面向 Java 开发人员的 Scala 指南

Ted Neward 将和您一起深入探讨 Scala 编程语言。在本系列专栏中，您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

由 dongfengyee（东风雨）整理

此文摘自developerWorks 中国 的 Java technology 系列

developerWorks 中国
本系列的更多信息:
面向 Java 开发人员的 Scala 指南
本文内容包括:
- 我将始终爱着你 .....
- 一种可伸缩语言
- 函数概念
- 开始认识您
- 将函数和表单最终结合起来
- 您说的是闭包吗?
- 匿名函数，您的函数是什么?
- 结束语
- 参考资料
- 关于作者
- 对本文的评价

相关链接:
- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南: 面向对象的函数编程

了解 **Scala** 如何利用两个领域的优点

文档选项
打印本页
将此页作为电子邮件发送
英文原文

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 2 月 04 日

在历史上，Java™ 平台一直属于面向对象编程的领域，但是现在，甚至 Java 语言的坚定支持者也开始注意应用程序开发中的一种新趋势：函数编程。在这个新的系列中，Ted Neward 介绍了 Scala，一种针对 JVM 将函数和面向对象技术组合在一起的编程语言。在本文中，Ted 将举例说明您为何应该花时间学习 Scala（例如并发），并介绍如何快速从中受益。

您永远不会忘记您的初恋!

对于我来说，她的名字是 Tabinda (Bindi) Khan。那是一段愉快的少年时光，准确地说是在七年级。她很美丽、聪明，而最好的是，她常常因我的笨拙的笑话而乐不可支。在七年级和八年级的时间里，我们经常“出去走走”（那时我们是这么说的）。但到了九年级，我们分开了，文雅一点的说法是，她厌倦了连续两年听到同样的笨拙的男孩笑话。我永远都不会忘记她（特别是因为我们在高中毕业 10 周年聚会时再次相遇）；但更重要的是，我将永远不会失去这些珍贵的（也许有点言过其实）回忆。

Java 编程和面向对象是许多程序员的“初恋”，我们对待它就像对待 Bindi 一样尊重和完全的爱慕。一些开发人员会告诉您 Java 将他们从内存管理和 C++ 的炼狱中解救出来了。其他一些人会告诉您 Java 编程使他们摆脱了对过程性编程的绝望。甚至对于一些开发人员来说，Java 代码中的面向对象编程就是“他们做事情的方式”。（嘿嘿，如果这对我爸爸，以及爷爷有用该多好！）

然而，时间最终会冲淡所有对初恋的记忆，生活仍然在继续。感情已经变了，故事中的主角也成熟了（并且学会了一些新笑话）。但最重要的是，我们周围的世界变了。许多 Java 开发人员意识到尽管我们深爱 Java 编程，但也应该抓住开发领域中的新机会，并了解如何利用它们。

关于本系列

Ted Neward 潜心研究 Scala 编程语言，并带您跟他一起徜徉。在这个新的 developerWorks 系列 中，您将深入了解 Scala，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在! 毕竟，如果 Java 代码可以做到的话，又何必学习 Scala 呢?

我将始终爱着你 .....

在最近五年中，对 Java 语言的不满情绪逐渐增多。尽管一些人可能认为 Ruby on Rails 的发展是主要因素，但是我要争辩的是，RoR（被称为 Ruby 专家）只是结果，而非原因。或者，可以更准确地说，Java 开发人员使用 Ruby 有着更深刻、更隐伏的原因。

简单地说，Java 编程略显老态了。

或者，更准确地说，Java 语言 略显老态了。

考虑一下：当 Java 语言最初诞生时，Clinton（第一位）在办公室中，很少有人使用 Internet，这主要是因为拨号是在家里使用网络的惟一方式。博客还没有发明出来，每个人相信继承是重用的基本方法。我们还相信，对象是为对世界进行建模的最好方法，摩尔定律将永远统治着世界。

实际上，摩尔定律引起了行业内许多人的特别关注。自 2002/2003 年以来，微处理器技术的发展使得具有多个“内核”的 CPU 得以创造出来：本质上是一个芯片内具有多个 CPU。这违背了摩尔定律，摩尔定律认为 CPU 速度将每隔 18 个月翻一倍。在两个 CPU 上同时执行多线程环境，而不是在单个 CPU 上执行标准循环周期，这意味着代码必须具有牢固的线程安全性，才能存活下来。

学术界已经展开了围绕此问题的许多研究，导致了过多新语言的出现。关键问题在于许多语言建立在自己的虚拟机或解释器上，所以它们代表（就像 Ruby

一样）到新平台的转换。并发冲突是真正的问题所在，一些新语言提供了强大的解决方案，太多的公司和企业对 10 年前从 C++ 到 Java 平台的迁移仍记忆犹新。许多公司都不愿意冒迁移到新平台的风险。事实上，许多公司对上一次迁移到 Java 平台仍心有余悸。

了解 Scala。

回页首

一种可伸缩语言

Scala 是一种函数对象混合的语言，具有一些强大的优点：

- 首先，Scala 可编译为 Java 字节码，这意味着它在 JVM 上运行。除了允许继续利用丰富的 Java 开源生态系统之外，Scala 还可以集成到现有的 IT 环境中，无需进行迁移。
- 其次，Scala 基于 Haskell 和 ML 的函数原则，大量借鉴了 Java 程序员钟爱的面向对象概念。因此，它可以将两个领域的优势混合在一起，从而提供了显著的优点，而且不会失去我们一直依赖的熟悉的技术。
- 最后，Scala 由 Martin Odersky 开发，他可能是 Java 社区中研究 Pizza 和 GJ 语言的最著名的人，GJ 是 Java 5 泛型的工作原型。而且，它给人一种“严肃”的感觉；该语言并不是一时兴起而创建的，它也不会以同样的方式被抛弃。

Scala 的名称表明，它还是一种高度可伸缩 的语言。我将在本系列的后续文章中介绍有关这一特性的更多信息。

下载并安装 Scala

可以从 Scala 主页 下载 Scala 包。截止到撰写本文时，最新的发行版是 2.6.1-final。它可以在 Java 安装程序版本 RPM 和 Debian 软件包 gzip/bz2/zip 包中获得，可以简单地将其解压到目标目录中，而且可以使用源码 tarball 从头创建。（Debian 用户可以使用“apt-get install”直接从 Debian 网站上获得 2.5.0-1 版。2.6 版本具有一些细微的差异，所以建议直接从 Scala 网站下载和安装。）

将 Scala 安装到所选的目标目录中 — 我是在 Windows® 环境中撰写本文的，所以我的目标目录是 C:/Prg/scala-2.6.1-final。将环境变量 `SCALA_HOME` 定义为此目录，将 `SCALA_HOME\bin` 放置于 `PATH` 中以便从命令行调用。要测试安装，从命令行提示符中激发 `scalac -version`。它应该以 Scala 版本 2.6.1-final 作为响应。

回页首

函数概念

开始之前，我将列出一些必要的函数概念，以帮助理解为何 Scala 以这种方式操作和表现。如果您对函数语言 — Haskell、ML 或函数领域的新成员 F# — 比较熟悉，可以 跳到下一节。

函数语言的名称源于这样一种概念：程序行为应该像数学函数一样；换句话说，给定一组输入，函数应始终返回相同的输出。这不仅意味着每个函数必须返回一个值，还意味着从一个调用到下一个调用，函数本质上不得具有内蕴状态（intrinsic state）。这种无状态的内蕴概念（在函数/对象领域中，默认情况下指的是永远不变的对象），是函数语言被认为是并发领域伟大的“救世主”的主要原因。

与许多最近开始在 Java 平台上占有一席之地的动态语言不同，Scala 是静态类型的，正如 Java 代码一样。但是，与 Java 平台不同，Scala 大量利用了类型推断（*type inferencing*），这意味着，编译器深入分析代码以确定特定值的类型，无需编程人员干预。类型推断需要较少的冗余类型代码。例如，考虑声明本地变量并为其赋值的 Java 代码，如清单 1 所示：

清单 **1.** 声明本地变量并为其赋值的 **Java** 代码

```
class BrainDead {
  public static void main(String[] args) {
    String message = "Why does javac need to be told message is a String?" +
      "What else could it be if I'm assigning a String to it?";
  }
```

```
}
```

Scala 不需要任何这种手动操作，稍后我将介绍。

大量的其他函数功能（比如模式匹配）已经被引入到 Scala 语言中，但是将其全部列出超出了本文的范围。Scala 还添加许多目前 Java 编程中没有的功能，比如操作符重载（它完全不像大多数 Java 开发人员所想象的那样），具有“更高和更低类型边界”的泛型、视图等。与其他功能相比，这些功能使得 Scala 在处理特定任务方面极其强大，比如处理或生成 XML。

但抽象概述并不够：程序员喜欢看代码，所以让我们来看一下 Scala 可以做什么。

回页首

开始认识您

根据计算机科学的惯例，我们的第一个 Scala 程序将是标准的演示程序“Hello World”：

Listing 2. Hello.Scala

```
object HelloWorld {
  def main(args: Array[String]): unit = {
    System.out.println("Hello, Scala!")
  }
}
```

和Python有很高的相似度：但是又有不同，参数的类型，以及返回值类型

使用 `scalac Hello.scala` 编译此程序，然后使用 Scala 启动程序（`scala HelloWorld`）或使用传统的 Java 启动程序运行（Scala 核心库包括在 JVM 的类路径（`java -classpath %SCALA_HOME%\lib\scala-library.jar;. HelloWorld`）中。不管使用哪一种方法，都应出现传统的问候。

清单 2 中的一些元素对于您来说一定很熟悉，但也使用了一些新元素。例如，首先，对 `System.out.println` 的熟悉的调用演示了 Scala 对底层 Java 平台的忠诚。Scala 充分利用了 Java 平台可用于 Scala 程序的强大功能。（事实上，它甚至会允许 Scala 类型继承 Java 类，反之亦然，但更多信息将在稍后介绍。）

另一方面，如果仔细观察，您还会注意到，在 `System.out.println` 调用的结尾处缺少分号；这并非输入错误。与 Java 平台不同，如果语句很明显是在一行的末尾终结，则 Scala 不需要分号来终结语言。但是，分号仍然受支持，而且有时候是必需的，例如，多个语句出现在同一物理行时。通常，刚刚入门的 Scala 程序员不用考虑需不需加分号，当需要分号的时候，Scala 编译器将提醒程序员（通常使用闪烁的错误消息）。

此外，还有一处微小的改进，Scala 不需要包含类定义的文件来反映类的名称。一些人将发现这是对 Java 编程的振奋人心的变革；那些没有这样做的人可以继续使用 Java“类到文件”的命名约定，而不会出现问题。

现在，看一下 Scala 从何处真正脱离传统的 Java/面向对象代码。

回页首

将函数和表单最终结合起来

对于初学者，Java 发烧友将注意到，`HelloWorld` 是使用关键字 `object` 来定义的，而不是使用 `class`。这是 Scala 对单例模式（Singleton pattern）的认可 — `object` 关键字告诉 Scala 编译器这将是个单例对象，因此 Scala 将确保只有一个 `HelloWorld` 实例存在。基于同样的原因，注意 `main` 没有像在 Java 编程中一样被定义为静态方法。事实上，Scala 完全避开了 `static` 的使用。如果应用程序需要同时具有某个类型的实例和某种“全局”实例，则 Scala 应用程序将允许以相同的名字同时定义 `class` 和 `object`。

接下来，注意 main 的定义，与 Java 代码一样，是 Scala 程序可接受的输入点。它的定义，虽然看起来与 Java 的定义不同，实际上是等同的：main 接受 String 数组作为参数且不返回任何值。但是，在 Scala 中，此定义看起来与 Java 版本稍有差异。args 参数被定义为 args: Array[String]。

在 Scala 中，数组表示为泛型化的 Array 类的实例，这正是 Scala 使用方括号（"[]"）而非尖括号（"<>"）来指明参数化类型的原因。此外，为了保持一致性，整个语言中都使用 name: type 的这种模式。

与其他传统函数语言一样，Scala 要求函数（在本例中为一个方法）必须始终返回一个值。因此，它返回称为 unit 的"无值"值。针对所有的实际目的，Java 开发人员可以将 unit 看作 void，至少目前可以这样认为。

方法定义的语法似乎比较有趣，当它使用 = 操作符时，就像将随后的方法体赋值给 main 标识符。事实上，真正发生的事情是：在函数语言中，就像变量和常量一样，函数是一级概念，所以语法上也是一样地处理。

回页首

您说的是闭包吗?

函数作为一级概念的一个含义是，它们必须被识别为单独的结构，也称为闭包，这是 Java 社区最近一直热烈争论的话题。在 Scala 中，这很容易完成。考虑清单 3 中的程序，此程序定义了一个函数，该函数每隔一秒调用一次另一个函数：

清单 **3. Timer1.scala**

```
object Timer
{
  def oncePerSecond(): unit =
  {
    while (true)
    {
      System.out.println("Time flies when you're having fun(ctionally)...")
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): unit =
  {
    oncePerSecond
  }
}
```

不幸的是，这个特殊的代码并没有什么功能 ..... 或者甚至没任何用处。例如，如果想要更改显示的消息，则必须修改 oncePerSecond 方法的主体。传统的 Java 程序员将通过为 oncePerSecond 定义 String 参数来包含要显示的消息。但甚至这样也是极端受限的：其他任何周期任务（比如 ping 远程服务器）将需要各自版本的 oncePerSecond，这很明显违反了"不要重复自己"的规则。我认为我可以做得更好。

清单 **4. Timer2.scala**

```
object Timer
{
  def oncePerSecond(callback: () => unit): unit =
  {
    while (true)
```

```
    {
      callback()
      Thread.sleep(1000)
    }
  }

  def timeFlies(): unit =
  { Console.println("Time flies when you're having fun(ctionally)..."); }

  def main(args: Array[String]): unit =
  {
    oncePerSecond(timeFlies)
  }
}
```

现在，事情开始变得有趣了。在清单 4 中，函数 `oncePerSecond` 接受一个参数，但其类型很陌生。形式上，名为 `callback` 的参数接受一个函数作为参数。只要传入的函数不接受任何参数（以 `()` 指示）且无返回（由 `=>` 指示）值（由函数值 `unit` 指示），就可以使用此函数。然后请注意，在循环体中，我使用 `callback` 来调用传递的参数函数对象。

幸运的是，我在程序的其他地方已经有了这样一个函数，名为 `timeFlies`。所以，我从 `main` 中将其传递给 `oncePerSecond` 函数。（您还会注意到，`timeFlies` 使用了一个 Scala 引入的类 `Console`，它的用途与 `System.out` 或新的 `java.io.Console` 类相同。这纯粹是一个审美问题；`System.out` 或 `Console` 都可以在这里使用。）

回页首

匿名函数，您的函数是什么?

现在，这个 `timeFlies` 函数似乎有点浪费 — 毕竟，它除了传递给 `oncePerSecond` 函数外毫无用处。所以，我根本不会正式定义它，如清单 5 所示：

清单 **5. Timer3.scala**

```
object Timer
{
  def oncePerSecond(callback: () => unit): unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): unit =
  {
    oncePerSecond(() =>
      Console.println("Time flies... oh, you get the idea."))
  }
}
```

在清单 5 中，主函数将一块任意代码作为参数传递给 `oncePerSecond`，看起来像来自 Lisp 或 Scheme 的 *lambda* 表达式，事实上，这是另一种闭包。这个匿名函数 再次展示了将函数当作一级公民处理的强大功能，它允许您在继承性以外对代码进行全新地泛化。（Strategy 模式的粉丝们可能已经开始唾沫横飞了。）

事实上，`oncePerSecond` 仍然太特殊了：它具有不切实际的限制，即回调将在每秒被调用。我可以通过接受第二个参数指明调用传递的函数的频率，来将其泛化，如清单 6 所示：

清单 **6. Timer4.scala**

```
object Timer
{
  def periodicCall(seconds: int, callback: () => unit): unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(seconds * 1000)
    }
  }

  def main(args: Array[String]): unit =
  {
    periodicCall(1, () =>
      Console.println("Time flies... oh, you get the idea."))
  }
}
```

这是函数语言中的公共主题：创建一个只做一件事情的高级抽象函数，让它接受一个代码块（匿名函数）作为参数，并从这个高级函数中调用这个代码块。例如，遍历一个对象集合。无需在 for 循环内部使用传统的 Java 迭代器对象，而是使用一个函数库在集合类上定义一个函数 — 通常叫做 “iter” 或 “map” — 接受一个带单个参数（要迭代的对象）的函数。例如，上述的 `Array` 类具有一个函数 `filter`，此函数在清单 7 中定义：

清单 **7. Array.scala** 的部分清单

```
class Array[A]
{
    // ...
        def filter  (p : (A) => Boolean) : Array[A] = ... // not shown
}
```

清单 7 声明 p 是一个接受由 `A` 指定的泛型参数的函数，然后返回一个布尔值。Scala 文档表明 `filter` “返回一个由满足谓词 p 的数组的所有元素组成的数组”。这意味着如果我想返回我的 Hello World 程序，查找所有以字母 G 开头的命令行参数，则可以编写像清单 8 一样简单的代码：

清单 **8. Hello, G-men!**

```
object HelloWorld
{
  def main(args: Array[String]): unit = {
    args.filter( (arg:String) => arg.startsWith("G") )
```

```
        .foreach( (arg:String) => Console.println("Found " + arg) )
  }
}
```

此处，`filter` 接受谓词，这是一个隐式返回布尔值（`startsWith()` 调用的结果）的匿名函数，并使用 `args` 中的每个元素来调用谓词。如果谓词返回 `true`，则它将此值添加到结果数组中。遍历了整个数组之后，它接受结果数组并将其返回，然后此数组立即用作 “foreach” 调用的来源，此调用执行的操作就像它名字的含义一样：`foreach` 接受另一个函数，并将此函数应用于数组中的每个元素（在本例中，仅显示每个元素）。

不难想象等同于上述 `HelloG.scala` 的 Java 是什么样的，而且也不难发现 Scala 版本非常简短，也非常清晰。

结束语

Scala 中的编程如此地熟悉，同时又如此地不同。相似之处在于您可以使用已经了解而且钟爱多年的相同的核心 Java 对象，但明显不同的是考虑将程序分解成部分的方式。在面向 *Java* 开发人员的 *Scala* 指南 的第一篇文章中，我仅仅简单介绍了 Scala 的功能。将来还有很多内容尚待挖掘，但是现在让我们陶醉在函数化的过程中吧！

参考资料

学习

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文 。
- “Java EE 迎合 Web 2.0”（Constantine Plotnikov、Artem Papkov、Jim Smith; developerWorks，2007 年 11 月）：指出与 Web 2.0 不兼容的 Java EE 平台的原则，并介绍弥合此裂缝的技术，其中包括 Scala。
- “*Java* 理论和实践：应用 fork-join 框架”（Brian Goetz，developerWorks，2007 年 11 月）：fork-join 抽象提供了一种自然的基于 Java 机制，来分解算法以有效利用硬件并行性。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员角度解释函数编程的优点和用法。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一部介绍 Scala 的图书，Scala 创始人 Martin Odersky 参与撰写。
- developerWorks Java 技术专区：有关 Java 编程方方面面的数百篇文章。

获得产品和技术

- Scala 主页：下载 Scala 并使用本系列开始学习！

讨论

- developerWorks 博客：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- (•) 太差！ (1)
- ( ) 需提高 (2)
- ( ) 一般；尚可 (3)
- ( ) 好文章 (4)
- ( ) 真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 类操作

理解 **Scala** 的类语法和语义

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 3 月 18 日

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- Scala 和 Java 一样使用类
- 用多少种方法构造类?
- 一些重要值
- 操作符
- Scala 内幕
- "更好的" Java
- 结束语
- 下载
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

文档选项

- 打印本页
- 将此页作为电子邮件发送
- 样例代码
- 英文原文

Java™ 开发人员可以将对象作为理解 Scala 的出发点。本文是面向 *Java* 开发人员的 *Scala* 指南 系列的第二期，作者 Ted Neward 遵循对一种语言进行评价的基本前提：一种语言的威力可以直接通过它集成新功能的能力衡量，在本文中就是指对复数的支持。跟随本文，您将了解在 Scala 中与类的定义和使用有关的一些有趣特性。

在上一期 文章 中，您只是稍微了解了一些 Scala 语法，这些是运行 Scala 程序和了解其简单特性的最基本要求。通过上一篇文章中的 Hello World 和 Timer 示例程序，您了解了 Scala 的 `Application` 类、方法定义和匿名函数的语法，还稍微了解了 `Array[]` 和一些类型推断方面的知识。Scala 还提供了很多其他特性，本文将研究 Scala 编程中的一些较复杂方面。

关于本系列

Ted Neward 潜心研究 Scala 编程语言，并带您跟他一起徜徉。在这个新的 developerWorks 系列 中，您将深入了解 Scala，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！毕竟，如果 Java 代码可以做到的话，又何必学习 Scala 呢?

Scala 的函数编程特性非常引人注目，但这并非 Java 开发人员应该对这门语言感兴趣的惟一原因。实际上，Scala 融合了函数概念和面向对象概念。为了让 Java 和 Scala 程序员感到得心应手，可以了解一下 Scala 的对象特性，看看它们是如何在语言方面与 Java 对应的。记住，其中的一些特性并不是直接对应，或者说，在某些情况下，"对应" 更像是一种类比，而不是直接的对应。不过，遇到重要区别时，我会指出来。

Scala 和 Java 一样使用类

我们不对 Scala 支持的类特性作冗长而抽象的讨论，而是着眼于一个类的定义，这个类可用于为 Scala 平台引入对有理数的支持（主要借鉴自 "Scala By Example"，参见 参考资料）：

清单 **1. rational.scala**

```
class Rational(n:Int, d:Int)
{
  private def gcd(x:Int, y:Int): Int =
  {
    if (x==0) y
    else if (x<0) gcd(-x, y)
    else if (y<0) -gcd(x, -y)
    else gcd(y%x, x)
  }
  private val g = gcd(n,d)

  val numer:Int = n/g
  val denom:Int = d/g
```

```
  def +(that:Rational) =
    new Rational(numer*that.denom + that.numer*denom, denom * that.denom)
  def -(that:Rational) =
    new Rational(numer * that.denom - that.numer * denom, denom * that.denom)
  def *(that:Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that:Rational) =
    new Rational(numer * that.denom, denom * that.numer)

  override def toString() =
    "Rational: [" + numer + " / " + denom + "]"
}
```

从词汇上看，清单 1 的整体结构与 Java 代码类似，但是，这里显然还有一些新的元素。在详细讨论这个定义之前，先看一段使用这个新 Rational 类的代码：

清单 **2. RunRational**

```
class Rational(n:Int, d:Int)
{
  // ... as before
}

object RunRational extends Application
{
  val r1 = new Rational(1, 3)
  val r2 = new Rational(2, 5)
  val r3 = r1 - r2
  val r4 = r1 + r2
  Console.println("r1 = " + r1)
  Console.println("r2 = " + r2)
  Console.println("r3 = r1 - r2 = " + r3)
  Console.println("r4 = r1 + r2 = " + r4)
}
```

清单 2 中的内容平淡无奇：先创建两个有理数，然后再创建两个 Rational，作为前面两个有理数的和与差，最后将这几个数回传到控制台上（注意，Console.println() 来自 Scala 核心库，位于 scala.* 中，它被隐式地导入每个 Scala 程序中，就像 Java 编程中的 java.lang 一样）。

回页首

用多少种方法构造类?

现在，回顾一下 Rational 类定义中的第一行：

清单 **3. Scala** 的默认构造函数

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

```
  // ...
```

您也许会认为清单 3 中使用了某种类似于泛型的语法，这其实是 `Rational` 类的默认的、首选的构造函数：*n* 和 *d* 是构造函数的参数。

Scala 优先使用单个构造函数，这具有一定的意义 —— 大多数类只有一个构造函数，或者通过一个构造函数将一组构造函数“链接”起来。如果需要，可以在一个 `Rational` 上定义更多的构造函数，例如：

清单 **4.** 构造函数链

```
class Rational(n:Int, d:Int)
{
  def this(d:Int) = { this(0, d) }
```

注意，Scala 的构造函数链通过调用首选构造函数（`Int,Int` 版本）实现 Java 构造函数链的功能。

实现细节

在处理有理数时，采取一点数值技巧将会有所帮助：也就是说，找到公分母，使某些操作变得更容易。如果要将 1/2 与 2/4 相加，那么 `Rational` 类应该足够聪明，能够认识到 2/4 和 1/2 是相等的，并在将这两个数相加之前进行相应的转换。

嵌套的私有 `gcd()` 函数和 `Rational` 类中的 *g* 值可以实现这样的功能。在 Scala 中调用构造函数时，将对整个类进行计算，这意味着将 *g* 初始化为 *n* 和 *d* 的最大公分母，然后用它依次设置 *n* 和 *d*。

回顾一下 清单 1 就会发现，我创建了一个覆盖的 `toString` 方法来返回 `Rational` 的值，在 `RunRational` 驱动程序代码中使用 `toString` 时，这样做非常有用。

然而，请注意 `toString` 的语法：定义前面的 `override` 关键字是必需的，这样 Scala 才能确认基类中存在相应的定义。这有助于预防因意外的输入错误导致难于觉察的 bug（Java 5 中创建 `@Override` 注释的动机也在于此）。还应注意，这里没有指定返回类型 —— 从方法体的定义很容易看出 —— 返回值没有用 `return` 关键字显式地标注，而在 Java 中则必须这样做。相反，函数中的最后一个值将被隐式地当作返回值（但是，如果您更喜欢 Java 语法，也可以使用 `return` 关键字）。

回页首

一些重要值

接下来分别是 `numer` 和 `denom` 的定义。这里涉及的语法可能让 Java 程序员认为 `numer` 和 `denom` 是公共的 `Int` 字段，它们分别被初始化为 *n-over-g* 和 *d-over-g*；但这种想法是不对的。

在形式上，Scala 调用无参数的 `numer` 和 `denom` 方法，这种方法用于创建快捷的语法以定义 accessor。`Rational` 类仍然有 3 个私有字段：*n*、*d* 和 *g*，但是，其中的 *n* 和 *d* 被默认定义为私有访问，而 *g* 则被显式地定义为私有访问，它们对于外部都是隐藏的。

此时，Java 程序员可能会问：“*n* 和 *d* 各自的 ‘setter’ 在哪里？” Scala 中不存在这样的 setter。Scala 的一个强大之处就在于，它鼓励开发人员以默认方式创建不可改变的对象。但是，也可使用语法创建修改 `Rational` 内部结构的方法，但是这样做会破坏该类固有的线程安全性。因此，至少对于这个例子而言，我将保持 `Rational` 不变。

当然还有一个问题，如何操纵 `Rational` 呢？与 `java.lang.String` 一样，不能直接修改现有的 `Rational` 的值，所以惟一的办法是根据现有类的值创建一个新的 `Rational`，或者从头创建。这涉及到 4 个名称比较古怪的方法：`+`、 `-`、`*` 和 `/`。

与其外表相反，这并非操作符重载。

回页首

操作符

记住，在 Scala 中一切都是对象。在上一篇 文章 中， 您看到了函数本身也是对象这一原则的应用，这使 Scala 程序员可以将函数赋予变量，将函数作为对象参数传递等等。另一个同样重要的原则是，一切都是函数；也就是说，在此处，命名为 add 的函数与命名为 + 的函数没有区别。在 Scala 中，所有操作符都是类的函数。只不过它们的名称比较古怪罢了。

在 Rational 类中，为有理数定义了 4 种操作。它们是规范的数学操作：加、减、乘、除。每种操作以它的数学符号命名：+、-、 * 和 /。

但是请注意，这些操作符每次操作时都构造一个新的 Rational 对象。同样，这与 java.lang.String 非常相似，这是默认的实现，因为这样可以产生线程安全的代码（如果线程没有修改共享状态 —— 默认情况下，跨线程共享的对象的内部状态也属于共享状态 —— 则不会影响对那个状态的并发访问）。

有什么变化？

一切都是函数，这一规则产生两个重要影响：

首先，您已经看到，函数可以作为对象进行操纵和存储。这使函数具有强大的可重用性，本系列 第一篇文章 对此作了探讨。

第二个影响是，Scala 语言设计者提供的操作符与 Scala 程序员认为应该 提供的操作符之间没有特别的差异。例如，假设提供一个“求倒数”操作符，这个操作符会将分子和分母调换，返回一个新的 Rational （即对于 Rational(2,5) 将返回 Rational(5,2)）。如果您认为 ~ 符号最适合表示这个概念，那么可以使用此符号作为名称定义一个新方法，该方法将和 Java 代码中任何其他操作符一样，如清单 5 所示：

清单 **5.** 求倒数

```
val r6 = ~r1
Console.println(r6) // should print [3 / 1], since r1 = [1 / 3]
```

在 Scala 中定义这种一元“操作符”需要一点技巧，但这只是语法上的问题而已：

清单 **6.** 如何求倒数

```
class Rational(n:Int, d:Int)
{
  // ... as before ...

  def unary_~ : Rational =
    new Rational(denom, numer)
}
```

当然，需要注意的地方是，必须在名称 ~ 之前加上前缀“unary_”，告诉 Scala 编译器它属于一元操作符。因此，该语法将颠覆大多数对象语言中常见的传统 reference-then-method 语法。

这条规则与“一切都是对象”规则结合起来，可以实现功能强大（但很简单）的代码：

清单 **7.** 求和

```
1 + 2 + 3 // same as 1.+(2.+(3))
r1 + r2 + r3 // same as r1.+(r2.+(r3))
```

当然，对于简单的整数加法，Scala 编译器也会“得到正确的结果”，它们在语法上是完全一样的。这意味着您可以开发与 Scala 语言“内置”的类型完全相同的类型。

Scala 编译器甚至会尝试推断具有某种预定含义的“操作符”的其他含义，例如 += 操作符。注意，虽然 Rational 类并没有显式地定义 +=，下面的代码仍然会正常运行：

清单 **8. Scala** 推断

```
var r5 = new Rational(3,4)
r5 += r1
Console.println(r5)
```

打印结果时，r5 的值为 [13 / 12]，结果是正确的。

回页首

Scala 内幕

记住，Scala 将被编译为 Java 字节码，这意味着它在 JVM 上运行。如果您需要证据，那么只需注意编译器生成以 0xCAFEBABE 开头的 *.class* 文件，就像 javac 一样。另外请注意，如果启动 JDK 自带的 Java 字节码反编译器（javap），并将它指向生成的 Rational 类，将会出现什么情况，如清单 9 所示：

清单 **9.** 从 **rational.scala** 编译的类

```
C:\Projects\scala-classes\code>javap -private -classpath classes Rational
Compiled from "rational.scala"
public class Rational extends java.lang.Object implements scala.ScalaObject{
    private int denom;
    private int numer;
    private int g;
    public Rational(int, int);
    public Rational unary_$tilde();
    public java.lang.String toString();
    public Rational $div(Rational);
    public Rational $times(Rational);
    public Rational $minus(Rational);
    public Rational $plus(Rational);
    public int denom();
    public int numer();
    private int g();
    private int gcd(int, int);
    public Rational(int);
    public int $tag();
}


C:\Projects\scala-classes\code>
```

Scala 类中定义的“操作符”被转换成传统 Java 编程中的方法调用，不过它们仍使用看上去有些古怪的名称。类中定义了两个构造函数：一个构造函数带有一个 int 参数，另一个带有两个 int 参数。您可能会注意到，大写的 Int 类型与 java.lang.Integer 有点相似，Scala 编译器非常聪明，会在类定义中将它们转换成常规的 Java 原语 int。

测试 Rational 类

一种著名的观点认为，优秀的程序员编写代码，伟大的程序员编写测试；到目前为止，我还没有对我的 Scala 代码严格地实践这一规则，那么现在看看将这个 Rational 类放入一个传统的 JUnit 测试套件中会怎样，如清单 10 所示：

清单 **10. RationalTest.java**

```
import org.junit.*;
import static org.junit.Assert.*;

public class RationalTest
{
    @Test public void test2ArgRationalConstructor()
    {
        Rational r = new Rational(2, 5);

        assertTrue(r.numer() == 2);
        assertTrue(r.denom() == 5);
    }

    @Test public void test1ArgRationalConstructor()
    {
        Rational r = new Rational(5);

        assertTrue(r.numer() == 0);
        assertTrue(r.denom() == 1);
            // 1 because of gcd() invocation during construction;
            // 0-over-5 is the same as 0-over-1
    }

    @Test public void testAddRationals()
    {
        Rational r1 = new Rational(2, 5);
        Rational r2 = new Rational(1, 3);

        Rational r3 = (Rational) reflectInvoke(r1, "$plus", r2); //r1.$plus(r2);

        assertTrue(r3.numer() == 11);
        assertTrue(r3.denom() == 15);
    }

    // ... some details omitted
}
```

除了确认 Rational 类运行正常之外，上面的测试套件还证明可以从 Java 代码中调用 Scala 代码（尽管在操作符方面有点不匹配）。当然，令人高兴的是，您可以将 Java 类迁移

SUnit

现在已经有一个基于 Scala 的单元测试套件，其名称为

至 Scala 类，同时不必更改支持这些类的测试，然后慢慢尝试 Scala。

您惟一可能觉得古怪的地方是操作符调用，在本例中就是 `Rational` 类中的 `+` 方法。回顾一下 `javap` 的输出，Scala 显然已经将 `+` 函数转换为 JVM 方法 `$plus`，但是 Java 语言规范并不允许标识符中出现 `$` 字符（这正是它被用于嵌套和匿名嵌套类名称中的原因）。

为了调用那些方法，需要用 Groovy 或 JRuby（或者其他对 `$` 字符没有限制的语言）编写测试，或者编写 `Reflection` 代码来调用它。我采用后一种方法，从 Scala 的角度看这不是那么有趣，但是如果您有兴趣的话，可以看看本文的代码中包含的结果（参见 下载）。

注意，只有当函数名称不是合法的 Java 标识符时才需要用这类方法。

SUnit。如果将 SUnit 用于清单 10 中的测试，则不需要基于 Reflection 的方法。基于 Scala 的单元测试代码将针对 Scala 类进行编译，所以编译器可以构成符号行。一些开发人员发现，使用 Scala 编写用于测试 POJO 的单元测试实际上更加有趣。

SUnit 是标准 Scala 发行版的一部分，位于 `scala.testing` 包中（要了解更多关于 SUnit 的信息，请参阅 参考资料）。

回页首

“更好的” Java

我学习 C++ 的时候，Bjarne Stroustrup 建议，学习 C++ 的一种方法是将它看作“更好的 C 语言”（参见 参考资料）。在某些方面，如今的 Java 开发人员也可以将 Scala 看作是“更好的 Java”，因为它提供了一种编写传统 Java POJO 的更简洁的方式。考虑清单 11 中显示的传统 `Person` POJO：

清单 **11. JavaPerson.java**（原始 **POJO**）

```
public class JavaPerson
{
    public JavaPerson(String firstName, String lastName, int age)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName()
    {
        return this.firstName;
    }
    public void setFirstName(String value)
    {
        this.firstName = value;
    }

    public String getLastName()
    {
        return this.lastName;
    }
    public void setLastName(String value)
    {
        this.lastName = value;
    }

    public int getAge()
    {
        return this.age;
```

```
    }
    public void setAge(int value)
    {
        this.age = value;
    }

    public String toString()
    {
        return "[Person: firstName" + firstName + " lastName:" + lastName +
            " age:" + age + " ]";
    }

    private String firstName;
    private String lastName;
    private int age;
}
```

现在考虑用 Scala 编写的对等物：

清单 **12. person.scala**（线程安全的 **POJO**）

```
class Person(firstName:String, lastName:String, age:Int)
{
    def getFirstName = firstName
    def getLastName = lastName
    def getAge = age

    override def toString =
        "[Person firstName:" + firstName + " lastName:" + lastName +
            " age:" + age + " ]"
}
```

这不是一个完全匹配的替换，因为原始的 `Person` 包含一些可变的 setter。但是，由于原始的 `Person` 没有与这些可变 setter 相关的同步代码，所以 Scala 版本使用起来更安全。而且，如果目标是减少 `Person` 中的代码行数，那么可以删除整个 `getFoo` 属性方法，因为 Scala 将为每个构造函数参数生成 accessor 方法 —— `firstName()` 返回一个 `String`，`lastName()` 返回一个 `String`，`age()` 返回一个 `int`。

即使必须包含这些可变的 setter 方法，Scala 版本仍然更加简单，如清单 13 所示：

清单 **13. person.scala**（完整的 **POJO**）

```
class Person(var firstName:String, var lastName:String, var age:Int)
{
    def getFirstName = firstName
    def getLastName = lastName
    def getAge = age

    def setFirstName(value:String):Unit = firstName = value
    def setLastName(value:String) = lastName = value
```

```
    def setAge(value:Int) = age = value

    override def toString =
        "[Person firstName:" + firstName + " lastName:" + lastName +
            " age:" + age + " ]"
}
```

注意，构造函数参数引入了 `var` 关键字。简单来说， `var` 告诉编译器这个值是可变的。因此，Scala 同时生成 accessor（`String firstName(void)`）和 mutator（`void firstName_$eq(String)`）方法。然后，就可以方便地创建 `setFoo` 属性 mutator 方法，它在幕后使用生成的 mutator 方法。

回页首

结束语

Scala 将函数概念与简洁性相融合，同时又未失去对象的丰富特性。从本系列中您可能已经看到，Scala 还修正了 Java 语言中的一些语法问题（后见之明）。

本文是面向 *Java* 开发人员的 *Scala* 指南 系列中的第二篇文章，本文主要讨论了 Scala 的对象特性，使您可以开始使用 Scala，而不必深入探究函数方面。应用目前学到的知识，您现在可以使用 Scala 减轻编程负担。而且，可以使用 Scala 生成其他编程环境（例如 Spring 或 Hibernate ）所需的 POJO。

但是，请继续关注本系列，下期文章将开始讨论 Scala 的函数方面。

回页首

下载

描述	名字	大小	下载方法
本文的示例 Scala 代码	j-scala02198-code.zip	2.6MB	**HTTP**

关于下载方法的信息

参考资料

学习

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文 。
- “面向 Java 开发人员的 Scala 指南：面向对象的函数编程”（Ted Neward，developerWorks，2008 年 1 月）：这是本系列的第一篇文章，概述了 Scala 和实现并发的函数方法等等。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- Programming in Scala（Martin Odersky、Lex Spoon 和 Bill Venners，Artima，2007 年 12 月）：这是第一部介绍 Scala 的图书。

Bjarne Stroustrup：设计并实现 C++，Stroustrup 认为 C++ 是一个“更出色的 C”（参见 Stroustrup 的 C++ Programming Language 中相应的内容）。

- The developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- Scala：下载 Scala 并使用本系列学习它！

- SUnit：这是标准 Scala 发行版的一部分， 位于 *scala.testing* 包中。

讨论

- developerWorks blog：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: Scala 控制结构内部揭密

级别: 初级

Ted Neward, 主管, Neward & Associates

2008 年 6 月 27 日

Scala 是专为 Java™ 平台编写的，因此其语法设计会使 Java 代码编码人员感觉很轻松。同时，Scala 为 JVM 提供了函数语言的固有的强大功能，并以这些函数设计概念为出发点。在这一期的 面向 *Java* 开发人员的 *Scala* 指南系列 文章中，Ted Neward 将介绍两种语言之间的细微差异，从一些控制结构（比如 `if`、`while` 和 `for`）开始介绍。正如您将要学习到的那样，Scala 为这些结构提供了一些在其 Java 等效物中无法获得的功能和复杂性。

迄今为止，在此 系列 中，我们已经讨论了 Scala 对生态环境的保真度，展示了 Scala 如何将众多的 Java 核心对象功能合并在一起。如果 Scala 只是编写对象的另一种方式，那么它不会有任何引人注意的地方，或者说不再那么功能强大。Scala 的函数概念和对象概念的合并，以及它对编程人员效率的重视，这些使得学习 Scala 语言比 Java-cum-Scala 编程人员所想象的体验更加复杂、更加微妙。

例如，对控制结构（比如 `if`、`while` 和 `for`）使用 Scala 的方法。尽管这些控制结构看起来类似一些老的、还比较不错的 Java 结构，但实际上 Scala 为它们增加了一些完全不同的特性。本月的文章是关于使用 Scala 控制结构时能够期望获得哪些东西的入门级读物，而不是在制造许多错误（并编写一堆错误代码）之后，让您冒着遭受挫折的风险去寻找差异。

关于本系列

Ted Neward 潜心研究 Scala 编程语言，并带您跟他一起徜徉。在这个新的 developerWorks 系列 中，您将深入了解 Scala，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！毕竟，如果 Java 代码可以做到的话，又何必学习 Scala 呢？

修订后的 Person.scala

在 本系列的上一篇文章 中，可以了解到 Scala 能够通过定义一些方法来定义 POJO，这些方法模仿基于 POJO 的环境所需的传统 "getter 和 setter"。在这篇文章发表之后，我收到了 Bill Venners 发来的电子邮件，Bill Venners 是即将发表的正式的 Scala 参考资料使用 *Scala* 编程（请参阅 参考资料）的合著者之一。Bill 指出了实现上述操作的一个更简单的方法，即使用 `scala.reflect.BeanProperty` 标注，如下所示:

清单 **1.** 修改后的 **Person.scala**

```
class Person(fn:String, ln:String, a:Int)
{
    @scala.reflect.BeanProperty
    var firstName = fn

    @scala.reflect.BeanProperty
    var lastName = ln

    @scala.reflect.BeanProperty
    var age = a

    override def toString =
        "[Person firstName:" + firstName + " lastName:" + lastName +
```

```
                " age:" + age + " ]"
    }
```

清单 1 中的方法（上一篇文章 中的清单 13 的修订版）为指定的 var 生成了 get/set 方法对。惟一的缺陷是这些方法并不实际存在于 Scala 代码中，因此其他 Scala 代码无法调用它们。这通常不是什么大问题，因为 Scala 将对为自己生成的字段使用已生成的方法；如果事先不知道，那么这些对您而言可能是一个惊喜。

在查看了清单 1 中的代码之后，最让我感到震动的是，Scala 并没有只演示组合函数概念和对象概念的强大威力，它还演示了自 Java 首次发布之后的 30 年里对象语言带来的一些益处。

回页首

控制是一种幻想

您将看到的许多奇怪的、不可思议的东西都可以归功于 Scala 的函数特性，因此，简单介绍一下函数语言开发和演变的背景可能非常有用。

在函数语言中，将越来越高级的结构直接构建到语言中是不常见的。此外，语言是通过一组核心原语结构定义的。在与将函数作为对象传递的功能结合之后，可用来定义功能的高阶函数 看起来 像是超出了核心语言的范围，但实际上它只是一个库。类似于任何库，此功能可以替换、扩充或扩展。

根据一组核心原语构建语言的合成 特性由来已久，可以追溯到 20 世纪 60 年代和 70 年代使用 Smalltalk、Lisp 和 Scheme 的时候。诸如 Lisp 和 Scheme 之类的语言因为它们在更低级别的抽象上定义更高级别抽象的能力而受到人们的狂热追捧。编程人员可以使用高级抽象，用它们构建更高级的抽象。如今听到讨论这个过程时，它通常是关于特定于域的语言（或 DSL）的（请参阅 参考资料）。实际上，它只是关于如何在抽象之上构建抽象的过程。

在 Java 语言中，惟一选择就是利用 API 调用完成此操作；在 Scala 中， 可以通过扩展语言本身实现它。试图扩展 Java 语言会带来创建极端场景（corner case）的风险，这些场景将威胁全局的稳定性。而试图扩展 Scala 则只意味着创建一个新库。

回页首

If 结构

我们将从传统的 if 结构开始 —— 当然，此结构必须是最容易处理的结构之一，不是吗？毕竟，从理论上说，if 只检查一个条件。如果条件为真，则执行后面跟着的代码。

但是，这种简单性可能带有欺骗性。传统上，Java 语言对 if 的 else 子句的使用是随意的，并且假定如果条件出错，可以只跳过代码块。但在函数语句中，情况不是这样。为了保持函数语句的算术特性，所有一切都必须以表达式计算的方式出现，包括 if 子句本身（对于 Java 开发人员，这正是三元操作符 —— ?: 表达式 —— 的工作方式）。

在 Scala 中，非真代码块（代码块的 else 部分）必须以与 if 代码块中值种类相同的形式呈现，并且必须产生同一种类的值。这意味着不论以何种方式执行代码，总会产生一个值。例如，请参见以下 Java 代码：

清单 **2.** 哪个配置文件？（**Java** 版）

```
// This is Java
String filename = "default.properties";
if (options.contains("configFile"))
  filename = (String)options.get("configFile");
```

因为 Scala 中的 if 结构自身就是一个表达式，所以重写上述代码会使它们成为清单 3 中所示的更正确的代码片段：

清单 **3.** 哪个配置文件？（**Scala** 版）

```
// This is Scala
val filename =
  if (options.contains("configFile"))
    options.get("configFile")
  else
    "default.properties"
```

尽管真正的赢家是 Scala，但可以通过编写代码将结果分配给 `val`，而不是 `var`。在设置之后，就无法对 `val` 进行更改，这与 Java 语言中 `final` 变量的操作方式是相同的。不可变本地变量最显著的副作用是很容易实现并发性。试图用 Java 代码实现同样的操作时，会带来许多不错的、易读的好代码，如清单 4 中所示：

清单 **4.** 哪个配置文件？（**Java** 版，三元式）

```
//This is Java
final String filename =
  options.contains("configFile") ?
    options.get("configFile") : "default.properties";
```

用代码评审解释这一点可能需要点技巧。也许这样做是正确的，但许多 Java 编程人员会不以为然并且询问“您做那个干什么”？

val 与 var

您可能想更多地了解 `val` 与 `var` 之间的不同，实际上，它们的不同之处在于 —— 一个是只读的值，另一个是可变的变量。通常，函数语言，特别是被认为是“纯”函数语言（不允许带有副作用，比如可变状态）的那些函数语言，只支持 `val` 概念；但是，因为 Scala 要同时吸引函数编程人员和命令/对象编程人员，所以这二种结构它都提供。

也就是说，Scala 编程人员通常应该首选 `val` 结构，并在明确需要可变性的时候选择 `var`。原因很简单：除了使编程更容易之外，`val` 还能确保程序的线程安全性，Scala 中的一个内在主题是：几乎每次认为需要可变状态时，其实都不需要可变状态。 让我们从不可变字段和本地变量（`val`）开始，这是展示上述情况的一种方法，甚至对最坚定的 Java 怀疑论者也是如此。从 Java 中的 `final` 开始介绍可能不是很合理，或许是因为 Java 的非函数特性，尽管此原因不可取。一些好奇的 Java 开发人员可能想尝试一下。

回页首

已公开的 while 结构

接下来，让我们来看一下 `while` 及其同胞 `do-while`。它们做的基本上是同一件事：测试一个条件，如果该条件为真，则继续执行提供的代码块。

通常，函数语言会避开 `while` 循环，因为 `while` 实现的大多数操作都可以使用递归来完成。函数语言真地非常类似于 递归。例如，可以考虑一下“Scala by Example”（请参阅 参考资料）中展示的 `quicksort` 实现，该实现可以与 Scala 实现一起使用：

清单 **5. Quicksort**（**Java** 版）

```
//This is Java
void sort(int[] xs) {
  sort(xs, 0, xs.length -1 );
}
void sort(int[] xs, int l, int r) {
  int pivot = xs[(l+r)/2];
  int a = l; int b = r;
  while (a <= b)
    while (xs[a] < pivot) { a = a + 1; }
    while (xs[b] > pivot) { b = b – 1; }
```

```
    if (a <= b) {
      swap(xs, a, b);
      a = a + 1;
      b = b – 1;
    }
  }
  if (l < b) sort(xs, l, b);
  if (b < r) sort(xs, a, r);
}
void swap(int[] arr, int i, int j) {
  int t = arr[i]; arr[i] = arr[j]; arr[j] = t;
}
```

不必深入太多的细节，就可以了解 `while` 循环的用法，它是通过数组中的各种元素进行迭代的，先找到一个支点，然后依次对每个子元素进行排序。毫不令人奇怪的是，`while` 循环也需要一组可变本地变量，在这里，这些变量被命名为 *a* 和 *b*，其中存储的是当前支点。注意，此版本甚至可以在循环自身中使用递归，两次调用循环本身，一次用于对列表左手边的内容进行排序，另一次对列表右手边的内容进行排序。

这足以说明清单 5 中的 `quicksort` 真的不太容易读取，更不用说理解它。现在来考虑一下 Scala 中的直接 等同物（这意味着该版本与上述版本尽量接近）：

清单 **6. Quicksort**（**Scala** 版）

```
//This is Scala
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length 1)
}
```

清单 6 中的代码看起来非常接近于 Java 版。也就是说，该代码很长，很难看，并且难以理解（特别是并发性那一部分），明显不具备 Java 版的一些优点。

So, I'll improve it ...

清单 **7. Quicksort**（更好的 **Scala** 版）

```
//This is Scala
def sort(xs: Array[Int]): Array[Int] =
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
```

显然，清单 7 中的 Scala 代码更简单一些。注意递归的使用，避免完全 `while` 循环。可以对 `Array` 类型使用 `filter` 函数，从而对其中的每个元素应用 "greater-than"、"equals" 和 "less-than" 函数。事实上，在引导装入程序之后，因为 `if` 表达式是返回某个值的表达式，所以从 `sort()` 返回的是 `sort()` 的定义中的（单个）表达式。

简言之，我已经将 `while` 循环的可变状态完全再次分解为传递给各种 `sort()` 调用的参数 —— 许多 Scala 狂热爱好者认为这是编写 Scala 代码的正确方式。

可能值得一提的是，Scala 本身并不介意您是否使用 `while` 代替迭代 —— 您会看到来自编译器的 "您在干什么，在做蠢事吗？" 的警告。Scala 也不会阻止您在可变状态下编写代码。但是，使用 `while` 或可变状态意味着牺牲 Scala 语言的另一个关键方面，即鼓励编写具有良好并行性的代码。只要有可能并且可行，"Scala 式作风" 会建议您优先在命令块上执行递归。

编写自己的语言结构

我想走捷径来讨论一下 Scala 的控制结构，做一些大多数 Java 开发人员根本无法相信的事 —— 创建自己的语言结构。

那些通过死读书学习语言的书呆子会发现一件有趣的事：`while` 循环（Scala 中的一个原语结构）可能只是一个预定义函数。Scala 文档以及假设的 "`While`" 定义中对此进行了解释说明：

```
// This is Scala
def While (p: => Boolean) (s: => Unit) {
  if (p) { s ; While(p)(s) }
}
```

上述语句指定了一个表达式，该表达式产生了一个布尔值和一个不返回任何结果的代码块（`Unit`），这正是 `while` 所期望的。

扩展这些代码行很容易，并且可以根据需要使用它们，只需导入正确的库即可。正如前面提到的，这是构建语言的综合方法。在下一节介绍 `try` 结构的时候，请将这一点牢记于心。

回页首

再三尝试

`try` 结构允许编写如下所示代码：

清单 **8.** 如果最初没有获得成功**……**

```
```

```
// This is Scala
val url =
  try {
    new URL(possibleURL)
  }
  catch {
    case ex: MalformedURLException =>
      new URL("www.tedneward.com")
  }
```

清单 8 中的代码与 清单 2 或 清单 3 中 `if` 示例中的代码相差甚远。实际上，它比使用传统 Java 代码编写更具技巧，特别是在您想捕获不可变位置上存储的值的时候（正如我在 清单 4 中最后一个示例中所做的那样）。这是 Scala 的函数特性的又一个优点!

清单 8 中所示的 `case ex:` 语法是另一个 Scala 结构（匹配表达式）的一部分，该表达式用于 Scala 中的模式匹配。我们将研究模式匹配，这是函数语言的一个常见特性，稍后将介绍它；现在，只把它看作一个将用于 `switch/case` 的概念，那么哪种 C 风格的 `struct` 将用于类呢?

现在，再来考虑一下异常处理。众所周知，Scala 支持异常处理是因为它是一个表达式，但开发人员想要的是处理异常的标准方法，并不仅仅是捕获异常的能力。在 AspectJ 中，是通过创建方面（aspect）来实现这一点的，这些方面围绕代码部分进行联系，它们是通过切入点定义的，如果想让数据库的不同部分针对不同种类异常采取不同行为，那么必须小心编写这些切入点 —— `SQLExceptions` 的处理应该不同于 `IOExceptions` 的处理，依此类推。

在 Scala 中，这只是微不足道的细节。请留神观察!

清单 **9.** 一个自定义异常表达式

```
// This is Scala
object Application
{
  def generateException()
  {
    System.out.println("Generating exception...");
    throw new Exception("Generated exception");
  }

  def main(args : Array[String])
  {
    tryWithLogging  // This is not part of the language
    {
      generateException
    }
    System.out.println("Exiting main()");
  }

  def tryWithLogging (s: => _) {
    try {
      s
    }
    catch {
      case ex: Exception =>
        // where would you like to log this?
        // I choose the console window, for now
        ex.printStackTrace()
```

```
    }
  }
}
```

与前面讨论过的 While 结构类似，tryWithLogging 代码只是来自某个库的函数调用（在这里，是来自同一个类）。可以在适当的地方使用不同的主题变量，不必编写复杂的切入点代码。

此方法的优点在于它利用了 Scala 的捕获一级结构中横切逻辑的功能 —— 以前只有面向方面的人才能对此进行声明。清单 9 中的一级结构捕获了一些异常（经过检查的和未经检查的都包括）并以特定方式进行处理。上述想法的副作用非常多，惟一的限制也许就是想象力了。您只需记得 Scala 像许多函数语言一样允许使用代码块（*aka* 函数）作为参数并根据需要使用它们即可。

回页首

"for" 生成语言

所有这些都引导我们来到了 Scala 控制结构套件的实际动力源泉：for 结构。该结构看起来像是 Java 的增强 for 循环的简单早期版，但它远比一般的 Java 编程人员开始设想的更强大。

让我们来看一下 Scala 如何处理集合上的简单顺序迭代，根据您的 Java 编程经验，我想您应该非常清楚该怎么做：

清单 **10.** 对一个对象使用 **for** 循环和对所有对象使用 **for** 循环

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1 to 10) // the left-arrow means "assignment" in Scala
      System.out.println("Counting " + i)
  }
}
```

此代码所做的正如您期望的那样，循环 10 次，并且每次都输出一些值。需要小心的是：表达式 "1 to 10" 并不意味着 Scala 内置了整数感知（awareness of integer）以及从 1 到 10 的计数方式。从技术上说，这里存在一些更微妙的地方：编译器使用 Int 类型上定义的方法 to 生成一个 Range 对象（Scala 中的任何东西都是对象，还记得吗？），该对象包含要迭代的元素。如果用 Scala 编译器可以看见的方式重新编写上述代码，那么该代码看起来很可能如下所示：

清单 **11.** 编译器看见的内容

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1.to(10)) // the left-arrow means "assignment" in Scala
      System.out.println("Counting " + i)
  }
}
```

实际上，Scala 的 `for` 并不了解那些成员，并且并不比其他任何对象类型做得更好。它所了解的是 `scala.Iterable`，scala.Iterable 定义了在集合上进行迭代的基本行为。提供 `Iterable` 功能（从技术上说，它是 Scala 中的一个特征，但现在将它视为一个接口）的任何东西都可以用作 `for` 表达式的核心。`List`、`Array`，甚至是您自己的自定义类型，都可以在 `for` 中使用。

特殊性

正如上面已经证明的那样，`for` 循环可以做许多事情，并不只是遍历可迭代的项列表。事实上，可以使用一个 `for` 循环在操作过程中过滤许多项，并在每个阶段都产生一个新列表：

让 **Scala** 与英语更接近

您可能已经注意到，理解清单 11 中的 Scala 的 `for` 循环版本更容易一些。这要感谢 `Range` 对象暗中将两端都包含在内，以下英语语言语法比 Java 语言更接近些。假如有一条 `Range` 语句说 "from 1 to 10, do this"，那么这意味着不再产生意外的 off-by-one 错误。

清单 **12.** 看一看还有哪些优点

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1 to 10; i % 2 == 0)
      System.out.println("Counting " + i)
  }
}
```

注意到清单 12 中 `for` 表达式的第二个子句了吗？它是一个过滤器，实际上，只有那些传递给过滤器（即计算 *true*）的元素 "向前传给" 了循环主体。在这里，只输出了 1 到 10 的偶数数字。

并不要求 `for` 表达式的各个阶段都成为过滤器。您甚至可以将一些完全平淡无奇的东西（从循环本身的观点来看）放入管道中。例如以下代码显示了在下一个阶段进行计算之前的 *i* 的当前值：

清单 **13.** 让我如何爱上您呢？别那么冗长

```
// This is Scala
object App
{
  def log(item : _) : Boolean =
  {
    System.out.println("Evaluating " + item)
    true
  }

  def main(args : Array[String]) =
  {
    for (val i <- 1 to 10; log(i); (i % 2) == 0)
      System.out.println("Counting " + i)
  }
}
```

在运行的时候，范围 1 到 10 中的每个项都将发送给 `log`，它将通过显式计算每个项是否为 *true* 来 "批准" 每个项。然后，`for` 的第三个子句将对这些项进行筛选，过滤出那些满足是偶数的条件的元素。因此，只将偶数传递给了循环主体本身。

简单性

在 Scala 中，可以将 Java 代码中复杂的一长串语句缩短为一个简单的表达式。例如，以下是遍历目录查找所有 .scala 文件并显示每个文件名称的方法：

清单 **14. Finding .scala**

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    val filesHere = (new java.io.File(".")).listFiles
    for (
      file <- filesHere;
      if file.isFile;
      if file.getName.endsWith(".scala")
    ) System.out.println("Found " + file)
  }
}
```

这种 for 过滤很常见（并且在此上下文中，分号很让人讨厌），使用这种过滤是为了帮助您做出忽略分号的决定。此外，Scala 允许将上述示例中的圆括号之间的语句直接作为代码块对待：

清单 **15. Finding .scala**（版本 **2**）

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    val filesHere = (new java.io.File(".")).listFiles
    for {
      file <- filesHere
      if file.isFile
      if file.getName.endsWith(".scala")
    } System.out.println("Found " + file)
  }
}
```

作为 Java 开发人员，您可能发现最初的圆括号加分号的语法更直观一些，没有分号的曲线括号语法很难读懂。幸运的是，这两种句法产生的代码是等效的。

一些有趣的事

在 `for` 表达式的子句中可以分配一个以上的项，如清单 16 中所示。

清单 **16.** 名称中有什么？

```
```

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    // Note the array-initialization syntax; the type (Array[String])
    // is inferred from the initialized elements
    val names = Array("Ted Neward", "Neal Ford", "Scott Davis",
      "Venkat Subramaniam", "David Geary")

    for {
      name <- names
      firstName = name.substring(0, name.indexOf(' '))
    } System.out.println("Found " + firstName)
  }
}
```

这被称为“中途赋值（midstream assignment）”，其工作原理如下：定义了一个新值 `firstName`，该值用于保存每次执行循环后的 `substring` 调用的值，以后可以在循环主体中使用此值。

这还引出了嵌套 迭代的概念，所有迭代都位于同一表达式中：

清单 **17. Scala grep**

```
// This is Scala
object App
{
  def grep(pattern : String, dir : java.io.File) =
  {
    val filesHere = dir.listFiles
    for (
      file <- filesHere;
      if (file.getName.endsWith(".scala") || file.getName.endsWith(".java"));
      line <- scala.io.Source.fromFile(file).getLines;
      if line.trim.matches(pattern)
    ) println(line)
  }

  def main(args : Array[String]) =
  {
    val pattern = ".*object.*"

    grep pattern new java.io.File(".")
  }
}
```

在此示例中，`grep` 内部的 `for` 使用了两个嵌套迭代，一个在指定目录（其中每个文件都与 `file` 连接在一起）中找到的所有文件上进行迭代，另一个迭代在目前正被迭代的文件（与 `line` 本地变量连接在一起）中发现的所有行上进行迭代。

使用 Scala 的 `for` 结构可以做更多的事，但目前为止提供的示例已足以表达我的观点：Scala 的 `for` 实际上是一条管道，它在将元素传递给循环主体之前

处理元素组成的集合，每次一个。此管道其中的一部分负责将更多的元素添加到管道中（生成器），一部分负责编辑管道中的元素（过滤器），还有一些负责处理中间的操作（比如记录）。无论如何，Scala 会带给您与 Java 5 中引入的“增强的 `for` 循环”不同的体验。

回页首

匹配

今天要了解的最后一个 Scala 控制结构是 `match`，它提供了许多 Scala 模式匹配功能。幸运的是，模式匹配会声明对某个值进行计算的代码块。首先，将执行代码块中最接近的匹配结果。因此，在 Scala 中可以包含以下代码：

清单 **18.** 一个简单的匹配

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    for (arg <- args)
      arg match {
        case "Java" => println("Java is nice...")
        case "Scala" => println("Scala is cool...")
        case "Ruby" => println("Ruby is for wimps...")
        case _ => println("What are you, a VB programmer?")
      }
  }
}
```

刚开始您可能将 Scala 模式匹配设想为支持 `String` 的“开关”，带有通常用作通配符的下划线字符，而这正是典型开关中的默认情况。但是，这样想会极大地低估该语言。模式匹配是许多（但不是大多数）函数语言中可以找到的另一个特性，它提供了一些有用的功能。

对于初学者（尽管这没什么好奇怪的），可能认为 `match` 表达式自身会产生一个值，该值可能出现在赋值语句的右边，正如 `if` 和 `try` 语句所做的那样。这一点本身也很有用，但匹配的真正威力体现在基于各种类型进行匹配时，而不是如上所述匹配单个类型的值，或者更多的时候，它是两种匹配的组合。

因此，假设您有一个声明返回 `Object` 的函数或方法 —— 在这里，Java 的 `java.lang.reflect.Method.invoke()` 方法的结果可能是一个好例子。通常，在使用 Java 语言计算结果时，首先应该确定其类型；但在 Scala 中，可以使用模式匹配简化该操作：

清单 **19.** 您是什么？

```
//This is Scala
object App
{
  def main(args : Array[String]) =
  {
    // The Any type is exactly what it sounds like: a kind of wildcard that
    // accepts any type
    def describe(x: Any) = x match {
      case 5 => "five"
      case true => "truth"
      case "hello" => "hi!"
```

```
      case Nil => "the empty list"
      case _ => "something else"
    }

    println describe(5)
    println describe("hello")
  }
}
```

因为 match 的很容易简单明了地描述如何针对各种值和类型进行匹配的能力，模式匹配常用于解析器和解释器中，在那里，解析流中的当前标记是与一系列可能的匹配子句匹配的。然后，将针对另一系列子句应用下一个标记，依此类推（注意，这也是使用函数语言编写许多语言解析器、编译器和其他与代码有关的工具的部分原因，这些函数语言中包括 Haskell 或 ML）。

关于模式匹配，还有许多可说的东西，但这些会将我们直接引导至 Scala 的另一个特性 *case* 类，我想将它留到下次再介绍。

回页首

结束语

Scala 在许多方面看起来都非常类似于 Java，但实际上只有 `for` 结构存在一些相似性。核心语法元素的函数特性不仅提供了一些有用的特性（比如已经提到的赋值功能），还提供了使用新颖有趣的方式扩展语言的能力，不必修改核心 `javac` 编译器本身。这使该语言更加符合 DSL 的定义（这些 DSL 是在现有语言的语法中定义的），并且更加符合编程人员根据一组核心原语（*a la* Lisp 或 Scheme）构建抽象的愿望。

关于 Scala，有如此多的内容可以谈论，但我们这个月的时间已经用完了。记得试用最新的 Scala bits（在撰写本文时是 2.7.0-final）并尝试提供的示例，感受一下该语言的操作（请参阅 参考资料）。请记住，到下一次的时候，Scala 会将一些有趣的（函数）特性放入编程中!

参考资料

学习

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文 。
- 面向 *Java* 开发人员的 *Scala* 指南（Ted Neward，developerWorks）：阅读该系列的所有文章。
- "跨越边界：Active Record 和 Java 编程中的特定于域的语言"（Bruce Tate，developerWorks，2006 年 4 月）：了解关于 Ruby 中的 DSL 的更多信息。
- "Java 语言中的函数编程"（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。
- "Scala by Example"（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima 于 2008 年 2 月提前印刷出版）：这是第一部介绍 Scala 的图书，由 Bill Venners 与他人合著。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：目前版本是 2.7.0-final。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 关于特征和行为

使用 **Scala** 版本的 **Java** 接口

级别: 初级

Ted Neward, 主管, Neward & Associates

2008 年 7 月 03 日

Scala 并不仅仅只给 JVM 引入了函数概念，它还为我们提供了一种对于面向对象语言设计的现代视角。在这一期的 面向 *Java* 开发人员的 *Scala* 指南 中，Ted Neward 介绍了 Scala 如何利用特征（trait）使对象更加简单、更易于构建。您将了解到， 特征与 Java™ 接口和 C++ 多重继承提供的传统极性既有相似之处，也有不同之处。

著名科学家、研究学者艾萨克.牛顿爵士有这样一句名言："如果说我看得比别人远一些，那是因为我站在巨人的肩膀上"。作为一名热心的历史和政治学家，我想对这位伟人的名言略加修改："如果说我看得比别人远一些，那是因为我站在历史的肩膀上"。而这句话又体现出另一位历史学家 George Santayana 的名言："忘记历史必将重蹈覆辙"。换句话说，如果我们不能回顾历史，从过去的错误（包括我们自己过去的经验）中吸取教训，就没有机会做出改进。

您可能会疑惑，这样的哲学与 Scala 有什么关系？继承就是我们要讨论的内容之一。考虑这样一个事实，Java 语言的创建已经是近 20 年前的事情，当时是"面向对象"的全盛时期。它设计用于模仿当时的主流语言 C++，尝试将使用这种语言的开发人员吸引到 Java 平台上来。毫无疑问，在当时看来，这样的决策是明智而且必要的，但回顾一下，就会发现其中有些地方并不像创建者设想的那样有益。

例如，在二十年前，对于 Java 语言的创建者来说，反映 C++ 风格的私有继承和多重继承是必要的。 自那之后，许多 Java 开发人开始为这些决策而后悔。在这一期的 Scala 指南中，我回顾了 Java 语言中多重继承和私有继承的历史。随后，您将看到 Scala 是怎样改写了历史，为所有人带来更大收益。

C++ 和 Java 语言中的继承

> 历史是人们愿意记录下来的事实。
>
> —拿破仑.波拿巴

从事 C++ 工作的人们能够回忆起，私有继承是从基类中获取行为的一种方法，不必显式地接受 *IS-A* 关系。将基类标记为"私有"允许派生类从该基类继承而来，而无需实际成为 一个基类。 但对自身的私有继承是未得到广泛应用的特性之一。继承一个基类而无法将它向下或向上转换到基类的理念是不明智的。

另一方面，多重继承往往被视为面向对象编程的必备要素。在建模交通工具的层次结构时，`SeaPlane` 无疑需要继承 `Boat`（使用其 `startEngine()` 和 `sail()` 方法）以及 `Plane`（使用其 `startEngine()` 和 `fly()` 方法）。`SeaPlane` 既是 `Boat`，也是 `Plane`，难道不是吗？

无论如何，这是在 C++ 鼎盛时期的想法。在快速转向 Java 语言时，我们认为多重继承与私有继承一样存在缺陷。所有 Java 开发人员都会告诉您，`SeaPlane` 应该继承 `Floatable` 和 `Flyable` 接口（或许还包括 `EnginePowered` 接口或基类）。继承接口意味着能够实现该类需要的所有方法，而不会遇到 虚拟多重继承 的难题（遇到这种难题时，要弄清楚在调用 `SeaPlane` 的 `startEngine()` 方法时应调用哪个基类的 `startEngine()`）。

> 关于本系列
>
> Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个新的 developerWorks 系列 中，您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

遗憾的是，彻底放弃私有继承和多重继承会使我们在代码重用方面付出昂贵的代价。Java 开发人员可能会因从虚拟多重继承中解放出来而高兴，但代价是程序员往往要完成辛苦而易于出错的工作。

回顾可重用行为

> 事情大致可以分为可能永远不会发生的和不重要的。
>
> —*William Ralph Inge*

JavaBeans 规范是 Java 平台的基础，它带来了众多 Java 生态系统作为依据的 POJO。我们都明白一点，Java 代码中的属性由 `get()/set()` 对管理，如清单 1 所示：

清单 **1. Person POJO**

```
//This is Java
public class Person
{
    private String lastName;
    private String firstName;
    private int age;

    public Person(String fn, String ln, int a)
    {
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String v) { firstName = v; }
    public String getLastName() { return lastName; }
    public void setLastName(String v) { lastName = v; }
    public int getAge() { return age; }
    public void setAge(int v) { age = v; }
}
```

这些代码看起来非常简单，编写起来也不难。但如果您希望提供通知支持 — 使第三方能够使用 POJO 注册并在变更属性时接收回调，事情会怎样？根据 JavaBeans 规范，必须实现 `PropertyChangeListener` 接口以及它的一个方法 `propertyChange()`。如果您希望允许任何 POJO 的 `PropertyChangeListener` 都能够对属性更改“投票”，那么 POJO 就需要实现 `VetoableChangeListener` 接口，该接口的实现又依赖于 `vetoableChange()` 方法的实现。

至少，事情应该是这样运作的。

实际上，希望成为属性变更通知接收者的用户必须实现 `PropertyChangeListener` 接口，发送者（本例中的 `Person` 类）必须提供接收该接口实例的公共方法和监听器需要监听的属性名称。最终得到更加复杂的 `Person`，如清单 2 所示：

清单 **2. Person POJO**，第 **2** 种形式

```
//This is Java
public class Person
{
    // rest as before, except that inside each setter we have to do something
    // like:
```

```
    // public setFoo(T newValue)
    // {
    //     T oldValue = foo;
    //     foo = newValue;
    //     pcs.firePropertyChange("foo", oldValue, newValue);
    // }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        // keep a reference to pcl
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        // find the reference to pcl and remove it
    }
}
```

保持引用属性变更监听器意味着 Person POJO 必须保留某种类型的集合类（例如 ArrayList）来包含所有引用。然后必须实例化、插入并移除 POJO — 由于这些操作不是原子操作，因此还必须包含恰当的同步保护。

最后，如果某个属性发生变化，属性监听器列表必须得到通知，通常通过遍历 PropertyChangeListener 的集合并对各元素调用 propertyChange() 来实现。此过程包括传入新的 PropertyChangeEvent 描述属性、原有值和新值，这是 PropertyChangeEvent 类和 JavaBeans 规范的要求。

在我们编写的 POJO 中，只有少数支持监听器通知，这并不意外。在这里要完成大量工作，必须手动地重复处理所创建的每一个 JavaBean/POJO。

除了工作还是工作 — 变通方法在哪里？

有趣的是，C++ 对于私有继承的支持在 Java 语言中得到了延续，今天，我们用它来解决 JavaBeans 规范的难题。一个基类为 POJO 提供了基本 add() 和 remove() 方法、集合类以及“firePropertyChanged()”方法，用于通知监听器属性变更。

我们仍然可以通过 Java 类完成，但由于 Java 缺乏私有继承，Person 类必须继承 Bean 基类，从而可向上转换 到 Bean。这妨碍了 Person 继承其他类。多重继承可能使我们不必处理后续的问题，但它也重新将我们引向了虚拟继承，而这是绝对要避免的。

针对这个问题的 Java 语言解决方案是运用众所周知的支持 类，在本例中是 PropertyChangeSupport：实例化 POJO 中的一个类，为 POJO 本身使用必要的公共方法，各公共方法都调用 Support 类来完成艰难的工作。更新后的 Person POJO 可以使用 PropertyChangeSupport，如下所示：

清单 **3. Person POJO**，第 **3** 种形式

```
//This is Java
import java.beans.*;

public class Person
{
    private String lastName;
    private String firstName;
    private int age;

    private PropertyChangeSupport propChgSupport =
        new PropertyChangeSupport(this);

    public Person(String fn, String ln, int a)
    {
```

```
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String newValue)
    {
        String old = firstName;
        firstName = newValue;
        propChgSupport.firePropertyChange("firstName", old, newValue);
    }

    public String getLastName() { return lastName; }
    public void setLastName(String newValue)
    {
        String old = lastName;
        lastName = newValue;
        propChgSupport.firePropertyChange("lastName", old, newValue);
    }

    public int getAge() { return age; }
    public void setAge(int newValue)
    {
        int old = age;
        age = newValue;
        propChgSupport.firePropertyChange("age", old, newValue);
    }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.removePropertyChangeListener(pcl);
    }
}
```

不知道您有何感想，但这段代码的复杂得让我想去重拾汇编语言。最糟糕的是，您要对所编写的每一个 POJO 重复这样的代码序列。清单 3 中的半数工作都是在 POJO 本身中完成的，因此无法被重用 — 除非是通过传统的“复制粘贴”编程方法。

现在，让我们来看看 Scala 提供什么样内容来实现更好的变通方法。

回页首

Scala 中的特征和行为重用

> 所有人都有义务考虑自己的性格特征。必须合理控制这些特征，而不去质疑他人的性格特征是否更适合自己。
>
> —西塞罗

Scala 使您能够定义处于接口和类之间的新型结构，称为特征（*trait*）。特征很奇特，因为一个类可以按照需要整合许多特征，这与接口相似，但它们还可

包含行为，这又与类相似。同样，与类和接口类似，特征可以引入新方法。但与类和接口不同之处在于，在特征作为类的一部分整合之前，不会检查行为的定义。或者换句话说，您可以定义出这样的方法，在整合到使用特征的类定义之前，不会检查其正确性。

特征听起来十分复杂，但一个实例就可以非常轻松地理解它们。首先，下面是在 Scala 中重定义的 `Person` POJO：

清单 **4. Scala** 的 **`Person` POJO**

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
{
}
```

您还可以确认 Scala POJO 具备基于 Java POJO 的环境中需要的 `get()/set()` 方法，只需在类参数 `firstName`、`lastName` 和 `age` 上使用 `scala.reflect.BeanProperty` 注释即可。现在，为简单起见，我们暂时不考虑这些方法。

如果 `Person` 类需要能够接收 `PropertyChangeListener`，可以使用如清单 5 所示的方式来完成此任务：

清单 **5. Scala** 的 **`Person` POJO** 与监听器

```
//This is Scala
object PCL
    extends java.beans.PropertyChangeListener
{
    override def propertyChange(pce:java.beans.PropertyChangeEvent):Unit =
    {
        System.out.println("Bean changed its " + pce.getPropertyName() +
            " from " + pce.getOldValue() +
            " to " + pce.getNewValue())
    }
}
object App
{
    def main(args:Array[String]):Unit =
    {
        val p = new Person("Jennifer", "Aloi", 28)

        p.addPropertyChangeListener(PCL)

        p.setFirstName("Jenni")
        p.setAge(29)

        System.out.println(p)
    }
}
```

注意，如何使用清单 5 中的 `object` 实现将静态方法注册为监听器 — 而在 Java 代码中，除非显式创建并实例化 `Singleton` 类，否则永远无法实现。这进一步证明了一个理论：Scala 从 Java 开发的历史 痛苦 中吸取了教训。

`Person` 的下一步是提供 `addPropertyChangeListener()` 方法，并在属性更改时对各监听器触发 `propertyChange()` 方法调用。在 Scala 中，以可

重用的方式完成此任务与定义和使用特征一样简单，如清单 6 所示。我将此特征称为 BoundPropertyBean，因为在 JavaBeans 规范中，“已通知”的属性称为绑定属性。

清单 **6.** 神圣的行为重用!

```
//This is Scala
trait BoundPropertyBean
{
    import java.beans._

    val pcs = new PropertyChangeSupport(this)

    def addPropertyChangeListener(pcl : PropertyChangeListener) =
        pcs.addPropertyChangeListener(pcl)

    def removePropertyChangeListener(pcl : PropertyChangeListener) =
        pcs.removePropertyChangeListener(pcl)

    def firePropertyChange(name : String, oldVal : _, newVal : _) : Unit =
        pcs.firePropertyChange(new PropertyChangeEvent(this, name, oldVal, newVal))
}
```

同样，我依然要使用 java.beans 包的 PropertyChangeSupport 类，不仅因为它提供了约 60% 的实现细节，还因为我所具备的行为与直接使用它的 JavaBean/POJO 相同。对“Support”类的其他任何增强都将传播到我的特征。不同之处在于 Person POJO 不需要再直接使用 PropertyChangeSupport，如清单 7 所示：

清单 **7. Scala** 的 **Person POJO**，第 **2** 种形式

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
    extends Object
    with BoundPropertyBean
{
    override def toString = "[Person: firstName=" + firstName +
        " lastName=" + lastName + " age=" + age + "]"
}
```

在编译后，简单查看 Person 定义即可发现它有公共方法 addPropertyChangeListener()、removePropertyChangeListener() 和 firePropertyChange()，就像 Java 版本的 Person 一样。实际上，Scala 的 Person 版本仅通过一行附加的代码即获得了这些新方法：类声明中的 *with* 子句将 Person 类标记为继承 BoundPropertyBean 特征。

遗憾的是，我还没有完全实现；Person 类现在支持接收、移除和通知监听器，但 Scala 为 firstName 成员生成的默认方法并没有利用它们。同样遗憾的是，这样编写的 Scala 没有很好的注释以自动地 生成利用 PropertyChangeSupport 实例的 get/set 方法，因此我必须自行编写，如清单 8 所示：

清单 **8. Scala** 的 **Person POJO**，第 **3** 种形式

```
                //This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
    extends Object
```

```
    with BoundPropertyBean
{
    def setFirstName(newvalue:String) =
    {
        val oldvalue = firstName
        firstName = newvalue
        firePropertyChange("firstName", oldvalue, newvalue)
    }

    def setLastName(newvalue:String) =
    {
        val oldvalue = lastName
        lastName = newvalue
        firePropertyChange("lastName", oldvalue, newvalue)
    }

    def setAge(newvalue:Int) =
    {
        val oldvalue = age
        age = newvalue
        firePropertyChange("age", oldvalue, newvalue)
    }

    override def toString = "[Person: firstName=" + firstName +
        " lastName=" + lastName + " age=" + age + "]"
}
```

应该具备的出色特征

特征不是一种函数编程 概念，而是十多年来反思对象编程的结果。实际上，您很有可能正在简单的 Scala 程序中使用以下特征，只是没有意识到而已：

清单 **9.** 再见，糟糕的 **main()**！

```
                //This is Scala
object App extends Application
{
    val p = new Person("Jennifer", "Aloi", 29)

    p.addPropertyChangeListener(PCL)

    p.setFirstName("Jenni")
    p.setAge(30)

    System.out.println(p)
}
```

`Application` 特征定义了一直都是手动定义的 `main()` 的方法。实际上，它包含一个有用的小工具：计时器，如果系统属性 `scala.time` 传递给了 `Application` 实现代码，它将为应用程序的执行计时，如清单 10 所示：

清单 **10.** 时间就是一切

```
$ scala -Dscala.time App
Bean changed its firstName from Jennifer to Jenni
Bean changed its age from 29 to 30
[Person: firstName=Jenni lastName=Aloi age=30]
[total 15ms]
```

回页首

JVM 中的特征

> 任何足够高级的技术都近乎魔术。
>
> *— Arthur C Clarke*

在这个时候，有必要提出这样一个问题，这种看似魔术的接口与方法结构（即 特征）是如何映射到 JVM 的。在清单 11 中，我们的好朋友 `javap` 展示了魔术背后发生了什么：

清单 **11. `Person` 内幕**

```
$ javap -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang
.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

请注意 `Person` 的类声明。该 POJO 实现了一个名为 `BoundPropertyBean` 的接口，这就是特征作为接口映射到 JVM 本身的方法。但特征方法的实现又是什么样的呢？请记住，编译器可以容纳所有技巧，只要最终结果符合 Scala 语言的语义含义即可。在这种情况下，它会将特征中定义的方法实现和字段声明纳入实现特征的类 `Person` 中。使用 `-private` 运行 `javap` 会使这更加显著 — 如果 `javap` 输出的最后两行体现的还不够明显（引用特征中定义的 *pcs* 值）：

清单 **12. Person** 内幕，第 **2** 种形式

```
$ javap -private -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    private final java.beans.PropertyChangeSupport pcs;
    private int age;
    private java.lang.String lastName;
    private java.lang.String firstName;
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

实际上，这个解释也回答了为何可以推迟特征方法的执行，直至用该检查的时候。因为在类实现特征的方法之前，它实际上并不是任何类的一“部分”，因此编译器可将方法的某些逻辑方面留到以后再处理。这非常有用，因为它允许特征在不了解实现特征的实际基类将是什么的情况下调用 `super()`。

关于特征的备注

在 `BoundPropertyBean` 中，我在 `PropertyChangeSupport` 实例的构建中使用了特征功能。其构造方法需要属性得到通知的 `bean`，在早先定义的特征中，我传入了“`this`”。由于在 `Person` 上实现之前并不会真正定义特征，“`this`”将引用 `Person` 实例，而不是 `BoundPropertyBean` 特征本身。特征的这个具体方面 — 定义的推迟解析 — 非常微妙，但对于此类的“迟绑定”来说可能非常强大。

对于 `Application` 特征的情况，有两部分很有魔力；`Application` 特征的 `main()` 方法为 `Java` 应用程序提供普适入口点，还会检查 `-Dscala.time` 系统属性，查看是否应该跟踪执行时间。但由于 `Application` 是一个特征，方法实际上会在子类上出现（`App`）。要执行此方法，必须创建 `App` 单体，也就是说构造 `App` 的一个实例，“处理”类的主体，这将有效地执行应用程序。只有在这种处理完成之后，特征的 `main()` 才会被调用并显示执行所耗费的时间。

虽然有些落后，但它仍然有效，尽管应用程序无权访问任何传入 `main()` 的命令行参数。它还表明特征的行为如何“下放到”实现类。

回页首

> 不是解决方法的一部分，就注定被淘汰。
>
> *— Henry J Tillman*

在将具体行为与抽象声明相结合以便为实现者提供便捷时，特征非常强大。例如，考虑经典的 Java 集合接口/类 `List` 和 `ArrayList`。`List` 接口保证此集合的内容能够按照插入时的次序被遍历，用更正规的术语来说，“位置语义得到了保证”。

`ArrayList` 是 `List` 的具体类型，在分配好的数组中存储内容，而 `LinkedList` 使用的是链表实现。`ArrayList` 更适合列表内容的随机访问，而 `LinkedList` 更适合在除了列表末尾以外的位置进行插入和删除操作。无论如何，这两种类之间存在大量相同的行为，它们继承了公共基类 `AbstractList`。

如果 Java 编程支持特征，它们应已成为出色的结构，能够解决“可重用行为，而无需诉诸于继承公共基类”之类的问题。特征可以作为 C++ “私有继承”机制，避免出现新 `List` 子类型是否应直接实现 `List`（还有可能忘记实现 `RandomAccess` 接口）或者扩展基类 `AbstractList` 的迷惑。这有时在 C++ 中称为“混合”，与 Ruby 的混合（或后文中探讨的 Scala 混合）有所不同。

在 Scala 文档集中，经典的示例就是 `Ordered` 特征，它定义了名字很有趣的方法，以提供比较（以及排序）功能，如清单 13 所示：

清单 **13.** 顺序、顺序

```
//This is Scala
trait Ordered[A] {
  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

在这里，`Ordered` 特征（具有参数化类型，采用 Java 5 泛型方式）定义了一个抽象方法 `compare`，它应获得一个 *A* 作为参数，并需要在“小于”的情况下返回小于 1 的值，在“大于”的情况下返回大于 1 的值，在相等的情况下返回 0。然后它继续使用 `compare()` 方法和更加熟悉的 `compareTo()` 方法（`java.util.Comparable` 接口也使用该方法）定义关系运算符（< 和 > 等）。

回页首

Scala 和 Java 兼容性

> 一张图片胜过千言万语。一个界面胜过上千图片。
>
> *—Ben Shneiderman*

实际上，伪实现继承并不是 Scala 内特征的最常见应用或最强大用法，与此不同，特征在 Scala 内作为 Java 接口的基本替代项。希望使用 Scala 的 Java 程序员也应熟悉特征，将其作为使用 Scala 的一种机制。

我在本系列的文章中一直强调，编译后的 Scala 代码并非总是能够保证 Java 语言的特色。例如，回忆一下，Scala 的“名字很有趣的方法”（例如“+”或“\”），这些方法往往会使用 Java 语言语法中不直接可用的字符编码（“$”就是一个需要考虑的严重问题）。出于这方面的原因，创建“Java 可调用”的接口往往要求深入研究 Scala 代码。

这个特殊示例有些憋足，*Scala* 主义者 通常并不需要特征提供的间接层（假设我并未使用“名字很有趣的方法”），但概念在这里十分重要。在清单 14 中，我希望获得一个传统的 Java 风格工厂，生成 `Student` 实例，就像您经常在各种 Java 对象模型中可以看到的那样。最初，我需要一个兼容 Java 的接口，接合到 `Student`：

清单 **14.** 我，学生

```
//This is Scala
trait Student
{
    def getFirstName : String;
    def getLastName : String;
    def setFirstName(fn : String) : Unit;
    def setLastName(fn : String) : Unit;

    def teach(subject : String)
}
```

在编译时，它会转换成 POJI：Plain Old Java Interface，查看 `javap` 会看到这样的内容：

清单 **15.** 这是一个 **POJI**！

```
$ javap Student
Compiled from "Student.scala"
public interface Student extends scala.ScalaObject{
    public abstract void setLastName(java.lang.String);
    public abstract void setFirstName(java.lang.String);
    public abstract java.lang.String getLastName();
    public abstract java.lang.String getFirstName();
    public abstract void teach(java.lang.String);
}
```

接下来，我需要一个类成为工厂本身。通常，在 Java 代码中，这应该是类上的一个静态方法（名称类似于“`StudentFactory`”），但回忆一下，Scala 并没有此类的实例方法。我认为这就是我在这里希望得到的结论，因此，我创建了一个 `StudentFactory` 对象，将我的 `Factory` 方法放在那里：

清单 **16.** 我构造 **Students**

```
                //This is Java
object StudentFactory
{
    class StudentImpl(var first:String, var last:String, var subject:String)
        extends Student
    {
        def getFirstName : String = first
        def setFirstName(fn: String) : Unit = first = fn
        def getLastName : String = last
        def setLastName(ln: String) : Unit = last = ln

        def teach(subject : String) =
            System.out.println("I know " + subject)
    }

    def getStudent(firstName: String, lastName: String) : Student =
```

```
    {
        new StudentImpl(firstName, lastName, "Scala")
    }
}
```

嵌套类 `StudentImpl` 是 `Student` 特征的实现，因而提供了必需的 `get()/set()` 方法对。切记，尽管特征可以具有行为，但它根据 JVM 作为接口建模这一事实意味着尝试实例化特征将产生错误 —— 表明 `Student` 是抽象的。

当然，这个简单示例的目的在于编写出一个 Java 应用程序，使之可以利用这些由 Scala 创建的新对象：

清单 **17.** 学生 **Neo**

```
//This is Java
public class App
{
    public static void main(String[] args)
    {
        Student s = StudentFactory.getStudent("Neo", "Anderson");
        s.teach("Kung fu");
    }
}
```

运行此代码，您将看到："I know Kung fu"。（我知道，我们经过了漫长的设置过程，只是得到了一部廉价电影的推介）。

回页首

结束语

> 人们不喜欢思考。思考总是要得出结论。而结论并非总是令人愉快。
>
> *— Helen Keller*

特征提供了在 Scala 中分类和定义的强大机制，目的在于定义一种接口，供客户端使用，按照 传统 Java 接口的形式定义；同时提供一种机制，根据特征内定义的其他行为来继承行为。或许我们需要的是一种全新的继承术语，用于 描述特征和实现类之间的关系。

除了本文所述内容之外，还有很多种方法可以使用特征，但本系列文章的部分目的就在于提供关于这种语言的足够信息，鼓励您在家中进一步开展实验；下载 Scala 实现，亲自试用，查看 Scala 可插入当前 Java 系统的什么位置。此外，如果您发现 Scala 非常有用，如果您对本文有任何意见建议，或者（叹息）您发现代码、行文中存在 bug，请 给我留言，让我知道您的意见。

这一期的内容到此为止，实用主义爱好者请关注本系列的下一期文章。

参考资料

学习

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文 。
- 面向 *Java* 开发人员的 *Scala* 指南（Ted Neward，developerWorks，2008 年）：阅读本系列的所有文章。
- "A Tour of Scala: Mixin Class Composition"（Scala 语言文档）：进一步了解 Scala 中的继承和混合类组合。

- “Thinking in Java: Comparing C++ and Java”（Bruce Eckel，Java Coffee Break）：这份来自 Bruce 图书的摘要强调了 Java 语言与 C++ 之间的差异。
- “Dinosaurs Can Take the Pain”（Cay Horstmann，Java.net，2008 年 1 月）：借用 Java 程序员的想法，以创新的方式应对僵局，提出某些解决方案。
- “What are your Java pain points, really?”（Bill Venners，Artima.com，2007 年 2 月）：Artima 论坛上的一个讨论主题，截至本文撰写完成之日，回复已达到 264 条。
- Interoperability happens - Languages：进一步了解 Ted Neward 对于语言设计和发展演进的思考。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：这一片简短的、代码驱动的 Scala 介绍文章，包括本文中使用的 Quicksort 应用程序（PDF 格式）。
- Programming in Scala（Martin Odersky、Lex Spoon 和 Bill Venners；Artima；上次刊印时间为 2008 年 2 月）：这是第一部介绍 Scala 的图书，由 Bill Venners 合著。
- developerWorks Java 技术专区：数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scala：目前最新版本是 2.7.0-final。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

◉太差！ (1)
○需提高 (2)
○一般；尚可 (3)
○好文章 (4)
○真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 实现继承

当 **Scala** 继承中的对象遇到函数

级别： 中级

Ted Neward, 主管, Neward & Associates

2008 年 8 月 04 日

Scala 对实现继承的支持与 Java™ 语言一样丰富 — 但 Scala 的继承带来了一些惊喜。这个月，Ted Neward 介绍了以 Scala 方式完成的多态，还介绍了混合函数与面向对象的语言风格，同时使您依然能够完美地映射到 Java 平台的继承模型。

近十几年来，面向对象语言设计的要素一直是继承的核心。不支持继承的语言（如 Visual Basic）被嘲讽是“玩具语言”，不适合真正的工作。与此同时，支持继承的语言所采用的支持方法五花八门，导致了许多争论。多重继承是否真的必不可少（就像 C++ 的创作者认定的那样），它是否不必要而丑陋的（就像 C# 和 Java 的创作者坚信的那样）？Ruby 和 Scala 是两种较新的语言，采取了多重继承的这种方法 — 正如我在上期介绍 Scala 的特征时所讨论的那样（参见 参考资料）。

与所有 杰出的语言一样，Scala 也支持实现继承（参见 参考资料）。在 Java 语言中，单一实现继承模型允许您扩展基类，添加新方法和字段等。尽管存在某些句法变更，Scala 的实现继承依然类似于 Java 语言中的实现。不同的是 Scala 融合了对象和函数语言设计，这非常值得我们在本期文章中进行讨论。

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个新的 developerWorks 系列 中，您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

普通 Scala 对象

与本系列之前的文章类似，我将使用 `Person` 类作为起点，探索 Scala 的继承系统。清单 1 展示了 `Person` 的类定义：

清单 **1.** 嘿，我是人类

```
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
  def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"
}
```

`Person` 是一个非常简单的 POSO（普通 Scala 对象，Plain Old Scala Object），具有三个只读字段。您可能会想起，要使这些字段可以读写，只需将主构造函数声明中的 `val` 更改为 `var` 即可

无论如何，使用 `Person` 类型也非常简单，如清单 2 所示：

清单 **2. PersonApp**

```
// This is Scala
object PersonApp
{
  def main(args : Array[String]) : Unit =
  {
    val bindi = new Person("Tabinda", "Khan", 38)
    System.out.println(bindi)
  }
}
```

这算不上什么令人惊讶的代码，但给我们提供了一个起点。

回页首

Scala 中的抽象方法

随着该系统的发展，越来越明显地意识到 Person 类缺乏一个成为 Person 的重要部分，这个部分是做些事情 的行为。许多人都会根据我们在生活中的作为来定义自己，而不是根据现有和占用的空间。因此，我会添加一个新方法，如清单 3 所示，这赋予了 Person 一些意义：

清单 **3.** 很好，做些事情!

```
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"

  def doSomething = // uh.... what?
}
```

这带来了一个问题：Person 的用途究竟是什么？有些 Person 绘画，有些唱歌，有些编写代码，有些玩视频游戏，有些什么也不做（问问十几岁青少年的父母）。因此，我会为 Person 创建 子类，而不是尝试去将这些活动直接整合到 Person 本身之中，如清单 4 所示：

清单 **4.** 这个人做的事情很少

```
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"

  def doSomething = // uh.... what?
}

class Student(firstName:String, lastName:String, age:Int)
  extends Person(firstName, lastName, age)
{
  def doSomething =
```

```
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }
}
```

当尝试编译代码时，我发现无法编译。这是因为 `Person.doSomething` 方法的定义无法工作；这个方法需要一个完整的主体（或许可抛出异常来表示它应在继承类中被覆盖），或者不需要主体，类似于 Java 代码中抽象方法的工作方式。我在清单 5 中尝试使用抽象的方法：

清单 **5.** 抽象类 **Person**

```
// This is Scala
abstract class Person(val firstName:String, val lastName:String, val age:Int)
{
  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"

  def doSomething; // note the semicolon, which is still optional
                   // but stylistically I like having it here
}

class Student(firstName:String, lastName:String, age:Int)
  extends Person(firstName, lastName, age)
{
  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }
}
```

请注意，我如何使用 `abstract` 关键字装饰 `Person` 类。`abstract` 为编译器指出，是的，这个类应该是抽象的。在这方面，Scala 与 Java 语言没有区别。

对象，遇到函数

由于 Scala 融合了对象和函数语言风格，我实际上建模了 `Person`（如上所述），但并未创建子类型。这有些古怪，但强调了 Scala 对于这两种设计风格的整合，以及随之而来的有趣理念。

回忆 前几期文章，Scala 将函数作为值处理，就像处理语言中的其他值一样，例如 `Int`、`Float` 或 `Double`。在建模 `Person` 时，我可以利用这一点来获得 `doSomething`，不仅将其作为一种继承类中覆盖的方法，还将其作为可调用、替换、扩展的 函数值。清单 6 展示了这种方法：

清单 **6.** 努力工作的人

```
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
  var doSomething : (Person) => Unit =
    (p:Person) => System.out.println("I'm " + p + " and I don't do anything yet!");

  def work() =
```

```
    doSomething(this)

  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"
}

object App
{
  def main(args : Array[String]) =
  {
    val bindi = new Person("Tabinda", "Khan", 38)
    System.out.println(bindi)

    bindi.work()

    bindi.doSomething =
      (p:Person) => System.out.println("I edit textbooks")

    bindi.work()

    bindi.doSomething =
      (p:Person) => System.out.println("I write HTML books")

    bindi.work()
  }
}
```

将函数作为第一建模工具是 Ruby、Groovy 和 ECMAScript（也就是 JavaScript）等动态语言以及许多函数语言的常用技巧。尽管其他语言也可以用函数作为建模工具，（C++ 通过函数指针和/或成员函数指针实现，Java 代码中通过接口引用的匿名内部类实现），但所需的工作比 Scala（以及 Ruby、Groovy、ECMAScript 和其他语言）多得多。这是函数语言使用的“高阶函数”概念的扩展。（关于高阶函数的更多内容，请参见 参考资料。）

多亏 Scala 将函数视为值，这样您就可以在运行时需要切换功能的时候利用函数值。可将这种方法视为角色模式 —— Gang of Four 战略模式的一种变体，在这种模式中，对象角色（例如 `Person` 的当前就职状态）作为运行时值得到了更好的表现，比静态类型的层次结构更好。

回页首

层次结构上层的构造函数

回忆一下编写 Java 代码的日子，有时继承类需要从构造函数传递参数至基类构造函数，从而使基类字段能够初始化。在 Scala 中，由于主构造函数出现在类声明中，不再是类的“传统”成员，因而将参数传递到基类将成为一个全新维度的问题。

在 Scala 中，主构造函数的参数在 `class` 行传递，但您也可以为这些参数使用 `val` 修饰符，以便在类本身上轻松引入读值器（对于 `var`，则为写值器）。

因此，清单 5 中的 Scala 类 `Person` 转变为清单 7 中的 Java 类，使用 javap 查看：

清单 **7.** 请翻译一下

```
// This is javap
C:\Projects\scala-inheritance\code>javap -classpath classes Person
```

```
Compiled from "person.scala"
public abstract class Person extends java.lang.Object implements scala.ScalaObje
ct{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public abstract void doSomething();
    public int age();
    public java.lang.String lastName();
    public java.lang.String firstName();
    public int $tag();
}
```

JVM 的基本规则依然有效：Person 的继承类在构造时向基类传递某些内容，而不管语言强调的是什么。（实际上，这并非完全 正确，但在语言尝试规避此规则时，JVM 会表现失常，因此大多数语言仍然坚持通过某种方法为其提供支持。）当然，Scala 需要坚守此规则，因为它不仅需要保持 JVM 正常运作，而且还要保持 Java 基类正常运作。这也就是说，无论如何，Scala 必须实现一种语法，允许继承类调用基类，同时保留允许我们在基类上引入读值器和写值器的语法。

为了将此放到更具体的上下文中，假设我通过以下方式编写了 清单 5 中的 Student 类：

清单 **8.** 坏学生！

```
// This is Scala
// This WILL NOT compile
class Student(val firstName:String, val lastName:String, val age:Int)
  extends Person(firstName, lastName, age)
{
  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }
}
```

本例中的编译器将运行很长一段时间，因为我尝试为 Student 类引入一组新方法（firstName、lastName 和 age）。这些方法将与 Person 类上名称类似的方法彼此冲突，Scala 编译器不一定了解我是否正在尝试覆盖基类方法（这很糟糕，因为我可以在这些基类方法后隐藏实现和字段），或者引入相同名称的新方法（这也很糟糕，因为我可以在这些基类方法后隐藏实现和字段）。简而言之，您将看到如何成功覆盖来自基类的方法，但那并不是我们目前要追求的目标。

您还应注意到，在 Scala 中，Person 构造函数的参数不必一对一地与传递给 Student 的参数联系起来；这里的规则实际上与 Java 构造函数的规则完全相同。我们这样做只是为了便于阅读。同样，Student 可要求额外的构造函数参数，与在 Java 语言中一样，如清单 9 所示：

清单 **9.** 苛求的学生！

```
// This is Scala
class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
```

```
  }
}
```

您又一次看到了 Scala 代码与 Java 代码有多么的相似，至少涉及继承和类关系时是这样。

回页首

语法差异

至此，您可能会对语法的细节感到迷惑。毕竟 Scala 并未像 Java 语言那样将字段与方法区分开来。这实际上是一项深思熟虑的设计决策，允许 Scala 程序员轻而易举地向使用基类的用户“隐藏”字段和方法之间的差异。考虑清单 10：

清单 **10.** 我是什么？

```
// This is Scala
abstract class Person(val firstName:String, val lastName:String, val age:Int)
{
  def doSomething

  def weight : Int

  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
                          " age="+age+"]"
}

class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def weight : Int =
    age // students are notoriously skinny

  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }
}

class Employee(firstName:String, lastName:String, age:Int)
  extends Person(firstName, lastName, age)
{
  val weight : Int = age * 4 // Employees are not skinny at all

  def doSomething =
  {
    System.out.println("I'm working hard, hon, I swear! (Pass the beer, guys!)")
  }
}
```

注意查看如何定义 `weight` 使其不带有任何参数并返回 `Int`。这是“无参数方法”。因为它看上去与 Java 语言中的“专有”方法极其相似，Scala 实际上允许将 `weight` 定义为一种方法（如 `Student` 中所示），也允许将其定义为字段/存取器（如 `Employee` 中所示）。这种句法决策使您在抽象类继承的实现方面有一定的灵活性。请注意，在 Java 中，即便是在同一个类中，只有通过 get/set 方法来访问各字段时，才能获得类似的灵活性。不知道判断正确与否，但我认为只有少数 Java 程序员会用这种方式编写代码，因此不经常使用灵活性。此外，Scala 的方法可像处理公共成员一样轻松地处理隐藏/私有成员。

回页首

从 `@Override` 到 `override`

继承类经常需要更改在其某个基类内定义的方法的行为；在 Java 代码中，我们通过为继承类添加相同名称、相同签名的新方法来处理这个问题。这种方法的缺点在于签名录入的错误或含糊不清可能会导致没有征兆的故障，这也就意味着代码可以编译，但在运行时无法正确完成操作。

为解决这个问题，Java 5 编译器引入了 `@Override` 注释。`@Override` 验证引入继承类的方法实际上已经覆盖了基类方法。在 Scala 中，`override` 已经成为语言的一部分，几乎可以忘记它会生成编译器错误。因而，继承 `toString()` 方法应如清单 11 所示：

清单 **11.** 这是继承的结果

```
// This is Scala
class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def weight : Int =
    age // students are notoriously skinny

  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }

  override def toString = "[Student: firstName="+firstName+
                          " lastName="+lastName+" age="+age+
                          " subject="+subject+"]"
}
```

非常简单明了。

回页首

敲定

当然，允许继承覆盖的反面就是采取措施防止它：基类需要禁止子类更改其基类行为，或禁止任何类型的继承类。在 Java 语言中，我们通过为方法应用修饰符 `final` 来实现这一点，确保它不会被覆盖。此外，也可以为类整体应用 `final`，防止继承。实现层次结构在 Scala 中的效果是相同的：我们可以向方法应用 `final` 来防止子类覆盖它，也可应用于类声明本身来防止继承。

牢记，所有这些关于 `abstract`、`final` 和 `override` 的讨论都同样适用于“名字很有趣的方法”（Java 或 C# 或 C++ 程序员可能会这样称呼运算符），与应用于常规名称方法的效果相同。因此，我们常常会定义一个基类或特征，为数学函数设定某些预期（可以称之为“Mathable”），这些函数定义抽象成员函数“+”、“-”、“*”和“/”，另外还有其他一些应该支持的数学运算，例如 *pow* 或 *abs*。随后，其他程序员可创建其他类型 — 可能是一个 `Matrix` 类，实现或扩展“Mathable”，定义一些成员，看上去就像 Scala 以开箱即用的方式提供的内置算术类型。

回页首

差别在于……

如果 Scala 能够如此轻松地映射到 Java 继承模型（就像本文至此您看到的那样），就应该能够从 Java 语言继承 Scala 类，或反之。实际上，这必须 可行，因为 Scala 与其他编译为 Java 字节码的语言相似，必须生成继承自 `java.lang.Object` 的对象。请注意，Scala 类可能也要继承自其他内容，例如特征，因此实际继承的解析和代码生成的工作方式可能有所不同，但最终我们必须能够以某种形式继承 Java 基类。（切记，特征类似于有行为的接口，Scala 编译器将特征分成接口并将实现推入特征编译的目标类中，通过这种方式来使之运作。）

但结果表明，Scala 的类型层次结构与 Java 语言中的对应结构略有不同；从技术上来讲，所有 Scala 类继承的基类（包括 `Int`、`Float`、`Double` 和其他数字类型）都是 `scala.Any` 类型，这定义了一组核心方法，可在 Scala 内的任意类型上使用：`==`、`!=`、`equals`、`hashCode`、`toString`、`isInstanceOf` 和 `asInstanceOf`，大多数方法通过名称即可轻松理解。在这里，Scala 划分为两大分支，“原语类型”继承自 `scala.AnyVal`；“类类型”继承自 `scala.AnyRe`。（`scala.ScalaObject` 又继承自 `scala.AnyRef`。）

通常，这并不是您要直接去操心的方面，但在考虑跨两种语言的继承时，可能会带来某些非常有趣的副作用。例如，考虑清单 12 中的 `ScalaJavaPerson`：

清单 **12.** 混合!

```
                                // This is Scala
class ScalaJavaPerson(firstName:String, lastName:String, age:Int)
  extends JavaPerson(firstName, lastName, age)
{
  val weight : Int = age * 2 // Who knows what Scala/Java people weigh?

  override def toString = "[SJPerson: firstName="+firstName+
                          " lastName="+lastName+" age="+age+"]"
}
```

……它继承自 `JavaPerson`：

清单 **13.** 看起来是否眼熟?

```
                                // This is Java
public class JavaPerson
{
    public JavaPerson(String firstName, String lastName, int age)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName()
    {
        return this.firstName;
    }
    public void setFirstName(String value)
    {
        this.firstName = value;
```

```
    }

    public String getLastName()
    {
        return this.lastName;
    }
    public void setLastName(String value)
    {
        this.lastName = value;
    }

    public int getAge()
    {
        return this.age;
    }
    public void setAge(int value)
    {
        this.age = value;
    }

    public String toString()
    {
        return "[Person: firstName" + firstName + " lastName:" + lastName +
            " age:" + age + " ]";
    }

    private String firstName;
    private String lastName;
    private int age;
}
```

在编译 ScalaJavaPerson 时，它将照常扩展 JavaPerson，但按照 Scala 的要求，它还会实现 ScalaObject 接口。并照例支持继承自 JavaPerson 的方法，因为 ScalaJavaPerson 是一种 Scala 类型，我们可以期望它支持 Any 引用的指派，根据 Scala 的规则：

清单 **14.** 使用 **ScalaJavaPerson**

```
                                // This is Scala
    val richard = new ScalaJavaPerson("Richard", "Campbell", 45)
    System.out.println(richard)
    val host : Any = richard
    System.out.println(host)
```

但在 Scala 中创建 JavaPerson 并将其指派给 Any 引用时会发生什么？

清单 **15.** 使用 **JavaPerson**

```

// This is Scala
    val carl = new JavaPerson("Carl", "Franklin", 35)
    System.out.println(carl)
```

```
val host2 : Any = carl
System.out.println(host2)
```

结果显示，这段代码如期编译并运行，因为 Scala 能确保 `JavaPerson` “做正确的事情”，这要归功于 `Any` 类型与 `java.lang.Object` 类型的相似性。实际上，几乎可以说，所有扩展 `java.lang.Object` 的内容都支持存储到 `Any` 引用之中。（存在一些极端情况，我听说过，但我自己还从未遇到过这样的极端情况。）

最终结果？出于实践的目的，我们可以跨 Java 语言和 Scala 混搭继承，而无需过分担心。（最大的麻烦将是试图了解如何覆盖“名字很有趣的 Scala 方法”，例如 `^=!#` 或类似方法。）

回页首

结束语

在本月的文章中，我为您介绍了 Scala 代码和 Java 代码之间的高度相似性意味着 Java 开发人员可以轻松理解并使用 Scala 的继承模型。方法覆盖的工作方式相同，成员可见性的工作方式相同，还有更多相同的地方。对于 Scala 中的所有功能，继承或许与 Java 开发中的对应部分最为相似。惟一需要技巧的部分就是 Scala 语法，这有着明显的差异。

习惯两种语言中继承方法的相似之处和细微的差异，您就可以轻松编写您自己的 Java 程序的 Scala 实现。例如，考虑流行的 Java 基类和框架的 Scala 实现，如 JUnit、Servlets、Swing 或 SWT。实际上，Scala 团队已经提供了一个 Swing 应用程序，名为 OOPScala（参见 参考资料），它使用 `JTable`，通过相当少的几行代码（数量级远远低于传统 Java 的对应实现）提供了简单的电子表格功能。

因此，如果您想知道如何在您的生产代码中应用 Scala，就应该准备好迈出探索的第一步。考虑在 Scala 中编写下一个程序的一小部分。正如您在这期文章中所了解到的那样，从恰当的基类继承，采用与 Java 程序中相同的方式提供覆盖，您就不会遇到任何麻烦。

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。
- 面向 *Java* 开发人员的 *Scala* 指南 （Ted Neward，IBM developerWorks，2008 年）：阅读本系列的所有文章。
- “Scala for Java refugees Part 5: Traits and types”（Daniel Spiewak，Code Commit，2008 年 2 月）：另外一份关于 Scala 内继承的主题的文章。
- “Implementation Inheritance with Mixins - Some Thoughts”（Debasish Ghosh，Ruminations of a Programmer，2008 年 2 月）：讨论定义 Java 语言继承的“可行折衷”。
- “A Tour of Scala: Higher-Order Functions”（Scala-lang.org）：一个简单的示例，介绍了 Scala 中的高阶函数。
- “OOPScala”：在 Scala 中编写的一个 Swing 应用程序。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和应用。
- “Scala by Example”（Martin Odersky，2008 年 5 月）：这篇简短的、代码驱动的 Scala 介绍文章，包括本文中使用的 Quicksort 应用程序（PDF 格式）。
- “Programming in Scala”（Martin Odersky、Lex Spoon 和 Bill Venners；上次刊印时间为 2008 年 2 月）：这是第一步介绍 Scala 的图书，由 Bill Venners 等合著。

- developerWorks Java 技术专区：数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scala：目前最新版本是 2.7.0-final。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 集合类型

在 **Scala** 使用元组、数组和列表

文档选项

打印本页

将此页作为电子邮件发送

样例代码

英文原文

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 8 月 15 日

在 Scala 中，对象占有一席之地，然而，也经常使用到一些函数类型，比如元组、数组和列表。在这一期由 Ted Neward 撰写的流行系列文章中，您将探究 Scala 中的函数部分，并且首先研究 Scala 对函数语言中常见类型的支持。

对于学习 Scala 的 Java™ 开发人员来说，对象是一个比较自然、简单的入口点。在 本系列 前几期文章中，我介绍了 Scala 中一些面向对象的编程方法，这些方法实际上与 Java 编程的区别不是很大。我还向您展示了 Scala 如何重新应用传统的面向对象概念，找到其缺点，并根据 21 世纪的新需求重新加以改造。Scala 一直隐藏的一些重要内容将要现身：Scala 也是一种函数语言（这里的函数性是与其他 *dys* 函数语言相对而言的）。

Scala 的面向函数性非常值得探讨，这不仅是因为已经研究完了对象内容。Scala 中的函数编程将提供一些新的设计结构和理念以及一些内置构造，它们使某些场景（例如并发性）的编程变得非常简单。

本月，您将首次进入 Scala 的函数编程领域，查看大多数函数语言中常见的四种类型：列表（list）、元组（tuple）、集合（set）和 `Option` 类型。您还将了解 Scala 的数组，后者对其他函数语言来说十分新鲜。

这些类型都提出了编写代码的新方式。当结合传统面向对象特性时，可以生成十分简洁的结果。

使用 Option(s)

在什么情况下，“无” 并不代表 “什么也没有”？当它为 0 的时候，与 null 有什么关系。

对于我们大多数人都非常熟悉的概念，要在软件中表示为 “无” 是一件十分困难的事。例如，看看 C++ 社区中围绕 `NULL` 和 0 进行的激烈讨论，或是 SQL 社区围绕 `NULL` 列值展开的争论，便可知晓一二。 `NULL` 或 null 对于大多数程序员来说都表示 “无”，但是这在 Java 语言中引出了一些特殊问题。

考虑一个简单操作，该操作可以从一些位于内存或磁盘的数据库查找程序员的薪资：API 允许调用者传入一个包含程序员名字的 `String`，这会返回什么呢？从建模角度来看，它应该返回一个 `Int`，表示程序员的年薪；但是这里有一个问题，如果程序员不在数据库中（可能根本没有雇用她，或者已经被解雇，要不就是输错了名字……），那么应该返回什么。如果返回类型是 `Int`，则不能返回 *null*，这个 “标志” 通常表示没有在数据库中找到该用户（您可能认为应该抛出一个异常，但是大多数时候数据库丢失值并不能视为异常，因此不应该在这里抛出异常）。

在 Java 代码中，我们最终将方法标记为返回 `java.lang.Integer`，这迫使调用者知道方法可以返回 *null*。自然，我们可以依靠程序员来全面归档这个场景，还可以依赖程序员读取精心准备的文档。这类似于：我们可以要求经理倾听我们反对他们要求的不可能完成的项目期限，然后经理再进一步把我们的反对传达给上司和用户。

C# 2.0 可变为 null 值的类型

其他语言已试图通过各种方法解决 “可 null 值化” 问题：C++ 一直都忽略了这个问题，直至最后确定 null 和 0 是不同的值。Java 语言仍然没有彻底解决这个问题，而是依赖于自动装箱（autobox）— 将原语类型自动转换为它们的包装器对象（在 1.1 以后引入）— 帮助 Java 程序员解决问题。一些模式爱好者建议每种类型都应该有一个对应的 “Null Object”，即将自己的所有方法重写为不执行任何操作的类型（实际上是子类型）的实例 — 实践证明这需要大量工作。C# 1.0 发布后，C# 设计者决定采取一种完全不同的方法解决 null 值化问题。

C# 2.0 引入了可变为 *null* 值的类型 的概念，重要的是添加了语法支持，认为任何特定值类型（基本指原语类型）都可以通过将 *null* 封装到一个泛型/模板类 `Nullable<T>`，从而提供 *null* 支持。`Nullable<T>` 本身是在类型声明中通过 `?` 修饰符号引入。因此，`int?` 表示一个整数也可能为 null。

表面上看，这似乎很合理，但是事情很快就变得复杂起来。`int` 和 `int?` 是否应该被视为可兼容类型，如果是的话，什么时候将 `int` 提升为 `int?`，反之呢？当将

Scala 提供了一种普通的函数方法，打破了这一僵局。在某些方面，Option 类型或 Option[T]，并不重视描述。它是一个具有两个子类 Some[T] 和 None 的泛型类，用来表示“无值”的可能性，而不需要语言类型系统大费周折地支持这个概念。实际上，使用 Option[T] 类型可以使问题更加清晰（下一节将用到）。

在使用 Option[T] 时，关键的一点是认识到它实质上是一个大小为“1”的强类型集合，使用一个不同的值 *None* 表示“nothing”值的可能性。因此，在这里方法没有返回 null 表示没有找到数据，而是进行声明以返回 Option[T]，其中 T 是返回的原始类型。那么，对于没有查找到数据的场景，只需返回 *None*，如下所示：

清单 **1.** 准备好踢足球了吗？

```
@Test def simpleOptionTest =
{
  val footballTeamsAFCEast =
    Map("New England" -> "Patriots",
        "New York" -> "Jets",
        "Buffalo" -> "Bills",
        "Miami" -> "Dolphins",
        "Los Angeles" -> null)

  assertEquals(footballTeamsAFCEast.get("Miami"), Some("Dolphins"))
  assertEquals(footballTeamsAFCEast.get("Miami").get(), "Dolphins")
  assertEquals(footballTeamsAFCEast.get("Los Angeles"), Some(null))
  assertEquals(footballTeamsAFCEast.get("Sacramento"), None)
}
```

注意，Scala Map 中 get 的返回值实际上并不对应于传递的键。相反，它是一个 Option[T] 实例，可以是与某个值有关的 Some()，也可以是 None，因此可以很清晰地表示没有在 map 中找到键。如果它可以表示 map 上存在某个键，但是有对应的 null 值，这一点特别重要了。比如清单 1 中 *Los Angeles* 键。

通常，当处理 Option[T] 时，程序员将使用模式匹配，这是一个非常函数化的概念，它允许有效地“启用”类型和/或值，更不用说在定义中将值绑定到变量、在 Some() 和 None 之间切换，以及提取 Some 的值（而不需要调用麻烦的 get() 方法）。清单 2 展示了 Scala 的模式匹配：

清单 **2.** 巧妙的模式匹配

```
@Test def optionWithPM =
{
  val footballTeamsAFCEast =
    Map("New England" -> "Patriots",
        "New York" -> "Jets",
        "Buffalo" -> "Bills",
        "Miami" -> "Dolphins")

  def show(value : Option[String]) =
  {
    value match
    {
      case Some(x) => x
```

int 添加到 int? 会发生什么，结果会是 null 吗？这类问题等等。随后类型系统进行了一些重要的调整，可变为 null 值的类型随后包含到了 2.0 中 — 而 C# 程序员几乎完全忽略了它们。

回顾一下 Option 类型的函数方法，它使 Option[T] 和 Int 之间的界限变得很清晰，看上去要比其他方法更加简单。在那些围绕可变为 null 值类型的反直觉（counterintuitive）提升规则之间进行比较时，尤其如此。（函数领域对该问题近二十年的思考是值得的）。要使用 Option[T] 必须付出一些努力，但是总的来说，它产生了更清晰的代码和期望。

```
        case None => "No team found"
      }
    }

    assertEquals(show(footballTeamsAFCEast.get("Miami")), "Dolphins")
  }
```

回页首

元组和集合

在 C++ 中，我们将之称为结构体。在 Java 编程中，我们称之为数据传输对象或参数对象。在 Scala 中，我们称为元组。实质上，它们是一些将其他数据类型收集到单个实例的类，并且不使用封装或抽象 — 实际上，不 使用任何抽象常常更有用。

在 Scala 创建一个元组类型非常的简单，这只是主体的一部分：如果首先将元素公开给外部，那么在类型内部创建描述这些元素的名称就毫无价值。考虑清单 3：

清单 **3. tuples.scala**

```
// JUnit test suite
//
class TupleTest
{
  import org.junit._, Assert._
  import java.util.Date

  @Test def simpleTuples() =
  {
    val tedsStartingDateWithScala = Date.parse("3/7/2006")

    val tuple = ("Ted", "Scala", tedsStartingDateWithScala)

    assertEquals(tuple._1, "Ted")
    assertEquals(tuple._2, "Scala")
    assertEquals(tuple._3, tedsStartingDateWithScala)
  }
}
```

创建元组非常简单，将值放入一组圆括号内，就好象调用一个方法调用一样。提取这些值只需要调用 "`_n`" 方法，其中 *n* 表示相关的元组元素的位置参数：`_1` 表示第一位，`_2` 表示第二位，依此类推。传统的 Java `java.util.Map` 实质上是一个分两部分的元组集合。

元组可以轻松地实现使用单个实体移动多个值，这意味着元组可以提供在 Java 编程中非常重量级的操作：多个返回值。例如，某个方法可以计算 `String` 中字符的数量，并返回该 `String` 中出现次数最多的字符，但是如果程序员希望同时 返回最常出现的字符和 它出现的次数，那么程序设计就有点复杂了：或是创建一个包含字符及其出现次数的显式类，或将值作为字段保存到对象中并在需要时返回字段值。无论使用哪种方法，与使用 Scala 相比，都需要编写大量代码；通过简单地返回包含字符及其出现次数的元组，Scala 不仅可以轻松地使用 "`_1`"、"`_2`" 等访问元组的各个值，还可以轻松地返回多个返回值。

如下节所示，Scala 频繁地将 `Option` 和元组保存到集合（例如 `Array[T]` 或列表）中，从而通过一个比较简单的结构提供了极大的灵活性和威力。

数组带您走出阴霾

让我们重新审视一个老朋友 — 数组 — 在 Scala 中是 `Array[T]`。和 Java 代码中的数组一样，Scala 的 `Array[T]` 是一组有序的元素序列，使用表示数组位置的数值进行索引，并且该值不可以超过数组的总大小，如清单 4 所示：

清单 **4. array.scala**

```
object ArrayExample1
{
  def main(args : Array[String]) : Unit =
  {
    for (i <- 0 to args.length-1)
    {
      System.out.println(args(i))
    }
  }
}
```

尽管等同于 Java 代码中的数组（毕竟后者是最终的编译结果），Scala 中的数组使用了截然不同的定义。对于新手，Scala 中的数组实际上就是泛型类，没有增加“内置”状态（至少，不会比 Scala 库附带的其他类多）。例如，在 Scala 中，数组一般定义为 `Array[T]` 的实例，这个类定义了一些额外的有趣方法，包括常见的“length”方法，它将返回数组的长度。因此，在 Scala 中，可以按照传统意义使用 `Array`，例如使用 `Int` 在 0 到 `args.length - 1` 间进行迭代，并获取数组的第 *i* 个元素（使用圆括号而不是方括号来指定返回哪个元素，这是另一种名称比较有趣的方法）。

扩展数组

事实证明 `Array` 拥有大量方法，这些方法继承自一个非常庞大的 parent 层次结构：`Array` 扩展 `Array0`，后者扩展 `ArrayLike[A]`，`ArrayLike[A]` 扩展 `Mutable[A]`，`Mutable[A]` 又扩展 `RandomAccessSeq[A]`，`RandomAccessSeq[A]` 扩展了 `Seq[A]`，等等。实际上，这种层次结构意味着 `Array` 可以执行很多操作，因此与 Java 编程相比，在 Scala 中可以更轻松地使用数组。

> 提示
>
> 请查看 Scaladocs 以了解完整的 `Array[T]` 内容。它在很多方面都让人联想到了 `java.util Collections` 类。

例如，如清单 4 所示，使用 `foreach` 方法遍历数组更加简单并且更贴近函数的方式，这些都继承自 `Iterable` 特性：

清单 **5. ArrayExample2**

```
object
{
  def main(args : Array[String]) : Unit =
  {
    args.foreach( (arg) => System.out.println(arg) )
  }
}
```

看上去您没有节省多少工作，但是，将一个函数（匿名或其他）传入到另一个类中以便获得在特定语义下（在本例中指遍历数组）执行的能力，是函数编程的常见主题。以这种方式使用更高阶函数并不局限于迭代；事实上，还得经常对数组内容执行一些过滤 操作去掉无用的内容，然后再处理结果。例如，在 Scala 中，可以轻松地使用 `filter` 方法进行过滤，然后获取结果列表并使用 `map` 和另一个函数（类型为 `(T) => U`，其中 T 和 U 都是泛型类型），或 `foreach` 来处理每个元素。我在清单 6 中采取了后一种方法（注意 `filter` 使用了一个 `(T) : Boolean` 方法，意味着使用数组持有的任意类型的参数，

并返回一个 Boolean）。

清单 **6.** 查找所有 **Scala** 程序员

```
class ArrayTest
{
  import org.junit._, Assert._

  @Test def testFilter =
  {
    val programmers = Array(
        new Person("Ted", "Neward", 37, 50000,
          Array("C++", "Java", "Scala", "Groovy", "C#", "F#", "Ruby")),
        new Person("Amanda", "Laucher", 27, 45000,
          Array("C#", "F#", "Java", "Scala")),
        new Person("Luke", "Hoban", 32, 45000,
          Array("C#", "Visual Basic", "F#")),
              new Person("Scott", "Davis", 40, 50000,
                Array("Java", "Groovy"))
      )

    // Find all the Scala programmers ...
    val scalaProgs =
      programmers.filter((p) => p.skills.contains("Scala") )

    // Should only be 2
    assertEquals(2, scalaProgs.length)

    // ... now perform an operation on each programmer in the resulting
    // array of Scala programmers (give them a raise, of course!)
    //
    scalaProgs.foreach((p) => p.salary += 5000)

    // Should each be increased by 5000 ...
    assertEquals(programmers(0).salary, 50000 + 5000)
    assertEquals(programmers(1).salary, 45000 + 5000)

    // ... except for our programmers who don't know Scala
    assertEquals(programmers(2).salary, 45000)
        assertEquals(programmers(3).salary, 50000)
  }
}
```

创建一个新的 Array 时将用到 map 函数，保持原始的数组内容不变，实际上大多数函数性程序员都喜欢这种方式：

清单 **7. Filter** 和 **map**

```
  @Test def testFilterAndMap =
  {
```

```
    val programmers = Array(
        new Person("Ted", "Neward", 37, 50000,
          Array("C++", "Java", "Scala", "C#", "F#", "Ruby")),
        new Person("Amanda", "Laucher", 27, 45000,
          Array("C#", "F#", "Java", "Scala")),
        new Person("Luke", "Hoban", 32, 45000,
          Array("C#", "Visual Basic", "F#"))
              new Person("Scott", "Davis", 40, 50000,
                Array("Java", "Groovy"))
      )

    // Find all the Scala programmers ...
    val scalaProgs =
      programmers.filter((p) => p.skills.contains("Scala") )

    // Should only be 2
    assertEquals(2, scalaProgs.length)

    // ... now perform an operation on each programmer in the resulting
    // array of Scala programmers (give them a raise, of course!)
    //
    def raiseTheScalaProgrammer(p : Person) =
    {
      new Person(p.firstName, p.lastName, p.age,
        p.salary + 5000, p.skills)
    }
    val raisedScalaProgs =
      scalaProgs.map(raiseTheScalaProgrammer)

    assertEquals(2, raisedScalaProgs.length)
    assertEquals(50000 + 5000, raisedScalaProgs(0).salary)
    assertEquals(45000 + 5000, raisedScalaProgs(1).salary)
  }
```

注意，在清单 7 中，`Person` 的 `salary` 成员可以标记为“`val`”，表示不可修改，而不是像上文一样为了修改不同程序员的薪资而标记为“`var`”。

Scala 的 `Array` 提供了很多方法，在这里无法一一列出并演示。总的来说，在使用数组时，应该充分地利用 `Array` 提供的方法，而不是使用传统的 `for ...` 模式遍历数组并查找或执行需要的操作。最简单的实现方法通常是编写一个函数（如果有必要的话可以使用嵌套，如清单 7 中的 `testFilterAndMap` 示例所示），这个函数可以执行所需的操作，然后根据期望的结果将该函数传递给 `Array` 中的 `map`、`filter`、`foreach` 或其他方法之一。

回页首

函数性列表

函数编程多年来的一个核心特性就是列表，它和数组在对象领域中享有相同级别的“内置”性。列表对于构建函数性软件非常关键，因此，您（作为一名刚起步的 Scala 程序员）必须能够理解列表及其工作原理。即使列表从未形成新的设计，但是 Scala 代码在其库中广泛使用了列表。因此学习列表是非常必要的。

在 Scala 中，列表类似于数组，因为它的核心定义是 Scala 库中的标准类 `List[T]`。并且，和 `Array[T]` 相同，`List[T]` 继承了很多基类和特性，首先使用 `Seq[T]` 作为直接上层基类。

基本上，列表是一些可以通过列表头或列表尾提取的元素的集合。列表来自于 Lisp，后者是一种主要围绕 “LISt 处理” 的语言，它通过 car 操作获得列表的头部，通过 cdr 操作获得列表尾部（名称渊源与历史有关；第一个可以解释它的人有奖励）。

从很多方面来讲，使用列表要比使用数组简单，原因有二，首先函数语言过去一直为列表处理提供了良好的支持（而 Scala 继承了这些支持），其次可以很好地构成和分解列表。例如，函数通常从列表中挑选内容。为此，它将选取列表的第一个元素 — 列表头部 — 来对该元素执行处理，然后再递归式地将列表的其余部分传递给自身。这样可以极大减少处理代码内部具有相同共享状态的可能性，并且，假如每个步骤只需处理一个元素，极有可能使代码分布到多个线程（如果处理是比较好的）。

构成和分解列表非常简单，如清单 8 所示：

清单 **8.** 使用列表

```
class ListTest
{
  import org.junit._, Assert._

  @Test def simpleList =
  {
    val myFirstList = List("Ted", "Amanda", "Luke")

    assertEquals(myFirstList.isEmpty, false)
    assertEquals(myFirstList.head, "Ted")
    assertEquals(myFirstList.tail, List("Amanda", "Luke")
    assertEquals(myFirstList.last, "Luke")
  }
}
```

注意，构建列表与构建数组十分相似；都类似于构建一个普通对象，不同之处是这里不需要 “new”（这是 “case 类” 的功能，我们将在未来的文章中介绍到）。请进一步注意 tail 方法调用的结果 — 结果并不是列表的最后一个元素（通过 last 提供），而是除第一个元素以外的其余列表元素。

当然，列表的强大力量部分来自于递归处理列表元素的能力，这表示可以从列表提取头部，直到列表为空，然后累积结果：

清单 **9.** 递归处理

```
  @Test def recurseList =
  {
    val myVIPList = List("Ted", "Amanda", "Luke", "Don", "Martin")

    def count(VIPs : List[String]) : Int =
    {
      if (VIPs.isEmpty)
        0
      else
        count(VIPs.tail) + 1
    }

    assertEquals(count(myVIPList), myVIPList.length)
  }
```

注意，如果不考虑返回类型 `count`，Scala 编译器或解释器将会出现点麻烦 — 因为这是一个尾递归（tail-recursive）调用，旨在减少在大量递归操作中创建的栈帧的数量，因此需要指定它的返回类型。即使是这样，也可以轻松地使用 `List` 的 "length" 成员获取列表项的数量，但关键是如何解释列表处理强大的功能。清单 9 中的整个方法完全是线程安全的，因为列表处理中使用的整个中间状态保存在参数的堆栈上。因此，根据定义，它不能被多个线程访问。函数性方法的一个优点就是它实际上与程序功能截然不同，并且仍然创建共享的状态。

列表 API

列表具有另外一些有趣的特性，例如构建列表的替代方法，使用 `::` 方法（是的，这是一种方法。只不过名称比较有趣）。因此，不必使用 "`List`" 构造函数语法构建列表，而是将它们 "拼接" 在一起（在调用 `::` 方法时），如清单 10 所示：

清单 **10.** 是 **:: == C++** 吗？

```
  @Test def recurseConsedList =
  {
    val myVIPList = "Ted" :: "Amanda" :: "Luke" :: "Don" :: "Martin" :: Nil

    def count(VIPs : List[String]) : Int =
    {
      if (VIPs.isEmpty)
        0
      else
        count(VIPs.tail) + 1
    }

    assertEquals(count(myVIPList), myVIPList.length)
  }
```

在使用 `::` 方法时要小心 — 它引入了一些很有趣的规则。它的语法在函数语言中非常常见，因此 Scala 的创建者选择支持这种语法，但是要正确、普遍地使用这种语法，必须使用一种比较古怪的规则：任何以冒号结束的 "名称古怪的方法" 都是右关联（*right-associative*）的，这表示整个表达式从它的最右边的 `Nil` 开始，它正好是一个 `List`。因此，可以将 `::` 认定为一个全局的 `::` 方法，与 `String` 的一个成员方法（本例中使用）相对；这又表示您可以对所有内容构建列表。在使用 `::` 时，最右边的元素必须是一个列表，否则将得到一个错误消息。

在 Scala 中，列表的一种最强大的用法是与模式匹配结合。由于列表不仅可以匹配类型和值，它还可以同时绑定变量。例如，我可以简化清单 10 的列表代码，方法是使用模式匹配区别一个至少具有一个元素的列表和一个空列表：

清单 **11.** 结合使用模式匹配和列表

```
  @Test def recurseWithPM =
  {
    val myVIPList = "Ted" :: "Amanda"
:: "Luke" :: "Don" :: "Martin" :: Nil

    def count(VIPs : List[String]) :
Int =
    {
      VIPs match
      {
        case h :: t => count(t) + 1
        case Nil => 0
```

什么是右关联？

要更好地理解 `::` 方法，要记住 "冒号" 这类操作符仅仅是一些名称比较有趣的方法。对于普通的左管理语法，左侧的标记一般是我将要对其调用方法名（右侧的标记）的对象。因此，通常来说，表达式 `1 + 2` 在编译器看来等同于 `1.+(2)`。

但是对于列表而言，这些都不适合 — 系统中的每个类都需要对系统中的所有类型使用 `::` 方法，而这严重违背了关注点分离原则。

Scala 的修复方法是：以冒号结束的任何具有奇怪名称的方法（例如 `::` 或 `:::`，甚至是我自己创建的方法，比如 `foo:`）都是右关联的。因此，比方说，`a :: b :: c :: Nil` 转换为

```
    }
  }

  assertEquals(count(myVIPList),
myVIPList.length)
 }
```

> `Nil.::(c.::(b.::(a)))`，后者正是我需要的：`List` 在首位，这样每次调用 `::` 都可以获取对象参数并返回一个 `List`，并继续执行下去。
>
> 最好为其他命名约定指定右关联属性，但是在撰写本文之际，Scala 已将这条规则硬编码到该语言中。就目前来说，冒号是惟一触发右关联行为的字符。

在第一个 `case` 表达式中，将提取列表头部并绑定到变量 *h*，而其余部分（尾部）则绑定到 *t*；在本例中，没有对 *h* 执行任何操作（实际上，更好的方法是指明这个头部永远不会被使用，方法是使用一个通配符 _ 代替 *h*，这表明它是永远不会使用到的变量的占位符）。但是 *t* 被递归地传递给 `count`，和前面的示例一样。还要注意，Scala 中的每一个表达式将隐式返回一个值；在本例中，模式匹配表达式的结果是递归调用 `count + 1`，当达到列表结尾时，结果为 `0`。

考虑到相同的代码量，使用模式匹配的价值体现在哪里？实际上，对于比较简单的代码，模式匹配的价值不很明显。但是对于稍微复杂的代码，例如扩展示例以匹配特定值，那么模式匹配非常有帮助。

清单 **12.** 模式匹配

```
@Test def recurseWithPMAndSayHi =
{
  val myVIPList = "Ted" :: "Amanda" :: "Luke" :: "Don" :: "Martin" :: Nil

  var foundAmanda = false
  def count(VIPs : List[String]) : Int =
  {
    VIPs match
    {
      case "Amanda" :: t =>
        System.out.println("Hey, Amanda!"); foundAmanda = true; count(t) + 1
      case h :: t =>
        count(t) + 1
      case Nil =>
        0
    }
  }

  assertEquals(count(myVIPList), myVIPList.length)
  assertTrue(foundAmanda)
}
```

示例很快会变得非常复杂，特别是正则表达式或 XML 节点，开始大量使用模式匹配方法。模式匹配的使用同样不局限于列表；我们没有理由不把它扩展到前面的数组示例中。事实上，以下是前面的 `recurseWithPMAndSayHi` 测试的数组示例：

清单 **13.** 将模式匹配扩展到数组

```
@Test def recurseWithPMAndSayHi =
{
  val myVIPList = Array("Ted", "Amanda", "Luke", "Don", "Martin")

  var foundAmanda = false
```

```
    myVIPList.foreach((s) =>
      s match
      {
        case "Amanda" =>
          System.out.println("Hey, Amanda!")
          foundAmanda = true
        case _ =>
          ; // Do nothing
      }
    )

    assertTrue(foundAmanda)
  }
```

如果希望进行实践，那么尝试构建清单 13 的递归版本，但这不用在 `recurseWithPMAndSayHi` 范围内声明一个可修改的 `var`。提示：需要使用多个模式匹配代码块（本文的 代码下载 中包含了一个解决方案 — 但是建议您在查看之前首先自己进行尝试）。

回页首

结束语

Scala 是丰富的集合的集合（双关语），这源于它的函数历史和特性集；元组提供了一种简单的方法，可以很容易地收集松散绑定的值集合；`Option[T]` 可以使用简单的方式表示和 “no” 值相对的 “some” 值；数组可以通过增强的特性访问传统的 Java 式的数组语义；而列表是函数语言的主要集合，等等。

然而，需要特别注意其中一些特性，特别是元组：学会使用元组很容易，并且会因为为了直接使用元组而忘记传统的基本对象建模。如果某个特殊元组 — 例如，名称、年龄、薪资和已知的编程语言列表 — 经常出现在代码库中，那么将它建模为正常的类类型和对象。

Scala 的优点是它兼具函数性和 面向对象特性，因此，您可以在享受 Scala 的函数类型的同时，继续像以前一样关注类设计。

关于本系列

和 Ted Neward 一起研究 Scala 编程语言。在这个新的 developerWorks 系列 中，您将深入了解 Scala，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！毕竟，如果用 Java 代码就可以实现的话，又何必学习 Scala 呢？

回页首

下载

描述	名字	大小	下载方法
本文的样例 Scala 代码	j-scala06278.zip	200KB	**HTTP**

关于下载方法的信息

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。

- “面向 Java 开发人员的 Scala 指南：面向对象的函数编程”（Ted Neward，developerWorks，2008 年 1 月）：本系列的第一篇文章提供了 Scala 的概述并解释了用于实现并发性的函数方法。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的观点解释函数性编程的优点和用法。
- “COBOL 式死亡”（Ted Neward，developerWorks，2008 年 5 月）：现在是否应该放弃 Java 平台而转向更新的技术？Ted 介绍了针对 Java 存亡的相关讨论。
- Ted Neward on why we need Scala（JavaWorld，2008 年 6 月）：在与 Andrew Glover 的 podcast 讨论中，Ted 讲述了更多函数编程知识并介绍了 Scala 在 Java 生态系统的地位。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：一个简短的代码驱动的 Scala 介绍（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners; Artima，2007 年 12 月）：第一份 Scala 介绍，篇幅和一本书差不多。
- Scaladocs：Scala Library 的 API 规范。
- developerWorks Java 技术专区：数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scala：开始学习本系列！

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

◉太差！ (1)
○需提高 (2)
○一般；尚可 (3)
○好文章 (4)
○真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 包和访问修饰符

Scala 中的 **public**、**private** 以及其他成员

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 9 月 16 日

在现实生活中，代码一定要引用并打包，在本期（第七期） 面向 *Java* 开发人员的 *Scala* 指南 系列中，Ted Neward 介绍了 Scala 的包和访问修饰符功能，纠正了以前的疏忽。然后，他继续探讨了 Scale 中的函数内容：“apply” 机制。

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- 打包
- 导入
- 访问
- 应用
- 结束语
- 下载
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

文档选项

- 打印本页
- 将此页作为电子邮件发送
- 样例代码
- 英文原文

最近，读者的反馈让我意识到在制作本系列的过程中我遗漏了 Scala 的语言的一个重要方面：Scala 的包和访问修饰符功能。所以在研究该语言的函数性元素 `apply` 机制前，我将先介绍包和访问修饰符。

打包

为了有助于隔离代码，使其不会相互冲突，Java™ 代码提供了 `package` 关键词，由此创建了一个词法命名空间，用以声明类。本质上，将类 `Foo` 放置到名为 com.tedneward.util 包中就将正式类名修改成了 `com.tedneward.util.Foo`；同理，必须按该方法引用类。如果没有，Java 编程人员会很快指出，他们会 `import` 该包，避免键入正式名的麻烦。的确如此，但这仅意味着根据正式名引用类的工作由编译器和字节码完成。快速浏览一下 `javap` 的输出，这点就会很明了。

然而，Java 语言中的包还有几个特殊的要求：一定要在包所作用的类所在的 .java 文件的顶端声明包（在将注释应用于包时，这一点会引发很严重的语言问题）；该声明的作用域为整个文件。这意味着两个跨包进行紧密耦合的类一定要在跨文件时分离，这会致使两者间的紧密耦合很容易被忽略。

Scala 在打包方面所采取的方法有些不同，它结合使用了 Java 语言的 `declaration` 方法和 C# 的 `scope`（限定作用域）方法。了解了这一点，Java 开发人员就可以使用传统的 Java 方法并将 `package` 声明放在 .scala 文件的顶部，就像普通的 Java 类一样；包声明的作用域为整个文件，就像在 Java 代码中一样。而 Scala 开发人员则可以使用 Scala 的包“（scoping）限定作用域”方法，用大括号限制 `package` 语句的作用域，如清单 1 所示：

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个新的 developerWorks 系列 中， 您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

清单 **1.** 简化的打包

```
package com
{
  package tedneward
  {
    package scala
    {
      package demonstration
      {
        object App
```

```
        {
          def main(args : Array[String]) : Unit =
          {
            System.out.println("Howdy, from packaged code!")
            args.foreach((i) => System.out.println("Got " + i) )
          }
        }
      }
    }
  }
}
```

这个代码有效地声明了类 App，或者更确切的说是一个称为 com.tedneward.scala.demonstration.App 的单个类。注意 Scala 还允许用点分隔包名，所以清单 1 中的代码可以更简洁，如清单 2 所示:

清单 **2.** 简化了的打包（**redux**）

```
package com.tedneward.scala.demonstration
{
  object App
  {
      def main(args : Array[String]) : Unit =
      {
        System.out.println("Howdy, from packaged code!")
        args.foreach((i) => System.out.println("Got " + i) )
      }
  }
}
```

用哪一种样式看起来都比较合适，因为它们都编译出一样的代码构造（Scala 将继续编译并和 javac 一样在声明包的子目录中生成 .class 文件）。

导入

与包相对的当然就是 import 了，Scala 使用它将名称放入当前词法名称空间。本系列的读者已经在此前的很多例子中见到过 import 了，但现在我将指出一些让 Java 开发人员大吃一惊的 import 的特性。

首先，import 可以用于客户机 Scala 文件内的任何地方，并非只可以用在文件的顶部，这样就有了作用域的关联性。因此，在清单 3 中，java.math.BigInteger 导入的作用域被完全限定到了在 App 对象内部定义的方法，其他地方都不行。如果 mathfun 内的其他类或对象要想使用 java.math.BigInteger，就需要像 App 一样导入该类。如果 mathfun 的几个类都想使用 java.math.BigInteger，可以在 App 的定义以外的包级别导入该类，这样在包作用域内的所有类就都导入 BigInteger 了。

清单 **3.** 导入的作用域

```
package com
{
  package tedneward
  {
    package scala
    {
```

```
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger

          def factorial(arg : BigInteger) : BigInteger =
          {
            if (arg == BigInteger.ZERO) BigInteger.ONE
            else arg multiply (factorial (arg subtract BigInteger.ONE))
          }

          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

不只如此，Scala 还不区分高层成员和嵌套成员，所以您不仅可以使用 `import` 将嵌套类型的成员置于词法作用域中，其他任何成员均可；例如，您可以通过导入 `java.math.BigInteger` 内的所有名称，使对 ZERO 和 ONE 的限定了作用域的引用缩小为清单 4 中的名称引用：

清单 **4.** 静态导入

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger
          import BigInteger._

          def factorial(arg : BigInteger) : BigInteger =
          {
```

```
            if (arg == ZERO) ONE
            else arg multiply (factorial (arg subtract ONE))
          }

          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

您可以使用下划线（还记得 Scala 中的通配符吧？）有效地告知 Scala 编译器 `BigInteger` 内的所有成员都需要置入作用域。由于 `BigInteger` 已经被先前的导入语句导入到作用域中，因此无需显式地使用包名限定类名。实际上，可以将所有这些都结合到一个语句中，因为 `import` 可以同时导入多个目标，目标间用逗号隔开（如清单 5 所示）：

清单 **5.** 批量导入

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object App
        {
          import java.math.BigInteger, BigInteger._

          def factorial(arg : BigInteger) : BigInteger =
          {
            if (arg == ZERO) ONE
            else arg multiply (factorial (arg subtract ONE))
          }

          def main(args : Array[String]) : Unit =
          {
            if (args.length > 0)
              System.out.println("factorial " + args(0) +
                " = " + factorial(new BigInteger(args(0))))
            else
```

```
              System.out.println("factorial 0 = 1")
          }
        }
      }
    }
  }
}
```

这样您可以节省一两行代码。注意这两个导入过程不能结合：先导入 `BigInteger` 类本身，再导入该类中的各种成员。

也可以使用 `import` 来引入其他非常量的成员。例如，考虑一下清单 6 中的数学工具库（或许不一定有什么价值）：

清单 **6. Enron** 的记帐代码

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object BizarroMath
        {
          def bizplus(a : Int, b : Int) = { a - b }
          def bizminus(a : Int, b : Int) = { a + b }
          def bizmultiply(a : Int, b : Int) = { a / b }
          def bizdivide(a : Int, b : Int) = { a * b }
        }
      }
    }
  }
}
```

使用这个库会越来越觉得麻烦，因为每请求它的一个成员，都需要键入 `BizarroMath`，但是 Scala 允许将 `BizarroMath` 的每一个成员导入最高层的词法空间，因此简直就可以把它们当成全局函数来使用（如清单 7所示）：

清单 **7.** 计算 **Enron**的开支

```
package com
{
  package tedneward
  {
    package scala
    {
      package demonstration
```

```
      {
        object App2
        {
          def main(args : Array[String]) : Unit =
          {
            import com.tedneward.scala.mathfun.BizarroMath._

            System.out.println("2 + 2 = " + bizplus(2,2))
          }
        }
      }
    }
  }
}
```

还有其他的一些构造很有趣，它们允许 Scala 开发人员写出更自然的 `2 bizplus 2`，但是这些内容本文不予讨论（想了解 Scala 潜在的可以用于其他用途的特性的读者可以看一下 Odersky、Spoon 和 Venners 所著的 *Programming in Scala* 中谈到的 Scala `implicit` 构造）。

访问

打包（和导入）是 Scala 封装的一部分，和在 Java 代码中一样，在 Scala 中，打包很大一部分在于以选择性方式限定访问特定成员的能力 — 换句话说，在于 Scala 将特定成员标记为“公有（public）”、“private（私有）”或介于两者之间的成员的能力。

Java 语言有四个级别的访问：公有（public）、私有（private）、受保护的（protected ）和包级别（它没有任何关键词）访问。Scala：

- 废除了包级别的限制（在某种程度上）
- 默认使用“公有”
- 指定“私有”表示“只有此作用域可访问”

相反，Scala 定义“protected”的方式与在 Java 代码中不同；Java protected 成员对于子类和在其中定义成员的包来说是可访问的，Scala 中则仅有子类可访问。这意味着 Scala 版本的 protected 限制性要比 Java 版本更严格（虽然按理说更加直观）。

然而，Scala 真正区别于 Java 代码的地方是 Scala 中的访问修饰符可以用包名来“限定”，用以表明直到 哪个访问级别才可以访问成员。例如，如果 `BizarroMath` 包要将成员访问权限授权给同一包中的其他成员（但不包括子类），可以用清单 8 中的代码来实现：

清单 **8. Enron** 的记帐代码

```
package com
{
  package tedneward
  {
    package scala
    {
        // ...

      package mathfun
      {
        object BizarroMath
        {
          def bizplus(a : Int, b : Int) = { a - b }
          def bizminus(a : Int, b : Int) = { a + b }
          def bizmultiply(a : Int, b : Int) = { a / b }
```

```
          def bizdivide(a : Int, b : Int) = { a * b }

              private[mathfun] def bizexp(a : Int, b: Int) = 0
        }
      }
    }
  }
}
```

注意此处的 `private[mathfun]` 表达。本质上，这里的访问修饰符是说该成员直到 包 `mathfun` 为止都是私有的；这意味着包 `mathfun` 的任何成员都有权访问 `bizexp`，但任何包以外的成员都无权访问它，包括子类。

这一点的强大意义就在于任何包都可以使用 "private" 或者 "protected" 声明甚至 `com`（乃至 `_root_`，它是根名称空间的别名，因此本质上 `private[_root_]` 等效于 "public" 同）进行声明。这使得 Scala 能够为访问规范提供一定程度的灵活性，远远高于 Java 语言所提供的灵活性。

实际上，Scala 提供了一个更高程度的访问规范：对象私有 规范，用 `private[this]` 表示，它规定只有被同一对象调用的成员可以访问有关成员，其他对象里的成员都不可以，即使对象的类型相同（这弥合了 Java 访问规范系统中的一个缺口，这个缺口除对 Java 编程问题有用外，别无他用。）

注意访问修饰符必须在某种程度上在 JVM 之上映射，这致使定义中的细枝末节会在从正规 Java 代码中调用或编译时丢失。例如，上面的 BizarroMath 示例（用 `private[mathfun]` 声明的成员 `bizexp`）将会生成清单 9 中的类定义（当用 javap 来查看时）：

Listing 9. Enron 的记帐库，**JVM** 视图

```
Compiled from "packaging.scala"
public final class com.tedneward.scala.mathfun.BizarroMath
   extends java.lang.Object
{
    public static final int $tag();
    public static final int bizexp(int, int);
    public static final int bizdivide(int, int);
    public static final int bizmultiply(int, int);
    public static final int bizminus(int, int);
    public static final int bizplus(int, int);
}
```

在编译的 `BizarroMath` 类的第二行很容易看出，`bizexp()` 方法被赋予了 JVM 级别的 `public` 访问修饰符，这意味着一旦 Scala 编译器结束访问检查，细微的 `private[mathfun]` 区别就会丢失。因此，对于那些要从 Java 代码使用的 Scala 代码，我宁愿坚持传统的 "private" 和 "public" 的定义（甚至 "protected" 的定义有时最终映射到 JVM 级别的 "public"，所有不确定的时候，请对照实际编译的字节码参考一下 javap，以确认其访问级别。）

应用

在本系列上一期的文章中（"集合类型"），当谈及 Scala 中的数组时（确切地说是 `Array[T]`）我说过："获取数组的第 i 个元素" 实际上是 "那些名称很有趣的方法中的一种……"。尽管当时是因为我不想深入细节，但不管怎么说事实证明这种说法严格来说 是不对的。

好吧，我承认，我说谎了。

技术上讲，在 `Array[T]` 类上使用圆括号要比使用 " 名称有趣的方法" 复杂一点；Scala 为特殊的字符序列（即那些有左右括号的序列）保留了一个特殊名称关联，因为它有着特殊的使用意图："做"……（或按函数来说，将……"应用" 到……）。

换句话说，Scala 有一个特殊的语法（更确切一些，是一个特殊的语法关系）来代替 " 应用" 操作符 "()"。更精确地说，当用 `()` 作为方法调用来调用所述对象时，Scala 将称为 `apply()` 的方法作为调用的方法。例如，一个想充当仿函数（*functor*）的类（一个充当函数的对象）可以定义一个 `apply` 方法来提供类似于函数或方法的语义：

清单 **10.** 使用 **Functor**!

```
class ApplyTest
{
  import org.junit._, Assert._

  @Test def simpleApply =
  {
    class Functor
    {
      def apply() : String =
      {
        "Doing something without arguments"
      }

      def apply(i : Int) : String =
      {
        if (i == 0)
          "Done"
        else
          "Applying... " + apply(i - 1)
      }
    }

    val f = new Functor
    assertEquals("Doing something without arguments", f() )
    assertEquals("Applying... Applying... Applying... Done", f(3))
  }
}
```

好奇的读者会想是什么使仿函数不同于匿名函数或闭包呢？事实证明，它们之间的关系相当明显：标准 Scala 库中的 Function1 类型（指包含一个参数的函数）在其定义上有一个 `apply` 方法。快速浏览一些为 Scala 匿名函数生成的 Scala 匿名类，您就会明白生成的类是 Function1（或者 Function2 或 Function3，这要看该函数使用了几个参数）的后代。

这意味着当匿名的或者命名的函数不一定适合期望设计方法时，Scala 开发人员可以创建一个 `functor` 类，提供给它一些初始化数据，保存在字段中，然后 通过 `()` 执行它，无需任何通用基类（传统的策略模式实现需要这个类）：

清单 **11.** 使用 **Functor**!

```
class ApplyTest
{
  import org.junit._, Assert._

  // ...

  @Test def functorStrategy =
  {
    class GoodAdder
```

```
    {
      def apply(lhs : Int, rhs : Int) : Int = lhs + rhs
    }
    class BadAdder(inflateResults : Int)
    {
      def apply(lhs : Int, rhs : Int) : Int = lhs + rhs * inflateResults
    }

    val calculator = new GoodAdder
    assertEquals(4, calculator(2, 2))
    val enronAccountant = new BadAdder(50)
    assertEquals(102, enronAccountant(2, 2))
  }
}
```

任何提供了被适当赋予了参数的 `apply` 方法的类，只要这些参数都按数字和类型排列了起来，它们都会在被调用时运行。

结束语

Scala 的打包、导入和访问修饰符机制提供了传统 Java 编程人员从未享受过的更高级的控制和封装。例如，它们提供了导入一个对象的选择方法的能力，使它们看起来就像全局方法一样，而且还克服了全局方法的传统的缺点；它们使得使用那些方法变得极其简单，尤其是当这些方法提供了诸如本系列早期 文章（"Scala 控制结构内部揭密"）引入的虚构的 `tryWithLogging` 函数这样的高级功能时。

同样，"应用" 机制允许 Scala 隐藏函数部分的执行细节，这样，编程人员可能会不知道（或不在乎）他们正调用的东西 事实上不是一个函数，而是一个非常复杂的对象。该机制为 Scala 机制的函数特性提供了另一个方面，当然 Java 语言（或者 C# 或 C++）也提供了这个方面，但是它们提供的语法纯度没有 Scala 的高。

这就是本期的全部内容；在下一期发布前，请尽情欣赏!

回页首

下载

描述	名字	大小	下载方法
本文的样例 Scala 代码	j-scala07298.zip	95KB	HTTP

关于下载方法的信息

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。
- "面向 Java 开发人员的 Scala 指南：面向对象的函数编程"（Ted Neward，developerWorks，2008 年 2 月）：本系列的第一篇文章，概述了 Scala 并解释了它的并发性函数方法。本系列中的其他文章有：
  - "类操作"（2008 年 2 月）详细介绍了 Scala 的类语法和语义。
  - "Scala 控制结构内部揭密"（2008 年 3 月）深入到 Scala 的控制结构内部。
  - "关于特征和行为"（2008 年 4 月）利用 Scala 版本的 Java 界面。
  - "实现继承"（2008 年 5 月）介绍了使用 Scala 方式实现的多态。
  - "集合类型"（2008 年 6 月）介绍了全部"元组、数组和列表"。

- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一本介绍 Scala 的书籍。
- Bjarne Stroustrup's FAQ：设计和实现 C++，他将其描述为“更好的 C”。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：开始从本系列学习使用 Scala。
- SUnit：*scala.testing* 包中的标准 Scala 发行版的一部分。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- 特定于领域的语言
- case 类
- 计算器 AST
- 结束语
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南: 构建计算器，第 1 部分

Scala 的 **case** 类和模式匹配

文档选项

打印本页

将此页作为电子邮件发送

英文原文

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 9 月 23 日

特定于领域的语言已经成为一个热门话题；很多函数性语言之所以受欢迎，主要是因为它们可以用于构建特定于领域的语言。鉴于此，在 面向 *Java™* 开发人员的 *Scala* 指南 系列的第 8 篇文章中，Ted Neward 着手构建 一个简单的计算器 DSL，以此来展示函数性语言的构建“外部” DSL 的强大功能。他研究了 Scala 的一个新的特性：*case* 类，并重新审视一个功能强大的特性：模式匹配。

上个月的文章发表后，我又收到了一些抱怨/评论，说我迄今为止在本系列中所用的示例都没涉及到什么实质性的问题。当然在学习一个新语言的初期使用一些小例子是很合理的，而读者想要看到一些更“现实的”示例，从而了解语言的深层领域和强大功能以及其优势，这也是理所当然的。因此，在这个月的文章中，我们来分两部分练习构建特定于领域的语言（DSL）— 本文以一个小的计算器语言为例。

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个新的 developerWorks 系列 中， 您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

特定于领域的语言

可能您无法（或没有时间）承受来自于您的项目经理给您的压力，那么让我直接了当地说吧：特定于领域的语言无非就是尝试（再一次）将一个应用程序的功能放在它该属于的地方 — 用户的手中。

通过定义一个新的用户可以理解并直接使用的文本语言，程序员成功摆脱了不停地处理 UI 请求和 功能增强的麻烦，而且这样还可以使用户能够自己创建脚本以及其他的工具，用来给他们所构建的应用程序创建新的行为。虽然这个例子可能有点冒险（或许会惹来几封抱怨的电子邮件），但我还是要说，DSL 的最成功的 例子就是 Microsoft® Office Excel “语言”，用于表达电子表格单元格的各种计算和内容。 甚至有些人认为 SQL 本身就是 DSL，但这次是一个旨在与关系数据库相交互的语言（想象一下如果程序员要通过传统 API `read()/write()` 调用来从 Oracle 中获取数据的话，那将会是什么样子）。

这里构建的 DSL 是一个简单的计算器语言，用于获取并计算数学表达式。其实，这里的目标是要 创建一个小型语言，这个语言能够允许用户来输入相对简单的代数表达式，然后这个代码来为它求值并产生结果。 为了尽量简单明了，该语言不会支持很多功能完善的计算器所支持的特性，但我不也不想把它的用途限定在教学上 — 该语言一定要具备足够的可扩展性，以使读者无需彻底改变该语言就能够将它用作一个功能更强大的语言的核心。 这意味着该语言一定要可以被轻易地扩展，并要尽量保持封装性，用起来不会有任何的阻碍。

关于 **DSL** 的更多信息

DSL 这个主题的涉及面很广；它的丰富性和广泛性不是本文的一个段落可以描述得了的。想要了解更多 DSL 信息的读者可以查阅本文末尾列出的 Martin Fowler 的“正在进展中的图书”；特别要注意 关于“内部”和“外部” DSL 之间的讨论。Scala 以其灵活的语法和强大的功能而成为最强有力的构建内部和外部 DSL 的语言。

换句话说，（最终的）目标是要允许客户机编写代码，以达到如下的目的：

清单 **1.** 计算器 **DSL**：目标

```
// This is Java using the Calculator
String s = "((5 * 10) + 7)";
double result =
com.tedneward.calcdsl.Calculator.evaluate(s);
System.out.println("We got " + result); //
Should be 57
```

我们不会在一篇文章完成所有的论述，但是我们在本篇文章中可以学习到一部分内容，在下一篇文章完成全部内容。

从实现和设计的角度看，可以从构建一个基于字符串的解析器来着手构建某种可以“挑选每个字符并动态计算”的解析器，这的确极具诱惑力，但是这只适用于较简单的语言，而且其扩展性不是很好。如果语言的目标是实现简单的扩展性，那么在深入研究实现之前，让我们先花点时间想一想如何设计语言。

根据那些基本的编译理论中最精华的部分，您可以得知一个语言处理器（包括解释器和编译器）的基本运算至少由两个阶段组成：

- 解析器，用于获取输入的文本并将其转换成 Abstract Syntax Tree（AST）。
- 代码生成器（在编译器的情况下），用于获取 AST 并从中生成所需字节码；或是求值器（在解释器的情况下），用于获取 AST 并计算它在 AST 里面所发现的内容。

拥有 AST 就能够在某种程度上优化结果树，如果意识到这一点的话，那么上述区别的原因就变得更加显而易见了；对于计算器，我们可能要仔细检查表达式，找出可以截去表达式的整个片段的位置，诸如在乘法表达式中运算数 为“0”的位置（它表明无论其他运算数是多少，运算结果都会是“0”）。

您要做的第一件事是为计算器语言定义该 AST。幸运的是，Scala 有 *case* 类：一种提供了丰富数据、使用了非常薄的封装的类，它们所具有的一些特性使它们很适合构建 AST。

`case` 类

在深入到 AST 定义之前，让我先简要概述一下什么是 case 类。case 类是使 scala 程序员得以使用某些假设的默认值来创建一个类的一种便捷机制。例如，当编写如下内容时：

清单 **2.** 对 **person** 使用 **case** 类

```
case class Person(first:String, last:String, age:Int)
{

}
```

Scala 编译器不仅仅可以按照我们对它的期望生成预期的构造函数 — Scala 编译器还可以生成常规意义上的 `equals()`、`toString()` 和 `hashCode()` 实现。事实上，这种 case 类很普通（即它没有其他的成员），因此 case 类声明后面的大括号的内容是可选的：

清单 **3.** 世界上最短的类清单

```
case class Person(first:String, last:String, age:Int)
```

这一点通过我们的老朋友 `javap` 很容易得以验证：

清单 **4.** 神圣的代码生成器，**Batman**！

```
C:\Projects\Exploration\Scala>javap Person
Compiled from "case.scala"
public class Person extends java.lang.Object implements scala.ScalaObject,scala.
Product,java.io.Serializable{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.Object productElement(int);
    public int productArity();
```

```
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public int age();
    public java.lang.String last();
    public java.lang.String first();
}
```

如您所见，伴随 case 类发生了很多传统类通常不会引发的事情。这是因为 case 类是要与 Scala 的模式匹配（在 "集合类型" 中曾简短分析过）结合使用的。

使用 case 类与使用传统类有些不同，这是因为通常它们都不是通过传统的 "new" 语法构造而成的；事实上，它们通常是通过一种名称与类相同的工厂方法来创建的：

清单 **5.** 没有使用 **new** 语法**?**

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
  }
}
```

case 类本身可能并不比传统类有趣，或者有多么的与众不同，但是在使用它们时会有一个很重要的差别。与引用等式相比，case 类生成的代码更喜欢按位（bitwise）等式，因此下面的代码对 Java 程序员来说有些有趣的惊喜：

清单 **6.** 这不是以前的类

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val ted2 = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println("ted == amanda: " +
      (if (ted == amanda) "Yes" else "No"))
    System.out.println("ted == ted: " +
      (if (ted == ted) "Yes" else "No"))
    System.out.println("ted == ted2: " +
      (if (ted == ted2) "Yes" else "No"))
  }
}

/*
```

```
C:\Projects\Exploration\Scala>scala App
ted == amanda: No
ted == ted: Yes
ted == ted2: Yes
 */
```

case 类的真正价值体现在模式匹配中，本系列的读者可以回顾一下模式匹配（参见 本系列的第二篇文章， 关于 Scala 中的各种控制构造），模式匹配类似 Java 的 “switch/case”，只不过它的本领和功能更加强大。模式匹配不仅能够检查匹配构造的值，从而执行值匹配，还可以针对局部通配符（类似局部 “默认值” 的东西）匹配值，case 还可以包括对测试匹配的保护， 来自匹配标准的值还可以绑定于局部变量，甚至符合匹配标准的类型本身也可以进行匹配。

有了 case 类，模式匹配具备了更强大的功能，如清单 7 所示：

清单 **7.** 这也不是以前的 **switch**

```
case class Person(first:String, last:String, age:Int);

object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println(process(ted))
    System.out.println(process(amanda))
  }
  def process(p : Person) =
  {
    "Processing " + p + " reveals that" +
    (p match
    {
      case Person(_, _, a) if a > 30 =>
        " they're certainly old."
      case Person(_, "Neward", _) =>
        " they come from good genes...."
      case Person(first, last, ageInYears) if ageInYears > 17 =>
        first + " " + last + " is " + ageInYears + " years old."
      case _ =>
        " I have no idea what to do with this person"
    })
  }
}

/*
C:\Projects\Exploration\Scala>scala App
Processing Person(Ted,Neward,37) reveals that they're certainly old.
Processing Person(Amanda,Laucher,27) reveals that Amanda Laucher is 27 years old
.
 */
```

清单 7 中发生了很多操作。下面就让我们先慢慢了解发生了什么，然后回到计算器，看看如何应用它们。

首先，整个 `match` 表达式被包裹在圆括号中：这并非模式匹配语法的要求，但之所以会这样是因为我把模式匹配表达式的结果根据其前面的前缀串联了起来（切记，函数性语言里面的任何东西都是一个表达式）。

其次，第一个 `case` 表达式里面有两个通配符（带下划线的字符就是通配符），这意味着该匹配将会为符合匹配的 `Person` 中那两个字段获取任何值，但是它引入了一个局部变量 `a`，`p.age` 中的值会绑定在这个局部变量上。这个 `case` 只有在同时提供的起保护作用的表达式（跟在它后边的 `if` 表达式）成功时才会成功， 但只有第一个 `Person` 会这样，第二个就不会了。第二个 `case` 表达式 在 `Person` 的 `firstName` 部分使用了一个通配符，但在 `lastName` 部分使用常量字符串 `Neward` 来匹配，在 `age` 部分使用通配符来匹配。

由于第一个 `Person` 已经通过前面的 `case` 匹配了，而且第二个 `Person` 没有姓 `Neward`，所以该匹配不会 为任何一个 `Person` 而被触发（但是，`Person("Michael", "Neward", 15)` 会由于第一个 `case` 中的 `guard` 子句失败而转到第二个 `case`）。

第三个示例展示了模式匹配的一个常见用途，有时称之为提取，在这个提取过程中，匹配对象 `p` 中的值为了能够在 `case` 块内使用而被提取到局部变量中（第一个、最后一个和 `ageInYears`）。最后的 `case` 表达式是普通 `case` 的默认值，它只有在其他 `case` 表达式均未成功的情况下才会被触发。

简要了解了 `case` 类和模式匹配之后，接下来让我们回到创建计算器 AST 的任务上。

计算器 AST

首先，计算器的 AST 一定要有一个公用基类型，因为数学表达式通常都由子表达式组成；通过 “5 + (2 * 10)” 就可以很容易地看到这一点，在这个例子中，子表达式 “(2 * 10)” 将会是 “+” 运算的右侧运算数。

事实上，这个表达式提供了三种 AST 类型：

- 基表达式
- 承载常量值的 Number 类型
- 承载运算和两个运算数的 BinaryOperator

想一下，算数中还允许将一元运算符用作求负运算符（减号），将值从正数转换为负数，因此我们可以引入下列基本 AST：

清单 **8.** 计算器 **AST**（**src/calc.scala**）

```
package com.tedneward.calcdsl
{
  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Number(value : Double) extends Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg : Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String, left : Expr, right : Expr)
   extends Expr
}
```

注意包声明将所有这些内容放在一个包（`com.tedneward.calcdsl`）中， 以及每一个类前面的访问修饰符声明表明该包可以由该包中的其他成员或子包访问。之所以要注意这个是因为需要拥有一系列可以测试这个代码的 JUnit 测试；计算器的实际客户机并不一定非要看到 AST。因此， 要将单元测试编写成 `com.tedneward.calcdsl` 的一个子包：

清单 **9.** 计算器测试（**testsrc/calctest.scala**）

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
```

```
    import org.junit._, Assert._

    @Test def ASTTest =
    {
      val n1 = Number(5)

      assertEquals(5, n1.value)
    }

    @Test def equalityTest =
    {
      val binop = BinaryOp("+", Number(5), Number(10))

      assertEquals(Number(5), binop.left)
      assertEquals(Number(10), binop.right)
      assertEquals("+", binop.operator)
    }
  }
}
```

到目前为止还不错。我们已经有了 AST。

再想一想，我们用了四行 Scala 代码构建了一个类型分层结构，表示一个具有任意深度的数学表达式集合（当然这些数学表达式很简单，但仍然很有用）。与 Scala 能够使对象编程更简单、更具表达力相比，这不算什么（不用担心，真正强大的功能还在后面）。

接下来，我们需要一个求值函数，它将会获取 AST，并求出它的数字值。有了模式匹配的强大功能，编写这样的函数简直轻而易举：

清单 **10.** 计算器（**src/calc.scala**）

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    def evaluate(e : Expr) : Double =
    {
      e match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

注意 `evaluate()` 返回了一个 `Double`，它意味着模式匹配中的每一个 case 都必须被求值成一个 `Double` 值。这个并不难： 数字仅仅返回它们的包含的

值。但对于剩余的 case（有两种运算符），我们还必须在执行必要运算（求负、加法、减法等）前计算运算数。 正如常在函数性语言中所看到的，会使用到递归，所以我们只需要在执行整体运算前对每一个运算数调用 `evaluate()` 就可以了。

大多数忠实于面向对象的编程人员会认为在各种运算符本身以外 执行运算的想法根本就是错误的 — 这个想法显然大大违背了封装和多态性的原则。坦白说，这个甚至不值得讨论； 这很显然违背 了封装原则，至少在传统意义上是这样的。

在这里我们需要考虑的一个更大的问题是：我们到底从哪里封装代码？要记住 AST 类在包外是不可见的， 还有就是客户机（最终）只会传入它们想求值的表达式的一个字符串表示。只有单元测试在直接与 AST case 类合作。

但这并不是说所有的封装都没有用了或过时了。 事实上恰好相反：它试图说服我们在对象领域所熟悉的方法之外，还有很多其他的设计方法也很奏效。 不要忘了 Scala 兼具对象和函数性；有时候 `Expr` 需要在自身及其子类上附加其他行为（例如，实现良好输出的 `toString` 方法），在这种情况下可以很轻松地将这些方法添加到 `Expr`。函数性和面向对象的结合提供了另一种选择，无论是函数性编程人员还是对象编程人员，都不会忽略到另一半的设计方法，并且会考虑如何结合两者来达到一些有趣的效果。

从设计的角度看，有些其他的选择是有问题的；例如，使用字符串来承载运算符就有可能出现小的输入错误，最终会导致结果不正确。 在生产代码中，可能会使用（也许必须使用）枚举而非字符串，使用字符串的话就意味着 我们可能潜在地“开放”了运算符，允许调用出更复杂的函数（诸如 abs、sin、cos、tan 等）乃至用户定义的函数；这些函数是基于枚举的方法很难支持的。

对所有设计和实现的来说，都不存在一个适当的决策方法，只能承担后果。后果自负。

但是这里可以使用一个有趣的小技巧。某些数学表达式可以简化，因而（潜在地）优化了表达式的求值（因此展示了 AST 的有用性）：

- 任何加上“0”的运算数都可以被简化成非零运算数。
- 任何乘以“1”的运算数都可以被简化成非零运算数。
- 任何乘以“0”的运算数都可以被简化成零。

不止这些。因此我们引入了一个在求值前执行的步骤，叫做 `simplify()`，使用它执行这些具体的简化工作：

清单 **11.** 计算器（**src/calc.scala**）

```
def simplify(e : Expr) : Expr =
{
  e match {
    // Double negation returns the original value
    case UnaryOp("-", UnaryOp("-", x)) => x
    // Positive returns the original value
    case UnaryOp("+", x) => x
    // Multiplying x by 1 returns the original value
    case BinaryOp("*", x, Number(1)) => x
    // Multiplying 1 by x returns the original value
    case BinaryOp("*", Number(1), x) => x
    // Multiplying x by 0 returns zero
    case BinaryOp("*", x, Number(0)) => Number(0)
    // Multiplying 0 by x returns zero
    case BinaryOp("*", Number(0), x) => Number(0)
    // Dividing x by 1 returns the original value
    case BinaryOp("/", x, Number(1)) => x
    // Adding x to 0 returns the original value
    case BinaryOp("+", x, Number(0)) => x
    // Adding 0 to x returns the original value
    case BinaryOp("+", Number(0), x) => x
    // Anything else cannot (yet) be simplified
    case _ => e
```

```
    }
  }
```

还是要注意如何使用模式匹配的常量匹配和变量绑定特性，从而使得编写这些表达式可以易如反掌。 对 `evaluate()` 惟一一个更改的地方就是包含了在求值前先简化的调用：

清单 **12.** 计算器（**src/calc.scala**）

```
def evaluate(e : Expr) : Double =
{
  simplify(e) match {
    case Number(x) => x
    case UnaryOp("-", x) => -(evaluate(x))
    case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
    case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
    case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
    case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
  }
}
```

还可以再进一步简化；注意一下：它是如何实现只简化树的最底层的？如果我们有一个包含 `BinaryOp("*", Number(0), Number(5))` 和 `Number(5)` 的 `BinaryOp` 的话，那么内部的 `BinaryOp` 就可以被简化成 `Number(0)`，但外部的 `BinaryOp` 也会如此，这是因为此时 外部 `BinaryOp` 的其中一个运算数是零。

我突然犯了作家的职业病了，所以我想将它留予读者来定义。其实是想增加点趣味性罢了。 如果读者愿意将他们的实现发给我的话，我将会把它放在下一篇文章的代码分析中。将会有两个测试单元来测试这种情况，并会立刻失败。您的任务（如果您选择接受它的话）是使这些测试 — 以及其他任何测试，只要该测试采取了任意程度的 `BinaryOp` 和 `UnaryOp` 嵌套 — 通过。

结束语

显然我还没有说完；还有分析的工作要做，但是计算器 AST 已经成形。我们无需作出大的变动就可以添加其他的运算，运行 AST 也无需大量的代码（按照 Gang of Four 的 Visitor 模式），而且我们已经有了一些执行计算本身的工作代码（如果客户机愿意为我们构建用于求值的代码的话）。

更重要的是，您已经看到了 case 类是如何与模式匹配合作，使得创建 AST 并对其求值变得轻而易举。这是 Scala 代码（以及大多数函数性语言）很常用的设计，而且如果您准备认真地研究这个环境的话，这是您应当掌握的内容之一。

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。
- "面向 Java 开发人员的 Scala 指南：集合类型"（Ted Neward，developerWorks，2008 年 6 月）论述了模式匹配。
- "面向 Java 开发人员的 Scala 指南：类操作"（Ted Neward, developerWorks, February 2008) 论述了 Scala 中的各种控制构造。
- " 面向 *Java* 开发人员的 *Scala* 指南（Ted Neward，developerWorks）：阅读整个系列。
- "Java 语言中的函数编程"（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。

- “Scala by Example”（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一份 Scala 介绍，篇幅和一本书差不多。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：开始学习本系列。
- SUnit：*scala.testing* 包中的标准 Scala 发行版的一部分。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- 解析
- 解析器组合子
- 解析器组合子概念入门
- 我们还没有完成，是吗?
- 结束语
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南: 构建计算器，第 2 部分

Scala 的解析器组合子

文档选项

打印本页

将此页作为电子邮件发送

英文原文

级别： 初级

Ted Neward, 主管, Neward & Associates

2008 年 11 月 17 日

特定领域语言（Domain-specific languages，DSL）已经成为一个热门话题；很多函数性语言之所以受欢迎，主要是因为它们可以用于构建 DSL。有鉴于此，在 面向 *Java* 开发人员的 *Scala* 指南 系列最近的一篇文章中，Ted Neward 继续讨论一个简单的计算器 DSL，以展示函数性语言在构建"外部"DSL 的强大功能，并在此过程中解决将文本输入转换成用于解释的 AST 的问题。为了解析文本输入，并将它转换成上一篇文章中解释器使用的树结构，Ted 引入了 解析器组合子（*parser combinator*），这是一个专门为这项任务设计的标准 Scala 库。（在 上一篇文章 中，我们构建了一个计算器解析器和 AST）。

回忆一下我们的英雄所处的困境：在试图创建一个 DSL（这里只不过是一种非常简单的计算器语言）时，他创建了包含可用于该语言的各种选项的树结构：

- 二进制加/减/乘/除运算符
- 一元反运算符
- 数值

它背后的执行引擎知道如何执行那些操作，它甚至有一个显式的优化步骤，以减少获得结果所需的计算。

最后的 代码 是这样的：

清单 **1.** 计算器 **DSL**：**AST** 和解释器

```
package com.tedneward.calcdsl
{
  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Variable(name : String) extends Expr
  private[calcdsl]  case class Number(value : Double) extends Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg : Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String, left : Expr, right : Expr)
   extends Expr

  object Calc
  {
    /**
     * Function to simplify (a la mathematic terms) expressions
     */
    def simplify(e : Expr) : Expr =
    {
      e match {
        // Double negation returns the original value
```

```
        case UnaryOp("-", UnaryOp("-", x)) => simplify(x)

        // Positive returns the original value
        case UnaryOp("+", x) => simplify(x)

        // Multiplying x by 1 returns the original value
        case BinaryOp("*", x, Number(1)) => simplify(x)

        // Multiplying 1 by x returns the original value
        case BinaryOp("*", Number(1), x) => simplify(x)

        // Multiplying x by 0 returns zero
        case BinaryOp("*", x, Number(0)) => Number(0)

        // Multiplying 0 by x returns zero
        case BinaryOp("*", Number(0), x) => Number(0)

        // Dividing x by 1 returns the original value
        case BinaryOp("/", x, Number(1)) => simplify(x)

        // Dividing x by x returns 1
        case BinaryOp("/", x1, x2) if x1 == x2 => Number(1)

        // Adding x to 0 returns the original value
        case BinaryOp("+", x, Number(0)) => simplify(x)

        // Adding 0 to x returns the original value
        case BinaryOp("+", Number(0), x) => simplify(x)

        // Anything else cannot (yet) be simplified
        case _ => e
      }
    }

    def evaluate(e : Expr) : Double =
    {
      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

前一篇文章的读者应该还记得，我布置了一个挑战任务，要求改进优化步骤，进一步在树中进行简化处理，而不是像清单 1 中的代码那样停留在最顶层。Lex Spoon 发现了我认为是最简单的优化方法：首先简化树的“边缘”（每个表达式中的操作数，如果有的话），然后利用简化的结果，再进一步简化顶层的表达式，如清单 2 所示：

清单 **2.** 简化、再简化

```
/*
 * Lex's version:
 */
def simplify(e: Expr): Expr = {
  // first simplify the subexpressions
  val simpSubs = e match {
    // Ask each side to simplify
    case BinaryOp(op, left, right) => BinaryOp(op, simplify(left), simplify(right))
    // Ask the operand to simplify
    case UnaryOp(op, operand) => UnaryOp(op, simplify(operand))
    // Anything else doesn't have complexity (no operands to simplify)
    case _ => e
  }

  // now simplify at the top, assuming the components are already simplified
  def simplifyTop(x: Expr) = x match {
    // Double negation returns the original value
    case UnaryOp("-", UnaryOp("-", x)) => x

    // Positive returns the original value
    case UnaryOp("+", x) => x

    // Multiplying x by 1 returns the original value
    case BinaryOp("*", x, Number(1)) => x

    // Multiplying 1 by x returns the original value
    case BinaryOp("*", Number(1), x) => x

    // Multiplying x by 0 returns zero
    case BinaryOp("*", x, Number(0)) => Number(0)

    // Multiplying 0 by x returns zero
    case BinaryOp("*", Number(0), x) => Number(0)

    // Dividing x by 1 returns the original value
    case BinaryOp("/", x, Number(1)) => x

    // Dividing x by x returns 1
    case BinaryOp("/", x1, x2) if x1 == x2 => Number(1)

    // Adding x to 0 returns the original value
    case BinaryOp("+", x, Number(0)) => x

    // Adding 0 to x returns the original value
    case BinaryOp("+", Number(0), x) => x

    // Anything else cannot (yet) be simplified
```

```
        case e => e
      }
      simplifyTop(simpSubs)
    }
```

在此对 Lex 表示感谢。

解析

现在是构建 DSL 的另一半工作：我们需要构建一段代码，它可以接收某种文本输入并将其转换成一个 AST。这个过程更正式的称呼是解析（*parsing*）（更准确地说，是标记解释（*tokenizing*）、词法解析（*lexing*） 和语法解析）。

以往，创建解析器有两种方法：

- 手工构建一个解析器。
- 通过工具生成解析器。

我们可以试着手工构建这个解析器，方法是手动地从输入流中取出一个字符，检查该字符，然后根据该字符以及在它之前的其他字符（有时还要根据在它之后的字符）采取某种行动。对于较小型的语言，手工构建解析器可能更快速、更容易，但是当语言变得更庞大时，这就成了一个困难的问题。

除了手工编写解析器外，另一种方法是用工具生成解析器。以前有 2 个工具可以实现这个目的，它们被亲切地称作*lex*（因为它生成一个 "词法解析器"）和 *yacc*（"Yet Another Compiler Compiler"）。对编写解析器感兴趣的程序员没有手工编写解析器，而是编写一个不同的源文件，以此作为 "lex" 的输入，后者生成解析器的前端。然后，生成的代码会与一个 "grammar" 文件 —— 它定义语言的基本语法规则（哪些标记中是关键字，哪里可以出现代码块，等等）—— 组合在一起，并且输入到 yacc 生成解析器代码。

由于这是 Computer Science 101 教科书，所以我不会详细讨论有限状态自动机（finite state automata）、LALR 或 LR 解析器，如果需要深入了解请查找与这个主题相关的书籍或文章。

同时，我们来探索 Scala 构建解析器的第 3 个选项：解析器组合子（*parser combinators*），它完全是从 Scala 的函数性方面构建的。解析器组合子使我们可以将语言的各种片段 "组合" 成部件，这些部件可以提供不需要代码生成，而且看上去像是一种语言规范的解决方案。

解析器组合子

了解 *Becker-Naur Form*（*BNF*）有助于理解解析器组合子的要点。BNF 是一种指定语言的外观的方法。例如，我们的计算器语言可以用清单 3 中的 BNF 语法进行描述：

清单 **3.** 对语言进行描述

```
input   ::= ws expr ws eoi;

expr    ::= ws powterm [{ws '^' ws powterm}];
powterm ::= ws factor [{ws ('*'|'/') ws factor}];
factor  ::= ws term [{ws ('+'|'-') ws term}];
term    ::= '(' ws expr ws ')' | '-' ws expr | number;

number  ::= {dgt} ['.' {dgt}] [('e'|'E') ['-'] {dgt}];
dgt     ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
ws      ::= [{' '|'\t'|'\n'|'\r'}];
```

语句左边的每个元素是可能的输入的集合的名称。右边的元素也称为 *term*，它们是一系列表达式或文字字符，按照可选或必选的方式进行组合。（同样，BNF 语法在 Aho/Lam/Sethi/Ullman 等书籍中有更详细的描述，请参阅 参考资料）。

用 BNF 形式来表达语言的强大之处在于，BNF 和 Scala 解析器组合子不相上下；清单 4 显示使用 BNF 简化形式后的清单 3：

清单 **4.** 简化、再简化

```
expr   ::= term {'+' term | '-' term}
term   ::= factor {'*' factor | '/' factor}
factor ::= floatingPointNumber | '(' expr ')'
```

其中花括号（{}）表明内容可能重复（0 次或多次），竖线（|）表明也/或的关系。因此，在读清单 4 时，一个 `factor` 可能是一个 `floatingPointNumber`（其定义在此没有给出），或者一个左括号加上一个 `expr` 再加上一个右括号。

在这里，将它转换成一个 Scala 解析器非常简单，如清单 5 所示：

清单 **5.** 从 **BNF** 到 **parsec**

```
package com.tedneward.calcdsl
{
  object Calc
  {
    // ...

    import scala.util.parsing.combinator._

    object ArithParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    def parse(text : String) =
    {
      val results = ArithParser.parse(text)
      System.out.println("parsed " + text + " as " + results + " which is a type "
       + results.getClass())
    }

        // ...
  }
}
```

BNF 实际上被一些解析器组合子语法元素替换：空格被替换为 ~ 方法（表明一个序列），重复被替换为 `rep` 方法，而选择则仍然用 | 方法来表示。文字字符串是标准的文字字符串。

从两个方面可以看到这种方法的强大之处。首先，该解析器扩展 Scala 提供的 `JavaTokenParsers` 基类（后者本身又继承其他基类，如果我们想要一种与

Java 语言的语法概念不那么严格对齐的语言的话），其次，使用 floatingPointNumber 预设的组合子来处理解析一个浮点数的细节。

这种特定的（一个中缀计算器的）语法很容易使用（这也是在那么多演示稿和文章中看到它的原因），为它手工构建一个解析器也不困难，因为 BNF 语法与构建解析器的代码之间的紧密关系使我们可以更快、更容易地构建解析器。

解析器组合子概念入门

为了理解其中的原理，我们必须简要了解解析器组合子的实现。实际上，每个“解析器”都是一个函数或一个 case 类，它接收某种输入，并产生一个“解析器”。例如，在最底层，解析器组合子位于一些简单的解析器之上，这些解析器以某种输入读取元素（一个 Reader）作为输入，并生成某种可以提供更高级的语义的东西（一个 Parser）：

清单 **6.** 一个基本的解析器

```
type Elem

type Input = Reader[Elem]

type Parser[T] = Input => ParseResult[T]

sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input) extends ParseResult[T]
case class Failure(msg: String, in: Input) extends ParseResult[Nothing]
```

换句话说，Elem 是一种抽象类型，用于表示任何可被解析的东西，最常见的是一个文本字符串或流。然后，Input 是围绕那种类型的一个 scala.util.parsing.input.Reader（方括号表明 Reader 是一个泛型；如果您喜欢 Java 或 C++ 风格的语法，那么将它们看作尖括号）。然后，T 类型的 Parser 是这样的类型：它接受一个 Input，并生成一个 ParseResult，后者（基本上）属于两种类型之一：Success 或 Failure。

显然，关于解析器组合子库的知识远不止这些 — 即使 ~ 和 rep 函数也不是几个步骤就可以得到的 — 但是，这让您对解析器组合子的工作原理有基本的了解。“组合”解析器可以提供解析概念的越来越高级的抽象（因此称为“解析器组合子”；组合在一起的元素提供解析行为）。

我们还没有完成，是吗？

我们仍然没有完成。通过调用快速测试解析器可以发现，解析器返回的内容并不是计算器系统需要的剩余部分：

清单 **7.** 第一次测试失败？

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

        // ...

    @Test def parseNumber =
    {
      assertEquals(Number(5), Calc.parse("5"))
      assertEquals(Number(5), Calc.parse("5.0"))
    }
  }
}
```

这次测试会在运行时失败，因为解析器的 `parseAll` 方法不会返回我们的 case 类 `Number`（这是有道理的，因为我们没有在解析器中建立 case 类与解析器的产生规则之间的关系）；它也没有返回一个文本标记或整数的集合。

相反，解析器返回一个 `Parsers.ParseResult`，这是一个 `Parsers.Success` 实例（其中有我们想要的结果）；或者一个 `Parsers.NoSuccess`、`Parsers.Failure` 或 `Parsers.Error`（后三者的性质是一样的：解析由于某种原因未能正常完成）。

假设这是一次成功的解析，要得到实际结果，必须通过 `ParseResult` 上的 `get` 方法来提取结果。这意味着必须稍微调整 `Calc.parse` 方法，以便通过测试。如清单 8 所示：

清单 **8.** 从 **BNF** 到 **parsec**

```
package com.tedneward.calcdsl
{
  object Calc
  {
    // ...

    import scala.util.parsing.combinator._

    object ArithParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    def parse(text : String) =
    {
      val results = ArithParser.parse(text)
      System.out.println("parsed " + text + " as " + results + " which is a type "
         + results.getClass())
          results.get
    }

        // ...
  }
}
```

成功了！真的吗？

对不起，还没有成功。运行测试表明，解析器的结果仍不是我前面创建的 AST 类型（`expr` 和它的亲属），而是由 List 和 String 等组成的一种形式。虽然可以将这些结果解析成 `expr` 实例并对其进行解释，但是肯定还有另外一种方法。

确实有另外一种方法。为了理解这种方法的工作原理，您将需要研究一下解析器组合子是如何产生非“标准”的元素的（即不是 String 和 List）。用适当的术语来说就是解析器如何才能产生一个定制的元素（在这里，就是 AST 对象）。这个主题下一次再讨论。

在下一期中，我将和您一起探讨解析器组合子实现的基础，并展示如何将文本片段解析成一个 AST，以便进行求值（然后进行编译）。

结束语

显然，我们还没有结束（解析工作还没有完成），但是现在有了基本的解析器语义，接下来只需通过扩展解析器产生元素来生成 AST 元素。

对于那些想领先一步的读者，可以查看 ScalaDocs 中描述的 ^^ 方法，或者阅读 *Programming in Scala* 中关于解析器组合子的小节；但是，在此提醒一下，这门语言比这些参考资料中给出的例子要复杂一些。

当然，您可以只与 String 和 List 打交道，而忽略 AST 部分，拆开返回的 String 和 List，并重新将它们解析成 AST 元素。但是，解析器组合子库已经包含很多这样的内容，没有必要再重复一遍。

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个 developerWorks 系列 中，您将深入了解 Sacla，并在实践中看到 Scala 的语言功能。进行相关比较时，Scala 代码和 Java™ 代码将放在一起展示，但是（您将发现）Scala 与 Java 中的许多东西都没有直接的联系，这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。
- "面向 Java 开发人员的 Scala 指南：面向对象的函数编程"（Ted Neward，developerWorks，2008 年 1 月）：本系列的第 1 篇文章概述了 Scala，并解释了它的函数性方法等。本系列中的其他文章：
  - "类操作"（2008 年 2 月）详述了 Scala 的类语法和语义。
  - "Scala 控制结构内部揭密"（2008 年 3 月）深入讨论了 Scala 的控制结构。
  - "关于特征和行为"（2008 年 4 月）：利用 Scala 版本的 Java 接口。
  - "实现继承"（2008 年 5 月）：Scala 式的多态。
  - "集合类型"（2008 年 6 月）包括所有"元组、数组和列表"。
  - "包和访问修饰符"（2008 年 7 月）讨论了 Scala 的包和访问修饰符，以及 `apply` 机制。
  - "构建计算器，第 1 部分"（2008 年 8 月）是本文的前半部分。
- "Java 语言中的函数编程"（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度解释函数性编程的优点和用法。
- "Scala by Example"（Martin Odersky，2007 年 12 月）：一篇简短的、含有大量代码的 Scala 入门介绍（PDF 格式）。
- Programming in Scala（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一次以一本书的篇幅介绍 Scala。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：通过这个系列开始学习 Scala。
- SUnit：标准 Scala 发行版的一部分，在 *scala.testing* 包中。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- (•) 太差！ (1)
- ( ) 需提高 (2)
- ( ) 一般；尚可 (3)
- ( ) 好文章 (4)
- ( ) 真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南: 构建计算器，第 3 部分

将 **Scala** 解析器组合子和 **case** 类结合起来

级别： 初级

Ted Neward, 主管, Neward & Associates

2009 年 1 月 19 日

特定领域语言（Domain-specific languages，DSL）已经成为一个热门话题；围绕函数性语言讨论得最多的话题是构建这种语言的能力。在构建了 AST 模式和基本前端解析器 之后（用于获取文本和生成适合解释的对象图形），作者在这篇文章中将这些知识无缝地整合起来（虽然有点麻烦）。然后他将推荐一些适合 DSL 语言及其解释器的扩展。

欢迎勇于探索的读者回到我们的系列文章中！本月继续探索 Scala 的语言和库支持，我们将改造一下计算器 DSL 并最终“完成它”。DSL 本身有点简单 — 一个简单的计算器，目前为止只支持 4 个基本数学运算符。但要记住，我们的目标是创建一些可扩展的、灵活的对象，并且以后可以轻松增强它们以支持新的功能。

继续上次的讨论.....

说明一下，目前我们的 DSL 有点零乱。我们有一个抽象语法树（Abstract Syntax Tree ），它由大量 case 类组成.....

清单 **1.** 后端（**AST**）

```
package com.tedneward.calcdsl
{
  // ...

  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Variable(name : String) extends
Expr
  private[calcdsl]  case class Number(value : Double) extends
Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg
: Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String,
left : Expr, right : Expr)
    extends Expr

}
```

.....对此我们可以提供类似解释器的行为，它能最大限度地简化数学表达式.....

清单 **2.** 后端（解释器）

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    def simplify(e: Expr): Expr = {
      // first simplify the subexpressions
      val simpSubs = e match {
        // Ask each side to simplify
        case BinaryOp(op, left, right) => BinaryOp(op, simplify(left), simplify(right))
        // Ask the operand to simplify
        case UnaryOp(op, operand) => UnaryOp(op, simplify(operand))
        // Anything else doesn't have complexity (no operands to simplify)
        case _ => e
      }

      // now simplify at the top, assuming the components are already simplified
      def simplifyTop(x: Expr) = x match {
        // Double negation returns the original value
        case UnaryOp("-", UnaryOp("-", x)) => x

        // Positive returns the original value
        case UnaryOp("+", x) => x

        // Multiplying x by 1 returns the original value
        case BinaryOp("*", x, Number(1)) => x

        // Multiplying 1 by x returns the original value
        case BinaryOp("*", Number(1), x) => x

        // Multiplying x by 0 returns zero
        case BinaryOp("*", x, Number(0)) => Number(0)

        // Multiplying 0 by x returns zero
        case BinaryOp("*", Number(0), x) => Number(0)

        // Dividing x by 1 returns the original value
        case BinaryOp("/", x, Number(1)) => x

        // Dividing x by x returns 1
        case BinaryOp("/", x1, x2) if x1 == x2 => Number(1)

        // Adding x to 0 returns the original value
        case BinaryOp("+", x, Number(0)) => x

        // Adding 0 to x returns the original value
        case BinaryOp("+", Number(0), x) => x

        // Anything else cannot (yet) be simplified
```

```
        case e => e
      }
      simplifyTop(simpSubs)
    }

    def evaluate(e : Expr) : Double =
    {
      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

.....我们使用了一个由 Scala 解析器组合子构建的文本解析器，用于解析简单的数学表达式.....

清单 **3.** 前端

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    object ArithParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    // ...
  }
}
```

.....但在进行解析时，由于解析器组合子当前被编写为返回 `Parser[Any]` 类型，所以会生成 String 和 List 集合，实际上应该让解析器返回它需要的任意类型（我们可以看到，此时是一个 String 和 List 集合）。

要让 DSL 成功，解析器需要返回 AST 中的对象，以便在解析完成时，执行引擎可以捕获该树并对它执行 `evaluate()`。对于该前端，我们需要更改解析

器组合子实现，以便在解析期间生成不同的对象。

清理语法

对解析器做的第一个更改是修改其中一个语法。在原来的解析器中，可以接受像 “5 + 5 + 5” 这样的表达式，因为语法中为表达式（expr）和术语（term）定义了 rep() 组合子。但如果考虑扩展，这可能会引起一些关联性和操作符优先级问题。以后的运算可能会要求使用括号来显式给出优先级，以避免这类问题。因此第一个更改是将语法改为要求在所有表达式中加 “()”。

回想一下，这应该是我一开始就需要做的事情；事实上，放宽限制通常比在以后添加限制容易（如果最后不需要这些限制），但是解决运算符优先级和关联性问题比这要困难得多。如果您不清楚运算符的优先级和关联性；那么让我大致概述一下我们所处的环境将有多复杂。考虑 Java 语言本身和它支持的各种运算符（如 Java 语言规范中所示）或一些关联性难题（来自 Bloch 和 Gafter 提供的 *Java Puzzlers*），您将发现情况不容乐观。

因此，我们需要逐步解决问题。首先是再次测试语法：

清单 **4.** 采用括号

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    object OldAnyParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }
    object AnyParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = (term~"+"~term) | (term~"-"~term) | term
      def term : Parser[Any] = (factor~"*"~factor) | (factor~"/"~factor) | factor
      def factor : Parser[Any] = "(" ~> expr <~ ")" | floatingPointNumber

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    // ...
  }
}
```

我已经将旧的解析器重命名为 `OldAnyParser`，添加左边的部分是为了便于比较；新的语法由 `AnyParser` 给出；注意它将 `expr` 定义为 `term + term`、`term - term`，或者一个独立的 `term`，等等。另一个大的变化是 `factor` 的定义，现在它使用另一种组合子 `~>` 和 `<~` 在遇到 `(` 和 `)` 字符时有效地抛出它们。

因为这只是一个临时步骤，所以我不打算创建一系列单元测试来查看各种可能性。不过我仍然想确保该语法的解析结果符合预期，所以我在这里编写一个不是很正式的测试：

清单 **5.** 测试解析器的非正式测试

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    _cnnew1@Test def parse =
    {
      import Calc._

      val expressions = List(
        "5",
        "(5)",
        "5 + 5",
        "(5 + 5)",
        "5 + 5 + 5",
        "(5 + 5) + 5",
        "(5 + 5) + (5 + 5)",
        "(5 * 5) / (5 * 5)",
        "5 - 5",
        "5 - 5 - 5",
        "(5 - 5) - 5",
        "5 * 5 * 5",
        "5 / 5 / 5",
        "(5 / 5) / 5"
      )

      for (x <- expressions)
        System.out.println(x + " = " + AnyParser.parse(x))
    }
  }
}
```

请记住，这纯粹是出于教学目的（也许有人会说我不想为产品代码编写测试，但我确实没有在编写产品代码，所以我不需要编写正式的测试。这只是为了方便教学）。但是，运行这个测试后，得到的许多结果与标准单元测试结果文件相符，表明没有括号的表达式（`5 + 5 + 5`）执行失败，而有括号的表达式则会执行成功。真是不可思议！

不要忘了给解析测试加上注释。更好的方法是将该测试完全删除。这是一个临时编写的测试，而且我们都知道，真正的 Jedi 只在研究或防御时使用这些源代码，而不在这种情况中使用。

清理语法

现在我们需要再次更改各种组合子的定义。回顾一下上一篇文章，`expr`、`term` 和 `factor` 函数中的每一个实际上都是 BNF 语句，但注意每一个函数返回的都是一个解析器泛型，参数为 `Any`（Scala 类型系统中一个基本的超类型，从其名称就可以知道它的作用：指示可以包含任何对象的潜在类型或引用）；这表明组合子可以根据需要返回任意类型。我们已经看到，在默认情况下，解析器可以返回一个 String，也可以返回一个 List（如果您还不信的话，可以在运行的测试中加入临时测试。这也会看到同样的结果）。

要将它更改为生成 case 类 AST 层次结构的实例（`Expr` 对象），组合子的返回类型必须更改为 `Parser[Expr]`。如果让它自行更改，编译将会失败；这三个组合子知道如何获取 String，但不知道如何根据解析的内容生成 `Expr` 对象。为此，我们使用了另一个组合子，即 `^^` 组合子，它以一个匿名函数为参数，将解析的结果作为一个参数传递给该匿名函数。

如果您和许多 Java 开发人员一样，那么就要花一点时间进行解析，让我们查看一下实际效果：

清单 **6.** 产品组合子

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    object ExprParser extends JavaTokenParsers
    {
      def expr: Parser[Expr] =
        (term ~ "+" ~ term) ^^ { case lhs~plus~rhs => BinaryOp("+", lhs, rhs) } |
        (term ~ "-" ~ term) ^^ { case lhs~minus~rhs => BinaryOp("-", lhs, rhs) } |
        term

      def term: Parser[Expr] =
        (factor ~ "*" ~ factor) ^^ { case lhs~times~rhs => BinaryOp("*", lhs, rhs) } |
        (factor ~ "/" ~ factor) ^^ { case lhs~div~rhs => BinaryOp("/", lhs, rhs) } |
        factor

      def factor : Parser[Expr] =
        "(" ~> expr <~ ")" |
        floatingPointNumber ^^ {x => Number(x.toFloat) }

      def parse(text : String) = parseAll(expr, text)
    }

    def parse(text : String) =
      ExprParser.parse(text).get

    // ...
  }

  // ...
}
```

`^^` 组合子接收一个匿名函数，其解析结果（例如，假设输入的是 `5 + 5`，那么解析结果将是 `((5~+)~5)`）将会被单独传递并得到一个对象 — 在本例中，是一个适当类型的 `BinaryObject`。请再次注意模式匹配的强大功能；我将表达式的左边部分与 `lhs` 实例绑定在一起，将 `+` 部分与（未使用的）`plus` 实

例绑定在一起，该表达式的右边则与 `rhs` 绑定，然后我分别使用 `lhs` 和 `rhs` 填充 `BinaryOp` 构造函数的左边和右边。

现在运行代码（记得注释掉临时测试），单元测试集会再次产生所有正确的结果：我们以前尝试的各种表达式不会再失败，因为现在解析器生成了派生 `Expr` 对象。前面已经说过，不进一步测试解析器是不负责任的，所以让我们添加更多的测试（包括我之前在解析器中使用的非正式测试）：

清单 **7.** 测试解析器（这次是正式的）

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    @Test def parseAnExpr1 =
      assertEquals(
        Number(5),
        Calc.parse("5")
      )
    @Test def parseAnExpr2 =
      assertEquals(
        Number(5),
        Calc.parse("(5)")
      )
    @Test def parseAnExpr3 =
      assertEquals(
        BinaryOp("+", Number(5), Number(5)),
        Calc.parse("5 + 5")
      )
    @Test def parseAnExpr4 =
      assertEquals(
        BinaryOp("+", Number(5), Number(5)),
        Calc.parse("(5 + 5)")
      )
    @Test def parseAnExpr5 =
      assertEquals(
        BinaryOp("+", BinaryOp("+", Number(5), Number(5)), Number(5)),
        Calc.parse("(5 + 5) + 5")
      )
    @Test def parseAnExpr6 =
      assertEquals(
        BinaryOp("+", BinaryOp("+", Number(5), Number(5)), BinaryOp("+", Number(5),
                  Number(5))),
        Calc.parse("(5 + 5) + (5 + 5)")
      )

    // other tests elided for brevity
  }
}
```

读者可以再增加一些测试，因为我可能漏掉一些不常见的情况（与 Internet 上的其他人结对编程是比较好的）。

完成最后一步

假设解析器正按照我们想要的方式在工作 — 即生成 AST — 那么现在只需要根据 AST 对象的计算结果来完善解析器。这很简单，只需向 `Calc` 添加代码，如清单 8 所示.....

清单 **8.** 真的完成啦!

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    def evaluate(text : String) : Double = evaluate(parse(text))
  }
}
```

.....同时添加一个简单的测试，确保 `evaluate("1+1")` 返回 2.0.....

清单 **9.** 最后，看一下 **`1 + 1`** 是否等于 **2**

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    @Test def add1 =
      assertEquals(Calc.evaluae("1 + 1"), 2.0)
  }
}
```

.....然后运行它，一切正常!

扩展 DSL 语言

如果完全用 Java 代码编写同一个计算器 DSL，而没有碰到我遇到的问题（在不构建完整的 AST 的情况下递归式地计算每一个片段，等等），那么似乎它是另一种能够解决问题的语言或工具。但以这种方式构建语言的强大之处会在扩展性上得到体现。

例如，我们向这种语言添加一个新的运算符，即 ^ 运算符，它将执行求幂运算；也就是说，*2 ^ 2* 等于 *2* 的平方 或 4。向 DSL 语言添加这个运算符需要一些简单步骤。

首先，您必须考虑是否需要更改 AST。在本例中，求幂运算符是另一种形式的二进制运算符，所以使用现有 `BinaryOp case` 类就可以。无需对 AST 进行

任何更改。

其次，必须修改 `evaluate` 函数，以使用 `BinaryOp("^", x, y)` 执行正确的操作；这很简单，只需添加一个嵌套函数（因为不必在外部看到这个函数）来实际计算指数，然后向模式匹配添加必要的代码行，如下所示：

清单 **10.** 稍等片刻

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    def evaluate(e : Expr) : Double =
    {
      def exponentiate(base : Double, exponent : Double) : Double =
        if (exponent == 0)
          1.0
        else
          base * exponentiate(base, exponent - 1)

      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
        case BinaryOp("^", x1, x2) => exponentiate(evaluate(x1), evaluate(x2))
      }
    }
  }
}
```

注意，这里我们只使用 6 行代码就有效地向系统添加了求幂运算，同时没有对 `Calc` 类进行任何表面更改。这就是封装！

（在我努力创建最简单求幂函数时，我故意创建了一个有严重 bug 的版本 —— 这是为了让我们关注语言，而不是实现。也就是说，看看哪位读者能够找到 bug。他可以编写发现 bug 的单元测试，然后提供一个无 bug 的版本）。

但是在向解析器添加这个求幂函数之前，让我们先测试这段代码，以确保求幂部分能正常工作：

清单 **11.** 求平方

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...
```

```
    @Test def evaluateSimpleExp =
    {
      val expr =
        BinaryOp("^", Number(4), Number(2))
      val results = Calc.evaluate(expr)
      // (4 ^ 2) => 16
      assertEquals(16.0, results)
    }
    @Test def evaluateComplexExp =
    {
      val expr =
        BinaryOp("^",
          BinaryOp("*", Number(2), Number(2)),
          BinaryOp("/", Number(4), Number(2)))
      val results = Calc.evaluate(expr)
      // ((2 * 2) ^ (4 / 2)) => (4 ^ 2) => 16
      assertEquals(16.0, results)
    }
  }
}
```

运行这段代码确保可以求幂（忽略我之前提到的 bug），这样就完成了一半的工作。

最后一个更改是修改语法，让它接受新的求幂运算符；因为求幂的优先级与乘法和除法的相同，所以最简单的做法是将它放在 `term` 组合子中：

清单 **12.** 完成了，这次是真的！

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    object ExprParser extends JavaTokenParsers
    {
      def expr: Parser[Expr] =
        (term ~ "+" ~ term) ^^ { case lhs~plus~rhs => BinaryOp("+", lhs, rhs) } |
        (term ~ "-" ~ term) ^^ { case lhs~minus~rhs => BinaryOp("-", lhs, rhs) } |
        term

      def term: Parser[Expr] =
        (factor ~ "*" ~ factor) ^^ { case lhs~times~rhs => BinaryOp("*", lhs, rhs) } |
        (factor ~ "/" ~ factor) ^^ { case lhs~div~rhs => BinaryOp("/", lhs, rhs) } |
        (factor ~ "^" ~ factor) ^^ { case lhs~exp~rhs => BinaryOp("^", lhs, rhs) } |
        factor

      def factor : Parser[Expr] =
```

```
        "(" ~> expr <~ ")" |
        floatingPointNumber ^^ {x => Number(x.toFloat) }

      def parse(text : String) = parseAll(expr, text)
    }

  // ...
  }
}
```

当然，我们需要对这个解析器进行一些测试.....

清单 **13.** 再求平方

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...

    @Test def parseAnExpr17 =
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("2 ^ 2")
      )
    @Test def parseAnExpr18 =
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("(2 ^ 2)")
      )
    @Test def parseAnExpr19 =
      assertEquals(
        BinaryOp("^", Number(2),
          BinaryOp("+", Number(1), Number(1))),
        Calc.parse("2 ^ (1 + 1)")
      )
    @Test def parseAnExpr20 =
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("2 ^ (2)")
      )
  }
}
```

.....运行并通过后，还要进行最后一个测试，看一切是否能正常工作：

清单 **14.** 从 **String** 到平方

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...

    @Test def square1 =
      assertEquals(Calc.evaluate("2 ^ 2"), 4.0)
  }
}
```

成功啦!

结束语

显然，还要做更多工作才能使这门简单的语言变得更好；不管您对该语言的各个部分测试（AST、解析器、简化引擎，等等）感觉如何，仅仅将该语言编写为基于解释器的对象都可以通过更少的代码来实现（也可能更快，这取决于您的熟练程度），甚至可以动态地计算表达式的值，而不是将它们转换为 AST 后再进行计算。

向系统添加另一种运算符是非常简单的。该语言的设计也使它的扩展非常容易，扩展时不需要修改很多代码。事实上，我们可以通过许多增强来演示该方法的内在灵活性：

关于本系列

Ted Neward 将和您一起探讨 Scala 编程语言。在这个 developerWorks 系列中，您将深入了解 Scala，并在实践中看到 Scala 的语言功能。进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但是（您将发现）Scala 与 Java 中的许多东西都没有直接联系，这正是 Scala 的魅力所在！如果用 Java 代码就能实现的话，又何必再学习 Scala 呢？

- 我们可以从使用 `Doubles` 转向使用 `BigDecimals` 或 `BigIntegers`，而不是用 java.math 包（以允许进行更大和/或更准确的计算）。
- 我们可以在语言中支持十进制数（当前解析器中不支持）。
- 我们可以使用单词（“sin”、“cos”、“tan” 等）而不是符号来添加运算符。
- 我们甚至可以添加变量符号（“`x = y + 12`”）并接受 `Map` 作为 `evaluate()` 函数的参数，该函数包含每个变量的初始值。

更重要的是，DSL 完全隐藏在 `Calc` 类后面，这意味着从 Java 代码调用它与从 Scala 调用它一样简单。所以即使在没有完全采用 Scala 作为首选语言的项目中，也可以用 Scala 编写部分系统（那些最适合使用函数性/对象混合语言的部分），而且 Java 开发人员可以轻松地调用它们。

本文到此结束。在下一篇文章中，我们将回顾 Scala 的更多语言功能（比如泛型，因为解析器组合子是由它们组成的）。Scala 还有许多东西需要学习，但希望您现在能够知道如何使用 Scala 解决那些用 Java 代码很难解决的问题。下次见！

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。

- “面向 Java 开发人员的 Scala 指南：面向对象的函数编程”（Ted Neward，developerWorks，2008 年 1 月）：本系列的第 1 篇文章概述了 Scala，并解释了它的函数性方法等。本系列中的其他文章：
  - “类操作”（2008 年 2 月）详述了 Scala 的类语法和语义。
  - “Scala 控制结构内部揭密”（2008 年 3 月）深入讨论了 Scala 的控制结构。
  - “关于特征和行为”（2008 年 4 月）：利用 Scala 版本的 Java 接口。
  - “实现继承”（2008 年 5 月）：Scala 式的多态。
  - “集合类型”（2008 年 6 月）包括所有“元组、数组和列表”。
  - “包和访问修饰符”（2008 年 7 月）讨论了 Scala 的包和访问修饰符，以及 `apply` 机制。
  - “构建计算器，第 1 部分”（2008 年 8 月）是本文的第一部分。
  - “构建计算器，第 2 部分”（2008 年 10 月）是本文的第二部分，即使用解析器组合子。

- "Java 语言中的函数编程"（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度解释函数性编程的优点和用法。
- "Scala by Example"（Martin Odersky，2007 年 12 月）：一篇简短的、含有大量代码的 Scala 入门介绍（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一次以一本书的篇幅介绍 Scala。
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases*（Addison-Wesley 专家，2005 年 7 月）：通过有趣的编程难题揭示 Java 编程语言的特点。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scala：通过这个系列开始学习 Scala。
- SUnit：标准 Scala 发行版的一部分，在 *scala.testing* 包中。

讨论

- 参与论坛讨论。
- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

◉太差！ (1)
○需提高 (2)
○一般；尚可 (3)
○好文章 (4)
○真棒！(5)

将您的建议发给我们或者通过参加讨论与其他人分享您的想法.

反馈意见

回页首

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- servlet 回顾
- Hello, Scala 与 Hello, Servlet
- Hello, Scala。这些是参数。
- 结束语
- 下载
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南:

Scala 和 servlet

将 **Scala** 与 **servlet** 结合使用

文档选项

打印本页

将此页作为电子邮件发送

样例代码

英文原文

级别： 初级

Ted Neward, 主管, Neward & Associates

2009 年 2 月 02 日

为了让一门语言适用于“现实”，并且使其“辉煌起来”，该语言必须能够服务于现实环境和应用程序。在这一期的 面向 *Java* 开发人员的 *Scala* 指南 系列中，Ted Neward 将介绍 Scala 在现实环境中的使用，即解释 Scala 如何与核心 Servlet API 交互，甚至可能会对其进行一些改正。

Scala 显然是一门有趣的语言，很适合体现语言理论和创新方面的新思想，但最终它要用在“现实”环境中，它必须能满足开发人员的某些需求并在“现实”环境中有一定的实用性。

了解 Scala 语言的一些核心功能之后，就能认识到 Scala 语言的一些灵活性，并能放心使用 Scala 创建 DSL。现在我们进入实际应用程序使用的环境，看看 Scala 如何适应环境。在本系列的新阶段中，我们将首先讨论大部分 Java™ 应用程序的核心：Servlet API。

servlet 回顾

回忆一下 Servlet 101 课程和教程，servlet 环境的核心实际上就是通过一个套接字（通常是端口 80）使用 HTTP 协议的客户机-服务器交换。客户机可以是任何“用户-代理”（由 HTTP 规范定义），服务器是一个 servlet 容器。servlet 容器在我编写的一个类上查找、加载和执行方法，该类最终必须实现 `javax.servlet.Servlet` 接口。

通常，实际的 Java 开发人员不会编写直接实现接口的类。因为最初的 servlet 规范是用于为 HTTP 之外的其他协议提供一个通用 API，所以 servlet 命名空间被分为了两部分：

- 一个“通用”包（`javax.servlet`）
- 一个特定于 HTTP 的包（`javax.servlet.http`）

这样，将在一个称为 `javax.servlet.GenericServlet` 的抽象基类的通用包中实现一些基本的功能；然后在派生类 `javax.servlet.http.HttpServlet` 中实现其他特定于 HTTP 的功能，该类通常用作 servlet 实际“内容”的基类。`HttpServlet` 提供了一个 `Servlet` 的完整实现，将 `GET` 请求委托给一个将要被覆盖的 `doGet` 方法，将 `POST` 请求委托给一个将要被覆盖的 `doPut` 方法，依此类推。

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个新的 developerWorks 系列 中，您将了解有关 Scala 的所有最新讨论，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但是（您将发现）Scala 与 Java 中的许多东西都没有直接的关联，这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

Hello, Scala 与 Hello, Servlet

显然，任何人编写的第一个 servlet 都是普遍的 “Hello, World” servlet；Scala 的第一个 servlet 示例也是如此。回忆一下许多年之前介绍的 servlet 教程，当时基本的 Java “Hello, World” servlet 只是输出清单 1 所示的 HTML 响应：

清单 **1.** 预期的 **HTML** 响应

```
<HTML>
   <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
   <BODY>Hello, Scala! This is a servlet.</BODY>
```

```
</HTML>
```

用 Scala 编写一个简单的 servlet 来实现这个操作非常简单，而且这个 servlet 与其相应的 Java 形式几乎一样，如清单 2 所示：

清单 **2. Hello, Scala servlet!**

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print("<HTML>" +
      "<HEAD><TITLE>Hello, Scala!</TITLE></HEAD>" +
      "<BODY>Hello, Scala! This is a servlet.</BODY>" +
      "</HTML>")
}
```

注意，我使用了一些适当的导入别名来缩短请求的类型名称和相应类型；除此之外，这个 servlet 几乎与其 Java servlet 形式一样。编译时请记得在 servlet-api.jar（通常随 servlet 容器一起发布；在 Tomcat 6.0 发行版中，它隐藏在 lib 子目录中）中包含一个引用，否则将找不到 servlet API 类型。

这还准备得不够充分；根据 servlet 规范，它必须使用一个 web.xml 部署描述符部署到 Web 应用程序目录中（或一个 .war 文件中），该描述符描述 servlet 应该与哪个 URL 结合。对于这样一个简单的例子，使用一个相当简单的 URL 来配合它最容易，如清单 3 所示：

清单 **3.** 部署描述符 **web.xml**

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>helloWorld</servlet-name>
    <servlet-class>HelloScalaServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>helloWorld</servlet-name>
    <url-pattern>/sayHello</url-pattern>
  </servlet-mapping>
</web-app>
```

从这里开始，我假设读者会在必要时调整/修改部署描述符，因为这跟 Scala 没有关系。

当然，格式良好的 HTML 与格式良好的 XML 非常相似；鉴于这一点，Scala 对 XML 字面值的支持使编写这个 servlet 简单得多（参阅 参考资料 中的 “Scala 和 XML” 一文）。Scala 不是在传递给 `HttpServletResponse` 的 String 中直接嵌入消息，它可以分离逻辑和表示形式（非常简单），方法是利用此支持将消息放在 XML 实例中，然后再传递回去：

清单 **4. Hello, Scala servlet!**

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! This is a servlet.</BODY>
    </HTML>

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

Scala 的内联表达式求值工具使用 XML 字面值，这意味着能够轻松地使 servlet 更有趣。例如，将当前日期添加到消息中与将 Calendar 表达式添加到 XML 中一样简单，不过增加了几行 `{ Text(java.util.Calendar.getInstance().getTime().toString() ) }`。这似乎显得有点冗长，如清单 5 所示：

清单 **5. Hello, timed Scala servlet!**

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! It's now { currentDate }</BODY>
    </HTML>
  def currentDate = java.util.Calendar.getInstance().getTime()

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

实际上，Scala 编译器与 XML 对象消息一起整合到一个 `scala.xml.Node` 中，然后在将它传递给响应的 `Writer` 的 `print` 方法时将其转换为一个 String。

不要小看这一点 — 表达形式从逻辑中分离出来完全在一个类内部进行。这条 XML 消息将进行编译时检查，以确保语法正确和格式良好，并获得一些标准 servlet（或 JSP）不具备的好处。由于 Scala 可以进行类型推断，因此可以省略有关 `message` 和 `currentDate` 的实际类型消息，使得这就像动态语言 Groovy/Grails 一样。初次使用效果不错。

当然，只读 servlet 相当麻烦。

Hello, Scala。这些是参数。

大多数 servlet 不会只返回类似静态内容或者当前日期和时间的简单消息，它们带有 `POST` 形式的内容，检查内容并进行相应的响应。例如，可能 Web 应

用程序需要知道使用者的身份，并询问其姓名：

清单 **6.** 挑战！

```
<HTML>
  <HEAD><TITLE>Who are you?</TITLE></HEAD>
  <BODY>
Who are you? Please answer:
<FORM action="/scalaExamples/sayMyName" method="POST">
Your first name: <INPUT type="text" name="firstName" />
Your last name: <INPUT type="text" name="lastName" />
<INPUT type="submit" />
</FORM>
  </BODY>
</HTML>
```

这个方法不会在任何用户界面设计大赛中夺冠，但是它达到了目的：这是一个 HTML 表单，它会将数据发送给一个新的 Scala servlet（绑定到 sayMyName 的相对 URL）。这些数据将根据 servlet 规范存储在一个名称-值对集合中，可通过 `HttpServletRequest.getParameter()` API 调用获得。在此调用中，我们将 `FORM` 元素的名称作为一个参数传递给 API 调用。

从 Java 代码直接转换相对容易一些，如清单 7 中的 servlet 所示：

清单 **7.** 响应（**v1**）

```
class NamedHelloWorldServlet1 extends HttpServlet
{
  def message(firstName : String, lastName : String) =
    <HTML>
      <HEAD><TITLE>Hello, {firstName} {lastName}!</TITLE></HEAD>
      <BODY>Hello, {firstName} {lastName}! It is now {currentTime}.</BODY>
    </HTML>
  def currentTime =
    java.util.Calendar.getInstance().getTime()

  override def doPost(req : HSReq, resp : HSResp) =
  {
    val firstName = req.getParameter("firstName")
    val lastName = req.getParameter("lastName")

    resp.getWriter().print(message(firstName, lastName))
  }
}
```

但这缺少了我之前提到的消息分离的一些好处，因为现在消息定义必须显式使用参数 `firstName` 和 `lastName`；如果响应 get 中使用的元素个数超过 3 个或 4 个，情况就会变得比较复杂。此外，`doPost` 方法在将参数传递给消息进行显示之前，必须自行提取每一个参数 — 这样的编码很繁琐，并且容易出错。

一种解决方法是将参数提取和 `doPost` 方法本身的调用添加到一个基类，如清单 8 中的版本 2 所示：

清单 **8.** 响应（**v2**）

```
abstract class BaseServlet extends HttpServlet
{
  import scala.collection.mutable.{Map => MMap}

  def message : scala.xml.Node;

  protected var param : Map[String, String] = Map.empty
  protected var header : Map[String, String] = Map.empty

  override def doPost(req : HSReq, resp : HSResp) =
  {
    // Extract parameters
    //
    val m = MMap[String, String]()
    val e = req.getParameterNames()
    while (e.hasMoreElements())
    {
      val name = e.nextElement().asInstanceOf[String]
      m += (name -> req.getParameter(name))
    }
    param = Map.empty ++ m

    // Repeat for headers (not shown)
    //

    resp.getWriter().print(message)
  }
}
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
    <HTML>
      <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!</TITLE></HEAD>
      <BODY>Hello, {param("firstName")} {param("lastName")}! It is now {currentTime}.
      </BODY>
    </HTML>

  def currentTime = java.util.Calendar.getInstance().getTime()
}
```

这个版本使 servlet 显示变得比较简单（相对上一版本而言），而且增加了一个优点，即 `param` 和 `header` 映射保持不变（注意，我们可以将 `param` 定义为一个引用请求对象的方法，但这个请求对象必须已经定义为一个字段，这将引发大规模的并发性问题，因为 servlet 容器认为每一个 `do` 方法都是可重入的）。

当然，错误处理是处理 Web 应用程序 `FORM` 的重要部分，而 Scala 作为一种函数性语言，保存的内容都是表达式，这意味着我们可以将消息编写为结果页面（假设我们喜欢这个输入），或编写为错误页面（如果我们不喜欢这个输入）。因此，检查 `firstName` 和 `lastName` 的非空状态的验证函数可能如清单 9 所示：

清单 **9.** 响应（**v3**）

```
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
    if (validate(param))
          <HTML>
                <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!
             </TITLE></HEAD>
                <BODY>Hello, {param("firstName")} {param("lastName")}!
                It is now {currentTime}.</BODY>
          </HTML>
    else
      <HTML>
        <HEAD><TITLE>Error!</TITLE></HEAD>
        <BODY>How can we be friends if you don't tell me your name?!?</BODY>
      </HTML>

  def validate(p : Map[String, String]) : Boolean =
  {
    p foreach {
      case ("firstName", "") => return false
      case ("lastName", "") => return false
          //case ("lastName", v) => if (v.contains("e")) return false
          case (_, _) => ()
    }
    true
  }

  def currentTime = java.util.Calendar.getInstance().getTime()
}
```

注意，模式匹配可以使编写比较简单的验证规则变得很容易。利用模式匹配绑定到原始值（比如上一个例子），或者绑定到一个本地变量（比如我们要排除任何姓名中有 “e” 的人，比如上一个注释）。

显然，还有事情需要做！困扰 Web 应用程序的典型问题之一是 SQL 注入攻击，它由通过 `FORM` 传入的未转义 SQL 命令字符引入，并且在数据库中执行之前连接到包含 SQL 结构的原始字符串。使用 scala.regex 包中的正则表达式支持，或者一些解析器组合子（在本系列最后三篇文章中讨论）可以确认 `FORM` 验证是否正确。事实上，整个验证过程会提交给使用默认验证实现的基类，该验证实现默认情况下只返回 true（因为 Scala 是函数性语言，所以不要忽略好的对象设计方法）。

结束语

虽然 Scala servlet 框架的功能不像其他一些 Java Web 框架的那样完整，但是我这里创建的这个小 Scala servlet 有两个基本用途：

- 展示以有趣的方式利用 Scala 的功能，使 JVM 编程更简单。
- 简单介绍将 Scala 用于 Web 应用程序，这自然会引入 “lift” 框架（参见 参考资料 小节）。

本期到此结束，我们下一期再见！

回页首

下载

描述	名字	大小	下载方法
本文的样例 Scala 代码	j-scala12238.zip	179KB	**HTTP**

➔ 关于下载方法的信息

参考资料

学习

- 您可以参阅本文在 developerWorks 全球网站上的 英文原文。
- "面向 Java 开发人员的 Scala 指南：面向对象的函数编程"（Ted Neward，developerWorks，2008 年 1 月）：本系列的第 1 篇文章概述了 Scala，并解释了它的函数性方法等。本系列的其他文章：
  - "类操作"（2008 年 2 月）详述了 Scala 的语法和语义。
  - "Scala 控制结构内部揭密"（2008 年 3 月）深入讨论了 Scala 的控制结构。
  - "关于特征和行为"（2008 年 4 月）：利用 Scala 版本的 Java 接口。
  - "实现继承"（2008 年 5 月）：Scala 式的多态。
  - "集合类型"（2008 年 6 月）包括所有"元组、数组和列表"。
  - "包和访问修饰符"（2008 年 7 月）讨论了 Scala 的包和访问修饰符，以及 `apply` 机制。
  - "构建计算器，第 1 部分"（2008 年 8 月）是本文的第一部分。
  - "构建计算器，第 2 部分"（2008 年 10 月）是本文的第二部分，即使用解析器组合子。
  - "构建计算器，第 3 部分"（2008 年 11 月）是本文的第三部分，说明如何将所有对象联合成一个无缝整体（并推荐了一些用于语言和解释器的扩展）。
- "Scala 和 XML"（developerWorks，2008 年 4 月）说明如何使用 Scala 导航和处理解析的 XML，并详述了对内置到该语言中的 XML 的良好支持。
- wiki on the Scala lift framework：这是用于编写 Web 应用程序的框架，它演示了框架的著名理论（Seaside、Rails、Django 和 Wicket 等）。这是一本 入门教程。
- "Java 语言中的函数编程"（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度解释函数性编程的优点和用法。
- "Scala by Example"（Martin Odersky，2007 年 12 月）：一篇简短的、含有大量代码的 Scala 入门介绍（PDF 格式）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：第一次以一本书的篇幅介绍 Scala。
- Bjarne Stroustrup：设计和实现的 C++，他将它称为"更好的 C"。
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases*（Addison-Wesley 专家，2005 年 7 月）：通过有趣的编程难题揭示 Java 编程语言的特点。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scala：通过这个系列开始学习 Scala。

- SUnit：标准 Scala 发行版的一部分，在 *scala.testing* 包中。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 Neward & Associates 的主管，负责有关 Java、.NET、XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿州西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- 并发性基础
- 良好的 Scala 并发性 (v1)
- Scala 并发性 v2
- Scala 并发性 v3
- 结束语
- 下载
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南: 深入了解 Scala 并发性

了解 **Scala** 如何简化并发编程并绕过陷阱

文档选项

打印本页

将此页作为电子邮件发送

样例代码

英文原文

级别： 初级

Ted Neward, 主管, ThoughtWorks, Neward & Associates

2009 年 7 月 10 日

对于许多（如果不是大多数）Java™ 程序员来说，Scala 的吸引力在于处理并发性以及编写线程安全的代码时非常轻松。在本期文章中，Ted Neward 将开始深入研究 Scala 语言及环境所提供的各种并发特性和库。

2003 年，Herb Sutter 在他的文章 “The Free Lunch Is Over” 中揭露了行业中最不可告人的一个小秘密，他明确论证了处理器在速度上的发展已经走到了尽头，并且将由全新的单芯片上的并行 “内核”（虚拟 CPU）所取代。这一发现对编程社区造成了不小的冲击，因为正确创建线程安全的代码，在理论而非实践中，始终会提高高性能开发人员的身价，而让各公司难以聘用他们。看上去，仅有少数人充分理解了 Java 的线程模型、并发 API 以及 “同步” 的含义，以便能够编写同时提供安全性和吞吐量的代码 —— 并且大多数人已经明白了它的困难所在。

据推测，行业的其余部分将自力更生，这显然不是一个理想的结局，至少不是 IT 部门努力开发软件所应得的回报。

与 Scala 在 .NET 领域中的姐妹语言 F# 相似，Scala 是针对 “并发性问题” 的解决方案之一。在本期文章中，我讨论了 Scala 的一些属性，这些属性使它更加胜任于编写线程安全的代码，比如默认不可修改的对象，并讨论了一种返回对象副本而不是修改它们内容的首选设计方案。Scala 对并发性的支持远比此深远；现在，我们有必要来了解一下 Scala 的各种库。

关于本系列

关于本系列 Ted Neward 潜心研究 Scala 编程语言，并带您跟他一起徜徉。在这个新的 developerWorks 系列 中，您将深入了解 Scala 并看到 Scala 的语言功能的实际效果。在进行相关比较时，Scala 代码和 Java 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！毕竟，如果 Java 代码可以做到的话，又何必学习 Scala 呢？

并发性基础

在深入研究 Scala 的并发性支持之前，有必要确保您具备了对 Java 基本并发性模型的良好理解，因为 Scala 的并发性支持，从某种程度上说，建立在 JVM 和支持库所提供的特性和功能的基础之上。为此，清单 1 中的代码包含了一个已知的 *Producer/Consumer* 并发性问题（详见 Sun Java Tutorial 的 “Guarded Blocks” 小节）。注意，Java Tutorial 版本并未在其解决方案中使用 `java.util.concurrent` 类，而是择优使用了 `java.lang.Object` 中的较旧的 `wait()/notifyAll()` 方法：

清单 **1. Producer/Consumer**（**Java5** 之前）

```
package com.tedneward.scalaexamples.notj5;

class Producer implements Runnable
{
  private Drop drop;
  private String importantInfo[] = {
    "Mares eat oats",
    "Does eat oats",
    "Little lambs eat ivy",
    "A kid will eat ivy too"
```

```
  };

  public Producer(Drop drop) { this.drop = drop; }

  public void run()
  {
    for (int i = 0; i < importantInfo.length; i++)
    {
      drop.put(importantInfo[i]);
    }
    drop.put("DONE");
  }
}

class Consumer implements Runnable
{
  private Drop drop;

  public Consumer(Drop drop) { this.drop = drop; }

  public void run()
  {
    for (String message = drop.take(); !message.equals("DONE");
         message = drop.take())
    {
      System.out.format("MESSAGE RECEIVED: %s%n", message);
    }
  }
}

class Drop
{
  //Message sent from producer to consumer.
  private String message;

  //True if consumer should wait for producer to send message,
  //false if producer should wait for consumer to retrieve message.
  private boolean empty = true;

  //Object to use to synchronize against so as to not "leak" the
  //"this" monitor
  private Object lock = new Object();

  public String take()
  {
    synchronized(lock)
    {
      //Wait until message is available.
      while (empty)
      {
        try
```

```
        {
          lock.wait();
        }
        catch (InterruptedException e) {}
      }
      //Toggle status.
      empty = true;
      //Notify producer that status has changed.
      lock.notifyAll();
      return message;
    }
  }

  public void put(String message)
  {
    synchronized(lock)
    {
      //Wait until message has been retrieved.
      while (!empty)
      {
        try
        {
          lock.wait();
        } catch (InterruptedException e) {}
      }
      //Toggle status.
      empty = false;
      //Store message.
      this.message = message;
      //Notify consumer that status has changed.
      lock.notifyAll();
    }
  }
}

public class ProdConSample
{
  public static void main(String[] args)
  {
    Drop drop = new Drop();
    (new Thread(new Producer(drop))).start();
    (new Thread(new Consumer(drop))).start();
  }
}
```

注意：我在此处展示的代码对 Sun 教程解决方案做了少许修改；它们提供的代码存在一个很小的设计缺陷（参见 Java 教程“缺陷”）。

Producer/Consumer 问题的核心非常容易理解：一个（或多个）生产者实体希望将数据提供给一个（或多个）使用者实体供它们使用和操作（在本例中，它包括将数据打印到控制台）。`Producer` 和 `Consumer` 类是相应直观的 `Runnable`-实现类：`Producer` 从数组中

Java 教程“缺陷”

好奇的读者可能会将此处的代码与 Java Tutorial 中的代码进行比较，寻找它们之间有哪些不同；他们会发现我并未“同步”`put` 和 `take` 方法，而是使用了存储在 `Drop` 中的 *lock* 对象。其原因非常简单：对象的监测程

获取 String，并通过 put 将它们放置到 Consumer 的缓冲区中，并根据需要执行 take。

问题的难点在于，如果 Producer 运行过快，则数据在覆盖时可能会丢失；如果 Consumer 运行过快，则当 Consumer 读取相同的数据两次时，数据可能会得到重复处理。缓冲区（在 Java Tutorial 代码中称作 Drop）将确保不会出现这两种情况。数据破坏的可能性就更不用提了（在 String 引用的例子中很困难，但仍然值得注意），因为数据会由 put 放入缓冲区，并由 take 取出。

关于此主题的全面讨论请阅读 Brian Goetz 的 *Java Concurrency in Practice* 或 Doug Lea 的 *Concurrent Programming in Java*（参见 参考资料），但是，在应用 Scala 之前有必要快速了解一下此代码的运行原理。

当 Java 编译器看到 synchronized 关键字时，它会在同步块的位置生成一个 try/finally 块，其顶部包括一个 monitorenter 操作码，并且 finally 块中包括一个 monitorexit 操作码，以确保监控程序（Java 的原子性基础）已经发布，而与代码退出的方式无关。因此，Drop 中的 put 代码将被重写，如清单 2 所示：

清单 **2.** 编译器失效后的 **Drop.put**

```
// This is pseudocode
public void put(String message)
{
  try
  {
        monitorenter(lock)

    //Wait until message has been retrieved.
    while (!empty)
    {
      try
      {
        lock.wait();
      } catch (InterruptedException e) {}
    }
    //Toggle status.
    empty = false;
    //Store message.
    this.message = message;
    //Notify consumer that status has changed.
    lock.notifyAll();
  }
      finally
      {
        monitorexit(lock)
      }
}
```

wait() 方法将通知当前线程进入非活动状态，并等待另一个线对该对象调用 notifyAll()。然后，通知的线程必须在能够继续执行的时候尝试再次获取监控程序。从本质上说，wait() 和 notify()/notifyAll() 允许一种简单的信令机制，它允许 Drop 在 Producer 和 Consumer 线程之间进行协调，每个 put 都有相应的 take。

序永远都不会封装在类的内部，因此 Java Tutorial 版本允许此代码打破此规则（显然很疯狂）：

```
public class
ProdConSample
{
  public static void
main(String[] args)
  {
    Drop drop = new
Drop();
    (new Thread(new
Producer(drop))).start();
    (new Thread(new
Consumer(drop))).start();

synchronized(drop)
        {

Thread.sleep(60 * 60 *
24 * 365 * 10); // sleep
for 10 years?!?
        }
  }
}
```

通过使用私有对象作为锁定所依托的监测程序，此代码将不会有任何效果。从本质上说，现在已经封装了线程安全的实现；然后，它才能依赖客户机的优势正常运行。

本文的 代码下载 部分使用 Java5 并发性增强（Lock 和 Condition 接口以及 ReentrantLock 锁定实现）提供 清单 2 的基于超时的版本，但基本代码模式仍然相同。这就是问题所在：编写清单 2 这样的代码的开发人员需要过度专注于线程和锁定的细节以及低级实现代码，以便让它们能够正确运行。此外，开发人员需要对每一行代码刨根知底，以确定是否需要保护它们，因为过度同步与过少同步同样有害。

现在，我们来看到 Scala 替代方案。

良好的 Scala 并发性 (v1)

开始应用 Scala 并发性的一种方法是将 Java 代码直接转换为 Scala，以便利用 Scala 的语法优势来简化代码（至少能简化一点）：

清单 **3. ProdConSample (Scala)**

```
object ProdConSample
{
  class Producer(drop : Drop)
    extends Runnable
  {
    val importantInfo : Array[String] = Array(
      "Mares eat oats",
      "Does eat oats",
      "Little lambs eat ivy",
      "A kid will eat ivy too"
    );

    override def run() : Unit =
    {
      importantInfo.foreach((msg) => drop.put(msg))
      drop.put("DONE")
    }
  }

  class Consumer(drop : Drop)
    extends Runnable
  {
    override def run() : Unit =
    {
      var message = drop.take()
      while (message != "DONE")
      {
        System.out.format("MESSAGE RECEIVED: %s%n", message)
        message = drop.take()
      }
    }
  }

  class Drop
  {
    var message : String = ""
    var empty : Boolean = true
    var lock : AnyRef = new Object()
```

```
    def put(x: String) : Unit =
      lock.synchronized
      {
        // Wait until message has been retrieved
        await (empty == true)
        // Toggle status
        empty = false
        // Store message
        message = x
        // Notify consumer that status has changed
        lock.notifyAll()
      }

    def take() : String =
      lock.synchronized
      {
        // Wait until message is available.
        await (empty == false)
        // Toggle status
        empty=true
        // Notify producer that staus has changed
        lock.notifyAll()
        // Return the message
        message
      }

    private def await(cond: => Boolean) =
      while (!cond) { lock.wait() }
  }

  def main(args : Array[String]) : Unit =
  {
    // Create Drop
    val drop = new Drop();

    // Spawn Producer
    new Thread(new Producer(drop)).start();

    // Spawn Consumer
    new Thread(new Consumer(drop)).start();
  }
}
```

`Producer` 和 `Consumer` 类几乎与它们的 Java 同类相同，再一次扩展（实现）了 `Runnable` 接口并覆盖了 `run()` 方法，并且 — 对于 `Producer` 的情况 — 分别使用了内置迭代方法来遍历 `importantInfo` 数组的内容。（实际上，为了让它更像 Scala，`importantInfo` 可能应该是一个 `List` 而不是 `Array`，但在第一次尝试时，我希望尽可能保证它们与原始 Java 代码一致。）

`Drop` 类同样类似于它的 Java 版本。但 Scala 中有一些例外，“synchronized” 并不是关键字，它是针对 `AnyRef` 类定义的一个方法，即 Scala “所有引用类型的根”。这意味着，要同步某个特定的对象，您只需要对该对象调用同步方法；在本例中，对 `Drop` 上的 lock 字段中所保存的对象调用同步方法。

注意，我们在 `await()` 方法定义的 `Drop` 类中还利用了一种 Scala 机制：`cond` 参数是等待计算的代码块，而不是在传递给该方法之前进行计算。在

Scala 中，这被称作 “call-by-name”；此处，它是一种实用的方法，可以捕获需要在 Java 版本中表示两次的条件等待逻辑（分别用于 put 和 take）。

最后，在 main() 中，创建 Drop 实例，实例化两个线程，使用 start() 启动它们，然后在 main() 的结束部分退出，相信 JVM 会在 main() 结束之前启动这两个线程。（在生产代码中，可能无法保证这种情况，但对于这样的简单的例子，99.99% 没有问题。）

但是，已经说过，仍然存在相同的基本问题：程序员仍然需要过分担心两个线程之间的通信和协调问题。虽然一些 Scala 机制可以简化语法，但这目前为止并没有相当大的吸引力。

Scala 并发性 v2

Scala Library Reference 中有一个有趣的包：scala.concurrency。这个包包含许多不同的并发性结构，包括我们即将利用的 MailBox 类。

顾名思义，MailBox 从本质上说就是 Drop，用于在检测之前保存数据块的单槽缓冲区。但是，MailBox 最大的优势在于它将发送和接收数据的细节完全封装到模式匹配和 case 类中，这使它比简单的 Drop（或 Drop 的多槽数据保存类 java.util.concurrent.BoundedBuffer）更加灵活。

清单 **4. ProdConSample, v2 (Scala)**

```
package com.tedneward.scalaexamples.scala.V2
{
  import concurrent.{MailBox, ops}

  object ProdConSample
  {
    class Producer(drop : Drop)
      extends Runnable
    {
      val importantInfo : Array[String] = Array(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "A kid will eat ivy too"
      );

      override def run() : Unit =
      {
        importantInfo.foreach((msg) => drop.put(msg))
        drop.put("DONE")
      }
    }

    class Consumer(drop : Drop)
      extends Runnable
    {
      override def run() : Unit =
      {
        var message = drop.take()
        while (message != "DONE")
        {
          System.out.format("MESSAGE RECEIVED: %s%n", message)
          message = drop.take()
        }
      }
```

```
    }

    class Drop
    {
      private val m = new MailBox()

      private case class Empty()
      private case class Full(x : String)

      m send Empty()  // initialization

      def put(msg : String) : Unit =
      {
        m receive
        {
          case Empty() =>
            m send Full(msg)
        }
      }

      def take() : String =
      {
        m receive
        {
          case Full(msg) =>
            m send Empty(); msg
        }
      }
    }

    def main(args : Array[String]) : Unit =
    {
      // Create Drop
      val drop = new Drop()

      // Spawn Producer
      new Thread(new Producer(drop)).start();

      // Spawn Consumer
      new Thread(new Consumer(drop)).start();
    }
  }
}
```

此处，v2 和 v1 之间的惟一区别在于 Drop 的实现，它现在利用 MailBox 类处理传入以及从 Drop 中删除的消息的阻塞和信号事务。（我们可以重写 Producer 和 Consumer，让它们直接使用 MailBox，但考虑到简单性，我们假定希望保持所有示例中的 Drop API 相一致。）使用 MailBox 与使用典型的 BoundedBuffer（Drop）稍有不同，因此我们来仔细看看其代码。

MailBox 有两个基本操作：send 和 receive。receiveWithin 方法仅仅是基于超时的 receive。MailBox 接收任何类型的消息。send() 方法将消息放置到邮箱中，并立即通知任何关心该类型消息的等待接收者，并将它附加到一个消息链表中以便稍后检索。receive() 方法将阻塞，直到接收到对于功能块合适的消息。

因此，在这种情况下，我们将创建两个 case 类，一个不包含任何内容（Empty），这表示 MailBox 为空，另一个包含消息数据（Full。

- put 方法，由于它会将数据放置在 Drop 中，对 MailBox 调用 receive() 以查找 Empty 实例，因此会阻塞直到发送 Empty。此时，它发送一个 Full 实例给包含新数据的 MailBox。
- take 方法，由于它会从 Drop 中删除数据，对 MailBox 调用 receive() 以查找 Full 实例，提取消息（再次得益于模式匹配从 case 类内部提取值并将它们绑到本地变量的能力）并发送一个 Empty 实例给 MailBox。

不需要明确的锁定，并且不需要考虑监控程序。

Scala 并发性 v3

事实上，我们可以显著缩短代码，只要 Producer 和 Consumer 不需要功能全面的类（此处便是如此） — 两者从本质上说都是 Runnable.run() 方法的瘦包装器，Scala 可以使用 scala.concurrent.ops 对象的 spawn 方法来实现，如清单 5 所示：

清单 **5. ProdConSample, v3 (Scala)**

```
package com.tedneward.scalaexamples.scala.V3
{
  import concurrent.MailBox
  import concurrent.ops._

  object ProdConSample
  {
    class Drop
    {
      private val m = new MailBox()

      private case class Empty()
      private case class Full(x : String)

      m send Empty()  // initialization

      def put(msg : String) : Unit =
      {
        m receive
        {
          case Empty() =>
            m send Full(msg)
        }
      }

      def take() : String =
      {
        m receive
        {
          case Full(msg) =>
            m send Empty(); msg
        }
      }
    }
```

```
    def main(args : Array[String]) : Unit =
    {
      // Create Drop
      val drop = new Drop()

      // Spawn Producer
      spawn
      {
        val importantInfo : Array[String] = Array(
          "Mares eat oats",
          "Does eat oats",
          "Little lambs eat ivy",
          "A kid will eat ivy too"
        );

        importantInfo.foreach((msg) => drop.put(msg))
        drop.put("DONE")
      }

      // Spawn Consumer
      spawn
      {
        var message = drop.take()
        while (message != "DONE")
        {
          System.out.format("MESSAGE RECEIVED: %s%n", message)
          message = drop.take()
        }
      }
    }
  }
}
```

spawn 方法（通过包块顶部的 ops 对象导入）接收一个代码块（另一个 by-name 参数示例）并将它包装在匿名构造的线程对象的 run() 方法内部。事实上，并不难理解 spawn 的定义在 ops 类的内部是什么样的：

清单 **6. scala.concurrent.ops.spawn()**

```
  def spawn(p: => Unit) = {
    val t = new Thread() { override def run() = p }
    t.start()
  }
```

……这再一次强调了 by-name 参数的强大之处。

ops.spawn 方法的一个缺点在于，它是在 2003 年 Java 5 concurrency 类还不可用的时候编写的。特别是，java.util.concurrent.Executor 及其同类的作用是让开发人员更加轻松地生成线程，而不需要实际处理直接创建线程对象的细节。幸运的是，在您自己的自定义库中重新创建 spawn 的定义是相当简单的，这需要利用 Executor（或 ExecutorService 或 ScheduledExecutorService）来执行线程的实际启动任务。

事实上，Scala 的并发性支持超越了 MailBox 和 ops 类；Scala 还支持一个类似的 “Actors” 概念，它使用了与 MailBox 所采用的方法相类似的消息传

递方法，但应用更加全面并且灵活性也更好。但是，这部分内容将在下期讨论。

结束语

Scala 为并发性提供了两种级别的支持，这与其他与 Java 相关的主题极为类似：

- 首先，对底层库的完全访问（比如说 java.util.concurrent）以及对“传统” Java 并发性语义的支持（比如说监控程序和 `wait()/notifyAll()`）。
- 其次，这些基本机制上面有一个抽象层，详见本文所讨论的 `MailBox` 类以及将在本系列下一篇文章中讨论的 Actors 库。

两个例子中的目标是相同的：让开发人员能够更加轻松地专注于问题的实质，而不用考虑并发编程的低级细节（显然，第二种方法更好地实现了这一目标，至少对于没有过多考虑低级细节的人来说是这样的。）

但是，当前 Scala 库的一个明显的缺陷就是缺乏 Java 5 支持；`scala.concurrent.ops` 类应该具有 `spawn` 这样的利用新的 `Executor` 接口的方法。它还应该支持利用新的 `Lock` 接口的各种版本的 `synchronized`。幸运的是，这些都是可以在 Scala 生命周期中实现的库增强，而不会破坏已有代码；它们甚至可以由 Scala 开发人员自己完成，而不需要等待 Scala 的核心开发团队提供给他们（只需要花费少量时间）。

回页首

下载

描述	名字	大小	下载方法
本文的示例 Scala 代码	j-scala02049.zip	10KB	**HTTP**

关于下载方法的信息

参考资料

学习

- 面向 *Java* 开发人员的 *Scala* 指南（Ted Neward，developerWorks）：阅读本系列的所有文章。
- *Java Concurrency in Practice*（Brian Goetz，Addison-Wesley Professional，2006 年）全面讨论了并发性的相关主题。
- *Concurrent Programming in Java*（Doug Lea，Prentice Hall PTR，1999 年）讨论了并行性和并发性的许多研究领域，通过大量模式和设计技巧介绍如何在 Java 中更好地利用多线程。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- Programming in Scala（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：这是第一本介绍 Scala 的图书。
- Bjarne Stroustrup：他设计和实现了 C++，他把 C++ 称为“更好的 C”。
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases*（Addison-Wesley Professional，2005 年 7 月）通过有趣和发人深省的编程难题介绍 Java 编程语言的特点。
- The developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：跟随本系列学习 Scala！
- SUnit：SUnit 是标准 Scala 发行版的一部分，在 *scala.testing* 包中提供。

讨论

- 通过参与 developerWorks 博客 加入 developerWorks 社区。

关于作者

Ted Neward 是 ThoughtWorks 的一名顾问，ThoughtWorks 是一家在全球提供咨询服务的公司，他还是 Neward & Associates 的主管，负责有关 Java、.NET 和 XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿西雅图附近。

对本文的评价

- ◉ 太差！ (1)
- ○ 需提高 (2)
- ○ 一般；尚可 (3)
- ○ 好文章 (4)
- ○ 真棒！(5)

建议？

反馈意见

回页首

developerWorks 中国

本系列的更多信息:
面向 Java 开发人员的 Scala 指南

本文内容包括:

- 什么是 “actor”?
- Scala actor
- 并发地执行动作
- 结束语
- 参考资料
- 关于作者
- 对本文的评价

相关链接:

- Java technology 技术文档库

面向 Java 开发人员的 Scala 指南: 深入了解 Scala 并发性

了解 **actor** 如何提供新的应用程序代码建模方法

文档选项

打印本页

将此页作为电子邮件发送

英文原文

级别: 初级

Ted Neward, 主管, ThoughtWorks, Neward & Associates

2009 年 8 月 13 日

主要芯片厂商已经开始提供同时运行两个或更多个核的芯片（虽然不一定更快），在这种情况下，并发性很快成为每个软件开发人员都关心的热门主题。本文延续 Ted Neward 的另一篇文章 深入了解 Scala 并发性。在本文中，Ted Neward 通过研究 *actor* 深入讨论并发性这个热门主题，actor 是通过传递消息相互协作的执行实体。

关于本系列

Ted Neward 潜心研究 Scala 编程语言，并带您跟他一起徜徉。在这个新的 developerWorks 系列 中，您将深入了解 Scala 并看到 Scala 的语言功能的实际效果。在进行相关比较时，Scala 代码和 Java™ 代码将放在一起展示，但（您将发现）Scala 中的许多内容与您在 Java 编程中发现的任何内容都没有直接关联，而这正是 Scala 的魅力所在！毕竟，如果 Java 代码可以做到的话，又何必学习 Scala 呢?

在 前一篇文章 中，我讨论了构建并发代码的重要性（无论是否是 Scala 代码），还讨论了在编写并发代码时开发人员面对的一些问题，包括不要锁住太多东西、不要锁住太少东西、避免死锁、避免生成太多线程等等。

这些理论问题太沉闷了。为了避免读者觉得失望，我与您一起研究了 Scala 的一些并发构造，首先是在 Scala 中直接使用 Java 语言的并发库的基本方法，然后讨论 Scala API 中的 `MailBox` 类型。尽管这两种方法都是可行的，但是它们并不是 Scala 实现并发性的主要机制。

真正提供并发性的是 Scala 的 *actor*。

什么是 “actor”?

“actor” 实现在称为 *actor* 的执行实体之间使用消息传递进行协作（注意，这里有意避免使用 “进程”、“线程” 或 “机器” 等词汇）。尽管它听起来与 RPC 机制有点儿相似，但是它们是有区别的。RPC 调用（比如 Java RMI 调用）会在调用者端阻塞，直到服务器端完成处理并发送回某种响应（返回值或异常），而消息传递方法不会阻塞调用者，因此可以巧妙地避免死锁。

仅仅传递消息并不能避免错误的并发代码的所有问题。另外，这种方法还有助于使用 “不共享任何东西” 编程风格，也就是说不同的 actor 并不访问共享的数据结构（这有助于促进封装 actor，无论 actor 是 JVM 本地的，还是位于其他地方） — 这样就完全不需要同步了。毕竟，如果不共享任何东西，并发执行就不涉及任何需要同步的东西。

这不算是对 actor 模型的正规描述，而且毫无疑问，具有更正规的计算机科学背景的人会找到各种更严谨的描述方法，能够描述 actor 的所有细节。但是对于本文来说，这个描述已经够了。在网上可以找到更详细更正规的描述，还有一些学术文章详细讨论了 actor 背后的概念（请您自己决定是否要深入学习这些概念）。现在，我们来看看 Scala actors API。

Scala actor

使用 actor 根本不困难，只需使用 `Actor` 类的 `actor` 方法创建一个 actor，见清单 1:

清单 **1.** 开拍!

```
import scala.actors._, Actor._

package com.tedneward.scalaexamples.scala.V4
```

```
{
  object Actor1
  {
    def main(args : Array[String]) =
    {
      val badActor =
        actor
        {
          receive
          {
            case msg => System.out.println(msg)
          }
        }

      badActor ! "Do ya feel lucky, punk?"
    }
  }
}
```

这里同时做了两件事。

首先，我们从 Scala Actors 库的包中导入了这个库，然后从库中直接导入了 `Actor` 类的成员；第二步并不是完全必要的，因为在后面的代码中可以使用 `Actor.actor` 替代 `actor`，但是这么做能够表明 `actor` 是语言的内置结构并（在一定程度上）提高代码的可读性。

下一步是使用 `actor` 方法创建 actor 本身，这个方法通过参数接收一个代码块。在这里，代码块执行一个简单的 `receive`（稍后讨论）。结果是一个 actor，它被存储在一个值引用中，供以后使用。

请记住，除了消息之外，actor 不使用其他通信方法。使用 `!` 的代码行实际上是一个向 `badActor` 发送消息的方法，这可能不太直观。`Actor` 内部还包含另一个 `MailBox` 元素（已讨论）；`!` 方法接收传递过来的参数（在这里是一个字符串），把它发送给邮箱，然后立即返回。

消息交付给 actor 之后，actor 通过调用它的 `receive` 方法来处理消息；这个方法从邮箱中取出第一个可用的消息，把它交付给一个模式匹配块。注意，因为这里没有指定模式匹配的类型，所以任何消息都是匹配的，而且消息被绑定到 `msg` 名称（为了打印它）。

一定要注意一点：对于可以发送的类型，没有任何限制 — 不一定要像前面的示例那样发送字符串。实际上，基于 actor 的设计常常使用 Scala case 类携带实际消息本身，这样就可以根据 case 类的参数/成员的类型提供隐式的“命令”或“动作”，或者向动作提供数据。

例如，假设希望 actor 用两个不同的动作来响应发送的消息；新的实现可能与清单 2 相似：

清单 **2.** 嗨，我是导演！

```
  object Actor2
  {
    case class Speak(line : String);
    case class Gesture(bodyPart : String, action : String);
    case class NegotiateNewContract;

    def main(args : Array[String]) =
    {
      val badActor =
        actor
        {
```

```
        receive
        {
          case NegotiateNewContract =>
            System.out.println("I won't do it for less than $1 million!")
          case Speak(line) =>
            System.out.println(line)
          case Gesture(bodyPart, action) =>
            System.out.println("(" + action + "s " + bodyPart + ")")
          case _ =>
            System.out.println("Huh? I'll be in my trailer.")
        }
      }

    badActor ! NegotiateNewContract
    badActor ! Speak("Do ya feel lucky, punk?")
    badActor ! Gesture("face", "grimaces")
    badActor ! Speak("Well, do ya?")
  }
}
```

到目前为止，看起来似乎没问题，但是在运行时，只协商了新合同；在此之后，JVM 终止了。初看上去，似乎是生成的线程无法足够快地响应消息，但是要记住在 actor 模型中并不处理线程，只处理消息传递。这里的问题其实非常简单：一次接收使用一个消息，所以无论队列中有多少个消息正在等待处理都无所谓，因为只有一次接收，所以只交付一个消息。

纠正这个问题需要对代码做以下修改，见清单 3：

- 把 `receive` 块放在一个接近无限的循环中。
- 创建一个新的 case 类来表示什么时候处理全部完成了。

清单 **3.** 现在我是一个更好的导演！

```
object Actor2
{
  case class Speak(line : String);
  case class Gesture(bodyPart : String, action : String);
  case class NegotiateNewContract;
  case class ThatsAWrap;

  def main(args : Array[String]) =
  {
    val badActor =
      actor
      {
        var done = false
        while (! done)
        {
          receive
          {
            case NegotiateNewContract =>
              System.out.println("I won't do it for less than $1 million!")
            case Speak(line) =>
```

```
              System.out.println(line)
            case Gesture(bodyPart, action) =>
              System.out.println("(" + action + "s " + bodyPart + ")")
            case ThatsAWrap =>
              System.out.println("Great cast party, everybody! See ya!")
              done = true
            case _ =>
              System.out.println("Huh? I'll be in my trailer.")
          }
        }
      }

    badActor ! NegotiateNewContract
    badActor ! Speak("Do ya feel lucky, punk?")
    badActor ! Gesture("face", "grimaces")
    badActor ! Speak("Well, do ya?")
    badActor ! ThatsAWrap
  }
}
```

这下行了！使用 Scala actor 就这么容易。

并发地执行动作

上面的代码没有反映出并发性 — 到目前为止给出的代码更像是另一种异步的方法调用形式，您看不出区别。（从技术上说，在第二个示例中引入接近无限循环之前的代码中，可以猜出有一定的并发性存在，但这只是偶然的证据，不是明确的证明）。

为了证明在幕后确实有多个线程存在，我们深入研究一下前一个示例：

清单 **4.** 我要拍特写了

```
object Actor3
{
  case class Speak(line : String);
  case class Gesture(bodyPart : String, action : String);
  case class NegotiateNewContract;
  case class ThatsAWrap;

  def main(args : Array[String]) =
  {
    def ct =
      "Thread " + Thread.currentThread().getName() + ": "
    val badActor =
      actor
      {
        var done = false
        while (! done)
        {
          receive
          {
```

```
              case NegotiateNewContract =>
                System.out.println(ct + "I won't do it for less than $1 million!")
              case Speak(line) =>
                System.out.println(ct + line)
              case Gesture(bodyPart, action) =>
                System.out.println(ct + "(" + action + "s " + bodyPart + ")")
              case ThatsAWrap =>
                System.out.println(ct + "Great cast party, everybody! See ya!")
                done = true
              case _ =>
                System.out.println(ct + "Huh? I'll be in my trailer.")
            }
          }
        }

      System.out.println(ct + "Negotiating...")
      badActor ! NegotiateNewContract
      System.out.println(ct + "Speaking...")
      badActor ! Speak("Do ya feel lucky, punk?")
      System.out.println(ct + "Gesturing...")
      badActor ! Gesture("face", "grimaces")
      System.out.println(ct + "Speaking again...")
      badActor ! Speak("Well, do ya?")
      System.out.println(ct + "Wrapping up")
      badActor ! ThatsAWrap
    }
  }
```

运行这个新示例，就会非常明确地发现确实有两个不同的线程：

- `main` 线程（所有 Java 程序都以它开始）
- `Thread-2` 线程，它是 Scala Actors 库在幕后生成的

因此，在启动第一个 actor 时，本质上已经开始了多线程执行。

但是，习惯这种新的执行模型可能有点儿困难，因为这是一种全新的并发性考虑方式。例如，请考虑 前一篇文章 中的 Producer/Consumer 模型。那里有大量代码，尤其是在 `Drop` 类中，我们可以清楚地看到线程之间，以及线程与保证所有东西同步的监视器之间有哪些交互活动。为了便于参考，我在这里给出前一篇文章中的 V3 代码：

清单 **5. ProdConSample，v3 (Scala)**

```
package com.tedneward.scalaexamples.scala.V3
{
  import concurrent.MailBox
  import concurrent.ops._

  object ProdConSample
  {
    class Drop
    {
      private val m = new MailBox()
```

```
  private case class Empty()
  private case class Full(x : String)

  m send Empty()  // initialization

  def put(msg : String) : Unit =
  {
    m receive
    {
      case Empty() =>
        m send Full(msg)
    }
  }

  def take() : String =
  {
    m receive
    {
      case Full(msg) =>
        m send Empty(); msg
    }
  }
}

def main(args : Array[String]) : Unit =
{
  // Create Drop
  val drop = new Drop()

  // Spawn Producer
  spawn
  {
    val importantInfo : Array[String] = Array(
      "Mares eat oats",
      "Does eat oats",
      "Little lambs eat ivy",
      "A kid will eat ivy too"
    );

    importantInfo.foreach((msg) => drop.put(msg))
    drop.put("DONE")
  }

  // Spawn Consumer
  spawn
  {
    var message = drop.take()
    while (message != "DONE")
    {
      System.out.format("MESSAGE RECEIVED: %s%n", message)
```

```
          message = drop.take()
        }
      }
    }
  }
}
```

尽管看到 Scala 如何简化这些代码很有意思，但是它实际上与原来的 Java 版本没有概念性差异。现在，看看如果把 Producer/Consumer 示例的基于 actor 的版本缩减到最基本的形式，它会是什么样子：

清单 **6. Take 1**，开拍！生产！消费！

```
object ProdConSample1
{
  case class Message(msg : String)

  def main(args : Array[String]) : Unit =
  {
    val consumer =
      actor
      {
        var done = false
        while (! done)
        {
          receive
          {
            case msg =>
              System.out.println("Received message! -> " + msg)
              done = (msg == "DONE")
          }
        }
      }

    consumer ! "Mares eat oats"
    consumer ! "Does eat oats"
    consumer ! "Little lambs eat ivy"
    consumer ! "Kids eat ivy too"
    consumer ! "DONE"
  }
}
```

第一个版本确实简短多了，而且在某些情况下可能能够完成所需的所有工作；但是，如果运行这段代码并与以前的版本做比较，就会发现一个重要的差异 — 基于 actor 的版本是一个多位置缓冲区，而不是我们以前使用的单位置缓冲。这看起来是一项改进，而不是缺陷，但是我们要通过对比确认这一点。我们来创建 `Drop` 的基于 actor 的版本，在这个版本中所有对 `put()` 的调用必须由对 `take()` 的调用进行平衡。

幸运的是，Scala Actors 库很容易模拟这种功能。希望让 Producer 一直阻塞，直到 Consumer 接收了消息；实现的方法很简单：让 Producer 一直阻塞，直到它从 Consumer 收到已经接收消息的确认。从某种意义上说，这就是以前的基于监视器的代码所做的，那个版本通过对锁对象使用监视器发送这种信号。

在 Scala Actors 库中，最容易的实现方法是使用 `!?` 方法而不是 `!` 方法（这样就会一直阻塞到收到确认时）。（在 Scala Actors 实现中，每个 Java 线

程都是一个 actor，所以回复会发送到与 main 线程隐式关联的邮箱）。这意味着 Consumer 需要发送某种确认；这要使用隐式继承的 reply（它还继承 receive 方法），见清单 7：

清单 **7. Take 2**，开拍！

```
object ProdConSample2
{
  case class Message(msg : String)

  def main(args : Array[String]) : Unit =
  {
    val consumer =
      actor
      {
        var done = false
        while (! done)
        {
          receive
          {
            case msg =>
              System.out.println("Received message! -> " + msg)
              done = (msg == "DONE")
              reply("RECEIVED")
          }
        }
      }

    System.out.println("Sending....")
    consumer !? "Mares eat oats"
    System.out.println("Sending....")
    consumer !? "Does eat oats"
    System.out.println("Sending....")
    consumer !? "Little lambs eat ivy"
    System.out.println("Sending....")
    consumer !? "Kids eat ivy too"
    System.out.println("Sending....")
    consumer !? "DONE"
  }
}
```

如果喜欢使用 spawn 把 Producer 放在 main() 之外的另一个线程中（这非常接近最初的代码），那么代码可能像清单 8 这样：

清单 **8. Take 4**，开拍！

```
object ProdConSampleUsingSpawn
{
  import concurrent.ops._

  def main(args : Array[String]) : Unit =
```

```
  {
    // Spawn Consumer
    val consumer =
      actor
      {
        var done = false
        while (! done)
        {
          receive
          {
            case msg =>
              System.out.println("MESSAGE RECEIVED: " + msg)
              done = (msg == "DONE")
              reply("RECEIVED")
          }
        }
      }

    // Spawn Producer
    spawn
    {
      val importantInfo : Array[String] = Array(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "A kid will eat ivy too",
        "DONE"
      );

      importantInfo.foreach((msg) => consumer !? msg)
    }
  }
}
```

无论从哪个角度来看，基于 actor 的版本都比原来的版本简单多了。读者只要让 actor 和隐含的邮箱自己发挥作用即可。

但是，这并不简单。actor 模型完全颠覆了考虑并发性和线程安全的整个过程；在以前的模型中，我们主要关注共享的数据结构（数据并发性），而现在主要关注操作数据的代码本身的结构（任务并发性），尽可能少共享数据。请注意 Producer/Consumer 示例的不同版本的差异。在以前的示例中，并发功能是围绕 `Drop` 类（有界限的缓冲区）显式编写的。在本文中的版本中，`Drop` 甚至没有出现，重点在于两个 actor（线程）以及它们之间的交互（通过不共享任何东西的消息）。

当然，仍然可以用 actor 构建以数据为中心的并发构造；只是必须采用稍有差异的方式。请考虑一个简单的“计数器”对象，它使用 actor 消息传达“increment”和“get”操作，见清单 9：

清单 **9. Take 5**，计数!

```
object CountingSample
 {
   case class Incr
   case class Value(sender : Actor)
```

```
  case class Lock(sender : Actor)
  case class UnLock(value : Int)

  class Counter extends Actor
  {
    override def act(): Unit = loop(0)

    def loop(value: int): Unit = {
      receive {
        case Incr()   => loop(value + 1)
        case Value(a) => a ! value; loop(value)
        case Lock(a)  => a ! value
                         receive { case UnLock(v) => loop(v) }
        case _        => loop(value)
      }
    }
  }

  def main(args : Array[String]) : Unit =
  {
    val counter = new Counter
    counter.start()
    counter ! Incr()
    counter ! Incr()
    counter ! Incr()
    counter ! Value(self)
    receive { case cvalue => Console.println(cvalue) }
    counter ! Incr()
    counter ! Incr()
    counter ! Value(self)
    receive { case cvalue => Console.println(cvalue) }
  }
}
```

为了进一步扩展 Producer/Consumer 示例，清单 10 给出一个在内部使用 actor 的 `Drop` 版本（这样，其他 Java 类就可以使用这个 `Drop`，而不需要直接调用 actor 的方法）：

清单 **10.** 在内部使用 **actor** 的 **Drop**

```
object ActorDropSample
{
  class Drop
  {
    private case class Put(x: String)
    private case object Take
    private case object Stop

    private val buffer =
      actor
      {
```

```
        var data = ""
        loop
        {
          react
          {
            case Put(x) if data == "" =>
              data = x; reply()
            case Take if data != "" =>
              val r = data; data = ""; reply(r)
            case Stop =>
              reply(); exit("stopped")
          }
        }
      }

    def put(x: String) { buffer !? Put(x) }
    def take() : String = (buffer !? Take).asInstanceOf[String]
    def stop() { buffer !? Stop }
  }

  def main(args : Array[String]) : Unit =
  {
    import concurrent.ops._

    // Create Drop
    val drop = new Drop()

    // Spawn Producer
    spawn
    {
      val importantInfo : Array[String] = Array(
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "A kid will eat ivy too"
      );

      importantInfo.foreach((msg) => { drop.put(msg) })
      drop.put("DONE")
    }

    // Spawn Consumer
    spawn
    {
      var message = drop.take()
      while (message != "DONE")
      {
        System.out.format("MESSAGE RECEIVED: %s%n", message)
        message = drop.take()
      }
      drop.stop()
```

```
      }
    }
  }
```

可以看到，这需要更多代码（和更多的线程，因为每个 actor 都在一个线程池内部起作用），但是这个版本的 API 与以前的版本相同，它把所有与并发性相关的代码都放在 `Drop` 内部，这正是 Java 开发人员所期望的。

actor 还有更多特性。

在规模很大的系统中，让每个 actor 都由一个 Java 线程支持是非常浪费资源的，尤其是在 actor 的等待时间比处理时间长的情况下。在这些情况下，基于事件的 *actor* 可能更合适；这种 actor 实际上放在一个闭包中，闭包捕捉 actor 的其他动作。也就是说，现在并不通过线程状态和寄存器表示代码块（函数）。当一个消息到达 actor 时（这时显然需要活动的线程），触发闭包，闭包在它的活动期间借用一个活动的线程，然后通过回调本身终止或进入“等待”状态，这样就会释放线程。（请参见 参考资料 中 Haller/Odersky 的文章）。

在 Scala Actors 库中，这要使用 `react` 方法而不是前面使用的 `receive`。使用 `react` 的关键是在形式上 `react` 不能返回，所以 `react` 中的实现必须重复调用包含 `react` 块的代码块。简便方法是使用 `loop` 结构创建一个接近无限的循环。这意味着 清单 10 中的 `Drop` 实现实际上只通过借用调用者的线程执行操作，这会减少执行所有操作所需的线程数。（在实践中，我还没有见过在简单的示例中出现这种效果，所以我想我们只能暂且相信 Scala 设计者的说法）。

在某些情况下，可能选择通过派生基本的 `Actor` 类（在这种情况下，必须定义 `act` 方法，否则类仍然是抽象的）创建一个新类，它隐式地作为 actor 执行。尽管这是可行的，但是这种思想在 Scala 社区中不受欢迎；在一般情况下，我在这里描述的方法（使用 `Actor` 对象中的 `actor` 方法）是创建 actor 的首选方法。

结束语

因为 actor 编程需要与“传统”对象编程不同的风格，所以在使用 actor 时要记住几点。

首先，actor 的主要能力来源于消息传递风格，而不采用阻塞-调用风格，这是它的主要特点。（有意思的是，也有使用消息传递作为核心机制的面向对象语言。最知名的两个例子是 Objective-C 和 Smalltalk，还有 ThoughtWorker 的 Ola Bini 新创建的 Ioke）。如果创建直接或间接扩展 `Actor` 的类，那么要确保对对象的所有调用都通过消息传递进行。

第二，因为可以在任何时候交付消息，而且更重要的是，在发送和接收之间可能有相当长的延迟，所以一定要确保消息携带正确地处理它们所需的所有状态。这种方式会：

- 让代码更容易理解（因为消息携带处理所需的所有状态）。
- 减少 actor 访问某些地方的共享状态的可能性，从而减少发生死锁或其他并发性问题的机会。

第三，actor 应该不会阻塞，您从前面的内容应该能够看出这一点。从本质上说，阻塞是导致死锁的原因；代码可能产生的阻塞越少，发生死锁的可能性就越低。

很有意思的是，如果您熟悉 Java Message Service (JMS) API，就会发现我给出的这些建议在很大程度上也适用于 JMS — 毕竟，actor 消息传递风格只是在实体之间传递消息，JMS 消息传递也是在实体之间传递消息。它们的差异在于，JMS 消息往往比较大，在层和进程级别上操作；而 actor 消息往往比较小，在对象和线程级别上操作。如果您掌握了 JMS，actor 也不难掌握。

actor 并不是解决所有并发性问题的万灵药，但是它们为应用程序或库代码的建模提供了一种新的方式，所用的构造相当简单明了。尽管它们的工作方式有时与您预期的不一样，但是一些行为正是我们所熟悉的 — 毕竟，我们在最初使用对象时也有点不习惯，只要经过努力，您也会掌握并喜欢上 actor。

本期就到这里了；下一期再见！

参考资料

学习

- 面向 *Java* 开发人员的 *Scala* 指南（Ted Neward，developerWorks）：阅读本系列的所有文章。

- 文章“Event-Based Programming without Inversion of Control”（Ecole Polytechnique Federale de Lausanne，作者 Philipp Haller 和 Martin Odersky）解释了并发性为什么是不可缺少的，介绍了 Scala actor 库背后的原则。作者的另一篇文章“Actors That Unify Events and Threads”进一步解释了 Scala actor 实现如何隐藏线程化 actor 和事件驱动 actor 之间的差异。

- *Java Concurrency in Practice*（Addison-Wesley Professional，2006 年）全面讨论了并发性的相关主题。

- *Concurrent Programming in Java*（Prentice Hall PTR，1999 年）讨论了并行性和并发性的许多研究领域，通过大量模式和设计提示介绍如何在 Java 中更好地利用多线程。

- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度了解函数编程的优点和用法。

- “Scala by Example”（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。

- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：这是第一本介绍 Scala 的图书。

- Bjarne Stroustrup：他设计和实现了 C++，他把 C++ 称为“更好的 C”。

- *Java Puzzlers: Traps, Pitfalls, and Corner Cases*（Addison-Wesley Professional，2005 年 7 月）通过有趣和发人深省的编程难题介绍 Java 编程语言的特点。

- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各方面的文章。

获得产品和技术

- 下载 Scala：跟随本系列学习 Scala！

- SUnit：SUnit 是标准 Scala 发行版的一部分，在 *scala.testing* 包中提供。

讨论

- 通过参与 developerWorks 博客 加入 developerWorks 社区。

关于作者

Ted Neward 是 ThoughtWorks 的一名顾问，ThoughtWorks 是一家在全球提供咨询服务的公司，他还是 Neward & Associates 的主管，负责有关 Java、.NET 和 XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿西雅图附近。

对本文的评价

◉太差！ (1)
○需提高 (2)
○一般；尚可 (3)
○好文章 (4)

○真棒！(5)

建议？

反馈意见

回页首

面向 Java 开发人员的 Scala 指南:

Scala + Twitter = Scitter

通过 **Scala** 可以实现更好的社交网络

级别: 初级

Ted Neward, 主管, ThoughtWorks

2009 年 8 月 20 日

抽象地讨论 Scala 是一件有趣的事情，但对于本专栏的大多数读者而言，需要通过实践才能理解理论和应用之间的区别。在本期文章中，Ted Neward 将使用 Scala 为客户构建基础框架，用于访问流行的微型博客系统 Twitter。

Twitter 迅速占领了 Internet 市场。您肯定知道这个出色的社交网络工具允许订阅者提供关于他们自身以及当前正在执行的任务的简要状态更新。追随者将接收到他们的 "Twitter 提要" 的更新，这与博客将更新生成到博客阅读者的提要中极为类似。

就其本身而言，Twitter 是对社交网络的有趣讨论，并且是用户之间的新一代 "高度互联"，它具备您能想到的所有优点和缺点。

由于 Twitter 很早就发布了其 API，因此大量 Twitter 客户机应用程序涌入到 Internet 上。由于该 API 主要建立在直观和易于理解的基础上，因此许多开发人员都发现有必要构建一个自己的 Twitter 客户机，这与学习 Web 技术的开发人员构建自己的博客服务器极为类似。

考虑到 Scala 的功能性（这看上去能很好地协同 Twitter 的 REST 式特性）以及非常出众的 XML 处理特性，因此尝试构建一个用于访问 Twitter 的 Scala 客户机库应该是一个非常不错的体验。

关于本系列

Ted Neward 将深入探讨 Scala 编程语言，并带领您一路随行。在本 系列 中，您将学习最新的热点以及 Scala 的语言功能。Scala 代码和 Java™ 代码将在必要时同时出现以方便进行比较，但您会发现 Scala 中的许多内容都与 Java 没有直接关系 — 这便是 Scala 的魄力所在！毕竟，如果 Java 能够做到，为什么还要大费周折来学习 Scala 呢?

何为 Twitter?

在详细讨论之前，我们先来看看 Twitter API。

简单来说，Twitter 是一个 "微型博客" — 关于您自己的简短个性化提要，不超过 140 个字符，任何 "追随者" 都可以通过 Web 更新、RSS、文本消息等方式接收它们。（140 字符的限制完全来自文本消息，它是 Twitter 的主要来源渠道，并受到类似的限制）。

从实际的角度来说，Twitter 是一个最具 *REST* 特征 的 API，您可以使用一些种类的消息格式 — XML、ATOM、RSS 或 JSON — 来发送或从 Twitter 服务器接收消息。不同的 URL，与不同的消息和它们所需及可选的消息部分相结合，可以发起不同的 API 调用。例如，如果您希望接收 Twitter 上所有人的所有 "Tweets"（Twitter 更新）的完整列表（也称作 "公共时间轴"），您需要准备一个 XML、ATOM、RSS 或 JSON 消息，将它发送给合适的 URL，并采用与 Twitter 网站（apiwiki.twitter.com）上相同的格式来使用结果:

最具 ***REST*** 特征 是什么意思?

一些读者会对我所使用的最具 *REST* 特征 短语感到好奇；这需要一些说明。"Twitter API 试图符合 Representational State Transfer (REST) 的设计原则"。并且在很大程度上说它做到了。该思想的创造者 Roy Fielding 可能不同意 Twitter 使用这个术语，但实现来说，Twitter 的方法将适合大多数人的 REST 定义。我只希望避免关于 REST 定义的激烈争论。因此，我使用了限定词 "最"。

---

public_timeline

返回设定了自定义用户图标的

非保护用户的 **20** 条最新状态。不需要身份验证。

注意，公共时间轴将缓存 60 秒钟

因此频繁请求它不再浪费资源。

URL: `http://twitter.com/statuses/public_timeline.format`

格式：xml、json、rss、atom

方法：GET

API 限制：不适用

返回：状态元素列表

从编程的角度来说，这意味着我们给 Twitter 服务器发送一个简单的 `GET HTTP` 请求，并且我们将获取一组封装在 XML、RSS、ATOM 或 JSON 消息中的“状态”消息。Twitter 站点将“状态”消息定义为类似清单 1 所示的内容：

清单 **1.** 您好世界，您在哪里？

```
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <title>Twitter / tedneward</title>
  <id>tag:twitter.com,2007:Status</id>
  <link type="text/html" rel="alternate" href="http://twitter.com/tedneward"/>
  <updated>2009-03-07T13:48:31+00:00</updated>
  <subtitle>Twitter updates from Ted Neward / tedneward.</subtitle>
  <entry>
        <title>tedneward: @kdellison Happens to the best of us...</title>
        <content type="html">tedneward: @kdellison Happens to the best of us...</content>
        <id>tag:twitter.com,2007:http://twitter.com/tedneward/statuses/1292396349</id>
        <published>2009-03-07T11:07:18+00:00</published>
        <updated>2009-03-07T11:07:18+00:00</updated>
        <link type="text/html" rel="alternate"
        href="http://twitter.com/tedneward/statuses/1292396349"/>
        <link type="image/png" rel="image"
        href="http://s3.amazonaws.com/twitter_production/profile_images/
         55857457/javapolis_normal.png"/>
        <author>
          <name>Ted Neward</name>
          <uri>http://www.tedneward.com</uri>
        </author>
  </entry>
</feed>
```

状态消息中的大部分元素（如果不是全部的话）都很直观，因此不再赘述。

由于我们可以采用三种基于 XML 的格式使用 Twitter 消息，以及 Scala 具备一些非常强大的 XML 特性，包括 XML 字面值和类似 XPath 的查询语法 API，因此编写可以发送和接收 Twitter 消息的 Scala 库只需要一些基础的 Scala 编码工作。举例来说，通过 Scala 使用清单 1 消息来提取状态更新的标题或内容可以利用 Scala 的 XML 类型和 \ 及 \\ 方法，如 清单 2 所示：

清单 **2.** 您好 **Ted**，您在哪里？

```
<![CDATA[
package com.tedneward.scitter.test
{
  class ScitterTest
  {
    import org.junit._, Assert._
```

```
    @Test def simpleAtomParse =
    {
      val atom =
        <feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
          <title>Twitter / tedneward</title>
          <id>tag:twitter.com,2007:Status</id>
          <link type="text/html" rel="alternate" href="http://twitter.com/tedneward"/>
          <updated>2009-03-07T13:48:31+00:00</updated>
          <subtitle>Twitter updates from Ted Neward / tedneward.</subtitle>
          <entry>
            <title>tedneward: @kdellison Happens to the best of us...</title>
            <content type="html">tedneward: @kdellison
                                 Happens to the best of us...</content>
            <id>tag:twitter.com,2007:
                http://twitter.com/tedneward/statuses/1292396349</id>
            <published>2009-03-07T11:07:18+00:00</published>
            <updated>2009-03-07T11:07:18+00:00</updated>
            <link type="text/html" rel="alternate"
                  href="http://twitter.com/tedneward/statuses/1292396349"/>
            <link type="image/png" rel="image"
                  href="http://s3.amazonaws.com/twitter_production/profile_images/
                        55857457/javapolis_normal.png"/>
            <author>
              <name>Ted Neward</name>
              <uri>http://www.tedneward.com</uri>
            </author>
          </entry>
        </feed>

      assertEquals(atom \\ "entry" \ "title",
                   "tedneward: @kdellison Happens to the best of us...")
    }
  }
}
]]>
```

有关 Scala 的 XML 支持的更多详细信息，请参阅“Scala 和 XML”（参见 参考资料）。

实际上，使用原始 XML 本身并不是一个有趣的练习。如果 Scala 的宗旨是让我们的生活更加轻松，那么可以创建一个或一组专用于简化 Scala 消息发送和接收任务的类。作为其中一个目标，应该能够在“普通” Java 程序中方便地使用库（这意味着可以方便地从任何可理解普通 Java 语义的环境中来访问它，比如说 Groovy 或 Clojure）。

API 设计

在深入了解 Scala/Twitter 库的 API 设计之前（根据同事 ThoughtWorker Neal Ford 的建议，我将它称作“Scitter”），需要明确一些需求。

首先，Scitter 显然会对网络访问有一些依赖 — 并且可扩展到 Twitter 服务器 — 这会使测试变得非常困难。

其次，我们需要解析（和测试）Twitter 发回的各种格式。

第三，我们希望隐藏 API 内部各种格式之间的差异，以便客户机不需要担心已记录的 Twitter 消息格式，但是可以仅使用标准类。

最后，由于 Twitter 依赖“通过身份验证的用户”才能使用大量 API，因此 Scitter 库需要适应“验证”和“未验证” API 之间的差异，而不会让事情变得过

于复杂。

网络访问需要一些形式的 HTTP 通信，以便联系 Twitter 服务器。虽然我们可以使用 Java 库本身（特别是 URL 类及其同胞），但由于 Twitter API 需要大量请求和响应主体连接，因此可以更加轻松地使用不同的 HTTP API，特别是 Apache Commons HttpClient 库。为了更便于测试客户机 API，实际通信将隐藏在一些 API 内部的 Scitter 库中，以便能够更加轻松地切换到另一个 HTTP 库（其必要性不太容易想到），并能模拟实际网络通信以简化测试（其作用很容易想到）。

结果，第一个测试是 Scala 化 HttpClient 调用，以确保基本通信模式就位；注意，由于 HttpClient 依赖另外两个 Apache 库（Commons Logging 和 Commons Codec），因此还需要在运行时提供这些库；对于那些希望开发相似种类代码的读者，确保类路径中包括所有三个库。

由于最易于使用的 Twitter API 是测试 API

因此在请求格式中返回 *"ok"*，并附带 *200 OK HTTP* 状态码。

我们将使用它作为 Scitter 测试中的保留条款。它位于 URL http://twitter.com/help/test.format（其中，"format" 为 "xml" 或 "json"；至于目前，我们将选择使用 "xml"），并且仅有的支持 HTTP 方法是 `GET`。HttpClient 代码简明易懂，如清单 3 所示：

清单 **3. Twitter PING**！

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    import org.apache.commons.httpclient._, methods._, params._, cookie._

    @Test def callTwitterTest =
    {
      val testURL = "http://twitter.com/help/test.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(testURL)

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()

      assertEquals(statusLine.getStatusCode(), 200)
      assertEquals(statusLine.getReasonPhrase(), "OK")
    }
  }
}
```

此代码中最重要的一部分是 HttpClient 样板 — 感兴趣的读者应该查阅 HttpClient API 文档了解详细信息。假设连接到公共 Internet 的网络可用（并且 Twitter 并未修改其公共 API），那么该测试应该能顺利通过。

鉴于此，我们详细分析 Scitter 客户机的第一部分。这意味着我们需要解决一个设计问题：如何构建 Scitter 客户机来处理经过验证和未经过验证的调用。目前，我将采用典型的 Scala 方式，假定验证是“按对象”执行的，因此将需要验证的调用放在类定义中，并在未验证的调用放在对象定义中：

清单 **4. Scitter.test**

```
package com.tedneward.scitter
{
  /**
   * Object for consuming "non-specific" Twitter feeds, such as the public timeline.
   * Use this to do non-authenticated requests of Twitter feeds.
   */
  object Scitter
  {
    import org.apache.commons.httpclient._, methods._, params._, cookie._

    /**
     * Ping the server to see if it's up and running.
     *
     * Twitter docs say:
     * test
     * Returns the string "ok" in the requested format with a 200 OK HTTP status code.
     * URL: http://twitter.com/help/test.format
     * Formats: xml, json
     * Method(s): GET
     */
    def test : Boolean =
    {
      val client = new HttpClient()

      val method = new GetMethod("http://twitter.com/help/test.xml")

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      statusLine.getStatusCode() == 200
    }
  }
  /**
   * Class for consuming "authenticated user" Twitter APIs. Each instance is
   * thus "tied" to a particular authenticated user on Twitter, and will
   * behave accordingly (according to the Twitter API documentation).
   */
  class Scitter(username : String, password : String)
  {
  }
}
```

目前，我们将网络抽象放在一边 — 稍后，当离线测试变得更加重要时再添加它。当我们更好地理解如何使用 HttpClient 类时，这还将帮助避免“过度抽象”网络通信。

由于已经明确区分了验证和未验证 Twitter 客户机，因此我们将快速创建一个经过验证的方法。看上去 Twitter 提供了一个可验证用户登录凭证的 API。再次，HttpClient 代码将类似于之前的代码，除了将用户名和密码传递到 Twitter API 中之外。

这引出了 Twitter 如何验证用户的概念。快速查看 Twitter API 页面后，可以发现 Twitter 使用的是一种 Stock HTTP 验证方法，这与任何经过验证的资源在 HTTP 中的方法相同。这意味着 HttpClient 代码必须提供用户名和密码作为 HTTP 请求的一部分，而不是作为 POST 的主体，如清单 5 所示：

清单 **5.** 您好 **Twitter**，是我！

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    def testUser = "TwitterUser"
        def testPassword = "TwitterPassword"

    @Test def verifyCreds =
    {
      val client = new HttpClient()

      val verifyCredsURL = "http://twitter.com/account/verify_credentials.xml"
      val method = new GetMethod(verifyCredsURL)

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.getParams().setAuthenticationPreemptive(true)
      val defaultcreds = new UsernamePasswordCredentials(testUser, testPassword)
      client.getState().setCredentials(new AuthScope("twitter.com", 80,
                                                     AuthScope.ANY_REALM), defaultcreds)

      client.executeMethod(method)

      val statusLine = method.getStatusLine()

      assertEquals(200, statusLine.getStatusCode())
      assertEquals("OK", statusLine.getReasonPhrase())
    }
  }
}
```

注意，要让此测试顺利通信，用户名和密码字段将需要输入 Twitter 能接收的内容 — 我在开发时使用了自己的 Twitter 用户名和密码，但显然您需要使用自己设定的用户名和密码。注册新的 Twitter 帐户相当简单，因此我假定您已经拥有一个帐户，或者知道如何注册（很好。我会等待完成此任务）。

完成后，使用用户名和密码构造函数参数将它映射到 Scitter 类非常简单，如清单 6 所示：

清单 **6. Scitter.verifyCredentials**

```
package com.tedneward.scitter
{
  import org.apache.commons.httpclient._, auth._, methods._, params._

  // ...

  /**
   * Class for consuming "authenticated user" Twitter APIs. Each instance is
   * thus "tied" to a particular authenticated user on Twitter, and will
   * behave accordingly (according to the Twitter API documentation).
   */
  class Scitter(username : String, password : String)
  {
    /**
     * Verify the user credentials against Twitter.
     *
     * Twitter docs say:
     * verify_credentials
     * Returns an HTTP 200 OK response code and a representation of the
     * requesting user if authentication was successful; returns a 401 status
     * code and an error message if not.  Use this method to test if supplied
     * user credentials are valid.
     * URL: http://twitter.com/account/verify_credentials.format
     * Formats: xml, json
     * Method(s): GET
     */
    def verifyCredentials : Boolean =
    {
      val client = new HttpClient()

      val method = new GetMethod("http://twitter.com/help/test.xml")

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.getParams().setAuthenticationPreemptive(true)
      val creds = new UsernamePasswordCredentials(username, password)
      client.getState().setCredentials(
        new AuthScope("twitter.com", 80, AuthScope.ANY_REALM), creds)

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      statusLine.getStatusCode() == 200
    }
  }
}
```

清单 7 中相应的 Scitter 类测试也相当简单:

清单 **7.** 测试 **Scitter.verifyCredentials**

```
package com.tedneward.scitter.test
{
  class ScitterTests
  {
    import org.junit._, Assert._
    import com.tedneward.scitter._

    def testUser = "TwitterUsername"
      def testPassword = "TwitterPassword"

      // ...

    @Test def verifyCreds =
    {
      val scitter = new Scitter(testUser, testPassword)
      val result = scitter.verifyCredentials
      assertTrue(result)
    }
  }
}
```

不算太糟。库的基本结构已经成形，但显然还有很长的路要走，特别是因为目前实际上未执行任何特定于 Scala 的任务 — 在面向对象设计中，库的构建并不像练习那样简单。因此，我们开始使用一些 XML，并通过更加合理的格式将它返回。

从 XML 到对象

现在可以添加的最简单的 API 是 public_timeline，它收集 Twitter 从所有用户处接收到的最新的 *n* 更新，并返回它们以便于进行使用。与之前讨论的另外两个 API 不同，public_timeline API 返回一个响应主体（而不是仅依赖于状态码），因此我们需要分解生成的 XML/RSS/ATOM/，然后将它们返回给 Scitter 客户机。

现在，我们编写一个探索测试，它将访问公共提要并将结果转储到 `stdout` 以便进行分析，如清单 8 所示：

清单 **8.** 大家都在忙什么?

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    @Test def callTwitterPublicTimeline =
    {
      val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(publicFeedURL)
      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
```

```
        new DefaultHttpMethodRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      assertEquals(statusLine.getStatusCode(), 200)
      assertEquals(statusLine.getReasonPhrase(), "OK")

      val responseBody = method.getResponseBodyAsString()
      System.out.println("callTwitterPublicTimeline got... ")
      System.out.println(responseBody)
    }
  }
}
```

运行后，结果每次都会有所不同，因为公共 Twitter 服务器上有许多用户，但通常应与清单 9 的 JUnit 文本文件转储类似：

清单 **9.** 我们的 **Tweets** 结果

```
<statuses type="array">
    <status>
      <created_at>Tue Mar 10 03:14:54 +0000 2009</created_at>
      <id>1303777336</id>
      <text>She really is. http://tinyurl.com/d65hmj</text>
      <source><a href="http://iconfactory.com/software/twitterrific">twitterrific</a>
        </source>
      <truncated>false</truncated>
      <in_reply_to_status_id></in_reply_to_status_id>
      <in_reply_to_user_id></in_reply_to_user_id>
      <favorited>false</favorited>
      <user>
          <id>18729101</id>
          <name>Brittanie</name>
          <screen_name>brittaniemarie</screen_name>
          <description>I'm a bright character. I suppose.</description>
          <location>Atlanta or Philly.</location>
          <profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images/
                              81636505/goodish_normal.jpg</profile_image_url>
          <url>http://writeitdowntakeapicture.blogspot.com</url>
          <protected>false</protected>
          <followers_count>61</followers_count>
      </user>
    </status>
    <status>
      <created_at>Tue Mar 10 03:14:57 +0000 2009</created_at>
      <id>1303777334</id>
      <text>Number 2 of my four life principles.  "Life is fun and rewarding"</text>
      <source>web</source>
      <truncated>false</truncated>
      <in_reply_to_status_id></in_reply_to_status_id>
```

```
      <in_reply_to_user_id></in_reply_to_user_id>
      <favorited>false</favorited>
      <user>
          <id>21465465</id>
          <name>Dale Greenwood</name>
          <screen_name>Greeendale</screen_name>
          <description>Vegetarian. Eat and use only organics.
                       Love helping people become prosperous</description>
          <location>Melbourne Australia</location>
          <profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images/
                              90659576/Dock_normal.jpg</profile_image_url>
          <url>http://www.4abundance.mionegroup.com</url>
          <protected>false</protected>
          <followers_count>15</followers_count>
       </user>
     </status>
       (A lot more have been snipped)
</statuses>
```

通过查看结果和 Twitter 文档可以看出，调用的结果是一组具备一致消息结构的简单“状态”消息。使用 Scala 的 XML 支持分离结果相当简单，但我们会在基本测试通过后立即简化它们，如清单 10 所示：

清单 **10.** 大家都在忙什么？

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    @Test def simplePublicFeedPullAndParse =
    {
      val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(publicFeedURL)
      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))
      val statusCode = client.executeMethod(method)
      val responseBody = new String(method.getResponseBody())

      val responseXML = scala.xml.XML.loadString(responseBody)
      val statuses = responseXML \\ "status"

      for (n <- statuses.elements)
      {
        n match
        {
          case <status>{ contents @ _*}</status> =>
```

```
          {
            System.out.println("Status: ")
            contents.foreach((c) =>
              c match
              {
                case <text>{ t @ _*}</text> =>
                  System.out.println("\tText: " + t.text.trim)
                case <user>{ contents2 @ _* }</user> =>
                {
                  contents2.foreach((c2) =>
                    c2 match
                    {
                      case <screen_name>{ u }</screen_name> =>
                        System.out.println("\tUser: " + u.text.trim)
                      case _ =>
                                                ()
                    }
                  )
                }
                case _ =>
                                    ()
              }
            )
          }
          case _ =>
            () // or, if you prefer, System.out.println("Unrecognized element!")
        }
      }
    }
  }
}
```

随着示例代码模式的变化，这并不值得推荐 — 这有点类似于 DOM，依次导航到各个子元素，提取文本，然后导航到另一个节点。我可以仅执行两个 XPath 样式的查询，如清单 11 所示：

清单 **11.** 替代解析方法

```
for (n <- statuses.elements)
{
     val text = (n \\ "text").text
       val screenName = (n \\ "user" \ "screen_name").text
}
```

这显然更加简短，但它带来了两个基本问题：

- 我们可以强制 Scala 的 XML 库针对每个元素或子元素遍历一次图，其速度会随时间减慢。
- 我们仍然需要直接处理 XML 消息的结构。这是两个问题中最为重要的。

也就是说，这种方式不具备可伸缩性 — 假设我们最终对 Twitter 状态消息中的每个元素都感兴趣，我们将需要分别从各状态中提取各个元素。

这又造成了另一个与各格式本身相关的问题。记住，Twitter 可以使用四种不同的格式，并且我们不希望 Scitter 客户机需要了解它们之间的任何差异，因此 Scitter 需要一个能返回给客户机的中间结构，以便未来使用，如清单 12 所示：

清单 **12. Breaker**，您的状态是什么？

```
abstract class Status
{
  val createdAt : String
  val id : Long
  val text : String
  val source : String
  val truncated : Boolean
  val inReplyToStatusId : Option[Long]
  val inReplyToUserId : Option[Long]
  val favorited : Boolean
  val user : User
}
```

这与 `User` 方式相类似，考虑到简洁性，我就不再重复了。注意，`User` 子元素有一个有趣的问题 — 虽然存在 Twitter 用户类型，但其中内嵌了一个可选的“最新状态”。状态消息还内嵌了一个用户。对于这种情况，为了帮助避免一些潜在的递归问题，我选择创建一个嵌入在 `Status` 内部的 `User` 类型，以反映所出现的 `User` 数据；反之亦然，`Status` 也可以嵌入在 `User` 中，这样可以明确避免该问题。（至少，在没发现问题之前，这种方法是有效的）。

现在，创建了表示 Twitter 消息的对象类型之后，我们可以遵循 XML 反序列化的公共 Scala 模式：创建相应的对象定义，其中包含一个 `fromXml` 方法，用于将 XML 节点分离到对象实例中，如清单 13 所示：

清单 **13.** 分解 **XML**

```
/**
 * Object wrapper for transforming (format) into Status instances.
 */
object Status
{
  def fromXml(node : scala.xml.Node) : Status =
  {
    new Status {
      val createdAt = (node \ "created_at").text
      val id = (node \ "id").text.toLong
      val text = (node \ "text").text
      val source = (node \ "source").text
      val truncated = (node \ "truncated").text.toBoolean
      val inReplyToStatusId =
        if ((node \ "in_reply_to_status_id").text != "")
          Some((node \"in_reply_to_status_id").text.toLong)
        else
          None
      val inReplyToUserId =
        if ((node \ "in_reply_to_user_id").text != "")
          Some((node \"in_reply_to_user_id").text.toLong)
        else
```

```
            None
        val favorited = (node \ "favorited").text.toBoolean
        val user = User.fromXml((node \ "user")(0))
      }
    }
  }
```

其中最强大的一处是，它可以针对 Twitter 支持的其他任何格式进行扩展 — `fromXml` 方法可以在分解节点之前检查它是否保存了 XML、RSS 或 Atom 类型的内容，或者 `Status` 可以包含 `fromXml`、`fromRss`、`fromAtom` 和 `fromJson` 方法。实际上，后一种方法是我的优先选择，因为它会平等对待基于 XML 的格式和 JSON（基于文本）格式。

好奇和细心的读者会注意到在 `Status` 及其内嵌 `User` 的 `fromXml` 方法中，我使用的是 XPath 样式的分解方法，而不是之前建议的遍历内嵌元素的方法。现在，XPath 样式的方法看上去更易于阅读，但幸运的是，我后来改变了注意，良好的封装仍然是我的朋友 — 我可以在随后修改它，Scitter 外部的任何人都不会知道。

注意 `Status` 内部的两个成员如何使用 `Option[T]` 类型；这是因为这些元素通常排除在 `Status` 消息外部，并且虽然元素本身会出现，但它们显示为空（类似于 `<in_reply_to_user_id></in_reply_to_user_id>`）。这正是 `Option[T]` 的作用所在。当元素为空时，它们将使用 “None” 值。（这表示考虑到基于 Java 的兼容性，访问它们会更加困难，但惟一可行方法是对最终生成的 `Option` 实例调用 `get()`，这不太复杂并且能很好地解决 “非 null 即 O” 问题）。

现在已经可以轻而易举地使用公共时间轴：

清单 **14.** 分解公共时间轴

```
  @Test def simplePublicFeedPullAndDeserialize =
  {
    val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

    // HttpClient API 101
    val client = new HttpClient()
    val method = new GetMethod(publicFeedURL)
    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
      new DefaultHttpMethodRetryHandler(3, false))
    val statusCode = client.executeMethod(method)
    val responseBody = new String(method.getResponseBody())

    val responseXML = scala.xml.XML.loadString(responseBody)
    val statuses = responseXML \\ "status"

    for (n <- statuses.elements)
    {
      val s = Status.fromXml(n)
      System.out.println("\t'@" + s.user.screenName + "' wrote " + s.text)
    }
  }
```

显然，这看上去更加简洁，并且易于使用。

将所有这些结合到 Scitter 单一实例中相当简单，仅涉及执行查询、解析各个 `Status` 元素以及将它们添加到 `List[Status]` 实例中，如清单 15 所示：

清单 **15. Scitter.publicTimeline**

```
package com.tedneward.scitter
{
  import org.apache.commons.httpclient._, auth._, methods._, params._
  import scala.xml._

  object Scitter
  {
    // ...

    /**
     * Query the public timeline for the most recent statuses.
     *
     * Twitter docs say:
     * public_timeline
     * Returns the 20 most recent statuses from non-protected users who have set
     * a custom user icon.  Does not require authentication.  Note that the
     * public timeline is cached for 60 seconds so requesting it more often than
     * that is a waste of resources.
     * URL: http://twitter.com/statuses/public_timeline.format
     * Formats: xml, json, rss, atom
     * Method(s): GET
     * API limit: Not applicable
     * Returns: list of status elements
     */
    def publicTimeline : List[Status] =
    {
      import scala.collection.mutable.ListBuffer

      val client = new HttpClient()

      val method =
          new GetMethod("http://twitter.com/statuses/public_timeline.xml")

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      if (statusLine.getStatusCode() == 200)
      {
        val responseXML =
          XML.loadString(method.getResponseBodyAsString())

        val statusListBuffer = new ListBuffer[Status]

        for (n <- (responseXML \\ "status").elements)
          statusListBuffer += (Status.fromXml(n))
```

```
        statusListBuffer.toList
      }
      else
      {
        Nil
      }
    }
  }
}
```

在实现功能全面的 Twiter 客户机之前，我们显然还有很长的路要走。但到目前为止，我们已经实现基本的行为。

结束语

构建 Scitter 库的工作进展顺利；目前，Scitter 测试实现相对比较简单，与产生 Scitter API 的探索测试相比时尤为如此。外部用户不需要担心 Twitter API 或者它的各种格式的复杂性，虽然目前测试 Scitter 库有点困难（对单元测试而言，依赖网络并不是个好方法），但我们会及时解决此问题。

注意，我故意在 Twitter API 中维持了面向对象的感觉，秉承了 Scala 的精神 — 因为 Scala 支持大量功能特性并不表示我们要放弃 Java 结构采用的对象设计方法。我们将接受有用的功能特性，同时仍然保留适用的“旧方法”。

这并不是说我们在此处提供的设计是解决问题最好的方法，只能说这是我们决定采用的设计方法；并且，因为我是本文的作者，所以我采用的是自己的方式。如果不喜欢，您可以编写自己的库和文章（并将 URL 发送给我，我会在未来的文章中向您发起挑战）。事实上，在未来的文章中，我会将所有这些封装在一个 Scala “sbaz” 包中，并上传到网上供大家下载。

现在，我们又要暂时说再见了。下个月，我将在 Scitter 库中添加更多有趣的特性，并开始考虑如何简化它的测试和使用。

参考资料

学习

- “面向 *Java* 开发人员的 *Scala* 指南”（Ted Neward，developerWorks）：阅读整个系列。
- “Scala 和 XML”（Michael Galpin，developerWorks，2008 年 4 月）展示了 Scala 如何简化 XML 的使用。
- “Java 语言中的函数编程”（Abhijit Belapurkar，developerWorks，2004 年 7 月）：从 Java 开发人员的角度阐述功能编程的好处和用法。
- “Scala by Example”（Martin Odersky，2007 年 12 月）：简短、代码丰富的 Scala 简介（PDF）。
- *Programming in Scala*（Martin Odersky、Lex Spoon 和 Bill Venners；Artima，2007 年 12 月）：详细介绍 Scala 的长篇文章。
- Bjarne Stroustrup：设计和实现 C++。他称之为“更好的 C”。
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases*（Addison-Wesley Professional，2005 年 7 月）通过分析编程难题展示了 Java 编程语言的独特性。
- The developerWorks Java 技术专区：这里提供数百篇关于 Java 编程的文章。

获得产品和技术

- Jakarta (Apache) Commons HttpClient 组件 提供了一个有效的、全新的、特性丰富的包，它实现了最新的 HTTP 标准和和推荐规范的客户机。您可能还需要 Commons Logging 和 Commons Codec 组件。

- 下载 Scala：通过本系列开始学习它！
- SUnit：标准 Scala 发行版的一部分，位于 *scala.testing* 包。

讨论

- developerWorks blogs：加入 developerWorks 社区。

关于作者

Ted Neward 是 ThoughtWorks 的顾问，ThoughtWorks 是一家向全球提供咨询服务的公司。他还是 Neward & Associates 的主管，负责有关 Java、.NET 和 XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿西雅图附近。

对本文的评价

- (•) 太差！ (1)
- ( ) 需提高 (2)
- ( ) 一般；尚可 (3)
- ( ) 好文章 (4)
- ( ) 真棒！(5)

建议？

反馈意见

回页首

developerWorks 中国
本系列的更多信息:
面向 Java 开发人员的 Scala 指南
本文内容包括:
- POST 到 Twitter
- 终止和评价
- 更新
- 显示
- 撤销
- 结束语
- 下载
- 参考资料
- 关于作者
- 对本文的评价

相关链接:
- Java technology 技术文档库
- Open source 技术文档库

面向 Java 开发人员的 Scala 指南: 用 Scitter 更新 Twitter

将 **POST** 和 **DELETE** 操作添加到 **Scitter** 库

文档选项
打印本页
将此页作为电子邮件发送
样例代码
英文原文

级别： 中级

Ted Neward, 主管, ThoughtWorks

2009 年 12 月 07 日

Scitter 客户机库即将发布，但是还差最后一步。在这一期 面向 *Java* 开发人员的 *Scala* 指南 中，Ted Neward 展示如何将更新、显示和删除功能添加到用于访问 Twitter 的基于 Scala 的库中。

在撰写本文时，夏季即将结束，新的学年就要开始，Twitter 的服务器上不断涌现出世界各地的网虫和非网虫们发布的更新。对于我们很多身在北美的人来说，从海滩聚会到足球，从室外娱乐到室内项目，各种各样的想法纷至沓来。为了跟上这种形势，是时候重访 Scitter 这个用于访问 Twitter 的 Scala 客户机库了。

如果 到目前为止 您一直紧随 Scitter 的开发，就会知道，这个库现在能够利用各种不同的 Twitter API 查看用户的好友、追随者和时间线，以及其他内容。但是，这个库还不具备发布状态更新的能力。在这最后一篇关于 Scitter 的文章中，我们将丰富这个库的功能，增加一些有趣的内容（终止和评价）功能和重要方法 `update()`、`show()` 和 `destroy()`。在此过程中，您将了解更多关于 Twitter API 的知识，它与 Scala 之间的交互如何，您还将了解如何克服两者之间不可避免的编程挑战。

注意，当您看到本文的时候，Scitter 库将位于一个 公共源代码控制库 中。当然，我还将在本文中包括 源代码，但是要知道，源代码库可能发生改变。换句话说，项目库中的代码与您在这里看到的代码可能略有不同，或者有较大的不同。

关于本系列

Ted Neward 将和您一起深入探讨 Scala 编程语言。在这个 developerWorks 系列 中，您将了解有关 Scala 的所有最新讨论，并在实践中看到 Scala 的语言功能。在进行相关比较时，Scala 代码和 Java™ 代码将放在一起展示，但是（您将发现）Scala 与 Java 中的许多东西都没有直接的关联 — 这正是 Scala 的魅力所在！如果用 Java 代码就能够实现的话，又何必再学习 Scala 呢？

POST 到 Twitter

到目前为止，我们的 Scitter 开发主要集中于一些基于 `HTTP GET` 的操作，这主要是因为这些调用非常容易，而我想轻松切入 Twitter API。将 `POST` 和 `DELETE` 操作添加到库中对于可见性来说迈出了重要一步。到目前为止，可以在个人 Twitter 帐户上运行单元测试，而其他人并不知道您要干什么。但是，一旦开始发送更新消息，那么全世界都将知道您要运行 Scitter 单元测试。

如果继续测试 Scitter，那么需要在 Twitter 上创建自己的“测试”帐户。（也许用 Twitter API 编程的最大缺点是没有任何合适的测试或模拟工具。）

目前的进展

在开始着手这个库的新的 `UPDATE` 功能之前，我们来回顾一下到目前为止我们已经创建的东西。（我不会提供完整的源代码清单，因为 Scitter 已经开始变得过长，不便于全部显示。但是，可以在阅读本文时，从另一个窗口查看 代码。）

大致来说，Scitter 库分为 4 个部分：

- 来回发送的请求和响应类型（`User`、`Status` 等），包含在 API 中；它们被建模为 case 类。
- `OptionalParam` 类型，同样在 API 中的某些地方；也被建模为 case 类，这些 case 类继承基本的 `OptionalParam` 类型。
- `Scitter` 对象，用于通信基础和对 Twitter 的匿名（无身份验证）访问。

Twitter API

如果您不熟悉 Twitter API，那么有必要花几分钟看看 Twitter API Wiki 页面 http://apiwiki.twitter.com/REST+API+Documentation，了解更详细的信息。其基本原理很简单 — 在 URL 查询中传递参数，响应可采用 4 种格式之一（JSON、XML、ATOM 或 RSS），等等 — 但是，和所有 API 一样，细节往往非常重要，这里假设读者阅读本文时，已经在浏览器中打开了 Twitter API，以便将注意力集中到 Scala 上来。

- `Scitter` 类，存放一个用户名和密码，用于访问给定 Twitter 帐户时进行验证。

注意，在这最后一篇文章中，为了使文件大小保持在相对合理的范围内，我将请求/响应类型分开放到不同的文件中。

回页首

终止和评价

那么，现在我们清楚了目标。我们将通过实现两个“只读”Twitter API 来达到目标：`end_session` API（结束用户会话）和 `rate_limit_status` API（描述在某一特定时段内用户帐户还剩下多少可用的 post）。

`end_session` API 与它的同胞 `verify_credentials` 相似，也是一个非常简单的 API：只需用一个经过验证的请求调用它，它将“结束”当前正在运行的会话。在 Scitter 类上实现它非常容易，如清单 1 所示：

清单 **1.** 在 **Scitter** 上实现 **`end_session`**

```
package com.tedneward.scitter
{
  import org.apache.commons.httpclient._, auth._, methods._, params._
  import scala.xml._

  // ...
  class Scitter
  {
    /**
     *
     */
    def endSession : Boolean =
    {
      val (statusCode, statusBody) =
        Scitter.execute("http://twitter.com/account/end_session.xml",
          username, password)
```

```
      statusCode == 200

    }

  }

}
```

好吧，我失言了。也不是那么容易。

POST

和我们到目前为止用过的 Twitter API 中的其他 API 不一样，`end_session` 要求传入的消息是用 `HTTP POST` 语义发送的。现在，`Scitter.execute` 方法做任何事情都是通过 `GET`，这意味着需要将那些期望 `GET` 的 API 与那些期望 `POST` 的 API 区分开来。

现在暂不考虑这一点，另外还有一个明显的变化：`POST` 的 API 调用还需将名称/值对传递到 `execute()` 方法中。（记住，在其他 API 调用中，若使用 `GET`，则所有参数可以作为查询参数出现在 URL 行；若使用 `POST`，则参数出现在 HTTP 请求的主体中。）在 Scala 中，每当提到名称/值对，自然会想到 Scala `Map` 类型，所以在考虑建模作为 `POST` 一部分发送的数据元素时，最容易的方法是将它们放入到一个 `Map[String,String]` 中并传递。

例如，如果将一个新的状态消息传递给 Twitter，需要将这个不超过 140 个字符的消息放在一个名称/值对 `status` 中，那么应该如清单 2 所示：

清单 **2.** 基本 **`map`** 语法

```
val map = Map("status" -> message)
```

在此情况下，我们可以重构 `Scitter.execute()` 方法，使之用 一个 `Map` 作为参数。如果 `Map` 为空，那么可以认为应该使用 `GET` 而不是 `POST`，如清单 3 所示：

清单 **3.** 重构 **`execute()`**

```
  private[scitter] def execute(url : String) : (Int, String) =

      execute(url, Map(), "", "")

    private[scitter] def execute(url : String, username : String,

                                      password : String) : (Int, String) =

      execute(url, Map(), username, password)

    private[scitter] def execute(url : String,

                                      dataMap : Map[String,String]) : (Int, String) =

      execute(url, dataMap, "", "")

    private[scitter] def execute(url : String, dataMap : Map[String,String],

                                  username : String, password : String) =
```

```
{
  val client = new HttpClient()
  val method =
    if (dataMap.size == 0)
    {
      new GetMethod(url)
    }
    else
    {
      var m = new PostMethod(url)

      val array = new Array[NameValuePair](dataMap.size)
      var pos = 0
      dataMap.elements.foreach { (pr) =>
        pr match {
          case (k, v) => array(pos) = new NameValuePair(k, v)
        }
        pos += 1
      }
      m.setRequestBody(array)

      m
    }

  method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
    new DefaultHttpMethodRetryHandler(3, false))
```

```
    if ((username != "") && (password != ""))
    {
      client.getParams().setAuthenticationPreemptive(true)
      client.getState().setCredentials(
        new AuthScope("twitter.com", 80, AuthScope.ANY_REALM),
          new UsernamePasswordCredentials(username, password))
    }

    client.executeMethod(method)

    (method.getStatusLine().getStatusCode(), method.getResponseBodyAsString())
  }
```

`execute()` 方法最大的变化是引入了 `Map[String,String]` 参数，以及与它的大小有关的 "if" 测试。该测试决定是处理 `GET` 请求还是 `POST` 请求。由于 Apache Commons `HttpClient` 要求 `POST` 请求的主体放在 `NameValuePairs` 中，因此我们使用 `foreach()` 调用遍历 map 的元素。我们以二元组 *pr* 的形式传入 map 的键和值，并将它们分别提取到本地绑定变量 *k* 和 *v*，然后使用这些值作为 `NameValuePair` 构造函数的构造函数参数。

我们还可以使用 `PostMethod` 上的 `setParameter(name, value)` API 更轻松地做这些事情。出于教学的目的，我选择了清单 3 中的方法：以表明 Scala 数组和 Java 数组一样，仍然是可变的，即使数组引用被标记为 *val* 仍是如此。记住，在实际代码中，对于每个 (*k*,*v*) 元组，使用 `PostMethod` 上的 `setParameter(name, value)` 方法要好得多。

还需注意，对于 if/else 返回的 "method" 对象的类型，Scala 编译器会进行 *does the right thing* 类型推断。由于 Scala 可以看到 if/else 返回的是 `GetMethod` 还是 `PostMethod` 对象，它会选择最接近的基本类型 `HttpMethodBase` 作为 "method" 的返回类型。这也意味着，在 `execute()` 方法的其余部分中，`HttpMethodBase` 中的任何不可用方法都是不可访问的。幸运的是，我们不需要它们，所以至少现在没有问题。

清单 3 中的实现的背后还潜藏着最后一个问题，这个问题是由这样一个事实引起的：我选择了使用 `Map` 来区分 `execute()` 方法是处理 `GET` 操作，还是处理 `POST` 操作。如果还需要使用其他 HTTP 动作（例如 `PUT` 或 `DELETE`），那么将不得不再次重构 `execute()`。到目前为止，还没有这样的问题，但是今后要记住这一点。

测试

在实施这样的重构之前，先运行 `ant test`，以确保原有的所有基于 `GET` 的请求 API 仍可使用 — 事实确实如此。（这里假设生产 Twitter API 或 Twitter 服务器的可用性没有变化）。一切正常（至少在我的计算机上是这样），所以实现新的 `execute()` 方法就非常容易：

清单 **4. Scitter v0.3: endSession**

```
def endSession : Boolean =
  {
```

```
      val (statusCode, statusBody) =
        Scitter.execute("http://twitter.com/account/end_session.xml",
          Map("" -> ""), username, password)

      statusCode == 200
    }
```

这实在是再简单不过了。

接下来要做的是实现 `rate_limit_status` API，它有两个版本，一个是经过验证的版本，另一个是没有经过验证的版本。我们将该方法实现为 `Scitter` 对象和 `Scitter` 类上的 `rateLimitStatus`，如清单 5 所示：

清单 **5. Scitter v0.3: rateLimitStatus**

```
package com.tedneward.scitter
{
  object Scitter
  {
    // ...

    def rateLimitStatus : Option[RateLimits] =
    {
      val url = "http://twitter.com/account/rate_limit_status.xml"
      val (statusCode, statusBody) =
        Scitter.execute(url)
      if (statusCode == 200)
      {
        Some(RateLimits.fromXml(XML.loadString(statusBody)))
      }
      else
```

```
    {
      None
    }
  }
}


class Scitter
{
  // ...


  def rateLimitStatus : Option[RateLimits] =
  {
    val url = "http://twitter.com/account/rate_limit_status.xml"
    val (statusCode, statusBody) =
      Scitter.execute(url, username, password)
    if (statusCode == 200)
    {
      Some(RateLimits.fromXml(XML.loadString(statusBody)))
    }
    else
    {
      None
    }
  }
}
```

```
}
```

我觉得还是很简单。

回页首

更新

现在，有了新的 POST 版本的 HTTP 通信层，我们可以来处理 Twitter API 的中心：update 调用。毫不奇怪，需要一个 POST，并且至少有一个参数，即 status。

status 参数包含要发布到认证用户的 Twitter 提要的不超过 140 个字符的消息。另外还有一个可选参数：in_reply_to_status_id，该参数提供另一个更新的 id，执行了 POST 的更新将回复该更新。

update 调用差不多就是这样了，如清单 6 所示：

清单 **6. Scitter v0.3: update**

```
package com.tedneward.scitter

{

  class Scitter

  {

    // ...


    def update(message : String, options : OptionalParam*) : Option[Status] =

    {

      def optionsToMap(options : List[OptionalParam]) : Map[String, String]=

      {

        options match

        {

          case hd :: tl =>

            hd match {

              case InReplyToStatusId(id) =>

                Map("in_reply_to_status_id" -> id.toString) ++ optionsToMap(tl)

              case _ =>
```

```
                optionsToMap(tl)

            }

          case List() => Map()

        }

      }


      val paramsMap = Map("status" -> message) ++ optionsToMap(options.toList)


      val (statusCode, body) =

        Scitter.execute("http://twitter.com/statuses/update.xml",
           paramsMap, username, password)

      if (statusCode == 200)

      {

        Some(Status.fromXml(XML.loadString(body)))

      }

      else

      {

        None

      }

    }

  }

}
```

也许这个方法中最“不同”的部分就是其中定义的嵌套函数 — 与使用 GET 的其他 Twitter API 调用不同，Twitter 期望传给 POST 的参数出现在执行 POST 的主体中，这意味着在调用 Scitter.execute() 之前需要将它们转换成 Map 条目。但是，默认的 Map（来自 scala.collections.immutable）是不可变的，这意味着可以组合 Map，但是不能将条目添加到已有的 Map 中。（实际上，还是可以做到，但是我们不愿意那样做。请参阅侧边栏“可变集合”，了解更多这方面的信息。）

解决这个小难题的最容易的方法是递归地处理传入的 OptionalParam 元素的列表（实际上是一个 Array[]）。我们将每个元素拆开，将它转换成各自的 Map 条目。然后，将一个新的 Map（由新创建的 Map 和从递归调用返回的 Map 组成）返回到 optionsToMap。

可变集合

在 Scala 中可以使用可变集合，只需导入 scala.collections.mutable 包，而不是

然后，将 OptionalParam 的 Array[] 传递到 optionsToMap 嵌套函数。然后，将返回的 Map 与我们构建的包含 status 消息的 Map 连接起来。最后，将新的 Map 和用户名、密码一起传递给 Scitter.execute() 方法，以传送到 Twitter 服务器。

随便说一句，所有这些任务需要的代码并不多，但是需要更多的解释，这是比较优雅的编程方式。

> scala.collections.immutable。但是，这样做要冒常见的 使用可变数据 的风险，所以常规的 Scala 编程风格建议从其他不可变集合创建不可变集合。如果确实需要或想要使用可变集合，那么可以导入 scala.collections，然后用部分包名前缀按 mutable.Map 或 immutable.Map 的方式引用 Map（或其他）集合类型。这样可以避免在一个块中同时使用可变集合和不可变集合时出现混淆。

潜在的重构

理论上，传给 update 的可选参数与传给其他基于 GET 的 API 调用的可选参数将受到同等对待；只是结果的格式有所不同（结果是用于 POST 的名称/值对，而不是用于 URL 的名称/值对）。

如果 Twitter API 需要其他 HTTP 动作支持（PUT 和/或 DELETE 就是可能需要的动作），那么总是可以将 HTTP 参数作为特定参数 — 也许又是一组 case 类 — 并让 execute() 以一个 HTTP 动作、URL、名称/值对的 map 以及（可选）用户名/密码作为 5 个参数。然后，必要时可以将可选参数转换成一个字符串或一组 POST 参数。这些内容只需记在脑中就行了。

回页首

显示

show 调用接受要检索的 Twitter 状态的 id，并显示 Twitter 状态。和 update 一样，这个方法非常简单，无需再作说明，如清单 7 所示：

清单 **7. Scitter v0.3: show**

```
package com.tedneward.scitter

{

  class Scitter

  {

    // ...


    def show(id : Long) : Option[Status] =

    {

      val (statusCode, body) =

        Scitter.execute("http://twitter.com/statuses/show/" + id + ".xml",

                  username, password)

      if (statusCode == 200)

      {

        Some(Status.fromXml(XML.loadString(body)))
```

```
      }
      else
      {
        None
      }
    }
  }
}
```

还有问题吗？

另一种显示方法

如果想再试一下模式匹配，那么可以看看清单 8 中是如何以另一种方式编写 `show()` 方法的：

清单 **8. Scitter v0.3: show redux**

```
package com.tedneward.scitter
{
  class Scitter
  {
    // ...

    def show(id : Long) : Option[Status] =
    {
      Scitter.execute("http://twitter.com/statuses/show/" + id + ".xml",
          username, password) match
      {
        case (200, body) =>
          Some(Status.fromXml(XML.loadString(body)))
```

```
        case (_, _) =>

          None

      }

    }

  }

}
```

这个版本比起 if/else 版本是否更加清晰，这很大程度上属于审美的问题，但公平而论，这个版本也许更加简洁。（很可能查看代码的人看到 Scala 的“函数”部分越多，就认为这个版本越吸引人。）

但是，相对于 if/else 版本，模式匹配版本有一个优势：如果 Twitter 返回新的条件（例如不同的错误条件或来自 HTTP 的响应代码），那么模式匹配版本在区分这些条件时可能更清晰。例如，如果某天 Twitter 决定返回 400 响应代码和一条错误消息（在主体中），以表明某种格式错误（也许是没有正确地重新 Tweet），那么与 if/else 方法相比，模式匹配版本可以更轻松（清晰）地同时测试响应代码和主体的内容。

还应注意，我们还可以使用清单 8 中的方式创建一些局部应用的函数，这些函数只需要 URL 和参数。但是，坦白说，这是一种自找麻烦的解放方案，所以我不会采用。

回页首

撤销

我们还想让 Scitter 用户可以撤销刚才执行的动作。为此，需要一个 `destroy` 调用，它将删除已发布的 Twitter 状态，如清单 9 所示：

清单 **9. Scitter v0.3: destroy**

```
package com.tedneward.scitter

{

  class Scitter

  {

    // ...



    def destroy(id : Long) : Option[Status] =

    {

      val paramsMap = Map("id" -> id.toString())
```

```
      val (statusCode, body) =
        Scitter.execute("http://twitter.com/statuses/destroy/" + id.toString() + ".xml",
          paramsMap, username, password)
      if (statusCode == 200)
      {
        Some(Status.fromXml(XML.loadString(body)))
      }
      else
      {
        None
      }
    }
    def destroy(id : Id) : Option[Status] =
      destroy(id.id.toLong)
  }
}
```

有了这些东西，我们可以考虑将这个 Scitter 客户机库作为“alpha”版，至少实现一个简单的 Scitter 客户机。（按照惯例，这个任务就留给您来完成，作为一项“读者练习”。）

回页首

结束语

编写 Scitter 客户机库是一项有趣的工作。虽然不能说 Scitter 已经可以完全用于生产，但是它绝对足以用于实现简单的、基于文本的 Twitter 客户机，这意味着它已经可以投入使用了。要发现什么人可以使用它，哪些特性是需要的，从而使之变得更有用，最好的方法就是将它向公众发布。

我已经将本文和之前关于 Scitter 的文章中的代码作为第一个修订版提交到 Google Code 上的 Scitter 项目主页。欢迎下载和试用这个库，并告诉我您的想法。同时也欢迎提供 bug 报告、修复和建议。

您也无需受我的代码库的束缚。见证了之前三篇文章中进行的 Scitter 开发，您应该对 Twitter API 的使用有很好的理解。如果对于使用该 API 有不同的想法，那么尽管去做：抛开 Scitter，构建自己的 Scala 客户机库。毕竟，做做这些内部项目也是挺有乐趣的。

现在，我们要向 Scitter 挥手告别，开始寻找新的用 Scala 解决的项目。愿您从中找到乐趣，如果发现了用 Scala 编程的工作，别忘了告诉我!

回页首

下载

描述	名字	大小	下载方法
本文的源代码	j-scala10209.zip	812KB	**HTTP**

关于下载方法的信息

参考资料

学习

- "面向 *Java* 开发人员的 *Scala* 指南"（Ted Neward，developerWorks）：阅读整个系列，包括前两篇关于 Scitter 的文章。
- "面向 Java 开发人员的 Scala 指南：Scala 控制结构内部揭密"（Ted Neward，developerWorks，2008 年 3 月）：深入解释 Scala 的基本理念，即开发人员不像我们认为的那样经常需要可变状态。
- "面向 Java 开发人员的 Scala 指南：集合类型"（Ted Neward，developerWorks，2008 年 6 月）：了解更多关于 Scala 中的元组、数组和列表的知识。
- "Twitter API wiki"：想用 Twitter 做更多事情？从这里开始吧。
- "Scala by Example"（Martin Odersky，2007 年 12 月）：这是一篇简短的、代码驱动的 Scala 介绍性文章（PDF 格式）。
- Programming in Scala （Martin Odersky，Lex Spoon 和 Bill Venners，Artima，2007 年 12 月）：第一份 Scala 介绍，篇幅和一本书差不多。
- developerWorks Java 技术专区：这里有数百篇关于 Java 编程各个方面的文章。

获得产品和技术

- 下载 Scitter 库，现在位于 Google Code 上。
- Jakarta (Apache) Commons HttpClient 组件 提供一个有效的、最新的、功能丰富的包，这个包实现了最新的 HTTP 标准和建议的客户端。您可能还需要 Commons Logging 和 Commons Codec 组件。
- 下载 Scala：通过这个系列开始学习 Scala。
- SUnit：标准 Scala 发行版的一部分，在 *scala.testing* 包中。

讨论

- 加入 developerWorks 社区。

关于作者

Ted Neward 是 ThoughtWorks 的顾问，ThoughtWorks 是一家向全球提供咨询服务的公司。他还是 Neward & Associates 的主

管，负责有关 Java、.NET 和 XML 服务和其他平台的咨询、指导、培训和推介。他现在居住在华盛顿西雅图附近。

对本文的评价

- (•) 太差！ (1)
- ( ) 需提高 (2)
- ( ) 一般；尚可 (3)
- ( ) 好文章 (4)
- ( ) 真棒！(5)

建议？

反馈意见

回页首