Scala核心编程

-函数式编程高级

尚硅谷研究院

- **偏函数(partial function)**

先看一个需求

给你一个集合val list = List(1, 2, 3, 4, "abc") ， 请完成如下要求:

1) 将集合list中的所有数字+1，并返回一个新的集合
2) 要求忽略掉 非数字 的元素，即返回的 新的集合 形式为 **(2, 3, 4, 5)**

- 偏函数

解决方式-map返回新的集合, 引出偏函数

- 思路1-map+fliter方式

```
val list = List(1, 2, 3, 4, "abc")
//思路1,使用map+fliter的思路
def f1(n:Any): Boolean = {
n.isInstanceOf[Int]
}
def f2(n:Int): Int = {
n + 1
}
def f3(n:Any): Int ={
n.asInstanceOf[Int]
}
val list2 = list.filter(f1).map(f3).map(f2)
println("list2=" + list2)

//写代码演示+问题分析
```

- 偏函数

解决方式-map返回新的集合，引出偏函数

➢ 思路2-模式匹配

```
def addOne( i : Int ): Int = {
i match {
case x:Int => x + 1
case _ => _
}
}
val list = List(1, 2, 3, 4, "abc")
val list2 = list.map(addOne2)
println("list2=" + list2)
```

改进

```
def addOne2( i : Any ): Any = {
i match {
case x:Int => x + 1
case _ =>
}
}
```

- **偏函数**

基本介绍

1) **在对符合某个条件，而不是所有情况进行逻辑操作时，使用偏函数是一个不错的选择**
2) 将包在大括号内的一组case语句封装为函数，我们称之为偏函数，它只对会作用于指定类型的参数或指定范围值的参数实施计算，超出范围的值会忽略（未必会忽略，这取决于你打算怎样处理）

3) 偏函数在Scala中是一个特质PartialFunction

- 偏函数

偏函数快速入门

使用偏函数解决前面的问题，【代码演示+说明】

```
val list = List(1, 2, 3, 4, "abc")
//说明
val addOne3= new PartialFunction[Any, Int] {
def isDefinedAt(any: Any) = if (any.isInstanceOf[Int]) true else false
def apply(any: Any) = any.asInstanceOf[Int] + 1
}
val list3 = list.collect(addOne3)
println("list3=" + list3) //?
```

- 偏函数

偏函数的小结

```
val addOne3= new PartialFunction[Any, Int] {
  def apply(any: Any) = any.asInstanceOf[Int] + 1
  def isDefinedAt(any: Any) = if (any.isInstanceOf[Int]) true else false
}
```


图片10.z

1) 使用构建特质的实现类(使用的方式是PartialFunction的匿名子类)
2) PartialFunction 是个特质(看源码)
3) 构建偏函数时，参数形式 [Any, Int]是泛型，第一个表示参数类型，第二个表示返回参数
4) 当使用偏函数时，会遍历集合的所有元素，编译器执行流程时先执行isDefinedAt()如果为true ,就会执行 apply, 构建一个新的Int 对象返回
5) 执行isDefinedAt() 为false 就过滤掉这个元素，即不构建新的Int对象.
6) map函数不支持偏函数，因为map底层的机制就是所有循环遍历，无法过滤处理原来集合的元素
7) collect函数支持偏函数

- 偏函数

偏函数简化形式

声明偏函数，需要重写特质中的方法，有的时候会略显麻烦，而Scala其实提供了简单的方法

1) 简化形式1

```
def f2: PartialFunction[Any, Int] = {
  case i: Int => i + 1 // case语句可以自动转换为偏函数
}
val list2 = List(1, 2, 3, 4,"ABC").collect(f2)
```

2) 简化形式2

```
val list3 = List(1, 2, 3, 4,"ABC").collect{ case i: Int => i + 1 }
println(list3)
```

- 作为参数的函数

基本介绍

函数作为一个变量传入到了另一个函数中，那么该作为参数的函数的类型是：function1，即：(参数类型) => 返回类型

应用实例

```
//说明
def plus(x: Int) = 3 + x
//说明
val result1 = Array(1, 2, 3, 4).map(plus(_))
println(result1.mkString(","))
```

[案例演示+代码说明]

- 作为参数的函数

应用实例小结

1) map(plus(_)) 中的plus(_) 就是将plus这个函数当做一个参数传给了map，_ 这里代表从集合中遍历出来的一个元素。

2) plus(_) 这里也可以写成 plus 表示对 Array(1,2,3,4) 遍历，将每次遍历的元素传给plus的 x
3) 进行 3 + x 运算后，返回新的Int ， 并加入到新的集合 result1中
4) def map[B, That](f: A => B) 的声明中的 f: A => B 一个函数

- 匿名函数

基本介绍

没有名字的函数就是匿名函数，可以通过函数表达式
来设置匿名函数

应用实例

```
val triple = (x: Double) => 3 * x
println(triple(3))
```

说明

1) (x: Double) => 3 * x 就是匿名函数
2) (x: Double) 是形参列表， => 是规定语法表示后面是函数体， 3 * x 就是函数体，如果有多行，可以 {} 换行写.
3) triple 是指向匿名函数的变量。

- 匿名函数

课堂案例

请编写一个匿名函数，可以返回2个整数的和，并输出该匿名函数的类型.

```
val f1 = (n1: Int, n2: Int ) => {
println("匿名函数被调用")
n1 + n2
}
println("f1类型=" + f1)
println(f1(10, 30))
```


图片11.z

- 高阶函数

基本介绍

能够接受函数作为参数的函数，叫做高阶函数 (higher-order function)。可使应用程序更加健壮。

高阶函数基本使用

```
//test 就是一个高阶函数，它可以接收f: Double =>
Double
def test(f: Double => Double, n1: Double) = {
f(n1)
}
//sum 是接收一个Double,返回一个Double
def sum(d: Double): Double = {
d + d
}
val res = test(sum, 6.0)
println("res=" + res)
```

- 高阶函数

高阶函数可以返回函数类型

```
def minusxy(x: Int) = {
 (y: Int) => x – y //匿名函数
}
val result3 = minusxy(3)(5)
println(result3)
```

【案例演示+说明】，修改代码问输出结果?

- 高阶函数

高级函数案例的小结

➢ 说明: def minusxy(x: Int) = (y: Int) => x - y

1) 函数名为 minusxy
2) 该函数返回一个匿名函数
   (y: Int) = > x -y

➢ 说明val result3 = minusxy(3)(5)

1) minusxy(3)执行minusxy(x: Int)得到 (y: Int) => 3 - y 这个匿名函
2) minusxy(3)(5)执行 (y: Int) => x - y 这个匿名函数
3) 也可以分步执行: val f1 = minusxy(3); val res = f1(90)

- 高阶函数

课堂练习

```
object Temp {
  def main(args: Array[String]): Unit = {
    def test1(x: Double) = {
      (y: Double) => x * x * y //
    }
    val res = test1(2.0)(3.0)
    println("res=" + res)// 输出什么
  }
}
```


图片1.z

res = 12.0

- 参数(类型)推断

基本介绍

参数推断省去类型信息（在**某些情况下[需要有应用场景]**，参数类型是可以推断出来的，如list=(1,2,3) list.map()　map中函数参数类型是可以推断的)，同时也可以进行相应的**简写**。

参数类型推断写法说明

1) 参数类型是可以推断时，可以省略参数类型
2) 当传入的函数，只有单个参数时，可以省去括号
3) 如果变量只在=>右边只出现一次，可以用_来代替

- 参数(类型)推断

应用案例

```
//分别说明
val list = List(1, 2, 3, 4)
println(list.map((x:Int)=>x + 1)) //(2,3,4,5)
println(list.map((x)=>x + 1))
println(list.map(x=>x + 1))
println(list.map(_ + 1))
val res = list.reduce(_+_)
```

- 参数(类型)推断

应用案例的小结

1) map是一个高阶函数，因此也可以直接传入一个匿名函数，完成map
2) 当遍历list时，参数类型是可以推断出来的，可以省略数据类型Int
   println(list.map((x)=>x + 1))
3) 当传入的函数，只有单个参数时，可以省去括号
   println(list.map(x=>x + 1))
4) 如果变量只在=>右边只出现一次，可以用_来代替
   println(list.map(_ + 1))

- 闭包(closure)

基本介绍

基本介绍：闭包就是**一个函数**和与其**相关的引用环境**组合的一个**整体**(实体)。

案例演示

```
//1.用等价理解方式改写 2.对象属性理解
def minusxy(x: Int) = (y: Int) => x - y
//f函数就是闭包.
val f = minusxy(20)
println("f(1)=" + f(1)) // 19
println("f(2)=" + f(2)) // 18
```

【案例演示+总结】

- 闭包

代码小结

1) 第1点

```
(y: Int) => x – y
```

返回的是一个匿名函数，因为该函数引用到到函数外的 x,那么 该函数和x整体形成一个闭包

如：这里 **val** f = minusxy(20) 的f函数就是闭包

2) 你可以这样理解，返回函数是一个对象，而x就是该对象的一个字段，他们共同形成一个闭包
3) 当多次调用f时（可以理解多次调用闭包），发现使用的是同一个x, 所以x不变。
4) 在使用闭包时，主要搞清楚返回函数引用了函数外的哪些变量，因为他们会组合成一个整体(实体),形成一个闭包

- 闭包

闭包的最佳实践

请编写一个程序，具体要求如下

1) 编写一个函数 makeSuffix(suffix: String) 可以接收一个文件后缀名(比如.jpg)，并返回一个闭包
2) 调用闭包，可以传入一个文件名，如果该文件名没有指定的后缀(比如.jpg) ,则返回 文件名.jpg , 如果已经有.jpg后缀，则返回原文件名。
3) 要求使用闭包的方式完成
4) String.endsWith(xx)

代码演示

- 闭包

体会闭包的好处

1) 返回的匿名函数和 makeSuffix (suffix string) 的 suffix 变量 组合成一个闭包,因为 返回的函数引用到suffix这个变量

2) 我们体会一下闭包的好处，如果使用传统的方法，也可以轻松实现这个功能，但是传统方法需要每次都传入 后缀名，比如 .jpg ,而闭包因为可以保留上次引用的某个值，所以我们传入一次就可以反复使用。大家可以仔细的体会一把！

- **函数柯里化(curry)**

基本介绍

1) 函数编程中，接受**多个参数的函数**都可以转化为接受**单个参数的函数**，这个转化过程就叫柯里化

2) 柯里化就是证明了函数只需要一个参数而已。其实我们刚才的学习过程中，已经涉及到了柯里化操作。

3) 不用设立柯里化**存在的意义**这样的命题。柯里化就是以**函数为主体这种思想**发展的必然产生的结果。(即：柯里化是面向函数思想的必然产生结果)

- 函数柯里化

函数柯里化快速入门

编写一个函数，接收两个整数，可以返回两个数的乘积，要求:

1) 使用常规的方式完成
2) 使用闭包的方式完成
3) 使用函数柯里化完成

注意观察编程方式的变化。[案例演示]

```
//说明
def mul(x: Int, y: Int) = x * y
println(mul(10, 10))

def mulCurry(x: Int) = (y: Int) => x * y
println(mulCurry(10)(9))

def mulCurry2(x: Int)(y:Int) = x * y
println(mulCurry2(10)(8))
```

- 函数柯里化

函数柯里化最佳实践

比较两个字符串在忽略大小写的情况下是否相等，注意，这里是两个任务：

1) 全部转大写（或小写）
2) 比较是否相等

针对这两个操作，我们用一个函数去处理的思想，其实也变成了两个函数处理的思想（柯里化）

【案例演示+代码说明】

方式1: **简单的方式,使用一个函数完成**.

```
def eq2(s1: String)(s2: String): Boolean = {
s1.toLowerCase == s2.toLowerCase
}
```

- 函数柯里化

函数柯里化最佳实践

```
//方式2：使用稍微高级的用法(隐式类)：形式为 str.方法()

def eq(s1: String, s2: String): Boolean = {
s1.equals(s2)}
implicit class TestEq(s: String) {
def checkEq(ss: String)(f: (String, String) => Boolean): Boolean = {
f(s.toLowerCase, ss.toLowerCase)
}}

//案例演示+说明+简化版(三种形式，直接传入匿名函数方式)
str1.checkEq(str2)(_.equals(_))
```

- 控制抽象

看一个需求

如何实现将一段代码(从形式上看)，作为参数传递给高阶函数，在高阶函数内部执行这段代码. 其使用的形式如 breakable{} 。

```
var n = 10
breakable {
while (n <= 20) {
n += 1
if (n == 18) {
  break()
}
}
}
```

- 控制抽象

控制抽象基本介绍

控制抽象是这样的函数，满足如下条件

1) 参数是函数
2) 函数参数没有输入值也没有返回值

快速入门

案例演示+代码说明

```
def myRunInThread(f1: () => Unit) = {
    new Thread {
      override def run(): Unit = {
        f1()
      }
    }.start()
  }
  myRunInThread {
    () => println("干活咯！ 5秒完成...")
      Thread.sleep(5000)
      println("干完咯！ ")
  }
```

- 控制抽象

简化处理，省略()，如下形式

```
def myRunInThread(f1:  => Unit): Unit = {//说明
  new Thread {
    override def run(): Unit = {
      f1
    }
  }.start()
}
myRunInThread { //说明
    println("干活咯！ 5秒完成...")
    Thread.sleep(5000)
    println("干完咯！ ")
}
```

- 控制抽象

进阶用法：实现类似while的until函数

```
var x = 10
def until(condition: => Boolean)(block: => Unit): Unit = {
//类似while循环，递归
if (!condition) {
block
until(condition)(block)
}
//      println("x=" + x)
//      println(condition)
//      block
//      println("x=" + x)
}

until(x == 0) {
x -= 1
println("x=" + x)
```

谢谢！ 欢迎收看！