howto

Ruby Metaprogramming

por Nando Vieira

Ruby Metaprogramming, Nando Vieira, 2ª versão

Veja outros e-books e screencasts publicados pela Hellobits no site do HOWTO

Conteúdo

Capítulo 1

Introdução

Poucas coisas no Ruby são tão pouco entendidas como a metaprogramação. Embora seja uma técnica extremamente útil, ela exige muito mais conhecimento da linguagem como um todo, que pode levar um certo tempo até que seja adquirido.

A metaprogramação faz parte do Ruby. Isso é tão verdade que você não consegue distinguir trechos de um código Ruby e afirmar se aquilo é ou não metaprogramação. A metaprogramação faz parte do dia-a-dia do desenvolvedor Ruby. Ou pelo menos deveria.

Infelizmente, muitos consideram a metaprogramação como pura *magia negra*, muito mais pelo fato de não entenderem certos aspectos da linguagem, do que pelo modo como o Ruby executa tal técnica.

> “Não existe magia negra. Apenas Ruby.”

O objetivo deste livro é ser um guia com técnicas utilizadas por desenvolvedores Ruby de todo o mundo, com exemplos práticos de como criar DSLs, executar e definir métodos dinamicamente, dentre muitos outros exemplos.

—Nando Vieira, Janeiro de 2011

Capítulo 1

Entendendo o self

No Ruby, o `self` será sempre uma referência ao receiver atual e pode ser diferente dependendo do contexto em que você estiver. Por exemplo, quando estamos no namespace global, nosso `self` será o objeto `main`. Já em uma classe, nosso `self` será a própria classe.

```
puts self
#=> main

class Thing
  puts self
end
#=> Thing
```

Toda vez que você executa um método, o Ruby irá verifica se ele existe no receiver padrão, `self`, a menos que você especifique explicitamente quem será o receiver.

```
class Thing
  def do_something
    puts "doing something"
  end

  def do_something_else
    do_something
  end
end
```

No método `Thing#do_something_else` poderíamos usar `self.do_something`, mas isso faria com que nosso código apenas ficasse mais poluído. No entanto, definir o receiver é uma coisa muito comum e que, você pode até não se dar conta, mas o faz constantemente quando escreve código Ruby.

```ruby
numbers = [3,1,2]
numbers.sort
#=> [1,2,3]
```

Na prática, o receiver é especificado antes do ponto na chamada de métodos, como em `numbers.sort`.

Variáveis de instância

Além de ser o receiver padrão, `self` também é o responsável por armazenar variáveis de instância de um objeto. Veja o exemplo abaixo.

```ruby
class Person
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

john = Person.new("John Doe")
john.name
#=> "John Doe"
```

A instância da classe `Person` possui uma única variável de instância associada ao seu objeto (`self`) que é retornada pelo método `Person#name`. Analogamente, podemos definir variáveis de instância em qualquer objeto, como classes.

```ruby
class Person
  def self.count
    @count ||= 0
  end

  def self.count=(increment)
```

```
    @count = increment
  end

  def initialize(name)
    @name = name
    self.class.count += 1
  end

  def name
    @name
  end
end

john = Person.new("John Doe")
Person.count
#=> 1
```

O exemplo acima mostra como variáveis de instância podem ser usadas em contextos diferentes. Primeiro, estamos definindo um contador de instâncias da classe `Person`, cujo valor será armazenado em `@count`. Depois, em nossa própria instância, definimos o nome com a variável `@name`.

Capítulo 1

Entendendo classes Singleton

O nome Singleton usado pelo Ruby nada tem a ver com o Singleton Pattern, que também está disponível com a biblioteca Singleton.

Todo objeto do Ruby está associado a duas classes: a classe que a instanciou e uma classe anônima, escondida, específica do objeto. Esta classe anônima é chamada de *Singleton Class*, mas antes de ter um nome oficial também era chamada de *anonymous class*, *metaclass*, *eigenclass* ou *ghost class*.

A sintaxe mais comum para acessar a classe Singleton é

```
class << object
end
```

onde `object` é o objeto cuja classe Singleton você quer. É muito comum vermos algo como o exemplo à seguir para definir métodos em uma classe.

```
class Person
  class << self
    def count
      @count ||= 0
    end
  end
end
```

Aqui, estamos definindo um método na classe Singleton do objeto `Person` (lembre-se: tudo no Ruby é objeto, inclusive classes). Como consequência, isso irá definir o método `Person.count`. O efeito é exatamente o mesmo que

```
class Person
  def self.count
```

```
    @count ||= 0
  end
end
```

No Ruby 1.9.2, foi adicionado o método `Object#singleton_class`, que é apenas um atalho para a sintaxe `class << self; self; end`. Em versões mais antigas, você pode injetar este método com um código como este

```
class Object
  def singleton_class
    class << self; self; end
  end unless respond_to?(:singleton_class)
end
```

Quando você cria uma classe Singleton em um objeto (uma instância), não poderá mais utilizar o método `Marshal.dump`, já que a biblioteca Marshal não suporta objetos com classes Singleton (ela irá lançar a exceção `TypeError: singleton can't be dumped`). A única maneira de fazer isso e ainda poder utilizar o Marshal é utilizando o método `Object#extend`.

Toda vez que injeta métodos em um objeto, eles são adicionados como métodos singleton. O que é realmente importante saber é que estes métodos pertecem unicamente ao objeto em que foram definidos, não afetando nenhum outro objeto da hierarquia.

```
string = "Hi there!"
another_string = "Hi there!"

def string.to_yo
  self.gsub(/\b(Hi|Hello)( there)\b?!?/, "Yo! Wassup?")
end

string.to_yo
#=> "Yo! Wassup?"

another_string.respond_to?(:to_yo)
#=> false
```

E para provar que o método `string#to_yo` é singleton, podemos utilizar o método `Object#singleton_methods`.

```
string.singleton_methods
#=> ["to_yo"]

another_string.singleton_methods
#=> []
```

Agora, sabendo que você pode adicionar métodos em um objeto com uma sintaxe como `def object.some_method; end`, perceba que é exatamente isso que fazemos quando definimos um método em uma classe; a única diferença é que passamos o próprio `self`.

```
class Person
  def self.say_hello
    "Hello there!"
  end
end

Person.singleton_methods
#=> ["say_hello"]
```

Com base nesse exemplo, é possível afirmar que **métodos de classe não existem no Ruby**! Pelo menos não no sentido de métodos estáticos! O que acontece é que estes métodos pertencem a um objeto, que por acaso é uma classe.

Capítulo 1

Classes

No Ruby, classes e módulos são como qualquer objeto. A palavra-chave `defined?` retorna informação sobre a expressão que está recebendo.

```
class Person
end

defined?(Person)
#=> "constant"
```

Como você pode ver, classes nada mais são do que simples constantes. Embora normalmente usamos as palavras-chave `class` e `module` para criarmos novas definições, você pode atingir o mesmo resultado utilizando as classes `Class` e `Module`.

```
cls = Class.new

defined?(cls)
#=> "local-variable"

obj = cls.new
obj.instance_of?(cls)
#=> true
```

Uma coisa muito interessante de classes dinâmicas é que elas não possuem nome.

```
cls.name
#=> nil
```

Mas se classes dinâmicas não possuem nome, muitas coisas baseadas nessa informação podem não funcionar. Na verdade, o Ruby utiliza um truque muito interessante para definir qual o nome da classe.

```
cls = Class.new
cls.name
#=> nil

MyClass = cls
MyClass.name
#=> "MyClass"
```

O nome da classe é inferido da constante ao qual ela foi atribuída. No entanto, este nome é definido apenas quando essa classe ainda não tem um nome.

```
cls = Class.new
cls.name
#=> nil

MyClass = cls
MyClass.name
#=> "MyClass"

OtherClass = cls
OtherClass.name
#=> "MyClass"
```

Se você quiser, não precisa nunca mais utilizar a palavra-chave `class` para definir suas classes. Claro que seu código perderá um pouco da elegância, mas esta técnica pode ser útil.

Também é possível definir qual a superclasse de uma classe dinâmica. Basta passar uma classe como argumento do método `Class.new`.

```
class Person; end

Author = Class.new(Person)
```

```ruby
Author.ancestors
#=> [Author, Person, Object, Kernel, BasicObject]
```

Para definir métodos desta classe dinâmica basta você passar um bloco, que irá ser o “corpo” de sua classe.

```ruby
Person = Class.new do
  def self.say(message)
    puts message
  end
end

Person.say "Hi there!"
#=> "Hi there!"
```

Capítulo 1

Módulos & Mixins

Conforme a popularidade do Ruby aumenta, a chance de diferentes bibliotecas terem classes com nomes iguais também aumenta. Imagine que você está desenvolvendo um aplicativo de lista de tarefas e tem uma classe chamada `Task`, que salva os dados em banco de dados.

```
class Task < ActiveRecord::Base; end
```

Se neste aplicativo você fosse utilizar uma versão antiga do Rake, teria um problema de colisão de classes, já que ele também implementava a classe `Task`. E como o Ruby possui classes abertas, você nem mesmo ficaria sabendo ou iria saber da pior maneira possível.

A solução para este problema foi bastante simples. Todos as classes que estavam no *toplevel* foram movidas para um módulo `Rake`[1].

```
require "rake"
Rake.class
#=> Module

Rake::Task.class
#=> Class
```

Além de serem ótimos para definir *namespaces*, módulos possuem uma outra utilidade. Eles permitem compartilhar funcionalidade entre diversos objetos com um conceito chamado *mixin*, evitando a necessidade de múltiplas heranças de classes[2].

---

[1] O Rake começou a utilizar o módulo `Rake` como namespace à partir da versão 0.6.0.

[2] Se você quiser ir além, você nunca mais precisará usar herança de classes com mixins.

Imagine que além da classe `Array`, você também quer que o método `Range#sum` seja definido. Em vez de duplicar o código deste método, você pode simplesmente criar um módulo e *incluí-lo* nas classes `Array` e `Range`.

Primeiro, vamos extrair o método `Array#sum` para o módulo `Summable`.

```
module Summable
  def sum
    inject(:+)
  end
end
```

Depois, basta incluir este módulo em qualquer classe que irá ter este comportamento.

```
class Array; include Summable; end
class Range; include Summable; end
```

Toda vez que você inclui um módulo, ele é adicionado como uma *superclasse*.

```
Array.ancestors
#=> [Array, Summable, Enumerable, Object, Kernel, BasicObject]

Range.ancestors
#=> [Range, Summable, Enumerable, Object, Kernel, BasicObject]
```

Você pode verificar a lista de módulos que foram incluídos com o método `Module.included_modules`.

```
Array.included_modules
#=> [Summable, Enumerable, Kernel]
```

O método `Module#include` irá adicionar o método como uma superclasse de `self`. No entanto, às vezes pode ser útil adicionar métodos de instância de um objeto em particular; neste caso, você deverá utilizar o método `Object#extend`.

```ruby
module Permalink
  def to_permalink
    self.to_s.downcase.gsub(/[^a-z0-9-]/, "-")
  end
end

title = "Ruby Metaprogramming: I really like it"
title.extend Permalink
title.to_permalink
#=> "ruby-metaprogramming--i-really-like-it"
```

O método `Object#extend` está disponível em todos os objetos. Por isso, você pode fazer algo como

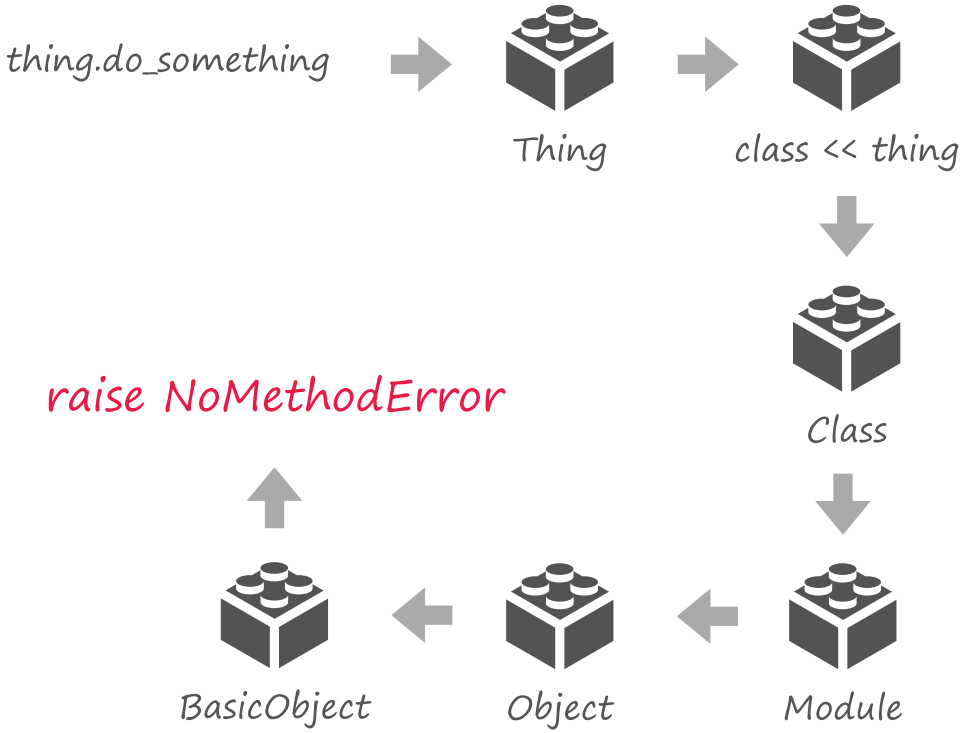
```ruby
module Person
  extend self

  def say
    "Hi there!"
  end
end
```

O módulo `Person` está estendendo ele mesmo! Isso irá adicionar os métodos de instância do módulo como métodos do próprio objeto (o módulo) `Person`, tornando estes métodos de classe.

```ruby
Person.say
#=> "Hi there!"
```

Métodos

Quando um método é invocado, Ruby irá procurá-lo no próprio objeto. Caso ele não seja encontrado, este método será procurado na classe Singleton. Se ele também não existir lá, então a busca será feita na hierarquia da classe, passando por cada uma das classes-pai deste objeto. Em último caso, a exceção `NoMethodError` será lançada. Este processo é chamado de *method lookup*.



O Ruby permite executar métodos de diversas maneiras. A mais comum, obviamente, é fazendo a chamada diretamente no receiver.

```ruby
class Person
  def say(message)
    puts message
  end
end

person = Person.new
person.say "Hello world!"
```

O único problema com esta abordagem é que você precisa saber previamente qual é o nome do método. Felizmente, o Ruby permite executar métodos dinamicamente, como você verá à seguir.

Executando métodos dinamicamente

Você também pode utilizar o método `Object#__send__`, que funciona exatamente como `Object#send`, mas que é o *método reservado*, já que o método `Object#send` pode ser sobrescrito. É sempre uma boa ideia utilizar o método reservado, para garantir que seu código funcione em qualquer situação.

Para executar métodos dinamicamente, devemos utilizar o método `Object#send`; este método irá enviar uma mensagem para o receiver, com os argumentos que o método irá receber. O exemplo anterior pode ser representado da seguinte forma:

```ruby
person = Person.new
person.send :say, "Hello world!"
```

O método `Object#send` irá receber o nome do método que vai ser executado (`Person#say`) e uma lista de argumentos. Métodos que esperam mais de um argumento podem ser executados como `receiver.send(:method_name, arg1, arg2, arg3)`.

Com o método `Object#send`, o nome do método torna-se apenas um argumento, permitindo que você *componha dinamicamente o nome do método* que vai ser executado. Esta técnica é conhecida como *Dynamic Dispatching* e é muito utilizada.

```
class Person
  def say(message)
    puts message
  end

  def say_hello
    say("Hey! Wassup?")
  end

  def say_goodbye
    say("Gotta go! Seeya!")
  end
end

person = Person.new

%w[hello goodbye].each do |name|
  person.send("say_#{name}")
end
```

Outra vantagem[1] do método `Object#send` é que você pode executar métodos privados sem lançar nenhum tipo de exceção. Se você não quiser executar métodos privados, pode utilizar o método `Object#public_send`, adicionado no Ruby 1.9.

Executar métodos dinamicamente pode ser uma mão na roda, mas você não está limitado apenas à execução. O Ruby também permite *definir* novos métodos dinamicamente, como você verá à seguir.

[1] Isto é uma vantagem ou não, dependendo do ponto de vista. Se um método é privado, significa que você não deveria executá-lo, para começo de conversa, a menos que tenha um motivo realmente forte para fazer isso.

Definindo novos métodos

No Ruby, é possível definir novos métodos em uma classe, aproveitando-se do conceito de *classes abertas*. Você pode definir um novo método na classe `Array`, por exemplo.

Embora este tipo de *monkey patching* seja extremamente útil, nem sempre é uma boa ideia alterar métodos existentes, modificando seu comportamento.

```
class Array
  def sum
    inject(:+)
  end
end
```

Para verificar que o método funciona, podemos escrever o seguinte teste:

```
require "test/unit"

class TestArray < Test::Unit::TestCase
  def test_sum
    assert_equal 6, [1,2,3].sum
  end
end
```

Mas nem sempre é possível utilizar esta técnica. Às vezes, precisamos definir dinamicamente novos métodos; isso pode ser feito com o método `Module#define_method`, que espera o nome do novo método e um bloco que será o corpo deste método.

```
class Person
  def say(message)
    puts message
  end

  define_method :scream! do |message|
```

```
    say(message.upcase)
  end
end

person = Person.new
person.scream! "Shut up!"
#=> SHUT UP!
```

O método `Module#define_method` irá criar um método de instância na classe `Person`. Você também pode criar métodos de classe, se executar o método `Module#define_method` na classe `Singleton` deste objeto.

```
class Person
  class << self
    define_method :description do
      puts "Hold a person information."
    end
  end
end

Person.description
#=> "Holds person's information."
```

Você também pode criar métodos executando código no contexto de objetos com os métodos `Kernel#eval`, `Module#class_eval` e `BasicObject#instance_eval`[2].

```
class Person
  class_eval do
    def say(message)
      puts message
    end
  end
end
```

[2] Você verá mais sobre estes métodos no capítulo sobre Execução de código.

```ruby
person = Person.new
person.say "Hi!"
```

Redefinindo métodos existentes

Embora definir novos métodos seja uma funcionalidade extremamente útil, às vezes queremos sobrescrever um método existente, estendendo seu comportamento. Uma das alternativas é criar um *alias* deste método, executando ou não o método original em nossa nova implementação.

```ruby
class String
  alias :original_downcase :downcase

  def downcase
    self.original_downcase.gsub(/(.)/m, '[\\1]')
  end
end
```

Embora esta técnica funcione, ela irá expor o método `String#original_downcase`, mesmo não fazendo parte da implementação. Além disso, algum desenvolvedor pode substituir o método `String#original_downcase`, o que pode causar resultados inesperados.

Uma alternativa é capturar o método `String#downcase` e vincular este método ao objeto, executando-o explicitamente com o método `Method#call`.

```ruby
class String
  downcase = instance_method(:downcase)

  define_method :downcase do
    downcase.bind(self).call.gsub(/(.)/m, '[\\1]')
  end
end
```

Também é possível redefinir métodos existentes de uma superclasse.

```ruby
class Person
  def initialize(options = {})
    options.each {|name, value| instance_variable_set "@#{name}", value }
  end
end
```

O método `Person#initialize` irá definir variáveis de instância baseado na combinação chave-valor do hash. Podemos definir uma nova classe chamada `Teacher`, que irá estender o comportamento do método `Person#initialize`.

```ruby
class Teacher < Person
  def initialize(options = {})
    options.merge!(:role => "Teacher")
    super
  end
end
```

A palavra-chave `super` irá executar o método de mesmo nome na superclasse desde objeto. Quando chamado sem argumentos e sem parenteses, a palavra-chave `super` irá utilizar os mesmos argumentos do método que a chamou; na prática, age como `super(*args)`. Se você quiser chamar `super` sem nenhum argumento, utilize `super()`.

Removendo métodos

Eventualmente, você vai querer remover algum método.

```ruby
class Person
  def say_hello
    puts "Hi there!"
  end
end

Person.new.respond_to?(:say_hello)
#=> true

Person.class_eval { remove_method :say_hello }
```

```ruby
Person.new.respond_to?(:say_hello)
#=> false
```

Note que o método `Module#remove_method` é privado; por isso é preciso executar tal método como vimos nos capítulos Métodos: Execução de código.

O método `Module#remove_method` irá remover o método apenas do objeto em que foi chamado. Caso exista algum método de mesmo nome em alguma superclasse, o método ainda será executado.

```ruby
class Parent
  def message
    "Hello from parent"
  end
end

class Child < Parent
  def message
    "Hello from child"
  end
end

Child.new.message
#=> "Hello from child"

Child.class_eval { remove_method :message }

Child.new.message
#=> "Hello from parent"
```

Para evitar que isso aconteça, você pode utilizar o método `Module#undef_method`. Este método previne qualquer chamada a um método, mesmo que ele esteja disponível através de alguma superclasse.

```ruby
Child.new.message
#=> "Hello from child"
```

```ruby
Child.class_eval { undef_method :message }

Child.new.message
#=> NoMethodError: undefined method ‘message’ for #<Child:0x0000010083ea90>
```

Capítulo 1

Execução de código

O Ruby permite que você execute strings e blocos de código em *runtime*. Para isso, ele disponibiliza diversos métodos como `Module#class_eval`, `Object#instance_eval` e `Kernel#eval`, que atuam em contextos diferentes.

Module#class_eval

Os métodos `Module#class_eval` e `Module#instance_eval` alteram o contexto de execução para `self`. Desse modo, é possível executar métodos privados sem a necessidade de ter que usar o método `Object#send`.

O método `Module#class_eval`, que na verdade é apenas um alias para `Module#module_eval`, permite executar código no contexto de uma classe ou módulo. Ele é muito utilizado para adicionar novos métodos ou incluir módulos em uma classe.

```ruby
class Person
end

Person.class_eval do
  def clap
    puts "*clap* *clap* *clap*"
  end
end

person = Person.new
person.clap
#=> *clap* *clap* *clap*
```

Este mesmo exemplo poderia ter sido definido sem o uso de `Module#class_eval`, mas isso deixaria o código muito menos legível.

```ruby
Person.send :define_method, :clap do
  puts "*clap* *clap* *clap*"
end
```

Você pode estar se perguntando porque o método foi adicionado como sendo de instância. A resposta é bastante simples: o método `Module#class_eval` irá executar o código utilizando a classe `Person` como seu contexto (`self`). Dessa maneira, se você quisesse adicionar um método de classe, poderia ter adicionado algo como

```ruby
Person.class_eval do
  def self.description
    puts "Hold a person information."
  end
end

Person.description
#=> Hold a person information.
```

Às vezes você perceberá que é mais fácil executar código como strings, em vez de passar um bloco, principalmente quando precisar acessar muitas variáveis de instância.

```ruby
Person.class_eval do
  %w[name age height].each do |name|
    define_method(name) do |*args|
      instance_variable_set("@#{name}", args.first) unless args.empty?
      instance_variable_get("@#{name}")
    end
  end
end
```

O código acima irá definir os métodos `Person#name`, `Person#age` e `Person#height`, que se receberem um único argumento, atuam como *setters*. Caso contrário, retornam o valor definido para aquele método. Para definir e retornar tais variáveis, estamos utilizando os métodos `Object#instance_variable_set` e `Object#instance_variable_get`. Se você precisa definir/ler muitas variáveis, a legibilidade pode ficar comprometida e você pode se perder em seu próprio código! Uma saída é reescrever o código, como no exemplo à seguir.

```ruby
class Person
  %w[name age height].each do |name|
    class_eval <<-RUBY, __FILE__, __LINE__
      def #{name}(*args)                          # def age(*args)
        @#{name} = args.first unless args.empty?  #   @age = args.first unless args.empty?
        @#{name}                                  #   @age
      end                                         # @end
    RUBY
  end
end
```

Esta abordagem é muito utilizada no Ruby on Rails. Normalmente, tal código vem acompanhado de comentários que mostram como cada linha será executada.

Note que estamos passandos os argumentos `__FILE__` e `__LINE__`. Isso permitirá que o método `Method#source_location` retorne o lugar onde o método foi definido. Esta informação também é utilizada pelas mensagens de erro.

Object#instance_eval

O método `Object#instance_eval` é similar ao método `Module#class_eval`, com a diferença de que também pode ser executado em instâncias de objetos. Ao contrário do método `Module#class_eval`, está presente em todos[1] os objetos.

```ruby
person = Person.new

Person.respond_to?(:class_eval)     #=> true
Person.respond_to?(:instance_eval)  #=> true

person.respond_to?(:class_eval)     #=> false
person.respond_to?(:instance_eval)  #=> true
```

[1] Você já deve estar cansado de saber, mas no Ruby classes também são objetos.

Além disso, `Object#instance_eval` irá alterar o contexto de execução para o próprio objeto (`self`), permitindo criar DSLs[2] muito facilmente.

Assim como o método `Module#class_eval`, você também pode definir novos métodos.

```
person = Person.new
person.instance_eval do
  def say(message)
    puts message
  end
end

person.respond_to?(:say)
#=> true

person.say "hi"
#=> hi
```

Como você está no escopo do objeto, é possível fazer tudo o que você faria normalmente no escopo de um objeto, inclusive **executar métodos privados**.

```
person.instance_eval { @name = "John Doe" }
person.instance_variable_get("@name")
#=> John Doe
```

No capítulo Criando DSLs você verá que o método `Object#instance_eval` tem um papel fundamental na criação de DSLs.

[2] Você verá como utilizar todos os conceitos de metaprogramação juntos no capítulo Construindo DSLs.

Kernel#eval

Ao contrário dos outros métodos, `Kernel#eval` permite executar apenas strings no contexto atual. Em contra-partida, você pode, opcionalmente, passar um contexto no qual esse código será executado.

```
eval "puts 1 + 1"
#=> 2
```

Embora o método `Kernel#eval` não pareça útil, existem casos onde executar código Ruby em tempo de execução pode ser uma boa ideia. É o caso de Rake.

O Rake permite que você defina suas *tasks* em um arquivo de build, normalmente chamado de *Rakefile*. Esse arquivo é lido do *filesystem* e executado em um contexto específico, com acesso aos métodos da biblioteca como `desc` e `task`.

Para entender melhor como este conceito funciona, vamos ler um arquivo que define as informações pessoais de um usuário. Tais informações servirão para preencher os atributos de um objeto, que pode executar tarefas específicas utilizando estes dados. Primeiro, crie um arquivo chamado `~/.aboutme` com o seguinte conteúdo:

```
name  "John Doe"
email "john@doe.com"
about "I'm just playing with Ruby code evaluation"
```

Nossa classe `Person` deverá implementar os métodos `Person#name`, `Person#email` e `Person#about`. Quando um argumento é passado, irá atuar como um *setter*. O código será o mesmo que usamos no exemplo mostrado em Object#instance_eval, mas com um pouco de açúcar sintético.

```
class Object
  def self.dsl_attr(name)
    class_eval <<-RUBY, __FILE__, __LINE__
      def #{name}(*args)                              # def about(*args)
        @#{name} = args.first unless args.empty?      #   @about = args.first unless args.empty?
        @#{name}                                      #   @about
      end                                             # @end
    RUBY
```

```
  end
end
```

Agora, em nossa classe `Person`, podemos definir nossos atributos.

```
class Person
  dsl_attr :name
  dsl_attr :email
  dsl_attr :about
end
```

Precisamos criar um método que irá ler o arquivo `~/.aboutme` e que irá executar o conteúdo do arquivo no contexto de uma instância da classe `Person`. Vamos chamar este método de `Person.from_manifest`.

```
class Person
  dsl_attr :name
  dsl_attr :email
  dsl_attr :about

  def self.from_manifest
    source = open(File.expand_path("~/.aboutme")).read
    person = new
    eval source, person.instance_eval { binding }
    person
  end
end
```

O código acima é muito simples. Primeiro, lemos o arquivo `~/.aboutme`. Depois, instanciamos um novo objeto da classe `Person`; note que utilizamos apenas `new`, pois o receiver padrão é `self`, que neste caso é a própria classe[3]. Por último, executamos o código lido no contexto da instância da classe `Person`, retornado pelo método `Kernel#binding`[4].

[3] Para saber mais sobre `self`, leia o capítulo Entendendo o self.

```
person = Person.from_manifest

person.name
#=> John Doe

person.email
#=> john@doe.com

person.about
#=> I'm just playing with Ruby code evaluation
```

O método `Kernel#eval` não é muito rápido; se você puder, evite fazer muitas chamadas a este método.

[4] `Kernel#binding` é um método privado. Por este motivo, utilizamos o método `Object#send` para executá-lo, como vimos no capítulo Métodos.

Callable Objects

O Ruby possui diversos objetos que são executáveis, como é o caso de métodos e procs. Estes objetos respondem ao método `call` e irão executar algum código associado ao objeto.

Procs e lambdas

Procs são objetos muito semelhantes a métodos: você pode executá-los e passar argumentos, com a diferença de que eles não precisam estar necessariamente associados a um *receiver*.

Procs podem ser executados de diversas maneiras diferentes.

```
output = Proc.new {|message| puts message}

output.call("hi!")
#=> "hi!"

output["hi!"]
#=> "hi!"

output.("hi!")    #=> Ruby 1.9+
#=> "hi!"
```

A palavra-chave `lambda` permite criar uma função anônima, que nada mais é que uma instância da classe `Proc`. Você precisa fornecer um bloco, que será usado como o corpo de sua função.

```
say = lambda { puts "hi!" }
say.call
#=> "hi!"
```

Assim como métodos, *lambdas* podem receber argumentos.

```
sum = lambda {|n1, n2| n1 + n2}
sum[1, 2]
#=> 3
```

No Ruby 1.9 você pode até definir valores-padrão para os argumentos!

```
say = lambda {|message = "you've got nothing to say"| puts message}

say["hi!"]
#=> "hi!"

say.call
#=> "you've got nothing to say"
```

Embora lambdas e procs sejam muito semelhantes, eles possuem duas diferenças muito importantes. A primeira é que lambdas podem forçar o retorno com a palavra-chave `return`[1]. No caso de procs, o uso do `return` irá forçar a saída do método em que o objeto estiver contido.

```
def returning
  lambda { return }.call
  puts "#1"
  Proc.new { return }.call
  puts "#2"
end

returning
```

No exemplo anterior, apenas a mensagem `#1` será exibida, já que `Proc.new { return }` irá parar a execução do método.

[1] Este comportamento foi introduzido no Ruby 1.9.

A outra diferença é que procs criados com `lambda` irão verificar a quantidade de argumentos que foram passados; se um lambda receber mais ou menos argumentos do que espera, a exceção `ArgumentError` será lançada.

```
output = proc {|message| p message}
output.call
#=> nil

output = lambda {|message| p message}
output.call
#=> ArgumentError: wrong number of arguments (0 for 1)
```

Blocos

No Ruby, **blocos não são objetos**. Eles apenas fazem parte da sintaxe de chamada de um método, por exemplo.

```
10.times {|i| puts i }
```

No exemplo acima, o receiver `10` está executando o método `times`, que recebe um bloco. Infelizmente, não é possível acessar diretamente o código `{|i| puts i}`. O que você precisa fazer é *capturar* tal bloco como um objeto da classe `Proc`.

O Ruby permite capturar blocos que são passados a um método utilizando um argumento com um `&` em sua lista de argumentos.

```
def say(&block)
  puts block.call
end

say { "hi!" }
#=> "hi!"
```

Toda vez que você utiliza esta sintaxe, o Ruby irá criar implicitamente uma instância da classe `Proc`, atribuindo o bloco ao argumento `block`. Este argumento pode ter qualquer nome; o que irá definir se ele irá armazenar o bloco passado ao método ou não é a presença do `&`.

Você não é obrigado a passar um bloco para um método, mesmo que ele tenha a definição de captura; neste caso, você precisará verificar se um bloco foi fornecido antes de utilizá-lo.

```ruby
def say(&block)
  if block_given?
    puts block.call
  else
    puts "you've got nothing to say!"
  end
end

say { "hi!" }
#=> "hi!"
```

Se você quiser, pode omitir a captura do bloco e, ainda assim, executá-lo. É para isto que serve a palavra-chave `yield`. Do mesmo modo, você precisará verificar se um bloco foi fornecido antes de utilizá-lo.

```ruby
def say
  if block_given?
    puts yield
  else
    puts "you've got nothing to say!"
  end
end

say { "hi!" }
#=> "hi!"
```

Mesmo sem definir a captura do bloco, é possível transformar o bloco fornecido em uma instância da classe `Proc`. Basta você criar uma instância sem passar nenhum bloco.

```ruby
def say
  if block_given?
    block = Proc.new
    puts block.call
  else
    puts "you've got nothing to say!"
  end

end

say { "hi!" }
#=> "hi!"
```

Toda vez que você cria uma instância da classe `Proc` sem passar um bloco, o bloco que foi passado ao método onde a chamada está sendo executada é que será utilizado.

Você também pode transformar procs em blocos. Basta utilizar o próprio `&` no último argumento que identificará o bloco.

```ruby
say = lambda { puts "hi!" }

def output(&block)
  yield
end

output(&say)
#=> "hi!"
```

Métodos

O Ruby é capaz de transformar métodos em objetos. Este objeto é uma instância da classe Method e também é um objeto executável, muito semelhante aos lambdas e procs.

```ruby
class Person
  def hi
    puts "hi!"
  end
end

person = Person.new
hi = person.method(:hi)

hi.class
#=> Method

hi.call
#=> "hi!"
```

Se o método envia algum argumento com a palavra-chave `yield`, você pode passar um bloco.

```ruby
def sum(n1, n2)
  yield n1 + n2
end

sum_method = method(:sum)
sum_method.call(1, 2) do |sum|
  puts sum
  #=> 3
end
```

Você também pode transformar métodos de instância em objetos. No entanto, estes métodos não estarão vinculados a nenhum objeto, mas só poderão ser executados depois que você associá-los a uma instância da classe que foi definido.

```ruby
class Person
  def hi
    puts "hi!"
  end
end
```

```ruby
hi = Person.instance_method(:hi)

hi.class
#=> UnboundMethod

hi.call
#=> NoMethodError: undefined method ‘call’ for #<UnboundMethod: Person#hi>

person = Person.new
hi.bind(person).call
#=> "hi!"
```

O classe `UnboundMethod` não é uma subclasse da classe `Method`. Somente após invocar o método `UnboundMethod#bind`, é que uma instância da classe `Method` é retornada. Neste momento, você poderá executar este método como um objeto executável, como vimos anteriormente.

Se você tentar associar este método a uma classe diferente, receberá a exceção `TypeError`, informando qual a classe que você deve utilizar.

```ruby
class Car; end

# hi.bind(Car.new).call
#=> TypeError: bind argument must be an instance of Person
```

Constantes

No Ruby, toda e qualquer referência que começa com uma letra maiúscula, incluindo nome de classes e módulos, é uma *constante*.

```
class Calculator
  PI = 3.141592653589793
end
```

Neste exemplo, `Calculator` e `Calculator::PI` são constantes. Perceba que constantes respeitam o seu namespace; é possível ter o mesmo nome de constante em diferentes namespaces.

```
module Company
  SOME_CONSTANT = 1

  class Person
    SOME_CONSTANT = 2
  end
end
```

Embora constantes sejam “constantes” na maioria das linguagens, o Ruby permite que sobrescreva uma constante existente.

```
module Company
  SOME_CONSTANT = 0
end

Company::SOME_CONSTANT = 1
```

Ao fazer isso, você receberá uma mensagem de alerta como esta: `warning: already initialized constant Company::SOME_CONSTANT`. No entanto, é possível desativar tal mensagem se você precisar sobrescrever uma constante; basta alterar a flag `$VERBOSE`.

```
old_verbose, $VERBOSE = $VERBOSE, nil
Company::SOME_CONSTANT = 2
$VERBOSE = old_verbose
```

O ActiveSupport implementa esta funcionalidade elegantemente através do método `Kernel#silence_warnings`.

```
silence_warnings { Company::SOME_CONSTANT = 3 }
```

Para retornar as constantes definidas em um namespace, você pode utilizar o método `Module#constants`.

```
Company.constants
#=> [:SOME_CONSTANT]
```

Todas as constantes de primeiro nível podem ser acessadas através do método `Object.constants`.

```
Object.constants
#=> [:Object, :Module, :Class, :Kernel, .., :Company]
```

Você também pode verificar se uma constante foi definida com o método `Module#const_defined?`.

```
Company.const_defined?("SOME_CONSTANT")
#=> true

Company.const_defined?("MISSING")
#=> false
```

Para pegar a referência de uma constante, você deve utilizar o método `Module#const_get`.

```
Company.const_get("SOME_CONSTANT")
#=> Company::SOME_CONSTANT
```

Se você tentar referenciar uma constante que não existe, irá lançar a exceção `NameError`.

```
Company.const_get("MISSING")
#=> NameError: uninitialized constant Company::MISSING
```

Para definir constantes dinamicamente, você deve utilizar o método `Module#const_set`.

```
Company.const_set("ANOTHER_CONSTANT", 2)
Company.const_defined?("ANOTHER_CONSTANT")
#=> true
```

Por último, também é possível remover constantes; é este o papel do método `Module#remove_const`.

```
Company.class_eval { remove_const("SOME_CONSTANT") }
#=> 1
```

Note que o método `Module#remove_const` é privado; por isso é preciso executar tal método como vimos nos capítulos Métodos e Execução de código.

Você pode remover qualquer constante, incluindo classes e módulos.

```
class Person
end

Object.class_eval { remove_const("Person") }
Object.const_defined?("Person")
#=> false
```

Capítulo 1

Hooks

O Ruby possui diversos *hooks* (ou *callbacks*) que permitem interceptar algumas ações do interpretador quando algum evento ocorre. Estes hooks são apenas métodos com nomes específicos que, quando presentes no contexto correto, são executados pelo Ruby. Existem diversos hooks para eventos diferentes, como você verá à seguir.

Módulos e Classes

included

É executado quando um módulo é incluído em outro[1]. Ele receberá o módulo que o incluiu como argumento.

Por padrão, métodos, constantes e variáveis deste módulo serão incluídas, a menos que este módulo já seja uma superclasse do objeto.

```
module SomeModule
  def self.included(base)
    puts "including #{name} on #{base.name}"
  end
end

class SomeClass
  include SomeModule
end
```

É semelhante ao hook `Module.append_features`.

[1] Isso também é válido para classes, já que `Module` é uma superclasse de `Class`.

const_missing

É executado quando uma constante não é encontrada. Ele receberá o nome da constante como argumento.

```ruby
class Object
  def self.const_missing(const)
    puts "The #{const} constant has not been found"
  end
end

User
#=> "The User constant has not been found"
```

extended

É executado quando um objeto é estendido com um módulo. Ele receberá o objeto como argumento.

```ruby
module SomeModule
  def self.extended(object)
    puts "Extending #{object.inspect}"
  end
end

some_object = Object.new
some_object.extend(SomeModule)
#=> Extending #<Object:0x00000100847d70>
```

Este hook é semelhante ao hook `Module.extend_object`.

inherited

É executado quando um classe é herdada por outra. Ele receberá a subclasse como argumento.

```ruby
class Parent
  @descendants = []
```

```ruby
  def self.inherited(child)
    @descendants << child.name unless @descendants.include?(child.name)
  end

  def self.descendants
    @descendants
  end
end

class Child < Parent
end

Parent.descendants
#=> ["Child"]
```

initialize_clone

É executado toda vez que um objeto invoca o método `Object#clone`. Ele receberá o objeto como argumento. Por padrão, este hook será delegado para `Object#initialize_copy`.

initialize_copy

É executado toda vez que os métodos `Object#clone` ou `Object#dup` forem invocados. Ele receberá o objeto como argumento.

```ruby
class SomeClass
  def initialize_clone(object)
    puts "You can't clone #{object.inspect}"
  end
end

SomeClass.new.clone
#=> "You can't clone #<SomeClass:0x0000010083e428>"
```

initialize_dup

É executado toda vez que um objeto invoca o método `Object#dup`. Ele receberá o objeto como argumento. Por padrão, este hook será delegado para `Object#initialize_copy`.

Métodos

method_added

É executado toda vez que um método for adicionado a um objeto. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.method_added(method)
    puts "A new method has been added: #{method}"
  end

  def do_something
  end
  #=> "A new method has been added: do_something"
end
```

method_missing

O `Object#method_missing` é, sem dúvida alguma, o hook mais utilizado do Ruby. Ele permite interceptar chamadas a métodos que não existem.

```ruby
class Finder
  def method_missing(method, *args)
    puts "Method #{method} not found"
  end
end

finder = Finder.new
```

```
finder.find_by_name
#=> "Method find_by_name not found"
```

Isso permite fazer coisas muito interessantes como construir chamadas dinâmicas para métodos, agir como um *delegator*, criar um proxy para constantes... enfim, as possibilidades são infinitas. Veja por exemplo, a biblioteca Builder. Ela permite criar XMLs de uma maneira muito elegante, sem que você precise chamar métodos específicos da biblioteca.

```
require "builder"

xml = Builder::XmlMarkup.new(:target => STDOUT, :indent => 2)
xml.instruct!
xml.person do |person|
  person.name "John Doe"
  person.phones do |phones|
    phones.phone "1234-5678", :type => "work"
    phones.phone "5678-1234", :type => "cellphone"
  end
end
```

Este código irá gerar um XML válido como este:

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <name>John Doe</name>
  <phones>
    <phone type="work">1234-5678</phone>
    <phone type="cellphone">5678-1234</phone>
  </phones>
</person>
```

O `Object#method_missing` é ótimo quando você quer agir como proxy para um mapeamento em hash, por exemplo.

```
class Person
  MAPPING = {
```

```
    :name    => "John Doe",
    :age     => 32,
    :address => "455 Larkspur Dr., California Springs, CA 92926",
    :phone   => "1234-5678"
  }

  def method_missing(method, *args)
    MAPPING[method]
  end
end

john = Person.new

john.name
#=> "John Doe"

john.age
#=> 32
```

Mas o que acontece se acessarmos uma propriedade que não existe?

```
john.email
#=> nil
```

Aparentemente, nós substituímos a funcionalidade original do `Object#method_missing`. Este problema é simples de resolver; basta verificarmos se a chave existe e em caso negativo, executar o método original.

```
class Person
  MAPPING = {
    :name    => "John Doe",
    :age     => 32,
    :address => "455 Larkspur Dr., California Springs, CA 92926",
    :phone   => "1234-5678"
  }

  def method_missing(method, *args)
```

```
    return MAPPING[method] if MAPPING.key?(method)
    super
  end
end
```

Agora, ao executar o método `Person#email`, teremos algo como

```
john.email
#=> NoMethodError: undefined method ‘email’ for #<Person:0x0000010101e0d0>
```

Sempre que você sobrescrever o método `Object#method_missing`, lembre-se também de sobrescrever o método `Object#respond_to?`. Isso porque em vez de verificar se o objeto é uma instância da classe (com os métodos `Object#is_a?` e `Object#kind_of?`), você deve *preferencialmente* verificar se o objeto responde ao método que você precisa [2].

```
class Person
  MAPPING = {
    :name    => "John Doe",
    :age     => 32,
    :address => "455 Larkspur Dr., California Springs, CA 92926",
    :phone   => "1234-5678"
  }

  def method_missing(method, *args)
    return MAPPING[method] if MAPPING.key?(method)
    super
  end

  def respond_to?(method, include_private = false)
    return true if MAPPING.key?(method.to_sym)
    super
  end
```

[2] Este é, basicamente, o conceito de Duck Typing: *Se um objeto se parece com um pato, anda como um pato e “fala” como um pato, então deve ser um pato.*

```ruby
end

john = Person.new
john.respond_to?(:name)
#=> true

john.respond_to?(:email)
#=> false
```

method_removed

É executado toda vez que um método for removido com o método `Module#remove_method`. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.method_removed(method)
    puts "The method #{method} has been removed"
  end

  def do_something; end
end

SomeClass.class_eval { remove_method :do_something }
#=> The method do_something has been removed
```

method_undefined

É executado toda vez que um método for removido com o método `Module#undef_method`. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.method_undefined(method)
    puts "The method #{method} has been undefined"
  end

  def do_something; end
```

```ruby
end

SomeClass.class_eval { undef_method :do_something }
#=> The method do_something has been undefined
```

singleton_method_added

É executado toda vez que um método singleton for adicionado a um objeto. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.singleton_method_added(method)
    puts "The singleton method #{method} has been added to #{self.inspect}"
  end
  #=> The singleton method singleton_method_added has been added

  def singleton_method_added(method)
    puts "The singleton method #{method} has been added to the instance"
  end
end

object = SomeClass.new
def object.do_something; end
#=> "The singleton method do_something has been added to the instance"
```

singleton_method_removed

É executado toda vez que um método singleton for removido com o método `Module#remove_method`. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.singleton_method_removed(method)
    puts "The singleton method #{method} has been removed"
  end

  def self.do_something; end
```

```ruby
end

SomeClass.singleton_class.class_eval { remove_method :do_something }
#=> The singleton method do_something has been removed
```

singleton_method_undefined

É executado toda vez que um método singleton for removido com o método `Module#undef_method`. Ele receberá o nome do método como argumento.

```ruby
class SomeClass
  def self.singleton_method_undefined(method)
    puts "The singleton method #{method} has been undefined"
  end

  def self.do_something; end
end

SomeClass.singleton_class.class_eval { undef_method :do_something }
#=> The singleton method do_something has been undefined
```

Capítulo 1

Aprenda com exemplos

Incorporar a metaprogramação em seu dia-a-dia não é uma tarefa das mais fáceis. Você provavelmente utilizará diversas técnicas vistas até agora; muitas vezes, você precisará combinar mais do que uma destas técnicas e é aí que a coisa pode complicar.

Para ajudar você, este capítulo mostrará diversos *patterns* utilizados por diversos desenvolvedores, com uma explicação sobre tal código.

Registrando classes

Muitas bibliotecas podem ser estendidas através de novos *adapters*. Mas como saber quais são os adapters disponíveis? Uma das saídas é *registrar* novas classes.

Neste exemplo, nós iremos escrever uma biblioteca chamada `Formatter`. Seu objetivo será converter textos escritos em diversas formatações como Textile e Markdown para HTML.

O primeiro passo é escrever a classe que será a base para todos os formatadores. Ela é muitos simples: terá um método `Formatter::Base#initialize` que irá atribuir o conteúdo a ser convertido em uma variável de instância (assim poderemos acessá-la posteriormente) e um outro método `Formatter::Base#to_html` que deverá ser implementado por cada um dos formatadores.

```
module Formatter
  class AbstractMethodError < StandardError; end

  class Base
    attr_accessor :content

    def initialize(content)
```

```
      @content = content
    end

    def to_html
      raise Formatter::AbstractMethodError
    end
  end
end
```

O próximo passo é criar um método que irá converter o texto em HTML. Este método receberá um parâmetro indicando qual é o tipo de markup do texto (Textile ou Markdown) e qual o conteúdo a ser convertido. Sua assinatura será algo como `Formatter.format(type, content)`.

Precisaremos utilizar o argumento `type` para descobrir qual a classe que vamos usar. Uma alternativa é utilizar um hash que irá mapear este argumento com a classe responsável.

```
module Formatter
  class UnknownFormatterError < StandardError; end

  FORMATTERS = {
    :textile  => "Formatter::Textile",
    :markdown => "Formatter::Markdown"
  }

  def self.format(type, content)
    formatter_name = FORMATTERS[type.to_sym]
    raise UnknownFormatterError unless formatter_name

    formatter = eval(formatter_name)
    formatter.new(content).to_html
  end
end
```

Note que não estamos utilizando a classe em si; isso é feito para evitar que o Ruby mantenha uma referência de uma classe em memória, mesmo sem utilizá-la. Em vez disso nós estamos utilizando apenas o nome da constante; para pegar a referência da classe iremos utilizar o método `Kernel#eval`.

Agora, vamos escrever os formatadores.

```
require "RedCloth"
require "rdiscount"

module Formatter
  class Textile < Base
    def to_html
      RedCloth.new(content).to_html
    end
  end

  class Markdown < Base
    def to_html
      RDiscount.new(content).to_html
    end
  end
end
```

Todos os formatadores precisam apenas implementar o método `Formatter::Base#to_html`. Agora podemos converter textos para HTML utilizando uma interface única.

```
Formatter.format(:textile, "h1. Ruby Metaprogramming")
#=> "<h1>Ruby Metaprogramming</h1>"

Formatter.format(:markdown, "# Ruby Metaprogramming")
#=> "<h1>Ruby Metaprogramming</h1>\n"
```

Mas o que acontece se quisermos adicionar um novo formatador com o RDoc? Precisaremos *registrar* este formatador manualmente, adicionando uma nova entrada no hash.

```
require "rdoc/markup"
require "rdoc/markup/to_html"

module Formatter
  class Rdoc < Base
```

```
    def to_html
      RDoc::Markup::ToHtml.new.convert(content)
    end
  end
end

Formatter::FORMATTERS[:rdoc] = "Rdoc"

Formatter.format(:rdoc, "= Ruby Metaprogramming")
#=> "<h1>Ruby Metaprogramming</h1>\n"
```

A princípio, isso pode não parecer um problema, mas a verdade é que quanto mais coisas tivermos que configurar, maiores são as chances de algo sair errado. Podemos utilizar o método `Module.inherited` para registrar automaticamente os formatadores.

Primeiro, remova cada um dos ítens do hash `Formatter::FORMATTERS`; eles serão adicionados automaticamente.

```
module Formatter
  FORMATTERS = {}
end
```

Depois, vamos adicionar o método `Formatter::Base.inherited`.

```
module Formatter
  class Base
    def self.inherited(child)
      type = child.name.split("::").last.downcase.to_sym
      Formatter::FORMATTERS[type] = child.name
    end
  end
end
```

Com esta alteração não precisaremos mais registar as classes manualmente, evitando um ponto de falha.

method_missing e definição de métodos

Nossa biblioteca `Formatter` possui um método para converter textos para HTML. No entanto, seria interessante se toda vez que adicionassemos um novo formatador, um método com seu nome também fosse adicionado. Assim, poderíamos converter textos como em `Formatter.textile("h1. Ruby Metaprogramming")`.

Existem dois caminhos possíveis para esta implementação. O primeiro é utilizando o método `Object#method_missing`. Toda vez que um método não for encontrado, não verificamos se existe uma chave na constante `Formatters::FORMATTERS` com o mesmo nome.

```
module Formatter
  def self.method_missing(method, *args)
    return format(method.to_sym, args.first) if FORMATTERS.key?(method)
    super
  end

  def self.respond_to?(method, include_private = false)
    return true if FORMATTERS.key?(method)
    super
  end
end

Formatter.textile("h1. Ruby Metaprogramming")
#=> "<h1>Ruby Metaprogramming</h1>"

Formatter.respond_to?(:textile)
#=> true
```

Note que estamos implementando os métodos `Formatter.method_missing` e `Formatter.respond_to?` como vimos em Hooks: method_missing.

O outro caminho para implementarmos tal funcionalidade é criar os métodos toda vez que um novo formatador herdar de `Formatter::Base`. Altere o método `Formatter::Base.inherited` de modo que ele crie um método com o mesmo nome do formatador. Para isso, utilize o método `Module#class_eval`.

```ruby
module Formatter
  class Base
    def self.inherited(child)
      type = child.name.split("::").last.downcase.to_sym
      Formatter::FORMATTERS[type] = child.name

      Formatter.class_eval <<-RUBY
        def self.#{type}(content)           # def self.textile(content)
          format :#{type}, content          #   format :textile, content
        end                                 # end
      RUBY
    end
  end
end
```

Normalmente, definir o método dinamicamente é mais rápido que utilizar o método `Object#method_missing`. Na dúvida, faça um benchmark[1] para saber qual é o seu caso.

Definindo macros

No Ruby, macros são métodos de classe que alteram/estendem objetos com novos comportamentos. Você já deve conhecer, por exemplo, os métodos `Module#attr_accessor`, `Module#attr_reader` e `Module#attr_writer`. Quem utiliza o Ruby on Rails, com certeza já viu a macro `ActiveRecord::Base.has_many`.

Vamos criar uma macro que permitirá normalizar atributos antes de atribuir tal valor à sua variável. Como queremos que nossa macro esteja disponível para todas as classes, vamos adicioná-la diretamente à classe `Object`.

```ruby
class Object
  def self.attr_normalize(name, options = {})
    attr_reader name
```

[1] Embora muitas pessoas digam que benchmarks são discutíveis, eles podem nos ajudar a identificar pontos de lentidão no código.

```ruby
    class_eval <<-RUBY
      def #{name}=(value)                                   # def name=(value)
        @#{name} = send(:#{options[:with]}, value)          #   @name = send(:normalize_whitespaces, value)
      end                                                   # end
    RUBY
  end
end

class MyClass
  attr_normalize :name, :with => :normalize_whitespaces

  private
  def normalize_whitespaces(value)
    value.to_s.gsub(/\s+/, " ").gsub(/^ ?(.*?) ?$/, '\1')
  end
end

object = MyClass.new

object.respond_to?(:name)
#=> true

object.respond_to?(:name=)
#=> true

object.name = "\n\nsome     awkward     spaces\n\n\t"
object.name
#=> "some awkward spaces"
```

O Ruby executa a macro `Object.attr_normalize` na classe `MyClass` assim que ela é definida. Como ela não passa de um método é necessário que ela exista previamente, como mostrado em Métodos. Por isso nós definimos tal macro na classe `Object`[2], que é uma superclasse de quase todos os objetos.

[2] A exceção é a classe `BasicObject`, que é a superclasse de `Object`.

No caso do ActiveRecord, que usa macros ostensivamente, você precisa criar uma macro na classe `ActiveRecord::Base`, evitando poluir todo o namespace com funcionalidades específicas do ActiveRecord. Por exemplo, vamos criar uma macro que definirá um atributo de permalink.

Vamos criar um módulo `Permalink` que estenderá a classe `ActiveRecord::Base`. Este módulo terá um método `Permalink.extended` que irá disponibilizar a macro `Permalink.permalink` para todas as classes do ActiveRecord. Além disso, ele irá adicionar algumas validações do atributo `ActiveRecord::Base#permalink` e um callback que irá gerar o valor normalizado do permalink.

```
module Permalink
  def permalink(attribute)
    include Permalink::InstanceMethods

    class << self
      attr_accessor :permalink_attribute
    end

    self.permalink_attribute = attribute

    validates_presence_of :permalink
    before_validation :generate_permalink
  end
end
```

Agora precisamos escrever o módulo `Permalink::InstanceMethods`. Este módulo é responsável por definir todos os métodos que serão utilizados pela instância do ActiveRecord na hora de gerar o permalink.

```
module Permalink
  module InstanceMethods
    def to_param
      "#{id}-#{permalink}"
    end

    private
    def generate_permalink
      write_attribute :permalink, value_for_permalink
```

```
    end

    def value_for_permalink
      __send__(self.class.permalink_attribute).to_s.downcase.gsub(/[^a-z0-9]/, "-")
    end
  end
end
```

O método `Permalink::InstanceMethods#generate_permalink` apenas define o valor do atributo. Quem faz a normalização é o método `Permalink::InstanceMethods#value_for_permalink`. Perceba que estamos utilizando o método `Object#__send__` para pegar o valor do atributo que será normalizado.

Já o método `Permalink::InstanceMethods#to_param` é utilizado na geração de rotas. Ele é chamado quando você passa a instância para um dos helpers de url, como em `article_path(article)`.

Agora só falta estendermos a classe `ActiveRecord::Base`.

```
ActiveRecord::Base.extend(Permalink)
```

Ao estender a classe `ActiveRecord::Base`, nós tornaremos o método `Permalink.permalink` disponível em todas as classes de ActiveRecord. Para utilizar esta macro, basta chamá-lo em nossa classe.

```
class Post < ActiveRecord::Base
  permalink :title
end

post = Post.create(:title => "Ruby Metaprogramming is a nice technique")
post.permalink
#=> "ruby-metaprogramming-is-a-nice-technique"
```

Criando DSLs

DSLs (Domain-Specific Languages) tem uma relação muito próxima com a metaprogramação no Ruby. Para criar uma DSL você irá utilizar diversas técnicas vistas até agora e, muitas vezes, ao mesmo tempo.

Criar DSLs exige muito mais que saber quais são as técnicas corretas. DSLs exigem um conhecimento mais aprofundado sobre o domínio do problema. E isso nem sempre é fácil. Além disso, manter uma DSL pode ser trabalhoso. Antes de partir para este caminho, pese estes fatores com cuidado.

yield

Uma maneira de criar uma DSL é recebendo um bloco que receberá algum argumento. Este é o caminho escolhido pelo RSpec para sua configuração.

```
RSpec.configure do |config|
  config.color_enabled = true
end
```

Agora, nós vamos criar nosso próprio módulo de configuração. Primeiro defina os atributos configuráveis.

```
module MyLib
  module Config
    class << self
      attr_accessor :name
      attr_accessor :description
    end
  end
end
```

Do modo como nosso módulo foi definido, precisaremos especificar o receiver toda hora que formos definir uma nova configuração.

```ruby
MyLib::Config.name = "My Awesome Lib"
MyLib::Config.description = "My Awesome Lib does amazing stuff"
```

Para facilitar esta configuração, nós podemos criar um método que irá passar o módulo `MyLib::Config` para o bloco.

```ruby
module MyLib
  def self.configure(&block)
    yield Config
  end
end

MyLib.configure do |config|
  config.name = "My Awesome Lib"
  config.description = "My Awesome Lib does amazing stuff"
end
```

Embora esta técnica tenha diminuído o ruído do código, ela nem sempre é útil, principalmente quando todas as chamadas que você fizer forem em um receiver específico. Neste caso, uma alternativa melhor é utilizar o método `Object#instance_eval`.

instance_eval

Como vimos em Execução de Código: Object#instance_eval, o método `Object#instance_eval` permite alterar o contexto de execução de um bloco. Ele é, sem dúvida, um dos mais importantes métodos na criação de DSLs internas no Ruby.

Vamos criar uma classe que permitirá escrever receitas culinárias. Embora esse exemplo seja muito utilizado, ele perfeito pois aplica um domínio muito fácil de ser entendido.

Primeiro vamos criar nossa classe `Recipe`. Ela possui alguns atributos necessários para definir uma nova receita.

```ruby
class Recipe
  attr_accessor :name
  attr_accessor :description
  attr_accessor :ingredients
```

```
  attr_accessor :instructions

  def initialize
    @ingredients = []
    @instructions = []
  end
end
```

Normalmente, você iria criar uma instância e definir cada uma das propriedades.

```
recipe = Recipe.new
recipe.name = "Caipirinha"
recipe.ingredients << "12 limões"
recipe.ingredients << "1 litro de cachaça"
recipe.ingredients << "açúcar à gosto"
recipe.ingredients << "gelo à gosto"
```

Embora não seja uma tarefa das mais complicadas, seria melhor se pudessemos escrever isso de uma forma mais elegante. Vamos, então, alterar nosso método `Recipe#initialize` de modo que ele execute um bloco, quando disponível.

```
class Recipe
  def initialize(&block)
    @ingredients = []
    @instructions = []

    instance_eval(&block) if block_given?
    self
  end
end
```

Agora, podemos definir os atributos de uma outra maneira.

```
recipe = Recipe.new do
  self.name = "Capirinha"
```

```
    self.ingredients << "12 limões"
    self.ingredients << "1 litro de cachaça"
    self.ingredients << "açúcar à gosto"
    self.ingredients << "gelo à gosto"
  end
```

Você não precisa definir o receiver quando for utilizar um *getter*, mas ele é necessário se estiver definindo um valor através de um *setter*. Neste caso, nossa DSL não ficou muito diferente da versão inicial. Para evitar que isso seja necessário, nós iremos utilizar o método `Object#dsl_attr` que criamos em Execução de Código: Kernel#eval. Altere a definição do atributos.

```
  class Recipe
    dsl_attr :name
    dsl_attr :description
    dsl_attr :ingredients
    dsl_attr :instructions
  end
```

Agora podemos utilizar uma sintaxe um pouco mais agradável.

```
    recipe = Recipe.new do
      name "Capirinha"
      description "A caipirinha é uma das bebidas brasileiras mais conhecidas internacionalmente."

      ingredients << "12 limões"
      ingredients << "1 litro de cachaça"
      ingredients << "açúcar à gosto"
      ingredients << "gelo à gosto"

      instructions << "Após lavar o limão, descasca-se parcialmente o mesmo."
      instructions << %Q[Corta-se o limão em duas metades, removendo de cada
uma delas a parte central (branca) do bagaço.]
      instructions << %Q[Após os procedimentos usuais usam-se dois copos iguais,
bem colados entre si para efetuar a mistura, como se fosse uma coqueteleira.]
    end
```

Para finalizar, vamos implementar um método que irá formatar as informações da receita de um modo mais legível.

```ruby
class Recipe
  def to_text
    "".tap do |text|
      text << name
      text << "\n" << ("=" * name.size) << "\n"

      text << "\n" << description if description
      text << "\n"

      text << "\nIngredientes" << "\n------------\n\n"
      text << ingredients.collect {|i| "- #{i}\n"}.join("")

      text << "\nComo preparar" << "\n-------------\n\n"
      text << instructions.each_with_index.collect {|i, n| "#{n + 1}. #{i}\n"}.join("")
    end
  end
end
```

Se você executar o método `Recipe#to_text` terá uma saída como

```
Capirinha
=========

A caipirinha é uma das bebidas brasileiras mais conhecidas internacionalmente.

Ingredientes
------------

- 12 limões
- 1 litro de cachaça
- açúcar à gosto
- gelo à gosto

Como preparar
-------------
```

```
1. Após lavar o limão, descasca-se parcialmente o mesmo.
2. Corta-se o limão em duas metades, removendo de cada
uma delas a parte central (branca) do bagaço.
3. Após os procedimentos usuais usam-se dois copos iguais,
bem colados entre si para efetuar a mistura, como se fosse uma coqueteleira.
```

Interfaces Fluentes

Interfaces fluentes permitem executar diversos métodos em cadeia, normalmente formando uma sentença que tornará a leitura do código mais agradável. Um exemplo de interface fluente é o ARel, que simplifica a geração de SQL no ActiveRecord.

```ruby
User.where(:email => "john@doe.com").includes(:profile).limit(1).first
```

Para criar sua própria interface fluente, é preciso que você sempre retorne o receiver após definir o atributo.

```ruby
class SQL < BasicObject
  def select(fields)
    @select = fields
    self
  end

  def from(tables)
    @from = tables
    self
  end

  def where(conditions)
    @where = conditions
    self
  end

  def to_sql
    "".tap do |query|
      query << "SELECT #{@select} FROM #{@from}"
```

```
      query << " WHERE #{build_conditions}" if @where
    end
  end

  private
  def build_conditions
    @where.collect {|name, value| %[#{name} = "#{value}"]}.join(" AND ")
  end
end
```

Obviamente nossa implementação é falha, já que não implementa todos os comandos SQL, nem sanitiza os dados informados pelo usuário, mas você pegou a ideia.

```
sql = SQL.new
sql.select("*").from("users").where(:email => "john@doe.com")
sql.to_sql
#=> SELECT * FROM users WHERE email = "john@doe.com"

sql = SQL.new
sql.select("*").from("users")
sql.to_sql
#=> SELECT * FROM users
```

Perceba que a implentação dos métodos é basicamente a mesma: atribua um valor à variável de instância e passe o próprio objeto como retorno do método. Nós podemos evitar duplicações gerando estes métodos dinamicamente.

```
class SQL
  %w[select from where includes limit joins offset].each do |name|
    class_eval <<-RUBY
      def #{name}(value)
        @#{name} = value
        self
      end
    RUBY
  end
end
```

E para finalizar…

A melhor maneira de se usar a metaprogramação é saber quando não usá-la. Ao longo deste e-book você deve ter percebido que esta é uma técnica extremamente poderosa. No entanto, o uso desnecessário da metaprogramação pode tornar seu código mais difícil de entender, ou o que é pior, manter. Principalmente se quem tiver que manter o código não souber esta técnica.

É como todos nós já sabemos: com grandes poderes, vem grandes responsabilidades.