Scala入门教程：<http://blog.csdn.net/mapdigit/article/details/21878083>

语法：

Scala中所有的基本数据类型也是对象。

**Val：定义常量**

**Var: 定义变量**

Scala可以根据赋值的内容推算出变量的类型

## 函数定义

def 函数名称（参数名：参数类型，参数名：参数类型）：返回值类型

{

//scala中每个表达式都有返回值，因此函数返回结果无需用return

//回溯函数（函数调用自身）还是需要指明返回结果类型的。

//返回值为为Unit时，相当于void

}

## 循环控制

Scala不支持++i,i++ 运算符，因此需要使用i+＝1 来加一

Scala访问数组的语法是使用()而非[].

Foreach语句：

args.foreach(arg => println(arg))

args.foreach(println)

for语句：for(arg <- args)

## 类型参数化数组

例如：

val greetStrings = new Array[String](3)

greetStrings(0)="Hello"

greetStrings(1)=","

greetStrings(2)="world!\n"

for(i <- 0 to 2)

print(greetStrings(i))

**scala的规则：若方法只有一个参数，调用的时候就可以省略点及括号**，例如: 0 to 2，实际上是调用(0).to(2)这个函数。

这里的greetString(i)，实际上是调用greetString.apply(i)方法。这是scala的通用法则：任何对于对象值参数的应用都将转换为对apply方法的调用。

同样，***当对带有扩号并包括一到若干参数的变量赋值时，编译器将使用对象的update方法对括号里的参数（索引值）和等号右边的对象执行调用***。例如: greetStrings(0)="Hello",将转化为：greetStrings.update(0,"Hello")。

Scala没有操作符重载，因为scala没有传统意义上的操作符。+、-、\*、/作为字符串都可以当做方法名，例如：1+2，其实是1.+(2)，调用了1的+方法。

上述代码等同于：

Val greetStrings=new Array[String](3)

greetStrings.update(0,”Hello”)

greetStrings.update(1,”,”)

greetStrings.update(2,”world!\n”)

for(i <- 0.to(2))

println(greetStrings.apply(i))

同理， val numNames = Array(“zero”,”one”,”two”)

等同于：val numNames=Array.apply(“zero”,”one”,”two”)

## 数组（Array）

（1）定长数组

声明数组的两种形式：

1. 声明指定长度的数组 val 数组名= new Array[类型](数组长度)

scala> val arr = new Array[String](10)

arr: Array[String] = Array(null, null, null, null, null, null, null, null, null,

null)

scala> val intArr = new Array[Int](4)

intArr: Array[Int] = Array(0, 0, 0, 0)

提供数组初始值的数组，无需new关键字:

scala> val charArr = Array('a','b','c')

charArr: Array[Char] = Array(a, b, c)

Scala声明数组时，需要带有Array类名，且使用 () 来指明长度或提供初始值序列。

（2）变长数组

scala> import scala.collection.mutable.\_

import scala.collection.mutable.\_

scala> val b = ArrayBuffer[Int]()

b: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> b += 1

res3: b.type = ArrayBuffer(1)

scala> b +=(1,2,3,5)

res4: b.type = ArrayBuffer(1, 1, 2, 3, 5)

scala> b ++= Array(8,9,10)

res5: b.type = ArrayBuffer(1, 1, 2, 3, 5, 8, 9, 10)

scala> b.trimEnd(5)

scala> b

res7: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 1, 2)

ArrayBuffer，全称scala.collection.mutable.ArrayBuffer，类似于Java中的ArrayList和C++中的vector，是长度可变数组类型的结构，称为数组缓冲。

通过：val 名 = ArrayBuffer[类型]() 来声明，声明ArrayBuffer时需要指明类型。

通过 += 操作符来添加元素或序列，++= 操作符来拼接数组。

【在ArrayBuffer的尾端添加或删除元素是一个高效的操作。】 ArrayBuffer还支持在指定位置插入、删除元素。

ArrayBuffer到数组的转换： toArray方法：

scala> b.toArray

res8: Array[Int] = Array(1, 1, 2)

（3）数组遍历

通过for循环来遍历数组：

scala> for(i <- 0 until b.length)

| println(b(i))

1

1

2

scala> for( i <- b)

| println(i)

1

1

2

指定遍历数组的步长——通过until(上界, 步长)

scala> val intArr = Array(1,2,3,4,5,6,7,8,9,10)

intArr: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> for(i <- 0 until(intArr.length,2))

| println(intArr(i))

1

3

5

7

9

通过reverse函数逆序遍历数组：

scala> for(i <- (0 until intArr.length).reverse)

| println(intArr(i))

10

9

8

7

6

5

4

3

2

1

（4）通过 for循环+yield 来获得新的数组或ArrayBuffer

scala> val intArr = Array(1,2,3,4,5,6,7,8,9,10)

intArr: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> for( a <- intArr) yield{

| a\*2

| }

res13: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

通过for+yield操作数组或ArrayBuffer之后将得到新的数组或ArrayBuffer。

（5）多维数组

Scala中的多维数组同Java中一样，多维数组都是数组的数组。通过 Array.ofDim[类型](维度1, 维度2, 维度3,....)来声明多维数组，如声明二维数组；

scala> val mulDimArr = Array.ofDim[Double](2,3)

mulDimArr: Array[Array[Double]] = Array(Array(0.0, 0.0, 0.0), Array(0.0, 0.0, 0.

0))

从二维数组的初始化中，我们可以看到，多维数组在Scala中也是数组的数组。通过 Array[ Array[Int]](维度1) 来声明数组，可以声明不规则数组；

scala> val difLenMulArr = new Array[Array[Int]](3)

difLenMulArr: Array[Array[Int]] = Array(null, null, null)

scala> for(i <- 1 to difLenMulArr.length){

| difLenMulArr(i-1) = new Array[Int](i)

| }

scala> difLenMulArr

res2: Array[Array[Int]] = Array(Array(0), Array(0, 0), Array(0, 0, 0))

多维数组是数组的数组，按照这种性质来声明多维数组，如例子中的二维数组，声明时，需要指定最外围的数组大小。

【注：可以通过scala.collection.JavaConversions包中隐式转换方法来实现Scala容器类与Java中类的转换。】

## 使用列表（List）

Scala中的List是不可变的同类对象序列，类似于java的string。

Val oneTwo = List(1,2)

Val threeFour = List(3,4)

Val oneTwoThreeFour=oneTwo ::: threeFour //合并两个List

把一个元素合并到一个list中：

Val twoThree=List(2,3)

Val oneTwoThree = 1 :: oneTwo // : cons

其中， 1 :: oneTwo是调用的：oneTwo.::(1)方法。

Scala规则：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，可以改写成a.\*(b)，除非方法名以冒号结尾。这种情况下，方法被右操作数调用。所以，1 :: twoThree里，方法的调用者是twoThree，1是方法的传入参数。

Nil，空列表的简写。

Val oneTwoThree = 1 :: 2 :: 3 :: Nil

**List不支持append操作**

Append操作的耗时会随着列表变长而成线性增长，而使用::做前缀操作仅耗用固定时间。如果构造列表，可采用：

1. 把元素前缀进去，然后调用reverse
2. 使用ListBuffer（提供append操作的可变列表），完成之后调用toList方法。

Scala中规定，凡是以:开头的都为右操作符

Val oneTwoThree = 1::2::3::Nil //Nil为空列表对象

Val oneTwoThreeFour = Nil.::(5).::(4).::(3).::(2).::(1)

List对象还有其他方法，例如head、last、length、reverse、tail等。

## 元组Tuples的使用

Tuples可以存储不同的类型，而List只能存储同一类型。

Tuples可以使用.\_和索引来访问组的元素。

Tuples的索引从1开始计算。

目前Scala中tuples支持最大的长度为22

val pair=(99,”HelloWorld”)

println(pair.\_1)

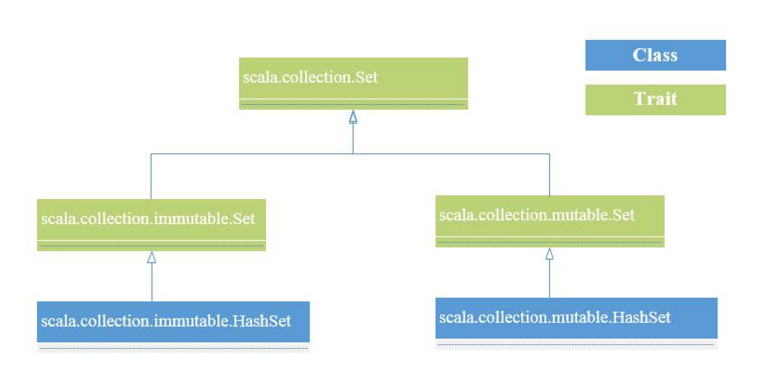
println(pair.\_2)

元组创建好之后，可以使用点号、下划线和基于1的索引访问其中的元素。

## Sets和Maps

Scala提供了两种（可变mutable和不可变immutable）类型的sets和maps类

比如Scala API 定义了Set的基Trait类型Set（Trait的概念类似于Java中的Interface，所不同的Scala中的Trait可以有方法的实现），分两个包定义Mutable（可变）和Immutable（不可变），使用同样名称的子Trait。下图为Trait和类的基础关系：



Var jetSet = Set(“Jack”,”Andy”)

jetSet += “Lily”

println(jetSet.contains(“Tom”))

缺省情况Set为Immutable Set，如果你需要使用可修改的集合类（Set类型），你可以使用全路径来指明Set，比如 scala.collection.mutalbe.Set。

Scala提供的另外一个类型为Map类型，Scala也提供了Mutable和Immutable 两种Map 类型

var romanNumeral = Map(1->"I",2->"II",3->"III",4->"IV",4->"V")  
println(romanNumeral(3))

其中，1->"I"等同于：(1).->("I")，返回一个二元组(1,” I”)

Scala的任何对象都能调用->方法，并返回包含键值对的二元组。

## 读取文件

引用包：Import

引用多个包：用-，不是\*

var fileName = "G:/data/test.txt"  
for(line <- Source.fromFile(fileName).getLines())  
 println( line)

## 类和对象（一）

Scala类定义和Java非常类似，也是以class开始，和Java不同的，Scala的缺省修饰符为public，也就是如果不带有访问范围的修饰符public,protected,private，Scala缺省定义为 public。

类的方法以def定义开始，要注意的Scala的方法的参数都是val类型，而不是var类型，因此在函数体内不可以修改参数的值

类的方法分两种，一种是有返回值的，一种是不含返回值的，没有返回值的主要是利用代码的“副作用”，比如修改类的成员变量的值或者读写文件等。Scala内部其实将这种函数的返回值定为Unit（类同Java的void类型），对于这种类型的方法，可以省略掉“=”号，因此如果你希望函数返回某个值，但忘了方法定义中的“=”，Scala会忽略方法的返回值，而返回Unit。

class ChecksumAccumulator {  
 private var sum =0  
 def add(b:Byte): Unit =  
 {  
 sum +=b  
 }  
 def checksum():Int = ~ (sum & 0xFF) +1  
}

**再强调一下**，***Scala代码无需使用“；”结尾，也不需要使用return返回值，函数的最后一行的值就作为函数的返回值。***

## 类和对象（二）

Scala不允许类保护静态元素(静态变量或静态方法）。在Scala中提供类似功能的是成为“Singleton（**单例对象**）“的对象。在Scala中定义Singleton对象的方法除了使用object，而非class关键字外和类定义非常类似。

object ChecksumAccumulator{  
 private val cache= scala.collection.mutable.Map[String,Int]()  
  
 def calculate(s:String): Int={  
 if(cache.contains(s))  
 cache(s)  
 else{  
 val acc=new ChecksumAccumulator  
 for(c<-s)  
 acc.add(c.toByte)  
 val cs = acc.checksum()  
 cache += (s ->cs)  
 cs  
 }  
 }

这个对象和上一篇创建的类ChecksumAccumulator同名，这在Scala中把这个对象成为其同名的类的**伴生对象**（Companion object)。 如果你需要定义的类的companion对象**，Scala要求你把这两个定义放在同一个文件中**。类和其companion对象可以互相访问对方的私有成员。

如果你是Java成员，可以把Singleton对象看成以前Java定义静态成员的地方。你可以使用类似Java静态方法的方式调用Singleton对象的方法。

Scala 的singleton对象不仅限于作为静态对象的容器，它在Scala中也是头等公民，但仅仅定义Singleton对象本身不会创建一个新的类型，你不可以使用new再创建一个新的Singleton对象（这也是Singleton名字的由来），此外和类定义不同的是，singleton对象不可以带参数，无法进行实例化。

**独立对象**（standalone object）：不与伴生类共享名称的单例对象叫独立对象，可作为相关功能方法的工具类，或者定义scala应用的入口点。

Scala的trait 和Java 的Interface相比，可以有方法的实现（这点有点像抽象类,但如果是抽象类，就不会允许继承多个抽象类）。Scala的Trait支持类和Singleton对象和多个Trait混合（使用来自这些Trait中的方法，而不时不违反单一继承的原则）。

Scala为Singleton对象的main定义了一个App trait类型。

object HelloWorld {

def main(args: Array[String]) {

println("Hello, world!")

}

}

以上代码等同于：

object HelloWorld extends App{

println("Hello, world!")3

}

## 基本数据类型

Scala的基本数据类型有:Byte,Short，Int，Long和Char（这些成为整数类型）。整数类型加上Float和Double成为数值类型。此外还有String类型，除String类型在java.lang包中定义，其它的类型都定义在包scala中。比如Int的全名为scala.Int。实际上Scala运行环境自动会载入包scala和java.lang中定义的数据类型，你可以使用直接使用Int，Short，String 而无需再引入包或是使用全称，String归于java.lang包。

**以上所有的基本类型都可以写成字面量（literal）。字面量就是直接写在代码里的常量值。**

Scala的基本数据类型的字面量也支持方法（这点和Java不同，Scala中所有的数值字面量也是对象），比如：

scala> 0 max 5

res4: Int = 5

scala> 4 to 6

res5: scala.collection.immutable.Range.Inclusive = Range(4, 5, 6)

scala> (-2.7).abs

res6: Double = 2.7

如果整数字面量结束于L或者l，就是Long类型，否则就是Int类型。

如果Int类型的字面量被赋值给Short或Byte类型的变量，字面量就会被当做是被赋值的类型，以便让字面量值处于有效范围内。例如：

scala> val little:Short=367

little: Short = 367

scala> val littler:Byte=38

littler: Byte = 38

**浮点数字面量**

浮点数字面量由十进制数字、可选的小数点、可选的E或e及指数部分组成。例如：

scala> val big=1.2345

big: Double = 1.2345

scala> val bigger=1.2345e1

bigger: Double = 12.345

scala> val biggerStill=123E45

biggerStill: Double = 1.23E47

如果浮点数字面量以F或f结束，则为Float类型，否则为Double类型。Double浮点数字面量也可以D或d结尾。

**字符字面量**

单引号之间的任意Unicode字符。

单引号之间提供一个前缀反斜杠的八进制或十六进制的表示字符编码号的数字。八进制数必须在’\\’和’\377’之间。

scala> val c='\101'

c: Char = A

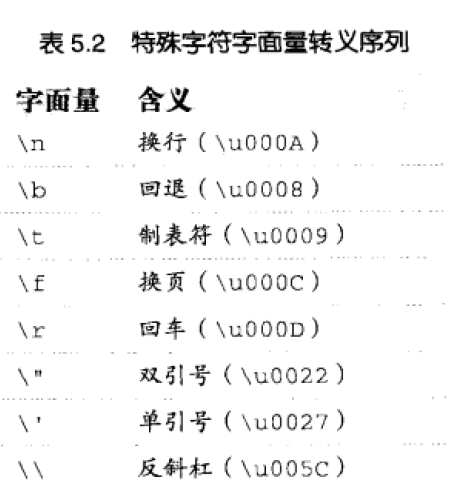
单引号之间也可以提供一个前缀\u的四位十六进制数字的通用Unicode字符方式。

scala> val d='\u0041'

d: Char = A

scala> val f='\u0044'

f: Char = D



**字符串字面量**

双引号包括的字符组成。

引号的字符语法与字符字面量相同。

特殊语法：以同一行里的三个引号（”””）作为开始和结束。内部的原始字符串可以包含任意字符（新行、引号、特殊字符）。例如：

scala> val escapes="\\\"\'"

escapes: String = \"'

scala> val escapesNew="""\"'"""

escapesNew: String = \"'

stripMargin：可以使每行进行对齐，使用时把管道符号（|）放每行的最前面。

scala> println("""|Welcome to Ultamix 3000.

| |Type "HELP" for help.""".stripMargin)

Welcome to Ultamix 3000.

Type "HELP" for help.

**符号字面量**

符号字面量写成’<标识符>，这里的<标识符>可以是任何字母或数字的标识符。这种字母量被映射为预定义类scala.Symbol的实例。

应用场景：在动态类型语言中使用一个标识符。

scala> val s='aSymbol

s: Symbol = 'aSymbol

scala> s.name

res3: String = aSymbol

符号字面量除了显示名字以外，什么都不能做。

符号是被限定，如果同一个符号字面量出现两次，那么两个字面量指向的是同一个Symbol对象。

**布尔型字面量**

包括true和false。

## 操作符和方法

Scala中，操作符是普通方法调用的另一种表现形式。例如：

scala> val sum1=(1).+(2)

sum1: Int = 3

scala> val longSum=1+2L

longSum: Long = 3

scala中，任何方法都可以当做操作符来标注。例如：

scala> val s="Hello, world!"

s: String = Hello, world!

scala> s indexOf 'o'

res4: Int = 4

scala> s indexOf ('o',5) //两个参数

res5: Int = 8

中缀操作符标注：调用的方法位于对象和传递给方法的参数和若干参数之间，例如“7 + 2”。

前缀操作符和后缀操作符都是一元（unary）的。

前缀标注：方法名被放在调用对象之前。如“- 7”

作为前缀操作符的有：+、-、!、~

scala> - 2.0

res6: Double = -2.0

scala> (2.0).unary\_-

res7: Double = -2.0

后缀标注：方法名被放在调用对象之后。如“7 toLong”。

不用点或括号调用的不带任何参数的方法。Scala里，方法调用的空括号可以省略。惯例是如果方法带有副作用就加上括号，例如println()；如果没有副作用就去掉括号，例如String的toLowerCase方法。

scala> val s="Hello, world!"

s: String = Hello, world!

scala> s.toLowerCase

res9: String = hello, world!

scala> s toLowerCase

warning: there were 1 feature warning(s); re-run with -feature for details

res10: String = hello, world!

## 对象相等性

操作== 和 != 对所有对象都起作用，不仅仅是基本类型。例如：

scala> 2 == 2

res3: Boolean = true

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

res4: Boolean = true

甚至可以比较null或任何可能是null的东西。

scala> List(1,2,3) == null

res6: Boolean = false

任何以’:’字符结尾的方法由它的右操作数调用。

## 富包装器

基本数据类型的一些方法（操作符）调用，是通过隐式转换（implicit conversion）“富包装类”，然后调用对应方法。例如：



## 函数式对象

如Rational的类定义，n和d为类参数，作为类的字段，println为相当于类的主构造器里面的代码。

class Rational(n:Int,d:Int){

require(d != 0) //先决条件  
 println("Created "+n+"/"+ d)  
}

这里的n和d只是构造器的一部分，只在类范围内有效，scala编译器不会为它们自动构造字段。

class Rational(n:Int,d:Int){  
 require(d != 0)  
 val number:Int = n  
 val denom:Int =d  
  
 override def toString = number+"/"+ denom  
 def add(that:Rational):Rational={  
 new Rational(number\*that.denom + denom\*that.number,that.denom\*denom)  
 }  
}

## 自指向

关键字this指向当前执行方法被调用的对象实例。

def lessThan(that:Rational)=

this.number\*that.denom < that.number\*this.denom

def max(that:Rational) =

if (this.lessThan(that)) that else this

## 辅助构造器

主构造器之外的构造器称为辅助构造器（auxiliary constructor）。例如：

class Rational(n:Int,d:Int){  
 require(d != 0)  
 val number:Int = n  
 val denom:Int =d  
  
 def this(n:Int) = this(n,1) //**辅助构造器**  
  
 override def toString = number+"/"+ denom  
 def add(that:Rational):Rational={  
 new Rational(number\*that.denom + denom\*that.number,that.denom\*denom)  
 }  
  
 def lessThan(that:Rational)= this.number\*that.denom < that.number\*this.denom  
 def max(that:Rational) = if (this.lessThan(that)) that else this  
}

scala的辅助构造器定义开始于def this(…)。

Scala里的每个辅助构造器都是调用同类的别的构造器。被调用的构造器可以是主构造器，也可以说辅助构造器。

Scala里的每个构造器调用终将结束于对主构造器的调用。因此，主构造器是类的唯一入口点。

## 私有字段

私有（private）：只能类的内部访问。

class Rational(n:Int,d:Int){  
 require(d != 0)  
 private val g = gcd(n.abs,d.abs) //私有字段  
  
 val number:Int = n / g  
 val denom:Int =d / g  
  
 def this(n:Int) = this(n,1) //辅助构造器  
  
 override def toString = number+"/"+ denom  
 def add(that:Rational):Rational={  
 new Rational(number\*that.denom + denom\*that.number,that.denom\*denom)  
 }  
  
 def lessThan(that:Rational)= this.number\*that.denom < that.number\*this.denom  
 def max(that:Rational) = if (this.lessThan(that)) that else this  
  
 private def gcd(a:Int,b:Int):Int={ //获取最大公约数  
 if ( b == 0) a else gcd(b,a%b)  
 }  
}

## 定义操作符

以下实现+、\*方法。

class Rational(n:Int,d:Int){  
 require(d != 0)  
 private val g = gcd(n.abs,d.abs) //私有字段  
  
 val number:Int = n / g  
 val denom:Int =d / g  
  
 def this(n:Int) = this(n,1) //辅助构造器  
  
 override def toString = number+"/"+ denom  
 def +(that:Rational):Rational={  
 new Rational(number\*that.denom + denom\*that.number,that.denom\*denom)  
 }  
   
 def \*(that:Rational):Rational={  
 new Rational(this.number\*that.number,this.denom\*that.denom)  
 }  
  
 def lessThan(that:Rational)= this.number\*that.denom < that.number\*this.denom  
 def max(that:Rational) = if (this.lessThan(that)) that else this  
  
 private def gcd(a:Int,b:Int):Int={ //获取最大公约数  
 if ( b == 0) a else gcd(b,a%b)  
 }  
}

## 方法重载

对方法+、-、\*、/实现重载（overload）。例如：

class Rational(n:Int,d:Int){  
 require(d != 0)  
 private val g = gcd(n.abs,d.abs) //私有字段  
  
 val number:Int = n / g  
 val denom:Int =d / g  
  
 def this(n:Int) = this(n,1) //辅助构造器  
  
 override def toString = number+"/"+ denom  
 def +(that:Rational):Rational={  
 new Rational(number\*that.denom + denom\*that.number,that.denom\*denom)  
 }  
  
 def +(i:Int):Rational={  
 new Rational(number+denom\*i,denom)  
 }  
  
 def -(that:Rational):Rational={  
 new Rational(number\*that.denom - that.number\*denom,denom\*that.denom)  
 }  
  
 def -(i:Int):Rational={  
 new Rational(number - denom\*i,denom)  
 }  
  
 def \*(that:Rational):Rational={  
 new Rational(this.number\*that.number,this.denom\*that.denom)  
 }  
  
 def \*(i:Int):Rational={  
 new Rational(number\*i,denom)  
 }  
  
 def /(that:Rational):Rational={  
 new Rational(number\*that.denom,denom\*that.number)  
 }  
  
 def /(i:Int):Rational={  
 new Rational(number,denom\*i)  
 }  
  
 def lessThan(that:Rational)= this.number\*that.denom < that.number\*this.denom  
 def max(that:Rational) = if (this.lessThan(that)) that else this  
  
 private def gcd(a:Int,b:Int):Int={ //获取最大公约数  
 if ( b == 0) a else gcd(b,a%b)  
 }  
}

编译器会根据传入的参数类型，选择适当类型的重载方法版本。

## 隐式转换

现在可以写r \* 2 了，或许可以交换一下操作数，就像2 \*r。但编译器会报错误：

Error:(17, 13) overloaded method value \* with alternatives:

(x: Double)Double <and>

(x: Float)Float <and>

(x: Long)Long <and>

(x: Int)Int <and>

(x: Char)Int <and>

(x: Short)Int <and>

(x: Byte)Int

cannot be applied to (Rational)

print(2 \* r)

^

这是因为2 \* r 等同于2.\*r，但整数Int类并没有带Rational参数的乘法。

为了解决这个问题，可以通过把整数转换为有理数的隐式转换实现。

implicit def intToRational(x:Int) = new Rational(x)

这段代码定义了从Int到Rational的转换方法。关键字implicit修饰符告诉编译器可以在一些情况下自动调用。

**注意：隐式转换起作用，需要定义在作用范围之内。**

## 内建控制结构

Scala内建的控制结构包括: if、 while、 for、 try 、match和函数调用。

几乎所有的控制结构都会产生某个值，这是函数式语言所采用的方式。

**（1）if表达式**

简单例子：

var fileName ="default.txt"  
if(!args.isEmpty)  
 fileName = args(0)

以上代码可以简化为：

val filename = if(!args.isEmpty) args(0) else “default.txt”

用val替换var的好处有两方面：一是节省审查变量作用域的所有代码；二是更好的支持等效推论（equational reasoning）。在表达式没有副作用的前提下，引入的变量等效于计算它的表达式。因此，无论何时都可以用表达式替代变量名。

**（2）while循环**

跟其他语言一样，包括状态判断和循环体。例如：

def gcdLoop(x:Long,y:Long):Long={  
 var a = x  
 var b = y  
 while (a != 0){  
 val temp = a  
 a = b % a  
 b = temp  
 }  
 b  
}

scala同样也有do-while循环，

var line = ""  
 do{  
 line = readLine()  
 println("Read: "+ line)  
 }while(line != "")  
}

while和do-while结构被称为“循环”，而不是表达式，是因为它们不能产生有意义的结果。

结果的类型是Unit，是表明存在并且唯一存在类型为Unit的值，称为unit value，写成（）。（）的存在是scala的Unit不同于java的void的地方。在解释器里尝试输入下列代码：

scala> def greet() { println("hi")}

greet: ()Unit

scala> greet() == ()

res0: Boolean = true

由于方法之前没有等号，greet被定义为结果类型为Unit的过程。因此，greet返回unit值，（）。

假如有下面几行代码，在scala里面按照java里while循环的方法读取文本行。

var line = ""  
 while((line = readLine()) != ""){  
 println("Read: "+readLine())  
 }

由于赋值语句“line = readLine()”的值永远返回（）而不是””，while循环的状态永远不会是假，于是循环永远无法结束。

Scala的while循环和其它语言如Java功能一样，它含有一个条件，和一个循环体，只有条件满足，就一直执行循环体的代码。

Scala 也有do-while循环，它和while循环类似，只是检查条件是否满足在循环体执行之后检查。

Scala 的while 和 do-while 称为“循环”而不是表达式，是因为它不产生有用的返回值（或是返回值为 Unit),可以写成（）。”()”的存在使得Scala的Unit和Java的void类型有所不同。

正因为While循环没有值，因此在纯函数化编程中应该避免使用while循环，Scala保留的While循环是因为在某些时候使用循环代码比较容易理解，而如果使用纯函数化编程时，需要执行一些重复运行的代码，通常就需要使用回溯函数来实现，回溯函数通常看起来不是很直观。

总的来说，推荐尽量避免在代码使用while循环，正如函数化编程要避免使用var变量一样。 而使用while循环时通常也会使用到var变量，因此在你打算使用while循环时需要特别小心，看是否可以避免使用它们。

**（3）for表达式**

**枚举集合元素**

这是使用for表示式的一个基本用法，和Java的for非常类型，比如下面的代码可以枚举当前目录下所有文件：

val filesHere = (new java.io.File(".")).listFiles()  
  
for(file <- filesHere)  
 println(file)

其中如 file < – filesHere 的语法结构，在Scala中称为“发成器 (generator)”语法。 本例中filesHere 的类型为 Array[File], 每次迭代 变量file 会初始化为该数组中一个元素， File的toString()为文件的文件名，因此println(file)打印出文件名。

Scala的for 表达式支持所有类型的集合类型，而不仅仅是数组。比如说Range。

scala> for(i <- 1 to 4)

| println("Iteration "+i)

Iteration 1

Iteration 2

Iteration 3

Iteration 4

如果不想包括被枚举的Range的上边界，可以用until代替to。例如：

scala> for(i <- 1 until 4)

| println("Iteration "+i)

Iteration 1

Iteration 2

Iteration 3

**过滤**

某些时候，你不想枚举集合中的每个元素，而是只迭代某些符合条件的元素，在Scala中，你可以为for表达式添加一个过滤器–在for的括号内添加一个if语句，例如：

for(file <- filesHere //多个filter  
 if file.isFile  
 if file.getName.endsWith(".mdb")  
) println(file)

val filesHere = (new java.io.File("./src")).listFiles  
for(file <- filesHere if file.getName.endsWith(".scala")) //函数式风格  
 println(file)

也可以写成这种形式：

val filesHere = (new java.io.File("./src")).listFiles  
for(file <- filesHere)  
 if(file.getName.endsWith(".scala")) //指令式风格  
 println(file)

如果有多个if过滤器，需要后面加分号（貌似加不加分号都可以）。

for(file <- filesHere //多个filter  
 if file.isFile;  
 if file.getName.endsWith(".mdb")  
) println(file)

**嵌套枚举**

如果加入多个<-子句，就会得到嵌套的“循环”。

val filesHere = (new java.io.File("./src")).listFiles  
  
def fileLines(file:java.io.File)=  
 scala.io.Source.fromFile(file).getLines.toList  
  
def grep(pattern:String)=  
 for(file <- filesHere  
 if file.isFile  
 if file.getName.endsWith(".txt"); //这里必须有分号  
 line <- fileLines(file)  
 if line.trim.matches(pattern)  
 )println(file+": "+line.trim)  
  
grep(".\*summer.\*") //匹配单词‘summer’

**流间（mid-stream）变量绑定**

前面代码例子中重复出现的表达式line.trim。这是个不可忽略的计算，因此你或许希望只算一遍。可以通过用等号（=）把结果绑定到新变量实现。绑定的变量被当做val引入和使用，不过不带关键字val。例如：

val filesHere = (new java.io.File("./src")).listFiles  
  
def fileLines(file:java.io.File)=  
 scala.io.Source.fromFile(file).getLines.toList  
  
def grep(pattern:String)=  
 for (file <- filesHere  
 if file.isFile  
 if file.getName.endsWith(".txt");  
 line <- fileLines(file);  
 trimmed = line.trim;  
 if trimmed.matches(pattern)  
 ) println(file + ": " + trimmed)  
  
grep(".\*summer.\*")

**制造新集合**

在进行循环时，可以创建一个值去记住每一次迭代，迭代完就可以产生新的集合。通过关键字yield实现。例如：

val filesHere = (new java.io.File("./src")).listFiles  
  
def scalaFiles =  
 for(file <- filesHere  
 if file.isFile  
 if file.getName.endsWith(".scala")  
 )yield file

关键字yield放在循环体之前。

for (子句)　yield {循环体}

另外一个例子：

val filesHere = (new java.io.File("./src")).listFiles  
  
 def fileLines(file:java.io.File)=  
 scala.io.Source.fromFile(file).getLines.toList  
  
 val forLineLengths =  
 for(file <- filesHere  
 if file.isFile  
 if file.getName.endsWith(".txt");  
 line <- fileLines(file);  
 trimmed = line.trim;  
 if trimmed.matches(".\*summer.\*")  
 )yield trimmed.length

**（4）Try表达式处理异常**

Scala的异常处理和其它语言比如Java类似，一个方法可以通过抛出异常的方法而不返回值的方式终止相关代码的运行。调用函数可以捕获这个异常作出相应的处理或者直接退出，在这种情况下，异常会传递给调用函数的调用者，依次向上传递，直到有方法处理这个异常。

**抛出异常**

Scala抛出异常的方法和Java一样，使用throw方法，例如，抛出一个新的参数异常：

throw new IllegalArgumentException

尽管看起来似乎有些自相矛盾，Scala中，throw也是一个表达式，也是有返回值的，

val n = 5  
val half =  
 if (n % 2 == 0) n/2  
 else throw new RuntimeException("n must be even")

从技术角度看，抛出异常的类型是Nothing。尽管throw不产生任何值，还是可以把它当作表达式。

**捕获异常**

类似于模式匹配。

try{  
 val f = new FileReader("G:/data/test.txt")  
}catch{  
 case ex:FileNotFoundException=>  
 case ex:IOException =>  
}

注意：和Java异常处理不同的一点是，Scala不需要你捕获checked的异常，这点和C#一样，也不需要使用throw来声明某个异常，当然如果有需要还是可以通过@throw来声明一个异常，但这不是必须的。

**Finally语句**

Scala也支持finally语句，你可以在finally块中添加一些代码，这些代码不管try块是否抛出异常，都会执行。比如，你可以在finally块中添加代码保证关闭已经打开的文件，而不管前面代码中是否出现异常。

**生成返回值**

和大部分Scala控制结构一样,Scala的try-catch-finally也生成某个值，比如下面的例子尝试分析一个URL，如果输入的URL无效，则使用缺省的URL链接地址：

通常情况下，finally块用来做些清理工作，而不应该产生结果。如果finally子句有计算值，也会被抛弃。但如果在finally语句中用return进行返回值，则之前的返回值将被覆盖。

def urlFor(path:String)={  
 try {  
 new URL(path)  
 }catch{  
 case ex: MalformedURLException =>  
 new URL("http://www.scala-lang.org")  
 }finally{

new URL("http://baidu.com") //finally计算值将被抛弃

}  
}

举个例子，例如：

def f():Int = try { return 1} finally { return 2 }

调用f()将返回2。相反：

def g():Int = try { 1 } finally {2}

调用g将返回1。

## Match表达式

Scala的Match表达式支持从多个选择中选取其一，类似其它语言中的switch语句。通常来说，Scala的matc表达式支持任意的匹配模式，这种基本模式将在后面介绍，本篇介绍类似switch用法的match表达式，也是在多个选项中选择其一。

val foodName="salt"  
foodName match {  
 case "salt" => println("pepper") //可以设置成返回值"pepper"  
 case "chips" => println("salsa") //可以设置成返回值"salsa"  
 case "eggs" => println("bacon") //可以设置成返回值"bacon"  
 case \_ => println("huh?") //可以设置成返回值"huh?"  
}

这段代码和Java的switch相比有几点不同：

一是任何类型的常量都可以用在case语句中，而不仅仅是int或是枚举类型。

二是每个case语句无需使用break，Scala不支持“fall through”。

三是Scala的缺省匹配为”\_”,其作用类似java中的default.

## 如何替代Break和continue

Scala特地没有在内置控制结构中包含break和continue是因为这两个控制结构和函数字面量有点格格不入，函数字面量我们将在后面介绍，函数字面量和其它类型字面量，比如数值字面量 4,5.6相比，他们在Scala的地位相同。

一个简单的方法是使用一个if语句来代替一个continue，使用一个布尔控制量来去除一个break。比如下面的Java代码使用continue和break在循环结构中：

def testFindChar(strValue:Array[String]):Int={  
 var i= 1  
 var foundIt = false  
 while(i< strValue.length && !foundIt){  
 if(!strValue(i).startsWith("\_")){  
 if(strValue(i).endsWith(".scala"))  
 foundIt = true  
 }  
 i =i+1  
 }  
 i  
}

这段代码和前面Java实现非常类似，并且使用了两个var变量，使用纯函数化编程的一个方法是去掉var变量的使用，递归函数（回溯函数）的使用是通常使用的一个方法来去除循环结构中使用var变量。

如果以递归方式实现，如：

def searchFrom(i:Int):Int=  
 if (i >= args.length) -1  
 else if (args(i).startsWith("-")) searchFrom(i +1)  
 else if (args(i).endsWith(".scala")) i  
 else searchFrom(i + 1)  
  
val i = searchFrom(0)

## 变量范围

Scala允许在嵌套范围内定义同名变量（与java不同的地方）。例如：

val a =1;  
{  
 val a = 2  
 println(a)  
}  
println(a)

这个脚本先打印出来的是2，然后打印出来的是1。因为定义在花括号里的a是不同的变量，所以仅在花括号内部有效。在scala程序里，内部变量被认为是遮蔽了（shadow）了同名的外部变量。

同理，在scala解释器里也可以随意重用变量名，这是因为scala解释器为每次输入的新语句创建了新的嵌套范围。

## 重构指令式风格的代码

以指令式风格的写的乘法代码如下：

def printMultiTable(): Unit ={  
 var i= 1  
 while(i <= 10){  
 var j =1  
 while (j <= 10){  
 val prod = (i\*j).toString  
 var k = prod.length  
 while (k <4){  
 print(" ")  
 k += 1  
 }  
 print(prod)  
 j +=1  
 }  
 println()  
 i += 1  
 }  
}

函数式风格代码如下：

def makeRowSeq(row:Int)=  
 for(col <- 1 to 10) yield{  
 val prod = (row \* col).toString  
 val padding = " " \* (4 - prod.length)  
 padding + prod  
 }  
  
def makeRow(row:Int) = makeRowSeq(row).mkString  
  
def multiTable() = {  
 val tableSeq =  
 for(row <- 1 to 10)  
 yield makeRow(row)  
  
 tableSeq.mkString("\n")  
}

## 函数和闭包

**（1）方法**

方法（method）：定义为对象成员的函数。例如：

import scala.io.Source  
  
object LongLines {  
 def processFile(fileName:String,width:Int): Unit ={  
 val source = Source.fromFile(fileName)  
 for(line <- source.getLines)  
 processLine(fileName,width,line)  
 }  
  
 def processLine(fileName:String,width:Int,line:String): Unit ={  
 if(line.length > width)  
 println(fileName+": "+ line.trim)  
 }  
}

**（2）本地函数**

函数式编程风格的重要设计原则：程序应该分解成若干个小的函数，每块实现一个定义完备的任务，每块都非常小。

在scala里面，可以把函数定义别的函数之内。就好像本地变量一样，这种本地函数仅在包含它的代码中可见。例如：

def processFile(fileName:String,width:Int): Unit ={  
 def processLine(line:String): Unit ={  
 if(line.length > width)  
 println(fileName + ": "+ line.trim)  
 }  
  
 val source = Source.fromFile(fileName)  
 for(line <- source.getLines)  
 processLine(line)  
}

**（3）头等函数**

Scala里的函数是头等函数（first-class function）。可以把函数写成匿名的字面量（literal），有可以把函数作为值（value）进行传递。

函数字面量被编译进类，并在运行期实例化为函数值（function value）。因此函数字面量和函数值的区别在于，函数字面量存在于源代码，而函数值作为对象存在于运行期。例如：

scala> val increase = (x:Int) => x + 1

increase: Int => Int = <function1>

scala> increase(10)

res1: Int = 11

函数可以作为其他函数的参数进行传入，例如集合类的foreach、filter方法。

scala> val someNumbers = List(-11,-10,-5,0,5,10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x:Int) => println(x))

-11

-10

-5

0

5

10

scala> someNumbers.filter((x:Int) => x >0)

res3: List[Int] = List(5, 10)

**（4）函数字面量的短格式**

**a.去除参数类型：**例如：

scala> val someNumbers = List(-11,-10,-5,0,5,10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter((x) => x>0) //去除参数类型

res4: List[Int] = List(5, 10)

b. 去除无效符号

scala> val someNumbers = List(-11,-10,-5,0,5,10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter(x => x>0) //去除无效符号

res4: List[Int] = List(5, 10)

**（5）占位符语法**

如果想让函数字面量更简洁，可以把下划线当作一个或多个参数的占位符，只要每个参数在函数字面量内仅出现一次。例如：

scala> someNumbers.filter(\_ > 0) //参数只出现一次

res5: List[Int] = List(5, 10)

因此这里的参数只出现一次，可以写成简洁的形式。

如果有多个参数，比如写成 val f = \_ + \_ ，编译器就会报错，这时就要指定参数的类型，比如：

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

在这里，第一个下划线代表第一个参数，第二个下划线代表第二个参数，依次类推。

**（6）部分应用函数**

以上例子中下划线替代的是单个参数，还可以使用单个下划线替代整个参数列表，例如println(\_)，也可以写成println \_，println和下划线之间要有空格。例如：

scala> someNumbers.foreach(println \_)

以上代码等同于：

scala> someNumbers.foreach( x => println(x))

scala里面，调用方法，把参数传入方法中，叫做函数应用到参数上。

部分应用函数（partially applied function）：是一种表达式，不需要提供函数需要的所有参数。代之以仅提供部分，或不提供所需参数。例如：

scala> def sum(a:Int,b:Int,c:Int)=a + b +c

sum: (a: Int, b: Int, c: Int)Int

scala> sum(1,2,3)

res12: Int = 6

scala> val a = sum \_

a: (Int, Int, Int) => Int = <function3>

scala> a(1,2,3)

res13: Int = 6

这里的sum \_为部分应用函数，缺省了三个整数参数。这里的a(1,2,3)等同于：a.apply(1,2,3)。

再如：

scala> val b=sum(1,\_ :Int,3) //b为部分应用函数

b: Int => Int = <function1>

scala> b(2) //b.apply调用了sum(1,2,3)

res16: Int = 6

**（7）函数字面量**

有如下代码：

scala> var more = 1

more: Int = 1

scala> val addMore =(x:Int) => x + more

addMore: Int => Int = <function1>

scala> addMore(3)

res17: Int = 4

如上面的代码，依照函数字面量addMore在运行时创建的函数值（对象）叫做闭包（closure）。

不带自由变量的函数字面量(x:Int) => x + 1，被称为封闭项（closed term），这里的项（term）指的是一小部分源代码。任何带有自由变量的函数字面量，如(x:Int) => x + more，叫做开放项（open term）。

当自由变量more发生变化时，闭包计算结果也会发生变化。

如果闭包访问了某些在程序运行时有若干不同备份的变量，例如闭包使用了某个函数的本地变量，并且函数在被调用很多次的情况下，使用的实例是那个在闭包被创建的时候活跃的那个。例如：

scala> def makeIncreaser(more:Int) = (x:Int) => x + more

makeIncreaser: (more: Int)Int => Int

scala> val inc1 = makeIncreaser(1)

inc1: Int => Int = <function1>

scala> val inc9999 = makeIncreaser(9999)

inc9999: Int => Int = <function1>

scala> inc1(10)

res18: Int = 11

scala> inc9999(10)

res20: Int = 10009

在上例中，scala编译器重新做了布局以使得捕获的参数继续存在于堆中，而不是堆栈中，因此可以保留在创建它的方法调用之外。

**（8）重复参数**

Scala中，可以指定函数的最后一个参数是重复的，这样就可以向函数传递变长度的参数列表。

使用方式：在最后一个参数后面加\*号。例如：

scala> def echo(args:String\*)=

| for(arg <- args)

| println(arg)

echo: (args: String\*)Unit

scala> echo("hello",",","world!")

hello

,

world!

在函数内部，重复参数被定义为参数类型的数组。这里定义的String\*类型的args实际上是Array[String]类型。如果把一个相同类型的数组作为重复参数传入函数，需在数组参数后面添加一个冒号和一个\_\*符号。例如：

scala> var arr = Array("one","two")

arr: Array[String] = Array(one, two)

scala> echo(arr:\_\*)

one

two

**（9）尾递归**

最后一个动作调用自己的函数，成为尾递归（tail recursive）。例如：

def approximate (guess:Double):Double=

if (isGoodEnough(guess)) guess

else approximate(improve(guess))

如果以循环结构实现类似的功能：

def approximateLoop(initialGuess:Double):Double={

var guess = initialGuess

while(!isGoodEnough(guess))

guess = improve(guess)

guess

}

从代码简洁性上看，利用尾递归的函数式风格更好些；从执行效率上看，两种代码执行的时间几乎相同，这主要是scala编译器对尾递归的方式进行了优化。

**尾递归函数追踪**

尾递归函数将不会为每个调用制造新的堆栈结构；所有的调用将在一个结构内执行。

def boom(x:Int):Int = {  
 if (x ==0 ) throw new Exception("boom!")  
 else boom(x -1 ) +1  
}

这段代码并不是尾递归，因为在递归调用之后执行了递增操作。

可以修改为尾递归方式：

def bang(x:Int):Int={  
 if (x == 0) throw new Exception("bang!")  
 else bang(x -1)  
}

**尾调用优化**

Approximate编译后的代码和approximateLoop的编译后代码相同，说明scala编译器优化了尾递归调用。

**尾递归的局限**

Scala里尾递归的使用局限很大，因为jvm指令集使实现更先进的尾递归形式变得很困难。Scala仅优化了直接递归调用使其返回同一个函数。如果递归是间接的，如下面两个相互递归的函数，就没有优化的可能性了。

def isEven(x:Int):Boolean=  
 if (x == 0) true else isOdd(x -1)  
  
def isOdd(x:Int):Boolean=  
 if (x == 0) true else isEven(x -1)

同样，如果最有一个调用时一个函数值你也不能获得尾调用优化。比如：

val funValue = nestedFun \_  
def nestedFun(x:Int){  
 if (x !=0) { println(x);funValue(x-1)}  
}

尾调用优化限定了方法或嵌套函数必须在最后一个操作调用本身，而不是转到某个函数值或者什么其他的中间函数的情况。而这里的funValue变量指向一个实质是包装了nestedFun的调用的函数值。

## 控制对象

**（1）减少代码重复**

高阶函数（higer-order function）：将其他函数作为参数的函数。

高阶函数的好处：可以创建控制抽象以减少代码重复。例如，编写一个搜索指定文件后缀的文件列表：

object FileMatcher {  
  
 private def filesHere = (new File("./src")).listFiles  
  
 def filesEnding(query:String)={  
 for( file <- filesHere; if file.getName.endsWith(".scala"))  
 yield file  
 }  
}

def filesContaining(query:String)={  
 for( file <- filesHere; if file.getName.contains(query))  
 yield file  
}  
  
def filesRegex(query:String)={  
 for( file <- filesHere; file.getName.matches(query))  
 yield file  
}

}

scala风格的简化方式：

private def filesHere = (new File("./src")).listFiles

def filesMatching(matcher:String => Boolean) ={  
 for(file <- filesHere; if matcher(file.getName))  
 yield file  
}  
  
def filesEnding(query:String)={  
 filesMatching(\_.endsWith(query))

}  
  
def filesContains(query:String)={  
 filesMatching(\_.contains(query))  
}  
  
def filesRegex(query:String)={  
 filesMatching(\_.matches(query))  
}

**（2）简化客户代码**

高阶函数的另一个重要应用是把它们放在API里使客户代码更加简洁。

举个例子，比如从一个集合中查找符合条件的元素，传统的写法是这样的：

def containsNeg(nums:List[Int]):Boolean={  
 var exists = false  
 for(num <- nums)  
 if (num < 0)  
 exists = true  
 exists  
}

对于上述代码实现的功能，可以有更简洁的定义：

def containsNeg(nums:List[Int]) = nums.exists(\_ < 0)

这里调用了List里内置的高阶函数exists。

同样，如果写一个containsOdd方法，传统写法可能是这样：

def containsOdd(nums:List[Int]): Boolean={

var exists = false

for(num <- nums)

if(num %2 == 1)

exists = true

exists

}

如果调用集合的exists方法，可以写成这样：

def containsOdd(nums:List[Int]) = nums.exists(\_ % 2 == 1)

（3）柯里化（currying）

柯里化的函数被应用于多个参数列表，而不是仅仅一个。

下面是一个规整的，未被柯里化的函数：

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

plainOldSum: (x: Int, y: Int)Int

scala> plainOldSum(1,2)

res0: Int = 3

然后，对这个函数进行柯里化之后，代码如下：

scala> def curriedSum (x:Int)(y:Int) = x + y

plainOldSum: (x: Int)(y: Int)Int

scala> plainOldSum(1)(2)

res1: Int = 3

当调用curriedSum时，实际上是连续调用了两个传统函数。第一个函数调用带单个的名为x的Int型参数，并返回第二个函数的函数值。第二个函数带Int型参数y。实际上执行了以下过程：

scala> def first(x:Int)=(y:Int)=>x+y

first: (x: Int)Int => Int

scala> def second = first(1)

second: Int => Int

scala> second(2)

res2: Int = 3

first和second函数只是柯里化过程的一个演示。它们并不直接连在curriedSum上。尽管如此，仍然有一个方式获得实际指向curriedSum的“第二个”函数的参考。可以用部分应用函数表达式方式，把占位符标注应用在curriedSum里。例如：

scala> val onePlus= curriedSum(1)\_

onePlus: Int => Int = <function1>

curriedSum(1)\_里的下划线是第二个参数列表的占位符。结果就是指向一个函数的参考，这个函数在被调用的时候，对它唯一的Int参数加1并返回结果：

scala> onePlus(2)

res4: Int = 3

同样，可以获取对唯一的Int参数加2函数的方式：

scala> val twoPlus = curriedSum(2)\_

twoPlus: Int => Int = <function1>

scala> twoPlus(2)

res5: Int = 4

**（4）编写新的控制结构**

创建控制结构的途径：创建带函数作为参数的方法。

例如，编写一个“双倍”控制结构：

scala> def twice(op:Double=>Double,x:Double)=op(op(x))

twice: (op: Double => Double, x: Double)Double

scala> twice(\_ + 1,5)

res0: Double = 7.0

任何时候，当你的代码中多个地方有重复的控制模式时，就应该考虑把它实现为 一个新的控制结构。

再比如，打一个资源，对它进行操作，然后关闭资源。

def withPrintWriter(file:File,op:PrintWriter => Unit): Unit ={  
 val writer = new PrintWriter(file) //借贷模式  
 try{  
 op(writer)  
 }finally {  
 writer.close  
 }  
}  
  
withPrintWriter(new File("G:/temp/date.txt"),writer => writer.println(new Date()))

这里的writer借给op操作，并保证资源最后关闭，成为（资源）借贷模式（loan pattern）。

**当向函数传入一个参数时，可以用花括号代替小括号的机制，其目的是让程序员写出包围在花括号内的函数字面量。当函数包含两个或两个以上参数，只能使用小括号。**

下面已 柯里化的方式对上述函数进行修改：

def withPrintWriter(file:File)(op:PrintWriter => Unit): Unit ={  
 val writer = new PrintWriter(file)  
 try{  
 op(writer)  
 }finally {  
 writer.close  
 }  
}  
  
val dateFile = new File("G:/temp/date.txt")  
withPrintWriter(dateFile){ writer => writer.println(new Date)} //调用方式

在这里，调用柯里化之后的withPrintWriter时，第二个参数可以使用花括号。

**（5）传名参数（by-name parameter）**

val assertionsEnabled = true  
def myAssert(predicate:() => Boolean) =  
 if(assertionsEnabled && !predicate())  
 throw new AssertionError

如果要调用上面定义的函数myAssert，可以这样写：

myAssert(() => 5>3)

如果想要省略函数字面量里面的空参数列表和=>号，就可以使用传名参数。例如：

def byNameAssert(predicate: => Boolean) =  
 if(assertionsEnabled && !predicate)  
 throw new AssertionError()  
  
byNameAssert(5>3)

如果要使用传名参数，要定义参数的类型开始于=>而不是()=>

如果使用传值参数，例如：

def boolAssert(predicate:Boolean)=  
 if(assertionsEnabled && !predicate)  
 throw new AssertionError  
  
boolAssert(5>3)

这里的boolAssert(5>3)，在调用时，先计算表达式5>3的值，然后将表达式的值传入到函数boolAssert中，而byNameAssert传入的是一个函数字面量，在!predicate时调用函数字面量的apply方法。

P142

P140

## 操作基本数据类型

任何Scala方法都可以作为操作符来使用。是否是操作符取决于你如何使用这个方法，当你使用 s.indexOf(‘o’) indexOf不是一个运算符。 而你写成 s indexOf ‘o’ ,indexOf 就是一个操作符，因为你使用了操作符的语法。

除了类似+的中缀运算符（操作符在两个操作符之间），还可以有前缀运算符和后缀运算符。顾名思义前缀运算符的操作符在操作数前面，比如 -7的“-”。后缀运算符的运算符在操作数的后面，比如 7 toLong中的toLong. 前缀和后缀操作符都使用一个操作数，而中缀运算符使用前后两个操作数。Scala在实现前缀和后缀操作符的方法，其方法名都以unary\_-开头。比如 表达式 -2.0 实际上调用 （2.0）.unary\_-方法。

如果你需要定义前缀方法，你只能使用+,-,! 和～四个符号作为前缀操作符。

后缀操作符在不使用.和括号调用时不带任何参数。在Scala中你可以省略掉没有参数的方法调用的空括号。按照惯例，如果你调用方法是为了利用方法的“副作用”，此时写上空括号，如果方法没有任何副作用（没有修改其它程序状态），你可以省略掉括号。

具体Scala的基本数据类型支持的操作符，可以参考Scala API文档。下面以示例介绍一些常用的操作符：

**算术运算符：** + – × /

**关系和逻辑运算符:** 包括 >,< ,>=,!、&&、|| （短路）等

scala> def salt()= {println("salt");false}

salt: ()Boolean

scala> def pepper() = { println("pepper");true}

pepper: ()Boolean

scala> salt() && pepper()

salt

res11: Boolean = false

**位操作符：**& | …… ~

**对象恒等比较:** 如果需要比较两个对象是否相等，可以使用==和!=操作符

Scala的==和Java不同，scala 的==只用于比较两个对象的值是否相同。而对于引用类型的比较使用另外的操作符 eq 和 ne。

**操作符的优先级和左右结合性**

Scala 的操作符的优先级和Java基本相同，如果有困惑时，可以使用（）改变操作符的优先级。 操作符一般为左结合，Scala规定了操作符的结合性由操作符的最后一个字符定义。对于以“：”结尾的操作符都是右结合，其它的操作符多是左结合。

例如： a\*b 为　a.\*(b)　而　a:::b 为 b.:::(a),而 a:::b:::c = a::: (b ::: c) , a\*b\*c= (a\*b)\*c

## 类和对象（三）

Functional objects(函数化对象或是方程化对象），**函数化对象**指的是所定义的类或对象不包含任何可以修改的状态。

Java不同的，Scala的类定义可以有参数，称为类参数。Scala使用类参数，并把类定义和主构造函数合并在一起，在定义类的同时也定义了类的主构造函数。因此Scala的类定义相对要简洁些。

Scala编译器会编译Scala类定义包含的任何不属于类成员和类方法的其它代码，这些代码将作为类的主构造函数。

**重新定义类的toString 方法**

使用override来重载基类定义的方法，而且必须使用override关键字表示重新定义基类中的成员。

class Rational (n:Int,d:Int) {  
 override def toString = n+"/"+d  
}

## 前提条件检查

对于类定义时的前提条件，可以使用require方法（require方法为Predef对象的定义的一个方法，Scala环境自动载入这个类的定义，因此无需使用import引入这个对象）解决。

class Rational (n:Int,d:Int) {

require(d!=0)

override def toString = n+"/"+d

}

## 类和对象（四）

尽管类参数在新定义的函数的访问范围之内，但仅限于定义类的方法本身(比如之前定义的toString方法，可以直接访问类参数）可以访问

如果其他需要访问类参数，则需要定类的成员变量。

class Rational (n:Int,d:Int) {  
 require(d!=0)  
 val number = n  
 val denom = d  
 override def toString = n+"/"+d  
  
 def add(that:Rational): Rational={  
 new Rational(number\*that.denom+ denom\*that.number,denom\*that.denom)  
 }  
}

## 自身引用

Scala 也使用this来引用当前对象本身，一般来说访问类成员时无需使用this，但如果需要引用对象自身，this就无法省略。

class Rational (n:Int,d:Int) {  
 require(d!=0)  
 val number = n  
 val denom = d  
 override def toString = n+"/"+d  
  
 def add(that:Rational): Rational={  
 new Rational(number\*that.denom+ denom\*that.number,denom\*that.denom)  
 }  
  
 def lessThan(that:Rational):Boolean ={  
 this.number\*that.denom < this.denom\*that.number  
 }  
  
 def lessThan2(that:Rational):Boolean={  
 number\*that.denom < denom\*that.number  
 }  
  
 def max(that:Rational):Rational={  
 if (lessThan(that)) that else this  
 }  
}

**辅助构造函数**

在定义类时，很多时候需要定义多个构造函数，在Scala中，除主构造函数之外的构造函数都称为辅助构造函数（或是从构造函数），比如对于Rational类来说，如果定义一个整数，就没有必要指明分母，此时只要整数本身就可以定义这个有理数。我们可以为Rational定义一个辅助构造函数，Scala定义辅助构造函数使用 this(…)的语法，所有辅助构造函数名称为this.

def this(n:Int) = this(n,1)

所有Scala的辅助构造函数的第一个语句都为调用其它构造函数，也就是this(…)，被调用的构造函数可以是主构造函数或是其它构造函数（最终会调用主构造函数），这样使得每个构造函数最终都会调用主构造函数，从而使得主构造函数称为创建类单一入口点。在Scala中也只有主构造函数才能调用基类的构造函数，这种限制有它的优点，使得Scala构造函数更加简洁和提高一致性。

**私有成员变量和方法**

Scala 类定义私有成员的方法也是使用private修饰符

## 类和对象（五）

**定义运算符**

如果使用add定义两个Rational对象的加法，两个Rational加法可以写成x.add(y)或者x add y

即使使用x add y 还是没有 x + y 来得简洁。

在Scala中运算符（操作符）和普通的方法没有什么区别，任何方法都可以写成操作符的语法 比如 上面的 x add y.

而在Scala中对方法的名称也没有什么特别的限制，你可以使用符号作为类方法的名称，比如使用+，- \* 等符号。

def +(that:Rational):Rational={  
 new Rational(number\*that.denom+ denom\*that.number,denom\*that.denom)  
}

def \*(that:Rational):Rational={  
 new Rational(number\*that.number,denom\*that.denom)  
}

## Scala中的标识符

Scala可以使用两种形式的标志符，字符数字和符号。

字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号”$”在Scala中也看作为字母。然而以“$”开头的标识符为保留的Scala编译器产生的标志符使用，应用程序应该避免使用“$”开始的标识符，以免造成冲突。

Scala的命名规则采用和Java类似的camel命名规则，首字符小写，比如toString。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。

符号标志符包含一个或多个符号，如+,:,? 等，比如：+ ++ ::: < ?> :->

S cala 内部实现时会使用转义的标志符，比如:-> 使用$colon$minus$greater 来表示这个符号。因此如果你需要在Java代码中访问:->方法，你需要使用Scala的内部名称$colon$minus$greater。

混合标志符（mixed identifier）由字母数字组成，后面跟着下划线和一个操作标识符。比如 unary\_+ 为Scala对+方法的内部实现时的名称。或myvar\_=被用来定义赋值操作符的方法名。myvar\_=是由scala编译器产生的用来支持属性（property）的。

字面量标识符（literal identifier）为使用“定义的字符串，比如 `x` `yield`. 你可以在“之间使用任何有效的Scala标志符，Scala将它们解释为一个Scala标志符，一个典型的使用为 Thread的yield方法，在Scala中你不能使用 Thread.yield()是因为yield为Scala中的关键字，你必须使用 Thread.`yield`()来使用这个方法。

Scala里，constant这个词并不等同于val。尽管val在被初始化之后的确保持不变，但它仍然是变量。

操作符标识符（operator identifier）由一个或多个操作符字符组成。操作符字符是一些如+，:，？，~或#的可打印的ASCII字符。Scala编译器将在内部“粉碎”操作符标识符以转换成合法的内嵌’$’的java标识符。例如，标识符：->将被内部表达为$colon$minus$greater。若想从java代码访问这个标识符，就应该用这种内部表达方式。

## **Yield关键字**

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。

## **方法重载**

和Java一样，Scala也支持方法重载，重载的方法参数类型不同而使用同样的方法名称，比如对于Rational对象，+的对象可以为另外一个Rational对象，也可以为一个Int对象，此时你可以重载+方法以支持和Int相加。和Java一样，Scala也支持方法重载，重载的方法参数类型不同而使用同样的方法名称，比如对于Rational对象，+的对象可以为另外一个Rational对象，也可以为一个Int对象，此时你可以重载+方法以支持和Int相加。

def +(that:Rational):Rational={  
 new Rational(number\*that.denom+ denom\*that.number,denom\*that.denom)  
}  
  
def +(i:Int):Rational={  
 new Rational(number+i\*denom,denom)  
}

## 隐式类型转换

上面我们定义Rational的加法，并重载+以支持整数，r + 2 ,当如果我们需要 2 + r 就会报错，这是因为整数类型不支持和Rational相加。我们不可能去修改Int的定义（除非你重写Scala的Int定义）以支持Int和Rational相加。如果你写过.Net代码，这可以通过静态扩展方法来实现，Scala提供了类似的机制来解决这种问题。

如果Int类型能够根据需要自动转换为Rational类型，那么2 + r 就可以相加。

**Scala通过implicit def 定义一个隐含类型转换**，比如定义由整数到Rational类型的转换如下：

var rational = new Rational(6)  
implicit def intToRational(x:Int) = new Rational(x)  
var newRational = 2+ rational

**IF表达式**

和其它语言（比如Java，C#)相比，Scala只内置了为数不多的几种程序控制语句：if ,while, for ,try match 以及函数调用，这是因为从Scala诞生开始就包含了函数字面量，Scala内核没有定义过多的控制结构，而是可以通过额外的库来扩展程序的控制结构。

**Scala的所有控制结构都有返回结果**，如果你使用过Java或C#，就可能了解Java提供的三元运算符 ?: ，它的基本功能和if一样，当可以返回结果。Scala在此基础上所有控制结构（while,try,if,等）都可以返回结果。这样做的一个好处是，可以简化代码，如果没有这种特点，程序员常常需要创建一个临时变量用来保存结果。

总的来说，Scala提供的基本程序控制结构，“麻雀虽小，五脏俱全”，虽然少，但足够满足其他指令式语言（如Java,C++）所支持的程序控制功能，而且由于这些指令都有返回结果，可以使得代码更为精简。

Scala语言的if 的基本功能和其它语言没有什么不同，它根据条件执行两个不同的分支，比如，使用Java风格编写下面Scala的if语句的一个例子：

var filename="default.txt"

if(!args.isEmpty)

filename =args(0)

上面代码和使用Java实现没有太多区别，看起来不怎么像Scala风格，我们重新改写一下，利用if可以返回结果这个特点。

val filename=

if(!args.isEmpty) args(0)

else "default.txt"

首先这种代码比前段代码短，更重要的是这段代码使用val而无需使用var类型的变量。使用val为函数式编程风格。

## 嵌套迭代

for表达式支持多重迭代，下面的例子使用两重迭代，外面的循环枚举filesHere,而内部循环枚举该文件的每一行文字。实现了类似Unix　grep命令：

val filesHere = (new java.io.File("G:/data")).listFiles()

def fileLines(file:java.io.File)=  
 scala.io.Source.fromFile(file).getLines().toList  
  
def grep(pattern:String)=  
 for(  
 file <- filesHere  
 if file.getName.endsWith(".txt");  
 line <- fileLines(file)  
 if line.trim.matches(pattern)  
 )println(file+":"+line.trim)

grep(".\*四月天.\*")

注意上面代码中两个迭代之间使用了”;”，如果你使用{} 替代for的()的括号，你可以不使用“；”分隔这两个“生成器”，这是因为Scala编译器不推算包含在括号内的省掉的“;”。使用{}改写的代码如下：

val filesHere = (new java.io.File("G:/data")).listFiles()

def fileLines(file:java.io.File)=  
 scala.io.Source.fromFile(file).getLines().toList  
  
def grep(pattern:String)=  
 for{  
 file <- filesHere  
 if file.getName.endsWith(".txt")  
 line <- fileLines(file)  
 if line.trim.matches(pattern)  
 }println(file+":"+line.trim)

grep(".\*四月天.\*")

## 绑定中间变量

前面代码使用了多次line.trim，如果trim是个耗时的操作，你可以希望trim只计算一次，Scala允许你使用=号来绑定计算结果到一个新变量。绑定的作用和val类似，只是不需要使用val关键字，例如，修改前面的例子，只计算一次trim，把结果保存在trimmed变量中。

def grep(pattern:String)=  
 for {  
 file <- filesHere  
 if file.getName.endsWith(".txt")  
 line <- fileLines(file)  
 trimmed = line.trim  
 if trimmed.matches(pattern)  
 }println(file+":"+trimmed)

## 生成新集合

For 表达式也可以用来生产新的集合，这是Scala的for表达式比Java的for语句功能强大的地方。它的基本语法如下：

for clauses yield body

关键字yield放在body的前面，for 没迭代一次，产生一个body,yield收集所有的body结果，返回一个body类型的集合。比如，前面列出所有.scala文件，返回这些文件的集合：

def scalaFiles=  
 for{  
 file <- filesHere  
 if file.getName.endsWith(".mdb")  
 }yield file

scalaFiles的类型为Array[File]。

## 函数-类成员函数

和Java相比，Scala提供了多种Java不支持的方法来定义函数，除了类成员函数外，Scala还支持嵌套函数，函数字面量，函数变量等。

**函数-局部函数**

也就是说可以在函数的内部再定义函数，如同定义一个局部变量，例如：

def processFile(fileName:String,width:Int): Unit ={  
 def processLine(fileName:String,width:Int,line:String): Unit ={  
 if(line.length>width)  
 println(fileName+":"+line.trim)  
 }  
  
 val source = scala.io.Source.fromFile(fileName)  
 for(line <- source.getLines()){  
 processLine(fileName,width,line)  
 }  
}

这个例子中私有成员函数processLine移动到processFile内部，成为一个局部函数，也正因为processLine变成了processFile的一个局部函数，因此processLine 可以直接访问到processFile的参数 filename和width,因此代码可以进一步优化如下：

**函数-头等公民**

Scala中函数为头等公民，不仅可以定义一个函数然后调用它，而且可以写一个未命名的函数字面量，然后把它当成一个值传递到其他函数或是赋值给其他变量。

（x:int）=> x+1

这是个函数字面量，它的功能为加1。***符号=>表示这个函数将符号左边的东西，转换成符号右边的东西。***

函数字面量为一个对象，因此可以把函数字面量保存到一个变量中。这个变量也是一函数，可以使用函数风格来调用它。例如：

scala> var increase=(x:Int)=>x+1

increase: Int => Int = <function1>

scala> increase(10)

res0: Int = 11

函数字面量(x:Int)=>x+1在scala中表示为带有一个参数的Function1的一个对象，其他如FunctionN代表带有N个参数的函数，Function0代表不含参数的函数类型。

如何函数定义需要多条语句，可以使用花括号{}，例如：

scala> var increase=(x:Int)=>{

| println("we")

| println("are")

| println("here")

| x+1

| }

increase: Int => Int = <function1>

scala> increase(10)

we

are

here

res1: Int = 11

scala中的很多库允许使用函数作为参数，比如foreach：

scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x:Int) => println(x))

-11

-10

-5

0

5

10

也可以在filter函数中使用函数参数。

scala> someNumbers.filter( x => x >0)

res1: List[Int] = List(5, 10)

## 函数-函数字面量的一些简化写法

Scala提供了多种方法简化函数字面量中多余的部分，比如可以省略参数类型，scala可以根据上下文推算出参数的类型。

(x:Int)=>x+1

可以简化成：

(x => x+1)

还可以进一步去掉扩号：

X=>x+1

Scala还可以进一步简化，可以**使用“占位符”下划线“\_”来代替一个或多个参数**，只要这个参数值在代码中出现一次，Scala编译器就可以推断出参数

scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter(\_ >0)

res0: List[Int] = List(5, 10)

可以看到简化后的函数定义为 \_ > 0 ,你可以这样来理解，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。

有时，如果你使用\_来定义函数，可能没有提供足够的信息给Scala编译器，此时Scala编译器将会报错，比如，定义一个加法函数如下：

scala> val f = \_ + \_

<console>:7: error: missing parameter type for expanded function ((x$1, x$2) => x$1.$plus(x$2))

val f = \_ + \_

^

<console>:7: error: missing parameter type for expanded function ((x$1: <error>, x$2) => x$1.$plus(x$2))

val f = \_ + \_

Scala 编译器无法推断出\_的参数类型，就报错了，但如果你给出参数的类型，依然可以使用\_来定义函数，比如：

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

f: (Int, Int) => Int = <function2>

scala> f (5,10)

res1: Int = 15

因为\_替代的参数在函数体中只能出现一次，因此多个“\_”代表多个参数。第一个“\_”代表第一个参数，第二个“\_”代表第二个参数，以此类推。

## 函数-部分应用的函数

前面例子中使用“\_” 来代替单个的参数，实际上你也可以使用“\_”来代替整个参数列表，比如说，你可以使用 print \_ 来代替 println (\_).

someNumbers.foreach(println \_)

Scala编译器自动将上面代码解释成：

someNumbers.foreach( x => println (x))

因此这里的“\_” 代表了Println的整个参数列表，而不仅仅替代单个参数。

当你采用这种方法使用“\_”，你就创建了一个部分应用的函数(partially applied function)。 在Scala中，当你调用函数，传入所需参数，你就把函数“应用”到参数。 比如：一个加法函数。

scala> def sum = (\_:Int) + (\_ :Int) + (\_ :Int)

sum: (Int, Int, Int) => Int

scala> sum (1,2,3)

res0: Int = 6

**一个部分应用的函数指的是你在调用函数时，不指定函数所需的所有参数，这样你就创建了一个新的函数，这个新的函数就称为原始函数的部分应用函数**，比如说我们固定sum的第一和第三个参数，定义如下的部分应用函数：

scala> val b = sum ( 1 , \_ :Int, 3)

b: Int => Int = <function1>

scala> b(2)

res1: Int = 6

变量b的类型为一函数，具体类型为Function1 （带一个参数的函数），它是由sum 应用了第一个和第三个参数，构成的。

调用b(2），实际上调用sum (1 ,2,3)。

**在Scala中，如果你定义一个部分应用函数并且能省去所有参数，比如println \_ ，你也可以省掉“\_”本身**，比如：

someNumbers.foreach(println \_)

可以写成：

someNumbers.foreach(println)

## 函数-闭包

假如我们定义如下的函数：1

(x:Int) => x + more

这里我们引入一个自由变量more.它不是所定义函数的参数，而这个变量定义在函数外面，比如：1

var more =1

那么我们有如下的结果：

scala> var more =1

more: Int = 1

scala> val addMore = (x:Int) => x + more

addMore: Int => Int = <function1>

scala> addMore (100)

8

res1: Int = 101

***这样定义的函数变量addMore 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。有意思的是，当这个自由变量发生变化时，Scala的闭包能够捕获到这个变化，因此Scala的闭包捕获的是变量本身而不是当时变量的值。

同样的，如果变量在闭包在发生变化，也会反映到函数外面定义的闭包的值。比如：

scala> val someNumbers = List ( -11, -10, -5, 0, 5, 10)

someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum =0

sum: Int = 0

scala> someNumbers.foreach ( sum += \_)

scala> sum

res4: Int = -11

可以看到在闭包中修改sum的值，其结果还是传递到闭包的外面。

如果一个闭包所访问的变量有几个不同的版本，比如一个闭包使用了一个函数的局部变量（参数），然后这个函数调用很多次，那么所定义的闭包应该使用所引用的局部变量的哪个版本呢？ **简单的说，该闭包定义所引用的变量为定义该闭包时变量的值，也就是定义闭包时相当于保存了当时程序状态的一个快照。**比如我们定义下面一个函数闭包：

scala> def makeIncreaser(more:Int) = (x:Int) => x + more

makeIncreaser: (more: Int)Int => Int

scala> val inc1=makeIncreaser(1)

inc1: Int => Int = <function1>

scala> val inc9999=makeIncreaser(9999)

inc9999: Int => Int = <function1>

scala> inc1(10)

res5: Int = 11

scala> inc9999(10)

res6: Int = 10009

当你调用makeIncreaser(1)时，你创建了一个闭包，该闭包定义时more的值为1, 而调用makeIncreaser(9999)所创建的闭包的more的值为9999。此后你也无法修改已经返回的闭包的more的值。因此inc1始终为加一，而inc9999始终为加9999.

## 函数-可变参数、命名参数、缺省参数

**（1）重复参数**

Scala在定义函数时**允许指定最后一个参数可以重复（变长参数）**，从而允许函数调用者使用变长参数列表来调用该函数，Scala中使用“\*”来指明该参数为重复参数。例如：

scala> def echo (args: String \*) =

| for (arg <- args) println(arg)

echo: (args: String\*)Unit

scala> echo()

scala> echo ("One")

One

scala> echo ("Hello","World")

Hello

World

在函数内部，变长参数的类型，实际为一数组，比如上例的String \* 类型实际为 Array[String]。 然而，如今你试图直接传入一个数组类型的参数给这个参数，编译器会报错：

scala> val arr= Array("What's","up","doc?")

arr: Array[String] = Array(What's, up, doc?)

scala> echo (arr)

<console>:10: error: type mismatch;

found : Array[String]

required: String

echo (arr)

为了避免这种情况，你可以通过在变量后面添加 \_\*来解决，这个符号告诉Scala编译器在传递参数时逐个传入数组的每个元素，而不是数组整体。

scala> echo (arr: \_\*)

What's

up

doc?

**（2）命名参数**

通常情况下，调用函数时，参数传入和函数定义时参数列表一一对应。

scala> def speed(distance: Float, time:Float) :Float = distance/time

speed: (distance: Float, time: Float)Float

scala> speed(100,10)

res0: Float = 10.0

使用命名参数允许你使用任意顺序传入参数，比如下面的调用：

scala> speed( time=10,distance=100)

res1: Float = 10.0

scala> speed(distance=100,time=10)

res2: Float = 10.0

**（3）缺省参数值**

Scala在定义函数时，允许指定参数的缺省值，从而允许在调用函数时不指明该参数，此时该参数使用缺省值。缺省参数通常配合命名参数使用，例如：

scala> def printTime(out:java.io.PrintStream = Console.out, divisor:Int =1 ) =

| out.println("time = " + System.currentTimeMillis()/divisor)

printTime: (out: java.io.PrintStream, divisor: Int)Unit

scala> printTime()

time = 1383220409463

scala> printTime(divisor=1000)

time = 1383220422

## 函数-尾递归

下面为一个使用逼近方法求解的一个递归函数表达：

def approximate(guess: Double) : Double =

if (isGoodEnough(guess)) guess

else approximate(improve(guess))

通过实现合适的isGoodEnough和improve函数，说明这段代码在搜索问题中经常使用。 如果你打算approximate运行的快些，你很可能使用下面循环来实现什么的算法：

def approximateLoop(initialGuess: Double) : Double = {

var guess = initialGuess

while(!isGoodEnough(guess))

guess=improve(guess)

guess

}

那么这两种实现哪一种更可取呢？ 从简洁度和避免使用var变量上看，使用函数化编程递归比较好。但是有while循环是否运行效率更高些？实际上，如果我们通过测试，两种方法所需时间几乎相同，这听起来有些不可思议，因为回调函数看起来比使用循环要耗时得多。

其实，对于approximate的递归实现，Scala编译器会做些优化，**我们可以看到approximate的实现，最后一行还是调用approximate本身，我们把这种递归叫做尾递归**。

**Scala编译器可以检测到尾递归而使用循环来代替。**

**（1）跟踪尾递归函数**

一个尾递归函数在每次调用不会构造一个新的调用栈(stack frame)。所有递归都在同一个执行栈中运行，如果你在调试时，如果在尾递归中调试错误，不会看到嵌套的调用栈，比如下面的代码，非尾递归的一个实现：

scala> def boom(x:Int):Int=

| if(x==0) throw new Exception("boom!") else boom(x-1) + 1

boom: (x: Int)Int

scala> boom(5)

java.lang.Exception: boom!

at .boom(<console>:8)

at .boom(<console>:8)

at .boom(<console>:8)

at .boom(<console>:8)

at .boom(<console>:8)

at .boom(<console>:8)

... 32 elided

boom不是一个尾递归，因为最后一行为一个递归加一操作，可以看到调用boom(5)的调用栈，为多层。我们修改这段代码，使它构成一个尾递归：

scala> def bang(x:Int):Int=

| if(x==0) throw new Exception("boom!") else bang(x-1)

bang: (x: Int)Int

scala> bang(5)

java.lang.Exception: boom!

at .bang(<console>:8)

... 32 elided

这一次，只看到一层调用栈，Scala编译器将尾递归优化从循环实现，如果你不想使用这个特性，可以添加scalac编译参数 -g:notailcalls 来取消这个优化，此后，如果抛出异常，尾递归也会显示多层调用栈。

**（2）尾递归的一些局限性**

目前来说，Scala编译器只能对那些直接实现尾递归的函数，比如前面的approximate和bang,如果一个函数间接实现尾递归，比如下面代码：

def isEven(x:Int): Boolean =

if(x==0) true else isOdd(x-1)

def isOdd(x:Int): Boolean=

if(x==0) false else isEven(x-1)

isEven和isOdd事件也是为递归，但不是直接定义的尾递归，scala编译器无法对这种递归进行优化

## 减低代码重复

Scala赋予了程序员自己扩展控制结构的能力。**Scala支持函数值（值的类型为函数，而非函数的返回值）**。为避免混淆，**使用函数类型值来指代类型为函数的值。**

所有的函数可以分成两个部分：一是共有部分，这部分在该函数的调用都是相同的，另外一部分为非公共部分，这部分在每次调用该函数上是可以不同的。公共部分为函数的定义体，非公共部分为函数的参数。但你使用函数类型值做为另外一个函数的参数时，函数的非公共部分本身也是一个算法（函数），调用该函数时，每次你都可以传入不同函数类型值作为参数，这个函数称为**高阶函数–函数的参数也可以是另外一个函数。**

使用高级函数可以帮助你简化代码，它支持创建一个新的程序控制结构来减低代码重复。

比如，你打算写一个文件浏览器，你需要写一个API支持搜索给定条件的文件。首先，你添加一个方法，该方法可以通过查询包含给定字符串的文件，比如你可以查所有”.scala”结尾的文件。你可以定义如下的API：

object FileMatcher {

private def filesHere = (new java.io.File(".")).listFiles

def filesEnding(query : String) =

for (file <-filesHere; if file.getName.endsWith(query))

yield file

}

filesEnding 方法从本地目录获取所有文件（方法filesHere)，然后使用过滤条件（文件以给定字符串结尾）输出给定条件的文件。

到目前为止，这代码实现非常好也没有什么重复的代码。后来，你有需要使用新的过滤条件，文件名包含指定字符串，而不仅仅以某个字符串结尾的文件列表。你又实现了下面的API：

def filesContaining( query:String ) =

for (file <-filesHere; if file.getName.contains(query))

yield file

filesContaining和filesEnding 的实现非常类似，不同点在于一个使用endsWith,另一个使用contains函数调用。有过了一段时间，你有想支持使用正则表达式来查询文件，你有实现了下面的对象方法：

def filesRegex( query:String) =

for (file <-filesHere; if file.getName.matches(query))

yield file

这三个函数的算法非常类似，所不同的是过滤条件稍有不同，在Scala中我们可以**定义一个高阶函数，**将这三个不同过滤条件抽象称一个函数作为参数传给搜索算法，我们可以定义这个高阶函数如下：

def filesMatching( query:String,

matcher: (String,String) => Boolean) = {

for(file <- filesHere; if matcher(file.getName,query))

yield file

}

这个函数的第二个参数matcher的类型也为函数（如果你熟悉C#，类似于delegate)，该函数的类型为 (String,String ) =>Boolean，可以匹配任意使用两个String类型参数，返回值类型为Boolean的函数。使用这个辅助函数，我们可以重新定义filesEnding,filesContaining和filesRegex。

def filesEnding(query:String) =

filesMatching(query,\_.endsWith(\_))

def filesContaining(query:String)=

filesMatching(query,\_.contains(\_))

def filesRegex(query:String) =

filesMatching(query,\_.matches(\_))

这个新的实现和之前的实现已经简化了不少，实际上代码还可以简化，我们注意到参数query在filesMatching的作用只是把它传递给matcher参数，这种参数传递实际也是无需的，简化后代码如下：

object FileMatcher {

private def filesHere = (new java.io.File(".")).listFiles

def filesMatching(

matcher: (String) => Boolean) = {

for(file <- filesHere; if matcher(file.getName))

yield file

}

def filesEnding(query:String) =

filesMatching(\_.endsWith(query))

def filesContaining(query:String)=

filesMatching(\_.contains(query))

def filesRegex(query:String) =

filesMatching(\_.matches(query))

}

函数类型参数 \_.endsWith(query)，\_.contains(query)和\_.matches(query) 为函数闭包，因为它们绑定了一个自由变量query,因此我们可以看到闭包也可以用来简化代码。

## 柯里化函数

柯里化(Currying)，是把接受多个参数的函数变换成接受一个单一参数（***最初函数的第一个参数***）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

下面先给出一个普通的非柯里化的函数定义，实现一个加法函数：

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

plainOldSum: (x: Int, y: Int)Int

scala> plainOldSum(1,2)

res0: Int = 3

下面再使用“柯里化”技术来定义这个加法函数，原来函数使用一个参数列表，“柯里化”把函数定义为多个参数列表：

scala> def curriedSum(x:Int)(y:Int) = x + y

curriedSum: (x: Int)(y: Int)Int

scala> curriedSum (1)(2)

res0: Int = 3

当你调用curriedSum (1)(2)时，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，我们使用下面两个分开的定义在模拟curriedSum柯里化函数：

首先定义第一个函数：

scala> def first(x:Int) = (y:Int) => x + y

first: (x: Int)Int => Int

然后我们使用参数1调用这个函数来生成第二个函数（回忆前面定义的闭包）。

scala> val second=first(1)

second: Int => Int = <function1>

scala> second(2)

res1: Int = 3

first,second的定义演示了柯里化函数的调用过程，它们本身和curriedSum 没有任何关系，但是我们可以使用curriedSum 来定义second，如下：

scala> val onePlus = curriedSum(1)\_

onePlus: Int => Int = <function1>

下划线“\_” 作为第二参数列表的占位符， 这个定义的返回值为一个函数，当调用时会给调用的参数加一。

scala> onePlus(2)

res2: Int = 3

通过柯里化，你还可以定义多个类似onePlus的函数，比如twoPlus

scala> val twoPlus = curriedSum(2) \_

twoPlus: Int => Int = <function1>

scala> twoPlus(2)

res3: Int = 4

## 创建新的控制结构

对于支持函数作为“头等公民”的语言，你可以有效的创建新的控制结构即使该语言语法上固定的。你所要做的是**创建一个方法，该方法使用函数类型作为参数。**

比如： 下面为一个“双倍”的控制结构，这个“双倍”控制结构可以重复一个操作，然后返回结果。

scala> def twice (op:Double => Double, x:Double) =op(op(x))

twice: (op: Double => Double, x: Double)Double

scala> twice(\_ + 1, 5)

res0: Double = 7.0

上面调用twice ，其中 \_+1调用两次，也就是5调用两次+1，结果为7.

你在写代码时，如果发现某些操作需要重复多次，你就可以试着将这个重复操作写成新的控制结构，在前面我们定义过一个filesMatching函数：

def filesMatching(

matcher: (String) => Boolean) = {

for(file <- filesHere; if matcher(file.getName))

yield file

}

如果我们把这个函数进一步通用化，可以定义一个通用操作如下：

打开一个资源，然后对资源进行处理，最后释放资源，你可以为这个“模式”定义一个通用的控制结构如下：

def withPrintWriter (file: File, op: PrintWriter => Unit) {

val writer=new PrintWriter(file)

try{

op(writer)

}finally{

writer.close()

}

}

使用上面定义，我们使用如下调用：

withPrintWriter(

new File("date.txt"),

writer => writer.println(new java.util.Date)

)

使用这个方法的优点在于withPrintWriter，而不是用户定义的代码，withPrintWriter可以保证文件在使用完成后被关闭，也就是不可能发生忘记关闭文件的事件。这种技术成为“**租赁模式**”，这是因为这种类型的控制结构，比如withPrintWriter 将一个PrintWriter 对象“租”给op操作，当这个op操作完成后，它通知不再需要租用的资源，在finally中可以保证资源被释放，而无论op是否出现异常。

这里调用语法还是使用函数通常的调用方法，使用（）来列出参数，***在Scala中如果你调用函数只有一个参数，你可以使用{}来替代().比如下面两种语法是等价的***：

scala> println ("Hello,World")

Hello,World

scala> println { "Hello,world" }

Hello,world

上面第二种用法，使用{}替代了()，但这只适用在使用一个参数的调用情况。 前面定义withPrintWriter 函数使用了两个参数，因此不能使用{}来替代()，但如果我们使用柯里化重新定义下这个函数如下：

import scala.io.\_

import java.io.\_

def withPrintWriter (file: File)( op: PrintWriter => Unit) {

val writer=new PrintWriter(file)

try{

op(writer)

}finally{

writer.close()

}

}

将一个参数列表，变成两个参数列表，每个列表含一个参数，这样我们就可以使用如下语法来调用withPrintWriter

val file = new File("date.txt")

withPrintWriter(file){

writer => writer.println(new java.util.Date)

}

第一个参数我们还是使用（）（我们也可以使用{}),第二个参数我们使用{}来替代(),这样修改过的代码使得withPrintWriter 看起来和Scala内置的控制结构语法一样。

## 传名参数

上篇我们使用柯里化函数定义一个控制机构withPrintWriter ，它使用时语法调用有如Scala内置的控制结构：

val file = new File("date.txt")

withPrintWriter(file){

writer => writer.println(new java.util.Date)

}

不过仔细看一看这段代码，它和scala内置的if或while表达式还是有些区别的，withPrintWriter的{}中的函数是带参数的含有“writer=>”。 如果你想让它完全和if和while的语法一致，在Scala中可以使用传名参数来解决这个问题。

***（注：我们知道通常函数参数传递的两种模式，一是传值，一是引用。而这里是第三种按名称传递。）***

下面我们以一个具体的例子来说明传名参数的用法：

var assertionsEnabled=true

def myAssert(predicate: () => Boolean ) =

if(assertionsEnabled && !predicate())

throw new AssertionError

这个myAssert函数的参数为一个函数类型，如果标志assertionsEnabled为True时，mymyAssert 根据predicate 的真假决定是否抛出异常，如果assertionsEnabled 为false,则这个函数什么也不做。

这个定义没什么问题，但调用起来看起来却有些别扭，比如：

myAssert(() => 5 >3 )

还需要 ()=> ,你可以希望直接使用 5>3,但此时会报错：

scala> myAssert(5 >3 )

<console>:10: error: type mismatch;

found : Boolean(true)

required: () => Boolean

myAssert(5 >3 )

此时，我们可以把按值传递（上面使用的是按值传递，传递的是函数类型的值）参数修改为按名称传递的参数，修改方法，是使用 => 开始而不是 ()=>来定义函数类型，如下：

def myNameAssert(predicate: => Boolean ) =

if(assertionsEnabled && !predicate)

throw new AssertionError

此时你就可以直接使用下面的语法来调用myNameAssert：

myNameAssert(5>3)

此时就和Scala内置控制结构一样了，看到这里，你可能会想我为什么不直接把参数类型定义为Boolean，比如：

def boolAssert(predicate: Boolean ) =

if(assertionsEnabled && !predicate)

throw new AssertionError

调用也可以使用

boolAssert(5>3)

和myNameAssert 调用看起来也没什么区别，其实两者有着本质的区别，一个是传值参数，一个是传名参数，在调用boolAssert(5>3)时，5>3是已经计算出为true，然后传递给boolAssert方法，而myNameAssert(5>3)，表达式5>3没有事先计算好传递给myNameAssert，而是先创建一个函数类型的参数值，这个函数的apply方法将计算5>3，然后这个函数类型的值作为参数传给myNameAssert。

因此这两个函数一个明显的区别是，如果设置assertionsEnabled 为false, 然后试图计算 x/0 ==0,

scala> assertionsEnabled=false

assertionsEnabled: Boolean = false

scala> val x = 5

x: Int = 5

scala> boolAssert ( x /0 ==0)

java.lang.ArithmeticException: / by zero

... 32 elided

scala> myNameAssert ( x / 0 ==0)

可以看到boolAssert 抛出 java.lang.ArithmeticException: / by zero 异常，这是因为这是个传值参数，首先计算 x /0 ，而抛出异常，而 myNameAssert 没有任何显示，这是因为这是个传名参数，传入的是一个函数类型的值，不会先计算x /0 ==0,而在myNameAssert 函数体内，由于assertionsEnabled为false,传入的predicate没有必要计算(短路计算），因此什么也不会打印。如果我们把myNameAssert 修改下，把predicate放在前面：

scala> def myNameAssert1(predicate: => Boolean ) =

| if( !predicate && assertionsEnabled )

| throw new AssertionError

myNameAssert1: (predicate: => Boolean)Unit

scala> myNameAssert1 ( x/0 ==0)

java.lang.ArithmeticException: / by zero

at $anonfun$1.apply$mcZ$sp(<console>:11)

at .myNameAssert1(<console>:9)

... 32 elided

这个传名参数函数也抛出异常（你可以想想是为什么？）

前面的withPrintWriter 我们暂时没法使用传名参数，去掉writer=>,否则就难以实现“租赁模式”，不过我们可以看看下面的例子，设计一个withHelloWorld控制结构，这个withHelloWorld总打印一个“hello,world”

import scala.io.\_

import java.io.\_

def withHelloWorld ( op: => Unit) {

op

println("Hello,world")

}

val file = new File("date.txt")

withHelloWorld{

val writer=new PrintWriter(file)

try{

writer.println(new java.util.Date)

}finally{

writer.close()

}

}

withHelloWorld {

println ("Hello,Guidebee")

}

-----

Hello,world

Hello,Guidebee

Hello,world

可以看到withHelloWorld 的调用语法和Scala内置控制结构非常象了。

## 组合和继承-概述

定义一个新类的方法主要有两种模式：一个通过组合的方式，新创建的类通过引用其它类组合而成，通过这些引用类组合来完成新功能，而是通过继承的方式来扩展基类。为了更好的介绍Scala类的组合和继承，以及抽象类，无参数方法，扩展类，方法的重载等，我们打算使用一个现实的例子来说明，因此本篇首先定义需要解答的问题。

我们的需要是定义一个函数库，这个库用来定义在平面上（二维空间）布局元素，每个元素使用一个含有文字的矩形来表示。为方便起见，我们定义一个类构造工厂方法“elem”根据传入的数据来创建一个布局元素。这个方法的接口定义如下：

elem(s: String) : Element

可以看到，布局元素使用类型Element来构造其模型，你可以调用above,和beside 方法来创建一个新的布局元素，这个新的布局元素有两个已经存在的布局元素组合而成，例如：下面的表达式使用多个布局元素构造一个更大区域的布局元素：

val column1 = elem(&quot;Hello&quot;) above elem(&quot;\*\*\*&quot;)

val column2 = elem(&quot;\*\*&quot;) above (&quot;World&quot;)

column beside column2

将打印出下面结果：

Hello \*\*\*

\*\*\* world

这个例子使用布局元素，是非常好的一个例子可以用来说明一个对象可以使用更简单的对象通过组合的方式来构造。后面的几篇文章将以此为基础，我们将定义一些类，这些类支持使用数组，线段，矩形（简单部件）来构造，并定义组合算子（操作符)above和beside.

使用组合算子的概念来设计函数库是一种非常好的方法，它能回报以考虑在应用域构建对象的基础方法。什么是简单对象？用什么方式能让更多有趣的对象通过简单对象构造出来？组合子是怎么挂在一起的？什么是最通用的组合？它们满足任何有趣的规则吗？如果你对这些问题都有好的答案，你的库设计就在正轨上了。

## 组合和继承-抽象类

上一篇我们定义了我们需要解决的问题，我们首要的任务是定义Element类型，这个类型用来表示一个布局元素。由于每个元素为一个具有二维矩形形状的字符串，因此我们理所当然的可以定义个成员变量contents,用来表示这个二维布局元素的内容。这个元素我们使用一个字符串的数组来表示，这个数组的每个字符串元素代表布局的一行。也就是contents的类型为 Array[String]。

abstract class Element {

def contents: Array[String]

}

***在这个类中，成员contents使用了没有定义具体实现的方法来定义，这个方法称为一“*抽象方法*”，一个含有抽象方法的类必须定义成抽象类，也就是使用abstract关键字来定义类。***

abstract 修饰符表示所定义的类可能含有一些没有定义具体实现的抽象成员，因此你不能构建抽象类的实例，如果你试图这么做，编译器将报错：

scala> new Element

<console>:9: error: class Element is abstract; cannot be instantiated

new Element

后面的文章将继续介绍如何创建这个抽象类的子类，你可以构造这些子类的具体实例,这是因为这些子类实现了抽象成员。

要注意的***是contents方法本身没有使用abstract修饰符，一个没有定义实现的方法就是抽象方法，和Java不同的是，抽象方法不需要使用abstract修饰符来表示，只要这个方法没有具体实现，就是抽象方法，相反，如果该方法有具体实现，称为“具体(concrete)”方法。***

另一个术语用法需要分辨声明：declaration和定义：definition。类Element声明了抽象方法contents，但当前没有定义具体方法。然而下一节，我们要定义一些具体方法来加强Element。

## 组合和继承-定义无参数方法

作为下一步，我们将向Element添加显示宽度和高度的方法，height方法返回contents里的行数。width方法返回第一行的长度，或如果元素没有行记录，返回零。

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = if (height == 0) 0 else contents(0).length

}

请注意Element的三个方法没一个有参数列表，甚至连个空列表都没有,这种无参数方法在Scala里是非常普通的。相对的，带有空括号的方法定义，如def height(): Int，被称为空括号方法：(empty-paren method).

**Scala的惯例是在方法不需要参数并且只是读取对象状态时使用无参数方法。**

此外，我们也可以使用成员变量来定义width和height，例如：

abstract class Element {

def contents: Array[String]

val height = contents.length

val width = if (height == 0) 0 else contents(0).length

}

从使用这个类的客户代码来说，这两个实现是等价的，唯一的差别是使用成员变量的方法调用速度要快些，因为字段值在类被初始化的时候被预计算，而方法调用在每次调用的时候都要计算。换句话说，字段在每个Element对象上需要更多的内存空间。

特别是如果类的字段变成了访问函数，且访问函数是纯函数的，就是说它没有副作用并且不依赖于可变状态，那么类Element的客户不需要被重写。这称为统一访问原则：uniform access principle， 就是说客户代码不应受通过字段还是方法实现属性的决定的影响。

Scala代码可以调用Java函数和类，而Java没有使用“**统一访问原则**”，因此Java里是string.length()，不是string.length。为了解决这个问题，Scala对于无参数函数和空括号函数的使用上并不是区分得很严格。也就是，你可以用空括号方法重载无参数方法，并且反之亦可。你还可以在调用任何不带参数的方法时省略空的括号。例如，下面两行在Scala里都是合法的：

Array(1, 2, 3).toString

"abc".length

原则上Scala的函数调用中可以省略所有的空括号。但如果使用的函数不是纯函数，也就是说这个不带参数的函数可能修改对象的状态或是我们需要利用它的一些副作用（比如打印到屏幕，读写I/o)，一般的建议还是使用空括号，比如：

"hello".length // 没有副作用，所以无须()

println() // 最好别省略()

总结起来***，Scala里定义不带参数也没有副作用的方法为无参数方法***，也就是说，省略空的括号，是鼓励的风格。另一方面，永远不要定义没有括号的带副作用的方法，因为那样的话方法调用看上去会像选择一个字段。

如果你调用的函数执行了某个操作就使用括号，但如果仅提供了对某个属性的访问，那么省略括号。

## 组合和继承-扩展类

我们需要能够创建新的布局元素对象，前面定义的Element为抽象类，不能直接用来创建该类的对象，因此我们需要创建Element的子类。这些子类需要实现Element类定义的抽象函数。

Scala中派生子类的方法和Java一样，也是通过extends关键字。比如定义一个ArrayElement:

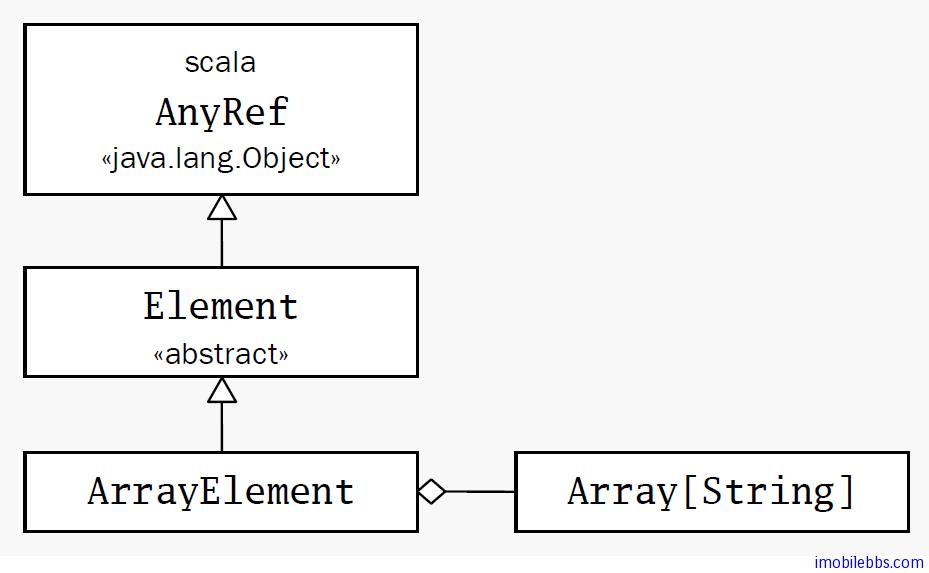
class ArrayElement(conts: Array[String]) extends Element {

def contents: Array[String] = conts

}

其中extends具有两个功效，一是让ArrayElement继承所有Element类的非私有成员，第二使得ArrayElement成为Element的一个子类。而Element称为ArrayElement的父类。

如果你在定义类时没有使用extends关键字，在Scala中这个定义类缺省继承自scala.AnyRef，如同在Java中缺省继承自java.lang.Object. 这种继承关系如下图：



这幅图中也显示了ArrayElement和Array[String]之间的“组合”关系”(composition),类ArrayElement 中定义了对Array[String]类型对象的一个引用。

ArrayElement继承了Element的所有非私有成员，同时定义了一个contents函数，这个函数中其父类（基类）中是抽象的，因此可以说ArrayElement中的contents函数实现了父类中的这个抽象函数，也可以说“重载”(override)了父类中的同名函数。ArrayElement继承了Element的width和height方法，因此你可以使用 ArrayElement.width来查询宽度，比如：

scala> val ae=new ArrayElement(Array("hello","world"))

ae: ArrayElement = ArrayElement@729c1e43

scala> ae.width

res0: Int = 5

派生也意味着子类的值可以用在任何可以使用同名父类值的地方，比如：

val e: Element = new ArrayElement(Array("hello"))

## 组合和继承-重载成员函数和方法

和Java稍有不同的一点是，Scala中成员函数和成员变量地位几乎相同，而且也处在同一个命名空间，也就是***Scala中不允许定义同名的成员函数和成员变量***，但带来的一个好处是，***可以使用成员变量来重载一个不带参数的成员函数***。比如，接着前面的例子，你可以通过一个成员变量来实现基类中定义的抽象函数contents.

class ArrayElement(conts: Array[String]) extends Element {

val contents: Array[String] = conts

}

可以看到，这是使用成员变量来实现基类中不带参数的抽象函数的一个非常恰当的例子。 Scala中的这种实现是Java语言所不支持的，一般来说只有两个不同的命名空间来定义类，而Java可以有四个，Scala支持的两个命名空间如下：

* 值（字段，方法，包还有单例对象）
* 类型（类和Trait名）

Scala把字段和方法放进同一个命名空间的理由很清楚，因为这样你就可以使用val重载无参数的方法。

Scala里面，字段和方法共享相同命名空间。这样就可以引用包，而不仅仅是引入类型名以及单例对象的字段和方法。

## 组合和继承-定义参数化成员变量

我们回到前面定义的类ArrayElement,它有一个参数conts，其唯一的目的是用来复制到contents成员变量。而参数名称conts是为了让它看起来和成员变量contents类似，而有不至于和成员变量名冲突。

***Scala支持使用参数化成员变量***，也就是把参数和成员变量定义合并到一起来避免上述代码：

class ArrayElement(val contents: Array[String])

extends Element {

}

要注意的是，现在参数contents前面加上了 val关键字，这是前面使用同名参数和同名成员变量的一个缩写形式。使用val定义了一个无法重新赋值的成员变量。这个成员变量初始值为参数的值，可以在类的外面访问这个成员变量。它的一个等效的实现如下：

class ArrayElement(val x123: Array[String])

extends Element {

val contents: Array[String] = x123

}

Scala 也允许你使用var关键字来定义参数化成员变量，使用var定义的成员变量，可以重新赋值。

此外Scala也允许你使用 private, protected ,override来修饰参数化成员变量，和你定义普通的成员变量的用法一样。 比如：

class Cat {

val dangerous =false

}

class Tiger (

override val dangerous: Boolean,

private var age: Int

) extends Cat

这段代码中Tiger的定义其实为下面类定义的一个缩写：

class Tiger(param1: Boolean, param2: Int) extends Cat {

override val dangerous = param1

private var age = param2

}

两个成员都初始化自相应的参数。我们任意选择了这些参数名，param1和param2。重要的是它们不会与范围内的任何其它名称冲突。

## 组合和继承-调用基类的构造函数

前面我们定义了两个类，一个为抽象类Element ，另外一个为派生的实类ArrayElement. 或许你打算再构造一个新类，这个类使用单个字符串来构造布局元素，使用面向对象的编程方法使得构造这种新类非常容易。比如下面的LineElement类：

class LineElement(s:String) extends ArrayElement(Array(s)) {

override def width = s.length

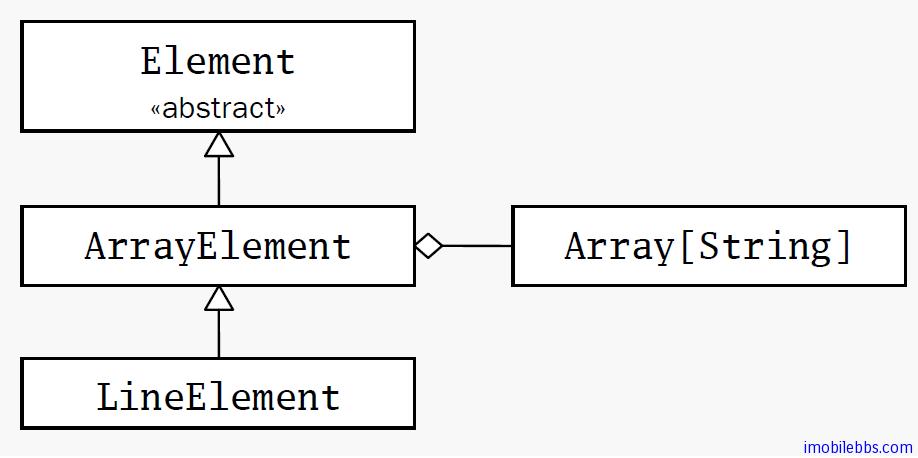
override def height = 1

}

由于LineElement扩展了ArrayElement，并且ArrayElement的构造器带一个参数（Array[String]），LineElement需要传递一个参数到它的基类的主构造器。要调用基类构造器，只要把你要传递的参数或参数列表放在基类名之后的括号里即可。例如，类LineElement传递了Array(s)到ArrayElement的主构造器，把它放在基类ArrayElement的名称后面的括号里：

... extends ArrayElement(Array(s)) ...

有了新的子类，布局元素的继承级别现在看起来就如下图所示：



## 组合和继承-使用override修饰符

在前面的例子LineElement使用了override来修饰width和height成员变量，***在Scala中需要使用override来重载父类的一个非抽象成员***，实现抽象成员无需使用override，如果子类没有重载父类中的成员，不可以使用override修饰符。

这个规则可以帮助编译器发现一些难以发现的错误，可以增强系统安全进化。比如，如果你把height拼写错误为hight,使用override编译器会报错。

这个规则对于系统的演讲尤为重要，假设你定义了一个2D图形库。你把它公开，并广泛使用。库的下一个版本里你想在你的基类Shape里增加一个新方法：

def hidden(): Boolean

你的新方法将被用在许多画图方法中去决定是否需要把形状画出来，这将可以大大提高系统绘图的性能，但你不可以冒着破坏客户代码的风险做这件事。毕竟客户说不定已经使用不同的hidden实现定义了Shape的子类。或许客户的方法实际上是让对象消失而不是检测是否对象是隐藏的。因为这两个版本的hidden互相重载，你的画图方法将停止对象的消失，这可真不是你想要的！

如果图形库和它的用户是用Scala写的，那么客户的hidden原始实现就不会有override修饰符，因为这时候还没有另外一个使用那个名字的方法。一旦你添加了hidden方法到你Shape类的第二个版本，客户的重编译将给出像下列这样的错误：

.../Shapes.scala:6: error: error overriding method

hidden in class Shape of type ()Boolean;

method hidden needs 'override' modifier

def hidden(): Boolean =

也就是说，代之以错误的执行，你的客户将得到一个编译期错误，这常常是更可取的。

## 组合和继承-多态和动态绑定

在前面的例子我们看到类型为Element的变量可以保存ArrayElement类型的对象，这种现象称为“多态”。也就是基类类型的变量可以保存其子类类型的对象，到目前为止我们定义了两个Element的子类，ArrayElement和LineElement。你还可以定义其它子类，比如：

class UniformElement (ch :Char,

override val width:Int,

override val height:Int

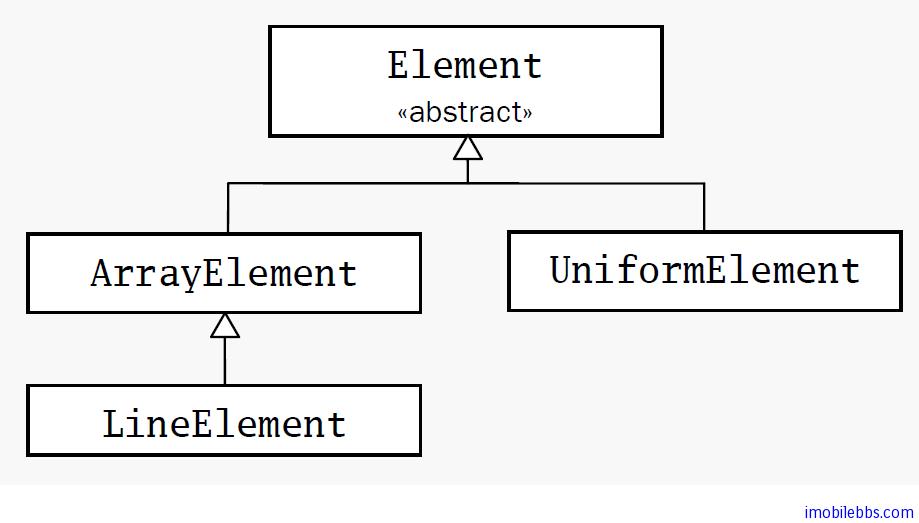
) extends Element{

private val line=ch.toString \* width

def contents = Array.fill(height)(line)

}

结合前面定义的类定义，我们就有了如下图所示的类层次关系：



Scala将接受所有的下列赋值，因为赋值表达式的类型符合定义的变量类型：

val e1: Element = new ArrayElement(Array("hello", "world"))

val ae: ArrayElement = new LineElement("hello")

val e2: Element = ae val

e3: Element = new UniformElement('x', 2, 3)

若你检查继承层次关系，你会发现这四个val定义的每一个表达式，等号右侧表达式的类型都在将被初始化的等号左侧的val类型的层次之下。

另一方面，如果调用变量（对象）的方法或成员变量，这个过程是一个动态绑定的过程，也就是说调用哪个类型的方法取决于运行时变量当前的类型，而不是定义变量的类型。

为了显示这种行为，我们在Element中添加一个demo方法，定义如下：

abstract class Element {

def demo() {

println("Element's implementation invoked")

}

}

class ArrayElement extends Element {

override def demo() {

println("ArrayElement's implementation invoked")

}

}

class LineElement extends ArrayElement {

override def demo() {

println("LineElement's implementation invoked")

}

}

// UniformElement inherits Element’s demo

class UniformElement extends Element

如果你使用交互式Scala解释器来测试，你可以定义如下的方法：

def invokeDemo(e: Element) {

e.demo()

}

下面我们分别使用ArrayElement, LineElement和UniformElement来调用这个方法：

scala> invokeDemo(new ArrayElement)

ArrayElement's implementation invoked

scala> invokeDemo(new LineElement)

LineElement's implementation invoked

scala> invokeDemo(new UniformElement)

Element's implementation invoked

可以看到由于ArrayElement和LineElement重载了Element的demo方法，因此调用invokeDemo时由于“动态绑定”因此会调用这些子类的demo方法，而由于UniformElement没有重载Element的demo方法，动态绑定时也会调用UniformElement的demo方法（但此时实际为基类的demo方法）

## 组合和继承-定义final成员

在定义类的继承关系时，有时你不希望基类的某些成员被子类重载，和Java类似，在Scala中也是使用final来修饰类的成员。比如在前面的ArrayElement例子，在demo方法前加上final修饰符，

class ArrayElement extends Element {

final override def demo() {

println("ArrayElement's implementation invoked")

}

}

在定义类的继承关系时，有时你不希望基类的某些成员被子类重载，和Java类似，在Scala中也是使用final来修饰类的成员。比如在前面的ArrayElement例子，在demo方法前加上final修饰符，

scala> class LineElement extends ArrayElement {

| override def demo() {

| println("LineElement's implementation invoked")

| }

|

| }

<console>:10: error: overriding method demo in class ArrayElement of type ()Unit;

method demo cannot override final member

override def demo() {

如果你希望某个类不可以派生子类，则可以在类定义前加上final修饰符：

final class ArrayElement extends Element {

override def demo() {

println("ArrayElement's implementation invoked")

}

}

此时如果还是重载LineElement的demo函数，则会报错：

scala> class LineElement extends ArrayElement {

| override def demo() {

| println("LineElement's implementation invoked")

| }

|

| }

<console>:9: error: illegal inheritance from final class ArrayElement

class LineElement extends ArrayElement {

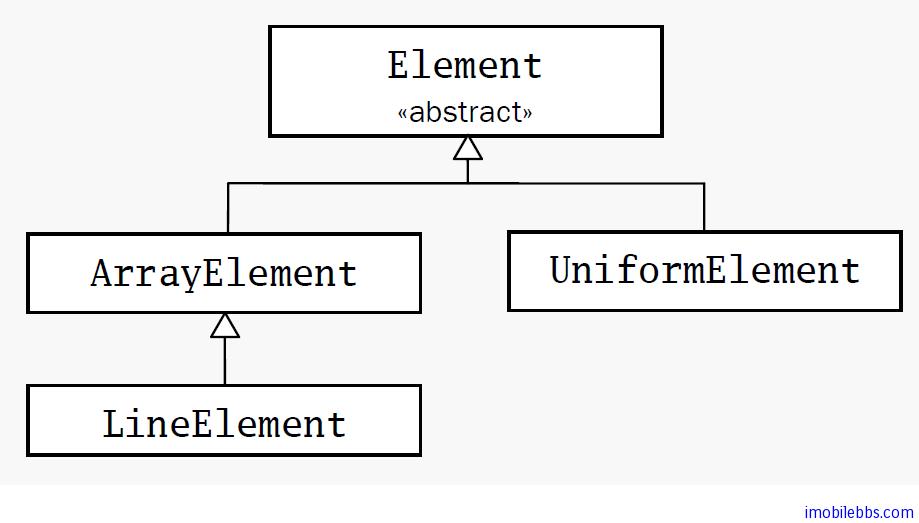
## 组合和继承-使用组合还是继承

前面我们说过，构建新类的两个基本方法是组合和继承，如果你的主要目的是代码重用，那么最好使用组合的方法构造新类，使用继承的方法构造新类造成的可能问题是，无意的修改基类可能会破坏子类的实现。

关于继承关系你可以问自己一个问题，是否它建模了一个is-a关系。例如，说ArrayElement是Element是合理的。你能问的另一个问题是，是否客户想要把子类类型当作基类类型来用。

前一个版本中，LineElement与ArrayElement有一个继承关系，从那里继承了contents。现在它在ArrayElement的例子里，我们的确期待客户会想要把ArrayElement当作Element使用。

请看下面的类层次关系图：



## 组合和继承-实现Element类的above、beside、toString()方法

我们接着实现类Element的其它方法，如above, beside和toString方法。

above方法，意味着把一个布局元素放在另外一个布局元素的上方，也就是把这两个元素的contents的内容连接起来。我们首先实现above函数的第一个版本：

def above(that: Element) :Element =

new ArrayElement(this.contents ++ that.contents)

Scala中Array 使用Java Array来实现，但添加很多其它方法，尤其是Scala中Array可以转换为scala.Seq类的实例对象，scala.Seq为一个序列结构并提供了许多方法来访问和转换这个序列。

实际上上面above的实现不是十分有效，因为它不允许你把不同长度的布局元素叠加到另外一个布局元素上面，但就目前来说，我们还是暂时使用这个实现，只使用同样长度的布局元素，后面再提供这个版本的增强版本。

下面我们再实现类Element的另外一个beside方法，把两个布局元素并排放置，和前面一样，为简单起见，我们暂时只考虑相同高度的两个布局元素：

def beside(that: Element) :Element = {

val contents = new Array[String](this.contents.length)

for(i <- 0 until this.contents.length)

contents(i)=this.contents(i) + that.contents(i)

new ArrayElement(contents)

}

尽管上面的实现满足beside要求，但采用的还是指令式编程，我们使用函数说编程可以实现下面简化代码：

def beside(that: Element) :Element = {

new ArrayElement(

for(

(line1,line2) <- this.contents zip that.contents

) yield line1+line2

)

}

这里我们使用了Array的zip 操作符，可以用来将两个数组转换成二元组的数组，zip 分别取两个数组对应的元素组成一个新的二元祖，比如：

scala> Array( 1,2,3) zip Array("a","b")

res0: Array[(Int, String)] = Array((1,a), (2,b))

如果一个数组长度大于另外一个数组，多余的元素被忽略。 for 的yield部分用来构成一个新元素。

最后我们实现Element的toString方法用来显示布局元素的内容：

override def toString = contents mkString "\n"

这里使用mkString函数，这个函数可以应用到任何序列数据结构（包括数组），也就是把contents的每个元素调用toString,然后使用”\n”分隔。

## 组合和继承-定义factory对象

到目前为止，我们定义了关于布局元素类的一个层次结构。你可以把包含这个层次关系的类作为API接口提供给其它应用，但有时你可以希望对函数库的用户隐藏这种层次关系，这通常可以使用factory(构造工厂）对象来实现。一个factory对象定义了用来构造其它对象的函数。库函数的用户可以通过工厂对象来构造新对象，而不需要通过类的构造函数来创建类的实例。使用工厂对象的好处是，可以统一创建对象的接口并且隐藏被创建对象具体是如何来表示的。这种隐藏可以使得你创建的函数库使用变得更简单和易于理解，也正是隐藏部分实现细节，可以使你有机会修改库的实现而不至于影响库的接口。

实现factory对象的一个基本方法是采用singleton模式，在Scala中，可以使用类的伴随对象(companion 对象）来实现。比如：

object Element {

def elem(contents: Array[String]):Element =

new ArrayElement(contents)

def elem(chr:Char, width:Int, height:Int) :Element =

new UniformElement(chr,width,height)

def elem(line:String) :Element =

new LineElement(line)

}

我们先把之前Element的实现列在这里：

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = if (height == 0) 0 else contents(0).length

def above(that: Element) :Element =

new ArrayElement(this.contents ++ that.contents)

def beside(that: Element) :Element = {

new ArrayElement(

for(

(line1,line2) <- this.contents zip that.contents

) yield line1+line2

)

}

override def toString = contents mkString "\n"

}

有了object Element（类Element的伴随对象），我们可以利用Element对象提供的factory方法，重新实现类Element的一些方法：

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = if (height == 0) 0 else contents(0).length

def above(that: Element) :Element =

Element.elem(this.contents ++ that.contents)

def beside(that: Element) :Element = {

Element.elem(

for(

(line1,line2) <- this.contents zip that.contents

) yield line1+line2

)

}

override def toString = contents mkString "\n"

}

这里我们重写了above和beside方法，使用伴随对象的factory方法Element.elem替代new 构造函数。

这样修改之后，库函数的用户不要了解Element的继承关系，甚至不需要知道类ArrayElement，LineElement定义的存在，为了避免用户直接使用ArrayElement或LineElement的构造函数来构造类的实例，因此我们可以把ArrayElement，UniformElement和LineElement 定义为私有，定义私有可以也可以把它们定义在类Element内部（嵌套类）。下面为这种方法的使用：

object Element {

private class ArrayElement(val contents: Array[String])

extends Element {

}

private class LineElement(s:String) extends ArrayElement(Array(s)) {

override def width = s.length

override def height = 1

}

private class UniformElement (ch :Char,

override val width:Int,

override val height:Int

) extends Element{

private val line=ch.toString \* width

def contents = Array.fill(height)(line)

}

def elem(contents: Array[String]):Element =

new ArrayElement(contents)

def elem(chr:Char, width:Int, height:Int) :Element =

new UniformElement(chr,width,height)

def elem(line:String) :Element =

new LineElement(line)

}

## 组合和继承-定义heighten和widen函数

我们还需要最后一个改进，之前的Element实现不够完善，只支持同样高度和同样宽度的Element使用above和beside函数，比如下面的代码将无法正常工作，因为组合元素的第二行比第一行要长：

new ArrayElement(Array("hello")) above

new ArrayElement(Array("world!"))

与之相似的，下面的表达式也不能正常工作，因为第一个ArrayElement高度为二，而第二个的高度只是一

new ArrayElement(Array("one", "two")) beside

new ArrayElement(Array("one"))

下面代码展示了一个私有帮助方法，widen，能够带个宽度做参数并返回那个宽度的Element。结果包含了这个Element的内容，居中，左侧和右侧留需带的空格以获得需要的宽度。这段代码还展示了一个类似的方法，heighten，能在竖直方向执行同样的功能。widen方法被above调用以确保Element堆叠在一起有同样的宽度。类似的，heighten方法被beside调用以确保靠在一起的元素具有同样的高度。有了这些改变，布局库函数可以使用了

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = if (height == 0) 0 else contents(0).length

def above(that: Element) :Element =

Element.elem(this.contents ++ that.contents)

def beside(that: Element) :Element = {

Element.elem(

for(

(line1,line2) <- this.contents zip that.contents

) yield line1+line2

)

}

def widen(w: Int): Element =

if (w <= width) this

else {

val left = Element.elem(' ', (w - width) / 2, height)

var right = Element.elem(' ', w - width - left.width, height)

left beside this beside right

}

def heighten(h: Int): Element =

if (h <= height) this

else {

val top = Element.elem(' ', width, (h - height) / 2)

var bot = Element.elem(' ', width, h - height - top.height)

top above this above bot

}

override def toString = contents mkString "\n"

}

## 组合和继承-小结

前面我们基本完成了布局元素的函数库，现在我们就可以写个程序来使用这个函数库，下面显示螺旋线的程序如下：

object Spiral {

val space = elem (" ")

val corner = elem ("+")

def spiral(nEdges:Int, direction:Int): Element = {

if(nEdges==1)

elem("+")

else{

val sp=spiral(nEdges -1, (direction +3) % 4)

def verticalBar = elem ('|',1, sp.height)

def horizontalBar = elem('-',sp.width,1)

if(direction==0)

(corner beside horizontalBar) above (sp beside space)

else if (direction ==1)

(sp above space) beside ( corner above verticalBar)

else if(direction ==2 )

(space beside sp) above (horizontalBar beside corner)

else

(verticalBar above corner) beside (space above sp)

}

}

def main(args:Array[String]) {

val nSides=args(0).toInt

println(spiral(nSides,0))

}

}

因为Sprial为一单例对象，并包含main方法，因此它为一Scala应用程序，可以在命令行使用scala Sprial xx来运行这个应用。

root@mail:~/scala# scala Spiral 5

+----

|

| ++

| |

+--+

root@mail:~/scala# scala Spiral 23

+----------------------

|

| +------------------+

| | |

| | +--------------+ |

| | | | |

| | | +----------+ | |

| | | | | | |

| | | | +------+ | | |

| | | | | | | | |

| | | | | +--+ | | | |

| | | | | | | | | | |

| | | | | ++ | | | | |

| | | | | | | | | |

| | | | +----+ | | | |

| | | | | | | |

| | | +--------+ | | |

| | | | | |

| | +------------+ | |

| | | |

| +----------------+ |

| |

+--------------------+

这个例子的完整代码如下：

object Element {

private class ArrayElement(val contents: Array[String])

extends Element

private class LineElement(s:String) extends Element {

val contents=Array(s)

override def width = s.length

override def height = 1

}

private class UniformElement (ch :Char,

override val width:Int,

override val height:Int

) extends Element{

private val line=ch.toString \* width

def contents = Array.fill(height)(line)

}

def elem(contents: Array[String]):Element =

new ArrayElement(contents)

def elem(chr:Char, width:Int, height:Int) :Element =

new UniformElement(chr,width,height)

def elem(line:String) :Element =

new LineElement(line)

}

import Element.elem

abstract class Element {

def contents: Array[String]

def height: Int = contents.length

def width: Int = contents(0).length

def above(that: Element) :Element = {

val this1=this widen that.width

val that1=that widen this.width

elem (this1.contents ++ that1.contents)

}

def beside(that: Element) :Element = {

val this1=this heighten that.height

val that1=that heighten this.height

Element.elem(

for(

(line1,line2) <- this1.contents zip that1.contents

) yield line1+line2

)

}

def widen(w: Int): Element =

if (w <= width) this

else {

val left = Element.elem(' ', (w - width) / 2, height)

var right = Element.elem(' ', w - width - left.width, height)

left beside this beside right

}

def heighten(h: Int): Element =

if (h <= height) this

else {

val top = Element.elem(' ', width, (h - height) / 2)

var bot = Element.elem(' ', width, h - height - top.height)

top above this above bot

}

override def toString = contents mkString "\n"

}

object Spiral {

val space = elem (" ")

val corner = elem ("+")

def spiral(nEdges:Int, direction:Int): Element = {

if(nEdges==1)

elem("+")

else{

val sp=spiral(nEdges -1, (direction +3) % 4)

def verticalBar = elem ('|',1, sp.height)

def horizontalBar = elem('-',sp.width,1)

if(direction==0)

(corner beside horizontalBar) above (sp beside space)

else if (direction ==1)

(sp above space) beside ( corner above verticalBar)

else if(direction ==2 )

(space beside sp) above (horizontalBar beside corner)

else

(verticalBar above corner) beside (space above sp)

}

}

def main(args:Array[String]) {

val nSides=args(0).toInt

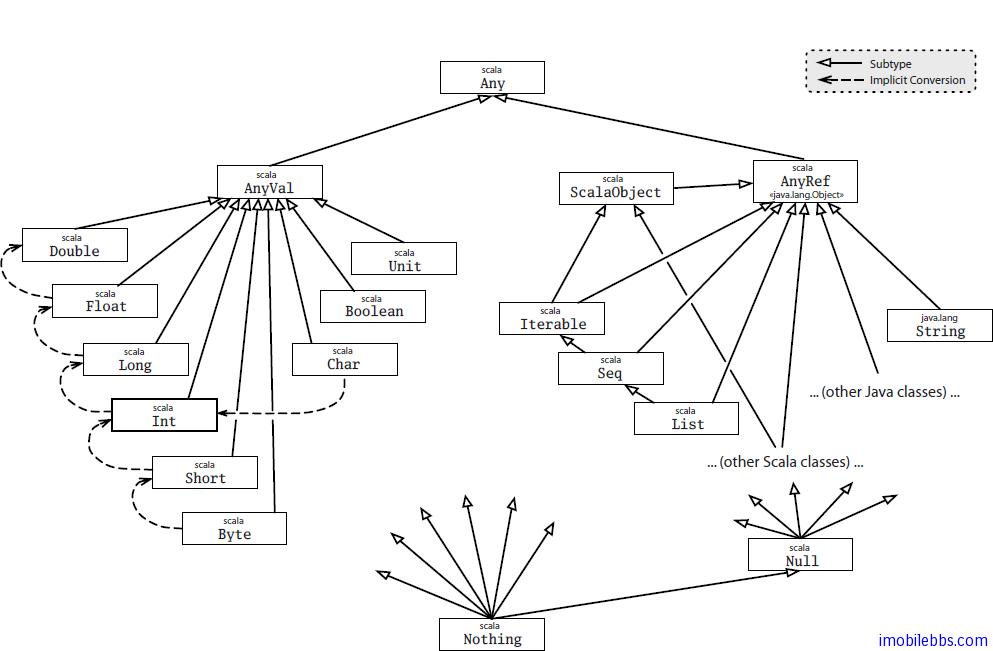
println(spiral(nSides,0))

}

}

## Scala的类层次关系

前面我们介绍了Scala的类的继承，本篇我们介绍Scala语言自身定义的类的层次关系，在Scala中，所有的类都有一个公共的基类称为Any，此外还定义了所有类的子类Nothing，下面的图给出的Scala定义的类层次关系的一个概要：



由于所有的类都继承自Any，因此Scala中的对象都可以使用==,!=,或equals来比较，使用##或hashCode给出hash值，使用toString 转为字符串。Any的==和!=定位为fianl，因此不可以被子类重载。==实际上和equals等价，!=和equals的否定形式等价，因此重载equals可以修改==和!=的定义。

根类Any有两个子类：AnyVal和AnyRef。AnyVal是Scala里每个内建值类型的父类。有九个这样的值类型：Byte，Short，Char，Int，Long，Float，Double，Boolean和Unit。其中的前八个对应到Java的基本数值类型，它们的值在运行时表示成Java的类型。Scala里这些类的实例都写成字面量。例如，42是Int的实例，’x’是Char的实例，false是Boolean的实例。值类型都被定义为即是抽象的又是final的，你不能使用new创造这些类的实例。

scala> new Int

<console>:8: error: class Int is abstract; cannot be instantiated

new Int

^

scala>

值类支持作为方法的通用的数学和布尔操作符。例如，Int有名为+和\*的方法，Boolean有名为||和&&的方法。值类也从类Any继承所有的方法。你可以在解释器里测试:

scala> 42 toString

res3: String = 42

scala> 42.hashCode

res6: Int = 42

可以看到Scala的值类型之间的关系是扁平的，所有的值类都是scala.AnyVal的子类型，但是它们不是互相的子类。代之以它们不同的值类类型之间可以隐式地互相转换。例如，需要的时候，类scala.Int的实例可以自动放宽（通过隐式转换）到类scala.Long的实例。隐式转换还用来为值类型添加更多的功能。例如，类型Int支持以下所有的操作：

scala> 42 max 43

res0: Int = 43

scala> 42 min 43

res1: Int = 42

scala> 1 until 5

res2: scala.collection.immutable.Range = Range(1, 2, 3, 4)

scala> 1 to 5

res3: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 3.abs

res4: Int = 3

scala> (-3).abs

res5: Int = 3

这里解释其工作原理：方法min，max，until，to和abs都定义在类scala.runtime.RichInt里，并且有一个从类Int到RichInt的隐式转换。当你在Int上调用没有定义在Int上但定义在RichInt上的方法时，这个转换就被应用了。

类Any的另一个子类是类AnyRef。这个是Scala里所有引用类的基类。正如前面提到的，在Java平台上AnyRef实际就是类java.lang.Object的别名。因此Java里写的类和Scala里写的都继承自AnyRef。如此说来，你可以认为java.lang.Object是Java平台上实现AnyRef的方式。因此，尽管你可以在Java平台上的Scala程序里交换使用Object和AnyRef，推荐的风格是在任何地方都只使用AnyRef。

Scala类与Java类不同在于它们还继承自一个名为ScalaObject的特别的Marker Trait（Trait我们在后面再进一步解释）

## Scala基本数据类型的实现方法

Scala的基本数据类型是如何实现的？实际上，Scala以与Java同样的方式存储整数：把它当作32位的字类型。这对于有效使用JVM平台和与Java库的互操作性方面来说都很重要，。标准的操作如加法或乘法都被实现为数据类型基本运算操作。然而，当整数需要被当作（Java）对象看待的时候，Scala使用了“备份”类java.lang.Integer。如在整数上调用toString方法或者把整数赋值给Any类型的变量时，就会这么做，需要的时候，Int类型的整数能自动转换为java.lang.Integer类型的“装箱整数(boxed integer)”。

这些听上去和Java的box操作很像，实际上它们也很像，但这里有一个重要的差异，Scala使用box操作比在Java中要少的多：

// Java代码

boolean isEqual(int x,int y) {

return x == y;

}

System.out.println(isEqual(421,421));

Scala 的 == 设计出自动适应变量类型的操作，对值类型来说，就是自然的（数学或布尔）相等。对于引用类型，==被视为继承自Object的equals方法的别名。比如对于字符串比较：

scala> val x = "abcd".substring(2)

x: String = cd

scala> val y = "abcd".substring(2)

y: String = cd

scala> x==y

res0: Boolean = true

而在Java里，，x与y的比较结果将是false。程序员在这种情况应该用equals，不过它容易被忘记。

然而，有些情况你需要使用引用相等代替用户定义的相等。例如，某些时候效率是首要因素，你想要把某些类哈希合并：hash cons然后通过引用相等比较它们的实例，为这种情况，类AnyRef定义了附加的eq方法，它不能被重载并且实现为引用相等（也就是说，它表现得就像Java里对于引用类型的==那样）。同样也有一个eq的反义词，被称为ne。例如：

scala> val x =new String("abc")

x: String = abc

scala> val y = new String("abc")

y: String = abc

scala> x == y

res0: Boolean = true

scala> x eq y

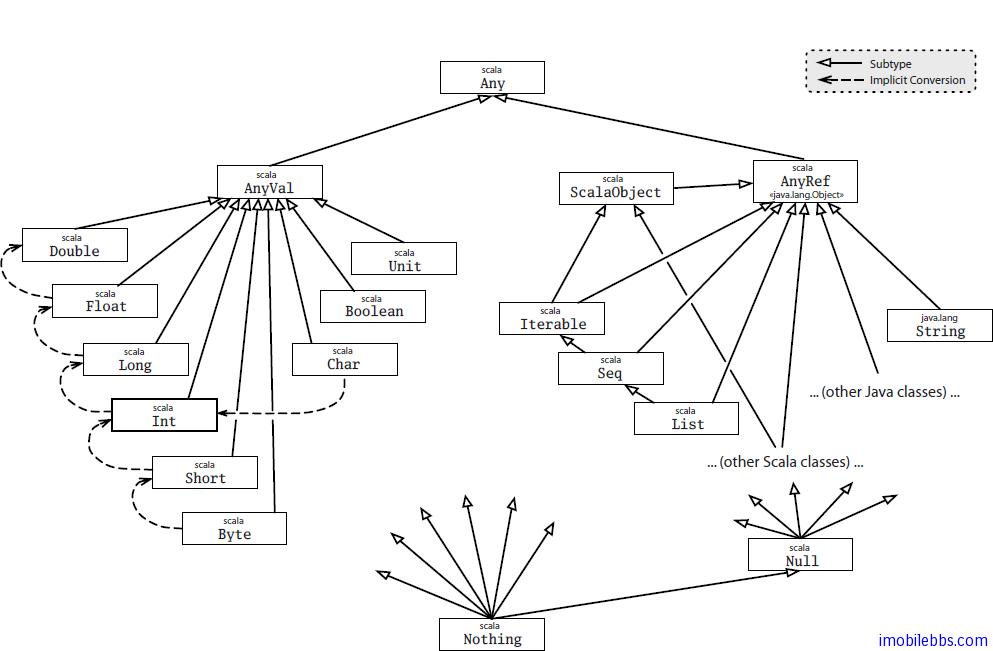
res1: Boolean = false

scala> x ne y

res2: Boolean = true

## 所有类的公共子类-底层类型

前面我们给出了Scala的类的一个关系图：



在这张图的最下方我们可以看到有两个类，scala.Null和scala.Nothing. 这两个类的作用是Scala支持统一方式用来处理面向对象的一些边角情况。因为它们在类层次图的下方，因此也称为底层类型

类Null代表null引用，它是所有引用类（每个由AnyRef派生的类）的子类。Null和值类型不兼容，也就是比如说，你不能把null赋值给一个整数类型变量：

scala> val i:Int=null

<console>:7: error: an expression of type Null is ineligible for implicit conversion

val i:Int=null

Nothing类型为图中类层次关系的最下方，它是所有其他类的子类，然而这个类型没有任何实例（也就是没有任何值对应Nothing类型）前面提到，Nothing类型的一个用法是示意应用程序非正常终止，比如Predef的有一个error方法：

def error(message:String) :Nothing =

throw new RuntimeException(message)

Nothing类型为图中类层次关系的最下方，它是所有其他类的子类，然而这个类型没有任何实例（也就是没有任何值对应Nothing类型）前面提到，Nothing类型的一个用法是示意应用程序非正常终止，比如Predef的有一个error方法：

def error(message:String) :Nothing =

throw new RuntimeException(message)

error的返回类型就是Nothing,告诉调用者该方法没有正常退出（抛出异常）。正因为Nothing为所有其它类型的子类，你可以灵活使用如error这样的函数。比如：

def divide(x:Int,y:Int):Int=

if(y!=0) x/y

else error("Cannot divide by Zero")

if “then”分支的类型为Int（x/y),else分支的类型为error返回值，其类型为Nothing，因为Nothing为所有类型的子类，它也是Int的子类，因此divide的类型为Int。

## Trait的基本概念

在Scala中Trait为重用代码的一个基本单位。一个Traits封装了方法和变量，和Interface相比，它的方法可以有实现，这一点有点和抽象类定义类似。但和类继承不同的是，Scala中类继承为单一继承，也就是说子类只能有一个父类。当一个类可以和多个Trait混合，这些Trait定义的成员变量和方法也就变成了该类的成员变量和方法，由此可以看出Trait集合了Interface和抽象类的优点，同时又没有破坏单一继承的原则。

下面我们来看看Trait的基本用法：

定义一个Trait的方法和定义一个类的方法非常类似，除了它使用trait而非class关键字来定义一个trait.

trait Philosophical{

def philosophize() {

println("I consume memeory, therefor I am!")

}

}

这个Trait名为Philosophical。它没有声明基类，因此和类一样，有个缺省的基类AnyRef。它定义了一个方法，叫做philosophize。这是个简单的Trait，仅够说明Trait如何工作。

一但定义好Trait，它就可以用来和一个类混合，这可以使用extends或with来混合一个trait.例如：

class Frog extends Philosophical{

override def toString="gree"

}

这里我们使用extends为Frog添加Philosophical Trait属性，因此Frog 缺省继承自Philosophical的父类AnyRef，这样Frog类也具有了Philosophical的性质（因此Trait也可以翻译成特质，但后面我们还是继续使用Trait原文）。

scala> val frog = new Frog

frog: Frog = green

scala> frog.philosophize

I consume memeory, therefor I am!

可以看到Frog添加了Philosophical（哲学性）也具有了哲学家的特性，可以说出类似“我思故我在”的话语了。和Interface一样，Trait也定义一个类型，比如：

scala> val phil:Philosophical = frog

phil: Philosophical = green

scala> phil.philosophize

I consume memeory, therefor I am!

变量phil的类型为Philosophical。

如果你需要把某个Trait添加到一个有基类的子类中，使用extends继承基类，而可以通过with 添加Trait。比如：

class Animal

class Frog extends Animal with Philosophical{

override def toString="green"

}

还是和Interface类似，可以为某个类添加多个Trait属性，此时使用多个with即可，比如：

class Animal

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs{

override def toString="green"

}

目前为止你看到的例子中，类Frog都继承了Philosophical的philosophize实现。此外Frog也可以重载philosophize方法。语法与重载基类中定义的方法一样。

class Animal

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs{

override def toString="green"

override def philosophize() {

println("It ain't easy being " + toString + "!")

}

}

因为Frog的这个新定义仍然混入了特质Philosophize，你仍然可以把它当作这种类型的变量使用。但是由于Frog重载了Philosophical的philosophize实现，当你调用它的时候，你会得到新的回应：

scala> val phrog:Philosophical = new Frog

phrog: Philosophical = green

scala> phrog.philosophize

It ain't easy being green!

这时你或许推导出以下结论：***Trait就像是带有具体方法的Java接口***，不过其实它能做的更多。**Trait可以，比方说，声明字段和维持状态值**。实际上，你可以用Trait定义做任何用类定义做的事，并且语法也是一样的，除了两点。***第一点，Trait不能有任何“类”参数***，也就是说，传递给类的主构造器的参数。换句话说，尽管你可以定义如下的类：

class Point(x: Int, y: Int)

但下面的Trait定义直接报错：

scala> trait NoPoint(x:Int,y:Int)

<console>:1: error: traits or objects may not have parameters

trait NoPoint(x:Int,y:Int)

类和特质的另一个差别在于无论在类的哪个角落，super调用的都是静态绑定的，而在特质中，它们是动态绑定的。如果你在类中写下”super.toString”，你很明确哪个方法实现将被调用。**然而如果你在特质中写了同样的东西，在你定义特质的时候super调用的方法实现尚未被定义。调用的实现将在每一次特质被混入到具体类的时候才被决定。？？？**

## 选择瘦接口还是胖接口设计？

Trait的一种主要应用方式是可以根据类已有的方法自动为类添加方法。也就是说，Trait可以使得一个瘦接口变得丰满些，把它变成胖接口。

选择瘦接口还是胖接口的体现了面向对象设计中常会面临的在实现者与接口用户之间的权衡。胖接口有更多的方法，对于调用者来说更便捷。客户可以捡一个完全符合他们功能需要的方法。另一方面瘦接口有较少的方法，对于实现者来说更简单。然而调用瘦接口的客户因此要写更多的代码。由于没有更多可选的方法调用，他们或许不得不选一个不太完美匹配他们所需的方法并为了使用它写一些额外的代码。

Java的接口常常是过瘦而非过胖。例如，从Java 1.4开始引入的CharSequence接口，是对于字串类型的类来说通用的瘦接口，它持有一个字符序列。下面是把它看作Scala中Trait的定义：

trait CharSequence {

def charAt(index: Int): Char

def length: Int

def subSequence(start: Int, end: Int): CharSequence

def toString(): String

}

尽管类String成打的方法中的大多数都可以用在任何CharSequence上，Java的CharSequence接口定义仅提供了4个方法。如果CharSequence代以包含全部String接口，那它将为CharSequence的实现者压上沉重的负担。任何实现Java里的CharSequence接口的程序员将不得不定义一大堆方法。因为Scala的Trait可以包含具体方法，这使得创建胖接口大为便捷。

在Trait中添加具体方法使得胖瘦对阵的权衡大大倾向于胖接口。不像在Java里那样，在Scala中添加具体方法是一次性的劳动。你只要在Trait中实现方法一次，而不再需要在每个混入Trait的方法中重新实现它。因此，与没有Trait的语言相比，Scala里的胖接口没什么工作要做。

要使用Trait加强接口，只要简单地定义一个具有少量抽象方法的Trait——Trait接口的瘦部分——和潜在的大量具体方法，所有的都实现在抽象方法之上。然后你就可以把丰满了的Trait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

## Trait示例-Rectangular对象

在设计绘图程序库时常常需要定义一些具有矩形形状的类型：比如窗口，bitmap图像，矩形选取框等。为了方便使用这些矩形对象，函数库对象类提供了查询对象宽度和长度的方法（比如width,height)和坐标的left,right,top和bottom等方法。然而在实现这些函数库的这样方法，如果使用Java来实现，需要重复大量代码，工作量比较大（这些类之间不一定可以定义继承关系）。但如果使用Scala来实现这个图形库，那么可以使用Trait，为这些类方便的添加和矩形相关的方法。

首先我们先看看如果使用不使用Trait如何来实现这些类，首先我们定义一些基本的几何图形类如Point和Rectangle：

class Point(val x:Int, val y:Int)

class Rectangle(val topLeft:Point, val bottomRight:Point){

def left =topLeft.x

def right =bottomRight.x

def width=right-left

// and many more geometric methods

}

这里我们定义一个点和矩形类，Rectangle类的主构造函数使用左上角和右下角坐标，然后定义了 left,right,和width一些常用的矩形相关的方法。

同时，函数库我们可能还定义了一下UI组件（它并不是使用Retangle作为基类），其可能的定义如下：

abstract class Component {

def topLeft :Point

def bottomRight:Point

def left =topLeft.x

def right =bottomRight.x

def width=right-left

// and many more geometric methods

}

可以看到left,right,width定义和Rectangle的定义重复。可能函数库还会定义其它一些类，也可能重复这些定义。

如果我们使用Trait，就可以消除这些重复代码，比如我们可以定义如下的Rectangular Trait类型：

trait Rectangular {

def topLeft:Point

def bottomRight:Point

def left =topLeft.x

def right =bottomRight.x

def width=right-left

// and many more geometric methods

}

然后我们修改Component 类定义使其“融入”Rectangular 特性：

abstract class Component extends Rectangular{

//other methods

}

同样我们也修改Rectangle定义：

class Rectangle(val topLeft:Point, val bottomRight:Point) extends Rectangular{

// other methods

}

这样我们就把矩形相关的一些属性和方法抽象出来，定义在Trait中，凡是“混合”了这个Rectangluar特性的类自动包含了这些方法：

object TestConsole extends App{

val rect=new Rectangle(new Point(1,1),new Point(10,10))

println (rect.left)

println(rect.right)

println(rect.width)

}

## Ordered Trait

比较对象也是胖接口来的比较方便的一个应用领域，当你需要比较两个有顺序关系的对象时，如果只需要一个方法就可以知道需要比较的结果就非常便利。比如，你需要“小于”关系，你希望使用“< “比较就可以了，如果是“小于等于”，使用”<=”就可以。如果使用瘦接口来定义类，也许你只定义了一个<比较方法，那么如果需要小于等于，你可能需要使用(x<y)|| (x==y)。一个胖接口定义了所有可能的比较运算符，使得你可以直接使用<=来书写代码。

但胖接口带来的便利也是有代价的,回头看看我们前面定义的Rational类;

如果我们需要定义比较操作，则需要定义如下代码：

class Rational (n:Int, d:Int) {

...

def < (that:Rational) = this.numer \* that.denom > that.numer \* this.denom

def > (that:Rational) = that < this

def <=(that:Rational) = (this < that) || (this == that) def >=(that:Rational) = (this > that) || (this == that)

}

这个类定义了四个比较运算符 < ,>,< =和>=。首先我们注意到后面三个比较运算符，都是通过第一个比较运算符来实现的。其次，我们也可以看到，后面三个比较操作对于任意对象都是适用的，和对象的类型无关。而需要实现这四个比较运算的胖接口都要重复这些代码。

Scala对于比较这种常见的操作，提供了Ordered Trait定义，使用它，你可以把所有的比较运算的代码使用一个compare定义来替代。这个ordered trait可以让需要实现比较运算的类，通过和Ordered特质“融合”，而只需实现一个compare方法即可。

因此我们可以修改前面的实现如下：

class Rational (n:Int, d:Int) extends Ordered[Rational]{

...

override def compare (that:Rational)=

(this.numer\*that.denom)-(that.numer\*that.denom)

}

要注意两点，一是Ordered 需要指明**类型参数** Ordered[T] ，类型参数我们将在后面介绍，这里只需要知道添加所需比较类型的类名称，本例为Rational,此外，需要使用compare方法。 它比较有序对象，=0，表示两个对象相同，>0表示前面大于后面对象，<0表示前面小于后面对象。

下面为测试结果：

scala> class Rational (n:Int, d:Int) extends Ordered[Rational]{

| require(d!=0)

| private val g =gcd (n.abs,d.abs)

| val numer =n/g

| val denom =d/g

| override def toString = numer + "/" +denom

| def +(that:Rational) =

| new Rational(

| numer \* that.denom + that.numer\* denom,

| denom \* that.denom

| )

| def \* (that:Rational) =

| new Rational( numer \* that.numer, denom \* that.denom)

| def this(n:Int) = this(n,1)

| private def gcd(a:Int,b:Int):Int =

| if(b==0) a else gcd(b, a % b)

|

| override def compare (that:Rational)=

| (this.numer\*that.denom)-(that.numer\*that.denom)

|

| }

defined class Rational

scala> val half =new Rational(1,2)

half: Rational = 1/2

scala> val third=new Rational(1,3)

third: Rational = 1/3

scala> half < third res0: Boolean = false

scala> half >= third

res1: Boolean = true

因此，你在需要实现比较对象时，首先需要考虑Ordered Trait,看看这个Trait能否满足要求，然后通过和这个Trait “混合”就可以很方便的实现对象之间的比较。

此外要注意Ordered Trait 没有定义equal 方法，因为如果需要定义equal方法，那么需要检查传入参数的类型，Ordered Trait无法实现，因此你如果需要==比较运算符，需要另外定义。

## Trait用来实现可叠加的修改操作

我们已经看到Trait的一个主要用法，将一个瘦接口变成胖接口，本篇我们介绍Trait的另外一个重要用法，为类添加一些可以叠加的修改操作。Trait能够修改类的方法，并且能够通过叠加这些操作（不同组合）修改类的方法。

我们来看这样一个例子，修改一个整数队列，这个队列有两个方法：put 为队列添加一个元素，get 从队列读取一个元素。队列是先进先出，因此get读取的顺序和put的顺序是一致的。

对于上面的队列，我们定义如下三个Trait类型：

Doubling : 队列中所有元素X2

Incrementing： 队列中所有元素递增

Filtering： 过滤到队列中所有负数。

这三个Trait代表了修改操作，因为它们可以用来修改队列类对象，而不是为队列类定义所有可能的操作。这三个操作是可以叠加的，也就是说，你可以通过这三个基本操作的任意不同组合和原始的队列类“混合”，从而可以得到你所需要的新的队列类的修改操作。

为了实现这个整数队列,我们可以定义这个整数队列的一个基本实现如下：

import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {

def get():Int

def put(x:Int)

}

class BasicIntQueue extends IntQueue{

private val buf =new ArrayBuffer[Int]

def get()= buf.remove(0)

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

}

这个实现完成了对象的基本操作，看起来了还可以，但如果此时有新的需求，希望在添加元素时，添加元素的双倍，并且过滤掉负数，你可以直接修改put方法 来完成，但之后需求又变了，添加元素时，添加的为参数的递增值，你也可以修改put方法，这样显得队列的实现不够灵活。

我们来看看如果使用Trait会有什么结果，我们实现Doubling, Incrementing, Filtering 如下：

trait Doubling extends IntQueue{

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

}

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)

}

}

我们可以看到所有的Trait实现都以IntQueue为基类，这保证这些Trait只能和同样继承了IntQueue的类“混合”，比如和BasicIntQueue混合，而不可以和比如前面定义的Rational类混合。

此外Trait的put方法中使用了super,通常情况下对于普通的类这种调用是不合法的，但对于trait来说，这种方法是可行的，这是因为trait中的super调用是动态绑定的，只要和这个Trait混合在其他类或Trait之后，而这个其它类或Trait定义了super调用的方法即可。这种方法是实现可以叠加的修改操作是必须的，并且注意使用abstract override修饰符，这种使用方法仅限于Trait而不能用作Class的定义上。

有了这三个Trait的定义，我们可以非常灵活的组合这些Trait来修改BasicIntQueue的操作。

首先我们使用Doubling Trait

scala> val queue = new BasicIntQueue with Doubling

queue: BasicIntQueue with Doubling = $anon$1@3b004676

scala> queue.put(10)

scala> queue.get()

res1: Int = 20

这里通过BasicIntQueue和 Doubling混合，我们构成了一个新的队列类型，每次添加的都是参数的倍增。

我们在使用BasicIntQueue同时和Doubling和Increment 混合，注意我们构造两个不同的整数队列，不同时Doubling和Increment的混合的顺序

scala> val queue1 = new BasicIntQueue with Doubling with Incrementing

queue1: BasicIntQueue with Doubling with Incrementing = $anon$1@35849932

scala> val queue2 = new BasicIntQueue with Incrementing with Doubling

queue2: BasicIntQueue with Incrementing with Doubling = $anon$1@4a4cdea2

scala> queue1.put(10)

scala> queue1.get()

res4: Int = 22

scala> queue2.put(10)

scala> queue2.get()

res6: Int = 21

可以看到结果和Trait混合的顺序有关，简单的说，越后混合的Trait作用越大。因此queue1先+1，然后X2，而queue先X2后+1.

最后我们看看三个Trait混合的一个例子：

scala> val queue = new BasicIntQueue with Doubling with Incrementing with Filtering

queue: BasicIntQueue with Doubling with Incrementing with Filtering = $anon$1@73a4eb2d

scala> queue.put(10)

scala> queue.put(-4)

scala> queue.put(20)

scala> queue.get()

res10: Int = 22

scala> queue.get()

res11: Int = 42

scala> queue.get()

java.lang.IndexOutOfBoundsException: 0

at scala.collection.mutable.ResizableArray$class.apply(ResizableArray.scala:44)

at scala.collection.mutable.ArrayBuffer.apply(ArrayBuffer.scala:44)

at scala.collection.mutable.ArrayBuffer.remove(ArrayBuffer.scala:163)

at BasicIntQueue.get(<console>:11)

at .<init>(<console>:15)

at .<clinit>(<console>)

at .<init>(<console>:11)

at .<clinit>(<console>)

at $print(<console>)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:606)

at scala.tools.nsc.interpreter.IMain$ReadEvalPrint.call(IMain.scala:704)

at scala.tools.nsc.interpreter.IMain$Request$$anonfun$14.apply(IMain.scala:920)

at scala.tools.nsc.interpreter.Line$$anonfun$1.apply$mcV$sp(Line.scala:43)

at scala.tools.nsc.io.package$$anon$2.run(package.scala:25)

at java.lang.Thread.run(Thread.java:744)

最后的异常时因为队列为空（过滤掉了负数），我们没有添加错误处理，元素-4没有被添加到了队列中。

由此可以看出，通过Trait可以提高类的实现的灵活性，你可以通过这些Trait的不同组合定义了多种不同的对列类型。

## 为什么不是多重继承

特质是一种继承多个类似于类的结构的方式，但是它与许多语言中的多重继承有很重要的区别。其中一个最为重要：super的解释。对于多重继承来说，super调用导致的方法调用可以在调用发生的地方明确决定。而对于特质来说，方法调用是由类和被混入到类的特质的线性化所决定的。

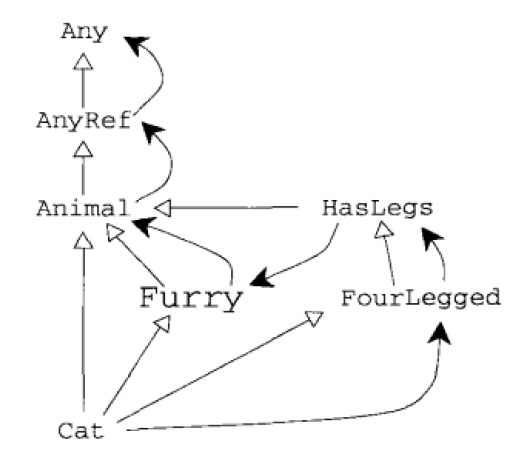
当你使用new实例化一个类的时候，scala把这个类和所有它继承的类还有其它的特质以线性的次序放在一起。然后，当你在其中的一个类中调用super，被调用的方法就是方法链的下一节。除了最后一个调用super之外的方法，其净结果就是可堆叠的行为。

线性化的精确次序由语言的规格说明书描述。规格的宗旨是：在任何的线性化中，某个类总是被线性化在其他所有超类和混入特质之前。因此，当你写了一个调用super的方法时，这个方法必将改变超类和混入特质的行为，没有其他路可走。

Scala的线性化的主要属性可以用下面的例子演示：假设你有一个类Cat，继承自超类Animal以及两个特质Furry和FourLegged。FourLegged又扩展了另一个特质HasLegs：

class Animal  
trait Furry extends Animal  
trait HasLegs extends Animal  
trait FourLegged extends HasLegs  
trait Cat extends Animal with Furry with FourLegged

Cat类的继承层级和线性化次序展示如下图：



继承次序使用传统的UML标注指明：白色三角箭头表明继承，箭头指向超类型。黑底非三角箭头说明线性化次序，箭头指向super调用解决的方向。

## 什么时候使用特质（Trait）

当你实现了一个可重用的行为集合时，是使用特质还是抽象类，有以下几条可供考虑的规则：

如果行为不会被重用，那么就把它做成具体类。具体类没有可重用的行为。

如果要在多个不相干的类中重用，就做成特质。只有特质可以混入到不同的类层级中。

如果你希望从Java代码中继承它，就使用抽象类。因为特质和它的代码没有近似的Java模拟，在java类里继承特质是很苯拙的。而继承scala的类和继承java的类完全一样。除了一个例外，只含有抽象成员的scala特质将直接翻译成java接口，因此即使你想使用java代码继承，也可以随心地定义这样的特质。

如果你计划以编译后的方式发布它，并且希望外部组织能够写一些继承自它的类，应该倾向于使用抽象类。原因是当特质获得或失去成员，所有继承自它的类就算没有改变也都要被重新编译。如果外边客户仅需要调用行为，而不是继承自它，那么使用特质没有问题。

如果效率非常重要，则应该倾向于使用类。大多数java运行时都能让类成员的虚方法调用快于接口方法调用。特质被编译成接口，因此会付出微小的性能代价。然而，仅当你知道哪个存疑的特质构成了性能瓶颈，并且有证据说明使用类代替确实能解决问题，才做这样的选择。

## 使用Package-将代码放入包中

软件开发过程减小程序之间的“耦合性”至关重要，降低耦合性的一个方法是模块化，Scala提供和Java类似的分包机制，但又稍有不同，因此即使你了解Java语言，还是建议您阅读本篇和后续几篇介绍Scala的Package和Import的文章。

我们之前的例子，没有明确使用package，因此它们存在于“未命名”的包中，或是缺省包中。

在Scala将代码定义到某个包中有两种方式：

第一种方法和Java一样，在文件的头定义包名，这种方法就后续所有代码都放在该报中。

比如：

package bobsrockets.navigation

class Navigator

第二种方法有些类似C#，如：

package bobsrockets.navigation {

class Navigator

}

第二种方法，可以在一个文件中定义多个包。

## 引用包中的代码

当我们把代码以层次关系放到包中时，它不仅仅可以帮助人们浏览代码，同时也说明了同一包中的代码具有某些相关性。Scala可以利用这些相关性来简化代码引用，比较使用短名称，而无需使用包的全路径来访问类定义。

下面我们给出三个简单的例子：

package bobsrockets{

package navigation{

class Navigator{

var map =new StarMap

}

class StarMap

}

class Ship {

val nav= new navigation.Navigator

}

class fleets{

class Fleet{

def addShip() {new Ship}

}

}

}

在第一个例子中，正如你可以预见的一样，访问同一包中定义的类型，无需使用前缀，直接使用类型的名称即可访问，也就是本例可以直接使用 new StarMap。类StarMap 和Navigator定义在同一个包中。

第二个例子，嵌套的package 也可以在其父包中被同级别的其它类型直接访问，而无需使用全称，因此第二个例子可以使用navigation来直接访问navigation包，而无需添加bobsrockets.

第三个例子，但使用包定义的{}语法结构时，内层的类型可以直接访问其外层定义的类型，因此在类Fleet中可以直接访问外层定义的类型Ship。

要注意的是这种用法只适用于你明确嵌套包定义，如果你采用Java语言风格-一个文件定义一个包，那么你只能访问该包中定义的类型。

访问包定义的类型还有一个技巧值得说明一下，比如你定义了一些类型之间可能存在相互隐藏的关系，也就是内层定义的同名类型可能会隐藏外层定义的同名类型，那么你怎么来访问外层定义的类型呢？请看下例：

package launch{

class Booster3

}

package bobsrockets{

package navigtion{

package launch{

class Booster1

}

class MissionControl{

val booster1 =new launch.Booster1

val booster2=new bobsrockets.launch.Booster2

val booster3=new \_root\_.launch.Booster3

}

}

package launch{

class Booster2

}

}

如何来访问Booster1，Booster2,和Booster3呢? 访问Booster1 比较容易，Booster2可以通过全称来访问。那么如何访问最外层的Booster3呢？内层的包launch隐藏了这个外部的同名包。为解决这种情况，***Scala提供了\_root\_,也就是所有最外层的类型都可以当成定义在\_root\_包中****。*因此\_root\_.launch.Booster3可以访问最外层定义的类型。

## 使用import

和Java一样，Scala也是通过import语句引用其它包中定义的类型，类型引入后，可以使用短名称来引用该类型而无需使用全路径。要注意的Scala使用“\_” 而非”\*”作为通配符。

//easy access to Fruit

import bobsdelights.Fruit

//easy access to all members of bobdelights

import bobsdelights.\_

//easy access to all member of Fruits

import bobsdelights.Fruits.\_

所定义的类型中包bobsdelights中：

package bobsdelights

abstract class Fruit(

val name: String,

val color:String

)

object Fruits{

object Apple extends Fruit ("apple","red")

object Orange extends Fruit("orange","orange")

object Pear extends Fruit("pear","yellowish")

val menu=List(Apple,Orange,Pear)

}

第一个为引用单个类型，第二个为按需引用，和Java不同的是，是使用“\_”代替“\*”,第三个类似于Java中的静态引用，可以直接使用Fruits中定义的对象。

此外Scala中的import语句的使用比较灵活，可以用在代码的任意部分，而不一定需要在文件开头定义。比如下面 import定义在函数内部：

import bobsdelights.Fruit

def showFruit(fruit:Fruit){

import fruit.\_ //对象当作模块被引用

println(name+"s are" + color)

}

方法showFruit 引入fruit对象（非类型）的所有成员，fruit的类型为Fruit，因此可以在函数直接使用fruit的成员变量，而无需使用fruit限定符。这个方法和下面代码是等价的：

import bobsdelights.Fruit

def showFruit(fruit:Fruit){

println(fruit.name+"s are" + fruit.color)

}

和Java相比，Scala的import的使用更加灵活：

（1）可以出现在文件中任何地方

（2）可以import对象（singleton或者普通对象）和package本身

（3）支持对引入的对象重命名或者隐藏某些类型

下面的例子直接引入包名称，而非包中成员，引入包后，可以使用相对名称来指代某个类型（有些类型文件系统的路径）

import java.util.regex

class AStarB {

val pat= regex.Pattern.compile("a\*b")

}

import 也可以用来重命名或者隐藏某些类型，比如：

import Fruits.{Apple,Orange}

仅仅引用Fruits中的Apple和Orangle类型。

下面的例子使用=>重命名类型：

import Fruits.{Apple=>MaIntosh,Orange}

同样重命名也可以重新定义包名称，比如：

import java.{sql => S}

将引入的包java.sql 该名为 java.S 因此可以使用 S.Date 来代替 sql.Date

如果需要隐藏某个类型，可以使用 Type => \_ ，将某个类型改名为\_就达到隐藏某个类型的效果，比如

import Fruits.{Apple=>\_,\_}

这个引用，引入Fruits中除Apple之外的其它类型。

引用选择器可以包括下列模式：

1. 简单名x，把x包含进引用名集。
2. 重命名子句x=>y。让名为x的成员以名称y出现。
3. 隐藏子句x=>\_。把x排除在引用名集之外。
4. 全包括’\_’。引用除了前面子句提到的之外的全体成员。如果存在全包括，那么必须是引用选择的最有一个。

## 隐含的import

Scala缺省为每个文件添加如下几个package. 这几个包无需明确指明。

import java.lang.\_ //everything in the java.lang package

import scala.\_ //everything in the scala package

import Predef.\_ //everything in the Predef object

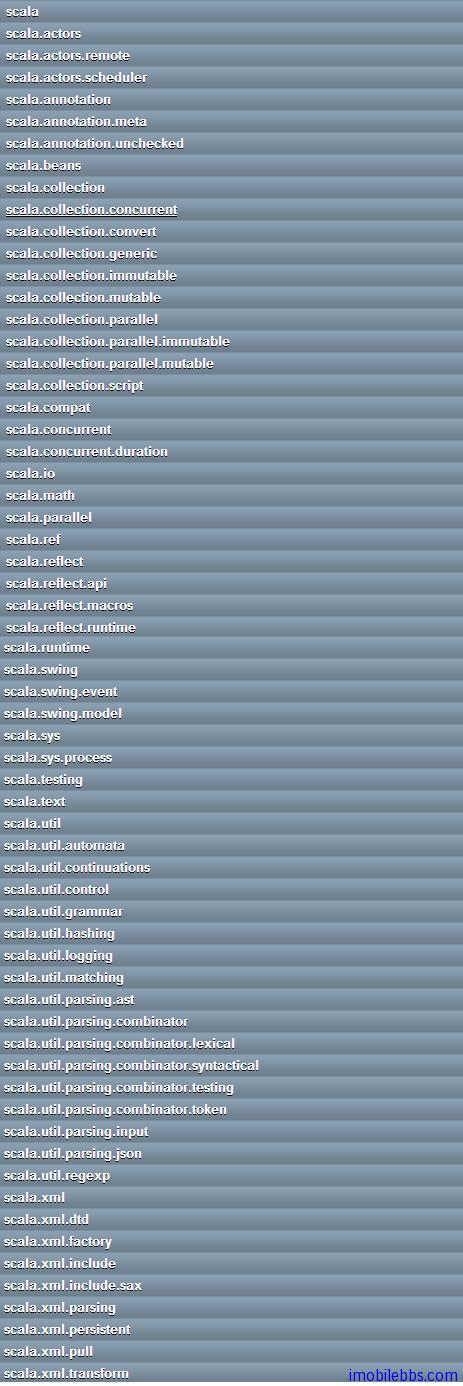
因此在写Scala应用之前，先了解下这些缺省包定义了那些类和功能。

此外这三个包的顺序也需要了解一下，比如StringBuilder类定义在包scala 和java.lang包中，后定义的import会覆盖前面的定义，因此如果不明确指明，

StringBuilder为scala.StringBuilder而非java.lang.StringBuilder.

注意这里的scala.\_ 指所有scala下的包，包括子包，也就是所有<http://www.scala-lang.org/files/archive/api/2.10.3/#package>

Predef为一对象（非包名），因此可以直接使用Predef对象定义的方法（静态引用）。因此在写代码之前了解Scala包和Predef定义的功能尤其重要.



## 访问控制修饰符

包的成员，类或对象可以使用访问控制修饰符，比如private和protected来修饰，通过这些修饰符可以控制其他部分对这些类，对象的访问。Scala和访问控制大体上和Java类似，但也有些重要的不同，本篇将介绍这些。

**（1）私有成员**

Scala的私有成员和Java类似，一个使用private修饰过的类或对象成员，只能在该类或对象中访问，在Scala中，也可以在嵌套的类或对象中使用。比如：

class Outer{

class Inner{

private def f(){

println("f")

}

class InnerMost{

f() //OK

}

}

(new Inner).f();// error: f is not accessible

}

在Scala中，(new Inner).f()是不合法的，因为它是在Inner中定义的私有类型，而在InnerMost中访问f却是合法的，这是因为InnerMost是包含在Inner的定义中（子嵌套类型）。 在Java语言中，两种访问都是可以的。Java允许外部类型访问其包含的嵌套类型的私有成员。

**（2）保护成员**

和私有成员类似，Scala的访问控制比Java来说也是稍显严格些。在Scala中，由Protected定义的成员只能由定义该成员和其派生类型访问。而在Java中，由Protected定义的成员可以由同一个包中的其它类型访问。在Scala中，可以通过其它方式来实现这种功能。

下面为protected的一个例子：

class p{

class Super{

protected def f() {

println("f")

}

}

class Sub extends Super{

f()

}

class Other{

(new Super).f() //error: f is not accessible

}

}

**（3）公开成员**

public访问控制为Scala定义的缺省方式，所有没有使用private和protected修饰的成员都是“公开的”，可以被自由访问。Scala不需要使用public来指定“公开访问”修饰符。

## 为访问修饰符添加作用域

Scala的访问修饰符可以添加作用域参数。作用域的语法如下：

private[x] 或 protected[x]

**其中x代表某个包，类或者对象**，表示“直到”x的私有或保护（表示可以访问这个Private的或protected的范围直到X）。

通过为访问修饰符添加作用域参数，可以非常精确的控制所定义的类型能够被其它类型访问的范围。尤其是可以支持Java语言支持的package private, package protected等效果。

C类里的protected[x]修饰符允许c的所有子类及修饰符所属的包、类或对象访问带有此标记的定义。

Private的限定词还能指向所属类或对象。

Scala还具有一种比private更严格的访问修饰符。被private[this]标记的定义仅能在包含了定义的同一个对象中被访问。这种定义被称为对象私有（object-private）。

把成员标记为private[this]可以保证它不能被同一个类中其他对象访问。这在把代码看作（程序员之间的契约）文档时比较有用。有时它也能让你写出更通用的变体注释。

下面的例子为这种用法的一个示例：

package bobsrockets

package navigation{

private[bobsrockets] class Navigator{

protected[navigation] def useStarChart(){}

class LegOfJourney{

private[Navigator] val distance=100

}

private[this] var speed = 200

}

}

package launch{

import navigation.\_

object Vehicle{

private[launch] val guide=new Navigator

}

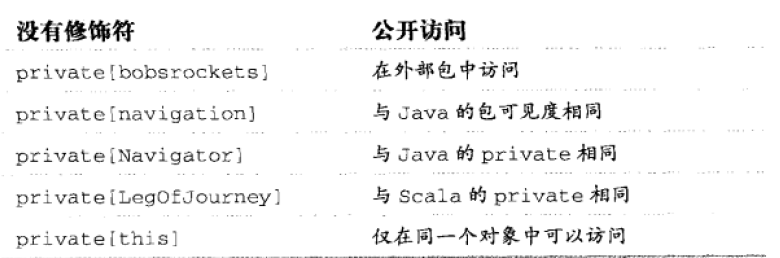
}

在这个例子中，类Navigator使用private[bobsrockets]来修饰，这表示这个类可以被bobsrockets中所有类型访问，比如通常情况下Vehicle无法访问私有类型Navigator,但使用包作用域之后，Vechile中可以访问Navigator.

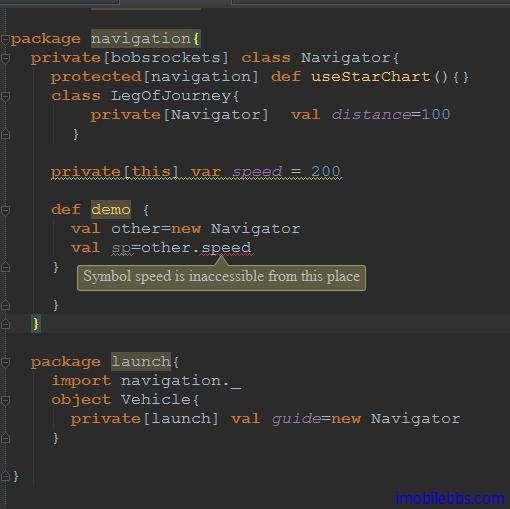
这种技巧在分散在多个Package的大型项目时非常有用，它允许你定义一些在多个子包中可以访问，但对使用这些API的外部客户代码隐藏，而这种效果在Java中是无法实现的。

此外，Scala还支持一种比private还要严格的访问控制，本例中的private[this]，只允许在定义该成员的类型中访问，它表示该成员不仅仅只能在定义该成员的类型中访问，而且只能是由该类型本身访问。

上述代码中，相应的私有限定符的效果如下：



本例中speed,使用 protected[this], speed,和this.speed只在定义该成员的实例中可以访问，下面的用法也是不合法的，即使它们也在Navigator里面。当由于是新创建的另外的实例，编译出错：



## 断言和单元测试

断言和单元测试是检查软件的行为是否符合需要的两种重要方式。

**（1）断言**

Scala里，断言被写为对预定义方法assert的调用。表达式assert(condation)将在condition条件不成立的时候抛出AssertionError。assert还有带两个参数的版本。表达式assert(condition,explanation)会测试condition，并且如果条件不成立，会抛出含有指定explanation作为说明的AssertionoError。Explanation的类型是Any，因此你可以把任何对象当作说明参数。assert方法会对传入的参数调用toString，以获得可以放在AssertionError中的字符串说明。

例如，在前面的类定义Element中的above方法，可以在调用widen之后放一个assert以确保变宽的元素具有相等的宽度。如：

def above(that:Element):Element={  
 val this1 = this widen that.width  
 val that1 = that widen this.width  
 assert(this1.width == that.width) //断言  
 elem( this.contents ++ that.contents)  
}

另一种或许会使用断言的方式识在widen方法的结束处，在返回结果值之前，检查一下宽度是否正确。你可以这么做，首先把结果存在val里，然后对这个结果进行断言，并且如果断言成功，则返回val值。不过，也可以使用Predef里的名为ensuring的方法来 简化这些操作，该用法如下：

def widen(w:Int):Element=  
 if(w < width) this  
 else {  
 val left = elem(' ',(w - width)/2,height)  
 var right = elem(' ',w - width - left.width,height)  
 left beside this beside right  
 }ensuring(w <= \_.width) //下划线为widen方法的返回结果Element

由于存在隐式转换，因此ensuring方法能被用在任何结果类型上。尽管这段代码看上去好像是对widen的结果（类型Element）调用ensuring，但实际上是对Element隐式转换成的类型调用了ensuring。Ensuring方法带一个函数作参数，该函数是接收一个结果类型对象并返回Boolean类型的论断函数（predicate function）。Ensuring会把结果传给这个函数。如果函数返回true，ensuring将返回结果，否则，ensuring将抛出AssertionError。

断言（以及ensuring检查）可以使用jvm的-ea和-da命令行标志开放和禁止。开放的时候，每个断言被当作对使用软件运行时产生的实际数据进行的小测试。

**（2）scala里的单元测试**

Java实现的工具：JUnit、TestNG

Scala实现的工具：ScalaTest、specs、ScalaCheck

## Case语句与偏函数

Scala通过case语句提供了形式简单、功能强大的模式匹配功能。但是也许你不知道，Scala还具有一个与case语句相关的语言特性，那就是：在Scala中，被“{}”包含的一系列case语句可以被看成是一个函数字面量，它可以被用在任何普通的函数字面量适用的地方，例如被当做参数传递。

scala> val defaultValue:Option[Int] => Int = {

case Some(x) => x

case None => 0 }

scala> defaultValue(Some(5))

res1: Int = 5

defaultValue是一个函数字面量，它的值是：

{

case Some(x) => x

case None => 0

}

看懂了以上的代码，我们就不难理解在Scala的Actor中经常使用的react函数的语法形式：

react {

case (name: String, actor: Actor) => {

actor ! getip(name)

act()

}

case msg => {

println("Unhandled message: "+ msg)

act()

}

}

react是一个函数，它接收一个函数字面量作为参数。至此我们都没有提到偏函数的概念。什么是偏函数？它与Case语句有什么关系？

**在Scala中，偏函数是具有类型PartialFunction[-A,+B]的一种函数。A是其接受的函数类型，B是其返回的结果类型。偏函数最大的特点就是它只接受和处理其参数定义域的一个子集，而对于这个子集之外的参数则抛出运行时异常**。这与Case语句的特性非常契合，因为我们在使用case语句是，常常是匹配一组具体的模式，最后用“\_”来代表剩余的模式。如果一一组case语句没有涵盖所有的情况，那么这组case语句就可以被看做是一个偏函数。

case语句作为偏函数字面量：

val second:PartialFunction[List[Int],Int] = {

case List(x::y::\_) => y

}

second函数的功能是返回一个List[Int]中的第二个值。case函数体只涵盖了当一个List的长度大于2的情况，而忽略Nil和长度为1的列表。

scala.MatchError: List(2)

at $anonfun$1.apply(<console>:9)

at $anonfun$1.apply(<console>:9)

at .<init>(<console>:11)

at .<clinit>(<console>)

at RequestResult$.<init>(<console>:9)

at RequestResult$.<clinit>(<console>)

at RequestResult$scala\_repl\_result(<console>)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

at java.lang.reflect.Method.invoke(Method.java:597)

at scala.tools.nsc.Interpreter$Request$$anonfun$loadAndRun$1$$anonfun$apply$18.apply(Interpreter.scala:981)

at scala.tools.nsc.Interpreter$Request$$anonfun$loadAndRun$1$$anonfun$apply$18.apply(Interpreter.scala:981)

at scala.util.contr...

scala> second(1::2::Nil)

res8: Int = 2

当我们试图传入一个不在偏函数的定义域范围内的参数时，抛出了异常。如果我们想在调用函数前先检查一个参数是否在定义域范围以避免抛出异常，那么可以使用偏函数的isDefinedAt方法。

scala> second.isDefinedAt(List(2,3,4))

res10: Boolean = true

实际上，scala编译器把函数字面量：

{

case List(x::y::\_) => y

}

编译成了如下的等价形式：

new PartialFunction[List[Int], Int] {

def apply(xs: List[Int]) = xs match {

case x :: y :: \_ => y

}

def isDefinedAt(xs: List[Int]) = xs match {

case x :: y :: \_ => true

case \_ => false

}

}

这种转换是一种编译期行为，我们必须把second显式的声明为PartialFunction类型，如果没有给second指定类型，那么scala编译器会把后面的一组case语句编译成Function1类型，即完整的函数。

**Tips:一组case语句要成为一个偏函数，那么它被赋予的变量必须被声明为PartionFunction[-A,+B]**

那么我们什么时候该使用偏函数？或者说偏函数给我们带来了什么好处？当我们确定我们的程序不会被传入不可处理的值时，我们就可以使用偏函数。这样万一程序被传入了不应该被传入的值，程序自动抛出异常，而不需要我们手工编写代码去抛出异常，减少了我们的代码量。

## 样本类和模式匹配

样本类（case class）和模式匹配（pattern matching），在编写规范的、无封装数据结构时用到的组件。对数型递归数据尤其有用。

样本类，是scala避免在对象上使用模式匹配时需要大量的固定写法而采用的方式。

**（1）简单例子**

abstract class Expr  
case class Var(name:String) extends Expr  
case class Number(num:Double) extends Expr  
case class UnOp(operator:String,arg:Expr) extends Expr  
case class BinOp(operator:String,left:Expr,right:Expr) extends Expr

带有case修饰符的类为样本类（case class）。这种修饰符会让scala编译器自动为你的类添加一些句法上的便捷设定。

1. 它会添加与类名一致的工厂方法，例如：

val v = Var(“x”) //相当于 val v = new Var(“x”)

尤其是把工厂方法嵌套在一起的时候，可以简化表达式代码：

val op = BinOp("+",Number(1.2),Var("x"))

1. 样本类参数列表中的所有参数隐式获得了val前缀，因此被参数被当作字段维护

scala>v.name

res0:String = x

scala> op.left

res1:Expr = Number(1.2)

1. 编译器为你的类添加了方法toString、hashCode和equals的“自然”实现。它们能够打印、哈希和比较由类及（递归地得到）其所有参数组成的整棵树。因为scala里的==直接转换成equals，也就意味着样本类的元素一直是在做结构化的比较。

scala> println(op)  
BinOp(+,Number(1.2),Var(x))

scala> op.right == Var("x")

res3；Boolean = true

**（2）模式匹配**

下面先举几个简单的规则：

UnOp("-",UnOp("-",e)) => e //双重负号  
BinOp("+",e,Number(0)) =>e //加0  
BinOp("\*",e,NUmber(1)) =>e //乘1

使用模式匹配，这些规则可以用几乎同样的形式作为scala简化函数的核心。

def simplifyTop(expr:Expr):Expr={  
 expr match {  
 case UnOp("-",UnOp("-",e)) => e //双重负号  
 case BinOp("+",e,Number(0)) => e //加0  
 case BinOp("\*",e,Number(1)) => e //乘1  
 case \_ => expr  
 }  
}

match对应于java里的switch，格式如下：

选择器 match { 备选项 }

一个模式匹配包含了一系列备选项（alternative），每个备选项都开始于case关键字，后面紧跟一个模式（pattern）和一个或多个表达式，它们将在模式匹配过程中被计算。符号=>将模式和表达式隔开。

**（3）match与switch比较**

Scala的match与java的switch之间的区别：

1. match是scala的表达式，它始终以值作为结果；
2. scala的备选项表达式永远不会“掉入”下一个case；
3. 如果没有模式匹配，MatchError异常会被抛出。要么所有的情况都要考虑到，要么添加一个默认情况什么都不做。

**（3）模式的种类**

**a. 通配模式**

通配模式（\_）匹配任意对象。

可以用作默认的“全匹配”(catch all)的备选项，如：

expr match{

case BinOp(op,left,right) => println(expr + "is a binary operation")

case \_ =>

}

还可以用来忽略对象中你不关心的部分：

expr match{

case BinOp(\_,\_,\_\_ => println(expr + "is a binary operation")

case \_ => println("It's something else")

}

**b. 常量模式**

常量模式仅匹配自身。任何字面量都可以用作常量。例如，5、true、“hello”。另外，任何的val或单例对象也可以被用作常量。例如，单例对象Nil用来匹配空列表模式。

def describe(x:Any)= x match {  
 case 5 => "five"  
 case true => "truth"  
 case "hello" => "hi"  
 case Nil => "the empty list"  
 case \_ => "something else"  
}

**c.变量模式**

变量模式类似于通配符，可以匹配任意对象。不过与通配符不同的地方在于，scala把变量绑定在匹配的对象上。可以使用这个变量操作对象。例如，以下模式对零特殊处理，对其他值默认处理：

expr match{

case 0 => "zero"

case somethingElse => "not zero: "+ somethingElse

}

常量模式可以有符号名。例如可以用E（2.71828）和PI（3.1415926..）

scala> import Math.{E,PI}

import Math.{E, PI}

scala> E match{

| case PI => "strange math? PI = " + PI

| case \_ => "OK"

| }

res0: String = OK

在这里，E不能匹配PI。Scala用简单的文字规则区分PI是Math中常量还是代表选择器自身值的变量。规则如下：用小写字母开始的简单名被当作是模式变量，所有的其他引用被认为是常量。例如：

scala> val pi=Math.PI

pi: Double = 3.141592653589793

scala> E match{

| case pi => "strange math? PI = " + pi

| }

res1: String = strange math? PI = 2.718281828459045

在这里，pi是变量模式，可以匹配任意输入。因此后面的情况没有可以访问到的，如果添加case \_=> “OK”，就会报错：

scala> E match{

| case pi => "strange math? PI = "+ pi

| case \_ => "OK"

| }

<console>:11: warning: patterns after a variable pattern cannot match (SLS 8.1.1

)

case pi => "strange math? PI = "+ pi

^

<console>:12: warning: unreachable code due to variable pattern 'pi' on line 11

case \_ => "OK"

^

res2: String = strange math? PI = 2.718281828459045

scala提供了两种方法，用于给模式常量小写字母命名：

首先，如果常量是某个对象的字段，可以在其之上使用限定符前缀。例如，pi是变量模式，this.pi或obj.pi虽然都开始于小写字母但都是常量。如果不起作用（例如pi是本地变量），可以使用**反引号**包住变量名，例如`pi`会再次被解释为常量，而不是变量：

scala> E match{

| case `pi` => "strange math? PI = "+ pi

| case \_ => "OK"

| }

res3: String = OK

scala里反引号的用法：

1. 可以解决小写字母标识符当作模式匹配常量的问题；
2. 可以用来处理关键字当作普通的标识符的问题，例如Thread.`yield`()，这里的yield被当作标识符。

**d.构造器模式**

构造器（模式）的存在使得模式匹配真正变得强大。构造器模式看上去就像“BinOp(“+”,e,Number(0))”。它由名称（BinOp）及若干括号之间的模式：”+”、e和Number(0)构成。假如这个名称指定了一个样本类，那么这个模式就是表示首先检查对象是该名称的样本类的成员，然后检查对象的构造器参数是符合额外提供的模式的。

这些额外的模式意味着scala模式支持深度匹配（deep match）。这种模式不只检查顶层对象是否一致，还会检查对象的内容是否匹配内层的模式。由于额外的模式自身可以形成构造器模式，因此可以使用它们检查到对象内部的任意深度。例如：下面的模式检查了对象的顶层是BinOp，及它的第三个构造器参数是Number，以及它 的值为数字0。这个模式仅有一行但却能检查三层深度。

expr match{

case BinOp(“+”,e,Number(0)) => println(“a deep match”)

case \_>

}

**e.序列模式**

scala里面，可以像匹配样本类那样匹配如List或Array这样的序列类型。不过同样的语法现在可以指定模式内任意数量的元素。例如，下面代码展示了检查开始于零的三元素列表的模式。

expr match{

case List(0,\_,\_) => println("found it")

case \_ =>

}

如果要匹配一个不指定长度的序列，可以指定\_\*作为模式的最后元素。这种模式能匹配序列中0到任意数量的元素。例如：

expr match{

case List(0,\_\*) => println(“found it”)

case \_ =>

}

**f.元组模式**

你还可以匹配元组。类似（a,b,c）这样的模式可以匹配任意三元组。例如：

def tupleDemo(expr:Any)=  
 expr match{  
 case (a,b,c) => println("matched "+ a + b + c)  
 case \_ =>  
 }

**g.类型模式**

可以把类型模式（typed pattern）当做类型测试和类型转换的简易代码。例如：

def generalSize(x:Any)= x match {  
 case s:String => s.length  
 case m:Map[\_,\_] => m.size  
 case \_ => 1  
}

能够得到与类型模式匹配相同效果但更为曲折的方式需要使用类型测试及类型转换。

Scala中，测试表达式expr是否为String类型的，是如下格式：

expr.isInstanceOf[String]

要转换同样的表达式类型为String，要写成如下格式：

expr.asInstanceOf[String]

**类型擦除**

Scala使用了泛型的擦除（erasure）模式，就如java那样。也就是类型参数信息没有保留到运行期。因此，运行期没有办法判断给定的Map对象创建时带了两个Int参数还是其他的什么类型。系统所能做的只是判断这个值是某种任意类型参数的Map。例如：

def isIntIntMap(x:Any) = x match {  
 case m:Map[Int,Int] => true  
 case \_ => false  
}

对上述函数isIntIntMap进行调用结果：

scala> isIntIntMap(Map(1 -> 1))

res1: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))

res2: Boolean = true

擦除规则唯一例外就是数组，因为在scala和java里面，它们都被特殊处理了。数组的元素类型和数组值保存在一起，因此数组可以做模式匹配。例如：

def isStringArray(x:Any) = x match {  
 case a:Array[String] => true  
 case \_ => false  
}

**变量绑定**

除了独立的变量模式之外，还可以对任何其他模式添加变量。只要简单地写上变量名、一个@符号，以及这个模式。这种写法创造了变量绑定模式，这种模式的意义在于它能像通常的那样做模式匹配，并且如果匹配成功，则把变量设置成匹配的对象，就像使用简单的变量模式那样。

举个例子，为了寻找一行中使用了两遍绝对值操作符的模式匹配，这样的表达式可以简化为仅使用一次绝对值操作：

expr match{

case UnOp("abs",e @ Unop("abs",\_)) => e

case \_ =>

}

上例中，有一个用e作为变量及UnOp(“abs”,\_)作为模式的变量绑定模式。如果整个模式匹配成功，那么符合UnOp(“abs”,\_)的部分就可以使用e指代。正如代码中写的，之后e就会保持原样被返回。

**（3）模式守卫**

有时候，语法的模式匹配还不够精确。例如，假如要去指定一个简化规则以通过乘二运算，也就是说，e\*2替代两个相同操作元的相加，如e+e。表示成Expr树的语言，就是表达式：

BinOp(“+”,Var(“x”),Var(“x”))

将通过此规则被转换为：

BinOp(“\*”,Var(“x”),Number(2))

你或许可以这样写：

def simplifyAdd(e:Expr) = e match {

case BinOp("+",x,x) => BinOp("\*",x,Number(2)) //会报x已经定义的错误

case \_ => e

}

Scala编译器会对以上写法会报错，因为***scala要求模式是线性的：模式变量仅允许在模式中出现一次***。不过，可以用模式守卫（pattern guard）重新制定这个匹配规则，如：

def simplifyAdd(e:Expr) = e match {

case BinOp("+",x,y) if x == y => BinOp("\*",x,Number(2))

case \_ => e

}

模式守卫接在模式之后，开始于if。守卫可以是任意的引用模式中变量的布尔表达式。如果存在模式守卫，那么只有在守卫返回true的时候匹配才成功。上面的例子只会匹配带有两个相同操作元的二元操作。

带守卫的模式的其他例子包括：

case n:Int if 0 < n => ... //仅匹配正整数

case s:String if s(0) == 'a' => ... //仅匹配以字母‘a'开始的字符串

**（4）模式重叠**

模式以代码编写的前后次序尝试。

下面的代码例子给出了需考虑样本次序的例子：

def simplifyAll(expr:Expr):Expr = expr match {  
 case UnOp("-",UnOp("-",e)) => simplifyAll(e) //"-"是自身的反转  
 case BinOp("+",e,Number(0)) => simplifyAll(e) //'0'对于’+‘来说不改变结果  
 case BinOp("\*",e,Number(1)) => simplifyAll(e) //'1'对于'+'来说不改变结果  
 case UnOp(op,e) => UnOp(op,simplifyAll(e))  
 case BinOp(op,l,r) => BinOp(op,simplifyAll(l),simplifyAll(r))  
 case - => expr  
 }  
}

以上代码的simplify版本将对表达式的任何地方应用简化规则，而不仅仅是如simplifyTop做的在最顶层。它发源于simplifyTop，不过多添加了两个样本处理通常的一元和二元表达式。

第四个样本包含了模式UnOp(op,e)；等内容，它匹配任何一元操作，操作符和操作元任选，它们相应地绑定为模式变量op和e。这个样本中的可选表达式对操作元e递归调用了simplifyAll方法并使用（可能是）简化了的操作元重建同样的一元操作。第五个样本对BinOp

是类似的；它是任意二元操作的“全匹配”，并且对两个操作元递归调用了简化方法。

这个例子中有一点很重要，就是全匹配的样本要跟在更具体的简化方法之后，如果写成其他次序，那么全匹配样本将比特定规则获得更高的优先级。在许多情况下，编译器将发出警告。

例如，下面的match表达式不会编译成功，因为第一个样本匹配任何能匹配第二个样本的东西：

def simplifyBad(expr:Expr):Expr = expr match {  
  
 case UnOp(op,e) => UnOp(op, simplifyBad(e))  
 case UnOp("-",UnOp("-",e)) => e  
}

**（5）封闭类**

一旦写好了模式匹配，你就需要确信已经考虑到了所有可能的情况。有些时候你可以通过在匹配的最后添加默认处理做到这点，不过这仅仅在的确有一个合理的默认行为的情况下有效。如果没有默认的情况该如何处理？怎样才能保证包括了所有的情况？

实际上，可以让scala编译器帮助你检测match表达式中遗漏的模式组合。要做到这点，编译器就需要搞清楚哪些情况是可能。通常，这在scala里是不可能的，因为新的样本类可以定义在任何时刻及任意编译单元中，比如，什么都无法阻止你在不同于Expr其他四个已定义样本的编译单元中，向Expr的类层级新加第五个样本类。

可选方案就是让样本类的超类被封闭（sealed）。封闭类除了类定义所在的文件之外不能添加任何新的子类。这对于模式匹配来说是非常有用的，因为这意味着你仅需要关心你已经知道的子类即可。这还意味着你可以获得更好的编译器帮助。如果你使用继承自封闭类的样本类做匹配，编译器将通过警告信息标志出缺失的模式组合。

因此，如果要写打算做模式匹配的类层级，应当考虑封闭它们。只要把关键字sealed放在最顶层类的前边即可。例如：

sealed abstract class Expr  
case class Var(name:String) extends Expr  
case class Number(num:Double) extends Expr  
case class UnOp(operator:String,arg:Expr) extends Expr  
case class BinOp(operator:String,left:Expr,right:Expr) extends Expr

现在若定义一个丢失了若干可能样本的模式匹配：

def describe(e: Expr) = e match {  
 case Number(\_) => "a number"  
 case Var(\_) => "a variable"  
 }

上述代码在编译时，编译器会发出警告：

Warning:(32, 30) match may not be exhaustive.

It would fail on the following inputs: BinOp(\_, \_, \_), UnOp(\_, \_)

def describe(e: Expr) = e match {

^

这是因为模式匹配没有考虑UnOp和BinOp两个类。

当然，为了消除此警告，可以为方法添加用作全匹配的第三个样本：

def describe(e: Expr) = e match {  
 case Number(\_) => "a number"  
 case Var(\_) => "a variable"

case \_ => Throw new RuntimeException  
 }

还有一种更轻量级的解决办法，是给匹配的选择器表达式添加@unchecked注解。如下：

def describle(e:Expr) = (e: @unchecked) match{

case Number(\_) => “a number”

case Var(\_) => “a variable”

}

这里@unchecked注解的作用是：抑制模式的穷举性检查。

**（6）option类型**

Scala为可选值定义了一个名为Option的标准类型。这种值有两种类型，可以是Some(x)的形式，其中x是实际值，或者也可以是None对象，代表缺失的值。

Scala集合类的某些标准操作会产生可选值。例如，scala的Map的get方法会在发现了指定键的情况下产生Some(value)，在没有找到指定键的时候产生None。

scala> val capitals =

| Map("France" -> "Pairs","Japan" -> "Tokyo")

capitals: scala.collection.immutable.Map[String,String] = Map(France -> Pairs, J

apan -> Tokyo)

scala> capitals get "France"

res0: Option[String] = Some(Pairs)

scala> capitals get "North Pole"

res1: Option[String] = None

分离可选值最通常的办法是通过模式匹配：

scala> def show(x:Option[String]) = x match{

case Some(s) => s

case None => "?"

}

show: (x: Option[String])String

scala> show(capitals get "Japan")

res3: String = Tokyo

scala> show(capitals get "France")

res4: String = Pairs

scala> show(capitals get "North Pole")

res6: String = ?

scala鼓励对Option的使用以说明值是可选的。这种处理可选值的方式有若干超越java的优点。首先，对于代码读者来说，Option[String]类型的变量是可选的String，这比String类型的变量或可能有时是null来说要更为明显。如果变量是Option[String]类型的，而你打算当做String使用，你的scala程序就不会编译通过。

**（7）模式无处不在**

**a. 模式在变量定义中**

在定义val或var的任何时候，都可以使用模式替代简单的标识符。例如，可以使用模式拆分元组并把其中的每个值赋给变量：

scala> val myTuple = (123,"abc")

myTuple: (Int, String) = (123,abc)

scala> val (num,str) = myTuple //一条赋值语句定义多个变量

num: Int = 123

str: String = abc

使用样本类时这种构造非常有用。如果你知道正在用的样本类的精确结构，那就可以使用模式解构它。例如：

scala> val exp = new BinOp("\*",Number(5),Number(1))

exp: BinOp = BinOp(\*,Number(5.0),Number(1.0))

scala> val BinOp(op,left,right) = exp

op: String = \*

left: Expr = Number(5.0)

right: Expr = Number(1.0)

**b.用作偏函数的样本序列**

花括号内的样本序列（备选项）可以用在能够出现函数字面量的任何地方。实质上，样本序列就是函数字面量，而且只有更普遍。函数字面量只有一个入口点和函数列表，当样本序列可以有多个入口点，每个都有自己的参数列表。每个样本都是函数的一个入口点，参数也被模式所特化，每个入口点的函数体都在样本的右侧。例如：

scala> val withDefault: Option[Int] => Int ={

| case Some(x) => x

| case None => 0

| }

withDefault: Option[Int] => Int = <function1>

scala> withDefault(Some(10))

res10: Int = 10

scala> withDefault(None)

res11: Int = 0

样本序列可以用作这样的偏函数，如果你把函数应用在它不支持的值上，将会产生一个运行期异常。例如，下面的偏函数能够返回整数列表的第二个元素：

scala> val second:List[Int] => Int ={

| case x :: y :: \_ => y

| }

<console>:7: warning: match may not be exhaustive.

It would fail on the following input: List(\_)

val second:List[Int] => Int ={

^

second: List[Int] => Int = <function1>

scala> second(List(5,6,7))

res12: Int = 6

以上代码在进行编译时，编译器会提示匹配不全面的警告。如果传递函数一个三元列表，它的执行没问题，如果传递一个空列表就会报错。

如果你想要检查是否一个偏函数有定义，必须首先告诉编译器你知道正在使用的是偏函数。类型List[Int] => Int包含了不管是否为偏函数的，从整数列表到整数的所有函数。仅包含从整数列表到整数的偏函数的，应该写成PartialFunction[List[Int],Int]。下面还是second函数，这次写成了使用偏函数类型：

scala> val second: PartialFunction[List[Int],Int] = {

| case x :: y :: \_ => y

| }

second: PartialFunction[List[Int],Int] = <function1>

偏函数有一个isDefinedAt方法，可以用来测试是否函数对某个特定值有定义。本例中，函数对任何有至少两个元素的列表有定义：

scala> second.isDefinedAt(List(1,2,3))

res1: Boolean = true

scala> second.isDefinedAt(List())

res2: Boolean = false

典型的偏函数例子是前面例子里的那种模式匹配函数字面量。实际上，scala编译器把这样的表达式转译成偏函数的时候，会对模式执行两次翻译：其中一次是真实函数的实现，另一次是测试函数是否（对特定参数）有定义的实现。例如，上面的函数字面量{ case x :: y :: - => y} 会被翻译成下列的偏函数值：

new PartialFunction[List[Int],Int] {

def apply(xs:List[Int]) = xs match {

case x :: y :: \_ => y

}

def isDefinedAt(xs:List[Int]) = xs match {

case x :: y :: \_ => true

case \_ => false

}

}

这种翻译只有在函数字面量的声明类型为PartialFunction的时候才起效。如果声明的类型只是Function1，或根本不存在，那么函数字面量就会代而转译为完整的函数。

通常，在可能的情况下你应该尝试使用完整函数，因为使用偏函数的话，编译器没办法帮你避免运行期故障的出现。不过有些时候偏函数的确很有用。你应该确信永远不会对它调用未处理的值。或者，你可能正在使用一个需要偏函数的架构。并且如果是这样的话，它会在调用函数之前始终检查isDefinedAt。

**for表达式里的模式**

for表达式也可以使用模式。例如：

scala> for((country,city) <- capitals)

| println("The capital of " + country + " is "+city)

The capital of France is Pairs

The capital of Japan is Tokyo

上述代码中，for表达式从capitals映射中获得所有的键/值对。每一对都匹配于模式(country,city)，并定义了两个变量country和city。

不过，有时候模式同样也可能无法匹配产生的值。例如：

scala> val results = List(Some("apple"),None,Some("orange"))

results: List[Option[String]] = List(Some(apple), None, Some(orange))

scala> for(Some(fruit) <- results) println(fruit) //None值被丢弃

apple

orange

上例中，产生出来的不能匹配于模式的值被丢弃。

**（8）例子**

P219

## **使用列表**

**（1）列表字面量**

例子：

scala> val fruit = List("apple","oranges","pears")

fruit: List[String] = List(apple, oranges, pears)

scala> val nums = List(1,2,3,4)

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

scala> val empty = List()

empty: List[Nothing] = List()

列表和数组非常相似，不过有两点重要的差别。首先，列表是不可变的。也就是说，不能通过赋值改变列表的元素。其次，列表具有递归结构，而数组是连续的。

**（2）List类型**

就像数组一样，列表是同质的：列表的所有元素都具有相同的类型。元素类型为T列表类型写成List[T]。例如：

scala> val fruit:List[String] = List("apple","oranges","pears")

fruit: List[String] = List(apple, oranges, pears)

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

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

scala> val empty:List[Nothing] = List()

empty: List[Nothing] = List()

scala里的列表类型是协变（convariant）的。这意味着对于每一对类型S和T来说，如果S是T的子类型，那么List[S]是List[T]的子类型。比如说，List[String]是List[Object]的子类型。这很自然，因为每个字符串列表同样也可被看作是对象列表。

注意：空列表的类型为List[Nothing]。Nothing是scala的类层级的底层类型。它是每个scala类型的子类。因为列表是协变的，所以对于任意类型T的List[T]来说，List[Nothing]都是其子类。

**（3）构造列表**

所有的列表都是由两个基础构造块Nil和::（读作cons）构造出来。Nil代表空列表。中缀操作符：：,表示列表从前端扩展。也就是说x::xs代表了第一个元素为x，后面跟着列表xs(的元素)的列表。之前的列表也可以如下定义：

scala> val fruit = "apple" :: ("oranges" :: ("pears" :: Nil))

fruit: List[String] = List(apple, oranges, pears)

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

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

scala> val empty = Nil

empty: scala.collection.immutable.Nil.type = List()

由于以冒号结尾，：：遵循右结合规则（右操作数是调用者），因此，A :: B :: C::等同于A :: (B :: C)，可以去掉前面定义里用到的括号。例如：

scala> val nums = 1 :: 2 :: 3 :: 4 :: Nil

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

**（4）列表的基本操作**

对于列表的所有操作都可以表达为以下三种形式：

head: 返回列表的第一个元素；

tail：返回除第一个之外所有元素组成的列表；

isEmpty：列表是否为空

在scala里面，插入排序算法大致如下：

def isort(xs:List[Int]):List[Int]=  
 if(xs.isEmpty) Nil  
 else insert(xs.head,isort(xs.tail))  
  
def insert(x:Int,xs:List[Int]):List[Int]=  
 if(xs.isEmpty || x <= xs.head) x :: xs  
 else xs.head :: insert(x,xs.tail)

**（5）列表模式**

列表可以使用模式匹配进行拆分，这时列表模式需逐一匹配要拆分的列表达式。既可以用List(…)的形式的模式对列表的所有元素做匹配，也可以使用::操作符和Nil常量组成的模式逐一拆分列表。比如：

scala> val List(a,b,c) = fruit

a: String = apple

b: String = oranges

c: String = pears

如果起先并不知道列表元素的数量，最好还是使用::做匹配。例如：

a :: b :: rest可以匹配长度至少为2的列表：

scala> val a :: b :: rest = fruit

a: String = apple

b: String = oranges

rest: List[String] = List(pears)

如果回顾之前的模式所有可能形式的话，会发现无论是List(…)还是::，看上去都不符合之前定义的模式类型。实际上，List(…)是由开发库定义的抽取器(extrator)模式的实例。“cons”模式x：：xs是中缀操作符模式的特例。如果被看做是表达式，那么中缀操作与方法调用等价。但对于模式来说规则有些不同：如果被当作模式，那么类似于p op q这样的中缀操作符等价于op(p,q)。也就是说，中缀操作符op被当作构造器模式。在这里，x::xs这样的cons模式被看作::(x,xs)。这么说，是否会有符合这种构造器模式的名为::的类呢？实际上也的确存在这个类。全称是 scala.::，它是可以创建非空列表的类。所以scala存在两个::，一个存在于scala包中，另一个是List类的方法。::方法的目的是实例化scala.::的对象。

采用模式匹配拆分列表是与采用基本方法head、tail和isEmpty做拆分一致的可选方案。比如，以下的代码同样实现了插入排序：

def isort(xs:List[Int]): List[Int] = xs match {  
 case List() => List()  
 case x :: xs1 => insert(x,isort(xs1))  
 }  
  
def insert(x:Int,xs:List[Int]):List[Int] = xs match {  
 case List() => List()  
 case y :: ys => if(x <= y) x :: xs  
 else y :: insert(x, x :: ys)  
}

通常情况下，采用模式匹配做拆分会比那些基本方法更为清晰。

**（6）List类的一阶方法**

一阶方法是指不以函数作入参的方法。

**链接列表**

操作方法： ::: （可以当作 append）

例如：

scala> List(1,2,3) ::: List(4,5,6)

res0: List[Int] = List(1, 2, 3, 4, 5, 6)

**分治原则**

链接操作符（:::）被实现为List类的方法。不过可以对列表使用模式匹配实来“手工”实现。类似于定义一个append方法：

def append[T](xs:List[T],ys:List[T]):List[T]

对递归数据结构（如列表）进行编程时的“分治”设计原则。列表的许多算法首先是用模式匹配把输入列表拆分为更简单的样本，这是原则里所说的“分”；然后根据每个样本构建结果。如果结果是非空列表，那么一块块部件将通过同样的递归遍历算法构建出来。这就是原则里所说的“治”。

把这个原则应用到append方法中来。首先要考虑需要匹配哪个列表，这里有两个选择，考虑到后续“治”的过程中需要把两个列表的所有元素组合成一个列表，由于列表是从后向前构造的，明智的做法是让ys保持完整而xs需要拆分并放到ys前面，这样就把注意力集中到xs的模式匹配上。最普通的模式匹配只是简单的把列表分为空的和非空的，因此可以获得一个简要的append方法：

def append[T](xs:List[T],ys:List[T]):List[T]=  
 xs match {  
 case List() => ys  
 case x :: xs1 => x :: append(xs1,ys)  
 }

第二个可选项的计算过程充分显示了分治原则的“治”的部分：首先需设想所要的输出应该有的形状；然后计算其中独立的部分，使用此处适用的算法做递归遍历；最终根据这些部分构建成输出。

**计算列表的长度：length方法**

length方法能够计算出列表的长度。例如：

scala> List(1,2,3).length

res0: Int = 3

相对于数组来说，列表的length方法是较费时的操作。为了找到尾部，需要遍历整个列表，因此其花费的时间与元素数量成正比。这也是判断列表是否为空时，应当采用xs.isEmpty方法，而不采用xs.length == 0 的理由。虽然两种测试的结果一致，但第二种更慢，尤其是是列表xs较长的时候。

**访问列表的尾部：init方法和last方法**

last: 返回（非空）列表的最后一个元素。

init：返回除最后一个元素之外余下的列表。

scala> val abcde = List('a','b','c','d','e')

abcde: List[Char] = List(a, b, c, d, e)

scala> abcde.last

res1: Char = e

scala> abcde.init

res2: List[Char] = List(a, b, c, d)

与head和tail方法一样的是，对空列表调用这些方法的时候，会抛出异常。

不一样的是，head和tail运行的时间都是常量，但init和last需要遍历整个列表以计算结果。因此所耗的时间与列表长度成正比。

**反转列表：reverse方法**

如果由于某种原因，算法需要频繁的访问列表的最后一个元素，可以先对列表反转过来再进行处理。例如：

scala> abcde.reverse

res4: List[Char] = List(e, d, c, b, a)

scala> abcde

res5: List[Char] = List(a, b, c, d, e)

要注意的是，reverse创建了新的列表而不是就地改变被操作列表。原有列表并没有改变。

反转操作可以用连接（:::)来实现，例如：

def rev[T](xs:List[T]):List[T] = xs match {  
 case List() => xs  
 case x :: xs1 => rev(xs1) ::: List(x)  
}

不过，这里的rev实现并非最佳。

**前缀和后缀：drop、take和splitAt**

drop和take泛化了tail和init，它们可以返回列表任意长度的后缀或前缀。表达式“xs take n”返回列表的前n个元素。如果n大于xs.length，则返回整个xs。表达式“xs drop n”返回列表除了前n个元素之外的所有元素。如果n大于xs.length，则返回空列表。

splitAt在指定位置拆分列表，并返回对偶（pair）列表。它的定义符合如下等式：

xs splitAt n 等价于 (xs take n, xs drop n)

例如：

scala> abcde take 2

res8: List[Char] = List(a, b)

scala> abcde drop 2

res9: List[Char] = List(c, d, e)

scala> abcde splitAt 2

res10: (List[Char], List[Char]) = (List(a, b),List(c, d, e))

**元素选择：apply方法和indices方法**

apply方法实现了随机元素的选择；不过与数组中的同名方法相比，它使用的并不广泛。

scala> abcde apply 2

res11: Char = c

与其他类型一样，当对象出现在应该是方法调用的函数位置上时，就会隐式的插入apply方法，上面的代码可以缩写为：

scala> abcde(2)

res12: Char = c

在列表中使用随机元素访问要比在数组中要少的多，原因之一在于xs(n)花费的时间与索引值n成正比。实际上apply简单的定义为drop和head的组合：

xs apply n 等价于 (xs drop n).head

indices方法返回指定列表的所有有效索引值组成的列表。

scala> abcde.indices

res13: scala.collection.immutable.Range = Range(0, 1, 2, 3, 4)

**啮合列表：zip**

zip操作可以把两个列表组合成一个对偶列表：

scala> abcde.indices zip abcde

res14: scala.collection.immutable.IndexedSeq[(Int, Char)] = Vector((0,a), (1,b),

(2,c), (3,d), (4,e))

如果两个列表的长度不一致，则任何不能匹配的元素将被丢弃：

scala> val zipped = abcde zip List(1,2,3)

zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))

常用到的情况是把列表元素和索引值啮合在一起，可以使用zipWithIndex方法会更为有效，它将列表中的每个元素和元素在列表中的位置组成一对：

scala> abcde.zipWithIndex

res15: List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4))

**显示列表：toString和mkString方法**

toString操作返回列表的标准字符串表达形式：

scala> abcde.toString

res16: String = List(a, b, c, d, e)

如果需要其他表达式，可以使用mkString方法，该方法的调用格式为：

xs mkString (pre,sep,post)

调用返回的结果格式为：

pre + xs(0) + sep + … + sep + xs(xs.length -1) + post

mkString有两个重载的变体：

第一个变体：xs mkString sep 等同于 xs mkString (“”,sep,””)

第二个变体：xs mkString 等同于 xs mkString “”

举例如下：

scala> abcde mkString ""

res18: String = abcde

scala> abcde.mkString

res20: String = abcde

mkString方法还有名为addString的变体，它可以把构建好的字符串添加到StringBuilder对象中，而不是作为结果返回;

scala> val buf = new StringBuilder

buf: StringBuilder =

scala> abcde addString (buf,"{",";","}")

res21: StringBuilder = {a;b;c;d;e}

**转换列表：elements、toArray、copyToArray**

列表向数组转换：List的toArray、copyToArray

数组向列表转换：Array的toList方法

例如：

scala> val arr = abcde.toArray

arr: Array[Char] = Array(a, b, c, d, e)

scala> arr.toList

res23: List[Char] = List(a, b, c, d, e)

还有一个方法copyToArray：

xs copyToArray(arr,start)

该方法将列表xs的元素逐一插入到数组arr中，插入位置（数组索引）从start开始。要确保数组有足够空间存放列表的全部元素。例如：

scala> val arr2 = new Array[Int](10)

arr2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> List(1,2,3) copyToArray (arr2,3)

scala> arr2

res25: Array[Int] = Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)

**归并排序**

归并排序的工作原理如下：首先如果列表长度为零或仅有一个元素，它就已经是排好序的了，因此可以不加改变的返回。长列表可以拆分为两个子列表，每个包含大概一半的原列表元素。每个子列表采用对排序函数的递归调用完成排序，然后再用归并操作把产生的两个排好序的列表结合在一起。

对于归并排序的通用实现来说，你可能希望对于待排序列表元素的类型不做限制，并同样希望对于用来做元素比较的函数也不做限制。可以通过把这两项作为传入参数已获得具有最大通用性的函数。代码如下：

def msort[T](less:(T,T) => Boolean)(xs:List[T]): List[T] = {  
 def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {  
 case (Nil, \_) => ys  
 case (\_, Nil) => xs  
 case (x :: xs1, y :: ys1) =>  
 if (less(x, y)) x :: merge(xs1, ys)  
 else y :: merge(xs, ys1)  
 }  
  
 val n = xs.length / 2  
 if (n == 0) xs  
 else {  
 val (ys, zs) = xs splitAt n  
 merge(msort(less)(ys), msort(less)(zs))  
 }  
}

该函数的用法如下：

msort((x:Int,y:Int) => x<y)(List(5,7,9,3))

List(3, 5, 7, 9)

以上函数使用了柯里化的概念，可以在此基础上定义新的函数：

val intSort = msort((x:Int,y:Int) => x<y)\_  
intSort(List(4,1,9,0,5,8,3,6,2,7))

**（7）List类的高阶方法**

**列表间的映射：map、flatMap和foreach**

map操作：xs map f 把函数f作用于列表的每个元素，组成新的列表返回。例如：

scala> List(1,2,3) map (\_ +1)

res0: List[Int] = List(2, 3, 4)

scala> val words = List("the","quick","brown","fox")

words: List[String] = List(the, quick, brown, fox)

scala> words.map(\_.length)

res1: List[Int] = List(3, 5, 5, 3)

scala> words.map(\_.toList.reverse.mkString)

res2: List[String] = List(eht, kciuq, nworb, xof)

flatmap操作：xs flatMap f 把函数作用于列表的每个元素，并把f操作后返回的所有结果进行合并，将合并的结果进行返回。例如：

scala> words map (\_.toList)

res3: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k), List(b, r, o,

w, n), List(f, o, x))

scala> words flatMap (\_.toList)

res4: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)

以下例子可以构造出1<= j <= i <=5的（i,j）对偶：

scala> List.range(1,5) flatMap( i => List.range(1,i) map (j => (i,j)))

res5: List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1), (4,2), (4,3))

用for表达式也可以实现以上方法：

for( i <- List.range(1,5); j <- List.range(1,i)) yield(i,j)

foreach是第三种与映射类似的操作，然而它与map、flatMap不同的是，foreach的右操作元是过程的（返回Unit）。foreach只是对每个列表元素都调用一遍过程，操作的结果仍然是Unit，不会产生结果列表。例如：

scala> var sum = 0

sum: Int = 0

scala> List(1,2,3,4,5) foreach (sum += \_)

scala> sum

res10: Int = 15

**列表过滤：filter、partition、find、takeWhile、dropWhile、span**

**filter**: xs filter p 产生xs中符合函数p(x)为true的所有元素x组成的列表。例如：

scala> List(1,2,3,4,5) filter (\_ %2 == 0)

res11: List[Int] = List(2, 4)

scala> words filter (\_.length == 3)

res12: List[String] = List(the, fox)

**partition**操作：返回列表对，其中一个列表包含论断为真的元素，另一个列表包含论断为假的元素。

xs partition p 等价于 (xs filter p , xs filter (!p(\_)))

例如：

scala> List(1,2,3,4,5) partition ( \_ %2 == 0)

res13: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))

find操作：返回第一个满足给定论断的元素，而不是符合论断的全部元素。有符合的返回Some(x)，否则返回None。

scala> List(1,2,3,4,5) find (\_ %2 == 0)

res14: Option[Int] = Some(2)

scala> List(1,2,3,4,5) find (\_ <= 0)

res15: Option[Int] = None

takeWhile操作：xs takeWhile p操作返回列表xs中最长的能够满足p的前缀。例如:

scala> List(1,2,3,-4,5) takeWhile (\_ > 0)

res16: List[Int] = List(1, 2, 3)

dropWhile操作：xs dropWhile p操作移除列表xs中最长的能够满足p的前缀。例如：

scala> List(1,2,3,-4,5) dropWhile (\_ > 0)

res17: List[Int] = List(-4, 5)

span操作: 把takeWhile和dropWhile组合成一个操作，返回一对列表。

xs span p 等价于 (xs takeWhile p, xs dropWhile p)

例如：

scala> List(1,2,3,-4,5) dropWhile (\_ > 0)

res17: List[Int] = List(-4, 5)

**列表的论断：forall 和 exists**

操作xs forall p 如果xs的所有元素都满足p，则返回true，否则返回false

操作xs exists p 如果xs只要有一个元素满足p，就返回true，否则返回false。

例如：

scala> def hasZeroRow(m:List[List[Int]])=

| m exists (row => row forall (\_ == 0))

hasZeroRow: (m: List[List[Int]])Boolean

**折叠列表：/: 和 :\**

其他常用的操作会对列表元素始终执行某种操作，例如：

sum(List(a,b,c)) 等价于 0+a+b+c

这可以认为折叠操作的一个具体实例：

scala> def sum(xs:List[Int]): Int = (0 /: xs) (\_+ \_)

sum: (xs: List[Int])Int

scala> sum(List(1,2,3))

res2: Int = 6

类似的：

product(List(a,b,c)) 等价于 1\*a\*b\*c

也是折叠操作的一个具体实例：

scala> def product(xs:List[Int]):Int = (1 /: xs) (\_ \* \_)

product: (xs: List[Int])Int

scala> product(List(1,2,3))

res3: Int = 6

左折叠（fold left）操作“(z /: xs) (op)”与三个对象有关：开始值z，列表xs，以及二元操作op。折叠的结果是op应用到前缀 值z和每个相邻元素上。如下：

(z /: List(a,b,c)) (op) 等价于 op(op(op(z,a) ,b),c)

举个左折叠的例子，比如用空格连接所有字符串列表中的单词：

scala> val words = List("the","quick","brown","fox")

words: List[String] = List(the, quick, brown, fox)

scala> (" " /: words) (\_ + " " + \_)

res4: String = " the quick brown fox"

可以看到，结果的前面多了一个空格，可以稍微改变一下：

scala> (words.head /: words.tail) (\_ + " " + \_)

res5: String = the quick brown fox

/: 操作符产生向左侧倾斜的操作树。与之类似，操作符 :\ 产生向右倾斜的操作数。如：

( List(a,b,c) :\ z) (op) 等价于 op(a,op( b,op(c,z)))

:\ 被称为 右折叠（fold right），与左折叠类似，操作单元的顺序有的不一致。

对于组合操作来说，左折叠和右折叠是等价的，但在效率上稍有差别。例如：

scala> def flattenLeft[T](xss:List[List[T]]) =

| (List[T]() /: xss) (\_ ::: \_)

flattenLeft: [T](xss: List[List[T]])List[T]

scala> def flattenRight[T](xss:List[List[T]]) =

| (xss :\ List[T]() ) ( \_ ::: \_)

flattenRight: [T](xss: List[List[T]])List[T]

由于列表的连接xs ::: ys耗时与第一个参数xs的长度成正比，因此右折叠形式完成的实现flattenRight比左折叠实现flattenLeft更有效率。原因在于flattenLeft(xss)需要复制第一个元素xss.head一共n-1次，这里n是指列表xss的长度。

**例子：使用折叠操作完成列表反转**

初步方案如下：

def reverseLeft[T](xs:List{t]) = (startvalue /: xs) (operation)

然后，只要只要填入正确的startvalue和operation，可以拿一些简单的例子来推导这一部分。要推导startvalue的正确值，可以从可能出现的最小列表List()开始，例如：

List()

等价于

reverseLeft(List())

等价于

(strartvalue /: List()) (operation)

等价于

startvalue

因此，startvalue必然是List()。

要推导第二个操作元，可以另选一个次小的列表作为样例。现在已经知道startvalue是List（），因此有推导计算如下：

List(x)

等价于

reverseLeft(List(x))

等价于

（List() /: List(x)）(operation)

等价于

operation(List(),x)

因此，operation(List(),x)等价于List(x) ，还可以写成 x :: List()。这暗示了可以用::操作符及交换操作元位置来替换operation。最终获得reverseLeft的实现：***？？？***

def reverseLeft[T] (xs:List[T]) =

（List[T]() /: xs) { (ys,y) => y :: ys }

**列表排序：sortWith**

对列表xs的操作，xs **sortWith** before，可以对列表的元素进行排序，其中“before”是比较元素的方法。表达式x before y在x应按照顺序处于y之前的时候要能够返回true。例如：

scala> List(1,-3,4,2,6) sortWith (\_ < \_)

res8: List[Int] = List(-3, 1, 2, 4, 6)

scala> words sortWith (\_.length > \_.length)

res9: List[String] = List(quick, brown, the, fox)

**（8）List对象的方法**

全局可访问对象：scala.List ，是List类的伴生对象

**通过元素创建列表：List.apply**

scala> List.apply(1,2,3)

res0: List[Int] = List(1, 2, 3)

**创建数值范围：List.range(from,until)**

返回从from到until-1的所有数值列表，不包括until。例如：

scala> List.range(1,5)

res1: List[Int] = List(1, 2, 3, 4)

还有一个带有step的方法变体：List.range(from,until,step)。例如：

scala> List.range(1,9,2)

res2: List[Int] = List(1, 3, 5, 7)

**创建统一的列表：List.make**

Make方法创建由相同元素的零份或多份拷贝组成的列表。包括两个参数：创建列表的长度，需要重复的元素。例如：

scala> List.make(5,"a")

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res3: List[String] = List(a, a, a, a, a)

**解除啮合列表：List.unzip**

把对偶列表拆分成两个列表。例如“

scala> val zipped = abcde.toList zip List(1,2,3,4)

zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3), (d,4))

scala> List.unzip(zipped)

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res5: (List[Char], List[Int]) = (List(a, b, c, d),List(1, 2, 3, 4))

**连接列表：List.flatten、List.concat**

flatten：以列表的列表作为参数，并把所有的元素列表连接在一起。例如：

scala> val xss = List(List('a','b'),List('c'),List('d','e'))

xss: List[List[Char]] = List(List(a, b), List(c), List(d, e))

scala> List.flatten(xss)

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res8: List[Char] = List(a, b, c, d, e)

concat；连接多个元素列表，将多个列表以重复参数的形式直接传递个方法。例如：

scala> List.concat(List('a','b'),List('c'))

res0: List[Char] = List(a, b, c)

scala> List.concat(List(),List('b'),List('c'))

res1: List[Char] = List(b, c)

**映射及测试配对列表**

map2：以两个列表作为参数输入，及两个参数的布尔型论断。例如：

scala> List.map2(List(10,20),List(3,4,5)) (\_ \* \_)

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res6: List[Int] = List(30, 80)

exist2和forall2方法同样类似于exist和forall，不过它们同样带两个列表及输入两个参数的布尔型论断。

scala> List.forall2(List("abc","de"),List(3,2)) (\_.length == \_)

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res10: Boolean = true

scala> List.exists2(List("abc","de"),List(3,2)) (\_.length != \_)

warning: there were 1 deprecation warning(s); re-run with -deprecation for detai

ls

res12: Boolean = false

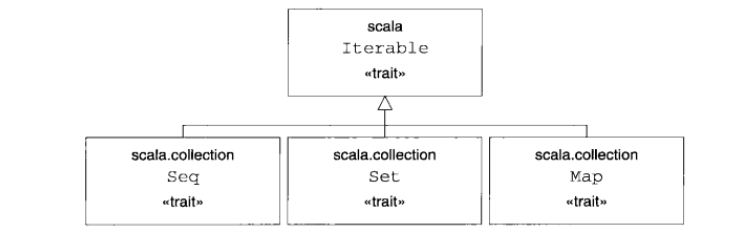
**（9）了解scala的类型推断算法**

Scala里的类型推断是基于流的。Scala采用局部的、基于流的类型推断方案。这种方式对于多态方法的类型推断存在局限性，可以通过添加类型标注进行弥补这种不足。

## 集合类型

**（1）集合库概览**

以下是scala集合类型的类层级：



Iterable是主要的特质，它是可变和不可变序列（seq）、集（set）、映射（map）的超特质。

集合对象可以通过elements方法产生Iterator（枚举器）。

def elements: Iterator[A]

Iterator有两个重要的方法：

def hasNext: Boolean

def next: A

Scala中所有的集合都扩展了Iterable特质。集合又可以分为三类：有先后次序的序列Seq；值的集合Set；键值对构成的对偶的映射Map。三类集合下分别还实现了特殊访问方式的特质：可以由下标来进行随机访问的IndexedSeq；可按照顺序访问的SortedSet；可按顺序访问的SortedMap。

**（2）序列**

序列是继承自特质seq的类，可以用来处理一组线性分布的数据。

**a.列表（List）**

列表（List）支持在列表的头部快速添加和删除元素，但不能提供对任意索引值的快速访问。

头元素的快速添加和删除可以更好的用于模式匹配，列表的不可变性有助于开发正确和高效的算法，因为无需创建列表的拷贝。

**b.数组**

数组支持基于从零开始的索引进行高效访问处于任意位置的元素。

**c.列表缓存**

List提供对列表头部，而非尾部的快速访问。如果需要通过向结尾添加对象的方式创建列表，可以考虑对表头前缀元素的方式反向构造列表，完成之后再调用reverse方法，获取需要的元素顺序。

另一种方式是使用ListBuffer，可以避免使用reverse方法。ListBuffer是可变对象，可以通过高效的添加元素方式创建列表。ListBuffer支持元素的添加和前缀操作。元素的添加使用 += 操作符，前缀使用 +: 操作符。完成之后调用ListBuffer的toList方法获取List。例如:

scala> import scala.collection.mutable.ListBuffer

import scala.collection.mutable.ListBuffer

scala> val buf = new ListBuffer[Int]

buf: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buf += 1

res0: buf.type = ListBuffer(1)

scala> buf += 2

res1: buf.type = ListBuffer(1, 2)

scala> buf.toList

res2: List[Int] = List(1, 2)

**d.数组缓存 ArrayBuffer**

ArrayBuffer与数组类型，额外还允许在序列的开始或结束的地方添加和删除元素。所有数组能使用的方法，数组缓存ArrayBuffer都能使用。例如：

scala> import scala.collection.mutable.ArrayBuffer

import scala.collection.mutable.ArrayBuffer

scala> val arrayBuf = new ArrayBuffer[Int]()

arrayBuf: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> arrayBuf += 1

res0: arrayBuf.type = ArrayBuffer(1)

scala> arrayBuf += 2

res1: arrayBuf.type = ArrayBuffer(1, 2)

scala> arrayBuf += 3

res2: arrayBuf.type = ArrayBuffer(1, 2, 3)

scala> arrayBuf.length

res3: Int = 3

scala> arrayBuf

res4: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

**e.队列 Queue**

队列的结构是先进先出。分为可变的和不可变的Queue。例如，创建不可变队列：

scala> import scala.collection.immutable.Queue

import scala.collection.immutable.Queue

scala> val empty = Queue[Int]()

empty: scala.collection.immutable.Queue[Int] = Queue()

scala> val has1 = empty.enqueue(1)

has1: scala.collection.immutable.Queue[Int] = Queue(1)

scala> val has123 = has1.enqueue(List(2,3)) //添加多个元素

has123: scala.collection.immutable.Queue[Int] = Queue(1, 2, 3)

scala> val (element,has23) = has123.dequeue //移除头部元素

element: Int = 1

has23: scala.collection.immutable.Queue[Int] = Queue(2, 3)

**f . 栈 Stack**

特点：后进先出的队列。分为可变的和不可变的。

元素的推入使用push，元素的取出用pop，只获取栈顶的元素而不移除可以使用top。例如：

scala> import scala.collection.mutable.Stack

import scala.collection.mutable.Stack

scala> val stack = new Stack[Int]()

stack: scala.collection.mutable.Stack[Int] = Stack()

scala> stack.push(1)

res0: stack.type = Stack(1)

scala> stack.push(2)

res1: stack.type = Stack(2, 1)

scala> stack.top

res2: Int = 2

scala> stack.pop

res3: Int = 2

scala> stack

res4: scala.collection.mutable.Stack[Int] = Stack(1)

**g. 字符串（经RichString隐式转换）**

RichString: Seq[Char] 因为Predef包含了从String到RichString的隐式转换，所有可以把任何字符串当成Seq[Char]。例如：

scala> def hasUpperCase(s:String) = s.exists(\_.isUpper)

hasUpperCase: (s: String)Boolean

scala> hasUpperCase("Robert Frost")

res5: Boolean = true

scala> hasUpperCase("e e cummings")

res6: Boolean = false

**（3）集（set）和映射（map）**

Scala中，提供了set和map的可变、不可变版本。默认情况下，获得的都是不可变版本。如果需要可变版本，需要先声明引用。

其中，默认获取不可变版本是在Predef中定义的，可以查看Predef的定义：

type Map[A, +B] = immutable.Map[A, B]  
type Set[A] = immutable.Set[A]  
val Map = immutable.Map  
val Set = immutable.Set

这里用Set定义的是不可变的，如果需要定义可变的，需要引用scala.collection.mutable，例如：

import scala.collection.mutable  
val mutaSet = mutable.Set(1,2,3)

**使用集：**

集的关键特性在于它可以使用对象的==操作检查，确保任何时候每个对象只在集中保留最多一个副本。例如，可以使用集对字符串不同单词计数。例如，首先利用正则表达式将字符进行拆分成单词：

scala> val text = "See Spot run. Run, Spot. Run!"

text: String = See Spot run. Run, Spot. Run!

scala> val wordsArray = text.split("[ !,.]+") //用空格及标点符号进行字符串拆分

wordsArray: Array[String] = Array(See, Spot, run, Run, Spot, Run)

要对不同的单词计数，可以先把它们变成同样的大小写然后加到集中。由于集不能重复添加，因此不同的单词将只在集中出现一次。例如：

scala> import scala.collection.mutable

import scala.collection.mutable

scala> val words = mutable.Set.empty[String]

words: scala.collection.mutable.Set[String] = Set()

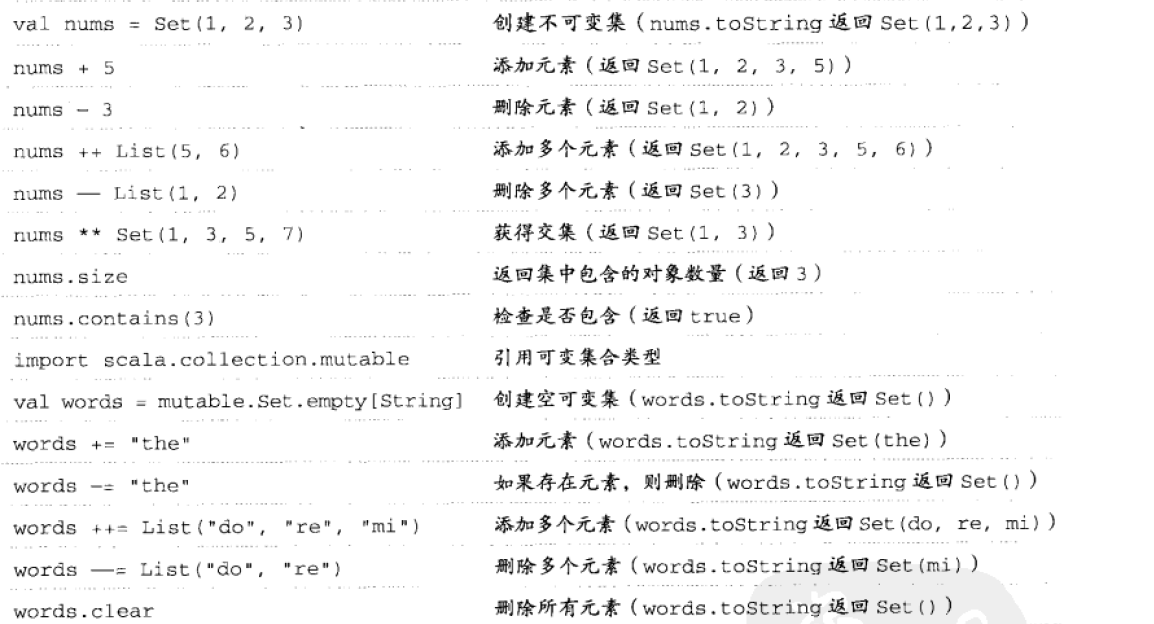
scala> for(word <- wordsArray)

| words += word.toLowerCase

scala> words

res9: scala.collection.mutable.Set[String] = Set(see, run, spot)

set的常用操作：



**使用映射Map**

映射可以创建键值对，映射的使用与数组接近，可以使用任何类型的键。映射也分可变和不可变两种类型。例如，创建可变的映射：

scala> val map = mutable.Map.empty[String,Int]

map: scala.collection.mutable.Map[String,Int] = Map()

scala> map("hello") = 1

scala> map("there") = 2

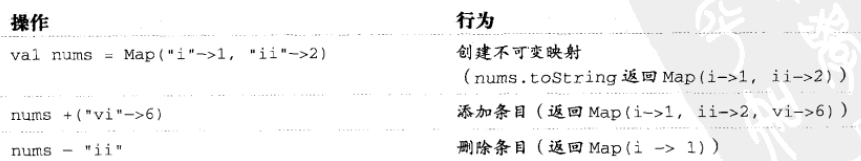
scala> map("hello")

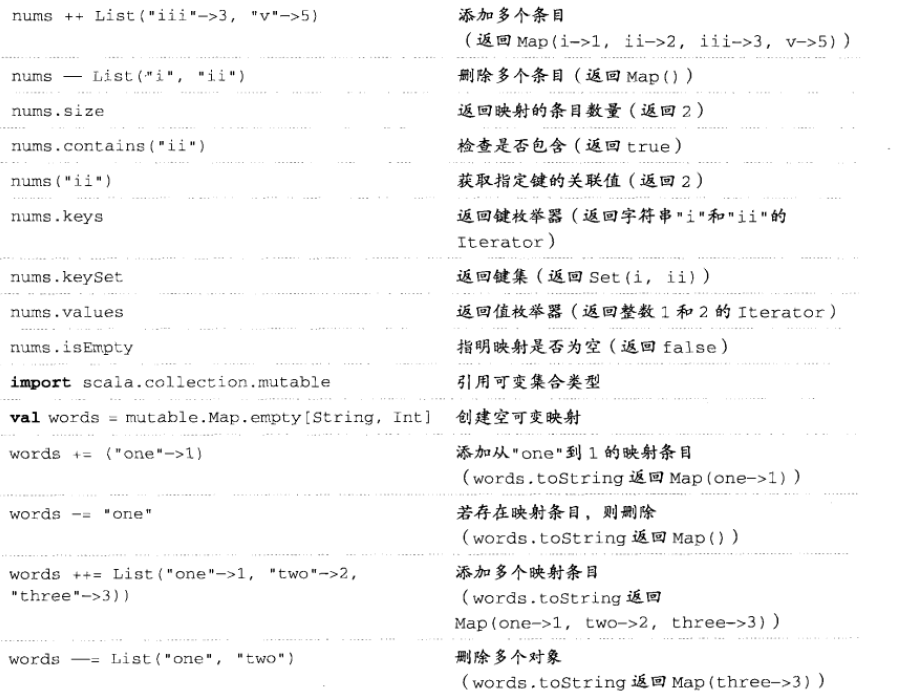
res15: Int = 1

利用Map，改造之前的单词统计函数：

def countWords(text:String) ={  
 val counts = mutable.Map.empty[String,Int]  
 for(rawWord <- text.split("[ ,!.]+")){  
 val word = rawWord.toLowerCase  
 val oldCount =  
 if(counts.contains(word)) counts(word) else 0  
 counts += (word -> (oldCount +1))  
 }  
 counts  
}  
  
countWords("See Spot run! Run, Spot, Run!")

映射Map的常用操作：





**默认的集和映射**

对于多数应用来说，由Set()、scala.collection.mutable.Map()等的工厂方法提供的可变及不可变的集和映射的实现已经能够满足要求。这些工厂方法提供的实现都使用了快速查找算法，通常都设计哈希表，因此它们都能快速反应对象是否存在于集合中。

不可变集和映射的情况更为复杂一些。例如，scala.collection.imutable.Set()工厂方法返回的类，取决于你传递给它多少元素。对于少于5个元素的集，类型完全取决于它的元素数量，已获得最优的性能。然而一旦请求的集合包含了5个元素以上，工厂方法返回的将是不可变的HashSet。

**有序的（Sorted）的集和映射**

有时候，需要集和映射的枚举器能够返回按特定顺序排列的元素。Scala提供了SortedSet和SortedMap特质。这两个特质分别由TreeSet和TreeMap实现，它们都使用了红黑色有序的保存元素或键。具体的顺序取决于Ordered特质，集的元素类型或映射的键值类型必须要么混入，要么能够隐式地转换成Ordered特质。

其中，TreeSet分为可变和不可变两种类型，TreeMap只有不可变类型的版本。

以下是TreeSet的例子：

scala> import scala.collection.immutable.TreeSet

import scala.collection.immutable.TreeSet

scala> val ts = TreeSet(9,3,1,8,0,2,7,4,6,5)

ts: scala.collection.immutable.TreeSet[Int] = TreeSet(0, 1, 2, 3, 4, 5, 6, 7, 8,

9)

scala> val cs = TreeSet('f','u','n')

cs: scala.collection.immutable.TreeSet[Char] = TreeSet(f, n, u)

下面是TreeMap的例子：

scala> import scala.collection.immutable.TreeMap

import scala.collection.immutable.TreeMap

scala> val tm = TreeMap(3 -> "three", 2 -> "two")

tm: scala.collection.immutable.TreeMap[Int,String] = Map(2 -> two, 3 -> three)

**同步的（Synchronized）集和映射**

如果需要线程安全的映射，可以把SynchronizedMap特质混入到你想要的特定类实现中。例如，把SynchronizedMap混入HashMap：

import scala.collection.mutable.{Map,SynchronizedMap,HashMap}  
  
object MapMaker {  
  
 def makeMap:Map[String,String] = {  
 new mutable.HashMap[String,String] with  
 SynchronizedMap[String,String]{  
 override def default (key:String) = //如果未查到键值对应的value，将返回default方法的返回值  
 "why do you want to know?"  
 }  
 }  
}

根据以上代码，scala编译器将产生混入了SynchronizedMap的HashMap合成类，并创建它的实例。这个合成子类还重载了default方法。如果根据键值查找value，该键的映射不存在的时候会返回Default方法的返回值。

由于makeMap方法返回的可变映射混入了SynchronizedMap特质，因此可以立即用于多线程环境。每次对映射的访问都被同步操作。下面演示单线程访问映射的情况：

scala> val captial = MapMaker.makeMap

captial: scala.collection.mutable.Map[String,String] = Map()

scala> captial ++= List("US" -> "Washington","France" -> "Pairs","Japan" -> "Tok

yo")

res2: captial.type = Map(France -> Pairs, Japan -> Tokyo, US -> Washington)

scala> captial

res3: scala.collection.mutable.Map[String,String] = Map(France -> Pairs, Japan -

> Tokyo, US -> Washington)

scala> captial("US")

res4: String = Washington

类似，可以以同样的方法创建同步集。例如可以通过混入SynchronizedSet特质创建同步的HashSet，例如：

import scala.collection.mutable  
import scala.collection.mutable.{Set,HashSet,SynchronizedSet}  
  
object SetMaker {  
  
 def makeSet:Set[String] ={  
 new mutable.HashSet[String] with mutable.SynchronizedSet[String]{  
  
 }  
 }  
}

在需要考虑使用同步集合的时候，还可以考虑使用java.util.concurrent的并发集合。还可以使用非同步的集合及scala.actor。

**（4）可变集合 vs 不可变集合**

如果集合中元素数量不多，使用不可变集合更为紧凑。如果需要，可以把不可变集合转为可变集合。

为了便于完成不可变集合到可变集合的转换，或反向转换，scala提供了一些语法糖。纵使不可变集和映射不支持+=方法，scala还是为此提供了+=的语法解释。如果写了a += b，而啊不支持+=的方法，scala尝试把它解释为 a = a + b。例如，不可变集不支持+=方法：

scala> val people = Set("Jensen","Lily")

people: scala.collection.immutable.Set[String] = Set(Jensen, Lily)

scala> people += "Andy"

<console>:9: error: value += is not a member of scala.collection.immutable.Set[S

tring]

people += "Andy"

^

但是如果把people定义为var，那么集合就可以用+=方法完成“更新”，尽管它是不可变类型的。例如：

scala> var people = Set("Andy","Jack")

people: scala.collection.immutable.Set[String] = Set(Andy, Jack)

scala> people += "Morris"

scala> people

res2: scala.collection.immutable.Set[String] = Set(Andy, Jack, Morris)

经过+=操作后，变量people指向新的不可变集。同样的理念可以应用于所有以=为结尾的方法。例如：

scala> people -= "Jack"

scala> people

res4: scala.collection.immutable.Set[String] = Set(Andy, Morris)

scala> people ++= List("Kim","Bruce")

scala> people

res6: scala.collection.immutable.Set[String] = Set(Andy, Morris, Kim, Bruce)

这种语法糖不仅对集合有效，它还具有普适性。例如，下面是用在浮点数上的：

scala> var roughyPi = 3.0

roughyPi: Double = 3.0

scala> roughyPi += 0.1

scala> roughyPi += 0.04

scala> roughyPi

res9: Double = 3.14

**（5）初始化集合**

初始化方法：把参数传给要调用的集合类型对应的伴生对象的工厂方法。例如：

scala> List(1,2,3)

res10: List[Int] = List(1, 2, 3)

scala> Set('a','b','c')

res11: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> import scala.collection.mutable

import scala.collection.mutable

scala> mutable.Map("hi" -> 2, "there" -> 5)

res12: scala.collection.mutable.Map[String,Int] = Map(hi -> 2, there -> 5)

scala> Array(1.0,2.0,3.0)

res13: Array[Double] = Array(1.0, 2.0, 3.0)

通常，scala编译器会根据传递给工厂方法的元素推断集合的元素类型，但当往集合中添加不同类型的元素时，会报类型不匹配的错误。例如：

scala> val stuff = mutable.Set(42)

stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> strff += "addbcddfsf"

<console>:9: error: not found: value strff

strff += "addbcddfsf"

^

scala> stuff += "addbcddfsf"

<console>:10: error: type mismatch;

found : String("addbcddfsf")

required: Int

stuff += "addbcddfsf"

^

如果需要制定stuff的类型为Any，需要定义时声明，例如：

scala> val stuff = mutable.Set[Any](42)

stuff: scala.collection.mutable.Set[Any] = Set(42)

scala> stuff += "aabbccdd"

res16: stuff.type = Set(aabbccdd, 42)

scala> stuff

res17: scala.collection.mutable.Set[Any] = Set(aabbccdd, 42)

另一种情况时，想要把集合初始化为指定类型。例如，下面代码中不能把colors传递给TreeSet的工厂方法：

scala> val colors = List("blue","yellow","red","green")

colors: List[String] = List(blue, yellow, red, green)

scala> import scala.collection.immutable.TreeSet

import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)

<console>:10: error: No implicit Ordering defined for List[String].

val treeSet = TreeSet(colors)

^

正确的方法是先创建TreeSet[String]对象并用++操作把列表元素添加进去：

scala> val treeSet = TreeSet[String]() ++ colors

treeSet: scala.collection.immutable.TreeSet[String] = TreeSet(blue, green, red,

yellow)

**数组与列表之间的转换**

集合转为列表或数组，例如：

scala> treeSet.toList

res19: List[String] = List(blue, green, red, yellow)

scala> treeSet.toArray

res20: Array[String] = Array(blue, green, red, yellow)

在这里，treeSet里面的元素已经经过了排序，转成的array或list也是经过排过序的。

在转成数组或列表时，需要复制集合中的所有元素。

**集和映射的可变与不可变转换**

将集在可变类型和不可变类型之间的转换：

scala> import scala.collection.mutable

import scala.collection.mutable

scala> treeSet

res22: scala.collection.immutable.TreeSet[String] = TreeSet(blue, green, red, ye

llow)

scala> val mutaSet = mutable.Set.empty ++ treeSet

mutaSet: scala.collection.mutable.Set[String] = Set(red, blue, green, yellow)

scala> val immutaSet = Set.empty ++ mutaSet

immutaSet: scala.collection.immutable.Set[String] = Set(red, blue, green, yellow

)

同样的方法，可以实现可变映射和不可变映射之间的转换：

scala> val mutaMap = mutable.Map("i" -> 1,"ii" -> 2)

mutaMap: scala.collection.mutable.Map[String,Int] = Map(ii -> 2, i -> 1)

scala> val immuMap = Map.empty ++ mutaMap

immu: scala.collection.immutable.Map[String,Int] = Map(ii -> 2, i -> 1)

**（6）元组**

元组可以把固定数量的条目组合在一起，以便于作为整体传送。

元组可以保存不同类型的对象，而数组和列表是同质的。例如：

scala> (1,"hello",Console)

res23: (Int, String, Console.type) = (1,hello,scala.Console$@7f34fa3c)

因为元组可以组合不同类型的对象，因此它不能继承自Iterable。

元组常用类返回方法的多个值。例如：

def longestWord(words:Array[String])={  
 var word = words(0)  
 var idx = 0  
 for(i <- 1 until words.length)  
 if(words(i).length > word.length){  
 word = words(i)  
 idx = i  
 }  
  
 (word,idx)  
}  
  
val (word,idx) = longestWord("The quick brown fox".split(" "))

访问元组的元素：

scala> longest.\_1

res0: String = quick

scala> longest.\_2

res1: Int = 1

可以把元组的元素赋值给其他变量：

scala> val (word,idx) = longest

word: String = quick

idx: Int = 1

## 有状态的对象

### （1）什么让对象具有状态

对于有状态的对象来说，方法调用或字段访问的结果将取决于之前已经执行过的操作。

例如，一个银行账户的例子：

class BankAccount {  
 private var bal:Int = 0  
 def balance:Int = bal  
 def deposit(amount:Int): Unit ={  
 require(amount >0 )  
 bal += amount  
 }  
 def withdraw(amount:Int):Boolean={  
 if(amount > bal) false  
 else{  
 bal -= amount  
 true  
 }  
 }  
}

使用BankAccount类：

scala> val account = new BankAccount

account: BankAccount = BankAccount@56ac4b99

scala> account deposit 100

scala> account withdraw 80

res1: Boolean = true

scala> account withdraw 80

res2: Boolean = false

可以看到最后两次调用withdraw方法时，产生不同的返回值。所以说，实例化的对象是一个有状态的对象。

### （2）可重新赋值的变量和属性

对可以重新赋值的变量可以执行两种基本操作：获取变量的值、更新变量的值。Scala里面，对象的每个非私有的var类型成员变量都隐含定义了getter和setter的方法。var变量x的getter方法命名为“x”，它的setter方法命名为“x\_=”。

例如，如果类中存在var定义：

var hour = 12

定义该变量之后，会额外产生getter方法“hour”，以及setter方法“hour\_”，然后该字段始终被标记为private[this]，表示它只能被包含它的对象访问。但是getter和setter方法获得了原本var变量的可见性。如果var定义为public，则它的getter和setter与之相同，如果定义为protected，那它们也是protected。

例如：

class Time{

var hour = 12

var minute = 0

}

上述代码的实现方式与下列类定义完全相同：

class Time{

private[this] var h = 12

private[this] var m = 0

def hour:Int

def hour\_ = (x:Int) { h = x}

def minute:Int = m

def minute\_ = (x:Int) { m = x}

}

这种支持getter和setter的方式，可以通过直接定义getter和setter方法以取代定义var变量，就可以扩展变量的赋值和访问操作。例如，重新定义Time类：

class Time {  
 private[this] var h = 12  
 private[this] var m = 12  
  
 def hour:Int = h  
 def hour\_= (x:Int){  
 require(0 <= x && x < 24)  
 }  
  
 def minute:Int = m  
 def minute\_= (x:Int): Unit ={  
 require(0 < x && x < 60)  
 m = x  
 }  
}

也可以只定义getter和setter方法，而不定义关联字段。例如：

class Thermometer{  
 var celsius: Float = \_ // =\_为字段的初始化器  
  
 def fahrenheit = celsius\*9 /5 +32  
 def fahrenheit\_= (f:Float){  
 celsius = (f - 32)\*5 /9  
 }  
 override def toString = fahrenheit + "F/" + celsius + "C"  
}

scala中不可随意省略初始化器“=\_”。如果写成：

var Celsius: Float

这将定义为抽象变量，而不是未初始化的变量。

调用类Thrmometer，如下：

scala> val t = new Thermometer

t: Thermometer = 32.0F/0.0C

scala> t.celsius = 100

t.celsius: Float = 100.0

scala> t

res4: Thermometer = 212.0F/100.0C

scala> t.fahrenheit

res5: Float = 212.0

scala> t.fahrenheit = -40

t.fahrenheit: Float = -40.0

### （3）案例研究：离散事件模拟

数字电路模拟器的设计与实现。

### （4）为数字电路定制的语言

## 类型参数化

类型参数化能够让你编写泛型类和特质。例如set是泛型的，带类型参数：定义为Set[T]。所以，任何特定set的实例都应该写成Set[String]，或者Set[Int]这样子，必须是某种东西的set。而且还定义了参数化类型的继承关系，例如Set[String]是Set[AnyRef]的子类型。

### （1）queus函数式队列

函数式队列是一种数据结构，具有以下三种操作方式：

head 返回队列的第一个元素

tail 返回除第一个元素之外的队列

append 返回尾部添加了指定元素的队列

函数式队列是不可变的，在作添加操作时，返回了包含添加元素的新队列。

函数式队列类似于List，是一种完全持久的数据结构，即使在扩展和改变之后，旧版本的对象仍然可用。它们都支持head和tail操作。不同的是，list用::操作在前端进行扩展，而队列使用append扩展在对象后段。

以下是基于List实现的队列，代码如下：

class SlowAppendQueue[T](elements:List[T]) { //效率不高  
  
 def head = elements.head  
  
 def tail = new SlowAppendQueue[T](elements.tail)  
  
 def append(x:T) = new SlowAppendQueue[T](elements ::: List(x))  
}

以上代码的问题在于append操作，它花费的时间与存储于队列的元素数量成正比。如果使append操作消耗常量时间，可以尝试把list里面的元素倒序排列，例如：

class SlowHeadQueue[T](smele:List[T]) { //效率不高  
 //smel是把elements反转过来  
 def head = smele.last  
 def tail = new SlowHeadQueue[T](smele.init)  
 def append(x:T) = new SlowHeadQueue[T](x :: smele)  
}

现在append是常量时间了，但head和tail不是了。

如果使三个操作都消耗常量时间，可以尝试把这两种操作进行组合。思想如下：用两个List，称其为leading和trailing，来表达队列。leading包含了前段元素，而trailing包含了反向排列的后段元素。队列在任意时刻都可以表达为“leading ：：： trailing.reverse”。

当需要添加新元素时，只需要使用::操作符把它cons到trailing，因此append是常量时间。这也意味着，当原始的空队列通过后继的append操作构建起来时，trailing将不断增加而leading将始终空白。于是，在对空的leading第一次执行head或tail操作之前，trailing应该被反转并复制给leading。我们称这个操作为mirror。以下是代码实现：

class AdvQueue[T](private val leading:List[T],private val trailing:List[T]){  
 private def mirror =  
 if(leading.isEmpty)  
 new AdvQueue(trailing.reverse,Nil)  
 else  
 this  
  
 def head = mirror.leading.head  
  
 def tail ={  
 val q = mirror  
 new AdvQueue[T](q.leading.tail,q.trailing)  
 }  
  
 def append(x:Int) =  
 new AdvQueue(leading, x :: trailing)  
}

### （2）信息隐藏

**私有构造器及工厂方法**

Scala中的构造器也可以私有化（只能被类本身及伴生对象访问），例如：

class AdvQueue[T] private(

private val leading:List[T],

private val trailing:List[T]

)

这时，在类的外部，无法访问该类的构造器。可以添加辅助构造器：

def this() = this(Nil,Nil)

还可以带上初始元素列表：

def this(elems: T\*) = this(elems.toList,Nil)

另一种方式识通过工厂方法，通过初始元素序列创建队列。比较简洁的方法是创建与类同名的AdvQueue对象（伴生对象与类具有相同的访问权）及Apply方法，例如：

object AdvQueue{  
 def apply[T](xs:T\*) = new AdvQueue[T](xs.toList,Nil)  
}

注意，这里的工厂方法为apply，因此可以用类似于AdvQueue(1,2,3)这样的表达式创建队列。Scala中没有全局可见的方法，每个方法都要被包含在对象或类中。

**可选方案：私有类**

把类本身隐藏掉 ，仅提供能够暴露类公共接口的特质。例如：

trait Queue[T] {  
 def head: T  
 def tail: Queue[T]  
 def append(x:T) : Queue[T]  
}  
  
object Queue {  
 def apply[T](xs:T\*): Queue[T] =  
 new QueueImpl[T](xs.toList,Nil)  
  
 private class QueueImpl[T] (  
 private val leading:List[T],  
 private val trailing:List[T]  
 ) extends CusQueue[T]{  
  
 def mirror =  
 if(leading.isEmpty)  
 new QueueImpl(trailing.reverse,Nil)  
 else  
 this  
  
 def head:T = mirror.leading.head  
  
 def tail:QueueImpl[T] = {  
 val q = mirror  
 new QueueImpl(q.leading.tail,q.trailing)  
 }  
  
 def append(x:T) =  
 new QueueImpl(leading,x :: trailing)  
 }  
}

### （3）变化型注解

在上例的代码中，Queue被定义为特质，后面加参数后例如Queue[String]就成为了类型。Queue可成为类型构造器，因为有了它就可以通过制定参数类型构造新的类型，比如Queue[Int]、Queue[AnyRef]。也可以认为Queue是泛型的特质。

如果S是类型T的子类型，那么是否可以把Queue[S]当作Queue[T]的子类型？如果是，你可以认为Queue特质是与它的类型参数T保持**协变（convaritant）**的。或者，由于它只有一个类型参数，你可以简单地说Queue是协变的。协变的Queue将意味着，你可以 把，比方说Queue[String]，传递给以Queue[AnyRef]为参数类型的方法。

直观上这些看上去都很合理，因为String队列似乎就是AnyRef的特例。然后在scala中，泛型类缺省值的是非协变的（novariant）子类型化。也就是说，不同元素类型的队列之间没有子类型关系。Queue[String]对象不能被用作Queue[AnyRef]。然而，可以用以下方式改变Queue类定义的第一行，以要求队列协变（弹性）的子类型化：

Trait Queue[+T] { …} //正号为变化型注解

在正常的类型参数前加上+号表明这个参数的子类型化是协变（弹性）的。这个符号是向Scala说明你希望可以把，比方说Queue[String]，当作Queue[AnyRef]的子类型。编译器将坚持Queue被这种子类型化所说的方式定义。

除了+号以外，还可以前缀加上-号，这表明是需要***逆变（contravariant）***子类型化。如果Queue定义如下：

trait Queue[-T] { … } //负号为变化型注解

那么如果T是类型S的子类型，这将隐含Queue[S]是Queue[T]的子类型。无论类型参数是协变的，逆变的，还是非协变的，都给称为***参数的变化型***。可以放在类型参数前的+号和-号被称为***变化型注解***。

在纯函数式中，许多类型都是自然协变（弹性）的。然后，一旦注入了可变数据，情况就改变了。可以假象有这么一种简单的只能读写一个元素的单元格（Cell）类型，代码如下：

class Cell[T] (init:T) {  
 private[this] var current = init  
 def get = current  
 def set(x:T) { current = x}  
}

代码中的Cell被声明为非协变的。为了便于讨论，假设暂时声明为协变的—也就是说，声明为Cell[+T]，并发送给Scala编译器。于是，可以构建如下语句：

val c1 = new Cell[String]("abc")

val c2:Cell[Any] = c1

c2.set(1)

val s:String = c1.get

这四条语句从自身看来，每一条都很正常。第一行创建String单元格并保存在c1中。第二行定义c2，类型是Cell[Any]，并初始化为c1。这没问题，因为前提假设是Cell为协变的。第三行把c2的值设为1。这也没问题，因为1是c2的元素类型Any的实例。第四行把c1的元素赋给字符串s，这也没有问题。但把这四行代码放在一起，完成是把整数1赋值给字符串s，这明显是对类型声明的破坏。

哪个操作应该对运行期的失败负责呢？一定是使用了协变子类型化的第二行。其他的语句都太简单基础了。因此，String类型的Cell并不就是Any类型的Cell，因为有些事可以对Any的Cell做，但不能对String的Cell做。比如，不能以Int类型的入参调用String的Cell的set方法。

实际上，如果把协变版的Cell类型传递给Scala编译器，将返回编译时错误：

Error:(7, 11) covariant type T occurs in contravariant position in type T of value x

def set(x:T) { current = x}

**变化型和数组**

在java中，数组被认为是协变的。例如下面是用java数组模拟了上述单元格的交互操作：

//在java里

String[] a1 = {"abc"};

Object[] a2 = a1;

a2[0] = new Integer(17);

String s= a1[0];

这个例子能够通过编译，但执行程序会在Integer对象赋值给a2[0]时引发ArrayStore异常：

Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer

at cn.com.dist.xml.appTest.main(appTest.java:44)

这里的情况是，java在运行时保存了数组的元素类型。于是每次数组元素更新时，新元素值都会用存储的类型校验合法性。如果不是该类型的实例，就会抛出ArrayStore异常。

Java中采用这样的设计，是希望有一个通用处理数组的简单方法。例如，想要能够编写方法排序数组的所有元素，使用以下的方法签名带入对象数组：

void sort(Object[] a ,Comparator cmp) { ... }

数组的协变被用来确保任意参考类型的数组都可以传入排序方法。当然，随着java引入了泛型，这种排序方法现在可以带有类型参数，因此数组的协变不再有用。只是考虑到兼容的原因，直到今天它都存在于java中。

Scala不认为数组是协变的，以尝试保持比java更高的存粹性。比如，在scala中实现以上代码是会报如下异常：

scala> val a1= Array("abc")

a1: Array[String] = Array(abc)

scala> val a2:Array[Any] = a1

<console>:8: error: type mismatch;

found : Array[String]

required: Array[Any]

Note: String <: Any, but class Array is invariant in type T.

You may wish to investigate a wildcard type such as `\_ <: Any`. (SLS 3.2.10)

val a2:Array[Any] = a1

^

这里的情况是scala把数组当作是非协变的，因此Array[String]对象不能当作与Array[Any]一致。然而，有时候需要使用对象数组作为模拟泛型数组的手段与java的遗留方法执行交互。例如，你或许希望调用类似于之前描述的，以字符串数组为参数的排序方法。为了满足这种情况，scala允许你把T类型的数组造型为任意T的超类型的数组：

scala> val a2:Array[Object] = a1.asInstanceOf[Array[Object]]

a2: Array[Object] = Array(abc)

编译时的造型始终合法，并且将在运行时永远成功，因为jvm的内含运行时模型与java语言一致，把数组当作是协变的。但是之后还是会收到ArrayStore异常，就如在java中遇到的一样。

### （4）检查变化型注解

现在你已经看到了一些不适用变化的例子，你或许想知道哪种类型的类定义是必须拒绝的，哪种是可以接受的。到目前为止，所有类型可靠性的冲突都与一些可重新赋值的字段或数组元素有关。相反，队列的纯函数式实现似乎看上去好像是协变很好的候选者。然后，下面的例子可以说明即使没有可重新赋值的字段，仍然可以“设计”出不健全的情况。

为了举例说明，假设之前定义的Queue是协变的，然后创建指定元素类型为 Int的队列子类，并重载append方法：

class StrangeIntQueue extends Queue[Int]{  
 override def append(x:Int) ={  
 println(Math.sqrt(x))  
 super.append(x)  
 }  
}

使用下面两行编写反例：

val x:Queue[Any] = new StrangeIntQueue  
x.append("abc")

第一行是有效的，因为StrangeIntQueue是Queue[Int]的子类，并且假设队列是协变的，Queue[Int]是Queue[Any]的子类型。第二行是有效的，因为你可以对Queue[Any]添加String对象。然而这两行放一起就是取字符串的平方根，毫无意义。

很明显，不是只有可变字段会使协变类型出错。这种问题更为普遍。只要泛型的参数类型被当作方法参数的类型，那么包含它的类或特质就有可能不能与这个类型参数一起协变。对于队列来说，append方法违反了以下情况：

class Queue[+T]{

def append(x:T) =

...

}

可重新赋值的字段是“不允许使用+号注解的类型参数用做方法参数类型”的。因为，可重新赋值的字段，“var x:T”在scala里被看作是一种getter方法，“def x:T”，和setter方法，“def x\_=(y:T)”。正如你所见的，setter方法带有字段类型为T的参数，因此类型将不是协变的。

### （5）下届

回到Queue类中来，已经看到了前面的Queue[T]定义中不能实现对T的协变，因为T作为类型参数类型出现在append方法中，而这里是负的位置。

有一个版本可以打开这个结：可以通过把append变为多态以使其泛型化（即提供给append方法类型参数）并使用它的类型参数的下界。代码如下：

class Queue[+T] (private val leading:List[T],

private val trailing:List[T]){

def append[U >: T](x:U) =

new Queue[U](leading, x :: trailing) //...

}

新的定义指定了append的类型参数U，并通过语法“U >: T”，定义了T为U的下界。结果，U必须是T的超类型。append的参数现在变为类型U而不是类型T，而方法的返回值现在也变为Queue[T]，取代了Queue[T]。

注意：超类型与子类型关系是自反的，也就是说类型对于自身来讲，既是超类型也是子类型。即使T是U的下界，也仍然能够把T传递给append方法。

至于例子，可以假设存在类Fruit及两个子类，Apple和Orange。通过使用Queue类的新定义，现在可以把Orange对象加入到Queue[Apple]。而返回的结果为Queue[Fruit]类型。

append的改进定义时类型正确的。直观的看，如果T比预期的类型更为特化（例如，用Apple替代Fruit），那么append的调用就仍然正确，因为U（Fruit）是T（Apple）的超类型。

可以认为append的新定义比旧的要好，因为它更为普遍。不像旧的版本，新定义允许你把队列元素类型T的任意超类型U的对象添加进来，结果为Queue[U]。它与队列的协变性一起，以自然的方式提供了不同元素类型队列建模的合适的弹性。

这也说明变化型注解与下界可以相互协作。这是很好的类型驱动设计的例子，由接口的类型导引其细节的设计和实现。在队列的例子中，你或许没有想到要使用下界重构append方法的实现，但你或许想要让队列是协变的。这种情况下，编译器将指出append变化型的错误。而通过添加下界来修正变化型错误使得append更为通用并且也连带使得队列作为整体功能更为有用。

我们观察到的这种情况，其实也是scala更推崇声明点的变化型，而非java的通配符中可以发现的使用点的变化型的主要原因。如果是使用点的变化型，那你是在独立设计类。需要类的客户把正确的值带入通配符位置，并且如果他们做错了，则某些重要的实例方法将不再可用。变化型变成了棘手的工作，用户常常会搞错，他们会想要逃避，认为通配符和泛型过于复杂。而对于定义过程中的变化型，你是在向编译器解释你的意图，编译器会复查你是否的确实现了意图。

### （6）逆变

到目前为止，所有的例子要么是协变的，要么是非协变的。但还有一些情况，逆变才显得自然。例如，下面一段输出通道特质：

trait OutputChannel[-T] {

def write(x: T)

}

这里，OutputChannel被定义为T的逆变。因此，AnyRef的输出通道，打个比方，将是string的输出通道的子类型。尽管这看起来不是很直观，但确实很有道理。如果想要明白这是为什么，可以考虑一下你能够对OutputChannel[String]做什么。唯一支持的操作就是写一个String给它。而同样的操作对于OutputChannel[AnyRef]来说也支持。因此用OutputChannel[AnyRef]替代OutputChannel[String]是安全的。相反，在需要OutputChannel[AnyRef]的地方替换成OutputChannel[String]却是不安全的。毕竟，你可以把任何对象发送给OutputChannel[AnyRef]，而OutputChannel[String]要求写入的值只能是字符串。

这也点出了类型系统设计的通用原则：如果你能再需要类型U的值的地方替换成类型T的值，那么类型T是类型U的子类型的假设是安全的。这被称为里氏替代原则（Liskov Substiturion Principle,LSP）。原则的前提是T要支持U所支持的操作并且对于T的所有操作来说，与U的相应操作比较，需求的要更少，而提供的要更多。

在输出通道的例子中，OutputChannel[AnyRef]可以是OutputChannel[String]的子类型，因为它们都支持同样的write操作，并且这个操作对于OutputChannel[AnyRef]来说要求的比OutputChannel[String]更少。这里的“更少”是指前者的参数仅需要AnyRef对象即可，而后者则需要String对象才行。

有时，协变与逆变会同时混合在一个类型中出现。比较典型的例子就是Scala的函数特质。比方说，你在编写函数类型A => B的时候，scala会把它扩展成Function1[A,B]。而Function1在标准库里的定义同时使用了协变和逆变。Function1特质的函数参数类型S是逆变的，而结果类型T是协变的，这可以满足里式替换原则，因为参数是需求的东西，而结果是提供的东西。具体代码如下：

trati Function1[-S,+T]{

def apply(x:S):T

}

现有如下代码：

class Publication(val title:String)  
class Book(title:String) extends Publication(title)  
  
object Library {  
 val books:Set[Book] =  
 Set{  
 new Book("Programming in Scala.")  
 new Book("Walden")  
 }  
 def printBookList(info: Book => AnyRef): Unit ={  
 for(book <- books) println(info(book))  
 }  
}

object Customer extends App{  
 def getTitle(p:Publication):String = p.title  
 Library.printBookList(getTitle)  
}

上述代码中，类Publication包含一个参数字段Titile，类型为String。类Book扩展了Publication并把它的字符串参数title前转给超类的构造器。Library单例对象定义了书籍对象的集（set）及方法printBookList，它带一个名为info，类型为Book => AnyRef的方法参数。换句话说，也就是printBookList唯一的参数是带Book参数并返回AnyRef的方法。Customer应用定义了方法getTitile，输入唯一的参数Publication，并返回String，为传入参数的标题。

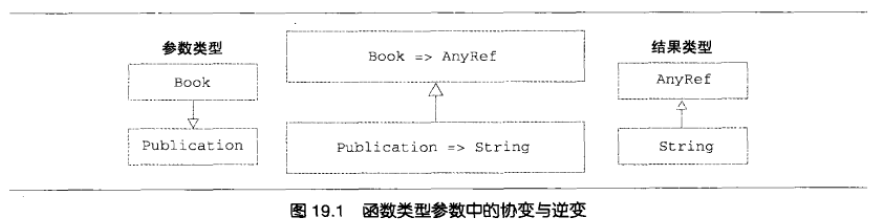
现在看一下Customer的最后一行。这行调用了Library的printBookList方法并把getTitle打包为函数值传递给方法。

Library.printBookList(getTitle)

尽管函数的结果类型String是AnyRef的子类型，这行代码仍然要对printBookList的info参数做类型检查。编译器通过了对代码的检查，因为函数结果类型被声明为协变的。如果你看过printBookList方法体的内部实现，你就可以清楚（编译器）这样做是有道理的。

printBookList方法枚举了它的书籍列表，并对每一本书调用传入的参数。它把返回的AnyRef类型的结果传递给println，由println对这个结果调用toString方法并打印。这个行为对于String及所有AnyRef子类的对象都有效，这也是函数结果类型的协变意义所在。

现在请考虑一下传递个printBookList方法的函数参数类型。尽管printBookList的参数类型被声明为Book，而我们传递给它的getTitle带publication参数，是Book的超类型。可以这样做的原因是printBookList的参数类型是Book，所有printBookList方法将仅会传递Book对象给函数。因为getTitle的参数类型为Publication，函数体将只能访问其参数p，是被声明为类Publication的成功。最后还由于任何声明在Publication内的方法在它的子类Book内也同样有效，因此所有环节都被打通了，这也就是函数参数类型的逆变的意义所在。如下图所示：



如上图所示，Publication => String是Book => AnyRef的子类型。由于Function1的结果类型被定义为协变的，所有可以看到图右侧显示的两个结果类型的继承关系与在中间的显示的两个函数的关系方向是一致的。相反，由于Function1的参数类型被定义为逆变的，所有左侧显示的两个函数类型的继承关系与函数关系的方向相反。

### （7）对象私有数据

到目前为止，Queue类仍然有一些问题存在。如果head被一遍遍地调用许多次，而leading列表为空，那么mirror操作可能会重复地把trailing复制到leading列表。这可以通过添加一些做判断的副作用代码对这些无用的复制工作加以避免。下面的代码实现了一种新的Queue实现，可以对任意系列的head操作执行最多一次trailing到leading的调节操作：

class Queue[+T] (  
 private[this] var leading:List[T],  
 private[this] var trailing:List[T]  
 ){  
 private def mirror() =  
 if(leading.isEmpty) {  
 while (!trailing.isEmpty){  
 leading = trailing.head :: leading  
 trailing = trailing.tail  
 }  
 }  
  
 def head:T = {  
 mirror()  
 leading.head  
 }  
  
 def tail:Queue[T] ={  
 mirror()  
 new Queue(leading.tail,trailing)  
 }  
 def append[U >: T](x:U) =  
 new Queue(leading,x :: trailing)  
  
}

与之前版本的不同之处在于现在的leading和trailing都是可以重新赋值的变量，而mirror从trailing反向复制到leading的操作时在当前队列上的副作用，不再返回新的队列。由于leading和trailing都是私有变量，因此这种副作用存粹是Queue操作的内部实现，从而使得它对于Queue的客户来说是不可见的。因此，可以说新版本的Queue仍然定义了纯函数式的对象，尽管包含了可重新赋值的字段。

你或许想知道是否这段代码能够通过scala的类型检查。毕竟，队列现在包含两个协变参数类型T的可重新赋值的字段。这难道没有破败变化型的规则吗？的确如此，原因在于leading和trailing具有private[this]修饰符，从而都被声明为对象私有的了。

对象私有成员仅能在被定义的对象内部访问。而在它们被定义的同一个对象内访问这些变量并不会引起与变化型有关的问题。较直观的解释是，为了构造变化型能够导致类型错误的例子，需要比定义的对象类型更弱的静态类型的对象引用。然而对于对象的私有值的访问来说，这完全不可能。

Scala的变化型检查规则包含了关于对象私有定义的特例。当检查到带有+/-号的类型参数只出现在具有相同变化型分类的位置上时，这种定义将被忽略。因此，以上代码将通过编译。而如果遗漏了两个private修饰符的[this]限定符，将会有如下错误：

Error:(24, 7) covariant type T occurs in contravariant position in type List[T] of value trailing\_=

class Queue[+T] (

^

Error:(24, 7) covariant type T occurs in contravariant position in type List[T] of value leading\_=

class Queue[+T] (

^

### （8）上届

之前的例子中，有通过把特质Ordered混入到类中，并实现Ordered唯一的抽象方法compare，就可以使用<，>,<=和>=等做类实例的比较。下面代码展示了Ordered被混入到Person类中，并进行比较：

class Person (  
 val firstName:String,  
 val lastName:String  
 ) extends Ordered[Person]{  
 def compare(that:Person) = {  
 val lastNameComparison =  
 lastName.compareToIgnoreCase(that.lastName)  
 if(lastNameComparison != 0)  
 lastNameComparison  
 else  
 firstName.compareToIgnoreCase(that.firstName)  
 }  
  
 override def toString = firstName + " " + lastName  
}

下面对定义类实例进行比较：

scala> val robert = new Person("Robert","Jones")

robert: Person = Robert Jones

scala> val sally = new Person("Sally","Smith")

sally: Person = Sally Smith

scala> robert < sally

res1: Boolean = true

为了让传递给你的新排序函数的列表类型混入到Ordered中，你需要使用上届。除了使用<:符号代替用于下界定义的>:符号，上届的指定与下界类似。使用“T<: Ordered[T]”语法，这里指明了类型参数T具有上届，Ordered[T]。即传递给orderedMergeSort列表的元素类型必须是Ordered的子类型。因此，可以传递List[Person]给orderedMergeSort，因为Person混入了Ordered。例如，请看如下列表：

def orderedMergeSort[T <: Ordered[T]](xs:List[T]):List[T] = {  
 def merge(xs:List[T],ys:List[T]):List[T] =  
 (xs,ys) match {  
 case (Nil,\_) => ys  
 case (\_,Nil) => xs  
 case (x :: xs1,y :: ys1) =>  
 if(x < y) x :: merge(xs1, ys)  
 else y :: merge(xs , ys1)  
 }  
  
 val n = xs.length/2  
 if( n == 0) xs  
 else{  
 val (ys,zs) = xs splitAt n  
 merge(orderedMergeSort(ys),orderedMergeSort(zs))  
 }

调用排序函数如下：

val people = List(  
 new Person("Larry","Wall"),  
 new Person("Anders","Hejlsberg"),  
 new Person("Guido","van Rossum"),  
 new Person("Alan","Kay"),  
 new Person("Yukihiro","Matsumoto")  
)

因为people列表的元素类型Person混入了Ordered[People]（因此也是它的子类型），所以可以把列表传递给orderedMergeSort函数：

orderedMergeSort(people).foreach(println)

Anders Hejlsberg

Alan Kay

Yukihiro Matsumoto

Guido van Rossum

Larry Wall

尽管orderedMergeSort函数很好地诠释了上届的用法，但它实际上并非设计能够充分利用Ordered特质的排序函数的通用方式。比方说，你不能用orderedMergeSort函数对整数列表做排序，因为Int不是Ordered[Int]的子类型：

val wontCompile = orderedMergeSort(List(3,2,1))

Error:(50, 21) inferred type arguments [Int] do not conform to method orderedMergeSort's type parameter bounds [T <: Ordered[T]]

val wontCompile = orderedMergeSort(List(3,2,1))

^

## 抽象成员

定义：不完全定义的类或特质的成员成为抽象成员。抽象成员将被声明类的子类实现。

抽象成员包括抽象方法、抽象字段、抽象类型（al、var、方法及类型）。

### （1）抽象成员的快速浏览

下面的特质对每种抽象成员各声明了一个例子：

trait Abstract{

type T

def transform(x:T):T

val initial:T

var current: T

}

Abstract的具体实现需要对每种抽象成员填入定义，例如：

class Concrete extends Abstract{  
 type T = String  
  
 def transform(x: String): String = x + x  
  
 val initial: String = "hi"  
 var current: String = initial  
}

### （2）类型成员

抽象类型这个术语在scala中是指不带具体定义的，（由“type”关键字）声明为类的或特质的成员的类型。类本身可以是抽象的，而特质本来就是抽象的，但无论哪种都不是Scala所指的抽象类型。Scala的抽象类型永远都是某个类或者特质的成员，就好像特质Abstract里的类型T那样。

可以把非抽象的类型成员，如Concreate类里的类型T，想象成是为类型定义新的名称，或别名的方式。例如Concreate类中，类型String被指定了别名T。因此，任何出现在Concreate定义中的T指的都是String。这也包含了transform的参数和结果类型，initial以及current，这些在Abstract超特质中声明的时候都提到了T的成员。因此，当Concreate类实现这些抽象成员的时候，所有的T都被解释为String。

使用类型成员的理由之一是为类型定义短小的、具有说明性的别名，因为类型的实际名称可能比别名要更冗长，或语义不清。这种类型成员有助于净化类或特质的代码。类型成员的另一种主要用途是声明必须被定义为子类的抽象类型。

### （3）抽象val

抽象val以如下形式定义：

val initial : String

定义了val的名称和类型，但不定义值。该值必须有子类的具体val定义提供。例如实现方法：

val initial = “hi”

如果你不知道类中定义的值的确切内容，但知道该值对于每个类实例来说都不能改变的时候，可以在类中使用抽象的val声明。

抽象的val声明类似于抽象的无参数方法声明，如：

def initial:String

客户代码将使用完全一致的方式（即obj.initial）引用val及方法。然而，如果initial是抽象val，那么客户就得到了保证，每次引用都将得到同样的值。如果initial是抽象方法，就不会获得这样的保证，因为在这种情况下initial可以被实现为每次调用时都返回不同值的具体方法。

换句话说，抽象的val限制了合法的实现方式：任何实现都必须是val类型的定义，不可以是var或def。另一方面，抽象方法声明可以被实现为具体的方法定义或者具体的val定义。例如：

abstract class Fruit{  
 val v : String  
 def m: String  
}  
  
abstract class Apple extends Fruit {  
 val v: String  
 val m: String //可以用val重新def  
}  
  
abstract class BadApple extends Fruit{  
 def v:String //错误：不能用def重新val  
 def m:String  
}

上述代码中，Apple是Fruit类的合法实现，而BadApple不是。

### （4）抽象var

与抽象val类型，抽象var只定义名称和类型，没有初始值。例如：

trait AbstractTime{  
 var hour:Int  
 var minute:Int  
}

声明为类成员的var实际配备了getter和setter方法，对于抽象var也是如此。比如你声明了名为hour的抽象var，实际上是隐式声明了抽象getter方法“hour”及抽象setter方法“hour\_=”。这里根本没有定义什么可重新赋值的字段。例如，上述代码的定义与下面的代码完全等价：

trait AbstractTime{  
 def hour:Int  
 def hour\_=(x:Int)  
   
 def minute:Int  
 def minute\_=(x:Int)  
}

### （5）初始化抽象val

抽象val有时候会扮演类似于超类的参数这样的角色：它们能够让你在子类中提供超类缺少的细节信息。这对于特质来说尤其重要，因为特质缺少能够用来传递参数的构造器。因此通常参数化特质的方式就是通过需要在子类中实现的抽象val完成。例如：

trait RationalTrait{  
 val numerArg:Int  
 val denomArg:Int  
}

为了实例化这个特质，需要实例化抽象val，如下：

new RationalTrait {

val denomArg: Int = 1  
 val numerArg: Int = 2  
}

这里关键字new出现在特质名称（RationalTrait）之前，然后是花括号包围的类结构他。这个表达式可以产生混入了特质并被结构体定义的匿名类（anonymous class）实例。这种特殊的匿名类实例化结果与使用new Raional（1,2）的方式创建的实例具有类似效果。不过这种类并非完美，两者在表达式的初始顺序方面存在着微妙的差别。当你写下：

new Raional(expr1, expr2)

两个表达式，expr1和expr2，会在类Rational初始化之前计算，因此在执行类的初始化操作时，expr1和expr2的值已经可用。然而对于特质来说，情况正好相反。当你写下：

new RatiionalTrait{

val numerArg = expr1

val denomArg = expr2

}

表达式，expr1和expr2被作为匿名类初始化的一部分计算，但匿名类的初始化在RationalTrait之后。因此numerArg和denomArg的值在RationalTrait初始化期间还没有准备好（更为精确的说，选用任何值都将得到Int类型的默认值0）。对于之前的RationalTrait定义来说，这不是问题，因为特质的初始化没有使用numerArg或denomArg的值。然而对于下述代码展示的RationalTrait来说，这将成为一个问题：

trait RationalTrait{  
 val numerArg:Int  
 val denomArg:Int  
 require(denomArg != 0)  
 private val g = gcd(numerArg,denomArg)  
 val number = numerArg/g  
 val denom = denomArg/g  
 private def gcd(a:Int,b:Int):Int=  
 if(b==0) a else gcd(b,a % b)  
 override def toString: String = number + "/" + denom  
}

如果尝试使用某种分子和分母的表达式而非简单的字面量实例化这个特质，将得到异常信息：

scala> val x = 2

x: Int = 2

scala> new RationalTrait {

val numerArg = 1\*x

val denomArg = 2\*x

}

java.lang.IllegalArgumentException: requirement failed

at scala.Predef$.require(Predef.scala:221)

at RationalTrait$class.$init$(<console>:10)

at $anon$1.<init>(<console>:10)

….

抛出异常的原因在于，RationalTrait初始化的时候，denomArg仍然为它的默认值0，使得require调用失败。

**这个例子演示了类参数和抽象字段的初始化顺序并不一致。类参数在被传递给类构造器之前计算（除非参数是传名的）。相反，子类对于val定义的实现，是在超类完成了初始化之后执行的。**

现在你应该明白了为什么抽象val与参数的行为不一致，最好也应该了解一下对于这种情况我们可以做些什么。是否能定义出可以安全初始化的RationalTrati，而不用害怕由于未初始化的字段导致的错误？实际上，scala为这个问题提供了两种可选方案，预初始化字段和懒加载val。

**fields预初始化字段**

预初始化字段，可以在调用超类之前初始化子类的字段。操作的方式是把字段定义加上花括号，放在超类构造器调用之前，两者以with隔开。例如：

scala> new {

val numerArg = 1\*x

val denomArg = 2\*x

}with RationalTrait

res3: RationalTrait = 1/2

预初始化对象不仅局限于匿名类，也可以被用于对象及有名称的子类。例如：

scala> object twoThirds extends {

val numerArg = 2

val denomArg = 3

} with RationalTrait

defined module twoThirds

由于预初始化字段在超类构造器调用之前比初始化，因此它们的初始化器不能引用正被构造的对象。相应的结果是，如果有引用this的这种初始化器，那么实际指向的是包含了正被构造的类或对象的对象，而不是被构造对象本身。如下：

scala> new {

val numerArg = 1

val denomArg = this.numerArg\*2

}with RationalTrait

<console>:11: error: value numerArg is not a member of object $iw

val denomArg = this.numerArg\*2

^

例子中不能编译通过的原因在于this.numerArg引用是在包含new的对象中寻找numerArg字段（这个例子是指名为$iw的合成对象，解释器会把用户输出的语句放在这个对象中）。预初始化字段与类构造器参数在这方面再次取得一致）。

**懒加载val**

定义格式：把lazy修饰符前缀在val定义上，那么右侧的表达式将直到val第一次被使用的时候才计算。

例如，按照非lazy方式定义带有val的对象Demo：

scala> object Demo{

| val x = {println("initializing x");"done"}

| }

defined module Demo

scala> Demo

initializing x

res5: Demo.type = Demo$@2c99b437

scala> Demo.x

res6: String = done

可以发现，用到Demo的时候，它的字段x已经完成了初始化，也就是说，x的初始化成了Demo初始化的一部分。如果把x字段定义为lazy的话，情况就改变了：

scala> object Demo{

| lazy val x = {println("initializing x");"done"}

| }

defined module Demo

scala> Demo

res7: Demo.type = Demo$@222f808

scala> Demo.x

initializing x

res8: String = done

在这段代码中，字段x的初始化在第一次调用时才进行。

下面使用懒加载val，把之前RationalTrait进行重新实现：

trait LazyRationalTrait{  
 val numerArg:Int  
 val denomArg:Int  
 lazy val numer = numerArg/g  
 lazy val denom = denomArg/g  
 private lazy val g = {  
 require(denomArg!=0)  
 gcd(numerArg,denomArg)  
 }  
 private def gcd(a:Int,b:Int):Int=  
 if(b==0) a else gcd(b,a % b)  
  
 override def toString: String = numer +"/"+ denom  
}

scala> val x =2

x: Int = 2

scala> new LazyRationalTrait{

| val numerArg = 1\*x

| val denomArg = 2\*x

| }

res9: LazyRationalTrait = 1/2

懒加载val的初始化都是按需的，但初始化的顺序有时候需要考虑，尤其是存在副作用的情况下。

**懒加载函数式语言**

懒加载函数式语言的特点：每个值和参数都已懒加载的形式完成初始化。例如Haskell[SPJ02]

### （6）抽象类型

与其他所有抽象声明一样，抽象类型声明也是将在子类中具体定义的事物的占位符。这里，它是将在之后的类层次中定义的类型。因此上文的T是对在声明点尚不可知的类型的引用。不同的子类可以提供不同的T实现。

下面是一个能够以自然的方式体现抽象类型的常识性例子，下面是一个为动物饮食习惯建模的任务代码：

class Food  
abstract class Animal{  
 def eat(food:Food)  
}  
  
class Grass extends Food  
class Cow extends Animal{  
 override def eat(food:Grass) {}  
}

尝试编译上述代码时，会报以下错误：

Error:(10, 8) class Cow needs to be abstract, since method eat in class Animal of type (food: Food)Unit is not defined

(Note that Food does not match Grass: class Grass is a subclass of class Food, but method parameter types must match exactly.)

class Cow extends Animal{

^

Error:(11, 17) method eat overrides nothing.

Note: the super classes of class Cow contain the following, non final members named eat:

def eat(food: Food): Unit

override def eat(food:Grass) {}

^

根据以上错误内容可以发现scala编译器对类型检查是非常严格的。尤其是对子类的参数类型的检查。让我们看一下编译器为啥要这样严格，假设上述代码能通过编译，那么上述代码可以继续扩展：

class Food  
abstract class Animal{  
 def eat(food:Food)  
}  
  
class Grass extends Food  
class Cow extends Animal{  
 override def eat(food:Grass) {} //这将不能通过编译  
} //如果能的话...  
  
class Fish extends Food  
val bessy:Animal = new Cow  
bessy.eat(new Fish) //..你将能用鱼喂牛

若放松限制，程序将能通过编译，因为Cow是Animal，而Animal的eat方法可以接受任何种类的Food，甚至是Fish。但明显让牛吃鱼不是我们想要的。

你应该做的是采用更为精确的建模方式。Animal的确能吃Food，但Animal具体吃什么类型的Food取决于Animal。这可以用抽象类型干净的表示出来，代码如下：

class Food  
abstract class Animal{  
 type SuitableFood <: Food  
 def eat(food: SuitableFood)  
}

有了新的定义，Animal就可以只吃合适的食物了。不过到底什么食物合适，这并不在Animal类的层面决定。这也就是SuitableFood被建模为抽象类型的原因。类型具有上届约束，Food，表达式“<:Food”子句。说明任何（Animal子类中的）SuitableFood的具体实例化结果都必须是Food的子类。例如：

class Food  
abstract class Animal{  
 type SuitableFood <: Food  
 def eat(food: SuitableFood)  
}  
  
class Grass extends Food  
class Cow extends Animal{  
 type SuitableFood = Grass  
 def eat(food:Grass){}  
}

现在，如果尝试定义“吃—鱼—的—牛”的反例，将会得到如下编译错误：

scala> class Fish extends Food

defined class Fish

scala> val bessy:Animal = new Cow

bessy: Animal = Cow@19c9e8e8

scala> bessy eat (new Fish)

<console>:14: error: type mismatch;

found : Fish

required: bessy.SuitableFood

bessy eat (new Fish)

^

### （7）路径依赖类型

看一下最后的那条错误信息：eat方法所需的类型：beasy.SuitableFood。这个类型由对象引用（bessy）加对象的类型字段SuitableFood构成。因此这说明了Scala里类型可以做对象的成员。beasy.SuitableFood的意思是“SuitableFood类型是由bessy引用的对象的成员”，或者也可以理解为适合于beasy的食物类型。类似于bessy.SuitableFood的类型被称为路径依赖类型。这里的“路径”是指对象的引用。它可以是单名（如bessy）或更长的访问路径（如farm.barn.bessy.SuitableFood），其中的farm、barn及bessy都是指向这个对象的变量。

正如术语“路径依赖类型”所要表述的，类型依赖于路径：通常情况下，不同的路径将产生不同的类型。例如，假如你定义了DogFood类和Dog类：

class DogFood extends Food  
class Dog extends Animal{  
 type SuitableFood = DogFood  
 def eat(food:DogFood){}  
}

如果尝试把适合牛的饲料拿来喂狗，代码将无法通过编译：

scala> val bessy = new Cow

bessy: Cow = Cow@2a76535b

scala> val lassie = new Dog

lassie: Dog = Dog@4036e1ed

scala> lassie eat (new bessy.SuitableFood)

<console>:16: error: type mismatch;

found : Grass

required: DogFood

lassie eat (new bessy.SuitableFood)

^

问题在于传递给eat方法的SuitableFood对象的类型（bess.SuitableFood），不能匹配eat的参数类型，lassie.SuitableFood。但如果同样是Dog的话，他们的SuitableFood类型是一样的。

路径依赖类型会让我们想起java中的内部类（inner class）语法，但两者有决定性的差别：路径依赖类型表达了外在的对象。而内部类表达了外在的类。Java类型的内部类同样也可以在Scala中表达，不过写法不同。参考下面的Outer和Inner：

class Outer{

class Inner

}

Scala中，内部类的表达形式为**Outer#Inner**，而不是java的Outer.Inner。“.”语法保留给对象使用。例如，假设你实例化了类型Outer的两个对象：

val o1 = new Outer  
val o2 = new Outer

这里o1.Inner和o2.Inner是两个路径依赖类型（并且是不同的类型）。两种类型都符合于（即，继承于）更一般的类型Outer#Inner（该类型代表任意Outer类型外部对象的Inner类）。相反，o1.Inner类型是指特定（o1引用的）外部对象的Inner类。同理，类型o2.Inner指的是不同的特点（o2引用的）外部对象的Inner类。

Scala中与java中一样，内部类实例持有外部类实例的引用。这使得内部类可以执行类似于访问其外部类的成员的操作。因此**不能在没有通过某种方式指定外部实例的情况下实例化内部类**。实例化内部类的方式之**一是直接在外部类的方法体中完成**。这种情况下将使用当前外部类实例（this引用的对象）；方式之**二是使用路径依赖类型**。例如，因为类型o1.Inner命名了特定外部对象，所有你可以用如下方式实现实例化：

scala> new o1.Inner

res0: o1.Inner = Outer$Inner@7b5ac5d0

返回的内部对象将包含其外部对象的引用，即o1的对象引用。相反，类型Outer#Inner没有指明任何特定的Outer实例，因此不能创建它的实例：

scala> new Outer#Inner

<console>:9: error: Outer is not a legal prefix for a constructor

new Outer#Inner

^

### （8）枚举

路径依赖类型可以在scala中的枚举支持中找到。Scala中定义新的枚举，只要定义扩展类scala.Enumeration的对象即可。下面的例子定义了新的枚举Color：

object Color extends Enumeration{  
 val Red = Value  
 val Green = Value  
 val Blue = Value  
}

scala可以让你简化掉若干右侧一致的相连val或var定义。例如：

object Color extends Enumeration{  
 val Red,Green,Blue = Value  
}

这个对象定义了三个值：Color.Red、Color.Green、Color.Blue。你可以引用Color的全部内容：impor Color.\_

然后，简单写成Red、Green、Blue。这些值的类型是什么？Enumeration定义了内部类，名为Value，以及同名的无参方法Value返回该类的新对象。也就是诸如Color.Red的值类型是Color.Value。而Color.Value也正是定义在对象Color中的所有枚举值的类型。它是路径依赖类型，其中Color是路径，Value是依赖类型。这里很重要的一点是，它是全新的类型，与其他所有的类型都不一样。尤其是，即使定义另外的枚举，例如：

object Direction extends Enumeration{  
 val North,East,South,West = Value  
}

那么Direction.Value和Color.Value不同，因为两种类型的路径部分不同。

Scala中，可以使用Value方法不同的重载变体把名称与枚举值联系起来：

object Direction extends Enumeration{  
 val North = Value("North")  
 val East = Value("East")  
 val South = Value("South")  
 val West = Value("West")  
}

可以使用foreach遍历枚举的所有值，或是使用带map、flatMap和filter的for表达式：

scala> for( d <- Direction.values) print(d + " ")

North East South West

枚举值从0开始计数，可以获取枚举值的id：

scala> Direction.East.id

res8: Int = 1

也可以通过id获取枚举值：

scala> Direction(1)

res9: Direction.Value = East

### （9）案例研究：货币

现在要设计一个Currency类，它的实例可以代表不同货币类型（美元、人民币、欧元等）的一笔金额，而且还需要一些货币方面的计算，比如，相同的货币可以进行相加，可以乘以代表利率的因子。

下面是Currency的第一个版本代码：

//第一版本

abstract class Currency {  
 val amount :Long  
 val designation:String  
 override def toString = amount +" " + designation  
 def +(that:Currency) = {}  
 def \*(x:Double) :Currency = {...}  
}

可以在这段代码基础上对Currency进行扩展，创建Dollar和Euro，例如：

在scala中，如果在类定义的时候有些东西还未知，可以在类中定义它的抽象。这对值和类型都有用。在货币的例子中，加法确切的参数和结果类型未知，因此可以使用抽象类型。这将产生如下的AbstractCurrency类草案：

//第二版本

abstract class AbstractCurrency{  
 type Currency <: AbstractCurrency  
 val amount: Long  
 val designation: String  
 override def toString: String = amount +" "+ designation  
 def +(that:Currency):Currency = {...}  
 def \*(x:Double):Currency = {...}  
}

可以在AbstractCurrency的基础上扩展Dollar，例如：

abstract class Dollar extends AbstractMethodError{  
 type Currency = Dollar  
 val designation = "USD"  
}

继续定义+方法，如果这样子写：

def +(that:Currency):Currency = new Currency{  
 val amount = this.amount + that.amount  
}

编译时，编译器将报以下错误：

Error:(17, 39) class type required but AbstractCurrency.this.Currency found

def +(that:Currency):Currency = new Currency{

^

**Scala对抽象类型的一种限制是你既不能创建抽象类型的实例，也不能把抽象类型当作其他类的超类型。**

然后，可以使用工厂方法绕过这个限制。代之以直接创建抽象类型的实例，可以声明抽象方法间接做到这点。于是，在抽象类型被修改为某种具体类型的地方，你还需要提供某种工厂方法的具体实现。对于AbstractCurrency类来说，应该看上去是下面的样子：

abstract class AbstractCurrency{

type Currency <: AbstractCurrency //抽象类型

def make(amount:Long):Currency //工厂方法

... //类的其余部分

}

这段代码可以工作，但存在的问题是，工厂方法make存在AbstractCurrency类中不够合理，存在代码重复的现象。解决办法是，把抽象类型和工厂方法移出来，放到另外一个类中，我们称之为CurrencyZone：

abstract class CurrencyZone{  
 type Currency <: AbstractCurrency  
 def make(x:Long):Currency  
 abstract class AbstractCurrency{  
 val amount:Long  
 val desigantion:String  
 override def toString: String = amount + " " + desigantion  
 def +(that:Currency):Currency =  
 make(this.amount + that.amount)  
 def \*(x:Double):Currency =  
 make((this.amount\*x).toLong)  
 }  
}

CurrencyZone的具体例子US，可以采用以下的方法定义：

object US extends CurrencyZone{  
 abstract class Dollar extends AbstractCurrency{  
 val desigantion: String = "USD"  
 }  
 type Currency = Dollar  
 def make(x: Long): US.Currency = new Dollar {val amount = x}  
}

目前为止的设计，每种货币仅用一个测量单位：美元、欧元或日元。然而，大多数货币都有子单位，例如有美元和美分。建模美分最直接的方式是US.Currency的amount字段代表美分而不是美元。要想转回（代表）美元，可以向CurrencyZone类引入新的字段CurrencyUnit，以包含此货币一个标准单位的金额。

class CurrencyZone{

……

val CurrencyUnit: Currency

}

US可以定义Cent、Dollar的数量及CurrentUnit，例如：

object US extends CurrencyZone{  
 abstract class Dollar extends AbstractCurrency{  
 val desigantion: String = "USD"  
 }  
 type Currency = Dollar  
  
 def make(cents: Long): US.Currency = new Dollar {val amount = cents}  
 val Cent = make(1)  
 val Dollar = make(100)  
 val CurrencyUnit = Dollar  
}

Currency类的toString方法也要考虑子单位的情况，比如，10美元和23美分的总和应该打印成小数：10.23USD。可以按一下方式进行重写：

override def toString: String =  
 ((amount.toDouble / CurrencyUnit.amount.toDouble)  
 formatted ("%." + decimals(CurrencyUnit.amount)+"f")  
 + " " + desigantion)  
  
private def decimals(n:Long):Int=  
 if( n == 1) 0 else 1 + decimals(n / 10)

同样，可以定义其他货币区域：

object Europe extends CurrencyZone{  
 abstract class Euro extends AbstractCurrency{  
 val desigantion: String = "EUR"  
 }  
 type Currency = Euro  
 def make(cents: Long): Euro = new Euro {  
 val amount: Long = cents  
 }  
 val Cent = make(1)  
 val Euro = make(100)  
 val CurrencyUnit = Euro  
}  
  
object Japan extends CurrencyZone{  
 abstract class Yen extends AbstractCurrency{  
 val desigantion: String = "JPX"  
 }  
 type Currency = Yen  
 def make(cents: Long): Japan.Currency = new Yen{ val amount = cents}  
  
 val Yen = make(1)  
 val CurrencyUnit = Yen  
}

另一项改进在于你可以在模型中添加货币转换的特征。首先，编写Converter对象，以包含适用的货币间汇率，如下：

object Converter{  
 var exchangeRate = Map{  
 "USD" -> Map("USD" -> 1.0,"EUR" -> 0.7596,  
 "JPY" -> 1.211,"CHF" -> 1.223),  
 "EUR" -> Map("USD" -> 1.316,"EUR" -> 1.0,  
 "JPY" -> 1.594, "CHF" -> 1.623),  
 "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,  
 "JPY" -> 1.0, "CHF" -> 1.018),  
 "CHF" -> Map("USD" -> 0.8108,"EUR" -> 0.6160,  
 "JPY" -> 0.982,"CHF" -> 1.0)  
 }   
}

然后，添加from方法到Currency类中，它可以把指定源货币转换为当前Currency对象：

def from(other:CurrencyZone#AbstractCurrency):Currency =  
 make(Math.round(  
 other.amount.toDouble\*Converter.exchangeRate  
 (other.desigantion)(this.desigantion)  
 ))

from方法可以带任意货币为参数。这个参数被表达为形式上的参数类型（CurrencyZone#AbstractCurrency），以指明other传递的参数必须是某种未知但任意CurrencyZone的AbstractCurrency类型。它通过把other货币的金额乘以other与当前货之间的汇率产生结果。

下面就可以使用这些类，进行汇率转换：

val a1 = Japan.Yen from US.Dollar\*100  
println(a1)  
val a2 = Europe.Euro from a1  
println(a2)  
val a3 = US.Dollar from a2  
println(a3)  
  
val a4 = a3 + US.Dollar\*100  
println(a4)

输出结果为：

12110 JPY

75.95 EUR

99.95 USD

199.95 USD

Scala为面向对象的抽象提供了系统化的并且是非常通用的支持。它不仅使你能在方法上使用抽象，还能在值、变量和类型上使用。设计类的时候，把所有未知的东西都变成抽象成员。

## 隐式转换和参数

### （1）隐式转换

Scala里的一个核心集合特质是RandomAccessSeq[T]，它描述了建立在类型T的元素上的随机访问序列。RandomAccessSeq有大多数数组或列表的实用方法，要定义新的随机访问序列，唯一的工作就是扩展RandomAccessSeq特质。你只需完成特质中的两个抽象方法的定义：length和apply，特质中的其他方法也就获得了。

如果你定义新的类可以这么做，但是对于已经存在的类该怎么办呢？scala提供了隐式操作的方法来解决这个问题。例如，需要定义从String到实际为RandomAccessSeq子类的适配类隐式转换。

implicit def stringWrapper(s:String) =  
 new RandomAccessSeq[Char]{  
 def length = s.length  
 def apply(i:Int) = s.charAt(i)  
 }

隐式转换只是普通的方法，唯一特殊的地方是前面要加implicti修饰符。可以显示的调用这个方法把String转换为RandomAccessSeq子类：

scala> stringWrapper("abc123") exists (\_.isDigit)

res0: Boolean = true

也可以省略这个转换，隐式的调用：

scala> "abc123" exists (\_.isDigit)

res0: Boolean = true

隐式操作与c#的扩展方法类似，它也能让你在现存的类中添加新方法。但是，隐式操作远比扩展方法简洁，因为所有的转换都在自动包含新增加的方法。

**隐式转换的另一个优点是它们支持目标类型的转换。**所谓目标类型，是指在代码的某个点需要的类型。例如 ：

def printWithSpace(seq: RandomAccessSeq[Char]) =

seq mkString " "

因为String对象可以隐式转换为RandomAccessSeq，因此可以把字符串传递给该方法：

scala> printWithSpace("xyz")

res2: String = x y z

这条语句与下面的等价：

scala> printWithSpace(stringWrapper("xyz"))

res2: String = x y z

### （2）隐式操作规则

隐式定义是指编译器为了修正类型错误而允许插入到程序中的定义。例如，如果x+y不能通过类型检查，那么编译器可能会把它变为conver(x) + y，这里的cover是某个可用的隐私转换。

隐私转换由以下通用规则掌控：

**标记规则：标记为implicit的定义才是可用的。**implicit关键字可以标记任何的变量、函数、或对象定义。

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

**作用域规则：插入的隐私转换必须以单一标识符的形式处于作用域中，或与转换的源或目标类型关联在一起。**Scala编译器将仅考虑处于作用域之内的隐私转换。从而，为了使隐私转换可用，你必须以某种方式把它带入作用域之内。此外，隐式转换必须以单一标识符的形式（除了一种情况）进入作用域。编译器将不能插入someVariable.convert的转换。

“单一标识符”规则有一个例外。编译器还将在源类型或转换的期望目标类型的伴生对象中寻找隐式定义。例如，如果你尝试传递Dollar对象到入参为Euro的方法，源类型是Dollar，而目标类型是Euro。因此你可以把从Dollar到Euro的隐私转换打包到Dollar或Euro类的伴生对象中。例如：

object Dollar{

implicit def dollarToEuro(x:Dollar):Euro = ...

}

class Dollar {...}

这里的dollarToEuro转换被称为是类型Dollar相关联的。

作用域规则有助于模块推断。

**无歧视规则：隐式转换唯有不存在其他可插入转换的前提下才能插入。**如果编译器有两个可选方法修正x+y，比方说既可使用convert1(x)+y，又可使用convert2(x)+y，那么它会报错并拒绝在两者之间做出选择，解决的办法是移除其中一个转换，只保留一个转换。

**单一调用规则：只会尝试一个隐式操作。编译器将不会把x+y重写成convert1(convert2(x)) +y。**

**显示操作先行规则：若编写的代码类型检查无误，则不会尝试任何隐私操作。**编译器不会改变已经正常工作的代码。这条规则的推论就是你始终可以用显示内容替换隐式标识符。

**命名隐式转换：隐式转换可以任意命名。**仅需要考虑两种情况：你是否需要在方法应用中明确写明，以及决定哪个隐式转换在程序的任何地方都有效。

**隐式操作在哪里尝试。**scala语言中能用到的隐私操作有三个地方：**转换为期望类型、指定（方法）调用者的转换、隐式参数。**

### （3）隐式转换为期望类型

编译器看到了x，但需要y，就会检查从x到的隐式转换函数。例如，通常双精度不能用作整数，因为这样会丢失精度：

scala> val i:Int = 3.5

<console>:7: error: type mismatch;

found : Double(3.5)

required: Int

val i:Int = 3.5

^

可以定义隐式转换消除这一个障碍：

scala> implicit def doubleToInt(x: Double) = x.toInt

warning: there were 1 feature warning(s); re-run with -feature for details

doubleToInt: (x: Double)Int

scala> val i:Int = 3.5

i: Int = 3

这种转换把Double转成了Int，导致精度降低。更合理的方式是反过来，从某些受限制的类型转为更一般的类型。例如，scala.Predef对象，定义了把较小的数值转换成较大的类型的隐式转换。

### （4）转换（方法调用的）接收者

隐式转换同样还应用于方法调用的接受者，也就是方法调用的对象。这种类型的隐式转换由两种主要用途。第一种，接收者转换使得新的类可以更为平滑地集成到现存类层级中。以及第二种，它们支持在语言中编写域特定语言（DSL）。

假设你编写了obj.doIt，但obj并没有名为doIt的成员。编译器将在放弃之前尝试插入转换代码。这里，转换需要应用于接收者obj。编译器会认为obj的期望“类型”应该“拥有名为doIt成员”。这个“有doIt”的类型不是普通的scala类型，不过它在概念上是存在的，并且是编译器在这种情况下插入隐式转换的理由。

**与新类型的交互操作**

可以把现存类型的实例当作新类型的实例使用。例如，之前的Rational类：

class Rational(n:Int,d:Int){

...

def +(that:Rational):Rational = ...

def +(that:Int): Rational = ...

}

Raitonal类有两个重载的+号方法变体，分别带Rational和Int为参数，即实现了有理数和有理数、有理数和整数的加法。

scala> val oneHalf = new Rational(1,2)

oneHalf: Rational = 1/2

scala> oneHalf + oneHalf

res1: Rational = 1/1

scala> oneHalf + 1

res2: Rational = 3/2

但如果是1+oneHalf表达式呢？因为整数没有合适的调用Rational的+法，所有会报错：

scala> 1+ oneHalf

<console>:12: error: overloaded method value + with alternatives:

(x: Double)Double <and>

(x: Float)Float <and>

(x: Long)Long <and>

(x: Int)Int <and>

(x: Char)Int <and>

(x: Short)Int <and>

(x: Byte)Int <and>

(x: String)String

cannot be applied to (Rational)

1+ oneHalf

^

为了允许这种类型的混合数学运算，需要定义从Int到Rational的隐式转换：

scala> implicit def intToRational(x:Int) =

| new Rational(x,1)

warning: there were 1 feature warning(s); re-run with -feature for details

intToRational: (x: Int)Rational

scala> 1 + oneHalf

res4: Rational = 3/2

**模拟新的语法**

隐式转换的另一个主要用途是模拟添加新的语法。例如，创建Map对象的语法

Map(1 -> “one”,2 –> “two”,3 -> “three”)

这里的“->”是定义在scala.Predef中的类ArrowAssoc的方法。这个序文还定义了从Any到ArrowAssoc的隐式转换。在编写1 -> “one”的时候，编译器会插入1到ArrowAssoc的转换以便找到->方法。以下是相关定义：

final class ArrowAssoc[A](val \_\_leftOfArrow: A) extends AnyVal {  
 @deprecated("Use `\_\_leftOfArrow` instead", "2.10.0")  
 def x = \_\_leftOfArrow  
  
 @inline def -> [B](y: B): Tuple2[A, B] = Tuple2(\_\_leftOfArrow, y)  
 def →[B](y: B): Tuple2[A, B] = ->(y)  
}

@inline implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)

### （5）隐式参数

可以让编译器插入隐式操作的最后的地方是通过参数列表。编译器有时会用someCall(a)(b)替代someCall(a)，或者用someClass(a)(b)替代someClass(a)，从而通过添加缺失的参数列表以满足函数调用。被提供的是完整的最后一节柯里化参数，而不仅是最后的参数。例如，若someCall缺失的最后参数列表带三个参数，那么编译器将会用someCall(a)(b,c,d)替代someCall(a)。对于这种用法来说，不仅被插入的标识符，如（b,c,d）中的b，c和d，必须在定义的时候被标记为implicit，连someCall或someClass定义的最后一个参数列表都必须被标记为implicit。

以下是一个简单的例子：

class PrefereedPrompt(val preference:String)  
object Greeter{  
 def greet(name:String)(implicit prompt: PrefereedPrompt): Unit ={  
 println("Welcome , "+ name+".The system is ready.")  
 println(prompt.preference)  
 }  
}

greet后面的参数被标示为implicit，也就是说它可以被隐式提供，不过仍然可以显示提供，例如：

scala> val bobsPromp = new PrefereedPrompt("relax> ")

bobsPromp: PrefereedPrompt = PrefereedPrompt@58f59add

scala> Greeter.greet("Bob")(bobsPromp)

Welcome , Bob.The system is ready.

relax>

为了让编译器隐式提供参数，必须首先定义期望类型的变量，在这里是PrefereedPrompt。例如，你可以在参数设置对象中这样做：

object JoesPrefs{  
 implicit val prompt = new PrefereedPrompt("Yes, master> ")  
}

这里的val也要被标示为implicit，而且要**以单一标识符处于作用域之内**，如果不满足，则不能用它，例如：

scala> Greeter.greet("Joe")

<console>:14: error: could not find implicit value for parameter prompt: Prefere

edPrompt

Greeter.greet("Joe")

^

通过引用把它带入作用域，就可以补充缺失的参数列表了：

scala> import JoesPrefs.\_

import JoesPrefs.\_

scala> Greeter.greet("Joe")

Welcome , Joe.The system is ready.

Yes, master>

注意：**implicit应用于全体参数列表，而不是单独的参数**。例如：下面定义了两个参数：

class PrefereedPrompt(val preference:String)  
class PrefereedDrink(val preference:String)  
  
object Greeter{  
 def greet(name:String)(implicit prompt: PrefereedPrompt,drink: PrefereedDrink): Unit ={  
 println("Welcome , "+ name+".The system is ready.")  
 print("But while you work, ")  
 println("why not enjoy a cup of "+ drink.preference + "?")  
 println(prompt.preference)  
 }  
}  
  
object JoesPrefs{  
 implicit val prompt = new PrefereedPrompt("Yes, master> ")  
 implicit val drink = new PrefereedDrink("tea")  
}

然后，通过import把两个隐式val带入作用域：

scala> import JoesPrefs.\_

import JoesPrefs.\_

scala> Greeter.greet("Joe")

Welcome , Joe.The system is ready.

But while you work, why not enjoy a cup of tea?

Yes, master>

同样，也可以显示应用：

scala> Greeter.greet("Joe")(prompt,drink)

Welcome , Joe.The system is ready.

But while you work, why not enjoy a cup of tea?

Yes, master>

上面的这个例子中有一件事需要注意，就是我们没有使用String作为prompt或drink的类型，尽管最终他们每一个通过preference字段提供出来的实际还是String对象。因为编译器选择隐式参数的方式是通过匹配参数类型与作用域内的值类型，隐式参数往往是很“稀少”或者很“特殊”的类型，以至于不会被碰巧匹配。

关于隐式参数的另一件需要知道的事情是，或许它们最常见的用法是为了提供前面（在隐式参数列表之前）显示标明的参数列表的类型信息，这与Haskell的 类型类很类似。例如 ，考虑下面的maxListUpBound函数，它可以返回传入列表的最大元素：

def maxListUpBound[T <: Ordered[T]](elements:List[T]) :T =  
 elements match {  
 case List() => throw new IllegalArgumentException("empty list!")  
 case List(x) => x  
 case x :: rest =>  
 val maxRest = maxListUpBound(rest)  
 if(x > maxRest) x  
 else maxRest  
 }

这里的maxListUpBound函数所需的参数为List[T]，这里的T通过上届指定必须是Ordered[T]的子类型。使用这种方法的弱点是你不能把这个函数用于元素类型并非已经是Ordered子类型的列表。例如，你不可以使用maxListUpBound函数发现整数列表的最大值，因为Int类型不是Ordered[Int]的子类型。

另外，组织maxListUpBound的更为一般的方式应该是除List[T]入参之外，还需要一个分离的第二入参，一个能够把T转换为Ordered[T]的函数。以下代码说明了这一方式，第二参数（orderer）被放在分离的入参列表中并被标记为implicit：

def maxListImpParm[T](elements:List[T])(implicit orderer: T => Ordered[T]): T =  
 elements match{  
 case List() => throw new IllegalArgumentException("empty list!")  
 case List(x) => x  
 case x :: rest =>  
 val maxRest = maxListImpParm(rest)(orderer)  
 if(orderer(x) > maxRest) x  
 else maxRest  
 }

上述代码中，orderer参数被用来描述T对象的排序。在maxListImpParm的方法体中，这种排序被用在两个地方：maxListImpParm的递归调用，以及检查是否列表头大于列表剩余元素的最大值的if表达式。

maxListImpParm函数是使用隐式参数提供较早显式标明的参数列表的类型信息的例子。更明确的说，是指隐式参数order（类型为T => Ordered[T]），提供了更多的T类型的信息——例子里是指如何做T对象的排序。类型T由参数elements的类型（List[T]）标明，它出现在较早的参数列表中。因为elements必须永远在maxListImpParm函数的每次调用中显示提供，所以编译器将在编译时知道T，并且从而判决是否T => Ordered[T]类型的隐私定义存在于作用域中。如果是，它将隐私地传入第二参数列表orderer。

这种模式是如此的普通以至于标准scala库为许多通用类型提供了隐私的“orderer”方法。因此可以把maxListImpParm方法用在许多类型上：

scala> maxListImpParm(List(1,5,10,3))

res0: Int = 10

scala> maxListImpParm(List(1.5,5,2,10.7,3.14159))

res2: Double = 10.7

scala> maxListImpParm(List("one","two","three"))

res3: String = two

在第一个例子中，编译器为Int对象插入了orderer对象；第二个例子中，是Double对象；第三个例子中，是String对象。

**隐式参数的样式规则**

样式规则是说，最好对隐式参数的类型使用自定义命名的类型。例如，前面例子中的prompt和drink类型并非String，而分别是PrederredPrompt和PrefereedDrink。作为反例，假设maxListImpParm函数就直接编写为使用如下的类型签名：

def maxListPoorStyle[T](elements: List[T])

(implicit orderer: (T,T) => Boolean) : T

不过为了使用函数的这个版本，调用者必须提供类型为（T,T）=> Boolean的orderer参数。这种类型过于平凡，包含了任何以两个对象产生Boolean的函数。它根本没能说明这个类型的目的；可以是相等性判断、小于判断、大于判断或者根本不清楚的什么东西。

在maxListImpParm例子中，采用了较好的样式。它使用了类型为T=>Ordered[T]的orderer参数。类型中的单词Ordered确切指明了隐式参数的用途：为了排序T类型的元素。这个orderer类型更为清晰，因此也扫清了把这种类型的隐私转换加入到标准中的障碍。相反，可以设想一下如果把(T,T) => Boolean类型的隐私操作加入标准库，而编译器又将它撒播在他人的代码中，必然引发混乱。对你来说代码通过编译并正常运行就结束了，但这段代码却会相当随意地发起对每条目的测试。

**因此有了样式规则：至少用一个角色确定的名称为隐式参数的类型命名。**

### （6）视界

如果把implicit用在参数上，编译器不仅会尝试用隐私值补足这个参数，还会把这个参数当作可用的隐私操作而使用于方法体中。因此上述代码例子中的orderer的两处应用都可以被省略。例如：

def maxList[T](elements:List[T])(implicit orderer: T => Ordered[T]):T =  
 elements match {  
 case List() => throw new IllegalArgumentException("empty list!")  
 case List(x) => x  
 case x :: rest =>  
 val maxRest = maxList(rest) //隐式转换出（orderer)  
 if( x > maxRest) x //隐式转换出（orderer(x)）  
 else maxRest  
 }

当编译器检查上述代码时，会发现类型不能匹配。例如，T类型的x不存在>方法，因此x>maxRest不起作用。然而编译器不会立刻停止。它会首先寻找能够修复这段代码的隐式转换。例子中，它将注意到orderer可用，因此可以把代码转换为orderer(x) > maxRest。并且同样处理了表达式maxList(rest)，把它转换为maxList(rest)(orderer)。经过了这两段隐式操作，方法现在能够通过类型检查了。

仔细看一下maxList，方法的文本中没有一处提到orderer参数的地方。所有多orderer的使用都是隐式的。这种代码模式在scala中非常普遍。隐式参数只是被用来做转换，因此它本身也可被隐式地使用。

现在，因为参数名从未被显式调用，所以名称可以任意指定。例如，只要不改变maxList的方法体，对于改变参数名称来说，方法的行为没有任何改变：

def maxList[T](elements: List[T])(implicit converter: T => Ordered[T]): T =

//相同的方法体

即使定义如下也没问题：

Def maxList[T](elements: List[T])(implicit iceCream: T => Ordered[T]): T =

//相同的方法体

由于这种模式很普遍，scala可以让你省略这个参数的名称并使用视界缩短方法头。通过使用视界，可以写出如下的代码：

def maxList[T <% Ordered[T]](elements: List[T]):T =  
 elements match {  
 case List() => throw new IllegalArgumentException("empty list!")  
 case List(x) => x  
 case x :: rest =>  
 val maxRest = maxList(rest)  
 if(x > maxRest) x  
 else maxRest  
 }

可以认为“T <% Ordered[T]”是在说，“任何的T都好，只要T能被当做Ordered[T]即可。”这与说T是Ordered[T]不一样，这是上届（T <: Ordered[T]）所有表达的意思。例如，尽管Int类不是Ordered[Int]的子类型，但只要Int到Ordered[T]的隐式转换可用，仍然把List[Int]传递给maxList方法。此外，如果类型T碰巧已经是Ordered[T]类型，也仍然可以把List[T]传递给maxList。编译器将调用声明在Predef中的隐式鉴别函数：

implicit def identity[A](x : A) : A = x

这里，转换什么也没做；它只是简单的把传入的参数再返还出来。

**视界与上届**

上届，T <：Ordered[T]，指定T是Ordered[T]类型的。视界，T <% Ordered[T]，指定T可以被当作Ordered[T]。在书写形式上，两者唯一的区别就是符号的不同，上届符号用<:，而视界符号用<%。

### （7）隐式操作调试

隐式操作是scala的非常强大的特征，但有时候很难用对也很难调试。

有时候你或许好奇为什么编译器不能发现你认为应该可以用的隐式转换。这种情况下，把这些转换明确写出来有助于发现问题。如果这样还有错误信息，就应该知道为什么编译器不能应用隐式操作了。例如，假设错误地把stringWrapper当作是String对象到List对象，而不是RandomAccessSeq对象的转换。你可能会不明白为什么下面的代码不能工作：

scala> val chars: List[Char] = "xyz"

<console>:7: error: type mismatch;

found : String("xyz")

required: List[Char]

val chars: List[Char] = "xyz"

^

这时，你可以明确写出stringWrapper转换，来发现到底错在哪里，这样，你就能发现错误的原因：stringWrapper返回了错误的类型。另一方面，也有可能由于显式地插入了转换使得错误消失。这种情况下，你就知道有其他的什么规则（诸如作用域规则）正在阻止隐式转换的使用。

调试程序的时候，如果能看到编译器正在插入的隐式转换可能会有一些帮助。编译器的-Xprint:typer选项可以用于这一目的。如果你使用这一选项运行scalac，编译器就将展示给你在类型检查器添加了所有的隐式转换之后的代码看起来的样子。例如下述代码：

object Mocha extends App{  
 class PreferredDrink(val preference:String)  
 implicit val pref = new PreferredDrink("mocha")  
  
 def enjoy(name:String)(implicit drink:PreferredDrink): Unit ={  
 print("Welcome, "+ name)  
 print(". Enjoy a ")  
 print(drink.preference)  
 println("!")  
 }  
  
 enjoy("reader")  
}

上述代码中的最后一句：

enjoy("reader")

在编译器中将被转换为：

Mocha.this.enjoy(“reader”)(Mocha.this.pref)

## 实现列表

List类是对scala的类型系统（具体来说，是它的泛型化概念）的精妙应用。

### （1）List原理

列表并非scala的“内建”语言结构，它们由List抽象类定义在scala包之中，并且包含了两个子类：:: 和 Nil。下面对List类作快速浏览：

package scala

abstract class List[+T] {

List类是抽象的，不能直接实例化。

List类具有类型参数T，类型参数前的T特指列表是协变的。例如：

scala> val xs = List(1,2,3)

xs: List[Int] = List(1, 2, 3)

scala> val ys:List[Any] = xs

ys: List[Any] = List(1, 2, 3)

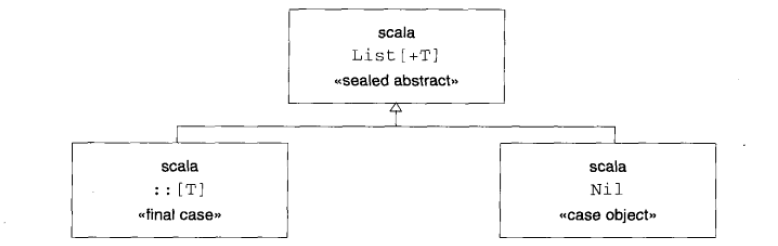
所有的列表操作都可以用以下三种基本方法定义：

def isEmpty:Boolean

def head:T

def tail: List[T]

这三种方法在List类中都是抽象的，它们被定义在子对象Nil和子类::中。List的层级如下：



**Nil对象**

Nil对象定义了空列表。Nil对象继承自List[Nothing]类型。因为协变的存在，意味着Nil可以兼容任何类型的List实例。

case object Nil extends List[Nothing]{

override def isEmpty = true

def head: Nothing =

throw New NoSuchElementException("head of empty list")

def tail: List[Nothing] =

throw new NoSuchElementException("tail of empty list")

}

**：：类**

::类，发音为”“cons”，意为构造，代表非空列表。如此命名是为了能够使用中缀::支持模式匹配。前面我们了解到，模式中的每个中缀操作都会被当作入参为操作符参数的操作符构造器调用。因此，模式 x :: xs 被当作样本类：：的构造器调用，:: (x , xs )。以下是::类的定义：

final case class :: [T] (hd: T, tl: List[T]) extends List[T] {

def head = hd

def tail = tl

override def isEmpty: Boolean = false

}

::类的实现同样很直接，hd和tl代表着列表的头和尾部。实际上，这个模式可以被缩略为由参数直接实现超类List的head和tail方法，与下面的代码定义等价：

final case class :: [T (head: T,tail: List[T]) extends List[T] {

override def isEmpty:Boolean =false

}

这样做之所以可行，是因为所有的样本类参数同时隐含着是类的字段。Scala中，允许用字段实现抽象的无参数方法，如head和tail。

**更多的类方法**

所有其他的List方法都可以用这三种基本方法编写。例如：

def length: Int =

if (isEmpty) 0 else 1 + tail.length

或：

def drop(n:Int) :List[T] =

if(isEmpty) Nil

else if(n <= 0 ) this

else tail.drop(n -1)

或：

def map[U] (f: T => U): List[U] =

if (isEmpty) Nil

else f(head) :: tail.map(f)

**列表构建**

列表的::和:::构建方法都很特殊，因为他们都以冒号结束，所以实际是绑定在右操作数上的。即，诸如x::xs这样的操作会被看作是xs.::(x)，而不是x.::(xs)的方法调用。实际上，x.::(xs)没有实际意义，因为x是列表元素的类型，可以是任意的，所以我们不能假设这种类型具有::方法。

也由于这个原因，::方法应该输入元素值并生成新列表。那么需要的元素值类型应该是什么呢？你或许会惯性的认为，它应该与列表的元素类型相同，但实际上没有这么局限。为了说明理由，可以参考下面的这种类层级关系：

abstract class Fruit

class Apple extends Fruit

class Orange extends Fruit

然后，构建水果列表：

scala> val apples = new Apple :: Nil

apples: List[Apple] = List(Apple@1f0f9da5)

scala> val fruits = new Orange :: apples

fruits: List[Fruit] = List(Orange@73eb454e, Apple@1f0f9da5)

fruits的例子说明，可以往列表中添加不同类型的元素。列表fruits的元素类型变为了Fruit，是原始列表类型Apple和新添加元素类型Orange的超类型。这种弹性是通过下面的方式定义::方法而获得的：

def :: [U >: T] (x: U) : List[U] = new scala.::(x,this)

请注意，方法本身是多态的——它带有名为U的类型参数，U通过[U >: T]的方式限定为列表元素T的超类型。被添加的元素必须是U类型，返回的结果为List[U]。

实际上，带有下界T的::多态定义不只是为了方便，这也是为了以正确类型完成List类定义的必然结果。这是因为List被定义为协变的。假设一段时间里我们把::定义为如下形式：

//（不会成功的）脑力实验

def ::(x: T): List[T] = new scala.::(x, this)

在这里，方法参数被认为是逆变位置，因此，上面定义里的列表元素类型T处于逆变位置。但这样一来，List就不能被声明为与T保持协变了。因此，下界[U >: T]实在是一石二鸟之技：一方面移除了类型化错误，另一方面也使得::方法在使用的时候更具有弹性。

列表连接方法:::定义的方式与::类似，如下：

def :::[U >: T](prefix: List[U]): List[U] =

if(prefix.isEmpty) this

else prefix.head :: prefix.tail ::: this

注意，上述代码中：

prefix.head :: prefix.tail ::: this

等价于（因为::和:::是右关联的）

prefix.head ::(prefix.tail ::: this)

等价于（因为::绑定在右操作元上）

(prefix.tail ::: this).:: prefix.head

等价于（因为:::绑定在右操作元上）

this.:::(prefix.tial).::(prefix.head)

P337

## 包对象

到目前为止，我们看到的添加到包的都是类型，Trait和单例对象(Object)。这些都是指包的定级层次定义的类型。Scala的定级层次除了可以定义类，Trait，Object之外，其它可以在类，Trait,Object内部定义的类型，也都可以直接定义在包中，比如一些通用的函数，变量，你都可以直接定义在包中。

在Scala中，可以把这些函数或方法放在一个称为“包对象”中。每个包只有一个包对象，任何放在包对象的类型都可以认为是包自身的成员。

例如：

//in file bobsdelights/package.scala

package object bobsdelights{

def showFruit(fruit: Fruit){

import fruit.\_

println(name + "s are " + color)

}

}

//in file PrintMenu.scala

package printmenu

import bobsdelights.Fruits

import bobsdelights.showFruit

object PrintMenu{

def main(args:Array[String]){

for(fruit <- Fruits.menu){

showFruit(fruit)

}

}

}

本例中对象PrintMenu可以引入包对象中定义的函数showFruit，方法和引入一个类定义一样，也是通过import 语句。

包对象通常被编译为package.class，其包名为定义的包。所有按照惯例一般包对象定义放在package.scala中，比如上面的包对象可以放在bobsdelights目录下的package.scala中。

**可见性和伴生对象**

Java里，静态成员和实例成员属于同一个类，因此访问修饰符可以统一地应用在它们之上。在scala里没有静态成员，作为替代，可以拥有包含成员的单例的伴生对象。例如，下述代码中，Rocket对象是Rocket类的伴生：

class Rocket{

import Rocket.fuel

private def canGoHomeAgain = fuel > 20

}

object Rocket{

private def fuel = 10

def chooseStrategy(rocket:Rocket){

if(rocket.canGoHomeAgain)

goHome()

else

pickAStar()

}

def goHome(){}

def pickAStar(){}

}

对于私有或保护访问来说，scala的访问规则给予了伴生对象和类一些特权。类的所有的访问权限都对伴生对象开发，反过来也是如此。具体的说，就是对象可以访问所有它的伴生类的私有成员，就好像类也可以访问伴生它的对象的所有私有成员一样。

Protected static成员对于scala和java在修饰符的相似性方面来说是一个例外。Java中c类的保护静态成员可以被c的所有子类访问。相反，伴生对象的protected成员没有意义，因为单例对象没有任何子类。

## Case类

Scala支持case类的概念。case类也是正规的类，暴露其构造器参数，并通过模式匹配提供递归解构机制。 以下是由抽象超类Term和三个具体case类Var，Fun，和App组成的类层次的例子。

abstract class Term

case class Var(name:String) extends Term

case class Fun(arg:String,body:Term) extends Term

case class App(f:Term,v:Term) extends Term

这个类层次可以用来表现无类型lambda算子。为了便于case类实例的构建，Scala不需要使用new原语。只要简单地把类名用作函数即可。举例如下：

Fun("x",Fun("y",App(Var("x"),Var("y")))

case类的构造器参数被当作公开值并可以直接被访问。

val x=Var("x")

Console.println(x.name)

对于每个case类，Scala编译器将产生实现了结构相等性检查的equals方法和一个toString方法。举例如下：

val x1 = Var("x")

val x2 = Var("x")

val y1 = Var("y")

println("" + x1 + " == " + x2 + " => " + (x1 == x2))

println("" + x1 + " == " + y1 + " => " + (x1 == y1))

将打印：

Var(x) == Var(x) => true

Var(x) == Var(y) => false

只有在分解数据结构的时候用到了模式匹配的情况下，定义case类才是有意义的。以下的对象为我们的lambda算子的展示定义了漂亮的打印函数：

object TermTest extends Application {

  def printTerm(term: Term) {

    term match {

      case Var(n) =>

        print(n)

      case Fun(x, b) =>

        print("^" + x + ".")

        printTerm(b)

      case App(f, v) =>

        Console.print("(")

        printTerm(f)

        print(" ")

        printTerm(v)

        print(")")

    }

  }

  def isIdentityFun(term: Term): Boolean = term match {

    case Fun(x, Var(y)) if x == y => true

    case \_ => false

  }

  val id = Fun("x", Var("x"))

  val t = Fun("x", Fun("y", App(Var("x"), Var("y"))))

  printTerm(t)

  println

  println(isIdentityFun(id))

  println(isIdentityFun(t))

}

我们的例子里，函数print表达为开始于match关键字的，由一系列的case Pattern => Body子句组成的模式匹配语句。

上面的程序还定义了函数isIdentityFun，用来检查是否给定的term对应于一个简单的身份函数。这个例子使用了深度模式和护卫语句。在使用给定值匹配了模式之后，护卫语句（定义于关键字if之后）被执行评估。如果返回true，匹配成功；反之，则匹配失败，并尝试下一个模式。

## [**Zip函数族详解**](http://www.iteblog.com/archives/1225)

**（1）zip函数**

将传进来的两个参数中相应位置上的元素组成一个pair数组。如果其中一个参数元素比较长，那么多余的参数会被删掉；

**（2）zipAll函数**

和上面的zip函数类似，但是如果其中一个元素个数比较少，那么将用默认的元素填充。

**（3）zipped函数**

he zipped method on tuples generalizes several common operations to work on multiple lists.

scala> val values = List.range(1, 5)

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

scala> (values, values).zipped toMap

res34: scala.collection.immutable.Map[Int,Int] = Map(1 -> 1, 2 -> 2, 3 -> 3, 4 -> 4)

scala> val sumOfSquares = (values, values).zipped map (\_ \* \_) sum

sumOfSquares: Int = 30

**（4）zipWithIndex函数**

将元素和其所在的下标组成一个pair。

**（5）unzip函数**

可以将一个元组的列表转变成一个列表的元组

## 偏函数

**对函数定义域的一个子集进行定义的函数。 scala中用scala.PartialFunction[-T, +S]类来表示。**

T是其接受的函数类型，S是其返回的结果类型。偏函数最大的特点就是它只接受和处理其参数定义域的一个子集，而对于这个子集之外的参数则抛出运行时异常。这与Case语句的特性非常契合，因为我们在使用case语句是，常常是匹配一组具体的模式，最后用“\_”来代表剩余的模式。如果一一组case语句没有涵盖所有的情况，那么这组case语句就可以被看做是一个偏函数。

val signal: PartialFunction[Int, Int] = {

    case x if x > 1 => 1

    case x if x < -1 => -1

}

这个signal所引用的函数除了0值外，对所有整数都定义了相应的操作。 signal(0) 会抛出异常，因此使用前最好先signal.isDefinedAt(0)判断一下。 偏函数主要用于这样一种场景：对某些值现在还无法给出具体的操作（即需求还不明朗），也有可能存在几种处理方式（视乎具体的需求）；我们可以先对需求明确的部分进行定义，比如上述除了0外的所有整数域，然后根据具体情况补充对其他域的定义。

val composed\_signal: PartialFunction[Int,Int] = signal.orElse{

case 0 => 0

}

或者对定义域进行一定的偏移（假如需求做了变更,  1 为无效的点）

val new\_signal: Function1[Int, Int] = signal.compose{

  case x => x  - 1

}

可以用andThen将两个相关的偏函数串接起来

val another\_signal: PartialFunction[Int, Int] = {

   case 0 =>  0

   case x if x > 0 => x - 1

   case x if x < 0 => x + 1

}

val then\_signal =  another\_signal andThen  signal

这里的then\_signal 剔除了-1, 0, 1三个点的定义。

## Option

Option 类型或 Option[T]，并不重视描述。它是一个具有两个子类 Some[T] 和 None 的泛型类，用来表示 “无值” 的可能性，而不需要语言类型系统大费周折地支持这个概念。实际上，使用 Option[T] 类型可以使问题更加清晰。

**避免null使用**

大多数语言都有一个特殊的关键字或者对象来表示一个对象引用的是“无”，在Java，它是null。在Java 里，null 是一个关键字，不是一个对象，所以对它调用任何方法都是非法的。但是这对语言设计者来说是一件令人疑惑的选择。为什么要在程序员希望返回一个对象的时候返回一个关键字呢？

**Scala的Option类型**

为了让所有东西都是对象的目标更加一致，也为了遵循函数式编程的习惯，Scala鼓励你在变量和函数返回值可能不会引用任何值的时候使用Option类型。在没有值的时候，使用None，这是Option的一个子类。如果有值可以引用，就使用Some来包含这个值。Some也是Option的子类。

None被声明为一个对象，而不是一个类，因为我们只需要它的一个实例。这样，它多少有点像null关键字，但它却是一个实实在在的，有方法的对象。

**应用例子**

Option类型的值通常作为Scala集合类型（List,Map等）操作的返回类型。比如Map的get方法：

scala> val capitals = Map("France"->"Paris", "Japan"->"Tokyo", "China"->"Beijing")

capitals: scala.collection.immutable.Map[String,String] = Map(France -> Paris, Japan -> Tokyo, China -> Beijing)

scala> capitals get "France"

res0: Option[String] = Some(Paris)

scala> capitals get "North Pole"

res1: Option[String] = None

Option有两个子类别，Some和None。当程序回传Some的时候，代表这个函式成功地给了你一个String，而你可以透过get()函数拿到那个String，如果程序返回的是None，则代表没有字符串可以给你。

在返回None，也就是没有String给你的时候，如果你还硬要调用get()来取得 String 的话，Scala一样是会抛出一个NoSuchElementException异常给你的。

我们也可以选用另外一个方法，getOrElse。这个方法在这个Option是Some的实例时返回对应的值，而在是None的实例时返回传入的参数。换句话说，传入getOrElse的参数实际上是默认返回值。

scala> capitals get "North Pole" get

warning: there was one feature warning; re-run with -feature for details

java.util.NoSuchElementException: None.get

at scala.None$.get(Option.scala:347)

at scala.None$.get(Option.scala:345)

... 33 elided

scala> capitals get "France" get

warning: there was one feature warning; re-run with -feature for details

res3: String = Paris

scala> (capitals get "North Pole") getOrElse "Oops"

res7: String = Oops

scala> capitals get "France" getOrElse "Oops"

res8: String = Paris

通过模式匹配分离可选值，如果匹配的值是Some的话，将Some里的值抽出赋给x变量：

def showCapital(x: Option[String]) = x match {

case Some(s) => s

case None => "?"

}

**提示**

Scala程序使用Option非常频繁，在Java中使用null来表示空值，代码中很多地方都要添加null关键字检测，不然很容易出现NullPointException。因此Java程序需要关心那些变量可能是null,而这些变量出现null的可能性很低，但一但出现，很难查出为什么出现NullPointerException。

Scala的Option类型可以避免这种情况，因此Scala应用推荐使用Option类型来代表一些可选值。使用Option类型，读者一眼就可以看出这种类型的值可能为None。

实际上，多亏Scala的静态类型，你并不能错误地尝试在一个可能为null的值上调用方法。虽然在Java中这是个很容易犯的错误，它在Scala却通不过编译，这是因为Java中没有检查变量是否为null的编程作为变成Scala中的类型错误（不能将Option[String]当做String来使用）。所以，Option的使用极强地鼓励了更加弹性的编程习惯。

**详解Option[T]**

在Scala里Option[T]实际上是一个容器，就像数组或是List一样，你可以把他看成是一个可能有零到一个元素的List。

当你的Option里面有东西的时候，这个List的长度是1（也就是 Some），而当你的Option里没有东西的时候，它的长度是0（也就是 None）。

**for循环**

如果我们把Option当成一般的List来用，并且用一个for循环来走访这个Option的时候，如果Option是None，那这个for循环里的程序代码自然不会执行，于是我们就达到了「不用检查Option是否为None这件事。

**map操作**

在函数式编程中有一个核心的概念之一是转换，所以大部份支持函数式编程语言，都支持一种叫map()的动作，这个动作是可以帮你把某个容器的内容，套上一些动作之后，变成另一个新的容器。

现在我们考虑如何用Option的map方法实现length: xxx的输出形式：

a. 先算出 Option 容器内字符串的长度

b. 然后在长度前面加上 "length: " 字样

c. 最后把容器走访一次，印出容器内的东西

scala> value1.map(\_.length).map("length: " + \_).foreach(println)

length: 6

scala> value1.map("length: " + \_.length).foreach(println)

length: 6

透过这样「转换」的方法，我们一样可以达成想要的效果，而且同样不用去做「是否为 None」的判断。

## BigInt

大整数类型

BigInt(10000000000000000000000000) // 报错

BigInt("10000000000000000000000000") // scala.math.BigInt = 10000000000000000000000000

## 函数式编程思想

**（1）函数是头等值**

函数式语言中，函数也是值，与整数或字符串处于同一个地位。函数可以当作参数传递给其他函数，可以当做结果从函数中返回或保存在变量里。可以在函数中定义其他函数，就好像在函数里定义整数一样。还可以定义匿名函数，并随意插入到代码的任何地方。

**（2）程序的操作应该把输入值映射为输出值而不是就地修改数据。**

比如java和scala中对字符串的实现，如果修改字符串对象中的一个字符，会产生一个新的、不同于原字符串对象的一个新对象。

不可变数据结构是函数式语言的基石。scala库在java api之上定义了更多的不可变数据类型。例如，scala有不可变的列表、元组、映射表和集。

**方法不应该有任何副作用（side effect）。**

**函数式语言鼓励使用不可变数据结构和指称透明的方法**（方法与其所在的环境交流的唯一方式应该是获得参数和返回结果，例如string.replace方法，称为referentially transparent).

## Scala是静态类型的

静态类型系统可以根据保存和计算的值的类型认定变量和表达式类型。scala以java的内嵌类型系统为基础，允许使用泛型参数化类型，用交集（intersection)组合类型及抽象类型（asbtract type）隐藏类型细节。

（1）可检验属性

（2）安全的重构

（3）文档

## Scala的根源

统一对象模型

通用嵌套的思想

函数式编程

## Actor模型

Actor模型是并发编程中比较常见的一种模型。Erlang、Scala等提供了原生的Actor模型。

Actor，可以看作是一个个独立的实体，他们之间是毫无关联的。通过消息来通信。多个消息通过建立消息队列进行存储。

Actor模型=数据+行为+消息

Actor模型内部的状态由自己的行为维护，外部线程不能直接调用对象的行为，必须通过消息才能激发行为，这样就保证Actor内部数据只有被自己修改。

## 函数式风格

每个有用的程序都会有某种形式的副作用，否则就不可能向程序之外提供什么有价值的东西。提倡无副作用的方法是为了尽量设计出没有副作用代码的程序。这种方式的好处之一是可以有助于程序更容易测试。

## 从文件里读取文本行

引用包： Scala.io.Source

Import scala.io.Source

Val textFile = “G:/temp/test.txt”

For(line <- Source.fromFile(textFile).getLines){

Print(line.length +” “ + line)

}

## 类和对象

如果没有发现任何显示的返回语句，Scala方法将返回方法中最后一次得到的值。

方法的推荐风格是尽量避免使用返回语句，尤其是多条返回语句。代之以把每个方法当作是创建返回值的表达式。

假如某个方法仅计算单个结果表达式，则可以去掉花括号。如果结果表达式很短，甚至可以把它放在def的同一行里。

class ChecksumAccumulator{

private var sum=0

def add(b:Byte):Unit=sum += b

def checksum():Int=~(sum & 0xFF) + 1

}

对于像ChecksumAccumulator的add方法（结果类型为Unit的方法）来说，执行的目的就是为了它的副作用。**通常定义副作用为能够改变方法以外的某种状态或执行I/O活动的方法。**比如，add这个例子里，副作用就是sum被重新赋值了。

Class ChecksumAccumulatro{

private var sum=0

def add(b:Byte):{ sum += b}

def checksum():Int =~(sum & 0xFF) + 1

**注意**：**scala编译器可以把任何类型转换为Unit。**

def f():Unit = “this String gets lost”

函数f声明了结果类型Unit，因此String被转变为Unit。

def g() {“this String gets lost too”}

函数g由于没有加等号，返回类型转变为Unit。

def h() = {“this String gets returned!”}

函数h的返回类型为String。

## 分号推断

Scala程序里，语句末尾的分号是可以选择的。如果一行里只要一条语句，则可以不加分号，若是一行里包含多条语句，分号则是必须的。

**除非以下情况的任何一种成立，否行尾被认作是一个分号：**

1. 疑问行由一个不能合法作为语句结尾的字结束，如句点或中缀操作符。
2. 下一行开始于不能作为语句开始的词。
3. 行结束于扩号（…）或方框[…]内部，因为这些符号不可能容纳多个语句。

## Scala程序

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。

Scala的每个源文件都隐含了对包java.lang、包scala，以及单例对象Predef的成员引用。

Scala对源文件的命名没有硬性规定，类名可以与源文件的名称不一致。

脚本：必须以结果表达式结束。

**脚本的编译与执行：**

（1）编译

scalac Summer.scala

或者：

fsc Summer.scala //多个文件以空格隔开

（2）执行

Scala Summer //后面可以添加参数，空格隔开

## App 特质

Scala提供了特质App（2.9.0之前为Application），可以减少一些输入。

使用方法：首先在单例对象后面加上extends App。然后代之以main方法，把想要执行的代码直接放在单例对象的花括号之间。

object FallWinterSpringSummer extends App{  
 println("App test!")  
}

implicit val ord = implicitly[Ordering[Int]].reverse