# 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。