# ========第一篇：Scala编程语言========

# 课程目标

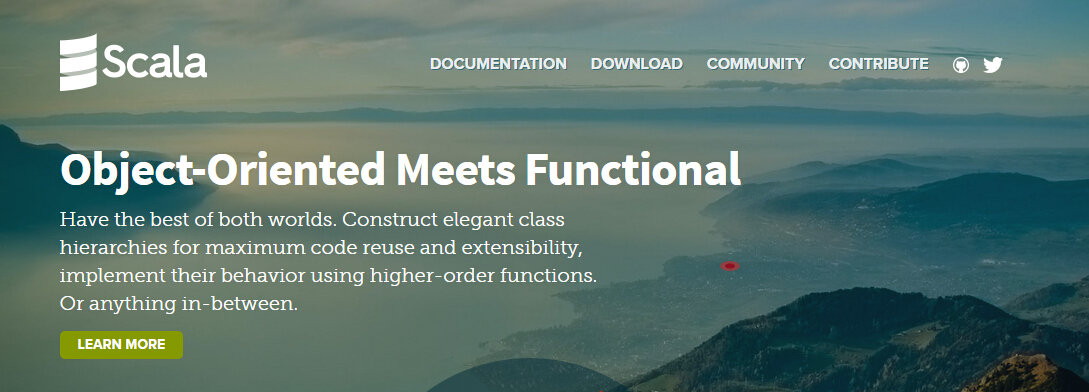
## 目标1：（初级）熟练使用scala编写Spark程序

## 目标2：（中级）动手编写一个简易Spark通信框架

## 目标3：（高级）为阅读Spark内核源码做准备

# Scala概述

## 什么是Scala

Scala是一种多范式的编程语言，其设计的初衷是要集成面向对象编程和函数式编程的各种特性。Scala运行于Java平台（Java虚拟机），并兼容现有的Java程序。

## 为什么要学Scala

* 优雅：这是框架设计师第一个要考虑的问题，框架的用户是应用开发程序员，API是否优雅直接影响用户体验。
* 速度快：Scala语言表达能力强，一行代码抵得上Java多行，开发速度快；Scala是静态编译的，所以速度会快很多。
* 能融合到Hadoop生态圈：Hadoop现在是大数据事实标准，Spark并不是要取代Hadoop，而是要完善Hadoop生态。JVM语言大部分可能会想到Java，但Java做出来的API太丑，或者想实现一个优雅的API太费劲。
* 学习Scala编程语言，为后续学习Spark和Kafka奠定基础。



## Scala语言的发展史

* 2001年,Martin Odersky基于Funnel的工作开始设计Scala
* 2003年底/2004年初,Java平台的Scala发布
* 2004年6月,.NET平台的Scala发布
* 2006年3月,该语言v2.0发布
* 2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala
* 截至2018年07月，最新版本是版本 2.12.7

## Scala与Java的关系

Scala与Java的关系是非常紧密的！！

因为Scala是基于Java虚拟机，也就是JVM的一门编程语言。所有Scala的代码，都需要经过编译为字节码，然后交由Java虚拟机来运行。

所以Scala和Java是可以无缝互操作的。Scala可以任意调用Java的代码。所以Scala与Java的关系是非常非常紧密的。

# Scala编译器安装

## 安装JDK

因为Scala是运行在JVM平台上的，所以安装Scala之前要安装JDK，安装JDK1.8版本。

## 安装Scala

### Windows安装Scala编译器

访问Scala官网http://www.scala-lang.org/下载Scala编译器安装包，目前最新版本是2.12.x，但是目前大多数的框架都是用2.11.x编写开发的，Spark2.x使用的就是2.11.x，所以这里推荐2.11.x版本，下载scala-2.11.12.msi后点击下一步就可以了

* 安装JDK
* 下载Scala：http://www.scala-lang.org/download/
* 安装Scala：
* 设置环境变量：SCALA\_HOME和PATH路径

SCALA\_HOME：D:\developer\scala-2.11.12

PATH：%SCALA\_HOME%\bin

* 验证Scala

### Linux安装Scala编译器

下载Scala地址http://downloads.typesafe.com/scala/2.11.12/scala-2.11.12.tgz然后解压Scala到指定目录

tar -zxvf scala-2.11.12.tgz -C /usr/java

配置环境变量，将scala加入到PATH中

vi /etc/profile

export JAVA\_HOME=/usr/java/jdk1.8.0\_111

export PATH=$PATH:$JAVA\_HOME/bin:/usr/java/scala-2.11.12/bin

### Scala的运行环境

REPL（Read Evaluate Print Loop）：命令行

IDE：图形开发工具

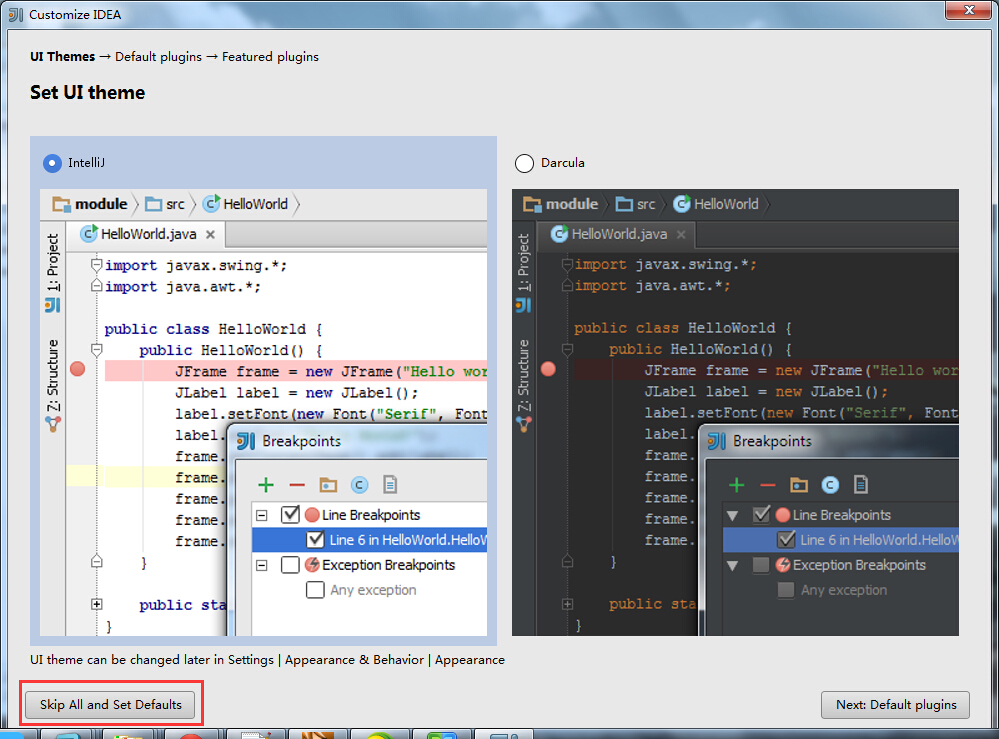
* The Scala IDE (Based on Eclipse)：http://scala-ide.org/
* IntelliJ IDEA with Scala plugin：http://www.jetbrains.com/idea/download/
* Netbeans IDE with the Scala plugin

### Scala开发工具安装

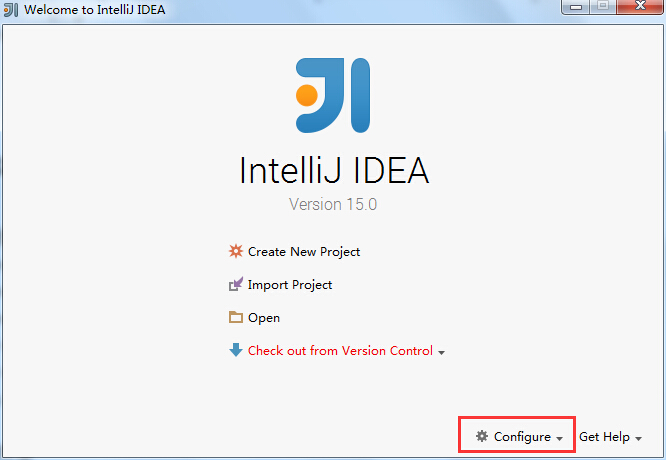
目前Scala的开发工具主要有两种：Eclipse和IDEA，这两个开发工具都有相应的Scala插件，如果使用Eclipse，直接到Scala官网下载即可http://scala-ide.org/download/sdk.html。

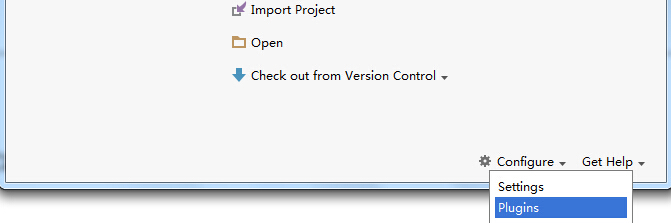
由于IDEA的Scala插件更优秀，大多数Scala程序员都选择IDEA，可以到http://www.jetbrains.com/idea/download/下载社区免费版，点击下一步安装即可，安装时如果有网络可以选择在线安装Scala插件。这里我们使用离线安装Scala插件：

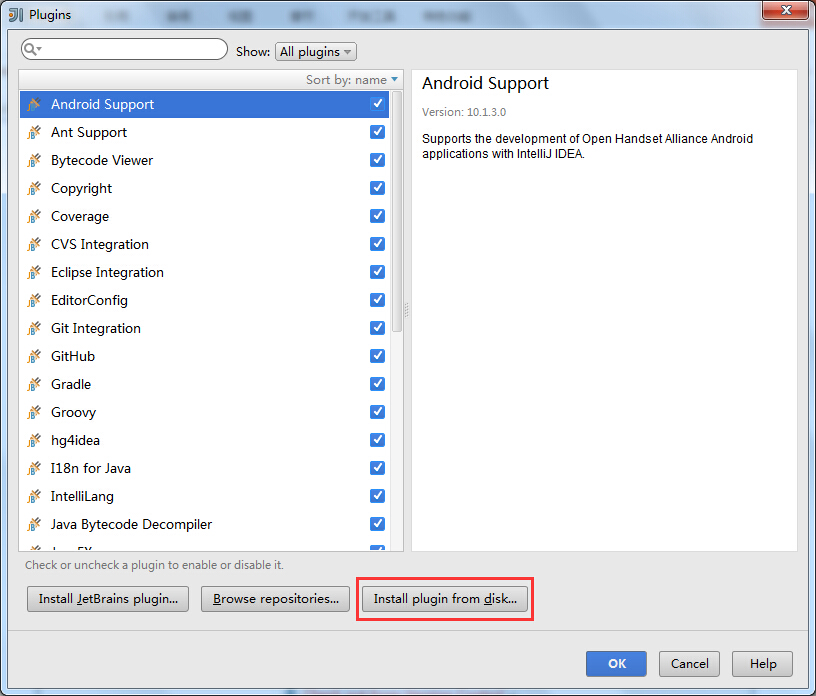
1. 安装IDEA，点击下一步即可。由于我们离线安装插件，所以点击Skip All and Set Defaul
2. 下载IEDA的scala插件，地址http://plugins.jetbrains.com/?idea\_ce



1. 安装Scala插件：Configure -> Plugins -> Install plugin from disk -> 选择Scala插件 -> OK -> 重启IDEA







# Scala基础

在scala中声明变量有两个关键字val和var

在scala程序中，通常建议使用val，也就是常量，因此比如类似于spark的大型复杂系统中，需要大量的网络传输数据，如果使用var，可能会担心值被错误的更改

## Scala变量声明和使用

* 变量(声明变量的通用格式)

例如：关键字 变量名:变量类型=变量值

* 使用val和var申明变量

例如：scala> val answer = 8 \* 3 + 2

可以在后续表达式中使用这些名称

* val：value 简写，表示的意思为值，不可变

要申明其值可变的变量：val

* var：variable 简写，表示的变量，可以改变值

要申明其值不可变的变量：var

* 例子

|  |
| --- |
| **object** VariableTest {  **def** main(args: Array[String]) {  *//使用val定义的变量值是不可变的，相当于java里用final修饰的变量* **val** i = 1  *//使用var定义的变量是可变得，在Scala中鼓励使用val* **var** s = **"hello"** *//Scala编译器会自动推断变量的类型，必要的时候可以指定类型*  *//变量名在前，类型在后* **val** str: String = **"world"** } } |

**注意：可以不用显式指定变量的类型，Scala会进行自动的类型推导**

## Scala的常用数据类型

在scala中，有以下数据类型。例如：

Byte、Char、Short、Int、Long、Float、Double、Boolean

貌似与Java的基本数据类型的包装类型相同，但是scala没有基本数据类型与包装类型的概念，统一都是类。scala自己会负责基本数据类型和引用类型的转换操作类型的加强版类型：scala使用很多加强类给数据类型增加了上百种增强的功能或方法。

例如，Scala还提供了RichInt、RichDouble、RichChar等类型，RichInt就提供了to方法，1.to(10)，此处Int先隐式转换为RichInt，然后再调用其to方法

## 基本操作符

scala的算术操作符与java的算术操作符也没有什么区别，比如+、-、\*、/、%等，以及&、|、^、>>、<<等。

但是，在scala中，这些操作符其实是数据类型的方法，比如1 + 1，可以写做1.+(1)

例如，1.to(10)，又可以写做1 to 10

scala中没有提供++、--操作符，我们只能使用+和-，比如num = 1，num ++是错误的，必须写做num += 1

## Scala的条件表达式

Scala的if/else语法结构和Java或C++一样。

不过，在Scala中，if表达式是有值的，就是if或者else中最后一行语句返回的值。

* Scala的的条件表达式比较简洁

例如：val num= 30; if (num> 18) 20 else 18

可以将if表达式赋予一个变量，例如，val num = if (age > 18) 1 else 0

* if表达式的类型推断：由于if表达式是有值的，而if和else子句的值类型可能不同，此时if表达式的值是什么类型呢？Scala会自动进行推断，取两个类型的公共父类型。

例如：if(age > 18) 1 else 0，表达式的类型是Int，因为1和0都是Int

例如：if(age > 18) "adult" else 0，此时if和else的值分别是String和Int，则 表达式的值是Any，Any是String和Int的公共父类型。

如果if后面没有跟else，则默认else的值是Unit，也用()表示，类似于java中的 void或者null。

例如：val age = 12; if(age > 18) "旺财"。此时就相当于if(age > 18) "旺财" else

* 例子

|  |
| --- |
| **object** ConditionTest {  **def** main(args: Array[String]) {  **val** x = 1  *//判断x的值，将结果赋给y* **val** y = **if** (x > 0) 1 **else** -1  *//打印y的值  println*(y)   *//支持混合类型表达式* **val** z = **if** (x > 1) 1 **else "error"** *//打印z的值  println*(z)   *//如果缺失else，相当于if (x > 2) 1 else ()* **val** m = **if** (x > 2) 1  *println*(m)   *//在scala中每个表达式都有值，scala中有个Unit类，写做(),相当于Java中的void* **val** n = **if** (x > 2) 1 **else** ()  *println*(n)   *//if和else if* **val** k = **if** (x < 0) 0  **else if** (x >= 1) 1 **else** -1  *println*(k)  } } |

## 循环

在scala中有for循环和while循环，用for循环比较多

### for循环

* 遍历（range范围，字符串，数组）
* 多重for循环
* if守卫：取偶数 for(i <- 1 to 100 if i % 2 == 0) println(i)
* for推导式：构造集合 for(i <- 1 to 10) yield i
* 例子

|  |
| --- |
| **object** ForTest {  **def** main(args: Array[String]) {  *//for(i <- 表达式),表达式1 to 10返回一个Range（区间）*  *//每次循环将区间中的一个值赋给i* **for** (i <- 1 to 10)  *println*(i)   *//for(i <- 数组)* **val** arr = *Array*(**"a"**, **"b"**, **"c"**)  **for** (i <- arr)  *println*(i)   *//高级for循环  //每个生成器都可以带一个条件，注意：if前面没有分号* **for**(i <- 1 to 3; j <- 1 to 3 **if** i != j)  *print*((10 \* i + j) + **" "**)  *println*()   *//for推导式：如果for循环的循环体以yield开始，则该循环会构建出一个集合*  *//每次迭代生成集合中的一个值* **val** v = **for** (i <- 1 to 10) **yield** i \* 10  *println*(v)   }  } |

### while循环

while do循环：Scala有while do循环，基本语义与Java相同。

var n = 10

while(n > 0) { println(n)； n -= 1}

跳出循环语句

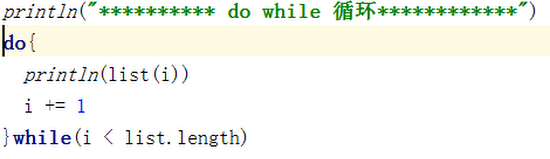
scala没有提供类似于java的break语句， 但是可以使用boolean类型变量、

return或者Breaks的break函数来替代使用

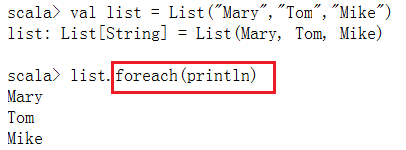


### do ... while循环

至少被执行一次



### 使用foreach进行迭代

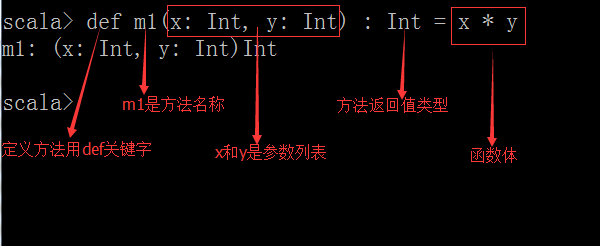


注意：在上面的例子中，foreach接收了另一个函数（println）作为值

# Scala语言的函数式编程

## 定义方法和函数

### Scala方法（定义方法）



定义方法的关键字: def

定义方法的通用格式: def 方法名 (参数列表):方法返回值类型={方法体}

如果方法体中有多行代码，则可以使用代码块的方式包裹多行代码，代码块中最后

一行的返回值就是整个方法的返回值。与Java中不同，不是使用return返回值的。

省略方法的返回值类型：scala会自动根据最后一行语句返回的值推断出方法的返回

值类型 def m1(num:Int)={num\*2}

定义没有返回值的方法

def m1(num:Int):Unit={print(num)}或def m1(num:Int){print(num)}

递归方法返回值的类型必须写明，不能省略掉

### Scala方法（方法的默认参数）

在Scala中，有时我们调用某些方法时，不希望给出参数的具体值，而希望使

用参数自身默认的值，此时就定义在定义方法时使用默认参数。

def sayHello(name: String = "xiaoimg", sex: String = "女", age: Int = 12) = {

println(name + "=" + age + "=" + sex)

}

如果给出的参数不够，则会从作往右依次应用参数。

### scala方法（带名参数）

在调用函数时，也可以不按照函数定义的参数顺序来传递参数，而是使用带

名参数的方式来传递。

sayHello(name = "xiaohong", sex = "nan", age = 12)

还可以混合使用未命名参数和带名参数，但是未命名参数必须排在带名参数

前面。

sayHello("xiaoming","nv",age = 12)

### scala方法（变长参数）

在Scala中，有时我们需要将方法定义为参数个数可变的形式，则此时可以使

用变长参数定义方法。

def sum(nums: Int\*) = {

var res = 0

for (num <- nums)

res += num

res

}

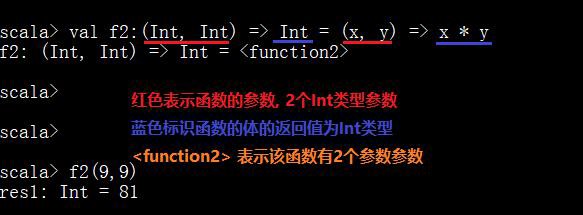
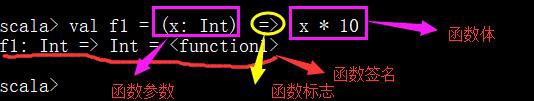
sum(1, 2, 3, 4, 5)

### Scala函数（定义函数）

定义函数的关键字:val

定义函数的通用格式: 关键字 函数名=(参数列表)=>{函数体}

例如：val func=(x:Int,y:Int)=>{x+y}

函数定义方式一:

调用: f1(2) , 其中 f1 为函数的引用, 也可以叫做函数名. function1 表示一个参数的函数. 函数定义方式二:

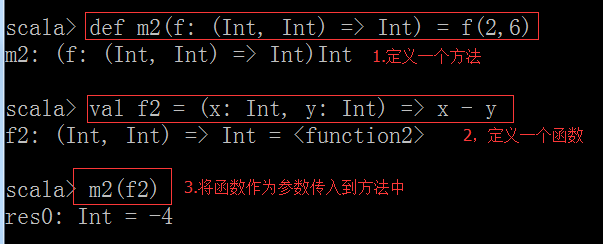
下面为没有任何参数的函数, 函数的返回值为 Int 类型.



### 方法和函数的区别

在函数式编程语言中，函数是“头等公民”，**它可以像任何其他数据类型一样被传递和操作，**在官方并没有明确的给出二者的区别，包括那么在网上看的一些常用资料也不会很详细的区分具体的区别

案例：首先定义一个方法，再定义一个函数，然后将函数传递到方法里面



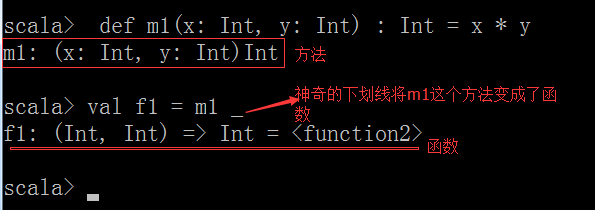
|  |
| --- |
| **object** MethodAndFunctionTest {  *//定义一个方法  //方法m2参数要求是一个函数，函数的参数必须是两个Int类型*  *//返回值类型也是Int类型* **def** m1(f: (Int, Int) => Int) : Int = {  f(2, 6)  }   *//定义一个函数f1，参数是两个Int类型，返回值是一个Int类型* **val** *f1* = (x: Int, y: Int) => x + y  *//再定义一个函数f2* **val** *f2* = (m: Int, n: Int) => m \* n   *//main方法* **def** main(args: Array[String]) {   *//调用m1方法，并传入f1函数* **val** r1 = *m1*(*f1*)  *println*(r1)   *//调用m1方法，并传入f2函数* **val** r2 = *m1*(*f2*)  *println*(r2)  } } |

### 将方法转换成函数（神奇的下划线）

在scala中方法可以转换成函数，提供了两种方法进行转换

一种是：下划线

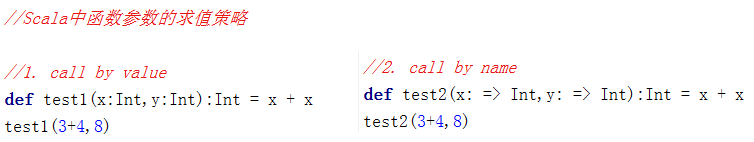
一种是：scala自动隐式的转换



## Scala函数的求值策略

### Scala有两种函数参数的求值策略

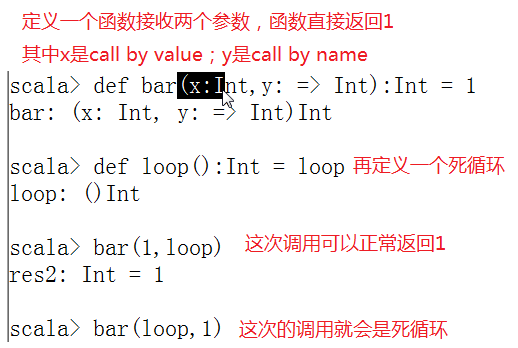
* Call By Value：对函数实参求值，且仅求一次
* Call By Name：函数实参每次在函数体内被用到时都会求值



我们来分析一下，上面两个调用执行的过程：



一份复杂一点的例子：



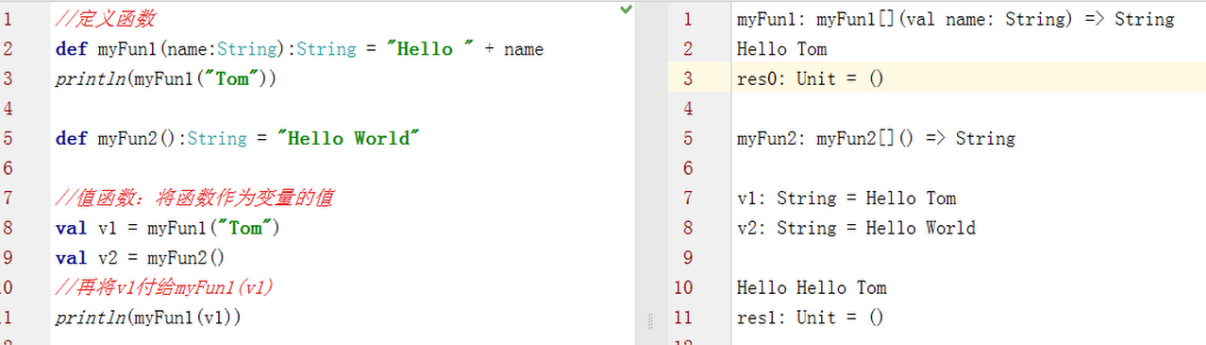
### Scala中的函数参数

* 默认参数
* 代名参数
* 可变参数



## Scala中的函数

在Scala中，函数是“头等公民”，就和数字一样。**可以在变量中存放函数，即：将函数作为变量的值（值函数）**。



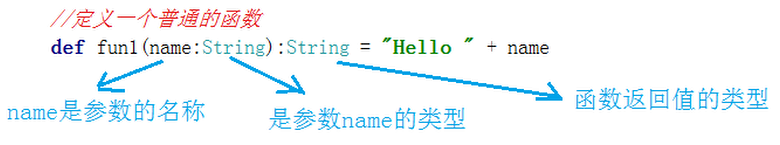
## 匿名函数



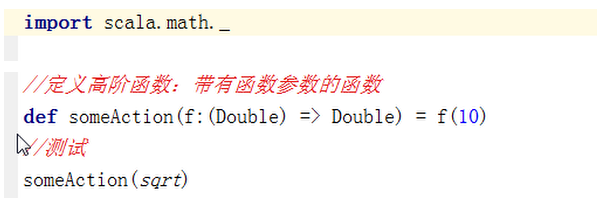
## 带函数参数的函数，即：高阶函数

***示例1：***

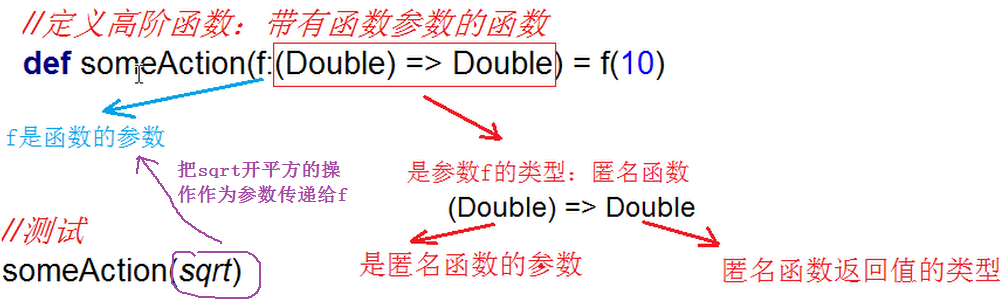
（\*）首先，定义一个最普通的函数



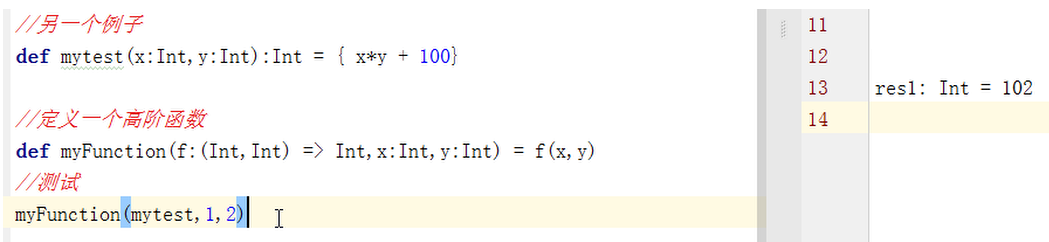
（\*）再定义一个高阶函数



（\*）分析这个高阶函数调用的过程



***示例2：***



在这个例子中，首先定义了一个普通的函数mytest，然后定义了一个高阶函数myFunction；myFunction接收三个参数：第一个f是一个函数参数，第二个是x，第三个是y。而f是一个函数参数，本身接收两个Int的参数，返回一个Int的值。

## 部分参数应用函数

如果函数传递所有预期的参数，则表示已完全应用它。 如果只传递几个参数并不是全部参数，那么将返回部分应用的函数。这样就可以方便地绑定一些参数，其余的参数可稍后填写

补上

*// 定义个输出的方法, 参数为date, message*

**def** log(date: Date, message: String) = {

println(s"$date, $message")

}

**val** date = new Date()

*// 调用log的时候, 传递了一个具体的时间参数, message 为待定参数*

*// logBoundDate 成了一个新的函数, 只有log的部分参数(message)*

val logBoundDate : (String) => Unit = log (date , \_: String)

*// 调用logBoundDate的时候, 只需要传递待传的message参数即可*

*logBoundDate*(**"fuck jerry "**)

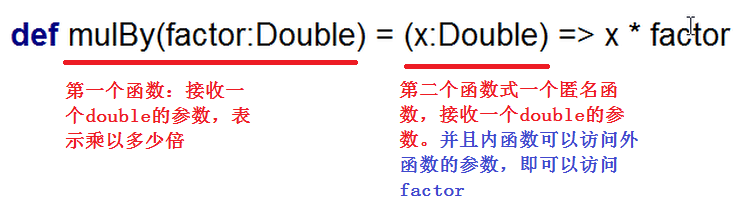
*logBoundDate*(**"fuck 涛涛 "**)

上述，log()方法有两个参数：date 和 message。 我们想要多次调用该方法，具有相同的日

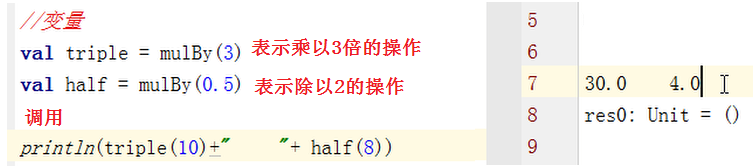
期值，但不同的消息值。可以通过将参数部分地应用到 log()方法来消除将日期传递给每个调用的干扰。为此，首先将值绑定到 date 参数，并将第二个参数绑定到其位置。 结果是存储在变量中的部分应用函数。

## 闭包

就是函数的嵌套，即：在一个函数定义中，包含另外一个函数的定义；**并且在内函数中可以访问外函数中的变量**。



测试上面的函数：



## 柯里化：Currying

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

// 我们看下这个方法的定义, 求2 个数的和

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。经过柯里化之后，函数的通用性有所降低，但是适用性有所提高。

// 分析下其演变过程

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

// (y: Int) => x + y 为一个匿名函数, 也就意味着add 方法的返回值为一个匿名函数

// 那么现在的调用过程为

val result = add(2)

val sum1 = result (3)

val sum2 = result (8)

## 偏函数（讲到模式匹配的时候再讲）

被包在花括号内没有match 的一组case 语句是一个偏函数，它是PartialFunction[A, B]的一个

实例，A 代表参数类型，B 代表返回类型，常用作输入模式匹配。

**object** PartialFuncDemo {

**def** func1: PartialFunction[String, Int] = { **case "one"** => 1

**case "two"** => 2 **case** \_ => -1

}

**def** func2(num: String) : Int = num **match** { **case "one"** => 1

**case "two"** => 2 **case** \_ => -1

}

**def** main(args: Array[String]) {

println (func1 ("one"))

println (func2 ("one"))

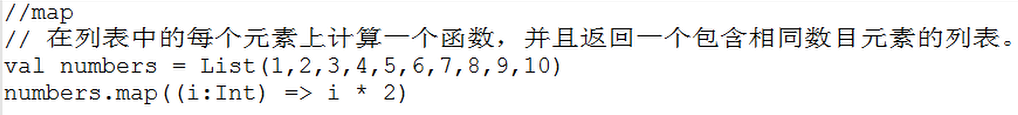
}

}

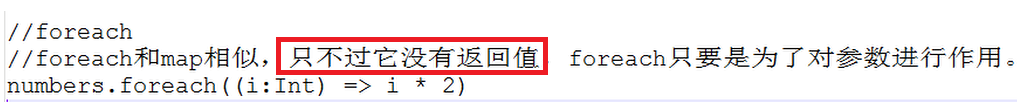
## 高阶函数示例



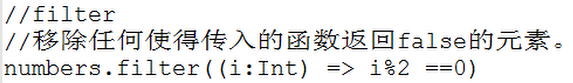
***示例1：***



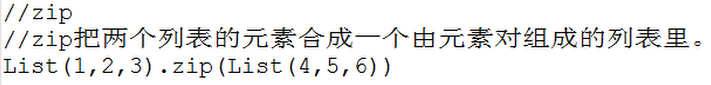
***示例2：***



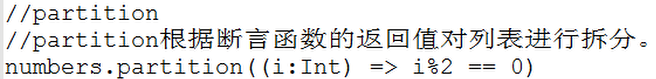
***示例3：***



***示例4：***

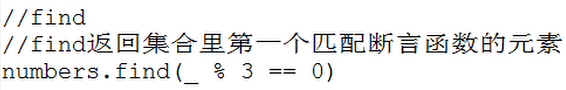


***示例5：***

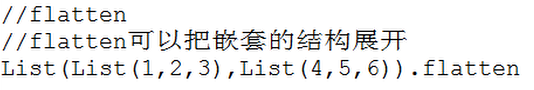


在这个例子中，可以被2整除的被分到一个分区；不能被2整除的被分到另一个分区。

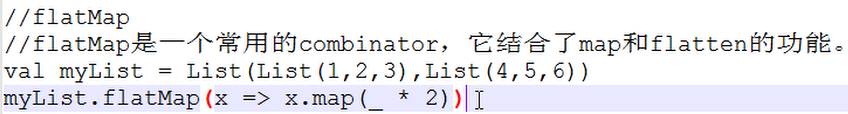
***示例6：***



***示例7：***



***示例8：***



在这个例子中，分为两步：

1. 将(1,2,3)和(4,5,6)这两个集合合并成一个集合
2. 再对每个元素乘以2

# 数组、映射、元组、集合

## 数组

数组分为不可变长数组（Array）和可变长数组（ArrayBuffer）。如字面意思，前者长度不可变，已经写死了，后者长度可变

注:只是长度不可变，但是对应角标元素可变

### 定长数组（不可变数组Array）

在Scala中，Array代表的含义与Java中类似，也是长度不可改变的数组。此外，由于Scala与Java都是运行在JVM中，双方可以互相调用，因此Scala数组的底层实际上是Java数组。例如字符串数组在底层就是Java的String[]，整数数组在底层就是Java的Int[]

|  |
| --- |
| **import** scala.collection.mutable.ArrayBuffer **object** ArrayTest {   **def** main(args: Array[String]) {   *//初始化一个长度为8的定长数组，其所有元素均为0* **val** arr1 = **new** Array[Int](8)  *//直接打印定长数组，内容为数组的hashcode值  println*(arr1)  *//将数组转换成数组缓冲，就可以看到原数组中的内容了  //toBuffer会将数组转换长数组缓冲  println*(arr1.toBuffer)   *//注意：如果new，相当于调用了数组的apply方法，直接为数组赋值  //初始化一个长度为1的定长数组* **val** arr2 = *Array*[Int](10)  *println*(arr2.toBuffer)   *//定义一个长度为3的定长数组* **val** arr3 = *Array*(**"hadoop"**, **"storm"**, **"spark"**)  *//使用()来访问元素  println*(arr3(2))  } } |

### 变长数组（可变数组ArrayBuffer）

在Scala中，如需要类似于Java中的ArrayList这种长度可变的集合类，则可以使用ArrayBuffer

如果不想每次都使用全限定名，则可以预先导入ArrayBuffer类

import scala.collection.mutable.ArrayBuffer

使用ArrayBuffer()的方式可以创建一个空的ArrayBuffer

val b = ArrayBuffer[Int]()

使用+=操作符，可以添加一个元素，或者多个元素

这个语法必须要谨记在心！因为spark源码里大量使用了这种集合操作语法！

b += 1

b += (2, 3, 4, 5)

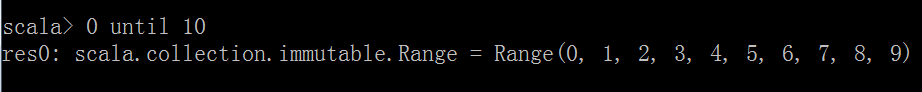
使用++=操作符，可以添加其他集合中的所有元素

|  |
| --- |
| **import** scala.collection.mutable.ArrayBuffer **object** ArrayTest {   **def** main(args: Array[String]) {  *//变长数组（数组缓冲）  //如果想使用数组缓冲，需要导入import scala.collection.mutable.ArrayBuffer包* **val** ab = ArrayBuffer[Int]()  *//向数组缓冲的尾部追加一个元素  //+=尾部追加元素* ab += 1  *//追加多个元素* ab += (2, 3, 4, 5)  *//追加一个数组++=* ab ++= *Array*(6, 7)  *//追加一个数组缓冲* ab ++= ArrayBuffer(8,9)  *//打印数组缓冲ab   //在数组某个位置插入元素用insert* ab.insert(0, -1, 0)  *//删除数组某个位置的元素用remove* ab.remove(8, 2)  *println*(ab)  } } |

### 遍历数组

1.增强for循环

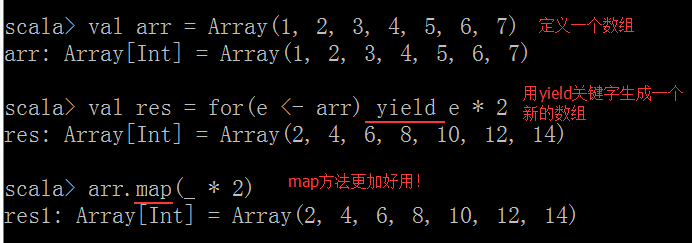
2.好用的until会生成脚标，0 until 10 包含0不包含10

****

|  |
| --- |
| **object** ForArrayTest {  **def** main(args: Array[String]) {  *//初始化一个数组* **val** arr = *Array*(1,2,3,4,5,6,7,8)  *//增强for循环* **for**(i <- arr)  *println*(i)   *//好用的until会生成一个Range  //reverse是将前面生成的Range反转* **for**(i <- (0 until arr.length).reverse)  *println*(arr(i))  } } |

### 数组转换

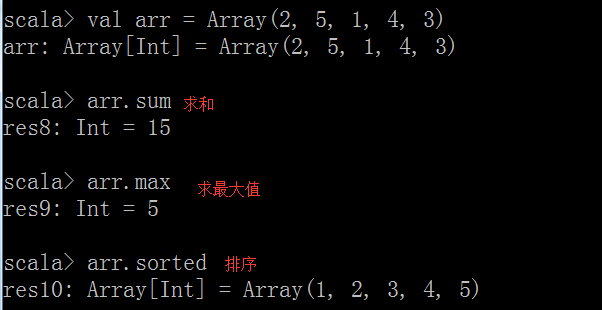
yield关键字将原始的数组进行转换会产生一个新的数组，原始的数组不变



|  |
| --- |
| **object** ArrayYieldTest {  **def** main(args: Array[String]) {  *//定义一个数组* **val** arr = *Array*(1, 2, 3, 4, 5, 6, 7, 8, 9)  *//将偶数取出乘以10后再生成一个新的数组* **val** res = **for** (e <- arr **if** e % 2 == 0) **yield** e \* 10  *println*(res.toBuffer)   *//更高级的写法,用着更爽  //filter是过滤，接收一个返回值为boolean的函数  //map相当于将数组中的每一个元素取出来，应用传进去的函数* **val** r = arr.filter(\_ % 2 == 0).map(\_ \* 10)  *println*(r.toBuffer)  } } |

### 数组常用算法

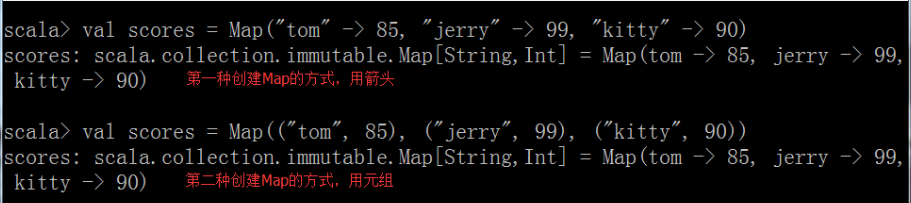
在Scala中，数组上的某些方法对数组进行相应的操作非常方便！



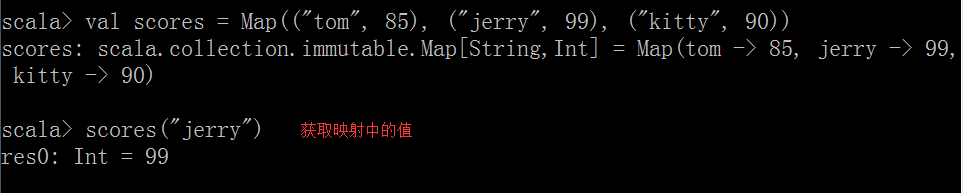
## 映射

在Scala中，把哈希表这种数据结构叫做映射

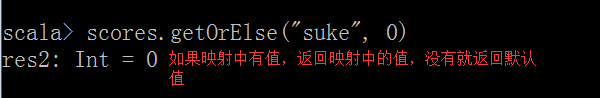
### 构建映射



### 获取和修改映射中的值

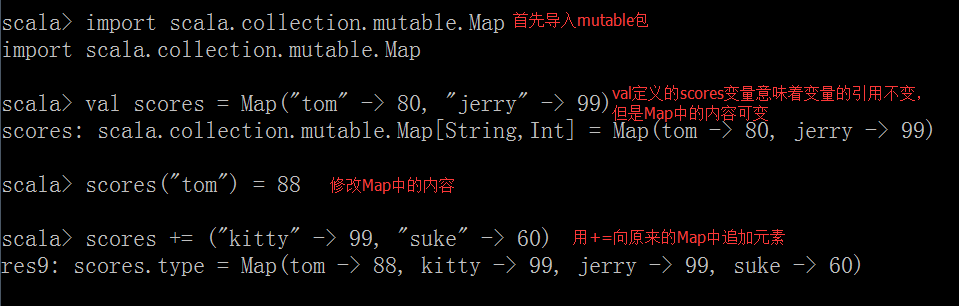


好用的getOrElse



注意：在Scala中，有两种Map，一个是immutable包下的Map，该Map中的内容不可变；另一个是mutable包下的Map，该Map中的内容可变

例子：

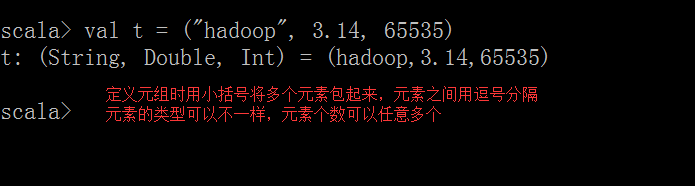


注意：通常我们在创建一个集合是会用val这个关键字修饰一个变量（相当于java中的final），那么就意味着该变量的引用不可变，该引用中的内容是不是可变，取决于这个引用指向的集合的类型

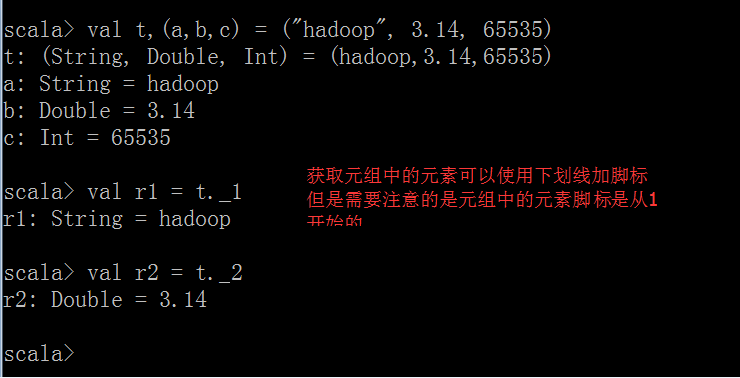
## 元组

映射是K/V对偶的集合，对偶是元组最简单的形式，元组可以封装多个不同类型的值，注意元组的角标是从1 开始的

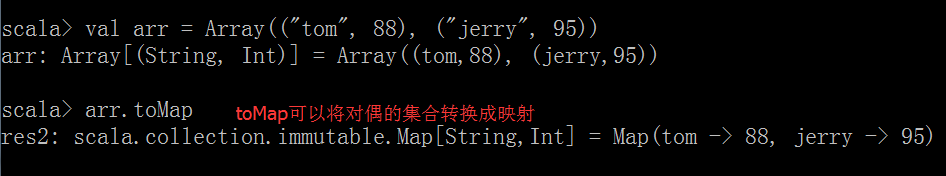
### 创建元组



### 获取元组中的值

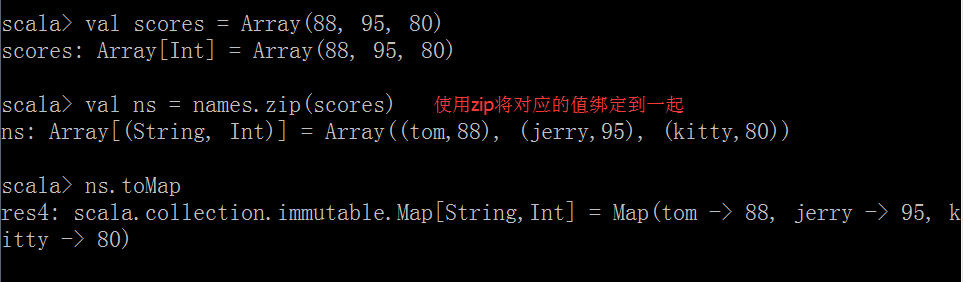


### 将对偶的集合转换成映射



### 拉链操作

**zip**命令可以将多个值绑定在一起



注意：如果两个数组的元素个数不一致，拉链操作后生成的数组的长度为较小的那个数组的元素个数

## 集合

Scala的集合有三大类：序列Seq、集Set、映射Map，所有的集合都扩展自Iterable特质

在Scala中集合有可变（mutable）和不可变（immutable）两种类型，immutable类型的集合初始化后就不能改变了（注意与val修饰的变量进行区别）

### 序列

序列分为不可变长序列（List）和可变长序列（ListBuffer）不可变集合中添加新元素，会生成一个新集合，和不可变集合并不矛盾

给集合中添加元素一般都是从集合头部添加，或者从集合尾部添加，下面是给集合头部添加

元素无放入有序，元素可重复

不可变的序列

import scala.collection.immutable.\_

在Scala中列表要么为空（Nil表示空列表）要么是一个head元素加上一个tail列表。

9 :: List(5, 2) :: 操作符是将给定的头和尾创建一个新的列表

注意：:: 操作符是右结合的，如9 :: 5 :: 2 :: Nil相当于 9 :: (5 :: (2 :: Nil))

|  |
| --- |
| **object** ImmutListTest {   **def** main(args: Array[String]) {  *//创建一个不可变的集合* **val** lst1 = *List*(1,2,3)  *//将0插入到lst1的前面生成一个新的List* **val** lst2 = 0 :: lst1  **val** lst3 = lst1.::(0)  **val** lst4 = 0 +: lst1  **val** lst5 = lst1.+:(0)   *//将一个元素添加到lst1的后面产生一个新的集合* **val** lst6 = lst1 :+ 3   **val** lst0 = *List*(4,5,6)  *//将2个list合并成一个新的List* **val** lst7 = lst1 ++ lst0  *//将lst0插入到lst1前面生成一个新的集合* **val** lst8 = lst1 ++: lst0   *//将lst0插入到lst1前面生成一个新的集合* **val** lst9 = lst1.:::(lst0)   *println*(lst9)  } } |

可变的序列

import scala.collection.mutable.\_

|  |
| --- |
| **import** scala.collection.mutable.ListBuffer  **object** MutListTest **extends** App{  *//构建一个可变列表，初始有3个元素1,2,3* **val** *lst0* = ListBuffer[Int](1,2,3)  *//创建一个空的可变列表* **val** *lst1* = **new** ListBuffer[Int]  *//向lst1中追加元素，注意：没有生成新的集合  lst1* += 4  *lst1*.append(5)   *//将lst1中的元素最近到lst0中， 注意：没有生成新的集合  lst0* ++= *lst1   //将lst0和lst1合并成一个新的ListBuffer 注意：生成了一个集合* **val** *lst2*= *lst0* ++ *lst1   //将元素追加到lst0的后面生成一个新的集合* **val** *lst3* = *lst0* :+ 5 } |

## Set

元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的）

Set集合分为不可变长和可变长

immtable包下面的set为不可变长，mutable包下的set为可变长

不可变的Set

|  |
| --- |
| **import** scala.collection.immutable.HashSet  **object** ImmutSetTest **extends** App{  **val** *set1* = **new** HashSet[Int]()  *//将元素和set1合并生成一个新的set，原有set不变* **val** *set2* = *set1* + 4  *//set中元素不能重复* **val** *set3* = *set1* ++ *Set*(5, 6, 7)  **val** *set0* = *Set*(1,3,4) ++ *set1  println*(*set0*.getClass) } |

可变的Set

|  |
| --- |
| **import** scala.collection.mutable  **object** MutSetTest **extends** App{  *//创建一个可变的HashSet* **val** *set1* = **new** mutable.HashSet[Int]()  *//向HashSet中添加元素  set1* += 2  *//add等价于+=  set1*.add(4)  *set1* ++= *Set*(1,3,5)  *println*(*set1*)  *//删除一个元素  set1* -= 5  *set1*.remove(2)  *println*(*set1*) } |

## Map

映射就是java里面的map，里面存放的是k-v类型的数据，底层是hash表

scala默认导入的是imutable包下面的Map映射，它是不可变的（长度和对应的值都是不可变的）

构建映射

构建映射有两种方式，一种是箭头 ->，一种是元组（）

|  |
| --- |
| **import** scala.collection.mutable  **object** MutMapTest **extends** App{  **val** *map1* = **new** mutable.HashMap[String, Int]()  *//向map中添加数据  map1*(**"spark"**) = 1  *map1* += ((**"hadoop"**, 2))  *map1*.put(**"storm"**, 3)  *println*(*map1*)   *//从map中移除元素  map1* -= **"spark"** *map1*.remove(**"hadoop"**)  *println*(*map1*) } |

# Scala语言的面向对象

## 面向对象的基本概念

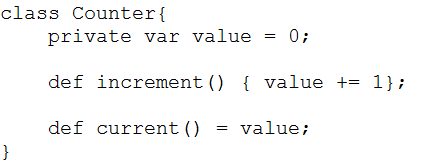
把数据及对数据的操作方法放在一起，作为一个相互依存的整体——对象

面向对象的三大特征：

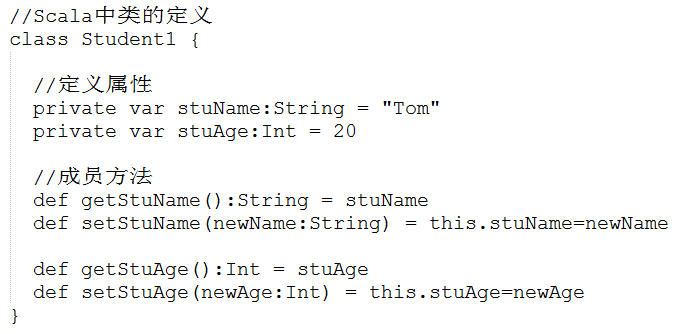
* 封装
* 继承
* 多态

## 类的定义

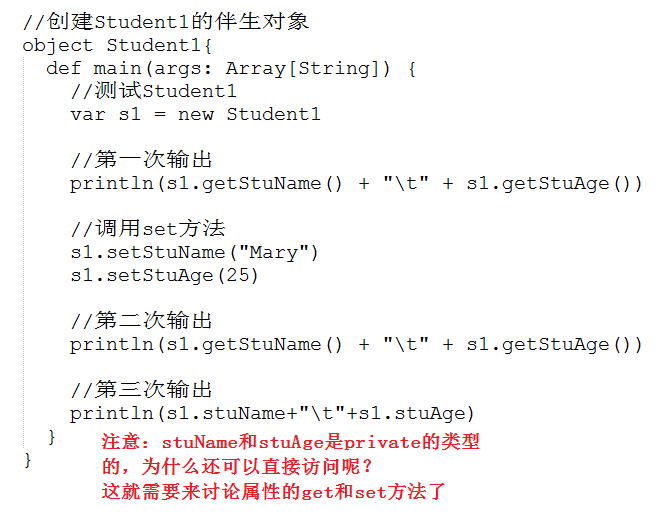
简单类和无参方法：



案例：**注意class前面没有public关键字修饰。**



如果要开发main方法，需要将main方法定义在该类的伴生对象中，即：object对象中，（后续做详细的讨论）。





## 属性的getter和setter方法

当定义属性是private时候，scala会自动为其生成对应的get和set方法

private var stuName:String = "Tom"

* get方法: stuName ----> s2.stuName() 由于stuName是方法的名字，所以可以加上一个括号
* set方法: stuName\_= ----> stuName\_= 是方法的名字

定义属性：private var money:Int = 1000 希望money只有get方法，没有set方法？？

* 办法：将其定义为常量private val money:Int = 1000

private[this]的用法：该属性只属于该对象私有，就不会生成对应的set和get方法。如果这样，就不能直接调用，例如：s1.stuName ---> 错误

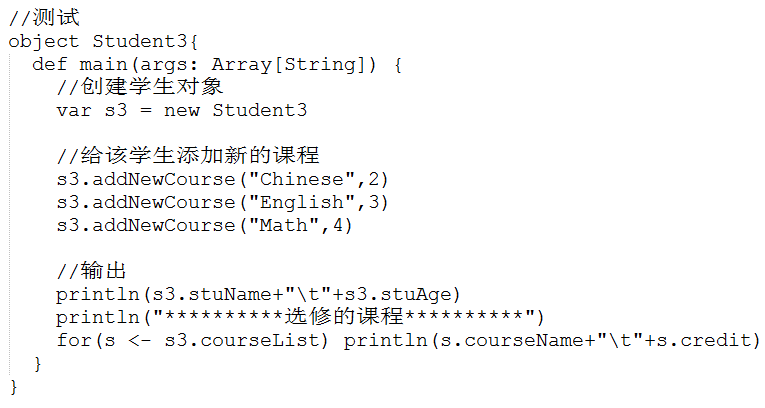


## 内部类（嵌套类）

我们可以在一个类的内部定义一个类，如下：我们在Student类中，再定义了一个Course类用于保存学生选修的课程。



开发一个测试程序进行测试：





## 类的构造器

类的构造器分为：主构造器、辅助构造器

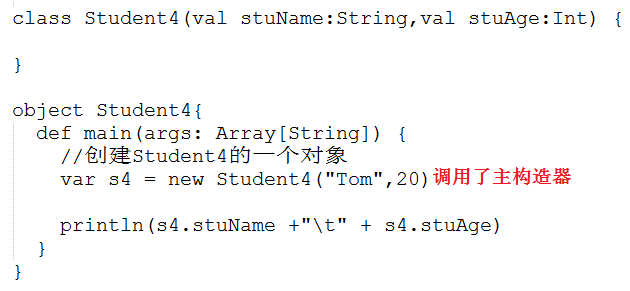
### 主构造器

和类的声明结合在一起，只能有一个主构造器

Student4(val stuName:String,val stuAge:Int)

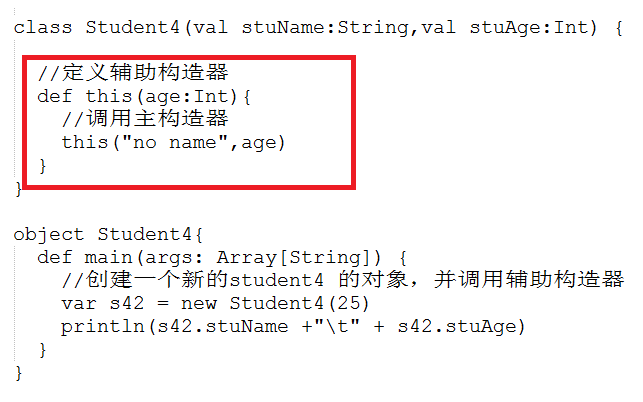
(1) 定义类的主构造器：两个参数

(2) 声明了两个属性：stuName和stuAge 和对应的get和set方法



### 辅助构造器

可以有多个辅助构造器，通过关键字this来实现

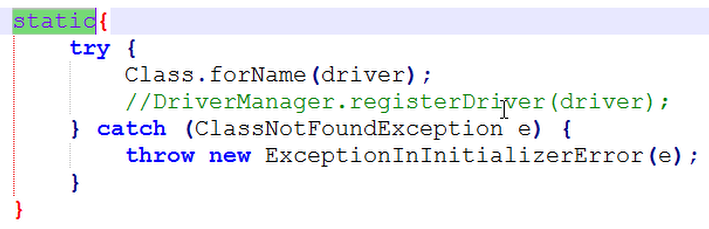




### Scala中的Object对象

Scala没有静态的修饰符，但Object对象下的成员都是静态的 ，若有同名的class，这其作为它的伴生类。在Object中一般可以为伴生类做一些初始化等操作。

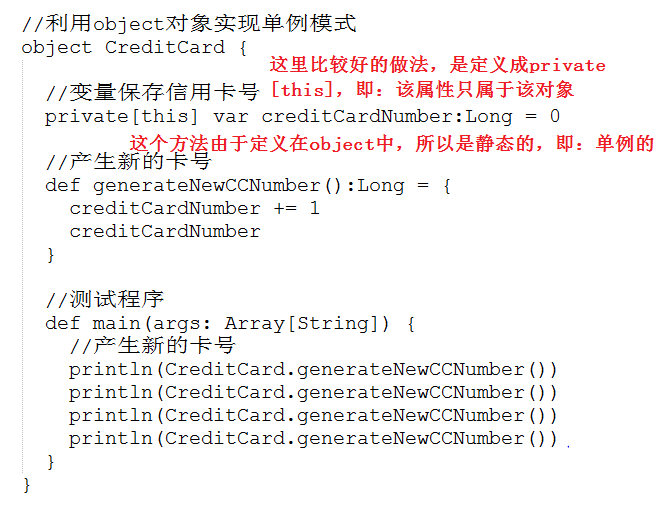
下面是Java中的静态块的例子。在这个例子中，我们对JDBC进行了初始化。



**而Scala中的Object就相当于Java中静态块。**

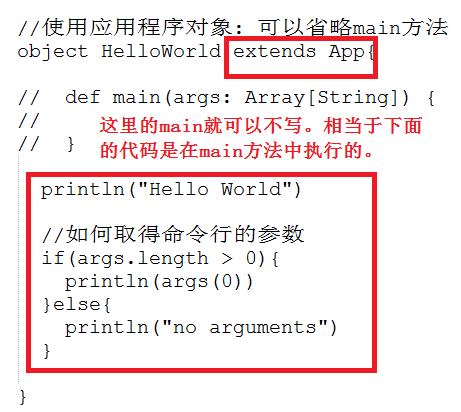
Object对象的应用

* 单例对象





* 使用应用程序对象：**可以省略main方法；需要从父类App继承。**



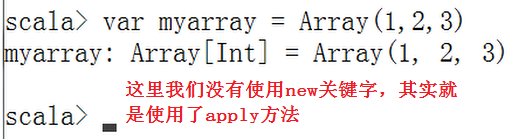


### Scala中的apply方法

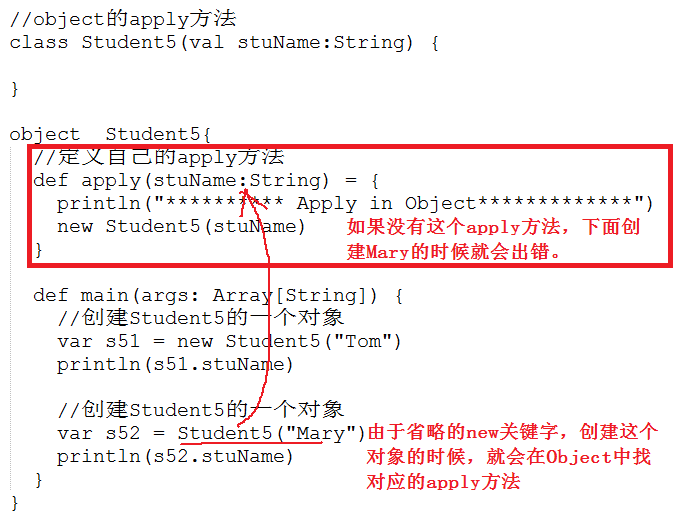
遇到如下形式的表达式时，apply方法就会被调用：

Object(参数1,参数2,......,参数N)

通常，这样一个apply方法返回的是伴生类的对象；**其作用是为了省略new关键字**



Object的apply方法举例：

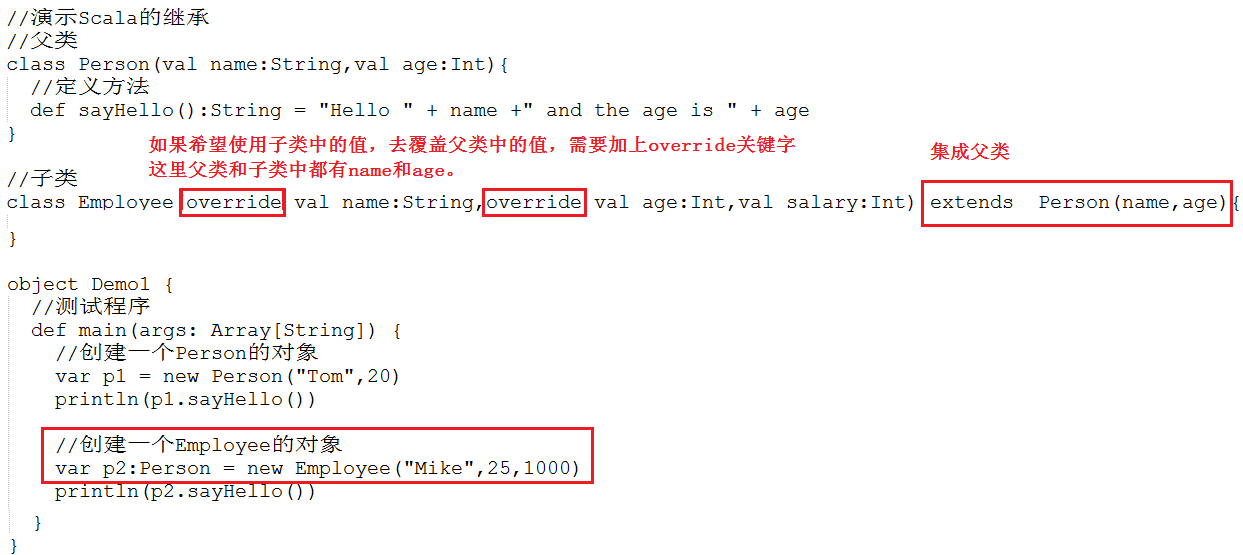




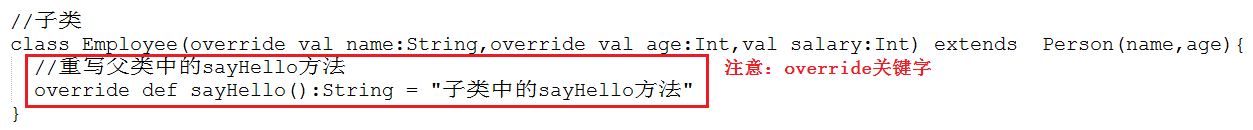
### Scala中的继承

Scala和Java一样，使用extends关键字扩展类。

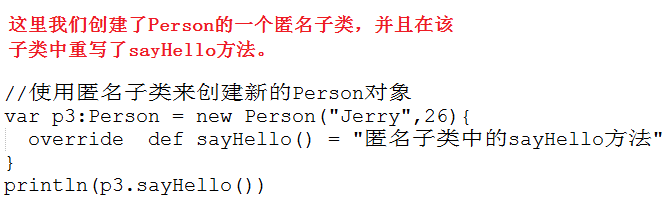
* 案例一：Employee类继承Person类



* 案例二：在子类中重写父类的方法



* 案例三：使用匿名子类



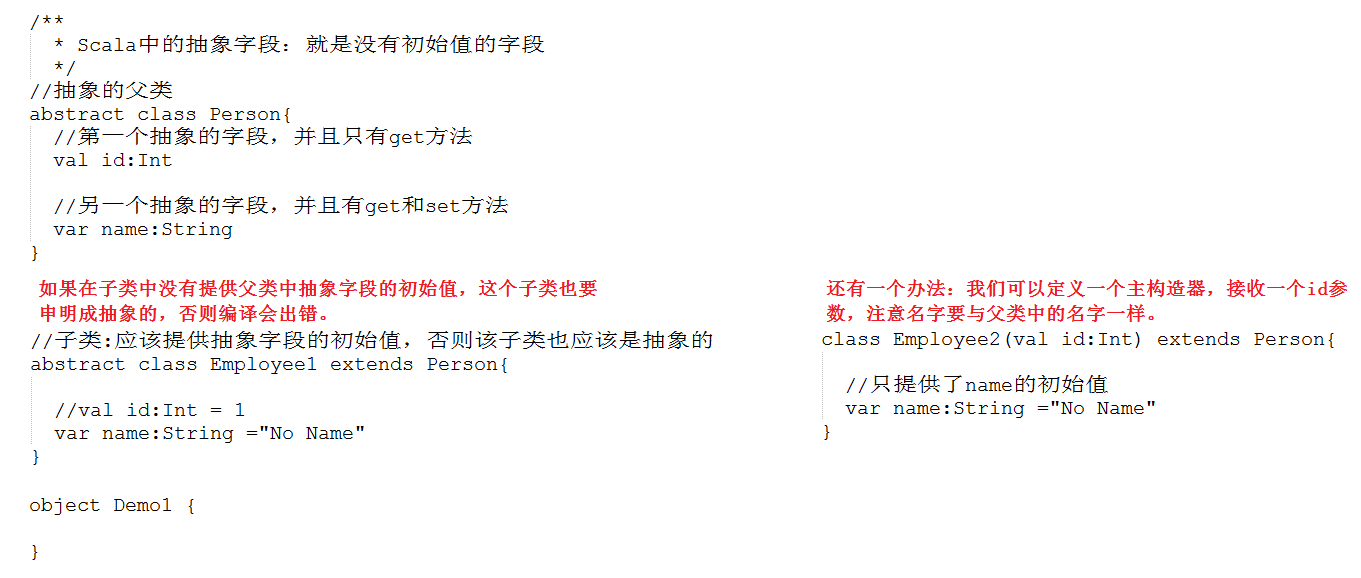


* 案例四：使用抽象类。抽象类中包含抽象方法，抽象类只能用来继承。





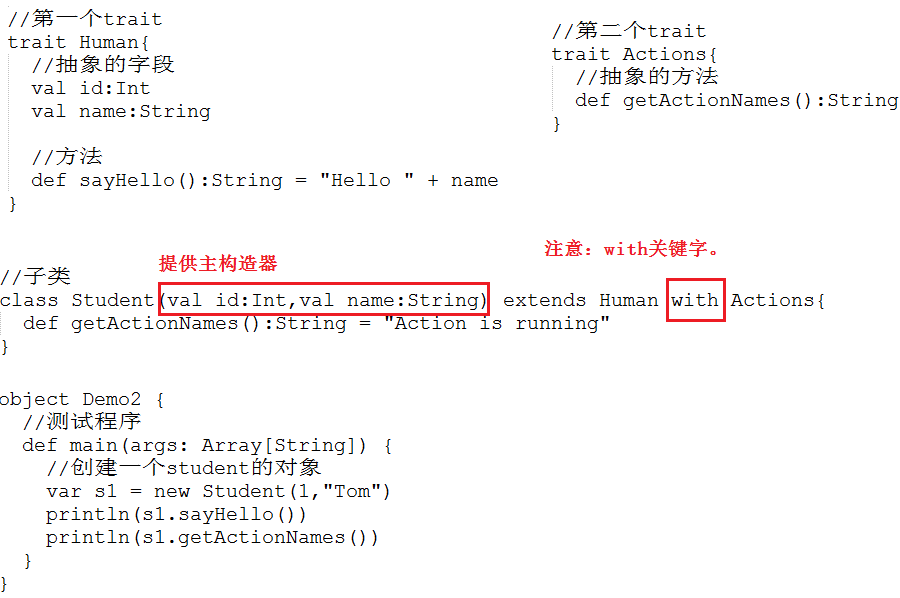
* 案例五：使用抽象字段。抽象字段就是一个没有初始值的字段





### Scala中的trait（特质）

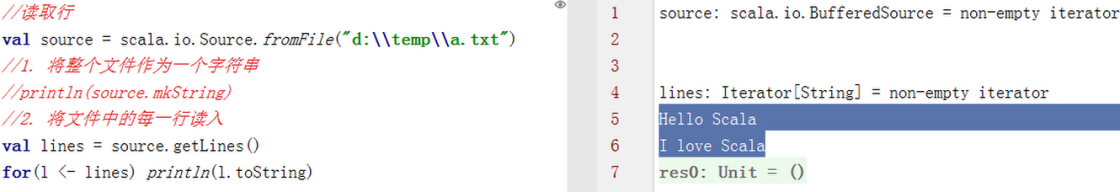
trait就是抽象类。trait跟抽象类最大的区别：**trait支持多重继承**





### Scala中的文件访问

* 读取行

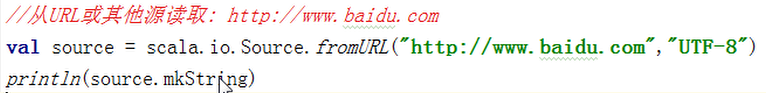


* 读取字符

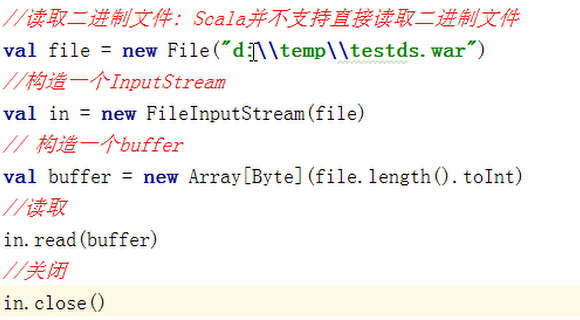


其实这里的source就指向了这个文件中的每个字符。

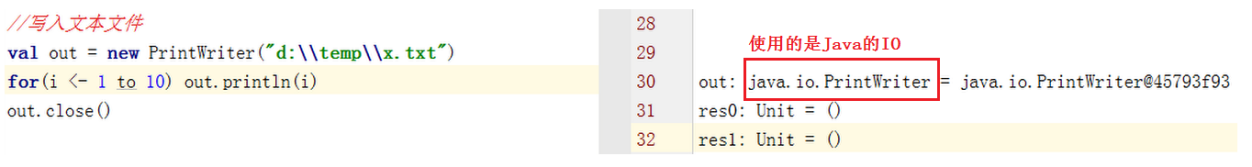
* 从URL或其他源读取：注意指定字符集UTF-8



* 读取二进制文件：Scala中并不支持直接读取二进制，但可以通过调用Java的InputStream来进行读入。



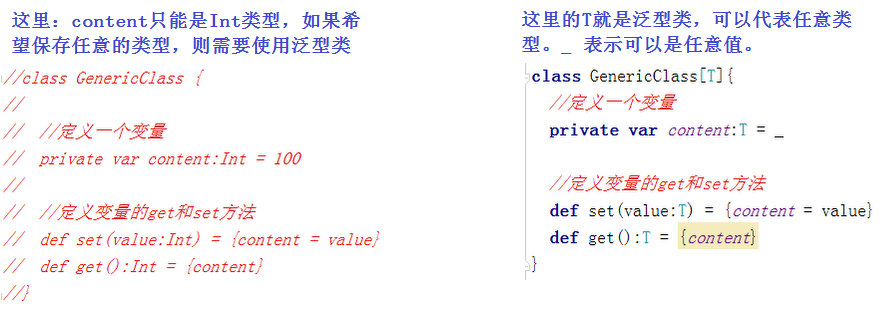
* 写入文本文件



# Scala语言的高级特性

## 什么是泛型类

和Java或者C++一样，类和特质可以带类型参数。在Scala中，使用方括号来定义类型参数



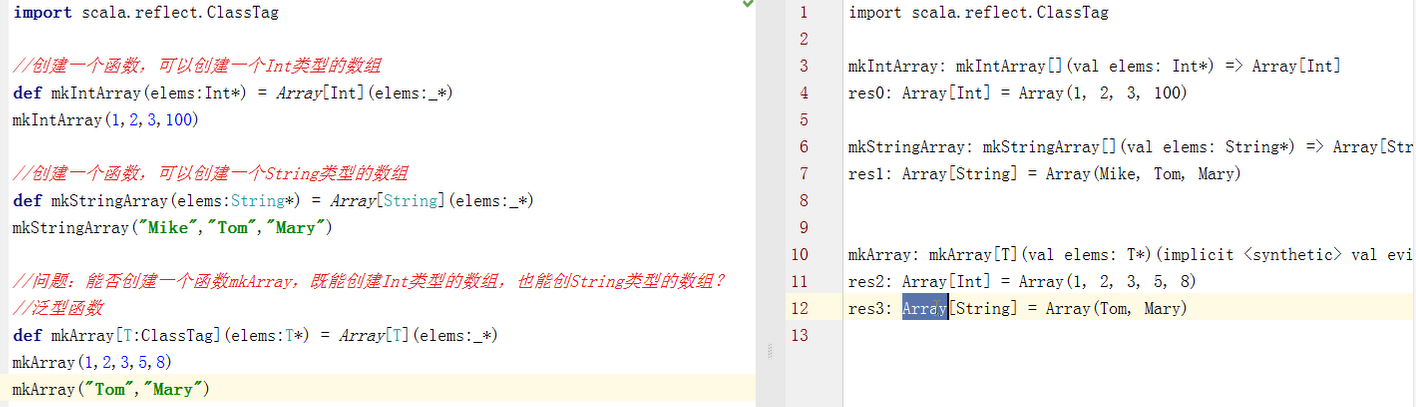
测试程序：



## 什么是泛型函数

函数和方法也可以带类型参数。和泛型类一样，我们需要把类型参数放在方法名之后。

**注意：这里的ClassTag是必须的，表示运行时的一些信息，比如类型。**



## Upper Bounds 与 Lower Bounds

类型的上界和下界，是用来定义类型变量的范围。它们的含义如下：

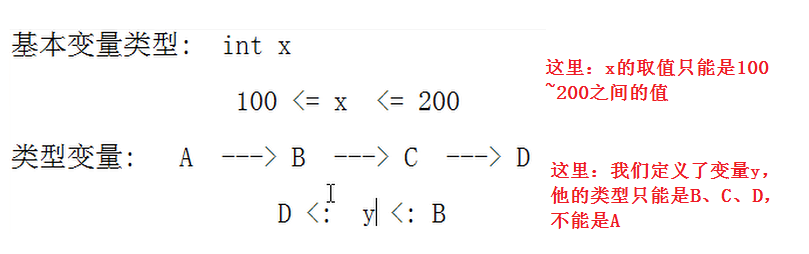
* S <: T

这是类型上界的定义。也就是S必须是类型T的子类（或本身，自己也可以认为是自己的子类。

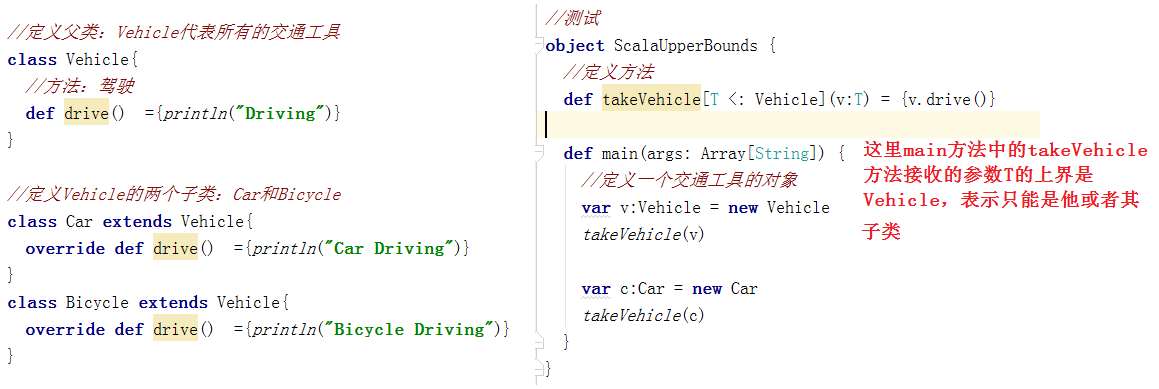
* U >: T

这是类型下界的定义。也就是U必须是类型T的父类(或本身，自己也可以认为是自己的父类)。

一个简单的例子：

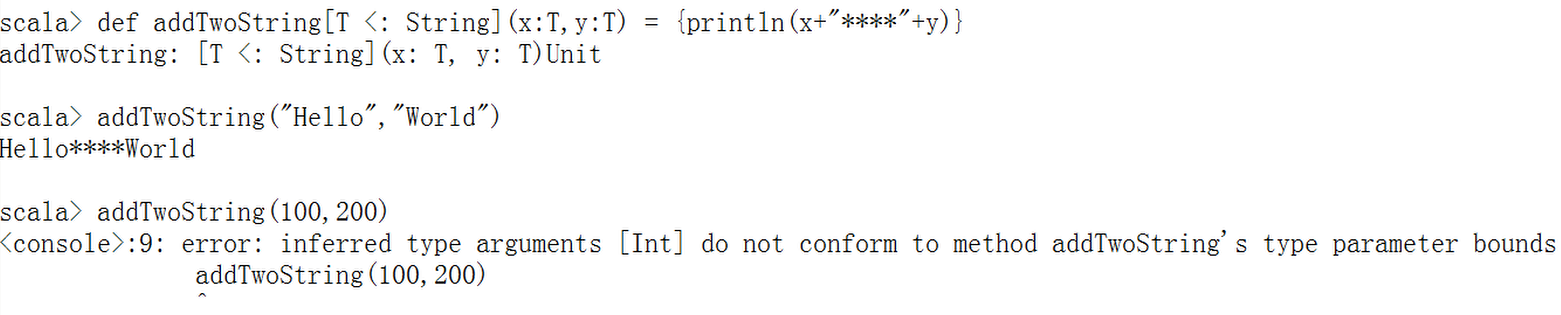


一个复杂一点的例子（上界）：





再来看一个例子：

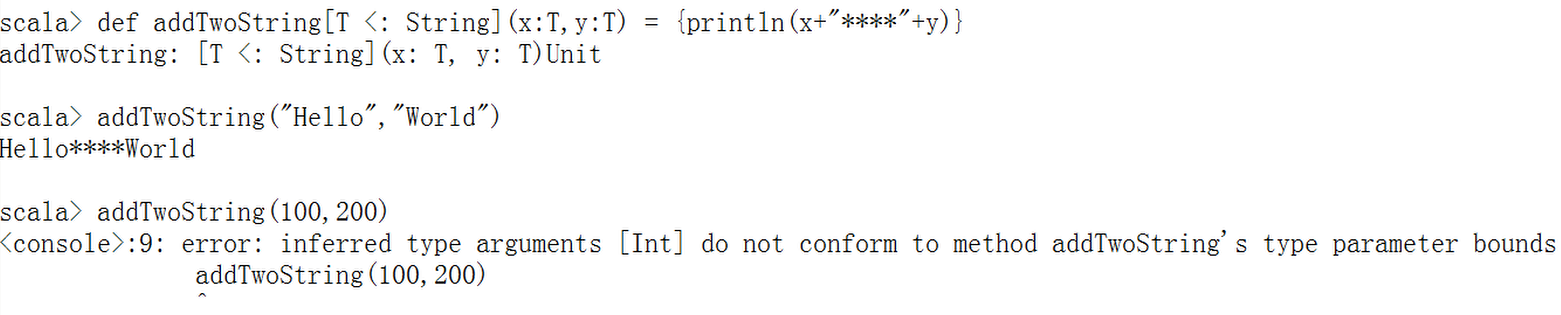


## 视图界定（View bounds）

它比 <: 适用的范围更广，除了所有的子类型，还允许隐式转换过去的类型。用 <% 表示。**尽量使用视图界定，来取代泛型的上界，因为适用的范围更加广泛。**

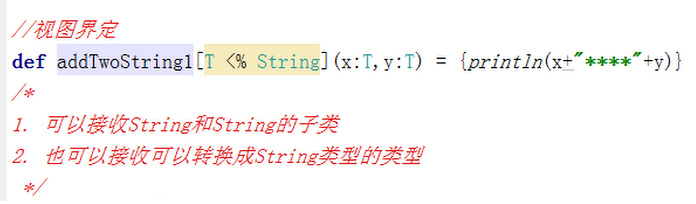
示例：

* 上面写过的一个列子。这里由于T的上界是String，当我们传递100和200的时候，就会出现类型不匹配。

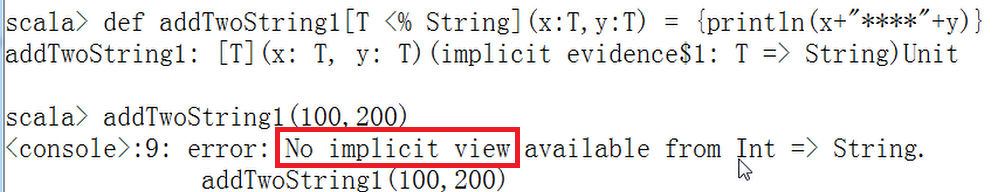


* 但是100和200是可以转成字符串的，所以我们可以使用视图界定让addTwoString方法接收更广泛的数据类型，**即：字符串及其子类、可以转换成字符串的类型**。

注意：使用的是 **<%**

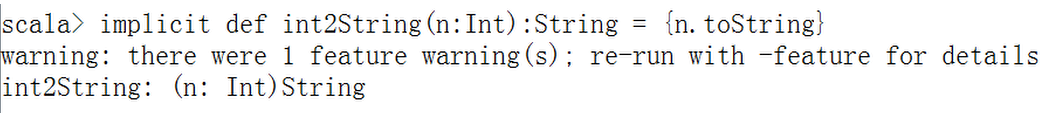


* 但实际运行的时候，会出现错误：

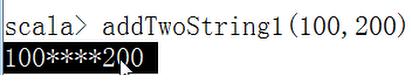


这是因为：Scala并没有定义如何将Int转换成String的规则，所以要使用视图界定，我们就必须创建转换的规则。

* 创建转换规则



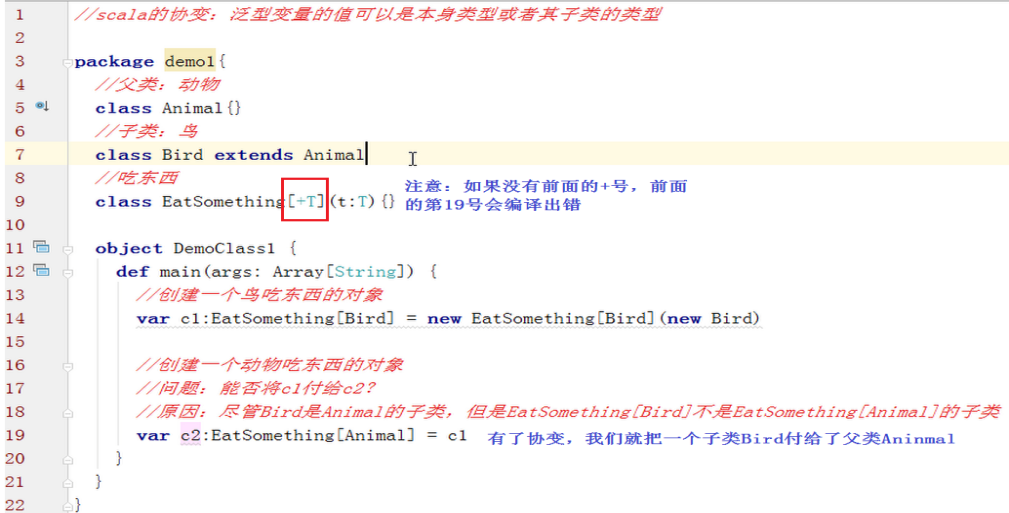
* 运行成功



## 协变和逆变

### 协变：

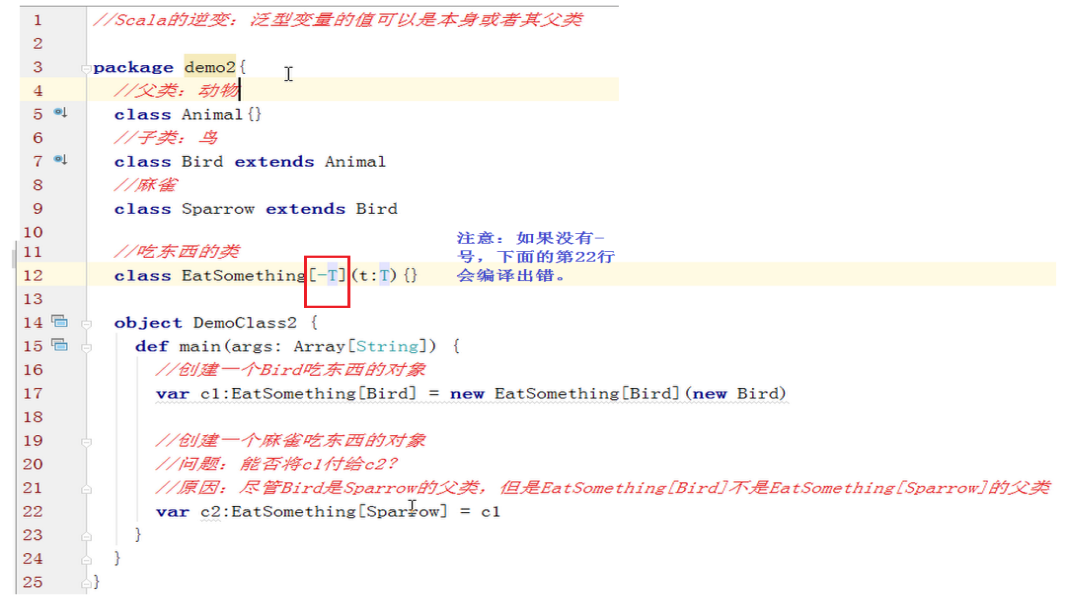
Scala的类或特征的范型定义中，如果在类型参数前面加入**+符号**，就可以使类或特征变为协变了。





### 逆变：

在类或特征的定义中，在类型参数之前加上一个-符号，就可定义逆变范型类和特征了。





**总结一下：**Scala的协变：泛型变量的值可以是**本身类型**或者其**子类**的类型

Scala的逆变：泛型变量的值可以是**本身类型**或者其**父类**的类型

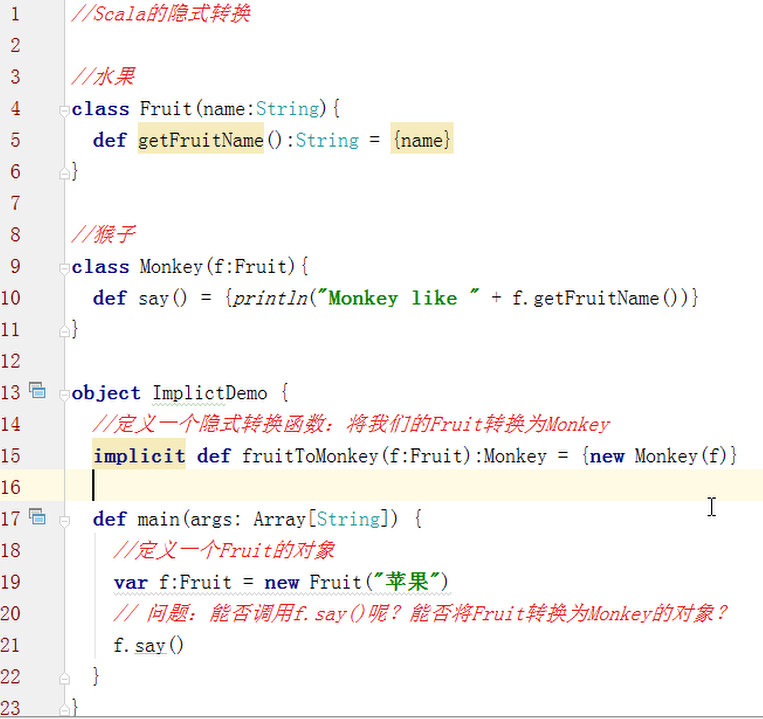
## 隐式转换函数

所谓隐式转换函数指的是以**implicit**关键字申明的带有**单个参数**的函数。

* 前面讲视图界定时候的一个例子：



* 再举一个例子：我们把Fruit对象转换成了Monkey对象





## 隐式参数

使用**implicit**申明的函数参数叫做隐式参数。我们也可以使用隐式参数实现隐式的转换





## 隐式类

所谓隐式类： 就是对类增加**implicit** 限定的类，其作用主要是**对类的功能加强**！

