напомним, что вы должны указать, какие сообщения Mock-объект предположительно может получать во время исполнения «спека». Если эти ожидания не оправдаются, Mock-объект вызовет отказ «спека». Там, где в Mocha ожидаемые сообщения описываются словом `expects`, в RSpec мы пишем `should_receive` или `should_not_receive`:

```
echo.should_receive(:sound)
```

В обеих библиотеках имеется сцепляемый метод `with`, с помощью которого можно задавать ожидаемые параметры. Но, если в Mocha для описания возвращаемого значения употребляется слово `returns`, то в RSpec – `and_return`. Настолько похоже, что перейти от одной библиотеки к другой не составит никакого труда (если такая необходимость возникнет):

```
echo.should_receive(:sound).with("hey").and_return("hey")
```

Null-объекты

Иногда для тестирования хочется найти объект, который принимает любое переданное ему сообщение. Этот паттерн называется *Null-объектом*. Такой объект можно создать методом `mock`, если передать ему параметр `:null_object`.

```
null_object = mock('null', :null_object => true)
```

---

[1] Вам непонятно, в чем разница между Mock-объектами и заглушками? Читайте объяснение Мартина Фаулера на странице *http://www.martinfowler.com/articles/mocksArentStubs.html*.

Объекты-заглушки

Объекты-заглушки в RSpec создаются фабричным методом `stub`. Ему передается имя (как и для Mock-объекта) и хеш с атрибутами по умолчанию:

```
yodeler = stub('yodeler', :yodels? => true)
```

Кстати, имя Mock-объекта или заглушки необязательно должно быть строкой. Довольно часто методу `mock` или `stub` передают ссылку на класс, соответствующий истинному типу объекта:

```
yodeler = stub(Yodeler, :yodels? => true)
```

Фабричный метод `stub` нужен лишь для удобства – он возвращает Mock-объект с предопределенными заглушенными методами, что наглядно видно из исходного текста (листинг 18.4).

Листинг 18.4. Файл rspec/lib/spec/mocks/spec_methods.rb, строка 22

```
def stub(name, stubs={})
  object_stub = mock(name)
  stubs.each { |key, value| object_stub.stub!(key).and_return(value) }
  object_stub
end
```

Частичные подделки и заглушки

Вы обратили внимание на метод `stub!` в листинге 18.4? Его можно использовать для вставки или замены метода в любом объекте, а не только в Mock-объектах. Эта техника называется частичным подделыванием или глушением.

Термином *парциал* (partial) в RSpec называют экземпляр существующего класса, в котором часть поведения подделана или заглушена. Хотя авторы RSpec в документации предостерегают от такой практики, для работы в контексте Rails эта возможность абсолютно необходима, особенно когда речь заходит о взаимодействиях, в которых участвуют методы `ActiveRecord create` и `find`.

Чтобы увидеть, как механизм подделывания и глушения RSpec применяется на практике, вернемся к спецификации модели Monkeycharger, но на этот раз обратимся к классу `Authorizer`. Он общается со шлюзом в платежную систему и определяет, как следует обрабатывать транзакции по кредитным картам.

Припомните, что в разделе «Mock-объекты в Rails» главы 17 мы уже касались вопроса о подделывании внешних служб, чтобы не посылать настоящей службе тестовые данные. В листинге 18.5 эта техника про-

демонстрирована в действии с помощью Mock-объектов и заглушек RSpec.

Листинг 18.5. Спецификация модели Authorizer в системе Monkeycharger

```
describe Authorizer, "processing a non-saved card" do

  before(:each) do
    @card = CreditCard.new(:name => 'Joe Van Dyk',
                           :number => '4111111111111111',
                           :year => 2009, :month => 9,
                           :cvv => '123')
  end

  it "should send authorization request to the gateway" do
    $gateway.should_receive(:authorize)
       .with(599, @card).and_return(successful_authorization)

    Authorizer::authorize!(:credit_card => @card, :amount => '5.99')
  end

  it "should return the transaction id it receives from the gateway" do
    $gateway.should_receive(:authorize)
       .with(599, @card).and_return(successful_authorization)

    Authorizer::authorize!(:credit_card => @card, :amount => '5.99')
       .should == successful_authorization.authorization
  end

  it "authorize! should raise AuthorizationError on failed authorize" do
    $gateway.should_receive(:authorize)
       .with(599, @card).and_return(unsuccessful_authorization)

    lambda {
      Authorizer::authorize!(:credit_card => @card, :amount => '5.99')
    }.should raise_error(AuthorizationError,
                         unsuccessful_authorization.message)
  end

  private

  def successful_authorization
    stub(Object, :success? => true, :authorization => '1234')
  end

  def unsuccessful_authorization
    stub(Object, :success? => false, :message => 'reason why it
failed')
  end
end
```

Прогон «спеков»

«Спеки» – это исполняемые документы. Блок каждого примера выполняется внутри отдельного объекта, чтобы гарантировать целостность последнего (с точки зрения переменных – экземпляра и т. д.).

Если прогнать «спеки» кредитных карт из листингов 18.1 и 18.2 с помощью команды `spec`, которая должна присутствовать в системе после установки RSpec, будет выведен результат, аналогичный `Test::Unit`, – знакомый, удобный... только не слишком информативный:

```
$ spec spec/models/credit_card_spec.rb
.........
Finished in 0.330223 seconds
9 examples, 0 failures
```

RSpec может выводить результаты прогона «спеков» в разных форматах. Традиционный формат с точками, аналогичный принятому в `Test::Unit`, называется *индикацией прогресса* и, как мы только что видели, подразумевается по умолчанию.

Прощай, Test::Unit

Если вы еще не поняли, скажу, что при работе с RSpec среда `Test::Unit` нам больше не нужен. Они преследуют одни и те же цели – специфицировать и верифицировать функции приложения. Любую библиотеку можно использовать для эволюционного проектирования приложения в согласии с принципами разработки, управляемой тестами (TDD).

Над `Test::Unit` надстроен проект test/spec, реализующий принципы разработки, управляемой поведением (BDD), но пока он далеко отстает от RSpec и, похоже, не слишком активно развивается.

Если при вызове команды `spec` задать флаг `-fs`, результат будет выведен совсем в другом, гораздо более интересном, формате, который называется *specdoc*. Он далеко превосходит все, на что способен «готовый» `Test::Unit`:

```
$ spec -fs spec/models/credit_card_spec.rb
A valid credit card
- should be valid

CreditCard
- should have a valid month
- should have a valid year
- date should not be in the past
- should have two words in the name
- should have two words in the last name if the name is three words
```

```
long
- should have one word in the first name if the name is three words
long

We only take Visa and MasterCard
- should not accept amex
- should not accept discover

Finished in 0.301157 seconds

9 examples, 0 failures
```

Симпатично, правда? Не удивлюсь, если, увидев это впервые, вы задумаетесь: а не может ли RSpec помочь управиться с садистскими требованиями к документации, идущими от PHB.

Можно также оформить вывод в стиле системы RDoc для Ruby:

```
$ spec -fr spec/models/authorization_spec.rb
# Authorizer a non-saved card
# * the gateway should receive the authorization
# * authorize! should return the transaction id
# * authorize! should throw an exception on a unsuccessful
authorization

Finished in 0.268268 seconds

3 examples, 0 failures
```

И, быть может, самый красивый формат – в виде HTML с цветовым кодированием, который открывается в отдельном окне редактора TextMate, когда я прогоняю в нем «спеки».

На рис. 18.1 показан успешный прогон «спеков». Если бы какие-то примеры завершились с ошибкой, в соответствующих строках отобра-

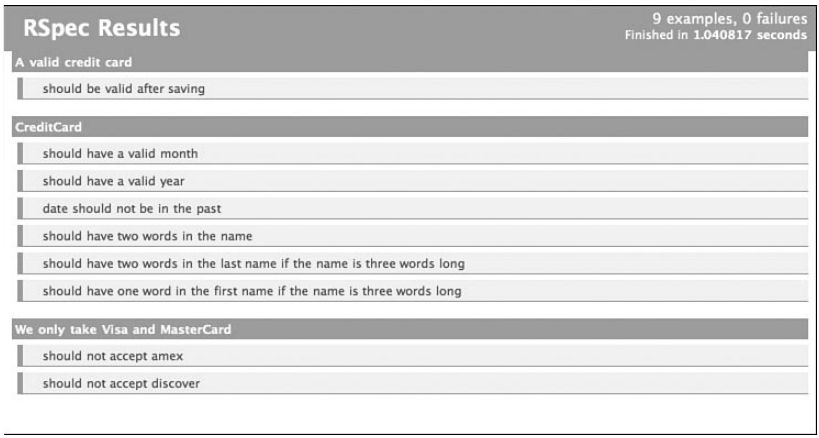

Рис. 18.1. *Результат работы RSpec, представленный в формате HTML*

жался бы красный индикатор. Наличие таких встроенных средств самодокументирования – одно из важнейших преимуществ, которые вы получаете, выбирая RSpec. Это даже побуждает разработчиков строить как можно более полное тестовое покрытие своих проектов. По собственному опыту знаю, что менеджерам тоже очень нравится вывод RSpec – они даже включают его в состав материалов, поставляемых с готовым продуктом.

Помимо различных видов форматирования, имеются еще и разнообразные флаги командной строки. Чтобы увидеть их все, наберите команду `spec --help`.

Установка RSpec и подключаемого модуля RSpec on Rails

Чтобы начать работать с RSpec on Rails, необходимо сначала установить библиотеку RSpec из gem-пакета. А затем установить в свой проект сам подключаемый модуль RSpec on Rails[1]:

```
sudo gem install rspec
script/plugin install
svn://rubyforge.org/var/svn/rspec/tags/CURRENT/rspec_on_rails
```

Авторитетные специалисты рекомендуют устанавливать RSpec как подключаемый модуль Rails, чтобы в разных проектах можно было использовать различные версии.

На этом мы завершаем введение в библиотеку RSpec и посмотрим, как она используется совместно с Ruby on Rails.

Подключаемый модуль RSpec on Rails

Подключаемый модуль RSpec on Rails предоставляет четыре контекста для «спеков», соответствующих четырем основным видам объектов, которые создаются в Rails. Помимо API, необходимого для написания «спеков» в Rails, он включает генераторы кода и ряд заданий Rake.

Генераторы

Предполагая, что подключаемый модуль уже установлен, вы должны запустить генератор `rspec` для подготовки проекта к работе с RSpec:

```
$ script/generate rspec
      create spec
      create spec/controllers
      create spec/fixtures
      create spec/helpers
```

[1] Если из-за схемы `svn://` возникнут проблемы с брандмауэром, следуйте инструкциям на странице *http://rspec.rubyforge.org/documentation/rails/install.html*.

```
      create spec/models
      create spec/views
      create spec/spec_helper.rb
      create spec/spec.opts
      create previous_failures.txt
      create script/spec_server
      create script/spec
```

В результате создастся каталог `spec`, содержащий по одному подкаталогу для каждого из четырех типов «спеков». Кроме того, сформируются различные дополнительные файлы, которые мы детально рассмотрим ниже.

Спецификации модели

Спецификации модели помогают специфицировать и верифицировать предметную модель приложения Rails – как классы `ActiveRecord`, так и ваши собственные классы. RSpec on Rails не предоставляет специальной функциональности для «спеков» модели, поскольку сверх того, что дает базовая библиотека, почти ничего и не нужно.

Имеется генератор `rspec_model`, который можно использовать вместо стандартного генератора `model`, включенного в Rails. Он работает почти так же, но создает заглушенный «спек» в каталоге `models` вместо заглушенного теста в каталоге `test`. Передайте ему имя класса (начинающееся с заглавной буквы) и пару `attribute_name:type`. В миграцию автоматически добавляются колонки `updated_at/created_at` типа `datetime`; указывать их явно необязательно:

```
$ script/generate rspec_model Schedule name:string
      exists app/models/
      exists spec/models/
      exists spec/fixtures/
      create app/models/schedule.rb
      create spec/fixtures/schedules.yml
      create spec/models/schedule_spec.rb
      exists db/migrate
      create db/migrate/001_create_schedules.rb
```

Сгенерированный класс `Schedule` пуст и не очень интересен. Заготовка `spec/models/schedule.rb` выглядит следующим образом:

```
require File.dirname(__FILE__) + '/../spec_helper'

describe Schedule do
  before(:each) do
    @schedule = Schedule.new
  end

  it "should be valid" do
    @schedule.should be_valid
  end
end
```

Предположим на секунду, что в классе Schedule имеется набор объектов, представляющих дни:

```
class Schedule < ActiveRecord::Base
  has_many :days
end
```

Специфицируем требование о наличии возможности получить накопительную сумму часов по объектам Schedule. Но фикстурами мы пользоваться не будем, а подделаем зависимость days:

```
require File.dirname(__FILE__) + '/../spec_helper'

describe Schedule do
  before(:each) do
    @schedule = Schedule.new
  end

  it "should calculate total hours" do
    days_proxy = mock('days')
    days_proxy.should_receive(:sum).with(:hours).and_return(40)
    @schedule.stub!(:days).and_return(days_proxy)
    @schedule.total_hours.should == 40
  end
end
```

Здесь мы воспользовались тем фактом, что прокси-объекты ассоциаций в Rails обладают собственными методами. ActiveRecord предоставляется несколько методов для выполнения агрегатных функций базы данных. Мы формулируем ожидание, согласно которому объект days_proxy должен отвечать на метод sum с одним аргументом (:hours) и возвращать число 40.

Этой спецификации удовлетворяет очень простая реализация:

```
class Schedule
  has_many :days

  def total_hours
    days.sum :hours
  end
end
```

Можно высказать справедливое критическое замечание относительно такого подхода – затруднен рефакторинг кода. Наш «спек» завершится с ошибкой, если не изменить реализацию метода total_hours, воспользовавшись методом Enumerable#inject, и это несмотря на то, что внешнее поведение не изменилось. Спецификации описывают не только видимое поведение объектов, но и взаимодействия между самим объектом и ассоциированными с ним. В таком случае подделка прокси-объекта ассоциации позволяет ясно выразить, как объект Schedule должен взаимодействовать с Days.

Существенное достоинство подделывания прокси-объекта `days` заключается в том, что мы больше не зависим от базы данных[1] при написании спецификации и реализации метода `total_hours`. Наши «спеки» будут работать очень быстро, и не придется иметь дело ни с какими фикстурами!

Авторитетные пропагандисты Mock-объектов считают их инструментом временного проектирования. Возможно, вы обратили внимание, что мы пока не определили класс `Day`. Вот и еще одно достоинство Mock-объектов – они позволяют специфицировать поведение изолированно, причем на этапе проектирования. Нет нужды отвлекаться от проектирования на создание класса `Day` и соответствующей ему таблицы в базе данных. Возможно, в таком простом примере это кажется не таким уж большим выигрышем, но при более сложной спецификации очень удобно иметь возможность целиком сосредоточиться на проектировании. Когда появится база данных и реальные модели объектов, можно будет вернуться и заменить Mock-объект `days_proxy` настоящим. Это неочевидная, но важная отличительная особенность Mock-объектов, которой часто пренебрегают.

Краткое знакомство с подделыванием моделей ActiveRecord

```
mock_model(model_class, stubs = {})
```

Метод `mock_model` создает Mock-объекты с автоматически генерируемыми числовыми идентификаторами и рядом заглушенных стандартных методов:

- `id` – возвращает автоматически генерируемое значение `id`;
- `to_param` – возвращает `id` в виде строки;
- `new_record?` – возвращает `false`;
- `errors` – возвращает заглушку для набора `errors`, которая сообщает, что в наборе 0 ошибок;
- `is_a?` – возвращает `true`, если параметр является объектом класса `model_class`;
- `class` – возвращает `model_class`.

Дополнительные заглушенные методы можно передать в хеше `stubs` или установить в блоке, которому передается экземпляр Mock-объекта.

[1] На самом деле это не совсем так. ActiveRecord все равно обращается к базе данных, чтобы получить информацию о колонках для объекта Schedule. Однако и это обращение можно заглушить и тем самым устранить все зависимости от базы данных.

Спецификации контроллеров

RSpec позволяет специфицировать контроллеры как изолированно от связанных с ними представлений, так и совместно (подобно обычным тестам Rails). Вот выдержка из документации по API:

> *В «спеках» контроллера используется класс* `Spec::Rails::DSL::ControllerBehaviour`*, который поддерживает прогон спецификаций в двух режимах, соответствующих детальному тестированию, обычному для TDD и более высокоуровневому, принятому в Rails. BDD занимает промежуточное положение: мы хотим добиться баланса между близостью «спеков» к коду (что позволяет быстро изолировать ошибки) и удаленностью от кода (чтобы можно было выполнять рефакторинг с минимальными изменениями существующих спецификаций).*

Класс `Controller` передается методу `describe` следующим образом:

```
describe MessagesController do
```

В необязательном втором параметре можно передать дополнительную информацию. А можно явно вызвать метод `controller_name` внутри блока `describe`, чтобы сообщить RSpec, какой контроллер использовать:

```
describe "Requesting /messages using GET" do
  controller_name :messages
fixtures :people
```

Я обычно группирую примеры для контроллера по действию и методу HTTP. При необходимости можно обращаться к фикстурам, равно как к любому другому тесту или «спеку». В этом примере требуется зарегистрировавшийся пользователь, поэтому для акцессора `current_person` из контроллера приложения я написал заглушку, которая возвращает фикстуру:

```
before(:each) do
  controller.stub!(:current_person, people(:quentin))
```

Далее я создаю Mock-объект `Message`, пользуясь методом `mock_model`. Я хочу, чтобы это подделанное сообщение возвращалось при каждом вызове метода `Message.find` во время прогона «спека»:

```
@message = mock_model(Message)
  Message.stub!(:find).and_return([@message])
end
```

Теперь можно приступить к специфицированию поведения действий (в данном случае – действия `index`). Основное ожидание состоит в том, что HTTP-код состояния в ответе должен быть равен 200:

```
it "should be successful" do
  get :index
  response.should be_success
end
```

Кроме того, я хочу специфицировать, что метод `find` объекта `Message` должен вызываться с правильными аргументами:

```
it "should find all the messages" do
  Message.should_receive(:find).with(:all).and_return [@message]
  get :index
end
```

Дополнительные ожидания, относящиеся к большинству действий контроллеров, касаются шаблона, который предстоит вывести, и присваивания значений переменным:

```
it "should render index.rhtml" do
  get :index
  response.should render_template(:index)
  end

it "should assign the found messages for the view" do
  get :index
  assigns[:messages].should include(@message)
end
```

Ранее мы видели, как заглушить прокси-ассоциацию модели. Хорошо было бы отказаться от использования фикстур в «спеках» контроллера. Заглушка метода `current_person` могла бы возвращать не фикстуру, а Mock-объект пользователя:

```
@mock_person = mock_model(Person, :name => "Quentin")
controller.stub!(:current_person).and_return @mock_person
```

Режимы изоляции и интеграции

По умолчанию «спеки» контроллеров в RSpec on Rails работают в *режиме изоляции*, то есть шаблоны представлений не задействуются. Достоинство этого режима в том, что специфицировать поведение контроллера можно абсолютно изолированно от представления, отсюда и название. Вдруг удастся спихнуть поддержку спецификации представлений кому-нибудь другому? (Специфицированию представлений целиком посвящен следующий раздел этой главы.)

Вообще-то, слово «спихнуть» я употребил в шутку. Разработка спецификаций представлений отдельно от контроллеров – не такое уж сложное дело, как иногда представляют. К тому же при этом обеспечивается гораздо лучшая изоляция ошибок, или по-простому – вам будет легче понять, почему программа не работает.

Если вы предпочитаете тестировать представления вместе с логикой контроллеров, описывая то и другое в одном наборе «спеков» контроллера, как в традиционных функциональных тестах Rails, то можно запускать RSpec on Rails в *режиме интеграции*, который устанавливается макросом `integrate_views`. Причем одно не исключает другого – можно задавать свой режим для каждого поведения:

```
describe "Requesting /messages using GET" do
  integrate_views
```

При работе в интегрированном режиме «спеки» контроллера исполняются один раз с включенным рендерингом представления.

Специфицирование ошибок

Обычно Rails перехватывает исключения, возникшие в ходе обработки действия, чтобы можно было вывести ответ с кодом 501 и знакомую вам страницу с распечаткой стека, значений переменных запроса и т. п. Если вы хотите явно специфицировать, что действие должно возбуждать исключение, то должны переопределить метод контроллера `rescue_action` примерно следующим образом:

```
controller.class.send(:define_method, :rescue_action) { |e| raise e }
```

Если вы согласны довольствоваться простой проверкой того, что ответ содержит код ошибки, можете воспользоваться предикатом `be_an_error` или акцессором `response_code` объекта `response`:

```
it "should return an error in the header" do
  response.should be_an_error
end

it "should return a 501" do
  response.response_code.should == 501
end
```

Специфицирование маршрутов

Маршрутизация – один из центральных компонентов Rails. Именно этот механизм позволяет Rails отобразить URL входящего запроса на контроллер и действие. Коль скоро он так важен, было бы неплохо специфицировать маршруты приложения. Это позволяет сделать метод `route_for` в «спеке» контроллера:

```
describe MessagesController, "routing" do
  it "should map { :controller => 'messages', :action => 'index' } to
  /messages" do
    route_for(:controller => "messages", :action => "index").should ==
  "/messages"
  end

  it "should map { :controller => 'messages', :action => 'edit',
  :id => 1 }
  to /messages/1;edit" do
    route_for(:controller => "messages", :action => "edit",
:id => 1).should == "/messages/1;edit"
  end
end
```

Спецификации представлений

Спецификации контроллеров позволяют интегрировать представление, чтобы убедиться в отсутствии ошибок в последнем, но мы можем поступить лучше – специфицировать сами представления. RSpec дает возможность писать спецификации представлений в полной изоляции от соответствующего контроллера. Можно оговорить существование определенных тегов и убедиться, что выведены правильные данные.

Пусть требуется написать страницу для вывода частных сообщений, которыми обмениваются участники интернет-форума. При запуске генератора `rspec_controller` RSpec создает каталог `spec/views/messages`. Сначала создадим в этом каталоге файл для представления `show` и назовем его `show_rhtml_spec.rb`. Затем подготовим информацию, отображаемую на странице:

```
describe "messages/show.rhtml" do
  before(:each) do
    @message = mock_model(Message, :subject => "RSpec rocks!")
    sender = mock_model(Person, :name => "Obie Fernandez")
    @message.stub!(:sender).and_return(sender)
    recipient = mock_model(Person, :name => „Pat Maddox“)
    @message.stub!(:recipient).and_return(recipient)
```

Если вы хотите добиться большей лаконичности, уложившись в одну длинную строку кода, которую придется разбить на несколько строчек, можете объединить предложения, в которых создаются Mock-объекты:

```
describe "messages/show.rhtml" do
  before(:each) do
    @message = mock_model(Message,
      :subject => "RSpec rocks!",
      :sender => mock_model(Person, :name => "Obie Fernandez"),
      :recipient => mock_model(Person, :name => „Pat Maddox“))
```

При любом варианте Mock-объекты создаются стандартным способом, который уже встречался нам прежде. Еще раз подчеркнем, что подделывание объектов, используемых в представлении, позволяет полностью изолировать эту спецификацию.

Присваивание значений переменным экземпляра

Теперь необходимо присвоить значение сообщению в представлении. Подключаемый модуль `rspec_on_rails` предоставляет знакомый метод `assigns`, с которым можно работать как с хешем:

```
    assigns[:message] = @message
  end
```

Фантастика! Все готово для специфицирования страницы представления. Мы хотели бы указать, что тема сообщения обернута тегом `<h1>`.

Ожидание have_tag принимает два аргумента – селектор тега и его внутреннее содержимое. Оно обертывает функциональность метода assert_select, являющегося частью стандартной подсистемы тестирования в Rails. Мы подробно рассматривали его в главе 17:

```
it "should display the message subject" do
  render "messages/show"
  response.should have_tag('h1', 'RSpec rocks!')
end
```

С HTML-тегами часто ассоциируется атрибут id. Мы хотели бы, чтобы на странице присутствовал тег с идентификатором message_info, в котором отображаются имена отправителя и получателя. Идентификатор id также можно передать методу have_tag:

```
it "should display a div with id message_info" do
  render "messages/show"
  response.should have_tag('div#message_info')
end
```

А что, если имена отправителя и получателя должны находиться внутри тега <h3>, погруженного в ?

```
it "should display sender and recipient names in div#message_info" do
  render "messages/show"
  response.should have_tag('div#message_info') do
    with_tag('h3#sender', 'Отправитель: Obie Fernandez')
    with_tag(,h3#recipient', ,Получатель: Pat Maddox')
  end
end
```

Заглушки для методов-помощников

Отметим, что методы-помощники не подмешиваются в «спеки» представлений автоматически, чтобы не нарушать изоляцию. Если код шаблона представления нуждается в помощниках, их придется подделать или заглушить в предоставляемом объекте template.

Выбор между Mock-объектом и заглушкой диктуется тем, играет ли метод-помощник активную роль в специфицируемом поведении:

```
it "should truncate subject lines" do
  template.should_receive(:truncate).exactly(2).times
  render "messages/index"
end
```

Если вы забудете подделать или заглушить вызов помощника, прогон спецификации завершится с ошибкой NoMethodError.

Спецификации помощников

Не составляет труда написать «спеки» для собственных модулей-помощников. Достаточно передать модуль методу describe, и он будет

подмешен в класс «спека», так что содержащиеся в нем методы станут доступны в коде примера:

```
describe ProfileHelper do
  it "profile_photo should return nil if user's photos is empty" do
    user = mock_model(User, :photos => [])
    profile_photo(user).should == nil
  end
end
```

Стоит отметить, что в отличие от спецификации представлений, в «спеки» помощников подмешиваются все модули `ActionView::Helper`, предоставляемые платформой, поэтому они доступны внутри кода вашего помощника. Добавляются также помощники всех динамически сгенерированных маршрутов.

Обстраивание

В состав Rails входит генератор `scaffold_resource`, позволяющий легко создавать REST-совместимые контроллеры и соответствующие модели. Подключаемый модуль `rspec_on_rails` предоставляет генератор `rspec_scaffold`, делающий то же самое с помощью RSpec, а не `Test::Unit`.

Поэкспериментируйте с `rspec_scaffold`, когда выдастся минутка; сгенерированные спецификации – еще один источник хороших примеров кода «спеков» для всех трех уровней паттерна MVC. Для сгенерированного Rails кода покрытие стопроцентное, так что есть чему поучиться.

Инструменты RSpec

Существует несколько проектов с открытыми исходными текстами, которые дополняют функциональность RSpec и могут повысить вашу продуктивность (ни один из этих инструментов не был написан специально для RSpec; первоначально все они разрабатывались для `Test::Unit`).

Autotest

Проект Autotest является частью комплекта инструментов ZenTest[1], созданного Райаном Дэвисом (Ryan Davis) и Эриком Ходелем (Eric Hodel). При каждом сохранении любого файла проекта Autotest прогоняет «спеки», на которые может повлиять изменение. Это отличный способ не выбиваться из ритма «красный-зеленый-рефакторинг», поскольку нет необходимости переходить из одного окна в другое, чтобы вручную прогнать тесты. Собственно, даже команд никаких запускать

[1] *http://rubyforge.org/projects/zentest/*.

не нужно! Достаточно один раз перейти в каталог проекта (`cd`) и набрать `autospec` – дальше все будет работать автоматически.

RCov

RCov – это инструмент определения тестового покрытия кода для Ruby[1]. Его можно «натравить» на файл «спека» и узнать, какая доля промышленного кода покрыта. На выходе генерируется HTML-документ, позволяющий легко понять, какой код покрыт «спеками», а какой – нет. Можно запускать RCov для отдельного «спека» или воспользоваться заданием `spec:rcov`, устанавливаемым вместе с подключаемым модулем `rspec_on_rails`, для прогона всех «спеков» под управлением RCov. Результаты выводятся в каталог `coverage` – набор HTML-файлов, который можно просматривать, открыв в броузере файл `index.html` (рис. 18.2).

Еще одним инструментом анализа покрытия кода является Heckle – часть впечатляющей коллекции проектов Seattle Ruby Brigade[2]. Heckle не просто проверяет покрытие тестами, но и помогает измерить эффективность спецификаций. Он «залезает» в код и начинает его всячески «корежить», например изменяет значения переменных и условия в предложениях `if`. Если ни один «спек» при этом не «ломается», значит, вы что-то упустили из виду.

В текущие версии RSpec поддержка Heckle уже встроена. Попробуйте задать флаг `--heckle` и посмотрите, что получится.

C0 code coverage information

Generated on Thu Sep 06 14:59:20 -0400 2007 with rcov 0.8.0

Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	630	391	89.5%	83.6%
app/controllers/application.rb	45	29	91.1%	86.2%
app/controllers/assets_controller.rb	105	62	94.3%	90.3%
app/controllers/session_controller.rb	30	25	100.0%	100.0%
app/controllers/user_assets_controller.rb	24	14	87.5%	78.6%
app/controllers/users_controller.rb	90	64	93.3%	90.6%
app/helpers/application_helper.rb	8	5	50.0%	40.0%
app/helpers/assets_helper.rb	2	2	100.0%	100.0%
app/helpers/user_assets_helper.rb	6	5	100.0%	100.0%
app/helpers/users_helper.rb	2	2	100.0%	100.0%
app/models/asset.rb	26	7	100.0%	100.0%
app/models/authenticated_base.rb	87	61	100.0%	100.0%
app/models/user.rb	44	17	81.8%	58.8%
lib/ar_extensions.rb	41	36	85.4%	83.3%
lib/authenticated_system.rb	120	62	75.8%	53.2%

Generated using the rcov code coverage analysis tool for Ruby version 0.8.0.

Рис. 18.2. *Пример отчета о покрытии, выданного RCov*

[1] *http://rubyforge.org/projects/rcov*.

[2] *http://rubyforge.org/projects/seattlerb/*.

Заключение

Вы получили представление о различных способах тестирования, поддерживаемых библиотекой RSpec. На первый взгляд может показаться, что это тот же `Test::Unit`, только какие-то слова подменены и все немножко переставлено местами. Но важно осознавать, что TDD – это методология *проектирования*, а не *тестирования*. Этот урок каждый приверженец TDD выучивает на многих практических примерах. Чтобы подчеркнуть данный аспект, в RSpec применяется иная терминология и другой стиль, что позволит вам сразу же начать получать дивиденды от использования TDD.

19

Расширение Rails с помощью подключаемых модулей

Повторю еще раз – когда речь идет о создании вещей людьми, раскрытие всегда происходит постепенно, чтобы получить удовольствие. Мы не можем заниматься процессом раскрытия, не зная, как доставить себе удовольствие.

Кристофер Александер

Хотя стандартные API Ruby on Rails очень богаты, рано или поздно наступает момент, когда вам требуется нечто, отсутствующее в ядре Rails или немного отличающееся от стандартного поведения. Тут-то и приходят на помощью подключаемые модули. В этой книге мы уже описали много полезных модулей, постоянно применяемых при создании приложений Rails.

А как насчет применения подключаемых модулей для обеспечения повторной используемости собственного кода? Может ли знакомство со способами их написания помочь в деле разработки более модульных приложений, да и принципов работы самой среды Rails? Безусловно!

В данной главе мы рассмотрим основные вопросы, относящиеся к управлению подключаемыми модулями, в том числе инструмент, считающийся в этом деле незаменимым: Piston. Кроме того, мы сообщим достаточно информации, чтобы вы могли приступить к написанию собственных подключаемых к Rails модулей.

Управление подключаемыми модулями

В Rails 1.0 появилась система подключения модулей, позволяющая разработчикам легко расширять функциональность среды. Официальный механизм позволяет выделить новые полезные возможности, открывшиеся при разработке приложений, и поделиться ими с другими разработчиками, оформив в виде автономных модулей, удобных для сопровождения и распространения.

Подключаемые модули полезны не только для обобществления новых возможностей. По мере взросления Rails им уделяется все больше внимания как инструменту тестирования изменений в самой среде. Почти любая значимая функциональность или заплата может быть реализована в виде подключаемого модуля и протестирована в «полевых условиях» многими разработчиками, прежде чем будет принято решение о включении ее в ядро. Если вы нашли в Rails ошибку и поняли, как ее исправить, или придумали новую интересную функцию, то, скорее всего, захотите оформить этот код в виде подключаемого модуля, чтобы его было проще распространять и тестировать.

Конечно, чтобы масштабно модифицировать поведение ядра среды, необходимо хорошо понимать внутреннее устройство Rails, а эта тема выходит за рамки настоящей книги. Однако некоторые демонстрируемые ниже приемы помогут вам разобраться в том, как реализован Rails. А это, в свою очередь, станет ценным подспорьем в тот день, когда вам потребуется изменить поведение ядра.

Повторное использование кода

Работа программиста требует умения абстрагировать постановку задачи. Мы решаем самые разные задачи, например поиск в базе данных, обновление онлайновых списков ожидающих дел или аутентификация пользователей. Результатом является набор решений конкретных задач, обычно в форме приложения.

Однако я сомневаюсь, что многие оставались бы программистами, если бы день ото дня приходилось решать одни и те же задачи. Нет, мы ищем способы применения уже найденных решений к новым задачам. Ваш код представляет собой абстрактное решение задачи, поэтому вы часто стремитесь либо повторно воспользоваться этой же абстракцией (пусть и в слегка отличающемся контексте), либо усовершенствовать решение так, чтобы его *можно было* повторно использовать. Повторное использование экономит деньги, время и силы и дает возможность сосредоточиться на новых интересных аспектах задачи, над которой вы сейчас работаете. В конце концов именно отыскание новых интересных решений и является залогом нашего успеха.

Сценарий plugin

Часто самым простым способом установки подключаемого модуля оказывается команда `script/plugin`. Запускать ее следует из корневого каталога разрабатываемого приложения. Прежде чем переходить к техническим деталям, взгляните на эту команду в действии:

```
$ cd /Users/obie/time_and_expenses
$ script/plugin install acts_as_taggable
+./acts_as_taggable/init.rb
+./acts_as_taggable/lib/README
+./acts_as_taggable/lib/acts_as_taggable.rb
+./acts_as_taggable/lib/tag.rb
+./acts_as_taggable/lib/tagging.rb
+./acts_as_taggable/test/acts_as_taggable_test.rb
```

Если после прогона зайти в каталог `vendor/plugins`, обнаружится, что в нем появился новый подкаталог `acts_as_taggable`.

Откуда берутся все эти файлы? Откуда `script/plugin` знает, куда помещать каталог `acts_as_taggable`? Чтобы понять, что происходит «под капотом», познакомимся с командами, составляющими сценарий `plugin`, поближе.

В следующих разделах мы рассмотрим каждую команду в отдельности.

script/plugin list

Поиск всех имеющихся подключаемых модулей осуществляется просто – с помощью команды `list`:

```
$ script/plugin list
account_location
http://dev.rubyonrails.com/svn/rails/plugins/account_location/
acts_as_taggable
http://dev.rubyonrails.com/svn/rails/plugins/acts_as_taggable/
browser_filters
http://dev.rubyonrails.com/svn/rails/plugins/browser_filters/
continuous_builder
http://dev.rubyonrails.com/svn/rails/plugins/continuous_builder/
deadlock_retry
http://dev.rubyonrails.com/svn/rails/plugins/deadlock_retry/
exception_notification
http://dev.rubyonrails.com/svn/rails/plugins/exception_notification/
localization
http://dev.rubyonrails.com/svn/rails/plugins/localization/
...
```

Эта команда возвращает список подключаемых модулей вместе с URL, по которому можно найти каждый модуль. Если внимательнее взглянуть на список URL, станет ясно, что группы подключаемых модулей часто помещаются в дерево с одним и тем же корнем, например *http://dev.rubyonrails.com/svn/rails/plugins*. Этот URL называется источни-

ком, и команда list при поиске подключаемых модулей пользуется набором источников.

Например, при запуске показанной выше команды script/plugin install acts_as_taggable она по очереди проверяет каждый источник в поисках того, который содержит каталог с указанным именем, в данном случае acts_as_taggable. Сценарий нашел такой каталог в источнике *http://dev.rubyonrails.com/svn/ rails/plugins* и загрузил его на локальную машину. В результате вы получили копию подключаемого модуля acts_as_taggable.

script/plugin sources

Ознакомиться со списком всех источников подключаемых модулей, которые просматривает Rails, позволяет команда sources:

```
$ script/plugin sources
http://dev.rubyonrails.com/svn/rails/plugins/
http://svn.techno-weenie.net/projects/plugins/
http://svn.protocool.com/rails/plugins/
http://svn.rails-engines.org/plugins/
http://lesscode.org/svn/rtomayko/rails/plugins/
...
```

Отметим, что в списке может быть как больше, так и меньше адресов; это вполне нормально, так что не пугайтесь. Список хранится в файле на локальной машине, его можно открыть в любом текстовом редакторе. В системах Mac OS X и Linux он находится в файле ~/.rails-plugin-sources.

script/plugin source [url [url2 [...]]]

Новый источник подключаемых модулей можно добавить вручную с помощью команды source:

```
$ script/plugin source http://www.our-server.com/plugins/
Added 1 repositories.
```

URL источника включает весь путь, кроме самого подключаемого модуля. Можете убедиться в этом, выполнив сразу после добавления команду script/plugin sources, – добавленный URL окажется в конце списка.

Если команда завершается неудачно, то, скорее всего, указанный URL уже есть в списке. Что и подводит нас к следующей команде – unsource.

script/plugin unsource [url[url2 [...]]]

Представьте, что вы добавили источник подключаемых модулей командой:

```
$ script/plugin source http:///www.our-server.com/plugins/
Added 1 repositories.
```

Из-за наличия трех знаков косой черты (///) между http и www этот URL работать не будет, поэтому необходимо удалить источник и ввести его правильно. Противоположная `source` команда `unsource` удаляет URL из списка активных источников подключаемых модулей:

```
$ script/plugin unsource http:///www.our-server.com/plugins/
Removed 1 repositories.
```

Можете убедиться, что источник удален, запустив еще раз команду `script/plugin sources`. Обе команды, `source` и `unsource`, принимают и сразу несколько URL; каждый из них будет добавлен в список (или удален из него).

script/plugin discover [url]

Команда `discover` проверяет наличие новых подключаемых модулей в Интернете и позволяет добавлять новые источники модулей. В действительности, эти источники обнаруживаются в результате разбора страницы Plugins на вики-сайте[1] Rails в поисках строки plugin в составе URL для протокола HTTP или Subversion. Легко видеть, что такому образцу соответствуют все возвращенные URL:

```
$ script/plugin discover
Add http://opensvn.csie.org/rails_file_column/plugins/? [Y/n] y
Add http://svn.protocool.com/rails/plugins/? [Y/n] y
Add svn://rubyforge.org//var/svn/laszlo-plugin/rails/plugins/? [Y/n] y
Add http://svn.hasmanythrough.com/public/plugins/? [Y/n] y
Add http://lesscode.org/svn/rtomayko/rails/plugins/? [Y/n] y
...
```

Вы можете указать другую страницу, которую должна разбирать команда `script/plugin discover`. Если в качестве аргумента задан URL, то `discover` отправится на соответствующую ему страницу, а не на вики-сайт Rails:

```
$ script/plugin discover http://internaldev.railsco.com/railsplugins
```

Это бывает полезно, например, когда вы сами поддерживаете список интересующих источников и хотите поделиться им со своими коллегами.

script/plugin install [plugin]

Мы уже видели, как работает эта команда, но в рукаве у `install` есть несколько трюков, которые могут оказаться очень кстати. Обычно вы задаете для этой команды всего один аргумент – имя устанавливаемого подключаемого модуля:

```
$ script/plugin install simply_restful
```

[1] *http://wiki.rubyonrails.org/rails/pages/Plugins*.

Выше мы видели, что модуль ищется в списке источников, которые вы добавили вручную или обнаружили с помощью команды `discover`. Но во многих случаях нужно проигнорировать список источников и установить модуль из известного URL, указав его в качестве аргумента:

```
$ script/plugin install http://www.pheonix.org/plugins/acts_as_macgyver
+./vendor/plugins/acts_as_macgyver/init.rb
+./vendor/plugins/acts_as_macgyver/lib/mac_gyver/chemistry.rb
+./vendor/plugins/acts_as_macgyver/lib/mac_gyver/swiss_army_knife.rb
+./vendor/plugins/acts_as_macgyver/assets/toothpick.jpg
+./vendor/plugins/acts_as_macgyver/assets/busted_up_bike_frame.html
+./vendor/plugins/acts_as_macgyver/assets/fire_extinguisher.css
```

Такой способ позволяет установить подключаемый модуль, не прибегая к поиску соответствия в списке источников. И, что самое важное, это экономит немало времени.

На этом таланты команды `install` не заканчиваются, однако более продвинутые возможности мы обсудим ниже в разделе «Система Subversion и сценарий script/plugin».

script/plugin remove [plugin]

Как и следовало ожидать, эта команда противоположна `install`: она удаляет подключаемый модуль из каталога `vendor/plugins`[1]:

```
$script/plugin -v remove acts_as_taggable
Removing 'vendor/plugins/acts_as_taggable'
```

Заглянув в каталог `vendor/plugins`, вы обнаружите, что папка `acts_as_taggable` действительно исчезла. При запуске команды `remove` попутно выполняется сценарий `uninstall.rb`, если таковой существует.

script/plugin update [plugin]

Интуитивно кажется очевидным, что запуск команды типа `$ script/plugin update acts_as_taggable` должен обновить текущую версию `acts_as_taggable`, если существует более поздняя. Однако на самом деле так происходит, только если вы пользовались одним из описанных в следующем разделе методов установки из системы Subversion. Если же подключаемый модуль был установлен с помощью простых манипуляций, описанных выше, то для его обновления на месте следует воспользоваться командой `install` с флагом `-f` (force):

```
$ script/plugin -f install my_plugin
```

Этот флаг приводит к принудительному удалению и повторной переустановке модуля.

---

[1] Флаг `-v` включает «говорливый» режим. Мы указали его, так как команда `remove` обычно ничего не печатает, поэтому без него было бы затруднительно показать, что происходят какие-то действия.

Система Subversion и сценарий script/plugin

Выше упоминалось, что большинство источников подключаемых модулей – репозитории системы управления версиями Subversion. Почему это важно для пользователей? Прежде всего потому, что для получения последних версий вы можете и не быть разработчиком, имеющим право обновлять репозиторий. Вы можете без труда (и даже автоматически) поддерживать свою копию модуля в актуальном состоянии, по мере того как автор добавляет новые функции, исправляет ошибки и вообще вносит любые изменения в центральное хранилище кода.

Но сначала нужно установить Subversion на локальные системы. Сам проект находится по адресу *http://subversion.tigris.org*, где имеется и ряд двоичных дистрибутивов прилагаемых к Subversion инструментов. Если вы работаете в ОС Linux или Mac OS X, то, скорее всего, система уже установлена, но пользователям Windows почти наверняка придется воспользоваться одним из имеющихся на сайте готовых инсталляторов.

Выгрузка подключаемого модуля

Если команда `install` запущена без флагов, она просто копирует файлы, входящие в состав подключаемого модуля, и помещает их в каталог, расположенный под `vendor/plugins`. Вам придется вручную загрузить все файлы в локальный репозиторий Subversion, при этом никаких упоминаний о месте, из которого они были взяты, не сохранится (разве что ваша собственная память и составленная автором документация). Это создает трудности, когда требуется обновить модуль, то есть получить версию с исправленными ошибками или новыми функциями.

Лучше воспользоваться системой Subversion и выгрузить из нее копию кода прямо в приложение, сохранив дополнительную информацию о текущей версии и источнике. На основе этой информации вы сможете автоматически обновлять подключаемый модуль до последней версии, хранящейся в репозитории.

Чтобы установить модуль методом выгрузки из Subversion, укажите при запуске команды `install` флаг `-o`:

```
$ script/plugin install -o white_list
A t_and_e/vendor/plugins/white_list/test
A t_and_e/vendor/plugins/white_list/test/white_list_test.rb
A t_and_e/vendor/plugins/white_list/Rakefile
A t_and_e/vendor/plugins/white_list/init.rb
A t_and_e/vendor/plugins/white_list/lib
A t_and_e/vendor/plugins/white_list/lib/white_list_helper.rb
A t_and_e/vendor/plugins/white_list/README
Checked out revision 2517.
```

В этом примере я выгрузил подключаемый модуль `white_list` в свой рабочий каталог, внутрь папки `plugins`, но он никак не связан ни с проектом, ни с системой управления версиями.

script/plugin update

Если модули устанавливались из системы Subversion, пригождается команда `update`. В этом случае сценарий `plugin` говорит Subversion (с помощью команды `svn`), что надо соединиться с репозиторием, из которого был взят модуль, и загрузить все изменения, доведя локальную копию до уровня последней версии. Как и в случае команды `install -o`, можно задать еще флаг `-r`, указав номер конкретной ревизии.

Внешние источники SVN

Хотя обращение к Subversion с помощью команды `install -o` в какой-то мере полезно, но при развертывании приложения вы можете столкнуться с трудностями. Напомню, что, если не считать хранения файлов в рабочем каталоге, подключаемый модуль никак не связан с вашим проектом. Следовательно, при развертывании придется вручную устанавливать все используемые в проекте подключаемые модули на конечный сервер. Это, как говорится, «не есть хорошо».

В действительности нам нужно как-то сообщить самому приложению, что для его запуска необходима версия X подключаемого модуля Y. Один из способов достичь желаемой цели – воспользоваться продвинутой функцией Subversion, а именно внешними источниками (externals).

Установив свойства `svn:externals` для папок приложения с контролируемыми исходными текстами, вы по сути дела объявляете Subversion: «Всякий раз при выгрузке или обновлении этого кода выгружай или обновляй подключаемый модуль, находящийся в указанном репозитории».

У команды `plugin install` имеется флаг `-x` специально для этой цели:

```
$ script/plugin install -x continuous_builder
A t_and_e/vendor/plugins/continuous_builder/tasks
A t_and_e/vendor/plugins/continuous_builder/tasks/test_build.rake
A t_and_e/vendor/plugins/continuous_builder/lib
A t_and_e/vendor/plugins/continuous_builder/lib/marshmallow.rb
A t_and_e/vendor/plugins/continuous_builder/lib/builder.rb
A t_and_e/vendor/plugins/builder/README.txt
Checked out revision 5651.
```

Команда `svn propget svn:externals` позволяет узнать, какие свойства были установлены для данного контролируемого каталога. Запустим ее для каталога `vendor/plugins` нашего приложения:

```
$ svn propget svn:externals vendor/plugins/ continuous_builder
http://dev.rubyonrails.com/svn/rails/plugins/continuous_builder
```

Поскольку подключаемый модуль `continuous_builder` был установлен с флагом `-x`, то при любой выгрузке приложения из репозитория (в том числе и на промышленный сервер) этот модуль тоже выгрузится автоматически. Однако такое решение не идеально, поскольку выгружена будет самая последняя головная ревизия модуля, работа которой с вашим приложением, возможно, не проверялась.

Фиксация конкретной версии

Как и в командах `install -o` и `update`, в команде `svn:externals` можно указать флаг `-r`, определяющий номер версии. При этом будет использоваться указанная версия подключаемого модуля, даже если его автор уже выложил новую.

Если вы задумаетесь, к какому хаосу может привести зависимость вашего приложения от новых, потенциально нестабильных версий, причем без всякого ведома с вашей стороны, то поймете, почему фиксация конкретных версий подключаемых модулей считается правильной практикой. Однако существует еще более простой способ управления зависимостями от подключаемых модулей.

Использование Piston

Утилита Piston (*http://piston.rubyforge.org/*) с открытыми исходными текстами позволяет управлять версиями библиотек в папке `vendor` проекта (подпапки Rails, Gems и Plugins) с меньшими затратами времени и более надежно, чем при прямой работе с Subversion.

Piston импортирует копии зависимых библиотек в собственный репозиторий, а не присоединяет их с помощью свойств `svn:externals`. Однако Piston хранит также метаданные, относящиеся к номеру версии зависимости, в виде свойств Subversion, ассоциированных с импортированным содержимым. Такое гибридное решение на практике работает очень неплохо.

Например, поскольку код подключаемого модуля становится частью вашего репозитория, вы можете вносить в него произвольные изменения (при использовании `svn:externals` локальные изменения невозможны). Когда вы решите перейти на более новую версию, в которой исправлены ошибки или добавлены новые функции, Piston автоматически включит в нее совместимые локальные изменения.

Установка

Piston распространяется в виде RubyGem-пакета. Для установки достаточно выполнить команду `gem install piston`:

```
$ sudo gem install –include-dependencies piston
Need to update 13 gems from http://gems.rubyforge.org
```

```
.............
complete
Successfully installed piston-1.2.1
```

После установки в системе появится новый исполняемый файл `piston`, поддерживающий следующие команды:

```
$ piston
Available commands are:
  convert    Converts existing svn:externals into Piston managed
folders
  help       Returns detailed help on a specific command
  import     Prepares a folder for merge tracking
  lock       Lock one or more folders to their current revision
  status     Determines the current status of each pistoned
directory
  unlock     Undoes the changes enabled by lock
  update     Updates all or specified folders to the latest revision
```

Импорт внешней библиотеки

Команда `import` означает, что Piston должен добавить в ваш проект внешнюю библиотеку. Например, воспользуемся Piston, чтобы посадить проект «на острие Rails». Это означает, что исполняемые файлы Rails будут браться из папки `vendor/rails`, а не из того места, куда был установлен RubyGem-пакет:

```
$ piston import http://dev.rubyonrails.org/svn/rails/trunk
vendor/rails
Exported r5731 from 'http://dev.rubyonrails.org/svn/rails/trunk' to
'vendor/rails'
```

По собственной воле Piston ничего не сохраняет в Subversion. Чтобы изменение стало постоянным, вы должны самостоятельно поставить его на учет:

```
$ svn commit -m "Импорт локальной копии Rails"
```

Кроме того, не забывайте, что в отличие от встроенного в Rails сценария `plugin` Piston принимает второй аргумент, указывающий, в какой каталог устанавливать библиотеку (если его опустить, по умолчанию она будет установлена в текущий каталог).

Вот, например, как из каталога `projects` устанавливается подключаемый модуль `white_list` Рика Олсона:

```
$ piston import
  http://svn.technoweenie.
net/projects/plugins/white_list/vendor/plugins/white_list
Exported r2562 from
'http://svn.techno-weenie.net/projects/plugins/white_list' to
'vendor/plugins/white_list'
```

Конвертация существующих внешних библиотек

Если вы уже успели воспользоваться свойствами `svn:externals` для связывания подключаемых модулей с исходным кодом проекта, следует конвертировать их в формат Piston, вызвав из каталога проекта команду `piston convert`:

```
$ piston convert
Importing 'http://macromates.com/svn/Bundles/trunk/Bundles/
Rails.tmbundle/Support/plugins/footnotes' to vendor/plugins/footnotes
(-r 6038)
Exported r6038 from 'http://macromates.com/svn/Bundles/trunk/Bundles/
Rails.tmbundle/Support/plugins/footnotes' to 'vendor/plugins/footnotes'
Importing 'http://dev.rubyonrails.com/svn/rails/plugins/
continuous_builder' to vendor/plugins/continuous_builder (-r 5280)
Exported r5280 from 'http://dev.rubyonrails.com/svn/rails/plugins/
continuous_builder' to 'vendor/plugins/continuous_builder'
Done converting existing svn:externals to Piston
```

Не забудьте, что после выполнения Piston необходимо поставить изменившиеся файлы проекта на учет в системе управления версиями.

Обновление

При желании получить из удаленного репозитория последние изменения библиотеки, установленной с помощью Piston, выполните команду `update`:

```
$ piston update vendor/plugins/white_list/
Processing 'vendor/plugins/white_list/'...
  Fetching remote repository's latest revision and UUID
  Restoring remote repository to known state at r2562
  Updating remote repository to r2384
  Processing adds/deletes
  Removing temporary files / folders
  Updating Piston properties
  Updated to r2384 (0 changes)
```

Блокировка и разблокировка

Можно предотвратить обновления в локальной папке, управляемой Piston, выполнив команду `piston lock`. А разблокировать ранее заблокированную папку позволяет команда `piston unlock`. Механизм блокировки служит дополнительной мерой предосторожности для коллективов разработчиков. Если вы знаете, что обновление подключаемого модуля приведет к неработоспособности приложения, можете заблокировать его, и тогда остальные члены команды увидят сообщение об ошибке, если попытаются обновить модуль, не выполнив предварительно разблокировку.

Свойства Piston

Если вы воспользуетесь командой `svn proplist` для просмотра свойств модуля `vendor/plugins/continuous_builder`, то увидите, что Piston хранит собственные свойства для папки каждого подключаемого модуля, а не для самой папки `plugins`:

```
$ svn proplist –verbose vendor/plugins/continuous_builder/

Properties on 'vendor/plugins/continuous_builder':

  piston:root :
http://dev.rubyonrails.com/svn/rails/plugins/continuous_builder

  piston:local-revision : 105
  piston:uuid : 5ecf4fe2-1ee6-0310-87b1-e25e094e27de
  piston:remote-revision : 5280
```

Написание собственных подключаемых модулей

В какой-то момент своей карьеры разработчика для Rails вы поймете, что имеет смысл вынести общий код, встречающийся в похожих проектах, над которыми вы работали; или, придумав какую-то интересную инновацию, решите поделиться ею со всем миром.

Rails легко позволяет вам стать автором подключаемого модуля. Активирован даже генератор подключаемых модулей, который создает базовую структуру каталогов и заготовки необходимых файлов:

```
$ script/generate plugin my_plugin
    create vendor/plugins/my_plugin/lib
    create vendor/plugins/my_plugin/tasks
    create vendor/plugins/my_plugin/test
    create vendor/plugins/my_plugin/README
    create vendor/plugins/my_plugin/MIT-LICENSE
    create vendor/plugins/my_plugin/Rakefile
    create vendor/plugins/my_plugin/init.rb
    create vendor/plugins/my_plugin/install.rb
    create vendor/plugins/my_plugin/uninstall.rb
    create vendor/plugins/my_plugin/lib/my_plugin.rb
    create vendor/plugins/my_plugin/tasks/my_plugin_tasks.rake
    create vendor/plugins/my_plugin/test/my_plugin_test.rb
```

Генератор предоставляет весь набор каталогов и начальных файлов, которые могут потребоваться для разработки подключаемого модуля, включая даже папку `/tasks` для относящихся к нему специальных заданий Rake. `install.rb` и `uninstall.rb` – необязательные файлы настройки и очистки, которые запускаются ровно один раз. Туда вы можете поместить произвольный код. Использовать все созданное генератором необязательно.

Для любого подключаемого модуля необходим файл `init.rb` и каталог `lib`. Если что-то из них не существует, Rails не будет считать подкаталог `vendor/plugins` подключаемым модулем. На самом деле многие популярные модули только и включают файл `init.rb` и несколько файлов в каталоге `lib`.

Точка расширения init.rb

Открыв заготовку файла `init.rb`, сгенерированную Rails, вы увидите простую инструкцию:

```
# insert hook code here (вставить сюда код точки расширения)
```

Под *кодом точки расширения* понимается код, подключаемый к процедурам инициализации Rails. Чтобы посмотреть, как это выглядит на практике, сгенерируйте подключаемый модуль в каком-нибудь из своих проектов и добавьте в файл `init.rb` такую строчку:

```
puts "Текущая версия Rails: #{Rails::VERSION::STRING}"
```

Поздравляю: вы только что написали свой первый простенький подключаемый модуль. Запустите консоль Rails, и вы поймете, что я имею в виду:

```
$ script/console
Loading development environment.
Текущая версия Rails: 1.2.3
>>
```

Код, находящийся в файле `init.rb`, выполняется на этапе инициализации (при любом способе инициализации Rails: в сервере, на консоли или с помощью сценария `script/runner`). Как правило, подключаемые модули помещают свои предложения `require` в файл `init.rb`.

Вашему коду в `init.rb` доступны некоторые специальные переменные, относящиеся к самому подключаемому модулю:

- `name` – имя вашего модуля (в этом простом примере `'my_plugin'`);
- `director` – каталог, в котором «живет» модуль; полезно, когда нужно читать или записывать нестандартные файлы, хранящиеся в каталоге подключаемого модуля;
- `loaded_plugins` – объект `Set`, содержащий имена уже загруженных модулей, включая и тот, что в данный момент инициализируется;
- `config` – конфигурационный объект, созданный сценарием `environment.rb` (см. главу 1 и онлайновую документацию по API `Rails::Configuration`, где приведена подробная информация о том, к чему можно получить доступ с помощью объекта `config`).

Наш примерчик, конечно, элементарен. Обычно с помощью подключаемого модуля вы хотите реализовать новую функциональность для своего приложения или модифицировать библиотеки Rails каким-нибудь более интересным способом, чем простая печать номера версии.

Каталог lib

Каталог `lib` подключаемого модуля добавляется в путь загрузки Ruby еще до выполнения сценария `init.rb`. Это означает, что вы можете затребовать свой код с помощью `require`, не задавая путь загрузки самостоятельно:

```
require File.dirname(__FILE__) + '/lib/my_plugin' # лишнее
```

Предполагая, что в каталоге `lib` есть файл `my_plugin.rb`, в `init.rb` достаточно просто написать:

```
require 'my_plugin'
```

Ничего сложного. Вы можете поместить в каталог `lib` любой класс или код на Ruby, а затем загрузить его в `init.rb` (или разрешить другим разработчикам подгружать его в `environment.rb`) с помощью предложения `require`. Это самый простой способ добиться обобществления кода между несколькими приложениями Rails.

Для подключаемых модулей типично изменять или дополнять поведение существующих классов Ruby. В качестве примера в листинге 19.1 приведен исходный текст модуля, который наделяет классы ActiveRecord итератором, аналогичным курсору (обратите внимание, что более сложная реализация этого приема могла бы включать транзакции, обработку ошибок и пакетные средства; дополнительную информацию по этому поводу см. на странице *http://weblog.jamisbuck.org/2007/4/6/faking-cursors-in-activerecord*).

Листинг 19.1. Добавление итератора Each в классы ActiveRecord

```
# в файле vendor/plugins/my_plugin/my_plugin.rb

class ActiveRecord::Base

  def self.each
    ids = connection.select_values("select id from #{table_name}")
    ids.each do |id|
      yield find(id)
    end
    ids.size
  end
end
```

Помимо открытия существующих классов с целью добавления или модификации поведения есть еще по меньшей мере три способа, которыми подключаемые модули могут расширять функциональность Rails:

- подмешивание модулей в существующие классы;
- динамическое расширение с помощью обратных вызовов и точек расширения Ruby, в частности `method_missing`, `const_missing` и `included`;

- динамическое расширение путем интерпретации кода на этапе выполнения при помощи таких предложений, как `eval`, `class_eval` и `instance_eval`.

Расширение классов Rails

Способ, которым мы открыли класс `ActiveRecord::Base` в листинге 19.1 и добавили в него новый метод, прост, но в большинстве подключаемых модулей используется техника, применяемая внутри самой среды Rails. Код разносится по двум модулям: один для методов класса, другой для методов экземпляра. Давайте добавим полезный метод экземпляра `to_param` во все объекты ActiveRecord[1].

Переделаем модуль `my_plugin` в этом стиле. Сразу после предложения `require 'my_plugin'` в файле `init.rb` мы пошлем сообщение `include` самому классу `ActiveRecord`:

```
ActiveRecord::Base.send(:include, MyPlugin)
```

Это еще один способ добиться того же самого результата – вы можете встретить его в исходных текстах популярных подключаемых модулей[2]:

```
ActiveRecord::Base.class_eval do
  include MyPlugin
end
```

Теперь необходимо написать Ruby-модуль `MyPlugin`, где будут находиться переменные класса и экземпляра, которыми мы расширим класс `ActiveRecord::Base` (листинг 19.2).

Листинг 19.2. Расширение класса ActiveRecord::Base

```
module MyPlugin
  def self.included(base)
    base.extend(ClassMethods)
    base.send(:include, InstanceMethods)
  end

module ClassMethods
  def each
    ids = connection.select_values("select id from #{table_name}")
    ids.each do |id|
      yield find(id)
    end
    ids.size
```

[1] На странице *http://www.jroller.com/obie/entry/seo_optimization_of_urls_in* вы найдете рассказ о том, как изобретательное использование метода `to_param` может помочь в оптимизации открытого сайта для поисковых машин.

[2] Джей Филдс поместил в своем блоге интересную заметку о мотивах, стоящих за различными способами расширения кода. См. *http://blog.jayfields.com/2007/01/class-reopening-hints.html*.

```
    end
  end

  module InstanceMethods
    def to_param
      has_name? ? "#{id}-#{name.gsub(/[^a-z0-9]+/i, '-')}" : super
    end

    private

      def has_name?
        respond_to?(:name) and not new_record?
      end

    end
  end
```

Аналогичная техника применима для расширения контроллеров и представлений[1]. Например, если вы хотите добавить методы-помощники, доступные во всех шаблонах представлений, то можете расширить класс `ActionView` следующим образом:

```
ActionView::Base.send(:include, MyPlugin::MySpecialHelper)
```

Познакомившись с основами написания подключаемых модулей Rails (файл `init.rb` и содержимое каталога `lib`), мы можем взглянуть на другие файлы, созданные генератором.

Файлы README и MIT-LICENSE

Первым делом разработчик, знакомящийся с новым подключаемым модулем, открывает файл `README`. Возникает сильное искушение этот файл проигнорировать, но вы должны по меньшей мере поместить в него краткое описание того, что делает модуль. Файл `README` также читает и обрабатывает включенный в Ruby инструмент RDoc, когда генерирует документацию по подключаемому модулю с помощью Rake-заданий `doc:`. Имеет смысл изучить основы форматирования в RDoc, если вы хотите, чтобы информация, помещенная вами в файл `README`, позднее предстала в эстетически привлекательном виде.

Rails, как и большинство популярных подключаемых модулей, поставляется с открытыми исходными текстами на условиях весьма либеральной лицензии MIT. В знаменательном обращении к участникам конференции Railsconf 2007 Дэвид объявил, что генератор подключаемых модулей будет автоматически создавать файл с лицензией MIT, чтобы помочь в решении проблемы модулей, распространяемых без

[1] В статье Алекса Янга по адресу *http://alexyoung.org/articles/show/40/a_taxonomy_of_rails_plugins* рассматриваются различные виды подключаемых модулей Rails, в том числе и полезное объяснение того, как обрабатывать дополнительные конфигурационные параметры на этапе выполнения.

лицензии, на использование открытых исходных текстов. Разумеется, вы можете изменить лицензию, как сочтете нужным, но лицензия MIT считается частью *Пути Rails*.

Файлы install.rb и uninstall.rb

Эти два файла помещаются в корень каталога подключаемого модуля вместе с `init.rb` и `README`. Если `init.rb` можно использовать для выполнения некоторых действий при каждом запуске сервера, то эти файлы позволяют подготовить все необходимые вашему модулю условия при установке командой `script/plugin install` и произвести очистку в момент деинсталляции командой `script/plugin remove`.

Установка

Предположим, что вы разработали подключаемый модуль, который генерирует промежуточные данные, сохраняемые во временных файлах приложения. Чтобы модуль мог работать, необходимо предварительно создать каталог для временных файлов. Вот отличная возможность применить сценарий `install.rb` (листинг 19.3).

Листинг 19.3. Создание каталога для временных файлов на этапе установки подключаемого модуля

```
require 'fileutils'
FileUtils.mkdir_p File.join(RAILS_ROOT, 'tmp', 'my_plugin_data')
```

В результате добавления этих строчек в файл `install.rb` в любом приложении Rails, для которого установлен данный подключаемый модуль, создается каталог `tmp/my_plugin_data`. Это одноразовое действие можно применять для различных целей, в том числе:

- копирования ресурсных файлов (HTML, CSS и т. д.) в каталог `public`;
- проверки существования зависимостей (например, библиотеки RMagick);
- установки других необходимых подключаемых модулей (листинг 19.4).

Листинг 19.4. Установка необходимого подключаемого модуля

```
# Установить подключаемый модуль engines, если его еще нет
unless File.exist?(File.dirname(__FILE__) + "/../engines")
  Commands::Plugin.parse!(['install',
    'http://svn.rails-engines.org/plugins/engines'])
end
```

В листинге 19.4 показано, как, проявив творческий подход и немного покопавшись в исходных текстах Rails, можно отыскать и повторно воспользоваться такой функциональностью, как директива `parse!` из класса `Commands::Plugin`.

Удаление

Как уже упоминалось, команда `script/plugin remove` проверяет наличие файла `uninstall.rb` в момент удаления подключаемого модуля. Если файл присутствует, он будет выполнен перед фактическим удалением. Обычно эта возможность применяется для выполнения действий, противоположных выполненным при установке, например для удаления каталогов или файлов данных, которые модуль мог создать на этапе установки или во время работы приложения.

О важности здравого смысла

Хотя поначалу это и неочевидно, описанная схема не является стопроцентно надежной. Пользователи подключаемых модулей часто неосознанно пропускают процедуры установки. Поскольку подключаемые модули почти всегда распространяются с помощью системы Subversion, очень легко добавить модуль в проект, выполнив простую команду выгрузки:

```
$ svn co http://plugins.com/svn/whoops vendor/plugins/whoops # без установки
```

А вот еще более простое развитие событий – модуль копируется из одного проекта в другой с помощью средств файловой системы. Я знаю – сам много раз так делал. То же самое относится и к удалению – разработчик, не знающий о существовании других возможностей, может просто удалить папку подключаемого модуля из каталога `vendor/plugins`, и тогда сценарий `uninstall.rb`, конечно, не отработает.

Если вы как автор подключаемого модуля хотите быть уверены, что сценарии установки и удаления таки будут выполнены, имеет смысл громогласно объявить об этом сообществу и написать большими буквами в прилагаемой документации, скажем, в файле `README`.

Специальные задания Rake

Часто в состав подключаемых модулей включаются задания Rake. Например, если модуль сохраняет файлы во временном каталоге (скажем, `/tmp`), можно включить вспомогательное задание для стирания временных файлов, чтобы пользователю не приходилось копаться в коде, пытаясь понять, где они находятся. Подобные задания следует определять в файле `.rake` в папке `tasks` вашего модуля (листинг 19.5).

Листинг 19.5. Задание Rake для очистки временного каталога подключаемого модуля

```
# vendor/plugins/my_plugin/tasks/my_plugin.rake

namespace :my_plugin do

  desc 'Удаление временных файлов'
```

```
  task :cleanup => :environment do
    Dir[File.join(RAILS_ROOT, 'tmp', 'my_plugin_data')].each do |f|
      FileUtils.rm(f)
    end
  end
end
```

Задания Rake, добавленные подключаемыми модулями, перечисляются вместе со стандартными заданиями Rails при запуске команды `rake -T`, которая выводит список всех заданий в проекте (в следующем фрагменте я ограничил размер выдачи, передав в качестве аргумента строку, с которой должны сопоставляться найденные имена заданий):

```
$ rake -T my_plugin
rake my_plugin:cleanup # Удаление временных файлов
```

Rakefile подключаемого модуля

У сгенерированных подключаемых модулей имеется собственный небольшой Rakefile, которым можно пользоваться внутри каталога модуля для прогона тестов и генерирования документации в формате RDoc (листинг 19.6).

Листинг 19.6. Сгенерированный Rakefile подключаемого модуля

```
require 'rake'
require 'rake/testtask'
require 'rake/rdoctask'

desc 'Default: прогон автономных тестов.'
task :default => :test

desc 'Тест подключаемого модуля my_plugin.'
Rake::TestTask.new(:test) do |t|
  t.libs << 'lib'
  t.pattern = 'test/**/*_test.rb'
  t.verbose = true
end

desc 'Генерирование документации для подключаемого модуля my_plugin.'
Rake::RDocTask.new(:rdoc) do |rdoc|
  rdoc.rdoc_dir = 'rdoc'
  rdoc.title = 'MyPlugin'
  rdoc.options << '--line-numbers' << '--inline-source'
  rdoc.rdoc_files.include('README')
  rdoc.rdoc_files.include('lib/**/*.rb')
end
```

Раз уж об этом зашла речь, упомяну, что в Rails имеются собственные принимаемые по умолчанию задания Rake, относящиеся к подключаемым модулям, которые практически не нуждаются в пояснениях:

```
$ rake -T plugin

rake doc:clobber_plugins    # Удалить документацию по подключаемому модулю
rake doc:plugins            # Сгенерировать документацию для установленных
                            # подключаемых модулей
rake test:plugins           # Прогнать тесты подключаемых модулей во всех
                            # каталогах vendor/plugins/*/**/test
                            # (или задайте с помощью PLUGIN=name)
```

Прежде чем завершить это раздел, хотелось бы отметить различие между Rakefile подключаемого модуля и файлами .rake в папке tasks:

- используйте Rakefile для заданий, работающих с исходными файлами модуля, например с целью проведения специального тестирования или генерирования документации. Они должны запускаться из самого каталога подключаемого модуля;
- используйте файлы tasks/*.rake для заданий, являющихся частью процесса разработки или развертывания приложения, в которое установлен модуль. Эти задания будут показываться командой `rake -T`, которая выводит список всех заданий Rake для данного приложения.

Тестирование подключаемых модулей

И последнее. Написав подключаемый модуль, вы должны приложить к нему тесты, верифицирующие поведение. Создание тестов для подключаемых модулей мало чем отличается от любых других видов тестирования в Rails или Ruby, и методы применяются в основном те же самые. Однако подключаемые модули зачастую не могут предсказать, в каком окружении им суждено работать, поэтому необходимы дополнительные меры предосторожности, направленные на изолирование поведения модуля от остальной части приложения в ходе тестирования.

Существует тонкое различие между прогоном тестов модуля с помощью глобального Rake-задания `test:plugins` и собственного модуля `Rakefile`. Хотя первое может протестировать сразу все установленные модули, внутренний `Rakefile` можно и до лжно применять для добавления специальных заданий, необходимых для надлежащего тестирования подключаемого модуля.

Среди приемов, применяемых для тестирования подключаемых модулей, стоит отметить создание отдельной базы данных, что позволяет полностью изолировать модуль от приложения. Это особенно полезно, когда модуль расширяет функциональность `ActiveRecord`, поскольку новые методы желательно тестировать в контролируемом окружении, сведя к минимуму взаимодействие с другими модулями и тестовыми данными самого приложения.

Вы, конечно, понимаете, что тестирование подключаемых модулей – обширная тема, представляющая интерес главным образом для их ав-

торов. К сожалению, я вынужден оставить этот разговор за рамками настоящей книги из практических соображений, чтобы не увеличивать ее объем до бесконечности.

Заключение

Вы познакомились со всеми основными аспектами подключаемых модулей в Rails. Теперь вы знаете, как их устанавливать и как использовать инструмент Piston для управления версиями модулей. Мы также немного поговорили о написании собственных подключаемых модулей – достаточно, чтобы вы могли приступить к самостоятельной работе.

Для рассмотрения всех вопросов, относящихся к подключаемым модулям в Rails, потребовалась бы отдельная книга, и эта тема выходит далеко за рамки потребностей большинства разработчиков. Мы почти не касались тестирования подключаемых модулей и продвинутых приемов, применяемых их авторами. Мы также не обсудили жизненный цикл модулей по завершении начальной разработки.

Тем, кто хочет глубже изучить тему расширения Rails с помощью подключаемых модулей, я настоятельно рекомендую книгу Джеймса Адама (James Adam) *Rails Plugins*, вышедшую в издательстве Addison-Wesley. Он считается самым авторитетным в мире экспертом по этому вопросу.

20

Конфигурации Rails в режиме эксплуатации

Люди, собравшиеся вокруг костра или свечки в поисках тепла или света, менее склонны к независимым мыслям или даже деяниям, чем те, у кого есть электрическое освещение. Точно так же в автоматизации скрыты социальные и образовательные паттерны – индивидуальная трудовая деятельность и профессиональная независимость.

Маршалл Мак-Лухан (Marshall McLuhan)

Хотите верьте, хотите нет, но при создании приложений Rails часто упускают из виду один аспект – развертывание и работу в режиме промышленной эксплуатации. Иногда вы не отвечаете за эту часть проекта, но в любом случае важно понимать, как современное веб-приложение работает на промышленном сервере. В этой главе мы покажем, как построить простую промышленную систему и запустить на ней приложение Rails. Мы не будем выходить за рамки типичной базовой конфигурации, чтобы вам было легче уяснить основные компоненты и оптимальные методики. Мы также кратко рассмотрим некоторым распространенные проблемы, для решения которых требуется более сложная конфигурация. Это будет полезно, если вы впервые занимаетесь промышленным развертыванием системы.

Даже если раньше вам уже приходилось запускать приложение Rails в промышленном режиме, все равно имеет смысл прочитать эту главу, так как в ней обсуждаются некоторые простые способы автоматизировать работу системы, не усложняя конфигурацию.

Краткая история промышленной эксплуатации Rails

К счастью, теперь развертывание Rails в промышленном окружении стало относительно простой задачей, но так было не всегда. Многие экономящие время принципы проектирования и наработанные методики, внутренне присущие Rails, в том числе такие хорошо известные, как «примат соглашения над конфигурацией» и «не повторяйся», перешли в практику конфигурирования и развертывания Rails в режиме эксплуатации. По-другому ту же мысль можно выразить, сказав, что автоматизация – ключ ко всему. Rails позволяет разработчику сконцентрироваться на поведении, специфическом для своего приложения, и автоматизировать все остальное. Следуя этой традиции, различные инструменты – Mongrel, Mongrel Cluster и Capistrano – стремятся упростить и автоматизировать утомительные части запуска приложения в промышленном окружении. В них уже учтены многие прошлые уроки.

Когда среда Rails только появилась, у разработчиков были ограниченные возможности для его запуска на реальном веб-сервере типа Apache: CGI, модуль Apache `mod_ruby` или FastCGI (FCGI). И у всех были свои недостатки. Находившиеся в каталоге `script` сценарии типа `spawner` и `reaper` были призваны сгладить проблемы, свойственные FastCGI.

В 2006 году Зед Шоу (Zed Shaw) написал HTTP-сервер промышленного качества Mongrel[1] (в основном, на языке Ruby). Mongrel с самого начала проектировался, чтобы заменить все прочие варианты, оставаясь при этом небольшой и простой программой. Он не заставляет обращенный наружу сервер, например Apache, преобразовывать входящие HTTP-запросы в вызовы CGI только для загрузки окружения Ruby, а является Ruby-процессом, который напрямую общается с Apache по протоколу HTTP. При этом Mongrel убирает две промежуточные стадии (преобразование запроса из HTTP в CGI и загрузку Ruby), сокращает количество «движущихся деталей», повышает надежность и предсказуемость промышленного окружения, а также ведет к росту производительности. Кроме того, это быстрый веб-сервер, пригодный для локальной разработки.

---

[1] Домашняя страница проекта Mongrel находится по адресу *http://mongrel.rubyforge.org*.

Предварительные условия

Для подготовки промышленной системы необходимы условия, перечисленные ниже.

- *Умение работать в Unix.*
- Абсолютное большинство промышленных приложений Rails работает под управлением того или иного варианта Unix (FreeBSD, OSX, Solaris, любой дистрибутив Linux и т. д.). В отсутствие серьезных препятствий (например, вы работаете в крупной компании и пытаетесь протащить Rails через черный вход) тоже следует ориентировать приложение на Unix. Мы предполагаем, что вы знаете, как работать с командной строкой, умеете выполнять основные команды, устанавливать пакеты, запускать и останавливать службы и т. д. Все примеры в этой главе написаны для оболочки Bash[1].
- *Доступ к серверу с правом выполнения sudo.*

  Мы будем устанавливать все на один сервер, поэтому вам понадобится доступ к нему с возможностью устанавливать программы и выполнять развертывание. Конкретные приложения и виды доступа мы перечислим в разделе «Контрольный перечень», а также расскажем о возможных вариантах хостинга на случай, если вы не сможете найти подходящий с помощью поисковой машины.
- *Уважение к промышленному окружению и сильное желание учиться.*

  Будем надеяться, что вашему приложению Rails предстоит долгая и счастливая жизнь. Даже если вы обычно не отвечаете за промышленную эксплуатацию приложения, все равно необходимо понимать, насколько критична эта часть его жизненного цикла. Не надо думать, что настройка и сопровождение промышленного окружения – черная работа. Может, это и не так приятно и интересно, как писать приложение, но уж точно не менее важно. Как и в любой другой технической отрасли (автомобилестроении, судостроении и т. п.), после выхода приложения возникают совершенно новые проблемы. Особенно важно это для современных веб-приложений, которые часто изменяются в ходе повторяющихся циклов разработки. Опыт, накопленный в процессе эксплуатации, учитывается на следующем цикле разработки, и так до бесконечности.

  Если вам доводилось прежде отвечать за работу или сопровождение промышленного приложения, то вы понимаете, сколько на этом этапе выявляется досадных ошибок и что громоздкие, плохо организованные протоколы и конфигурационные файлы – последнее,

[1] Информацию о Bash см. на странице *http://en.wikipedia.org/wiki/Bash*.

что вы хотели бы видеть (когда приходится вскакивать в четыре утра, чтобы устранить аварию). Признавая важность содержания промышленной среды в порядке, вы устраните немало проблем в будущем. Многие советы, приводимые в этой главе (а также в рекомендуемых нами инструментах и библиотеках), родились из попыток избежать таких неприятностей.

- *Готовность порвать со старыми привычками и обзавестись новыми.*

  Одна из ключевых концепций промышленной эксплуатации – *автоматизация*. Автоматизируйте все и вся! Сценарии развертывания и конфигурирования должны делать за вас всю работу. Если вы любите что-то изменять в файлах вручную или модифицировать систему, уже развернутую на сервере, придется порвать с этой привычкой. Такое поведение (а мы наблюдали его и у разработчиков, и у системных администраторов) приведет только к одному – промышленная система станет хрупкой, непредсказуемой, и ее постигнет печальная судьба.

  Если имеется возможность работать в высокоавтоматизированной промышленной системе, вы должны всемерно пользоваться ею и устранить любые лазейки для ручного вмешательства. Возможно, вам кажется, что будет быстрее внести мелкое изменение, отредактировав какой-то файл вручную на «живом» сервере, но это гибельная склонность. Такие изменения трудно отслеживать, а автоматизированные системы плохо реагируют на изменения, внесенные вручную.

- *Готовность на время отказаться от неверия.*

  В какой-то мере это следствие предыдущего пункта. Иногда разработчики и системные администраторы любят усложнять, особенно если считают, что есть более правильный путь. Если вы из таких людей (пусть даже не готовы это признать) и, читая эту главу, решите, что наш подход никуда не годится, он слишком все упрощает и т. д. и т. п., расслабьтесь. Большинство простых приложений Rails работает как раз в таком окружении, и рекомендуемые нами пакеты активно применяются на профессиональных хостингах, ориентированных на Rails. Например, кому-то не нравится сама мысль об использовании какого-либо другого веб-сервера, кроме Apache. Обычно это объясняется тем, что человек эксплуатирует его уже много лет и ничего другого не знает. Не бойтесь сервера Nginx – не исключено, что вы даже полюбите его, поняв, насколько он прост и быстро работает (к тому же без ошибок).

Контрольный перечень

Прежде чем приступать к детальному обсуждению необходимого программного обеспечения, сформулируем общие предположения о конфигурации, которую собираемся построить. Уже отмечалось, что орга-

низовать промышленную систему можно разными способами, но мы остановимся на простой конфигурации, проверенной на практике, которая хорошо зарекомендовала себя во многих приложениях Rails.

В следующих разделах описываются ключевые компоненты нашего варианта промышленной системы. Обратите внимание, что помимо стандартных ярусов: веб-сервер, сервер приложений и база данных, мы выделяем еще две критически важные части, которые обычно остаются без внимания: серверное и сетевое окружение и компоненты для мониторинга.

Серверное и сетевое окружение

Разумеется, приложение должно где-то работать...

Автономный/выделенный сервер или часть виртуального частного сервера

Поскольку мы собираемся запускать все на одном сервере, понадобится доступ либо к выделенному серверу, либо к части виртуального частного сервера (VPS slice). Некоторые компании, предоставляющие VPS-хостинг (например, Rails Machine, EngineYard, Slicehost), уже настроены для размещения приложений Rails и готовы сделать за вас всю грязную работу. Но мы начнем с «голой» машины и будем ее постепенно обустраивать. В конечном итоге получится примерно такая же конфигурация.

Новая установка ОС

Должен работать любой из популярных дистрибутивов Linux. Мы предпочитаем Debian, Gentoo, CentOS или RedHat. У кого-то есть личное пристрастие к другим дистрибутивам, но именно с этими обычно работают хостинговые компании.

Вы можете установить требуемые продукты с помощью своего любимого менеджера пакетов или собрать их из исходных текстов. В примерах ниже мы будем как компилировать «исходники», так и устанавливать готовые двоичные дистрибутивы (например, MySQL) в зависимости от того, что проще.

В начале главы мы говорили, что вам понадобится пользователь с правом выполнения sudo (или просто root). Можно запустить приложение Rails и без этого, но тогда придется координировать свою работу с администратором, который имеет доступ с правом sudo, чтобы все было установлено корректно.

Доступ к сети

Хоть мы и запускаем все на одном сервере, все равно потребуется заходить на него для администрирования, а поскольку речь идет о веб-сай-

те, то пользователи должны иметь доступ к нему из броузера. Вам также будет необходим доступ по протоколу SSH и открытые порты 80 и 443. Если сервер находится за брандмауэром, эти порты необходимо открыть. Мы настоятельно рекомендуем запускать SSH на нестандартном порту во избежание некоторых распространенных атак (дополнительную информацию о портах см. в разделе, посвященном безопасности).

Ярус веб-сервера

В настоящее время для промышленной эксплуатации приложений Rails в качестве фронтального сервера рекомендуется использовать быстрый «статический» веб-сервер, например Apache или Nginx, за которым находится кластер серверов Mongrel. Фронтальный веб-сервер получает входящий запрос и при необходимости передает его одному из процессов Mongrel, выступая в роли инвертированного прокси-сервера (например, вы можете настроить правила так, что статические ресурсы или кэшированные файлы будут отдаваться напрямую). Такой подход применяется большинством компаний, предоставляющих профессиональный хостинг для Rails, а также на многих промышленных кластерах. Он более надежен, чем FastCGI, SCGI и другие решения, применявшиеся в прошлом.

В качестве фронтальных веб-серверов лучше использовать Apache 2.2.x и Nginx. Мы будем работать с сервером Nginx, потому что он быстрый, надежный и гораздо проще Apache. Кроме того, Nginx потребляет совсем немного памяти. Для западного полушария он в новинку, но на сайте сервера утверждается, что в России на нем работает примерно пятая часть всех сайтов[1]. Одна из основных причин, по которым мы предпочитаем данный сервер Apache, заключается в том, что он надежнее работает в режиме инвертированного прокси-сервера. Apache попрежнему остается хорошим решением, но, если вы склоняетесь к нему только потому, что уже знакомы с этим сервером, предлагаем хотя бы попробовать Nginx. Вы будете приятно удивлены. Если же вы *вынуждены* использовать Apache, то на сайте Mongrel имеется отличная подборка материалов на тему о том, как настроить модуль `mod_proxy_balancer` для работы с кластером Mongrel Cluster[2].

Ярус сервера приложений

На ярусе сервера приложений необходим минимальный набор инструментов. Для начала хватит Ruby, RubyGems, Rails и gem-пакетов, от которых они зависят. В разделе «Установка» мы перечислим gem-пакеты, необходимые для работы самой системы.

---

[1] Домашняя страница проекта Nginx находится по адресу *http://nginx.net*.

[2] Инструкции по настройке Apache для взаимодействия с Mongrel Cluster с помощью `mod_proxy_balancer` см. на странице *http://mongrel.rubyforge.org/docs/apache.html*.

Наша конфигурация ориентирована на развертывание одного простого приложения Rails. Если ваши требования этим не исчерпываются, например необходим процесс BackrounDRb, задания `cron` и т. д., займитесь этими деталями самостоятельно. Не забывайте, что при более сложной конфигурации необходимо правильно настроить задания Capistrano. Подробнее о том, что такое Capistrano, мы расскажем в следующей главе.

Ярус базы данных

В большинстве приложений Rails используется только одна база данных. В простой конфигурации база данных размещается там же, где веб-сервер. В более сложных случаях может возникнуть необходимость настраивать базу данных с учетом избыточности, перехвата управления при отказе и производительности. Коль скоро сервер базы данных настраивается простейшим образом, эта процедура не будет сильно отличаться от обычной разработки на локальной машине.

Мы рекомендуем MySQL версии 5.x, однако версия 4.x в значительной мере равноценна и будет прекрасно работать с большинством проектов. Вы сами знаете, какие требования к базе данных предъявляет ваше приложение, но, выбирая другую СУБД, обязательно познакомьтесь поддержкой, которую предоставляет для нее Rails.

Мониторинг

Технически инструменты мониторинга не являются необходимыми для промышленной эксплуатации, но, следуя проверенной практике, мы рассматриваем их как обязательные. Без инструментов мониторинга эксплуатация приложений Rails подобна езде без приборов. Поскольку речь идет о промышленном окружении, вы, наверное, захотите знать, что сайт стал недоступен или что кто-то пытается загрузить файл и потребляет слишком много процессорного времени или ресурсов MySQL.

Управление версиями

Для управления версиями мы рекомендуем систему Subversion. Вы, вероятно, уже храните свое приложение в какой-то подобной системе (если нет, самое время начать). Советуем также хранить в системе управления версиями и другие важные файлы. Об этом поговорим в разделе «Конфигурация».

Установка

В данном разделе мы установим все программы и инструменты, необходимые для развертывания. Предварительно удостоверьтесь, что вы-

полнены все сформулированные в предыдущем разделе требования или вы знаете, как их обойти.

Мы будем собирать инструменты из исходных текстов, но вы, если хотите, можете воспользоваться своим любимым менеджером пакетов. Иногда менеджеры пакетов упрощают последующее обновление, но обычно они отстают на несколько ревизий от последних версий инструментов.

Примечание

У каждого есть собственные представления о том, где должны находиться библиотеки. Если вы предпочитаете устанавливать программы не в места, показанные в примерах, мы не возражаем. Мы установим библиотеку в каталог `/usr/local`, а само приложение – в каталог `/var/www/apps/railsway/`. Именно этот путь вы встретите в некоторых конфигурационных файлах.

Ruby

Текущей рекомендованной версией Ruby является ветвь 1.8.5. В версии 1.8.6 были обнаружены некоторые ошибки, поэтому если на этапе разработки вы не удостоверились на 100% в ее работоспособности, лучше пользоваться последней выпущенной версией ветви 1.8.5. На момент работы над данной книгой это была версия с уровнем 52. Для загрузки и сборки Ruby из исходных текстов выполните следующие команды:

```
$ curl -O ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.5-p52.tar.gz
$ tar zxvf ruby-1.8.5-p52.tar.gz
...
$ cd ruby-1.8.5-p52
$ ./configure –prefix=/usr/local
$ make
$ sudo make install
```

Система RubyGems

Текущая версия RubyGems имеет номер 0.9.4. После ее установки все остальные библиотеки для Ruby можно устанавливать как gem-пакеты. Для загрузки RubyGems и сборки из исходных текстов выполните показанные ниже команды. После установки система RubyGems может сама обновлять себя по мере выпуска новых версий:

```
$ curl -O http://files.rubyforge.mmmultiworks.com/rubygems/rubygems-
0.9.4.tgz
$ tar zxvf rubygems-0.9.4.tgz
...
$ cd rubygems-0.9.4
$ sudo ruby setup.rb
...
```

Rails

Мы работаем с последней версией ветви 1.2.x, на момент работы над книгой она имела номер 1.2.3. Флаг `-y` в команде ниже эквивалентен `–include-dependencies`. Следующая команда устанавливает gem-пакеты Rails (по одному для каждой платформы):

```
$ sudo gem install rails -y
```

Mongrel

Мы работаем с версией Mongrel 1.0.1, которая наиболее широко используется для промышленной эксплуатации. На локальной машине, где ведется разработка, команда `mongrel_rails` (или `script/server`) обычно набирается вручную. Но, настроив сценарии `init`, мы сможем пользоваться ими для запуска, останова и перезапуска кластера. Следующая команда установит gem-пакет Mongrel и пакеты, от которых он зависит:

```
$ sudo gem install mongrel -y
```

Mongrel Cluster

Mongrel Cluster – это gem-пакет, позволяющий организовать набор процессов Mongrel с общей конфигурацией, чтобы Nginx могла служить для них прокси-сервером. Мы пользуемся версией 1.0.2.

Следующая команда установит gem-пакет `mongrel_cluster`, который позволит конфигурировать и запускать процессы Mongrel «одним блоком». Во время настройки статического веб-сервера мы укажем на сконфигурированный кластер `mongrel_cluster`. Программа `mongrel_rails` предоставляет вам команды для управления кластером:

```
$ sudo gem install mongrel_cluster -y
```

Nginx

Nginx – это быстрый и простой фронтальный статический веб-сервер. Он отвечает за обработку входящих HTTP-запросов как собственными силами (например, для раздачи статического контента, уже хранящегося на диске), так и путем передачи их процессам Mongrel, работающим в составе кластера. Следующие команды загружают и собирают текущую стабильную версию Nginx из ветви 0.5.x:

```
$ curl -O http://sysoev.ru/nginx/nginx-0.5.28.tar.gz
$ tar zxvf nginx-0.5.28.tar.gz
$ cd nginx-0.5.28
$ ./configure --sbin-path=/usr/local/sbin --with-http_ssl_module
...
$ make
```

```
...
$ sudo make install
```

Subversion

Система управления версиями должна входить в ваш стандартный арсенал. Совместно с Capistrano можно использовать любую подобную систему, но предпочтение отдается Subversion. Следующие команды загружают, собирают и устанавливают Subversion и некоторые дополнительные зависимости для нее:

```
$ curl -O http://subversion.tigris.org/downloads/subversion-
1.4.4.tar.gz
$ curl -O http://subversion.tigris.org/downloads/subversion-deps-
1.4.4.tar.gz
$ tar zxvf subversion-1.4.4.tar.gz
$ tar zxvf subversion-deps-1.4.4.tar.gz
$ cd subversion-1.4.4
$ ./configure --prefix=/usr/local --with-openssl --with-ssl --with-zlib
...
$ make
...
$ sudo make install
```

MySQL

MySQL 5.x можно установить с помощью менеджера пакетов, собрать из исходных текстов или взять заранее откомпилированный дистрибутив для вашей платформы. Конкретный способ зависит от ваших предпочтений и ограничений. Мы остановились на двоичном дистрибутиве.

Найти подходящий пакет вы можете на странице *http://dev.mysql.com/downloads/mysql/5.0.html*.

Мы настоятельно рекомендуем обезопасить MySQL, задав пароль для пользователя root и ограничив доступ к локальной машине. Дополнительную информацию по этому поводу можно найти на странице *http://www.securityfocus.com/infocus/1726*.

Monit

Monit[1] – отличный инструмент мониторинга для управления процессами и слежения за потреблением ресурсов. Он прекрасно конфигурируется и может извещать о многих важных метриках (потребление ЦП, место на диске и т. д.). Следующие команды загружают и собирают Monit из исходных текстов:

---

[1] Домашняя страница проекта Monit находится по адресу *http://www.tildeslash.com/monit/*.

```
$ curl -O http://www.tildeslash.com/monit/dist/monit-4.9.tar.gz
$ tar zxvf monit-4.9.tar.gz
...
$ cd monit-4.9
$ ./configure
...
$ make
...
$ sudo make install
...
```

Capistrano

Устанавливать систему Capistrano следует не на сервер, а на локальную машину. Мы пользуемся последней версией – 2.x. Подробно Capistrano будет рассматриваться в главе 21. Следующая команда загружает и устанавливает gem-пакет Capistrano и зависимости для него:

```
$ sudo gem install capistrano -y
```

Конфигурация

Теперь все установлено, и мы можем заняться конфигурированием каждого инструмента. Для простейших приложений Rails вам понадобится сконфигурировать только Mongrel Cluster, Nginx и Monit.

Один из способов упростить и автоматизировать управление системой – создать каталог `deploy` прямо в каталоге `config/` вашего приложения, а в нем – отдельные подкаталоги для каждого вида развертывания (например, `dev`, `staging`, `production`). В подкаталогах будут лежать отдельные копии файлов `mongrel_cluster.yml`, `database.yml`, `nginx.conf`, `railsway.conf` и т. д. В заданиях Capistrano, выполняемых после развертывания, вы можете скопировать эти файлы в каталог `/var/www/railsway/shared/config` и поставить на них символические ссылки из тех мест, где они должны находиться (например, `/etc/nginx/nginx.conf -> /var/www/railsway/shared/config/nginx.conf`).

Конфигурирование Mongrel Cluster

Мы сгенерируем базовый конфигурационный файл `mongrel_cluster` с помощью команды `configure`, входящей в состав gem-пакета. Вы в любой момент сможете отредактировать этот файл вручную, но первоначальную версию удобнее создать автоматически. Сделать это можно, например, так:

```
$ mongrel_rails cluster::configure -p 8000 -e production \
-a 127.0.0.1 -N 2 --user deploy --group deploy \
-P /var/www/apps/railsway/shared/pids/mongrel.pid \
-c /var/www/apps/railsway/current
```

В результате будет создан файл `mongrel_cluster.yml`, показанный в следующем листинге. Можете сохранить его в каталоге `shared` дерева Capistrano (например, `/var/www/railsway/shared/config/mongrel_cluster.yml`).

```
--
cwd: /var/www/apps/railsway/current
port: '8000'
user: deploy
group: deploy
environment: production
address: 127.0.0.1
pid_file: /var/www/apps/railsway/shared/pids/mongrel.pid
servers: 2
```

Конфигурирование Nginx

Мы создадим для Nginx два конфигурационных файла: `nginx.conf` и `railsway.conf`. Так проще отделить глобальные параметры от параметров, относящихся к конкретному приложению.

nginx.conf

Это основной конфигурационный файл Nginx. В нем определяется pid, базовые средства протоколирования, gzip-сжатие и типы MIME. Обратите внимание на последнюю строчку, в которой включается файл `railsway.conf`. Можно было бы просто вставить в это место содержимое файла `railsway.conf`, но читать и изменять отдельный файл проще. Такой подход особенно удобен, когда на одном сервере работает несколько приложений.

```
# пользователь и группа, от имени которых работает сервер
user deploy deploy;
# количество рабочих процессов nginx
worker_processes 4;
# pid главного процесса nginx
pid /var/run/nginx.pid;
error_log /var/log/nginx/default.error.log debug;
# Количество соединений. По умолчанию принимается 1024
events {
  worker_connections 8192;
  use epoll; # только для linux!
}
# запустить модуль http, где конфигурируется доступ по протоколу http.
http {
  # Включить типы mime. Конфигурационный файл можно разбить на сколько
  # угодно включаемых файлов, чтобы сделать его проще
  include /etc/nginx/mime.types;
  # задать тип, принимаемый по умолчанию, для тех редких ситуаций, когда
  # ни один из включенных типов MIME не подходит
  default_type application/octet-stream;
  # конфигурируем формат протокола
```

```
  log_format main '$remote_addr - $remote_user [$time_local] '
                  ''$request' $status $body_bytes_sent
'$http_referer' '
                  ''$http_user_agent' '$http_x_forwarded_for'';
  # в OSX sendfile нет
  sendfile on;
  # Эти значения вполне можно принять по умолчанию.
  tcp_nopush on;
  tcp_nodelay on;
  # сжатие выходного потока экономит сетевые ресурсы
  gzip on;
  gzip_http_version 1.0;
  gzip_comp_level 2;
  gzip_proxied any;
  gzip_types text/plain text/html text/css application/
x-javascript text/xml application/xml application/xml+rss text/
javascript;
  access_log /var/log/nginx.default.access.log main;
  error_log /var/log/nginx.default.error.log info;
  # Управляемые приложения
  include /etc/nginx/railsway.conf;
}
```

railsway.conf

Это конфигурационный файл конкретного приложения. Он включается в главный файл `nginx.conf`, показанный выше. Как и в случае Apache, при конфигурировании SSL необходимо просто скопировать набор параметров. Мы включили пример конфигурации SSL, но, если вы не пользуетесь данным протоколом, можете удалить соответствующий раздел.

```
upstream railsway {
  server 127.0.0.1:8000;
  server 127.0.0.1:8001;
}
server {
  # Какой порт слушать. Можно также задать значение IP:PORT
  listen 80 default;
  # Максимальный размер загружаемого файла - 50Mb
  client_max_body_size 50M;
  # определить домен(ы), которые обслуживает этот виртуальный сервер
  server_name railsway.com;
  # корневой каталог
  root /var/www/apps/railsway/current/public;
  # протоколы конкретного виртуального сервера
  access_log /var/www/apps/railsway/shared/log/railsway.access.log
main;
  error_log /var/www/apps/railsway/shared/log/railsway.error.log
notice;
  # Следующая строка переадресует все запросы к maintenance.html, если
```

```
  # такая страница существует в корневом каталоге. Необходимо для
  # задания capistrano disable web
  if (-f $document_root/system/maintenance.html) {
    rewrite ^(.*)$ /system/maintenance.html last;
    break;
  }
  # Запретить доступ к путям, содержащим .svn
  location ~* ^.*\.svn.*$ {
    internal;
  }
  location / {
    index index.html index.htm;
    # Передавать Rails IP-адрес клиента
    proxy_set_header X-Real-IP $remote_addr;
    # необходимо для HTTPS
    proxy_set_header X_FORWARDED_PROTO https;
    proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect false;
    proxy_max_temp_file_size 0;
    location ~ ^/(images|javascripts|stylesheets)/ {
      expires 10y;
    }
    if (-f $request_filename) {
      break;
    }
    if (-f $request_filename/index.html) {
      rewrite (.*) $1/index.html break;
    }
    if (-f $request_filename.html) {
      rewrite (.*) $1.html break;
    }
    if (! -f $request_filename) {
      proxy_pass http://railsway;
      break;
    }
  }
  error_page 500 502 503 504 /500.html;
  location = /500.html {
    root /var/www/apps/railsway/current/public;
  }
}

server {
  # Какой порт слушать. Можно также задать значение IP:PORT
  listen 443 default;
  # Максимальный размер загружаемого файла - 50Mb
  client_max_body_size 50M;
  # определить домен(ы), которые обслуживает этот виртуальный сервер
  server_name railsway.com;
```

```
  # конфигурация SSL-сертификатов
  ssl on;
  ssl_certificate /etc/nginx/ssl/railsway.cert;
  ssl_certificate_key /etc/nginx/ssl/railsway.key;
  keepalive_timeout 70;
  add_header Front-End-Https on;
  # корневой каталог
  root /var/www/apps/railsway/current/public;
  # протоколы конкретного виртуального сервера
  access_log /var/www/apps/railsway/shared/log/railsway.access.log
main;
  error_log /var/www/apps/railsway/shared/log/railsway.error.log
notice;
  # Следующая строка переадресует все запросы к maintenance.html, если
  # такая страница существует в корневом каталоге. Необходимо для
  # задания capistrano disable web
  if (-f $document_root/system/maintenance.html) {
    rewrite ^(.*)$ /system/maintenance.html last;
    break;
  }
  # Запретить доступ к путям, содержащим .svn
  location ~* ^.*\.svn.*$ {
    internal;
  }
  location / {
    index index.html index.htm;
    # Передавать Rails IP-адрес клиента
    proxy_set_header X-Real-IP $remote_addr;
    # необходимо для HTTPS
    proxy_set_header X_FORWARDED_PROTO https;
    proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect false;
    proxy_max_temp_file_size 0;
    location ~ ^/(images|javascripts|stylesheets)/ {
      expires 10y;
    }
    if (-f $request_filename) {
      break;
    }
    # кэширование страниц в Rails, часть 1
    # Добавить '/index.html' в конец пути текущего запроса и
    # посмотреть,есть ли такой файл в системе.
    if (-f $request_filename/index.html) {
      rewrite (.*) $1/index.html break;
    }
    # кэширование страниц в Rails, часть 2
    if (-f $request_filename.html) {
      rewrite (.*) $1.html break;
    }
```

```
    if (! -f $request_filename) {
      proxy_pass http://railsway;
      break;
    }
  }
  error_page 500 502 503 504 /500.html;
  location = /500.html {
    root /var/www/apps/railsway/current/public;
  }
}
```

Можете протестировать эти конфигурационные файлы, выполнив следующую команду:

```
$ sudo /usr/local/sbin/nginx -t -c config/nginx.conf
```

Конфигурирование Monit

Конфигурационный файл Monit читается совсем легко. В следующем, довольно обширном, примере основные процессы, работающие на сервере, проверяются один раз в минуту, и, если какое-нибудь из определенных условий оказывается не выполненным, вам посылается уведомление. Можно проверять потребление диска и процессора на уровне системы или процесса, а также среднюю загрузку. Можно следить за тем, какие процессы работают, и даже за тем, запускался ли некоторый процесс заданное число раз в заданный промежуток времени. Другие полезные примеры конфигурации Monit можно найти в файле `monitrc`, который поставляется вместе с исходными текстами программы.

```
set daemon 60 # Опрашивать с интервалом 1 минута

set logfile /var/log/monit.log

set alert monit-alerts@railsway.com

set mail-format {
     from: monit@railsway.com
     subject: $SERVICE service - $EVENT
     message: $ACTION $SERVICE on $HOST: $DESCRIPTION
}

set httpd port 1380
  allow localhost # Разрешать соединение localhost

check system railsway.com
  alert monit-alerts@railsway.com but not on { instance }
  if loadavg(1min) > 4 for 3 cycles then alert
  if loadavg(5min) > 3 for 3 cycles then alert
  if memory usage > 80% for 3 cycles then alert
  if cpu usage (user) > 70% for 5 cycles then alert
```

```
  if cpu usage (system) > 30% for 5 cycles then alert
  if cpu usage (wait) > 20% for 5 cycles then alert

check process nginx with pidfile /var/run/nginx.pid
  start program = "/etc/init.d/nginx start"
  stop program = "/etc/init.d/nginx stop"
  if 2 restarts within 3 cycles then timeout
  if failed host localhost port 80 protocol http then restart
  if failed host localhost port 443 then restart

check process sendmail with pidfile /var/run/sendmail.pid
  start program = "/etc/init.d/sendmail start"
  stop program = "/etc/init.d/sendmail stop"

check process mysql with pidfile /var/run/mysqld/mysqld.pid
  start program = "/etc/init.d/mysqld start"
  stop program = "/etc/init.d/mysqld stop"

check process mongrel_8000 with pidfile /var/www/railsway/shared/pids/
mongrel.8000.pid

  start program = "/usr/bin/mongrel_rails cluster::start -C
/var/www/railsway/shared/config/mongrel_cluster.yml --clean --only
8000"
  stop program = "/usr/bin/mongrel_rails cluster::stop -C
/var/www/railsway/shared/config/mongrel_cluster.yml --clean --only
8000"

if failed port 8000 protocol http
  with timeout 10 seconds
  then restart

if totalmem is greater than 128 MB for 4 cycles then restart
  # жрет память?
if cpu is greater than 60% for 2 cycles then alert
  # послать администратору уведомление по почте
if cpu is greater than 90% for 5 cycles then restart
  # зависший процесс?
if loadavg(5min) greater than 10 for 8 cycles then restart
  # очень, очень плохо
if 3 restarts within 5 cycles then timeout
  # что-то не так, пора вызывать сисадмина
group mongrel

check process mongrel_8001 with pidfile /var/www/railsway/shared/
pids/mongrel.8001.pid
start program = "/usr/bin/mongrel_rails cluster::start -C /var/www/
railsway/shared/config/mongrel_cluster.yml --clean --only 8001"
stop program = "/usr/bin/mongrel_rails cluster::stop -C /var/www/
railsway/shared/config/mongrel_cluster.yml --clean --only 8001"
```

```
if failed port 8001 protocol http
  with timeout 10 seconds
  then restart

if totalmem is greater than 128 MB for 4 cycles then restart
  # жрет память?
if cpu is greater than 60% for 2 cycles then alert
  # послать администратору уведомление по почте
if cpu is greater than 90% for 5 cycles then restart
  # зависший процесс?
if loadavg(5min) greater than 10 for 8 cycles then restart
  # очень, очень плохо
if 3 restarts within 5 cycles then timeout
  # что-то не так, пора вызывать сисадмина
group mongrel
```

Конфигурирование Capistrano

Capistrano играет решающую роль в завершении настройки промышленной системы. В следующей главе подробно описано, как настраивать конфигурацию развертывания для работы вашего приложения в режиме промышленной эксплуатации.

Конфигурирование сценариев init

Выше мы уже упоминали, что экономия времени и сил при конфигурировании промышленного окружения обусловлена прежде всего *автоматизацией*. Управление процессами отлично поддается автоматизации. Для этого предназначены сценарии `init`, которые служат двум целям. Во-первых, они запускаются на этапе начальной загрузки системы, а, во-вторых, останова и перезапуска – с помощью оболочки (вручную) или при выполнении команд Capistrano, описанных в заданиях запуска. Мы включили примеры для Mongrel, Monit и Nginx, хотя два последних представляют собой сценарии оболочки и зависят от конкретной операционной системы. Сценарий `init` для Mongrel написан на Ruby и является достаточно общим.

Сценарий init для Nginx

Этот сценарий `init` необходим для запуска, останова или изменения конфигурации Nginx. В сети (с помощью любого поисковика) нетрудно найти сценарий именно для вашей ОС. Следующий пример ориентирован на дистрибутив CentOS Linux. Файл должен называться `/etc/init.d/nginx`.

```
#!/bin/sh
# v1.0
# nginx - Запуск, останов и переконфигурация nginx
```

```
#
# chkconfig: - 60 50
# описание: nginx [engine x] – это облегченный http web/proxy сервер,
#           который отвечает также на запросы по протоколу ftp.
# processname: nginx
# config: /etc/nginx/nginx.conf
# pidfile: /var/run/nginx.pid

# Загрузить библиотеку функций.
. /etc/rc.d/init.d/functions

# Загрузить конфигурацию сети.
. /etc/sysconfig/network

# Проверить, что сеть работает.
[ ${NETWORKING} = "no" ] && exit 0

BINARY=/usr/sbin/nginx
CONF_FILE=/etc/nginx/nginx.conf
PID_FILE=/var/run/nginx.pid
[ -x $BINARY ] || exit 0

RETVAL=0
prog="nginx"

start() {
  # Запустить демонов.
  if [ -e $BINARY ] ; then

    echo -n $"Starting $prog: "
    $BINARY -c $CONF_FILE
    RETVAL=$?
    [ $RETVAL -eq 0 ] && {
      touch /var/lock/subsys/$prog
      success $"$prog"
    }
    echo
  else
    RETVAL=1
  fi
  return $RETVAL
}

stop() {
  # Остановить демонов.
  echo -n $"Shutting down $prog: "
  kill -s QUIT `cat $PID_FILE 2>/dev/null`
  RETVAL=$?
  echo
  [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/$prog
  return $RETVAL
```

```
}

# Смотрим, как нас вызвали.
case "$1" in
start)
  start
  ;;
stop)
  stop
  ;;
reconfigure)
  if [ -f /var/lock/subsys/$prog ]; then
    kill -s HUP `cat $PID_FILE 2>/dev/null`
    RETVAL=$?
  fi
  ;;

status)
  status $prog
  RETVAL=$?
  ;;
*)
  echo $"Usage: $0 {start|stop|reconfigure|status}"
  exit 1
esac

exit $RETVAL
```

Сценарий init для Mongrel

Этот Ruby-сценарий должен находиться в файле `/etc/init.d/mongrel` (в вашей ОС сценарии `init` могут размещаться в другом месте). Он позволяет запускать, останавливать и перезапускать процессы Mongrel. Если вы поместите данный сценарий в каталог `init.d`, то он будет выполняться при старте сервера.

```
#! /usr/bin/env ruby
#
# mongrel Сценарий запуска кластеров Mongrel.
#
# chkconfig: 345 85 00
#
# описание: mongrel_cluster_управляет совокупностью процессов Mongrel
# для работы за балансировщиком нагрузки.
#
MONGREL_RAILS = '/usr/bin/mongrel_rails'
CONF_FILE     = '/etc/railsway/mongrel_cluster.yml'
SUBSYS        = '/var/lock/subsys/mongrel'
SUDO          = '/usr/bin/sudo'
case ARGV.first
  when 'start'
```

```
    '#{MONGREL_RAILS} cluster::start -C #{CONF_FILE}'
    '#{SUDO} touch #{SUBSYS}'
  when 'stop'
    '#{MONGREL_RAILS} cluster::stop -C #{CONF_FILE}'
    '#{SUDO} rm -f #{SUBSYS}'
  when 'restart'
    '#{MONGREL_RAILS} cluster::restart -C #{CONF_FILE}'
  when 'status'
    '#{MONGREL_RAILS} cluster::status -C #{CONF_FILE}'
  else
    puts 'Usage: /etc/init.d/mongrel {start-stop-restart-status}'
    exit 1
end
exit $?
```

Сценарий init для Monit

Вам потребуется также сценарий `init` для управления Monit. Если вы решите использовать Monit для наблюдения за процессами, он будет вызывать другие сценарии из заданий Capistrano. Необходимо, чтобы Monit отвечал за запуск, останов и перезапуск заданий, потому что, если вы остановите процесс вручную, Monit скоро обнаружит, что процесс не работает, и запустит его заново. Но именно такого развития событий вы хотите избежать, когда на сервере возникают проблемы, из-за которых требуется остановить некий процесс.

Как и показанный выше сценарий `init` для Nginx, этот сценарий «заточен» под CentOS. Вариант для вашей системы можете поискать в Сети.

```
#! /bin/sh
#
# monit Монитор Unix-систем
#
# Автор: Clinton Work, <work@scripty.com>
#
# chkconfig: 2345 98 02
# описание: Monit - утилита для мониторинга и управления процессами,
# файлами, каталогами и устройствами в Unix-системах.
# processname: monit
# pidfile: /var/run/monit.pid
# config: /etc/mcommons/monitrc

# Загрузить библиотеку функций.
. /etc/rc.d/init.d/functions

# Загрузить конфигурацию сети.
. /etc/sysconfig/network

MONIT=/usr/local/bin/monit
CONFIG=/etc/monitrc
```

```
# Загрузить конфигурацию monit.
if [ -f /etc/sysconfig/monit ] ; then
      . /etc/sysconfig/monit
fi

[ -f $MONIT ] || exit 0

RETVAL=0

# Смотрим, как нас вызвали.
case "$1" in
  start)
       echo -n "Starting monit: "
       daemon $NICELEVEL $MONIT -c $CONFIG
       RETVAL=$?
       echo
       [ $RETVAL = 0 ] && touch /var/lock/subsys/monit
       ;;
  stop)
       echo -n "Stopping monit: "
       killproc monit
       RETVAL=$?
       echo
       [ $RETVAL = 0 ] && rm -f /var/lock/subsys/monit
       ;;
  restart)
       $0 stop
       $0 start
       RETVAL=$?
       ;;
  condrestart)
       [ -e /var/lock/subsys/monit ] && $0 restart
       ;;
  status)
       status monit
       RETVAL=$?
       ;;
  *)
       echo "Usage: $0 {start|stop|restart|condrestart|status}"
       exit 1
esac

exit $RETVAL
```

Развертывание и запуск

Итак, сервер работает, и все готово для развертывания на нем приложения. Инструкции по подготовке приложения к развертыванию мы отложим до главы 21. Исходя из вышеупомянутой структуры каталогов, вы можете понять, что мы собираемся запускать приложение из

каталога `/var/www/apps/railsway` и ожидаем, что `monit` будет управлять процессами.

Вы должны знать, где хранятся протоколы вашего приложения, а также серверов Mongrels, Nginx, Monit и т. д. Часто именно в них вы заглядываете, когда возникает какая-то ошибка. Чтобы быстро зайти на сайт, не открывая броузера, можно воспользоваться программой `curl`.

Другие замечания по поводу промышленной системы

При проектировании промышленного окружения всегда следует принимать во внимание описанные ниже соображения. Каждое из них может потребовать серьезных усилий и затрат на оборудование. Необходимость в описанных действиях зависит от ваших приоритетов и в некоторых случаях от параноидального склада ума. Имейте в виду, что речь в любом случае идет о компромиссе между временем, деньгами и эффективностью.

Избыточность и перехват управления при отказе

Что происходит, когда «падает» база данных или сбоит диск? Избыточность и механизм перехвата управления при отказе (failover) позволяют отреагировать на сбой за счет одного или нескольких уровней программного или аппаратного обеспечения. Мы не собираемся подробно говорить на эту тему, поскольку на каждом ярусе промышленной системы есть много возможностей организовать избыточность и перехват управления.

Кэширование

Кэширование призвано повысить производительность системы за счет сохранения результатов наиболее популярных запросов. Это позволяет возвращать ответ быстрее, чем если бы пришлось обрабатывать каждый запрос от начала до конца. Обычно кэширование реализуется путем сохранения уже сгенерированных HTTP-ответов в виде файлов на диске (кэширование страниц), в памяти (например, с помощью Memcache) и т. д. На каждом уровне промышленной системы существует много способов организовать кэширование. Прекрасное учебное руководство по кэшированию в Rails имеется на странице *http://www.railsenvy.com/2007/2/28/rails-caching-tutorial*.

Производительность и масштабируемость

Термины «производительность» и «масштабируемость» часто считают взаимозаменяемыми. Они действительно связаны между собой, но все

же описывают разные понятия и решения. Производительность измеряет поведение данной «единицы» ресурсов. Например, можно измерить производительность системы регистрации, получающей 20 тыс. запросов к странице входа в минуту при условии, что она работает на одном веб-сервере с одним соединением к базе данных. Повторные замеры для различных компонентов прикладного стека позволят составить представление об эталонной производительности, ожидаемой от данной «единицы».

Масштабируемость – это мера того, насколько эффективно архитектура системы допускает наращивание возможностей при увеличивающейся потребности. Иными словами, как изменяется отношение между потребностями и ресурсами по мере роста потребностей. Предположим, например, что имеется система, состоящая из трех серверов: веб-сервера, сервера приложений и сервера базы данных. Сколько дополнительной аппаратуры потребуется для обслуживания 2000 запросов/с, если замеры производительности показали, что она способна обслуживать 1000 запросов/с? В идеале архитектура должна обеспечивать «горизонтальное» (то есть пропорциональное) масштабирование. Это означает, что можно добавлять новые «единицы» ресурсов без падения производительности системы в целом. В предыдущем примере мы могли бы просто удвоить аппаратные мощности, но так бывает не всегда.

В реальных промышленных системах возможности масштабирования зависят от многих факторов, в частности, от поведения вашего приложения. Например, приложения, выполняющие много операций записи (то есть предложений `INSERT` для вставки в базу данных) – скажем, социальные сети или обработка финансовых транзакций, – требуют более сложных решений на базе кластерных СУБД, чем приложения, занимающиеся преимущественно чтением (блоги или агрегаторы новостей). Кроме того, различные части приложения могут масштабироваться по-разному. Точки интеграции с внешней системой, например с агентом доставки почты, шлюзом в платежную систему или внешней службой хранения данных, могут масштабироваться иначе, чем прочие компоненты приложения.

Дополнительную информацию по этим вопросам можно найти на странице *http://en.wikipedia.org/wiki/Scalability*. Масштабирование системы – слишком сложная тема, чтобы можно было надеяться хоть как-то раскрыть ее в этой главе, но в Сети имеются отличные ресурсы, в том числе в виде презентаций и заметок в блогах. Архитектура Rails основана на принципе «ничего не разделяй», как и архитектура ее предшественника, LAMP (Linux-Apache-MySQL-PHP). На практике доказано, что структурирование приложений таким способом эффективно и экономично. Этот подход эволюционировал по мере складывания индустрии в течение последнего десятилетия и отлично зарекомендовал себя в многочисленных приложениях. Многие ветераны архитектуры «ничего не разделяй» заработали свои лавры на ниве LAMP, но ничто

не мешает приложить их мудрость и к Rails. Поскольку наша промышленная система проста, в будущем будет нетрудно построить на ее базе более сложные конфигурации.

Безопасность

Здесь мы не можем рассказать о безопасности хоть сколько-нибудь полно, но, пожалуй, стоит упомянуть о нескольких простых шагах, которые позволят вам защититься хотя бы от самых очевидных атак.

Безопасность на уровне приложения

Вы можете закрыть свой сервер и сетевое оборудование на замок и упрятать всю аппаратуру в старой пусковой шахте, но это не спасет вас, если приложение открыто для атак. К счастью, инструменты и методики Rails позволяют без особого труда сделать приложение относительно безопасным с самого начала, а следование проверенным рекомендациям позволит избежать наиболее распространенных атак, например внедрения SQL или кросс-сайтовых сценариев (XSS). Дополнительную информацию можно найти в блоге, посвященном безопасности ROR, по адресу *http://www.rorsecurity.info*.

Закрывайте порты

Следует закрывать сетевые порты, оставляя открытыми только 80, 443 и нестандартный порт SSH. Если установлен брандмауэр, и вы можете сами его конфигурировать, фильтрацию следует производить на уровне брандмауэра. Если доступа к брандмауэру нет, закрыть порты можно на уровне сервера с помощью программы `iptables`[1].

Удобство сопровождения

После запуска приложения за ним следует наблюдать и поддерживать систему в чистоте. Кроме того, процедура диагностики и устранения ошибок должна быть достаточно отработанной. В режиме промышленной эксплуатации возможны разные странные вещи, которые трудно, а то и невозможно воспроизвести на локальной машине для разработки.

Заключение

Подготовка промышленного окружения – не такое уж устрашающее мероприятие, как было когда-то. И разработчики, и системные администраторы склонны придерживаться знакомых инструментов, что

[1] Домашняя страница iptables находится по адресу *http://www.netfilter.org/projects/iptables/index.html*.

и понятно в наш век специализации. Но Rails стремится порвать с этой тенденцией, заставляя разработчиков вылезти из своей норки и попробовать себя в роли администраторы базы данных, системного администратора или дизайнера пользовательских интерфейсов. Все необходимое для этого уже есть под руками. Вообще говоря, Сеть как среда для производства, публикации и распространения существенно упростила задачу продвижения новых знаний, продуктов и услуг в массы. Сама среда Rails и многочисленные библиотеки и инструменты, созданные для его поддержки, немало поспособствовали этой парадигме, облегчив выполнение даже таких «черных работ», как развертывание и системное администрирование.

В этой главе мы рассмотрели основные технические и философские компоненты организации современного промышленного окружения. Конечно, есть миллионы вариаций, и какие-то из них могут подойти вам больше. Но даже если вы пользуетесь PostgreSQL, FreeBSD и Perforce, то все равно сможете без труда адаптировать рекомендованную нами конфигурацию. Хотя советоваться с коллегами всегда полезно, особенно при первой настройке промышленной системы. Все же имейте в виду, что описанные выше подходы считаются оптимальной практикой, по крайней мере для приложений Rails.

Перед тем как запускать приложение в промышленную эксплуатацию, осталось сделать еще один большой шаг – сконфигурировать само приложение с помощью системы Capistrano, чтобы его можно было развернуть в только что подготовленном окружении. Этим мы и займемся в следующей главе.

21

Capistrano

Когда мы, разработчики для .NET, говорим, что Rails – это сложно, то имеем в виду оболочку Linux, серверные приложения и другие вещи, которые для нас выглядят непривычно и пугающе.

Брайан Энг,
http://www.softiesonrails.com/2007/4/5/the-absolute-moron-sguide-to-capistrano

Нужда – мать всех изобретений, и, работая над сайтом 37signals, Джейми создал систему Switchtower (позднее переименованную в Capistrano[1]), когда Basecamp разросся настолько, что одного промышленного сервера перестало хватать. Это средство для автоматизации выполнения заданий на удаленных серверах.

Ныне Capistrano прочно утвердилась в качестве стандартного решения задач развертывания в Rails, однако принятый в ней мультисерверный транзакционный подход к удаленному выполнению команд делает систему пригодной для гораздо более широкого спектра сценариев развертывания.

---

[1] Первоначальное название Switchtower было изменено на Capistrano из-за конфликта с существующей торговой маркой. Подробности см. на странице *http://weblog.rubyonrails.org/2006/3/6/switchtower-is-now-capistrano*.

В этой главе мы покажем, что система Capistrano (версии 2[1]) даже без всякой модификации способна решать сложные и отнимающие много времени задачи. Мы расскажем также о том, чего она делать не умеет, и как можно ее расширить с учетом особенностей развертывания в каждом конкретном случае. В заключение мы рассмотрим несколько полезных рецептов, разработанных сообществом пользователей Capistrano, которые получили всеобщее признание.

Обзор системы Capistrano

Начнем с изучения общей структуры Capistrano. Нам нужно будет познакомиться с применяемой в ней предметно-ориентированной терминологией.

Терминология

В Capistrano имеется собственный словарь, который делает ее не просто расширением системы Rake, на которой основана система.

- *Машины развертывания* – один или несколько компьютеров, на которых развертывается приложение.
- *Рецепты* – аналог заданий в Rake. Рецепт может состоять из одного или нескольких заданий и обеспечивает желаемое решение.
- *Задания* – атомарные единицы функциональности. Они могут вызываться напрямую конечным пользователем или другим заданием. Задания располагаются внутри пространств имен.
- *Пространство имен* – позволяет логически группировать несколько заданий. При этом в разных пространствах имен могут находиться одноименные задания, что позволяет авторам рецептов не опасаться конфликта имен.
- *Роли* (например, `:app` или `:db`) – представляют собой средства группового выполнения заданий. Мы можем сказать, что задание должно выполняться только в контексте конкретной роли. Можете считать, что роль – это некий квалификатор задания, обычно обозначающий класс машины развертывания, например `:app`, `:db` или `:web`, для которого будет выполняться задание.
- *Переменные* – глобальные переменные, доступные в любом месте сценария.

[1] Версия Capistrano 1.x хорошо документирована в Сети, но уже устарела, поэтому мы о ней ничего говорить не будем. На сайте *capify.org* имеются подробные инструкции по переходу на последние версии Capistrano.

Основы

Когда я только начал работать с Capistrano, я задал себе следующие вопросы: «Что нужно для использования Capistrano? Чего она ожидает от моего приложения? От моего сервера? Что Capistrano сделала по завершении развертывания, а чего не сделала?»

В следующих разделах мы ответим на эти вопросы и укажем, какое место занимает Capistrano в общей картине работы над проектом.

Что необходимо для использования Capistrano

Прежде всего, важно понимать, что Capistrano устанавливается только на машине, где ведется разработка (с которой производится развертывание). На машине развертывания устанавливать Capistrano необязательно. Для ее работы достаточно, чтобы эта машина удовлетворяла некоторым базовым требованиям и предположениям, ведь все команды исполняются с помощью безопасной инфраструктуры SSH.

Поэтому коротко на поставленный вопрос можно ответить так: изучите базовые требования и предположения и удовлетворите их. Вероятно, первый опыт развертывания будет последовательностью проб и ошибок. Но со временем вы поймете, что Capistrano – ваш главный союзник в этом деле.

Чего ожидает Capistrano

Некоторые требования обязательны, другие являются лишь предположениями, которые можно переопределить. Идеологически система поощряет применение рекомендованных методик, но не навязывает их.

Базовые требования Capistrano таковы:

- для доступа к удаленной машине используется SSH;
- на машине развертывания установлена совместимая с POSIX[1] оболочка;
- если вы пользуетесь паролями, то на всех машинах развертывания пароли одинаковы (рекомендуется применять технологию PKI)[2].

Следующие предположения могут и не выполняться:

- вы собираетесь развертывать приложение Rails;
- вы используете Subversion для управления версиями исходных текстов;

---

[1] Это о тебе, Windows, хотя кое-кому удавалось добиться успеха с cygwin.

[2] Наверное, об этом даже и говорить не стоит, но умоляю вас – подумайте о применении PKI. Так вы серьезно уменьшите риск взлома.

- вы уже подготовили промышленное окружение (операционная система, Ruby, Rails, другие gem-пакеты, СУБД, веб-сервер, сервер приложений, сервер базы данных);
- пароли доступа к машине развертывания и к репозиторию `svn` одинаковы;
- вы создали базу данных для развертывания и пользователя, имеющего к ней доступ;
- в Subversion имеются все конфигурационные файлы, готовые к работе в промышленном окружении (в частности, настроенные на имена/пароли пользователей для вышеупомянутой базы данных развертывания);
- весь набор миграций прогоняется без ошибок (`deploy:migrate`);
- веб-сервер и сервер приложений сконфигурированы сценарием `spin` для запуска/останова/перезапуска веб-приложения.

Как уже было сказано, эти требования и предположения подразумеваются по умолчанию. Если они вас не устраивают, не отчаивайтесь, – мы покажем, как создать задания и обратные вызовы для настройки Capistrano на ваши потребности.

Что Capistrano сделала, а что – нет

Установив Capistrano и удовлетворив ее требования (в исходном виде или путем настраивания), вы можете считать, что «оказались в нирване развертывания». Это, пожалуй, слишком сильно сказано, но ваш код будет развертываться прямо из репозитория, миграции будут запускаться, Apache и сервер приложений (Mongrel, fastcgi) – стартовать и... в общем, считайте, что все развернуто и готовьтесь к следующему подвигу.

Теперь можно делать такие важные вещи, как обновление сервера из последней сохраненной в Subversion ветви с помощью задания `cap deploy:update` или использование задания `cap deploy` для установки с нуля последней версии и перезапуска серверов. Можно даже на время длительного техобслуживания выводить специальную страницу (`cap deploy:web:disable`) и откатываться назад, если вы что-то напортачили с развертыванием.

При управлении несколькими серверами сами команды не изменяются, модифицировать приходится лишь конфигурацию. Небольшая модификация конфигурационных файлов – и вы сможете выполнять задания `cap deploy` и `cap deploy:invoke` сразу на всех серверах.

Приступаем к работе

Обозрев ландшафт с высоты птичьего полета, спустимся на землю. В первом упражнении мы будем считать, что все ожидания Capistrano, описанные в предыдущем разделе, удовлетворены.

Установка

Для начала установим Capistrano:

```
$ sudo gem install capistrano
Install required dependency net-ssh? [Yn]
Install required dependency net-sftp? [Yn]
Install required dependency highline? [Yn]
Successfully installed capistrano-2.0
```

Команда `cap` с флагом `--tasks` расскажет обо всех известных Capistrano заданиях[1].

```
$ cap --tasks
cap invoke #Выполнение одной команды на удаленных серверах.
cap shell #Начало интерактивного сеанса работы с Capistrano.
Для получения дополнительной информации о команде задайте флаг -e
cap -e command #т.е. cap -e deploy:pending:diff
```

Это два встроенных задания общего назначения, позволяющих выполнять одну или несколько команд на машинах развертывания. Задание `invoke` служит для выполнения одной команды, а задание `shell` открывает интерактивный сеанс (например, `irb`), где можно выполнять команды одну за другой. Но где же все те удивительные вещи, которые умеет делать Capistrano?

Готовим приложение Rails для работы с Capistrano

Для подготовки проекта к развертыванию Capistrano предоставляет команду `capify`, которая создает базовые конфигурационные файлы. Вам предстоит слегка их отредактировать – и можно развертывать. Для проекта `my_project`, удовлетворяющего стандартным предположениям Capistrano, будут созданы два файла:

```
$ cd my_project
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

Но прежде чем открывать эти файлы, запустим `cap --tasks` еще раз:

```
$ cap --tasks
cap deploy            #Развертывает ваш проект.
cap deploy:check      #Проверяет зависимости для развертывания.
cap deploy:cleanup    #Стирает старые версии.
cap deploy:cold       #Развертывает и запускает 'холодное' приложение.
cap deploy:migrate    #Запускает rake-задание migrate.
cap deploy:migrations #Развертывает и запускает ожидающие миграции.
```

[1] Команда `cap` печатает сообщения на английском языке. Они переведены для удобства читателя. – *Прим. перев.*

```
cap deploy:pending       #Отображает все сохранения в системе управления
                         #версиями с момента последнего развертывания.
cap deploy:pending:diff  #Отображает дельту с момента последнего развертывания.
cap deploy:restart       #Перезапускает приложение.
cap deploy:rollback      #Откатывает на предыдущую версию и перезапускает.
cap deploy:rollback_code #Откатывает на ранее развернутую версию.
cap deploy:setup         #Подготавливает один или несколько серверов
                         #к развертыванию.
cap deploy:start         #Запускает серверы приложений.
cap deploy:stop          #Останавливает серверы приложений.
cap deploy:symlink       #Обновляет символическую ссылку на развернутую версию.
cap deploy:update        #Копирует проект и обновляет символическую ссылку.
cap deploy:update_code   #Копирует проект на удаленные серверы.
cap deploy:upload        #Копирует файлы в текущую развернутую версию.
cap deploy:web:disable   #Выводит посетителям страницу о закрытии на
                         #техобслуживание.
cap deploy:web:enable    #Восстанавливает доступность приложения из веб.
cap invoke               #Выполнение одной команды на удаленных серверах.
cap shell                #Начало интерактивного сеанса работы с Capistrano.
```

Вот теперь на что-то похоже! Но откуда взялись эти новые задания? Для ответа на этот вопрос заглянем в файл `Capfile`**:**

```
$ cat Capfile
load 'deploy' if respond_to?(:namespace) # cap2 differentiator
Dir['vendor/plugins/*/recipes/*.rb'].each { |plugin| load(plugin)
}load 'config/deploy'
```

Совсем короткий! Как видите, команда `cap` **загружает рецепты, читая содержимое файла** `Capfile` **в текущем каталоге. Как и Rake, Capistrano поднимается по дереву каталогов, пока не найдет** `Capfile`**, следовательно, вы можете вызывать** `cap` **из любого подкаталога проекта.**

Файл `Capfile`**, который строит** `capify`**, копирует ряд стандартных рецептов Capistrano в сценарий** `deploy`**, а рецепты, специфичные для вашего проекта, – в сценарий** `config/deploy`**:**

```
my_project> $ cat config/deploy.rb
set :application, "set your application name here"
set :repository, "set your repository location here"

# Если развертывание производится не в каталог /u/apps/#{application} на
# конечных серверах (это умолчание), то можно задать реальное
# местоположение с помощью переменной :deploy:
# set :deploy_to, "/var/www/#{application}"

# Если для управления исходными тестами вы пользуетесь не Subversion, то
# ниже укажите имя системы управления версиями:
# set :scm, :subversion

role :app, "your app-server here"
role :web, "your web-server here"
role :db,  "your db-server here", :primary => true
```

Конфигурирование развертывания

Мелкие правки в трафаретном файле `config/deploy.rb` – вот и все, что нужно для подготовки приложения к развертыванию. Этот файл описывает порядок развертывания приложения на удобном для восприятия языке.

Придумайте имя для своего приложения

Первая и самая главная настройка – имя приложения:

```
set :application, "set your application name here" # станет именем каталога
```

Хотя в тексте `deploy.rb` подсказка содержит пробелы, в реальном имени приложения употреблять их не стоит, поскольку в конечном итоге оно станет именем каталога на машине развертывания.

Информация о репозитории

Далее необходимо сообщить Capistrano, где искать исходный текст приложения:

```
set :repository, "set your repository location here"
```

Предполагая, что имеется сервер Subversion, запишите в `:repository` его URL (будь то `http`, `svn` или `svn+ssh`). Еще одно встроенное предположение заключается в том, что имя пользователя и пароль для учетной записи в Subversion такие же, как у пользователя, от имени которого производится развертывание. Соединение с `svn` устанавливается от имени пользователя, запустившего Capistrano, и, если сервер требует аутентификации, вы увидите соответствующее сообщение.

Определите роли

Далее мы должны проинформировать Capistrano о доменном имени или IP-адресе машины (машин) развертывания. Capistrano будет открывать SSH-сеанс для выполнения заказанных вами действий. В данном простом случае все три роли (они же классы) машин будут идентичны:

```
role :app, "my_deployment_machine" #можно также указать IP-адрес
role :web, "my_deployment_machine"
role :db,  "my_deployment_machine", :primary => true
```

Роли – это мощное средство, позволяющее выполнять определенные задания на машинах одного класса. Например, с помощью задания `deploy shell` можно выполнить команду `grep` на всех машинах класса `:app`.

Дополнительные свойства ролей

В нашем примере роль `:db` помечена опцией `:primary => true`. Так обозначается основной сервер базы данных. Некоторые задания, напри-

мер миграции базы данных, выполняются только на основном сервере, так как в большинстве кластерных конфигураций СУБД подчиненные серверы синхронизируются с основным. Вы можете также указать, что данная роль относится к подчиненному серверу (опция `:slave => true`), чтобы можно было направлять на него некоторые задания, например резервное копирование. Считайте эти атрибуты квалификаторами для тонкой настройки ролей. Хотя Capistrano предлагает ряд стандартных квалификаторов, вы можете также определить свои собственные.

Вот и все! Задав имя приложения, сообщив информацию о системе управления версиями и определив роли, вы закончили конфигурирование.

О сценарии spin

После успешного развертывания Capistrano попытается запустить (spin) ваше приложение. По умолчанию она ищет файл `./script/spin`, ожидая, что в нем находится сценарий запуска серверов приложений. Написать этот сценарий должны вы сами (в составе Rails он не поставляется). Проще всего обратиться к сценарию `spawner`, входящему в состав Rails, поскольку он знает, как запускать FCGI и Mongrel.

Узнать об инструменте `spawner` несложно, достаточно ввести в консоли команду `script/process/spawner --help`. Простейший сценарий `spin`, вызывающий `spawner` для запуска Mongrel, выглядит следующим образом[1]:

```
/deploy_to/current/script/process/spawner -p 8256 -i 2
```

Поместите этот код в качестве файла `./script/spin` в репозиторий, и после успешного развертывания Capistrano запустит два экземпляра Mongrel, которые будут прослушивать соответственно порты 8256 и 8257. У стандартного сценария `spawner` в Rails есть преимущество – он отслеживает идентификаторы процессов, а следовательно, доступны и другие стандартные сценарии Rails. В частности, это относится к сценарию `script/reaper`, который служит для перезапуска, мониторинга и останова серверов приложений.

Если вы решите не ограничиваться тесно интегрированным решением на базе сценария `spin`, возможно, потому что придумаете иной механизм управления фоновыми процессами или захотите воспользоваться сторонними сценариями запуска, например `mongrel_cluster`, надо будет переопределить стандартные задания развертывания. Ниже в этой главе мы покажем, как это сделать.

Подготовка машины развертывания

Итак, конфигурация по умолчанию есть, все предположения выполнены, и мы можем попросить Capistrano подготовить машину разверты-

[1] Хотя на сегодняшний день Mongrel считается оптимальным решением, сконфигурировать fastcgi ничуть не сложнее.

вания. Что это означает? Задание deploy:setup создает структуру каталогов для развертывания приложения:

```
$ cap deploy:setup
```

Структура каталога развертывания

Выполнив задание deploy:setup, зайдите по SSH на сервер, где производится развертывание, и посмотрите, какие там были созданы каталоги. По умолчанию начальным является каталог /var/www/apps/application_name, содержащий следующие подкаталоги:

```
releases
current
shared
shared/log
shared/system
shared/pids
```

Эта структура нуждается в некоторых пояснениях. Во время развертывания Capistrano выгружает (экспортирует) ваш проект из svn и помещает файлы в папку releases, причем *каждая версия сохраняется в отдельном каталоге, имя которого содержит текущую дату и время*. Если все прошло успешно, Capistrano создает ссылку с папки releases на каталог *application_name*/current, где должна находиться текущая версия развернутого веб-приложения.

Символические ссылки

После каждого развертывания Capistrano также создает следующие символические ссылки:

- *application_name*/shared/log ссылается на каталог log текущего проекта, так что протоколы не пропадают при смене версии;
- *application_name*/shared/pids ссылается на каталог tmp/pids текущего проекта;
- *application_name*/shared/system ссылается на каталог public/system текущего проекта. Здесь Capistrano хранит HTML-файлы, показывающие, что проект находится в режиме техобслуживания (см. cap deploy:web:disable/enable).

Проверка подготовленного окружения

Прежде чем приступать к развертыванию, можно выполнить задание deploy:check, которое проверит, что все предположения выполнены, и все кусочки заняли свои места:

```
cap --quiet deploy:check # quiet подавляет вывод информационных сообщений
```

Помимо стандартной проверки разрешений, наличия необходимых утилит (svn) и других требований, deploy:check также содержит сред-

ства верификации зависимостей, специфичных для конкретного приложения. Их можно объявить в файле deploy.rb, причем механизм работает как для локальных, так и для удаленных зависимостей:

```
depend :remote, :gem, "runt", ">= 0.3.0"
depend :remote, :directory, :writeable, /var/www/current/config
depend :remote, :command, "monit"
depend :local, :command, "svn"
```

Если deploy:check обнаружит отсутствующую зависимость, вы увидите примерно такое сообщение:

```
The following dependencies failed. Please check them and try again:
--> gem 'runt' >= 0.3.0 could not be found (my_deployment_machine)
```

Развертываем!

Если до этого момента в настройке сценария deploy.rb и подготовке удаленных машин все прошло успешно, можно выполнить само развертывание.

Первое развертывание приложения выполняется заданием cap deploy:cold, последующие – cap deploy. Единственное различие между ними заключается в том, что cap deploy сначала попытается остановить сервер, а при первом развертывании это не получится, так как сервер еще не был запущен:

```
$ cap deploy:cold # cold выполняет "svn co", запускает миграции, создает
                  # ссылку с этой версии на текущую и запускает серверы
```

В большинстве случаев вам будет предложено ввести пароль доступа к svn-серверу. Именно для таких ситуаций удобно заранее подготовить SSH-ключи на локальной и клиентской машине. Тогда аутентификация будет производиться автоматически по сертификатам, и не придется каждый раз вводить пароль.

Если все закончится хорошо, то никаких сообщений об ошибках вы не увидите, и броузер подтвердит, что приложение работает. В противном случае прочитайте подробное сообщение, проверьте все, снова пройдитесь по предположениям и начинайте сначала.

Переопределение предположений Capistrano

Итак, мы научились работать с Capistrano при условии выполнения стандартных предположений. А теперь посмотрим, когда приходится их модифицировать. В некоторых случаях достаточно задать дополнительные переменные Capistrano, а в других – переопределять существующие задания или подключаться к функциям обратного вызова. Иногда необходимо даже определять совсем новые задания.

Использование удаленной учетной записи пользователя

Чтобы использовать учетную запись не текущего зарегистрированного, а другого пользователя на удаленной машине, достаточно присвоить переменной `:user` имя удаленного пользователя:

```
set :user, "deploy"
```

Это простой случай, но все же отметим важный момент: Capistrano пытается максимально облегчить выход за рамки стандартных предположений. В следующем примере мы укажем систему управления версиями, отличную от Subversion.

Изменение системы управления версиями, используемой Capistrano

По умолчанию предполагается, что для управления версиями применяется система Subversion, но поддерживаются также Perforce, Bzr и Darcs:

```
set :scm, :subversion # default. :perforce, :bzr, :darcs
```

Можно изменить и стратегию развертывания. Более того, мы даже не рекомендуем оставлять умалчиваемую (`:checkout`), так как она очень неэффективна. Эти маленькие каталоги `.svn` пожирают место на диске, как бешеные! Попробуйте вместо этого задать режим `:export` — тогда для копирования вашего кода в каталог `release` будет вызываться команда `svn export`. Режим `:remote_cache` тоже работает быстро, так как делает копию последней версии, а затем выполняет `svn up`, чтобы получить самый свежий код. Но нам `:export` нравится больше:

```
set :deploy_via, :checkout # default
set :deploy_via, :export
set :deploy_via, :remote_cache # копирует из кэша, затем svn up
```

Работа без доступа к системе управления версиями с машины развертывания

Иногда из соображений безопасности у вас отсутствует (или вы не хотите давать) доступ к системе управления версиями (СУВ) с машины развертывания. Для таких ситуаций Capistrano предлагает задание `deploy_via :copy`. Стратегия `:copy` создает tar-архив проекта, сжимает его с помощью gzip и загружает по протоколу SFTP в каталог `release` на удаленной машине. В тех редких случаях, когда на локальной машине нет нужных исполняемых программ, вы можете сказать Capistrano, чтобы она пользовалась внутренней библиотекой zip, реализованной на Ruby:

```
set :deploy_via, :copy # локальная выгрузка из СУВ.
    # Cap вызывает tar/gzip, затем sftp на машину развертывания
```

```
set :copy_strategy, :export # changes deploy_via :copy to :export
    # вместо принимаемой по умолчания стратегии выгрузки из СУВ
set :copy_compression, :zip # если нет tar/gzip
    # Cap сама сожмет
```

Надо полагать, вы обратили внимание, что, хоть стандартные предположения Capistrano достаточно хороши – вам предоставляется немалая гибкость для задания альтернатив, не прибегая к написанию собственных заданий.

Если файл database.yml не хранится в репозитории СУВ

Некоторые разработчики по ряду причин, в том числе из соображений безопасности, не любят хранить конфигурационные файлы, например `database.yml`, в репозитории. Мы не рекомендуем так поступать, поскольку это усложняет распределенную разработку и требования, предъявляемые к локальной конфигурации. Однако знание того, как решить проблему отсутствия конфигурационных файлов в системе управления версиями, даст нам ценную возможность научиться выходу за пределы базовых навыков работы с Capistrano.

Отметим, что из следующих трех вариантов только один, на наш взгляд, действительно хорош. Впрочем, мы рассмотрим все три.

Вариант A: все равно храните в СУВ, но под другим именем

В этом случае вы добавляете в репозиторий файл с именем типа `production.database.yml`, но при развертывании автоматически переименовываете его в `database.yml`. Да, возможно, эта задача интересна, потому что вы вообще не хотите хранить в репозитории пароли. Тем не менее хотя бы некоторым разработчикам для Rails это решение может показаться допустимым. В конце концов более распространенная причина заключается в том, у каждого члена команды могут быть свои настройки соединения с локальной базой данных.

Плюсы. Простота. Не надо никакой конфигурации по умолчанию, достаточно создать дополнительное задание и поместить файл `production.database.yml` в репозиторий.

Минусы. В репозитории хранятся пароли в открытом виде, что может составлять серьезную проблему для некоторых организаций.

Для реализации этого варианта добавьте файл `production.database.yml` в репозиторий, включив в него параметры промышленной базы данных. Затем включите в файл `config/deploy.rb` определение задания, как показано в листинге 21.1.

Листинг 21.1. Копирование файла production.database.yml после обновления кода

```
task :after_update_code, :roles => :app do
  run "cp #{release_path}/config/production.database.yml
          #{release_path}/config/database.yml"
end
```

Задания с именем `:after_update_code` работают как обратные вызовы, активируемые после обновления кода в результате установки новой версии. Мы также продемонстрировали команду `run`, чтобы показать, насколько просто выполняются удаленные команды.

Вариант Б: храните файл database.yml с настройками для промышленной эксплуатации в папке shared/config

Это еще одно решение, но имейте в виду, что и его оптимальным не назовешь. Мы приводим его, чтобы продемонстрировать обратный вызов `:after_symlink`.

Плюсы. В репозитории не хранится имя пользователя и пароль.

Минусы. Конфигурационные файлы приходится вручную копировать на машину развертывания.

Для реализации этого варианта добавьте в файл `config/deploy.rb` следующее задание:

```
task :after_symlink, :roles => :app do
   run "cp #{shared_path}/config/database.yml
           #{release_path}/config/database.yml"
end

then...
  a. cap setup
  b. copy production-ready database.yml file to the shared/config
folder
  c. cap cold_deploy
```

Вариант В – самый лучший: автоматическая генерация database.yml

Этот вариант настолько лучше, что мы надеемся, что остальные два вы даже рассматривать не будете. Вам предстоит автоматически сгенерировать файл `database.yml` (как показано в листинге 21.2) и поместить его в папку `shared/config` на удаленной машине. А затем поставить ссылки на конфигурационную папку, относящуюся к текущей версии.

Плюсы. Легкость повторного использования, гибкость; пароли не хранятся в репозитории.

Минусы. Программировать немного сложнее, но это делается всего один раз[1].

Листинг 21.2. Создание файла database.yml в общей папке по шаблону

```
require 'erb'

before "deploy:setup", :db
after  "deploy:update_code", "db:symlink"
```

[1] *http://shanesbrain.net/articles/2007/05/30/database-yml-management-with-capistrano-2-0.*

```
namespace :db do
  desc "Создание database.yml в общей папке"
  task :default do
    db_config = ERB.new <<-EOF
      base: &base
      adapter: mysql
      socket: /tmp/mysql.sock
      username: #{user}
      password: #{password}

    development:
      database: #{application}_dev
      <<: *base

    test:
      database: #{application}_test
      <<: *base

    production:
      database: #{application}_prod
      <<: *base
    EOF

    run "mkdir -p #{shared_path}/config"
    put db_config.result, "#{shared_path}/config/database.yml"
  end

  desc "Создание символической ссылки на database.yml"
  task :symlink do
    run "ln -nfs #{shared_path}/config/database.yml
                 #{release_path}/config/database.yml"
  end
end
```

А затем с помощью нового yml-файла выполните следующие команды:

```
$ cap deploy:setup
$ cap deploy:update
```

Если миграции не проходят без ошибок

Рецепт Capistrano `deploy:migrations` ожидает, что в файле `database.yml` находятся настройки промышленной базы данных. Предполагая это, он выполняет развертывание по обычной схеме, а потом прогоняет ожидающие миграции. К сожалению, нередко бывает, что при прогоне всего набора миграций не обходится без ошибок.

Одно из возможных решений – создать задание, подготавливающее вашу базу данных с помощью сценария `db/schema.rb`, который (прибегнув к Migrations API) сохраняет версию схемы базы данных (описание на языке DDL) после самой последней миграции. Задание Capistrano, показанное в листинге 21.3, загружает эту схему.

Листинг 21.3. Удаленная загрузка схемы базы данных с помощью сценария schema.rb

```
namespace :deploy do
  desc "запускает 'rake db:schema:load' для текущей версии"
  task :load_schema, :roles => :db do

    rake = fetch(:rake, "rake")
    rails_env = fetch(:rails_env, "production")
    run "cd #{current_release}; #{rake} RAILS_ENV=#{rails_env}
db:schema:load"
  end
end
```

Запустите это задание в два шага:

1. `cap deploy:cold`.
2. `cap deploy:load_schema`.

Полезные рецепты Capistrano

Одна из самых ярких сторон системы Capistrano – это легкость создания собственных рецептов. Но сначала давайте разберемся с переменными.

Переменные и их область видимости

Есть два способа задания переменных. Первый – из определений рецептов (с ними мы уже встречались). Второй – из командной строки.

В командной строке значения переменных можно задать с помощью флага `-s` или `-S`. Команда `cap -s foo=bar` эквивалентна выполнению предложения `set :foo, "bar"` *после* загрузки всех рецептов, а команда `cap -S foo=bar` – выполнению того же предложения *до* их загрузки.

Как и в Rake, переменные можно задавать в окружении ОС. Поскольку в Windows это несколько сложнее, чем в Unix, то оптимальное кроссплатформенное решение – пользоваться флагами в командной строке Capistrano.

По возможности я предпочитаю создавать задания Capistrano, которые сами устанавливают переменные, а не зависят от сценариев оболочки или задания флагов в командной строке. Тем самым вся информация о развертывании оказывается сосредоточенной внутри Capistrano, а не размазывается по окружению.

Нам еще надо понять, какова область видимости переменных Capistrano. Являются ли они локальными для одного задания? Или локальными в пространстве имен? Или глобальными для Capistrano в целом? В документации по Capistrano ответ на эти вопросы отсутствует, поэтому проведем небольшое исследование и напишем несколько заданий (лис-

тинг 21.4). Начнем с создания двух пространств имен (`:one` и `:two`) и заведем в них переменные с одним и тем же именем (`:var_one`). Затем присвоим этим переменным значения и прочитаем их из третьего пространства имен. Результат должен многое сказать нам о правилах областей видимости в Capistrano.

Листинг 21.4. Исследование областей видимости переменных в Capistrano

```
namespace :one do
  task :default do
    set :var_one, "var_one value"
  end
end

namespace :two do
  task :default do
    set :var_one, "var_two value"
  end
end

namespace :three do
  task :default do
    puts "!!!! one.var_one == #{one.var_one}"
    puts "!!!! global name space var_one == #{var_one}"

    two.default

    puts "!!!! one.var_one == #{one.var_one}"
    puts "!!!! two.var_one == #{two.var_one}"
    puts "!!!! global name space var_one == #{var_one}"
  end
end

before "deploy:update", :one
before "deploy:update", :two
after  "deploy:update", :three

$cap deploy:update
```

Данная программа выводит на консоль следующее:

```
  * executing 'three' # run one
!!!! one.var_one == var_one value
!!!! global name space var_one == var_one value

  * executing 'two' # run two
!!!! one.var_one == var_two value
!!!! two.var_one == var_two value
!!!! global name space var_one == var_two value
```

И что это означает? Во-первых, видно, что переменные one.var_one и var_one (глобальная) имеют одно и то же значение; похоже, что это одна и та же переменная. При втором прогоне мы вызываем задание two.default. Установив лишь переменную two.var_one, мы убеждаемся, что никакого локального пространства имен для переменных не существует – все они глобальны.

Возможно, вы обратили внимание на два странных обращения к two.default. Так вызываются задания, погруженные в пространство имен. В случае задания default мы указываем имя явно (одно лишь имя two не разрешится, как в случае сокращенного синтаксиса пространств имен :two).

Короче говоря, мы доказали, что область видимости переменных не ограничена пространством имен. При ссылке с указанием пространства имен все равно возвращается глобальное значение, и это может запутать как вас, так и тех, кому предстоит сопровождать ваш код. Однако имейте в виду, что переменные с ограниченной областью видимости планируется ввести в следующей версии Capistrano.

Упражнение 1. Промежуточный сервер

Capistrano допускает один весьма любопытный трюк: инициализировать начальную конфигурацию для заданий. Польза такой предварительной инициализации состоит в том, что она позволяет подготовить окружение, в котором мы будем производить развертывание.

Сделать это можно двумя способами. Первый способ – задать начальное значение с помощью флага -S:

```
$ cap -S app_server=the_rails_way.com,secure_ssh_port=8256 deploy
```

При запуске задания deploy уже установлены параметры app_server и secure_ssh_port. Но возможности этой стратегии быстро исчерпываются по мере добавления новых параметров. Так что же, заводить специальный сценарий оболочки для задания таких параметров? А вот и нет! Capistrano может стать вашим могущественным помощником, особенно если логика развертывания не является распределенной.

Второй способ кодирования заданий, гораздо лучше согласующийся принципом DRY, заключается в определении каждого из промежуточных окружений, чтобы можно было написать что-то вроде cap production deploy, позволив заданию production подготовить необходимые переменные (см. листинг 21.5).

Листинг 21.5. Задания для промышленного и промежуточного окружения

```
desc "развертывание в промышленном окружении"
task :production do
   set :tag, "release-1.0" unless variables[:tag]
```

```
    set :domain, "the-rails-way.com"
    set :repository, "http://svn.nnovation.ca/svn/the-railsway/
tags/#{tag}"
    set :rails_env, "production"
    set :app_server, "the-rails-way.net"
    set :secure_ssh_port, 8256

    role :app, "#{app_server}:#{secure_ssh_port}"
    role :web, "#{app_server}:#{secure_ssh_port}"
    role :db, "#{app_server}:#{secure_ssh_port}", :primary => true
end

desc " развертывание в промежуточном окружении "
task :staging do
    set :domain, "staging.the-rails-way.com"
    set :repository, "http://svn.nnovation.ca/svn/the-rails-way/trunk"
    set :rails_env, "development"
    set :app_server, "staging.the-rails-way.com"
    set :secure_ssh_port, 8256

    role :app, "#{app_server}:#{secure_ssh_port}"
    role :web, "#{app_server}:#{secure_ssh_port}"
    role :db, "#{app_server}:#{secure_ssh_port}", :primary => true
end
```

Таким образом, мы можем очень лаконично настраивать окружение развертывания, не прибегая к параметрам в командной строке:

```
$ cap staging deploy # основной ствол на промежуточный сервер
$ cap production deploy # tags/release-1.0 на промышленный сервер
```

Не знаю, как вам, но мне эти повторяющиеся присваивания ролей не нравятся. Нельзя ли вынести их, быть может, поместив под определениями обоих заданий. Мы попробовали и получили важный урок: Capistrano говорит, что не понимает, что такое `app_server` и `secure_ssh_port`. В чем тут дело? В областях видимости? Или в порядке выполнения?

И в том, и в другом. Когда присваивание ролей выносится из блоков `do..end` в тело главного сценария, изменяется порядок выполнения. Теперь эти строки выполняются раньше, чем код внутри определения заданий. Следовательно, в данном случае это произойдет до присвоения значения переменной `app_server`.

На наше счастье, решение довольно простое и позволяется организовать код более элегантно. Мы воспользуемся поддерживаемым Capistrano вариантом одного из способов рефакторинга – *выделением метода*, только оформим общий код не в виде метода, а в виде еще одного задания.

Определим задание `:finalize_staging_init` и вызовем его в конце каждого из заданий `staging` и `production`:

```
task :staging do
   set :domain, "staging.the-rails-way.com"
   set :repository, "http://svn.nnovation.ca/svn/the-rails-way/trunk"
   set :rails_env, "development"
   set :app_server, "staging.the-rails-way.com"
   set :secure_ssh_port, 8256

   finalize_staging_init
end

task :production do
   set :tag, "release-1.0" unless variables[:tag]
   set :domain, "the-rails-way.com"
   set :repository, "http://svn.nnovation.ca/svn/the-railsway/
tags/#{tag}"
   set :rails_env, "production"
   set :app_server, "the-rails-way.net"
   set :secure_ssh_port, 8256

   finalize_staging_init
end

task :finalize_staging_init do
   role :app, "#{app_server}:#{secure_ssh_port}"
   role :web, "#{app_server}:#{secure_ssh_port}"
   role :db, "#{app_server}:#{secure_ssh_port}", :primary => true
end
```

Упражнение 2. Управление другими службами

В типичном сценарии развертывания Rails участвуют кластер серверов Mongrel и, возможно, некоторые дополнительные процессы, например `backgroundrb`, `memcache` и поисковики.

Готовые задания `deploy:start`, `:stop` и `:start` настроены на один экземпляр Mongrel, находящийся за Apache. В этом упражнении (листинг 21.6) мы переопределим задания по умолчанию, добавив управление кластером Mongrel и запуск процесса `BackgrounDRb`. Заметим, однако, что управление Apache здесь не осуществляется, поскольку я редко останавливаю и запускаю его. Сейчас вы уже достаточно знаете о Capistrano, чтобы разобраться в этом рецепте и при необходимости включить в него поддержку Apache.

Листинг 21.6. Более сложное задание развертывания

```
namespace :deploy do

  desc "Перезапустить кластер Mongrel и backgroundrb"
  task :restart, :roles => :app do
    stop
    start
  end
```

```
desc "Запустить кластер Mongrel и backgroundrb"
task :start, :roles => :app do
  start_mongrel
  start_backgroundrb
end

desc "Остановить кластер Mongrel и backgroundrb"
task :stop, :roles => :app do
  stop_mongrel
  stop_backgroundrb
end

desc "Запустить Mongrel"
task :start_mongrel, :roles => :app do
  begin
    run "mongrel_cluster_ctl start -c #{app_mongrel_config_dir}"
  rescue RuntimeError => e
    puts e
    puts "Mongrel уже запущен. "
  end
end

desc "Остановить Mongrel"
task :stop_mongrel, :roles => :app do
  begin
    run "mongrel_cluster_ctl stop -c #{app_mongrel_config_dir}"
  rescue RuntimeError => e
     puts e
     puts "Mongrel уже остановлен. "
  end
end

desc "Запустить сервер backgroundrb"
task :start_backgroundrb , :roles => :app do
  begin
    puts "запускается brb в папке #{current_path}"
    run "cd #{current_path} && RAILS_ENV=#{rails_env} nohup
./script/backgroundrb start > /dev/null 2>&1"
  rescue RuntimeError => e
    puts e
    puts "Ошибка при запуске backgroundrb - уже работает?"
  end
end

desc "Остановить сервер backgroundrb"
  task :stop_backgroundrb , :roles => :app do
  begin
    puts "останавливается brb в папке #{current_path}"
    run "cd #{current_path} && ./script/backgroundrb stop"
```

```
    rescue RuntimeError => e
      puts e
      puts "Backgroundrb уже остановлен."
    end
  end
end
```

Развертывание на нескольких серверах

На одном сервере все отлично работает. Но жизнь не стоит на месте, бизнес процветает, и вы принимаете решение построить *кластер* серверов. На одних будет работать ваше приложение, на других – веб-сервер, отдельный сервер будет выделен для асинхронной обработки и т. д. Процветание бизнеса – это прекрасно, но традиционная процедура развертывания в этом случае – не сахар. Развернуть на 10 машин? Гм, может быть, сказаться больным в тот день, на который намечено это мероприятие?

Только не с Capistrano! Эта система с самого начала задумывалась для поддержки развертывания на нескольких серверах. Джеймис хотел, чтобы развернуть приложение на 100 машинах было так же просто, как на одной. Многие даже считают, что именно в этом отношении достоинства Capistrano проявляются в полной мере.

Capistrano настолько хорошо справляется с несколькими серверами, что вы даже не заметите разницы ни при запуске из командной строки, ни при написании заданий. Секрет заключается в команде `role`[1], которая позволяет связать с заданием одну или несколько машин развертывания. Если запустить многомашинное задание, оно будет выполняться на всех серверах, приписанных соответствующей роли, причем параллельно:

```
role :app, "uno.booming-biz.com", "dos.booming-biz.com"
role :db, "kahuna.booming-biz.com", :primary => true
```

Если включить в роль второй (или третий) сервер, то на нем автоматически начнут выполняться все связанные с этой ролью задания, то есть на новом сервере будут выполняться все задания, для которых явно указана роль `:app`, а также те, для которых не указано никакой роли:

```
namespace :monitor
  task :exceptions :roles => :app do
    run "grep 'Exception' #{shared_path}/log/*.log"
  end
end
```

[1] Можно также указать для задания квалификатор `:host`, но распределение хостов по ролям здорово упростит вам жизнь, когда придется добавлять новые серверы.

Так, задание `cap monitor:exceptions` будет выполняться на всех машинах, принадлежащих роли `:app`. Capistrano запустит команду `grep` для одновременного просмотра всех протоколов и выведет объединенные результаты на ваш терминал.

Приверженность Capistrano принципу DRY означает, что физическую процедуру развертывания можно масштабировать, никак не затрагивая правил развертывания и с минимальным изменением конфигурации развертывания.

А что можно сказать о самом процессе развертывания? Чем больше машин, тем серьезнее последствия ошибок кодирования и неожиданных ситуаций, разве не так? Не обязательно, поскольку в Capistrano встроен механизм, который обычно ассоциируется с базами данных, а не с системами развертывания. Это транзакции!

Транзакции

Даже на одной машине нелегко устранить последствия неудачно завершившейся или не доведенной до конца процедуры развертывания. Что уж говорить о нескольких машинах, когда на каждой может возникнуть своя ошибка! Capistrano наделяет задания транзакционной инфраструктурой, которая защищает основные команды развертывания. Есть в ней и уникальные обработчики `on_rollback`, призванные обеспечить восстановление после неудачного развертывания с минимальным ущербом.

Взгляните, например, на код заданий `:update_code` и `:symlink`. В обоих имеются блоки `on_rollback`, которые при необходимости откатывают соответствующие действия. Здесь прослеживается аналогия с методами `up` и `down` миграций `ActiveRecord`:

```
namespace :deploy do
task :update do
  transaction do
    update_code
    symlink
  end
end

task :update_code do
  on_rollback { run "rm -rf #{release_path}" }
  strategy.deploy!
  finalize_update
end

task :symlink, :except => { :no_release => true } do
  on_rollback do
    run "rm -f #{current_path}"
    run "ln -s #{previous_release} #{current_path}; true"
```

```
    end
    run "rm -f #{current_path} && ln -s #{release_path} #{current_path}"
    end
  end
```

Этот пример взят из исходных текстов Capistrano[1] – в задании `:update` для обновления развернутой системы применяется стратегия выгрузки из СУВ, после чего создаются символические ссылки с новой версии на папку `./current`.

Но что если ошибка произошла в задании `strategy.deploy!` (быть может, из-за невозможности соединения с сервером Subversion)? Продолжит ли задание `deploy` создавать символические ссылки? А вдруг выгрузка прошла успешно, но не удалось создать ссылки? И в том, и в другом случае приложение останется в полуразобранном состоянии. Проблема осложнится еще больше, если на одной машине развертывание прошло успешно, а на другой «грохнулось», – мы даже не сразу заметим, что случилось.

```
task :update_code do
  on_rollback { run "rm -rf #{release_path}" }
  strategy.deploy!
  finalize_update
end
```

Чтобы свести к минимуму последствия ошибок и обработать как можно больше причин сбоя, задание `:update` погружено в транзакцию. Если произойдет сбой, будут вызваны все блоки `on_rollback` – в обратном порядке. Поэтому блоки `on_rollback` следует проектировать так, чтобы они алгоритмически откатывали результат выполнения одной из операций задания.

Например, показанный выше блок `on_rollback` удаляет все файлы, которые могли быть созданы заданиями `strategy.deploy!` и `finalize_update`, возвращая приложение в корректное состояние.

Транзакционная система, применяемая в Capistrano, не похожа ни на что, с чем вам доводилось сталкиваться прежде. Например, она не хранит локальные или удаленные изменения объектов. Простой, но эффективный механизм позволяет *вам* управлять откатом. Следует также отметить, что Capistrano не включает миграции в состав транзакции – транзакционность языка DDL поддерживается не всеми СУБД[2], поэтому откатить ошибочно завершившуюся миграцию очень сложно.

---

[1] Найдите исходный текст с помощью команды `gem environment` и покопайтесь в нем. Это прекрасный способ обучения.

[2] MySQL точно не поддерживает, но, насколько я понимаю, Postgres умеет включать предложения DDL в транзакции.

Доступ к машинам развертывания через прокси-серверы

На практике развертывание на серверах приложений часто производится через безопасные прокси-серверы и брандмауэры. Поэтому установить SSH-сеанс непосредственно с машиной развертывания не всегда возможно. Но пусть это вас не останавливает, ведь Capistrano поддерживает «прокси-доступ» к машинам развертывания с помощью параметра `:gateway`:

```
set :gateway, 'gateway.booming-biz.com'
role :app, "192.168.1.100", "192.168.1.101"
role :db, "192.168.1.200", :primary => true
```

При наличии параметра `:gateway` все запросы, адресованные машинам, принадлежащим некоторой роли, направляются в защищенный туннель, проходящий через указанные компьютеры-шлюзы. Capistrano предполагает, что эти машины напрямую недоступны, поэтому нужно сначала организовать соединение с `gateway.booming-biz.com`, а оттуда уже открывать SSH-туннели.

Это волшебство доступно благодаря механизму переадресации портов[1]. Вам нужно лишь гарантировать доступность машин, принадлежащих данной роли, по TCP/IP, а больше от вас для поддержки шлюза почти ничего не требуется. Если вы пользуетесь для аутентификации паролями, то вообще ничего не требуется – просто будет предложено ввести пароль. Если же применяетеся PKI, нужно будет добавить открытый ключ сервера-шлюза на недоступные извне ролевые серверы.

Заключение

Эта глава представляет собой краткий курс по использованию системы Capistrano для автоматизации развертывания в Rails. Надеюсь, что мне удалось убедить вас начать использование Capistrano в качестве личного системного администратора. Особенно с учетом присущего ей умения надежно автоматизировать выполнение заданий и параллельно выполнять их как на одном, так и на десятках удаленных серверов.

[1] Технические детали изложены в статье Джеймиса Бака по адресу *http://weblog.jamisbuck.org/2006/9/26/insidecapistrano-the-gateway-implementation*.

22

Фоновая обработка

Ожидание railsapplication.com...

Сообщение в строке состояния
броузера пользователя

У интернет-пользователя есть только одна возможность понять, что ваше приложение работает, – если оно отвечает на запросы. Классический пример – обработка платежа кредитной картой. Какой сайт вы предпочтете: тот, что говорит «Транзакция обрабатывается» и показывает какую-нибудь подходящую анимацию, или тот, что выдает пустую страницу?

Помимо такого рода аспектов, касающихся *удобства пользователя*, к приложению могут предъявляться требования, которые просто невозможно удовлетворить за несколько секунд. Быть может, вы отвечаете за популярный сайт, позволяющий пользователям загружать видеофайлы и обмениваться ими с другими людьми. Тогда вам предстоит конвертировать различные виды видеоконтента в формат Flash. Ни один сервер, сколь угодно быстрый, не способен справиться с этой задачей, пока пользователь ожидает ответа в броузере.

Знакомые ситуации? Если да, то самое время подумать о том, как организовать в приложении фоновую обработку. В этой главе под словом *фоновая* мы понимаем все, что происходит за пределами обычного цикла запрос/ответ. Большинству разработчиков рано или поздно приходится проектировать и реализовывать тот или иной вид фоновой обработки. К счастью, в Rails и Ruby есть несколько библиотек для решения этой задачи, в частности:

- `script/runner` – сценарий, встроенный в Rails;
- DRb – библиотека распределенной обработки, написанная Масатоси Секи (Masatoshi Seki);
- BackgrounDRb – подключаемый модуль, который написал Эзра Зигмунтович (Ezra Zygmuntowicz) и поддерживает Скаар (Skaar);
- Daemons – упрощает создание постоянно работающих системных служб. Автор – Томас Юлингер (Thomas Uehlinger).

С помощью этих инструментов вы легко сможете добавить фоновую обработку в свое приложение Rails. В этой главе мы расскажем о каждом из них достаточно, чтобы вы могли решить, какое в наибольшей степени отвечает потребностям вашего приложения.

Сценарий script/runner

В состав Rails входит инструмент для запуска заданий независимо от цикла HTTP-обработки. Сценарий `runner` просто загружает стандартное окружение Rails и выполняет заданный код на Ruby. Он часто используется для решения следующих задач:

- пакетного импорта внешних данных;
- выполнения произвольного метода (класса), имеющегося в какой-то из ваших моделей;
- выполнения длительных вычислений, пакетной рассылка электронной почты и выполнения заданий по расписанию.

Любой ценой следует избегать применения `script/runner` для:

- обработки входящей почты;
- выполнения заданий, на которые уходит тем больше времени, чем объемнее база данных.

Приступаем к работе

Пусть, например, имеется модель Report. В ней есть метод класса `generate_rankings`, который следующим образом вызывается из командной строки:

```
$ ruby script/runner 'Report.generate_rankings'
```

Поскольку у нас имеется полномасштабный доступ к Rails, можно было бы даже воспользоваться методами поиска `ActiveRecord` для выборки данных из приложения[1]:

```
$ ruby script/runner 'User.find(:all).map(&:email).each { |e| \
puts "<#{e}>"}'
```

[1] Не забудьте экранировать символы, имеющие специальный смысл для оболочки.

```
<charles.quinn@highgroove.com>
<me@seebq.com>
<bill.gates@microsoft.com>
# ...
<obie@obiefernandez.com>
```

Этот пример демонстрирует, что у нас есть доступ к модели User, и мы можем выполнить произвольный код в Rails. В данном случае мы отобрали несколько почтовых адресов, на которые можем спамить, сколько душе угодно (шучу, конечно).

Несколько слов об использовании

При работе со сценарием script/runner нужно помнить о нескольких моментах. Если необходим промышленный режим, задайте флаг -e; по умолчанию загружаются параметры режима разработки. Если задать флаг -h, то script/runner расскажет о себе:

```
$ script/runner -h

Usage: script/runner [options] ('Some.ruby(code)' or a
filename)

  -e, --environment=name Specifies the environment for the runner
                         to operate in (test/development/production)

                         Default: development
```

Можно также указать runner в первой строке вашего сценария:

```
#!/usr/bin/env /path/to/script/runner
```

С помощью script/runner мы можем без труда программировать пакетные операции, запускаемые через cron или какой-либо иной системный планировщик.

Например, можно было бы каждые десять минут или ночью находить самый популярный или имеющий наивысший рейтинг продукт в вашем электронном магазине, а не выполнять накладный запрос к базе данных при каждом обращении:

```
$ script/runner -e production 'Product.calculate_top_ranking'
```

Запись в таблице crontab для запуска этого сценария могла бы выглядеть следующим образом:

```
0 */5 * * * root /usr/local/bin/ruby \
/apps/exampledotcom/current/script/runner -e production \
'Product.calculate_top_ranking'
```

Здесь сценарий запускается раз в пять часов и обновляет максимальные рейтинги в модели Product.

Замечания по поводу script/runner

Плюсы: проще трудно что-нибудь придумать, и не надо устанавливать дополнительных библиотек. На этом плюсы заканчиваются.

Минусы: процесс `script/runner` загружает окружение Rails целиком. Для некоторых, особенно краткосрочных, задач это ненужная трата ресурсов. Кроме того, ничто не мешает одновременно запустить несколько экземпляров одного и того же сценария. Последствия могут быть катастрофическими.

> **Говорит Уилсон...**
>
> Не обрабатывайте входящую почту с помощью `script/runner`. Это прямое приглашение к DoS-атаке.
>
> Пользуйтесь вместо этого программой Fetcher (или аналогичной):
>
> *http://slantwisedesign.com/rdoc/fetcher/*

Резюме: `script/runner` хорош для коротких задач, которые запускаются не слишком часто.

DRb

Возможно, вы знаете, что после некоторого конфигурирования DRb можно использовать в качестве контейнера сеансов для Rails, но безо всякой настройки он уже способен принимать простые TCP/IP-запросы и инициировать фоновую обработку.

> *DRb расшифровывается как Distributed Ruby (Распределенный Ruby). Эта библиотека позволяет отправлять и принимать запросы от удаленных объектов Ruby по протоколу TCP/IP. Что-то вроде RPC, CORBA или Java RMI? Пожалуй. Это простой ответ Ruby на все вышеперечисленные технологии.*
>
> Чед Фоулер, «Введение в DRb» (*http://chadfowler.com/ruby/drb.html*).

Простой DRb-сервер

Давайте создадим DRb-сервер, выполняющий простое вычисление. Мы запустим его на локальной машине (localhost), но не будем забывать, что он может работать на одном или более удаленных серверах с целью распределения нагрузки или обеспечения отказоустойчивости.

Создайте файл `distributed_server.rb` и поместите в него код, показанный в листинге 22.1.

Листинг 22.1. DRb-служба для выполнения простого вычисления

```
#!/usr/bin/env ruby -w
# DRb server

# load DRb
require 'drb'

class DistributedServer
  def perform_calculation(num)
    num * num
  end
end

DRb.start_service("druby://localhost:9000",
DistributedServer.new)
puts "Запускается DRb-сервер по адресу: #{DRb.uri}"
DRb.thread.join
```

Сделав этот файл исполняемым (команда `chmod +x` или эквивалентная ей), запустите его, чтобы он прослушивал порт 9000:

```
$./distributed_server
Запускается DRb-сервер по адресу: druby://localhost:9000
```

Использование DRb из Rails

Чтобы теперь обратиться к этому коду из Rails, мы можем затребовать библиотеку DRb в начале контроллера, где планируем использовать код:

```
require 'drb'
class MessagesController < ApplicationController
```

Для вызова метода объекта на распределенном сервере нужно добавить в контроллер примерно такое действие:

```
def calculation
  DRb.start_service
  drb_client = DRbObject.new(nil, 'druby://localhost:9000')
  @calculation = drb_client.perform_calculation(5)
end
```

Теперь у нас есть доступ к переменной экземпляра `@calculation`, которую вычислил распределенный сервер. Это тривиальный пример, но на нем хорошо видно, как просто можно выносить процессы на удаленный сервер.

Данный код выполняется в составе нормального цикла запрос/ответ. Rails ждет, пока метод DRb-сервера `perform_calculation` завершится, и только потом приступает к обработке шаблонов представлений и отправке данных пользовательскому агенту. Мы можем таким образом задействовать мощности нескольких серверов, но все равно это не сов-

сем то, что обычно понимается под фоновой обработкой. Чтобы завершить наше путешествие на темную сторону Луны, придется обернуть этот метод в некий код для управления заданиями.

Раз сообщить, что сделать это легко, но еще более приятно, что это уже сделано за вас. Читайте раздел «BackgrounDRb».

Замечания о DRb

Плюсы: DRb – часть стандартной библиотеки Ruby, поэтому ничего нового устанавливать не придется. Очень надежный механизм. Подходит для организации постоянно работающих процессов, которые должны быстро возвращать результат.

Минусы: DRb – сравнительно «низкоуровневая» библиотека, она не поддерживает ни управление заданиями, ни конфигурирование. Работая с ней напрямую, придется изобретать собственные соглашения о номерах портов, именах классов и т. д.

Применяйте DRb, когда хотите реализовать свой механизм балансирования нагрузки или если никакое другое решение не дает нужной вам точности управления.

Ресурсы

Более полную информацию о принципах работы DRb и о том, что на самом деле происходит в приведенных выше примерах, можно почерпнуть из следующих статей:

- «Введение в DRb» Эрика Ходеля по адресу *http://segment7.net/projects/ruby/drb/introduction.html*.
- «Введение в DRb» Чеда Фоулера по адресу *http://chadfowler.com/ruby/drb.html*.
- «Краткое введение в распределенный Ruby» Фрэнка Спихальски (Frank Spychalski) по адресу *http://amazingdevelopment.com/archives/2006/03/16/rails-and-distributed-ruby-in-a-nutshell/*.

BackgrounDRb

BackgrounDRb – это «сервер и планировщик заданий, написанный на Ruby». Адрес проекта – *http://backgroundrb.devjavu.com/*. Подключаемый модуль BackgrounDRb для Rails используется прежде всего для «вынесения длительных задач из цикла запрос/ответ»[1].

Помимо поддержки асинхронной фоновой обработки, BackgrounDRb (совместно с Ajax-кодом вашего приложения) применяется для обнов-

[1] *http://backgroundrb.rubyforge.org/*.

ления состояния и его индикаторов. Например, BackgrounDRb часто используется для реализации индикаторов прогресса во время загрузки больших файлов.

Ветвь 0.2.x BackgrounDRb была полностью переписана, и механизм создания и выполнения заданий в ней кардинально изменен. Теперь для обработки заданий применяется несколько процессов, а не один многопоточный процесс. Результаты хранятся в исполнителе `Result`, чтобы у каждого задания был свой процесс, где можно хранить результаты и извлекать их по мере надобности. У проекта имеется активное сообщество пользователей и открытый репозиторий исходных текстов с хорошим тестовым покрытием на базе RSpec.

Приступаем к работе

BackgrounDRb можно запускать автономно или как подключаемый к Rails модуль. Он зависит от двух gem-пакетов: Slave 1.1.0 (или старше) и Daemons 1.0.2 (или старше). Установка в существующее приложение Rails производится следующей командой:

```
svn co http://svn.devjavu.com/backgroundrb/tags/release-0.2.1
vendor/plugins/backgroundrb
```

Отметим, что команда

```
script/plugin install svn://rubyforge.org//var/svn/backgroundrb
```

устанавливает предыдущую версию BackgrounDRb с одним процессом, но *вам это не надо*. Мы рассмотрим только более новую версию 0.2.x, поскольку именно она продолжает разрабатываться и документироваться.

Убедитесь, что все тесты успешно проходят, для чего зайдите в каталог `plugin`. Вам потребуется gem-пакет RSpec:

```
$ rake
(in /Users/your_login/your_app/vendor/plugins/backgroundrb)
/usr/local/bin/ruby -Ilib:lib "test/backgroundrb_test.rb"
"test/scheduler_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader
Started
..................
Finished in 3.107323 seconds.

18 tests, 26 assertions, 0 failures, 0 errors
```

Если все тесты прошли, вернитесь в каталог `RAILS_ROOT` и выполните команду `rake backgroundrb:setup`, которая установит конфигурационные файлы и сценарии BackgrounDRb, а также создаст каталоги для заданий и исполнителей.

Конфигурирование

Принимаемый по умолчанию файл `config/backgroundrb.yml` выглядит следующим образом:

```
---
:rails_env: development
:host: localhost
:port: 2000
```

По умолчанию сервер BackgrounDRb работает в режиме разработки и прослушивает порт 2000 на машине localhost. Для перехода в режим эксплуатации необходимо изменить переменную `rails_env`. В официальной документации по BackgrounDRb, включенной в дистрибутив, имеется дополнительная информация.

Знакомство с принципами работы BackgrounDRb

В основе BackgrounDRb лежит класс `MiddleMan`, который создает *исполнителей* (workers), следит за ними и предоставляет доступ к сформированным ими результатам.

BackgrounDRb позволяет определять исполнителей, которые представляют собой классы, содержащие код, выполняемый в фоновом режиме. По умолчанию они хранятся в каталоге `lib/workers` проекта Rails.

Каждый исполнитель является подклассом одного из двух базовых классов, предоставляемых подключаемым модулем:

- `BackgrounDRb::Worker::Base` — простой исполнитель, нуждающийся лишь в минимальной настройке окружения;
- `BackgrounDRb::Worker::RailsBase` — исполнитель, которому нужен доступ к полностью сконфигурированному окружению Rails.

Исполнитель, являющийся подклассом `RailsBase`, потребляет больше ресурсов, поэтому, если вам не нужен доступ к моделям `ActiveRecord` или другим средствам Rails, попытайтесь обойтись простыми исполнителями.

Если исполнитель должен вернуть что-то приложению, мы можем вызвать его метод `results`. Результат выглядит как обычный объект `Hash`, но за ним стоит специальный исполнитель `Result`. В BackgrounDRb есть также метод `logger`, который позволяет заносить сообщения в протокол.

Каждый исполнитель должен определить метод `do_work`, принимающий единственный параметр `args`. BackgrounDRb автоматически вызывает его на этапе инициализации исполнителя. Обычно этот метод просто делегирует основную работу другим, определенным вами методам.

Использование класса MiddleMan

Давайте создадим в каталоге lib/workers исполнителя. Для создания базового класса воспользуемся предоставляемым генератором

```
$script/generate worker Counter
```

Добавим код, выполняющий 10 тыс. итераций, чтобы имитировать длительную задачу. В реальности это может быть обработка загруженного файла, конвертация изображения или генерация и отправка отчета. В листинге 22.2 весь этот код находится в методе do_work, но на практике следует придерживаться обычных принципов проектирования моделей и соответствующим образом разнести его по отдельным методам.

Листинг 22.2. Класс CounterWorker выполняет 10 тыс. итераций

```
class CounterWorker < BackgrounDRb::Worker::RailsBase
  def do_work(args)
    logger.info 'CounterWorker начинает работу'
    1.upto 10_000 do |x|
      results[:count] = x
      logger.info "Счетчик: #{x}"
    end
    logger.info 'Закончен отсчет до 10,000'
  end
end

CounterWorker.register
```

Подготовив исполнителя, можем запускать сервер BackgrounDRb:

```
$ ruby script/backgroundrb start
```

Проверим, что процессы BackgrounDRb запущены, выполнив команду ps[1]:

```
$ps aux | grep background
you 617 0.6 -0.2 3628 ?? R 4:20PM 0:00.23
backgroundrb
you 618 0.0 -0.7 14640 ?? S 4:20PM 0:00.10
backgroundrb_logger
you 619 0.0 -0.7 14572 ?? S 4:20PM 0:00.09
backgroundrb_results
```

Теперь можно обратиться к исполнителю из действия контроллера. Метод new_worker класса MiddleMan создает нового исполнителя и возвращает ключ, который позволит сослаться на него позже.

Ниже создается экземпляр CounterWorker, и его ключ сохраняется в сеансе:

[1] В Windows для этой цели можно воспользоваться командой tasklist.

```
def start_counting
  session[:key] = MiddleMan.new_worker(:class =>
:counter_worker)
  redirect_to :action => 'check_counter'
end
```

Пойдем дальше и создадим еще одно действие, где будет проверяться состояние исполнителя. Чтобы получить доступ к работающему исполнителю, понадобится сохраненный ключ, и тогда мы сможем обратиться к методу `results` за текущим значением счетчика:

```
def check_counter
  count_worker = MiddleMan.worker(session[:key])
  @count = count_worker.results[:count]
end
```

Соответствующее действию `check_counter` представление можно сделать совеем простым:

```
<p>Пока считаем. Текущее значение <%= @count %>.</p>
```

Внутри действия `start_counting` метод `new_worker` сразу же вызывает метод `do_work` из нашего класса `CounterWorker`. Это неблокирующий вызов, поэтому приложение спокойно продолжает работать и переадресует нас на указанный URL, пока исполнитель продолжает считать.

Если щелкнуть по кнопке Refresh (Обновить), попросив действие `check_counter` перезагрузить результаты работы исполнителя, мы увидим, что переменная `@count` увеличивается, по мере того как исполнитель неуклонно приближается к завершению работы.

Подводные камни

К сожалению, при любом изменении исполнителей BackgrounDRb приходится перезапускать. Исполнители загружаются один раз и потом кэшируются, как модели `ActiveRecord` в режиме эксплуатации.

Если вы получаете сообщение об ошибке вида:

```
/usr/local/lib/ruby/site_ruby/1.8/rubygems/custom_require.rb:
27:in `gem_original_require': no such file to load – slave
(LoadError)
```

вспомните, что BackgrounDRb зависит от gem-пакетов `slave` и `daemons`.

Если процесс `backgroundrb` завершается или умирает, необходимо стереть файлы, содержащие идентификаторы процессов. О том, что этого не произошло, вы узнаете, когда при следующей попытке запустить службу получите сообщение:

```
ERROR: there is already one or more instance(s) of the program
running
```

Для удаления файлов `log/backgroundrb.pid` и `log/backgroundrb.ppid` можно воспользоваться встроенной командой `zap`:

```
$ script/backgroundrb zap
```

BackgrounDRb должен нормально запуститься после удаления старых файлов.

Замечания о BackGrounDRb

Плюсы:

- обеспечивает управление заданиями и асинхронную работу без какой-либо настройки;
- широко распространен, в Сети есть много примеров;
- оптимален для заданий «основанных на событиях», например, возникающих всякий раз, как пользователь инициирует выполнение некоторого действия.

Минусы:

- текущая версия считается сопровождающими программистами «экспериментальной». По мере эволюции API, возможно, придется изменить код исполнителей или действий;
- поддержка запланированных заданий добавлена недавно и может быть не так стабильна, как прочие части кода;
- некоторые конфигурационные параметры «зашиты», поэтому их трудно настроить, если промышленное окружение необычно.

Учитывая все за и против, следует сделать вывод, что BackgrounDRb отлично приспособлен для заданий, которые инициируются из действий контроллера или обратного вызова модели.

Daemons

На сайте *http://daemons.rubyforge.org/* представлена отличная библиотека Ruby, которая позволяет «демонизировать» ваш сценарий, сохранив простоту управления и сопровождения.

Порядок применения

В листинге 22.3 приведен пример простого сценария, показывающего, как можно использовать библиотеку `daemons` для запуска задания по расписанию.

Листинг 22.3. Простой пример использования библиотеки Daemons для фонового обновления RSS-каналов

```
require 'daemons'

class BackgroundTasks
  include Singleton
```

```
  def update_rss_feeds
    loop do
      Feed.update_all
      sleep 10
    end
  end
end

Daemons.run_proc('BackgroundTasks') do
  BackgroundTasks.instance.update_rss_feeds
end
```

Здесь определено простое задание `update_rss_feeds`, которое выполняется в цикле. Если сохранить этот код в файле `background_tasks.rb` и запустить его следующим образом:

```
script/runner background_tasks.rb
```

вы увидите все параметры, поддерживаемые библиотекой `daemons`:

```
Usage: BackgroundTasks <command> <options> -- <application
options>

* где <command> может принимать значения:
start    запустить экземпляр приложения
stop     остановить все экземпляры приложения
restart  остановить, а потом перезапустить все экземпляры
run      запустить приложение и остаться в приоритетном режиме
zap      перевести приложение в состояние «остановлено»

* а <options> - сочетание следующих флагов:

-t, --ontop    Остаться в приоритетном режиме (не становиться демоном)
-f, --force    Принудительно выполнить операцию

Стандартные флаги:
-h, --help     Показать это сообщение
--version      Показать номер версии
```

Фоновым процессом можно управлять с помощью простых команд.

Библиотека Daemon гарантирует также, что в любой момент времени исполняется только один экземпляр вашего задания. Это устраняет необходимость в логике управления, которая обычно присутствует в сценариях, запускаемых через `script/runner` или `cron`.

Введение в потоки

В предыдущем примере было показано, какие средства управления предоставляет библиотека Daemons. Однако сейчас этот пример почти ничего не делает. Давайте модифицируем его, чтобы он еще и забирал почту с внешнего сервера (листинг 22.4). Поскольку для получения почты необходима сеть, мы воспользуемся потоками, чтобы успеть сделать больше работы за меньшее время.

Листинг 22.4. Многопоточный получатель почты

```
require 'thread'
require 'daemons'

class BackgroundTasks
  include Singleton

  def initialize
    ActiveRecord::Base.allow_concurrency = true
  end

  def run
    threads = []
    [:update_rss_feeds, :update_emails].each do |task|
      threads << Thread.new do
        self.send task
      end
    end
    threads.each {|t| t.join }
  end

  protected

  def update_rss_feeds
    loop do
      Feed.update_all
      sleep 10
    end
  end

  def update_emails
    loop do
      User.find(:all, :conditions => "email IS NOT NULL").each do
|user|
        user.fetch_emails
      end
      sleep 60
    end
  end
end

Daemons.run_proc('BackgroundTasks') do
  BackgroundTasks.instance.start
end
```

Обратите особое внимание на добавленную в метод `initialize` строчку:

```
ActiveRecord::Base.allow_concurrency = true
```

Это критически важно для одновременного использования `ActiveRecord` в нескольких потоках. Каждому потоку теперь выделяется отдельное соединение с базой данных. Если забыть об этом шаге, то неминуема порча данных и прочие ужасы. *Вас предупредили!*

> **Говорит Уилсон...**
>
> Если в коде, где встречается параллельное выполнение `ActiveRecord`, есть ошибки, вам, возможно, удастся лицезреть неприглядную картину нехватки соединений с базой данных. Следует быть очень осторожным при обработке исключений в многопоточном коде.
>
> Не исключено, что вам захочется вызвать в цикле обработки метод `ActiveRecord::Base.verify_active_connections!`, чтобы освободить «застрявшие» соединения. Это довольно накладный метод, но, если вы включаете режим параллельной работы `ActiveRecord`, то он становится немаловажным.

В только что написанном нами демоне поддержка планирования тривиальна. Вашему приложению может потребоваться нечто большее, чем просто `sleep 60`. В таком случае обратите внимание на не вполне удачно названную библиотеку OpenWFEru по адресу *http://openwferu.rubyforge.org/scheduler.html*, которая предоставляет различные механизмы планирования.

Замечания о библиотеке Daemons

Плюсы: библиотека Daemons – самый *экономичный* способ реализации фонового процесса, который должен работать непрерывно. Она предоставляет точный контроль того, какие библиотеки загружать и какие параметры конфигурировать.

Daemons легко управлять с помощью таких средств мониторинга, как `monit` (*http://www.tildeslash.com/monit/*).

Минусы: настройка библиотеки Daemons не до такой степени автоматизирована, как в случае BackgrounDRb, и не так проста, как для `script/runner` (программисты-фундаменталисты могут вообще испугаться работы с Daemons).

Подумайте о применении Daemons, когда вам нужно, чтобы какой-то процесс работал постоянно.

Заключение

В последней главе книги мы рассмотрели вопрос о расширении поведения Rails за пределы обычного цикла обработки запросов – вопрос о работе в *фоновом режиме*. Это обширная тема, и мы лишь поверхностно затронули наиболее очевидные ее аспекты.

A

Справочник по ActiveSupport API

ActiveSupport — это входящая в Rails библиотека, содержащая вспомогательные классы и расширения встроенных библиотек Ruby. Сама по себе она обычно не привлекает внимания, можно даже назвать ее вторым составом ансамбля Rails.

Однако тот факт, что ActiveSupport остается в тени, не уменьшает ее значения в повседневном программировании в Rails. Чтобы эта книга могла быть вашим помощником во всех ситуациях, мы включили полное справочное руководство по ActiveSupport API, снабдив его примерами из реальной практики и комментариями.

Прямые расширения классов и модулей Ruby помещены под заголовками, соответствующими имени этого класса или модуля. Расширения с помощью модулей-примесей озаглавлены в соответствии с именем модуля в ActiveSupport.

Примечание

Этот справочник был подготовлен на основе ревизии 7360 «острия Rails» (до выхода Rails 2.0). Некоторые методы были сочтены неуместными и опущены, поскольку устарели и сейчас не используются.

Array

Rails добавляет непосредственно в класс Array только один метод: blank?.

Открытые методы экземпляра

blank?

Возвращает `true`, если массив пуст.

Array::Conversions (в ActiveSupport::CoreExtensions)

Предоставляет методы для преобразования массивов Ruby в другие форматы.

Открытые методы экземпляра

to_formatted_s(format = :default)

Поддерживается два формата: `:default` и `:db`.

Формат `:default` делегирует работу обычному методу массива `to_s`, который конкатенирует содержимое в одну строку.

Гораздо интереснее значение `:db` – в этом режиме возвращается `"null"`, если массив пуст, а в противном случае поля `id` его элементов конкатенируются в строку с разделителями-запятыми:

```
collect { |element| element.id }.join(",")
```

to_param

Преобразует строковые элементы массива в строку, разделяя их знаками косой черты (применяется для генерации путей в URL):

```
>> ["riding","high","and","I","want","to","make"].to_param
=> "riding/high/and/I/want/to/make"
```

to_sentence(options = {})

Преобразует массив в предложение, которое состоит из слов, разделенных запятыми, где последнему слову предшествует слово-союз:

```
>> %w(alcohol tobacco firearms).to_sentence
=> "alcohol, tobacco, and firearms"
```

В хеше `options` можно передать следующие параметры:

- `:connector` – слово-союз, отделяющее последнее слово в строке от остальных для массивов, содержащих по крайней мере два элемента (по умолчанию `and`);
- `:skip_last_comma` – задайте `true`, если нужно вернуть «a, b and c», а не «a, b, and c».

to_xml(options = {}) {|xml if block_given?| ...}

В главе 15 «XML м ActiveResource» отмечалось, что метод `to_xml` из класса `Array` можно использовать для создания XML-документа – достаточно лишь поочередно вызвать его для каждого элемента массива и обернуть результат в объемлющий элемент.

Все элементы массива должны отвечать на сообщение `to_xml`:

```
>> ["riding","high"].to_xml
RuntimeError: Not all elements respond to to_xml
```

В следующем примере необязательному блоку передается объект `Buider`, чтобы в конец сгенерированного фрагмента XML можно было вставить произвольную разметку, которая станет последним потомком объемлющего элемента. Программа:

```
{:foo => "foo", :bar => :bar}.to_xml do |xml|
  xml.did_it "again"
end
```

выводит следующий XML-документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<hash>
  <bar:bar/>
  <foo>foo</foo>
  <did_it>again</did_it>
</hash>
```

Метод `to_xml` принимает следующие параметры в хеше `options`:

- `:builder` – по умолчанию новый экземпляр класса `Builder::XmlMarkup`. Задавайте явно, если вызов `to_xml` для данного массива – часть более крупной процедуры конструирования XML-документа;
- `:children` – задает имя тега для элементов. По умолчанию имя, определенное параметром `:root`, в единственном числе;
- `:dasherize` – нужно ли преобразовывать подчерки в дефисы в именах тегов (по умолчанию `true`);
- `:indent` – величина отступа в сгенерированном XML (по умолчанию два пробела);
- `:root` – имя тега объемлющего элемента. Если параметр `:root` не задан и все элементы массива принадлежат одному и тому же классу, то по умолчанию используется множественная форма имени класса первого элемента, в котором слова разделены дефисами. В противном случае корневой элемент будет называться `records`;
- `:skip_instruct` – нужно ли включать в сгенерированный XML команду обработки путем вызова метода `instruct!` объекта `Builder`;
- `:skip_types` – нужно ли включать атрибут `type="array"` для объемлющего элемента.

Array::ExtractOptions (в ActiveSupport::CoreExtensions)

Предоставляет метод извлечения опций в стиле Rails из набора аргументов переменной длины.

Открытые методы экземпляра

extract_options!

Извлекает опции из набора аргументов переменной длины. В конце имени стоит восклицательный знак, поскольку он удаляет и возвращает последний элемент массива, если тот является хешем. В противном случае возвращает пустой хеш, а исходный массив не изменяется:

```
def options(*args)
  args.extract_options!
end

options(1, 2) # => {}
options(1, 2, :a => :b) # => {:a=>:b}
```

Array::Grouping (в ActiveSupport::CoreExtensions)

Предоставляет два метода для разбиения элементов массива на логические группы.

Открытые методы экземпляра

in_groups_of(number, fill_with = nil) {|group| ...}

Настоящая суперзвезда Rails, метод `in_groups_of`, разбивает массив на группы размера `size`, заполняя оставшиеся свободными позиции в последней группе. Параметр `fill_with` определяет, каким значением заполнять данные позиции, и по умолчанию равен `nil`.

Если задан блок, то при вызове ему передается каждая группа; в противном случае возвращается двухмерный массив:

```
>> %w(1 2 3 4 5 6 7).in_groups_of(3)
=> [[1, 2, 3], [4, 5, 6], [7, nil, nil]

>> %w(1 2 3).in_groups_of(2, ' ') {|group| puts group }
[1, 2]
[3, " "]

>> %w(1 2 3).in_groups_of(2, false) {|group| puts group }
[1, 2]
[3]
```

Метод in_groups_of особенно полезен в объектах моделей, занимающихся пакетной обработкой, а также для генерации строк таблицы в шаблонах представлений.

split(value = nil, &block)

Разбивает массив на один или более подмассивов, исходя из значения разделителя:

```
[1, 2, 3, 4, 5].split(3) #=> [[1, 2], [4, 5]]
```

или результата, возвращенного необязательным блоком:

```
(1..8).to_a.split { |i| i % 3 == 0 } # => [[1, 2], [4, 5], [7, 8]]
```

CachingTools::HashCaching (в ActiveSupport)

Предоставляет метод, который упрощает кэширование вызовов с помощью вложенных хешей. Согласно документации по API, «данный паттерн является полезным и общеупотребительным в Ruby, поэтому делать это вручную не пристало».

Может быть, и так, но модуль CachingTools::HashCaching практически бесполезен, так как его единственный метод hash_cache автоматически недоступен ни в каком контексте Rails, а изучение кода показывает, что и для внутренних целей он не используется (если не считать автономных тестов).

Открытые методы экземпляра

hash_cache(method_name, options = {})

Динамически создает структуру из вложенных хешей для кэширования обращений к методу с именем method_name. Кэшированный метод называется method_name_cache, если только не задан параметр :as => :alternate_name.

Например, следующий (медленно работающий) метод slow_method:

```
def slow_method(a, b)
  a ** b
end
```

можно кэшировать, вызвав hash_cache :slow_method, в результате чего будет определен метод slow_method_cache.

Затем мы можем вычислить (и кэшировать) результат a ** b, воспользовавшись таким синтаксисом:

```
slow_method_cache[a][b]
```

Структура хеша создается с помощью вложенных вызовов Hash.new с блоками инициализации. Так, структура, возвращенная методом slow_method_cache, выглядит следующим образом:

```
Hash.new do |as, a|
  as[a] = Hash.new do |bs, b|
    bs[b] = slow_method(a, b)
  end
end
```

В реализации `hash_cache` активно используется метапрограммирование. Сгенерированный код сжат в одну строчку, чтобы в случае исключений показывалась осмысленная трассировка стека.

Class

Rails расширяет объект Ruby `Class`, добавляя в него целый ряд методов.

Открытые методы экземпляра

cattr_accessor(*syms)

Определяет один или несколько методов чтения и записи атрибутов класса в том же стиле, что встроенные акцессоры `attr*` для доступа к атрибутам экземпляра. Широко используется в коде Rails для сохранения опций. Значения доступны в подклассах по ссылке, что принципиально отличается от механизма `class_inheritable_accessor`.

cattr_reader(*syms)

Определяет один или несколько методов чтения атрибутов класса.

cattr_writer(*syms)

Определяет один или несколько методов записи атрибутов класса.

class_inheritable_accessor(*syms)

Допускает общий доступ к атрибутам в рамках иерархии наследования, но каждый потомок получает свою копию атрибутов родителя, а не указатель на одно и то же значение. Это означает, что потомок может, например, добавлять элементы в массив, не опасаясь, что изменения будут видны родителю, братьям или потомкам. Это совершенно непохоже на стандартные атрибуты уровня класса, которые являются общими для всей иерархии.

class_inheritable_array

Вспомогательный метод, который создает наследуемые методы чтения и записи и по умолчанию присваивает соответствующему атрибуту пустой массив в качестве значения, чтобы вам не приходилось инициализировать его самостоятельно.

class_inheritable_hash

Вспомогательный метод, который создает наследуемые методы чтения и записи и по умолчанию присваивает соответствующему атрибуту пустой хеш в качестве значения, чтобы вам не приходилось инициализировать его самостоятельно.

class_inheritable_reader(*syms)

Определяет один или несколько наследуемых методов чтения атрибутов класса.

class_inheritable_writer(*syms)

Определяет один или несколько наследуемых методов записи атрибутов класса.

const_missing(class_id)

Обращение к обратному вызову `const_missing` происходит, когда Ruby не может найти указанную константу в текущей области видимости. Именно это лежит в основе механизма автоматической загрузки классов в Rails. Дополнительную информацию см. в описании модуля `Dependencies`.

remove_class(*klasses)

Удаляет константу, ассоциированную с указанными классами, в результате чего они становятся недоступными и бесполезными.

remove_subclasses

Удаляет все подклассы данного класса.

subclasses

Возвращает все подклассы данного класса.

CGI::EscapeSkippingSlashes (в ActiveSupport::CoreExtensions)

Открытые методы экземпляра

escape_skipping_slashes(str)

Принимает строку, которая будет использоваться как URL, и экранирует в ней все символы, кроме букв и цифр (исключая знаки косой черты):

```
>> CGI.escape_skipping_slashes "/amc/shows/mad men on thursday nights"
=> "/amc/shows/mad+men+on+thursday+nights"
```

Date::Behavior (в ActiveSupport::CoreExtensions)

Открытые методы экземпляра

acts_like_date?

Просто возвращает `true`, чтобы динамическая типизация `Date`-подобных классов была более предсказуемой:

```
>> Date.today.acts_like_date? #=> true
```

Date::Calculations (в ActiveSupport::CoreExtensions)

Реализует вычисления с объектами `Date`.

Открытые методы экземпляра

+ (other)

Rails переопределяет существующий оператор `+`, так что если аргумент `other` – экземпляр класса `ActiveSupport::Duration` (тип, возвращаемый такими методами, как `10.minutes` и `9.months`), вызывается метод `since`:

```
>> Date.today + 1.day == Date.today.tomorrow
=> true
```

advance(options)

Реализует точные вычисления с годами, месяцами и днями. Параметр `options` – хеш, в котором могут присутствовать следующие ключи: `:months`, `:days`, `:years`:

```
>> Date.new(2006, 2, 28) == Date.new(2005, 2, 28).advance(:years => 1)
=> true
```

ago(seconds)

Преобразует `Date` в `Time` (или при необходимости в `DateTime`), причем часть, соответствующая времени, устанавливается на начало суток (0:00), а затем вычитает заданное количество секунд:

```
>> Time.local(2005, 2, 20, 23, 59, 15) == Date.new(2005, 2, 21).ago(45)
=> true
```

at_beginning_of_day, at_midnight, beginning_of_day, midnight

Преобразует Date в Time (или при необходимости в DateTime), причем часть, соответствующая времени, устанавливается на начало суток (0:00):

```
>> Time.local(2005,2,21,0,0,0) == Date.new(2005,2,21).beginning_of_day
=> true
```

at_beginning_of_month и beginning_of_month

Возвращает объект DateTime, представляющий начало месяца (первое число месяца, время установлено в 0:00):

```
>> Date.new(2005, 2, 1) == Date.new(2005,2,21).beginning_of_month
=> true
```

at_beginning_of_quarter и beginning_of_quarter

Возвращает объект Date/DateTime, представляющий начало квартала (1 января, 1 апреля, 1 июля и 1 октября):

```
>> Date.new(2005, 4, 1) == Date.new(2005, 6, 30).beginning_of_quarter
=> true
```

at_beginning_of_week, beginning_of_week и monday

Возвращает объект Date (или DateTime), представляющий начало недели (первым днем недели считается понедельник):

```
>> Date.new(2005, 1, 31) == Date.new(2005, 2, 4).beginning_of_week
=> true
```

at_beginning_of_year и beginning_of_year

Возвращает объект Date/DateTime, представляющий начало календарного года (1 января):

```
>> Date.new(2005, 1, 1) == Date.new(2005, 2, 22).beginning_of_year
=> true
```

at_end_of_month и end_of_month

Возвращает объект Date/DateTime, представляющий последний день календарного месяца:

```
>> Date.new(2005, 3, 31) == Date.new(2005,3,20).end_of_month
=> true
```

change(options)

Возвращает новый объект Date, в котором один или несколько элементов изменены в соответствии с параметрами, заданными в хеше options.

Допустимы следующие параметры: :year, :month и :day.

```
>> Date.new(2007, 5, 12).change(:day => 1) == Date.new(2007, 5, 1)
=> true

>> Date.new(2007, 5, 12).change(:year => 2005, :month => 1) == ➥
Date.new(2005, 1, 12)
=> true
```

end_of_day()

Преобразует `Date` в `Time` (или при необходимости в `DateTime`), причем часть, соответствующая времени, устанавливается на конец суток (23:59:59):

```
>> Time.local(2005,2,21,23,59,59) == Date.new(2005, 2, 21).end_of_day
=> true
```

in(seconds) и since(seconds)

Преобразует `Date` в `Time` (или при необходимости в `DateTime`), причем часть, соответствующая времени, устанавливается на начало суток (0:00), а затем прибавляет заданное количество секунд:

```
>> Time.local(2005, 2, 21, 0, 0, 45) == Date.new(2005, 2, 21).since(45)
=> true
```

last_month()

То же самое, что `months_ago(1)`.

last_year()

То же самое, что `years_ago(1)`.

months_ago(months)

Возвращает новый объект `Date` (или `DateTime`), представляющий момент времени `month` месяцев назад:

```
>> Date.new(2005, 1, 1) == Date.new(2005, 3, 1).months_ago(2)
=> true
```

months_since(months)

Возвращает новый объект `Date` (или `DateTime`), который представляет момент времени, отстоящий от текущего на `month` месяцев в прошлом или в будущем. Для представления момента в прошлом задайте отрицательное количество месяцев:

```
>> Date.today.months_ago(1) == Date.today.months_since(-1)
=> true
```

next_month()

То же самое, что `months_since(1)`.

next_week(day = :monday)

Возвращает новый объект `Date` (или `DateTime`), представляющий начало указанного дня на следующей календарной неделе. Принимаемый по умолчанию день недели можно переопределить, задав название дня в виде символа:

```
>> Date.new(2005, 3, 4) == Date.new(2005, 2, 22).next_week(:friday)
=> true
```

next_year()

То же самое, что `years_since(1)`.

tomorrow()

Вспомогательный метод, возвращающий объект `Date` (или `DateTime`), который представляет момент времени, отстоящий от текущего на один день в будущем:

```
>> Date.new(2007, 3, 2) == Date.new(2007, 2, 28).tomorrow.tomorrow
=> true
```

years_ago(years)

Возвращает новый объект `Date` (или `DateTime`), представляющий момент времени `years` лет назад:

```
>> Date.new(2000, 6, 5) == Date.new(2007, 6, 5).years_ago(7)
=> true
```

years_since(years)

Возвращает новый объект `Date` (или `DateTime`), представляющий момент времени `years` лет вперед:

```
>> Date.new(2007, 6, 5) == Date.new(2006, 6, 5).years_since(1)
=> true
```

yesterday()

Вспомогательный метод, возвращающий объект `Date` (или `DateTime`), который представляет момент времени, отстоящий от текущего на один день в прошлом:

```
>> Date.new(2007, 2, 21) == Date.new(2007, 2, 22).yesterday
=> true
```

Date::Conversions (в ActiveSupport::CoreExtensions)

Этот модуль подмешивает в класс Date методы для получения дат в различных строковых форматах, а также в виде других объектов.

Константы

Константа DATE_FORMATS содержит хеш форматов, используемых в методе to_formatted_s:

```
DATE_FORMATS = {
  :short        => "%e %b",
  :long         => "%B %e, %Y",
  :db           => "%Y-%m-%d",
  :long_ordinal => lambda {|date| date.strftime("%B
#{date.day.ordinalize}, %Y") }, # => "April 25th, 2007"
  :rfc822       => "%e %b %Y" }
```

Открытые методы экземпляра

to_date

Применяется, чтобы объекты классов Time, Date и DateTime были взаимозаменяемы в преобразованиях.

to_datetime

Преобразует объект Date в объект встроенного в Ruby класса DateTime. Время устанавливается на начало суток.

to_formatted_s(format = :default)

Преобразует объект Date в строковое представление в соответствии с предопределенными форматами, хранящимися в хеше DATE_FORMATS (синоним to_s; исходному методу to_s назначен синоним to_default_s):

```
def test_to_s
  date = Date.new(2005, 2, 21)
  assert_equal "2005-02-21", date.to_s
  assert_equal "21 Feb", date.to_s(:short)
  assert_equal "February 21, 2005", date.to_s(:long)
  assert_equal "February 21st, 2005", date.to_s(:long_ordinal)
  assert_equal "2005-02-21", date.to_s(:db)
  assert_equal "21 Feb 2005", date.to_s(:rfc822)
end
```

to_time(timezone = :local)

Преобразует объект `Date` в объект встроенного в Ruby класса `Time`. Время устанавливается на начало суток. В качестве часового пояса можно указать `:local` или `:utc`.

```
>> Time.local(2005, 2, 21) == Date.new(2005, 2, 21).to_time
=> true
```

xmlschema

Возвращает строку, которая представляет время в формате, определенном стандартом XML Schema (известен также под названием iso8601):

```
CCYY-MM-DDThh:mm:ssTZD
```

Если объект `Date` представлен в виде UTC, то в качестве TZD указывается Z. В противном случае смещение от нулевого часового пояса прописывается в виде `[+-]hh:mm`.

DateTime::Calculations (в ActiveSupport::CoreExtensions)

Реализует вычисления внутри самого класса `DateTime`.

Открытые методы экземпляра

at_beginning_of_day, at_midnight, beginning_of_day, midnight

Вспомогательные методы, представляющие начало суток (00:00). Реализованы как `change(:hour => 0)`.

advance(options)

Использует класс `Date` для выполнения точных вычислений с годами, месяцами и днями. Параметр `options` – хеш, в котором могут присутствовать следующие ключи: `:months`, `:days`, `:years`.

ago(seconds)

Возвращает новый объект `DateTime`, представляющий момент времени `seconds` секунд назад. Противоположен `since`.

change(options)

Возвращает новый объект `DateTime`, в котором одни или несколько элементов изменены в соответствии с параметрами, заданными в хеше

options. Допустимые параметры для даты: :year, :month, :day. Допустимые параметры для времени: :hour, :min, :sec, :offset и :start.

end_of_day

Вспомогательный метод, возвращающий конец суток (23:59:59). Реализован как change(:hour => 23, :min => 59, :sec => 59).

seconds_since_midnight

Возвращает количество секунд, прошедших с полуночи.

since(seconds)

Возвращает новый объект DateTime, который представляет момент времени, отстоящий от указанного момента на заданное количество секунд. Противоположен ago.

DateTime::Conversions (в ActiveSupport::CoreExtensions)

Открытые методы экземпляра

readable_inspect

Переопределяет стандартный метод inspect, возвращая привычное для человека представление момента времени:

```
Mon, 21 Feb 2005 14:30:00 +0000
```

to_date

Преобразует self в объект Ruby Date, отбрасывая время.

to_datetime

Возвращает объект self, модифицированный так, что он способен хранить экземпляры классов Time, Date и DateTime и подставлять один вместо другого при преобразованиях.

to_formatted_s(format=:default)

См. параметры, передаваемые методу to_formatted_s в классе Time.

to_time

Пытается преобразовать `self` в объект Ruby `Time`. Возвращает `self`, если значение оказывается вне диапазона, представимого классом `Time`. Если `self.offset` равно 0, пытается привести к времени UTC, в противном случае – к локальному часовому поясу.

Dependencies (в ActiveSupport)

Содержит логику, необходимую для механизма автоматической загрузки классов в Rails, который позволяет ссылаться на любую константу в путях загрузки Rails, не требуя директивы `require`.

Этот модуль расширяет *сам себя* – любопытный трюк, применяющийся для модулей, которые желательно использовать в любом месте программы в функциональном духе:

```
module Dependencies
extend self
```

В результате можно вызывать методы, указывая слева от точки просто константу, соответствующую модулю (как статические методы классов в Java):

```
Dependencies.search_for_file('.erb')
```

Этот модуль нужен нечасто и в основном используется внутри Rails и в подключаемых модулях. Но во время отладки хитрых ошибок, связанных с загрузкой классов, полезно понимать, как он работает.

Атрибуты модуля

Некоторые из описанных ниже атрибутов устанавливаются на основе конфигурационных настроек в файлах окружения для различных режимов (см. главу 1).

autoloaded_constants

Массив квалифицированных имен уже загруженных констант. Если добавить имя в этот массив, то при следующей очистке `Dependencies` соответствующая константа будет выгружена.

clear

Очищает список загруженных классов и удаляет невыгружаемые константы.

constant_watch_stack

Внутренний стек, используемый для отслеживания констант, которые были загружены каждым блоком.

explicitly_unloadable_constants

Массив имен констант, которые следует выгружать при каждом запросе. Используется, чтобы помечать произвольные константы как подлежащие выгрузке.

history

Объект `Set`, в котором хранятся все *когда-либо* загруженные файлы.

load_once_paths

Объект `Set`, в котором хранятся каталоги, откуда автоматически загружаемые константы загружаются только один раз. Все каталоги, присутствующие в этом множестве, должны присутствовать также в `+load_paths+`.

load_paths

Объект `Set`, в котором хранятся каталоги, откуда Rails может автоматически загружать файлы. Файлы, находящиеся в этих каталогах, будут перезагружаться при каждом запросе в режиме разработки, если только каталог не присутствует также во множестве `load_once_paths`.

loaded

Объект `Set`, в котором хранятся все файлы, загруженные *в текущий момент*.

log_activity

Присвойте этому атрибуту параметр `true`, если хотите разрешить протоколирование вызовов `const_missing` и загрузки файлов (по умолчанию `false`).

mechanism

Этот атрибут определяет, какие файлы загружены (по умолчанию) или затребованы. От его значения зависит, будет ли Rails перезагружать классы при каждом запросе, как в режиме разработки.

warnings_on_first_load

Данный атрибут определяет, нужно ли активировать предупреждения Ruby при первой загрузке зависимых файлов. По умолчанию `true`.

Открытые методы экземпляра

associate_with(file_name)

Вызывает метод `depend_on` с параметром `swallow_load_errors`, равным `true`. Обертывается методом `require_association` из класса `Object`.

autoload_module!(into, const_name, qualified_name, path_suffix)

Пытается автоматически загрузить модуль с указанным именем, ища каталог с ожидаемым суффиксом `path suffix`. Если каталог найден, модуль создается и записывается в набор констант `into` с именем `+const_name+`. Если каталог был найден на одном из перезагружаемых путей, то модуль добавляется также в набор констант, подлежащих выгрузке.

autoloadable_module?(path_suffix)

Проверяет, соответствует ли суффикс `path_suffix` какому-нибудь автозагружаемому модулю. Но возвращается не булево значение, а базовый путь к этому модулю.

autoloaded?(constant)

Определяет, был ли модуль, соответствующий заданной константе `constant`, автоматически загружен.

depend_on(file_name, swallow_load_errors = false)

Ищет файл с указанным именем `file_name` и устанавливает новую зависимость в соответствии со значением аргумента `require_or_load`. Аргумент `swallow_load_errors` говорит, следует ли подавить исключение `LoadError`. Обертывается методом `require_dependency` из класса `Object`.

load?

Возвращает `true`, если `mechanism` установлен в `:load`.

load_file(path, const_paths = loadable_constants_for_path(path))

Загружает файл с указанным путем `path`. Аргумент `const_paths` – множество полностью квалифицированных константных имен. Во время загрузки файла модуль `Dependencies` проверяет, была ли добавлена какая-нибудь из этих констант. Любая добавленная константа помечается как автозагруженная и удаляется при следующем обращении к `Dependencies.clear`.

Если второй аргумент опущен, то `Dependencies` сконструирует множество имен, которые может определить файл с путем `path`. Дополнительную информацию см. в описании метода `loadable_constants_for_path`.

load_once_path?(path)

Возвращает `true`, если указанный путь `path` присутствует в списке `load_once_path`.

load_missing_constant(mod, const_name)

Загружает отсутствующую константу с именем `const_name` из модуля `mod`. Если загрузить ее из этого модуля невозможно, пробует родительский модуль, вызывая метод `const_missing` для него.

loadable_constants_for_path(path, bases = load_paths)

Возвращает массив констант для файла с указанным путем `path`. При этом `Dependencies` пытается загрузить данный файл.

mark_for_unload(constant)

Помечает указанную константу `constant` для выгрузки. Константа будет выгружаться при каждом запросе, а не только при следующем.

new_constants_in(*descs) {...}

Выполняет заданный блок и обнаруживает все новые константы, которые были загружены при его выполнении. Константа может считаться *новой* только один раз. Если блок вызовет `new_constants_in` еще раз, константы, определенные во внутреннем вызове, не будут обнаружены.

Если блок не завершает выполнение нормально, а возбуждает исключение, то все новые константы считаются частично определенными и немедленно удаляются.

qualified_const_defined?(path)

Возвращает `true`, если метод `defined?` возвращает `true` для указанного пути к константе.

qualified_name_for(parent_module, constant_name)

Возвращает квалифицированный путь для указанного родительского модуля `parent_module` и константы с именем `constant_name`.

remove_unloadable_constants!

Удаляет константы, которые были автозагружены или помечены для выгрузки.

require_or_load(file_name, const_path = nil)

Реализует основной механизм загрузки классов. Обертывается методом `require_or_load` из класса `Object`.

search_for_file(path_suffix)

Ищет файл по указанным в `load_paths` путям с заданным суффиксом `path_suffix`.

will_unload?(constant)

Возвращает `true`, если указанная константа поставлена в очередь на выгрузку при следующем запросе.

Deprecation (в ActiveSupport)

Этот модуль предоставляет разработчикам ядра Rails и приложений формальный механизм, позволяющий явно указать, что метод считается устаревшим (то есть *будет удален в будущих версиях*). Rails помещает в протокол предупреждение при вызове устаревших методов.

Чтобы пометить метод как устаревший, достаточно вызвать метод `deprecate` и передать ему имя метода в виде символа. Не забудьте, что вызов `deprecate` должен стоять *после определения метода*.

```
deprecate :subject_of_regret
```

Метод `deprecate` подмешан в класс Ruby `Module`, поэтому доступен в любом месте.

Deprecation::Assertions (в ActiveSupport)

В этом модуле находятся утверждения, позволяющие проверить «устарелость» методов.

Открытые методы экземпляра

assert_deprecated(match = nil) { ... }

Утверждает, что код внутри блока приводит к выдаче предупреждения об устарелости. Необязательный аргумент `match` позволяет точнее сформулировать утверждение, включив в него проверку имени метода. Достаточно передать регулярное выражение, соответствующее именам ожидаемых устаревших методов:

```
def test_that_subject_of_regret_is_deprecated
  assert_deprecated do
    subject_of_regret
  end
end
```

assert_not_deprecated { ... }

Утверждает, что код внутри блока не вызывает никаких устаревших методов.

Duration (в ActiveSupport)

Предоставляет точные средства работы с датой и временем, пользуясь методом `advance`, имеющимся в классах `Date` и `Time`. Поддерживает методы из класса `Numeric`, как показано в следующем примере:

```
1.month.ago # эквивалентно Time.now.advance(:months => -1)
```

Открытые методы экземпляра

+ (other)

Прибавляет объект типа `Duration` или `Numeric` к `Duration`. Значение объекта `Numeric` трактуется как количество секунд.

– (other)

Вычитает объект типа `Duration` или `Numeric` из `Duration`. Значение объекта `Numeric` трактуется как количество секунд.

ago(time = Time.now)

Возвращает новый объект типа `Time` или `Date`, который представляет момент времени в прошлом, отстоящий от указанного на величину `Duration`:

```
birth = 35.years.ago
```

from_now(time = Time.now)

Синоним метода `since`, который читается чуть более естественно, если в качестве аргумента `time` используется значение по умолчанию `Time.now`:

```
expiration = 1.year.from_now
```

inspect

Вычисляет время, представленное объектом `Duration`, и форматирует его в виде строки, пригодной для вывода на консоль (напомним, что

IRB и консоль Rails автоматически вызывают метод `inspect` для возвращенных им объектов; вы можете пользоваться этим приемом и при создании собственных объектов):

```
>> 10.years.ago
=> Sun Aug 31 17:34:15 -0400 1997
```

since(time = Time.now)

Возвращает новый объект типа `Time` или `Date`, который представляет момент времени в будущем, отстоящий от указанного на величину `Duration`:

```
expiration = 1.year.since(account.created_at)
```

until(time = Time.now)

Синоним метода `ago`. Читается чуть более естественно, если в качестве аргумента `time` используется значение по умолчанию `Time.now`:

```
membership_duration = created_at.until(expires_at)
```

Enumerable

Расширения встроенного в Ruby метода `Enumerable`, наделяющие массивы и другие типы наборов возможностями итерирования.

Открытые методы экземпляра

group_by(&block)

Группирует перечисляемый объект, создавая множества на основе результата, возвращенного блоком. Полезен, например, для группировки записей по датам:

```
latest_transcripts.group_by(&:day).each do |day, transcripts|
  puts "[#{day}] #{transcripts.map(&:class).join ', '}"
end

"[2006-03-01] Transcript"
"[2006-02-28] Transcript"
"[2006-02-27] Transcript, Transcript"
"[2006-02-26] Transcript, Transcript"
"[2006-02-25] Transcript"
"[2006-02-24] Transcript, Transcript"
"[2006-02-23] Transcript"
```

Начиная с версии 1.9, пользуется встроенным в Ruby методом `group_by`.

sum(default = 0, &block)

Вычисляет сумму элементов перечисляемого объекта, основываясь на результатах выполнения блока:

```
payments.sum(&:price)
```

Этот метод проще для понимания, чем более хитрый метод `inject`, встроенный в Ruby:

```
payments.inject { |sum, p| sum + p.price }
```

Для более сложных вычислений применяйте полный синтаксис блока (а не трюк, основанный на `to_proc`):

```
payments.sum { |p| p.price * p.tax_rate }
```

Кроме того, метод `sum` может вычислять результат и без блока:

```
[5, 15, 10].sum # => 30
```

Принимаемый по умолчанию *нейтральный элемент* (заумный способ сказать «сумма пустого списка») равен 0. Но вы можете переопределить его, задав аргумент `default`:

```
[].sum(Payment.new(0)) { |i| i.amount } # => Payment.new(0)
```

index_by

Преобразует перечисляемый объект в хеш, основываясь на блоке, который определяет ключи. Чаще всего употребляется с именем одного атрибута:

```
>> people.index_by(&:login)
=> { "nextangle" => <Person ...>, "chad" => <Person ...>}
```

Для генерации более сложных ключей применяйте полный синтаксис блока (а не трюк, основанный на `to_proc`):

```
>> people.index_by { |p| "#{p.first_name} #{p.last_name}" }
=> {"Chad Fowler" => <Person ...>, "David Hansson" => <Person ...>}
```

Exception

Расширения встроенного в Ruby класса `Exception`.

Открытые методы экземпляра

application_backtrace

Возвращает трассу вызовов, приведших к данному исключению, без строк, которые указывают на файлы в следующих каталогах: `generated`, `vendor`, `dispatch`, `ruby`, `script`.

blame_file!(file)

Применяется, чтобы возложить ответственность за исключение на конкретный файл.

blamed_files

Возвращает массив файлов, на который была возложена ответственность за исключение.

copy_blame!(other)

Копирует массив «обвиняемых» файлов из одного исключения в другое.

framework_backtrace

Противоположность методу `application_backtrace`: возвращает трассу вызовов, приведших к данному исключению, в которой оставлены только строки, указывающие на файлы в следующих каталогах: `generated`, `vendor`, `dispatch`, `ruby`, `script`.

FalseClass

Напомним, что в Ruby все является объектами, даже литерал `false`, представляющий собой специальную ссылку на единственный экземпляр класса `FalseClass`.

Открытые методы экземпляра

blank?

Всегда возвращает `true`.

File

Подмешивает метод `atomic_write` в класс Ruby `File`.

Открытые методы экземпляра

atomic_write(file_name, temp_dir = Dir.tmpdir)

Выполняет атомарную запись в файл, для чего сначала записывает во временный файл, а затем переименовывает последний в файл с именем `file_name`. Полезен, когда необходима строгая гарантия того, что никакой другой процесс или поток не увидит частичного записанного файла:

```
File.atomic_write("important.file") do |file|
  file.write("hello")
end
```

Если каталог temp находится не в той же файловой системе, что и файл, в который вы хотите записать, можно указать другой временный каталог с помощью аргумента temp_dir:

```
File.atomic_write("/data/something.imporant", "/data/tmp") do |f|
  file.write("hello")
end
```

Hash

Хеши используются в Rails повсеместно, и все же ActiveSupport добавляет один метод прямо в класс Hash.

Открытые методы экземпляра

blank?

Имеет синоним empty? и возвращает true, если в хеше нет ни одного элемента.

Hash::ClassMethods (в ActiveRecord::CoreExtensions)

Содержит метод from_xml, который позволяет быстро преобразовать правильно отформатированный XML-документ в структуру, состоящую из вложенных хешей.

Открытые методы экземпляра

from_xml(xml)

Преобразует произвольные строки, содержащие XML-разметку, во вложенные массивы и хеши Ruby. Очень удобен для наспех сляпанной интеграции с REST-совместимыми веб-службами.

Ниже приведен пример использования в консоли для случайного XML-содержимого. Единственное требование – правильность XML-разметки:

```
>> xml = %(<people>
  <person id="1">
    <name><family>Boss</family> <given>Big</given></name>
    <email>chief@foo.com</email>
  </person>
  <person id="2">
    <name>
    <family>Worker</family>
    <given>Two</given></name>
```

```
    <email>two@foo.com</email>
  </person>
</people>)
=> "<people>...</people>"

>> h = Hash.from_xml(xml)
=> {"people"=>{"person"=>[{"name"=>{"given"=>"Big", "family"=>"Boss"},
"id"=>"1", "email"=>"chief@foo.com"}, {"name"=>{"given"=>"Two",
"family"=>"Worker"}, "id"=>"2", "email"=>"two@foo.com"}]}}
```

Теперь к данным из исходного XML-документа можно легко обратиться:

```
>> h["people"]["person"].first["name"]["given"] => "Big"
```

Hash::Conversions (в ActiveSupport::CoreExtensions)

Предоставляет методы для преобразования хешей в другие формы.

Константы

Хеш XML_TYPE_NAMES показывает, как классы Ruby отображаются на типы, определенные в спецификации XML Schema:

```
XML_TYPE_NAMES = {
  "Fixnum"     => "integer",
  "Bignum"     => "integer",
  "BigDecimal" => "decimal",
  "Float"      => "float",
  "Date"       => "date",
  "DateTime"   => "datetime",
  "Time"       => "datetime",
  "TrueClass"  => "boolean",
  "FalseClass" => "boolean"
}
```

Хеш XML_FORMATTING содержит набор Proc-объектов, используемых для преобразования некоторых видов объектов Ruby в строку в формате XML:

```
XML_FORMATTING = {
  "date"     => Proc.new { |date| date.to_s(:db) },
  "datetime" => Proc.new { |time| time.xmlschema },
  "binary"   => Proc.new { |binary| Base64.encode64(binary) },
  "yaml"     => Proc.new { |yaml| yaml.to_yaml }
}
```

Хеш XML_PARSING содержит набор Proc-объектов, используемых для преобразования строк в формате XML в объекты Ruby:

```
XML_PARSING = {
  “date”      => Proc.new { |date| ::Date.parse(date) },
  “datetime”  => Proc.new { |time| ::Time.parse(time).utc },
  “integer”   => Proc.new { |integer| integer.to_i },
  “float”     => Proc.new { |float| float.to_f },
  “decimal”   => Proc.new { |number| BigDecimal(number) },
  “boolean”   => Proc.new do |boolean|
    %w(1 true).include?(boolean.strip)
end,
  “string”       => Proc.new { |string| string.to_s },
  “yaml”         => Proc.new { |yaml| YAML::load(yaml) rescue yaml },
  “base64Binary” => Proc.new { |bin| Base64.decode64(bin) },
  “file”         => Proc.new do |file, entity|
    f = StringIO.new(Base64.decode64(file))
    eval “def f.original_filename()
      ‘#{entity[“name”]}’ || ‘untitled’
   end”
   eval “def f.content_type()
      ‘#{entity[“content_type”]}’ || ‘application/octet-stream’
   end”
   f
  end
}

XML_PARSING.update(
  “double”   => XML_PARSING[“float”],
  “dateTime” => XML_PARSING[“datetime”]
)
```

Открытые методы экземпляра

to_query

Составляет из ключей и значений хеша строку запроса в формате URL, употребляя знаки амперсанда и равенства в качестве разделителей:

```
>> {:foo => “hello”, :bar => “goodbye”}.to_query
=> “bar=goodbye&foo=hello”
```

to_xml(options={})

Составляет из ключей и значений хеша простое XML-представление:

```
>> print ({:greetings => {
              :english => “hello”,
              :spanish => “hola”}}).to_xml

<?xml version=”1.0” encoding=”UTF-8”?>
```

```
<hash>
  <greetings>
    <english>hello</english>
    <spanish>hola</spanish>
  </greetings>
</hash>
```

Полный список опций см. в описании метода `Array::Conversions to_xml`.

Hash::Diff (в ActiveSupport::CoreExtensions)

Содержит метод для вычисления разности между двумя хешами.

Открытые методы экземпляра

diff(hash2)

Метод для вычисления разности между двумя хешами. Возвращает разность между данным хешем и хешем, переданным в качестве параметра.

Вот пример использования в консоли:

```
>> {:a => :b}.diff({:a => :b})
=> {}
>> {:a => :b}.diff({:a => :c})
=> {:a=>:b}
```

Hash::Except (в ActiveSupport::CoreExtensions)

Возвращает хеш, включающий все ключи, кроме заданных. Полезен, когда нужно быстро исключить из хэша некоторые ключи:

```
@person.update_attributes(params[:person].except(:admin))
```

Открытые методы экземпляра

except(*keys)

Возвращает новый хеш, в котором нет указанных ключей, оставляя исходный хеш без изменения.

except!(*keys)

Удаляет указанные ключи прямо из данного хеша.

Hash::Keys (в ActiveSupport::CoreExtensions)

Содержит методы для работы с ключами хеша. Методы `stringify` и `symbolize` широко используются в коде Rails, поэтому обычно не имеет значения, передавать ли имена в виде символов или строк.

Вы можете пользоваться методом `assert_valid_keys`, который принимает хеши опций в стиле Rails и в собственных приложениях.

Открытые методы экземпляра

assert_valid_keys(*valid_keys)

Возбуждает исключение `ArgumentError`, если хеш содержит хотя бы один ключ, отсутствующий среди аргументов `valid_keys`:

```
def my_method(some_value, options={})
  options.assert_valid_keys(:my_conditions, :my_order, ...)
  ...
end
```

stringify_keys

Возвращает новую копию хеша, в которой все ключи преобразованы в строки.

stringify_keys!

Деструктивно преобразует все ключи хеша в строки.

symbolize_keys и to_options

Возвращает новый хеш, в котором все ключи преобразованы в символы.

symbolize_keys! и to_options!

Деструктивно преобразует все ключи хеша в символы.

Hash::ReverseMerge (in ActiveSupport::CoreExtensions)

Реализует инверсное слияние, когда ключи в хеше, для которого вызывается метод, имеют приоритет по сравнению с ключами в хеше `other_hash`. Это особенно полезно для инициализации переданного хеша опций значениями по умолчанию:

```
def setup(options = {})
  options.reverse_merge! :size => 25, :velocity => 10
end
```

В данном примере опциям `:size` и `:velocity` присваиваются значения по умолчанию, только когда в хеше `options` отсутствуют явно заданные значения.

Открытые методы экземпляра

reverse_merge(other_hash)

Возвращает результат слияния двух хешей, когда значения ключей из `other_hash` трактуются как умолчания. Исходный хеш не изменяется.

reverse_merge!(other_hash) и reverse_update

Деструктивные варианты `reverse_merge`; оба метода модифицируют исходный хеш.

Hash::Slice (in ActiveSupport::CoreExtensions)

Метод для вырезания части хеша, содержащей только указанные ключи. Полезен, когда до передачи методу хеша опций из него нужно удалить все недопустимые ключи:

```
def search(criteria = {})
  assert_valid_keys(:mass, :velocity, :time)
end

search(options.slice(:mass, :velocity, :time))
```

Открытые методы экземпляра

slice(*keys)

Возвращает новый хеш, содержащий только ключи из аргумента `keys`.

slice!(*keys)

Деструктивный вариант `slice`; модифицирует исходный хеш, удаляя из него ключи, не содержащиеся в аргументе `keys`.

HashWithIndifferentAccess

Подкласс `Hash`, используемый внутри Rails. В исходном тексте говорится:

> *У этого класса сомнительная семантика, и мы оставили его только для того, чтобы была возможность писать `params[:key]` вместо `params['key']`.*

Inflector::Inflections (в ActiveSupport)

Класс `Inflections` преобразует слова из единственного числа во множественное, имена классов – в имена таблиц, имена классов с указанием модуля – в имена без модуля, имена классов – во внешние ключи. Принимаемые по умолчанию флексии для преобразования в единственное или множественное число, а также неисчисляемые имена существительные хранятся в файле `activesupport/lib/active_support/inflections.rb`.

Метод `Inflector.inflections` поставляет единственный экземпляр класса `Inflections`, который можно использовать для задания дополнительных правил словоизменения в вашем файле `config/environment.rb`.

Примеры:

```
Inflector.inflections do |inflect|
  inflect.plural /^(ox)$/i, '\1en'
  inflect.singular /^(ox)en/i, '\1'
  inflect.irregular 'octopus', 'octopi'
  inflect.uncountable "equipment"
end
```

Новые правила добавляются в начало. В примере выше исключение для слова `octopus` окажется первым среди проверяемых правил преобразования в единственное и множественное число. Тем самым Rails гарантирует, что ваши правила будут иметь более высокий приоритет по сравнению с ранее загруженными.

Открытые методы экземпляра

Ниже перечислены методы словоизменения из модулей, где они фактически используются: `Numeric::Inflections` и `String::Inflections`.

irregular(singular, plural)

Задает новое исключение из правил, действующее для преобразования как в единственное, так и во множественное число. Аргументы `singular` и `plural` должны быть строками, а не регулярными выражениями. Просто передайте слово-исключение в единственном и во множественном числе:

```
irregular 'octopus', 'octopi'
irregular 'person', 'people'
```

plural(rule, replacement)

Задает новое правило преобразования во множественное число. Аргумент `rule` может быть либо строкой, либо регулярным выражением. Аргумент `replacement` должен быть только строкой, но может включать ссылки на сопоставившиеся с `rule` данные (с помощью синтаксической конструкции с обратной чертой):

```
Inflector.inflections do |inflect|
  inflect.plural /^(ox)$/i, '\1en'
end
```

singular(rule, replacement)

Задает новое правило преобразования в единственное число и соответствующую замену. Аргумент `rule` может быть либо строкой, либо регулярным выражением. Аргумент `replacement` должен быть только строкой, но может включать ссылки на сопоставившиеся с `rule` данные (с помощью синтаксической конструкции с обратной чертой):

```
Inflector.inflections do |inflect|
  inflect.singular /^(ox)en/i, '\1'
end
```

uncountable(*words)

Добавляет в список правил словоизменения неисчисляемые имена существительные, которые вообще не должны меняться:

```
uncountable "money"
uncountable "money", "information"
uncountable %w( money information rice )
```

Integer::EvenOdd (в ActiveSupport::CoreExtensions)

Методы для проверки того, является ли число четным, нечетным или кратным другому числу.

Открытые методы экземпляра

even?

Возвращает `true`, если целое число четное. Нулевое значение считается четным числом[1].

```
1.even? # => false
```

multiple_of?(number)

Возвращает `true`, если целое число является кратным `number`.

```
9.multiple_of? 3 # => true
```

[1] Интересное рассуждение на тему о том, почему многие считают 0 четным числом, см. на странице *http://ask.yahoo.com/20020909.html*.

odd?

Возвращает `true`, если целое число нечетное.

```
1.odd? # => false
```

Integer::Inflections (в ActiveSupport::CoreExtensions)

Содержит метод для преобразования целых чисел в форму порядкового числительного.

Открытые методы экземпляра

ordinalize

Преобразует целое число в строку, содержащую его порядковую форму, например: 1st, 2nd, 3rd, 4th.

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
1002.ordinalize # => "1002nd"
1003.ordinalize # => "1003rd"
```

JSON (в ActiveSupport)

Аббревиатура JSON расшифровывается как JavaScript Object Notation. Этот формат можно использовать для сериализации данных. Он не так тяжеловесен, как XML, и легко разбирается интерпретаторами языка JavaScript, поскольку представляет собой естественную форму литеральных объектов в этом языке:

```
{ drink: "too much", smoke: "too much" }
```

Возможно, в версии Ruby 1.9 и выше будет встроена улучшенная поддержка JSON, поскольку в язык добавлена нотация хеша-литерала, которая выглядит в точности так, как в JavaScript:

```
{ :drink: "too much", :smoke: "too much" } # допустимый хеш в Ruby 1.9
```

В последнее время формат JSON стал популярным способом транспортировки данных в Ajax-приложениях. В главе 12 «Ajax on Rails» один из разделов целиком посвящен JSON.

Константы

Употребление следующих слов в качестве идентификаторов в JSON-кодированных данных приведет к ошибкам, поскольку они зарезервированы в языке JavaScript:

```
RESERVED_WORDS = %w(
  abstract    delete     goto       private      transient
  boolean     do         if         protected    try
  break       double     implements public       typeof
  byte        else       import     return       var
  case        enum       in         short        void
  catch       export     instanceof static       volatile
  char        extends    int        super        while
  class       final      interface  switch       with
  const       finally    long       synchronized
  continue    float      native     this
  debugger    for        new        throw
  default     function   package    throws
)
```

Атрибуты модуля

unquote_hash_key_identifiers

Если этот атрибут равен `true`, то метод `to_json` объекта `Hash` опускает заключение строк или символов в кавычки в ключах при условии, что получающийся ключ – допустимый идентификатор JavaScript. Отметим, что, строго говоря, такой JSON-код некорректен (предполагается, что все ключи в объектах должны быть заключены в кавычки), поэтому если вы стремитесь к строгому соответствию спецификации JSON, задайте для этого атрибута значение `false`:

```
ActiveSupport::JSON.unquote_hash_key_identifiers = false
```

Методы класса

decode(json)

Преобразует строку JSON в объект Ruby. Декодирование выполняется с промежуточным преобразованием в формат YAML, который с точки зрения синтаксиса очень близок к JSON.

Возбуждает исключение `ParseError`, если задан недопустимый JSON-код.

encode(object)

Преобразует объект Ruby в строку JSON:

```
>> print ActiveSupport::JSON.encode(:drink => "too much")
{drink: "too much" }
```

На практике перекодирование моделей `ActiveRecord` в формат JSON может представлять серьезные трудности, так как наличие ассоциаций приводит к циклическим зависимостям:

```
ActiveSupport::JSON::CircularReferenceError: object references itself
```

Возможное решение – написать специальные классы Ruby, содержащие только данные, которые нужно сериализовать.

reserved_word?(word)

Возвращает `true`, если `word` – зарезервированное слово JavaScript, что приводит к проблемам при обработке JSON-кодированных данных.

valid_identifier?(str)

Возвращает `true`, если `str` – допустимый идентификатор JSON (включает проверку зарезервированности слова).

Kernel

Методы, добавленные в класс Ruby `Kernel`, доступны в любом контексте.

Открытые методы экземпляра

daemonize

Преобразует текущий сценарий в процесс-демон, неассоциированный с консолью. Для завершения ему нужно послать сигнал `TERM`.

Приведенный ниже исходный код понятнее любых пояснений:

```
def daemonize
  exit if fork             # Родитель завершается, потомок продолжает работать
  Process.setsid           # Стать лидером сеанса
  exit if fork             # Прибить лидера сеанса
  Dir.chdir "/"            # Сменить старый рабочий каталог
  File.umask 0000          # Установить разумную маску umask
  STDIN.reopen "/dev/null"       # Освободить дескрипторы файлов и...
  STDOUT.reopen "/dev/null", "a" # направить их на что-то разумное.
  STDERR.reopen STDOUT     # TODO: лучше переадресовать на файл протокола

  trap("TERM") { exit }
end
```

debugger

Начинает сеанс отладки, если загружен модуль `ruby-debug`. Вызывает сценарий `script/server –debugger` для запуска Mongrel с отладчиком (только в Rails 2.0).

enable_warnings {...}

Устанавливает переменную $VERBOSE в true на время работы данного блока и восстанавливает исходное значение по выходе.

require_library_or_gem

Требует библиотеку, а в случае ее отсутствия – gem-пакет. Предупреждения во время загрузки библиотеки подавляются, чтобы не отвлекать от предупреждений, выдаваемых приложением.

silence_stream(stream) { ... }

Подавляет вывод в любые потоки на время работы блока:

```
silence_stream(STDOUT) do
  puts 'Это не будет видно'
end

puts 'А это будет'
```

silence_warnings { ... }

Устанавливает переменную $VERBOSE в false на время работы данного блока и восстанавливает исходное значение по выходе.

suppress(*exception_classes) { ... }

Этот метод лучше было бы назвать swallow (глотать). Подавляет возбуждение исключений указанных классов внутри блок. Применять с осторожностью.

Logger

Расширение встроенного в Ruby объекта протоколирования, который с помощью свойства logger доступен в различных контекстах Rails, в том числе в моделях ActiveRecord и классах контроллера. Доступен в любом месте с помощью константы RAILS_DEFAULT_LOGGER. Работа с объектом logger подробно рассмотрена в главе 1.

Если хотите воспользоваться стандартным форматером протоколов, определенным в ядре Ruby, установите форматер для объекта logger следующим образом:

```
logger.formatter = Formatter.new
```

Затем можно задать и другие свойства, например формат даты:

```
logger.datetime_format = "%Y-%m-%d"
```

Открытые методы экземпляра

around_debug(start_message, end_message) { ... }

Упрощает всем известный прием – размещение отладочных комментариев до и после некоторого участка кода:

```
logger.debug "Начало рендеринга компонента (#{options.inspect}): "
result = render_component_stuff(...)
logger.debug "\n\nКонец рендеринга компонента"
result
```

С помощью `around_debug` то же самое можно написать следующим образом:

```
around_debug "Начало рендеринга компонента (#{options.inspect}):",
             "Конец рендеринга компонента" do
  render_component_stuff(...)
end
```

around_error, around_fatal, and around_info

То же, что `around_debug`, но с другим уровнем протоколирования.

datetime_format

Получает текущий формат даты для протокола. Возвращает `nil`, если форматер не поддерживает форматирования дат.

datetime_format=(datetime_format)

Задает форматную строку, которая передается методу `strftime` для генерации временных штампов в протоколе.

formatter

Получает текущий форматер. По умолчанию в Rails применяется `SimpleFormatter`, который просто выводит в протокол сообщение.

silence(temporary_level = Logger::ERROR)

«Приглушает» `logger` на время выполнения блока:

```
RAILS_DEFAULT_LOGGER.silence do
# Какая-то особо говорливая (или секретная) операция
end
```

Module

Расширения класса Ruby `Module`, доступные в любом контексте.

Открытые методы экземпляра

alias_attribute(new_name, old_name)

Этот исключительно полезный метод позволяет без труда создавать синонимы для атрибутов, включая методы чтения, записи и опроса.

В следующем примере класс `Content` является базовым для `Email` в механизме наследования с одной таблицей (STI), но у почтовых сообщений имеется тема (subject), а не заголовок (title):

```
class Content < ActiveRecord::Base
  # имеет колонку 'title'
end

class Email < Content
  alias_attribute :subject, :title
end
```

В результате выполнения `alias_attribute` атрибуты `title` и `subject` стали взаимозаменяемы:

```
>> e = Email.find(:first)

>> e.title
=> "Superstars"

>> e.subject
=> "Superstars"

>> e.subject?
=> true

>> e.subject = "Megastars"
=> "Megastars"

>> e.title
=> "Megastars"
```

alias_method_chain(target, feature)

Инкапсулирует следующий распространенный прием:

```
alias_method :foo_without_feature, :foo
alias_method :foo, :foo_with_feature
```

Метод `alias_method_chain` позволяет получить оба синонима, написав всего одну строчку кода:

```
alias_method_chain :foo, :feature
```

Для методов с вопросительным или восклицательным знаком в конце имени пунктуация сохраняется. Следующий вызов:

```
alias_method_chain :foo?, :feature
```

эквивалентен вызовам:

```
alias_method :foo_without_feature?, :foo?
alias_method :foo?, :foo_with_feature?
```

as_load_path

Возвращает путь загрузки, соответствующий данному модулю.

attr_accessor_with_default(sym, default = nil, &block)

Объявляет акцессор атрибута и возвращаемое по умолчанию значение.

Чтобы присвоить атрибуту `:age` начальное значение `25`, следует написать:

```
class Person
  attr_accessor_with_default :age, 25
end
```

Если у атрибута `:element_name` должно быть значение по умолчанию, динамически вычисляемое в контексте `self`, напишите:

```
attr_accessor_with_default(:element_name) { name.underscore }
```

attr_internal

Синоним `attr_internal_accessor`.

attr_internal_accessor(*attrs)

Объявляет атрибуты, за которыми стоят имена *внутренних* переменных экземпляра (используя соглашение об именах, начинающихся с `@_`). По существу, это механизм для более строгого контроля доступа к секретным атрибутам.

Например, метод `copy_instance_variables_from` из класса `Object` не будет копировать внутренние переменные экземпляра.

attr_internal_reader(*attrs)

Объявляет метод чтения атрибута, за которым стоит переменная экземпляра с внутренним именем.

attr_internal_writer(*attrs)

Объявляет метод записи атрибута, за которым стоит переменная экземпляра с внутренним именем.

const_missing(class_id)

Ruby обращается к обратному вызову const_missing, когда не может найти указанную константу в текущей области видимости. Именно на этом основан механизм автозагрузки классов в Rails. Дополнительную информацию см. в описании модуля Dependencies.

delegate(*methods)

Метод класса delegate позволяет раскрывать методы агрегированных объектов как свои собственные. Передайте один или несколько методов (в виде строк или символов), а в последнем параметре :to – имя целевого объекта (тоже в виде строки или символа). Требуется задать хотя бы один метод и параметр :to.

Делегирование особенно полезно в сочетании с ассоциациями ActiveRecord:

```
class Greeter < ActiveRecord::Base
  def hello
    "hello"
  end

  def goodbye
    "goodbye"
  end
end

class LazyFoo < ActiveRecord::Base
  belongs_to :greeter
  delegate :hello, :to => :greeter
end
```

Допускается задавать несколько делегатов для одного и того же целевого объекта:

```
class Foo < ActiveRecord::Base
  belongs_to :greeter
  delegate :hello, :goodbye, :to => :greeter
end
```

deprecate(*method_names)

Объявляет метод устаревшим. Дополнительную информацию см. в описании модуля Deprecation.

included_in_classes

Возвращает список классов, в которые включен данный модуль. Использует библиотеку Ruby ObjectSpace.

local_constants

Возвращает список констант, локально определенных в данном объекте, но не в его предках. Этот метод может пропускать некоторые константы, если их определение в предке идентично определению в данном объекте.

mattr_accessor(*syms)

Определяет один или несколько методов чтения и записи атрибутов в стиле встроенных акцессоров `attr*` для атрибутов экземпляра.

mattr_reader(*syms)

Определяет один или несколько методов чтения атрибутов.

mattr_writer(*syms)

Определяет один или несколько методов записи атрибутов.

parent

Возвращает модуль, содержащий данный; для корневых модулей, например `::MyModule`, возвращает `Object`.

parents

Возвращает упорядоченный список всех предков данного модуля – от самого близкого до самого далекого. Сам модуль в список не включается.

unloadable(const_desc = self)

Помечает данную константу как выгружаемую, то есть подлежащую удалению при каждой очистке зависимостей (см. описание метода `unloadable` в классе `Object`).

MissingSourceFile

Исключение `LoadError`, возбуждаемое, когда основанный на именах механизм загрузки классов Rails не может найти класс (объяснение того, как Rails ищет и загружает классы, см. в разделе «Rails, модули и код автозагрузки» главы 1).

Multibyte::Chars (в ActiveSupport)

Метод `chars` позволяет прозрачно работать с многобайтными кодировками в классе Ruby `String`, не обладая обширными познаниями в области кодирования текста.

Объект Chars принимает в момент инициализации строку и замещает методы класса String безопасным относительно кодировки способом. Объект Chars замещает все стандартные методы String и может быть получен с помощью метода chars. Методы, которые обычно возвращают объект типа String, теперь возвращают объект Chars, чтобы можно было безопасно производить сцепление вызовов:

```
>> "The Perfect String".chars.downcase.strip.normalize
=> "the perfect string"
```

Объекты Chars взаимозаменяемы с объектами String при условии, что не производится явная проверка типа класса. Если какие-то методы проверяют класс явно, вызывайте метод to_s до передачи им объектов Chars, чтобы вернуться к стандартному объекту String:

```
bad.explicit_checking_method("T".chars.downcase.to_s)
```

Сами операции над строками делегируются обработчикам. Теоретически можно реализовать обработчики для любой кодировки, но по умолчанию обрабатывается кодировка UTF-8. Этот обработчик устанавливается на этапе инициализации.

Отметим, что некоторые методы определены в самом классе Chars, а не в обработчике, поскольку они принадлежат классам Object или Kernel, и механизм method_missing (применяемый для делегирования) не может их перехватить.

Методы класса

handler=(klass)

Если вы хотите реализовать собственный обработчик или воспользоваться сторонним, можете задать его для класса Chars вручную:

```
ActiveSupport::Multibyte::Chars.handler = MyHandler
```

Пример реализации разработчика смотрите в исходном тексте класса UTF8Handler. Если вы будете реализовывать обработчик для кодировки, отличной от UTF-8, то, вероятно, захотите также переопределить метод handler в классе Chars.

Открытые методы экземпляра

<=> (other)

Возвращает −1, 0 или +1, если данный объект Chars соответственно меньше, равен или больше объекта в правой части оператора сравнения. Иными словами, работает, как вы и ожидаете.

=~ (other)

Работает, как соответствующий метод в классе String, но возвращает смещение, отсчитываемое в символах (кодовых позициях), а не в байтах.

gsub(*a, &b)

Работает аналогично методу `gsub` для обычных строк.

handler

Возвращает подходящий обработчик для инкапсулированной строки, зависящий от значения `$KCODE` и кодировки строки. Этот метод используется внутри Rails для перенаправления сообщений нужным классам в зависимости от контекста.

method_missing(m, *a, &b)

Пытается переадресовать вызовы всех неопределенных методов указанному обработчику. Если метод не определен и в обработчике, посылает его самой инкапсулированной строке. Также делает методы с восклицательным знаком в конце деструктивными, поскольку обработчик не способен изменить инкапсулированный экземпляр строки.

respond_to?(method)

Делает возможной динамическую типизацию.

split(*args)

Работает аналогично методу `split` в классе `String`, но в результирующий список помещает экземпляры `Chars`, а не `String`. Это упрощает сцепление вызовов.

string

Возвращает инкапсулированный экземпляр `String`. С ним не следует выполнять никаких операций через объект `Chars`.

NilClass

Напомним, что в Ruby объектами является все, даже `nil`; это особая ссылка на единственный экземпляр класса `NilClass`.

Помимо добавления метода `blank?`, расширения `nil` пытаются возбуждать исключения с более понятными сообщениями, чтобы помочь начинающим пользователям Rails. Идея в том, чтобы в случае непреднамеренного вызова метода для `nil` выдавать не сообщение с упоминанием ошибки `NoMethodError` и имени какого-то метода среды, а информацию об ожидаемом типе объекта. В кругу посвященных такое поведение в шутку называют *хнычущий nil* (whiny nil).

При перехвате методов, ошибочно вызванных для объекта `nil`, применяется механизм `method_missing`. Чтобы можно было выдать осмысленную рекомендацию, имя метода ищется в хеше, сопоставляющем имена методов классам Rails.

Если вы когда-нибудь программировали в Rails, то, наверное, знакомы с результатом этой процедуры по сообщению, которое сопровождает исключение `NoMethodError`:

> *You have a `nil` object when you didn't expect it! You might have expected an instance of `class_name`. The error occurred while evaluating `nil.method_name`.*
>
> *Не хотели, а получили объект `nil`! Наверное, вы ожидали увидеть экземпляр `class_name`. Ошибка произошла при вызове `nil.method_name`.*

Поведением `whiny nil` в различных режимах можно управлять, задав в конфигурационном файле такую строку:

```
config.whiny_nils = true
```

По умолчанию этот параметр равен `true` в режиме разработки и `false` в режиме эксплуатации.

Открытые методы экземпляра

blank?

Всегда возвращает `true`.

id

Возбуждает исключение примерно с таким сообщением: `Called id for nil, which would mistakenly be 4 -- if you really wanted the id of nil, use object_id` (вызван метод `id` для `nil`, который вернет 4, но это неправильно. Если вы действительно хотите узнать `id` объекта `nil`, пользуйтесь методом `object_id`).

Numeric

Как и для класса `Hash`, ActiveSupport добавляет непосредственно в класс `Numeric` только метод `blank?`.

Открытые методы экземпляра

blank?

Всегда возвращает `false`.

Numeric::Bytes (в ActiveSupport::CoreExtensions)

Применяется для вычислений с байтами, например:

`45.bytes + 2.6.megabytes`.

Открытые методы экземпляра

byte и bytes

Возвращает значение `self`.

kilobyte и kilobytes

Возвращает `self * 1024`.

megabyte и megabytes

Возвращает `self * 1024.kilobytes`.

gigabyte и gigabytes

Возвращает `self * 1024.megabytes`.

terabyte и terabytes

Возвращает `self * 1024.gigabytes`.

petabyte и petabytes

Возвращает `self * 1024.terabytes`.

exabyte и exabytes

Возвращает `self * 1024.petabytes`[1].

Numeric::Time (в ActiveSupport::CoreExtensions)

Позволяет удобно выполнять вычисления со временем, выраженным в секундах, например:

```
1.minute + 45.seconds == 105.seconds #=> true
```

Методы, собранные в этом модуле, пользуются методом `advance` из класса `Time` для точных вычислений с датами, а также для операций сложения и вычитания с объектами `Time`:

```
# эквивалентно Time.now.advance(:months => 1)
1.month.from_now
```

[1] Согласно исследованию IDC, заказанному компанией – производителем устройств внешней памяти EMC, в 2006 году было создано и скопировано 161 экзабайт цифровой информации. Один экзабайт равен миллиарду гигабайт. Эксперты IDC ожидают, что в 2010 году объем созданной и скопированной за год информации возрастет в шесть раз и достигент 988 экзабайт.

```
# эквивалентно Time.now.advance(:years => 2)
2.years.from_now

# эквивалентно Time.now.advance(:months => 4, :years => 5)
(4.months + 5.years).from_now
```

Хотя в приведенных выше примерах эти методы производят точные вычисления, следует принимать во внимание возможную потерю точности при приведении к целочисленным типам. Для более точных вычислений с датами и временем пользуйтесь встроенными в Ruby классами `Date` и `Time`.

Открытые методы экземпляра

ago и until

Прибавляет числовое значение, чтобы обозначить момент времени в прошлом.

```
10.minutes.ago
```

day и days

Промежуток времени, равный `self * 24.hours`.

fortnight и fortnights

Промежуток времени, равный `self * 2.weeks`.

from_now(time = Time.now) и since(time = Time.now)

Момент времени в будущем, отсчитываемый от указанного момента (по умолчанию `Time.now`).

hour и hours

Промежуток времени, равный `self * 3600.seconds`.

minute и minutes

Промежуток времени, равный `self * 60.seconds`.

month и months

Промежуток времени, равный `self * 30.days`.

second и seconds

Промежуток времени в секундах, равный `self`.

week и weeks

Промежуток времени, равный `self * 7.days`.

year и years

Промежуток времени, равный `self * 365.25.days`.

Object

Rails подмешивает в класс `Object` несколько методов, которые становятся доступны любому объекту во время выполнения.

Открытые методы экземпляра

`(command)

Определяет метод «обратные кавычки» так, что он ведет себя примерно одинаково на различных платформах. В `win32` попытка выполнить несуществующую команду возбуждает исключение `Errno::ENOENT`; в UNIX запущенная оболочка печатает сообщение на `STDERR` и устанавливает переменную `$?`. Переопределенный метод в `win32` эмулирует поведение Unix в первом, но не во втором отношении, то есть только выводит сообщение на `STDERR`.

acts_like?(duck)

Вспомогательный метод для динамической типизации с очень простой реализацией:

```
def acts_like?(duck)
  respond_to? "acts_like_#{duck}?"
end
```

`ActiveSupport` расширяет класс `Date`, добавляя метод `acts_like_date?`, и класс `Time`, добавляя метод `acts_like_time?`. Поэтому для выполнения динамически типизированных сравнений можно писать `x.acts_like?(:time)` и `y.acts_like?(:date)`, поскольку если мы хотим, чтобы некоторый класс мог работать как `Time`, достаточно определить метод `acts_like_time?`, возвращающий `true`.

blank?

Пустая строка (""), строка, содержащая только пробельные символы (" "), `nil`, пустой массив ([]) и пустой хеш ({}) считаются «пустыми» (blank) объектами.

Работает, вызывая метод strip (для удаления пробельных символов), если таковой имеется, а затем – метод empty?. Если метода empty? нет, возвращает логическое отрицание self.

copy_instance_variables_from(object, exclude = [])

Полезен для копирования переменных экземпляра из одного объекта в другой.

extended_by

Возвращает массив модулей, являющихся предками данного объекта (включенными в состав ancestors). Для иллюстрации приведу список модулей, включенных в класс Person из одного моего реального проекта. Не ожидали, что будет так много?

```
>> Person.find(:first).extended_by.sort_by(&:name)
=> [ActiveRecord::Acts::List, ActiveRecord::Acts::NestedSet,
ActiveRecord::Acts::Tree, ActiveRecord::Aggregations,
ActiveRecord::Associations, ActiveRecord::AttributeMethods,
ActiveRecord::Calculations, ActiveRecord::Callbacks,
ActiveRecord::Locking::Optimistic, ActiveRecord::Locking::Pessimistic,
ActiveRecord::Observing, ActiveRecord::Reflection,
ActiveRecord::Timestamp, ActiveRecord::Transactions,
ActiveRecord::Validations, ActiveRecord::XmlSerialization, Base64,
Base64::Deprecated, ERB::Util, GeoKit::ActsAsMappable, LatLongZoom,
PP::ObjectMixin, PhotosMixin, Reloadable::Deprecated,
ScottBarron::Acts::StateMachine, UJS::BehaviourHelper, UJS::Helpers,
UJS::JavascriptProxies, WhiteListHelper, WillPaginate::Finder]
```

extend_with_included_modules_from(object)

Вызывает метод extend для данного объекта, последовательно передавая каждый модуль, включаемый объектом, который передается в аргументе object. Реализация предельно проста:

```
def extend_with_included_modules_from(object)
  object.extended_by.each { |mod| extend mod }
end
```

instance_exec(*arguments, &block)

Метод instance_exec позволяет (довольно эффективно) выполнить блок кода на Ruby в контексте другого объекта.

```
>> t = Tag.find(:first)
=> #<Tag id: 1, name: “politics”>
>> t.instance_exec { name }
=> “politics”
```

instance_values

Возвращает переменные экземпляр данного объекта в виде хеша:

```
>> Tag.find(:first).instance_values
=> {"attributes" => {"name" => "politics", "id" => "1"}}
```

load(file, *extras)

Rails переопределяет встроенный в Ruby метод `load`, увязывая его с подсистемой `Dependencies`.

require(file, *extras)

Rails переопределяет встроенный в Ruby метод `require`, увязывая его с подсистемой `Dependencies`.

require_association(file_name)

Используется в Rails для внутренних надобностей. Вызывает `Dependencies.associate_with (file_name)`.

require_dependency(file_name)

Используется в Rails для внутренних надобностей. Вызывает `Dependencies.depend_on(file_name)`.

require_or_load(file_name)

Используется в Rails для внутренних надобностей. Вызывает `Dependencies.require_or_load(file_name)`.

returning(value) { ... }

Переведенная на Ruby реализация K-комбинатора, которым мы обязаны Микаэлю Брокману (Mikael Brockman). Полезна, когда вы знаете, что хотите вернуть некий объект, но сначала с ним нужно что-то сделать, например:

```
def foo
  returning values = [] do
    values << 'bar'
    values << 'baz'
  end
end

foo # => ['bar', 'baz']
```

Несколько более элегантный способ доступа к возвращаемому значению дает блок. Вот тот же пример, где в качестве `values` используется блок:

```
def foo
  returning [] do |values|
    values << 'bar'
    values << 'baz'
  end
end

foo # => ['bar', 'baz']
```

unloadable(const_desc)

Помечает указанную константу как *выгружаемую*. Выгружаемые константы удаляются при каждой очистке зависимостей.

Отметим, что помечать константу как выгружаемую нужно только один раз. В сценариях настройки или инициализации можно перечислить все подлежащие выгрузке константы, и тогда они будут удаляться при всех последующих вызовах `Dependencies.clear`, а не только при первом.

Дескриптор константы (аргумент `const_desc`) может быть (не анонимным) модулем, классом или квалифицированным именем константы в виде строки или символа.

Возвращает `true`, если константа ранее не была помечена как выгружаемая, `false` – в противном случае.

with_options(options)

Элегантный способ вынести общие опции:

```
with_options(:class_name => 'Comment', :order => 'id desc') do |post|
  post.has_many :approved, :conditions => ['approved = ?', true]
  post.has_many :unapproved, :conditions => ['approved = ?', false]
  post.has_many :all_comments
end
```

Можно также использовать, явно указывая объект, который будет передан как параметр блока:

```
map.with_options :controller => "people" do |people|
  people.connect "/people", :action => "index"
  people.connect "/people/:id", :action => "show"
end
```

OrderedHash (в ActiveSupport)

Реализация хеша как подкласса встроенного в Ruby класса `Array`. В отличие об обычных хешей Ruby, сохраняет порядок элементов. Этот класс определен в пространстве имен во избежание конфликтов с другими реализациями. Если вы не хотите постоянно писать полностью

квалифицированное имя, можете отнести его к пространству имен верхнего уровня:

```
OrderedHash = ActiveSupport::OrderedHash
```

Реализован обычный оператор [], но в остальном это Array:

```
>> oh = ActiveSupport::OrderedHash.new
=> []
>> oh[:one] = 1
=> 1
>> oh[:two] = 2
=> 2
>> oh[:three] = 3
=> 3
>> oh
=> [[:one, 1], [:two, 2], [:three, 3]]
```

OrderedOptions (в ActiveSupport)

Подкласс OrderedHash, добавляющий реализацию метода method_missing так, чтобы элементы хеша можно было читать и изменять, пользуясь обычной семантикой атрибутов, отделяемых точками:

```
def method_missing(name, *args)
  if name.to_s =~ /(.*)=$/
    self[$1.to_sym] = args.first
  else
    self[name]
  end
end
```

Мелочи Rails: файл initializer.rb содержит точную копию этого класса, только в пространстве имен Rails. Почему? Потому что он необходим еще до загрузки ActiveSupport, на этапе инициализации.

Proc

Расширения встроенного в Ruby класса Proc, воплощающие в жизнь магию метода instance_exec.

Открытые методы экземпляра

bind(object) { ... }

Реализует привязку Proc-объекта к произвольному объекту таким образом, что при вызове он исполняется в контексте последнего. Именно этот механизм лежит в основе работы метода instance_exec из класса Object.

В следующем примере мы сначала проверяем, что имя name не определено в текущем контексте. Затем создаем Proc-объект, который вызывает name, и убеждаемся, что при его вызове методом call он по-прежнему возбуждает исключение NameError:

```
>> name
NameError: undefined local variable or method `name' ...

>> p = Proc.new { name }
=> #<Proc:0x031bf5b4@(irb):15>

>> p.call
NameError: undefined local variable or method `name' ...
```

Далее мы используем метод bind для привязки Proc-объекта к контексту двух разных объектов, в которых метод name определен:

```
>> p.bind(Person.find(:first)).call
=> "Admin"

>> p.bind(Tag.find(:first)).call
=> "politics"
```

Реализация устроена достаточно хитро: сначала в конечном объекте определяется новый метод с уникальным сгенерированным именем, его телом является заданный Proc-объект. Затем сохраняется ссылка на новый экземпляр класса Method, а из конечного объекта он удаляется с помощью remove_method. Наконец, конечный объект привязывается к новому методу и возвращается, так что вызов call выполняет Proc-объект в контексте конечного объекта.

Range

Расширения встроенного в Ruby класса Range.

Константы

Константа DATE_FORMATS содержит единственный Proc-объект, применяемый для преобразования диапазона в выражение SQL:

```
DATE_FORMATS = {
  :db => Proc.new {|start, stop|
    "BETWEEN '#{start.to_s(:db)}' AND '#{stop.to_s(:db)}'"
  }
}
```

Открытые методы экземпляра

to_formatted_s(format = :default)

Генерирует строковое представление диапазона:

```
>> (20.days.ago..10.days.ago).to_formatted_s
=> "Fri Aug 10 22:12:33 -0400 2007..Mon Aug 20 22:12:33 -0400 2007"
>> (20.days.ago..10.days.ago).to_formatted_s(:db)
=> "BETWEEN '2007-08-10 22:12:36' AND '2007-08-20 22:12:36'"
```

String

Расширения встроенного в Ruby класса `String`.

Открытые методы экземпляра

at(position)

Возвращает символ в позиции `position`, при этом строка рассматривается как массив, в котором первый символ находится в позиции 0. Возвращает `nil`, если позиция оказывается за пределами строки:

```
"hello".at(0) # => "h"
"hello".at(4) # => "o"
"hello".at(10) # => nil
```

blank?

Возвращает результат метода `empty?` (после удаления начальных и конечных пробелов).

first(number)

Возвращает первые `number` символов строки.

from(position)

Возвращает остаток строки, начиная с позиции `position`, при этом строка рассматривается как массив, в котором первый символ находится в позиции 0. Возвращает `nil`, если позиция оказывается за пределами строки:

```
"hello".at(0) # => "hello"
"hello".at(2) # => "llo"
"hello".at(10) # => nil
```

last(number)

Возвращает последние `number` символов строки:

```
"hello".last # => "o"
"hello".last(2) # => "lo"
"hello".last(10) # => "hello"
```

to(position)

Возвращает часть строки от начала до позиции `position`, при этом строка рассматривается как массив, в котором первый символ находится в позиции 0. Не возбуждает исключение, если `position` превышает длину строки:

```
"hello".at(0) # => "h"
"hello".at(2) # => "hel"
"hello".at(10) # => "hello"
```

to_date

Использует `ParseDate.parsedate` для преобразования строки в объект `Date`.

to_datetime

Использует `ParseDate.parsedate` для преобразования строки в объект `DateTime`.

to_time(form = :utc)

Использует `ParseDate.parsedate` для преобразования строки в объект `Time` в часовом поясе `:utc` (по умолчанию) или `:local`.

String::Inflections (в ActiveSupport::CoreExtensions)

В модуле `String::Inflections` определены новые методы класса `String`, необходимые для различных трансформаций имен.

Например, можно построить имя базы данных по имени класса:

```
"ScaleScore".tableize => "scale_scores"
```

Если вы считаете, что лингвистические возможности Rails чересчур ограничены, попробуйте отличную библиотеку Linguistics Майкла Грейнджера (Michael Granger), расположенную по адресу *http://www.deveiate.org/projects/Linguistics*. Она не умеет выполнять все виды словоизменения, которые поддерживает Rails, но то, что умеет, делает лучше (см., например, метод `titleize`).

Открытые методы экземпляра

camelize(first_letter = :upper)

По умолчанию метод `camelize` преобразует строки в ВерблюжьюНотацию. Если задан аргумент `:lower`, порождается строка в верблюжьей-

Нотации. Кроме того, метод `camelize` преобразует / в `::`, что полезно для трансформации путей в пространства имен:

```
"active_record".camelize #=> "ActiveRecord"
"active_record".camelize(:lower) #=> "activeRecord"
"active_record/errors".camelize #=> "ActiveRecord::Errors"
"active_record/errors".camelize(:lower) #=> "activeRecord::Errors"
```

classify

Создает имя класса по имени таблицы. Используется в `ActiveRecord` для преобразования имен таблиц в классы моделей. Отметим, что метод `classify` возвращает строку, а не `Class` (для преобразования в настоящий класс после `classify` выполните еще и метод `constantize`):

```
"egg_and_hams".classify #=> "EggAndHam"
"post".classify #=> "Post"
```

constantize

Метод `constantize` пытается найти объявленную константу с именем, заданным в данной строке. Возбуждает исключение `NameError`, если подходящая константа не найдена.

```
"Module".constantize #=> Module
"Class".constantize #=> Class
```

dasherize

Заменяет подчерки дефисами:

```
"puni_puni" #=> "puni-puni"
```

demodulize

Удаляет все префиксы модулей из полностью квалифицированного имени модуля или класса:

```
>> "ActiveRecord::CoreExtensions::String::Inflections".demodulize
=> "Inflections"
"Inflections".demodulize #=> "Inflections"
```

foreign_key(separate_class_name_and_id_with_underscore = true)

Строит имя внешнего ключа по имени класса:

```
"Message".foreign_key #=> "message_id"
"Message".foreign_key(false) #=> "messageid"
"Admin::Post".foreign_key #=> "post_id"
```

humanize

Преобразует первую букву строки в заглавную, подчерки – в пробелы и убирает суффикс `_id`. Аналогичен методу `titleize`, который служит для вывода строки в удобочитаемом виде:

```
"employee_salary" #=> "Employee salary"
"author_id" #=> "Author"
```

pluralize

Возвращает слово во множественном числе:

```
"post".pluralize #=> "posts"
"octopus".pluralize #=> "octopi"
"sheep".pluralize #=> "sheep"
"words".pluralize #=> "words"
"the blue mailman".pluralize #=> "the blue mailmen"
"CamelOctopus".pluralize #=> "CamelOctopi"
```

singularize

Противоположен методу `pluralize`; возвращает слово в единственном числе:

```
"posts".singularize #=> "post"
"octopi".singularize #=> "octopus"
"sheep".singluarize #=> "sheep"
"word".singluarize #=> "word"
"the blue mailmen".singularize #=> "the blue mailman"
"CamelOctopi".singularize #=> "CamelOctopus"
```

tableize

Создает имя таблицы базы данных, строя множественное число и разделяя слова подчерками, как принято в Rails. Используется в `ActiveRecord` при конструировании имени таблицы для класса модели. К последнему слову в строке применяется метод `pluralize`:

```
"RawScaledScorer".tableize #=> "raw_scaled_scorers"
"egg_and_ham".tableize #=> "egg_and_hams"
"fancyCategory".tableize #=> "fancy_categories"
```

titlecase

Синоним `titleize`.

titleize

Начинает все слова с заглавной буквы и заменяет некоторые символы в строке для создания удобочитаемого заголовка. Метод `titleize` предназначен только для «украшения» и не используется внутри Rails:

```
>> "The light on the beach was like a sinus headache".titleize
=> "The Light On The Beach Was Like A Sinus Headache"
```

Но и это не идеально. С заглавной буквы начинаются все слова в строке, в том числе и те, что не должны бы, например артикли «a» и «the». Кроме того, возникают проблемы с апострофами:

```
>> "Her uncle's cousin's record albums".titleize
=> "Her Uncle'S Cousin'S Record Albums"
```

Упомянутый в начале этого раздела gem-пакет `Linguistics` содержит метод `proper_noun`, который, по моему опыту, работает гораздо лучше, чем `titleize`:

```
>> "Her uncle's cousin's record albums".en.proper_noun
=> "Her Uncle's Cousin's Record Albums"
```

underscore

Противоположен методу `camelize`. Преобразует строку в форму с подчерками. Изменяет `::` на `/` для преобразования пространств имен в пути:

```
"ActiveRecord".underscore #=> "active_record"
"ActiveRecord::Errors".underscore #=> active_record/errors
```

String::Iterators (в ActiveSupport::CoreExtensions)

Содержит специализированный строковый итератор, которым можно пользоваться для последовательных операций над каждым символом с учетом кодировки Unicode.

Открытые методы экземпляра

each_char { |char| ... }

Передает в блок по одному символу из строки. Если константа `$KCODE` равна `'UTF8'`, то многобайтные символы передаются корректно.

String::StartsEndsWith (в ActiveSupport::CoreExtensions)

Добавляет в класс `String` ряд методов для проверки условий.

Открытые методы экземпляра

starts_with?(prefix)

Возвращает `true`, если строка начинается с указанного префикса `prefix`.

ends_with?(suffix)

Возвращает `true`, если строка заканчивается указанным суффиксом `suffix`.

String::Unicode (в ActiveSupport::CoreExtensions)

Определяет методы для работы со строками в кодировке Unicode.

Открытые методы экземпляра

chars

Метод `chars` возвращает экземпляр `ActiveSupport::Multibyte::Chars` — безопасного относительно Unicode прокси-класса, инкапсулирующего исходную строку. В объекте прокси-класса определены Unicode-версии всех методов класса `String`, поэтому вы можете быть уверены, что не испортите данные в строке.

Неопределенные методы переадресуются классу `String`, поэтому все переопределенные методы строки тоже можно спокойно вызывать через прокси-объект, возвращенный методом `chars`:

Примеры:

```
name = 'Claus Muller'
name.reverse #=> "rell??M sualC" # запорчено!!
name.length #=> 13 # неправильно!!
name.chars.reverse.to_s #=> "relluM sualC"
name.chars.length #=> 12
```

Методы, которые в классе `String` возвращают строку, в прокси-классе возвращают объект типа `Chars`. Это позволяет сцеплять вызовы методов:

```
name.chars.reverse.length #=> 12
```

Прокси-класс `Char` старается быть максимально взаимозаменяемым с классом `String`: сортировка и сравнение объектов `String` и `Chars` работают без неожиданностей. Методы с восклицательным знаком в конце изменяют внутреннюю строку, инкапсулированную в объекте `Chars`. Проблемы интероперабельности можно легко разрешить с помощью метода `to_s` (дополнительную информацию о методах, определенных в прокси-классе `Chars`, см. в исходном тексте классов `Multibyte::Chars` и `Multibyte::Handlers::UTF8Handler`).

is_utf8?(suffix)

Возвращает `true`, если строка имеет семантику UTF-8, а не просто является потоком байтов.

Symbol

Расширения встроенного в Ruby класса `Symbol`.

Открытые методы экземпляра

to_proc

Синтаксическая глазурь Rails с дурной репутацией. Преобразует символ в простой Proc-объект, что особенно полезно для перечислений:

```
# То же самое, что people.collect { |p| p.name }
people.collect(&:name)
# То же самое, что people.select { |p| p.manager? }.collect { |p| p.salary }
people.select(&:manager?).collect(&:salary)
```

Test::Unit::Assertions

Rails добавляет ряд утверждений к тем, что уже определены в `Test::Unit`.

Открытые методы экземпляра

assert_difference(expressions, difference = 1, message = nil, &block)

Проверяет, равно ли число `difference` разности между результатами вычисления в выражении `expression` до и после выполнения блока (проще показать, чем объяснить).

В следующем примере выражение `Article.count` вычисляется с помощью `eval`, и результат сохраняется. Затем управление передается блоку, который выполняет метод `post :create` и возвращает управление методу `assert_difference`. В этот момент выражение `post :create` вычисляется снова, и проверяется, равна ли разность нового и старого значений 1 (подразумевается по умолчанию):

```
assert_difference 'Article.count' do
  post :create, :article => {...}
end
```

Можно передать и вычислить произвольное выражение:

```
assert_difference 'assigns(:article).comments(:reload).size' do
  post :create, :comment => {...}
end
```

Допускается также произвольное значение разности. По умолчанию подразумевается +1, но можно задавать и отрицательные числа:

```
assert_difference 'Article.count', -1 do
  post :delete, :id => ...
end
```

Можно передавать массив выражений – они будут вычисляться по очереди:

```
assert_difference [ 'Article.count', 'Post.count' ], +2 do
  post :create, :article => {...}
end
```

Также можно задать сообщение об ошибке:

```
assert_difference 'Article.count', -1, "Новость должна быть стерта" do
  post :delete, :id => ...
end
```

assert_no_difference(expressions, message = nil, &block)

Проверяет, что значение, возвращенное переданным выражением, *не* изменяется в результате действий, выполненных внутри блока:

```
assert_no_difference 'Article.count' do
  post :create, :article => invalid_attributes
end
```

Time::Calculations (в ActiveSupport::CoreExtensions)

Расширения встроенного в Ruby класса `Time`.

Методы класса

days_in_month(month, year = nil)

Возвращает количество дней в заданном месяце. Если указан год, то количество дней в феврале определяется с учетом високосности. В противном случае считается, что в феврале 28 дней.

local_time(*args)

Обертывает метод класса `time_with_datetime_fallback`, устанавливая аргумент `utc_or_local` в `:local`.

time_with_datetime_fallback(utc_or_local, year, month=1, day=1, hour=0, min=0, sec=0, usec=0)

Возвращает новый объект `Time`, если указанный год попадает в диапазон, представляемый классом `Time`, то есть 1970..2038 или 1902..2038 в зависимости от архитектуры системы. Если год выходит за пределы этого диапазона, возвращается объект класса `DateTime`.

utc_time(*args)

Обертывает метод класса `time_with_datetime_fallback`, устанавливая аргумент `utc_or_local` в `:utc`.

Открытые методы экземпляра

+ (other)

Реализован с помощью метода `plus_with_duration`. Позволяет складывать объекты, представляющие время:

```
expiration_time = Time.now + 3.days
```

– (other)

Реализован с помощью метода `minus_with_duration`. Позволяет вычитать объекты, представляющие время:

```
two_weeks_ago = Time.now - 2.weeks
```

advance(options)

Обеспечивает точные вычисления с объектами `Time`. Параметр `options` – хеш, который может содержать следующие ключи `:months`, `:days`, `:years`, `:hour`, `:min`, `:sec`, `:usec`.

ago(seconds)

Возвращает новый объект `Time`, представляющий момент времени `seconds` секунд назад; по существу обертка вокруг одноименного расширения класса `Numeric`. Если нужна максимальная точность, не пользуйтесь этим методом в сочетании с `x.months` – применяйте метод `months_ago`!

at_beginning of_day

Синоним `beginning_of_day`.

at_beginning of_month

Синоним `beginning_of_month`.

at_beginning of_week

Синоним `beginning_of_week`.

at_beginning of_year

Синоним `beginning_of_year`.

at_end of_day

Синоним `end_of_day`.

at_end of_month

Синоним `end_of_month`.

at_end of_week

Синоним `end_of_week`.

at_end of_year

Синоним `end_of_year`.

beginning of_day

Возвращает новый объект `Time`, представляющий начало суток для текущего экземпляра. В коде «зашито» значение 00:00.

beginning of_month

Возвращает новый объект `Time`, представляющий начало месяца (первое число месяца, время 00:00).

beginning_of_quarter

Возвращает новый объект `Time`, представляющий начало календарного квартала (первое января, апреля, июля, октября – 00:00).

beginning of_week

Возвращает новый объект `Time`, представляющий начало недели для текущего экземпляра. В коде «зашит» понедельник – 00:00.

beginning of_year

Возвращает новый объект `Time`, представляющий начало года (1 января – 00:00).

change(options)

Возвращает новый объект `Time`, в котором элементы изменены в соответствии с параметрами в хеше `options`. Допустимые значения для даты – `:year`, `:month`, `:day`. Допустимые значения для времени – `:hour`, `:min`, `:sec`, `:offset`, `:start`.

end_of_day

Возвращает новый объект `Time`, представляющий конец суток (23:59:59).

end_of_month

Возвращает новый объект `Time`, представляющий конец месяца (последнее число месяца – 00:00).

last_month

Вспомогательный метод, эквивалентный `months_ago(1)`.

last_year

Вспомогательный метод, эквивалентный `years_ago(1)`.

monday

Синоним `beginning of_week`.

months_ago(months)

Возвращает новый объект `Time`, который представляет момент времени в прошлом, отстоящий на `months` месяцев назад.

months_since(months)

Противоположен методу `months_ago`. Возвращает новый объект `Time`, который представляет момент времени в будущем, отстоящий на `months` месяцев вперед.

next_month

Вспомогательный метод, эквивалентный `months_since(1)`.

next_year

Вспомогательный метод, эквивалентный `years_since(1)`.

seconds_since_midnight

Возвращает число секунд, прошедших с полуночи.

since(seconds)

Возвращает новый объект `Time`, который представляет момент времени в будущем, отстоящий на `seconds` секунд вперед. По существу обертка вокруг одноименного расширения класса `Numeric`. Если нужна максимальная точность, не пользуйтесь этим методом в сочетании с `x.months` – применяйте метод `months_since`!

tomorrow

Вспомогательный метод, эквивалентный `self.since(1.day)`.

years_ago(years)

Возвращает новый объект `Time`, который представляет момент времени в прошлом, отстоящий на `years` лет назад.

years_since(years)

Противоположен методу `years_ago`. Возвращает новый объект `Time`, который представляет момент времени в будущем, отстоящий на `years` лет вперед.

yesterday

Вспомогательный метод, эквивалентный `self.ago(1.day)`.

Time::Conversions (в ActiveSupport::CoreExtensions)

Расширения встроенного в Ruby класса `Time` для преобразования объектов, представляющих время, в удобно отформатированные строки и объекты других типов.

Константы

Хеш `DATE_FORMATS` содержит образцы форматирования, применяемые в методе `to_formatted_s` для преобразования объекта `Time` в строковое представление:

```
DATE_FORMATS = {
  :db            => "%Y-%m-%d %H:%M:%S",
  :time          => "%H:%M",
  :short         => "%d %b %H:%M",
  :long          => "%B %d, %Y %H:%M",
  :long_ordinal  => lambda { |time|
    time.strftime("%B #{time.day.ordinalize}, %Y %H:%M") },
  :rfc822        => "%a, %d %b %Y %H:%M:%S %z"
}
```

Открытые методы экземпляра

to_date

Возвращает новый объект `Date`, построенный по `Time`, но с отбрасыванием части, относящейся ко времени.

to_datetime

Возвращает новый объект `Date`, построенный по `Time`, с сохранением смещения относительно `utc`. По существу является оберткой вокруг фабричного метода `DateTime.civil`:

```
DateTime.civil(year, month, day, hour, min, sec, Rational(utc_offset,
86400), 0)
```

to_formatted_s(format = :default)

Преобразует объект `Time` в строковое представление. Параметр `:default` соответствует собственному методу `to_s` класса `Time`:

```
>> Time.now.to_formatted_s(:long_ordinal)
=> "August 31st, 2007 15:00"
```

to_time

Возвращает `self`.

TimeZone

Значащий объект, представляющий часовой пояс. Часовым поясом называется именованное смещение (в секундах) от гринвичского времени GMT. Отметим, что два часовых пояса считаются эквивалентными, только если они называются одинаково и определяют одно и то же смещение.

Если пользователи приложения разбросаны по всему миру, то время на сервере хранится как UTC, а часовой пояс запоминается в учетной записи пользователя. В этом случае при отображении времени вы можете скорректировать хранящееся на сервере время с учетом часового пояса конкретного пользователя.

Петер Марклунд (Peter Marklund) написал краткое руководство по этой технике, с которым можно ознакомиться на странице *http://www.marklunds.com/articles/one/311*. Обратите внимание на его совет использовать библиотеку `TZInfo` для Ruby – она знает, как обращаться с летним временем, чего сама среда Rails делать не умеет.

В руководстве Петера рассмотрено все: настройка `TZInfo`, включение данных о часовом поясе в класс `User` с помощью метода `composed_of` и вопросы построения интерфейса, в том числе получение информации о часовом поясе с помощью метода-помощника Rails `time_zone_select`.

Константы

`US_ZONES` – это регулярное выражение, которое сопоставляется с именами всех часовых поясов США:

```
US_ZONES = /US|Arizona|Indiana|Hawaii|Alaska/
```

Методы класса

[] (arg)

Ищет заданный объект, представляющий часовой пояс. Если аргумент – строка, она интерпретируется как название часового пояса:

```
>> TimeZone['Dublin']
=> #<TimeZone:0x3208390 @name="Dublin", @utc_offset=0>
```

Если аргумент – число, оно может быть смещением пояса в часах или в секундах. Возвращается первый найденный часовой пояс с таким смещением.

Возвращает `nil`, если запрошенный часовой пояс неизвестен системе.

all

Возвращает массив всех объектов `TimeZone`. Для одного часового пояса часто предусматривается несколько объектов `TimeZone`, чтобы пользователю было проще найти свой пояс. Ниже приведен весь перечень часовых поясов, включенных в класс `TimeZone`:

```
[[-43_200, "International Date Line West" ],
 [-39_600, "Midway Island", "Samoa" ],
 [-36_000, "Hawaii" ],
 [-32_400, "Alaska" ],
 [-28_800, "Pacific Time (US & Canada)", "Tijuana" ],
 [-25_200, "Mountain Time (US & Canada)", "Chihuahua", "La Paz",
           "Mazatlan", "Arizona" ],
 [-21_600, "Central Time (US & Canada)", "Saskatchewan",
           "Guadalajara",
           "Mexico City", "Monterrey", "Central America" ],
 [-18_000, "Eastern Time (US & Canada)", "Indiana (East)", "Bogota",
           "Lima", "Quito" ],
 [-14_400, "Atlantic Time (Canada)", "Caracas", "La Paz", "Santiago" ],
 [-12_600, "Newfoundland" ],
 [-10_800, "Brasilia", "Buenos Aires", "Georgetown", "Greenland" ],
 [ -7_200, "Mid-Atlantic" ],
 [ -3_600, "Azores", "Cape Verde Is." ],
 [      0, "Dublin", "Edinburgh", "Lisbon", "London", "Casablanca",
           "Monrovia" ],
 [  3_600, "Belgrade", "Bratislava", "Budapest", "Ljubljana", "Prague",
           "Sarajevo", "Skopje", "Warsaw", "Zagreb", "Brussels",
           „Copenhagen“, „Madrid“, „Paris“, „Amsterdam“, „Berlin“,
           „Bern“, „Rome“, „Stockholm“, „Vienna“,
           „West Central Africa“ ],
 [  7_200, „Bucharest“, „Cairo“, „Helsinki“, „Kyev“, „Riga“, „Sofia“,
           „Tallinn“, „Vilnius“, „Athens“, „Istanbul“, „Minsk“,
           „Jerusalem“, „Harare“, „Pretoria“ ],
 [ 10_800, „Moscow“, „St. Petersburg“, „Volgograd“, „Kuwait“,
           „Riyadh“,
           „Nairobi“, „Baghdad“ ],
```

```
[ 12_600, „Tehran“ ],
[ 14_400, „Abu Dhabi“, „Muscat“, „Baku“, „Tbilisi“, „Yerevan“ ],
[ 16_200, “Kabul” ],
[ 18_000, “Ekaterinburg”, “Islamabad”, “Karachi”, “Tashkent” ],
[ 19_800, „Chennai“, „Kolkata“, „Mumbai“, „New Delhi“ ],
[ 20_700, „Kathmandu“ ],
[ 21_600, „Astana“, „Dhaka“, „Sri Jayawardenepura“, „Almaty“,
          „Novosibirsk“ ],
[ 23_400, „Rangoon“ ],
[ 25_200, „Bangkok“, „Hanoi“, „Jakarta“, „Krasnoyarsk“ ],
[ 28_800, „Beijing“, „Chongqing“, „Hong Kong“, „Urumqi“,
          “Kuala Lumpur”, “Singapore”, “Taipei”, “Perth”, “Irkutsk”,
          “Ulaan Bataar” ],
[ 32_400, “Seoul”, “Osaka”, “Sapporo”, “Tokyo”, “Yakutsk” ],
[ 34_200, “Darwin”, “Adelaide” ],
[ 36_000, “Canberra”, “Melbourne”, “Sydney”, “Brisbane”, “Hobart”,
          “Vladivostok”, “Guam”, “Port Moresby” ],
[ 39_600, “Magadan”, “Solomon Is.”, “New Caledonia” ],
[ 43_200, “Fiji”, “Kamchatka”, “Marshall Is.”, “Auckland”,
          „Wellington“ ],
[ 46_800, „Nuku‘alofa“ ]]
```

create(name, offset)

Создает новый экземпляр `TimeZone` с заданным названием и смещением:

```
>> TimeZone.create(“Atlanta”, -5.hours)
=> #<TimeZone:0x31e6d44 @name=”Atlanta”, @utc_offset=-18000 seconds>
```

new(name)

Возвращает экземпляр `TimeZone` с заданным названием или `nil`, если такого часового пояса не существует. Этот метод служит для того, чтобы данный класс можно было использовать с методом-макросом `composed_of` в моделях `ActiveRecord` следующим образом:

```
class Person < ActiveRecord::Base
  composed_of :tz, :class_name => ‘TimeZone’,
              :mapping => %w(time_zone name)
end
```

us_zones

Вспомогательный метод, возвращающий набор объектов `TimeZone`, которые соответствуют часовым поясам в США:

```
>> TimeZone.us_zones.map(&:name)
=> [“Hawaii”, “Alaska”, “Pacific Time (US & Canada)”, “Arizona”,
“Mountain Time (US & Canada)”, “Central Time (US & Canada)”, “Eastern
Time (US & Canada)”, “Indiana (East)”]
```

Открытые методы экземпляра

<=> (other)

Сравнивает данный часовой пояс с параметром `other`. Сначала сравниваются смещения, потом – названия.

adjust(time)

Приводит заданное время к данному часовому поясу.

```
>> TimeZone['Fiji'].adjust(Time.now)
=> Sat Sep 01 10:42:42 UTC 2007
```

formatted_offset(colon = true)

Возвращает смещение данного часового пояса в виде строки в формате `HH:MM`. Если смещение равно нулю, возвращает пустую строку. Если `colon` равно `false`, в строку не включается двоеточие.

initialize(name, utc_offset)

Этот конструктор вызывается из метода `TimeZone.create`. Создает экземпляр `TimeZone` с заданными названием и смещением. Смещение представляет собой количество секунд между данным часовым поясом и UTC (GMT). В качестве единицы измерения выбраны секунды, поскольку именно в них в Ruby представляются смещения часовых поясов (см. метод `utc_offset` из класса `Time`).

now

Возвращает объект `Time.now`, приведенный к данному часовому поясу:

```
>> Time.now
=> Fri Aug 31 22:39:58 -0400 2007
>> TimeZone['Fiji'].now
=> Sat Sep 01 14:40:00 UTC 2007
```

to_s

Возвращает текстовое представление данного часового пояса:

```
TimeZone['Dublin'].to_s #=> "(GMT) Dublin"
```

today

Возвращает текущую дату, приведенную к данному часовому поясу:

```
>> Date.today.to_s
=> "2007-08-31"
```

```
>> TimeZone['Fiji'].today.to_s
=> "2007-09-01"
```

TrueClass

Напомним, что в Ruby все является объектами, даже литерал `true`, представляющий собой специальную ссылку на единственный экземпляр класса `TrueClass`.

Открытые методы экземпляра

blank?

Всегда возвращает `false`.

B

Предметы первой необходимости для Rails

О Rails вы, наверное, узнали, увидев какую-то из презентаций Дэвида или натолкнувшись в Сети на одно из многочисленных руководств. Возможно, вы даже прочитали книгу *Agile Web Development with Rails*[1] – она разошлась в таком количестве экземпляров, что стремительно приближается к первой позиции в списке самых успешных книг по программированию всех времен.

Это приложение представляет собой попурри из различных знаний и умений, необходимых, чтобы стать эффективным профессиональным разработчиком на платформе Rails. Такую информацию во всяческих введениях и учебных руководствах вы не найдете.

«Острие Rails»

Термином *острие* (edge) в сообществе принято называть самую последнюю ревизию Rails, которая хранится в репозитории и сопровождается DHH и командой разработчиков ядра. Большинство профессиональных разработчиков для Rails «сидят на острие», чтобы не пропустить важных усовершенствований и исправлений ошибок. Поэтому многие из самых полезных подключаемых к Rails модулей требуют, чтобы вы «сидели на острие», иначе отказываются работать.

---

[1] Д. Томас, Д. Х. Хэнссон «Гибкая разработка веб-приложений в среде Rails». – Пер. с англ. – СПб.: Питер, 2008.

Чтобы воспользоваться «острием Rails» (EdgeRails), вы должны включить копию Rails непосредственно в свое приложение, а не пользоваться версией, установленной на машине в виде библиотеки. Сделать это несложно, достаточно ввести команду `rake rails:freeze:edge`, находясь в каталоге своего Rails-проекта. Это задание Rake обращается к системе Subversion и экспортирует Rails из репозитория в каталог `vendor/rails` проекта. Начиная с этого момента, все компоненты Rails (консоль, сервер и т. д.) в проекте будут пользоваться острием Rails, а не версией, установленной из gem-пакета.

Если вы опытный разработчик, то, возможно, задаетесь вопросом, зачем нужно работать не с выпущенной *официально* версией Rails, которая существует только в виде головной ветви в репозитории. Не опасно ли это? Что если изменения в репозитории нарушат работу приложения? Иногда такое случается, поэтому не стоит тащить самую последнюю версию Rails при каждом изменении кода. Рекомендуется выбрать некую конкретную ревизию острия, которая считается относительно стабильной. Это можно сделать, указав номер ревизии в переменной, передаваемой Rake-заданию `freeze edge`:

```
rake rails:freeze:edge REVISION=1234
```

Минутку, а как узнать номер последней стабильной ревизии? И, если какая-то ревизия стабильна, почему она не выпущена в виде официальной версии? Признаюсь, убедительных ответов на эти вопросы у меня нет. Решение о том, какую ревизию острия использовать, – скорее искусство, чем наука, и для каждого проекта его приходится принимать заново в зависимости от возникающих потребностей. Обычно я начинаю с последней ревизии, доступной на момент создания проекта, и работаю с ней, переходя на более свежую, только когда возникает необходимость.

Команда разработчиков ядра Rails придерживается очень строгой политики в части тестового покрытия, поэтому очень редко изменения в репозитории «ломают систему». Но, хотя острие, как правило довольно стабильно, официальные версии обычно отстают от него на несколько месяцев. Организовано несколько серверов непрерывной интеграции, которые автоматически тестируют все версии на основном стволе Rails с различными адаптерами СУБД. Извещения об отказах направляются в список рассылки rails-core, на который можно подписаться на странице *http://lists.rubyonrails.org/mailman/listinfo/rails-core*.

Замечания об окружающей среде

Нет, я не собираюсь говорить о квотах на выброс углерода или о глобальном потеплении. Любой профессиональный разработчик Rails часто пользуется командной строкой, поэтому хотелось бы как можно реже попадать в этой ситуации впросак и избавить себя от лишних на-

жатий клавиш. Приводимые ниже советы и хитрости помогут вам сделать свою жизнь легче и радостнее.

Синонимы

Как минимум следует завести в конфигурационном файле оболочки синонимы для запуска сервера и консоли Rails. Джеффри Грозенбах предлагает такие варианты[1]: `alias ss './script/server'` и `alias sc './script/console'`.

Библиотека color

PJ Hyett и Крис Уонстрэт (Chris Wanstrath), авторы блога blog Err[2], подарили сообществу ценные инструменты для раскрашивания. Для начала установите gem-пакет `color`:

```
sudo gem install color -source require.errtheblog.com
```

Теперь ничего не стоит раскрасить строки в окне терминала, поскольку в класс `String` добавлены методы для ANSI-цветов.

Библиотека Redgreen

Раз уж зашла об этом речь, раскрасим заодно результаты прогона тестов в красный и зеленый цвета. Сначала установите gem-пакет `redgreen`:

```
sudo gem install redgreen -source require.errtheblog.com
```

Затем откройте в каком-нибудь проекте Rails файл `test_helper.rb` и добавьте в него предложение `require 'redgreen'`. Теперь при прогоне комплекта тестов успешно завершившиеся тесты будут печататься зеленым цветом, а завершившиеся с отказом или ошибкой – красным.

Необходимые подключаемые модули

Некоторые подключаемые модули настолько ценны, что (на мой взгляд) должны были бы войти в ядро Rails. Но, поскольку разработчики ядра придерживаются философии «меньше – значит больше», то дистрибутив Rails склонен сокращаться, а не разрастаться.

Ниже перечислены подключаемые модули, которые я (и читатели моего блога) считаю предметами первой необходимости, каковыми любой профессионал в области Rails должен пользоваться регулярно.

Если хотите познакомиться с полным перечнем предложений для этого раздела, читайте комментарии в моем блоге по адресу *http://www.jroller.com/obie/entry/rails_plugins_worth_their_something*.

---

[1] *http://nubyonrails.com/articles/2006/01/19/sxsw-aliases.*

[2] *http://errtheblog.com/.*

Умолчания для ActiveRecord

`http://svn.viney.net.nz/things/rails/plugins/active_record_defaults`.

Этот модуль позволяет легко задавать умалчиваемые значения атрибутов для новых объектов модели.

Помощники для отладки представлений

`script/plugin install debug_view_helper`.

Этот модуль позволяет добавить кнопку, при щелчке по которой появляется всплывающее окно, содержащее следующую отладочную информацию, которую мы привыкли видеть в окне ошибок в режиме разработки:

- параметры запроса
- переменные сеанса
- флэш-переменные
- значения, присвоенные переменным шаблона

Уведомление об исключении

`http://svn.rubyonrails.org/rails/plugins/exception_notification`.

Этот модуль написан одним из разработчиков ядра, Джеймисом Баком. Он автоматически отправляет вам сообщение по электронной почте, когда на вашем сайте возникает исключение. Отличная замена отделу контроля качества для низкобюджетных проектов, которые *могут* позволить себе работать нестабильно. Модуль подстрахует вас, если проект *не может* себе такого позволить.

Протоколирование исключений

`http://svn.techno-weenie.net/projects/plugins/exception_logger`.

Брайан Хелмкамп (Bryan Helmkamp) сказал: «Я рекомендую пользоваться `exception_logger`, а не `exception_notification`. RSS-каналы и пользовательский веб-интерфейс – неотразимые функции».

Has Finder

`gem install has_finder`.

Это расширение `ActiveRecord`, которое упрощает создание специализированных методов поиска и подсчета в моделях. Подробное описание см. на странице *http://www.pivotalblabs.com/articles/2007/09/02/hasfinder-its-now-easier-thanever-to-create-complex-re-usable-sql-queries*.

Has Many Polymorphs

`http://blog.evanweaver.com/files/doc/fauna/has_many_polymorphs`.

Подключаемый к `ActiveRecord` модуль для создания самоссылающихся и двусторонних полиморфных ассоциаций. Проще всего описать его так: «`has_many_polymorphs` похож на `has_many :through`, где цель `belongs_to` – полиморфная ассоциация». Хотите узнать больше? Читайте руководство по адресу *http://m.onkey.org/2007/8/14/excuse-me-wtf-is-polymorphs*.

Трассировка запросов

`https://terralien.devguard.com/svn/projects/plugins/query_trace`.

В протокол режима разработки в Rails заносятся все SQL-предложения, генерируемые `ActiveRecord`. Этот небольшой подключаемый модуль, написанный широко известным Ruby-стом Натаниэлем Тэлботтом (Nathaniel Talbott), добавляет к каждому запротоколированному предложению короткую трассировку вызовов. Его полезность вы в полной мере осознаете, когда будете искать причину проблемы «N+1 select» или попытаетесь отлаживать ошибки, связанные с кэшированием.

Spider Tester

`http://sample.caboo.se/plugins/court3nay/spider_test`.

SpiderTester (автор – Кортенэ) – сценарий для автоматизированного тестирования сопряжений, который посещает все страницы вашего приложения.

Он выполняет несколько полезных функций:

- разбирает HTML-код каждой страницы и предупреждает обо всех случаях некорректной разметки;
- находит все внутренние ссылки на ваш сайт (как статические, так и динамические) и следует по ним;
- находит все ссылки, сформированные `Ajax.Updater`, и следует по ним;
- находит все формы и пытается отправить их, заполняя поля там, где это возможно.

Данный модуль удобен для решения следующих задач:

- обнаружения отсутствующих статических страниц (с расширением .html);
- выявления недостаточности тестового покрытия. Забыли протестировать файл? Не ждите, пока на ошибку наткнется пользователь;
- простого тестирования значений полей в формах методом случайных проб (fuzzing);

- автоматизированного тестирования путей в формах. Часто встречаются формы, указывающие на некорректный адрес, и до сих пор было невозможно автоматизировать тестирование этой ситуации, не встраивая дополнительные проверки непосредственно в код приложения.

Другие подключаемые модули

Описания других полезных подключаемых модулей приведены в различных главах книги.

Демо-ролики

Демо-ролики (screencast) – это распространяемые в Сети видеоматериалы для обучения по какой-то узкой теме. Демонстрируя запись изображения с экрана монитора (отсюда и название), автор разъясняет те или иные концепции и пишет код.

Сайт PeepCode

`http://peepcode.com/`

Продюсер Ruby on Rails Podcast Джеффри Грозенбах характеризует свои демо-ролики на сайте PeepCode как «интенсивный способ изучения разработки веб-сайтов на платформе Ruby on Rails».

В реальности он пользуется своими обширными познаниями и успокаивающим голосом, чтобы *мягко* познакомить зрителя с некоторыми сложными сторонами мудрости Rails. Часовые ролики выходят ежемесячно. Каждый стоит 9 долларов США, но, на мой взгляд, это себя окупает.

Сайт Railcasts

`http://railscasts.com/`

Жалко денег? Райан Бейтс (Ryan Bates) почти каждую неделю выкладывает новые ролики бесплатно. Эпизоды короче и посвящены более узким темам, чем на сайте PeepCode. Рассчитаны они на разработчиков среднего уровня.

Система Subversion

В последние годы Subversion (SVN) заняла доминирующую позицию в нише систем управления исходными текстами (SCM) и небезосновательно. Она работает быстро, стабильно и во многих отношениях усовершенствовала свою предшественницу – систему CVS.

Мир Rails связан с Subversion многими нитями. В частности, вся система подключаемых модулей зависит от репозиториев Subversion, откуда эти модули извлекаются.

Что делать, если вы вынуждены пользоваться другой SCM, например ClearCase или StarTeam (и не можете сменить работодателя более вменяемым)? Иногда помогает такая стратегия: организуйте локальный SVN-репозиторий для повседневной работы, а в основную систему управления версиями складывайте только готовые версии.

Rake-задания для Subversion

Многие блогеры предлагают различные Rake-задания, упрощающие работу с системой Subversion. Создайте в своем проекте Rails файл `lib/tasks/svn.rake` и поместите в него определения заданий, приведенных ниже.

В листинге B.1 представлено задание, которое автоматически подготавливает новый проект Rails, настраивая различные параметры Subversion, в том числе игнорируемые расширения.

Листинг B.1. Rake-задание, конфигурирующее Subversion для Rails

```
namespace :rails do
  desc "Конфигурирование Subversion для Rails"
  task :configure_for_svn do
    system "svn propset svn:ignore -R '.DS_Store' . --force"
    system "svn update"
    system "svn commit -m 'ignore all .DS_Store files'"
    system "svn remove log/* --force"
    system "svn commit -m 'removing all log files from subversion'"
    system "svn propset svn:ignore '*.log' log/ --force"
    system "svn update log/"
    system "svn commit -m 'Ignoring all files in /log/ ending in .log'"
    system "svn propset svn:ignore '*' tmp/sessions tmp/cache
tmp/sockets"
    system "svn commit -m 'Ignoring all files in /tmp/'"
    system "svn propset svn:ignore '*.db' db/ --force"
    system "svn update db/"
    system "svn commit -m 'Ignoring all files in /db/ ending in .db'"
    system "svn move config/database.yml config/database.example --
force"
    system "svn commit -m 'Moving database.yml to database.example to
provide a template for anyone who checks out the code'"
    system 'svn propset svn:ignore "locomotive.yml\ndatabase.yml"
config/ -force'
    system "svn update config/"
    system "svn commit -m 'Ignoring locomotive.yml and database.yml'"
    system "script/plugin install -x
http://dev.rubyonrails.org/svn/rails/plugins/exception_notification/"
  end
end
```

В листинге B.2 объявлено пространство имен svn и пять заданий в нем для проверки состояния, добавления и удаления рабочих файлов и постановки кода на учет.

Листинг B.2. Полезные Rake-задания, относящиеся к Subversion

```
namespace :svn do
  task :st do
    puts %x[svn st]
  end

  task :up do
    puts %x[svn up]
  end

  task :add do
    %x[svn st].split(/\n/).each do |line|
      trimmed_line = line.delete('?').lstrip
      if line[0,1] =~ /\?/
        %x[svn add #{trimmed_line}]
        puts %[added #{trimmed_line}]
      end
    end
  end

  task :delete do
    %x[svn st].split(/\n/).each do |line|
      trimmed_line = line.delete('!').lstrip
      if line[0,1] =~ /\!/
        %x[svn rm #{trimmed_line}]
        puts %[removed #{trimmed_line}]
      end
    end
  end
end

desc "Прогон перед постановкой на учет"
  task :pc => ['svn:add', 'svn:delete', 'svn:up', :default, 'svn:st']
```

Сайт WorkingWithRails.com

На составление полного обзора и описания онлайнового сообщества пользователей Rails ушло бы слишком много времени, а результат быстро устарел бы. Однако я хочу привлечь ваше внимание к сайту *workingwithrails.com* (сокращенно – WWR), на котором находится открытая база данных обо всем, что касается индивидуумов и групп, занимающихся разработкой на платформе Rails. Он любовно поддерживается (с помощью Rails) Мартином Сэдлером (Martin Sadler) и его компанией DSC, базирующейся в Великобритании.

Со временем я понял, что WWR – один из самых эффективных способов дать знать о себе как о члене профессионального сообщества Rails, особенно если вы ищете работу в качестве независимого подрядчика. Если же вы ищете специалистов или просто хороших людей для совместной работы, WWR – великолепный ресурс для отыскания популярных участников сообщества и талантливых разработчиков в непосредственной близости от вас.

Если эта книга вам понравилась и оказалась полезной, пожалуйста, дайте рекомендации ее автору, его помощникам и основным рецензентам:

http://www.workingwithrails.com/person/5391-obie-fernandez

http://www.workingwithrails.com/person/8048-matt-bauer

http://www.workingwithrails.com/person/5747-david-a-black

http://www.workingwithrails.com/person/5541-trotter-cashion

http://www.workingwithrails.com/person/1363-matt-pelletier

http://www.workingwithrails.com/person/4746-jodi-showers

http://www.workingwithrails.com/person/5137-james-adam

http://www.workingwithrails.com/person/848-pat-maddox

http://www.workingwithrails.com/person/582-sebastian-delmont

http://www.workingwithrails.com/person/5167-sam-aaron

Использование точек расширения Subversion

Если вы хотите поднять применение Subversion в своих проектах Rails на новый уровень, познакомьтесь с отличной статьей по адресу *http://railspikes.com/2007/8/20/subversion-hooks-in-ruby*.

Послесловие

Что означает Путь Rails для вас

Я люблю сообщество Rails. Оно полно жизни, остроумия и интеллекта. Честно говоря, наше сообщество – одна из самых лучших сторон работы с Rails и в немалой степени – причина его успеха. Приближаясь к завершению работы над книгой, я ощутил потребность включить хоть какую-то зарисовку о сообществе, чтобы читатель почувствовал, на что оно похоже. Эта идея не давала мне покоя. Все мы знаем, что Путь Rails *существует*, но каждый интерпретирует его по-своему, в зависимости от личного опыта и способности к радостному мироощущению. Можно ли придумать лучший способ закончить книгу, чем привести подборку *ваших мыслей*!

И вот я попросил, а вы – все вы – откликнулись. Надеюсь, вы получите такое же наслаждение от чтения следующих цитат, остроумных замечаний и эссе, какое они доставили мне.

Оби Фернандес, Джексонвилль Бич (12 сентября 2007)

Моей первой мыслью была: «Путь Rails – это дорога к счастью». Но потом я вспомнила поэму «Элеонора» Эдгара По, которая мне очень нравится. В начале герой говорит о безумии и высшем разуме и задается вопросом, а не одно ли это и то же. Потом он жалеет тех, кто грезит лишь ночью во сне. Я рада, что принадлежу к тем, кто умеет видеть сны наяву, а не только в ночи. Я знаю тайну!

> *Меня называли безумным, но вопрос еще далеко не решен, не есть ли безумие высший разум и не проистекает ли многое из того, что славно, и все, что глубоко, из болезненного состояния мысли, из особых настроений ума, вознесшегося ценой утраты разумности. Тем, кто видят сны наяву, открыто многое, что ускользает от тех, кто грезит лишь ночью во сне. В туманных видениях мелькают им проблески вечности, и, пробудясь, они трепещут, помня, что были на грани великой тайны.*[1]
>
> Эдгар Алан По

Дейзи Мак-Адам, мой любимый разработчик для Rails

[1] Эдгар По «Элеонора», перевод Н. Демуровой.

Создателям Rails было так неприятно работать со всеми существующими инструментами разработки веб-приложений, что они решили начать с нуля. Сам объем этой книги говорит о том, что это оказалось серьезным предприятием. Насколько лучше стал бы PHP, если бы авторы продолжали модернизировать его вместо написания Rails?

Быть может, и стал бы немного лучше, но сторонний наблюдатель это вряд ли заметил бы. Перефразируя владельца магазинчика из мультфильма Whisper Of the Heart («Шепот сердца»), скажу: «Можно отполировать грубую руду, но кому это нужно? Маленький самоцвет внутри чище. Но, чтобы найти его, нужно потратить время и силы». Rails учит нас, что временами нужно отбросить то, что привык видеть, и только тогда найдешь то, что искал.

Путь Ruby состоит в том, чтобы полировать камень, пока не найдешь в нем драгоценность. Ни Ruby, ни Rails, при всей их мощи и элегантности, не могут помешать вам писать громоздкий нечитаемый код. И лишь вы сами решаете, когда написанный код уже не поддается дальнейшей полировке. Ruby предлагает бросить вызов самому себе и создавать такие системы, которыми было бы приятно пользоваться и любоваться. Rails понуждает нас постоянно проверять свои предположения и не бояться выбрасывать большие куски кода, если мы ими недовольны.

Две движущие силы – неудовлетворенность текущим состоянием дел и постоянная самокритика – превратили Ruby и Rails в столь популярные и могучие инструменты. Если вы ощущаете эти силы в себе, то сможете писать более качественный код. Если вы сумеете направить их в нужное русло, то отыщете Путь Rails, пусть даже он приведет вас к написанию чего-то, что сможет заменить сам Rails.

Уилсон Билкович,
верный почитатель Rails
и разработчик ядра Rubinius

Работая в Ruby on Rails, я чувствую себя так, будто рисую остро отточенными карандашами на хорошей бумаге, имея под рукой отличный ластик. Я могу очень быстро набросать эскиз идеи и посмотреть, заслуживает ли она добрых слов. Если нет, приложение можно выбросить в корзину без особых сожалений, потому что на него был затрачен минимум усилий. Если же все получается, я могу улучшить то, что мне нравится, убрать то, что не нравится, и постепенно переходить от прототипа к продукту. А спираль проектирования оказывается туже, чем мне доводилось видеть раньше в ходе разработки программного обеспечения.

Дэн Гебхардт

Мудрый ученик слышит о пути Rails и избирает его.
Средний ученик слышит о пути Rails и забывает его.
Глупый ученик слышит о пути Rails и громко смеется.
Но если бы не было смеха, то не было бы и пути Rails.

Йон Ларковски, принося извинения Лао Цзы

Несколько последних лет я работал на платформе .NET. Я тратил все свое время на .NET в той или иной форме, а когда спрашивал себя, зачем это делаю, то неизменно отвечал: «Надо же зарабатывать на жизнь». Я не говорю, что люблю эту технологию, я утверждаю, что это путь в никуда.

Думаю, что увидел свет в конце длинного туннеля своей карьеры – Ruby и Ruby on Rails. Работа с ними приносит огромное удовлетворение, даже просветление, свойственное в своем высшем проявлении философам. Когда коллеги спрашивают меня, что дает мне Rails, я часто затрудняюсь объяснить это понятными им словами. Это как обретение религии – не той, что писана на скрижалях или отправляется в храмах, а той, что объединяет людей общим опытом, делая жизнь такой полной.

Путь или практика Rails – это обретение чувства, что поступаешь правильно. Технология – это хорошо, но истинная радость – в сообществе.

Я работаю с Rails всего год с небольшим, но, имея за плечами 20 с лишним лет опыта разработки программного обеспечения, нутром чую, что нашел нечто стоящее.

Роб Базинет, разработчик программного обеспечения, который сейчас с удовольствием изучает Ruby и Rails

Путь Rails – это M: Magical V: Velocity Focused C: Community Driven[1].

Мэтт Марголис

Освободи себя от бремени принятия решения – вот Путь Rails. Все важные решения уже приняты за вас, и вместо того чтобы думать, какую выбрать технологию и как структурировать приложение, вы просто сидите и программируете. А, освоив процесс разработки, начинаете расширять Rails, создавая необходимые подключаемые модули. Это и есть дзен разработки.

Бен Штиглиц

Я сравнительно недавно занялся веб-разработкой. Я не писал CGI-сценарии на Perl, не знаю PHP и уж тем более J2EE. Rails стал моим введением в серьезное веб-программирование. Для меня Rails превратил разработку для Сети в нечто, чего не стоит всеми силами сторониться.

Не могу удержаться от сравнения Rails с C. Я изучил C на ранних этапах своего романа с программированием. Голая мощь, выразительность (до того я писал в основном на ассемблере), способность очень быстро создавать «крутые» вещи. Smalltalk заставил меня пережить те же чувства еще раз. И вот теперь история повторяется с Rails... Конечно, тут велика заслуга Ruby... способность достичь многого за очень короткое время и с очень небольшим объемом кода.

---

[1] Непереводимая игра слов. Речь идет об аббревиатуре MVC (модель-вид-контроллер), которую автор расшифровывает как «магия, приверженность быстроте и развитие всем сообществом». – *Прим. перев.*

Для меня Путь Rails дан в ощущениях. Необузданная продуктивность, текучесть, выразительность, пластичность, почти полное отсутствие преград и препятствий...

Путь Rails – это стремительная езда по прибрежному хайвею в Калифорнии ярким солнечным днем в кабриолете Corvette с откинутым верхом.

Дэйв Эстелс, автор TDD,
наставник и менеджер по тестированию
в компании Google, Inc.

Путь Rails – это...

Сойти с пути, чтобы помочь, не ожидая награды.

Воспринимать идеи – новые и старые.

Осознать, что у всех нас есть более полезные занятия.

Джим Ремсик, разработчик на платформе .NET,
которого в ближайшем будущем ожидает смена работы

Итак, что же такое Путь Rails, о котором там много говорят? Есть ли он вообще и, если есть, то поддается ли определению? Я полагаю, что Путь Rails – субъективное понятие, а раз так, объясню, что оно означает для меня. Если мы когда-нибудь встретимся, будет интересно узнать, что такое Путь Rails для вас.

Начнем с простой, пожалуй, даже неловкой аналогии. Поговорим о паровозах, вагонах, поездах и железных дорогах. Нет, не о тех моделях, которые ваш папа старательно собирает и раскрашивает в гараже. О настоящих. О больших мощных машинах. О путях, тянущихся на многие километры. О грузах. О пассажирах. О тяжелом, наполненном гарью запахе дыма, пара и угля. О технике, которая привела к промышленной революции. О технике, которую люди постоянно совершенствовали и изобретали снова и снова. Так происходит и теперь. Например, если уж говорить о железнодорожной технике, то шанхайский экспресс на магнитной подушке ох как далеко ушел от паровозов, бегавших по английским железным дорогам в начале XIX века. О технологиях железнодорожного сообщения были написаны такие же книги, как эта, для людей, которые интересовались или создавали эти технологии. Но большинству людей сама железнодорожная техника неинтересна. Интересно то, что она дает: отдых на море, поездку к бабушке, доставку товаров на новые рынки, возможность жить далеко от места работы. Но лично мне в железных дорогах интересны внутренне присущие им ограничения; они не позволяют добраться куда угодно. Список пунктов назначения определен за вас. Уже приняты решения о том, куда прокладывать рельсы и где строить станции. Кто-то решил, что вам нужно ехать именно из Лондона в Ньюкасл. «Эй, погодите! – восклицаете вы. – Я сам хочу решать, куда мне ехать». Свобода – это возможность выбора, не так ли?

У Ruby on Rails тоже есть ограничения. Он прокладывает для вас рельсы. Он говорит, куда идти. Конечно, вы не связаны жестко указанным направлением – можете идти, куда заблагорассудится. Но я уверен, что путешествие на поезде по ухабам и колдобинам будет не самым комфортабельным. И тут возникает важный вопрос: «Откуда я знаю, что Rails ведет меня в правильном направлении?» Понятно, что вопрос субъективный. Ответ зависит от того, что вы считаете «правильным». Однако для меня направление, предложенное Rails, почти всегда оказывалось правильным. Позвольте добавить немного конкретики.

Я был погружен в работу над кандидатской диссертацией. Я реализовал на Java прототип для оценки собственных идей. Проблема была только в том, что система, написанная на Java, была подобна постоянно рушащемуся карточному домику. Оглядываясь назад, я объясняю это тем, что мои идеи все время находились в движении, непрерывно изменялись. В конце концов, это же была исследовательская работа. К сожалению, код на Java в результате этих изменений стал слишком сложным и запутанным. В идеальном воображаемом мире я бы просто сформулировал новые идеи, которые волшебным образом реализовал бы для меня какой-нибудь искусственный интеллект. Увы, реальность далека от идеала, и код приходится писать самому. Однако случилось так, что реализация на Java стала управлять моими идеями, а не идеи – реализацией. Например, не раз я говорил себе: «Вот что хорошо бы уметь, но это или слишком сложно, или слишком долго программировать». И это было для меня источником разочарования, за которым следовала депрессия.

Но это все эмоции. Поговорите с кем-нибудь, кто писал кандидатскую, и вы поймете, что любой испытывал депрессию на каком-то этапе исследований. Надеюсь, вы простите меня за тавтологию, но сама природа написания кандидатской диссертации депрессивна. В моем случае депрессия возникала из-за неудовлетворенности инструментами и непонимания, за каким поворотом находится выход из тупика, в котором я оказался. Я был растерян и потерял ориентиры.

И тут я наткнулся на Rails. Не помню, что меня привлекло в нем. Быть может, это было связано с тем, что я уже некоторое время пользовался – и с удовольствием – простенькой вики-системой Instiki (одно из самых первых приложений, написанных с помощью Rails). А, может быть, это было отчетливое ощущение сообщества. Или умные, интересные и часто забавные заметки в блогах. Или то очевидное волнение и восторг, с которым люди отзывались о Rails. Ох, чуть было не забыл о раскрывающих глаза демо-роликах. Наверное, всего понемножку. Rails предложил мне то, в чем я нуждался: сообщество умных, охваченных энтузиазмом, интересных людей, работающих с платформой, которые позволяли создавать полезные вещи замечательно быстро и без больших усилий. Именно то, что мне было нужно!

Следующий год или два были потеряны для исследовательской работы – я усиленно учился. Однако я узнал о программировании куда больше, чем в любой другой сравнимый период времени. Я проглатывал статьи в блогах и книги, как будто изголодался по информации. Изучение Ruby раскрыло мне глаза на интересные грани языков программирования. Внезапно я увидел, что программирование не обязательно должно быть инженерной дисциплиной – у него есть и много других аспектов. Это и искусство, и ремесло, и даже исследование самого языка. Лежащие в основе Rails идейные принципы («примат соглашения над конфигурацией» и «не повторяйся») не просто имели смысл – их разумность не вызывала никаких сомнений. На меня нашло просветление. Все, что делали и о чем говорили участники сообщества Rails, казалось интересным. Сообщество работало, как чудесный фильтр, указующий, что читать и изучать, и – самое главное – я стал этому фильтру доверять.

Но вернемся к поездам и свободе – «железные дороги ограничивают выбор места назначения» и «решение о выборе конечных пунктов уже принято за вас». Мы говорили, что ответом на эти заявления может быть: «я хочу сам решать, куда мне поехать» и «свобода – это возможность выбора». А теперь взгляните на это с точки зрения моей ситуации. У меня была полная свобода выбора, куда двигаться в своих исследованиях. Тут как раз никакой проблемы не было. Проблема заключалась в том, что я не знал, куда ехать, поэтому не ехал никуда. Слишком большое количество неясных и неизвестных возможностей по существу стали препятствием на моем пути.

Но после передышки, отданной изучению Rails, я вернулся к своим исследованиям с новым пылом. Я решил выбросить свою реализацию и сделать все заново на Ruby и Rails. Это заняло малую толику времени, потраченного на реализацию оригинального решения, и объем кода тоже уменьшился пропорционально. Более того, новая реализация была полностью протестирована, не осталось никаких карточных домиков и, ко всему прочему, я получил истинное наслаждение, работая над ней. Я наслаждался работой над тем, что раньше приносило только разочарование и депрессию. Выяснилось, что использование Rails в качестве фильтра возможных вариантов реализации вовсе не кажется ограничением и тюрьмой; напротив, Rails проясняет и указывает направление. Я полагаю, что именно эту особенность и называют Путем Rails, который для меня стал началом увлекательного и многообещающего путешествия.

Сэм Аарон, который не дает кратких ответов ☺

Ruby on Rails доказывает, что быстро – необязательно небрежно, а лучшее – необязательно сложное. Это идеальное подтверждение принципа 80/20 в мире современных веб-приложений – Rails не может удовлетворить всех и каждого, но если он подходит, решение оказывается великолепным.

Габе да Сильвейра, сайт websaviour.com

Rails и Ruby, Ruby и Rails – что еще сказать?

Я занимаюсь объектно-ориентированным программированием вот уже 30 лет. Я был одним из счастливчиков, участвовавших в сообществе Smalltalk в пору его расцвета. Я неохотно начал работать с Java, когда в мире торжествовал лозунг «пишешь один раз, выполняешь всюду».

Потом я уволился и начал развлекаться с разными технологиями, удовлетворяя собственные капризы и берясь за случайную работу, чтобы заработать доллар-другой.

Я пробовал различные веб-приложения с открытыми исходными текстами, иногда хорошие, иногда не очень. Я поддерживал вики-систему, построенную с помощью mediawiki, и некоторое время активно экспериментировал с ее кодом. Она доказывает, что даже в PHP можно написать хороший код. С другой стороны, некоторые выполненные мной консалтинговые работы напомнили, сколь ужасны могут быть результаты программы, собранной на коленке из разнородных частей.

Как-то раз один приятель спросил, пробовал ли я Ruby, я решил познакомиться с этим языком. И внезапно почувствовал себя дома. Ruby позаимствовал многие объектно-ориентированные концепции из Smalltalk, добавил такие передовые вещи, как модели и синглетные методы, а также кое-какие идеи – в основном, здравые – из Perl. Я интересовался также Python'ом, но Ruby показался мне более полным и удобным языком.

Потом я взялся за одну работу на Rails – добавить некоторые новые функции в существующее приложение. По сравнению с аналогичной работой по расширению ответственного приложения на PHP это был рай. Мне не пришлось тратить все время, чтобы понять, что же должна была делать программа; структура кода для Rails была очевидной. Но самым замечательным оказались ТЕСТОВЫЕ СЦЕНАРИИ. Великое благо для человека, вошедшего в проект со стороны.

Разумеется, код был так хорош не просто потому, что был написан с помощью Rails. Но это сильно помогло. Еще один аспект, общий у Ruby и в Rails со Smalltalk, – наличие в литературе большого объема хорошего кода, который можно читать, разбирать и использовать в качестве образца.

Я не хочу сказать, что на Rails нельзя написать ужасный код, но для этого еще придется постараться! Если вы доверяете своему внутреннему голосу, работая в Rails, то быстро поймете, когда сделаете что-то не то, и попробуете поискать, как другие решали аналогичные задач. И нужные примеры найдутся без труда!

Рик де Натали, уволившийся гуру по языку Smalltalk в IBM,
глава DenHaven Consulting и член команды Terralien

Вдыхая в полночь ароматы,

Я признаюсь в любви к Rails.

И ночь становится прекрасной,

Звезды мерцают в глазах.

О, прекрасный мир Rails,

Я люблю тебя.

...

Хакем, студент университета из Китая

По моему опыту (сначала в бытность наемным работником, а теперь владельцем компании), Путь Rails можно выразить тремя С: *с*отрудничество (collaboration), *с*огласованность (consistency) и *с*оучастие (contribution).

Сотрудничество: следование Пути Rails означает не только сотрудничество между членами команды, но и более тесное сотрудничество с клиентами. У членов команды проявляется ощутимое волнение, когда они в полной мере задействуют потенциал Rails. Поэтому экспертная оценка программы (code review) происходит более динамично и дает более полезные результаты, а не заводит в тупик. С точки зрения взаимодействия с клиентом это означает более частый выпуск версий и получение откликов на ранних стадиях разработки. С Rails мы гораздо точнее попадаем в движущуюся мишень, чем с PHP.

Согласованность: имея опыт работы с различными языками и средами, не занимавшими отчетливую пристрастную позицию, я отлично знаю, как трудно бывает расшифровать стиль кодирования другого программиста. Приняв Путь Rails, мы сократили время, необходимое для объяснения другим того, что хотели сказать. Мы просто все оказались «на одной странице». Это бесценное преимущество.

Соучастие: следуя Пути Rails, нам гораздо проще делиться с сообществом. Я уже лет десять являюсь потребителем программ с открытыми исходными текстами, но Ruby on Rails – первый проект, в котором я поучаствовал своими исправлениями (и испытал радость, когда увидел, что они включены в ядро Rails). Соучастие возможно на любом уровне: подключаемые модули, gem-пакеты, документация, заметки в блогах о новых открытиях и даже выступления на местных мероприятиях, связанных с Rails. Дух кооперации просто разлит на всем пути Rails. Такого я не видел ни в одном другом программном проекте, и я счастлив быть его частью.

Джаред Хэворт, основатель Alloy Code,
компании по веб-разработкам в Рэйли, штат Северная Каролина

Для меня Путь Rails – это свобода. Свобода создавать решения, а не корм для компилятора. Свобода изучать один базовый язык, а не эзотерические форматы конфигурационных файлов. Свобода вылепить из этого языка, как из куска глины, элегантное программное решение, не вступая в споры с проектировщиками языка или API, уверенными, что все знают лучше. Свобода получать удовольствие и не сетовать на то, что на рождение красивого кода ушли долгие, трудные часы. Свобода вернуться в знакомый край динамических языков.

Мэл Риффл, бывший программист на Smalltalk

Путь Rails – это способ находить самые красивые решения любой задачи. Такие решения не найти с помощью одноразового хакерского трюка – они основаны на разумных соглашениях и принципах, разделяемых другими разработчиками. Долой мракобесов, добро пожаловать художникам. Да здравствует Путь Rails!

Хэмптон Кэтлин, самопровозглашенный Пророк Ruby

Rails привел меня в такие места, о существовании которых я уже забыл. Я помню то время, не так уж давно, когда сидел ночами, медленно вкушая радость от создания программы. Учеба в колледже была порой исследований, поиска пределов моего разума и способностей.

Позже Корпорация и Комфорт лишили меня Волнения и Развития. Учение ушло на задний план, и я превратился в вылепленный по стандартам автомат. Работа, достаточная, чтобы произвести впечатление, но без всяких восторгов. Хорошее считается великим, а великое – несбыточная мечта.

Осознание того, что я обменял свою страсть на стабильную зарплату, пришло подозрительно близко к тому моменту, когда я открыл для себя Rails. Rails увел меня из Корпорации и вернул к Осуществлению. Ruby дал мне пощечину и оставил смеющимся, несмотря на боль.

Барри Хесс, которого можно найти на сайте bjhess.com

У Rails есть свое пристрастное мнение о том, как надо конструировать приложения – от самых простых до весьма сложных, – с минимальными затратами усилий и обеспечивая тесную интеграцию. Каждый разработчик должен испытать Путь Rails, чтобы решить, подходит ли он ему, и понять, что из этого подхода можно привнести в собственную работу.

Джеффри Вайзман,
универсал в разработке ПО
и свободный писатель/редактор

Путь Rails состоит в том, чтобы стать частью сообщества людей, стремящихся помогать друг другу работать более здраво, создавать более качественные продукты и получать при этом больше удовольствия.

Это свобода не быть обязанным какой-то одной компании. Путь Rails протоптан теми, кто ценит красоту и мастерство в разработке программного обеспечения. Следуя по нему, вы сами оставляете пометки для тех, кто пойдет за вами.

Люк Мелиа, идущий по пути Rails в Нью-Йорк Сити

Rails – красота, воплощенная в коде.

Творческое, радостное, совершенное

Будущее, свободное от нечистоты.

Дэвид Паркер, все, что нужно знать,
на сайте davidwparker.com

Я начала программировать, когда мне было 13 лет. Хорошо помню то лето; я нагулялась и пришла с ободранными коленками. Мне на глаза попалась книга о программировании на BASIC для нашего домашнего компьютера TRS-80. Моим монитором был телевизор, файлы хранились на магнитофонной кассете, а печатала я на пятидюймовой термобумаге. Я подсела на программирование. Со временем мои компьютеры совершенствовались, как и умение программировать. Я освоила другие версии Basic и C/C++. Потратив четыре года на изучение Pascal и один семестр в колледже на Java, я изучила по книжке PHP и устроилась на свою первую работу. Я развивалась, приобретала новые знания о веб-разработке, осваивала правильные принципы проектирования: паттерн MVC, представление строк таблицы в виде объектов и шаблоны. Я пыталась не отставать от новых платформ для создания веб-приложений и экспериментировала с несколькими из них. Но ни один не показался мне достаточно простым или хотя бы достаточно логичным. Типичная часть любой разработки – CRUD (создание, чтение, обновление, удаление) – казалась мне скучной, поскольку повторялась из раза в раз. Веб-разработка мне надоела и стала раздражать! Примерно два года я находилась в состоянии «программистской депрессии», когда ничто, относящееся к кодированию, не радует. Приходя домой с работы, я даже не включала компьютер.

Сообщества программистов вселяют новые силы. В 2006 году я возобновила знакомство с разработчиком и давним приятелем Китом Кэйси, с которым мы несколько лет не общались. Он участвовал в нескольких проектах с открытыми исходными текстами, в том числе DotProject – системе управления проектами на платформе LAMP (Linux-Apache-MySQL-PHP). Я заключила с ним контракт, влилась в сообщество, стала участвовать в форумах, посвященных проектам, отвечать на вопросы и обсуждать дизайн. Он советовал мне ходить на мероприятия местных групп пользователей в Чикаго. Я нашла группу пользователей PHP и встретилась с несколькими программистами на Perl (только тихо, никому не говорите, что на собрании, посвященном PHP,

были «Perlовики»!), начала изучать Perl, а позже наткнулась на группу пользователей Ruby. Вскоре я стала ежемесячно посещать встречи всех трех групп – Ruby, Perl и PHP. Изредка общалась и с группой пользователей Python. Я узнавала много нового и постоянно контактировала с программистами как лично, так и в ходе работы над открытыми проектами. В то время я была единственным программистом в компании и истосковалась по разговорам с такими же чокнутыми. PHP – хороший инструмент для некоторых приложений, потому что поддерживается на большинстве веб-серверов. Perl я полюбила за его библиотеки для тестирования и гибкость; к тому же узнала много интересного о проектировании. Rails я люблю, потому что паттерн MVC кажется мне правильным, а `ActiveRecord` позволяет легко отображать строки базы данных на объекты. Обстраивание дает возможность быстро реализовать базовый код CRUD, оставляя мне время для работы над бизнес-логикой приложения. Программирование снова стало доставлять радость!

Rails – гибкая среда. Продолжая развиваться, я изучила процесс гибкой разработки, и Rails оказался самым необходимым инструментом. Обстраивание помогает мне быстро создать прототип и обсудить его с заказчиком на ранней стадии проекта. Как-то раз к разработчикам пришел бизнес-аналитик и пытался объяснить, чего он хочет. У нас уже было похожее готовое приложение, но ему надо было кое-что изменить, а для этого, к сожалению, требовалось значительное перепроектирование. Послушав его в течение 15 минут, я подошла к своему столу и с помощью программы Streamlined написала остов Rails-приложения с несколькими контроллерами и моделями. Почти полный сайт я сделала меньше чем за час. Я показала ему, что получилось, и после нескольких изменений в формах и добавления некоторых функций у нас было развернутое промышленное приложение. На все ушло примерно 18 часов.

Rails способствует быстрой разработке. Кстати, я поняла, что менеджерам, привыкшим к другим языкам, не так-то просто освоиться со скоростью разработки в Rails. Как-то мне поручили сделать несколько задач для одного проекта, и прочитали лекцию на тему о том, что в указанные мной сроки уложиться никак невозможно. И, конечно, мне ни за что не написать качественный код за такое время. Но я-то знала, что могу это сделать и при том качественно! С той встречи я ушла, исполненная решимости, и даже нашла время объяснять новичку в Rails весь процесс создания REST-совместимого сайта, обстраивания и подгонки отдельных частей под требования заказчика. Я прогнала тесты (46 штук!), и на этом разработка закончилась. На случай, если кто-то захочет усомниться в качестве кода, я натравила на свои тесты `rcov` (инструмент для определения тестового покрытия) и получила 100%-ное покрытие. Ну не круто ли?

Rails придал мне уверенность и смелость. Вообще-то я очень застенчивый человек и не часто разговариваю с незнакомцами. Но как-то раз я ехала на электричке в Чикаго, и рядом со мной уселся человек с Play

Station Portable (PSP) в удивительной сумке. Я спросила его про сумку и сообщила, что собираюсь стащить у своего мужа PSP и поставить на нее Linux. Он сказал, что работает программистом на мейнфрейме, и работа ему не нравится. Он хотел заниматься веб-разработками. Я тут же начала рассказывать о Rails, о том, какая это замечательная среда и в каком я от него восторге. Он стал делать заметки. Я рассказала, на какие сайты заглянуть, какие книги прочитать и даже показала ему классическое видео «Блог на Rails за 15 минут» на своем ноутбуке. Теперь вот мечтаю снова повстречать его в электричке и спросить, попробовал ли он!

Превыше всего я ценю подходящие для работы инструменты. Мне нравятся разные языки, но для работы над веб-приложениями я чаще всего выбираю Ruby и Rails. Если сайт содержит всего одну динамическую страницу или, скажем, форму для отправки почты, могу написать его на PHP. Perl идеально подходит для задач системного плана и обработки данных. У каждого языка есть свои сильные стороны, и я не стару спорить с теми, кто думает иначе. Я считаю, что самая большая проблема – отсутствие межязыкового сообщества разработчиков. Обычно кто-то просто пытается доказать, что язык X плохой. Ребята, успокойтесь... так ведь никому не станет лучше. Можно почерпнуть пользу из другого языка, даже если ты на нем не пишешь. Иногда, программируя на одном языке, я думаю, как бы сделала это на другом. И часто такое «вылезание из своего домика» помогает мне найти правильное решение.

Rails безусловно вдохновил меня и изменил мои представления о веб-разработке. Сообщество, сформировавшееся вокруг Ruby и Rails, обладает выдающимися достоинствами, и из-за этого я как программист становлюсь лучше.

Нола Стоуи, любительница языков

Путь Rails прагматичен до самых корней, его двигали две силы: большой опыт и необходимость решать реальные программистские задачи. Он отбрасывает традиционные представления о том, как надо работать, и предлагает свежий взгляд на устранение препятствий, снижающих продуктивность разработчика.

Курт Гиббс, автор самых успешных онлайновых учебных руководств по Rails

Иногда мне кажется, что Путь Rails – это выразительный, итеративный, простой, элегантный способ разработки веб-приложений.

Иногда мне кажется, что это игнорирование десятилетий коллективного опыта и изобретение колеса из кукурузного крахмала. Оно легче, а ехать нам недалеко, так зачем лишние сложности?

Иногда мне кажется, что Путь Rails означает «черт знает что».

Все зависит от дня.

Джей Левитт, бывший главный архитектор почтовых систем в America Online, проживает в Бостоне, штат Массачусетс

Путь Rails – это способ осчастливить программиста. Это автоматизация, абстрагирование и рефакторинг до тех пор, пока в коде веб-приложения не останется минимально необходимое количество строчек. Это когда хороший код писать проще, чем плохой.

А если программист доволен, то он может сконцентрироваться на написании замечательных приложений, которыми и пользователи будут довольны!

Джеффри Грозенбах, автор PeepCode Publishing и голос на подкастах, посвященных Ruby on Rails

“c7Fd9uk3nu4ck0td8iv9oz1nv0ak5ljjSw2iv6mu9pz1ly3im0cq7il4ta2y”.gsub(/\w\d/,

’’).gsub(‘jj’,’ ‘)

Джереми Хуберт, уж какой есть

Работа с Rails – это как программирование на пару с талантливым, опытным и имеющим собственное мнение веб-разработчиком. Если вы готовы смирить гордыню и следовать по пути Rails, наградой будет колоссальное увеличение продуктивности. Если у вас имеется предвзятое мнение о веб-разработке и проектировании баз данных, неизбежны споры, а о продуктивности можно будет забыть. Поначалу разработка в Rails означает обуздание себя, готовность уступить мнению Rails и смотреть на веб-разработку глазами Rails. Но потерпите! Когда вы освоите Ruby, споры с Rails все чаще станут заканчиваться в вашу пользу. В конечном итоге разработка в Rails сводится к такой цепочке: прислушаться к мнению Rails ⇒ определить свои приоритеты ⇒ пользуясь мощью Ruby, подчинить Rails своей воле.

Дэйв Гувер, мастер по программам

В начале лекций по SICP (Structure and Interpretation of Computer Programs – структура и интерпретация компьютерных программ) – знаменитого курса по функциональному программированию – Гарольд Абельсон так определяет, что такое computer science (информатика):

> *Computer science – никуда не годное название. Во-первых, это не наука (*science*); скорее, инженерная дисциплина или искусство... К тому же эта дисциплина не исчерпывается компьютерами, как физика не ис-*

черпывается ускорителями частиц, биология – микроскопами и чашками Петри, а геометрия – измерительными инструментами.

Мы думаем, что computer science – наука о компьютерах по той же причине, по которой древние египтяне считали, что обмер делянок после нильских наводнений – предмет, касающийся измерительных инструментов, а не геометрии:

Когда некоторая область знаний только зарождается, и мы еще не вполне ее понимаем, то очень легко спутать сущность с используемыми инструментами.

Точно так же, я полагаю, что Путь Rails на самом деле не связан ни с Rails, ни с Ruby, ни с новыми инструментами, которыми мы пользуемся. Его «сущность» – бескомпромиссное желание увеличить продуктивность и сделать людей счастливыми. Это прагматизм, доставшийся нелегким путем: набор процессов для людей, которые не желают решать одну и ту же задачу дважды, которых настолько раздражают задержки, вносимые инструментами, что они счастливы поддать газу паровому катку.

Из мысли о том, что секрет Rails не в коде его дистрибутива, с необходимостью следует, что придет день, когда эти практики получат еще более достойное воплощение. Я люблю Ruby, Rails и сформировавшееся вокруг них сообщество, но знаю, что движение вперед продолжается. И когда этот день настанет, и какой-нибудь 10-минутный ролик заставит нас визжать от восторга, у нас возникнет желание прыгнуть туда очертя голову с той же отчаянной решимостью, с которой мы отбросили все то, чем *привыкли* зарабатывать на жизнь.

Крис Кампмейер,
блог на сайте http://www.shiftcommathree.com

Если DHH так не делает, то и вы не делайте (похоже, что всякий раз, когда какой-нибудь умник попадает впросак, причина именно в этом).

Зед Шоу, автор веб-сервера Mongrel

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-137-0, название «Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Алфавитный указатель

Symbols

#, ограничитель 306
-, ограничитель 306
307, код переадресации 73

A

AccountController, класс, Acts_as_ Authenticated, метод 466
Action Cache, подключаемый модуль 329
ActionMailer, структура, конфигурирование 494
ActionView 303
active_record_defaults, подключаемый модуль 734
ActiveRecord
  абстрактные базовые классы моделей 290
  классы, модификация во время выполнения 299
  наблюдатели 282
    регистрация 283
    соглашения об именовании 282
  наследование с одной таблицей 283
    отображение наследования на базу данных 285
  обратные вызовы 272
    after_find 278
    after_initialize 278
    классы 279
    прерывание выполнения 275
    примеры 275
  общее поведение, повторное использование 294
  полиморфные отношения
    has_many 291
ActiveRecord
  регистрация обратного вызова 273
    обратные вызовы before/after 274
Active Record, структура 157, 175
  CRUD 179
  Migration API 164
  атрибуты
    значения по умолчанию 177
    сериализованные 179
  блокировка базы данных 194
    оптимистическая 194
    пессимистическая 196
  конфигурирование 208
  кэш запросов 187
  методы в стиле макросов 171
    объявление отношений 172
    приведение к множественному числу 173
    примат соглашения над конфигурацией 173
  методы поиска
    параметры 200
    упорядочение результатов 199
    условия 198
  миграции 160
  определение колонок 166
  соединения с базами данных 203
  схемы именования 175
ActiveRecord, модели, создание форм 351
ActiveRecordHelper, модуль 333
ActiveRecord SessionStore 446
ActiveResource 485
  Create, метод 487
  Delete, метод 489
  Find, метод 486
  Update, метод 489
  заголовки 490
  настройка 491

ActiveSupport, библиотека
Array, класс 663
Class, объект 668
Dependencies 677
атрибуты модуля 677
открытые методы экземпляра 679
Deprecation, модуль 681
Enumerable, модуль 683
Exception, модуль 684
FalseClass, модуль 685
File, модуль 685
Hash, модуль 686
Inflections, класс 692
Module, класс 698
NilClass, класс 704
Numeric, класс 705
Object, класс 708
Proc, класс 712
Range, класс 713
String, класс 714
Symbol, класс 719
Time, класс 721
форматер протокола по умолчанию 697
acts_as_authenticated, подключаемый модуль
AccountController, класс 466
authenticate, метод 464
before_save, обратный вызов 464
remember_token, маркер 465
User, модель 458
валидаторы 463
получение имени пользователя из cookies 468
текущий пользователь 469
установка 457
after_find, обратный вызов 278
after_initialize, обратный вызов 278
Ajax, Prototype 401
Class, объект 405
Enumerable, объект 416
FireBug 402
Hash, класс 421
Object, класс 406
Prototype, объект 422
Responders, объект 415
функции верхнего уровня 403
alert(), метод (RJS) 432
API
Migration 164
Prototype 403
RSelenese 550
частичные сценарии 550
API
тестов сопряжения 543
application.rhtml, макет 308
Around-фильтры 78
Array, класс
to_xml, метод 478
библиотека ActiveSupport 663
расширения JavaScript 407
assert_block, утверждение 523
AssetTagHelper, модуль 339
AssociationProxy, класс 253
attributes, метод 183
authenticate, метод подключаемого модуля acts_as_authenticated 464
AuthenticatedTestHelper, модуль 470
auto_discovery_link_tag, метод 339
auto_link, метод 378
Autotest, проект 575

B

BackgrounDRb
добавление фоновой обработки в приложение 654
BAM
бизнес-деятельности. *См.* Мониторинг
BDD (разработка, управляемая поведением 564
before/after, обратные вызовы 274
before_save, обратный вызов, Acts as Authenticated 464
belongs_to, ассоциация 218
параметры 220
benchmark, метод 343
BenchmarkHelper, модуль 343
BNL (естественный язык бизнеса) 301
breadcrumbs, помощник 392
Builder API 480
button_to, метод 384

C

call(), метод (RJS) 432
capify, команда 629
Capistrano 625
capify, команда 629
database.yml, сохранение 636
конфигурирование 616
настройка шлюза 648
переменные 639
подготовка к развертыванию 632
символические ссылки 633

Capistrano 625
предварительная инициализация 641, 642
развертывание 631, 634
на нескольких серверах 645
с помощью :copy 635
рецепты, управление кластерами Mongrel 643
система управления версиями, изменение 635
схема базы данных, загрузка 638
сценарий spawner 632
транзакции 646
требования 627
удаленные учетные записи пользователя 635
установка 609, 629
CaptureHelper, модуль 344
CEO. См. Высшее исполнительное лицо
check_box_tag, метод 365
Class, объект (Prototype) 405
color, gem-пакет 733
concat, метод 378
content_tag_for, метод 376
Cookies 453
чтение и запись 454
CookieStore, хранилище сеансов 449
Create, метод (ActiveResource) 487
CRUD (Create Read Update Delete) 117
обновление 189
создание 179
удаление 193
чтение 180
current_page?, метод 385
cycle, метод 379

D

Daemons, добавление фоновой обработки в приложение 659
database.yml, сохранение 636
DateHelper, модуль 345
debug_view_helper, подключаемый модуль 734
DebugHelper, модуль 351
delay(), метод (RJS) 433
DELETE, запросы, обработка в REST 122
Delete, метод (ActiveResource) 489
destroy(), метод 193
discover, команда 582
distance_of_time_in_words(), метод 349
div_for(), метод 376
dom_class(), метод 375
dom_id(), метод 375
draggable(), метод (RJS) 433
DRb (Distributed Ruby)
добавление фоновой обработки в приложение 652
хранение сеансов 447
drop_receiving(), метод (RJS) 433

E

end_form_tag(), метод 365
Enumerable, объект (Prototype) 416
environment.rb, файл
TZ, переменная окружения 47
переопределение параметров 45
переопределение уровня протоколирования 46
режим эксплуатации 39
ERb (Embedded Ruby) 304
ограничители 306
erb, команда 304
error_messages_for(), метод 334
error_messages_on(), метод 334
Event, класс (JavaScript), расширения 409
exception_logger, подключаемый модуль 734
excerpt(), метод 380
externals, свойства 585

F

file_field_tag(), метод 365
Find, метод (ActiveResource) 486
FireBug 402
form(), метод 336
form_tag(), метод 366
FormHelper, модуль 351
фиктивные акцессоры 357
FormOptionsHelper, модуль 359
Function, класс (JavaScript), расширения 410

G

H

h(), метод 314
has_and_belongs_to_many(), метод 233
колонки, добавление в связующие таблицы 239
специальные параметры для SQL 236

has_finder, подключаемый модуль 734
has_many, ассоциация 225
  методы прокси-классов 232
  параметры 225
has_many_polymorphs, подключаемый модуль 735
has_one, ассоциация 247
  параметры 249
Hash, класс (Prototype) 421
hide(), метод (RJS) 433
highlight(), метод 380
HTML
  генерация тегов 376
  генерация тегов input 365
  генерация формы 365
  добавление информации в тег HEAD 339
  методы-помощники select 359
  модуль TagHelper 376
  отправка сообщений 498
  помощники для создания календарных данных 345
HTTP, коды состояния 69
  для переадресации 73
HTTP-запросы, обработка в REST 122
HTTPS, безопасные сеансы 446
Hyett, PJ 733

I

image_path(), метод 340
image_tag(), метод 341
Inflections, класс (ActiveSupport) 692
Inflector, класс 479
init.rb, файл 590
initializer.rb, файл 42
  подразумеваемые пути загрузки 42
init сценарии, конфигурирование
  Mongrel 618
  Monit 619
  Nginx 616
input(), метод 337
insert_html(), метод (RJS) 433
install, команда 582
install.rb, файл 594
invalid?(), метод 268

J

JavaScript
  JSON 435
  RJS. См. RJS
JavaScript
  расширения класса Array 407
  расширения класса Event 409
  расширения класса Function 410
  расширения класса Number 412
javascript_include_tag(), метод 341
javascript_path(), метод 342
JavaScriptHelper, модуль 368
JSON (JavaScript Object Notation) 401, 435, 694
  константы 694
  методы класса 695
  рендеринг 68

K

Kernel, класс
  открытые методы экземпляра 696
KPI. См. Основные показатели эффективности (KPI)

L

link_to(), метод 385
link_to_remote(), метод 422
list, команда 580
literal(), метод (RJS) 434

M

mail_to(), метод 387
markdown(), метод 381
memcache, хранилище сеансов 448
Migration API 164
MIT-LICENSE, файл 593
Mocha, библиотека 510
  заглушки 510
mock_model, метод 569
mocks, папка, добавление классов 509
Module, класс 698
Mongrel 600
  сценарий init, конфигурирование 618
  установка 607
Mongrel Cluster
  конфигурирование 609
  установка 607
Monit
  конфигурирование 614
  сценарий init 619
  установка 608
MySQL, установка 608

N

Nginx
конфигурирование 610
сценарий init 616
установка 607
NilClass 704
Null-объекты 561
Number, класс (JavaScript), расширения 412
NumberHelper, модуль 370
Numeric, класс (библиотека ActiveSupport) 705

O

Object, класс (Prototype) 406
Object, класс (библиотека ActiveSupport) 708
observe_field, метод 428
observe_form, метод 429
on(), метод 268
OOS. См. Нестандарт

P

PaginationHelper, модуль 372
partial_path, метод 375
PeepCode, демо-ролики 736
periodically_call_remote, метод 427
photo_for, метод 391
Piston
внешние библиотеки
импорт 587
конвертация 588
обновление 588
ревизии, блокировка и разблокировка 588
свойства 589
установка 586
pluralize, метод 381
Proc, класс (библиотека ActiveSupport) 712
Prototype, библиотека 401
Ajax, объект 415
Class, объект 405
Enumerable, объект 416
FireBug 402
Hash, класс 421
Object, класс 406
Prototype, объект 422
Responders, объект 415
функции верхнего уровня 403
PrototypeHelper, модуль
link_to_remote, метод 422
observe_field, метод 428
observe_form, метод 429
periodically_call_remote, метод 427
remote_form_for, метод 426
PSA
запуска проекта. См. Структура
PStore, формат файлов 447
PUT, запросы, обработка в REST 122

Q

query_trace
подключаемый модуль 735

R

RAD. См. Быстрая разработка приложений
Rake, задания 595
Rakefileы 596
для SVN 737
заморозка и разморозка 40
относящиеся к тестированию 544
Range, класс (ActiveSupport) 713
RCov 576
README, файл 593
RecordIdentificationHelper, модуль 374
RecordInvalid, исключение 264
RecordTagHelper, модуль 375
redirect_to(), метод (RJS) 434
redirect_to, команда 72
reload, метод 185
remember_token, маркер (Acts as Authenticated) 465
remote_form_for, метод 426
remove(), метод (RJS) 434
remove, команда 583
render, команда 65
replace(), метод (RJS) 434
replace_html(), метод (RJS) 434
reset_cycle, метод 381
respond_to, метод 97, 138
Responders, объект (Prototype) 415
REST 87, 115, 116
create, действие 144
destroy, действие 143
edit/update, операции 123, 146
HTTP-запросы 120
new/create, операции 123
show, действие 143
действия контроллеров 121

REST 87, 115, 116
  контроллеры
    возвраты 135
    явное задание 129
  множественные маршруты 123
  настройка маршрутов
    дополнительные маршруты к наборам 134
    маршруты к дополнительным действия 133
  одиночные маршруты 123, 124
  ресурсы 118
    ассоциированные только с контроллером 136
    вложенные 125
    представления 138
  синтаксис нестандартных действий 134
  синтаксический уксус 136
  форматированные именованные маршруты 139
REXML 482
RJS (Ruby JavaScript) 429
  alert(), метод 432
  call(), метод 432
  delay(), метод 433
  draggable(), метод 433
  drop_receiving(), метод 433
  hide(), метод 433
  insert_html(), метод 433
  literal(), метод 434
  redirect_to(), метод 434
  remove(), метод 434
  replace(), метод 434
  replace_html(), метод 434
  select(), метод 435
  show(), метод 435
  sortable(), метод 435
  toggle(), метод 435
  visual_effect(), метод 435
  тестирование поведения 539
Routing Navigator, подключаемый модуль 155
RSelenese 550
Rselenese
  частичные сценарии 550
RSpec 507
  Autotest, проект 575
  mock-объекты 561
  null-объекты 561
  RCov 576
  обстраивание 575
RSpec 507
  объекты-заглушки 562
  ожидания 554
  ожидания, нестандартные верификаторы 556
  поведения 557
    разделяемые 558
  прогон спецификаций 564
  произвольные предикаты 554
  спецификации контроллеров 570
  спецификации помощников 574
  спецификации представлений 573
    заглушки
      для методов-помощников 574
    присваивание значений переменным экземпляра 573
  сценарии 553
  установка 566
  формат спецификаций 564
  частичные подделки и заглушки 562
RSpec on Rails 566
  генераторы 566
  ошибки, специфицирование 572
  режим изоляции 571
  режим интеграции 571
  спецификации модели 567
  специфицирование маршрутов 572
Ruby, установка 606
RubyGems 42
  установка 606

S

sanitize, метод 382
script/plugin, команда 580
Scriptaculous
  перетаскивание мышью 437
  редактирование на месте 440
  сортируемые списки 439
SDLC
  цикла разработки ПО. *См.* Метод
select(), метод (RJS) 435
Selenium 547
  RSelenese, частичные сценарии 550
  действия 547
  локаторы 548
  образцы 548
  утверждения 548
Selenium on Rails 548
send_data(), метод 81
send_file(), метод 82
show(), метод (RJS) 435

simple_format, метод 382
SLA. См. Соглашение об уровне обслуживания (SLA)
SMTP-сервер, конфигурирование 505
sortable(), метод (RJS) 435
sources, команда 581
SpiderTester, сценарий 735
SQL
  специальные запросы 186
  специальные параметры 236
String, класс (ActiveSupport) 714
String, класс (JavaScript), расширения 413
strip_links, метод 382
strip_tags, метод 383
stylesheet_link_tag(), метод 342
stylesheet_path(), метод 343
Subversion 736
  веб-сайт 584
  внешние источники 585
  задания Rake 737
  обновление подключаемых модулей 585
  установка 608
Sweeper, класс 328
Symbol, класс 719
Syslog 58

T

TagHelper, модуль 376
TextHelper, модуль 378
textilize, метод 383
textilize_without_paragraph, метод 383
tiles, помощник, написание 393
Time, класс (AcriveSupport) 721
TMail 502
  вложение файлов, прием 504
  методы 503
to_xml, метод 472
  :include, параметр 475
  :methods, параметр 476
  :procs, параметр 477
  класс Array 478
  настройка результата работы 473
  переопределение 478
toggle(), метод (RJS) 435
truncate, метод 383
TZ, переопределение в файле environment.rb 47

U

uninstall.rb, файл 595
unsource, команда 582
Update, метод (ActiveResource) 489
URL, генерация 151
url_for, метод 104, 388
UrlHelper, модуль 384
User, модель (Acts as Authenticated) 458

V

validates_acceptance_of, метод 257
validates_associated, метод 258
validates_confirmation_of, метод 258
validates_each, метод 259
validates_exclusion_of, метод 259
validates_existence_of, метод 260
validates_format_of, метод 261
validates_inclusion_of, метод 259
validates_length_of, метод 262
validates_numericality_of, метод 262
validates_presence_of, метод 262
validates_uniqueness_of, метод 263
visual_effect(), метод (RJS) 435

W

will_paginate
  подключаемый модуль 372
word_wrap, метод 384

X

XML
  Builder API 480
  to_xml, метод 472
    :include, параметр 475
    :methods, параметр 476
    :procs, параметр 477
    настройка результата работы 473
  преобразование в хеши Ruby 482
  разбор 483
  рендеринг 68
XmlSimple, библиотека 483
XUnit, структура 513

Y

YAML (Yet Another Markup Language) 515
yield, ключевое слово 308

А

абстрактные базовые классы моделей 290
автозавершение 439
автозагрузка классов и модулей 44
автоматическая перезагрузка классов 50
автономные тесты 527
Адам, Джеймс 598
анализ протоколов 56
аргументы
  для именованных маршрутов 111
  синтаксис хешей 154
ассоциации 211
  belongs_to 218
    параметры 220
  has_many 225
    методы прокси-классов 232
    параметры 225
  добавление объектов в набор 215
  иерархия классов 211
  несохраненные 251
  отношения многие-ко-многим 233
    has_and_belongs_to_many, метод 233
    through, ассоциация 240
  отношения один-ко-многим 213
  отношения один-к-одному 247
    has_one, ассоциация 247
  расширения 252
ассоциированные объекты, проверка наличия 262
атаки внедрением, защита от 313
атрибуты
  значения по умолчанию 177
  сериализованные 179

Б

Бак, Джеймис 131, 151, 734
Бейтс, Райан 736
Билкович, Уилсон 254
блокировка базы данных 194
  оптимистическая 194
  пессимистическая 196

В

валидаторы, методы
  allow_nil, параметр 265
  message, параметр 265
  on, параметр 265
  нестандартный контроль 268
  условная проверка 266
    когда применять 266
включение сеансов 445
вложения файлов
  отправка 501
  прием 504
вложенные ресурсы 125, 130
  :name_prefix, параметр 127
  :path_prefix, параметр 127
  глубокая вложенность 131
возвраты 135
встройки 44

Г

гарантии уникальности модели соединения 263
генераторы 566
генерация
  HTML-тегов 365
    модуль TagHelper 376
  маршрута по умолчанию 96
  маршрутов 151
  фикстур из данных, используемых в режиме разработки 518
глубокая вложенность 131
Грозенбах, Джеффри 733

Д

двусторонние отношения 235
действия (Selenium) 547
демо-ролики 736
диспетчер 62
добавление
  источников подключаемых модулей 581
  классов в папку mocks 509
  путей загрузки 45
дублирование схемы 46
Дэвис, Райан 575

З

заглушки 510, 562
  частичные 562
заголовки (ActiveResource), задание 490
загрузчик классов 50
заморозка и разморозка приложения 40

запросы
диспетчер 62
переадресация 71

И

именованные маршруты 108
аргументы 111
выполнение в консоли приложения 153
создание 108
методы-помощники 108
импорт внешних библиотек в Piston 587
индикатор
зеленый 514
красный 514

К

классы
AssociationProxy 253
абстрактные базовые моделей 290
автозагрузка 44
добавление в папку mocks 509
обратных вызовов 279
расширение 592
фильтров 77
код точки расширения 590
козярский, Майкл 175
колонки
добавление в связующие таблицы для has_and_belongs_to_many 239
определение 166
комплекты тестов 514
консоль приложения
маршруты
именованные, выполнение 153
распечатка 148
распознавание и генерация 151
объект Route 149
константы, JSON 694
контроллеры 60
REST-совместимые действия 121
явное задание 129
session, метод класса 444
возвраты 135
и пространства имен 132
переадресация 71
переменные экземпляра 74
потоковая отправка 81
send_data(), метод 81
send_file(), метод 82
контроллеры 60
рендеринг шаблонов 65
представления 64
спецификации 570
фильтры 75
around 78
внешние классы 77
встраивание 77
наследование 76
прерывание цепочки 81
пропуск цепочки 80
упорядочение цепочки 78
условные 80
функциональное тестирование 529
методы 531
утверждения 531
кэш запросов 187
кэширование 320
Action Cache, подключаемый модуль 329
действий 321
истечение срока хранения
содержимого 326
класс Sweeper 328
фрагментов 327
протоколирование работы кэша 329
страниц 321
фрагментов 323
глобальные фрагменты 325
именованные фрагменты 324
хранилища 330

Л

локаторы (Selenium) 548
лямбда-выражения 397

М

макеты
шаблон application.rhtml 308
макросы 75, 171
объявление отношений 172
приведение к множественному числу 173
примат соглашения над конфигурацией 173
Марклунд, Петер 726
маршруты
routes.rb, файл 93
маршрут по умолчанию 94
предпоследний маршрут 97

маршруты
url_for, метод 104
именование 109
именованные 108
аргументы 111
выполнение в консоли приложения 153
создание 108
создание с помощью метода with_options 112
литеральные URL 106
метапараметры 91
нестандартные, написание 100
порядок обработки 102
пустые 99
распечатка 148
распознавание и генерация вручную 151
регулярные выражения 103
связанные параметры 90
синтаксис 89
статические строки 91, 100
тестирование 153
умолчания, параметр :id 105
маршруты по умолчанию (routers.rb) 94
генерация 96
модификация 97
поле
id 95
предпоследний маршрут 97
метод respond_to 98
маскирование маршрутов 106, 107
маскирование пар ключ/значение 107
машина развертывания (Capistrano), подготовка 632
метапараметры шаблонов 91
метапрограммирование 172
методы
attributes 183
auto_discovery_link_tag 339
auto_link 378
benchmark 343
button_to 384
check_box_tag 365
concat 378
content_tag_for 376
Create 487
current_page? 385
cycle 379
delete 489
destroy 193
distance_of_time_in_words 349
методы
div_for 376
dom_class 375
dom_id 375
end_form_tag 365
error_message_on 334
error_messages_for 334
excerpt 380
file_field_tag 365
find 486
form 336
form_tag 366
h 314
has_and_belongs_to_many 233
highlight 380
image_path 340
image_tag 341
input 337
invalid? 268
javascript_include_tag 341
javascript_path 342
link_to 385
mail_to 387
markdown 381
mock_model 569
on 268
partial_path 375
pluralize 381
reload 185
render 68
reset_cycle 381
respond_to 97
представления ресурсов 138
sanitize 382
send_data 81
send_file 82
simple_format 382
strip_links 382
strip_tags 383
stylesheet_link_tag 342
textilize 383
textilize_without_paragraph 383
TMail 503
to_xml 472
include, параметр 475
methods, параметр 476
procs, параметр 477
настройка результата работы 473
переопределение 478
truncate 383
update 489

методы
url_for 104, 388
word_wrap 384
контроля
:allow_nil, параметр 265
:message, параметр 265
:on, параметр 265
RecordInvalid 264
validates_acceptance_of 257
validates_associated 258
validates_confirmation_of 258
validates_each 259
validates_exclusion_of 259
validates_existence_of 260
validates_format_of 261
validates_inclusion_of 259
validates_length_of 262
validates_numericality_of 262
validates_presence_of 262
validates_uniqueness_of 263
нестандартный 268
условная проверка 266
макросы 75, 171
очистки (teardown) 514
подготовки (setup) 514
помощники 307
написание 390
почтальона. См. Почтальона методы
утверждения 523
assert_block 523
функциональных тестов 531
методы-помощники 307
заглушки 574
методы поиска
параметры 200
по атрибутам 185
упорядочение результатов 199
условия 198
миграции 160
именование 162
определение колонок 166
подвохи 162
создание 161
многочастные сообщения
неявные 501
отправка 499
множественные REST-совместимые маршруты 123
модели почтальона 494
создание 495
модели соединения, гарантии уникальности 263
модули-помощники, написание 390

H

наблюдатели 47, 282
регистрация 283
соглашения об именовании 282
наборы
добавление ассоциированных объектов 215
прокси 213
рендеринг 319
наследование
с одной таблицей (STI) 283
отображение на базу данных 285
фильтров 76
начальная загрузка 40
initializer.rb, файл 42
подразумеваемые пути загрузки 42
автозагрузка классов и модулей 44
версия Rails Gem 39
пакеты RubyGem 42
переопределение режима 39
несохраненные ассоциации 251
неявные многочастные сообщения 501

O

обработка запроса 61
образцы (Selenium) 548
обратные вызовы 272
обстраивание 575
объектно-реляционное отображение, структура, ActiveRecord
абстрактные базовые классы моделей 290
классы, модификация во время выполнения 299
наблюдатели 282
наследование с одной таблицей 283
обратные вызовы 272, 275
прерывание выполнения 275
регистрация 273
обратные вызовы before/after 274
общее поведение, повторное использование 294
полиморфные отношения has_many 291
объекты
добавление в набор 215
ошибки, нахождение 256
хранение в сеансе 443
ограничители 304, 305
удаление пустых строк 306

одиночные REST-совместимые маршруты 123, 124
ожидания 554
нестандартные верификаторы 556
Олсон, Рик 155, 457
оптимистическая блокировка 194
«Острие Rails» 41
заморозка и разморозка приложения 40
отключение сеансов для роботов 445
отношения многие-ко-многим 233
отношения один-к-одному 247
отношения один-ко-многим 213
отправка
HTML-сообщений 499
многочастных сообщений 499
файлов во вложении 501
электронной почты 502
отслеживание сеансов 452
ошибки 267
набор Errors, манипулирование 268
нахождение 256
нестандартный контроль 268
отказ от контроля 270
условная проверка 266

П

параметры
ассоциации belongs_to 220
ассоциации has_many 225
ассоциации has_one 249
ассоциации through 244
методов почтальона 496
переадресация
запросы 71
коды состояния HTTP 73
Capistrano 639
переменные 310
передача подшаблонам 317
экземпляра 310
переменные экземпляра 74, 310
в спецификации представления, присваивание значений 573
перетаскивание мышью 437
пессимистическая блокировка 196
поведения 553, 557
разделяемые 558
повторное использование
кода 579
общего поведения 294
подшаблонов 316
подделывание
mock-объекты 561
частичное 562
подключаемые модули 579
внешние источники Subversion 585
выгрузка из Subversion 584
добавление источников 581
задания Rake 595
написание 589, 593
подключаемые модули 579
обновление 585
переустановка 583
список 580
тестирование 597
удаление 583
удаление источников 581
установка 580
фиксация версии 586
подразумеваемые пути загрузки (initializer.rb) 42
подшаблоны 314
передача переменных 317
повторное использование 316
протоколирование 320
разделяемые 316
рендеринг 67
рендеринг наборов 319
полиморфные ассоциации 291
поразумеваемый форматер протоколов 697
построение промышленной системы, необходимые компоненты 603
инструменты мониторинга 605
ярус базы данных 605
ярус веб-сервера 604
ярус сервера приложений 604
потоковая отправка 81
почтальона методы 496
HTML-сообщения, отправка 499
вложение файлов, получение 501
многочастные сообщения, отправка 499
параметры 496
почта
отправка 502
получение 503
правила маршрутизации, тестирование 541
предикаты 555
предпоследний маршрут 97
метод respond_to 98
представления ресурсов 138

преобразование
XML в хеши Ruby 482
внешних библиотек в команды Piston 588
числовых данных
в отформатированные строки, модуль NumberHelper 370
приемники 91, 101
приемочные тесты 545
примат соглашения над конфигурацией 65
принцип одной функции 282
присваивание значений переменным экземпляра спецификации представления 573
производительностб приложения, анализ 56
произвольные предикаты 554
прокси-наборы 213
промышленное окружение 600
компоненты 603
вопросы производительности 621
избыточность 621
инструменты мониторинга 605
масштабируемость 621
предварительные условия 601
ярус базы данных 605
ярус веб-сервера 604
ярус сервера приложений 604
кэширование 621
протоколы 55
syslog 58
анализ 56
вывод подшаблонов 320
уровень протоколирования 54
пустые маршруты 99
пути загрузки 51
дополнительные 45
загрузчик классов 51

P

развертывание
Capistrano 631, 634
на нескольких серверах 645
с помощью :copy 635
промышленное
замечания по поводу 621
установка Capistrano 609
установка Mongrel 607
установка Monit 608
установка MySQL 608
развертывание
промышленное
установка Nginx 607
установка Ruby 606
установка RubyGems 606
установка Subversion 608
распечатка маршрутов 148
расширение ассоциаций 252
расширения классов 592
регистрация обратного вызова 273
before/after, обратные вызовы 274
регулярные выражения в маршрутах 103
редактирование на месте 440
режим изоляции (RSpec on Rails) 571
режим интеграции (RSpec on Rails) 571
режим разработки 49
автоматическая перезагрузка классов 50
загрузчик классов 50
кэширование 321
фикстуры 518
режим тестирования 52
режим эксплуатации 39
рендеринг
наборов 319
представления 64
шаблонов 65
встроенных 67
подшаблонов 67
структурированных данных 68
текста 67
ресурсы (REST) 118
ассоциированные только с контроллером 136
вложенные 125
name_prefix, параметр 127
path_prefix, параметр 127
глубокая вложенность 131
представления 138
рецепты (Capistrano), управление кластером Mongrel 643
роботы, отключение сеансов 445

C

самоссылающиеся отношения 234
Сассер, Джош 134, 240, 357
связанные параметры маршрутов 90
сеансы 442, 544
избирательное включение 445
истечение срока хранения 451
обеспечение безопасности 446, 453
отключение для роботов 445

сеансы 442, 544
отслеживание 452
способы организации 444
удаление 453
сериализованные атрибуты 179
синонимы 733
синтаксическая глазурь 136
синтаксический уксус 136
соглашения об именовании
в схеме базы данных 175
для маршрутов 109
для миграций 162
для наблюдателей 282
создание
именованных маршрутов 108
методы-помощники 108
с помощью метода with_options 112
миграций 161
модели почтальона 495
сортируемых списков 439
сообщения
об ошибках 267
проверка 268
сортируемые списки, создание 439
спецификации 553
контроллеров 570
модели 567
ожидания 554
помощников 574
представлений 573
прогон 564
статические строки 91, 100
стефенсон, Сэм 401
структурированные данные, рендеринг 68
суффиксы шаблонов 307
схема базы данных
структура Active Record 158, 175
CRUD 179
Migration API 164
атрибуты 176
блокировка базы данных 194
конфигурирование 208
кэш запросов 187
методы в стиле макросов 171
методы поиска 197
миграции 160
соглашения об именовании 175
соединения 203
сценарий начальной загрузки, initializer.rb 42

Т

теги формы, генерация 365
тестирование 513
Selenium. См. Selenium
автономное 527
задания Rake 544
маршрутов 153
отказы 514
ошибки 514
подключаемых модулей 597
приемочные тесты 545
семейство xUnit 513
сопряжений 511, 542
утверждения 523
assert 523
принцип одного утверждения 526
фикстуры 515
в режиме разработки 518
в формате CSV 516
генерация из данных, используемых в режиме разработки 518
динамические данные 517
недостатки 520
параметры 520
функциональные тесты 529, 535
виды сравнения 539
методы 531
методы выборки 540
поведение RJS 539
правила маршрутизации 541
утверждения 531
тесты сопряжения 511, 542
и сеансы 544
транзакции (Capistrano) 646

У

удаление
источников подключаемых модулей 581
подключаемых модулей 583
пустых строк 306
удаленные учетные записи пользователя 635
Уильямс, Брюс 373
унаследованные схемы именования 175
Уонстрэт, Крис 733
уровень протоколирования, переопределение 46
условная проверка 266
когда применять 266

условный вывод 306
установка подключаемых модулей 580
Acts as Authenticated 457
Routing Navigator 155
утверждение 513, 523
для функциональных тестов 531
принцип одного утверждения 526

Ф

Фаулер, Мартин 157
фиксация версии подключаемого модуля 586
фикстуры 515
в режиме разработки 518
в формате CSV 516
генерация из данных, используемых в режиме разработки 518
динамические данные 517
недостатки 520
параметры 520
транзакционные 520
фиктивные акцессоры 357
Филдинг, Рой Т. 115
Филдс, Джей 526
фильтры 75
around 78
внешние классы 77
встраивание 77
наследование 76
прерывание цепочки 81
пропуск цепочки 80
упорядочение цепочки 78
условные 80
фоновая обработка
BackgrounDRb 654
Daemons 659
DRb 652
script/runner 650
функциональные тесты 529, 535
виды сравнения 539
методы 531
методы выборки 540
поведение RJS, тестирование 539
правила маршрутизации, тестирование 541
утверждения 531

Х

Хелмкамп, Брайан 734
хнычущий nil 704
Ходель, Эрик 575
хранение
идентификатора текущего пользователя 443
объектов 443
сеансов
в ActiveRecord SessionStore 446
в CookieStore 449
в DRb 447
в memcache 448
в PStore 447
файла database.yml 636

Ч

часовые пояса 726
частичные сценарии 550

Ш

шаблоны
application.rhtml, шаблон макета 308
RJS 431
ограничители 304
переменные 310
экземпляра 310
подшаблоны 314
передача переменных 317
повторное использование 316
протоколирование 320
разделяемые 316
рендеринг наборов 319
рендеринг 65
структурированные данные, рендеринг 68
суффиксы 307
текст, рендеринг 67
Шоу, Зед 600

Э

электронная почта
вложение файлов
отправка 501
прием 504
конфигурирование SMTP-серверов 505
методы TMail 503
Эстелс, Дэйв 526