第八章 函数编程

第一节

1.在object中定义的函数,可以直接使用

object LongLines {

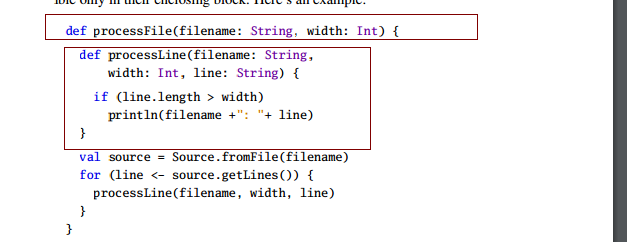
def processFile(filename: String, width: Int) {

}

}

LongLines.processFile()

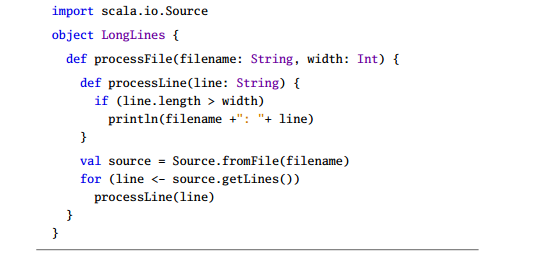
第二节 Local functions 本地函数



即def 里面套着def,,被套着的就是本地函数

特点:

1. 内部的是私有函数,外界不能调用
2. 内部私有函数可以访问外部的属性,因此width和filename是可以不用作为参数传递到私有函数的,比如:



第三节 First-class functions

1. 函数作为一等公民,可以当作一个值传递到其他函数中
2. 函数定义

(x: Int) => x + 1 函数是有返回值的,因此可以将函数赋予任意一个变量

var increase = (x: Int) => x + 1

输出 increase: (Int) => Int =<function> 表示increase是一个函数,输入int 输出int

调用 increase(10)

someNumbers.filter( (x: Int) => x > 0 ) filter接收一个function作为参数

第四节 Short forms of function literals 函数的短小形式

1. 移除参数类型

someNumbers.filter( (x) => x > 0 )

因为编译器知道集合是int类型,因此x的类型可以移除,主要是filter接收的function是要处理集合中的每一个元素.因此集合是知道元素的类型的,所以可以移除

1. 当类型是可以推测出来的时候,那么括号也可以被移除,即

someNumbers.filter(x => x > 0)

第五节 Placeholder syntax 占位符

当每一个参数只是使用一次的时候,可以使用占位符,占位符按照出现的次数,赋予不同参数值

1. 为了让函数更简洁,可以使用\_去代替一个或者多个参数

例如 someNumbers.filter(\_ > 0)

该表达式是说 每一个元素都用\_代替,然后就可以进行与0的比较了

1. val f = \_ + \_ 定义一个函数,用两个占位符,但是编译器是不知道类型的,因此会报错.此时应该赋予占位符类型

val f = (\_: Int) + (\_: Int)

输出 f: (Int, Int) => Int = <function>

即表示f需要两个int参数 参与运算

第一个占位符表示第一个参数，第二个占位符表示第二个参数,因此一个参数只能使用一次的时候才可以用占位符占位

课外学习

方法和函数

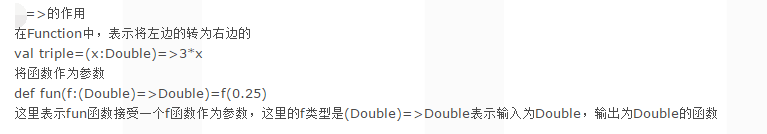
首先函数在scala中是一等公民,而不是方法,即方法不是一等公民,函数可以作为变量或者作为参数传递给另外一个函数方法:







注意:上面传递方法当作函数的时候,按道理应该调用方法的\_,将其转换成函数才可以,但是发现却没有,说明有一个隐式转换操作。



scala> def m(x:Int) = x

m: (x: Int) Int

以上表示定义了一个方法m,需要int参数,返回值也是int

调用

scala> m(3)

res2: Int = 3

说明调用后返回的是int为3,赋予给变量res2了

函数:

scala> val f = (x:Int) => x

f: Int => Int = <function1>

注意:函数f不需要设置返回值,而def方法需要设置返回值,函数的返回值是会自动推导出来的

表示定义了一个函数f,接收int参数,返回值是int,注意后面函数与方法的区别是有= <function1>

即函数在scala中就是实现了22个特质（trait）抽象出了函数的概念。这22特质从Function1到Function22

scala> f(2)

res2: Int = 2 说明函数和方法一样,可以直接被加入参数就调用.返回值给一个变量

区别:

1．执行f 返回 res0: Int => Int = <function1> 即res0这个变量表示的是一个函数

而执行m,则抛异常,即m不能被单独执行,必须加入参数,得到具体的返回值,而不是指代一个函数给一个变量。

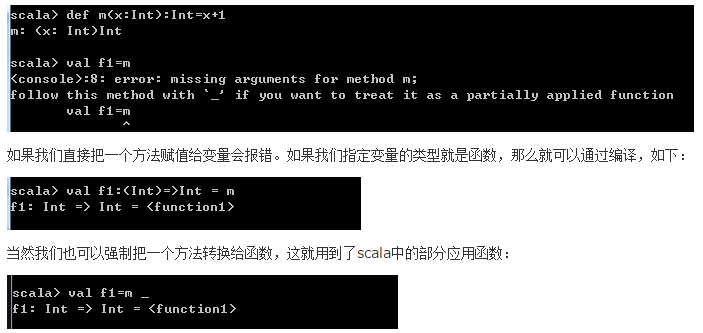
即:方法不能作为单独的表达式而存在（参数为空的方法除外），而函数可以

原因是f本身就是一个对象（实现了FunctionN特质的对象）,因此相当于变量赋予了一个对象,因此函数当作表达式传递是没问题的,可以相当于传递的是一个对象,但是方法却不行

2．方法名是方法调用，而函数名只是代表函数对象本身

因为函数保存字面量的变量本身就是保存了实现了FunctionN特质的类的对象，因此要调用对象的apply方法,就需要使用obj()的语法,因此函数名后面加()才是真的调用函数

3. 在需要函数的地方，如果传递一个方法，会自动进行ETA展开（把方法转换为函数）



即如果一个地方需要函数,则不能传递方法进去,这样会报错,当然我们可以强制转换方法到函数,使用的就是scala的Partially Applied Function

4．

注意:def m(x: => Int) = List(x,x) 这个参数比较奇怪,含义如何?

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

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

scala> def method1 = 1

method1: Int

scala> m(method1)

res0: List[Int] = List(1, 1)

scala> def method2(x:Int) = x+2

method2: (x: Int)Int

scala> m(method2(5))

res1: List[Int] = List(7, 7)

scala> def method3(x:Int,y:Int) = x+y

method3: (x: Int, y: Int)Int

scala> m(method3(6,7))

res2: List[Int] = List(13, 13)

scala> def method4(x:Double) = 5

method4: (x: Double)Int

scala> m(method4(6.7))

res3: List[Int] = List(5, 5)

以上代码的说明:

1.首先m方法接收一个参数是,该参数是一个函数,无论参数是什么,返回值是int的函数即可

2.List(x,x),说明调用了x函数两次,因此无论x函数多么复杂,都会被调用2次,因为List中相当于执行了两次x函数

3.定义了无参数的方法method1,以及一个参数的方法method2,以及两个参数的method3,以及非Int参数的method4,但是返回值都是Int,因此发现都可以运行成功,

说明m接受的参数就是一个函数,返回int的函数,参数值是不关注的



第六节 Partially applied functions 偏函数

自己理解:

偏函数本身就是生成一个新的函数的方式,因此可以使方法def \_ 转换成函数,这一个过程也是一个生成新函数的过程

val b = println

输出 b: Unit = ()

val b = println \_ 即将println方法转换成函数,而且是无参数的函数对象,而不是方法了,是一个class对象

输出 b: () => Unit = <function0>

1. 上节讲述的是\_占位符表示的一个独立的参数,但是其实他还可以表示全部参数集合

此时\_就不再表示一个独立的参数了,而是表示全部的参数集合

1. 表示独立参数和参数集合有什么区别呢,

当表示参数集合的时候,要在函数和\_之间有一个空格,比如println \_

否则如果没有空格.则编译器会寻找println\_这个方法名,因此是找不到该方法的,因此报错。

而println \_ 的含义就是表示重新定义一个println函数,参数和println需要的参数相同的函数。

1. 当执行一个函数的时候,需要传递函数需要的全部参数,比如

def sum(a: Int, b: Int, c: Int) = a + b + c

返回值 sum: (a: Int,b: Int,c: Int) Int

而partially applied function这个函数是有所不同的,不需要全部参数都给,甚至不给任意一个参数都可以。

比如

val a = sum \_ 将一个方法转换成函数,即函数是一个function对象,有sum参数个数相同的参数在apply方法里面,apply方法的实现体就是sum方法的具体实现

输出 a: (Int, Int, Int) => Int = <function>

调用 a(1, 2, 3)

1. 也可以将一个方法转换成一个新的函数,不需要有方法的所有参数

scala> val b = sum(1, \_: Int, 3)

b: (Int) => Int =

注意:上述转换中其实省略了一个隐式转换,即将sum方法转换成函数的sum \_,然后对函数进行部分属性赋值,产生一个新的函数,新的函数接收一个参数,在apply中调用sum真正的函数,带有3个参数

第7节 Closures 闭包

1. 到现在为止,我们用的函数中引用的参数都是传递到函数中的参数,比如(x: Int) => x > 0,函数体用到的x是函数中参数传递进来的,然而引用的参数被定义在其他地方呢?如何?
2. 看一下下面运行良好的例子

var more = 1

scala> val addMore = (x: Int) => x + more

addMore: (Int) => Int =

scala> addMore(10)

res17: Int = 11

我们可以看到函数可以使用外部定义的more变量

但是引用的外部属性是值还是属性的引用呢

1. 例子

scala> more = 9999

more: Int = 9999

scala> addMore(10)

res18: Int = 10009

我们会发现当我们改变more的值的时候,依然会变化,因此我们此时认为闭包引用的是外部的变量引用,而不是值

1. 可以理解成java中一个方法可以调用该类里面的属性一样的道理

第8节 Special function call forms 特别的函数调用形式

1. Scala除了正常的函数调用外,还支持三种特别的调用方式

repeated parameters, named arguments, and default arguments.

1. repeated parameters 可以在最后一个参数位置上.使用\*号,传递一组参数

scala> def echo(args: String\*) = for (arg <- args) println(arg)

echo: (args: String\*) Unit 可以看到定义一个方法echo,有任意多个string参数类型,无返回值

注意:

a.任意多,表示0个和任意个

b.真实的最后一个参数,其实是Array[String]类型的参数

c.如果传递一个Array参数的时候也是非法的,即

scala> val arr = Array("What's", "up", "doc?")

arr: Array[java.lang.String] = Array(What's, up, doc?)

scala> echo(arr) 会抛异常,因为很明显echo不支持数组参数

因此要想调用的话,则使用echo(arr: \_\*),这会告诉编译器,传递的是该数组的每一个元素作为参数

d.同样,返过来理解:

def echo(aa:String,bb:String) = {println(aa+"=="+bb)}

val arr:Array[String] = Array("333","444")

echo(arr:\_\*) 也是报错的,因为echo需要的是两个String参数.而不是一个数组

1. Named arguments 通过名字去为函数对应的参数赋值调用

即正常的函数参数是按照顺序赋值的,使用命名服务,则不会按照顺序对参数赋值。

def speed(distance: Float, time: Float): Float = distance / time

speed(distance = 100, time = 10)

注意:也是可以混合使用位置和name方式的,直到第一个name出现为止,前面的都是按照顺序赋值的

def speed(time1: Int, time2: Int, time3: Int, time4: Int) = { println("aaa=>"+time1+"=="+time2+"=="+time3+"=="+time4) }

调用speed(1,2,time4=3,time3=4)

aaa=>1==2==4==3

1. Default parameter value函数参数的默认值

def speed(time1: Int = 66, time2: Int, time3: Int, time4: Int) = { println("aaa=>"+time1+"=="+time2+"=="+time3+"=="+time4) }

设置time1默认值是66

因此调用speed(time2=2,time4=3,time3=4)

aaa=>66==2==4==3

注意,测试需要传递所有的需要的参数,通过名字传递了,不能通过位置传递了,因为不太容易区分位置了

第9节 Tail recursion 尾递归

1．