Suporte a TAD e POO em Ruby

Felipe Emídio Esteves da Silva
Vinícius dos Santos Oliveira

10 de dezembro de 2014

Sumário

1 Introdução

1.1 História

A linguagem Ruby foi criada, em 1995, no Japão por Yukihiro Matsumoto (também conhecido como "Matz"), esta é uma linguagem interpretada multiparadigma de tipagem forte e dinâmica. Foi inspirada em Python, Perl, Smalltalk, Eiffel, Ada e Lisp. Ruby surgiu para ser uma linguagem de script mais poderosa que Perl e mais orientada a objetos que Python, de acordo com Matz.

O nome Ruby foi escolhido pelo fato da pedra rubi ser a pedra zodiacal de um colega de Yukihiro Matsumoto.

1.2 Características Gerais

Ruby é uma linguagem puramente orientada a objetos e mesmo valores primitivos são objetos. Quando um programa Ruby começa a executar, ele já se encontra em um ambiente orientado a objetos, diferente de Java, que requer a ponte `public static void main`.

Ruby é uma linguagem bem dinâmica que foi projetada de tal forma que se torna fácil estender qualquer uma das abstrações que já foram feitas. Não só a tipagem é dinâmica, como novas definições podem ser adicionadas a abstrações já definidas ou antigas abstrações podem ser removidas ou modificadas. Muitas das decisões da linguagem também ajudam a encontrar pontos de customização em códigos-fonte feitos na linguagem, como o fato de qualquer operador ser sobrecarregável, o uso extensivo de referências e algumas pequenas decisões que mantém a sintaxe consistente, como o fato de todo método implicitamente retornar o valor da última instrução executada e `foo.bar = 42` ser na verdade uma chamada ao método `bar=` do objeto `foo` passando `42` como argumento, o que elimina a necessidade de getters e setters que tenham uma sintaxe alienígena e inconsistente com o resto das abstrações.

A sintaxe de Ruby é parecida com C, Java e Python para a especificação de literais de primitivos e tipos comuns (como string):

- Um inteiro: `42`
- Um booleano: `true`
- Uma string: `"abc"`

A sintaxe para especificação de listas literais é parecida com JavaScript e Python:

Uma lista: `[1, 2, 3]`

Enquanto a especificação para dicionários literais é inspirada pela abordagem da linguagem PHP (eca!):

```
{"dog" => "canine", 12 => "dodecine"}
```

O valor nulo é especificado através da palavra chave `nil`.

A palavra reservada `end` é usada para terminar os blocos mais comuns da linguagem. O token `end` indica o final da definição de funções, classes, módulos, métodos e outros. Existe um `;` opcional para indicar o final de algumas expressões, mas um pulo de linha possui o mesmo efeito. Como exemplo, abaixo segue a definição de uma classe vazia.

```
class Foobar
end
```

Também é possível ter o mesmo efeito com o seguinte código:

```
class Foobar; end
```

Mas o código a seguir não é código Ruby válido:

```
class Foobar end
```

Funções são definidas através da palavra-chave `def`, nome da função, argumentos separados por vírgulas e a palavra-chave `end`. Exemplo:

```
def print3 a, b, c
  print a
  print "\n"
  print b
  print "\n"
  print c
  print "\n"
end
```

A função é invocada através de seu nome e os argumentos separados por vírgula:

```
print3 "foo", "bar", "baz"
```

Em Ruby, a expressão lambda é feita através de um bloco delimitado por chaves. Instruções são colocadas dentro das chaves. Caso seja necessário criar uma função que receba argumentos, os argumentos são especificados no início do bloco, delimitados por barras verticais. Exemplo:

```
5.times { |x, unused1, unused2| print x; print "\n" }
```

Uma sintaxe alternativa para alcançar o mesmo efeito seria:

```
5.times do
  |x, unused1, unused2| print x; print "\n"
end
```

Variáveis que ainda não foram definidas possuem o valor `nil`, e, quando algum valor é a elas atribuído, o seu conteúdo é modificado de acordo. Isso é importante no caso de Mixins, que permitem várias abstrações de dados serem misturadas, de forma análoga a herança múltipla. Entretanto, no caso de Ruby, o problema do diamante não ocorre, pois todas as variáveis de mesmo nome são compartilhadas. Entretanto, essa colisão de nomes pode ser indesejada e também resultar em erros.

2 Exemplo de Ruby

Esse é um exemplo interessante que demonstra como Ruby é dinâmica e flexível, onde é definida uma abstração reutilizável que reusa abstrações desconhecidas no momento da definição (referência a função `each` dentro da abstração `Inject.inject`) e, em vez de reutilizar a nova abstração para uma nova abstração (e.g. reusar a nova abstração durante a definição de uma nova classe), a nova abstração é usada para modificar uma antiga abstração (i.e. abstração `Inject` é usada para modificar a abstração `Array`). O mais digno de apláusos é que a abstração modificada depois de sua definição não é uma abstração qualquer, mas sim uma das abstrações bases para toda a linguagem, a abstração da classe `Array`.

```
module Inject
  def inject(n)
      each do |value|
        n = yield(n, value)
      end
      n
  end
  def sum(initial = 0)
    inject(initial) { |n, value| n + value }
  end
  def product(initial = 1)
    inject(initial) { |n, value| n * value }
  end
end
```

```ruby
class Array
  include Inject
end

[ 1, 2, 3, 4, 5 ].sum # retorna 15
[ 1, 2, 3, 4, 5 ].product # retorna 120
```

3 TAD em Ruby

Ruby permite o agrupamento de funções relacionadas dentro de entidades chamadas classes, que possuem o usual significado da proposta de programação orientada a objetos que já é bem comum atualmente.

```ruby
class MinhaClassePorradona
end
```

Uma classe introduz um novo tipo e novas variáveis desse tipo podem ser instanciadas usando o método `new`.

```ruby
class MinhaClassePorradona
end

variavel = MinhaClassePorradona.new
```

Quando uma função é definida dentro de uma classe, ela passa a ser um método dessa classe.

```ruby
class MinhaClassePorradona
    def minhaFuncaoPorradonaAgoraEhUmMetodo
    end
end
```

Dentro de uma classe, dois novos escopos são introduzidos: O escopo de classe e o escopo de instância. Um método declarado no escopo de instância (padrão) requer uma instância para ser executado e um método declarado no escopo de classe (dito método estático) não precisa de uma instância para ser executado.

Para declarar um método estático, ele precisa ser prefixado com o nome da classe e um ponto. Exemplo:

```ruby
class MinhaClassePorradona
    def MinhaClassePorradona.meuMetodoEstatico
    end
```

```
end
```

O método não-estático recebe implicitamente um argumento, o argumento `self`, que referencia o próprio objeto.

Ao definir os métodos, as variáveis podem ser usadas sem a necessidade de terem sido antes declaradas. Toda variável começa com o valor `nil`, até que um novo valor seja a ela atribuído. Para acessar uma variável que possui escopo de instância, o caractere `@` deve ser precedido ao nome da variável. Já para acessar uma variável que possui escopo de classe, `@@` deve ser usado.

Variáveis com escopo de classe podem ser inicializadas diretamente em um bloco que define a classe. Um método `initialize` pode ser definido, que age como o construtor da classe (i.e. será chamado para todos os objetos que forem alocados), e pode ser usado para inicializar as variáveis com escopo de instância. O exemplo abaixo demonstra esses conceitos:

```
class MinhaClasse
    @@w = 0

    def initialize
        @x, @y, @z = 0, 1, 2
    end

    def x
        @x
    end

    def x= x
        @x = x
    end

    def MinhaClasse.w
      @@w
    end

    def MinhaClasse.w= w
      @@w = w
    end
end

var, var2 = MinhaClasse.new, MinhaClasse.new
puts var.x # imprime 0
puts var2.x # imprime 0
```

```
puts MinhaClasse.w # imprime 0

var.x = 1
MinhaClasse.w = 1
puts var.x # imprime 1
puts var2.x # imprime 0
puts MinhaClasse.w # imprime 1
```

O único jeito de ter acesso as variáveis de instância e de classe é através dos métodos definidos dentro da classe. Como não é necessário o uso de parênteses para chamar funções, esse detalhe não suja a sintaxe e faz parecer que as variáveis estão sendo diretamente manipuladas. O controle de acesso é então feito através das funções que são pontes para os atributos internos e suas respectivas operações. Há ainda especificadores mais elaborados, porém que são direcionados a POO.

Para poluir menos as definições de classes, diminuindo ou mesmo eliminando as definições de getters e setters que são indênticos para todas as classes, é possível fazer uso das facilidades `attr_reader`, `attr_writer` e `attr_accessor`. Seu uso é resumido no código abaixo:

```
class Foobar
    attr_reader :x

    def initialize
        @x = "42"
    end
end

foobar = Foobar.new
puts foobar.x # imprime 42

class Foobar2
    attr_writer :x

    def what_value_is_x
        @x
    end
end

foobar2 = Foobar2.new
foobar2.x = 42
puts foobar2.what_value_is_x # imprime 42
```

```ruby
class Foobar3
    attr_accessor :x
end

foobar3 = Foobar3.new
foobar3.x = 42
puts foobar3.x # imprime 42
```

attr_reader, attr_writer e attr_accessor são métodos, e, quando usados dessa forma, são executados para a definição da classe. Esse é um exemplo de metaprogramação em Ruby. Uma chamada de método é uma passagem de mensagem para um objeto e inclui não só a execução do método capaz de responder aquela mensagem, como também a busca por tal método. Quando o método attr_accessor é usado dessa forma, estamos usando como receptor (i.e. o objeto que responde a mensagem) o objeto que corresponde a classe que está sendo definida (i.e. um metaobjeto), que indiretamente herda da classe Module. Assim, Module faz parte do caminho de busca pelo método attr_accessor, e é exatamente nessa classe que tal método está definido. Logo, caso queiramos implementar facilidades parecidas com attr_accessor, só precisamos adicionar novos métodos a classe Module, como demontra o código a seguir:

```ruby
class Module
  def var method_name
    inst_variable_name = "@#{method_name}".to_sym
    define_method method_name do
      instance_variable_get inst_variable_name
    end
    define_method "#{method_name}=" do |nv|
      instance_variable_set inst_variable_name, nv
    end
  end
end

class Foo
  var :bar
end

f = Foo.new
p f.bar     #=> imprime nil
f.bar = 42
```

```
p f.bar      #=> imprime 42
```

Encapsular e esconder. Ruby dá suporte a tipos abstratos de dados.

4 OO e ruby

Classes são definidas como foi mostrado anteriormente na seção sobre TAD em Ruby. Algumas notas extras que os autores desse documento acharam que seriam melhor agrupadas em uma seção diferente, essa seção, desse documento:

- Em Ruby, classes nunca são fechadas, isto é, você sempre pode adicionar novos métodos a uma classe já existente. Também é possível remover antigos métodos, como mostra o exemplo a seguir:

```
class Foobar
end

$f = Foobar.new

def test
    begin
        $f.someMethod
    rescue NoMethodError
        puts "Method␣\"someMethod\"␣not␣found"
    end
end

test

class Foobar
    def someMethod
        puts "someMethod␣called"
    end
end

test

class Foobar
    remove_method :someMethod
end
```

```
test
```

O uso do prefixo $ no nome da variável é feito para especificar o intento de acessar uma variável global.

- Ruby é uma linguagem muito dinâmica, permitindo que métodos que já foram definidos sejam removidos ou modificados. Devido a isso, torna-se ambíguo se a definição de um novo método que tenha o mesmo nome que um antigo método é uma sobrecarga ou uma substituição do antigo método de mesmo nome. Assim sendo, já seria impossível suportar sobrecarga de métodos baseado no número de argumentos sem complicar a sintaxe. E como Ruby é uma linguagem dinamicamente tipada, sintaxe para sobrecarga de funções baseada nos tipos dos argumentos também complicaria a linguagem. Assim sendo, perfeitamente justificado, Ruby não suporta sobrecarga de métodos.

4.1 Herança

Ruby só suporta herança simples e a superclasse de uma classe é definida no momento em que a primeira das definições da classe é avaliada. Uma vez que a superclasse foi definida, ela não mais pode ser mudada. Quando uma definição de classe é feita, uma classe-mãe não é especificada e essa classe está sendo definida pela primeira vez, a classe `Object` é usada como super-classe da classe sendo definida. Para especificar a super-classe para uma classe, é usado o caractere < após o nome da classe, seguido do nome da super-classe, como mostra o exemplo a seguir:

```
class Mother
end

class Child < Mother
end
```

Caso uma classe seja especificada, a classe já tenha uma super-classe e a classe mãe especificada seja diferente da super-classe atual, um erro é disparado.

Todas as classes herdam de `Object`, diretamente (quando uma classe mãe não é especificada, que por regra da linguagem significa que Object é a classe-mãe) ou indiretamente (quando uma classe que herda de Object direta ou indiretamente é usada como classe-mãe).

A palavra-chave `super` é usada dentro de um método para se referir ao método de mesmo nome na classe-mãe. Isso é útil para, por exemplo, chamar o construtor da classe pai na classe filha.

Ruby não suporta classes abstratas, mas não há a necessidade de Ruby suportar classes abstratas, pois a motivação de classes abstratas é permitir uma mesma visualização para classes que tenham implementações diferentes de métodos comuns. A motivação não é válida em Ruby, pois (1) Ruby é uma linguagem dinâmica e uma variável qualquer pode armazenar uma referência para um valor/objeto de qualquer tipo e (2) Ruby é uma linguagem de tipagem dinâmica, então não se pode fazer testes em tempo de compilação para garantir que classes estão corretamente implementando uma interface. Chamar um método de um objeto vai apenas enviar uma mensagem aquele objeto e não importa sintaticamente para o código chamador onde o método capaz de responder a chamada foi implementado.

Ruby não suporta herança múltipla, mas sua emulação, de forma limitada, é possível através do uso de Mixins. Esse documento não tem o objetivo de mostrar como Ruby suporta Mixins.

4.2 Controle de Acesso

Em Ruby, o controle de acesso não pode ser customizado para os atributos, que são sempre acessíveis somente através de métodos das classes que os definem. Em Ruby, controle de acesso só pode ser customizado para os métodos. Os métodos `private`, `protected` e `public` recebem um método como argumento e alteram sua visibilidade. No momento em que é feita a requisição de envio de uma mensagem a um objeto, a visibilidade é verificada e o controle de acesso é feito. Para liberar ou rejeitar o acesso, Ruby adota uma interessante estratégia de verificar como o objeto para o qual a tentativa de envio de mensagem é referenciado. Se for uma referência implícita (i.e. apenas o nome do método é usado, sem nenhum referência de qual é o objeto, que é implicitamente assumido como `self`), então o acesso sempre é liberado. Se for uma referência explícita e o objeto for do mesmo tipo da classe do método, então o acesso é liberado caso o método seja público ou protegido. Nos outros casos, o acesso só é liberado se for um método público. O código abaixo exemplifica essas regras:

```
class ImplicitRef
  def foobar
    # acesso implicito (nenhum objeto especificado)
    pub; pro; pri
  end
```

```
  public; def pub; end
  protected; def pro; end
  private; def pri; end
end

class ExplicitSelf
  def foobar
    # acesso explicito por um metodo da mesma classe
    self.pub; self.pro
  end

  def foobar2 x
    x.pub; x.pro
  end

  public; def pub; end
  protected; def pro; end
  private; def pri; end
end

ImplicitRef.new.foobar
ExplicitSelf.new.foobar
ExplicitSelf.new.foobar2 ExplicitSelf.new

class Whatever
  public; def pub; end
  protected; def pro; end
  private; def pri; end
end

# acesso explicito por um metodo de uma classe
# diferente
Whatever.new.pub
```

Em classes filhas, o controle de acesso a métodos pode ser modificado, como demonstra o código abaixo:

```
class Mother
  public; def pub; end
  protected; def pro; end
  private; def pri; end
```

```
end

class Child < Mother
  public :pri # Muda a visibilidade do metodo
end

Child.new.pri
```

O controle de acesso pode ainda ser ignorado caso a mensagem seja enviada ao objeto através do método `send`, como demonstra o código abaixo:

```
class Whatever
  private; def pri; end
end

Whatever.new.send :pri
```

4.3 Vinculação Dinâmica

Ruby usa vinculação dinâmica de nome, ou seja, variáveis não precisam ser especificadas, pois todas são referencias a objetos de qualquer classe, então todas as variáveis são polimórficas.

4.4 Avaliação

Como Ruby é uma linguagem puramente orientada a objetos, seu uso é intuitivo. Ruby não tem suporte a herança múltipla, classes abstratas ou interface. Porém o uso de Mixins pode ser usado pra suprir a falta de herança múltipla. E ainda, seus controle de acesso a membros da classe é mais fraco que o de Java, pois só se aplica a métodos, não atributos, e ainda pode ser facilmente ignorado.