CAPÍTULO 19

Apêndice: Integrando Java e Ruby

"Não há poder. Há um abuso do poder nada mais"
— Montherlant, Henri

Nesse capítulo você aprenderá a acessar código escrito anteriormente em Java através de scripts escritos em Ruby: o projeto JRuby.

19.1 - O PROJETO

JRuby (http://www.jruby.org/) é uma implementação de um interpretador Ruby escrito totalmente em java, e mais ainda, com total integração com a Virtual Machine.

Além de ser open-source, ele disponibiliza a integração entre as bibliotecas das duas linguagens.

Atualmente há algumas limitações como, por exemplo, não é possível herdar de uma classe abstrata. O suporte a Ruby on Rails também não está completo.

Os líderes desse projeto open source já trabalharam na Sun, o que permitiu termos uma implementação muito rápida e de boa qualidade.

19.2 - TESTANDO O JRUBY

Vamos criar um script que imprima um simples "Testando o JRuby" na tela do seu console:

```
print "Testando o JRuby!\n"
```

Pode parecer muito simples, mas a grande diferença é que agora quem estará realmente rodando é uma Virtual Machine Java! Não há mais a necessidade de

instalar o ruby na máquina, apenas a JVM e algumas bibliotecas do JRuby! Isto pode ajudar muito na adoção da linguagem.

Agora se executarmos tanto com os comandos "ruby" ou "jruby" o resultado será o mesmo:

19.3 - Exercícios

1. Crie um arquivo chamado testando.rb que imprime na tela "Testando o JRuby!":

   a. Edite o arquivo: testando.rb.

   b. Adicione o seguinte conteúdo:

```
print "Testando o JRuby!\n"
```

   c. Rode o arquivo com o JRuby:

```
jruby testando.rb
```

19.4 - Compilando ruby para .class com JRuby

Existe a possibilidade de compilarmos o arquivo .rb para um .class através do JRuby. Para isso devemos utilizar o jrubyc(JRuby Compiler) de modo muito semelhante ao javac:

```
jrubyc <path do arquivo .rb>
```

Vamos criar um arquivo ola_mundo_jruby.rb:

```
# ola_mundo_jruby.rb
puts 'Ola Mundo com JRuby!'
```

Agora vamos compilar esse arquivo:

```
jrubyc ola_mundo_jruby.rb
```

Após isso, o arquivo ola_mundo_jruby.class já foi criado na mesma pasta do arquivo ola_mundo_jruby.rb e nós podemos utilizá-lo a partir de outro arquivo .rb através do require, porém esse .class é diferente do que o javac cria a partir do .java, sendo assim é impossível rodá-lo direto na JVM como rodamos outra classe qualquer do java.

19.5 - Rodando o .class do ruby na JVM

Como foi dito anteriormente, não é possível executar diretamente na JVM um arquivo compilado pelo jruby, isso acontece pelas características dinâmicas do ruby que tornam necessário a utilização de um jar. Tal jar pode ser baixada no site do Jruby(http://jruby.org/download).

Com o .jar em mãos, é fácil executar um bytecode do jruby na JVM, simplesmente devemos utilizar a opção "-jar" da seguinte maneira:

```
java -jar <path do arquivo .jar> <path do arquivo .class>
```

Lembrando que é necessário que a extensão do arquivo(.class) esteja explícita.

Vamos copiar o arquivo .jar do jruby para a pasta onde o ola_mundo_jruby.class está e rodar o nosso olá mundo:

```
java -jar jruby.jar ola_mundo_jruby.class
```

Após isso veremos nosso "Ola Mundo com JRuby!".

19.6 – Importando um bytecode(.class) criado pelo JRuby

Para importar um bytecode que foi criado a partir de um arquivo .rb utilizamos o conhecido require.

```
require '<path do bytecode>'
```

Obs.: Lembre-se de retirar a extensão do arquivo (.class), o certo seria fazer algo como:

```
require 'app/funcionario'
```

e não:

```
# desta maneira o arquivo não será encontrado
require 'app/funcionario.class'
```

19.7 – Importando classes do Java para sua aplicação JRuby

Para importar classes Java utilizamos o método java_import, porém devemos ter o cuidado de antes requerer a biblioteca que tem esse método.

Vamos criar uma classe Pessoa em Java, e importar ela para dentro do JRuby. Primeiramente criaremos a classe Pessoa no java:

```
// Pessoa.java
        public class Pessoa {
            private String nome;

            public Pessoa(String meuNome) {
                this.nome = meuNome;
            }

            public void setNome(String novoNome) {
                this.nome = novoNome;
            }

            public String getNome(){
                return this.nome;
            }

            public void seMostra(){
                System.out.println(this.getNome());
            }
        }
```

Agora vamos compilar o código fonte com o javac utilizando:

```
javac Pessoa.java
```

Teremos então o arquivo Pessoa.class. Vamos criar um arquivo testando_jruby.rb onde vamos testar essa classe:

```
# testando_jruby.rb
        require 'java' # o java_import faz parte desta biblioteca
        java_import 'Pessoa'

        pessoa = Pessoa.new 'João'
        pessoa.se_mostra
    # Observe que o nome do método no código Java é
        # seMostra, porém o JRuby faz um alias para
        # todos os métodos passando-os de Camelcase para
        # Underscore case.
        # Obs.: o método seMostra ainda existe.

        pessoa.nome = 'Carlos'
    # Observe que ao criarmos um setter
        # para o nome(setNome), o JRuby criou
        # o método nome= automaticamente.
    # Obs.: Os métodos setNome e set_nome
        # continuam existindo.

        puts pessoa.nome
    # Observe que ao criarmos um getter
        # para o nome(getNome), o JRuby criou
        # o método nome automaticamente
        # Obs.: Os métodos getNome e get_nome
        # continuam existindo.
```

Ao executarmos o exemplo acima, teremos como saída:

```
João
Carlos
```

Lembrando que para executar este exemplo basta utilizar

```
jruby testando_jruby.rb
```

19.8 – Testando o JRuby com Swing

Agora vamos integrar nosso "Testando o JRuby" com um pouco de Java, criando uma janela. Instanciamos um objeto Java em JRuby usando a notação:

```
require 'java'

module Swing
  include_package 'java.awt'
```

```
  include_package 'javax.swing'
end

module AwtEvent
  include_package 'java.awt.event'
end

# Reparem que não é necessário herdar nem
# implementar nada, apenas definir o metodo
# com o nome que o java exige (duck typing)
class ListenerDoBotao
  def action_performed(evento)
    Swing::JOptionPane.showMessageDialog(
    nil, "ActionListener feito em ruby")
  end
end

frame = Swing::JFrame.new
painel = Swing::JPanel.new
frame.add painel

label = Swing::JLabel.new
# label.setText("Testando o JRuby!")
label.text = "Testando o JRuby!"
painel.add label

botao = Swing::JButton.new 'clique aqui'
botao.add_action_listener ListenerDoBotao.new
painel.add botao

frame.pack
frame.set_size(400, 400)
frame.visible = true
```

O include_package é parecido com um import, e depois estamos criando uma instância de JFrame. Dessa mesma maneira você pode acessar qualquer outra classe da biblioteca do Java. Assim você tem toda a expressividade e poder do Ruby, somado a quantidade enorme de bibliotecas do Java.

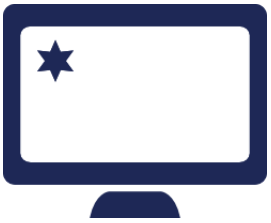
Para saber mais: Suporte a closure com JRuby

O JRuby permite a passagem de blocos de código como argumento para métodos do java que recebem como parâmetro uma interface que define apenas um método, assim como o futuro suporte a closures prometido para o Java 8. No exemplo acima poderíamos ter passado o ActionListener para o botão sem necessidade de escrever uma classe só para isso, e nem mesmo seria preciso instanciar um objeto, fazendo desta forma:

```
botao.add_action_listener do |evento|
  Swing::JOptionPane.showMessageDialog(nil, "ActionListener em
closure")
```

```
end
```

Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

Conheça a Alura.

CAPÍTULO ANTERIOR:

Apêndice: Design Patterns em Ruby

PRÓXIMO CAPÍTULO:

Apêndice: Deployment

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter