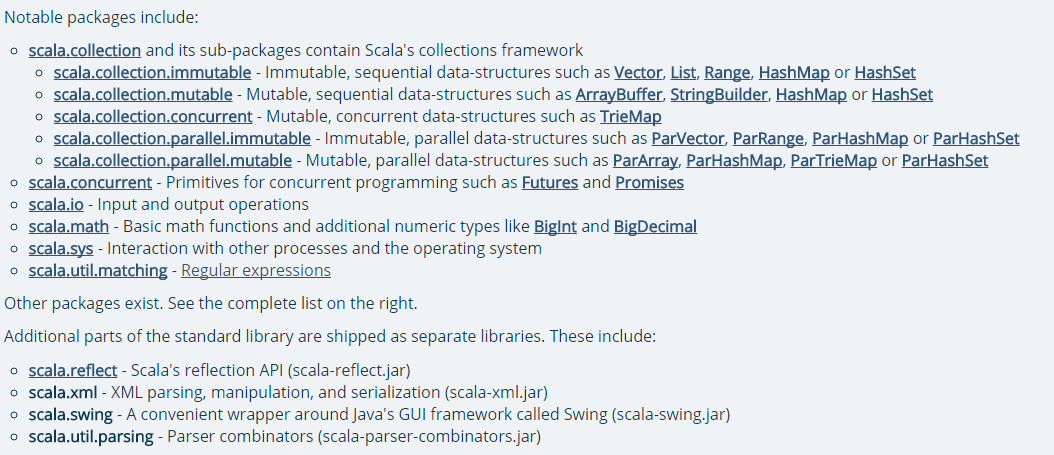
**1、Scala数据类型**

Scala的数据类型全部相同于Java中，具有相同的内存占用和精度。以下表是有关可在Scala中所有的数据类型的细节：



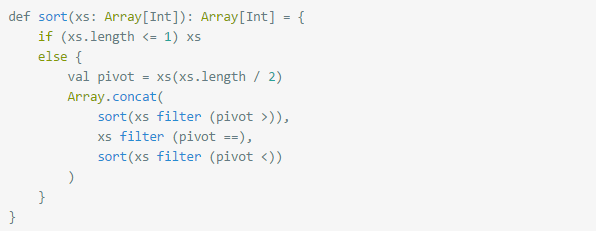


上面列出的所有的数据类型是对象。没有原始的类型，如Java中。这意味着可以调用int，long等上的方法

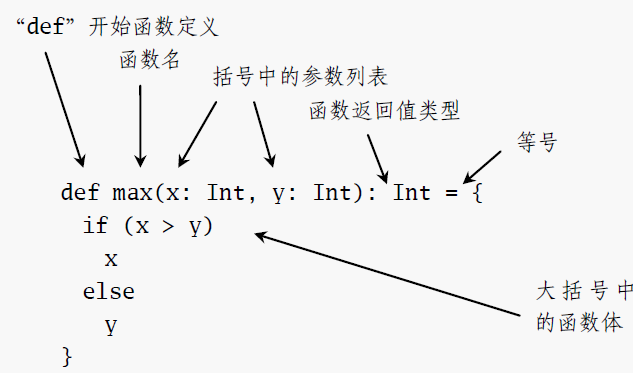


Scala有两种变量，val和var。***val类似于Java里的final变量。一旦初始化了，val就不能再赋值了***。与之对应的，var如同Java里面的非final变量。var可以在它生命周期中被多次赋值。

**2、函数知识**

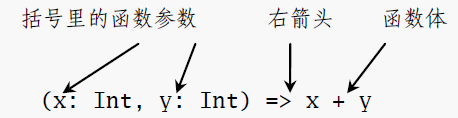


函数的基本构成：



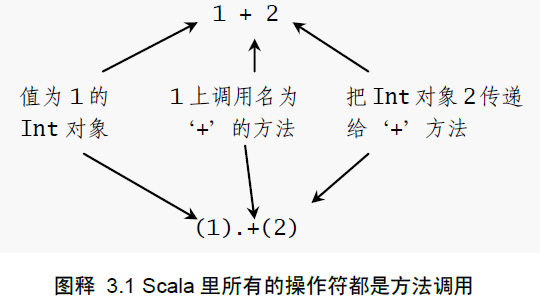
***Unit的结果类型指的是函数没有返回有用的值***。Scala的Unit类型比较接近Java的void类型，而且实际上Java里每一个返回void的方法都被映射为Scala里返回Unit的方法。因此结果类型为Unit的方法，仅仅是为了它们的副作用而运行。

函数文本格式：



我们还没有介绍filter，不过可以想象**xs filter (pivot >)** 部分从xs中就是得到大于pivot的数列。想象**xs filter someting就相当于xs.filter(something)**。同理，**x > y就是x.>(y)**。而**pivot >就相当于pivot.>()**。一个缺省参数简化编程表达的应用。

从技术上讲，Scala没有操作符重载，因为它根本没有传统意义上的操作符。取而代之的是，***诸如+，-，\*和/这样的字符可以用来做方法名***。因此，当第一步里你在Scala解释器里输入1 + 2，你实际上正在Int对象1上调用一个名为+的方法，并把2当作参数传给它。如图3.1所示，你也可以使用传统的方法调用语法***把1 + 2替代写成(1).+(2)***。



**1）函数的参数**

参数可以是可变长的。



参数可以给与名字：



使用参数名调用，可以不考虑参数顺序

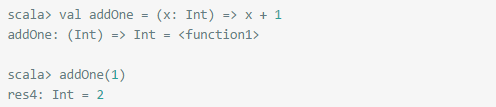


参数可以有默认值。这个例子默认打印到Console



**2）匿名函数**

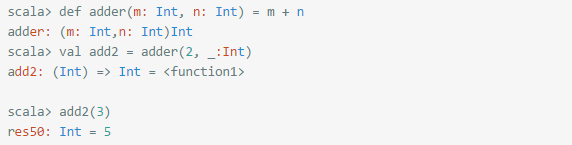
函数可以***没有名称，可以直接赋值***。如果***函数有多行表达式组成，就加上大括号***。这一点对匿名函数同样适用。（类似Java中的lambda表达式）



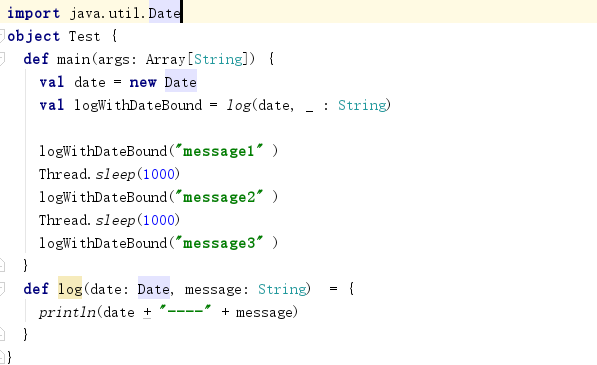
注意到***Scala中一切都是对象。所以addOne是一个对象***。同时它又是个包含一个参数的函数。此时，函数和对象得到了统一。

**3）偏函数应用 （Partial Application）**

偏函数应用解决这样的问题：如果我们有***函数是多个参数的，我们希望能固定其中某几个参数的值***。在Scala中，我们可以这么做



或者



结果：



**4）函数柯里化(Currying)**

偏函数应用不是什么新东西，几乎所有编程语言中都可以简单地实现。但更简洁的实现是**柯里化**。你可以这样定义柯里化函数



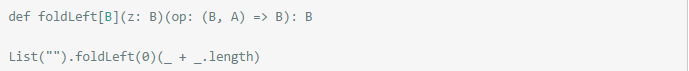
也可以***对一个已有的函数进行转换***。例如之前的adder函数



函数柯里化依赖匿名函数，***把一个多参数的函数转化为多个单参数的函数连接在一起***。这样一来，语意的表达就更加丰富而直观，函数组合也更加灵活。

**A. 类型推断**

一个应用场景就是**类型推断**。Scala类库中的foldLeft 就是这样一个例子。



如果使用普通的多参数函数定义

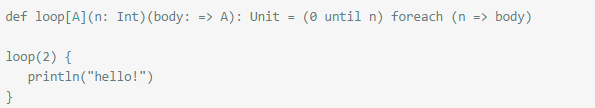


那么，你在使用时就必须显示地表明类型。



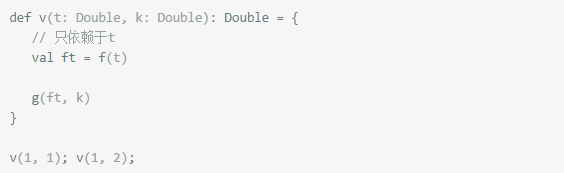
**B. 简化API**

另一个currying的应用是简化API。在下面的例子中，你可以使用大括号来传递body，使得API调用层次分明。

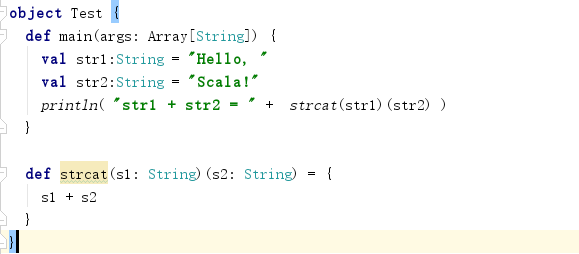


**C. 多阶段介绍**

如果你的多参数函数的计算实际上是分步骤的，某个步骤只依赖某个参数，比如下面的例子。那么**柯里化会更加简单**。

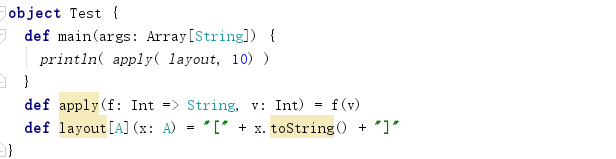


可以根据柯里函数需求定义两个以上的参数。让我们以一个完整的例子来说明柯里的概念：



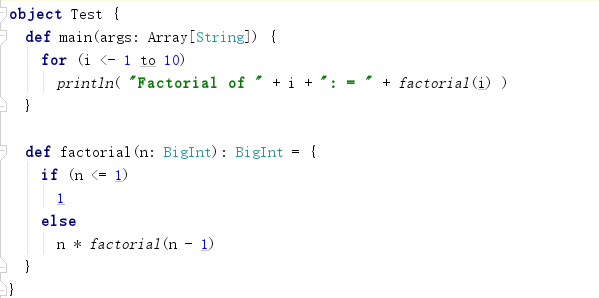
**5）高阶函数**

Scala允许高阶函数的定义。这些都是***采取其他函数参数，或它的结果是一个功能的函数***。例如在下面的代码，适用于apply()函数将另一个函数f和v值并应用函数f到v：



**6）递归函数**

递归起着纯粹函数式编程很大的作用，***Scala支持递归功能非常好***。递归是指的函数可以反复调用自身。以下是递归，用于计算阶乘数的一个很好的例子：

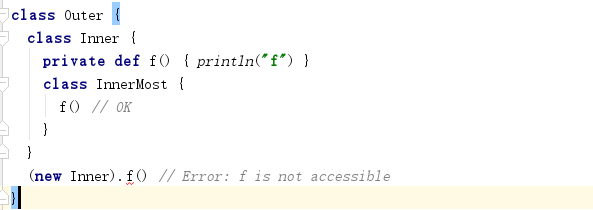


**7）Scala访问修饰符**

包，类或对象的成员可以标记访问修饰符*private和protected*，如果我们不使用这两种关键字，那么访问将被默认设置为public。这些修饰限制为成员的代码的某些区域访问。要使用访问修饰符，包括它的包，类或对象的成员定义的关键字，我们将在下一节中看到。

**A. 私有成员**

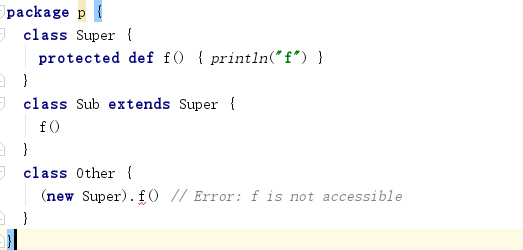
私有成员只能看到里面包含的成员定义的类或对象。下面是一个例子：



在Scala中，访问 (new Inner).f() 是非法的，因为***f被声明为private内部类并且访问不是在内部类内***。与此相反，到f第一接入类最内层是确定的，因为该访问包含在类内的主体。 Java将允许这两种访问，因为它可以让其内部类的外部类访问私有成员。

**B. 保护成员**

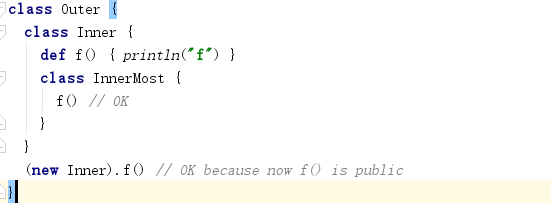
受保护的成员是从***该成员定义的类的子类才能访问***。下面是一个例子：



类分给 f 访问是正常的，因为f被声明为受保护的超类和子类是超级的子类。相比之下，访问 f 在其他是不允许的，因为其他没有从超级继承。***在Java中，后者访问将仍然允许的，因为其他的是在同一个包子***。

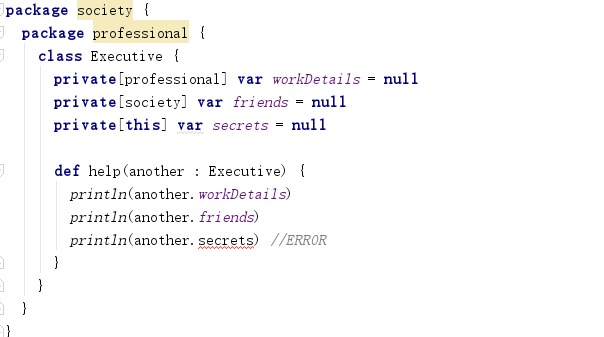
**C. 公有成员**

未标示私有或受保护的每一个成员是公开的。不需要明确使用修饰符public。这样的成员可以从任何地方访问。下面是一个例子：



**D. 保护范围**

Scala中的访问修饰符可以**增加使用修饰符**。形式：***private[X]或protected[X]的修饰符意味着访问私有或受保护“达到”X，其中X代表了一些封闭的包，类或单个对象***。考虑下面的例子：

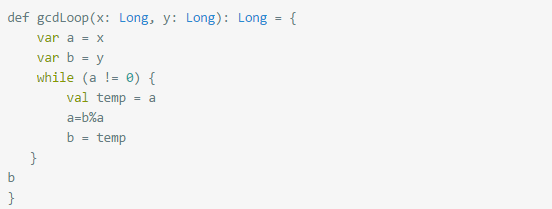


注意，上面的例子中以下几点：

* 变量workDetails将可对任何一类封闭包professional范围内。
* 变量friends 将可对任何一类封闭包society中。
* 变量secrets 将可只在实例方法隐含的对象（this）。

**3、简单循环**

while

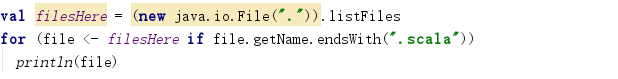


简单for循环

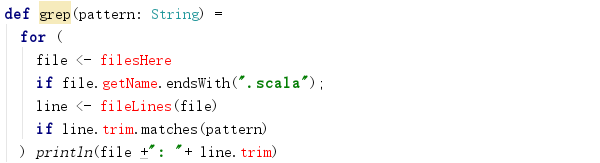


**高级的For循环**

加入Filter

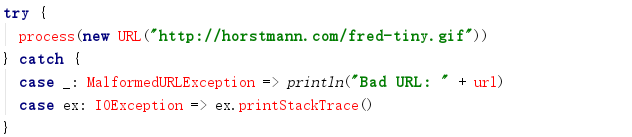


多个生成器

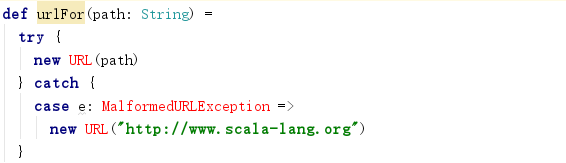


**4、异常**

***Scala基本沿用Java的异常。但是没有checked exception***，捕捉异常可以*使用灵活的模式匹配*



和其他Scala控制结构一样，try-catch-finally是有返回值的。下面的例子教你如何在异常发生时返回一个默认值。注意的时***finally通常用来做资源回收，不要依赖finally返回值***。



**5、数组**

数组是最基本的数据结构。通常的语法或数据结构书都会先介绍数组，而后再介绍集合，链表，树，哈希表等等。我们也不例外。

**1）定长数组**

最简单的数组创建如下，记住***方括号在Scala中用做泛型，相当于<>在Java中作用***。

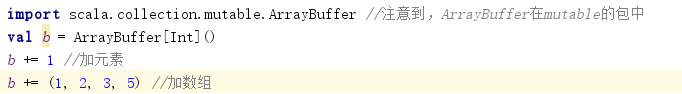


更简洁的数组创建像这样：val s = Array("Hello", "World")

省略new的初始化，并且利用类型推断，无需指明类型。***数组元素的访问使用（）***。s(0) = "Goodbye"。s不是val吗？为什么可以改变呢？***s这里存的是一个数组的reference，reference不能变，但是其内容是可以改变的***。另外说一句，遗憾的是，从C时代开始，Index从0开始这一违反人类正常思维的“习惯”一直沿用。

**2）变长数组**

***Scala中的ArrayBuffer和Java中的ArrayList类似***，长度可以改变。



**3）数组遍历**

可以使用Index



或者

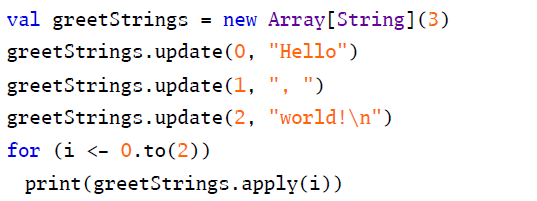


要强调的是，***对一个数组（或者其他集合）进行遍历以一种命令式的编程***（imperative programming）。在Scala中，我们往往避免使用，而是使用**函数式的方法**。强调“做什么”而不是怎么做。以后还会遇到很多例子，你会逐步感受到它们的区别。

**4）通用算法**

Scala提供了很多内建通用算法，作用于数组，以及以后会谈到的其他数据类型如集合。比如sum，count，max，mKString等等。***mKString是一个很有用的方法***。参见下面的例子。



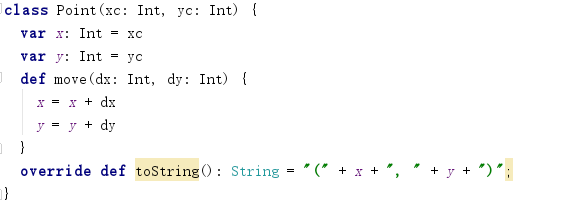




**6、类和对象**

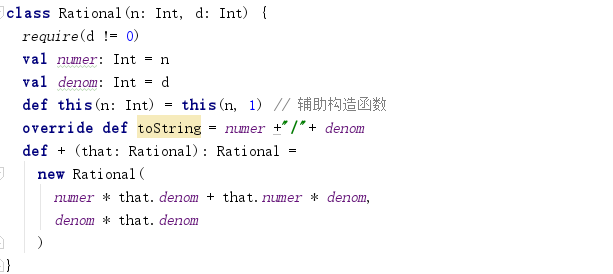
**1）定义**

类的定义例子如下。



move方法有两个参数，没有返回值(这里***Scala缺省了Unit，类似于Java中void的返回***）。toString是一个重载的方法，必须使用override。

***类的定义中可以有多个构造参数***。这比C++或者Java的构造函数更简洁。另外，***Scala中只有一个主要构造函数，其他都是辅助构造函数。辅助构造函数必须调用主构造函数***。看下面一个分数的例子，辅助构造函数默认分母为1。



这种比较严格的规定使得Scala类的构造和继承更加简单。在讲继承的时候会具体讲。

**2）隐式转换**

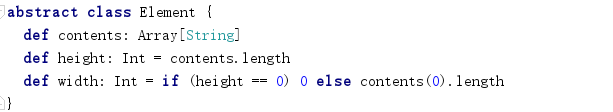
如果我们想要创建一个分数r，然后计算2+r。2是个Int，没有以Rational为参数+的方法。应该怎么实现呢***？Scala提供了隐式转换的方法***。下面的例子就将一个Int转换成了Rational。这样2+r就可以调用了。



隐式转换功能强大，在开发特定领域语言（DSL）时很有用。当然使用是也需要十分谨慎，不但要追求程序的简洁，还要确保其可读性。

**3）无参数的方法**

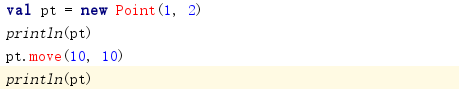
在定义类的方法时，***如果该方法没有参数，也不会改变类的状态，那么方法定义的括号可以省去***。



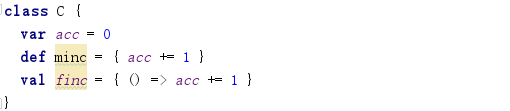
这样做符合统一化访问的原则。你以后可以使用**方法（def height）或者成员(val height)**。而客户端无需改变。

**4）类的使用**

通过类可以创建对象的实例，调用类方法。

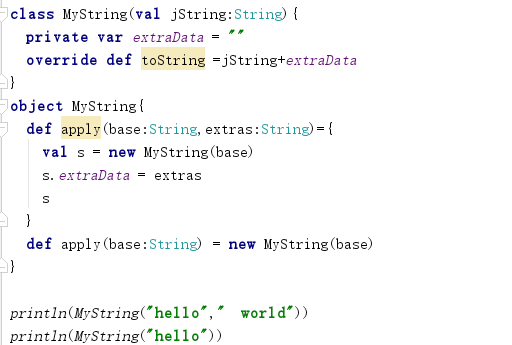


***函数和方法在很大程度上是可以互换的***。在下面的例子中，调用minc和finc的方式完全一样，效果也基本相同。其细微的差别你现在也不需要关心。

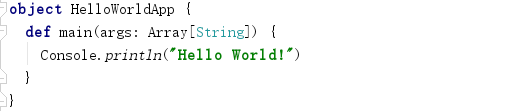


**5）单例（singleton）对象**

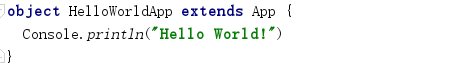
Scala中***没有static成员。取而代之的是单例（singleton）对象***。单例对象的定义和类定义相同，除了使用**关键字object**。我们常常*使用相同的名字定义一个类和一个单例对象*。这样的对象是***该类的伴生对象，可以访问该类的成员和方法***。通过这样的方法（和apply结合使用），我们可以实现类型的factory，从而去除了外部new的使用。



为了运行一个Scala程序，必须定义这样一个单例对象作为main的入口。

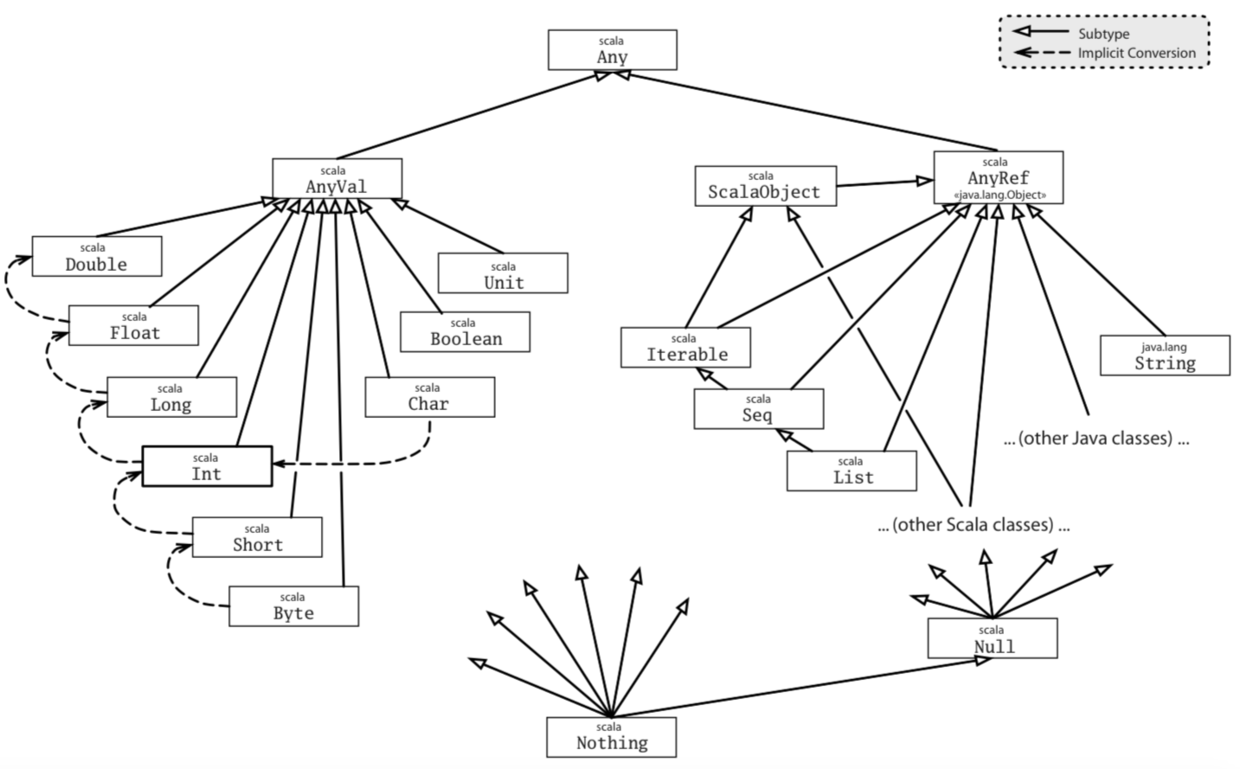


Scala也提供了App让你继承，以达到相同的效果。



**6）类结构**

Scala的***所有类的组织结构***如下图所示。



***所有的类都继承了Any。AnyVal是所有内建值类的父类，而其他类的父类是AnyRef***。AnyRef相当于Java中java.lang.Object。所以你定义Scala类又是Java类。***最下面的Nothing和Null是为了统一一些特殊情况***。比如error在Scala中的定义，它说明***函数不会返回，而是会抛出异常，这和返回Unit是不同的***。



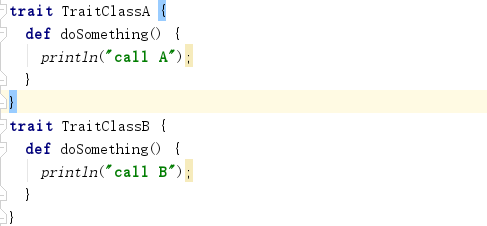
例外，图中的虚线箭头使用了前面提到的隐式转换的技术。***Scala中继承，抽象类，重载等概念和其他面向对象的语言基本相关***，除了特质（trait）。我们会把特质单独拿出来讲。

**7、特质trait**

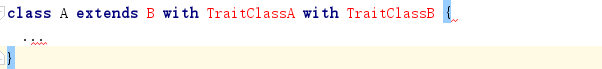
类似Java中的接口性质，更加灵活

**1）更灵活的接口**

Scala的特质（trait）定义和class几乎相同，只是trait不能的构造不能包含参数。

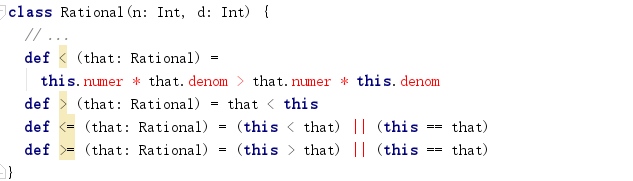


在使用trait的时候，可以用extend或者with。***with的方式成为混入（mixin）。用户可以with任意多个特质***。这会让你想起接口（Interface）。

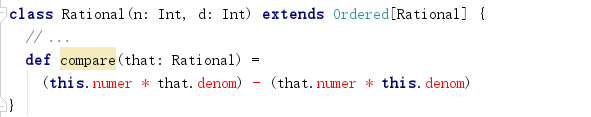


Trait可以看做是更灵活，更丰富的接口。和Java中接口不同的是，***Trait可以包含成员和方法的实现***。这样做的好处***需要某个trait就拿来用，而不需要重复实现接口。trait和混入的类是正交的***。

举例说明，在Java中，***定义一个Rational类就需要重写多个比较的方法***。



而如果使用trait，这些比较的方法就可以封装在Ordered（类似Comparable接口）这个特质中（注意到这里使用到了类型参数[Rational]）。以后所有需要实现比较特质的类，它只要实现compare方法。

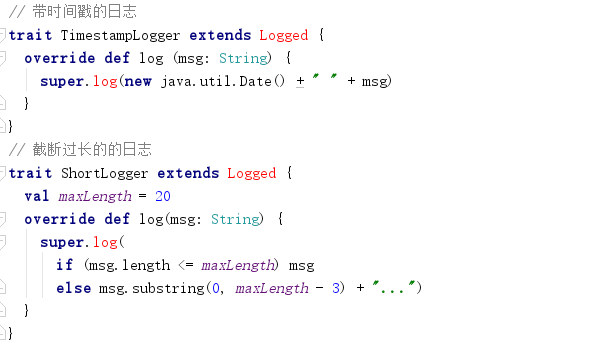


而在使用的时候，你还是可以使用>，<=这样的方法，因为它们已经在**Ordered特质**中得到了实现。

值得注意的是，Java8中也借鉴了类似的方式。现在你可以在Java的接口中定义默认的实现了。但是它的限制还是很多。对于Java8一些新特性和Scala的比较，今后会专门讲。

**2）堆叠和线性化**

在下面的例子里，我们定义了带***时间戳的日志和截断过长的的日志***，这两个特质。



我们的应用就可以混入这两个特质



你也许觉得trait和C++中的多继承很像。但是它们有着根本的区别。在多继承的框架里，你***必须指定哪个super被调用（比如通过虚函数）***。这样，在上面的例子中，只有一个log方法会调用。而***trait恰恰可以堆叠多个特性，每个log都会被调用。这才符合多个特质的语义***。

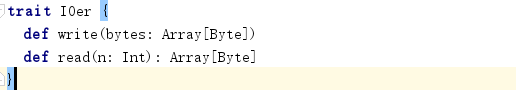
注意到混入是有顺序的，***基本上可以认为是最右边的先调用***。在更复杂的情景中，***Scala使用了线性化的方式，将复杂的继承图映射到一条线上，以保障多个特质执行的顺序***。

**3）使用原则**

什么时候使用特质呢？

* 不需要复用的，或者执行效率非常关键，考虑使用具体类。
* 会在多个***不太相关的类中使用，考虑使用特质***。
* 继承自Java的代码，或者希望Java client以后调用，使用抽象函数。
* 不太确定使用哪种方式，考虑先使用特质，以后再调整。

使用特质的一个重要原则就是***保持traits简短并且是正交的。不要把分离的功能混在一个trait里，考虑将最小的相关的意图放在一起。***例如，想象一下你要做一些IO的操作：



分离两个行为：



可以将它们以混入的方式实现一个



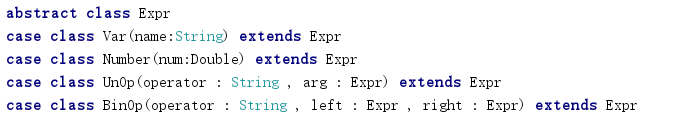
接口最小化促使更好的正交性和更清晰的模块化。

**8、模式匹配**

Scala强大的模式匹配机制，可以应用在***switch语句、类型检查以及“析构”***等场合。***样本类对模式匹配进行了优化***。这里介绍的是模式匹配的基本知识。

**1）样本类（case class）**

添加了case关键字的类便是样本类。例如



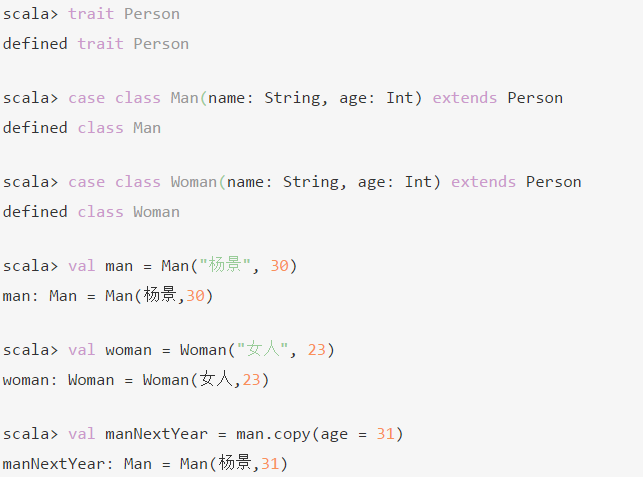
这种修饰符可以***让Scala编译器自动为这个类添加一些语法糖的效果***：

* 添加与类名一致的**工厂方法**。也就是说，可以写成Var("x")来构造Var对象，而不用new。
* 样本类参数列表中的所有参数隐式获得了val前缀，因此它被当作字段维护。
* 编译器为这个类***添加了方法toString,hashCode和equals等方法***。

case class是Scala中学用的一个特性，像Kotlin这样的语言也学习并引入了类似特性（在Kotlin中叫做：data class）。case class具有如下特性：

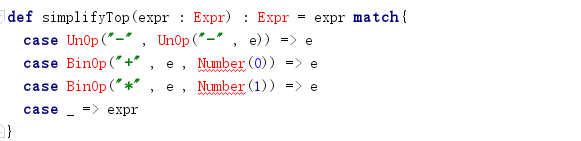
* 不需要使用new关键词创建，**直接使用类名**即可
* ***默认变量都是public final的，不可变的***。当然也可以显示指定var、private等特性，但一般不推荐这样用
* 自动实现了：equals、hashcode、toString等函数
* 自动实现了：Serializable接口，默认是可序列化的
* 可应用到**match case（模式匹配）**中
* 自带一个copy方法，可以方便的根据某个case class实例来生成一个新的实例

这里给出一个case class的使用样例：



**2）模式匹配**

模式匹配的例子如下：



这里的match对应Java里的switch，但是写在选择器表达式之后。即：

选择器 match {备选项}。

一个模式匹配包含了一系列备选项，每个都开始于关键字case。每个备选项都包含了一个模式及一到多个表达式。**箭头符号 => 隔开了模式和表达式**。

Scala中匹配表达式可以被看作Java风格Switch的泛化。但有三点不同：

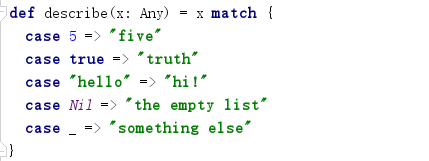
* match是Scala的表达式，始终以值作为结果；
* Scala的备选项表达式永远不会落入下一个case；
* 如果没有模式匹配，MatchError异常会被抛出。这意味着必须始终确信所有的情况都考虑到了，或者至少添加一个默认情况什么都不做。如 case \_ =>

**3）模式种类**

具体说，有这么几种匹配模式

**A. 常量模式**

区分一些常量



match表达式通过以代码编写的先后次序尝试每个模式来完成计算。

**B. 变量模式**

单纯的变量模式***没有匹配判断的过程，只是把传入的对象给起了一个新的变量***名。通常不会单独使用，而是在多种模式组合时使用，比如

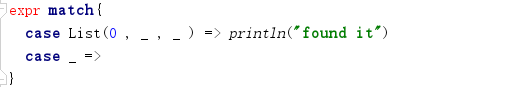


**C. 构造模式**

它的存在使得模式匹配真正变得强大。它由***名称及若干括号之内的模式构成***。如BinOp("+" , e , Number(0))。这种模式对于树型递归数据尤其有用。下面的序列模式和元组模式也可以认为是构造模式的特例。

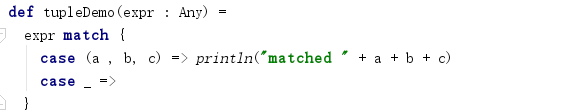
**D. 序列模式**

可以像匹配样本类那样匹配List或Array这样的序列类型。同样的语法现在可以指定模式内任意数量的元素。如：

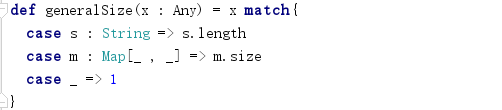


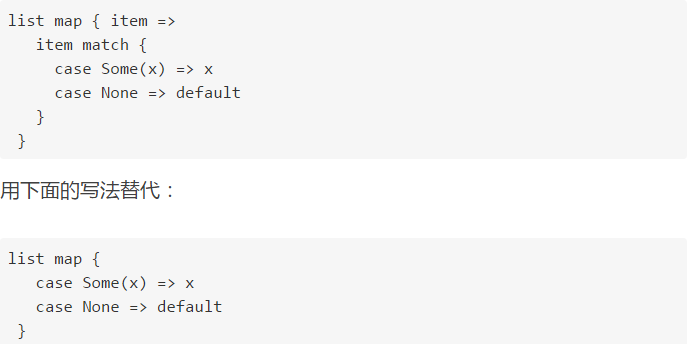
如果想***匹配一个不指定长度的序列***，可以指定**\_\*作为模式的最后元素**。它能匹配序列中0到任意数量的元素。

**E. 元祖模式**



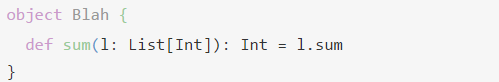
**F. 类型模式**





**4）object**

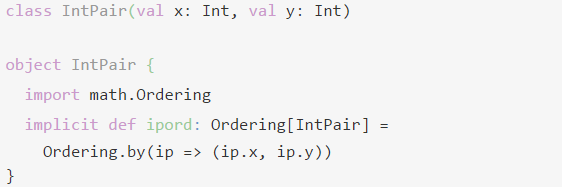
Scala有一种不同于Java的特殊类型，Singleton Objects。



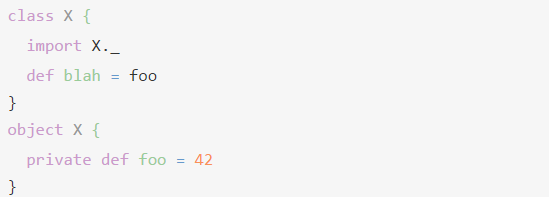
在Scala中，*没有Java里的static静态变量和静态作用域的概念*，取而代之的是：object。它除了可以实现Java里static的功能，它同时还是一个**线程安全的单例类**。

**伴身对象**

大多数的object都不是独立的，通常它都会***与一个同名的class定义在一起。这样的object称为伴身对象***。



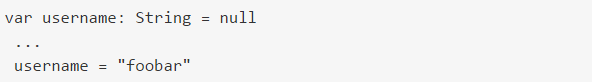
**伴身对象必需和它关联的类定义定义在同一个.scala文件**。伴身对象和它相关的类之间可以相互访问受保护的成员。在Java程序中，很多时候会把static成员设置成private的，在Scala中需要这样实现此特性：



**5）options**

**Option类型是一个容器，空(None)或满(Some(value))二选一**。它提供了使用null的另一种安全选择，应该尽可能的替代null。它是一个集合(最多只有一个元素)并用集合操所修饰，尽量用Option。

用 var username: Option[String] = None … username = Some(“foobar”)代替



前一种更安全：*Option类型静态的强制username必须对空(emptyness)做检测*。

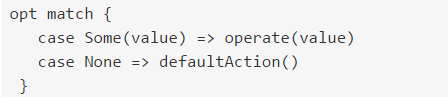
对一个Option值做条件判断应该用foreach

if (opt.isDefined) operate(opt.get)

上面的代码应该用下面的方式替代：

opt foreach { value =>operate(value)}

风格看起来有些古怪，但提供了更多的安全和简洁。如果两种情况都有(Option的None或Some)，用模式匹配



如果都缺少缺省值，用**getOrElse**方法：operate(opt getOrElse defaultValue)

不要过度使用Option： 如果有一个明确的缺省值——**一个Null对象**——直接用Null而不必用Option

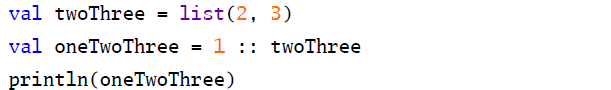
Option还有一个方便的构造器**用于包装空值(nullable value)**

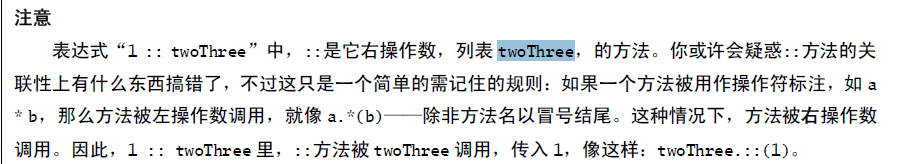
Option(getClass.getResourceAsStream("foo"))

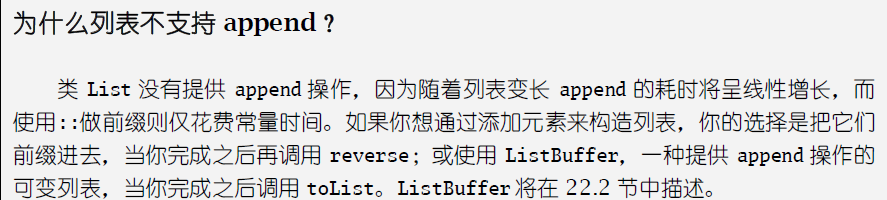
得到一个 Option[InputStream] 假定空值(None)时getResourceAsStream会返回null

**9、列表**

列表（List）应该是使用最多的数据结构了。因为***List是不可变的，他们表现得有些像Java的String***：当你在一个List上调用方法时，似乎这个名字指代的List看上去被改变了，而实际上它只是用新的值创建了一个List并返回







**1）列表构造**

注意列表中的元素类型必须一致。



构造列表的两个**基本单位是Nil和::**。上面的构造可以写成



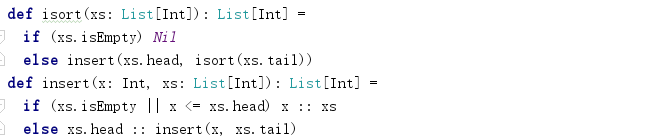
理解这一点对***列表的操作和模式匹配***很有帮助。

**2）列表操作**

Scala列表有三个基本操作：

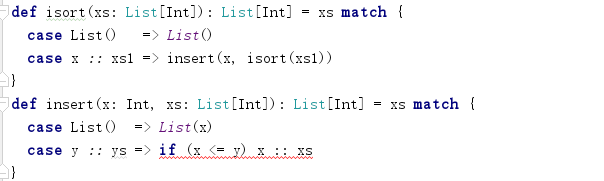
* head 返回列表第一个元素
* tail 返回一个列表，包含除了第一元素之外的其他元素
* isEmpty 在列表为空时返回true

对于Scala列表的任何操作都可以使用这三个基本操作来表达。比如插入排序算法就可以这样实现。



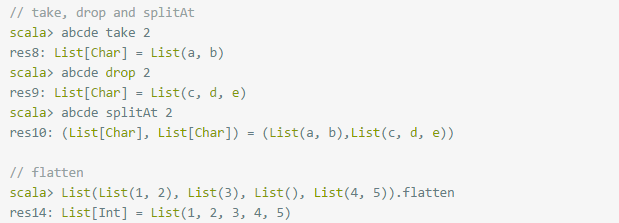
**3）列表模式**

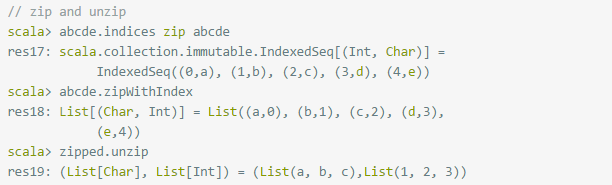
使用前面提到的模式匹配，可以使插入排序算法更加直观。

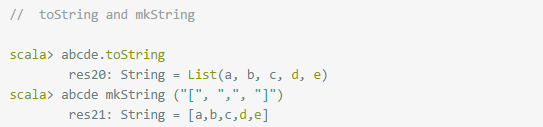


**4）列表常用操作**









**5）列表的常用高阶方法**

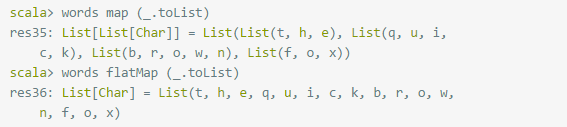
***高阶方法是指包含函数作为参数的方法***。这些方法是Scala作为一个函数式语言所特有的，能够方便的***实现列表的转变（Trasnformation）***。最近Java8也引入了类似的方法集合。

**A. map、flatmap**

xs map f 就是***将函数f作用于List xs中的每个元素，返回一个新的List***。比如：



flatmap和map十分相似，***唯一的区别就是返回时会把所用列表中的元素连起来，生成一个扁平的列表***。下面的例子比较了map和flatmap。



**B. filter**

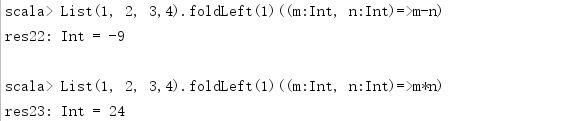
xs filter p，返回一个列表，包含xs中满足predicate p的元素。比如：



**C. fold**

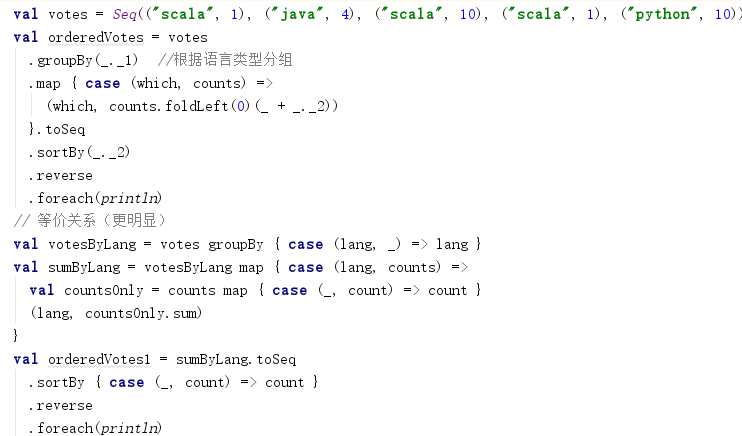
另外一类常用的列表操作是***将某种操作以某种累计的形式作用于列表中的元素（叠加或加法计算）***。这就是fold。通常又分为foldLeft和foldRight,只是执行顺序不同。比如：





这里***0为初始值（记住numbers是List[Int]类型）***，m作为一个**累加器**。从1加到10。如果使用foldRight那就是从10加到1。

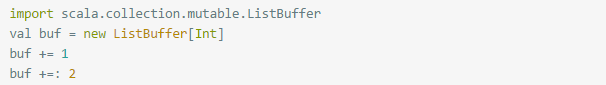
此外，还有很多高阶方法比如range, fill，sortWith等等，不再赘述。



**10、Collection**

**1）序列**

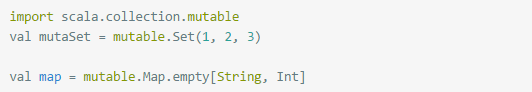
Array和List都属于序列（Sequence）。List支持从head插入，如果需要append，考虑使用**ListBuffer**。***ListBuffer是mutable的类型。支持constant time的从前和从后插入***。从前使用+=:，从后使用+=。ArrayBuffer和ListBuffer相似，不再赘述。



除了Array和List外，Scala中的String也是序列。

**2）Set、Map**

Set和Map，默认是**immutable**，除非显示指明。



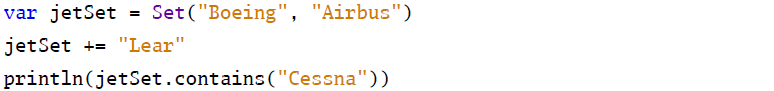
SortedMap和SortedSet，顾名思义，其元素或key是有序的。

Set可以通过***toList和toArray方法***转换为List或者Set

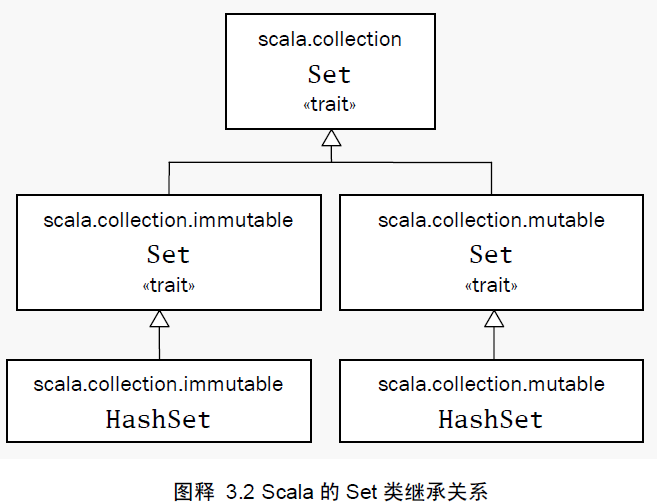
通过空集，可以实现immutable和mutable的转换。



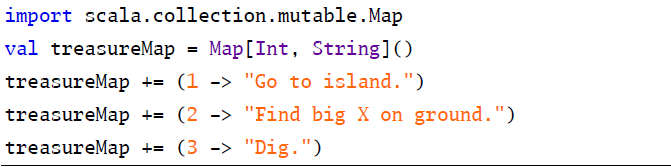
例如，Scala的API包含了集的一个基本特质：trait，特质这个概念接近于Java的接口。（你将在第12章找到更多关于特质的说明。）***Scala于是提供了两个子特质，一个是可变的集，另一个是不可变的集***。就如你在图3.2里会看到的，这三个特质都共享同样的简化名，Set。然而它们的全称不一样，因为每个都放在不同的包里。Scala的API里具体的Set类，如图3.2的***HashSet类，扩展了要么是可变的，要么不可变的Set特质***。（尽管Java里面称为“实现”了接口，在Scala里面称为“扩展”或“混入”了特质。）因此，如果你想要使用HashSet，你可以根据你的需要选择可变的或不可变的变体。创造集的缺省方法展示在代码3.5中：

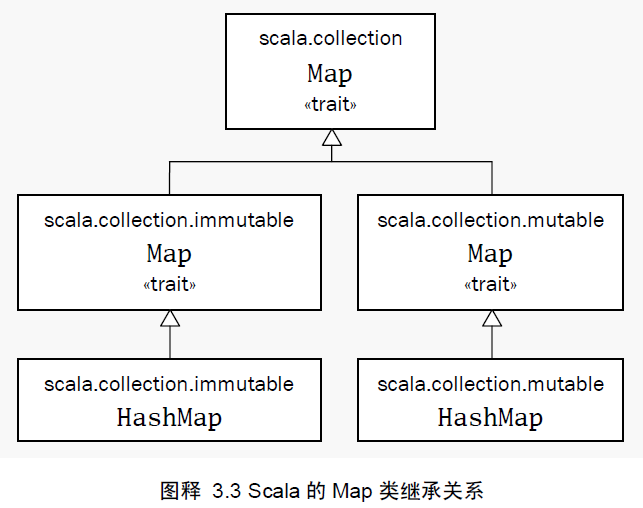


代码里，定义了名为jetSet的新var，并使用了包含两个字串，***"Boeing"和"Airbus"的不可变集完成了初始化***。就像例子中展示的，Scala中创建集的方法与创建列表和数组的类似：通过***调用Set伴生对象的名为apply的工厂方法***。代码3.5中，对scala.collection.immutable.Set的伴生对象调用了apply方法，***返回了一个缺省的，不可变Set的实例***。Scala编译器推断jetSet的类型为不可变Set[String]。



Map集合



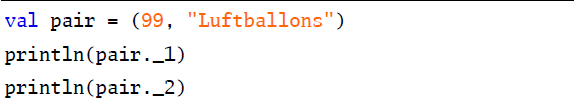


引用了可变形式的Map。然后就定义了一个叫做treasureMap的val并使用空的包含整数键和字串值的可变Map初始化它。***映射为空是因为你没有向工厂方法传递任何值（“Map[Int, String]()”的括号里面是空的）***。下面的三行里你使用->和+=方法把键/值对添加到Map里。像前面例子里演示的那样，Scala编译器把如**1 -> "Go to island"这样的二元操作符表达式转换为(1).->("Go to island.")**。因此，当你输入1 -> "Go to island."，你实际上是在值为1的Int上**调用->方法**，并传入值为"Go to island."的String。***这个->方法可以调用Scala程序里的任何对象，并返回一个包含键和值的二元元组***。然后你在把这个元组传递给treasureMap指向的Map的+=方法。最终，最后一行输出打印了treasureMap中的与键2有关的值。

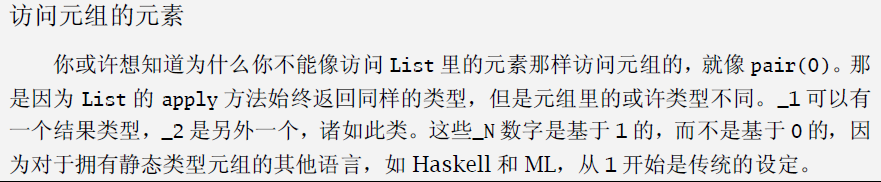
**3）元组**

和Array，List不同，Tuple（元组）中的对象可以有不同的类型。元组的一个典型应用就是返回多个对象。比如在统计文本时，需要返回词和其所在位置。

另一种有用的容器对象是元组：tuple。与列表一样，元组也是不可变的，但与列表不同，元组可以包含不同类型的元素。而列表应该是List[Int]或List[String]的样子，元组可以同时拥有Int和String。元组很有用，比方说，如果你需要在方法里返回多个对象。Java里你将经常创建一个JavaBean样子的类去装多个返回值，Scala里你可以简单地返回一个元组。而且这么做的确简单：实例化一个装有一些对象的新元组，只要把这些对象放在括号里，并用逗号分隔即可。一旦你已经***实例化了一个元组，你可以用点号，下划线和一个基于1的元素索引访问它***。

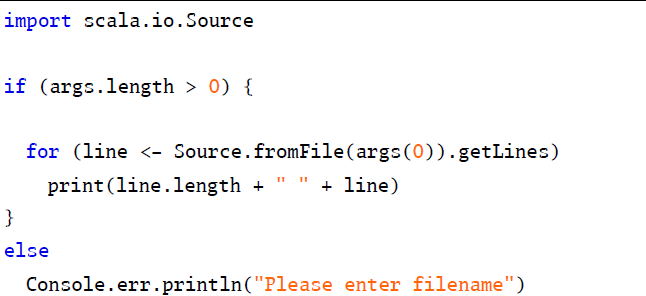


***元组的实际类型取决于它含有的元素数量和这些元素的类型***。因此，(99, "Luftballons")的类型是Tuple2[Int, String]。('u', 'r', 'the', 1, 4, "me")是Tuple6[Char, Char, String, Int, Int, String]。

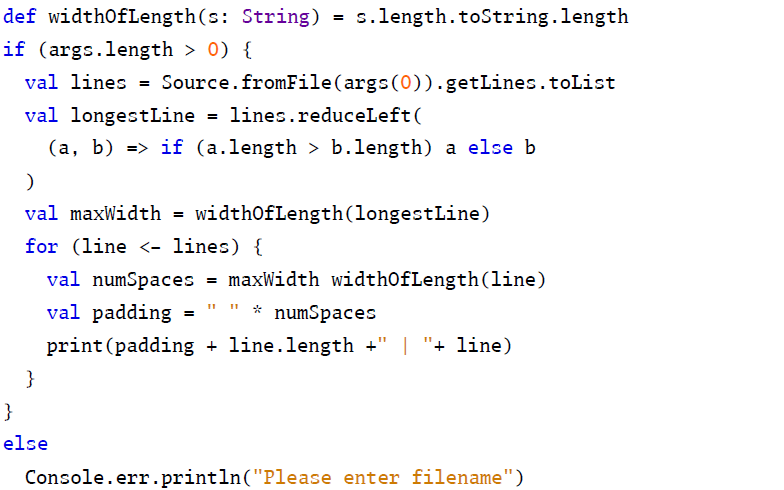


**11、从文件里读取信息行**

建立一个从文件中读行记录，并把行中字符个数前置到每一行，打印输出的脚本



此脚本开始于***从包scala.io引用名为Source的类***。然后检查是否命令行里定义了至少一个参数。若是，则第一个参数被解释为要打开和处理的文件名。***表达式Source.fromFile(args(0))，尝试打开指定的文件并返回一个Source对象，你在其上调用getLines。函数返回Iterator[String]***，在每个枚举里提供一行包括行结束符的信息。for表达式枚举这些行并打印每行的长度，空格和这行记录。如果命令行里没有提供参数，最后的else子句将在标准错误流中打印一条信息。

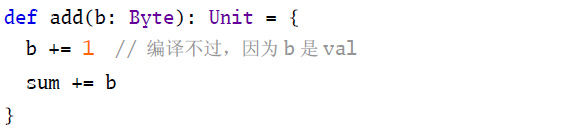


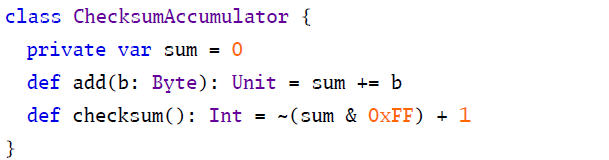
**reduceLeft**方法把传入的方法应用于lines的前两个元素，然后再应用于第一次应用的结果和lines接下去的一个元素，等等，直至整个列表。每次这样的应用，结果将是碰到的最长一行，因为传入的函数，***(a, b) => if (a.length > b.length) a else b，返回两个传入字串的最长那个***。reduceLeft将传回最后一次应用的结果，也就是本例lines中包含的最长字串。

在这个for表达式里，你再一次枚举了全部行记录。对于每一行，首先计算行长度前所需的空格并把它赋给numSpaces。然后用表达式：" " \* numSpaces创建包含numSpaces个空格的字串。最终，你打印出你想要格式的信息。

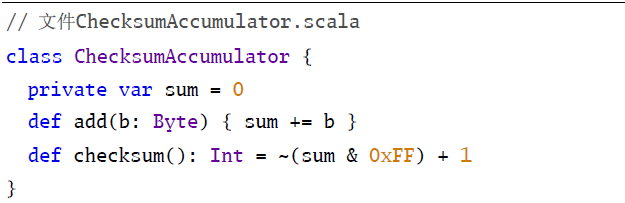
**注意：**

传递给方法的任何参数都可以在方法内部使用。***Scala里方法参数的一个重要特征是它们都是val，不是var***（参数是val的理由是val更容易讲清楚。你不需要多看代码以确定是否val被重新赋值，而var则不然。） 如果你想在方法里面给参数重新赋值，结果是编译失败：





像ChecksumAccumulator的add方法那样的结果类型为Unit的方法，执行的目的就是它的副作用。通常我们***定义副作用为在方法外部某处改变状态或者执行I/O活动***。比方说，在add这个例子里，副作用就是sum被重新赋值了。表达这个方法的另一种方式是去掉结果类型和等号，把方法体放在大括号里。这种形式下，***方法看上去很像过程：procedure，一种仅为了副作用而执行的方法***。

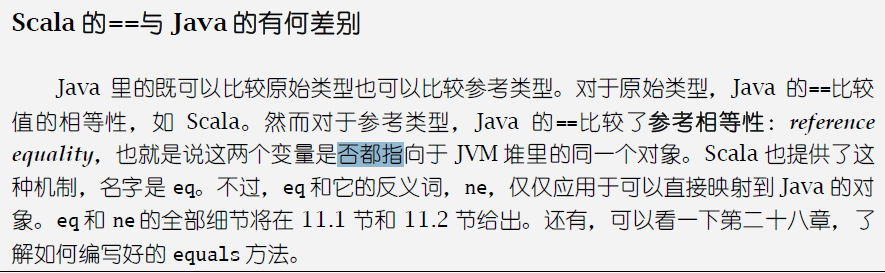


应该注意到令人困惑的地方是当你***去掉方法体前面的等号时，它的结果类型将注定是Unit***。不论方法体里面包含什么都不例外，因为Scala编译器可以把任何类型转换为Unit。例如，如果方法的最后结果是String，但方法的**结果类型**被声明为Unit，那么String将被转变为Unit并失去它的值。



例子里，String被转变为Unit因为Unit是函数f声明的结果类型。***Scala编译器会把一个以过程风格定义的方法，就是说，带有大括号但没有等号的，在本质上当作是显式定义结果类型为Unit的方法***。例如：



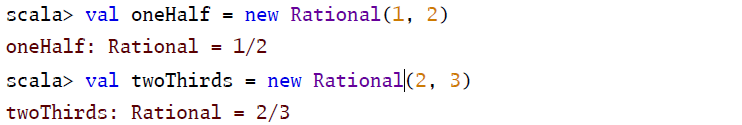


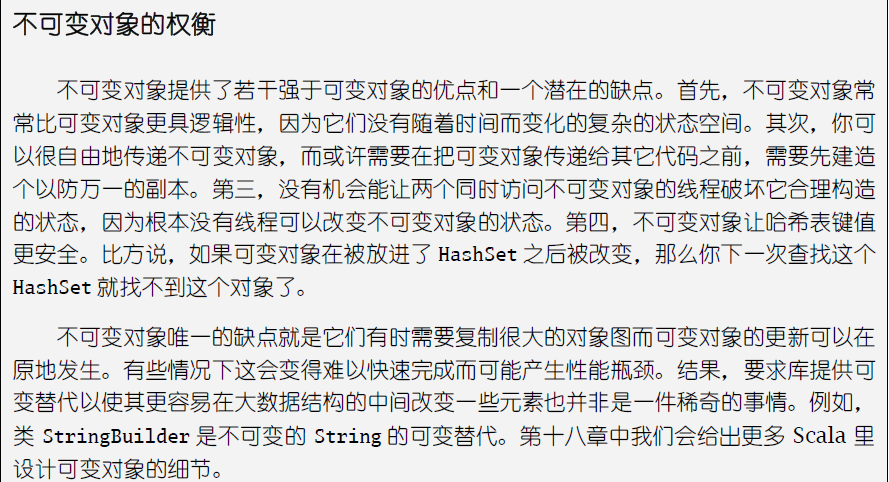
当同样优先级的多个操作符肩并肩地出现在表达式里，**操作符的关联性**：associativity决定了操作符分组的方式。Scala里操作符的关联性取决于它的最后一个字符。正如第3章里49页提到的，任何以‘:’字符结尾的方法由它的右手侧操作数调用，并传入左操作数。以其他字符结尾的方法有其他的说法。它们都是被左操作数调用，并传入右操作数。因此***a \* b变成a.\*(b)，但是a:::b变成b.:::(a)***。然而，不论操作符具有什么样的关联性，它的操作数总是从左到右评估的。因此如果b是一个表达式而不仅仅是一个不可变值的指针的话，那么更精确的意义上说，a:::b将会当作是：

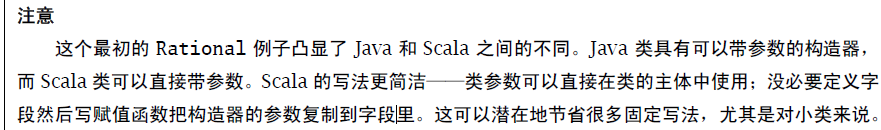


**12、函数式对象**

一个分数加到另外一个分数上，产生的结果是一个新的分数。而***原来的数不会被“改变”。我们将在本章设计的不可变的Rational类将秉承这一属性***。每个分数将都被表示成一个**Rational对象**。当两个Rational对象相加时，一个新的带着累加结果的Rational对象将被创建出来。

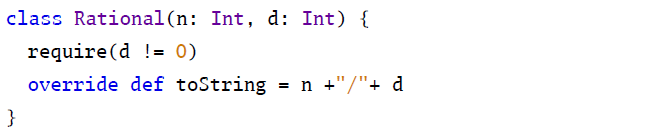






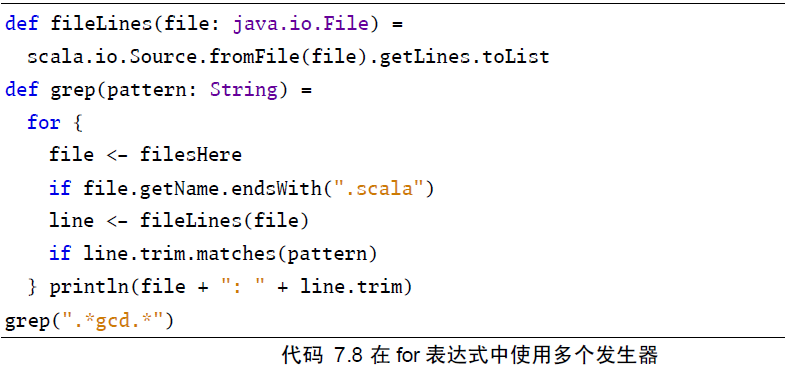
面向对象编程的一个优点就是它允许你把数据封装在对象之内以便于你确保数据在整个生命周期中是有效的。像***Rational这样的不可变对象，这就意味着你必须确保在对象创建的时候数据是有效的（并且，确保对象的确是不可变的，这样数据就不会在之后变成无效的状态）***。由于零做分母对Rational来说是**无效状态**，因此在把零传递给d的时候，务必不能让Rational被构建出来。

解决这个问题的最好办法是为***主构造器定义一个先决条件***：precondition说明d必须为非零值。***先决条件是对传递给方法或构造器的值的限制，是调用者必须满足的需求***。一种方式是使用require方法



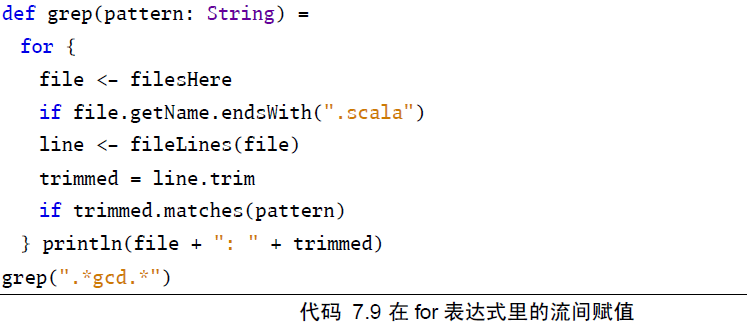
**require方法带一个布尔型参数**。如果传入的值为真，require将正常返回。反之，require将通过抛出IllegalArgumentException来阻止对象被构造。

**13、for循环**



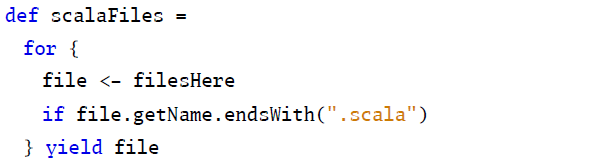
**mid-stream（流间）变量绑定**

请注意前面的代码段中重复出现的表达式line.trim。这不是个可忽略的计算，因此你或许想每次只算一遍。通过***用等号（=）把结果绑定到新变量可以做到这点***

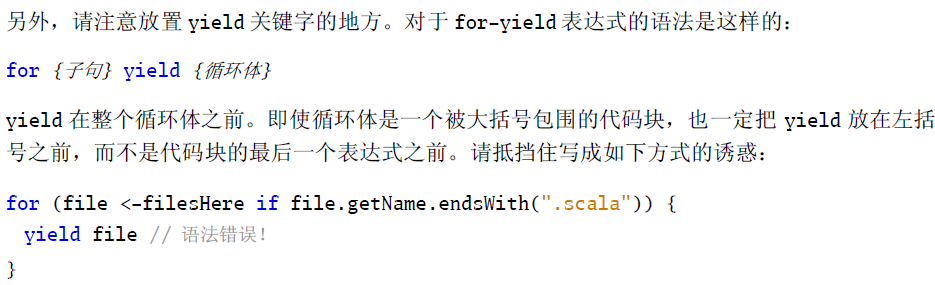


代码中，名为trimmed的变量被从半当中引入for表达式，并被**初始化为line.trim**的结果值。之后的***for表达式就可以在两个地方使用这个新变量，一次在if中，一次在println中***。

**制造新集合（得到的集合类型与原始集合类型一致）**



for表达式在每次执行的时候都会制造一个值，本例中是file。当for表达式完成的时候，结果将是一个包含了所有产生的值的集合。结果集合的类型基于枚举子句处理的集合类型。本例中结果为Array[File]，因为filesHere是数组并且产生的表达式类型是File。

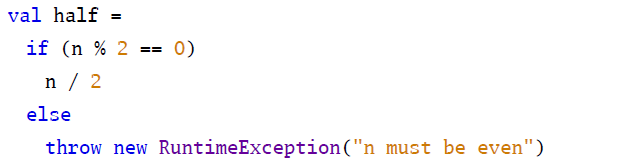


**抛出异常**

异常的抛出看上去与Java的一模一样。首先创建一个异常对象然后用throw关键字抛出：



Scala里， ***throw也是有结果类型的表达式***。下面举一个有关结果类型的例子：

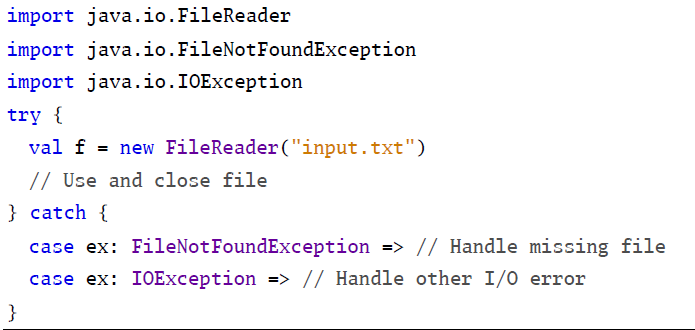


这里发生的事情是，如果n是偶数，half将被初始化为n的一半。如果n不是偶数，那么在half能被初始化为任何值之前异常将被抛出。因此，无论怎么说，***把抛出的异常当作任何类型的值都是安全的***。任何使用从throw返回值的尝试都不会起作用，因此这样做无害。

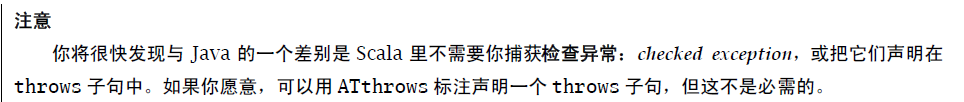
从技术角度上来说，***抛出异常的类型是Nothing***。尽管throw不实际得出任何值，你还是可以把它当作表达式。这种小技巧或许看上去很怪异，但像在上面这样的例子里却常常很有用。if的一个分支计算值，另一个抛出异常并得出Nothing。整个if表达式的类型就是那个实际计算值的分支的类型。

**捕获异常**

选择catch子句这样的语法的原因是为了与Scala很重要的部分：模式匹配：pattern matching保持一致。

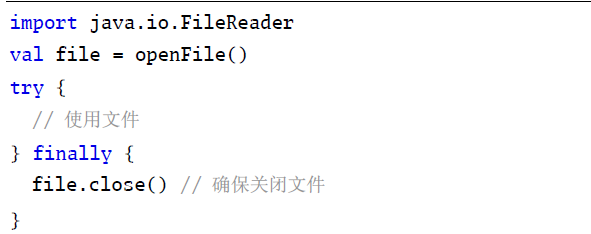


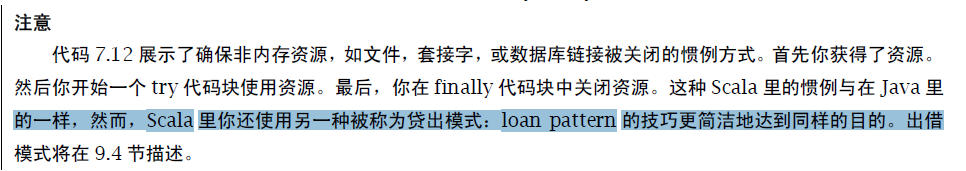
这个try-catch表达式的行为与其它语言中的异常处理一致。程序体被执行，如果抛出异常，每个catch子句依次被尝试。本例中，如果异常是FileNotFoundException，那么第一个子句将被执行。如果是IOException类型，第二个子句将被执行。***如果都不是，那么try-catch将终结并把异常上升出去***。



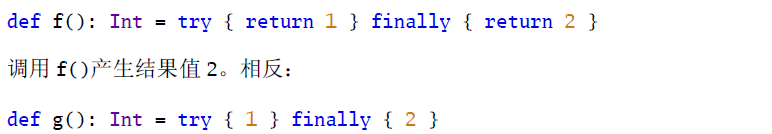
**finally子句**

如果想让某些代码无论方法如何中止都要执行的话，可以把表达式放在finally子句里。如，你或许想让打开的文件即使是方法抛出异常退出也要确保被关闭。



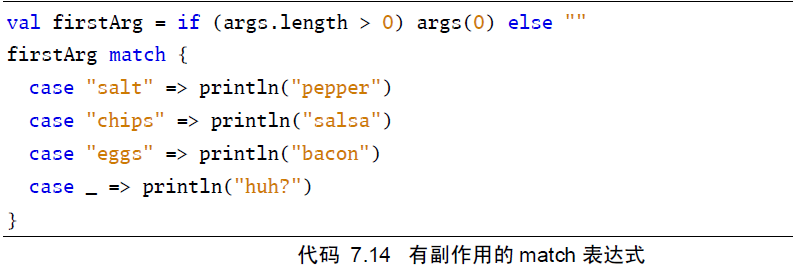


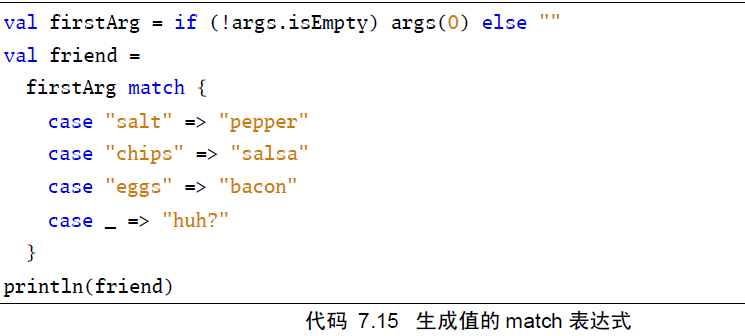
如果熟悉Java，不说你也知道，Scala的行为与Java的差别仅源于Java的try-finally不产生值。Java里，***如果finally子句包含一个显式返回语句，或抛出一个异常，这个返回值或异常将“凌驾”于任何之前源于try代码块或某个它的catch子句产生的值或异常之上***。如：



调用g()产生1。这两个例子展示了有可能另大多数程序员感到惊奇的行为，因此通常最好还是避免从finally子句中返回值。最好是***把finally子句当作确保某些副作用，如关闭打开的文件，发生的途径***。

**match表达式**

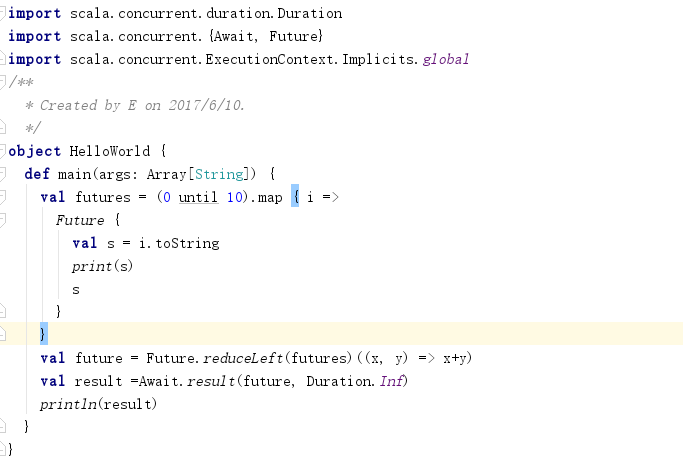




**14、并发**

Scala是对多核和并发编程的支付做得非常好，它的***Future类型提供了执行异步操作的高级封装***。

Future对象完成构建工作以后，控制权便会立刻返还给调用者，这时结果还不可以立刻可用。**Future实例是一个句柄，它指向最终可用的结果值**。不论操作成功与否，在future操作执行完成前，代码都可以继续执行而不被阻塞。Scala提供了多种方法用于处理future



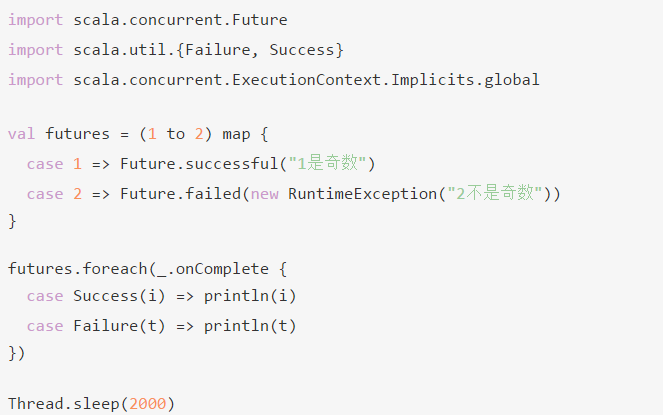
（**注意：**必须把隐式的ExecutionContext对象包引入，否则报错）

上面代码创建了**10个Future对象**，Future.apply方法有两个参数列表。**第一个参数**列表包含一个需要并发执行的命名方法体（by-name body）；而**第二个参数列表**包含了隐式的ExecutionContext对象，可以简单的把它看作一个线程池对象，它决定了这个任务将在哪个异步（线程）执行器中执行。futures对象的类型为IndexedSeq[Future[String]]。本示例中使用Future.reduce把一个futures的IndexedSeq[Future[String]]类型压缩成单独的Future[String]类型对象。Await.result用来阻塞代码并获取结果，输入的Duration.Inf用于设置超时时间，这里是无限制。

这里可以看到，**在Future代码内部的println语句打印输出是无序的**，但**最终获取的result结果却是有序的**。这是因为*虽然每个Future都是在线程中无序执行，但Future.reduce方法将按传入的序列顺序合并结果*。

除了使用**Await.result阻塞代码获取结果**，我们还可以使用**事件回调的方式异步获取结果**。Future对象提供了几个方法通过回调将执行的结果返还给调用者，常用的有：

* onComplete: **PartialFunction**[Try[T], Unit]：当任务执行完成后调用，无论成功还是失败
* onSuccess: PartialFunction[T, Unit]：当任务成功执行完成后调用
* onFailure: PartialFunction[Throwable, Unit]：当任务执行失败（异常）时调用



futures.onComplete方法是一个**偏函数**，它的参数是：**Try[String]**。Try有两个子类，成功是返回Success[String]，失败时返回Failure[Throwable]，可以*通过模式匹配的方式获取这个结果*。

线程提供了一种表达并发的方式：它们给你独立的，**堆共享的(heap-sharing)由操作系统调度的执行上下文**。然而，在*java里线程的创建是昂贵的，是一种必须托管的资源，通常借助于线程池*。这对程序员创造了额外的复杂，也造成高度的耦合：很难从所使用的基础资源中分离应用逻辑。

这种复杂度尤其明显当创建高度分散(fan-out)的服务时： 每个输入请求导致一大批对另一层系统的请求。在这些系统中，线程池必须被托管以便根据每一层请求的比例来平衡：管理不善的线程池会渗入到另一个里(bleeds into another)。

一个健壮系统必须**考虑超时和取消**，两者都需要引入另一个“控制”线程，使问题更加复杂。注意若线程很廉价这些问题也将会被削弱：不再需要一个线程池，超时的线程将被丢弃，不再需要额外的资源管理。

因此，资源管理危害了模块。