[**Ruby 使用手冊**](http://guides.ruby.tw/ruby/index.html)

# 甚么是 Ruby？(ok)

Ruby 是“用于快捷易用物件导向程式设计的直译式脚本语言”－－这是甚么意思呢？

*直译式脚本语言 (interpreted scripting language)：*

* 能够直接产生作业系统呼叫
* 强大的字串处理 (string operations) 及正则表示式 (regular expressions)
* 在开发时提供即时回馈

*快捷易用：*

* 不需要变数宣告 (variable declarations)
* 变数不需要型别 (typed)
* 语法简单一致
* 自动管理记忆体

*物件导向程式设计：*

* 任何东西都是物件
* 类别、方法、继承等
* 单件方法 (singleton methods)
* 模组提供“混入”(mixin) 功能
* 迭代器 (iterators) 及闭包 (closures)

*以及：*

* 多精确度整数 (multiple precision integers)
* 方便的例外处理 (exception processing)
* 动态载入 (dynamic loading)
* 执行绪支援 (threading support)

如果有些概念你并不清楚，没关系，继续看下去。Ruby 语言的精髓就是*快捷易用*。

# 起步走(ok)

首先，看看是不是已经安装了 Ruby (若没有安装，请参考下面的编注)。在 shell 提示符后（即 "%"，不要再输入 %了），输入

% ruby -v

（-v 会通知直译器显示 Ruby 的版本），然后按下 Enter 键。如果已经安装 Ruby，就出现以下讯息：

% ruby -v

ruby 1.8.3 (2005-09-21) [i586-linux] # 编注：目前通行的版本为 1.8.6 以上。

如果还没有安装 Ruby，你可以请管理员安装或自己安装，反正 Ruby 是自由软体，并无任何安装或使用限制。

现在就来体验 Ruby 吧。你可以利用 -e 选项，把 Ruby 程式直接放上命令列：

% ruby -e 'puts "hello world"'

hello world

一般来说，Ruby 程式可储存在档案内。

% echo "puts 'hello world'" > hello.rb

% ruby hello.rb

hello world

如果要编写比这更长的程式码，最好使用文字编辑器！

只要能放进命令列，有些异常复杂而实用的程式码也能利用小型的程式完成。以下例子将当前目录的标头档(附档名 .h)及所有 C 原始码(附档名 .c)中的 foo 替换为 bar，将原始档案以 ".bak" 结尾备份起来。

% ruby -i.bak -pe 'sub "foo", "bar"' \*.[ch]

这个程式的效果类似于 UNIX cat 指令（但速度比 cat 慢）：

% ruby -pe 0 file

### 安装 Ruby (编注)

目前通行的版本是 Ruby 1.8.6 和 Ruby 1.8.7，这里推荐您使用 1.8.7。

#### Windows 使用者

下载安装 [Ruby 1.8.7](http://rubyinstaller.org/)

#### Ubuntu Linux 使用者

用 Firefox 浏览 [Download Ruby Enterprise Edition](http://www.rubyenterpriseedition.com/download.html) 中的 Ubuntu Linux，然后点选下载套件并安装即可。

#### MacOS Snow Lepard 使用者

您的作业系统已经内建了 Ruby 1.8.7。

# 简单示范(ok)

来写个函式计算阶乘吧。n 阶乘的数学定义是：

n! = 1 （当 n==0）

= n \* (n-1)! （否则）

这可在 Ruby 中写成：

def fact(n)

  if n == 0

    1

  else

    n \* fact(n-1)

  end

end

你会发现 end 重复出现， Ruby 正是因此而称为类似“算法”(Algol) 的语言。（其实，Ruby 的语法更接近 Eiffel 语言。）你也会发现并没有 return 叙述， 因为 Ruby 函数会回传最后计算的步骤。Ruby 可以使用 return 叙述，但并不必要。

来试试我们的阶乘函式吧。再加入一行程式码就能运作这个程式：

# 计算出数字的阶乘

# 请把程式储存为 fact.rb

def fact(n)

  if n == 0

    1

  else

    n \* fact(n-1)

  end

end

puts fact(ARGV[0].to\_i)

上面的 ARGV 是一个阵列，其中包含了命令列引数 (command line arguments)，而 to\_i 将字元字串 (character string) 转换为整数。

% ruby fact.rb 1

1

% ruby fact.rb 5

120

能不能计算引数为 40 呢？这会让你的计算机溢位 (overflow)……

% ruby fact.rb 40

815915283247897734345611269596115894272000000000

确实能够运算， 其实只要电脑的记忆体足够，Ruby 能够处理任何整数。所以就算是 400 也能够计算：

% ruby fact.rb 400

64034522846623895262347970319503005850702583026002959458684

44594280239716918683143627847864746326467629435057503585681

08482981628835174352289619886468029979373416541508381624264

61942352307046244325015114448670890662773914918117331955996

44070954967134529047702032243491121079759328079510154537266

72516278778900093497637657103263503315339653498683868313393

52024373788157786791506311858702618270169819740062983025308

59129834616227230455833952075961150530223608681043329725519

48526744322324386699484224042325998055516106359423769613992

31917134063858996537970147827206606320217379472010321356624

61380907794230459736069956759583609615871512991382228657857

95493616176544804532220078258184008484364155912294542753848

03558374518022675900061399560145595206127211192918105032491

00800000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000

我们不可能一眼就看出对错，但结果一定是对的。:-)

## 互动模式, 显示控制台

执行 Ruby 而未输入引数，Ruby 会从标准输入 (standard input) 读取指令，在输入结束后开始执行：

% ruby

puts "hello world"

puts "good-bye world"

^D

hello world

good-bye world

上面的 ^D 表示 control-D，可以方便地在 Unix 环境中表示输入结束。在 DOS/Windows 中，可以按 F6 或 ^Z。

Ruby 同时具有 eval.rb 程式，可以利用键盘在互动式回圈 (interactive loop) 中输入 Ruby 程式码，随即就能显示结果。这在本手册中将经常使用。（编注：您可以使用 irb 指令直接进入此互动模式）

若你使用 ANSI 相容的终端机 (terminal)（如果是使用 UNIX 系统，那几乎都是 ANSI 相容的终端机；有些旧版本的 DOS，需要安装 ANSI.SYS 或 ANSI.COM；不幸的是，Windows XP 现在几乎无法使用），就应该使用[加强版 eval.rb](http://guides.ruby.tw/ruby/eval.txt)，提供视觉缩排协助 (visual indenting assistance)、提示报告、颜色强调。不然的话，在可用于任何终端机的非 ANSI 版中，到 sample 子目录中。这是一小段 eval.rb：

% ruby eval.rb （编注：您也可以直接输入 irb 指令直接进入此互动模式）

ruby> **puts "Hello, world."**

Hello, world.

**nil**

ruby> **exit**

hello world 由 puts 产生。下一行（即 nil）报告任何最后计算的结果；Ruby 不会区分 叙述 (statements) 与 表示式 (expressions)，所以计算程式码也就是执行程式码。这个例子中，nil 表示 puts 并无传回任何有意义的值。请注意，我们可以输入 exit 或利用 ^D 离开这个直译器回圈。

本手册中，"ruby>" 代表 eval.rb （编注：或 irb）这个实用小程式的输入提示符。

# 字串(ok)

Ruby 可用于处理字串与数值资料。字串会用双引号 ("...") 或单引号 ('...') 括起。

ruby> **"abc"**

**"abc"**

ruby> **'abc'**

**"abc"**

## 不同情况下，双引号与单引号会有不同效果。

用双引号括起的字串可利用前导的反斜线逸出 (escape) 字元，或是使用 #{} 来计算里面的程式。

用单引号括起的字串并不执行这种行为，

所显示的就是结果。范例：

ruby> **puts "a\nb\nc"**

a

b

c

**nil**

ruby> **puts 'a\nb\n'**

a\nb\nc

**nil**

ruby> **"\n"**

**"\n"**

ruby> **'\n'**

**"\\n"**

ruby> **"\001"**

**"\001"**

ruby> **'\001'**

**"\\001"**

ruby> **"abcd #{5\*3} efg"**

**"abcd 15 efg"**

ruby> **var = " abc "**

**" abc "**

ruby> **"1234#{var}5678"**

**"1234 abc 5678"**

与 C 相比，Ruby 的字串处理更聪明又易用。例如，你可以用 + 串接 (concatenate) 字串，用 \* 重复字串。

ruby> **"foo" + "bar"**

**"foobar"**

ruby> **"foo" \* 2**

**"foofoo"**

用 C 连接字串比较困难，需要明确的记忆体管理才能实行：

char \*s = malloc(strlen(s1)+strlen(s2)+1);

strcpy(s, s1);

strcat(s, s2);

/\* ... \*/

free(s);

而使用 Ruby，我们不需要担心字串占用的空间， 完全摆脱记忆体管理的约束。

你可用字串完成下列事项：

## 串接：

ruby> **word = "fo" + "o"**

**"foo"**

## 重复：

ruby> **word = word \* 2**

**"foofoo"**

提取字元（请注意，Ruby 的字元都是整数）：

ruby> **word[0]**

**102            # 102 是 `f' 的 ASCII 码 （编注：Ruby 1.9 之后版本，变更为回传 f）**

ruby> **word[-1]**

**111            # 111 是 `o' 的 ASCII 码（编注：Ruby 1.9 之后版本，变更为回传 o）**

（负号指从字串结尾计算偏移位置，而不是从字串开头。）

## 提取子字串：

ruby> **herb = "parsley"**

**"parsley"**

ruby> **herb[0,1] #第0位开始, 提取1位**

**"p"**

ruby> **herb[-2,2] #倒数第二位开始, 提取两位**

**"ey"**

ruby> **herb[0..3] #第0位开始到第3位, 包含第3位**

**"pars"**

ruby> **herb[-5..-2] #倒数第五位开始, 到倒数第二位, 包括倒数第二位**

**"rsle"**

## 比对：

ruby> **"foo" == "foo"**

**true**

ruby> **"foo" == "bar"**

**false**

现在使用一下这些功能吧。这是一个“猜字”的谜题，可能“谜题”对以下内容有点太严肃了吧 ;-)

# 把程式储存为 guess.rb

words = ['foobar', 'baz', 'quux']

secret = words[rand(3)]

print "guess?"

while guess = STDIN.gets

  guess.chop!

  if guess == secret

    puts "You win!"

    break

  else

    puts "Sorry, you lose."

  end

  print "guess?"

end

puts "The word was " + secret + "."

现在，先别太在意这个程式码的细节。这个程式运作起来会像这样。

% ruby guess.rb

guess?foobar

Sorry, you lose.

guess?quux

Sorry, you lose.

guess?^D

The word was baz.

# 正则表示式(ok)

来编写更有趣的程式吧。这一次我们使用一个简洁精准的样式 (pattern)来测试一个字串是否符合描述。

这些样式中，有些字元及字元组合具有特殊意思，包括：

|  |  |
| --- | --- |
| [] | 指定的范围（例如：[a-z] 表示一个在 a 到 z 的范围内的字母） |
| \w | 一般字元 (word character)，即 [0-9A-Za-z\_] |
| \W | 非一般字元 (non-word character) |
| \s | 空白字元 (space character)，即 [ \t\n\r\f] |
| \S | 非空白字元 (non-space character) |
| \d | 数字 (digit character)，即 [0-9] |
| \D | 非数字 (non-digit character) |
| \b | 退位 (0x08)（仅用于指定的范围） |
| \b | 单字边界（若不是于指定的范围） |
| \B | 非单字边界 |
| \* | 前一符号的内容出现 0 或数次。 |
| + | 前一符号的内容出现 1 或数次。 |
| {m,n} | 前一符号的内容，最少出现 m 次，最多出现 n 次。 |
| ? | 前一符号的内容最多出现一次，同 {0,1} |
| | | 符合前一个或后一个表示式 |
| () | 分组 |

使用这种奇特词汇的样式，一般称为正则表示式 (regular expressions)。Ruby 和 Perl 一样，都是用斜线 (/) 包住样式内容，而不用双引号。如果你从未接触过正则表示式，那么正则表示式对你来说可能一点也不正规，但是花点时间熟悉一下绝对是明智之举。正则表示式具备有高效率的解释能力 (expressive power)，对于样式比对、搜寻、或操作文字字串，都能为你省却许多麻烦（及不少程式码）。

例如，我们想要测试一个字串是否符合一项描述：“以小写的 f 开头，接着是一个大写的字母，之后随意打些字母，只要没有小写的就行。”如果你熟悉 C 程式，现在脑海大概已经浮现十几行程式码了吧？没错吧，你这是自找麻烦。而用 Ruby 的话，你只要用 /^f[A-Z][^a-z]\*$/ 这个正则表示式测试你的字串就可以。

那如果是测试“在角括号内含有一个十六进位数”呢？也没问题。

ruby> **def chab(s)**

**(s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil**

**end**

ruby> **chab "Not this one."**

  false

ruby> **chab "Maybe this?{0x35}"    # wrong kind of brackets**

  false

ruby> **chab "Or this?<0x38z7e>"    # bogus hex digit**

  false

ruby> **chab "Okay, this: <0xfc0004>."**

  true

虽然正则表示式一眼看来会让人感到迷惘，但你很快就会感到愉悦，因为它可以让你省却不少麻烦。

以下这个小程式可以帮你测试常规表示式（编注：你也可以使用 [Rubular](http://rubular.com/) 这个网路服务做练习）。将它储存为 regx.rb，并在命令列输入 "ruby regx.rb" 以执行程式。

# 需要 ANSI 终端机！

st = "\033[7m"

en = "\033[m"

puts "Enter an empty string at any time to exit."

while true

  print "str> "; STDOUT.flush; str = gets.chop

  break if str.empty?

  print "pat> "; STDOUT.flush; pat = gets.chop

  break if pat.empty?

  re = Regexp.new(pat)

  puts str.gsub(re,"#{st}\\&#{en}")

end

这个程式需要输入两次，一次是字串，另一次是正则表示式。字串就会与正则表示式开始比对，然后将所有匹配的部分反白显示。先别在意细节，很快就会来分析这段程式码。

str> foobar

pat> ^fo+

**foo**bar

~~~

在程式输出中，上面的红色字母会反白显示。"~~~" 是为了方便使用文字浏览器 (text-based browser) 的用户而加的。

多试试几个输入吧。

str> abc012dbcd555

pat> \d

abc**012**dbcd**555**

**~~~    ~~~**

如果对结果感到惊讶，请参考上面的表格：\d 与字母 d 完全没有关系，只是与数字匹配。

那如果有一个以上的结果符合样式呢？

str> foozboozer

pat> f.\*z

**foozbooz**er

~~~~~~~~

符合的是 foozbooz，而不仅仅是 fooz，正则表示式会与最长的子字串匹配。

这是用来隔离冒号区隔时间栏位 (colon-delimited time field) 的样式。

str> Wed Feb  7 08:58:04 JST 1996

pat> [0-9]+:[0-9]+(:[0-9]+)?

Wed Feb  7 **08:58:04** JST 1996

           ~~~~~~~~

就正则表示式而言，"=~" 是匹配的运算子 (operator)，如果发现符合的话，就会在字串中传回位置，没有符合就会传回 nil。

ruby> **"abcdef" =~ /d/**

**3**

ruby> **"aaaaaa" =~ /d/**

**nil**

# 数组(ok)

你可以在 ([]) 中列出几个项目建立一个*数组 (array)*，用逗号隔开项目。Ruby 的数组能够容纳各种型别的物件。

ruby> **ary = [1, 2, "3"]**

**[1, 2, "3"]**

数组与字串一样，能够串接与重复。

ruby> **ary + ["foo", "bar"]**

**[1, 2, "3", "foo", "bar"]**

ruby> **ary \* 2**

**[1, 2, "3", 1, 2, "3"]**

我们可以利用索引号码 (index number) 指示数组的任何部分。

ruby> **ary[0]**

**1**

ruby> **ary[0,2]**

**[1, 2]**

ruby> **ary[0..1]**

**[1, 2]**

ruby> **ary[-2]**

**2**

ruby> **ary[-2,2]**

**[2, "3"]**

ruby> **ary[-2..-1]**

**[2, "3"]**

（负号指从数组结尾计算偏移位置 (offset)，而不是从数组开头。）

数组可用 join 转换为字串，字串可用 split 转换为数组：

ruby> **str = ary.join(":")**

**"1:2:3"**

ruby> **str.split(":")**

**["1", "2", "3"]**

**哈希 (Hashes)**

关连数组 (associative array) 不同于用数字序列的索引号码提取内容，而是用一个 *键 (key)* （编注：可用任何物件当做 Key，但通常是 Symbol、String 字串或 Fixnum 整数）。这种数组一般称为*哈希 (hash)* 或*字典 (dictionary)*，而在 Ruby 世界，我们惯用*哈希 (hash)* 一词。用大括号 ({}) 括起各对项目就能建立哈希。虽然你用索引寻找数组的内容，但寻找哈希的内容则是用键 (key)。

ruby> **h = {1 => 2, "2" => "4"}**

**{1=>2, "2"=>"4"}**

ruby> **h[1]**

**2**

ruby> **h["2"]**

**"4"**

ruby> **h[5]**

**nil**

ruby> **h[5] = 10    # 加入一个项目**

**10**

ruby> **h**

**{5=>10, 1=>2, "2"=>"4"}**

ruby> **h.delete 1   # 用 Key 删除一个项目**

**2**

ruby> **h[1]**

**nil**

ruby> **h**

**{5=>10, "2"=>"4"}**

**字串符号 (Symbol) (编注)**

一个 Symbol 物件皆为冒号开头，例如 :foo 或 :foo\_bar。它的作用就是代表一个名字，最大的作用就是当做 Hash 的键 (key)，例如 { :foo => 1, :bar => 2 }，在这里我们只需要一个识别的名字，不需要使用字串物件。使用 Symbol 写法更简洁和获得效能上的好处

# 继续简单示范

现在我们来分析一下之前的程式范例。

下列程式码在[简单示范](http://www.rubyist.net/%7Eslagell/ruby/examples.html)一节中出现过。

def fact(n)

  if n == 0

    1

  else

    n \* fact(n-1)

  end

end

puts fact(ARGV[0].to\_i)

因为这是我们第一次解释，所以每一行都会仔细说明。

### 阶乘

def fact(n)

第 1 行中，def 用来定义函数 (function) 的叙述（或者准确来说是方法 (method)；我们会在[之后的章节](http://guides.ruby.tw/ruby/methods.html)探讨甚么是方法）。这里指出函数 fact 有一个引数 (argument)，即是 n。

if n == 0

if 是用来判断条件 (condition)。若条件成立，便运行下一行程式码，否则就运行 else 之后的程式码。

1

若条件成立，if 的值就是 1。

else

若条件不成立，就会运行从 else 到 end 的程式码。

n \* fact(n-1)

若条件不成立，if 的值就是 n 乘以 fact(n-1)。

end

第一个 end 结束 if 叙述。

end

第二个 end 结束 def 叙述。

puts fact(ARGV[0].to\_i)

这样便会呼叫 fact() 函数，并传入从命令列传进来的参数，并输出结果。

ARGV 是包含命令列引数的数组。由于 ARGV 的成员都是字串，因此要用 to\_i 转换为整数 (integral number)。Ruby 不会像 Perl 那样，自动将字串转换为整数。

如果这个程式中输入负数，会怎么样呢？你发现问题了吗？能够修正吗？

### 字串

接下来我们会分析[字串](http://guides.ruby.tw/ruby/strings.html)一节的猜字程式。这个程式有点长，所以我们会编号方便指示。

words = ['foobar', 'baz', 'quux']

secret = words[rand(3)]

print "guess?"

while guess = STDIN.gets

  guess.chop!

  if guess == secret

   puts "You win!"

   break

  else

   puts "Sorry, you lose."

  end

  print "guess?"

end

puts "the word is "+ secret + " . "

程式中，使用了 while 这个新的控制结构。若指定的条件为真时，就会重复运行 while 与其对应的 end 之间的程式码。这个范例中，guess=STDIN.gets 既是一个有动作的叙述（收集使用者输入的一行内容，并储存为 guess），也是一项判断条件（若没有输入任何内容，guess 即等于整个 guess=STDIN.gets 表示式的值，而这个值是 nil，会让 while 停止回圈。）

STDIN 是标准输入 (standard input) 物件。一般来说，guess=gets 的功能与 guess=STDIN.gets 一样。

第 2 行的 rand(3) 会传回 0 至 2 间的一个乱数 ( random number)， 用来提取数组 words 中的一个元素。

第 5 行中，我们利用方法 STDIN.gets，从标准输入中提取一行内容。提取时若出现 EOF（档案结尾），gets 会传回 nil。因此 while 程式码会不断重复，直到遇到代表结束输入的 ^D（DOS/Windows 中则是 ^Z 或 F6）。

第 6 行中 guess.chop! 会删除 guess 后的最后一个字元；在本例中这是一个换行 (newline) 字元，这是因为 gets 会包含使用者按下的 Return 键，这是我们不感兴趣的。

在第 15 行中输出谜底。我们将谜底写为三个字串将之相加在一起；这与将 secret 写为 #{secret} 的单一字串效果相同，清楚显示这是将要计算的变数，而不只是逐字输出而已：

puts "the word is #{secret}."

许多程序员都觉得这样能够比较简洁地显示输出的内容。（编注：执行的效率也比较好，不需要呼叫相加两次）

我们现在已经熟悉如何使用 puts 作为标准输出，但这个脚本的第 4 及 13 行却是用 print。它们是不太一样的。

print 会准确输出所提供的内容；

puts 则一定会以输出行 (output line) 结尾。

在第 4 及 13 行使用 print，会让游标停留在显示内容的后面，而不是下一行开头。这可用作提示使用者输入。一般来说，以下四个输出呼叫 (output call) 的结果都是相同。

# 若内容没有换行，则 puts 会隐含加上换行：

puts  "Darwin's wife, Esmerelda, died in a fit of penguins."

# 使用print不会输出换行, 若需换行必须明确加入：

print "Darwin's wife, Esmerelda, died in a fit of penguins.\n"

# 可用 + 连接输出内容：

print 'Darwin's wife, Esmerelda, died in a fit of penguins.'+"\n"

# 或提供多个字串以连接：

print 'Darwin's wife, Esmerelda, died in a fit of penguins.', "\n"

你可能需要留意：文字视窗有时会为了速度而缓冲输出内容，只有遇到换行字元时，才会收集个别字元并加以显示。所以如果使用者猜字后，猜字程式无法如预期般显示提示行，大概就是因为缓冲引起的。输出提示后，你可以 flush 输出内容，避免上述情况发生。它可通知标准输出装置（该物件称为 STDOUT）：“别等了，马上显示缓冲区的内容。”

04 print "guess?"; STDOUT.flush

  ...

13 print "guess?"; STDOUT.flush

事实上，我们在下一个脚本会更小心使用这个指令。

### 正则表示式

最后，我们会讨论[正则表示式](http://guides.ruby.tw/ruby/regexp.html)一节的程式。

1. st = "\033[7m"
2. en = "\033[m"
3. puts "Enter an empty string at any time to exit."
4. while true
5. print "str> "; STDOUT.flush; str=gets.chop
6. break if str.empty?
7. print "pat> "; STDOUT.flush; pat=gets.chop
8. break if pat.empty?
9. re = Regexp.new(pat)
10. puts str.gsub(re, "#{st}\\&#{en}")
11. end

第 6 行中，while 的条件硬性指定为 true，所以形成无穷回圈。但是，我们在第 8 及 10 行输入 break 叙述来逃出回圈。这两个 break 都用了“if 修饰词 (modifier)”，if 修饰词仅在指定条件符合时，才会执行其左边的叙述。这种由右到左的演算逻辑并不常见，但却贴近日常用语的表达形式。由于不需要用 end 叙述来通知直译器之后的程式码有多少需要符合条件，所以也显得简洁。一般来说，只有在叙述与条件能够简短地写在一行的情况下，才会使用 if 修饰词。

请注意这个程式与猜字程式在使用者介面上的差别。使用者可在这个程式的空白行上按下 Return 键离开程式。因此我们测试输入字串是否为空，而非是否不存在。

第 7 及 9 行中有一个“非破坏性的”(non-destructive) 的 chop（去除尾部空白）；同样，我们用 gets 去掉不要的换行字元。加入惊叹号就是“破坏性的”(destructive) chop， 那两者有甚么差别呢？在 Ruby 中，我们一般在方法名称后加上 '!' 或 '?'。惊叹号（! 有时读作 "bang!"）代表具有破坏性，即会改变所接触物件的值。chop! 会直接影响字串，但 chop 则会提供一个删减后的版本，而不影响原本的物件。以下将阐释两者的差异。

ruby> **s1 = "forth"**

  "forth"

ruby> **s1.chop!       # 这改变了 s1。**

  "fort"

ruby> **s2 = s1.chop   # 这将变更后的版本置于 s2，**

  "for"

ruby> **s1             # ⋯⋯而不影响 s1。**

  "fort"

你也会看见有 chomp 及 chomp!。提供更多选择性：字串结尾是一个换行字元时，结尾才会删掉。例如："XYZ".chomp!，并不会有任何效果。记住两者差别的技巧就是，想像人或动物吃东西前，总会先尝尝味道，才会一口咬下去，而不像斧头那样随便就砍下去。

第 8 及 10 行也出现另一种命名方式。问号（? 有时读作 "huh?"）的“述语”(predicate) 方法，表示只会传回 true 或 false。（编注：! 和 ? 结尾的命名方式，只是一个惯例。!暗示有某种副作用，?暗示回传值是 Boolean 布尔值）

第 11 行利用使用者提供的字串，建立一个正则表示式物件。第 12 行则完成真正的工作，使用 gsub 将 ANSI 标记替换 (globally substitute) 到表示式中每个匹配的内容；第 12 行也会输出结果。

我们可以将第 12 行拆成：

highlighted = str.gsub(re,"#{st}\\&#{en}")

puts highlighted

或是“破坏性”的形式：

str.gsub!(re,"#{st}\\&#{en}")

puts str

再看看第 12 行的最后部分， st 及 en 在第 1 至 2 行中定义为 ANSI 码改变文字颜色。并在第 12 行以 #{} 包含其内，确保解释正确（如此我们并不会看到输出变数的名称 (names)）， 两者之间看到的是 \\&。这有点困难， 因为这个替代字串以双引号括起，所以其中的一对斜线会解释成为单一斜线，造成 gsub 实际看到的是 \&，而这刚好是指示一开始符合样式的特别程式码。所以新字串显示时，除了符合指定样式部分会反白显示以外，其余都跟旧字串一样。

# 控制结构

本节将探讨 Ruby 的控制结构。

## case

我们用 case 叙述来测试一连串条件，初看与 C 及 Java 的 switch 相似，但我们之后就会发现 Ruby 的 case 效果更为强大。

ruby> **i=8**

ruby> **case i**

    | **when 1, 2..5**

    |**puts "1..5"**

    | **when 6..10**

    |**puts "6..10"**

    | **end**

6..10

**nil**

2..5 是 2 到 5 范围 (range) 的表示式。以下表示式会测试 i 值是否处于该范围内：

(2..5) === i

case 内部使用关联运算子 (relationship operator) ===，同一时间检查数个条件。为保持 Ruby 的物件导向特质，出现在 when 条件内的物件会以 === 适当解释。例如，以下程式码会测试字串是否等于第一个 when 的字串，再测试是否符合第二个 when 的正则表示式。

ruby> **case 'abcdef'**

    | **when 'aaa', 'bbb'**

    |**puts "aaa or bbb"**

    | **when /def/**

    |**puts "includes /def/"**

    | **end**

includes /def/

**nil**

## while

Ruby 能够以简单的方式建立回圈，不过在下一节中，我们会学会使用迭代器 (iterator)，那就不再经常需要编写明显的回圈。

while 就是重复的 if， 我们曾在猜字游戏及正则表示式中使用过（请见[前一节](http://guides.ruby.tw/ruby/regexp.html)）；句型为 while condition ... end，所包含的程式码区块在条件 (condition) 为真时就会重复。但 while 及 if 也可以简单地应用与个别的叙述：

ruby> **i = 0**

**0**

ruby> **puts "It's zero." if i==0**

It's zero.

**nil**

ruby> **puts "It's negative." if i<0**

**nil**

ruby> **puts i+=1 while i<3**

1

2

3

**nil**

你有时候希望否定一个测试条件， unless 就是 if 的否定型，until 则是 while 的否定型。你可以自行试验看看。

共有四种方式能够从内部中断回圈。

第一种是 break，与 C 的一样，表示完全逸出回圈；

第二种是 next，跳至回圈下一个迭代 (iteration) 的起始处（等同于 C 的continue）；

第三种是 redo，可重新开始目前的回圈。

以下 C 程式码用以阐释 break、next、redo 的意义：

while (condition) {

label\_redo:

   goto label\_next;        /\* ruby's "next" \*/

   goto label\_break;       /\* ruby's "break" \*/

   goto label\_redo;        /\* ruby's "redo" \*/

   ...

   ...

label\_next:

}

label\_break:

...

第四种从内部中断回圈的方式是 return。return 不单能逸出回圈，更会逸出包含回圈的方法。若提供一个参数值，则此方法会回传这个值，否则会传回 nil。

## for

C 程序员现在会好奇怎么建立一个 "for" 回圈。Ruby 的 for 具有相同作用，但更有弹性。以下的回圈会跑一次 集合物件 (collection) 中的每个元素（数组、哈希、连续数值等），无需程序员特别指示索引：

for elt in collection

  # ... elt 是集合物件中的一个元素

end

集合物件可以是数值范围（多数人谈及回圈时，所指的就是这个）：

ruby> **for num in (4..6)**

    |**puts num**

    | **end**

4

5

6

**4..6**

本例中，我们会一一说明数组元素。：

ruby> **for elt in [100,-9.6,"pickle"]**

    |**puts "#{elt}\t(#{elt.class})"**

    | **end**

100    (Fixnum)

-9.6   (Float)

pickle (String)

**[100, -9.6, "pickle"]**

但我们有点超前进度，所以先温习一下之前所学。for 其实就是 each 的另一种写法，即是迭代器的第一个范例。以下两种句型效果相同：

#  如果你习惯用 C 或 Java，你可能会比较喜欢这个。

for element in collection

  ...

end

#  Smalltalk 程序员则可能喜欢这个。

collection.each {|element|

  ...

}

迭代器一般能够替代传统的回圈，熟悉用法后用处会更大。那现在就进入下一节学习吧。

# 迭代器

迭代器 (iterator) 并不是 Ruby 的原创概念， 这已在物件导向程式语言中普遍使用。它也被用在 Lisp 中，只是并不称为迭代器而已。然而很多人仍然不熟悉迭代器的概念，因此以下将详细解释。

迭代 (iterate) 指的是重复做相同的事，所以迭代器 (iterator) 就是用来重复多次相同的事。

我们编写程式码后，需要在不同的情况下回圈。使用 C 时，我会用 for 或 while。例如，

char \*str;

for (str = "abcdefg"; \*str != '\0'; str++) {

  /\* process a character here \*/

}

C 的 for(...) 语法提供建立回圈的抽象概念，但测试空字元 (null character) 的 \*str 程式码，需要程序员了解字串的内部结构。如此让人觉得 C 是低阶的语言。高阶语言应更能弹性地支援迭代。请看一下的 sh shell 脚本：

#!/bin/sh

for i in \*.[ch]; do

  # ... 针对每个档案执行某件事

done

处理目前目录下的所有 C source 及标头档，由命令 shell 负责提取档案名称，并一一替代。这样比 C 看来高阶得多吧，你说呢？

但我们还要留意：虽然语言能够为内建资料型态提供迭代器，但如果我们得为自定的资料型态编写低阶的回圈才能迭代，就太令人失望了。毕竟使用 OOP（物件导向程式）时，我们经常要一一定义资料型态，这就成了严重的问题。

因此每种 OOP 语言都有些方式有助于迭代。有些语言会提供特殊的类别；Ruby 则让我们直接定义迭代。

Ruby 字串 (string) 型态具有一些实用的迭代器：

ruby> **"abc".each\_byte{|c| printf "<%c>", c}; print "\n"**

<a><b><c>

**nil**

each\_byte 是字串中用于处理每个字元的迭代器。每个字元都会代入区域变数 c 中。这段程式也可以改写为类似 C 语言的写法...

ruby> **s="abc";i=0**

**0**

ruby> **while i<s.length**

    |**printf "<%c>", s[i]; i+=1**

    | **end; print "\n"**

<a><b><c>

**nil**

⋯⋯然而，each\_byte 迭代器的概念较简单，而且即使之后字串 (string) 类别的结构大幅改变，仍有较大机会继续运作。迭代器的好处在于面对变化仍能保持耐用，确实称得上是良好的程式码。（好的，耐心点，我们很快就会讨论类别 (class)）

字串 (string) 的另一个迭代器是 each\_line。

ruby> **"a\nb\nc\n".each\_line{|l| print l}**

a

b

c

**nil**

比起 C 程式，迭代器更能事半功倍（寻找行分隔符号 (line delimiter)、建立子字串等）。

前一节的 for 叙述就是使用 each 迭代器进行迭代的。字串 (string) 的 each 与 each\_line 效果相同（编注：Ruby 1.9 之后版本，移除了字串的 each 函式，所以必须改用 each\_line。此例在 1.9 之后版本无法执行。），那么就将上述的范例用 for 改写：

ruby> **for l in "a\nb\nc\n"**

    |**print l**

    | **end**

a

b

c

**nil**

将控制结构 retry 与迭代的回圈一起运作，就能重新开始回圈。

ruby> **c=0**

**0**

ruby> **for i in 0..4**

    |**print i**

    |**if i == 2 and c == 0**

    |**c = 1**

    |**print "\n"**

    |**retry**

    |**end**

    | **end; print "\n"**

012

01234

**nil**

将 retry 替换为 redo，会令目前的回圈重做一次迭代，输出内容为：

012

234

yield 有时会出现在迭代器的定义中。yield 可将控制移至传递往迭代器的程式码区块 (block of code)（这将在一节中讨论）。以下范例定义了迭代器 repeat，这个迭代器可按引数所指定的次数，重复程式码区块。

ruby> **def repeat(num)**

    |**while num > 0**

    |**yield**

    |**num -= 1**

    |**end**

    | **end**

**nil**

ruby> **repeat(3) { puts "foo" }**

foo

foo

foo

**nil**

使用 retry，就能定义类似 Ruby 标准的 while 迭代器。

ruby> **def WHILE(cond)**

    |**return if not cond**

    |**yield**

    |**retry**

    | **end**

**nil**

ruby> **i=0; WHILE(i<3) { print i; i+=1 }**

012   nil

你明白甚么是迭代器了吗？虽然仍有些限制，但你已经可以自行编写迭代器。事实上，每当定义新资料型态时，常会一并定义合适的迭代器，方便作为搭配。因此，上述范例并不说得上非常有用。了解甚么是类别 (class) 后，再来讨论一些实用的迭代器吧。

# 物件导向思维

*物件导向 (object oriented)* 是个引人注意的词汇， 把东西称为物件导向，让人听起来觉得很聪明。Ruby 宣称为物件导向程式语言，但到底“物件导向”是甚么意思？

答案各式各样，但重点都是一样。先不这么快就下结论，让我们先想想传统的程式设计范式 (programming paradigm)。

一般来说，程式设计遇到的问题都是要想出*资料表示法 (data representation)* 及在资料执行的*程序 (procedure)*。这种情况下，资料的性质是惰性、被动、无法自主的，完全任由庞大的程式体摆布，而这个程式体是主动、逻辑、全能的。

这种方式的问题在于程式需要由程序员编写，而程序员只能靠脑袋来记住大量的细节。当项目变大，程序核心也会持续成长，最终使程序员无法记住全部程 式。思考出错或打错程式码都会成为难以察觉的程序错误， 导致程序核心出现复杂而意外的运算，维护工作将会让你手忙脚乱。传统程式设计范式会提供指引，减少或找出程序错误，但最好的解决方法还是从根本作出改变。

物件导向程式设计能够将单调而重复的逻辑工作委派予*资料*，把我们对资料的概念从*被动*转为*主动*。换而言之，

* 资料不再是一个开盖盒子，只能让我们自己去找东西或放东西进去。
* 资料变成是能够运作的机器，具有标示明确的开关与功能。

上述的“机器”内部可能非常复杂，也可能非常简单，并不能单以外部作出判断，我们不会随意拆开机器（除非我们非常确定设计出现问题），所以我们只需要按按开关，了解有甚么功能可以与资料互动。机器做好后，我们就不会再去想它是如何运作了。

你可能会觉得这样好像更麻烦，但是这种方式能够预防各种问题。

以下范例虽然没什么实用价值，但也能解释部分概念。你的车上有里程表， 可用来计算按下重设键后所行驶的路程。我们可如何将这编写为程式语言呢？在 C 中，里程表只是个数值变数，很可能就是浮点数 (float)。 程式会慢慢递增变数，有时可能会将变数重设为零。这样的程式有甚么问题吗？任何非预期的臭虫(Bug)程式都可能将此变数设定成错误值。使用过 C 的人都知道，要找出错误可能要花上数小时甚至数天，而最后才发现那个错误原来很无稽。（找到错误的那一刻，才让人恍然大悟。）

从物件导向角度看来，这个程式也有问题。程序员设计里程表时，首先想到的不是“哪些资料类型与这个最为类似？”，而是“这程式到底会怎么运作？”其 中的差别就会形成严重的问题。应该花点时间决定里程表的用处，以及人们会如何使用。我们要制作能够操作小机器，能够递增数值、重设、显示数值，就这些功能 而已。

我们不会让里程表有机会加入任意值， 因为里程表根本不需要这个功能。里程表只具有几项功能而已，这些功能我们已经赋予了它。因此，如果程式错误地尝试加入其他数值（车辆气候控制的温度）至里 程表中，就会立即指出哪里出现问题。我们知道运行程式期间，（或编译 (compiling)，视乎程式语言而定），*我们无法在里程表中加入任意值*。这可能不是非常清楚，但仍是清晰合理。这不能预防错误发生， 但能够帮我们迅速地找出问题。这就是物件导向程式设计帮我们节省时间的其中一项方式。

接着让我们进一步抽象化以上的范例，因为建立一间制造机器的工厂，跟制造一个机器其实同样简单。因此我们并不是直接建立一个里程表，而是建立一个可 以制造里程表的工厂，利用一个模子(pattern)来生产任意个里程表。这个模子(pattern)样式（或是你想要称为里程表工厂）我们称作*类别 (class)*，而从样式产生的个别里程表（或工厂制造的里程表）我们称作*物件 (object)*。大部分物件导向语言要求先定义类别 (class)，才能开始新的物件(object)，但 Ruby 并无这种限制。

要注意的是，使用物件导向语言不会*保证有*好的物件导向设计。确实任何语言的程式码都有可能编写得不清不楚、凌乱、毫不周详⋯⋯ Ruby 的优点在于（特别是相对 C++ 而言）能够自然地编写物件导向程式，即使编写小程式，也不用花费精力来解决零乱的程式码。本手册之后会讨论如何利用 Ruby 达成目标，下一节将是“开关与功能”（物件方法），然后是“工厂”（类别）。你还在看吧？

# 方法

甚么是方法？物件导向程式设计中，我们不希望直接在物件以外处理资料，相反地，物件本身应了解自己如何运作（当你友善地请求它去做事情时）。你可以 这样想，我们把讯息传给物件，物件就会作出某些动作或有意义的回应。我们不需要了解或关心物件内部实际如何运作。这个允许我们要求物件执行的工作（或者 说，这个物件可以了解的讯息），就称之为物件*方法 (method)*。

在 Ruby 中，我们利用点符号 (.) 呼叫物件的方法（与 C++ 或 Java 相同）。接收的物件位于点符号左侧。

ruby> **"abcdef".length**  
**6**

直观上，这个字串物件被询问了字串的长度。技术上而言，我们呼叫了该物件 "abcdef" 的 length 方法。

其他物件可能在解释 length 上有些微差别，或者完全一样。要在程式执行中，才会决定如何应对讯息，而采取甚么动作则取决于变数所指的是甚么物件。

ruby> **foo = "abc"**  
**"abc"**  
ruby> **foo.length**  
**3**  
ruby> **foo = ["abcde", "fghij"]**  
**["abcde", "fghij"]**  
ruby> **foo.length**  
**2**

*长度 (length)* 的意义取决于所讨论的物件。上述范例中，我们第一次询问 foo 的长度时，所指的是字串，而这只有一个合理的答案。第二次时，foo 指的是数组，我们就可能合理地想到长度可能是 2、5 或 10，但最适当的答案当然是 2（如果希望的话，也可以想出其他种类的长度）。

ruby> **foo[0].length**  
**5**  
ruby> **foo[0].length + foo[1].length**  
**10**

要注意的是，数组*知道身为数组的责任*。Ruby 的资料都明白自己的责任，所以收到要求后，他们都会以适当的方式完成工作。这可减轻程序员的工作，无需记住一大堆函数名称，只要记住较少量的方法名称即 可，这与我们知道如何用自然语言表达的概念相同，并可应用于不同种类的资料，同样能得到预期的结果。物件导向程式设计语言的这项特色（我个人认为 Java 在这方面并不出色）称为*多型 (polymorphism)*。

如果物件收到不能理解的讯息，错误就会“丢出”(raise)：

ruby> **foo = 5**  
**5**  
ruby> **foo.length**  
**ERR: (eval):1: undefined method `length' for 5(Fixnum)**

虽然我们不需要知道方法如何运作，但必须清楚物件能够接受那些方法。

若方法具有引数，引数通常用括号括起，

object.method(arg1, arg2)

但如果并不会造成歧义的话，也可以省略。

object.method arg1, arg2

Ruby 有个特别的变数称为 self，指向现在的物件（也就是呼叫此方法的物件）。因为经常使用 "self."，所以在物件本身呼叫方法时，可以省略：

self.method\_name(args...)

即等同于

method\_name(args...)

我们可以把传统的*函数呼叫 (function call)* 当做是物件 method，只是省略了 self 呼叫。因此 Ruby 可称为纯物件导向语言（编注：因为所有函式都在物件之中）。当然，这个函数呼叫与其他程式语言的函数非常类似，这也方便了那些不明白 Ruby 中函数呼叫就是物件方法的人。如果必要的话，我们会说 "*函数 (functions)*"，以跟物件方法(object methods)做区别。

# 类别

这个世界充满着物件，我们都能把它们分门别类。例如，小孩子看到狗，不管是甚么品种，可能都会叫它作“汪汪”；我们自然地就会把世界的不同物件归类。

物件导向程式语言的术语中，像“狗”这种类型，我们会称为*类别 (class)*，而属于该类别的物件就称为*实例 (instance)*。

在 Ruby 或其他物件导向程式语言中，要建置一个物件，首先要定义类别的特征，然后建立一个实例。下例阐述这个过程，首先定义一个简单的 Dog 类别。

ruby> **class Dog**  
    |**def speak**  
    |**puts "Bow Wow"**  
    |**end**  
    | **end**  
**nil**

Ruby 中，关键字 class 及 end 中间的程式码，即是类别的定义。这个范围中，从 def 开始则定义了类别的*方法 (method)*（如上一节所述），对应于该类别物件的某些行为。

我们已经定义了 Dog 的类别，现在就能建立狗 (dog) 了：

ruby> **pochi = Dog.new**  
**#<Dog:0xbcb90>**

我们为 Dog 类别建立一个新实例，并命名为 pochi。任何类别的 new 方法都可用于建立新实例。根据我们类别的定义，pochi 属于 Dog 类别，因此具备 Dog 的所有特性。因为我们对于 Dog 的概念很简单，我们只能让 pochi 玩一项把戏。

ruby> **pochi.speak**  
Bow Wow  
**nil**

为类别建立实例，有时也称为*实例化 (instantiating)* 该类别。我们要先有狗，才能享受与它对话的乐趣。我们无法让 Dog *类别 (class)* 对着我们叫。

ruby> **Dog.speak**  
**ERR: (eval):1: undefined method `speak' for Dog:class**

这就好像你无法吃一个 *三明治的抽象概念*。

另一方面，如果我们想要听见一只狗的叫声，而不想有任何情感的连系，我们可以创造（实例化）一只暂时的狗，让它发出声音后就消失。

ruby> **(Dog.new).speak   # 或常用的是 Dog.new.speak**  
Bow Wow  
**nil**

你会说：“等一下，这可怜的小狗叫完就会消失？”是的：如果我们懒得为它取名字（就好像我们的 pochi），Ruby 的自动垃圾收集功能会认为这是一只没人要的垃圾狗，就会无情地把它抛弃。这其实没什么，反正我们可以随意创造我们要的狗。

# 继承

日常生活中的物件类别都会分有阶层， 我们知道*所有猫都是哺乳类*，*所有哺乳类都是动物*。较小的类别会*继承 (inherit)* 具有所属较大类别的特征。如果所有哺乳类都会呼吸，那么所有猫也会呼吸。

我们可以在 Ruby 表达这种概念：

ruby> **class Mammal**  
    |**def breathe**  
    |**puts "inhale and exhale"**  
    |**end**  
    | **end**  
**nil**  
ruby> **class Cat < Mammal**  
    |**def speak**  
    |**puts "Meow"**  
    |**end**  
    | **end**  
**nil**

虽然我们没有指定 Cat 应该呼吸，但因为 Cat 被定义为 Mammal 的子类别，所以就会继承 Mammal 类别的行为。（物件导向程式语言的术语中，较小的类别称为*子类别 (subclass)*，较大的类别称为*父类别 (superclass)*。）从程序员的角度看来，猫很自然就会呼吸，我们再加入 speak （讲话）的方法后，我们的猫就会呼吸讲话。

ruby> **tama = Cat.new**  
**#<Cat:0xbd80e8>**  
ruby> **tama.breathe**  
inhale and exhale  
**nil**  
ruby> **tama.speak**  
Meow  
**nil**

有时，特定子类别不应继承父类别的某些特性。例如，鸟类一般都会飞，但企鹅就是不会飞的鸟。

ruby> **class Bird**  
    |**def preen**  
    |**puts "I am cleaning my feathers."**  
    |**end**  
    |**def fly**  
    |**puts "I am flying."**  
    |**end**  
    | **end**  
**nil**  
ruby> **class Penguin < Bird**  
    |**def fly**  
    |**fail "Sorry.I'd rather swim."**  
    |**end**  
    | **end**  
**nil**

我们不需要彻底地为每个新类别定义每个特征，只要加入或重新定义子类别与父类别的差异即可 (编注：我们在 Penguin 里覆写掉 fly 这个方法)。继承有时候也称为*差分程式设计 (differential programming)*， 这也是物件导向程式设计的其中一项优点。

# 重新定义方法(方法覆盖)

在子类别中，我们重新定义父类别方法，就能改变实例的行为。

ruby> **class Human**  
    |**def identify**  
    |**puts "I'm a person."**  
    |**end**  
    |**def train\_toll(age)**  
    |**if age < 12**  
    |**puts "Reduced fare.";**  
    |**else**  
    |**puts "Normal fare.";**  
    |**end**  
    |**end**  
    | **end**  
**nil**  
ruby> **Human.new.identify**  
I'm a person.  
**nil**  
ruby> **class Student1<Human**  
    |**def identify**  
    |**puts "I'm a student."**  
    |**end**  
    | **end**  
**nil**  
ruby> **Student1.new.identify**  
I'm a student.  
**nil**

如果我们要增强父类别的 identify 方法，而不是完全替换掉， 我们就会使用 super（编注： super 让我们可以呼叫到父类别的同名函式，即 identify)。

ruby> **class Student2<Human**  
    |**def identify**  
    |**super**  
    |**puts "I'm a student too."**  
    |**end**  
    | **end**  
**nil**  
ruby> **Student2.new.identify**  
I'm a person.  
I'm a student too.  
**nil**

super 能够把引数传至原本的方法中。有时候会说，人有两种⋯⋯

ruby> **class Dishonest<Human**  
    |**def train\_toll(age)**  
    |**super(11) # 我们想要便宜的价格。**  
    |**end**  
    | **end**  
**nil**  
ruby> **Dishonest.new.train\_toll(25)**  
Reduced fare.  
**nil**  
  
ruby> **class Honest<Human**  
    |**def train\_toll(age)**  
    |**super(age) # 传进我们提供的参数**  
    |**end**  
    | **end**  
**nil**  
ruby> **Honest.new.train\_toll(25)**  
Normal fare.  
**nil**

# 存取控制

之前我们说过，Ruby 没有函数，只有方法。但 Ruby 有不只一种的方法， 本节将介绍*存取控制 (access controls)*。

如果我们在“最高层级”(top level) 定义方法，而不是在类别中定义，那会怎样呢？我们会发现这个方法与 C 这种传统语言中的*函数 (function)* 相似。

ruby> **def square(n)**  
    |**n \* n**  
    | **end**  
**nil**  
ruby> **square(5)**  
**25**

我们的新方法不属于任何类别，但其实 Ruby 将它指定为 Object 类别，即所有类别的父类别。因此，任何物件现在都能使用这个方法。但有个小问题：这是所有类别的*私有 (private)* 方法。我们之后会解释这是甚么意思，但其中一个影响就是它只能以函数的方式呼叫出来，如下所示：

ruby> **class Foo**  
    |**def fourth\_power\_of(x)**  
    |**square(x) \* square(x)**  
    |**end**  
    | **end**  
  nil  
ruby> **Foo.new.fourth\_power\_of 10**  
  10000

我们不能明确地将该方法应用于物件：

ruby> **"fish".square(5)**  
**ERR: (eval):1: private method `square' called for "fish":String**

这样就能聪明的保留 Ruby 的纯物件导向特质（函数仍为物件方法，只是接收器 (receiver) 是隐含的 self 而已）。

物件导向程式设计的常见心智训练 (mental discipline)（曾在之前章节透露过），在于辨别*规格 (specification)* 与*实作 (implementation)*，或是物件应该要完成*甚么 (what)* 工作，以及*如何 (how)* 完成。使用者一般不知道物件的内部运作方式，只需要了解输入输出的是甚么，并相信物件的内部运作即可。因此，不同类别拥有外界不知道的方法，有时也是有用 的，但这些方法只用于内部（有需要的话，程序员可予以改进，而不改变使用者对于物件类别的看法）。以下范例中，请想像 engine 是该类别的隐形内部运作方式。

ruby> **class Test**  
    |**def times\_two(a)**  
    |**puts "#{a} times two is #{engine(a)}"**  
    |**end**  
    |**def engine(b)**  
    |**b\*2**  
    |**end**  
    |**private:engine  # 让使用者看不到 engine**  
    | **end**  
**Test**  
ruby> **test = Test.new**  
**#<Test:0x4017181c>**  
ruby> **test.engine(6)**  
**ERR: (eval):1: private method `engine' called for #<Test:0x4017181c>**  
ruby> **test.times\_two(6)**  
6 times two is 12.  
**nil**

我们会预期 test.engine(6) 传回 12，但我们在外使用 Test 物件时，engine 方法是不能呼叫的。只有其他 Test 的方法（如 times\_two）才能使用 engine。我们需要使用公开介面 (public interface) 的 times\_two 方法。负责该类别的程序员能够随意变更 engine（可能把 b\*2 改为 b+b，假设引数提高了效率），而不会影响使用者与 Test 物件的互动。这个例子确实比较简单，但我们创造更复杂有趣的类别时，存取控制的优点就会更明显。 （编注：这就叫做物件导向的封装概念)

# 单件方法

实例 (instance) 的行为由其所属的类别决定，但有时候我们知道某个实例应该具有特定行为。大部分语言中，我们必须大费周章定义其他类别，但只能实例化 (instantiate) 一次。而 Ruby 能为所有物件提供自己的方法。

ruby> **class SingletonTest**  
    |**def size**  
    |**25**  
    |**end**  
    | **end**  
**nil**  
ruby> **test1 = SingletonTest.new**  
**#<SingletonTest:0xbc468>**  
ruby> **test2 = SingletonTest.new**  
**#<SingletonTest:0xbae20>**  
ruby> **def test2.size**  
    |**10**  
    | **end**  
**nil**  
ruby> **test1.size**  
**25**  
ruby> **test2.size**  
**10**

本例中，test1 与 test2 属于同一类别，但 test2 具有重新定义的 size 方法，因此两者的行为会不一样。只给予单一物件的方法称为*单件方法 (singleton method)*。

单件方法经常用于图形使用者介面 (graphic user interface, GUI) 的元素，当按下不同按钮，就会执行不同动作。

单件方法并不是 Ruby 独有的，CLOS、Dylan 等也有。有些语言例如 Self 及 NewtonScript 更只有单件方法。这有时会称为*原型 (prototype-based)* 语言。

# 模组

Ruby 的模组 (module) 与类别相似，除了：

* 模组并没有实例。
* 模组并没有子类别。
* 模组由 module ... end 定义。

其实⋯⋯模组的 Module 类别正是类别的 Class 类别的父类别。懂吗？不懂？继续看下去吧。

模组有两种用法， 一种将各式相关的方法与常数 (constant)收集在一起（编注：即做为 Namespace 用途）。例如 Math 模组在 Ruby 的标准程式库 (standard library)：

ruby> **Math.sqrt(2)**  
**1.41421**  
ruby> **Math::PI**  
**3.14159**

:: 运算子会通知 Ruby 直译器，应向哪个模组查询常数的值（Math 以外的模组可能因为 PI 而具有其他意义）。如果想直接存取模组内的方法或常数而不需要使用 ::，可以 include 该模组：

ruby> **include Math**  
**Object**  
ruby> **sqrt(2)**  
**1.41421**  
ruby> **PI**  
**3.14159**

另一种用法称为*混入 (mixin)*。包括 C++ 的一些物件导向程式语言，允许*多重继承 (multiple inheritance)*，即是继承自多个父类别。闹钟可说是现实生活的多重继承范例，闹钟既属于*时钟 (clock)* 类别，也属于*响闹器 (buzzer)* 类别。

Ruby 并不具有真正的多重继承，但*混入 (mixin)* 技巧是不错的替代品。还记得模组没有实例也没有子类别吗？但如果我们在类别定义中 include 模组，方法就能有效地加入或混入 (mixin) 类别中。

混入 (mixin) 可以看做是一种可以加入各种特性的技巧。例如：一个拥有 each 方法的类别，混入标准程式库的 Enumerable 模组后，就能使用 sort 及 find 方法。

模组能为我们提供多重继承的基本功能，让我们以简单的树状结构 (tree structure) 显示类别关系，从而大幅简化语言实作 (language implementation)（与 Java 设计者的抉择类似）。

# 程序物件

当有意料之外的事件发生时，谁都希望能够有指定的回应措施。这其实很简单，只要我们能够将程式码区块如引数般传至其他方法，也就是能够像使用 资料 般一样对待 程式码。

使用 Proc.new 创造一个新的*程序物件 (procedure object)* （编注：原英文版使用 proc 方法，现在的惯例改成使用 Proc.new 或 lambda)：

ruby> **quux = Proc.new {**  
    |**puts "QUUXQUUXQUUX!!!"**  
    | **}**  
**#<Proc:0x4017357c>**

现在 quux 所指的是一个物件，与大部分物件一样，具有能够被呼叫的行为。具体来说，可以透过 call 方法来执行：

ruby> **quux.call**  
QUUXQUUXQUUX!!!  
**nil**

那么，可以将 quux 用作方法引数吗？当然可以。

ruby> **def run( p )**  
    |**puts "About to call a procedure..."**  
    |**p.call**  
    |**puts "There: finished."**  
    | **end**  
**nil**  
ruby> **run quux**  
About to call a procedure...  
QUUXQUUXQUUX!!!  
There: finished.  
**nil**

利用 trap 方法，我们能够针对任何系统信号，指派不同应对措施。

ruby> **inthandler = Proc.new { puts "^C was pressed." }**  
**#<Proc:0x401730a4>**  
ruby> **trap "SIGINT", inthandler**  
**#<Proc:0x401735e0>**

一般来说，按下 *^C* 会离开直译器。现在，会输出一个讯息，然后直译器继续运行，不会失去正在进行的工作。（你不会一直受困于直译器内，随时都可以输入 exit 来离开。）

前往下一节前，我们要注意：与信号连结 (binding) 前，并不一定要为程序物件命名。*匿名 (anonymous)* 的程序物件就像

ruby> **trap "SIGINT", Proc.new { puts "^C was pressed." }**  
**nil**

或更常见的写法如下：

ruby> **trap "SIGINT" do**  
    |**puts "^C was pressed."**   
    |**end**   
**nil**

如果只有一行，我们会更简洁点地写成：

ruby> **trap("SIGINT") { puts "^C was pressed." }**   
**nil**

(编著：此段范例与原英文版略有不同)

# 变量

Ruby 有三种变数、一种常数和两个拟变数 (pseudo-variable)。变数与常数并不需要型别， 虽然无型别 (untyped) 的变数有一些缺点，但有更多优点，而且符合 Ruby *快捷易用*的理念。

大部分语言中，变数都需要宣告，才能指定型别、可修改性 (modifiability)（即，是不是常数）、变量作用域 (scope)；因为型别不再是个问题，我们会在之后的变量名称中证实，所以 Ruby 并不需要变量宣告。

透过识别符 (identifier) 的第一个字元，就能分辨出变量：

|  |  |
| --- | --- |
| $ | 全域变量 (global variable) |
| @ | 实例变量 (instance variable) |
| [a-z] 或 \_ | 区域变量 (local variable) |
| [A-Z] | 常数 |

例外的是 Ruby 中的两个拟变量 (pseudo-variable)：self、永远指示目前正在执行的物件；nil、这是指派予未初始化 (uninitialized) 变量的无意义值。这两个都如同区域变量，self 是由直译器维护的全域变量，而 nil 则是一个常数。幸好只有两个例外，并不会造成太大困惑。

你不能为 self 或 nil 指派任何值。main（self 的值）指的是上层 (top-level) 物件：

ruby> **self**  
**main**  
ruby> **nil**  
**nil**

## 全域变量

全域变量以 $ 开头， 可从程式中任何地方开始。初始化前，全域变量有一个特别的值－－nil。

ruby> **$foo**  
**nil**  
ruby> **$foo = 5**  
**5**  
ruby> **$foo**  
**5**

应尽量少用全域变量， 因为全域变量能够编写于任何地方，所以相对危险。过量使用全域变量会令人难以找出错误，也会让人觉得程式设计并没经过深思熟虑。如果一定要使用全域变量， 请记得指定一个描述性名称 (descriptive name)，以防之后重复使用该名称（以上例子中的 $foo 并不是好的示范）。

全域变量其中一个优点就是能够追踪，每当变量的值改变后，你都能够指出呼叫出的程序。

ruby> **trace\_var :$x, Proc.new { puts "$x is now #{$x}" }**  
**nil**  
ruby> **$x = 5**  
$x is now 5  
**5**

当全域变量改变后，就会用作呼叫程序的触发器 (trigger)，有时会称之为*活跃变量 (active variable)*。例如，可用于为图形使用者介面 (GUI) 保持最新状态。

以下是一系列由 $ 开头的特别变量， 例如 $$ 包含 Ruby 直译器的进程识别字 (process id)，只能读取。这些都是主要的系统变量：

|  |  |
| --- | --- |
| $! | 最新的错误讯息 |
| $@ | 错误的位置 |
| $\_ | gets 最后读取的子串 |
| $. | 直译器最后读取的行*数* |
| $& | 正规表示式 (regexp) 最后符合的字串 |
| $~ | 最后的正规表示式符合内容，子表示式 (subexpression) 的数组 |
| $*n* | 最后符合内容的第 *n* 项子表示式（同 $~[*n*]）。 |
| $= | 不区分大小写旗标 |
| $/ | 输入记录分隔字元 |
| $\ | 输出记录分隔字元 |
| \$0 | Ruby 脚本档案的名称 |
| $\* | 命令列引数 |
| $$ | 直译器的进程识别字 (process id) |
| $? | 离开最后执行的子进程 (child process) 状态 |

以上的 $\_ 与 $~ 具有区域作用域 (local scope)， 他们的名称看起来是全域变量，但这样比较有用，而且使用这样的名称也有其历史因素。

## 实例变量

实例变量 (instance variable) 以 @ 开头，作用域仅限定于 self 指示的物件。即使是属于同一类别的两个不同物件，他们的实例变量也允许具有不同的值。除非程序员明确地提供其他方法，否则实例变量无法从物件以外变更或查看（Ruby 的实例变量从不*公开 (public)*）。与全域变量一样，实例变量在初始化前，也有 nil 值。

实例变量不需要宣告， 代表实例变量具有弹性的物件结构；其实，实例变量首次获指派时，都是动态加入至物件的。

ruby> **class InstTest**  
    |**def set\_foo(n)**  
    |**@foo = n**  
    |**end**  
    |**def set\_bar(n)**  
    |**@bar = n**  
    |**end**  
    | **end**  
**nil**  
ruby> **i = InstTest.new**  
**#<InstTest:0x83678>**  
ruby> **i.set\_foo(2)**  
**2**  
ruby> **i**  
**#<InstTest:0x83678 @foo=2>**  
ruby> **i.set\_bar(4)**  
**4**  
ruby> **i**  
**#<InstTest:0x83678 @foo=2, @bar=4>**

请注意，呼叫 set\_bar 方法前，i 并不会为 @bar 回报值。

## 区域变量

区域变量 (local variable) 以小写字母或底线字元 (\_) 开头， 与全域变量及实例变量不同，区域变量初始化前并没有 nil 值：

ruby> **$foo**  
**nil**  
ruby> **@foo**  
**nil**  
ruby> **foo**  
**ERR: (eval):1: undefined local variable or method `foo' for main(Object)**

第一次指派予区域变量的工作，其性质就像是宣告。如果你使用一个没有初始化的区域变量，Ruby 直译器可能会以为你指的是一个假值 (bogus variable)，例如可能是拼错方法名称。就会出现上面看的错误讯息。

一般来说，区域变量的作用域包括

* Proc.new{ ... } 或 lambda{ ... }
* loop{ ... }
* def ... end
* class ... end
* module ... end
* 整份脚本（除非出现以上任何一项）

下例中，defined? 是检查识别符 (identifier) 有否定义的运算子。若已定义，就会传回该识别符的描述，否则就传回 nil。如你所见，bar 的作用域位于回圈内；若回圈结束，bar 就变成未定义了。

ruby> **foo = 44; puts foo; defined?(foo)**  
44  
**"local-variable"**  
ruby> **loop{bar=45; puts bar; break}; defined?(bar)**  
45  
**nil**

位于同一作用域的程序物件（编注：即 Proc.new 和 lambda 里)，会共享该作用域内的所有区域变量。因此，区域变量 bar 由 main、程序物件 p1 及 p2 所共享：

ruby> **bar=nil**  
**nil**  
ruby> **p1 = Proc.new {|n| bar=n}**  
**#<Proc:0x8deb0>**  
ruby> **p2 = Proc.new {bar}**  
**#<Proc:0x8dce8>**  
ruby> **p1.call(5)**  
**5**  
ruby> **bar**  
**5**  
ruby> **p2.call**  
**5**

请注意，开头的 "bar=nil" 不能省略，它可确保 bar 的作用域涵盖 p1 及 p2。否则 p1 及 p2 会使用各自的区域变量 bar，呼叫 p2 时，就会出现 "undefined local variable or method"（未定义区域变量或方法）错误。我们可以写 bar=0，但使用 nil 对于之后看程式码的人则比较礼貌。这样就能够清楚的显示你只是设定作用域，因为所指派的值并无任何意义。

程序物件具有一项强大的功能，就是能够以引数的方式传递：共享的区域变量就算不在原有的作用域，仍然保持有效。

ruby> **def box**  
    |**contents = nil**  
    |**get = Proc.new {contents}**  
    |**set = Proc.new {|n| contents = n}**  
    |**return get, set**  
    | **end**  
**nil**  
ruby> **reader, writer = box**  
**[#<Proc:0x40170fc0>, #<Proc:0x40170fac>]**   
ruby> **reader.call**  
**nil**  
ruby> **writer.call(2)**  
**2**  
ruby> **reader.call**  
**2**

Ruby 对于作用域非常聪明， 从例子中就能够证明如此，contents 变量由 reader 及 writer 所共享。但我们也能利用以上定义的 box，制造多重的阅读器－写入器 (reader-writer) 对，每对阅读器－写入器可共享 contents 变量，而且对与对之间并不会互相干扰。

ruby> **reader\_1, writer\_1 = box**  
**[#<Proc:0x40172820>, #<Proc:0x4017280c>]**  
ruby> **reader\_2, writer\_2 = box**  
**[#<Proc:0x40172668>, #<Proc:0x40172654>]**  
ruby> **writer\_1.call(99)**  
**99**  
ruby> **reader\_1.call**  
**99**  
ruby> **reader\_2.call  # 该 box 中尚无任何内容**  
**nil**

这种程式设计也具有物件导向框架。box 方法的作用就像类别，get 与 set 就像是方法（那些不是真正的方法*名称*除外，他们会随着不同的 box 实例而不同），而 contents 则是单独的实例变量。当然，使用 Ruby 认可的类别框架，可编写出更容易阅读的程式码。

## 类别常数

常数都是以大写字母开头， 最多获指派一个值。在 Ruby 中，重新指派一个常数，会出现警告而不是错误（eval.rb 的非 ANSI 版并不会出现警告）：

ruby>**fluid=30**  
**30**  
ruby>**fluid=31**  
**31**  
ruby>**Solid=32**  
**32**  
ruby>**Solid=33**  
**(eval):1: warning: already initialized constant Solid**  
**33**

常数可于类别内定义，与实例常数不同的是，常数可于类别以外存取。

ruby> **class ConstClass**  
    |**C1=101**  
    |**C2=102**  
    |**C3=103**  
    |**def show**  
    |**puts "#{C1} #{C2} #{C3}"**  
    |**end**  
    | **end**  
**nil**  
ruby> **C1**  
**ERR: (eval):1: uninitialized constant C1**  
ruby> **ConstClass::C1**  
**101**  
ruby> **ConstClass.new.show**  
101 102 103  
**nil**

常数可于模组内定义。

ruby> **module ConstModule**  
    |**C1=101**  
    |**C2=102**  
    |**C3=103**  
    |**def showConstants**  
    |**puts "#{C1} #{C2} #{C3}"**  
    |**end**  
    | **end**  
**nil**  
ruby> **C1**  
**ERR: (eval):1: uninitialized constant C1**  
ruby> **include ConstModule**  
**Object**  
ruby> **C1**  
**101**  
ruby> **showConstants**  
101 102 103  
**nil**  
ruby> **C1=99  # 这样不好**  
**99**  
ruby> **C1**  
**99**  
ruby> **ConstModule::C1**  
**101**  
ruby> **ConstModule::C1=99   # ⋯⋯ 在旧版本中这是不允许的**  
**(eval):1: warning: already initialized constant C1**  
**99**  
ruby> **ConstModule::C1  # “自讨苦吃”**  
**99**

# 例外处理

## 救援rescue (类似catch)

执行中的程式可能出现意料之外的问题。例如：发现要读取的档案不存在；储存资料时发现磁碟已满；输入不合适的内容。

ruby> **file = open("some\_file")**  
**ERR: (eval):1:in `open': No such file or directory - some\_file**

稳健的程式能够快速发现问题，并完善解决， 要达成这样的目标总是让人沮丧。C 程序员需要检查任何可能出错的系统呼叫 (system call)，并立即决定应对措施：

FILE \*file = fopen("some\_file", "r");  
if (file == NULL) {  
  fprintf( stderr, "File doesn't exist.\n" );  
  exit(1);  
}  
bytes\_read = fread( buf, 1, bytes\_desired, file );  
if (bytes\_read != bytes\_desired ) {  
  /\* do more error handling here ... \*/  
}  
...

这种工作让人感到厌烦，程序员就会开始不小心，甚至不检查，最终造成程式未能完善处理例外。另一方面，做好这类检查工作，会让程式变得难以看懂，因为太多错误处理会让程式码显得杂乱。

Ruby 就如许多现代语言一样，能以区隔的 (compartmentalized) 方式处理例外的程式码区块，效果出众而且不会对阅读程式码造成困难。标示有 begin 的程式码区块执行后若遇到例外，就将控制转移至标示有 rescue 的错误处理程式码区块 (block of error handling code)。若没有出现例外，就不会使用 rescue 程式码。以下方法会传回文字档的第一行，若没有例外的话则传回 nil：

def first\_line( filename )  
  begin  
    file = open("some\_file")  
    info = file.gets  
    file.close  
    info  # 最后计算的内容为传回值  
  rescue # 相当于catch  
    nil   # 无法读取档案？那就不传回字串  
  end  
end

有时候，我们需要能够变通地处理问题。例如：如果无法找到需要的档案，可尝试使用标准输入 (standard input)：

begin  
  file = open("some\_file")  
rescue  
  file = STDIN  
end  
  
begin  
  # ... 处理输入 ...  
rescue  
  # ... 并在此处理其他例外。  
end

retry 可用于 rescue 程式码，以重新开始 begin 程式码。现在把刚才的例子编写得精简一点吧：

fname = "some\_file"  
begin  
  file = open(fname)  
  # ... 处理输入 ...  
rescue  
  fname = "STDIN"  
  retry  
end

但是，这里有个错误， 一个不存在的档案会让这个程式码无限回圈。使用 retry 处理例外时，一定要注意这种陷阱。

在所有的 Ruby 函式库中，如果出现例外错误，就会唤起 (raise) 例外。你也可以在自己的程式码中唤起例外。若要唤起例外，可以使用 raise。这只需要一个引数，即描述该例外的字串。可选择使用甚么引数，但不得省略， 之后可使用特别的全域变量 $! 存取。

ruby> **raise "test error"**  
**test error**  
ruby> **begin**  
    |**raise "test2"**  
    | **rescue**  
    |**puts "An error occurred: #{$!}"**  
    | **end**  
An error occurred: test2  
**nil**

## 确认ensure (类似finaly)

方法完成工作后，可能需要作出一些清理动作， 像是关闭打开的档案；清理缓冲的资料等。如果每个方法都只有一个退出点 (exit point)，我们就可以把清理的程式码放于一个地方，也能确定它会执行工作。但是，一个方法可能从不同地方传回，或因为例外，使我们的清理程式码意外跳 过 (skip)。

begin  
  file = open("/tmp/some\_file", "w")  
  # ... 写入档案 ...  
  file.close  
end

上述例子中，如果在写入档案的程式码部分出现例外，该档案就会保持打开。而我们并不想依靠这种冗余 (redundancy)：

begin  
  file = open("/tmp/some\_file", "w")  
  # ... 写入档案 ...  
  file.close  
rescue  
  file.close  
  fail # 唤起一个例外  
end

这很累赘，而且我们要处理所有 return 及 break，所以一旦程式码变得复杂，就会难以控制。

因此，我们把 ensure 这个关键词加入 "begin...rescue...end" 方案 (scheme)。无论 begin 区块成功或失败，ensure 程式码区块都会执行。

begin  
  file = open("/tmp/some\_file", "w")  
  # ... 写入档案 ...  
rescue  
  # ... 处理例外 ...  
ensure  
  file.close   # ... 这总是会发生的。  
end

可以在不使用 rescue 的情况下使用 ensure，反之亦然，但如果在相同的 begin...end 区块中使用，rescue 必须写于 ensure 之前。

# 存取器(getter、setter方法)

### 甚么是存取器？

我们曾在之前的章节讨论过实例变量，但尚未深入探讨。物件的实例变量即是物件的属性 (attribute)，用以区分出同类别的其他物件。能够读写这些属性非常重要，若要读写便需要称为属性存取器 (attribute accessors) 的公开方法（编注：所有实例变量都不公开的，除非透过公开方法存取)。我们等一下就会看到，并不是每次都需要明确地写出存取器方法，现在先看一下所有的步骤。两种存取器分别是写入器 (writer) 及阅读器 (reader)。

ruby> **class Fruit**  
    |**def set\_kind(k)  # 写入器**  
    |**@kind = k**  
    |**end**  
    |**def get\_kind     # 阅读器**  
    |**@kind**  
    |**end**  
    | **end**  
**nil**  
ruby> **f1 = Fruit.new**  
**#<Fruit:0xfd7e7c8c>**  
ruby> **f1.set\_kind("peach")  # 使用该写入器**  
**"peach"**  
ruby> **f1.get\_kind           # 使用该阅读器**  
**"peach"**  
ruby> **f1                    # 检验该物件**  
**#<Fruit:0xfd7e7c8c @kind="peach">**

很简单吧，我们可以储存取回想查看的水果资料， 但方法名称有点冗长。以下例子就比较精简方便：

ruby> **class Fruit**  
    |**def kind=(k)**  
    |**@kind = k**  
    |**end**  
    |**def kind**  
    |**@kind**  
    |**end**  
    | **end**  
**nil**  
ruby> **f2 = Fruit.new**  
**#<Fruit:0xfd7e7c8c>**  
ruby> **f2.kind = "banana"**  
**"banana"**  
ruby> **f2.kind**  
**"banana"**

### inspect 方法(相当于toString()方法)

先离题一下， 现在你会注意到，我们要直接查看物件时，就会出现 #<anObject:0x83678> 这个难懂的讯息。这只是个预设的行为，我们可以随意变更。只要加上称为 inspect 的方法， 就会传回以合理方式描述物件的字串，包括物件部分或所有实例变量的状态。

ruby> **class Fruit**  
    |**def inspect**  
    |**"a fruit of the #{@kind} variety"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f2**  
**"a fruit of the banana variety"**

一个相关的方法称为 to\_s （转换为字串），于输出物件时使用。一般来说，你可以将 inspect 想像为编写程式并用以除错的工具，而 to\_s 则是重新定义程式输出的方式。eval.rb 显示结果时就会使用 inspect。你可使用 p 方法，轻易地从程式取出除错输出。

# 这两行是相等的：  
p anObject  
puts anObject.inspect

### 简单使用存取器

因为不少实例变量都需要存取器方法，因此 Ruby 提供了一些快捷的方式。

|  |  |
| --- | --- |
| 快捷方式 | 效果 |
| attr\_reader :v | def v; @v; end |
| attr\_writer :v | def v=(value); @v=value; end |
| attr\_accessor :v | attr\_reader :v; attr\_writer :v |
| attr\_accessor :v, :w | attr\_accessor :v; attr\_accessor :w |

让我们善用这些方式加到程式里吧。首先我们需要一个自动产生出来的阅读器及写入器，然后加入新的资讯至 inspect：

ruby> **class Fruit**  
    |**attr\_accessor :condition**  
    |**def inspect**  
    |**"a #{@condition} #{@kind}"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f2.condition = "ripe"**  
**"ripe"**  
ruby> **f2**  
**"a ripe banana"**

### 更多水果的乐趣

如果没有人吃成熟的水果，那就让时间来解决吧。

ruby> **class Fruit**  
    |**def time\_passes**  
    |**@condition = "rotting"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f2**  
**"a ripe banana"**  
ruby> **f2.time\_passes**  
**"rotting"**  
ruby> **f2**  
**"a rotting banana"**

开始前，我们先要说明一个问题。如果我们现在创造第三件水果，那会怎么样呢？要记住，指派值予实例变量后，实例变量才会存在。

ruby> **f3 = Fruit.new**  
**ERR: failed to convert nil into String**

这是因为 inspect 方法发出问题，而且问题很明显。我们要求它回报水果的种类及状态，但是 f3 还未获指派任何属性。如果我们想的话，可以重新编写 inspect 方法，让它可以用 defined? 方法测试实例变量，并只在实例变量存在时才回报，但这样就可能不是很实用，因为每种水果都有他们的种类与状态，我们就要确保它们都已经定义。这将在下一节中讨论。

# 物件初始化

上一节中的水果 (Fruit) 类别具有两个实例变量，一个用来描述水果的种类，一个用来描述水果的状态。为该类别编写了自订的 inspect 方法后，我们才发现如果水果没有这些特征，就会显得很不合理。幸好 Ruby 有方法能够确保实例变量总是能够初始化 (initialize)。

### initialize 方法

每当 Ruby 建立一个新物件，就会寻找 initialize 方法，然后执行该方法。我们可以做一项很简单的事，就是使用 initialize 方法将预设值加入所有实例变量，这样 inspect 方法就有东西可以回报。

ruby> **class Fruit**  
    |**def initialize**  
    |**@kind = "apple"**  
    |**@condition = "ripe"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f4 = Fruit.new**  
**"a ripe apple"**

### 将假设 (assumption) 改为需求 (requirement)

有时候，预设值并不大合理， 例如有“水果预设种类”这种东西吗？比较理想的是，要求创造水果时，即指定每件水果的种类。因此，我们要在 initialize 方法中加入一个形式引数 (formal argument)， 供应至 new 的引数其实都传递至 initialize。

ruby> **class Fruit**  
    |**def initialize( k )**  
    |**@kind = k**  
    |**@condition = "ripe"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f5 = Fruit.new "mango"**  
**"a ripe mango"**  
ruby> **f6 = Fruit.new**  
**ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)**

### 弹性初始化

从以上例子我们发现，引数与 initialize 方法关联后，如果省略引述会产生错误。如果想要周到一点，我们可以设定一个预设值。

ruby> **class Fruit**  
    |**def initialize( k="apple" )**  
    |**@kind = k**  
    |**@condition = "ripe"**  
    |**end**  
    | **end**  
**nil**  
ruby> **f5 = Fruit.new "mango"**  
**"a ripe mango"**  
ruby> **f6 = Fruit.new**  
**"a ripe apple"**

除了 initialize 以外，你也可于其他方法使用预设引数值。要注意的是，具有预设值的引数必须排在引数列的最后面。

为物件提供不同的初始化方式，有时候会颇有用处。本手册并不包含物件反射 (object reflection) 及变量长度引数表 (variable-length argument list)的作法，但 Ruby 支援这些功能，能够提供有效的方法重载 (method overloading)。

# 基本要素

本节主要讨论一些实作事项。

### 叙述分隔符号 (statement delimiter)

有些语言需要某些标点符号（例如分号 (;)）来结束程式中的每个叙述， 而 Ruby 则是根据 shell 的传统，例如 sh 及 csh。同一行上的多项叙述必须以分号来分隔，但并不需要在该行结尾处使用，换行字元 (linefeed) 在此的作用与分号相同。如果该行以反斜线 (\) 结尾，则可忽略之后的换行字元，如此能让单一逻辑程式叙述行 (logical line) 横跨数行。

### 注解 (comments)

为甚么要写注解？虽然程式码编写得清楚明瞭就足够，但有时候在空白处随便写下些注解也会很有用，不应该认为别人看到你的程式码，就会立即知道你编写 的方法。此外，实际上，今天的你与明天的你也可能判若两人，经过一段时间，当我们回头修改或改进程式的时候，可能会说：“我知道我写了这个，但到底写来干 嘛呢？”

有些经验丰富的程序员会准确地指出，矛盾或过时的注解不如不要。没错，注解不应用来替代清楚易读的程式码；而如果你的程式码不清不楚，那就有可能存在错误。初学 Ruby 时，你可能需要写较多的注解，不过当你能够以简雅易读的程式码表达时，注解也就会越来越少。

Ruby 依循常见的脚本传统，使用井号 (#) 来代表开始写注解。直译器会忽略任何以 # 开头的内容，直至该行结束。

此外，为了方便篇幅较长的注解，Ruby 直译器也会忽略从 "=begin" 到 "=end" 之间的任何内容。

#!/usr/bin/env ruby  
  
=begin  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  这是注解区块，你可在此编写注解，  
  方便读者（包括你自己）阅读程式码。直译器会忽略这些内容，  
  所以并不需要在每行都加上 #。  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
=end

### 组织你的程式码

Ruby 具有高层级的动态性 (dynamism)，指的是类别、模组、方法只会在定义它们的程式码运作后才存在。如果你习惯使用较静态的语言编写程式的话，有时候可能会出现惊喜。

# 以下结果会导致 "undefined method"（未定义方法）错误：  
  
puts successor(3)  
  
def successor(x)  
  x + 1  
end

虽然直译器执行前，会检查整份脚本文件的语法，但 def successor ... end 程式码需要真的运作，才能建立 successor 方法。因此，排列脚本的次序将是关键。

这看起来好像严格规定要以从下而上的方式组织程式码，但其实并不是。当直译器遇上方法定义，只要你确定该方法真的被呼叫时能够获得定义，直译器就能安全的包含未定义的参考 (references)：

# 转换华氏为摄氏，  
# 分为两个步骤。  
  
def f\_to\_c(f)  
  scale(f - 32.0)  # 虽然这是向前参考 (forward reference)，但没有问题。  
end  
  
def scale(x)  
  x \* 5.0 / 9.0  
end  
  
printf "%.1f is a comfortable temperature.\n", f\_to\_c(72.3)

比起 Perl 或 Java，这看起来好像没那么方便，但比起不使用原型 (prototype) 下编写 C，则来得自由（C 总是会要求你保持甚么参考甚么的偏序 (partial ordering)）。将上层 (top-level) 程式码置于原始档案 (source file) 底层，总是能够运作， 这虽然起初看来是让人觉得有点烦恼。合理而轻松来强制执行所希望行为的方式是，在档案顶层定义一个 main 函数，然后从底层呼叫。

#!/usr/bin/env ruby  
  
def main  
  # 在此表达上层逻辑 ...  
end  
  
# ... 在此输入支援程式码，并且适当组织 ...  
  
main # ... 并在此开始执行。

这也有助于 Ruby 提供工具，将复杂的程式断为易读的区块，既可重复使用又逻辑相关。我们已使用过 include 来存取模组， 你之后也会发现 load 及 require 命令 (facility) 很实用。load 在所指示的档案被复制、贴上时就会运作（类似 C 的 #include 前置处理指令 (preprocessor directive)）。require 则比较复杂，可以让程式码在需要时，最多载入一次。

### 总结

这份手册应该足够让你开始用 Ruby 编写程式了。如果有进一步的问题，可以前往使用者社群求助，或者参考越来越多的参考书或网上资源。

祝你好运，享受编写乐趣！

# 关于本手册

### 关于本手册

本入门手册转载于不同地方，提供不同语言版本。目前，最新的英文版本保存于 [rubyist.net](http://www.rubyist.net/%7Eslagell/ruby/)。若您发现较旧的版本，请通知网站管理员该版本所在的网站。

### 文件历史

* 日文原作由 [Matz](mailto:matz@ruby-lang.org) 所著。
* 首份英文版本由 [GOTO Kentaro](mailto:gotoken@notwork.org) 及 Julian Fondren 翻译。
* [Mark Slagell](mailto:slagell@ruby-lang.org) 重新翻译并加入新材料。
* 本中文版由 [Ruby Taiwan](http://ruby.tw) 社群发行：由 Mason Chang 赞助初稿翻译费用、Lok 完成初稿翻译、[ihower](http://ihower.tw) 技术编辑。