Ruby 语言介绍

目录

1. 基本的 ruby 语法

1.1 变量、常量和类型

1) 定义变量

变量类型	描述	示例
局部变量（或伪变量）	以小写字母或下划线卡头	var _var
全局变量	以$开头	$var
类变量	类中定义，以@@开头	@@var
实例变量	对象中定义，以@开头	@var
常量	以大写字母开头	Var

2) 变量内插

在双引号内使用“#{变量名}”内插变量

```
a = 2
b = 3
puts "#{a} + #{b} = #{a+b}"   #输入结果为：2 + 3 = 5
```

1.2 注释

1） 单行注释：以#开头，如： #注释内容

2） 多行注释：在=begin 和 =end 之间定义，如：

```
=begin
  注释内容
=end
```

1.3 循环和分支

1.3.1 条件语句

If 形式	unless 形式
a =1 if y==3	a=1 unless y!=3
x= if a > 0 then b else c end	x= unless a<=0 then a else b end

if x<5 then
  a =1
else
  a =2
	unless x<5 then
	  a =2
	else
	  a =1

end	end

1.3.2 循环结构

#while 循环
i= 0
while i< list.size do
  print “#list[i] ”
  I += 1
end
	#until 循环
	i= 0
	until i == list.size do
	  print “#list[i]”
	  i += 1
	end

#for 循环
for x in lisy do
  print “#{x}”
end
	#each 循环
	list.each do |x|
	  print “#{x}”
	end

#loop 循环
i = 0
n = list.size-1
loop do
  print “#{list[i]}”
  i += 1
  break id i > n
end
	#times 循环
	n = list.size
	n.times do |i|
	  print “#{list[i]}”
	end

#upto 循环
n =list.size–1
0.upto(n) do |i|
  print “#{list[i]}”
end
	#each_index 循环
	list.each_index do |x|
	  print “#{list[x]}”
	end

1.3.3 异常

```
begin
    x = Math.sqrt(y/z)
rescue ArgumentError       #匹配错误类型
    puts "Error taking square root"
rescue ZeroDivisionError     #匹配错误类型
    puts "Attempted division by zero"
else
    puts "Other Error"
ensure
    #这部分代码在最后一定会执行
end
```

1.3.4 类

```ruby
class Hello  #类名必须以大写字母开头
  @@myname = "John"   #类变量

  def initialize(name,phone)
      @name,@phone = name,phone  #定义实例变量
  end

  def hello  #定义类方法
    puts "#{@name}的电话是#{@phone}"
  end
end

f = Hello.new("Tom","13100000000")  #创建类对象
f.hello  #调用类方法，结果为：Tom 的电话是 13100000000
```

1.3.5 模块

方法	说明
include ‘watir’	将模块的功能添加到当前空间中，不加载已加载的文件
extend ‘watir’	将模块的函数添加到对象中
load ‘watir’	读取文件，并将其插入到源文件的当前位置，以便从这个位置开始就可以使用其定义

1.3.6 case 语句

```ruby
case x
  when 1..10             #匹配数字
    puts "First branch"
  when foobar()          #批量方法返回的值
    puts "Second branch"
  when /^hel.*/          #匹配正则表达式
    puts "Third branch"
  else
    puts "Last branch"
end
```

1.4 正则表达式

1.4.1 普通字符

普通字符由字母、数字、下划线和特殊符号组成。

如：

表达式/b/在匹配字符“abcd”时，匹配的内容是“b”

表达式/b_/在匹配字符“ab_cd”时，匹配的内容是“b_”

1.4.2 转义符

	表达式	可匹配内容
特殊字符	\r 或 \n	代表回车和换行符
	\t	制表符
	\\	代表“\”
	\xXX	代表标号在 0~255 范围的字符
	\uXXXX	任何字符可以使用“\u”加上其编号的 4 位十六进制数表示
特殊语义	^	匹配输入字符串的开始位置
	$	匹配输入字符串的结尾位置
	( )	标记一个子表达式的开始和结束位置
	[ ]	用来自定义能给匹配“多种字符”的表达式
	{ }	修饰匹配次数的符号
	.	匹配除了换行符(\n)外的任意一个字符
	?	修饰匹配次数为 0 次或 1 次
	+	修饰匹配次数至少为 1 次
	*	修饰匹配次数至少为 0 次或任意次

1.4.3 匹配多种字符

	表达式	可匹配内容
多字符匹配	\d	匹配任意一个数字，即：0~9
	\w	匹配任意一个字母、数字或下划线，即：A~Z、a~z、0~9、_
	\s	匹配任意一个空格、制表符、换页符等空白字符
	\S	匹配所有非空字符
	\D	匹配所有非数字字符
	\W	匹配所有字母、数字或下划线以外的字符
	\B	匹配非单词边界

1.4.4 修饰匹配次数的方法

	表达式	可匹配内容
匹配次数	{n}	表达式重复 n 次，如：/\w{2}/相当于/\w\w/

修饰符	**{m,n}**	表达式至少重复 m 次，最多重复 n 次
	{m,}	表达式至少重复 m 次
	?	修饰匹配次数为 0 次或 1 次
	+	修饰匹配次数至少为 1 次
	*****	修饰匹配次数至少为 0 次或任意次

1.4.5 匹配模式

	表达式	可匹配内容
ruby 支持	**Ignorecase**	该模式下正则表达式不区分大小写
	Multiline	该模式下正则表达式可以匹配多行
ruby 不支持	**Singleline**	该模式下小数点可以匹配包括换行符在内的所有字符
	Global	主要在替换表达式时起作用 配置为 **Global** 表示替换所有的匹配

2. 常用函数

2.1 Numeric 类

函数名称	说明	示例
chr	返回数字的 ACSII 码	65.chr >> “A”
downto	接收一个 block,从大到小循环执行	5.downto(2) { |i| puts i} >> 5 4 3 2
upto	接收一个 block,从小到大循环执行	2.upto(5) { |i| puts i} >> 2 3 4 5
next 或 succ	返回下一个数	1.next >> 2 1.succ >>2
step	以固定步长循环执行	1.step(10,2) { |i| puts i} >> 1 2 3 5 7
times	循环执行 n 次	5.times { |i| puts i} >> 1 2 3 4 5

2.2 Float 类

函数名称	说明	示例
ceil	返回比 float 大的最小整数	(2.98).ceil >> 3
floor	返回比 float 小的最大整数	(2.98). floor >> 2
round	四舍五入到一个整数	(2.98). round >> 3
to_i	返回 float 截掉小数点后的整数	(2.98).to_i >> 2

2.3 String 类

函数名称	说明	示例
*	将字符串拷贝 N 次	“ha”*4 >> “hahahaha”
+ << concat	连接字符串	“yes” + “no” >> “yesno” "yes" << "no" >> “yesno” "yes".concat("no")
<=>	比较字符串，返回值如下： 大于：-1 等于：0 小于：1	"Ab" <=> "ab" >> -1 "ab" <=> "ab" >> 0 "ab" <=> "Ab" >> 1
== 或 ===	判断两个对象是否相等	"1" == "1" >>true "1" == 1 >>flase

=~	匹配正则表达式	**"abc123abc" =~ /\d/ >> 3**
[ ] 或 slice	返回字符串的某一范围内的值	**“abc”[0,2] >> “ab”** **"hello"[/llo/] >> “llo”** **“abc” slice [0,2] >> “ab”**
[ ]=	替换字符串的某一范围内的值	**a = “hello word”** **a[1,2]= “OO”** **puts a >>”** `hOOlo word”`
capitalize **capitalize!**	把字符串的首字母大写，其他字母小写	**"hi,Ruby".capitalize** **>>** `“Hi,ruby”`
chomp **chomp!**	删除字符串后的空白字符	**"string\r\n".chomp** **>> ” string”**
chop	删除最后一个字符	**"string".chop >> “strin”**
count	返回该字符串中含的字符个数	**a = "hello world"** **a.count "lo" >> 5** **(l 出现 3 次，o 出现 2 次)**
delete **delete!**	删除字符	**"hello".delete "l","lo" » "heo"** **"hello".delete "lo" » "he"**
downcase **downcase！**	将大写字母改写为小写	**"hEllO".downcase » "hello"**
upcase **upcase！**	将小写字母改写为大写	**"hEllO".upcase » "HELLO"**
swapcase **swapcase!**	将所有的大写字母改为小写字母， 小写字母改为大写字母。	**"Hello".swapcase » "hELLO"**
each	对字符串中的各行进行迭代操作	**"Hi\nRuby". each { |s| puts s}**
each_byte	对字符串中的各个字节进行迭代操作	**"Hi\nRuby". each_byte { |s| puts s}**
each_line	对字符串中的每一行进行迭代操作	**"Hi\nRuby". each_line { |s| puts s}**
empty?	判断字符串是否为空	**"hello".empty? » false** **"".empty? » true**
gsub **gsub！**	以 replace 来替换字符串中所有与 pattern 相匹配的部分	**"hello".gsub(/[aeiou]/, '*') » "h*ll*"**
hash	返回字符串的哈希值	**"h".hash >> 107**
include?	若字符串中包含 *substr* 子字符串的话，就返回真	**"hello".include? "lo" » true** **"hello".include? "ol" » false**
index	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**
length	返回字符串的字节数	**"hello".length >> 5**
replace	替换字符串的内容	**s = "hello" » "hello"** **s.replace "world" » "world"**
sub 或 sub!	用 *replace* 来替换 *首次*匹配 *pattern* 的部分。	**"hello".sub(/[aeiou]/, '*')** **» "h*llo"**
reverse **reverse!**	对字符串进行反转	**"stressed".reverse » "desserts"**

scan	使用正则表达式 *re* 反复对 self 进行匹配操作,并以数组的形式返回匹配成功的子字符串	**a = "cruel world"** **a.scan(/\w+/) » ["cruel", "world"]** **a.scan(/.../) » ["cru", "el ", "wor"]**
split	使用 *sep* 指定的 pattern 来分割字符串,并将分割结果存入数组	**"mellow yellow".split("ello")** **» ["m", "w y", "w"]**
squeeze **squeeze!**	压缩由 *str* 所含字符构成的重复字符串	**"yellow moon".squeeze** **» "yelow mon"** **" now is the".squeeze(" ")** **» " now is the"**
strip **strip!**	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"。	**" hello ".strip » "hello"** **"\tgoodbye\r\n".strip » "goodbye"**
tr 或 tr!	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	**hello".tr('aeiou', '*') » "h*ll*"** **"hello".tr('^aeiou', '*') » "*e**o"**
tr_s 或 tr_s!	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。同时，若替换部分中出现重复字符串时，就将其压缩为 1 个字符	**"hello".tr_s('l', 'r') » "hero"** **"hello".tr_s('el', '*') » "h*o"** **"hello".tr_s('el', 'hx') » "hhxo"**
upto	在从 self 到 *max* 的范围内，依次取出下一个字符串"后将其传给块，进行迭代操作	**"a1".upto("a3") {|s| puts s}** **» a1\na2\na3**
to_f	将字符串转为浮点数	**"45.67 degrees".to_f » 45.67**
to_i	将字符串转为整数	**"99 red balloons".to_i » 99**
to_s	将字符串转为字符串	

2.4 Array 类

函数名称	说明	示例
&	数组与，返回两数组的交集	**[1,2] & [2,3] » [2]**
*	复制数组 n 次	**[1,2]*2 » [1,2,1,2]**
+	返回两数组的并集，但不排除重复元素	**[1,2]+[2,3] » [1,2,2,3]**
<<	追加元素，但不排除重复元素	**[1,2]<<[2,3] » [1,2,2,3]**
|	追加元素，但排除重复元素	**1,2] | [2,3] » [1,2,3]**
-	返回第一个数组与第二个数组不同的元素	**[1,2]-[2,3] » [1]**
<=>	比较数组	**[1,2]<=>[2,3] »flase**
==	比较数组，若所有元素均相等时返回真	**[1,2]==[2,1] »flase**
assoc	从数组的每个元素中寻找指定对象	**[[1,2],[3,4]].assoc(2) » [1,2]**
at	找到数组的第 N 个元素 负数表示逆向查找	**["a","b","c","d","e"].at(0) » "a"** **["a","b","c","d","e"].at(-1) » "e"**
clear	删除数组中的所有元素	**["a","b","c","d","e"]. clear**

collect **collect!**	用一个过程块对数组的每个元素进行处理	**["a","b","c","d"].collect {|x| x + "!" }** **» ["a!", "b!", "c!", "d!"]**
compact **compact!**	删除值为 nil 的元素后生成新数组并返回它	**["a",nil,"b",nil,"c",nil].compact** **» ["a", "b", "c"]**
delete	删除元素，如果元素重复，全部删除	**a = ["a", "b", "b", "b", "c"]** **a.delete("b")** **puts a » ["a","c"]**
delete_at	删除 *pos* 所指位置的元素并返回它。若 *pos* 超出数组范围则返回 nil	**a = %w(ant bat cat dog)** **a.delete_at(2) » "cat"** **a » ["ant", "bat", "dog"]** **a.delete_at(99) » nil**
delete_if	根据条件删除	**a = ["a", "b", "c"]** **a.delete_if {|x| x >= "b" } » ["a"]**
each	对数组的每个元素按值进行迭代操作	**a = ["a", "b", "c"]** **a.each {|x| print x, " -- " }** **»** **"a -- b -- c --"**
each_index	对数组的每个元素按索引进行迭代操作	**a = ["a", "b", "c"]** **a.each_index {|x| print x, " -- " }** **» "0 -- 1 -- 2 --"**
empty?	判断数组是否为空，为空则返回真	**[].empty? » true**
eql!	比较两数组是否相等	**["a","b","c"].eql?(["a","b","c"]) » true**
fill	填充数组	**["a","b","c","d"].fill("x")** **» ["x","x","x","x"]** **["a","b","c","d"].fill("z", 2, 2)** **» ["x", "x", "z", "z"]**
first	返回数组的首元素。若没有首元素则返回 nil	**["q", "r", "s", "t"].first » "q"**
last	返回数组末尾的元素。若数组为空时，返回 nil	**["w","x","y","z"].last » "z"**
include?	判断数组中是否包含元素	**a = ["a", "b", "c"]** **a.include?("b") » true** **a.include?("z") » false**
index	返回数组中第一个==*val* 的元素的位置	**a = ["a", "b", "c"]** **a.index("b") » 1** **a.index("z") » nil**
indexes	以数组形式返回其索引值与各参数值相等的元素	**a = ["a", "b", "c", "d", "e", "f", "g"]** **a.indexes(0, 2, 4) » ["a", "c", "e"]** **a.indexes(2, 4, 12) » ["c", "e", nil]**
insert	在索引为 *nth* 的元素前面插入第 2 参数以后的值	**ary = %w(foo bar baz)** **ary.insert 2,'a','b'** **p ary » ["foo", "bar", "a", "b", "baz"]**
join	将数组元素按一定的分隔符连接起来	**["a", "b", "c"].join » "abc"** **["a", "b", "c"].join("-") » "a-b-c"**

length **size**	返回数组长度。若数组为空则返回 0	**[1,2,3].length » 3** **[1,2,3].size » 3**
nitems	返回非 nil 元素的个数	**[1, nil, 3, nil, 5].nitems » 3**
pop	删除末尾元素并返回它。若数组为空则返回 nil	**a = ["a", "m", "z"]** **a.pop » "z"** **p a » ["a", "m"]**
push	添加新元素	**["a","b"].push(['1','2'])** **»** `["a", "b", ["1", "2"]]`
rassoc	遍历数组每个元素（元素必须是数组），匹配索引为 1 的值是否与查找的字符相等，返回第一个相等的元素	**a = [[15,1], [25,2], [35,2]]** **p a.rassoc(2) » [25, 2]**
replace	替换数组元素	**a = ["a","b"]** **a.replace(["x","y","z"])** **p a »["x", "y", "z"]**
reverse **reverse!**	将所有元素以逆序重新排列生成新数组并返回它	**["a","b","c"].reverse » ["c", "b", "a"]**
rindex	返回最后一个值相等的元素的索引值	**a = ["a","b","b","b","c"]** **a.rindex("b") » 3**
shift	删除数组的首元素并返回它。剩余元素依次提前。若数组为空返回 nil。	**args = ["-m","-q","filename"]** **args.shift » "-m"** **args » ["-q", "filename"]**
sort **sort!**	从小到大排序	**a = ["d", "a", "e", "c", "b"]** **a.sort » ["a", "b", "c", "d", "e"]**
uniq **uniq!**	删除数组中的重复元素后生成新数组并返回它	**a = ["a", "a", "b", "b", "c"]** **a.uniq » ["a", "b", "c"]**
unshift	在数组第一个元素前添加元素	**a = ["b", "c", "d"]** **a.unshift("a") » ["a", "b", "c", "d"]**
to_s	将数组的所有元素连接成字符串	**["a","e","i","o"].to_s » "aeio"**

2.5 Hash 类

函数名称	说明	示例
==	判断两个 Hash 是否相等	**h1 = {"a" => 1, "c" => 2}** **h2 = { "a" => 1, "c" => 2, 7 => 35 }** **h1 == h2 » false**
[ ]	返回指定键值对应的对象	**h = { "a" => 100, "b" => 200 }** **h["a"] » 100** **h["c"] » nil**
[ ]=	向 Hash 添加记录	**h = { "a" => 100, "b" => 200 }** **h["a"] = 9** **h["c"] = 4** **h » {"a"=>9, "b"=>200, "c"=>4}**

clear	清空哈希表的内容. 返回 self.	**h = { "a" => 100, "b" => 200 }** **h.clear » {}**
default	返回哈希表的默认值	**h = { "a" => 100, "b" => 200 }** **h.default = "Go fish"** **h["a"] » 100** **h["z"] » "Go fish"**
delete	从词典中删除和键值相符的记录	**h = { "a" => 100, "b" => 200 }** **h.delete("a") » 100** **h.delete("z") » nil** **h.delete("z") { |el| "#{el} not found" }** **» "z not found"**
delete_if	通过过程块来删除特定键值的记录	**h = { "a" => 100, "b" => 200 }** **h.delete_if {|key, value| key >= "b" }** **» {"a"=>100}**
each	Hash 表的迭代操作，对表的每一个词对进行迭代操作	**h = { "a" => 100}** **h.each {|key, value| print key, " is ",value,"\n"} » a is 100**
each_key	对表的每一个键对进行迭代操作	**h = { "a" => 100, "b" => 200 }** **h.each_key {|key| print key } » ab**
each_value	针对 value 进行迭代操作	**h = { "a" => 100, "b" => 200 }** **h.each_value {|value| print value }** **» 100200**
empty?	判断哈希表是否为空，空则返回 true	**{}.empty? » true**
fetch	如果能找到键值为 key 的 hash 值，则返回 Hash 值； 如果找不到，则返回默认值或指定值； 如果默认值和指定值都找不到，抛异常	**h = { "a" => 100, "b" => 200 }** **h.fetch("a") » 100** **h.fetch("z", "go fish") » "go fish"** **h.fetch("k")** **»in `fetch': key not found (IndexError)**
has_key?	判断是否存在相符的 key 值	**h = { "a" => 100, "b" => 200 }** **h.has_key?("a") » true** **h.has_key?("z") » false**
key?		**h. key?("z") » false**
include?		**h.include?("b") » true**
has_value?	判断是否存在相符的 value 值	**h = { "a" => 100, "b" => 200 }** **h.has_value?(100) » true** **h.has_value?(999) » false**
value?		**h.value?(100) » true** **h.value?(999) » false**
index	返回给定值的键值，未找到返 nil	**h = { "a" => 100, "b" => 200 }** **h.index(200) » "b"** **h.index(999) » nil**
indexes	返回一系列给定的键值对应值组成的数组	**h = { "a" =>100, "b" =>200, "c" =>300 }** **h.indexes("a", "c") » [100,300]** **h.indexes("a","z") » [100,nil]**

indices		**h.indexes("a", "c") » [100,300]** **h.indexes("a", "z") » [100,nil]**
invert	将元素值和索引互换,返回变换后的哈希表. *注意:若原哈希表中若干不同的索引对应相同的元素值时,其变换结果将无法预测.*	**h = { "n" => 100,"m" => 100,"y" => 300}** **h.invert » {300=>"y",100=>"n"}**
keys	返回一个包含所有 key 的数组.	**h = { "a" => 100, "b" => 200 }** **h.keys » ["a", "b"]**
values	返回一个包含所有 vakue 的数组.	**h = { "a" => 100,"b" => 200,"c" => 300 }** **h.values » [100, 200, 300]**
length **size**	返回词典中元素的个数	**h = { "d" =>100, "a"=> 200, "v" => 300}** **h.length » 3** **h. size » 3**
rehash	重新计算索引对应的哈希表值。 当与索引对应的哈希表值发生变化时，若不使用该方法来重新计算的话，将无法取出与索引对应的哈希表值。	**a = ["a", "b"]** **c = ["c", "d"]** **h = { a => 100, c => 300 }** **a[0] = "z"** **h.rehash** **» {["z", "b"]=>100, ["c", "d"]=>300}** **h[a] » 100**
replace	以另外一张 Hash 表的内容来替换当前 Hash 表的内容	**h = {"a" =>100, "b" =>200 }** **h.replace({ "c" => 300, "d" => 400 })** **» {"c"=>300, "d"=>400}**
shift	删除一个哈希表元素后 再以[key,value]数组的形式将其返回	**h = { 1 => "a", 2 => "b", 3 => "c" }** **h.shift » [1, "a"]** **h » {2=>"b", 3=>"c"}**
sort	对 Hash 进行排序 按键值从小到大排序	**h = { "b" => 30, "a" => 20, "c" => 10 }** **h.sort » [["a", 20], ["b", 30], ["c", 10]]** **h.sort {|a,b| a[1]<=>b[1]}** **» [["c", 10], ["a", 20], ["b", 30]]**
to_a	把 Hash 表转换为数组 数组按 Hash 表的键值从小到大排序	**h = {"c" =>300,"a" =>100,"d" =>400}** **» [["a", 100], ["c", 300], ["d", 400]]**
to_s	把 Hash 表转换为字符串	**h = { "c" => 300, "a" => 100, "d" => 400}** **h.to_s » "a100c300d400"**
update	用一张 Hash 表去更新另外张 Hash 表	**h1 = { "a" => 100, "b" => 200 }** **h2 = { "b" => 254, "c" => 300 }** **h1.update(h2)** **» {"a"=>100, "b"=>254, "c"=>300}**