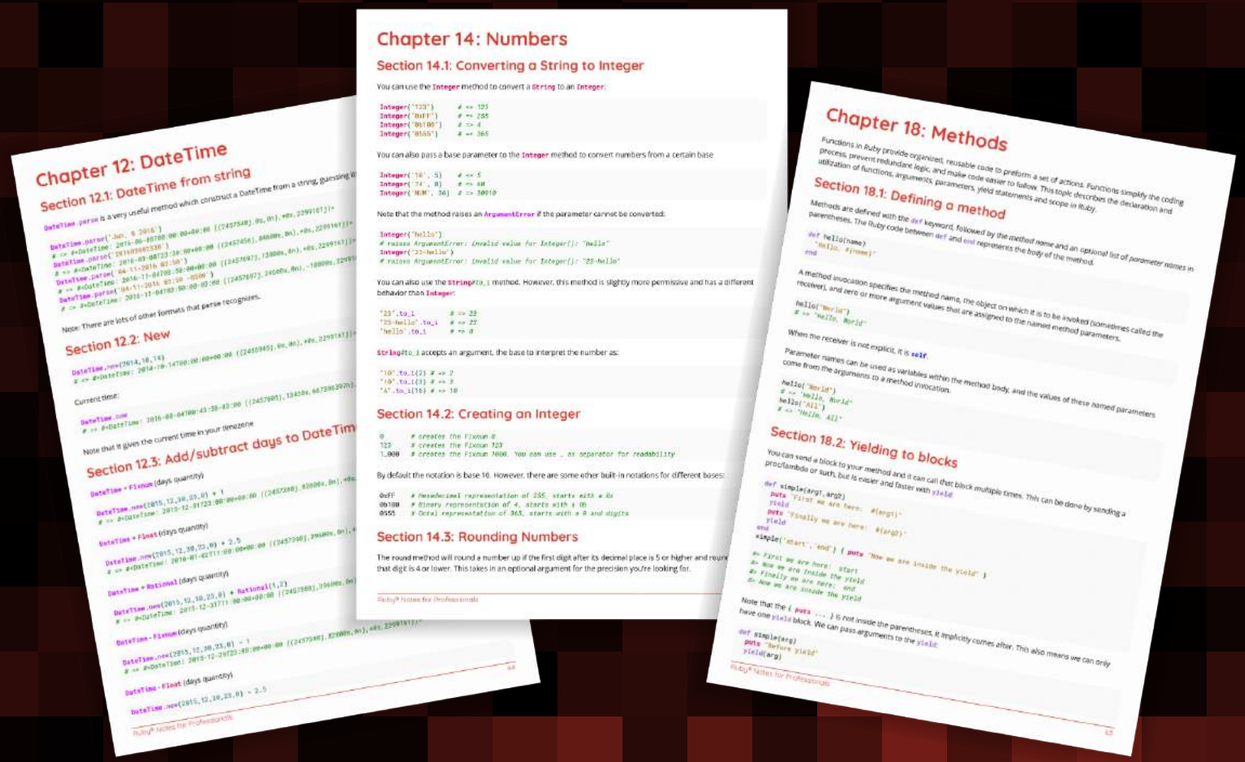
Ruby®

专业人士笔记



200多页

专业提示和技巧

GoalKicker.com

免费编程书籍

Ruby®

Notes for Professionals

200+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

目录

Contents

关于

请随意免费分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/RubyBook

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/RubyBook

Please send feedback and corrections to web@petercv.com

第1章：开始使用Ruby语言

版本	发布日期
2.5.1	2018-03-28
2.4	2016-12-25
2.3	2015-12-25
2.2	2014-12-25
2.1	2013-12-25
2.0	2013-02-24
1.9	2007-12-25
1.8	2003-08-04
1.6.8	2002-12-24

第1.1节：你好，世界

> 本示例假设已安装Ruby。

将以下内容放入名为hello.rb的文件中：

```
puts 'Hello World'
```

在命令行中，输入以下命令以执行源文件中的Ruby代码：

```
$ ruby hello.rb
```

这将输出：

```
Hello World
```

输出将立即显示在控制台上。Ruby源文件无需编译即可
执行。Ruby解释器在运行时编译并执行Ruby文件。

第1.2节：作为自执行文件的Hello World——使用Shebang（仅限类Unix操作系统）

你可以在脚本中添加解释器指令（shebang）。创建一个名为hello_world.rb的文件，内容如下：

```
#!/usr/bin/env ruby

puts 'Hello World!'
```

赋予脚本可执行权限。在Unix中操作方法如下：

```
$ chmod u+x hello_world.rb
```

现在你无需显式调用Ruby解释器即可运行脚本。

```
$ ./hello_world.rb
```


Chapter 1: Getting started with Ruby Language

Version	Release Date
2.5.1	2018-03-28
2.4	2016-12-25
2.3	2015-12-25
2.2	2014-12-25
2.1	2013-12-25
2.0	2013-02-24
1.9	2007-12-25
1.8	2003-08-04
1.6.8	2002-12-24

Section 1.1: Hello World

> This example assumes Ruby is installed.

Place the following in a file named `hello.rb`:

```
puts 'Hello World'
```

From the command line, type the following command to execute the Ruby code from the source file:

```
$ ruby hello.rb
```

This should output:

```
Hello World
```

The output will be immediately displayed to the console. Ruby source files don't need to be compiled before being executed. The Ruby interpreter compiles and executes the Ruby file at runtime.

Section 1.2: Hello World as a Self-Executable File—using Shebang (Unix-like operating systems only)

You can add an interpreter directive (shebang) to your script. Create a file called `hello_world.rb` which contains:

```
#!/usr/bin/env ruby

puts 'Hello World!'
```

Give the script executable permissions. Here's how to do that in Unix:

```
$ chmod u+x hello_world.rb
```

Now you do not need to call the Ruby interpreter explicitly to run your script.

```
$ ./hello_world.rb
```

第1.3节：从IRB运行Hello World

或者，你可以使用交互式Ruby Shell（IRB）立即执行之前在Ruby文件中编写的Ruby语句。

通过输入以下内容开始IRB会话：

```
$ irb
```

然后输入以下命令：

```
puts "Hello World"
```

这将产生以下控制台输出（包括换行）：

```
Hello World
```

如果你不想换行，可以使用print：

```
print "Hello World"
```

第1.4节：无源文件的Hello World

在安装Ruby后，在shell中运行以下命令。这展示了如何在不创建Ruby文件的情况下执行简单的Ruby程序：

```
ruby -e 'puts "Hello World"'
```

你也可以将Ruby程序输入到解释器的标准输入中。实现这一点的一种方法是在shell命令中使用here document：

```
ruby <<END
puts "Hello World"
END
```

第1.5节：使用tk的Hello World

Tk是Ruby的标准图形用户界面（GUI）。它为Ruby程序提供了跨平台的GUI。

示例代码：

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

结果：

Section 1.3: Hello World from IRB

Alternatively, you can use the Interactive Ruby Shell (IRB) to immediately execute the Ruby statements you previously wrote in the Ruby file.

Start an IRB session by typing:

```
$ irb
```

Then enter the following command:

```
puts "Hello World"
```

This results in the following console output (including newline):

```
Hello World
```

If you don't want to start a new line, you can use `print`:

```
print "Hello World"
```

Section 1.4: Hello World without source files

Run the command below in a shell after installing Ruby. This shows how you can execute simple Ruby programs without creating a Ruby file:

```
ruby -e 'puts "Hello World"'
```

You can also feed a Ruby program to the interpreter's standard input. One way to do that is to use a here document in your shell command:

```
ruby <<END
puts "Hello World"
END
```

Section 1.5: Hello World with tk

Tk is the standard graphical user interface (GUI) for Ruby. It provides a cross-platform GUI for Ruby programs.

Example code:

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

The result:

逐步说明：

```
require "tk"
```

加载tk包。

```
TkRoot.new{ title "Hello World!" }
```

定义一个标题为Hello World的小部件

```
Tk.mainloop
```

启动主循环并显示控件。

第1.6节：我的第一个方法

概述

创建一个名为`my_first_method.rb`的新文件

将以下代码放入文件中：

```
def hello_world
  puts "Hello world!"
end

hello_world() # 或者直接写 'hello_world'（不带括号）
```

现在，在命令行中执行以下命令：

```
ruby my_first_method.rb
```

输出应为：

```
你好，世界！
```

解释

- def 是一个关键字，表示我们正在def定义一个方法——在本例中，hello_world 是我们的方法名称。
- puts "Hello world!" puts（或输出到控制台）字符串 Hello world!
- end 是一个关键字，表示我们结束对 hello_world 方法的定义

Step by Step explanation:

```
require "tk"
```

Load the tk package.

```
TkRoot.new{ title "Hello World!" }
```

Define a widget with the title `Hello World`

```
Tk.mainloop
```

Start the main loop and display the widget.

Section 1.6: My First Method

Overview

Create a new file named `my_first_method.rb`

Place the following code inside the file:

```
def hello_world
  puts "Hello world!"
end

hello_world() # or just 'hello_world' (without parenthesis)
```

Now, from a command line, execute the following:

```
ruby my_first_method.rb
```

The output should be:

```
Hello world!
```

Explanation

- `def` is a keyword that tells us that we're `def`-ining a method - in this case, `hello_world` is the name of our method.
- `puts "Hello world!"` `puts` (or pipes to the console) the string `Hello world!`
- `end` is a keyword that signifies we're ending our definition of the `hello_world` method

- 由于 `hello_world` 方法不接受任何参数，调用该

方法时可以省略括号

- as the `hello_world` method doesn't accept any arguments, you can omit the parenthesis by invoking the method

第二章：类型转换（Casting）

第2.1节：转换为浮点数

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

但是，当字符串不是有效的Float时，会有所不同：

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

第2.2节：转换为字符串

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

通常，String() 只是调用 #to_s。

方法 Kernel#sprintf 和 String#% 的行为类似于C语言：

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

第2.3节：转换为整数

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

字符串在开头会取任意整数值，但不会从其他位置取整数：

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

但是，当字符串不是有效的整数时，会有区别：

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

第2.4节：浮点数和整数

```
1/2 #=> 0
```

由于我们除的是两个整数，结果也是整数。为了解决这个问题，我们需要将其中至少一个转换为浮点数：

```
1.0 / 2       #=> 0.5
1.to_f / 2    #=> 0.5
1 / Float(2) #=> 0.5
```

或者，fdiv 可用于返回除法的浮点结果，而无需显式转换任一操作数

Chapter 2: Casting (type conversion)

Section 2.1: Casting to a Float

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

However, there is a difference when the string is not a valid `Float`:

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

Section 2.2: Casting to a String

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

Usually, `String()` will just call `#to_s`.

Methods `Kernel#sprintf` and `String#%` behave similar to C:

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

Section 2.3: Casting to an Integer

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

A string will take the value of any integer at its start, but will not take integers from anywhere else:

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

However, there is a difference when the string is not a valid Integer:

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

Section 2.4: Floats and Integers

```
1/2 #=> 0
```

Since we are dividing two integers, the result is an integer. To solve this problem, we need to cast at least one of those to Float:

```
1.0 / 2       #=> 0.5
1.to_f / 2    #=> 0.5
1 / Float(2) #=> 0.5
```

Alternatively, `fdiv` may be used to return the floating point result of division without explicitly casting either

操作数：

```
1.fdiv 2 # => 0.5
```

operand:

```
1.fdiv 2 # => 0.5
```

第3章：运算符

第3.1节：运算符优先级和方法

从高到低，以下是Ruby的优先级表。高优先级操作先于低优先级操作执行。

运算符	操作	方法？
.	方法调用（例如 foo.bar）	
[] []=	括号查找，括号赋值	✓[1]
! ~ +	布尔非，补码，一元加号	✓[2]
**	乘方	✓
-	一元减号	✓[2]
* / %	乘法，除法，取模	✓
+ -	加法，减法	✓
<>	位移	✓
&	位与	✓
| ^	位或，位异或	✓
< <= >= >	比较	✓
<=> == != === =~ !~	等价，模式匹配，比较	✓[3]
&&	布尔与（Boolean AND）	
||	布尔或（Boolean OR）	
.. ...	包含范围，排除范围	
? :	三元运算符	
rescue	修饰符 rescue	
= += -=	赋值运算符	
defined?	defined 操作符	
not	布尔非（Boolean NOT）	
or and	布尔或，布尔与（Boolean OR, Boolean AND）	
if unless while until	修饰符 if、unless、while、until	
{ }	大括号代码块	
do end	使用 do end 的代码块	

一元加号和一元减号用于 +obj，-obj 或 -(some_expression)。

修饰符 if、修饰符 unless 等用于这些关键字的修饰符版本。例如，下面是一个修饰符-unless 表达式：

```
a += 1 unless a.zero?
```

带有 ✓ 的运算符可以定义为方法。大多数方法的名称与运算符名称完全相同，例如：

```
class Foo
  def **(x)
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "左移 #{y}"
  end
```


Chapter 3: Operators

Section 3.1: Operator Precedence and Methods

From highest to lowest, this is the precedence table for Ruby. High precedence operations happen before low precedence operations.

Operators	Operations	Method?
.	Method call (e.g. foo.bar)	
[] []=	Bracket Lookup, Bracket Set	✓[1]
! ~ +	Boolean NOT, complement, unary plus	✓[2]
**	Exponentiation	✓
-	Unary minus	✓[2]
* / %	Multiplication, division, modulo	✓
+ -	Addition, subtraction	✓
<>	Bitwise shift	✓
&	Bitwise AND	✓
| ^	Bitwise OR, Bitwise XOR	✓
< <= >= >	Comparison	✓
<=> == != === =~ !~	Equality, pattern matching, comparison	✓[3]
&&	Boolean AND	
||	Boolean OR	
.. ...	Inclusive range, Exclusive range	
? :	Ternary operator	
rescue	Modifier rescue	
= += -=	Assignments	
defined?	Defined operator	
not	Boolean NOT	
or and	Boolean OR, Boolean AND	
if unless while until	Modifier if, unless, while, until	
{ }	Block with braces	
do end	Block with do end	

Unary + and unary - are for `+obj`, `-obj` or `-(some_expression)`.

Modifier-if, modifier-unless, etc. are for the modifier versions of those keywords. For example, this is a modifier-unless expression:

```
a += 1 unless a.zero?
```

Operators with a ✓ may be defined as methods. Most methods are named exactly as the operator is named, for example:

```
class Foo
  def **(x)
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "Shifting left by #{y}"
  end
```

```ruby
  def !
    puts "布尔取反"
  end
end

Foo.new ** 2      #=> "幂运算，指数为2"
Foo.new << 3      #=> "左移3位"
!Foo.new          #=> "布尔取反"
```

[1] 括号查找和括号赋值方法（[] 和 []=）其参数定义在名称之后，例如：

```ruby
class Foo
  def [](x)
    puts "查找项目 #{x}"
  end
  def []=(x,y)
    puts "设置项 #{x} 为 #{y}"
  end
end

f = Foo.new
f[:cats] = 42     #=> "设置项 cats 为 42"
f[17]             #=> "查找项 17"
```

[2] “一元加号”和“一元减号”操作符被定义为名为 +@ 和 -@ 的方法，例如

```ruby
class Foo
  def -@
    puts "一元减号"
  end
  def +@
    puts "一元加号"
  end
end

f = Foo.new
+f                #=> "一元加号"
-f                #=> "一元减号"
```

[3] 在早期版本的 Ruby 中，不等于运算符 != 和不匹配运算符 !~ 不能被定义为方法。相反，调用了对应的等于运算符 == 或匹配运算符 =~ 的方法，然后 Ruby 对该方法的结果进行了布尔取反。

如果你没有定义自己的!=或!~操作符，上述行为仍然成立。然而，从Ruby 1.9.1开始，这两个操作符也可以被定义为方法：

```ruby
class Foo
  def ==(x)
    puts "正在检查与 #{x} 的相等性，返回 false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "正在检查与 42 的相等性，返回 false"
puts x            #=> "false"
x = (f != 42)     #=> "正在检查与 42 的相等性，返回 false"
puts x            #=> "true"
```



```ruby
  def !
    puts "Boolean negation"
  end
end

Foo.new ** 2      #=> "Raising to the power of 2"
Foo.new << 3      #=> "Shifting left by 3"
!Foo.new          #=> "Boolean negation"
```

[1] The Bracket Lookup and Bracket Set methods ([ ] and []=) have their arguments defined after the name, for example:

```ruby
class Foo
  def [](x)
    puts "Looking up item #{x}"
  end
  def []=(x,y)
    puts "Setting item #{x} to #{y}"
  end
end

f = Foo.new
f[:cats] = 42     #=> "Setting item cats to 42"
f[17]             #=> "Looking up item 17"
```

[2] The "unary plus" and "unary minus" operators are defined as methods named `+@` and `-@`, for example

```ruby
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f                #=> "unary plus"
-f                #=> "unary minus"
```

[3] In early versions of Ruby the inequality operator `!=` and the non-matching operator `!~` could not be defined as methods. Instead, the method for the corresponding equality operator == or matching operator =~ was invoked, and the result of that method was boolean inverted by Ruby.

If you do not define your own `!=` or `!~` operators the above behavior is still true. However, as of Ruby 1.9.1, those two operators may also be defined as methods:

```ruby
class Foo
  def ==(x)
    puts "checking for EQUALITY with #{x}, returning false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "false"
x = (f != 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "true"
```

```ruby
class Foo
  def !=(x)
    puts "正在检查与 #{x} 的不等性"
  end
end

f != 42         #=> "正在检查与 42 的不等性"
```

第3.2节：案例相等操作符（===）

也称为三重等号。

该运算符不测试相等性，而是测试右操作数是否与左操作数存在"是一个"关系。因此，流行的名称案例相等运算符具有误导性。

这个SO回答这样描述它：描述 `a === b`的最佳方式是"如果我有一个标记为 a的抽屉，放入 b是否合理？"换句话说，集合 a是否包含成员 b？

示例（源代码）

```ruby
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

重写了===的类

许多类重写了===以在case语句中提供有意义的语义。其中一些是：

类	同义词
数组	==
日期	==
模块	是_a?
对象	==
范围	包含?
正则表达式	=~
字符串	==

推荐做法

应避免显式使用case相等运算符===。它不是测试相等，而是测试*包含关系*，使用它可能会引起混淆。使用同义方法时，代码更清晰、更易理解。


```ruby
class Foo
  def !=(x)
    puts "Checking for INequality with #{x}"
  end
end

f != 42          #=> "checking for INequality with 42"
```

Section 3.2: Case equality operator (===)

Also known as *triple equals*.

This operator does not test equality, but rather tests if the right operand has an IS A relationship with the left operand. As such, the popular name *case equality operator* is misleading.

This SO answer describes it thus: the best way to describe `a === b` is "if I have a drawer labeled `a`, does it make sense to put `b` in it?" In other words, does the set `a` include the member `b`?

Examples (source)

```ruby
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

Classes that override `===`

Many classes override `===` to provide meaningful semantics in case statements. Some of them are:

Class	Synonym for
Array	==
Date	==
Module	is_a?
Object	==
Range	include?
Regexp	=~
String	==

Recommended practice

Explicit use of the case equality operator `===` should be avoided. It doesn't test equality but rather *subsumption*, and its use can be confusing. Code is clearer and easier to understand when the synonym method is used instead.

```
# 错误示例
整数 === 42
(1..5) === 3
/ell/ === 'Hello'

# 好的，使用同义词方法
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

第3.3节：安全导航操作符

Ruby 2.3.0 添加了*安全导航操作符*，`&.`。该操作符旨在简化条件语句中`对象 && 对象.属性 && 对象.属性.方法`的模式。

例如，你有一个房屋（House）对象，带有一个地址（address）属性，你想从地址中获取街道名称（street_name）。为了在旧版本Ruby中安全编程以避免nil错误，你会使用类似如下的代码：

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

安全导航操作符简化了这个条件。你可以改写为：

```
if house&.address&.street_name
  house.address.street_name
end
```

注意：
安全导航操作符的行为与链式条件判断不完全相同。使用链式条件判断（第一个例子）时，如果address为false，if代码块将不会执行。安全导航操作符只识别nil值，但允许false等值。如果address是false，使用安全导航操作符将导致错误：

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

第3.4节：赋值运算符

简单赋值

= 是简单赋值。如果变量之前未被引用，则会创建一个新的局部变量。

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

这将输出：

```
x 是 3，y 是 9
```

并行赋值

变量也可以并行赋值，例如 x, y = 3, 9。这在交换值时尤其有用：


```
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

Section 3.3: Safe Navigation Operator

Ruby 2.3.0 added the *safe navigation operator*, `&.`. This operator is intended to shorten the paradigm of `object && object.property && object.property.method` in conditional statements.

For example, you have a `House` object with an `address` property, and you want to find the `street_name` from the `address`. To program this safely to avoid nil errors in older Ruby versions, you'd use code something like this:

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

The safe navigation operator shortens this condition. Instead, you can write:

```
if house&.address&.street_name
  house.address.street_name
end
```

Caution:
The safe navigation operator doesn't have *exactly* the same behavior as the chained conditional. Using the chained conditional (first example), the `if` block would not be executed if, say `address` was **`false`**. The safe navigation operator only recognises **`nil`** values, but permits values such as **`false`**. If `address` is **`false`**, using the SNO will yield an error:

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Section 3.4: Assignment Operators

Simple Assignment

= is a simple assignment. It creates a new local variable if the variable was not previously referenced.

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

This will output:

```
x is 3, y is 9
```

Parallel Assignment

Variables can also be assigned in parallel, e.g. `x, y = 3, 9`. This is especially useful for swapping values:

```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

这将输出：

```
x 是 9，y 是 3
```

简化赋值

可以混合使用运算符和赋值。例如：

```
x = 1
y = 2
puts "x 是 #{x}，y 是 #{y}"

x += y
puts "x 现在是 #{x}"
```

显示以下输出：

```
x 是 1，y 是 2
x 现在是 3
```

简化赋值中可以使用各种运算：

操作员	描述	等价示例
`+=`	对变量进行加法并重新赋值	`x += y` `x = x + y`
`-=`	对变量进行减法并重新赋值	`x -= y` `x = x - y`
`*=`	对变量进行乘法并重新赋值	`x *= y` `x = x * y`
`/=`	除以并重新赋值变量	`x /= y` `x = x / y`
`%=`	除以，取余数，并重新赋值变量	`x %= y` `x = x % y`
`**=`	计算指数并重新赋值变量	`x **= yx = x ** y`

第3.5节：比较运算符

操作员	描述
`==`	**如果两个值相等，则为 true。**
`!=`	**如果两个值不相等，则为 true。**
`<`	**如果左操作数的值小于右操作数的值，则为 true。**
`>`	**如果左操作数的值大于右操作数的值，则为 true。**
`>=`	**如果左操作数的值大于或等于右操作数的值，则为 true。**
`<=`	如果左操作数的值小于或等于右操作数的值，则为 true。
`<=>`	0如果左操作数的值等于右操作数的值， 1如果左操作数的值大于右操作数的值， 如果左操作数的值小于右操作数的值，则为 -1。


```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

This will output:

```
x is 9, y is 3
```

Abbreviated Assignment

It's possible to mix operators and assignment. For example:

```
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Shows the following output:

```
x is 1, y is 2
x is now 3
```

Various operations can be used in abbreviated assignment:

Operator	Description	Example	Equivalent to
`+=`	Adds and reassigns the variable	`x += y`	`x = x + y`
`-=`	Subtracts and reassigns the variable	`x -= y`	`x = x - y`
`*=`	Multiplies and reassigns the variable	`x *= y`	`x = x * y`
`/=`	Divides and reassigns the variable	`x /= y`	`x = x / y`
`%=`	Divides, takes the remainder, and reassigns the variable	`x %= y`	`x = x % y`
`**=`	Calculates the exponent and reassigns the variable	`x **= y`	`x = x ** y`

Section 3.5: Comparison Operators

Operator	Description
`==`	`true` if the two values are equal.
`!=`	`true` if the two values are *not* equal.
`<`	`true` if the value of the operand on the left is *less than* the value on the right.
`>`	`true` if the value of the operand on the left is *greater than* the value on the right.
`>=`	`true` if the value of the operand on the left is *greater than* **or** *equal to* the value on the right.
`<=`	`true` if the value of the operand on the left is *less than* **or** *equal to* the value on the right.
`<=>`	`0` if the value of the operand on the left is *equal to* the value on the right, `1` if the value of the operand on the left is *greater than* the value on the right, `-1` if the value of the operand on the left is *less than* the value on the right.

第4章：变量作用域和可见性

第4.1节：类变量

类变量具有类范围的作用域，可以在类中的任何位置声明。当变量前缀为@@时，该变量将被视为类变量。

```
class Dinosaur
@@classification = “像爬行动物，但又像鸟类”

    def self.classification
       @@classification
    end

    def classification
       @@classification
    end
end

dino = Dinosaur.new
dino.classification
# => "像爬行动物，但像鸟类"

Dinosaur.classification
# => "像爬行动物，但像鸟类"
```

类变量在相关类之间共享，并且可以被子类重写

```
class TRex < Dinosaur
    @@classification = "大牙鸟！"
end

TRex.classification
# => "大牙鸟！"

Dinosaur.classification
# => "大牙鸟！"
```

这种行为大多数情况下是不希望的，可以通过使用类级别的实例变量来避免。

定义在模块内部的类变量不会覆盖包含它们的类的类变量：

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "大牙鸟！"
```


Chapter 4: Variable Scope and Visibility

Section 4.1: Class Variables

Class variables have a class wide scope, they can be declared anywhere in the class. A variable will be considered a class variable when prefixed with `@@`

```
class Dinosaur
    @@classification = "Like a Reptile, but like a bird"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = Dinosaur.new
dino.classification
# => "Like a Reptile, but like a bird"

Dinosaur.classification
# => "Like a Reptile, but like a bird"
```

Class variables are shared between related classes and can be overwritten from a child class

```
class TRex < Dinosaur
    @@classification = "Big teeth bird!"
end

TRex.classification
# => "Big teeth bird!"

Dinosaur.classification
# => "Big teeth bird!"
```

This behaviour is unwanted most of the time and can be circumvented by using class-level instance variables.

Class variables defined inside a module will not overwrite their including classes class variables:

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "Big teeth bird!"
```

第4.2节：局部变量

局部变量（与其他变量类别不同）没有任何前缀

```
local_variable = "local"
p local_variable
# => local
```

其作用域取决于它被声明的位置，不能在“声明容器”之外使用。
例如，如果在方法中声明了一个局部变量，则只能在该方法内部使用该变量。

```
def some_method
method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

当然，局部变量不限于方法，作为经验法则，你可以说，一旦你在 do ... end块内声明一个变量，或者用大括号{}包裹，它就是局部的，并且作用域限定在声明它的块内。

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: 未定义的局部变量或方法 `local_var'
```

但是，在 if 或 case 代码块中声明的局部变量可以在父作用域中使用：

```
if true
usable = "yay"
end

p usable
# yay
# => "yay"
```

虽然局部变量不能在其声明的代码块外使用，但它会被传递到子代码块中：

```
my_variable = "foo"

my_variable.split("").each_with_index do |char, i|
    puts "字符串 '#{my_variable}' 中索引 #{i} 处的字符是 #{char}"
end
# 字符串 'foo' 中索引 0 处的字符是 f
# 字符串 'foo' 中索引 1 处的字符是 o
# 字符串 'foo' 中索引 2 处的字符是 o
```


Section 4.2: Local Variables

Local variables (unlike the other variable classes) do not have any prefix

```
local_variable = "local"
p local_variable
# => local
```

Its scope is dependent on where it has been declared, it can not be used outside the "declaration containers" scope. For example, if a local variable is declared in a method, it can only be used inside that method.

```
def some_method
    method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

Of course, local variables are not limited to methods, as a rule of thumb you could say that, as soon as you declare a variable inside a `do ... end` block or wrapped in curly braces {} it will be local and scoped to the block it has been declared in.

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: undefined local variable or method `local_var'
```

However, local variables declared in `if` or `case` blocks can be used in the parent-scope:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

While local variables can not be used outside of its block of declaration, it will be passed down to blocks:

```
my_variable = "foo"

my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
```

```
# => ["f", "o", "o"]
```

但不适用于方法 / 类 / 模块定义

```
my_variable = "foo"

def some_method
    puts "你不能在这里使用局部变量，明白吗？#{my_variable}"
end

some_method
# NameError: 未定义的局部变量或方法 `my_variable'
```

用于块参数的变量（当然）是块的局部变量，但会遮蔽之前定义的
变量，而不会覆盖它们。

```
overshadowed = "阳光"

["黑暗"].each do |overshadowed|
    p overshadowed
end
# 黑暗
# => ["黑暗"]

p 被掩盖
# "sunlight"
# => "sunlight"
```

第4.3节：全局变量

全局变量具有全局作用域，因此可以在任何地方使用。它们的作用域不依赖于定义的位置。当变量前缀带有$符号时，
该变量将被视为全局变量。

```
$i_am_global = "omg"

class Dinosaur
    def instance_method
                p "全局变量可以在任何地方使用。看见了吗？ #{$i_am_global}, #{$another_global_var}"
    end

    def self.class_method
        $another_global_var = "srsly?"
        p "全局变量可以在任何地方使用。看见了吗？ #{$i_am_global}"
    end
end

恐龙。类方法
# “全局变量可以在任何地方使用。看到了吗？天哪”
# => “全局变量可以在任何地方使用。看到了吗？天哪”

dinosaur = Dinosaur.new
dinosaur.instance_method
# “全局变量可以在任何地方使用。看到了吗？天哪，真的？”
# => “全局变量可以在任何地方使用。看到了吗？天哪，真的？”
```

由于全局变量可以在任何地方定义并且在任何地方可见，调用一个“未定义”的全局
变量将返回 nil，而不是抛出错误。


```
# => ["f", "o", "o"]
```

But not to method / class / module definitions

```
my_variable = "foo"

def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end

some_method
# NameError: undefined local variable or method `my_variable'
```

The variables used for block arguments are (of course) local to the block, but will overshadow previously defined variables, without overwriting them.

```
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"
```

Section 4.3: Global Variables

Global variables have a global scope and hence, can be used everywhere. Their scope is not dependent on where they are defined. A variable will be considered global, when prefixed with a $ sign.

```
$i_am_global = "omg"

class Dinosaur
    def instance_method
       p "global vars can be used everywhere. See? #{$i_am_global}, #{$another_global_var}"
    end

    def self.class_method
       $another_global_var = "srsly?"
       p "global vars can be used everywhere. See? #{$i_am_global}"
    end
end

Dinosaur.class_method
# "global vars can be used everywhere. See? omg"
# => "global vars can be used everywhere. See? omg"

dinosaur = Dinosaur.new
dinosaur.instance_method
# "global vars can be used everywhere. See? omg, srsly?"
# => "global vars can be used everywhere. See? omg, srsly?"
```

Since a global variable can be defined everywhere and will be visible everywhere, calling an "undefined" global variable will return nil instead of raising an error.

```ruby
p $undefined_var
# nil
# => nil
```

虽然全局变量使用方便，但强烈建议使用常量代替。

第4.4节：实例变量

实例变量具有对象范围，它们可以在对象的任何地方声明，然而在类级别声明的实例变量只在类对象中可见。当变量前缀为@时，该变量被视为实例变量。实例变量用于设置和获取对象的属性，若未定义则返回 nil。

```ruby
class 恐龙
    @base_sound = "rawrr"

    def 初始化(声音 = nil)
        @声音 = 声音 || self.class.base_sound
    end

    def 说话
        @声音
    end

    def 尝试说话
        @base_sound
    end

    def 计算并存储声音长度
        @声音.chars.each_with_index do |字符, 索引|
            @声音长度 = 索引 + 1
            p "#{字符}: #{声音长度}"
        end
    end

    def 声音长度
        @声音长度
    end

    def self.base_sound
        @base_sound
    end
end

dino_1 = 恐龙。新建
dino_2 = 恐龙。新建 "grrr"

恐龙。基础声音
# => "rawrr"
dino_2.说话
# => "grrr"
```

在类级别声明的实例变量不能在对象级别访问：

```ruby
dino_1.尝试说话
# => nil
```

但是，我们使用实例变量@base_sound在没有传入声音时实例化声音


```ruby
p $undefined_var
# nil
# => nil
```

Although global variables are easy to use its usage is strongly discouraged in favour of constants.

Section 4.4: Instance Variables

Instance variables have an object wide scope, they can be declared anywhere in the object, however an instance variable declared on class level, will only be visible in the class object. A variable will be considered an instance variable when prefixed with `@`. Instance variables are used to set and get an objects attributes and will return nil if not defined.

```ruby
class Dinosaur
    @base_sound = "rawrr"

    def initialize(sound = nil)
        @sound = sound || self.class.base_sound
    end

    def speak
        @sound
    end

    def try_to_speak
        @base_sound
    end

    def count_and_store_sound_length
        @sound.chars.each_with_index do |char, i|
            @sound_length = i + 1
            p "#{char}: #{sound_length}"
        end
    end

    def sound_length
        @sound_length
    end

    def self.base_sound
        @base_sound
    end
end

dino_1 = Dinosaur.new
dino_2 = Dinosaur.new "grrr"

Dinosaur.base_sound
# => "rawrr"
dino_2.speak
# => "grrr"
```

The instance variable declared on class level can not be accessed on object level:

```ruby
dino_1.try_to_speak
# => nil
```

However, we used the instance variable `@base_sound` to instantiate the sound when no sound is passed to the new

方法：

```
dino_1.说话
# => "rawwr"
```

实例变量可以在对象的任何地方声明，甚至在代码块内部：

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

实例变量不会在同一个类的实例之间共享

```
dino_2.sound_length
# => nil
```

这可以用来创建类级变量，这些变量不会被子类覆盖，因为类在Ruby中也是
对象。

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
# => "quack quack"
DuckDuckDinosaur.base_sound
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```


method:

```
dino_1.speak
# => "rawwr"
```

Instance variables can be declared anywhere in the object, even inside a block:

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

Instance variables are **not** shared between instances of the same class

```
dino_2.sound_length
# => nil
```

This can be used to create class level variables, that will not be overwritten by a child-class, since classes are also objects in Ruby.

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
# => "quack quack"
DuckDuckDinosaur.base_sound
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```

第5章：环境变量

第5.1节：获取用户配置文件路径的示例

```
# 将检索我的主目录路径
ENV['HOME'] # => "/Users/username"

# 将尝试检索环境变量 'FOO'。如果失败，将获取 'bar'
ENV.fetch('FOO', 'bar')
```


Chapter 5: Environment Variables

Section 5.1: Sample to get user profile path

```
# will retrieve my home path
ENV['HOME'] # => "/Users/username"

# will try retrieve the 'FOO' environment variable. If failed, will get 'bar'
ENV.fetch('FOO', 'bar')
```

第6章：常量

第6.1节：定义常量

```
MY_CONSTANT = "Hello, world" # 常量
Constant = '这也是常量' # 常量
my_variable = "Hello, venus" # 不是常量
```

常量名以大写字母开头。所有以大写字母开头的内容在Ruby中都被视为常量。
因此，class和module也是常量。最佳实践是使用全大写字母来声明常量。

第6.2节：修改常量

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

上述代码会产生警告，因为如果你想更改变量的值，应该使用变量。
但是，可以在不发出警告的情况下一次更改常量中的一个字母，示例如下：

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

现在，在更改了MY_CONSTANT的第二个字母后，它变成了"Hullo, world"。

第6.3节：常量不能在方法中定义

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

上述代码会导致错误：SyntaxError: (irb):2: dynamic constant assignment。

第6.4节：在类中定义和更改常量

```
class Message
DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

常量DEFAULT_MESSAGE 可以通过以下代码更改：

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```


Chapter 6: Constants

Section 6.1: Define a constant

```
MY_CONSTANT = "Hello, world" # constant
Constant = 'This is also constant' # constant
my_variable = "Hello, venus" # not constatn
```

Constant name start with capital letter. Everything that start with capital letter are considered as `constant` in Ruby. So `class` and `module` are also constant. Best practice is use all capital letter for declaring constant.

Section 6.2: Modify a Constant

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

The above code results in a warning, because you should be using variables if you want to change their values. However it is possible to change one letter at a time in a constant without a warning, like this:

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

Now, after changing the second letter of `MY_CONSTANT`, it becomes `"Hullo, world"`.

Section 6.3: Constants cannot be defined in methods

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

The above code results in an error: `SyntaxError: (irb):2: dynamic constant assignment`.

Section 6.4: Define and change constants in a class

```
class Message
  DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

The constant `DEFAULT_MESSAGE` can be changed with the following code:

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```

第7章：Ruby中的特殊常量

第7.1节：__FILE__

是从当前执行目录到文件的相对路径
假设我们有如下目录结构：/home/stackoverflow/script.rb
script.rb 内容为：

```
puts __FILE__
```

如果你在 /home/stackoverflow 目录下执行脚本，命令为 `ruby script.rb` 那么 `__FILE__` 会输出 `script.rb` 如果你在 /home 目录下执行，则会输出 `stackoverflow/script.rb`

在2.0版本之前没有 __dir__ 时，获取脚本路径非常有用。

注意__FILE__不等于__dir__

第7.2节：__dir__

`__dir__` 不是常量而是一个函数
`__dir__` 等于 `File.dirname(File.realpath(__FILE__))`

第7.3节：$PROGRAM_NAME 或 $0

包含正在执行的脚本名称。
如果你正在执行该脚本，则与 __FILE__ 相同。

第7.4节：$$

运行此脚本的 Ruby 进程号

第7.5节：$1, $2 等

包含最后一次成功匹配的对应括号组中的子模式，不包括已退出的嵌套块中匹配的模式，如果最后一次匹配失败则为nil。
这些变量都是只读的。

第7.6节：ARGV或$*

为脚本提供的命令行参数。Ruby解释器的选项已被移除。

第7.7节：STDIN

标准输入。$stdin的默认值

第7.8节：STDOUT

标准输出。$stdout的默认值

第7.9节：STDERR

标准错误输出。$stderr的默认值

Chapter 7: Special Constants in Ruby

Section 7.1: __FILE__

Is the relative path to the file from the current execution directory
Assume we have this directory structure: /home/stackoverflow/script.rb
script.rb contains:

```
puts __FILE__
```

If you are inside /home/stackoverflow and execute the script like `ruby script.rb` then `__FILE__` will output `script.rb` If you are inside /home then it will output `stackoverflow/script.rb`

Very useful to get the path of the script in versions prior to 2.0 where `__dir__` doesn't exist.

Note `__FILE__` is not equal to `__dir__`

Section 7.2: __dir__

`__dir__` is not a constant but a function
`__dir__` is equal to `File.dirname(File.realpath(__FILE__))`

Section 7.3: $PROGRAM_NAME or $0

Contains the name of the script being executed.
Is the same as `__FILE__` if you are executing that script.

Section 7.4: $$

The process number of the Ruby running this script

Section 7.5: $1, $2, etc

Contains the subpattern from the corresponding set of parentheses in the last successful pattern matched, not counting patterns matched in nested blocks that have been exited already, or nil if the last pattern match failed. These variables are all read-only.

Section 7.6: ARGV or $*

Command line arguments given for the script. The options for Ruby interpreter are already removed.

Section 7.7: STDIN

The standard input. The default value for $stdin

Section 7.8: STDOUT

The standard output. The default value for $stdout

Section 7.9: STDERR

The standard error output. The default value for $stderr

第7.10节：$stderr

当前的标准错误输出。

第7.11节：$stdout

当前标准输出

第7.12节：$stdin

当前标准输入

第7.13节：ENV

类似哈希的对象包含当前环境变量。在ENV中设置一个值会改变子进程的环境。

Section 7.10: $stderr

The current standard error output.

Section 7.11: $stdout

The current standard output

Section 7.12: $stdin

The current standard input

Section 7.13: ENV

The hash-like object contains current environment variables. Setting a value in ENV changes the environment for child processes.

第8章：注释

第8.1节：单行和多行注释

注释是程序员可读的注释，在运行时被忽略。它们的目的是使源代码更易于理解。

单行注释

使用#字符添加单行注释。

```
#!/usr/bin/ruby -w
# 这是单行注释。
puts "Hello World!"
```

执行时，上述程序将输出 Hello World!

多行注释

多行注释可以通过使用 =begin 和 =end 语法（也称为注释块标记）
如下添加：

```
#!/usr/bin/ruby -w
=begin
这是多行注释。
可以写任意多行。
=end
puts "Hello World!"
```

执行时，上述程序将输出 Hello World!

Chapter 8: Comments

Section 8.1: Single & Multiple line comments

Comments are programmer-readable annotations that are ignored at runtime. Their purpose is to make source code easier to understand.

Single line comments

The # character is used to add single line comments.

```
#!/usr/bin/ruby -w
# This is a single line comment.
puts "Hello World!"
```

When executed, the above program will output `Hello World!`

Multiline comments

Multiple-line comments can be added by using =begin and =end syntax (also known as the comment block markers) as follows:

```
#!/usr/bin/ruby -w
=begin
This is a multiline comment.
Write as many line as you want.
=end
puts "Hello World!"
```

When executed, the above program will output `Hello World!`

第9章：数组

第9.1节：创建字符串数组

可以使用 Ruby 的percent string语法创建字符串数组：

```
array = %w(one two three four)
```

这在功能上等同于定义数组为：

```
array = ['one', 'two', 'three', 'four']
```

除了%w()之外，你还可以使用其他匹配的定界符对：%w{...}、%w[...]或%w<...>。

也可以使用任意非字母数字的定界符，例如：%w!...!、%w#...#或%w@...@。

%W可以替代%w以支持字符串插值。请看以下示例：

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

多个单词可以通过用反斜杠转义空格来表示。

```
%w（科罗拉多 加利福尼亚 纽约） # => ["Colorado", "California", "New York"]
```

第9.2节：使用 Array::new 创建数组

可以使用 Array 的类方法 Array::new 创建一个空数组（[]）：

```
Array.new
```

要设置数组的长度，可以传入一个数字参数：

```
Array.new 3 #=> [nil, nil, nil]
```

有两种方法可以用默认值填充数组：

- 传入一个不可变的值作为第二个参数。
- 传入一个块，该块接收当前索引并生成可变值。

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # 不推荐。
a[1].替换"C"                     # a => ["C", "C", "C"]

b = Array.new(3) { "X" }          # 推荐的方式。
b[1].替换"C"                  # b => ["X", "C", "X"]
```


Chapter 9: Arrays

Section 9.1: Create Array of Strings

Arrays of strings can be created using ruby's percent string syntax:

```
array = %w(one two three four)
```

This is functionally equivalent to defining the array as:

```
array = ['one', 'two', 'three', 'four']
```

Instead of `%w()` you may use other matching pairs of delimiters: `%w{...}`, `%w[...]` or `%w<...>`.

It is also possible to use arbitrary non-alphanumeric delimiters, such as: `%w!...!`, `%w#...#` or `%w@...@`.

`%W` can be used instead of `%w` to incorporate string interpolation. Consider the following:

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

Multiple words can be interpreted by escaping the space with a \.

```
%w(Colorado California New\ York) # => ["Colorado", "California", "New York"]
```

Section 9.2: Create Array with Array::new

An empty Array ([ ]) can be created with Array's class method, `Array::new`:

```
Array.new
```

To set the length of the array, pass a numerical argument:

```
Array.new 3 #=> [nil, nil, nil]
```

There are two ways to populate an array with default values:

- Pass an immutable value as second argument.
- Pass a block that gets current index and generates mutable values.

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # Not recommended.
a[1].replace "C"                # a => ["C", "C", "C"]

b = Array.new(3) { "X" }        # The recommended way.
b[1].replace "C"                # b => ["X", "C", "X"]
```

第9.3节：创建符号数组

版本 ≥ 2.0

```
array = %i(one two three four)
```

创建数组 [:one, :two, :three, :four]。

除了使用 %i(...)，你还可以使用 %i{...} 或 %i[...] 或 %i!...!

此外，如果你想使用插值，可以使用 %I 来实现。

版本 ≥ 2.0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

创建数组：array_one = [:hello, :goodbye, :world] 和 array_two = [:"\#{a}", :"\#{b}", :world]

第9.4节：操作数组元素

添加元素：

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

移除元素：

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // 从索引2删除
# => 3
array
```


Section 9.3: Create Array of Symbols

Version ≥ 2.0

```
array = %i(one two three four)
```

Creates the array [:one, :two, :three, :four].

Instead of %i(...), you may use %i{...} or %i[...] or %i!...!

Additionally, if you want to use interpolation, you can do this with %I.

Version ≥ 2.0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

Creates the arrays: array_one = [:hello, :goodbye, :world] and array_two = [:"\#{a}", :"\#{b}", :world]

Section 9.4: Manipulating Array Elements

Adding elements:

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

Removing elements:

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // delete from index 2
# => 3
array
```

```
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # 删除了所有的2
array - [2, 3, 4]
# => [1]       # 4没有影响
```

合并数组：

```
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

你也可以对数组进行乘法运算，例如

```
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

第9.5节：访问元素

你可以通过索引访问数组的元素。数组索引从0开始计数。

```
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

你可以使用范围来截取数组

```
%w(a b c d)[1..2] # => ['b', 'c']（索引从1到2，包括2）
%w(a b c d)[1...2] # => ['b']（索引从1到2，不包括2）
```

这会返回一个新数组，但不会影响原数组。Ruby也支持使用负数索引。

```
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

你也可以结合使用负数和正数索引

```
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

其他有用的方法

使用first来访问数组中的第一个元素：

```
[1, 2, 3, 4].first # => 1
```



```
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # removed all the 2s
array - [2, 3, 4]
# => [1]       # the 4 did nothing
```

Combining arrays:

```
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

You can also multiply arrays, e.g.

```
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

Section 9.5: Accessing elements

You can access the elements of an array by their indices. Array index numbering starts at `0`.

```
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

You can crop an array using range

```
%w(a b c d)[1..2] # => ['b', 'c'] (indices from 1 to 2, including the 2)
%w(a b c d)[1...2] # => ['b'] (indices from 1 to 2, excluding the 2)
```

This returns a new array, but doesn't affect the original. Ruby also supports the use of negative indices.

```
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

You can combine negative and positive indices as well

```
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

Other useful methods

Use `first` to access the first element in an array:

```
[1, 2, 3, 4].first # => 1
```

或者使用first(n)来访问数组中返回的前 n个元素：

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

同样适用于last和last(n)：

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

使用sample来访问数组中的随机元素：

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

或者使用sample(n)：

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

第9.6节：使用字面量构造器 [ ] 创建数组

数组可以通过将元素列表放在方括号（[和]）中创建。此表示法中的数组元素用逗号分隔：

```
array = [1, 2, 3, 4]
```

数组可以包含任何类型的对象，且可以任意组合，没有类型限制：

```
array = [1, 'b', nil, [3, 4]]
```

第9.7节：解构

任何数组都可以通过将其元素赋值给多个变量来快速解构。一个简单的例子：

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
c = arr[2]
# --- 或者，等同于
a, b, c = arr
```

在变量前加上 splat 操作符（*）会将所有未被其他变量捕获的元素放入该变量的数组中。如果没有剩余元素，则赋值为空数组。单次赋值中只能使用一个 splat：

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # 语法错误: 意外的 *
```

分解是安全的且永远不会引发错误。 nil会被赋值给元素不足的位置，这与访问越界索引时[]操作符的行为一致：

```
arr[9000] # => nil
```


Or `first(n)` to access the first n elements returned in an array:

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

Similarly for `last` and `last(n)`:

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

Use `sample` to access a random element in a array:

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

Or `sample(n)`:

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

Section 9.6: Creating an Array with the literal constructor [ ]

Arrays can be created by enclosing a list of elements in square brackets ([and]). Array elements in this notation are separated with commas:

```
array = [1, 2, 3, 4]
```

Arrays can contain any kind of objects in any combination with no restrictions on type:

```
array = [1, 'b', nil, [3, 4]]
```

Section 9.7: Decomposition

Any array can be quickly **decomposed** by assigning its elements into multiple variables. A simple example:

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
c = arr[2]
# --- or, the same
a, b, c = arr
```

Preceding a variable with the *splat* operator (*) puts into it an array of all the elements that haven't been captured by other variables. If none are left, empty array is assigned. Only one splat can be used in a single assignment:

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # SyntaxError: unexpected *
```

Decomposition is *safe* and never raises errors. `nil`s are assigned where there's not enough elements, matching the behavior of [ ] operator when accessing an index out of bounds:

```
arr[9000] # => nil
```

```
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

分解会隐式调用被赋值对象的to_ary方法。通过在你的类型中实现此方法，你可以获得分解该对象的能力：

```
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

如果被分解的对象不respond_to? to_ary，则将其视为单元素数组：

```
1.respond_to?(:to_ary) # => false
a, b = 1 # a = 1; b = nil
```

分解也可以通过使用()分隔的分解表达式来嵌套，代替原本应为单个元素的部分：

```
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

这实际上是 splat的相反操作。

实际上，任何分解表达式都可以用()括起来。但对于第一层分解，括号是可选的。

```
a, b = [1, 2]
(a, b) = [1, 2] # 同样的效果
```

边界情况：单个标识符不能用作解构模式，无论是外层还是嵌套的：

```
(a) = [1] # 语法错误
a, (b) = [1, [2]] # 语法错误
```

当将数组字面量赋值给解构表达式时，外层的[]可以省略：

```
a, b = [1, 2]
a, b =  1, 2  # 完全相同
```

这被称为并行赋值，但它在底层使用相同的分解方法。这对于在不使用额外临时变量的情况下交换变量的值特别方便：

```
t = a; a = b; b = t # 一种显而易见的方法
a, b = b, a         # 一种惯用的方法
(a, b) = [b, a]     # ...以及它的工作原理
```

在构建赋值右侧时会捕获值，因此使用相同的变量作为源和目标是相对安全的。

第9.8节：数组的并集、交集和差集

```
x = [5, 5, 1, 3]
y = [5, 2, 4, 3]
```



```
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

Decomposition tries to call `to_ary` implicitly on the object being assigned. By implementing this method in your type you get the ability to decompose it:

```
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

If the object being decomposed doesn't `respond_to? to_ary`, it's treated as a single-element array:

```
1.respond_to?(:to_ary) # => false
a, b = 1 # a = 1; b = nil
```

Decomposition can also be **nested** by using a `()`-delimited decomposition expression in place of what otherwise would be a single element:

```
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

This is effectively the opposite of *splat*.

Actually, any decomposition expression can be delimited by `()`. But for the first level decomposition is optional.

```
a, b = [1, 2]
(a, b) = [1, 2] # the same thing
```

Edge case: *a single identifier* cannot be used as a destructuring pattern, be it outer or a nested one:

```
(a) = [1] # SyntaxError
a, (b) = [1, [2]] # SyntaxError
```

When assigning **an array literal** to a destructuring expression, outer [ ] can be omitted:

```
a, b = [1, 2]
a, b =  1, 2  # exactly the same
```

This is known as **parallel assignment**, but it uses the same decomposition under the hood. This is particularly handy for exchanging variables' values without employing additional temporary variables:

```
t = a; a = b; b = t # an obvious way
a, b = b, a         # an idiomatic way
(a, b) = [b, a]     # ...and how it works
```

Values are captured when building the right-hand side of the assignment, so using the same variables as source and destination is relatively safe.

Section 9.8: Arrays union, intersection and difference

```
x = [5, 5, 1, 3]
y = [5, 2, 4, 3]
```

并集（|）包含两个数组中的元素，重复元素会被移除：

```
x | y
=> [5, 1, 3, 2, 4]
```

交集（&）包含同时出现在第一个和第二个数组中的元素：

```
x & y
=> [5, 3]
```

差集（-）包含存在于第一个数组但不存在于第二个数组的元素：

```
x - y
=> [1]
```

第9.9节：使用#compact从数组中移除所有nil元素

如果数组中恰好有一个或多个 `nil`元素且需要移除，可以使用`Array#compact`或`Array#compact!`方法，如下所示。

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#注意该方法返回一个移除nil的新数组副本，
#不会影响原数组

array = [ 1, nil, 'hello', nil, '5', 33]

array = [ 1, nil, 'hello', nil, '5', 33]

#如果需要修改原数组，可以重新赋值

array = array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]

#或者你可以使用更优雅的"惊叹号"版本方法

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

最后，注意如果对没有 nil元素的数组调用#compact或#compact!，这些方法将返回nil。

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

第9.10节：获取数组的所有组合/排列

当调用`permutation`方法并传入代码块时，会生成一个二维数组，包含集合中所有数字的有序序列。

Union (|) contains elements from both arrays, with duplicates removed:

```
x | y
=> [5, 1, 3, 2, 4]
```

Intersection (&) contains elements which are present both in first and second array:

```
x & y
=> [5, 3]
```

Difference (-) contains elements which are present in first array and not present in second array:

```
x - y
=> [1]
```

Section 9.9: Remove all nil elements from an array with #compact

If an array happens to have one or more `nil` elements and these need to be removed, the `Array#compact` or `Array#compact!` methods can be used, as below.

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#notice that the method returns a new copy of the array with nil removed,
#without affecting the original

array = [ 1, nil, 'hello', nil, '5', 33]

array = [ 1, nil, 'hello', nil, '5', 33]

#If you need the original array modified, you can either reassign it

array = array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]

#Or you can use the much more elegant 'bang' version of the method

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

Finally, notice that if `#compact` or `#compact!` are called on an array with no `nil` elements, these will return nil.

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

Section 9.10: Get all combinations / permutations of an array

The `permutation` method, when called with a block yields a two dimensional array consisting of all ordered sequences of a collection of numbers.

如果不传入代码块调用此方法，它将返回一个enumerator。要转换为数组，请调用to_a方法。

示例	结果
[1,2,3].permutation	#<Enumerator: [1,2,3]:permutation
[1,2,3].permutation.to_a	[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
[1,2,3].permutation(2).to_a	[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]

[1,2,3].permutation(4).to_a [] -> 没有长度为4的排列

另一方面，combination 方法在带块调用时会返回一个二维数组，包含一个数字集合的所有序列。与排列不同，组合中不考虑顺序。例如，
[1,2,3] 与 [3,2,1] 是相同的

示例	结果
[1,2,3].combination(1)	#<Enumerator: [1,2,3]:combination
[1,2,3].combination(1).to_a	[[1],[2],[3]]
[1,2,3].combination(3).to_a	[[1,2,3]]

[1,2,3].combination(4).to_a [] -> 没有长度为4的组合

单独调用 combination 方法会返回一个枚举器。要获得数组，请调用 to_a 方法。

repeated_combination 和 repeated_permutation 方法类似，不同之处在于同一个元素可以被多次重复。

例如，序列[1,1]、[1,3,3,1]、[3,3,3]在常规组合和排列中是不合法的。

示例	# 组合
[1,2,3].组合(3).转为数组.长度	1
[1,2,3].重复组合(3).转为数组.长度	6
[1,2,3,4,5].组合(5).转为数组.长度	1
[1,2,3].重复组合(5).转为数组.长度	126

第9.11节：注入（Inject），归约（reduce）

注入和归约是同一操作的不同名称。在其他语言中，这些函数通常被称为折叠（folds）（如foldl或foldr）。这些方法在每个Enumerable对象上都可用。

注入接受一个两个参数的函数，并将其应用于数组中所有元素对。

对于数组[1, 2, 3]，我们可以通过指定起始值和代码块，将所有元素加起来，示例如下：

```
[1,2,3].归约(0) {|a,b| a + b} # => 6
```

这里我们传入一个起始值和一个代码块，代码块表示将所有值相加。代码块第一次运行时，0作为a，1作为b，之后将结果作为下一次的a，接着将1与第二个值2相加。
然后我们取那个结果（3）并将其加到列表中的最后一个元素（也是3）上，得到我们的结果（6）。

如果省略第一个参数，它将把 a 设置为列表中的第一个元素，所以上面的例子等同于：

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

此外，我们可以不传递带有函数的代码块，而是传递一个作为符号的命名函数，可以带有

If this method is called without a block, it will return an `enumerator`. To convert to an array, call the `to_a` method.

Example	Result
[1,2,3].permutation	#<Enumerator: [1,2,3]:permutation
[1,2,3].permutation.to_a	[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
[1,2,3].permutation(2).to_a	[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]

[1,2,3].permutation(4).to_a [ ] -> No permutations of length 4

The `combination` method on the other hand, when called with a block yields a two-dimensional array consisting of all sequences of a collection of numbers. Unlike permutation, order is disregarded in combinations. For example, `[1,2,3]` is the same as `[3,2,1]`

Example	Result
[1,2,3].combination(1)	#<Enumerator: [1,2,3]:combination
[1,2,3].combination(1).to_a	[[1],[2],[3]]
[1,2,3].combination(3).to_a	[[1,2,3]]

[1,2,3].combination(4).to_a [ ] -> No combinations of length 4

Calling the combination method by itself will result in an enumerator. To get an array, call the `to_a` method.

The `repeated_combination` and `repeated_permutation` methods are similar, except the same element can be repeated multiple times.

For example the sequences `[1,1]`, `[1,3,3,1]`, `[3,3,3]` would not be valid in regular combinations and permutations.

Example	# Combos
[1,2,3].combination(3).to_a.length	1
[1,2,3].repeated_combination(3).to_a.length	6
[1,2,3,4,5].combination(5).to_a.length	1
[1,2,3].repeated_combination(5).to_a.length	126

Section 9.11: Inject, reduce

Inject and reduce are different names for the same thing. In other languages these functions are often called folds (like foldl or foldr). These methods are available on every Enumerable object.

Inject takes a two argument function and applies that to all of the pairs of elements in the Array.

For the array `[1, 2, 3]` we can add all of these together with the starting value of zero by specifying a starting value and block like so:

```
[1,2,3].reduce(0) {|a,b| a + b} # => 6
```

Here we pass the function a starting value and a block that says to add all of the values together. The block is first run with `0` as `a` and `1` as `b` it then takes the result of that as the next `a` so we are then adding `1` to the second value `2`. Then we take the result of that (3) and add that on to the final element in the list (also 3) giving us our result (6).

If we omit the first argument, it will set `a` to being the first element in the list, so the example above is the same as:

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

In addition, instead of passing a block with a function, we can pass a named function as a symbol, either with a

起始值，也可以不带。这样，上面的例子可以写成：

```
[1,2,3].reduce(0, :+) # => 6
```

或者省略起始值：

```
[1,2,3].reduce(:+) # => 6
```

第9.12节：数组过滤

我们经常只想对满足特定条件的数组元素进行操作：

选择

将返回匹配特定条件的元素

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

拒绝

将返回不匹配特定条件的元素

```
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

select 和 reject 都返回数组，因此它们可以链式调用：

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

第9.13节：#map

#map，由Enumerable提供，通过对每个元素调用块并收集结果来创建数组：

```
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

原始数组不会被修改；会返回一个新数组，包含与源值顺序相同的转换值。如果想修改原始数组，可以使用map!方法。

在map方法中，你可以对数组中的所有元素调用方法或使用proc。

```
# 对所有元素调用to_i方法
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 对所有元素使用proc（lambda）
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```


starting value, or without. With this, the above example could be written as:

```
[1,2,3].reduce(0, :+) # => 6
```

or omitting the starting value:

```
[1,2,3].reduce(:+) # => 6
```

Section 9.12: Filtering arrays

Often we want to operate only on elements of an array that fulfill a specific condition:

Select

Will return elements that match a specific condition

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

Reject

Will return elements that do not match a specific condition

```
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

Both *#select* and *#reject* return an array, so they can be chained:

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

Section 9.13: #map

#map, provided by Enumerable, creates an array by invoking a block on each element and collecting the results:

```
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

The original array is not modified; a new array is returned containing the transformed values in the same order as the source values. `map!` can be used if you want to modify the original array.

In `map` method you can call method or use proc to all elements in array.

```
# call to_i method on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# using proc (lambda) on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

map与collect同义。

第9.14节：数组和splat（*）操作符

*操作符可以用来解包变量和数组，使它们可以作为单独的参数传递给方法。

如果对象还不是数组，可以用它来将单个对象包装成数组：

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

在上述示例中，wrap_in_array方法接受一个参数，value。

如果value是一个Array，其元素将被展开，并创建一个包含这些元素的新数组。

如果value是单个对象，则创建一个包含该单个对象的新数组。

如果value是 nil，则返回一个空数组。

在某些情况下，splat操作符作为方法参数使用时特别方便。例如，它允许以一致的方式处理 **nil**、单个值和数组：

```
def list(*values)
  values.each do |value|
    # 对value执行某些操作
    puts value
  end
end

list(100)
#> 100

list([100, 200])
#> 100
#> 200

list(nil)
# 没有输出
```

第9.15节：二维数组

使用Array::new构造函数，可以初始化一个指定大小的数组，并在每个位置创建一个新的数组。内部数组也可以指定大小和初始值。

例如，创建一个3x4的零数组：

`map` is synonymous with `collect`.

Section 9.14: Arrays and the splat (*) operator

The `*` operator can be used to unpack variables and arrays so that they can be passed as individual arguments to a method.

This can be used to wrap a single object in an Array if it is not already:

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

In the above example, the `wrap_in_array` method accepts one argument, `value`.

If `value` is an **`Array`**, its elements are unpacked and a new array is created containing those element.

If `value` is a single object, a new array is created containing that single object.

If `value` is **`nil`**, an empty array is returned.

The splat operator is particularly handy when used as an argument in methods in some cases. For example, it allows **`nil`**, single values and arrays to be handled in a consistent manner:

```
def list(*values)
  values.each do |value|
    # do something with value
    puts value
  end
end

list(100)
#> 100

list([100, 200])
#> 100
#> 200

list(nil)
# nothing is outputted
```

Section 9.15: Two-dimensional array

Using the **`Array`**`::new` constructor, your can initialize an array with a given size and a new array in each of its slots. The inner arrays can also be given a size and and initial value.

For instance, to create a 3x4 array of zeros:

```
array = Array.new(3) { Array.new(4) { 0 } }
```

上述生成的数组用p打印时如下所示：

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

你可以这样读取或写入元素：

```
x = array[0][1]
array[2][3] = 2
```

第9.16节：将多维数组转换为一维（扁平化）数组

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

如果你有一个多维数组，需要将其变成一个简单的（一维）数组，可以使用 ***#flatten方法。***

第9.17节：获取数组中的唯一元素

如果你需要读取数组元素，避免重复，可以使用#uniq方法：

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

如果你想从数组中移除所有重复的元素，可以使用#uniq!方法：

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

虽然输出结果相同，#uniq!方法还会将新的数组存储起来：

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

第9.18节：创建数字数组

创建数字数组的常规方法：

```
numbers = [1, 2, 3, 4, 5]
```

范围对象可以广泛用于创建数字数组：


```
array = Array.new(3) { Array.new(4) { 0 } }
```

The array generated above looks like this when printed with p:

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

You can read or write to elements like this:

```
x = array[0][1]
array[2][3] = 2
```

Section 9.16: Turn multi-dimensional array into a one-dimensional (flattened) array

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

If you have a multi-dimensional array and you need to make it a *simple* (i.e. one-dimensional) array, you can use the ***#flatten*** method.

Section 9.17: Get unique array elements

In case you need to read an array elements ***avoiding repetitions*** you case use the ***#uniq*** method:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

Instead, if you want to remove all duplicated elements from an array, you may use ***#uniq!*** method:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

While the output is the same, ***#uniq!*** also stores the new array:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

Section 9.18: Create Array of numbers

The normal way to create an array of numbers:

```
numbers = [1, 2, 3, 4, 5]
```

Range objects can be used extensively to create an array of numbers:

```ruby
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

#step 和 #map 方法允许我们对数字范围施加条件：

```ruby
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64, 81,
100]
```

以上所有方法都是立即加载数字。如果你需要懒加载它们：

```ruby
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

第9.19节：创建一个连续数字或字母的数组

这可以通过对一个Range对象调用Enumerable的#to_a方法轻松实现：

```ruby
(1..10).to_a     #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

(a..b)表示它将包含a和b之间的所有数字。要排除最后一个数字，使用a...b

```ruby
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

或

```ruby
('a'..'f').to_a    #=> ["a", "b", "c", "d", "e", "f"]
('a'...'f').to_a   #=> ["a", "b", "c", "d", "e"]
```

创建数组的一个方便快捷方式是[*a..b]

```ruby
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

第9.20节：从任意对象转换为数组

要从任意对象获取数组，使用Kernel#Array。

以下是一个示例：

```ruby
Array('something') #=> ["something"]
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

例如，您可以替换以下代码中的join_as_string方法


```ruby
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

#step and *#map* methods allow us to impose conditions on the range of numbers:

```ruby
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64, 81,
100]
```

All the above methods load the numbers eagerly. If you have to load them lazily:

```ruby
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Section 9.19: Create an Array of consecutive numbers or letters

This can be easily accomplished by calling **Enumerable#to_a** on a **Range** object:

```ruby
(1..10).to_a     #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

(a..b) means that it will include all numbers between a and b. To exclude the last number, use a...b

```ruby
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

or

```ruby
('a'..'f').to_a    #=> ["a", "b", "c", "d", "e", "f"]
('a'...'f').to_a   #=> ["a", "b", "c", "d", "e"]
```

A convenient shortcut for creating an array is [*a..b]

```ruby
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

Section 9.20: Cast to Array from any object

To get Array from any object, use **Kernel#Array**.

The following is an example:

```ruby
Array('something') #=> ["something"]
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

For example, you could replace join_as_string method from the following code

```ruby
def join_as_string(arg)
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

到以下代码。

```ruby
def join_as_string(arg)
  Array(arg).join(',')
end
```



```ruby
def join_as_string(arg)
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
    arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

to the following code.

```ruby
def join_as_string(arg)
  Array(arg).join(',')
end
```

第10章：多维数组

Ruby中的多维数组就是其元素为其他数组的数组。

唯一的问题是，由于 Ruby 数组可以包含混合类型的元素，你必须确信你操作的数组实际上是由其他数组组成的，而不是例如数组和字符串混合在一起。

第10.1节：初始化二维数组

我们先回顾一下如何初始化一个一维的 Ruby 整数数组：

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

二维数组只是数组的数组，你可以这样初始化它：

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

第10.2节：初始化三维数组

你可以再深入一层，添加第三层数组。规则不变：

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

第10.3节：访问嵌套数组

访问第一个子数组的第3个元素：

```
my_array[1][2]
```

第10.4节：数组扁平化

给定一个多维数组：

```
my_array = [[1, 2], ['a', 'b']]
```

扁平化操作是将所有数组子元素分解到根数组中：

Chapter 10: Multidimensional Arrays

Multidimensional Arrays in Ruby are just arrays whose elements are other arrays.

The only catch is that since Ruby arrays can contain elements of mixed types, you must be confident that the array that you are manipulating is effectively composed of other arrays and not, for example, arrays and strings.

Section 10.1: Initializing a 2D array

Let's first recap how to initialize a 1D ruby array of integers:

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

Being a 2D array simply an array of arrays, you can initialize it like this:

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

Section 10.2: Initializing a 3D array

You can go a level further down and add a third layer of arrays. The rules don't change:

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

Section 10.3: Accessing a nested array

Accessing the 3rd element of the first subarray:

```
my_array[1][2]
```

Section 10.4: Array flattening

Given a multidimensional array:

```
my_array = [[1, 2], ['a', 'b']]
```

the operation of flattening is to decompose all array children into the root array:

```
my_array.flatten

# [1, 2, 'a', 'b']
```



```
my_array.flatten

# [1, 2, 'a', 'b']
```

第11章：字符串

第11.1节：单引号和双引号字符串字面量的区别

主要区别在于双引号字符串字面量支持字符串插值和完整的转义序列。

例如，它们可以通过插值包含任意Ruby表达式：

```
# 单引号字符串不支持插值
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# 双引号字符串支持插值
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

双引号字符串还支持完整的转义序列集，包括"\"，"\"等...

```
puts 'HelloWorld'# Hell
oWorld

puts "HelloWorld"# Hel
lo
# World
```

... 而单引号字符串不支持任何转义序列，仅保留单引号字符串有用的最小集合：字面单引号和反斜杠，分别是'\''和'\\'。

第11.2节：创建字符串

Ruby提供了多种创建字符串对象的方法。最常见的方法是使用单引号或双引号来创建一个"字符串字面量"：

```
s1 = 'Hello'
s2 = "Hello"
```

主要区别在于双引号字符串字面量更灵活，因为它们支持插值和一些反斜杠转义序列。

还有几种使用任意字符串定界符创建字符串字面量的可能方法。任意字符串定界符是一个%，后面跟着一对匹配的定界符：

```
%(一个字符串)
%{一个字符串}
%<一个字符串>
%|一个字符串|
%!一个字符串!
```

最后，你可以使用%q和%Q序列，它们分别等同于'和"：

```
puts %q(一个字符串)
# 一个字符串
puts %q(现在是 #{Time.now})
```


Chapter 11: Strings

Section 11.1: Difference between single-quoted and double-quoted String literals

The main difference is that double-quoted `String` literals support string interpolations and the full set of escape sequences.

For instance, they can include arbitrary Ruby expressions via interpolation:

```
# Single-quoted strings don't support interpolation
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# Double-quoted strings support interpolation
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

Double-quoted strings also support the entire set of escape sequences including `"\n"`, `"\t"`...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... while single-quoted strings support *no* escape sequences, baring the minimal set necessary for single-quoted strings to be useful: Literal single quotes and backslashes, `'\''` and `'\\'` respectively.

Section 11.2: Creating a String

Ruby provides several ways to create a `String` object. The most common way is using single or double quotes to create a "string literal":

```
s1 = 'Hello'
s2 = "Hello"
```

The main difference is that double-quoted string literals are a little bit more flexible as they support interpolation and some backslash escape sequences.

There are also several other possible ways to create a string literal using arbitrary string delimiters. An arbitrary string delimiter is a `%` followed by a matching pair of delimiters:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Finally, you can use the `%q` and `%Q` sequence, that are equivalent to `'` and `"`:

```
puts %q(A string)
# A string
puts %q(Now is #{Time.now})
```

```ruby
# 现在是 #{Time.now}

puts %Q(一个字符串)
# 一个字符串
puts %Q(现在是 #{Time.now})
# 现在是 2016-07-21 12:47:45 +0200
```

%q 和 %Q 序列在字符串包含单引号、双引号或两者混合时非常有用。这样，你就不需要对内容进行转义：

```ruby
%Q(<a href="/profile">用户的个人资料<a>)
```

您可以使用多种不同的分隔符，只要成对匹配即可：

```ruby
%q(一个字符串)
%q{一个字符串}
%q<一个字符串>
%q|一个字符串|
%q!一个字符串!
```

第11.3节：大小写操作

```ruby
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

这四个方法不会修改原始接收者。例如，

```ruby
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

有四个类似的方法执行相同操作，但会修改原始接收者。

```ruby
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

例如，

```ruby
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

注释：

- 在 Ruby 2.4 之前，这些方法不支持处理 Unicode。

第11.4节：字符串连接

使用 + 运算符连接字符串：

```ruby
s1 = "Hello"
s2 = " "
```



```ruby
# Now is #{Time.now}

puts %Q(A string)
# A string
puts %Q(Now is #{Time.now})
# Now is 2016-07-21 12:47:45 +0200
```

%q and %Q sequences are useful when the string contains either single quotes, double quotes, or a mix of both. In this way, you don't need to escape the content:

```ruby
%Q(<a href="/profile">User's profile<a>)
```

You can use several different delimiters, as long as there is a matching pair:

```ruby
%q(A string)
%q{A string}
%q<A string>
%q|A string|
%q!A string!
```

Section 11.3: Case manipulation

```ruby
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

These four methods do not modify the original receiver. For example,

```ruby
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

There are four similar methods that perform the same actions but modify original receiver.

```ruby
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

For example,

```ruby
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Notes:

- prior to Ruby 2.4 these methods do not handle unicode.

Section 11.4: String concatenation

Concatenate strings with the + operator:

```ruby
s1 = "Hello"
s2 = " "
```

```ruby
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
puts s
# => Hello World
```

或者使用 << 运算符：

```ruby
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

请注意，<< 运算符会修改左侧的对象。

你也可以对字符串进行乘法操作，例如

```ruby
"wow" * 3
# => "wowwowwow"
```

第11.5节：字符串定位

在Ruby中，字符串可以左对齐、右对齐或居中要左对齐字符串，

使用ljust方法。该方法接受两个参数，第一个是整数，表示新字符串的字符数，第二个是字符串，表示填充的模式。

如果整数大于原字符串的长度，新字符串将左对齐，剩余空间由可选的字符串参数填充。如果未提供字符串参数，则用空格填充字符串。

```ruby
str ="abcd"
str.ljust(4)          => "abcd"
str.ljust(10)         => "abcd      "
```

要右对齐字符串，使用 rjust方法。该方法接受两个参数，第一个是整数，表示新字符串的字符数，第二个是字符串，表示填充的模式。

如果整数大于原字符串的长度，新字符串将右对齐，剩余空间由可选的字符串参数填充。如果未提供字符串参数，则用空格填充字符串。

```ruby
str = "abcd"
str.rjust(4)          => "abcd"
str.rjust(10)         => "      abcd"
```

要使字符串居中，请使用center方法。该方法接受两个参数，第一个是表示新字符串宽度的整数，第二个是用于填充的字符串。原字符串将被居中对齐。

```ruby
str = "abcd"
str.center(4)          => "abcd"
str.center(10)         => "   abcd   "
```



```ruby
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
puts s
# => Hello World
```

Or with the << operator:

```ruby
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Note that the << operator modifies the object on the left hand side.

You also can multiply strings, e.g.

```ruby
"wow" * 3
# => "wowwowwow"
```

Section 11.5: Positioning strings

In Ruby, strings can be left-justified, right-justified or centered

To left-justify string, use the `ljust` method. This takes in two parameters, an integer representing the number of characters of the new string and a string, representing the pattern to be filled.

If the integer is greater than the length of the original string, the new string will be left-justified with the optional string parameter taking the remaining space. If the string parameter is not given, the string will be padded with spaces.

```ruby
str ="abcd"
str.ljust(4)          => "abcd"
str.ljust(10)         => "abcd      "
```

To right-justify a string, use the `rjust` method. This takes in two parameters, an integer representing the number of characters of the new string and a string, representing the pattern to be filled.

If the integer is greater than the length of the original string, the new string will be right-justified with the optional string parameter taking the remaining space. If the string parameter is not given, the string will be padded with spaces.

```ruby
str = "abcd"
str.rjust(4)          => "abcd"
str.rjust(10)         => "      abcd"
```

To center a string, use the `center` method. This takes in two parameters, an integer representing the width of the new string and a string, which the original string will be padded with. The string will be aligned to the center.

```ruby
str = "abcd"
str.center(4)          => "abcd"
str.center(10)         => "   abcd   "
```

第11.6节：拆分字符串

String#split根据分隔符将String拆分成Array。

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

空的String会得到一个空的Array：

```
"".split(",")
# => []
```

不匹配的分隔符会得到一个只包含单个元素的Array：

```
"alpha,beta".分割(".")
# => ["alpha,beta"]
```

你也可以使用正则表达式来分割字符串：

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

分隔符是可选的，默认情况下字符串会按空白字符分割：

```
"alpha beta".split
# => ["alpha", "beta"]
```

第11.7节：字符串开头判断

要判断字符串是否以某个模式开头，start_with?方法非常有用

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

你也可以使用index方法检查模式的位置

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

第11.8节：字符串连接

Array#join 将一个 Array 根据分隔符连接成一个 String：

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

分隔符是可选的，默认是一个空的 String。

```
["alpha", "beta"].join
# => "alphabeta"
```

无论使用什么分隔符，空的 Array 都会得到一个空的 String。

```
[].join(",")
```


Section 11.6: Splitting a String

`String#split` splits a `String` into an `Array`, based on a delimiter.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

An empty `String` results into an empty `Array`:

```
"".split(",")
# => []
```

A non-matching delimiter results in an `Array` containing a single item:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

You can also split a string using regular expressions:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

The delimiter is optional, by default a string is split on whitespace:

```
"alpha beta".split
# => ["alpha", "beta"]
```

Section 11.7: String starts with

To find if a string starts with a pattern, the `start_with?` method comes in handy

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

You can also check the position of the pattern with `index`

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

Section 11.8: Joining Strings

`Array#join` joins an `Array` into a `String`, based on a delimiter:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

The delimiter is optional, and defaults to an empty `String`.

```
["alpha", "beta"].join
# => "alphabeta"
```

An empty `Array` results in an empty `String`, no matter which delimiter is used.

```
[].join(",")
```

```
# => ""
```

第11.9节：字符串插值

双引号分隔符 " 和 %Q 序列支持使用 #{ruby_expression} 进行字符串插值：

```
puts "现在是 #{Time.now}"
# 现在是 现在是 2016-07-21 12:47:45 +0200

puts %Q(现在是 #{Time.now})
# 现在是 2016-07-21 12:47:45 +0200
```

第11.10节：字符串结尾

要判断字符串是否以某个模式结尾，end_with? 方法非常有用

```
str = "我喜欢菠萝"
str.end_with?("pineaaples")       => false
```

第11.11节：格式化字符串

Ruby 可以通过用提供的数组中的值替换任何占位符，将一组值注入字符串中。

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

占位符由两个 %s 表示，值由数组 ['Hello', 'br3nt'] 提供。 % 操作符指示字符串注入数组中的值。

第11.12节：字符串替换

```
p "这是 %s" % "foo"
# => "这是 foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

有关更多细节，请参见 String % 文档和 Kernel::sprintf 。

第11.13节：多行字符串

创建多行字符串最简单的方法是直接在引号之间使用多行：

```
address = "八十七年前，我们的先辈在这片大陆上建立了一个新国家，
这个国家诞生于自由，并致力于
所有人生而平等的主张。"
```

这种方法的主要问题是，如果字符串中包含引号，会破坏字符串的语法。为了解决这个问题，可以使用heredoc：


```
# => ""
```

Section 11.9: String interpolation

The double-quoted delimiter " and %Q sequence supports string interpolation using *#{ruby_expression}*:

```
puts "Now is #{Time.now}"
# Now is Now is 2016-07-21 12:47:45 +0200

puts %Q(Now is #{Time.now})
# Now is Now is 2016-07-21 12:47:45 +0200
```

Section 11.10: String ends with

To find if a string ends with a pattern, the end_with? method comes in handy

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

Section 11.11: Formatted strings

Ruby can inject an array of values into a string by replacing any placeholders with the values from the supplied array.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

The place holders are represented by two %s and the values are supplied by the array ['Hello', 'br3nt']. The % operator instructs the string to inject the values of the array.

Section 11.12: String Substitution

```
p "This is %s" % "foo"
# => "This is foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

See String % docs and Kernel::sprintf for more details.

Section 11.13: Multiline strings

The easiest way to create a multiline string is to just use multiple lines between quotation marks:

```
address = "Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal."
```

The main problem with that technique is that if the string includes a quotation, it'll break the string syntax. To work around the problem, you can use a heredoc instead:

```
puts <<-RAVEN
从前在一个阴郁的午夜，当我沉思时，虚弱且疲惫，
翻阅着许多古怪且奇异的遗忘典籍—
        当我点头，几乎打盹时，突然传来敲击声，
仿佛有人轻轻敲打，敲打我的房门。
  “是某个访客，”我喃喃道，“敲打我的房门—
              仅此而已，别无他物。”
渡鸦
```

Ruby 支持使用<<EOT的 shell 风格 here 文档，但终止文本必须从行首开始。这会破坏代码缩进，因此没有太多理由使用这种风格。不幸的是，字符串会根据代码本身的缩进而带有缩进。

Ruby 2.3 通过引入<<~解决了这个问题，它会去除多余的前导空格：

版本 ≥ 2.3

```
def build_email(address)
  return (<<~EMAIL)
    收件人: #{address}

    敬启者：

请停止在日出时吹风笛！

此致，
你的邻居
  EMAIL
end
```

百分号字符串也可以用来创建多行字符串：

```
%q(
哈姆雷特        你看到那边的云了吗，它几乎像骆驼的形状？
波洛涅斯        说实话，它确实像骆驼。
哈姆雷特        我觉得它更像黄鼠狼。
波洛涅斯        它的背部像黄鼠狼。
哈姆雷特        或者像鲸鱼？
波洛涅斯        非常像鲸鱼
)
```

有几种方法可以避免插值和转义序列：

- 单引号代替双引号：' 是回车符。'百分比字符串中的小写字母 q ：%q[
- #{not-a-variable}]
- 在 heredoc 中用单引号括起终端字符串：

```
<<-'CODE'
puts 'Hello world!'
CODE
```

第11.14节：字符串字符替换

tr 方法返回一个字符串的副本，其中第一个参数的字符被第二个参数的字符替换。


```
puts <<-RAVEN
  Once upon a midnight dreary, while I pondered, weak and weary,
  Over many a quaint and curious volume of forgotten lore—
      While I nodded, nearly napping, suddenly there came a tapping,
  As of some one gently rapping, rapping at my chamber door.
  "'Tis some visitor," I muttered, "tapping at my chamber door—
              Only this and nothing more."
  RAVEN
```

Ruby supports shell-style here documents with `<<EOT`, but the terminating text must start the line. That screws up code indentation, so there's not a lot of reason to use that style. Unfortunately, the string will have indentations depending no how the code itself is indented.

Ruby 2.3 solves the problem by introducing <<~ which strips out excess leading spaces:

Version ≥ 2.3

```
def build_email(address)
  return (<<~EMAIL)
    TO: #{address}

    To Whom It May Concern:

    Please stop playing the bagpipes at sunrise!

    Regards,
    Your neighbor
  EMAIL
end
```

Percent Strings also work to create multiline strings:

```
%q(
HAMLET        Do you see yonder cloud that's almost in shape of a camel?
POLONIUS        By the mass, and 'tis like a camel, indeed.
HAMLET        Methinks it is like a weasel.
POLONIUS        It is backed like a weasel.
HAMLET        Or like a whale?
POLONIUS        Very like a whale
)
```

There are a few ways to avoid interpolation and escape sequences:

- Single quote instead of double quote: `'\n is a carriage return.'`
- Lower case q in a percent string: `%q[#{not-a-variable}]`
- Single quote the terminal string in a heredoc:

```
<<-'CODE'
  puts 'Hello world!'
CODE
```

Section 11.14: String character replacements

The `tr` method returns a copy of a string where the characters of the first argument are replaced by the characters of the second argument.

```
"string".tr('r', 'l') # => "stling"
```

要仅替换模式的第一个出现项，请使用 sub 方法

```
"string ring".sub('r', 'l') # => "stling ring"
```

如果想替换模式的所有出现项，请使用 gsub

```
"string ring".gsub('r','l') # => "stling ling"
```

要删除字符，请为第二个参数传入空字符串

你也可以在所有这些方法中使用正则表达式。

重要的是要注意，这些方法只会返回字符串的新副本，而不会原地修改字符串。要做到这一点，你需要分别使用tr!、sub!和gsub!方法。

第11.15节：理解字符串中的数据

在Ruby中，字符串只是一个字节序列以及一个编码名称（例如UTF-8、US-ASCII、ASCII-8BIT），该编码指定了如何将这些字节解释为字符。

Ruby字符串可以用来保存文本（基本上是字符序列），在这种情况下通常使用UTF-8编码。

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Ruby字符串也可以用来保存二进制数据（字节序列），在这种情况下通常使用ASCII-8BIT编码。

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

字符串中的字节序列可能与编码不匹配，如果你尝试使用该字符串，可能会导致错误。

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```



```
"string".tr('r', 'l') # => "stling"
```

To replace only the first occurrence of a pattern with with another expression use the **sub** method

```
"string ring".sub('r', 'l') # => "stling ring"
```

If you would like to replace *all* occurrences of a pattern with that expression use **gsub**

```
"string ring".gsub('r','l') # => "stling ling"
```

To delete characters, pass in an empty string for the second parameter

You can also use regular expressions in all these methods.

It's important to note that these methods will only return a new copy of a string and won't modify the string in place. To do that, you need to use the `tr!`, **`sub!`** and **`gsub!`** methods respectively.

Section 11.15: Understanding the data in a string

In Ruby, a string is just a sequence of bytes along with the name of an encoding (such as `UTF-8`, `US-ASCII`, `ASCII-8BIT`) that specifies how you might interpret those bytes as characters.

Ruby strings can be used to hold text (basically a sequence of characters), in which case the UTF-8 encoding is usually used.

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Ruby strings can also be used to hold binary data (a sequence of bytes), in which case the ASCII-8BIT encoding is usually used.

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

It is possible for the sequence of bytes in a string to not match the encoding, resulting in errors if you try to use the string.

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

第12章：日期时间

第12.1节：从字符串解析日期时间

DateTime.parse 是一个非常有用的方法，可以从字符串构造一个DateTime对象，自动猜测其格式。

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

注意：parse 还识别许多其他格式。

第12.2节：New

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

当前时间：

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

注意它会给出你所在时区的当前时间

第12.3节：对DateTime加/减天数

DateTime + Fixnum（天数数量）

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

DateTime + Float（天数）

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

DateTime + Rational（天数）

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

DateTime - Fixnum（天数）

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

DateTime - Float（天数）

```
DateTime.new(2015,12,30,23,0) - 2.5
```


Chapter 12: DateTime

Section 12.1: DateTime from string

`DateTime.parse` is a very useful method which construct a DateTime from a string, guessing its format.

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

Note: There are lots of other formats that `parse` recognizes.

Section 12.2: New

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Current time:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Note that it gives the current time in your timezone

Section 12.3: Add/subtract days to DateTime

`DateTime` + `Fixnum` (days quantity)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (days quantity)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (days quantity)

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Fixnum` (days quantity)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (days quantity)

```
DateTime.new(2015,12,30,23,0) - 2.5
```

```
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

DateTime - 有理数（天数）

```
DateTime.new(2015,12,30,23,0) - 有理数(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```



```
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

DateTime - **Rational** (days quantity)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

第13章：时间

第13.1节：如何使用strftime方法

将时间转换为字符串在Ruby中是非常常见的操作。strftime是用于将时间转换为字符串的方法。

以下是一些示例：

```
Time.now.strftime("%Y-%m-%d  %H:%M:%S") #=> "2016-07-27  08:45:42"
```

这还可以进一步简化

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

第13.2节：创建时间对象

获取当前时间：

```
Time.now
Time.new # 如果不带参数，等同于此用法
```

获取指定时间：

```
Time.new(2010, 3, 10) # 2010年3月10日（午夜）
Time.new(2015, 5, 3, 10, 14) # 2015年5月3日10:14 AM
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") # 2050年5月3日21:08:16（+10:00时区）
```

要将时间转换为epoch，可以使用to_i方法：

```
Time.now.to_i # => 1478633386
```

你也可以使用at方法将epoch转换回时间：

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```


Chapter 13: Time

Section 13.1: How to use the strftime method

Converting a time to a string is a pretty common thing to do in Ruby. `strftime` is the method one would use to convert time to a string.

Here are some examples:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

This can be simplified even further

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

Section 13.2: Creating time objects

Get current time:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Get specific time:

```
Time.new(2010, 3, 10) #10 March 2010 (Midnight)
Time.new(2015, 5, 3, 10, 14) #10:14 AM on 3 May 2015
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") #09:08:16 PM on 3 May 2050
```

To convert a time to epoch you can use the `to_i` method:

```
Time.now.to_i # => 1478633386
```

You can also convert back from epoch to Time using the `at` method:

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

第14章：数字

第14.1节：将字符串转换为整数

您可以使用Integer方法将String转换为Integer：

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

您也可以向Integer方法传递一个进制参数，以将某个进制的数字转换过来

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

请注意，如果参数无法转换，该方法会抛出ArgumentError异常：

```
Integer("hello")
# 抛出 ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# 抛出 ArgumentError: invalid value for Integer(): "23-hello"
```

你也可以使用String#to_i方法。然而，该方法的容错性稍强，并且其行为与Integer不同：

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

String#to_i 接受一个参数，表示将数字解释为的进制：

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

第14.2节：创建整数

```
0       # 创建Fixnum 0
123     # 创建Fixnum 123
1_000   # 创建Fixnum 1000。你可以使用_作为分隔符以提高可读性
```

默认情况下，表示法是十进制。然而，还有一些内置的不同进制表示法：

```
0xFF    # 255的十六进制表示，以0x开头
0b100   # 4的二进制表示，以0b开头
0555    # 365的八进制表示，以0开头和数字组成
```

第14.3节：数字四舍五入

round 方法会根据小数点后第一位数字进行四舍五入：如果该数字是5或更大，则向上取整；如果是4或更小，则向下取整。该方法接受一个可选参数，用于指定所需的精度。

Chapter 14: Numbers

Section 14.1: Converting a String to Integer

You can use the `Integer` method to convert a `String` to an `Integer`:

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

You can also pass a base parameter to the `Integer` method to convert numbers from a certain base

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Note that the method raises an `ArgumentError` if the parameter cannot be converted:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

You can also use the `String#to_i` method. However, this method is slightly more permissive and has a different behavior than `Integer`:

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` accepts an argument, the base to interpret the number as:

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

Section 14.2: Creating an Integer

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

By default the notation is base 10. However, there are some other built-in notations for different bases:

```
0xFF    # Hexadecimal representation of 255, starts with a 0x
0b100   # Binary representation of 4, starts with a 0b
0555    # Octal representation of 365, starts with a 0 and digits
```

Section 14.3: Rounding Numbers

The round method will round a number up if the first digit after its decimal place is 5 or higher and round down if that digit is 4 or lower. This takes in an optional argument for the precision you're looking for.

```
4.89.round          # => 5
4.25.round          # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

浮点数也可以使用floor

方法向下取整到小于该数的最大整数

```
4.9999999999999.floor # => 4
```

也可以使用ceil方法向上取整到大于该数的最小整数

```
4.0000000000001.ceil  # => 5
```

第14.4节：偶数和奇数

可以使用 even? 方法来判断一个数字是否为偶数

```
4.even?       # => true
5.even?       # => false
```

可以使用 odd? 方法来判断一个数字是否为奇数

```
4.odd?        # => false
5.odd?        # => true
```

第14.5节：有理数

Rational 表示一个分子和分母形式的有理数：

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.分子   # => 19
r3.分母 # => 6
有理数(2, 4) # => (1/2)
```

创建有理数的其他方法

```
有理数('2/3')  # => (2/3)
有理数(3)      # => (3/1)
有理数(3, -5)  # => (-3/5)
有理数(0.2)    # => (3602879701896397/18014398509481984)
有理数('0.2')  # => (1/5)
0.2.to_r        # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

第14.6节：复数

```
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)
```



```
4.89.round          # => 5
4.25.round          # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

Floating point numbers can also be rounded down to the highest integer lower than the number with the `floor` method

```
4.9999999999999.floor # => 4
```

They can also be rounded up to the lowest integer higher than the number using the `ceil` method

```
4.0000000000001.ceil  # => 5
```

Section 14.4: Even and Odd Numbers

The even? method can be used to determine if a number is even

```
4.even?       # => true
5.even?       # => false
```

The odd? method can be used to determine if a number is odd

```
4.odd?        # => false
5.odd?        # => true
```

Section 14.5: Rational Numbers

`Rational` represents a rational number as numerator and denominator:

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Other ways of creating a Rational

```
Rational('2/3')  # => (2/3)
Rational(3)      # => (3/1)
Rational(3, -5)  # => (-3/5)
Rational(0.2)    # => (3602879701896397/18014398509481984)
Rational('0.2')  # => (1/5)
0.2.to_r         # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

Section 14.6: Complex Numbers

```
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)
```

```
polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.909297426825682i)
```

第14.7节：将数字转换为字符串

Fixnum#to_s 接受一个可选的进制参数，并以该进制表示给定的数字：

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

如果未提供参数，则默认以十进制表示数字

```
2.to_s # => "2"
10423.to_s # => "10423"
```

第14.8节：两个数字相除

在除两个数字时，请注意你希望返回的类型。请注意，除两个整数时会**调用整数除法**。如果你的目标是执行浮点除法，至少有一个参数应该是浮点数类型。

整数除法：

```
3 / 2 # => 1
```

浮点除法

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```



```
polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.909297426825682i)
```


Section 14.7: Converting a number to a string

Fixnum#to_s takes an optional base argument and represents the given number in that base:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

If no argument is provided, then it represents the number in base 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

Section 14.8: Dividing two numbers

When dividing two numbers pay attention to the type you want in return. Note that dividing **two integers will invoke the integer division**. If your goal is to run the float division, at least one of the parameters should be of `float` type.

Integer division:

```
3 / 2 # => 1
```

Float division

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```

第15章：符号

第15.1节：创建符号

创建Symbol对象最常见的方法是给字符串标识符前加冒号：

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

以下是结合String字面量定义Symbol的一些替代方法：

```
:"a_symbol"
"a_symbol".to_sym
```

符号还支持%s序列，允许使用任意定界符，类似于字符串的%q和%Q：

```
%s(a_symbol)
%s{a_symbol}
```

%s特别适合从包含空白的输入创建符号：

```
%s{一个符号} # => :"一个符号"
```

虽然可以用某些字符串标识符创建一些有趣的符号（`:/`, `:[]`, `:^` 等），但请注意符号
不能用数字标识符创建：

```
:1 # => 语法错误，意外的 tINTEGER，...
:0.3 # => 语法错误，意外的 tFLOAT，...
```

符号可以以单个 ？ 或 ！ 结尾，而不需要使用字符串字面量作为符号的标识符：

```
:hello?  # :"hello?" 并非必要。
:world!  # :"world!" 并非必要。
```

注意，所有这些不同的方法创建的符号都会返回相同的对象：

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

自 Ruby 2.0 起，有一种快捷方式可以从单词创建符号数组：

```
%i(numerator denominator) == [:numerator, :denominator]
```

第15.2节：将字符串转换为符号

给定一个字符串：

```
s = "something"
```

有几种方法可以将其转换为符号：

```
s.to_sym
# => :something
```


Chapter 15: Symbols

Section 15.1: Creating a Symbol

The most common way to create a `Symbol` object is by prefixing the string identifier with a colon:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Here are some alternative ways to define a `Symbol`, in combination with a `String` literal:

```
:"a_symbol"
"a_symbol".to_sym
```

Symbols also have a `%s` sequence that supports arbitrary delimiters similar to how `%q` and `%Q` work for strings:

```
%s(a_symbol)
%s{a_symbol}
```

The `%s` is particularly useful to create a symbol from an input that contains white space:

```
%s{a symbol} # => :"a symbol"
```

While some interesting symbols (`:/`, `:[]`, `:^`, etc.) can be created with certain string identifiers, note that symbols cannot be created using a numeric identifier:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

Symbols may end with a single ? or ! without needing to use a string literal as the symbol's identifier:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Note that all of these different methods of creating symbols will return the same object:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Since Ruby 2.0 there is a shortcut for creating an array of symbols from words:

```
%i(numerator denominator) == [:numerator, :denominator]
```

Section 15.2: Converting a String to Symbol

Given a `String`:

```
s = "something"
```

there are several ways to convert it to a `Symbol`:

```
s.to_sym
# => :something
```

```
:"#{s}"
# => :something
```

第15.3节：将符号转换为字符串

给定一个符号：

```
s = :something
```

将其转换为String的最简单方法是使用Symbol#to_s方法：

```
s.to_s
# => "something"
```

另一种方法是使用Symbol#id2name方法，它是Symbol#to_s方法的别名。但这是Symbol类特有的方法：

```
s.id2name
# => "something"
```


:"#{s}"
=> :something

Section 15.3: Converting a Symbol to String

Given a `Symbol`:

```
s = :something
```

The simplest way to convert it to a `String` is by using the `Symbol#to_s` method:

```
s.to_s
# => "something"
```

Another way to do it is by using the `Symbol#id2name` method which is an alias for the `Symbol#to_s` method. But it's a method that is unique to the `Symbol` class:

```
s.id2name
# => "something"
```

第16章：Comparable

参数	详细信息
other	与self比较的实例

第16.1节：按面积可比较的矩形

`Comparable`是Ruby中最流行的模块之一。它的目的是提供方便的比较方法。

要使用它，您必须includeComparable并定义太空船操作符（<=>）：

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)

r2 >= r1 # => true
r2.between? r1, r3 # => true
r3.between? r1, r2 # => false
```


Chapter 16: Comparable

Parameter	Details
other	The instance to be compared to `self`

Section 16.1: Rectangle comparable by area

`Comparable` is one of the most popular modules in Ruby. Its purpose is to provide with convenience comparison methods.

To use it, you have to `include Comparable` and define the space-ship operator (`<=>`):

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)

r2 >= r1 # => true
r2.between? r1, r3 # => true
r3.between? r1, r2 # => false
```

第17章：控制流

第17.1节：if、elsif、else 和 end

Ruby 提供了预期的if和else表达式用于分支逻辑，以end关键字结束：

```
# 模拟抛硬币
result = [:heads, :tails].sample

if result == :heads
  puts '硬币结果为“正面”'
else
  puts '硬币结果为“反面”'
end
```

在 Ruby 中，if语句是会计算出值的表达式，结果可以赋值给变量：

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby 还提供了类似 C 风格的三元运算符（详情见此），可以表达为：

```
some_statement ? if_true : if_false
```

这意味着上面的使用 if-else 的示例也可以写成

```
status = age < 18 ? :minor : :adult
```

此外，Ruby 提供了 elsif 关键字，它接受一个表达式以实现额外的分支逻辑：

```
label = if shirt_size == :s
          '小号'
        elsif shirt_size == :m
          '中号'
        elsif shirt_size == :l
          '大号'
        else
          '未知尺寸'
        end
```

如果 if/elsif 链中的所有条件都不为真，且没有 else 分支，则表达式的值为 nil。这在字符串插值中很有用，因为 nil.to_s 是空字符串：

```
"user#{'s' if @users.size != 1}"
```

第17.2节：Case语句

Ruby 使用case关键字来表示 switch 语句。

根据Ruby 文档：

Chapter 17: Control Flow

Section 17.1: if, elsif, else and end

Ruby offers the expected `if` and `else` expressions for branching logic, terminated by the `end` keyword:

```
# Simulate flipping a coin
result = [:heads, :tails].sample

if result == :heads
  puts 'The coin-toss came up "heads"'
else
  puts 'The coin-toss came up "tails"'
end
```

In Ruby, `if` statements are expressions that evaluate to a value, and the result can be assigned to a variable:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby also offers C-style ternary operators (see here for details) that can be expressed as:

```
some_statement ? if_true : if_false
```

This means the above example using if-else can also be written as

```
status = age < 18 ? :minor : :adult
```

Additionally, Ruby offers the `elsif` keyword which accepts an expression to enables additional branching logic:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
        end
```

If none of the conditions in an `if`/`elsif` chain are true, and there is no `else` clause, then the expression evaluates to nil. This can be useful inside string interpolation, since `nil.to_s` is the empty string:

```
"user#{'s' if @users.size != 1}"
```

Section 17.2: Case statement

Ruby uses the `case` keyword for switch statements.

As per the Ruby Docs:

> Case 语句由一个可选条件组成，该条件作为case的参数位置，后面跟零个或多个when子句。第一个匹配条件的when子句（或者当条件为空时，评估为布尔真值的when子句）“赢得胜利”，其代码块将被执行。case语句的值是成功匹配的when子句的值，如果没有匹配的子句，则为nil。
>
> case 语句可以以`else`子句结束。每个`when`语句可以有多个候选值，用逗号分隔。

示例：

```
case x
when 1,2,3
  puts "1、2 或 3"
when 10
  puts "10"
else
  puts "其他数字"
end
```

简短版本：

```
case x
当 1、2、3 时 输出 "1、2 或 3"
当 10 时 输出 "10"
否则 输出 "其他数字"
结束
```

`case` 子句的值通过 `===` 方法（不是 `==`）与每个 `when` 子句匹配。因此它可以用于多种不同类型的对象。

可以将 case 语句与 Ranges 一起使用：

```
case 17
当 13..19
  输出 "青少年"
结束
```

可以将 case 语句与 Regexp 一起使用：

```
case "google"
当 /oo/
  输出 "单词包含 oo"
结束
```

可以将 case 语句与 Proc 或 lambda 一起使用：

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "偶数或小于零"
end
```

case 语句可以与类一起使用：

```
case x
when Integer
  puts "这是一个整数"
```


> Case statements consist of an optional condition, which is in the position of an argument to `case`, and zero or more `when` clauses. The first `when` clause to match the condition (or to evaluate to Boolean truth, if the condition is null) “wins”, and its code stanza is executed. The value of the case statement is the value of the successful `when` clause, or `nil` if there is no such clause.
>
> A case statement can end with an `else` clause. Each `when` a statement can have multiple candidate values, separated by commas.

Example:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Shorter version:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

The value of the `case` clause is matched with each `when` clause using the `===` method (not `==`). Therefore it can be used with a variety of different types of objects.

A `case` statement can be used with Ranges:

```
case 17
when 13..19
  puts "teenager"
end
```

A `case` statement can be used with a Regexp:

```
case "google"
when /oo/
  puts "word contains oo"
end
```

A `case` statement can be used with a Proc or lambda:

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
end
```

A `case` statement can be used with Classes:

```
case x
when Integer
  puts "It's an integer"
```

```ruby
when String
  puts "这是一个字符串"
end
```

通过实现===方法，你可以创建自己的匹配类：

```ruby
class Empty
  def self.===(object)
    !object or "" == object
  end
end

案例 "
when 空
  puts "名字为空"
else
  puts "名字不为空"
end
```

A `case` 语句可以在没有值的情况下使用来匹配：

```ruby
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts '既不是A也不是B'
end
```

A `case` 语句有一个值，因此你可以将它用作方法参数或赋值中：

```ruby
description = case 16
              when 13..19 then "青少年"
              else ""
              end
```

第17.3节：真值和假值

在Ruby中，只有两个值被视为“假值”，在作为条件测试时会返回false，用于`if`表达式。它们是：

- `nil`
- 布尔值false

所有其他值都被视为“真值”，包括：

- `0`- 数值零（整数或其他类型）
- `""` - 空字符串
- `" "` - 仅包含空白字符的字符串[] - 空数组
- `[]`
- `{}` - 空哈希

例如，以下代码：

```ruby
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
```



```ruby
when String
  puts "It's a string"
end
```

By implementing the `===` method you can create your own match classes:

```ruby
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

A `case` statement can be used without a value to match against:

```ruby
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

A `case` statement has a value, so you can use it as a method argument or in an assignment:

```ruby
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

Section 17.3: Truthy and Falsy values

In Ruby, there are exactly two values which are considered "falsy", and will return false when tested as a condition for an `if` expression. They are:

- `nil`
- boolean `false`

All other values are considered "truthy", including:

- `0` - numeric zero (Integer or otherwise)
- `""` - Empty strings
- `"\n"` - Strings containing only whitespace
- `[]` - Empty arrays
- `{}` - Empty hashes

Take, for example, the following code:

```ruby
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
```

```
  puts "#{var_name} 是 #{is_truthy}"
end

check_truthy("false", false)check_tr
uthy("nil", nil)check_truthy("0"
, 0)check_truthy("空字符
串", "")check_truthy("\\n", "")

check_truthy("空数组", [])
check_truthy("空哈希", {})
```

将输出：

```
false 是假值
nil 是假值
0是真值
空 字符串 是真值
 是真值
空 数组 是真值
空哈希 是真值
```

第17.4节：内联 if/unless

一个常见的模式是使用内联，或尾随的 if 或 unless：

```
puts "x 小于 5" 如果 x < 5
```

这被称为条件修饰符，是添加简单保护代码和提前返回的方便方法：

```
def save_to_file(data, filename)
  raise "未提供文件名" 如果 filename.empty?
  除非 data.valid?，否则返回 false

  File.write(filename, data)
end
```

无法为这些修饰符添加 else 子句。通常也不建议在主逻辑中使用条件修饰符——对于复杂代码，应使用普通的 if、elsif、else 语句。

第17.5节：while，until

while 循环在给定条件满足时执行代码块：

```
i = 0
while i < 5
  输出 "迭代 ##{i}"
  i +=1
end
```

一个until循环在条件为假时执行代码块：

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```



```
  puts "#{var_name} is #{is_truthy}"
end

check_truthy("false", false)
check_truthy("nil", nil)
check_truthy("0", 0)
check_truthy("empty string", "")
check_truthy("\\n", "\n")
check_truthy("empty array", [])
check_truthy("empty hash", {})
```

Will output:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

Section 17.4: Inline if/unless

A common pattern is to use an inline, or trailing, `if` or `unless`:

```
puts "x is less than 5" if x < 5
```

This is known as a conditional *modifier*, and is a handy way of adding simple guard code and early returns:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

It is not possible to add an `else` clause to these modifiers. Also it is generally not recommended to use conditional modifiers inside the main logic -- For complex code one should use normal `if`, `elsif`, `else` instead.

Section 17.5: while, until

A `while` loop executes the block while the given condition is met:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

An `until` loop executes the block while the conditional is false:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

第17.6节：翻转-翻转操作符

翻转-翻转操作符..用于条件语句中的两个条件之间：

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

该条件在第一部分变为true之前评估为false。然后它评估为true直到第二部分变为`true`。之后它再次切换为`false`。

此示例说明了被选择的内容：

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

翻转-翻转操作符仅在if语句（包括unless）和三元操作符中有效。否则，它被视为范围操作符。

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

它可以多次在false和true之间切换：

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

第17.7节：或等于/条件赋值运算符（||=）

Ruby 有一个或等于运算符，允许仅当变量的值为 nil 或 false 时，才将一个值赋给该变量。

```
||= # 这是实现该功能的运算符。
```

这个运算符中，双竖线表示"或"，等号表示赋值。你可能会认为它表示类似这样的内容：

```
x = x || y
```

上述示例是不正确的。或等于运算符实际上表示的是：

```
x || x = y
```

如果 x 的值为 nil 或 false，则将 x 赋值为 y，否则保持不变。

这里是 or-equals 操作符的一个实际用例。假设你的代码中有一部分需要向用户发送电子邮件。如果由于某种原因该用户没有电子邮件地址，你会怎么做呢？你可能会写成这样：

Section 17.6: Flip-Flop operator

The flip flop operator `..` is used between two conditions in a conditional statement:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

The condition evaluates to **`false`** *until* the first part becomes **`true`**. Then it evaluates to **`true`** *until* the second part becomes **`true`**. After that it switches to **`false`** again.

This example illustrates what is being selected:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

The flip-flop operator only works inside ifs (including `unless`) and ternary operator. Otherwise it is being considered as the range operator.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

It can switch from **`false`** to **`true`** and backwards multiple times:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Section 17.7: Or-Equals/Conditional assignment operator (||=)

Ruby has an or-equals operator that allows a value to be assigned to a variable if and only if that variable evaluates to either **`nil`** or **`false`**.

```
||= # this is the operator that achieves this.
```

this operator with the double pipes representing or and the equals sign representing assigning of a value. You may think it represents something like this:

```
x = x || y
```

this above example is not correct. The or-equals operator actually represents this:

```
x || x = y
```

If `x` evaluates to **`nil`** or **`false`** then `x` is assigned the value of `y`, and left unchanged otherwise.

Here is a practical use-case of the or-equals operator. Imagine you have a portion of your code that is expected to send an email to a user. What do you do if for what ever reason there is no email for this user. You might write something like this:

```
 if user_email.nil?
user_email = "error@yourapp.com"
 end
```

使用 or-equals 操作符，我们可以简化这整段代码，提供清晰、简洁的控制和功能。

```
user_email ||= "error@yourapp.com"
```

在 false 是有效值的情况下，必须小心不要意外覆盖它：

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

第17.8节：unless

一个常见的语句是if !(某个条件)。Ruby提供了unless语句作为替代。

结构与if语句完全相同，只是条件为否定。此外，unless语句不支持elsif，但支持else：

```
# 输出 not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

第17.9节：throw，catch

与许多其他编程语言不同，Ruby中的throw和catch关键字与异常处理无关。

在Ruby中，throw和catch有点像其他语言中的标签。它们用于改变控制流，但与异常（Exception）所代表的“错误”概念无关。

```
catch(:out) do
  catch(:nested) do
    puts "nested"
  end

  puts "before"
throw :out
  puts "不会被执行"
end
puts "之后"
# 输出 "nested", "before", "after"
```

第17.10节：三元运算符

Ruby 有一个三元运算符（?:），根据条件是否为真返回两个值中的一个：

```
conditional ? value_if_truthy : value_if_falsy
```



```
if user_email.nil?
   user_email = "error@yourapp.com"
end
```

Using the or-equals operator we can cut this entire chunk of code, providing clean, clear control and functionality.

```
user_email ||= "error@yourapp.com"
```

In cases where **false** is a valid value, care must be taken to not override it accidentally:

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

Section 17.8: unless

A common statement is `if !(some condition)`. Ruby offers the alternative of the `unless` statement.

The structure is exactly the same as an `if` statement, except the condition is negative. Also, the `unless` statement does not support `elsif`, but it does support `else`:

```
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

Section 17.9: throw, catch

Unlike many other programming languages, the `throw` and `catch` keywords are not related to exception handling in Ruby.

In Ruby, `throw` and `catch` act a bit like labels in other languages. They are used to change the control flow, but are not related to a concept of "error" like Exceptions are.

```
catch(:out) do
  catch(:nested) do
    puts "nested"
  end

  puts "before"
  throw :out
  puts "will not be executed"
end
puts "after"
# prints "nested", "before", "after"
```

Section 17.10: Ternary operator

Ruby has a ternary operator (`?:`), which returns one of two value based on if a condition evaluates as truthy:

```
conditional ? value_if_truthy : value_if_falsy
```

```ruby
value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

这与写成`if a then b else c end`是一样的，尽管三元表达式更受欢迎

示例：

```ruby
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

第17.11节：使用 break、next 和 redo 控制循环

Ruby 块的执行流程可以通过break、next和redo语句来控制。

`break`

`break`语句会立即退出块。块中剩余的指令将被跳过，
迭代将结束：

```ruby
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

index += 1
  puts "当前正在执行的动作：#{action}"
end

# 当前正在执行的动作：run
# 当前正在执行的动作：jump
# 当前正在执行的动作：swim
```

`next`

next语句会立即返回到代码块的顶部，并继续下一次迭代。代码块中剩余的任何指令将被跳过：

```ruby
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "当前正在执行的动作：#{action}"
```



```ruby
value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

it is the same as writing `if a then b else c end`, though the ternary is preferred

Examples:

```ruby
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

Section 17.11: Loop control with break, next, and redo

The flow of execution of a Ruby block may be controlled with the `break`, `next`, and `redo` statements.

`break`

The `break` statement will exit the block immediately. Any remaining instructions in the block will be skipped, and the iteration will end:

```ruby
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

  index += 1
  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
```

`next`

The `next` statement will return to the top of the block immediately, and proceed with the next iteration. Any remaining instructions in the block will be skipped:

```ruby
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "Currently doing this action: #{action}"
```

```
end

# 当前正在执行的动作：run
# 当前正在执行的动作：jump
# 当前正在执行的动作：swim
# 当前正在执行的动作：macarena
```

redo

redo 语句会立即返回到代码块的顶部，并重试同一次迭代。代码块中剩余的任何指令将被跳过：

```
actions = %w(跑 跳 游泳 睡觉 macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "当前执行的动作: #{action}"

  if action == "睡觉"
    repeat_count += 1
    redo if repeat_count < 3
  end

index += 1
end

# 当前执行的动作: 跑
# 当前执行的动作: 跳
# 当前执行的动作: 游泳
# 当前执行的动作: 睡觉
# 当前执行的动作: 睡觉
# 当前执行的动作: 睡觉
# 当前执行的动作: macarena
```

Enumerable 迭代

除了循环，这些语句还适用于 Enumerable 迭代方法，例如 each 和 map：

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "项目: #{item}"
end

# 项目：1
# 项目：3
```

块结果值

在 break 和 next 语句中，都可以提供一个值，该值将被用作块的结果值：

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "第一个偶数值是：#{even_value}"
```



```
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: macarena
```

redo

The redo statement will return to the top of the block immediately, and retry the same iteration. Any remaining instructions in the block will be skipped:

```
actions = %w(run jump swim sleep macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "Currently doing this action: #{action}"

  if action == "sleep"
    repeat_count += 1
    redo if repeat_count < 3
  end

  index += 1
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: macarena
```

Enumerable iteration

In addition to loops, these statements work with Enumerable iteration methods, such as each and map:

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

Block result values

In both the break and next statements, a value may be provided, and will be used as a block result value:

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "The first even value is: #{even_value}"
```

```
# 第一个偶数值是：2
```

第17.12节：return 与 next：块中的非局部返回

考虑这个 broken 代码片段：

```
def foo
bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'bar'
  bar
end
foo # => 0
```

人们可能会期望 return 返回 map 块结果数组的一个值。因此，foo 的返回值会是 [1, 0, 3, 0]。相反，**return 返回的是方法 foo 的值**。注意 baz 没有被打印，这意味着执行从未到达那一行。

带值的 next 可以解决这个问题。它充当块级的 return。

```
def foo
bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'bar'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

如果没有 return，块返回的值就是其最后一个表达式的值。

第17.13节：开始，结束

begin 块是一种控制结构，用于将多个语句组合在一起。

```
begin
a = 7
  b = 6
  a * b
end
```

begin 块将返回块中最后一个语句的值。以下示例将返回 3。

```
begin
  1
  2
  3
end
```

begin 块对于使用 ||= 运算符进行条件赋值非常有用，此时可能需要多个语句来返回结果。

```
circumference ||=
```



```
# The first even value is: 2
```

Section 17.12: return vs. next: non-local return in a block

Consider this *broken* snippet:

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

One might expect `return` to yield a value for `map`'s array of block results. So the return value of `foo` would be `[1, 0, 3, 0]`. Instead, `return` **returns a value from the method** `foo`. Notice that `baz` isn't printed, which means execution never reached that line.

`next` with a value does the trick. It acts as a block-level `return`.

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

In the absence of a `return`, the value returned by the block is the value of its last expression.

Section 17.13: begin, end

The `begin` block is a control structure that groups together multiple statements.

```
begin
  a = 7
  b = 6
  a * b
end
```

A `begin` block will return the value of the last statement in the block. The following example will return 3.

```
begin
  1
  2
  3
end
```

The `begin` block is useful for conditional assignment using the `||=` operator where multiple statements may be required to return a result.

```
circumference ||=
```

```
  begin
radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

它还可以与其他块结构如 rescue、ensure、while、if、unless 等结合使用，以提供更灵活的程序流程控制。

Begin 块不是代码块，如 { ... } 或 do ... end；它们不能作为参数传递给函数。

第17.14节：使用逻辑语句的控制流程

虽然这看起来有些违反直觉，但你可以使用逻辑运算符来判断某个语句是否执行。

例如：

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

这将检查文件是否存在，只有在文件不存在时才打印错误信息。这里的 or 语句是惰性的，这意味着一旦确定其值为真或假，它就会停止执行。一旦第一个条件被判定为真，就不需要检查另一个条件的值。但如果第一个条件为假，则必须检查第二个条件。

一个常见的用法是设置默认值：

```
glass = glass or 'full' # 乐观主义者！
```

这会将 glass 的值设置为 'full'，前提是它尚未被设置。更简洁的写法是使用符号版本的 or：

```
glass ||= 'empty' # 悲观主义者。
```

也可以仅在第一个条件为假时执行第二个语句：

```
File.exist?(filename) and puts "#{filename} found!"
```

再次，and 是惰性的，因此只有在需要得出一个值时才会执行第二个语句。

“or”运算符的优先级低于“and”。同样，“||”的优先级低于“&&”。符号形式的优先级高于单词形式。当你想将这种技巧与赋值混合使用时，这一点很有用：

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

注意Ruby风格指南推荐：

> 禁止使用“and”和“or”关键字。它们带来的可读性提升极小，不值得引入高概率的细微错误。对于布尔表达式，总是使用“&&”和“||”。对于流程控制，使用“if”和“unless”；“&&”和“||”也可以接受，但不够清晰。


```
  begin
    radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

It can also be combined with other block structures such as `rescue`, `ensure`, `while`, `if`, `unless`, etc to provide greater control of program flow.

`Begin` blocks are not code blocks, like `{ ... }` or `do ... end`; they cannot be passed to functions.

Section 17.14: Control flow with logic statements

While it might seem counterintuitive, you can use logical operators to determine whether or not a statement is run. For instance:

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

This will check to see if the file exists and only print the error message if it doesn't. The `or` statement is lazy, which means it'll stop executing once it's sure which whether it's value is true or false. As soon as the first term is found to be true, there's no need to check the value of the other term. But if the first term is false, it must check the second term.

A common use is to set a default value:

```
glass = glass or 'full' # Optimist!
```

That sets the value of `glass` to 'full' if it's not already set. More concisely, you can use the symbolic version of `or`:

```
glass ||= 'empty' # Pessimist.
```

It's also possible to run the second statement only if the first one is false:

```
File.exist?(filename) and puts "#{filename} found!"
```

Again, `and` is lazy so it will only execute the second statement if necessary to arrive at a value.

The `or` operator has lower precedence than `and`. Similarly, `||` has lower precedence than `&&`. The symbol forms have higher precedence than the word forms. This is handy to know when you want to mix this technique with assignment:

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

Note that the Ruby Style Guide recommends:

> The `and` and `or` keywords are banned. The minimal added readability is just not worth the high probability of introducing subtle bugs. For boolean expressions, always use `&&` and `||` instead. For flow control, use `if` and `unless`; `&&` and `||` are also acceptable but less clear.

第18章：方法

Ruby中的函数提供了组织良好、可重用的代码来执行一组操作。函数简化了编码过程，避免了冗余逻辑，并使代码更易于理解。本章节介绍了函数的声明与使用、参数、yield语句以及作用域。

第18.1节：定义方法

方法使用“def”关键字定义，后跟方法名和可选的参数列表（用括号括起）。“def”和“end”之间的Ruby代码构成方法的主体。

```
def hello(name)
  "Hello, #{name}"
end
```

方法调用指定方法名、要调用该方法的对象（有时称为接收者），以及零个或多个赋值给命名方法参数的参数值。

```
hello("World")
# => "Hello, World"
```

当接收者未明确指定时，默认为self。

参数名可以在方法体内作为变量使用，这些命名参数的值来自方法调用时传入的参数。

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

第18.2节：向块传递控制权

你可以向方法传递一个块，并且方法可以多次调用该块。这可以通过传递proc/lambda等实现，但使用yield更简单、更快速：

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "终于到了这里：#{arg2}"
  yield
end
simple('开始','结束') { puts "现在我们在 yield 内部" }

#> 首先我们在这里： 开始
#> 现在我们在 yield 内部
#> 最后我们在这里： 结束
#> 现在我们在 yield 内部
```

注意 `{ puts ... }` 不在括号内，它隐式地出现在后面。这也意味着我们只能有一个 `yield` 块。我们可以向 `yield` 传递参数：

```
def simple(参数)
  puts "yield 之前"
  yield(参数)
```


Chapter 18: Methods

Functions in Ruby provide organized, reusable code to preform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, yield statements and scope in Ruby.

Section 18.1: Defining a method

Methods are defined with the `def` keyword, followed by the *method name* and an optional list of *parameter names* in parentheses. The Ruby code between `def` and `end` represents the *body* of the method.

```
def hello(name)
  "Hello, #{name}"
end
```

A method invocation specifies the method name, the object on which it is to be invoked (sometimes called the receiver), and zero or more argument values that are assigned to the named method parameters.

```
hello("World")
# => "Hello, World"
```

When the receiver is not explicit, it is `self`.

Parameter names can be used as variables within the method body, and the values of these named parameters come from the arguments to a method invocation.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

Section 18.2: Yielding to blocks

You can send a block to your method and it can call that block multiple times. This can be done by sending a proc/lambda or such, but is easier and faster with `yield`:

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "Finally we are here:  #{arg2}"
  yield
end
simple('start','end') { puts "Now we are inside the yield" }

#> First we are here:  start
#> Now we are inside the yield
#> Finally we are here:  end
#> Now we are inside the yield
```

Note that the `{ puts ... }` is not inside the parentheses, it implicitly comes after. This also means we can only have one `yield` block. We can pass arguments to the `yield`:

```
def simple(arg)
  puts "Before yield"
  yield(arg)
```

```
  puts "yield 之后"
end
simple('戴夫') { |名字| puts "我的名字是 #{名字}" }

#> yield 之前
#> 我的名字是 戴夫
#> yield 之后
```

通过 yield，我们可以轻松创建迭代器或任何作用于其他代码的函数：

```
def 倒计时(数字)
  数字.times do |计数|
    yield(数字-计数)
  end
end
```

```
倒计时(5) { |计数| puts "调用次数 #{计数}" }
```

```
#> 呼叫号码 5
#> 呼叫号码 4
#> 呼叫号码 3
#> 呼叫号码 2
#> 呼叫号码 1
```

事实上，正是通过yield，像foreach、each和times这样的功能通常在类中实现。

如果你想知道是否传入了代码块，可以使用block_given?：

```
class 员工
  def 名字
ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
ret.push(emp.name)
      end
    end

    ret
  end
end
```

这个例子假设Employees类有一个@employees列表，可以用each迭代，获取通过name方法获得员工名字的对象。
如果传入了代码块，我们就将名字yield给代码块，
否则我们将名字推入一个数组并返回。

第18.3节：默认参数

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

可以为多个参数设置默认值：


```
  puts "After yield"
end
simple('Dave') { |name| puts "My name is #{name}" }

#> Before yield
#> My name is Dave
#> After yield
```

With yield we can easily make iterators or any functions that work on other code:

```
def countdown(num)
  num.times do |i|
    yield(num-i)
  end
end
```

```
countdown(5) { |i| puts "Call number #{i}" }
```

```
#> Call number 5
#> Call number 4
#> Call number 3
#> Call number 2
#> Call number 1
```

In fact, it is with `yield` that things like `foreach`, `each` and `times` are generally implemented in classes.

If you want to find out if you have been given a block or not, use `block_given?`:

```
class Employees
  def names
    ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
        ret.push(emp.name)
      end
    end

    ret
  end
end
```

This example assumes that the `Employees` class has an `@employees` list that can be iterated with `each` to get objects that have employee names using the `name` method. If we are given a block, then we'll `yield` the name to the block, otherwise we just push it to an array that we return.

Section 18.3: Default parameters

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

It's possible to include defaults for multiple arguments:

```ruby
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap 牛
```

但是，不能只提供第二个参数而不提供第一个参数。与其使用位置参数，不如尝试使用关键字参数：

```ruby
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # 鸭子低语
```

或者使用存储选项的哈希参数：

```ruby
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

默认参数值可以由任何 Ruby 表达式设置。该表达式将在方法的上下文中运行，因此你甚至可以在这里声明局部变量。注意，这不会通过代码审查。感谢 caius 指出这一点。

```ruby
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (运行时错误)
```

第18.4节：可选参数（splat 操作符）

```ruby
def welcome_guests(*guests)
    guests.each { |guest| puts "欢迎 #{guest}!" }
end
```

```ruby
welcome_guests('Tom')     # 欢迎 Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # 欢迎 Rob!
                                        # 欢迎 Sally!
                                        # 欢迎 Lucas!
```

请注意，welcome_guests(['Rob', 'Sally', 'Lucas']) 将输出 Welcome ["Rob", "Sally", "Lucas"]！
相反，如果你有一个列表，你可以这样做`welcome_guests(*['Rob', 'Sally', 'Lucas'])`，这样就相当于`welcome_guests('Rob', 'Sally', 'Lucas')`。

第18.5节：必需的默认可选参数混合

```ruby
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
```



```ruby
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

However, it's not possible to supply the second without also supplying the first. Instead of using positional parameters, try keyword parameters:

```ruby
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

Or a hash parameter that stores options:

```ruby
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

Default parameter values can be set by any ruby expression. The expression will run in the context of the method, so you can even declare local variables here. Note, won't get through code review. Courtesy of caius for pointing this out.

```ruby
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (RuntimeError)
```

Section 18.4: Optional parameter(s) (splat operator)

```ruby
def welcome_guests(*guests)
    guests.each { |guest| puts "Welcome #{guest}!" }
end
```

```ruby
welcome_guests('Tom')     # Welcome Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # Welcome Rob!
                                        # Welcome Sally!
                                        # Welcome Lucas!
```

Note that `welcome_guests(['Rob', 'Sally', 'Lucas'])` will output `Welcome ["Rob", "Sally", "Lucas"]`! Instead, if you have a list, you can do `welcome_guests(*['Rob', 'Sally', 'Lucas'])` and that will work as `welcome_guests('Rob', 'Sally', 'Lucas')`.

Section 18.5: Required default optional parameter mix

```ruby
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
```

```
end
```

调用方式如下：

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

第18.6节：将函数作为块使用

Ruby中的许多函数接受块作为参数。例如：

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

如果你已经有一个实现所需功能的函数，可以使用&method(:fn)将其转换为块：

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

第18.7节：单个必需参数

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')     # Hello Charles
```

第18.8节：元组参数

方法可以接受一个数组参数，并立即将其解构为命名的局部变量。见于 Mathias Meyer的博客。

```
def feed( amount, (animal, food) )

    p "#{amount} #{animal}s 咀嚼一些 #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 只兔子咀嚼一些草"
```



```
end
```

Call as follows:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

Section 18.6: Use a function as a block

Many functions in Ruby accept a block as an argument. E.g.:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

If you already have a function that does what you want, you can turn it into a block using `&method(:fn)`:

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

Section 18.7: Single required parameter

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')    # Hello Charles
```

Section 18.8: Tuple Arguments

A method can take an array parameter and destructure it immediately into named local variables. Found on Mathias Meyer's blog.

```
def feed( amount, (animal, food) )

    p "#{amount} #{animal}s chew some #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 rabbits chew some grass"
```

第18.9节：捕获未声明的关键字参数（双星号）

** 操作符的作用类似于 * 操作符，但它适用于关键字参数。

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: '完成！', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

在上述示例中，如果不使用 **other_options，则会抛出 ArgumentError: unknown keyword: foo, bar 错误。

```
def without_double_splat(required_key:, optional_key: nil)
  # 什么也不做
end

without_double_splat(required_key: '完成！', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: 未知关键字: foo, bar
```

当你有一个选项哈希想传递给方法且不想过滤键时，这非常方便。

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

也可以使用 **操作符解包一个哈希。这允许你直接向方法提供关键字，
同时还可以传入其他哈希中的值：

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

第18.10节：多个必需参数

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('嗨', '索菲')    # 嗨 索菲
```

第18.11节：方法定义是表达式

在 Ruby 2.x 中定义方法会返回一个表示方法名的符号：

```
类 示例
```


Section 18.9: Capturing undeclared keyword arguments (double splat)

The ** operator works similarly to the * operator but it applies to keyword parameters.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

In the above example, if the `**other_options` is not used, an `ArgumentError: unknown keyword: foo, bar` error would be raised.

```
def without_double_splat(required_key:, optional_key: nil)
  # do nothing
end

without_double_splat(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: unknown keywords: foo, bar
```

This is handy when you have a hash of options that you want to pass to a method and you do not want to filter the keys.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

It is also possible to *unpack* a hash using the `**` operator. This allows you to supply keyword directly to a method in addition to values from other hashes:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Section 18.10: Multiple required parameters

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('Hi', 'Sophie')    # Hi Sophie
```

Section 18.11: Method Definitions are Expressions

Defining a method in Ruby 2.x returns a symbol representing the name:

```
class Example
```

```
  puts def hello
  end
end

#=> :hello
```

这使得有趣的元编程技术成为可能。例如，方法可以被其他
方法包裹：

```
类 类
  def logged(name)
original_method = instance_method(name)
    define_method(name) do |*args|
      puts "调用 #{name}，参数为 #{args.inspect}。"
     original_method.bind(self).call(*args)
      puts "完成 #{name}。"
    end
  end
end

class Meal
  def initialize
    @food = []
  end

logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# 调用 add，参数为 ["Coffee"]。
# 完成 add。
```



```
  puts def hello
  end
end

#=> :hello
```

This allows for interesting metaprogramming techniques. For instance, methods can be wrapped by other methods:

```
class Class
  def logged(name)
    original_method = instance_method(name)
    define_method(name) do |*args|
      puts "Calling #{name} with #{args.inspect}."
      original_method.bind(self).call(*args)
      puts "Completed #{name}."
    end
  end
end

class Meal
  def initialize
    @food = []
  end

  logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# Calling add with ["Coffee"].
# Completed add.
```

第19章：哈希（Hashes）

哈希是一种类似字典的集合，由唯一的键及其对应的值组成。也称为关联数组，它们类似于数组，但数组使用整数作为索引，而哈希允许使用任何对象类型。通过引用键，可以检索或创建哈希中的新条目。

第19.1节：创建哈希

Ruby中的哈希是实现了哈希表的对象，将键映射到值。Ruby支持使用{}的特定字面量语法来定义哈希：

```
my_hash = {}  # 一个空哈希
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

哈希值也可以使用标准的new方法创建：

```
my_hash = Hash.new  # 任意空哈希
my_hash = {}        # 任意空哈希
```

哈希的值可以是任何类型，包括数组、对象和其他哈希等复杂类型：

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

键也可以是任何类型，包括复杂类型：

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

符号通常用作哈希键，Ruby 1.9 引入了一种新语法专门用于简化此过程。以下哈希是等价的：

```
# 适用于所有 Ruby 版本
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# 适用于 Ruby 1.9 及以上版本
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

以下哈希（在所有 Ruby 版本中有效）不同，因为所有键都是字符串：

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

虽然两种语法版本可以混用，但不推荐以下写法。

```
mapping = { :length => 45, width: 10 }
```

在 Ruby 2.2 及以上版本中，有一种用于创建符号键哈希的替代语法（如果符号
包含空格，这种语法最有用）：

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```


Chapter 19: Hashes

A Hash is a dictionary-like collection of unique keys and their values. Also called associative arrays, they are similar to Arrays, but where an Array uses integers as its index, a Hash allows you to use any object type. You retrieve or create a new entry in a Hash by referring to its key.

Section 19.1: Creating a hash

A hash in Ruby is an object that implements a hash table, mapping keys to values. Ruby supports a specific literal syntax for defining hashes using {}:

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

A hash can also be created using the standard `new` method:

```
my_hash = Hash.new  # any empty hash
my_hash = {}        # any empty hash
```

Hashes can have values of any type, including complex types like arrays, objects and other hashes:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

Also keys can be of any type, including complex ones:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

Symbols are commonly used as hash keys, and Ruby 1.9 introduced a new syntax specifically to shorten this process. The following hashes are equivalent:

```
# Valid on all Ruby versions
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# Valid in Ruby version 1.9+
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

The following hash (valid in all Ruby versions) is *different*, because all keys are strings:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

While both syntax versions can be mixed, the following is discouraged.

```
mapping = { :length => 45, width: 10 }
```

With Ruby 2.2+, there is an alternative syntax for creating a hash with symbol keys (most useful if the symbol contains spaces):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

第19.2节：设置默认值

默认情况下，尝试查找不存在的键的值将返回 nil。你可以选择指定当哈希被访问不存在的键时返回的其他值（或执行的操作）。虽然这被称为"默认值"，但它不必是单一的值；例如，它可以是一个计算值，比如键的长度。

哈希的默认值可以传递给其构造函数：

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 返回默认值而非 nil
```

也可以在已构造的 Hash 上指定默认值：

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

需要注意的是，**默认值不会在每次访问新键时被复制**，这在默认值是引用类型时可能导致意想不到的结果：

```
# 使用空数组作为默认值
authors = Hash.new([])

# 添加书名
authors[:homer] << '奥德赛'

# 所有新键都映射到同一个数组的引用：
authors[:plato] # => ['奥德赛']
```

为了解决这个问题，Hash 构造函数接受一个块，该块在每次访问新键时执行，返回值被用作默认值：

```
authors = Hash.new { [] }

# 注意这里我们使用 += 而不是 <<，见下文
authors[:homer] += ['奥德赛']
authors[:plato] # => []

authors # => {:homer=>["奥德赛"]}
```

注意上面我们必须使用 += 而不是 <<，因为默认值不会自动赋给哈希；使用 << 会添加到数组，但 authors[:homer] 仍然是未定义的：

```
authors[:homer] << '奥德赛' # ['奥德赛']
authors[:homer] # => []
authors # => {}
```

为了能够在访问时赋予默认值，以及计算更复杂的默认值，默认块会同时传入哈希和键：

```
authors = Hash.new { |hash, key| hash[key] = [] }
```


Section 19.2: Setting Default Values

By default, attempting to lookup the value for a key which does not exist will return **`nil`**. You can optionally specify some other value to return (or an action to take) when the hash is accessed with a non-existent key. Although this is referred to as "the default value", it need not be a single value; it could, for example, be a computed value such as the length of the key.

The default value of a hash can be passed to its constructor:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

A default can also be specified on an already constructed Hash:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

It is important to note that the **default value is not copied** each time a new key is accessed, which can lead to surprising results when the default value is a reference type:

```
# Use an empty array as the default value
authors = Hash.new([])

# Append a book title
authors[:homer] << 'The Odyssey'

# All new keys map to a reference to the same array:
authors[:plato] # => ['The Odyssey']
```

To circumvent this problem, the Hash constructor accepts a block which is executed each time a new key is accessed, and the returned value is used as the default:

```
authors = Hash.new { [] }

# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Note that above we had to use += instead of << because the default value is not automatically assigned to the hash; using << would have added to the array, but authors[:homer] would have remained undefined:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

In order to be able to assign default values on access, as well as to compute more sophisticated defaults, the default block is passed both the hash and the key:

```
authors = Hash.new { |hash, key| hash[key] = [] }
```

```
authors[:homer] << '奥德赛'
authors[:柏拉图] # => []

authors # => {:homer=>["奥德赛"], :plato=>[]}
```

你也可以使用默认块来根据键（或其他数据）执行操作和/或返回值：

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

你甚至可以创建更复杂的哈希：

```
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

为了将默认值设置为已存在哈希的Proc，使用default_proc=：

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['奥德赛']
authors[:plato] # => []

作者 # {:homer=>["奥德赛"]}
```

第19.3节：访问值

哈希的单个值通过[]和[]=方法进行读取和写入：

```
my_hash = { 长度: 4, 宽度: 5 }

my_hash[:长度] #=> => 4

my_hash[:高度] = 9

my_hash #=> {:长度 => 4, :宽度 => 5, :高度 => 9 }
```

默认情况下，访问尚未添加到哈希中的键会返回nil，意味着尝试查找键的值总是安全的：

```
my_hash = {}

my_hash[:年龄] # => nil
```

哈希也可以包含字符串形式的键。如果你尝试正常访问它们，只会返回nil，正确的做法是通过它们的字符串键访问：

```
my_hash = { "名字" => "用户" }

my_hash[:名字]     # => nil
my_hash["名字"]    # => 用户
```

对于预期或要求键存在的情况，哈希具有一个fetch方法，当访问不存在的键时会引发异常：


```
authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

You can also use a default block to take an action and/or return a value dependent on the key (or some other data):

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

You can even create more complex hashes:

```
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

In order to set the default to a Proc on an *already-existing* hash, use `default_proc=`:

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

Section 19.3: Accessing Values

Individual values of a hash are read and written using the [ ] and `[]=` methods:

```
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4

my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

By default, accessing a key which has not been added to the hash returns `nil`, meaning it is always safe to attempt to look up a key's value:

```
my_hash = {}

my_hash[:age] # => nil
```

Hashes can also contain keys in strings. If you try to access them normally it will just return a `nil`, instead you access them by their string keys:

```
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

For situations where keys are expected or required to exist, hashes have a `fetch` method which will raise an exception when accessing a key that does not exist:

```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

fetch 方法接受一个默认值作为第二个参数，如果键之前未设置，则返回该默认值：

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

fetch 方法也可以接受一个块，如果键之前未设置，则返回该块的值：

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

哈希（Hashes）也支持store方法，作为[]=的别名：

```
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

你也可以使用values方法获取哈希的所有值：

```
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

注意：这仅适用于 Ruby 2.3+ 版本。#dig 对嵌套的Hash非常有用。通过依次调用 dig，按 idx 对象序列提取嵌套值，如果中间任何一步为 nil，则返回 nil。

```
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

第19.4节：自动创建深层哈希

Hash 对请求但不存在的键有默认值（nil）：

```
a = {}
p a[ :b ] # => nil
```

创建新的 Hash 时，可以指定默认值：

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```



```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` accepts a default value as its second argument, which is returned if the key has not been previously set:

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` can also accept a block which is returned if the key has not been previously set:

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Hashes also support a `store` method as an alias for `[]=`:

```
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

You can also get all values of a hash using the `values` method:

```
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

Note: This is only for Ruby 2.3+ `#dig` is handy for nested `Hash`s. Extracts the nested value specified by the sequence of idx objects by calling dig at each step, returning nil if any intermediate step is nil.

```
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

Section 19.4: Automatically creating a Deep Hash

Hash has a default value for keys that are requested but don't exist (nil):

```
a = {}
p a[ :b ] # => nil
```

When creating a new Hash, one can specify the default:

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Hash.new 也接受一个块，这允许你自动创建嵌套的哈希，类似于 Perl 的自动生成功能行为或 `mkdir -p`：

```
# h 是你正在创建的哈希，k 是键。
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

第19.5节：遍历哈希

Hash 包含了 Enumerable 模块，该模块提供了多种迭代方法，如：Enumerable#each、**Enumerable#each_pair、Enumerable#each_key 和 Enumerable#each_value。**

.each 和 .each_pair 遍历每个键值对：

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

.each_key 仅遍历键：

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

.each_value 仅遍历值：

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

.each_with_index 遍历元素并提供迭代的索引：

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```


Hash.new also takes a block, which allows you to automatically create nested hashes, such as Perl's autovivification behavior or `mkdir -p`:

```
# h is the hash you're creating, and k the key.
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

Section 19.5: Iterating Over a Hash

A **Hash** includes the **Enumerable** module, which provides several iteration methods, such as: **Enumerable***#each*, **Enumerable***#each_pair*, **Enumerable***#each_key*, and **Enumerable***#each_value*.

.each and .each_pair iterate over each key-value pair:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

.each_key iterates over the keys only:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

.each_value iterates over the values only:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

.each_with_index iterates over the elements and provides the index of the iteration:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```

第19.6节：过滤哈希

SELECT 返回一个新的哈希，包含块计算结果为 true 的键值对。

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

当你在过滤块中不需要使用 key 或 value 时，惯例是在该位置使用 _：

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

reject 返回一个新的哈希，包含块计算结果为 false 的键值对：

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

第19.7节：与数组的相互转换

哈希可以自由地转换为数组，也可以从数组转换回来。将一个键/值对的哈希转换成数组时，会生成一个包含嵌套数组的数组，每个嵌套数组表示一对键值：

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

反过来，也可以从相同格式的数组创建一个哈希：

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

同样，哈希可以使用Hash[]和交替的键值列表来初始化：

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

或者从每个包含两个值的数组数组中初始化：

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

哈希可以使用flatten()方法转换回交替的键和值组成的数组：

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

轻松地在数组之间转换使得Hash能够很好地配合许多Enumerable方法，例如collect和zip：

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }

people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

第19.8节：重写哈希函数

Ruby哈希使用方法 hash 和 eql? 来执行哈希操作，并将存储在哈希中的对象分配到内部哈希桶中。Ruby中 hash 的默认实现是对被哈希对象的所有成员字段使用 murmur哈希函数。要重写此行为，可以重写 hash 和 eql? 方法。

Section 19.6: Filtering hashes

SELECT returns a new hash with key-value pairs for which the block evaluates to true.

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

When you will not need the *key* or *value* in a filter block, the convention is to use an _ in that place:

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

reject returns a new hash with key-value pairs for which the block evaluates to false:

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

Section 19.7: Conversion to and from Arrays

Hashes can be freely converted to and from arrays. Converting a hash of key/value pairs into an array will produce an array containing nested arrays for pair:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

In the opposite direction a Hash can be created from an array of the same format:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

Similarly, Hashes can be initialized using Hash[] and a list of alternating keys and values:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

Or from an array of arrays with two values each:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Hashes can be converted back to an Array of alternating keys and values using flatten():

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

The easy conversion to and from an array allows Hash to work well with many Enumerable methods such as collect and zip:

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }

people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

Section 19.8: Overriding hash function

Ruby hashes use the methods hash and eql? to perform the hash operation and assign objects stored in the hash to internal hash bins. The default implementation of hash in Ruby is the murmur hash function over all member fields of the hashed object. To override this behavior it is possible to override hash and eql? methods.

与其他哈希实现一样，如果两个对象a和b满足 a.hash == b.hash 并且 a.eql?(b)为真，则它们会被哈希到同一个桶中并被视为相同。因此，在重新实现 hash 和 eql? 时，应确保如果 a 和 b 在 eql? 下相等，则它们必须返回相同的 hash 值。否则，这可能导致哈希中出现重复条目。反之， hash 实现选择不当可能导致许多对象共享同一个哈希桶，实际上破坏了O(1)的查找时间，调用 eql? 时变成对所有对象的O(n)操作。

下面的示例中，只有类 A 的实例被存储为键，因为它是第一个被添加的：

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # 返回外部给定的值
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

第19.9节：获取哈希的所有键或值

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

第19.10节：修改键和值

您可以创建一个新的哈希，修改键或值，实际上您也可以添加或删除键，使用inject（也称为reduce）。例如，生成一个键为字符串形式且值为大写的哈希：

```
fruit = { name: '苹果', color: '绿色', shape: '圆形' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit 是 {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

哈希（Hash）本质上是一个可枚举的键/值对集合。因此它具有诸如 each、map 和 inject 等方法。

对于哈希中的每一个键/值对，都会执行给定的代码块，第一次运行时 memo 的值是传递给 inject 的初始值，在我们的例子中是一个空哈希 {}。

As with other hash implementations, two objects a and b, will be hashed to the same bucket if `a.hash == b.hash` and will be deemed identical if `a.eql?(b)`. Thus, when reimplementing `hash` and `eql?` one should take care to ensure that if `a` and `b` are equal under `eql?` they must return the same `hash` value. Otherwise this might result in duplicate entries in a hash. Conversely, a poor choice in `hash` implementation might lead many objects to share the same hash bucket, effectively destroying the O(1) look-up time and causing O(n) for calling `eql?` on all objects.

In the example below only the instance of class `A` is stored as a key, as it was added first:

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # Return the value given externally
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

Section 19.9: Getting all keys or values of hash

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

Section 19.10: Modifying keys and values

You can create a new hash with the keys or values modified, indeed you can also add or delete keys, using inject (AKA, reduce). For example to produce a hash with stringified keys and upper case values:

```
fruit = { name: 'apple', color: 'green', shape: 'round' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit is {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

Hash is an enumerable, in essence a collection of key/value pairs. Therefore is has methods such as `each`, `map` and `inject`.

For every key/value pair in the hash the given block is evaluated, the value of memo on the first run is the seed

后续执行时，memo 的值是前一次代码块执行的返回值，这就是为什么我们通过设置键和值来修改 memo，然后在最后返回 memo。最终代码块执行的返回值就是 inject 的返回值，在我们的例子中是

memo。

为了避免必须提供最终值，你可以改用 each_with_object：

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

或者甚至使用 map：

版本 ≥ 1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

（详见this answer，了解更多细节，包括如何原地操作哈希。）

第19.11节：哈希的集合操作

- **哈希的交集**

  要获取两个哈希的交集，返回共享的键且其对应的值相等：

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 2, :c => 3 }
  hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
  ```

- **哈希的并集（合并）：**

  哈希中的键是唯一的，如果要合并的两个哈希中都存在同一个键，则以来自那个哈希的键为准调用 merge 时被覆盖：

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 4, :c => 3 }

  hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
  hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
  ```


value passed to `inject`, in our case an empty hash, `{}`. The value of `memo` for subsequent evaluations is the returned value of the previous blocks evaluation, this is why we modify `memo` by setting a key with a value and then return `memo` at the end. The return value of the final blocks evaluation is the return value of `inject`, in our case `memo`.

To avoid the having to provide the final value, you could use each_with_object instead:

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

Or even map:

Version ≥ 1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(See this answer for more details, including how to manipulate hashes in place.)

Section 19.11: Set Operations on Hashes

- **Intersection of Hashes**

  To get the intersection of two hashes, return the shared keys the values of which are equal:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 2, :c => 3 }
  hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
  ```

- **Union (merge) of hashes:**

  keys in a hash are unique, if a key occurs in both hashes which are to be merged, the one from the hash that `merge` is called on is overwritten:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 4, :c => 3 }

  hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
  hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
  ```

第20章：块、Proc和Lambda

第20.1节：Lambda

```
# 使用箭头语法的lambda
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# 使用箭头语法的lambda，接受1个参数
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

You can also use -> to create and .() to call lambda

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 表示 hello
```

这里你可以看到 lambda 几乎和 proc 一样。然而，有几个注意事项：

- lambda 的参数个数是强制检查的；传递错误数量的参数给 lambda 会引发 `ArgumentError`。它们仍然可以有默认参数、splat 参数等。
- **从 lambda 内部 return 会返回 lambda，而从 proc 内部 return 会返回到** 外层作用域：

```
def try_proc
x = Proc.new {
    return # 从 try_proc 返回
  }
```


Chapter 20: Blocks and Procs and Lambdas

Section 20.1: Lambdas

```
# lambda using the arrow syntax
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# lambda using the arrow syntax accepting 1 argument
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

You can also use `->` to create and `.()` to call lambda

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 means hello
```

Here you can see that a lambda is almost the same as a proc. However, there are several caveats:

- The arity of a lambda's arguments are enforced; passing the wrong number of arguments to a lambda, will raise an `ArgumentError`. They can still have default parameters, splat parameters, etc.
- `return`ing from within a lambda returns from the lambda, while `return`ing from a proc returns out of the enclosing scope:

```
def try_proc
  x = Proc.new {
    return # Return from try_proc
  }
```

```
x.call
  puts "After x.call" # 这行永远不会被执行
end

def try_lambda
  y = -> {
    return # 从 y 返回
  }
y.调用
  puts "调用 y 之后" # 这行不会被跳过
end

try_proc # 无输出
t try_lambda # 输出 "调用 y 之后"
```

第20.2节：部分应用与柯里化

严格来说，Ruby 没有函数，只有方法。然而，Ruby 方法的行为几乎与其他语言中的函数完全相同：

```
def double(n)
  n * 2
end
```

这个普通的方法/函数接受一个参数 n，将其乘以二并返回结果。现在我们定义一个高阶函数（或方法）：

```
def triple(n)
  lambda {3 * n}
end
```

triple 不返回数字，而是返回一个方法。你可以使用交互式 Ruby 解释器进行测试：

```
$ irb --简单-快速
>> 定义 double(n)
>>   n * 2
>> 结束
=> :double
>> 定义 triple(n)
>>   lambda {3 * n}
>> 结束
=> :triple
>> double(2)
=> 4
>>  triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

如果你想真正得到三倍的数字，需要调用（或“执行”）这个lambda：

```
triple_two = triple(2)
triple_two.call # => 6
```

或者更简洁地写成：

```
triple(2).call
```



```
  x.call
  puts "After x.call" # this line is never reached
end

def try_lambda
  y = -> {
    return # return from y
  }
  y.call
  puts "After y.call" # this line is not skipped
end

try_proc # No output
try_lambda # Outputs "After y.call"
```

Section 20.2: Partial Application and Currying

Technically, Ruby doesn't have functions, but methods. However, a Ruby method behaves almost identically to functions in other language:

```
def double(n)
  n * 2
end
```

This normal method/function takes a parameter `n`, doubles it and returns the value. Now let's define a higher order function (or method):

```
def triple(n)
  lambda {3 * n}
end
```

Instead of returning a number, `triple` returns a method. You can test it using the Interactive Ruby Shell:

```
$ irb --simple-prompt
>> def double(n)
>>   n * 2
>> end
=> :double
>> def triple(n)
>>   lambda {3 * n}
>> end
=> :triple
>> double(2)
=> 4
>> triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

If you want to actually get the tripled number, you need to call (or "reduce") the lambda:

```
triple_two = triple(2)
triple_two.call # => 6
```

Or more concisely:

```
triple(2).call
```

柯里化与部分应用

这在定义非常基础的功能时并不实用，但如果你想拥有不会立即调用或简化的方法/函数时，它就很有用。例如，假设你想定义一些方法，通过特定数字来加一个数字（例如add_one(2)=3）。如果你必须定义大量这样的函数，你可以这样做：

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

然而，你也可以这样做：

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

使用λ演算，我们可以说add是(λa.(λb.(a+b)))。柯里化是一种对add进行部分应用的方法。所以add.curry.(1)是(λa.(λb.(a+b)))(1)，可以化简为(λb.(1+b))。部分应用意味着我们传递了一个参数给add，但另一个参数留待以后提供。输出是一个专门化的方法。

柯里化的更多实用示例

假设我们有一个非常大的通用公式，如果我们为它指定某些参数，就可以从中得到特定的公式。考虑这个公式：

```
f(x, y, z) = sin(x\\*y)*sin(y\\*z)*sin(z\\*x)
```

该公式是为三维空间工作而设计的，但假设我们只想要关于 y 和 z 的公式。假设为了忽略 x，我们想将其值设为 π/2。首先让我们写出通用公式：

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

现在，让我们使用柯里化来得到我们的yz公式：

```
f_yz = f.curry.(Math::PI/2)
```

然后调用存储在f_yz中的lambda函数：

```
f_xy.call(some_value_x, some_value_y)
```

这很简单，但假设我们想得到xz的公式。如果y不是最后一个参数，如何将其设置为Math::PI/2呢？这就有点复杂了：

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

在这种情况下，我们需要为未预先填充的参数提供占位符。为了保持一致，我们可以这样写f_xy像这样：

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```


Currying and Partial Applications

This is not useful in terms of defining very basic functionality, but it is useful if you want to have methods/functions that are not instantly called or reduced. For example, let's say you want to define methods that add a number by a specific number (for example `add_one(2) = 3`). If you had to define a ton of these you could do:

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

However, you could also do this:

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

Using lambda calculus we can say that `add` is `(λa.(λb.(a+b)))`. Currying is a way of *partially applying* `add`. So `add.curry.(1)`, is `(λa.(λb.(a+b)))(1)` which can be reduced to `(λb.(1+b))`. Partial application means that we passed one argument to `add` but left the other argument to be supplied later. The output is a specialized method.

More useful examples of currying

Let's say we have really big general formula, that if we specify certain arguments to it, we can get specific formulae from it. Consider this formula:

```
f(x, y, z) = sin(x\\*y)*sin(y\\*z)*sin(z\\*x)
```

This formula is made for working in three dimensions, but let's say we only want this formula with regards to y and z. Let's also say that to ignore x, we want to set it's value to pi/2. Let's first make the general formula:

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

Now, let's use currying to get our yz formula:

```
f_yz = f.curry.(Math::PI/2)
```

Then to call the lambda stored in `f_yz`:

```
f_xy.call(some_value_x, some_value_y)
```

This is pretty simple, but let's say we want to get the formula for `xz`. How can we set `y` to `Math::PI/2` if it's not the last argument? Well, it's a bit more complicated:

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

In this case, we need to provide placeholders for the parameter we aren't pre-filling. For consistency we could write `f_xy` like this:

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```

以下是f_yz的lambda演算工作原理：

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # 化简 =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

现在让我们来看一下f_xz

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # 化简 =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t)))))
```

想了解更多关于lambda演算的内容，请试试this。

第20.3节：将对象作为方法的块参数

在参数前加上&（和号）将其作为方法的块传递。对象将通过to_proc方法转换为Proc。

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

这是 Ruby 中的一个常见模式，许多标准类都提供了它。

例如，Symbol类通过将自身发送给参数来实现to_proc方法：

```
# 示例实现
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

这使得有用的&:symbol惯用法成为可能，通常用于Enumerable对象：

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

第20.4节：转换为 Proc

响应 to_proc 的对象可以通过 & 操作符转换为 proc（这也允许它们作为块传递）。

类 Symbol 定义了 #to_proc，因此它会尝试调用接收对象作为参数的对应方法。

Here's how the lambda calculus works for `f_yz`:

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # Reduce =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

Now let's look at `f_xz`

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # Reduce =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t)))))
```

For more reading about lambda calculus try this.

Section 20.3: Objects as block arguments to methods

Putting a **&** (ampersand) in front of an argument will pass it as the method's block. Objects will be converted to a **Proc** using the `to_proc` method.

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

This is a common pattern in Ruby and many standard classes provide it.

For example, **Symbol**s implement `to_proc` by sending themselves to the argument:

```
# Example implementation
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

This enables the useful **&**`:symbol` idiom, commonly used with **Enumerable** objects:

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

Section 20.4: Converting to Proc

Objects that respond to `to_proc` can be converted to procs with the **&** operator (which will also allow them to be passed as blocks).

The class Symbol defines *`#to_proc`* so it tries to call the corresponding method on the object it receives as parameter.

```
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

方法对象也定义了 #to_proc。

```
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbitgrass"
```

第20.5节：块

块是用大括号 {}（通常用于单行块）或 do..end（用于多行块）包围的代码片段。

```
5.times { puts "Hello world" } # 推荐的单行块写法

5.times do
    打印"Hello "
    puts "world"
end   # 多行代码块推荐的风格

5.次 {
    print "hello "
    puts "world" } # 不会抛出错误，但不推荐
```

注意：大括号的优先级高于 do..end

Yielding

代码块可以在方法和函数中使用 yield 关键字调用：

```
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "我自己的代码块" } # 代码块作为参数传递给方法。
#some code
#My own block
#other code
```

不过要小心，如果yield在没有块的情况下调用，会引发LocalJumpError。为此，Ruby提供了另一个方法block_given?，它允许你在调用yield之前检查是否传入了块

```
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
```



```
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

Method objects also define `#to_proc`.

```
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbit\ngrass"
```

Section 20.5: Blocks

Blocks are chunks of code enclosed between braces {} (usually for single-line blocks) or `do..end` (used for multi-line blocks).

```
5.times { puts "Hello world" } # recommended style for single line blocks

5.times do
    print "Hello "
    puts "world"
end   # recommended style for multi-line blocks

5.times {
    print "hello "
    puts "world" } # does not throw an error but is not recommended
```

Note: braces have higher precedence than `do..end`

Yielding

Blocks can be used inside methods and functions using the word `yield`:

```
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "My own block" } # the block is passed as an argument to the method.
#some code
#My own block
#other code
```

Be careful though if `yield` is called without a block it will raise a `LocalJumpError`. For this purpose ruby provides another method called `block_given?` this allows you to check if a block was passed before calling yield

```
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
```

```
# 未违约
# 其他代码
```

yield 也可以向代码块传递参数

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

虽然这是一个简单的例子，yield 在允许直接访问实例变量或在另一个对象的上下文中进行求值时非常有用。例如：

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# 配置
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

正如你所见，以这种方式使用`yield`使代码比不断调用`app.configuration.#method_name`更易读。你可以在块内执行所有配置，使代码保持封装。

变量

块的变量是块的局部变量（类似于函数的变量），块执行完后变量就消失了。

```
my_variable = 8
3.次 执行 |x|
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

代码块不能被保存，执行后即消失。要保存代码块，你需要使用procs和lambdas。


```
# not defaulted
# other code
```

`yield` can offer arguments to the block as well

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

While this is a simple example `yield`ing can be very useful for allowing direct access to instance variables or evaluations inside the context of another object. For Example:

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# Configuration
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

As you can see using `yield` in this manner makes the code more readable than continually calling `app.configuration.#method_name`. Instead you can perform all the configuration inside the block keeping the code contained.

Variables

Variables for blocks are local to the block (similar to the variables of functions), they die when the block is executed.

```
my_variable = 8
3.times do |x|
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

Blocks can't be saved, they die once executed. In order to save blocks you need to use `procs` and `lambdas`.

第21章：迭代

第21.1节：each

Ruby有许多类型的枚举器，但最简单且最先接触的枚举器类型是each。我们将打印even或odd，针对1到10之间的每个数字，来展示each的工作原理。

基本上，有两种方式传递所谓的blocks。一个block是一段代码，它会被调用该方法时执行。each方法接受一个block，对它调用的对象集合中的每个元素都会执行该block。

传递代码块给方法有两种方式：

方法1：内联

(1..10).each { |i| puts i.even? ? 'even' : 'odd' }

这是一个非常简洁且符合ruby风格的解决方案。让我们逐步拆解这段代码。

1. (1..10)是一个包含1到10的区间。如果我们想要1到10不包含10，则写作(1...10)。
2. .each 是一个枚举器，用于枚举它所作用对象中的每个元素。在本例中，它作用于范围内的每个数字。
3. **{ |i| puts i.even? ? 'even' : 'odd' } 是 .each 语句的代码块，该代码块本身还可以进一步拆分。**进一步拆分。
    1. **|i| 表示范围内的每个元素在代码块中由标识符 i 代表。**
    2. **puts 是 Ruby 中的一个输出方法，每次打印后会自动换行。（如果不想自动换行，可以使用 print）**use print if we don't want the automatic line break)
    3. i.even? 检查 i 是否为偶数。我们也可以使用 i % 2 == 0；但更推荐使用内置方法中。
    4. ? “even” : “odd” 这是 Ruby 的三元运算符。三元运算符的构造方式是表达式 ? a : b。这是 if 表达式 a els

e b end 的简写

对于多于一行的代码，块 应该作为 多行块 传递。

方法二：多行

(1..10).each do |i| if i.even? puts 'even' else puts 'odd' end end

在 多行块 中，do 替代了开括号，end 替代了 内联

样式的闭括号。

Ruby 也支持 reverse_each。它会反向迭代数组。

```
@arr = [1,2,3,4]
puts @arr.inspect # 输出是 [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # 输出是 4 3 2 1
end
print "]"
```


Chapter 21: Iteration

Section 21.1: Each

Ruby has many types of enumerators but the first and most simple type of enumerator to start with is `each`. We will print out `even` or `odd` for each number between `1` and `10` to show how `each` works.

Basically there are two ways to pass so called `blocks`. A `block` is a piece of code being passed which will be executed by the method which is called. The `each` method takes a `block` which it calls for every element of the collection of objects it was called on.

There are two ways to pass a block to a method:

Method 1: Inline

(1..10).each { |i| puts i.even? ? 'even' : 'odd' }

This is a very compressed and *ruby* way to solve this. Let's break this down piece by piece.

1. `(1..10)` is a range from `1` to `10` inclusive. If we wanted it to be `1` to `10` exclusive, we would write `(1...10)`.
2. `.each` is an enumerator that enumerates over `each` element in the object it is acting on. In this case, it acts on each number in the range.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` is the block for the `each` statement, which itself can be broken down further.
    1. `|i|` this means that each element in the range is represented within the block by the identifier `i`.
    2. `puts` is an output method in Ruby that has an automatic line break after each time it prints. (We can use `print` if we don't want the automatic line break)
    3. `i.even?` checks if `i` is even. We could have also used `i % 2 == 0`; however, it is preferable to use built in methods.
    4. `? "even" : "odd"` this is ruby's ternary operator. The way a ternary operator is constructed is `expression ? a : b`. This is short for

if expression a else b end

For code longer than one line the `block` should be passed as a `multiline block`.

Method 2: Multiline

(1..10).each do |i| if i.even? puts 'even' else puts 'odd' end end

In a `multiline block` the `do` replaces the opening bracket and `end` replaces the closing bracket from the `inline` style.

Ruby supports reverse_each as well. It will iterate the array backwards.

```
@arr = [1,2,3,4]
puts @arr.inspect # output is [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # output is 4 3 2 1
end
print "]\n"
```

第21.2节：类中的实现

`Enumerable` 是 Ruby 中最流行的模块。它的目的是为你提供可迭代的方法，如 `map`、`SELECT`、`reduce` 等。使用 `Enumerable` 的类包括 `Array`、`Hash`、`Range`。要使用它，你必须 `include Enumerable` 并实现 `each`。

```
class 自然数
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = 自然数.new(6)

n.reduce(:+)                  # => 21
n.select(&:even?)             # => [0, 2, 4, 6]
n.map { |数字| 数字 ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

第21.3节：遍历复杂对象

数组

你可以遍历嵌套数组：

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

以下语法也被允许：

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

将产生：

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

哈希

你可以遍历键值对：

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

将产生：

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

你可以同时遍历键和值：

Section 21.2: Implementation in a class

`Enumerable` is the most popular module in Ruby. Its purpose is to provide you with iterable methods like `map`, `SELECT`, `reduce`, etc. Classes that use `Enumerable` include `Array`, `Hash`, `Range`. To use it, you have to `include Enumerable` and implement `each`.

```
class NaturalNumbers
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = NaturalNumbers.new(6)

n.reduce(:+)                  # => 21
n.select(&:even?)             # => [0, 2, 4, 6]
n.map { |number| number ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

Section 21.3: Iterating over complex objects

Arrays

You can iterate over nested arrays:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

The following syntax is allowed too:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Will produce:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

Hashes

You can iterate over key-value pairs:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Will produce:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

You can iterate over keys and values simultaneously:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

将产生：

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

第21.4节：for迭代器

这将从4迭代到13（包含13）。

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

我们也可以使用 for 来遍历数组

```
names = ['西瓦', '查兰', '纳雷什', '马尼什']

for name in names
    puts name
end
```

第21.5节：带索引的迭代

有时你想在遍历枚举器时知道当前元素的位置（索引）。为此，Ruby提供了with_index方法。它可以应用于所有枚举器。基本上，通过在枚举中添加with_index，你可以对该枚举进行带索引的遍历。索引作为第二个参数传递给代码块。

```
[2,3,4].map.with_index { |e, i| puts "数组元素编号 #{i} => #{e}" }
#数组元素编号 0 => 2
#数组元素编号 1 => 3
#数组元素编号 2 => 4
#=> [nil, nil, nil]
```

with_index有一个可选参数–第一个索引，默认值为0：

```
[2,3,4].map.with_index(1) { |e, i| puts "数组元素编号 #{i} => #{e}" }
#数组元素编号 1 => 2
#数组元素编号 2 => 3
#数组元素编号 3 => 4
#=> [nil, nil, nil]
```

有一个特定的方法 each_with_index。它与 each.with_index的唯一区别是你不能传递参数给它，因此第一个索引始终是0。

```
[2,3,4].each_with_index { |e, i| puts "数组元素编号 #{i} => #{e}" }
#数组元素编号 0 => 2
#数组元素编号 1 => 3
#数组元素编号 2 => 4
```



```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Will produce:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

Section 21.4: For iterator

This iterates from 4 to 13 (inclusive).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

We can also iterate over arrays using for

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

Section 21.5: Iteration with index

Sometimes you want to know the position (**index**) of the current element while iterating over an enumerator. For such purpose, Ruby provides the `with_index` method. It can be applied to all the enumerators. Basically, by adding `with_index` to an enumeration, you can enumerate that enumeration. Index is passed to a block as the second argument.

```
[2,3,4].map.with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [nil, nil, nil]
```

`with_index` has an optional argument – the first index which is `0` by default:

```
[2,3,4].map.with_index(1) { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 1 => 2
#Element of array number 2 => 3
#Element of array number 3 => 4
#=> [nil, nil, nil]
```

There is a specific method `each_with_index`. The only difference between it and `each.with_index` is that you can't pass an argument to that, so the first index is `0` all the time.

```
[2,3,4].each_with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
```

```
#=> [2, 3, 4]
```

第21.6节：Map（映射）

返回修改后的对象，但原始对象保持不变。例如：

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

map! 会修改原始对象：

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

注意：你也可以使用 collect 来实现相同的功能。


```
#=> [2, 3, 4]
```

Section 21.6: Map

Returns the changed object, but the original object remains as it was. For example:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` changes the original object:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Note: you can also use `collect` to do the same thing.

第22章：异常

第22.1节：创建自定义异常类型

自定义异常是任何继承Exception或Exception子类的类。

通常，你应该始终继承StandardError或其子类。Exception系列通常用于虚拟机或系统错误，捕获它们可能会阻止预期的强制中断。

```
# 定义一个名为 FileNotFound 的新自定义异常
class FileNotFound < StandardError
end

def read_file(path)
  File.exist?(path) || raise(FileNotFound, "文件 #{path} 未找到")
  File.read(path)
end

read_file("missing.txt")  #=> 抛出 FileNotFound.new("文件 `missing.txt` 未找到")
read_file("valid.txt")    #=> 读取并返回文件内容
```

通常给异常命名时会在末尾加上Error后缀：

- ConnectionError
- DontPanicError

但是，当错误本身已经说明问题时，你不需要添加Error后缀，因为那样会显得多余：

- FileNotFound 与 FileNotFoundError
- DatabaseExploded 与 DatabaseExplodedError

第22.2节：处理多个异常

你可以在同一个rescue声明中处理多个错误：

```
begin
  # 可能失败的执行
rescue FirstError, SecondError => e
  # 如果发生FirstError或SecondError时执行某些操作
end
```

你也可以添加多个rescue声明：

```
begin
  # 可能失败的执行
rescue FirstError => e
  # 如果发生FirstError时执行某些操作
rescue SecondError => e
  # 如果发生SecondError时执行某些操作
rescue => e
  # 如果发生 StandardError，则执行某些操作
end
```

rescue 块的顺序很重要：第一个匹配的块会被执行。因此，如果你将 StandardError 作为第一个条件，并且所有异常都继承自 StandardError，那么其他的 rescue 语句将永远不会被执行。

Chapter 22: Exceptions

Section 22.1: Creating a custom exception type

A custom exception is any class that extends `Exception` or a subclass of `Exception`.

In general, you should always extend `StandardError` or a descendant. The `Exception` family are usually for virtual-machine or system errors, rescuing them can prevent a forced interruption from working as expected.

```
# Defines a new custom exception called FileNotFound
class FileNotFound < StandardError
end

def read_file(path)
  File.exist?(path) || raise(FileNotFound, "File #{path} not found")
  File.read(path)
end

read_file("missing.txt")  #=> raises FileNotFound.new("File `missing.txt` not found")
read_file("valid.txt")    #=> reads and returns the content of the file
```

It's common to name exceptions by adding the `Error` suffix at the end:

- `ConnectionError`
- `DontPanicError`

However, when the error is self-explanatory, you don't need to add the `Error` suffix because would be redundant:

- `FileNotFound` vs `FileNotFoundError`
- `DatabaseExploded` vs `DatabaseExplodedError`

Section 22.2: Handling multiple exceptions

You can handle multiple errors in the same `rescue` declaration:

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

You can also add multiple `rescue` declarations:

```
begin
  # an execution that may fail
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
rescue => e
  # do something if a StandardError occurs
end
```

The order of the `rescue` blocks is relevant: the first match is the one executed. Therefore, if you put `StandardError` as the first condition and all your exceptions inherit from `StandardError`, then the other `rescue` statements will never be executed.

```ruby
begin
  # 可能失败的执行
rescue => e
  # 这将捕获所有错误
rescue FirstError => e
  # 如果发生 FirstError，则执行某些操作
rescue SecondError => e
  # 如果发生 SecondError，则执行某些操作
end
```

有些代码块如 def、class 和 module 具有隐式的异常处理功能。这些代码块允许你省略 begin 语句。

```ruby
def foo
   ...
rescue CustomError
   ...
ensure
    ...
end
```

第22.3节：异常处理

使用begin/rescue块来捕获（rescue）异常并进行处理：

```ruby
begin
  # 可能失败的执行
rescue
  # 失败时执行的操作
end
```

一个rescue子句类似于像C#或Java这类大括号语言中的catch块。

像这样一个裸露的rescue会捕获StandardError。

注意：要避免捕获Exception而不是默认的StandardError。因为Exception类包括SystemExit、NoMemoryError以及其他通常不希望捕获的严重异常。通常应考虑捕获默认的StandardError。

你也可以指定应该被捕获的异常类：

```ruby
begin
  # 可能失败的执行
rescue CustomError
  # 在发生CustomError
  # 或其子类时执行的操作
end
```

此 rescue 子句不会捕获任何非CustomError的异常。

你也可以将异常存储在特定变量中：

```ruby
begin
  # 可能失败的执行
rescue CustomError => error
  # error 包含异常信息
  puts error.message # 提供关于错误原因的可读详细信息。
```



```ruby
begin
  # an execution that may fail
rescue => e
  # this will swallow all the errors
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
end
```

Some blocks have implicit exception handling like `def`, `class`, and `module`. These blocks allow you to skip the `begin` statement.

```ruby
def foo
   ...
rescue CustomError
   ...
ensure
   ...
end
```

Section 22.3: Handling an exception

Use the `begin`/`rescue` block to catch (rescue) an exception and handle it:

```ruby
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
end
```

A `rescue` clause is analogous to a `catch` block in a curly brace language like C# or Java.

A bare `rescue` like this rescues `StandardError`.

Note: Take care to avoid catching `Exception` instead of the default `StandardError`. The `Exception` class includes `SystemExit` and `NoMemoryError` and other serious exceptions that you usually don't want to catch. Always consider catching `StandardError` (the default) instead.

You can also specify the exception class that should be rescued:

```ruby
begin
  # an excecution that may fail
rescue CustomError
  # something to execute in case of CustomError
  # or descendant
end
```

This rescue clause will not catch any exception that is not a `CustomError`.

You can also store the exception in a specific variable:

```ruby
begin
  # an excecution that may fail
rescue CustomError => error
  # error contains the exception
  puts error.message # provide human-readable details about what went wrong.
```

```
  puts error.backtrace.inspect # 返回表示调用堆栈的字符串数组
end
```

如果你未能处理异常，可以在 rescue 块中随时重新抛出它。

```
begin
   # 这里写你的代码
rescue => e
    # 处理失败
    raise e
end
```

如果你想重试你的 begin 块，调用 retry：

```
begin
   # 这里写你的代码
rescue StandardError => e
   #出于某种原因你想重试你的代码
   retry
end
```

如果在每次重试中都捕获异常，你可能会陷入循环。为避免这种情况，请将retry_count限制在一定的尝试次数内。

```
retry_count = 0
begin
      # 可能失败的执行
rescue
     if retry_count < 5
retry_count = retry_count + 1
        retry
    else
        #重试次数超过限制，执行其他操作
    end
```

你也可以提供一个else块或一个ensure块。当begin块没有抛出异常时，else块会被执行。无论如何，ensure块都会被执行。ensure块类似于使用大括号的语言（如C#或Java）中的finally块。

```
begin
  # 可能失败的执行
rescue
  # 失败时执行的操作
else
  # 成功时执行的操作
ensure
  # 始终执行的操作
end
```

如果你处于def、module或class代码块内，则无需使用begin语句。

```
def foo
   ...
rescue
    ...
end
```



```
  puts error.backtrace.inspect # return an array of strings that represent the call stack
end
```

If you failed to handle an exception, you can raise it any time in a rescue block.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

If you want to retry your `begin` block, call `retry`:

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

You can be stuck in a loop if you catch an exception in every retry. To avoid this, limit your `retry_count` to a certain number of tries.

```
retry_count = 0
begin
     # an excecution that may fail
rescue
    if retry_count < 5
        retry_count = retry_count + 1
        retry
    else
        #retry limit exceeds, do something else
    end
```

You can also provide an `else` block or an `ensure` block. An `else` block will be executed when the `begin` block completes without an exception thrown. An `ensure` block will always be executed. An `ensure` block is analogous to a `finally` block in a curly brace language like C# or Java.

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
else
  # something to execute in case of success
ensure
  # something to always execute
end
```

If you are inside a `def`, `module` or `class` block, there is no need to use the begin statement.

```
def foo
    ...
rescue
    ...
end
```

第22.4节：引发异常

要引发异常，使用Kernel#raise，传入异常类和/或消息：

```
raise StandardError # 引发一个 StandardError.new
raise StandardError, "An error" # 引发一个 StandardError.new("An error")
```

你也可以直接传入错误消息。在这种情况下，消息会被封装成一个RuntimeError：

```
raise "An error" # 引发一个 RuntimeError.new("An error")
```

下面是一个示例：

```
def hello(subject)
  raise ArgumentError, "`subject` 缺失" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` 缺失
hello("Simone") # => "Hello Simone"
```

第22.5节：向（自定义）异常添加信息

在异常中包含额外信息可能会很有帮助，例如用于日志记录或在捕获异常时允许有条件的处理：

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = '发生了错误')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

抛出异常：

```
raise CustomError.new(true)
```

捕获异常并访问提供的额外信息：

```
begin
  # 执行操作
rescue CustomError => e
  retry if e.safe_to_retry
end
```


Section 22.4: Raising an exception

To raise an exception use `Kernel#raise` passing the exception class and/or message:

```
raise StandardError # raises a StandardError.new
raise StandardError, "An error" # raises a StandardError.new("An error")
```

You can also simply pass an error message. In this case, the message is wrapped into a `RuntimeError`:

```
raise "An error" # raises a RuntimeError.new("An error")
```

Here's an example:

```
def hello(subject)
  raise ArgumentError, "`subject` is missing" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` is missing
hello("Simone") # => "Hello Simone"
```

Section 22.5: Adding information to (custom) exceptions

It may be helpful to include additional information with an exception, e.g. for logging purposes or to allow conditional handling when the exception is caught:

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = 'Something went wrong')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

Raising the exception:

```
raise CustomError.new(true)
```

Catching the exception and accessing the additional information provided:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

第23章：枚举器

参数	详细信息
`yield`	响应yield，该方法的别名为<<。向该对象yield实现了迭代。

一个Enumerator是一个以受控方式实现迭代的对象。

对象不是循环直到满足某个条件，而是根据需要*enumerates*值。循环的执行会暂停，直到对象的拥有者请求下一个值。

Enumerator使得无限值流成为可能。

第23.1节：自定义枚举器

让我们为斐波那契数列创建一个Enumerator。

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
yielder << a
    a, b = b, a + b
  end
end
```

现在我们可以对fibonacci使用任何Enumerable方法：

```
fibonacci.取10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

第23.2节：现有方法

如果像each这样的迭代方法在没有块的情况下被调用，应返回一个Enumerator。

这可以使用enum_for方法来实现：

```
def each
  return enum_for :each unless block_given?

  yield :x
  yield :y
  yield :z
end
```

这使程序员能够组合Enumerable操作：

```
each.drop(2).map(&:upcase).first
# => :Z
```

第23.3节：倒绕

使用 rewind 来重置枚举器。

```
N = Enumerator.new do |yielder|
  x = 0
  loop do
```


Chapter 23: Enumerators

Parameter	Details
`yield`	Responds to `yield`, which is aliased as `<<`. Yielding to this object implements iteration.

An `Enumerator` is an object that implements iteration in a controlled fashion.

Instead of looping until some condition is satisfied, the object *enumerates* values as needed. Execution of the loop is paused until the next value is requested by the owner of the object.

Enumerators make infinite streams of values possible.

Section 23.1: Custom enumerators

Let's create an `Enumerator` for Fibonacci numbers.

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
    yielder << a
    a, b = b, a + b
  end
end
```

We can now use any **`Enumerable`** method with `fibonacci`:

```
fibonacci.take 10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Section 23.2: Existing methods

If an iteration method such as `each` is called without a block, an `Enumerator` should be returned.

This can be done using the `enum_for` method:

```
def each
  return enum_for :each unless block_given?

  yield :x
  yield :y
  yield :z
end
```

This enables the programmer to compose **`Enumerable`** operations:

```
each.drop(2).map(&:upcase).first
# => :Z
```

Section 23.3: Rewinding

Use `rewind` to restart the enumerator.

```
N = Enumerator.new do |yielder|
  x = 0
  loop do
```

```
yielder << x
    x += 1
  end
end

ℕ.next
# => 0

ℕ.next
# => 1

ℕ.next
# => 2

ℕ.rewind

ℕ.next
# => 0
```



```
    yielder << x
    x += 1
  end
end

ℕ.next
# => 0

ℕ.next
# => 1

ℕ.next
# => 2

ℕ.rewind

ℕ.next
# => 0
```

第24章：Ruby中的Enumerable

Enumerable模块，提供一组方法用于遍历、排序、搜索等操作，适用于集合（数组、哈希、集合、哈希映射）。

第24.1节：Enumerable模块

1. For 循环：

```
CountriesName = ["印度", "加拿大", "美国", "伊拉克"]
for country in CountriesName
     puts 国家
end
```

2. 每个 迭代器:
与使用for 循环相同的工作也可以用每个loop完成。

```
国家名称 = ["印度", "加拿大", "美国", "伊拉克"]
国家名称.each do |国家|
     puts 国家
end
```

每个迭代器都会遍历数组中的每一个元素。

```
each ---------- 迭代器
do ------------ 代码块开始
|国家| ----- 传递给代码块的参数
puts 国家----代码块
```

3. each_with_index 迭代器:
each_with_index 迭代器为当前迭代提供元素和该元素在特定集合中的索引。

```
国家名称 = ["印度", "加拿大", "美国", "伊拉克"]
国家名称.each_with_index do |国家, 索引|
    puts 国家 + "   " + 索引.to_s
end
```

4. each_index 迭代器:
只是为了知道元素在集合中所处的索引位置。

```
CountriesName = ["印度", "加拿大", "美国", "伊拉克"]
CountriesName.each_index do |index|
    puts index
end
```

5. map:
“map” 作为迭代器 并且 也用于获取数组的转换副本。用于获取新的数组集合，而不是在同一个特定数组中引入更改。
让我们先处理for循环：
你有一个数组arr = [1,2,3,4,5]
你需要生成新的数组集合。

```
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end
```

上述数组可以被迭代并且可以使用map方法生成新的数组集合。

Chapter 24: Enumerable in Ruby

Enumberable module, a set of methods are available to do traversing, sorting, searching etc across the collection(Array, Hashes, Set, HashMap).

Section 24.1: Enumerable module

```
1. For Loop:
CountriesName = ["India", "Canada", "America", "Iraq"]
for country in CountriesName
    puts country
end
```

```
2. Each Iterator:
Same set of work can be done with each loop which we did with for loop.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each do |country|
    puts country
end
```

```
Each iterator, iterate over every single element of the array.
each ---------- iterator
do ------------ start of the block
|country| ----- argument passed to the block
puts country----block
```

```
3. each_with_index Iterator:
each_with_index iterator provides the element for the current iteration and index of the element in
that specific collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_with_index do |country, index|
    puts country + "   " + index.to_s
end
```

```
4. each_index Iterator:
Just to know the index at which the element is placed in the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_index do |index|
    puts index
end
```

```
5. map:
"map" acts as an iterator and also used to fetch the transformed copy of the array. To fetch the
new set of the array rather than introducing the change in the same specific array.
Let us deal with for loop first:
You have an array arr = [1,2,3,4,5]
You need to produce new set of array.
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end
```

```
The above mentioned array can be iterated and can produce new set of the array using map method.
```

```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]
```

map 返回的是集合当前值的修改副本。arr 的值未被改变。

每个和map 之间的区别：
1. map 返回集合的修改值。

让我们看个例子：

```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]
```

map 方法是迭代器并且也返回转换后集合的副本。

```ruby
arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]
```

each 块会抛出数组，因为这只是迭代器。
每次迭代，不会实际改变迭代中的每个元素。

6. map!
带感叹号的 map 会改变原始集合并且返回修改后的集合，不是修改后集合的副本。

```ruby
arr = [1,2,3,4,5]
arr.map! do |x|
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]
```

7. 结合 map 和 each_with_index
这里 each_with_index 会遍历集合，而 map 会返回集合的修改副本。

```ruby
CountriesName = ["印度", "加拿大", "美国", "伊拉克"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "值是 #{value}，索引是 #{index}"
```



```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]
```

map is returning the modified copy of the current value of the collection. arr has unaltered value.

Difference between each and map:
1. map returned the modified value of the collection.

Let us see the example:

```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]
```

map method is the iterator and also return the copy of transformed collection.

```ruby
arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]
```

each block will throw the array because this is just the iterator.
Each iteration, does not actually alter each element in the iteration.

6. map!
map with bang changes the orginal collection and returned the modified collection not the copy of the modified collection.

```ruby
arr = [1,2,3,4,5]
arr.map! do |x|
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]
```

7. Combining map and each_with_index
Here each_with_index will iterator over the collection and map will return the modified copy of the collection.

```ruby
CountriesName = ["India", "Canada", "America", "Iraq"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "Value is #{value} and the index is #{index}"
```

```
    "值是 #{value}，索引是 #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "值是 #{value}，索引是 #{index}"
        "值是 #{value}，索引是 #{index}"
    end
end

puts newArray
 ["值是印度，索引为0", nil, "值是美国，索引为2", nil]
```

8. 选择

```
混合数组 = [1, "印度", 2, "加拿大", "美国", 4]
混合数组.选择 执行 |值|
    (值.类).等于?整数
end
```

select 方法根据满足某些条件来获取结果。

9. 注入 方法

注入方法将集合归约为某个最终值。

假设你想计算集合的和。

使用 for 循环 它将如何工作

```
数组 = [1,2,3,4,5]
和 = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

上述求和可以通过单个方法简化
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
sum = sum + x
end
inject(0) - 传入初始值 sum = 0
如果 使用 inject 不带参数，sum = arr[0]
sum - 每次迭代后，总和等于块末尾的 return 值。
x  - 指当前迭代元素

inject 方法也是一个迭代器。
```

总结：转换集合的最佳方式是利用 Enumerable 模块来简化繁琐的代码。


```
    "Value is #{value} and the index is #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "Value is #{value} and the index is #{index}"
        "Value is #{value} and the index is #{index}"
    end
end

puts newArray
 ["Value is India and the index is 0", nil, "Value is America and the index is 2", nil]
```

```
8. select
MixedArray = [1, "India", 2, "Canada", "America", 4]
MixedArray.select do |value|
    (value.class).eql?Integer
end

select method fetches the result based on satifying certain condition.
```

```
9. inject methods
inject method reduces the collection to a certain final value.
Let us say you want to find out the sum of the collection.
With for loop how would it work
arr = [1,2,3,4,5]
sum = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

So above mentioned sum can be reduce by single method
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
    sum = sum + x
end
inject(0) - passing initial value sum = 0
If used inject with no argument sum = arr[0]
sum - After each iteration, total is equal to the return value at the end of the block.
x  - refers to the current iteration element

inject method is also an iterator.
```

Summary: Best way to transform the collection is to make use of Enumerable module to compact the clunky code.

第25章：类

第25.1节：构造函数

一个类只能有一个构造函数，即一个名为initialize的方法。当创建该类的新实例时，该方法会被自动调用。

```
class Customer
  def initialize(name)
    @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

第25.2节：创建类

你可以使用class关键字定义一个新类。

```
class MyClass
end
```

定义后，你可以使用.new方法创建一个新实例

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

第25.3节：访问级别

Ruby 有三种访问级别。它们是public、private和protected。

跟随private或protected关键字的方法被定义为相应的访问级别。在这些关键字之前的方法隐式地是public方法。

公共方法

公共方法应描述所创建对象的行为。这些方法可以从
创建对象的作用域之外调用。

```
class Cat
  def initialize(name)
    @name = name
  end

  def 说话
    puts "我是#{@name}，今年2岁"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.说话
```


Chapter 25: Classes

Section 25.1: Constructor

A class can have only one constructor, that is a method called `initialize`. The method is automatically invoked when a new instance of the class is created.

```
class Customer
  def initialize(name)
    @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

Section 25.2: Creating a class

You can define a new class using the `class` keyword.

```
class MyClass
end
```

Once defined, you can create a new instance using the `.new` method

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

Section 25.3: Access Levels

Ruby has three access levels. They are `public`, `private` and `protected`.

Methods that follow the `private` or `protected` keywords are defined as such. Methods that come before these are implicitly `public` methods.

Public Methods

A public method should describe the behavior of the object being created. These methods can be called from outside the scope of the created object.

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    puts "I'm #{@name} and I'm 2 years old"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.speak
```

```
#=> 我是加菲猫，我两岁了
```

这些方法是公共的 Ruby 方法，它们描述了初始化一只新猫的行为以及 speak 方法的行为。

public 关键字不是必须的，但可以用来取消 private 或 protected 的限制

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

私有方法

私有方法无法从对象外部访问。它们在对象内部使用。再次使用猫的
示例：

```
class Cat
  def initialize(name)
    @name = name
  end

  def 说话
age = calculate_cat_age # 这里我们调用私有方法
    puts "我是 #{@name}，我今年 #{age} 岁"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> 我是 Bilbo，我今年 2 岁
my_cat.calculate_cat_age #=> NoMethodError: 调用了私有方法 `calculate_cat_age`
#<Cat:0x2321868 @name="Bilbo">
```

如上例所示，新创建的 Cat 对象可以在内部访问 calculate_cat_age 方法。我们将变量 age 赋值为运行私有 calculate_cat_age 方法的结果，该方法将猫的名字和年龄打印到控制台。

当我们尝试从 my_cat 对象外部调用 calculate_cat_age 方法时，会收到 **NoMethodError** 因为它是私有的。明白了吗？

受保护的方法

受保护的方法与私有方法非常相似。它们不能在对象实例之外被访问


```
#=> I'm garfield and I'm 2 years old
```

These methods are public ruby methods, they describe the behavior for initializing a new cat and the behavior of the speak method.

`public` keyword is unnecessary, but can be used to escape `private` or `protected`

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

Private Methods

Private methods are not accessible from outside of the object. They are used internally by the object. Using the cat example again:

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    age = calculate_cat_age # here we call the private method
    puts "I'm #{@name} and I'm #{age} years old"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> I'm Bilbo and I'm 2 years old
my_cat.calculate_cat_age #=> NoMethodError: private method `calculate_cat_age' called for
#<Cat:0x2321868 @name="Bilbo">
```

As you can see in the example above, the newly created Cat object has access to the `calculate_cat_age` method internally. We assign the variable age to the result of running the private `calculate_cat_age` method which prints the name and age of the cat to the console.

When we try and call the `calculate_cat_age` method from outside the `my_cat` object, we receive a **NoMethodError** because it's private. Get it?

Protected Methods

Protected methods are very similar to private methods. They cannot be accessed outside the instance of object in

同样，私有方法不能这样做。然而，使用self Ruby方法，受保护的方法可以在同类型对象的上下文中被调用。

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def 说话
    puts "我是 #{@name}，今年 #{@age} 岁"
  end

  # this == 方法允许我们比较两个对象自身的年龄。
  # 如果两只猫的年龄相同，它们将被视为相等。
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

你可以看到我们给猫类添加了一个年龄参数，并创建了三个新的猫对象，带有名字和年龄。我们将调用own_age受保护的方法来比较猫对象的年龄。

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

看这里，我们能够使用self.own_age受保护的方法获取cat1的年龄，并通过在cat1内部调用cat2.own_age来与cat2的年龄进行比较。

第25.4节：类方法类型

类有三种类型的方法：实例方法、单例方法和类方法。

实例方法

这些方法可以从类的实例调用。

```
class Thing
  def somemethod
    puts "something"
  end
end
```


the same way private methods can't be. However, using the `self` ruby method, protected methods can be called within the context of an object of the same type.

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def speak
    puts "I'm #{@name} and I'm #{@age} years old"
  end

  # this == method allows us to compare two objects own ages.
  # if both Cat's have the same age they will be considered equal.
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

You can see we've added an age parameter to the cat class and created three new cat objects with the name and age. We are going to call the `own_age` protected method to compare the age's of our cat objects.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Look at that, we were able to retrieve cat1's age using the `self.own_age` protected method and compare it against cat2's age by calling `cat2.own_age` inside of cat1.

Section 25.4: Class Methods types

Classes have 3 types of methods: instance, singleton and class methods.

Instance Methods

These are methods that can be called from an `instance` of the class.

```
class Thing
  def somemethod
    puts "something"
  end
end
```

```
foo = Thing.new # 创建类的一个实例
foo.somemethod # => something
```

类方法

这些是静态方法，即它们可以在类上调用，而不是在该类的实例上调用。

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

使用self代替类名是等效的。以下代码与上面的代码等效：

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

通过以下方式调用该方法

```
Thing.hello("john Doe")  # 输出: "Hello, John Doe!"
```

单例方法

这些方法仅对类的特定实例可用，而不是对所有实例都可用。

```
# 创建一个空类
class Thing
end

# 类的两个实例
thing1 = Thing.new
thing2 = Thing.new

# 创建一个单例方法
def thing1.makestuff
  puts "我属于thing one"
end

thing1.makestuff # => 输出: 我属于thing one
thing2.makestuff # NoMethodError: 未定义方法 `makestuff' 于 #<Thing>
```

单例和类方法都被称为eigenclass。基本上，ruby所做的是创建一个匿名类来保存这些方法，以免干扰已创建的实例。

另一种实现方式是通过class <<构造器。例如：

```
# 一个类方法（与上述示例相同）
class Thing
  class << self # 匿名类
    def hello(name)
      puts "你好，#{name}！"
    end
  end
end

Thing.hello("sarah") # => 你好，sarah！
```



```
foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

Class Method

These are static methods, i.e, they can be invoked on the class, and not on an instantiation of that class.

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

It is equivalent to use `self` in place of the class name. The following code is equivalent to the code above:

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Invoke the method by writing

```
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

Singleton Methods

These are only available to specific instances of the class, but not to all.

```
# create an empty class
class Thing
end

# two instances of the class
thing1 = Thing.new
thing2 = Thing.new

# create a singleton method
def thing1.makestuff
  puts "I belong to thing one"
end

thing1.makestuff # => prints: I belong to thing one
thing2.makestuff # NoMethodError: undefined method `makestuff' for #<Thing>
```

Both the `singleton` and `class` methods are called `eigenclasses`. Basically, what ruby does is to create an anonymous class that holds such methods so that it won't interfere with the instances that are created.

Another way of doing this is by the `class <<` constructor. For example:

```
# a class method (same as the above example)
class Thing
  class << self # the anonymous class
    def hello(name)
      puts "Hello, #{name}!"
    end
  end
end

Thing.hello("sarah") # => Hello, sarah!
```

```ruby
# 单例方法

class Thing
end

thing1 = Thing.new

类 << thing1
定义 makestuff 方法
输出 "我属于 thing one"
  结束
end

thing1.makestuff # => 输出: "我属于 thing one"
```

第25.5节：使用 getter 和

setter 访问实例变量

我们有三种方法：

1. attr_reader：用于允许在类外读取变量。
2. attr_writer：用于允许在类外修改变量。
3. attr_accessor：结合了上述两种方法。

```ruby
class Cat
attr_reader :age # 你可以读取 age，但不能修改
  attr_writer :name # 你可以修改 name，但不能读取
  attr_accessor :breed # 你既可以修改 breed，也可以读取

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def 说话
    puts "我是 #{@name}，我是一只 #{@breed} 猫"
  end
end

my_cat = Cat.new("Banjo", "birman")
# 读取值：

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> 错误

# 修改值

my_cat.age = 3 #=> 错误
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> 我是 Bilbo，我是一只 sphynx 猫
```

请注意，参数是符号。这是通过创建一个方法来实现的。

```ruby
class Cat
attr_accessor :breed
end
```



```ruby
# singleton method

class Thing
end

thing1 = Thing.new

class << thing1
  def makestuff
    puts "I belong to thing one"
  end
end

thing1.makestuff # => prints: "I belong to thing one"
```

Section 25.5: Accessing instance variables with getters and setters

We have three methods:

1. **`attr_reader`**: used to allow reading the variable outside the class.
2. **`attr_writer`**: used to allow modifying the variable outside the class.
3. **`attr_accessor`**: combines both methods.

```ruby
class Cat
  attr_reader :age # you can read the age but you can never change it
  attr_writer :name # you can change name but you are not allowed to read
  attr_accessor :breed # you can both change the breed and read it

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def speak
    puts "I'm #{@name} and I am a #{@breed} cat"
  end
end

my_cat = Cat.new("Banjo", "birman")
# reading values:

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> Error

# changing values

my_cat.age = 3 #=> Error
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> I'm Bilbo and I am a sphynx cat
```

Note that the parameters are symbols. this works by creating a method.

```ruby
class Cat
  attr_accessor :breed
end
```

基本上与以下内容相同：

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

第25.6节：new、allocate和initialize

在许多语言中，类的新实例是通过一个特殊的new关键字创建的。在Ruby中，new也用于创建类的实例，但它不是关键字；相反，它是一个静态/类方法，与其他任何静态/类方法没有区别。定义大致如下：

```
class MyClass
    def self.new(*args)
      obj = allocate
obj.initialize(*args) # 过于简化；initialize 实际上是私有的
      obj
    end
end
```

`allocate` 执行创建类的未初始化实例的真正“魔法”

注意，`initialize` 的返回值被丢弃，取而代之的是返回 obj。这立即说明了为什么你可以编写 initialize 方法而不必担心最后返回 `self`。

所有类从 Class 继承的“正常”new方法如上所述工作，但你可以根据需要重新定义它，或者定义不同的替代方法。例如：

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

第25.7节：动态类创建

类可以通过使用 Class.new 动态创建。

```
# 动态创建一个新类
MyClass = Class.new

# 实例化一个 MyClass 类型的对象
my_class = MyClass.new
```

在上述示例中，创建了一个新类并赋值给常量 MyClass。该类可以像其他任何类一样被实例化和使用。

Class.new方法接受一个Class，该类将成为动态创建类的超类。

Is basically the same as:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

Section 25.6: New, allocate, and initialize

In many languages, new instances of a class are created using a special `new` keyword. In Ruby, `new` is also used to create instances of a class, but it isn't a keyword; instead, it's a static/class method, no different from any other static/class method. The definition is roughly this:

```
class MyClass
  def self.new(*args)
    obj = allocate
    obj.initialize(*args) # oversimplied; initialize is actually private
    obj
  end
end
```

`allocate` performs the real 'magic' of creating an uninitialized instance of the class

Note also that the return value of `initialize` is discarded, and obj is returned instead. This makes it immediately clear why you can code your initialize method without worrying about returning `self` at the end.

The 'normal' `new` method that all classes get from `Class` works as above, but it's possible to redefine it however you like, or to define alternatives that work differently. For example:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Section 25.7: Dynamic class creation

Classes can be created dynamically through the use of `Class.new`.

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

In the above example, a new class is created and assigned to the constant `MyClass`. This class can be instantiated and used just like any other class.

The `Class.new` method accepts a `Class` which will become the superclass of the dynamically created class.

```ruby
# 动态创建一个子类
Staffy = Class.new(Dog)

# 实例化一个Staffy类型的对象
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

Class.new方法也接受一个块。块的上下文是新创建的类。这允许定义方法。

```ruby
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# 实例化一个Duck类型的对象
duck = Duck.new
duck.quack # 'Quack!!'
```

第25.8节：类变量和实例变量

类可以使用几种特殊的变量类型来更方便地共享数据。

实例变量，前面加@。如果你想在不同方法中使用相同的变量，它们非常有用。

```ruby
类 Person
  定义 initialize(name, age)
my_age = age # 局部变量，构造函数结束时销毁
    @name = name # 实例变量，只有对象销毁时才销毁
  结束

  def some_method
    输出 "我的名字是 #{@name}。" # 我们可以无问题地使用 @name
  结束

  定义 another_method
    输出 "我的年龄是 #{my_age}。" # 这将不起作用！
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> 我的名字是 Mark。
mhmd.another_method #=> 抛出错误
```

类变量，前面加@@。它们在类的所有实例中包含相同的值。

```ruby
类 Person
  @@persons_created = 0 # 类变量，所有该类的对象均可使用
  def initialize(name)
    @name = name

    # 类变量的修改会在该类的所有对象中持续生效
    @@persons_created += 1
  结束
```



```ruby
# dynamically create a class that subclasses another
Staffy = Class.new(Dog)

# instantiate an object of type Staffy
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

The `Class.new` method also accepts a block. The context of the block is the newly created class. This allows methods to be defined.

```ruby
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

Section 25.8: Class and instance variables

There are several special variable types that a class can use for more easily sharing data.

Instance variables, preceded by `@`. They are useful if you want to use the same variable in different methods.

```ruby
class Person
  def initialize(name, age)
    my_age = age # local variable, will be destroyed at end of constructor
    @name = name # instance variable, is only destroyed when the object is
  end

  def some_method
    puts "My name is #{@name}." # we can use @name with no problem
  end

  def another_method
    puts "My age is #{my_age}." # this will not work!
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> My name is Mark.
mhmd.another_method #=> throws an error
```

Class variable, preceded by `@@`. They contain the same values across all instances of a class.

```ruby
class Person
  @@persons_created = 0 # class variable, available to all objects of this class
  def initialize(name)
    @name = name

    # modification of class variable persists across all objects of this class
    @@persons_created += 1
  end
```

```
  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> 到目前为止创建的人数：1
helen = Person.new("海伦")

mark.how_many_persons #=> 到目前为止创建的人数：2
helen.how_many_persons #=> 到目前为止创建的人数：2
# 你可以问 mark 或 helen
```

全局变量，前面带有$。这些变量在程序的任何地方都可用，因此请确保明智地使用它们。

```
$total_animals = 0

class 猫
  def initialize
    $total_animals += 1
  end
end

class 狗
  def initialize
    $total_animals += 1
  end
end

bob = 猫.new()
puts $total_animals #=> 1
fred = 狗.new()
puts $total_animals #=> 2
```



```
  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> persons created so far: 1
helen = Person.new("Helen")

mark.how_many_persons #=> persons created so far: 2
helen.how_many_persons #=> persons created so far: 2
# you could either ask mark or helen
```

Global Variables, preceded by $. These are available anywhere to the program, so make sure to use them wisely.

```
$total_animals = 0

class Cat
  def initialize
    $total_animals += 1
  end
end

class Dog
  def initialize
    $total_animals += 1
  end
end

bob = Cat.new()
puts $total_animals #=> 1
fred = Dog.new()
puts $total_animals #=> 2
```

第26章：继承

第26.1节：子类

继承允许类基于现有类定义特定行为。

```
class 动物
  def say_hello
    '喵！'
  end

  def eat
    '好吃！'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

在这个例子中：

- Dog 继承自 Animal，成为一个 子类。
- Dog 从 Animal 获得了 say_hello 和 eat 方法。
- Dog 重写了 say_hello 方法，具有不同的功能。

第26.2节：继承了什么？

方法是被继承的

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

类方法是继承的

```
class A
  def self.boo; p 'boo' end
end

class B < A; end

p B.boo # => 'boo'
```

常量是继承的

Chapter 26: Inheritance

Section 26.1: Subclasses

Inheritance allows classes to define specific behaviour based on an existing class.

```
class Animal
  def say_hello
    'Meep!'
  end

  def eat
    'Yumm!'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

In this example:

- `Dog` Inherits from `Animal`, making it a *Subclass*.
- `Dog` gains both the `say_hello` and `eat` methods from `Animal`.
- `Dog` overrides the `say_hello` method with different functionality.

Section 26.2: What is inherited?

Methods are inherited

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

Class methods are inherited

```
class A
  def self.boo; p 'boo' end
end

class B < A; end

p B.boo # => 'boo'
```

Constants are inherited

```ruby
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

但要注意，它们可以被重写：

```ruby
class B
WOO = WOO + 1
end

p B::WOO # => 2
```

实例变量是继承的：

```ruby
class A
attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

注意，如果你重写了初始化实例变量的方法而没有调用 super，实例变量将会是 nil。
接着上面内容：

```ruby
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

类实例变量不会被继承：

```ruby
class A
    @foo = 'foo'
    class << self
        attr_accessor :foo
    end
end

class B < A; end

p B.foo # => nil

# 访问器是继承的，因为它是类方法
#
B.foo = 'fob' # 可行
```

类变量并不是真正继承的


```ruby
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

But beware, they can be overridden:

```ruby
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

Instance variables are inherited:

```ruby
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Beware, if you override the methods that initialize instance variables without calling `super`, they will be nil.
Continuing from above:

```ruby
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

Class instance variables are not inherited:

```ruby
class A
    @foo = 'foo'
    class << self
        attr_accessor :foo
    end
end

class B < A; end

p B.foo # => nil

# The accessor is inherited, since it is a class method
#
B.foo = 'fob' # possible
```

Class variables aren't really inherited

它们在基类和所有子类之间作为一个变量共享：

```
class A
@@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

所以接着上面继续：

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

第26.3节：多重继承

多重继承是一种允许一个类继承自多个类（即多个父类）的特性。
Ruby 不支持多重继承。它只支持单继承（即一个类只能有一个父类），
但你可以使用*组合*来通过模块构建更复杂的类。

第26.4节：混入（Mixins）

混入是一种实现类似多重继承的优雅方式。它允许我们继承或更准确地说将模块中定义的方法包含到类中。这些方法可以作为实例方法或类方法被包含。下面的例子展示了这种设计。

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "这是一个静态方法"
    end

  end

  def insta_method
    puts "这是一个实例方法"
  end

end
```


They are shared between the base class and all subclasses as 1 variable:

```
class A
    @@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

So continuing from above:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Section 26.3: Multiple Inheritance

Multiple inheritance is a feature that allows one class to inherit from multiple classes(i.e., more than one parent).
Ruby does not support multiple inheritance. It only supports single-inheritance (i.e. class can have only one parent),
but you can use *composition* to build more complex classes using Modules.

Section 26.4: Mixins

Mixins are a beautiful way to achieve something similar to multiple inheritance. It allows us to inherit or rather include methods defined in a module into a class. These methods can be included as either instance or class methods. The below example depicts this design.

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "This is a static method"
    end

  end

  def insta_method
    puts "This is an instance method"
  end

end
```

```ruby
class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "这是一个实例方法"

sc.class.method_static

prints "这是一个静态方法"
```

第26.5节：重构现有类以使用继承

假设我们有两个类，猫和狗。

```ruby
class Cat
  def eat
die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "喵"
  end
end

class 狗
  def eat
die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "汪"
  end
end
```

这两个类中的 eat 方法完全相同。虽然这样可以工作，但维护起来很困难。如果有更多具有相同 eat 方法的动物，问题会变得更糟。继承可以解决这个问题。

```ruby
class 动物
  def eat
die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # 没有 sound 方法
end

class 猫 < 动物
  def sound
    puts "喵"
  end
end

class 狗 < 动物
  def sound
```



```ruby
class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "This is an instance method"

sc.class.method_static

prints "This is a static method"
```

Section 26.5: Refactoring existing classes to use Inheritance

Let's say we have two classes, `Cat` and `Dog`.

```ruby
class Cat
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Meow"
  end
end

class Dog
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Woof"
  end
end
```

The `eat` method is exactly the same in these two classes. While this works, it is hard to maintain. The problem will get worse if there are more animals with the same `eat` method. Inheritance can solve this problem.

```ruby
class Animal
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # No sound method
end

class Cat < Animal
  def sound
    puts "Meow"
  end
end

class Dog < Animal
  def sound
```

```
    puts "Woof"
  end
end
```

我们创建了一个新的类，Animal，并将我们的 eat方法移到了该类中。然后，我们让Cat和Dog继承自这个新的公共超类。这消除了重复代码的需要


```
    puts "Woof"
  end
end
```


We have created a new class, `Animal`, and moved our `eat` method to that class. Then, we made `Cat` and `Dog` inherit from this new common superclass. This removes the need for repeating code

第27章：method_missing

参数	详细信息
方法	被调用的方法名称（在上述示例中是:say_moo，注意这是一个符号）
*args	传入该方法的参数。可以是任意数量，也可以没有
&block	被调用方法的代码块，这可以是一个 do块，或者一个{ }包围的块

第27.1节：捕获对未定义方法的调用

```
class 动物
  def method_missing(method, *args, &block)
    "不能在Animal上调用#{method}"
  end
end

=> Animal.new.say_moo
> "无法在 Animal 上调用 say_moo"
```

第27.2节：与 block 一起使用

```
class 动物
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

第27.3节：带参数使用

```
class 动物
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
     speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```


Chapter 27: method_missing

Parameter	Details
method	The name of the method that has been called (in the above example this is `:say_moo`, note that this is a symbol.
*args	The arguments passed in to this method. Can be any number, or none
&block	The block of the method called, this can either be a `do` block, or a `{ }` enclosed block

Section 27.1: Catching calls to an undefined method

```
class Animal
  def method_missing(method, *args, &block)
    "Cannot call #{method} on Animal"
  end
end

=> Animal.new.say_moo
> "Cannot call say_moo on Animal"
```

Section 27.2: Use with block

```
class Animal
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
      block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

Section 27.3: Use with parameter

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
      speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```

第27.4节：使用缺失方法

```
class 动物
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```


Section 27.4: Using the missing method

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```

第28章：正则表达式和基于正则的操作

第28.1节：=~ 运算符

```
if /hay/ =~ 'haystack'
   puts "单词 haystack 中有 hay"
end
```

注意： 顺序 很重要。虽然 'haystack' =~ /hay/ 在大多数情况下是等价的，但副作用可能 differ:

- 只有在调用 `Regexp#=~` 时，才会将命名捕获组捕获的字符串赋值给局部变量 （`regexp =~ str`）；
- 由于右操作数可能是任意对象，对于 `regexp =~ str` 会调用 `Regexp#=~` 或 `String#=~` 中的一个。

注意这不会返回真/假值，而是返回匹配的索引（如果找到），否则返回 nil。因为 Ruby 中所有整数都是真值（包括 0），而 nil 是假值，所以这样是可行的。如果你想要布尔值，使用

#===

如另一个示例所示。

第 28.2 节：Case 语句中的正则表达式

你可以使用 switch 语句测试一个字符串是否匹配多个正则表达式。

示例

```
case "Ruby is #1!"
when /\APython/
     puts "哔——。"
when /\ARuby/
     puts "你说得对。"
else
     puts "抱歉，我没听懂。"
end
```

这是可行的，因为 case 语句使用 === 运算符而不是 == 运算符来检查相等性。当正则表达式位于 === 比较的左侧时，它会测试字符串是否匹配。

第28.3节：分组，命名分组及其他

Ruby 扩展了标准的分组语法 (...)，增加了命名分组 (?<name>...)。这允许通过名称提取，而不必计算有多少个分组。

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i 表示不区分大小写

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #返回 MatchData 对象或 nil
match_data = name_input.match(name_reg) #两种写法都可用
```


Chapter 28: Regular Expressions and Regex Based Operations

Section 28.1: =~ operator

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

Note: The order **is significant**. Though `'haystack' =~ /hay/` is in most cases an equivalent, side effects might differ:

- Strings captured from named capture groups are assigned to local variables only when `Regexp#=~` is called (`regexp =~ str`);
- Since the right operand might be is an arbitrary object, for `regexp =~ str` there will be called either `Regexp#=~` or `String#=~`.

Note that this does not return a true/false value, it instead returns either the index of the match if found, or nil if not found. Because all integers in ruby are truthy (including 0) and nil is falsy, this works. If you want a boolean value, use

#===

as shown in another example.

Section 28.2: Regular Expressions in Case Statements

You can test if a string matches several regular expressions using a switch statement.

Example

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

This works because case statements are checked for equality using the === operator, not the == operator. When a regex is on the left hand side of a comparison using ===, it will test a string to see if it matches.

Section 28.3: Groups, named and otherwise

Ruby extends the standard group syntax `(...)` with a named group, `(?<name>...)`. This allows for extraction by name instead of having to count how many groups you have.

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i means case insensitive

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #returns either a MatchData object or nil
match_data = name_input.match(name_reg) #works either way
```

```ruby
if match_data.nil? #务必检查是否为 nil！常见错误。
  puts "没有匹配"
else
match[0] #=> "嗨，我的名字是扎福德·比布尔布罗克斯"
  match[1] #=> "i" #第一个分组，(i|ello)
  match[2] #=> "扎福德·比布尔布罗克斯"
  #因为它是一个命名分组，我们可以通过名称获取它
match[:name]  #=> "扎福德·比布尔布罗克斯"
  match["name"] #=> "扎福德·比布尔布罗克斯"
  puts "你好 #{match[:name]}!"
end
```

匹配的索引是根据左括号的顺序计算的（整个正则表达式作为第一个分组，索引为0）

```ruby
reg = /((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

第28.4节：量词

量词允许指定重复字符串的次数。

- 零次或一次：

```ruby
/a?/
```

- 零次或多次：

```ruby
/a*/
```

- 一次或多次：

```ruby
/a+/
```

- 确切次数：

```ruby
/a{2,4}/  # 两次、三次或四次
/a{2,}/   # 两次或更多
/a{,4}/  # 少于四次（包括零次）
```

默认情况下，量词是贪婪的，这意味着它们会尽可能多地匹配字符，同时仍然满足匹配条件。通常这不会被注意到：

```ruby
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

命名捕获组 site 将被设置为预期的 'Motor Vehicle Maintenance & Repair'。但如果 'StackExchange' 是字符串的可选部分（因为它可能是 'Stack Overflow'），则简单的解决方案将无法按预期工作：


```ruby
if match_data.nil? #Always check for nil! Common error.
  puts "No match"
else
  match[0] #=> "Hi, my name is Zaphod Beeblebrox"
  match[1] #=> "i" #the first group, (i|ello)
  match[2] #=> "Zaphod Beeblebrox"
  #Because it was a named group, we can get it by name
  match[:name]  #=> "Zaphod Beeblebrox"
  match["name"] #=> "Zaphod Beeblebrox"
  puts "Hello #{match[:name]}!"
end
```

The index of the match is counted based on the order of the left parentheses (with the entire regex being the first group at index 0)

```ruby
reg = /((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

Section 28.4: Quantifiers

Quantifiers allows to specify count of repeated strings.

- Zero or one:

```ruby
/a?/
```

- Zero or many:

```ruby
/a*/
```

- One or many:

```ruby
/a+/
```

- Exact number:

```ruby
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

By default, quantifiers are greedy, which means they take as many characters as they can while still making a match. Normally this is not noticeable:

```ruby
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

The named capture group `site` will be set to "Motor Vehicle Maintenance & Repair' as expected. But if 'Stack Exchange' is an optional part of the string (because it could be 'Stack Overflow' instead), the naive solution will not work as expected:

```
/(?<site>.*)( Stack Exchange)?/
```

此版本仍然匹配，但命名捕获将包含“Stack Exchange”，因为*贪婪地匹配了这些字符。解决方法是在*后面加一个问号，使其变为懒惰匹配：

```
/(?<site>.*?)( Stack Exchange)?/
```

在任何量词后添加?都会使其变为懒惰匹配。

第28.5节：常用快速用法

正则表达式经常作为参数用于方法中，以检查其他字符串是否存在，或用于搜索和/或替换字符串。

你经常会看到如下用法：

```
string = "My not so long string"
string[/so/] # 返回 so
string[/present/] # 返回 nil
string[/present/].nil? # 返回 true
```

所以你可以简单地用它来检查字符串是否包含子字符串

```
如果字符串[/so/]则输出"found"
```

更高级但仍然简短快速：通过使用第二个参数搜索特定的分组，2 是第二个在此示例中，因为编号从1开始而不是0，分组是指括号内的内容。

```
string[/(n.t).+(l.ng)/, 2] # 返回 long
```

也经常使用：使用 sub 或 gsub 进行搜索和替换，\1 表示第一个找到的分组，\2 表示第二个：

```
string.gsub(/(n.t).+(l.ng)/, '\1 非常 \2') # 我的 not very long 字符串
```

最后的结果被记住，可以在后续行中使用

```
$2 # 返回 long
```

第28.6节：match? - 布尔结果

返回 `true` 或 `false`，表示正则表达式是否匹配，但不会更新 $~ 和其他相关变量。如果存在第二个参数，则指定开始搜索的字符串位置。

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

第28.7节：定义正则表达式

在Ruby中，可以通过三种不同的方式创建正则表达式。


```
/(?<site>.*)( Stack Exchange)?/
```

This version will still match, but the named capture will include 'Stack Exchange' since * greedily eats those characters. The solution is to add another question mark to make the * lazy:

```
/(?<site>.*?)( Stack Exchange)?/
```

Appending ? to any quantifier will make it lazy.

Section 28.5: Common quick usage

Regular expressions are often used in methods as parameters to check if other strings are present or to search and/or replace strings.

You'll often see the following:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

So you can simply use this as a check if a string contains a substring

```
puts "found" if string[/so/]
```

More advanced but still short and quick: search for a specific group by using the second parameter, 2 is the second in this example because numbering starts at 1 and not 0, a group is what is enclosed in parentheses.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

Also often used: search and replace with `sub` or `gsub`, `\1` gives the first found group, `\2` the second:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

The last result is remembered and can be used on the following lines

```
$2 # gives long
```

Section 28.6: match? - Boolean Result

Returns `true` or `false`, which indicates whether the regexp is matched or not without updating $~ and other related variables. If the second parameter is present, it specifies the position in the string to begin the search.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

Section 28.7: Defining a Regexp

A Regexp can be created in three different ways in Ruby.

- 使用斜杠：`/ /`
- 使用`%r{}`
- 使用Regex.new

```
#以下形式是等价的
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#所有这些都会返回真值
string_to_match =~ regexp_slash     # => 0
string_to_match =~ regexp_bracket   # => 0
string_to_match =~ regexp_new       # => 0
```

第28.8节：字符类

描述符号的范围

你可以显式列举符号

```
/[abc]/ # 'a' 或 'b' 或 'c'
```

或者使用范围

```
/[a-z]/ # 从 'a' 到 'z'
```

可以组合范围和单个符号

```
/[a-cz]/ # 'a' 或 'b' 或 'c' 或 'z'
```

开头的短横线（-）被视为字符

```
/[-a-c]/ # '-' 或 'a' 或 'b' 或 'c'
```

当符号前有 ^ 时，字符类可以取反

```
/[^a-c]/ # 不是 'a'、'b' 或 'c'
```

对于常用字符类和特殊字符，以及行尾，有一些快捷方式

```
^  # 行首
$  # 行尾
\A # 字符串开头
\Z # 字符串结束，不包括字符串末尾的任何换行符
\z # 字符串结束
.  # 任意单个字符
\s # 任意空白字符
\S # 任意非空白字符
\d # 任意数字
\D # 任意非数字字符
\w # 任意单词字符（字母、数字、下划线）
\W # 任意非单词字符
```


- using slashes: `/ /`
- using `%r{}`
- using `Regex.new`

```
#The following forms are equivalent
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#All of these will return a truthy value
string_to_match =~ regexp_slash    # => 0
string_to_match =~ regexp_bracket  # => 0
string_to_match =~ regexp_new      # => 0
```

Section 28.8: Character classes

Describes ranges of symbols

You can enumerate symbols explicitly

```
/[abc]/ # 'a' or 'b' or 'c'
```

Or use ranges

```
/[a-z]/ # from 'a' to 'z'
```

It is possible to combine ranges and single symbols

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

Leading dash (-) is treated as character

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Classes can be negative when preceding symbols with `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

There are some shortcuts for widespread classes and special characters, plus line endings

```
^  # Start of line
$  # End of line
\A # Start of string
\Z # End of string, excluding any new line at the end of string
\z # End of string
.  # Any single character
\s # Any whitespace character
\S # Any non-whitespace character
\d # Any digit
\D # Any non-digit
\w # Any word character (letter, number, underscore)
\W # Any non-word character
```

```
\b # 任意单词边界
```

将被简单理解为换行符

要转义任何保留字符，如 / 或 [] 及其他字符，请使用反斜杠（左斜杠）

```
\\ # => \
\[\] # => []
```



```
\b # Any word boundary
```

\n will be understood simply as new line

To escape any reserved character, such as / or [ ] and others use backslash (left slash)

```
\\ # => \
\[\] # => []
```

第29章：文件和输入输出操作

标志	含义
"r"	只读，从文件开头开始（默认模式）。
"r+"	读写，从文件开头开始。
"w"	只写，截断现有文件为零长度或创建新文件进行写入。
"w+"	读写，截断现有文件为零长度或创建新文件进行读写。
"a"	只写，如果文件存在则从文件末尾开始，否则创建新文件进行写入。
"a+"	读写，如果文件存在则从文件末尾开始，否则创建新文件进行读写。
"b"	二进制文件模式。在Windows上抑制EOL <-> CRLF转换。并将外部编码设置为ASCII-8BIT除非明确指定。（此标志只能与上述标志一起出现。例如，**File.new("test.txt", "rb") 会以二进制文件的只读模式打开 test.txt。**）
"t"	文本文件模式。（此标志只能与上述标志一起出现。例如，File.new("test.txt", "wt") 会以文本文件的只写模式打开 test.txt。）

第29.1节：向文件写入字符串

可以使用 File 类的实例将字符串写入文件。

```
file = File.new('tmp.txt', 'w')file.write("
NaNaNaNa")
file.write('Batman!')file.close
```

File 类还提供了 open 方法，作为 new 和 close 操作的简写。

```
File.open('tmp.txt', 'w') do |f|
f.write("NaNaNaNa")
f.write('Batman!')
end
```

对于简单的写操作，也可以直接使用File.write将字符串写入文件。请注意，这将默认覆盖文件内容。

```
File.write('tmp.txt', "NaNaNaNa" * 4 + 'Batman!')
```

要在File.write中指定不同的模式，可以将其作为名为mode的键的值传入哈希作为另一个参数。

```
File.write('tmp.txt', "NaNaNaNa" * 4 + 'Batman!', { mode: 'a'})
```

第29.2节：从STDIN读取

```
# 从STDIN获取两个数字，以换行符分隔，并输出结果
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i##
运行方式：$ ruby a_plus_b.rb## 或者
：$ echo -e "12" | ruby a_plus_b.rb
```

第29.3节：从ARGV参数读取

```
number1 = ARGV[0]
```


Chapter 29: File and I/O Operations

Flag	Meaning
"r"	Read-only, starts at beginning of file (default mode).
"r+"	Read-write, starts at beginning of file.
"w"	Write-only, truncates existing file to zero length or creates a new file for writing.
"w+"	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
"a"	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
"a+"	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.
"b"	Binary file mode. Suppresses EOL <-> CRLF conversion on Windows. And sets external encoding to ASCII-8BIT unless explicitly specified. (This flag may only appear in conjunction with the above flags. For example, `File.new("test.txt", "rb")` would open `test.txt` in `read-only` mode as a `binary` file.)
"t"	Text file mode. (This flag may only appear in conjunction with the above flags. For example, `File.new("test.txt", "wt")` would open `test.txt` in `write-only` mode as a `text` file.)

Section 29.1: Writing a string to a file

A string can be written to a file with an instance of the `File` class.

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
file.write('Batman!\n')
file.close
```

The `File` class also offers a shorthand for the `new` and `close` operations with the `open` method.

```
File.open('tmp.txt', 'w') do |f|
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

For simple write operations, a string can be also written directly to a file with `File.write`. **Note that this will overwrite the file by default.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

To specify a different mode on `File.write`, pass it as the value of a key called `mode` in a hash as another parameter.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

Section 29.2: Reading from STDIN

```
# Get two numbers from STDIN, separated by a newline, and output the result
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb
## or:       $ echo -e "1\n2" | ruby a_plus_b.rb
```

Section 29.3: Reading from arguments with ARGV

```
number1 = ARGV[0]
```

```
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## 运行方式：$ ruby a_plus_b.rb 1 2
```

第29.4节：打开和关闭文件

手动打开和关闭文件。

```
# 使用new方法
f = File.new("test.txt", "r") # 读取
f = File.new("test.txt", "w") # 写入
f = File.new("test.txt", "a") # 追加

# 使用open方法
f = open("test.txt", "r")

# 记得关闭文件
f.close
```

使用代码块自动关闭文件。

```
f = File.open("test.txt", "r") do |f|
  # 对文件 f 进行操作
  puts f.read # 例如，读取文件内容
end
```

第29.5节：获取单个输入字符

与 gets.chomp 不同，这不会等待换行符。

必须包含标准库的第一部分

```
require 'io/console'
```

然后可以编写一个辅助方法：

```
def get_char
input = STDIN.getch
   control_c_code = "\u0003"
   exit(1) if input == control_c_code
  input
end
```

如果按下control+c，退出非常重要。


```
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb 1 2
```

Section 29.4: Open and closing a file

Manually open and close a file.

```
# Using new method
f = File.new("test.txt", "r") # reading
f = File.new("test.txt", "w") # writing
f = File.new("test.txt", "a") # appending

# Using open method
f = open("test.txt", "r")

# Remember to close files
f.close
```

Automatically close a file using a block.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

Section 29.5: get a single char of input

Unlike `gets.chomp` this will not wait for a newline.

First part of the stdlib must be included

```
require 'io/console'
```

Then a helper method can be written:

```
def get_char
  input = STDIN.getch
  control_c_code = "\u0003"
  exit(1) if input == control_c_code
  input
end
```

Its' imporant to exit if `control+c` is pressed.

第30章：Ruby访问修饰符

对各种方法、数据成员、初始化方法的访问控制（作用域）。

第30.1节：实例变量和类变量

首先复习一下什么是实例变量：它们更像是对象的属性。它们在对象创建时被初始化。实例变量可以通过实例方法访问。每个对象都有各自的实例变量。实例变量不在对象之间共享。

Sequence类有@from、@to和@by作为实例变量。

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
x = @from
        while x < @to
            yield x
x = x + @by
        end
    end

    def *(factor)
Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end
```

输出：

```
1
3
5
7
9
```

```
                    new(1,10,3)
object1.each do |x|
    puts x
end
```

输出：

```
1
4
```


Chapter 30: Ruby Access Modifiers

Access control(scope) to various methods, data members, initialize methods.

Section 30.1: Instance Variables and Class Variables

Let's first brush up with what are the **Instance Variables:** They behave more like properties for an object. They are initialized on an object creation. Instance variables are accessible through instance methods. Per Object has per instance variables. Instance Variables are not shared between objects.

Sequence class has @from, @to and @by as the instance variables.

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
```

类变量 将类变量视为Java的静态变量，这些变量在该类的各个对象之间共享。类变量存储在堆内存中。

```
类 序列
    包含 Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
@@count = @@count + 1
    结束

    def each
x = @from
        while x < @to
            yield x
x = x + @by
        end
    end

    def *(factor)
Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    定义 getCount
        @@count
    结束
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

输出：
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

输出：
1
4
7
输
出

getCount
输出: 2
```


Class Variables Treat class variable same as static variables of java, which are shared among the various objects of that class. Class Variables are stored in heap memory.

```
class Sequence
    include Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
        @@count = @@count + 1
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    def getCount
        @@count
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7

puts object1.getCount
Output: 2
```

在object和object1之间共享。

比较Ruby的实例变量和类变量与Java的区别：

```
类 序列{
    int from, to, by;
Sequence(from, to, by){// Java 的构造方法相当于 Ruby 的初始化方法
        this.from = from;// Java 中的 this.from 相当于 @from，表示当前对象的 from
        this.to = to;
this.by = by;
    }
public void each(){
int x = this.from;// 对象的属性在对象的上下文中是可访问的。
         while x > this.to
             x = x + this.by
    }
}
```

第30.2节：访问控制

Java 与 Ruby 访问控制的比较： 如果方法在 Java 中声明为 private，则只能被同一类中的其他方法访问。如果方法声明为 protected，则可以被同一包内的其他类以及不同包中该类的子类访问。当方法为 public 时，所有人都可见。在 Java 中，访问控制的可见性取决于这些类在继承/包层次结构中的位置。

而在 Ruby 中，继承层次结构或包/模块并不适用。关键在于哪个对象是方法的接收者。

对于 Ruby 中的私有方法，它永远不能用显式接收者调用。我们只能用隐式接收者调用私有方法。

这也意味着我们可以从声明该私有方法的类内部以及该

```
class Test1
  def main_method
    method_private
  end

  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2
```


Shared among object and object1.

Comparing the instance and class variables of Ruby against Java:

```
Class Sequence{
    int from, to, by;
    Sequence(from, to, by){// constructor method of Java is equivalent to initialize method of ruby
        this.from = from;// this.from of java is equivalent to @from indicating currentObject.from
        this.to = to;
        this.by = by;
    }
    public void each(){
        int x = this.from;//objects attributes are accessible in the context of the object.
        while x > this.to
            x = x + this.by
    }
}
```

Section 30.2: Access Controls

Comparison of access controls of Java against Ruby: If method is declared private in Java, it can only be accessed by other methods within the same class. If a method is declared protected it can be accessed by other classes which exist within the same package as well as by subclasses of the class in a different package. When a method is public it is visible to everyone. In Java, access control visibility concept depends on where these classes lie's in the inheritance/package hierarchy.

Whereas in Ruby, the inheritance hierarchy or the package/module don't fit. It's all about which object is the receiver of a method.

For a private method in Ruby, it can never be called with an explicit receiver. We can (only) call the private method with an implicit receiver.

This also means we can call a private method from within a class it is declared in as well as all subclasses of this class.

```
class Test1
  def main_method
    method_private
  end

  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2
```

```
class Test3 < Test1
  def main_method
    self.method_private #我们试图用显式接收者调用一个私有方法，如果在同一个类中用self调用会失败。

  end
end

Test1.new.main_method
这将抛出NoMethodError

你永远不能从定义该方法的类层次结构之外调用私有方法。
```

受保护的方法可以像私有方法一样通过隐式接收者调用。此外，受保护的方法也可以通过显式接收者调用（仅限）如果接收者是“self”或“同一类的对象”。

```
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "在#{self.class}的method_protected内部"
  end
end

class Test2 < Test1
  def main_method
method_protected # 通过隐式接收者调用
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # 通过显式接收者“同一类的对象”调用
  end
end


InSide method_protected for Test1
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
Test2.new.method_protected # "Test2.new 是与 self 相同类型的对象"
  end
end

Test4.new.main_method

class Test5
  def main_method
Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```



```
class Test3 < Test1
  def main_method
    self.method_private #We were trying to call a private method with an explicit receiver and if
called in the same class with self would fail.
  end
end

Test1.new.main_method
This will throw NoMethodError

You can never call the private method from outside the class hierarchy where it was defined.
```

Protected method can be called with an implicit receiver, as like private. In addition protected method can also be called by an explicit receiver (only) if the receiver is "self" or "an object of the same class".

```
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "InSide method_protected for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_protected # called by implicit receiver
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # called by explicit receiver "an object of the same class"
  end
end


InSide method_protected for Test1
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
    Test2.new.method_protected # "Test2.new is the same type of object as self"
  end
end

Test4.new.main_method

class Test5
  def main_method
    Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```

考虑最大可见性的公共方法

总结

1. Public: 公共方法具有最大可见性
2. 受保护的：受保护的方法可以像私有方法一样通过隐式接收者调用。此外，受保护的方法也只能由显式接收者调用（仅限于）当接收者是“self”或“同一类的对象”时。
3. 私有：对于 Ruby 中的私有方法，永远不能用显式接收者调用。我们只能（仅）用隐式接收者调用私有方法。这也意味着我们可以从声明该方法的类内部以及该类的所有子类中调用私有方法。

Consider Public methods with maximum visibility

Summary

1. **Public:** Public methods have maximum visibility
2. **Protected: Protected method** can be called with an implicit receiver, as like private. In addition protected method can also be called by an explicit receiver (only) if the receiver is "self" or "an object of the same class".
3. **Private: For a private method in Ruby**, it can never be called with an explicit receiver. We can (only) call the private method with an implicit receiver. This also means we can call a private method from within a class it is declared in as well as all subclasses of this class.

第31章：Ruby中的设计模式和惯用法

第31.1节：装饰器模式

装饰器模式为对象添加行为，而不影响同一类的其他对象。装饰器模式是创建子类的一个有用替代方案。

为每个装饰器创建一个模块。这种方法比继承更灵活，因为你可以以更多组合方式混合和匹配职责。此外，由于透明性允许装饰器递归嵌套，它允许无限数量的职责。

假设 Pizza 类有一个返回300的 cost 方法：

```
class Pizza
  def cost
    300
  end
end
```

用加了一层芝士爆浆的披萨来表示，成本增加50。最简单的方法是创建一个PizzaWithCheese子类，在cost方法中返回350。

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

接下来，我们需要表示一个大号披萨，它比普通披萨的成本多加100。我们可以用Pizza的一个LargePizza子类来表示。

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

我们还可以有一个ExtraLargePizza，它在LargePizza的基础上再增加15的成本。如果考虑到这些披萨类型可以加芝士，我们就需要添加LargePizzaWithCheese和ExtraLargePizzaWithCheese子类。最终我们会有总共6个类。

为简化方法，使用模块动态地为Pizza类添加行为：

模块 + extend + super 装饰器:->

```
class Pizza
  def cost
    300
  end
end

模块 奶酪披萨
  def 成本
    super + 50
```


Chapter 31: Design Patterns and Idioms in Ruby

Section 31.1: Decorator Pattern

Decorator pattern adds behavior to objects without affecting other objects of the same class. The decorator pattern is a useful alternative to creating sub-classes.

Create a module for each decorator. This approach is more flexible than inheritance because you can mix and match responsibilities in more combinations. Additionally, because the transparency allows decorators to be nested recursively, it allows for an unlimited number of responsibilities.

Assume the Pizza class has a cost method that returns 300:

```
class Pizza
  def cost
    300
  end
end
```

Represent pizza with an added layer of cheese burst and the cost goes up by 50. The simplest approach is to create a `PizzaWithCheese` subclass that returns 350 in the cost method.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

Next, we need to represent a large pizza that adds 100 to the cost of a normal pizza. We can represent this using a LargePizza subclass of Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

We could also have an ExtraLargePizza which adds a further cost of 15 to our LargePizza. If we were to consider that these pizza types could be served with cheese, we would need to add LargePizzaWithChese and ExtraLargePizzaWithCheese subclasses.we end up with a total of 6 classes.

To simplify the approach, use modules to dynamically add behavior to Pizza class:

Module + extend + super decorator:->

```
class Pizza
  def cost
    300
  end
end

module CheesePizza
  def cost
    super + 50
```

```ruby
  end
end

module LargePizza
  def cost
    super + 100
  end
end

pizza = Pizza.new         #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

第31.2节：观察者模式

观察者模式是一种软件设计模式，其中一个对象（称为主题）维护其依赖对象的列表（称为观察者），并在状态发生变化时自动通知它们，通常通过调用它们的某个方法来实现。

Ruby 提供了一个简单的机制来实现观察者设计模式。模块Observable提供了通知订阅者Observable对象中任何变化的逻辑。

为了使其工作，Observable必须声明其状态已改变并通知观察者。

观察的对象必须实现一个update()方法，该方法将作为观察者的回调函数。

让我们实现一个小型聊天系统，用户可以订阅其他用户，当其中一个用户发送消息时，订阅者会收到通知。

```ruby
require "observer"

class Moderator
  include Observable

  def initialize(name)
    @name = name
  end

  def write
message = "Computer says: No"
    changed
notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end
```



```ruby
  end
end

module LargePizza
  def cost
    super + 100
  end
end

pizza = Pizza.new         #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

Section 31.2: Observer

The observer pattern is a software design pattern in which an object (called `subject`) maintains a list of its dependents (called `observers`), and notifies them automatically of any state changes, usually by calling one of their methods.

Ruby provides a simple mechanism to implement the Observer design pattern. The module `Observable` provides the logic to notify the subscriber of any changes in the Observable object.

For this to work, the observable has to assert it has changed and notify the observers.

Objects observing have to implement an `update()` method, which will be the callback for the Observer.

Let's implement a small chat, where users can subscribe to users and when one of them write something, the subscribers get notified.

```ruby
require "observer"

class Moderator
  include Observable

  def initialize(name)
    @name = name
  end

  def write
    message = "Computer says: No"
    changed
    notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
    moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end
```

```
moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

生成以下输出：

```
# 计算机说：不
# 计算机说：不
```

我们在 Moderator 类中触发了write方法两次，通知了它的订阅者，在本例中只有一个订阅者。

我们添加的订阅者越多，变更传播的范围也越广。

第31.3节：单例模式

Ruby 标准库中有一个 Singleton 模块，它实现了单例模式。创建单例类的第一步是引入并包含Singleton模块：

```
require 'singleton'

class Logger
  include Singleton
end
```

如果你像通常实例化普通类那样尝试实例化该类，会抛出NoMethodError异常。构造函数被设为私有，以防止意外创建其他实例：

```
Logger.new

#=> NoMethodError: 调用了 AppConfig:Class 的私有方法 `new`
```

要访问此类的实例，我们需要使用instance()：

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Logger示例

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

为了使用Logger对象：


```
moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

Producing the following output:

```
# Computer says: No
# Computer says: No
```

We've triggered the method `write` at the Moderator class twice, notifying its subscribers, in this case just one.

The more subscribers we add the more the changes will propagate.

Section 31.3: Singleton

Ruby Standard Library has a Singleton module which implements the Singleton pattern. The first step in creating a Singleton class is to require and include the `Singleton` module in a class:

```
require 'singleton'

class Logger
  include Singleton
end
```

If you try to instantiate this class as you normally would a regular class, a `NoMethodError` exception is raised. The constructor is made private to prevent other instances from being accidentally created:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

To access the instance of this class, we need to use the `instance()`:

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Logger example

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

In order to use `Logger` object:

```
Logger.instance.log('message 2')
```

不包含 Singleton 模块

上述单例实现也可以在不包含 Singleton 模块的情况下完成。这可以通过以下方式实现：

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

这是以下代码的简写形式：

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

但是，请记住 Singleton 模块经过测试和优化，因此使用它来实现单例是更好的选择。

第31.4节：代理（Proxy）

代理对象通常用于确保对另一个对象的受保护访问，我们不希望将安全性要求混入该对象的内部业务逻辑中。

假设我们想要保证只有具有特定权限的用户才能访问资源。

代理定义：（它确保只有实际上可以查看预订的用户才能使用预订服务）

```
类 Proxy
  定义 初始化(当前用户, 预订服务)
    @当前用户 = 当前用户
    @预订服务 = 预订服务
  结束

  定义 最高总价预订(起始日期, 结束日期, 预订数量)
    如果 @当前用户.能查看预订?
      @预订服务.最高总价预订(
        起始日期,
结束日期,
预订数量
      )
    否则
      []
    结束
  end
end
```

模型和预订服务：

```
class 预订
  attr_reader :总价, :日期
```



```
Logger.instance.log('message 2')
```

Without Singleton include

The above singleton implementations can also be done without the inclusion of the Singleton module. This can be achieved with the following:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

which is a shorthand notation for the following:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

However, keep in mind that the Singleton module is tested and optimized, therefore being the better option to implement your singleton with.

Section 31.4: Proxy

Proxy object is often used to ensure guarded access to another object, which internal business logic we don't want to pollute with safety requirements.

Suppose we'd like to guarantee that only user of specific permissions can access resource.

Proxy definition: (it ensure that only users which actually can see reservations will be able to consumer reservation_service)

```
class Proxy
  def initialize(current_user, reservation_service)
    @current_user = current_user
    @reservation_service = reservation_service
  end

  def highest_total_price_reservations(date_from, date_to, reservations_count)
    if @current_user.can_see_reservations?
      @reservation_service.highest_total_price_reservations(
        date_from,
        date_to,
        reservations_count
      )
    else
      []
    end
  end
end
```

Models and ReservationService:

```
class Reservation
  attr_reader :total_price, :date
```

```
  def 初始化(日期, 总价)
    @日期 = 日期
    @总价 = 总价
  end
end

class 预订服务
  def 最高总价预订(起始日期, 结束日期, 预订数量)
    # 通常情况下，这些数据会从数据库/外部服务读取
预订列表 = [
预订.new(Date.new(2014, 5, 15), 100),
      预订.new(Date.new(2017, 5, 15), 10),
      预订.new(Date.new(2017, 1, 15), 50)
    ]

筛选后的预订 = 预订列表.select do |预订|
      预订.日期.between?(起始日期, 结束日期)
    end

filtered_reservations.获取(reservations_count)
  end
结束

类 用户
  只读属性 :姓名

  定义 初始化(能否查看预订, 姓名)
    @能否查看预订 = 能否查看预订
    @姓名 = 姓名
  end

  定义 能否查看预订?
    @能否查看预订
  结束
end
```

消费者服务：

```
类 统计服务
  定义 初始化(预订服务)
    @预订服务 = 预订服务
  结束

  定义 年度前100预订平均总价(年份)
    预订 = @预订服务.最高总价预订(
      日期.新建(年份, 1, 1),
      日期.新建(年份, 12, 31),
      100
    )

    if reservations.length > 0
sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

sum / reservations.length
    else
      0
    end
  end
end
```



```
  def initialize(date, total_price)
    @date = date
    @total_price = total_price
  end
end

class ReservationService
  def highest_total_price_reservations(date_from, date_to, reservations_count)
    # normally it would be read from database/external service
    reservations = [
      Reservation.new(Date.new(2014, 5, 15), 100),
      Reservation.new(Date.new(2017, 5, 15), 10),
      Reservation.new(Date.new(2017, 1, 15), 50)
    ]

    filtered_reservations = reservations.select do |reservation|
      reservation.date.between?(date_from, date_to)
    end

    filtered_reservations.take(reservations_count)
  end
end

class User
  attr_reader :name

  def initialize(can_see_reservations, name)
    @can_see_reservations = can_see_reservations
    @name = name
  end

  def can_see_reservations?
    @can_see_reservations
  end
end
```

Consumer service:

```
class StatsService
  def initialize(reservation_service)
    @reservation_service = reservation_service
  end

  def year_top_100_reservations_average_total_price(year)
    reservations = @reservation_service.highest_total_price_reservations(
      Date.new(year, 1, 1),
      Date.new(year, 12, 31),
      100
    )

    if reservations.length > 0
      sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

      sum / reservations.length
    else
      0
    end
  end
end
```

测试：

```
def test(user, year)
reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} 将看到: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
test(User.new(false, "Guest"),         2017)
```

优势

- **我们避免在访问权限更改时对ReservationService进行任何更改。**
- **我们没有在服务中将业务相关数据（date_from、date_to、reservations_count）与领域无关的概念（用户权限）混合。**
- **消费者（StatsService）也不涉及权限相关的逻辑。**

注意事项

- 代理接口始终与其隐藏的对象完全相同，因此使用被代理包装的服务的用户甚至不会察觉代理的存在。

Test:

```
def test(user, year)
  reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
  average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} will see: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
test(User.new(false, "Guest"),         2017)
```

BENEFITS

- **we're avoiding any changes in `ReservationService` when access restrictions are changed.**
- **we're not mixing business related data (`date_from`, `date_to`, `reservations_count`) with domain unrelated concepts (user permissions) in service.**
- **Consumer (`StatsService`) is free from permissions related logic as well**

CAVEATS

- Proxy interface is always exactly the same as the object it hides, so that user that consumes service wrapped by proxy wasn't even aware of proxy presence.

第32章：加载源文件

第32.1节：确保文件只加载一次

Kernel#require 方法只会加载文件一次（多次调用 require 只会导致该文件中的代码被执行一次）。如果参数不是绝对路径，它会搜索你的 Ruby $LOAD_PATH 来查找所需文件。扩展名如 .rb、.so、.o 或 .dll 是可选的。相对路径将解析为进程的当前工作目录。

```
require 'awesome_print'
```

Kernel#require_relative 允许你加载相对于调用 require_relative 的文件的文件。

```
# 将在相对于当前源文件的 myproj 目录中搜索。
#
require_relative 'myproj/version'
```

第32.2节：自动加载源文件

方法Kernel#autoload注册文件名，以便在首次访问模块（可以是字符串或符号）时加载该文件（使用Kernel::require）。

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

方法Kernel#autoload?如果名称被注册为autoload，则返回要加载的文件名。

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

第32.3节：加载可选文件

当文件不可用时，require系列方法会抛出LoadError。以下示例说明仅在可用时加载可选模块。

```
模块 TidBits

@@unavailableModules = []

[
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \

].each do |lib|

    begin

require_relative lib[ :file ]

    rescue LoadError

@@unavailableModules.push lib

    end
```


Chapter 32: Loading Source Files

Section 32.1: Require files to be loaded only once

The Kernel#require method will load files only once (several calls to `require` will result in the code in that file being evaluated only once). It will search your ruby `$LOAD_PATH` to find the required file if the parameter is not an absolute path. Extensions like `.rb`, `.so`, `.o` or `.dll` are optional. Relative paths will be resolved to the current working directory of the process.

```
require 'awesome_print'
```

The Kernel#require_relative allows you to load files relative to the file in which `require_relative` is called.

```
# will search in directory myproj relative to current source file.
#
require_relative 'myproj/version'
```

Section 32.2: Automatically loading source files

The method `Kernel#autoload` registers filename to be loaded (using `Kernel::require`) the first time that module (which may be a String or a symbol) is accessed.

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

The method Kernel#autoload? returns filename to be loaded if name is registered as `autoload`.

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

Section 32.3: Loading optional files

When files are not available, the `require` family will throw a `LoadError`. This is an example which illustrates loading optional modules only if they exist.

```
module TidBits

@@unavailableModules = []

[
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \

].each do |lib|

    begin

        require_relative lib[ :file ]

    rescue LoadError

        @@unavailableModules.push lib

    end
```

```
end

end # module TidBits
```

第32.4节：重复加载文件

Kernel#load方法将会执行给定文件中的代码。搜索路径的构造方式与require相同。它会在每次调用时重新执行该代码，这与require不同。没有load_relative方法。

```
load `somefile`
```

第32.5节：加载多个文件

您可以使用任何 Ruby 技巧动态创建要加载的文件列表。示例说明了以test开头的文件的通配符匹配，按字母顺序加载。

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|

    require_relative source

end
```



```
end

end # module TidBits
```

Section 32.4: Loading files repeatedly

The Kernel#load method will evaluate the code in the given file. The search path will be constructed as with **`require`**. It will re-evaluate that code on every subsequent call unlike **`require`**. There is no `load_relative`.

```
load `somefile`
```

Section 32.5: Loading several files

You can use any ruby technique to dynamically create a list of files to load. Illustration of globbing for files starting with `test`, loaded in alphabetical order.

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|

    require_relative source

end
```

第33章：线程

第33.1节：访问共享资源

使用互斥锁（mutex）来同步访问多个线程共享的变量：

```
计数器 = 0
计数器互斥锁 = Mutex.new

# 启动三个并行线程并递增计数器
3.times.map do |索引|
  Thread.new do
计数器互斥锁.synchronize { 计数器 += 1 }
  end
end.each(&:join) # 等待所有线程完成后再结束进程
```

否则，当前对一个线程可见的counter值可能会被另一个线程更改。

没有使用Mutex的示例（例如见Thread0，其中Before和After相差超过1）：

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before: #{counter}";
counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
[Thread 1] After: 3
[Thread 2] After: 1
```

使用Mutex的示例：

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize {
puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }
}.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

第33.2节：基本线程语义

可以使用Thread.new创建一个与主线程执行分离的新线程。

```
thr = Thread.new {
  sleep 1 # 子线程休眠1秒
  puts "这有什么大不了的"
}
```

这将自动启动新线程的执行。

要冻结主线程的执行，直到新线程停止，使用join：

```
thr.join #=> ... "这有什么大不了的"
```


Chapter 33: Thread

Section 33.1: Accessing shared resources

Use a mutex to synchronise access to a variable which is accessed from multiple threads:

```
counter = 0
counter_mutex = Mutex.new

# Start three parallel threads and increment counter
3.times.map do |index|
  Thread.new do
    counter_mutex.synchronize { counter += 1 }
  end
end.each(&:join) # Wait for all threads to finish before killing the process
```

Otherwise, the value of `counter` currently visible to one thread could be changed by another thread.

Example **without** `Mutex` (see e.g. `Thread 0`, where `Before` and `After` differ by more than 1):

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before: #{counter}";
counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
[Thread 1] After: 3
[Thread 2] After: 1
```

Example **with** `Mutex`:

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize {
puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }
}.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

Section 33.2: Basic Thread Semantics

A new thread separate from the main thread's execution, can be created using `Thread.new`.

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

This will automatically start the execution of the new thread.

To freeze execution of the main Thread, until the new thread stops, use `join`:

```
thr.join #=> ... "Whats the big deal"
```

请注意，当你调用join时，线程可能已经结束，在这种情况下执行将正常继续。
如果子线程从未被join，而主线程完成，子线程将不会执行任何剩余代码。

第33.3节：终止线程

线程在到达其代码块末尾时终止。提前终止线程的最佳方法是让它达到代码块的末尾。这样，线程可以在结束前运行清理代码。

此线程在实例变量continue为true时运行循环。将此变量设置为false，线程将自然终止：

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "我数到了 #{@count}，然后被残忍地打断了。"
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
```

第33.4节：如何终止线程

你可以使用 Thread.kill 或 Thread.terminate：

```
thr = Thread.new { ... }
Thread.kill(thr)
```


Note that the Thread may have already finished when you call join, in which case execution will continue normally.
If a sub-thread is never joined, and the main thread completes, the sub-thread will not execute any remaining code.

Section 33.3: Terminating a Thread

A thread terminates if it reaches the end of its code block. The best way to terminate a thread early is to convince it to reach the end of its code block. This way, the thread can run cleanup code before dying.

This thread runs a loop while the instance variable continue is true. Set this variable to false, and the thread will die a natural death:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
```

Section 33.4: How to kill a thread

You call use `Thread.kill` or `Thread.terminate`:

```
thr = Thread.new { ... }
Thread.kill(thr)
```

第34章：范围

第34.1节：作为序列的区间

范围最重要的用途是表示一个序列

语法：

```
(begin..end) => 该结构将包含 end 值
(begin...end) => 该结构将不包含 end 值
```

或

```
Range.new(begin,end,exclude_end) => exclude_end 默认值为 false
```

最重要的是 end 值必须大于 begin，否则将返回空。

示例：

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

第34.2节：遍历范围

你可以轻松地对范围内的每个元素执行操作。

```
(1..5).each do |i|
    print i
end
# 12345
```

第34.3节：日期范围

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "期间 #{date1.strftime("%d/%m/%Y")} 到 #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
  p date.strftime("%d/%m/%Y")
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```


Chapter 34: Range

Section 34.1: Ranges as Sequences

The most important use of ranges is to express a sequence

Syntax:

```
(begin..end) => this construct will include end value
(begin...end) => this construct will exclude end value
```

or

```
Range.new(begin,end,exclude_end) => exclude_end is by default false
```

Most important `end` value must be greater the `begin`, otherwise it will return nothing.

Examples:

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

Section 34.2: Iterating over a range

You can easily do something to each element in a range.

```
(1..5).each do |i|
    print i
end
# 12345
```

Section 34.3: Range between dates

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "Period #{date1.strftime("%d/%m/%Y")} to #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
  p date.strftime("%d/%m/%Y")
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```

第35章：模块

第35.1节：带有include的简单mixin

```
模块 SomeMixin
  def foo
    puts "foo!"
  end
end

类 Bar
  包含 SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # => "foo!"
# 由于混入而起作用
```

现在 Bar 是它自身方法和 SomeMixin 方法的混合体。

注意混入在类中的使用方式取决于它是如何被添加的：

- include关键字在类的上下文中评估模块代码（例如，方法定义将成为类实例的方法），
- extend将在对象的单例类上下文中评估模块代码（方法直接在扩展的对象上可用）。

第35.2节：模块与类的组合

你可以使用模块通过组合来构建更复杂的类。include ModuleName 指令将模块的方法合并到类中。

```
模块 Foo
  定义 foo_method
    输出 'foo_method 被调用！'
  结束
end

模块 Bar
  定义 bar_method
    输出 'bar_method 被调用！'
  结束
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```


Chapter 35: Modules

Section 35.1: A simple mixin with include

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  include SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # => "foo!"
# works thanks to the mixin
```

Now `Bar` is a mix of its own methods and the methods from `SomeMixin`.

Note that how a mixin is used in a class depends on how it is added:

- the `include` keyword evaluates the module code in the class context (eg. method definitions will be methods on instances of the class),
- `extend` will evaluate the module code in the context of the singleton class of the object (methods are available directly on the extended object).

Section 35.2: Modules and Class Composition

You can use Modules to build more complex classes through *composition*. The `include ModuleName` directive incorporates a module's methods into a class.

```
module Foo
  def foo_method
    puts 'foo_method called!'
  end
end

module Bar
  def bar_method
    puts 'bar_method called!'
  end
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```

Baz 现在包含了来自 Foo 和 Bar 的方法，除此之外还有它自己的方法。

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

第35.3节：模块作为命名空间

模块可以包含其他模块和类：

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # 现在可以通过以下方式访问 Foo：
    #
    Child::Foo

end # 模块 Namespace

# 现在必须通过以下方式访问 Foo：
#
Namespace::Child::Foo
```

第35.4节：使用 extend 的简单混入

混入只是一个可以添加（混入）到类中的模块。实现方式之一是使用 extend 方法。
extend 方法将混入模块的方法作为类方法添加。

```
模块 SomeMixin
  def foo
    puts "foo!"
  end
end

类 Bar
extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # NoMethodError，因为该方法未添加到实例中
Bar.foo        # => "foo!"
# 仅对类本身有效
```


Baz now contains methods from both Foo and Bar in addition to its own methods.

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

Section 35.3: Module as Namespace

Modules can contain other modules and classes:

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # Foo can now be accessed as:
    #
    Child::Foo

end # module Namespace

# Foo must now be accessed as:
#
Namespace::Child::Foo
```

Section 35.4: A simple mixin with extend

A mixin is just a module that can be added (mixed in) to a class. one way to do it is with the extend method. The extend method adds methods of the mixin as class methods.

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # NoMethodError, as the method was NOT added to the instance
Bar.foo        # => "foo!"
# works only on the class itself
```

第36章：Ruby中的自省

什么是内省？

内省是向内观察以了解内部。这是内省的简单定义。

在编程和Ruby中，内省是指能够在运行时查看对象、类，以了解它们的能力。

第36.1节：类的内省

以下是类的定义

class A def a; end end module B def b; end end class C < A include B def c; end end

C的实例方法有哪些？

C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]

仅在C上声明的实例方法有哪些？

C.instance_methods(false) # [:c]

C类的祖先有哪些？

C.ancestors # [C, B, A, Object,...]

Superclass of C?

C.superclass # A

第36.2节：让我们来看一些例子

示例：

```
s = "Hello"  # s 是一个字符串
```

然后我们来了解一下 s。开始吧：

所以你想知道 s 在运行时的类是什么？

```
irb(main):055:0* s.class
=> String
```

哦，好。那 s 有哪些方法呢？

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
```


Chapter 36: Introspection in Ruby

What is introspection?

Introspection is looking inward to know about the inside. That is a simple definition of introspection.

In programming and Ruby in general…introspection is the ability to look at object, class… at run time to know about that one.

Section 36.1: Introspection of class

Lets following are the class definition

class A def a; end end module B def b; end end class C < A include B def c; end end

What are the instance methods of C?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

What are the instance methods that declare only on C?

```
C.instance_methods(false) # [:c]
```

What are the ancestors of class C?

```
C.ancestors # [C, B, A, Object,...]
```

Superclass of C?

```
C.superclass # A
```

Section 36.2: Lets see some examples

Example:

```
s = "Hello"  # s is a string
```

Then we find out something about s. Lets begin:

So you want to know what is the class of s at run time?

```
irb(main):055:0* s.class
=> String
```

Ohh, good. But what are the methods of s?

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
```

```
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :is_a?, :extend,
:to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method, :public_method,
:singleton_method, :define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup,
:itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods,
:frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```

你想知道 s 是否是 String（字符串）类的实例吗？

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

s 的公共方法有哪些？

```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :pretty_print, :pretty_print_cycle,
:pretty_print_instance_variables, :pretty_print_inspect, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :pretty_inspect,
:is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method,
:public_method, :singleton_method, :define_singleton_method, :nil?, :class, :singleton_class,
:clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods,
:protected_methods, :frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?,
:instance_eval, :instance_exec, :__id__]
```

和私有方法....

```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding, :sprintf,
:format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop, :block_given?, :Complex,
:set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational, :caller, :caller_locations,
:select, :test, :fork, :exit, :`, :gem_original_require, :sleep, :pp, :respond_to_missing?, :load,
:exec, :exit!, :system, :spawn, :abort, :syscall, :printf, :open, :putc, :print, :readline, :puts,
:p, :srand, :readlines, :gets, :rand, :proc, :lambda, :trap, :initialize_clone, :initialize_dup,
:gem, :require, :require_relative, :autoload, :autoload?, :binding, :local_variables, :warn,
:raise, :fail, :global_variables, :__method__, :__callee__, :__dir__, :eval, :iterator?,
:method_missing, :singleton_method_added, :singleton_method_removed, :singleton_method_undefined]
```

是的，s 确实有一个方法名为 upper。你想获取 s 的大写版本吗？我们试试：

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

看起来没有，正确的方法是 upcase，我们来检查一下：


```
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :is_a?, :extend,
:to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method, :public_method,
:singleton_method, :define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup,
:itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods,
:frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```


You want to know if s is an instance of String?


```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```


What are the public methods of s?


```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :pretty_print, :pretty_print_cycle,
:pretty_print_instance_variables, :pretty_print_inspect, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :pretty_inspect,
:is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method,
:public_method, :singleton_method, :define_singleton_method, :nil?, :class, :singleton_class,
:clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods,
:protected_methods, :frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?,
:instance_eval, :instance_exec, :__id__]
```


and private methods....


```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding, :sprintf,
:format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop, :block_given?, :Complex,
:set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational, :caller, :caller_locations,
:select, :test, :fork, :exit, :`, :gem_original_require, :sleep, :pp, :respond_to_missing?, :load,
:exec, :exit!, :system, :spawn, :abort, :syscall, :printf, :open, :putc, :print, :readline, :puts,
:p, :srand, :readlines, :gets, :rand, :proc, :lambda, :trap, :initialize_clone, :initialize_dup,
:gem, :require, :require_relative, :autoload, :autoload?, :binding, :local_variables, :warn,
:raise, :fail, :global_variables, :__method__, :__callee__, :__dir__, :eval, :iterator?,
:method_missing, :singleton_method_added, :singleton_method_removed, :singleton_method_undefined]
```


Yes, do s have a method name upper. You want to get the upper case version of s? Lets try:


```
irb(main):044:0> s.respond_to?(:upper)
=> false
```


Look like not, the correct method is upcase lets check:

```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
=> true
```



```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
=> true
```

第37章：Ruby中的猴子补丁

猴子补丁是一种修改和扩展Ruby中类的方法。基本上，你可以修改已经定义的Ruby类，添加新方法，甚至修改之前定义的方法。

第37.1节：修改现有的Ruby方法

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

第37.2节：猴子补丁类

猴子补丁是指在类本身之外修改类或对象。

有时添加自定义功能是有用的。

示例： 重写字符串类以提供布尔值解析功能

```
类 String
  定义 to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

如你所见，我们向字符串类添加了to_b()方法，这样我们就可以将任何字符串解析为布尔值。

```
>> 'true'.to_b
=> true
>> 'foo bar'.to_b
=> false
```

第37.3节：对对象进行猴子补丁

与类的补丁类似，你也可以补丁单个对象。不同之处在于，只有该实例可以使用新方法。

示例：重写字符串对象以提供布尔值解析

```
s = '真'
t = '假'

def s.to_b
  self =~ /true/ ? true : false
end

>>  s.to_b
=> 真
>>  t.to_b
=> 未定义方法 `to_b' 用于 "false":String (NoMethodError)
```


Chapter 37: Monkey Patching in Ruby

Monkey Patching is a way of modifying and extending classes in Ruby. Basically, you can modify already defined classes in Ruby, adding new methods and even modifying previously defined methods.

Section 37.1: Changing an existing ruby method

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

Section 37.2: Monkey patching a class

Monkey patching is the modification of classes or objects outside of the class itself.

Sometimes it is useful to add custom functionality.

Example: Override String Class to provide parsing to boolean

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

As you can see, we add the `to_b()` method to the String class, so we can parse any string to a boolean value.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

Section 37.3: Monkey patching an object

Like patching of classes, you can also patch single objects. The difference is that only that one instance can use the new method.

Example: Override a string object to provide parsing to boolean

```
s = 'true'
t = 'false'

def s.to_b
  self =~ /true/ ? true : false
end

>> s.to_b
=> true
>> t.to_b
=> undefined method `to_b' for "false":String (NoMethodError)
```

第37.4节：使用Refinements进行安全的猴子补丁

自Ruby 2.0起，Ruby允许通过refinements实现更安全的猴子补丁。基本上，它允许限制猴子补丁代码仅在被请求时才应用。

首先我们在一个模块中创建一个refinement：

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

然后我们可以决定在哪里使用它：

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end

  def reverse
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

第37.5节：带参数的方法修改

您可以访问与您重写的方法完全相同的上下文。

```
class Boat
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"
```


Section 37.4: Safe Monkey patching with Refinements

Since Ruby 2.0, Ruby allows to have safer Monkey Patching with refinements. Basically it allows to limit the Monkey Patched code to only apply when it is requested.

First we create a refinement in a module:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Then we can decide where to use it:

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end

  def reverse
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

Section 37.5: Changing a method with parameters

You can access the exact same context as the method you override.

```
class Boat
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"
```

```ruby
class Boat
  def name
    " #{@name} "
  end
end

puts Boat.new("Moat").name # => " Moat "
```

第37.6节：添加功能

你可以向Ruby中的任何类添加方法，无论它是内置的还是自定义的。调用对象通过self引用。

```ruby
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

第37.7节：更改任何方法

```ruby
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

第37.8节：扩展现有类

```ruby
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```



```ruby
class Boat
  def name
    "□ #{@name} □"
  end
end

puts Boat.new("Moat").name # => "□ Moat □"
```

Section 37.6: Adding Functionality

You can add a method to any class in Ruby, whether it's a builtin or not. The calling object is referenced using **self**.

```ruby
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

Section 37.7: Changing any method

```ruby
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

Section 37.8: Extending an existing class

```ruby
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

第38章：Ruby中的递归

第38.1节：尾递归

许多递归算法可以用迭代来表达。例如，最大公约数函数可以递归地编写：

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

或者用迭代方式：

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

这两种算法在理论上是等价的，但递归版本有可能引发SystemStackError。然而，由于递归方法以调用自身结束，它可以被优化以避免栈溢出。换句话说：如果编译器能够识别方法末尾的递归调用，递归算法可以生成与迭代算法相同的机器代码。Ruby默认不进行尾调用优化，但你可以开启它，方法如下：

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
trace_instruction: false
}
```

除了开启尾调用优化外，你还需要关闭指令跟踪。不幸的是，这些选项仅在编译时生效，因此你需要从另一个文件中require递归方法，或者使用 eval执行方法定义：

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
EOF
me_myself_and_i # 无限循环，不是栈溢出
```

最后，最终的返回调用必须返回方法本身且*仅返回方法本身*。这意味着你需要重写
标准的阶乘函数：

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

类似于：

```
def fact(x, acc=1)
```


Chapter 38: Recursion in Ruby

Section 38.1: Tail recursion

Many recursive algorithms can be expressed using iteration. For instance, the greatest common denominator function can be written recursively:

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

or iteratively:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

The two algorithms are equivalent in theory, but the recursive version risks a SystemStackError. However, since the recursive method ends with a call to itself, it could be optimized to avoid a stack overflow. Another way to put it: the recursive algorithm can result in the same machine code as the iterative *if* the compiler knows to look for the recursive method call at the end of the method. Ruby doesn't do tail call optimization by default, but you can turn it on with:

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

In addition to turning on tail-call optimization, you also need to turn off instruction tracing. Unfortunately, these options only apply at compile time, so you either need to **`require`** the recursive method from another file or **`eval`** the method definition:

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
EOF
me_myself_and_i # Infinite loop, not stack overflow
```

Finally, the final return call must return the method and *only the method*. That means you'll need to re-write the standard factorial function:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

To something like:

```
def fact(x, acc=1)
```

```
  return acc if x <= 1
  return fact(x-1, x*acc)
end
```

此版本通过第二个（可选）参数传递累积和，默认值为1。

进一步阅读：Ruby中的尾调用优化和Tailin' Ruby。

第38.2节：递归函数

让我们从一个简单的算法开始，看看如何在Ruby中实现递归。

一家面包店有产品出售。产品以包装形式存在。它只按包装数量处理订单。包装从
最大包装尺寸开始，然后用下一个可用包装尺寸填充剩余数量。

例如，如果收到16的订单，面包店分配2个5包装和2个3包装。*25+23* = 16。让我们看看
这是如何用递归实现的。“allocate”是这里的递归函数。

```
#!/usr/bin/ruby

class Bakery
attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # 包装尺寸为5和3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ============================================
    # 包装按大包装优先顺序分配
    # 以最小化包装空间
    # ============================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

剩余数量
  end
end

bakery = Bakery.new
bakery.allocate(16)
puts "包装组合是: #{bakery.selected_packs.inspect}"
```



```
  return acc if x <= 1
  return fact(x-1, x*acc)
end
```

This version passes the accumulated sum via a second (optional) argument that defaults to 1.

Further reading: Tail Call Optimization in Ruby and Tailin' Ruby.

Section 38.2: Recursive function

Let's start with a simple algorithm to see how recursion could be implemented in Ruby.

A bakery has products to sell. Products are in packs. It services orders in packs only. Packaging starts from the largest pack size and then the remaining quantities are filled by next pack sizes available.

For e.g. If an order of 16 is received, bakery allocates 2 from 5 pack and 2 from 3 pack. *25+23* = 16. Let's see how this is implemented in recursion. "allocate" is the recursive function here.

```
#!/usr/bin/ruby

class Bakery
  attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # pack sizes 5 and 3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ============================================
    # packs are allocated in large packs first order
    # to minimize the packaging space
    # ============================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
        ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

    remaining_qty
  end
end

bakery = Bakery.new
bakery.allocate(16)
puts "Pack combination is: #{bakery.selected_packs.inspect}"
```

输出是:

> 包装组合是: [3, 3, 5, 5]

Output is:

> Pack combination is: [3, 3, 5, 5]

第39章：展开运算符（*）

第39.1节：可变数量的参数

展开运算符将数组的各个元素拆开，变成一个列表。这通常用于创建一个接受可变数量参数的方法：

```
# 第一个参数是主体，后面的参数是他们的配偶
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

注意，数组在列表中只算作一个项目，因此如果你想传递一个数组，调用时也需要使用展开运算符（splat operator）：

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

第39.2节：将数组强制转换为参数列表

假设你有一个数组：

```
pair = ['Jack','Jill']
```

以及一个接受两个参数的方法：

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

你可能会认为你可以直接传递数组：

```
print_pair(pair) # 参数数量错误（1 个，期望 2 个）(ArgumentError)
```

由于数组只是一个参数，而不是两个，所以 Ruby 抛出异常。你可以单独取出每个元素：

```
print_pair(pair[0], pair[1])
```

或者你可以使用展开运算符来省点力气：

```
print_pair(*pair)
```


Chapter 39: Splat operator (*)

Section 39.1: Variable number of arguments

The splat operator removes individual elements of an array and makes them into a list. This is most commonly used to create a method that accepts a variable number of arguments:

```
# First parameter is the subject and the following parameters are their spouses
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

Notice that an array only counts as one item on the list, so you will need to us the splat operator on the calling side too if you have an array you want to pass:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Section 39.2: Coercing arrays into parameter list

Suppose you had an array:

```
pair = ['Jack','Jill']
```

And a method that takes two arguments:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

You might think you could just pass the array:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Since the array is just one argument, not two, so Ruby throws an exception. You *could* pull out each element individually:

```
print_pair(pair[0], pair[1])
```

Or you can use the splat operator to save yourself some effort:

```
print_pair(*pair)
```

第40章：Ruby中的JSON

第40.1节：在Ruby中使用JSON

JSON（JavaScript对象表示法）是一种轻量级的数据交换格式。许多网络应用使用它来发送和接收数据。

在Ruby中，你可以轻松处理JSON。

首先你必须require'json'，然后你可以通过JSON.parse()命令解析JSON字符串。

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

这里发生的情况是，解析器将 JSON 生成了一个 Ruby 哈希。

反过来，从 Ruby 哈希生成 JSON 和解析一样简单。首选的方法是 to_json:

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

第40.2节：使用符号

你可以将 JSON 与 Ruby 符号一起使用。通过为解析器设置 symbolize_names 选项，生成的哈希中的键将是符号而不是字符串。

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```


Chapter 40: JSON with Ruby

Section 40.1: Using JSON with Ruby

JSON (JavaScript Object Notation) is a lightweight data interchange format. Many web applications use it to send and receive data.

In Ruby you can simply work with JSON.

At first you have to `require 'json'`, then you can parse a JSON string via the `JSON.parse()` command.

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

What happens here, is that the parser generates a Ruby Hash out of the JSON.

The other way around, generating JSON out of a Ruby hash is as simple as parsing. The method of choice is `to_json`:

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

Section 40.2: Using Symbols

You can use JSON together with Ruby symbols. With the option symbolize_names for the parser, the keys in the resulting hash will be symbols instead of strings.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

第41章：纯RSpec JSON API测试

第41.1节：测试序列化器对象并将其引入控制器

假设你想构建符合jsonapi.org规范的API，结果应如下所示：

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

序列化器对象的测试可能如下所示：

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
article: be_kind_of(Hash)
      })
    end

context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
expect(article_hash).to match({
          id: article.id.to_s,
type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

context 'attributes' do
let(:article_hash_attributes) { article_hash.fetch(:attributes) }

        it do
expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
          })
        end
      end
    end
  end
```


Chapter 41: Pure RSpec JSON API testing

Section 41.1: Testing Serializer object and introducing it to Controller

Let say you want to build your API to comply jsonapi.org specification and the result should look like:

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

Test for Serializer object may look like this:

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
  subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
    let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
        article: be_kind_of(Hash)
      })
    end

    context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
        expect(article_hash).to match({
          id: article.id.to_s,
          type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

      context 'attributes' do
        let(:article_hash_attributes) { article_hash.fetch(:attributes) }

        it do
          expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
          })
        end
      end
    end
  end
end
```

```
end
```

序列化器对象可能看起来像这样：

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

当我们运行“序列化器”测试时，所有测试都通过了。

这相当无聊。让我们在文章序列化器中引入一个拼写错误：不是返回 type: "articles"，而是返回 type: "events" 并重新运行我们的测试。

```
rspec spec/serializers/article_serializer_spec.rb

.F.

失败:

  1) ArticleSerializer#as_json 文章哈希应包含类型和ID
     失败/错误:
expect(article_hash).to match({
         id: article.id.to_s,
type: 'articles',
         attributes: be_kind_of(Hash)
       })

预期 {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} 匹配 {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
差异:

@@ -1,4 +1,4 @@
        -:attributes => (是某种 Hash),
        +:attributes => {:title=>"Bring Me The Horizon"},
         :id => "678",
        -:type => "articles",
        +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```



```
end
```

Serializer object may look like this:

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
      article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

When we run our "serializers" specs everything passes.

That's pretty boring. Let's introduce a typo to our Article Serializer: Instead of `type: "articles"` let's return `type: "events"` and rerun our tests.

```
rspec spec/serializers/article_serializer_spec.rb

.F.

Failures:

  1) ArticleSerializer#as_json article hash should contain type and id
     Failure/Error:
       expect(article_hash).to match({
         id: article.id.to_s,
         type: 'articles',
         attributes: be_kind_of(Hash)
       })

       expected {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} to match {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
       Diff:

       @@ -1,4 +1,4 @@
       -:attributes => (be a kind of Hash),
       +:attributes => {:title=>"Bring Me The Horizon"},
        :id => "678",
       -:type => "articles",
       +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```

运行测试后，很容易发现错误。

修正错误（将类型改为article）后，可以这样将其引入控制器：

```
# app/controllers/v2/articles_controller.rb
模块 V2
  类 ArticlesController < ApplicationController
    def show
渲染 json: serializer.as_json
    end

私有
      定义 article
        @article ||= Article.find(params[:id])
      结束

      定义 serializer
        @serializer ||= ArticleSerializer.new(article)
      结束
  end
end
```

此示例基于文章: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Once you've run the test it's pretty easy to spot the error.

Once you fix the error (correct the type to be `article`) you can introduce it to Controller like this:

```
# app/controllers/v2/articles_controller.rb
module V2
  class ArticlesController < ApplicationController
    def show
      render json: serializer.as_json
    end

    private
      def article
        @article ||= Article.find(params[:id])
      end

      def serializer
        @serializer ||= ArticleSerializer.new(article)
      end
  end
end
```

This example is based on article: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

第42章：Gem 创建/管理

第42.1节：Gemspec文件

每个 gem 都有一个格式为 `<gem name>.gemspec` 的文件，其中包含有关该 gem 及其文件的元数据。gemspec 的格式如下：

```
Gem::Specification.new do |s|
  # 有关 gem 的详细信息。它们以以下格式添加：
s.<detail name> = <detail value>
end
```

RubyGems 要求的字段有：

`要么是 author = string，要么是 authors = array`

如果只有一个作者，使用 author =；如果有多个作者，使用 authors =。对于 authors=，使用一个列出作者姓名的数组。

```
files = array
```

这里的 array 是 gem 中所有文件的列表。也可以与 Dir[] 函数一起使用，例如，如果所有文件都在 /lib/ 目录下，则可以使用 files = Dir["/lib/"]。

```
name = string
```

这里的 string 就是你的 gem 的名称。RubyGems 建议在命名 gem 时遵循一些规则。

1. 使用下划线，禁止空格
2. 仅使用小写字母
3. 对于 gem 扩展使用连字符（例如，如果你的 gem 名称为 example，扩展名应命名为 `example-extension`），这样在需要扩展时可以通过 **require** `"example/extension"` 来加载。

RubyGems 还补充道："如果你在 rubygems.org 上发布的 gem 名称令人反感、侵犯知识产权或 gem 内容符合这些标准，可能会被移除。你可以在 RubyGems 支持网站举报此类 gem。"

```
platform=
```

我不知道

```
require_paths=
```

我不知道

```
summary= 字符串
```

字符串是对 gem 目的的简要说明，以及你想分享的关于该 gem 的任何信息。

```
版本= 字符串
```


Chapter 42: Gem Creation/Management

Section 42.1: Gemspec Files

Each gem has a file in the format of **<gem** name**>**.gemspec which contains metadata about the gem and it's files. The format of a gemspec is as follows:

```
Gem::Specification.new do |s|
  # Details about gem. They are added in the format:
  s.<detail name> = <detail value>
end
```

The fields required by RubyGems are:

Either `author = string` or `authors = array`

Use `author =` if there is only one author, and `authors =` when there are multiple. For `authors=` use an array which lists the authors names.

```
files = array
```

Here **array** is a list of all the files in the gem. This can also be used with the **Dir[]** function, for example if all your files are in the **/**lib**/** directory, then you can use `files = Dir["/lib/"]`.

```
name = string
```

Here string is just the name of your gem. Rubygems recommends a few rules you should follow when naming your gem.

1. Use underscores, NO SPACES
2. Use only lowercase letters
3. Use hypens for gem extension (e.g. if your gem is named `example` for an extension you would name it `example-extension`) so that when then extension is required it can be required as **require** `"example/extension"`.

RubyGems also adds "If you publish a gem on rubygems.org it may be removed if the name is objectionable, violates intellectual property or the contents of the gem meet these criteria. You can report such a gem on the RubyGems Support site."

```
platform=
```

I don't know

```
require_paths=
```

I don't know

```
summary= string
```

String is a summery of the gems purpose and anything that you would like to share about the gem.

```
version= string
```

当前版本号。

推荐的字段有：

```
email = string
```

与该 gem 关联的电子邮件地址。

```
homepage= string
```

该 gem 所在的网站。

要么是 `license=` 要么是 `licenses=`

我不知道

第 42.2 节：构建一个 Gem

创建好你的 gem 后，要发布它需要遵循几个步骤：

1. 使用 gem build <gem name>.gemspec 构建你的 gem（gemspec 文件必须存在）
2. 如果您还没有 RubyGems 账户，请创建一个 here
3. 确认没有与您的 gem 名称相同的 gem 存在
4. 使用 `gem publish <gem 名称>.<gem 版本号>.gem` 发布您的 gem

第42.3节：依赖关系

列出依赖树：

```
gem dependency
```

列出哪些 gem 依赖于特定的 gem（例如 bundler）

```
gem dependency bundler --reverse-dependencies
```


The current version number of the gem.

The recommended fields are:

```
email = string
```

An email address that will be associated with the gem.

```
homepage= string
```

The website where the gem lives.

Either `license=` or `licenses=`

I don't know

Section 42.2: Building A Gem

Once you have created your gem to publish it you have to follow a few steps:

1. Build your gem with `gem build <gem name>.gemspec` (the gemspec file must exist)
2. Create a RubyGems account if you do not already have one here
3. Check to make sure that no gems exist that share your gems name
4. Publish your gem with `gem publish <gem name>.<gem version number>.gem`

Section 42.3: Dependencies

To list the dependency tree:

```
gem dependency
```

To list which gems depend on a specific gem (bundler for example)

```
gem dependency bundler --reverse-dependencies
```

第43章：rbenv

第43.1节：卸载 Ruby

卸载特定版本的 Ruby 有两种方法。最简单的方法是直接删除目录 ~/.rbenv/versions:

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

或者，你可以使用卸载命令，它的作用完全相同：

```
$ rbenv uninstall 2.1.0
```

如果该版本正在某处被使用，你需要更新你的全局或本地版本。要恢复到路径中第一个版本（通常是系统提供的默认版本），请使用：

```
$ rbenv global system
```

第43.2节：使用rbenv安装和管理Ruby版本

使用rbenv安装和管理各种Ruby版本最简单的方法是使用ruby-build插件。

首先将rbenv仓库克隆到你的主目录：

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

然后克隆ruby-build插件：

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

确保在你的 shell 会话中初始化了 rbenv，通过将以下内容添加到你的 .bash_profile 或 .zshrc 文件中：

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

（这实际上是先检查 rbenv 是否可用，然后初始化它）。

你可能需要重启你的 shell 会话——或者简单地打开一个新的终端窗口。

注意： 如果你使用的是 OSX，还需要通过以下命令安装 Mac OS 命令行工具：

```
$ xcode-select --install
```

你也可以使用 rbenv 通过 Homebrew 安装，而不是从源码构建：

```
$ brew update
$ brew install rbenv
```

然后按照以下提示操作：

```
$ rbenv init
```


Chapter 43: rbenv

Section 43.1: Uninstalling a Ruby

There are two ways to uninstall a particular version of Ruby. The easiest is to simply remove the directory from `~/.rbenv/versions`:

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

Alternatively, you can use the uninstall command, which does exactly the same thing:

```
$ rbenv uninstall 2.1.0
```

If this version happens to be in use somewhere, you'll need to update your global or local version. To revert to the version that's first in your path (usually the default provided by your system) use:

```
$ rbenv global system
```

Section 43.2: Install and manage versions of Ruby with rbenv

The easiest way to install and manage various versions of Ruby with rbenv is to use the ruby-build plugin.

First clone the rbenv repository to your home directory:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Then clone the ruby-build plugin:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Ensure that rbenv is initialized in your shell session, by adding this to your `.bash_profile` or `.zshrc`:

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(This essentially first checks if `rbenv` is available, and initializes it).

You will probably have to restart your shell session - or simply open a new Terminal window.

Note: If you're running on OSX, you will also need to install the Mac OS Command Line Tools with:

```
$ xcode-select --install
```

You can also install `rbenv` using Homebrew instead of building from the source:

```
$ brew update
$ brew install rbenv
```

Then follow the instructions given by:

```
$ rbenv init
```

安装一个新的 Ruby 版本：

列出可用的版本：

```
$ rbenv install --list
```

选择一个版本并安装：

```
$ rbenv install 2.2.0
```

将已安装的版本标记为全局版本——即系统默认使用的版本：

```
$ rbenv global 2.2.0
```

检查当前的全局版本：

```
$ rbenv global
=> 2.2.0
```

您可以通过以下命令指定本地项目版本：

```
$ rbenv local 2.1.2
=> (在当前目录创建一个指定版本的 .ruby-version 文件)
```

脚注：

[1]: 理解 PATH

Install a new version of Ruby:

List the versions available with:

```
$ rbenv install --list
```

Choose a version and install it with:

```
$ rbenv install 2.2.0
```

Mark the installed version as the global version - i.e. the one that your system uses by default:

```
$ rbenv global 2.2.0
```

Check what your global version is with:

```
$ rbenv global
=> 2.2.0
```

You can specify a local project version with:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

Footnotes:

[1]: Understanding PATH

第44章：Gem 使用

第44.1节：安装 ruby gems

本指南假设您已经安装了 Ruby。如果您使用的是 Ruby < 1.9，则需要手动安装 RubyGems，因为它不会被原生包含。

要安装 ruby gem，请输入命令：

```
gem install [gemname]
```

如果你正在处理一个包含 gem 依赖列表的项目，那么这些依赖会被列在一个名为Gemfile的文件中。要在项目中安装一个新的 gem，请在Gemfile中添加以下代码行：

```
gem 'gemname'
```

这个Gemfile被Bundler gem用来安装项目所需的依赖，不过这也意味着你需要先安装 Bundler，方法是运行（如果你还没有安装的话）：

```
gem install bundler
```

保存文件，然后运行命令：

```
bundle install
```

指定版本

版本号可以在命令行中通过-v标志指定，例如：

```
gem install gemname -v 3.14
```

在Gemfile中指定版本号时，你有多种选项可用：

- 未指定版本（gem 'gemname'） -- 将安装与Gemfile中其他gem兼容的最新版本。
- 指定精确版本（gem 'gemname'，'3.14'） -- 只会尝试安装版本3.14（如果与Gemfile中其他gem不兼容则安装失败）。
- 乐观的最低版本号（gem 'gemname'，'>=3.14'） -- 只会尝试安装与Gemfile中其他gem兼容的最新版本，且如果没有大于或等于3.14的兼容版本则安装失败。操作符>也可使用。
- 悲观的最低版本号（gem 'gemname'，'~>3.14'） -- 功能上等同于使用gem 'gemname'，'>=3.14'，'<4'。换句话说，只有最后一位小数点后的数字允许增加。

最佳实践：你可能想使用像 rbenv或 rvm这样的Ruby版本管理库。
通过这些库，你可以相应地安装不同版本的Ruby运行时和gem。因此，在项目中工作时，这尤其方便，因为大多数项目都是针对已知的Ruby版本编写的。

第44.2节：从github/文件系统安装Gem

你可以从github或文件系统安装gem。如果gem已经从git检出或以某种方式已经存在于文件系统中，你可以使用以下命令安装

Chapter 44: Gem Usage

Section 44.1: Installing ruby gems

This guide assumes you already have Ruby installed. If you're using Ruby < `1.9` you'll have to manually install RubyGems as it won't be included natively.

To install a ruby gem, enter the command:

```
gem install [gemname]
```

If you are working on a project with a list of gem dependencies, then these will be listed in a file named `Gemfile`. To install a new gem in the project, add the following line of code in the `Gemfile`:

```
gem 'gemname'
```

This `Gemfile` is used by the Bundler gem to install dependencies your project requires, this does however mean that you'll have to install Bundler first by running (if you haven't already):

```
gem install bundler
```

Save the file, and then run the command:

```
bundle install
```

Specifying versions

The version number can be specified on the command live, with the `-v` flag, such as:

```
gem install gemname -v 3.14
```

When specifying version numbers in a `Gemfile`, you have several options available:

- No version specified (`gem 'gemname'`) -- Will install the *latest* version which is compatible with other gems in the `Gemfile`.
- Exact version specified (`gem 'gemname', '3.14'`) -- Will only attempt to install version `3.14` (and fail if this is incompatible with other gems in the `Gemfile`).
- **Optimistic** minimum version number (`gem 'gemname', '>=3.14'`) -- Will only attempt to install the *latest* version which is compatible with other gems in the `Gemfile`, and fails if no version greater than or equal to `3.14` is compatible. The operator `>` can also be used.
- **Pessimistic** minimum version number (`gem 'gemname', '~>3.14'`) -- This is functionally equivalent to using `gem 'gemname', '>=3.14', '<4'`. In other words, only the number after the *final period* is permitted to increase.

As a best practice: You might want to use one of the Ruby version management libraries like rbenv or rvm.
Through these libraries, you can install different versions of Ruby runtimes and gems accordingly. So, when working in a project, this will be especially handy because most of the projects are coded against a known Ruby version.

Section 44.2: Gem installation from github/filesystem

You can install a gem from github or filesystem. If the gem has been checked out from git or somehow already on the file system, you could install it using

```
gem install --local path_to_gem/filename.gem
```

从github安装gem。从github下载源码

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

构建 gem

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

第44.3节：从代码中检查是否安装了所需的 gem

要从代码中检查是否安装了所需的 gem，可以使用以下方法（以 nokogiri 为例）：

```
begin
found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <你的其余代码>
rescue Gem::LoadError
end
```

但是，这可以进一步扩展为一个函数，用于在代码中设置功能。

```
def gem_installed?(gem_name)
  found_gem = false
  begin
found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
     return false
  else
    return true
  end
end
```

现在你可以检查所需的 gem 是否已安装，并打印错误信息。

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required"exit 1

end
```

或

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```



```
gem install --local path_to_gem/filename.gem
```

Installing gem from github. Download the sources from github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Build the gem

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

Section 44.3: Checking if a required gem is installed from within code

To check if a required gem is installed, from within your code, you can use the following (using nokogiri as an example):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
rescue Gem::LoadError
end
```

However, this can be further extended to a function that can be used in setting up functionality within your code.

```
def gem_installed?(gem_name)
  found_gem = false
  begin
    found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
     return false
  else
    return true
  end
end
```

Now you can check if the required gem is installed, and print an error message.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

or

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

第44.4节：使用 Gemfile 和 Bundler

Gemfile 是组织应用程序依赖项的标准方式。一个基本的 Gemfile 看起来像这样：

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

您可以按如下方式指定所需的 gem 版本：

```
# 匹配除点版本外的版本。仅使用 1.5.X
gem 'rack', '~>1.5.2'
# 使用特定版本。
gem 'sinatra', '1.4.7'
# 使用至少该版本或更高版本。
gem 'uglifier', '>= 1.3.0'
```

您也可以直接从 git 仓库拉取 gems：

```
# 从 GitHub 拉取一个 gem
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# 你可以指定一个 sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# 你也可以指定一个分支，尽管这通常不安全
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

你也可以根据用途对 gems 进行分组。例如：

```
group :development, :test do
    # 这个 gem 只在开发和测试环境可用，生产环境不可用。
gem 'byebug'
end
```

如果您的应用需要能够在多个平台上运行，您可以指定某些 gem 应该在哪些平台上运行。例如：

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

要安装 Gemfile 中的所有 gem，请执行：

```
gem install bundler
bundle install
```

第44.5节：Bundler/inline（bundler v1.10及以后版本）

有时您需要为某人编写脚本，但不确定他机器上安装了什么。您的脚本所需的所有内容都在吗？不用担心。Bundler 有一个很棒的功能叫做 inline。

Section 44.4: Using a Gemfile and Bundler

A `Gemfile` is the standard way to organize dependencies in your application. A basic Gemfile will look like this:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

You can specify the versions of the gem you want as follows:

```
# Match except on point release. Use only 1.5.X
gem 'rack', '~>1.5.2'
# Use a specific version.
gem 'sinatra', '1.4.7'
# Use at least a version or anything greater.
gem 'uglifier', '>= 1.3.0'
```

You can also pull gems straight from a git repo:

```
# pull a gem from github
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# you can specify a sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# you can also specify a branch, though this is often unsafe
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

You can also group gems depending on what they are used for. For example:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

You can specify which platform certain gems should run on if you application needs to be able to run on multiple platforms. For example:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

To install all the gems from a Gemfile do:

```
gem install bundler
bundle install
```

Section 44.5: Bundler/inline (bundler v1.10 and later)

Sometimes you need to make a script for someone but you are not sure what he has on his machine. Is there everything that your script needs? Not to worry. Bundler has a great function called in line.

它提供了一个 gemfile 方法，在脚本运行之前会下载并加载所有必要的 gem。举个小例子：

```
require 'bundler/inline' #只加载所需内容

#启动 bundler，并在其中使用您已经熟悉的语法
gemfile(true) do
source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```


It provides a `gemfile` method and before the script is run it downloads and requires all the necessary gems. A little example:

```
require 'bundler/inline' #require only what you need

#Start the bundler and in it use the syntax you are already familiar with
gemfile(true) do
  source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```

第45章：单例类

第45.1节：介绍

Ruby 有三种类型的对象：

- 类和模块，它们是 Class 类或 Module 类的实例。
- 类的实例。
- 单例类。

每个对象都有一个包含其方法的类：

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

对象本身不能包含方法，只有它们的类可以。但通过单例类，可以向任何对象（包括其他单例类）添加方法。

```
def object.foo
  :foo
end
object.foo #=> :foo
```

foo 定义在 singleton class 的 object 上。其他 Example 实例无法响应 foo。

Ruby 会按需创建 singleton class。访问它们或向它们添加方法会促使 Ruby 创建它们。

第45.2节：Singleton Class 的继承

子类化也会对子类的 Singleton Class 进行子类化

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.单例类.超类 #=> #<Class:Example>
```

扩展或包含模块不会扩展单例类

```
module ExampleModule
end
```


Chapter 45: Singleton Class

Section 45.1: Introduction

Ruby has three types of objects:

- Classes and modules which are instances of class Class or class Module.
- Instances of classes.
- Singleton Classes.

Each object has a class which contains its methods:

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

Objects themselves can't contain methods, only their class can. But with singleton classes, it is possible to add methods to any object including other singleton classes.

```
def object.foo
  :foo
end
object.foo #=> :foo
```

`foo` is defined on singleton class of `object`. Other `Example` instances can not reply to `foo`.

Ruby creates singleton classes on demand. Accessing them or adding methods to them forces Ruby to create them.

Section 45.2: Inheritance of Singleton Class

Subclassing also Subclasses Singleton Class

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.singleton_class.superclass #=> #<Class:Example>
```

Extending or Including a Module does not Extend Singleton Class

```
module ExampleModule
end
```

```ruby
def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

第45.3节：单例类

所有对象都是某个类的实例。然而，事实并非如此简单。在Ruby中，每个对象还有一个有些隐藏的*单例类*。

这使得可以在单个对象上定义方法。单例类位于对象本身和其实际类之间，因此所有定义在单例类上的方法都只对该对象可用，仅限该对象。

```ruby
object = Object.new

def object.exclusive_method
  '只有此对象会响应此方法'
end

object.exclusive_method
# => "只有这个对象会响应这个方法"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

上面的例子也可以用define_singleton_method来写：

```ruby
object.define_singleton_method :exclusive_method do
  "该方法实际上定义在对象的单例类中"
end
```

这与在object的singleton_class上定义方法是一样的：

```ruby
# 使用send是因为define_method是私有方法
object.singleton_class.send :define_method, :exclusive_method do
  "现在我们直接在单例类上定义了一个实例方法"
end
```

在singleton_class成为Ruby核心API的一部分之前，单例类被称为*metaclasses*，可以通过以下惯用法访问：

```ruby
class << object
  self  # 指对象的单例类
end
```

第45.4节：使用单例类进行消息传播

实例从不包含方法，它们只携带数据。然而，我们可以为任何对象定义一个单例类，包括类的实例。

当消息传递给一个对象（调用方法）时，Ruby 首先检查该对象是否定义了单例类


```ruby
def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

Section 45.3: Singleton classes

All objects are instances of a class. However, that is not the whole truth. In Ruby, every object also has a somewhat hidden *singleton class*.

This is what allows methods to be defined on individual objects. The singleton class sits between the object itself and its actual class, so all methods defined on it are available for that object, and that object only.

```ruby
object = Object.new

def object.exclusive_method
  'Only this object will respond to this method'
end

object.exclusive_method
# => "Only this object will respond to this method"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

The example above could have been written using define_singleton_method:

```ruby
object.define_singleton_method :exclusive_method do
  "The method is actually defined in the object's singleton class"
end
```

Which is the same as defining the method on `object`'s singleton_class:

```ruby
# send is used because define_method is private
object.singleton_class.send :define_method, :exclusive_method do
  "Now we're defining an instance method directly on the singleton class"
end
```

Before the existence of `singleton_class` as part of Ruby's core API, singleton classes were known as *metaclasses* and could be accessed via the following idiom:

```ruby
class << object
  self  # refers to object's singleton_class
end
```

Section 45.4: Message Propagation with Singleton Class

Instances never contain a method they only carry data. However we can define a singleton class for any object including an instance of a class.

When a message is passed to an object (method is called) Ruby first checks if a singleton class is defined for that

对象以及它是否能回复该消息，否则 Ruby 会检查实例的类的祖先链并沿着链向上查找。

```
class Example
  def foo
    :example
  end
end

示例。新。foo #=> :example

模块 PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

第45.5节：重新打开（猴子补丁）单例类

重新打开单例类有三种方法

- 在单例类上使用class_eval。
- 使用class << 块。
- 使用def直接在对象的单例类上定义方法

```
class Example
end

示例。singleton_class.class_eval do
  def foo
    :foo
  end
end

示例。foo #=> :foo
```

```
class Example
end

class << 示例
  def bar
:bar
  end
end
```


object and if it can reply to that message otherwise Ruby checks instance's class' ancestors chain and walks up on that.

```
class Example
  def foo
    :example
  end
end

Example.new.foo #=> :example

module PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

Section 45.5: Reopening (monkey patching) Singleton Classes

There are three ways to reopen a Singleton Class

- Using `class_eval` on a singleton class.
- Using `class <<` block.
- Using `def` to define a method on the object's singleton class directly

```
class Example
end

Example.singleton_class.class_eval do
  def foo
    :foo
  end
end

Example.foo #=> :foo
```

```
class Example
end

class << Example
  def bar
    :bar
  end
end
```

```
示例.bar #=> :bar
```

```
class Example
end

def 示例.baz
  :baz
end

示例.baz #=> :baz
```

每个对象都有一个单例类，你可以访问它

```
class Example
end
ex1 = 示例.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = 示例.new
ex2.foobar #=> NoMethodError
```

第45.6节：访问单例类

获取对象的单例类有两种方法

- singleton_class 方法。
- 重新打开对象的单例类并返回self。

```
object.singleton_class
```

```
singleton_class = class << object
  self
end
```

第45.7节：访问单例类中的实例变量/类变量

单例类与其对象共享实例变量/类变量。

```
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

```
class Example
  def initialize
    @foo = 1
  end
```



```
Example.bar #=> :bar
```

```
class Example
end

def Example.baz
  :baz
end

Example.baz #=> :baz
```

Every object has a singleton class which you can access

```
class Example
end
ex1 = Example.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = Example.new
ex2.foobar #=> NoMethodError
```

Section 45.6: Accessing Singleton Class

There are two ways to get singleton class of an object

- `singleton_class` method.
- Reopening singleton class of an object and returning **`self`**.

```
object.singleton_class
```

```
singleton_class = class << object
  self
end
```

Section 45.7: Accessing Instance/Class Variables in Singleton Classes

Singleton classes share their instance/class variables with their object.

```
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

```
class Example
  def initialize
    @foo = 1
  end
```

```
  def foo
    @foo
  end
end

e = Example.new

e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

块闭包围绕它们的实例/类变量目标。使用块访问实例或类变量在 class_eval 或 instance_eval 中是不可能的。将字符串传递给 class_eval 或使用 class_variable_get 可以绕过该问题。

```
class Foo
  @@foo = :foo
end

类 示例
@@foo = :example

Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```



```
  def foo
    @foo
  end
end

e = Example.new

e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

Blocks close around their instance/class variables target. Accessing instance or class variables using a block in `class_eval` or `instance_eval` isn't possible. Passing a string to `class_eval` or using `class_variable_get` works around the problem.

```
class Foo
  @@foo = :foo
end

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

第46章：队列

第46.1节：多个工作者一个汇聚点

我们想收集多个工作者创建的数据。

首先我们创建一个队列：

```
sink = Queue.new
```

然后16个工作者都生成一个随机数并将其推入sink：

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

要获取数据，将队列转换为数组：

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

第46.2节：将队列转换为数组

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

或者一行代码：

```
[].tap { |array| array < queue.pop until queue.empty? }
```

第46.3节：一个源多个工作线程

我们想要并行处理数据。

让我们用一些数据填充源：

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

然后创建一些工作线程来处理数据：

```
(1..16).to_a.map do
  Thread.new do
    until source.empty?
item = source.pop
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
```


Chapter 46: Queue

Section 46.1: Multiple Workers One Sink

We want to gather data created by multiple Workers.

First we create a Queue:

```
sink = Queue.new
```

Then 16 workers all generating a random number and pushing it into sink:

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

And to get the data, convert a Queue to an Array:

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

Section 46.2: Converting a Queue into an Array

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

Or a one liner:

```
[].tap { |array| array < queue.pop until queue.empty? }
```

Section 46.3: One Source Multiple Workers

We want to process data in parallel.

Let's populate source with some data:

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

Then create some workers to process data:

```
(1..16).to_a.map do
  Thread.new do
    until source.empty?
      item = source.pop
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
```

```
end.map(&:join)
```

第46.4节：一个源 - 工作流水线 - 一个汇

我们希望并行处理数据，并将其传递给其他工作线程继续处理。

由于工作线程既消费又产生数据，我们必须创建两个队列：

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

第一波工作线程从`first_input_source`读取一个项目，处理该项目，并将结果写入`first_output_sink`：

```
(1..16).to_a.map do
  Thread.new do
    loop do
item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

第二波工作线程使用`first_output_sink`作为其输入源，读取、处理后写入另一个输出接收端：

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

现在second_output_sink是接收端，让我们将其转换为数组：

```
sleep 5 # 作为同步的变通方法
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

第46.5节：将数据推入队列 - #push

```
q = Queue.new
q << "任何对象，包括另一个队列"
# 或者
q.push :data
```

- 没有高水位标记，队列可以无限增长。
- #push 永不阻塞


```
end.map(&:join)
```

Section 46.4: One Source - Pipeline of Work - One Sink

We want to process data in parallel and push it down the line to be processed by other workers.

Since Workers both consume and produce data we have to create two queues:

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

First wave of workers read an item from `first_input_source`, process the item, and write results in `first_output_sink`:

```
(1..16).to_a.map do
  Thread.new do
    loop do
      item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

Second wave of workers uses `first_output_sink` as its input source and reads, process then writes to another output sink:

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
      item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

Now `second_output_sink` is the sink, let's convert it to an array:

```
sleep 5 # workaround in place of synchronization
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

Section 46.5: Pushing Data into a Queue - #push

```
q = Queue.new
q << "any object including another queue"
# or
q.push :data
```

- There is no high water mark, queues can infinitely grow.
- *#push* never blocks

第46.6节：从队列中拉取数据 - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- #pop 会阻塞直到有数据可用。
- #pop 可用于同步。

第46.7节：同步——在某个时间点之后

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
syncer << :ok
end

[a, b].map(&:join)
```

第46.8节：合并两个队列

- 为了避免无限阻塞，读取队列不应在合并线程正在进行的线程上进行。
- 为了避免在一个队列有数据时，另一个队列同步或无限等待，读取队列不应在同一线程中进行。

让我们先定义并填充两个队列：

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

我们应该创建另一个队列，并从其他线程向其中推送数据：

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
merged << q.pop
    end
  end
end
```

如果你知道可以完全消费两个队列（消费速度高于生产速度，且不会耗尽内存），有一种更简单的方法：

```
merged = Queue.new
merged << q1.pop until q1.empty?
```


Section 46.6: Pulling Data from a Queue - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- *#pop* will block until there is some data available.
- *#pop* can be used for synchronization.

Section 46.7: Synchronization - After a Point in Time

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
  syncer << :ok
end

[a, b].map(&:join)
```

Section 46.8: Merging Two Queues

- To avoid infinitely blocking, reading from queues shouldn't happen on the thread merge is happening on.
- To avoid synchronization or infinitely waiting for one of queues while other has data, reading from queues shouldn't happen on same thread.

Let's start by defining and populating two queues:

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

We should create another queue and push data from other threads into it:

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
      merged << q.pop
    end
  end
end
```

If you know you can completely consume both queues (consumption speed is higher than production, you won't run out of RAM) there is a simpler approach:

```
merged = Queue.new
merged << q1.pop until q1.empty?
```

```
merged << q2.pop until q2.empty?
```



```
merged << q2.pop until q2.empty?
```

第47章：解构

第47.1节：概述

解构的大部分魔法都使用了展开符号（*）操作符。

示例	结果 / 注释
`a, b = [0,1]`	`a=0, b=1`
`a, *剩余 = [0,1,2,3]`	`a=0, 剩余=[1,2,3]`
`a, * = [0,1,2,3]`	*a=0* *等同于 .first*
`*, z = [0,1,2,3]`	*z=3* *等同于 .last*

第47.2节：解构块参数

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```


Chapter 47: Destructuring

Section 47.1: Overview

Most of the magic of destructuring uses the splat (*) operator.

Example	Result / *comment*
`a, b = [0,1]`	`a=0, b=1`
`a, *rest = [0,1,2,3]`	`a=0, rest=[1,2,3]`
`a, * = [0,1,2,3]`	`a=0` *Equivalent to* `.first`
`*, z = [0,1,2,3]`	`z=3` *Equivalent to* `.last`

Section 47.2: Destructuring Block Arguments

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```

第48章：结构体（Struct）

第48.1节：为数据创建新结构体

Struct 定义具有指定属性和访问器方法的新类。

```
Person = Struct.new :first_name, :last_name
```

然后你可以实例化对象并使用它们：

```
person = Person.new '约翰', '多伊'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

第48.2节：自定义结构体类

```
Person = Struct.new :name do
  def greet(someone)
    "你好 #{someone}！我是 #{name}！"
  end
end

Person.new('爱丽丝').greet '鲍勃'
# => "你好，鲍勃！我是爱丽丝！"
```

第48.3节：属性查找

属性可以通过字符串和符号作为键来访问。数字索引也适用。

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```


Chapter 48: Struct

Section 48.1: Creating new structures for data

Struct defines new classes with the specified attributes and accessor methods.

```
Person = Struct.new :first_name, :last_name
```

You can then instantiate objects and use them:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

Section 48.2: Customizing a structure class

```
Person = Struct.new :name do
  def greet(someone)
    "Hello #{someone}! I am #{name}!"
  end
end

Person.new('Alice').greet 'Bob'
# => "Hello Bob! I am Alice!"
```

Section 48.3: Attribute lookup

Attributes can be accessed strings and symbols as keys. Numerical indexes also work.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

第49章：元编程

元编程可以用两种方式描述：

“计算机程序将其他程序（或自身）作为数据进行编写或操作，或者在编译时完成部分工作，而这些工作本应在运行时完成”。

更简单地说：元编程是在运行时编写生成代码的代码，以简化你的工作。

第49.1节：使用实例求值实现“with”

许多语言都有一个with语句，允许程序员省略方法调用的接收者。

在Ruby中，可以使用instance_eval轻松模拟with：

```
def with(object, &block)
  object.instance_eval &block
end
```

with方法可以用来无缝地在对象上执行方法：

```
hash = Hash.new

with hash do
store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

第49.2节：send()方法

send() 用于向object发送消息。send() 是Object类的一个实例方法。send() 的第一个参数是你发送给对象的消息——也就是方法名。它可以是string或symbol，但更推荐使用symbols。然后需要传入方法的参数，将作为send() 的剩余参数。

```
class Hello
   def hello(*args)
     puts 'Hello ' + args.join(' ')
   end
end
h = 你好。new
h.send :hello, 'gentle', 'readers'   #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> 这里 :hello 是方法，其余是该方法的参数。
```

这是一个更详细的示例

```
class Account
attr_accessor :name, :email, :notes, :address

   def assign_values(values)
     values.each_key do |k, v|
        # send 方法的实现示例
        # self.name = values[k]
        self.send("#{k}=", values[k])
      end
   end
```


Chapter 49: Metaprogramming

Metaprogramming can be described in two ways:

“Computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime”.

More simply put: **Metaprogramming is writing code that writes code during runtime to make your life easier**.

Section 49.1: Implementing "with" using instance evaluation

Many languages feature a `with` statement that allows programmers to omit the receiver of method calls.

`with` can be easily emulated in Ruby using `instance_eval`:

```
def with(object, &block)
  object.instance_eval &block
end
```

The `with` method can be used to seamlessly execute methods on objects:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Section 49.2: send() method

`send()` is used to pass message to `object`. `send()` is an instance method of the `Object` class. The first argument in `send()` is the message that you're sending to the object - that is, the name of a method. It could be `string` or `symbol` but **symbols** are preferred. Then arguments those need to pass in method, those will be the remaining arguments in `send()`.

```
class Hello
  def hello(*args)
    puts 'Hello ' + args.join(' ')
  end
end
h = Hello.new
h.send :hello, 'gentle', 'readers'   #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> Here :hello is method and rest are the arguments to method.
```

Here is the more descriptive example

```
class Account
  attr_accessor :name, :email, :notes, :address

  def assign_values(values)
    values.each_key do |k, v|
      # How send method would look a like
      # self.name = value[k]
      self.send("#{k}=", values[k])
    end
  end
```

```ruby
end

user_info = {
name: '马特',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "烦人的客户"
}

account = Account.new
如果属性增加那么我们会弄乱代码
#--------- 错误示范 --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- 元编程方式 --------------
account.assign_values(user_info) # 通过一行代码可以赋值任意数量的属性

puts account.inspect
```

注意：send() 本身不再推荐使用。请使用 __send__()，它可以调用私有方法，或者（推荐）`public_send()`

第49.3节：动态定义方法

使用Ruby，你可以在程序执行时修改程序结构。实现方式之一是使用方法 method_missing 动态定义方法。

假设我们想用以下语法测试一个数字是否大于另一个数字
777.is_greater_than_123?.

```ruby
# 打开 Numeric 类
类 Numeric
  # 重写 `method_missing` 方法
  定义 method_missing(method_name,*args)
    # 测试 method_name 是否匹配我们想要的语法
    如果 method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # 捕获 method_name 中的数字
the_other_number = $1.to_i
      # 返回该数字是否大于另一个数字
      self > the_other_number
    否则
      # 如果 method_name 不匹配我们想要的，交由之前定义的
`method_missing` 处理
      super
    end
  end
end
```

使用 method_missing 时需要记住的一件重要事情是，也应该重写 respond_to? 方法：

```ruby
类 Numeric
    定义 respond_to?(method_name, include_all = false)
      method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
    结束
end
```



```ruby
end

user_info = {
  name: 'Matt',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "annoying customer"
}

account = Account.new
If attributes gets increase then we would messup the code
#--------- Bad way --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- Meta Programing way --------------
account.assign_values(user_info) # With single line we can assign n number of attributes

puts account.inspect
```

Note: `send()` itself is not recommended anymore. Use `__send__()` which has the power to call private methods, or (recommended) `public_send()`

Section 49.3: Defining methods dynamically

With Ruby you can modify the structure of the program in execution time. One way to do it, is by defining methods dynamically using the method `method_missing`.

Let's say that we want to be able to test if a number is greater than other number with the syntax
`777.is_greater_than_123?`.

```ruby
# open Numeric class
class Numeric
  # override `method_missing`
  def method_missing(method_name,*args)
    # test if the method_name matches the syntax we want
    if method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # capture the number in the method_name
      the_other_number = $1.to_i
      # return whether the number is greater than the other number or not
      self > the_other_number
    else
      # if the method_name doesn't match what we want, let the previous definition of
`method_missing` handle it
      super
    end
  end
end
```

One important thing to remember when using `method_missing` that one should also override `respond_to?` method:

```ruby
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

忘记这样做会导致不一致的情况，比如你可以成功调用 600.is_greater_than_123，但 600.respond_to(:is_greater_than_123) 返回 false。

第49.4节：在实例上定义方法

在 Ruby 中，你可以向任何类的现有实例添加方法。这允许你为某个类的实例添加行为，而不改变该类其他实例的行为。

```
class 示例
  def method1(foo)
     puts foo
  end
end

#在对象 exp 上定义 method2 方法
exp = 示例。new
exp.define_method(:method2) {puts "Method2"}

#带方法参数
exp.define_method(:method3) {|name| puts name}
```


Forgetting to do so leads to a inconsistent situation, when you can successfully call `600.is_greater_than_123`, but `600.respond_to(:is_greater_than_123)` returns false.

Section 49.4: Defining methods on instances

In ruby you can add methods to existing instances of any class. This allows you to add behavior to and instance of a class without changing the behavior of the rest of the instances of that class.

```
class Example
  def method1(foo)
    puts foo
  end
end

#defines method2 on object exp
exp = Example.new
exp.define_method(:method2) {puts "Method2"}

#with method parameters
exp.define_method(:method3) {|name| puts name}
```

第50章：动态求值

参数	详细信息
`"source"`	任何 Ruby 源代码
绑定	Binding 类的一个实例
过程	Proc 类的一个实例

第50.1节：实例评估

instance_eval 方法在所有对象上都可用。它在接收者的上下文中执行代码：

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

instance_eval 在代码块执行期间将 self 设置为 object：

```
object.instance_eval { self == object }  # => true
```

接收者也作为其唯一参数传递给代码块：

```
object.instance_eval { |argument| argument == object }  # => true
```

instance_exec 方法在这方面有所不同：它将参数传递给代码块。

```
object.instance_exec :@variable do |name|
  instance_variable_get name  # => :value
end
```

第50.2节：评估字符串

任何 String 都可以在运行时被评估。

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

第50.3节：在绑定中评估

Ruby通过一个称为binding的对象跟踪局部变量和self变量。我们可以通过调用Kernel#binding获取作用域的binding，并通过Binding#eval在binding中评估字符串。

```
b = proc do
local_variable = :local
  binding
end.call
```


Chapter 50: Dynamic Evaluation

Parameter	Details
`"source"`	Any Ruby source code
binding	An instance of **Binding** class
proc	An instance of **Proc** class

Section 50.1: Instance evaluation

The instance_eval method is available on all objects. It evaluates code in the context of the receiver:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

instance_eval sets **self** to object for the duration of the code block:

```
object.instance_eval { self == object }  # => true
```

The receiver is also passed to the block as its only argument:

```
object.instance_eval { |argument| argument == object }  # => true
```

The instance_exec method differs in this regard: it passes its arguments to the block instead.

```
object.instance_exec :@variable do |name|
  instance_variable_get name  # => :value
end
```

Section 50.2: Evaluating a String

Any **String** can be evaluated at runtime.

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

Section 50.3: Evaluating Inside a Binding

Ruby keeps track of local variables and **self** variable via an object called binding. We can get binding of a scope with calling **Kernel**#binding and evaluate string inside a binding via **Binding**#eval.

```
b = proc do
  local_variable = :local
  binding
end.call
```

```
b.eval "local_variable" #=> :local
```

```
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
    :bar
  end
end

示例。foo #=> :foo
示例。new.bar #=> :bar
```

第50.4节：从字符串动态创建方法

Ruby 提供了define_method作为模块和类上的私有方法，用于定义新的实例方法。
但是，方法的"主体"必须是一个Proc或另一个已存在的方法。

从原始字符串数据创建方法的一种方式是使用eval从代码创建一个Proc：

```
xml = <<ENDXML
<methods>
<method name="go">puts "我正在前进！"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
```



```
b.eval "local_variable" #=> :local
```

```
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
    class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
    class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
    :bar
  end
end

Example.foo #=> :foo
Example.new.bar #=> :bar
```

Section 50.4: Dynamically Creating Methods from Strings

Ruby offers define_method as a private method on modules and classes for defining new instance methods. However, the 'body' of the method must be a **Proc** or another existing method.

One way to create a method from raw string data is to use **eval** to create a Proc from the code:

```
xml = <<ENDXML
<methods>
  <method name="go">puts "I'm going!"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
```

```
p f.public_methods(false)      #=> [:go, :stop]
f.go                           #=> "I'm going!"
p f.stop                       #=> 42
```



```
p f.public_methods(false)      #=> [:go, :stop]
f.go                           #=> "I'm going!"
p f.stop                       #=> 42
```

第51章：instance_eval

参数	详细信息
字符串	包含要执行的Ruby源代码。
文件名	用于错误报告的文件名。
行号	用于错误报告的行号。
代码块	要执行的代码块。
对象	接收者作为唯一参数传递给代码块。

第51.1节：实例评估

instance_eval 方法在所有对象上都可用。它在接收者的上下文中执行代码：

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

instance_eval 在代码块执行期间将 self 设置为 object：

```
object.instance_eval { self == object } # => true
```

接收者也作为其唯一参数传递给代码块：

```
object.instance_eval { |argument| argument == object } # => true
```

instance_exec 方法在这方面有所不同：它将参数传递给代码块。

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

第51.2节：使用以下方式实现

许多语言都有一个with语句，允许程序员省略方法调用的接收者。

在Ruby中，可以使用instance_eval轻松模拟with：

```
def with(object, &block)
  object.instance_eval &block
end
```

with方法可以用来无缝地在对象上执行方法：

```
hash = Hash.new

with hash do
存储 :key，:value
  has_key? :key        # => true
  values               # => [:value]
```


Chapter 51: instance_eval

Parameter	Details
string	Contains the Ruby source code to be evaluated.
filename	File name to use for error reporting.
lineno	Line number to use for error reporting.
block	The block of code to be evaluated.
obj	The receiver is passed to the block as its only argument.

Section 51.1: Instance evaluation

The instance_eval method is available on all objects. It evaluates code in the context of the receiver:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

instance_eval sets self to object for the duration of the code block:

```
object.instance_eval { self == object } # => true
```

The receiver is also passed to the block as its only argument:

```
object.instance_eval { |argument| argument == object } # => true
```

The instance_exec method differs in this regard: it passes its arguments to the block instead.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

Section 51.2: Implementing with

Many languages feature a with statement that allows programmers to omit the receiver of method calls.

with can be easily emulated in Ruby using instance_eval:

```
def with(object, &block)
  object.instance_eval &block
end
```

The with method can be used to seamlessly execute methods on objects:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key        # => true
  values               # => [:value]
```

```
end
```



```
end
```

第52章：消息传递

第52.1节：介绍

在*面向对象设计*中，对象*接收*消息并*回复*它们。在Ruby中，发送消息就是*调用方法*
该方法的结果即为回复。

在Ruby中，消息传递是动态的。当消息到达时，Ruby并不确切知道如何回复它，
而是使用一套预定义的规则来查找可以回复该消息的方法。我们可以利用这些规则来中断并回复消息，
将消息发送给另一个对象或修改消息等操作。

每当对象接收到消息时，Ruby会检查：

1.该对象是否有单例类且能回复该消息。
2.查找该对象的类及该类的祖先链。
3. 一个接一个地检查该祖先是否有该方法，并沿着链向上移动。

第52.2节：通过继承链传递消息

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end

  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

s = Subexample.new
```

为了找到适合SubExample#subexample_method的方法，Ruby 首先查看了SubExample的祖先链

```
SubExample。ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```


Chapter 52: Message Passing

Section 52.1: Introduction

In *Object Oriented Design*, objects *receive* messages and *reply* to them. In Ruby, sending a message is *calling a method* and result of that method is the reply.

In Ruby message passing is dynamic. When a message arrives rather than knowing exactly how to reply to it Ruby uses a predefined set of rules to find a method that can reply to it. We can use these rules to interrupt and reply to the message, send it to another object or modify it among other actions.

Each time an object receives a message Ruby checks:

1. If this object has a singleton class and it can reply to this message.
2. Looks up this object's class then class' ancestors chain.
3. One by one checks if a method is available on this ancestor and moves up the chain.

Section 52.2: Message Passing Through Inheritance Chain

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end

  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

s = Subexample.new
```

To find a suitable method for `SubExample#subexample_method` Ruby first looks at ancestors chain of `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

它从SubExample开始。如果我们发送subexample_method消息，Ruby会选择SubExample中可用的方法并忽略Example*#subexample_method*。

```
s.subexample_method # => :subexample
```

在SubExample之后，它会检查Example。如果我们发送example_method，Ruby会检查SubExample是否能响应，由于不能，Ruby会沿着继承链向上查找Example。

```
s.example_method # => :example
```

当Ruby检查完所有定义的方法后，它会运行method_missing来查看是否能响应。如果我们发送missing_subexample_method，Ruby无法在SubExample找到定义的方法，便向上查找Example。它在Example或继承链上更高的类中也找不到定义的方法。Ruby重新开始并运行method_missing。SubExample的method_missing可以响应missing_subexample_method。

```
s.missing_subexample_method # => :subexample
```

然而，如果方法已定义，Ruby会使用已定义的方法，即使它在继承链上层。例如，如果我们发送not_missed_method，尽管SubExample的method_missing可以响应它，Ruby会在SubExample上向上查找，因为它没有该名称的已定义方法，接着查找Example，后者有该方法。

```
s.not_missed_method # => :example
```

第52.3节：通过模块组合进行消息传递

Ruby沿着对象的祖先链向上移动。该链可以包含模块和类。关于沿链向上移动的规则同样适用于模块。

```
class Example
end

模块 Prepended
  定义 initialize *args
    如果 args.empty?，则返回 super :default
    super
  end
end

模块 FirstIncluded定义 f
  oo
    :first
  end
end

模块 SecondIncluded定义
  foo
    :second
  end
end

类 SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
```


It starts from SubExample. If we send subexample_method message Ruby chooses the one available one SubExample and ignores Example*#subexample_method*.

```
s.subexample_method # => :subexample
```

After SubExample it checks Example. If we send example_method Ruby checks if SubExample can reply to it or not and since it can't Ruby goes up the chain and looks into Example.

```
s.example_method # => :example
```

After Ruby checks all defined methods then it runs method_missing to see if it can reply or not. If we send missing_subexample_method Ruby won't be able to find a defined method on SubExample so it moves up to Example. It can't find a defined method on Example or any other class higher in chain either. Ruby starts over and runs method_missing. method_missing of SubExample can reply to missing_subexample_method.

```
s.missing_subexample_method # => :subexample
```

However if a method is defined Ruby uses defined version even if it is higher in the chain. For example if we send not_missed_method even though method_missing of SubExample can reply to it Ruby walks up on SubExample because it doesn't have a defined method with that name and looks into Example which has one.

```
s.not_missed_method # => :example
```

Section 52.3: Message Passing Through Module Composition

Ruby moves up on ancestors chain of an object. This chain can contain both modules and classes. Same rules about moving up the chain apply to modules as well.

```
class Example
end

module Prepended
  def initialize *args
    return super :default if args.empty?
    super
  end
end

module FirstIncluded
  def foo
    :first
  end
end

module SecondIncluded
  def foo
    :second
  end
end

class SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
```

```
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example, Object,
Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

第52.4节：中断消息

有两种方法可以中断消息。

- 使用method_missing来中断任何未定义的消息。
- 在链的中间定义一个方法以拦截消息

中断消息后，可以：

- 回复它们。
- 将它们发送到其他地方。
- 修改消息或其结果。

通过method_missing中断并回复消息：

```
class 示例
  def foo
    @foo
  end

  def method_missing name, data
    return super 除非 name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
  end
end

e = Example.new

e.foo = :foo
e.foo # => :foo
```

拦截消息并修改它：

```
类 示例
  def initialize title, body
  end
end

class SubExample < Example
end
```

现在假设我们的数据是“title:body”，并且在调用Example之前必须先拆分它们。我们可以在SubExample上定义initialize。

```
class SubExample < 示例
  def initialize raw_data
    processed_data = raw_data.split ":"
```



```
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example, Object,
Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

Section 52.4: Interrupting Messages

There are two ways to interrupt messages.

- Use `method_missing` to interrupt any non defined message.
- Define a method in middle of a chain to intercept the message

After interrupting messages, it is possible to:

- Reply to them.
- Send them somewhere else.
- Modify the message or its result.

Interrupting via `method_missing` and replying to message:

```
class Example
  def foo
    @foo
  end

  def method_missing name, data
    return super unless name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
  end
end

e = Example.new

e.foo = :foo
e.foo # => :foo
```

Intercepting message and modifying it:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Now let's imagine our data is "title:body" and we have to split them before calling `Example`. We can define `initialize` on `SubExample`.

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"
```

```
    super processed_data[0], processed_data[1]
  end
end
```

拦截消息并将其发送到另一个对象：

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
  end
end
```



```
    super processed_data[0], processed_data[1]
  end
end
```

Intercepting message and sending it to another object:

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
  end
end
```

第53章：关键字参数

第53.1节：使用带有splat操作符的任意关键字参数

您可以使用双星号（**）操作符定义一个方法，以接受任意数量的关键字参数：

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

参数会被捕获到一个哈希中。您可以操作该哈希，例如提取所需的参数。

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

使用带有关键字参数的展开运算符将阻止关键字参数的验证，方法在遇到未知关键字时
永远不会引发`ArgumentError`错误。

对于标准的展开运算符，你可以将一个Hash重新转换为方法的关键字参数：

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

这通常用于你需要操作传入参数，并将它们传递给底层方法的情况：

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
  puts something
params = {}
params[:foo] = foo || "默认 foo"
  params[:bar] = bar || "默认 bar"
  inner(**params)
end
```


Chapter 53: Keyword Arguments

Section 53.1: Using arbitrary keyword arguments with splat operator

You can define a method to accept an arbitrary number of keyword arguments using the *double splat* (**) operator:

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

The arguments are captured in a `Hash`. You can manipulate the `Hash`, for example to extract the desired arguments.

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

Using a the splat operator with keyword arguments will prevent keyword argument validation, the method will never raise an `ArgumentError` in case of unknown keyword.

As for the standard splat operator, you can re-convert a `Hash` into keyword arguments for a method:

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

This is generally used when you need to manipulate incoming arguments, and pass them to an underlying method:

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
  puts something
  params = {}
  params[:foo] = foo || "Default foo"
  params[:bar] = bar || "Default bar"
  inner(**params)
end
```

```
outer "你好:", foo: "自定义 foo"
# 你好:
# 自定义 foo
# 默认 bar
```

第 53.2 节：使用关键字参数

你可以通过在方法定义中指定名称来定义关键字参数：

```
def say(message: "你好，世界")
  puts message
end

say
# => "你好，世界"

say message: "今天是星期一"
# => "今天是星期一"
```

你可以定义多个关键字参数，定义顺序无关紧要：

```
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "今天是星期一"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

关键字参数可以与位置参数混合使用：

```
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

在 Ruby 2.1 之前，将关键字参数与位置参数混合使用是一种非常常见的做法，因为当时无法定义必需的关键字参数。

此外，在 Ruby 2.0 之前，通常会在方法定义的末尾添加一个Hash来用于可选参数。其语法与关键字参数非常相似，以至于通过Hash实现的可选参数与 Ruby 2 的关键字参数兼容。

```
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
end

# 该方法调用在语法上等同于关键字参数的调用
say "Hello World", before: "<span>", after: "</span>"
```



```
outer "Hello:", foo: "Custom foo"
# Hello:
# Custom foo
# Default bar
```

Section 53.2: Using keyword arguments

You define a keyword argument in a method by specifying the name in the method definition:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

You can define multiple keyword arguments, the definition order is irrelevant:

```
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "Today is Monday"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

Keyword arguments can be mixed with positional arguments:

```
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Mixing keyword argument with positional argument was a very common approach before Ruby 2.1, because it was not possible to define required keyword arguments.

Moreover, in Ruby < 2.0, it was very common to add an **Hash** at the end of a method definition to use for optional arguments. The syntax is very similar to keyword arguments, to the point where optional arguments via **Hash** are compatible with Ruby 2 keyword arguments.

```
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
end

# The method call is syntactically equivalent to the keyword argument one
say "Hello World", before: "<span>", after: "</span>"
```

```
# => "<span>Hello World</span>"
```

注意，尝试传递未定义的关键字参数将导致错误：

```
def say(message: "你好，世界")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

第53.3节：必需的关键字参数

版本 ≥ 2.1

必需的关键字参数是在 Ruby 2.1 中引入的，作为对关键字参数的改进。

要将关键字参数定义为必需，只需声明该参数而不指定默认值。

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "你好，世界"
```

你也可以混合使用必需和非必需的关键字参数：

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```



```
# => "<span>Hello World</span>"
```

Note that trying to pass a not-defined keyword argument will result in an error:

```
def say(message: "Hello World")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

Section 53.3: Required keyword arguments

Version ≥ 2.1

Required keyword arguments were introduced in Ruby 2.1, as an improvement to keyword arguments.

To define a keyword argument as required, simply declare the argument without a default value.

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "Hello World"
```

You can also mix required and non-required keyword arguments:

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

第54章：真值性

第54.1节：所有对象都可以在

Ruby中转换为布尔值

使用双重否定语法来检查值的真值性。所有值都对应一个布尔值，无论它们的类型如何。

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: 警告: 字符串字面量在条件中
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

除 nil 和 false 之外，所有值都是真值。

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

第54.2节：值的真值性可以用于 if-else 结构

在 if-else 语句中不需要使用双重否定。

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

上述代码在屏幕上打印“hey!”。

Chapter 54: Truthiness

Section 54.1: All objects may be converted to booleans in Ruby

Use the double negation syntax to check for truthiness of values. All values correspond to a boolean, irrespective of their type.

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: warning: string literal in condition
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

All values except `nil` and `false` are truthy.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

Section 54.2: Truthiness of a value can be used in if-else constructs

You do not need to use double negation in if-else statements.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

The above code prints 'hey!' on the screen.

第55章：隐式接收者与理解self

第55.1节：总是存在隐式接收者

在Ruby中，所有方法调用总是有一个隐式接收者。语言会将当前隐式接收者的引用存储在变量self中。某些语言关键字如class和module会改变self指向的对象。理解这些行为对于掌握这门语言非常有帮助。

例如，当你第一次打开irb

```
irb(main):001:0> self
=> main
```

在这种情况下，main对象是隐式接收者（关于main的更多信息见http://stackoverflow.com/a/917842/417872）。

你可以使用def关键字在隐式接收者上定义方法。例如：

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

这已经在你运行的 repl 中的主对象实例上定义了方法 foo。

请注意，局部变量的查找优先于方法名，因此如果你定义了一个同名的局部变量，其引用将覆盖方法引用。接着前面的例子：

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

method 方法仍然可以找到 foo 方法，因为它不检查局部变量，而普通引用 foo 会检查。

第 55.2 节：关键字改变隐式接收者

当你定义一个类或模块时，隐式接收者变成对该类本身的引用。例如：

```
puts "我是 #{self}"
class 示例
  puts "我是 #{self}"
end
```


Chapter 55: Implicit Receivers and Understanding Self

Section 55.1: There is always an implicit receiver

In Ruby, there is always an implicit receiver for all method calls. The language keeps a reference to the current implicit receiver stored in the variable `self`. Certain language keywords like `class` and `module` will change what `self` points to. Understanding these behaviors is very helpful in mastering the language.

For example, when you first open `irb`

```
irb(main):001:0> self
=> main
```

In this case the `main` object is the implicit receiver (see http://stackoverflow.com/a/917842/417872 for more about `main`).

You can define methods on the implicit receiver using the `def` keyword. For example:

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

This has defined the method foo on the instance of main object running in your repl.

Note that local variables are looked up before method names, so that if you define a local variable with the same name, its reference will supersede the method reference. Continuing from the previous example:

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

The `method` method can still find the `foo` method because it doesn't check for local variables, while the normal reference `foo` does.

Section 55.2: Keywords change the implicit receiver

When you define a class or module, the implicit receiver becomes a reference to the class itself. For example:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

执行上述代码将打印：

```
"我是 main"
"我是 Example"
```

第55.3节：何时使用 self？

大多数 Ruby 代码使用隐式接收者，因此刚接触 Ruby 的程序员常常不清楚何时使用self。实际的答案是，self主要有两种用法：

1. 改变接收者。

通常情况下，类或模块内部的def行为是创建实例方法。Self 可以用来定义类方法。

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. 用于消除接收者的歧义

当局部变量可能与方法同名时，可能需要显式接收者来消除歧义。

示例：

```
class 示例
   def foo
    1
  end

  def bar
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo 是方法，foo 是局部变量
  end

  def qux
bar = 2
    self.bar + bar # self.bar 是方法，bar 是局部变量
  end
end

示例.new.foo    #=> 1
示例.new.bar    #=> 2
示例.new.baz(2) #=> 3
示例.new.qux    #=> 4
```


Executing the above code will print:

```
"I am main"
"I am Example"
```

Section 55.3: When to use self?

Most Ruby code utilizes the implicit receiver, so programmers who are new to Ruby are often confused about when to use `self`. The practical answer is that `self` is used in two major ways:

1. To change the receiver.

Ordinarily the behavior of `def` inside a class or module is to create instance methods. Self can be used to define methods on the class instead.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. To disambiguate the receiver

When local variables may have the same name as a method an explicit receiver may be required to disambiguate.

Examples:

```
class Example
  def foo
    1
  end

  def bar
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo is the method, foo is the local variable
  end

  def qux
    bar = 2
    self.bar + bar # self.bar is the method, bar is the local variable
  end
end

Example.new.foo    #=> 1
Example.new.bar    #=> 2
Example.new.baz(2) #=> 3
Example.new.qux    #=> 4
```

另一种常见的需要消歧义的情况涉及以等号结尾的方法。例如：

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
foo = input # 将创建一个局部变量
  end

  def baz(input)
    self.foo = input # 将调用该方法
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```


The other common case requiring disambiguation involves methods that end in the equals sign. For instance:

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
    foo = input # will create a local variable
  end

  def baz(input)
    self.foo = input # will call the method
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```

第56章：自省

第56.1节：查看对象的方法

检查对象

你可以使用methods或public_methods方法来查找对象可以响应的公共方法，这两个方法都会返回一个符号数组：

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

为了获得更有针对性的方法列表，你可以去除所有对象共有的方法，例如：

```
p (f.methods - Object.methods).sort
#=> [:bar, :yay]
```

或者，你可以向methods或public_methods传入false参数：

```
p f.methods(false) # `f`的公共和受保护的单例方法
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

你可以使用private_methods和protected_methods来查找对象的私有和受保护方法：

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```


Chapter 56: Introspection

Section 56.1: View an object's methods

Inspecting an Object

You can find the public methods an object can respond to using either the `methods` or `public_methods` methods, which return an array of symbols:

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

For a more targeted list, you can remove methods common to all objects, e.g.

```
p (f.methods - Object.methods).sort
#=> [:bar, :yay]
```

Alternatively, you can pass **`false`** to `methods` or `public_methods`:

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

You can find the private and protected methods of an object using `private_methods` and `protected_methods`:

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```

与methods和public_methods一样，你可以向private_methods和protected_methods传递false来去除继承的方法。

检查类或模块

除了methods、public_methods、protected_methods和private_methods之外，类和模块还提供了instance_methods、public_instance_methods、protected_instance_methods和private_instance_methods，用于确定继承自该类或模块的对象所暴露的方法。如上所述，你可以向这些方法传递false以排除继承的方法：

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

最后，如果你将来忘记了这些方法的大部分名称，可以使用methods找到所有这些方法：

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

第56.2节：查看对象的实例变量

可以使用instance_variables、instance_variable_defined?和instance_variable_get查询对象的实例变量，并使用instance_variable_set和remove_instance_variable修改它们：

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                 #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)       #=> 42
f.instance_variable_set(:@bar, 17)    #=> 17
f.bar                                 #=> 17
f.remove_instance_variable(:@bar)     #=> 17
f.bar                                 #=> nil
f.instance_variables                 #=> []
```


As with methods and public_methods, you can pass **false** to private_methods and protected_methods to trim away inherited methods.

Inspecting a Class or Module

In addition to methods, public_methods, protected_methods, and private_methods, classes and modules expose instance_methods, public_instance_methods, protected_instance_methods, and private_instance_methods to determine the methods exposed for objects that inherit from the class or module. As above, you can pass **false** to these methods to exclude inherited methods:

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

Finally, if you forget the names of most of these in the future, you can find all of these methods using methods:

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

Section 56.2: View an object's Instance Variables

It is possible to query an object about its instance variables using instance_variables, instance_variable_defined?, and instance_variable_get, and modify them using instance_variable_set and remove_instance_variable:

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                 #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)       #=> 42
f.instance_variable_set(:@bar, 17)    #=> 17
f.bar                                 #=> 17
f.remove_instance_variable(:@bar)     #=> 17
f.bar                                 #=> nil
f.instance_variables                 #=> []
```

实例变量的名称包含@符号。如果省略它，将会出现错误：

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' 不能作为实例变量名
```

第56.3节：查看全局变量和局部变量

Kernel 提供了获取 global_variables 和 local_variables 列表的方法：

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$|, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

与实例变量不同，没有专门用于获取、设置或删除全局变量或局部变量的方法。
寻找这种功能通常表明你的代码应该重写为使用哈希来存储值。

但是，如果你必须通过名称修改全局或局部变量，可以使用 eval 和字符串：

```
var = "$demo"
eval(var)            #=> "进行中"
eval("#{var} = 17")
p $demo              #=> 17
```

默认情况下，`eval` 会在当前作用域中计算你的变量。要在不同作用域中计算局部变量，
你必须捕获局部变量所在的*binding*。

```
def local_variable_get(name, bound=nil)
  foo = :inside
  eval(name,bound)
end

def test_1
foo = :outside
  p local_variable_get("foo")
end

def test_2
foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

在上面，`test_1` 没有传递 binding 给 `local_variable_get`，因此 `eval` 在该方法的上下文中执行，
该上下文中名为 `foo` 的局部变量被设置为 `:inside`。

第56.4节：查看类变量

类和模块拥有与其他对象相同的方法来检查实例变量。类和

The names of instance variables include the @ symbol. You will get an error if you omit it:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

Section 56.3: View Global and Local Variables

The `Kernel` exposes methods for getting the list of `global_variables` and `local_variables`:

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$\, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

Unlike instance variables there are no methods specifically for getting, setting, or removing global or local variables. Looking for such functionality is usually a sign that your code should be rewritten to use a Hash to store the values. However, if you must modify global or local variables by name, you can use `eval` with a string:

```
var = "$demo"
eval(var)            #=> "in progress"
eval("#{var} = 17")
p $demo              #=> 17
```

By default, `eval` will evaluate your variables in the current scope. To evaluate local variables in a different scope, you must capture the *binding* where the local variables exist.

```
def local_variable_get(name, bound=nil)
  foo = :inside
  eval(name,bound)
end

def test_1
  foo = :outside
  p local_variable_get("foo")
end

def test_2
  foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

In the above, `test_1` did not pass a binding to `local_variable_get`, and so the `eval` was executed within the context of that method, where a local variable named `foo` was set to `:inside`.

Section 56.4: View Class Variables

Classes and modules have the same methods for introspecting instance variables as any other object. Class and

模块也有类似的方法来查询类变量（@@these_things）：

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
@@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                   #=> [:@@instances]
p Bar.class_variables                   #=> [:@@instances]
p Foo.class_variable_get(:@@instances)  #=> 8
p Bar.class_variable_get(:@@instances)  #=> 8
```

与实例变量类似，类变量的名称必须以@@开头，否则会报错：

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```


modules also have similar methods for querying the class variables (`@@these_things`):

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
  @@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                   #=> [:@@instances]
p Bar.class_variables                   #=> [:@@instances]
p Foo.class_variable_get(:@@instances)  #=> 8
p Bar.class_variable_get(:@@instances)  #=> 8
```

Similar to instance variables, the name of class variables must begin with `@@`, or you will get an error:

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```

第57章：精炼（Refinements）

第57.1节：有限范围的猴子补丁（Monkey patching）

Monkey patching 的主要问题是它会污染全局作用域。你的代码能否正常工作取决于你所使用的所有模块是否不会相互干扰。Ruby 对此的解决方案是 refinements（精炼），它们基本上是在有限作用域内的 monkey patching。

```
模块 补丁
  refine Fixnum 执行
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

类 RefinementTest
  # 可以访问我们的补丁
  使用 Patches

  def 初始化
    puts 1.plus_one
    puts 3.concat_one
  end
end

# 主作用域没有更改

1.plus_one
# => 未定义方法 `plus_one' 用于 1:Fixnum (NoMethodError)

RefinementTest.新建
# => 2
# => '31'
```

第57.2节：双重用途模块（refinements 或全局补丁）

使用 Refinements 来限定补丁的作用域是一种好习惯，但有时全局加载它也很方便（例如在开发或测试中）。

例如，假设你想启动一个控制台，加载你的库，然后让修补过的方法在全局作用域中可用。你无法通过 refinements 实现这一点，因为using必须在类/模块定义中调用。但可以编写代码，使其具有双重用途：

```
模块 Patch
  定义 patched?; true; 结束
  refine String 执行
    包含 Patch
```


Chapter 57: Refinements

Section 57.1: Monkey patching with limited scope

Monkey patching's main issue is that it pollutes the global scope. Your code working is at the mercy of all the modules you use not stepping on each others toes. The Ruby solution to this is refinements, which are basically monkey patches in a limited scope.

```
module Patches
  refine Fixnum do
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

class RefinementTest
  # has access to our patches
  using Patches

  def initialize
    puts 1.plus_one
    puts 3.concat_one
  end
end

# Main scope doesn't have changes

1.plus_one
# => undefined method `plus_one' for 1:Fixnum (NoMethodError)

RefinementTest.new
# => 2
# => '31'
```

Section 57.2: Dual-purpose modules (refinements or global patches)

It's a good practice to scope patches using Refinements, but sometimes it's nice to load it globally (for example in development, or testing).

Say for example you want to start a console, require your library, and then have the patched methods available in the global scope. You couldn't do this with refinements because `using` needs to be called in a class/module definition. But it's possible to write the code in such a way that it's dual purpose:

```
module Patch
  def patched?; true; end
  refine String do
    include Patch
```

```
  end
end

# 全局范围内
String.包含 Patch
"".patched? # => true

# refinement
类 LoadPatch
  使用 Patch
  "".patched? # => true
结束
```

第57.3节：动态 refinements

Refinements 有特殊限制。

refine只能在模块作用域中使用，但可以通过 send :refine来编程实现。

using的使用更有限。它只能在类/模块定义中调用。不过，它可以接受指向模块的变量，并且可以在循环中调用。

展示这些概念的示例：

```
模块 Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

由于using非常静态，如果精炼文件没有先加载，可能会出现加载顺序问题。解决方法之一是将已修补的类/模块定义包装在proc中。例如：

```
模块 Patch
refine String do
    def patched; true; end
  end
end

class Foo
end

# 这是一个proc，因为方法中不能包含类定义
create_patched_class = Proc.new do
  Foo.class_exec do
    类 Bar
使用 Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
```



```
  end
end

# globally
String.include Patch
"".patched? # => true

# refinement
class LoadPatch
  using Patch
  "".patched? # => true
end
```

Section 57.3: Dynamic refinements

Refinements have special limitations.

`refine` can only be used in a module scope, but can be programmed using `send :refine`.

`using` is more limited. It can only be called in a class/module definition. Still, it can accept a variable pointing to a module, and can be invoked in a loop.

An example showing these concepts:

```
module Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
  patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

Since `using` is so static, there can be issued with load order if the refinement files are not loaded first. A way to address this is to wrap the patched class/module definition in a proc. For example:

```
module Patch
  refine String do
    def patched; true; end
  end
end

class Foo
end

# This is a proc since methods can't contain class definitions
create_patched_class = Proc.new do
  Foo.class_exec do
    class Bar
      using Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
```

```
Foo::Bar.patched? # => true
```

调用该 proc 会创建已修补的类 Foo::Bar。此操作可以延迟到所有代码加载完成之后。


```
Foo::Bar.patched? # => true
```

Calling the proc creates the patched class `Foo::Bar`. This can be delayed until after all the code has loaded.

第58章：使用
Begin / Rescue 捕获异常

第58.1节：基本的错误处理块

让我们写一个函数来除以两个数字，对输入非常信任：

```
def divide(x, y)
  return x/y
end
```

这对于许多输入来说都能很好地工作：

```
> puts divide(10, 2)
5
```

但并非所有情况

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String 不能被强制转换为 Fixnum
```

我们可以通过将有风险的除法操作包裹在 `begin... end` 块中来重写函数以检查错误，并使用 `rescue` 子句在出现问题时输出信息并返回 `nil`。

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "发生了错误"
    return nil
  end
end

> puts divide(10, 0)
发生了错误

> puts divide(10, 'a')
发生了错误
```

第58.2节：保存错误

如果你想在rescue子句中使用它，可以保存该错误

```
def divide(x, y)
  begin
x/y
  rescue => e
    puts "发生了一个 %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
```


Chapter 58: Catching Exceptions with Begin / Rescue

Section 58.1: A Basic Error Handling Block

Let's make a function to divide two numbers, that's very trusting about its input:

```
def divide(x, y)
  return x/y
end
```

This will work fine for a lot of inputs:

```
> puts divide(10, 2)
5
```

But not all

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String can't be coerced into Fixnum
```

We can rewrite the function by wrapping the risky division operation in a `begin... end` block to check for errors, and use a `rescue` clause to output a message and return `nil` if there is a problem.

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "There was an error"
    return nil
  end
end

> puts divide(10, 0)
There was an error

> puts divide(10, 'a')
There was an error
```

Section 58.2: Saving the Error

You can save the error if you want to use it in the `rescue` clause

```
def divide(x, y)
  begin
    x/y
  rescue => e
    puts "There was a %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
```

```
发生了一个 ZeroDivisionError （除以 0）
      来自 (irb):10:在 `/'
来自 (irb):10
来自 /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
发生了一个 TypeError（字符串无法强制转换为 Fixnum）
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:在 `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:在 `block (2 levels) in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels) in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

第58.3节：检查不同的错误

如果你想根据错误类型做不同的处理，可以使用多个`rescue`子句，每个子句带有不同的错误类型作为参数。

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "不要除以零！"
    return nil
  rescue TypeError
    puts "除法只适用于数字！"
    return nil
  end
end

> divide(10, 0)
不要除以零！

> divide(10, 'a')
除法只适用于数字！
```

如果你想保存错误以便在rescue块中使用：

```
rescue ZeroDivisionError => e
```

使用不带参数的rescue子句来捕获未在其他rescue子句中指定类型的错误。

```
def divide(x, y)
  begin
    return x/y
```



```
There was a ZeroDivisionError (divided by 0)
      from (irb):10:in `/'
      from (irb):10
      from /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
There was a TypeError (String can't be coerced into Fixnum)
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:in `block (2 levels) in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels) in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

Section 58.3: Checking for Different Errors

If you want to do different things based on the kind of error, use multiple `rescue` clauses, each with a different error type as an argument.

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  end
end

> divide(10, 0)
Don't divide by zero!

> divide(10, 'a')
Division only works on numbers!
```

If you want to save the error for use in the `rescue` block:

```
rescue ZeroDivisionError => e
```

Use a `rescue` clause with no argument to catch errors of a type not specified in another `rescue` clause.

```
def divide(x, y)
  begin
    return x/y
```

```
  rescue ZeroDivisionError
    puts "不要除以零！"
    return nil
  rescue TypeError
    puts "除法只适用于数字！"
    return nil
  rescue => e
    puts "不要那样做 (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
不要那样做 (NoMethodError)
```

在这种情况下，尝试将 `nil` 除以 2 既不是 `ZeroDivisionError`也不是 `TypeError`，因此由默认的 `rescue` 子句处理，该子句会打印一条消息，告诉我们这是一个 `NoMethodError`。

第58.4节：重试

在rescue子句中，可以使用retry来重新运行begin子句，通常是在改变了导致错误的情况之后。

```
def divide(x, y)
  begin
    puts "准备进行除法..."
    return x/y
  rescue ZeroDivisionError
    puts "不能除以零！"
    y = 1
    retry
  rescue TypeError
    puts "除法只适用于数字！"
    return nil
  rescue => e
    puts "不要那样做 (%s)" % [e.class]
    return nil
  end
end
```

如果传入已知会导致TypeError的参数，begin子句会被执行（这里通过打印"准备进行除法"来标记），错误会像之前一样被捕获，并返回nil：

```
> divide(10, 'a')
准备进行除法...
除法只适用于数字！
 => nil
```

但如果传入会导致ZeroDivisionError的参数，begin子句会被执行，错误被捕获，除数从0改为1，然后retry使begin块重新执行（从头开始），这次使用了不同的y。第二次执行时没有错误，函数返回了一个值。

```
> divide(10, 0)
准备除法...     # 第一次，10 ÷ 0
不要除以零！
准备除法...     # 第二次 10 ÷ 1
=> 10
```



```
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
Don't do that (NoMethodError)
```

In this case, trying to divide `nil` by 2 is not a `ZeroDivisionError` or a `TypeError`, so it handled by the default `rescue` clause, which prints out a message to let us know that it was a `NoMethodError`.

Section 58.4: Retrying

In a `rescue` clause, you can use `retry` to run the `begin` clause again, presumably after changing the circumstance that caused the error.

```
def divide(x, y)
  begin
    puts "About to divide..."
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    y = 1
    retry
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end
```

If we pass parameters that we know will cause a `TypeError`, the `begin` clause is executed (flagged here by printing out "About to divide") and the error is caught as before, and `nil` is returned:

```
> divide(10, 'a')
About to divide...
Division only works on numbers!
 => nil
```

But if we pass parameters that will cause a `ZeroDivisionError`, the `begin` clause is executed, the error is caught, the divisor changed from 0 to 1, and then `retry` causes the `begin` block to be run again (from the top), now with a different `y`. The second time around there is no error and the function returns a value.

```
> divide(10, 0)
About to divide...     # First time, 10 ÷ 0
Don't divide by zero!
About to divide...     # Second time 10 ÷ 1
=> 10
```

第58.5节：检查是否未引发错误

你可以使用else子句来编写在没有错误发生时运行的代码。

```
def divide(x, y)
  begin
z = x/y
  rescue ZeroDivisionError
    puts "不要除以零！"
  rescue TypeError
    puts "除法只适用于数字！"
    return nil
  rescue => e
    puts "不要那样做 (%s)" % [e.class]
    return nil
  else
    puts "如果没有错误，这段代码将运行。"
    return z
  end
end
```

如果发生错误并转移控制到某个rescue子句，else子句将不会运行：

```
> divide(10,0)
不要除以零！
=> nil
```

但如果没有错误发生，else 子句将执行：

```
> divide(10,2)
如果没有错误，这段代码将会运行。
=> 5
```

注意，如果你从begin代码块返回，else代码块将不会被执行

```
def divide(x, y)
  begin
z = x/y
    return z                # 将阻止else代码块的运行！
  rescue ZeroDivisionError
    puts "不要除以零！"
  else
    puts "如果没有错误，这段代码将会运行。"
    return z
  end
end

> divide(10,2)
=> 5
```

第58.6节：应始终运行的代码

如果有代码是你总想执行的，可以使用ensure子句。

```
def divide(x, y)
  begin
z = x/y
    return z
```


Section 58.5: Checking Whether No Error Was Raised

You can use an `else` clause for code that will be run if no error is raised.

```
def divide(x, y)
  begin
    z = x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  else
    puts "This code will run if there is no error."
    return z
  end
end
```

The `else` clause does not run if there is an error that transfers control to one of the `rescue` clauses:

```
> divide(10,0)
Don't divide by zero!
=> nil
```

But if no error is raised, the `else` clause executes:

```
> divide(10,2)
This code will run if there is no error.
=> 5
```

Note that the `else` clause will not be executed *if you return from the `begin` clause*

```
def divide(x, y)
  begin
    z = x/y
    return z                # Will keep the else clause from running!
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  else
    puts "This code will run if there is no error."
    return z
  end
end

> divide(10,2)
=> 5
```

Section 58.6: Code That Should Always Run

Use an `ensure` clause if there is code you always want to execute.

```
def divide(x, y)
  begin
    z = x/y
    return z
```

```
  rescue ZeroDivisionError
    puts "不要除以零！"
  rescue TypeError
    puts "除法只适用于数字！"
    return nil
  rescue => e
    puts "不要那样做 (%s)" % [e.class]
    return nil
  ensure
    puts "这段代码总是会执行。"
  end
end
```

当发生错误时，ensure子句会被执行：

```
> divide(10, 0)
不要除以零！    # rescue 子句
这段代码总是会执行。    # ensure 子句
=> nil
```

当没有错误时：

```
> divide(10, 2)
这段代码总是会执行。    # ensure 子句
=> 5
```

当你想确保例如文件被关闭时，ensure 子句非常有用。

请注意，与else子句不同，ensure子句会在begin或rescue子句返回值之前执行。如果ensure子句中有return语句，它将覆盖其他任何子句的return值！


```
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  ensure
    puts "This code ALWAYS runs."
  end
end
```

The `ensure` clause will be executed when there is an error:

```
> divide(10, 0)
Don't divide by zero!   # rescue clause
This code ALWAYS runs.   # ensure clause
=> nil
```

And when there is no error:

```
> divide(10, 2)
This code ALWAYS runs.   # ensure clause
=> 5
```

The ensure clause is useful when you want to make sure, for instance, that files are closed.

Note that, unlike the `else` clause, the `ensure` clause *is executed* before the `begin` or `rescue` clause returns a value. If the `ensure` clause has a `return` that will override the `return` value of any other clause!

第59章：命令行应用程序

第59.1节：如何编写一个通过邮政编码获取天气的命令行工具

这将是一个相对全面的教程，介绍如何编写一个命令行工具，根据提供给命令行工具的邮政编码打印天气。第一步是用Ruby编写程序来完成此操作。让我们从编写一个方法weather(zip_code)开始（此方法需要 yahoo_weatherman gem。如果你没有安装此gem，可以通过命令行输入gem install yahoo_weatherman来安装）

```
require 'yahoo_weatherman'

def weather(zip_code)
client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

我们现在有了一个非常基础的方法，当提供邮政编码时可以返回天气。现在我们需要将其做成一个命令行工具。快速回顾一下命令行工具如何从shell调用及相关变量。当工具这样调用时tool argument other_argument，在Ruby中有一个变量ARGV，它是一个数组，等于['argument', 'other_argument']。现在让我们在应用程序中实现它

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

很好！现在我们有了一个可以运行的命令行应用程序。注意文件开头的she-bang行（#!/usr/bin/ruby）。这使文件成为可执行文件。我们可以将此文件保存为weather。（注意：不要保存为weather.rb，文件扩展名不是必须的，she-bang会告诉系统这是一个Ruby文件）。现在我们可以在shell中运行以下命令（不要输入$符号）。

```
$ chmod a+x weather
$ ./weather [ZIPCODE]
```

测试确认可用后，我们现在可以通过运行以下命令将其软链接到/usr/bin/local/目录

```
$ sudo ln -s weather /usr/local/bin/weather
```

现在无论你处于哪个目录，都可以在命令行调用 weather 。

Chapter 59: Command Line Apps

Section 59.1: How to write a command line tool to get the weather by zip code

This will be a relatively comprehensive tutorial of how to write a command line tool to print the weather from the zip code provided to the command line tool. The first step is to write the program in ruby to do this action. Let's start by writing a method `weather(zip_code)` (This method requires the `yahoo_weatherman` gem. If you do not have this gem you can install it by typing `gem install yahoo_weatherman` from the command line)

```
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

We now have a very basic method that gives the weather when a zip code is provided to it. Now we need to make this into a command line tool. Very quickly let's go over how a command line tool is called from the shell and the associated variables. When a tool is called like this `tool argument other_argument`, in ruby there is a variable `ARGV` which is an array equal to `['argument', 'other_argument']`. Now let us implement this in our application

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

Good! Now we have a command line application that can be run. Notice the *she-bang* line at the beginning of the file (`#!/usr/bin/ruby`). This allows the file to become an executable. We can save this file as `weather`. (**Note**: Do not save this as `weather.rb`, there is no need for the file extension and the she-bang tells whatever you need to tell that this is a ruby file). Now we can run these commands in the shell (do not type in the $).

```
$ chmod a+x weather
$ ./weather [ZIPCODE]
```

After testing that this works, we can now sym-link this to the `/usr/bin/local/` by running this command

```
$ sudo ln -s weather /usr/local/bin/weather
```

Now `weather` can be called on the command line no matter the directory you are in.

第60章：IRB

选项	详细信息
-f	禁止读取 ~/.irbrc 文件
-m	Bc 模式（可加载 mathn、fraction 或 matrix）
-d	将 $DEBUG 设置为 true（等同于 `ruby -d`）
-r 加载模块	与 `ruby -r` 相同
-I 路径	指定 $LOAD_PATH 目录
-U	与 ruby -U 相同
-E 编码	与 ruby -E 相同
-w	与 ruby -w 相同
-W[level=2]	与 ruby -W 相同
--inspect	使用 `inspect` 进行输出（默认，除 bc 模式外）
--noinspect	不使用 inspect 进行输出
--readline	使用 Readline 扩展模块
--noreadline	不使用 Readline 扩展模块
--prompt prompt-mode	切换提示符模式。预定义的提示符模式有 default'、simple'、xmp' 和inf-ruby'
--inf-ruby-mode	在 emacs 的 inf-ruby-mode 中使用适当的提示符。抑制 --readline。
--simple-prompt	简单提示符模式
--noprompt	无提示符模式
--tracer	显示每次命令执行的跟踪信息。
--back-trace-limit n	显示前 n 条和后 n 条回溯。默认值为 16。
--irb_debug n	设置内部调试级别为 n（非普遍使用）
-v，--version	打印 irb 的版本

IRB 意味着“交互式 Ruby 终端”。基本上它允许你实时执行 Ruby 命令（就像普通的 shell 一样）。IRB 是处理 Ruby API 时不可或缺的工具。它作为经典的 rb 脚本工作。用于简短且简单的命令。IRB 的一个很好的功能是，当你在输入方法时按下 Tab 键，它会给出你可以使用的建议（这不是 IntelliSense）

第 60.1 节：在 Ruby 脚本中启动 IRB 会话

从 Ruby 2.4.0 开始，你可以使用以下代码在任何 Ruby 脚本中启动交互式 IRB 会话：

```
require 'irb'
binding.irb
```

这将启动一个 IRB REPL，你将获得预期的 self 值，并且能够访问所有作用域内的局部变量和实例变量。输入 Ctrl+D 或 quit 以继续执行你的 Ruby 程序。

这对于调试非常有用。

第 60.2 节：基本用法

IRB 是“交互式 Ruby 解释器”的意思，允许我们从标准输入执行 Ruby 表达式。

首先，在你的终端中输入irb。你可以用 Ruby 编写任何内容，从简单的表达式开始：

```
$ irb
2.1.4 :001 > 2+2
```


Chapter 60: IRB

Option	Details
-f	Suppress read of ~/.irbrc
-m	Bc mode (load mathn, fraction or matrix are available)
-d	Set $DEBUG to true (same as `ruby -d')
-r load-module	Same as `ruby -r'
-I path	Specify $LOAD_PATH directory
-U	Same as `ruby -U`
-E enc	Same as `ruby -E`
-w	Same as `ruby -w`
-W[level=2]	Same as `ruby -W`
--inspect	Use `inspect' for output (default except for bc mode)
--noinspect	Don't use inspect for output
--readline	Use Readline extension module
--noreadline	Don't use Readline extension module
--prompt prompt-mode	Switch prompt mode. Pre-defined prompt modes are `default',simple', `xmp' andinf-ruby'
--inf-ruby-mode	Use prompt appropriate for inf-ruby-mode on emacs. Suppresses --readline.
--simple-prompt	Simple prompt mode
--noprompt	No prompt mode
--tracer	Display trace for each execution of commands.
--back-trace-limit n	Display backtrace top n and tail n. The default value is 16.
--irb_debug n	Set internal debug level to n (not for popular use)
-v, --version	Print the version of irb

IRB means "Interactive Ruby Shell". Basically it lets you execute ruby commands in real time (like the normal shell does). IRB is an indispensable tool when dealing with Ruby API. Works as classical rb script. Use it for short and easy commands. One of the nice IRB functions is that when you press tab while typing a method it will give you an advice to what you can use (This is not an IntelliSense)

Section 60.1: Starting an IRB session inside a Ruby script

As of Ruby 2.4.0, you can start an interactive IRB session inside any Ruby script using these lines:

```
require 'irb'
binding.irb
```

This will start an IBR REPL where you will have the expected value for `self` and you will be able to access all local variables and instance variables that are in scope. Type Ctrl+D or `quit` in order to resume your Ruby program.

This can be very useful for debugging.

Section 60.2: Basic Usage

IRB means "Interactive Ruby Shell", letting us execute ruby expressions from the standart input.

To start, type `irb` into your shell. You can write anything in Ruby, from simple expressions:

```
$ irb
2.1.4 :001 > 2+2
```

```
=> 4
```

到像方法这样复杂的情况：

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```



```
=> 4
```

to complex cases like methods:

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```

第61章：ERB

ERB代表嵌入式Ruby，用于在模板（例如HTML和YAML）中插入Ruby变量。ERB是一个Ruby类，接受文本，并评估和替换被ERB标记包围的Ruby代码。

第61.1节：解析ERB

此示例是来自IRB会话的过滤文本。

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# 这是一个注释 %>
    <li><%= i %> 是 <%= i.even? ? '偶数' : '奇数' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> 打印 输出
<ul>


    <li>0 是偶数。</li>


    <li>1 是奇数。</li>

    <li>2 是偶数。</li>

    <li>3 是奇数。</li>

    <li>4 是偶数。</li>

    <li>5 是奇数。</li>

    <li>6 是偶数。</li>

    <li>7 是奇数。</li>

    <li>8 是偶数。</li>

    <li>9 是奇数。</li>

    <li>10 是偶数。</li>
</ul>
```


Chapter 61: ERB

ERB stands for Embedded Ruby, and is used to insert Ruby variables inside templates, e.g. HTML and YAML. ERB is a Ruby class that accepts text, and evaluates and replaces Ruby code surrounded by ERB markup.

Section 61.1: Parsing ERB

This example is filtered text from an `IRB` session.

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# This is a comment %>
    <li><%= i %> is <%= i.even? ? 'even' : 'odd' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> print output
<ul>


    <li>0 is even.</li>


    <li>1 is odd.</li>

    <li>2 is even.</li>

    <li>3 is odd.</li>

    <li>4 is even.</li>

    <li>5 is odd.</li>

    <li>6 is even.</li>

    <li>7 is odd.</li>

    <li>8 is even.</li>

    <li>9 is odd.</li>

    <li>10 is even.</li>
</ul>
```

第62章：生成随机数

如何在Ruby中生成随机数。

第62.1节：6面骰子

```
   # 掷一个6面骰子，rand(6)返回0到5（含）之间的数字
dice_roll_result = 1 + rand(6)
```

第62.2节：从区间生成随机数（含端点）

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # 根据需要更改范围运算符
```


Chapter 62: Generate a random number

How to generate a random number in Ruby.

Section 62.1: 6 Sided die

```
   # Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
   dice_roll_result = 1 + rand(6)
```

Section 62.2: Generate a random number from a range (inclusive)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

第63章：Hanami入门

我的使命是为社区做出贡献，帮助想要学习这个惊人框架——Hanami的新手。

但这将如何运作呢？

简短轻松的教程，通过示例展示Hanami，接下来的教程中我们将学习如何测试我们的应用程序并构建一个简单的REST API。

让我们开始吧！

第63.1节：关于Hanami

除了Hanami是一个轻量且快速的框架外，最引人注目的点之一是Clean Architecture（清晰架构）概念，它向我们展示了这个框架不是我们的应用程序，正如罗伯特·马丁之前所说。

Hanami 架构设计为我们提供了使用Container的方式，在每个 Container 中，我们的应用程序独立于框架。这意味着我们可以拿起我们的代码并将其放入例如 Rails 框架中。

Hanami 是一个 MVC 框架吗？

MVC 框架的理念是构建一个遵循 Model -> Controller -> View 的结构。Hanami 遵循 Model | Controller -> View -> Template。其结果是一个更加解耦的应用程序，遵循SOLID原则，并且更加简洁。

- 重要链接。

Hanami http://hanamirb.org/

Robert Martin - Clean Arquitecture https://www.youtube.com/watch?v=WpkDN78P884

Clean Arquitecture https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

SOLID Principles http://practicingruby.com/articles/solid-design-principles

第63.2节：如何安装 Hanami？

- 步骤 1: 安装 Hanami gem。

```
$ gem install hanami
```

- 步骤 2: 生成一个新项目，设置RSpec作为测试框架。

  打开命令行或终端。要生成一个新的 hanami 应用程序，使用 hanami new，后跟应用程序名称和 rspec 测试参数。

```
$ hanami new "myapp" --test=rspec
```

注意：默认情况下，Hanami 将 Minitest 设为测试框架。

Chapter 63: Getting started with Hanami

My mission here is to contribute with the community to help new people who wants to learn about this amazing framework - Hanami.

But how it is going to work?

Short and easygoing tutorials showing with examples about Hanami and following the next tutorials we will see how to test our application and build a simple REST API.

Let's start!

Section 63.1: About Hanami

Besides Hanami be a lightweight and fast framework one of the points that most call attention is the **Clean Architecture** concept where shows to us that the framework is not our application as Robert Martin said before.

Hanami arquitecture design offer to us the use of **Container**, in each Container we have our application independently of the framework. This means that we can grab our code and put it into a Rails framework for example.

Hanami is a MVC Framework?

The MVC's frameworks idea is to build one structure following the Model -> Controller -> View. Hanami follows the Model | Controller -> View -> Template. The result is an application more uncopled, following **SOLID** principles, and much cleaner.

- Important links.

Hanami http://hanamirb.org/

Robert Martin - Clean Arquitecture https://www.youtube.com/watch?v=WpkDN78P884

Clean Arquitecture https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

SOLID Principles http://practicingruby.com/articles/solid-design-principles

Section 63.2: How to install Hanami?

- **Step 1:** Installing the Hanami gem.

```
$ gem install hanami
```

- **Step 2**: Generate a new project setting RSpec as testing framework.

  Open up a command line or terminal. To generate a new hanami application, use hanami new followed by the name of your app and the rspec test param.

```
$ hanami new "myapp" --test=rspec
```

Obs. By default Hanami sets Minitest as testing framework.

这将创建一个名为 myapp 的 hanami 应用程序，位于 myapp 目录中，并使用 bundle install 安装 Gemfile 中已提及的 gem 依赖。

要切换到该目录，请使用 cd 命令，cd 代表更改目录。

```
$ cd my_app
$ bundle install
```

myapp 目录包含多个自动生成的文件和文件夹，构成了 Hanami 应用程序的结构。以下是默认创建的文件和文件夹列表：

- Gemfile 定义了我们的 Rubygems 依赖（使用 Bundler）。
- Rakefile 描述了我们的 Rake 任务。
- apps 包含一个或多个兼容 Rack 的 Web 应用程序。在这里我们可以找到第一个生成的 Hanami 应用程序，名为 Web。这里是我们找到控制器、视图、路由和模板的地方。
- config 包含配置文件。
- config.ru 是用于Rack服务器的文件。
- db 包含我们的数据库模式和迁移文件。
- lib 包含我们的业务逻辑和领域模型，包括实体和仓库。
- public 将包含编译后的静态资源。
- spec 包含我们的测试。

- **重要链接。**

Hanami gem https://github.com/hanami/hanami

Hanami官方入门指南 http://hanamirb.org/guides/getting-started/

第63.3节：如何启动服务器？

- 步骤1： 要启动服务器，只需输入以下命令，然后你将看到启动页面。

```
$ bundle exec hanami server
```


This will create a hanami application called myapp in a myapp directory and install the gem dependencies that are already mentioned in Gemfile using bundle install.

To switch to this directory, use the cd command, which stands for change directory.

```
$ cd my_app
$ bundle install
```

The myapp directory has a number of auto-generated files and folders that make up the structure of a Hanami application. Following is a list of files and folders that are created by default:

- **Gemfile** defines our Rubygems dependencies (using Bundler).
- **Rakefile** describes our Rake tasks.
- **apps** contains one or more web applications compatible with Rack. Here we can find the first generated Hanami application called Web. It's the place where we find our controllers, views, routes and templates.
- **config** contains configuration files.
- **config.ru** is for Rack servers.
- **db** contains our database schema and migrations.
- **lib** contains our business logic and domain model, including entities and repositories.
- **public** will contain compiled static assets.
- **spec** contains our tests.

- **Important links.**

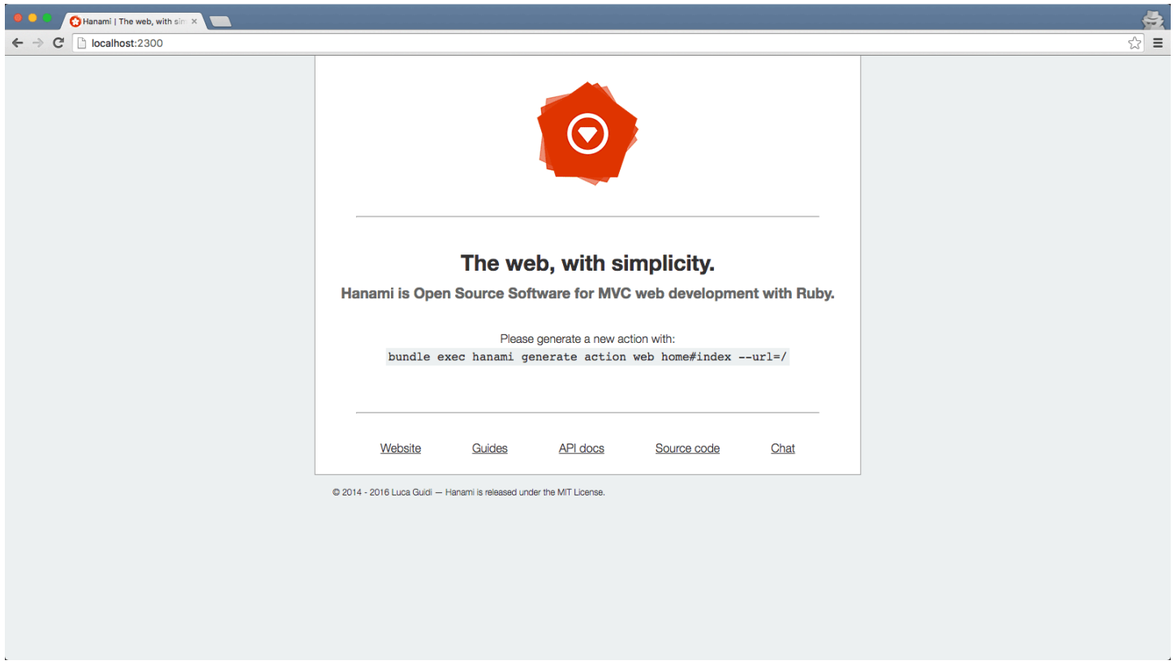
Hanami gem https://github.com/hanami/hanami

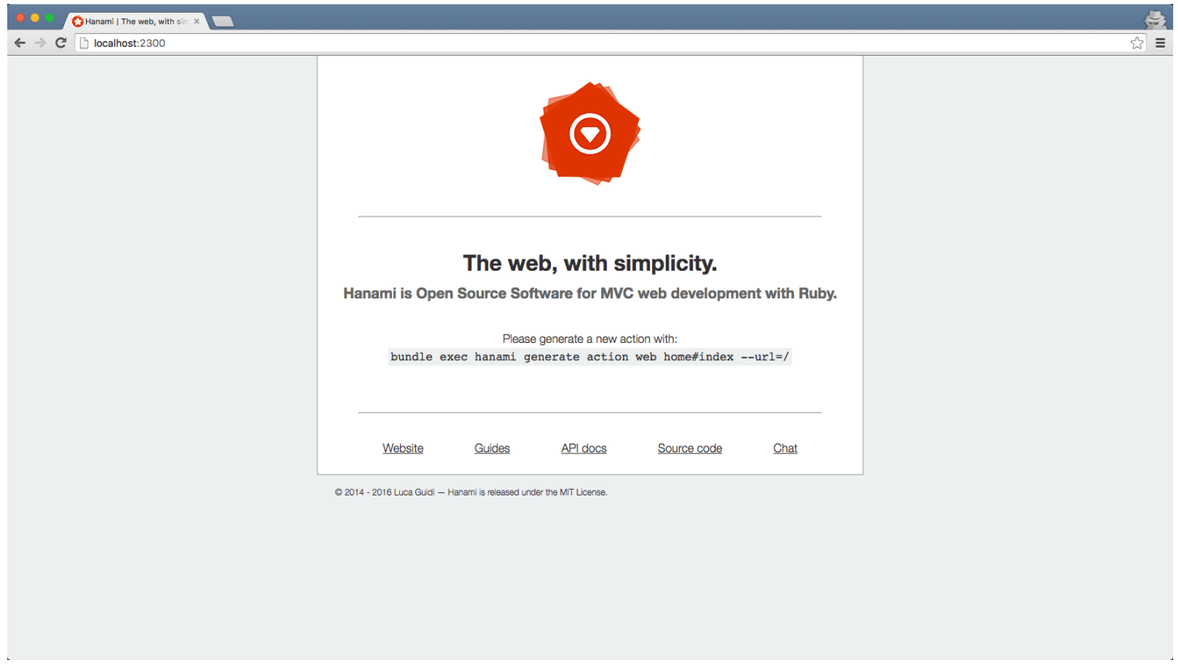
Hanami official Getting Started http://hanamirb.org/guides/getting-started/

Section 63.3: How to start the server?

- **Step 1:** To start the server just type the command bellow then you'll see the start page.

```
$ bundle exec hanami server
```


Hanami | The web, with sim
localhost:2300
The web, with simplicity.
Hanami is Open Source Software for MVC web development with Ruby.
Please generate a new action with:
bundle exec hanami generate action web home#index --url=/
Website
Guides
API docs
Source code
Chat
© 2014 - 2016 Luca Guidi — Hanami is released under the MIT License.


Hanami | The web, with sim
localhost:2300
The web, with simplicity.
Hanami is Open Source Software for MVC web development with Ruby.
Please generate a new action with:
bundle exec hanami generate action web home#index --url=/
Website
Guides
API docs
Source code
Chat
© 2014 - 2016 Luca Guidi — Hanami is released under the MIT License.

第64章：OptionParser

OptionParser 可用于解析来自ARGV的命令行选项。

第64.1节：必选和可选命令行选项

如果你不需要处理太复杂的情况，手动解析命令行相对来说比较简单：

```
# 简单的错误检查
除非ARGV.length >= 2，否则中止('用法: ' + $0 + ' site id ...')

# 第一个参数（site）是必需的
site = ARGV.shift

ARGV.each do | id |
  # 对每个id执行一些有趣的操作
end
```

但当你的选项变得更复杂时，你可能需要使用一个选项解析器，比如说，OptionParser：

```
require 'optparse'

# 实际的选项将存储在此哈希中
options = {}

# 设置你要查找的选项
optparse = OptionParser.new do |opts|
  opts.banner = "用法: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "站点名称") do |s|
    options[:site] = s
  end

opts.on( '-h', '--help', "显示此帮助信息" ) do
    puts opts
    exit
  end
end

# parse! 方法也会从 ARGV 中移除它找到的任何选项。
optparse.parse!
```

还有一个非破坏性的 parse 方法，但如果你计划使用 ARGV 中剩余的内容，它的用处要小得多。

OptionParser 类没有强制必需参数（例如本例中的 --site）的功能。不过你可以在运行 parse! 后自行进行检查：

```
# 稍微复杂一些的错误检查
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

有关更通用的强制选项处理，请参见this answer。如果不清楚，所有选项都是可选的，除非你特别设置为强制。

Chapter 64: OptionParser

OptionParser can be used for parsing command line options from ARGV.

Section 64.1: Mandatory and optional command line options

It's relatively easy to parse the command line by hand if you aren't looking for anything too complex:

```
# Naive error checking
abort('Usage: ' + $0 + ' site id ...') unless ARGV.length >= 2

# First item (site) is mandatory
site = ARGV.shift

ARGV.each do | id |
  # Do something interesting with each of the ids
end
```

But when your options start to get more complicated, you probably will need to use an option parser such as, well, OptionParser:

```
require 'optparse'

# The actual options will be stored in this hash
options = {}

# Set up the options you are looking for
optparse = OptionParser.new do |opts|
  opts.banner = "Usage: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "Site name") do |s|
    options[:site] = s
  end

  opts.on( '-h', '--help', 'Display this screen' ) do
    puts opts
    exit
  end
end

# The parse! method also removes any options it finds from ARGV.
optparse.parse!
```

There's also a non-destructive parse, but it's a lot less useful if you plan on using the remainder of what's in ARGV.

The OptionParser class doesn't have a way to enforce mandatory arguments (such as --site in this case). However you can do you own checking after running parse!:

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

For a more generic mandatory option handler, see this answer. In case it isn't clear, all options are optional unless you go out of your way to make them mandatory.

第64.2节：默认值

使用OptionsParser，设置默认值非常简单。只需预先填充存储选项的哈希表：

```
options = {
  :directory => ENV['HOME']
}
```

当你定义解析器时，如果用户提供了值，它将覆盖默认值：

```
OptionParser.new do |opts|
opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

第64.3节：详细描述

有时你的描述可能会相当长。例如irb -h 列出了一个参数，内容是：

```
  --context-mode n   设置 n[0-3] 为创建 Binding 对象的方法，
                     当 新工作区被创建时
```

如何支持这一点并不立即清楚。大多数解决方案需要调整，使第二行及后续行的缩进与第一行对齐。幸运的是，on方法通过将它们作为单独的参数添加，支持多行描述：

```
opts.on("--context-mode n",
          "设置 n[0-3] 为创建 Binding 对象的方法，",
          "当新工作区被创建时") do |n|
optons[:context_mode] = n
  end
```

你可以添加任意多的描述行来充分解释该选项。

Section 64.2: Default values

With `OptionsParser`, it's really easy to set up default values. Just pre-populate the hash you store the options in:

```
options = {
  :directory => ENV['HOME']
}
```

When you define the parser, it will overwrite the default if a user provide a value:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

Section 64.3: Long descriptions

Sometimes your description can get rather long. For instance `irb -h` lists on argument that reads:

```
  --context-mode n   Set n[0-3] to method to create Binding Object,
                     when new workspace was created
```

It's not immediately clear how to support this. Most solutions require adjusting to make the indentation of the second and following lines align to the first. Fortunately, the `on` method supports multiple description lines by adding them as separate arguments:

```
  opts.on("--context-mode n",
          "Set n[0-3] to method to create Binding Object,",
          "when new workspace was created") do |n|
    optons[:context_mode] = n
  end
```

You can add as many description lines as you like to fully explain the option.

第65章：操作系统或Shell命令

有许多与操作系统交互的方式。在Ruby内部，你可以运行shell/系统命令或子进程。

第65.1节：在

Ruby中执行shell代码的推荐方式：

Open3.popen3 或 Open3.capture3：
Open3 实际上只是使用了 Ruby 的 spawn 命令，但为你提供了更好的 API。

Open3.popen3

Popen3 在子进程中运行，并返回 stdin、stdout、stderr 和 wait_thr。

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

或

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
   puts "stdout 是:" + stdout.read
   puts "stderr 是:" + stderr.read
end
```

将输出：**stdout 是: stderr 是:fatal: Not a git repository (or any of the parent directories): .git**

或者

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

将输出：

正在 ping www.google.com [216.58.223.36]，数据包大小为 32 字节：
来自 216.58.223.36 的回复：字节=32 时间=16ms TTL=54
来自 216.58.223.36 的回复：字节=32 时间=10ms TTL=54
来自 216.58.223.36 的回复：字节=32 时间=21ms TTL=54
来自 216.58.223.36 的回复：字节=32 时间=29ms TTL=54
216.58.223.36 的 Ping 统计信息：
数据包：已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程时间的近似值（毫秒）：
最短 = 10ms，最长 = 29ms，平均 = 19ms

Chapter 65: Operating System or Shell commands

There are many ways to interact with the operating system. From within Ruby you can run shell/system commands or sub-processes.

Section 65.1: Recommended ways to execute shell code in Ruby:

Open3.popen3 or Open3.capture3:
Open3 actually just uses Ruby's spawn command, but gives you a much better API.

Open3.popen3

Popen3 runs in a sub-process and returns stdin, stdout, stderr and wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

or

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  puts "stdout is:" + stdout.read
  puts "stderr is:" + stderr.read
end
```

will output: **stdout is: stderr is:fatal: Not a git repository (or any of the parent directories): .git**

or

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

will output:

Pinging www.google.com [216.58.223.36] with 32 bytes of data:
Reply from 216.58.223.36: bytes=32 time=16ms TTL=54
Reply from 216.58.223.36: bytes=32 time=10ms TTL=54
Reply from 216.58.223.36: bytes=32 time=21ms TTL=54
Reply from 216.58.223.36: bytes=32 time=29ms TTL=54
Ping statistics for 216.58.223.36:
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 10ms, Maximum = 29ms, Average = 19ms

Open3.capture3：

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # 命令成功完成，执行更多操作
else
  raise "发生错误"
end
```

或

```
Open3.capture3('/some/binary with some args')
```

不推荐这样做，因为会增加额外开销并且可能导致shell注入风险。

如果命令从标准输入读取，并且你想给它传递一些数据：

```
Open3.capture3('my_funky_command', stdin_data: '从标准输入读取')
```

通过使用chdir，以不同的工作目录运行命令：

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

第65.2节：在Ruby中执行shell代码的经典方法：

Exec：

```
exec 'echo "hello world"'
```

或

```
exec ('echo "hello world"')
```

系统命令：

```
system 'echo "hello world"'
```

将在命令窗口输出 "hello world"。

或

```
system ('echo "hello world"')
```

system 命令如果执行成功会返回 true，失败时返回 nil。

```
result = system 'echo "hello world"'
puts result  # 将在命令窗口返回 true
```

反引号 (`)：

echo "hello world" 将在命令窗口输出 "hello world"。

你也可以捕获结果。

Open3.capture3:

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # command completed successfully, do some more stuff
else
  raise "An error occurred"
end
```

or

```
Open3.capture3('/some/binary with some args')
```

Not recommended though, due to additional overhead and the potential for shell injections.

If the command reads from stdin and you want to feed it some data:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Run the command with a different working directory, by using chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

Section 65.2: Clasic ways to execute shell code in Ruby:

Exec:

```
exec 'echo "hello world"'
```

or

```
exec ('echo "hello world"')
```

The System Command:

```
system 'echo "hello world"'
```

Will output "hello world" in the command window.

or

```
system ('echo "hello world"')
```

The system command can return a true if the command was successful or nill when not.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

The backticks (`):

echo "hello world" Will output "hello world" in the command window.

You can also catch the result.

```
result = `echo "hello world"`
puts "我们总是编写代码 " + result
```

IO.popen:

```
# 将从系统获取并返回当前日期
IO.popen("date") { |f| puts f.gets }
```



```
result = `echo "hello world"`
puts "We always code a " + result
```

IO.popen:

```
# Will get and return the current date from the system
IO.popen("date") { |f| puts f.gets }
```

第66章：C扩展

第66.1节：你的第一个扩展

C扩展由两个主要部分组成：

1. C代码本身。
2. 扩展配置文件。

要开始你的第一个扩展，请将以下内容放入名为 extconf.rb 的文件中：

```
require 'mkmf'

create_makefile('hello_c')
```

有几点需要指出：

首先，名称 hello_c 是你编译后的扩展输出的名称。你将会在使用 require 时用到它。

其次，extconf.rb 文件实际上可以命名为任何名称，只是传统上用于构建包含本地代码的 gem，实际编译扩展的是运行 ruby extconf.rb 时生成的 Makefile。默认生成的 Makefile 会编译当前目录下所有的 .c 文件。

将以下内容放入名为 hello.c 的文件中，然后运行 ruby extconf.rb && make

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {prin
  tf("Hello World!");return Qnil;

}

// 该模块的初始化方法
void Init_hello_c() {
VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

代码解析：

名称 Init_hello_c 必须与 extconf.rb 文件中定义的名称匹配，否则在动态加载扩展时，Ruby 将无法找到引导扩展所需的符号。

调用 rb_define_module 正在创建一个名为 HelloC 的Ruby模块，我们将把C函数放在该命名空间下。

最后，调用b_define_singleton_method创建了一个直接绑定到HelloC模块的模块级方法
我们可以通过ruby使用HelloC.world来调用它。

在调用make编译扩展后，我们可以运行C扩展中的代码。

打开一个控制台！

```
irb(main):001:0> require './hello_c'
```


Chapter 66: C Extensions

Section 66.1: Your first extension

C extensions are comprised of two general pieces:

1. The C Code itself.
2. The extension configuration file.

To get started with your first extension put the following in a file named `extconf.rb`:

```
require 'mkmf'

create_makefile('hello_c')
```

A couple of things to point out:

First, the name `hello_c` is what the output of your compiled extension is going to be named. It will be what you use in conjunction with **`require`**.

Second, the `extconf.rb` file can actually be named anything, it's just traditionally what is used to build gems that have native code, the file that is actually going to compile the extension is the Makefile generated when running `ruby extconf.rb`. The default Makefile that is generated compiles all `.c` files in the current directory.

Put the following in a file named `hello.c` and run `ruby extconf.rb && make`

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {
  printf("Hello World!\n");
  return Qnil;
}

// The initialization method for this module
void Init_hello_c() {
  VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

A breakdown of the code:

The name `Init_hello_c` must match the name defined in your `extconf.rb` file, otherwise when dynamically loading the extension, Ruby won't be able to find the symbol to bootstrap your extension.

The call to `rb_define_module` is creating a Ruby module named `HelloC` which we're going to namespace our C functions under.

Finally, the call to `rb_define_singleton_method` makes a module level method tied directly to the `HelloC` module which we can invoke from ruby with `HelloC.world`.

After having compiled the extension with the call to `make` we can run the code in our C extension.

Fire up a console!

```
irb(main):001:0> require './hello_c'
```

```
=> 真
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

第66.2节：使用C结构体

为了能够将C结构体作为Ruby对象使用，你需要用`Data_Wrap_Struct`和`Data_Get_Struct`进行封装。

Data_Wrap_Struct将C数据结构封装到Ruby对象中。它接收指向你的数据结构的指针，以及一些回调函数指针，返回一个VALUE。宏Data_Get_Struct接收该VALUE并返回指向你的C数据结构的指针。

下面是一个简单的示例：

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
free(self->name);
  }
ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free, ruby_xmalloc(sizeof(example_struct)));
}

static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

Check_Type(name, T_STRING);

Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);

  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
Data_Get_Struct(self, example_struct, p);

  printf("%s", p->name);return

  Qnil;
}

void Init_example()
{
VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);
```



```
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

Section 66.2: Working with C Structs

In order to be able to work with C structs as Ruby objects, you need to wrap them with calls to `Data_Wrap_Struct` and `Data_Get_Struct`.

`Data_Wrap_Struct` wraps a C data structure in a Ruby object. It takes a pointer to your data structure, along with a few pointers to callback functions, and returns a VALUE. The `Data_Get_Struct` macro takes that VALUE and gives you back a pointer to your C data structure.

Here's a simple example:

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
    free(self->name);
  }
  ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free, ruby_xmalloc(sizeof(example_struct)));
}

static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

  Check_Type(name, T_STRING);

  Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);

  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
  Data_Get_Struct(self, example_struct, p);

  printf("%s\n", p->name);

  return Qnil;
}

void Init_example()
{
  VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);
```

```
rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

以及 extconf.rb:

```
require 'mkmf'

create_makefile('example')
```

编译扩展后：

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

第66.3节：编写内联C代码 - RubyInLine

RubyInline是一个框架，允许你在Ruby代码中嵌入其他语言。它定义了Module#inline方法，该方法返回一个构建器对象。你向构建器传递一个包含非Ruby语言代码的字符串，构建器将其转换为可以从Ruby调用的内容。

当提供C或C++代码（默认RubyInline安装支持的两种语言）时，构建器对象会将一个小的扩展写入磁盘，编译并加载它。你不必自己处理编译，但可以在你的主目录下的.ruby_inline子目录中查看生成的代码和编译的扩展。

直接在你的Ruby程序中嵌入C代码：

- RubyInline（作为 rubyinline gem提供）会自动创建扩展

> RubyInline无法在irb中使用

```
#!/usr/bin/ruby -w
      # copy.rb
      require 'rubygems'
      require 'inline'

      class 复制器
inline 执行 |构建器|
构建器.c <<END
void 复制文件(const char *源文件, const char *目标文件)
      {
FILE *源文件_f = fopen(源文件, "r");
        if (!源文件_f)
        {
rb_raise(rb_eIOError, "无法打开源文件 : '%s'", 源文件);
        }

FILE *目标文件_f = fopen(目标文件, "w+");
        if (!目标文件_f)
        {
```



```
  rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

And the extconf.rb:

```
require 'mkmf'

create_makefile('example')
```

After compiling the extension:

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

Section 66.3: Writing Inline C - RubyInLine

RubyInline is a framework that lets you embed other languages inside your Ruby code. It defines the Module#inline method, which returns a builder object. You pass the builder a string containing code written in a language other than Ruby, and the builder transforms it into something that you can call from Ruby.

When given C or C++ code (the two languages supported in the default RubyInline install), the builder objects writes a small extension to disk, compiles it, and loads it. You don't have to deal with the compilation yourself, but you can see the generated code and compiled extensions in the .ruby_inline subdirectory of your home directory.

Embed C code right in your Ruby program:

- RubyInline (available as the rubyinline gem) create an extension automatically

> RubyInline won't work from within irb

```
#!/usr/bin/ruby -w
    # copy.rb
    require 'rubygems'
    require 'inline'

    class Copier
    inline do |builder|
      builder.c <<END
    void copy_file(const char *source, const char *dest)
    {
      FILE *source_f = fopen(source, "r");
      if (!source_f)
      {
        rb_raise(rb_eIOError, "Could not open source : '%s'", source);
      }

      FILE *dest_f = fopen(dest, "w+");
      if (!dest_f)
      {
```

```
rb_raise(rb_eIOError, "无法打开目标文件：'%s'", dest);
      }

char buffer[1024];

int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

C 函数 copy_file 现在作为 Copier 的实例方法存在：

```
open('source.txt', 'w') { |f| f << '一些文本.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```



```
        rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
      }

      char buffer[1024];

      int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
        fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

C function `copy_file` now exists as an instance method of `Copier`:

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

第67章：调试

第67.1节：使用Pry和Byebug逐步调试代码

首先，您需要安装pry-byebug gem。运行以下命令：

```
$ gem install pry-byebug
```

在您的.rb文件顶部添加这一行：

```
require 'pry-byebug'
```

然后在您想要设置断点的地方插入这一行：

```
binding.pry
```

一个hello.rb示例：

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # 在这里设置断点
  puts "World"
end
```

当你运行 `hello.rb` 文件时，程序将在该行暂停。然后你可以使用
`step` 命令逐步执行代码。输入变量名以查看其值。使用 `exit-program` 或 `!!!` 退出调试器。

Chapter 67: Debugging

Section 67.1: Stepping through code with Pry and Byebug

First, you need to install `pry-byebug` gem. Run this command:

```
$ gem install pry-byebug
```

Add this line at the top of your `.rb` file:

```
require 'pry-byebug'
```

Then insert this line wherever you want a breakpoint:

```
binding.pry
```

A `hello.rb` example:

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # break point here
  puts "World"
end
```

When you run the `hello.rb` file, the program will pause at that line. You can then step through your code with the `step` command. Type a variable's name to learn its value. Exit the debugger with `exit-program` or `!!!`.

第68章：Ruby版本管理器

第68.1节：如何创建gemset

要创建gemset，我们需要创建一个 .rvmrc 文件。

语法：

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

示例：

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

上述命令将在应用程序根目录创建一个 .rvmrc 文件。

要获取可用gemset列表，请使用以下命令：

```
$ rvm list gemsets
```

第68.2节：使用RVM安装Ruby

Ruby 版本管理器（Ruby Version Manager）是一款命令行工具，用于简便地安装和管理不同版本的 Ruby。

- rvm install 2.3.1 例如安装机器上的 Ruby 版本 2.3.1。
- 使用 rvm list 可以查看已安装的版本以及当前设置使用的版本。

```
user@dev:~$ rvm list

  rvm rubies

=* ruby-2.3.1 [ x86_64 ]

  # => - 当前
  # =* - 当前且默认
  #  * - 默认
```

- 使用 rvm use 2.3.0 可以在已安装的版本之间切换。

Chapter 68: Ruby Version Manager

Section 68.1: How to create gemset

To create a gemset we need to create a `.rvmrc` file.

Syntax:

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

Example:

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

The above line will create a `.rvmrc` file in the root directory of the app.

To get the list of available gemsets, use the following command:

```
$ rvm list gemsets
```

Section 68.2: Installing Ruby with RVM

The *Ruby Version Manager* is a command line tool to simply install and manage different versions of Ruby.

- `rvm istall 2.3.1` for example installs Ruby version 2.3.1 on your machine.
- With `rvm list` you can see which versions are installed and which is actually set for use.

```
user@dev:~$ rvm list

rvm rubies

=* ruby-2.3.1 [ x86_64 ]

# => - current
# =* - current && default
#  * - default
```

- With `rvm use 2.3.0` you can change between installed versions.

附录 A：安装

A.1 节：在 macOS 上安装 Ruby

好消息是苹果贴心地自带了 Ruby 解释器。不幸的是，它通常不是最新版本：

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 修订版 53162) [universal.x86_64-darwin16]
```

如果你已经安装了Homebrew，可以通过以下方式获取最新的 Ruby：

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 修订版 58053) [x86_64-darwin16]
```

（如果你尝试这个命令，可能会看到更高版本。）

为了在不使用完整路径的情况下调用 Homebrew 安装的版本，你需要将 /usr/local/bin 添加到你的 $PATH 环境变量的开头：

```
export PATH=/usr/local/bin:$PATH
```

将该行添加到 ~/.bash_profile 中，确保重启系统后你会使用这个版本：

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew 将安装gem用于安装 Gems。如果需要，也可以从源码构建。Homebrew 也包含该选项：

```
$ brew install ruby --build-from-source
```

第 A.2 节：Gems

在本例中，我们将使用 'nokogiri' 作为示例 gem。'nokogiri' 以后可以替换为任何其他 gem 名称。

要使用 gems，我们使用一个名为gem的命令行工具，后跟选项如install或update，然后是我们想要安装的 gem 名称，但这还不是全部。

安装 gems：

```
$> gem install nokogiri
```

但这不是我们唯一需要的。我们还可以指定版本、安装或搜索 gems 的来源。让我们从一些基本用例（UC）开始，您以后可以提出更新请求。

列出所有已安装的 gems：

```
$> gem list
```

卸载 gems：

Appendix A: Installation

Section A.1: Installing Ruby macOS

So the good news is that Apple kindly includes a Ruby interpreter. Unfortunately, it tends not to be a recent version:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

If you have Homebrew installed, you can get the latest Ruby with:

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(It's likely you'll see a more recent version if you try this.)

In order to pick up the brewed version without using the full path, you'll want to add `/usr/local/bin` to the start of your `$PATH` environment variable:

```
export PATH=/usr/local/bin:$PATH
```

Adding that line to `~/.bash_profile` ensures that you will get this version after you restart your system:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew will install `gem` for installing Gems. It's also possible to build from the source if you need that. Homebrew also includes that option:

```
$ brew install ruby --build-from-source
```

Section A.2: Gems

In this example we will use 'nokogiri' as an example gem. 'nokogiri' can later on be replaced by any other gem name.

To work with gems we use a command line tool called `gem` followed by an option like `install` or `update` and then names of the gems we want to install, but that is not all.

Install gems:

```
$> gem install nokogiri
```

But that is not the only thing we need. We can also specify version, source from which to install or search for gems. Lets start with some basic use cases (UC) and you can later on post request for an update.

Listing all the installed gems:

```
$> gem list
```

Uninstalling gems:

```
$> gem uninstall nokogiri
```

如果我们有多个版本的 nokogiri gem，系统会提示我们指定要卸载的版本。我们会得到一个有序且编号的列表，只需输入编号即可。

更新 gems

```
$> gem update nokogiri
```

或者如果我们想更新所有

```
$> gem update
```

命令gem还有更多用法和选项可供探索。更多内容请参考官方文档。如果有不清楚的地方，请提出请求，我会补充。

附录 A.3：Linux - 从源码编译

这样你将获得最新的 Ruby，但它也有缺点。这样做的话，Ruby 将不会被任何应用程序管理。

!! 请记得更改版本号以对应您的版本 !!

1. 您需要下载一个压缩包，链接可在官方网站找到（https://www.ruby-lang.org/en/downloads/）
2. 解压压缩包
3. 安装

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

这将把 ruby 安装到 /usr/local 目录。如果您不满意此位置，可以向 ./configure --prefix=DIR 传递参数，其中 DIR 是您想安装 ruby 的目录。

A.4 节：Linux——使用包管理器安装

可能是最简单的选择，但请注意，版本不一定是最新的。只需打开终端并输入（取决于您的发行版）

在 Debian 或 Ubuntu 中使用 apt

```
$> sudo apt install ruby
```

在 CentOS、openSUSE 或 Fedora 中

```
$> sudo yum install ruby
```

你可以使用 -y 选项，这样安装时不会提示你确认，但我认为最好还是检查一下包管理器试图安装的内容，这是一种良好的做法。


```
$> gem uninstall nokogiri
```

If we have more version of the nokogiri gem we will be prompted to specify which one we want to uninstall. We will get a list that is ordered and numbered and we just write the number.

Updating gems

```
$> gem update nokogiri
```

or if we want to update them all

```
$> gem update
```

Comman `gem` has many more usages and options to be explored. For more please turn to the official documentation. If something is not clear post a request and I will add it.

Section A.3: Linux - Compiling from source

`This way you will get the newest ruby but it has its downsides. Doing it like this ruby will not be managed by any application.

!! Remember to chagne the version so it coresponds with your !!

1. you need to download a tarball find a link on an official website (https://www.ruby-lang.org/en/downloads/)
2. Extract the tarball
3. Install

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

This will install ruby into `/usr/local`. If you are not happy with this location you can pass an argument to the `./configure --prefix=DIR` where `DIR` is the directory you want to install ruby to.

Section A.4: Linux—Installation using a package manager

Probably the easiest choice, but beware, the version is not always the newest one. Just open up terminal and type (depending on your distribution)

in Debian or Ubuntu using apt

```
$> sudo apt install ruby
```

in CentOS, openSUSE or Fedora

```
$> sudo yum install ruby
```

You can use the `-y` option so you are not prompted to agree with the installation but in my opinion it is a good practice to always check what is the package manager trying to install.

附录 A.5：Windows - 使用安装程序安装

在 Windows 上安装 Ruby 最简单的方法可能是访问 http://rubyinstaller.org/ ，然后从那里下载一个可执行文件进行安装。

你几乎不需要设置任何东西，但会出现一个重要的窗口。窗口中有一个复选框，写着 将 ruby 可执行文件添加到你的 PATH。确认它已被选中，如果没有选中，请勾选，否则你将无法运行 Ruby，必须自己设置 PATH 变量。

然后一直点击下一步直到安装完成，就这样。

附录 A.6：Linux - 解决 gem install 问题

示例中的第一个用例Gems$> gem install nokogiri 可能在安装 gems 时遇到问题，因为我们没有相应的权限。这个问题可以通过多种方式解决。

第一种UC解决方案 a：

你可以使用 sudo。这样会为所有用户安装gem。这个方法应该被慎用。只应在你确定所有用户都能使用该gem时使用。通常在实际情况下，你不希望某些用户拥有 sudo的访问权限。

```
$> sudo gem install nokogiri
```

第一种UC解决方案 b

你可以使用选项 --user-install，将gem安装到你用户的gem文件夹中（通常在 ~/.gem）

```
&> gem install nokogiri --user-install
```

第一种UC解决方案 c

你可以设置 GEM_HOME 和 GEM_PATH，这样命令 gem install 会将所有gem安装到你指定的文件夹中。
我可以给你一个示例（常用方法）

- 首先你需要打开 .bashrc。使用 nano 或你喜欢的文本编辑器。

```
$> nano ~/.bashrc
```

- 然后在此文件末尾写入

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- 现在你需要重启终端或输入. ~/.bashrc以重新加载配置。这将使你能够使用gem install nokogiri，并且它会将这些 gem 安装到你指定的文件夹中。

Section A.5: Windows - Installation using installer

Probably the easies way to set up ruby on windows is to go to http://rubyinstaller.org/ and from there donwload an executable that you will install.

You don't have to set almost anything, but there will be one important window. It will have a check box saying *Add ruby executable to your PATH*. Confirm that it is **checked**, if not check it or else you won't be able to run ruby and will have to set the PATH variable on your own.

Then just go next until it installs and thats that.

Section A.6: Linux - troubleshooting gem install

First UC in the example **Gems** `$> gem install nokogiri` can have a problem installing gems because we don't have the permissions for it. This can be sorted out in more then just one way.

First UC solution a:

U can use `sudo`. This will install the gem for all the users. This method should be frowned upon. This should be used only with the gem you know will be usable by all the users. Usualy in real life you don't want some user having access to `sudo`.

```
$> sudo gem install nokogiri
```

First UC solution b

U can use the option `--user-install` which installs the gems into your users gem folder (usualy at `~/.gem`)

```
&> gem install nokogiri --user-install
```

First UC solution c

U can set GEM_HOME and GEM_PATH wich then will make command `gem install` install all the gems to a folder which you specify. I can give you an example of that (the usual way)

- First of all you need to open .bashrc. Use nano or your favorite text editor.

```
$> nano ~/.bashrc
```

- Then at the end of this file write

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Now you will need to restart terminal or write `. ~/.bashrc` to re-load the configuration. This will enable you to use `gem isntall nokogiri` and it will install those gems in the folder you specified.

鸣谢

非常感谢所有来自 Stack Overflow Documentation 的人员帮助提供此内容，
更多更改可以发送至web@petercv.com以发布或更新新内容

阿卜杜拉	第11章和第22章
亚当·桑德森	第18章
艾迪生	第28章
AJ 格雷戈里	第11章
Ajedi32	第9章
alebruck	第9章
alexunger	第14章和第31章
阿里·马苏迪安普尔	第31章
Alu	第40章和第68章
amingilani	第61章
安德烈亚·马扎雷拉	第9章
安德鲁	第55章
angelparras	第17章
安东尼·斯汤顿	第44章
阿尔曼·琼·维拉洛博斯	第19章
ArtOfCode	第3章
津田阿图尔	第15章和第27章
阿伦·库马尔·M	第15章
阿图尔·坎杜里	第19章
奥斯汀·弗恩·宋格	第33章、第47章和第66章
自动	第9章
br3nt	第9、11、17、18、20、25和26章
C点 StrifeVII	第19和49章
CalmBit	第1章
查兰·库马尔·博拉	第21章
查理·伊根	第11章和第19章
克里斯	第21章
克里斯托夫·佩茨尼格	第19章
克里斯托弗·厄兹贝克	第19章
coreyward	第20章
D	第9章和第17章
daniero	第9章和第17章
DarKy	第17章
达尔潘·查特拉瓦拉	第1章
大卫·格雷森	第1、9、11、17、19和60章
大卫·隆·麦迪逊	第18章
davidhu2000	第9、11和25章
DawnPaladin	第1、9、34和67章
Dimitry_N	第17章
迪维娅·夏尔马	第31章
divyum	第19章
djaszczurowski	第31章
小玩意儿	第29章
多里安	第37章
埃勒尼安	第17、25和28章
伊莱·萨多夫	第9、14、20、21和59章
engineersmnky	第20章

Credits

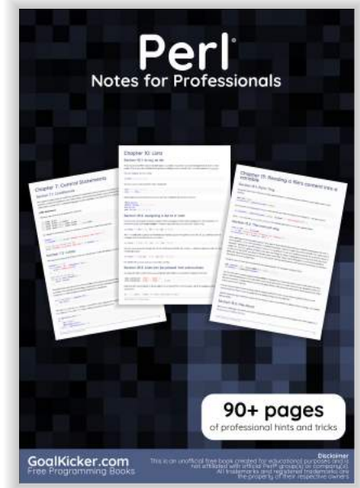
Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Abdullah	Chapters 11 and 22
Adam Sanderson	Chapter 18
Addison	Chapter 28
AJ Gregory	Chapter 11
Ajedi32	Chapter 9
alebruck	Chapter 9
alexunger	Chapters 14 and 31
Ali MasudianPour	Chapter 31
Alu	Chapters 40 and 68
amingilani	Chapter 61
Andrea Mazzarella	Chapter 9
Andrew	Chapter 55
angelparras	Chapter 17
Anthony Staunton	Chapter 44
Arman Jon Villalobos	Chapter 19
ArtOfCode	Chapter 3
Artur Tsuda	Chapters 15 and 27
Arun Kumar M	Chapter 15
Atul Khanduri	Chapter 19
Austin Vern Songer	Chapters 33, 47 and 66
Automatico	Chapter 9
br3nt	Chapters 9, 11, 17, 18, 20, 25 and 26
C dot StrifeVII	Chapters 19 and 49
CalmBit	Chapter 1
Charan Kumar Borra	Chapter 21
Charlie Egan	Chapters 11 and 19
Chris	Chapter 21
Christoph Petschnig	Chapter 19
Christopher Oezbek	Chapter 19
coreyward	Chapter 20
D	Chapters 9 and 17
daniero	Chapters 9 and 17
DarKy	Chapter 17
Darpan Chhatravala	Chapter 1
David Grayson	Chapters 1, 9, 11, 17, 19 and 60
David Ljung Madison	Chapter 18
davidhu2000	Chapters 9, 11 and 25
DawnPaladin	Chapters 1, 9, 34 and 67
Dimitry_N	Chapter 17
Divya Sharma	Chapter 31
divyum	Chapter 19
djaszczurowski	Chapter 31
Doodad	Chapter 29
Dorian	Chapter 37
Elenian	Chapters 17, 25 and 28
Eli Sadoff	Chapters 9, 14, 20, 21 and 59
engineersmnky	Chapter 20

西蒙·索里亚诺	第49章
西蒙娜·卡莱蒂	第1、11、14、15、18、19、22、25和53章
snonov	第5章
苏拉布·乌帕迪亚伊	第49章
spencer.sm	第25章
小队	第9章
史蒂夫	第1章和第17章
stevendaniels	第13章、第19章、第25章和第58章
suhao399	第36章
苏里亚	第33章
thesecretmaster	第42章
汤姆·哈里森 Jr	第3章和第27章
汤姆·洛德	第9章、第14章、第17章、第19章和第44章
托特·扎姆	第1章
乌芒·拉古万希	第54章
user1213904	第31章
user1489580	第44章
user1821961	第62章
user2367593	第6章
Vasfed	第11章、第26章和第35章
Ven	第17章和第19章
vgoff	第17章
Vidur	第43章
Vishnu Y S	第1章
wirefox	第14章和第19章
wjordan	第11章
xavdid	第57章
约纳塔·阿尔梅达	第34章
尤尔	第17章和第27章
Zaz	第18章和第47章

Simon Soriano	Chapter 49
Simone Carletti	Chapters 1, 11, 14, 15, 18, 19, 22, 25 and 53
snonov	Chapter 5
Sourabh Upadhyay	Chapter 49
spencer.sm	Chapter 25
squadette	Chapter 9
Steve	Chapters 1 and 17
stevendaniels	Chapters 13, 19, 25 and 58
suhao399	Chapter 36
Surya	Chapter 33
thesecretmaster	Chapter 42
Tom Harrison Jr	Chapters 3 and 27
Tom Lord	Chapters 9, 14, 17, 19 and 44
Tot Zam	Chapter 1
Umang Raghuvanshi	Chapter 54
user1213904	Chapter 31
user1489580	Chapter 44
user1821961	Chapter 62
user2367593	Chapter 6
Vasfed	Chapters 11, 26 and 35
Ven	Chapters 17 and 19
vgoff	Chapter 17
Vidur	Chapter 43
Vishnu Y S	Chapter 1
wirefox	Chapters 14 and 19
wjordan	Chapter 11
xavdid	Chapter 57
Yonatha Almeida	Chapter 34
Yule	Chapters 17 and 27
Zaz	Chapters 18 and 47

你可能也喜欢



You may also like