Scala基础

目录

[Scala基础 1](#_Toc495679220)

[目录 2](#_Toc495679221)

[图表 4](#_Toc495679222)

[1、 Scala简介 5](#_Toc495679223)

[2、Scala安装及IDEA环境配置 5](#_Toc495679224)

[2.1 安装Scala 5](#_Toc495679225)

[2.2 安装Idea 6](#_Toc495679226)

[3、基本语法规则 6](#_Toc495679227)

[3.1 变量声明 6](#_Toc495679228)

[3.2 函数声明 7](#_Toc495679229)

[3.3 Class 7](#_Toc495679230)

[3.4 Object 8](#_Toc495679231)

[3.5 apply方法 8](#_Toc495679232)

[3.6 块表达式 9](#_Toc495679233)

[4、控制结构 9](#_Toc495679234)

[4.1 if 9](#_Toc495679235)

[4.2 for 9](#_Toc495679236)

[5、函数 10](#_Toc495679237)

[5.1 函数基本构成 11](#_Toc495679238)

[5.2 匿名函数（函数字面量） 11](#_Toc495679239)

[5.2.1 语法 11](#_Toc495679240)

[5.2.2 占位符简化 13](#_Toc495679241)

[5.3 函数传名调用（call-by-name） 13](#_Toc495679242)

[5.3.1 概念 13](#_Toc495679243)

[5.3.2 举例：酒鬼喝酒 14](#_Toc495679244)

[5.3.3 传名调用 vs 传值调用 15](#_Toc495679245)

[5.4 指定函数参数名 16](#_Toc495679246)

[5.5 可变参数 17](#_Toc495679247)

[5.6 默认参数值 17](#_Toc495679248)

[5.7 递归函数 18](#_Toc495679249)

[5.8 高阶函数 19](#_Toc495679250)

[5.9 内嵌函数 19](#_Toc495679251)

[5.10 偏函数 20](#_Toc495679252)

[5.10.1 isDefinedAt 21](#_Toc495679253)

[5.10.2 orElse 22](#_Toc495679254)

[5.10.3 andThen 23](#_Toc495679255)

[5.10.4 applyOrElse 24](#_Toc495679256)

[5.11 偏应用函数 24](#_Toc495679257)

[5.12 函数柯里化 25](#_Toc495679258)

[5.12.1 实例 25](#_Toc495679259)

[5.12.2 实现过程 25](#_Toc495679260)

[5.12.3 完整实例 26](#_Toc495679261)

[5.13 闭包 26](#_Toc495679262)

[6、集合（Collection） 27](#_Toc495679263)

[6.1 可变和不可变 27](#_Toc495679264)

[6.2 List 列表 31](#_Toc495679265)

[6.2.1 List基本操作 32](#_Toc495679266)

[6.2.2 连接列表 33](#_Toc495679267)

[6.2.3 List.fill() 34](#_Toc495679268)

[6.2.4 List.tabulate() 34](#_Toc495679269)

[6.2.5 其他常用操作 35](#_Toc495679270)

[6.3 Set 35](#_Toc495679271)

[6.3.1 集合基本操作 36](#_Toc495679272)

[6.3.2 连接集合 37](#_Toc495679273)

[6.3.3 查找集合中最大与最小元素 38](#_Toc495679274)

[6.3.4 交集 38](#_Toc495679275)

[6.4 Map 39](#_Toc495679276)

[6.4.1 Map 基本操作 40](#_Toc495679277)

[6.4.2 Map 合并 41](#_Toc495679278)

[6.4.3 输出 Map 的 keys 和 values 42](#_Toc495679279)

[6.4.4 查看 Map 中是否存在指定的 Key 42](#_Toc495679280)

[6.5 Tuple 元组 43](#_Toc495679281)

[6.5.1 迭代元组 44](#_Toc495679282)

[6.5.2 元组转为字符串 44](#_Toc495679283)

[6.5.3 元素交换 45](#_Toc495679284)

[6.6 Option 45](#_Toc495679285)

[6.7 map和flatMap方法 46](#_Toc495679286)

[6.8 Java与Scala集合相互转换 47](#_Toc495679287)

[6.8.2 双向类型转换 47](#_Toc495679288)

[6.8.2 单向类型转换 48](#_Toc495679289)

[7、类和对象 49](#_Toc495679290)

[7.1 类、字段和方法 49](#_Toc495679291)

[7.2 Singleton 单例对象 51](#_Toc495679292)

[7.3.1 单例对象作为程序的入口点 52](#_Toc495679293)

[7.3.2 伴生对象实例 52](#_Toc495679294)

[8、特质（特征） 53](#_Toc495679295)

[8.1 概念 53](#_Toc495679296)

[8.2 瘦接口对阵胖接口 56](#_Toc495679297)

[8.3 特质用来做可堆叠的改变 56](#_Toc495679298)

[8.4 特质构造顺序 59](#_Toc495679299)

[8.5 class，Object，Trait有什么区别？ 60](#_Toc495679300)

[8.5.1 class 60](#_Toc495679301)

[8.5.2 object 60](#_Toc495679302)

[8.5.3 trait 60](#_Toc495679303)

[8.5.4 包含class，Object，Trait的完整实例 61](#_Toc495679304)

[9、case class和模式匹配 62](#_Toc495679305)

[9.1 模式匹配 62](#_Toc495679306)

[9.2 case class 样例类 63](#_Toc495679307)

[10、隐式转换 65](#_Toc495679308)

[10.1 什么是隐式转换 65](#_Toc495679309)

[10.2 隐式值 65](#_Toc495679310)

[10.3 隐式视图 66](#_Toc495679311)

[10.3.1 隐式转换为目标类型：把一种类型自动转换到另一种类型 66](#_Toc495679312)

[10.3.2 隐式转换调用类中本不存在的方法 67](#_Toc495679313)

[10.3.3 结论 68](#_Toc495679314)

[10.4 隐式类 68](#_Toc495679315)

[10.5 隐式操作规则 69](#_Toc495679316)

[10.5.1 隐式转换的时机： 69](#_Toc495679317)

[10.5.2 隐式解析的查找范围 69](#_Toc495679318)

[10.5.3 隐式转换规则 70](#_Toc495679319)

[11、Future和Promise 70](#_Toc495679320)

[11.1 Future 70](#_Toc495679321)

[11.2 Promise 72](#_Toc495679322)

[12、问题进阶 72](#_Toc495679323)

[12.1 apply方法是什么 73](#_Toc495679324)

[12.1.1 Class和Object中的Apply()方法 73](#_Toc495679325)

[12.1.2 apply方法的用法 74](#_Toc495679326)

[12.2 Scala中的class怎么定义构造器（get/set） 75](#_Toc495679327)

[12.2.1 主从构造器 75](#_Toc495679328)

[12.2.2 访问器 76](#_Toc495679329)

图表

[图6- 1 scala.collection.immutable. 29](file:///D:\我的文档\马燊偲\工作\Scala基础.docx#_Toc495679215)

[图6- 2 scala.collection.mutable. 30](file:///D:\我的文档\马燊偲\工作\Scala基础.docx#_Toc495679216)

# Scala简介

酷炫的一行代码：

Source.fromFile("D:/f.txt", "UTF-8").getLines().toList.distinct.sortBy(s => (s.charAt(0), s.length)).foreach( println \_)

虽然只有一行代码，但是却做了很多事情：以UTF-8编码读取文件所有行 -> 去重 -> 按首字符排序，首字符相同按长度排序 -> 打印结果。各位脑补一下Java的实现。

Scala将**面向对象编程(OOP)编程语言**和**函数式编程(FP)语言**完美的融合到一起，形成一门更加强大的JVM语言。与Java相比，Scala的设计更加一致：

* **一切都是对象**

1.toDouble //可以直接调用基本类型上的方法

"1".toInt //将字符串转换成整型

* **一切都是方法**

"a" \* 3 //等价于: "a".\*(3)

2 - 1 //等价于: 2.-(1)

* **一切都是表达式**

val i = if(true){ 1 } else { 0 } // i = 1

Scala里，函数就是对象；函数类型是能够被子类继承的类；每个值都是对象；每个操作都是方法调用。

# 2、Scala安装及IDEA环境配置

## 2.1 安装Scala

1、下载scala-2.11.8.msi安装包：

去官网<http://www.scala-lang.org/>，然后点击导航栏的DOWNLOAD，进入下载链接：<http://www.scala-lang.org/download/>，直接点击上面的按钮下载Windows安装包，准备安装。

2、双击scala-2.11.8.msi安装包，启动安装程序。

3、配置scala环境变量：和配置JDK环境变量一样：

① 系统变量新增：SCALA\_HOME；值：C:\Program Files (x86)\scala（scala安装路径）

② 编辑Path变量：在后追加：;%SCALA\_HOME%\bin;%SCALA\_HOME%\jre\bin

4、cmd下输入scala -version查看是否安装成功

## 2.2 安装Idea

1、下载 ideaIU-2017；

2、双击ideaIU-2016.3.1.exe进行安装；

3、安装完之后，打开IDEA；

4、安装scala插件：File->Setting->Plugins->scala进行安装；

5、创建项目时，要引入jdk和scala。

# 3、基本语法规则

## 3.1 变量声明

val用于定义不可变变量，var用于定义可变变量，这里的"可变"指的是引用的可变性。val定义的变量类似于Java的final变量，即变量只能赋一次值：

（val类型的变量指的是引用不可更改，即指向的地址不变。但是引用对象内容可以改变。）

val msg = "hello" // 等价于：val msg: String = "hello"

var i = 1 // 等价于：var i: Int = 1

i = i + 1

* 变量后面的类型声明可以省略，每行代码末尾的分号“;”也可以省略。

## 3.2 函数声明

def用于定义函数：

def max(x: Int, y: Int): Int = {

if (x > y) { x } else { y }

}

val maxVal = max(1, 2) // 2

Scala是函数式语言，所以你可以像基本类型那样把函数赋给一个变量：

val max = (x: Int, y: Int) => {

if (x > y) { x } else { y }

}

val maxVal = max(1, 2) // 2

等号"="右边是一个匿名函数，也就是我们常说的Lambda函数，匿名函数由参数和函数体两部分组成，中间用"=>"隔开，这里省略了max变量的类型，因为编译器可以自动推断出来，完整的写法如下：

val max: (Int, Int) => Int = (x: Int, y: Int) => {

if (x > y) { x } else { y }

}

max的类型是(Int, Int) => Int，即接受两个Int参数，产生一个Int返回值的函数类型。

## 3.3 Class

Scala的class定义和Java很相似：

class Counter {

private var value = 0 //你必须初始化字段

def increment() { value += 1} //方法默认public

def current() = value

}

Scala的源文件中可以定义多个类，并且默认都是public，所以外界都可以看见。class的使用也很简单：

val myCounter = new Counter //或new Counter()

myCounter.increment()

println(myCounter.current) //或myCounter.current()

* Scala中如果对象方法或类的构造器没有参数，则括号"()"可以省略。

## 3.4 Object

Scala没有静态方法和静态字段，而是提供了object对象，也就是Java中的单例对象，即全局只有一个实例。

object Accounts {

private var lastNumber = 0

def newUniqueNumber() = { lastNumber += 1; lastNumber }

}

因为Accounts是一个单例对象，可以直接使用而无需初始化：

val uniqueNumber = Accounts.newUniqueNumber

object的另一个用法是作为类的伴生对象， 类似于Java类上的静态方法，只不过Scala将Java类上的静态功能全交给object实现了。object作为伴生对象时必须和类在同一个源文件中定义，并且可以相互访问私有属性。

## 3.5 apply方法

如果某个对象obj上定义了apply方法，则我们可以这样调用：

obj(arg1, ... , argn)

是的，你猜对了，伴生对象上的apply方法立马就派上用场了，例如List类有一个同名的伴生对象List，那么你可以这样初始化一个列表：

val list = List("a", "b", "c")

想想下面的Java版本，是不是感觉幸福感油然而生：

List<String> list = new ArrayList<String>();

list.add("a");

list.add("b");

list.add("c");

## 3.6 块表达式

在Scala中一切都是表达式，如果表达式含有多条语句，则使用大括号"{}"括起来，形成一个块表达式，块表达式的最后一条语句的值作为整个块的返回值。

val r = {

val i = 1

val j = 2

i + j

} // r = 3

# 4、控制结构

## 4.1 if

if语句同样是表达式，拥有返回值：

val i = 1

val r = if(i > 0){ 1 } else { 0 } // r = 1

## 4.2 for

Scala中for语句功能比Java要丰富很多，你可以使用for遍历一个List：

val list = List(1, 2, 3)

for(i <- list){

println(i)

}

你也可以使用模式匹配遍历一个Map：

val map = Map(("a", 1), ("b", 2))

for((k, v) <- map){

println(k + ": " + v)

}

如果循环体以yield开始，for语句会返回一个新的集合：

val newList1 = for(i <- List(1, 2, 3)) yield i \* 2 // List(2, 4, 6)

val newList2 =

for{

i <- List(1, 2)

j <- List(3, 4)

} yield i + j //List(4, 5, 5, 6)

* 如果有多个集合需要遍历，则for语句后面的圆括号"()"要换成大括号"{}"。

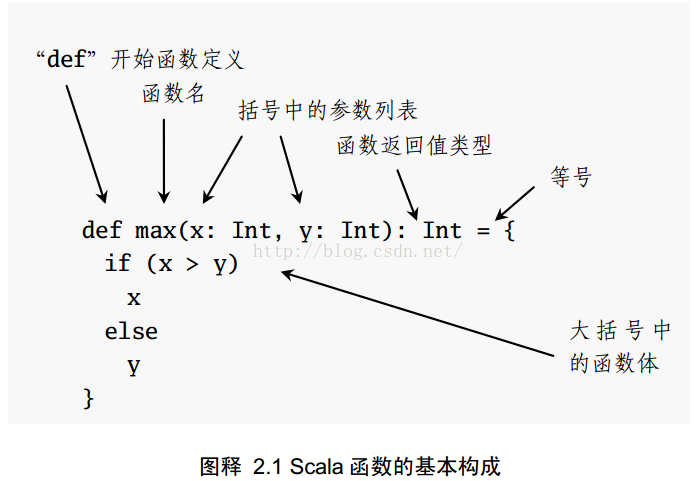
# 5、函数

函数是一组一起执行一个任务的语句。你可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由你来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

Scala 有函数和方法，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。

我们可以在任何地方定义函数，甚至可以在函数内定义函数（内嵌函数）。更重要的一点是 Scala 函数名可以有以下特殊字符：+, ++, ~, &,-, -- , \, /, : 等。

## 5.1 函数基本构成

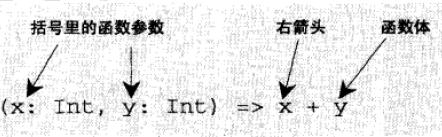


这是Scala中的函数基本构成。

## 5.2 匿名函数（函数字面量）

### 5.2.1 语法

Scala 中定义匿名函数的语法很简单，箭头左边是参数列表，右边是函数体。



=> 指明这个函数把左边的东西转换成右边的东西。

使用匿名函数后，我们的代码变得更简洁了。

下面的表达式就定义了一个接受一个Int类型输入参数的匿名函数:

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

上述定义的匿名函数，其实是下面这种写法的简写：

def add2 = new Function1[Int,Int]{

    def apply(x:Int):Int = x+1;

}

以上实例的 inc 现在可作为一个函数，使用方式如下：

var x = inc(7)-1

同样我们可以在匿名函数中定义多个参数：

var mul = (x: Int, y: Int) => x\*y

mul 现在可作为一个函数，使用方式如下：

println(mul(3, 4))

我们也可以不给匿名函数设置参数，如下所示：

var userDir = () => { System.getProperty("user.dir") }

userDir 现在可作为一个函数，使用方式如下：

println( userDir() )

实例：

object Demo {

def main(args: Array[String]) {

println( "multiplier(1) value = " + multiplier(1) )

println( "multiplier(2) value = " + multiplier(2) )

}

var factor = 3

val multiplier = (i:Int) => i \* factor

}

将以上代码保持到 Demo.scala 文件中，执行以下命令：

$ scalac Demo.scala

$ scala Demo

输出结果为：

multiplier(1) value = 3

multiplier(2) value = 6

### 5.2.2 占位符简化

占位符简化即用下划线“\_”来替代一个或多个参数。只要某个参数只在函数体里出现一次，则可以使用下划线 \_ 来替换这个参数：

scala> someNumbers.filter(\_ > 0)

res9: List[Int] = List(5, 10)

\_ > 0 相当于x => x > 0，遍历时会使用当前相应元素来替换下划线。可以这样来理解：就像我们以前做过的填空题，“\_”为要填的空，Scala 来完成这个填空题，你来定义填空题。

使用占位符时，有时无法推导出类型，如：

scala> val f = \_ + \_

此时需明确写出类型：

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

有多少个下划线，则就表示有多少个不同的参数。多个占位符时，第一个下划线表示第一个参数，第二个下划线表示第二个参数，以此类推；多个下划线指代多个参数，而不是单个参数的重复使用。所以同一参数多处出现时是无法使用这种占位符来表示的。

## 5.3 函数传名调用（call-by-name）

### 5.3.1 概念

Scala的解释器在解析函数参数(function arguments)时有两种方式：

* 传值调用（call-by-value）：先计算参数表达式的值，再应用到函数内部；
* 传名调用（call-by-name）：将未计算的参数表达式直接应用到函数内部。

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。

object Add {

  def addByName(a: Int, b: => Int) = a + b   //传名调用

  def addByValue(a: Int, b: Int) = a + b    //传值调用

}

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：

//传名调用：将未计算的参数表达式直接应用到函数内部

addByName(2, 2 + 2)

->2 + (2 + 2)

->2 + 4

->6

//传值调用：先计算参数表达式的值，再应用到函数内部

  addByValue(2, 2 + 2)

->addByValue(2, 4)

->2 + 4

->6

### 5.3.2 举例：酒鬼喝酒

假设有一只酒鬼，他最初有十元钱，每天喝酒都会花掉一元钱。设他有一个技能是数自己的钱，返回每天他口袋里钱的最新数目。

package com.doggie

object Drunkard {

  //最开始拥有的软妹币

  var money = 10

  //每天喝掉一个软妹币

  def drink: Unit = {

    money -= 1

  }

  //数钱时要算上被喝掉的软妹币

  def count: Int = {

    drink

    money

  }

  //每天都数钱

  def printByName(x: => Int): Unit = {

    for(i <- 0 until 5)

      println("每天算一算，酒鬼还剩" + x + "块钱！")

  }

  //第一天数一下记墙上，以后每天看墙上的余额

  def printByValue(x: Int): Unit = {

    for(i <- 0 until 5)

      println("只算第一天，酒鬼还剩" + x + "块钱！")

  }

  def main(args: Array[String]) = {

    printByName(count)

    printByValue(count)

  }

}

我们使用成员变量money来表示酒鬼剩下的软妹币数量，每次发动drink技能就消耗一枚软妹币，在count中要计算因为drink消费掉的钱。我们定义了两种计算方式，printByName是传名调用，printByValue是传值调用。查看程序输出：

每天算一算，酒鬼还剩9块钱！

每天算一算，酒鬼还剩8块钱！

每天算一算，酒鬼还剩7块钱！

每天算一算，酒鬼还剩6块钱！

每天算一算，酒鬼还剩5块钱！

只算第一天，酒鬼还剩4块钱！

只算第一天，酒鬼还剩4块钱！

只算第一天，酒鬼还剩4块钱！

只算第一天，酒鬼还剩4块钱！

只算第一天，酒鬼还剩4块钱！

可以看到，酒鬼最初5天每天都会数一下口袋里的软妹币(call-by-name)，得到了每天喝酒花钱之后剩下的软妹币数量，钱越来越少，他深感不能再这么堕落下去了。于是想出了一个聪明的方法，在第六天他将口袋里还剩下的余额数写在了墙上，以后每天看一下墙上的数字(call-by-value)，就知道自己还剩多少钱了-\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

真是个聪明的酒鬼。

### 5.3.3 传名调用 vs 传值调用

传值调用在进入函数体之前就对参数表达式进行了计算，这避免了函数内部多次使用参数时重复计算其值，在一定程度上提高了效率。

但是传名调用的一个优势在于，如果参数在函数体内部没有被使用到，那么它就不用计算参数表达式的值了。在这种情况下，传名调用的效率会高一点。例子：

package com.doggie

object WhyAlwaysMe {

  var flag: Boolean = true

  def useOrNotUse(x: Int, y: => Int) = {

    flag match{

      case true => x

      case false => x + y

    }

  }

  def main(args: Array[String]) =

  {

    println(useOrNotUse(1, 2))

    flag = false

    println(useOrNotUse(1, 2))

  }

}

## 5.4 指定函数参数名

一般情况下函数调用参数，就按照函数定义时的参数顺序一个个传递。但是我们也可以通过指定函数参数名，并且不需要按照顺序向函数传递参数，实例如下：

object Test {

def main(args: Array[String]) {

printInt(b=5, a=7);

}

def printInt( a:Int, b:Int ) = {

println("Value of a : " + a );

println("Value of b : " + b );

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Value of a : 7

Value of b : 5

## 5.5 可变参数

Scala 允许你指明函数的最后一个参数可以是重复的，即我们不需要指定函数参数的个数，可以向函数传入可变长度参数列表。

Scala 通过在参数的类型之后放一个星号来设置可变参数(可重复的参数)。例如：

object Test {

def main(args: Array[String]) {

printStrings("Runoob", "Scala", "Python"); //经验证，可以不传参

}

def printStrings( args:String\* ) = { //可变参数

var i : Int = 0;

for( arg <- args ){

println("Arg value[" + i + "] = " + arg );

i = i + 1;

}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Arg value[0] = Runoob

Arg value[1] = Scala

Arg value[2] = Python

## 5.6 默认参数值

Scala 可以为函数参数指定默认参数值，使用了默认参数，你在调用函数的过程中可以不需要传递参数，这时函数就会调用它的默认参数值，如果传递了参数，则传递值会取代默认值。实例如下：

object Test {

def main(args: Array[String]) {

println( "返回值 : " + addInt() );

}

def addInt( a:Int=5, b:Int=7 ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

返回值 : 12

## 5.7 递归函数

Scala 同样支持递归函数。

递归函数意味着函数可以调用它本身。

以上实例使用递归函数来计算阶乘：

object Test {

def main(args: Array[String]) {

for (i <- 1 to 10)

println(i + " 的阶乘为: = " + factorial(i) )

}

def factorial(n: BigInt): BigInt = {

if (n <= 1)

1

else

n \* factorial(n - 1)

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

1 的阶乘为: = 1

2 的阶乘为: = 2

3 的阶乘为: = 6

4 的阶乘为: = 24

5 的阶乘为: = 120

6 的阶乘为: = 720

7 的阶乘为: = 5040

8 的阶乘为: = 40320

9 的阶乘为: = 362880

10 的阶乘为: = 3628800

## 5.8 高阶函数

高阶函数（Higher-Order Function）就是操作其他函数的函数。

Scala 中允许使用高阶函数, 高阶函数可以使用其他函数作为参数，或者使用函数作为输出结果。

以下实例中，apply() 函数使用了另外一个函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v：

object Test {

def main(args: Array[String]) {

println( apply( layout, 10) )

}

// 函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v

def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = "[" + x.toString() + "]"

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

[10]

## 5.9 内嵌函数

我们可以在 Scala 函数内定义函数，定义在函数内的函数称之为局部函数。

以下实例我们实现阶乘运算，并使用内嵌函数：

object Test {

def main(args: Array[String]) {

println( factorial(0) )

println( factorial(1) )

println( factorial(2) )

println( factorial(3) )

}

def factorial(i: Int): Int = {

def fact(i: Int, accumulator: Int): Int = {

if (i <= 1)

accumulator

else

fact(i - 1, i \* accumulator)

}

fact(i, 1)

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

1

1

2

6

## 5.10 偏函数

偏函数(Partial Function)，是一个数学概念它不是"函数"的一种, 它跟函数是平行的概念。  
Scala中的Partia Function是一个Trait，其的类型为PartialFunction[A,B]，其中接收一个类型为A的参数，返回一个类型为B的结果。

举个例子：

scala> val pf:PartialFunction[Int,String] = {

| case 1=>"One"

| case 2=>"Two"

| case 3=>"Three"

| case \_=>"Other"

| }

pf: PartialFunction[Int,String] = <function1>

scala> pf(1)

res0: String = One

scala> pf(2)

res1: String = Two

scala> pf(3)

res2: String = Three

scala> pf(4)

res3: String = Other

偏函数内部有一些方法，比如isDefinedAt、OrElse、 andThen、applyOrElse等等。

### 5.10.1 isDefinedAt

isDefinedAt : 这个函数的作用是判断传入来的参数是否在这个偏函数所处理的范围内。  
刚才定义的pf来尝试使用isDefinedAt()，只要是数字都是正确的，因为有case \_=>"Other"这一句。如果换成其他类型就会报错。

scala> pf.isDefinedAt(1)

res4: Boolean = true

scala> pf.isDefinedAt(2)

res5: Boolean = true

scala> pf.isDefinedAt("1")

<console>:13: error: type mismatch;

found : String("1")

required: Int

pf.isDefinedAt("1")

^

scala> pf.isDefinedAt(100)

res7: Boolean = true

那我们再定义一个PartialFunction

scala> val anotherPF:PartialFunction[Int,String] = {

| case 1=>"One"

| case 2=>"Two"

| case 3=>"Three"

| }

anotherPF: PartialFunction[Int,String] = <function1>

scala> anotherPF.isDefinedAt(1)

res8: Boolean = true

scala> anotherPF.isDefinedAt(2)

res9: Boolean = true

scala> anotherPF.isDefinedAt(3)

res10: Boolean = true

scala> anotherPF.isDefinedAt(4)

res11: Boolean = false

去掉了原先的最后一句，再执行anotherPF.isDefinedAt(4)会返回false。

### 5.10.2 orElse

orElse : 将多个偏函数组合起来使用，效果类似case语句。

scala> val onePF:PartialFunction[Int,String] = {

| case 1=>"One"

| }

onePF: PartialFunction[Int,String] = <function1>

scala> val twoPF:PartialFunction[Int,String] = {

| case 2=>"Two"

| }

twoPF: PartialFunction[Int,String] = <function1>

scala> val threePF:PartialFunction[Int,String] = {

| case 3=>"Three"

| }

threePF: PartialFunction[Int,String] = <function1>

scala> val otherPF:PartialFunction[Int,String] = {

| case \_=>"Other"

| }

otherPF: PartialFunction[Int,String] = <function1>

scala> val newPF = onePF orElse twoPF orElse threePF orElse otherPF

newPF: PartialFunction[Int,String] = <function1>

scala> newPF(1)

res0: String = One

scala> newPF(2)

res1: String = Two

scala> newPF(3)

res2: String = Three

scala> newPF(4)

res3: String = Other

这样，newPF跟原先的pf效果是一样的。

### 5.10.3 andThen

andThen: 相当于方法的连续调用，比如g(f(x))。

scala> val pf1:PartialFunction[Int,String] = {

| case i if i == 1 => "One"

| }

pf1: PartialFunction[Int,String] = <function1>

scala> val pf2:PartialFunction[String,String] = {

| case str if str eq "One" => "The num is 1"

| }

pf2: PartialFunction[String,String] = <function1>

scala> val num = pf1 andThen pf2

num: PartialFunction[Int,String] = <function1>

scala> num(1)

res4: String = The num is 1

pf1的结果返回类型必须和pf2的参数传入类型必须一致，否则会报错。

### 5.10.4 applyOrElse

applyOrElse：它接收2个参数，第一个是调用的参数，第二个是个回调函数。如果第一个调用的参数匹配，返回匹配的值，否则调用回调函数。

scala> onePF.applyOrElse(1,{num:Int=>"two"})

res5: String = One

scala> onePF.applyOrElse(2,{num:Int=>"two"})

res6: String = two

在这个例子中，第一次onePF匹配了1成功则返回的是"One"字符串。第二次onePF匹配2失败则触发回调函数，返回的是"Two"字符串。

## 5.11 偏应用函数

偏应用函数(Partial Applied Function)也叫部分应用函数，跟偏函数(Partial Function)从英文名来看只有一字之差，但他们二者之间却有天壤之别。

部分应用函数, 是指一个函数有n个参数, 而我们为其提供少于n个参数, 那就得到了一个部分应用函数。

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

add: (x: Int, y: Int, z: Int)Int

scala> def addX = add(1,\_:Int,\_:Int) // x 已知

addX: (Int, Int) => Int

scala> addX(2,3)

res1: Int = 6

scala> addX(3,4)

res2: Int = 8

scala> def addXAndY = add(10,100,\_:Int) // x 和 y 已知

addXAndY: Int => Int

scala> addXAndY(1)

res3: Int = 111

scala> def addZ = add(\_:Int,\_:Int,10) // z 已知

addZ: (Int, Int) => Int

scala> addZ(1,2)

res4: Int = 13

## 5.12 函数柯里化

柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

### 5.12.1 实例

首先我们定义一个函数:

def add(x:Int,y:Int)=x+y

那么我们应用的时候，应该是这样用：add(1,2)

现在我们把这个函数变一下形：

def add(x:Int)(y:Int) = x + y

那么我们应用的时候，应该是这样用：add(1)(2),最后结果都一样是3，这种方式（过程）就叫柯里化。

### 5.12.2 实现过程

add(1)(2) 实际上是依次调用两个普通函数（非柯里化函数），第一次调用使用一个参数 x，返回一个函数类型的值，第二次使用参数y调用这个函数类型的值。

实质上最先演变成这样一个方法：

def add(x:Int)=(y:Int)=>x+y

那么这个函数是什么意思呢？ 接收一个x为参数，返回一个匿名函数，该匿名函数的定义是：接收一个Int型参数y，函数体为x+y。现在我们来对这个方法进行调用。

val result = add(1)

返回一个result，那result的值应该是一个匿名函数：(y:Int)=>1+y

所以为了得到结果，我们继续调用result。

val sum = result(2)

最后打印出来的结果就是3。

### 5.12.3 完整实例

下面是一个完整实例：

object Test {

def main(args: Array[String]) {

val str1:String = "Hello, "

val str2:String = "Scala!"

println( "str1 + str2 = " + strcat(str1)(str2) )

}

def strcat(s1: String)(s2: String) = {

s1 + s2

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

str1 + str2 = Hello, Scala!

## 5.13 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

如下面这段匿名的函数：

val multiplier = (i:Int) => i \* 10

函数体内有一个变量 i，它作为函数的一个参数。如下面的另一段代码：

val multiplier = (i:Int) => i \* factor

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor不是形式参数，而是自由变量，考虑下面代码：

var factor = 3

val multiplier = (i:Int) => i \* factor

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

完整实例：

object Test {

def main(args: Array[String]) {

println( "muliplier(1) value = " + multiplier(1) )

println( "muliplier(2) value = " + multiplier(2) )

}

var factor = 3

val multiplier = (i:Int) => i \* factor

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

muliplier(1) value = 3

muliplier(2) value = 6

# 6、集合（Collection）

## 6.1 可变和不可变

scala的集合系统的区分了***可变（ mutable  ）***和***不可变（immutable ）***集合。

* mutable集合：能够更新甚至扩展空间，这意味着你能改变、增加或者删除一个集合的元素。
* immutable集合：刚好相反，不能改变。你仍然可以做一些类似的增加、删除或者更新，但是实际上（跟java的string一样）它返回了一个新的对象，这里面就是指返回了一个新的集合，而老的集合没有改变。

所有的集合类在scala.collection 包中，或者他的子包中，分为mutable，immutable以及generic。 大部分集合都有三个同名的类，每个同名类有不同的特征。

scala.collection.immutable包中元素不可变，可以保证你在任何时间访问他时元素值都是相同的；scala.collection.mutable包中的元素可变，所以你要知道它在何时何地变化了。

* *有一些集合类不在上述两个包下，如:*[*collection.IndexedSeq[T]*](http://www.scala-lang.org/api/current/scala/collection/IndexedSeq.html)*。它是*[*collection.immutable.IndexedSeq[T]*](http://www.scala-lang.org/api/current/scala/collection/immutable/IndexedSeq.html)*和*[*collection.mutable.IndexedSeq[T]*](http://www.scala-lang.org/api/current/scala/collection/mutable/IndexedSeq.html)*的父类。一般情况下会在collection包下定义接口，由mutable和immutable两个包实现。*

下图展示了scala.collection.immutable.和scala.collection.mutable.

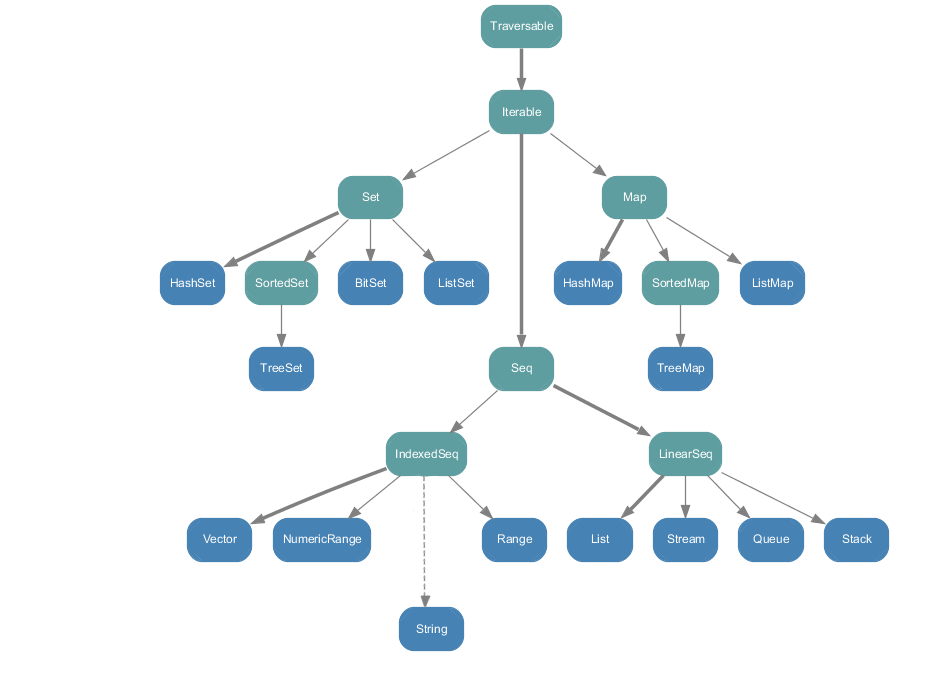


图6- 1 scala.collection.immutable.

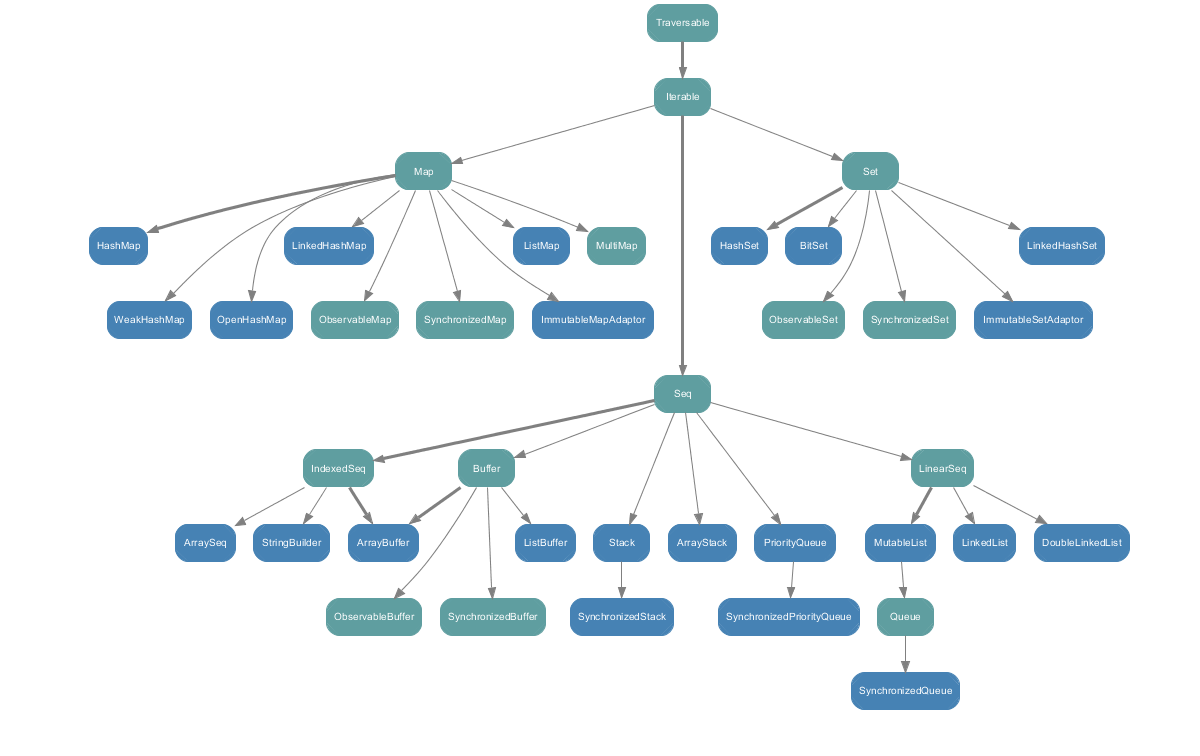


图6- 2 scala.collection.mutable.

## 6.2 List 列表

在Scala中，List都是不可变的。List要么是Nil(空列表)，要么就是由head和tail组成的递归结构。head是首元素，tail是剩下的List。所以你可以这样构建List：

val list = 1 :: Nil // 等价于：val list = List(1)

连续的两个冒号" :: "就像是胶水，将List的head和tail粘在一起。

” :: ” 发音cons，把新元素组合到现有列表的前端。

* *如果方法使用操作符来标注，如a\*b，那么左操作数是方法的调用者，可以写作a.\*(b)；除非方法名以冒号结尾，方法被右操作数调用。如1 :: Nil里，::的调用者是Nil，1是方法的传入参数，可以改写成：Nil.::(1)*

列表的元素类型 T 可以写成 List[T]。例如，以下列出了多种类型的列表：

// 字符串列表

val site: List[String] = List("Runoob", "Google", "Baidu")

// 整型列表

val nums: List[Int] = List(1, 2, 3, 4)

// 空列表

val empty: List[Nothing] = List()

// 二维列表

val dim: List[List[Int]] =

List(

List(1, 0, 0),

List(0, 1, 0),

List(0, 0, 1)

)

以上实例也可以写成如下所示：

// 字符串列表

val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

// 整型列表

val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// 空列表

val empty = Nil

// 二维列表

val dim = (1 :: (0 :: (0 :: Nil))) ::

(0 :: (1 :: (0 :: Nil))) ::

(0 :: (0 :: (1 :: Nil))) :: Nil

### 6.2.1 List基本操作

Scala列表有三个基本操作：

* head 返回列表第一个元素
* tail 返回一个列表，包含除了第一元素之外的其他元素
* isEmpty 在列表为空时返回true

对于Scala列表的任何操作都可以使用这三个基本操作来表达。实例如下：

object Test {

def main(args: Array[String]) {

val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

val nums = Nil

println( "第一网站是 : " + site.head )

println( "最后一个网站是 : " + site.tail )

println( "查看列表 site 是否为空 : " + site.isEmpty )

println( "查看 nums 是否为空 : " + nums.isEmpty )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

第一网站是 : Runoob

最后一个网站是 : List(Google, Baidu)

查看列表 site 是否为空 : false

查看 nums 是否为空 : true

### 6.2.2 连接列表

可以使用 ::: 运算符（实现叠加功能）或 List.:::() 方法或 List.concat() 方法来连接两个或多个列表。实例如下：

object Test {

def main(args: Array[String]) {

val site1 = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

val site2 = "Facebook" :: ("Taobao" :: Nil)

// 使用 ::: 运算符

var fruit = site1 ::: site2

println( "site1 ::: site2 : " + fruit )

// 使用 Set.:::() 方法

fruit = site1.:::(site2)

println( "site1.:::(site2) : " + fruit )

// 使用 concat 方法

fruit = List.concat(site1, site2)

println( "List.concat(site1, site2) : " + fruit )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

site1 ::: site2 : List(Runoob, Google, Baidu, Facebook, Taobao)

site1.:::(site2) : List(Facebook, Taobao, Runoob, Google, Baidu)

List.concat(site1, site2) : List(Runoob, Google, Baidu, Facebook, Taobao)

：：与：：：的区别：

val a = List(1, 2)

val b = List(3, 4)

val c = a::b //结果为：(List(1, 2), 3, 4)

val d = a:::b //结果为：(1, 2, 3, 4)

### 6.2.3 List.fill()

我们可以使用 List.fill() 方法来创建一个指定重复数量的元素列表：

object Test {

def main(args: Array[String]) {

val site = List.fill(3)("Runoob") // 重复 Runoob 3次

println( "site : " + site )

val num = List.fill(10)(2) // 重复元素 2, 10 次

println( "num : " + num )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

site : List(Runoob, Runoob, Runoob)

num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

### 6.2.4 List.tabulate()

List.tabulate() 方法是通过给定的函数来创建列表。

方法的第一个参数为元素的数量，可以是二维的，第二个参数为指定的函数，我们通过指定的函数计算结果并返回值插入到列表中，起始值为 0，实例如下：

object Test {

def main(args: Array[String]) {

// 通过给定的函数创建 6 个元素

val squares = List.tabulate(6)(n => n \* n)

println( "一维 : " + squares )

// 创建二维列表

val mul = List.tabulate( 4,5 )( \_ \* \_ )

println( "多维 : " + mul )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

一维 : List(0, 1, 4, 9, 16, 25)

多维 : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12))

### 6.2.5 其他常用操作

val list = List(1, 3, 2)

//获取第1个元素

list.headOption.getOrElse(0) // 1

//查找

list.find(\_ % 2 == 0).getOrElse(0) // 2

//过滤

list.filter(\_ % 2 == 1) // List(1, 3)

//排序 list.sorted // List(1, 2, 3)

//最小值/最大值/求和

list.min // 1

list.max // 3

list.sum // 6

//转化成字符串 list.mkString(",") // "1, 3, 2"

Scala提供的List基本可以实现SQL查询的所有功能，这也是Spark为什么基于Scala开发的原因。更多功能可以参考[官方文档](http://www.scala-lang.org/api/current/#scala.collection.immutable.List)。

* *在Scala中默认的集合类例如List，Set，Map，Tuple等都是不可变的，所以调用其修改方法会返回一个新的实例。如果要使用可变集合，应使用scala.collection.mutable包下相应的类。不可变类型在编写并发代码时很有用。*

## 6.3 Set

Scala Set(集合)是没有重复的对象集合，所有的元素都是唯一的。

Scala 集合分为可变的和不可变的集合。

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 scala.collection.mutable.Set 包。

默认引用 scala.collection.immutable.Set，不可变集合实例如下：

val set = Set(1,2,3)

println(set.getClass.getName) //

println(set.exists(\_ % 2 == 0)) //true

println(set.drop(1)) //Set(2,3)

如果需要使用可变集合需要引入 scala.collection.mutable.Set：

import scala.collection.mutable.Set // 可以在任何地方引入 可变集合

val mutableSet = Set(1,2,3)

println(mutableSet.getClass.getName) // scala.collection.mutable.HashSet

mutableSet.add(4)

mutableSet.remove(1)

mutableSet += 5

mutableSet -= 2

println(mutableSet) // Set(5, 3, 4)

val another = mutableSet.toSet

println(another.getClass.getName) // scala.collection.immutable.Set

* *注意：* *虽然可变Set和不可变Set都有添加或删除元素的操作，但是有一个非常大的差别。对不可变Set进行操作，会产生一个新的set，原来的set并没有改变，这与List一样。 而对可变Set进行操作，改变的是该Set本身，与ListBuffer类似。*

### 6.3.1 集合基本操作

Scala集合有三个基本操作：

* head 返回集合第一个元素
* tail 返回一个集合，包含除了第一元素之外的其他元素
* isEmpty 在集合为空时返回true

对于Scala集合的任何操作都可以使用这三个基本操作来表达。实例如下:

object Test {

def main(args: Array[String]) {

val site = Set("Runoob", "Google", "Baidu")

val nums: Set[Int] = Set()

println( "第一网站是 : " + site.head )

println( "最后一个网站是 : " + site.tail )

println( "查看列表 site 是否为空 : " + site.isEmpty )

println( "查看 nums 是否为空 : " + nums.isEmpty )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

第一网站是 : Runoob

最后一个网站是 : Set(Google, Baidu)

查看列表 site 是否为空 : false

查看 nums 是否为空 : true

### 6.3.2 连接集合

你可以使用 ++ 运算符或 Set.++() 方法来连接两个集合。如果元素有重复的就会移除重复的元素。实例如下：

object Test {

def main(args: Array[String]) {

val site1 = Set("Runoob", "Google", "Baidu")

val site2 = Set("Faceboook", "Taobao")

// ++ 作为运算符使用

var site = site1 ++ site2

println( "site1 ++ site2 : " + site )

// ++ 作为方法使用

site = site1.++(site2)

println( "site1.++(site2) : " + site )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

site1 ++ site2 : Set(Faceboook, Taobao, Google, Baidu, Runoob)

site1.++(site2) : Set(Faceboook, Taobao, Google, Baidu, Runoob)

### 6.3.3 查找集合中最大与最小元素

你可以使用 Set.min 方法来查找集合中的最小元素，使用 Set.max 方法查找集合中的最大元素。实例如下：

object Test {

def main(args: Array[String]) {

val num = Set(5,6,9,20,30,45)

// 查找集合中最大与最小元素

println( "Set(5,6,9,20,30,45) 集合中的最小元素是 : " + num.min )

println( "Set(5,6,9,20,30,45) 集合中的最大元素是 : " + num.max )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

Set(5,6,9,20,30,45) 集合中的最小元素是 : 5

Set(5,6,9,20,30,45) 集合中的最大元素是 : 45

### 6.3.4 交集

你可以使用 Set.& 方法或 Set.intersect 方法来查看两个集合的交集元素。实例如下：

object Test {

def main(args: Array[String]) {

val num1 = Set(5,6,9,20,30,45)

val num2 = Set(50,60,9,20,35,55)

// 交集

println( "num1.&(num2) : " + num1.&(num2) )

println( "num1.intersect(num2) : " + num1.intersect(num2) )

}

}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala

num1.&(num2) : Set(20, 9)

num1.intersect(num2) : Set(20, 9)

## 6.4 Map

Map(映射)是一种可迭代的键值对（key/value）结构（二元组），所有的值都可以通过键来获取。Map 中的键都是唯一的。Map 也叫哈希表（Hash tables）。

Map 有两种类型，可变与不可变，区别在于可变对象可以修改它，而不可变对象不可以。默认情况下 Scala 使用不可变 Map。如果你需要使用可变集合，你需要显式的引入 import scala.collection.mutable.Map 类。

在 Scala 中 你可以同时使用可变与不可变 Map，不可变的直接使用 Map，可变的使用 mutable.Map。以下实例演示了不可变 Map 的应用：

// 空哈希表，键为字符串，值为整型

var A:Map[Char,Int] = Map()

// 键值对演示。Map其实是二元组的集合，定义 Map 时，需要为键值对定义类型。

val map = Map("a" -> 1, "b" -> 2)

// 也可以这样构建Map

val map = Map(("a", 1), ("b", 2))

"->"其实是String类型上的方法，返回一个二元组：

"a" -> 1 //等价于: ("a", 1)

常用操作：

val map = Map("a" -> 1, "b" -> 2)

//读取

map("a") // 1

//写入或添加键值

map("a") = 0

//删除键值

map - "a" // Map(b -> 2)

如果需要添加 key-value 对，可以使用 + 号，如下所示：

A += ('I' -> 1)

A += ('J' -> 5)

A += ('K' -> 10)

A += ('L' -> 100)

### 6.4.1 Map 基本操作

Scala Map 有三个基本操作：

|  |  |
| --- | --- |
| 方法 | 描述 |
| keys | 返回 Map 所有的键(key) |
| values | 返回 Map 所有的值(value) |
| isEmpty | 在 Map 为空时返回true |

以下实例演示了以上三个方法的基本应用：

object Test {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000",

"azure" -> "#F0FFFF",

"peru" -> "#CD853F")

val nums: Map[Int, Int] = Map()

println( "colors 中的键为 : " + colors.keys )

println( "colors 中的值为 : " + colors.values )

println( "检测 colors 是否为空 : " + colors.isEmpty )

println( "检测 nums 是否为空 : " + nums.isEmpty )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

colors 中的键为 : Set(red, azure, peru)

colors 中的值为 : MapLike(#FF0000, #F0FFFF, #CD853F)

检测 colors 是否为空 : false

检测 nums 是否为空 : true

### 6.4.2 Map 合并

你可以使用 ++ 运算符或 Map.++() 方法来连接两个 Map，Map 合并时会移除重复的 key。以下演示了两个 Map 合并的实例:

object Test {

def main(args: Array[String]) {

val colors1 = Map("red" -> "#FF0000",

"azure" -> "#F0FFFF",

"peru" -> "#CD853F")

val colors2 = Map("blue" -> "#0033FF",

"yellow" -> "#FFFF00",

"red" -> "#FF0000")

// ++ 作为运算符

var colors = colors1 ++ colors2

println( "colors1 ++ colors2 : " + colors )

// ++ 作为方法

colors = colors1.++(colors2)

println( "colors1.++(colors2)) : " + colors )

}

}

执行以上代码，输出结果为：（后盖前）

$ scalac Test.scala

$ scala Test

colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)

colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)

### 6.4.3 输出 Map 的 keys 和 values

以下通过 foreach 循环输出 Map 中的 keys 和 values：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "http://www.runoob.com",

"baidu" -> "http://www.baidu.com",

"taobao" -> "http://www.taobao.com")

sites.keys.foreach{ i =>

print( "Key = " + i )

println(" Value = " + sites(i) )}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Key = runoob Value = http://www.runoob.com

Key = baidu Value = http://www.baidu.com

Key = taobao Value = http://www.taobao.com

### 6.4.4 查看 Map 中是否存在指定的 Key

你可以使用 Map.contains 方法来查看 Map 中是否存在指定的 Key。实例如下：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "http://www.runoob.com",

"baidu" -> "http://www.baidu.com",

"taobao" -> "http://www.taobao.com")

if( sites.contains( "runoob" )){

println("runoob 键存在，对应的值为 :" + sites("runoob"))

}else{

println("runoob 键不存在")

}

if( sites.contains( "baidu" )){

println("baidu 键存在，对应的值为 :" + sites("baidu"))

}else{

println("baidu 键不存在")

}

if( sites.contains( "google" )){

println("google 键存在，对应的值为 :" + sites("google"))

}else{

println("google 键不存在")

}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

runoob 键存在，对应的值为 :http://www.runoob.com

baidu 键存在，对应的值为 :http://www.baidu.com

google 键不存在

## 6.5 Tuple 元组

Tuple(元组)可以容纳不同类型的元素（例如同时拥有String和Int），最简单的形态是二元组，即由两个元素构成的Tuple, 可以使用\_1, \_2等方法访问其元素（基于1的索引；数组和List是基于0的索引）：

val t = ("a", 1) // 等价于：val t: Tuple2[String, Int] = ("a", 1)

t.\_1 // "a"

t.\_2 // 1

也可以使用模式匹配利用Tuple同时初始化一组变量：

val t = ("a", 1)

val (v1, v2) = t

v1 // "a"

v2 // 1

### 6.5.1 迭代元组

你可以使用 Tuple.productIterator() 方法来迭代输出元组的所有元素：

object Test {

def main(args: Array[String]) {

val t = (4,3,2,1)

t.productIterator.foreach{ i =>println("Value = " + i )}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Value = 4

Value = 3

Value = 2

Value = 1

### 6.5.2 元组转为字符串

你可以使用 Tuple.toString() 方法将元组的所有元素组合成一个字符串，实例如下：

object Test {

def main(args: Array[String]) {

val t = new Tuple3(1, "hello", Console)

println("连接后的字符串为: " + t.toString() )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

连接后的字符串为: (1,hello,scala.Console$@4dd8dc3)

### 6.5.3 元素交换

你可以使用 Tuple.swap 方法来交换元组的元素。如下实例：

object Test {

def main(args: Array[String]) {

val t = new Tuple2("www.google.com", "www.runoob.com")

println("交换后的元组: " + t.swap )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

交换后的元组: (www.runoob.com,www.google.com)

## 6.6 Option

Scala用Option类型表示一个值是可选的（有值或无值），用来避免Java的NullPointerException。它有两个子类：Some和None。Some类型表示值存在，None类型则表示值不存在。

常用操作：

val opt: Option[String] = Some("hello")

//判断是否为None

opt.isEmpty // false

//用 getOrElse() 方法来获取元组中存在的元素或者使用其默认的值。如果为None（元组中无元素），则返回默认值"default"，否则返回opt持有的值

opt.getOrElse("default")

//如果为None则返回"DEFAULT"，否则将字符转为大写

opt.fold("DEFAULT"){ value => value.toUpperCase } // "HELLO"

//功能同上

opt match {

case Some(v) => v.toUpperCase

case None => "DEFAULT"

}

## 6.7 map和flatMap方法

Scala中所有的容器类型(例如Option, Either, Future, Set, ...)都内置了map和flatMap这两个方法，这里我们只在List上演示map和flatMap的基本用法。

* *除了map和flatMap，Scala的容器类型上还有很多类似的方法，例如filter, find, sortBy等等，详见"Programming in Scala, 3nd Edition"。）*

map和flatMap是两个高阶函数，所谓高阶函数就是接受函数作为参数的函数。这两个方法各自接受一个一元函数（即只有一个参数的函数，类型为：(A) => B），利用这个一元函数，你可以对数据流中的每一个元素进行一些操作或转换，最终得到一个全新的数据流。

map方法接受的一元函数类型为：(A) => B

List(1, 2, 3).map((i: Int) => { i + 1 }) // List(2, 3, 4)

也可以简写如下两种形式：

List(1, 2, 3).map(i => i + 1 )

List(1, 2, 3).map(\_ + 1 )

* 你可以把第2种形式中的下划线理解成每个元素的占位符，其实这只是编译器的语法糖，编译后的结果和前两种写法相同。使用这个语法糖的前提是下划线"\_"在函数体内只能出现一次。

在上面的例子里，map方法接受的一元函数类型是：(Int) => Int，元素的类型没有发生改变，我们可以尝试改变元素类型：

List(1, 2, 3).map(i => i.toString \* i) // List(1, 22, 333)

这次传入的一元函数类型是: (Int) => String，将原List从List[Int]类型转换成了List[String]类型，完成一次数据流类型转换。

flatMap方法接受的一元函数类型为：(A) => List[B]，我们发现该一元函数返回的类型也是一个List，flatMap方法会自动将由每个元素A转换成的小List[B]展平成一个大的List[B]，这也是flatMap中的"flat"所要表达的意思：

List(1, 2, 3).flatMap(i => List(i, i)) // List(1, 1, 2, 2, 3, 3)

## 6.8 Java与Scala集合相互转换

和Scala一样，Java同样提供了丰富的容器库，Scala和Java容器库有很多相似点，例如，他们都包含迭代器、可迭代结构、集合、 映射和序列。但是他们有一个重要的区别。Scala的容器库特别强调不可变性，因此提供了大量的新方法将一个容器变换成一个新的容器。

某些时候，你需要将一种容器类型转换成另外一种类型。例如，你可能想要像访问Scala容器一样访问某个Java容器，或者你可能想将一个Scala容器像Java容器一样传递给某个Java方法。

### 6.8.2 双向类型转换

Scala提供了大量的方法来隐式转换所有主要的Java和Scala容器类型。其中提供了如下的双向类型转换：

* scala.collection.Iterable <=> [Java](http://lib.csdn.net/base/17).lang.Iterable
* scala.collection.Iterable <=> java.util.Collection
* scala.collection.Iterator <=> java.util.{ Iterator, Enumeration }
* scala.collection.mutable.Buffer <=> java.util.List
* scala.collection.mutable.Set <=> java.util.Set
* scala.collection.mutable.Map <=> java.util.{ Map, Dictionary }
* scala.collection.mutable.ConcurrentMap <=> java.util.concurrent.ConcurrentMap
* *当然这些转换均需要在scala文件中引入：scala.collection.JavaConversions.\_*

Import之后，就可以在Scala容器和与之对应的Java容器之间进行隐式转换了。

**scala>** import collection.mutable.\_

import collection.mutable.\_

//将scala的Buffer转换为java的List

**scala>** val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3)

jul: java.util.List[Int] = [1, 2, 3]

//将Java List形式的jul转换为scala的Seq

**scala>** val buf: Seq[Int] = jul

buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

//将scala的HashMap转换为java的Map

**scala>** val m: java.util.Map[String, Int] = HashMap("abc" -> 1, "hello" -> 2)

m: java.util.Map[String, Int] = {hello=2, abc=1}

在Scala内部，这些转换是通过一系列“包装”对象完成的，这些对象会将相应的方法调用转发至底层的容器对象。所以容器不会在Java和Scala之间拷贝来拷贝去。一个值得注意的特性是，如果你将一个Java容器转换成其对应的Scala容器，然后再将其转换回同样的Java容器，最终得到的是一个和一开始完全相同的容器对象（译注：这里的相同意味着这两个对象实际上是指向同一片内存区域的引用，容器转换过程中没有任何的拷贝发生）。

### 6.8.2 单向类型转换

还有一些Scala容器类型可以转换成对应的Java类型，但是并没有将相应的Java类型转换成Scala类型的能力，它们是：

* scala.collection.Seq         => java.util.List
* scala.collection.mutable.Seq => java.util.List
* scala.collection.Set         => java.util.Set
* scala.collection.Map         => java.util.Map
* java.util.Properties         => scala.collection.mutable.Map[String, String]

因为Java并未区分可变容器不可变容器类型，所以，虽然能将scala.immutable.List转换成java.util.List，但所有的修改操作都会抛出“UnsupportedOperationException”。参见下例：

**scala>** jul = List(1, 2, 3)

jul: java.util.List[Int] = [1, 2, 3]

**scala>** jul.add(7)

java.lang.UnsupportedOperationException

at java.util.AbstractList.add(AbstractList.java:131)

# 7、类和对象

## 7.1 类、字段和方法

类是对象的蓝图。一旦定义了类，就可以用关键字new根据类的蓝图创建对象。比方说，有如下类的定义：

class ChecksumAccumulator {

//此处为类定义

}

就能创建Point对象：

new ChecksumAccumulator

类定义里，可以放置字段和方法，这些被笼统地称为成员（member）。字段，不管使用val还是用var定义的，都是指向对象的变量。方法，用def定义，包含了可执行的代码。例如，定义了ChecksumAccumulator类并内置名为sum的var字段：

class ChecksumAccumulator{

　　var sum = 0;

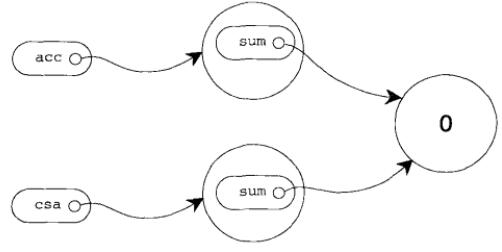
}

然后实例化两次：

val acc = new ChecksumAccumulator

val csa = new ChecksumAccumulator

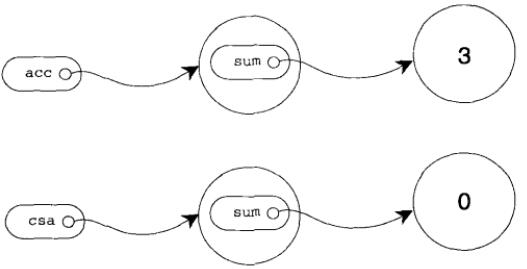
那么内存里对象的状态映像看上去大概是这样：



尽管acc,csa都是val的，但仍可以修改它们所指向的对象sum（因为sum是var的），因此之后可以重新赋给不同的Int值：

acc.sum = 3

现在，映像会变成：



注意图中同时存在两个sum变量，一个在acc指向的对象里，另一个在csa指向的对象里。

另外还需注意的是，尽管acc是val，但仍可以修改acc指向的对象。val类型对象对acc和csa的限制仅在于不可以把它们再次赋值为其他对象。例如，下面的尝试将会失败：

//编译不过，因为acc是val

acc = new ChecksumAccumulator

因此，我们可以得出结论：acc将始终指向初始化时的ChecksumAccumulator对象，但是对象里包含的字段可以随时改动。

传递给方法的任何参数都可以在方法内部使用。Scala里方法参数的一个重要特征是它们都是val不是var。如果你想在方法里面给参数重新赋值，结果是编译失败：

def add(b: Byte): Unit = {

b = 1 //编译不过，因为b是val

sum += b

}

## 7.2 Singleton 单例对象

在 Scala 中不能定义静态成员，而是代之以定义单例对象（singleton object）。除了用object关键字替换了class关键字以外，单例对象的定义看上去与类定义一致。

// 文件ChecksumAccumulator.scala

**import** scala.collection.mutable.Map

//单例对象ChecksumAccumulator

**object** ChecksumAccumulator {

**private val** cache = Map[String, Int]()

**def** calculate(s: String): Int =

**if** (cache.contains(s))

cache(s)

**else** {

//关键字new只用来实例化类，所以这里创造的新对象是ChecksumAccumulator类

的一个实例，而不是同名的单例对象

val acc = **new** ChecksumAccumulator

**for** (c <- s)

acc.add(c.toByte)

**val** cs = acc.checksum()

cache += (s -> cs)

cs

}

}

表中的单例对象叫做ChecksumAccumulator，与前一个例子里的类同名。**当单例对象与某个类共享同一个名称时，它就被称为是这个类的伴生对象。**类和它的伴生对象必须定义在一个源文件里，类被称为是这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。

ChecksumAccumulator单例对象包含了方法calucate以及私有字段cache。对于Java程序员来说，可以把单例对象当作是Java中可能会用到的静态方法工具类。也可以用类似的语法做方法调用：**单例对象名.方法名**。例如：

ChecksumAccumulator. calucate(“Every value is an object.”)

类和单例对象的差别是：

1. 单例对象不能带参数，而类可以。因为单例对象不是用new关键字实例初始化的，所以没机会传递给它实例化参数。
2. 单例对象在第一次被访问的时候才会被初始化。

不与伴生类共享名称的单例对象被称为独立对象。它可以用在很多地方，例如作为相关功能方法的工具类，或者定义Scala应用的入口点。

### 7.3.1 单例对象作为程序的入口点

想要编写能够独立运行的Scala程序，就必须创建有main方法（仅带一个参数Array[String]且结果类型为Unit）的单例对象。任何拥有合适签名的main方法的单例对象都可以用来作为程序的入口点。

import ChecksumAccumulator. calucate

object Summer {

def main(args: Array[String]) {

for (arg <- args)

println ( arg + ":" + calculate(arg))

}

}

### 7.3.2 伴生对象实例

/\* 文件名：Marker.scala

// 私有构造方法

class Marker private(val color:String) {

println("创建" + this)

override def toString(): String = "颜色标记："+ color

}

// 伴生对象，与类共享名字，可以访问类的私有属性和方法

object Marker{

private val markers: Map[String, Marker] = Map(

"red" -> new Marker("red"),

"blue" -> new Marker("blue"),

"green" -> new Marker("green")

)

def apply(color:String) = {

if(markers.contains(color)) markers(color) else null

}

def getMarker(color:String) = {

if(markers.contains(color)) markers(color) else null

}

def main(args: Array[String]) {

println(Marker("red"))

// 单例函数调用，省略了.(点)符号

println(Marker getMarker "blue")

}

}

执行以上代码，输出结果为：

$ scalac Marker.scala

$ scala Marker

创建颜色标记：red

创建颜色标记：blue

创建颜色标记：green

颜色标记：red

颜色标记：blue

# 8、特质（特征）

## 8.1 概念

Scala Trait(特质) 相当于 Java 的接口，与接口不同的是，它还可以定义属性和方法的实现。特质封装了方法和字段的定义，并可以通过“混入”到类中重用它们。与类的继承时每个类都只能继承唯一的超类不同，类可以混入任意多个特质。特质的定义除了使用关键字trait之外，与类定义无异，如代码8-1：

//代码8-1

trait Bird {

  def fly = println("鸟飞翔")

  def singing

}

这个特质名为Bird，它没有声明超类，因此和类一样，有个默认的超类AnyRef。它定义了一个具体方法fly，也定义了一个抽象方法singing，等待被混入的类实现。一旦特质被定义了，就可以使用extends或with关键字，把它混入到类中，scala的混入特质并不是继承它们，将在本文的后面说明。

模拟一个场景：一个机器人继承了人类，人会说话，而机器人想要继续继承鱼的游泳、和鸟的飞翔和唱歌，需要用到特质。如代码8-2：

//代码8-2

**scala>** class Person {

  def say = println("人说话")

}

trait Fish {

  def swim = println("鱼游泳")

}

trait Bird {

  def fly = println("鸟飞翔")

  def singing

}

class Rebot extends Person with Fish with Bird {

  def work = println("机器人工作")

  def singing = println("机器人唱歌")

}

defined class Person

defined trait Fish

defined trait Bird

defined class Rebot

**scala>** val rebot = new Rebot()

rebot: Rebot = Rebot@3e44f2a5<br><br>scala> rebot.work<br>机器人工作

**scala>** rebot.fly

鸟飞翔

**scala>** rebot.singing

机器人唱歌

//因为Rebot这个类混入了Fish特质，所以可以用Fish的变量去接收Rebot对象。

**scala>** val fish: Fish = rebot

fish: Fish = Rebot@3e44f2a5

**scala>** fish.swim

鱼游泳

Rebot类最先开始继承Person类，接着混入Fish特质和Bird特质，因此，机器人除了能工作，还能模拟鸟飞翔。另外，因为Rebot这个类混入了Fish特质，所以可以用Fish的变量去接收Rebot对象。特质有点类似Java的抽象类，可以声明抽象方法和具体方法，但是Java的一个类只能继承一个抽象类，而scala一个类却可以混入多个特质。

此外，特质不能带有任何类参数。代码8-3是合法的，而代码8-4则会报错

代码8-3

class Cat(color: String, age: Int)

代码8-4 //会报错

trait Cat(color: String, age: Int)

 如果想在特质里加入参数时，可如代码8-5这样做：

//代码8-5

**scala>** trait Cat {

//在特质里加入参数

val color: String

val age: Int

def cry = println("小猫喵喵叫")

def printColor = println("小猫的颜色是" + color)

def printAge = println("小猫的年龄是" + age + "岁")}

**defined trait Cat**

**scala>** val cat = new {

val color = "白色"

val age = 1

} with Cat

**cat: Cat = $anon$1@6bea52d4**

**scala>** cat.printColor

**小猫的颜色是白色**

**scala>** cat.printAge

**小猫的年龄是1岁**

**scala>** cat.cry

**小猫喵喵叫**

## 8.2 瘦接口对阵胖接口

在Scala中，在特质中添加具体方法使得胖瘦对阵的权衡大大倾向于胖接口。只要在特质中实现一次方法，而不再需要在每个混入特质的方法中重新实现它。

要使用特质丰富接口，只要简单的定义一个具有少量抽象方法的特质，特质接口的瘦部门，和潜在的大量具体方法，所有的都实现在抽象方法之上，然后就可以把丰富了的特质混入到类中，实现接口的瘦部分，并最终获得具有全部胖接口内容的类。

## 8.3 特质用来做可堆叠的改变

特质的第一个主要用法：把瘦接口转变成胖接口。

特质的第二个主要用法：为类提供可堆叠的改变。

举个例子，思考一下对一个整数队列堆叠改动。队列有两种操作：put，把整数放入队列；和get，从队列取出整数，队列是先进先出的，因此get应该依整数进入队列时的顺序把它们取出来。

假设有一个类实现了这样的队列，你可以定义特质执行如下的改动：

* Doubling：把所有放入到队列的数字加倍。
* Incrementing：把所有放入到队列的数字增值
* Filtering：从队列中过滤掉负数

这三种特质代表了改动，因为它们改变了原始队列的行为而并非定义了全新的队列。这三种同样也是可堆叠的。你可以任选三者中的若干，把它们混入类中，并获得你所需改动的新类。

代码8-8为抽象的IntQueue类，IntQueue有一个put方法把整数添加到队列中，和一个get方法移除开头的整数并返回它们，length返回队列的长度。

//代码8-8

abstract class IntQueue {

def get(): Int

def put(x: Int)

def length(): Int

}

代码8-9的BasicIntQueue类是抽象类IntQueue的实现：

//代码8-9

Import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {

//有一个私有字段buf持有数组缓存

private val buf = new ArrayBuffer[Int]

// get从缓存的一端移除成员，put从另一端加入元素

def get() = buf.remove(0)

def put(x: Int) = { buf += x }

def length() = buf.length

}

代码8-10是实现运行时的样子：

//代码8-10

object TEST {  
 def main(args: Array[String]) = {  
 val queue = new BasicIntQueue  
 queue.put(1)  
 queue.put(2)  
 queue.put(3)  
 queue.length()  
 println(queue.get())  
 println(queue.get())  
 println(queue.get())  
 }  
}

运行结果：

1

2

3

现在用特质改变它的行为：代码8-11展示了在把整数放入队列时对其加倍，Doubling做了两件事情：第1件是它定义了超类IntQueue，这个定义意味着它只能混入扩展了IntQueue的类中。因此可以把Doubling混入到BasicIntQueue，但不能混入Rational。

//代码8-11

trait Doubling extends IntQueue {

abstract override def put(x: Int) = super.put(2 \* x)

}

第2件事情是特质在声明为抽象的方法中有一个super的调用。这种调用对于普通的类来说是非法的，执行时必然失败。然而对于特质来说，则能调用成功。因为特质的super调用是动态绑定的。**特质Doubling的super调用将直到被混入另一个特质或类之后，有了具体的方法定义时才工作。**

这种安排对于实现可堆叠改动的特质来说是常常要用到的，为了告诉编译器你的目的，你必须在这种方法打上abstract override的标志。这种标识符的组合仅在特质成员的定义中被认可，在类中不行，它意味着特质必须混入某个具有期待方法的具体定义的类中。

代码8-12是使用这个特质的效果：

代码8-12

**scala>** class MyQueue extends BasicIntQueue with Doubling

**defined class MyQueue**

**scala>** val queue = new MyQueue

**queue: MyQueue = MyQueue@59221b97**

**scala>** queue.put(10)

**scala>** queue.get()

**res: Int = 20**

在第一行中，我们定义了类MyQueue，它扩展了BasicIntQueue并混入了Doubling。把10放入队列中，由于混入的Doubling重写了方法put，因此10会被加倍。当从队列中取出整数时，它变成了20。

为了看到如何堆叠改动，我们再加入两个特质，如代码8-13：

//代码8-13

trait Incrementing extends IntQueue {

abstract override def put(x: Int) = super.put(x + 1)

}

trait Filtering extends IntQueue {

abstract override def put(x: Int) = if (x >= 0) super.put(x)

}

有了这些改动，现在可以挑选想要的组成特定的队列。

代码8-14

scala> val queue = **new** BasicIntQueue

**with** Incrementing **with** Filtering

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get() // 1

scala> queue.get() // 2

混入的次序很重要，越靠近右侧的特质越先起作用。当你调用带混入的类的方法时，最右侧特质的方法首先被调用。如果那个方法调用了super，它调用其左侧特质的方法，以此类推。在代码8-14里，Filtering的put首先被调用，因此它移除了开始的负整数。Incrementing的put第二个被调用，因此它对剩下的整数增量。

如果逆转特质的顺序，那么整数首先会加1，然后如果仍然是负的才会被抛弃。

## 8.4 特质构造顺序

特质也可以有构造器，由字段的初始化和其他特质体中的语句构成。这些语句在任何混入该特质的对象在构造时都会被执行。

构造器的执行顺序：

* 调用超类的构造器；
* 特质构造器在超类构造器之后、类构造器之前执行；
* 特质由左到右被构造；
* 每个特质当中，父特质先被构造；
* 如果多个特质共有一个父特质，父特质不会被重复构造
* 所有特质被构造完毕，子类被构造。

构造器的顺序是类的线性化的反向。线性化是描述某个类型的所有超类型的一种技术规格。

## 8.5 class，Object，Trait有什么区别？

### 8.5.1 class

在scala中，类名可以和对象名为同一个名字，该对象称为该类的伴生对象，类和伴生对象可以相互访问他们的私有属性，但是他们必须在同一个源文件内。**类只会被编译，不能直接被执行**，类的申明和主构造器在一起被申明，在一个类中，主构造器只有一个所有必须在内部申明主构造器或者是其他申明主构造器的辅构造器，主构造器会执行类定义中的所有语句。scala对每个字段都会提供getter和setter方法，同时也可以显示的申明，但是针对val类型，只提供getter方法，默认情况下，字段为公有类型，可以在setter方法中增加限制条件来限定变量的变化范围，在scala中方法可以访问改类所有对象的私有字段

### 8.5.2 object

在scala中没有静态方法和静态字段，所以在scala中可以用object来实现这些功能，直接用对象名调用的方法都是采用这种实现方式，例如Array.toString。对象的构造器在第一次使用的时候会被调用，如果一个对象从未被使用，那么他的构造器也不会被执行；对象本质上拥有类（scala中）的所有特性，除此之外，object还可以一扩展类以及一个或者多个特质：例如，

abstract class ClassName（val parameter）{}

object Test extends ClassName(val parameter){}

注意：object不能提供构造器参数，也就是说object必须是无参的。

### 8.5.3 trait

在java中可以通过interface实现多重继承，在Scala中可以通过特质（trait）实现多重继承，不过与java不同的是，它可以定义自己的属性和实现方法体，在没有自己的实现方法体时可以认为它时java interface是等价的，在Scala中也是一般只能继承一个父类，可以通过多个with进行多重继承。

trait TraitA{}

trait TraitB{}

trait TraitC{}

object Test1 extends TraitA with TraitB with TraitC{}

### 8.5.4 包含class，Object，Trait的完整实例

以下演示了特质的完整实例：

/\* 文件名：Test.scala

\*/

trait Equal {

def isEqual(x: Any): Boolean

def isNotEqual(x: Any): Boolean = !isEqual(x)

}

class Point(xc: Int, yc: Int) extends Equal {

var x: Int = xc

var y: Int = yc

def isEqual(obj: Any) =

obj.isInstanceOf[Point] &&

obj.asInstanceOf[Point].x == x

}

object Test {

def main(args: Array[String]) {

val p1 = new Point(2, 3)

val p2 = new Point(2, 4)

val p3 = new Point(3, 3)

println(p1.isNotEqual(p2))

println(p1.isNotEqual(p3))

println(p1.isNotEqual(2))

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

false

true

true

# 9、case class和模式匹配

## 9.1 模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。一个模式匹配包含了一系列备选项，每个都开始于关键字 case。每个备选项都包含了一个模式及一到多个表达式。箭头符号 =>隔开了模式和表达式。

以下是一个简单的整型值模式匹配实例：

object Test {

def main(args: Array[String]) {

println(matchTest(3))

}

def matchTest(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case \_ => "many"

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

many

match 对应 Java 里的 switch，但是写在选择器表达式之后。即： 选择器 match {备选项}。match 表达式通过以代码编写的先后次序尝试每个模式来完成计算，只要发现有一个匹配的case，剩下的case不会继续匹配。

接下来我们来看一个不同数据类型的模式匹配：

object Test {

def main(args: Array[String]) {

println(matchTest("two"))

println(matchTest("test"))

println(matchTest(1))

println(matchTest(6))

}

def matchTest(x: Any): Any = x match {

case 1 => "one"

case "two" => 2

case y: Int => "scala.Int"

case \_ => "many"

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

2

many

one

scala.Int

实例中第一个 case 对应整型数值 1，第二个 case 对应字符串值 two，第三个 case 对应类型模式，用于判断传入的值是否为整型，相比使用isInstanceOf来判断类型，使用模式匹配更好。第四个 case 表示默认的全匹配备选项，即没有找到其他匹配时的匹配项，类似 switch 中的 default。

## 9.2 case class 样例类

使用了case关键字的类定义就是就是样例类(case classes)，样例类是种特殊的类，经过优化以用于模式匹配。

以下是样例类的简单实例：

object Test {

// 样例类（写在Object的里面，main函数的外面）

case class Person(name: String, age: Int)

def main(args: Array[String]) {

    val alice = new Person("Alice", 25)

    val bob = new Person("Bob", 32)

    val charlie = new Person("Charlie", 32)

for (person <- List(alice, bob, charlie)) {

    person match {

case Person("Alice", 25) => println("Hi Alice!")

case Person("Bob", 32) => println("Hi Bob!")

case Person(name, age) =>

println("Age: " + age + " year, name: " + name + "?")

}

}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Hi Alice!

Hi Bob!

Age: 32 year, name: Charlie?

另有例子如下：

mport scala.collection.mutable.\_  
object test {  
 case class Person(id: String, name: String, sex: String)  
 def main(args: Array[String]): Unit = {  
 val p1 = Person(id = "1", name = "zhangsan", "true")  
  
 val pp = Set("1,zhangsan,true", "2,lisi,false", "3,wangwu,true")  
 val result = pp.map(i => i.split(",")).map(i => Person(i(0), i(1), i(2))).foreach(println)  
 }  
}

输出结果如下：

Person(2,lisi,false)

Person(3,wangwu,true)

Person(1,zhangsan,true)

定义一个case class的代码如下：

case class Currency(value: Double, unit: String)

当你定义了一个case class之后，编译器会自动帮你做如下事情：

* 构造器的每个参数都成为val，除非显式被声明为var，但是并不推荐这么做；
* 在伴生对象中提供了apply方法，所以可以不使用new关键字就可构建对象；
* 提供unapply方法使模式匹配可以工作；
* 生成toString、equals、hashCode和copy方法，除非显示给出这些方法的定义。

# 10、隐式转换

## 10.1 什么是隐式转换

通过隐式转换，程序员可以在编写Scala程序时故意漏掉一些信息，让编译器去尝试在编译期间自动推导出这些信息来，这种特性可以极大的减少代码量，忽略那些冗长，过于细节的代码。

使用方式：

* 将方法或变量标记为implicit
* 将方法的参数列表标记为implicit
* 将类标记为implicit

Scala支持两种形式的隐式转换：

* 隐式值：用于给方法提供参数
* 隐式视图：用于类型间转换或使针对某类型的方法能调用成功

## 10.2 隐式值

例1：声明person方法。其参数为name，类型String

scala> def person(implicit name : String) = name //name为隐式参数

person: (implicit name: String)String

直接调用person方法

scala> person

<console>:9: error: could not find implicit value for parameter name: String

person

^

报错！编译器说无法为参数name找到一个隐式值。

定义一个隐式值后再调用person方法：

scala> implicit val p = "mobin" //p被称为隐式值

p: String = mobin

scala> person

res1: String = mobin

因为将p变量标记为implicit，所以编译器会在方法省略隐式参数的情况下去搜索作用域内的隐式值作为缺少参数。

但是如果此时你又在REPL中定义一个隐式变量，再次调用方法时就会报错：

scala> implicit val p1 = "mobin1"

p1: String = mobin1

scala> person

<console>:11: error: ambiguous implicit values:

both value p of type => String

and value p1 of type => String

match expected type String

person

^

匹配失败，所以隐式转换必须满足无歧义规则，在声明隐式参数的类型是最好使用特别的或自定义的数据类型，不要使用Int、String这些常用类型，避免碰巧匹配。

## 10.3 隐式视图

### 10.3.1 隐式转换为目标类型：把一种类型自动转换到另一种类型

例2：将整数转换成字符串类型：

scala> def foo(msg : String) = println(msg)

foo: (msg: String)Unit

scala> foo(10)

<console>:11: error: type mismatch;

found : Int(10)

required: String

foo(10)

^

显然不能转换成功，解决办法就是定义一个转换函数给编译器将int自动转换成String

scala> implicit def intToString(x : Int) = x.toString

intToString: (x: Int)String

scala> foo(10)

10

### 10.3.2 隐式转换调用类中本不存在的方法

例3：通过隐式转换，使对象能调用类中本不存在的方法

class SwingType{

def wantLearned(sw : String) = println("兔子已经学会了"+sw)

}

object swimming{

implicit def learningType(s : AminalType) = new SwingType

}

object AminalType extends App{

import com.mobin.scala.Scalaimplicit.swimming.\_

val rabbit = new AminalType

rabbit.wantLearned("breaststroke") //蛙泳

}

编译器在rabbit对象调用时发现对象上并没有wantLearned方法，此时编译器就会在作用域范围内查找能使其编译通过的隐式视图，找到learningType方法后，编译器通过隐式转换将对象转换成具有这个方法的对象，之后调用wantLearned方法。

可以将隐式转换函数定义在包中，在使用时导入隐式视图到作用域中即可，如例4的learningType函数。

例4：

class SwingType{

def wantLearned(sw : String) = println("兔子已经学会了"+sw)

}

package swimmingPage {

object swimming {

//将转换函数定义在包中

implicit def learningType(s : AminalType) = new SwingType

}

}

class AminalType

object AminalType extends App{

//使用时显示的导入

import com.mobin.scala.Scalaimplicit.swimmingPage.swimming.\_

val rabbit = new AminalType

rabbit.wantLearned("breaststroke") //蛙泳

}

### 10.3.3 结论

像intToString、learningType这类的方法就是隐式视图，通常为Int => String的视图，定义的格式如下：

implicit def originalToTarget (<argument> : OriginalType) : TargetType

其通常用在于以两种场合中：

1.如果表达式不符合编译器要求的类型，编译器就会在作用域范围内查找能够使之符合要求的隐式视图。如例2，当要传一个整数类型给要求是字符串类型参数的方法时，在作用域里就必须存在Int => String的隐式视图。

2.给定一个选择e.t，如果e的类型里并没有成员t，则编译器会查找能应用到e类型并且返回类型包含成员t的隐式视图，如例3。

## 10.4 隐式类

在scala2.10后提供了隐式类，可以使用implicit声明类，但是需要注意以下几点：

1、其所带的构造参数有且只能有一个；

2、隐式类必须被定义在类，伴生对象和包对象里；

3、隐式类不能是case class（case class在定义会自动生成伴生对象与2矛盾）；

4、作用域内不能有与之相同名称的标识符。

例5：

object Stringutils {

implicit class StringImprovement(val s : String){ //隐式类

def increment = s.map(x => (x +1).toChar)

}

}

object Main extends App{

import com.mobin.scala.implicitPackage.Stringutils.\_

println("mobin".increment)

}

编译器在mobin对象调用increment时发现对象上并没有increment方法，此时编译器就会在作用域范围内搜索隐式实体，发现有符合的隐式类可以用来转换成带有increment方法的StringImprovement类，最终调用increment方法。

## 10.5 隐式操作规则

### 10.5.1 隐式转换的时机：

1、当方法中的参数的类型与目标类型不一致时。

2、当对象调用类中不存在的方法或成员时，编译器会自动将对象进行隐式转换。

### 10.5.2 隐式解析的查找范围

隐式转换本身是一种代码查找机制，编译器是如何查找到缺失信息的，解析具有以下两种规则：

* 当前代码作用域。首先会在当前代码作用域下查找隐式实体（隐式方法、隐式类、隐式对象）；
* 如果第一条规则查找隐式实体失败，编译器会继续在隐式参数类型的隐式作用域里查找。一个类型的隐式作用域指的是与该类型相关联的所有的伴生对象。

**对于一个隐式实体的类型T它的隐式搜索区域包括如下：**

* 假如T定义为：T with A with B with C，那么A, B, C的伴生对象都是T的搜索区域。
* 如果T是类型参数，那么参数类型和基础类型都是T的搜索部分。比如对于类型List[Foo]，List和Foo都是搜索区域。
* 如果T是一个单例类型p.T，那么p和T都是搜索区域。
* 如果T是类型注入p#T，那么p和T都是搜索区域。

所以，只要在上述的任何一个区域中搜索到合适的隐式转换，编译器都可以使编译通过。

### 10.5.3 隐式转换规则

1、标记规则：只有标记为implicit的变量，函数或对象定义才能被编译器当做隐式操作目标。

2、作用域规则：插入的隐式转换必须是单一标示符的形式处于作用域中，或与源/目标类型关联在一起。单一标示符是说当隐式转换作用时应该是这样的形式：file2Array(arg).map(fn)的形式，而不是foo.file2Array(arg).map的形式。假设file2Array函数定义在foo对象中，我们应该通过import foo.\_或者import foo.file2Array把隐式转换导入。简单来说，隐式代码应该可以被"直接"使用，不能再依赖类路径。

假如我们把隐式转换定义在源类型或者目标类型的伴生对象内，则我们可以跳过单一标示符的规则。因为编译器在编译期间会自动搜索源类型和目标类型的伴生对象，以尝试找到合适的隐式转换。

3、无歧义规则：不能存在多于一个隐式转换使某段代码编译通过。因为这种情况下会产生迷惑，编译器不能确定到底使用哪个隐式转换。（如例1）

4、单一调用规则：不会叠加（重复嵌套）使用隐式转换。一次隐式转化调用成功之后，编译器不会再去寻找其他的隐式转换。

5、显示操作优先规则：当前代码类型检查没有问题，编译器不会尝试查找隐式转换。

# 11、Future和Promise

Future和Promise是Scala提供的最吸引人的特性之一，借助Future和Promise你可以轻松地编写完全异步非阻塞的代码，这在多处理器时代显得格外重要。

## 11.1 Future

Future用于获取异步任务的返回结果。Future有两种状态：完成(completed)和未完成(not completed)。处于完成状态的Future可能包含两种情况的信息，一种是异步任务执行成功了，Future中包含异步任务执行成功的返回结果；另一种是异步任务执行失败了，Future中包含了相应的Exception信息。Future的独特之处在于它的值只能被写入一次，之后就会变为一个不可变值，其中包含成功或失败信息。你可以在Future上注册一个回调函数，以便在任务执行完成后得到通知：

import scala.concurrent.ExecutionContext.Implicits.global

val f = Future{ 1 + 2 }

f.onComplete{ t =>

t match{

case Success(v) => println("success: " + v)

case Failure(t) => println("failed: " + t.getMessage)

}

} //等待任务结束

Await.ready(f, 10 seconds)

onComplete方法接受一个一元函数，类型为：Try[T] => U。Try类型和Option类型很像，也有两个子类Success和Failure，前者表示任务执行成功，后者表示任务执行失败。

* *第1行import语句导入了一个隐式的ExecutionContext，你可以把它理解成是一个线程池，Future类在需要时会自动使用其上的线程。在Scala中你不需要直接和线程打交道。*

由于Future也是一个容器类，所以可以使用for语句取回它的值：

val f = Future{ 1 + 2 }

for(v <- f) {

println(v) // 3

}

也可以使用map方法对任务结果进行转换：

val f1 = Future{ 1 + 2 }

val f2 = f1.map(v => v % 2)

for(v <- f2) {

println(v) // 1

}

利用for语句可以等待多个Future的返回结果：

val f1 = Future{ 1 + 2 }

val f2 = Future{ 3 + 4 }

for{

v1 <- f1

v2 <- f2

} {

println(v1 + v2) // 10

}

结合yield可以返回一个新的Future：

val f1 = Future{ 1 + 2 }

val f2 = Future{ 3 + 4 }

val f3 =

for{

v1 <- f1

v2 <- f2

} yield {

v1 + v2

}

## 11.2 Promise

有时我们需要精细地控制Future的完成时机和返回结果，也就是说我们需要一个控制Future的开关，这个开关就是Promise。每个Promise实例都会有一个唯一的Future与之相关联：

val p = Promise[Int]()

val f = p.future

for(v <- f) { println(v) }

//3秒钟之后返回3

Thread.sleep(3000)

p.success(3)

//等待任务结束

Await.ready(f, 10 seconds)

# 12、问题解决

这部分是汇总的在串讲时提出的问题。

## 12.1 apply方法是什么

### 12.1.1 Class和Object中的Apply()方法

Scala中如果一个Class和一个Object同名，则称Class是Object的伴生类。Scala没有Java的Static修饰符，Object下的成员和方法都是静态的，类似于Java里面加了Static修饰符的成员和方法。

Class和Object都可以定义自己的Apply()方法，**类名()**调用Object下的Apply()方法，**变量名()**调用Class下的Apply()方法。

class ApplyTest{

def apply() {

println("This is a class, apply()...")

}

}

object ApplyTest{

def apply() = {

println("This is an object, apply()...")

new ApplyTest()

}

}

object ApplyExample {

def main(args: Array[String]) {

var at=ApplyTest()

at()

}

}

分析：

var at=ApplyTest() //这是类名()，调用的是Object下的Apply()方法，由于此时并没有创建类的对象，所以也只能调用Object下的静态方法。

new ApplyTest() //创建了一个类的对象，并且返回给了变量at。

at() //这是变量名()，显然调用的是Class的Apply()方法。

### 12.1.2 apply方法的用法

在明确了方法调用的接收者的情况下，若方法只有一个参数时，调用的时候就可以省略点及括号。如 “0 to 2”，实际完整调用是 “0.to(2)”。但 “println(2)” 不能写成 “println 10“”，因为未写出方法调用的接收者 Console，所以可以写成 “Console println 10”。 到这里就要讲到 apply方法相关的约定，描述的是直接在对象后直接加圆括号的代码的情况下，那就是：

在用括号传递给变量(对象)一个或多个参数时，即：Object(参数1，参数2，......，参数n)，会自动调用apply()方法。

对上面那段话的解释可由下面几个例子得到很好的理解：

例1、theArray(0), 取数组的第一个元素的操作会转换成 theArray.apply(0) 操作，这也能解释为什么 Scala 数组取值不用中括号括下标的方式，因为它也是一次方法调用

例2、anyObject("key1") 会被转换成 anyObject.apply("key") 操作，这也会让我们想到 Map 的取值操作，的确如此。我们可以作个例子：

class SomeClass {

    def apply(key: String): String = {

        println("apply method called, key is: " + key)

        "Hello World!"

    }

}

val anyObject = new SomeClass

println(anyObject("key1"))

执行后输出结果是：

apply method called, key is: key1  
Hello World!

说明是调用到了相应的 apply 方法的。

例3、我们在构造 Array 或  Map 时，会简单的写成：

val numNames = Array("zero", "one", "two")

这里也是调用的 apply 方法，我们看起来好像是作用在类 Array 上的，其实不然，而是作用在 Array 的伴生对象(object Array)上的，调用的是伴生对象 Array 的  apply 方法，即：

val numNames = Array.apply("zero", "one", "two")

上面的功效像是调用的 Array 类的 apply 工厂方法。同样看个单例对象的例子，也解释了伴生对象的 apply 方法的调用。

object EMail {

    def apply(user: String, domain: String): String = {

        println("apply method called")

        user + "@" + domain

    }

}

val email = EMail("fantasia", "sina.com")

println(email)

上面代码执行后输出结果是：

apply method called  
[fantasia@sina.com](mailto:fantasia@sina.com)

## 12.2 Scala中的class怎么定义构造器（get/set）

### 12.2.1 主从构造器

在Scala中，有两种构造器，主构造器（primary constructor）和从构造器（auxiliary constructor）。

/\* 构造器 \*/

object constractTest {

    def main(args: Array[String]): Unit = {

        var p = new Person(20)

        println(p.name)

        println(p.age)

        println(p.message)

    }

    //定义类的时候，就定义了构造器。即：是（）内的内容。

    //主构造器会将类定义{}里的所有语句执行一次。

    //如果主构造器参数不带val或var，那么会根据是否被方法使用来决定。

    //如果不带val或var的参数被方法使用了，它会变为对象私有字段；

    //如果没有被方法使用，则被当成一个普通的参数，不升级成字段。

//这里的主构造器是(pname: String, page: Int).

    class Person(val pname: String, page: Int) {

        println("start...")

        var name = pname

        var age = page

        var message = ""

        //从构造器（辅助构造器），调用主构造器。

        //规则：每一个从构造器的第一个动作都是调用同一个类里面其他的构造器，因此主构造器是类的唯一入口点。

        //这里的从构造器，调用了主构造器。

        def this(pname: String) = this(pname, 20)

        //同时加入其他逻辑

        def this(page: Int) {

            this("mike", page)

            message = "hello"

        }

        println("end...")

    }

}

执行结果：

start...  
end...  
mike  
20  
hello

### 12.2.2 访问器

和Java一样，Scala也有setter和getter访问器。在Scala里，对象的每个非私有的var类型成员变量都隐含定义了setter和getter方法。但是setter和getter方法的命名方式并没有沿袭Java的约定。在Scala中，var变量x的getter方法命名为“y”，setter方法命名为“y\_=”（一般情况下，为了代码的可读性，y和x是一样的）。如：

//访问类里的var变量

class Person {

    private var \_name: String = \_

    private[this] var \_age: Int = \_

    //类里定义的变量，不能用构造器初始化。

    private var \_address: String = \_

    //getter

    def name: String = \_name

    //setter

    def name\_= (name: String): Unit = {

        \_name = name

    }

    //getter

    def age: Int = \_age

    //setter

    def age\_= (age: Int): Unit = {

        \_age = age

    }

}

另外，setter和getter访问器不仅可以访问类里面定义的var变量，而且还可以访问主构造器里的var变量，而且getter方法既可以访问var变量，也可以访问val变量（因为val变量不可变，所以没有setter）。如：

//访问主构造器里的var变量

class Person private( private var \_name: String, private var \_age: Int){

    //类里定义的变量，不能用构造器初始化。

    private var \_address: String = \_

    def this(age: Int) = this("xx", age)

    //getter

    def name: String = \_name

    //setter

    def name\_= (name: String): Unit = {

        \_name = name

    }

    //getter

    def age: Int = \_age

    //setter

    def age\_= (age: Int): Unit = {

        \_age = age

}

    //getter

    def address: String = \_address

    //setter

    def address\_= (address: String): Unit = {

        \_address = address

    }

}

注意：getter和setter方法获取了原var变量的可见性。如var变量定义为public，则其getter和setter也是public；var变量定义为protected，则其getter和setter也是protected。