Capítulo 2

A linguagem Ruby

"Ruby is simple in appearance, but is very complex inside, just like our human body"
— Yukihiro Matsumoto, Criador da linguagem Ruby

2.1 – A história do Ruby e suas características

Ruby nasceu em 1993 mas foi apresentada ao público pela primeira vez somente em 1995, pelo seu criador: **Yukihiro Matsumoto**, mundialmente conhecido como **Matz**. É uma linguagem orientada a objetos, com tipagem forte e dinâmica. Curiosamente é uma das únicas linguagens nascidas fora do eixo EUA – Europa

que atingiram enorme sucesso comercial.

Uma de suas principais características é a expressividade que possui. Teve-se como objetivo desde o início que fosse uma linguagem muito simples de ler e ser entendida, para facilitar o desenvolvimento e manutenção de sistemas escritos com ela.

Ruby é uma linguagem interpretada e, como tal, necessita da instalação de um interpretador em sua máquina antes de executar algum programa.

2.2 - Instalação do interpretador

Antes da linguagem Ruby se tornar popular, existia apenas um interpretador disponível: o escrito pelo próprio Matz, em C. É um interpretador simples, sem

modernas de interpretadores como a compilação em tempo de execução (conhecida como JIT). Hoje a versão mais difundida é a 2.0, também conhecida como **YARV** (Yet Another Ruby VM), já baseada em uma máquina virtual com recursos mais avançados.

A maioria das distribuições Linux possuem o pacote de uma das última versões estáveis pronto para ser instalado. O exemplo mais comum é o de instalação para o Ubuntu:

```
sudo apt-get install ruby2.0 ruby2.0-dev build-essential libssl-dev
zlib1g-dev \
 ruby-switch
```

O comando acima instala além do Ruby, pacotes necessários para instalar gems que contenham código C, não sendo essenciais para quem quer apenas executar código Ruby.

Apesar de existirem soluções prontas para a instalação do Ruby em diversas plataformas (*one-click-installers* e até mesmo gerenciadores de versões de Ruby), sempre é possível baixá-lo pelo site oficial:

http://ruby-lang.org

Após a instalação, não deixe de conferir se o interpretador está disponível na sua variável de ambiente *PATH*:

```
$ ruby --version
ruby 2.0.0p247 (2013-06-27 revision 41674) [x86_64-darwin13.0.0]
```

Tire suas dúvidas no GUJ Respostas

O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

Faça sua pergunta.

2.3 – RubyGems

O Ruby possui um gerenciador de pacotes bastante avançado, flexível e eficiente: **RubyGems**. As *gems* podem ser vistas como bibliotecas reutilizáveis de código Ruby, que podem até conter algum código nativo (em C, Java, .Net). São análogos aos *jars* no ambiente Java, ou os *assemblies* do mundo .Net. RubyGems é um sistema gerenciador de pacotes comparável a qualquer um do mundo *NIX, como os `.debs` do `apt-get`, os `rpms` do `yum`, entre outros.

Até a versão 1.9.3 do Ruby, para sermos capazes de instalar e utilizar as centenas de *gems* disponíveis, precisavámos instalar além do interpretador Ruby, o RubyGems. Com a versão 2.0, o comando `gem` já vem incluso no pacote `ruby2.0`

> **Mais sobre instalação**
>
> No final da apostila, apresentamos um apêndice exclusivo sobre a instalação completa do interpretador e ferramentas essenciais nas plataformas Linux, Mac e Windows.

Após a instalação do Ruby podemos instalar os pacotes necessários para a criação de uma aplicação Rails com o comando **rails new**:

```
rails new nome_da_aplicacao
```

2.4 - Bundler

Ao desenvolver novas aplicações utilizando Ruby, notaremos que uma série de funcionalidades serão necessárias - ler e parsear JSON, fazer autenticação de

usuário, entre outras coisas. A maioria dessas funcionalidades já foi implementada em alguma *gem*, e para usufruir desses recursos basta colocar a gem em nossa aplicação.

Nossos projetos podem precisar de uma ou mais gems para funcionar corretamente. Se quiséssemos compartilhar nosso projeto com a comunidade, como faríamos para fixar as gems necessárias para o funcionamento do projeto? Escreveríamos um arquivo de texto especificando quais gems o outro desenvolvedor deve instalar? Esse seria um processo manual? Há diversas formas de fazermos um bom controle de versão de código, mas como poderíamos, no mundo Ruby, controlar as *dependências* de gems necessárias ao nosso projeto?

Uma forma eficaz de controlar as dependências de nosso projeto Ruby é utilizar uma gem chamada *Bundler*. Instalar o *bundler* é bem simples. Basta rodar o comando:

```
gem install bundler
```

Com o Bundler, declaramos as dependências necessárias em um arquivo chamado *Gemfile*. Esse aquivo deverá conter, primeiramente, a fonte de onde onde o Bundler deve obter as gems e em seguida a declaração das dependências que usaremos no projeto.

```
source "http://rubygems.org"

gem "rails"
gem "devise"
```

Note que a declaração *source* indica de onde o Bundler obterá as gems (no caso mais comum, de `http://rubygems.org`) e a declaração *gem* indica a gem declarada como dependência.

Podemos então rodar o comando `bundle` (atalho para o comando `bundle install`) para obter as gems necessárias para o nosso projeto. Note que ao estruturar a declaração das gems dessa forma não declaramos versões específicas para as gems, portanto a versão mais recente de cada uma dela será baixada para nosso repositório de gems local. Poderíamos fazer o seguinte:

```
gem "rails", "4.0.0"
```

Dessa forma explicitaríamos o uso da versão 4.0.0 da gem `rails`. Uma outra forma válida seria:

```
gem "rails", "~> 4.0.0"
```

Assim obteríamos a gem em uma versão maior ou igual à versão 4.0.0, mas

menor que a versão 4.1.0.

Ao rodar o comando `bundle` será gerado um novo arquivo chamado *Gemfile.lock*, que especifica todas as gems obtidas para aquele Gemfile e sua respectiva versão baixada. O Gemfile.lock é uma boa alternativa para congelar as versões das gems a serem utilizadas, uma vez que ao rodarmos o comando `bundle` sobre a presença de um Gemfile.lock, as versões presentes nesse arquivo serão utilizadas para especificar as gems a serem baixadas.

2.5 – Outras implementações de interpretadores Ruby

Com a popularização da linguagem Ruby, principalmente após o surgimento do Ruby on Rails, implementações alternativas da linguagem começaram a surgir. A maioria delas segue uma tendência natural de serem baseados em uma Máquina Virtual ao invés de serem interpretadores simples. Algumas implementações possuem até compiladores completos, que transformam o código Ruby em alguma linguagem intermediária a ser interpretada por uma máquina virtual.

A principal vantagem das máquinas virtuais é facilitar o suporte em diferentes plataformas. Além disso, ter código intermediário permite otimização do código

em tempo de execução, feito através da **JIT**.

JRuby

JRuby foi a primeira implementação alternativa completa da versão 1.8.6 do Ruby e é a principal implementação da linguagem Java para a JVM. Com o tempo ganhou compatibilidade com as versões 1.8.7 e 1.9.3 na mesma implementação.

Como roda na JVM, não é um simples interpretador, já que também opera nos modos de compilação AOT (*Ahead Of Time*) e JIT (*Just In Time*), além do modo interpretador tradicional *Tree Walker*.

Uma de suas principais vantagens é a interoperabilidade com código Java existente, além de aproveitar todas as vantagens de uma das plataformas de execução de código mais maduras (Garbage Collector, Threads nativas, entre outras).

Muitas empresas apoiam o projeto, além da própria Oracle (após ter adquirido a Sun Microsystems), outras empresas de expressão como IBM, Borland e ThoughtWorks mantém alguns projetos na plataforma e algumas delas têm funcionários dedicados ao projeto.

A versão atual do JRuby é a 1.7 e é compatível com o Ruby 1.9.3. A versão compatível com o Ruby 2.0 ainda está em desenvolvimento e nela será retirado o suporte para o Ruby 1.8 além de precisar de depender do Java 7.

http://jruby.org

IronRuby

A comunidade .Net também não ignorou o sucesso da linguagem e patrocinou o projeto **IronRuby**, que era mantido pela própria *Microsoft*. IronRuby foi um dos primeiros projetos verdadeiramente de código aberto dentro da Microsoft. Em 2010 a Microsoft parou de manter o projeto e a versão atual do IronRuby, 1.1.3, implementa apenas até o Ruby 1.9.2.

http://ironruby.net

Rubinius

Criada por Evan Phoenix, **Rubinius** é um dos projetos que tem recebido mais atenção da comunidade Ruby, por ter o objetivo de criar a implementação de Ruby com a maior parte possível do código em Ruby. Além disso, trouxe ideias de máquinas virtuais do SmallTalk, possuindo um conjunto de instruções (bytecode)

próprio e implementada em C/C++. O Rubinius está sempre atualizado com as últimas versões do Ruby e roda no MacOS X e em sistemas Unix/Linux. Windows ainda não é suportado por ele.

http://rubini.us

RubySpec

O projeto Rubinius possui uma quantidade de testes enorme, escritos em Ruby, o que incentivou a iniciativa de especificar a linguagem Ruby. O projeto RubySpec (http://rubyspec.org/) é um acordo entre os vários implementadores da linguagem Ruby para especificar as características da linguagem Ruby e seu comportamento, através de código executável, que funciona como um **TCK** (*Test Compatibility Kit*).

RubySpec tem origem na suíte de testes de unidade do projeto Rubinius, escritos com uma versão mínima do RSpec, conhecida como **MSpec**. O RSpec é uma ferramenta para descrição de especificações no estilo pregado pelo *Behavior Driven Development*.

Avi Bryant durante seu *keynote* na **RailsConf de 2007** lançou um desafio à comunidade:

*"I'm from the future, I know how this story ends. All the people who are saying you can't implement Ruby on a fast virtual machine are wrong. That machine already exists today, it's called **Gemstone**, and it could certainly be adapted to Ruby. It runs Smalltalk, and Ruby essentially is Smalltalk. So adapting it to run Ruby is absolutely within the realm of the possible."*

Ruby e Smalltalk são parecidos, então Avi basicamente pergunta: por que não criar máquinas virtuais para Ruby aproveitando toda a tecnologia de máquinas virtuais para SmallTalk, que já têm bastante maturidade e estão no mercado a tantos anos?

Integrantes da empresa Gemstone, que possui uma das máquinas virtuais para SmallTalk mais famosas (Gemstone/S), estavam na plateia e chamaram o Avi Bryant para provar que isto era possível.

Na RailsConf de 2008, o resultado foi que a Gemstone apresentou o produto que estão desenvolvendo, conhecido como **Maglev**. É uma máquina virtual para Ruby, baseada na existente para Smalltalk. As linguagens são tão parecidas que apenas poucas instruções novas tiveram de ser inseridas na nova máquina virtual.

Os números apresentados são surpreendentes. Com tão pouco tempo de

desenvolvimento, conseguiram apresentar um ganho de até 30x de performance em alguns micro benchmarks. Um dos testes mais conhecidos sobre a performance de interpretadores e máquinas virtuais Ruby feito por Antonio Cangiano em 2010 - **The Great Ruby Shootout** (http://programmingzen.com/2010/07/19/the-great-ruby-shootout-july-2010/) - mostra que apesar de destacar-se em alguns testes, o Maglev mostra-se muito mais lento em alguns outros, além do alto consumo de memória.

Ruby Enterprise Edition

Para melhorar a performance de aplicações Rails e diminuir a quantidade de memória utilizada, **Ninh Bui**, **Hongli Lai** e **Tinco Andringa** (da Phusion) modificaram o interpretador Ruby e lançaram com o nome de **Ruby Enterprise Edition**.

As principais modificações no REE foram no comportamento do *Garbage Collector*, fazendo com que funcione com o recurso de *Copy on Write* disponível na maioria dos sistemas operacionais baseados em UNIX (Linux, Solaris, etc.).

Outra importante modificação foi na alocação de memória do interpretador, com o uso de bibliotecas famosas como `tcmalloc`. Os desenvolvedores da Phusion

ofereceram as modificações (*patches*) para entrar na implementação oficial do Ruby.

A implementação oficial do Ruby, lançada na versão 1.9, codinome *YARV*, adicionou algumas novas construções a linguagem em si, mas também resolveu muitos dos problema antes endereçados pela *REE*.

Como consequência de tudo isso, a própria *Phusion*, mantenedora do *REE*, anunciou o fim da manutenção do projeto. Caso queira ler mais a respeito do assunto, visite o anúncio oficial em: *http://blog.phusion.nl/2012/02/21/ruby-enterprise-edition-1-8-7-2012-02-released-end-of-life-imminent/*.

e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

CAPÍTULO ANTERIOR:

Agilidade na Web

PRÓXIMO CAPÍTULO:

Ruby básico

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter