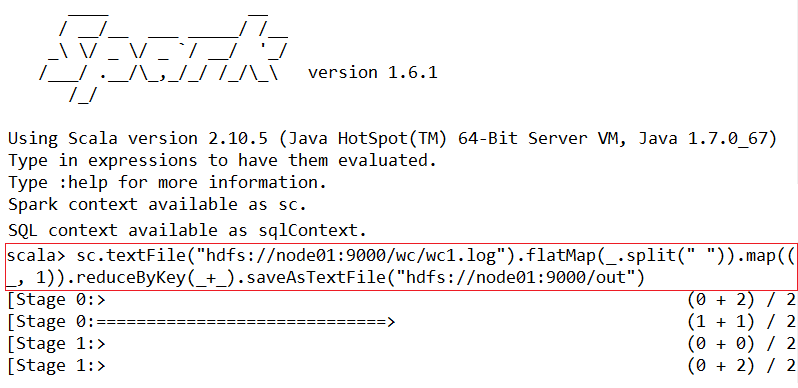
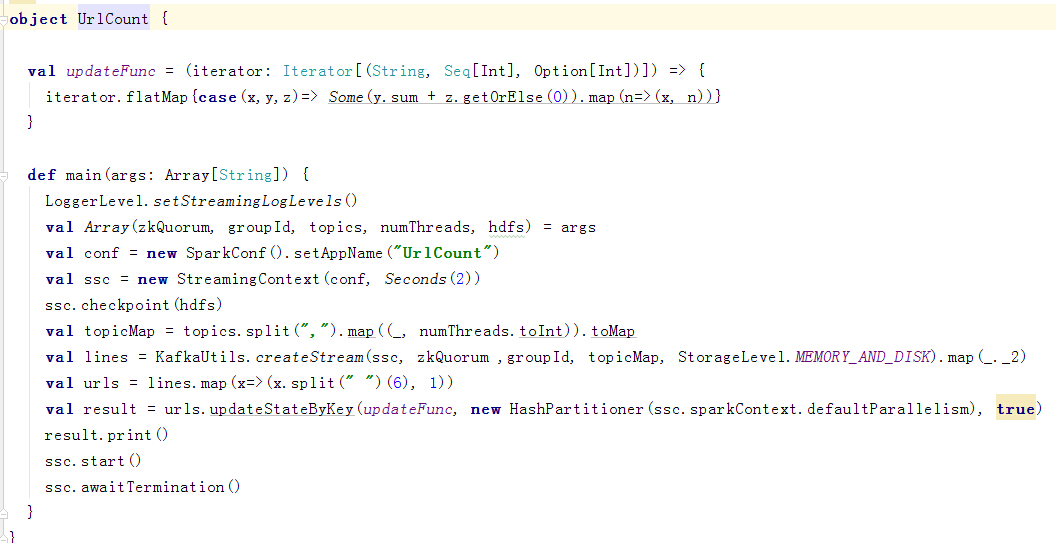
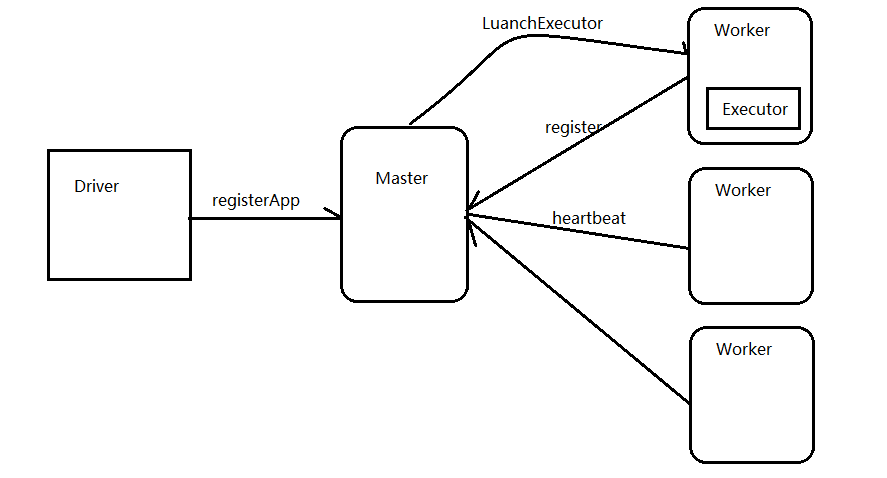
# 课程目标

## 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

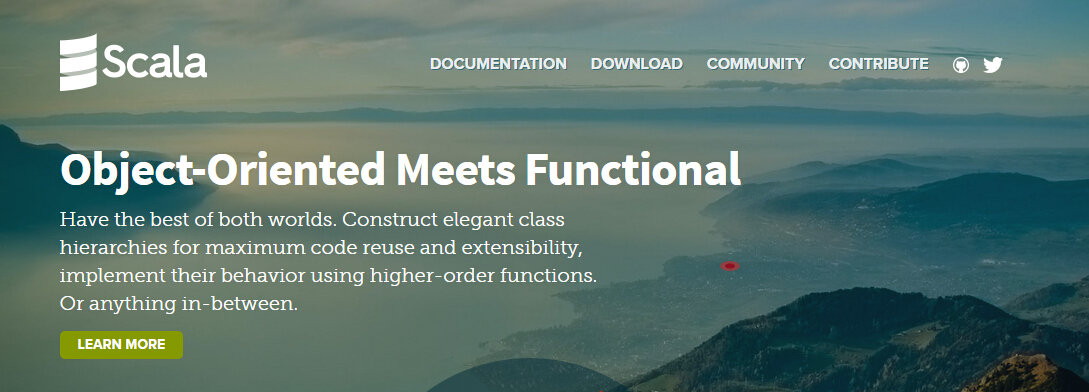


## 目标3：（高级）为阅读Spark内核源码做准备



# 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

1. 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。
2. 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，所以和JRuby,Groovy比起来速度会快很多。
3. 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。



# 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2.10.x编写开发的，所以这里推荐2.10.x版本，下载scala-2.10.6.msi后点击下一步就可以了

### Linux安装Scala编译器

下载Scala地址http://downloads.typesafe.com/scala/2.10.6/scala-2.10.6.tgz然后解压Scala到指定目录

tar -zxvf scala-2.10.6.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/jdk1.7.0\_45

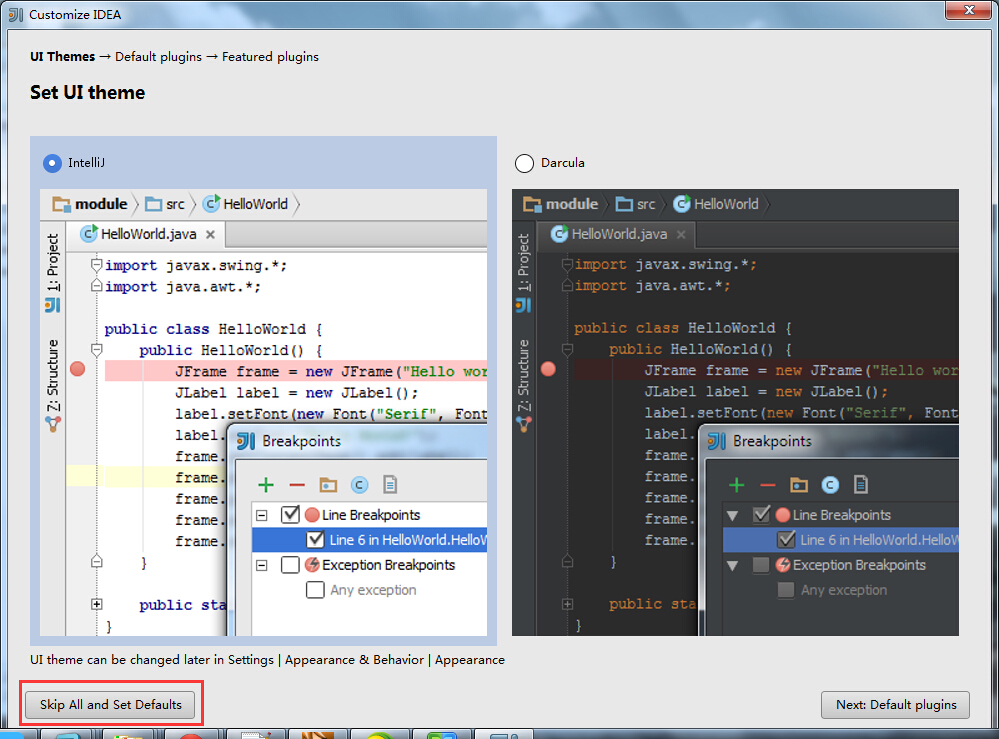
export PATH=$PATH:$JAVA\_HOME/bin:/usr/java/scala-2.10.6/bin

### Scala开发工具安装

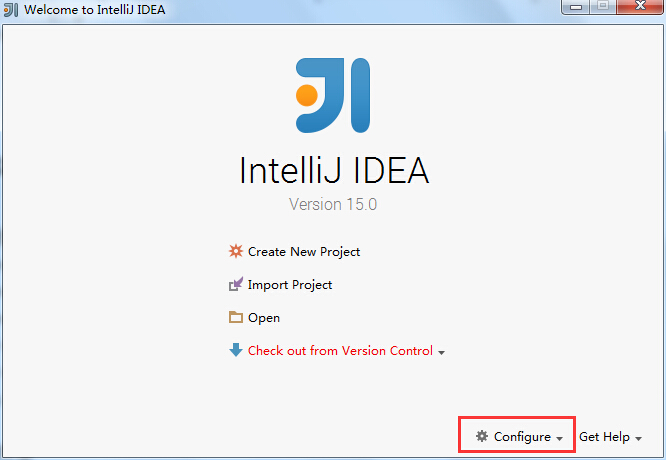
目前Scala的开发工具主要有两种：Eclipse和IDEA，这两个开发工具都有相应的Scala插件，如果使用Eclipse，直接到Scala官网下载即可http://scala-ide.org/download/sdk.html。

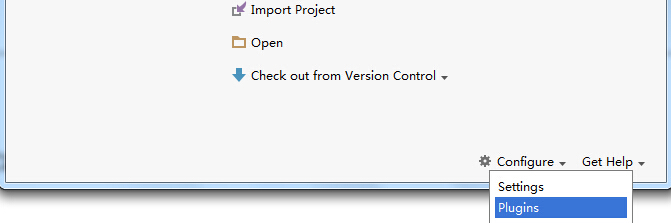
由于IDEA的Scala插件更优秀，大多数Scala程序员都选择IDEA，可以到http://www.jetbrains.com/idea/download/下载社区免费版，点击下一步安装即可，安装时如果有网络可以选择在线安装Scala插件。这里我们使用离线安装Scala插件：

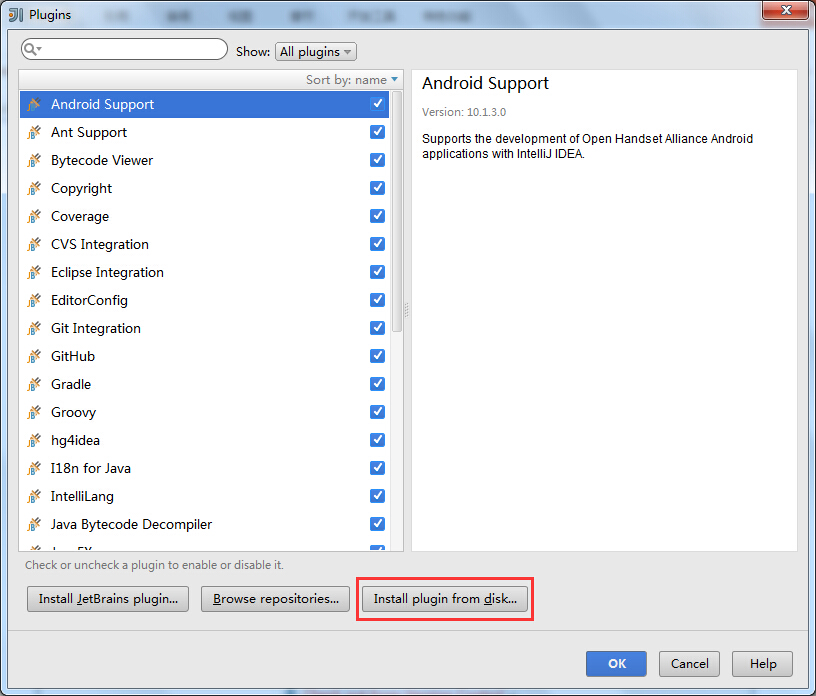
1. 安装IDEA，点击下一步即可。由于我们离线安装插件，所以点击Skip All and Set Defaul
2. 下载IEDA的scala插件，地址http://plugins.jetbrains.com/?idea\_ce

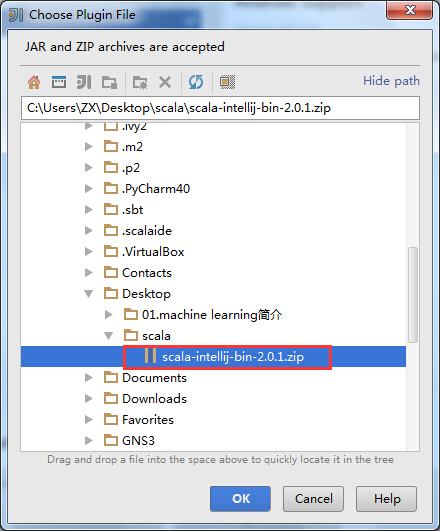
=

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









# Scala基础

## 声明变量

注意：使用lazy定义变量后，只有在调用该变量时才会实例化这个变量的值。而且惰性变量只能是不可变变量。

|  |
| --- |
| **package** com.qf.scala **object** VariableDemo {  **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定义的变量是可变的，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 = **"Scala"** } } |

## 常用类型

Scala和Java一样，有7种数值类型Byte、Char、Short、Int、Long、Float和Double（无包装类型）和一个Boolean类型

## 条件表达式

Scala的的条件表达式比较简洁，例如：

|  |
| --- |
| **package** com.qf.scala **object** ConditionDemo {  **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)  } } |

## 块表达式

|  |
| --- |
| **package** com.qf.scala **object** BlockExpressionDemo {  **def** main(args: Array[String]) {  **val** x = 0  *//在scala中{}中课包含一系列表达式，块中最后一个表达式的值就是块的值  //下面就是一个块表达式*  *//用lazy关键字修饰，该代码块在调用时才运行* **val** result = {  **if** (x < 0){  -1  } **else if**(x >= 1) {  1  } **else** {  **"error"** }  }  *//result的值就是块表达式的结果  println*(result)  } } |

## 循环

在scala中有for循环和while循环，用for循环比较多

for循环语法结构：**for** (i <- 表达式/数组/集合)

|  |
| --- |
| **package** com.qf.scala **object** ForDemo {  **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)  for(i <- 0 until a.length){  println(a(i))  }  for(i <- 0 to a.length-1){  println(a(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或数组,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)  } } |

## 调用方法和函数

Scala中的+ - \* / %等操作符的作用与Java一样，位操作符 & | ^ >> <<也一样。只是有

一点特别的：这些操作符实际上是方法。例如：

a + b 1 to 10

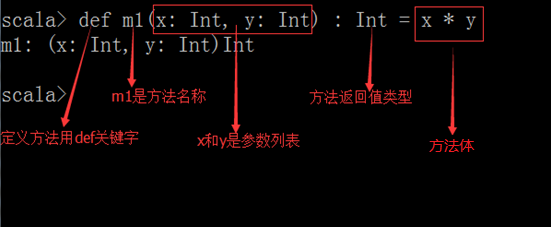
是如下方法调用的简写：

1. +(b) 1.to(10)

a 方法 b可以写成 a.方法(b)

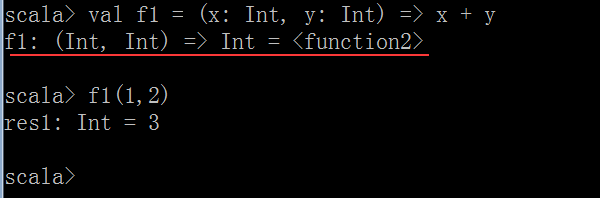
## 定义方法和函数

### 定义方法



方法的返回值类型可以不写，编译器可以自动推断出来，但是对于递归方法，必须指定返回类型

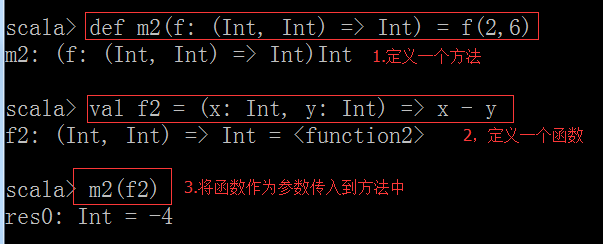
### 定义函数



### 方法和函数的区别

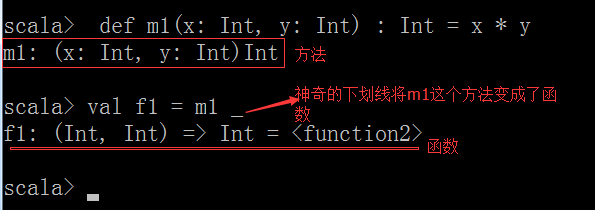
在函数式编程语言中，函数是“头等公民”，**它可以像任何其他数据类型一样被传递和操作**

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



|  |
| --- |
| **package** com.qf.scala **object** MethodAndFunctionDemo {  *//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）



# 数组、映射、元组、集合

## 数组

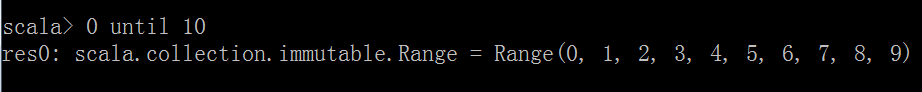
### 定长数组和变长数组

|  |
| --- |
| **package** com.qf.scala  **import** scala.collection.mutable.ArrayBuffer **object** ArrayDemo {   **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：如果不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（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

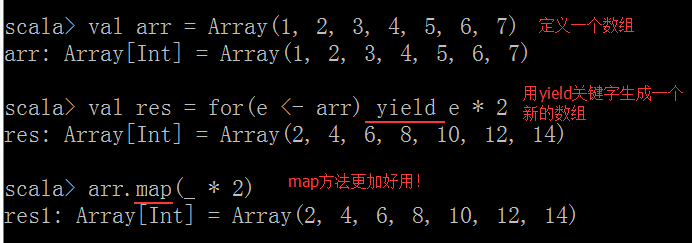
2.好用的until会生成脚标，0 until 10 包含0不包含10

****

|  |
| --- |
| **package** com.qf.scala **object** ForArrayDemo {   **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

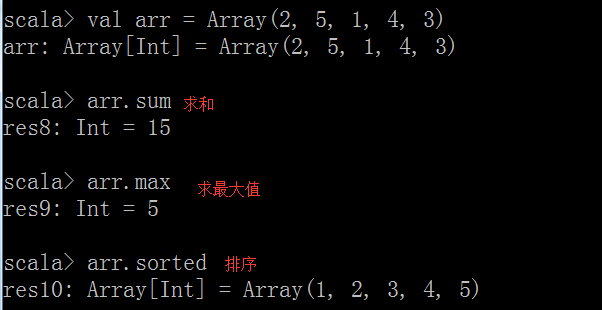
yield关键字将原始的数组进行转换会产生一个新的数组，原始的数组不变



|  |
| --- |
| **package** com.qf.scala **object** ArrayYieldDemo {  **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

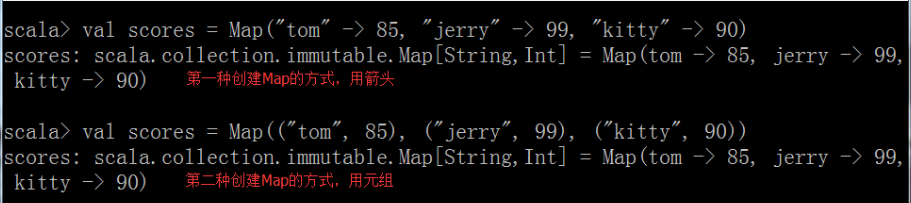
在Scala中，数组上的某些方法对数组进行相应的操作非常方便！



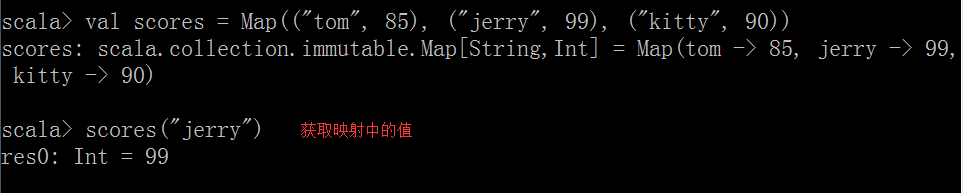
## 映射

在Scala中，把哈希表这种数据结构叫做映射

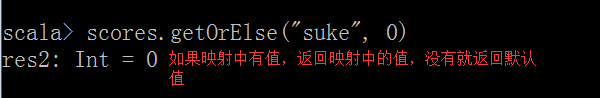
### 构建映射



### 获取和修改映射中的值

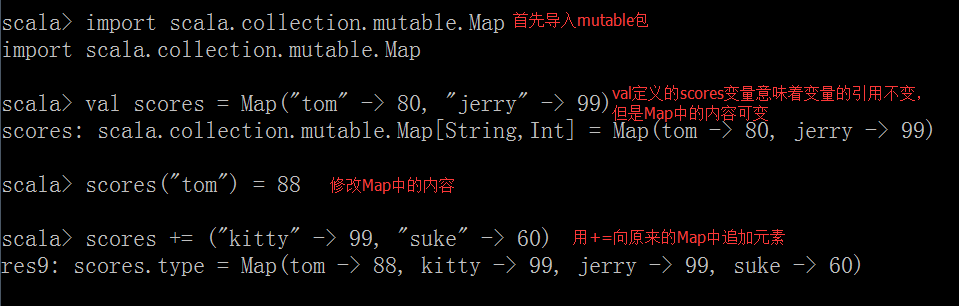


好用的getOrElse



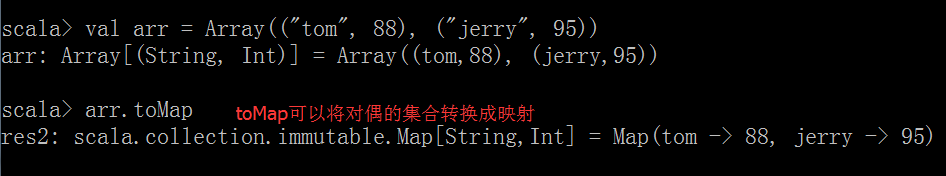
注意：在Scala中，有两种Map，一个是immutable包下的Map，该Map中的内容不可变；另一个是mutable包下的Map，该Map中的内容可变

例子：



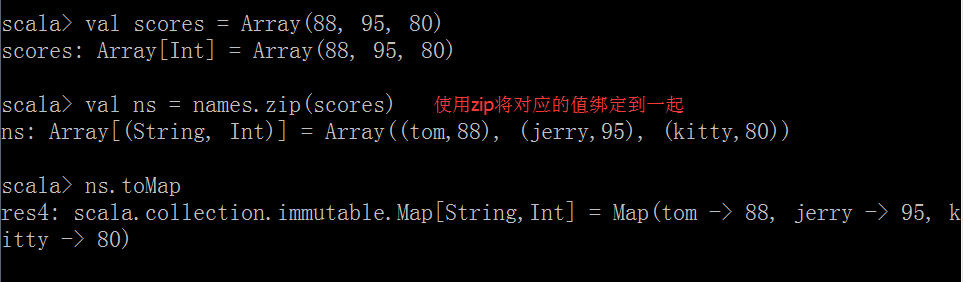
注意：通常我们在创建一个集合是会用val这个关键字修饰一个变量（相当于java中的final），那么就意味着该变量的引用不可变，该引用中的内容是不是可变，取决于这个引用指向的集合的类型

### 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



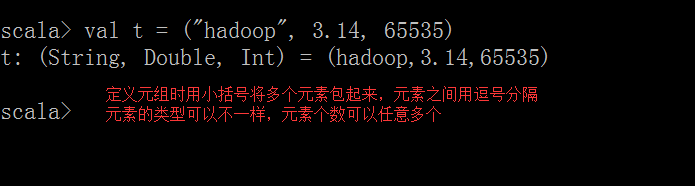
注意：如果两个数组的元素个数不一致，拉链操作后生成的数组的长度为较小的那个数组的元素个数

## 元组

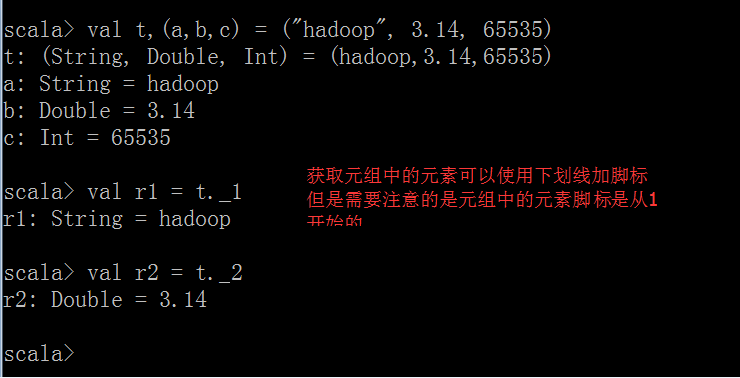
映射是K/V对偶的集合，对偶是元组的最简单形式，元组可以装着多个不同类型的值。

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者扩展元组。

### 创建元组



### 获取元组中的值



## 集合

Scala的集合有三大类：序列Seq、集合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

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))

|  |
| --- |
| **package** com.qf.collect **object** ImmutListDemo {  **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.\_

|  |
| --- |
| **package** com.qf.collect **import** scala.collection.mutable.ListBuffer  **object** MutListDemo **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中的元素追加到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

不可变的Set

|  |
| --- |
| **package** com.qf.collect **import** scala.collection.immutable.HashSet  **object** ImmutSetDemo **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

|  |
| --- |
| **package** com.qf.collect **import** scala.collection.mutable  **object** MutSetDemo **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

|  |
| --- |
| **package** com.qf.collect **import** scala.collection.mutable  **object** MutMapDemo **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*) } |

# 类、对象、继承、特质

Scala的类与Java、C++的类比起来更简洁，学完之后你会更爱Scala！！！

## 类

### 类的定义

|  |
| --- |
| *//在Scala中，类并不用声明为public。*  *//Scala源文件中可以包含多个类，所有这些类都具有公有可见性。* **class** Person {  *//用val修饰的变量是只读属性，有getter但没有setter*  *//（相当与Java中用final修饰的变量）* **val** *id* = **"9527"**  *//用var修饰的变量既有getter又有setter* **var** *age*: Int = 18  *//类私有字段,只能在类的内部使用* **private var** *name*: String = **"唐伯虎"**   *//对象私有字段,访问权限更加严格的，Person类的方法只能访问到当前对象的字段* **private**[**this**] **val** *pet* = **"小强"**  **def sayHi(): Unit ={**  **print(name)**  **}**  } |

### 构造器

注意：主构造器会执行类定义中的所有语句

|  |
| --- |
| */\*\*  \*每个类都有主构造器，主构造器的参数直接放置类名后面，与类交织在一起  \*/* **class** Student(**val** name: String, **val** age: Int){  *//主构造器会执行类定义中的所有语句  println*(**"执行主构造器"**)   **try** {  *println*(**"读取文件"**)  **throw new** IOException(**"io exception"**)  } **catch** {  **case** e: NullPointerException => *println*(**"打印异常Exception : "** + e)  **case** e: IOException => *println*(**"打印异常Exception : "** + e)  } **finally** {  *println*(**"执行finally部分"**)  }   **private var** *gender* = **"male"** *//用this关键字定义辅助构造器* **def this**(name: String, age: Int, gender: String){  *//每个辅助构造器必须以主构造器或其他的辅助构造器的调用开始*  **this**(name, age)  *println*(**"执行辅助构造器"**)  **this**.*gender* = gender  } } |

|  |
| --- |
| */\*\*  \*构造器参数可以不带val或var，如果不带val或var的参数至少被一个方法所使用，*  *\*那么它将会被提升为字段  \*/*  *//在类名后面加private就变成了私有的* **class** Queen **private**(**val** name: String, prop: Array[String], **private var** age: Int = 18){    *println*(prop.size)   *//prop被下面的方法使用后，prop就变成了不可变得对象私有字段，等同于private[this] val prop  //如果没有被方法使用该参数将不被保存为字段，仅仅是一个可以被主构造器中的代码访问的普通参数* **def** description = name + **" is "** + age + **" years old with "** + prop.toBuffer }  **object** Queen{  **def** main(args: Array[String]) {  *//私有的构造器，只有在其伴生对象中使用* **val** q = **new** Queen(**"hatano"**, *Array*(**"蜡烛"**, **"皮鞭"**), 20)  *println*(q.description())  } } |

## 对象

### 单例对象

在Scala中没有静态方法和静态字段，但是可以使用object这个语法结构来达到同样的目的

1. 存放工具方法和常量
2. 高效共享单个不可变的实例
3. 单例模式

|  |
| --- |
| **package** com.qf.scala  **import** scala.collection.mutable.ArrayBuffer **object** SingletonDemo {  **def** main(args: Array[String]) {  *//单例对象，不需要new，用【类名.方法】调用对象中的方法* **val** session = SessionFactory.*getSession*()  println(session)  } } **object** SessionFactory{  *//该部分相当于java中的静态块* **var** *counts* = 5  **val** *sessions* = **new** ArrayBuffer[Session]()  **while**(*counts* > 0){  *sessions* += **new** Session  *counts* -= 1  }   *//在object中的方法相当于java中的静态方法* **def** getSession(): Session ={  *sessions*.remove(0)  } }  **class** Session{  } |

### 伴生对象

在Scala的类中，与类名相同并且用object修饰的对象叫做伴生对象，类和伴生对象之间可以相互访问私有的方法和属性

|  |
| --- |
| **package** com.qf.scala **class** Dog {  **val** *id* = 1  **private var** *name* = **"二哈"   def** printName(): Unit ={  *//在Dog类中可以访问伴生对象Dog的私有属性  println*(Dog.*CONSTANT* + *name* )  } }  */\*\*  \* 伴生对象  \*/* **object** Dog {   *//伴生对象中的私有属性* **private val** *CONSTANT* = **"汪汪汪 : "   def** main(args: Array[String]) {  **val** p = **new** Dog  *//访问私有的字段name* p.*name* = **"123"** p.printName()  } } |

### apply方法

通常我们会在类的伴生对象中定义apply方法，当遇到类名(参数1,...参数n)时apply方法会被调用

|  |
| --- |
| **package** com.qf.scala  **object** ApplyDemo {  **def** main(args: Array[String]) {  *//调用了Array伴生对象的apply方法  //def apply(x: Int, xs: Int\*): Array[Int]  //arr1中只有一个元素5* **val** arr1 = *Array*(5)  *println*(arr1.toBuffer)   *//new了一个长度为5的array，数组里面包含5个null* **var** arr2 = **new** Array(5)  } } |

### 应用程序对象

Scala程序都必须从一个对象的main方法开始，可以通过扩展App特质，不写main方法。

|  |
| --- |
| **package** com.qf.scala **object** AppObjectDemo **extends** App{  *//不用写main方法  println*(**"I love you Scala"**) } |

## 继承

### 扩展类

在Scala中扩展类的方式和Java一样都是使用extends关键字

### 重写方法

在Scala中重写一个非抽象的方法必须使用override修饰符

### 类型检查和转换

|  |  |  |
| --- | --- | --- |
|  | **Scala** | **Java** |
| 判断对象是否为C类型的实例 | obj.isInstanceOf[C] | obj instanceof C |
| 将对象类型强制转换为C类型 | obj.asInstanceOf[C] | (C)obj |
| 获取类型T的Class对象 | classOf[C] | C.class |

### 超类的构造

|  |
| --- |
| **package** com.qf.scala **object** ClassDemo {  **def** main(args: Array[String]) {  *//val h = new Human  //println(h.fight)* } }  **trait** Flyable{  **def** fly(): Unit ={  *println*(**"I can fly"**)  }   **def** fight(): String }  **abstract class** Animal {  **def** run(): Int  **val** name: String }  **class** Human **extends** Animal **with** Flyable{   **val** *name* = **"a"** *//打印几次"ABC"?* **val** *t1*,*t2*,(*a*, *b*, *c*) = {  *println*(**"ABC"**)  (1,2,3)  }   *println*(*a*)  *println*(*t1*.\_1)   *//在Scala中重写一个非抽象方法必须用override修饰* **override def** fight(): String = {  **"fight with 棒子"** }  *//在子类中重写超类的抽象方法时，不需要使用override关键字，写了也可以* **def** run(): Int = {  1  } } |

# 模式匹配和样例类

Scala有一个十分强大的模式匹配机制，可以应用到很多场合：如switch语句、类型检查等。

并且Scala还提供了样例类，对模式匹配进行了优化，可以快速进行匹配

## 匹配字符串

|  |
| --- |
| **package** com.qf.cases **import** scala.util.Random **object** CaseDemo01 **extends** App{  **val** *arr* = *Array*(**"YoshizawaAkiho"**, **"YuiHatano"**, **"AoiSola"**)  **val** *name* = *arr*(Random.nextInt(*arr*.length))  *name* **match** {  **case "YoshizawaAkiho"** => *println*(**"吉泽老师..."**)  **case "YuiHatano"** => *println*(**"波多老师..."**)  **case** \_ => *println*(**"真不知道你们在说什么..."**)  } } |

## 匹配类型

|  |
| --- |
| **package** com.qf.cases **import** scala.util.Random **object** CaseDemo02 **extends** App{ *//val v = if(x >= 5) 1 else if(x < 2) 2.0 else "hello"* **val** *arr* = *Array*(**"hello"**, 1, 2.0, CaseDemo)  **val** *v* = *arr*(Random.nextInt(4))  *println*(*v*)  *v* **match** {  **case** x: Int => *println*(**"Int "** + x)  **case** y: Double **if**(y >= 0) => *println*(**"Double "**+ y)  **case** z: String => *println*(**"String "** + z)  **case** \_ => **throw new** Exception(**"not match exception"**)  } } |

**注意：**case y: Double if(y >= 0) => ...

模式匹配的时候还可以添加守卫条件。如不符合守卫条件，将掉入case \_中

## 匹配数组、元组

|  |
| --- |
| **package** com.qf.cases **object** CaseDemo03 **extends** App{   **val** *arr* = *Array*(1, 3, 5)  *arr* **match** {  **case** *Array*(1, x, y) => *println*(x + **" "** + y)  **case** *Array*(0) => *println*(**"only 0"**)  **case** *Array*(0, \_\*) => *println*(**"0 ..."**)  **case** \_ => *println*(**"something else"**)  }   **val** *lst* = *List*(3, -1)  *lst* **match** {  **case** 0 *:: Nil* => *println*(**"only 0"**)  **case** x *::* y *:: Nil* => *println*(**s"x: $**x **y: $**y**"**)  **case** 0 *::* tail => *println*(**"0 ..."**)  **case** \_ => *println*(**"something else"**)  }   **val** *tup* = (2, 3, 7)  *tup* **match** {  **case** (1, x, y) => *println*(**s"1, $**x **, $**y**"**)  **case** (\_, z, 5) => *println*(z)  **case** \_ => *println*(**"else"**)  } } |

注意：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))

## 样例类

在Scala中样例类是一种特殊的类，可用于模式匹配。case class是多例的，后面要跟构造参数，case object是单例的

|  |
| --- |
| **package** com.qf.cases **import** scala.util.Random  **case class** SubmitTask(id: String, name: String) **case class** HeartBeat(time: Long) **case object** CheckTimeOutTask  **object** CaseDemo04 **extends** App{  **val** *arr* = *Array*(CheckTimeOutTask, *HeartBeat*(12333), *SubmitTask*(**"0001"**, **"task-0001"**))   *arr*(Random.nextInt(*arr*.length)) **match** {  **case** *SubmitTask*(id, name) => {  *println*(**s"$**id**, $**name**"**)  }  **case** *HeartBeat*(time) => {  *println*(time)  }  **case** CheckTimeOutTask => {  *println*(**"check"**)  }  } } |

## Option类型

在Scala中Option类型样例类用来表示可能存在或也可能不存在的值(Option的子类有Some和None)。Some包装了某个值，None表示没有值

|  |
| --- |
| **package** com.qf.cases  **object** OptionDemo {  **def** main(args: Array[String]) {  **val** map = *Map*(**"a"** -> 1, **"b"** -> 2)  **val** v = map.get(**"b"**) **match** {  **case** *Some*(i) => i  **case** None => 0  }  *println*(v)  *//更好的方式* **val** v1 = map.getOrElse(**"c"**, 0)  *println*(v1)  } } |

## 偏函数

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

|  |
| --- |
| **package** com.qf.cases **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"**))  } } |