在Scala中函数和方法有什么区别

方法可以作为一个表达式的一部分出现（调用函数并传参），但是方法（带参方法）不能作为最终的表达式，

但是函数可以作为最终的表达式出现：

scala> //定义一个方法

scala> def m(x:Int) = 2\*x

m: (x: Int)Int

scala> //定义一个函数

scala> val f = (x:Int) => 2\*x

f: Int => Int = <function1>

scala> //方法不能作为最终表达式出现

scala> m

<console>:9: error: missing arguments for method m;

follow this method with `\_‘ if you want to treat it as a partially applied function

m

^

scala> //函数可以作为最终表达式出现

scala> f

res9: Int => Int = <function1>

无参方法可以作为最终表达式出现，其实这属于方法调用，scala规定无参函数的调用可以省略括号

（关于方法调用我们下面会涉及到）

scala> def m1()=1+2

m1: ()Int

scala> m1

res10: Int = 3

参数列表对于方法是可选的，但是对于函数是强制的

方法的可以没有参数列表，参数列表也可以为空。但是函数必须有参数列表（也可以为空），见下面例子

scala> //方法可以没有参数列表

scala> def m2 = 100;

m2: Int

scala> //方法可以有一个空的参数列表

scala> def m3() = 100

m3: ()Int

scala> //函数必须有参数列表，否则报错

scala> var f1 = => 100

<console>:1: error: illegal start of simple expression

var f1 = => 100

^

scala> //函数也可以有一个空的参数列表

scala> var f2 = () => 100

f2: () => Int = <function0>

那么方法为什么可以没有参数列表呢，往下看。

方法名意味着方法调用，函数名只是代表函数自身

因为方法不能作为最终的表达式存在，所以如果你写了一个方法的名字并且该方法不带参数（没有参数列表或者无参)

该表达式的意思是：调用该方法得到最终的表达式。因为函数可以作为最终表达式出现，如果你写下函数的名字，函数

调用并不会发生，该方法自身将作为最终的表达式进行返回，如果要强制调用一个函数，你必须在函数名后面写()

scala> //该方法没有参数列表

scala> m2

res11: Int = 100

scala> //该方法有一个空的参数列表

scala> m3

res12: Int = 100

scala> //得到函数自身，不会发生函数调用

scala> f2

res13: () => Int = <function0>

scala> //调用函数

scala> f2()

res14: Int = 100

为什么在函数出现的地方我们可以提供一个方法

在scala中很多高级函数，如map(),filter()等，都是要求提供一个函数作为参数。但是为什么我们可以提供一个方法呢

？就像下面这样：

scala> val myList = List(3,56,1,4,72)

myList: List[Int] = List(3, 56, 1, 4, 72)

scala> // map()参数是一个函数

scala> myList.map((x) => 2\*x)

res15: List[Int] = List(6, 112, 2, 8, 144)

scala> //尝试给map()函提供一个方法作为参数

scala> def m4(x:Int) = 3\*x

m4: (x: Int)Int

scala> //正常执行

scala> myList.map(m4)

res17: List[Int] = List(9, 168, 3, 12, 216)

这是因为，如果期望出现函数的地方我们提供了一个方法的话，该方法就会自动被转换成函数。该行为被称为ETA expansion。

这样的话使用函数将会变得简单很多。你可以按照下面的代码验证该行为：

scala> //期望出现函数的地方，我们可以使用方法

scala> val f3:(Int)=>Int = m4

f3: Int => Int = <function1>

scala> //不期望出现函数的地方，方法并不会自动转换成函数

scala> val v3 = m4

<console>:8: error: missing arguments for method m4;

follow this method with `\_‘ if you want to treat it as a partially applied function

val v3 = m4

^

利用这种自动转换，我们可以写出很简洁的代码，如下面这样

scala> //10.<被解释成obj.method，即整形的<的方法，所以该表达式是一个方法，会被解释成函数

scala> myList.filter(10.<)

res18: List[Int] = List(56, 72)

因为在scala中操作符被解释称方法

* 前缀操作符：op obj 被解释称obj.op
* 中缀操作符：obj1 op obj2被解释称obj1.op(obj2)
* 后缀操作符：obj op被解释称obj.op

你可以写成10<而不是10.<

scala> myList.filter(10<)

warning: there were 1 feature warning(s); re-run with -feature for details

res19: List[Int] = List(56, 72)

如何强制把一个方法变成函数

可以在方法名后面加一个下划线强制变成函数，部分应用函数

scala> val f4 = m4 \_

f4: Int => Int = <function1>

scala> f4(2)

res20: Int = 6

1. =>表示函数的类型(Function Type)  
  
例如：

|  |  |
| --- | --- |
| 1 | def double(x: Int): Int = x\*2 |

函数double的类型就是 (x: Int) => Int 或者 Int => Int  
  
备注： 当函数只有一个参数的时候，函数类型里面括起来函数参数的括号是可以省略的。  
  
使用函数的类型，就可以定义函数变量(Function Value/Function Variable)  
  
那么现在定义一个函数变量:  
  
var x : (Int) => Int = double  
  
2. 匿名函数(Anonymous functions/Function Literals/Lambda)  
  
例如：通过匿名函数定义一个函数变量xx  
  
var xx = (x: Int) => x + 1  
  
或者，给一个高阶函数，传递一个函数：  
  
val newList = List(1,2,3).map { (x: Int) => x \* 2 }  
  
3. Call-by-name and Call-by-value

Scala的解释器在解析函数参数(function arguments)时有两种方式：先计算参数表达式的值(reduce the arguments)，再应用到函数内部；或者是将未计算的参数表达式直接应用到函数内部。前者叫做传值调用（call-by-value），后者叫做传名调用（call-by-name）。

**[plain]** [view plain](http://blog.csdn.net/asongoficeandfire/article/details/21889375) [copy](http://blog.csdn.net/asongoficeandfire/article/details/21889375)

1. package com.doggie
3. object Add {
4. def addByName(a: Int, b: => Int) = a + b
5. def addByValue(a: Int, b: Int) = a + b
6. }

addByName是传名调用，addByValue是传值调用。语法上可以看出，使用传名调用时，在参数名称和参数类型中间有一个=》符号。

以a为2，b为2 + 2为例，他们在Scala解释器进行参数规约（reduction）时的顺序分别是这样的：

**[plain]** [view plain](http://blog.csdn.net/asongoficeandfire/article/details/21889375) [copy](http://blog.csdn.net/asongoficeandfire/article/details/21889375)

1. addByName(2, 2 + 2)
2. ->2 + (2 + 2)
3. ->2 + 4
4. ->6
6. addByValue(2, 2 + 2)
7. ->addByValue(2, 4)
8. ->2 + 4
9. ->6

可以看出，在进入函数内部前，传值调用方式就已经将参数表达式的值计算完毕，而传名调用是在函数内部进行参数表达式的值计算的。

这就造成了一种现象，每次使用传名调用时，解释器都会计算一次表达式的值。对于有副作用(side-effect)的参数来说，这无疑造成了两种调用方式结果的不同。

假设有如下两个函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def doubles(x: => Int) = {      println("Now doubling " + x)      x\*2  }    def f(x: Int): Int {      println(s"Calling f($i)")      i  } |

那么对于函数doubles而言，它的参数x就是by-name的。如果调用doubles的时候，直接给个普通的值或者非函数变量。那么doubles的执行结果就跟普通的函数没有区别。但是当把一个返回值为Int类型的函数，例如f(2)，传递给doubles的时候。那么f(2)会被先计算出返回值2，返回值2传入doubles参与运算。运算完成以后，f(2)会被doubles在执行以后，再调用一遍。

传名参数是一个方法

传名参数实质是一个没有参数列表的方法。正是因此你才可以使用名字调用而不用添加()

scala> //使用两次‘x‘，意味着进行了两次方法调用

scala> def m1(x: => Int)=List(x,x)

m1: (x: => Int)List[Int]

scala> import util.Random

import util.Random

scala> val r = new Random()

r: scala.util.Random = scala.util.Random@d4c330b

scala> //因为方法被调用了两次，所以两个值不相等

scala> m1(r.nextInt)

res21: List[Int] = List(-1273601135, 2004676878)

如果你在方法体部分缓存了传名参数（函数），那么你就缓存了值（因为x函数被调用了一次）

scala> //把传名参数代表的函数缓存起来

scala> def m1(x: => Int) ={val y=x;List(y,y)}//此时的y为Int

m1: (x: => Int)List[Int]

scala> m1(r.nextInt)

res22: List[Int] = List(-1040711922, -1040711922)

能否在函数体部分引用传名参数所代表的方法呢，是可以的(缓存的是传名参数所代表的方法)。

scala> def m1(x: => Int)={val y=x \_;List(y(),y())}//此时的y为函数

m1: (x: => Int)List[Int]

scala> m1(r.nextInt)

res23: List[Int] = List(-1982925840, -933815401)