

一步一步学 RUBY

Ruby 介绍

Ruby 是如何来的

Ruby 的发明者松本行弘 Yukihiro "matz" Matsumoto, 混合了他喜欢的语言 (Perl、Smalltalk、Eiffel、Ada 和 Lisp) 产生了一种具有函数式及指令程序设计特性的新语言。他常说, 他是“试着让 Ruby 更自然, 而不是简单, 就像生活一样”。

除此之外, 他还提到: Ruby 就像人的身体一样, 表面上看来简单, 但是内部却相当的复杂。在 1993 年, 没有人会相信一个由日本业余语言设计者创建的面向对象的语言, 能最终在世界范围内被广泛使用并且变得几乎像 Perl 那样流行。自从 1995 年 Ruby 公开发表以来, Ruby 在全球吸引了许多忠实的程序设计员。在 2006 年, Ruby 被广泛接受。在各大城市都有活跃的使用者并通过社区举办许许多多场爆满的研讨会。

在 TIOBE, 最流行的开发语言排名调查中, Ruby 排名为全球第 11 位。根据这样的成长情况, 他们预测“在半年之中 Ruby 将会进入最受欢迎开发语言的前 10 名。”有越来越多受欢迎的软件如 Ruby on Rails web framework 是使用 Ruby 撰写而成, 也是造成 Ruby 如此快速成长的原因。

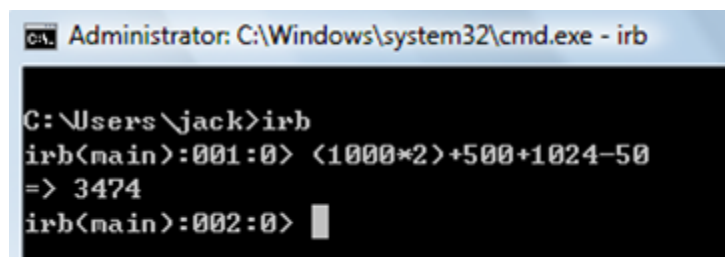
Ruby 是开源软件。不只是免费, 而且可以自由的使用、复制、修改与发布。

准备

首先下载安装 Ruby <http://www.ruby-lang.org/en/downloads/>

安装后, 在命令行 `ruby -v` 检查是否安装正确

ruby 提供了很好的 irb 的环境, 直接在命令行敲入 `irb` 就可以, 然后可以直接运行 ruby 语句, 这对实验 ruby 是非常好的环境, 比如我就经常拿它来当计算器



```
Administrator: C:\Windows\system32\cmd.exe - irb

C:\Users\jack>irb
irb(main):001:0> <1000*2>+500+1024-50
=> 3474
irb(main):002:0> █
```

ScreenShot

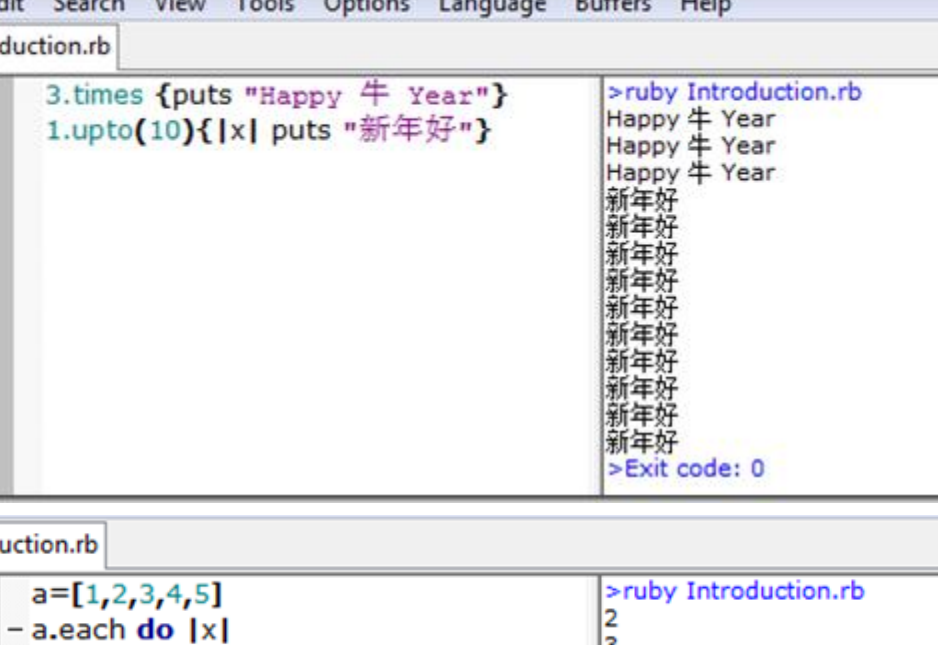
1. 一切皆是对象

```
C:\Users\jack>ruby -v
ruby 1.8.6 (2007-03-13 patchlevel 0) [i386-mswin32]

C:\Users\jack>irb
irb(main):001:0> 1.class
=> Fixnum
irb(main):002:0> 0.1.class
=> Float
irb(main):003:0> true.class
=> TrueClass
irb(main):004:0> false.class
=> FalseClass
irb(main):005:0> nil.class
=> NilClass
irb(main):006:0> █
```

从上图可以看出，从简单类型，到真、假、空都是对象，方法调用参数也是可选的

2. 语句块和迭代器



The screenshot displays the SciTE Ruby IDE interface. The top window, titled "Introduction.rb - SciTE", shows a Ruby script with three lines of code. The first line uses `3.times` to print "Happy 牛 Year". The second line uses `1.upto(10)` to print "新年好" ten times. The output on the right shows the execution results: "Happy 牛 Year" printed three times and "新年好" printed ten times. The bottom window, also titled "Introduction.rb", shows a Ruby script with four lines of code. The first line initializes an array `a = [1, 2, 3, 4, 5]`. The next three lines use `a.each` to iterate over the array, printing each element plus one. The output on the right shows the execution results: the numbers 2, 3, 4, and 5 printed on separate lines. The status bar at the bottom indicates the exit code is 0.

```
1 3.times {puts "Happy 牛 Year"}
2 1.upto(10){|x| puts "新年好"}
3

>ruby Introduction.rb
Happy 牛 Year
Happy 牛 Year
Happy 牛 Year
新年好
新年好
新年好
新年好
新年好
新年好
新年好
新年好
新年好
新年好
>Exit code: 0
```

```
1 a=[1,2,3,4,5]
2 - a.each do |x|
3   puts x+1
4 end

>ruby Introduction.rb
2
3
4
5
6
>Exit code: 0
```

```

Administrator: C:\Windows\system32\cmd.exe - irb

C:\Users\jack>irb
irb(main):001:0> a=[1,2,3,4]
=> [1, 2, 3, 4]
irb(main):002:0> b=a.map {|x| x*2}
=> [2, 4, 6, 8]
irb(main):003:0> c=a.select {|x| x%2==0}
=> [2, 4]
irb(main):004:0>

```

上图显示的是处理集合的威力

```

Introduction.rb - SciTE
File Edit Search View Tools Options Language Buffers Help

1 Introduction.rb
1  - h={
2    :one=>1,
3    :two=>2
4  }
5  h[:one]
6  h[:three]=3
7  - h.each do |key,value|
8    print "#{value}:#{key};"
9  end
10

>ruby Introduction.rb
2:two;3:three;1:one;>Exit code: 0

```

上图显示 Hash 处理的展示

3.表达式和操作符

```

1  x=10000
2  y=20000
3  c=if x<y then x else y end
4  puts c

>ruby Introduction.rb
10000
>Exit code: 0

```

第三行，你看到了吗？

```

1 Introduction.rb
1  puts 1+2
2  puts 1*2
3  puts 1+2==3
4  puts 2**8
5  puts "Hello" + " World!"
6  puts "Happy New Year" *3
7  puts "%s %d %s" % ["jack",20,4
8  puts max=5>4?4:5
9

>ruby Introduction.rb
3
2
true
256
Hello World!
Happy New YearHappy New YearHappy New Year
jack 20 cool
4
>Exit code: 0

```

第 7 行，原来字符串也可以乘

4. 方法

a.自定义方法

1 Introduction.rb	
<pre> 1 # definition function 2 - def SayHello(name) 3 puts name 4 - 3.times { 5 puts "你好!" 6 } 7 end 8 # call 9 SayHello("Jack") 10 </pre>	<pre> >ruby Introduction.rb Jack 你好! 你好! 你好! >Exit code: 0 </pre>

b.为已有的类添加方法

1 Introduction.rb	
<pre> 1 puts Math.sqrt(4) 2 - def Math.AddTwo(x) 3 x+2 4 end 5 puts Math.AddTwo(5) </pre>	<pre> >ruby Introduction.rb 2.0 7 >Exit code: 0 </pre>

5. 赋值语句

1 Introduction.rb	
<pre> 1 x=1 2 y=10 3 x+=1 4 puts "x=#{x}" #x=2 5 y-=1 6 puts "y=#{y}" #y=9 7 a,b=1,2 8 puts "a=#{a},b=#{b}" #a=1,b=2 9 a, b=b,a 10 puts "a=#{a},b=#{b}" #a=2,b=1 11 x,y,z=[1,2,3] 12 puts "x=#{x},y=#{y},z=#{z}" #x=1,y=2,z=3 </pre>	<pre> >ruby Introduction.rb x=2 y=9 a=1,b=2 a=2,b=1 x=1,y=2,z=3 >Exit code: 0 </pre>

为多个变量赋值是比较便利的

下图展示方法是可以返回多个值的

<pre> 1 - def LoveMovieAndSport(name) 2 [name+" Love StarWar", name+ " Love FootBar"] 3 end 4 movie, sport=LoveMovieAndSport("Jack") 5 puts movie 6 puts sport </pre>	<pre> >ruby Introduction.rb Jack Love StarWar Jack Love FootBar >Exit code: 0 </pre>
---	--

6. 后缀

? 检查方法结果是否为真, !立即改变当前对象

```
Administrator: C:\Windows\system32\cmd.exe - irb

C:\Users\jack>irb
irb(main):001:0> a=[4,2,1,3,7]
=> [4, 2, 1, 3, 7]
irb(main):002:0> a.empty?
=> false
irb(main):003:0> a.sort
=> [1, 2, 3, 4, 7]
irb(main):004:0> puts a
4
2
1
3
7
=> nil
irb(main):005:0> a.sort!
=> [1, 2, 3, 4, 7]
irb(main):006:0> puts a
1
2
3
4
7
=> nil
irb(main):007:0>
```

7. Class 和 Module

<pre>1 - class Animal 2 - def GetLeg 3 4 4 end 5 end 6 animal=Animal.new() 7 puts animal.GetLeg</pre>	<pre>>ruby Introduction.rb 4 >Exit code: 0</pre>
---	--

```
1 Introduction.rb
1 - module FourLegAnimal
2 - class Animal
3 -   def GetLeg
4     4
5   end
6 end
7 end
8
9 - module TwoLegAnimal
10 - class Animal
11 -   def GetLeg
12     2
13   end
14 end
15 end
16
17 animal=FourLegAnimal::Animal.new()
18 puts "FourLegAnimal Leg is:" + animal.GetLeg.to_s
19 animal2=TwoLegAnimal::Animal.new()
20 puts "TwoLegAnimal Leg is: " + animal2.GetLeg.to_s

>ruby Introduction.rb
FourLegAnimal Leg is:4
TwoLegAnimal Leg is: 2
>Exit code: 0
```

上图展示如何用 module 来组织类，是不是像 .Net 的命名空间？实际不是，这个我们后面会说到。

8. gem 包管理

ruby 的第三方插件是用 gem 方式来管理，这是一个非常容易发布和共享，一个简单的命令就可以安装上第三方的扩展库

gem install rails 安装 ruby on rails

gem list 列出所有安装的 ruby 包

gem enviroment 显示 gem 的配置

gem update rails 更新指定的包

gem update 更新所有的包

gem update --system 更新 gem 自身

gem uninstall rails 卸载指定的包

变量,常量

本文内容

- 变量
- 变量范围
- 伪变量
- 常量
-

一、变量

1. 变量定义无需指定类型

1 2.rb		
1	x=10	>ruby 2.rb 10 Fixnum Happy New Year String >Exit code: 0
2	a="Happy New Year"	
3	puts x	
4	puts x.class	
5	puts a	
6	puts a.class	

2. 只能包含字母、数字、下划线，但只能以字母或下划线开头
x, y2, _mCount

二、变量范围

1. 局部变量

以小写字母或下划线开头的标识符在 Ruby 中即为局部变量（如果引用未被声明的标识符则会被解释成无参数的方法调用）

局部变量只在代码段类有效

1	- def TestLocalVar()	>ruby 2.rb 200 100 200 >Exit code: 0
2	x=100	
3	puts x	
4	end	
5		
6	x=200	
7	puts x	
8	TestLocalVar()	
9	puts x	

2. 全局变量，实例变量，类变量
\$ 全局变量，所有的实例都能访问
@ 实例变量，只能在实例内部访问
@@ 类变量，这个相当于 C#的类静态成员

<pre> 1 - class Square 2 @@count=0 3 - def initialize(length) 4 @length=length 5 @@count+=1 6 end 7 8 - def area 9 @length*@length #实例变量 10 end 11 12 def self.GetCount 13 @@count #类变量 14 end 15 end 16 x=Square.new(10).area() 17 y=Square.new(5).area() 18 puts x 19 puts Square.GetCount </pre>	<pre> >ruby 2.rb 100 2 >Exit code: 0 </pre>
---	---

三、伪变量

在 Ruby 中有一种被称为伪变量的标识符，伪变量有点像环境变量，同时它也是只读的

13.rb	
<pre> 1 # 当前方法的执行主体 2 puts "#{self}" 3 4 # NilClass类的唯一实例 5 puts "#{nil}" 6 7 # TrueClass 类的唯一实例 8 puts "#{true}" 9 10 # FalseClass 类的唯一实例 11 puts "#{false}" 12 13 # 当前源文件名 14 puts "#{__FILE__}" 15 16 # 当前源文件中的行号 17 puts "#{__LINE__}" </pre>	<pre> >ruby 3.rb main true false 3.rb 17 >Exit code: 0 </pre>

四、常量

以大写字母打头的标识符是常量，对常量进行二次赋值解释器会提示警告，而引用未被赋值的常量实抛出 `NameError` 异常。

在类、模块外部定义的常量属于 `Object`，可以使用“`::常量名`”引用属于 `Object` 的常量，以“`模块名/类名::常量名`”的形式引用外部的常量

<pre>1 SiteName="博客园" #Object常量 2 puts SiteName 3 SiteName="cnblogs" #将会警告 4 5 - module Cnblogs 6 Address="http://www.cnblogs.com" #模块 常量 7 - class NewsChannel 8 Title="Cnblogs-News" #类常量 9 end 10 end 11 puts SiteName #使用Object常量 12 puts Cnblogs::Address #使用Module常量 13 puts Cnblogs::NewsChannel::Title #使用类的常量</pre>	<pre>>ruby 3.rb 博客园 cnblogs http://www.cnblogs.com Cnblogs-News 3.rb:3: warning: already initialized constant SiteName >Exit code: 0</pre>
--	--

注意：Ruby 里常量是可以改变的，但是编译器会发出警告

Ruby 代码注释

本文内容

- 单行注释
- 多行注释
- 特殊处理
- 文档注释

一、单行注释

#开头，以# 开头直到本行末尾的内容都被当作注释

二、多行注释

=begin 注释文字 =end, 注释关键字和注释的文本至少要一个空格的距离

<pre>1 puts "cnblogs" #博客园 2 =begin 3 这里是多行注释 4 测试多行注释 5 =begin 和 =end 和注释之间的距离至少一个空格 6 =end</pre>	<pre>>ruby 3.rb cnblogs >Exit code: 0</pre>
--	---

三、特殊处理

a. 字符串类的#不会当作注释

<pre>1 puts "#test" 2 3</pre>	<pre>>ruby 3.rb #test >Exit code: 0</pre>
-------------------------------	---

b. 正则表达式内的#不会被当作注释

/#This is regular express/

四、文档注释

文档注释，这里指可以根据代码中的注释生成漂亮的文档

我们按如下的格式，一级标题用=,二级标题用==,以此类推

```
1  # = Person 类
2  # 此类是Person的行为
3  # ==Run
4  # 跑的能力
5  # ===SlowRun
6  # ===QuickRun
7  # ==Work
8  # 走的能力
9  - class Person
10 -   def Run
11     end
12
13 -   def Work
14     end
15   end
16
17
18
```

使用 rdoc 命令，可以很用以生成漂亮的注释文档

```
c:\RubyStudy>rdoc comment.rb

                                comment.rb: c..

Generating HTML...

Files:    1
Classes:  1
Modules:  0
Methods:  2
Elapsed:  0.320s
```

默认生成到代码文件所在位置。

l Disk (C:) ▸ RubyStudy ▸

Burn

Name	Date modified	Type
doc	2009/1/22 22:39	File Fo
Comment.rb	2009/1/22 22:37	Ruby P

RDoc Documentation - Mozilla Firefox

文件(F) 编辑(E) 查看(V) 历史(S) 书签(B) 工具(T) 帮助(H)

file:///C:/RubyStudy/doc/index.html

快乐生活, 快乐工作, 快乐编程 - ... × 博客园 - 程序员的网上家园, 最有... × RDoc Documentation ×

Files	Classes	Methods
comment.rb	Person	Run (Person) Work (Person)

comment.rb
Path: comment.rb
Last Update: Thu Jan 22 22:37:34 +0800 2009

Person 类
此类是Person的行为

Run
跑的能力

SlowRun

QuickRun

Work
走的能力

完成

YSlow 0.186s

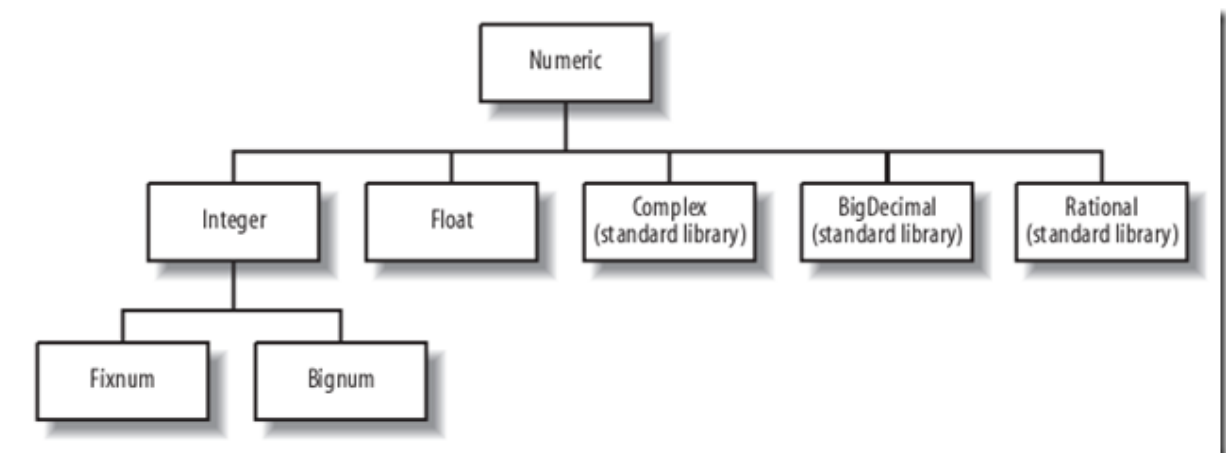
Ruby 标准类型

Ruby 的标准类型包括数字(Numeric)、字符串(String)、区间(Range)以及正则表达式(Regexp)。

- 数字
- 字符串
- 区间
- 正则表达式

数字

Ruby 的数字包括整数(Integer)和浮点数(Float)，这些类型都直接或间接继承自 Numeric。Ruby 的整数并没有长度限制，一定范围内的整数是 Fixnum 对象，当超出该范围则会自动转换成 Bignum。



Fixnum < Integer < Numeric

Bignum < Integer < Numeric

Float < Numeric

```
i=100
puts i.class    #output is Fixnum
a=10000000000
puts a.class    #output is Fixnum

b=1024**1024
puts b.class    #output is Bignum
```

所有的浮点数类型都是 Float，无论它有多长。

```

a=1.0
puts a.class          #output is Float

b=2.5678954334566
puts b.class          #output is Float

c=123123682368236823623868.327923792372392392379237923
puts c.class          #output is Float

```

和所有的语言一样，我们可以通过前导符号标定数字的格式。

- 0 : 八进制。
- 0d : 十进制。
- 0x : 十六进制。
- 0b : 二进制。

```

puts 0123    # output is 83
puts 0d20    # output is 20
puts 0xabc   # output is 2748
puts 0b010   # output is 2

```

我们可以用下划线(_)来分隔数字。

a = 100_874_6 类似我们习惯的 123,456,789。

```

a=100_874_6
puts a          #output is 1008746

```

"?x" 和 "?\cx(或 ?\C-x)" 分别用于显示字符或控制字符的整数值。

```
puts ?a
```

```
puts ?\n
```

```
puts ?\r
```

```
puts ?\ca # Ctrl + a
```

```

puts ?a    #output 97
puts ?\n   #output 10
puts ?n    #output 110
puts ?\r   #output 13
puts ?r    #output 114
puts ?\ca  #output 1

```

字符串

Ruby 字符串由 8bit 字节序列组成。字符串定义方式有很多种

简单点的是用单引号或双引号，当然包括我们熟悉的转义符。

```
s = 'That\'s Right!'
```

```
s = 'escape using "\\''
```

```
s = "Hello, World!"
```

其中双引号字符串内部可以使用 `#{expr}`，类似 C# `String.Format`。

可以使用局部变量，全局变量可省略大括号，也可以表达式。

```
$s = "Wang"
name = "Jack"

puts "Happy New Year #{name}"    # Happy New Year Jack
puts "Yours #s"                  # Yours Wang
puts "1 + 2 + 3 = #{1+2+3}"      # 1 + 2 + 3 = 6
```

另外还有更奇怪的，`%q` 相当于单引号，`%Q` 相当于双引号，还有 `"here documents"`。这些标记都通过特定的分界标记来定义字符串。

```
$s = "Tom"
puts %q/My name is jack/
puts %Q|Your name is #s|
puts <<START, <<END
START
1 + 2 + 3 = #{1+2+3}
1 + 2 + 3 = #{1+2+3}
1 + 2 + 3 = #{1+2+3}
END
```

字符串类 `String` 提供了大量的操作方法，细节可参考类库文档。

区间

区间表示一种序列。在 Ruby 中，使用 `".."` 表示闭区间，而使用 `"..."` 表示半闭半开区间，**区别在于闭区间包含右端的值，而半闭半开区间则不包含。**

```
a=1..9
puts a.min
puts a.max
a.each do |x|
  puts "Hello"+x.to_s
end
```

区间不仅仅是整数，也可以是字符串等。

```
("a".."d").each do |x|
  puts x
end
# Output a b c d
("a..."d").each do |x|
  puts x
end
# Output a b c
```

区间类型 Range 提供了很多操作方法。

```
r=1..5
puts r.min #1
puts r.max #5
puts r.include?(2) #true
puts r==2 #true
```

正则表达式

Ruby 在语言层面提供了正则表达式的支持，我们可以用下面三种方法创建 Regexp 对象。

```

a = Regexp.new(/\d+/)
puts a.class
#Output Regexp

b = /\d+/
puts b.class
#Output Regexp

c = %r{[\d]+}
puts c.class
#Output Regexp

```

我们可以用操作符 "`=~`" 来完成匹配(Match)操作。(操作副 "`!~`" 和此作用相反, 是否定匹配)

```

s = "Number: 100234, ..."

puts s =~ /\d+/

puts $& # 返回匹配的字符串100234

puts $` #键盘波浪线下的字符,返回匹配之前的字符串 "Number: "
puts $' #返回匹配后的字符串 返回匹配之后的字符串 ", ..."

```

"`$~`" 返回 MatchData 对象。

```

s = "Number: 100234,234 ..."

puts s =~ /([\d]+)([\d]+)/

puts $& # 返回匹配的字符串

puts $1 # Group1 "100234"

puts $2 # Group2 "234"

```


Class, Module, Object, Kernel 的关系

1. Class, Module, Object, Kernel 的关系

我们看到 Ruby 里，可以直接写 puts, print 等，感觉像是命令动词一样，这和我们说的 Ruby 里一切都是对象有点冲突，其实我们理解了 Ruby 中 Class, Module, Object, Kernel 的关系，就明白了，通过下面的代码，我们知道 Module 是 Class 的父类。

```
puts Class.ancestors
#Output is Class,Module,Object,Kernel

puts Module.ancestors
#Output is Module,Object,Kernel

puts Object.ancestors
#Output is Object,Kernel

puts Kernel.ancestors
#Output is Kernel
```

```
puts Class.class
#Output is Class
puts Module.class
#Output is Class
puts Object.class
#Output is Class
puts Kernel.class
#Output is Module
```

- 通过上图可以看出 Class, Module, Object 都是 class, 而 Kernel 是 Module。
- Object 是 Ruby 中所有类的父类，Object 混入了 Kernel 这个模块，所以 Kernel 中内建的核心函数就可以被 Ruby 中所有的类和对象访问。
- Object 的实例方法由 Kernel 模块定义。

2. Kernel

我们可以把 Kernel 理解为系统预定义的一些方法，我们可以在所有的对象上使用，使用时不需要使用类型作为前缀，当然我们也可以加上 Kernel, 看起来像 C# 的静态方法。

```
Kernel.puts "Hello"
#Output is Hello
```

Kernel 模块中定义了 private method 和 public method, 我们可以在 irb 里输入 Kernel.methods, Kernel.public_methods, Kernel.private_methods.

对于一个普通的对象，可以直接调用 Kernel 的 public method

而要想调用一个普通对象所包含的 Kernel 的函数，用一般的调用方法无法做到，只有通过 Send 来实现.

方法定义及调用

一、方法定义

过程式方法

Ruby 虽然是一个纯面向对象的语言，但是却允许我们面向过程的方法来使用，我们定义了一个方法后，我们可以直接使用，看下图

```
def bannian
  puts "Happy New Year"
end

def add(a,b)
  a+b
end

bannian
#output is Haapy New Year
puts add(2,3)
#output is 5
puts add(10,5)
#output is 15
```

Kernel 的方法

```
puts "Happy new year"

puts("Happy new year")
```

对象方法

```
class Blogger
  def say(name)
    puts name+",Happy new year"
  end
end

cnblogs=Blogger.new
cnblogs.say("Jack")
#Output is "Jack,Happy new year"
```

类方法

"Jack Wang"是一个字符串对象

```
puts "Jack Wang".length  
puts "Jack Wang".capitalize
```

二、方法调用

方法调用可以带括号，也可以不带

```
puts "Happy new year"  
puts("Happy new year")
```

类方法的调用，像 C# 里的扩展方法

```
puts "Jack Wang".length  
puts "Jack Wang".capitalize
```

方法可以连续调用

```
puts "Jack Wang".upcase.downcase  
#Output is jack wang
```

数学表达式

我们知道，在计算机的最底层，是完全基于数据的，我们编写程序，就是操作数据。

表达式就是能被计算机理解的数字，操作符，变量的联合。比如加、减、乘、除等，这和其他语言一样，下面都是表达式

```
10
1+2
7-2
(7+2)*5-4
"a"+"b"+"c"
10/2
10%2
```

这里我们注意与其它语言区别的是 $10/2$ ， $10/2.0$ 的区别，Ruby 只有在表达式里有一个是浮点数时，结果才是浮点数

```
puts 9/2
#Output is 4
puts 9/2.0
#Output is 4.5

puts 9%2
#Output is 1
puts 9%2.0
#Output is 1.0
```

$0/0$ 会报一个异常，但 $0.0/0.0$ 结果将是 NaN

```
puts 0/0
#It will throw an exception: ZeroDivisionError
puts 0.0/0.0
#Output is NaN
```

负数操作

```
puts 7/-2
#Output is -4
puts 7%-2
#Output is -1
puts -7/2
#Output is -4
puts -7%2
#Output is 1
puts -7/-2
#Output is 3
puts -7%-2
#Output is -1
```

****** 操作符,可计算次方, 以及次方根

```
puts 2**2 #Output is 4
puts 2**3 #Output is 8

puts 10**-1 #Output is 1/10
puts 10**-2 #Output is 1/100
puts 10**-3 #Output is 1/1000
puts 11**-2 #Output is 1/121

puts 9**(1/2.0) #Output is 3.0, Square root
puts 9**(1/2) #Output is 1, Integer division same is x**0, always 1
puts 9**(1/3.0) #Output is 2.0800838230519, cube root

puts 9**(1/4.0) #Output is 1.73205080756888, fourth_root
```

联合计算时, 从右至左

```
puts 4**3**2 #same is 4**9, not 64**2, Output is 262144
```

******的优先级比 $+$, $-$, $*$, $/$ 的优先级高

```
puts 4**3*2 #Output is 128
```

整数可以非常的大, 但浮点数不能大于 **Float::MAX**

```
puts 10**1000 #very large
puts 9.9**1000 #Infinity
```

```
puts Float::MAX #Output is 1.79769313486232e+308
```

简写形式 $x+=y$ 和 $x=x+y$

```
x=10
puts x
#Output is 10
puts x+=1
#Output is 11
puts x-=1
#Output is 10
puts x/=2
#Output is 5
puts x*=3
#Output is 15
```

浮点数运算

因为浮点数有精确值，所以我们取得都是近似值，看下面的代码

```
puts 0.4-0.3
#Output is 0.1
puts 0.4-0.3==0.1
#Output is false
```

字符串表达式

Ruby 中，内置很多对字符串操作的方法，下面我们看看最主要的一些方法

连接字符串

```
puts "JackWang" if "Jack"+"Wang"=="JackWang"  
#Output is JackWang
```

可以对字符串做乘法

```
puts "Happy"*3  
#Output is HappyHappyHappy
```

字符串比较

```
puts "x">"y"  
#Output is false  
puts "y">"x"  
#Output is true
```

我们知道字符串其实存储的是数字，对字符串进行比较其实就比较 **ASCII** 值
用?求字符的 **ASCII** 值

```
puts ?x  
#120  
puts ?y  
#121
```

数字代表的字符

```
puts 120.chr #=>x  
puts 121.chr #=>y
```

将需要替换的表达式的放入#{..}

```
x=5  
y=6  
puts "#{x}+#{y}= #{x+y}"  
#output: 5+6=11
```

字符串也可以插入字符串变量


```
man="Jack"

puts "#{man} is handsome"
#Output: Jack is handsome

puts "#{man} is #{'handsome!'*3}"
#Output: Jack is handsome!handsome!handsome!
```

字符串常用方法

```
puts "Jack"+" Wang"           #Output: Jack Wang
puts "jack".capitalize         #Output: Jack
puts "JACK".downcase           #Output: jack
puts "Jack".chop               #Output: Jac
puts "Jack".hash               #Output: 540559965
puts "Jack".next               #Output: Jack
puts "Jack".reverse            #Output: kcaJ
puts "Jack".sum                #Output: 377
puts "Jack".swapcase           #Output: jACK
puts "Jack".upcase             #Output: JACK
puts "Jack".upcase.reverse     #Output: KCAJ
puts "Jack".upcase.reverse.next #Output: KCAK
```

```
puts "yes"<<"no" #Output: yesno
puts "yes".concat("no") #Output: yesno
puts "jack"[0,2] #Output: ja
puts "jack".slice(0, 2) #Output: ja
puts "hello".count("l") #Output: 2
```

替换字符串的某一范围内的值

```
a="Hello World"
a[1,3]="jack"
puts a
#Output: Hjacko World
```

删除字符

```
puts "Hello".delete("l" "lo")  
#Output: He
```

判断空

```
a=""  
puts a.empty? # true  
b="Jack"  
puts b.empty? # false
```

以 **replace** 来替换字符串中所有与 **pattern** 相匹配的部分

```
puts "hello".gsub(/[aeiou]/, '*')  
# Output: h*ll*
```

判断包含指定的字符串

```
"hello".include? "lo" # true  
"hello".include? "ol" # false
```

按照从左到右的顺序搜索子字符串,并返回搜索到的子字符串的左侧位置。若没有搜索到则返回 **nil**

```
"hello".index('lo')      # 3  
"hello".index('a')      # nil
```

用 **replace** 来替换首次匹配 **pattern** 的部分

```
puts "hello".sub(/[aeiou]/, '*')  
#Output: h*llo
```

对字符串中的各行进行迭代操作,对字符串中的各个字节进行迭代操作

```
"Hi\nRuby".each {|s| puts s}
# Hi Ruby

"Hi\nRuby". each_line { |s| puts s}
# Hi Ruby

"Hi\nRuby". each_byte { |s| puts s}
# 72 105 10 82 117 98 121
```

拆分字符串

```
puts "mellow yellow".split("ello")
#Output: m, w y, w
```

压缩重复的字符串

```
puts "yellow moon".squeeze
#Output: yelow mon
```

删除头部和尾部的所有空白字符。空白字符是指 "t" "r" "n" "f" "v"

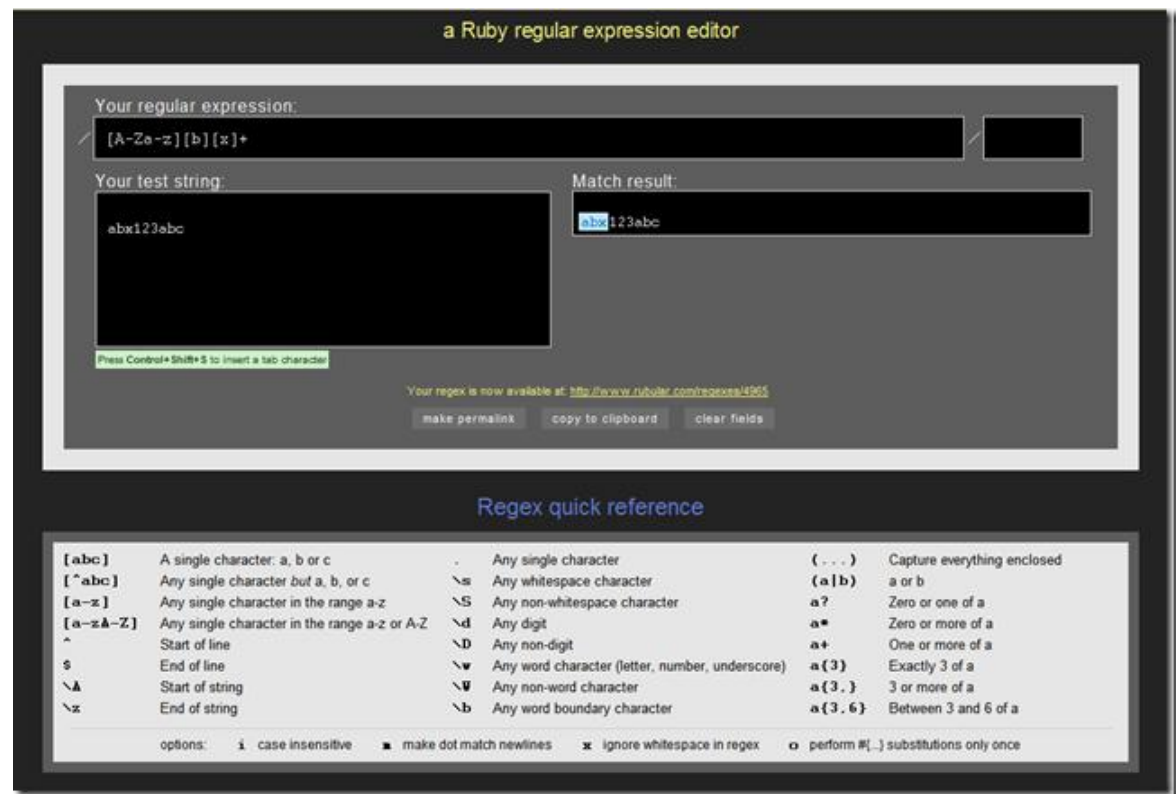
```
puts "    hello    ".strip
#Output: hello
```

若字符串中包含 **search** 字符串中的字符时,就将其替换为 **replace** 字符串中相应的字符

```
puts "hello".tr('aeiou', '*')
#Output: h*ll*
puts "hello".tr('^aeiou', '*')
#Output: *e**o
```

Ruby 正则表达式（上）

先推荐一个在线的 Ruby 正则表达式编辑器 <http://www.rubular.com/>



Ruby 的正则表达式以 `"/"/` 作为构造方法。表达式返回一个 `Regexp` 的对象。

```
puts /a/.class #Output: Regexp
```

一般规则：

`/a/` 匹配字符 `a`。

`/\?/` 匹配特殊字符 `?`。特殊字符包括 `^`, `$`, `?`, `.`, `/`, `\`, `[`, `]`, `{`, `}`, `(`, `)`, `+`, `*`。

`.` 匹配任意字符，例如 `/a./` 匹配 `ab` 和 `ac`。

`/[ab]c/` 匹配 `ac` 和 `bc`, `[]` 之间代表范围。例如：`/[a-z]/`，`/[a-zA-Z0-9]/`。

`/[^a-zA-Z0-9]/` 匹配不在该范围内的字符串。

`/[\d]/` 代表任意数字

`/[\w]/` 代表任意字母，数字或者 `_`

`/[\s]/` 代表空白字符，包括空格，TAB 和换行。

`/[\D]/`，`/[\W]/`，`/[\S]/` 均为上述的否定情况。

高级规则：

`?` 代表 0 或 1 个字符。`/Mrs?\./` 匹配 `"Mr"`，`"Mrs"`，`"Mr."`，`"Mrs."`。

`*` 代表 0 或多个字符。`/Hello*/` 匹配 `"Hello"`，`"HelloJack"`。

`+` 代表 1 或多个字符。`/a+c/` 匹配：`"abc"`，`"abbdrec"` 等等。

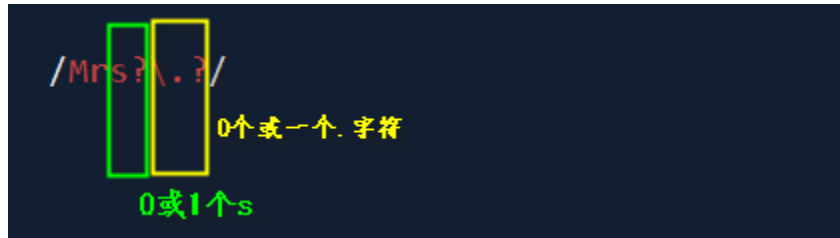
/d{3}/匹配 3 个数字。

/d{1,10}/匹配 1-10 个数字。

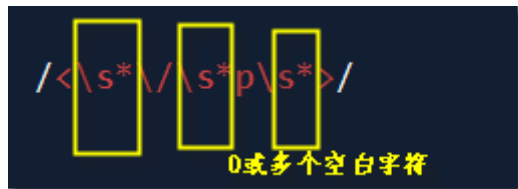
/d{3,}/匹配 3 个数字以上。

/([A-Z]\d){5}/匹配首位是大写字母，后面 4 个是数字的字符串。

下图匹配的是：M 后面是字符 r，后面是 0 或 1 个字符 s，在后面是 0 或 1 个字符"."，匹配 Mr、Mrs、Mr.、Mrs.



下面可以匹配 `</p>`, `</ p>`, `</p >`, `< /p>`.



正则表达式操作

String 和 RegExp 均支持 `=~` 和 `match` 2 个查询匹配方法

```
puts "I can say my abc" =~/abc/  
#Output: 13  
  
a=/abc/.match("I can say my abc, abc I can say") # a is MatchData  
puts a[0]  
#Output: abc
```

可以看出，如果能够匹配，`=~` 返回匹配的字符串位置，而 `match` 返回一个 MatchData 对象。如果不匹配返回 nil。

MatchData 可以取出其中符合各个子匹配的内容。看下面的例子：

```

b=/[A-Za-z]+,[A-Za-z]+,Mrs?\./.match("Jack,Wang,Mrs., better man")
puts b[0]
#Output: Jack,Wang,Mrs

c=/([A-Za-z]+),([A-Za-z]+),Mrs?\./.match("Jack,Wang,Mrs., better man")
puts c[0] #Output: Jack,Wang,Mrs
puts c[1] #Output: Jack
puts c[2] #Output: Wang

d=/([([A-Za-z]+),([A-Za-z]+)),Mrs?\./.match("Jack,Wang,Mrs., better man")
puts d[0] #Output: Jack,Wang,Mrs
puts d[1] #Output: Jack,Wang
puts d[2] #Output: Jack
puts d[3] #Output: Wang

```

m[0]返回匹配主表达式的字符串

下面的方法是等同的:

m[n] == m.captures[n]

Ruby 也自动的为我们填写了一些全局变量，它们以数字做名: \$1,\$2,等等, **\$1** 包含的是正则表达式中从左侧开始的第一对小括号内的子模式所匹配的字符串

我们看出匹配时，是从外到内，从左到右的顺序

其它的一些方法

Ruby 正则表达式（下）

本文内容：

- 贪婪量词和不贪婪量词
- 锚
- 前视断言
- 修饰语
- 正则与字符串的转换
- 正则表达式常用的地方

一、贪婪量词和不贪婪量词

量词*(零个或多个)和+(一个或多个)是贪婪的，它们会匹配尽可能多的字符，**我们可以在*和+后面加一个?**，使它成为非贪婪量词

下面代码是： 1 个或多个字符后接一个感叹号。

```
test="abcdefg!hijkl!!!"
match=/.+!/.match(test)

puts match[0]
#Output: abcdefg!hijkl!!!

limitmatch=/.+?!/.match(test)

puts limitmatch[0]
#output: abcdefg!
```

二、锚

锚是指必须满足一定的条件，才能继续匹配

^ 行首

\$ 行尾

\A 字符串的开始

\z 字符串的结尾

\Z 字符串的结尾(不包括最后的换行符)

\b 单词边界

示例：

```
test="      # Ruby comment line"
a=/^\s*#/.match(test)

puts a[0]
#Output:      #
```

下面演示我们用正则取出注释的内容

```
test="      # Ruby comment line"
a=/^\s*#(.+)/.match(test)

puts a[1]
#Output: Ruby comment line
```

```
b=/\ABetter man/.match("Better man is jack")
puts b[0]
#Output: Better man
```

```
c=/\b\w+\b/.match("!!!!jack****")
puts c[0]
#Output: jack
```

三、前视断言

前视断言表示想要知道下一个指定的是什么，但并不匹配

1. 肯定的前视断言 (?=)

假设我们想要匹配一个数的序列，该序列以一个圆点结束，但并不想把圆点作为模式匹配的一部分

```
str="123 456 789. 012"
m=/\d+(?=\.)/.match(str)
puts m[0]
#Output: 789
```

2. 否定的前视断言 (?!)


```
str="123_456_789. 012"  
m=/\d+(?!\\.)/.match(str)  
puts m[0]  
#Output: 123
```

四、修饰语

修饰语位于正则表达式最结束正则表达式的正斜杠的后面

1. i 使正则表达式对大小写不敏感

```
str="AbcDef"  
m=/abc/i.match(str)  
puts m[0]  
#Output: Abc
```

2. m 使得正则表达式可以和任何字符匹配，包括换行符，通常情况下圆点通配符不匹配换行符

```
str="how are you?\nI am fine, thank you"  
a=/./+.match(str)  
puts a[0]  
#Output: how are you?  
b=/./+m.match(str)  
puts b[0]  
#Output: how are you?  
#          I am fine, than you
```

五、字符串与正则表达式的相互转换

1. 字符串内插进正则表达式

```

str="a.c"
matchdot=/#{str}/
puts matchdot.match("a.c")[0]
#Output: a.c
puts matchdot.match("abc")[0]
#Output: abc

re=/#{Regexp.escape(str)}/
puts re.match("a.c")[0]
#Output: a.c
test=re.match("adc")
puts test[0]
#Output: Nil

```

2. 正则表达式转换成字符串

```

puts /abc/
#Output: (?-mix:abc)

puts /abc/.inspect
#Output: /abc/

```

六、使用正则表达式的常见方法

```

a=["1","a","b","2","5","c"]
result=a.find_all { |x| /\d/.match(x) }
puts result
#Output: 1 ,2, 5

```

```

puts "test 1 2 3 and test 4 5 6".scan(/\d/)
#Output: ["1","2","3","4","5","6"]

```

还有 sub/sub! 和 gsub/gsub!, grep 这些也常用到正则表达式。

控制语句

一、比较语句

大部分和其他的语言一样，这里注意<=>

比较	意思
$x > y$	大于
$x < y$	小于
$x == y$	等于
$x \geq y$	大于等于
$x \leq y$	小于等于
$x \lt;=> y$	如果x大于y, 返回1, 等于返回0, 小于返回-1
$x \neq y$	不等于

```
x=10
y=20

puts x>y
#Output is false
puts x<y
#Output is true
puts x==y
#Output is false
puts x<=y
#Output is tre
puts x>=y
#Output is false
puts x<=>y
#Output is -1
puts x!=y
#output is true
```

比较运算符主要用来判断，返回真假, && 与, ||或

```
age=70
puts "You are old" if age>50 && age<200
#Output is You are old

age=10
puts "You are young or you are old" if age>50 || age<18
#Output is You are young or you are old
```

二、条件修饰语句

这里注意 unless, unless 就是 if not

```
age=20
puts "You are so young" if age<28
#Output is You are so young
age=24
puts "You have one holiday for 5.4" if age>18 && age<28
#Output is You have one holiday for 5.4
age=15
#unless meaning that if not
puts "You have not one holiday for 5.4" unless age>18 && age<28
#Output is You have not one holiday for 5.4
age=20
puts "You are 20" if age==20
#Output is You are 20
```

三、条件语句

如下几种形式

- if
- if ..else.. end
- if..elsif..else..end
- unless(if not)
- case..when

```
age=20
if age<100
  puts "Young"
end
#Output Young
puts "Young" if age<100
#Output Young
age=105
if age<100
  puts "Young"
else
  puts "old"
end
#Output Old
puts "Young" if age<100
#
puts "old" unless age<100
#Output is Old
```

```
x=60
case x
when 80..100
  puts "Top"
when 60..79
  puts "Good"
when 0..60
  puts "bad"
end
#Output: Good
```

四、循环迭代语句

x.times

```

3.times {puts "Happy 0X Year"}
#Output is Happy 0X year, Happy 0X year, Happy 0X year
3.times { |i| puts i }
#Output is 0, 1,2
3.times do
  puts "Happy 0X Year"
end
#Output is Happy 0X year, Happy 0X year, Happy 0X year
3.times do |x|
  puts x
end
#Output is 0, 1,2

```

x.upto(y), x.downto(y), x.step(limit,steplength)看输出，很明白它们的意思

```

1.upto(10) { |i| puts i }
#output is 1 2 3 4 5 6 7 8 9 10
10.downto(5) { |i| puts i }
#output is 10 9 8 7 6 5 4 3 2 1
0.step(50, 10) { |i| puts i }
#output is 0 10 20 30 40 50

```

五：循环语句

Loop

break 跳出整个循环

```

x=1
loop do
  x=x+1
  break if x>10
end

```

next 跳出本次循环

```

y=1
loop do
  y=y+1
  next unless y==10
  break
end

```

while

```
#while..end
n=1
while n<5
  puts n
  n+=1
end

# begin .. end while..
a=1
begin
  puts a
  a+=1
end while a<5
```

until

```
# until..end
b=1
until b>5
  puts b
end

c=1
c+=1 until n==10
puts c

d=1
d+=1 while d<10
```

基于值列表的循环

```
doublesalary=[1000,2000,3000,4000]
for c in doublesalary
  puts c*2
end
#Output: 2000, 4000,6000,8000
```

六、代码块、yield 关键字

在一个方法里放入 yield，那么，当方法执行到这句时，将执行调用这个方法时传入的代码段,类似.net 的委托

1、无参实例

```
def sayhello
  puts "begin:"
  yield
  puts "end"
end

sayhello {puts "Hello World"}
```

Output:
begin:
Hello World
end

代码块

2、有参代码块

```
def addsalary
  yield(5, 7)
end

addsalary{
  |x,y|
  puts x*1000
  puts y*1000
}
```

#Output: 5000 7000

3.有返回值

```
def money
  result=yield(3)
  puts result
end

money {|x| x*1000}
```

#output: 3000

4.执行多个迭代


```
a=[10,20,30]
def doublearray(temps)
  for temp in temps
    c=yield(temp)
    puts c
  end
end

doublearray(a) {|x| x*2}
#output: 20 40 60
```

Ruby 类

一、类的定义、类的实例方法

```
class Person
  def say
    puts "Hello World"
  end
end
```

类的定义

类的方法定义

```
tom=Person.new
```

类的实例化

```
tom.say    #Output:Hello World
```

二、特定对象的方法

在 Ruby 里，我们可以为一个对象单独定义方法

```
class Person
  def say
    puts "Hello World"
  end
end
tom=Person.new
def tom.bye
  puts "Goodbye"
end
tom.say    #Output:Hello World
tom.bye    #Output:Goodbye
```

三、重定义方法

从下面代码可以看出，后面的定义会覆盖前面的定义

```

class Person
  def say
    puts "Hello World"
  end

  def say
    puts "How are you?"
  end
end
tom=Person.new
tom.say    #output: how are you?

```

四、重新打开类

Ruby里我们可以再打开一个类，向这类添加方法，有点像 C# 里的分部类，但不用加特别标记。

我们一般不要拆开类的定义，因为那样不好理解，拆开类的一个原因是将它们分散到多个文件里。

```

class Person
  def say
    puts "Hi"
  end
end
    重新打开类
class Person
  def bye
    puts "Bye"
  end
end

tom=Person.new

tom.say #output: Hi
tom.bye #output: Bye

```

五、实例变量

这个前面有说过，这里补充一下

实例变量主要是用来记录单个对象的状态。Ruby 的实例变量有以下特点

- 以@开头
- 只对单个对象可见
- 不管在类的任何地方定义，在别的地方使用都是同一个

```

class Person
  def say
    @start="today"
  end

  def bye
    puts "we meet #{@start}"
  end
end

tom=Person.new
tom.say
tom.bye
#Output: we meet today

```

六、初始化对象状态

类似其它语言的构造函数

```

class Person
  def initialize(age,sex)
    @age=age
    @sex=sex
  end

  def age
    @age
  end

  def sex
    @sex
  end
end

tom=Person.new(20,"boy")
puts tom.age
puts tom.sex
#Output: 20
#         boy

```

七、更好的属性读写，=号的威力

上面我们可以看到可以用多种方法对属性都写操作，但最熟悉和方便的，还是向其它语言一样，对属性直接赋值，然后读取

```
class Person
  def age=(age)
    @age=age
  end

  def age
    @age
  end
end

tom=Person.new
tom.age=(20)
puts tom.age
#Output: 20
```

方法名 age=

方法调用

语法糖

语法糖就是指特别的规则，不符合常规的写法，Ruby 提供了调用写方法的语法糖，当解释器看到 `age =` 时，会忽略等号前面的空格，并且单参数的方法是可以省略括号，所以也可以这样

```
class Person
  def age=(age)
    @age=age
  end

  def age
    @age
  end
end

tom=Person.new
tom.age=20
puts tom.age
#Output: 20
```

写读

因为有了=的威力，我们可以在赋值时做自己的处理，有点像.net 里的{get;set;}

八、自动生成属性的读写操作 `attr_*`

如果每个简单的属性都像上面那样定义，是一件很繁琐的事情，还好 Ruby 提供了自动生成读写操作的方法，看下表

方法名	效果	例子	等价的代码
attr_reader	读方法	attr_reader :age	def age @age end
attr_writer	写方法	attr_writer :price	def age= (age) @age=age end
attr_accessor	产生读写方法	attr_accessor :age	def age @age end def age= (age) @age=age end
attr	产生读方法和可选的写方法(如果第二个参数是 true)	1. attr :age 2. attr :age, true	1. 参见 attr_reader 2. 参见 attr_accessor

```
class Person
  attr_accessor :age
end

tom=Person.new
tom.age=20
puts tom.age
#Output: 20
```

九、类方法

类是特殊的对象，是唯一可以创建新对象的一类对象

定义是，在方法前加上类名

```
class Person
  attr_accessor :age
  def Person.oldest(*persons)
    persons.sort_by { |x|x.age }.last.age
  end
end

tom=Person.new
tom.age=20
jack=Person.new
jack.age=25
angel=Person.new
angel.age=22

puts Person.oldest(tom,jack,angel)
#Output: 25
```

类方法有两种调用方式：

- Person.oldest()
- Person::oldest()

十、继承

Ruby 里用 < 实现继承

```
class Car
  attr_accessor :wheel
end

class Bus < Car
  attr_accessor :chaircount
end

bus22=Bus.new
bus22.wheel=4
bus22.chaircount=25

puts bus22.wheel
puts bus22.chaircount

#output : 4
#       25
```


Ruby 模块

Class 类是 Module 的子类，类是一种特殊形式的模块,这个可看我的这篇文章 [一步一步学 Ruby\(五\): Class, Module, Object, Kernel 的关系](#)。

我们知道最顶级的类是 Object,那么最顶级的模块就是 Kernel

我们使用模块的主要目的是用来组织代码，模块化代码，有点类似命名空间，但却有很大的不同

一、创建和使用模块

用 module 关键字来定义模块

```
module FirstModule
  def say
    puts "Hello"
  end
end
```

module 没有实例，我们使用时把 module 混合到类中来使用，我们也可以这么理解，把 Module 里的内容拷贝一份放到类里，成为类的一部分

```
module FirstModule
  def say
    puts "Hello"
  end
end

class ModuleTest
  include FirstModule
end

test=ModuleTest.new
test.say
#output: Hello
```

我们可以把模块放到一个单独的文件里，然后使用时进行加载，看下面的示例，假设我们有一个 project.rb 的文件，包含了 Project 模块

```

module Project
  attr_accessor :members
  def initialize
    @members=Array.new
  end
  def add(obj)
    @members.push(obj)
  end

  def remove
    @members.pop
  end
end

```

我们将 Project 混合到 Manager 类里

```

require "project"
class Manager
  include Project
end

test=Manager.new
test.add("jack")
test.add("crystal")
puts test.members[0]
#output: jack

```

注意：在使用 **require** 或 **load** 时，请求加载的内容放到引号里，而 **include** 不是用引号，这是因为 **require** 或 **load** 使用字符串做为参数，而 **include** 使用常量形式的模块名，**require** 和 **load** 使用字符串变量也可以。

二、混合进模块的类的方法查找

```

module M
  def say
    puts "Hello World"
  end
end
class C
  include M
end

class D<C
end

obj=D.new
obj.say
#output: Hello World

```

上面 say 方法查找路径为 D 类-->D 类里包含的模块-->C 类-->C 类包含的模块.....>Object-->Kernel，当找到第一个时，搜索停止。

同名方法的查找,后面覆盖前面的

```

module M
  def say
    puts "Hello from M"
  end
end
module N
  def say
    puts "Hello from N"
  end
end

class C
  include M
  include N
end

test=C.new
test.say
#output: Hello from N

```

用 super 提升查找路径（调用查找路径上下一个匹配的方法），同样我们使用 super 可以调用父类的同名方法，initialize 是自动执行

```

module M
  def say
    puts "say from M"
  end
end
class C
  include M
  def say
    puts "say in C"
    super
    puts "Say from c"
  end
end
obj=C.new
obj.say

#say in C
#say from M
#Say from c

```

调用下一个符合的同名方法

- super 调用时，自动传递当前的参数
- super(), 不传任何参数
- super(a,b,c) 传递指定的参数

三、模块和类可以相互嵌套

module 也可以包含类，但调用时需要这样使用 模块名::类名.new

```

module Human
  class Boy
    def say
      puts "cool"
    end
  end
end

test=Human::Boy.new
test.say

#output: cool

```

self 和作用域

一、默认对象或当前对象是 self

为了知道哪个对象是当前对象，必须知道当前的上下文

1、顶层 self 对象

```
def say
  puts self
end
say #output: main
```

2、类和模块中的 self

```
class C
  puts self ← #output: C
  module M
    puts self ← #output: C::M
  end
  puts self ← #output: C
end
```

3、实例方法的 self

```
class C
  def say
    puts self
  end
end
test=C.new
puts test.say #output: #<C:0x3e41134>
```

内存位置的引用，表示的是一个实例

4、单例方法的 self

```
obj=Object.new

def obj.say
  puts self
end
obj.say #<Object:0x361190>
puts obj #<Object:0x361190>
```

当前对象本身

5.类方法

```

class C
  def self.x
    puts self
  end
end

C.x #output C

```

A diagram with red arrows pointing from the `self` variable in the `def self.x` method to the `class C` definition and the `puts self` output, illustrating that `self` refers to the class object.

6、实例变量和 self

```

class C
  def say
    @hello="Good morning"
    puts @hello ← 3
    puts self ← 4
  end
  @hello="Hi" ← 这个变量属于类，而不是实例
  puts self ← 1
  puts @hello ← 2
end

C.new.say

# C
# Hi
# Good morning
# #<C:0x39a0f58>

```

A diagram with red arrows and numbers indicating variable states:

- `@hello="Good morning"` is labeled '3' and '这个变量属于实例' (this variable belongs to the instance).
- `puts @hello` is labeled '3'.
- `puts self` is labeled '4'.
- `@hello="Hi"` is labeled '1' and '这个变量属于类，而不是实例' (this variable belongs to the class, not the instance).
- `puts self` is labeled '1'.
- `puts @hello` is labeled '2'.

 The output shows the instance variable `@hello` is 'Good morning' and the class variable `@hello` is 'Hi'.

二、Ruby 代码的保护级别

- ruby 默认的方法是公有的,任何地方都可以调用
- `private` 定义私有，只有对象内部可以调用
- `protected` 定义保护的，同一个类的实例之间可调用

```
class Person
  def say
    puts "Hi"
  end

  def age
    puts "20"
  end

  def sex
    puts "boy"
  end
  private :age
  protected :sex
end
```

```
class Person
  private
  def say
    puts "Hi"
  end

  def age
    puts "20"
  end
  protected
  def sex
    puts "boy"
  end
end
```

← 私有方法

← 受保护方法

错误处理和异常

一、常见异常

异常名	常见原因	怎样抛出
RuntimeError	raise 抛出的默认异常	raise
NoMethodError	对象找不到对应的方法	a=Object.new a.jackmethod
NameError	解释器碰到一个不能解析为变量或方法名的标识符	a=jack
IOError	读关闭的流，写只读的流，或类似的操作	STDIN.puts("不能写入")
Errno::error	与文件 IO 相关的一类错误	File.open(-10)
TypeError	方法接受到它不能处理的参数	a=3+"abc"
ArgumentError	传递参数的数目出错	def o(x) end o(1,2,3)

二、捕获异常

用 rescue 捕获异常

```
begin
  result=20/0
  puts result
rescue ZeroDivisionError
  puts "Zero error"
rescue
  puts "Unkonw error"
end
```

3.抛出异常

raise 抛出异常


```
def divide(x)
  raise ArgumentError if x==0
end

begin
  divide(0)
rescue ArgumentError
  puts "ArgumentError"
end

#Output: ArgumentError
```

三、异常保存到变量

```
def divide(x)
  raise ArgumentError if x==0
end

begin
  divide(0)
rescue =>e
  puts e.to_s
end

#Output: ArgumentError
```

四、创建异常类

```
class ThrowExceptionLove<Exception
  puts "some Error"
end

begin
  raise ThrowExceptionLove, "got error"
rescue ThrowExceptionLove=>e
  puts "Error: #{e}"
end

#Output: some Error
#        Error: got error
```

符号

一、符号的定义

符号是 Ruby 内建类 Symbol 的实例，它们的标志是前导冒号。

```
:a
```

```
:person
```

```
:"Hello World"
```

二、字符串与符号可以相互转换

字符串转符号(to_sym 或 intern),符号也可以转换成字符串(to_s)

```
irb(main):001:0> :a
=> :a
irb(main):002:0> "a".to_sym
=> :a
irb(main):003:0> "a".intern
=> :a
irb(main):004:0> :a.to_s
=> "a"
irb(main):005:0>
```

三、符号与字符串的关键不同点

1. 对于给定的文本，只存在一个符号对象，某个相同的写法(:a),表示同一个符号对象，但相同的字符串则表示不同的字符串对象。

```
irb(main):005:0> :a.equal?(:a)
=> true
irb(main):006:0> "a".equal?("a")
=> false
irb(main):007:0>
```

2. 符号是不可变的，不能对符号的进行增、删、该， 但字符串可以。

3. 符号是 Ruby 内部用来存取标识的系统元素，当给一个标量赋值时(a=1), Ruby 就创建一个符号:a

实际上，我们可以看到符号是个引用对象，在内存中只有一份。

Ruby 动态特性

Ruby 中的一切都是动态的，例如，我们可以在程序运行时，动态的添加方法，类等。前面我们已经看到了 Ruby 的动态特性，例如：给单个对象添加方法，重新打开类等。

如果熟悉 Rails，就知道 ActiveRecord 提供基于数据库表的字段名的方法。每一个字段都有一个方法，这个就依赖于 Ruby 的动态特性。

一、单例类的位置

我们可以为一个对象定义只属于自己的方法

```
obj=Object.new
def obj.say
  puts "Hello World"
end
obj.say #输出 Hello World
```

那么单例方法定义在哪里呢？Ruby 把单例方法定义在单例类(singleton class)中. 每一个对象实际上都有两个类：

- 多个实例共享的类
- 单例类

对象调用的方法就是定义在这两个类中的方法，以及祖先类和混含模块中的方法，对象可以调用它所属的类的实例方法，也可以调用单例类中的方法。单例类是匿名的，他们是类对象的实例(Class 类的实例)，但他们是自动生成且没有名字

```
str="Hello World"
class<<str
  def bye
    self+", bye"
  end
end
puts str.bye
输出 "Hello World, bye"
```

在方法查找上，单例方法是最先找到的。

二、eval 方法

这个和其它很多语言一样，具有在运行时执行以字符串形式保存代码的功能。

1. 直接执行代码

```

C:\Users\Jack> irb
irb(main):001:0> eval("1+2")
=> 3
irb(main):002:0> eval <"puts 2+2">
4
=> nil
irb(main):003:0> 

```

2. 运行时提供方法名的例子

```

print "Greeting Method:"
m=gets.chomp
eval("def #{m};puts 'Hello'; end")
eval(m)
输出:Hello

```

如果输入 hi, eval 求值的字符串是 def hi; puts 'Hello'; end

三、eval 的危险性

eval 很强大,但是它也潜在着危险,这个有点像 sql 注入

假如,上面我们输入的不是 hi 而是下面的内容

```

hi; end; system("rm -rf /*"); #
eval 求值以后是这样的
def hi; end; system("rm -rf /*"); # puts 'Hello'; end

```

求值的结果是: #后面的所有内容会被作为注释忽略掉。使用 system 命令试图删除系统所有文件

Ruby 有一个全局变量 \$SAFE (取值范围是 0 到 4)、以获取对如非法写文件这一类危险的防护。

四、instance_eval

该方法把 self 变为 instance_eval 调用的接收者,对字符串或代码块进行求值

```

p self
a=[]
a.instance_eval(p self)
输出:
main
[]

```

instance_eval 常用于访问其它对象的私有数据,特别是实例变量

```

class C
  def initialize
    @a=1
  end
end
c=C.new

```

```
c.instance_eval {puts @a}
```

五、class_eval

class_eval 可进入类定义体中

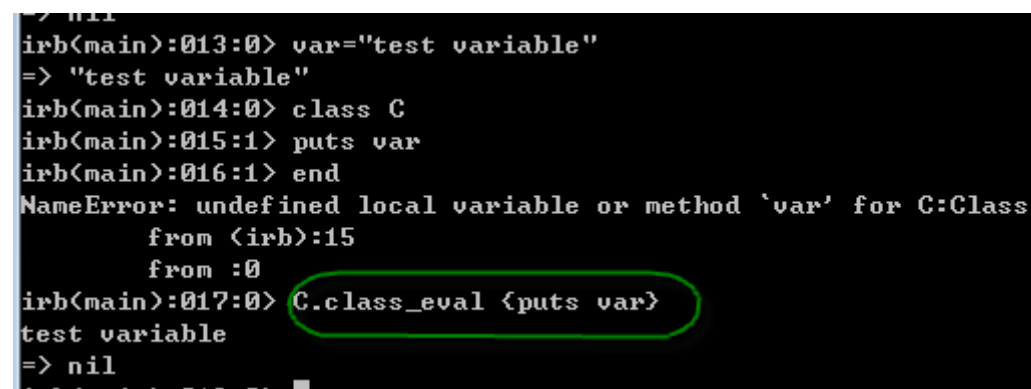
```
c=Class.new
c.class_eval do
  def some_method
    puts "created in class_eval"
  end
end

c=c.new
c.some_method
```

利用 class_eval 可以访问外围作用域的变量。

```
var="test variable"
class C
  puts var
end
C.class_eval {puts var}
```

变量 var 在标准的类定义体的作用域之外，但是在 class_eval 的代码块的作用域之内



```
> nil
irb(main):013:0> var="test variable"
=> "test variable"
irb(main):014:0> class C
irb(main):015:1> puts var
irb(main):016:1> end
NameError: undefined local variable or method `var' for C:Class
    from <irb>:15
    from :0
irb(main):017:0> C.class_eval {puts var}
test variable
=> nil
```

当在 class_eval 的块中定义一个实例方法时，又有不同

```
var="test"
class C
end
C.class_eval {def hi; puts var; end}
c.new.hi
# undefined local variable or method `c' for main:Object (NameError)
```

但我们可以使用另外一种方法

```
C.class_eval {define_method("hi") { puts var}}
```

六、Proc 对象

```
pr=Proc.new {puts "Hello from inside of proc block"}  
pr.call  
#输出: Hello from inside of proc block
```

1、做为闭包的 Proc 对象

Proc 对象随身携带了它的上下文，下面上下文包含一个变量 a，该变量被赋值为一个特殊的字符串保存在 Proc 对象中。像这样带着产生它的上下文信息的一段代码被称为闭包。产生一个闭包就像是打包一件行李：任何时候打开行李，都包含打包进去的东西，在打开一个闭包时（通过调用它），它包含产生它的时候你放进去的东西。

```
def call_proc(pr)  
  a="Jack"  
  puts a  
  pr.call  
end  
a="Tom"  
pr=Proc.new {puts a}  
pr.call  
call_proc(pr)  
#输出: Tom  
#      Jack  
#      Tom
```

2. Proc 对象的参数

产生 Proc 对象时所提供的代码块可以接收参数

```
pr=Proc.new {|x| puts "#{x} is better man"}  
pr.call("Jack")  
输出:Jack is better man
```

七、匿名函数(lambda)

```
lam=lambda {puts "Hello World"}
```

```
lam.call  
输出: Hello World
```

lambda 不是 Lambda 类的对象，他们是 Proc 类的对象

```
lam.class  
输出: Proc
```

和所有的 Proc 对象一样，lambda 是闭包；他们随身携带了生成他们的局部的上下文环境
lambda 生成的 Proc 对象和用 Proc.new 生成的对象之间的差别与 return 有关，lambda
中的 return 从 lambda 返回，而 Proc 中的 return 从外围方法返回

```
def test_return  
  l=lambda {return}  
  l.call  
  puts "I am here"  
  p=Proc.new {return}  
  p.call  
  puts "bye"  
end  
test_return  
输出: "I am here"
```

八、再论代码块

可以在方法中吧代码块转换成 Proc 对象，可以通过参数中最后一个变量，且必须&开头来捕获代码块

```
def say_block(&block)  
  block.call  
end  
say_block {puts "How are you?"}  
#output: how are you?  
def test_block(x,y, &block)  
  puts x+y  
  block.call  
end  
test_block(1,2) {puts "Hi"}  
#output: 3  
#      Hi  
也可以将 Proc 对象或 lambda 转换为代码块  
def say_block(&block)  
  block.call
```



```
end
```

```
lam=lambda {puts "Hello world"}
```

```
say_block(&lam)
```

```
#output: Hello world
```

数组

一、数组定义和基本操作

1. 数组的定义和其它语言一样，Ruby 里的数组并不限定类型。

```
x=[1, 2, 3, 4]
puts x[2]      #输出 3
x[2]+=1
puts x[2]      #输出 4
x[2]="Jack"*3  #输出 JackJackJack
```

2. 数组可以不设初始值, 可以使用<< 和 pop 的操作, 后进先出

```
x=[]
x<<"Jack"
x<<"Tom"
x<<"Crystal"
puts x[2]      #输出 Crystal
x.pop
puts x[x.length-1] #输出 Tom
```

3. 可以把数组连接在一起

```
x=["Jack", "Better", "Man"]
puts x.join
#输出 JackBetterMan
x=["Jack", "Better", "Man"]
puts x.join(', ')
#输出: Jack, Better, Man
```

4. 字符串拆分为数组

```
puts "Jack is better man".scan(/\w/).join(", ")
#输出 J,a,c,k,i,s,b,e,t,t,e,r,m,a,n
puts "www.cnblogs.com".split(/\./).inspect
#输出:["www", "cnblogs", "com"]
```

Insepct 是显示一个对象的变量, 用 P 也可以

<http://blog.12spokes.com/articles/2008/03/15/ruby-tip-stop-using-puts-object-in-spect%E2%80%94use-p-instead>

```
p "Jack Tom Crystal".split(/\s/)
```

```
#输出 ["Jack", "Tom", "Crystal"]
```

二、数组方法

1. 数组迭代

```
[1, "jack", 2.0, 4].each{|x| print x.to_s+"!"}  
#输出 1!jack!2.0!4!
```

2. 批量改变数组的值 collect

```
p [1, "jack", 2.0, 4].collect{|x| x*2}  
#输出: [2, "jackjack", 4.0, 8]
```

3. 数组合并

```
a=[1, 2, 3]  
b=["jack", "crystal"]  
c=a+b  
p c  
#输出: [1, 2, 3, "jack", "crystal"]
```

4. 数组相减

```
a=[1, 2, 3, 4, 5]  
b=[2, 3]  
c=a-b  
p c  
#输出 [1, 4, 5]
```

5. 检查数组是否为空

```
x=[]  
puts "It's empty" if x.empty?  
#输出: It's empty
```

6. 检查数组是否包含某值

```
x=[1, 2, 3]  
puts x.include?("jack")  
#输出:false  
puts x.include?(2)  
#输出: true
```

7. first 和 last 存取元素

```
x=[1, 2, 3, 4]
puts x.first
#output: 1
puts x.last
#output: 4
puts x.first(2).join("-")
#output: 1-2
puts x.last(2).join("-")
#output: 3-4
```

8. 反转数组

```
x=[1, 2, 3, 4]
puts x.reverse.inspect
#output:[4, 3, 2, 1]
```

Hash

一、Hash 的定义和赋值

Hash 代表键值对的集合，Ruby 里的键值可以是任意类型，字符串，数字，甚至是数组

```
dictionary={'Boy'=>'Jack','Girl'=>'Crystal'}
puts dictionary.size  #=> 2

puts dictionary['Boy']  #=>Jack

dictionary['Boy']='Tom'
puts dictionary['Boy']  #=>Tom

dict={2=>'a', 3=>'b'}
puts dict[2]  #=>a

dict={ [1,2]=>'ab', ["a",2]=>'cd' }
puts dict[["a",2]]  #>cd
```

二、Hash 的基本方法

1. 迭代 Hash 的元素

```
a={"username"=>"jack","password"=>"1234"}
a.each{|key,value| puts "key is #{key},value is #{value}"}
#output: key is username,value is jack
#         key is password,value is 1234
```

2. 检索 Key 值

```
puts a.keys.inspect  #output: ["username", "password"]
```

3. 删除 Hash 的元素

```
a={"username"=>"jack","password"=>"1234","code"=>"abc"}
a.delete("username")
puts a.inspect  #output: {"code"=>"abc", "password"=>"1234"}
a.delete_if { |key,value| value=="1234" }
puts a.inspect  #output: {"code"=>"abc"}
```

三、Hash 内部的 Hash

```
friends={  
  "Tom"=>{"gender"=>"boy", "age"=>24},  
  "Crystal"=>{"gender"=>"boy", "age"=>24},  
  "Bruce"=>{"gender"=>"girl", "age"=>28, "likebooks"=>{1=>'.net', 2=>'Java'}}  
}  
  
puts friends["Tom"]["gender"] #ouptut: gender  
puts friends["Bruce"]["likebooks"][2] #output: Java
```

文件使用

一、新建文件

?

```
1 f=File.new(File.join("C:", "Test.txt"), "w+")
2 f.puts("I am Jack")
3 f.puts("Hello World")
```

文件模式

"r" : Read-only. Starts at beginning of file (default mode).

"r+" : Read-write. Starts at beginning of file.

"w" : Write-only. Truncates existing file to zero length or creates a new file for writing.

"w+" : Read-write. Truncates existing file to zero length or creates a new file for reading and writing.

"a" : Write-only. Starts at end of file if file exists; otherwise, creates a new file for writing.

"a+" : Read-write. Starts at end of file if file exists; otherwise, creates a new file for reading and writing.

"b" : (DOS/Windows only.) Binary file mode. May appear with any of the key letters listed above

二、读取文件

?

```
1 file=File.open(File.join("C:", "Test.txt"), "r")
2 file.each { |line| print "#{file.lineno}. ", line }
3 file.close
```

输出:

1.白日依山尽

2.黄河入海流

3.欲穷千里目

4.更上一层楼

三、新建、删除、重命名文件

?

```
1 File.new( "books.txt", "w" )
2 File.rename( "books.txt", "chaps.txt" )
3 File.delete( "chaps.txt" )
```

三、目录操作

?

1	创建目录
2	Dir.mkdir("c:/testdir")
3	
4	#删除目录
5	Dir.rmdir("c:/testdir")
6	
7	#查询目录里的文件
8	p Dir.entries(File.join("C:", "Ruby")).join(' ')
9	
10	#遍历目录
11	Dir.entries(File.join("C:", "Ruby")).each {
12	e puts e
13	}

输出:

?	
1	"C:/studyruby"
2	"c:/ruby"

查看目录信息

?	
1	d:\Study\rubysample>irb
2	irb(main):001:0> dir=Dir.open(File.join("C:", "Ruby"))
3	=> #
4	irb(main):002:0> dir.path
5	=> "C:/Ruby"
6	irb(main):003:0> dir.tell
7	=> 0
8	irb(main):004:0> dir.read
9	=> "."
10	irb(main):005:0> dir.rewind
11	=> #
12	irb(main):006:0> dir.each{ e puts e}
13	.
14	..
15	bin
16	ChangeLog.txt
17	doc
18	lib
19	LICENSE.txt
20	man
21	MANIFEST
22	misc
23	README.1st
24	ReleaseNotes.txt

25	ruby.ico
26	rubyopt.del
27	rubyw.ico
28	samples
29	scite
30	share
31	src
32	uninstall.exe
33	=> #
34	irb(main):007:0>