## 基础说明

val 说明是引用不可变

例如 val a=123 或者 val a:Int = 123

a = 555 就会报错



var 说明是可以变的

Scala的7种类型

Byte，Char ，Short， Int， Long ，Float， Double

混合类型的表达式

**val** y = **if**(x>1) 1 **else** -1

返回1或者-1

**val** y = **if**(x>1) 1

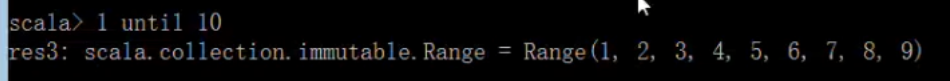
有可能为AnyVal

AnyVal 相当于java里的Object

1 to 10 或 1.to(10)



1 until 10 输出1-9



使用循环

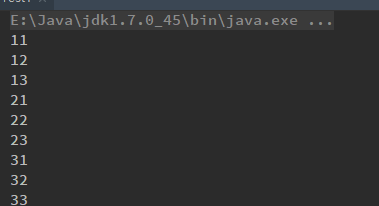
**for** (i<- 1 to 10){  
 *println*(i)  
}

数组循环

**val** arr = *Array*("java","scala","C#")  
**for**(i <- arr){  
 *println*(i)  
}

高级for循环

**for**(i <- 1 to 3;j <- 1 to 3 ) *println* (i+""+j)



**val** arrs = **for**(i <- 1 to 3;j <- 1 to 3 ) **yield** (i+""+j)

关键字yield 会把每次循环出来值封装到一个集合中 并且赋值给arrs 类型为vector

for 循环中的 yield 会把当前的元素记下来，保存在集合中，循环结束后将返回该集合。Scala 中 for 循环是有返回值的。如果被循环的是 Map，返回的就是  Map，被循环的是 List，返回的就是 List，以此类推。

Scala +-\*/ 中每个 操作符都是一个方法 1+2 等价于 1.+（2）

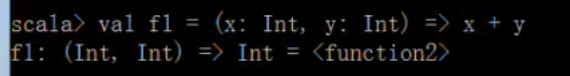
2下shift 可以在idea 中查找所有该名称的类

方法声明(方法和函数不一样)

**def** method(x:Int, y:Int): Int = x+y

函数声明

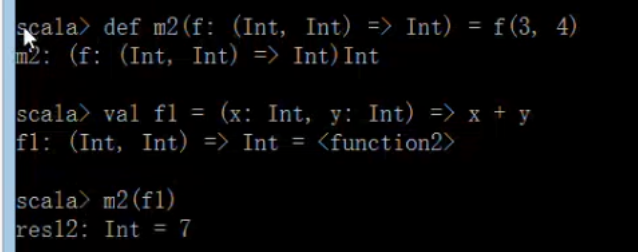
**val** *f1* =(x: Int, y: Int) =>x+y



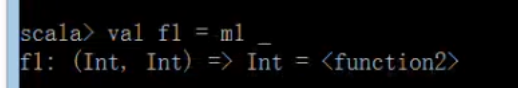
Function2 代表2个入参 Function1代表1个入参

**def** method(f:(Int, Int) => Int) = f(3,4)  
**val** *f1* =(x: Int, y: Int) =>x+y

把函数当做某个值传到方法里



方法转换成函数

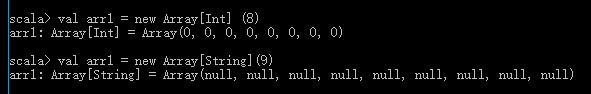


通过下划线 \_ 来转 val f1 = m1 \_

传到 时候也会 隐式转换

# 集合

定长数组



通过toBuffer 缓冲出来





arr（2） 就类似于java里面的 arr[2]

变长数组

**import** scala.collection.mutable.ArrayBuffer

**val** arr3 = ArrayBuffer[Int]()

插入元素

**val** arr3 = ArrayBuffer[Int]()  
 //追加 一个元素1  
 arr3+=1  
 //追加 一个元组  
 arr3+=(2,3,4)  
 // 追加 一个定量数组  
 arr3 ++=*Array*(5,6)  
 // 追加一个变长数组  
 arr3 ++=ArrayBuffer(7,8)

// 第一个参数是下标 后面是要在第0个参数前面插入的东西 插入的顺序也是-1,0 不变 比如原来 是123 插入-1,0 就会变成 -1,0,1，2，3  
 arr3.insert(0,-1,0)  
 // 第一个参数是下标 后面表示要在第2个参数开始删除 2个参数 就是要删除 下标为0和1的参数  
 arr3.remove(0,3)

遍历数组

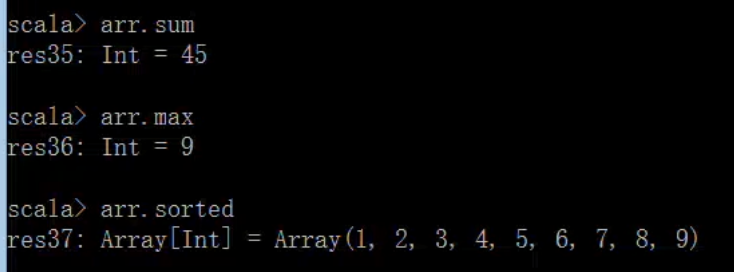
**val** arr = *Array*(1,2,3,4,5,6)  
  
// 遍历输出 arr里的东西  
**for** (i<-arr) *println*(i)  
// 根据下标取   
**for**(i <-0 until arr.length) *println*(arr(i))  
// 根据下标取 倒着来  
**for**(i <-(0 until arr.length).reverse) *println*(arr(i))

产生一个新的数组

**val** arr = *Array*(1,2,3,4,5,6)  
// 产生一个新的数组  
**val** res = **for** (i <- arr) **yield** i\*10

总计 最大值 最小值

排序



映射

// 通过->  
**val** map1 = *Map*("scala" -> 1, "java"->2)  
//通过元组的方式  
**val** map2 = *Map*(("scala",1),("java",2))

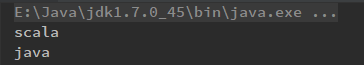
获取Map中的值

// 通过->  
**val** map1 = *Map*("scala" -> 1, "java"->2)  
//通过元组的方式  
**val** map2 = *Map*(("scala",1),("java",2))  
  
// 获取值 如果没有的对应的 就会报错  
map1("scala")  
// 第一个参数是key值 第二个是如果没有第一个key的值 那么就返回第二个默认的值  
map1.getOrElse("scala",-1)  
// 修改  
map1("scala") = 3

元组 元组的下标是从1开始的

创建元组方法1

// 创建一个元组 元组里可以有元组   
**val** t = ("scala",100L,3.14,("java",1))  
// 获取元组里 第一个值  
*println*(t.\_1)  
// 获取元组里 第四个值(他也是一个元组) 然后再获取这个元组里第一个值  
*println*(t.\_4.\_1)



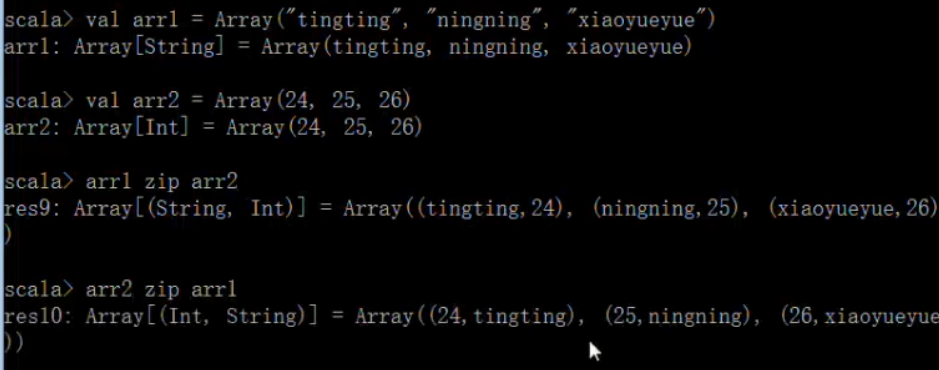
创建元组方法2

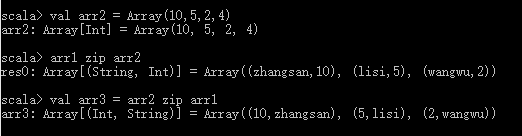
// 在里面的时候直接给这些元素给一个名称  
**val** t,(a,b,c,d) = ("scala",100L,3.14,("java",1))  
// 直接输出a  
*println*(a)

// 创建一个元组数组  
**val** arr = *Array*(("zhangsan", 23), ("lisi", 30), ("wangwu", 50), ("zhangliu", 10))  
// 把他装换成map  
arr.toMap

// 拉链操作  
**val** arr1 = *Array*("tingting", "ninging", "zhangsan")  
**val** arr2 = *Array*(20, 45, 10)  
// 前面的数组是key 后面的是value 等价于 arr1.zip(arr2)  
arr1 zip arr2

产生的是元组数组





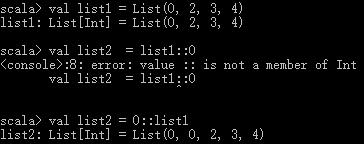
取2个数组短的长度

## Scala集合：Seq(序列) Set(集) Map（映射 ）

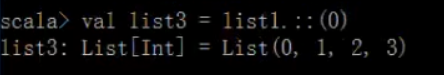
### List

List 是定长的 ListBuffer是变长的

把某一数添加到一个List的前面



使用：：相当于String里的+ 只能往前面增加







把某一数据添加到List的后面



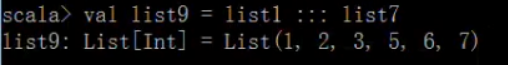
把2个数组合并 ++前面的list元素 最后还是在前面 ++ 后面的list元素还是在后面



等价于

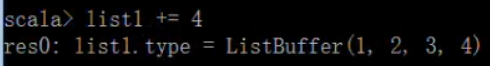


等价于



如果要想把一个数组插到某个数组前面

变长 ListBuffer

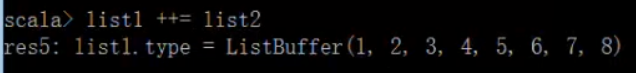


往list1 里追加4

往list1 里追加5



把list2 的内容追加到list里 元素顺序是list1 然后再list2



### Set

**import** scala.collection.immutable.HashSet

HashSet 不可变的set

**val** set1 = **new** HashSet[Int]()  
**val** set2 = set1 + 1  
**val** set3 = set2 ++ *Set*(2,3,4)

通过 + 来增加 里面的元素

通过 ++ 来增加set

HashSet 可变的

**import** scala.collection.mutable.HashSet

**val** set1 = **new** HashSet[Int]()  
set1 += 1  
set1.add(3)  
set1 ++=*Set*(4,5,6) // set添加的时候有一个随机性 有去重的功能  
set1 -=1 // 把1删除   
set1.remove(3) // 把3移除掉

Map

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

.\_ 相当于 java里的 .\*

可变的

HashMap

**val** map1 = **new** HashMap[String,Int]()  
map1("scala") = 1  
map1 += (("java",2))  
map1 += (("python",3),("C",4))  
map1 +=(("C++"->5))  
map1.put("C#",6)  
  
map1 -="java"  
map1.remove("java")

Lazy 懒性变量关键字

会实现延迟加载，lazy 惰性变量 只能是不可变的 变量 val

并且只有在调用 惰性变量时才会去实例化这个变量

**lazy val** property = *init*() // 用lazy修饰  
*println*("after init()")  
*println*(property)

例如上面的 在初始化的时候调用了 init

只有正真的调用了这个变量的时候才会去执行他的实例化语句例如

property = *init*()

函数式编程

隐式转换和隐式参数

作用： 能丰富先有类库的功能，对垒的方法进行增强

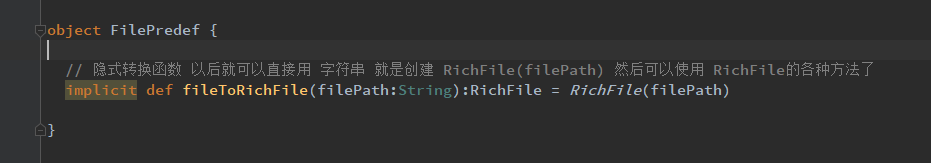
隐式转换函数 ： 以implicit 关键字来声明

开发的时候 不太用 但是源码里有

装饰模式 和 门面模式

隐式转换函数 只能是引用什么隐式函数就调用什么函数

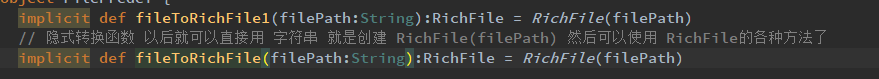
比如 创建一个隐式函数

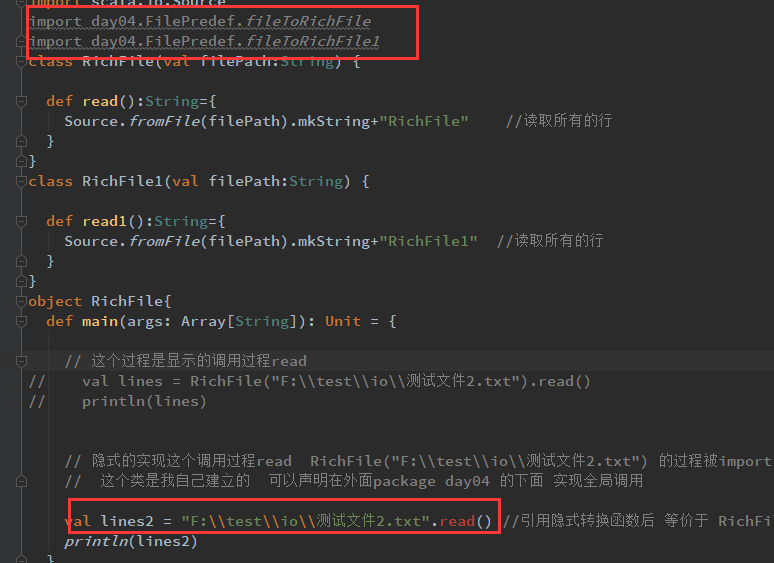


相当于是 一个字符串对象 就是调用 比如RichFile的类里有read方法

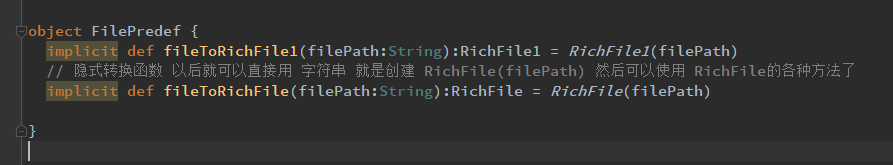
那么就可以“123”.read 是与 new RichFile（“123”）.read 一样的

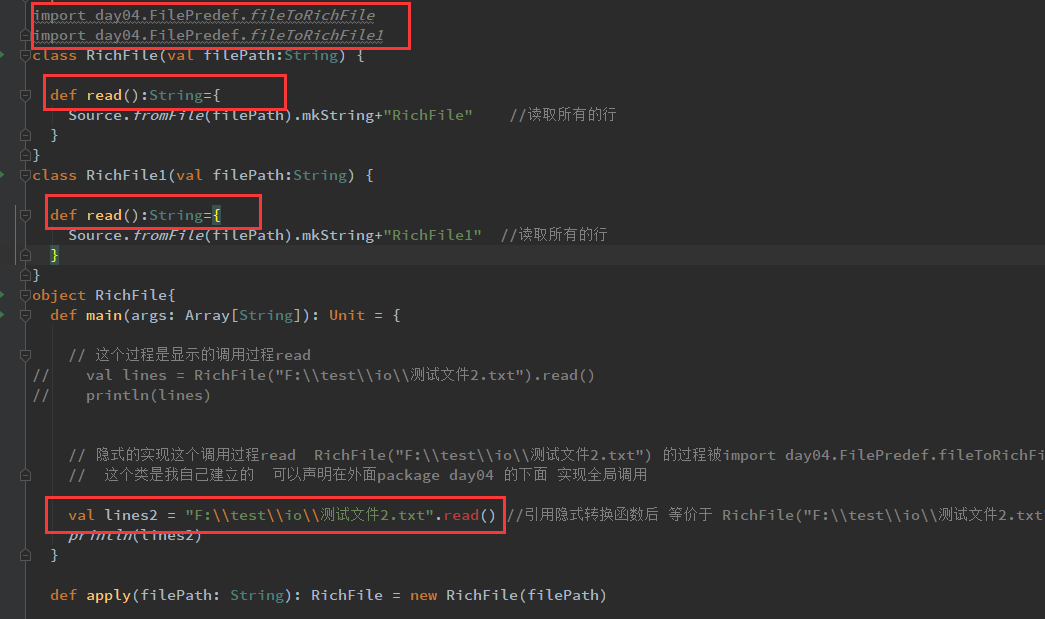
但是如果 另外一个隐式函数 返回的参数 也是RichFile 那么 隐式调用就会报错





如果隐式调用返回的对象 之间有名称相同的方法 那么这些方法也是不可以隐式调用的





泛型

[ B<:A] upperBound 上界 ：B类型的上界是A类型 即B类型的父类是A类型

[ B>:A] lowerBound 下界 ：B类型的上界是A类型 即B类型的子类是A类型

[B<%A] viewBound 表示B类型要转换成A类型，即需要一个隐式转换函数

[B : A] ContextBound 需要一个隐式转换的值

[-A,+B]

[-A] 逆变 ，作为参数类型，如果A是T的子类，那么C[T]是C[A]的子类

[+B] 协变 ，作为返回类型，如果A是T的子类，那么C[A]是C[T]的子类