RUBY语言入门教程

Version 1.0

编著　张开川

kaichuan_zhang@126.com

开源文档

目录

自序 3
第一章　概述..4
1.1　编程语言的发展简史..4
1.2　编程其实很容易..7
1.3　Ruby的由来..8
1.4　Ruby的特性..9
第二章 11
2.1　下载Ruby 1.8.5..11
2.2　安装Ruby 1.8.5 for Windows..12
2.3　开始第一个小程序..14
2.4　Ruby语言的集成开发环境..17
第三章　语法快览..18
3.1　注释与分行..18
3.2　分隔符..20
3.3　关键字..21
3.4　运算符..23
3.5　标识名和变量的作用域..25
3.6　类库..26
3.7　数据类型..27
3.8　赋值和条件运算符..28
3.9　条件判断语句..30
一. 单行 if（如果） 语句..30
二. 多行 if 语句..30
三. unless(除非) 条件语句：..30
四. case分支条件语句..31
3.10　循环语句..32
一. while（当…） 循环..32
二. 单行 while 循环..32
三. until（直到…）　循环..32
四. for…in 循环..33
五．break ， next & redo & retry..33
六. 求 50 以内的素数。..36
七. times , upto , downto , each ,step..37
3.11　异常与线程..38
3.12　一个综合小练习..39
第四章　一切都是对象..40
4.1　两种思维方式..40
4.2　对象..41
4.3　封装..43

4.4 继承......45
4.5 多态......47
第五章 48
5.1 为什么是Ruby？ 48
5.2 Ruby会长久么？ 49
5.3 详解变量—— 动态类型......51
5.4 蓝图可以改变吗？ ——动态语言......55
5.5 一些编码建议......57
第六章 深入面象对象......61
6.1 重载？重写......61
6.2 增强父类方法......65
6.3 实例变量、类变量、类方法......67
6.4 单例方法......71
6.5 访问控制......73
第七章 79
7.1 模块......79
7.2 命名空间......80
7.3 糅和(Mix-in) 与多重继承 83
7.4 require 和 load 86
第八章 88
8.1 再说数组......88
8.2 再说字符串......93
8.3 正则表达式......97
8.4 迭代器、代码块、闭包......103
第九章 元编程......109
小跋 112

自序

其它编程语言的学习都主要来自于书本，而 Ruby 的学习却是完全经由网络。我想，回报网络的最好方式就是在网络上还没有 Ruby 中文书籍的时候，编著一本 Ruby 中文入门教材。感谢编程语言发展史上的前辈们；感谢网络论坛上程序员们或是理智，或是激烈的讨论；感谢一切看到这本书的人。

曾经经受了 SCJP 的挖掘、挖掘、再挖掘（基于 1.4，还没有 Java 5，Java 6 的许多特性），初遇 Ruby，觉得十分亲切，仿佛童年时得到一个新奇的玩具。把玩之后，才发现玩具的塑料外壳里，藏着一把瑞士军刀。自此，模式、框架常常变得多余，很多时候可以更直接。好比在量子时代，星际航行只是一次时空转换而已，航天飞机静静地躺在博物馆里，那是旧时代科学的极致代表。

从物理课本中，我们感受到爱因斯坦的伟大，但对牛顿则怀以更加崇敬的心情。身体终将消逝，而你，我，他——我们的意识却将在网络中延续。旧时代文明的延续依赖于纸质书籍，书籍传递了理性之光。也许直觉才是这个宇宙本体的最初相用，直觉是一种天赋，我无从把握，但是理性，如此真切实在，她照亮了我，照亮了你，直至未来。

思，亘古如斯又倏忽闪现，谁的惊愕能深究它。

——海德格尔

张开川

2006 年 12 月 31 日

第一章　概述

1.1　编程语言的发展简史

本小节是一些编程语言的简单介绍，你如果不感兴趣，可以略过不看。

先说程序是什么？程序就是指令序列的有序集合。指令即代码，可以是数字，也可以是文字，文字最终要转换成数字。也就是说，程序是许多数字串，当然，也可以合并成一个很长很长的数字串。

程序的作用是什么？程序能够做事，做你想叫它做的事。换一种说法，程序的功能是完成它的使命。它的使命由编写程序的人来决定，或者由编写程序的程序来决定。

编写程序就是写文章，写能够转换成很长很长数字串的文章，给计算机看。

最初的计算机编程语言是一长串二进制代码。所谓的二进制就是只有 1 和 0，所以第一代的计算机编程语言就象下面的许多 1 和 0。

1001101001010011

0100011101111110

1010101000010100

11110100001010001

……

这是给计算机看的，你看不懂，我也看不懂，程序员自己下次再看的时候，也

是搞不清。但是，这是最基本的。直到现在，计算机的中央处理器（CPU）还是只认 1 和 0 组成的二进制代码。

第二代的计算机编程语言叫汇编语言，就象这样：

```
LD   x,37H
MOV a,x
……
```

程序员之间能够很容易地交流，但是这样的语言编写效率很低，而且不同的 CPU 有不同的指令集，根本无法重复使用。

第三代的计算机编程语言称之为高级语言。容易编写，容易阅读，容易交流，而且可以与 CPU、操作系统无关。从 1958 年的 LISP 和 1957 年的 FORTRAN 开始到现在，我们接触到的编程语言几乎都是第三代语言。其中又有两大类。一类叫函数式语言，如：LISP，Haskell。一类叫命令式语言，如：FORTRAN，Pascal，C++，Java。

第四代的计算机编程语言，你只要告诉它你要什么，无须告诉它怎么做，它就能把答案给你。上个世纪八十年代末发展起来的数据库查询语言（SQL）就是一个例子。第四代的编程语言还在起步阶段，智能化程度还有待于加强。SQL 是当今最为强大的数据检索机制之一；SQL 并不能完成一个过程语言所能完成的所有任务。

需要说明的是，面向对象的编程语言并不是第四代语言。面向对象是一种认识事物的方式、理念，你可以面向对象地编程（OOP），也可以面向对象地设计（OOD）。面向对象的编程语言是过程语言的自然延续，同属于第三代命令式语言。

大家平常所说的编程语言是指第三代的计算机编程语言。有的语言是解释执行，如 Python，Ruby，交互性很好；有的语言是编译执行，如 Pascal, C，速度较快。 有的语言是本地执行，如 C，C++；有的语言是虚拟机执行，如 Java, C#。

有的语言是动态语言，如 JavaScript，Ruby；有的语言是静态语言，如 C++，Java。

1.2 编程其实很容易

一门语言包含三个方面：语义，语法和语用。汉语，英语是如此，计算机编程语言也是如此。

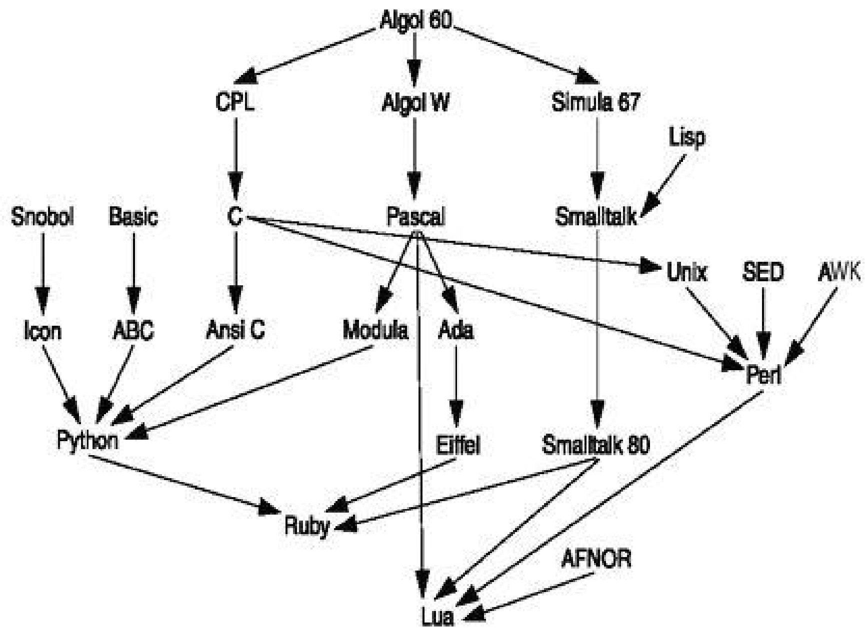
从语义方面来看，人类语言的单词量非常大，要以万计，常用单词也有几千；计算机编程语言的基本单词只有几十个，我们称之为关键字。

学习一门计算机编程语言只要搞清楚这几十个关键字的意思、用法，就大致及格了，六十分到手了。如果要灵活应用一门计算机编程语言，就必须在反复使用的过程中去不断加强理解，不断加深体会。如果碰上好的教材，遇到好的老师，那么入门拿到六十分是很容易的。要想拿到八十分、九十分，就看各人修行了，所谓拳不离手，曲不离口。多多练习，“无它，但手熟尔”。

Ruby 是一种动态语言，语法简单，就象当年的 BASIC 一样，非常容易上手，适合初学者。对于编程高手来说，Ruby 里有许多深奥之处，等待你去发掘。

1.3 Ruby 的由来

松本行弘（Matz）是日本一家开源软件公司的程序员，有15年的编程经验。在工作中，他希望有一种比 Perl 强大，比 Python 更面向对象的语言。从1993年2月，他开始设计一个全新的自己的语言，1994年12月发布了第一个 alpha 版本，并且将这种新语言定名为Ruby（红宝石）。发展到现在，最新版本是Ruby 1.8.5(2006-8-25)。



1.4 Ruby 的特性

计算机编程语言的发展总是与飞速变化的世界息息相关的，Ruby 是为了适应变化、提高和完善编程艺术而出现的。

- **完全开源**
- **多平台**　Ruby可以运行在 Linux， UNIX， Windows， MS-DOS，BeOS，OS/2…
- **多线程**　线程就是指能在一个程序中处理若干控制流的功能。与 OS 提供的进程不同的是，线程可以共享内存空间。
- **完全面向对象**
- **不需要内存管理**　具有垃圾回收（Garbage Collect，GC）功能，能自动回收不再使用的对象。
- **解释执行**　其程序无需编译即可轻松执行。
- **功能强大的字符串操作 / 正则表达式**
- **具有异常处理功能**
- **可以直接访问OS**　Ruby可以使用（UNIX的）绝大部分的系统调用。单独使用Ruby也可以进行系统编程。
- **动态类型语言**　Ruby的变量没有类型，因此不必为类型匹配而烦恼。
- **动态语言**　程序运行中，可以新加入属性，行为，也可以重写方法。
- **支持操作符重写**
- **支持无限精度的数字**　例如计算400的阶乘也轻而易举。
- **丰富的库函数**
- **用模块进行混合插入（Mix-in）**　Ruby舍弃了多重继承，但拥有混合插入功能。使用模块来超越类的界限来共享数据和方法等。
- **语法简单**　它是脚本语言，没有指针，学习曲线比较低。.

……

Ruby 吸取了 perl 的正则表达式， python 的简单性可读性，smalltalk 的纯面向对象语法和单继承，LISP 的无穷嵌套的语法，Java的线程…

第二章

你如果只是想了解 Ruby 语言，不准备实践，请跳过本章，从第三章开始我们的 Ruby 语言之旅。

2.1 下载 Ruby 1.8.5

首先下载 Ruby 1.8.5(2006-8-25) for Windows：

在网页**http://www.rubychina.net/downloads/** 上找到

--〉Ruby on Windows

--〉Ruby 1.8.5 One-Click Installer Stable version (*recommended*)

点击鼠标右键，另存为…，存入你的本地硬盘，这就完成了下载。

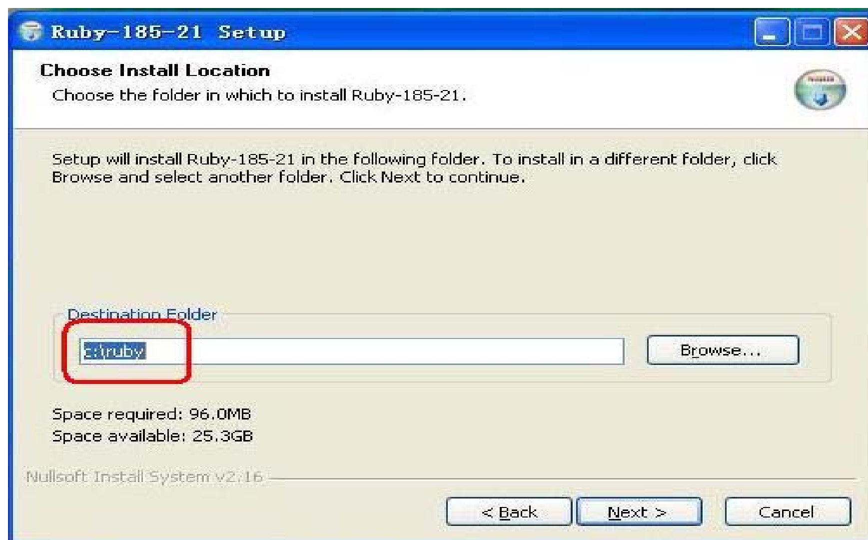
2.2 安装 Ruby 1.8.5 for Windows

运行下载好的文件 ruby185-21 ，出现安装向导界面，

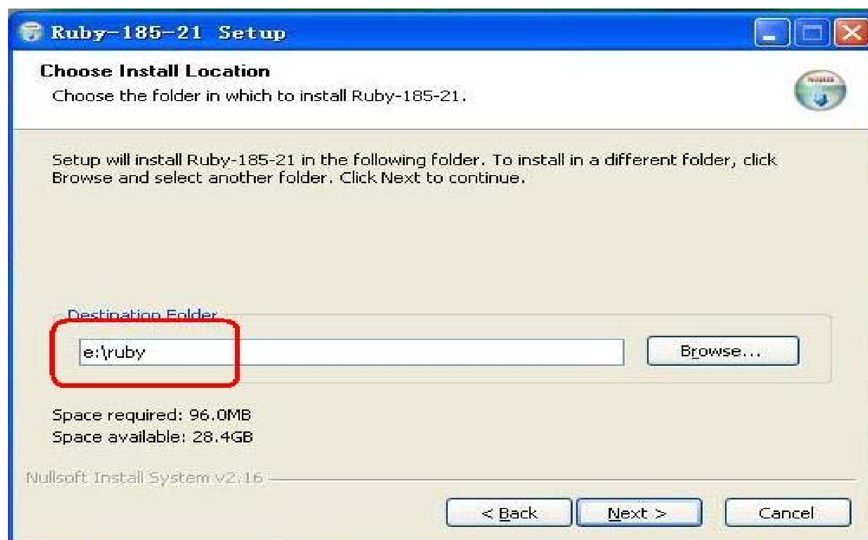
点击 next;

点击 I Agree;

点击 next; 出现如下的 选择安装位置 界面，



改变你想安装 Ruby 的路径，我选择了 e: 盘；



点击 next;

点击 Install;

点击 next;

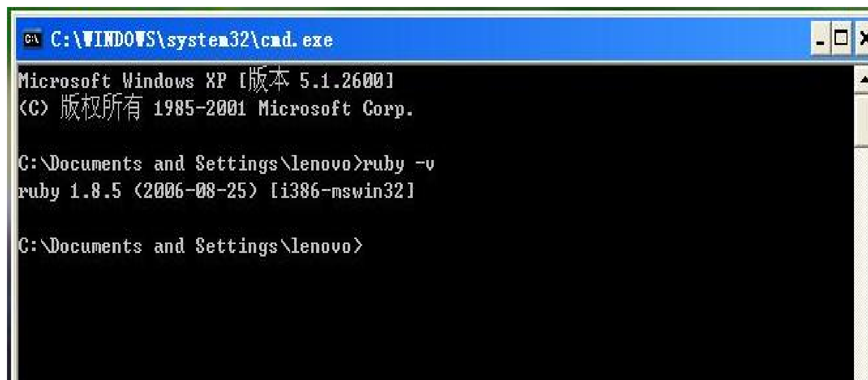
点击 Finish,一切 OK，安装完成。

2.3　开始第一个小程序

从 Windows XP 的开始—〉运行 —〉输入 cmd，打开一个 DOS 窗口；

1. 版本

输入 ruby –v　，



屏幕显示了版本号。

2.开始第一个小程序，打印“hello,world”

第一种方式：

输入 ruby –e ‘print “hello,world” ’ ，

ruby　　　　的意思：运行这个 ruby 语言程序；

-e　　　　　的意思：把后面的一行脚本作为一个 ruby 程序；

print　　　　的意思：打印；

hello,world　的意思：这是我们要输出的内容。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\lenovo>ruby -v
ruby 1.8.5 (2006-08-25) [i386-mswin32]

C:\Documents and Settings\lenovo>ruby -e 'print "hello world"'
hello world
C:\Documents and Settings\lenovo>
```

第二种方式：

Ruby 语言自带了一个交互式的编程环境 irb，这是一个 shell 窗口。在 e:\ruby ,

输入： irb ；

输入： print "hello world" ；

输入： print "中文 world" ；

如果要退出 irb 交互式环境，输入： exit 。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\lenovo>e:

E:\>cd ruby

E:\ruby>irb
irb(main):001:0> print "hello world"
hello world=> nil
irb(main):002:0> print "中文hello"
中文hello=> nil
irb(main):003:0> exit

E:\ruby>_
```

第三种方式：

你如果觉得上面的方式太繁琐，请在 e:\ruby 下 ，输入下面图示内容：

copy con hello.rb

print “hello world”

print “ 3*7=”,3*7

^Z

ruby hello.rb

2.4 Ruby 语言的集成开发环境

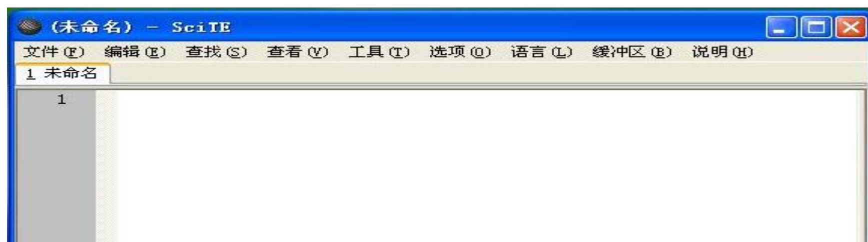
上一节第三种方式编写代码与运行程序分开操作，你很不习惯。这一节，我们试着建立 Ruby 语言的集成开发环境。

1. 使用 Eclipse 开发 Ruby 应用程序　　Eclipse 是一个功能强大的跨平台集成开发环境，支持对 Java，jsp，php 等的开发，若是开发 Ruby 应用程序，需要下载并配置 RDT (Ruby Development Tools)，一组 Eclipse 插件。 运行 Eclipse 需要 jre。编程语言的初学者，不熟悉 Java，所以我不详细介绍 Eclipse + RDT 的搭建。如果你是一个 Java 程序员，下载并配置 RDT 是很 easy 的。

2. 使用 Ruby 自带的集成开发环境　　Ruby 1.8.5 for Windows 自带了一个集成开发环境 FreeRIDE，和一个代码编辑工具 SciTE 。后面的程序我们将使用 SciTE 编辑并运行 Ruby 程序。

点击 Windows XP 的开始 —〉所有程序 —〉Ruby-185-21 —〉SciTE ,

打开的窗口是英文，关闭它。去 SciTE 的网站下载一个文件 locale.zh_gb.properties （10k)，改名为 locale.properties 复制到 E:\ruby\scite\ 下，重新打开 SciTE ，



OK，都是中文了。今后，你在 SciTE 中写好代码，保存为 .rb 文件（文件扩展名必须是 rb ）。选择菜单栏上的工具 —〉 运行，或者按下键盘 F5 键，就看到了执行结果。

第三章　语法快览

第二章的内容可以跳过不看，本章的内容虽然不需要熟记，但是应该建立一个大致印象，为后面章节的内容打下基础。

3.1　注释与分行

Ruby 中的注释有单行与多行两种，先看一个程序 E3.1-1.rb：

```
#    E3.1-1.rb      从#开始到行尾是单行注释
puts   3/5
puts   3/5.0
=begin
  puts   6/5
  puts   6/5.0
  多行注释可以用=begin   和   =end  ；
  实际上，这也是 Ruby 的内嵌文档（Rdoc）注释，类似 javadoc   ，
  可以用命令 r i 从源文件生产文档。
=end
```

运行结果：

```
>ruby E3.1-1.rb
0
```

```
0.6
>Exit code: 0
```

Rdoc 是内嵌在 ruby 代码之中的，可以转换为 html 文档说明。类似 javadoc。ri 是一个命令行程序，用来查看函数说明、类说明。函数说明、类说明应该放置在 =begin 和 =end 之中。“=begin”一定要写在行首，也就是说，这一行的前六个字符是“=begin”，不允许有空格在这之前。

Ruby 中用分号“ ； ”来表示一个语句的结束。一行如果有多个语句，每个语句用分号隔开，而最后一个语句可以省略分号。换行符表示一行结束。如果语句太长，可以用“ \ ”连接下一行。看程序 E3.1-2.rb：

```
#    E3.1-2.rb      演示分行
puts   3/5 ; puts   3/5.0
puts   "这里演示" \
"连行"
```

运行结果：

```
>ruby E3.1-2.rb
0
0.6
这里演示连行
>Exit code: 0
```

3.2　分隔符

关键字、运算符、分隔符一起构成了一门编程语言的基本定义。3.2 、3.3 、3.4 节分别对分隔符、关键字、运算符作一些介绍。如果有不详细的地方，将在本书后面部分介绍；或者是因为这些不常用到；还有一种情况是：可以被其它常用语法定义代替。

Ruby 中的常用分隔符如下：

符号	名称	用途
;	分号	用来分隔一行中的多个语句
（）	圆括号	提高优先级；定义方法时容纳参数列表
	空格	分隔字符；在可省略（）的地方，代替（）
,	逗号	隔开多个参数
.	点	将对象与它的方法隔开
::	紧连的两个冒号	域作用符， 将模块（类）与它的常量隔开

3.3 关键字

Ruby 中的关键字如下：

模块定义：module

类定义： class

方法定义：def ， undef

检查类型：defined?

条件语句：if ， then ， else ， elsif ，case ，when ，unless

循环语句：for ，in ，while ，until ，next， break ，do ，

redo ，retry ，yield

逻辑判断：not ，and ，or

逻辑值和空值：true ，false ， nil

异常处理：rescue ，ensure

对象引用：super ，self

块的起始：begin/end

嵌入模块：BEGIN ，END

文件相关：__FILE__ ， __LINE__

方法返回：return

别名： alias

BEGIN 模块相当于 C 语言中的宏， END 模块用来作一些收尾工作。有了 require，include，应该取消 BEGIN 和 END 的语法定义。

3.4　运算符

Ruby 中的运算符如下：

优先级	能否重写	运算符	描述
最高	Y	[]　[]=	数组下标　　数组元素赋值
	Y	**	乘冥
	Y	!　~　+　-	非　位非　一元加　负号
	Y	*　/　%	乘　除　模
	Y	+　-	加　减
	Y	>>　<<	右移　　左移
	Y	&	位与
	Y	^ |	位异或　　位或
	Y	<=　<　>　>=	小于等于　　小于　　大于 大于等于
	Y	<=>　==　=== =~　!=　!~	各种相等判断（!=　!~ 不能重写）
		&&	短路与
		||	短路或
		..　...	区间的开始点到结束点
		? :	三元条件运算符
		=　%=　~=　/=　-= +=　|=　&=　>>= <<=　*=　&&=　||= **=	各种赋值 例如：a = 5; b += 3(意思是：b = b+3)；
		defined?	检查类型
		not	逻辑非
		or　and	逻辑或　逻辑与

		if　unless while　until	判断与循环
最低		begin/end	定义方法、类、模块的范围

3.5 标识名和变量的作用域

Ruby 的标识名用来指向常量，变量，方法，类和模块。标识名的首字符用来帮助我们确定标识所指向内容的作用域。一些标识名，就是上面所示的关键字，不能用来当作常量，变量，方法，类或模块的名字。

Ruby 的标识名区分大小写。

Ruby 使用一个约定来帮助它区别一个名字的用法：名字前面的第一个字符表明这个名字的用法。局部变量、方法参数和方法名称应该用一个小写字母开头或者用一个下划线开头；全局变量用美元符作为前缀 $；而实例变量用 @ 开头；类变量用 @@ 开头；类名、模块名和常量应该用大写字母开头。

词首字母后面可以是字母、数字和下划线的任意组合； @ 后面不可以直接跟数字。

Ruby 程序代码现在是用7位ACSII 码来表示，通过语言扩展来支持 EUC, SJIS 或 UTF-8 等8位编码系统。Ruby 2.0 版本将支持16位的 Unicode 编码。

3.6　类库

Ruby 像 C++ 一样，有许多类库可以供你使用，其中的 I/O 库很完善。前面我们使用了 puts 和 print ，你一定注意到它们并不是关键字，为什么能够直接使用？

一门编程语言，A)关键字可以直接使用；B)还有其它常用的应用程序，我们将它们放在一个专门的目录下，称为类库（许多类的仓库）。如果当前程序要用到类库中某个程序已经定义好的类、方法，就应该使用 require 或者 include 将类库程序名包含在当前程序中；C)从父类继承得到的方法可以直接使用。

I/O 就是 输入/输出，这是 Ruby 语言 Kernel 模块的方法，Mix-in 在根类 Object 中的。

puts 把它的所有参数写出来，每一个参数结束都加入一个换行符， print 也写出它的参数，不过没有换行。你可以指明输出到一个文件，不说明，则一般输出到显示器。

还有一个常用的输出方法是 printf，它按格式输出参数。

```
printf "Number: %4.3f, String: %s", 7.8, "hi!"
```

运行结果：

```
Number:   7.800, String: hi!
```

这个例子中，格式字符串“Number: %4.3f, String: %s”告诉 printf 用一个浮点数（总共允许4位，小数点后3位）和一个字符串来代替。 printf 和 print 一样，不主动换行，换行可以用“\n”。

介绍了三个输出方法，再介绍一个输入方法：gets ，它从你的程序的标准输入流中返回一行。一般用来从键盘或文件读入一行数据。

3.7　数据类型

Ruby 数据类型有数字，字符串，数组，哈希表，区间，正则表达式。

数字分为整数型（1，0，75 ，1e3），浮点型（2.4 ，7.0 ，0.99）。浮点型数据小数点后必须跟数字（ 1.e3 不可以，1.1e3可以）。数字可以有前缀：0表示八进制, 0x表示十六进制, 0b表示二进制（0724，0x5AC4，0b11101）。

字符串是在‘ ’（单引号）、“ ”（双引号）之间的代码。

数组的下标从0开始。Ruby的数组和其它语言不同，数组的每个元素可以是不同的类型：[2.4， 99，“thank you”，[a, b ,c] ，78]。

区间：1..5 表示1，2，3，4，5 ；

1...5表示1，2，3，4 。

3.8 赋值和条件运算符

Ruby 基本的赋值用“＝”来完成，就像 E3.8-1.rb 如下示例：（在不产生歧义的地方，我用 # 表示答案）

```
a = 1 ； b =  2 + 3            #a=1 ,b=5
a ,b = b ,a                    #a=5 ,b=1
a = b = 1 + 2 + 3              #a=6 ,b=6
a = (b = 1 + 2) + 3            #a=6 ,b=3
x = 0                          #x=0
a,b,c = x, (x+1), (x+2)        #a=0 ,b=1,c=2
```

Ruby 的条件运算符比 Java 更加复杂，看例子 E3.8-1.rb：

== （等于） ！= （不等于）	比较两个对象的值是否相等 ,返回 true, flase a=1; b=1.0; a==b #true
eql?	比较两个对象的值、类型是否相等,返回 true, flase a=1; b=1.0; a.eql?(b) #flase（a为整数型，b为浮点型）
equal?	比较两个对象在内存中地址是否相同,返回 true, flase a=1.0; b=1.0; a.equal?(b) #flase a=1.0; b=a ; a.equal?(b) # true

<=>	比较两个对象的大小，大于、等于、小于 分别返回1,0,-1 "aab" <=> "acb" # -1 （第二个 a 的 ASCII 码小于 c） [5] <=> [4,9] # 1 （第一个元素 5 > 4）
===	右边的对象是否在左边区间之内,返回 true, flase puts (0..9)=== 3.14 #true puts ('a'..'f')=== 'c' # true
=~（匹配）	用来比较是否符合一个正则表达式,返回模式在字符串中被匹配到的位置，否则返回 nil
!~ (不匹配)	断言不符合一个正则表达式,返回 true, flase
<= < > >=	小于等于 小于 大于 大于等于

3.9 条件判断语句

判断条件是否相等用“==”，注意不要写成“=”。

一．单行 if（如果） 语句

1) if 条件[1] then 语句1; 语句2 ; 语句… end

2) （ 语句1; 语句2 ; 语句… ） if 条件

二．多行 if 语句

```
if 条件
      语句1; 语句2 ; 语句…
elsif 条件
      语句1; 语句2 ; 语句…
else
      语句1; 语句2 ; 语句…
end
```

三. unless(除非) 条件语句：

unless 条件 = if not (条件)

[1] Ruby 里，nil 和 false 为假，其它都为真； puts "is true" if 5 #is true
str="false" ; puts "is true" if str #is true

四. case分支条件语句

看程序 E3.9-1.rb：

```
case　对象
when 可能性1
    语句1; 语句2 ; 语句…
when 可能性2
    语句1; 语句2 ; 语句…
when 可能性…
    语句1; 语句2 ; 语句…
else
    语句1; 语句2 ; 语句…
end
```

例：x=3

```
case　x
  when　1..2
    print "x=",x,";在 1..2中"
  when　4..9, 0
    print "x=",x,";在4..9,0中,或是0"
  else
    print　"x=",x,";其它可能"
end
```

结果：x=3;其它可能

3.10 循环语句

一. while（当…） 循环

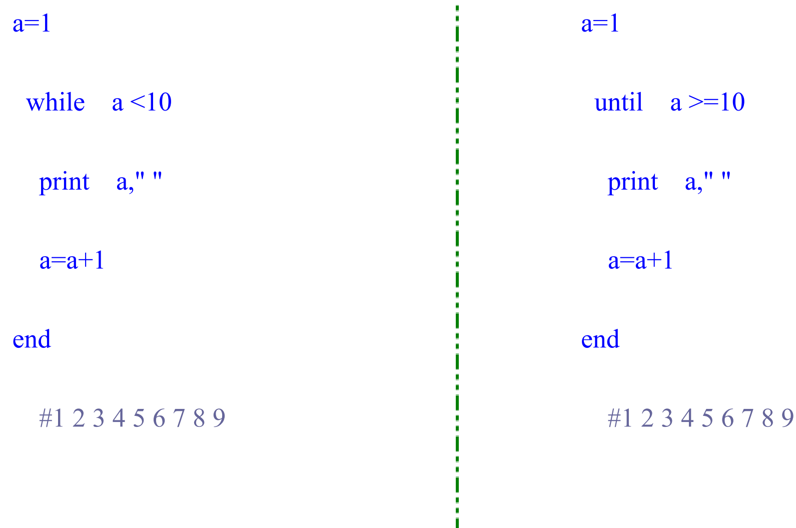
```
while 条件
      语句1; 语句2 ; 语句…
end
```

二. 单行 while 循环

```
（ 语句1; 语句2 ; 语句… ）   while   条件
```

三. until（直到…） 循环

我们想输出数字1到9，看程序 E3.10-1.rb，E3.10-2.rb ：



```
a=1
 while   a <10
  print   a," "
  a=a+1
end
  #1 2 3 4 5 6 7 8 9
```

```
a=1
 until   a >=10
  print   a," "
  a=a+1
end
  #1 2 3 4 5 6 7 8 9
```

一边是 while a <10，一边是 until a >=10，其它代码都一样，结果也一样。

从这两个小程序可以看出： until 条件 = while not (条件)

四. for…in 循环

```
for 变量 in 对象
    语句1; 语句2 ; 语句…
end
```

对象可以是数组，区间，集合…，看程序 E3.10-3.rb：

```
# E3.10-3.rb
for i in 1..9
  print i," "
end
  #1 2 3 4 5 6 7 8 9
```

五．break ， next & redo & retry

在循环体内，如果遇到：

break ，跳出当层循环；

next ，忽略本次循环的剩余部分，开始下一次的循环；

redo ，重新开始循环，还是从这一次开始；

retry ，重头开始这个循环体。

1）.看程序 E3.10-5.rb 。

```
puts "演示break"
c='a'
  for i in 1..4
```

```
      if i == 2 and c =='a'
        c = 'b'
        print "\n"
        break
      end
      print i,c," "
    end
  puts "\n\n"
#演示break
  1a
```

变量 c 在循环体之前赋值为 'a' ，程序执行了 2 次，遇到 break ，退出了本层循环，不再执行第 2 次的print i,c," "。

2）.看程序 E3.10-5.rb 。

```
puts "演示next"
c='a'
  for i in 1..4
    if i == 2 and c =='a'
      c = 'b'
      print "\n"
      next
    end
    print i,c," "
```

```
  end
  puts "\n\n"
#演示next
  1a
  3b 4b
```

变量 c 在循环体之前赋值为 'a' ，程序执行到第 2 次，c 又赋值为 'b'，遇到 next ，退出了本次循环，不再执行第 2 次的print i,c," "，开始第 3 次循环，此时，c = 'b'，i = 3。

3）.看程序 E3.10-5.rb 。

```
puts "演示redo"
c='a'
  for i in 1..4
    if i == 2 and c =='a'
      c = 'b'
      print "\n"
      redo
    end
    print i,c," "
  end
  puts "\n\n"
#演示redo
  1a
  2b 3b 4b
```

变量 c 在循环体之前赋值为 'a'，程序执行到第 2 次，c 又赋值为 'b'，遇到 redo，重新开始循环，还是从这一次开始，此时，c = 'b'，i = 2。

4）.看程序 E3.10-5.rb 。

```
puts "演示retry"
c='a'
  for i in 1..4
    if i == 2 and c =='a'
      c = 'b'
      print "\n"
      retry
    end
    print i,c," "
  end
 puts "\n\n"
#演示retry
  1a
  1b 2b 3b 4b
```

变量 c 在循环体之前赋值为 'a'，程序执行到第 2 次，c 又赋值为 'b'，遇到 retry，重头开始这个循环体，此时，c = 'b'，i = 1。

六. 求50以内的素数。

```
#   E3.10-4.rb            求50以内的素数

for  i  in  2..50         #50以内
  f=true                  #起始假定每个数都是素数
  for  p  in  2...i       #比自身小的正整数（1和自身除外）
    if  i%p==0            #如果能整除
      f=!f                #那么这个数不是素数
      break               #并且跳出这层循环
    end                   # if 结束
  end                     #内层循环结束
  print  i," "  if  f     #如果这个数保持起始假定，则打印
end                       #外层循环结束

   #2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

E3.10-6.rb 这个小程序不是最优化的，你可以复制到自己的编辑器内，试着反复重构优化。

七. times , upto , downto , each ,step

看程序 E3.10-7.rb 。

```
3.times { print  "Hi!" }                  #Hi!Hi!Hi!
1.upto(9) {|i| print  i  if  i<7 }        #123456
9.downto(1){|i| print  i  if  i<7 }       #654321
(1..9).each {|i| print  i  if  i<7}       #123456
0.step(11,3) {|i| print  i  }             #0369
```

3.11 异常与线程

与 Java 中的 try…catch…finally…throw 相对应，Ruby 中用 begin/end …rescue…ensure … raise 来处理异常，retry 可以用在 rescue 中。可以只用 rescue 或是 ensure，两者都使用时，rescue 必须在 ensure 前。

如果你初识 Ruby，不必理会异常与线程。Java 程序员用异常来保证文件和数据库连接的关闭。

3.12 一个综合小练习

还是求50以内的素数，尽可能地用到本章更多的语法，看程序 E3.12-1.rb 。

```
#  E3.12-1.rb        求50以内的素数

$arr=[ ]      #建立一个全局数组  $arr

$arr[0]=2

 def add_prime(n)    #定义方法  将 n以内的奇素数加入$arr

    3.step(n,2){|num|$arr <<num if is_prime?num }

 end

 def  is_prime?(number)   #定义方法  判断一个数是否是素数

    j=0                  #数组下标

    while  $arr[j] * $arr[j] <=number

      return false  if  number  %  $arr[j] ==0

      j +=1

    end

    return true

 end

add_prime(50)

print  $arr.join(", "),"\n"        #转换成字符串输出
```

$arr <<num 的含义是：将素数num作为数组的一个元素加入到$arr中。你很容易地理解了这个小程序，说明本章内容你已经熟练掌握了。你如果不做程序员，有点儿遗憾。

第四章　一切都是对象

4.1　两种思维方式

人们起初使用计算机来进行科学计算，比如说：计算级数和，计算积分值。那时，程序要处理的都是带小数点的数字。后来，人们要处理文本，有点麻烦，好在文字也可以转换成数字。这两个时期，编写程序的思想是：一个步骤，一个步骤地告诉计算机如何做，做完一大段步骤，就算完成了一个功能模块。

再后来，人们希望计算机能应用在管理中。比如说：机场管理。一个大都市的民航机场，每天有上千架飞机起落，乘客上万，行李、包裹无数，乘务、地勤人员上千，安检信息上万，气象信息上万……这样百万、千万的数据要及时处理，靠旧的编程思维是无法实现的。

为了应付越来越复杂的管理应用，人们转变编写程序的思想：

（1）．将一架飞机看作飞机类事物的一个具体实例，将一座塔台看作塔台类事物的一个具体实例，将一名乘客看作乘客类事物的一个具体实例……

（2）．一个具体实例有变化的时候，就自己主动传递消息给相关联的具体实例；

（3）．相关联的具体实例收到消息后，根据之前定好的策略，作出应有的反应（回应，转发……）。

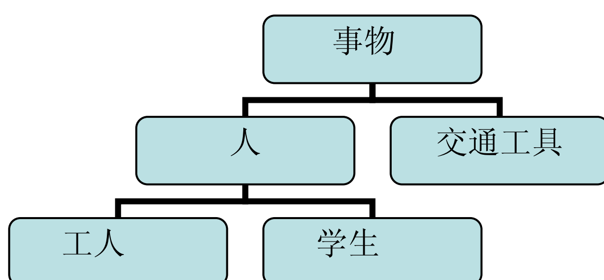
初期的编程思想是：以“如何做”为指导来编写代码。这时期的编程语言叫过程语言，提倡结构化地设计程序代码。代表语言是 FORTRAN 和 C。

现在的编程思想是：以“谁将被影响”为指导来编写代码。叫面向对象的编程语言，以类为模块，以消息来驱动程序的执行。代表语言是 C++ 和 Java。

4.2　对象

英文 Object ，计算机业界现在已经习惯翻译为“对象”；口语化一些，中文的意思就是“事物”。

“事物”这个词有点抽象，你当然可以具体到人，或是具体到交通工具。人可以再具体一些，学生？工人？如果是学生，叫什么名字？



每个事物都有一些特点，人有身高，体重，在程序中我们称之为属性；还可以有一些行为，人要吃饭，睡觉，在程序中我们称之为方法。

学生是人，自然有身高，体重，自然要吃饭，睡觉。如果你把人看作一类事物，把学生看作一类事物；那么，人是父类型，学生是子类型。子类型从父类型自然得到属性、方法，我们称之为继承。

学生要考试，工人不要；工人拿工资，学生不拿（一般而言）。同一个父类，不同的子类有不同的行为和状态，我们称之为多态。

人们编写程序，也就是在描述一类事物的特点（属性）、行为（方法）。有时候是模拟描述自然界中已有的一类事物，还有时候是创造地描述自然界中没有的一类事物。

当人们决定了代码世界中一类事物的属性、方法，在代码世界中，这类事物的属性、方法只有定义代码的人知道，其它的类是不知道的。这就是封装。

封装、继承、多态是面向对象编程的三个本质特征。

（人们可以决定代码世界中一类事物的属性、方法，当然可以修改代码世界中一类事物的属性、方法，而且可以委托其它的类来修改，甚至删除。这是动态语言超越静态语言之处。由于代码是一直运行着，与其它代码一直交互着，修改、删除应该慎重，避免产生副作用）。

4.3　封装

让我们来定义一个类，类名是 Person，类名首字母要大写；属性有姓名@name、年龄@age、国籍@motherland，实例变量用@开头；　方法有一个，叫 talk, 方法名和参数名应该用一个小写字母开头或者用一个下划线开头，看程序 E4.3-1.rb 。

```
class  Person
  def  initialize( name, age=18 )
    @name = name
    @age = age
    @motherland = "China"
  end  #初始化方法结束
  def  talk
    puts "my name is "+@name+", age is "+@age.to_s
    if  @motherland == "China"
      puts "I am a Chinese."
    else
      puts "I am a foreigner."
    end
  end  # talk 方法结束
  attr_writer :motherland
end    # Person 类结束
p1=Person.new("kaichuan",20)
p1.talk
```

```
p2=Person.new("Ben")

p2.motherland="ABC"

p2.talk

#my name is kaichuan, age is 20

  I am a Chinese.

  my name is Ben, age is 18

  I am a foreigner.
```

@age.to_s 的含义是：将数@age 转换为字符串。

initialize 是初始化方法，相当于 Java 的构造器。参数 age 有一个缺省值 18，可以在任何方法内使用缺省参数,而不仅仅是 initialize。如果有缺省参数，参数表必须以有缺省值的参数结尾。

attr_writer :motherland 相当于

```
def motherland=(value)

  return @motherland =value

end
```

attr_ reader :motherland 相当于

```
def motherland

  return @motherland

end
```

这就是我们熟悉的 getter 和 setter 方法的简写形式。你不熟悉也不重要。

attr_accessor :motherland 相当于 attr_reader:motherland； attr_writer :motherland

这个 Person 类可以 talk，如何实现的？写 Person 类的人知道，其它的类不知道，只是调用而已。封装完成了隐藏实现。

4.4　继承

如果我们要写一个学生类，他当然有姓名、年龄、国籍，他也可以 talk，但是应该表明身份是学生。看程序 E4.4-1.rb ，接着上一节的代码，添加如下：

```
class  Student < Person
  def  talk
    puts "I am a student. my name is "+@name+", age is "+@age.to_s
  end  # talk 方法结束
end  # Student 类结束
p3=Student.new("kaichuan",25); p3.talk
p4=Student.new("Ben"); p4.talk
#I am a student. my name is kaichuan, age is 25
  I am a student. my name is Ben, age is 18
```

用“ < ”表示 Student 类是 Person 类的子类。Person 类的一切，Student 类都能继承。但是 Student 类重写了 talk 方法，所以我们看到了不同的运行结果。子类继承父类的时候，除了重写方法；也可以添加一些新的方法；或是增强父类的方法(用关键字 super 指明)。

现在说一说 new 方法。Person 类没有定义 new 方法，为什么生成 Person 类的具体实例要用 new？ Ruby 语言已经定义了一个类 Object，如果你在定义新类的时候，没有指明新类的父类，那么，Ruby 解释器认为，新类的父类是 Object 类。类 Object 含有 new 方法、initialize 方法…只要你不重写这些方法，你就自然在使用类 Object 的方法。

从这里，我们发现一个有趣的事实：你写一个类的时候，是在创造一类事物的蓝图；当你 new 的时候，一个实例就按照蓝图生成了。

一个实例生成了，又一个实例生成了…他们或许是同类型的，或许不是同类型的，或许是有亲缘关系的。无数的实例在代码世界中交互、缠结，忽生忽死，无休无止…

蓝图早已设计好了，new 的时候就是出生的时刻，那么，何时消亡呢？这里没有 C++的析构函数，也没有 Java 的 finalize( ) 方法，Ruby 语言内建了一个比 Java 更灵巧的垃圾收集器，当某个实例不再与其它代码交互了，垃圾收集器就回收它占用的系统资源，这个实例自然也就不存在了。垃圾收集器是一段代码，它作它的工作，自动地、不知疲倦地随着系统一同运作，并无自己的喜恶。

4.5　多态

不同的子类继承一个父类，不仅子类和父类的行为有变异，而且子类彼此的行为也有差异，这就是多态。看程序 E4.5-1.rb ，接着 4.3 节的代码，添加如下：

```
class   Worker < Person
  def talk
    puts "I am a worker. my name is "+@name+", age is "+@age.to_s
  end   # talk 方法结束
end      # Worker 类结束
p5=Worker.new("kaichuan",30);p5.talk
p6=Worker.new("Ben");p6.talk
# I am a worker. my name is kaichuan, age is 30
  I am a worker. my name is Ben, age is 18
```

Worker 类与 Student 类同样继承自 Person 类，亲缘关系是兄弟，当他们 talk 时，能准确表明自己身份，因为他们都重写了各自的 talk 方法。

Ruby 语言，只有重写（override），没有其它语言具有的严格意义上的重载（overload）。Ruby 语言有自己的单例方法，还有模块插入(Mix-in)，后面会深入探讨 Ruby 语言的面向对象特征。

第五章

5.1 为什么是 Ruby？

现在软件生产的代表语言是 Java。但 Java 已老，在主流市场，虽然它还会将继续存在许多年。

为什么说 Java 已老？很怀念 Pascal 的严谨、优美与高效。Java 也一样，是静态语言，沉稳的同时注定了笨重，不善腾挪；而且 Java 是强静态语言，在需要简略的地方，依然沉稳笨重。仿佛一个绅士，在宴会大厅中彬彬有礼，在厨房里彬彬有礼，在卧室里仍然彬彬有礼。灵气沾不上 Java，神来之笔沾不上 Java。这一切，注定了 Java 与软件的快速开发无缘。

为什么不是 C++，C#？从 C++开始，C 系列语言已经走上一条不归路。C++不仅兼容 C，而且囊括了模板、范型等特性，包罗万象。无论是系统调用、网络开发、数据库操作都能显试身手，可是程序员很难掌握这些，即使想熟练应用其中某一方面也不容易。一个软件，一门语言，或是一个人，当他（它）想要得到整个世界的时候，危机已经埋下了。

当我们厌倦了静态语言的时候，当我们饱受大象思维折磨的时候，Ruby 浮现出来，灵巧，快速。

5.2 Ruby 会长久么？

Ruby 会长久么？这很难回答。重要的是，编程语言将向动态回归，命令式语言将与函数式语言融合。终究有一天，编程语言完全智能化，人们用自然语言来编程。而 Ruby 有可能在编程语言的智能化发展道路上起到承上启下的作用。

Ruby 灵巧，快速，但其实并不简单。

Ruby中实现一个小功能，可以有 3 种甚至 4 种完全不同的思路与方法，因为Ruby在语法层次实现了冗余[1]，但是这样一来：

1）.程序员深入掌握 Ruby 变得不很容易；

2）.程序员们相互读懂代码也很难；

3）.软件生产是一种大规模地、群体合作的行为。许多软件公司有自己的编码规范，促使员工编码风格统一，以便于 A)程序解耦重构、B)代码复用、C)人员流动后项目如期推进。Java 撇下 C++，成为软件工业的支柱语言，正是得力于此。Ruby 灵巧，快速，千变万化，没有统一风格，难于解耦，在目前，自然不适合工业生产。

如果说语法定义的冗余增加了灵活性，没有伤害，那么，Ruby 坚持缺陷也许是一种个性美吧。在 3.6 节我叙述 I/O 的输入方法 gets 的时候，没有给出例程，是因为 Ruby for mswin32 的版本在 windows 下不能正确处理标准输入和标准输出，要想使用，只好打开一个 DOS 窗口，几个版本了，依旧如此。还有就是 Ruby 的语法中有许多容易产生歧义的地方，恕不举例，假如没有较深的功力、良好的编码风格（比如空格的使用），很容易犯错。软件生产总是偏向于成熟方案、成熟工具的。

[1]例如， length 与 size 都表示数组长度，为什么 Ruby 要定义两个方法来表示数组长度呢？因为在英语中提到长度，有人会用 length，有人会用 size。这样的例子在 Ruby 语言里非常普遍。

Ruby 语言具有动态特征，代码行为随时可以改变，本书后面内容都将围绕这一特征展开介绍。

产生高级编程语言以来的 50 年间，从没有哪一种语言像 Ruby 这样近似于现实世界。看看网络，数不清的信息扑面而来，你知道了什么是冗余；看看每一天的生活，环境污染，交通拥挤，日复一日，许多的无奈，你理解了什么是缺陷；生命里充满了不可预知，明天将发生什么，谁也不知道，也许是悲伤的事，也许是令人欣喜的事，这就是动态。Ruby 语言的冗余性、缺陷性和动态性正是现实世界的真实写照。

5.3 详解变量—— 动态类型

变量是什么？变量有哪些特征呢？学习编程的过程，就是深化理解变量的过程。

先说变量是什么？变量是代号。

在数学中，你写下一个小写的英文字母“f”，这个 f 可以是数字 5 ，也可以是一个函数式 f = n * n + 1 ，还可以是一个曲面，或者是一个逻辑蕴涵关系…数学里，我们把常用的符号约定俗成，比如π代表圆周率，∑表示求和。

编程语言的产生，建立在数学的基础上。在汇编语言的时代，一条语句

LD x,37H ，

其中的 LD 是操作码，代表一种操作；x 和 37H 是操作数，是被操作的对象。无论是 LD ，还是 x 和 37H ，对于机器来说，都只是符号。

后来，编程语言发展成两大类，一类函数式语言，一类命令式语言。命令式语言将操作数演化成现在我们熟悉的变量，将操作码演化成方法（或叫函数），对变量执行各种操作。面向对象编程又将基本变量和方法封装在一起，成为一个更复杂的变量——对象。但是，在一个类中仍然区分基本变量和方法。函数式语言则不同，一开始的函数式语言不区分变量和方法，一切都是表（list），表就是能够不断分解成单个元素的数学符号。表可以是变量，可以是方法。后来的有些函数式语言，吸取了命令式语言的语法，也区分变量和方法。

也有一些命令式语言，融合了函数式语言的语法， Ruby 就是这样的语言，变量和方法区分得不很明显。

其次，说一说变量有哪些特征呢？

1). 变量有名字；

2).变量代表的那个事物应该有一个可以用数学度量的值；长度，面积，速度大小，磁场强度…

3).为了区别事物，我们将事物分成几个基本类型。所以，代表不同类型的事物，变量也就有了不同的类型。

4). 事物总是有产生到消亡的一个过程，因此，代表事物的变量，也就有了生命期。计算机科学，是一门将时间转换成空间的科学。在程序中，我们把变量的生命期，称之为变量的作用域。

变量名，变量值，变量类型，变量的作用域，是我们学习命令式语言不可回避的几个要素。

如果你是一门编程语言的设计者，仔细考虑一下，上面四个要素，对于编程语言的使用者都是必须的吗？

作为一个使用者，1).　2). 是必须的。至于类型、生命期，与我何干？某个变量，我使用一下就丢弃了，要我操心太多，还不如我从头设计呢。

由编译内核（或解释内核）在运行时刻来判断变量类型的语言，叫动态类型语言。

变量既然是代号，那么可以代表数字，文字（字符串），代码序列（块，闭包），一段程序（文件）…在运行中，变量能够随时代表不同的事物，而不管事物是什么类型，这种语言，叫弱类型语言。这里的“弱”，是指弱化了类型的概念，不理会类型的差异。

Ruby 语言还是有基本类型。至于变量作用域，纯粹的函数式语言中是没有这个概念的。Ruby 中是有变量作用域概念的，还记得变量名前缀字符吗？实际应用中，有时会比较复杂，使用闭包时就知道了。

Ruby 语言中，一切都是对象，变量是不是对象呢？变量不是对象，变量只是引用某个对象的时候，你看到的一个代号而已。

Ruby 是动态类型语言，不用给任何变量指定数据类型，解释器会在你第一次赋值给变量时，在内部将数据类型记录下来。请看程序E5.3-1.rb ：

```
#    E5.3-1.rb
a=5
b="hh"
puts   "a = #{a}"
puts   "b = #{b}"
```

运行结果：

```
>ruby E5.3-1.rb
a = 5
b = hh
>Exit code: 0
```

Ruby 语言中，一个变量被赋予了某个数据类型的值，在程序中你可以随时再赋予这个变量其它数据类型的值。请看程序 E5.3-2.rb ：

```
#    E5.3-2.rb
a=5
print "a = ",a,"       ", a.class, "\n"
a="hh"     # a: 5 --> "hh"
print "a = ",a,"       ", a.class, "\n"
```

运行结果：

```
>ruby E5.3-2.rb
a = 5       Fixnum
a = hh       String
```

>Exit code: 0

相对于 Java， Ruby 对于变量的使用给予了你很大的自由。在 Java 中，编译的时候，就完成了类型匹配的检测，这属于前期绑定； Ruby 是在运行中检测，检测类型匹配吗？不是检测类型匹配，而是检测语法，只要与语法定义不矛盾，就能通过。 Ruby 的动态类型特点是一把双刃剑，熟手游刃有余，生手常常伤着自己。在没有了编译器查错的日子里，又没有完全驾驭 Ruby 之前，如何避免常常出错呢？有一个下口之处，就是死盯住变量的命名。用一些有意义的名字，不必太长，但是应该少用单字符，除非是循环指针变量。你也许认为我自己能看懂就行了，这是十分有害的想法。在一个项目组中，程序员是要彼此相互沟通合作的。当坏习惯养成后，要改是很难的。

5.4 蓝图可以改变吗？ ——动态语言

Ruby 是动态语言，你可以改变 Ruby 程序的结构，功能，在Ruby程序运行中。方法、属性可以被加入或去除，新的类或对象可以被建立，新的模块可以出现。请看程序 E5.4-1.rb ：

```
#    E5.4-1.rb
class Person
  def talk
    puts "Today is Saturday. "
  end
end

p1=Person.new
p1.talk     # Today is Saturday.

class Person
  def talk
    puts "Today is #@date. "
  end
 attr_accessor :date
end

p1.date="Sunday"
```

```
p1.talk     # Today is Sunday.
```

当然，除了修改方法，添加方法，你还可以除去方法。看程序 E5.4-2.rb ：

```
#    E5.4-2.rb
class Person
  def talk
    puts "Today is Saturday. "
  end
end

p1=Person.new
p1.talk     # Today is Saturday.

class Person
  undef :talk
end
#p1.talk        talk方法已经不存在
```

Ruby 语言灵活，因为 Ruby 是动态语言； Ruby 语言强大，因为 Ruby 是动态语言； Ruby 语言初学者容易犯错误，也因为 Ruby 是动态语言。

5.5　一些编码建议

这里不是 Ruby语言的编码约定，只是建议，很少的一些建议。在语法正确的前提下，你可以按照自己的编码风格自由组织你的代码。

一. 命名

常量全用大写的字母，用下划线分割单词。例如：MAX， ARRAY_LENGTH。

类名和模块名用大写字母开头的单词组合而成。例如：MyClass, Person。

方法名全用小写的字母，用下划线分割单词。例如：talk, is_prime?。在Ruby里,有时将“!”和“?”附于某些方法名后面。惊叹号“!”暗示这个方法具有破坏性, 有可能会改变传入的参数。问号“?”表示这个方法是一个布尔方法，只会返回 true 或 false。

变量和参数用小写字母开头的单词组合而成。例如：name, currentValue。

类名、模块名、变量名、参数名最好使用“名词”或者“形容词＋名词”。方法名最好使用“动词”或者“动词＋名词”。例如：aStudent.talk 。

二. 空格和圆括号

关键字之后要留空格。

逗号“，”、 分号“；”之后要留空格。 “，”、 “；”向前紧跟，紧跟处不留空格。

赋值操作符、比较操作符、算术操作符、逻辑操作符，如“=”、“+=” “>=”、“<=”、“+”、“*”、“%”、“&&”、“||”等二元操作符的前后应当加空格。

一元操作符如“!”、“~”等之后不加空格。

象“［］”、“.”、“：：”这类操作符前后不加空格。

函数名之后不要留空格，紧跟左圆括号"（"，以与关键字区别。左圆括号"（"向后紧跟，右圆括号"）"向前紧跟，紧跟处不留空格。

Ruby中圆括号常常被省略，看程序 E5.5-1.rb ：

```
#E5.5-1.rb

def   talk name
   "Hi! " + name
end

puts talk "kaichuan"          #Hi! kaichuan
puts talk("kaichuan")         #Hi! kaichuan
puts (talk "kaichuan")        #Hi! kaichuan
puts (talk("kaichuan"))       #Hi! kaichuan
```

优先规则会自动确定哪个参数被哪个方法使用。但是，生活并不总是美好的，事情经常变得复杂，看程序 E5.5-2.rb ：

```
#E5.5-2.rb

a=5
b=3
puts     a>b   ?   "a>b"   :   "a<=b"                    # a>b
puts    (a>b)? ("a>b") : ("a<=b")                        # a>b
#puts     a>b?   "a>b"   :   "a<=b"        错误语句
```

最后一条语句，变量b 与三元条件运算符的问号“？”之间没有空格，没有圆括号，产生错误。所以建议除了极简单的情况，还是使用圆括号为好。

圆括号还可以把几个语句约束成一个语句集合，看程序 E5.5-3.rb ：

```
#E5.5-3.rb


a = 3
 b = 1;   a += b       if   3 > 5
print     "a = ", a, "\n"                    # a = 3
print     "b = ", b, "\n"                    # b = 1


c = 3
(d = 1;   c += d)   if   3 > 5
print     "c = ", c, "\n"                  # c = 3
print     "d = ", d, "\n"                  # d = nil
```

条件为假，语句集合里的变量d 没有被赋值。

三. 使用 return

你在定义方法的时候，在最后一行可以显式地 return 某个值或几个值，但却不是必须的。 Ruby 方法的最后一行语句如果是表达式，表达式的值会被自动返回；最后一行语句如果不是表达式，就什么也不返回。

return 并不仅仅用在方法的最后一行。使用 break 你能够跳出本层循环，如果要从多重循环体中跳出，可以使用return ，结束这个方法； return还能够从方法的某个执行点立即退出，而不理会方法的其余代码，例如程序 E3.12-1.rb 的方法

is_prime?。

四. 注释

养成写注释的习惯吧！你见过没有路标的高速公路吗？

注释表明了一段代码块的功能、意图或是代码块的解释，应该简洁明了，错误的注释不如没有注释。一般地，注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不要放在代码的下方。

第六章　深入面象对象

6.1　重载？重写

在 Java 中，重载（overload）和重写（override）是用来表现多态性的两种重要方式。override 也有译作“覆盖”、“覆写”。Java 中称作“覆写”比较恰当。

重载方法是指一个类中，方法名相同、参数列表不同的几个方法，调用时根据不同的参数调用不同的方法。方法重载与返回类型无关。

覆写方法是指子类有一个方法，方法名、参数列表、返回类型与父类的某个方法完全一致。调用时会调用子类的方法，而屏蔽掉父类的同名方法。需要注意的是，子类覆写的方法，其可访问性一定要强于或等同于，父类被覆写的同名方法。

覆写发生在子类和父类之间，当然也可以是子类和父类的父类之间。重载不仅仅是发生在子类和父类之间，大多数时候，发生在同一个类中。

4.5 节说到：“ Ruby 语言，只有重写，没有其它语言具有的严格意义上的重载。” 下面仔细分析 Ruby 为何没有重载，只有重写。

Java 的重载，参数列表不同有三种形式。

一．参数个数不同

```
sum(int a)

sum(int a ,int b)
```

二．参数个数相同，参数类型不同

```
sum(int a)

sum(float a)
```

三．参数个数相同，参数类型相同，对应位置不同

```
sum(int a , float b)

sum(float a ,int b)
```

Ruby 支持缺省参数，我们看程序 E6.1-1.rb ：

```
def  sum( a, b=5 )
  a+b
end
puts sum(3,6)          #9
puts sum(3)            #8
```

调用 sum(3,6) 与 调用 sum(3) 在 Java 中是调用不同的方法，在 Ruby 中其实是在调用同一个方法。

Ruby 还支持可变参数，我们看程序 E6.1-2.rb ：

```
#  E6.1-2.rb
def sum( *num )
  numSum = 0
  num.each { |i| numSum+=i }
  return numSum
end
puts sum()                    #0
puts sum(3,6)                 #9
puts sum(1,2,3,4,5,6,7,8,9)   #45
```

由于缺省参数和可变参数，参数个数不同而产生的重载，在 Ruby 中不再有效。Ruby 语言中，定义方法时，不指定参数类型，因此第二种形式的重载也不存在。第三种形式的重载，实际是第二种形式的演化，所以，也就不存在了。综上所述，Ruby 语言，没有方法的重载。

方法的重写，我们在 4.4 节和 4.5 节已经看到了。Worker 类与 Student 类继承自 Person 类，并且各自重写了 Person 类的 talk 方法。

这是在子类和父类之间，如果在同一个类中写两个同名方法呢？看程序 E6.1-3.rb ：

```
#   E6.1-3.rb
def talk (a)
    puts "This is talk version 1."
end


def talk (a,b=1)
    puts "This is talk version 2."
end


talk (2)            # This is talk version 2.
talk (2,7)          # This is talk version 2.
```

总是写在后面的方法被执行。

从上面三个小程序可知：在 Ruby 中，我们说覆写是指重写，我们说重载也是指重写。

Java 和 C++ 是静态语言，程序代码运行中不可以再改变类的属性、方法，为了更好地表现面向对象的多态特征，所以用覆写和重载来加强程序的灵活性，在

程序运行的时候，动态地选择要使用的方法， 完成后期绑定。而 Ruby 是动态语言，可以随时改变类的属性、方法，所以覆写和重载的重要性就降低了。仔细体会一下，一者是增大可选择性，一者是随时修改。

6.2　增强父类方法

如果我们只是想增强父类的方法，而不是完全地替代它,就可以用关键字 super 指明。看程序　E6.2-1.rb　：

```
#E6.2-1.rb
class Person
   def talk(name)
     print "my name is #{name}."
  end
end

class Student < Person
  def talk(name)
     super
     print "and I'm a student.\n"
  end
end

aPerson=Person.new
aPerson.talk("kaichuan")   #my name is kaichuan.
print "\n\n"

aStudent=Student.new
```

```
aStudent.talk("kaichuan")    #my name is kaichuan.and I'm a student.
```

Person 类的 talk 方法只是报告姓名。 Student 类的 talk 方法用 super 来调用 Person 类的 talk 方法，报告姓名；随后又加上了一条语句，来表明身份。

6.3 实例变量、类变量、类方法

学习编程的过程，就是深化理解变量的过程。在面向对象编程中也不例外，这一节，我们继续深化理解变量。

先把 Ruby 放在一边，从编程语言的视角来探讨变量。

如果一个变量，第一次赋值后，就不再允许改变变量值，这样的变量称之为常变量，简称常量。就像数学分析中的常函数，y = 3 是一个平行于 x 轴，并且函数值总是为 3 的函数。常量名用大写字母开头。

如果一个变量，在其作用域上的每一个执行点，都可以改变变量值，这样的变量称之为可变量，简称变量。

如果一个变量，其作用域遍及在程序的任何位置，这样的变量称之为全局变量；与之相对，作用域仅限于在程序的某一单元的变量，称之为局部变量。

面向对象的编程，以类为单元模块。类是设计蓝图，具体的事物是实例对象。前面 5.3 节说到：“变量名，变量值，变量类型，变量的作用域，是我们学习命令式语言不可回避的几个要素”。对于面向过程的命令式语言，这四个要素已经够了；对于面向对象的命令式语言，还要加上变量的第五个要素——共享性。

如果一个变量，只能被某个实例对象使用，这样的变量称之为实例变量；如果一个变量，能被某个类的所有实例对象共享，这样的变量称之为类变量。

回到 Ruby 。常量可以定义在类和模块中，不能定义在方法中。如果在外部访问类或模块中的常量，要使用域作用符:: 。

全局变量用$ 开头。

实例变量，变量名用@ 开头；类变量，变量名用@@ 开头。

Ruby中所说的局部变量，可以是存在于类中、方法中、模块中、一个循环中、一个过程对象中。局部变量名用小写字母开头。

在4.3节，我们使用的@name（姓名）、@age（年龄）、@motherland（国籍），都是实例变量，被每个实例独享。程序 E4.3-1.rb 中，实例 p1与实例 p2的姓名不一样，年龄也不一样。

如果我们希望一类事物共享某个变量，比如：在 4.4节 Student 类中，需要有一个变量来代表班级人数，实例 p3与实例 p4假使在一个班级里，班级人数这个属性应该是相同的，当然，随着 Student 类的实例不断生成，班级人数也要不断增加，如何解决呢？

类变量能够很好地实现这个需求，看程序 E6.3-1.rb：

```
#E6.3-1.rb
class StudentClass
  @@count=0

  def initialize( name )
    @name = name
    @@count+=1
  end

  def talk
    puts "I am #@name, This class have #@@count students."
  end
end
```

```
p1=StudentClass.new("Student 1 ")

p2=StudentClass.new("Student 2 ")

p3=StudentClass.new("Student 3 ")

p4=StudentClass.new("Student 4 ")

p3.talk          # I am Student 3 , This class have 4 students.

p4.talk          # I am Student 4 , This class have 4 students.
```

与全局变量和实例变量不同，类变量在使用前必须要初始化；全局变量和实例变量如果没有初始化，其值为 nil 。

如果教务主任想知道某个班级现在有多少人数，不需要到这个班级去问学生，应该可以通过其它途径来获取信息。这里要用到类方法——不依赖于任何特定实例对象的方法。类方法与实例方法的定义方式不同，定义类方法要在方法名前加上类名和一个点号“.”。看程序 E6.3-2.rb：

```
#E6.3-2.rb

class StudentClass

  @@count=0


  def initialize

    @@count+=1

  end


  def  StudentClass.student_count

    puts "This class have #@@count students."

  end

end
```

```
p1=StudentClass.new

p2=StudentClass.new

StudentClass.student_count     # This class have 2 students.


p3=StudentClass.new

p4=StudentClass.new

StudentClass.student_count     # This class have 2 students.
```

调用一个类方法，与定义类方法一样，要在方法名前加上类名和一个点号“.”。类方法提供了一个途径，在类的外部访问类变量，无须通过类的实例方法。

类变量，类方法在 Java 里与之相对应的是 static变量，static方法。

在 Java 里，你写一个类，是在画设计图纸，当你 new 的时候，才生成一个实例对象。Ruby 语言中，一切都是对象，单个实例（具体事物）是对象，类（蓝图）也是对象。你拿着设计图纸可以生产出很多汽车，而设计图纸对于纸来说，只是纸这类事物的一个具体实例。 Ruby 里，有元类的概念，通过关键字 self 与类方法的灵活使用，程序代码可以产生很多变化，这里不再展开叙述。可以思考用类方法来实现《设计模式》一书中的单子模式。

6.4　单例方法

同一份设计蓝图（类），不同的实例对象，可以表现出不同的行为特征，这种（不牵涉继承的）多态性在 Java 这样的静态语言里，通过方法重载得到具体实现。6.1 节中我们分析知道了 Ruby 中的重载是指重写，Ruby 如何来反映不同实例对象的不同行为特征呢？

在 Ruby 里，可以给具体的实例对象添加实例方法，这个方法只属于这个实例对象，我们把这样的方法称之为单例方法。

看程序　E6.4-1.rb ：

```
#E6.4-1.rb
class Person
  def talk
    puts "Hi! "
  end
end


p1=Person.new
p2=Person.new


  def  p2.talk            #定义单例方法 p2.talk
    puts "Here is p2. "
  end
```

```
def p2.laugh                #定义单例方法 p2. laugh
    puts "ha,ha,ha... "
  end


p1.talk          # Hello!
p2.talk          # Here is p2.
p2.laugh         # ha,ha,ha...
```

单例方法也叫作单件方法。定义单例方法，首先要生成一个实例对象，其次，要在方法名前加上对象名和一个点号“.”。

程序 E6.4-1.rb 中，对象 p1 不可以 laugh， laugh 方法只属于 p2 对象。

实例方法，属于类的每个实例对象。单例方法只出现在单个实例对象中。用单例方法可以极大地丰富多态性在 Ruby 中的表现力。

6.5 访问控制

在 Ruby 里，要读取，或是改变对象的属性，唯一的途径是调用对象的方法。控制了对方法的访问，也就控制了对对象属性的访问。

控制对方法的访问，有三种方式：

访问控制	意义
public	可以被任何实例对象调用，不存在访问控制；
protected	可以被定义它的类和其子类访问， 可以在类中或子类中指定给实例对象；
private	可以被定义它的类和其子类访问， 不能被实例对象调用。

方法默认都是公有的（ initialize 方法除外，它永远是私有的）。

看程序 E6.5-1.rb ：

```
#E6.5-1.rb
class Person
 def talk
      puts "      public :talk,     将调用 speak"
      speak
   end


   def speak
      puts "protected :speak,将调用 laugh"
```

```
      laugh
    end

    def laugh
      puts "     private:laugh"
    end
    protected :speak
    private      :laugh
end

p1=Person.new
p1.talk
#p1.speak    实例对象不能访问 protected 方法
#p1.laugh     实例对象不能访问 private 方法
```

运行结果：

```
>ruby E6.5-1.rb
      public :talk，将调用 speak
protected:speak,将调用 laugh
    private:laugh
>Exit code: 0
```

再看程序 E6.5-2.rb ：

```
#E6.5-2.rb
```

```
class Person

  def speak
    "protected:speak  "
  end

  def laugh
    "   private:laugh"
  end

  protected :speak
  private     :laugh
end

class Student < Person

  def useLaugh
    puts laugh
  end

  def useSpeak
    puts speak
  end
```

```
end

p2=Student.new

p2. useLaugh          #    private:laugh

p2. useSpeak          # protected:speak
```

从程序 E6.5-1.rb 和程序 E6.5-2.rb ，我们没有看出 protected 和 private 的区别，到底区别在哪里呢？

答案在程序 E6.5-3.rb ：

```
#E6.5-3.rb


class Person
  def speak
    "protected:speak  "
  end


  def laugh
    "   private:laugh"
  end


  protected :speak
  private    :laugh


  def useLaugh(another)
```

```
    puts another.laugh    #这里错误，私有方法不能指定对象
  end

  def useSpeak(another)
    puts another.speak
  end

end

p1=Person.new
p2=Person.new

p2.useSpeak(p1)           # protected:speak
#p2.useLaugh(p1)
```

从上面三个程序可以得出结论：

- public 方法，可以被定义它的类和其子类访问，可以被类和子类的实例对象调用；
- protected 方法，可以被定义它的类和其子类访问，不能被类和子类的实例对象直接调用，但是可以在类和子类中指定给实例对象；
- private 方法，可以被定义它的类和其子类访问，私有方法不能指定对象。

Ruby 语言的访问控制是动态的，是在程序运行时刻确立的。

```
#E6.5-4.rb
class Person
  private    #后面的方法设定为 private
```

```
    def talk
      puts " already talk "
    end
end

p1=Person.new
#p1.talk     private 方法不能访问

class Person
    public :talk
end

p1.talk        # already talk
```

你可以根据自己的需要，在程序的不同位置，改变某个方法的访问控制级别，让你的程序更加富于变化。

第七章

7.1　模块

我们常常把许多零散的小物件放在一个盒子里，或者放在一个抽屉里，这些小物件可能是铅笔，墨水，字典等学习用品，也有可能是不相关的几件物品。在程序中，相关的、不相关的代码的组合，叫作模块。一般情况下，我们总是把功能相关的代码放在一个模块里。

把功能相关的程序代码放在一个模块里，体现了模块的第一个作用：可以被其它程序代码重复使用。

看程序 E7.1-1.rb ：

```
#E7.1-1.rb

puts   Math.sqrt(2)      # 1.4142135623731

puts   Math::PI          # 3.14159265358979
```

Ruby标准包里的 Math 模块提供了许多方法，比如：求平方根 sqrt ，使用的时候要这么写：模块名.方法名（参数）。你可以 Math.sqrt(37.2/3) ，Math.sqrt(a*5+b)； Math 模块还提供了两个常量，圆周率π 和自然对数底 e，使用的时候要这么写：模块名：：常量名。

数学中常用的函数，Math 模块都提供了。每个使用 Math 模块的程序员无须再重复编写这些常用的函数与常数。

定义模块用 module...end 。模块与类非常相似，但是：

A) 模块不可以有实例对象；

B) 模块不可以有子类。

7.2　命名空间

如果你觉得 Ruby 标准包里的 Math 模块提供的 sqrt 方法不好，不能够设置迭代区间和精度，你重写了一个 sqrt 方法。你的同事在他的程序里需要调用你的 sqrt 方法，也要调用标准 Math 模块提供的 sqrt 方法，怎么办呢？

模块的第二个作用：提供了一个命名空间（namespace），防止命名冲突。

看程序 E7.2-1.rb ：

```
#E7.2-1.rb


module    Me
   def sqrt(num, rx=1, e=1e-10)
      num*=1.0
      (num - rx*rx).abs <e ?      rx :   sqrt(num, (num/rx + rx)/2, e)
   end
end


include Math
puts      sqrt(293)                  # 17.1172427686237
#puts      sqrt(293, 5, 0.01)


include Me
puts      sqrt(293)                  # 17.1172427686237
```

```
puts    sqrt(293, 5, 0.01)      # 17.1172429172153
```

如你所见，只要 include 模块名，就能使用不同模块的 sqrt 方法，Math 模块的 sqrt 方法不能有三个参数，Me 模块的 sqrt 方法可以是一个参数，或者二个参数，或者三个参数。Me 模块被 include 在 Math 模块后面，Math 模块的 sqrt 方法就被 Me 模块的 sqrt 方法覆盖了。

现在出现一个问题，你喜欢像 Math 模块那样调用 sqrt 方法，

```
puts    Math.sqrt(2)
```

而不喜欢像 Me 模块那样调用 sqrt 方法，

```
include Me

puts    sqrt(2)
```

还记得类方法吗？我们可以定义一个模块方法，在方法名前加上模块名和一个点号“.”。

看程序 E7.2-2.rb ：

```
#E7.2-2.rb


module Me
  def sqrt(num, rx=1, e=1e-10)
    num*=1.0
  (num - rx*rx).abs <e ?  rx :  sqrt(num, (num/rx + rx)/2, e)
  end
end


module Me2
```

```
  def Me2.sqrt(*num)
    "This is text sqrt. "
  end
  PI=3.14
end


puts    Math.sqrt(1.23)    #1.10905365064094
puts    Math::PI           #3.14159265358979
puts    Me2.sqrt(55, 66, 77, 88, 99)   #This is text sqrt.
puts    Me2::PI                        #3.14


include Me
puts    sqrt(456, 7, 0.01)    #21.3541565188558
```

调用一个模块方法，与定义模块方法一样，要在方法名前加上模块名和一个点号“.”。模块方法提供了一个途径，在模块的外部访问模块内部方法，无须 include 模块。定义模块常量不需要如此。

7.3 糅和(Mix-in) 与多重继承

糅和，也译作混合插入，也许就称作 Mix-in 比较合适。

现实生活中，一个乒乓球不仅是球类物体，也是有弹性的物体。 C++ 支持多重继承，多重继承有时会导致继承关系的混乱，Java 只提供了单继承，通过接口可以得到多重继承的优点，又没有多重继承的缺点。Ruby 也是单继承，不是通过接口，而是通过 Mix-in 模块，来实现多重继承的优点。

模块的第三个作用：实现了类似多重继承的功能。

我们有一个 Student 类，有着 Person 类的属性和方法，还会做数学题——求平方根。已经有了 Me 模块，只要 Mix-in 在 Student 类里就可以了。

看程序 E7.3-1.rb ：

```
#E7.3-1.rb

module Me
  def sqrt(num, rx=1, e=1e-10)
    num*=1.0
    (num - rx*rx).abs <e ?    rx :  sqrt(num, (num/rx + rx)/2, e)
  end
end

class Person
  def talk
```

```
        puts "I'm talking."
    end
end

class Student < Person
    include Me
end

aStudent=Student.new
aStudent.talk                    # I'm talking.
puts aStudent.sqrt(20.7,3.3)    # 4.54972526643248
```

通过“ < 父类名 ”，一个子类可以得到父类的属性和方法；通过“ include 模块名 ”，一个子类可以得到某个模块的常量和方法。类不能被 include 。

与 include 方法相对应的，还有一个 extend 方法。如果并不是 Student 类的每个对象都会求平方根，只有某一个学生会，如何办到呢？

看程序 E7.3-2.rb ：

```
#E7.3-2.rb

module Me
    def sqrt(num, rx=1, e=1e-10)
        num*=1.0
    (num - rx*rx).abs <e ?  rx :  sqrt(num, (num/rx + rx)/2, e)
    end
```

```
end


class Student

end


aStudent=Student.new

aStudent.extend(Me)

puts aStudent.sqrt(93.1, 25)    # 9.64883412646315
```

include 方法为一个类的所有对象包含某个模块； extend 方法为一个类的某个对象包含某个模块。

7.4 require 和 load

程序 E7.3-1.rb 中先写了 Me 模块，然后 include Me 模块，实现了 Mix-in 功能，但是，这样没能做到代码复用。

我将 Me 模块写在文件 E7.4-1.rb 中，将 Person 类写在文件 E7.4-2.rb 中，这时候 Student 类如何使用 Me 模块和 Person 类呢？这里要用到 require 方法。

看程序 E7.4-3.rb ：

```
#E7.4-3.rb
require "E7.4-1"
require "E7.4-2"

class Student < Person
  include Me
end

aStudent=Student.new
aStudent.talk                # I'm talking.
puts aStudent.sqrt(77,2)     # 8.77496438739435
```

使用 require 方法让你的程序文件变得简洁有力。 require 方法包含另一个文件，另一个文件名需要是一个字符串。

还有一个 load 方法与 require 方法相对应，也用来包含另一个文件。

看程序 E7.4-4.rb ：

```
#E7.4-4.rb

load "E7.4-1.rb"


class Student

end


aStudent=Student.new

aStudent.extend(Me)

puts aStudent.sqrt(100.1, 12)    # 10.0049987506246
```

require 包含文件，只加载一次，遇到同一文件时自动忽略；不同路径下的同名文件会多次加载。 load 包含文件，加载多次，即使是相同路径下同一文件。

总结一下：

- require，load 用于包含文件；include，extend 则用于包含模块。
- require 加载文件一次，load 加载文件多次。
- require 加载文件时可以不加后缀名，load 加载文件时必须加后缀名。
- require 一般情况下用于加载库文件，而 load 用于加载配置文件。

利用 load 多次加载文件的特性，可以用来实现程序的无缝升级和系统的热部署。程序功能改变了，你只需要重新 load 一次，其它代码与它再次交互的时候，这个程序实际上已经不是原来的程序了。

第八章

8.1 再说数组

一. 建立一个数组

```
#E8.1-1.rb


arr1=[]

arr2=Array.new

arr3=['4 ','5 ','6 ']


print arr1, "\n"

print arr2, "\n"

print arr3, "\n"
```

运行结果：

```
>ruby E8.1-1.rb




4 5 6

>Exit code: 0
```

二. 访问数组元素

Ruby 以整数作为下标，访问数组元素通过数组下标，数组下标称作数组索引

比较好一些。

```
#E8.1-2.rb

arr=[3,4,5,6,7,8,9]

puts arr[0]              #3
puts arr.first           #3
puts arr[arr.length-1]   #9
puts arr[arr.size-1]     #9
puts arr.last            #9
puts arr[-1]             #9
puts arr[-2]             #8
print arr[1..3]   ,"\n"  #456
print arr[-3,2]   ,"\n"  #78
```

数组的索引从 0 开始，一直到数组的长度减去 1；负数表示从数组末尾开始的索引；用一对数字来索引数组，第一个数字表示开始位置，第二数字表示从开始位置起的元素数目。

三. 增加、删除数组元素

Ruby 的数组大小是动态的，你能够随时增加、删除数组元素。

print arr.join(", "),"\n" 意思是：将数组 arr 转换成字符串输出，用", "隔开每个元素，并且换行。

```
#E8.1-3.rb

arr=[4,5,6]
print arr.join(", "),"\n"                    #4, 5, 6

arr[4] = "m"      #把 4 号索引位置元素赋值为"m"
print arr.join(", "),"\n"                    #4, 5, 6, , m
print arr[3] ,"\n"      #打印 3 号索引位置元素      #nil

arr.delete_at(3)      #删除 3 号索引位置元素
print arr.join(", "),"\n"                    #4, 5, 6, m

arr[2] = ["a","b","c"]   #把 2 号索引位置元素赋值为["a","b","c"]
print arr.join(", "),"\n"                    #4, 5, a, b, c, m
print arr[2] ,"\n"      #打印 2 号索引位置元素      #abc

arr[0..1] = [7,"h","b"]   #把 0..1 号元素替换为 7,"h","b"
print arr.join(", "),"\n"                    #7, h, b, a, b, c, m

arr.push("b" )          #加入元素"b"
print arr.join(", "),"\n"                    #7, h, b, a, b, c, m, b

arr.delete(["a","b","c"] )      #删除元素["a","b","c"]
```

```
print arr.join(", "),"\n"                              #7, h, b, m, b

arr.delete("b")          #删除所有元素"b"

print arr.join(", "),"\n"                              #7, h, m


arr.insert(3,"d")        #在 3 号索引位置插入元素"d"

print arr.join(", "),"\n"                              #7, h, m, d


arr<<"f"<<2              #加入元素"f"；加入元素 2

print arr.join(", "),"\n"                             #7, h, m, d, f, 2

arr.pop                  #删除尾元素

print arr.join(", "),"\n"                             #7, h, m, d, f

arr.shift                #删除首元素

print arr.join(", "),"\n"                            #h, m, d, f

arr.clear                #清空数组 arr

print arr.join(", "),"\n"                            #
```

四. 数组运算

```
#E8.1-4.rb


aaaa=[" aa ",4,5," bb "]

bbbb=[4,1,3,2,5]


print aaaa + bbbb      ,"\n"       # aa 45 bb 41325
```

```
print aaaa * 2          ,"\n"      # aa 45 bb    aa 45 bb

print bbbb - aaaa       ,"\n"      #132


        #并运算；交运算

print aaaa | bbbb       ,"\n"      # aa 45 bb 132

print aaaa & bbbb       ,"\n"      #45


        #排序；倒置

print bbbb.sort         ,"\n"      #12345

print aaaa.reverse      ,"\n"      # bb 54 aa
```

8.2　再说字符串

一．生成一个字符串

字符串是 String 类的对象，一般使用字面值来创建。

```
#E8.2-1.rb


str1 = 'this is str1'

str2 = "this is str2"

str3 = %q/this is str3/

str4 = %Q/this is str4/

str5 = <<OK_str

   Here is string document, str5

        line one;

        line two;

        line three.

   OK

OK_str


puts str3

puts str4

puts str5
```

运行结果：

```
>ruby E8.2-1.rb
this is str3
this is str4
   Here is string document, str5
        line one;
        line two;
        line three.
   OK
>Exit code: 0
```

%q 用来生成单引号字符串；%Q 用来生成双引号字符串。%q 或者%Q 后面跟着的是分隔符，可以是配对的！ !； / /； < >； ( )； [ ] ；{ }；等等。

str5 是一个字符串文档，从 <<和文档结束符的下一行开始，直到遇到一个放置在行首的文档结束符，结束整个字符串文档。

一个数组可以用 join 方法转换成字符串，join( ) 内的参数也是一个字符串，用来分隔数组的每个元素，例如：arr.join(", ")。

二. 字符串操作

字符串既然是 String 类的对象，String 类的方法你都可以使用在字符串变量上，String 类的方法非常多，下面略举几例。

```
#E8.2-2.rb

str = ' this' + " is"
str += " you"
```

```
str << " string" << " ."

puts   str*2          # this is you string . this is you string .
puts   str[-12,12]    #you string .
```

三. 字符串转义

双引号括起来的字符串会有转义，例如："\n" 表示换行。还有一些其它的转义符号，比如制表符之类。

```
#E8.2-3.rb

str = " this is you string ."
puts   str*2                    # this is you string . this is you string .

str = " this is you string .\n"
puts   str*2                    # this is you string .
                                  this is you string .
str = " \tthis is you string ."
puts   str                      #     this is you string .

str = ' this\'s you string .\n'
puts   str                      # this's you string .\n
```

单引号括起来的字符串并不会对字符串作任何解释，你看到的是什么便是什

么，有一个例外：单引号字符串里的 单引号 需要转义。

四. 字符串内嵌表达式

在双引号扩起来的字符串中，不仅可以使用各种转义符，而且可以放置任意的 Ruby 表达式在 #{ } 之中，这些表达式在使用这个字符串的时候被计算出值，然后放入字符串。

```
#E8.2-4.rb

def hello(name)
    " Welcome, #{name} !"
end

puts  hello("kaichuan")  # Welcome, kaichuan !
puts  hello("Ben")       # Welcome, Ben !
```

字符串内嵌表达式，使得你能够更加灵活地组织代码，表现出更强、更多的动态特性。

8.3　正则表达式

一本入门小书，不想涉及正则表达式，但是正则表达式与 Ruby 语言联系如此紧密，总让人感觉绕不开它。所以匆匆结束字符串的介绍，边学边说正则表达式。

正则表达式之强大、复杂，由来已久。自从 1956 年提出了"正则集代数"，正则表达式就逐渐被广泛地应用于操作系统，编程语言，算法设计，人工智能……

现在，除了 Perl、Python 这样支持强大正则表达式功能的语言之外，Java，JavaScript， C# 等语言都纷纷支持正则表达式，只不过支持的程度不同。而 Ruby 正是一种强烈而灵活地支持正则表达式的语言。

下面，我努力尝试尽可能简单地描述 Ruby 中的正则表达式。

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串；将匹配的子串做替换；或者从某个串中取出符合某个条件的子串；等等。

Ruby 中，可以使用构造器显式地创建一个正则表达式，也可以使用字面值形式 /正则模式/ 来创建一个正则表达式。

```
#E8.3-1.rb

str="Hello,kaichuan,Welcome!"

puts   str =~   /kaichuan/          #6
puts   str =~   /a/                 #7
puts   str =~   /ABC/               #nil
```

我在字符串 str 中找我的名字 kaichuan。找到了，在字符串 str 的第 6 个字符

处。和数组一样，字符串的起始索引位置是 0。

在字符串 str 中找小写字母 a，也找到了，第一个小写字母 a 在字符串 str 的第 7 个字符处；在字符串 str 中找大写字母 ABC，没有找到。

匹配一个正则表达式，用“=~” ，不能用“==”。 “=~”用来比较是否符合一个正则表达式,返回模式在字符串中被匹配到的位置，否则返回 nil。

不匹配一个正则表达式，用“！~” ，不能用“！=”。 “！~”用来断言不符合一个正则表达式,返回 true,flase。

```
#E8.3-2.rb

str="Hello,kaichuan,Welcome!"

puts   str !~ /kaichuan/        # false
puts   str !~ /a/               # false
puts   str !~ /ABC/             # true
```

假设现在有一篇很短的文章如下：

This is windows2000 or windows98 system.

Windows system is BEST?

Windows2000 running in 12-31-2006,……

我们需要将文章中所有的 windows2000 或者 windows98 换成 Windows XP,不论单词开头大小写，但是不带数字的 windows 不换；并且要把 2006 年 12 月 31 日改成当前时间，如何使用正则表达式来替换呢？

给出例程 E8.3-3.rb 之前，先学习一些烦琐的东西。

一些字符或字符组合在正则表达式中有特殊的意义，分别如下：

特别字符

<table>
<tr><th>特别字符</th><th>描述</th></tr>
<tr><td>( )</td><td>标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。</td></tr>
<tr><td>[]</td><td>范围描述符 (比如,[a - z] 表示在 a 到 z 范围内的一个字母)， 要匹配 [，请使用 \[。</td></tr>
<tr><td>{}</td><td>标记限定符表达式。要匹配 {，请使用 \{。</td></tr>
<tr><td>\</td><td>将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。

例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。</td></tr>
<tr><td>|</td><td>指明两项之间的一个选择。要匹配 |，请使用 \|。</td></tr>
<tr><td>.</td><td>匹配除换行符 \n 之外的任何单字符。要匹配 .，请使用 \。</td></tr>
</table>

非打印字符

非打印字符	描述
\f	匹配一个换页符。等价于 \x0c。
\n	匹配一个换行符。等价于 \x0a。
\r	匹配一个回车符。等价于 \x0d。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'字母或数字;相当于 [0-9A-Za-z]
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'非字母,数字
\d	匹配一个数字字符。等价于 [0-9]。 [0-9]数字;相当于 [0-9]
\D	匹配一个非数字字符。等价于 [^0-9]。非数字字符
\b	退格符 (0x08) (仅在范围描述符内部时)

限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。

* 和 + 限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上一个 ? 就可以实现非贪婪或最小匹配。

限定符	描述
*	前面元素出现 0 或多次。* 等价于{0,}。 例如，zo* 能匹配 "z" 以及 "zoo"。 。要匹配 * 字符，请使用 *。
+	前面元素出现 1 或多次。+ 等价于 {1,}。 例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。 要匹配 + 字符，请使用 \+。
?	前面元素最多出现 1 次;相当于 {0,1}。 例如，”do(es)?” 可以匹配 “do” 或 “does” 中的"do" 。 要匹配 ? 字符，请使用 \?。
{n}	n 是一个非负整数。匹配确定的 n 次。 例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。 例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。
{n,m}	m 和 n 均为非负整数，其中 n <= m。前面元素最少出现 n 次,最多出现 m 次。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。 例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。

定位符

用来描述字符串或单词的边界， ^ 和 $ 分别指字符串的开始与结束，\b 描述单词的前或后边界，\B 表示非单词边界。不能对定位符使用限定符。

定位符	描述
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。

$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'

各种操作符的运算优先级

相同优先级的从左到右进行运算，不同优先级的运算先高后低。各种操作符的优先级从高到低如下：

优先级	操作符	描述
高	\	转义符
	(), []	圆括号和方括号
	*, +, ?, {n}, {n,}, {n,m}	限定符
	^, $,	位置和顺序
低	|	"或"操作

正则表达式强大，但是枯燥。有一个办法，就是等你需要用的时候再来学习。

下面解释例程 E8.3-3.rb，

```
#E8.3-3.rb


strdoc=<<DOC_EOF
This is windows2000 or windows98 system.
```

```
Windows system is BEST?

Windows2000 running in 12-31-2006,……

DOC_EOF


re = /[w|W]indows(?:98|2000) /

strdoc.gsub!(re, "Windows XP ")

re = /[1-9][0-9]\-[1-9][0-9]\-\d\d\d\d/

time = Time.now.strftime("%m-%d-%Y")

strdoc.gsub!(re, time)

puts strdoc
```

运行结果：

```
>ruby E8.3-3.rb

This is Windows XP or Windows XP system.

Windows system is BEST?

Windows XP running in 02-06-2007,……

>Exit code: 0
```

strdoc.gsub!(re, "Windows XP ")，是把字符串 strdoc 里所有匹配正则模式 re 的子串替换为 "Windows XP "。 gsub!是替换所有子串。

strdoc.gsub!(re, time)，是把字符串 strdoc 里所有匹配正则模式 re 的子串替换为字符串 time。

time = Time.now.strftime("%m-%d-%Y")，取出系统当前时间，并且格式化成（ 月-日-年 ）的形式，生成一个字符串 time。

8.4　迭代器、代码块、闭包

在 3.10 节我们已经接触了迭代器和代码块，不过当时没有提到迭代器和代码块的概念。程序 E3.10-7.rb 中

```
(1..9).each {|i| print  i  if  i<7}
```

迭代器 each 是数组类的一个方法；大括号{　}里的代码是代码块，简称块。你可以用大括号{　}将代码组织成块，也可以用 do…end 将代码组织成块。大括号{　}的优先级高于 do…end。

我们来写一个最简单的块；

```
#E8.4-1.rb

def one_block
  yield
  yield
  yield
end

one_block { puts "This is a block. " }
```

运行结果：

```
This is a block.
This is a block.
```

```
This is a block.
```

从程序 E8.4-1.rb 可以看到调用一个块要用关键字 yield。每一次 yield，块就被调用一次。yield 还可以带参数调用块，看程序 E8.4-2.rb ：

```
#E8.4-2.rb


def  one_block
  for num in 1..3
    yield(num)
  end
end


one_block  do |i|
  puts "This is  block #{i}. "
end
```

运行结果：

```
This is  block 1.
This is  block 2.
This is  block 3.
```

一个块可以接收 yield 传来的参数，还可以将结果返回给调用它的方法。到目前为止，实在看不出使用代码块的优势，可以把块里的代码直接写在方法中。如果我们还没有决定块里写什么代码，又或者块里的代码会随着不同的情形而变化，那么就看出代码块的灵活性了。

```
#E8.4-3.rb
```

```
def do_something
   yield
end

do_something do
  (1..9).each {|i| print  i  if  i<5}
  puts
end

do_something do
  3.times { print  "Hi!" }
  puts
end
```

运行结果：

```
>ruby E8.4-3.rb
1234
Hi!Hi!Hi!
>Exit code: 0
```

两次使用方法 do_something，第一次 do_something 遇到 yield，调用了代码块{ 输出 1..9 中小于 5 的数 }；在程序的另一处 do_something 的时候，我们希望做一些不同的事，所以我们写了一个不同于前一次的代码块{ 输出 3 次"Hi!"}。这是一个简单的例子，但是你能发现其中的技巧：先写出方法的大致框架，调用方法的时候才告诉方法要作什么。

虽然与代码块有关联的方法不都是迭代器，但是，迭代器确实是一个与代码块

有关联的方法。让我们为数组类增加一个迭代器 one_by_one；

```
#E8.4-6.rb


class Array
  def   one_by_one
    for i in 0...size
      yield(self[i] )
    end
    puts
  end
end


arr = [1,3,5,7,9]
arr.one_by_one {|k| print   k ,   ", "}      # 1, 3, 5, 7, 9,
arr.one_by_one {|h| print h*h,   ", "}      # 1, 9, 25, 49, 81,
```

代码块是一段代码，相当于一个匿名方法，被调用它的方法所调用。如果我们不仅仅想调用代码块，还想把代码块作为参数传递给其它方法，就要使用闭包了。闭包也是一段代码，一个代码块，而且能够共享其它方法的局部变量。

闭包既然是一段代码，也就有自己的状态，属性，作用范围，也就是一个可以通过变量引用的对象，我们称之为过程对象。一个过程对象用 proc 创建，用 call 方法来调用。

先看一个闭包作为参数传递给其它方法的例子；

```
#E8.4-4.rb


def method(pr)
   puts pr.call(7)
end


oneProc=proc{|k|   k *=3 }
method(oneProc)
```

运行结果：

```
>ruby E8.4-4.rb
21
>Exit code: 0
```

再看一个闭包共享其它方法局部变量的例子；

```
#E8.4-5.rb


def method(n)
   return   proc{|i|   n +=i }
end


oneProc=method(3)
puts oneProc.call(9)            #12
puts oneProc.call(5)            #17
```

方法 method 返回一个 Proc 对象，这个对象引用了这个函数的参数：n 。即

使 n 这个参数在闭包被调用时已经不在自己的作用域里了，这个闭包还是可以访问 n 这个参数，并且和方法 method 共同拥有变量 n 。开始的时候，方法 method 的变量 n 是 3；oneProc.call(9)的时候，oneProc 更新了变量 n，把 n=12 传回给方法 method；oneProc.call(5)的时候，oneProc 取出方法 method 的变量 n=12，更新为 n=17，传回给方法 method 的同时，也把 n=17 作为自己的返回值输出。

第九章 元编程

你编写了一个能够生成其它程序的程序，那么，你就在做元编程的工作。

在 4.3 节我们定义了一个 Person 类， Person 类中有一个 talk 方法，每个 Person 类的实例都可以 talk。可是现实生活中的人不仅能说话，还能大笑，吃饭，演讲，睡觉......

如果在定义 Person 类的时候，就把“大笑，吃饭，演讲，睡觉......”作为方法写在其中，似乎可行，但是人的行为是捉摸不定的，也是层出不穷的，某个人想“呕吐”，偏偏 Person 类没有定义方法“呕吐”，看来只好憋在肚子里了！

Ruby 语言强大的动态特征，赋予了我们灵活地进行元编程的能力。

我们先看程序 E9-2.rb，

```
#E9-2.rb
require "E9-1"
class Person < MetaPerson
end


person1 = Person.new
person2 = Person.new


person1.sleep              #sleep, sleep, sleep...
person1.running            #running, running, running...


person1.modify_method("sleep", "puts 'ZZZ...'")
```

```
person1.sleep              # ZZZ...

person2.sleep              # ZZZ...
```

在程序 E9-2.rb 中，开始的时候，没有写方法 sleep，也没有写方法 running，就直接调用了，后来觉得方法 sleep 的表达不够形象，就修改了 sleep 的方法体。你可能会认为在程序 E9-1.rb 中写了方法 sleep 和方法 running，是不是这样呢？

看程序 E9-1.rb，

```
#E9-1.rb

class MetaPerson
  def MetaPerson.method_missing(methodName, *args)
    name = methodName.to_s
    begin
      class_eval(%Q[
      def #{name}
        puts '#{name}, #{name}, #{name}...'
      end
     ])
    rescue
      super(methodName, *args)
    end
  end

  def method_missing(methodName, *args)
    MetaPerson.method_missing(methodName, *args)
```

```
      send(methodName)
    end

    def MetaPerson.modify_method(methodName, methodBody)
      class_eval(%Q[
        def #{methodName}
          #{methodBody}
        end
       ])
    end

    def modify_method(methodName, methodBody)
      MetaPerson.modify_method(methodName, methodBody)
    end

  end
```

程序 E9-1.rb 是一个能够生成其它程序的程序，你可以在 E9-1.rb 中将类名改换，只要在其它子类继承就可以了。其它子类调用方法时，如果是已有的方法，则正常调用；如果是不存在的方法，就由 E9-1.rb 中的超类自动生成；如果你觉得方法不合你意，你能够很轻易地修改方法体，这个工作也由 E9-1.rb 中的超类自动完成。E9-1.rb 中的超类是元编程的一个简单展现。你加入自己修改后的代码，能够使这个程序更加智能化。

小跋

编程语言，教材里称之为程序设计语言，这本书中，将“程序设计”弱化为“编程”，根源于实践工作中的体会。几年前的某个晚上，在海轮上过夜，当明月无声地升起，我有一种特别的感觉，不是感慨万千，而是心中空空洞洞，一无所想。甲板上的同伴说些什么，我已经听不见了，不因为海浪的颠簸起伏，也不因为周围的一望无际，我无法言语。“大哉，斯矣！”

夜色已深，霓虹消失，浮华散尽，一切都回归本来面目。我在思索，我们应该把肩膀降得低一些，再低一些，好让后面的人踏着向前。明天，太阳照常升起，一切新生力量又将开始新的一天。

我努力使这本书面向没有编程经验的人，可是书中太多的痕迹表明我常常以 Java 的习惯在思考问题，好在多一些角度思考问题并不是坏事。书中许多内容来自网络，我已经没法一一历数出处，再次感谢你和他——生活在网络的人们，再次感谢这个网络世界。

2007 年 1 月 16 日