

Rubylearning 教程-中文版

作者: Satish Talim 来自: <http://rubylearning.com>

译者: 想飞的马 来自: <http://mayabin.javaeye.com>

版本号: Draft-20071020 (完成至 “Summary 5”)

这是一份来自 <http://rubylearning.com> 的 ruby 教程, 该教程即包含 ruby 的基本语法, 又包含高级应用, 还不乏例子程序和项目实战。如果你想从头开始学 ruby, 可以跟着教程, 由浅入深的进行。

这份教程的作者 Satish Talim 致力于 ruby 的教学及推广工作, 他非常乐意把这份教程翻译成中文。目前, 这份教程已有意大利语, 西班牙语, 法语和中文正在翻译中。作者不收取任何费用, 只要保留他的名称以及网站地址即可。非常可敬。

同样, 这份中文版, 也可以免费下载和传播, 但是必须要注明出处, 保留作者和译者的署名权。

由于水平有限, 翻译中难免有错误和遗漏, 欢迎[指正](#)。英文足够好的朋友, 最好直接阅读英文版本。

第一部分： 核心 Ruby

这一部分主要讲 ruby 的基本语法及一些简单的例子，不包含高级应用。

第一章 介绍

Ruby Study Notes 是一个一步一步教你如何用 ruby 编程的教程，主要介绍 ruby 的基本语法。你可以边读边跟着做里面的例子，这些例子也都非常简单。

<http://rubylearning.com> 网站有这份教程的英文原版，如果英文足够好的话，最好直接读英文，以免翻译遗漏。

在这份教程中，会使用带背景的文字做一些特殊说明或者解释，比如下面这样：

这是一个解释说明，可以略过不看

译者注：

蓝色的部分呢，代表是译者的补充

这份教程里的代码样例如下图所示：

```
def hello  
  puts 'hello'  
end
```

但是请记住 ruby 是一种语法灵活的语言，解决同一个问题可以有好多途径。

第二章 安装

什么是 ruby?

Ruby 是一种跨平台，面向对象的解释型语言。他是本着简单实用的原则设计出来的，他的创始人 Matz 说，“我想用最省力的方法来写程序，这是我设计 ruby 的初衷。在 ruby 被广泛使用后，大家也是这么认为的”。工具都是“懒人”发明的，这句话真对。

2004 年，随着 David Heinemeier Hansson 对 Ruby on Rails Web 应用框架的介绍，ruby 开始火起来了。

日本人 Yukihiro Matsumoto，也就是 Matz，1993 年创建 ruby。有关 ruby 的历史可以见介绍 [An interview with him in 2001](#)

既然说他简单实用，Ruby 能帮着做哪些事情？

在 David Black 的《Ruby for Rails》一书中，他总结了，对于一个 Rails 开发者来说，主要有四点帮助：

1. By helping you know what the code in your application (including Rails boilerplate code) is doing
2. By helping you do more in, and with, your Rails applications than you can if you limit yourself to the readily available Rails idioms and techniques (as powerful as those are)
3. By allowing you to familiarize yourself with the Rails source code, which in turn enables you to participate in discussions about Rails and perhaps even submit bug reports and code patches
4. By giving you a powerful tool for administrative and organization tasks (for example, legacy code conversion) connected with your application

下载 ruby 和编辑器

Ruby 是一个开源的编程语言，提供了多种版本，可以运行在多种操作系统和平台上。你可以把在一种平台下编写的代码直接移植到另一种平台下。

不过最简单的就是在 [Windows](#) 下的安装了，下载安装包，双击打开。安装程序除了会安装必须的文件外，还会自动设置环境变量，还安装了文档和 SciTE 文本编辑器。

SciTE 是一个类似于 Emacs 的文本编辑器。

Ruby 的发行版中，子版本号是偶数的是稳定版，例如，1.6，1.8 等等。

使用 eclipse 的朋友，可以下载 [RDT](#) 插件，非常不错。

其他平台的下载和安装，可以分别参考 [这里](#)，[这里](#)，[这里](#)。

我 [下载](#) 安装了 linux 版本，源代码版本，需要编译安装。安装方法和标准方法一样。

1. `tar xvzf ruby-xxx.tar.gz`
2. `./configure --prefix=$HOME`
3. `make`
4. `make install`

安装完后，可以使用 `ruby -v` 测试版本号。

Ruby 安装目录介绍

以 windows 下为例，假设安装在 **c:/ruby** 下

c:/ruby/bin	可执行文件
c:/ruby/lib/ruby/1.8	ruby 库文件
c:/ruby/lib/ruby/1.8/i386-mswin32	和平台相关的库文件，以 dll 或 so 形式存储
c:/ruby/lib/ruby/site_ruby	自己写的代码或者第三方库目录
c:/ruby/lib/ruby/gems	Ruby-Gems
c:/ruby/src	ruby 源文件
c:/ruby/samples/RubySrc-1.8.6/sample	例子程序

第三章 第一个 Ruby 程序

rubylearning 的作者推崇使用 SciTE 来编写程序，下面就介绍一下 SciTE 的使用。

首先要配置一下：

Options/Open Global Options，找到'tabsize'，设置 tabsize=2 和 indent.size=2。

设置 position.width=-1 和 position.height=-1，这样能是窗口全屏。保存。

Ctrl+Shift+I 打开 Indentation Settings 窗口，设置 tabsize=2 和 indent.size=2。

好了，就配置这些，开始写程序。

创建一个文件夹，比如 d:/rubypprogram 用来存放所有 ruby 程序。

打开 SciTE，编写如下代码：

```
puts 'hello'
```

选择 File/Save As...，记住，所有的 ruby 源文件都以.rb 作为扩展名，文件名和目录名要小写，对应 ruby 中的类和模块。写好程序后，按 F5 运行。

ruby 程序的执行是从上到下一行行按顺序执行的。puts 标准输出语句，已经包含在 ruby 核心中，它能够自动换行，类似于 java 中的 system.out.println()。

a.圆括号在方法调用中是可选的，下面的写法都对。

foobar

foobar()

foobar(a, b, c)

foobar a, b, c

b.在 ruby 中，不管是整数，字符串，还是对象，都看作是对象。并且每一个对象都已经内置了一系列方法。

使用时，在对象名后加“.”，然后跟方法名。有些方法比如 puts，gets 可以直接使用，前面不用带对象名。

这些核心方法都内置在 ruby 核心模块中，可以被任何 ruby 对象调用。

当你运行一个程序时，一个叫 main 的 class 对象被自动创建，由它作为程序入口，当然这个对象可以调用核心方法。

下面的程序可以验证上面的论述（不要害怕现在看不懂程序）。

```
puts 'I am in class = ' + self.class.to_s
puts 'I am an object = ' + self.to_s
print 'The object methods are = '
puts self.private_methods.sort
```

从上面程序，发现些什么？

1.对于 c 和 java 程序员来说，不用写 main()方法了。

2.字符串就是用单括号或双括号扩起来的连续字符，单括号比双括号要高效（后面会提到）。

3.ruby 是解释型语言，所以修改程序后，不需要重新编译。

4.ruby 版本号中，子版本号是偶数的代表稳定版本，比如 1.6，1.8。

5.ruby 规范声明，文件/目录的名称要小写，并且要和类/模块的名称对应，类文件要以.rb 作

为扩展名。

熟悉 eclipse 的程序员可以使用 **RDT** 插件，使用习惯和你编写 java 代码时是一样的，具体有多像，使用一下就有体会了。

第四章 特性

ruby 的一些特性如下：

- 1.格式灵活。
- 2.大小写敏感。
- 3.单行注释以“#”开头，多行注释包括在“=begin”和“=end”中。
- 4.分行符：多行语句写在同一行时，必须用分号隔开，行末尾可以不写，换行符相当于分号。单行语句要书写到多行时，要使用“\”作为结尾。
- 5.关键字。ruby 中有 38 个保留关键字，和其他语言一样，不能把他们作为变量，类或者方法名。在有的语言中，0，null，空字符串可能被当作 false，但是在 ruby 中，他们被当作 true，除了保留关键字“false”和“nil”外，其他都是 true。

文档：全部ruby在线文档，可以访问[这儿](#)。Ruby Cheat Sheet在[这儿](#)。Windows下的安装指导可以看[fxri - Interactive Ruby Help & Console](#)。

第五章 ruby 中的数值类型

让我们看看 ruby 中的数值类型。在 ruby 中，不带小数点的叫整数，带小数点的叫浮点数或实数。

看下面的程序 p002rubynumbers.rb。

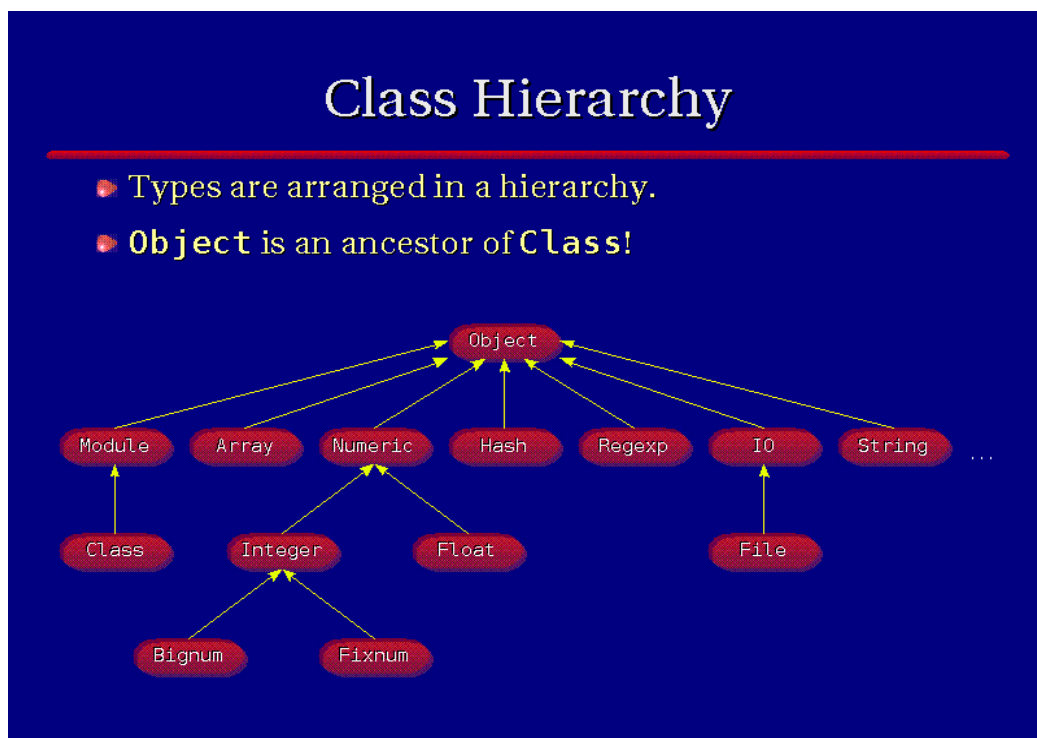
```
=begin
  Ruby Numbers
  Usual operators:
  + addition
  - subtraction
  * multiplication
  / division
=end

puts 1 + 2
puts 2 * 3
# Integer division
# When you do arithmetic with integers, you'll get integer answers
puts 3 / 2
puts 10 - 11
puts 1.5 / 2.6
```

Fixnum 或者 Bignum 类的对象都是整数，他们的区别只是位长不同，他们继承自 Integer--Numeric。

浮点数是 Float 类的对象，包括了系统中的 float 和 double 类型。(后面章节会详述)。

类的继承关系见下图：(转载自: [Donald Craig](#))



下面是 Peter Cooper 的《Beginning Ruby》书中的例子，(不要怕看不懂)。

```
rice_on_square = 1
64.times do |square|
  puts "On square #{square + 1} are #{rice_on_square} grain(s)"
  rice_on_square *= 2
end
```

计算 2 的 64 次方，可以得到一个用兆来衡量的数。

这个例子说明 ruby 可以处理很大数量级的数，不像有的语言有位数的限制。
Fixnum 类处理稍小数量级的数，Bignum 处理大数量级的数。ruby 会自动处理 Fixnum 和 Bignum，你只要关注运算就可以了，不用担心位数。
但是有时候运算结果依赖于你的系统架构，但是 ruby 依然会替你自动调整。

运算符和优先级

下图是按优先级从高到低排列的运算符，转自Hal Fulton - [The Ruby Way](#)。

::	Scope
[]	Indexing
**	Exponentiation
+ - ! ~	Unary pos/neg, not,...
* / %	Multiplication, Division...
+ -	Addition, subtraction...
<< >>	Logical shifts...
&	Bitwise and
^	Bitwise or, xor
> >= < <=	Comparison
= == === <=> != =~ !~	Equality, inequality...
&&	Boolean and
	Boolean or
... ..	Range operators
= (also +=, -=,...)	Assignment
?:	Ternary decision
not	Boolean negation
and, or	Boolean and, or

- a.ruby 中没有++和--运算符
- b.括号可以和其他运算符一起使用，括在括号中的表达式优先运算。

ruby 中的求模运算如下，

```
puts (5 % 3)      # prints 2
puts (-5 % 3)     # prints 1
puts (5 % -3)     # prints -1
puts (-5 % -3)    # prints -2
```

作业：写一个程序计算一年有多少分钟（不考虑闰年）。

第六章 字符串类型

字符串是用单括号或者双括号括起来的字符组合。‘’中间没有任何字符，叫做空字符串。程序 p003rubystings.rb 是字符串的例子：

```
=begin
  Ruby Strings
  In Ruby, strings are mutable
=end

puts "Hello World"
# Can use " or ' for Strings, but ' is more efficient
puts 'Hello World'
# String concatenation
puts 'I like' + ' Ruby'
# Escape sequence
puts 'It\'s my Ruby'
# New here, displays the string three times
puts 'Hello' * 3
# Defining a constant
# More on Constants later, here
# http://rubylearning.com/satishtalim/ruby\_names.html
PI = 3.1416
puts PI
```

在 ruby 中，字符串是变长的。它会根据需要扩展长度，这样做并不会耗费太多的时间和内存。ruby 是按字节存储字符串的。

还有一种特殊的字符串，用``括起来。

```
| puts `dir`
```

括起来的字符串会作为系统命令发送给操作系统。比如上面的‘dir’在 windows 下会执行列目录的动作，并把结果返回给 ruby 程序。

第七章 变量和定义变量

为了把数值或字符串保存在内存中供后面程序使用，需要给他们命名。

程序员把这个过程叫定义变量，定义的名称叫变量。

只有当解释器看到有变量定义后，这个变量才会产生，也就是说，不会给变量预先分配地址和空间。

```
s = 'Hello World!'
x = 10
```

```
# p004stringusage.rb
# Defining a constant
PI = 3.1416
puts PI
# Defining a local variable
myString = 'I love my city, Pune'
puts myString
=begin
  Conversions
  .to_i, .to_f, .to_s
=end
var1 = 5;
var2 = '2'
puts var1 + var2.to_i
# << appending to a string
a = 'hello '
a<<'world.
I love this world...'
puts a
=begin
  << marks the start of the string literal and
  is followed by a delimiter of your choice.
  The string literal then starts from the next
  new line and finishes when the delimiter is
  repeated again on a line on its own.
=end
a = <<END_STR
This is the string
And a second line
END_STR
puts a
```

变量的命名有一定规则：以小写字母或下划线开头，变量中只能包含字母，数字和下划线。关键字不能作为变量名使用。

当 `ruby` 解释器读到一个单词的时候，他会把他解释成变量名，方法名或者保留关键字中的一种。

如果单词的后面跟一个“=”，说明是一个变量；如果是一个关键字，那就只能作为关键字使用；其他情况视为方法名。

在下面的例子中：

```
x = "100".to_i
```

“.”的意思是方法 `to_i` 被字符串“100”调用。

字符串“100”是方法的调用者。

“.”前面的对象和后面的方法是调用和被调用的关系。

总结一

截至上章结束，我们总结了以下要点。

1. 我们的例子来自 Windows 平台，ruby 版本号是 1.8.6
2. Ruby 是一个解释型语言。
3. 记住在 Ruby 中，解决一个问题会有多种方法
4. 例子程序都是在 SciTE 编辑器中运行的，如何配置它，请看 "第一个 ruby 程序"。
5. 所有的 ruby 源程序都以 .rb 做扩展名。
6. ruby 程序的执行是从上到下一行行顺序执行的。
7. 特性：格式灵活，大小写敏感，两种注释方式，语句分行符，38 个关键字。
8. 只有保留关键字 false 和 nil 代表 false，其他都是 true。
9. ruby 有丰富的在线文档。
10. puts (s 代表 string; puts 意思是输出字符串) 输出它后面内容到控制台，并且自动换行。
11. 方法调用中的括号是可选的。下面的方法调用都是合法的: `foobar foobar()` `foobar(a, b, c)`
`foobar a, b, c`
12. 在 Ruby 中，带小数点的数叫浮点数或实数，不带小数点的叫整数。
13. 常用的操作符，`+-×/`
14. 没有 `++` 和 `--` 运算符
15. 括号中的表达式优先运算。
16. 了解求模运算规则。
17. 当你用整数运算时，结果也是整数类型的。
18. 字符串是包括在单引号或双引号中间的字符组合。
19. 在 Ruby 中，字符串是变长的，可以按需扩展，这样做不会耗费太多时间和内存。
20. 使用 `+` 可以把两个字符串连在一起。
21. `"\"` 是转义字符，比如: `"\"`, `\\`, `\n`
22. `"` 是一个空字符串。
23. 如果得到一个编译错误- `#<TypeError: cannot convert Fixnum into String>`，意思是你不能把一个整数加到字符串中，或者不能用一个字符串乘另一个字符串。
24. 常量以大写字母开头。
25. 变量不会预留地址和空间，只有读到变量定义后才会定义变量。给变量初始化为 `nil` 是一个好习惯。
26. `x, y = y, x` 会得到 `y` 的值，`y` 会得到 `nil`；变量和值的数目要对应。值的数目少了，用 `nil` 补；如果多了，丢弃。
27. 本地变量要以小写字母或者下划线开头，包含并且只能包含字母，数字和下划线。
28. `.to_i`, `.to_f`, `.to_s` 用来转换成 `integer`, `float`, `string`。
29. 操作符 `<<` 用来 `append` 字符串。

第八章 作用域

作用域指的是变量的生命空间或者说作用范围。不同类型的变量有不同的作用域。本章主要讲两种，全局变量（Global）和局部变量（Local）。

全局作用域和全局变量

全局作用域指的是能够覆盖整个应用程序运行期（从应用程序开始运行到结束）。全局变量的生命周期是全局作用域。

全局变量以一个美元符号（\$）开头，在整个应用程序运行期，任何地方都可以使用全局变量。但是稍微有经验的人都知道，应该尽量少的使用全局变量，以减少空间的占用。

内建的全局变量

ruby 解释器在开始运行的时候初始化了一部分全局变量，在应用程序的任何地方都可以调用这些内建全局变量。

比如 '\$0' 代表的是所运行应用程序的文件名称；'\$:' 代表的是默认的文件搜索路径；'\$\$_' 代表的是 ruby 程序的进程 id。

可以查看文档来了解更多的内建全局变量。

本地作用域

本地作用域存在于下面几种情况中：

- 1.最顶级的程序。---我的理解是 main 程序。
- 2.类和模块。
- 3.方法。

本地作用域的介绍有些抽象，可以在后面使用过程中慢慢体会。

第九章 输入方法

前面我们已经学到了使用 `puts` 命令把字符串输出，那么输入命令又是什么呢？使用 `gets` 和 `chomp` 方法，下面是他们的例子程序：

```
# gets and chomp
puts "In which city do you stay?"
STDOUT.flush
city = gets.chomp
puts "The city is " + city
```

运行上面的程序，需要你输入城市名称，然后回车，就会显示你输入的城市名称。

STDOUT 是一个全局常量，代表的是标准输出流。`flush` 方法的作用是把所有 `io` 里面缓冲的数据输送给 **STDOUT**。

`gets` 方法得到输入字符串（包含一个 `\n` 回车符），**chomp** 方法把字符串中的 `\n` 回车符去掉。

在 **RAILS** 中，数据来源有很多。

对于一个 **Rails** 开发者来说，典型的来源是数据库，**Rails** 会为你抓取数据。

如果你是一个用户，当你从键盘输入时，**Web form** 会接收你的输入。所以你很少会用到这些技巧。

作业：输入一个华氏温度度数，转化成摄氏度输出（要求保留 2 位小数）。

Tip:

保留 2 位小数，一个方法是使用内置的 `format` 方法，比如 `x = 45.5678`，使用 `format("%.2f", x)` 后，会得到 `45.57`。另一个方法是使用 `round` 方法，`puts (x*100).round/100.0`

第十章 命名规则

ruby 中需要给常量，变量，方法，类，模块命名，保留的关键字不能作为这些名称使用。

ruby 可以根据名称的第一个字符来区分是什么名称。

名称可以是大小写字母，数字，下划线的任意组合（这些字符是合法字符），但是只能以字母或下划线开头。

变量命名规则：

ruby 中的变量可以包含任何类型的数据。使用变量时不需要类型声明，变量名决定了它的作用域。

本地变量（在对象中定义）以小写字母或者下划线开头，后面跟合法字符。

实例变量以 '@' 开头，后面至少跟一个合法字符。

在类中定义的变量，以 '@@'，两个 '@' 开头，后面至少跟一个合法字符。

全局变量以 '\$' 开头，后面跟合法字符，也可以使用 '\$-' 开头，后跟一个字母（是字母）。

常量命名规则：

常量以大写字母开头，后跟合法字符。类和模块的命名规则和常量一样。

方法命名规则：

方法名以小写字母开头，后跟合法字符。但是在方法名的结尾，可以跟 '?' 或 '!'（! 或者 bang 代表这是一个比较危险的方法，说危险是相对和它重名，但是不带 '!' 的方法。后面会有专门的章节介绍 'Bang Method'）。

Ruby 规范说明如果一个变量或者方法名由多个词组成，需要用下划线隔开。如果是类，模块或者常量名称由多个词组成时，不需要用下划线，只要把后续词的第一个字母大写就可以。需要注意的是，一个变量可以是任何类型对象的引用，而且引用是可以动态变换的。常量也是对象的引用，可以在类和模块中定义常量，但是在方法中不可以。

Ruby 允许修改常量的值，但是会给一个警告信息。

变量只是对象的引用，引用失效后，ruby 会自动做垃圾收集。

下面的例子可以说明 ruby 程序的动态类型特性：

```
# Ruby is dynamic
x = 7          # integer
x = "house"    # string
x = 7.5        # real
puts x
# In Ruby, everything you manipulate is an object
'I love Ruby'.length
```

Ruby 中的基本类型包括 Numeric (子类型包括 Fixnum, Integer, and Float), String, Array, Hash, Object, Symbol, Range, and RegEx。在 ruby 中，不需要维护对象的类型，如果它是整数，那就是一个整型类型；如果是字符串，就是字符串类型。**Class** 方法用来得到对象的类型。

```
s = 'hello'
s.class # String
```

在 Ruby 中，需要处理的都是对象，并且返回的也是对象。

```
5.times { puts "Mice!\n" } # more on blocks later  
"Elephants Like Peanuts".length
```

第十一章 创建方法

如果说对象是名词的话，那么方法就是动词，代表了一个对象要执行什么动作。方法是和对象相对应的，“.”前面的对象就是方法所属的对象。但是有时候可以省略对象名，比如 `puts`，`gets` 是直接使用的，前面没有对象名。在对象内部调用对象自己的方法时，是可以省略对象名的。

但是有时候我们不知道在哪个对象里（当前对象），可以使用 `self` 方法查看。

注意：ruby 会自动生成一个 `main` 对象。

```
puts self
```

更多`self`细节，查看[这儿](#)。

下面我们学着写一个方法。方法体需要包括在 `def` 和 `end` 中间，参数列表需要括在括号里。

注意：方法中并没有定义返回类型，一个方法的返回值就是该方法的最后一行。

多个方法定义之间最好用空行隔开，方便区分。

ruby 规范说明，方法的参数需要用括号括起来。

但是看以前的例子，比如 `puts`，`p`，`gets` 等方法的参数都没有括起来。以后会讲到参数不需要括起来的情况。

```
def hello
  puts 'Hello'
end
#use the method
hello

# Method with an argument - 1
def hello1(name)
  puts 'Hello ' + name
  return 'success'
end
puts(hello1('satish'))

# Method with an argument - 2
def hello2 name2
  puts 'Hello ' + name2
  return 'success'
end
puts(hello2 'talim')
```

这段代码在我机器上的运行结果如下：

```
Hello
test2.rb:32 warning: parenthesize argument(s) for future version
Hello satish
success
Hello talim
success
```

可以看到，参数没有括在括号中，一样输出了结果，但是给出了警告。

定义方法时，可以给参数定义默认值。如果调用方法时，没有传参数值过来，方法会使用这些默认值。看下面的例子：

```
# interpolation refers to the process of inserting the result of an
# expression into a string literal
# the interpolation operator #{...} gets calculated separately
def mtd(arg1="Dibya", arg2="Shashank", arg3="Shashank")
  "#{arg1}, #{arg2}, #{arg3}."
end
puts mtd
puts mtd("ruby")
```

在我的机器上，输出结果如下：

```
Dibya, Shashank, Shashank
ruby, Shashank, Shashank
```

在上面的例子中，有‘#{...}’这样的代码，它的作用是，计算{}中表达式的结果，并把结果转换成字符串。运行这样的代码后，可以得到结果字符串，并不会看到’#{...}’在输出结果中。

再比如下面的例子：

```
puts "100 * 5 = #{100 * 5}"
```

输出结果为：

```
100 * 5 = 500
```

在 ruby 中可以给一个方法指定别名。别名对应的是原方法的一个拷贝，如果原方法改变了，别名不会跟着变化，看例子：

```
def oldmtd
  "old method"
end
alias newmtd oldmtd
def oldmtd
  "old improved method"
end
puts oldmtd
puts newmtd
```

运行结果为(别名并没有变化):

```
old improved method
old method
```

ruby 支持可变数量参数的用法吗? 答案是肯定的。

```
# variable number of parameters example
# The asterisk is actually taking all arguments you send to the method
# and assigning them to an array named my_string as shown below
# The do end is a Ruby block which we talk in length later
def foo(*my_string)
  my_string.each do |words|
    puts words
  end
end
foo('hello','world')
foo()
```

'*'会接受传过来的所有参数, 并且把他们放到数组 my_string 中。

do end是[程序块](#)。

通过使用 '*', 即可以传递多个参数, 也可以一个参数也不传。

上面例子的结果如下:

```
hello
world
```

如果你想包含多个可选参数的话, 一定要把可选参数放在最后, 比如

```
def opt_args(a,b,*x) # right
def opt_args(a,*x,b) # wrong
```

那么一共可以给方法传递多少参数? 数目是没有限制的, 这个问题, 可以参考[这篇文档](#)。
参数的排列顺序是从左至右排列的, 比如:

```
def mtd(a=99, b=a+1)
  [a,b]
end
puts mtd
```

那么传过去的参数是值传递还是引用传递? 请参考[这儿](#)。再看下面的例子:

```
def downer(string)
  string.downcase!
end
a = "HELLO"      # -> "HELLO"
downer(a)        # -> "hello"
puts a
```

结果是: "hello"。可以看出是按引用传递, 但是这个问题一直在讨论中。

重磅方法：(bang or ! methods)

Ruby 中，方法名后跟一个 ‘!’ 的方法叫重磅方法。带上 ‘!’ 标志，就说明这个方法是比较危险的，当然这个危险是相对于同名不带 ‘!’ 的方法而言的。

你会经常发现这样成对的方法，有相同的方法名，但是一个带 ‘!’ 标志，一个不带。执行不带 ‘!’ 的方法后，会返回一个全新的对象。但是如果带 ‘!’ 的方法，返回的是原来的对象。这就是他们的区别。

看几个这样的例子，数组的 `sort/sort!` 方法，字符串的 `upcase/upcase!`，`chomp/chomp!` 方法，字符串和数组的 `reverse/reverse!` 方法。

方法这一章涉及到的知识点比较多，从方法定义，别名，参数的类型和个数，以及重磅。幸好，每个知识点后面都有例子。运行这些例子，分析一下结果，对理解这些知识点，还是很有帮助的。

总结二

1. 尽量少使用全局变量，因为全局变量能在整个应用程序运行期生存，占用资源。全局变量以 '\$' 开头。Ruby 解释器在开始运行时，能自动初始化一定数目的内建全局变量。
2. `gets` 和 `chomp` 方法用来接收和处理用户输入。
3. `gets` 方法返回用户输入的字符串，并加上一个换行符，`chomp` 方法删除字符串中的换行符。
4. `STDOUT` 是一个全局常量，代表的是标准输出流。`flush` 方法能把缓存在 `io` 中的数据释放出来。
5. 内置的 `format` 方法能够格式化数据，比如 `format("%.2f",x)` 。
6. 在 Ruby 中，可以给常量，变量，方法，类，模块命名，并且可以通过名称的第一个字符确定是什么名称。
7. 小写字母指的是从 'a' 到 'z'，下划线是 '_'，大写字母指的是从 'A' 到 'Z'，数字指的是从 '0' 到 '9'，这些都是合法字符。
8. 名称以小写字母或者大写字母或者下划线开头，后面可以跟大小写字母，数字以及下划线的任意组合。
9. 在 ruby 程序中，不需要给变量做任何声明（类型，作用域）。通过变量名可以区分出变量的作用域。
10. 谨记本地变量，实例变量，类和模块，全局变量，常量，以及方法的命名规则。
11. "?" 和 "!" 是唯一的可以跟在方法名后面的不合法字符。
12. ruby 规范说明，如果方法或变量名由多个词组成，词和词之间用 "_" 隔开。如果是类，模块或者常量，不使用 '_'，只需要把单词的首字母大写就可以了。比如，`my_variable`, `MyModule`, `MyClass`, `MyConstant`。
13. 变量可以引用任何类型的对象，并且同一个变量可以在不同时刻引用不同类型的对象。
14. ruby 中的变量都是对象的引用，变量生效后，ruby 会自动做垃圾收集。
15. ruby 是动态类型的语言，你所操作的都是对象，返回的结果也是对象。
16. ruby 中的基本类型有 `Numeric`(包括 `Fixnum`, `Integer` 和 `Float` 子类型), `String`, `Array`, `Hash`, `Object`, `Symbol`, `Range` 和 `RegEx`。
17. 通过 `self` 方法可以查看在哪个对象中。
18. 使用 `def` 和 `end` 来声明方法。参数是扩在括号里的本地变量。
19. 方法不需要声明返回类型，返回的是最后一行的值。
20. 最好在不同的方法声明间加一行空行。
21. ruby 规范说明要把参数扩在括号中。
22. 方法名后面唯一可以跟的非合法字符就是 '!' 和 '?'。
23. 以 '!' 结尾的方法总是返回对象本身，而不是一个拷贝。
24. 可以使用 '=' 给参数指定默认值，如果调用方法时，没有传参数值，会使用默认值。
25. `#{...}` 返回的是括号中的表达式的结果。
26. 方法的别名能给方法生成一个拷贝，所以即使原方法发生变化，别名不会跟着变化。
27. 方法的参数个数可以是变化的。
28. 方法的参数个数没有限制。
29. 参数的排列顺序是从左到右。
30. 方法是按值传递还是引用传递，一直还在讨论中，等待结果吧。

第十二章 method_missing 方法

当调用一个对象的方法时，会根据方法名查找是否有这个方法存在。如果不存在这个方法，会抛出一个 NoMethodError 异常。

但是可以定义一个 method_missing 方法，来处理找不到方法这种情况，并且可以传递参数给这个方法。

method_missing 提供的是一个友好的处理办法，它拦截这种发生了错误并且没有返回结果的信息（我们马上就要讲到[自定义类](#)）。看下面的例子：

```
class Dummy
  def method_missing(m, *args)
    puts "There's no method called #{m} here -- please try again."
  end
end
Dummy.new.anything
```

输出结果是：

```
>ruby tmp.rb
There's no method called anything here -- please try again.
>Exit code: 0
```


第十三章 更多字符串方法

有很多方法可以用于字符串对象。比如 `reverse` 方法得到一个反向排列的字符串（原来的字符串不会改变）。`length` 方法得到字符串的长度。`upcase` 方法把字符串中的所有字母转换成大写。`downcase` 方法把字符串中的所有字母转换成小写。`swapcase` 方法也是转化字符串中字母的大小写，但是它会大写字母换成小写，小写字母换成大写。`capitalize` 方法把字符串的第一个字母换成大写，其他都换成小写。`slice` 方法返回字符串的一个子字符串。

`upcase`, `downcase`, `swapcase` 和 `capitalize` 方法返回的都是原字符串的一个拷贝。但是 `upcase!`, `downcase!`, `swapcase!` 和 `capitalize!` 这些重磅方法，都是直接修改并返回原对象。如果你确认没有必要保留原字符串，可以使用它们。

前面我们提到过字符串是括在单引号或者双引号中间的字符序列，当时我们还提到双引号和单引号是不同的。那么他们的区别是什么？

他们的主要区别就是构造字符串的过程。使用单引号时很简单，几乎不做什么检查，就构造完成。但是使用双引号就要复杂一些。首先它会检查字符串中是否有转义字符（比如 `\n`, `\r`），如果有，它会转义字符转换成特定的二进制值。然后它还会检查是否有 `#{expression}` 存在，如果有，它会先计算表达式，并返回一个字符串。

在下面的例子中，第二个方法去掉了临时变量 `result` 和 `return` 语句，因为 `ruby` 方法可以自动返回最后一行的值，所以去掉他们不会对程序有影响。

```
def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Satish')

# modified program
def say_goodnight2(name)
  "Good night, #{name}"
end
puts say_goodnight2('Talim')
```

ruby 怎样管理内存中的字符串？有专门的字符串管理池吗？

所有的字符串都是类 `string` 的对象，有多于 75 个方法在 `string` 类中，用于处理字符串。

如果你去读[Ruby User's Guide](#)，就会看到“我们不必考虑字符串的位长问题，因为他们是变长的，不需要为字符串做特殊的内存管理。”

列出一个类或对象的所有方法

`String.methods.sort` 列出类 `string` 的所有方法。

`String.instance_methods.sort` 列出 `string` 实例的所有实例方法。

`String.instance_methods(false).sort` 列出 `string` 实例的所有非继承的实例方法。

比较两个字符串

ruby 中比较字符串的方法有好几种。其中最普通的就是“`==`”，它比较两个字符串的内容是

否相同；还有 `String.eql?`，它比较的也是两个字符串的内容。还有 `String.equal?`，它比较两个字符串是否同一个对象。看下面例子：

```
# String.eql?, tests two strings for identical content.
# It returns the same result as ==
# String.equal?, tests whether two strings are the same object
s1 = 'Jonathan'
s2 = 'Jonathan'
s3 = s1
if s1 == s2
  puts 'Both Strings have identical content'
else
  puts 'Both Strings do not have identical content'
end
if s1.eql?(s2)
  puts 'Both Strings have identical content'
else
  puts 'Both Strings do not have identical content'
end
if s1.equal?(s2)
  puts 'Two Strings are identical objects'
else
  puts 'Two Strings are not identical objects'
end
if s1.equal?(s3)
  puts 'Two Strings are identical objects'
else
  puts 'Two Strings are not identical objects'
end
```

使用%w

有时候定义一个字符串或字符数组，需要给每一个元素加上引号和逗号，比较麻烦。`ruby` 提供了一种简洁的方法，使用%w。

```
names1 = [ 'ann', 'richard', 'william', 'susan', 'pat' ]
puts names1[0] # ann
puts names1[3] # susan
# this is the same:
names2 = %w{ ann richard william susan pat }
puts names2[0] # ann
puts names2[3] # susan
```

练习：

给你一个字符串，把字符串中的单词反转。

首先用 `String.split` 方法得到包含每个单词的数组，然后用数组的 `reverse` 方法给单词重新排序，最后用 `join` 方法把单词重新组成一个字符串。

```
words = 'Learning Ruby - Your one stop guide'  
puts words.split(" ").reverse.join(" ")  
# guide stop one Your - Ruby Learning
```

所有有关字符串的文档，见[这儿](#)。

第十四章 控制语句

在 ruby 中，除了 nil 和 false 是假外，其他所有的都是真。nil 也是一个真实的对象，你可以调用 nil 的方法，也可以为 nil 定义新的方法。

下面讲一下 ruby 的控制语句。

ruby 规范说明，if 和 while 语句不需要括起来。

下面的例子用的是 if else end 语句。

```
# In Ruby, nil and false evaluate to false,
# everything else (including true, 0) means true
# nil is an actual object
# if else end
var = 5
if var > 4
  puts "Variable is greater than 4"
  puts "I can have multiple statements here"
  if var == 5
    puts "Nested if else possible"
  else
    puts "Too cool"
  end
else
  puts "Variable is not greater than 5"
  puts "I can have multiple statements here"
end
```

下面的例子用到了 elsif:

```
# elsif example

# Original example
puts "Hello, what's your name?"
STDOUT.flush
name = gets.chomp
puts "Hello, " + name + "!"

if name == 'Satish'
  puts "What a nice name!!"
else
  if name == 'Sunil'
    puts "Another nice name!"
  end
end
```

```

# Modified example with elsif
puts "Hello, what's your name?"
STDOUT.flush
name = gets.chomp
puts 'Hello, ' + name + '!'

if name == 'Satish'
  puts 'What a nice name!!'
elsif name == 'Sunil'
  puts 'Another nice name!'
end

# Further modified
puts "Hello, what's your name?"
STDOUT.flush
name = gets.chomp
puts 'Hello, ' + name + '!'

# || is the logical or operator
if name == 'Satish' || name == 'Sunil'
  puts 'What a nice name!!'
end

```

在条件判断语句中，可以使用一些常用的关系运算符，比如，`==`, `!=`, `>=`, `<=`, `>`, `<`。
 ruby 还有一种条件判断语句，`unless end`。除非 `unless` 后面跟的条件为真，才会运行 `unless end` 间的程序，否则跳过。

```

unless ARGV.length == 2
  puts "Usage: program.rb 23 45"
  exit
end

```

这段程序的意思是除非数组 `ARGV` 的元素个数为 2 个（也就是程序的 2 个参数都给定），才会运行程序块中的语句。
`exit` 是 ruby 内核内置方法，意思是退出当前应用程序，并返回给操作系统一个值。
 循环语句的使用像下面这样：

```

# Loops
var = 0
while var < 10
  puts var.to_s
  var += 1
end

```

作业:

1. 写一个程序，接受用户输入的年份，输出是闰年还是平年。
2. 写一个方法 `leap_year`，接受用户输入的年份，输出是闰年还是平年，并且输出该年共有多少分钟。

分支选择语句 (case)

分支选择语句类似于多个 `if` 语句。它会在一系列条件语句中找到第一个为真的语句。

比如闰年年份要求能被 4 或 400 整除，但是不能被 100 整除。

分支选择语句也会返回运行过程中，最后一条语句的值。

注意是运行过程中最后一条语句，如果这条语句根本没执行，不会返回它的值。

```
year = 2000
leap = case
  when year % 400 == 0: true
  when year % 100 == 0: false
  else year % 4 == 0
end
puts leap
# output is: true
```

第十五章 数组

数组包含了一系列按顺序排列起来的元素。元素可以是同一种类型，也可以是不同类型。定义数组的时候使用方括号([])把元素括起来。仔细看下面的例子：

```
# Arrays

# Empty array
var1 = []
# Array index starts from 0
puts var1[0]

# an array holding a single number
var2 = [5]
puts var2[0]

# an array holding two strings
var3 = ['Hello', 'Goodbye']
puts var3[0]
puts var3[1]

flavour = 'mango'
# an array whose elements are pointing
# to three objects - a float, a string and an array
var4 = [80.5, flavour, [true, false]]
puts var4[2]

# a trailing comma is ignored
name = ['Satish', 'Talim', 'Ruby', 'Java',]
puts name[0]
puts name[1]
puts name[2]
puts name[3]
# the next one outputs nil
# nil is Ruby's way of saying nothing
puts name[4]
# we can add more elements too
name[4] = 'Pune'
puts name[4]
# we can add anything!
name[5] = 4.33
puts name[5]
# we can add an array to an array
name[6] = [1, 2, 3]
```

```

puts name[6]

# some methods on arrays
newarr = [45, 23, 1, 90]
puts newarr.sort
puts newarr.length
puts newarr.first
puts newarr.last

# method each (iterator) - extracts each element into lang
# do end is a block of code
# we shall talk about blocks soon here -
# http://rubylearning.com/satishtalim/ruby\_blocks\_and\_procs.html
# variable lang refers to each item in the array as it goes through the
oop
languages = ['Pune', 'Mumbai', 'Bangalore']

languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Don\'t you?'
end

# delete an entry in the middle and shift the remaining entries
languages.delete('Mumbai')
languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Don\'t you?'
end

```

each 方法的作用是遍历数组中的每一个对象，并且可以处理这些对象。

从上面的例子看到，遍历数组对象并不需要根据数组长度做循环。

但是要记住下面几点：

1. `|lang|` 之间的变量名可以是任意的，代表的是数组中的每一个对象。
2. 对每一个数组元素，`do end` 之间的程序块都会执行。以后我们会详细的讲解程序块。

下面的例子中，方法返回了一个数组，非常有意思：

```

# if you give return multiple parameters,
# the method returns them in an array
# The times method of the Integer class iterates block num times,
# passing in values from zero to num-1

```



```
def mtdarry
  10.times do |num|
    square = num * num
    return num, square if num > 5
  end
end

# using parallel assignment to collect the return value
num, square = mtdarry
puts num
puts square
```

输出结果是：

```
6
36
```

整数的 `times` 方法会迭代整数次，并且初始值是 0。上面的例子中，`num` 会从 0 到 9 迭代，但是由于有一个 `if` 条件存在，所以当 `num=6` 时，跳出了循环。

方法返回多个参数时，参数被放在数组中返回，可以定义多个变量来接收数组中的元素。

ruby 支持像 `awk` 一样的关联数组吗？是的，`Hashes` 也是可以用在 `ruby` 中的。
更多的数组的文档，请看[这儿](#)。

作业：

1. 写一个程序，计算数组[1, 2, 3, 4, 5]中所有元素的和。
2. 写一个程序，判断数组[12, 23, 456, 123, 4579]中每一个元素是偶数还是奇数。

总结三

1. `method_missing` 提供了一个友好的办法, 处理找不到方法这种情况, 代替了返回一个异常。
2. 查找文档能得到详细的有关 `string` 的方法。
3. 双引号括起来的字符串处理过程要复杂一些。它会处理转义字符和 `#{expression}` 中的表达式。
4. 记住那些能列出类和对象的所有方法的方法。
5. `==` 或者 `eql?` 比较的是两个字符串的内容, `equal?` 比较两个字符串是否同一个对象。
6. `%w` 可以用来简化数组的初始化。
7. 熟悉常用控制语句: `if else end`, `while`, `if elsif end`
8. 另一个条件语句: `unless end`
9. 多分支选择语句: `case when else end`。
10. 数组是按顺序排列的对象, 数组元素可以是同一类型, 也可以是不同类型。定义数组使用方括号。
11. 数组的索引都是整数, 并且以 0 开始。
12. 数组定义时, 如果末尾有逗号, 忽略它。
13. 如果访问的数组元素越界, 会返回 `nil`。
14. 可以动态的给数组添加元素。
15. 查找相关文档获得更多数组信息。
16. `each` 方法能遍历数组中的所有元素, 并且可以操作这些元素。
17. 包含在 `"|"` 中的变量名称是任意的, 代表的是数组中的每个元素。
18. 遍历数组中的每一个元素时, 都会运行 `do end` 程序块。

It is to be noted that every time a string literal is used in an assignment or as a parameter, a new String object is created.

原来的总结中有上面一句话, 意思是当定义一个字符串时, 就生成了一个字符串对象。如果按字面翻译, 这句话容易和按值传递还是引用传递冲突, 所以我把它放在这个注释里, 可以先不理解这句话。

第十六章 范围

范围这个概念用在一系列元素（序列）的情况下。这一系列元素有一个起点，有一个终点，中间是连续的，有规律的元素。

在 `ruby` 中，可以使用 `'..'` 和 `'...'` 生成这样的范围对象。用 `'..'` 生成的序列包含终点，而 `'...'` 生成的序列不包含终点。

范围可以转换成数组，比如，`(1..10).to_a -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` 。

范围也有一系列方法用来迭代和对范围元素进行操作。

```
=begin  
Sequences have a start point, an end point, and a way to  
produce successive values in the sequence  
In Ruby, sequences are created using the ".." and "..."  
range operators.  
The two dot form creates an inclusive range.  
The three-dot form creates a range that excludes the specified  
high value  
The sequence 1..100000 is held as a Range object  
=end  
digits = -1..9  
puts digits.include?(5)           # true  
puts digits.min                   # -1  
puts digits.max                   # 9  
puts digits.reject { |i| i < 5 }  # [5, 6, 7, 8, 9]
```

范围对象使用最多的就是判断某个值是否在某个范围内，比如，

```
(1..10) === 5      -> true  
(1..10) === 15     -> false  
(1..10) === 3.14159 -> true  
( 'a'..'j' ) === 'c' -> true  
( 'a'..'j' ) === 'z' -> false
```

在这儿使用的是 `'==='` 操作符。

作业：

给定一个字符串，`s = 'key=value'`，从这个字符串中分离出两个子字符串，一个包含 `'key'`，另一个包含 `'value'`。

第十七章 Blocks 和 Procs

代码块(Block)是 ruby 特有的一个特性, 它包含在大括号或者 do-end 语句之间, 并且和方法关联。

Block 的定义跟在方法名后面, 如果方法带参数, 那么 Block 的位置还要后移, 要跟在最后一个参数后面。

Block 包含了一组程序语句, 在定义时这些语句并不会执行, 只有在方法中才能触发这些语句的执行。

当 Block 中包含单行语句时, 使用大括号; 如果包含多行语句, 就要使用 do-end。大括号的优先级比 do-end 要高。

那么什么时候才会执行这些语句呢? 在和 Block 关联的方法内, 调用 yield 关键字会触发 Block 的执行, 并且调用几次就会触发几次。

Block 能把语句中的对象, 变量, 上下文传给方法, 所以不用担心方法无法识别 Block 中的对象或者变量。

```
greet {puts 'Hello'}
```

上面是一个 Block 和方法关联的例子。{puts 'Hello'}这个 Block 和方法 greet 进行了关联。如果方法带参数, 那么关联像下面这样子:

```
verbose_greet("PuneRuby") {puts 'Hello'}
```

再看下面一个完整的例子:

```
=begin  
  Ruby Code blocks are chunks of code between braces or  
  between do- end that you can associate with method invocations  
=end  
def call_block  
  puts 'Start of method'  
  # you can call the block using the yield keyword  
  yield  
  yield  
  puts 'End of method'  
end  
  # Code blocks may appear only in the source adjacent to a method call  
  call_block {puts 'In the block'}
```

输出结果为:

```
Start of method  
In the block  
In the block  
End of method
```

call_block 这个方法和{puts 'In the block'}进行了关联。在执行 call_block 的过程中，调用了 yield。那么正常的方法执行过程会挂起，转而执行 Block 中的语句，当这些语句执行完成后，才会继续方法中后续语句的执行。

yield 是可以传参数给 Block 的，Block 会接受这些参数，进行处理。接受的方式是使用"||"。看下面例子：

```
# You can provide parameters to the call to yield:
# these will be passed to the block
def call_block
  yield('hello', 99)
end
call_block { |str, num| puts str + ' ' + num.to_s}
```

输出结果：

```
hello 99
```

yield 传了两个参数 'hello'和 99 给 Block，输出结果中打印出了这两个参数。

Block 的返回值也是最后一行的值，yield 会返回这个值，并且在方法中可以使用这个返回值。

Block 不是对象，但是可以通过 ruby 内核的内置方法 lambda 把它转化成一个 proc 对象。lambda 方法的参数是必须的，比如下面这样：

```
prc = lambda {"hello"}
```

Proc 对象绑定了 Block 里的代码块，并且可以通过内置方法 call 来激活这些语句的执行。

```
# Blocks are not objects
# they can be converted into objects of class Proc by calling lambda method
prc = lambda {puts 'Hello'}
# method call invokes the block
prc.call

# another example
toast = lambda do
  puts 'Cheers'
end
toast.call
```

输出结果为：

```
Hello
Cheers
```

Proc 和方法类似，但是 Proc 还有它特殊的地方，你可以把 Proc 作为一个参数传给方法，或者从方法返回一个 Proc 类型的对象。

下面的例子就演示了这些用法：

```
=begin  
  You cannot pass methods into other methods (but you can pass procs into  
methods),  
  and methods cannot return other methods (but they can return procs)  
=end  
  
def some_mtd some_proc  
  puts 'Start of mtd'  
  some_proc.call  
  puts 'End of mtd'  
end  
  
say = lambda do  
  puts 'Hello'  
end  
  
some_mtd say
```

输出结果：

```
Start of mtd  
Hello  
End of mtd
```

下面是一个给 lambda 传参数的例子：

```
aBlock = lambda { |x| puts x }  
aBlock.call 'Hello World!'  
# output is: Hello World!
```

第十八章 随机数

Ruby 中可以使用 `rand` 方法获得随机数。默认情况下，`rand` 方法得到一个大于或等于 0.0，小于 1.0 的浮点数。如果给 `rand` 带一个整数的参数，比如 `rand(5)`，就会得到一个大于或等于 0 小于 5 的随机整数。

```
=begin
  If you call rand, you'll get a float greater than or equal to 0.0
  and less than 1.0. If you give it an integer parameter (by calling rand(5) ),
  you will get an integer value greater than or equal to 0 and less than 5
=end

# from each of the three lists and prints out the result
word_list_one = ['24/7', 'multi-Tier', '30,000 foot', 'B-to-B', 'win-win', '
front-end', 'web-based', 'pervasive', 'smart', 'six-sigma', 'critical-path',
'dynamic']
word_list_two = ['empowered', 'sticky', 'value-added', 'oriented', 'centric', '
distributed', 'clustered', 'branded', 'outside-the-box', 'positioned',
'networked', 'focused', 'leveraged', 'aligned', 'targeted', 'shared',
'cooperative', 'accelerated']
word_list_three = ['process', 'tipping-point', 'solution', 'architecture', 'core
competency', 'strategy', 'mindshare', 'portal', 'space', 'vision', 'paradigm',
'mission']

one_len = word_list_one.length
two_len = word_list_two.length
three_len = word_list_three.length

rand1 = rand(one_len)
rand2 = rand(two_len)
rand3 = rand(three_len)

phrase = word_list_one[rand1] + " " + word_list_two[rand2] + " " +
word_list_three[rand3]

puts phrase
```

上面的程序中，有三个由单词组成的 list，从每个 list 中随机抽取一个并输出。

作业：

作业来自 [Chris Pine's Book](#)

1. 写一个聋子奶奶的程序。当你同这个聋子奶奶说话(也就是输入一些单词)时，因为她听不到，所以只会说：HUH?! SPEAK UP, SONNY! 除非你大声跟她说(也就是全输入大写)，她才会说：NO, NOT SINCE 1938! 为了程序的可信度，输入的年份要从 1930 到 1950 中随机抽取。如果想结束程序，需要跟奶奶说 BYE。

2.修改上面的程序，当你同奶奶说 **BYE** 时，她假装听不见。除非你同她大声说三次 **BYE**。
记住输入的三个 **BYE** 必须要在同一行上，不可以分三行输入。

第十九章 文件读写

下面的程序是文件读写的例子：

```
# Open and read from a text file
# Note that since a block is given, file will
# automatically be closed when the block terminates
File.open('constructs.rb', 'r') do |f1|
  while line = f1.gets
    puts line
  end
end

# Create a new file and write to it
File.open('test.rb', 'w') do |f2|
  # use "\n" for two lines of text
  f2.puts "Created by Satish\nThank God!"
end
```

File.open 方法有下面几种读写模式：

1. 'r', read-only, 只读模式，从文件头开始；
2. 'r+', read/write, 读写模式，从文件头开始；
3. 'w', write-only, 只写模式，覆盖原来文件或创建一个新文件。

查看[文档](#)获得详细信息。

File.open 后面跟的 Block 是对文件的操作，Block 是可选的。如果不跟 Block，File.open 会打开一个新文件。如果跟了 Block，文件会被作为参数传给 Block 进行处理，并且 Block 结束后，文件会被自动关闭。

注意：文件操作结束后，要 Close 文件，以防文件读写错误。

File 实现了一个 readlines 方法，把文件读到一个数组中，一行是一个数组元素。

方法 open 和 readlines 都属于 IO 类，File 是它的子类。我们现在还没有讲到类、对象、继承的概念。现在只要知道 open 和 readlines 方法都是 File 类从 IO 类继承来的就可以了。

遍历目录

Find 模块支持对目录的遍历，只要给 find 方法传一个目录的 URL 参数，它会列出该目录下的所有文件及子目录。

```
require 'find'
Find.find('.') do |f|
  type = case
    when File.file?(f): "F"
    when File.directory?(f): "D"
    else "?"
  end
  puts "#{type}: #{f}"
end
```

我们后面会谈[到require的使用](#)。

文件访问

在 ruby 中对文件的访问非常容易，下面是一个文本文件，文件名叫 hellousa.rb，内容如下：

```
puts 'Hello USA'
```

我们的目的是显示从 USA 开始的内容，看下面程序：

```
# We now need to display the contents of the file from the word USA
f = File.new("hellousa.rb")

# SEEK_CUR - Seeks to first integer number parameter plus current
position
# SEEK_END - Seeks to first integer number parameter plus end of stream
# (you probably want a negative value for first integer number
parameter)
# SEEK_SET - Seeks to the absolute location given by first integer number
parameter
# :: is the scope operator - more on this later
f.seek(12, IO::SEEK_SET)
print f.readline
```

seek 也是 IO 类的方法，它会按着第二个参数的方法，从第一个参数给定的位置开始查找。

IO::SEEK_CUR - 从当前位置加上第一个参数的位置开始（相对位置）。

IO::SEEK_END - 从文件尾开始反向读取，位置是第一个参数的绝对值。

IO::SEEK_SET - 从第一个参数给定的位置开始（绝对位置）。

更多有关作用域操作符::的内容，请看[这儿](#)。

IO::SEEK_CUR ， IO::SEEK_END ， IO::SEEK_SET 这三个参数是相对于第一个参数的。

如果使用 IO::SEEK_CUR，那就说明第一个参数给的位置是相对位置，真正的 seek 起始点应该是第一个参数值+当前位置。如果使用 IO::SEEK_SET，那就说明第一个参数给的位置是绝对位置，真正的 seek 起始点就是第一个参数值的位置。如果使用 IO::SEEK_END，说明要从文件尾开始方向读取，seek 的起始点就是第一个参数值的绝对值，为什么要说是绝对值，是因为在这种情况下，第一个参数要给定一个负整数。还有一个算法是用这个负整数+文件的长度，结果值就是 seek 的起始点位置。

对象序列化

Ruby 也支持像 java 中的序列化的功能。使用 Marshal.dump 把对象或组件存储起来，需要的时候可以使用 Marshal.load 把它们读取出来。

Ruby使用marshal存储session数据。后面会有[对象序列化](#)的详细介绍。

总结四

1. 使用'..'和'...'能生成范围对象，并且可以对范围内的元素进行操作。
2. 两个点生成的范围对象包括终点，三个点生成的范围对象不包括终点。
3. 1..100000 就是一个范围对象的例子。
4. .to_a 方法能把范围对象转换成数组。
5. 范围对象使用最多的就是判断某个值是否在某个范围内，使用'==='操作符。
6. 范围的元素并不局限于数值，也可以是 ruby 对象。
7. **IMPORTANT:** Block 是括在大括号或者 do-end 中间的一组程序语句，和方法关联。
8. Block 的定义跟在方法名后面，如果方法带参数，那么 Block 的位置还要后移，要跟在最后一个参数后面。Block 包含了一组程序语句，在定义时这些语句并不会执行，只有在方法中才能触发这些语句的执行。
9. 当 Block 中包含单行语句时，使用大括号；如果包含多行语句，就要使用 do-end。大括号的优先级比 do-end 要高。
10. 在和 Block 关联的方法内，调用 yield 关键字会触发 Block 的执行，并且调用几次就会触发几次。
11. yield 可以传参数给 Block，Block 会接受这些参数并进行处理。接受的方式是使用"|"。
12. Block 不是对象，但是可以通过 ruby 内核的内置方法 lambda 把它转化成一个 proc 对象。
13. 记住，不能把方法作为参数传给另一个方法，也不能把方法作为返回值。但是可以把 Proc 作为一个参数传给方法，或者从方法返回一个 Proc 类型的对象。
14. Ruby 中可以使用 rand 方法获得随机数。
15. 默认情况下，rand 方法得到一个大于或等于 0.0，小于 1.0 的浮点数。如果给 rand 带一个整数的参数，比如 rand(5)，就会得到一个大于或等于 0 小于 5 的随机整数。
16. File.open 方法的几种读写模式：1.'r', read-only，只读模式，从文件头开始；2.'r+', read/write，读写模式，从文件头开始；3.'w', write-only，只写模式，覆盖原来文件或创建一个新文件。
17. File.open 后面跟的 Block 是对文件的操作，Block 是可选的。如果不跟 Block，File.open 会打开一个新文件。如果跟了 Block，文件会被作为参数传给 Block 进行处理，并且 Block 结束后，文件会被自动关闭。
18. 注意：文件操作结束后，要 Close 文件，以防文件读写错误。
19. seek 也是 IO 类的方法，它会按着第二个参数的方法，从第一个参数给定的位置开始查找。IO::SEEK_CUR - 从当前位置加上第一个参数的位置开始（第一个参数是相对位置）。IO::SEEK_END - 从文件尾开始反向读取，位置是第一个参数的绝对值。IO::SEEK_SET - 从第一个参数给定的位置开始（绝对位置）。

第二十章 自定义类

在面向过程的程序语言里，比如 C，注重的是程序逻辑，而不是数据本身。

那么在面向对象的语言里，对象是关键，也是重点。你可以从对象得到信息，可以让对象执行特定的任务；可以让它执行某些运算并把结果返回给你；还可以让多个对象协同工作。

在设计类的时候，首先要想到，基于这个类的对象会有哪些属性，以及这个对象能做什么。对象所具有的属性也可以叫实例变量，它代表的是对象的状态。每个类可以有好多实例对象，而且每个对象所具有的状态可以是不同的。

而对象能执行的动作，我们叫做方法，方法能改变对象的状态。

下面是 Hal Fulton 对对象的理解：

An object is an entity that serves as a container for data and also controls access to the data.

Associated with an object is a set of attributes, which are essentially no more than variables belonging to that object.

Also associated with an object is a set of functions that provide an interface to the functionality of the object, called methods. - Hal Fulton

类是由一系列属性和方法组成的。

类是对象的模板。比如说，你可能会用 `Button` 类构造了好多的按钮对象，但是每一个按钮对象可能会有不同的颜色，尺寸，形状，标签等等。对象是类的实例。

仔细体会下面的内容。

类 `Class` 是所有类的基类，任何类都是 `Class` 的实例。

当定义一个新类的时候，一个 `Class` 类的对象被创建，并被分配给一个常量（也就是类名）。当调用 `.new` 创建一个类的对象时，会首先调用 `allocate` 方法给对象分配内存，然后才是调用 `initialize` 方法初始化对象。

对象的构造和初始化是不同的，但它们都可以被覆盖。初始化要调用 `initialize` 方法；构造是在调用 `new` 方法时完成的，所以说 `initialize` 不是构造器函数。

有关更多类的继承关系，请看[Class Hierarchy](#)。

下面我们开始写本教程中的第一个自定义类：

```
# define class Dog
class Dog
  def initialize(breed, name)
    # Instance variables
    @breed = breed
    @name = name
  end

  def bark
    puts 'Ruff! Ruff!'
  end
end
```

```
def display
  puts "I am of #{@breed} breed and my name is #{@name}"
end

# make an object
# method new used to create object
d = Dog.new('Labrador', 'Benzy')
puts d.methods.sort
puts "The id of obj is #{d.object_id}."

if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, the object doesn't understand the 'talk' message."
end

d.bark
d.display

d1 = d
d1.display

d = nil
d.display
```

我们定义了一个 Dog 类。然后我们使用 new 方法创建了一个 Dog 类的对象 d。d 只是这个对象的一个引用（类似于 C 语言的指针），并不是对象本身。然后使用 '.' 操作符调用对象的方法，比如，

```
d.bark
```

在一个 Rails 应用程序中，比如说你正在写一个 Customer 的 model，如果你想给 Customer 添加电话号码或者网址，那么就需要使用 '.' 操作符调用相应的方法。调用方法的过程也就是传递给对象一个消息，让它执行某个动作。

哪怕是刚创建的一个新对象，它也已经继承了一些方法，使用 methods 方法可以查看这些方法。比如下面，

```
puts d.methods.sort
```

sort 方法用来排序，方便查看。

运行上面的语句，你会得到好多方法的列表，其中 object_id 和 respond_to? 方法是比较重要的两个。

在 ruby 中，每一个对象都会有一个唯一的 id，使用 object_id 方法可以查看这个 id。

前面讲过，调用方法就等于给对象传递消息，但是有时候，你事先并不知道传递的这个消息是否是合法消息。respond_to?方法就是用来判断是否是合法消息的。比如，

```
if d.respond_to?("talk")
  d.talk
else
  puts "Sorry, the object doesn't understand the 'talk' message."
end
```

在事先并不知道 talk 是否是合法消息（合法方法）的时候，使用一个判断语句，如果 talk 存在，就调用它；如果不存在，就给出一个错误信息。
再接下来的语句：

```
d1 = d
d1.display
```

创建了另一个对象的引用 d1，和 d 指向的是同一个对象。

可以使用 Object.class 方法来判断当前的对象来自哪个类，比如，

```
d = Dog.new('Alsatian', 'Lassie')
puts d.class.to_s
```

输出结果为：

```
Dog
```

instance_of?方法能够判断对象是否是某个类的实例，比如，

```
num = 10
puts (num.instance_of? Fixnum) # output true
```

返回结果 true，说明 num 是整数对象。

Literal Constructors

对于某些特殊的类，可以不使用 new 方法就可以构造该类的对象。看下面的列表：

Class	Literal constructor	Example(s)
String	Quotation marks	"new string" or 'new string'
Symbol	Leading colon	:symbol or : "symbol with spaces"
Array	Square brackets	[1,2,3,4,5]
Hash	Curly braces	{"New York" => "NY", "Oregon" => "OR"}
Range	Two or three dots	0...10 or 0..9
Regexp	Forward slashes	/([a-z]+)/

Literal Constructors, 我一直没有找到一个合适的翻译, 意思就是说给对象指定初始值的时候, 可以省略 new 方法。

垃圾收集

例子中的语句,

```
d = nil
```

把 d 引用 nil。 nil 是什么都没有的意思
如果我们再把 d1 引用 nil,

```
d1 = nil
```

那么我们创建的那个 dog 对象, 就没有被任何变量引用, 它就可以被作为垃圾回收了。
Ruby 中的对象堆分配了最少 8 兆的内存空间。

ruby 的垃圾回收属于“标记-清除”型。“标记”的过程会检查对象是否仍在使用中。如果对象仍然被某个变量引用, 那么它会被打上一个“保留”的标志。如果对象没有被任何变量引用, 它不会被打上“保留”标志。清除过程只会对没有“保留”标志的对象进行垃圾回收。

对象创建后, 被变量引用, 说明这个对象是在使用中。只有一个对象没有被任何变量引用, 才会被垃圾回收。那么对象和变量的引用关系的脱离, 有可能是因为变量离开了自己的作用域, 变量消失, 当然这种引用关系也就自动脱离; 也有可能是因为程序员书写了某些语句, 手动把引用关系扯断, 比如上面的例子, 我们把 d 和 d1 都重新引用了 nil, 所以 dog 对象就失去了任何引用而会被垃圾回收。

Ruby 的垃圾回收比较保守, 所以不能保证废弃了的对象被立刻回收。

如果把对象放到了数组中, 只要数组有效, 那么数组中的对象都会被标记为“保留”。

对于全局变量和常量引用的对象, 总是被标记为“保留”。

类方法

类方法概念的提出是为了使得能够给类传递消息, 执行某个动作。是的, 目的就是调用类的某个方法, 而不是对象的方法。new 方法就是个例子, Dog.new 能够创建一个 Dog 类的对象, new 方法是作用给 Dog 类的, 而不是一个 Dog 的对象 (给对象一个 new 方法没有意义)。

类本身总是可以保留一些特殊的東西, 而不扩展给对象, 可以是方法, 也可以是属性, 这些保留的属性和方法, 类的任何对象都是没有的。

下面的图来自书《Ruby For Rails》,

Notation	Method referred to
Ticket#price	The instance method price in the class Ticket
Ticket.most_expensive	The class method most_expensive, in the class Ticket
Ticket::most_expensive	Another way to refer to the class method most_expensive

`File#chmod` 表示是一个对象实例方法，使用了“#”；`File.chmod` 表示是一个类方法，使用“.”。这些符号不是 ruby 的语法，只是一种约定。

有关如何写一个类方法，请参考[这儿](#)。

第二十一章 引入文件

开始写 ruby 程序时，我们把所有代码都写到一个文件里。

但是对一个有经验的开发者来说，这并不是个好习惯。随着经验的增长，会把代码写到多个文件中，并且按着一定的逻辑分门别类。这样做的目的是提高代码的可读性，重用性，以及可维护性。

在 ruby 中，使用 `require` 和 `load` 方法，把放到多个文件中的代码组合起来，就好像来自一个文件一样。

`load` 用法像下面这样，

```
load 'filename.rb'
```

每次运行 `load` 方法的时候，被引入文件中的代码都会被引入一次。

`require` 的用法如下，

```
require 'filename'
```

运行一次和多次上面的代码，效果是一样。也就是说只要运行了一次 `require` 方法，被引入的文件代码就会嵌入进来，即使你再次 `require`，这部分内容也不会改变。

`require` 的用法会更广泛一些，比如引入 ruby 的标准库或者其他第三方的库。这些库提供了我们需要的 api，能帮助我们更加快速的进行开发。

要注意一点，使用 `require` 的时候，只要指定库的文件名就可以了，不需要提供扩展名。为什么呢？因为并不是所有的库都是以 `.rb` 的形式存储的。比如说用 `c` 写的库有可能会被编译成 `.so` 或 `.dll` 文件。

`load` 是动态引入，`require` 是静态引入，这是它们的区别。

使用 `load` 还是 `require` 要遵循一个标准，如果是第三方的标准库文件，它的代码不会轻易改变，所以用 `require` 好些。

如果引入的是一段正在开发中的代码，`load` 就比较合适，因为可以随时把代码的更新加载进来。

在 Rails 中，`load` 要比 `require` 更受欢迎。比如，在开发模式下，你总是希望能即时看到代码修改后的效果。`load` 就能做到这样的效果，`require` 却不行，即使你执行多次 `require`，加载的都是第一次引入的内容。

看下面的例子，我们先写一个自定义的类，然后引用它：

```
class Motorcycle
  def initialize(make, color)
    # Instance variables
    @make = make
    @color = color
  end
end
```

```
    end
  def startEngine
    if (@engineState)
      puts 'Engine Running'
    else
      @engineState = true
      puts 'Engine Idle'
    end
  end
end
end
```

我们再写一个文件来引用它，

```
#p031motorcycletest.rb
require 'p030motorcycle'
m = Motorcycle.new('Yamaha', 'red')
m.startEngine
```

第二十二章 Ruby 类的开放性

在 Ruby 中，Class 是开放的。意思是说，你总是可以修改现有的类（添加或删除内容），不管是自定义的类还是 ruby 内置的类。并且，这种修改甚至可以不在该类的定义文件中。

看下面的例子，使用到了上一章定义的类 `motorcycle`：

```
require 'motorcycle'
m = Motorcycle.new('Yamaha', 'red')
m.startEngine

class Motorcycle
  def dispAttr
    puts 'Color of Motorcycle is ' + @color
    puts 'Make of Motorcycle is ' + @make
  end
end
m.dispAttr
m.startEngine
puts self.class
puts self
```

上面的例子创建了一个 `MotorCycle` 类的对象，调用了 `startEngine` 方法。然后为 `MotorCycle` 类添加了一个方法。

注意，方法不是添加在 `MotorCycle` 类的定义文件中。

这种为类添加的方法，一般都是临时方法（不提倡这种做法）。

如果想添加长久性或者共用性的方法，需要加到类的定义文件中。

`self` 方法返回当前所处的对象；`self.class` 方法返回当前所处对象的类。在这个例子中，分别是 `main` 和 `object`，`main` 对象是 `object` 类的实例。

再看下面的例子，使用到了前面章节定义的类 `Dog`，并且给这个类添加了一个方法，

```
# define class Dog
class Dog
  def big_bark
    puts 'Woof! Woof!'
  end
end
# make an object
d = Dog.new('Labrador', 'Benzy')
d.bark
d.big_bark
d.display
```

再看下面的例子，给 `String` 类添加一个方法，

```
class String
  def writesize
    puts self.size
  end
end
size_writer = "Tell me my size!"
size_writer.writesize
```

我们可以猜想到结果就是输出了字符串的长度。

如果你想给现有的类添加一个新方法，可以打开类文件，把新代码加进去。但是，需要考虑的是，这个方法最好是通用方法，并且不能够跟其他方法冲突。

如果这不是个通用方法，并且不想修改原有类文件，可以把它加到一个子类中。

新建子类的好处就是，不需要修改原文件。如果需要的话，还可以添加新方法或者覆盖父类方法，并且这样做对父类没有任何影响。

第二十三章 继承

继承指的是类之间的一种关系。就好像说，猫是哺乳动物，哺乳动物属于动物一样。子类继承了父类的所有特性，并且子类还可以有自己的特性。比如，哺乳动物可以呼吸，那么所有的猫科动物肯定都会呼吸，并且还可以“miao miao”的叫。Ruby 只支持单继承，也就是说子类只能有一个父类。看下面的例子：

```
class Mammal
  def breathe
    puts "inhale and exhale"
  end
end

class Cat < Mammal
  def speak
    puts "Meow"
  end
end

rani = Cat.new
rani.breathe
rani.speak
```

在上面的例子中，我们并没有给 Cat 类定义 breathe 方法，但是它从 Mammal 类继承了 breathe 方法。（在 OO 思想中，super-class, parent class 指的是父类; subclass child-class 指的是子类）Cat 类自己有一个 speak 方法，并且继承了 breathe 方法，所以所有的 Cat 即可以 speak，也可以 breather。

有时候，父类的特性并不想被子类继承。比如，鸟类有飞的特性，但是企鹅属于鸟类，但是不会飞。看下面的例子，企鹅类覆盖了鸟类飞的特性，

```
class Bird
  def preen
    puts "I am cleaning my feathers."
  end
  def fly
    puts "I am flying."
  end
end

class Penguin < Bird
  def fly
    puts "Sorry. I'd rather swim."
  end
end
```

```
end

p = Penguin.new
p.preen
p.fly
```

继承的一个好处是，当定义一个新类的时候，不需要把所有特性全部重新写一遍，只要把子类特有的特性加上。或者有时候子类和父类共有的特性有些差异，只要覆盖一下就可以了。继承有时候被叫做 **differential programming**，这也是 O-O 的一个特性。上面的两个程序都来自[Ruby User's Guide](#)。

总之，继承允许你通过改善，定制父类来创建一个子类。继承的符号是"<"。再看下面的例子：

```
class GF
  @m = 10
  puts @m.object_id
  def initialize
    puts 'In GF class'
  end
  def gfmethord
    puts 'GF method call'
  end
end

# class F sub-class of GF
class F < GF
  def initialize
    super
    puts 'In F class'
  end
end

# class S sub-class of F
class S < F
  def initialize
    super
    puts @m
    puts @m.object_id
    puts 'In S class'
  end
end

son = S.new
son.gfmethord
```

Ruby 类的继承是单继承，一个子类只能有一个父类，但是这个父类也可以是其他类的子类。

Ruby 有好多内置的类和模块。在任何程序中都可以使用它们，并且不需要 `require`。

比如类，`Array`, `Bignum`, `Class`, `Dir`, `Exception`, `File`, `Fixnum`, `Float`, `Integer`, `IO`, `Module`, `Numeric`, `Object`, `Range`, `String`, `Thread`, `Time`。。

比如模块，`Comparable`, `Enumerable`, `GC`, `Kernel`, `Math`。

`Object` 是所有类的父类，所以 `Object` 的方法可以被任何类继承和覆盖。

继承对 Rails 来说也是一个关键特性。

第二十四章 方法覆盖

方法覆盖是面向对象语言的一个特性，它允许子类重新实现父类的方法。子类的实现覆盖了父类的实现。

看下面例子，

```
class A
  def a
    puts 'In class A'
  end
end

class B < A
  def a
    puts 'In class B'
  end
end

b = B.new
b.a
```

子类 B 覆盖了来自父类 A 的方法 a。

super 的用法

几种不同的 super 的调用方法，

1. super。调用父类的同名方法，自动传参数过去。
2. super()。调用父类同名方法，不传任何参数过去。
3. super(a,b,c)。调用父类同名方法，传参数 a, b, c 过去。

看下面的例子，

```
class Bicycle
  attr_reader :gears, :wheels, :seats
  def initialize(gears = 1)
    @wheels = 2
    @seats = 1
    @gears = gears
  end
end

class Tandem < Bicycle
  def initialize(gears)
    super
    @seats = 2
  end
end
```



```
t = Tandem.new(2)
puts t.gears
puts t.wheels
puts t.seats
b = Bicycle.new
puts b.gears
puts b.wheels
puts b.seats
```

输出结果为，

```
2
2
2
1
2
1
```

我们后面会详细介绍[attr reader](#)的知识。

方法再定义

选自 David Black 的书《Ruby For Rails》。

在一个类里面，可以对方法定义多次。

```
class OR
  def mtd
    puts "First definition of method mtd"
  end
  def mtd
    puts "Second definition of method mtd"
  end
end
OR.new.mtd
```

输出结果，

```
Second definition of method mtd
```

说明调用的是第二个方法。

当定义方法多次时，越靠后定义的方法优先级越高。这是在同一类里的覆盖。

第二十五章 方法重载

有时候，你想定义两个方法，有相同的方法名，但是却有不同的参数类型或个数。这在 java 语言里叫重载。

但是在 Ruby 里，方法名是唯一的。要想达到重载的效果，可以通过判断参数的个数和类型来做到。

比如下面矩形的例子。定义一个矩形，可以通过定义左上角和右下角的坐标，或者左上角的坐标加上长，宽。我们要做的就是在一个方法 `initialize` 里，同时处理这两种情况。

比如下面矩形的例子。

定义一个矩形，可以通过定义左上角和右下角的坐标，或者左上角的坐标加上长，宽。

我们要做的就是在一个方法 `initialize` 里，同时处理这两种情况。

```
# The Rectangle constructor accepts arguments in either
# of the following forms:
#   Rectangle.new([x_top, y_left], length, width)
#   Rectangle.new([x_top, y_left], [x_bottom, y_right])
class Rectangle
  def initialize(*args)
    if args.size < 2 || args.size > 3
      # modify this to raise exception, later
      puts 'This method takes either 2 or 3 arguments'
    else
      if args.size == 2
        puts 'Two arguments'
      else
        puts 'Three arguments'
      end
    end
  end
end

Rectangle.new([10, 23], 4, 10)
Rectangle.new([10, 23], [14, 13])
```

上面的这个例子，对定义一个真正的矩形是不够的。但是它解释了如何实现重载的效果。

还要注意的是 `initialize` 方法接受了一个参数 `args`，可以得到传入的参数。

总结五

1. 对象是包含了一系列属性和行为的实体。对象的属性用变量表示；对象的功能（可以采取的行为或动作）用方法表示。
2. 实例变量（属性）代表的是对象的状态。
3. 方法用来改变对象的状态。
4. 类是由实例变量和方法组成的。
5. 类用来构造对象。类是对象的蓝本。
6. Ruby 中有超过 30 个的内置类。类的继承关系请见 class hierarchy。
7. 在 Ruby 中，不管是整数还是字符串，任何事物都是对象。并且这些对象内置了一系列方法。如果想调用方法，只要在对象后加一个"."，然后跟上方法名就可以了，比如，`object.method`。有时候，有些方法可以用在任何地方，比如 `puts` 和 `gets`，不用加对象名，直接用就可以了。这些方法已经内置在 ruby 的内核模块里，所有的对象都已经默认继承了这些方法，所以可以直接用。当运行一个 ruby 程序的时候，一个叫 `main` 的对象被创建，它属于 `object` 类，是程序的入口。
8. 表示整数的类是 `Fixnum` 和 `Bignum`，它们的区别是位长不同。浮点数用 `Float` 表示。
9. 类的定义方式为，`Name...End`。
10. 类在 ruby 中也是对象，是 `Class` 类的对象。
11. 类是对象，是 `Class` 类的对象；`Object` 是一个类。好好理解这句话。
12. 当定义一个新类的时候，一个 `Class` 类的对象被创建，并被分配给一个常量（也就是类名）。当调用 `.new` 创建一个类的对象时，会首先调用 `allocate` 方法给对象分配内存，然后才是调用 `initialize` 方法初始化对象。对象的构造和初始化是不同的，但它们都可以被覆盖。初始化要调用 `initialize` 方法；构造是在调用 `new` 方法时完成的，所有说 `initialize` 不是构造器函数。
13. 对象是在堆里创建的。
14. `d = Dog.new('Labrador', 'Benzy')`。d 只是对象的引用，并不是对象本身。
15. 哪怕是刚创建的一个新对象，它也已经继承了一些方法，使用 `methods` 方法可以查看这些方法。其中 `object_id` 和 `respond_to?` 方法是比较重要的两个。
16. 在 ruby 中，每一个对象都会有一个唯一的 id，使用 `object_id` 方法可以查看这个 id。
17. 使用 `respond_to?` 方法能够判断调用的方法是否存在。
18. `Object.class` 方法来判断当前的对象来自哪个类。
19. `instance_of?` 方法能够判断对象是否是某个类的实例。
20. `Literal Constructors` 的意思是，可以用某些特殊的写法来代替 `new` 方法创建新对象。
21. Ruby 中的对象堆分配了最少 8 兆的内存空间。ruby 的垃圾回收属于“标记-清除”型。“标记”的过程会检查对象是否仍在使用中。如果对象仍然被某个变量引用，那么它会被打上“保留”的标志。如果对象没有被任何变量引用，它不会被打上“保留”标志。清除过程只会对没有“保留”标志的对象进行垃圾回收。
22. 变量是对对象的引用，它不是对象本身，也没有类型。
23. `load` 方法能把文件引入多次。
24. `require` 比较常用，它只引用一次，即使多次调用。
25. `require 'filename'`，不带文件扩展名。
26. 在 Ruby 中，`Class` 是开放的。你总是可以修改现有的类（添加或删除内容），不管是自定义的类还是 ruby 内置的类。并且，这种修改甚至可以不在该类的定义文件中。
27. 继承的好处是，子类可以继承父类的属性和方法，不需要重新定义，只要加上自己特有的特性就可以了。

28. Ruby 的继承是单继承。
29. Object 是所有类的父类，它的方法可以被继承，也可以被覆盖。
30. 覆盖允许子类重新实现父类的方法，覆盖父类的实现。
31. 当你覆盖了某个方法，新的方法就会生效。
32. super 调用父类的同名方法，自动传参数过去。
33. super()调用父类同名方法，不传任何参数过去。
34. super(a,b,c)调用父类同名方法，传参数 a, b, c 过去。
35. Ruby 中，方法名是唯一的，重载需要写在一个方法中。

第二十六章 Symbol

第二十七章 访问控制