Ruby-15

2021年8月27日 11:00

https://www.ruby-lang.org/en/documentation/quickstart/

《Ruby in Twenty Minutes》

```
irb(main):001:0> "Hello World"
=> "Hello World"

irb(main):002:0> puts "Hello World"
Hello World
=> nil

irb(main):003:0> 3+2
=> 5

irb(main):005:0> 3**2
=> 9

irb(main):006:0> Math.sqrt(9)
=> 3.0

irb(main):010:0> def hi
irb(main):011:1> puts "Hello World!"
irb(main):012:1> end
=> :hi
```

。。不，这个是返回值是 这样的。

```
irb(main):013:0> hi
Hello World!
=> nil
irb(main):014:0> hi()
Hello World!
=> nil

irb(main):015:0> def hi(name)
irb(main):016:1> puts "Hello #{name}!"
irb(main):017:1> end
=> :hi
irb(main):018:0> hi("Matz")
Hello Matz!
=> nil

irb(main):019:0> def hi(name = "World")
```

```
irb(main):020:1> puts "Hello #{name.capitalize}!"
irb(main):021:1> end
=> :hi
irb(main):022:0> hi "chris"
Hello Chris!
=> nil
irb(main):023:0> hi
Hello World!
=> nil


irb(main):024:0> class Greeter
irb(main):025:1>   def initialize(name = "World")
irb(main):026:2>     @name = name
irb(main):027:2>   end
irb(main):028:1>   def say_hi
irb(main):029:2>     puts "Hi #{@name}!"
irb(main):030:2>   end
irb(main):031:1>   def say_bye
irb(main):032:2>     puts "Bye #{@name}, come back soon."
irb(main):033:2>   end
irb(main):034:1> end
=> :say_bye
```

Also notice @name. This is an instance variable, and is available to all the methods of the class.

```
irb(main):035:0> greeter = Greeter.new("Pat")
=> #<Greeter:0x16cac @name="Pat">
irb(main):036:0> greeter.say_hi
Hi Pat!
=> nil
irb(main):037:0> greeter.say_bye
Bye Pat, come back soon.
=> nil

irb(main):038:0> greeter.@name
SyntaxError: (irb):38: syntax error, unexpected tIVAR, expecting '('
。。private

irb(main):039:0> Greeter.instance_methods
=> [:say_hi, :say_bye, :instance_of?, :public_send,
    :instance_variable_get, :instance_variable_set,
    :instance_variable_defined?, :remove_instance_variable,
    :private_methods, :kind_of?, :instance_variables, :tap,
    :is_a?, :extend, :define_singleton_method, :to_enum,
    :enum_for, :<=>, :===, :=~, :!~, :eql?, :respond_to?,
    :freeze, :inspect, :display, :send, :object_id, :to_s,
    :method, :public_method, :singleton_method, :nil?, :hash,
```

```
    :class, :singleton_class, :clone, :dup, :itself, :taint,
    :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods,
    :protected_methods, :frozen?, :public_methods, :singleton_methods,
    :!, :==, :!=, :__send__, :equal?, :instance_eval, :instance_exec, :__id__]


irb(main):040:0> Greeter.instance_methods(false)
=> [:say_hi, :say_bye]

irb(main):041:0> greeter.respond_to?("name")
=> false
irb(main):042:0> greeter.respond_to?("say_hi")
=> true
irb(main):043:0> greeter.respond_to?("to_s")
=> true


irb(main):044:0> class Greeter
irb(main):045:1>   attr_accessor :name
irb(main):046:1> end
=> nil
```

In Ruby, you can reopen a class and modify it. The changes will be present in any new objects you create and even available in existing objects of that class.

```
irb(main):047:0> greeter = Greeter.new("Andy")
=> #<Greeter:0x3c9b0 @name="Andy">
irb(main):048:0> greeter.respond_to?("name")
=> true
irb(main):049:0> greeter.respond_to?("name=")
=> true
irb(main):050:0> greeter.say_hi
Hi Andy!
=> nil
irb(main):051:0> greeter.name="Betty"
=> "Betty"
irb(main):052:0> greeter
=> #<Greeter:0x3c9b0 @name="Betty">
irb(main):053:0> greeter.name
=> "Betty"
irb(main):054:0> greeter.say_hi
Hi Betty!
=> nil
```

Using attr_accessor defined two new methods for us, name to get the value, and name= to set it.

To quit IRB, type “quit”, “exit” or just hit Control-D.

```ruby
#!/usr/bin/env ruby

class MegaGreeter
  attr_accessor :names

  # Create the object
  def initialize(names = "World")
    @names = names
  end

  # Say hi to everybody
  def say_hi
    if @names.nil?
      puts "..."
    elsif @names.respond_to?("each")
      # @names is a list of some kind, iterate!
      @names.each do |name|
        puts "Hello #{name}!"
      end
    else
      puts "Hello #{@names}!"
    end
  end

  # Say bye to everybody
  def say_bye
    if @names.nil?
      puts "..."
    elsif @names.respond_to?("join")
      # Join the list elements with commas
      puts "Goodbye #{@names.join(", ")}.  Come back soon!"
    else
      puts "Goodbye #{@names}.  Come back soon!"
    end
  end
end


if __FILE__ == $0
  mg = MegaGreeter.new
  mg.say_hi
  mg.say_bye

  # Change name to be "Zeke"
  mg.names = "Zeke"
```

```
  mg.say_hi
  mg.say_bye

  # Change the name to an array of names
  mg.names = ["Albert", "Brenda", "Charles",
              "Dave", "Engelbert"]
  mg.say_hi
  mg.say_bye

  # Change to nil
  mg.names = nil
  mg.say_hi
  mg.say_bye
end
```

Save this file as “ri20min.rb”, and run it as “ruby ri20min.rb”. The output should be:

```
Hello World!
Goodbye World.  Come back soon!
Hello Zeke!
Goodbye Zeke.  Come back soon!
Hello Albert!
Hello Brenda!
Hello Charles!
Hello Dave!
Hello Engelbert!
Goodbye Albert, Brenda, Charles, Dave, Engelbert.  Come
back soon!
...
...
```

duck type

__FILE__ is the magic variable that contains the name of the current file. $0 is the name of the file used to start the program.

https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-c-and-cpp/

To Ruby From C and C++

Ruby itself is written in C.

Ruby doesn’t have ++ or --.
You’ve got __FILE__ and __LINE__.
Strings are mutable.

<< is often used for appending elements to a list.
with Ruby you never use -> it’s always just . .

Inheritance syntax is still only one character, but it’s < instead of :.
。。。我竟然不记得 cpp的继承了。。 确实是 类定义的时候用 :

All variables live on the heap. Further, you don’t need to free them yourself—the garbage collector takes care of that.

It’s require ’foo’ instead of #include <foo> or #include "foo".

Parentheses for method (i.e. function) calls are often optional.

The do keyword is for so-called “blocks”. There’s no “do statement” like in C.

When tested for truth, only false and nil evaluate to a false value. Everything else is true (including 0, 0.0, and "0").

Arrays just automatically get bigger when you stuff more elements into them.

The “constructor” is called initialize instead of the class name.

All methods are always virtual.

“Class” (static) variable names always begin with @@ (as in @@total_widgets).

You don’t directly access member variables—all access to public member variables (known in Ruby as attributes) is via methods.

It’s self instead of this.

Some methods end in a ’?’ or a ’!’. It’s actually part of the method name.

There’s no multiple inheritance per se. Though Ruby has “mixins” (i.e. you can “inherit” all instance methods of a module).

There’s only two container types: Array and Hash.

http://ruby-doc.com/docs/ProgrammingRuby/
。。这个是ruby1.6， 2000 9月发行的。。

https://ruby-doc.org/core-3.0.2/

。。。这个有点邪门，硬是找不到 介绍语言的文档，都是介绍 api的。。

。。就用1.6的 随便看看吧。 可能语言方面没有大改吧。。不过也是，就 一些关键字的用法， 改不了的。而且应该都是向后/前兼容的。

```
Ruby.new

"gin joint".length      »     9
"Rick".index("c")       »     2
-1942.abs    »     1942
sam.play(aSong)    »     "duh dum, da dum de dum ..."


def sayGoodnight(name)
  result = "Goodnight, " + name
  return result
end


# Time for bed...
puts sayGoodnight("John-Boy")
puts sayGoodnight("Mary-Ellen")


The following lines are all equivalent.
puts sayGoodnight "John-Boy"
puts sayGoodnight("John-Boy")
puts(sayGoodnight "John-Boy")
puts(sayGoodnight("John-Boy"))
```

Ruby does with double-quoted strings is expression interpolation. Within the string, the sequence #{ expression } is replaced by the value of expression.
。。双引号的string 是一个 表达式插值。 #{expression} 被替换为 expression的值。

```
def sayGoodnight(name)
  result = "Goodnight, #{name}"
  return result
end
```

As a shortcut, you don't need to supply the braces when the expression is simply a

global, instance, or class variable
。。如果 表达式 是一个 全局，实例，类 的变量。 可以不需要 {}

Arbitrarily complex expressions are allowed in the #{...} construct.
。。可以嵌套任意复杂表达式。

The value returned by a Ruby method is the value of the last expression evaluated
。最后一个表达式的值 就是方法的返回

```
def sayGoodnight(name)
  "Goodnight, #{name}"
end
```

the first characters of a name indicate how the name is used. Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore. Global variables are prefixed with a dollar sign ($), while instance variables begin with an ``at'' sign (@). Class variables start with two ``at'' signs (@@). Finally, class names, module names, and constants should start with an uppercase letter.
。。名字的第一个字符表明了 名字怎么用。
。。本地变量，方法参数，方法名 是以 小写字母或 下划线
。。全局变量是 $ 开头。。。。。这个是指声明的时候就美刀，还是 使用的时候 就需要加一个$ ？ (就是 $是名字的一部分，还是 一种标记 表明 后续是全局变量)。。应该是名字的一部分。 参照下面的 @ 。。 类.xx ，这个xx是方法，等于不带@的属性。。
。。实例变量 @开头
。。类变量 @@开头。
。。类名，模块名，常量， 大写字母开头。

Example variable and class names

Variables Local	Global	Instance	Class	Constants and Class Names
name	$debug	@name	@@total	PI
fishAndChips	$CUSTOMER	@point_1	@@symtab	FeetPerMile
x_axis	$_	@X	@@N	String
thx1138	$plan9	@_	@@x_pos	MyClass
_26	$Global	@plan9	@@SINGLE	Jazz_Song

Arrays and Hashes
都是带下标的集合，都保存 对象的集合，通过key访问。
数组的key 是integer。
hash的key 可以是任何对象。
都会自动增长。

同一个 数组和hash 可以存放不同类型的元素。

[]创建和初始化一个 数组。

```
a = [ 1, 'cat', 3.14 ]   # array with three elements
# access the first element
a[0]  »     1
# set the third element
a[2] = nil
# dump out the array
a     »     [1, "cat", nil]
```

```
empty1 = []
empty2 = Array.new
```

。创建空数组

```
a = %w{ ant bee cat dog elk }
a[0]  »     "ant"
a[3]  »     "dog"
```

。。可以省略 "", 。。。 这个目前是 string数组

hash字面量 使用{}。

```
instSection = {
  'cello'     => 'string',
  'clarinet'  => 'woodwind',
  'drum'      => 'percussion',
  'oboe'      => 'woodwind',
  'trumpet'   => 'brass',
  'violin'    => 'string'
}
```

。。不知道 "z" 'z' '''z''' 有没有区别。

```
instSection['oboe']     »     "woodwind"
instSection['cello']    »     "string"
instSection['bassoon']  »     nil
```

hash不包含key时，返回nil。 这个返回值可以设定：

```
histogram = Hash.new(0)
histogram['key1']       »     0
histogram['key1'] = histogram['key1'] + 1
histogram['key1']       »     1
```

Control Structures

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
  puts "Enter a number"
end
```

```
while weight < 100 and numPallets <= 30
  pallet = nextPallet()
  weight += pallet.weight
  numPallets += 1
end
```

一种简写，如果 if/while的 body 是一个 single 表达式：
直接写表达式，后面跟一个 if/while。

```
if radiation > 3000
  puts "Danger, Will Robinson"
end

puts "Danger, Will Robinson" if radiation > 3000

while square < 1000
   square = square*square
end

square = square*square  while square < 1000
```

Regular Expressions

/pattern/ 声明一个 正则表达式。 这个re是对象，能像对象一样被操作。。

匹配the text ``Perl'' or the text ``Python'':
/Perl|Python/
/P(erl|ython)/

We can put all this together to produce some useful regular expressions.

```
/\d\d:\d\d:\d\d/     # a time such as 12:34:56
/Perl.*Python/       # Perl, zero or more other chars, then Python
/Perl\s+Python/      # Perl, one or more spaces, then Python
/Ruby (Perl|Python)/ # Ruby, a space, and either Perl or Python
```

=~ 用来匹配。
如果找到，返回 开始下标
否则，返回 nil。

这意味着，你可以 使用 re 作为 if/while 的条件。

下面的代码，输出msg，如果 string 包含 Perl 或 Python

```
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end
```

string中匹配re的部分 会被替换

```
line.sub(/Perl/, 'Ruby')    # replace first 'Perl' with 'Ruby'
line.gsub(/Python/, 'Ruby') # replace every 'Python' with 'Ruby'
```

Blocks and Iterators

code blocks: chunks of code that you can associate with method invocations, almost as if they were parameters.
。。代码块， 可以通过 方法调用 associate 的代码块， 即使 它们是 参数。

Code blocks are just chunks of code between braces or do...end.

```
{ puts "Hello" }       # this is a block

do                     #
  club.enroll(person)  # and so is this
  person.socialize     #
end                    #
```

do… end 或 {} 包围的 代码块

创建block后，你可以associate 它 通过 调用一个方法。。。
那个方法能invoke block 一次或多次 通过 Ruby的 yield 语句。

下面的例子，定义了一个方法 调用yield 2次。
我们调用它，放一个block 在同层。

```
def callBlock
  yield
  yield
end
callBlock { puts "In the block" }
```

生成：

```
In the block
```

In the block

。。黄色是block。

能提供 yield 调用的 形参。 这些形参会传入到 block。
在block中， 通过 列举在 '|'之间的 参数 来 接受变量。

```
  def callBlock
    yield ,
  end

  callBlock { |, | ... }
```

Code blocks are used throughout the Ruby library to implement iterators: methods that return successive elements from some kind of collection, such as an array.
。。。这是什么英语。。
。。整个Ruby库都使用代码块来实现迭代器：从某种集合（如数组）返回连续元素的方法。

```
a = %w( ant bee cat dog elk )    # create an array
a.each { |animal| puts animal }  # iterate over the contents
```

产生：

```
ant
bee
cat
dog
elk
```

each 方法 类似：

```
# within class Array...
def each
  for each element
    yield(element)
  end
end
```

。。感觉像是一种占位符， 可以替换为 代码 的那种占位符。

```
[ 'cat', 'dog', 'horse' ].each do |animal|
  print animal, " -- "
end
```

生成：

```
cat -- dog -- horse --
```

。。。这里是print， 看来 print 不带换行， puts带换行。

```
5.times {  print "*" }
3.upto(6) {|i|  print i }
('a'..'e').each {|char| print char }
```

生成：
*****3456abcde

写：
puts writes each of its arguments, adding a newline after each. print also writes its arguments, but with no newline.
。。。puts 写每个参数，每个参数后面加一个 新行。
。。print 写每个参数，但是不加新行。

printf， 输出 参数 到 一个 format string。
printf "Number: %5.2f, String: %s", 1.23, "hello"
生成：
Number: 1.23, String: hello

读：
```
line = gets
print line
```

gets有一个副作用， 返回 读取的行 ，保存到 全局变量 $_ 。

如果调用 print，不带参数， 就会 输出 $_ 的内容。
```
while gets           # assigns line to $_
  if /Ruby/          # matches against $_
    print            # prints $_
  end
end
```

ruby方式的写 会使用iterator
```
ARGF.each { |line|  print line  if line =~ /Ruby/ }
```

Classes, Objects, and Variables

创建basic class Song

```
class Song
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
  end
end
```

initialize 是一个特殊的方法，当你使用Song.new 来创建一个新的Song对象，Ruby创建一个 未初始化的 对象，然后调用 对象的 initialize方法，把传给new的所有参数传给initialize。

```
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.inspect      »      "#<Song:0x401b4924 @duration=260, @artist=\"Fleck\",
@name=\"Bicylops\">"
```

inspect消息能被传递给任何对象，dump out 对象id和实例变量。

to_s 消息，传递给任何对象，render(使成为，回报，提交) 一个string

```
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.to_s  »      "#<Song:0x401b499c>"
```

可以重写to_s

类不会关闭，你可以往现有的类中添加方法。 标准的内置类也可以被 添加。

```
class Song
  def to_s
    "Song: #{@name}--#{@artist} (#{@duration})"
  end
end
```

添加重写的方法，

Inheritance and Messages

```
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end
```

继承

```
class KaraokeSong
  # ...
  def to_s
    "KS: #{@name}--#{@artist} (#{@duration}) [#{@lyrics}]"
  end
```

```
end
aSong = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
aSong.to_s  »     "KS: My Way--Sinatra (225) [And now, the...]"
```

重写父类的to_s

这里，子类直接访问了父类的实例变量，这是一个 差的实现

我们可能决定 以毫秒数 来保存duration(持续时间)。 子类就。。

每个类只处理它自己的内部状态。
调用子类.to_s时，子类会调用父类的to_s，然后把自己的信息追加到后面。
关键字super。

当你直接调用super (不带任何参数)时， ruby会调用父类的 和当前方法同名的方法，把传递给当前方法的 参数 也传递给 父类的方法。

```
class KaraokeSong < Song
  # Format ourselves as a string by appending
  # our lyrics to our parent's #to_s value.
  def to_s
    super + " [#{@lyrics}]"
  end
end
aSong = KaraokeSong.new("My Way", "Sinatra", 225, "And now, the...")
aSong.to_s  »     "Song: My Way--Sinatra (225) [And now, the...]"
```

Inheritance and Mixins

类只能有一个直接父类，但，类可以包含任意数量mixin的功能

Objects and Attributes

访问

```
class Song
  def name
    @name
  end
  def artist
    @artist
  end
  def duration
    @duration
  end
end
aSong = Song.new("Bicylops", "Fleck", 260)
```

```
aSong.artist      »     "Fleck"
aSong.name  »     "Bicylops"
aSong.duration    »     260
```

简写 attr_reader

```
class Song
 attr_reader :name, :artist, :duration
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.artist   »      "Fleck"
aSong.name   »      "Bicylops"
aSong.duration      »      260
```

。。后面的应该是方法名，会自动寻找 @+方法名 的属性。

In this example, we named the accessor methods name, artist, and duration. The corresponding instance variables, @name, @artist, and @duration, will be created automatically.

。。这样就不需要 initialize 中声明了吗？ 就是 最终的 实例变量 是 initialize + attr_reader 中声明的变量的 并集？

Writable Attributes

```
class Song
  def duration=(newDuration)
    @duration = newDuration
  end
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.duration    »     260
aSong.duration = 257   # set attribute with updated value
aSong.duration    »     257
```

。。这2个等于 有没有 可能一起替换掉？可以的。。。=是方法名的一部分。。不，调用的时候 = 是有空格的。。估计是 定义了一个 operator 。

The assignment ``aSong.duration = 257'' invokes the method duration= in the aSong object,

简写

```
class Song
  attr_writer :duration
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.duration = 257
```

Virtual Attributes

希望使用分钟数，而不是秒数来 获得 duration

```
class Song
  def durationInMinutes
    @duration/60.0   # force floating point
  end
  def durationInMinutes=(value)
    @duration = (value*60).to_i
  end
end
aSong = Song.new("Bicylops", "Fleck", 260)
aSong.durationInMinutes       »     4.333333333
aSong.durationInMinutes = 4.2
aSong.duration    »     252
```

Class Variables and Class Methods

类变量，2个@开头。

和global ，instance 变量不同， 类变量必须先初始化，然后使用。
。。估计是 global instance 有默认值， 类变量没有默认值。 不过任何时候都应该先初始化，再使用。

```
class Song
  @@plays = 0
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
    @plays    = 0
  end
  def play
    @plays += 1
    @@plays += 1
    "This  song: #@plays plays. Total #@@plays plays."
  end
end
```

。。@play 是 实例变量， @@play是类变量。。 类变量不在 initialize中初始化。

```
s1 = Song.new("Song1", "Artist1", 234)  # test songs..
s2 = Song.new("Song2", "Artist2", 345)
s1.play     »     "This  song: 1 plays. Total 1 plays."
s2.play     »     "This  song: 1 plays. Total 2 plays."
s1.play     »     "This  song: 2 plays. Total 3 plays."
s1.play     »     "This  song: 3 plays. Total 4 plays."
```

类变量是 private， 需要方法才能访问。

类方法是 类名.方法名

```
class Example
  def instMeth              # instance method
  end
  def Example.classMeth     # class method
  end
end
```

检查一首歌是否太长了。

```
class SongList
  MaxTime = 5*60            #  5 minutes。。。 这个是 constant
  def SongList.isTooLong(aSong)
    return aSong.duration > MaxTime
  end
end
song1 = Song.new("Bicylops", "Fleck", 260)
SongList.isTooLong(song1)     »     false
song2 = Song.new("The Calling", "Santana", 468)
SongList.isTooLong(song2)     »     true
```

单例，保证只有一个 log对象。

```
class Logger
  private_class_method :new
  @@logger = nil
  def Logger.create
    @@logger = new unless @@logger
    @@logger
  end
end
```

把new方法设置为private，组织任何人通过常规构造器来 创建新的 logger对象。
这里的例子不是线程安全的。

有一个类：

```
class Shape
  def initialize(numSides, perimeter)
    # ...
  end
end
```

一段时间后，可能需要 按照名字(正三角形，正方形)和 长度 来创建：

```
class Shape
  def Shape.triangle(sideLength)
    Shape.new(3, sideLength*3)
  end
  def Shape.square(sideLength)
    Shape.new(4, sideLength*4)
```

```
  end
end
```

Access Control

public method: 默认都是public的 (除了initialize，这个始终是private的)。
protected method: 类和子类调用
private method: 只能在定义的类中， 和 相同对象的直接后代中 调用
。。private 不懂。 private methods can be called only in the defining class and by direct descendents within that same object.

protected 和 private 的差别非常微妙。
如果方法是protected，可以被 任何 定义这个方法的 类 或类的子类 的 实例 调用。
private，只能在 calling object的上下文中 被调用， 不可能被其他对象的私有方法直接访问，即使对象 和 caller是同一个类。。

。。感觉好像是说，protected 是 可以调用 相同类型的其他对象的 protected方法。，
private是 只能 调用自己的 private方法，相同类型的其他对象的private方法调不到的。

2种声明 访问权限 方式

```
class MyClass
      def method1    # default is 'public'
        #...
      end
  protected          # subsequent methods will be 'protected'
      def method2    # will be 'protected'
        #...
      end
  private            # subsequent methods will be 'private'
      def method3    # will be 'private'
        #...
      end
  public             # subsequent methods will be 'public'
      def method4    # and this will be 'public'
        #...
      end
end
```

```
class MyClass
  def method1
  end
  # ... and so on
  public    :method1, :method4
  protected :method2
  private   :method3
```

```
end
```

```
class Accounts
  private
    def debit(account, amount)
      account.balance -= amount
    end
    def credit(account, amount)
      account.balance += amount
    end
  public
    #...
    def transferToSavings(amount)
      debit(@checking, amount)
      credit(@savings, amount)
    end
    #...
end
```

```
class Account
  attr_reader :balance       # accessor method 'balance'
  protected :balance         # and make it protected
  def greaterBalanceThan(other)
    return @balance > other.balance
  end
end
```

。。666

Variables

```
person = "Tim"
person.id   »     537771100
person.type       »     String
person      »     "Tim"

person1 = "Tim"
person2 = person1
person1[0] = 'J'
person1     »     "Jim"
person2     »     "Jim"
```

。指向同一个对象。

```
person1 = "Tim"
person2 = person1.dup
person1[0] = "J"
person1     »     "Jim"
person2     »     "Tim"
```

阻止任何人修改

```
person1 = "Tim"
person2 = person1
person1.freeze       # prevent modifications to the object
person2[0] = "J"
```

触发一个异常。

```
prog.rb:4:in `=': can't modify frozen string (TypeError)
    from prog.rb:4
```

Containers, Blocks, and Iterators

```
a = [ 3.14159, "pie", 99 ]
a.type      »     Array
a.length    »     3
a[0]  »     3.14159
a[1]  »     "pie"
a[2]  »     99
a[3]  »     nil
b = Array.new
b.type      »     Array
b.length    »     0
b[0] = "second"
b[1] = "array"
b     »     ["second", "array"]

a = [ 1, 3, 5, 7, 9 ]
a[-1]       »     9
a[-2]       »     7
a[-99]      »     nil


a = [ 1, 3, 5, 7, 9 ]
a[1, 3]     »     [3, 5, 7]
a[3, 1]     »     [7]
a[-3, 2]    »     [5, 7]
```

返回新数组。[start, count]

```
a = [ 1, 3, 5, 7, 9 ]
a[1..3]       »     [3, 5, 7]
a[1...3]      »     [3, 5]
```

a[3..3] » [7]
a[-3..-1] » [5, 7, 9]
start,end，2个. 包含end， 3个. 不包含end

a = [1, 3, 5, 7, 9] » [1, 3, 5, 7, 9]
a[1] = 'bat' » [1, "bat", 5, 7, 9]
a[-3] = 'cat' » [1, "bat", "cat", 7, 9]
a[3] = [9, 8] » [1, "bat", "cat", [9, 8], 9]
a[6] = 99 » [1, "bat", "cat", [9, 8], 9, nil, 99]

如果是 2个数字(start,length)，或者range。 那么 选择的多个元素会被替换。

```
a = [ 1, 3, 5, 7, 9 ]   »     [1, 3, 5, 7, 9]
a[2, 2] = 'cat'   »     [1, 3, "cat", 9]
a[2, 0] = 'dog'   »     [1, 3, "dog", "cat", 9]
a[1, 1] = [ 9, 8, 7 ]   »     [1, 9, 8, 7, "dog", "cat", 9]
a[0..3] = []      »     ["dog", "cat", 9]
a[5] = 99   »     ["dog", "cat", 9, nil, nil, 99]
```

数组还有许多其他方法，通过这些方法，可以把数组 视为 stack，set，queue，dequeue，fifo。

哈希

```
h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine' }
h.length    »     3
h['dog']    »     "canine"
h['cow'] = 'bovine'
h[12]     = 'dodecine'
h['cat'] = 99
h     »     {"cow"=>"bovine", "cat"=>99, 12=>"dodecine", "donkey"=>"asinine",
"dog"=>"canine"}
```

元素是无序的，所以很难作为 stack queue。

保存Song。

```
class SongList
  def initialize
    @songs = Array.new
  end
end

class SongList
  def append(aSong)
    @songs.push(aSong)
```

```
    self
  end
end

class SongList
  def deleteFirst
    @songs.shift
  end
  def deleteLast
    @songs.pop
  end
end

class SongList
  def [](key)
    if key.kind_of?(Integer)
      @songs[key]
    else
      # ...
    end
  end
end
list[0]     »     Song: title1--artist1 (1)
list[2]     »     Song: title3--artist3 (3)
list[9]     »     nil


class SongList
  def [](key)
    if key.kind_of?(Integer)
      return @songs[key]
    else
      for i in 0...@songs.length
        return @songs[i] if key == @songs[i].name
      end
    end
    return nil
  end
end

class SongList
  def [](key)
    if key.kind_of?(Integer)
      result = @songs[key]
    else
      result = @songs.find { |aSong| key == aSong.name }
    end
    return result
  end
```

```
end
```
Array的find方法。

```
class SongList
  def [](key)
    return @songs[key] if key.kind_of?(Integer)
    return @songs.find { |aSong| aSong.name == key }
  end
end
```

Ruby iterator 是一个方法，它可以invoke 代码块。
块只出现在 源码种调用方法的 相连的地方。 同 方法的最后一个参数 写在同一行。
块中的代码 在遇到时不会执行，ruby会记住块的上下文(local变量，当前对象等)，然后进入到方法中。
在方法中，使用yield 来invoke 块。

```
def threeTimes
  yield
  yield
  yield
end
threeTimes { puts "Hello" }
```

输出：
```
Hello
Hello
Hello
```

```
def fibUpTo(max)
  i1, i2 = 1, 1        # parallel assignment
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fibUpTo(1000) { |f| print f, " " }
```

可以给 block参数， 也可以从block 接收参数。

当yield 的实参 和block的形参不匹配时 会发生什么？ 这里会使用 parallel assignment的规则(parallel assignment 就是 a,b=1,2，但是我不知道 数目不匹配会发生什么。。。)
当形参只有一个，但是实参有多个时， 实参会被转为数组，赋给形参。
。。所以 形参可能是单个值，也可能是数组。

传给block的参数可能是一个已存在的局部变量。这个变量的新的值在block结束后依然存在这可能导致 意外的行为，

block也可以返回值 到方法。 block中最后一个表达式的eval出的值就是 yield返回给 方法的值。

```
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
[1, 3, 5, 7, 9].find {|v| v*v > 30 }
```

```
[ 1, 3, 5 ].each { |i| puts i }
```

产生：

```
1
3
5
```

```
["H", "A", "L"].collect { |x| x.succ }    »    ["I", "B", "M"]
```

创建新数组。

```
f = File.open("testfile")
f.each do |line|
  print line
end
f.close
```

```
class Array
  def inject(n)
    each { |value| n = yield(n, value) }
    n
  end
  def sum
    inject(0) { |n, value| n + value }
  end
  def product
    inject(1) { |n, value| n * value }
  end
end
[ 1, 2, 3, 4, 5 ].sum    »    15
[ 1, 2, 3, 4, 5 ].product    »    120
```

block 可以用于，确保 一些操作必须执行(如，文件打开后的关闭操作)

```
class File
  def File.openAndProcess(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end

File.openAndProcess("testfile", "r") do |aFile|
  print while aFile.gets
end
```

do..end 和 {} 定义块的 区别是， do...end 的优先级 低。

block作为闭包

上下文：需要按钮来启动歌，和暂停歌。一般情况下，是 2个类，一个开始按钮类，一个结束按钮类，都继承 按钮类。这样会有2个问题： 1个是 会导致很多子类，如果修改了按钮类，可能导致修改子类。。 2，按钮按下时，行为的执行 时在一个错误的 层级，这不应该是按钮的功能，而是歌曲列表的功能。
我们用block 修复这2个问题。

```
class JukeboxButton < Button
  def initialize(label, &action)
    super(label)
    @action = action
  end
  def buttonPressed
    @action.call(self)
  end
end

bStart = JukeboxButton.new("Start") { songList.start }
bPause = JukeboxButton.new("Pause") { songList.pause }
```

这里的关键是 initialize的 第二个参数。
如果一个方法的 最后一个参数 是以 & 开头， 当这个方法被调用的时候，ruby寻找 代码块。代码块转为一个Proc类的对象，并且赋值给参数。 使用Proc类的call方法来 执行代码块。
So what exactly do we have when we create a Proc object? The interesting thing is that it's more than just a chunk of code. Associated with a block (and hence a Proc object) is all the context in which the block was defined: the value of self, and the methods, variables, and constants in scope. Part of the magic of Ruby is that the block can still use all this original scope information even if the environment in which it was defined would otherwise have disappeared. In other

languages, this facility is called a closure.
。。翻译：那么，当我们创建一个Proc对象时，我们到底拥有什么呢？有趣的是，它不仅仅是一段代码。与块（以及Proc对象）相关联的是定义块的所有上下文：self的值，以及范围中的方法、变量和常量。Ruby的神奇之处在于，即使定义它的环境已经消失，块仍然可以使用所有这些原始范围信息。在其他语言中，此功能称为闭包。

下面使用proc方法 将block 转为 Proc对象

```
def nTimes(aThing)
  return proc { |n| aThing * n }
end
p1 = nTimes(23)
p1.call(3)  »     69
p1.call(4)  »     92
p2 = nTimes("Hello ")
p2.call(3)  »     "Hello Hello Hello "
```

Standard Types
numbers, strings, ranges, and regular expressions.

支持 整数 和 浮点数。

整型可以任意长度。
在-2^30 到2^30-1 或 2^62到2^62-1 范围内的是 Fixnum类的对象。
超出范围的 是 Bignum 类的对象
ruby自动转换

```
123456                  # Fixnum
123_456                 # Fixnum (underscore ignored)
-543                    # Negative Fixnum
123_456_789_123_345_789 # Bignum
0xaabb                  # Hexadecimal
0377                    # Octal
-0b101_010              # Binary (negated)
```

正负号，基数，下划线。

可以从ascii字符 或 转义序列 中获得 整型，通过在 它 前面放一个 问号。
控制符和元组合 也可以 生成，通过 ?\C-x，?\M-x，?\M-\C-x。
一个值的控制版本 等同于 ``value & 0x9f''
一个值的元版本的值是 ``value | 0x80''.
?\C-? 生成ascii的删除 0177

```
?a              # character code
?\n             # code for a newline (0x0a)
?\C-a           # control a = ?A & 0x9f = 0x01
```

```
?\M-a              # meta sets bit 7
?\M-\C-a           # meta and control a
?\C-?              # delete character
```

包含小数点的数值字面量 被转为一个 Float对象，这个相当于 双精度数据类型。。

小数点后面 空或者必须是数字。 1.e3会尝试 调用 Fixnum类的 e3方法。

```
aNumber.abs, not abs(aNumber).
```

```
3.times         { print "X " }
1.upto(5)       { |i| print i, " " }
99.downto(95)   { |i| print i, " " }
50.step(80, 5)  { |i| print i, " " }
```

只包含数字的 string 在表达式中不会自动变成 数字。 这是Perl的特性。

```
DATA.each do |line|
  vals = line.split    # split line, storing tokens in val
  print vals[0] + vals[1], " "
end
```

```
DATA.each do |line|
  vals = line.split
  print vals[0].to_i + vals[1].to_i, " "
end
```

Strings
是 8bit bytes 的序列。 一般保存可打印字符， 也可以保存 二进制数据。
字符串是 String类的 对象。

```
'escape using "\\"'      »     escape using "\"
'That\'s right'   »     That's right
```

双引号 可以 计算表达式。 如果表达式只是一个 全局/类/实例 变量，可以省略 {}

```
"Seconds/day: #{24*60*60}" »       Seconds/day: 86400
"#{'Ho! '*3}Merry Christmas"       »       Ho! Ho! Ho! Merry Christmas
"This is line #$."       »       This is line 3
```

有额外三种方式 来构建string 字面量， %q, %Q, and ``here documents.''

%q, %Q, 开始 定位 一个 单引号 或双引号的 string。

```
%q/general single-quoted string/    »     general single-quoted string
%Q!general double-quoted string!    »     general double-quoted string
%Q{Seconds/day: #{24*60*60}}   »     Seconds/day: 86400
```

在Q/q 后面的是 分隔符， 如果是 ([{< 之一， 那么会一直读到 匹配的另外一部分。

否则，读到下一次 相同的分隔符 出现。

可以使用 here document 来构建字符串

```
aString = <<END_OF_STRING
    The body of the string
    is the input lines up to
    one ending with the same
    text that followed the '<<'
END_OF_STRING

print <<-STRING1, <<-STRING2
   Concat
   STRING1
      enate
      STRING2
```

生成：

```
     Concat
        enate
```

。。似乎是 << 直接跟一个 标记， 然后 一直读 读到 这个标记。

if you put a minus sign after the << characters, you can indent(缩进) the terminator.

。。就是 <<- 可以 包含缩进。？

现在有歌曲的信息，保存在文件中，按下面的格式：

```
/jazz/j00132.mp3  | 3:45 | Fats     Waller      | Ain't Misbehavin'
/jazz/j00319.mp3  | 2:58 | Louis    Armstrong   | Wonderful World
/bgrass/bg0732.mp3| 4:09 | Strength in Numbers | Texas Red
         :           :          :                    :
```

我们从文件中提取信息 来创建Song对象。我们需要：把行切分成属性，转换mm:ss到秒，移除歌手名字的额外的空格

使用String类的 split 。传给split一个 正则，来切分行。
使用String#chomp 来 strip

```
songs = SongList.new
songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  songs.append Song.new(title, name, length)
end
puts songs[1]
```

。。顺序是 从后往前。
这里的名字是分得很开得，我们需要使用String#squeeze 来 删除重复的字符。

```
songs = SongList.new
songFile.each do |line|
```

```
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  songs.append Song.new(title, name, length)
end
puts songs[1]
```

。。原地操作的。

我们通过: 来split 时间，获得 分，秒 数。

```
mins, secs = length.split(/:/)
```

有一个类似split的方法 String#scan，这个根据 pattern 打碎一个string 到多个块。scan的pattern 是你想要获得的 数据的pattern。

```
songs = SongList.new
songFile.each do |line|
  file, length, name, title = line.chomp.split(/\s*\|\s*/)
  name.squeeze!(" ")
  mins, secs = length.scan(/\d+/)
  songs.append Song.new(title, name, mins.to_i*60+secs.to_i)
end
puts songs[1]
```

关键字搜索，输入歌名或歌手名 的 一个单词 来寻找。

```
class WordIndex
  def initialize
    @index = Hash.new(nil)
  end
  def index(anObject, *phrases)
    phrases.each do |aPhrase|
      aPhrase.scan /\w[-\w']+/ do |aWord|   # extract each word
        aWord.downcase!
        @index[aWord] = [] if @index[aWord].nil?
        @index[aWord].push(anObject)
      end
    end
  end
  def lookup(aWord)
    @index[aWord.downcase]
  end
end
```

。。*参数 是什么。。。应该是数组，但是 之前好像没有遇到， 那么**是hash?

```
class SongList
  def initialize
    @songs = Array.new
    @index = WordIndex.new
  end
```

```
  def append(aSong)
    @songs.push(aSong)
    @index.index(aSong, aSong.name, aSong.artist)
    self
  end
  def lookup(aWord)
    @index.lookup(aWord)
  end
end
```

Ranges

```
1..10
'a'..'z'
0...anArray.length
```

2个. 是[] 3个点是()

```
(1..10).to_a    »    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a    »    ["bar", "bas", "bat"]
```

range就是 Range类的对象， 底层并不是数组， 而是 包含2个Fixnum对象的 Range对象。

```
digits = 0..9
digits.include?(5)  »    true
digits.min    »    0
digits.max    »    9
digits.reject {|i| i < 5 }    »    [5, 6, 7, 8, 9]
digits.each do |digit|
  dial(digit)
end
```

可以基于对象创建range，只要这个对象有 succ方法 来返回下一个对象，及<=>方法用于比较，返回-1,0,1代表第一个元素小鱼，等于大于 第二个。

```
class VU
  include Comparable
  attr :volume
  def initialize(volume)  # 0..9
    @volume = volume
  end
  def inspect
    '#' * @volume
  end
```

```
  # Support for ranges
  def <=>(other)
    self.volume <=> other.volume
  end
  def succ
    raise(IndexError, "Volume too big") if @volume >= 9
    VU.new(@volume.succ)
  end
end

medium = VU.new(4)..VU.new(7)
medium.to_a       »     [####, #####, ######, #######]
medium.include?(VU.new(3))    »     false
```

range也能用在条件表达式中，下面的代码 打印从 标准输入 进来的行，并且 是以 start 开头，end结尾的 行。

```
while gets
  print if /start/../end/
end
```

。。应该是 2个 re 。

range可以作为interval test， 观察是否 一些值 能掉落到这个range中。 使用===

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

RE

RE是Regexp类的 对象
3种创建方式：constructor /pattern/ %r\pattern\.

```
a = Regexp.new('^\s*[a-z]')   »     /^\s*[a-z]/
b = /^\s*[a-z]/   »     /^\s*[a-z]/
c = %r{^\s*[a-z]}      »     /^\s*[a-z]/
```

Regexp#match(aString) 或 =~ (positive match) !~ (negative match) 来匹配

```
a = "Fats Waller"
a =~ /a/      »     1
a =~ /z/      »     nil
a =~ "ll"     »     7
```

返回匹配的字符的下标，副作用：$& 被赋值 匹配的部分， $` 被复制匹配之前的部分，$' 收到匹配之后的部分。

。。whole load of ruby variable， 估计上面的3个 是全局唯一的？ 因为下面有 线程的。

```
def showRE(a,re)
  if a =~ re
    "#{$`}<<#{$&}>>#{$'}"
  else
    "no match"
  end
end
showRE('very interesting', /t/)     »     very in<<t>>eresting
showRE('Fats Waller', /ll/)   »     Fats Wa<<ll>>er
```

匹配也会设置 thread-global 变量 $~ 和 $1到$9 。

$~是一个 MatchData对象，包含了此次匹配的所有信息。
$1 和后续 保存了 匹配的部分。

每个RE包含了一个 pattern。

在pattern中，所有的字符都匹配它们自己，除了 ., |, (,), [, {, +, \, ^, $, *, and ?

```
showRE('kangaroo', /angar/)   »     k<<angar>>oo
showRE('!@%&-_=+', /%&/)      »     !@<<%&>>-_=+
```

上面的特殊字符 需要前面加 \ 才是匹配它们。

```
showRE('yes | no', /\|/)      »     yes <<|>> no
showRE('yes (no)', /\(no\)/)  »     yes <<(no)>>
showRE('are you sure?', /e\?/)      »     are you sur<<e?>>
```

RE可能包含 #{...}表达式替换

Anchors
^ $ 匹配行的 头和尾，它们经常作为一个 anchor。

/^option/ 匹配出现在 行头的 option

\A 匹配stirng的开始。
\z \Z 匹配string的结束(\z 匹配string的结束，除非string以 \n结尾，这时，它会匹配\n前面的字符)。

```
showRE("this is\nthe time", /^the/)       »     this is\n<<the>> time
showRE("this is\nthe time", /is$/)  »     this <<is>>\nthe time
showRE("this is\nthe time", /\Athis/)     »     <<this>> is\nthe time
showRE("this is\nthe time", /\Athe/)      »     no match
```

\b \B 匹配边界 (boundary) 和 不是边界。

```
showRE("this is\nthe time", /\bis/)       »     this <<is>>\nthe time
```

```
showRE("this is\nthe time", /\Bis/)        »     th<<is>> is\nthe time
```

字符类
是方括号里的字符和的集合。 匹配任何单独的在[]中的的char

```
[aeiou] will match a vowel

showRE('It costs $12.', /[aeiou]/)  »     It c<<o>>sts $12.
showRE('It costs $12.', /[\s]/)     »     It<< >>costs $12.
```

在[]中， c1-c2代表了 所有c1 c2之间的char，含c1c2
如果要匹配-，那么需要把-写最前面

```
a = 'Gamma [Design Patterns-page 123]'
showRE(a, /[]]/)  »     Gamma [Design Patterns-page 123<<]>>
showRE(a, /[B-F]/)      »     Gamma [<<D>>esign Patterns-page 123]
showRE(a, /[-]/)  »     Gamma [Design Patterns<<->>page 123]
showRE(a, /[0-9]/)      »     Gamma [Design Patterns-page <<1>>23]
```

在[后面加^ 代表取反，[^a-z] matches any character that isn't a lowercase alphabetic.

对于 用得比较多的 字符类， 有一些简写

```
\d    [0-9]       Digit character
\D    [^0-9]      Nondigit
\s    [\s\t\r\n\f]      Whitespace character
\S    [^\s\t\r\n\f]     Nonwhitespace character
\w    [A-Za-z0-9_]      Word character
\W    [^A-Za-z0-9_]     Nonword character
```

[] 外的 . 代表任何char，除了 新行。

```
a = 'It costs $12.'
showRE(a, /c.s/)  »     It <<cos>>ts $12.
showRE(a, /./)    »     <<I>>t costs $12.
showRE(a, /\./)   »     It costs $12<<.>>
```

Repetition

r代表一个RE (immediately preceding regular expression)

```
r *   matches zero or more occurrences of r.
r +   matches one or more occurrences of r.
r ?   matches zero or one occurrence of r.
r {m,n}     matches at least ``m'' and at most ``n'' occurrences of r.
r {m,}      matches at least ``m'' occurrences of r.
```

重复的优先级很高，所以 /ab+/ 代表一个a，>=1个b， 而不是 多个ab

匹配是贪婪的。

```
a = "The moon is made of cheese"
showRE(a, /\w+/)  »     <<The>> moon is made of cheese
showRE(a, /\s.*\s/)     »     The<< moon is made of >>cheese
showRE(a, /\s.*?\s/)    »     The<< moon >>is made of cheese
showRE(a, /[aeiou]{2,99}/)    »     The m<<oo>>n is made of cheese
showRE(a, /mo?o/)       »     The <<moo>>n is made of cheese
```

Alternation
使用 | 表示 either，优先级非常低。注意最后一个例子。

```
a = "red ball blue sky"
showRE(a, /d|e/)  »     r<<e>>d ball blue sky
showRE(a, /al|lu/)      »     red b<<al>>l blue sky
showRE(a, /red ball|angry sky/)     »     <<red ball>> blue sky
```

Grouping
使用() 来分组。 每个组 被当作一个 整体。

```
showRE('banana', /an*/)       »     b<<an>>ana
showRE('banana', /(an)*/)     »     <<>>banana
showRE('banana', /(an)+/)     »     b<<anan>>a

a = 'red ball blue sky'
showRE(a, /blue|red/)   »     <<red>> ball blue sky
showRE(a, /(blue|red) \w+/)   »     <<red ball>> blue sky
showRE(a, /(red|blue) \w+/)   »     <<red ball>> blue sky
showRE(a, /red|blue \w+/)     »     <<red>> ball blue sky

showRE(a, /red (ball|angry) sky/)   »     no match
a = 'the red angry sky'
showRE(a, /red (ball|angry) sky/)   »     the <<red angry sky>>
```

()可以用于收集匹配结果。
在pattern中 \1代表 第一个组的匹配， \2代表第二个组的匹配 以此类推。
在pattern外，$1,$2… 相同的作用。

```
"12:50am" =~ /(\d\d):(\d\d)(..)/    »     0
"Hour is #$1, minute #$2"     »     "Hour is 12, minute 50"
"12:50am" =~ /((\d\d):(\d\d))(..)/  »     0
"Time is #$1"     »     "Time is 12:50"
"Hour is #$2, minute #$3"     »     "Hour is 12, minute 50"
"AM/PM is #$4"    »     "AM/PM is am"

# match duplicated letter
showRE('He said "Hello"', /(\w)\1/)       »     He said "He<<ll>>o"
# match duplicated substrings
showRE('Mississippi', /(\w+)\1/)    »     M<<ississ>>ippi

showRE('He said "Hello"', /(["']).*?\1/)  »     He said <<"Hello">>
showRE("He said 'Hello'", /(["']).*?\1/)  »     He said <<'Hello'>>
```

Pattern-Based Substitution

```
a = "the quick brown fox"
a.sub(/[aeiou]/,  '*')  »     "th* quick brown fox"
a.gsub(/[aeiou]/, '*')  »     "th* q**ck br*wn f*x"
a.sub(/\s\S+/,  '')     »     "the brown fox"
a.gsub(/\s\S+/, '')     »     "the"
```

String#sub! and String#gsub! modify the original string.
。。看来带! 就是会修改原对象。。。不清楚返回是什么？

```
a = "the quick brown fox"
a.sub(/^./) { $&.upcase }     »     "The quick brown fox"
a.gsub(/[aeiou]/) { $&.upcase }     »     "thE qUIck brOwn fOx"
```

第二个参数可以是 string 或block

```
def mixedCase(aName)
  aName.gsub(/\b\w/) { $&.upcase }
end
mixedCase("fats waller")      »     "Fats Waller"
mixedCase("louis armstrong")  »     "Louis Armstrong"
mixedCase("strength in numbers")    »     "Strength In Numbers"
```

Backslash Sequences in the Substitution

```
"fred:smith".sub(/(\w+):(\w+)/, '\2, \1')       »     "smith, fred"
"nercpyitno".gsub(/(.)(.)/, '\2\1')       »     "encryption"
```

。。

\& (last match), \+ (last matched group), \` (string prior to match), \' (string after match), and \\ (a literal backslash).

将\替换为\\ 非常的繁琐。

```
str = 'a\b\c'     »     "a\b\c"
str.gsub(/\\/, '\\\\\\\\')    »     "a\\b\\c"
```

可以使用\& 来替换为 匹配的string。

```
str = 'a\b\c'     »     "a\b\c"
str.gsub(/\\/, '\&\&')  »     "a\\b\\c"
```

使用block

```
str = 'a\b\c'     »     "a\b\c"
str.gsub(/\\/) { '\\\\' }     »     "a\\b\\c"
```

```
def unescapeHTML(string)
```

```
  str = string.dup
  str.gsub!(/&(.*?);/n) {
    match = $1.dup
    case match
    when /\Aamp\z/ni           then '&'
    when /\Aquot\z/ni          then '"'
    when /\Agt\z/ni            then '>'
    when /\Alt\z/ni            then '<'
    when /\A#(\d+)\z/n         then Integer($1).chr
    when /\A#x([0-9a-f]+)\z/ni then $1.hex.chr
    end
  }
  str
end
puts unescapeHTML("1&lt;2 &amp;&amp; 4&gt;3")
puts unescapeHTML("&quot;A&quot; = &#65; = &#x41;")
```

产生：

```
1<2 && 4>3
"A" = A = A
```

Object-Oriented Regular Expressions

```
re = /cat/
re.type     »     Regexp

re = /(\d+):(\d+)/     # match a time hh:mm
md = re.match("Time: 12:34am")
md.type     »     MatchData
md[0]         # == $&   »     "12:34"
md[1]         # == $1   »     "12"
md[2]         # == $2   »     "34"
md.pre_match  # == $`   »     "Time: "
md.post_match # == $'   »     "am"
```

Regexp#match 进行匹配，如果不成功，返回nil，成功返回 MatchData 对象。

。。可以进行对比。永久保存。

```
re = /(\d+):(\d+)/     # match a time hh:mm
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
md1[1, 2]   »     ["12", "34"]
md2[1, 2]   »     ["10", "30"]
```

把结果保存在一个 thread local 变量 $~ 中 ， 所有其他的 re变量都是从这里 获得值得。

```
re = /(\d+):(\d+)/
md1 = re.match("Time: 12:34am")
md2 = re.match("Time: 10:30pm")
```

```
[ $1, $2 ]   # last successful match      »     ["10", "30"]
$~ = md1
[ $1, $2 ]   # previous successful match  »     ["12", "34"]
```

More About Methods
其他语言有 function, procedure, method, routine ， ruby只有method

定义方法
使用def 关键字定义。 方法名是 小写开头 (大写，ruby会认为是constant，无法执行得)。
有些方法最后是? 这些方法 表示 是一个 查询。 如 instance_of?
最后是! ，这些方法是危险的，或者修改了 receiver (。。指调用者)。 如 String 提供了 chop 和 chop! ，2个方法，第一个 返回一个 修改后的string， 第二个原地修改receiver。

下面 声明了参数

```
def myNewMethod(arg1, arg2, arg3)     # 3 arguments
  # Code for the method would go here
end
def myOtherNewMethod                  # No arguments
  # Code for the method would go here
end
```

默认值

```
def coolDude(arg1="Miles", arg2="Coltrane", arg3="Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end
coolDude     »     "Miles, Coltrane, Roach."
coolDude("Bart")  »     "Bart, Coltrane, Roach."
coolDude("Bart", "Elwood")    »     "Bart, Elwood, Roach."
coolDude("Bart", "Elwood", "Linus")       »     "Bart, Elwood, Linus."
```

。。从前往后匹配。

可变长度参数列表
多余的参数会变成一个数组。

```
def varargs(arg1, *rest)
  "Got #{arg1} and #{rest.join(', ')}"
end
varargs("one")    »     "Got one and "
varargs("one", "two")   »     "Got one and two"
varargs "one", "two", "three"       »     "Got one and two, three"
```

方法和block

```
def takeBlock(p1)
  if block_given?
```

```
    yield(p1)
  else
    p1
  end
end
takeBlock("no block")    »      "no block"
takeBlock("no block") { |s| s.sub(/no /, '') }  »      "block"
```

如果最后一个参数是&开头， block会 转为一个 Proc 对象，然后对象赋给参数。

```
class TaxCalculator
  def initialize(name, &block)
    @name, @block = name, block
  end
  def getTax(amount)
    "#@name on #{amount} = #{ @block.call(amount) }"
  end
end
tc = TaxCalculator.new("Sales tax") { |amt| amt * 0.075 }
tc.getTax(100)     »      "Sales tax on 100 = 7.5"
tc.getTax(250)     »      "Sales tax on 250 = 18.75"
```

&不是参数名的一部分。

调用方法
通过 指定一个 receiver，方法的名字，参数，block， 来调用方法。

```
connection.downloadMP3("jitterbug") { |p| showProgress(p) }
```

对于类，模块 方法，receiver 是 类或模块的名字。

```
File.size("testfile")
Math.sin(Math::PI/4)
```

省略receiver，则默认是self，当前对象

```
self.id      »      537794160
id     »      537794160
self.type    »      Object
type   »      Object
```

方法名字后面是可选的参数，如果 没有二义性，可以省略()

```
a = obj.hash     # Same as
a = obj.hash()   # this.
obj.someMethod "Arg1", arg2, arg3    # Same thing as
obj.someMethod("Arg1", arg2, arg3)   # with parentheses.
```

前面说到，在一个方法定义的 参数的前面加一个*， 调用方法时，多个实参会被 绑定到一个数组。 the same thing works in reverse。。。
。。实参是数组，形参是一个个的参数。

调用方法时，能分离数组，它的每个元素 作为一个 独立的参数。，通过在数组实参(这个必须在规则参数后面)前面加一个星号

```
def five(a, b, c, d, e)
  "I was passed #{a} #{b} #{c} #{d} #{e}"
end
five(1, 2, 3, 4, 5 )     »     "I was passed 1 2 3 4 5"
five(1, 2, 3, *['a', 'b'])     »     "I was passed 1 2 3 a b"
five(*(10..14).to_a)     »     "I was passed 10 11 12 13 14"
```

。。元素个数 不相等 会发生什么呢，。

在方法调用时，关联一个 blokc

```
listBones("aardvark") do |aBone|
  # ...
end
```

你可能需要更 多变，
假设我们在教数学， 有些学生想要一个 n+表，或者n倍表。。。(反正就是2个不同的东西。。原先想翻为 等差，等比，但是例子不是。。n-plus,n-time。)

```
print "(t)imes or (p)lus: "
times = gets
print "number: "
number = gets.to_i

if times =~ /^t/
  puts((1..10).collect { |n| n*number }.join(", "))
else
  puts((1..10).collect { |n| n+number }.join(", "))
end
```

生成：

```
(t)imes or (p)lus: t
number: 2
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

如果最后一个实参 前面有 &， Ruby会认为这个是 Proc对象， ruby会把这个参数从 参数列表中移除，转换 Proc对象到 block，关联block到 方法。

这个技术能被用来增加语法糖 到 block的使用。
比如，有时，你想 拿到一个iterator，保存每个值 到数组中：

```
a = []
fibUpTo(20) { |val| a << val }      »     nil
a.inspect   »     "[1, 1, 2, 3, 5, 8, 13]"
```

我们定义一个方法 名字叫 into。它返回一个 block，用来填充数组。注意，此时，返回的block是一个 closure，它ref了 参数anArray ,即使into已经返回。

```
def into(anArray)
 return proc { |val| anArray << val }
end
fibUpTo 20, &into(a = [])
a.inspect       »       "[1, 1, 2, 3, 5, 8, 13]"
```

收集hash参数
使用 keyword argument， 而不是 按照顺序 数量。
。。现在应该有了，这本书上说1.6 不支持，计划在1.8中实现。。。现在都3.0 了。。

使用hash 来实现相同的效果。。

```
class SongList
  def createSearch(name, params)
    # ...
  end
end
aList.createSearch("short jazz songs", {
                   'genre'            => "jazz",
                   'durationLessThan' => 270
                   } )
```

k-v在最后，所有的 key => value 对 会被收集到一个hash，然后传递给方法，所以 可以不用{}

```
aList.createSearch("short jazz songs",
                   'genre'            => "jazz",
                   'durationLessThan' => 270
                   )
```

Expressions

```
a = b = c = 0    »       0
[ 3, 1, 7, 0 ].sort.reverse       »       [7, 3, 1, 0]
```

if和case 都返回 最后执行的表达式的值。

```
songType = if song.mp3Type == MP3::Jazz
         if song.written < Date.new(1935, 1, 1)
           Song::TradJazz
         else
           Song::Jazz
         end
        else
         Song::Other
        end

 rating = case votesCast
       when 0...10   then Rating::SkipThisOne
       when 10...50  then Rating::CouldDoBetter
```

```
    else          Rating::Rave
    end
```

许多操作符 实际上是 通过 调用方法来实现的
a*b+c， 执行 a的*方法，把b作为参数。 然后结果 在执行+方法，把c作为参数。等价于：(a.*(b)).+(c)

因为所有都是对象，且 你可以重定义实例方法， 所以你总是可以重定义 基本的算术运算，如果你不喜欢你获得的结果。

```
class Fixnum
  alias oldPlus +
  def +(other)
    oldPlus(other).succ
  end
end
1 + 2       »     4
a = 3
a += 4      »     8
```

。。原本的+，然后后移一个元素。

```
class Song
  def [](fromTime, toTime)
    result = Song.new(self.title + " [extract]",
                      self.artist,
                      toTime - fromTime)
    result.setStartTime(fromTime)
    result
  end
end
aSong[0, 0.15].play
```

Miscellaneous Expressions
各种各样的表达式

Command 表达式
用`` 包围字符串，或 使用 前缀%x， 它会将后续的string 作为一个 系统的 cmd命令来执行，结果输出到 cmd。新行不会被切分。

```
`date`       »     "Sun Jun  9 00:08:26 CDT 2002\n"
`dir`.split[34]   »     "lib_singleton.tip"
%x{echo "Hello there"}  »     "Hello there\n"
```

在cmd string中 可以使用表达式扩展 和 转义字符

```
for i in 0..3
  status = `dbmanager status id=#{i}`
  # ...
```

```
end
```

cmd的 exit status 可以在 全局变量$? 中获得。

被`` 包围的string 会被 作为cmd命令执行， 实际上是调用了 Kernel::` 方法。 可以重写

```
alias oldBackquote `
def `(cmd)
  result = oldBackquote(cmd)
  if $? != 0
    raise "Command #{cmd} failed"
  end
  result
end
print `date`
print `data`
```

生成：

```
Sun Jun  9 00:08:26 CDT 2002
prog.rb:3: command not found: data
prog.rb:5:in ``': Command data failed (RuntimeError)
    from prog.rb:10
```

赋值

返回右值最为 赋值的结果。 所以可以链式。

```
a = b = 1 + 2 + 3
a     »     6
b     »     6
a = (b = 1 + 2) + 3
a     »     6
b     »     3
File.open(name = gets.chomp)
```

2种基本的赋值 格式
把对象赋给 变量或常量

```
instrument = "piano"
MIDDLE_A   = 440
```

左值是 对象的属性 或元素。

```
aSong.duration    = 234
instrument["ano"] = "ccolo"
```

它们是通过 调用左值的方法来实现的，所以可以重载。

我们已经看过了 如何定义一个可写的对象属性。 只要定义个 方法名以=结尾的 方法，这个方法接受 赋值操作中的右值。

```
class Song
  def duration=(newDuration)
    @duration = newDuration
  end
end
```

setter的方法名 不需要和 属性名字一致。

```
class BrokenAmplifier
  attr_accessor :leftChannel, :rightChannel
  def volume=(vol)
    leftChannel = self.rightChannel = vol
  end
end
ba = BrokenAmplifier.new
ba.leftChannel = ba.rightChannel = 99
ba.volume = 5
ba.leftChannel    »     99
ba.rightChannel   »     5
```

ruby认为leftChannel 是一个 local变量 而不是一个 setter的方法调用

```
a, b = b, a

x = 0        »     0
a, b, c   =   x, (x += 1), (x += 1)        »     [0, 1, 2]
```

右侧的 按顺序执行。

左侧多，则 多余的左值被赋予nil
右侧多，则 右值被忽视

在1.6.2中，如果 一个左值，多个右值， 那么右值被转为数组 赋给左值。。
。。这个LT0002 Stephone Pochemann 的 python 就是这个吧。

如果最后一个 左值 以*开头，那么你能 打破 和扩展 数组， 通过 并行赋值， 所有剩余的右值会被作为一个数组 赋值给 那个左值。
。。多余的只有一个，也是 变成数组。

```
a = [1, 2, 3, 4]
b,   c = a    »      b == 1,      c == 2
b, *c = a     »      b == 1,      c == [2, 3, 4]
b,   c = 99,   a      »      b == 99,     c == [1, 2, 3, 4]
b, *c = 99,   a       »      b == 99,     c == [[1, 2, 3, 4]]
b,   c = 99, *a       »      b == 99,     c == 1
b, *c = 99, *a        »      b == 99,     c == [1, 2, 3, 4]
```

。。。*a 是什么

内嵌的赋值
左值可能包含一个 括号包围值的列表， ruby对待它们就像 它们是一个 嵌入的赋值语句。
ruby提取到相应的 右值，然后把值 赋值给 嵌入的东西， 在 外层的赋值 之前。

```
b, (c, d), e = 1,2,3,4  »     b == 1,     c == 2,     d == nil,    e == 3
b, (c, d), e = [1,2,3,4]      »     b == 1,     c == 2,     d == nil,    e == 3
b, (c, d), e = 1,[2,3],4      »     b == 1,     c == 2,     d == 3,      e == 4
b, (c, d), e = 1,[2,3,4],5    »     b == 1,     c == 2,     d == 3,      e == 5
b, (c,*d), e = 1,[2,3,4],5    »     b == 1,     c == 2,     d == [3, 4],      e ==
5
```

The second form is converted internally to the first. This means that operators that you have defined as methods in your own classes work as you'd expect.
。。感觉是指 第二个参数会 隐士转为 第一个参数类型，然后再执行 + 操作。

```
class Bowdlerize
  def initialize(aString)
    @value = aString.gsub(/[aeiou]/, '*')
  end
  def +(other)
    Bowdlerize.new(self.to_s + other.to_s)
  end
  def to_s
    @value
  end
end
a = Bowdlerize.new("damn ")   »     d*mn
a += "shame"      »     d*mn sh*m*
```

Conditional Execution

非nil，非false 就是true (0 is true)

IO#gets，读到就返回读取的行，到文件尾就返回nil，所以能：

```
while line = gets
  # process line
end
```

ruby支持所有的bool操作，并且引入了一个 defined? 操作。

and && and低于&&
or || or 低于 ||
and 优先级等于 or， &&优先级等于||
短路的。

not !, not 低

defined? 操作，返回nil 如果它的实参没有被定义，否则返回 实参的描述。
如果实参是 yield， defined? 返回"yield"，如果一个code block 关联到了当前上下文。

```
defined? 1  »     "expression"
defined? dummy    »     nil
defined? printf   »     "method"
defined? String   »     "constant"
defined? $&       »     nil
defined? $_       »     "global-variable"
defined? Math::PI       »     "constant"
defined? ( c,d = 1,2 )  »     "assignment"
defined? 42.abs   »     "method"
```

为了增加布尔操作，ruby提供了 额外的 比较函数，在Object 类中， 如==,===,<=>, =~,eql?,equal?。 <=>不是定义在Object类中，其他都是。
这些方法经常被子类重写。比如Array类重写了==，只有在 长度相同，且每个元素相同的情况下，返回true

Operator	Meaning
==	Test for equal value.
===	Used to test equality within a when clause of a case statement.
<=>	General comparison operator. Returns -1, 0, or +1, depending on whether its receiver is less than, equal to, or greater than its argument.
<, <=, >=, >	Comparison operators for less than, less than or equal, greater than or equal, and greater than.
=~	Regular expression pattern match.
eql?	True if the receiver and argument have both the same type and equal values. 1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object id.

==，=~ 有相反的操作， !=, !~。 ruby会自动转换代码， a!=b 等价于 !(a==b), a!~b等价于!(a=~b) 。 所以只要你重写了 == ~= 方法，那么你就自动拥有了 != ,!~ 方法， 同时也意味者，你无法 定义不依赖于 == =~ 的 != ~=。

ruby range 也可以作为一个 布尔表达式。 。 exp1..exp2 被eval为false，出给exp1变成true。 变真以后，会一直是真， 直到 exp2 eval出真。 如果这发生，那么range会重置。

你可以使用一个纯的RE 作为一个布尔表达式， ruby扩展它为 $_=~/re/

```
if aSong.artist == "Gillespie" then
  handle = "Dizzy"
elsif aSong.artist == "Parker" then
```

```
 handle = "Bird"
else
 handle = "unknown"
end
```

如果if分为多行，可以省略 then

```
if aSong.artist == "Gillespie"
 handle = "Dizzy"
elsif aSong.artist == "Parker"
 handle = "Bird"
else
 handle = "unknown"
end
```

if是一个表达式，不是一个语句

```
handle = if aSong.artist == "Gillespie" then
           "Dizzy"
         elsif aSong.artist == "Parker" then
           "Bird"
         else
           "unknown"
         end
```

ruby有一个 if 相反语义的东西 unless

```
unless aSong.duration > 180 then
 cost = .25
else
 cost = .35
end
```

三目

```
cost = aSong.duration > 180 ? .35 : .25
```

```
mon, day, year = $1, $2, $3 if /(\d\d)-(\d\d)-(\d\d)/
puts "a = #{a}" if fDebug
print total unless total == 0
```

if后面为真，前面的表达式才会执行。
unless 反之。

```
while gets
  next if /^#/            # Skip comments
  parseLine unless /^$/   # Don't parse empty lines
end
```

if是一个表达式，所以 可以从 语句中获得它：

```
if artist == "John Coltrane"
```

```
  artist = "'Trane"
end unless nicknames == "no"

This path leads to the gates of madness.
。。。。通向疯狂之门

case 表达式，是一个 多重if

case inputLine
  when "debug"
    dumpDebugInfo
    dumpSymbols
  when /p\s+(\w+)/
    dumpVariable($1)
  when "quit", "exit"
    exit
  else
    print "Illegal command: #{inputLine}"
end

kind = case year
         when 1850..1889 then "Blues"
         when 1890..1909 then "Ragtime"
         when 1910..1929 then "New Orleans Jazz"
         when 1930..1939 then "Swing"
         when 1940..1950 then "Bebop"
         else                 "Jazz"
       end
如果一行，则需要 then

对case后的表达式进行计算，得出的结果 与 when后的比较表达式 进行 === 比较。
只要类定义了===操作 （所有内置类都定义了这个操作），那么就能作为case 表达式的对象。

RE 定义=== 作为一个 简单模式匹配
case line
  when /title=(.*)/
    puts "Title is #$1"
  when /track=(.*)/
    puts "Track is #$1"
  when /artist=(.*)/
    puts "Artist is #$1"
end

ruby的类 是 Class类的 实例，也定义了===，来 测试 实参是不是 类 或类的超类 的 一个实例

case shape
  when Square, Rectangle
```

```
    # ...
  when Circle
    # ...
  when Triangle
    # ...
  else
    # ...
end


while gets
  # ...
end
```

until 是 while 的 反义

```
until playList.duration > 60
  playList.add(songList.pop)
end
```

就像if unless, while 和until 也都可以作为 statement modifier (语句修饰)

```
a *= 2 while a < 100
a -= 10 until a < 100


file = File.open("ordinal")
while file.gets
  print  if /third/ .. /fifth/
end
```

。。。。？？？ 这个确实，好像不是范围， 而是之前说的， 直到 exp1为true，开始执行，然后 直到 exp2为true，停止执行。。 确实啊，range 只需要 start，end。 中间不管了，让外面的管。

```
file = File.open("ordinal")
while file.gets
  print if ($. == 1) || /eig/ .. ($. == 3) || /nin/
end

print "Hello\n" while false
begin
  print "Goodbye\n"
end while false
```

当 while until 作为 语句修饰。 如果 语句是 begin/end 块包围的， 那么 这个块中的代码至少会被执行一次。。。。。。do..while....

。。GoodBye 被输出一次。

```
3.times do
  print "Ho! "
```

```
end
```

作为times的附加：整型可以循环通过 downto,upto,step。
下面是一个for 0到9的循环
。包含9

```
0.upto(9) do |x|
  print x, " "
end
```

0到12，步长3 的循环：

```
0.step(12, 3) {|x| print x, " " }
```

。包含12

```
[ 1, 1, 2, 3, 5 ].each {|val| print val, " " }
```

一旦一个类支持 each， 那么 Enumerable 模块中的 方法 就可用了。

比如，File 类提供了 each 方法，这个方法返回文件的每行， 使用Enumrable中的grep方法，我们能遍历 满足条件的 行

```
File.open("ordinal").grep /d$/ do |line|
  print line
end
```

内置迭代loop。 死循环。

```
loop {
  # block ...
}
```

```
for aSong in songList
  aSong.play
end
```

ruby会把上面转换成下面：

```
songList.each do |aSong|
  aSong.play
end
```

each和for的唯一不同是 body中定义的local变量的 scope。

能用for 遍历任何 实现了each方法的 对象。

```
for i in ['fee', 'fi', 'fo', 'fum']
  print i, " "
end
for i in 1..3
  print i, " "
end
for i in File.open("ordinal").find_all { |l| l =~ /d$/}
  print i.chomp, " "
```

```
end
```

只要你的类定义了一个显示的 each 方法，能使用for traverse 它。

```
class Periods
  def each
    yield "Classical"
    yield "Jazz"
    yield "Rock"
  end
end

periods = Periods.new
for genre in periods
  print genre, " "
end
```

break,redo,next

break,中断现在的循环， 从 被中断的block 后面开始执行。
redo 重新执行循环，不会重新eval 循环条件， 也不会 拿下一个 迭代元素。
next，跳到 loop的 最后，开始下次循环。

```
while gets
  next if /^\s*#/   # skip comments
  break if /^END/   # stop at end
                    # substitute stuff in backticks and try again
  redo if gsub!(/`(.*?)`/) { eval($1) }
  # process line ...
end
```

这3个关键字 能用在任何 基于迭代的 循环结构中

```
i=0
loop do
  i += 1
  next if i < 3
  print i
  break if i > 4
end
```

redo导致循环重复当前迭代， 有时，你需要回到最开始 来开始循环，retry语句是用来完成这种的。
retry重新开始 任何类型的 迭代循环。

```
for i in 1..100
  print "Now at #{i}. Restart? "
  retry if gets =~ /^y/i
end
```

retry会重新eval 所有迭代的实参，然后开始迭代。

```
def doUntil(cond)
  yield
  retry unless cond
end

i = 0
doUntil(i > 3) {
  print i, " "
  i += 1
}
```

。。i>3被重新eval。

while,until,for 内置在语言中，不会引入新的 作用域， 之前存在的local变量 能在循环中使用， 循环中创建的的local变量 在之后能被访问到

。。所以不引入作用域， 就没有作用域。

迭代(loop,each) 中使用的block 有些不同， 在这些block中创建的local变量，在外面无法访问到。

```
[ 1, 2, 3 ].each do |x|
  y = x + 1
end
[ x, y ]
```

产生：

```
prog.rb:4: undefined local variable or method `x'
for #<Object:0x401c2ce0> (NameError)
```

如果使用的是之前已经定义的同名变量，则是更新之前的同名变量，而不是新创建一个。

```
x = nil
y = nil
[ 1, 2, 3 ].each do |x|
  y = x + 1
end
[ x, y ]    »    [3, 4]
```

Exceptions, Catch, and Throw

Exception类 或它的子类 的对象 包含了 异常的信息。

如果自定义的话，你可能需要 自定义的类 成为 StandardError 或它的子类的 的一个子类。如果不这样做，默认情况下，你的异常不会被 捕捉

每个Exception 都有一个string类型的消息 和 一个 栈信息。

目前，通过tcp socket下载歌曲：

```
opFile = File.open(opName, "w")
while data = socket.read(512)
  opFile.write(data)
end
```

如果半路发生一个错误，我们不希望存储 半首歌。
我们增加一些错误处理机制。我们用 begin/end 块 包围 可能抛出异常的代码， 使用 rescue（营救） 从句 来告知 ruby 我们想要处理的异常的类型。 在这里，我们对 SystemCallError异常感兴趣。
在错误处理block中，我们 报告错误，关闭 和删除 文件，重新抛出异常。

```
opFile = File.open(opName, "w")
begin
  # Exceptions raised by this code will
  # be caught by the following rescue clause
  while data = socket.read(512)
    opFile.write(data)
  end

rescue SystemCallError
  $stderr.print "IO failed: " + $!
  opFile.close
  File.delete(opName)
  raise
end
```

当异常发生时，ruby会把 全局变量 $! ref到异常，这个动作独立于任何后续的(subsequent)异常处理

raise 会重新抛出 $! 中的异常

可以有多个rescue 从句，在一个begin块中，每个rescue从句 能指定 多个需要捕获的 异常。， 在rescue的最后，你可以告诉ruby 一个 local变量 来接受 匹配的异常。这个local变量 比$! 更具有可读性。

```
begin
  eval string
rescue SyntaxError, NameError => boom
  print "String doesn't compile: " + boom
rescue StandardError => bang
  print "Error running script: " + bang
end
```

按序执行rescue，如果匹配，就执行，不再继续匹配。
是否匹配靠的是 $!.kind_of?(parameter) 判断。 如果parameter是 异常的类或异常的父类，就匹配成功。
如果rescue 不写 参数列表， 那么就默认 匹配 StandardError

没有匹配的，就 查看caller 的 异常处理，然后caller 的caller 的异常处理。。。。

rescue后面可以放 任意的表达式(包括方法调用)，只要返回 一个 异常的类。

有时，你需要确保一些操作 在block之后 被执行， 无论block中是否 抛出异常。 这时需要 ensure 从句。

ensure放在 最后一个 rescue之后，包含一些代码，这些代码始终都会被执行。

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close unless f.nil?
end
```

else从句是一个类似的，但是更少用的。放在 rescue后面 ，ensure前面。
else的body 只有在 没有异常产生时 才会执行。

```
f = File.open("testfile")
begin
  # .. process
rescue
  # .. handle error
else
  puts "Congratulations-- no errors!"
ensure
  f.close unless f.nil?
end
```

有时，你能够修复异常的原因， 在这些情况下，你可以在 rescue 中使用 retry 语句 来 重复整个 begin/end 块
这里存在一个 tremendous(巨大，极好，精彩，了不起的) scope for infinite loop，所以需要注意。
。。感觉可能是说， 这可能导致 死循环。。

```
@esmtp = true
begin
  # First try an extended login. If it fails because the
  # server doesn't support it, fall back to a normal login
  if @esmtp then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end
rescue ProtocolError
  if @esmtp then
```

```
    @esmtp = false
    retry
  else
    raise
  end
end
```

第一次使用EHLO命令，第二次使用HELO命令，如果第二次失败，抛出异常。

可以通过 Kernel:raise 方法来 抛出异常。

```
raise
raise "bad mp3 encoding"
raise InterfaceException, "Keyboard failure", caller
```

第一种，抛出现有的异常($!中的) 如果现在没有异常，就抛出 RuntimeError
第二种，新建一个 RuntimeError异常，设置它的消息为 给出的string，然后抛出
第三种，第一个参数 创建一个异常，然后设置 相关的信息为 第二个参数，栈信息为第三个参数。。第一个参数可以是 Exception的类名 也可以是 那些类的一个实例。（严格上来说，第一个参数可以是任何对象，只要这个对象 object.kind_of?(Exception) 成立）

```
raise

raise "Missing name" if name.nil?

if i >= myNames.size
  raise IndexError, "#{i} >= size (#{myNames.size})"
end

raise ArgumentError, "Name too big", caller
```

最后一个例子中，我们移除了当前的程序(routine) 从栈中。
下面移除了 2个程序 从栈中。

```
raise ArgumentError, "Name too big", caller[1..-1]
```

定义自己的异常类，来保存 任何你想要的信息

```
class RetryException < RuntimeError
  attr :okToRetry
  def initialize(okToRetry)
    @okToRetry = okToRetry
  end
end

def readData(socket)
  data = socket.read(512)
  if data.nil?
    raise RetryException.new(true), "transient read error"
```

```
  end
  # .. normal processing
end

begin
  stuff = readData(socket)
  # .. process stuff
rescue RetryException => detail
  retry if detail.okToRetry
  raise
end
```

Catch and Throw

raise 和 rescue的架构 很适合 放弃执行。

有时，我们更希望在 正常执行中 跳出 比较深的 嵌套的结构。 可以使用 catch，throw

```
catch (:done)  do
  while gets
    throw :done unless fields = split(/\t/)
    songList.add(Song.new(*fields))
  end
  songList.play
end
```

catch 定义一个block，块 被 给定的名字标记(符号或string)。block 正常执行，直到碰到 throw。

当ruby碰到一个 throw，它在调用栈信息中找 匹配的 catch block。 如果找到，调用栈开始pop，终止block(应该是指 :done 这个block)。

如果throw 有第二个 可选的参数，这个值 作为catch的返回值。

下面的例子中，使用throw来终止 和用户的交互，如果"!" 被输入。

```
def promptAndGet(prompt)
  print prompt
  res = readline.chomp
  throw :quitRequested if res == "!"
  return res
end

catch :quitRequested do
  name = promptAndGet("Name: ")
  age  = promptAndGet("Age:  ")
  sex  = promptAndGet("Sex:  ")
  # ..
  # process information
```

end

。。goto。

Modules