# 开发环境

搜索

find /var -name mysql.sock

## 搭建开发环境

换ruby源

|  |
| --- |
| gem sources --remove https://rubygems.org/  gem sources -a http://ruby.taobao.org/  gem sources -l |
| gem sources --remove http://ruby.taobao.org/  gem sources -a https://rubygems.org/  gem sources -l |

安装rvm

|  |
| --- |
| curl -L https://get.rvm.io | bash -s stable |

安装ruby

|  |
| --- |
| rvm install 1.9.3 |

设置default ruby

|  |
| --- |
| rvm use 1.9.3 --default |

安装rvm和ruby

|  |
| --- |
| curl -L https://get.rvm.io | bash -s stable --ruby=1.9.3 |

安装rails

|  |
| --- |
| sudo gem install rails -v 3.2.13 |

git

|  |
| --- |
| 在CentOS上安装Git  http://www.ccvita.com/370.html |

debug

安装 libyaml

|  |
| --- |
| rvm pkg install libyaml |

|  |
| --- |
| No source for ruby-1.9.3-p484 provided with debugger-ruby\_core\_source gem. |
| gem install debugger-linecache -v '1.1.2' -- --with-ruby-include=\$rvm\_path/src/ruby-1.9.3-p484 |

|  |
| --- |
| mysql.h is missing. please check your installation of mysql and try again. |
|  |

## Ruby版区别

## 控制台

进入控制台用irb, 退出控制台:

|  |  |
| --- | --- |
| windows | 输入 irb 指令直接进入此互动模式  ctrl+D或输入exit退出互动模式 |
| linux | ctrl+Z |
| mac |  |

# 数字

# 日期 & 时间

# 字符串

### 单引号和双引号的不同效果

|  |  |
| --- | --- |
| ruby> **puts "a\nb\nc"**  a  b  c  **nil** | ruby> **puts 'a\nb\n'**  a\nb\nc  **nil** |
| ruby> **"\n"**  **"\n"** | ruby> **'\n'**  **"\\n"** |
| ruby> **"\001"**  **"\001"** | ruby> **'\001'**  **"\\001"** |
| ruby> **"abcd #{5\*3} efg"**  **"abcd 15 efg"** |  |
| ruby> **var = " abc "**  **" abc "**  ruby> **"1234#{var}5678"**  **"1234 abc 5678"** |  |

### 用 + 串接

|  |
| --- |
| ruby> **"foo" + "bar"**  **"foobar"** |

### 用 \* 重复字串

|  |
| --- |
| ruby> **"foo" \* 2**  **"foofoo"** |

### 提取字元

|  |
| --- |
| ruby> **word[0]**  **102            # 102 是 `f' 的 ASCII 码 （编注：Ruby 1.9 之后版本，变更为回传 f）**  ruby> **word[-1]**  **111            # 111 是 `o' 的 ASCII 码（编注：Ruby 1.9 之后版本，变更为回传 o）**  （负号指从字串结尾计算偏移位置，而不是从字串开头。） |

### 用索引号码提取子字串

|  |
| --- |
| ruby> **herb = "parsley"**  **"parsley"**  ruby> **herb[0,1] #第0位开始, 提取1位**  **"p"**  ruby> **herb[-2,2] #倒数第二位开始, 往后提取两位**  **"ey"**  ruby> **herb[0..3] #第0位开始到第3位, 包含第3位**  **"pars"**  ruby> **herb[-5..-2] #倒数第五位开始, 到倒数第二位, 包括倒数第二位**  **"rsle"** |

### 比对

|  |
| --- |
| ruby> **"foo" == "foo"**  **true**  ruby> **"foo" == "bar"**  **false** |

### 例子:

#### 猜字

|  |  |
| --- | --- |
| 1. words = ['foobar', 'baz', 'quux'] 2. secret = words[rand(3)] 3. print "guess?" 4. while guess = STDIN.gets ## 5. guess.chop! 6. if guess == secret 7. puts "You win!" 8. break 9. else 10. puts "Sorry, you lose." 11. end 12. print "guess?" 13. end 14. puts "the word is "+ secret + " . " | guess=STDIN.gets 既是一个有动作的叙述（收集使用者输入的一行内容，并储存为 guess），也是一项判断条件（若没有输入任何内容，guess 即等于整个 guess=STDIN.gets 表示式的值，而这个值是 nil，会让 while 停止回圈。）  STDIN 是标准输入 (standard input) 物件。一般来说，guess=gets 的功能与 guess=STDIN.gets 一样。  guess.chop! 会删除 guess 后的最后一个字元；在本例中这是一个换行 (newline) 字元，这是因为 gets 会包含使用者按下的 Return 键，这是我们不感兴趣的。  14行将 secret 写为 #{secret} 的单一字串效果相同 |

#### 字符串与数组 互相转换

|  |
| --- |
|  |

# 数组

### 创建数组

|  |
| --- |
| a = ['ant', 'bee', 'cat', 'dog'] |
| a = %w{ant bee cat dog} |

数组与字串一样，能够串接与重复。

### +串接

|  |  |  |
| --- | --- | --- |
|  | **表达式** | **结果** |
| **定义** | **ary = [1, 2, "3"]** | **[1, 2, "3"]** |
| **用"+"连接两个数组** | **ary = [1, 2, "3"]**  **ary + ["foo", "bar"]** | **[1, 2, "3", "foo", "bar"]** |
| **用"\*"重复一个数组** | **ary = [1, 2, "3"]**  **ary \* 2** | **[1, 2, "3", 1, 2, "3"]** |

### 用索引号码提取子数组

ary = [0, 1, 2, 3, 4, 5, 6]

|  |  |  |
| --- | --- | --- |
|  | 结果 | 备注 |
| **ary[0]** | 0 |  |
| **ary[-2]** | **2** | 从后面数, 第N位 |
| **ary[0, 2]** | **[0, 1]** | 第N位开始, 提取K个 |
| **ary[0..2]** | **[0, 1, 2]** | 从N位到K位, 包含K位 |
| **ary[0...2]** | **[0, 1]** | 从N位到K位, 不包含K位 |

### 替换原数组制定区间元素

ary = [1, 3, 5, 7, 9]

|  |  |  |
| --- | --- | --- |
| a[2, 2] = 'cat' | [1, 3, "cat", 9] |  |
| a[2, 0] = 'dog' | [1, 3, "dog", "cat", 9] |  |
|  |  |  |

|  |
| --- |
| ruby>  **1**  ruby>    ruby>    ruby>    ruby>    ruby>    （负号指从数组结尾计算偏移位置 (offset)，而不是从数组开头。） |

### 数组--join-->字符串

|  |
| --- |
| ruby> **str = ary.join(":")**  **"1:2:3"** |

### 字符串--split-->数组

|  |
| --- |
| ruby> **str.split(":")**  **["1", "2", "3"]** |

### 迭代数组

|  |
| --- |
| 搜索歌曲名称为title 的歌曲  def with\_title(title)  @songs.find{|song| title == song.name}  end |
| 备注: find是一种迭代器, 反复调用block中的代码. |

# 哈希

Hashes, 也称关联数组, 图, 词典

哈希是无序的

+

#### 创建哈希

|  |
| --- |
| 使用Hash的[]方法来创建一个hash:  a1 = Hash.[]("flat",3,"curved",2)  a2 = Hash.[]("flat"=>3,"curved"=>2)  b1 = Hash["flat",3,"curved",2]  b2 = Hash["flat"=>3,"curved"=>2]  c2 = {"flat"=>3,"curved"=>2} |
| 使用new来创建一个hash  d = Hash.new # 创建一个空的hash  e = Hash.new(99) # 创建一个默认值为99的hash  e["angled"] # 99  e.inspect # {}  f = Hash.new("a"=>3) #  f["b"] # {"a"=>3} (刚好默认值也是一个hash)  f.inspect # {} |
| 由一个数组来创建hash   1. array = [2, 3, 4, 5, 6, 7] 2. hash = Hash[\*array] 3. # hash is now: {2=>3, 4=>5, 6=>7} |

#### 使用哈希

|  |
| --- |
| a['name'] |

#### 设置哈希的默认值

|  |
| --- |
| 用一个散列表没有包含的键进行索引, 返回nil  如果不想返回nil, 可以设置默认值  user = Hash.new(0);  user['name'] -->0  user['name'] = user['name'] + 1  user['name'] -->1  a = Hash.new("missing") # default value object is "missing"  a["hello"] # "missing"  a.default="nothing" # 修改默认值  a["hello"] # "nothing"  a["good"] << "bye" # "nothingbye"  a.default # "nothingbye" |
| 如果当一个key不存在时，你想抛出一个异常来代替返回默认值的话，你可以使用fetch方法,当你使用fetch 方法的时候如果key不存在，它则会抛出一个IndexError异常，它还有第二个参数，这个参数就是当你key不存在的时候所返回的值：   1. a = {"flat"=>3,"curved"=>2,"angled"=>5} 2. a.fetch("pointed")                   # IndexError 3. puts a.fetch("curved","na")               # 2 4. puts a.fetch("x","na")                    # "na" 5. puts a.fetch("flat") {|x| x.upcase}       # 3 6. puts a.fetch("pointed") {|x| x.upcase}    # "POINTED" |

#### 存取或者加一个键值对

|  |
| --- |
|  a = {}   a["flat"] = 3        # {"flat"=>3}   a.[]=("curved",2)    # {"flat"=>3,"curved"=>2}   a.store("angled",5)  # {"flat"=>3,"curved"=>2,"angled"=>5}  在这里要注意，nil对象也可以作为一个健或者值   1. b={} 2. b[2]      # nil 3. b[3]=nil 4. b         # {3=>nil} 5. b[2].nil? # true 6. b[3].nil? # true 7. b[nil]=5 8. b         # {3=>nil,nil=>5} 9. b[nil]    # 5 10. b[b[3]]   # 5 |

#### 删除一个键值对

|  |
| --- |
| 删除的话我们能够使用 clear, delete, delete\_if, reject, reject!, 和shift 这些方法都能做到。  clear 将会把一个hash清空.  shift则将会删除掉最前面的一个键值对：   1. a = {1=>2, 3=>4} 2. b = a.shift       # [1,2] a现在是[3=>4]   delete删除一个指定位置的键值对，如果不存在这个位置，则将会返回block里的字符串：   a = {1=>1, 2=>4, 3=>9, 4=>16}   a.delete(3)                     # 9   # a is now {1=>1, 2=>4, 4=>16}   a.delete(5)                     # nil in this case   a.delete(6) { "not found" }     # "not found" |

#### 哈希和数组的比较

|  |
| --- |
| 数组用数字序列的索引号码提取内容  哈希用一个*键 (key)* 提取内容 |

#### 哈希键:

|  |  |  |
| --- | --- | --- |
| String字串 | **h = {1 => 2, "2" => "4"}** |  |
| Symbol | h = {:foo => 1, :bar => 2 } | 简洁 |
| Fixnum 整数 |  |  |

|  |
| --- |
| ruby> **h = {1 => 2, "2" => "4"}**  **{1=>2, "2"=>"4"}**  ruby> **h[1]**  **2**  ruby> **h["2"]**  **"4"**  ruby> **h[5]**  **nil**  ruby> **h[5] = 10    # 加入一个项目**  **10**  ruby> **h**  **{5=>10, 1=>2, "2"=>"4"}**  ruby> **h.delete 1   # 用 Key 删除一个项目**  **2**  ruby> **h[1]**  **nil**  ruby> **h**  **{5=>10, "2"=>"4"}** |

#### 迭代一个hash

|  |
| --- |
| 我们能够使用each方法来做这个，它还有each\_key, each\_pair和 each\_value方法，其中each\_pair是each方法的别名：  {"a"=>3,"b"=>2}.each do |key, val|  print val, " from ", key, "; " # 3 from a; 2 from b;  end |
| each\_key 和each\_value的block只有一个参数  {"a"=>3,"b"=>2}.each\_key do |key|  print "key = #{key};" # Prints: key = a; key = b;  end    {"a"=>3,"b"=>2}.each\_value do |value|  print "val = #{value};" # Prints: val = 3; val = 2;  end |

#### 反转hash

|  |
| --- |
| 也就是将一个hash的键和值交换  a = {"fred"=>"555-1122","jane"=>"555-7779"}  b = a.invert  b["555-7779"] # "jane" |
| 注意这里有一个问题，那就是值有可能是重复的，这时如果你使用invert的话，它将会删除掉一个重复的键值对，后面的将覆盖前面的：  a = {"fred"=>"555-7779","jane"=>"555-7779"}  b = a.invert  puts b["555-7779"] # "jane" |

#### 检测一个健或者值是否属于一个hash

|  |
| --- |
| 我们能够使用has\_key?或者它的任意一个别名，include?, key?, 和member?来测试一个key是否属于一个hash:  a = {"a"=>1,"b"=>2}  a.has\_key? "c" # false  a.include? "a" # true  a.key? 2 # false  a.member? "b" # true |
| 如果想测试值的话，就使用has\_value?或者value?:   1. a.has\_value? 2        # true 2. a.value? 99           # false |

#### 你还能使用empty? 方法来测试一个hash是否为空，length来得到一个hash的大小：

|  |
| --- |
| 1. a.empty?        # false 2. a.length        # 2 |

#### 将一个hash转换为一个数组

|  |
| --- |
| 可以使用to\_a来将一个hash转换成一个数组：   h = {"a"=>1,"b"=>2}   h.to\_a        # ["a",1,"b",2] |
| 如果只想把值或者健转换成数组，可以使用keys或者values ：   1. h.keys         # ["a","b"] 2. h.values       # [1,2] |
| 最终你还可以基于一个健序列，来将他们所对应的值转换换成数组：   1. h = {1=>"one",2=>"two",3=>"three",4=>"four","cinco"=>"five"} 2. h.values\_at(3,"cinco",4)     # ["three","five","four"] 3. h.values\_at(1,3)             # ["one","three"] |

#### 根据条件选择键值对

|  |
| --- |
| Hash类混入了Enumerable 模块，因此你能够使用detect (find), select, (find\_all), grep, min, max, 和reject 这些方法.   detect (别名find)得到一个单独的键值对：  Java代码   1. names = {"fred"=>"jones","jane"=>"tucker", 2. "joe"=>"tucker","mary"=>"SMITH"} 3. # 得到一个tucker 4. names.detect {|k,v| v=="tucker" }    # ["joe","tucker"] 5. # 得到一个大写的 6. names.find {|k,v| v==v.upcase }    # ["mary", "SMITH"]   select(别名finad\_all)将会返回所有符合条件的键值对：  Java代码   1. names.select {|k,v| v=="tucker" } 2. # [["joe", "tucker"], ["jane", "tucker"]] 3. names.find\_all {|k,v| k.count("r")>0} 4. # [["mary", "SMITH"], ["fred", "jones"]] |

#### 排序一个hash

|  |
| --- |
| ruby排序的时候是将hash转换为一个数组，然后才进行排序的：   1. names = {"Jack"=>"Ruby","Monty"=>"Python", 2. "Blaise"=>"Pascal", "Minnie"=>"Perl"} 3. list = names.sort 4. # list is now: 5. # [["Blaise","Pascal"], ["Jack","Ruby"], 6. #  ["Minnie","Perl"], ["Monty","Python"]] |

#### 合并两个散列

|  |
| --- |
| ruby能够使用merge（别名update）方法，来合并两个hash，如果他们中的健有相同的话，一个将会被删除：  Java代码   1. dict = {"base"=>"foundation", "pedestal"=>"base"} 2. added = {"base"=>"non-acid", "salt"=>"NaCl"} 3. new\_dict = dict.merge(added) 4. # {"base"=>"non-acid", "pedestal"=>"base", "salt"=>"NaCl"}   merge还可以使用block来处理，两个健有冲突的hash的合并，我们下面的代码是假设我们有两个冲突的健的话，我们选择值较小的那个键值对：  Java代码   1. dict = {"base"=>"foundation", "pedestal"=>"base"} 2. added = {"base"=>"non-acid", "salt"=>"NaCl"} 3. new\_dict = dict.merge(added) {|key,old,new| old < new ? old : new } 4. # {"salt"=>"NaCl", "pedestal"=>"base", "base"=>"foundation"} |

#### 计算hash键的差集和交集

|  |
| --- |
| 由于hash的key能够被转换成一个数组，因此我们能够使用一些数组的方法：  a = {"a"=>1,"b"=>2,"z"=>3}  b = {"x"=>99,"y"=>88,"z"=>77}  intersection = a.keys & b.keys  difference = a.keys - b.keys  c = a.dup.update(b)  inter = {}  intersection.each {|k| inter[k]=c[k] }  # inter is {"z"=>77}  diff={}  difference.each {|k| diff[k]=c[k] }  # diff is {"a"=>1, "b"=>2} |

使用hash作为一个Sparse Matrix（稀疏矩阵）

|  |
| --- |
| 1. cube = Hash.new(0) 2. cube[[2000,2000,2000]] = 2 3. z = cube[[36,24,36]]       # 0 |

## True、False和Nil

# 正则表达式

正则表达式通常写在斜杠之间, 类似: /Perl | Python/

字元及字元组合具有特殊意思，包括：

|  |  |  |
| --- | --- | --- |
| [] | 指定的范围（例如：[a-z] 表示一个在 a 到 z 的范围内的字母） |  |
| \w | 一般字元 (word character)，即 [0-9A-Za-z\_] |  |
| \W | 非一般字元 (non-word character) |  |
| \s | 空白字元 (space character)，即 [ \t\n\r\f] | 制表符/回车符/空格符 |
| \S | 非空白字元 (non-space character) |  |
| \d | 数字 (digit character)，即 [0-9] |  |
| \D | 非数字 (non-digit character) |  |
| \b | 退位 (0x08)（仅用于指定的范围） |  |
| \b | 单字边界（若不是于指定的范围） |  |
| \B | 非单字边界 |  |
| \* | 前一符号的内容出现 0 或数次。 | /ab\*c/ 匹配一个含有a, 后面跟零个或多个b, 其后是c的字符串 |
| + | 前一符号的内容出现 1 或数次。 | /ab+c/ 匹配一个含有a, 后面跟一个或多个b, 其后是c的字符串 |
| {m,n} | 前一符号的内容，最少出现 m 次，最多出现 n 次。 |  |
| ? | 前一符号的内容最多出现一次，同 {0,1} |  |
| | | 符合前一个或后一个表示式, 即: 要么左边, 要么右边 | /aaa | bbb/ 要么aaa, 要么bbb |
| () | 分组 | /P(aaa | bbb)/ 要么Paaa, 要么Pbbb |

一般称为正则表示式 (regular expressions)。都是用斜线 (/) 包住样式内容，而不用双引号。

## 例子

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

|  |
| --- |
| ruby> **"abcdef" =~ /d/**  **3**  ruby> **"aaaaaa" =~ /d/**  **nil** |

### 例子1: 是否包含某些字符串

|  |
| --- |
| 是否含有一个十六进位数  **def chab(s)**  **(s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil**  **end** |
| **chab "Not this one." -->** false  **chab "Maybe this?{0x35}"    -->** false  **chab "Or this?<0x38z7e>"   -->** false  **chab "Okay, this: <0xfc0004>." -->true** |

### 例子2: 替换字符串中某些字符

|  |
| --- |
| str.sub(/perl/, "ruby") #替换一个 |
| str.gsub(/perl/, "ruby") #替换所有 |

# 控制结构

## 条件式

### if

unless 就是 if 的否定型

单行if语句

|  |
| --- |
| puts "It's zero." **if** i**==**0 |
| if i=0 then puts "It's zero."; end |

多行if语句

|  |
| --- |
| i=0  if i**==**0  puts "It's zero."  elsif i**==**1  puts "It's 1."  else  puts ""  end |

例子:

1. 阶乘

|  |  |
| --- | --- |
| def fact(n)    if n == 0      1    else      n \* fact(n-1)    end  end | def 用来定义函数 (function) 的叙述  函数 fact 有一个引数 (argument)，即是 n  第一个 end 结束 if 叙述。  第二个 end 结束 def 叙述。 |

### unless

|  |
| --- |
| unless 条件 ============== if not(条件) |

### case

|  |
| --- |
| **case** i  **when** 1, 2..5  puts "1..5"  **when** 6..10  puts "6..10"  else  puts "other"  **end** |
| **case** 'abcdef'  **when** 'aaa', 'bbb'  puts "aaa or bbb"  **when** /def/  puts "includes /def/"  **end** |

## 循环

共有四种方式能够从内部中断回圈。

|  |
| --- |
| 第一种是 break，与 C 的一样，表示完全逸出回圈； |
| 第二种是 next，跳至回圈下一个迭代 (iteration) 的起始处（等同于 C 的continue）； |
| 第三种是 redo，可重新开始目前的回圈, redo后面只能跟end, 不能跟其它代码。 |
| 第四种从内部中断回圈的方式是 return。return 不单能逸出回圈，更会逸出包含回圈的方法。若提供一个参数值，则此方法会回传这个值，否则会传回 nil。 |

### while

until 则是 while 的否定型

while 就是重复的 if

单行while

|  |
| --- |
| i**=**0  puts i**+=**1 **while** i**<**3 |

多行while

|  |
| --- |
| s="abc";  i**=**0  **while** i**<**s.length  printf "<%c>", s[i];  i**+=**1  **end**; |

### until

|  |
| --- |
| until (直到...) 循环 |

### for

for 其实就是 each 的另一种写法

|  |
| --- |
| **for** elt **in** collection  *# ... elt 是集合物件中的一个元素*  **end** |
| **for** elt **in** [100,**-**9.6,"pickle"]  puts "#{elt}\t(#{elt.class})"  **end**  **结果**  100 (Fixnum)  -9.6 (Float)  pickle (String) |
| **for** num **in** (4..6)  puts num  **end** |
|  |

### each (迭代iterate)

for 其实就是 each 的另一种写法

|  |
| --- |
| [1,2,3,4,5,6,7,8,9].each do |x|  puts x  end |
| "abc".**each\_byte**{**|**c**|**  printf "<%c>", c  }  结果  <a><b><c> |
| str **=** "a\nb\nc\n"  str.**each\_line**{**|**l**|**  print l  } |

## retry/ redo/ yield/break/next

#### retry

以重新开始 begin 程式码

|  |
| --- |
| begin  c**=**0  **for** i **in** 0..4  print i  **if** i **==** 2 **and** c **==** 0  c **=** 1  print "\n"  **retry**  **end**  **end**  **输出结果**  012  01234 |

#### redo

从循环头重新执行循环, 但不重计算循环条件表达式或者获得迭代中的下一个元素.

redo， 类似next

但是他**继续下次循环的时候， 不检查条件**

|  |
| --- |
| i = 5    while i > 0  i=i-1    if i == 0  redo  end    puts i  end |

#### yield

yield 可将控制移至传递往迭代器的程式码区块

|  |
| --- |
| **def repeat**(*num*)  **while** num **>** 0  **yield** *#转向程序区块 puts "foo"*  num **-=** 1  **end**  **end**  repeat(3) { puts "foo" }  **输出:**  foo  foo  foo |
| **yield传参数**  **def call\_block**  **yield**("hello", 99)  **end**    call\_block{**|**str, num**|** puts str} |

#### break

终止最接近的封闭循环体, 然后继续执行block后面的语句

#### next

跳到本次循环的末尾, 并开始下一次迭代.

|  |
| --- |
| i**=**0  loop **do**  i **+=** 1  **next if** i**<**3  print i  **break if** i**>**4  **end**  **输出**  **345** |

## 代码块

## 改变控制流

# 异常

|  |
| --- |
| **begin**      file = open("/tmp/some\_file", "w") **rescue**    **puts "An error occurred: #{$!}"**  ensure  file.close **end** |

## BEGIN和END

## 线程、纤程和连续体

# 类

## 定义一个简单类

|  |
| --- |
| **class** Song  *#初始化方法, 在用Song.new("youxuan", 19)时调用*  **def initialize**(*name*, *age*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  **end**    *#重写to\_s方法*  **def to\_s**  "name:#@name, age:#@age"  **end**  *#自定义方法*  **def speak**  puts "Bow Wow"  **end**  **end** |
| song **= Song**.**new**("youxuan", 19)  puts song #song等同于song.to\_s  **song.speak** |

## 常量和变量

|  |  |  |
| --- | --- | --- |
| $ | 全域变数 (global variable) | 初始化前，全域变数有一个特别的值－－nil  应尽量少用全域变数， 因为全域变数能够编写于任何地方，所以相对危险。过量使用全域变数会令人难以找出错误  全域变数其中一个优点就是能够追踪，每当变数的值改变后，你都能够指出呼叫出的程序。例如，可用于为图形使用者介面 (GUI) 保持最新状态。  ruby> trace\_var :$x, Proc.new { puts "$x is now #{$x}" }  nil  ruby> $x = 5  $x is now 5  5 |
| @ | 实例变数 (instance variable) | 作用域仅限定于 self 指示的物件  除非程式员明确地提供其他方法，否则实例变数无法从物件以外变更或查看  与全域变数一样，实例变数在初始化前，也有 nil 值。 |
| [a-z] 或 \_ | 区域变数 (local variable) | 以小写字母或底线字元 (\_) 开头  初始化前并没有 nil 值 |
| [A-Z] | 常数 |  |

Ruby 中的两个拟变数 (pseudo-variable)：

|  |
| --- |
| self、永远指示目前正在执行的物件；self 是由直译器维护的全域变数， |
| nil、这是指派予未初始化 (uninitialized) 变数的无意义值。nil 是一个常数 |
| 你不能为 self 或 nil 指派任何值 |

## 实例属性

|  |  |
| --- | --- |
| **第一种** | **class** Song  **def name**  @name  **end**  **def age**  @age  **end**  **def sex**  @sex  **end**  **def initialize**(*name*, *age*, *sex*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  @sex **=** sex  **end**  **end**  song **= Song**.**new**("abc", 19, "fail")  puts song.name |
| **第二种** | **class** Song  attr\_reader **:name**, **:age**, **:sex**  **def initialize**(*name*, *age*, *sex*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  @sex **=** sex  **end**  **end**  song **= Song**.**new**("abc", 19, "fail")  puts song.name |

### 给属性赋值的方法

|  |  |
| --- | --- |
| **第一种** | **class** Song  attr\_reader **:name**, **:age**, **:sex**  **def initialize**(*name*, *age*, *sex*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  @sex **=** sex  **end**  **def name=**(*new\_name*)  @name **=** new\_name  **end**  **end**  song **= Song**.**new**("abc", 19, "fail")  puts song.name  song.name**=**"jinwan"  puts song.name |
| **第二种** | **class** Song  attr\_reader **:name**, **:age**, **:sex #只读**  attr\_writer **:name**, **:age**, **:sex #只写**  **#或者 attr\_accessor :name, :age, :sex #可读写**    **def initialize**(*name*, *age*, *sex*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  @sex **=** sex  **end**  **end**  song **= Song**.**new**("abc", 19, "fail")  puts song.name  song.name**=**"jinwan"  puts song.name |

### 虚拟属性

### 类变量

类变量被类的所有对象共享, 对一个给定的类来说, 类变量只存在一份拷贝.

类变量用@@开头

与全局变量和实例变量不同, 类变量在使用前必须被初始化.

类变量对类及其实例都是私有的, 如果你想让他们能够被外界反问道, 你需要编写访问方法(实例方法 或 类方法).

### 类方法

|  |
| --- |
| **class** Song  *#自定义方法*  **def speak**  puts "Bow Wow"  **end**  **def Song**.**say**  puts "this is Class method"  **end**  **end** |

### Singleton模式

以下单例并非是线程安全的; 可以用Ruby提供的Singleton mixin实现线程安全.

|  |
| --- |
| **class** MyLogger  *#将MyLogger类的new方法标记为私有, 阻止了传统的构造方法创建对象. 提供了一个类方法, 这个方法用了一个类变量来保存唯一的实例.*  private\_class\_method **:new**    @@logger **= nil**    **def MyLogger**.**create**  @@logger **= new unless** @@logger  @@logger  **end**    **end** |

## 方法可见性：Public、Protected、Private

|  |
| --- |
| **class** Test  **def times\_two**(*a*)  puts "#{a} times two is #{engine(a)}"  **end**  **def engine**(*b*)  b**\***2  **end**  private:engine *# 让使用者看不到 engine*  **end** |

## 子类化和继承

|  |
| --- |
| **继承**  **class** Mammal  **def breathe**  puts "inhale and exhale"  **end**  **end**  **class** Cat **<** Mammal  **def speak**  puts "Meow"  **end**  **end**  hauhua **= Cat**.**new**  hauhua.breathe  hauhua.speak |
| **方法覆盖**  **class** Bird  **def preen**  puts "I am cleaning my feathers."  **end**  **def fly**  puts "I am flying."  **end**  **end**  **class** Penguin **<** Bird  **def fly**  fail "Sorry.I'd rather swim."  **end**  **end** |
| **方法增强1: 单纯调用父类方法** #增强父类别的 identify 方法，而不是完全替换掉, super 能够把引数传至原本的方法中  **class** Human  **def identify**  puts "I'm a person."  **end**  **def train\_toll**(*age*)  **if** age **<** 12  puts "Reduced fare.";  **else**  puts "Normal fare.";  **end**  **end**  **end**  **class** Student2**<**Human  **def identify**  **super**  puts "I'm a student too."  **end**  **end** |
| **方法增强2: 传参数给父类方法**  **class** Human  **def identify**  puts "I'm a person."  **end**  **def train\_toll**(*age*)  **if** age **<** 12  puts "Reduced fare.";  **else**  puts "Normal fare.";  **end**  **end**  **end**  **class** Honest**<**Human  **def train\_toll**(*age*)  **super**(age) *# 传进我们提供的参数*  **end**  **end** |

## 对象创建和初始化

|  |
| --- |
| **class** Foo  **def fourth\_power\_of**(*x*)  square(x) **\*** square(x)  **end**  **end**  **Foo**.**new**.fourth\_power\_of 10 |

# 类继承

|  |
| --- |
| **class** Song  *#初始化方法, 在用Song.new("youxuan", 19)时调用*  **def initialize**(*name*, *age*)  @name **=** name *#把参数name赋值给实例变量@name*  @age **=** age  **end**    *#重写to\_s方法*  **def to\_s**  "name:#@name, age:#@age"  **end**  *#自定义方法*  **def speak**  puts "Bow Wow"  **end**  **end** |
| **class** Kaka **<** Song    **def initialize**(*name*, *age*, *sex*)  **super**(name, age)  @sex **=** sex  **end**    *#重写to\_s方法*  **def to\_s**  **super +** " ,sex:#@sex"  **end**    **end** |

方法继承

方法重写

## 方法的访问控制

|  |  |
| --- | --- |
| public | 方法默认都是public的, 除了initialize方法是private的 |
| protected | 本类或子类 |
| private | 本类 |

## 重写to\_s方法和inspect方法

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

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

# 方法、Proc、Lambda和闭包

## 方法

在类外面定义方法, 这个方法属于object类

## 方法名

### !结尾

|  |
| --- |
| ruby> **s1 = "forth"**    "forth"  ruby> **s1.chop!       # 这改变了 s1。**    "fort"  ruby> **s2 = s1.chop   # 这将变更后的版本置于 s2，**    "for"  ruby> **s1             # ⋯⋯而不影响 s1。**    "fort" |
| 加入惊叹号就是“破坏性的”(destructive) chop， 那两者有甚么差别呢？在 Ruby 中，我们一般在方法名称后加上 '!' 或 '?'。惊叹号（! 有时读作 "bang!"）代表具有破坏性，即会改变所接触物件的值。chop! 会直接影响字串，但 chop 则会提供一个删减后的版本，而不影响原本的物件。以下将阐释两者的差异。 |

### ?结尾

|  |
| --- |
| 问号（? 有时读作 "huh?"），表示只会传回 true 或 false。（编注：! 和 ? 结尾的命名方式，只是一个惯例。!暗示有某种副作用，?暗示回传值是 Boolean 布尔值） |

## 方法和圆括号

## 方法参数

|  |  |
| --- | --- |
| object.method(arg1, arg2) | object.method arg1, arg2  如果并不会造成歧义的话，可以省略括号。 |

**类似this的东西**

self.method\_name(args...)

## Proc和Lambda

|  |
| --- |
| 使用 Proc.new 创造一个新的*程序物件 (procedure object)* （编注：原英文版使用 proc 方法，现在的惯例改成使用 Proc.new 或 lambda)：  quux = Proc.new {  puts "QUUXQUUXQUUX!!!"  } |
| quux.call  QUUXQUUXQUUX!!! |

## Method对象

# 模块block

block只是在花括号或者do...end之间的一组代码

一般: 单行block用花括号, 多行block用do...end

**使用自定义模块**

|  |  |
| --- | --- |
| load "**mod.rb**"  include Util  getDate |  |
| **mod.rb文件**  **module** Util  **def getDate**  puts "--=hehe";  **end**  **end** |  |

**使用系统内置模块**

|  |
| --- |
| include Math  puts sqrt(2)  puts PI |

## 引用

## mix in

## 加载和请求模块

## 单键方法和Eigenclass

## 方法查找

## 常量查找

# Ruby API

## 字符串

## 正则表达式

## 数字和数学运算

数字格式化

http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html

## 日期和时间

|  |
| --- |
| @time = Time.now |

### 日期格式化输出

|  |  |
| --- | --- |
| 格式 | 语法 |
| 2013-11-28 10:10:36 | created\_at.to\_s(:db) |
| rails 格林威治时间，有时少8小时 | 在application.rb配置中添加  config.time\_zone = 'Beijing' |
| 2013-11-28 |  |
| 2013年12月19日 03:01 | <%= l payment.created\_at, format: :long %> |

### 字符串转日期

## 集合

## 文件和目录

## 输入／输出

### print和puts的区别

|  |  |  |
| --- | --- | --- |
| 输出 | puts | puts 则一定会以输出行 (output line) 结尾, 也就算会输出换行符。 |
| print | print 会准确输出所提供的内容；  使用 print，会让游标停留在显示内容的后面，而不是下一行开头。这可用作提示使用者输入。  print "guess?"; STDOUT.flush  文字视窗有时会为了速度而缓冲输出内容，只有遇到换行字元时，才会收集个别字元并加以显示。  输出提示后，你可以 flush 输出内容，避免上述情况发生。 |
| printf | 输出格式化字符串  print("number: %5.2f, \nstring: %s\n", 1.23, "hello")  结果:  number: 1.23  String: hello |
| 输入 | gets | 从程序的标准输入流中读取下一行  例如:  print line **while**(line **=** gets) #输入什么, 打印什么 |

## 网络

## 线程和并发

# ----------未分类----------------

## yield关键字

|  |
| --- |
| **def call\_block**  puts "Start of method"  **yield**  **yield**  puts "End of method"  **end**  call\_block { puts "In the block" }  输出:  Start of method  In the block  In the block  End of method |
| **def call\_block**  puts "Start of method"  **yield**("hello", 99)  puts "Start of method"  **end**  call\_block{**|**str, num**|** puts str; puts num} |

## 计算阶乘的方法

|  |
| --- |
| def fact(n)    if n == 0      1    else      n \* fact(n-1)    end  end |

## 变量和方法的引用方式

|  |  |
| --- | --- |
| Fred#do\_something | 实例方法 |
| Fred.new  或  Fred::new | 类方法 |
| Fred::EOF | 类常量 |

## 字符串和数字常用API

|  |  |  |
| --- | --- | --- |
| 示例代码 | 结果 | 解释 |
| "jinwanlin".length | 9 | 求字符串长度 |
| "jinwanlin".index("in"); | 1 | 查找子字符串在字符串中首次出现的位置 |
| -1024.abs | 1024 | 求绝对值 |

## 注释和语句分割

|  |
| --- |
| 一个语句单独占一行, 就无需用分号结尾  注释以#开头 |

## 调用方法传参数, 可省略括号

|  |
| --- |
| def say\_goodnight(name)  result = "Good night, #{name}"  return result  end  puts say\_goodnight("pa")  puts(say\_goodnight("pa")) |

## 全局变量/实例变量/局部变量 被引用是否加{}

|  |  |  |
| --- | --- | --- |
| 全局变量 | $name = "jinwanlin"  puts "age: #@name" | 引用全局变量和实例变量无需花括号括起来 |
| 实例变量 | @age="24"  puts "age: #@age" |
| 局部变量 | def say\_goodnight(name)  result = "Good night, #{name}"  return result  end | 引用局部变量需花括号括起来 |

## 命名规范

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **说明** |
| 局部变量/方法参数/方法名称 | 小写字母或下划线开始 | 词与词之间用下划线分隔 |  |
| 全局变量 | $开始 |  |  |
| 实例变量 | @开始 | 词与词之间用下划线分隔 | 写在方法内部, 通常创建对象时初始化 |
| 类变量 | @@开始 | 驼峰式分隔单词 |  |
| 类名/模块名/常量 | 大写字母开始 | 驼峰式分隔单词 |  |
| 方法名称 | 可以 ?、！、= 结束 |  |  |
| 字母数字下划线组合, 但@后面不能是数字 | | |  |

## 方法返回值

|  |  |
| --- | --- |
| def say\_goodnight(name)  result = "Good night, #{name}"  return result #可去掉  end | def say\_goodnight(name)  "Good night, #{name}"  end  #ruby方法所返回的值, 是最后一个呗求值的表达式的值. |

## 集合(数组和哈希)

# 测试

## 断言

assert\_equal(s1, list[0])

assert\_nil(s1)

## 读取文件

|  |
| --- |
| **def first\_line**( *filename* )  **begin**  file **=** open(filename)    file.each\_byte{**|**str**|**  print str  pu  }  file.close  info *# 最后计算的内容为传回值*  **rescue**  *#nil # 无法读取档案？那就不传回字串*  puts "22222"  **end**  **end** |
| first\_line("c:\\aaa.txt") |