Scala总结

--2015-1-19 12:33:54

本文档针对scala 2.10.x，由于scala目前发展迅速，因此可能会和其他版本的不同。

===概述

scala是一门以java虚拟机（JVM）为目标运行环境并将面向对象和函数式编程的最佳特性结合在一起的静态类型编程语言。

scala是纯粹的面向对象的语言。java虽然是面向对象的语言，但是它不是纯粹的，因为java的基本数据类型不是类，并且在java中还有静态成员变量和静态方法。相反，scala是纯粹面向对象的，每个值都是对象，每个操作都是方法调用。

scala也是一个成熟的函数式语言。函数式编程有两个指导思想：①函数是头等值，也就是说函数也是值，并且和其他类型（如整数、字符串等）处于同一地位，函数可以被当作参数传递，也可以被当作返回值返回，还可以在函数中定义函数等等；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，也就是说函数调用不应产生副作用，虽然函数式编程语言鼓励使用“无副作用”的方法，但是scala并不强制你必须这么做。scala允许你使用指令式的编程风格，但是随着你对scala的深入了解，你可能会更倾向于一种更为函数式的编程风格。向函数式编程转变，你就应该尽量去使用val、不可变对象、无副作用方法，而不是var、可变对象、有副作用方法。要明白的是，从指令式编程向函数式编程的转变会很困难，因此你要做好充分的准备，并不断的努力。

scala运行于JVM之上，并且它可以访问任何的java类库并且与java框架进行互操作，scala也大量重用了java类型和类库。

第一个scala程序：

object ScalaTest {

def main(args: Array[String]) {

println("hello scala.")

}

}

===scala解释器

安装好scala并配置好PATH环境变量之后，就可以在终端中输入“scala”命令打开scala解释器。在其中，你可以像使用shell一样，使用TAB补全、Ctrl+r搜索、上下方向键切换历史命令等等。退出scala解释器，可以使用命令：“:q”或者“:quit”。

由于解释器是输入一句执行一句，因此也常称为REPL。REPL一次只能看到一行代码，因此如果你要在其中粘贴代码段的话，可能会出现问题，这时你可以使用粘贴模式，键入如下语句：

:paste

然后把代码粘贴进去，再按下Ctrl+d，这样REPL就会把代码段当作一个整体来分析。

===scala作为脚本运行

scala代码也可以作为脚本运行，只要你设置好代码文件的shell前导词（preamble），并将代码文件设置为可执行。如下：

#!/usr/bin/env scala

println("这是scala脚本")

设置代码文件为可执行，即可执行之啦。

scala脚本的命令行参数保存在名为args的数组中，你可以使用args获取命令行输入的程序参数。

===scala编译运行

scala编译器scalac会将scala代码编译为jvm可以运行的字节码，然后就可以在jvm上执行了。假设有一个Hello.scala 文件，我们就可以使用 scalac Hello.scala 编译，然后使用 scala Hello 运行。当然也可以使用java工具来运行，但需要在classpath里指定scala-library.jar。对于classpath，在Unix家族的系统上，类路径的各个项目由冒号“:”分隔，在MS Windows系统上，它们由分号“;”分隔。例如，在linux上你可以输入这样的命令来运行（注意classpath最后加一个“.”）：

java -classpath /usr/local/scala-2.10.4/lib/scala-library.jar:. Hello

===scala IDE开发环境

你可以使用 eclipse 或者 intellij idea 作为scala的IDE开发环境，但都需要安装scala插件才行。下面分别介绍这两种方式：

eclipse开发环境配置：

scala ide for eclipse（下载地址：http://scala-ide.org）中集成了scala插件，你可以直接使用它进行开发，不过它包含的可能不是我们想要的scala版本，因此，还是在该网站上下载对应的scala插件，插在eclipse上，这样更好啊。

我们先安装eclipse juno，然后下载eclipse juno以及scala 2.10.4对应的scala sdk插件升级包：update-site.zip。将插件解压缩，将 features 和 plugins 目录下的所有东东都复制到eclipse中的对应目录中，重启eclipse即可。然后就可以新建scala project了。

intellij idea开发环境配置：

我们先安装好intellij idea，然后安装scala插件，自动安装插件有时会非常慢，尤其是在china。我们还是手动配置插件吧。请注意插件的版本，必须与当前idea版本兼容。手动配置插件方法如下：

(1) 进入 setting > plugins > browse repositorits 搜索你要下载的插件名称，右侧可以找到下载地址。

(2) 解压插件压缩包，把插件的全部文件都复制到IntelliJ IDEA安装程序的plugins文件夹中，注意插件最好以一个单独的文件夹放在plugins目录下。

(3) 一般重启intellij idea就会自动加载插件，进入 setting > plugins 看看有木有。如果不自动加载的话，进入setting > plugins > install plugin from disk，找到刚才复制的插件位置，再然后就好了。

接下来就可以新建scala project，新建时我选择的是“Scala”（不是sbt，因为我这选择sbt之后，等半天sbt都不会配置好，郁闷啊）。

相关姿势：

什么是SBT？ SBT = (not so) Simple Build Tool，是scala的构建工具，与java的maven地位相同。其设计宗旨是让简单的项目可以简单的配置，而复杂的项目可以复杂的配置。

===scala特点

在scala中，语句之后的“;”是可选的，这根据你的喜好。当有多个语句在同一行时，必须加上分号，但不建议把多个语句放在一行。

在scala中，建议使用2个空格作为代码缩进，不过我咋喜欢一个tab呢 ⊙﹏⊙!

在scala中，符号“\_”相当于java中的通配符“\*”。

scala类似于c++、java，索引也是从0开始，但元组是个例外，它从1开始。

===数据类型

scala有7种数值类型：Byte、Char、Short、Int、Long、Float和Double，以及2种非数值类型：Boolean和Unit（只有一个值“()”，相当于java和c++中的void，即空值）。这些类型都是抽象的final类(不能使用new新建，也不能被继承)，在scala包中定义，是对java基本数据类型的包装，因此与java基本数据类型有相同的长度。同时，scala还提供了RichInt、RichChar等等，它们分别提供Int、Char等所不具备的便捷方法。

另外，scala沿用了java.lang包中的String。在scala中，常量也称作字面量，字符串字面量由双引号包含的字符组成，同时scala提供了另一种定义字符串常量的语法——原始字符串，它以三个双引号作为开始和结束，字符串内部可以包含无论何种任意字符。

在scala中，我们使用方法，而不是强制类型转换，来做数值类型之间的转换，如99.44.toInt、97.toChar。另外也可以参见显式类型转换和隐式转换。

===变量

scala有两种变量：val和var。val如同java中的final变量，var如同java中的非final变量。由于scala是完全面向对象的，因此val和var只是声明了对象的引用是不可变的还是可变的，并不能说明引用指向的对象的可变性。声明变量的同时需要初始化之，否则该变量就是抽象的。如果不指定变量的类型，编译器会从初始化它的表达式中推断出其类型。当然你也可以在必要的时候指定其类型，但注意，在scala中变量或函数的类型总是写在变量或函数的名称的后边。示例如下：

val answer = “yes”

val answer, message: String = “yes”

===标识符

scala标识符有四种形式：字母数字标识符、操作符标识符、混合标识符、字面量标识符。

字母数字标识符：跟其他语言类似，由字母、数字和下划线组成，但需注意“$”字符被保留作为scala编译器产生的标识符之用，你不要随意使用它啊。

操作符标识符：由一个或多个操作符字符组成。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。若你想从java代码中访问这个标识符，就应该使用这种内部表示方式。

混合标识符：由字母数字以及后面跟着的下划线和一个操作符标识符组成。如unary\_+定义了一个前缀操作符“+”。

字面量标识符：是用反引号`…`包含的任意字符串，scala将把被包含的字符串作为标识符，即使被包含字符串是scala的关键字。例如：你可以使用Thread.`yield`()来访问java中的方法，即使yield是scala的关键字。

===操作符

scala的操作符和你在java和C++中的预期效果是一样的，但注意scala并不提供++、--操作符。不过，scala中的操作符实际上都是方法，任何方法都可以当作操作符使用，如 a + b 相当于 a.+(b)。

需要注意的是：对于不可变对象(注：对象的不可变并不是说它的引用变量是val的)，并不真正支持类似于“+=”这样以“=”结尾的操作符(即方法)，不过scala还是提供了一些语法糖，用以解释以“=”结尾的操作符用于不可变对象的情况。假设a是不可变对象的引用，那么在scala中a += b将被解释为a = a + b，这时就相当于新建一个不可变对象重新赋值给引用a，前提是引用变量a要声明为var的，因为val变量定义之后是不可变的。

更多信息参见函数（方法）部分。

===块表达式与赋值

在scala中，{}块包含一系列表达式，其结果也是一个表达式，块中最后一个表达式的值就是其值。

在scala中，赋值语句本身的值是Unit类型的。因此如下语句的值为“()”：

{r = r \* n; n -= 1}

正是由于上述原因，scala中不能多重赋值，而java和c++却可以多重赋值。因此，在scala中，如下语句中的x值为“()”：

x = y = 1

===控制结构

scala和其他编程语言有一个根本性差异：在scala中，几乎所有构造出来的语法结构都有值。这个特性使得程序结构更加精简。scala内建的控制结构很少，仅有if、while、for、try、match和函数调用等而已。如此之少的理由是，scala从语法层面上支持函数字面量。

if表达式：

scala的if/else语法结构与java等一样，但是在scala中if/else表达式有值，这个值就是跟在if/esle后边的表达式的值。如下：

val s = if(x > 0) 1 else -1

同时注意：scala的每个表达式都有一个类型，比如上述if/esle表达式的类型是Int。如果是混合类型表达式，则表达式的类型是两个分支类型的公共超类型。String和Int的超类型就是Any。如果一个if语句没有else部分，则当if条件不满足时，表达式结果为Unit。如：

if(x > 0) 1

就相当于：

if(x > 0) 1 else ()

while循环：

scala拥有与java和c++中一样的while和do-while循环，while、do-while结果类型是Unit。

for表达式：

scala中没有类似于for(; ; )的for循环，你可以使用如下形式的for循环语句：

for(i <- 表达式)

该for表达式语法对于数组和所有集合类均有效。具体介绍如下：

枚举：for(i <- 1 to 10)，其中“i <- 表达式”语法称之为发生器，该语句是让变量i（注意此处循环变量i是val的（但无需你指定），该变量的类型是集合的元素类型）遍历表达式中的所有值。1 to 10产生的Range包含上边界，如果不想包含上边界，可以使用until。

过滤：也叫守卫，在for表达式的发生器中使用过滤器可以通过添加if子句实现，如：for(i <- 1 to 10 if i!=5)，如果要添加多个过滤器，即多个if子句的话，要用分号隔开，如：for(i <- 1 to 10 if i!=5; if i!=6)。

嵌套枚举：如果使用多个“<-”子句，你就得到了嵌套的“循环”，如：for(i <- 1 to 5; j <- 1 to i)。

流间变量绑定：你可以在for发生器以及过滤器等中使用变量保存计算结果，以便在循环体中使用，从而避免多次计算以得到该结果。流间变量绑定和普通变量定义相似，它被当作val，但是无需声明val关键字。

制造新集合：for(…) yield 变量/循环体，最终将产生一个集合对象，集合对象的类型与它第一个发生器的类型是兼容的。

实际上：for表达式具有等价于组合应用map、flatMap、filter和foreach这几种高阶函数的表达能力。实际上，所有的能够yield（产生）结果的for表达式都会被编译器转译为高阶方法map、flatMap及filter的组合调用；所有的不带yield的for循环都会被转译为仅对高阶函数filter和foreach的调用。正是由于这几个高阶函数支持了for表达式，所以如果一个数据类型要支持for表达式，它就要定义这几个高阶函数。有些时候，你可以使用for表达式代替map、flatMap、filter和foreach的显式组合应用，或许这样会更清晰明了呢。

scala中没有break和continue语句。如果需要类似的功能时，我们可以：

1) 使用Boolean类型的控制变量

2) 使用嵌套函数，你可以从函数当中return

3) ...

match表达式与模式匹配：

scala中没有switch，但有更强大的match。它们的主要区别在于：

1. 任何类型的常量/变量，都可以作为比较用的样本；
2. 在每个case语句最后，不需要break，break是隐含的；
3. 更重要的是match表达式也有值；
4. 如果没有匹配的模式，则MatchError异常会被抛出。

match表达式的形式为：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以及一到多个表达式，它们将在模式匹配过程中被计算。箭头符号“=>”隔开了模式和表达式。按照代码先后顺序，一旦一个模式被匹配，则执行“=>”后边的表达式((这些)表达式的值就作为match表达式的值)，后续case语句不再执行。示例如下：

a match {

case 1 => "match 1"

case \_ => "match \_"

}

match模式的种类如下：

1. 通配模式：可以匹配任意对象，一般作为默认情况，放在备选项最后，如：

case \_ =>

1. 变量模式：类似于通配符，可以匹配任意对象，不同的是匹配的对象会被绑定在变量上，之后就可以使用这个变量操作对象。所谓变量就是在模式中临时生成的变量，不是外部变量，外部变量在模式匹配时被当作常量使用，见常量模式。注意：同一个模式变量只能在模式中出现一次。
2. 常量模式：仅匹配自身，任何字面量都可以作为常量，外部变量在模式匹配时也被当作常量使用，如：

case "false" => "false"

case true => "truth"

case Nil => "empty list"

对于一个符号名，是变量还是常量呢？scala使用了一个简单的文字规则对此加以区分：用小写字母开始的简单名被当作是模式变量，所有其他的引用被认为是常量。如果常量是小写命名的外部变量，那么它就得特殊处理一下了：如果它是对象的字段，则可以加上“this.”或“obj.”前缀；或者更通用的是使用字面量标识符解决问题，也即用反引号“`”包围之。

1. 抽取器模式：抽取器机制基于可以从对象中抽取值的unapply或unapplySeq方法，其中，unapply用于抽取固定数量的东东，unapplySeq用于抽取可变数量的东东，它们都被称为抽取方法，抽取器正是通过隐式调用抽取方法抽取出对应东东的。抽取器中也可以包含可选的apply方法，它也被称作注入方法，注入方法使你的对象可以当作构造器来用，而抽取方法使你的对象可以当作模式来用，对象本身被称作抽取器，与是否具有apply方法无关。样本类会自动生成伴生对象并添加一定的句法以作为抽取器，实际上，你也可以自己定义一个任意其他名字的单例对象作为抽取器使用，以这样的方式定义的抽取器对象与样本类类型是无关联的。你可以对数组、列表、元组进行模式匹配，这正是基于抽取器模式的。
2. 类型模式：你可以把类型模式当作类型测试和类型转换的简易替代，示例如下：

case s: String => s.length

1. 变量绑定：除了独立的变量模式之外，你还可以把任何其他模式绑定到变量。只要简单地写上变量名、一个@符号，以及这个模式。

模式守卫：模式守卫接在模式之后，开始于if，相当于一个判断语句。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，只有在守卫返回true的时候匹配才算成功。

Option类型：scala为可选值定义了一个名为Option的标准类型，一个Option实例的值要么是Some类型的实例，要么是None对象。分离可选值最通常的办法是通过模式匹配，如下：

case Some(s) => s

case None => “?”

模式无处不在：在scala中，模式可以出现在很多地方，而不单单在match表达式里。比如：

1. 模式使用在变量定义中，如下：

val myTuple = (123, “abc”)

val (number, string) = myTuple

1. 模式匹配花括号中的样本序列(即备选项)可以用在能够出现函数字面量的任何地方，实质上，样本序列就是更普遍的函数字面量，函数字面量只有一个入口点和参数列表，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，参数被模式所特化。如下：

val withDefault: Option[Int] => String = {

case Some(x) => "is int"

case None => "?"

}

1. for表达式里也可以使用模式。示例如下：

for((number, string) <- myTuple) println(number + string)

模式匹配中的中缀标注：带有两个参数的方法可以作为中缀操作符使用，使用中缀操作符时实际上是其中一个操作数在调用操作符对应的方法，而另一个操作数作为方法的参数。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这样的中缀标注等价于op(p,q)，也就是说中缀标注符op被用做抽取器模式。

===函数

函数定义：

定义函数时，除了递归函数之外，你可以省略返回值类型声明，scala会根据=号后边的表达式的类型推断返回值类型，同时=号后边表达式的值就是函数的返回值，你无需使用return语句（scala推荐你使用表达式值代替return返回值，当然根据你的需要，也可以显式使用return返回值）。示例如下：

def abs(x: Double) = if(x >= 0) x else -x

def fac(n: Int) = {

var r = 1

for(i <- 1 to n) r = r \* i

r

}

对于递归函数必须指定返回值类型，如下：

def fac(n: Int) : Int = if(n <= 0 ) 1 else n \* fac(n-1)

但你要知道的是：声明函数返回类型，总是有好处的，它可以使你的函数接口清晰。因此建议不要省略函数返回类型声明。

函数体定义时有“=”时，如果函数仅计算单个结果表达式，则可以省略花括号。如果表达式很短，甚至可以把它放在def的同一行里。

去掉了函数体定义时的“=”的函数一般称之为“过程”，过程函数的结果类型一定是Unit。因此，有时定义函数时忘记加等号，结果常常是出乎你的意料的。

没有返回值的函数的默认返回值是Unit。

函数调用：

scala中，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，如果没有副作用就去掉括号。如果在函数定义时，省略了空括号，那么在调用时，就不能加空括号。另外，函数作为操作符使用时的调用形式参见相应部分。

函数参数：

一般情况下，scala编译器是无法推断函数的参数类型的，因此你需要在参数列表中声明参数的类型。对于函数字面量来说，根据其使用环境的不同，scala有时可以推断出其参数类型。

scala里函数参数的一个重要特征是它们都是val（这是无需声明的，在参数列表里你不能显式地声明参数变量为val），不是var，所以你不能在函数里面给参数变量重新赋值，这将遭到编译器的强烈反对。

重复参数：

在scala中，你可以指明函数的最后一个参数是重复的，从而允许客户向函数传入可变长度参数列表。要想标注一个重复参数，可在参数的类型之后放一个星号“\*”。例如：

def echo(args: String\*) = for(arg <- args) println(arg)

这样的话，echo就可以被零至多个String参数调用。在函数内部，重复参数的类型是声明参数类型的数组。因此，echo函数里被声明为类型“String\*”的args的类型实际上是Array[String]。然而，如果你有一个合适类型的数组，并尝试把它当作重复参数传入，会出现编译错误。要实现这个做法，你需要在数组名后添加一个冒号和一个\_\*符号，以告诉编译器把数组中的每个元素当作参数，而不是将整个数组当作单一的参数传递给echo函数，如下：

echo(arr: \_\*)

默认参数与命名参数：

函数的默认参数与java以及c++中相似，都是从左向右结合。另外，你也可以在调用时指定参数名。示例如下：

def fun(str: String, left: String = “[”, right: String = “]”) = left + str + right

fun(“hello”)

fun(“hello”, “<<<”)

fun(“hello”, left = “<<<”)

函数与操作符：

从技术层面上来说，scala没有操作符重载，因为它根本没有传统意义上的操作符。诸如“+”、“-”、“\*”、“/”这样的操作符，其实调用的是方法。方法被当作操作符使用时，根据使用方式的不同，可以分为：中缀标注（操作符）、前缀标注、后缀标注。

中缀标注：中缀操作符左右分别有一个操作数。方法若只有一个参数（实际上是两个参数，因为有一个隐式的this），调用的时候就可以省略点及括号。实际上，如果方法有多个显式参数，也可以这样做，只不过你需要把参数用小括号全部括起来。如果方法被当作中缀操作符来使用（也即省略了点及括号），那么左操作数是方法的调用者，除非方法名以冒号“:”结尾（此时，方法被右操作数调用）。另外，scala的中缀标注不仅可以在操作符中存在，也可以在模式匹配、类型声明中存在，参见相应部分。

前缀标注：前缀操作符只有右边一个操作数。但是对应的方法名应该在操作符字符上加上前缀“unary\_”。标识符中能作为前缀操作符用的只有+、-、!和~。

后缀标注：后缀操作符只有左边一个操作数。任何不带显式参数的方法都可以作为后缀操作符。

在scala中，函数的定义方式除了作为对象成员函数的方法之外，还有内嵌在函数中的函数，函数字面量和函数值。

嵌套定义的函数：

嵌套定义的函数也叫本地函数，本地函数仅在包含它的代码块中可见。

函数字面量：

在scala中，你不仅可以定义和调用函数，还可以把它们写成匿名的字面量，也即函数字面量，并把它们作为值传递。函数字面量被编译进类，并在运行期间实例化为函数值（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，如Function0是没有参数的函数，Function1是有一个参数的函数等等。每一个FunctionN特质有一个apply方法用来调用函数）。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中，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。

以下是对给定数执行加一操作的函数字面量：

(x: Int) => x + 1

其中，=>指出这个函数把左边的东西转变为右边的东西。在=>右边，你也可以使用{}来包含代码块。

函数值是对象，因此你可以将其存入变量中，这些变量也是函数，你可以使用通常的括号函数调用写法调用它们。如：

val fun = (x: Int) => x + 1

val a = fun(5)

有时，scala编译器可以推断出函数字面量的参数类型，因此你可以省略参数类型，然后你也可以省略参数外边的括号。如：

(x) => x + 1

x => x + 1

如果想让函数字面量更简洁，可以把通配符“\_”当作单个参数的占位符。如果遇见编译器无法识别参数类型时，在“\_”之后加上参数类型声明即可。如：

List(1,2,3,4,5).filter(\_ > 3)

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

部分应用函数：

你还可以使用单个“\_”替换整个参数列表。例如可以写成：

List(1,2,3,4,5).foreach(println(\_))

或者更好的方法是你还可以写成：

List(1,2,3,4,5).foreach(println \_)

以这种方式使用下划线时，你就正在写一个部分应用函数。部分应用函数是一种表达式，你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。如下先定义一个函数，然后创建一个部分应用函数，并保存于变量，然后该变量就可以作为函数使用：

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

val a = sum \_

println(a(1,2,3))

实际发生的事情是这样的：名为a的变量指向一个函数值对象，这个函数值是由scala编译器依照部分应用函数表达式sum \_，自动产生的类的一个实例。编译器产生的类有一个apply方法带有3个参数（之所以带3个参数是因为sum \_表达式缺少的参数数量为3），然后scala编译器把表达式a(1,2,3)翻译成对函数值的apply方法的调用。你可以使用这种方式把成员函数和本地函数转换为函数值，进而在函数中使用它们。不过，你还可以通过提供某些但不是全部需要的参数表达一个部分应用函数。如下，此变量在使用的时候，可以仅提供一个参数：

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

如果你正在写一个省略所有参数的部分应用函数表达式，如println \_或sum \_，而且在代码的那个地方正需要一个函数，你就可以省略掉下划线（不是需要函数的地方，你这样写，编译器可能会把它当作一个函数调用，因为在scala中，调用无副作用的函数时，默认不加括号）。如下代码就是：

List(1,2,3,4,5).foreach(println)

闭包：

闭包是可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。比如说，在函数字面量中使用定义在其外的局部变量，这就形成了一个闭包。如下代码foreach中就创建了一个闭包：

var sum = 0

List(1,2,3,4,5).foreach(x => sum += x)

在scala中，闭包捕获了变量本身，而不是变量的值。变量的变化在闭包中是可见的，反过来，若闭包改变对应变量的值，在外部也是可见的。

尾递归：

递归调用这个动作在最后的递归函数叫做尾递归。scala编译器可以对尾递归做出重要优化，当其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头的跳转。

你可以使用开关“-g:notailcalls”关掉编译器的尾递归优化。

别高兴太早，scala里尾递归优化的局限性很大，因为jvm指令集使实现更加先进的尾递归形式变得困难。尾递归优化限定了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。

在scala中，你不要刻意回避使用递归，相反，你应该尽量避免使用while和var配合实现的循环。

高阶函数：

带有其他函数作为参数的函数称为高阶函数。

柯里化：

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且返回结果的新函数的技术。如下就是一个柯里化之后的函数：

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

这里发生的事情是当你调用curriedSum时，实际上接连调用了两个传统函数。第一个调用的函数带单个名为x的参数，并返回第二个函数的函数值；这个被返回的函数带一个参数y，并返回最终计算结果。你可以使用部分应用函数表达式方式，来获取第一个调用返回的函数，也即第二个函数，如下：

val onePlus = curriedSum(3)\_

高阶函数和柯里化配合使用可以提供灵活的抽象控制，更进一步，当函数只有一个参数时，在调用时，你可以使用花括号代替小括号，scala支持这种机制，其目的是让客户程序员写出包围在花括号内的函数字面量，从而让函数调用感觉更像抽象控制，不过需要注意的是：花括号也就是块表达式，因此你可以在其中填写多个表达式，但是最后一个表达式的值作为该块表达式的值并最终成为了函数参数。如果函数有两个以上的参数，那么你可以使用柯里化的方式来实现函数。

传名参数：

对于如下代码，myAssert带有一个函数参数，该参数变量的类型为不带函数参数的函数类型：

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

if(!predicate())

throw new AssertionError

}

在使用时，我们需要使用如下的语法：

myAssert(() => 5 > 3)

这样很麻烦，我们可以使用如下称之为“传名参数”的语法简化之：

myAssert(predicate: => Boolean) = {

if(!predicate)

throw new AssertionError

}

以上代码在定义参数类型时是以“=>”开头而不是“() =>”，并在调用函数（通过函数类型的变量）时，不带“()”。现在你就可以这样使用了：

myAssert(5 > 3)

其中，“predicate: => Boolean”说明predicate是函数类型，在使用时传入的是函数字面量。注意与“predicate: Boolean”的不同，后者predicate是Boolean类型的(表达式)。

偏函数：

偏函数和部分应用函数是无关的。偏函数是只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]来表示。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，以后可以再对定义域进行修改。PartialFunction中可以使用的方法如下：

isDefinedAt：判断定义域是否包含指定的输入。

orElse：补充对其他域的定义。

compose：组合其他函数形成一个新的函数，假设有两个函数f和g，那么表达式f \_ compose g \_则会形成一个f(g(x))形式的新函数。你可以使用该方法对定义域进行一定的偏移。

andThen：将两个相关的偏函数串接起来，调用顺序是先调用第一个函数，然后调用第二个，假设有两个函数f和g，那么表达式f \_ andThen g \_则会形成一个g(f(x))形式的新函数，刚好与compose相反。

===类(class)和对象(object)

类(class)和构造器：

类的定义形式如下：

class MyClass(a: Int, b: Int) {

println(a.toString)

}

在scala中，类也可以带有类参数，类参数可以直接在类的主体中使用，没必要定义字段然后把构造器的参数赋值到字段里，但需要注意的是：类参数仅仅是个参数而已，不是字段，如果你需要在别的地方使用，就必须定义字段。不过还有一种称为参数化字段的定义形式，可以简化字段的定义，如下：

class MyClass(val a: Int, val b: Int) {

println(a.toString)

}

以上代码中多了val声明，作用是在定义类参数的同时定义类字段，不过它们使用相同的名字罢了。类参数同样可以使用var作前缀，还可以使用private、protected、override修饰等等。scala编译器会收集类参数并创造出带同样的参数的类的主构造器，并将类内部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。除了主构造器，scala也可以有辅助构造器，辅助构造器的定义形式为def this(…)。每个辅助构造器都以“this(…)”的形式开头以调用本类中的其他构造器，被调用的构造器可以是主构造器，也可以是源文件中早于调用构造器定义的其他辅助构造器。其结果是对scala构造器的调用终将导致对主构造器的调用，因此主构造器是类的唯一入口点。在scala中，只有主构造器可以调用超类的构造器。

你可以在类参数列表之前加上private关键字，使类的主构造器私有，私有的主构造器只能被类本身以及伴生对象访问。

可以使用require方法来为构造器的参数加上先决条件，如果不满足要求的话，require会抛出异常，阻止对象的创建。

如果类的主体为空，那么可以省略花括号。

访问级别控制：

公有是scala的默认访问级别，因此如果你想使成员公有，就不要指定任何访问修饰符。公有的成员可以在任何地方被访问。

私有类似于java，即在之前加上private。不同的是，在scala中外部类不可以访问内部类的私有成员。

保护类似于java，即在之前加上protected。不同的是，在scala中同一个包中的其他类不能访问被保护的成员。

scala里的访问修饰符可以通过使用限定词强调。格式为private[X]或protected[X]的修饰符表示“直到X”的私有或保护，这里X指代某个所属的包、类或单例对象。

scala还有一种比private更严格的访问修饰符，即private[this]。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，这种限制被称为对象私有。这可以保证成员不被同一个类中的其他对象访问。

对于私有或者保护访问来说，scala的访问规则给予了伴生对象和类一些特权，伴生对象可以访问所有它的伴生类的私有成员、保护成员，反过来也成立。

成员(类型、字段和方法)：

scala中也可以定义类型成员，类型成员以关键字type声明。通过使用类型成员，你可以为类型定义别名。

scala里字段和方法属于相同的命名空间，scala禁止在同一个类里用同样的名称定义字段和方法，尽管java允许这样做。

getter和setter：

在scala中，类的每个非私有的var成员变量都隐含定义了getter和setter方法，但是它们的命名并没有沿袭java的约定，var变量x的getter方法命名为“x”，它的setter方法命名为“x\_=”。你也可以在需要的时候，自行定义相应的getter和setter方法，此时你还可以不定义关联的字段，自行定义setter的好处之一就是你可以进行赋值的合法性检查。

如果你将scala字段标注为@BeanProperty时，scala编译器会自动额外添加符合JavaBeans规范的形如getXxx/setXxx的getter和setter方法。这样的话，就方便了java与scala的互操作。

样本类：

带有case修饰符的类称为样本类(case class)，这种修饰符可以让scala编译器自动为你的类添加一些句法上的便捷设定，以便用于模式匹配，scala编译器自动添加的句法如下：

1. 帮你实现一个该类的伴生对象，并在伴生对象中提供apply方法，让你不用new关键字就能构造出相应的对象；
2. 在伴生对象中提供unapply方法让模式匹配可以工作；
3. 样本类参数列表中的所有参数隐式地获得了val前缀，因此它们被当作字段维护；
4. 添加toString、hashCode、equals、copy的“自然”实现。

封闭类：

带有sealed修饰符的类称为封闭类(sealed class)，封闭类除了类定义所在的文件之外不能再添加任何新的子类。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。这还意味你可以获得更好的编译器帮助。

单例对象(singleton object)：

scala没有静态方法，不过它有类似的特性，叫做单例对象，以object关键字定义（注：main函数也应该在object中定义，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）。定义单例对象并不代表定义了类，因此你不可以使用它来new对象。当单例对象与某个类共享同一个名称时，它就被称为这个类的伴生对象(companion object)。类和它的伴生对象必须定义在同一个源文件里。类被称为这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。不与伴生类共享名称的单例对象被称为独立对象(standalone object)。

apply与update：在scala中，通常使用类似函数调用的语法。当使用小括号传递变量给对象时，scala都将其转换为apply方法的调用，当然前提是这个类型实际定义过apply方法。比如s是一个字符串，那么s(i)就相当于c++中的s[i]以及java中的s.charAt(i)，实际上 s(i) 是 s.apply(i) 的简写形式。类似地，BigInt(“123”) 就是 BigInt.apply(“123”) 的简写形式，这个语句使用伴生对象BigInt的apply方法产生一个新的BigInt对象，不需要使用new。与此相似的是，当对带有括号并包含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，如arr(0) = “hello”将转换为arr.update(0, “hello”)。

类和单例对象之间的差别是，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，所以没机会传递给它实例化参数。单例对象在第一次被访问的时候才会被初始化。当你实例化一个对象时，如果使用了new则是用类实例化对象，无new则是用伴生对象生成新对象。同时要注意的是：我们可以在类或(单例)对象中嵌套定义其他的类和(单例)对象。

对象相等性：

与java不同的是，在scala中，“==”和“!=”可以直接用来比较对象的相等性，“==”和“!=”方法会去调用equals方法，因此一般情况下你需要覆盖equals方法。如果要判断引用是否相等，可以使用eq和ne。

在使用具有哈希结构的容器类库时，我们需要同时覆盖hashCode和equals方法，但是实现一个正确的hashCode和equals方法是比较困难的一件事情，你需要考虑的问题和细节很多，可以参见java总结中的相应部分。另外，正如样本类部分所讲的那样，一旦一个类被声明为样本类，那么scala编译器就会自动添加正确的符合要求的hashCode和equals方法。

===抽象类和抽象成员

与java相似，scala中abstract声明的类是抽象类，抽象类不可以被实例化。

在scala中，抽象类和特质中的方法、字段和类型都可以是抽象的。示例如下：

trait MyAbstract {

type T // 抽象类型

def transform(x: T): T // 抽象方法

val initial: T // 抽象val

var current: T // 抽象var

}

抽象方法：抽象方法不需要（也不允许）有abstract修饰符，一个方法只要是没有实现（没有等号或方法体），它就是抽象的。

抽象类型：scala中的类型成员也可以是抽象的。抽象类型并不是说某个类或特质是抽象的(特质本身就是抽象的)，抽象类型永远都是某个类或特质的成员。

抽象字段：没有初始化的val或var成员是抽象的，此时你需要指定其类型。抽象字段有时会扮演类似于超类的参数这样的角色，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。因此参数化特质的方式就是通过在子类中实现抽象字段完成。如对于以下特质：

trait MyAbstract {

val test: Int

println(test)

def show() {

println(test)

}

}

你可以使用如下匿名类语法创建继承自该特质的匿名类的实例，如下：

new MyAbstract {

val test = 1

}.show()

你可以通过以上方式参数化特质，但是你会发现这和“new 类名(参数列表)”参数化一个类实例还是有区别的，因为你看到了对于test变量的两次println(第一次在特质主体中，第二次是由于调用了方法show)，输出了两个不同的值(第一次是0，第二次是1)。这主要是由于超类会在子类之前进行初始化，而超类抽象成员在子类中的具体实现的初始化是在子类中进行的。为了解决这个问题，你可以使用预初始化字段和懒值。

预初始化字段：

预初始化字段，可以让你在初始化超类之前初始化子类的字段。预初始化字段用于对象或有名称的子类时，形式如下：

class B extends {

val a = 1

} with A

预初始化字段用于匿名类时，形式如下：

new {

val a = 1

} with A

需要注意的是：由于预初始化的字段在超类构造器调用之前被初始化，因此它们的初始化器不能引用正在被构造的对象。

懒值：

加上lazy修饰符的val变量称为懒值，懒值右侧的表达式将直到该懒值第一次被使用的时候才计算。如果懒值的初始化不会产生副作用，那么懒值定义的顺序就不用多加考虑，因为初始化是按需的。

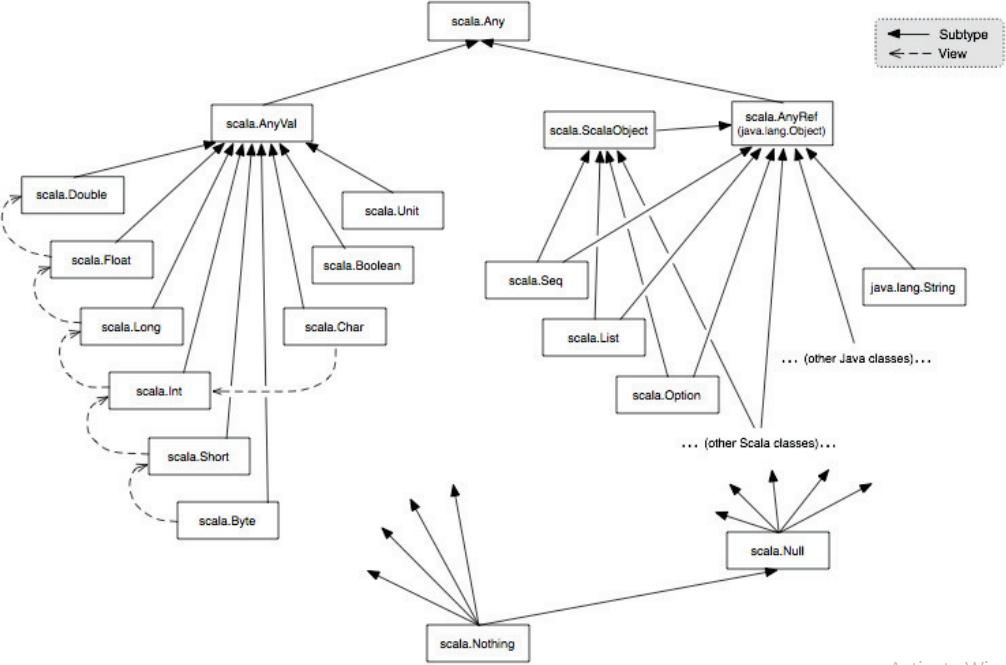
===继承与覆盖(override)

继承：

继承时，如果父类主构造器带有参数，子类需要把要传递的参数放在父类名之后的括号里即可，如下：

class Second(a: Int, b: Int) extends First(a) {…}

scala继承层级：



如上图所示：Any是所有其他类的超类。Null是所有引用类(继承自AnyRef的类)的子类，Null类型的值为null。Nothing是所有其他类(包括Null)的子类，Nothing类型没有任何值，它的一个用处是它标明了不正常的终止（例如抛出异常，啥也不返回）。AnyVal是scala中内建值类(共9个)的父类。AnyRef是scala中所有引用类的父类，在java平台上AnyRef实际就是java.lang.Object的别名，因此java里写的类和scala里写的类都继承自AnyRef，你可以认为java.lang.Object是scala在java平台上实现AnyRef的方式。scala类与java类的不同之处在于，scala类还继承了一个名为ScalaObject的特别记号特质，目的是想让scala程序执行得更高效。

覆盖：

由于scala里字段和方法属于相同的命名空间，这让字段可以覆盖无参数方法或空括号方法，但反过来好像不可以啊。另外，你也可以用空括号方法覆盖无参数方法，反之亦可。在scala中，若子类覆盖了父类的具体成员则必须带override修饰符；若是实现了同名的抽象成员时则override是可选的；若并未覆盖或实现基类中的成员则禁用override修饰符。

===特质(trait)

特质相当于接口，不能被实例化。特质定义使用trait关键字，与类相似，你同样可以在其中定义而不仅是声明字段和方法等。你可以使用extends或with将多个特质“混入”类中。注意当在定义特质时，使用extends指定了特质的超类，那么该特质就只能混入扩展了指定的超类的类中。

特质与类的区别在于：①特质不能带有“类参数”，也即传递给主构造器的参数；②不论在类的哪个地方，super调用都是静态绑定的，但在特质中，它们是动态绑定的，因为在特质定义时，尚且不知道它的超类是谁，因为它还没有“混入”，由于在特质中使用super调用超类方法是动态绑定的，因此你需要对特质中相应的方法加上abstract声明（虽然加上了abstract声明，但方法仍可以被具体定义，这种用法只有在特质中有效），以告诉编译器特质中的该方法只有在特质被混入某个具有期待方法的具体定义的类中才有效。你需要非常注意特质被混入的次序：特质在构造时顺序是从左到右，构造器的顺序是类的线性化（线性化是描述某个类型的所有超类型的一种技术规格）的反向。由于多态性，子类的方法最先起作用，因此越靠近右侧的特质越先起作用，如果最右侧特质调用了super，它调用左侧的特质的方法，依此类推。

Ordered特质：

Ordered特质扩展自java的Comparable接口。Ordered特质用于排序，为了使用它，你需要做的是：首先将其混入类中，然后实现一个compare方法。需要注意的是：Ordered并没有为你定义equals方法，因为通过compare实现equals需要检查传入对象的类型，但是因为类型擦除，导致它无法做到。因此，即使继承了Ordered，也还是需要自己定义equals。

Ordering特质：

Ordering特质扩展自java的Comparator接口。Ordering特质也用于排序，为了使用它，你需要做的是：定义一个该特质的子类的单独的实例，需要实现其中的compare方法，并将其作为参数传递给排序函数。此乃策略模式也。

Application特质：

特质Application声明了带有合适签名的main方法。但是它存在一些问题，所以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。Application特质相对于APP特质来说，有些陈旧，你应该使用更新的APP特质。

APP特质：

APP特质同Application特质一样，都提供了带有合适签名的main方法，在使用时只需将它混入你的类中，然后就可以在类的主构造器中写代码了，无需再定义main方法。如果你需要命令行参数，可以通过args属性得到。

===显式类型转换

正如之前所述的，scala中类型转换使用方法实现，以下是显式类型测试和显式类型转换的示例：

a.isInstanceOf[String] // 显式类型测试

a.asInstanceOf[String] // 显式类型转换

===隐式转换、隐式参数

隐式转换：

隐式转换只是普通的方法，唯一特殊的地方是它以修饰符implicit开始，implicit告诉scala编译器可以在一些情况下自动调用（比如说如果当前类型对象不支持当前操作，那么scala编译器就会自动添加调用相应隐式转换函数的代码，将其转换为支持当前操作的类型的对象，前提是已经存在相应的隐式转换函数且满足作用域规则），而无需你去调用（当然如果你愿意，你也可以自行调用）。隐式转换函数定义如下：

implicit def functionName(…) = {…}

隐式转换满足以下规则：

作用域规则：scala编译器仅会考虑处于作用域之内的隐式转换。隐式转换要么是以单一标识符的形式（即不能是aaa.bbb的形式，应该是bbb的形式）出现在作用域中，要么是存在于源类型或者目标类型的伴生对象中。

单一调用规则：编译器在同一个地方只会添加一次隐式操作，不会在添加了一个隐式操作之后再在其基础上添加第二个隐式操作。

显式操作先行规则：若编写的代码类型检查无误，则不会尝试任何隐式操作。

隐式参数：

柯里化函数的完整的最后一节参数可以被隐式提供，即隐式参数。此时最后一节参数必须被标记为implicit（整节参数只需一个implicit，并不是每个参数都需要），同时用来提供隐式参数的相应实际变量也应该标记为implicit的。对于隐式参数，我们需要注意的是：

1. 隐式参数也可以被显式提供；
2. 提供隐式参数的实际变量必须以单一标识符的形式出现在作用域中；
3. 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，因此隐式参数应该是很稀少或者很特殊的类型（最好是使用自定义的角色确定的名称来命名隐式参数类型），以便不会被碰巧匹配；
4. 如果隐式参数是函数，编译器不仅会尝试用隐式值补足这个参数，还会把这个参数当作可用的隐式操作而使用于方法体中。

视界：

视界使用“<%”符号，可以用来缩短带有隐式参数的函数签名。比如，“T <% Ordered[T]”是在说“任何的T都好，只要T能被当作Ordered[T]即可”，因此只要存在从T到Ordered[T]的隐式转换即可。

注意视界与上界的不同：上界“T <: Ordered[T”是说T是Ordered[T]类型的。

隐式操作调试：

隐式操作是scala的非常强大的特性，但有时很难用对也很难调试。

有时如果编译器不能发现你认为应该可以用的隐式转换，你可以把该转换显式地写出来，这有助于发现问题。

另外，你可以在编译scala程序时，使用“-Xprint:typer”选项来让编译器把添加了所有的隐式转换之后的代码展示出来。

===类型参数化

在scala中，类型参数化(类似于泛型)使用方括号实现，如：Foo[A]，同时，我们称Foo为高阶类型。如果一个高阶类型有2个类型参数，则在声明变量类型时可以使用中缀形式来表达，此时也称该高阶类型为中缀类型，示例如下：

class Foo[A,B]

val x: Int Foo String = null // Int Foo String 等同于 Foo[Int,String]

与java相似，scala的类型参数化也使用类型擦除实现(类型擦除是很差劲的泛型机制，不过可能是由于java的原因，scala也这样做了)，类型擦除的唯一例外就是数组，因为在scala中和java中，它们都被特殊处理，数组的元素类型与数组值保存在一起。在scala中，数组是“不变”的(这点与java不同)，泛型默认是“不变”的。

协变、逆变与不变：

拿Queue为例，如果S是T的子类型，那么Queue[S]是Queue[T]的子类型，就称Queue是协变的；相反，如果Queue[T]是Queue[S]的子类型，那么Queue是逆变的；既不是协变又不是逆变的是不变的，不变的又叫严谨的。

在scala中，泛型默认是不变的。当定义类型时，你可以在类型参数前加上“+”使类型协变，如Queue[+A]。类似地，你可以在类型参数前加上“-”使类型逆变。在java中使用类型时可以通过使用extends和super来达到协变逆变的目的，它们都是“使用点变型”，java不支持“声明点变型”。而scala中同时提供了声明点变型(“+”和“-”，它们只能在类型定义时使用)和使用点变型(“<:”和“>:”，类似于java中的extends和super，在使用类型时声明)。不管是“声明点变型”还是“使用点变型”，都遵循PECS法则，详见java泛型。需要注意的是：变型并不会被继承，父类被声明为变型，子类若想保持仍需要再次声明。

继承中的协变逆变：

c++、java、scala都支持返回值协变，也就是说在继承层次中子类覆盖超类的方法时，可以指定返回值为更具体的类型。c#不支持返回值协变。

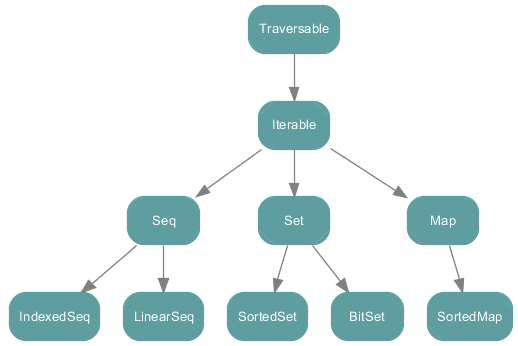
允许参数逆变的面向对象语言并不多——c++、java、scala和c#都会把它当成一个函数重载。

更多信息参见java泛型。

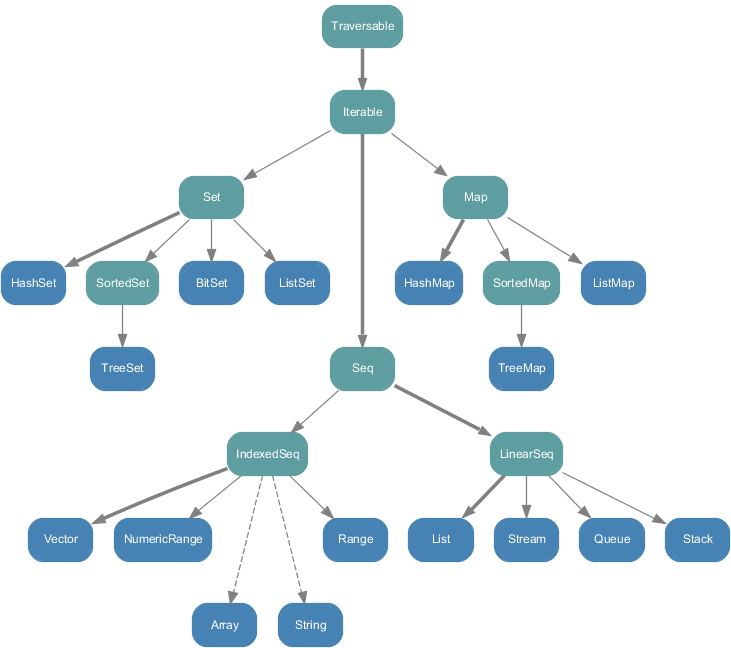
===集合

scala的集合（collection）库分为可变（mutable）类型与不可变（immutable）类型。以Set为例，特质scala.collection.immutable.Set和scala.collection.mutable.Set都扩展自scala.collection.Set。

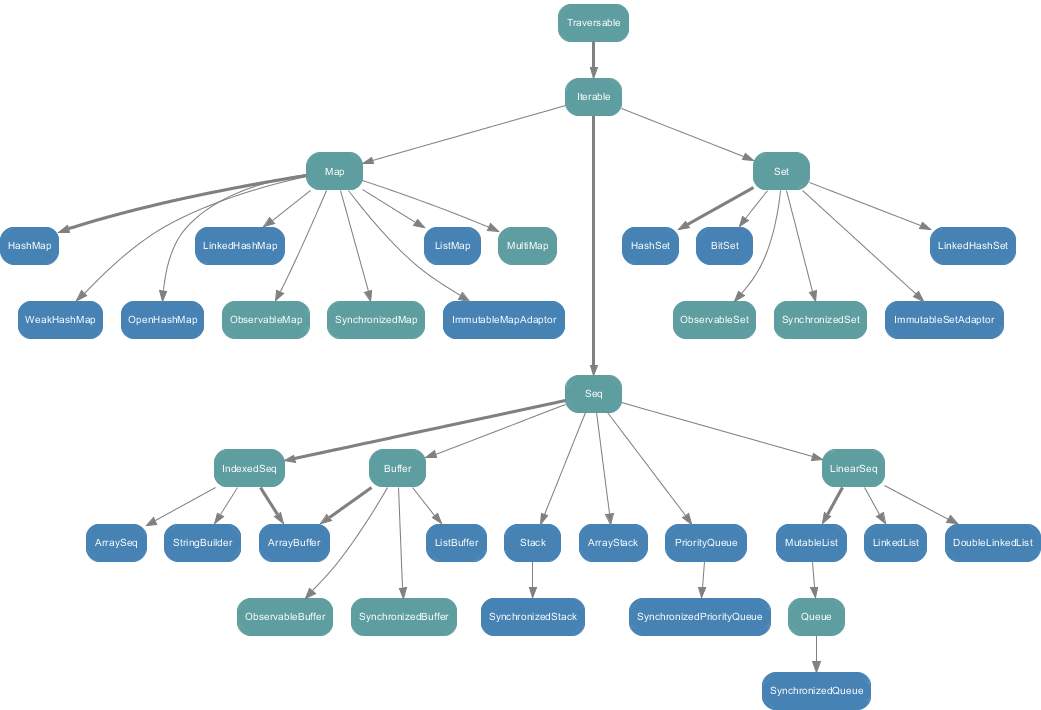
scala集合的顶层抽象类和特质：



scala.collection.immutable：



scala.collection.mutable：



不可变集合与可变集合之间的对应关系：

|  |  |
| --- | --- |
| 不可变（collection.immutable.\_） | 可变（collection.mutable.\_） |
| Array | ArrayBuffer |
| List | ListBuffer |
| String | StringBuilder |
| - | LinkedList, DoubleLinkedList |
| List | MutableList |
| Queue | Queue |
| Array | ArraySeq |
| Stack | Stack |
| HashMap HashSet | HashMap HashSet |
| - | ArrayStack |

Iterable与Iterator：

Iterable是可变和不可变序列、集、映射的超特质。集合对象可以通过调用iterator方法来产生迭代器Iterator。Iterable与Iterator之间的差异在于：前者指代的是可以被枚举的类型，而后者是用来执行枚举操作的机制。尽管Iterable可以被枚举若干次，但Iterator仅能使用一次。

数组：

在scala中，数组保存相同类型的元素，其中包含的元素值是可变的。数组也是对象，访问数组使用小括号。在JVM中，scala的数组以java数组方式实现。scala中数组是非协变的。

定长数组使用Array，创建之后长度不可改变。变长数组使用ArrayBuffer。

与java一样，scala中多维数组也是通过数组的数组来实现的。构造多维数组可以使用ofDim方法或者直接使用for循环来new。示例如下：

val matrix = Array.ofDim[Double](3,4) // ofDim方法创建多维数组

matrix(1)(2) = 12.36

val mutliarr = new Array[Array[Int]](10) // for循环方式创建多维数组

for(i <- 0 until mutliarr.length)

mutliarr(i) = new Array[Int](5)

列表：

列表保存相同类型的元素。scala里的列表类型是协变的，这意味着如果S是T的子类，那么List[S]也是List[T]的子类。

不可变列表使用List，一旦创建之后就不可改变。可变列表使用ListBuffer。

List是抽象类，它有两个子类型：Nil和::。Nil是空列表对象，类型是List[Nothing]。::是样本类，可以创建非空列表，::的伴生对象可以以中缀标注的形式用于模式匹配。所以在scala中存在两个::，一个是样本类，另一个是List的方法，因此在构造一个列表时，我们就有了多种方法，如下：

val list1 = List("A") // 这里List是伴生对象，相当于 List.apply()

val list2 = ::("A",Nil) // 这里::是伴生对象, 相当于 ::.apply()

val list3 = "A" :: Nil // 这里::是方法, 相当于 Nil.::()

List类没有提供append操作（向列表尾部追加），因为随着列表变长，效率将逐渐低下。List提供了“::”做前缀插入，因为这将消耗固定时间。如果你想通过添加元素来构造列表，你的选择是先把它们前缀插入，完成之后再调用reverse；或者使用ListBuffer，一种提供append操作的可变列表，完成之后调用toList。

栈和队列：

scala集合库提供了可变和不可变的栈类Stack，也提供了可变和不可变的队列类Queue。

元组与对偶：

元组Tuple也是不可变的，但元组可以包含不同类型的元素，并且因此而不能继承自Iterable。

元组实例化之后，可以使用点号、下划线和从1开始的索引访问其中的元素。因为元组可以保存不同类型的元素，所以不能使用apply方法访问其元素（apply返回同样的类型）。元组的索引从1开始，是因为对于拥有静态类型元组的其他语言，如Haskell和ML，从1开始是传统的设定。

scala的任何对象都可以调用“->”方法，并返回包含键值对的二元组（也叫对偶，是元组的最简单形态），比如 “hello” -> 100 则创建出 (“hello”, 100)。

元组相应操作示例如下：

val t = (1400, “Jim”, “haha”, 3.14) // 定义一个元组

val second = t.\_2 // 引用元组第二个组元

val (first, second, third, fourth) = t // 分别获取元组的第1、2、3、4个组元

val (first, secong, \_) = t // 只获取前两个组元

集和映射：

集中保存着不重复的元素。映射可以把键和值关联起来保存。

拉链操作：

val symbols = Array(“<”, “-”, “>”)

val counts = Array(2, 10, 2)

val pairs = symbols.zip(counts)

以上代码生成对偶类型的数组，如下：

Array((<,2), (-,10), (>,2))

可变集合vs不可变集合：

可变集合性能更好，不可变集合更易于理清头绪。对于某些问题来说，可变集合能够很好的处理；而另一些，不可变集合更为合适。如果在使用可变集合时，你发现需要担忧何时复制可变集合的副本，或者思考很多关于谁“主宰”或“拥有”可变集合的时候，那么请考虑是否可用不可变集合代替。

===异常

scala的异常工作机制与java的类似，但也有区别。区别如下：

1. scala没有“受检”异常——你不需要声明函数或方法可能会抛出某种异常。
2. throw表达式是有值的，其值是Nothing类型。
3. try-catch-finally表达式也是有值的，但是情况有些特殊。当没有抛出异常时，try子句为表达式值；如果抛出异常并被捕获，则对应于相应的catch子句；如果没有被捕获，表达式就没有返回值。finally子句计算得到的值，总是被抛弃（除非使用return语句），所以你应该在finally子句中干一些它应该干的事，比如说：关闭文件、套接字、数据库连接等，而最好别干什么其他事。

===断言、检查

scala里，断言使用assert函数，检查使用ensuring函数，如果条件不成立，它们将会抛出AssertionError。它们都在Predef中定义。你可以使用JVM的-ea和-da命令行标志来开放和禁止断言以及检查。

===包和引用

打包：

scala的代码采用了java平台完整的包机制。你可以使用两种方式把代码放进包里：

1. 使用放在文件顶部的package子句来把整个文件放入包中；
2. 使用package子句把要放入到包中的代码用花括号括起来，这种方式像C#的命名空间。使用这种方式，你可以定义出嵌套的包，注意：scala的包可以嵌套，java则不可以。任何你自己写的顶层包都被隐含地包含在\_root\_包中，因此你可以在多层嵌套的包代码中通过\_root\_来访问顶层包中的代码。

引用：

与java类似，scala使用import来引用，与java不同的是，scala的import子句：

1. 可以出现在任何地方，而不仅仅在文件开始处；
2. 可以引用对象和包；
3. 可以重命名或隐藏一些被引用的成员。这可以通过在被引用成员的对象之后加上括号里的引用选择器子句来做到，示例如下（令p为包名）：

import p.{x} // 从p中引入x，等价于 import p.x

import p.{x => y} // 从p中引入x，并重命名为y

import p.{x => \_, \_} // 从p中引入除了x之外的所有东东。注意单独的“\_”称作全包括，必须位于选择器的最后。import p.{\_} 等价于 import p.\_

隐式引用：

scala隐含地为每个源文件都加入如下引用：

import java.lang.\_

import scala.\_

import Predef.\_

包scala中的Predef对象包含了许多有用的方法。例如：通常我们所使用的println、readLine、assert等。

===scala I/O

由于scala可以和java互操作，因此目前scala中的I/O类库并不多，你可能需要使用java中的I/O类库。下面介绍scala中有的东东：

scala.Console对象可以用于终端输入输出，其中终端输入函数有：readLine、readInt、readChar等等，终端输出函数有：print、println、printf等等。其实，Predef对象中提供的预定义的readLine、println等等方法都是Console对象中对应方法的别名。

scala.io.Source可以以文本的方式迭代地读取源文件或者其他数据源。用完之后记得close啊。

对象序列化：

为了让对象可序列化，你可以这样定义类：

@SerialVersionUID(42L) class Person extends Serializable {…}

其中，@SerialVersionUID注解指定序列化ID，如果你能接受缺省的ID，也可省去该注解；Serializable在scala包中，因此你无需引入。你可以像java中一样对对象进行序列化。scala集合类都是可以序列化的，因此你可以把它们作为你的可序列化类的成员。

===Actor和并发

与java的基于共享数据和锁的线程模型不同，scala的actor包则提供了另外一种不共享任何数据、依赖消息传递的模型。设计并发软件时，actor是首选的工具，因为它们能够帮助你避开死锁和争用状况，这两种情形都是在使用共享和锁模型时很容易遇到的。

创建actor：

actor是一个类似于线程的实体，它有一个用来接收消息的邮箱。实现actor的方法是继承scala.actors.Actor特质并完成其act方法。你可以通过actor的start方法来启动它。actor在运行时都是相互独立的。你也可以使用scala.actors.Actor对象的actor方法来创建actor，不过此时你就无需再调用start方法，因为它在创建之后马上启动。

发送接收消息：

Actor通过相互发送消息的方式进行通信，你可以使用“!”方法来发送消息，使用receive方法来接收消息，receive方法中包含消息处理的模式匹配(偏函数)。发送消息并不会导致actor阻塞，发送的消息在接收actor的邮箱中等待处理，直到actor调用了receive方法，如果actor调用了receive但没有模式匹配成功的消息，那么该actor将会阻塞，直到收到了匹配的消息。创建actor并发送接收消息的示例如下：

object ScalaTest extends Actor {

def act() {

while (true) {

receive {

case msg => println(msg)

}

}

}

def main(args: Array[String]) {

start()

this ! "hello."

}

}

将原生线程当作actor：

Actor子系统会管理一个或多个原生线程供自己使用。只要你用的是你显式定义的actor，就不需要关心它们和线程的对应关系是怎样的。该子系统也支持反过来的情形：即每个原生线程也可以被当作actor来使用。此时，你应该使用Actor.self方法来将当前线程作为actor来查看，也就是说可以这样使用了：Actor.self ! "message"。

通过重用线程获取更好的性能：

Actor是构建在普通java线程之上的，如果你想让程序尽可能高效，那么慎用线程的创建和切换就很重要了。为帮助你节约线程，scala提供了react方法，和receive一样，react带有一个偏函数，不同的是，react在找到并处理消息后并不返回(它的返回类型是Nothing)，它在处理完消息之后就结束了。由于react不需要返回，故其不需要保留当前线程的调用栈。因此actor库可以在下一个被唤醒的线程中重用当前的线程。极端情况下，如果程序中所有的actor都使用react，则它们可以用单个线程实现。

由于react不返回，接收消息的消息处理器现在必须同时处理消息并执行actor所有余下的工作。通常的做法是用一个顶级的工作方法(比如act方法自身)供消息处理器在处理完消息本身之后调用。编写使用react而非receive的actor颇具挑战性，不过在性能上能够带来相当的回报。另外，actor库提供的Actor.loop函数可以重复执行一个代码块，哪怕代码调用的是react。

良好的actor风格：

良好的actor风格的并发编程可以使你的程序更容易调试并且减少死锁和争用状况。下面是一些actor风格编程的指导意见：

actor不应阻塞：编写良好的actor在处理消息时并不阻塞。因为阻塞可能会导致死锁，即其他多个actor都在等待该阻塞的actor的响应。代替阻塞当前actor，你可以创建一个助手actor，该助手actor在睡眠一段时间之后发回一个消息，以告诉创建它的actor。记住一句话：会阻塞的actor不要处理消息，处理消息的actor请不要使其阻塞。

只通过消息与actor通信：Actor模型解决共享数据和锁的关键方法是提供了一个安全的空间——actor的act方法——在这里你可以顺序地思考。换个说法就是，actor让你可以像一组独立的通过异步消息传递来进行交互的单线程的程序那样编写多线程的程序。不过，这只有在消息是你的actor的唯一通信途径的前提下才成立。一旦你绕过了actor之间的消息传递机制，你就回到了共享数据和锁的模型中，所有那些你想用actor模型避开的困难又都回来了。但这并不是说你应该完全避免绕开消息传递的做法，虽然共享数据和锁要做正确很难，但也不是完全不可能。实际上scala的actor和Erlang的actor实现方式的区别之一就是，scala让你可以在同一个程序中混用actor与共享数据和锁两种模型。

优选不可变消息：actor模型提供了每个actor的act方法的单线程环境，你无需担心它使用到的对象是否是线程安全的，因为它们都被局限于一个线程中。但例外的是在多个线程中间传送的消息对象，它被多个actor共享，因此你需要担心消息对象是否线程安全。确保消息对象是线程安全的最佳途径是在消息中只使用不可变对象。如果你发现你有一个可变对象，并且想通过消息发送给另一个actor，你应该制作并发送它的一个副本。

让消息自包含：actor在发送请求消息之后不应阻塞，它继续做着其他事情，当收到响应消息之后，它如何解释它(也就是说它如何能记起它在发送请求消息时它在做着什么呢)，这是一个问题。解决办法就是在响应消息中增加冗余信息，比如说可以把请求消息中的一些东东作为响应消息的一部分发送给请求者。

===GUI编程

scala图形用户界面编程可以使用scala.swing库，该库提供了对java的Swing框架的GUI类的访问，对其进行包装，隐藏了大部分复杂度。示例如下：

object MyScalaGUI extends SimpleSwingApplication {

def top = new MainFrame {

title = "My Scala GUI"

location = new Point(600, 300)

preferredSize = new Dimension(400, 200)

val button = new Button {

text = "Click me"

}

val label = new Label {

text = "Who am I?"

}

contents = new BoxPanel(Orientation.Vertical) {

contents += button

contents += label

}

listenTo(button)

reactions += {

case ButtonClicked(b) => label.text = "Hello, I'm Tom."

}

}

}

要编写图形界面程序，你可以继承SimpleSwingApplication类，该类已经定义了包含一些设置java Swing框架代码的main方法，main方法随后会调用top方法，而top方法是抽象的，需要你来实现。top方法应当包含定义顶级GUI组件的代码，这通常是某种Frame。

GUI类库：

Frame：即可以包含任意数据的窗体。Frame有一些属性(也就是getter和setter)，其中比较重要的有title(将被写到标题栏)、contents(将被显示在窗体中)。Frame继承自Container，每个Container都有一个contents属性，让你获得和设置它包含的组件，不过，Frame的contents只能包含一个组件，因为框架的contents属性只能通过“=”赋值，而有些Container (如Panel)的contents可以包含多个组件，因为它们的contents属性可以通过“+=”赋值。

MainFrame：就像是个普通的Swing的Frame，只不过关闭它的同时也会关闭整个GUI应用程序。

Panel：面板是根据某种固定的布局规则显示所有它包含的组件的容器。面板可以包含多个组件。

处理事件：

为了处理事件，你需要为用户输入事件关联一个动作，scala和java基本上用相同的“发布/订阅”方式来处理事件。发布者又称作事件源，订阅者又称作事件监听器。举例来说，Button是一个事件源，它可以发布一个事件ButtonClicked，表示该按钮被点击。scala中事件是真正的对象，创建一个事件也就是创建一个样本类的实例，样本类的参数指向事件源。事件(样本)类包含在scala.swing.event包中。

在scala中，订阅一个事件源source的方法是调用listenTo(source)，取消订阅的方法是调用deafTo(source)。比如：你可以让一个组件A监听它其中的一个组件B，以便A在B发出任何事件时得到通知。为了让A对监听到的事件做出响应，你需要向A中名为reactions的属性添加一个处理器，处理器也就是带有模式匹配的函数字面量，可以在单个处理器中用多个样本来匹配多种事件。可使用“+=”向reactions中添加处理器，使用“-=”从中移除处理器。从概念上讲，reactions中安装的处理器形成一个栈，当接收到一个事件时，最后被安装的处理器首先被尝试，但不管尝试是否成功，后续的处理器都会被一一尝试。

===结合scala和java

scala和java高度兼容，因此可以进行互操作，大多数情况下结合这两种语言时并不需要太多顾虑，尽管如此，有时你还是会遇到一些结合java和scala的问题。基本上，scala使用java代码相对于java使用scala代码更容易一些。

scala代码如何被翻译：

scala的实现方式是将代码翻译为标准的java字节码。scala的特性尽可能地直接映射为相对等的java特性。但scala中的某些特性（如特质）在java中没有，而还有一些特性（如泛型）与java中的不尽相同，对于这些特性，scala代码无法直接映射为java的语法结构，因此它必须结合java现有的特性来进行编码。这些是一般性的原则，现在我们来考虑一些特例：

值类型：类似Int这样的值类型翻译成java有两种不同的方式，只要可能就直接翻译为java的int以获得更好的性能，但有时做不到，就翻译为包装类的对象。

单例对象：scala对单例对象的翻译采用了静态和实例方法相结合的方式。对每一个scala单例对象，编译器都会为这个对象创建一个名称后加美元符号的java类，这个类拥有scala单例对象的所有方法和字段，这个java类同时还有一个名为MODULE$的静态字段，保存该类在运行期创建的一个实例。也就是说，在java中要这样使用scala中的单例对象：单例对象名$.MODULE$.方法名();

特质：编译任何scala特质都会创建一个同名的java接口，这个接口可以作为java类型使用，你可以通过这个类型的变量来调用scala对象的方法。如果特质中还有已经实现的方法，那么还会生成对应的实现类“特质名$class”。

存在类型：

所有java类型在scala中都有对等的概念，这是必要的，以便scala可以访问任何合法的java类。scala中的存在类型实际上主要是用于从scala访问java泛型中的通配类型以及没有给出类型参数的原始类型。存在类型的通用形式如下：

type forSome { declarations }

type部分是任意的scala类型，而declarations部分是一个抽象val和type的列表。这个定义解读为：声明的变量和类型是存在但未知的，正如类中的抽象成员那样。这个类型进而被允许引用这些声明的变量和类型，虽然编译器不知道它们具体指向什么。例如对于如下的java语句：

Iterator<?>

Iterator<? extends Component>

在scala中可以用存在类型分别表示为：

Iterator[T] forSome { type T }

Iterator[T] forSome { type T <: Component }

另外，存在类型也可以使用下划线占位符语法以更简短的方式来写。如果在可以使用类型的地方使用了下划线，那么scala会为你做出一个存在类型，每个下划线在forSome语句中都变成一个类型参数。如前我们在介绍scala的泛型时，指定类型参数上界下界时，使用的就是这种简写的语法。作为示例，我们可以将上述两个语句简写为：

Iterator[\_]

Iterator[\_ <: Component]

隐式转换为java类型：

在scala和java代码之间传递数据时，如果使用的是容器类库（当然包括数组等），那么可能需要进行一些转换，而scala类库本身就提供了这样的隐式转换库，因此你可以方便地在scala和java之间传递数据。你唯一需要做的是：import scala.collection.JavaConversions.\_。