第9章 运算符

作者 bluetea
网站:https://github.com/bluetea

一.|| 运算符
val1 || val2
只有条件1不是false或者nil的时候，才会判断条件2
返回值：逻辑表达式的返回值为最后一个表达式的值
1.如果val1为真的时候，就不会判断 val2了，所以实际val1为最后一个表达式，所以这个时候的表达式返回值为val1

```
[9] pry(main)> "python" || "ruby"
=> "python"
```

2.如果val1 的值为false或者nil的时候，需要判断val2的值，无论 val2 为真还是 nil，都会返回val2的值

```
[10] pry(main)> nil || nil
=> nil
[11] pry(main)> nil || "ruby"
=> "ruby"
```

二，&&运算法
val1 && val2
只有当val1 为真的时候，才会判断val2呢,
返回值：这个为逻辑表达式，所以返回值为最后一个表达式的值
1. val1 为nil 所以不会判断 val2 ，这时候返回值为nil

```
[12] pry(main)> nil && "ruby"
=> nil
```

2.val1 为 “pyhthon” 为真，在判断 val2，val2为nil的话，val2为最后一个表达式，所以返回值为 nil

```
[15] pry(main)> "python" && nil
=> nil
```

3.val1 为“pythone”为真， val2为“ruby” 为真，所以val2为最后一个表达式，所以返回值为”ruby”

```
[14] pry(main)> "python" && "ruby"
=> "ruby"
```

总结：
1.判断val1 和val2 ，将为真的赋值给name
给一个变量值赋值，如果val1为真就赋值为name，如果为假就将val2赋值给name

```
[10] pry(main)> val1 = 22
=> 22
[11] pry(main)> val2 = 44
=> 44
[12] pry(main)> name = val1 || val2
=> 22
```

```
[10] pry(main)> val1 = 22
=> 22
[11] pry(main)> val2 = 44
=> 44
[12] pry(main)> name = val1 || val2
=> 22
[13] pry(main)> val1 = nil
=> nil
[14] pry(main)> name = val1 || val2
=> 44
[15] pry(main)>
```

2.判断val1和val2，只有都为真的时候，才执行赋值动作
修改一下，这个程序可以改为

```
item = nil
if array #如果array不是nil或者false的时候
  item = array[0] #将array[0]赋值给item
end
p item
```

改为

```
[6] pry(main)> name = array && array[0]
NameError: undefined local variable or method `array' for main:Object
from (pry):5:in `__pry__'
[7] pry(main)> array = [22,33,44]
=> [22, 33, 44]
[8] pry(main)> name = array && array[0]
=> 22
```

所以一行可以判断并赋值，注意不要写成
name = array[0] && array

所以 || 是编程中赋值默认的一个好的方法
name = val || 1,如果val为nil 或者false的时候，就赋值1，这是赋值默认值的好方法

9.3 条件运算符——三元运算符
条件？表达式1：表达式2

```
[4] pry(main)> nil ? 22 : 33
=> 33
[5] pry(main)> true ? 22 : 33
=> 22
```

9.4范围运算符
Range类返回范围

```
[1] pry(main)> Range.new(1,9)
=> 1..9
[2] pry(main)> a = Range.new(1,9)
=> 1..9
[3] pry(main)> a.each {|x| puts x}
1
2
3
4
5
6
7
8
9
=> 1..9
```

或者另外一种Range函数的表达形式

```
[4] pry(main)> a = 1..9
=> 1..9
```

同一个效果
字符串也可以的，并由to_a 转换成array函数

```
[8] pry(main)> (1..9).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9]
[9] pry(main)> ("a".."f")
=> "a".."f"
[10] pry(main)> ("a".."f").to_a
=> ["a", "b", "c", "d", "e", "f"]
[11] pry(main)> ('a'..'f').to_a
=> ["a", "b", "c", "d", "e", "f"]
```

其实Range 对象的内容，是使用succ函数根据当前值生成下一个值的，
，具体来说就是，对succ方法的返回值调用succ方法，然后对该返回值再调用succ方法，然后对返回值再调用succ方法，一直到值

```
1  #自己写的一个生成range值的方法，关键的方法就是succ方法
2  def range(b, e)
3    a = Array.new
4    b1 = b
5    e1 = e+1
6    i = 0
7    while (b1 != e1)
8      a[i] = b1
9      b1 = b1.succ
10     i += 1
11   end
12   return a
13 end
14
15 p range(3,9)
```

9.6 定义运算符
很多运算符可以重新定义，但是如下的是不能重新定义的
:: && || .. ... ?: not = and or
9.6.1 定义二元运算符
定义二元运算符时，会将运算符作为方法名， 运算符左边为接受者，右侧为方法的参数传递:
对比下 self.class和Point在方法内的区别，继承的时候，会区别很大，返回的对象的class就不对了

yunsuan.rb

```ruby
class Point
  attr_reader :x, :y

  def initialize(x=0, y=0)
    @x, @y = x, y
  end

  def inspect
    "#{x},  #{y}"
  end

  def +(other)
    #此时的返回值是self.class类型的对象，这个对象就是Point类型的
    Point.new(x + other.x, y + other.y) #所以只能用self.class确保继承的可用
  end

  def -(other)
    self.class.new(x - other.x, y - other.y)
  end
end

class Point1 < Point

end

p1 = Point1.new(11,3)
p2 = Point1.new(2,3)
p (p1 + p2).class
p (p1 - p2).class
```

1. bash

ruby | bash

```
wangmjcdeMacBook-Pro:ruby wangmjc$ ruby yunsuan.rb
Point
Point1
```

yunsuan.rb

```ruby
class Point
  attr_reader :x, :y

  def initialize(x=0, y=0)
    @x, @y = x, y
  end

  def inspect
    "#{x},  #{y}"
  end

  def +(other)
    #此时的返回值是self.class类型的对象，这个对象就是Point类型的
    self.class.new(x + other.x, y + other.y) #所以只能用self.class确保继承的可用
  end

  def -(other)
    self.class.new(x - other.x, y - other.y)
  end
end

class Point1 < Point

end

p1 = Point1.new(11,3)
p2 = Point1.new(2,3)
p p1 + p2
p p1 - p2
```

1. bash

ruby | bash

```
wangmjcdeMacBook-Pro:ruby wangmjc$ ruby yunsuan.rb
13,  6
9,  0
```

Line 1