

JRuby

Сборник рецептов



Джастин Эдельсон
Генри Лю

Джастин Эдельсон, Генри Лю

JRuby Сборник рецептов

Перевод с английского А. Слинкина



УДК 32.973.26-018.2

ББК 004.438

Э19

Э19 Джастин Эдельсон, Генри Лю

JRuby. Сборник рецептов. – М.: ДМК Пресс, 2010. – 240 с.: ил.

ISBN 978-5-94074-589-1

Интересующемуся языком JRuby вряд ли нужны руководства по Ruby, Rails или Java – требуется лишь узнать, как решить ту или иную конкретную задачу. В этой книге вы найдете практические рецепты по работе с данной реализацией языка Ruby на платформе Java. Специально рассмотрены такие вопросы, как развертывание веб-приложений Rails на Java-серверах, интеграция программ на JRuby с технологиями из мира Java, разработка персональных приложений на JRuby с помощью инструментальных библиотек на Java и многое другое. Интерпретатор JRuby сочетает в себе простоту и легкость использования Ruby с развитыми библиотеками и технологиями Java, и это открывает совершенно новые возможности для применения Ruby, Rails и Java.

Эта книга даст вам возможность воспользоваться всем потенциалом языка JRuby.

УДК 519.6

ББК В162я73

Authorized translation of the English edition of **JRuby Cookbook** ISBN 9780596519803 © 2009, Justin Edelson and Henry Liu. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-596-51980-3

Copyright © 2009 Justin Edelson, Henry Liu.

All right reserved

ISBN 978-5-94074-589-1

© Оформление, издание, ДМК Пресс, 2010

Краткое содержание

ПРЕДИСЛОВИЕ	8
ГЛАВА 1. ПРИСТУПАЯ К РАБОТЕ	13
ГЛАВА 2. JRUBY ON RAILS	49
ГЛАВА 3. ИНТЕГРАЦИЯ С JAVA	85
ГЛАВА 4. JAVA В СИСТЕМАХ МАСШТАБА ПРЕДПРИЯТИЯ	111
ГЛАВА 5. ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС И ГРАФИЧЕСКИЕ ПРИЛОЖЕНИЯ	147
ГЛАВА 6. ИНСТРУМЕНТЫ СБОРКИ	181
ГЛАВА 7. ТЕСТИРОВАНИЕ	205
ГЛАВА 8. СООБЩЕСТВО ПОЛЬЗОВАТЕЛЕЙ JRUBY	227
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	232

Содержание

Предисловие	8
Предполагаемая аудитория	8
Организация материала	8
Типографские соглашения	9
О примерах кода	10
Доступность на Safari	10
Вопросы и замечания	10
Благодарности	11
Глава 1. Приступая к работе	13
1.0. Введение	14
1.1. Установка JRuby	17
1.2. Управление пакетами с помощью RubyGems	21
1.3. Одновременное использование Ruby и JRuby	22
1.4. Совместное использование gem-пакетов	24
1.5. Обращение к Java-классам из Ruby	25
1.6. Преобразование Ruby-массива в Java-массив	29
1.7. Включение JAR-файлов в путь поиска классов	30
1.8. Расширение Java-класса в Ruby	31
1.9. Реализация Java-интерфейса на Ruby	32
1.10. Открытие Java-классов в JRuby	37
1.11. Конфигурирование Eclipse для разработки на языке JRuby	39
1.12. Конфигурирование NetBeans для разработки на JRuby	43
1.13. Распознавание платформы в приложении JRuby	47
Глава 2. JRuby on Rails	49
2.0. Введение	50
2.1. Установка и настройка Rails	51
2.2. Пакетирование приложения Rails для работы в среде Java EE	54
2.3. Веб-приложения с внешним репозиторием gem-пакетов	56
2.4. Конфигурирование сервлета JRuby-Rack	57
2.5. Пакетирование приложения Rails с источником данных JNDI	58
2.6. Развертывание Rails на сервере Tomcat	59
2.7. Развертывание Rails на сервере JBoss	62
2.8. Развертывание Rails на сервере Jetty	64

2.9. Развертывание Rails с помощью jetty_rails	65
2.10. Развертывание Rails на сервере Mongrel	68
2.11. Развертывание Rails на сервере GlassFish v2	69
2.12. Gem-пакет GlassFish v3	71
2.13. Использование библиотеки ActiveRecord вне Rails	72
2.14. Получение информации о Java-сервлете	74
2.15. Конфигурирование хранилища сеансов	75
2.16. Управление классами, каталогами и прочими файлами, упакованными в WAR-файл	75
2.17. Изменение имени WAR-файла и местоположения рабочей области	77
2.18. Развертывание приложения Rails в корневом контексте	77
2.19. Создание приложения Rails в интегрированной среде Apptana Studio	79
2.20. Доступ к статическим файлам из приложения Rails, работающего в контейнере Java EE	82

Глава 3. Интеграция с Java

3.0. Введение	86
3.1. Выполнение Ruby-кода из Java-приложения	87
3.2. Вызов JRuby с помощью каркаса Bean Scripting Framework	91
3.3. Вызов JRuby с помощью технологии Java Scripting	93
3.4. Протоколирование из Ruby с помощью библиотеки Jakarta Commons Logging	95
3.5. Использование встроенных в Java средств параллельной обработки	97
3.6. Создание методов-акцессоров в духе JavaBean	100
3.7. Написание единообразного кода	101
3.8. Преобразование XML-документов с помощью библиотеки TrAX	102
3.9. Создание пула сред исполнения JRuby	104
3.10. Удаленное управление с помощью технологии JMX	106
3.11. Доступ к платформенно-зависимым библиотекам из JRuby	108

Глава 4. Java в системах масштаба

предприятия	111
4.0. Введение	112
4.1. Создание контекста JNDI	113

4.2. Отправка JMS-сообщений	115
4.3. Получение JMS-сообщений	118
4.4. Реализация компонента Enterprise JavaBean на JRuby	120
4.5. Определение Spring-компонентов на JRuby	123
4.6. Создание самообновляемых Spring-компонентов на JRuby	127
4.7. Встраивание Spring-компонентов, написанных на JRuby	130
4.8. Реализация Aware-интерфейсов Spring в JRuby-объектах	131
4.9. Создание MVC-контроллеров Spring с помощью JRuby	134
4.10. Hibernate и JRuby	137
4.11. Java Persistence API и JRuby	140
4.12. Выполнение вызовов по протоколу SOAP	141
4.13. Упрощение доступа к LDAP-каталогу	143

Глава 5. Пользовательский

интерфейс и графические приложения

5.0. Введение	148
5.1. Создание приложений Swing	148
5.2. Обработка событий Swing	150
5.3. Долго работающие задачи в приложениях Swing	151
5.4. Пакетирование автономных приложений	153
5.5. Пакетирование JRuby-приложений, запускаемых по технологии Web Start	155
5.6. Написание апплетов на JRuby	157
5.7. Манипулирование изображениями	161
5.8. Создание приложений SWT	164
5.9. Доступ к рабочему столу	166
5.10. Доступ к системному лотку	167
5.11. Разработка приложений Swing на предметно-ориентированных языках на базе JRuby	169
5.12. Использование библиотеки Monkeybars для разработки приложений Swing	173
5.13. Создание приложений Qt с помощью JRuby	177

Глава 6. Инструменты сборки

6.0. Введение	182
6.1. Включение Ruby-сценариев в процесс сборки системой Ant	182

6.2. Применение Ruby в условных конструкциях Ant	185
6.3. Написание задания Ant на Ruby	187
6.4. Включение Ruby-сценариев в процесс сборки системой Maven	188
6.5. Написание подключаемого к Maven модуля на JRuby	190
6.6. Сборка Java-проектов с помощью Raven	193
6.7. Ссылка на библиотеки в Raven	195
6.8. Организация частного репозитория Raven	196
6.9. Прогон тестов JUnit с помощью Raven	197
6.10. Сборка Java-проектов с помощью Buildr	198
6.11. Ссылка на библиотеки в Buildr	201
6.12. Сборка с помощью Rake в контексте сервера Hudson	202
6.13. Добавление Ruby-сценария в качестве задачи сервера Hudson	203
Глава 7. Тестирование	205
7.0. Введение	206
7.1. Автономное тестирование Java-кода с помощью Test/Unit	206
7.2. Автономное тестирование Java-кода с помощью библиотеки dust	209
7.3. Автономное тестирование Java-кода с помощью библиотеки Expectations	210
7.4. Тестирование Java-кода с помощью RSpec	212
7.5. Создание mock-объектов с помощью библиотеки Mocha	217
7.6. Модификация пути поиска классов для JtestR	219
7.7. Группировка тестов для JtestR	219
7.8. Аргументы командной строки при запуске JtestR	220
7.9. Совместное использование JtestR и Ant	222
7.10. Совместное использование JtestR и Maven	223
7.11. Повышение производительности JtestR	224
Глава 8. Сообщество пользователей JRuby	227
8.0. Введение	228
8.1. Сборка JRuby из исходных кодов	228
8.2. Отправка извещения о недоработке в JRuby	229
8.3. Списки рассылки JRuby	231
Предметный указатель	232

Предисловие

Язык JRuby – это просто Ruby, воспользовавшийся преимуществами виртуальной машины Java. Он взял все лучшее от Java и дополнил Ruby целым рядом фантастических возможностей.

– *Чарльз Наттер,*
руководитель проекта JRuby,
Twitter, 7 августа 2008

В этом высказывании Чарльз Наттер кратко выразил оба аспекта, которые в последнее время привлекли внимание к проекту JRuby: во-первых, тот факт, что Java – это отличная платформа, пригодная для самых разных языков, а не только собственно для Java, а, во-вторых, рост интереса к языку программирования Ruby. В этой книге мы рассмотрим множество возможных вариантов использования JRuby. Если говорить словами Чарльза, то в одних рецептах речь пойдет о том лучшем, что взято от Java, в других – о фантастических возможностях, привнесенных в Ruby, а в третьих – о том и другом вместе.

Предполагаемая аудитория

Если вы хотите получить от JRuby максимум возможного, то должны свободно переходить от Java к Ruby и обратно. Работая над этой книгой, мы ориентировались на читателя, который в той или иной мере владеет обоими языками, хотя, быть может, одним лучше, чем другим. Поэтому обширного введения вы здесь не найдете, разве что в первой главе, где мы описываем, в каких вопросах Ruby и Java похожи, а в каких – различаются.

Вообще говоря, мы ставили себе целью не рассказать о том или ином уже имеющемся в Ruby или Java средстве, а объяснить, как JRuby позволяет воспользоваться имеющимися средствами или расширить их. Так, рецепты в главе, посвященной JRuby on Rails, рассчитаны на читателя, который уже создал хотя бы одно (работающее) приложение Rails.

Организация материала

Глава 1. *Приступая к работе*

Эта глава представляет собой краткое введение в JRuby, за которым следует описание ряда простейших приемов работы с этим языком, в том числе применение системы управления пакетами RubyGems и техника обращения к Java-коду из программы на Ruby. В конце главы приводятся рецепты, касающиеся установки и настройки различных интегрированных сред разработки (IDE) для работы с JRuby.

Глава 2. *JRuby on Rails*

Эта глава посвящена разнообразным сценариям развертывания приложений Ruby on Rails с использованием JRuby.

Глава 3. *Интеграция с Java*

Эта глава начинается несколькими рецептами о вызове Ruby-кода из Java-программы. Затем следуют рецепты, описывающие обращение из Ruby к таким популярным Java-библиотекам, как Java Native Access (JNA) и Jakarta Commons Logging.

Глава 4. *Java в системах масштаба предприятия*

Все рецепты, собранные в этой главе, относятся к использованию JRuby в таких предназначенных для создания крупномасштабных систем каркасах, как JMS, JNDI, EJB, Spring и Hibernate.

Глава 5. *Пользовательский интерфейс и графические приложения*

В этой главе описываются некоторые основанные на JRuby каркасы для создания графических пользовательских интерфейсов. Сюда же включены рецепты, касающиеся операций над графическими изображениями, апплетов и интеграции с рабочим столом.

Глава 6. *Инструменты сборки*

В этой главе речь пойдет об использовании JRuby для усовершенствования процедуры сборки Java-проектов. Оба наиболее популярных в мире Java инструмента сборки – Ant и Maven – допускают различные способы подключения JRuby. Здесь же вы найдете рецепты, посвященные программам Raven и Buildr, которые ориентированы исключительно на JRuby.

Глава 7. *Тестирование*

Эта глава посвящена главным образом пакету JtestR, который включает в себя JRuby и ряд популярных инструментов тестирования для языка Ruby. Содержащиеся здесь рецепты научат вас писать на Ruby тесты для Java-программ.

Глава 8. *Сообщество пользователей JRuby*

Это последняя глава. В нее включены сведения о том, как можно принять участие в жизни сообщества пользователей JRuby.

Типографские соглашения

В книге применяются следующие соглашения:

Курсив

Таким начертанием выделяются URL, имена каталогов и файлов, параметры. Иногда оно применяется для выделения важной информации.

Моноширинный шрифт

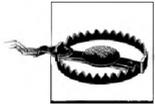
Применяется для листингов программ, а также внутри текста для обозначения элементов программы, как то: имена пространств имен, классов и методов.

Моноширинный курсив

Текст, вместо которого надо подставить значения, вводимые пользователем.



Этот значок обозначает совет, рекомендацию или замечание общего характера.



Этот значок обозначает предупреждение или предостережение.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, не требуется разрешение, чтобы включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске нужно получить разрешение. Можно без ограничений цитировать книгу и примеры в ответах на вопросы. Но чтобы включить значительные объемы кода в документацию по собственному продукту, нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «*JRuby Cookbook*, by Justin Edelson and Henry Liu. Copyright 2009 Justin Edelson and Henry Liu, 978-0-596-51980-3».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу permissions@oreilly.com.

Доступность на Safari

Если на обложке вашей любимой книги присутствует значок Safari® Enabled, это означает, что книга доступна он-лайн в сетевой библиотеке Safari издательства O'Reilly. У Safari есть преимущество перед обычными электронными книгами. Это виртуальная библиотека, которая позволяет легко находить тысячи технических книг, копировать примеры программ, загружать отдельные главы и быстро получать точную и актуальную информацию. Бесплатный доступ по адресу <http://safari.oreilly.com>.



Вопросы и замечания

Мы тщательно проверили информацию, приведенную в этой книге, и протестировали код. Однако полностью исключить ошибки и опечатки невозможно. Если вы

обнаружите какую-нибудь ошибку или захотите высказать пожелание для будущих изданий, пожалуйста, обращайтесь по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Замечания и вопросы технического характера следует отправлять по адресу:
bookquestions@oreilly.com

Для этой книги есть веб-страница, на которой выкладываются списки замеченных ошибок, примеры и планы для будущих изданий. Адрес страницы:

<http://www.oreilly.com/catalog/9780596519803>

Дополнительную информацию о наших книгах и прочих предприятиях можно найти на сайте:

<http://www.oreilly.com>

Благодарности

Мы благодарны коллективу издательства O'Reilly, особенно нашему редактору Майку Лукидесу (Mike Loukides) и литературному редактору Коллин Горман (Colleen Gorman). Спасибо также Стивену Шинглеру (Steven Shingler) за его вклад в главу 4. И всем рецензентам этой книги: Хуану Пабло Таркино (Juan Pablo Tarquino), Джону Пэрселлу (John Purcell) и Дэвиду Кунтцу (David Koontz).

Эта книга никогда не появилась бы на свет без неустанных трудов всей команды, работающей над проектом JRuby, а именно: Чарльза Наттера (Charles Nutter), Томаса Энебо (Nick Sieger) и Ола Бини (Ola Bini). Благодарим также корпорацию Sun и компанию ThoughtWorks за постоянную поддержку JRuby. Проект JRuby размещен на сайте The Codehaus; мы благодарны его создателю Бобу Маквиртеру (Bob McWhirter).

Мы также признательны Нику Рокуэллу (Nick Rockwell), который не перестает подбадривать и воодушевлять нас.

Justin Edelson

Эта книга не состоялась бы, если бы не любовь и поддержка моей чудесной жены Элизабет. Отдельное спасибо моим сыновьям: Оуэну, который сам напечатал свое имя, и Бенджамину, который пока еще этого не умеет.

Благодарю своих коллег в компании MTV Networks: Майкла Бенуа (Michael Benoit), Кита Гриффина (Keith Griffin), Рамеша Нуталапати (Ramesh Nuthalapati), Илью Резникова (Ilya Reznikov), Криса Синдела (Chris Sindel), Джеффа Йемина (Jeff Yemin) и Чжун Чжоу (Jun Zhou) за достойную работу.

Спасибо также Уоррену Хабибу (Warren Habib) за поддержку.

Генри Лю

Спасибо моему другу Йону Баэру (Jon Baer), который впервые привел меня на мероприятие, посвященное языку Ruby, и на протяжении многих лет составляет мне компанию в работе. Я благодарен Фрэнсису Хуангу (Francis Hwang), Мэтту Пеллетье (Matt Pelletier), Себастьяну Дельмонту (Sebastian Delmont), Троттеру Кэшину (Trotter Cashion) и всем остальным участникам группы пользователей Ruby в Нью-Йорке. Они учили меня Ruby и Rails, отвечали на мои наивные вопросы. Именно их увлеченность этой технологией и побудила меня копать глубже. Благодарю всех своих коллег по компании MTV Networks и в особенности Марка Эйка (Mark Ache), Люка Мэрфи (Luke Murphy) и Стива Азуэту (Steve Aзуета) за непрерывную поддержку. А больше всех благодарю свою семью и подругу Наоми; без нее ничего с этой книгой не получилось бы.

Приступая к работе

1.0. Введение	14
1.1. Установка JRuby	17
1.2. Управление пакетами с помощью RubyGems	21
1.3. Одновременное использование Ruby и JRuby	22
1.4. Совместное использование gem-пакетов	24
1.5. Обращение к Java-классам из Ruby	25
1.6. Преобразование Ruby-массива в Java-массив	29
1.7. Включение JAR-файлов в путь поиска классов	30
1.8. Расширение Java-класса в Ruby	31
1.9. Реализация Java-интерфейса на Ruby	32
1.10. Открытие Java-классов в JRuby	37
1.11. Конфигурирование Eclipse для разработки на языке JRuby ...	39
1.12. Конфигурирование NetBeans для разработки на JRuby	43
1.13. Распознавание платформы в приложении JRuby	47

1.0. Введение

JRuby – это реализация языка программирования Ruby для виртуальной машины Java (JVM), распространяемая с открытым исходным кодом. Она позволяет запускать Ruby-приложения на виртуальной машине и обращаться к библиотекам, написанным как на Java, так и на Ruby. Работа над проектом JRuby началась еще в 2001 году, но за последние несколько лет интерес к нему значительно возрос, что отражает общий рост интереса к Ruby, обусловленный успехом каркаса Ruby on Rails. Корпорация Sun немало содействовала популяризации JRuby в частности тем, что приняла в свой штат основных разработчиков и включила поддержку JRuby в среду разработки NetBeans. В настоящее время официальный сайт проекта JRuby расположен по адресу <http://www.jruby.org>.

Ruby

Ruby – это динамический объектно-ориентированный язык, автором которого является Юкиhiro Мацумото (Yukihiro Matsumoto), известный под псевдонимом Мац (Matz). Язык был создан в середине 1990-х годов. При нумерации версий Ruby применяется то же соглашение, что для ядра Linux: четными номерами обозначаются стабильные версии, а нечетными – находящиеся в разработке. Поэтому существуют две *текущие* версии Ruby: 1.8.6, выпущенная в марте 2007 года (текущая стабильная версия), и 1.9.0, выпущенная в декабре 2007 года (текущая разрабатываемая версия). Стандартный интерпретатор Ruby* написан на языке C. Есть также несколько альтернативных реализаций, в том числе JRuby, IronRuby (для Microsoft .NET Framework) и Rubinius. Для Ruby не существует формальной спецификации языка, однако работа в этом направлении ведется на вики-сайте <http://spec.ruby-doc.org>.

Поскольку Ruby является объектно-ориентированным языком, многие базовые идеи знакомы разработчикам на Java, хотя синтаксис, конечно, отличается. Самая существенная особенность Ruby – поддержка *блоков*. В Ruby блоком называется группа предложений, которая передается вызываемому методу. Метод-получатель может вызывать блок несколько раз, передавая ему параметры. Обсуждается включение похожей конструкции, *замыкания*, в версию Java 7. На данный момент представлено несколько конкурирующих предложений; какое из них будет выбрано и будет ли выбрано вообще, пока неясно. В листинге 1.1 на примере простого класса Ruby демонстрируются два способа определения блока. Первый вариант – с фигурными скобками – обычно применяется для создания блока, содержащего одно предложение; второй – с ключевыми словами `do` и `end` – для блоков из нескольких предложений.

Пример 1.1. Введение в блоки Ruby

```
class HelloWorldSayer
  def hello_world
    yield "Hello"
```

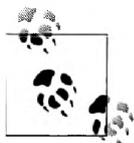
* Обычно называется Matz's Ruby Interpreter (MRI).

```
        yield "World"
        yield "from Ruby"
    end
end

sayer = HelloWorldSayer.new
sayer.hello_world { |message| puts message.swapcase }

# или

sayer.hello_world do |it|
  puts it.swapcase
end
```



Функция Ruby `yield` передает управление методу-аргументу блока.

Мы вовсе не хотим сказать, что блоки – единственное заметное различие между Ruby и Java, но оно, безусловно, является одним из самых существенных, поскольку блоки встречаются в программах на Ruby повсеместно. Так, в примере 1.2 приведена Java-программа для распечатки списка целых чисел от 1 до 10. Соответствующий код на Ruby показан в примере 1.3.

Пример 1.2. Цикл в Java

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Пример 1.3. Цикл в Ruby

```
1.upto(10) { |x| puts x }
```

Вокруг языка Ruby образовалось активное сообщество разработчиков – как в Сети, так и в виде местных групп. На сайте <http://www.ruby-lang.org> имеется дополнительная информация об этих группах пользователей. Языку Ruby посвящено немало книг, из которых наиболее известны знаменитая «Киркомотыга» (по картинке на обложке) – *Programming Ruby: The Pragmatic Programmers' Guide* (Pragmatic Bookshelf) by Dave Thomas, Chad Fowler, and Andy Hunt – и *The Ruby Programming Language* by David Flanagan and Yukihiro Matsumoto (O'Reilly).

JRuby

Первая версия JRuby была написана Яном Арне Петерсенем (Jan Arne Petersen) в 2001 году и явилась прямым переносом интерпретатора Ruby 1.6, реализованного на C. В течение нескольких последующих лет проект оставался интересным, но имел серьезные ограничения в плане производительности. Вслед за выпуском

версии Ruby 1.8 в 2003 году, а затем и каркаса Ruby on Rails для веб-приложений в 2004 разработке JRuby уделялось очень много внимания, особенно в части совместимости и производительности. В сентябре 2006 года корпорация Sun Microsystems по сути дела взяла на себя финансирование проекта, приняв в штат двух ведущих разработчиков, Чарльза Наттера и Томаса Энебо. Позже взяли на работу также третьего разработчика, Ника Сигера*.

Для корпорации Sun язык JRuby – это возможность распространить виртуальную машину Java на новые сферы. Первоначально JVM была очень тесно связана с языком Java, но появление таких проектов, как JRuby, Jython (реализация языка Python на платформе Java), Groovy (язык сценариев, берущий истоки в Ruby) и Scala (одновременно функциональный и объектно-ориентированный язык) доказало, что JVM может стать платформой для самых разных языков. Эта тенденция увенчалась созданием спецификации Java Specification Request (JSR) 223, описывающей технологию сценариев на платформе Java. В документе JSR 223 определен стандартный API (интерфейс прикладного программирования), позволяющий интегрировать сценарные языки с JVM. На сайте <https://scripting.dev.java.net> имеются реализации JSR 223 API для 25 языков программирования. Мы еще вернемся к обсуждению этого API в главе 3.

Для пользователей JRuby открывает возможность воспользоваться всей мощностью динамического языка, каковым является Ruby, не отказываясь от существующих библиотек и серверов приложений, написанных на Java. Этот аспект мы будем рассматривать в двух первых главах.

С выходом версии JRuby 1.1 в апреле 2008 года проблема производительности JRuby была решена за счет нового написанного на C интерпретатора Ruby; во многих случаях язык стал работать быстрее. С точки зрения совместимости разработчики JRuby стремятся дублировать поведение стандартного интерпретатора Ruby, даже если для этого приходится поступаться согласованностью с принципами Java. Включена большая часть базовых классов Ruby, а также многие классы из стандартной библиотеки Ruby, система управления пакетами RubyGems, система документирования RDoc и система сборки Rake. Несмотря на прилагаемые усилия, в некоторых областях поведение JRuby все же отличается от стандартного интерпретатора Ruby. Самый наглядный пример – работа с потоками в JRuby. В этом отношении JRuby на самом деле ушел вперед от стандартного интерпретатора. Поточковая модель, которую JRuby поддерживает уже сейчас, предположительно будет включена только в версию Ruby 2.0.

В этой главе мы рассмотрим процедуру установки JRuby, приведем кое-какую информацию об интеграции Java и Ruby и обсудим разнообразные настройки IDE, относящиеся к интеграции.

* Четвертый из основных разработчиков, Ола Бини, трудится во влиятельной консалтинговой ИТ-компании ThoughtWorks.

1.1. Установка JRuby

Задача

Требуется установить JRuby.

Решение

Скачайте последнюю двоичную версию с сайта JRuby <http://www.jruby.org> и распакуйте архив. Добавьте каталог *bin* в переменную окружения `PATH`.

Обсуждение

Windows

На сайте JRuby имеются двоичные дистрибутивы в форматах ZIP и TGZ. Начиная с Windows XP, средства для работы с ZIP-архивами уже встроены в операционные системы Windows. Существуют также открытые и коммерческие программы, поддерживающие формат TGZ, например: WinZip (<http://www.winzip.com>), 7-Zip (<http://www.7-zip.org>) и IZArc (<http://www.izarc.com>).

JRuby может размещаться в любом месте. Лично я предпочитаю устанавливать библиотеки и исполняемые файлы Java в подкаталоги каталога `C:\java`. На рис. 1.1 показан результат распаковки двоичного дистрибутива последней версии (во время работы над книгой текущей была версия 1.1).

Сразу после распаковки можно начинать работать с JRuby. Полюбоваться на JRuby в действии проще всего, запустив программу `jirb`, вариант Interactive Ruby (`irb`) для JRuby. Как и `irb`, `jirb` позволяет вводить предложения языка Ruby и

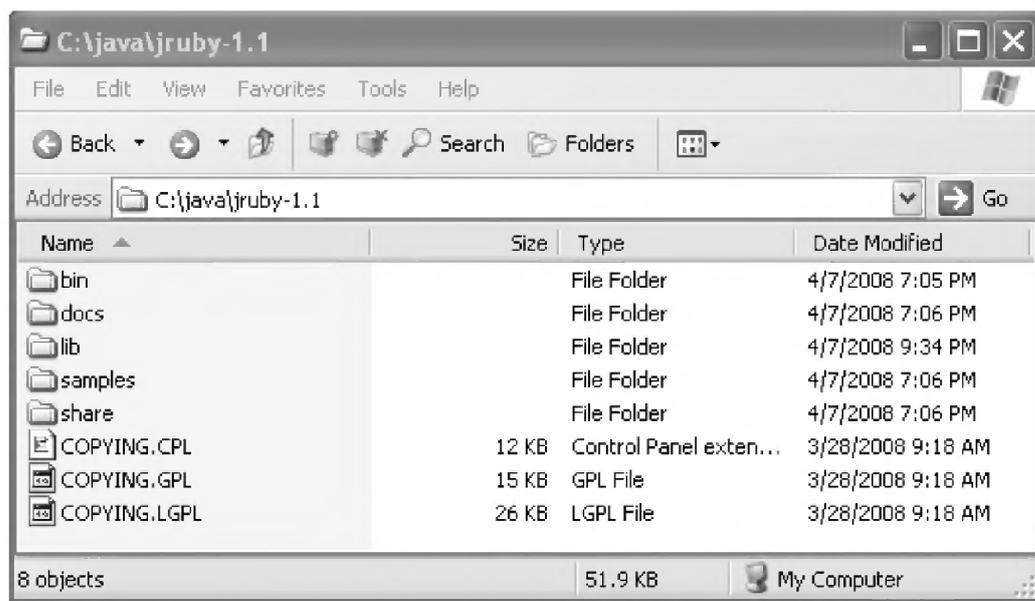


Рис. 1.1. Распакованный двоичный дистрибутив JRuby

сразу же видеть результат их выполнения. В каталоге *bin* имеются командная и графическая версии *jirb*. Для запуска командной версии (рис. 1.2) следует выполнить файл *bin\jirb.bat*, а для запуска графической (рис. 1.3) – файл *bin\jirb_swing.bat*. На обоих рисунках показаны результаты выполнения тривиального Ruby-кода. Вы видите как строку, которую печатает метод `puts` (`Hello World`), так и возвращаемый этим методом результат (`nil`).



```
C:\WINDOWS\system32\cmd.exe
irb(main):001:0> puts "Hello World"
Hello World
=> nil
irb(main):002:0>
```

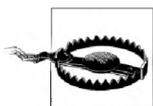
Рис. 1.2. Командная версия *jirb*



```
JRuby IRB Console (tab will autocomplete)
Welcome to the JRuby IRB Console [1.1RC1]

irb(main):001:0> puts "Hello World"
Hello World
=> nil
irb(main):002:0> |
```

Рис. 1.3. Графическая версия *jirb*



Если при запуске *jirb.bat* или *jirb_swing.bat* из Windows Explorer вы видите, как появляется и тут же пропадает черное окошко, то, вероятно, не установлена или неправильно установлена переменная среды `JAVA_HOME`. Для установки переменных среды в Windows откройте миниприложение **Система** на панели управления и перейдите на вкладку **Дополнительно**. Переменная `JAVA_HOME` должна указывать на каталог, в который установлена Java.

Можно также протестировать JRuby из командной строки, задав параметр `-e` (evaluate):

```
C:\java\jruby-1.1\bin>jruby -e "puts 'Hello World'"
```

Чтобы не вводить каждый раз полный путь к каталогу *bin* JRuby, добавьте его в переменную среды `PATH`. Для этого откройте миниприложение **Система** на панели управления и перейдите на вкладку **Дополнительно**. Нажмите кнопку **Переменные среды**. Откроется диалоговое окно «Переменные среды», показанное на рис. 1.4. С помощью кнопок **Создать** и **Изменить** в секции **Системные переменные** добавьте переменную среды `JRUBY_HOME` и допишите в начало переменной `PATH` путь `%JRUBY_HOME%\bin`. Можно вместо этого просто дописать в начало `PATH` полный путь к каталогу *bin*, но при наличии отдельной переменной среды будет проще переходить на новую версию.

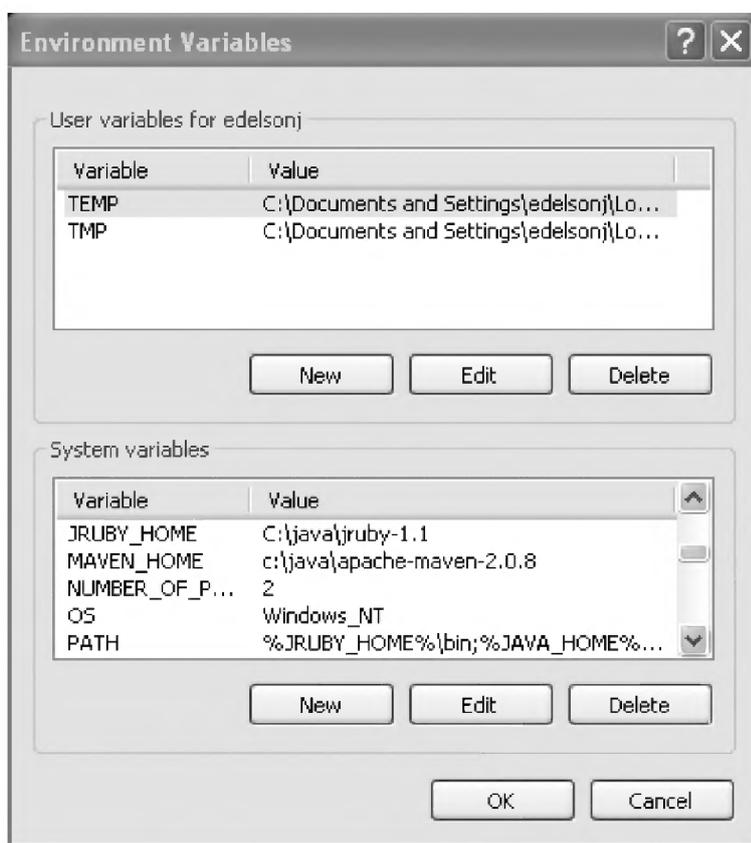


Рис. 1.4. Диалоговое окно «Переменные среды»

Linux и Mac OS X

На сайте JRuby имеются двоичные дистрибутивы в форматах ZIP и TGZ. Хотя в большинстве дистрибутивов Linux и в OS X есть утилиты для работы с архивами обоих типов, предпочтительно все же использовать TGZ-файл, так как в нем сохраняется информация о правах доступа (в ZIP-архиве она отсутствует).



На сайте JPackage Project по адресу <http://www.jpackage.org> имеется дистрибутив в формате RPM. Во время работы над этой книгой последней версии JRuby там еще не было, но, когда вы будете ее читать, ситуация может измениться.

Если у вас есть привилегии суперпользователя `root` в той системе, где устанавливается JRuby, то установку следует производить в общепринятые каталоги. В зависимости от конкретной ОС это может быть каталог `/usr/local/jruby`, `/usr/share/jruby`, `/opt/JRuby` или еще какой-нибудь. В системе Mac OS X пользователям рекомендуется производить установку пакетов в каталоги `/opt/local/jruby` или `/usr/local/jruby`. Если привилегий `root` у вас нет, то устанавливать, скорее всего, придется в подкаталог собственного начального каталога, например `~/jruby`. По умолчанию версии JRuby распаковываются в каталог, имя которого включает номер версии, поэтому мы создадим символическую ссылку с `~/jruby` на `~/jruby-1.1`. Это упростит переход на следующую версию в будущем:

```
$ cd ~
$ tar -xzf jruby-bin-1.1.tar.gz
$ ln -s jruby-1.1 jruby
```

Создайте переменную среды `JRUBY_HOME`, указывающую на инсталляционный каталог, и добавьте путь к каталогу `bin` JRuby в переменную среды `PATH`; вставьте в файл `~/.profile` строки, показанные в примере 1.4.

Пример 1.4. Добавление пути к JRuby в переменную среды PATH с помощью файла .profile

```
export JRUBY_HOME=~/.jruby
export PATH=$JRUBY_HOME/bin:$PATH
```

Добавив путь в каталогу `bin` в переменную среды `PATH`, вы можете проверить правильность установки, попытавшись выполнить такой Ruby-сценарий:

```
$ jruby -e "puts 'Hello World'"
Hello World
```



Чтобы воспользоваться какой-либо командной утилитой, входящей в дистрибутив JRuby, в частности `jirb`, необходимо предварительно добавить путь к каталогу `bin` в переменную среды `PATH`.

См. также

- Рецепт 8.1 «Сборка JRuby из исходных кодов»
- Рецепт 1.3 «Одновременное использование Ruby и JRuby»

1.2. Управление пакетами с помощью RubyGems

Задача

Требуется установить Ruby on Rails или другие Ruby-пакеты для использования совместно с JRuby.

Решение

Воспользуйтесь встроенной в JRuby поддержкой системы RubyGems. Начать работу с ней можно сразу же после установки JRuby, достаточно запустить сценарий *gem*, находящийся в каталоге *bin* JRuby. Для установки пакета выполните следующую команду:

```
$ gem install packagename
```

Например, чтобы установить каркас Ruby on Rails, выполните команду:

```
$ gem install rails
```

Обсуждение

RubyGems – это стандартная система управления пакетами и подготовки дистрибутивов, ориентированная на пакеты Ruby. В репозитории RubyGems по адресу <http://gems.rubyforge.org> есть тысячи таких пакетов (они называются *gem-пакетами*), большинство совместимы с любой реализацией Ruby.

К числу наиболее употребительных команд RubyGems относятся *install*, *query*, *update*, *uninstall* и *rdoc*. Полный список можно получить с помощью команды *help**:

```
$ gem help commands
```

Список команд GEM:

<i>build</i>	Создать gem-пакет по gem-спецификации
<i>cert</i>	Управление параметрами сертификатов и электронной подписью RubyGems
<i>check</i>	Проверить установленные gem-пакеты
<i>cleanup</i>	Удалить старые версии gem-пакетов из локального репозитория
<i>contents</i>	Вывести содержимое установленных gem-пакетов
<i>dependency</i>	Показать зависимости установленного gem-пакета

* Для удобства читателя справка переведена на русский язык. *Прим. перев.*

environment	Вывести информацию о среде исполнения RubyGems
fetch	Скачать gem-пакет и поместить его в текущий каталог
generate_index	Сгенерировать индексы для каталога gem-сервера
help	Вывести справку по команде 'gem'
install	Установить gem-пакет в локальный репозиторий
list	Вывести перечень всех gem-пакетов, имена которых начинаются со строки STRING
lock	Сгенерировать список фиксированных версий gem-пакетов
mirror	Организовать зеркало репозитория gem-пакетов
outdated	Вывести перечень gem-пакетов, нуждающихся в обновлении
pristine	Восстановить первоначальное состояние установленных gem-пакетов из файлов, находящихся в gem-кэше
query	Запросить информацию о gem-пакете, находящемся в локальном или удаленном репозитории
rdoc	Сгенерировать RDoc-документацию для предустановленных gem-пакетов
search	Вывести перечень всех gem-пакетов, имена которых содержат строку STRING
server	HTTP-сервер, на котором находится документация и репозиторий gem-пакетов
sources	Управление источниками и кэшем, в которых RubyGems ищет gem-пакеты
specification	Вывести спецификацию gem-пакета (в формате yaml)
uninstall	Удалить gem-пакеты из локального репозитория
unpack	Распаковать установленный gem-пакет в текущий каталог
update	Обновить указанные (или все установленные) gem-пакеты в локальном репозитории
which	Найти, где находится библиотека

Для получения справки по конкретной команде введите 'gem help COMMAND'.

Можно набирать только начальные буквы названий команд при условии, что не возникает неоднозначности.

Например, 'gem i rake' – то же самое, что 'gem install rake'.

См. также

- Руководства по системе RubyGems, <http://rubygems.org>
- Рецепт 1.3 «Одновременное использование Ruby и JRuby»

1.3. Одновременное использование Ruby и JRuby

Задача

На одном и том же компьютере установлены Ruby и JRuby. Требуется, чтобы Ruby-сценарий обрабатывался нужным интерпретатором.

Решение

Воспользуйтесь флагом `-S` при запуске исполняемых файлов *ruby* и *jruby*. Так, обычно RubyGems запускается такой командой:

```
gem install rails
```

а вы вместо этого наберите:

```
$ jruby -S gem install rails
```

или

```
$ ruby -S gem install rails
```

Обсуждение

В состав таких популярных Ruby-пакетов, как Rake, Ruby on Rails и RubyGems, входят собственные исполняемые Ruby-сценарии. В большинстве руководств рекомендуется запускать их напрямую. Будет ли сценарий исполняться интерпретатором Ruby или JRuby, зависит от значения переменной среды `RATH`, от используемой платформы и от конкретного пакета. Поскольку факторов так много, мы предлагаем общий, всегда работающий метод – точно указывать имя программы (*ruby* или *jruby*), а имя сценария передавать во флаге `-S`.

Флаг `-S` сообщает Ruby или JRuby о том, что нужно загрузить файл сценария, находящийся в одном из каталогов, перечисленных в переменной `RATH`. В состав JRuby входят отдельные экземпляры систем Rake и RubyGems, расположенные в каталогах *bin/rake* и *bin/gem*. Но, поскольку они являются точными копиями оригинальных файлов, не имеет значения, какой вариант сценария выбран. *Важно лишь, каким интерпретатором он исполняется.*

Эта рекомендация особенно важна применительно к сценарию *gem*, который реализует систему RubyGems. Для создания нового приложения Rails можно запустить любую из двух команд:

```
$ ruby -S rails sampleapp
```

или:

```
$ jruby -S rails sampleapp
```

Результат будет один и тот же. Но при запуске команд

```
$ ruby -S gem install rails
```

и

```
$ jruby -S gem install rails
```

gem-пакет Rails будет установлен в разные места. Убедиться в этом можно, выполнив команду `environment`:

```
$ ruby -S gem environment
```

```
RubyGems Environment:
```

```
- RUBYGEMS VERSION: 1.0.1 (1.0.1)
- RUBY VERSION: 1.8.5 (2007-09-24 patchlevel 114) [i386-linux]
- INSTALLATION DIRECTORY: /usr/lib/ruby/gems/1.8
- RUBY EXECUTABLE: /usr/bin/ruby
- RUBYGEMS PLATFORMS:
  - ruby
  - x86-linux
- GEM PATHS:
  - /usr/lib/ruby/gems/1.8
- GEM CONFIGURATION:
  - :update_sources => true
  - :verbose => true
  - :benchmark => false
  - :backtrace => false
  - :bulk_threshold => 1000
- REMOTE SOURCES:
  - http://gems.rubyforge.org
```

```
$ jruby -S gem environment
```

```
RubyGems Environment:
```

```
- RUBYGEMS VERSION: 1.0.1 (1.0.1)
- RUBY VERSION: 1.8.6 (2008-01-07 patchlevel 5512) [java]
- INSTALLATION DIRECTORY: /home/justin/jruby-1.1/lib/ruby/gems/1.8
- RUBY EXECUTABLE: /home/justin/jruby-1.1/bin/jruby
- RUBYGEMS PLATFORMS:
  - ruby
  - universal-java-1.6
- GEM PATHS:
  - /home/justin/jruby-1.1/lib/ruby/gems/1.8
- GEM CONFIGURATION:
  - :update_sources => true
  - :verbose => true
  - :benchmark => false
  - :backtrace => false
  - :bulk_threshold => 1000
- REMOTE SOURCES:
  - http://gems.rubyforge.org
```

См. также

- Рецепт 1.2 «Управление пакетами с помощью RubyGems»

1.4. Совместное использование gem-пакетов

Задача

Вы уже установили ряд gem-пакетов и хотите использовать их из JRuby, не устанавливая еще раз.

Решение

Настройте переменную среды `GEM_HOME`, так чтобы она указывала на каталог, где находятся уже установленные gem-пакеты. Нужное значение выводит команда `gem environment` в строке `INSTALLATION DIRECTORY`:

```
$ ruby -S gem environment | grep -i 'installation directory'
- INSTALLATION DIRECTORY: /usr/lib/ruby/gems/1.8
$ export GEM_HOME=/usr/lib/ruby/gems/1.8
$ jruby -S gem environment | grep -i 'installation directory'
INSTALLATION DIRECTORY: /usr/lib/ruby/gems/1.8
```

Обсуждение

Некоторые gem-пакеты реализованы целиком на Ruby, но многие написаны на смеси Ruby и C (а теперь все чаще на смеси Ruby и Java). Gem-пакеты на чистом Ruby можно устанавливать с помощью как JRuby, так и стандартного Ruby. Те же, что написаны на разных языках, должны устанавливаться совместимым интерпретатором. Перечень поддерживаемых платформ дает команда `gem environment`. Поскольку среда исполнения RubyGems знает, какие платформы поддерживаются, разрешается хранить пакеты, предназначенные для разных платформ, в одном и том же каталоге; среда сама выберет подходящие библиотеки.

1.5. Обращение к Java-классам из Ruby

Задача

Требуется написать на Ruby программу, в которой используется один или несколько Java-классов.

Решение

Прежде всего, необходимо сообщить JRuby о том, что вы собираетесь обращаться к Java-классам из кода на Ruby. Для этого поместите в начало Ruby-файла объявление `include`:

```
include Java
```

Синтаксис обращения к конкретному Java-классу зависит от того, в каком пакете этот класс находится. Если имя пакета начинается с `java`, `javax`, `org` или `com`, то можно просто сослаться на полное имя класса или воспользоваться предложением `import`, как показано в примере 1.5.

Пример 1.5. Создание объекта класса Java TreeMap из Ruby

```
# с использованием полного имени
```

```
map = java.util.TreeMap.new
```

```
# с использованием предложения import
```

```
import java.util.TreeMap
map = TreeMap.new
```

Если же имя пакета, содержащего класс, не начинается ни с одной из строк `java`, `javax`, `org`, `com` или класс находится в пакете, подразумеваемом по умолчанию, то придется воспользоваться функцией `include_class`, как показано в примере 1.6.

Пример 1.6. Ссылка на Java-класс с помощью функции `include_class`

```
include_class 'EDU.oswego.cs.dl.util.concurrent.ConcurrentHashMap'
map = ConcurrentHashMap.new
```



Если вы предпочитаете единообразие, то имейте в виду, что функция `include_class` применима и к классам, которые находятся в пакетах с именами, начинающимися со строк `java`, `javax`, `org` или `com`.

Функция `include_class` позволяет также создавать псевдонимы в тех случаях, когда имя Java-класса конфликтует с именем Ruby-класса. Для этого следует передать функции блок. В примере 1.7 для Java-класса `String` создается псевдоним `JString`, чтобы избежать конфликта с Ruby-классом `String`.

Пример 1.7. Создание псевдонима для разрешения конфликта имен

```
include Java

include_class 'java.lang.String' do |package,name|
  "JString"
end

p JString.new("A quick brown fox").indexOf("brown")
```

Функции `include_class` можно передать и список имен классов. В таком случае для задания псевдонимов применяется предложение `case`, как показано в примере 1.8.

Пример 1.8. Задание псевдонимов для нескольких классов

```
include_class ['java.lang.String','java.lang.Integer'] do |package,name|
  case name
  when "String"
    "JString"
  when "Integer"
    "JInteger"
  end
end
```

Можно вместо задания псевдонимов описанным выше способом обернуть Java-пакет в Ruby-модуль, воспользовавшись функцией `include_package` (пример 1.9).

Пример 1.9. Обертывание Java-пакета в Ruby-модуль

```
include Java

module JavaLang
  include_package 'java.lang'
end

p JavaLang::String.new("A quick brown fox").indexOf("brown")
```

Обсуждение

В JRuby ссылки на Java-классы с точки зрения разработчика на Java выглядят вполне естественно. Для большинства популярных пакетов достаточно просто воспользоваться директивой `import` – точно так же, как в обычной Java-программе.

При вызове методов Java-класса JRuby автоматически выполняет некоторые преобразования типов. Так, экземпляры ряда основных классов Ruby: `FixNum`, `Float` и `String` – преобразуются в соответствующие типы Java в момент передачи в качестве параметров Java-методам. В JRuby включены реализации интерфейсов `java.util.List` и `java.util.Map` для объектов Ruby-классов `Array` и `Hash`. Объекты Ruby-класса `Array` можно привести к Java-классу `Array`, вызвав метод `to_java`. В примере 1.10 приведен код на смеси Java и Ruby, демонстрирующий описанную выше функциональность.

Пример 1.10. Преобразование типов Ruby в типы Java

```
package org.jrubycookbook.ch01;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import org.jruby.Ruby;
import org.jruby.javasupport.JavaEmbedUtils;

public class PrintJavaClass {

    // Вывести имя класса и список интерфейсов для одного объекта
    public String output(Object o) {
        String className = o.getClass().getName();
        List<Class> interfaces = Arrays.asList(o.getClass().getInterfaces());
        return String.format("%s, implements %s\n", className, interfaces);
    }

    // Вывести имя класса и список интерфейсов для каждого объекта в массиве
    public String output(Object[] objects) {
```

```

    PrintWriter writer = new PrintWriter(new StringWriter());
    for (Object o : objects) {
        String className = o.getClass().getName();
        List<Class> interfaces = Arrays
            .asList(o.getClass().getInterfaces());
        writer.printf("%s (в массиве), implements %s\n", className,
            interfaces);
    }
    return writer.toString();
}

public static void main(String[] args) {
    Ruby runtime = JavaEmbedUtils.initialize(Collections.EMPTY_LIST);
    String script = "@printer = org.jrubycookbook.ch01.PrintJavaClass.new\n"
        + "def output(o)\n"
        + "puts \"#{o.to_s} - #{@printer.output(o)}\"\n"
        + "end\n"
        + "output(1)\n"
        + "output(0.5)\n"
        + "output('string')\n"
        + "output(true)\n"
        + "output([4, 8, 15, 16, 23, 42])\n"
        + "output([4, 8, 15, 16, 23, 42].to_java)\n"
        + "output({ 'NY' => 'New York', 'MA' => 'Massachusetts'})\n";

    runtime.evalScriptlet(script);
    JavaEmbedUtils.terminate(runtime);
}
}

```



О классе `JavaEmbedUtils`, который встречается в примере 1.10, см. рецепт 3.1.

Во время выполнения этот класс выводит следующую информацию:

```

1 - Class is java.lang.Long, implements [interface java.lang.Comparable]
0.5 - Class is java.lang.Double, implements [interface java.lang.Comparable]
string - Class is java.lang.String, implements [interface java.io.Serializable,\
interface java.lang.Comparable, interface java.lang.CharSequence]
true - Class is java.lang.Boolean, implements [interface java.io.Serializable,\
interface java.lang.Comparable]
4815162342 - Class is org.jruby.RubyArray, implements [interface java.util.List]
[Ljava.lang.Object;@8b058b - Received an array
In array: class is java.lang.Integer, implements [interface java.lang.Comparable]

```

```
In array: class is java.lang.Integer, implements [interface java.lang.Comparable]
NYNew YorkMAMassachusetts - Class is org.jruby.RubyHash, implements\
 [interface java.util.Map]
```

JRuby предоставляет оператор `::` для доступа к открытым статическим методам и переменным. В примере 1.11 показано, как получить доступ к статическим методам и переменным Java-класса `Math`.

Пример 1.11. Доступ к статическим методам и переменным

```
require 'java'

puts java.lang.Math::max(100,200)
puts java.lang.Math::PI
```

1.6. Преобразование Ruby-массива в Java-массив

Задача

Требуется передать Ruby-массив методу, который принимает Java-массив конкретного типа.

Решение

Вызовите метод `to_java` Ruby-массива, передав в качестве аргумента тип конечного Java-массива. Вот, например, как можно создать массив объектов типа `javax.xml.transform.stream.StreamSource`:

```
import javax.xml.transform.stream.StreamSource

cnn = StreamSource.new "http://rss.cnn.com/rss/cnn_topstories.rss"
mtv = StreamSource.new "http://www.mtv.com/rss/news/news_full.jhtml"

# Вызываем Java API трансформации. В объявлении этого метода должна быть
# указана следующая сигнатура:
# public String transform(StreamSource[] sources)
p transformer.transform([cnn,mtv].to_java(StreamSource))
```

Для примитивных типов Java, а также для типа `java.lang.String` определены символы Ruby. Например, так можно создать массив объектов примитивного типа `int`:

```
[1,2,3,4,5,6,7,8,9,10].to_java(:int)
```

Обсуждение

Эта особенность JRuby исключительно важна для доступа к различным Java API. Например, при вызове метода с помощью подсистемы Java Management Extensions (JMX) методу `invoke()` класса `javax.management.MBeanServer` нужно пере-

дать два массива. Один состоит из объектов типа `Object` и применяется для хранения параметров метода, другой – из объектов типа `String`, в которых хранится сигнатура метода. Чтобы обратиться к методу `invoke()` из Ruby, следует написать примерно такой код:

```
brokerName = ObjectName.new('org.apache.activemq:BrokerName=localhost,Type=Broker')
params = ["MyQueue"].to_java()
signature = ["java.lang.String"].to_java(:string)
server.invoke(brokerName, 'addQueue', params, signature)
```

1.7. Включение JAR-файлов в путь поиска классов

Задача

Требуется сослаться на Java-класс, который находится в JAR-файле, не включенном в путь поиска классов (`classpath`).

Решение

Вызовите метод `Ruby require`, передав ему путь к JAR-файлу. Этот путь может быть задан относительно текущего рабочего каталога:

```
require 'lib/commons-logging-1.1.jar'
```

или абсолютно:

```
require '/opt/java/commons-logging/bin/commons-logging-1.1.jar'
```

При работе в Windows компоненты пути могут разделяться как прямой, так и обратной косой чертой:

```
require 'c:\java\commons-logging-1.1\bin\commons-logging-1.1.jar'
# или
require 'c:/java/commons-logging-1.1/bin/commons-logging-1.1.jar'
```

Обсуждение

Хотя эта возможность JRuby весьма полезна, прибегать к ней следует с осторожностью, особенно если вы указываете абсолютные пути, зависящие от платформы или способа установки. Относительные пути, на первый взгляд, выглядят лучше, но в действительности налагают существенные ограничения, так как вычисляются относительно текущего рабочего каталога, а не относительно каталога, в котором находится сценарий. Впрочем, не все потеряно.

У этого механизма JRuby есть любопытная особенность: JAR-файл добавляется в путь поиска классов динамически, во время работы приложения. Это позволяет воспользоваться имеющейся в Ruby функцией интерполяции строк для по-

строения абсолютных путей. В примере 1.12 показан метод, который строит путь к JAR-файлу, находящемуся в локальной репозитории Maven*.

Пример 1.12. Динамическое построение пути к JAR-файлу

```
# Задать переменную среды HOME, если установлена переменная USERPROFILE
ENV['HOME'] = ENV['USERPROFILE'] if (ENV['USERPROFILE'])

def require_from_maven(group, artifact, version)
  maven_path = «#{group}/#{artifact}/#{version}/#{artifact}-#{version}.jar»
  require «#{ENV['HOME']}/.m2/repository/#{maven_path}»
end
```

Прикладная программа может включить этот сценарий с помощью предложения `require`, а затем вызвать метод `require_from_maven` для ссылки на конкретный JAR-файл:

```
require 'require_from_maven'
require_from_maven "commons-logging", "commons-logging", "1.1"
```

1.8. Расширение Java-класса в Ruby

Задача

Чтобы воспользоваться каким-то Java API, вам требуется создать Ruby-класс, являющийся подклассом Java-класса.

Решение

Воспользуйтесь стандартным для Ruby оператором наследования `<` и укажите Java-класс, которому хотите унаследовать. В примере 1.13 показан Ruby-класс, который расширяет Java-класс `Thread` и переопределяет его метод `run()`.

Пример 1.13. Наследование Java-класса в Ruby

```
include Java

class MyThread < java.lang.Thread
  def run
    puts 'hello world'
  end
end

MyThread.new.start
```

* Это наивный пример использования репозитория Maven, поскольку предполагается, что JAR-файл уже находится в репозитории. Buildr, написанная на Ruby система сборки для Java, позволяет загружать файлы из удаленных репозиториях Maven. Дополнительную информацию о системе Maven см. в главе 6.

Обсуждение

То, что в JRuby для расширения Java- и Ruby-классов применяется один и тот же синтаксис, – далеко не случайность. Это проектное решение улучшает интеграцию обоих языков.



Классы, в которых используются родовые типы Java 5 (generics), не подпадают под этот рецепт. В настоящее время невозможно унаследовать Ruby-класс от такого класса Java.

Абстрактные Java-классы тоже можно расширять Ruby-классами. В примерах 1.14 и 1.15 показан абстрактный Java-класс и расширяющий его класс Ruby. Метод `hello()`, объявленный в Java-классе абстрактным, реализован в Ruby-классе.

Пример 1.14. Абстрактный Java-класс

```
package org.jrubycookbook.ch01;

public abstract class AbstractElement {
    public abstract void hello();

    public void sayHello(int count) {
        for (int i = 0; i < count; i++) {
            hello();
        }
    }
}
```

Пример 1.15. Ruby-класс, наследующий абстрактному Java-классу

```
include Java

import org.jrubycookbook.ch01.AbstractElement

class RubyElement < AbstractElement
    def hello
        puts 'hello world'
    end
end

RubyElement.new.sayHello 5
```

1.9. Реализация Java-интерфейса на Ruby

Задача

Чтобы воспользоваться каким-то Java API, вам требуется создать Ruby-класс, который реализует Java-интерфейс.

Решение

Создайте свой класс, в котором имена методов совпадают с именами методом в Java-интерфейсе. Начиная с версии 1.1, среда исполнения JRuby поддерживает *динамическую типизацию* (duck typing) для реализации Java-интерфейсов. В Ruby, как и во многих других динамических языках, динамическая типизация означает, что тип объекта определяется методами, которые этот объект реализует. В примере 1.16 показано, как эта техника применяется для создания нового потока Java, конструктору которого передается объект, реализующий интерфейс `java.lang.Runnable`. Класс `HelloThread` содержит метод `run` без аргументов, сигнатура которого соответствует методу, определенному в интерфейсе `java.lang.Runnable`. Для создания объекта класса `Thread` JRuby не требует наличия в классе `HelloThread` никакой дополнительной информации о типе.

Пример 1.16. Реализация Java-интерфейса в Ruby

```
include Java
```

```
class HelloThread
  def run
    puts 'hello world'
  end
end
```

```
java.lang.Thread.new(HelloThread.new).start
```

Обсуждение

Есть несколько случаев, когда динамической типизации оказывается недостаточно и интерпретатору необходимо сообщать дополнительную информацию о типе. Один такой случай – это передача динамически типизированного объекта JRuby в качестве аргумента перегруженному методу Java. Без дополнительной информации о типе интерпретатор JRuby не может решить, какой именно метод вызывать. Решение состоит в том, чтобы с помощью предложения Ruby `include` явно сопоставить с Ruby-классом интерфейс Java. Такой информации интерпретатору JRuby достаточно для выбора правильного метода. В примере 1.17 класс `HelloThread` сопоставляется с интерфейсом `Runnable`, вследствие чего JRuby вызывает требуемый метод `exec()`, и на консоль выводится строка `runnable`.

Пример 1.17. Объявление Java-интерфейсов в JRuby

```
Balloon.java
```

```
public interface Balloon {
  void pop();
}
```

```
Bubble.java
```

```
public interface Bubble {
```

```
    void pop();
}
```

Child.java

```
public class Child{
    public void give(Bubble bubble){
        System.out.println("Thanks for the bubble.");
        bubble.pop();
    }

    public void give(Balloon balloon){
        System.out.println("Thanks for the balloon.");
        balloon.pop();
    }
}
```

main.rb

```
include Java

class MylarBalloon
    include Java::Balloon
    def pop
        puts 'Oh No!!!'
    end
end

child = Java::Child.new
child.give(MylarBalloon.new)
```

Поскольку Ruby-сценарии неявно создают класс верхнего уровня, то для реализации Java-интерфейса даже необязательно определять новый класс. Эта функциональность, как видно из примера 1.18, может быть весьма полезна при создании прототипов и для целей тестирования.

Пример 1.18. Работа с Java-интерфейсами в JRuby – краткая версия

```
include Java

def pop
    puts 'Bang'
end

child = Java::Child.new
child.give(self)
```

Модули Ruby – естественный инструмент реализации Java-интерфейсов. В некоторых отношениях они напоминают абстрактные Java-классы, однако есть и существенное отличие: один класс может включать много модулей. В примере

1.19 показано применение модуля для реализации Java-интерфейса, а также повторное использование этого модуля.

Пример 1.19. Реализация Java-интерфейса с помощью модуля

```
include Java

module RunModule
  def run
    1.upto(10) { |i| puts "You're number #{i}" }
  end
end

class HelloThread
  include RunModule
end

java.lang.Thread.new(HelloThread.new).start
```

JRuby позволяет создавать экземпляр интерфейса с помощью метода `impl`, который динамически присоединяется ко всем Java-интерфейсам. Этот метод принимает в качестве параметра блок, который выполняется при вызове любой функции интерфейса. У самого блока есть два аргумента: имя метода интерфейса, инициировавшего выполнение блока, и параметр переменной длины, через который передаются аргументы метода. В примере 1.20 метод `impl` применяется для задания критерия сортировки в Java-интерфейсе `Comparator`.

Пример 1.20. Использование метода JRuby `impl`

```
include Java

v = java.util.Vector.new
v.add_element("Lions")
v.add_element("Tigers")
v.add_element("Bears")

java.util.Collections::sort(v, java.util.Comparator.impl do |method, *args|
  case method.to_s
  when "compare"
    args[0] <=> args[1]
  when "equals"
    args[0] == args[1]
  end
end)

v.each do |val|
  puts val
end
```

Еще один интересный прием работы с интерфейсами – воспользоваться блоком Ruby в качестве параметра метода в том месте, где обычно вы применили бы

Java-интерфейс с одним-единственным методом. Эту технику можно применять к неперегруженным методам, принимающим один параметр, который является Java-интерфейсом. Если такому методу передается блок, то среда исполнения JRuby пытается сгенерировать прокси-объект, реализующий интерфейс. Если интерфейс содержит несколько перегруженных или неперегруженных методов, то эта процедура становится неоднозначной и, следовательно, непригодной. В примере 1.21 показано, как этот механизм позволяет заметно сократить объем кода, в котором применяется библиотека Java Swing.

Пример 1.21. Реализация Java-интерфейса с помощью блока Ruby

```
frame = javax.swing.JFrame.new
frame.set_size 500,200

a = javax.swing.JButton.new("hello")
b = javax.swing.JButton.new("world")

# определить функцию с помощью блока
a.add_action_listener do |evt|
  puts 'hello'
end

# определить функцию с помощью Proc-объекта Ruby
p = lambda{ |evt| puts 'world' }
b.add_action_listener &p

frame.add a
frame.add b
frame.set_layout(javax.swing.GridLayout.new(1, 2))
frame.show
```

Можно передать и Proc-объект Ruby, если только преобразовать его в блок с помощью оператора `&`.



Java-интерфейс с единственным методом часто называют *типом с единственным абстрактным методом* (single abstract method type), или SAM-типом. Все предложения по добавлению замыканий/блоков в Java 7 стремятся сделать реализацию таких типов гораздо более простой и близкой к тому, что уже имеется в JRuby.

См. также

- Рецепт 5.1 «Создание приложений Swing»

1.10. Открытие Java-классов в JRuby

Задача

Требуется добавить новые методы в Java-класс.

Решение

Импортируйте Java-класс, так чтобы на него можно было сослаться, после чего добавьте методы, как в любой класс Ruby.

Обсуждение

В Ruby определение класса не считается окончательным; в любой момент в класс можно добавить новые методы. Быть может, это одно из самых фундаментальных различий между Java и Ruby. В Java определения классов тесно связаны с именами файлов и структурой каталогов. Полное определение Java-класса `java.util.HashMap` должно находиться в файле `/java/util/HashMap.class`. В Ruby подобных отношений нет и в помине, определение любого класса может быть разбросано по нескольким исходным файлам. JRuby позволяет распространить эту особенность языка и на классы Java. В примере 1.22 показано, как добавить в класс `java.util.HashMap` метод с именем `is?`.

Пример 1.22. Добавление метода в класс `HashMap`

```
include Java

import java.util.HashMap

class HashMap
  def is?(key,value)
    value == get(key)
  end
end
```

Как видите, из нового метода мы можем вызывать методы, определенные в исходном Java-классе. После выполнения этого кода все экземпляры класса `HashMap`, в том числе и созданные ранее, будут содержать новый метод. Это относится даже к экземплярам класса, созданным в Java-коде. В примерах 1.23 и 1.24 показан Java-класс, в котором создается объект `HashMap`, и код на Ruby, где этот класс повторно открывается и в него добавляется новый метод.

Пример 1.23. Простой класс для порождения объекта HashMap

```
package org.jrubycookbook.ch01;
```

```
import java.util.*;
```

```
public class MapMaker {
    public static Map makeMap() {
        Map m = new HashMap();
        m.put("k1", "v1");
        m.put("k2", "v2");
        return m;
    }
}
```

Пример 1.24. Применение семантики открытого класса к экземпляру, созданному в Java-коде

```
include Java
```

```
import java.util.HashMap
import org.jrubycookbook.ch01.MapMaker
```

```
h = MapMaker.makeMap()
```

```
class HashMap
    def isNot?(key, value)
        value != get(key)
    end
end
```

```
puts (h.isNot? 'k1', 'v1')
puts (h.isNot? 'k2', 'v3')
```

Однако добавленные таким образом методы видны только среде исполнения JRuby. Если передать экземпляр модифицированного класса HashMap в Java-код, то новые методы будут недоступны.

В JRuby есть также служебный метод `extend_proxy`, который позволяет добавлять новые методы во все реализации некоторого интерфейса. Воспользовавшись этим механизмом, пример 1.24 можно переписать так, что он будет работать с любой реализацией `java.util.Map` (см. пример 1.25).

Пример 1.25. Использование метода extend_proxy для открытия всех реализаций интерфейса

```
include Java
```

```
import org.jrubycookbook.ch01.MapMaker
```

```
h = MapMaker.makeMap()
JavaUtilities.extend_proxy('java.util.Map') do
```

```
def isNot?(key,value)
  value != get(key)
end

puts (h.isNot? 'k1', 'v1')
puts (h.isNot? 'k2', 'v3')
```

См. также

- Рецепт 1.5 «Обращение к Java-классам из Ruby»

1.11. Конфигурирование Eclipse для разработки на языке JRuby

Задача

Вы пользуетесь интегрированной средой Eclipse Integrated Development Environment для разработки на языке Ruby и хотели бы исполнять Ruby-код с помощью интерпретатора JRuby.

Решение

Если вы пользуетесь подключаемым модулем Ruby Development Tools (RDT), то создайте определение виртуальной машины Ruby, указав на инсталляционный каталог JRuby и задав тип JRuby VM. Если же вы работаете с подключаемым модулем Dynamic Language Toolkit (DLTK), то создайте новое определение интерпретатора Ruby, которое ссылается на сценарий запуска JRuby: *bin\jruby.bat* (для Windows) или *bin/jruby* (для Linux и Mac OS X), находящийся в инсталляционном каталоге JRuby.

Обсуждение

И RDT, и DLTK можно сконфигурировать для работы с несколькими интерпретаторами Ruby. В модуле RDT предусмотрен специальный параметр для интерпретатора JRuby, тогда как DLTK считает JRuby обычным интерпретатором Ruby.

RDT

Подключаемый модуль RDT, который можно скачать с сайта <http://rubyclipse.sourceforge.net>, поддерживает задание нескольких интерпретаторов Ruby исходя из пути к инсталляционному каталогу. Чтобы добавить JRuby в качестве интерпретатора, откройте диалоговое окно **Preferences** (Настройки) и перейдите на страницу **Installed Interpreters** (Установленные интерпретаторы). Нажмите кнопку **Add** (Добавить) – откроется диалоговое окно **Add RubyVM** (Добавить

виртуальную машину Ruby) (рис. 1.5). В списке **RubyVM type** (Тип VM Ruby) выберите значение JRuby VM, а в поле **RubyVM home directory** (Начальный каталог VM Ruby) укажите путь к инсталляционному каталогу JRuby. В поле **RubyVM display name** можете указать более осмысленное название. Задав значения параметров, нажмите кнопку **ОК**.

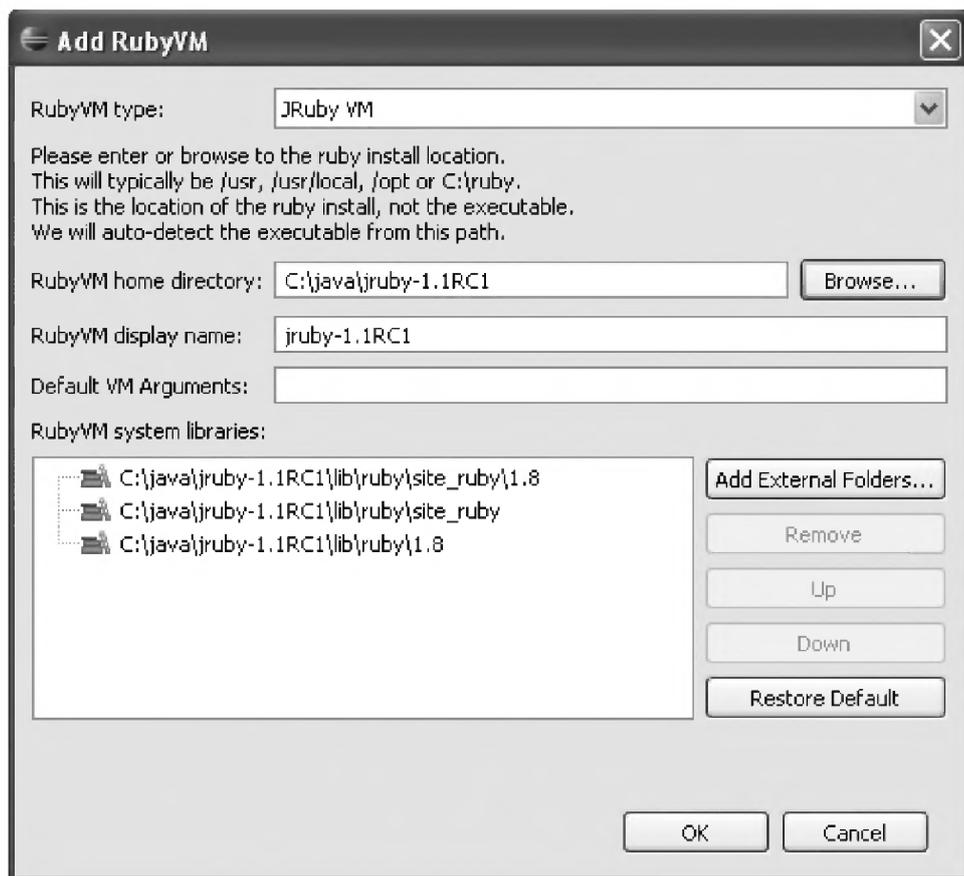


Рис. 1.5. Диалоговое окно Add RubyVM в подключаемом модуле RDT

DLTK

Проект Dynamic Language Toolkit, размещенный на сайте <http://www.eclipse.org/dltk>, финансируется фондом Eclipse Foundation и ставит целью обеспечить общую поддержку динамических языков в рамках среды разработки Eclipse. В настоящее время в DLTK реализована поддержка языков Ruby, TCL и Python. Подключаемый модуль DLTK Ruby не делает различия между стандартным интерпретатором Ruby и интерпретатором JRuby. Как и при конфигурировании RDT, откройте диалоговое окно **Preferences** и перейдите на страницу **Interpreters** (Интерпретаторы). Нажмите кнопку **Add** – откроется диалоговое окно **Add interpreter** (Добавить интерпретатор), показанное на рис. 1.6. В качестве исполняемого файла укажите *bin\jruby.bat* (для Windows) или *bin/jruby* (для Linux и Mac OS X). Как и в RDT, можно еще ввести осмысленное имя интерпретатора. Нажмите кнопку **ОК**.

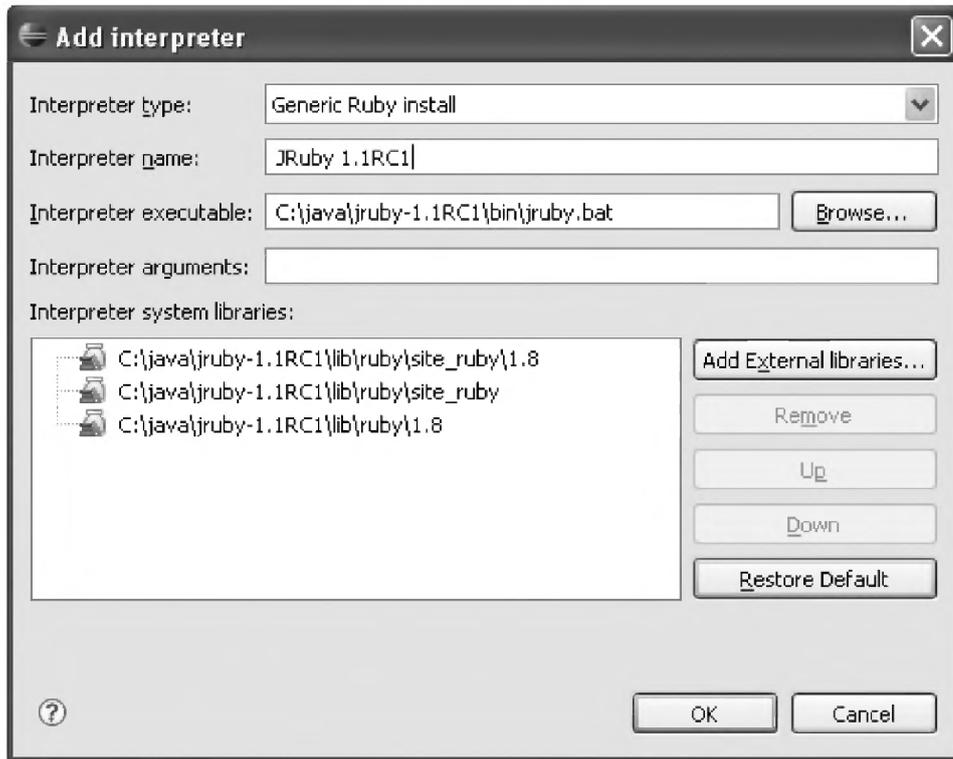


Рис. 1.6. Диалоговое окно *Add interpreter* в подключаемом модуле *DLTK*

Запуск JRuby как Java-приложения

RDT и DLTK безо всякого труда могут взаимодействовать с интерпретатором JRuby, так как проектировались специально для разработки на Ruby. Однако они не позволяют управлять путем поиска классов (classpath), необходимым виртуальной машине Java, на которой исполняется JRuby. Это создает проблему в случае, когда вы ссылаетесь на Java-классы, находящиеся во внешних JAR-файлах. Но ведь интерпретатор JRuby представляет собой обычный Java-класс, поэтому его можно выполнить в Eclipse. Для этого откройте диалоговое окно **Run** (Запуск), выбрав команду **Open Run Dialog...** (Открыть диалог Запустить) из меню **Run**. Выберите пункт **Java Application** (Приложение Java) и нажмите кнопку **New** (Создать) для создания новой конфигурации запуска. В качестве класса Main укажите `org.jruby.Main`. Перейдите на вкладку **Arguments** (Аргументы) и в секции **Program arguments** (Аргументы программы) введите путь к Ruby-файлу, который собирается запускать (вместе с относящимися к конкретному приложению аргументами). В числе аргументов VM должны быть указаны системные свойства `jruby.base`, `jruby.home` и `jruby.lib`. Свойства `jruby.base` и `jruby.home` должны указывать на инсталляционный каталог JRuby, а свойство `jruby.lib` – на его каталог *lib* для JRuby. В среду Eclipse встроен язык выражений; в данном диалоговом окне им можно воспользоваться, чтобы сослаться на переменную среды `JRUBY_HOME` из значений рассматриваемых свойств:

```
-Djruby.base="${env_var:JRUBY_HOME}" -Djruby.home="${env_var:JRUBY_HOME}"
-Djruby.lib="${env_var:JRUBY_HOME}/lib"
```

Наконец, перейдите на вкладку **Classpath** (Путь поиска классов) и добавьте файлы *bsf.jar* и *jruby.jar* из каталога *lib JRuby*, а также все JAR-файлы, необходимые вашей программе. Нажмите кнопку **Run**, чтобы запустить программу.

Eclipse также поддерживает выражения в сообщениях, выводимых пользователю. С помощью этого механизма можно сделать конфигурацию запуска пригодной для повторного использования. Например, вы можете запросить путь к файлу, и тогда пользователь увидит стандартный диалог операционной системы для выбора файла:

```
${file_prompt:Ruby Script Name}
```

Чтобы запросить файл в пределах рабочего пространства, напишите такую команду:

```
${resource_loc:${string_prompt:Ruby Script Name}}
```

В результате пользователя попросят выбрать файл в рабочем пространстве Eclipse, который затем будет автоматически преобразован в путь в файловой системе. Порядок применения таких выражений показан на рис. 1.7.

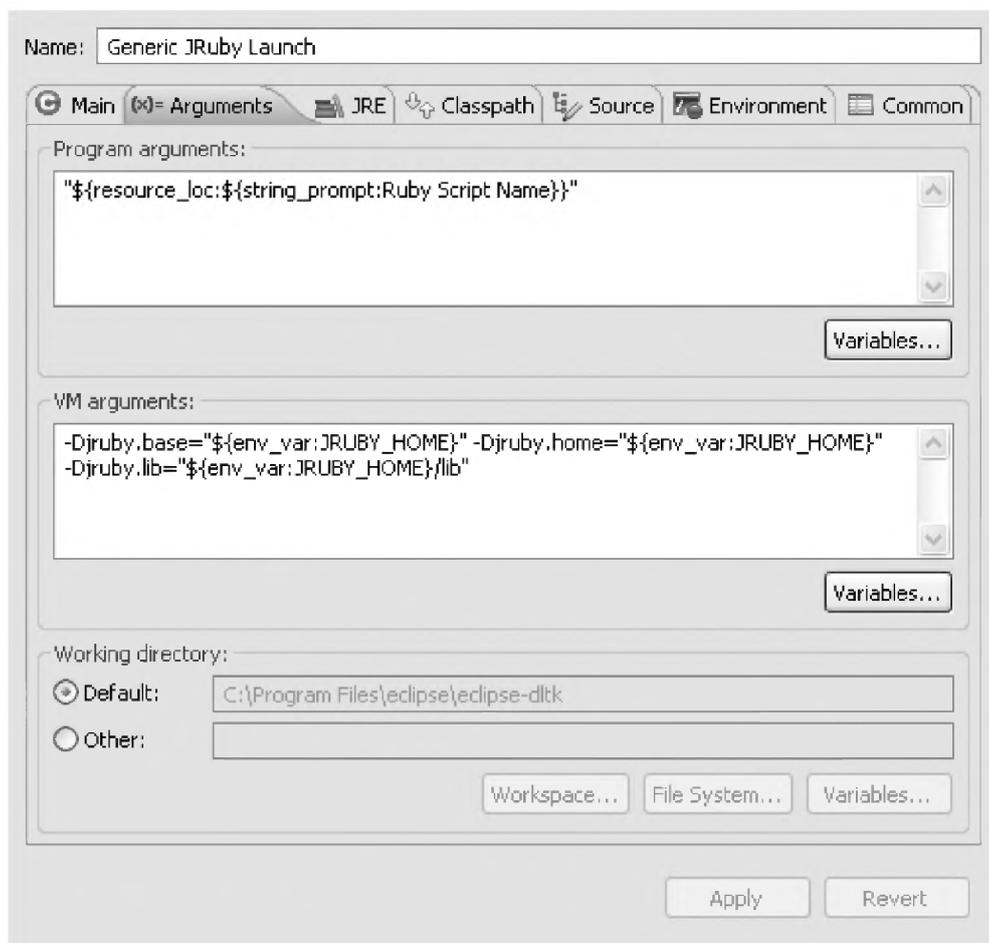


Рис. 1.7. Общая конфигурация запуска JRuby

Запустив таким образом сконфигурированный интерпретатор, вы увидите диалоговое окно, показанное на рис. 1.8, где сможете ввести путь к требуемому Ruby-сценарию в рабочем пространстве. При каждом последующем запуске Eclipse автоматически будет помещать в поле последнее введенное значение.

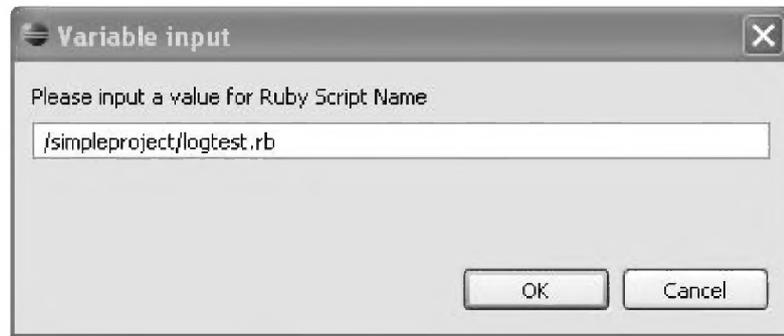


Рис. 1.8. Диалоговое окно для ввода переменной в Eclipse

Отметим, что такой способ конфигурирования запуска не требует наличия RDT или DLTК, хотя эти подключаемые модули все же предоставляют целый ряд полезных функций, в том числе автоматическое завершение кода и интеграцию с системой RDoc.

См. также

- Рецепт 1.1 «Установка JRuby»

1.12. Конфигурирование NetBeans для разработки на JRuby

Задача

Вы хотите разрабатывать на Ruby приложения, в которых используется технология NetBeans.

Решение

Скачайте версию NetBeans 6.5 с сайта <http://www.netbeans.org> и установите ее. NetBeans поставляется в различных комплектациях; дистрибутивы (bundles) Ruby и All включают поддержку разработки на языке Ruby. Помимо Ruby, в дистрибутив All включена поддержка Java, Web, Mobile и C/C++, а также серверов приложений Apache Tomcat и Sun GlassFish.

Если вы уже применяете NetBeans 6.5, то установить поддержку для Ruby позволит диалоговое окно **Plugins** (Подключаемые модули), показанное на рис. 1.9.

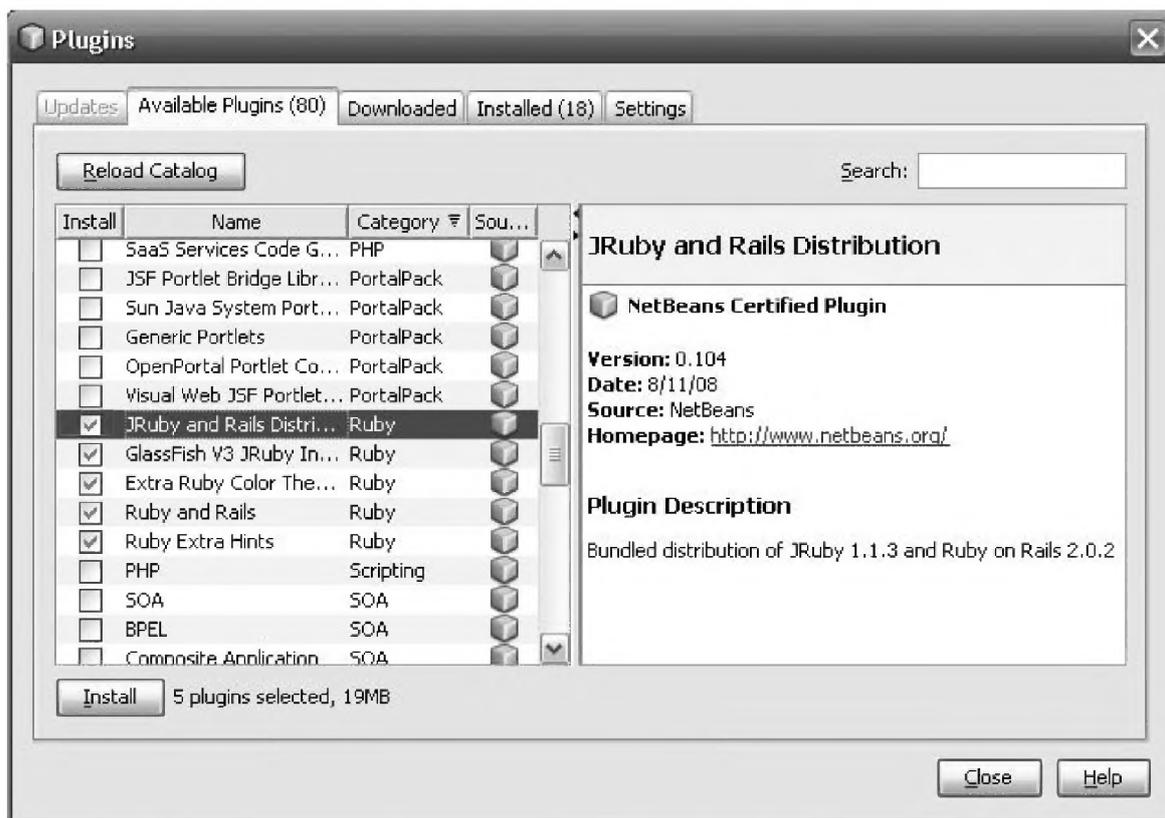


Рис. 1.9. Установка подключаемого модуля NetBeans Ruby из диалогового окна Plugins

Выделенный на рисунке модуль добавляет новые типы проектов: NetBeans для Ruby и Rails, графические отладчики для Ruby и Rails, редактор Ruby Code Editor и клиент RubyGems.

Установив подключаемый модуль для Ruby, перейдите на страницу Ruby в диалоговом окне **Ruby Platforms** (рис. 1.10); здесь вы сможете настроить среду исполнения Ruby для своих проектов. Обратите внимание, что можно добавить новую среду исполнения или модифицировать местоположение репозитория gem-пакетов и уровень отладки. По умолчанию в вашем проекте на Ruby будет использоваться среда исполнения JRuby, поставляемая вместе с подключаемым модулем, но в диалоговом окне свойств проекта можно связать с приложением другую платформу Ruby.

Обсуждение

В течение нескольких лет корпорация Sun играла вторую скрипку в проекте Eclipse, а недавно инвестировала значительные средства в развитие проекта NetBeans, и результат налицо – причем наиболее явно это видно на примере подключаемого модуля для Ruby. В редакторе NetBeans Ruby Code Editor реализованы синтаксическая подсветка, цветовое выделение, поддержка рефакторинга и развитые средства автозавершения кода. Последняя функция иллюстрируется на рис. 1.11.

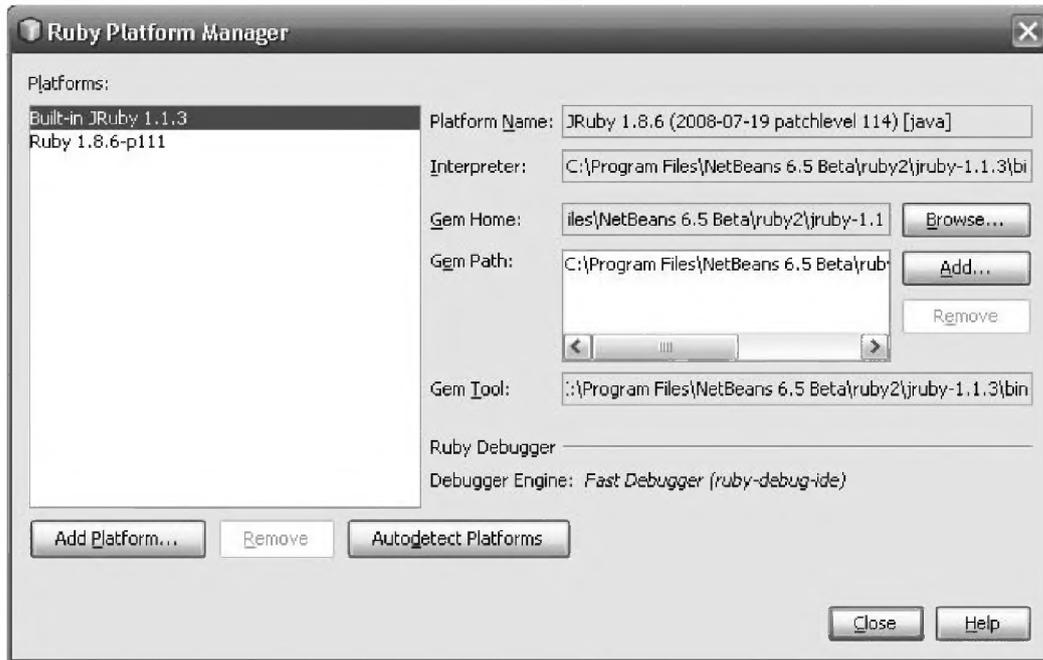


Рис. 1.10. Диалоговое окно Platform Manager из модуля NetBeans Ruby

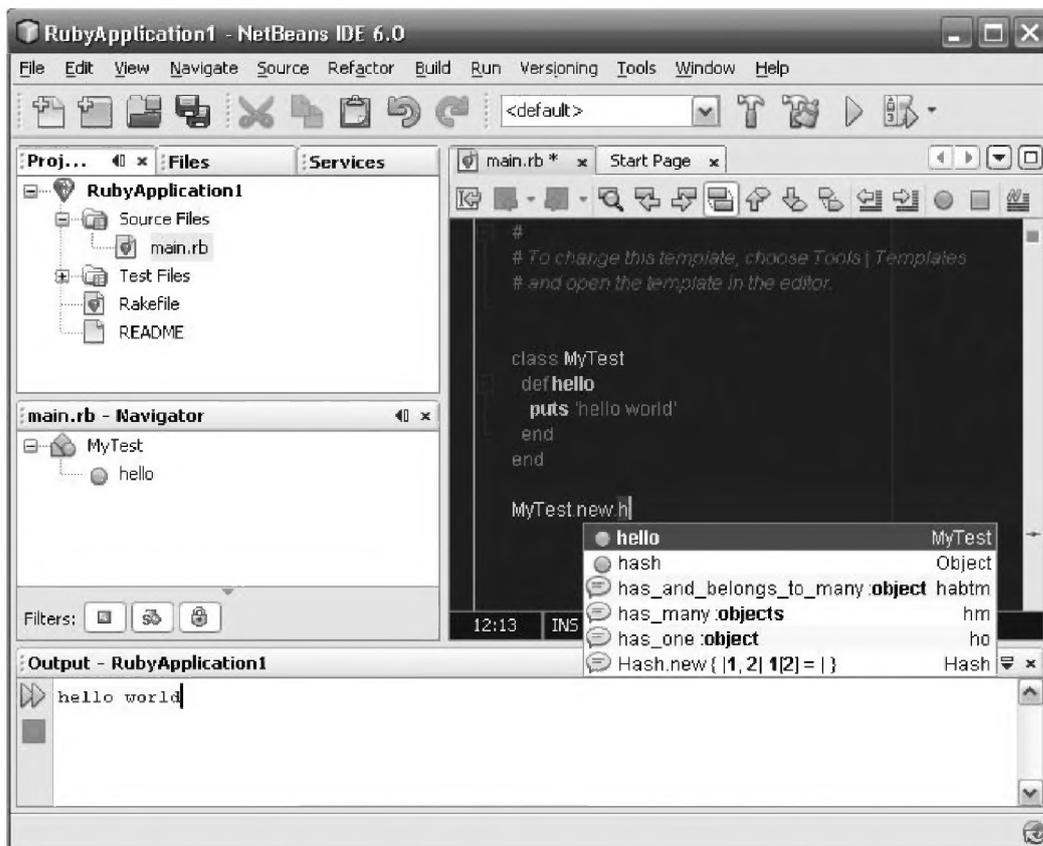


Рис. 1.11. Автоматическое завершение кода в модуле NetBeans Ruby

Редактор выводит в небольшом окошке список возможных методов, включающий как встроенные, так и определенные пользователем Ruby-классы. Если в этот момент нажать пробельную клавишу, то в текст программы будет вставлено полное имя метода.

Можно также изменить шрифт, цвета подсветки и назначить различным операциям комбинации клавиш по своему усмотрению. Настройка выполняется в диалоговом окне **Options** (Параметры), показанном на рис. 1.12. Перейдите на вкладку **Fonts & Colors** (Шрифты и цвета) и выберите профиль из списка **Profile**. В операционной системе Mac OS X разработчикам на Ruby может показаться интересной тема Aloha, заимствованная из редактора TextMate (<http://pages.huikau.com/AlohaTheme.nbm>), поскольку реализованная в ней цветовая палитра и правила выделения им хорошо знакомы. На странице **Keymap** (Раскладка клавиатуры) имеются привязки для Eclipse, Emacs и предыдущих версий NetBeans.

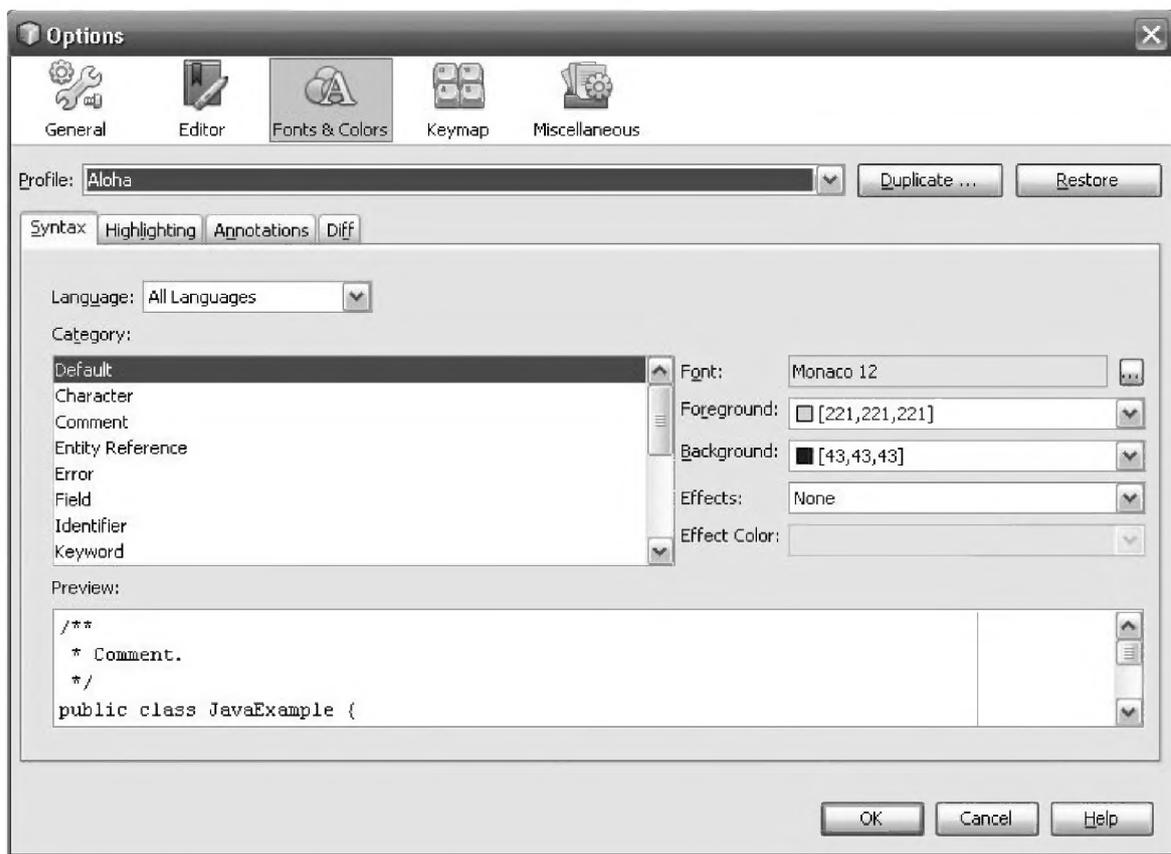


Рис. 1.12. Диалоговое окно NetBeans Fonts & Colors Options

См. также

- Рецепт 2.11 «Развертывание Rails на сервере приложений GlassFish v2 Server»

1.13. Распознавание платформы в приложении JRuby

Задача

Требуется определить, на какой платформе работает среда исполнения Ruby, и подстроить свой код под среду исполнения JRuby.

Решение

Определить, работает ли ваше приложение под управлением JRuby, можно, проанализировав переменную среды `JRUBY_VERSION`. Ее устанавливает только среда исполнения JRuby и никакая другая. В методе `generate_random_number` из примера 1.26 применяется генератор случайных чисел из класса `Java Math`, если код исполняется в среде JRuby, в противном случае вызывается стандартный метод `Ruby rand`.

Пример 1.26. Распознавание платформы JRuby

```
class DetectionExample

  def generate_random_number
    if(defined?(JRUBY_VERSION))
      require 'java'
      puts 'исполняется метод java'
      java.lang.Math.random
    else
      puts 'исполняется метод ruby'
      rand(0)
    end
  end
end

end

d = DetectionExample.new
puts d.generate_random_number
```

Обсуждение

В переменной среды `RUBY_PLATFORM` содержится информация о среде исполнения; JRuby записывает в нее значение `java`. Именно эта переменная применялась для распознавания платформы в предыдущих версиях JRuby, но позже была добавлена переменная `JRUBY_VERSION`, которая определенно говорит, что код исполняется под управлением JRuby, а не какого-то другого интерпретатора Ruby, написанного на Java. Эта новая переменная также дала возможность писать код, зависящий от конкретной версии JRuby.

JRuby on Rails

2.0. Введение	50	2.15. Конфигурирование хранилища сеансов	75
2.1. Установка и настройка Rails	51	2.16. Управление классами, каталогами и прочими файлами, упакованными в WAR-файл	75
2.2. Пакетирование приложения Rails для работы в среде Java EE ...	54	2.17. Изменение имени WAR-файла и местоположения рабочей области	77
2.3. Веб-приложения с внешним репозиторием gem-пакетов	56	2.18. Развертывание приложения Rails в корневом контексте	77
2.4. Конфигурирование сервлета JRuby-Rack	57	2.19. Создание приложения Rails в интегрированной среде Aptana Studio	79
2.5. Пакетирование приложения Rails с источником данных JNDI	58	2.20. Доступ к статическим файлам из приложения Rails, работающего в контейнере Java EE	82
2.6. Развертывание Rails на сервере Tomcat	59		
2.7. Развертывание Rails на сервере JBoss	62		
2.8. Развертывание Rails на сервере Jetty	64		
2.9. Развертывание Rails с помощью jetty_rails	65		
2.10. Развертывание Rails на сервере Mongrel	68		
2.11. Развертывание Rails на сервере GlassFish v2	69		
2.12. Gem-пакет GlassFish v3	71		
2.13. Использование библиотеки ActiveRecord вне Rails	72		
2.14. Получение информации о Java-сервлете	74		

2.0. Введение

С момента появления в середине 2004 года каркас веб-приложений Ruby on Rails очень быстро обрел массу приверженцев в сообществе разработчиков программ для веб. Именно это стало главной причиной общего роста интереса к языку программирования Ruby. Аналогично, способность JRuby выполнять написанные для Rails приложения на виртуальной машине Java привлекло внимание к JRuby. В этой главе мы рассмотрим некоторые приемы запуска приложений Rails в среде Java.

Ruby on Rails – это каркас для разработки веб-приложений, основанный на архитектуре модель-вид-контроллер (MVC). Через весь каркас красной нитью проходит принцип примата соглашения над конфигурацией, и наиболее отчетливо он прослеживается в подсистеме объектно-реляционного отображения (ORM) ActiveRecord. В подсистеме ActiveRecord метаданные, извлекаемые из базы данных (имена таблиц и столбцов), используются для динамического определения классов предметной области. Достаточно добавить новый столбец в таблицу базы данных, как ActiveRecord автоматически включит соответствующее ему поле в ассоциированный класс предметной области.

У запуска приложений Rails под управлением JRuby есть несколько преимуществ:

- Приложения Rails можно развернуть в уже имеющихся контейнерах Java EE, например: Tomcat, JBoss или GlassFish.
- С помощью интерфейса Java Database Connectivity (JDBC) приложение Rails может установить соединение с любой СУБД, для которой существует драйвер JDBC.
- За счет интерфейса к корпоративной службе каталогов Java Naming and Directory Interface (JNDI) приложение Rails может получить доступ к пулу соединений с базой данных, который поддерживается контейнером.

Короче говоря, сочетание JRuby и Rails позволяет создать пакет, который легко интегрируется с существующим окружением Java EE и хорошо приспособлен к работе в системах масштаба предприятия. С точки зрения развертывания, приложение Rails – не более чем еще одно веб-приложение, работающее в среде Java EE; если используются источники данных, совместимые с JNDI, то человеку, отвечающему за развертывание, даже не нужно будет заглядывать в конфигурационные файлы Rails.

Помимо JRuby, есть еще одна важная библиотека, которая обеспечивает сопряжение контейнера Java EE с Rails; она называется JRuby-Rack. По существу, JRuby-Rack представляет собой сервлет-фильтр Java, который направляет запросы приложению Rails, работающему под управлением JRuby. JRuby-Rack создает пул экземпляров среды исполнения JRuby. Конфигурирование сервлета JRuby-Rack обсуждается в рецепте 2.4. Ранее интеграция с контейнерами Java EE и сервлетами основывалась на проекте GoldSpike, но сейчас этот подход считается устаревшим, и вместо него применяется сервлет JRuby-Rack.

В дополнение к JRuby-Rack команда разработчиков JRuby создала инструмент Warbler для упаковки приложения Rails в WAR-файл; это позволяет упростить развертывание.

В середине этой главы мы рассмотрим шаги развертывания приложений Rails на основных серверах приложений Java EE с открытым исходным кодом. Хотя все эти рецепты похожи, мы сочли важным описать детали, специфичные для каждого контейнера. В последних рецептах из этой главы речь пойдет о некоторых дополнительных конфигурационных параметрах и сценариях совместного использования JRuby и Rails.



Когда книга уже была запущена в производство, команда разработчиков Rails анонсировала включение в готовящуюся версию Rails 2.2 ряда изменений, направленных на повышение безопасности ядра относительно потоков. Пока еще слишком рано говорить о том, насколько эффективными окажутся эти изменения, но весьма вероятно, что развертывание приложений Rails на платформе JRuby существенно упростится. Команда, работающая над JRuby, внимательно следит за продвижениями в этой области и без сомнения продолжит развивать описанные в этой главе инструменты, принимая на вооружение все новшества в будущих версиях Rails.

2.1. Установка и настройка Rails

Задача

Требуется запустить Ruby on Rails на платформе JRuby.

Решение

Установите gem-пакет с последней версией Ruby on Rails, выполнив следующую команду:

```
$ jruby -S gem install rails
```

Если вы работаете с версией Rails 2.x, то рекомендуется установить gem-пакет `jruby-openssl`, который позволит задействовать все средства повышения безопасности и варианты хранения сеансовых данных. Этот пакет представляет собой реализацию на Java gem-пакета `openssl`:

```
$ jruby -S gem install jruby-openssl
```

Теперь создайте приложение Rails, которое будет работать с JRuby:

```
$ jruby -S rails MyKillerApplication
```

Протестируйте свое приложение Rails:

```
$ cd MyKillerApplication
$ jruby ./script/server
```

Запустите браузер и перейдите по адресу *http://localhost:3000*. Должно появиться знакомое приветствие Rails (рис. 2.1).

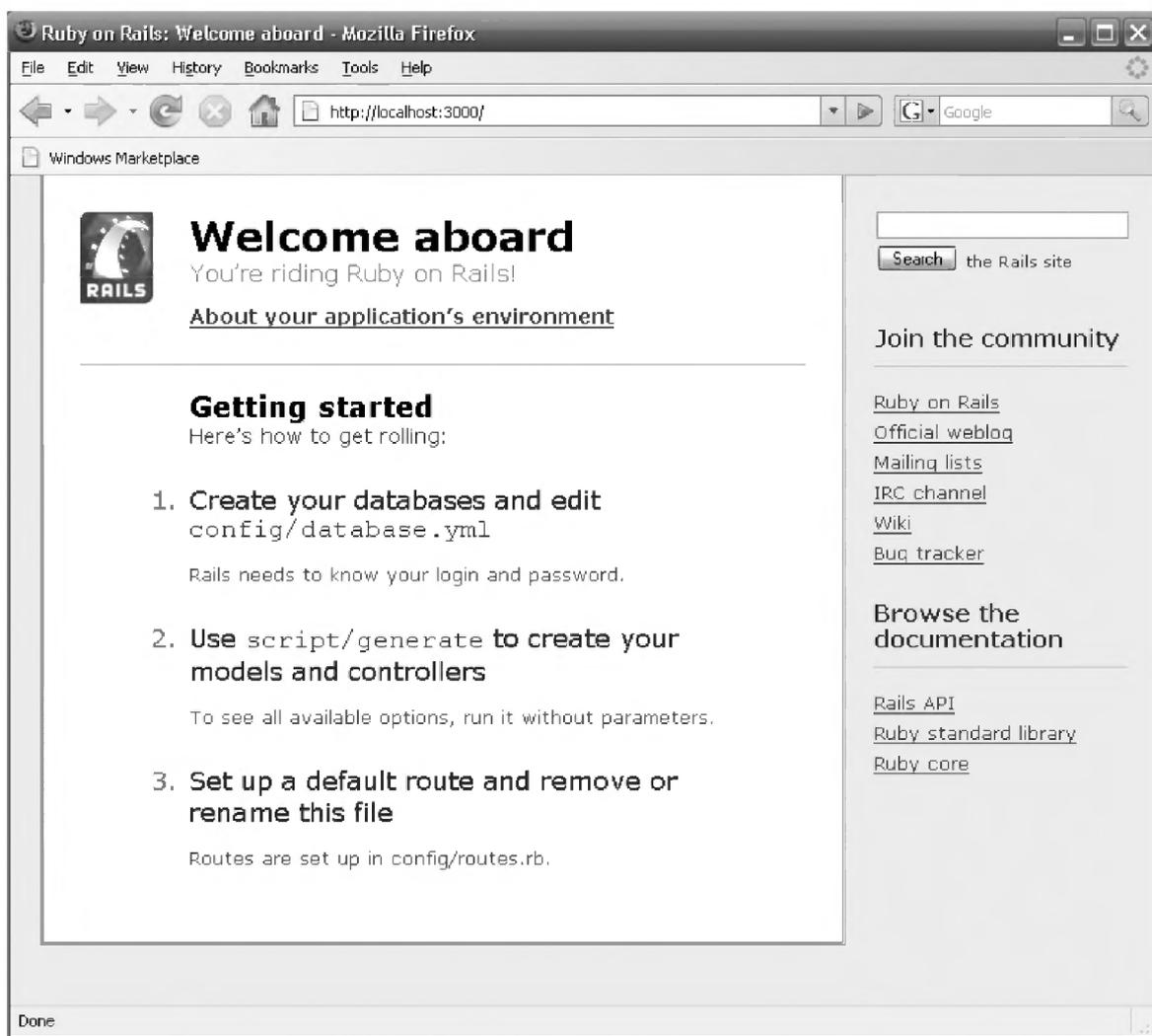


Рис. 2.1. Приветствие Ruby on Rails

Обсуждение

Следующий шаг – сконфигурировать подключение Rails к вашей базе данных. Команда разработчиков JRuby упростила решение этой задачи, дав возможность использовать в Rails хорошо знакомые и повсеместно поддерживаемые драйверы JDBC. Первым делом следует установить gem-пакет `activerecord-jdbc-adapter`:

```
$ jruby -S gem install activerecord-jdbc-adapter -y --no-ri --no-rdoc
```

Этот пакет наделяет входящую в Rails подсистему работы с базами данных, ActiveRecord, возможностью устанавливаться через JDBC соединение с базой и пользоваться пулом соединений. Для конфигурирования необходимо прописать

в стандартном файле Rails *database.yml* URL драйвера JDBC или JNDI-адрес. В примере 2.1 файл *database.yml* сконфигурирован так, что в режиме разработки используется JDBC-соединение, а в режиме эксплуатации – предоставляемый контейнером источник данных `javax.sql.DataSource` с JNDI-именем `java:comp/env/jdbc/rails_db`. Если вы собираетесь использовать стандартный адаптер `jdbc`, то не забудьте включить JDBC-драйвер в путь поиска классов.

Пример 2.1. Пример файла database.yml с параметрами JDBC

```
development:
  adapter: jdbc
  url: jdbc:mysql://localhost:3306/jrubycookbook_development
  driver: com.mysql.jdbc.Driver
  username: jruby
  password: cookbook

production:
  adapter: jdbc
  jndi: java:comp/env/jdbc/rails_db
  driver: com.mysql.jdbc.Driver
```

Проект JRuby Extras содержит ряд адаптеров баз данных для наиболее популярных открытых СУБД, с которыми обычно работают программисты на Java, в том числе: `h2`, `JavaDB (Derby)`, `MySQL`, `HSQLDB (Hypersonic)` и `Postgres`. Gem-пакеты с этими адаптерами позволяют по желанию задать обычные для Rails конфигурационные параметры базы данных в файле *database.yml*, а не указывать класс и URL драйвера JDBC. Кроме того, эти gem-пакеты включают и автоматически загружают соответствующие JAR-файлы с драйверами JDBC, поэтому вручную включать эти классы необязательно. Если вы работаете с одной из поддерживаемых СУБД, то можете установить соответствующий gem-пакет, вписав имя СУБД (`mysql`, `postgres`, `derby`, `hsqldb` или `h2`) в шаблон имени пакета, `activerecord-jdbc<database name>-adapter`. Вот, например, как устанавливается адаптер для СУБД MySQL:

```
$ jruby -S gem install activerecord-jdbcmysql-adapter
```

В файле *database.yml* из примера 2.2 показана конфигурация, в которой используется установленный gem-пакет. Обратите внимание, что в этот раз мы не стали задавать URL драйвера JDBC, а воспользовались стандартными конфигурационными параметрами Rails.

Пример 2.2. Пример файла database.yml, в котором используется адаптер activerecord-jdbcmysql-adapter

```
development:
  adapter: jdbcmysql
  encoding: utf8
  database: jrubycookbook_development
  username: jruby
  password: cookbook
```

```
port: 3306
host: localhost
```

См. также

- Рецепт 2.5 «Пакетирование приложения Rails с источником данных JNDI»
- Проект JRuby Extras <http://rubyforge.org/projects/jruby-extras>

2.2. Пакетирование приложения Rails для работы в среде Java EE

Задача

Требуется подготовить приложение Rails в виде веб-приложения, готового к развертыванию в стандартном контейнере Java EE.

Решение

Воспользуйтесь инструментом Warbler для упаковки приложения Rails в WAR-файл. Для начала установите необходимый gem-пакет:

```
$ jruby -S gem install warbler
```

Этот пакет добавляет команду `warble`, которая позволяет создать, сконфигурировать и стереть WAR-файл. Все команды Warbler следует выполнять, находясь в корневом каталоге приложения Rails. Начнем с создания конфигурационного файла Warbler командой:

```
$ jruby -S warble config
```

Новый конфигурационный файл называется `config/warble.rb`. В нем можно задать параметры создания WAR-файлов и определить, как Rails должен работать в веб-контейнере. Откройте файл `warble.rb` и пропишите в параметре `config.webxml.rails.env` название режима Rails, в котором собираетесь развернуть приложение. Затем включите в хэш `config.gems` все gem-пакеты, необходимые вашему веб-приложению, кроме `rails` (этот пакет включается по умолчанию). В примере 2.3 показан файл `warble.rb`, сконфигурированный, как описано выше.

Пример 2.3. Пример конфигурационного файла Warbler

```
# Значение RAILS_ENV для веб-приложения
config.webxml.rails.env = 'development'

# Список все gem-пакетов, необходимых приложению
config.gems << "activerecord-jdbcmysql-adapter"
config.gems << "jruby-openssl"
```

Теперь все готово для создания WAR-файла. Выполните такую команду:

```
$ jruby -S warble war
```

Она генерирует WAR-файл, который по умолчанию называется так же, как начальный каталог проекта Rails. Например, если ваш проект Rails находится в папке *MyKillerApplication*, то WAR-файл будет называться *MyKillerApplication.war*. Этот WAR-файл можно затем развернуть в контейнере Java EE, применив стандартную для контейнера процедуру.

Обсуждение

Warbler – это gem-пакет для пакетирования приложения Rails для работы в среде Java EE. Он основан на системе сборки Rake и адаптере сервлетов JRuby-Rack. В подразумеваемой по умолчанию реализации адаптера используется сервлет-фильтр, который передает запросы на статические страницы не каркасу Rails, а сервлету контейнера, принимаемому по умолчанию. В ранних версиях Warbler применялся сервлет GoldSpike, но сейчас этот проект объявлен устаревшим и заменен на JRuby-Rack. Библиотека JRuby-Rack включает заглушку сервлета GoldSpike, чтобы сохранить совместимость с унаследованными приложениями на основе GoldSpike.

Неупакованное содержимое WAR-файла находится в папке *tmp/war* в начальном каталоге проекта. Просматривая эту папку, вы обнаружите как некоторые части вашего приложения Rails, так и другие знакомые папки Java EE. Warbler изменяет структуру приложения Rails в соответствии со стандартами Java EE: статическое содержимое, которое обычно находится в папке Rails *public*, перемещается на верхний уровень WAR-файла, а прочие части приложения Rails оказываются в каталоге *WEB-INF*. Кроме того, Warbler помещает в каталог *WEB-INF/lib* архив *jruby-rack.jar*, который содержит классы, необходимые для интеграции с контейнером Java EE, и архив *jruby-complete.jar*, содержащий автономный дистрибутив JRuby со всеми необходимыми ему классами.

Задание *war* на самом деле состоит из нескольких подзаданий, которые можно выполнять по отдельности. Поскольку Warbler – обертка вокруг Rake, то вы можете указать флаг *-T* и получить полный список команд Warbler с краткими описаниями:

```
$ jruby -S warble -T
rake config           # Сгенерировать конфигурационный файл для последующей
                       # модификации
rake pluginize        # Распаковать warbler как модуль, подключаемый к вашему
                       # приложению Rails
rake version          # Вывести версию warbler
rake war              # Создать файл MyKillerApplication.war
rake war:app          # Скопировать все файлы приложения в war-файл
rake war:clean        # Стереть war-файл и вспомогательные каталоги
rake war:gems         # Распаковать все gem-пакеты в каталог WEB-INF/gems
rake war:jar          # Запустить команду jar для создания war-файла
rake war:java_classes # Скопировать java-классы в war-файл
rake war:java_libs   # Скопировать все java-библиотеки в war-файл
rake war:public       # Скопировать все открытые HTML-файлы в корень war-файла
rake war:webxml       # Сгенерировать файл web.xml для веб-приложения
```

По умолчанию Warbler включает последние версии всех gem-пакетов из вашего репозитория, но существует возможность указать конкретные версии при создании WAR-файла. Номер нужной версии gem-пакета указывается в хэше `config.gems`:

```
config.gems["rails"] = "2.0.2"
config.gems["activerecord-jdbcmysql-adapter"] = "0.8.2"
```

См. также

- Рецепт 2.1 «Установка и настройка Rails»

2.3. Веб-приложения с внешним репозиторием gem-пакетов

Задача

Вы не хотите включать gem-пакеты в один пакет с веб-приложением, а предпочитаете воспользоваться репозиторием в файловой системе.

Решение

Бывает так, что нужно использовать репозиторий gem-пакетов, отличный от подразумеваемого средой исполнения JRuby по умолчанию. Например, это имеет смысл, когда вы поддерживаете общий набор gem-пакетов, доступных как C Ruby, так и JRuby. Чтобы веб-приложение обращалось к отдельному репозиторию, нужно задать системное свойство `gem.path` или `gem.home`. Их можно задать как в дескрипторном файле WAR-архива, *web.xml*, так и с помощью системного свойства при запуске контейнера (см. пример 2.4).

Пример 2.4. Задание контекстного параметра `gem.path` в файле `web.xml`

```
<context-param>
  <param-name>gem.path</param-name>
  <param-value>C:\projects\jruby\jruby-1.1\lib\ruby\gems</param-value>
</context-param>
```

<!-- Или так

```
<context-param>
  <param-name>gem.home</param-name>
  <param-value>C:\projects\jruby\jruby-1.1\lib\ruby\gems</param-value>
</context-param-->
```

Свойство `gem.path` можно задать и в качестве параметра при запуске контейнера сервлетов:

```
$ java -jar start.jar etc/jetty.xml \
-Dgem.path="C:\projects\jruby\jruby-1.1\lib\ruby\gems"
```

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

2.4. Конфигурирование сервлета JRuby-Rack

Задача

Требуется сконфигурировать в контейнере несколько экземпляров среды исполнения JRuby.

Решение

Задайте подходящие параметры в файле *warble.rb*:

```
config.webxml.jruby.min.runtimes = 2
config.webxml.jruby.max.runtimes = 4
```

Сгенерируйте WAR-файл для Rails:

```
$ jruby -S warble war
```

Обсуждение

Сервлет JRuby-Rack позволяет интегрировать Rails в большинство контейнеров Java EE. Поскольку до выхода версии 2.2 многие части Rails были небезопасны относительно потоков, то в одной среде исполнения нельзя одновременно обрабатывать несколько запросов. JRuby-Rack использует конфигурируемый пул экземпляров среды исполнения JRuby, которым диспетчер направляет адресованные Rails запросы. Количество одновременно обрабатываемых запросов ограничено количеством доступных экземпляров среды исполнения. Дополнительные запросы ставятся в очередь, где ожидают освобождения среды. Настоятельно рекомендуется задавать максимальное количество экземпляров среды в режиме эксплуатации, поскольку Warbler по умолчанию не налагает никаких ограничений. Ниже перечислены относящиеся к этой теме конфигурационные параметры:

```
config.webxml.jruby.max.runtimes
```

Задаёт максимальное количество активных экземпляров среды исполнения JRuby в пуле, а, следовательно, и максимальное количество одновременно обрабатываемых запросов. По умолчанию не ограничено.

```
config.webxml.jruby.min.runtimes
```

Задаёт минимальное количество экземпляров среды исполнения в пуле. Столько экземпляров создается в момент запуска приложения. По умолчанию 0.

```
config.webxml.jruby.runtime.initializer.threads
```

Определяет, сколько потоков будет задействовано для инициализации экземпляров среды исполнения JRuby в пуле. Это значение зависит от того, сколько экземпляров среды вы хотите создать в самом начале, и от желательного времени инициализации пула. По умолчанию 4.

```
config.webxml.jruby.runtime.timeout.sec
```

Сколько миллисекунд поступивший запрос может ожидать в очереди, прежде чем среда исполнения JRuby вернет код ошибки. По умолчанию 30 секунд.

Минимальное и максимальное количество экземпляров среды исполнения зависит от располагаемых ресурсов: процессора, памяти и операционной системы. Среда исполнения JRuby потребляет много памяти, поэтому рекомендуется запускать ее, выделив достаточно памяти для поколения постоянных объектов и для кучи. Особенно это относится к случаю, когда количество экземпляров среды велико.



Те разработчики, которые переходят на новую версию Warbler, а раньше работали с сервлетом GoldSpike, могут и дальше использовать существующий файл *warble.rb* для конфигурирования пулов JRuby. Адаптер JRuby-Rack поддерживает унаследованные от GoldSpike конфигурационные параметры, однако рекомендуется все же перейти на параметры JRuby-Rack, поскольку в будущих версиях эта поддержка может быть прекращена.

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

2.5. Пакетирование приложения Rails с источником данных JNDI

Задача

Требуется сконфигурировать приложение Rails для доступа к источнику данных JDBC через интерфейс Java Naming and Directory Interface (JNDI).

Решение

Установите gem-пакет *activerecord-jdbc-adapter* (как в рецепте 2.1) и отредактируйте файл *database.yml*. Служба поиска имен JNDI реализована в пакете адаптера *jdbc*. Запишите в параметр *driver* имя класса JDBC-драйвера для вашей СУБД и укажите местоположение в смысле JNDI источника данных JDBC. В следующем примере файл *database.yml* сконфигурирован в расчете на использование фабрики JDBC для СУБД MySQL:

```
development:
  adapter: jdbc
  jndi: java:comp/env/jdbc/rails_db
  driver: com.mysql.jdbc.Driver
```

Используйте Warbler для создания пакета, содержащего приложение Rails (см. рецепт 2.2). В файле *warble.rb* задайте справочное имя ресурса для источника данных JNDI:

```
# Имя источника данных JNDI
config.webxml.jndi = 'jdbc/rails_db'
```

Создайте WAR-файл, запустив задание war:

```
$ jruby -S warble war
```

Обсуждение

Задания war и war:webxml создают или перезаписывают дескрипторный файл */WEB-INF/web.xml* для развертывания в веб-контейнере Java EE. Файл создается в рабочем каталоге *tmp/war*. Оба задания добавляют определение resource-ref и всю необходимую информацию о новом источнике данных JDBC. Ниже приведен пример файла *web.xml* для приложения Rails, в котором используется источник данных JNDI со справочным именем jdbc/rails_db:

```
<resource-ref>
  <res-ref-name>jdbc/rails_db</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Ничто не мешает вручную изменить значения, которые сгенерировал Warbler. Если вы хотите отредактировать файл *web.xml*, определив другие источники данных или значения конфигурационных параметров либо справочных имен, то можете воспользоваться заданием war:jar, чтобы пропустить шаг генерации этого файла и поместить все файлы, находящиеся в рабочем каталоге, в WAR-файл приложения:

```
$ jruby -S warble war:jar
```

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

2.6. Развертывание Rails на сервере Tomcat

Задача

Требуется развернуть приложение Rails, используя в качестве контейнера Java EE сервер приложений Apache Tomcat.

Решение

Поместите свое приложение Rails в WAR-файл, предназначенный для развертывания в контейнере Java EE (см. рецепт 2.2). Скопируйте получившийся WAR-файл в каталог *webapps* на сервере Tomcat. Если вы воспользовались каким-нибудь gem-пакетом с адаптером JDBC для конкретной СУБД, то можете сразу запустить Tomcat. Если же в вашем приложении Rails применяется стандартный адаптер jdbc, то включите его JAR-файл в путь поиска классов или скопируйте JAR-файл в каталог *\$TOMCAT_HOME/common/lib*.

Не забудьте установить переменную среду `JAVA_HOME`, так чтобы она указывала на папку, в которую установлена Java. В целях повышения производительности запускайте Tomcat с флагом `-server`. Также рекомендуется ограничить размер кучи и поколения постоянных объектов (PermGen), чтобы потенциальные утечки памяти не привели к исчерпанию всех ресурсов сервера с последующей перезагрузкой машины.

Windows

```
> set JAVA_HOME=c:\Program Files\Java\jdk1.5.0_12
> set CATALINA_OPTS=-server -Xms512m -Xmx1024m -XX:PermSize=256m \
-XX:MaxPermSize=512m
> catalina.bat start
```

Linux и OS X

```
$ export JAVA_HOME=/usr/java/jdk1.5.0_12
$ export CATALINA_OPTS='-server -Xms512m -Xmx1024m -XX:PermSize=256m
-XX:MaxPermSize=512m'
$ ./catalina.sh start
```

Обсуждение



Важно понимать, как JRuby работает с памятью; это поможет вам правильно настроить свои приложения. Виртуальная машина Java выделяет несколько отдельных областей памяти. Одна из них – поколение постоянных объектов (PermGen) – зарезервирована для внутренних представлений классов и структур данных, необходимых самой JVM. Другая область, куча, известна более широко и обычно используется для хранения данных классов. Своим успехом JRuby в немалой степени обязан способности обходить правила, действующие для статически компилируемого языка (Java) путем порождения классов и структур данных во время выполнения. Но за это приходится расплачиваться тем, что в некоторых случаях JRuby вынужден создавать довольно много объектов, которые хранятся в поколении постоянных объектов, а не в куче. Возьмем в качестве примера Rails, здесь обработка одного-единственного запроса может приводить к созданию сотен объектов JRuby. Такая эксплуатация области

PermGen во много раз превышает лимиты, на которые она рассчитана по умолчанию, поэтому размер выделяемой по умолчанию памяти часто оказывается недостаточен. Разработчики JRuby приложили немало усилий для смягчения этой проблемы; например, несколько экземпляров среды исполнения JRuby могут совместно использовать область PermGen. Тем не менее, вы должны проявить осмотрительность и задать начальное и максимальное значения размера PermGen и кучи, особенно для приложений, работающих в режиме эксплуатации.

Если вы используете Tomcat с источником данных JNDI, то начните с создания пакета, содержащего приложение Rails (см. рецепт 2.2). Перейдите в рабочий каталог *tmp/war* и добавьте файл *context.xml* в папку *META-INF*. Если такой папки нет, создайте ее. В примере 2.5 показано, как следует определить в файле *context.xml* ресурс для доступа к СУБД MySQL. В определение ресурса входит информация о соединении с базой данных, JNDI-имя ресурса и контекстный путь к приложению, который совпадает с началом унифицированного идентификатора ресурса (URI) вашего веб-приложения.

Пример 2.5. Конфигурация JNDI для сервера Tomcat в файле *context.xml*

```
<Context path="/MyKillerApplication" docBase="MyKillerApplication"
  debug="5" reloadable="true" crossContext="true">

  <Resource name="jdbc/rails_db" auth="Container" type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="root" password="password"
    driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/jrubycookbook_development?autoReconnect=true"/>
</Context>
```

Ресурс можно было бы определить и в файле *\$TOMCAT_HOME/conf/server.xml*, но авторы Tomcat не советуют так поступать, поскольку этот файл распространяется на все веб-приложения. Подход с определением ресурса в том же пакете, где находится веб-приложение, следует предпочесть по двум причинам: во-первых, он соответствует рекомендуемой практике организации кода, а, во-вторых, позволяет переопределить или изменить источник данных путем повторного развертывания автономного веб-приложения без перезагрузки сервера.

Перестройте WAR-файл с помощью команды *warble*, переместите свое приложение в папку развертывания и запустите сервер, как описано в настоящем решении.

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»
- Рецепт 2.5 «Пакетирование приложения Rails с источником данных JNDI»

2.7. Развертывание Rails на сервере JBoss

Задача

Требуется развернуть приложение Rails на сервере приложений JBoss.

Решение

Поместите свое приложение Rails в WAR-файл, предназначенный для развертывания в контейнере Java EE (см. рецепт 2.2). Скопируйте получившийся WAR-файл в каталог `$JBOSS_HOME\server\default\deploy` (в JBoss он по умолчанию используется для развертывания) или в любой другой каталог развертывания, который прописан в конфигурационных файлах JBoss. Если же в вашем приложении Rails применяется стандартный адаптер `jdbc`, не зависящий от СУБД, то включите его JAR-файл в путь поиска классов. Можете вместо этого скопировать JAR-файл, содержащий драйвер JDBC, в каталог `$JBOSS_HOME\server\default\lib`, если вы не меняли настроек, подразумеваемых по умолчанию.

Запускайте сервер приложений с флагом `-server` и задайте ограничения на размер кучи и поколения постоянных объектов (PermGen). Обычно это делается с помощью переменной среды `JAVA_OPTS`.

Windows

```
> SET JAVA_HOME=c:\Program Files\Java\jdk1.5.0_12
> SET JAVA_OPTS=-server -Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m
> run.bat
```

Linux и OS X

```
$ export JAVA_HOME=/usr/java/jdk1.5.0_12
$ export JAVA_OPTS='-server -Xms512m -Xmx1024m -XX:PermSize=256m\
-XX:MaxPermSize=512m'
$ ./run.sh
```

Обсуждение

Если для установления соединения с базой данных из приложения Rails вы используете JNDI-ресурс, то нужно будет создать на сервере JBoss описание источника данных. В дистрибутиве JBoss имеются примеры конфигураций источников данных для большинства популярных СУБД; находятся они в папке примеров `$JBOSS_HOME\docs\examples\jca`. С изучения этих примеров и надо начинать настройку базы данных. Отредактировав файл, вы можете развернуть описание источника данных на сервере JBoss, скопировав файл в каталог развертывания. По умолчанию таковым считается каталог `$JBOSS_HOME\server\default\deploy`.

Если вы работаете с СУБД MySQL, то измените конфигурационный параметр `<jndi-name>MySqlDS</jndi-name>`, записав в него имя своего источника данных,

в примере ниже `rails_db`. Задайте в том же файле прочие параметры своей базы данных. В примере 2.6 представлен отредактированный файл `mysql-ds.xml`, содержащий определение источника данных для гипотетического приложения. Обратите внимание, что параметр `jndi-name` не содержит префикс `jdbc`. Скопируйте файл `mysql-ds.xml` в каталог развертывания.

Пример 2.6. Пример конфигурационного файла `mysql-ds.xml` с описанием источника данных для сервера JBoss

```
<datasources>
  <local-tx-datasource>
    <jndi-name>rails_db</jndi-name>
    <connection-url>
      jdbc:mysql://localhost:3306/jrubycookbook_development
    </connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password>password</password>
    <connection-property name="autoReconnect">true</connection-property>
    <!-- Отображение типов для JBoss 4.0 -->
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Хотя вы и определили источник данных, надо еще установить соответствие между этим ресурсом и веб-приложением. Для этого служит файл `jbossweb.xml`, который включается в один пакет с веб-приложением. Warbler этот файл не генерирует, так что вам придется создать его вручную в папке `WEB-INF` внутри рабочего каталога Warbler `tmp/war` (см. пример 2.7).

Пример 2.7. Пример дескрипторного файла для развертывания на сервере JBoss

```
<jboss-web>
  <context-root>/MyKillerApplication</context-root>
  <resource-ref>
    <res-ref-name>jdbc/rails_db</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:rails_db</jndi-name>
  </resource-ref>
</jboss-web>
```

Вы можете сказать, что ваш источник данных должен считаться источником по умолчанию. Для этого назовите его `DefaultDS` и удалите принимаемое по умолчанию описание источника данных, которое при установке было записано в файл `$_JBOSS_HOME/server/all/deploy/hsqldb-ds.xml`.

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»
- Рецепт 2.5 «Пакетирование приложения Rails с источником данных JNDI»

2.8. Развертывание Rails на сервере Jetty

Задача

Требуется развернуть приложение Rails в контейнере сервлетов Jetty.

Решение

Поместите свое приложение Rails в WAR-файл, предназначенный для развертывания в контейнере Java EE (см. рецепт 2.2). Если вы определили JDBC-соединение с помощью адаптера jdbc или воспользовались источником данных JNDI, то не забудьте включить JAR-файл адаптера JDBC в путь поиска классов или скопируйте этот файл в каталог `$JETTY_HOME/lib`, чтобы развернутые веб-приложения могли его найти. Поместите WAR-файл в папку `$JETTY_HOME/webapp`. Запустите сервер с флагом `-server VM` и подразумеваемыми по умолчанию размерами кучи и поколения постоянных объектов:

```
$ java -server -Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m\  
-jar start.jar etc/jetty.xml
```

Обсуждение

Если вы предпочитаете описывать соединение с базой данных с помощью ресурса JNDI, то определите источник данных в WAR-файле. Создайте в папке `WEBINF` внутри временного каталога файл `jetty-env.xml`. Образец такого файла для СУБД MySQL приведен в примере 2.8.

Пример 2.8. Пример файла `jetty-env.xml`

```
<?xml version="1.0"?>  
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"  
  "http://jetty.mortbay.org/configure.dtd">  
  
<Configure class="org.mortbay.jetty.webapp.WebAppContext">  
  <New id="rails_db" class="org.mortbay.jetty.plus.naming.Resource">  
    <Arg>jdbc/rails_db</Arg>  
    <Arg>  
      <New class="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource">  
        <Set name="Url">jdbc:mysql://localhost:3306/jrubycookbook_development</Set>  
        <Set name="User">root</Set>  
        <Set name="Password">password</Set>  
      </New>  
    </Arg>  
  </New>  
</Configure>
```

Упакуйте свое приложение с помощью Warbler и разверните его в контейнере Jetty. По умолчанию модуль JNDI для Jetty в стандартном каталоге развертывания *webapps* не активирован, поэтому либо измените файл *\$JETTY_HOME/etc/jetty.xml*, активировав JNDI для этого каталога, либо сконфигурируйте Jetty так, чтобы использовался другой каталог. По соглашению, принятому в Jetty, приложения, нуждающиеся в JNDI, устанавливаются в каталог *webapps-plus*:

```
$ copy MyKillerApplication.war $JETTY_HOME/webapps-plus
```

В состав дистрибутива Jetty заботливо включен файл *\$JETTY_HOME/etc/jetty-plus.xml*, который конфигурирует Jetty под использование этого каталога. Находясь в начальном каталоге Jetty, запустите Jetty с поддержкой JNDI, выполнив следующую команду:

```
$ java -server -Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m -jar\
start.jar etc/jetty.xml etc/jetty-plus.xml
```

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»
- Рецепт 2.5 «Пакетирование приложения Rails с источником данных JNDI»

2.9. Развертывание Rails с помощью jetty_rails

Задача

Многие из нынешних разработчиков приложений Rails никогда не встречались с процедурой пакетирования приложения для контейнера Java EE. Им привычна другая процедура запуска приложения – перейти на верхний уровень проекта Rails и запустить один из двух популярных веб-серверов: Mongrel или WEBrick. Вы хотели бы запустить сервер приложений Jetty, сконфигурированный под ваше приложение Rails, но при этом использовать метод развертывания, лучше знакомый разработчикам для Rails.

Решение

Воспользуйтесь gem-пакетом *jetty_rails*, который позволяет запускать приложение Rails на сервере Jetty, не выполняя процедуру пакетирования для контейнера Java EE. Первым делом установите пакет *jetty_rails*:

```
jrubby -S gem install jetty_rails
```

Затем перейдите на верхний уровень приложения Rails и запустите сервер Jetty:

```
$ cd jrubbycook_application
$ jrubby -S jetty_rails
```

Обсуждение

Перечень наиболее употребительных параметров запуска выводит следующая команда:

```
jruby -S jetty_rails -help
```

Параметры `port` и `environment` обычно используются при запуске HTTP-серверов Mongrel и WEBrick:

- Флаг `-port <port>` или `-p <port>` задает номер порта веб-приложения. По умолчанию равен 3000.
- Флаг `-environment <env>` или `-e <env>` задает окружение, в котором выполняется Rails. По умолчанию подразумевается режим разработки `development`.
- Флаг `-context-path <path>` или `-u <path>` позволяет изменить контекстный корень ваших приложений. Не забудьте уведомить приложение Rails об этом изменении, включив в файл `environment.rb` следующую строку:

```
ActionController::AbstractRequest.relative_url_root = "/my_new_context_root"
```

- Флаг `-c` или `-config` загружает конфигурационные параметры сервера из внешнего файла. Если путь к файлу не задан, сервер будет по умолчанию искать файл `config/jetty_rails.yml`.

Конфигурационный файл полезен не только потому, что позволяет убрать параметры запуска из числа аргументов в командной строке. Начиная с версии 0.6, этот файл позволяет настраивать приложение путем задания конфигурационных параметров JRuby и Jetty, задействовать все средства развитой многоуровневой системы конфигурирования и запускать несколько приложений Rails в одном экземпляре Jetty.

Ниже демонстрируются некоторые из этих возможностей: мы конфигурируем несколько приложений Rails, каждое в своем контексте, задавая для них различные значения параметров `content_path` и `port`. Значения по умолчанию располагаются в конце файла и могут быть переопределены в секции конфигурационного файла, относящейся к конкретному приложению. Обратите внимание на то, что в режиме разработки начальное число экземпляров среды исполнения уменьшено с пяти до двух.

Пример 2.9. Конфигурационный файл `jetty_rails.xml`

```
---
:servers:
- :context_path: /dev-one
  :adapter: :rails
  :environment: development
  :base: development-dir
  :port: 3000
```

```
:jruby_initial_runtimes: 2
- :context_path: /prod-one
  :adapter: :rails
  :base: production-dir
- :port: 4000
  :apps:
- :context_path: /dev-two
  :adapter: :rails
  :base: development-dir
  :environment: development
  :jruby_initial_runtimes: 2
- :context_path: /prod-two
  :base: production-dir
  :adapter: :rails
:environment: production
:jruby_initial_runtimes: 5
:jruby_max_runtimes: 10
:thread_pool_min: 5
:thread_pool_max: 40
:acceptor_size: 20
```

Приведем перечень менее известных конфигурационных параметров:

`jruby_initial_runtimes`

Определяет, сколько экземпляров среды исполнения JRuby должно быть создано при запуске. Отметим, что для каждого контекста приложения создаются отдельные пулы сред исполнения.

`jruby_max_runtimes`

Задаёт максимальное количество сред исполнения в пуле и тем самым ограничивает число одновременно обрабатываемых запросов.

`thread_pool_min`

Задаёт начальный размер пула потоков для обработки запросов.

`thread_pool_max`

Задаёт максимальный размер пула потоков для обработки запросов.

`acceptor_size`

Задаёт количество акцепторов для объекта `SelectChannelConnector`, принадлежащего написанной на Java подсистеме сетевого ввода/вывода (NIO) сервера Jetty.

См. также

- Сайт Jetty-Rails <http://jetty-rails.rubyforge.org>

2.10. Развертывание Rails на сервере Mongrel

Задача

Требуется запустить написанное на JRuby приложение Rails на веб-сервере Mongrel.

Решение

Установите gem-пакет Mongrel. Исполняемый в среде JRuby инсталлятор gem-пакетов должен выбрать последнюю версию этого пакета для Java:

```
$ jruby -S gem install mongrel --no-ri --no-rdoc
Updating metadata for 165 gems from http://gems.rubyforge.org
.....
complete
Successfully installed gem_plugin-0.2.3
Successfully installed mongrel-1.1.4-java
2 gems installed
```

Включите JAR-файл адаптера JDBC для вашей СУБД в путь поиска классов, если только вы не используете специфичный для этой СУБД адаптер `jdbс`, который содержит нужный драйвер и загружает его. Перейдите в начальный каталог приложения Rails и запустите Mongrel:

```
$ jruby -S mongrel_rails start
** Starting Mongrel listening at 0.0.0.0:3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready. TERM => stop. USR2 => restart. INT => stop (no restart).
** Rails signals registered. HUP => reload (without restart). It might not work well.
** Mongrel 1.1.4 available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Обсуждение

Mongrel – это небольшой высокопроизводительный веб-сервер, который первоначально был написан на языках Ruby и C. Относительно недавно написанные на C части были перенесены на Java, так что теперь Mongrel можно запускать на платформе JRuby. Это стало знаменательным этапом проекта, учитывая, сколько разработчиков для Rails применяют Mongrel в режиме разработки и эксплуатации.

Существует экспериментальный gem-пакет `mongrel_jcluster`, поддерживающий кластеризацию для Java-версии Mongrel. К сожалению, пока он работает

только на платформах Linux, OS X и Cygwin для Windows. Стандартная оболочка Windows DOS еще не поддерживается. Этот gem-пакет позволяет запускать и останавливать группы серверов Mongrel, стремясь частично воспроизвести функциональность пакета `mongrel_cluster`, не совместимого с JRuby. Начните с установки gem-пакета:

```
$ jruby -S gem install mongrel_jcluster
Successfully installed mongrel_jcluster-0.0.1
1 gem installed
```

Затем сгенерируйте конфигурационный файл для кластера серверов Mongrel:

```
$ jruby -S mongrel_rails jcluster::configure -p 4000 -N 3 -e development -R 20202\
-K thesecretkey
Writing configuration file to config/mongrel_jcluster.yml.
```

В результате создается файл `config/mongrel_jcluster.yml`, в котором можно задать номер порта для первого сервера в кластере (флаг `-p`), количество серверов (флаг `-N`) и режим работы кластера (флаг `-e`).

Запустите кластер серверов Mongrel такой командой:

```
$ jruby -S mongrel_rails jcluster::start
Starting JRuby server...
Starting 3 Mongrel servers...
```

Открыв в браузере адреса `http://localhost:4000`, `http://localhost:4001` и `http://localhost:4002`, убедитесь, что кластер успешно запустился. Чтобы остановить кластер, выполните следующую команду:

```
$ jruby -S mongrel_rails jcluster::stop
Stopping 3 Mongrel servers...
```

2.11. Развертывание Rails на сервере GlassFish v2

Задача

Требуется развернуть приложение Rails на сервере приложений GlassFish v2.

Решение

Установите сервер GlassFish и перейдите в его начальный каталог. Настройте область развертывания и сконфигурируйте сам сервер с помощью входящего в дистрибутив задания `ant`:

```
$ $GLASSFISH_HOME/lib/ant/bin/ant -f setup.xml
```

В результате будет установлено несколько библиотек и создан каталог развертывания `$GLASSFISH_HOME\domains\domain1\autodeploy`. Создайте WAR-файл,

содержащий пакет с вашим приложением Rails для развертывания в контейнере Java EE (см. рецепт 2.2). Если вы определили JDBC-соединение с помощью адаптера jdbc или источника данных JNDI, не забудьте включить соответствующий JAR-файл в путь поиска классов или скопировать его в каталог `$GLASSFISH_HOME/lib`, чтобы веб-приложения могли его найти.

Запустите сервер следующей командой:

```
$ $GLASSFISH_HOME/bin/asadmin start-domain
```

Подождите несколько секунд, дав серверу возможность установить WAR-файл с вашим приложением (рис. 2.2).

```
C:\WINDOWS\system32\cmd.exe
C:\glassfish>bin\asadmin start-domain
Starting Domain domain1, please wait.
Log redirected to C:\glassfish\domains\domain1\logs\server.log.
Redirecting output to C:\glassfish\domains\domain1\logs\server.log
Domain domain1 is ready to receive client requests. Additional services are being started in background.
Domain [domain1] is running [Sun Java System Application Server 9.1_01 (build b09d-fcs)] with its configuration and logs at: [C:\glassfish\domains\...].
Admin Console is available at [http://localhost:4848].
Use the same port [4848] for "asadmin" commands.
User web applications are available at these URLs:
[http://localhost:8080 https://localhost:8181].
Following web-contexts are available:
[/web1 /__wstx-services].
Standard JMX Clients (like JConsole) can connect to JMXServiceURL:
[service:jmx:rmi:///jndi/rmi://hh9a5b:8686/jmxrmi] for domain management purposes.
Domain listens on at least following ports for connections:
[8080 8181 4848 3700 3820 3920 8686].
Domain does not support application server clusters and other standalone instances.

C:\glassfish>
```

Рис. 2.2. Запуск сервера GlassFish

Откройте браузер и введите URL `http://localhost:8080/MyKillerApplication`, чтобы посмотреть на свое приложение.

Обсуждение

Для тех приложений Rails, где используется источник данных JNDI, имеется команда `asadmin`, которая позволяет задать свойства источника данных. В примере ниже по адресу `jdbc/rails_db` создается пул соединений с сервером MySQL:

```
$ $GLASSFISH_HOME/bin/asadmin create-jdbc-connection-pool -datasourceclassname \
com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource \
-restype javax.sql.DataSource -property User=root:Password=password:\
URL=jdbc:mysql://localhost:3316/jrubycookbook_development jdbc/rails_db
Command create-jdbc-connection-pool executed successfully.
```

Затем сделайте новый источник данных доступным своему WAR-файлу и другим установленным на том же сервере приложения Java EE:

```
$ $GLASSFISH_HOME\bin\asadmin create-jdbc-resource -connectionpoolid \  
jdbc/rails_db jdbc/rails_db  
Command create-jdbc-resource executed successfully.
```

См. также

- Рецепт 2.12 «Gem-пакет GlassFish v3»

2.12. Gem-пакет GlassFish v3

Задача

Требуется запустить приложение Rails на сервере приложений GlassFish v3.

Решение

Прежде всего установите gem-пакет GlassFish v3:

```
$ jruby -S gem install glassfish  
Successfully installed glassfish-0.1.2-universal-java  
1 gem installed
```

Запустите свое приложение Rails командой `glassfish_rails`, появившейся в третьей версии. Вы должны запускать сервер из того каталога, где находится приложение Rails*:

```
$ jruby -S glassfish_rails MyKillerApplication
```

Откройте в браузере URL `http://localhost:3000` – должно появиться стандартное приглашение Rails.

Обсуждение

Сервер GlassFish v3 – это последняя по времени попытка корпорации Sun создать общепризнанный Java EE-сервер. Оформлен он в виде gem-пакета Ruby, а конфигурирование для запуска приложений Rails требует всего нескольких простых команд.

В этом пакете реализован пул сред исполнения JRuby, очень похожий на тот, что организован в сервлете GoldSpike. Количество экземпляров среды исполнения в пуле задается при запуске сервера с помощью флага `-n` (или `-runtimes`). В примерах ниже запускается сервер с тремя экземплярами среды исполнения в пуле:

```
$ jruby -S glassfish_rails MyKillerApplication -n 3
```

или

```
$ jruby -S glassfish_rails MyKillerApplication --runtimes 3
```

* Вероятно, в будущих версиях это требование будет отменено.

См. также

- Рецепт 2.11 «Развертывание Rails на сервере GlassFish v2»

2.13. Использование библиотеки ActiveRecord вне Rails

Задача

Требуется применить библиотеку ActiveRecord в качестве средства объектно-реляционного отображения (ORM) в приложении, работающем вне каркаса Rails.

Решение

Если вы не устанавливали Rails, то установите отдельно gem-пакет activerecord:

```
$ jruby -S gem install activerecord -no-ri -no-rdoc
```

Установите также пакет activerecord-jdbc-adapter, который дает доступ к базе данных по JDBC-соединению:

```
$ jruby -S gem install activerecord-jdbc-adapter -no-ri -no-rdoc
```

Включите JAR-файл адаптера JDBC для вашей СУБД в путь поиска классов или в папку *lib* JRuby (если только вы не используете специфичный для СУБД адаптер). Например, для работы с СУБД MySQL потребуется gem-пакет activerecord-jdbcmysql-adapter. Дополнительную информацию о драйверах и gem-пакетах для конкретных СУБД см. в рецепте 2.1.

```
$ jruby -S gem install activerecord-jdbcmysql-adapter -no-ri -no-rdoc
```

Создайте YAML-файл *database.yml*, в котором определяются параметры соединения с базой данных. Образец такого файла приведен в примере 2.10.

Пример 2.10. Образец файла *database.yml*

```
development:
  adapter: jdbcmysql
  database: jrubbycookbook_development
  host: localhost
  port: 3306
  username: root
  password: password
```

После того как параметры заданы, вы можете загрузить этот файл и установить соединение с одной из описанных в нем баз данных. В примере 2.11 мы соединяемся с базой данных *development*, определенной в конфигурационном файле из примера 2.10. Установив соединение, мы выполняем запрос и обходим набор результатов. И в конце вызываем один из динамических методов поиска, которые библиотека ActiveRecord присоединяет к объектам.

Пример 2.11. Загрузка файла database.yml и доступ к базе данных

```
require 'rubygems'
gem 'activerecord-jdbcmysql-adapter'
require 'active_record'
require 'yaml'

@connections = YAML.load_file("database.yml")
ActiveRecord::Base.establish_connection(@connections["development"])

stmt = "select id, title from games"
@val = ActiveRecord::Base.connection.select_all(stmt)
@val.each do |g|
  puts "game id: #{g["id"]} #{g["title"]}"
end

class Game < ActiveRecord::Base
end

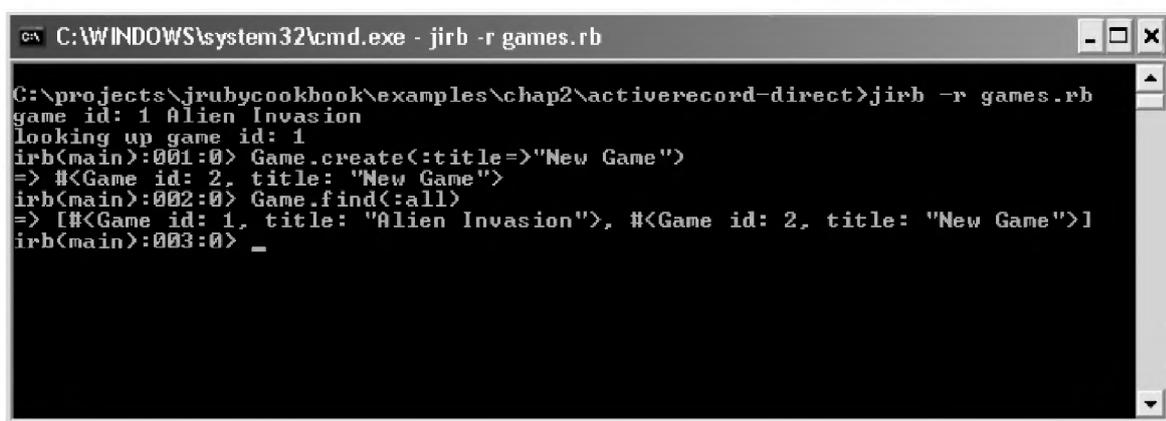
puts "found game id: #{Game.find(1).id}"
```

Вот что выводит эта программа:

```
$ jruby games.rb
game id: 1 Alien Invasion
looking up game id: 1
```

Обсуждение

Поставляемая вместе с JRuby интерактивная консоль jirb – отличный инструмент для создания прототипа и тестирования приложения. Запустив нашу программу в jirb, мы получим интерактивный сеанс работы с базой данных (рис. 2.3).



```
C:\WINDOWS\system32\cmd.exe - jirb -r games.rb
C:\projects\jrubbycookbook\examples\chap2\activerecord-direct>jirb -r games.rb
game id: 1 Alien Invasion
looking up game id: 1
irb(main):001:0> Game.create(:title=>"New Game")
=> #<Game id: 2, title: "New Game">
irb(main):002:0> Game.find(:all)
=> [#<Game id: 1, title: "Alien Invasion">, #<Game id: 2, title: "New Game">]
irb(main):003:0> _
```

Рис. 2.3. Сеанс jirb, в котором используются библиотека ActiveRecord и JDBC-соединение

2.14. Получение информации о Java-сервлете

Задача

Требуется получить из контроллера Rails доступ к объекту, представляющему запрос к Java-сервлету.

Решение

Для каждого запроса сервлет-фильтр JRuby-Rack предоставляет приложению Rails несколько относящихся к запросу переменных. Для доступа к стандартному объекту `javax.servlet.ServletRequest` следует найти в хэше окружения, который формирует Rack, значение с ключом `java.servlet_request`. Из того же хэша по ключу `java.servlet_context` можно получить объект `ServletContext`, который доступен также через глобальную переменную `$servlet_context`. В примере 2.12 показан контроллер, в котором используются некоторые из вышеупомянутых переменных.

Пример 2.12. Доступ к объектам Java-сервлета из контроллера Rails

```
class HelloWorldController < ApplicationController
  def hello
    ctx = request.env['java.servlet_context']
    puts "server info: #{ctx.server_info}"
    puts "server info: #{servlet_context.server_info}"
    req = request.env['java.servlet_request']
    puts "uri: #{req.request_uri}"
    puts "query string: #{req.query_string}"
    puts "port: #{req.server_port}"
    puts "param hello: #{req.get_parameter("hello")}"
    puts "session id: #{req.get_session.id}"
  end
end
```

При обращении к URL `http://localhost:3000/MyKillerApplication/hello?hello=world` в протокол контейнера будут выведены такие сообщения:

```
server info: jetty-6.1.9
server info: jetty-6.1.9
uri: /hello_world/hello
query string: hello=world
port: 3000
param hello: world
session id: 2026
```

Обсуждение

Сервлет JRuby-Rack не позволяет обратиться из вашего контроллера к объекту `ServletResponse`. Такая возможность поддерживалась в ранних версиях Warbler,

в которых еще использовался сервлет GoldSpike, но после интеграции с JRuby-Rack была исключена.

См. также

- Рецепт 2.4 «Конфигурирование сервлета JRuby-Rack»

2.15. Конфигурирование хранилища сеансов

Задача

Требуется сконфигурировать механизм хранения информации о сеансах для использования в приложении Rails.

Решение

Откройте файл *web.xml* для своего веб-приложения и присвойте контекстному параметру `jruby.session_store` значение `db`, как показано ниже:

```
<context-param>
  <param-name>jruby.session_store</param-name>
  <param-value>db</param-value>
  <!-- Это значение говорит, что заботу о хранении сеансов берет на себя Rails -->
</context-param>
```

Обсуждение

По умолчанию сервлет-фильтр JRuby-Rack пользуется хранилищем сеансов, встроенным в контейнер сервлетов Java EE. Если записать в контекстный параметр `jruby.session_store` значение `db`, то JRuby-Rack поручит заботу о сохранении сеансов механизму, встроенному в Rails.

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

2.16. Управление классами, каталогами и прочими файлами, упакованными в WAR-файл

Задача

Требуется включить в WAR-файл или исключить из него некие классы или иные файлы.

Решение

Откройте конфигурационный файл Warbler `config/warbler.rb` и измените показанные ниже параметры:

```
# Какие каталоги проекта включать в веб-приложение.
config.dirs = %w(app config lib log vendor tmp)
# Дополнительные включаемые файлы/каталоги, помимо перечисленных в config.dirs
config.includes = FileList["db"]
# Дополнительные исключаемые файлы/каталоги
config.excludes = FileList["lib/tasks/*"]
# Дополнительные включаемые jar-файлы. Если jar-файл помещается в библиотеку
# (и более нигде не исключается), то указывать его здесь не нужно.
# JRuby и JRuby-Rack уже считаются включенными в этот список.
# Если вы задаете значение непосредственно, не забудьте включить свои версии.
config.java_libs += FileList["lib/java/*.jar"]
# Отдельные Java-классы и различные файлы, помещаемые в папку WEB-INF/classes.
config.java_classes = FileList["target/classes/**/*.*"]
# Одно или несколько отображений путей, определяющих, как следует копировать
# Java-классы в папку WEB-INF/classes. Следующее отображение путей соответствует
# приведенному выше значению параметра java_classes. Дополнительную информацию
# о задании отображения путей см. на странице
# http://rake.rubyforge.org/classes/String.html#M000017
config.pathmaps.java_classes << "%{target/classes/,}"
```

Обсуждение

По умолчанию Warbler включает в генерируемый WAR-файл среду исполнения JRuby и сервлет JRuby-Rack. Но иногда предпочтительнее установить эти JAR-файлы в область совместно используемых библиотек, а не упаковывать вместе с каждым веб-приложением. Этого можно достичь за счет создания разделяемых пакетов, но некоторым разработчикам больше нравится смешанный подход, при котором WAR-файл содержит все зависимые gem-пакеты, за исключением среды исполнения JRuby и сервлета JRuby-Rack. Свойство `config.java_libs` представляет собой обычный массив Ruby, поэтому для исключения тех или иных элементов в процессе сборки пакета Warbler'ом можно использовать хорошо знакомые операции с массивами. Например, вызвать метод `reject!`, передав ему регулярное выражение для исключения из WAR-файла всех версий JRuby и JRuby-Rack:

```
config.java_libs.reject! {|lib| lib =~ /jruby-complete|jruby-rack/ }
```

После изменения этих конфигурационных параметров рекомендуется запустить между сборками задание `war:clean`, чтобы в WAR-файл случайно не попали ненужные файлы. Это тем более стоит делать, если вы экспериментируете с заданием правил исключения.

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

2.17. Изменение имени WAR-файла и местоположения рабочей области

Задача

Требуется изменить имя WAR-файла и/или местоположение рабочей области.

Решение

По умолчанию Warbler выбирает для генерируемого WAR-файла такое же имя, как у каталога, содержащего приложение Rails. Это имя можно изменить с помощью параметра `config.war_name` в конфигурационном файле `config/warbler.rb`:

```
# Имя war-файла file (без расширения .war) – по умолчанию совпадает с базовым именем
# RAILS_ROOT
config.war_name = "mywar"
```

Можно также изменить местоположение рабочей области, где хранятся исходные файлы, которые должны войти в окончательный WAR-файл. Для этого служит параметр `config.staging_dir` в файле `warbler.rb`:

```
# Рабочий каталог для формирования WAR-файла
config.staging_dir = "tmp/war"
```

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»
- Рецепт 2.18 «Развертывание приложения Rails в корневом контексте»

2.18. Развертывание приложения Rails в корневом контексте

Задача

Требуется, чтобы веб-приложение Java EE было доступно из корневого контекста контейнера сервлетов.

Решение

В общем случае проще всего решить эту задачу, упаковав приложение Rails в файл с именем `ROOT.war`. Для этого достаточно изменить один параметр в конфигурационном файле `warble.rb`:

```
config.war_name = «ROOT»
```

Но перед тем как разворачивать этот WAR-файл, обязательно удалите каталоги с именем *ROOT* и файлы *ROOT.war* из всех каталогов развертывания в вашем контейнере.

Обсуждение

Соглашение о том, что имя файла *ROOT.war* означает желание развернуть приложение в корневом контексте контейнера сервлетов, общепринято, хотя и не оговорено в стандарте Java EE. Для каждого контейнера определен свой дескриптор развертывания. Примеры таких дескрипторных файлов мы видели в предыдущих рецептах. Если вы пользуетесь источниками данных JNDI, то необходимо будет модифицировать дескрипторы развертывания, так чтобы они соответствовали имени контекста.

Tomcat

Отредактируйте файл *context.xml* в каталоге *META-INF* внутри рабочей области (см. рецепт 2.7). Атрибутам *path* и *docBase* присвойте значение */* (пример 2.13). По умолчанию Warbler не создает этот файл, так что вам придется создать его вручную и заново собрать WAR-файл.

Пример 2.13. Изменение контекстного пути для развертывания на сервере Tomcat

```
<Context path="/" docBase="/" debug="5" reloadable="true" crossContext="true">
  <Resource name="jdbc/rails_db" auth="Container" type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="root" password="password"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/
    jrubbycookbook_development?autoReconnect=true"/>
</Context>
```

JBoss

Отредактируйте файл *jboss-web.xml* в каталоге *WEB-INF* внутри рабочей области (см. рецепт 2.8). Элемент *context-root* присвойте значение */* (пример 2.14). По умолчанию Warbler не создает этот файл, так что вам придется создать его вручную и заново собрать WAR-файл.

Пример 2.14. Изменение контекстного пути для развертывания на сервере JBoss

```
<jboss-web>
  <context-root>/</context-root>
  <resource-ref>
    <res-ref-name>jdbc/rails_db</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:rails_db</jndi-name>
  </resource-ref>
</jboss-web>
```

Jetty

Отредактируйте файл *jetty-web.xml* в каталоге *WEB-INF* внутри рабочей области (см. рецепт 2.9). Добавьте конфигурационный файл, показанный в примере 2.15. По умолчанию Warbler не создает этот файл, так что вам придется создать его вручную и заново собрать WAR-файл.

Пример 2.15. Изменение контекстного пути для развертывания на сервере Jetty

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
  "http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">
  <Set name="contextPath"/></Set>
</Configure>
```

Для того чтобы Jetty мог найти источник данных JNDI, не нужно задавать никакие специальные конфигурационные параметры.

См. также

- Рецепт 2.6 «Развертывание Rails на сервере Tomcat»
- Рецепт 2.7 «Развертывание Rails на сервере JBoss»
- Рецепт 2.8 «Развертывание Rails на сервере Jetty»
- Рецепт 2.17 «Изменение имени WAR-файла и местоположения рабочей области»

2.19. Создание приложения Rails в интегрированной среде Aptana Studio

Задача

Требуется создать приложение Rails в Aptana Studio.

Решение

Скачайте и установите программу Aptana Studio с сайта компании Aptana по адресу <http://www.aptana.com/download>. Запустите программу, выберите из меню пункт **Help** ⇒ **Aptana Start Page** и пролистайте раздел **RadRails** в колонке **Plugins**. Нажмите кнопку **Install** на начальной странице и завершите работу мастера установки. Установить этот подключаемый модуль можно также, выбрав пункт RadRails из списка, который расположен на вкладке **Plugin Manager** на нижней панели, и щелкнув по значку установки. Оба варианта показаны на рис. 2.4.

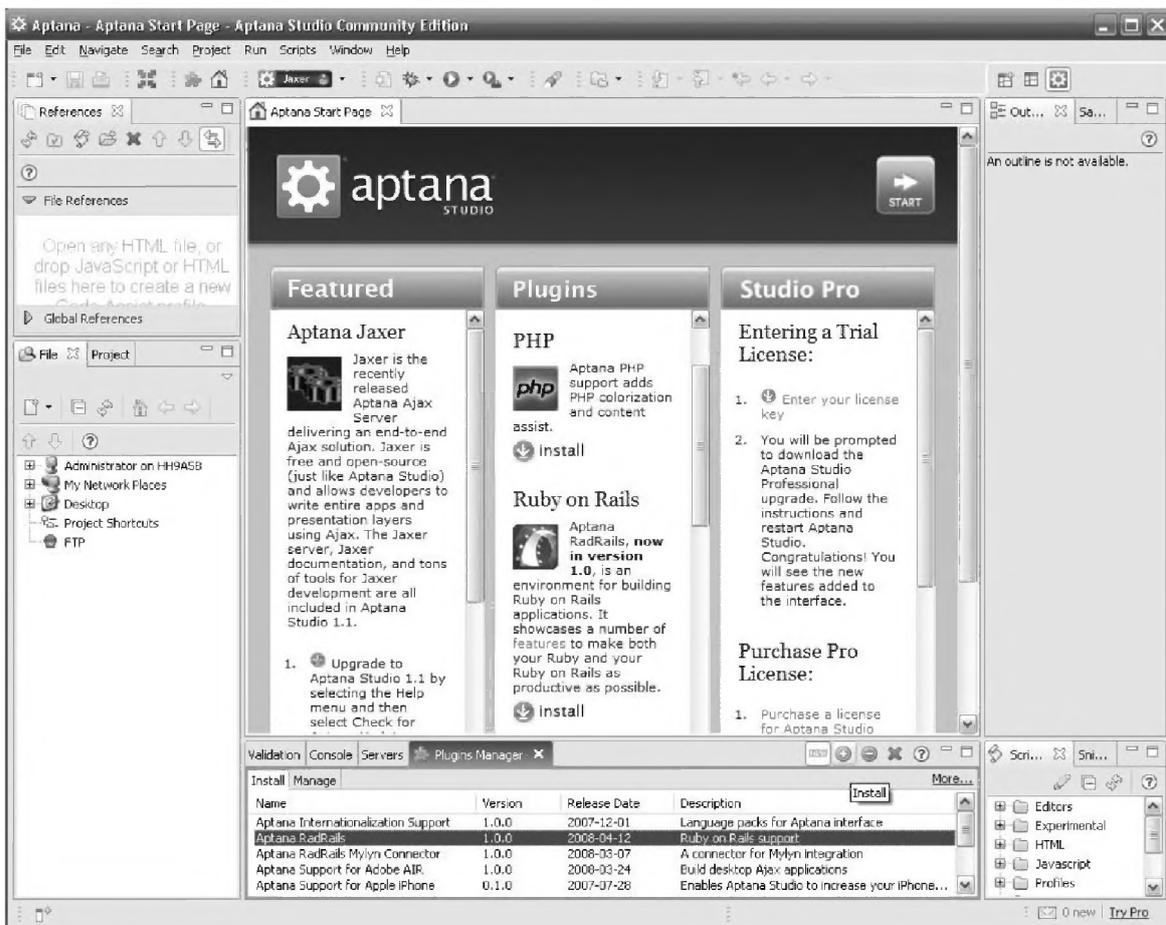


Рис. 2.4. Aptana Studio: варианты установки модуля RadRails

Программа Aptana построена на основе платформы Eclipse IDE. Поэтому для конфигурирования среды исполнения JRuby и задания других параметров Eclipse следуйте рецепту Recipe 1.11. Закончив настройку, выберите представление (perspective) RadRails, щелкнув по значку RadRails или выбрав пункт **Other** ⇒ **RadRails** из списка представлений в правом верхнем углу окна. Создайте новое приложение Rails, выбрав из главного меню команду **File** ⇒ **New** ⇒ **Rails Project**. Назовите свой проект и выберите одну из поддерживаемых СУБД. После нажатия кнопки **Finish** модуль RadRails сгенерирует необходимые приложению Rails файлы, которые будут перечислены в окне обозревателя файлов Rails слева. По умолчанию мастер также создаст и запустит экземпляр сервера Mongrel. В редакторе должно появиться окно приветствия Aptana, показанное на рис. 2.5. Откройте конфигурационный файл базы данных `config/database.yml`, присутствующий в окне обозревателя, и введите параметры своей базы данных. Чтобы запустить или остановить сервер Mongrel, перейдите на вкладку **Servers**, которая находится в среднем нижнем окне. Выберите из списка свое приложение Rails и с помощью предлагаемых элементов управления запустите сервер в обычном или отладочном режиме.

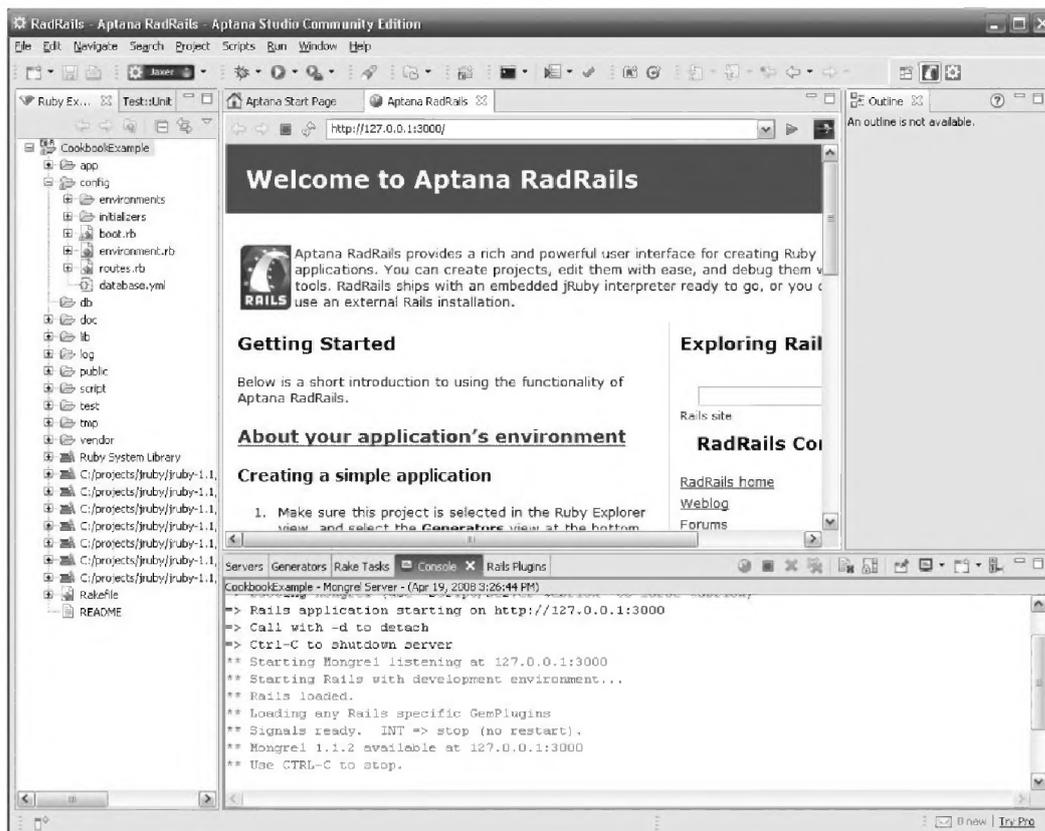


Рис. 2.5. Графический интерфейс RadRails и окно приветствия

Обсуждение

Оболочка Rails Shell появилась в модуле RadRails версии 1.0. Она позволяет вводить команды Rails. Перейдите на вкладку **Console** на нижней панели или выберите пункт **Open a Rails Shell** в меню консоли. Окно оболочки и местоположение значков показаны на рис. 2.6. Оболочка Rails Shell дополняет графические средства выполнения типичных для Rails задач и делает IDE более удобной тем разработчикам Rails, которые привыкли администрировать свои приложения из командной строки. Rails Shell позволяет выполнять порождающие сценарии, за-

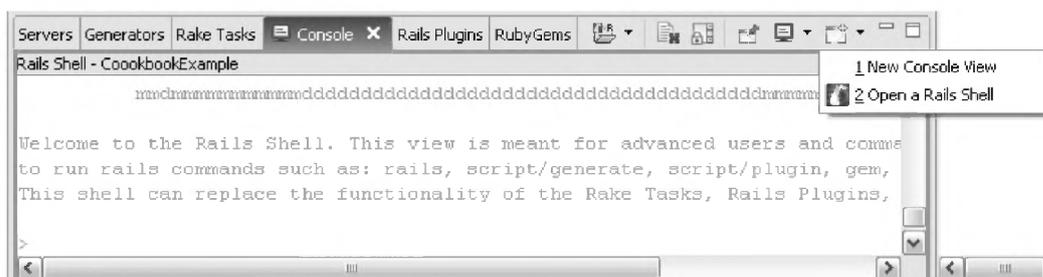


Рис. 2.6. Командная оболочка Aptana Rails Shell

дания Rake и миграции, а также создавать новые проекты Rails и устанавливать gem-пакеты и подключаемые модули.

См. также

- Рецепт 1.11 « Конфигурирование Eclipse для разработки на языке JRuby »

2.20. Доступ к статическим файлам из приложения Rails, работающего в контейнере Java EE

Задача

В WAR-файле, в который Warbler упаковывает приложение Rails, статическое содержимое отделено от исполняемого кода и помещено в каталог верхнего уровня. Это создает проблемы для некоторых функций Rails, например `render :file`, поскольку генерируемые пути к файлам оказываются неправильными. Хотелось бы, чтобы приложение Rails могло обслуживать запросы к статическому содержимому как при стандартном развертывании, так и при развертывании из файла, созданного Warbler.

Решение

Добавьте в свое приложение Rails точку подключения, создав файл `public_dir.rb` в каталоге `initializers`. Проверьте переменную `$servlet_context`, которая устанавливается только при работе в среде Java EE, и установите путь к каталогу `public` в зависимости от того, существует эта переменная или нет. Такая техника подмены каталога `public` демонстрируется в примере 2.16.

Пример 2.16. Код обнаружения каталога public

```
PUBLIC_DIR = if defined?($servlet_context)
  $servlet_context.getRealPath('/')
else
  "#{RAILS_ROOT}" + '/public'
end
```

Замените в своем приложении Rails все обращения `render :file => «/public/data/jobs.log»` на `render :file => «#{PUBLIC_DIR}/data/jobs.log»`.

Обсуждение

Потребуется также подправить внутренние функции Rails, которые строят пути к статическим файлам. Для исправления функции `render_optional_error_file` в модуле `ActionController` можно добавить в файл `application.rb` вашего приложения код, приведенный в примере 2.17. Новый модуль с модифицированным методом подмешивается к оригинальному модулю `ActionController` на этапе выполнения.

Пример 2.17. Модификация функций, обслуживающих запросы на статические файлы

```
module Cookbook
  module PublicRescueExtensions
    protected
    def render_optional_error_file(status_code)
      status = interpret_status(status_code)
      path = "#{PUBLIC_DIR}/#{status[0,3]}.html"
      if File.exists?(path)
        render :file => path, :status => status
      else
        head status
      end
    end
  end
end
ActionController::Rescue.send :include,
  Cookbook::PublicRescueExtensions
```

См. также

- Рецепт 2.2 «Пакетирование приложения Rails для работы в среде Java EE»

Интеграция с Java

3.0. Введение	86
3.1. Выполнение Ruby-кода из Java-приложения	87
3.2. Вызов JRuby с помощью каркаса Bean Scripting Framework	91
3.3. Вызов JRuby с помощью технологии Java Scripting	93
3.4. Протоколирование из Ruby с помощью библиотеки Jakarta Commons Logging	95
3.5. Использование встроенных в Java средств параллельной обработки	97
3.6. Создание методов-акцессоров в духе JavaBean	100
3.7. Написание единообразного кода	101
3.8. Преобразование XML-документов с помощью библиотеки TrAX	102
3.9. Создание пула сред исполнения JRuby	104
3.10. Удаленное управление с помощью технологии JMX	106
3.11. Доступ к платформенно- зависимым библиотекам из JRuby	108

3.0. Введение

В первых двух главах мы рассматривали JRuby в основном с точки зрения Ruby. А в нескольких последующих главах мы увидим, как можно использовать JRuby в качестве инструментария для интеграции Ruby и Java. В этой главе мы изучим два основных подхода к такой интеграции. Первый ставит во главу угла то, как с помощью JRuby можно расширить функциональность Java-приложения, второй – то, как программа на Ruby может воспользоваться широчайшим набором готовых библиотек, написанных на Java. Зачастую оба подхода к интеграции комбинируются. Например, при программировании на смеси Java и Ruby было бы полезно использовать единообразную методику конфигурирования протоколов приложения (см. рецепт 3.4).

Существует три основных API, позволяющих внедрить Ruby-код в Java-приложение:

- низкоуровневый API JRuby;
- каркас Bean Scripting Framework (BSF);
- технология Java Scripting, описанная в документе JSR 223.

Этим API посвящены первые три рецепта в настоящей главе. Разница между низкоуровневым API и BSF или Java Scripting довольно очевидна: низкоуровневый API тесно привязывает Java-код к JRuby, тогда как BSF и Java Scripting представляют собой абстракции среды исполнения JRuby и на самом деле поддерживают различные сценарные языки. В общем случае использовать JRuby API имеет смысл только тогда, когда необходим строгий контроль над конфигурацией среды исполнения. Выбор же между BSF и Java Scripting диктуется в основном средой развертывания – в Java 5 поддержка BSF организована лучше, в то время как реализация Java Scripting – лишь ретроперенос с более поздней версии.

Но вне зависимости от внутренних механизмов ценность такого использования JRuby проистекает из того факта, что Ruby-код интерпретируется, а не компилируется. Это позволяет хранить Ruby-код в объекте Java типа `String` и вычислять его на этапе работы приложения. Например, генератор отчетов мог бы хранить необходимый для конкретного отчета Ruby-код в базе данных. Или можно представить себе приложение, допускающее расширение за счет написанных на Ruby подключаемых модулей, которые можно добавлять или удалять по ходу работы; Java-приложения обычно такого не умеют. Подобная техника давно уже активно применяется в играх, особенно в популярной массивно многопользовательской игре *World of Warcraft*, которую пользователи могут расширять за счет подключаемых модулей, написанный на сценарном языке Lua (при том, что ядро игры написано на C++)*.

Но за эту мощь приходится платить. Создание среды исполнения JRuby, неважно, используете вы низкоуровневый API, BSF или Java Scripting, обходится дорого, да и для ее работы требуется немало ресурсов. Расходы на создание из-

* Очевидно, при выполнении написанного пользователем кода в любом окружении необходимо проявлять изрядную осторожность.

меряются временем: на запуск JRuby уходят тысячи миллисекунд. Операционные расходы – это в основном память и, прежде всего, та, что выделяется для поколения постоянных объектов (PermGen). Первую проблему можно сгладить путем создания пула сред исполнения (см. рецепт 3.8). Вторая обычно решается за счет резервирования для PermGen достаточного объема памяти с помощью флага командной строки `-XX:PermSize`. Как правило, 256 Мб хватает. В рецепте 2.7 имеется дополнительная информация по вопросам использования памяти в JRuby.

3.1. Выполнение Ruby-кода из Java-приложения

Задача

Требуется выполнить некоторый Ruby-код из Java-приложения.

Решение

Получите экземпляр класса `org.jruby.Ruby` и вызовите метод `evalScriptlet()`. В классе `org.jruby.javasupport.JavaEmbedUtils` имеются статические фабричные методы для создания экземпляра среды исполнения JRuby. В примере 3.1 показано, как можно использовать эти классы.

Пример 3.1. Вызов Ruby из Java

```
package org.jrubycookbook.ch03;

import java.util.Collections;

import org.jruby.Ruby;
import org.jruby.javasupport.JavaEmbedUtils;

public class RubyRunner {
    public static void main(String[] args) {
        // Создать экземпляр среды исполнения JRuby. Параметром метода initialize()
        // служит список путей, которые нужно добавить в путь загрузки Ruby.
        Ruby runtime = JavaEmbedUtils.initialize(Collections.EMPTY_LIST);
        runtime.evalScriptlet("puts 'hello world'");
    }
}
```

Во время выполнения этот класс печатает классическое приветствие:

```
hello world
```



До выхода версии JRuby 1.0.3 для получения экземпляров среды исполнения JRuby использовался метод `Ruby.getDefaultInstance()`. Хотя теперь он объявлен устаревшим, время от времени его можно встретить в примерах кода.

Обсуждение

При каждом выполнении метода `JavaEmbedUtils.initialize()` создается новый экземпляр среды исполнения JRuby. Кроме того, JRuby предлагает механизм повторного использования среды исполнения в контексте одного потока Java. Для этого следует присвоить системному свойству Java `jruby.runtime.threadlocal` значение «true». В таком случае метод `JavaEmbedUtils.initialize()` создает новый экземпляр и сохраняет его в переменной `ThreadLocal`. Для доступа к этому экземпляру нужно вызвать метод `Ruby.getCurrentInstance()`. В листинге 3.2 иллюстрируется повторное использование экземпляра на примере установки и последующего считывания глобальной переменной внутри среды исполнения.

Пример 3.2. Использование текущей среды исполнения JRuby

```
package org.jrubycookbook.ch03;

import java.util.Collections;

import org.jruby.Ruby;
import org.jruby.javasupport.JavaEmbedUtils;

public class RubyRunner2 {
    public static void main(String[] args) {
        // Включить поддержку ThreadLocal
        System.setProperty("jruby.runtime.threadlocal", "true");
        // Создать экземпляр JRuby
        Ruby runtime = JavaEmbedUtils.initialize(Collections.EMPTY_LIST);
        // Выполнить Ruby-код, который создает переменную
        runtime.evalScriptlet("$message = 'hello world from JRuby'");
        runtime.evalScriptlet("$counter = 0");
        for (int i = 0; i < 5; i++) {
            outputMessage();
        }
    }

    private static void outputMessage() {
        Ruby runtime = Ruby.getCurrentInstance();
        String scriptlet = "puts \"<#{ $counter }> #{ $message }\"";
        runtime.evalScriptlet("$counter = $counter.next");
        runtime.evalScriptlet(scriptlet);
    }
}
```

Этот класс печатает такие строки:

```
<1> hello world from JRuby
<2> hello world from JRuby
<3> hello world from JRuby
<4> hello world from JRuby
<5> hello world from JRuby
```

Класс `Ruby` позволяет также порождать новые экземпляры стандартных классов `JRuby` и передавать их среде исполнения `JRuby`, которая делает их доступными выполняемым сценариям. Воспользовавшись этой техникой, метод `main()` из примера 3.2 можно было бы переписать следующим образом:

```
public static void main(String[] args) {
    System.setProperty("jruby.runtime.threadlocal", "true");
    Ruby runtime = getOrCreateInstance();
    RubyString message = runtime.newString("hello world");
    runtime.getGlobalVariables().set("$message", message);
    for (int i = 0; i < 5; i++) {
        outputMessage(i + 1);
    }
}
```

В каждой среде исполнения `JRuby` есть путь загрузки, вычисляемый исходя из значения системного свойства `jruby.home`. По умолчанию в состав пути загрузки для `JRuby 1.1` входят следующие каталоги:

1. `jruby.home/lib/ruby/site_ruby/1.8`
2. `jruby.home/lib/ruby/site_ruby`
3. `jruby.home/lib/ruby/1.8`
4. `jruby.home/lib/ruby/1.8/java`
5. `lib/ruby/1.8` (относительно текущего рабочего каталога)
6. `.` (текущий рабочий каталог)

Если исполняемый файл `jruby` используется, как было описано в главе 1, то системное свойство `jruby.home` автоматически устанавливается на основе переменной среды `JRUBY_HOME`. При написании же Java-приложений, в которых используется `JRuby`, это свойство необходимо устанавливать вручную, для чего можно задать в командной строке флаг `-D`:

```
java -cp bin:/opt/java/jruby-1.1/lib/jruby.jar\
-Djruby.home=/opt/java/jruby-1.1 org.jrubycookbook.ch03.RubyRunner
```

Это системное свойство может установить также IDE при запуске вашего приложения (на рис. 3.1 показано используемое для этой цели диалоговое окно **Eclipse Run...**) или сценарий сборки (в примере 3.3 приведен такой сценарий для системы `Ant`).

Пример 3.3. Задание системного свойства `jruby.home` в сценарии сборки для `Apache Ant`

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="run">
  <property name="jruby.home" value="/opt/java/jruby-1.1"/>

  <target name="run">
    <java classname="org.jrubycookbook.ch03.RubyRunner4" fork="true">
      <classpath>
        <pathelement location="bin"/>
        <pathelement location="\${jruby.home}/lib/jruby.jar"/>
      </classpath>
    </java>
  </target>
</project>
```

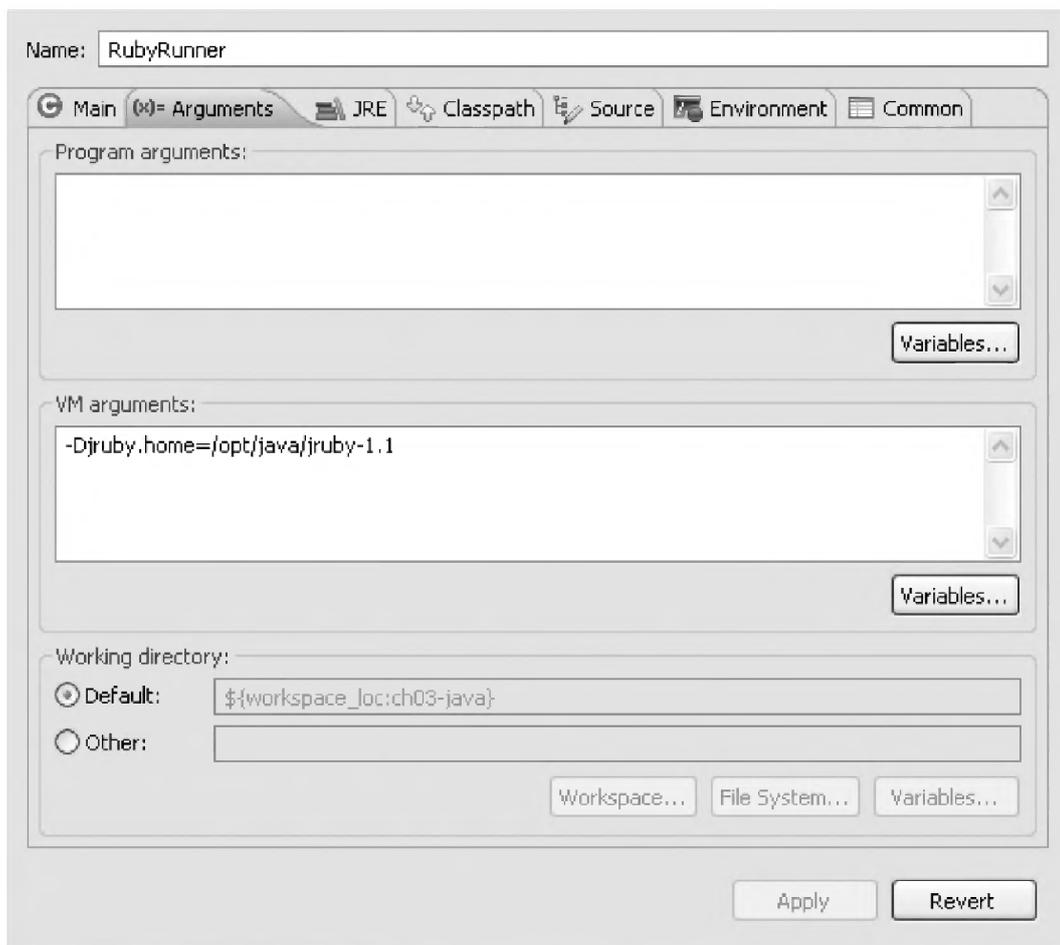


Рис. 3.1. Задание системного свойства `jruby.home` в Eclipse

```

</classpath>
<sysproperty key="jruby.home" value="${jruby.home}"/>
</java>
</target>
</project>

```

Если переменная среды `JRUBY_HOME` установлена, то вы можете получить ее значение методом `System.getenv()` и воспользоваться им для установки системного свойства `jruby.home`:

```
System.setProperty («jruby.home», System.getenv («JRUBY_HOME»));
```

Из комментариев в примере 3.1 видно, что метод `initialize()` класса `JavaEmbedUtils` принимает список путей, которые добавляются в начало описанного выше умалчиваемого пути загрузки.

См. также

- Рецепт 3.2 «Вызов JRuby с помощью каркаса Bean Scripting Framework»
- Рецепт 3.3 «Вызов JRuby с помощью технологии Java Scripting»

3.2. Вызов JRuby с помощью каркаса Bean Scripting Framework

Задача

Требуется выполнить некоторый Ruby-код из Java-приложения и при этом сохранить возможность работы с разными сценарными языками.

Решение

Воспользуйтесь каркасом Bean Scripting Framework (BSF):

1. Добавьте в путь поиска классов файл *bsf.jar*, который включен во все дистрибутивы JRuby.
2. Зарегистрируйте обработчик сценариев в среде исполнения BSF.
3. Создайте экземпляр класса `org.apache.bsf.BSFManager`.
4. Вызовите метод `eval()` или `exec()` объекта `BSFManager`.

В листинге 3.4 приведен простой пример использования JRuby с помощью BSF.

Пример 3.4. Вызов JRuby с помощью BSF

```
package org.jrubycookbook.ch03;

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;

public class RubyBSFRunner {
    public static void main(String[] args) throws BSFException {
        BSFManager.registerScriptingEngine("ruby",
            "org.jruby.javasupport.bsf.JRubyEngine", new String[] { "rb" });
        BSFManager manager = new BSFManager();
        manager.exec("ruby", "<script>", 1, 1, "puts 'hello world'");
    }
}
```

Обсуждение

Bean Scripting Framework – это каркас с открытым исходным кодом, первоначально разработанный компанией IBM, а ныне являющийся частью проекта Apache Jakarta. Он предоставляет общий API для поддержки сценарных языков внутри Java-приложений. В комплекте с BSF поставляются готовые реализации нескольких сценарных языков, а именно:

- JavaScript;
- NetRexx;
- Python;
- Tcl;
- XSLT.

Помимо этих языков, каркас Bean Scripting Framework определяет интерфейс поставщика служб (service provider interface – SPI), который позволяет подключать другие сценарные языки путем реализации интерфейса `org.apache.bsf.BSFEngine`. JRuby предоставляет реализацию этого интерфейса в виде класса `org.jruby.javasupport.bsf.JRubyEngine`. Как видно из примера 3.4, вы должны зарегистрировать этот класс в BSF, обратившись к методу `BSFManager.registerScriptingEngine()`. При регистрации этой (или любой другой) реализации обработчика необходимо сообщить BSF имя языка программирования (`ruby`) и список возможных расширений имен файлов (`rb`). `BSFManager` предоставляет два метода для вызова сценарного языка: `eval()` и `exec()`. Разница между ними в том, что `eval()` ожидает, что будет возвращено значение, а `exec()` – нет. Оба метода принимают имя вызываемого обработчика сценариев и кое-какую информацию, необходимую для извещения об ошибках и отладки: имя исходного файла (из которого загружает сценарий), номер строки и номер колонки. Последним параметром обоих методов является сам текст сценария.

В каркасе BSF имеется механизм обеспечения доступа к Java-объектам из сценариев. Делается это с помощью метода `declareBean()` класса `BSFManager`. В случае JRuby Java-объекты доступны в виде глобальных переменных в среде исполнения JRuby. Эта техника демонстрируется на примере 3.5. Обратите внимание, что имя переменной, передаваемой методу `declareBean()`, не содержит префикса `$`, тогда как в ссылке на эту переменную из Ruby-кода префикс присутствует. Префикс `$` добавляется к имени переменной автоматически. Это позволяет избежать упоминания в коде имен, специфичных только для языка Ruby, и, следовательно, упрощает использование в одном приложении сразу нескольких сценарных языков.

Пример 3.5. Использование метода `declareBean()`

```
package org.jrubycookbook.ch03;

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;

public class RubyBSFRunner2 {
    public static void main(String[] args) throws BSFException {
        BSFManager.registerScriptingEngine("ruby",
            "org.jruby.javasupport.bsf.JRubyEngine", new String[] { "rb" });
        BSFManager manager = new BSFManager();
        manager.declareBean("message", "hello world", String.class);
        manager.exec("ruby", "<script>", 1, 1, "puts $message");
    }
}
```

На сайте BSF <http://jakarta.apache.org/bsf> содержится много дополнительной документации об использовании этого каркаса.

См. также

- Рецепт 3.1 «Выполнение Ruby-кода из Java»
- Рецепт 3.3 «Вызов JRuby с помощью технологии Java Scripting»

3.3. Вызов JRuby с помощью технологии Java Scripting

Задача

Вы работаете с версией Java 6 (или более поздней) и хотите выполнить некоторый Ruby-код из Java-приложения, сохранив при этом возможность работы с разными сценарными языками.

Решение

Воспользуйтесь встроенной в Java поддержкой сценарных языков, определенной в документе JSR (Java Specification Request) 223:

1. Скачайте файл *jsr223-engines.zip* с сайта <https://scripting.dev.java.net>.
2. Распакуйте этот архив и добавьте находящийся в нем файл *jruby/build/jruby-engine.jar* в путь поиска классов.
3. Создайте экземпляр класса `javax.script.ScriptEngineManager`.
4. Вызовите метод `getEngineByName` («ruby») для получения экземпляра класса `javax.script.ScriptEngine`.
5. Вызовите метод `eval`() объекта `ScriptEngine`.

В листинге 3.6 приведен простой пример использования JRuby с помощью JSR 223 API.

Пример 3.6. Вызов JRuby с помощью `javax.script.ScriptEngineManager`

```
package org.jrubycookbook.ch03;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class Ruby223Runner {
    public static void main(String[] args) throws ScriptException {
        ScriptEngineManager scriptManager = new ScriptEngineManager();
        ScriptEngine engine = scriptManager.getEngineByName("ruby");
        engine.eval("puts 'hello world'");
    }
}
```

Обсуждение

Спецификация «JSR 223: Scripting for the Java Platform» (Сценарные языки на платформе Java) стала одним из самых долгожданных дополнений платформы Java в версии Java 6. На самом простом уровне она представляет собой стандартизованную версию API (и SPI), которые за много лет до того были реализованы в каркасе Bean Scripting Framework (BSF). Но едва не ли более важно то послание,

которое документ JSR 223 несет всему программистскому сообществу, формализуя различие между *языком* Java и *платформой* Java. Само существование этого документа подразумевает, что платформа Java предоставляет среду исполнения, пригодную для самых разных сценарных языков, в том числе Ruby/JRuby.

Сравнив примеры 3.6 и 3.4, нетрудно убедиться, что JSR 223 API проще, чем BSF API, поскольку не нужно предварительно регистрировать обработчики сценариев. В спецификации JSR определен механизм обнаружения, позволяющий автоматически находить обработчики сценариев, исходя из наличия определенного файла в каталоге *META-INF*. Обнаружив обработчик сценариев, среда исполнения запрашивает у него ряд атрибутов, в том числе общепринятые расширения имен файлов и одно или несколько имен, по которым можно распознать этот обработчик. В случае JRuby для обработчика сценариев регистрируются имена `ruby` и `jruby` и расширение `rb`. Таким образом, любое из следующих обращений вернет обработчик сценариев на языке JRuby:

```
scriptManager.getEngineByName("ruby");
scriptManager.getEngineByName("jruby");
scriptManager.getEngineByExtension("rb");
```

Аналогично низкоуровневому интерфейсу с JRuby и каркасу BSF, JSR 223 API предоставляет механизм для передачи Java-объектов в обработчик сценариев. В случае JRuby эти объекты представляются как глобальные переменные в среде исполнения JRuby. В примере 3.7 продемонстрирована эта возможность.

Пример 3.7. Создание глобальной переменной в соответствии со спецификацией JSR 223

```
package org.jrubycookbook.ch03;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class Ruby223Runner2 {

    public static void main(String[] args) throws ScriptException {
        ScriptEngineManager scriptManager = new ScriptEngineManager();
        ScriptEngine engine = scriptManager.getEngineByName("ruby");
        engine.put("message", "hello world");
        engine.eval("puts $message");
    }
}
```

Как и в BSF, имя переменной автоматически дополняется префиксом `$`, обозначающим в языке Ruby глобальную переменную.

См. также

- Рецепт 3.1 «Выполнение Ruby-кода из Java»
- Рецепт 3.2 «Вызов JRuby с помощью каркаса Bean Scripting Framework»

Так зачем использовать BSF?

Как следует из двух предыдущих рецептов, BSF и Java Scripting обеспечивают практически эквивалентную функциональность. В новых приложениях настоятельно рекомендуется применять интерфейс Java Scripting, а не BSF. Тем не менее, иногда все же имеет смысл прибегнуть к BSF.

Самая главная причина – совместимость с версией Java 5. С сайта <http://jcp.org/aboutJava/communityprocess/final/jsr223/index.html> можно скачать JAR-файл, содержащий базовые интерфейсы Java Scripting для Java 5; однако для работы обработчика сценариев на JRuby требуется Java 6*. Поэтому, если вы работаете на платформе Java 5, то воспользоваться технологией Java Scripting для взаимодействия с JRuby просто не сможете.

Второе достоинство BSF заключается в том, что обработчик сценариев JRuby для этого каркаса уже включен в дистрибутив JRuby. Это означает, что BSF гарантированно будет работать с той версией JRuby, которой вы пользуетесь. На протяжении цикла разработки, увенчавшегося версией JRuby 1.1, низкоуровневый интерфейс с JRuby несколько раз претерпевал существенные изменения, из-за чего существующий обработчик сценариев для Java Scripting перестал работать**.

И наконец, если в приложении уже используется каркас BSF и требуется лишь добавить поддержку Ruby как еще одного сценарного языка, то было бы вполне естественно и дальше работать с BSF.

3.4. Протоколирование из Ruby с помощью библиотеки Jakarta Commons Logging

Задача

Вы выполняете Ruby-код из Java-приложения, в котором уже используется библиотека Jakarta Commons Logging (JCL), и хотите, чтобы сообщения в протокол записывались в едином формате.

Решение

Воспользуйтесь классом наподобие показанного в примере 3.8 для преобразования полных имен классов Ruby в идентификаторы, напоминающие полные имена классов Java.

* Эта проблема зарегистрирована в проекте JSR223 как требующий решения вопрос, см. https://scripting.dev.java.net/issues/show_bug.cgi?id=28.

** На самом деле, во время работы над этой книгой обработчик сценариев JRuby, находящийся в файлах *jsr223-engines.zip* и *jsr223-engines.tar.gz*, не работает с JRuby 1.1. По адресу <https://scripting.dev.java.net/servlets/ProjectDocumentList?folderID=8848&expandFolder=8848&folderID=8847> имеется совместимый обработчик (версия 1.1.2).

Пример 3.8. Специальный класс-мост JRubyLogFactory

```

package org.jrubycookbook.ch03;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.jruby.RubyObject;

public class JRubyLogFactory {
    public static Log getLog(RubyObject o) {
        String rubyClassName = o.getMetaClass().getName();
        String logName = rubyClassName.replace("::", ".");
        return LogFactory.getLog(logName);
    }
}

```

Теперь вы можете ссылаться на этот класс в Ruby-коде и создавать новые объекты типа `Log`, передавая переменную `self` методу `getLog()`. При записи сообщений в протокол будет указываться имя, образованное из полного имени класса Ruby. Сценарий в примере 3.9 протоколирует сообщение с именем `Log.LogTest`.

Пример 3.9. Использование класса-моста JRubyLogFactory

```

include Java

import org.jrubycookbook.ch03.JRubyLogFactory

module Log
  class LogTest
    def initialize
      @log = JRubyLogFactory.getLog(self)
    end

    def hello
      @log.info("hello via jcl")
    end
  end
end

Log::LogTest.new.hello

```

Обсуждение

Jakarta Commons Logging – это широко распространенная библиотека Java-классов, предоставляющая единообразный интерфейс к нескольким реализациям протоколирования, в том числе: Log4J, пакет `java.util.logging`, LogKit и встроенный непосредственно в JCL класс `SimpleLog`. JCL особенно популярна среди разработчиков библиотек, поскольку позволяет работать с разными механизмами протоколирования, не привязываясь ни к одному из них на стадии компиляции. Программа обычно получает реализацию интерфейса `org.apache.commons.logging.Log`, обращаясь к одному из двух фабричных методов:

- `LogFactory.getLog(Class)`
- `LogFactory.getLog(String)`

Первый метод вызывает второй, передавая ему полное имя класса. Поскольку многие пакеты позволяют формировать иерархическую модель протоколирования, когда, например, все протоколы, имена которых начинаются с префикса `org.apache.commons`, отображаются на конкретный файл, то имя класса становится полезным источником имен протоколов.

Есть две причины для написания класса-моста, показанного в примере 3.8. Во-первых, в JRuby классы Ruby *не* являются классами Java, поэтому следующий код некорректен:

```
@log = org.apache.commons.logging.LogFactory.getLog(self.class)
```

Во-вторых, хотя имя класса можно было бы получить и так:

```
@log = org.apache.commons.logging.LogFactory.getLog(self.class.name)
```

но тогда имя протокола содержало бы двоеточия вместо ожидаемых точек, поэтому реализации протоколирования, ориентированные на использование Java-пакетов, не поймут, что имена протоколов для классов из одного и того же модуля как-то связаны между собой. Насколько серьезна эта проблема, зависит от объема протоколирования в вашей программе и от количества различных классов.

Возможно, вы обратили внимание на то, что в примерах 3.8 и 3.9 метод `getLog()` принимает экземпляр класса `org.jruby.RubyObject`. Можно было бы написать его и так, чтобы принимался экземпляр `org.jruby.RubyClass`, а затем вызывать из Ruby-кода следующим образом:

```
@log = JRubyLogFactory.getLog(self.class)
```

Однако это длиннее и шансы создать разнобой в именах протоколов выше. Смысл упражнения в том, чтобы обеспечить единообразные имена протоколов, поэтому кажется более разумным инкапсулировать логику построения имени протокола из объекта Ruby. Если понадобится нестандартное имя протокола, то всегда можно вернуться к исходному методу `LogFactory.getLog()`:

```
@log = org.apache.commons.logging.LogFactory.getLog("Some Other Log Name")
```

3.5. Использование встроенных в Java средств параллельной обработки

Задача

Требуется воспользоваться классами, входящими в пакет `java.util.concurrent`, для написания кода, который был бы безопасен относительно потоков и обеспечивал высокую производительность.

Решение

Просто сошлитесь на эти классы из Ruby-кода. Например, для создания экземпляра класса `java.util.concurrent.ConcurrentHashMap` достаточно вызвать конструктор:

```
$hash = java.util.concurrent.ConcurrentHashMap.new
```

Аналогично фабричным классом `java.util.concurrent.Executors` можно воспользоваться для создания весьма развитых пулов потоков, с которыми тем не менее просто работать. В примере 3.10 создается пул с двумя потоками, который затем используется из кода на Ruby.

Пример 3.10. Использование в коде на Ruby пула потоков из пакета `java.util.concurrent`

```
include Java

class MyLongTask
  include java.util.concurrent.Callable

  def initialize(label)
    @label = label
  end

  def call
    puts "about to sleep in task labeled #{@label}\n"
    # искусственно создаем более длительную задержку
    sleep 5
    puts "done sleeping in task labeled #{@label}\n"
    return "result of the long task labeled #{@label}\n"
  end
end

# создаем новый пул потоков
executor = java.util.concurrent.Executors::newFixedThreadPool(2)

# создаем массив для хранения ссылок на будущие результаты
future = Array.new

puts "submitting first task"
future[0] = executor.submit(MyLongTask.new("first"))

puts "submitting second task"
future[1] = executor.submit(MyLongTask.new("second"))

puts "submitting third task"
future[2] = executor.submit(MyLongTask.new("third"))

puts "All tasks have been submitted"

# вызов этого метода блокирует программу до завершения первой задачи
```

```
puts future[0].get()

# вызов этого метода блокирует программу до завершения второй задачи
puts future[1].get()

# вызов этого метода блокирует программу до завершения третьей задачи
puts future[2].get()
```

Точная последовательность строк, печатаемых этой программой, может меняться от запуска к запуску, но в любом случае вы сначала увидите, что все три задачи переданы на выполнение, а затем первые две задачи засыпают. Через какое-то время эти задачи завершатся и начнется выполнение третьей. Однако, поскольку потоков несколько, первые две задачи могут завершиться в любом порядке:

```
submitting first task
submitting second task
about to sleep in task labeled first
submitting third task
about to sleep in task labeled second
All tasks have been submitted
done sleeping in task labeled second
done sleeping in task labeled first
result of the long task labeled first
about to sleep in task labeled third
result of the long task labeled second
done sleeping in task labeled third
result of the long task labeled third
```

Обсуждение

Создавая прокси-объекты Ruby для классов Java-наборов, JRuby добавляет различные вспомогательные методы, которые присутствуют в соответствующих классах Ruby-наборов. Это позволяет трактовать Java-наборы, в том числе классы из пакетов `the java.util.concurrent`, оптимизированные для параллельной работы, как Ruby-наборы *в некоторых, но не во всех случаях*. Например, в коде на JRuby экземпляры класса `java.util.concurrent.ConcurrentHashMap` имеют метод `each`, который ведет себя точно так же, как метод `each` из Ruby-класса `Hash` (см. пример 3.11).

Пример 3.11. Использование `ConcurrentHashMap` как `Hash`

```
include Java
import java.util.concurrent.ConcurrentHashMap

states = ConcurrentHashMap.new
states['NY'] = 'New York'
states['ND'] = 'North Dakota'

states.each do |key,value|
  puts "The abbreviation for #{value} is #{key}."
end
```

Такие же методы добавлены в экземпляры классов `java.util.List` и `java.util.Set`. Однако вы не можете воспользоваться имеющимся в Ruby методом `instance_of?`, чтобы проверить являются ли эти объекты экземплярами соответствующих классов Ruby-наборов. Зато можно с помощью метода `respond_to?` проверить наличие конкретных методов:

```
irb(main):001:0> java.util.concurrent.CopyOnWriteArrayList.new.respond_to? 'each'
=> true
```

3.6. Создание методов-акцессоров в духе JavaBean

Задача

Разработчики на Ruby применяют функцию `attr_accessor` как удобный способ объявить переменную экземпляра и одновременно создать в классе соответствующие ей методы `read` и `write`. Хотелось бы иметь аналогичную функцию для лаконичного добавления в класс методов `get` и `set`, наличия которых требует спецификация JavaBean.

Решение

Начните с создания Ruby-модуля, который будет содержать новый метод. Нужную функцию можно было бы включить прямо в ваши классы, но модули позволяют писать код, обеспечивающий повторное использование без ненужного дублирования. Создайте метод `java_attr_accessor`, который принимает список символов по образцу `attr_accessor`. Символы именованы, как принято в Ruby, то есть слова разделяются подчеркиками, однако функция преобразует каждый символ в эквивалентное имя в духе JavaBean, добавляя префиксы `get` и `set` к имени, записанному в «верблюжьей нотации». В примере 3.12 показан как сам модуль, так и класс, который расширяет этот модуль и добавляет несколько переменных экземпляра с помощью метода `java_attr_accessor`.

Пример 3.12. Вспомогательный модуль для добавления акцессоров в духе JavaBean

```
module Helper
  def java_attr_accessor(*symbols)
    symbols.each { |symbol|
      camelcased = symbol.to_s.capitalize.gsub(/\_[a-zA-Z]/) {|s| s[1..1].upcase}
      module_eval( "def get#{camelcased}() @#{symbol}; end" )
      module_eval( "def set#{camelcased}(val) @#{symbol} = val; end" )
    }
  end
end
```

```
end

class Example
  extend Helper
  java_attr_accessor :title, :first_name
end

mc = Example.new
mc.setTitle('Cookbook')
mc.setFirstName("John")
```

Обсуждение

Эта вспомогательная функция оказывается очень полезна при работе с приложениями или библиотеками, в которых интенсивно используются компоненты JavaBean, например Hibernate или Spring.

3.7. Написание единообразного кода

Задача

Вы пишете программу на Ruby, в которой используются библиотеки, написанные как на Ruby, так и на Java, и хотели бы, чтобы код выглядел единообразно. Так, в следующей строке, взятой из примера 3.9, очевидно, вызывается метод Java:

```
@log = JRubyLogFactory.getLog(self)
```

Решение

Замените имена методов в «верблюжьей нотации» именами, следующими принятому в Ruby соглашению: используются только строчные буквы, а слова разделяются подчеркиками. Тогда приведенную выше строку можно было бы записать в таком виде:

```
@log = JRubyLogFactory.get_log(self)
```

JRuby автоматически производит такую трансляцию имен методов, чтобы привести вызовы методов Java и Ruby к единому стилю.

Обсуждение

JRuby не затирает существующие методы. Если уже имеется метод с именем `get_log()`, то он имеет преимущество. Если не считать этой тонкости, описанный механизм позволяет добиться более согласованного стиля кодирования.

3.8. Преобразование XML-документов с помощью библиотеки TrAX

Задача

Требуется преобразовать XML-документы с помощью XSLT-шаблонов, применяя написанную на Java библиотеку Transformation API for XML (TrAX).

Решение

Импортируйте класс `javax.xml.transform.TransformerFactory`, а также классы, которые понадобятся для ввода и вывода, обычно `javax.xml.transform.stream.StreamSource` и `javax.xml.transform.stream.StreamResult`. Если один и тот же шаблон будет использован для преобразования нескольких документов, то создайте объект `javax.xml.transform.Templates`, в котором можно сохранить откомпилированное представление шаблона. Если же требуется однократное преобразование, то достаточно будет создать объект `javax.xml.transform.Transformer`. В примере 3.13 демонстрируются оба варианта.

Пример 3.13. Использование библиотеки TrAX из JRuby

```
include Java
```

```
import javax.xml.transform.TransformerFactory
import javax.xml.transform.stream.StreamResult
import javax.xml.transform.stream.StreamSource

# Создаем экземпляр TransformerFactory
factory = TransformerFactory.new_instance

# Компилируем шаблон в объект Template
style_input = StreamSource.new("rss.xslt")
templates = factory.new_templates(style_input)

# Настраиваем источники для ввода и вывода
input = StreamSource.new("http://www.mtv.com/rss/news/news_full.jhtml")
output = StreamResult.new(java.lang.System.out)

# Запрашиваем объект Transformer у Template
transformer = templates.new_transformer

# Выполняем преобразование
transformer.transform(input, output)

# Упрощенный вариант – сразу создаем объект Transformer для данного шаблона
transformer = factory.new_transformer(style_input)
transformer.transform(input, output)
```

Обсуждение

В библиотеке TrAX есть несколько интерфейсов, которые легко реализовать на Ruby для настройки процедуры преобразования. В интерфейсе `javax.xml.transform.ErrorListener` определены обратные вызовы, к которым объект `Transformer` обращается, когда нужно вывести предупреждение или сообщение об ошибке. В примере 3.14 показана простая реализация этого интерфейса на Ruby.

Пример 3.14. Реализация интерфейса `javax.xml.transform.ErrorListener` на Ruby

```
class ErrorCounter
  attr_reader :errors
  attr_reader :warnings
  attr_reader :fatals

  def error(ex)
    @errors = 0 if (@errors == nil)
    @errors = @errors + 1
  end

  def warning(ex)
    @warnings = 0 if (@warnings == nil)
    @warnings = @warnings + 1
  end

  def fatalError(ex)
    @fatals = 0 if (@fatals == nil)
    @fatals = @fatals + 1
  end
end

# Использование класса ErrorCounter
counter = ErrorCounter.new
transformer = factory.new_transformer(style_input)
transformer.error_listener = counter
transformer.transform(input, output)

p "Errors: #{counter.errors}"
```

Еще один достойный упоминания интерфейс TrAX – `javax.xml.transform.URIResolver` – позволяет перехватывать ссылки из шаблона к внешним ресурсам. В примере 3.15 показана простая реализация `URIResolver`, в которой перехватываются относительные ссылки на конкретный шаблон. Этот метод вызывается вне зависимости от того, где встречается ссылка на `rss.xslt`: в функции `XSLT document()`, в команде `xsl:import` или в команде `xsl:include`. Для любого другого URI метод `resolve` возвращает `nil`, то есть объект `Transformer` должен разрешать URI самостоятельно.

Пример 3.15. Реализация интерфейса `javax.xml.transform.URIResolver` в Ruby

```
class MySiteResolver
  def resolve(href, base)
    if (href == 'rss.xslt')
      return StreamSource.new('http://www.mysite.com/rss.xslt')
    end
  end
end
```

3.9. Создание пула сред исполнения JRuby

Задача

Требуется выполнить небезопасный относительно потоков Ruby-код, который нуждается в монопольном контроле над средой исполнения JRuby, но вы не хотите создавать в каждом потоке отдельный экземпляр среды исполнения.

Решение

Воспользуйтесь библиотекой Jakarta Commons Pool для создания пула сред исполнения JRuby. Когда программе потребуется вызвать JRuby, она сможет получить среду исполнения из пула, а по завершении работы вернуть ее обратно. Первым делом скачайте библиотеку Jakarta Commons Pool с сайта <http://jakarta.apache.org/commons/pool/> и добавьте JAR-файл в путь поиска классов. Реализуйте подкласс `org.apache.commons.pool.BasePoolableObjectFactory`, который будет создавать экземпляры среды исполнения JRuby с помощью методов, описанных в рецепте 3.1. Затем воспользуйтесь этим объектом-фабрикой для конструирования экземпляра `org.apache.commons.pool.impl.GenericObjectPool`. В примере 3.16 приведен подкласс класса `GenericObjectPool`, применяемый для организации пула сред исполнения JRuby.

Пример 3.16. Создание пула сред исполнения JRuby

```
package org.jrubycookbook.ch03;

import java.util.Collections;
import java.util.Date;

import org.apache.commons.pool.BasePoolableObjectFactory;
import org.apache.commons.pool.impl.GenericObjectPool;
import org.jruby.Ruby;
import org.jruby.javasupport.JavaEmbedUtils;

public class JRubyRuntimePool extends GenericObjectPool {

    private static class JRubyRuntimeFactory extends BasePoolableObjectFactory {
```

```
public Object makeObject() throws Exception {
    Ruby runtime = JavaEmbedUtils.initialize(Collections.EMPTY_LIST);
    return runtime;
}

public JRubyRuntimePool() {
    super(new JRubyRuntimeFactory());
}

public Ruby borrowRuntime() throws Exception {
    return (Ruby) borrowObject();
}

public void returnRuntime(Ruby runtime) throws Exception {
    returnObject(runtime);
}

public static void main(String[] args) throws Exception {
    JRubyRuntimePool pool = new JRubyRuntimePool();
    // в пуле всегда должно быть по меньшей мере 5 экземпляров среды исполнения.
    pool.setMinIdle(5);

    // если в пуле больше 10 экземпляров среды исполнения, удалить лишние
    pool.setMaxIdle(10);

    // и не допускать, чтобы в пуле оказалось более 40 экземпляров среды исполнения
    pool.setMaxActive(40);

    // проверять соблюдение ограничений раз в минуту
    pool.setTimeBetweenEvictionRunsMillis(60000);

    // запустить приложение
}
}
```

Обсуждение

В классе `GenericObjectPool` имеется довольно много конфигурационных параметров, в том числе:

`maxActive`

Максимальное количество объектов в пуле. Может быть неограниченным. По умолчанию 8.

`maxIdle`

Максимальное количество объектов, которые могут оставаться в пуле в отсутствие запросов. Может быть неограниченным. По умолчанию 8.

`minIdle`

Минимальное количество объектов, которые должны оставаться в пуле в отсутствие запросов. Если в пуле оказывается меньше объектов (а параметр `timeBetweenEvictionRunsMills` больше нуля, см. ниже), то создаются новые экземпляры. По умолчанию 0.

`whenExhaustedAction`

Определяет, что должно происходить, когда пул пуст и поступает запрос на получение объекта. Может принимать значения `fail` (возбуждается исключение `java.util.NoSuchElementException`), `grow` (создается новый объект) или `block` (выполнение программы блокируется). По умолчанию `block`.

`timeBetweenEvictionRunsMills`

Время между последовательными запусками асинхронной задачи, которая следит за соблюдением ограничений, заданных параметрами `maxIdle` и `minIdle`. По умолчанию эта задача отключена. Поскольку на создание среды исполнения JRuby затрачивается много времени, обязательно задавайте параметры `minIdle` и `timeBetweenEvictionRunsMills`.

См. также

- Сайт Jakarta Commons Pool <http://commons.apache.org/pool/>
- <http://jruby-extras.rubyforge.org/svn/trunk/rails-integration/>, исходный текст сервлета GoldSpike

3.10. Удаленное управление с помощью технологии JMX

Задача

Требуется написать на Ruby клиент, который с помощью технологии Java Management Extensions (JMX) будет управлять выполнением удаленного Java-приложения.

Решение

Воспользуйтесь gem-пакетом `jmx4r`. Он заметно упрощает работу с JMX API. Для установки `jmx4r` выполните следующую команду:

```
jrubby -S gem install jmx4r
```

Чтобы установить соединение со службой JMX, воспользуйтесь методом класса `establish_connection`:

```
JMX::MBean.establish_connection :host => "localhost", :port => 1099
```

Чтобы найти объект MBean по имени, вызовите метод класса `find_by_name` method:

```
os = JMX::MBean.find_by_name "java.lang:type=OperatingSystem"
```

Метод `find_by_name` возвращает динамический объект, повторяющий интерфейс MBean. Для имени `java.lang:type=OperatingSystem` виртуальная машина Java возвращает Mbean-объект с несколькими атрибутами, описывающими операционную систему. К этим JMX-атрибутам можно обращаться как к обычным свойствам. Вот, например, как вывести количество имеющихся процессоров:

```
p "Running with #{os.available_processors} processors."
```



На самом деле атрибут называется `AvailableProcessors`. Но библиотека `jmx4r` преобразует это имя к виду, более привычному для Ruby.

Точно также операции JMX можно вызывать как методы. Например, для принудительного запуска сборки мусора можно написать:

```
memory = JMX::MBean.find_by_name "java.lang:type=Memory"
memory.gc
```

Обсуждение

Библиотека `jmx4r` также поддерживает запросы на поиск MBean-объектов. В примере 3.17 показано, как это делается. В данном случае JMX используется для того, чтобы узнать, какие JMS-очереди имеются в JMS-сервере Apache ActiveMQ.

Пример 3.17. Запрос на поиск MBean-объектов

```
include Java

require 'rubygems'
gem 'jmx4r'
require 'jmx4r'

JMX::MBean.establish_connection :host => "localhost", :port => 1099

queues = JMX::MBean.find_all_by_name \
  "org.apache.activemq:BrokerName=localhost,Type=Queue,*"
queues.each do |queue|
  p "Queue #{queue.name} contains #{queue.queue_size} queued messages."
end
```

Эта программа может вывести примерно такие строки:

```
Queue LogQueue contains 25 queued messages.
Queue OrderQueue contains 5 queued messages.
```

См. также

- Сайт jmx4r <http://code.google.com/p/jmx4r/>
- *Java Management Extensions* by J. Steven Perry (издательство O'Reilly)

3.11. Доступ к платформенно-зависимым библиотекам из JRuby

Задача

Требуется получить из программы на JRuby доступ к платформенно-зависимым библиотекам, например Windows DLL или разделяемым библиотекам (so-файлам) в Unix.

Решение

Java Native Access (JNA) API дает доступ к библиотекам операционной системы из Java или любого другого языка, работающего на платформе JVM, в частности JRuby. В JNA применяется динамическая архитектура, что устраняет необходимость в создании, компиляции и распространении файлов, обеспечивающих интерфейс с операционной системой (напомним, что в других технологиях, например Java Native Interface (JNI), это приходилось делать). В примере 3.18 показано, как можно получить информацию о диске, обратившись к низкоуровневой библиотеке Windows.

Пример 3.18. Использование JNA для получения информации о емкости диска в Windows

```
include Java

import com.sun.jna.ptr.LongByReference

Kernel32 = com.sun.jna.NativeLibrary.getInstance('kernel32')
GetDiskFreeSpace = Kernel32.getFunction('GetDiskFreeSpaceExA')
avail = LongByReference.new
total = LongByReference.new
total_free = LongByReference.new
num = GetDiskFreeSpace.invokeInt(["C:\\", avail, total, total_free].to_java)
puts "available: #{avail.value}"
puts "total: #{total.value}"
puts "total_free #{total_free.value}"
```

Обсуждение

Технология JNA – это прекрасное дополнение к JRuby, вместе они упрощают создание кросс-платформенных приложений, которые работают на виртуальной машине Java, но при необходимости могут обращаться к платформенно-зависимым API. Динамическая архитектура и по духу близка к языку Ruby, поскольку способствует устранению лишнего кода и ускорению разработки.

См. также

- Сайт Java Native Access <https://jna.dev.java.net>

Java в системах масштаба предприятия

4.0. Введение	112
4.1. Создание контекста JNDI ..	113
4.2. Отправка JMS-сообщений ..	115
4.3. Получение JMS-сообщений	118
4.4. Реализация компонента Enterprise JavaBean на JRuby ...	120
4.5. Определение Spring-компонентов на JRuby ..	123
4.6. Создание самообновляемых Spring-компонентов на JRuby ..	127
4.7. Встраивание Spring-компонентов, написанных на JRuby	130
4.8. Реализация Aware-интерфейсов Spring в JRuby-объектах	131
4.9. Создание MVC-контроллеров Spring с помощью JRuby	134
4.10. Hibernate и JRuby	137
4.11. Java Persistence API и JRuby	140
4.12. Выполнение вызовов по протоколу SOAP	141
4.13. Упрощение доступа к LDAP-каталогу	143

4.0. Введение

Во введении к главе 1 мы уже говорили, что одной из самых сильных сторон JRuby является способность к бесшовной интеграции с многочисленными написанными на Java библиотеками. Это особенно важно в приложениях масштаба предприятия, где язык Java занимает прочные позиции. Своим успехом Java в небольшой степени обязан стандартизированной платформе Java Enterprise Edition (Java EE, прежнее название J2EE). Но и те платформы, которые не подпадают под стандарты Java, не менее важны. В этой главе мы рассмотрим две из них: каркасы Spring и Hibernate*. Неважно, стандартизована некоторая технология или нет, все платформы Java масштаба предприятия призваны помочь разработчикам сосредоточиться на решении задач бизнеса и на логике презентации, а не на вопросах инфраструктуры и интеграции.

Мы начнем эту главу с рецепта, касающегося обращения из Ruby к интерфейсу корпоративной службы каталогов Java Naming and Directory Interface (JNDI). Как следует из самого названия, JNDI представляет собой API доступа к службе каталогов. JNDI предоставляет разработчикам унифицированный интерфейс, который может распространяться на различные службы и типы служб. Внутри сервера приложений на платформе Java EE интерфейс JNDI используется прикладными программами для обнаружения ресурсов, управляемых сервером. Это могут быть источники данных (тема, обсуждавшаяся в главе 2), компоненты Enterprise JavaBean (EJB), объекты службы сообщений Java Messaging Service (JMS) и многие другие ресурсы. В документации по вашему серверу приложений должна быть приведена подробная информация о том, какие ресурсы имеются и как можно добавить новые ресурсы. JNDI можно использовать также для доступа к внешним службам. Во втором и третьем рецептах мы с помощью JNDI установим соединение с удаленным JMS-брокером, в роли которого будет выступать сервер Apache ActiveMQ, и тем самым сможем посылать и получать JMS-сообщения. В следующем рецепте мы воспользуемся JNDI, чтобы соединиться с LDAP-сервером (Lightweight Directory Access Protocol – облегченный протокол службы каталогов) и с помощью JRuby упростим работу с JNDI API.

Вслед за JMS мы рассмотрим реализацию технологии Enterprise JavaBean (EJB). Благодаря поддержке аннотированной конфигурации, которая была включена в версию EJB 3, создание EJB-компонентов значительно упростилось, однако в JRuby аннотации не поддерживаются, поэтому для реализации EJB-компонентов придется написать кое-какой связующий код. Хотя на первый взгляд JRuby и EJB могут показаться странной парочкой, при ближайшем рассмотрении оказывается, что модель EJB при использовании совместно с JRuby может дать заметные преимущества из-за наличия пула экземпляров, организуемого контейнерами Java EE. Все контейнеры поддерживают пул экземпляров и гарантируют, что у каждого экземпляра в любой момент времени имеется не более одного кли-

* Было время, когда сочетание Spring и Hibernate называлось J3EE, но, похоже, в последние годы этот термин вышел из употребления.

ента. Это означает, что при написании EJB-объекта, не важно, на Java или на Ruby, вам не нужно заботиться о параллелизме; за вас все сделает контейнер. Известно, что многие написанные на Ruby библиотеки и, прежде всего ActiveRecord и Rails, страдают от проблем, связанных с параллелизмом; применение EJB устраняет необходимость в создании специализированных пулов экземпляров. Этот вопрос обсуждается в рецепте 3.9 и при рассмотрении Rails в главе 2.

В нескольких рецептах из этой главы обсуждается интеграция JRuby с каркасом Spring Framework, который иногда называют просто Spring. По существу Spring – это платформа для создания приложений путем определения составляющих их компонентов (в виде Java-классов) и взаимосвязей между ними. Это называется внедрением зависимости (Dependency Injection – DI) и/или инверсией управления (Inversion of Control – IoC)*. На фундаменте этой базовой платформы Spring предоставляет также поддержку для аспектно-ориентированного программирования (Aspect-Oriented Programming – AOP), транзакций, аутентификации и авторизации, удаленных вызовов, применения архитектуры модель-вид-контроллер (MVC) к разработке веб-приложений и т.д. Начиная с версии 2.0, Spring также поддерживает динамические языки, в том числе и JRuby. Эта поддержка, которая рассматривается в нескольких рецептах, позволяет прозрачно интегрировать объекты, определенные в JRuby, с объектами, определенными в Java (или в каком-то другом динамическом языке).

В этой главе мы рассмотрим также каркас объектно-реляционного отображения (Object-Relational Mapping – ORM) Hibernate и технологию обеспечения устойчивых объектов Java Persistence API (JPA)**. Благодаря интеграции JRuby с Java использование этих каркасов из JRuby не представляет особых сложностей; по большей части Hibernate и JPA «просто работают». Поэтому в рецептах речь пойдет об использовании JRuby как средства повышения продуктивности при работе с этими API.

4.1. Создание контекста JNDI

Задача

Требуется создать объект JNDI Context, позволяющий установить соединение с LDAP-сервером или JMS-брокером.

Решение

Создайте хэш Ruby, включив в него свойства, которые должны присутствовать в окружении, а затем передайте этот хэш конструктору объекта `java.naming`.

* Строго говоря, внедрение зависимости – частный случай применения паттерна «Инверсия управления», но на практике эти термины часто употребляют как синонимы.

** Которая во многих отношениях есть не что иное, как стандартизированный вариант Hibernate.

InitialContext, предварительно обернув его в объект `java.util.Hashtable`. Так, в примере 4.1 создается объект `JNDIContext` для обращения к публичному LDAP-серверу Мичиганского университета.

Пример 4.1. Создание контекста JNDI

```
include Java

import java.util.Hashtable
import javax.naming.InitialContext
import javax.naming.Context

env = {Context::INITIAL_CONTEXT_FACTORY => "com.sun.jndi.ldap.LdapCtxFactory",
      Context::PROVIDER_URL => "ldap://ldap.itd.umich.edu:389" }
ctx = InitialContext.new(Hashtable.new(env))
```

Обсуждение

Хотя JRuby автоматически преобразует хэши Ruby в объекты Java, реализующие интерфейс `java.util.Map`, объекты класса `InitialContext` конфигурируются с помощью экземпляров `Hashtable`. Поэтому хэш необходимо обернуть в объект типа `Hashtable`.

Свойства, используемые для создания объекта `InitialContext`, можно также поместить в файл *jndi.properties*, который должен быть представлен в пути поиска классов Java. Для примера 4.1 этот файл должен был бы состоять из таких строк:

```
java.naming.factory.initial = com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url = ldap://ldap.itd.umich.edu:389
```

При таком способе конфигурирования объект `InitialContext` можно создать с помощью конструктора без аргументов:

```
ctx = InitialContext.new
```

Вне зависимости от способа конфигурирования значением свойства `java.naming.factory.initial` должен быть класс, представленный в пути поиска классов. В рецепте 1.7 отмечалось, что JRuby умеет добавлять JAR-файлы в путь поиска классов динамически. Однако этот механизм не применяется к такого рода фабричным классам. Объясняется это тем, что динамически добавленные JAR-файлы видны только из кода на Ruby. Например, в следующем рецепте в свойство `java.naming.factory.initial` записывается значение `org.apache.activemq.jndi.ActiveMQInitialContextFactory`. Если вы попытаетесь добавить этот класс (и его зависимости) в путь поиска классов в JRuby, то будет возбуждено исключение `javax.naming.NoInitialContextException`:

```
$ jirb
irb(main):001:0> include Java
irb(main):002:0>
irb(main):003:0* require '/opt/java/libs/geronimo-j2ee-management_1.0_spec-1.0.jar'
irb(main):004:0> require '/opt/java/libs/geronimo-jms_1.1_spec-1.1.1.jar'
```

```
irb(main):005:0> require '/opt/java/libs/activemq-core-5.1.0.jar'
irb(main):006:0>
irb(main):007:0* import java.util.Hashtable
irb(main):008:0> import javax.naming.InitialContext
irb(main):009:0> import javax.naming.Context
irb(main):010:0>
irb(main):011:0* env = { Context::INITIAL_CONTEXT_FACTORY =>
irb(main):012:1*     "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
irb(main):013:1*     Context::PROVIDER_URL =>
irb(main):014:1*     "tcp://localhost:61616" }
irb(main):015:0> ctx = InitialContext.new(Hashtable.new(env))
NativeException: javax.naming.NoInitialContextException: Cannot
    org.apache.activemq.jndi.ActiveMQInitialContextFactory
```

Эту проблему можно решить – создать экземпляр класса напрямую:

```
import org.apache.activemq.jndi.ActiveMQInitialContextFactory

env = { Context::PROVIDER_URL => "tcp://localhost:61616" }
ctx = ActiveMQInitialContextFactory.new.get_initial_context(Hashtable.new(env))
```

См. также

- Сайт JNDI <http://java.sun.com/products/jndi/>

4.2. Отправка JMS-сообщений

Задача

Требуется отправить из приложения сообщения брокеру Java Messaging Service (JMS).

Решение

Добавьте необходимые JAR-файлы в путь поиска классов. Создайте объект `javax.naming.InitialContext`, как описано в рецепте 4.1. Параметры окружения должны быть описаны в документации, поставляемой вместе с JMS-брокером. Например, для соединения с экземпляром Apache ActiveMQ следует задать такие свойства:

```
env = { Context::INITIAL_CONTEXT_FACTORY =>
    "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
    Context::PROVIDER_URL =>
    "tcp://localhost:61616" }
```

После создания объекта `InitialContext` поищем объект `JMS ConnectionFactory` и получателя:

```
connection_factory = ctx.lookup("ConnectionFactory")
```

```
destination = ctx.lookup("dynamicQueues/output.queue")
```

Все остальное – это трафаретный код, характерный для работы с JMS, который можно инкапсулировать в класс Ruby, как показано в примере 4.2.

Пример 4.2. Отправка JMS-сообщения из Ruby

```
include Java

import java.util.Hashtable
import javax.naming.InitialContext
import javax.naming.Context
import javax.jms.Session

class JmsSender

  def initialize(environment)
    @context = InitialContext.new(Hashtable.new(environment))
    @connection_factory = @context.lookup("ConnectionFactory")
  end

  def send_text_message(destination_name, message_text)
    destination = @context.lookup(destination_name)
    connection = @connection_factory.create_connection()
    session = connection.create_session(false, Session::AUTO_ACKNOWLEDGE)
    producer = session.create_producer(destination)
    message = session.create_text_message
    message.text = message_text
    producer.send(message)
    session.close
  end
end

env = { Context::INITIAL_CONTEXT_FACTORY =>
        "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
        Context::PROVIDER_URL =>
        "tcp://localhost:61616" }
sender = JmsSender.new(env)

sender.send_text_message("dynamicQueues/output.queue", "hello to JMS from Ruby")
```

Отправленное сообщение должно появиться в административном веб-интерфейсе ActiveMQ, как показано на рис. 4.1.

Обсуждение

В рецепте 4.1 отмечалось, что для того чтобы можно было создать объект `javax.naming.InitialContext` со свойством `org.apache.activemq.jndi.ActiveMQInitialContextFactory`, JAR-файлы, составляющие сервер ActiveMQ, должны находиться в пути поиска классов в момент запуска приложения; динамическое добавление средствами JRuby здесь не годится.

The screenshot shows the ActiveMQ web interface. At the top, there is a navigation bar with links: Home | Queues | Topics | Subscribers | Send | Support. The main content area is divided into several sections:

- Headers:** A table listing message metadata:

Message ID	ID:ny-edelsonj-3682-1208660802671-0:0:1:1:1
Destination	queue://output.queue
Correlation ID	
Group	
Sequence	0
Expiration	0
Persistence	Persistent
Priority	4
Redelivered	false
Reply To	
Timestamp	1208660803062
Type	
- Properties:** A section for additional message properties.
- Queue Views:** A section with links for Graph and XML.
- Useful Links:** A section with links for Documentation, FAQ, Downloads, and Forums.
- Message Details:** A section showing the message content: "hello to JMS from Ruby".

At the bottom, there is a copyright notice: Copyright 2005-2007 The Apache Software Foundation. (printable version)

Рис. 4.1. Сообщение, отправленное JRuby, видно веб-интерфейсе ActiveMQ

В JMS API определено пять типов сообщений:

Stream

Сообщения этого типа описываются интерфейсом `javax.jms.StreamMessage` и содержат одно или несколько примитивных значений или Java-объектов, следующих друг за другом.

Map

Сообщения этого типа описываются интерфейсом `javax.jms.MapMessage` и состоят из одной или нескольких пар имя-значение. Имена представляются Java-объектами типа `String`, а значения могут быть произвольными примитивами или объектами.

Text

Сообщения этого типа описываются интерфейсом `javax.jms.TextMessage` и состоят из одного объекта типа `String`.

Object

Сообщения этого типа описываются интерфейсом `javax.jms.ObjectMessage` и содержат один Java-объект, который реализует интерфейс `Serializable`.

Bytes

Сообщения этого типа описываются интерфейсом `javax.jms.BytesMessage` и предназначены главным образом для поддержки других систем передачи сообщений (отличных от JMS).

Все эти типы сообщений можно использовать и в JRuby, но необходимо принимать специальные меры при отправке объектов, поскольку объекты JRuby не могут быть корректно сериализованы с помощью механизмов Java. Это относится даже к тому случаю, когда получателем сообщения является приложение, написанное на JRuby. Давайте, к примеру, добавим метод `send_object_message` в класс из примера 4.2:

```
def send_object_message(destination_name, message_object)
  destination = @context.lookup(destination_name)
  connection = @connection_factory.create_connection()
  session = connection.create_session(false, Session::AUTO_ACKNOWLEDGE)
  producer = session.create_producer(destination)
  message = session.create_object_message message_object
  producer.send(message)
  session.close
end
```

Если вызвать этот метод, передав ему массив Ruby

```
arr = ["one", "two", "three"]
send_object_message("dynamicQueues/output.queue", arr)
```

то в момент получения сообщения будет возбуждено исключение, так как массив сериализован в виде объекта `org.jruby.RubyArray`. Для решения проблемы необходимо создать из исходного массива Ruby объект типа `java.util.ArrayList`:

```
arr = ["one", "two", «three"]
send_object_message("dynamicQueues/output.queue", java.util.ArrayList.new(arr))
```

4.3. Получение JMS-сообщений

Задача

Приложению требуется получать сообщения от JMS-брокера.

Решение

Начальные действия такие же, как при отправке JMS-сообщения: создать объект `JNDI InitialContext` и найти в контексте JNDI фабрику соединений `ConnectionFactory` и получателя. С помощью `ConnectionFactory` создаем объект `Connection`, а от него получаем объект `Session`. Последним можно воспользоваться для создания получателя сообщений `MessageConsumer`. У объекта `MessageConsumer`

есть два перегруженных метода `receive` для получения сообщений. При вызове без параметров метод `receive` блокирует программу до поступления сообщения. Если же передан один (числовой) параметр, то программа блокируется до тех пор, пока либо не поступит сообщение, либо не пройдет указанное количество миллисекунд.

В примере 4.3 приведен типичный код для получения сообщения. Если полученное сообщение содержит текст, то он печатается.

Пример 4.3. Получение JMS-сообщения

```
include Java

import java.util.Hashtable
import javax.naming.InitialContext
import javax.naming.Context
import javax.jms.Session

env = { Context::INITIAL_CONTEXT_FACTORY =>
        "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
        Context::PROVIDER_URL =>
        "tcp://localhost:61616" }
context = InitialContext.new(Hashtable.new(env))
connection_factory = context.lookup("ConnectionFactory")

destination = context.lookup("dynamicQueues/output.queue")
connection = connection_factory.create_connection()
session = connection.create_session(false, Session::AUTO_ACKNOWLEDGE)
consumer = session.create_consumer(destination)

connection.start

message = consumer.receive
if (message.respond_to? 'text')
  p "message = #{message.text}"
else
  p "message isn't a text message"
end

connection.stop
session.close
```

Обсуждение

Обратите внимание, что в примере 4.3 мы сначала устанавливаем соединение, а только потом получаем сообщение. Для получения сообщения активное соединение необходимо, тогда как для отправки оно ни к чему.

4.4. Реализация компонента Enterprise JavaBean на JRuby

Задача

Требуется инкапсулировать Ruby-код в виде компонента Enterprise JavaBean (EJB), чтобы было проще интегрировать его с другими EJB-компонентами и сервлетами, а также иметь доступ к таким предоставляемым контейнером EJB службам, как пул соединений, обеспечение безопасности и транзакционная целостность.

Решение

Создайте интерфейс и реализующий его класс EJB-компонента. Простой интерфейс EJB, снабженный аннотацией `@Local`, приведен в примере 4.4.

Пример 4.4. Локальный интерфейс EJB

```
package org.jrubycookbook.j2ee.ejb;

import javax.ejb.Local;

@Local
public interface Reverser {
    public String reverse(String string);
}
```

В реализующем классе заведите метод инициализации и в нем создайте экземпляр среды исполнения JRuby. Для этого можно применить любой из подходов, описанных в главе 3. Снабдите метод инициализации аннотацией `@PostConstruct`. Затем во всех бизнес-методах (тех, что определены в интерфейсе EJB) оберните аргументы в объекты Ruby, добавьте их в среду исполнения и выполните содержательный Ruby-код. В примере 4.5 показан класс EJB-компонента, написанный на JRuby. В данном случае реализующий код встроен в определение интерфейса, но с тем же успехом его можно было бы вынести во внешний файл.

Пример 4.5. EJB-компонент на JRuby

```
package org.jrubycookbook.j2ee.ejb;

import javax.annotation.PostConstruct;
import javax.ejb.Stateless;
import org.jruby.Ruby;
import org.jruby.RubyString;
import org.jruby.javasupport.JavaEmbedUtils;

@Stateless
```

```
public class ReverserBean implements Reverser {

    private Ruby ruby;

    @PostConstruct
    public void init() {
        ruby = JavaEmbedUtils.initialize(Collections.EMPTY_LIST);
    }

    public String reverse(String string) {
        ruby.getGlobalVariables().set("$message", ruby.newString(string));
        return ruby.evalScriptlet("$message.reverse").asJavaString();
    }
}
```

К этому EJB-компоненту теперь могут обращаться сервлеты и другие EJB-компоненты, находящиеся в том же контейнере. Один такой сервлет приведен в примере 4.6.

Пример 4.6. Сервлет, обращающийся к EJB-компоненту, написанному на JRuby

```
package org.jrubycookbook.j2ee.servlet;

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.jrubycookbook.j2ee.ejb.Reverser;

public class ReverseServlet extends HttpServlet {

    @EJB
    private Reverser reverser;

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        String result = reverser.reverse(req.getParameter("word"));
        resp.getWriter().println(result);
    }
}
```

Можно было бы также определить интерфейс удаленного вызова и снабдить его аннотацией `@Remote`, тогда к этому EJB-компоненту можно было бы обращаться с помощью технологии Remote Method Invocation (RMI).

Обсуждение

Как легко видеть, класс из примера 4.5 – не более чем мост между контейнером EJB-компонентов и средой исполнения JRuby. Необходим он главным образом потому, что JRuby пока не поддерживает аннотаций Java. Если в будущем такая поддержка будет добавлена, то этот класс (а, возможно, и сам интерфейс) можно будет исключить. Кроме того, весьма вероятно, что поставщики контейнеров Java EE добавят прямую поддержку написанных на JRuby EJB-компонентов, если это будет востребовано.

Класс в примере 4.5 – это компонент-сеанс без сохранения состояния (stateless session bean – SLSB), но точно такая же техника применима и к компонентам-сеансам с сохранением состояния (stateful session beans – SFSB), и к компонентам, управляемым сообщениями (message-driven beans – MDB). Кроме того, можно без труда раскрыть такой EJB-компонент через интерфейс веб-службы, добавив дополнительные аннотации, как показано в примере 4.7.

Пример 4.7. EJB-компонент с аннотациями веб-службы, написанный на JRuby

```
package org.jrubycookbook.j2ee.ejb;

import javax.ws.WebMethod;
import javax.ws.WebService;

// прочие директивы import из рецепта 4.5

@WebService(targetNamespace = "http://jrubycookbook.org/ejb")
@Stateless
public class ReverserBean implements Reverser {

    private Ruby ruby;

// метод init() из примера 4.5

    @WebMethod
    public String reverse(String string) {
        RubyString message = ruby.newString(string);
        ruby.getGlobalVariables().set("$message", message);
        return ruby.evalScriptlet("$message.reverse").asJavaString();
    }
}
```

На рис. 4.2 показан результат тестирования этой веб-службы с помощью предназначенной специально для этой цели страницы, включенной в состав дистрибутива Sun Java System Application Server.

См. также

- Рецепт 3.9 «Создание пула сред исполнения JRuby»

reverse Method invocation

Method parameter(s)

Type	Value
java.lang.String	JRuby Cookbook

Method returned

java.lang.String : "koobkooC ybuRJ"

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:reverse xmlns:ns2="http://jrubbycookbook.org/ejb">
      <arg0>JRuby Cookbook</arg0>
    </ns2:reverse>
  </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:reverseResponse xmlns:ns2="http://jrubbycookbook.org/ejb">
      <return>koobkooC ybuRJ</return>
    </ns2:reverseResponse>
  </S:Body>
</S:Envelope>
```

Рис. 4.2. Тестирование написанного на JRuby EJB-компонента, раскрываемого через веб-службу

4.5. Определение Spring-компонентов на JRuby

Задача

Вы используете каркас Spring Framework в качестве контейнера с внедренной зависимостью (Dependency Injection – DI) и хотели бы некоторые компоненты написать на JRuby.

Решение

Определите на Java интерфейс, определив в нем методы, которые будут реализованы в Ruby-классе. В XML-файл, описывающий конфигурацию Spring, включите элемент `jrubby` в пространстве имен `lang` и определите в нем компонент, указав интерфейс и местоположение Ruby-сценария. Написанные на JRuby компоненты можно конфигурировать также с помощью элемента `lang:property`. Определение простого компонента на JRuby приведено в примере 4.8.

Пример 4.8. Определение простого Spring-компонента на JRuby

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:lang="http://www.springframework.org/schema/lang"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans.xsd
7         http://www.springframework.org/schema/lang
8         http://www.springframework.org/schema/lang/spring-lang.xsd">
9
10 <lang:jruby id="rubyListener"
11     script-interfaces="org.jrubycookbook.ch04.Listener"
12     script-source="classpath:org/jrubycookbook/ch04/ruby_listener.rb">
13     <lang:property name="prefix" value="(from Ruby) " />
14 </lang:jruby>
15
16 </beans>

```

Здесь строки 2–8 – это стандартная конфигурация Spring, они необходимы для определения пространств имен: умалчиваемого и lang. В строках 10–14 содержится собственно определение компонента, в том числе свойства с именем prefix. Интерфейс этого компонента определен в примере 4.9, а его реализация на Ruby – в примере 4.10.

Пример 4.9. Простой интерфейс Spring-компонента

```

package org.jrubycookbook.ch04;

public interface Listener {
    public void receiveMessage(String message);
}

```

Пример 4.10. Ruby-сценарий, на который ссылается конфигурационный файл Spring

```

class RubyListener
  # метод установки свойства prefix
  def setPrefix(p)
    @prefix = p
  end

  # реализация интерфейса Listener
  def receiveMessage(s)
    puts "#{@prefix}Got Message: #{s}"
  end
end

RubyListener.new

```

Отметим, что для того чтобы Spring мог установить свойство prefix, должен быть определен метод setPrefix(). Если бы вы писали код, как принято в Ruby, то, наверное, назвали бы этот метод prefix= и сгенерировали бы его автоматически

с помощью функции `attr_accessor` или `attr_writer`. Но, поскольку Spring придерживается спецификации `JavaBean`, то ожидает, что метод будет называться `setPrefix()`.

Чтобы JRuby мог использовать каркас Spring, путь поиска классов должен содержать следующие JAR-файлы (все они входят в дистрибутив Spring)*:

- *spring.jar*
- *asm-2.2.3.jar*
- *backport-util-concurrent.jar*
- *cglib-nodep-2.1_3.jar*
- *commons-logging.jar*
- *jruby.jar*

Во время работы над этой книгой в каркас Spring была включена только поддержка версии JRuby 1.0, с версией 1.1 он был несовместим.

Обсуждение

Из динамических языков Spring в настоящее время поддерживает JRuby, а также Groovy и BeanShell. Реализовано это путем создания динамического прокси-объекта, который реализует интерфейсы, перечисленные в атрибуте `script-interfaces`. Этот объект получает все вызовы методов и делегирует их объекту, созданному в файле сценария, который прописан в атрибуте `script-source`. Для записи атрибута `script-source` используется стандартный синтаксис доступа к ресурсам, принятый в Spring. В примере 4.8 мы ссылаемся на исходный файл Ruby с помощью схемы `classpath`, но с тем же успехом это может быть путь в файловой системе, URL ресурса или, если это имеет смысл, ресурс в контексте сервлета.

Spring-компонентам, написанным на динамическом языке, необходимы некоторые возможности, определенные в интерфейсе `ApplicationContext`, поэтому наивная реализация `BeanFactory`, которая приведена в примере 4.11, работать не будет.

Пример 4.11. JRuby-код внутри объекта `BeanFactory` не работает

```
package org.jrubycookbook.ch04;
```

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class ListenerBootstrap {
    public static void main(String[] args) {
        ClassPathResource config =
            new ClassPathResource("org/jrubycookbook/ch04/listener_beans.xml");
        XmlBeanFactory ctx = new XmlBeanFactory(config);

        Listener listener = (Listener) ctx.getBean("rubyListener");
```

* Эта информация относится к версии Spring 2.5.1. Для других версий см. документацию.

```

        listener.receiveMessage("Hello");
    }
}

```

Вместо этого придется воспользоваться какой-нибудь реализацией интерфейса `ApplicationContext`, например, классом `ClassPathXmlApplicationContext`, как показано в примере 4.12.

Пример 4.12. Использование JRuby внутри Application Context

```

package org.jrubycookbook.ch04;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ListenerBootstrap {
    public static void main(String[] args) {
        String config = "org/jrubycookbook/ch04/listener_beans.xml";
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext(config);
        Listener listener = (Listener) ctx.getBean("rubyListener");
        listener.receiveMessage("Hello");
    }
}

```

Возвращаясь к примеру 4.10, мы видим, что в этом сценарии определяется Ruby-класс `RubyListener` и возвращается новый экземпляр этого класса. В данном случае это необязательно; Spring сумел бы понять, что в этом сценарии определен класс, и самостоятельно сгенерировал бы экземпляр этого класса, если его еще нет. Однако лучше все-таки включать эту команду, поскольку иногда Spring создает экземпляр не того класса, что нужно. Так может случиться, например, если в упоминаемом Ruby-файле содержится несколько определений классов, как в примере 4.13.

Пример 4.13. Ruby-сценарий, на котором Spring ошибается

```

class RubyListener
    def setPrefix(p)
        @prefix = p
    end

    # реализация интерфейса Listener
    def receiveMessage(s)
        puts "#{@prefix}Got Message: #{s}"
    end
end

class OtherRubyListener < RubyListener
    # реализация интерфейса Listener
    def receiveMessage(s)
        puts "#{@prefix}Got A Message: #{s}"
    end
end

```

Поэтому проще всегда включать команду `new` в последней строке Ruby-сценария, чтобы Spring гарантированно обращался к правильному объекту.

См. также

- Сайт Spring Framework <http://www.springframework.org/>

4.6. Создание самообновляемых Spring-компонентов на JRuby

Задача

Требуется, чтобы находящиеся в контейнере Spring компоненты автоматически перезагружались при изменении их определения.

Решение

Включите атрибут `refresh-check-delay` в элемент `lang:jruby` конфигурационного XML-файла Spring. Этот атрибут заставляет Spring следить за ресурсом, упомянутым в атрибуте `script-source`. Его значение определяет, с какой периодичностью (в миллисекундах) проверять, изменился ли ресурс.

Можно вместо этого задать в пространстве имен `lang` элемент `defaults`, содержащий значение по умолчанию для атрибута `refresh-check-delay`. Так, чтобы установить секундный интервал между проверками всех написанных на динамических языках компонентов в контексте `ApplicationContext`, включите в конфигурационный файл такую строчку:

```
<lang:defaults refresh-check-delay=»1000"/>
```

Обсуждение

Один из простейших способов продемонстрировать описываемую функциональность состоит в том, чтобы воспользоваться встроенной в Spring поддержкой объектов `Java Timer`. В конфигурационном файле Spring из примера 4.14 определен тот же самый компонент `rubyListener`, что и в примере 4.10, а, кроме того, добавлена реализация интерфейса `java.util.TimerTask`, которая выводит текущее время. Сюда же включен инфраструктурный код, необходимый для того, чтобы Spring вызывал это задание каждые пять секунд.

Пример 4.14. Самообновляемый Spring-компонент, вызываемый из задания `TimerTask`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:lang="http://www.springframework.org/schema/lang"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/lang
    http://www.springframework.org/schema/lang/spring-lang.xsd">

<lang:defaults refresh-check-delay="1000" />

<lang:jruby id="rubyListener"
    script-interfaces="org.jrubycookbook.ch04.Listener "
    script-source="classpath:org/jrubycookbook/ch04/ruby_listener.rb">
    <lang:property name="prefix" value="(from Timer) " />
</lang:jruby>

<bean id="sendDateTask" class="org.jrubycookbook.ch04.SendDateTask">
    <property name="listener" ref="rubyListener"/>
</bean>

<bean id="scheduledTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="period" value="5000" />
    <property name="timerTask" ref="sendDateTask" />
</bean>

<bean id="timerFactory"
    class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
    <list>
        <ref bean="scheduledTask" />
    </list>
    </property>
</bean>
</beans>
```

Класс `SendDateTask`, показанный в примере 4.15, просто форматирует текущую дату и передает ее внедренной реализации интерфейса `Listener`.

Пример 4.15. Класс `SendDateTask`

```
package org.jrubycookbook.ch04;

import java.util.Date;
import java.util.TimerTask;

public class SendDateTask extends TimerTask {

    private Listener listener;

    public void setListener(Listener listener) {
```

```

    this.listener = listener;
  }

  public void run() {
    listener.receiveMessage(String.format("%tT", new Date()));
  }
}

```

Имея эти два класса, мы можем создать контекст `ApplicationContext`, как показано в примере 4.16. После того как контекст будет запущен, любые изменения файла `ruby_listener.rb` станут видны при каждом выполнении задания `SendDateTask`.

Пример 4.16. Запуск контекста `ApplicationContext` с поддержкой таймера

```

package org.jrubycookbook.ch04;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TimedBootstrap {
  public static void main(String[] args) {
    String config = "org/jrubycookbook/ch04/timer_beans.xml";
    ClassPathXmlApplicationContext ctx =
      new ClassPathXmlApplicationContext(config);
  }
}

```

Например, можно было бы изменить порядок сообщений в классе `RubyListener` на противоположный:

```

class RubyListener
  def setPrefix(p)
    @prefix = p
  end

  # реализация интерфейса Listener
  def receiveMessage(s)
    puts "#{@prefix}Got Message: #{s}".reverse
  end
end

RubyListener.new

```

Если произвести это изменение во время работы `ApplicationContext`, то будут напечатаны такие строки:

```

(from Timer) Got Message: 21:21:48
(from Timer) Got Message: 21:21:53
85:12:12 :egasseM toG )remiT morf(

```

4.7. Встраивание Spring-компонентов, написанных на JRuby

Задача

Требуется написать код Spring-компонентов на JRuby прямо в конфигурационном XML-файле, а не во внешнем файле.

Решение

Вместо того чтобы задавать местоположение ресурса в атрибуте `script-source`, вы можете поместить код на JRuby внутри элемента `inline-script` в пространстве имен `lang`, как показано в примере 4.17.

Пример 4.17. JRuby-сценарий внутри элемента `inline-script`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:lang="http://www.springframework.org/schema/lang"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/lang
    http://www.springframework.org/schema/lang/spring-lang.xsd">

  <lang:jruby id="rubyListener"
    script-interfaces="org.jrubycookbook.ch04.Listener">
    <lang:inline-script><![CDATA[
class RubyListener
  def setPrefix(p)
    @prefix = p
  end

  # реализация интерфейса Listener
  def receiveMessage(s)
    puts "#{@prefix}Got Message: #{s}"
  end
end

RubyListener.new
  ]]></lang:inline-script>
    <lang:property name="prefix" value="(from Ruby) " />
  </lang:jruby>
</beans>
```

4.8. Реализация Aware-интерфейсов Spring в JRuby-объектах

Задача

Контекст `ApplicationContext` содержит написанные на JRuby компоненты, которые должны реализовать какой-то из Aware-интерфейсов, например `org.springframework.context.ApplicationContextAware`.

Решение

Включите реализации методов соответствующего интерфейса в JRuby-класс и добавьте имя интерфейса в атрибут `script-interfaces`.

Обсуждение

В каркасе Spring есть ряд интерфейсов, предназначенных для того, чтобы компонент мог узнать о своем окружении. Обычно в таких интерфейсах определен всего один метод, который вызывается контейнером на этапе инициализации. Ниже перечислены некоторые интерфейсы этого типа:

`org.springframework.context.ApplicationContextAware`

Методу `setApplicationContext()` передается экземпляр контекста `ApplicationContext`, содержащего этот компонент.

`org.springframework.beans.factory.BeanFactoryAware`

Методу `setBeanFactory()` передается имя этого компонента в объемлющей фабрике `BeanFactory`.

`org.springframework.beans.factory.BeanNameAware`

Методу `setBeanName()` передается имя этого компонента в объемлющей фабрике `BeanFactory`.

`org.springframework.context.ResourceLoaderAware`

Методу `setResourceLoader()` передается объект `ResourceLoader`, который способен преобразовать идентификатор, представленный строкой `String`, в объект класса `Resource`.

`org.springframework.context.MessageSourceAware`

Методу `setMessageSource()` передается объект `MessageSource`, который способен построить локализованное сообщение, зная его код и параметры.

org.springframework.web.context.ServletContextAware

Методу `setServletContext()` передается объект `javax.servlet.ServletContext`. В примере 4.18 приведена встроенная реализация интерфейса `BeanNameAware`.

Пример 4.18. Встроенный Spring-компонент, написанный на JRuby, который реализует интерфейс `BeanNameAware`

```
<lang:jruby id="rubyListener"
  script-interfaces="org.jrubycookbook.ch04.Listener,
    org.springframework.beans.factory.BeanNameAware">
  <lang:inline-script><![CDATA[
class RubyListener
  # реализация интерфейса BeanNameAware
  def setBeanName(beanName)
    @beanName = beanName
  end

  # реализация интерфейса Listener
  def receiveMessage(s)
    puts "Hello, I'm named #{@beanName}"
    puts "#{@prefix}Got Message: #{s}"
  end
end

RubyListener.new
  ]]></lang:inline-script>
</lang:jruby>
```

Реализации всех этих интерфейсов очень похожи – нужно лишь сохранить внедренный объект в переменной экземпляра, – поэтому просто напрашивается мысль применить модули Ruby. В примере 4.19 приведен модуль Ruby с именем `Spring`, который включает трафаретные реализации перечисленных выше интерфейсов.

Пример 4.19. Модуль Ruby, содержащий реализации `Aware`-интерфейсов Spring

```
module Spring
  # реализация интерфейса ApplicationContextAware
  module ApplicationContextAware
    def setApplicationContext(ctx)
      @applicationContext = ctx
    end
  end

  # реализация интерфейса BeanFactoryAware
  module BeanFactoryAware
    def setBeanFactory(bf)
      @beanFactory = bf
    end
  end
end
```

```
    end
  end

  # реализация интерфейса BeanNameAware
  module BeanNameAware
    def setBeanName(beanName)
      @beanName = beanName
    end
  end

  # реализация интерфейса ResourceLoaderAware
  module ResourceLoaderAware
    def setResourceLoader(loader)
      @resourceLoader = loader
    end
  end

  # реализация интерфейса MessageSourceAware
  module MessageSourceAware
    def setMessageSource(source)
      @messageSource = source
    end
  end

  # реализация интерфейса ServletContextAware
  module ServletContextAware
    def setServletContext(ctx)
      @servletContext = ctx
    end
  end
end
```

Чтобы воспользоваться этим модулем в классе Ruby, достаточно просто включить его командой `include`, как показано в примере 4.20.

Пример 4.20. Использование модуля Spring

```
require "spring.rb"

class RubyListener
  include Spring::BeanNameAware

  # реализация интерфейса Listener
  def receiveMessage(s)
    puts "Hello, I'm named #{@beanName}"
    puts "#{@prefix}Got Message: #{s}"
  end
end
```

Определение пути загрузки JRuby

Если вы решите включать в JRuby-сценарии внешние файлы, как в примере 4.20, то очень важно знать, откуда они загружаются. Путь загрузки может отличаться в зависимости от того, используется ли JRuby внутри контейнера Spring или запускается из командной строки. Ниже показан простой Spring-компонент, который выводит путь загрузки в момент загрузки в контейнер:

```
<lang:jruby id="loadPathOutputter" script-interfaces=\
"org.springframework.beans.factory.InitializingBean">
  <lang:inline-script><![CDATA[
class LoadPathOutputter
  def afterPropertiesSet()
    puts "Ruby Path is #{$:.join(';')}"
  end
end

LibOutputter.new
  ]]></lang:inline-script>
</lang:jruby>
```

Для изменения пути загрузки JRuby можно воспользоваться системным свойством `java.home`. Дополнительную информацию см. в рецепте 3.1.

См. также

- Рецепт 3.1 «Выполнение Ruby-кода из Java»

4.9. Создание MVC-контроллеров Spring с помощью JRuby

Задача

Повторное развертывание написанного на Java контроллера в архитектуре Spring MVC может занять много времени и отвлечь от разработки. Особенно это справедливо для веб-приложений, которые состоят из многих модулей и на этапе инициализации должны загружать много данных. Хотелось бы, чтобы код контроллера можно было модифицировать, не перезагружая работающее веб-приложение.

Решение

Поддержка динамических языков в Spring может ускорить разработку MVC-приложений за счет определения контроллеров в виде объектов JRuby. Тогда вы сможете не только устранить этап компиляции, неизбежный в случае Java, но и воспользоваться встроенным в Spring механизмом самообновляемых компонентов

(см. рецепт 4.5), благодаря которому классы контроллеров можно изменять без повторного развертывания всего веб-приложения. Откройте конфигурационный файл Spring и создайте контроллер на JRuby, определив Spring-компонент с помощью атрибутов, описанных в рецептах 4.4 и 4.5. Запишите в атрибут `script-interfaces` значение `org.springframework.web.servlet.mvc.Controller`, а в атрибут `script-source` – местоположение Ruby-файла, в котором определен класс контроллера и создается его экземпляр. Отметим, что значение `script-source` задается относительно папки веб-приложения. В примере 4.21 показан конфигурационный файл Spring, содержащий контроллер `hellocontroller`, который формирует JSP-страницу.

Пример 4.21. Конфигурационный файл Spring, содержащий простой контроллер на JRuby

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:lang="http://www.springframework.org/schema/lang"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-2.5.xsd">

  <lang:jruby id="hellocontroller" refresh-check-delay="3000"
    script-source="/WEB-INF/ruby/hello.rb"
    script-interfaces="org.springframework.web.servlet.mvc.Controller">
  </lang:jruby>

  <bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
      value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp"/>
    <property name="suffix" value=".jsp"/>
  </bean>

  <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/hello.htm">hellocontroller</prop>
      </props>
    </property>
  </bean>
</beans>
```

Откройте Ruby-файл, заданный в атрибуте `script-source`, и создайте класс JRuby с методом `handleRequest`, который принимает в качестве аргументов объекты `HttpServletRequest` и `HttpServletResponse`. Метод `handleRequest` вызывается для каждого веб-запроса и возвращает объект Java-класса `ModelAndView`, который

содержит имя вида и карту модели. В последнем предложении Ruby-файла следует создать экземпляр класса контроллера. В примере 4.22 приведен написанный на JRuby контроллер, который добавляет в модель несколько значений и генерирует страницу по шаблону *hello.jsp*.

Пример 4.22. Класс JRuby, реализующий MVC-контроллер Spring

```
include Java

import org.springframework.web.servlet.ModelAndView

class HelloController
  def handleRequest(request, response)
    mav = ModelAndView.new "hello"
    mav.add_object("example", "hello!")
    mav.add_object("example_hash", {"foo"=>"bar", "alpha"=>"beta"})
    return mav
  end
end

HelloController.new
```

В JSP-странице из примера 4.23 используется стандартный синтаксис доступа к данным модели, и работает она независимо от того, какой язык реализации выберет контроллер. Ruby-хэш `example_hash`, который был добавлен в модель, преобразуется в словарь Java, а для доступа к нему используется принятая в JSP сокращенная запись вывода словарей.

Пример 4.23. Простой JSP-шаблон

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>My Sample JSP</title>
  </head>

  <body>
    String val: ${example}<br/>
    Hash val foo: ${example_hash.foo}<br/>
    Hash val moo: ${example_hash.alpha}<br/>
  </body>
</html>
```

Выполните повторное развертывание контроллера, переписав существующий Ruby-файл в своем веб-приложении. Если вы развертываете распакованный WAR-архив, то обновите этот файл в папке веб-приложения. В противном случае найдите временную папку, в которую контейнер распаковал WAR или EAR-файл

и обновите этот контроллер в ней. Справьтесь с документацией по своему серверу приложений, поскольку местоположение этой папки зависит от конкретного сервера и платформы; обычно она находится в каталоге для временных файлов или там же, где сам WAR-файл. Местоположение распакованного WAR-файла также часто выводится на консоль во время запуска и остается в протоколах сервера приложений.

Обсуждение

Контроллеры, написанные на JRuby, можно также определить в конфигурационном файле Spring, воспользовавшись техникой встраивания компонентов (см. рецепт 4.5). Компонент `inlinecontroller` в примере 4.24 содержит тот же код, который обычно помещается в Ruby-файл, заданный атрибутом `script-source`. Все веб-приложение так строить не рекомендуется, поскольку становится сложнее управлять версиями кода и утрачивается возможность повторного развертывания, но, когда нужно быстро создать прототип контроллеров или реализовать переадресацию запросов, такой прием бывает полезен. Этот подход иллюстрируется в примере 4.24.

Пример 4.24. Встроенное определение контроллера на JRuby

```
<lang:jruby id="inlinecontroller"
  script-interfaces="org.springframework.web.servlet.mvc.Controller">
  <lang:inline-script>
include Java
import org.springframework.web.servlet.ModelAndView
class MySecController
  def handleRequest(request, response)
    ModelAndView.new "redirect:/hello.htm"
  end
end
MySecController.new
  </lang:inline-script>
</lang:jruby>
```

См. также

- Рецепт 4.5 «Определение Spring-компонентов на JRuby»
- Рецепт 4.6 «Создание самообновляемых Spring-компонентов на JRuby»
- Рецепт 4.7 «Встраивание Spring-компонентов на JRuby»

4.10. Hibernate и JRuby

Задача

Требуется использовать в написанном на JRuby приложении каркас Hibernate.

Решение

В идеале работа с объектом доступа к данным (Data Access Object – DAO), который является основой каркаса Hibernate, не должна была бы отличаться от работы с любым другим классом Java. Но разработчики на JRuby сталкиваются с тем, что в Hibernate повсеместно используются родовые типы (Java Generics), а JRuby не умеет создавать такие классы или вызывать методы, в аргументах которых встречаются родовые типы. Hibernate предоставляет разработчикам на Java большую гибкость в реализации DAO-объектов, и во многих реализациях применяются родовые типы, чтобы уменьшить размер классов и количество методов. Однако типичный способ создания DAO-объектов, который описан в большинстве онлайн-руководств, не подразумевает наличия родовых типов в открытом интерфейсе DAO, пусть даже внутри они применяются. Обычно DAO-объекты порождаются с помощью некоего фабричного интерфейса или путем создания обертывающих DAO-объектов для классов. Программа на JRuby в примере 4.25 получает объект PersonDao от фабрики, тогда как объект EventDao создается напрямую.

Пример 4.25. Получение DAO-объектов в каркасе Hibernate

```
include Java

import example.dao.PersonDao
import example.dao.DaoFactory
import example.dao.EventDao
import example.model.Person
import example.model.Event
import util.HibernateUtil

event_dao = EventDao.new
event_dao.set_session HibernateUtil::get_session_factory.get_current_session
dao.create(Event.new("JRuby Meeting", java.util.Date.new))
dao.find_all.each do |e| puts "#{e.get_title } #{e.get_date}"; end

person_dao = DaoFactory.instantiate(PersonDao.class)
dao.create(Person.new("Justin", "Wood"))
dao.create(Person.new("Brian", "Henry"))
dao.find_all.each do |p| puts "#{p.get_firstname} #{p.get_lastname}"; end
```

Обсуждение

Для получения сеанса Hibernate используется статический метод класса HibernateUtil. Затем сеанс вручную передается объекту EventDao. В Hibernate стандартно применяется паттерн проектирования, согласно которому доступ к фабрике сеансов дает статический метод глобального служебного класса. Класс HibernateUtil становится единым местом конфигурирования и управления и позволяет скрыть многие детали объектно-реляционного отображения от DAO-объектов.

Транзакции базы данных элегантно выражаются с помощью написанной на Ruby функции, которая вызывает переданный ей блок. Этот блок содержит код взаимодействия с базой данных и выполняется после того, как объемлющая функция начала транзакцию, но перед тем, как она ее зафиксировала. Ошибки можно обнаружить и обработать в функции, которая управляет транзакцией, не загромождая этими деталями содержательный код. В результате мы получаем чистый API, где устранены многословные повторяющиеся вызовы для работы с транзакциями. При этом транзакционная логика выражается метафорой вызова функции, а не явными обращениями к методам инициирования и завершения транзакции. В примере 4.26 определен модуль `TransactionHelper`, в котором собраны функции для инициирования стандартной транзакции JDBC и транзакции, определенной в более общем Java Transaction API (JTA). Сюда же включен контроллер для демонстрации порядка работы с этим модулем, показывающий, как легко снабдить код работы с базой данных тем или иным транзакционным механизмом.

Пример 4.26. Применение блоков для определения транзакций

```
include Java

import util.HibernateUtil
import javax.naming.InitialContext

module TransactionHelper

  def with_transaction
    begin
      tx = HibernateUtil.session_factory.current_session.beginTransaction
      yield
      tx.commit
      HibernateUtil.session_factory.current_session.close
    rescue
      tx.rollback
    end
  end

  def with_jta_transaction
    begin
      ctx = InitialContext.new
      utx = ctx.lookup("java:comp/UserTransaction");
      utx.begin();
      yield
      utx.commit
    rescue
      utx.rollback
    end
  end
end

class UserController
```

```
    extend TransactionHelper
  def create
    with_transaction do
      @id = User.create("Tom")
    end

    with_jta_transaction do
      tom = User.find_by_id(@id)
    end
  end
end
end
```

4.11. Java Persistence API и JRuby

Задача

Требуется воспользоваться в написанном на JRuby приложении технологией Java Persistence API (JPA).

Решение

Воспользуйтесь статическим методом `Persistence.createEntityManagerFactory()`, определенном в JPA, для порождения фабрики единиц сохранения (`persistence unit`). При вызове метода `createEntityManager()` этой фабрики создается объект класса `EntityManager`, который и является основным средством доступа к Persistence API. Класс `EntityManager` аналогичен классу `Session` из каркаса Hibernate или классу `ClientSession` из пакета TopLink; он содержит методы для взаимодействия с базой данных и объектами вашей модели. Объект `EntityManager` небезопасен относительно потоков, поэтому его не следует использовать для параллельной обработки нескольких запросов. Он проектировался не как долгоживущий программный компонент, а в предположении, что будет использован и почти сразу же уничтожен. В примере 4.27 показано приложение на JRuby, в котором создается несколько объектов `User`, после чего база данных опрашивается на предмет того, успешно ли они были добавлены.

Пример 4.27. Пример работы с JPA из JRuby

```
include Java

import javax.persistence.Persistence
import cookbook.User

def with_trans(em)
  t = em.getTransaction();
  begin
    t.begin()
    yield
  end
end
```

```
t.commit
ensure
  t.rollback if t.isActive
end
end

emf = Persistence.createEntityManagerFactory(«hello-world»)
em = emf.createEntityManager

with_trans(em) do
  u = User.new("stephen", "lee", "slee", "password", "stephen@ora.com")
  u2 = User.new("stephen", "smith", "ssmith", "password", "ssmith@ora.com")
  em.persist(u)
  em.persist(u2)
end

query = em.createQuery("select u from User u where u.firstname = :firstname").
query.set_parameter("firstname", "stephen").
hu = query.get_result_list

hu.each do |u|
  puts "found #{u.firstname} #{u.lastname}"
end

em.close
emf.close
```

Обсуждение

В этом примере демонстрируется еще одно (см. рецепт 4.9) применение блока для выражения семантики JPA-транзакции. Этот вспомогательный метод заодно автоматически откатывает транзакцию в случае ошибки.

См. также

- Рецепт 4.10 «Hibernate и JRuby»

4.12. Выполнение вызовов по протоколу SOAP

Авторы выражают за этот рецепт благодарность Стивену Шинглеру (Steven Shingler).

Задача

Требуется вызвать удаленный метод с помощью веб-службы на основе протокола SOAP.

Решение

Воспользуйтесь клиентским модулем Mule (<http://mule.mulesource.org>) и какой-нибудь библиотекой анализа XML для Ruby: REXML или Hpricot. В примере 4.28 модуль Mule используется для обращения с запросом к одной из веб-служб, предоставляемых Национальным управлением США по изучению океана и атмосферы (NOAA).

Пример 4.28. Отправка SOAP-запроса с помощью клиентского модуля Mule

```
include Java

require "rexml/document"
import org.mule.module.client.MuleClient

url = "axis:http://www.weather.gov/forecasts/xml/SOAP_server/ndfdXMLserver.php"
method = "method=LatLonListZipCode"
client = MuleClient.new
message = client.send("#{url}?#{method}", "10036", nil)
doc = REXML::Document.new message.payload
puts doc.root.elements[1].text
exit
```

Для запуска этого сценария в путь поиска классов необходимо добавить Mule и несколько зависимостей для него. Из-за требований, предъявляемых загрузчиком классов, эти зависимости должны быть включены в системный путь поиска (т.е. заданы в переменной среды CLASSPATH); их нельзя добавить в путь поиска классов с помощью реализованного в JRuby расширения метода require, как описано в рецепте 1.7. Для этого конкретного сценария зависимости можно добавить в путь поиска классов такими командами:

```
export MULE_LIB=/opt/mule/lib
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/activation-1.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/axis-1.4.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/axis-jaxrpc-1.4.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/backport-util-concurrent-3.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-beanutils-1.7.0.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-codec-1.3.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-collections-3.2.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-discovery-0.2.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-httpclient-3.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-io-1.3.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-lang-2.3.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-logging-1.1.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/commons-pool-1.4.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/dom4j-1.6.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/geronimo-j2ee-connector_1.5_spec-1.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/geronimo-servlet_2.5_spec-1.1.jar
```

```
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/jaxen-1.1.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/jug-2.0.0-asl.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/mule/mule-core-2.0.2.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/mule/mule-module-client-2.0.2.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/mule/mule-transport-axis-2.0.2.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/saaj-api-1.3.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/stax-api-1.0.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/wsdl4j-1.6.1.jar
export CLASSPATH=$CLASSPATH:$MULE_LIB/opt/wstx-asl-3.2.6.jar
```



В «полный дистрибутив» Mule включены все зависимости, кроме пакета Jakarta Commons Logging; последний можно скачать с сайта <http://commons.apache.org/logging/>.

Обсуждение

Метод `send` класса `MuleClient` принимает в качестве полезной нагрузки сообщения произвольный объект. Однако будьте осторожны при передаче объектов, отличных от примитивов Java и их эквивалентов в Ruby. В этом случае воспользуйтесь инструментом Axis WSDL2Java для генерации классов Java по описанию веб-службы:

```
$ java org.apache.axis.wsdl.WSDL2Java \
http://www.weather.gov/forecasts/xml/SOAP_server/ndfdXMLserver.php?wsdl
```

В примере 4.28 URL конечной точки веб-службы NOAA снабжен префиксом `axis`, который служит для Mule указанием на то, что мы хотим воспользоваться библиотекой Axis для вызова веб-службы. Если включить в путь поиска классов другие или дополнительные зависимости, то можно будет использовать иные библиотеки и транспортные механизмы.

См. также

- Сайт Mule <http://mule.mulesource.org/>
- Сайт Apache Axis <http://ws.apache.org/axis/>
- Сайт REXML <http://www.germane-software.com/software/rexml/>
- Сайт Hpricot <http://code.whytheluckystiff.net/hpricot/>

4.13. Упрощение доступа к LDAP-каталогу

Задача

Требуется найти записи и атрибуты в LDAP-каталоге с помощью технологии JNDI, но хочется по возможности упростить API.

Решение

Воспользуйтесь механизмом открытых классов JRuby (описан в рецепте 1.9) для добавления вспомогательных методов в класс `com.sun.jndi.ldap.LdapCtx`.

Обсуждение

JNDI API, безусловно, является чрезвычайно мощным средством, но зачастую оказывается излишне многословным. Так, на Java код, который нужно написать для доступа к единственному атрибуту, выглядит чересчур громоздко:

```
// Найти запись
LdapContext entry = ctx.lookup("uid=mts,ou=People,dc=umich,dc=edu");

// Сначала получить все атрибуты, ассоциированные с этой записью.
Attributes attributes = entry.getAttributes("");

// Затем получить один атрибут по имени.
Attribute attribute = attributes.get("mail");

// И только теперь можно получить его значение.
String value = (String) attribute.get();
```

Для атрибута с несколькими значениями все еще хуже:

```
// Найти запись
LdapContext entry = ctx.lookup("uid=mts,ou=People,dc=umich,dc=edu");

// Сначала получить все атрибуты, ассоциированные с этой записью.
Attributes attributes = entry.getAttributes("");

// Затем получить один атрибут по имени.
Attribute attribute = attributes.get("mail");

// Затем получить объект NamingEnumeration, содержащий все значения атрибута.
NamingEnumeration ne = attribute.getAll();

// Создать список, в цикле обойти NamingEnumeration
// и добавить каждое значение в список
List<String> values = new ArrayList<String>();
while (ne.hasMore()) {
    values.add(ne.next());
}
```

В примере 4.29 показаны два добавленных в класс `LdapCtx` метода, которые значительно упрощают работу с этим API.

Пример 4.29. Добавление методов в класс `LdapCtx`

```
include Java

import com.sun.jndi.ldap.LdapCtx
```

```

class LdapCtx
  def get_attribute_value(key)
    get_attributes("", [key].to_java(:string)).get(key).get
  end
  def get_attribute_values(key)
    values = []
    enum = get_attributes("", [key].to_java(:string)).get(key).get_all
    while enum.has_more
      values << enum.next
    end
    return values
  end
end
end

```

Применение этих методов позволяет следующим образом переписать код доступа к атрибутам в LDAP-каталоге:

```

entry = ctx.lookup("uid=mts,ou=People,dc=umich,dc=edu")

p "Email = #{entry.get_attribute_value("mail")}
entry.get_attribute_values("cn").each do |name|
  p "Name = #{name}"
end

```

Чтобы пример 4.29 заработал, необходимо воспользоваться поддержкой LDAP JNDI, которую корпорация Sun предлагает в виде пакета `com.sun.ldap.jndi`. Как правило для этой цели создается объект `JNDI Context`, как показано в рецепте 4.1. При желании можно легко адаптировать пример 4.29 и к другой библиотеке работы с LDAP. Нужно лишь узнать имя класса, который реализует интерфейс `javax.naming.directory.DirContext`. Для этой цели проще всего воспользоваться программой `jirb`:

```

$ jirb
irb(main):001:0> include Java
irb(main):002:0> import java.util.Hashtable
irb(main):003:0> import javax.naming.InitialContext
irb(main):004:0> import javax.naming.Context
irb(main):005:0> env = {
irb(main):006:1*   Context::INITIAL_CONTEXT_FACTORY,
irb(main):007:1*   "com.sun.jndi.ldap.LdapCtxFactory",
irb(main):008:1* Context::PROVIDER_URL,
irb(main):009:1* "ldap://ldap.itd.umich.edu:389"
irb(main):010:1> }
irb(main):011:0> ctx = InitialContext.new(Hashtable.new(env))
irb(main):012:0> ctx.lookup("uid=mts,ou=People,dc=umich,dc=edu").java_class
=> com.sun.jndi.ldap.LdapCtx

```

Пользовательский интерфейс и графические приложения

5.0. Введение	148
5.1. Создание приложений Swing	148
5.2. Обработка событий Swing .	150
5.3. Долго работающие задачи в приложениях Swing	151
5.4. Пакетирование автономных приложений	153
5.5. Пакетирование JRuby-приложений, запускаемых по технологии Web Start	155
5.6. Написание апплетов на JRuby	157
5.7. Манипулирование изображениями	161
5.8. Создание приложений SWT	164
5.9. Доступ к рабочему столу ...	166
5.10. Доступ к системному лотку	167
5.11. Разработка приложений Swing на предметно-ориентированных языках на базе JRuby	169
5.12. Использование библиотеки Monkeybars для разработки приложений Swing	173
5.13. Создание приложений Qt с помощью JRuby	177

5.0. Введение

Сообщество JRuby уделяет очень много внимания разработке веб-приложений, однако JRuby не менее полезен и в качестве инструмента создания персональных клиентских программ. Открывая среде исполнения доступ к графической подсистеме, JRuby позволяет писать графические приложения с использованием таких широко известных библиотек, как Abstract Windowing Toolkit (AWT), Swing и Simple Widget Toolkit (SWT), а также и более современных, например Qt Jambi. Каждая из этих библиотек не только предлагает богатый набор элементов для построения пользовательского интерфейса, но и поддерживает тесную интеграцию с операционной системой. В нескольких рецептах из этой главы объясняется, как с помощью JRuby создать значок в системном лотке, получить доступ к рабочему столу и обратиться к платформенным библиотекам, реализующим графический интерфейс пользователя (ГИП).

Учитывая растущую популярность декларативного программирования и отличную приспособленность Ruby к построению предметно-ориентированных языков (Domain-Specific Language – DSL), следует ожидать, что разработчики на JRuby постараются усовершенствовать традиционные подходы к программированию пользовательского интерфейса Java UI. Существует несколько решений, позволяющих упростить разработку на платформе Swing: Swigby, Cheri::Swing, Monkeybars и Profligacy. Аналогично для разработки на SWT был организован проект Glimmer Eclipse, а в пакет Qt::JRuby встроена поддержка DSL.

Gem-пакет Rawr представляет собой полезный инструмент для пакетирования JRuby-приложений, предназначенных как для запуска на локальном компьютере, так и в веб. В этот пакет включен ряд заданий Rake, которые можно сконфигурировать для пакетирования JRuby-программы в виде исполняемого JAR-файла, исполняемого файла Windows, приложения Mac OS X или приложения, запускаемого по технологии Web Start. В том же рецепте описывается техника применения JRuby для создания Java-апплетов.

Обработка изображений – это одна из немногих областей, в которых среды исполнения Ruby все еще полагаются на платформенные библиотеки или написанный на C код. В качестве альтернатив популярным gem-пакетам RMagic и ImageScience можно воспользоваться пакетами RMagic4J и ImageVoodoo. Если вам нужны более продвинутое средства, то можете обратиться к Java 2D API.

5.1. Создание приложений Swing

Задача

Требуется разработать пользовательский интерфейс на базе библиотеки Java Swing с использованием JRuby.

Решение

Среда исполнения JRuby поддерживает графические библиотеки и компоненты Swing. В примере 5.1 показано простое приложение Swing, которое выводит сообщение в окне.

Пример 5.1. Простой пользовательский интерфейс на основе Swing

```
include Java
import javax.swing.JFrame

frame = JFrame.new "JRuby Message"
frame.default_close_operation = JFrame::EXIT_ON_CLOSE
msg = javax.swing.JLabel.new "JRuby Rocks"
frame.content_pane.add msg
frame.pack
frame.visible = true
```

Обсуждение

JRuby может обращаться ко всем компонентам Swing API, даже таким продвинутым, как библиотеки стилистического оформления. В примере 5.2 показано, как перейти от принимаемой по умолчанию в Swing темы Metal к платформенному внешнему облику.

Пример 5.2. Изменение внешнего облика приложения

```
include Java

import javax.swing.JFrame
import javax.swing.UIManager

frame = JFrame.new "JRuby Look And Feel"
frame.default_close_operation = JFrame::EXIT_ON_CLOSE
frame.content_pane.layout = java.awt.GridLayout.new(1, 2)

{:metal => "javax.swing.plaf.metal.MetalLookAndFeel",
 :system => UIManager::getSystemLookAndFeelClassName}.each do |l,c|
  but = javax.swing.JButton.new l.to_s
  but.add_action_listener do |evt|
    UIManager::look_and_feel = c
    javax.swing.SwingUtilities::updateComponentTreeUI frame
    frame.pack
  end
  frame.add(but)
end
```

```
frame.pack
frame.visible = true
```

Можно также воспользоваться сторонними библиотеками стилистического оформления, например Substance или Narkin. Для этого достаточно включить соответствующие JAR-файлы в путь поиска классов и указать имя класса Look and Feel, отвечающего за внешний облик приложения.

См. также

- Рецепт 5.2 «Обработка событий Swing»
- Рецепт 5.3 «Долго работающие задачи в приложениях Swing»

5.2. Обработка событий Swing

Задача

Требуется обрабатывать события, возбуждаемые компонентами Swing.

Решение

Обычно для обработки событий в JRuby применяются блоки. Для приемников событий с одним-единственным методом таких, как `javax.awt.event.ActionListener`, этот подход позволяет писать очень лаконичный код обработки. В примере 5.3 блоки используются для обработки нажатия на кнопку и изменений в текстовом поле.

Пример 5.3. Обработка событий в блоках Ruby

```
include Java
import javax.swing.JFrame

frame = JFrame.new "Event Handler - Coerced"
frame.default_close_operation = JFrame::EXIT_ON_CLOSE

t = javax.swing.JTextField.new(10)
b = javax.swing.JButton.new("search")
b.add_action_listener { |evt| puts "searching" };
t.document.add_document_listener { |evt| puts "checking #{t.text}" };

frame.layout = java.awt.GridLayout.new(1, 2)
frame.add t
frame.add b
frame.pack
frame.visible = true
```

Обсуждение

Можно также реализовать Java-интерфейс приемника с помощью метода `impl`, передавая блок, внутри которого обрабатывается событие. Этот подход полезен,

когда в интерфейсе обработчика событий определено несколько методов. В примере 5.4 показано, как можно перехватить события от компонента меню.

Пример 5.4. Обработка события с помощью экземпляра интерфейса Java

```
include Java
import javax.swing.JFrame

frame = JFrame.new
frame.default_close_operation = JFrame::EXIT_ON_CLOSE

bar = javax.swing.JMenuBar.new
menu = javax.swing.JMenu.new "File"
item = javax.swing.JMenuItem.new "Open"

menu.add_menu_listener(javax.swing.event.MenuListener.impl do |method, evt|
  puts evt.class
  case method.to_s
    when "menuDeselected"
      puts 'hidden'
    when "menuSelected"
      puts 'visible'
  end
end)

menu.add item
bar.add menu
frame.jmenu_bar = bar
frame.pack
frame.visible = true
```

См. также

- Рецепт 1.9 «Реализация Java-интерфейса в Ruby»

5.3. Долго работающие задачи в приложениях Swing

Задача

Диспетчерский поток в Swing отвечает за прорисовку пользовательского интерфейса и обработку событий. Требуется запустить долго работающую задачу в ответ на событие Swing, но так, чтобы интерфейс при этом не «подвисал».

Решение

Класс `javax.swing.SwingWorker` специально предназначен для запуска долго работающих задач и обеспечивает безопасное обновление элементов пользовательско-

го интерфейса в контексте диспетчерского потока. Его реализация уточнялась на протяжении нескольких лет в различных проектах с открытым исходным кодом, и в конце концов была официально включена в базовые библиотеки Java в версии Java 6. При работе с классом `SwingWorker` создается расширяющий его новый класс-исполнитель (`worker`), в котором следует реализовать метод `doInBackground`, где и выполняется долго работающая задача. Применение класса `SwingWorker` продемонстрировано в примере 5.5. Отметим, что компонент `button` является членом класса-исполнителя, так как эта переменная недоступна в области видимости нового класса.

Пример 5.5. Применение класса `SwingWorker` для реализации долго работающих задач

```
include Java
import javax.swing.JFrame

frame = JFrame.new "Swing Worker"
frame.default_close_operation = JFrame::EXIT_ON_CLOSE

start = javax.swing.JButton.new("start")

# определить функцию с помощью блока
start.add_action_listener do |evt|
  class MySwingWorker < javax.swing.SwingWorker
    attr_accessor :button
    def doInBackground
      10.times do
        puts "thread #{self.hashCode} working"
        sleep(1)
      end
      self.button.text = "Completed"
    end
  end
end

sw = MySwingWorker.new
sw.button = start
sw.execute

end

frame.add start
frame.pack
frame.visible = true
```

Обсуждение

В версии 1.1 JRuby не может инстанцировать абстрактные классы Java, поэтому для реализации абстрактных методов необходимо создавать подкласс `SwingWorker`. Это один из немногих аспектов, в которых код на JRuby получается менее прозрачным и элегантным, чем на Java, однако разработчики JRuby планируют улучшить поддержку абстрактных классов в будущих версиях.

В классе `SwingWorker` есть дополнительные методы для реализации таких возможностей, как извещение о ходе выполнения задачи, отмена задачи и уведомление о завершении ее работы. Чтобы воспользоваться этими функциями, изучите документацию по API и перегрузите нужные вам методы в своем классе Ruby.

Существует версия класса `SwingWorker` для Java 5. Концептуально она аналогична версии для Java 6, но родовые типы Java в ней не используются, а названия методов немного отличаются. Например, метод `construct` в классе для Java 5 – аналог метода `doInBackground` в версии `SwingWorker` для Java 6.

См. также

- Рецепт 1.9 «Реализация Java-интерфейса в Ruby»
- Java 5 `SwingWorker` <http://java.sun.com/products/jfc/tsc/articles/threads/src/SwingWorker.java>

5.4. Пакетирование автономных приложений

Задача

Требуется собрать пакет, содержащий приложение на JRuby в виде исполняемого JAR-файла, исполняемого файла Windows или приложения для Mac OS X.

Решение

Установите gem-пакет `Rawr`. Его разработал Дэвид Кунц (David Koontz) с целью упростить пакетирование JRuby-приложений для Windows, Mac, Linux и сред исполнения Java:

```
$ jruby -S gem install rawr
```

Сконфигурируйте среду сборки, выполнив команду `rawr install` в папке сборки, обычно это корневая папка проекта:

```
$ cd /projects/rawrdemo
$ jruby -S rawr install
```

Эта команда создает два файла: `build_configuration.yaml` и `src/org/rubyforge/rawr/Main.java`. Последний содержит класс Java, который создает экземпляр среды исполнения JRuby и выполняет в ней Ruby-сценарий вашего приложения. Скопируйте все файлы своего JRuby-приложения в созданную папку `src`.

Если в вашем проекте используются написанные вами классы Java, то упакуйте их в JAR-файл и поместите JAR-файл своего проекта, а также JAR-файлы, от которых зависит ваше приложение, в папку `lib/java`. В этой же папке должен находиться файл `jruby-complete.jar`.

Откройте файл *build_configuration.yaml* и присвойте параметру `project_name` имя, которое должен иметь конечный исполняемый файл. Параметру `main_ruby_file` присвойте имя главного исполняемого сценария приложения или же назовите этот файл *main.rb* (это имя подразумевается по умолчанию). В примере 5.6 приведен образец конфигурационного файла.

Пример 5.6. Пример конфигурационного файла *Rawr*

```
# Имя создаваемого jar-файла
project_name: jruby_cookbook_app

# Какой Ruby-файл вызывать при запуске jar-файла
main_ruby_file: jruby_cookbook_main
```

Исполняемый JAR-файл

Запустите Rake-задание `rawr:jar` для генерации исполняемого JAR-файла:

```
$ jruby -S rake rawr:jar
```

Созданные в результате файлы находятся в каталоге *package/deploy*. В их число входят главный исполняемый JAR-файл *jruby_cookbook_app.jar*, конфигурационный файл и JAR-файл, содержащий среду исполнения JRuby. Все эти файлы нужно будет поместить в ту же папку, где находятся другие JAR-файлы, распространяемые вместе с вашим приложением. Для тестирования созданного JAR-файла выполните такую команду:

```
$ java -jar package/deploy/jruby_cookbook_app.jar
```

Исполняемый файл Windows

Запустите Rake-задание `rawr:bundle:exe` для генерации исполняемого файла Windows:

```
$ jruby -S rake rawr:bundle:exe
```

Приложение для Windows состоит из одного EXE-файла, нескольких JAR-файлов и конфигурационного файла; все они находятся в каталоге *package/native_deploy/windows*. Включать в дистрибутив и устанавливать нужно все файлы в этом каталоге, а не только EXE-файл.

Приложение для Mac OS X

Запустите Rake-задание `rawr:bundle:app` для генерации приложения для Mac OS X:

```
$ jruby -S rake rawr:bundle:app
```

Упакованная папка приложения OS X называется *project_name.app* и находится в каталоге *package/native_deploy/mac*.

Обсуждение

Файл *build_configuration.yml* хорошо документирован и содержит многочисленные параметры для настройки процедуры сборки. Они позволяют задать путь поиска классов, местоположение исходных файлов на Java и JRuby, пути поиска библиотечных файлов и папку, в которой должны размещаться результирующие исполняемые файлы. С помощью параметра `jars_data_dirs` можно также включить в приложение произвольные файлы данных или мультимедиа.

Для получения полного списка Rake-заданий укажите при запуске Rake флаг `-T`. Чтобы случайно не «подцепить» лишние файлы рекомендуется перед каждой сборкой выполнять задание `rawr:clean`.

```
$ jruby -S rake -T
rake rawr:bundle:app # Пакетирует jar-файл, созданный rawr:jar, в приложение для
Mac O...
rake rawr:bundle:exe # Пакетирует jar-файл, созданный rawr:jar, в приложение для
Windo...
rake rawr:bundle:web # Пакетирует jar-файл, созданный rawr:jar в приложение для
Java Web Sta...
rake rawr:clean # Удаляет каталог для генерируемых файлов
rake rawr:compile # Компилирует все исходные файлы Java в каталоге...
rake rawr:jar # Создает из откомпилированных файлов исполняемый jar-файл...
rake rawr:prepare # Создает каталог для генерируемых файлов и его подкаталоги...
rake rawr:setup_consts # Подготавливает различные константы, необходимые Rawr...
```

См. также

- Рецепт 3.1 «Выполнение Ruby-кода из Java»
- Рецепт 5.9 «Доступ к рабочему столу»
- Сайт Rawr website <http://gitorious.org/projects/rawr>

5.5. Пакетирование JRuby-приложений, запускаемых по технологии Web Start

Задача

Требуется оформить JRuby-программу в виде пакета, запускаемого по технологии Java Web Start.

Решение

Установите gem-пакет Rawr. О том, как конфигурировать и использовать этот пакет, см. рецепт 5.4. Из-за модели безопасности Web Start и способа работы JRuby с виртуальной машиной, главный JAR-файл и JAR-файл, содержащий среду ис-

полнения JRuby, должны быть подписаны, иначе они не смогут работать в «песочнице» Web Start. Поэтому начните с генерации файла хранилища ключей с именем *myKeystore*, указав в качестве псевдонима строку *myself*. Получив приглашение, введите и пароль и другую запрашиваемую информацию:

```
$ keytool -genkey -keystore myKeystore -alias myself
Enter keystore password: dumbpassword
What is your first and last name?
[Unknown]: Henry Liu
What is the name of your organizational unit?
[Unknown]: Global Digital
What is the name of your organization?
[Unknown]: MTV Networks
What is the name of your City or Locality?
[Unknown]: New York
What is the name of your State or Province?
[Unknown]: NY
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Henry Liu, OU=Global Digital, O=MTV Networks, L=New York, ST=NY, C=US
correct?
[no]: yes

Enter key password for <myself>
(RETURN if same as keystore password):
```

С помощью только что созданного хранилища ключей сгенерируйте самоподписанный сертификат, задав флаг *selfcert*:

```
$ keytool -selfcert -alias myself -keystore myKeystore
```

Откройте файл *build_configuration.yaml* и создайте в нем хэш *web_start*, в котором будет ключ *self_sign*, равный *true*, и ключ *self_sign_passphrase*, равный паролю сертификата. Создайте также хэш *jnlp*, в котором должны быть заданы значения для ключей *codebase*, *description*, *vendor* и *homepage_href*. В примере 5.7 показано, как задаются хэши в конфигурационном YAML-файле.

Пример 5.7. Параметры Web Start в конфигурационном файле Rawr

```
web_start: { self_sign: true, self_sign_passphrase: password }

jnlp: {
  codebase: http://localhost:8080,
  description: My Webstart Demo,
  vendor: Your Name,
  homepage_href: http://www.ora.com
}
```

Подпишите JAR-файл со средой исполнения JRuby и все прочие включенные в пакет JAR-файлы, которые напрямую вызывают операционную систему, обра-

щаются к сетевым службам или выдают сообщения об ошибках, связанных с безопасностью:

```
$ jarsigner -keystore myKeystore -storepass password lib/java/jruby-complete.jar
```

Запустите Rake-задание `rawr:bundle:web`, которое сгенерирует приложение для Web Start:

```
$ jruby -S rake rawr:bundle:web
```

Приложение будет помещено в каталог `package/native_deploy/web`. Переместите все файлы в папку, из которой веб-сервер раздает файлы, и запустите Web Start-приложение, открыв в браузере JNLP-файл. Например, если веб-сервер работает на машине `localhost` и использует порт `8080`, то следует ввести URL `http://localhost:8080/jruby_cookbook.jnlp`.

Обсуждение

Можно использовать схему `file://` и инструмент `javaws` для тестирования Web Start-приложения без веб-сервера и браузера. Запишите в параметр `codebase` в файле `build_configuration.yaml` путь к каталогу развертывания, как показано ниже, и пересоберите приложение:

```
codebase: file:///C:/rawdemo/package/native_deploy/web
```

Запустите Web Start-приложение командой `javaws`:

```
$ javaws package\native_deploy\web\jruby_cookbook.jnlp
```

Не забудьте перед окончательным развертыванием записать в `codebase` настоящий веб-адрес своего приложения.

См. также

- Рецепт 5.4 «Пакетирование автономных приложений»

5.6. Написание апплетов на JRuby

Задача

Требуется написать на JRuby Java-апплет.

Решение

Создание апплетов на языке JRuby немного отличается от создания приложений для рабочего стола, поскольку в коде на Ruby нельзя создавать главное окно приложения, а следует добавлять компоненты в панель контента, предоставляемую родительским апплетом. В примере 5.8 показано одно из возможных решений – передать среде исполнения JRuby панель контента в виде глобальной переменной.

Пример 5.8. Написанный на JRuby апплет, в котором панель контента представлена глобальной переменной

JRubyApplet.java

```
package org.jrubycookbook;

import java.util.ArrayList;
import org.jruby.Ruby;
import org.jruby.javasupport.*;
import java.awt.Container;
import org.jruby.runtime.builtin.IRubyObject;
import org.jruby.runtime.*;

public class JrubyApplet extends javax.swing.JApplet {
    public void init(){
        Ruby runtime = JavaEmbedUtils.initialize(new ArrayList<String>());
        runtime.evalScriptlet("require \"java\"\n\nclass FreshForJava\n\nd\n");
        final IRubyObject blankRuby = runtime.evalScriptlet("FreshForJava.new");
        IRubyObject globValue = JavaUtil.convertJavaToRuby(runtime,
            this.getContentPane());
        globValue = Java.java_to_ruby(blankRuby, globValue, Block.NULL_BLOCK);
        GlobalVariable gv = new GlobalVariable(runtime, "$content_pane",
            globValue);
        runtime.defineVariable(gv);
        String bootRuby = "require 'appletmain' \n";
        runtime.evalScriptlet( bootRuby );
    }
}
```

appletmain.rb

```
include Java

import javax.swing.JPanel
import javax.swing.JButton

jp = JPanel.new
but = JButton.new("OK")
but.add_action_listener do |evt|
    puts "pressed"
end
jp.add(but)
$content_pane.add(jp)
```

Упакуйте Ruby-сценарии в JAR-файл и сошлитесь на этот файл из HTML-тега `applet`. В атрибуте `archive` укажите также файл `jruby-complete.jar`, содержащий среду исполнения JRuby. В примере 5.9 приведен образец тега `applet` на HTML-странице.

Пример 5.9. Тег `applet` для апплета, написанного на JRuby

```
<applet width="200" height="200" align="baseline"
  code="org.jrubycookbook.JrubyApplet.class"
  codebase="." pluginspage="http://java.sun.com/j2se/1.6.0/download.html"
  archive="jrubyapplet.jar,jruby-complete.jar"
</applet>
```

В версии Java 6 update 10 появился еще один метод внедрения апплета с помощью кода на языке JavaScript. Эта техника демонстрируется в примере 5.10.

Пример 5.10. Развертывание апплета с помощью кода на JavaScript

```
<script src="http://java.com/js/deployJava.js"></script>
<script>
  deployJava.runApplet({codebase:"",
    archive:"jruby-complete.jar,jrubyapplet.jar",
    code:"org.jruby.JRubyApplet.class",
    width:"320", Height:"400"}, null, "1.6");
</script>
```

Обсуждение

В примере 5.11 показан другой подход: создать объект Swing Panel и вернуть его из JRuby-сценария. Внешний вид и поведение пользовательского интерфейса определяются классом `MyPanel`, который находится в файле `appletmainclass.rb`.

Пример 5.11. Альтернативная реализация апплета, написанного на JRuby

```
JRubyApplet.java
```

```
public class JrubyApplet extends javax.swing.JApplet {
    public void init() {
        Ruby runtime = JavaEmbedUtils.initialize(Collections.emptyList());
        String bootRuby = "require 'appletmainclass' \n MyPanel.new \n";
        IRubyObject ro = runtime.evalScriptlet(bootRuby);
        Container panel = (Container)JavaEmbedUtils.rubyToJava
(runtime, ro, Container.class);
        this.getContentPane().add(panel);
        this.setSize(100,100);
    }
}
```

```
appletmainclass.rb
```

```
include Java
```

```
class MyPanel < javax.swing.JPanel
  include_package 'javax.swing'

  def initialize
    super
```

```

        but = JButton.new("OK")
        but.add_action_listener do |evt|
            puts "pressed too"
        end
        add(but)
    end
end
end

```

В Java 6 и более поздних изданиях с поддержкой спецификации JSR-223 этот код можно немного упростить. В примере 5.12 устранено делегирование из Java объекту JRuby, поэтому код стал понятнее.

Пример 5.12. Написанный на JRuby апплет с применением технологии Java Scripting

```

package org.jrubycookbook;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
import java.awt.Container;

public class JrubyApplet extends javax.swing.JApplet {

    public void init(){
        ScriptEngine runtime = new ScriptEngineManager().getEngineByName("jruby");
        String bootRuby = "require 'main' \n MyPanel.new \n";
        try{
            Container c = (Container)runtime.eval(bootRuby);
            this.getContentPane().add(c);
            this.setSize(100,100);
        } catch(ScriptException e) {
            e.printStackTrace();}
    }
}

```

В примере 5.13 показано, как с помощью JSR-223 API сделать панель контента апплета доступной через глобальную переменную.

Пример 5.13. Апплет, в котором применяется технология Java Scripting и подход на основе глобальной переменной

```

public class JrubyApplet extends javax.swing.JApplet {

    public void init(){
        ScriptEngine runtime = new ScriptEngineManager().getEngineByName("jruby");
        runtime.put("content_pane",this.getContentPane());
        String bootRuby = "require 'mainpassed' \n";
        try{
            runtime.eval(bootRuby);
        } catch(ScriptException e) {
            e.printStackTrace();
        }
    }
}

```

См. также

- Рецепт 3.3 «Вызов JRuby с помощью технологии Java Scripting»

5.7. Манипулирование изображениями

Задача

Требуется написать на JRuby код, который масштабирует или еще как-то модифицирует изображение.

Решение

Если нужно решить относительно простую задачу, например, сгенерировать миниатюру, то воспользуйтесь какой-нибудь совместимой с JRuby библиотекой обработки изображений, скажем RMagick4J или ImageVoodoo. Когда требуется более сложная обработка, к вашим услугам Java 2D API.

RMagick4J

Gem-пакет RMagick часто используется разработчиками на Ruby для генерации миниатюр или редактирования изображений, но для него необходима написанная на C библиотека ImageMagick. Пакет RMagick4J был создан для того, чтобы программисты на JRuby могли работать с привычным API и сохранять совместимость с существующими приложениями, в которых используется RMagick. Для начала установите gem-пакет RMagick4J:

```
$ jruby -S gem install rmagick4j
```

В примере 5.14 демонстрируется простая операция создания миниатюр. Здесь же показано, как простейший код, который распознает платформу и предварительно загружает нужный gem-пакет, позволяет обеспечить совместимость с RMagick.

Пример 5.14. Создание миниатюр с помощью пакета RMagick4J

```
require 'rubygems'  
gem defined?(JRUBY_VERSION) ? 'rmagick4j' : 'rmagick'  
require 'RMagick'  
include Magick
```

```
img = Image.new "avatar.jpg"  
thumb = img.resize(0.25)  
thumb.write "avatar-thumb.jpg"
```

В RMagick4J реализованы многие, но не все функции исходного gem-пакета RMagick. Впрочем, разработчики говорят, что в будущем планируют добиться полной совместимости с написанным на C пакетом RMagick.

ImageVoodoo

Библиотека ImageVoodoo написана членами команды разработчиком JRuby Томом Энебо и Ником Сигером. Изначально ставилась цель обеспечить совместимость на уровне API с библиотекой Райана Дэвиса ImageScience, еще одной широко распространенной в сообществе Ruby библиотекой для обработки изображений. Начните с установки gem-пакета ImageVoodoo:

```
$ jruby -S gem install image_voodoo
```

В примере 5.15 показано, как с помощью этой библиотеки создать миниатюру изображения.

Пример 5.15. Создание миниатюр с помощью ImageVoodoo

```
require 'image_voodoo'

ImageVoodoo.with_image('logo-240-480.jpg') do |img|
  img.thumbnail(240) do |img|
    img.save "logo-120-240.jpg"
  end
end
```

В gem-пакет ImageVoodoo входит файл *image_science.rb*, обеспечивающий совместимость с существующим кодом ImageScience. Открыв этот файл, вы увидите, что класс ImageScience просто ссылается на класс ImageVoodoo. В примере 5.16 показано, как можно заменить ссылки на ImageVoodoo в примере 5.15 кодом, эквивалентным ImageScience. Если использовать имя класса ImageScience, то код будет полностью переносим с интерпретатора C-Ruby на JRuby.

Пример 5.16. Использование класса ImageScience

```
require 'image_science'

ImageScience.with_image('logo-240-480.jpg') do |img|
  img.thumbnail(100) do |img|
    img.save "logo-120-240-imagescience.jpg"
  end
end
```

В каждой новой версии разработчики ImageVoodoo добавляют все новые средства обработки изображений, например, преобразование цветов, манипулирование яркостью и перевод изображения в серую шкалу. В примере 5.17 демонстрируется новый метод `from_url`, который загружает изображение из сети, после чего к нему применяется последовательность фильтров. Метод `preview` в этом примере открывает изображение в окне, это полезно для быстрой отладки и настройки параметров фильтров.

Пример 5.17. Дополнительные возможности ImageVoodoo

```
require 'image_voodoo'

ImageVoodoo.from_url("http://www.google.com/intl/en_ALL/images/logo.gif") do |img|
```

```
img.adjust_brightness(1.4,30) do |img3|
  img3.greyscale do |img4|
    img4.negative do |img5|
      img5.preview
    end
  end
end
end
end
```

Обсуждение

Для низкоуровневой или специализированной обработки изображений вы можете воспользоваться Java 2D API. Приведенный в примере 5.18 код порождает миниатюру очень высокого качества за счет применения техники сглаживания. Правда, за это приходится расплачиваться повышенной загрузкой процессора.

Пример 5.18. Генерация миниатюры с помощью Java 2D API

```
include Java

import java.awt.Image
import java.awt.image.BufferedImage
import java.awt.image.ConvolveOp

quality = 0.5
newWidth = 300
i = javax.swing.ImageIcon.new("source-image.jpg").image
newImg, i_w, i_h = nil, i.width, i.height

if (i_w > i_h)
  newImg = i.getScaledInstance(newWidth, (newWidth * i_h)/i_w, Image::SCALE_SMOOTH)
else
  newImg = i.getScaledInstance((newWidth * i_h)/i_w, newWidth, Image::SCALE_SMOOTH)
end
tmp =(javax.swing.ImageIcon.new(newImg)).image

# Создать объект BufferedImage для фильтра.
bufferedImage = BufferedImage.new(tmp.width,
tmp.height,BufferedImage::TYPE_INT_RGB)
g = bufferedImage.createGraphics()
g.color = java.awt.Color::white
g.fillRect(0, 0, tmp.width, tmp.height)
g.drawImage(tmp, 0, 0, nil)
g.dispose()

# Применить сглаживающий фильтр.
softFact = 0.05
softArray = [0, softFact, 0, softFact, 1-(softFact*4), softFact, 0, softFact, 0]
kernel = java.awt.image.Kernel.new(3, 3, softArray.to_java(:float))
```

```
op = ConvolveOp.new(kernel, ConvolveOp::EDGE_NO_OP, nil);
bufferedImage = op.filter(bufferedImage,nil)

# Записать в файл.
out = java.io.FileOutputStream.new("output.jpg")
encoder = com.sun.image.codec.jpeg.JPEGCodec::createJPEGEncoder(out)
param = encoder.getDefaultJPEGEncodeParam(bufferedImage)
param.setQuality(quality, true)
encoder.setJPEGEncodeParam(param)
encoder.encode(bufferedImage)
puts "finished"
```

См. также

- Сайт ImageScience <http://seattlerb.rubyforge.org/ImageScience.html>
- Сайт RMagick <http://rmagick.rubyforge.org/>
- Сайт RMagick4J <http://code.google.com/p/rmagick4j/>

5.8. Создание приложений SWT

Задача

Требуется написать на JRuby приложение, работающее с библиотекой SWT. Standard Widget Toolkit (SWT) – пожалуй, самая популярная после AWT и Swing технология создания клиентских приложений на языке Java. Исходный код этой библиотеки открыт, а самое известное приложение, написанное на ее основе, – пользовательский интерфейс Eclipse IDE.

Решение

Скачайте библиотеку SWT и включите файл *swt.jar* в путь поиска классов либо воспользуйтесь методом `require`, чтобы загрузить этот JAR-файл из Ruby-приложения. JRuby прекрасно интегрируется с классами `org.eclipse.swt.Shell` и `org.eclipse.swt.widgets.Display` и может обращаться ко всем виджетам, имеющимся в этой библиотеке. В примере 5.19 демонстрируется обработка кнопок в приложении SWT.

Пример 5.19. Просто приложение SWT, написанной на JRuby

```
include Java
require 'swt'

import org.eclipse.swt.SWT
import org.eclipse.swt.layout.RowLayout
import org.eclipse.swt.widgets.Listener

d = org.eclipse.swt.widgets.Display.new
```

```
s = org.eclipse.swt.widgets.Shell.new(d)
but = org.eclipse.swt.widgets.Button.new(s, SWT::PUSH)
but.text = "Search"

l = org.eclipse.swt.widgets.Label.new(s, SWT::NONE)
l.text = "Click to Search"
l.set_size(100,75)

but.addListener(SWT::Selection, Listener.impl do |method, evt|
  l.text = 'searching...'
end)

s.layout = RowLayout.new
s.set_size(300,200)
s.open

while(!s.is_disposed) do d.sleep if(!d.read_and_dispatch) end
d.dispose
```

Обсуждение

Проект Glimmer – это написанный на JRuby предметно-ориентированный язык (DSL) для создания приложений SWT с применением декларативного синтаксиса. Он создан Энди Малехом (Andy Maleh) и имеет статус официального проекта Eclipse. Установите gem-пакет Glimmer, выполнив следующую команду:

```
$ jruby -S gem install glimmer
```

В этом пакете содержится специализированный DSL для конструирования приложений SWT. Декларативный синтаксис этого языка устроен так, что графические контейнеры и отдельные компоненты определяются с помощью ключевых слов и сопутствующих им блоков. Модели виджетов и обработчики событий от них можно ассоциировать с методами Ruby, которые выполняют обработку событий и изменяют состояние. Ниже приведен пример использования языка Glimmer DSL.

Пример 5.20. Написание приложения SWT на языке Glimmer

```
include Java
require File.dirname(__FILE__) + "../src/swt"

include Glimmer
import 'org.eclipse.swt.layout.GridLayout'

def user_name
  "default text"
end

def enabled
```

```
    true
end

@shell = shell {
  text "SWT"
  composite {
    layout GridLayout.new(2, false) #two columns with differing widths
    label { text "Hello World!"}
    text {
      text bind(self, :user_name)
      enabled bind(self, :enabled)
    }
  }
}
@shell.open
```

См. также

- Рецепт 5.10 «Доступ к системному лотку»
- Рецепт 5.11 «Разработка приложений Swing на предметно-ориентированных языках на базе JRuby»
- Сайт SWT <http://www.eclipse.org/swt/>
- Сайт Glimmer <http://rubyforge.org/projects/glimmer/>

5.9. Доступ к рабочему столу

Задача

Требуется создать приложение для конкретной платформы или обмениваться данными с таким приложением.

Решение

Появившийся в версии Java 6 класс `java.awt.Desktop` позволяет получить доступ к ограниченному подмножеству наиболее употребительных функций, присущих платформенно-зависимому рабочему столу. Не покрывая все возможности рабочих столов, класс `Desktop` все же позволяет выполнять такие типичные операции, как открытие браузера, подразумеваемого по умолчанию, запуск умалчиваемого почтового клиента, а также открытие и печать файла в предпочтительном для него приложении (пример 5.21).

Пример 5.21. Использование Java Desktop API

```
include Java
import java.awt.Desktop
import java.net.URI
```

```
import java.io.File

d = Desktop::desktop

# Открыть браузер
d.browse(URI.new("http://www.ora.com/")) if d.isSupported(Desktop::Action::BROWSE)

# Открыть почтовый клиент и отправить сообщение
d.mail(URI.new("mailto:jruby@ora.com")) if d.isSupported(Desktop::Action::MAIL)

# Запустить приложение для просмотра jpg-файлов
d.open(File.new("conference_pic_1.jpg")) if d.isSupported(Desktop::Action::OPEN)

# Напечатать документ
d.print(File.new("directions.twxt")) if d.isSupported(Desktop::Action::PRINT)
```

5.10. Доступ к системному лотку

Задача

Требуется написать на JRuby приложение, которое будет работать в системном лотке Mac OS X, Windows или KDE.

Решение

Swing

Доступ к системному лотку Windows или Linux дает класс `java.awt.SystemTray`, появившийся в версии Java 6 (пример 5.22).

Пример 5.22. Java-приложение, работающее в системном лотке

```
include Java

import java.awt.TrayIcon
import java.awt.event.MouseListener

if (java.awt.SystemTray::isSupported())

  tray = java.awt.SystemTray::system_tray
  image = java.awt.Toolkit::default_toolkit.get_image("tray.gif")

  popup = java.awt.PopupMenu.new
  exititem = java.awt.MenuItem.new("Exit")
  exititem.addActionListener {java.lang.System::exit(0)}

  oraitem = java.awt.MenuItem.new("Go To ORA")
  oraitem.addActionListener do
    java.awt.Desktop::desktop.browse(java.net.URI.new("http://www.ora.com"))
  end
end
```

```

end

popup.add(exititem)
popup.add(oraitem)
trayIcon = TrayIcon.new(image, "Tray Demo", popup)
trayIcon.image_auto_size = true

trayIcon.addActionListener do |evt|
  trayIcon.displayMessage("Action", "Tray Action!", \
    TrayIcon::MessageType::WARNING)
end

trayIcon.addMouseListener(MouseListener.impl do |method, evt|
  puts "mouse event #{method.to_s}"
end

tray.add(trayIcon)
end

```

SWT

В библиотеке SWT также имеется класс для доступа к системному лотку: `org.eclipse.swt.widgets.Tray` (пример 5.23). Его преимуществом является доступность на платформах Windows, Linux и Mac. В реализации для OS X значок помещается в область состояния рабочего стола.

Пример 5.23. Приложение SWT, работающее в системном лотке

```

include Java
require 'swt-debug'

import org.eclipse.swt.SWT
import org.eclipse.swt.widgets.Listener
import org.eclipse.swt.widgets.MenuItem

d = org.eclipse.swt.widgets.Display.new
s = org.eclipse.swt.widgets.Shell.new(d)
image = org.eclipse.swt.graphics.Image.new(d, "tray.gif")
tray = d.system_tray
item = org.eclipse.swt.widgets.TrayItem.new(tray, SWT::NONE)
item.tool_tip_text = "SWT TrayItem"

item.addListener(SWT::DefaultSelection, Listener.impl do |evt|
  puts("default selection")
end)

menu = org.eclipse.swt.widgets.Menu.new(s, SWT::POP_UP)
menuitem = MenuItem.new(menu, SWT::PUSH)
menuitem.text = "Exit"

```

```
menuItem.addListener(SWT::Selection, Listener.impl do |method, evt|
  s.close
end)

item.addListener(SWT::MenuDetect, Listener.impl do |method, evt|
  menu.visible = true
end)

item.image = image
# исключаем эти параметры, чтобы скрыть главное окно
#s.setBounds(10, 10, 100,100)
#s.open()
while(!s.is_disposed) do d.sleep if(!d.read_and_dispatch) end
image.dispose
d.dispose
```

См. также

- Рецепт 5.8 «Создание приложений SWT»

5.11. Разработка приложений Swing на предметно-ориентированных языках на базе JRuby

Задача

Тенденция к декларативному проектированию ГИП проявляется в росте количества веб-приложений и переходе от традиционных технологий к таким декларативным моделям, как Adobe Flex или JavaFX Script. Вам тоже хочется применить основанный на JRuby предметно-ориентированный язык (DSL) к разработке приложений Swing.

Решение

Существует несколько проектов DSL-языков для разработки пользовательских интерфейсов на базе библиотеки Swing.

Swiby

Проект Swiby – это адаптация к JRuby той части языка JavaFX Script, которая относится к декларативному конструированию ГИП. Установите gem-пакет Swiby:

```
$ jruby -S gem install swiby
```

На синтаксис и дизайн Swiby оказал влияние язык JavaFX Script, в котором для представления иерархий контейнеров и компонентов применяются блоки.

Свойства определяются в однострочных объявлениях пар имя-значение. Имена классов Swing и AWT в DSL заменены более короткими. Swiby частично отказывается от характерного для JavaFX Script употребления заглавных букв в определениях виджетов и от двоеточий после объявлений свойств. В примере 5.24 язык Swiby демонстрируется в действии.

Пример 5.24. Простое приложение на Swiby

```
require 'rubygems'
require 'swiby'
require 'swiby/form'

class LabelModel
  attr_accessor :text
end
model = LabelModel.new
model.text = "Click to Search"

f = frame {
  title "Swiby Example"
  width 300
  height 100
  content {
    panel :layout => :flow do
      button("Search") { model.text="Searching...."}
      label {label bind(model, :text)}
    end
  }
}
f.visible = true
```

В проекте Swiby есть также некоторые черты, отсутствующие в JavaFX Script, например, возможность определять стили во внешнем файле. Для загрузки и применения стилей служит простое объявление `use_styles`. В примере 5.25 показано, как можно изменить шрифт путем создания и загрузки файла *styles.rb*.

Пример 5.25. Определение стилей в Swiby

```
swibyapp.rb

frame {
  title "Swiby Example"
  width 300
  height 74
  use_styles "styles.rb"
.
.
styles.rb

create_styles {
```

```

label(
  :font_family => Styles::VERDANA,
  :font_style => :italic,
  :font_size => 14,
  :color => 0xAA0000
)
}

```

В тот же gem-пакет включен полезный DSL для конструирования форм. Он ориентирован на формы с простым размещением элементов в ячейках сетки.

Cheri::Swing

Проект Cheri – это каркас, упрощающий создание предметно-ориентированных языков, в котором реализован паттерн «Построитель» (Builder) для создания иерархий объектов. Cheri::Swing представляет собой один из таких языков. Для начала установите gem-пакет Cheri:

```
$ jruby -S gem install cheri
```

Его декларативный синтаксис очень напоминает Swiby, он также предоставляет доступ к компонентам из библиотеки Swing и к изображениям и геометрическим упаковщикам из библиотеки AWT. В примере 5.26 представлено приложение, написанное на языке Cheri.

Пример 5.26. Простое приложение на языке Cheri::Swing

```

require 'rubygems'
require 'cheri/swing'
include Cheri::Swing

swing[:auto=>true]

f = frame('Cheri App') { |myframe|
  size 250,100
  flow_layout
  on_window_closing {|event| f.dispose}
  button('Search') {
    on_click {@l.set_text "Searching..."}
  }
  separator
  @l = label('Click to search')
}
f.visible = true

```

Если задать параметр `swing[:auto=>true]`, то от префикса `swing` в объявлениях компонентов можно будет отказаться: `swing.frame` сократится до `frame` и т.д.

Profligacy

Библиотеку Profligacy написал Зед Шоу (Zed Shaw); как мы вскоре увидим, в ней применяется иной подход, нежели в Cheri и Swiby. Но сначала установите gem-пакет Profligacy:

```
$ jruby -S gem install profligacy
```

Как видно из примера 5.27, Profligacy предлагает DSL, включающий целый ряд синтаксических улучшений, позволяющих экономить время.

Пример 5.27. Программа поиска для демонстрации библиотеки Profligacy

```
require 'rubygems'
require 'profligacy/swing'

class SearchDemo
  include_package 'javax.swing'
  include_package 'java.awt'
  include Profligacy

  def initialize
    @ui = Swing::Build.new JFrame, :search, :lab do |c,i|
      c.search = JButton.new "Search"
      c.lab = JLabel.new "Click to Search"
      i.search = { :action => proc {|t,e| c.lab.text = "Searching..." } }
    end

    @ui.layout = FlowLayout.new
    @ui.build("Layout").default_close_operation = JFrame::EXIT_ON_CLOSE
  end
end

SwingUtilities.invokeLater lambda { SearchDemo.new }
```

В Profligacy используется специальный язык верстки LEL, программа на котором напоминает уже позабытые рисунки, выполненные только с помощью ASCII-символов. А служит такой «рисунок» для определения макета, в котором позициям компонентов присваиваются имена (пример 5.28).

Пример 5.28. Демонстрация языка Profligacy LEL

```
require 'rubygems'
require 'profligacy/swing'
require 'profligacy/lel'

class LelSearchTest
  include_package 'javax.swing'
  include Profligacy

  layout = "
  [ search | _ ]
  [ _ | lab]
  "

  ui = Swing::LEL.new(JFrame,layout) do |c,i|
    c.search = JButton.new "Search"
  end
end
```

```
c.lab = JLabel.new "Click To Search"  
i.search= { :action => proc {|t,e| c.lab.text = "Searching..." } }  
end  
ui.build(:args => "LEL Search Example")  
end
```

Квадратными скобками обозначаются строки, а вертикальными черточками разделяются столбцы. На рис. 5.1 показана форма, получающаяся в результате выполнения пример 5.28.



Рис. 5.1. Пользовательский интерфейс, сконструированный демонстрационной программой

См. также

- Сайт Cheri <http://cheri.rubyforge.org/>
- Сайт Swiby <http://swiby.codehaus.org/>
- Сайт Profligacy <http://ihate.rubyforge.org/profligacy/>

5.12. Использование библиотеки Monkeybars для разработки приложений Swing

Задача

Требуется разработать приложение Swing, придерживаясь паттерна модель-вид-контроллер (MVC).

Решение

Воспользуйтесь библиотекой Monkeybars, которую написал Дэвид Кунц, автор Rawr. В ней для разработки клиентских приложений на JRuby применяется паттерн проектирования MVC, аналогично тому, как это делается в веб-каркасах Rails и Struts. Сначала установите gem-пакет Monkeybars:

```
$ jruby -S gem install monkeybars
```

Этот пакет добавляет инструмент Monkeybars в путь исполнения JRuby. Здесь прослеживается аналогия с командой *rails*, знакомой разработчикам на платформе Ruby on Rails. В результате выполнения команды *monkeybars* создается главная папка и заготовка проекта:

```
$ jruby -S monkeybars search_demo
```

В примере 5.29 показан Java-класс, которым мы будем пользоваться при изучении Monkeybars. Он расширяет класс JFrame и содержит кнопку, снабженную надписью. Этот файл должен находиться в каталоге *src*.

Пример 5.29. Java-класс пользовательского интерфейса Java для использования совместно с Monkeybars

```
import javax.swing.*;

public class SearchDemoJava extends JFrame {
    private JLabel message = new JLabel("Click to search");
    private JButton search = new JButton("Search");

    public SearchDemoJava() {
        this.setLayout(new java.awt.FlowLayout());
        this.setSize(300,100);
        add(search);
        add(message);
    }
}
```

Код обработки событий и данные модели записаны на Ruby. Rake-задание *generate*, которое было добавлено вместе с JAR-файлом Monkeybars и несколькими классами Ruby на этапе генерации проекта, применяется для создания классов модели, вида и контроллера. Чтобы создать все три класса сразу, укажите параметр *ALL*:

```
$ cd search_demo
$ jruby -S rake generate ALL="src/search"
(in C:/projects/search_demo)
Generating controller SearchController in file search_controller.rb
Generating model SearchModel in file search_model.rb
Generating view SearchView in file search_view.rb
```

В классе модели имеется переменная экземпляра, в которой хранятся сообщения, появляющиеся в текстовой метке (пример 5.30). Позже эта переменная отображается на компонент ГИП в ассоциированном файле вида.

Пример 5.30. Файл модели Monkeybars

```
class SearchModel
  attr_accessor :search_message
  def initialize
    @search_message = "Starting"
  end
end
```

Откройте файл `search_view.rb` и назначьте класс `SearchDemoJava` в качестве компонента отображения вида, для чего следует вызвать метод `set_java_class`. С помощью метода `map` свяжите переменную экземпляра модели со свойством `text` метки; в результате любые изменения в классе модели будут отображаться в этом компоненте вида. Модифицированный класс вида показан в примере 5.31.

Пример 5.31. Класс вида Monkeybars

```
class SearchView < ApplicationView
  set_java_class 'SearchDemoJava'
  map :model => :search_message, :view => "message.text"
end
```

Класс контроллера отвечает за определение объектов вида и модели, обработку событий и управление состоянием приложения. Открыв файл `search_controller.rb`, вы увидите, что генератор уже определил классы вида и модели. Однако добавить функцию обработки событий от кнопки вам придется самостоятельно. Контроллер поиска перехватывает события, возбуждаемые видом, и направляет их функции, имя которой составлено из имени переменной экземпляра, представляющей источник события (`search`), и записанного строчными буквами Java-типа события (`action_performed`). Это еще одна идея, заимствованная у Rails, и проявление принципа примата соглашения над конфигурацией. В примере 5.32 показан модифицированный класс контроллера.

Пример 5.32. Класс контроллера Monkeybars

```
class SearchController < ApplicationController
  set_model 'searchModel'
  set_view 'SearchView'
  set_close_action :exit

  def search_action_performed
    model.search_message = "Searching..."
    update_view
  end
end
```

Обратите внимание, что новое текстовое значение присваивается в модели, написанной на Ruby, а не в компоненте на Java. Метод `update_view` перерисовывает компоненты ГИП, то есть заново вычисляет отображения модели на вид и выводит новое сообщение.

Установите gem-пакет `Rawr` и выполните команду `rawr install`, находясь в корневом каталоге проекта. Откройте файл `src/main.rb` и добавьте в свое приложение новую точку подключения, создав экземпляр класса контроллера (пример 5.33).

Пример 5.33. Главный исполняемый файл Monkeybars

```
begin
  # Сюда помещается логика вашего приложения, т.е. вызов метода
  YourController.instance.open
```

```
require 'search_demo/search_controller'
SearchController.instance.open
rescue Exception => e
```

Скачайте или сами соберите файл *jruby-complete.jar* и поместите его в каталог *lib/java*. Оформите пакет приложения в виде исполняемого JAR-файла, выполнив Rake-задание `rawr:jar` из корневого каталога проекта:

```
$ jruby -S rake rawr:jar
```

По умолчанию созданный JAR-файл записывается в каталог *package/deploy*. Вы можете изменить имя результирующего JAR-файла, отредактировав конфигурационный файл Rawr *build_configuration.yml*. Протестируйте приложение, выполнив JAR-файл:

```
$ java -jar package/deploy/change_me.jar
```

Обсуждение

В примере 5.31 показано применение компонента ГИП, написанного на Java, но вы можете использовать для генерации пользовательского интерфейса также JRuby или какой-нибудь другой каркас. Класс, определенный в примере 5.34, по существу ничем не отличается от того, что мы видели в примере 5.29.

Пример 5.34. Компонент ГИП, написанный на JRuby

```
include Java

class SearchDemoRuby < javax.swing.JFrame
  attr_accessor :search, :message
  def initialize
    super
    self.layout = java.awt.FlowLayout.new
    add(@search = javax.swing.JButton.new("search"))
    add(@message = javax.swing.JLabel.new("Click to Search"))
    self.set_size(300,100)
  end
end
```

Этот пример убедительно доказывает, что слабая связанность компонентов позволяет легко подменить уровень вида. Файл вида – это единственный файл MVC-части приложения, который вообще нужно модифицировать. Поддержка, предоставляемая Monkeybars для компонентов, написанных на Ruby, чуть менее элегантна, чем поддержка для Java, но ожидается, что в будущих версиях она улучшится. Удалите старое объявление `set_java_class` и запишите в переменную `@main_view_component` новый экземпляр класса, описывающего компонент ГИП на Ruby. Это показано в примере 5.35. Не забудьте вызвать конструктор родительского класса при переопределении подразумеваемого по умолчанию конструктора вида.

Пример 5.35. Класс вида Monkeybars, в котором используется компонент ГИП, написанный на JRuby

```
class SearchView < ApplicationView
# set_java_class "SearchDemoJava"
  def initialize
    super
    @main_view_component = SearchDemo.new
  end
  def search
    @main_view_component.search
  end
  def message
    @main_view_component.message
  end
  map :model => :search_message, :view => "message.text"
end
```

Кроме того, необходимо изменить файл *main.rb*, так чтобы он загружал написанный на JRuby класс ГИП:

```
begin
# Сюда помещается логика вашего приложения, т.е. вызов метода
YourController.instance.open
  require 'search_demo_ruby'
  require 'search_demo/search_controller'
  SearchController.instance.open
rescue Exception => e
```

Проделав все это, можно сгенерировать с помощью `Rawt` новый исполняемый JAR-файл и протестировать свое приложение.

См. также

- Начальная страница Monkeybars <http://monkeybars.rubyforge.org/>
- Рецепт 5.4 «Пакетирование автономных приложений»

5.13. Создание приложений Qt с помощью JRuby

Задача

Требуется применить JRuby для написания приложений на базе графической библиотеки. Qt – это популярная кросс-платформенная библиотека для разработки графических интерфейсов пользователя. Она располагает обширным набором компонентов, в том числе виджетами Web Browser и System Tray.

Решение

Проект Qt Jambi дает разработчикам возможность работать с библиотекой Qt из языка Java. Скачать Qt Jambi можно со страницы <http://trolltech.com/downloads/>. Загрузив дистрибутив, относящийся к нужной вам платформе, добавьте файлы *qtjambi-version.jar* и *qtjambi-platformversion.jar* в путь поиска классов.

В библиотеке Qt::JRuby имеется несколько удобных средств интеграции, которые полезны при работе с библиотекой Qt Jambi напрямую из JRuby, в том числе предметно-ориентированный язык для Qt. Qt::JRuby необходимо собрать самостоятельно из исходных текстов. Прежде всего получите последнюю версию Qt::JRuby из Git-репозитория. Затем с помощью системы Rake соберите архив *qtjruby-core.jar* и установите обертывающий gem-пакет:

```
$ git clone git://github.com/nmerouze/qtjruby.git
Initialize qtjruby/.git
Initialized empty Git repository in /home/henry/qtjruby/.git/
remote: Counting objects: 391, done.
remote: Compressing objects: 100% (182/182), done.
Receiving objects: 100% (391/391), 59.30 KiB | 78 KiB/s, done.
Resolving deltas: 100% (180/180), done.
$ cd qtjruby/qtjruby-core
$ jruby -S rake
(in /home/henry/qtjruby/qtjruby-core)
ant -lib /opt/jruby-1.1.2/bin/../../lib
Buildfile: build.xml
qtjruby-core:
[javac] Compiling 14 source files to C:\home\devel\qtjruby\qtjruby-core\build
[javac] Note: Some input files use unchecked or unsafe operations.
[javac] Note: Recompile with -Xlint:unchecked for details.
[jar] Building jar: C:\home\devel\qtjruby\qtjruby-core\lib\qtjruby-core.jar
BUILD SUCCESSFUL
Total time: 1 second
WARNING: no rubyforge_project specified
WARNING: RDoc will not be generated (has_rdoc == false)
Successfully built RubyGem
Name: qtjruby-core
Version: 0.2.0
File: qtjruby-core-0.2.0.gem
/opt/jruby-1.1.2/bin/../../bin/jruby -S gem install pkg/qtjruby-core-0.2.0.gem
Successfully installed qtjruby-core-0.2.0
1 gem installed
```

В библиотеку Qt::JRuby входит Ruby-модуль Qt, позволяющий ссылаться на классы из Qt Jambi, не указывая имя пакета или префикс Q. Например, на класс `com.trolltech.qt.gui.QpushButton` можно сослаться просто по имени `Qt::PushButton`. Конечно, это мелкая деталь, но код при этом становится яснее и удобнее для восприятия. Кроме того, библиотека отображает сигналы Qt на блоки, применяя ту

же технику, что в обработчиках событий из библиотеки JRuby Swing. В примере 5.36 приведено простое приложение Qt::JRuby.

Пример 5.36. Приложение Qt::JRuby

```
Qt::Application.initialize(ARGV)
window = Qt::Widget.new
window.resize(300, 200)
l = Qt::HBoxLayout.new
window.window_title = 'QTJRuby Example'
window.layout = l

quit = Qt::PushButton.new("Search", window)
quit.font = Qt::Font.new("Times", 14, Qt::Font::Weight::Bold.value)

searchlab = Qt::Label.new("Click to Search", window)
quit.clicked { searchlab.text = "Searching..." }

l.add_widget quit
l.add_widget searchlab

window.show
Qt::Application.exec
```

Запустите приложение, выполнив следующую команду:

```
$ jruby -S qtjruby qt_search_demo.rb
```

Обсуждение

Можно обойтись и без вызова команды `qtjruby`, если включить в свое приложение содержащий ее файл `qtjruby-core.jar` и код загрузки gem-пакета. Это бывает полезно, когда нужно оформить свой код в виде приложения, допускающего дальнейшее распространение:

```
require 'qtjruby-core'

gem_path = Qt::JRuby.root / 'gems'
if File.exist? gem_path
  Gem.clear_paths
  Gem.path.unshift(gem_path)
end

Qt::Application.initialize(ARGV)
window = Qt::Widget.new
...
```

Такую программу можно запустить непосредственно:

```
$ jruby qt_search_demo.rb
```

В настоящее время ведется работа над DSL-языком для Qt JRuby. Существующая версия считается экспериментальной, поэтому в будущем API может измениться. Соберите и установите gem-пакет `qtjruby-dsl`:

```
$ cd qtjruby/qtjruby-dsl
$ jruby -S rake
```

Возможности этого DSL-языка можно продемонстрировать на примере виджета `browser`, входящего в дистрибутив `Qt::JRuby` (пример 5.37). Как и в других DSL-языках, имена компонентов укорочены, а для представления отношений между контейнерами и компонентами, равно как и для обработки событий, используются блоки.

Пример 5.37. Экспериментальный DSL для Qt::JRuby

```
require 'rubygems'
require 'qtjruby-dsl'

Qt.app do
  window :id => 'main' do
    create :browser_win, :type => :browser
    create :le_address, :type => :line_edit

    hbox do
      le_address
      button('Go').clicked do
        browser_win.load le_address.text
      end
    end

    browser_win.load 'http://www.ora.com'
  end
end
```

См. также

- Сайт Qt Jambi <http://trolltech.com/products/qt/features/language-support/java>
- Блог Qt::JRuby <http://qtjruby.org/blog>
- Сайт Git <http://git.or.cz/>

Инструменты сборки

6.0. Введение	182
6.1. Включение Ruby-сценариев в процесс сборки системой Ant	182
6.2. Применение Ruby в условных конструкциях Ant	185
6.3. Написание задания Ant на Ruby	187
6.4. Включение Ruby-сценариев в процесс сборки системой Maven	188
6.5. Написание подключаемого к Maven модуля на JRuby	190
6.6. Сборка Java-проектов с помощью Raven	193
6.7. Ссылка на библиотеки в Raven	195
6.8. Организация частного репозитория Raven	196
6.9. Прогон тестов JUnit с помощью Raven	197
6.10. Сборка Java-проектов с помощью Buildr	198
6.11. Ссылка на библиотеки в Buildr	201
6.12. Сборка с помощью Rake в контексте сервера Hudson	202
6.13. Добавление Ruby-сценария в качестве задачи сервера Hudson	203

6.0. Введение

Едва ли не каждый программный проект, вне зависимости от языка или размера, нужно тем или иным способом *собирать*. Процесс сборки может включать в частности такие шаги, как компиляция кода, прогон автоматизированных тестов, обработку файлов, пакетирование и развертывание. Поскольку все процессы сборки в чем-то похожи, разработано немало специализированных *систем сборки*. Подобная система описывает процесс сборки в виде последовательности взаимосвязанных, повторно используемых заданий. К примеру, система Ant позволяет заменить такие команды:

```
$ javac *.java
$ jar -cf my.jar *.class
```

одной:

```
$ ant jar
```

Или даже просто (если цель `jar` принимается по умолчанию):

```
$ ant
```

В настоящей главе рассматриваются различные способы сборки проектов на Java. В этом плане Ruby можно использовать либо как основной инструмент сборки, либо как дополнение к существующему процессу. Для сборки Java-проектов чаще всего применяется одна из двух систем: Ant и Maven. Обе являются проектами фонда Apache Software Foundation и обладают механизмами расширения для поддержки JRuby. На них мы и остановимся в нескольких первых рецептах. Далее будут описаны две написанные целиком на Ruby системы сборки Java-проектов: Raven и Buildr. У каждой из четырех систем есть свои достоинства; какой из них пользоваться в конкретном проекте – в значительной степени дело вкуса. Raven и Buildr появились гораздо позже, чем Ant и Maven, поэтому сформировавшееся вокруг них сообщество пока насчитывает меньше членов.

Мы закончим эту главу двумя рецептами, относящимися к серверу непрерывной интеграции Hudson. В первом описывается, как собирать Ruby-проекты с помощью системы сборки Rake. Во втором рассматривается вопрос об использовании Ruby для написания дополнительных сценариев, вызываемых сервером Hudson в процессе сборки.

6.1. Включение Ruby-сценариев в процесс сборки системой Ant

Задача

Вы пользуетесь для сборки системой Apache Ant, но хотите включить в процесс сборки некую логику, которую трудно или невозможно реализовать средствами языка Ant XML.

Решение

Добавьте в каталог *lib* Ant необходимые для JRuby зависимости и воспользуйтесь заданием `script`, чтобы включить написанный на Ruby код в файл сборки Ant. В примере 6.1 продемонстрировано очень простое применение такой техники.

Пример 6.1. Печать строки «Hello World» с помощью вызова JRuby из Ant

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="package">
  <target name="simple">
    <script language="ruby">
      print "Hello World!"
    </script>
  </target>
</project>
```

Обсуждение

Это задание поддерживает спецификации Bean Scripting Framework (BSF) и Java Scripting (JSR 223), которые обсуждались в главе 3, поэтому сценарии можно писать на самых разных языках, а не только на Ruby. Но чтобы этим заданием воспользоваться, необходимо предоставить в распоряжение Ant соответствующие зависимости. В случае BSF это файлы *jruby.jar* и *bsf.jar*, которые включены в дистрибутив JRuby и находятся в каталоге *lib*. В случае Java Scripting понадобится файл *jruby.jar* из дистрибутива JRuby и файл *jruby-engine.jar*, который можно скачать с сайта <https://scripting.dev.java.net/>. В рецептах 3.2 и 3.3 приведена дополнительная информация об этих API. Выше, в разделе «Решение» отмечалось, что эти JAR-файлы можно поместить в каталог Ant *lib*. Но можно поступить и по-другому: объявить зависимости в файле сборки Ant, как показано в примере 6.2. Этот способ требует чуть больше усилий на конфигурирование, так как необходимо задать свойства Ant: `jruby.home` и `jsr223.engines.home` (см. пример 6.2). В данном случае указанные свойства определены в файле *build.properties*, который находится в начальном каталоге пользователя.

Пример 6.2. Определение зависимостей JRuby в файле сборки Ant

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="package">

  <property file="${user.home}/build.properties" />
  <path id="jruby">
    <fileset file="${jruby.home}/lib/jruby.jar" />
    <fileset file="${jsr223.engines.home}/lib/jruby-engine.jar" />
  </path>
  <target name="simple">
    <script language="ruby" classpathref="jruby">
      print «Hello #{$project.getProperty('user.name')}»
    </script>
  </target>
</project>
```

```

    </script>
  </target>
</project>

```

В примере 6.2 также видно, что объект, представляющий проект Ant, доступен в коде на Ruby в виде глобальной переменной `$project`. Помимо самого проекта, доступны также все определенные в нем свойства, ссылки и цели. Однако нередко (и в частности в примере 6.2) имя свойства содержит точки. В таких случаях для получения значения свойства нужно воспользоваться методом `getProperty()`. Если бы имя пользователя было представлено свойством Ant `user_name`, то можно было бы написать короче:

```
print "Hello #{$user_name}"
```

Цель, определенную в файле сборки Ant, можно выполнить, вызвав ее метод `execute`. В примере 6.3 демонстрируется применение Ruby внутри Ant для выражения сложного условия. В данном случае мы хотим выполнить на этапе развертывания дополнительный шаг при условии, что сборка производится в среде непрерывной интеграции (Continuous Integration – CI), причем CI-сервером является Hudson. Все параметры передаются в файл сборки Ant в виде свойств, которые затем анализируются в Ruby-сценарии.

Пример 6.3. Вызов цели Ant из Ruby

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="package">

  <property name="src.dir" value="${basedir}/src" />
  <property name="output.dir" value="${basedir}/bin" />
  <property name="deploy.dir" value="${basedir}/deploy" />
  <property name="output.file" value="${output.dir}/package.zip" />

  <target name="init">
    <mkdir dir="${output.dir}" />
    <mkdir dir="${deploy.dir}" />
  </target>

  <target name="package" depends="init">
    <zip destfile="${output.file}">
      <fileset dir="${src.dir}" />
    </zip>
    <script language="ruby" classpathref="jruby">
      <![CDATA[
        if ($cibuild == "true") && ($ciserver == "Hudson") then
          $deploy.execute()
        end
      ]]>
    </script>
  </target>

  <target name="deploy">
    <echo>Deploying file ${output.file}</echo>

```

```
<copy file="${output.file}" todir="${deploy.dir}"/>
</target>
</project>
```

Из кода на Ruby можно обращаться и к другим сценариям или библиотекам. Например, описанный в примере 6.3 шаг развертывания можно было бы выполнить прямо из Ruby-кода, воспользовавшись модулем FileUtils из стандартной библиотеки Ruby:

```
<![CDATA[
require 'fileutils'
if ($cibuild != true) && ($ciserver == 'Hudson') then
  puts "Deploying file #{$project.getProperty('output.file')}..."
  FileUtils.cp $project.getProperty("output.file"),
    $project.getProperty("deploy.dir")
end
]]>
```

Чтобы этот код заработал, необходимо задать системное свойство `jruby.home`, например, с помощью переменной среды `ANT_OPTS`. На платформе Windows нужно было бы выполнить команду:

```
set ANT_OPTS=-Djruby.home=%JRUBY_HOME%
```

а в Linux или Mac OS X – команду:

```
export ANT_OPTS=-Djruby.home="$JRUBY_HOME"
```

И последнее замечание – вовсе необязательно встраивать Ruby-сценарий прямо в файл сборки внутри задания `script`. В этом задании можно указать атрибут `src`, содержащий путь к сценарию. Что предпочесть – встроенный сценарий или внешний файл, зависит главным образом от длины: если код насчитывает больше 10 строк, то, наверное, имеет смысл вынести его во внешний файл. Сценарии во внешних файлах полезны и в том случае, когда один и тот же кусок кода используется в нескольких файлах сборки Ant.

6.2. Применение Ruby в условных конструкциях Ant

Задача

В файле сборки Ant встречается условная конструкция, которую удобнее всего выразить на Ruby.

Решение

Задайте для Ant путь поиска классов, как описано в рецепте 6.1, и воспользуйтесь условным элементом `scriptcondition`. Этот элемент очень похож на задание

script, которое мы рассматривали в рецепте 6.1. Основное различие между ними заключается в том, что перед выполнением сценария вычисляются булевские условия. Обычно для условия задается некое значение по умолчанию, которое при необходимости можно переопределить. Так, в примере 6.4 Ant присвоит свойству `user_has_text_files` значение `true`, если в начальном каталоге пользователя есть хотя бы один текстовый файл.

Пример 6.4. Использование элемента `scriptcondition`

```
<target name="setup">
  <condition property="user_has_text_files">
    <scriptcondition language="ruby" value="false">
      cwd = Dir.pwd
      Dir.chdir $project.getProperty("user.home")
      $self.setValue(true) if Dir.glob("**/*.txt")
      Dir.chdir cwd
    </scriptcondition>
  </condition>
</target>
```

В этом примере условие по умолчанию равно `false`. Однако в результате анализа переменной `$self`, которая представляет объект условия, ему может быть присвоено значение `true`. Как и в случае задания `script`, рассмотренного в рецепте 6.1, переменная `$project` содержит объект `Project`, представляющий проект Ant, и все свойства Ant тоже доступны Ruby-сценарию в виде переменных.

Обсуждение

Условия в Ant можно объединять с помощью элементов `and`, `or`, `not` или `xor`. В примере 6.5 демонстрируется комбинация условия из примера 6.4 с одним из встроенных в Ant условий, `os`. В данном случае условие `user_has_text_files` проверяется лишь на платформе Windows.

Пример 6.5. Комбинирование условия `scriptcondition` с другими условиями Ant

```
<target name="setup">
  <condition property="user_has_text_files">
    <and>
      <os family="windows"/>
      <scriptcondition language="ruby" value="false">
        puts "hello"
        cwd = Dir.pwd
        Dir.chdir $project.getProperty("user.home")
        $self.setValue(true) if Dir.glob("**/*.txt")
        Dir.chdir cwd
      </scriptcondition>
    </and>
  </condition>
</target>
```

В Ant при вычислении составных условий применяется «закорачивание», то есть второе (а также третье, четвертое и т.д.) условие вычисляется лишь при необходимости. Например, если задание, показанное в примере 6.5, запущено не на платформе Windows, то Ruby-код вообще не выполняется, так как вычисление первого условия (`<os family=windows>/>`) дает `false`. Об этом стоит помнить, так как иногда вычисление одних условий занимает гораздо больше времени, чем других.

6.3. Написание задания Ant на Ruby

Задача

Требуется выполнить один и тот же Ruby-сценарий в нескольких файлах сборки Ant.

Решение

Воспользуйтесь элементом Ant `scriptdef`, в котором определите задание, состоящее в выполнении Ruby-сценария. У элемента `scriptdef` имеется дочерний элемент `attribute`, позволяющий передавать заданию атрибуты. В примере 6.6 определено задание Ant `start-webrick`, которое можно использовать для запуска HTTP-сервера WEBrick с заданными номером порта и корневой папкой документов.

Пример 6.6. Использование элемента `scriptdef` для определения задания Ant

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="start">

  <scriptdef name="start-webrick" language="ruby">
    <attribute name="port"/>
    <attribute name="root"/>
    <![CDATA[
      require 'webrick'
      include WEBrick
      server = HTTPServer.new(:Port => $attributes.get('port').to_i)
      server.mount("/", HTTPServlet::FileHandler, $attributes.get('root'))
      server.start
    ]]>
  </scriptdef>

  <target name="start">
    <start-webrick port="8000" root="${basedir}/files"/>
  </target>

</project>
```

6.4. Включение Ruby-сценариев в процесс сборки системой Maven

Задача

Вы пользуетесь для сборки системой Apache Maven и хотите быстро добавить в процесс сборки какие-то дополнительные шаги.

Решение

Включите в файл определения проекта Maven, *pom.xml*, подключаемый модуль JRuby Maven, как показано в примере 6.7. В данном случае цель подключаемого модуля `run`, в которой выполняется Ruby-сценарий, связана с фазой `process-resources`. Это означает, что встроенный Ruby-сценарий будет выполнен до начала компиляции или прогона тестов.

Пример 6.7. Использование подключаемого модуля JRuby Maven

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jrubycookbook</groupId>
  <artifactId>maven-sample</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jruby-maven-plugin</artifactId>
        <executions>
          <execution>
            <phase>generate-resources</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <ruby>
                require 'fileutils'
                FileUtils.touch 'target/timestamp'
              </ruby>
            </configuration>
          </execution>
        </executions>
        <!-- Необходимо из-за ошибки в дистрибутиве JRuby's Maven -->
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
```

```
        <groupId>backport-util-concurrent</groupId>
        <artifactId>backport-util-concurrent</artifactId>
        <version>3.0</version>
    </dependency>
    <dependency>
        <groupId>asm</groupId>
        <artifactId>asm-all</artifactId>
        <version>2.2.3</version>
    </dependency>
</dependencies>
</plugin>
</plugins>
</build>
</project>
```

Обсуждение

Можно также выполнить сценарий, находящийся во внешнем файле; для этого вместо элемента `ruby` воспользуйтесь элементом `script`:

```
<configuration>
  <script>src/main/scripts/touch_timestamp.rb</script>
</configuration>
```

Как и в Ant, для настройки пути загрузки Ruby этот подключаемый модуль пользуется системным свойством `jruby.home`. Но, если в Ant для передачи системных свойств служит переменная среды `ANT_OPTS`, то Maven просматривает переменную среду `MAVEN_OPTS`. В Windows для ее задания следует выполнить команду:

```
set MAVEN_OPTS=-Djruby.home="%JRUBY_HOME%"
```

А в Linux или Mac OS X – команду:

```
export MAVEN_OPTS=-Djruby.home="$JRUBY_HOME"
```

По умолчанию путь загрузки для сценариев, выполняемых внутри подключаемого модуля Maven, состоит из следующих путей относительно каталога, заданного в системном свойстве `jruby.home`:

- `lib/ruby/site_ruby/1.8`
- `lib/ruby/site_ruby`
- `lib/ruby/1.8`
- `lib/ruby/1.8/java`

В этот список можно добавить и другие пути, для чего служит элемент `libraryPaths`:

```
<configuration>
  <script>src/main/scripts/touch_timestamp.rb</script>
  <libraryPaths>
    <libraryPath>${user.home}/ruby/lib</libraryPath>
  </libraryPaths>
</configuration>
```

Недостатком этого подключаемого модуля служит то, что последняя на момент написания этой книги версия (1.0-beta-4) написана в расчете на устаревшую версию JRuby 0.9.9. Поинтересуйтесь, не появилась ли на сайте более свежая версия.

См. также

- Сайт подключаемого модуля JRuby Maven <http://mojo.codehaus.org/jruby-maven-plugin/>
- Сайт Apache Maven <http://maven.apache.org/>

6.5. Написание подключаемого к Maven модуля на JRuby

Задача

Вы работаете с системой сборки Apache Maven и хотите использовать один и тот же Ruby-сценарий в нескольких проектах. Примером может служить использование Ruby-библиотеки RedCloth для генерации проектной документации из исходных файлов, размеченных в соответствии с языком Textile.

Решение

Создайте новый проект подключаемого модуля Maven и добавьте обсуждавшиеся в рецепте 6.4 зависимости в проект и в модуль `maven-plugin-plugin`. Ниже приведен пример простого дескрипторного файла `pom.xml`.

Пример 6.8. Файл `pom.xml` для системы Maven с определением подключаемого модуля на JRuby

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jrubycookbook</groupId>
  <artifactId>maven-textile-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Textile Plugin</name>
  <description>
    Генерирует онлайн-документацию из исходных Textile-файлов с помощью
    RedCloth.
  </description>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jruby-maven-plugin</artifactId>
      <version>1.0-beta-4</version>
```

```
</dependency>
<dependency>
  <groupId>backport-util-concurrent</groupId>
  <artifactId>backport-util-concurrent</artifactId>
  <version>3.0</version>
</dependency>
<dependency>
  <groupId>asm</groupId>
  <artifactId>asm-all</artifactId>
  <version>2.2.3</version>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-plugin-plugin</artifactId>
      <dependencies>
        <dependency>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>jruby-maven-plugin</artifactId>
          <version>1.0-beta-4</version>
        </dependency>
        <dependency>
          <groupId>backport-util-concurrent</groupId>
          <artifactId>
            backport-util-concurrent
          </artifactId>
          <version>3.0</version>
        </dependency>
        <dependency>
          <groupId>asm</groupId>
          <artifactId>asm-all</artifactId>
          <version>2.2.3</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
</project>
```

Установите gem-пакет RedCloth:

```
$ gem install RedCloth
```

Затем создайте в каталоге *src/main/scripts* Ruby-класс, который расширяет класс *Mojo*, входящий в подключаемый модуль *jruby-maven-plugin*. Как и подключаемые к Maven модули, написанные на Java, этот класс аннотирован именем цели и может быть параметризован. Поскольку Ruby – динамически типизированный язык, необходимо явно задать тип параметра, воспользовавшись атрибутом *type*

в аннотации `parameter`. Определив все параметры, можно закодировать логику подключаемого модуля в методе `execute`. В примере 6.9 определена цель `generate`, которая ищет Textile-файлы и преобразует их в формат HTML с помощью `RedCloth`.

Пример 6.9. Подключаемый к Maven модуль, написанный на Ruby

```
include Java

require 'rubygems'
gem 'RedCloth'
require 'redcloth'

# Этот подключаемый модуль преобразует все Textile-размеченные файлы в формат HTML
# @goal "generate"
class GenerateMojo < Mojo

  # @parameter type="java.io.File" default-value="${basedir}/src/main/site/textile"
  def sourceDirectory;;end

  # @parameter type="java.io.File" default-value="${basedir}/target/site"
  def outputDirectory;;end

  def execute
    $outputDirectory.mkdirs

    Dir.chdir $sourceDirectory.absolutePath

    Dir.glob("*.tx") do |entry|
      info "Opening #{entry}"
      open(entry) { |f| @contents = f.read }
      r = RedCloth.new @contents
      # получить имя файла без расширения
      short_name = entry.slice(0, entry.length - 3)
      out = java.io.File.new($outputDirectory, "#{short_name}.html").absolutePath
      info "Writing to #{out}"
      open(out, 'w') { |f|
        f.puts "<html><body>"
        f.puts r.to_html
        f.puts "</body></html>"
      }
    end
  end
end

run_mojos GenerateMojo
```

Обсуждение

Помимо аннотации `@goal`, встречающейся в примере 6.9, подключаемый модуль JRuby Maven поддерживает аннотации уровня класса, которые допустимы в модулях, написанных на Java. Аннотация `@phase` позволяет связать модуль с опреде-

ленной фазой процесса сборки Maven, аннотация `@requiresProject false` – запустить модуль без проекта Maven и так далее. Полный перечень аннотаций опубликован на сайте Maven. К сожалению, в текущей версии не работает функция автоматизированного документирования подключаемого модуля. Для модулей, написанных на Java, Maven умеет на основе имеющихся аннотаций генерировать документацию для каждой цели, но для модулей на JRuby это пока не реализовано.

Как и в примерах из рецепта 6.4, необходимо с помощью переменной среды `MAVEN_OPTS` задать системное свойство `jruby.home`. Если gem-пакеты не находятся в каталоге, подразумеваемом по умолчанию, то необходимо также указать их местоположение с помощью переменной среды `GEM_HOME`.

Читатели, знакомые с подключаемыми модулями Maven, вероятно, заметили, что код в примере 6.9 не является отчетом Maven и на самом деле не будет вызываться в процессе генерирования сайта с помощью Maven. Модули отчетов Maven предъявляют дополнительные требования к локализации, дописывание необходимого кода мы оставляем читателю в качестве упражнения.

См. также

- Центр разработки подключаемых модулей для Maven <http://maven.apache.org/plugin-developers/index.html>
- Сайт JRuby Maven Plugin <http://mojo.codehaus.org/jruby-maven-plugin/>

6.6. Сборка Java-проектов с помощью Raven

Задача

Требуется собрать Java-проект, но вы хотите написать сценарий сборки на Ruby, а не на XML.

Решение

Используйте Raven, инструмент сборки Java-проектов, построенный на базе программы Rake, написанной на Ruby. По существу, Raven представляет собой надстройку над Rake, в которой определены дополнительные Rake-задания для сборки Java-проектов. Raven распространяется в виде gem-пакета, поэтому для установки достаточно выполнить следующую команду:

```
$ gem install raven
```

Чтобы воспользоваться Raven, создайте файл *Rakefile* в корне своего проекта и включите в него все необходимые задания. В примере 6.10 приведен простейший сценарий сборки на Raven.

Пример 6.10. Простой сценарий сборки на Raven

```
require 'raven'  
javac 'compile'
```

Для запуска этого сценария выполните команду:

```
$ rake compile
```

Или:

```
$ jruby -S rake compile
```

В результате, следуя соглашениям, принятым в проекте Maven, будут откомпилированы все Java-файлы в каталоге *src/main/java* (см. врезку ниже). Это подразумеваемое по умолчанию местоположение легко переопределить, как видно из примера 6.11.

Пример 6.11. Изменение принимаемого по умолчанию каталога с исходными текстами

```
require 'raven'  
  
javac 'compile' do |t|  
  t.build_path << "src/java"  
end
```

Обсуждение

Поскольку система Raven основана на Rake, то для сборки проекта Raven можно применять любое из существующих Rake-заданий. В качестве примера упомянем задание `clean`. Поскольку такое задание уже определено в Rake, включать его еще и в Raven необязательно (см. пример 6.12).

Пример 6.12. Файл сборки Raven с Rake-заданиями

```
require 'raven'  
require 'rake/clean'  
  
CLEAN.include('target')  
  
javac 'compile'
```

Отметим, что для работы Raven наличие JRuby необязательно, поэтому команда `require` для загрузки соответствующего файла отсутствует.

См. также

- Сайт проекта Raven <http://raven.rubyforge.org/>
- Документации по Rake <http://docs.rubyrake.org/>

Что общего между Raven и Maven?

Если коротко, то почти ничего. Raven ни в коей мере не является переносом Maven на язык Ruby. Честно говоря, Raven гораздо ближе к Ant, чем к Maven, особенно если принять во внимание, что Ant, Rake и Raven ведут происхождение от make. В отличие от Maven (и от системы Buildr, которая рассматривается в рецепте 6.10), Raven – это *процедурная* система сборки. В файле Rakefile описана последовательность шагов, необходимых для сборки проекта. Maven же (по крайней мере, частично) представляет собой *декларативную* систему сборки; вы предоставляете метаданные о своем проекте, а Maven сам решает, какие шаги необходимы для его сборки.

Raven не следует принятым в Maven соглашениям об именовании каталогов. По умолчанию предполагается, что исходные Java-файлы находятся в каталоге `src/main/java`, тесты JUnit – в каталоге `src/test/java`, откомпилированные классы Java – в каталоге `target/classes` и т.д.

Raven также умеет импортировать локальный репозиторий Maven и оберывать все находящиеся там JAR-файлы в gem-пакеты. Для этого следует выполнить команду:

```
$ jruby -S raven import
```

6.7. Ссылка на библиотеки в Raven

Задача

Вы хотите собрать с помощью Raven Java-проект, в котором есть зависимости от сторонних библиотек, например Jakarta Commons.

Решение

Воспользуйтесь заданием Raven dependency для определения набора зависимостей, а затем сошлитесь на этот набор из тех заданий, которым эти зависимости нужны. В примере 6.13 показан файл Rakefile для проекта, зависящего от библиотек Jakarta Commons Logging и Jakarta Commons HttpClient, причем для библиотеки HttpClient с помощью оператора `=>` указана минимальная версия, 3.1.

Пример 6.13. Rakefile с зависимостями

```
1 require 'raven'  
2  
3 dependency 'compile_deps' do |t|  
4 t.deps << ['commons-logging', {'commons-httpclient' => '3.1'}]
```

```
5 end
6
7 javac 'compile' => 'compile_deps'
8
9 javadoc 'jdoc' => 'compile_deps'
```

Обсуждение

Если оператор `=>` встречается в определении задания, как в строках 7 и 9 из примера 6.13, то он обозначает наличие зависимости между заданиями.

В Raven для управления зависимостями используется система пакетирования RubyGems, и достигается это путем обертывания JAR-файлов в gem-пакеты. Чтобы не засорять стандартный локальный репозиторий RubyGems, местоположение которого определено переменной среды `GEM_HOME`, Raven сохраняет сгенерированные пакеты в подкаталоге `raven` начального каталога пользователя. Как отмечалось на врезке в рецепте 6.6, в этот каталог можно импортировать все содержимое локального репозитория Maven, выполнив команду:

```
raven import
```

На сайте <http://gems.rubyraven.org/> разработчики Raven организовали общедоступный репозиторий gem-пакетов, в котором хранятся обернутые версии всех библиотек, которые есть в центральном репозитории Maven (<http://repo1.maven.org/maven2/>). В следующем рецепте описано, как организовать свой собственный частный репозиторий.

6.8. Организация частного репозитория Raven

Задача

Вы собираете Java-проект с помощью Raven и хотите, чтобы потенциальная недоступность внешней сети не мешала процессу сборки.

Решение

Создайте частный репозиторий Raven, импортировав в него содержимое из репозитория Maven. Для этого достаточно нескольких простых команд:

```
# Вместо /home/raven ниже укажите удобный для вас каталог.
$ mkdir /home/raven
$ cd /home/raven
$ raven repository
$ raven server
```

В результате импортируются все артефакты из репозитория Maven, а затем на порту 2233 запускается веб-сервер. Для ссылки на этот репозиторий из своего Rakefile добавьте после предложений `require` такую строку:

```
set_sources(["http://localhost:2233"])
```

Обсуждение

У команды `repository` есть несколько интересных возможностей. Во-первых, вы можете ограничить импорт некоторым подмножеством репозитория, передав список идентификаторов проектов. Например, чтобы импортировать только библиотеки Jakarta Commons HttpClient и JUnit, нужно выполнить такую команду:

```
$ raven repository commons-httpclient junit
```

Кроме того, флаг `-m` позволяет импортировать содержимое другого репозитория Maven. Так, следующая команда импортирует репозиторий проекта JBoss:

```
$ raven -m http://repository.jboss.com/maven2/ repository
```

6.9. Прогон тестов JUnit с помощью Raven

Задача

Вы собираете Java-проект с помощью Raven и хотите прогнать автономные тесты JUnit.

Решение

Поместите свои автономные тесты в каталог `src/test/java`, создайте задание `dependency`, в котором описаны необходимые для тестирования зависимости, и воспользуйтесь заданием Raven `junit`, как показано в примере 6.14. По умолчанию Raven ищет классы, имена которых начинаются со строки `Test`, но, как видно из примера 6.14, это соглашение можно переопределить – скажем, включать только классы, *оканчивающиеся* строкой `Test`.

Пример 6.14. Автономное тестирование с помощью Raven

```
require 'raven'
```

```
dependency 'compile_deps' do |t|
  t.deps << ['commons-logging', {'commons-httpclient' => '3.1'}]
end
```

```
dependency 'test_deps' => 'compile_deps' do |t|
  t.deps << {'junit' => '3.8.2'}
end
```

```
javac 'compile' => 'compile_deps'  
  
junit 'test' => ['compile', 'test_deps'] do |t|  
  t.test_classes << "**/*Test.java"  
end
```

Результаты прогона тестов выводятся на консоль. Если все тесты прошли успешно, то вы увидите сообщение ОК:

```
$ rake test  
(in /home/justin/raven-sample1)  
...  
Running test org.jrubycookbook.SomeTest  
.  
Time: 0  
  
OK (1 test)
```

Если какой-то тест не прошел, будет выведена трассировка стека:

```
$ rake test  
(in /home/justin/raven-sample1)  
...  
Running test org.jrubycookbook.SomeTest  
.F  
Time: 0  
There was 1 failure:  
1) testTest(org.jrubycookbook.SomeTest)junit.framework.AssertionFailedError  
at org.jrubycookbook.SomeTest.testTest(SomeTest.java:8)  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.  
java:39)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces  
sorImpl.java:25)  
  
FAILURES!!!  
Tests run: 1, Failures: 1, Errors: 0  
  
There were failures!
```

См. также

- Сайт JUnit <http://www.junit.org>

6.10. Сборка Java-проектов с помощью Buildr

Задача

Требуется собрать Java-проект, но вы хотите написать сценарий сборки на Ruby, а не на XML.

Решение

Воспользуйтесь декларативной системой сборки Buildr, которая предназначена для сборки Java-проектов, но написана на Ruby. Buildr распространяется в виде gem-пакета, для установки следует выполнить команду:

```
$ gem install buildr
```

В системе Buildr для определения проекта используется файл с именем *buildfile**. Минимальный *buildfile* (см. пример 6.15) должен содержать имя проекта (строка 11), группу проекта (строка 13), описание проекта (строка 10), номер текущей версии проекта (строка 12) и способ пакетирования проекта (строка 14).

Пример 6.15. Минимальный *buildfile* в системе Buildr

```
10 desc "The Chapter 6 buildr project"
11 define "ch06-buildr" do
12   project.version = "1.0"
13   project.group = "org.jrubycookbook"
14   package(:jar)
15 end
```

Видя такой *buildfile*, Buildr предполагает, что это проект, содержащий исходные тексты на языке Java в каталоге *src/main/java* и автономные тесты JUnit в каталоге *src/test/java*. Сгенерированный JAR-файл будет называться *ch06-buildr-1.0.jar*. Для сборки проекта (с точки зрения Buildr она сводится к компиляции исходного кода и прогону тестов) достаточно выполнить команду:

```
$ buildr
```

Для построения JAR-файла выполните такую команду:

```
$ buildr package
```

Ту и другую команду можно запускать, находясь как в корневом каталоге проекта, так и в любом его подкаталоге.

Обсуждение

При наличии существующего Java-проекта, особенно если для его сборки используется система Maven, Buildr может автоматически создать такой файл. Находясь в корневом каталоге проекта, выполните команду `buildr` и выберите нужный режим:

```
$ buildr
To use Buildr you need a buildfile. Do you want me to create one?:
1. From maven2 pom file
2. From directory structure
3. Skip
? 1
Created /home/edelsonj/kramer/buildfile
```

* Buildr также ищет файл с именем Buildfile.

Если файл *buildfile* создается из имеющегося файла Maven-проекта *pom.xml*, то Buildr может извлечь всю информацию, показанную в примере 6.15, а также все зависимости проекта. Впрочем, с зависимостями программа импорта может и переборщить, поэтому обязательно проверяйте получившийся *buildfile*. Если Maven-проект включает подмодули, то в сгенерированный *buildfile* войдет информация и о них.

Если для создания *buildfile* выбран режим «From directory structure» (По структуре каталогов), то Buildr сумеет определить только имя проекта (исходя из имени текущего каталога) и способ пакетирования. В примере 6.16 приведен файл *buildfile*, сгенерированный подобным образом.

Пример 6.16. Сгенерированный *buildfile*

```
# Generated by Buildr 1.3.1.1, change to your liking
# Version number for this release
VERSION_NUMBER = "1.0.0"
# Version number for the next release
NEXT_VERSION = "1.0.1"
# Group identifier for your projects
GROUP = "buildr-temp"
COPYRIGHT = ""

# Specify Maven 2.0 remote repositories here, like this:
repositories.remote << "http://www.ibiblio.org/maven2/"

desc "The Buildr-test project"
define "buildr-test" do

  project.version = VERSION_NUMBER
  project.group = GROUP
  manifest["Implementation-Vendor"] = COPYRIGHT
  compile.with # Add classpath dependencies
  package(:jar)
end
```

Сравнение Raven и Buildr

Как отмечалось в рецепте 6.6, между Raven и Buildr существует важное архитектурное различие, такое же, как между Ant и Maven. Raven – это процедурная система сборки, в файле *Rakefile* вы явно определяете, какие задания необходимы для сборки проекта. С другой стороны, Buildr – декларативная система сборки, то есть в файле *buildfile* предоставляется информация о проекте, а Buildr на ее основании решает, как этот проект собирать. Сравнение примера 6.15 с некоторыми файлами сборки Raven из предыдущих рецептов очень выпукло иллюстрирует это различие: в файлах сборки Raven все задания приходится объявлять явно (даже если для них имеются разумные умолчания), тогда как минимального *buildfile* из примера 6.15 вполне достаточно для компиляции, прогона тестов, генерирования документации в формате javadoc и т.д.

Чтобы ознакомиться с полным перечнем заданий Buildr, выполните команду:

```
$ buildr help:tasks
```

См. также

- Сайт Buildr <http://incubator.apache.org/buildr/>

6.11. Ссылка на библиотеки в Buildr

Задача

Вы хотите собрать с помощью Buildr Java-проект, в котором есть зависимости от сторонних библиотек, например Jakarta Commons.

Решение

Передайте список зависимостей методу `compile.with`. Каждая зависимость определяется четырьмя атрибутами: группа, имя, способ пакетирования и номер версии. Например, если ваша программа зависит от библиотек Apache HttpClient и Jakarta Commons Logging, то следует написать так:

```
compile.with "org.apache.httpcomponents:httpclient:jar:4.0-alpha4",  
"org.apache.httpcomponents:httpcore:jar:4.0-beta1",  
"commons-logging:commons-logging:jar:1.1.1"
```

Buildr будет искать зависимости в локальном репозитории Maven (в подкаталоге `.m2/repository` вашего начального каталога). Если там зависимости не обнаружатся, то Buildr попытается загрузить их из удаленного репозитория. Поэтому необходимо также добавить в *buildfile* следующую строку:

```
repositories.remote << "http://repo1.maven.org/maven2/"
```

Обсуждение

Механизм работы с зависимостями в Buildr целиком основан на структуре репозитория Maven. В отличие от системы Raven, в которой используется механизм зависимостей на основе RubyGems, любая библиотека, присутствующая в локальном репозитории Maven, может использоваться Buildr в процесс сборки. Это относится к библиотекам, находящимся как в репозитории *central* (по адресу <http://repo1.maven.org/maven2/>), так и в любом другом общедоступном репозитории, в частности поддерживаемых компаниями Sun (<http://download.java.net/maven/2/>) и JBoss (<http://repository.jboss.com/maven2/>). Индивидуальные разработчики и компании, занимающиеся разработкой ПО, могут организовывать и свои частные репозитории.

Иногда Buildr правильно выполняет транзитивное разрешение зависимостей, а иногда допускает ошибки. Ожидается, что в будущих версиях поддержка в этой области будет улучшена.

См. также

- Введение в репозитории Maven, <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>

6.12. Сборка с помощью Rake в контексте сервера Hudson

Задача

Требуется непрерывно собирать программный проект с помощью системы сборки Rake. Сборка может производиться по расписанию (например, ежедневно в полдень) или после любой поправки на учет в систему управления версиями (например, Subversion).

Решение

Воспользуйтесь сервером непрерывной интеграции с поддержкой Rake, например Hudson. Установив подключаемый модуль Rake, вы сможете просто включить вызов Rake как один из шагов задач (рис. 6.1).



Рис. 6.1. Шаг сборки с помощью Rake в конфигурации задачи сервера Hudson

Обсуждение

Для установки подключаемого модуля Rake в Hudson следует воспользоваться менеджером Hudson Plugin Manager, который находится в меню **Manage**. Открыв Plugin Manager, перейдите на вкладку **Available**, на которой перечислены все доступные подключаемые модули. Отметьте флажок слева от модуля Rake (рис. 6.2)

<input checked="" type="checkbox"/>	Rake plugin	1.0
-------------------------------------	-------------	-----

Рис. 6.2. Подключаемый модуль Rake в менеджере Plugin Manager

и нажмите кнопку **Install**. После установки модуля сервер Hudson необходимо будет перезапустить.

Подключаемый к серверу Hudson модуль Rake позволяет конфигурировать несколько сред исполнения Ruby. Поэтому на одном и том же сервере непрерывной интеграции можно собирать как проекты, написанные как для стандартного интерпретатора Маца, так и для JRuby. Делается это на странице конфигурирования системы System Configuration, показанной на рис. 6.3.

The screenshot shows the 'Rake' configuration section in Hudson. It contains a table of Ruby installations. The first entry has 'name' 'jruby-1.1.2' and 'RUBY_HOME' '/home/justin/jruby-1.1.2'. The second entry has 'name' 'c-ruby' and 'RUBY_HOME' '/usr/lib/ruby'. Each entry has a 'Delete' button. At the bottom of the table is an 'Add' button. Below the table is the text 'List of Rake installations on this system'.

Рис. 6.3. Конфигурирование нескольких сред исполнения Ruby

См. также

- Сайт Hudson <https://hudson.dev.java.net/>
- <http://hudson.gotdns.com/wiki/display/HUDSON/Rake+plugin>, подключаемый модуль Hudson Rake

6.13. Добавление Ruby-сценария в качестве задачи сервера Hudson

Задача

Требуется включить ряд дополнительных шагов в процесс сборки, выполняемый под управлением сервера непрерывной интеграции Hudson.

Решение

Воспользуйтесь подключаемым к серверу Hudson модулем Ruby. Он позволяет задавать в качестве шага задачи произвольный Ruby-сценарий. На рис. 6.4 показана задача, состоящая из двух шагов. Первый выполняет цель Ant с именем war, а второй запускает Ruby-сценарий, который копирует все WAR-файлы во временный каталог.

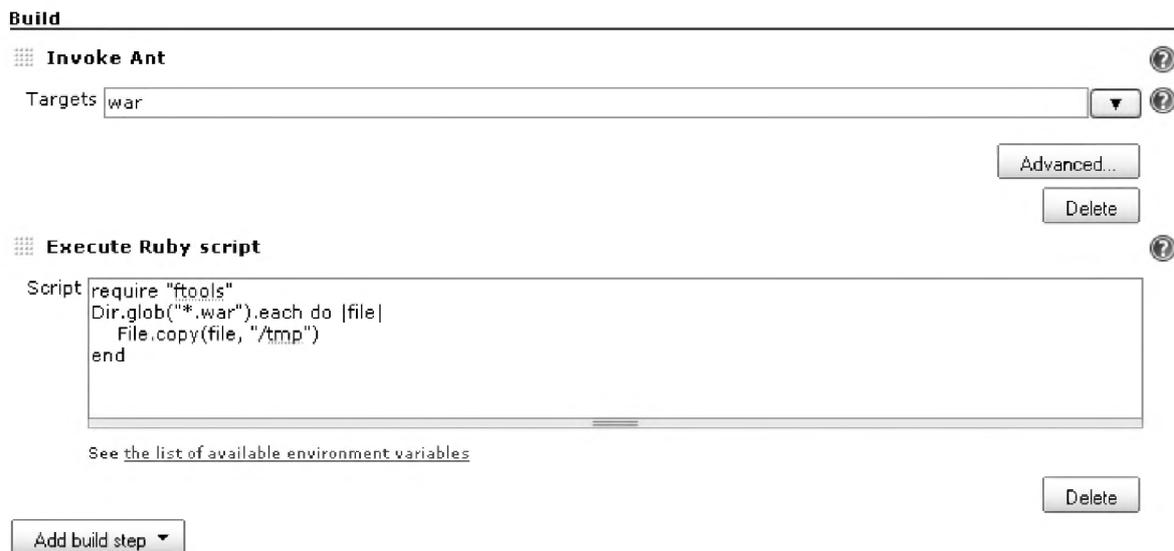


Рис. 6.4. Использование подключаемого к серверу Hudson модуля Ruby

Обсуждение

Как и подключаемый модуль Rake, рассмотренный в рецепте 6.12, модуль Ruby можно скачать и установить с помощью менеджера Hudson Plugin Manager. Но в отличие от Rake, модуль Ruby не поддерживает несколько сред исполнения. Он может выполнять только команду `ruby`, путь к которой указан в переменной среды `PATH`. Поэтому, если вы хотите использовать этот модуль совместно с JRuby, то должны будете скопировать (или создать символическую ссылку) сценарий *jruby*, включенный в дистрибутив JRuby, назвав его *ruby*, и убедиться, что путь к этому сценарию в списке путей, указанных в переменной `PATH`, предшествует пути к любой другой реализации Ruby. Чтобы облегчить отладку, этот подключаемый модуль выводит номер версии на консоль сборки:

```
[workspace] $ ruby -v /tmp/hudson35926.rb
ruby 1.8.5 (2007-09-24 patchlevel 114) [i386-linux]
```

Сервер Hudson создает целый ряд переменных среды, которые становятся доступными Ruby-сценариям, запускаемым описанным образом. Это, в частности, имя задачи (`JOB_NAME`), номер сборки (`BUILD_NUMBER`) и URL сервера Hudson (`HUDSON_URL`). Полный перечень переменных среды можно посмотреть через веб-интерфейс сервера Hudson.

См. также

- <http://hudson.gotdns.com/wiki/display/HUDSON/Rake+plugin>, подключаемый модуль Hudson Rake

Тестирование

7.0. Введение	206
7.1. Автономное тестирование Java-кода с помощью Test/Unit ..	206
7.2. Автономное тестирование Java-кода с помощью библиотеки dust	209
7.3. Автономное тестирование Java-кода с помощью библиотеки Expectations	210
7.4. Тестирование Java-кода с помощью RSpec	212
7.5. Создание mock-объектов с помощью библиотеки Mocha .	217
7.6. Модификация пути поиска классов для JtestR	219
7.7. Группировка тестов для JtestR	219
7.8. Аргументы командной строки при запуске JtestR	220
7.9. Совместное использование JtestR и Ant	222
7.10. Совместное использование JtestR и Maven .	223
7.11. Повышение производительности JtestR	224

7.0. Введение

Эта глава посвящена автоматизированному тестированию, а точнее тестированию Java-кода с помощью Ruby. Подход, основанный на применении динамического языка (например, Ruby) для тестирования кода, написанного на статически типизированном языке (например, Java) обладает рядом достоинств:

- Обычно для прогона автоматизированных тестов требуется выполнять много подготовительного кода. Применение таких *предметно-ориентированных языков* (DSL), как *dust* и *Expectations*, написанных на Ruby, позволяет заметно уменьшить объем этого повторяющегося кода.
- На динамическом языке очень просто создавать *mock*-объекты. Например, JRuby позволяет непосредственно инстанцировать интерфейсы Java.
- Механизм открытых классов позволяет модифицировать код во время выполнения, что тоже упрощает тестирование.

На Ruby реализовано немало каркасов тестирования; наиболее популярные включены в написанный на JRuby проект JtestR. Это проект с открытым исходным кодом, который в 2008 году запустили Ола Бини (Ola Bini) и Анда Абрамовичи (Anda Abramovici), работающие в компании ThoughtWorks. Идея проекта в том, чтобы упростить тестирование Java-кода, применяя самые разные каркасы тестирования, написанные на Ruby. В текущей версии 0.3 JtestR поддерживает следующие библиотеки:

- Test/Unit
- RSpec
- Expectations
- dust
- Mocha

Помимо этого, JtestR поддерживает и каркасы тестирования, написанные на Java: JUnit и TestNG, что делает его универсальным центром тестирования. Скачать JtestR можно с сайта <http://jtestr.codehaus.org>.

7.1. Автономное тестирование Java-кода с помощью Test/Unit

Задача

Вы хотели бы протестировать свой Java-код, избежав при этом такого многословного синтаксиса, как в написанных на Java каркасах JUnit и TestNG. Кроме того, вы знакомы с JUnit и предпочли бы, чтобы кривая обучения новому инструменту была не слишком крута.

Решение

Воспользуйтесь включенной в JtestR поддержкой каркаса Test/Unit, написанного на Ruby. Семантика Test/Unit напоминает JUnit*: каждый тестовый пример расширяет некий класс, а имена тестовых методов строятся по определенному соглашению. В случае каркаса Test/Unit все тестовые примеры должны расширять класс `Test::Unit::TestCase`, а имена тестовых методов начинаться с префикса `test_`. В примере 7.1 показан простой класс, предназначенный для тестирования метода `size()` из класса `java.util.ArrayList`.

Пример 7.1. Простой класс для каркаса Test/Unit

```
class TestArrayList < Test::Unit::TestCase

  def test_that_size_method_works
    list = java.util.ArrayList.new
    assert_equal(0, list.size)
    list << 'first'
    list << 'second'
    assert_equal(2, list.size)
  end
end
```

Обсуждение

Как и JUnit, каркас Test/Unit поддерживает метод подготовки (он называется `setup`), где размещается код, который нужно выполнить перед прогоном каждого теста. Так, если бы мы решили добавить в пример 7.1 второй тестовый метод, то было бы разумно перенести создание экземпляра класса `ArrayList` в метод `setup`, как показано в примере 7.2.

Пример 7.2. Класс Test/Unit с методом `setup`

```
class TestArrayList < Test::Unit::TestCase

  def setup
    @list = java.util.ArrayList.new
  end

  def test_that_size_method_works
    assert_equal(0, @list.size)
    @list << 'first'
    @list << 'second'
  end
end
```

* Точнее, той версии JUnit, которая существовала до появления в Java аннотаций.

```
    assert_equal(2, @list.size)
  end

  def test_that_empty_works
    assert(@list.empty)
    @list << 'first'
    @list << 'second'
    assert(!@list.empty)
  end
end
```

Каркас Test/Unit поддерживает также метод `teardown` для выполнения очистки после прогона каждого теста.

Тесты Test/Unit можно прогонять без какого-либо дополнительного конфигурирования в JtestR. Достаточно поместить файлы с тестовыми классами в каталог *test/unit* и запустить командную утилиту прогона тестов, которая является частью JtestR. Этот класс, а также все зависимости JtestR находится в JAR-файле, который можно скачать с сайта JtestR. Для запуска командной утилиты выполните следующую команду:

```
$ java -cp ~/jtestr-0.3.jar org.jtestr.JtestRRunner
```

Чтобы меньше печатать, можете поместить JAR-файл JtestR в путь поиска классов:

```
$ export CLASSPATH=~/jtestr-0.3.jar:$CLASSPATH
```

Если JtestR запускается с параметрами по умолчанию, то выполняются все тестовые примеры, а результаты выводятся на консоль:

```
$ java org.jtestr.JtestRRunner
Unit TestUnit: 2 tests, 0 failures, 0 errors
```

Если какие-то тесты не проходят, то будут выведены имена тестовых методов, в которых обнаружены ошибки. Так, если бы класс `java.util.ArrayList` работал неправильно, то было бы напечатано примерно такое сообщение:

```
Failure:
test_that_empty_works(TestArrayList)
...
<false> is not true.

Failure:
test_that_size_method_works(TestArrayList)
...
<2> expected but was
<3>.

Unit TestUnit: 2 tests, 2 failures, 0 errors
```

```
Exception in thread "main" java.lang.RuntimeException: Tests failed
```

```
at org.jtestr.JtestRRunner.execute(JtestRRunner.java:117)
at org.jtestr.JtestRRunner.main(JtestRRunner.java:163)
```

См. также

- Документация по <http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/>

7.2. Автономное тестирование Java-кода с помощью библиотеки dust

Задача

Вы хотели бы протестировать свой Java-код, избежав при этом такого многословного синтаксиса, как в написанных на Java каркасах JUnit и TestNG. Каркас Test/Unit, на ваш взгляд, тоже недостаточно лаконичен.

Решение

Воспользуйтесь библиотекой dust Джея Филдса (Jay Fields), поддержка которой включена в JtestR. Эта библиотека предлагает альтернативный синтаксис написания тестов, в котором задействованы средства Ruby по созданию предметно-ориентированных языков. В примере 7.3 тесты из примера 7.2 переписаны с использованием dust.

Пример 7.3. Автономное тестирование с использованием библиотеки dust

```
unit_tests do
  test "that size method works" do
    @list = java.util.ArrayList.new
    assert_equal(0, @list.size)
    @list << 'first'
    @list << 'second'
    assert_equal(2, @list.size)
  end

  test "that empty method works" do
    @list = java.util.ArrayList.new
    assert @list.empty
    @list << 'first'
    @list << 'second'
    assert !@list.empty
  end
end
```

Как и в случае Test/Unit, для прогона dust-тестов в среде JtestR не требуется никакого дополнительного конфигурирования.

Обсуждение

Библиотека `dust` преобразует тело блока, переданного методу `unit_tests`, в Ruby-класс в модуле `Units`. Имя этого класса образуется из имени файла. Если бы пример 7.3 находился в файле `lists_test.rb`, то сгенерированный класс назывался бы `Units::ListsTests`. Любое обращение к тестовому методу преобразуется в вызов метода в сгенерированном классе. Имя этого метода образуется из указанного в описании имени. Так, класс, сгенерированный для примера 7.3, содержит методы `test_that_size_method_works` и `test_that_empty_method_works`.

Помимо метода `unit_tests`, показанного в примере 7.3, `dust` поддерживает также метод `functional_tests`. Разница между ними лишь в том, что тесты, определенные в методе `functional_tests`, помещаются в класс, который находится в модуле `Functionals`. Для конфигурации JtestR, принимаемой по умолчанию, имена сгенерированного класса и его методов видны лишь в случае, когда тест не проходит, например:

```
Failure:
test_that_size_method_works(Units::ListTests)
...
<2> expected but was
<3>.
```



В отличие от Test/Unit, `dust` не поддерживает методов `setup` и `teardown`.

См. также

- Документация по `dust` <http://dust.rubyforge.org>
- Написанное Джем Филдсом введение в `dust` <http://blog.jayfields.com/2007/08/rubygems-dust.html>

7.3. Автономное тестирование Java-кода с помощью библиотеки Expectations

Задача

Вы хотели бы протестировать свой Java-код, избежав при этом такого многословного синтаксиса, как в написанных на Java каркасах JUnit и TestNG. При этом желательно следовать рекомендуемой практике тестирования, в частности, не употреблять более одного утверждения (`assertion`) в каждом тесте.

Решение

Воспользуйтесь включенной в каркас JtestR поддержкой библиотеки Expectations. Как и dust, Expectations предлагает предметно-ориентированный язык для написания тестов. Но в отличие от dust, в Expectations вы не встретите стандартных методов-утверждений, принятых в Test/Unit. Вместо этого каждый тест содержит утверждение относительно возвращаемого значения. В примере 7.4 рассмотренные выше тесты переписаны с использованием Expectations.

Пример 7.4. Автономное тестирование с использованием библиотеки Expectations

```
Expectations do
  expect 0 do
    list = java.util.ArrayList.new
    list.size
  end

  expect 2 do
    list = java.util.ArrayList.new
    list << 'first'
    list << 'second'
    list.size
  end

  expect true do
    list = java.util.ArrayList.new
    list.empty
  end

  expect false do
    list = java.util.ArrayList.new
    list << 'first'
    list << 'second'
    list.empty
  end
end
```

Поддержка Expectations в JtestR по умолчанию отключена, вы должны включить ее явно. Для этого сначала определитесь с соглашением об именовании тестов. Затем создайте файл *jtestr_config.rb* в каталоге *test* своего проекта. В этом файле должна быть строка такого вида:

```
expectation Dir["test/expectations/*.rb"]
```

Здесь мы говорим, что любой файл в каталоге *test/expectations* содержит тесты, прогоняемые с помощью Expectations. Можно было бы принять иное соглашение об именах файлов:

```
expectation Dir["test/**/*_expect.rb"]
```

Или даже объявить отдельные файлы:

```
expectation Dir["test/unit/list_tests_expect.rb"]
```

Обсуждение

Формат вывода в Expectations отличается от принятого в Test/Unit или dust, но содержание информации аналогично:

```
Expectations .F.F  
Finished in 0.00206 seconds
```

```
Failure: 2 failed, 0 errors, 2 fulfilled
```

```
-Failures-
```

```
file </home/justin/list-tests/test/expectations/test.rb>  
line <7>  
expected: <3> got: <2>
```

```
file </home/justin/list-tests/test/expectations/test.rb>  
line <19>  
expected: <true> got: <false>
```

См. также

- Документация по Expectations <http://expectations.rubyforge.org>
- Написанное Джейм Филдсом введение в Expectations, <http://blog.jayfields.com/2007/12/ruby-expectation-gem.html>

7.4. Тестирование Java-кода с помощью RSpec

Задача

Требуется написать для Java-кода поведенческие тесты (behavior-orientated tests).

Решение

Воспользуйтесь включенной в JtestR поддержкой каркаса RSpec, реализующего методологию разработки, основанной на функционировании (Behavior-Driven Development – BDD). На самом деле, RSpec состоит из двух разных каркасов для написания тестов: Spec и Story.

Каркас Spec

Тесты, прогоняемые с помощью каркаса Spec, описывают функционирование объекта посредством последовательности утверждений о его поведении. Эти

утверждения называются *примерами* (examples). Так, Spec-файл в примере 7.5 описывает поведение класса `java.util.HashSet`.

Пример 7.5. Spec-файл для класса `java.util.HashSet`

```
import java.util.HashSet

describe HashSet do
  before(:each) do
    @set = HashSet.new
  end

  it "should be empty" do
    @set.should be_empty
  end

  it "should be of size one after an item is added" do
    @set << "foo"
    @set.size.should == 1
  end

  it "should be of size one after an item is added twice" do
    @set << "foo"
    @set << "foo"
    @set.size.should == 1
  end

  it "should be of size two after two items are added" do
    @set << "foo"
    @set << "bar"
    @set.size.should == 2
  end
end
```

По умолчанию JtestR просматривается все каталоги тестов и выполняет все обнаруженные там файлы с именами, оканчивающимися на `_spec.rb`, считая, что в них находятся спецификации тестов для Spec. Если поместить файл с тестами из примера 7.5 в каталог *unit* и выполнить команду прогона тестов, то на консоль будет выведена такая строчка:

```
$ java org.jtestr.JtestRRunner
Unit Spec: 4 examples, 0 failures, 0 errors
```

Каркас Story

Истории в каркасе RSpec обычно состоят из двух файлов: в одном поведение объекта описывается в виде более-менее обычного текста, который и называется *историей*, а в другом это описание транслируется в последовательность вызовов методов объекта, которые называются *шагами*. Например, в примере 7.6 приведена история, в которой описывается поведение метода `retains()` из класса

`java.util.ArrayList`, а в примере 7.7 – шаги, соответствующие этой истории. Эти файлы ассоциированы с блоком в конце файла шагов.

Пример 7.6. История о классе `java.util.ArrayList`

Story: retain the content of one ArrayList in another

I want to retain only the contents of one ArrayList in another
To create the union of the two lists

Scenario: there is no overlap

```
Given my ArrayList is a new ArrayList
And my other ArrayList is a new ArrayList
And my ArrayList contains "one"
And my ArrayList contains "two"
And my other ArrayList contains "three"
When I retain only the contents of my other ArrayList to my ArrayList
Then my ArrayList should be empty
```

Scenario: there is some overlap

```
Given my ArrayList is a new ArrayList
And my other ArrayList is a new ArrayList
And my ArrayList contains "one"
And my ArrayList contains "two"
And my ArrayList contains "three"
And my other ArrayList contains "one"
And my other ArrayList contains "two"
When I retain only the contents of my other ArrayList to my ArrayList
Then my ArrayList should have a size of 2
And my ArrayList should contain "one"
And my ArrayList should contain "two"
```

Пример 7.7. Шаги для истории о классе `java.util.ArrayList`

```
import java.util.ArrayList
```

```
$lists = { }
```

```
steps_for(:arraylist) do
  Given('my $list_name is a new ArrayList') do |list_name|
    $lists[list_name] = ArrayList.new
  end
  Given('my $list_name contains "$object"') do |list_name, object|
    $lists[list_name] << object
  end
  When('I retain only the contents of my $other_list_name to my $list_name') do
    |other_list_name, list_name|
    $lists[list_name].retain_all($lists[other_list_name])
  end
  Then('my $list_name should have a size of $size') do |list_name, size|
    $lists[list_name].size.should == size.to_i
  end
end
```

```

end
Then('my $list_name should contain "$object"') do |list_name, object|
  $lists[list_name].contains(object).should == true
end
Then('my $list_name should be empty') do |list_name|
  $lists[list_name].should be_empty
end
end

with_steps_for(:arraylist) do
  run 'test/stories/arraylist.story'
end

```

Чтобы прогнать RSpec-истории с помощью каркаса JtestR, достаточно поместить файлы, содержащие историю и шаги, в подкаталог *stories* каталога *test*. Если выполнить команду прогона тестов, то на консоль будет выведена такая строка:

```

$ java org.jtestr.JtestRRunner
Stories: 2 scenarios, 0 failures, 0 errors

```

Обсуждение

Для обоих каркасов Spec и Story можно включить режим вывода подробной информации: либо с помощью флагов в командной строке (как описано в рецепте 7.7), либо создав файл *jtestr_config.rb* в каталоге *test*. В первом случае включите такую строку в конфигурационный файл:

```
output_level :VERBOSE
```

Например, при запуске истории и соответствующей ей спецификации из примера выше на консоль выводится такая информация:

```

$ java org.jtestr.JtestRRunner
should be empty(Java::JavaUtil::HashSet): .
should be of size one after an item is added(Java::JavaUtil::HashSet): .
should be of size one after an item is added twice(Java::JavaUtil::HashSet): .
should be of size two after two items are added(Java::JavaUtil::HashSet): .
Unit Spec: 4 examples, 0 failures, 0 errors
there is no overlap(retain the content of one ArrayList in another): .
there is some overlap(retain the content of one ArrayList in another): .
Stories: 2 scenarios, 0 failures, 0 errors

```

Для каркаса Spec JtestR поддерживает несколько форматов вывода. Наиболее интересен формат HTML, позволяющий создавать красиво оформленные отчеты. Чтобы включить этот режим, добавьте в файл *jtestr_config.rb* такую строку:

```
rspec_formatter ["h", "spec_output.html"]
```

В результате отчет будет выведен в файл *spec_output.html*. На рис. 7.1 показано, как такой отчет выглядит в браузере.

RSpec Results 4 examples, 0 failures
Finished in **0.041 seconds**

Java::JavaUtil::HashSet

- should be empty
- should be of size one after an item is added
- should be of size one after an item is added twice
- should be of size two after two items are added

Рис. 7.1. HTML-отчет RSpec об успешном прогоне тестов

На рис. 7.2 показано, как выглядит отчет, если один из примеров завершается с ошибкой.

RSpec Results 4 examples, 1 failure

Java::JavaUtil::HashSet

- should be empty
- should be of size one after an item is added
- should be of size one after an item is added twice
- should be of size three after two items are added

```

expected: 3,
  got: 2 (using ==)

/home/justin/jttestr-java/test/unit/set_spec.rb:26
file://home/justin/jttestr-0.3.jar!/timeout.rb:48:in `timeout'
file://home/justin/jttestr-0.3.jar!/jttestr/rspec_support.rb:89:in `run_rspec'
file://home/justin/jttestr-0.3.jar!/jttestr/test_runner.rb:255:in `run_group_with'
file://home/justin/jttestr-0.3.jar!/jttestr/test_runner.rb:243:in `run_tests'
file://home/justin/jttestr-0.3.jar!/jttestr/test_runner.rb:242:in `each'
file://home/justin/jttestr-0.3.jar!/jttestr/test_runner.rb:242:in `run_tests'
file://C:/java/jttestr-0.3.jar!/jttestr/test_runner.rb:45:in `run'

```

Рис. 7.2. HTML-отчет RSpec о неудачном прогоне тестов

См. также

- Сайт RSpec <http://rspec.info>
- Введение в BDD <http://dannorth.net/introducing-bdd>
- Рецепт 7.8 «Аргументы командной строки при запуске JtestR»

7.5. Создание мокк-объектов с помощью библиотеки Mocha

Задача

Требуется протестировать Java-класс, зависящий от других классов, при этом тесты должны быть изолированы от изменений в поведении последних.

Решение

Воспользуйтесь включенной в JtestR библиотекой Mocha для создания заглушек и имитаций (мокк-объектов). Mocha позволяет создавать экземпляры интерфейсов и классов Java, демонстрирующие необходимое поведение. Мокк-объекты можно использовать в любых тестах, которые поддерживает каркас JtestR. В примере 7.8 с помощью Mocha создается мокк-объект, имитирующий интерфейс `java.util.Collection`, который затем передается экземпляру класса `java.util.ArrayList`. Этот тест проверяет поведение метода `retainAll()`, точнее то, что он вызывает метод `contains()` переданного объекта `Collection` столько раз, сколько нужно.

Пример 7.8. Автономный тест с использованием библиотек `dust` и `Mocha`

```
unit_tests do
  test "that retainAll only calls contains" do
    list = java.util.ArrayList.new
    list << 'first'
    list << 'second'
    list << 'third'
    other = java.util.Collection.new
    other.expects(:contains).returns(true).times(3)

    list.retainAll(other)
  end
end
```

Если будет вызван еще один метод `other` для объекта `Collection`, то будет возбуждено исключение и тест не пройдет. Например, если в классе `java.util.ArrayList` метод `retainAll()` реализован путем обхода набора, то будет выведено такое сообщение об ошибке:

```
#<Mock:0x4f4>.contains - expected calls: 1, actual calls: 0
```

Это означает, что мокк-объект ожидал, что будет вызван метод `contains()`, но этого не произошло.

Mocha позволяет также задавать ожидаемые параметры. Этой возможностью можно воспользоваться в тесте из примера 7.8, чтобы дополнительно проверить, что `ArrayList` вызывает метод `contains()` в правильном порядке. Модифицированный вариант теста приведен в примере 7.9.

Пример 7.9. Ожидание конкретного значения параметра

```

unit_tests do
  test "that retainAll calls contains once per item in the list" do
    list = java.util.ArrayList.new
    list << 'first'
    list << 'second'
    list << 'third'
    other = java.util.Collection.new
    other.expects(:contains).with('first').returns(true)
    other.expects(:contains).with('second').returns(true)
    other.expects(:contains).with('third').returns(true)
    list.retainAll(other)
  end
end

```

Обсуждение

Моча может создавать моки-объекты как для классов, так и для интерфейсов Java. Для имитации класса передайте этот класс методу `mock`. Единственное ограничение заключается в том, что имитируемый класс не должен быть финальным и не должен содержать финальных методов. Например, не получится следующим образом создать моки-объект для класса `java.lang.String`:

```

s = mock(java.lang.String)
s.expect(:length).return(5)

```

По умолчанию при создании моки-объекта для конкретного класса исходное поведение класса не сохраняется – все методы, которые предполагает вызывать, должны быть определены с помощью метода `expects`. Это можно изменить, передав методу `mock` массив имен методов. Однако из-за возможной путаницы (см. пример 7.10) применяйте эту технику с осторожностью.

Пример 7.10. Имитация конкретного класса с сохранением методов

```

unit_tests do
  test "that using a Java class in JRuby string calls toString" do
    preservedMethods = ['size', JtestR::Mocha::METHODS_TO_LEAVE_ALONE].flatten

    list = mock(java.util.ArrayList, preservedMethods)

    list.expects(:add).times(2).returns(true)

    assert list.size == 0

    list.add «one»
    list.add «two»

    assert list.size == 0 # равен нулю, поскольку метод add имитирован
  end
end

```

См. также

- Сайт Mocha <http://mocha.rubyforge.org/>

7.6. Модификация пути поиска классов для JtestR

Задача

Требуется протестировать классы, отсутствующие в подразумеваемом по умолчанию в JtestR пути поиска классов. По умолчанию этот путь содержит следующие каталоги:

- *build/classes*
- *build/test_classes*
- *target/classes*
- *target/test_classes*

В умалчиваемый путь поиска входят также все JAR-файлы в каталогах *lib* и *build_lib* (и их подкаталогах).

Решение

Определите нужный путь поиска классов в конфигурационном файле JtestR, который по умолчанию называется *jtestr_config.rb* и находится в каталоге *test*. Например, если вы хотите, чтобы поиск классов производился в каталоге *bin*, добавьте в конфигурационный файл строку:

```
classpath 'bin'
```

В конфигурационном файле может быть несколько определений *classpath*.

Обсуждение

Если задать конфигурационный параметр *classpath*, как описано выше, то путь поиска классов, подразумеваемый по умолчанию, отменяется. Если вы хотите не игнорировать, а дополнить умалчиваемый путь, то добавьте в конфигурационный файл еще и такую строку:

```
add_common_classpath true
```

7.7. Группировка тестов для JtestR

Задача

Требуется объединить в группу несколько тестов, прогоняемых с помощью JtestR.

Решение

Для группировки тестов придерживайтесь принятых в JtestR соглашений об именовании каталогов. В пределах главного каталога тестов *test* JtestR автоматически группирует тесты, исходя из каталога, в котором они находятся, и прогоняет тесты в следующем порядке:

1. Автономные тесты в каталоге *unit*.
2. Функциональные тесты в каталоге *functional*.
3. Тесты сопряжений в каталоге *integration*.
4. Прочие тесты, не находящиеся ни в одном из каталогов *unit*, *functional* и *integration*.

Тесты внутри каждой группы, определяемой каталогом, далее разбиваются на подгруппы в зависимости от применяемого каркаса тестирования. Если программа прогона тестов обнаружит несколько групп тестов, то результаты тестирования тоже будут сгруппированы:

```
Unit TestUnit: 4 test, 0 failures, 0 errors
Integration TestUnit: 2 test, 0 failures, 0 errors
```

Здесь мы видим, что в каталоге *unit* было четыре теста, а в каталоге *integration* – два, причем для прогона использовался каркас Test/Unit.

Обсуждение

Хотя JtestR поддерживает автоматическую группировку тестов по каталогам, окружение, в котором выполняются, скажем, автономные тесты, ничем не отличается от окружения функциональных тестов или тестов сопряжения.

7.8. Аргументы командной строки при запуске JtestR

Задача

Требуется тем или иным способом изменить поведение программы прогона тестов, входящей в каркас JtestR, например, ограничить количество тестов или включить дополнительное протоколирование.

Решение

У командной утилиты прогона тестов в JtestR есть довольно много аргументов. К сожалению, они должны задаваться в строго определенном порядке:

port

Этот аргумент, по умолчанию равный 22332, позволяет подключить утилиту прогона тестов к серверному процессу, работающему в фоновом режиме. Это сокращает время, потребное для прогона тестов. Подробнее рассматривается в рецепте 7.11.

tests

Этот аргумент, по умолчанию равный `test`, задает каталог верхнего уровня, в котором следует искать подкаталоги, определяющие группы тестов.

logging

Этот аргумент задает уровень протоколирования. Допустимы следующие значения: `NONE` (протоколирование отключено), `ERR` (сообщения об ошибках), `WARN` (предупреждения), `INFO` (информационные сообщения) и `DEBUG` (отладочные сообщения). По умолчанию подразумевается значение `WARN`.

configFile

Задает имя конфигурационного файла JtestR.

outputLevel

Задает объем выводимой информации о каждом тесте. Допустимы следующие значения: `NONE` (ничего не выводить), `QUIET` (минимальный объем), `NORMAL` (нормальный объем), `VERBOSE` (максимально подробная информация) и `DEFAULT` (по умолчанию).

output

Указывает JtestR, куда выводить информацию. По умолчанию `STDOUT`.

groups

Определяет, какую группу тестов следует прогонять (можно задать несколько групп, разделенных запятыми). По умолчанию прогоняются все обнаруженные тесты.

Часто описанные аргументы применяются для вывода имени каждого прогоняемого теста. В примерах выше вы видели, что по умолчанию JtestR выводит имя теста только, если в нем произошла ошибка. Если задать аргумент `outputLevel` равным `VERBOSE`, то имена тестов будут печататься всегда:

```
$ java org.jtestr.JtestRRunner 22332 test WARN jtsetr_config.rb VERBOSE
test_that_empty_works(TestArrayList): .
test_that_size_method_works(TestArrayList): .
Unit TestUnit: 1 test, 0 failures, 0 errors
```

См. также

- Рецепт 7.7 «Группировка тестов для JtestR»
- Рецепт 7.11 «Повышение производительности JtestR»

7.9. Совместное использование JtestR и Ant

Задача

Вы собираете проект с помощью Apache Ant и хотите добавить тесты, написанные на Ruby.

Решение

Воспользуйтесь заданием Ant, которое входит в состав JtestR. Для этого следует добавить такое определение задания в файл *build.xml*, описывающий процесс сборки для Ant:

```
<taskdef name="jtestr"  
  classname="org.jtestr.ant.JtestRAntRunner" classpath="lib/jtestr-0.3.jar" />
```

Затем вызовите это задание из цели Ant:

```
<target name="test">  
  <jtestr />  
</target>
```

Теперь эту цель можно собрать, выполнив такую команду:

```
$ ant test
```

Обсуждение

Задание JtestR Ant поддерживает все аргументы командной утилиты прогона тестов (см. рецепт 7.8). Например, чтобы включить режим печати подробной информации, можно определить цель следующим образом:

```
<target name="test">  
  <jtestr outputLevel="VERBOSE" />  
</target>
```

Помимо описанных выше аргументов командной строки, есть еще флаг `failOnError`, по умолчанию равный `true`. Установите его в `false`, если хотите, чтобы Ant продолжал сборку даже в случае, когда тесты не проходят.

См. также

- Сайт Ant <http://ant.apache.org/>
- Рецепт 7.6 «Модификация пути поиска классов для JtestR»
- Рецепт 7.11 «Повышение производительности JtestR»

7.10. Совместное использование JtestR и Maven

Задача

Вы собираете проект с помощью Maven и хотите добавить тесты, написанные на Ruby.

Решение

Воспользуйтесь подключаемым к Maven модулем, который входит в состав JtestR. Для этого следует добавить такую ссылку на модуль в файл *pom.xml*:

```
<plugin>
  <groupId>org.jttestr</groupId>
  <artifactId>jttestr</artifactId>
  <version>0.3</version>
  <executions>
    <execution>
      <goals>
        <goal>test</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Теперь JtestR будет автоматически запускаться всякий раз, как Maven выполняет фазу тестирования. К сожалению, в последней версии (0.3) поддержка Maven в JtestR зависела от нестандартной библиотеки JRuby. Поэтому при попытке воспользоваться этим подключаемым модулем выдавалось такое сообщение об ошибке:

```
[ERROR] BUILD ERROR
[INFO] -----
[INFO] Failed to resolve artifact.
```

Missing:

```
-----
1) org.jruby:jruby-complete:jar:r6947
```

Try downloading the file manually from the project website.

```
...
-----
```

1 required artifact is missing.

for artifact:

```
org.jttestr:jttestr:maven-plugin:0.3
```

Чтобы исправить ошибку, скачайте JAR-файл <http://dist.codehaus.org/jtestr/jruby-complete-r6947.jar> и установите его в локальный репозиторий Maven. Для этого нужно выполнить такие команды:

```
$ wget http://dist.codehaus.org/jtestr/jruby-complete-r6947.jar
$ mvn install:install-file -Dfile=jruby-complete-r6947.jar -Dversion=r6947 \
  -DartifactId=jruby-complete -Dpackaging=jar -DgroupId=org.jruby
```

Обсуждение

Поключаемый к Maven модуль JtestR поддерживает все аргументы командной утилиты прогона тестов (см. рецепт 7.8). Например, если вы хотите прогонять только автономные тесты, то сконфигурируйте модуль следующим образом:

```
<plugin>
  <groupId>org.jtestr</groupId>
  <artifactId>jtestr</artifactId>
  <version>0.3</version>
  <executions>
    <execution>
      <goals>
        <goal>test</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <groups>Unit TestUnit</groups>
  </configuration>
</plugin>
```

Помимо описанных выше аргументов командной строки, есть еще флаг `failOnError`, по умолчанию равный `true`. Установите его в `false`, если хотите, чтобы Maven продолжал сборку даже в случае, когда тесты не проходят.

См. также

- Сайт Maven <http://maven.apache.org/>
- Рецепт 7.6 «Модификация пути поиска классов для JtestR»
- Рецепт 7.11 «Повышение производительности JtestR»

7.11. Повышение производительности JtestR

Задача

Требуется ускорить время выполнения автономных тестов при использовании JtestR.

Решение

Запустите сервер JtestR в фоновом режиме. Для этого предназначен класс `org.jtestr.BackgroundServer`:

```
$ java org.jtestr.BackgroundServer
```

При этом создается сервер, прослушивающий порт 22332, который по умолчанию поддерживает две среды исполнения, то есть одновременно можно прогонять два набора тестов. Для изменения этих параметров можно указать в командной строке номер порта, а вслед за ним количество экземпляров среды исполнения. Например, для прослушивания порта 1000 и создания пяти экземпляров среды исполнения нужно выполнить такую команду:

```
$ java org.jtestr.BackgroundServer 1000 5
```

Отметим, что при использовании нестандартного номера порта его нужно будет указывать также при запуске утилиты прогона тестов. В случае командной утилиты это должен быть первый аргумент командной строки:

```
$ java org.jtestr.JtestRRunner 1000
```

Обсуждение

JtestR включает также классы, которые позволяют запустить сервер тестирования из процесса сборки Ant или Maven. Для Ant класс называется `JtestRAntServer`:

```
<target name="server">
  <taskdef name="jtestr-server"
    classname="org.jtestr.ant.JtestRAntServer" classpath="lib/jtestr-0.3.jar" />
  <jtestr-server />
</target>
```

В случае Maven, если вы сконфигурировали подключаемый модуль JtestR в файле *pom.xml*, то можете запустить сервер, введя такую команду:

```
$ mvn jtestr:server
```

См. также

- Рецепт 7.9 «Совместное использование JtestR и Ant»
- Рецепт 7.10 «Совместное использование JtestR и Maven»

Сообщество пользователей JRuby

8.0. Введение	228
8.1. Сборка JRuby из исходных кодов	228
8.2. Отправка извещения о недоработке в JRuby	229
8.3. Списки рассылки JRuby	231

8.0. Введение

В эту, последнюю, главу мы включили ряд рецептов о том, как принять участие в жизни сообщества, сформировавшегося вокруг JRuby. Сначала мы расскажем, как собрать JRuby из исходных кодов, поскольку любому разработчику, желающему узнать, «что у JRuby внутри», рано или поздно придется это делать. Затем мы дадим краткий обзор системы учета недоработок для JRuby и закончим главу описанием способов общения членов сообщества между собой.

8.1. Сборка JRuby из исходных кодов

Задача

Требуется собрать JRuby из исходных файлов, например, чтобы воспользоваться какой-то функцией, не включенной в официальную версию, или создать JAR-файл JRuby для последующего распространения.

Решение

Скачайте исходные файлы из системы управления версиями Subversion:

```
$ svn co http://svn.codehaus.org/jruby/trunk/jruby/
```

JRuby собирается с помощью Apache Ant. В сценарии сборки есть несколько полезных целей:

jar

Создает файл *jruby.jar*.

jar-complete

Создает файл *jruby-complete.jar*, который включает все, что входит в *jruby.jar*, а также все стандартные библиотеки Ruby.

test

Прогоняет комплект автономных тестов JRuby.

dist-bin

Создает двоичный дистрибутив JRuby, то есть ZIP-файл, который можно скачать с сайта <http://dist.codehaus.org/jruby/>.

Обсуждение

Приведенная выше команда Subversion извлекает самую свежую версию исходного кода (основную ветвь) из репозитория JRuby. Но иногда бывает нужно извлечь исходный код, соответствующий той или иной официальной выпускной версии. Для этого следует указать одну из меток, перечисленных на странице <http://svn.codehaus.org/jruby/tags/>. Например, исходный код версии JRuby 1.1 находится по адресу http://svn.codehaus.org/jruby/tags/jruby-1_1/.

В сценарии сборки Ant также есть две цели, относящиеся к совместимости JRuby с другими интерпретаторами Ruby. Хотя формальной спецификации языка Ruby не существует, в ходе работы над проектом Rubinius был создан комплект тестов с очень широкой областью покрытия. В Ant-сценарий сборки JRuby включены следующие связанные с этим цели:

`spec`

Тестировать на соответствие всем выпущенным спецификациям, которым JRuby точно должен удовлетворять.

`spec-all`

Тестировать на соответствие всем выпущенным спецификациям Ruby.

`spec-show-excludes`

Перечислить спецификации, которым JRuby точно не удовлетворяет.

`spec-latest`

Тестировать на соответствие всем выпущенным спецификациям Ruby, которым JRuby точно должен удовлетворять, предварительно получив файл спецификации из системы управления версиями.

`spec-latest-all`

Тестировать на соответствие всем выпущенным спецификациям Ruby, предварительно получив файл спецификации из системы управления версиями.

См. также

- Документация по спецификациям Rubinius <http://rubinius.lighthouseapp.com/projects/5089/the-rubinius-specs>

8.2. Отправка извещения о недоработке в JRuby

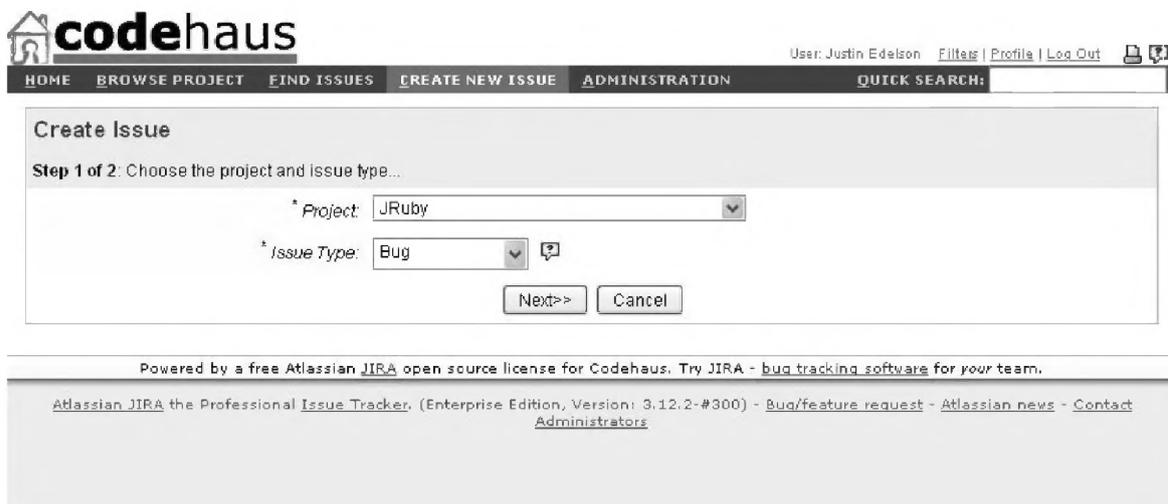
Задача

Вы столкнулись с ошибкой в JRuby или хотите попросить о включении в будущую версию какой-то новой функциональности.

Решение

Для отслеживания извещений о недоработках JRuby используется система Atlassian JIRA. Чтобы посмотреть список зарегистрированных недоработок или создать новое извещение, зайдите на страницу <http://jira.codehaus.org/browse/JRUBY>. Просматривать извещения можно анонимно, но для регистрации нового или добавления комментария к существующему необходимо зарегистрироваться. Прежде чем создавать новое извещение, посмотрите, не было ли чего-то подобного раньше, чтобы не засорять систему дубликатами.

Чтобы создать новое извещение, зарегистрируйтесь, войдите в систему и щелкните по ссылке **Create New Issue** на главной навигационной панели. На рис. 8.1 показано, как выглядит соответствующая страница:



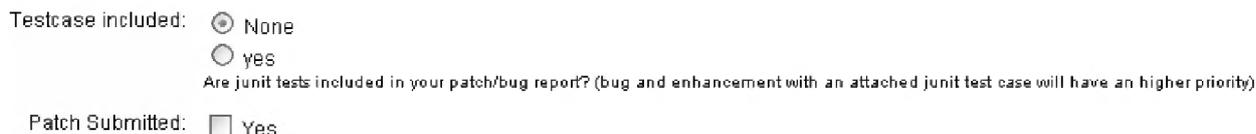
The screenshot shows the 'Create Issue' interface in JIRA. At the top, there's a navigation bar with links like 'HOME', 'BROWSE PROJECT', 'FIND ISSUES', 'CREATE NEW ISSUE', and 'ADMINISTRATION'. The user is identified as 'Justin Edelson'. The main form area is titled 'Create Issue' and indicates 'Step 1 of 2: Choose the project and issue type...'. It features two dropdown menus: '* Project:' set to 'JRuby' and '* Issue Type:' set to 'Bug'. Below these are 'Next>>' and 'Cancel' buttons. A footer section provides information about the Atlassian JIRA license and version (3.12.2-#300).

Рис. 8.1. Страница создания извещения в системе JIRA

Выберите тип извещения и нажмите кнопку **Next** (Далее), после чего введите в форму максимально подробную информацию. Это поможет разработчикам JRuby разобраться в причинах ошибки.

Обсуждение

Внизу формы Issue Details (Подробное описание недоработки) есть два поля (рис. 8.2), которые влияют на очередность рассмотрения вашего извещения.



The screenshot shows two form fields. The first is 'Testcase included:' with radio buttons for 'None' (selected) and 'yes'. Below it is a note: 'Are junit tests included in your patch/bug report? (bug and enhancement with an attached junit test case will have an higher priority)'. The second field is 'Patch Submitted:' with a checkbox for 'Yes'.

Рис. 8.2. Поля Testcase и Patch

Первое поле, «Testcase included» (Включен тестовый пример) говорит, что вы приложили (или приложите) тестовый пример, позволяющий воспроизвести ошибку. Возможность уверенно воспроизвести ошибку чрезвычайно важна для ее скорейшего устранения. Второе поле, «Patch Submitted» (Заплата прилагается) означает, что вы приложили (или приложите) заплату, наложение которой на исходный код JRuby устраняет ошибку. Принято создавать заплаты для самой свежей версии в системе управления версиями, а не для последней официальной выпускной версии.

См. также

- Рецепт 8.3 «Списки рассылки JRuby»

8.3. Списки рассылки JRuby

Задача

Вам необходима помощь по JRuby или относящемуся к нему инструменту.

Решение

Подпишитесь на списки рассылки JRuby. Эти списки поддерживаются системой управления проектами Xircles, которая была разработана специально для сайта Codehaus. Какие существуют списки рассылки по тематике, связанной с JRuby, можно узнать на странице <http://xircles.codehaus.org/projects/jruby/lists>. Там же имеется архив списков рассылки с возможностью поиска.

Обсуждение

Помимо списков рассылки, разработчиков ядра JRuby часто можно застать в IRC-канале #jruby на сайте irc.freenode.net. Все сообщения в этом канале записываются, а по адресу <http://codingbitch.com/irc/channel?channel=%23jruby> можно посмотреть их архив.

Предметный указатель

СИМВОЛЫ

::, оператор, 29
<, оператор наследования, 31
==>, оператор, 195

A

ActionController,
 render_optional_error_file, 83
ActiveMQInitialContextFactory, 114
ActiveRecord, 50
 использование вне Rails, 72
activerecord-jdbc-adapter, gem-пакет, 52, 58
Ant
 включение Ruby-сценариев в процесс сборки, 182
 задание свойства jruby.home, 89
 запуск сервера тестирования из процесса сборки Ant, 225
 использование совместно с JtestR, 222
 написание заданий на Ruby, 187
 сборка Ruby, полезные цели, 228
 условное выполнение, выражение на Ruby, 185
ANT_OPTS, переменная среды, 185
Apache Ant. См. Ant
Apache Maven. См. Maven
Apache Tomcat. См. Tomcat, веб-сервер
ApplicationContext, объект, 126
 запуск с поддержкой таймера, 129
Aptana Studio, создание приложений Rails, 79
ArrayList, объект, 118
asadmin, команда (GlassFish), 70
Atlassian JIRA, система учета недоработок, 229
attr_accessor, функция, 100
Axis WSDL2Java, инструмент, 143

B

BDD (Behavior-Driven Development), 212
BeanNameAware, интерфейс, 131
Bean-компоненты, 123
 встраивание Spring-компонентов, написанных на JRuby, 130
 определение Spring-компонентов на JRuby, 123
 создание самообновляемых Spring-компонентов на JRuby, 127
BSF (Bean Scripting Framework), 86, 183

вызов JRuby из Java-приложения, 91
использование вместо Java Scripting, причины, 95
поддерживаемые сценарные языки, 91
сайт, 92

BSFManager, класс, 92
build_configuration.yaml, 155
Buildr, 198
 вывод списка заданий, 201
 сборка проекта, 199
 создание файла buildfile на основе существующего проекта Maven, 199
 сравнение с Raven, 200
 ссылка на библиотеки, 201
 установка, 199
 файл buildfile, 199

C

case предложение, создание псевдонимов классов, 26
ClassPathXmlApplicationContext класс, 126
codebase, значение для Web Start-приложений, 157
ConcurrentHashMap класс, 98
config.java_libs свойство, 76
configuration элемент, Maven, 189
Connection объект, 118
ConnectionFactory класс, 118
context.xml файл, конфигурирование JNDI в Tomcat, 61

D

DAO, объект доступа к данным (Hibernate), 138
database.yml файл, пример, 72
DLTK (Dynamic Language Toolkit), 39
dust, библиотека
 functional_tests метод, 210
 unit_tests метод, 210
автономное тестирование, 209
использование совместно с Mocha, 217

E

Eclipse IDE
 конфигурирование для разработки на языке JRuby, 39

установка свойства `jruby.home`, 89
 EJB-компоненты (Enterprise JavaBean), 112
 реализация на JRuby, 120
 EntityManager класс, 140
 ErrorListener объект, реализация на Ruby, 103
 evalScriptlet() метод, 87
 EventDao класс, 138
 Executors фабричный класс, 98
 Expectations библиотека
 автономное тестирование (пример), 210
 вывод списка тестов, 212

F

failOnError флаг (JUnit), 222
 file:// схема URL, 157
 FileUtils модуль, развертывание
 приложений, 185
 Functionals, модуль, 210

G

gem сценарий, 23
 GEM_HOME переменная среды, 25
 gem-пакеты
 веб-приложения с внешним
 репозиторием, 56
 указание конкретной версии, 56
 GenericObjectPool класс, 104
 GlassFish v2 сервер, развертывание
 приложений Rails, 69
 GlassFish v3 сервер, 71
 GoldSpike сервлет, 55

H

handleRequest метод, 135
 Hashtable объект, 114
 Hibernate, использование совместно
 с JRuby, 137
 HibernateUtil класс, 138
 HttpServletRequest объект, 135
 HttpServletResponse объект, 135
 Hudson, сервер непрерывной интеграции
 добавление Ruby-сценария в качестве
 задачи, 203
 использование Rake, 202
 переменные среды, 204

I

impl метод, 35
 include предложение, сопоставление
 интерфейса Java с классом Ruby, 33
 include_class функция, 26
 include_package функция, 26
 initial свойство, 114
 InitialContext объект, 114
 irc.freenode.net, канал #jruby, 231

J

Jakarta Commons Logging (JCL), 95
 Jakarta Commons Pool, 104
 Jakarta Commons библиотеки, ссылка
 из Raven, 195
 JAR-файлы
 включение в путь поиска классов, 30
 для апплетов, написанных на JRuby, 158
 исполняемые, генерация с помощью Rawr, 154
 подписание, для Web Start-приложений, 156
 Java
 запуск JRuby как Java-приложения, 41
 обращение к классам из Ruby, 25
 открытие классов в JRuby, 37
 преобразование Ruby-массива
 в Java-массив, 29
 преобразование типов Java в типы Ruby, 27
 расширение класса в Ruby, 31
 реализация Java-интерфейса на Ruby, 32
 Java EE
 См. также Java в масштабе предприятия
 доступ к статическим файлам
 из приложения Rails, 55
 и Rails, 50
 пакетирование приложения Rails, 54
 Java Management Extensions (JMX), 29, 104
 Java Messaging Service. *См.* JMS
 Java Naming and Directory Interface.
 См. JNDI
 Java Persistence API (JPA), 140
 Java Scripting (JSR 223), 86, 183
 Java Server Pages (JSP), простой шаблон, 136
 Java в системах масштаба предприятия, 112
 встраивание Spring-компонентов,
 написанных на JRuby, 130
 выполнение вызовов по протоколу
 SOAP, 141
 использование Hibernate совместно
 с JRuby, 137
 использование Java Persistence API
 совместно с JRuby, 140
 определение Spring-компонентов
 на JRuby, 123
 отправка JMS-сообщений, 115
 получение JMS-сообщений, 118
 реализация Aware-интерфейсов Spring
 в JRuby-объектах, 131
 реализация компонента Enterprise JavaBean
 на JRuby, 120
 создание MVC-контроллеров Spring
 с помощью JRuby, 134
 создание контекста JNDI, 113
 создание самообновляемых
 Spring-компонентов на JRuby, 127
 упрощение доступа к LDAP-каталогу, 143
 java.naming.factory.initial, свойство, 114
 java.util.concurrent, пакет, 97

- JAVA_HOME, переменная среды, 18
- JAVA_OPTS, переменная среды, 62
- JavaEmbedUtils класс, 87
 - initialize() метод, 88
- JavaScript, развертывание, 159
- javaws инструмент, 157
- JBoss, сервер приложений
 - изменение контекстного пути для развертывания, 78
 - развертывание Rails, 62
- JCL (Jakarta Commons Logging), 95
- JDBC (Java Database Connectivity), 52
 - доступ к базе данных с помощью пакета activerecord-jdbc-adapter, 72
 - конфигурирование приложения Rails для доступа к источнику данных JDBC через JNDI, 58
- Jetty, контейнер сервлетов
 - контекстного пути для развертывания, 79
 - развертывание Rails, 64
- jetty_rails, gem-пакет, 65
- jetty_rails.yml, конфигурационный файл, 66
- jetty-plus.xml, файл, 65
- JIRA, система учета недоработок, 229
- jirb
 - интерактивный сеанс работы с базой данных, 73
 - командная и графическая версии, 18
- JMS (Java Messaging Service), 112, 115
 - отправка JMS-сообщения из Ruby, 116
 - получение JMS-сообщений, 118
 - типы сообщений, 117
- JMX (Java Management Extensions), 29, 106
 - jmx4r, gem-пакет, 106
- JNA (Java Native Access) API, 108
- JNDI (Java Naming and Directory Interface), 112
 - использование Tomcat с источником данных JNDI, 61
 - использование с Jetty, 64
 - недостатки, 144
 - пакетирование приложения Rails с источником данных JNDI, 58
 - поддержка LDAP JNDI, 145
 - создание контекста, 113
- JPA (Java Persistence API), 140
- JRuby, 15
 - включение JAR-файлов в путь поиска классов, 30
 - обращение к Java-классам из Ruby, 25
 - открытие Java-класса, 37
 - преобразование типов Java в типы Ruby, 27
 - распознавание платформы, 47
 - расширение класса в Ruby, 31
 - реализация Java-интерфейса на Ruby, 32
 - сайт проекта, 14
 - управление пакетами с помощью RubyGems, 21
 - установка в Linux и Mac OS X, 19
 - установка в Windows, 17
- JRuby on Rails, 50
 - доступ к статическим файлам из приложения Rails, работающего в контейнере Java EE, 82
 - изменение имени WAR-файла и местоположения рабочей области, 77
 - конфигурирование Rails для соединения с базой данных, 52
 - конфигурирование сервлета JRuby-Rack, 57
 - конфигурирование хранилища сеансов, 75
 - пакетирование приложения Rails с источником данных JNDI, 58
 - получение информации о Java-сервлете из контроллера Rails, 74
 - развертывание приложения Rails в корневом контексте, 77
 - развертывание приложения Rails на сервере GlassFish v2, 69
 - развертывание приложения Rails на сервере JBoss, 62
 - развертывание приложения Rails на сервере Jetty, 64
 - развертывание приложения Rails на сервере Mongrel, 68
 - развертывание приложения Rails с помощью jetty_rails, 65
 - сервер приложений GlassFish v3, 71
 - создание приложения Rails в Aptana Studio, 79
 - управление классами, каталогами и прочими файлами, упакованными в WAR-файл, 75
 - установка и настройка, 51
- jrubby, исполняемый файл, флаг -S, 23
- jrubby, элемент, 123
- jrubby.home, системное свойство
 - задание с помощью переменной среды ANT_OPTS, 185
 - MAVEN_OPTS, 189
- установка, 89
- jrubby.runtime.threadlocal, системное свойство, 88
- JRUBY_HOME переменная среды
 - установка в Linux и Mac OS X, 20
 - установка в Windows, 19
- JRUBY_VERSION переменная среды, 47
- JRubyEngine класс, 92
- JRubyLogFactory класс-мост, 96
- jrubby-openssl, gem-пакет, 51
- JRuby-Rack, 50
 - и Warbler, 55
 - конфигурирование сервлета, 57
- JSR (Java Specification Request) 223, 86, 93
- JtestR, 206
 - dust, имена сгенерированного класса и методов, 210
 - аргументы командной строки, 220
 - группировка тестов, 219

использование совместно с Ant, 222
 использование совместно с Maven, 223
 модификация пути поиска классов, 219
 повышение производительности, 224
 поддержка Expectations, 211
 прогон RSpec-историй, 215
 прогон тестов Test/Unit, 208
 форматы вывода для каркаса Spec, 215
 jtestr_config.rb файл, 219
 JtestRAntServer класс, 225
 JUnit тесты, 197. *См. также* Тестирование
 прогон с помощью Raven

L

lang:jruby, элемент, атрибут
 refresh-check-delay, 127
 lang:property элемент, 123
 LdapCtx класс, 144
 LDAP-сервер, 113
 упрощение доступа, 143
 libraryPaths, элемент конфигурационного
 файла, 189
 Linux, установка JRuby, 19

M

Mac OS X
 пакетирование приложения JRuby, 154
 установка JRuby, 20
 Map интерфейс, 114
 Maven
 включение Ruby-сценариев в процесс
 сборки, 188
 генерация файла сборки Buildr
 по проекту Maven, 199
 запуск сервера тестирования JtestR
 из процесса сборки Maven, 225
 импорт репозитория, 197
 путь к JAR-файлу в локальном
 репозитории, 31
 сравнение с Raven, 195
 структура репозитория, поиск
 зависимостей в Buildr, 201
 MAVEN_OPTS, переменная среды, 189, 193
 Mbean-объекты, 107
 MessageConsumer класс, 118
 Mocha, библиотека для создания заглушек
 и имитаций, 217
 автономное тестирование с помощью dust
 и Mocha, 217
 имитация конкретного класса
 с сохранением методов, 218
 ModelAndView класс, 135
 Mojo класс, 191
 Mongrel веб-сервер
 параметры port и environment, 66
 поддержка кластеризации, 68

развертывание Rails, 68
 mongrel_jcluster gem-пакет, 68
 Mule клиентский модуль, 142
 MVC (модель-вид-контроллер),
 архитектура, 50
 создание MVC-контроллеров Spring
 с помощью JRuby, 134
 MySQL СУБД
 соединение с помощью пакета activerecord-
 jdbcmysql-adapter, 72
 установка адаптера для, 53
 mysql-ds.xml, конфигурационный файл
 JBoss, 63

N

NetBeans
 конфигурирование для разработки
 на JRuby, 43
 редактор Ruby Code Editor, 44
 new, команда, 127

O

openssl, gem-пакет, 51

P

PATH, переменная среды
 в Linux и Mac OS X, 20
 в Windows, 19
 PermGen, область памяти, 60, 87
 Persistence класс, метод
 createEntityManagerFactory(), 140
 pom.xml, файл (Maven), 188
 для подключаемого к Maven модуля
 на JRuby, 190
 prefix, свойство, 124
 Proc-объект, преобразование в блок Ruby, 36
 public_dir.rb файл, 82

R

RadRails, Aptana Studio, 79
 Rake
 и Raven, 193
 и Warbler, 55
 сборка в контексте сервера Hudson, 202
 Raven, 193
 изменение подразумеваемого
 по умолчанию каталога с исходными
 файлами, 194
 пример сценария сборки, 194
 прогон тестов JUnit, 197
 сборка с помощью задания Rake, 193
 сравнение с Buildr, 200
 сравнение с Maven, 195

ссылка на библиотеки, 195
 частный репозиторий, 196
 Rawr, gem-пакет, 153
 генерация Web Start-приложения, задание
 rawr:bundle:web, 157
 задание rawr:bundle:app, 154
 задание rawr:bundle:exe, 154
 получение полного списка заданий, 155
 пример конфигурационного файла, 154
 RDT (Ruby Development Tools), 39
 RedCloth, библиотека, 190
 refresh-check-delay атрибут, 127
 reject! метод, 76
 require_from_maven метод, 31
 respond_to? метод, 100
 RSpec
 вывод подробной информации, 215
 каркас Spec, 212
 каркас Story, 213
 Rubinius проект, 14, 229
 Ruby, 14
 блоки, 14
 выполнение из Java, 87
 официальный сайт, 15
 Ruby on Rails, 50
 использование внешнего репозитория
 gem-пакетов, 56
 пакетирование для работы
 в среде Java EE, 54
 развертывание в контейнере Java EE
 на сервере Tomcat, 60
 Ruby класс, 89
 RUBY_PLATFORM, переменная среды, 47
 RubyGems
 команды, 21
 совместное использование gem-пакетов, 24
 управление пакетами, 21
 RubyListener класс, 126
 модификация для изменения порядка
 сообщений на противоположный, 129

S

script, задание (Ant), 183
 атрибут src, 185
 script, элемент конфигурационного файла
 (Maven), 189
 scriptcondition, элемент (Ant), 185
 scriptdef, задание (Ant), 187
 script-interfaces атрибут, 125
 script-source атрибут, 125
 SendDateTask класс, 128
 -server VM, флаг, 64
 ServletContext объект, 74
 ServletRequest объект, 74
 ServletResponse объект, 74
 Session объект, 118
 SLSB (компонент-сеанс без сохранения
 состояния), 122

SOAP, вызов удаленного метода
 с помощью клиента Mule, 142
 Spec каркас, 212
 форматы вывода, 215
 Spring каркас, 113, 123
 встраивание компонентов, написанных
 на JRuby, 130
 реализация Aware-интерфейсов
 в JRuby-объектах, 131
 создание MVC-контроллеров на JRuby, 134
 создание самообновляемых
 компонентов, 127
 Story каркас, 213
 Subversion, извлечение исходного кода
 JRuby, 228
 Swing, 148
 долго работающие задачи, 151
 обработка событий, 150
 разработка ГИП на JRuby, 148
 SwingWorker класс, 152
 System.getenv() метод, 90

T

Test/Unit, 207
 класс с методом setup, 207
 метод teardown, 208
 прогон тестов в среде JtestR, 208
 тестирование метода size() из класса
 ArrayList, 207
 Textile язык разметки, 190
 TimerTask класс, 127
 to_java метод, 29
 Tomcat сервер приложений
 запуск с флагом -server, 60
 изменение контекстного пути
 для развертывания, 78
 конфигурирование JNDI, 61
 развертывание Rails, 59
 TRaX (Transformation API for XML), 102

U

Units модуль, 210
 URIResolver интерфейс, реализация на Ruby, 103

W

Warbler, 51
 WAR-файлы, 54
 gem.path, контекстный параметр, 56
 jruby.session_store, контекстный
 параметр, 75
 war:jar задание, 59
 war:webxml задание, 59
 изменение имени, 77
 пакетирование приложения Rails, 54

управление классами, каталогами
и прочими упакованными файлами, 75
Web Start технология, пакетирование
приложений, 155
web.xml файл
 пример приложения Rails с источником
 данных JNDI, 59
WEBrick веб-сервер
 параметры port и environment, 66
Windows системы
 пакетирование приложений JRuby, 154
 установка JRuby, 17
WSDL2Java инструмент, 143

Х

XML, преобразование с помощью
 библиотеки TrAX, 102
-XX:PermSize, флаг командной строки, 87

А

Автономное тестирование, 206.
 См. также Тестирование
 с помощью Raven, 197
Автономные приложения, пакетирование, 153
Адаптеры баз данных, 53
Аннотации, подключаемый модуль Maven, 192
Аплеты
 написание на JRuby, 157
 развертывание с помощью JavaScript, 159

Б

Безопасность относительно потоков
 до выхода Rails 2.2, 57
Библиотеки
 общедоступный репозиторий gem-пакетов
 для Raven, 196
 платформенно-зависимые, доступ
 из JRuby, 108
 поддержка Maven в JtestR, 223
 ссылка из Buildr, 201
 ссылка из Raven, 195
Блоки
 определение транзакций, 139
 реализация Java-интерфейса, 34

В

Виртуальная машина Java, 16
Внедрение зависимости, 113
Внешний облик приложения
 изменение, 149
 сторонние библиотеки, 150
Встраивание сценария в задание script, 185
Встроенное определение контроллера
 на JRuby, 137

Г

Глобальные переменные
 представление Java-объектов в среде
 исполнения Ruby, 92
 создание с помощью JSR, 94

Д

Дескрипторный файл развертывания (Jboss),
 пример, 63
Динамическая типизация, 33

З

Зависимости
 JRuby, определение в файле сборки Ant, 183
 импорт проекта Maven в Buildr, 199
 определение с помощью задания Raven
 dependency, 195
 ссылка на библиотеки в Buildr, 201

И

Инверсия управления, 113
Инструменты сборки, 182
 Ant, включение Ruby-сценариев в процесс
 сборки, 182
 Maven, включение Ruby-сценариев
 в процесс сборки, 188
 добавление Ruby-сценария в качестве
 задачи сервера Hudson, 203
 написание заданий Ant на Ruby, 187
 написание подключаемого к Maven модуля
 на JRuby, 190
 организация частного репозитория
 Raven, 196
 прогон тестов JUnit с помощью Raven, 197
 сборка Java-проектов с помощью Buildr, 198
 сборка Java-проектов с помощью Raven, 193
 сборка с помощью Rake в контексте сервера
 Hudson, 202
 ссылка на библиотеки в Raven, 195
 условные конструкции Ant, применение
 Ruby, 185
Интеграция с Java, 86
 API для внедрения Ruby-кода
 в Java-приложение, 86
 вызов JRuby с помощью каркаса Bean
 Scripting Framework, 91
 вызов JRuby с помощью технологии Java
 Scripting, 93
 выполнение Ruby-кода
 из Java-приложения, 87
 доступ к платформенно-зависимым
 библиотекам из JRuby, 108
 использование встроенных в Java средств
 параллельной обработки, 97

написание единообразного кода, 101
 преобразование XML-документов
 с помощью библиотеки TrAX, 102
 создание методов-аксессоров в духе
 JavaBean, 100
 создание пула сред исполнения JRuby, 104
 удаленное управление с помощью
 технологии JMX, 106

Интерпретаторы, 15
 выбор подходящего для исполнения
 сценариев на Ruby и JRuby, 23
 конфигурирование DLTK, 40
 конфигурирование RDT, 39
 языка Ruby на C, 14

Интерфейсы
 Java, реализация на JRuby, 32
 Spring, уведомление компонента
 о его окружении, 131

Исполняемый JAR-файл, генерация
 с помощью Rawr, 154

Использование памяти
 JRuby и виртуальная машина Java, 60
 поколение постоянных объектов
 (PermGen), 87

Источник данных (JNDI)
 использование в Tomcat, 61
 пакетирование Rails с, 58
 развертывание в JBoss, 63

К

Кластеризация, поддержка в сервере Mongrel, 68
 Контекст (JNDI), создание, 113
 Корневой контекст контейнера сервлетов,
 развертывание приложения Rails в, 77

М

Массивы, преобразование Ruby-массива
 в Java-массив, 29
 Мацумото Юкиhiro, 14

Методы
 вызов метода Java-класса из Ruby,
 преобразование типов, 27
 вызов с помощью Java Management
 Extensions (JMX), 29
 доступ к открытым статическим методам, 29

Методы-аксессоры (в духе спецификации
 JavaBean), создание, 100

Модули
 реализация Aware-интерфейсов Spring, 132
 реализация Java-интерфейсов, 35

Н

Наборы, создание прокси-объектов Ruby
 для классов Java-наборов, 99

Наследования оператор (<), 31
 Непрерывной интеграции среда, 183

О

Объектно-реляционное отображение (ORM).
 См. ActiveRecord
 Открытые классы, 144

П

Пакетирование приложений JRuby
 для Windows, Mac, Linux и Java, 154
 для запуска по технологии Web Start, 155

Пакеты
 Java, 25
 обертывание Java-пакета в Ruby-модуль, 26

Переменные среды
 Hudson, 204
 в Windows, 18

Платформенно-зависимые библиотеки
 доступ из JRuby, 108

Псевдонимы классов для устранения
 конфликтов имен, 26

Пул сред исполнения JRuby
 сервер GlassFish v3, 71
 создание с помощью библиотеки Jakarta
 Commons Pool, 104

Путь загрузки, 134

Путь поиска классов
 включение JAR-файлов, 30
 динамическое добавление файлов, 114
 добавление Mule и зависимостей, 142
 задание для использования совместно
 с Spring, 125
 модификация для JtestR, 219

Р

Регулярные выражения в методе reject!, 76

С

Самоподписанный сертификат, создание, 156

Сервлеты, обращение к EJB-компоненту
 на JRuby, 121

Сертификаты, создание самоподписанных
 сертификатов, 156

События
 обработка с помощью блоков, 150
 обработка с помощью экземпляра
 интерфейса Java, 151

Соединение с базой данных,
 конфигурирование в Rails, 52

Сообщество JRuby, 228
 IRC-канал #jruby, 231
 отправка извещения о недоработке, 229

сборка JRuby из исходных кодов, 228
списки рассылки, 231
Списки рассылки
сообщество JRuby, 231
Среда исполнения
задание количества экземпляров
для JRuby-Rack, 57
задание максимального количества
экземпляров приложения Rails, 57
конфигурирование в файле
jetty_rails.xml, 66
организация пула в сервере GlassFish v3, 71
Средства параллельной обработки (Java), 97
Статические методы и переменные, доступ
из JRuby, 29
Статические файлы
доступ из приложения Rails, работающего
в контейнере Java EE, 82

Т

Тестирование, 206
автономное с помощью dust, 209
автономное с помощью Expectations, 210
автономное с помощью Test/Unit, 206
аргументы командной строки при запуске
JtestR, 220
группировка тестов для JtestR, 219
использование RSpec, 212
модификация пути поиска классов для
JtestR, 219
повышение производительности JtestR, 224
совместное использование JtestR и Ant, 222

совместное использование JtestR и Maven, 223
создание mock-объектов, 217
Типы данных
генерация классов Java по описанию
веб-службы, 143
динамическая типизация, 33
подключаемый к Maven модуль,
написанный на Ruby, 192
преобразование Ruby-массива
в Java-массив, 29
преобразование типов Java в типы Ruby, 27
Транзакции, определение с помощью
блоков, 139

У

Удаленные приложения Java, управление
с помощью JMX, 106
Условные конструкции, Ant, 185
scriptcondition, 185
комбинирование scriptcondition с другими
условиями, 186

Х

Хранилище ключей, файл, 156
Хранилище сеансов, конфигурирование
для приложения Rails, 75

Ц

Цели Ant, вызов из Ruby, 183

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Джастин Эдельсон, Генри Лю

JRuby **Сборник рецептов**

Главный редактор	<i>Мовчан Д. А.</i>
	dm@dmk-press.ru
Перевод с английского	<i>Слинкин А. А.</i>
Корректор	<i>Синява Г. И.</i>
Верстка	<i>Чаннова А. А.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Подписано в печать 31.08.2009. Формат 70×100^{1/16}.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 36. Тираж 1000 экз.

№

Web-сайт издательства: www.dmk-press.ru