Scala学习笔记

harli

2014/12/29

# 基础概念

1. “均码”哲学：
2. 增加新的控制结构：可以由Scala的基于Actor的并发编程API阐明。

Actor是能够实现于线程模型之上的并发抽象。可以通过彼此间传递消息来实现通信。

1. 函数式编程有两种指导理念：
   * 第一种理念是函数是头等值：函数作为头等值这种理念简化了操作符的抽象和新控制结构的创建。
   * 函数式编程的第二种理念是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：不可变数据结构是函数式语言的一块基石。
2. 简单的规则：如果方法实用操作符来标注，那么左操作符是方法的调用者 —— 除非——方法名以冒号结尾。这种情况下，方法被右操作数调用。
3. Scala的一个基本规则：方法若只有一个 参数，调用的时候就可以省略点及括号。

注意：这个语法只有在明确指定方法调用的接受者时才有效。

例如：不可以写成“println 10”，但是可以写成“Console println 10”。

1. 从技术层面上来说，Scala没有操作符重装，因为它根本没有传统意义上的操作符。
2. 与Java相比，Scala鲜有特例。与Scala其他的类一样，数组也只是类的实例。
3. 一个原则：任何对于对象的值参数应用将都被转换为对apply方法的调用。当然前提是这个类型实际定义过apply方法。所以这不是特例，而是通用法则。
4. 与之类似的是，当对带有括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。

array(0) = “hello” 将被转化为： array.update(0, “hello”)。

1. Scala编译器会尽可能在编译完成的代码里利用Java数组、原始类型和原生的数字计算方法。
2. 方法没有副作用是函数式风格编程的重要理念，计算并返回值应该是方法唯一的目的。
3. 变长参数列表，也可以称为重复参数（repeated parameters）。
4. 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。
5. 类StringBuilder是不可变的String的可变版本。

## by-name与by-value

1. 如果操作符都只是方法的话，短路机制（关系运算符）是怎么工作的？

所有的Scala方法都有延迟其参数评估乃至取消其评估的机制，被称为传名参数（by-name parameter）。

Harli-扩展：Cousera的函数式编程原理公开课程。

by-name 、 by-value对比。

## 对象相等性（Scala与Java）的差异

《Scala编程》：P90

1. Java里的==既可以比较原始类型也可以比较引用类型。

* 原始类型：Java的==比较值的相等性，与Scala一致。
* 引用类型：Java的==比较了引用相等性（reference equality），也就是说比较的是这两个变量是否都指向JVM堆里的同一个对象。Scala也提供了这种机制，名字是eq。不过，eq和它的反义词ne，仅仅应用于可以直接映射到Java的对象。

1. 自动检查机制不会检查右侧的参数，但是任何合理的equals方法都应在参数为null的时候返回false。

## 源文件

1. Scala对于源文件的命名没有硬性规定。然而通常情况下如果不是脚本，推荐的风格是Java那样按照所包含的类名来命名文件。
2. 脚本必须以结果表达式结束 —— 如果以定义结尾则不是脚本，此时以脚本方式执行时，Scala解释器讲会报错说不是以结果表达式结束的。

## 分号推断

1. 分号推断：Scala程序里，语句末尾的分号通常的可选的。
   * 愿意可以加；
   * 如果一行里仅有一个语句时可以不加；
   * 如果一行包含多条语句时，分号则是必须的。

跨越多行的语句时，多数情况无须特别处理，Scala将在正确的位置分割语句。

Scala通常的风格是把操作符放在行尾而不是行头。

1. 分号推断的规则

分割语句的具体规则既出人意料地简单又非常有效。那就是，除非以下情况的一种成立，否则行尾被认为是一个分号：—— 即以下情况时，行尾不是分号……

* 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。
* 下一行开始于不能作为语句开始的词。
* 行结束于括号(…)或方框[…]内部，因为这些符号不可能容纳多个语句。

harli总结：行尾、行首是否合法，以及是否在配对符合（偶的用词，嘿嘿）中间断开。

## 常量

Scala与Java的习惯不一致的地方在于常量名。

Scala，constant不等同于val。

Java中，习惯上常量名全部大写，Scala中习惯只是第一个字母必须大写。即其常数也用驼峰式风格。

## 等效推论（equational reasoning）

1. 等效推论：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. 使用val而不是var的一个好处：它能更好地支持等效推论。

Harli-扩展：Causera 课程 – 函数式编程原理。

## 函数

1. Scala提供了许多Java中没有的定义函数的方式：除了成员函数外，还有内嵌在函数中的函数，函数字面量和函数值。
2. 定义在函数内的函数：这种本地函数仅在包含它的代码块中可见。

本地函数可以访问包含它的函数的参数 —— 对外层函数入参的直接使用。

1. 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。

这个区别很像类（源代码）和对象（运行期）之间的关系。

如函数字面量 (x: Int) => x + 1 ：=> 指明这个函数把左边的东西转变为右边的东西。

1. 任何函数值都是某个扩展了scala包的若干FunctionN特质之一的类的实例。
2. 函数字面量的短格式：
   * 一种方式是去除参数类型 —— 目标类型化：target typing，因为表达式的目标使用影响了表达式的类型化。

xxx.filter((x)=>x>0) 其中x的类型被省略。

* + 某些参数的类型是被推断的，省略其外的括号，是第二种去除无用字符的方式。

xxx.filter(x=>x>0) 其中x两边的括号不是必须的。

* + 第三种方式：可以把下划线当做一个或更多参数的占位符，主要每个参数在函数字面量内进出现一次。

xxx.filter( \_>0) 。

使用下划线当做参数的占位符时，有时编译器可能无法推断缺失的类型参数，这时可以使用冒号指定类型。

val f = (\_:Int) + (\_: Int) ： \_+\_将扩展成带两个参数的函数字面量。

1. 高阶函数的好处之一是它们能让你创造控制抽象从而减少代码重复。
2. 高阶函数的另一个重要应用是把它们放在API里使客户代码更简洁。
3. Scala不允许在运行期粘合代码。
4. 在拥有头等函数的语义中，即使语言的语法是固定的，你也可以有效地制作新的控制结构。所有你需要做的就是创建带有函数做参数的方法。
5. 借贷模式（loan pattern） ：《Sccala编程》P141
6. Scala的任何方法调用，如果你确定只传入一个参数，就能可选地使用花括号替代小括号包围参数。

在传入一个参数时，可以用花括号替代小括号的机制，其目的是让客户程序员能写出包围在花括号内的函数字面量。这可以让方法调用感觉更像控制抽象。

### 部分应用函数

1. 在Scala中，当你调用函数，传入任何需要的参数，实际是在把函数应用到参数上。
2. 当使用下划线\_，表示整个参数列表的占位符时，你就是在写一个部分应用函数（partially applied function）。
3. 部分应用函数是一种表达式，你不需要提供函数所需的所有参数，待之以仅提供部分，或不提供所需参数。

此时编译器的工作：《Scala编程》：P127

编译器会产生一个新的函数类，并带有apply方法，带所需的参数。

1. 尽管不能把方法或嵌套函数赋值给变量，或当做参数传递给其他方法，但是如果你通过在名称后面加下划线的方式把方法或嵌套函数包装在函数值中，就可以做到了。
2. 如果是省略所有参数的偏程序表达式，如println \_，而且在代码的那个地方正需要一个函数，你可以去掉下划线，从而表达式更简明。

xxx.foreach(println \_) ===》 xxx.foreach(println)

在不需要函数的地方使用时，编译器会报错：如val a = println

harli：对应产生的新的函数类，自动用apply方法。

### 偏函数

1. 偏函数：名字源自于函数未被应用于它所有的参数。
2. 为什么要使用尾下划线？

《Scala编程》 ： P128

1. 为什么要使用尾下划线？

Scala的部分应用函数语法凸显了Scala与经典函数式语言，如Haskell或ML之间设计折衷的差异。在经典函数式语言中，部分应用函数被当做普遍的例子。更进一步，这些语言拥有非常严格的静态类型系统能够曝露出你在部分应用中可能犯的错误。进一步说，子类型推断的面向对象的传统和全局的根类型能接受一些被经典函数式语言认为是错误的程序。

Scala需要你指定显示省略的函数参数，尽管标志简单到仅用一个‘\_’。也只有在需要函数类型的地方，Scala才允许你省略这个仅用的\_。

### 柯里化

1. 柯里化的函数被应用于多个参数列表，而不是仅仅一个。

### 传名参数（by-name parameter）

**Harli** - 扩展：与传值参数进行对比，参考Causera上的Scala课程。

1. 要实现一个传名函数，要定义参数的类型开始于=> 而不是()=>。
2. 传名类型中，空的参数列表，()，被省略，它仅在参数中被允许。
3. 没有什么传名变量或传名字段这样的东西。

### 闭包

1. 函数字面量在运行时创建的函数值（对象）被称为闭包：closure。名称源自于通过“捕获”自由变量的绑定，从而对函数字面量执行的“关闭”行动。
2. 不带自由变量的函数字面量，被称为封闭项（closed term），这里项（term）指的是一小部分源代码。因此按照这个函数字面量在运行时创建的函数值严格意义上来讲就不是闭包。因为在编写时就已经是封闭了。
3. 但任何带有自由变量的函数字面量，都是开放项（open term）。因此，在运行期创建的函数值将必须捕获对自由变量的绑定。
4. 创建的闭包看到闭包之外对自由变量的变化，反过来同样，闭包对捕获变量做出的改变在闭包之外也可见。
5. 如下的说法通用于所有情况：使用的实例是那个在闭包被创建的时候活跃的。

## 重复参数

1. Scala中，你可以指明函数的最后一个参数是重复的。在参数的类型之后放一个星号。
2. 函数内部，重复参数的类型是声明参数类型的数组。
3. 如果要将数组传入重复参数，需要在数组参数后添加一个冒号和一个\_\*符号。这个标注告诉编译器把该传入数组的每个元素当做参数，而不是当作单一的参数。

## 尾递归

1. 在最后一个动作调用自己的函数，被称为尾递归： tail recursive。
2. Scala编译器检测到尾递归就用新值更新函数参数，然后把它替换成一个回到函数开通的跳转。
3. 递归经常是比基于循环的更优美和简明的方案。
4. 如果方案是尾递归，就无须付出任何运行期开销。
5. 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。

这可能会让检查程序的堆栈跟踪信息并失败的程序员感到惊奇。

尾递归开关项：-g:notailcalls

传递该参数到scala的shell或者scalac编译器。

1. 尾递归的局限：Scala里尾递归的使用局限很大，因为JVM指令集使实现更加先进的尾递归形式变得很困难。
2. Scala仅优化了直接递归调用使其返回同一个函数。如果递归是简介的，就没有优化的可能性了。
3. 同样，如果最后一个调用是一个函数值你也不能获得尾递归调用优化。
4. 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什么其他的中间函数的情况。

## 不可变对象的权衡

不可变对象提供了若干强于可变对象的有点和一个潜在的缺点。

1. 优点
   * 不可变对象常常比可变对象更易理清头绪，因为它们没有随着时间变化的复杂的状态空间。
   * 可以很自由地传递不可变对象，但对于可变对象来说，传递之前需建副本以防万一。
   * 一旦不可变对象完成构造之后，就不会有线程因为并发访问而破坏对象内部状态，因为根本没有线程可以改变不可变对象的状态。
   * 不可变对象让哈希表键值更安全。比方说，如果可变对象在进入HashSet之后被改变，那么你下一次查找这个HashSet时就找不到这个对象了。
2. 缺点：有时需要复制很大的对象表而可变对象的更新可以在原址发生。

# 数组

1. Scala数组是可变的同类对象序列。尽管数组在实例化之后长度固定，但它的元素值却是可变的。所以说数组是可变的。

# 列表

1. Scala列表类（List）是不可变的同类对象序列。实际上，Scala的列表是为了实现函数式风格的编程而设计的。
2. 列表类最常用的操作符或许是“::”，发音为“cons”。可以把新元素组合到现有列表的最前端，然后返回作为执行结果的新列表。
3. Nil：是空列表的简写。
4. 要在最后用到Nil的理由是::是定义在List类上的方法。
5. 列表为什么不支持添加（append）操作？

List类没有提供append操作，因为随着列表边长，append的耗时将呈线性增长，而使用::做前缀则仅耗用固定的时间。

1. ListBuffer：一种提供append操作的可变列表，完成之后调用toList。

# 元组

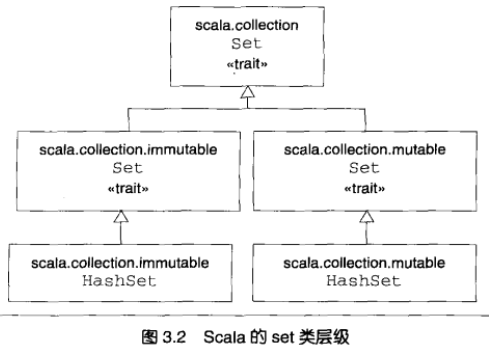
1. 元组也是不可变的；
2. 与列表不同，元组可以包含不同类型的元素。
3. 元组实例化之后，可以用点号、下划线和基于1的索引访问其中的元素。其中点号“.”与访问字段或调用方法的点没有区别。
4. 元组的实际类型取决于它含有的元素数量和这些元素的类型。
5. 访问元组的元素：

你或许想知道为什么不能用访问列表的方法来访问元组，如pari(0)。那是因为列表的apply方法始终返回同样的类型，但元组的类型不尽相同。\_1的结果类型可能与\_2的不一致。诸如此类，因此两者的访问方法也不一样。

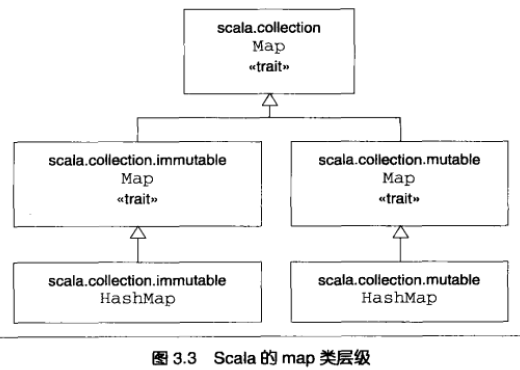
另外，这些\_N的索引是基于1的，而不是基于0的，这是因为对于拥有静态类型元组的其他语言，如Haskell何ML， 从1开始是传统的设定。

# 集(set)和映射(map)

1. Scala致力于充分利用函数式和指令式风格两方面的好处，因此它的集合（collection）库区分为可变类型和不可变类型。
2. 对于set和map来说，Scala同样有可变的和不可变的，不过并非各提供两种类型，而是通过类继承的差别把可变性差异蕴含其中。
3. 只有可变集提供的才是真正的+=方法，不可变集不是。
4. set类层级



1. map类层级



1. 向函数式风格转变的方式之一，就是尝试不用任何var编程。

# 文件

1. scala.io.Source

# 内建控制结构

1. Scala内建的控制结构：仅有if、while、for、try、match和函数调用。

如此少的理由是，Scala从语法层面支持函数字面量。

1. 几乎所有的Scala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。
2. While和do-while结构之所以被称为“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()。

与此类似：对var再赋值等式本身也是Unit值。

1. ()的存在是Scala的Unit不同于Java的void的地方。

《Scala编程》P107

# 隐式转换

## 基础知识

1. 要隐式转换起作用，需要定义在作用范围之内。

# 编译与运行

1. 编译：
   * scalac xxx.scala yyy.scala ： 编译
   * Scala发布包中包含了一个叫做fsc（快速Scala编译器，fast Scala compiler）的Scala编译器后台服务（daemon）。

使用方法：fsc xxx.scala yyy.scala

第一次执行fsc时，会创建一个绑定在你计算机端口上的本地服务器后台进程。然后它就会把文件列表通过端口发送给后台进程，由后台进程完成编译。

下一次执行fsc时，检测到后台进程已经在运行了，于是fsc将只把文件列表发给后台进程，它会立即开始编译文件。

使用fsc，只需在首次运行的时候等待Java运行环境的启动。

停止fsc后台进程：fsc –shutdown

两种方式，最终都将完成Java类文件的创建，然后可以用scala命令，像之前的例子里调用解释器那样运行它。但这里用的是包含了正确签名的mian方法的独立对象名。

如scala xxx params

1. scala xxx.scala params ： 运行

Scala程序用来“解释”Scala源文件的真正机制是，它把Scala源码编译成字节码，然后立即通过类装载器装载它们，并执行它们。

## Scala的标识符

1. 构建方式：
   * 字母数字式：字母数字标识符（alphanumeric identifier）以字母或下划线开始，之后可以跟字母、数字或下划线。
   * 操作符标识符（operator identifier）由一个或多个操作符字符组成。
   * 混合标识符（mixed identifier）：由字母数字组成，后面跟着下划线和一个操作符标识符。

如unary\_+被用作定义一元的“+”操作符的方法名。

或myvar\_=被用作定义赋值操作符的方法名。该混合标识符格式是由Scala编译器产生的用来支持属性（property）的。

* + 字面量标识符（literal identifier）：用反引号`…`包括的任意字符串。

如：`x` `<clinit>` `yield`

思路是你可以把运行时环境认可的任意字符串放在反引号之间当做标识符。

在Java的Thread类中访问静态的yield方法是它的典型用例。你不能写Thread.yield()，因为yield是Scala的保留字。但可以在反引号里引用方法的名称，例如：Thread.`yield`()。

1. Scala遵循Java的驼峰式标识符习惯。
2. 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。如标识符:->将被内部表达为 $colon$minus$greater。若你想从Java代码访问这个标识符，就应使用这种内部表达式方式。
3. 注意：

不建议在标识符结尾使用下划线。比如：“val name\_: Int = 1”，将导致编译器错误。编译器会认为你在定义名为“name\_:”的变量。需要在冒号前插入额外的空格，才能编译通过。

## Scala表达式的操作符

### 优先级和关联性

《Scala编程》：P90。

1. 操作符标注不仅限于像+这种其他语言里看上去像操作符的东西，任何方法都可以被当做操作符来标注。
2. 任何方法都可以是操作符：

Scala里的操作符不是特殊的语法：任何方法都可以是操作符。到低是方法还是操作符取决于你如何使用它。即是否以操作符标注方式使用它。

1. 操作符标注方式：《Scala编程》P85
   * 前缀标注：这些前缀操作符与中缀操作符一致，是值类型对象调用方法的简写形式。然而这种情况下，方法名在操作符字符上前缀“unary\_”。

标识符中能作为前缀操作符用的只有+、-、!和~。因此，如果对类型定义了名为unary\_!的方法，就可以对值或变量用!p这样的前缀操作符方式调用方法。

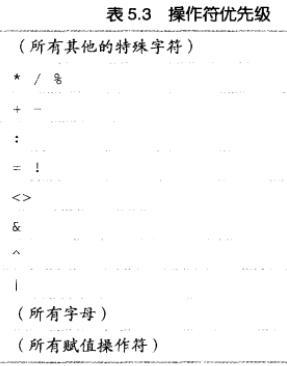
如果定义名为unary\_\*的方法，由于\*不是四种可以当做前缀操作符用的标识符之一，因此没有办法将其用成前缀操作符。—— \*p，Scala会理解成\*.p。

可以通过操作符和显示方法名两种方式调用方法。

* + 中缀标注
  + 后缀标注：后缀操作符是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。

1. 要想知道Scala的值类型有哪些操作符，只要在Scala的API文档里查下定义在值类型上的方法即可。
2. 操作符的优先级决定了表达式的哪个部分先于其他部分被评估。
3. Scala没有操作符，实际上，操作符只是方法的一种表达式。
4. 对于以操作符形式使用的方法，Scala根据操作符的第一个字符判断方法的优先级（这个规则有个例外）。



1. 除以上优先级规则外，还有以等号结束的赋值操作符（assignment operator）。如果操作符以等号字符（=）结束，且操作符并非比较操作符，那么这个 操作符的优先级与赋值操作符（=）相同。也就是说，它比任何其他操作符的优先级都低。
2. 当同样优先级的多个操作符并列出现在表达式时，操作符的关联性（associativity）决定了操作符分组的方式。
3. Scala里操作符的关联性取决于它的最后一个字符。
4. 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。任何以“:”字符结尾的方法由它的右操作数调用，并传入左操作符。以其他字符结尾的方法与之相反。

# 类

## 基础概念

1. val、var ： 定义成员变量。
2. def ： 定义成员方法，包含可执行代码。
3. 保持对象健壮性的重要方法之一就是保证对象的状态——即实例变量的值——在对象整个生命周期中持续有效。
4. 基本类型：除了String归于java.lang包之外，其余所有的基本类型都是包scala的成员。
5. Scala的基本类型与Java的对应类型范围完全一样。
6. Scala编译器将把类内部的任何既不是字段也不是方法定义的代码编译至主构造器中。
7. toString方法：类会继承java.lang.Object类的toString实现，只会打印类名、@符号和十六进制数。
8. 先决条件：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。

为主构造器定义先决条件（precondition）。

在scala包的孤立对象Predef上定义的require方法，

1. 自指向：关键字this指向当前执行方法被调用的对象实例，或者如果使用在构造器里的话，就是正被构建的对象实例。
2. 辅助构造器（auxiliary constructor）：Scala里主构造器之外的构造器被称为辅助构造器。

Scala的辅助构造器定义开始于def this(…)。

Scala里的每个辅助构造器的第一个动作都是调用同类的别的构造器。

规则的根本结果是：每个Scala的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。

比较：

* Java的构造器的第一个动作只有两个选择：要么调用同类的其他构造器，要么直接调用超类的构造器。
* Scala的类里面只有主构造器可以调用超类的构造器。

## 嵌套与作用域（scope）

1. 嵌套与作用域原则可以应用于所有的Scala架构，包括函数。
2. 类的主构造器：

主构造器的参数在该类范围内有效，但对外无效。

添加val、var后，主构造器的参数会自动变成自动，此时，对外有效。

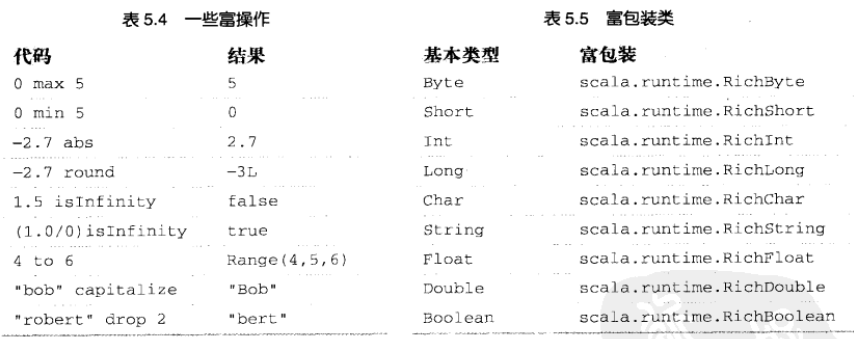
即，这些参数作为字段时才能被访问。

《Scala编程》：P96

1. 与Java的差异：Scala允许在嵌套范围内定义同名变量。
2. Scala程序里所有的变量定义都存在有效作用范围（scope）。
3. 花括号通常引入了新的作用范围，所以任何定义在花括号里的东西超出括号之后就脱离了范围。
4. 在解释器里：概念上，解释器为每次输入的新语句都创建了新的嵌套范围。

Harli-扩展：参见Ruby的解释器。

## 富包装类



## Scala的原字符串和符号字符串。

1. Scala为原始字符串（raw string）引入了一种特殊的语法。它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无论何种任意字符，包括新行、引号和特殊字符。

同时字符串类引入了stripMargin方法。使用的方式是，把管道符号（|）放在每行前面，然后对整个字符串调用stripMargin。

1. 符号字面量：被写成’<标识符>，这里<标识符>可以是任何字母或数字的标识符。

这种字面量被映射成预定义类scala.Symbol的实例。

具体地说，就是字面量’cymbal将被编译器扩展为工厂方法调用：Symbol(“cymbal”)。

符号字面量典型的应用场景是在动态类型语言中实用一个标识符。

《Scala编程》P83。

符号字面量除了显示名字之外，什么都不能做。

还有就是符号是被限定（interned）的。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。

Harli — 扩展： 参考ruby中的符号。

## 方法

1. 如果没有发现任何显式的返回语句，Scala方法将返回方法中最后一次计算得到的值。

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。待之以把每个方法当做是创建返回值的表达式。这种逻辑鼓励你分层简化方法，把较大的方法分解为多个更小的方法。

1. 另一种简写方式：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。
2. 另一种表达方式是去掉结果类型和等号，把方法体放在花括号里。在这种形式下，方法看上去很像过程（procedure），一种仅为了副作用而执行的方法。

通常我们定义副作用为能够改变方法之外的某处状态或执行I/O活动的方法。

比较容易出错的地方是如果去掉方法体前面的等号，那么方法的结果类型就必定是Unit。这种说法不论方法体里包含什么都成立，因为Scala编译器可以把任何类型转换为Unit。

就是说，带有花括号但没有等号的，本质上当做Unit结果类型的方法。

1. 如果去掉结果类型，但有等号的话，结果类型可以自动推导。

## Object

1. Scala比Java更为面向对象的特点之一是Scala不能定义静态成员，而是待之以定义单例对象（singleton object）。

除了用object关键字替换了class关键字之外，单例对象的定义看上去与类定义一致。

1. 不与伴生类共享名称的单例对象被称为独立对象（standalone object）。它可以用在很多地方，例如作为相关功能方法的工具类，或者定义scala应用的入口点。
   * 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。

Harli：问题，共享名称但又不在同一个文件，即不是伴生对象的单例对象是啥？？？

1. 伴生对象与伴生类

当单例对象与某个类共享同一个名称时，它就被称为是这个类的伴生对象（companion object）。

类和它的伴生对象必须定义在一个源文件里。

类被 称为是这个单例对象的伴生类（companion class）。

类和它的伴生对象可以互相访问其私有成员。

1. 对于Java程序员来说，可以把单例对象当做是Java中可能会用到的静态方法 工具类。也可以用类似的语法做方法调用：单例对象名，点，方法名。
2. 然而单例对象不只是静态方法的工具类。它同样是头等的对象。因此单例对象的名字可以被看做是贴在对象上的“名签”。
3. 定义单例对象并没有定义类型（在Scala的抽象层次上说）。
4. 或者可以认为，单例对象的类型是由单例对象的伴生类定义的。然而，单例对象扩展了父类并可以混入特质。因此，可以使用类型调用单例对象的方法，或者用类型的实例变量指代单例对象，并把它传递给需要类型参数的方法。
5. 类和单例对象间的差别是，单例对象不带参数，而类可以。因为单例对象不是用new关键字实例化的，所以没有机会传递给它实例化参数。
6. 每个单例对象都被实现为虚构类（synthetic class — 类名为对象名上加$）的实例，并指向静态的变量，因此它们与Java静态类有这相同的初始化语义。

Harli — 扩展：静态类是指在一个类的内部，又定义了一个用static修饰的类。那静态类的功能又体现在哪里呢？可以用C中的结构体内嵌结构体来理解，其次需要了解2个概念：内部类和静态修饰符static。  
A，首先，用内部类是因为内部类与所在外部类有一定的关系，往往只有该外部类调用此内部类。所以没有必要专门用一个Java文件存放这个类。

B，静态都是用来修饰类的内部成员的。比如静态方法、静态成员变量。它唯一的作用就是随着类的加载（而不是随着对象的产生）而产生，以致可以用类名+静态成员名直接获得。这样静态内部类就可以理解了，它可以直接被用 外部类名+内部类名 获得。

1. 特别要指出的是，单例对象在第一次被访问的时候才会被初始化。

## Application特质

1. Scala提供了特质scala.Application，可以减少一些输入工作。可以把 想要执行的代码直接放在单例对象的花括号之间。
2. 特质Application声明了带有合适签名的main方法，并被你写的单例对象继承，使它可以像Scala程序那样。花括号之间的代码被收集进了单例对象的主构造器（primary constructor），并在类被初始化时执行。
3. 缺点：
   * 如果想访问命令行参数的话即不能用它，因为args数组不可访问。
   * 因为某些JVM线程模型里的局限，如对于多线程的程序需要自行编写main方法；
   * 某些JVM的实现没有优化被Application特质执行的对象的初始化代码。

因此，只有当程序相对简单并且是单线程的情况下才可以继承Application特质。

扩展：JVM线程模型、JVM的优化。

## 访问修饰符

1. Scala的默认访问级别：public。

# 语法

1. 占位符语法：下划线\_，这是通常在Scala里作为占位符，来表示未知值的通配符。
2. 不再使用break和continue。
3. 在Scala里有时候可以用花括号代替小括号。For表达式的可选语法就是这种用法的一个例子。

## 递归调用

1. 尾调用优化——尾递归：递归调用在尾调用（tail-call）位置，编译器会产生出于while循环类似的代码。每个递归调用将被实现为回到函数开始位置的跳转。

## For表达式

1. 发生器（generator）的语法：“xxx <- xxxSeq” —— 参见for表达式。

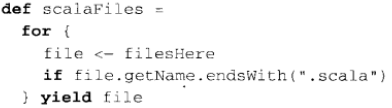
如果加入多个 <- 子句，就得到了嵌套的“循环”。

1. 过滤器（filter）：for表达式中的if字句。
2. 注意：如果在发生器中加入超过一个过滤器，if字句必须用分号分隔。
3. 可以使用花括号代替小括号包裹发生器和过滤器。使用花括号的好处是可以省略使用小括号时必须加的分号。
4. 流间（mid-stream）变量绑定：如：for表达式里的流间赋值。变量从半路引入for表达式。

《Scala编程》P111

1. for-yield表达式的语法：for { 子句 } yield { 循环体 }

制造新集合：在for表达式之前加上关键字yield。每次循环产生一个file值。



## try表达式与异常

1. 把抛出的异常当做任何类型的值都是安全的。任何使用经throw返回值的尝试都不会起作用，因此这样做不会有害处。
2. 从技术角度上，抛出异常的类型是Nothing。
3. 尽管throw不实际产生任何值，但你还是可以把他当做表达式。
4. 选择catch子句这种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。
5. Scala与Java的差别在于不需要补货检查异常（checked exception），或把它们声明在throws子句中。可以用注解声明@throws子句，但不是必须的。
6. Finally子句 —— Scala可以使用另一种被称为出借模式（loan pattern）的技巧更简洁地达到同样的目的。
7. Try-catch-finally也产生值：
   * 没有异常抛出：对应try子句；
   * 抛出异常被捕获：对应相应的catch子句；
   * 抛出异常单没有被捕获：表达式没有值。

由finally子句计算得到的值，即使有也会被抛弃。—— 即finally不应该修改主函数体或catch子句中计算的值。

与Java对比：Scala的行为差异仅在于Java的try-finally不产生值。

在Java中，如果finally子句也包含返回语句，或抛出异常，这个返回值或异常将“凌驾”于任何之前的try子句或catch子句里产生的值或异常之上。