

# 第一章 Scala 语法介绍

## 1.1 阅读说明

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。

本手册适合对象: 有 Java 编程经验的程序员。阅读时如果发现难以理解, 可以根据关键词自行上网搜索对应内容进行辅助学习。

\*标注的小节, 表示阅读优先级较低或者可以不用阅读。

阅读时遵循先易后难得准则, 从有代码示例的地方着手会比较简单。

## 1.2 第一个 scala 程序

参考: <http://www.scala-lang.org/documentation/getting-started.html>

```
object ScalaTest {  
  def main(args: Array[String]) {  
    println("hello scala.")  
  }  
}
```

### 1.2.1 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。

### 1.2.2 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。

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:

hello.scala 文件中: `println("hello,"+args(0))`

在命令行中执行: `scala hello.scala vitohuang`

---

### 1.2.3 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
```

## 1.3 Scala 开发环境

### 1.3.1 Scala 下载安装的三种方法

<http://www.scala-lang.org/download/>

#### 1.3.1.1 Win8 下配置 Scala 系统环境

##### 1. 下载 Scala 2.9.2

由于最新的 Scala 2.10 稳定版还没完成，所以最好是下载最新的 Scala 稳定版：2.9.2 版。（2015 年 3 月），注意对应的版本必须是 1.6 或 1.7。

下载地址：<http://www.scala-lang.org/downloads/distrib/files/scala-2.9.2.msi>

下载 msi 版本的好处在于，环境变量自动配置，否则你需要手动设置两个环境变量：SCALA\_HOME 环境变量，指向 Scala 的安装目录。

PATH 环境变量，要包含 %SCALA\_HOME%\bin 的值。

##### 2. 安装 Scala 2.9.2

下载完成后，执行 scala-2.9.2.msi，按提示一步步安装。我安装在 C:\scala 这里。

##### 3. 验证

按下 Windows 键+R 键，输入 CMD，回车后进入 WindowsCMD 命令行模式。

键入命令：

复制代码代码如下：

```
scala -version
```

显式结果如下：

```
C:\Users\LiSir>scala -version
Scala code runner version 2.9.2 -- Copyright 2002-2011, LAMP/EPFL
C:\Users\LiSir>
```

说明 Scala 安装和工作均正常！

### 1.3.2 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：

#### 1.3.2.1 eclipse 开发环境配置

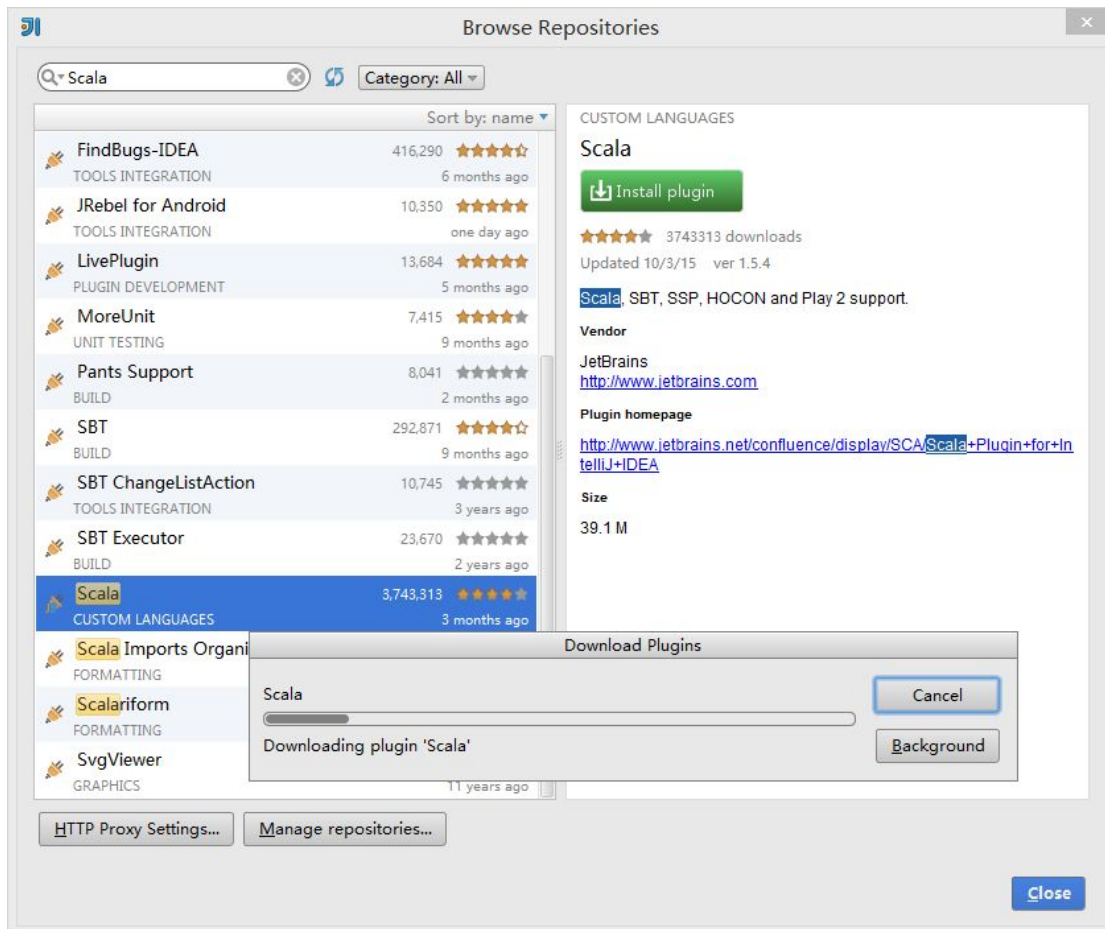
scala ide for eclipse（下载地址：<http://scala-ide.org>）中集成了 scala 插件，你可以直接使用它进行开发，不过它包含的可能不是我们想要的 scala 版本，因此，还是在该网站上下载对应的 scala 插件，插在 eclipse 上，这样更好啊。

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。

#### 1.3.2.2 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 repositories 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)。

### 1.3.2.3 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。

## 1.4 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。

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。

使用逗号分隔语句不被支持。

## 1.5 变量定义

```
object Val extends App {
  var ans: Int = 1
  ans += 1
  println(ans)
}
```

### 1.5.1 基本类型

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。

字符串

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。

类型转换

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。

### 1.5.2 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:

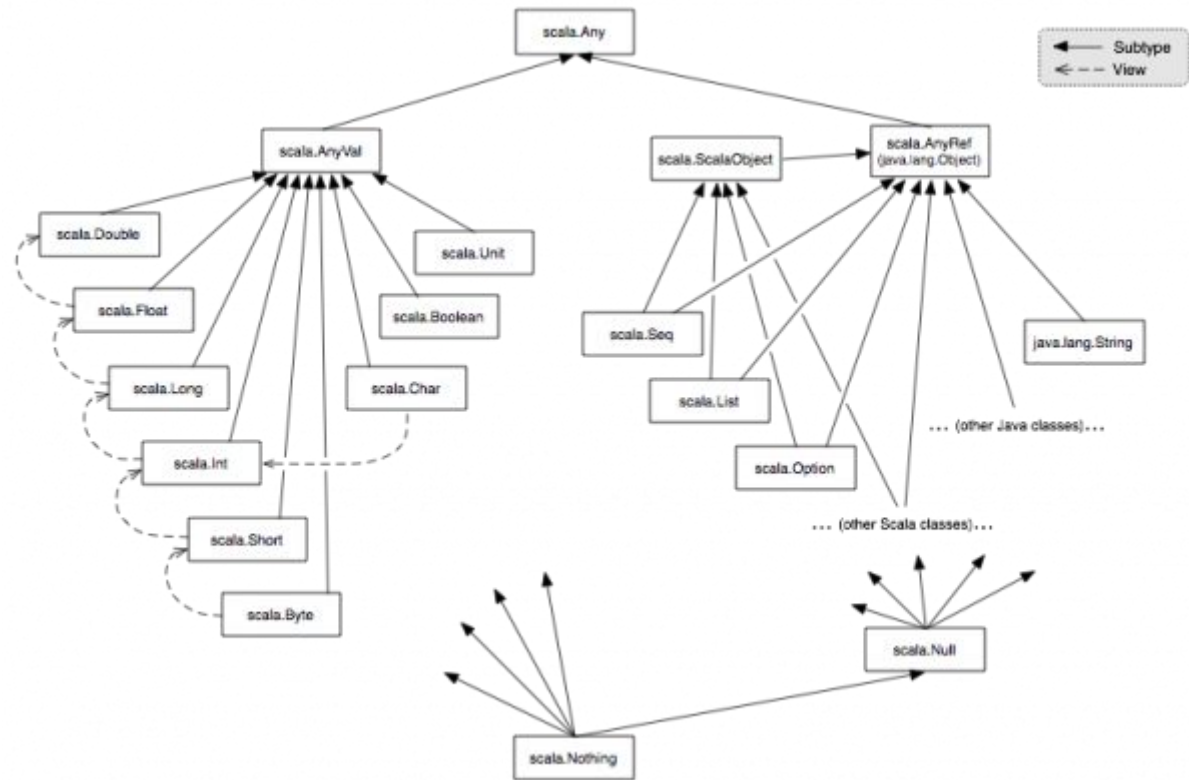
```
val answer = "yes"
val answer, message: String = "yes"
```

```
object Val extends App {
  val ans = 'string'
  ans = 'another string'
}
```

```
object Val extends App {
  var ans = 'string'
  ans = 'another string'
}
```

如图前者会报错, 后者不会报错

## 1.6 类继承关系 Scala Class Hierarchy



```
object UnifiedTypes extends App {  
  val set = new scala.collection.mutable.LinkedHashSet[Any]  
  set += "This is a string" // add a string  
  set += 732 // add a number  
  set += 'c' // add a character  
  set += true // add a boolean value  
  set += main _ // add the main function  
  val iter: Iterator[Any] = set.iterator  
  while (iter.hasNext) {  
    println(iter.next.toString())  
  }  
}
```

Here is the output of the program:

This is a string

732

c

true

<function>

## 1.7 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置“\$”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。

## 1.8 操作符\*

注意 scala 并不提供++、--操作符。

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。（?）

## 1.9 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，表达式用分号分隔，而不像 Java 中用逗号。

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
```

## 1.10 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数字面量。

### 1.10.1 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/else 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/else 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 ()
```

### 1.10.2 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。

注意下面的这种写法是错误的：

```
while(var i=0 ; i <- 1 to 9){  
    println(i)  
}
```

### 1.10.3 for 表达式

#### 1.10.3.1 语法发生器

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)，其中 “i <- 表达式” 语法称之为发生器

#### 1.10.3.2 用于枚举 for(i <- 1 to 10)

for(i <- 1 to 10)，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。



### 1.10.3.3 Range

[http://www.scala-lang.org/docu/files/collections-api/collections\\_18.html](http://www.scala-lang.org/docu/files/collections-api/collections_18.html)

### 1.10.3.4 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)。`

```
for(i<- 1 to 5; j <- 1 to i){
  print(i + " ")
  if(i == j) println()
}
```

输出：

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

### 1.10.3.5 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)。`

```
for(i<- 1 to 5; j <- 1 to i;if (!(i==3&& j==3)); if(i!=2 || j!=2)){
  print(i + " ")
  if(i == j) println()
}
```

输出：

```
1
2 3 3 4 4 4 4
5 5 5 5 5
```

### 1.10.3.6 遍历数组和所有集合

```
var set = Set("beijing", "shanghai");
set += "chongqing"
println(set.apply("guangzhou"))
for(i <- set) println(i)
```

### 1.10.3.7 流间变量绑定 (\*)

---

#### 1.10.4 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) ...

#### 1.10.5 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：

任何类型的常量/变量，都可以作为比较用的样本；

在每个 case 语句最后，不需要 break，break 是隐含的；

更重要的是 match 表达式也有值；

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：

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 _ =>
```

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。

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。

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。

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
```

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：

模式使用在变量定义中，如下：

```
val myTuple = (123, "abc")
```

```
val (number, string) = myTuple
```

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 => "?"
}
```

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。

## 1.11 数组

### 1.11.1 定义时指定元素

```
object Val extends App {
  val arr1 = Array(5,6,7) //注意是括号, 没有 new

  def f(a:Array[Int]) {
    for (arg <- a)
      println(arg)
  }
}
```

```
}  
  
f(arr1);  
}
```

### 1.11.2 数组定义

```
val arr1 = new Array[Int](3)
```

### 1.11.3 数组元素赋值或更改

两种方式：

```
arr1(1) = 66;  
arr1.update(1, 666)
```

### 1.11.4 数组元素选择

两种方式：

```
println(arr1(1))  
println(arr1.apply(1))
```

## 1.12 foreach 和匿名方法

### 1.12.1 foreach

```
arr1.foreach(arg => println(arg))
```

arr1 是上一小节中定义的数组，这一句的作用是将数组元素逐行打印出来。

arg 可以定义上类型：

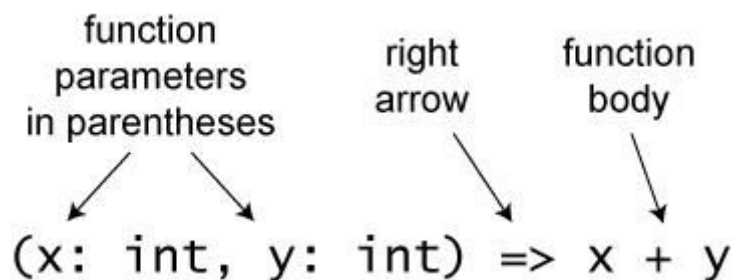
```
args.foreach((arg: String) => println(arg))
```

注意当给变量定义类型的时候，必须要加括号！

### 1.12.2 匿名方法

这里传给 foreach 的实际上就是一个匿名方法！

匿名方法的定义是这样的：



方法参数 => 方法体

我们这个传入的匿名方法就是： `(arg: String) => println(arg)`

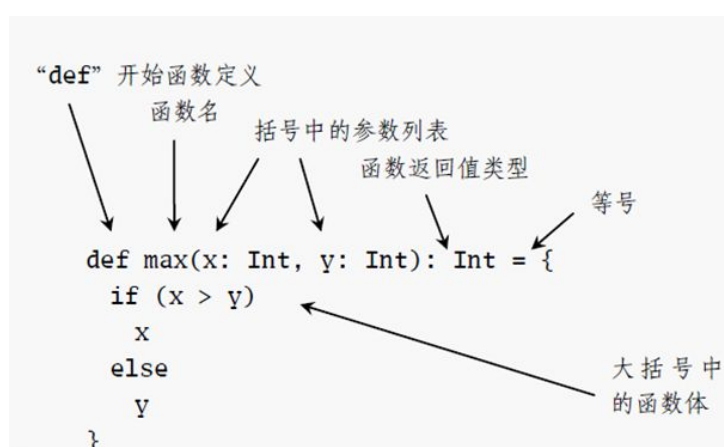
### 1.12.3 省略匿名函数的参数

很懒的程序员会发现，这里 `arg` 好像没什么必要，唯一参数传给唯一的方法体。所以我们可以省略成：

```
args.foreach(println)
```

## 1.13 函数

### 1.13.1 函数定义



每个函数参数后面必须带前缀冒号的类型标注，因为 Scala 编译器没办法推断函数参数类型。

除了递归函数之外，你可以省略返回值类型声明。

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
}
```

### 1.13.2 递归函数

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。

---

### 1.13.3 过程

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。

如：

```
def f(a: Int) {  
    println(a)  
}
```

```
println(f(1))
```

输出结果：

```
1  
()
```

没有返回值的函数的默认返回值是 Unit。

### 1.13.4 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。（?）

### 1.13.5 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。

### 1.13.6 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)
```

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: _*)
```

### 1.13.7 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 = "<<<")
```

### 1.13.8 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。

### 1.13.9 函数的其他定义方式

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。

#### 1.13.9.1 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。

#### 1.13.9.2 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)
```

#### 1.13.10 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。

## 1.14 类(class)和对象(object)

参考：[http://www.tutorialspoint.com/scala/scala\\_classes\\_objects.htm](http://www.tutorialspoint.com/scala/scala_classes_objects.htm)（英文的）

### 1.14.1 A quick start about Class

```

import java.io._
class Point(val xc: Int, val yc: Int) {
    var x: Int = xc
    var y: Int = yc
    def move(dx: Int, dy: Int) {
        x = x + dx
        y = y + dy
        println ("Point x location : " + x);
        println ("Point y location : " + y);
    }
}

```

```

    }}
object Test {
  def main(args: Array[String]) {
    val pt = new Point(10, 20);

    // Move to a new location
    pt.move(10, 10);
  }}

```

运行结果:

```
C:/>scalac Test.scala
```

```
C:/>scala TestPoint x location : 20Point y location : 30
```

### 1.14.2 继承的例子

```

import java.io._

class Point(val xc: Int, val yc: Int) {
  var x: Int = xc
  var y: Int = yc
  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
    println ("Point x location : " + x);
    println ("Point y location : " + y);
  }
}

class Location(override val xc: Int, override val yc: Int,
  val zc :Int) extends Point(xc, yc){
  var z: Int = zc

  def move(dx: Int, dy: Int, dz: Int) {
    x = x + dx
    y = y + dy
    z = z + dz
    println ("Point x location : " + x);
    println ("Point y location : " + y);
    println ("Point z location : " + z);
  }
}

object Test {
  def main(args: Array[String]) {
    val loc = new Location(10, 20, 15);

```

---

```

        // Move to a new location
        loc.move(10, 10, 5);
    }
}

```

### 1.14.3 类概述

正如我们所见，Scala 是一门面向对象的语言，因此它拥有很多关于“类”的描述。Scala 类使用 and Java 类似的语法进行定义。但是一个重要的不同点在于 Scala 中的类可以拥有参数，这样就可以得出我们下面关于对复数类

（Complex）的定义：

```

class Complex(real: Double, imaginary: Double) {
    def re() = real
    def im() = imaginary
}

```

我们的复数类（Complex）接受两个参数：实部和虚部。这些参数必须在实例化 时进行传递，就像这样：new Complex(1.5, 2.3)。类定义中包括两个叫做 re 和 im 的方法，分别接受上面提到的两个参数。值得注意的是这两个方法的返回类型并没有显式的声明出来。他们会被编译器自动识别。在本例中他们被识别为 Double 但是编译器并不总是像本例中的那样进行自动识别。不幸的是关于什么时候识别，什么时候不识别的规则相当冗杂。在实践中这通常不会成为一个问题，因为当编译器处理不了的时候会发出相当的抱怨。作为一个推荐的原则，Scala 的新手们通常可以试着省略类型定义而让编译器通过上下文自己判断。久而久之，新手们就可以感知到什么时候应该省略类型，什么时候不应该。

### 1.14.4 无参方法

关于方法 re 和 im 还有一个小问题：你必须在名字后面加上一对括号来调用它们。请看下面的例子：

```

object ComplexNumbers {
    def main(args: Array[String]) {
        val c = new Complex(1.2, 3.4)
        println("imaginary part: " + c.im())
    }
}

```

你可能觉得把这些函数当作变量使用，而不是当作函数进行调用，可能会更加令人感到舒服。事实上我们可以通过定义无参函数在 Scala 做到这点。这类函数 与其他的具有 0 个参数的函数的不同点在于他们定义时不需要在名字后面加括弧，所以在使用时也不用加（但是无疑的，他们是函数），因此，我们的 Complex 类可以重新写成下面的样子：

```

class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
}

```

### 1.14.5 继承和覆盖

Scala 中的所有类都继承一个父类，当没有显示声明父类时（就像上面定义的 `Complex` 一样），它们的父类隐式指定为 `scala.AnyRef`。

在子类中覆盖父类的成员是可能的。但是你需要通过 `override` 修饰符显示指定成员的覆盖。这样的规则可以避免意外覆盖的情况发生。作为演示，我们在 `Complex` 的定义中覆盖了 `Object` 的 `toString` 方法。

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() = "" + re + (if (im < 0) "" else "+") + im +
  "i"
}
```

### 1.14.6 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。

### 1.14.7 访问级别控制

公有性是 `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。

#### 1.14.8 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。

#### 1.14.9 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。

#### 1.14.10 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：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；

在伴生对象中提供 `unapply` 方法让模式匹配可以工作；

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；

添加 `toString`、`hashCode`、`equals`、`copy` 的“自然”实现。

#### 1.14.11 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。这还意味着你可以获得更好的编译器帮助。

#### 1.14.12 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)。

#### 1.14.13 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。

#### 1.14.14 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。

#### 1.14.15 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。

#### 1.14.16 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。

### 1.14.17 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。

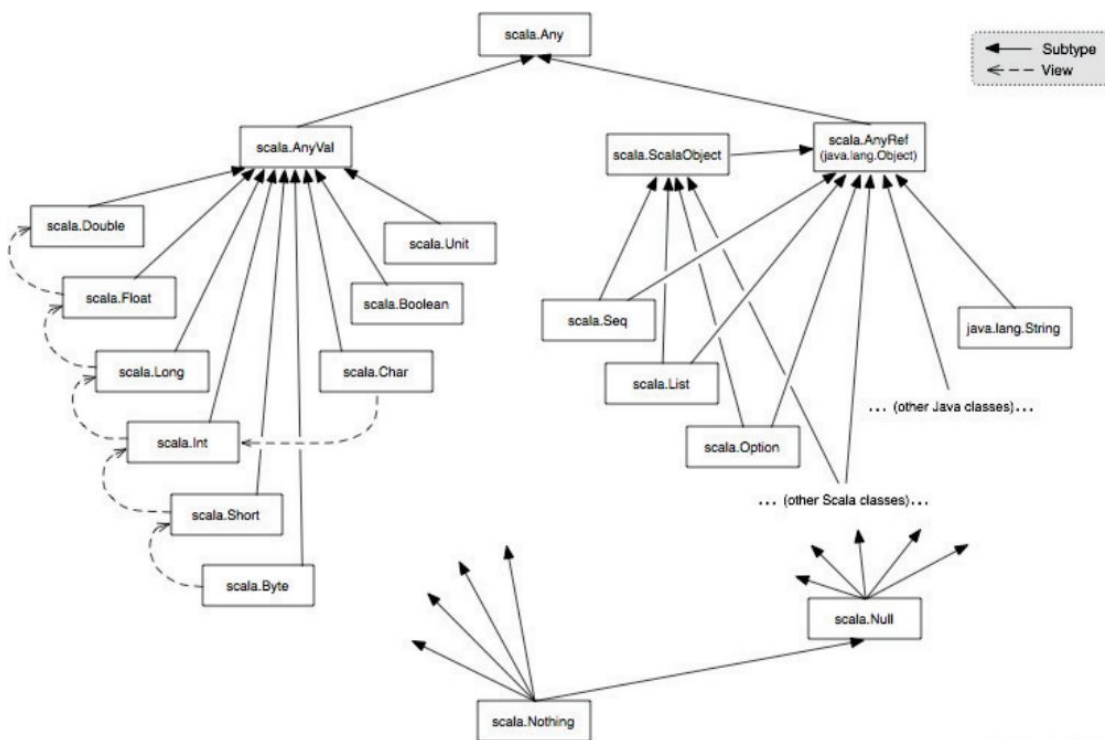
### 1.14.18 继承与覆盖(override)

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。

## 1.15 特质(trait)

### 1.15.1 A quick start!

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。

```
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)}
class Point(xc: Int, yc: Int) extends Equal {
  var x: Int = xc
  var y: Int = yc
  def isEqual(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x}
object Test {
  def main(args: Array[String]) {
    val p1 = new Point(2, 3)
    val p2 = new Point(2, 4)
    val p3 = new Point(3, 3)

    println(p1.isNotEqual(p2))
    println(p1.isNotEqual(p3))
    println(p1.isNotEqual(2))
  }
}
```

### 1.15.2 和类的区别

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。

### 1.15.3 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`。

### 1.15.4 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。

### 1.15.5 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。

### 1.15.6 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。

## 1.16 类型转换

### 1.16.1 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
```

### 1.16.2 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：

隐式参数也可以被显式提供；

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；

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；

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。

### 1.16.3 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。

### 1.16.4 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。

### 1.16.5 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。

### 1.16.6 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。

## 1.17 Set

Scala 致力于帮助你充分利用函数式和指令式风格两方面的好处，它的集合类型库于是就区分了集合类的可变和不可变。例如，数组始终是可变的，而列表始终不可变。

参考：<http://docs.scala-lang.org/zh-cn/overviews/collections/sets.html>

### 1.17.1 示例

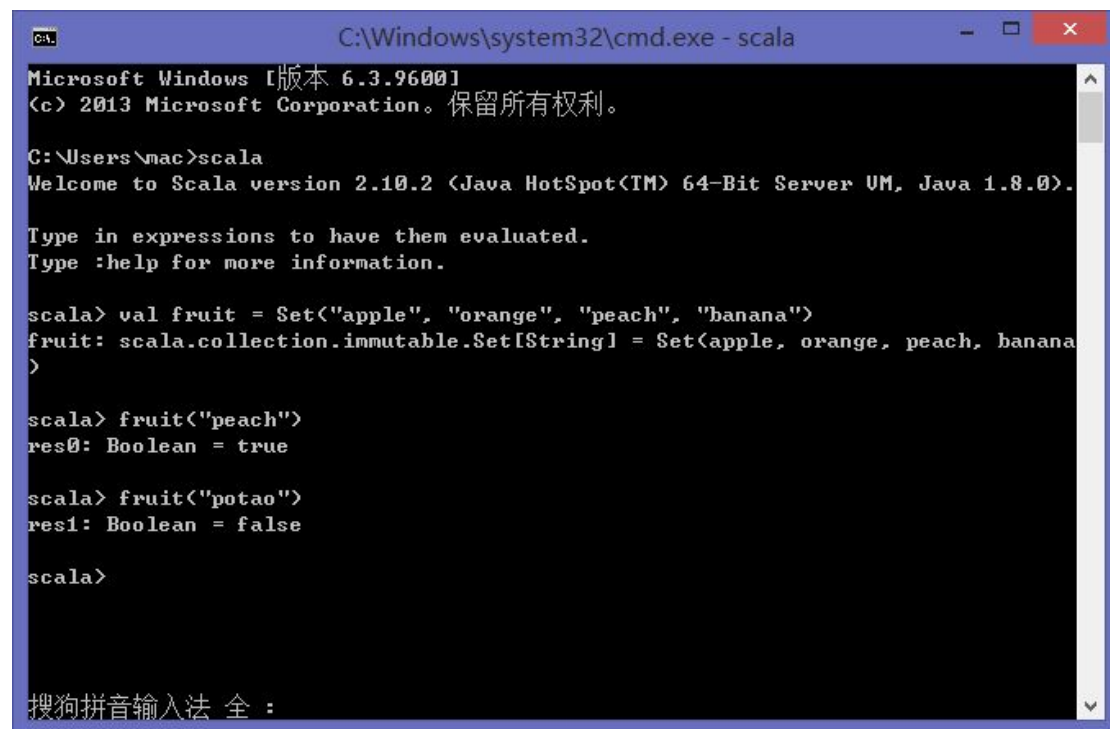
集合是不包含重复元素的可迭代对象。

contains 方法用于判断集合是否包含某元素。

集合的 apply 方法和 contains 方法的作用相同，因此 set(elem) 等同于 set contains elem。

示例:

```
val fruit = Set("apple", "orange", "peach", "banana")
fruit: scala.collection.immutable.Set[java.lang.String] =
Set(apple, orange, peach, banana)
scala> fruit("peach")
res0: Boolean = true
scala> fruit("potato")
res1: Boolean = false
```



## 1.17.2 Set 类的操作

### 1.17.2.1 测试类操作

xs contains x	测试 x 是否是 xs 的元素。	
xs(x)	与 xs contains x 相同。	
xs subsetOf ys	测试 xs 是否是 ys 的子集。	

### 1.17.2.2 加法、减法

xs + x	包含 xs 中所有元素以及 x 的集合。	
xs + (x, y, z)	包含 xs 中所有元素及附加元素的集合	

<code>xs ++ ys</code>	包含 <code>xs</code> 中所有元素及 <code>ys</code> 中所有元素的集合	
<code>xs - x</code>	包含 <code>xs</code> 中除 <code>x</code> 以外的所有元素的集合。	
<code>xs - (x,y,z)</code>	包含 <code>xs</code> 中除去给定元素以外的所有元素的集合。	
<code>xs -- ys</code>	集合内容为: <code>xs</code> 中所有元素, 去掉 <code>ys</code> 中所有元素后剩下的部分。	
<code>xs.empty</code>	与 <code>xs</code> 同类的空集合。	

### 1.17.2.3 二值操作（交、并、差）

<code>xs &amp; ys</code>	集合 <code>xs</code> 和 <code>ys</code> 的交集。	
<code>xs intersect ys</code>	等同于 <code>xs &amp; ys</code> 。	
<code>xs   ys</code>	集合 <code>xs</code> 和 <code>ys</code> 的并集。	
<code>xs union ys</code>	等同于 <code>xs ys</code>	
<code>xs &amp;~ ys</code>	集合 <code>xs</code> 和 <code>ys</code> 的差集。	
<code>xs diff ys</code>	等同于 <code>xs &amp;~ ys</code> 。	

可变集合提供加法类方法, 可以用来添加、删除或更新元素。下面对这些方法做下总结。

### 1.17.3 mutable.Set 类的操作

加法:		
<code>xs += x</code>	把元素 <code>x</code> 添加到集合 <code>xs</code> 中。该操作有副作用, 它会返回左操作符, 这里是 <code>xs</code> 自身。	
<code>xs += (x, y, z)</code>	添加指定的元素到集合 <code>xs</code> 中, 并返回 <code>xs</code> 本身。(同样有副作用)	
<code>xs ++= ys</code>	添加集合 <code>ys</code> 中的所有元素到集合 <code>xs</code> 中, 并返回 <code>xs</code> 本身。(表达式有副作用)	
<code>xs add x</code>	把元素 <code>x</code> 添加到集合 <code>xs</code> 中, 如集合 <code>xs</code> 之前没有包含 <code>x</code> , 该操作返回 <code>true</code> , 否则返回 <code>false</code> 。	
移除:		

<code>xs -= x</code>	从集合 <code>xs</code> 中删除元素 <code>x</code> ，并返回 <code>xs</code> 本身。（表达式有副作用）
<code>xs -= (x, y, z)</code>	从集合 <code>xs</code> 中删除指定的元素，并返回 <code>xs</code> 本身。（表达式有副作用）
<code>xs --= ys</code>	从集合 <code>xs</code> 中删除所有属于集合 <code>ys</code> 的元素，并返回 <code>xs</code> 本身。（表达式有副作用）
<code>xs remove x</code>	从集合 <code>xs</code> 中删除元素 <code>x</code> 。如之前 <code>xs</code> 中包含了 <code>x</code> 元素，返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>xs retain p</code>	只保留集合 <code>xs</code> 中满足条件 <code>p</code> 的元素。
<code>xs.clear()</code>	删除集合 <code>xs</code> 中的所有元素。
<b>**更新：**</b>	
<code>xs(x) = b</code>	（同 <code>xs.update(x, b)</code> ）参数 <code>b</code> 为布尔类型，如果值为 <code>true</code> 就把元素 <code>x</code> 加入集合 <code>xs</code> ，否则从集合 <code>xs</code> 中删除 <code>x</code> 。
克隆：	
<code>xs.clone</code>	产生一个与 <code>xs</code> 具有相同元素的可变集合。

#### 1.17.4 SortedSet\*

#### 1.17.5 BitSet\*

### 1.18 Map 映射

映射（Map）是一种可迭代的键值对结构（也称映射或关联）。Scala 的 `Predef` 类提供了隐式转换，允许使用另一种语法：`key -> value`，来代替 `(key, value)`。如：`Map("x" -> 24, "y" -> 25, "z" -> 26)`等同于 `Map(("x", 24), ("y", 25), ("z", 26))`，却更易于阅读。

映射（Map）的基本操作与集合（Set）类似。

查询类操作：`apply`、`get`、`getOrElse`、`contains` 和 `DefinedAt`。它们都是根据主键获取对应的值映射操作。例如：`def get(key): Option[Value]`。“`m get key`”返回 `m` 中是否用包含了 `key` 值。如果包含了，则返回对应 `value` 的 `Some` 类型值。否则，返回 `None`。这些映射中也包括了 `apply` 方法，该方法直接返回主键对应的值。`apply` 方法不会对值进行 `Option` 封装。如果该主键不存在，则会抛出异常。

添加及更新类操作：`+`、`++`、`updated`，这些映射操作允许你添加一个新的绑定或更改现有的绑定。



删除类操作：-、-，从一个映射（Map）中移除一个绑定。

子集类操作：keys、keySet、keysIterator、values、valuesIterator，可以以不同形式返回映射的键和值。

filterKeys、mapValues 等变换用于对现有映射中的绑定进行过滤和变换，进而生成新的映射。

### 1.18.1 Map 类的操作

WHAT IT IS	WHAT IT DOES
查询：	
ms get k	返回一个 Option，其中包含和键 k 关联的值。若 k 不存在，则返回 None。
ms(k)	（完整写法是 ms apply k）返回和键 k 关联的值。若 k 不存在，则抛出异常。
ms getOrElse (k, d)	返回和键 k 关联的值。若 k 不存在，则返回默认值 d。
ms contains k	检查 ms 是否包含与键 k 相关联的映射。
ms isDefinedAt k	同 contains。
添加及更新：	
ms + (k -> v)	返回一个同时包含 ms 中所有键值对及从 k 到 v 的键值对 k -> v 的新映射。
ms + (k -> v, l -> w)	返回一个同时包含 ms 中所有键值对及所有给定的键值对的新映射。
ms ++ kvs	返回一个同时包含 ms 中所有键值对及 kvs 中的所有键值对的新映射。
ms updated (k, v)	同 ms + (k -> v)。
移除：	
ms - k	返回一个包含 ms 中除键 k 以外的所有映射关系的映射。
ms - (k, l, m)	返回一个滤除了 ms 中与所有给定的键相关联的映射关系的新映射。
ms - ks	返回一个滤除了 ms 中与 ks 中给出的键相关联的映射关系的新

WHAT IT IS	WHAT IT DOES
	映射。
子 容 器 (Subcollection) :	
ms.keys	返回一个用于包含 ms 中所有键的 iterable 对象（译注：请注意 iterable 对象与 iterator 的区别）
ms.keySet	返回一个包含 ms 中所有的键的集合。
ms.keyIterator	返回一个用于遍历 ms 中所有键的迭代器。
ms.values	返回一个包含 ms 中所有值的 iterable 对象。
ms.valuesIterator	返回一个用于遍历 ms 中所有值的迭代器。
变换:	
ms filterKeys p	一个映射视图（Map View），其包含一些 ms 中的映射，且这些映射的键满足条件 p。用条件谓词 p 过滤 ms 中所有的键，返回一个仅包含与过滤出的键值对的映射视图（view）。
ms mapValues f	用 f 将 ms 中每一个键值对的值转换成一个新的值，进而返回一个包含所有新键值对的映射视图（view）。

Scala 采用了类继承机制提供了可变的和不可变的两种版本的 Map，Map 的类继承机制看上去和 Set 的很像。

scala.collection 包里面有一个基础 Map 特质和两个子特质 Map：可变的 Map 在 scala.collection.mutable 里，不可变的在 scala.collection.immutable 里。

Map 可变映射的创造过程：

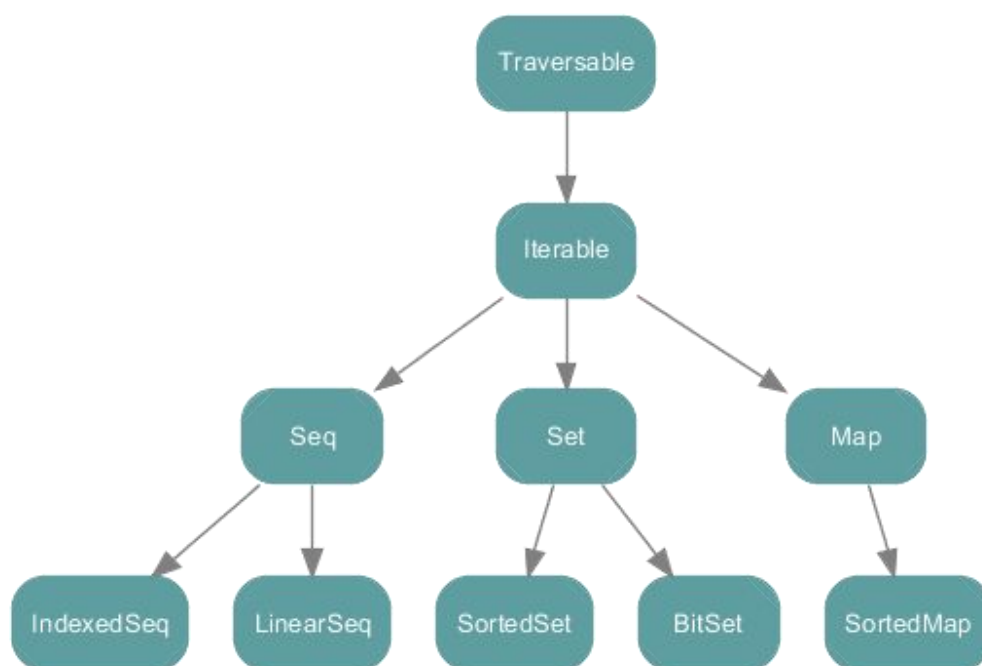
```
import scala.collection.mutable.Map
val treasureMap = Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

显式类型初始化 “[Int, String]”，对于可变的集合且映射为空 **Map[Int, String]()**，没有任何值被传递给工厂方法，编译器无法推断映射的类型参数，因此可变的集合且映射为空必须进行显示类型初始化。

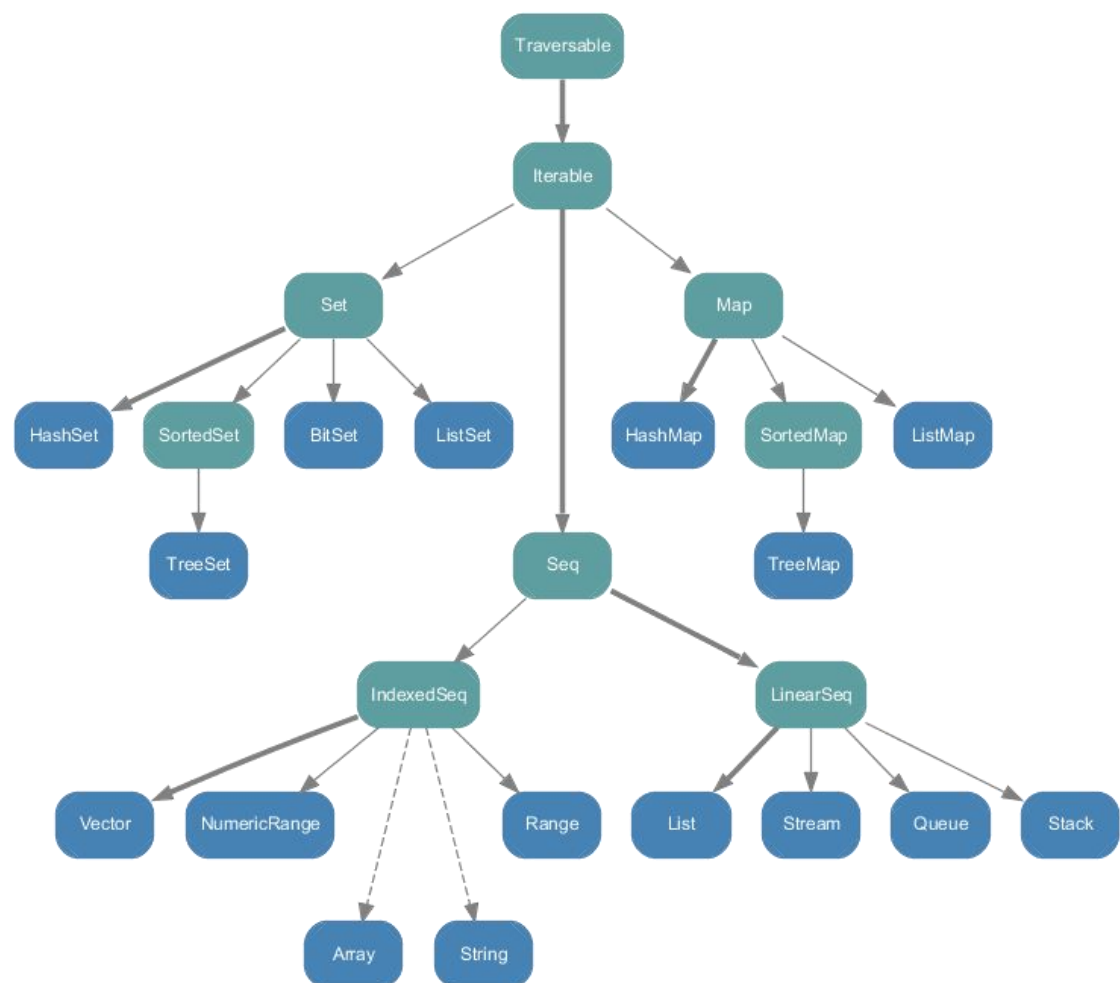
## 1.19 集合

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`。

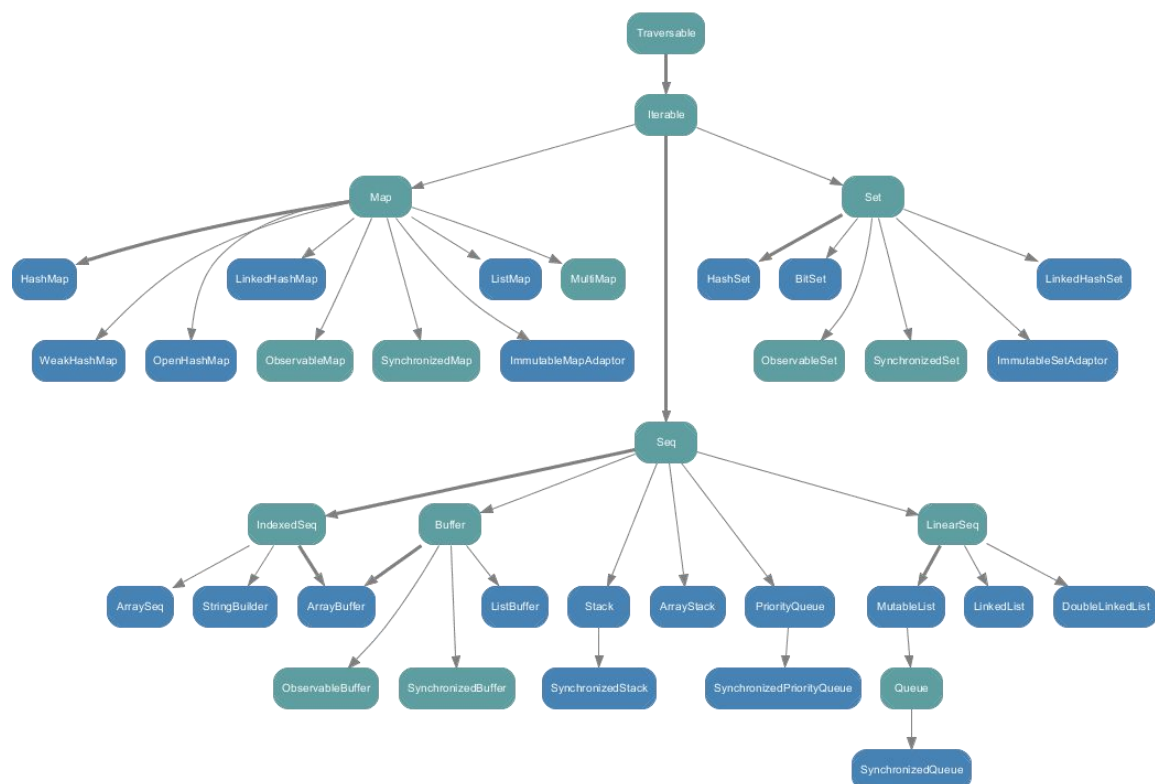
### 1.19.1 scala 集合的顶层抽象类和特质



`scala.collection.immutable:`



scala.collection.mutable:



### 1.19.2 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

### 1.19.3 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。(?)

### 1.19.4 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。

---

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

#### 1.19.4.1 多维数组

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)
```

### 1.19.5 列表

#### 1.19.5.1 保存相同类型

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。

#### 1.19.5.2 元素不可变

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`。

`Array[String]`仅包含 `String`。尽管实例化之后你无法改变 `Array` 的长度，它的元素值却是可变的。因此，`Array` 是可变的对象。

和数组一样，`List[String]`包含的仅仅是 `String`。但是 Scala 的 `List`，即 `scala.List`，不同于 Java 的 `java.util.List`，总是不可变的（而 Java 的 `List` 可变）。

#### 1.19.5.3 创建

创建一个 `List` 很简单。代码 3.3 做了展示：

```
val oneTwoThree = List(1, 2, 3)
```

代码 3.3 中的代码完成了一个新的叫做 `oneTwoThree` 的 `val`，并已经用带有整数元素值 1，2 和 3 的新 `List[Int]`初始化。

#### 1.19.5.4 ::: 叠加操作

`List` 的不可变性表现得有些像 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。

比方说，`List` 有个叫 “`:::`” 的方法实现叠加功能。你可以这么用：

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwooneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new List.")
```

如果你执行这个脚本，你会看到：

List(1, 2) and List(3, 4) were not mutated.

Thus, List(1, 2, 3, 4) is a new List.

#### 1.19.5.5 :: 添加元素操作

或许 List 最常用的操作符是发音为“cons”的“::”。Cons 把一个新元素组合到已有 List 的最前端，然后返回结果 List。例如，若执行这个脚本：

```
val twoThree = list(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
你会看到：
List(1, 2, 3)
```

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。](#)

#### 1.19.5.6 Nil 和::

Nil 是空 List，相当于 List()。

由于定义空类的捷径是 Nil，所以一种初始化新 List 的方法是把所有元素用 cons 操作符串起来，Nil 作为最后一个元素。如下：

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
不要企图在这个语句中省略 Nil，会报错。
```

#### 1.19.5.7 List 的一些方法和作用（表格）

表格 3.1 类型 List 的一些方法和作用

方法名	方法作用
List() 或 Nil	空 List
List("Cool", "tools", "rule")	创建带有三个值"Cool", "tools"和"rule"的新 List[String]
val thrill = "Will"::"fill"::"until"::Nil	创建带有三个值"Will", "fill"和"until"的新 List[String]
List("a", "b") ::: List("c", "d")	叠加两个列表 (返回带"a", "b", "c"和"d"的新 List[String])
thrill(2)	返回在 thrill 列表上索引为 2 (基于 0) 的元素 (返回"until")
thrill.count(s => s.length == 4)	计算长度为 4 的 String 元素个数 (返回 2)
thrill.drop(2)	返回去掉前 2 个元素的 thrill 列表 (返回 List("until"))
thrill.dropRight(2)	返回去掉后 2 个元素的 thrill 列表 (返回 List("Will"))
thrill.exists(s => s == "until")	判断是否有值为"until"的字符串元素在 thrill 里 (返回 true)
thrill.filter(s => s.length == 4)	依次返回所有长度为 4 的元素组成的列表 (返回 List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	辨别是否 thrill 列表里所有元素都以"l"结尾 (返回 true)
thrill.foreach(s => print(s))	对 thrill 列表每个字符串执行 print 语句 ("Willfilluntil")
thrill.foreach(print)	与前相同, 不过更简洁 (同上)
thrill.head	返回 thrill 列表的第一个元素 (返回"Will")
thrill.init	返回 thrill 列表除最后一个以外其他元素组成的列表 (返回 List("Will", "fill"))
thrill.isEmpty	说明 thrill 列表是否为空 (返回 false)
thrill.last	返回 thrill 列表的最后一个元素 (返回"until")
thrill.length	返回 thrill 列表的元素数量 (返回 3)
thrill.map(s => s + "y")	返回在 thrill 列表里每一个 String 元素都加了"y"构成的列表 (返回 List("Willy", "filly", "untily"))
thrill.mkString(", ")	用列表的元素创建字符串 (返回"will, fill, until")
thrill.remove(s => s.length == 4)	返回去除了 thrill 列表中长度为 4 的元素后依次排列的元素列表 (返回 List("until"))
thrill.reverse	返回含有 thrill 列表的逆序元素的列表 (返回 List("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)	返回包括 thrill 列表所有元素, 并且第一个字符小写按照字母顺序排列的列表 (返回 List("fill", "until", "Will"))
thrill.tail	返回去掉第一个元素的 thrill 列表 (返回 List("fill", "until"))

## 1.19.6 元组

### 1.19.6.1 元组

另一种有用的容器对象是元组: tuple。与列表 List 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并且因此而不能继承自 Iterable。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。



代码 3.4 展示了一个例子：

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

Scala 推断元组类型为 `Tuple2[Int, String]`，并把它赋给变量 `pair`。第二行，你访问 `_1` 字段，从而输出第一个元素，99。第二行的这个 “.” 与你用来访问字段或调用方法的点没有区别。本例中你正用来访问名叫 `_1` 的字段。如果执行这个脚本，你能看到：

```
99
```

```
Luftballons
```

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]`。

访问元组的元素：

这些 `_N` 数字是基于 1 的，而不是基于 0 的，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。

代码示例 2：

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

val t2 = t._2 // 引用元组第二个组元
println(t2)

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

val (first1, second2, _, _) = t // 只获取前两个组元
println()
println(first1, second2)
```

输出结果：

```
Jim
```

```
(1400,Jim,haha,3.14)
```

```
1400
```

```
Jim
```

```
haha
```

```
3.14
```

---

(1400,Jim)

### 1.19.6.2 对偶

对偶是  $n=2$  的元组，如 Map（映射）是键值对偶。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)。

我们可以这样构造一个映射：

```
val scores = Map("Alice"-> 10, "Bob"->3, "Cindy"->8)
```

我们也可以这样构造一个映射：

```
val scores = Map(("Alice",10),("Bob",3),("Cindy",8))
```

### 1.19.7 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。

### 1.19.8 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。

### 1.19.9 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))
```

## 1.20 异常

scala 的异常工作机制与 java 的类似，但也有区别。区别如下：

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。

throw 表达式是有值的，其值是 Nothing 类型。

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。

### 1.21 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。

## 1.22 包和引用

### 1.22.1 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：

使用放在文件顶部的 `package` 子句来把整个文件放入包中；

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。

### 1.22.2 引用

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

可以出现在任何地方，而不仅仅在文件开始处；

可以引用对象和包；

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 之外的所有东西。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._
```

### 1.22.3 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。

## 1.23 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 中有的东西：

`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`。

---

## 1.24 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。

## 1.25 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。

### 1.25.1 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。

### 1.25.2 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."  
  }  
}
```

### 1.25.3 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"`。

### 1.25.4 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`。

### 1.25.5 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。

## 1.26 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`。

### 1.26.1 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。

### 1.26.2 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。

## 1.27 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。

### 1.27.1 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._`。