Scala 语言概览

Release 第二版

Martin Odersky **Philippe Altherr**
Vincent Cremet **Iulian Dragos**
Gilles Dubochet **Burak Emir**
Sean McDirmid **Stéphane Micheloud**
Nikolay Mihaylov **Michel Schinz**
Erik Stenman **Lex Spoon** **Matthias Zenger**

February 27, 2015

CONTENTS

SCALA 语言概览 (AN OVERVIEW OF THE SCALA PROGRAMMING LANGUAGE)

Second Edition

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, Matthias Zenger

École Polytechnique Fédérale de Lausanne (EPFL)

1015 Lausanne, Switzerland

Technical Report LAMP-REPORT-2006-001

翻译：王玮

排版/校对：邓草原

2015 年 2 月

> “学 Scala 这种大型语言，速度不能太慢，否则学了后面忘了前面。速读一下这份文档有助于快速切入，这一点我有体会”。
>
> —孟岩

这份文档由王玮翻译完成，我做了校对，并用 reStructedText 和 sphinx 排版输出。本文对于理解 Scala 为什么会设计成这样非常有帮助。

自本文发表之后 Scala 的若干变化

需要说明的是，原文撰写于 2006 年，对应的大约是 Scala 2.0，少量内容跟现在的 Scala（截至 2.11.x）有些不同了，这些不同可以在这里找到：

http://www.scala-lang.org/download/changelog.html

跟本文有关的主要变化有：

AnyVal 子类的小写字母别名已被放弃 [1]

AnyVal（值类型）是用来对应在底层的宿主系统中（比如 JVM ），没有实现成引用对象的类型（比如 JVM 中的原生类型）。在 Scala 2.x 之前，为了和 Java 中的原生类型对应，它们都有一个全小写字母的别名，在 scala/Predef.scala 中定义为：

```
type byte    = scala.Byte
type short   = scala.Short
type char    = scala.Char
type int     = scala.Int
type long    = scala.Long
type float   = scala.Float
```

[1] Scala 2.8 及以前的源码树在 http://lampsvn.epfl.ch/trac/scala/browser/scala

```
type double  = scala.Double
type boolean = scala.Boolean
type unit    = scala.Unit
```

但从 Scala 2.7.2 开始，它们被标为废弃：

```
@deprecated("lower-case type aliases will be removed") type byte    = scala.Byte
@deprecated("lower-case type aliases will be removed") type short   = scala.Short
@deprecated("lower-case type aliases will be removed") type char    = scala.Char
@deprecated("lower-case type aliases will be removed") type int     = scala.Int
@deprecated("lower-case type aliases will be removed") type long    = scala.Long
@deprecated("lower-case type aliases will be removed") type float   = scala.Float
@deprecated("lower-case type aliases will be removed") type double  = scala.Double
@deprecated("lower-case type aliases will be removed") type boolean = scala.Boolean
@deprecated("lower-case type aliases will be removed") type unit    = scala.Unit
```

最后，从 Scala 2.8.0 开始，这些小写别名被全部移除。

For-comprehensions 的语法变化

For-comprehensions 的语法从 Scala 2.5 开始有了改变，例如：

```
for (val x <- List(1, 2, 3); x % 2 == 0) println(x)
```

现在要写成：

```
for (x <- List(1, 2, 3) if x % 2 == 0) println(x)
```

也即，生成器（generator）中的变量 x 前不再需要 val 关键字，而且可以直接跟一个 if 开头的守护子句（guarded）。

怎样自己制作本电子书

本电子书作为 sphinx 项目安家在

https://github.com/wecite/papers/tree/master/An-Overview-of-the-Scala-Programming-Language/book

制作的步骤为（以 linux 环境为例）：

安装 python 和 pip:

```
sudo yum install python
sudo easy_install pip
```

安装 sphinx:

```
sudo pip install sphinx
```

安装 texlive-scheme-small:

```
sudo yum install texlive-schema-small
```

安装其它 texlive 包，请检查下列包是否已安装，如果没有则需要安装:

- texlive-titlesec
- texlive-framed
- texlive-threeparttable
- texlive-wrapfig
- texlive-helvetic
- texlive-courier
- texlive-multirow
- texlive-upquote
- texlive-fandol

其中 texlive-fandol 中文字体包可能需要在安装完毕后，执行下列操作以注册字体:

```
cp /usr/share/texlive/texmf-dist/fonts/opentype/public/fandol/* ~/fonts/
fc-cache -fv
```

以上准备工作完成后，就可以自己制作本电子书了，步骤为:

```
git clone https://github.com/wecite/papers.git wecite.papers
cd wecite.papers/An-Overview-of-the-Scala-Programming-Language/book
make latex
cd build/latex
vi ScalaOverview.tex
```

因为是输出中文 PDF，这时需要把 ScalaOverview.tex 中以下两行删掉，否则会出现各种异况 (跟 xeCJK 包貌似有冲突):

```
\usepackage[utf8]{inputenc}
\DeclareUnicodeCharacter{00A0}{\nobreakspace}
```

最后，用 xelatex 将.tex 文件输出为 PDF:

```
xelatex ScalaOverview.tex
```

另外，你也可以直接制作 epub，html 等格式的输出，这个简单多了，不需要安装前面提到的 texlive 相关包，只需要:

```
cd wecite.papers/An-Overview-of-the-Scala-Programming-Language/book
make html
make epub
```

关于本电子书，您如果发现有任何错误和建议，可以直接到 github 上向该项目提出或者提交 pull-request。

邓草原 2015 年 2 月于北京

2008 年那会儿，Scala 刚刚冒头的样子，虽非默默无闻，但也远没有现在这样被人看好。当时我正好对 Scala 开始感兴趣，在学习的过程中，也看了很多资料和文章，其中这一篇相对比较喜欢。原因可能有些特殊，因为个人背景的因素，我一直是一个“理论派”，总喜欢“理论指导实践”，而这篇文章恰好是 Scala 发明者们所阐述的创建这门语言的动机和初始设计，包括很多理论基础，对于喜欢理论的人，读起来就有对这门语言“放心”的感觉。就内容而言，说实话，当时翻译到一半稍微有点后悔——感觉这篇文章的后半部分有点简略且凌乱，不如前半部分那样是充分构思过的文章，当然，也不排除是我本人阅读水平和习惯的问题。另外，时至今日，有些内容和 Scala 最新的发展对比起来，可能已经有点过时了，毕竟很多具体语法都已经有了变化。尤其是这几年互联网技术的发展和人们对软件开发领域的认识，说不定这篇文章一开始所描述的 Scala 语言的立意，都未必会让很多人认同。不过，这件事情不做完，总觉得心里不踏实，毕竟还曾经专门为此给 Martin 写了邮件，获得了人家的同意。因此，我还是坚持把最后一点工作完成，而对于有兴趣的人而言，我建议阅读此文时，重点去看其讲解的思路，而非某些具体的代码。另外不要忽视每一段内容，因为文中经常出现讲解某一方面内容的时候，穿插其他相关思路的说明。

王玮 2015 年 2 月于北京

《Scala 语言概览》(An Overview of the Scala Programming Language) 是瑞士洛桑联邦理工学院 (EPFL) 的程序设计实验室的 Scala 发明者们写的一篇技术报告，针对现行的 Scala 版本。由于要对这种语言进行比较完整的描述，篇幅又不太长，因此学术味有点浓，而且部分内容略显简略、杂乱。但是，我仍然感觉这篇文章是长期以来看到过的对一门语言介绍最完整、清晰的文章，不但让人对 Scala 有较为深入的了解，而且对编程语言设计、函数式/面向对象编程等领域的基本概念和最新进展都能够有所接触，是难得的文献。因此自然有了翻译过来的冲动，内容错漏难免，拿出来大家讨论而已。

王玮 2008 年 9 月于北京

Scala 将面向对象和函数式编程融合在一个静态类型语言中，其目标是组件或组件体系的开发。本文给出 Scala 语言的概览，主要面向具有编程方法和编程语言设计经验的读者。

CHAPTER

ONE

简介

真正的组件系统，一直是软件业一个近乎虚幻（elusive）的目标。从理想的角度看，软件应该可以使用组件库中预先设计好组件的构建，一如硬件可以用预制的芯片进行组装。然而现实中，任何软件中都有很大一部分是用“从头开始”（“from scratch”）的方式编写的，正因为如此，软件开发迄今为止仍然是一种工艺（Craft）而没有成为工业。

这里所说的组件是指软件的某种部件（parts），他们可以被更大的部件或整个应用以某种形式所调用；组件可以有很多形式，如：模块（modules）、类（classes）、框架（frameworks）、过程（processes）或 web services。其代码规模也可以从几行到成百上千行。他们可以与其他组件以很多方式相连接，包括聚合、参数传递、继承、远程调用及消息传递等。

我们始终认为，基于组件的软件开发一直以来缺乏进展，很大程度上是由于我们用来定义和集成各种组件的编程语言本身的缺陷。绝大部分编程语言为组件的抽象和编写（composite）提供的支持极为有限，这一点在 Java 或者 C# 这种静态类型语言中尤为明显，恰恰当今的大部分组件系统都是由他们写成的。

洛桑联邦理工学院（EPFL）的程序设计实验室从 2001 年开始研发 Scala，并于 2004 年 1 月发布了基于 JVM 的版本，6 月又发布了基于.NET 的版本。Scala 的第二版在 2006 年 3 月发布，这是一个改进的版本，也就是本文描述内容所基于的版本。

Scala 工作起源于研究更好地支持组件开发的编程语言，我们有两个假设希望通过 Scala 进行验证：首先，我们认为一种适合于组件系统的编程语言应该具有某种可扩展性（scalable），也就是说：相同的概念可以既适用于描述小型部件，也同样适用于描述大型部件。因此，我们专注于组件的抽象、组成和分解的机制，而非加入一整套基本语法结构来描述组件，因为这样做很可能在某种规模的组件层面上有效，而在另一个规模的层面上却无效。其次，我们认为可扩展的组件支持可以来自于这样一种编程语言：它能够统一并进一步泛化（generalize）面向对象编程和函数式编程这两种编程模式。迄今为止，在静态类型语言中（Scala 属于其中之一），这两个范式差距是很大的。（很显然这里所说“*Scala* 属于其中之一”并非是指 *Scala* 中这两种编程模式也存在很大差异，而是在表明把 *Scala* 设计

成为静态类型语言的意义所在——译注）

为验证我们的假设，需要能够真正用于设计组件和组件系统，因为只有得到用户社区的正式应用才能说明 Scala 语言所体现的这些概念是否在设计组件系统时真正有效。为了使用户更容易采用这种新语言，它必须能很好的与现有平台和组件相整合。因此 Scala 被设计成为能够与 Java 和 C# 很好的工作在一起，他采纳了这两种语言的主要语法和类型系统。当然，为了获得真正的进步，也必须抛弃很多固有的传统，这也就是为什么 Scala 并不是 Java 的超集：有些功能被去掉了，另一些概念被重新解释，以便获得一致性。

尽管 Scala 的语法刻意地遵守传统，但其类型系统至少在三方面有新的突破：首先，抽象类型定义和路径依赖类型（path-dependent types）将 νObj *Calculus* [36] 应用到了具体的语言设计上。其次，模块化的 mixin 组合方式（modular mixin composition）结合了 mixins 和 traits 的优势。第三，视图（views）的概念使组件得以按照模块化的方式被使用。

本文其余部分给出了 Scala 的概述，按照下列几个关键方面进行组织：

- Scala 程序与 Java 在很多方面十分相像，并且可以与 Java 程序无缝衔接（第 2 章）
- Scala 有一个统一的对象模型，这意味着所有的值都是对象，所有的操作都是方法调用（第 3 章）
- Scala 是一个函数式语言，这意味着函数是“一等公民”的值对象（first-class values）（第 4 章）
- Scala 有统一和强有力的抽象概念来描述类型和值（第 5 章）
- Scala 有灵活的，模块化的 mixin 组合构造用于编写类和特征（traits）(第 6 章)
- 它允许使用模式匹配来对对象进行分解（decomposition）（第 7 章）
- 模式（Pattern）和表达式被进一步泛化，用于对 XML 文档的自然处理（第 8 章）
- Scala 支持通过视图（views）对组件进行外部扩展（第 9 章）

最后第 10 章讨论相关工作，第 11 章是结论。

本文是 Scala 语言的高层概括，主要面向对程序设计语言有一定知识的读者，它既不是简化版的 Scala 语言参考，也不是一个教程。完整的语言参考可以参见《Scala 语言规范》[35]，教程则可参见 [34,18] 等。

CHAPTER

TWO

一种类似 JAVA 的语言

Scala 被设计成为可以与主流平台很好的交互，包括 Java 和 C#。因此其绝大部分的基本操作符、数据类型和控制逻辑都与这两种语言相同。为简单起见，后面我们仅将 Scala 与 Java 作对比，由于 Java 与 C# 本身就有很多共同点，因此 Scala 绝大部分与 Java 的相似之处都可直接应用于 C#。其实有些情况下 Scala 甚至更接近 C#，例如对泛型的处理方面。表 1 列出了 Java 和 Scala 实现同样功能的两段程序，都是打印出程序命令行包含的所有选项。

```
// Java
class PrintOptions {
  public static void main(String[] args) {
    System.out.println("Options selected:");
    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
        System.out.println(" " + args[i].substring(1));
  }
}

// Scala
object PrintOptions {
  def main(args: Array[String]): unit = {
    System.out.println("Options selected:")
    for (val arg <- args)
      if (arg.startsWith("-"))
        System.out.println(" " + arg.substring(1))
  }
}
```

表 1：Scala 与 Java 程序示例

这个示例显示出两种语言很多相似性：他们都有 String 这种基本类，调用同名的方法，使用同样的操作符、同样的条件控制结构等。示例同时也体现两种语言的差异，包括：

- Scala 除类定义之外，还有对象定义的概念（以 object 开头）。对象定义实际上是定义了一个只有单个实例的类，也就是通常所说的单例（singleton）。在上例中，PrintOptions 这个单例类有一个名为 main 的成员方法。
- Scala 采用“名称：类型”的方式进行变量定义和参数传递，而 Java 是使用类型前缀的形式，也就是“类型名称”。
- Scala 的语法比 Java 的相对更规范：所有的定义都由一个关键字开始，如上例中的 def main，开始定义一个方法。
- Scala 不强制要求语句结束使用分号，分号是可选的。（原文中 Java 语句也没有分号，明显是错漏——译注）
- Scala 没有特定语法来描述数组类型和数组的存取，一个元素类型为 T 的数组用 Array[T] 来表示。这里 Array 是一个标准的类，而 [T] 是一个类型参数。实际上，Scala 的数组是继承自函数的，因此数组存取形式就像函数调用 a(i)，而不是像 Java 的 a[i]。数组将在 4.3 节进一步讨论。
- main 函数的返回类型是 unit，Java 中是 void。这种处理源自于这样一个事实：Scala 中语句和表达式没有区别。每一个函数都返回一个值，如果一个函数的右侧是一个 block，那么这个 block 的最后一个表达式的求值将作为其返回值。unit 类型的返回值可能是一个平凡值 {}。其他的控制结构，如 if-then-else 等，也被泛化为表达式。
- Scala 采纳了 Java 绝大部分控制结构，但不包含传统的 for 语句。作为替代，Scala 包含 for-comprehensions，它允许直接遍历数组、列表、迭代器等数据结构而无需使用索引。Java 5.0 也包括了扩展的 for 循环，有点类似于 Scala 的 for-comprehensions，但要严格很多。

尽管语法上存在差异，Scala 程序和 Java 程序进行互操作没有任何困难。如上例所示，Scala 程序调用了 String 类的 startWith 和 substring 方法，而这里的 String 是一个 Java 类。它还访问了 Java 的 System 类的 out 这一静态属性，并且调用其经过重载的 println 方法。实际上 Scala 的类根本没有静态成员这一概念，但是上述调用仍然成为可能，这是因为每一个 Java 类在 Scala 中对应两个实体：一个是包含了所有动态成员的类，一个是包含了所有静态成员的单例对象。因此 System.out 这一属性是通过 Scala 的 System 这个单例对象来访问的。此外，上例中并未描述的是，Scala 的类/对象还能够继承 Java 类或实现 Java 定义的接口，这使得在 Java 框架中调用 Scala 代码成为可能。例如，一个 Scala 类可以实现 Java 的 java.util.EventListener 接口，从而使该 Scala 类

的实例可以接收到 Java 代码发出的事件。

CHAPTER

THREE

统一的对象模型

Scala 采用了一种纯粹的面向对象的模型，如同 Smalltalk 一样：每一个值都是对象，每一个操作都是消息传递。

3.1 类（Classes）

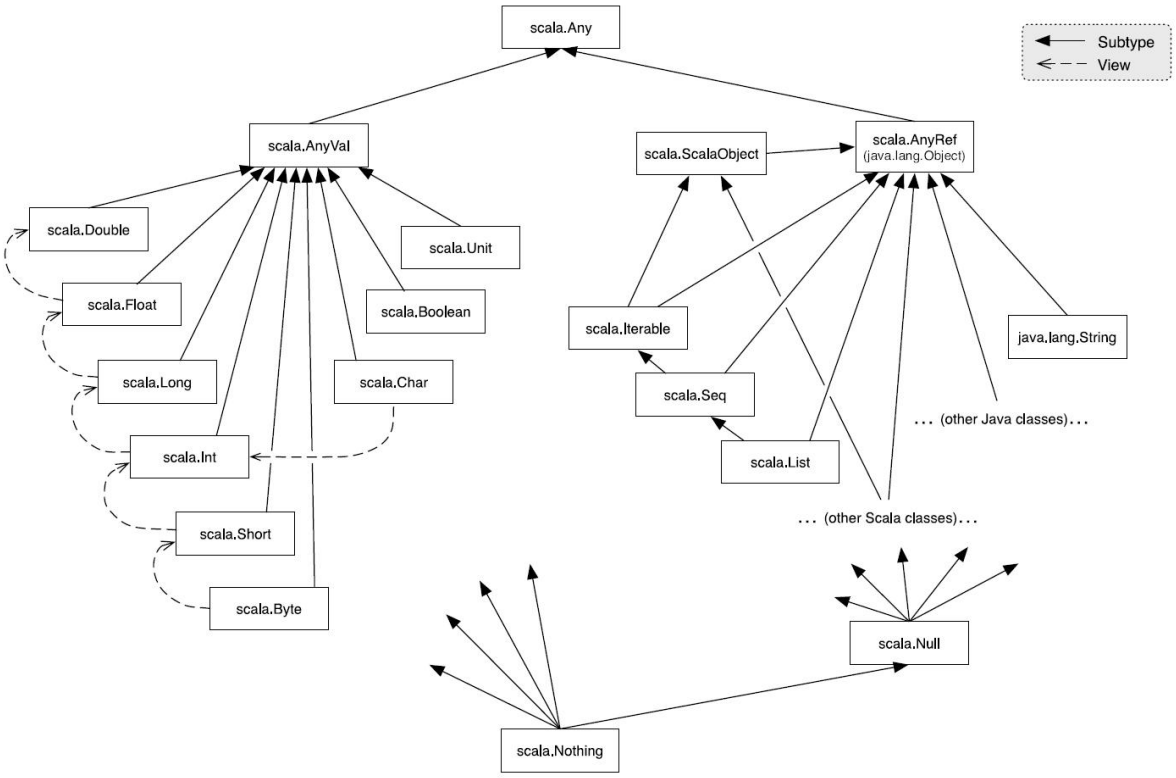

Figure 1: Class hierarchy of Scala.

图 1 (*原文为 Figure 2, 但图上是 Figure 1 ——译注*) 展示了 Scala 的类层次结构。每一个类都继承自 scala.Any，Any 的子类可以划分为两个主要范畴 (Categories)，值类型 (values classes) 继承自 scala.AnyVal，引用类型 (reference classes) 继承自 scala.AnyRef。每一种 Java 的基本数据类型都对应于一种值类型，通过预定义的类型别名进行映射，而 AnyRef 则对应于 Java 环境中的根类：java.lang.Object。引用类型的实例一般是通过指向堆中的一个对象的指针来实现的，而值类型则是直接表示的，不通过指针引用。两种类型对象可以相互转换，例如将一个值类型实例看做是根类 Any 的实例时，此时的装箱 (boxing) 操作及其逆操作是自动完成的，无需额外编码。

需要注意的是，值类型的类空间是平面的，即所有值类型继承自 scala.AnyVal，但他们相互之间没有继承关系。作为替代，值类型之间有视图 (即隐式类型转换，详见第 9 节)。我们曾经考虑另一种方式，即值类型之间相互继承，例如可以让 Int 类型继承自 Float，而不是采用隐式类型转换。最终我们没有这样选择，主要是因为我们希望保持一点：将一个值解释为一个子类型的实例时，其表现形式不应与将其解释为其父类型的实例时相异。此外，我们还希望保证：对于任意两个类型 S <: T (*S 是 T 的子类型——译注*)，S 的每一个实例都应当满足 [1]：

```
x.asInstanceOf[T].asInstanceOf[S] = x
```

由于浮点数是非精确表示的，因此类型转换有可能带来精度的损失，例如：目前 Scala 当中 Integer.MAX_VALUE-1=2147483646，这样的值经过 toFloat 、toInt 两次转换后就得到了 2147483647 ——译注

整个类层次结构的最底层有两个类型：scala.Null 和 scala.Nothing。Null 是所有引用类型的子类，它只有一个实例，就是对象 null。由于 Null 不是值类型的子类，所以 null 也不属于任何值类型，例如：将 null 赋值给一个 int 型变量是不可能的。Nothing 则是所有其他类型的子类，这个类没有任何实例。即便如此，它仍然可以作为类型参数而体现其存在价值。例如：Scala 的 library 定义了一个值 Nil，它是 List[Nothing] 的实例，由于 Scala 中 List 是协变 (covariant) 的，从而对于所有类型 T，Nil 都是 List[T] 的实例。

等词 ("==" 操作符) 被设计为与类型的表现无关。对于值类型，它表示自然意义 (数值、布尔值) 上的相等，对于引用类型，它实际上相当于 java.lang.Object 的 equals 方法的别名。该方法本身被定义为引用相等，但可以被子类重新实现，用于表示子类在自然意义上的相等概念。例如：装箱后的值类型可以重新实现 == 用于比较被装箱的数值。而在

[1] asInstaneOf 是 Scala 标准的类转换方法，在 Scala.Any 中定义

Java 中，== 对于引用类型永远表示引用相等，虽然这样实现起来很容易，但却带来了很严重的一致性问题：两个本来相等的值装箱后再用 == 比较，可能就不再相等了。

有些情况下需要使用引用相等而非自定义比较，例如 Hash-consing（Hash 构建），因为此时性能至关重要。对以这种情况，AnyRef 类型定义了另一个方法，eq，与 Java 的 "==" 相同，实现的是引用相等的比较，但不同的是它不能被子类重载。

3.2 操作（Operations）

Scala 统一对象模型的另一个方面体现为每一个操作都是一个消息传递，也就是说是一个方法调用。例如：x 与 y 相加操作 x + y 被解释为 x.+y，也就是调用 x 这个对象的方法 +，而 y 是该方法的参数。这种思想最早在 Smalltalk 中实现，在 Scala 中得到进一步改进，形成如下语法规约：首先，Scala 将操作符作为普通标识符，也就是说，任何标识符或者以字母开头的一串字符、数字形成，或者以一串操作符形成。因此我们可以定义诸如 +、<=、:: 等名称的方法。其次，Scala 将任何两个表达式之间的标识符视为一个方法调用，例如：前述列表 1 当中的代码中，我们可以用 (arg startsWith "-") 作为语法糖衣（syntactic sugar）来替代默认的用法 (arg.startsWith("-"))。下面用一个例子来说明用户自定义操作符如何声明和使用：一个表示自然数的类 Nat，它用 Zero 和 Succ 这两个类的实例来表示一个数字（当然很低效），每一个数字 N 用 **new** Succ^{N} (Zero) 来表示。我们先定义一个抽象类来描述自然数所支持的所有操作。根据 Nat 的定义，自然数有两个抽象方法：isZero、pred，和三个具体方法：succ、+、-。

```
abstract class Nat {
  def isZero: boolean
  def pred: Nat
  def succ: Nat = new Succ(this)
  def + (x: Nat): Nat =
    if (x.isZero) this else succ + x.pred
  def - (x: Nat): Nat =
    if (x.isZero) this else pred - x.pred
}
```

注意，Scala 允许定义无参数方法，这种方法一旦名字被引用到即会调用，无需传递参数列表。另外，Scala 类的抽象成员在语法上就通过没有定义来体现，无需添加 abstract 修饰符。

现在我们通过一个单例对象 Zero 和一个类 Succ 来扩展 Nat，分别表示 0 和非 0 的自然数。

```
object Zero extends Nat {
  def isZero: boolean = true
  def pred: Nat = throw new Error("Zero.pred")
  override def toString: String = "Zero"
}

class Succ(n: Nat) extends Nat {
  def isZero: boolean = false
  def pred: Nat = n
  override def toString: String = "Succ("+n+")"
}
```

Succ 类显示了 Scala 和 Java 的一些不同之处：Scala 中类的构造函数紧接着类的名称出现，不需要在类的定义体中出现与类同名的构造函数。这样的构造函数称为主构造函数（primary constructor），当一个主构造函数因为对象实例化而被调用时，整个类定义被调用。另外还存在次构造函数的语法定义，用于需要不止一个构造函数的情况，参见 [35] 的第 5.2.1 节。

Zero 对象和 Succ 类都实现了其父类 Nat 的两个抽象方法，同时还都覆盖了从 Any 继承来的 toString 方法。override 关键字在覆盖被继承类的具体方法时是必须的，而用于实现父类中的抽象方法时则可以省略。这个操作符给出足够的冗余用来避免两类错误：一个是意外覆盖，即子类并不是有意覆盖父类中的方法，此时编译器将给出没有 override 操作符的错误信息。另一种类型的错误是覆盖路径中断，即父类方法参数变了，但没有修改子类对应方法，此时 Scala 编译器会给出没覆盖任何方法的错误信息，而不是自动将这个子类方法视为重载（overloading）。

允许用户自定义中缀（infix）操作符引出一个问题，即他们的优先级和结合性（precedence and associativity）。一个解决方案是像 Haskell 或 SML 那样在定义每一个操作符时可以给出“结合度”（fixity），但是这种方式与模块化编程之间不能很好交互。Scala 采用一种相对简化的固定优先级与结合性的策略。每个中缀操作符由其第一个字符所决定，这与 Java 当中所有以非字母字符开头的操作符的优先级是一致的。下面是从低到高的操作符优先级：

```
(all letters)
|
^
&
< >
= !
:
```

```
+ -
* / %
(all other special characters)
```

操作符一般是左结合的，x + y + z 被解释为 (x + y) + z，唯一的例外是以冒号 (:) 结尾的操作符是右结合的。一个例子是列表构造 (list-consing) 操作符 ::，x :: y :: zs 被解释为 x :: (y :: zs)。右结合的操作符在方法方法查找上也是相反的，左结合操作符以其左方对象作为消息接收者，右结合操作符当然以右方对象为消息接收者。例如：x :: y:: zs 被视作 zs.::(y).::(x)。实际上，:: 是 Scala 的 List 类的一个方法，他将该方法参数对应的列表添加在接收消息的对象对应的列表的前面，并将合并成的新列表作为结果返回。某些 Scala 的操作符并不总对所有参数求值，例如标准布尔操作符 && 和 ||，这种操作符也可以是方法调用，因为 Scala 的参数是允许传名的。下面是一个 Bool 类，模拟系统内建的布尔类型。

```
abstract class Bool {
  def && (x: => Bool): Bool
  def || (x: => Bool): Bool
}
```

在这个类中，&& 和 || 的形参是 => Bool，里面的箭头表示实际参数以未求值的状态进行传递，即参数在每一次被引用的时候才求值 (也就是说这个参数实际上像一个无参数的函数一样)。这是 Bool 类型的两个典型 (canonical) 实例：

```
object False extends Bool {
  def && (x: => Bool): Bool = this
  def || (x: => Bool): Bool = x
}
object True extends Bool {
  def && (x: => Bool): Bool = x
  def || (x: => Bool): Bool = this
}
```

从上述实现可以看出，&& (或者相应地 ||) 操作，只有在左侧对象是 True (或相应地 False) 的时候，右侧对象才会被求值。如同本节所示，在 Scala 中可以把所有操作符定义为方法，每一个操作则是一个方法调用。为了性能需求，Scala 的编译器会把参数为值类型的操作直接编译为基本操作指令，但这对于编程者是完全透明的。

在前面的例子中，Zero 和 Succ 都继承一个类，这并不是唯一的可能性。在 Scala 中一个类或对象可以同时继承一个类以及若干个特征 (traits)，一个特征是一个抽象类，作用就

是用来与其它类组合。特征有时候类似于 Java 中的接口，可以用于定义一套抽象方法，用于被其他类实现。但不同的是 Scala 的特征可以有属性以及具体方法。

3.3 变量和属性（Variables and Properties）

如果所有操作都是方法调用，那么变量引用（dereferencing）和赋值语句呢？实际上，如果这两种操作是针对类成员变量，那么也是被解释为方法调用的。对于所有类成员变量 `var x: T`，Scala 这样定义其 *getter* 和 *setter*：

```
def x: T
def x_=(newval: T): unit
```

这些方法引用和更新一个可修改（mutable）的内存单元，它不能被 Scala 程序直接访问。每一次 x 这个名称被引用，都会导致调用 x 这个无参数方法，同样，每次调用赋值语句：`x = e`，都是 `x_=(e)` 这样一个方法调用。由于变量访问也是方法调用，从而使 Scala 可以定义类似 C# 的属性概念（properties），例如，下述 `Celsius` 类定义了一个属性 `degree`，只能设置大于 `-273` 的值：

```
class Celsius {
  private var d: int = 0
  def degree: int = d
  def degree_=(x: int): unit = if (x >= 273) d = x
}
```

使用者可以使用这两个方法，如同他们是一个类成员变量一样：

```
val c = new Celsius; c.degree = c.degree - 1
```

CHAPTER

FOUR

操作也是对象（OPERATIONS ARE OBJECTS）

Scala 是一种函数式编程语言，也就是说每一个函数都是一个值。Scala 有很简洁的语法用于定义匿名和 curry 化函数（curried function），以及嵌套函数等。

4.1 方法是函数式值（Methods are Functional Values）

为了演示如何将函数作为一个值使用，我们定义一个 exists 函数，用于检测一个数组当中是否有符合条件的元素：

```
def exists[T] (xs: Array[T], p: T => boolean) = {
  var i: int = 0
  while (i < xs.length && !p(xs(i))) i = i + 1
  i < xs.length
}
```

方法参数 [T] 表示数组类型是任意的（类型参数在 5.1 节中详细介绍），参数 p 表示验证条件也是任意的，p 的类型是函数类型（function type）T => boolean，表示所有定义域是 T 类型，值域是 boolean 的函数。函数参数可以像普通函数一样执行，如上面的循环体中显示的 p 被调用那样。以函数作为参数或者返回值的函数，称为高阶函数。

定义了 exists，我们可以通过双重否定来定义一个函数 forall，表示数组的所有元素没有一个不符合条件的，该函数定义如下：

```
def forall[T](xs: Array[T], p: T => boolean) = {
  def not_p(x: T) = !p(x)
  !exists(xs, not_p)
}
```

函数 forall 内部定义了一个嵌套函数 not_p，表示不满足条件 p。嵌套函数可以访问所在环境的函数参数和本地变量，例如 not_p 访问了 forall 的参数 p。

Scala 还可以定义一个没有名字的函数，例如下面这个简版的 forall 函数：

```
def forall_short[T](xs: Array[T], p: T => boolean) =
  !exists(xs, (x: T) => !p(x))
```

其中 (x: T) => !p(x) 定义了一个匿名函数（anonymous function），将类型为 T 的参数 p 映射为 !p(x)。

有了 exists 和 forall，我们可以定义一个函数 hasZeroRow，用以检验一个二维矩阵是否有一行全是 0：

```
def hasZeroRow(matrix: Array[Array[int]]) =
  exists(matrix, (row: Array[int]) => forall(row, 0 ==))
```

表达式 forall(row, 0 ==) 用于检测 row 是否只包含 0。这里，0 的 == 方法被作为参数传递给 forall 的参数 p，这显示了方法本身也是值，有点类似于 C# 中的“delegates”。

4.2 函数也是对象（Functions are Objects）

既然方法是值，值是对象，方法当然也就是对象。实际上，函数类型和函数值（注意：指函数本身作为值——译注）只不过是相应的类及其实例的语法糖衣。函数类型 S => T 等价于参数化类型 scala.Function1[S, T]，这个类型定义在 Scala 标准类库中：

```
package scala
abstract class Function1[-S, +T] {
  def apply(x: S): T
}
```

参数超过一个的函数也可类似地定义，一般而言，n-元函数类型：$(T1, T2, \cdots, Tn) => T$ 被解释为 Functionn[T1, T2, …, Tn, T]。也就是说，函数就是拥有 apply 方法的对象。例如，匿名函数“+1”：x: int => x+1，就是如下函数 Function1 的实例：

```
new Function1[int, int] {
  def apply(x: int): int = x + 1
}
```

反之，当一个函数类型的值被应用于参数之上（也就是调用——译注）时，这个类型的

apply 方法被自动插入，例如：对于 Function1[S，T] 类型的函数 p，p(x) 调用自然扩展为 p.apply(x)。

4.3 函数的具化（Refining Functions）

既然 Scala 中函数类型是类，那么也可以再细化成为子类。以 Array 为例，这是一种以整数为定义域的特殊函数。Array[T] 继承自 Function1[int，T]，并添加了数组更新、长度等方法：

```
package scala
class Array[T] extends Function1[int, T]
                  with Seq[T] {
  def apply(index: int): T = ...
  def update(index: int, elem: T): unit= ...
  def length: int = ...
  def exists(p: T => boolean): boolean = ...
  def forall(p: T => boolean): boolean = ...
  ...
}
```

赋值语句左侧的函数调用是存在特殊语法的，他们使用 update 方法。例如，a(i) = a(i) + 1 被翻译成：

```
a.update(i, a.apply(i) + 1)
```

将 Array 存取翻译成方法调用看上去代价比较高，但是 Scala 中的 inlining 变换可以将类似于上面的代码翻译成宿主系统的原生数组存取。上述 Array 类型还定义了 exists 和 forall 方法，这样也就不必手工定义了，使用这些方法，hasZeroRow 可以如下定义：

```
def hasZeroRow(matrix: Array[Array[int]]) =
  matrix exists (row => row forall (0 ==))
```

注意上述代码和相关操作的语言描述的对应性：“test whether in the *matrix* there *exists* a *row* such that in the *row all* elements are zeroes”（“检测一个矩阵，看看它是否有一行的所有元素都等于 *0*”。这里保留英语原文，因为原文中使用斜体部分对应于上述代码的内容，体现两种语法的对应关系——译注）。还要注意一点：在上面的匿名方法中，我们略去了参数 row 的类型，因为其类型可以被 Scala 编译器根据 matrix.exists 方法推断出来。

4.4 序列（Sequences）

高阶函数的使用是序列处理时的一个普遍特点。Scala 类库定义了几种不同类型的序列：数组（arrays）、列表（lists）、流（streams）和迭代器（iterators）。所有的序列都继承自特征 `trait Scala.Seq`；从而都定义了一套方法使得相应处理更简介、高效。例如：`map` 方法将一个函数应用于一个序列的所有元素，产生一个以相应结果为元素的序列。另一个例子是 `filter` 方法，将一个断言（predicate）函数应用于所有的元素，产生一个由所有使该断言为真的元素组成的序列。下面这个 `sqrts` 函数展示了上述两个方法的使用，它以一个双精度浮点数的列表 `xs` 为参数，返回一个以其所有非负元素的平方根为元素的列表：

```
def sqrts(xs: List[double]): List[double] =
  xs filter (0 <=) map Math.sqrt
```

注意，`Math.sqrt` 是一个 Java 函数，但它可以与 Scala 定义的函数一样的方式作为参数传递给高阶函数。

4.5 For Comprehensions

Scala 有一些特殊的语法用于更自然的表示某些特定的高阶函数的组合，其中 for comprehensions 就是对 Haskell 等语言中的 list comprehensions 进一步泛化。用 for comprehensions 写的 `sqrts` 如下：

```
def sqrts(xs: List[double]): List[double] =
  for (val x <- xs; 0 <= x) yield Math.sqrt(x)
```

这里，`val x <- xs` 是一个生成器（generator），产生一个序列，而 `0 <= x` 是一个过滤器（filter），从前者产生的序列中根据某些条件过滤掉一些元素。这个 comprehension 返回另一个由 `yield` 子句所产生的数值构成的序列。一个 comprehension 可以有多个生成器。

For comprehension 对应于 `map`、`flatMap` 和 `filter` 等高阶方法的组合，上面这个使用 for comprehension 实现的 `sqrts` 与 4.4 节那个实现是相同的。

For comprehension 的主要优势在于它并不受特定数据类型的限制，任何定义了 `map`、`flatMap` 和 `filter` 方法的类型都可以使用它，包括所有序列类型 [1]、选项值（optional values）和数据库接口以及一些其他类型。用户还可以将 for comprehension 用于自定义类型，只要定义和实现了相关方法。在 Scala 中，For 循环与 comprehensions 类似，对应于

[1] Arrays 尚未定义所有的序列的方法，因为其中的一些方法需要运行时的类型，而这尚未实现。

foreach 和 filter 的组合。例如：列表 1 当中的 for 循环：for (val arg <- args) ... 对应于 args foreach (arg => ...) 。

CHAPTER

FIVE

抽象（ABSTRACTION）

在组件系统中，一个重要的议题就是如何抽象所需的组件。编程语言当中有两种最主要的抽象方式：参数化和抽象成员。前一种主要是函数式抽象方式，而后一种主要是面向对象的方式。传统意义上，Java 对值提供函数式抽象，而对操作提供面向对象的抽象。Java 5.0 所支持的泛型，对类型也提供了一定的函数式抽象。Scala 对于值和类型提供上述两者抽象模式的统一支持，值和类型都可以作为参数，也可以作为抽象成员。本节对这两种模式进行讨论，并且对 Scala 类型系统的很大一部分进行回顾。

5.1 函数式抽象（Functional Abstraction）

下面这个类定义了一个可以读取和写入数值的单元（cell）：

```
class GenCell[T](init: T) {
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}
```

这个类用一个类型参数 T 抽象了 cell 的值的类型，因此我们称 GenCell 为*泛型*（*generic*）。与类相近，方法也可以有类型参数，下面这个 swap 方法交换两个 cell 的内容，只要它们包含的值类型相同：

```
def swap[T](x: GenCell[T], y: GenCell[T]): unit = {
  val t = x.get; x.set(y.get); y.set(t)
}
```

下面这段程序创建两个整数单元，并且交换它们的值：

```
val x: GenCell[int] = new GenCell[int](1)
val y: GenCell[int] = new GenCell[int](2)
swap[int](x, y)
```

参数的实际类型用方括号括起来，用于替代类和方法定义中的形式参数。Scala 定义了一套复杂的类型推理（type inference）系统，使得这两种情况下参数的实际类型都可以省略。类的方法和构造函数的类型参数对应的实际类型，可以通过局部类型推理（local type inference [41, 39]）根据预期的返回值以及参数类型推理出来。因此，上面的程序可以写成这种省略参数类型的方式：

```
val x = new GenCell(1)
val y = new GenCell(2)
swap(x, y)
```

参数限定（**Parameter bounds**）。考虑这样一个方法：updateMax，它将一个 cell 的值设置为其当前值与一个给定值之间较大的那个。我们希望这个函数能够作用于所有的 cell 类型，只要其值类型能够按照一个特征 trait Ordered 定义的“<”操作符进行比较。目前假定这个特征定义如下（更精确的定义在 Scala 标准类库中）：

```
trait Ordered[T] {
  def < (x: T): boolean
}
```

这样，updateMax 方法可以通过如下方式进行泛型定义，其中使用到的方法称为限定多态（Bounded polymorphism）：

```
def updateMax[T <: Ordered[T]](c: GenCell[T], x: T) =
  if (c.get < x) c.set(x)
```

这里，类型参数定义子句 [T <: Ordered[T]] 引入了受限类型参数 T，它限定参数类型 T 必须是 Ordered[T] 的子类型。这样，“<”操作符就可以应用于类型为 T 的参数了。同时，这个例子还展现出一个受限参数类型本身可以作为其限定类型的一部分，也就是说 Scala 支持 F-受限多态（F-bounded polymorphism [10]）。

协变性（**Variance**）。泛型和子类型（subtyping）组合在一起产生这样一个问题：它们如何相互作用。如果 C 是一个类型构造子（type constructor），S 是 T 的一个子类，那么 C[S] 是不是也是 C[T] 的子类呢？我们把有这种特性的类型构造子称为协变的（covariant）。可以看出 GenCell 这个类型构造子显然不是协变的，否则的话，下面这段代码就是合法的，但实际上它将会在运行时抛出错误：

```
val x: GenCell[String] = new GenCell[String]("abc")
val y: GenCell[Any] = x; // illegal!
y.set(1)
val z: String = y.get
```

GenCell 中的可变（mutable）变量使其无法成为协变的。实际上，GenCell[String] 不是 GenCell[Any] 的子类，因为有些可以针对 GenCell[Any] 的操作不能应用于 GenCell[String]，例如将其设置一个整型值。另一方面，对于不可变数据类型，构造子的协变性是很自然成立的。例如：一个不可变的整数列表自然可以被看做是一个 Any 列表的特例。此外，在另一些情况下我们正好需要逆协变性（contravariance），例如一个输出管道 Chan[T]，有一个以 T 为类型参数的写操作，我们自然希望对于所有 T <: S，都有 Chan[S] <: Chan[T]。

Scala 允许通过“+/-”定义类型参数的协变性，用“+”放在类型参数前表示构造子对于该参数是协变的，“-”则表示逆协变，没有任何符号则表示非协变。

下面的 GenList 定义了一个协变的列表，包含 isEmpty、head 和 tail 等三个方法。

```
abstract class GenList[+T] {
  def isEmpty: boolean
  def head: T
  def tail: GenList[T]
}
```

Scala 的类型系统通过跟踪类型参数的每一次使用来确保协变性确实成立。这些使用位置被分为几类：出现在不可变字段和方法返回结果被认为是协变的；出现在方法参数和类型参数上界时被认为是逆协变的；非协变的类型参数永远出现在非协变的位置；在一个逆协变类型参数的内部，协变与逆协变是反转的。类型系统保证协变（逆协变）的类型参数总是出现在协变（逆协变）的位置上。下面是 GenList 的两个实现：

```
object Empty extends GenList[Nothing] {
  def isEmpty: boolean = true
  def head: Nothing = throw new Error("Empty.head")
  def tail: GenList[Nothing] = throw new Error("Empty.tail")
}
class Cons[+T](x: T, xs: GenList[T]) extends GenList[T] {
  def isEmpty: boolean = false
  def head: T = x
  def tail: GenList[T] = xs
}
```

注意：Empty 对象代表一个空列表，其元素可以是任何类型。这一点就是由协变性保证的，因为 Empty 的类型是 GenList[Nothing]，对于任何 T 而言，它都是 GenList[T] 的子类型。

二元操作和参数下界（Binary methods and lower bounds）。 迄今为止，我们一直将协变性与不可变数据结构联系在一起，然而由于二元操作（Binary methods，就是指一个对象的方法，其参数类型也是这个对象类型，例如：x + y 这种——译注）的存在，这种做法并不完全正确。例如，为 GenList 类增加一个 prepend（前追加）方法，最自然的做法是将其定义成为接收一个相应的 list 元素类型参数：

```
abstract class GenList[+T] { ...
  def prepend(x: T): GenList[T] = // illegal!
    new Cons(x, this)
}
```

可惜这样做会导致类型错误，因为这种定义使得 T 在 GenList 中处于逆协变的位置，从而不能标记为协变参数（+T）。这一点非常遗憾，因为从概念上说不可变的 list 对于其元素类型而言应该是协变的，不过这个问题可以通过参数下界对 prepend 方法进行泛化而解决：

```
abstract class GenList[+T] { ...
  def prepend[S >: T](x: S): GenList[S] = // OK
    new Cons(x, this)
}
```

这里 prepend 是一个多态方法，接收 T 的某个父类型 S 作为参数，返回元素类型为 S 的 list。这个定义是合法的，因为参数下界被归类为协变位置，从而 T 在 GenList 中只出现在协变位置上。

与通配符模式相比较（Comparison with wildcards）。 Java 5.0 中可以提供一种通过通配符标记协变性的方法 [45]，这种模式本质上是 Igarashi 和 Viroli 提出的可变类型参数 [26] 的一种语法变体。与 Scala 不同的是，Java 5.0 的标注是针对类型表达式而不是类型定义。例如：在每一个需要用到协变的 generic list 的时候，都将其声明为 GenList<? extends T>，这是一个类型表达式，表示其所声明的对象实例的所有元素都是 T 的子类型。协变通配符可以用于任何类型表达式当中，但是要注意，出现在非协变的位置上的类型成员将会被忽略（forgotten），这对于保证类型的正确性是必须的。例如：GenCell<? extends Number> 类型只有那个 get 方法（返回 Number 类型）才有效，而其 set 方法，由于其类型参数是逆协变的，会被忽略。

在 Scala 的早期版本中，我们也实验过在调用时标注协变性的方式，类似于通配符。初看之下，这种方式有很大的灵活性，因为一个类型的成员既可以是协变的，也可以是非协变的，用户可以根据情况选择是不是使用通配符。但是，这种灵活性也是有代价的，因为这样作要有用户而不是设计者来保证对协变性的使用是一致的。在实践中我们发现，调用时标注协变性的方式很难保证一致性，经常会出现类型错误。相反，定义时标注协变性对于正确地设计类型有很大帮助，例如可以很好地指导人们设计方法时知道哪些应当使用参数下界。另外，Scala 的 mixin 合成（见第 6 章）可以让人很容易将一个类分成协变的和非协变的部分，而在 Java 这种单根结构 + 接口的继承模型中，这样做是非常麻烦的。因此，Scala 的新版本将标注协变性的方式从使用时标注改为了定义时标注。

5.2 抽象成员（Abstraction Members）

在 Scala 中，面向对象的抽象方式可以与函数式抽象起到同样的作用。例如：下例是一个以 OO 抽象方式定义的 Cell 类型。

```
abstract class AbsCell {
  type T
  val init: T
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}
```

AbsCell 类既没有类型参数也没有值参数，而是定义了一个抽象类型成员 T 和一个抽象值成员 init。我们可以通过给出这两个成员的具体定义来对这个类型进行实例化。例如：

```
val cell = new AbsCell { type T = int; val init = 1 }
cell.set(cell.get * 2)
```

这里，cell 的类型是 AbsCell { type T = int }，也就是 AbsCell 被 { type T = int } 细化（refinement）而形成的类型。访问 cell 值的代码认为其类型别名 cell.T=int，因此上面第二条语句是合法的。

路径依赖类型（Path-dependent types）。 不知道 AbsCell 绑定的类型情况下，也可以对其进行访问。下面这段代码将一个 cell 的值恢复成为其初始值（init），而无需关心 cell 值的类型是什么。

```
def reset(c: AbsCell): unit = c.set(c.init)
```

为什么可以这样做呢？因为 c.init 的类型是 c.T，而 c.set 是 c.T => unit 类型的函数，因此形参与实参类型一致，方法调用是类型正确的。c.T 是一个路径依赖类型的例子，通常来讲，这种类型的形式是：$x1. \ldots .xn.t$，这里 $n > 0$，$x1, \ldots, xn$ 是不可变的值，而 t 是 xn 的类型成员。路径依赖类型是 Scala 的一个新颖的特性，其理论基础是 νObj *calculus* [36]。路径依赖类型要依靠其前缀路径的不可变性，下面给出一个违反了不可变性的例子：

```
var flip = false
def f(): AbsCell = {
  flip = !flip
  if (flip) new AbsCell { type T = int; val init = 1 }
  else new AbsCell { type T = String; val init = "" }
}
f().set(f().get) // illegal!
```

在上例中，每一次调用 f() 分别返回 int 和 String 类型的值，因此最后一句是错误的，因为它要将 String 类型的值赋给一个 int 值的 cell。Scala 类型系统禁止这种调用，因为 f().get 的类型是 f().T，而这不是一个有效类型，因为 f() 不是一个有效路径。

类型选择与单例类型（**Type selection and singleton types**）。在 Java 中，类型定义可以嵌套，嵌套类型用其外部类型做前缀的形态表示。在 Scala 中，则通过“外部类型 # 内部类型”（Outer#Inner）的方式来表示，“#”就称作类型选择（Type Selection）。从概念上说，这与路径依赖类型（例如：p.Inner）不同，因为 p 是一个值，不是一个类型。进一步而言，Outer#t 也是一个无效表达式，如果 t 是一个定义在 Outer 中的抽象类型的话。实际上，路径依赖类型可以被扩展成为类型选择，p.t 可以看做是 p.type#t，这里 p.type 就称作单例类型，仅代表 p 所指向对象的类型。单例类型本身对于支持方法调用串接很有作用，考虑如下代码：C 有一个 incr 方法，对其值 +1，其子类 D 由一个 decr 方法，对其值 -1。

```
class C {
  protected var x = 0
  def incr: this.type = { x = x + 1; this }
}
class D extends C {
  def decr: this.type = { x = x - 1; this }
}
```

从而我们可以将相关调用串接起来：

```
val d = new D; d.incr.decr
```

如果没有 this.type 这个单例类型，上述调用是非法的，因为 d.incr 的类型应该是 C，但 C 并没有 decr 方法。从这个意义上说，this.type 类似于 Kim Bruce 的 mytype [29] 的一个协变的使用方式。

族多态和 self 类型（**Family polymorphism and self types**）。 Scala 的抽象类型概念非常适合于描述相互之间协变的一族（families）类型，这种概念称作族多态。例如：考虑 publish/subscribe 模式，它有两个主要类型：subjects 和 observers。Subjects 定义了 subscribe 方法，用于给 observers 进行注册，同时还有一个 publish 方法，用于通知所有的注册者；通知是通过调用所有注册者的 notify 方法实现的。一般来说，当 subject 的状态发生改变时，会调用 publish 方法。一个 subject 可以有多个 observers，一个 observer 也可以观察多个 subject。subscribe 方法一般用 observer 的标识为参数，而 notify 方法则以发出通知的 subject 对象为参数。因此，这两个类型在方法签名中都引用到了对方。这个模式的所有要素都在如下系统中：

```
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  abstract class Subject requires S {
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }

  trait Observer {
    def notify(sub: S): unit
  }
}
```

顶层的 SubjectObserver 类包含两个类成员：一个用于 subject，一个用于 observer。Subject 类定义了 subscribe 方法和 publish 方法，并且维护一个所有注册的 observer 的列表。Observer 这个 trait 只定义了一个抽象方法 notify。需要注意的是，Subject 和 Observer 并没有直接引用对方，因为这种“硬”引用将会影响客户代码对这些类进行协变的扩展。相反，SubjectOberver 定义了两个抽象类型 S 和 O，分别以 Subject 和 Observer 作为上界。Subject 和 Observer 的类型分别通过这两个抽象类型引用对方。另

外还要注意，Subject 类使用了一个特殊的标注 requires：

```
abstract class Subject requires S { ...
```

这个标注表示 Subject 类只能作为 S 的某个子类被实例化，这里 S 被称作 Subject 的 self-type。在定义一个类的时候，如果指定了 self-type，则这个类定义中出现的所有 this 都被认为属于这个 self-type 类型，否则被认为是这个类本身。在 Subject 类中，必须将 self-type 指定为 S，才能保证 obs.notify(this) 调用类型正确。Self-type 可以是任意类型，并不一定与当前正在定义的类型相关。依靠如下两个约束，类型正确性仍然可以得到保证：(1) 一个类型的 self-type 必须是其所有父类型的子类，(2) 当使用 new 对一个类进行实例化时，编译器将检查其 self-type 必须是这个类的父类。这个 publish/subscribe 模式中所定义的机制可以通过继承 SubjectObserver，并定义应用相关的 Subject 和 Observer 类来使用。例如下面的 SensorReader 对象，将传感器（sensors）作为 subjects，而将显示器（displays）作为 observers。

```
object SensorReader extends SubjectObserver {
  type S = Sensor
  type O = Display
  abstract class Sensor extends Subject {
    val label: String
    var value: double = 0.0
    def changeValue(v: double) = {
      value = v
      publish
    }
  }

  class Display extends Observer {
    def println(s: String) = ...
    def notify(sub: Sensor) =
      println(sub.label + " has value " + sub.value)
  }
}
```

在这个对象中，S 被 Sensor 限定，而 O 被 Display 限定，从而原先的两个抽象类型现在分别通过覆盖而获得定义，这种“系绳节”（“tying the knot”）在创建对象实例的时候是必须的。当然，用户也可以再定义一个抽象的 SensorReader 类型，未来再通过继承进行实例化。此时，这两个抽象类型也可以通过抽象类型来覆盖，如：

```
class AbsSensorReader extends SubjectObserver {
  type S <: Sensor
  type O <: Display
  ...
}
```

下面的代码演示了 SensorReader 如何使用：

```
object Test {
  import SensorReader._
  val s1 = new Sensor { val label = "sensor1" }
  val s2 = new Sensor { val label = "sensor2" }
  def main(args: Array[String]) = {
    val d1 = new Display; val d2 = new Display
    s1.subscribe(d1); s1.subscribe(d2)
    s2.subscribe(d1)
    s1.changeValue(2); s2.changeValue(3)
  }
}
```

另外值得注意的是其中的 import 语句，它使 Test 可以直接访问 SensorReader 的成员，而无需前缀。Scala 的 import 比 Java 中用法更广泛，可以在任何地方使用，可以从任何对象中导入成员，而不仅仅从一个 package 中。

5.3 用抽象类型建立泛型模型（Modeling Generics with Abstract Types）

一种语言里有两套抽象语法体系肯定会让人产生对这种语言复杂性的疑问：能不能就用一种形式化体系来实现？本节当中我们将会展示，函数式的类型抽象机制（也就是泛型）实际上可以通过面向对象的类型抽象机制（也就是抽象类型）来表达。这种表达方式的思路如下所述：假定一个参数化类型 C 有一个类型参数 t（可以直接推广到多个类型参数的情况），那么这种表达方式有四个关键组成部分：分别是类型自身的定义、类型实例的创建、基类构造子的调用以及这个类的类型实例（type instances）。

1. 类型定义，C 的定义可以重写如下：

```
class C {
  type t
  /* rest of class */
}
```

也就是说，C 的类型参数可以用其抽象成员来重新定义。如果类型参数有上界或者下界，则可以带到抽象成员的定义上。类型参数的协变性则不带到抽象成员的定义上，参见第 4 点。

2. 以 T 为参数创建实例的调用：new C[T] 可以写成：

```
new C { type t = T }
```

3. 如果 C[T] 出现在调用基类构造符的场合，则其子类的定义将会进行如下扩充：

```
type t = T
```

4. 每一个 C[T] 形式的类型定义都被扩充为如下的细化形式：

   - C { type t = T } 如果 t 被声明为非协变
   - C { type t <: T } 如果 t 被声明为协变
   - C { type t >: T } 如果 t 被声明为逆协变

这种表达方式在一种情况下会有问题：命名冲突。这是因为参数的名称成为了类的成员，可能和其他成员冲突，包括其父类的类型参数转化成的成员。这种冲突可以通过重命名解决，例如给每个类型名称指定一个唯一数字标识。

两种抽象模式之间可以转换，对于一种语言还是有价值的，因为可以降低其内在的概念复杂性。例如，Scala 的泛型，实际上就是一种语法糖，完全可以被抽象类型替代掉。既然如此，也许会有人问，这种语法糖有没有必要性？或者说为什么不只用抽象类型呢，这样可以使语法本身简化很多。实际上，Scala 中引入泛型有两重意义：首先，手工把泛型转化为成为抽象类型表达形式并不那么简单，不仅会丧失语法的简洁性，而且还可能带来前述的命名冲突等问题。其次，泛型和抽象类型在 Scala 中一般扮演不同的角色，泛型一般用于类型的实例化，而抽象类型主要用于在调用者代码中对相应的抽象类型进行引用。后者主要来自于两个场合：一个是有人需要在客户代码中隐藏相关类型信息，用于构造类似于 SML 模式的模块系统。另一个是在子类中协变地继承父类的类型，从而获得族多态。

可能有人会问，那么是否可以反过来用泛型来替代抽象类型呢？一些对于两种抽象方式都支持的系统进行的研究 [27] 证实，这样做要困难得多，至少整个程序都需要重写。不仅如此，如果系统要实现受限多态的话，重写类型上/下界的部分会呈平方级增长 [8]。实际上

这一点也不奇怪，因为这两种类型体系的理论基础就不同，泛型（不带 F-界的）可以用 $\mathrm{F}_{<:}$ 系统来表达 [11]，而抽象类型则建立在类型依赖的基础之上。后者比前者的表现力更强，例如，带路径依赖类型的 νObj 演算是可以涵盖 $\mathrm{F}_{<:}$ 的。

CHAPTER

SIX

构成（COMPOSITION）

解释了 Scala 的类型抽象体系之后，本节主要描述类的构成方式。Scala 的基于混入的类构成（mixin class composition）体系是 Brach [6] 中的面向对象的线性混入构成（linear mixin compostion）和 [14、25] 中提出的更加对称的混入模块（mixin modules），以及 traits [42] 这三者的融合。我们先看一个例子，如下这个迭代器的抽象描述：

```
trait AbsIterator[T] {
  def hasNext: boolean
  def next: T
}
```

注意上面出现的关键字 trait。Trait 是一种特殊的抽象类，它的构造方法没有任何值参数。Traits 可以出现任何抽象类可以出现的地方，但反之不然，只有 traits 可以用于混入。下面，我们用一个 trait 继承自 AbsIterator，并增加一个方法 foreach，用于将一个函数作用于该迭代子返回的每一个元素上。

```
trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => unit): unit =
    while (hasNext) f(next)
}
```

下面是一个具体的迭代子类定义，用于连续返回一个字符串的每一个字符：

```
class StringIterator(s: String) extends AbsIterator[char] {
  private var i = 0
  def hasNext = i < s.length
  def next = { val x = s charAt i; i = i + 1; x }
}
```

混入式类构成（Mixin-class composition）。下面我们将 RichIterator 和 StringIt-

erator 的功能合并在一个类中。只有单根继承和接口的情况下这是不可能的，因为这两个类都有具体的实现代码。因此，Scala 提供了混入式类构成的机制，使程序设计者可以重用一个类的增量内容，也就是非继承的内容。这种机制使人可以将 RichIterator 和 StringIterator 合并，在如下所示的例子将一个字符串的所有字母打成一列。

```
object Test {
  def main(args: Array[String]): unit = {
    class Iter extends StringIterator(args(0))
                  with RichIterator[char]
    val iter = new Iter
    iter foreach System.out.println
  }
}
```

Iter 类通过 RichIterator 和 StringIterator 这两个父类型混入构成，第一个父类型仍然称为超类（superclass），第二个父类型则称为混入（mixin）。

类的全序化（**Class Linearization**）。混入式类构成是多重继承的一种形式，因此也会面临单继承所没有的问题。最典型的就是：如果多个父类型定义了同名的成员，哪一个成员被继承？调用父类方法时那一个成员被引用？如果一个类从多个路径被继承了怎么办？在 Scala 中，解决这些问题的基础构造就是类的全序化（class linearization）。（*linearization 可以翻成线性化或者全序化，在计算机领域一般取后者。另外，后面大部分情况下用"全序"来替代，表示全序化的结果——译注*）

一个类 C 所直接继承的类形成的可递闭包当中所有类称为 C 的基类（base classes）。由于有混入类，一个类与它的基类之间的继承关系，构成一个有向无环图（directed acyclic graph）。C 的全序 $L(C)$ 是 C 的所有基类的一个全排序（total order），根据如下规则构成：假设 C 的定义为：

```
class C extends B0 with ... with Bn { ... } .
```

这个全序以 C 的基类 $B0$ 的全序为最后一部分，前面是 $B1$ 的全序（排除掉已经包含在 $B0$ 的全序当中的类），再前面是 $B2, \ldots, Bn$，同样排除掉前面已经出现过的类。最前面的是类 C 本身，作为这个全序的头一个类。例如，Iter 类的全序是：

```
{ Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any }
```

类的全序对于类的继承关系而言是一种改进：如果一个类 C 是 D 的子类，则在任何同时继承 C、D 的类的全序中，C 永远出现在 D 之前。全序化还满足另一个性质：一个类的全序永远包括其基类的全序作为后缀。例如，StringIterator 的全序化：

```
{ StringIterator, AbsIterator, AnyRef, Any }
```

就是其子类 Iter 的全序的后缀。不过对于混入类，这个性质并不成立，一个混入类的全序当中的类，在其子类的全序当中可能以不同的顺序出现，也就是说，Scala 中全序化不是单调（monotonic [1]）的。

成员（**Membership**）。如前所示，Iter 类从 StringIterator 和 RichIterator 同时继承了类成员（members）。简单而言，一个类以混入构成方式继承 Cn **with**⋯**with** $C1$，将会继承其中所有类的成员，同时还可以自定义新的成员。由于 Scala 保留了 Java 和 C# 的静态重载机制，因此可能从父类继承同名的方法，也可以再定义同名的方法 [1]。为了判断类 C 的一个方法到底是覆盖父类中的同名方法，还是这两个方法并存——即重载的关系，Scala 采用了匹配（matching）法，这也是从 Java 和 C# 中类似的概念衍生来的：简单地说，如果两个类成员同名，并且具有相同的参数类型（如果两个都是方法），就称之为相匹配。

一个类的成员总共两种类型——具体和抽象的，每种类型分别对应一个判定规则：

- 一个类 C 的具体成员是指其或其父类的所有具体声明 M，除非在其某个父类（也就是在 $L(C)$）中已有一个匹配的具体成员。
- 一个类 C 的抽象成员是指其或其父类的所有抽象声明 M，除非在 C 中已有一个匹配的具体成员，或者其某个父类（也就是在 $L(C)$）中有一个匹配的抽象成员。

这些规则同样决定了一个类 C 与其父类之间匹配成员的覆盖关系。首先，具体成员一定覆盖抽象成员。其次，如果 M 和 M' 同为具体成员或抽象成员，且 M 在 C 的全序化当中出现在 M' 之前，则 M 覆盖 M'。

父类调用（**Super Calls**）。我们考虑设计一个同步迭代器，也就是其操作在多线程之间是互斥的。

```
trait SyncIterator[T] extends AbsIterator[T] {
  abstract override def hasNext: boolean = synchronized(super.hasNext)
  abstract override def next: T = synchronized(super.next)
}
```

想要构造一个针对 String 的 Rich SyncIterator，可以用这三个类构成：

```
StringIterator(someString) with RichIterator[char] with SyncIterator[char]
```

[1] 有人可能反对这种设计方式，认为这样太复杂，但是为了保证互操作性，这样做是必须的，例如一个 Scala 类继承一个 Java Swing 类的时候。

这个构成类从 SyncIterator 继承了 hasNext 和 next，这两个方法都是对其父类的相应方法调用加了一个 sychronized() 包装。由于 RichIterator 和 SyncIterator 定义的方法相互不重合（*原文是“RichIterator”和“StringIterator”，应该有误——译注*），因此它们出现在 mixin 中的顺序没有影响，即上例写成这样也是等价的：

```
StringIterator(someString) with SyncIterator[char] with RichIterator[char]
```

但是，这里有一个小细节要注意：在 SyncIterator 中的 super 这个调用并不是静态地绑定到其父类 AbsIterator 上，因为显然这是毫无意义的，AbsIterator 定义的 next 和 hasNext 都是抽象方法。实际上，这个 super 调用实际上指向这个 mixin 构成中的 superclass：StringIterator 的相应方法。从这个意义上讲，一个 mixin 构成的 superclass 覆盖了其各个 mixin 当中静态声明的超类。这也就意味着 super 调用在一个类当中无法被静态解析，必须延迟到一个类被实例化或被继承的时候才能解析出来。这一概念有如下精确定义：

假设 C 是 D 的父类，在 C 当中的表达式 $\mathbf{super}.M$ 应该能够静态解析为 C 的某个父类当中的成员 M，这样才能保证类型正确。而在 D 的语境中，这个表达式应该表示一个与 M 相匹配的 M'，这个成员应该在 D 的全序当中位于 C 之后的某个类里定义。

最后注意一点：在 Java 或 C# 等语言中，上述 SyncIterator 当中的这种 super 调用明显是不合法的，因为它会被指派为父类当中的抽象成员（方法）。如同我们在上面看到的，这种构造在 Scala 中是合法的，只要保证一个前提，那就是这个类所出现的语境当中，其 super 调用所访问的父类成员必须是具体定义了的。这一点是由 SyncIterator 当中的 abstract 和 override 这两个关键字保证的。在 Scala 中，abstract override 这两个关键字成对出现在方法定义中，表明这个方法并没有获得完全的定义，因为它覆盖（并使用）了其父类当中的抽象成员。一个类如果有非完整定义的成员，它自身必须是抽象类，其子类必须将这些非完整定义的成员重新定义，才能进行实例化。

对 super 的调用可以是级联的，因此要遵从类的全序化（这是 Scala 的混入构成方式与多重继承方式之间最主要的差异）。例如，考虑另一个与 SyncIterator 类似的类，它将其返回的每个元素都打印到标准输出上：

```
trait LoggedIterator[T] extends AbsIterator[T] {
  abstract override def next: T = {
    val x = super.next; System.out.println(x); x
  }
}
```

我们可以将这两种迭代子（sychronized 和 logged）通过 mixin 组合在一起：

```
class Iter2 extends StringIterator(someString)
               with SyncIterator[char]
               with LoggedIterator[char]
```

在这里，Iter2 的全序化是：

```
{ Iter2, LoggedIterator, SyncIterator, StringIterator, AbsIterator, AnyRef, Any }
```

这样一来，Iter2 的 next 方法继承自 LoggedIterator，而该方法中的 super.next 则指向 SyncIterator 的 next 方法，而后者当中的 super.next 则最终引用 StringIterator 的 next 方法。如果想对记录日志的动作进行同步，仅需要把两个 mixin 的顺序反过来即可实现：

```
class Iter2 extends StringIterator(someString)
               with LoggedIterator[char]
               with SyncIterator[char]
```

无论哪种情况，Iter2 的 next 方法当中 super 的调用都遵循其全序当中的父类顺序。

6.1 面向服务的组件模型（Service-Oriented Component Model）

在 Scala 中，类的抽象与构成机制可以被视作面向服务的组件模型的基础。软件组件是一个个有明确定义的，提供特定服务的计算单元。一般而言，一个软件组件并不是完全自含的，也就是说它的实现依赖于其他一系列提供相应服务的组件。

在 Scala 中，软件组件即对应着类和 trait，类和 trait 的具体成员相当于他们获得的服务，抽象成员相当于他们所需要的服务。组件的组合通过混入的方式来实现，从而使得开发者可以通过较小的组件来产生较大的组件。

Scala 的基于混入的构成机制通过名称来识别服务，举例而言，一个抽象方法 $\mathbf{m}$，可以被任何一个定义了方法 $\mathbf{m}$ 的类 C 来实现，只需要将类 C 进行混入即可。Scala 的组件构成机制通过这种方式将需要实现的服务与服务的提供者自动结合起来，再加上“类的具体成员总是覆盖抽象成员”这一规则，Scala 就形成了一套递归的、插件式的组件机制，使得组件服务不需要显式进行组装。

上述机制可以简化大型的、存在大量递归依赖关系的组件之间的集成，并且能够有效保证（性能方面的）可扩展性，因为服务与调用者之间的结合是依靠编译器来推断的。这一机

制与传统的黑盒式组件相比，最大好处是每个组件都是可扩展的实体：通过集成、重载而实现进化，并且可以对已有组件增加新服务或升级现有服务。总体来说，这些特性有效地支持了一种平滑的、渐进式的软件演进过程。

CHAPTER

SEVEN

解构（DECOMPOSITION）

7.1 面向对象的解构模式（Object-Oriented Decomposition）

处理结构化数据是程序设计很重要的一项工作，在面向对象的语言中，结构化数据通常都用一系列不同结构的类来实现，开发者一般通过调用这些类的各种虚拟方法来访问类内部的数据。我们看一个例子：针对仅包含加法的代数项（algebraic term）求值。采用面向对象的模式，我们可以根据一个代数项的结构进行解构来实现：

```
abstract class Term {
  def eval: int
}
class Num(x: int) extends Term {
  def eval: int = x
}
class Plus(left: Term, right: Term) extends Term {
  def eval: int = left.eval + right.eval
}
```

这段代码用 Term 类来表示一个代数项，包含一个 eval 方法，Term 的具体子类可以实现不同类型的代数项，每一个具体的子类必须给出 eval 方法的具体实现。

这种面向对象的解构模式要求人们预先知道一个数据结构的全部访问方式，这样一来，即使是一些内部方法有时也需要暴露出来。为这些类增加新方法往往是无聊且容易出错的，因为几乎所有的类都要被改写或者继承。另一个问题是，各种操作的实现散落在所有这些类中，不利于人们理解和修改。

7.2 模式匹配替代类层次结构（Pattern Matching Over Class Hierarchies）

上面的代码是一个很好的例子，用于展现函数式解构模式优于面向对象解构模式的场景。在函数式语言中，人们一般将数据结构及其上的操作分开来定义，数据结构通常用代数类型来定义，而操作则定义为普通函数，通过模式匹配来实现对数据结构的操作，这是函数式语言的基本解构原则。这种方式可以让人们仅仅实现 eval 函数，而无需暴露那些人工的辅助函数。

Scala 提供了一套很自然的方式来用函数式编程模型来完成上述工作，让开发者能够定义代数式的结构化结构类型，从而能采用基于模式匹配的解构模式。Scala 没有在语言层面增加代数式数据类型，而是通过类的抽象机制层面的强化，来简化结构化数据的构造：一个增加了 case 描述符的类，自动定义了一个工厂方法，这个方法的参数与类的构造方法一致。同时，Scala 引入了一种机制，可以将 case class 的构造方法作为模式匹配表达式的匹配模式。采用 case class 实现的代数项计算方式如下：

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

在此基础上，类似 1 + 2 + 3 这样的表达式，可以无需使用 new 操作，而是直接采用 case class 的构造方法：

```
Plus(Plus(Num(1), Num(2)), Num(3))
```

Scala 的模式匹配表达式提供了将 case class 的构造方法作为模式的手段，采用模式匹配实现的 eval 方法如下：

```
object Interpreter {
  def eval(term: Term): int = term match {
    case Num(x) => x
    case Plus(left, right) => eval(left) + eval(right)
  }
}
```

其中的模式匹配表达式：x match { case pat1 => e1 case pat2 => e2 ...} 将 x 与模式 pat1、pat2 按顺序进行匹配。上述代码中采用了 Constr(x1, …, xn) 形式的模式，Constr 表示一个 case class 的构造函数，xi 表示一个变量。一个对象如果是某个拥有对应的构造函数的 case class 实例，则与对应的模式相匹配。同时，匹配过程针对相应

的变量进行实例化，而后执行匹配结果对应的右侧的表达式。

这种函数式解构模式的优势就在于可以很容易地在系统中增加新的函数。另一方面，引入一个新的 case class 也可能会引起所有模式匹配表达式的修改。此外，模式匹配还可以进行嵌套，例如可以定义：case Plus(x, y) if x == y => ...，这样就能仅匹配 x == y 的情况，也就是说只有形如 t + t 的表达式才执行对应的计算。嵌套的模式匹配又叫“守护”模式（patterns with guards），顾名思义，x、y 相等通过 x == y 这个模式来保证。

虽然解构比较精确地对应着 *deconstruction* 这个词，这里还是把 *decomposition* 也翻译成解构，这种译法在计算机领域实际上也不少。——译注

CHAPTER

EIGHT

XML 处理（XML PROCESSING）

XML 是一种流行的数据结构，Scala 针对处理 XML 的程序进行了设计，使之更易于创建和维护。Scala 当中包含一些特定的 traits 和类用于构造 XML 的数据模型，从而可以采用模式匹配的解构方式来处理 XML 类型的数据。

8.1 数据模型

在 Scala 中，XML 的数据模型代表一个有序无权树（ordered unranked tree）的不可变类型，树的每一个节点有一个 label、若干子节点以及一个属性名到属性值的 map。这些结构在 `scala.xml.Node` 这个 trait 中定义，它同时还包含了 XPath 的操作符 child 和 descendant-or-self 的定义，分别用 \ 和 \\ 来表示，元素（elements）、文本节点（text nodes）、注释（comments）、处理指令（processing instructions）和实体引用（entity references）则都是它的实体子类。

Scala 的赋值语句中可以直接使用 XML 的语法：

```
val labPhoneBook =
  <phonebook>
    <descr>Phone numbers of<b>XML</b> hackers.</descr>
    <entry>
      <name>Burak</name>
      <phone where="work"> +41 21 693 68 67 </phone>
      <phone where="mobile"> +41 78 601 54 36 </phone>
    </entry>
  </phonebook>
```

上述这段代码中，`labPhoneBook` 是一个 XML 树，它的某个节点拥有一个标签为 `phone`，

它的子节点又包含了一个标签为 +41 21 ... 的文本节点，以及一个以 where 和 “work” 为键值对的 map。在 XML 的语法中，还可以通过 “{” 和 “}” 来转义成 Scala（与 XQuery 的约定类似），例如：一个拥有文本子节点的 date 节点，就可以用 <date>{ df.format(new java.util.Date()) }</date> 来表示当前日期。

8.2 模式校验（Schema Validation）

XML 文档的类型一般通过 schema 来定义，主要的形式化 schema 类型包括 DTD、XML Scahma（也就是 XSD）和 RELAX NG。目前 Scala 只提供了对 DTD 的支持，通过 dtd2scala 这一工具，通过将 DTD 转化为一系列类，这些类只能在符合该 DTD 的 XML 模型中实例化。这样就可以采用 XML 文档的一个特殊的 load 方法，通过模式匹配的方式实例化相关类来针对 DTD 进行校验。未来 Scala 还将支持针对 XML Schema 的校验，包含针对普通类型的静态语法检查。

8.3 序列匹配（Sequence Matching）

Scala 支持通过模式匹配来分解 XML 节点数据，并且匹配过程扔可以使用 XML 语法，虽然只能用于匹配元素。下例展现了如何给 phonebook 这个元素增加一个子项：

```
import scala.xml.Node
def add(phonebook: Node, newEntry: Node): Node =
  phonebook match {
    case <phonebook>{ cs @ _* }</phonebook> =>
      <phonebook>{ cs }{ newEntry }</phonebook>
  }

val newPhoneBook =
  add(scala.xml.XML.loadFile("savedPhoneBook"),
    <entry>
      <name>Sebastian</name>
      <phone where="work">+41 21 693 68 67</phone>
    </entry>)
```

上述 add 函数通过对 phonebook 元素进行匹配，将其子序列匹配到 cs 这个变量（模式 “_*” 表示匹配任何序列），并构造一个新的 phonebook 元素，其中 newEntry 被加在 cs 的后面。包含了 _* 的序列模式，是对第 7 节中描述的代数型模式（algebraic patterns）的

一种扩充，提供了匹配从 0 到任意数量元素的序列的能力。序列模式可以用来匹配任意序列（任何 Seq[A] 类型的序列），并用于可接受序列型输入的 case class，例如：

```
def findRest(z: Seq[Char]): Seq[Char] = z match {
  case Seq('G', 'o', 'o', 'g', 'l', 'e', rest@_*) => rest
}
```

上面这个匹配模式用于检测以“Google”开头的字符串，如果输入字符串匹配，则返回剩余的部分，否则产生一个运行时错误。Scala 的上个版本还（直接）支持通用的正则表达式，不过上面介绍的这种（相对于正则表达式而言）特殊的模式已经基本能满足现实世界的绝大部分需要了，并且避免了全面支持正则表达式而需要的自顶向下的对正则树模式的检测和翻译，因为他们与 Scala 的模式语法非常不匹配。

8.4 用 For Comprehension 实现 XML 查询（XML Queries through For Comprehension）

一般的模式匹配方法，每次只能匹配一个模式，对于 XML 查询而言，我们一般希望一次把所有匹配结果都找到。Scala 灵活的 comprehension 机制，能够提供简洁而优雅的 XML 查询方式，非常接近 XQuery。如下例所示：这段代码可以将 labAddressbook 和 labPhoneBook 的 entry 元素分别取到变量 a 和 p 中，并且对这两个 entry 的 name 元素进行比较，一旦相等，则生成一个名为 result 的 XML 元素，其中包括前者的所有子节点和后者的 phone 元素，也就是地址和电话号码。

```
for (val a <-labAddressBook \\ "entry";
     val p <-labPhoneBook \\ "entry";
     a \ "name" == p \ "name") yield
  <result>{ a.child }{ p \ "phone" }</result>
```

CHAPTER

NINE

组件适配（COMPONENT ADAPTATION）

即便是抽象和构成能力很强大的组件体系，在整合不同团队开发出来的子系统的时候，也会遇到一个问题：那就是一个团队开发出来的组件，其对外接口并不一定满足想要用到他的团队的需要。举例而言，假设有个类库，包含第 5 节中的 GenList 类，可能有个使用者希望这种 List 有集合的特点，提供判断成员的包含或者属于关系的操作。但这个类的设计者没考虑到这方面需求，从而在 GenList 的接口中就没有设计此类操作。

可能有人说，通过继承就可以满足这种需求了，但实际上这只能涵盖使用者自己创建实例的情况。举例而言，假定这个类库本身就提供一个方法，类似于：

```
def fromArray(xs: Array[T]): GenList[T]
```

继承方法是没法将 fromArray 返回的 GenList 改变成 SetList 类型的。当然你也可以用工厂类 [21]（也就是著名的《设计模式》——译注）来部分解决这一困难，但这样一来，你的整个类库结构就会变得更复杂，学习和使用成本更高，而且也解决不了 GenList 其他子类的扩展问题。

这种状况一般称作外部可扩展性问题（external extensibility problem），有文章 [28] 甚至认为正是这一问题阻碍了导致软件组件的开发无法成为成熟的工业体系，因为组件的独立开发和部署是成熟工业体系所必需的能力。

Scala 引进了一个新的概念来解决外部可扩展性问题：视图（views），允许人们通过对现有类增加新的成员和 traits 来进行扩展。

Scala 的视图实际上是隐式参数（implicit parameters）的一种特例。隐式参数是组织复杂功能的一种有效工具，他可以帮助人们写出类似于 Haskell 的 type classes [12] 或者 C++ 意义上的 “concepts”，参见 Siek 和 Lumsdaine [43]。与 type class 不同，隐式参数的作用域是可控的，因此互相冲突（competing）的隐式参数可以在同一程序的不同部分并存。

动机（**Motivation**）。作为例子，我们先定义一个半群（semi-group）的抽象类，其中包

括一个未实现的 add 方法：

```
abstract class SemiGroup[a] {
  def add(x: a, y: a): a
}
```

然后定义他的一个 Monoid 子类，包含一个 unit 成员：

```
abstract class Monoid[a] extends SemiGroup[a] {
  def unit: a
}
```

最后定义两个具体实现类：

```
object Monoids {
  object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  object intMonoid extends Monoid[int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

这样，我们就可以定义一个 sum 方法，他对所有的 monoids 都生效，普通的 Scala 写法可以是这样：

```
def sum[a](xs: List[a])(m: Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))
```

调用这个方法的时候如下所示：

```
sum(List("a", "bc", "def"))(Monoids.stringMonoid)
sum(List(1, 2, 3))(Monoids.intMonoid)
```

这样的写法肯定没问题，但是还不够好，因为每次调用的时候，都必须把具体实现类传递进去。我们肯定希望系统能自动判断合适的参数类型，就像前面所看到的类型参数自动推断一样，这就是隐式参数的作用。

9.1 隐式参数：基础（Implicit Parameters: The Basics）

下面这段代码对 sum 略微改写了一下，引入了一个隐式参数：m

```
def sum[a](xs: List[a])(implicit m: Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

从这里可以看出，一个方法可以既有普通参数又有隐式参数，不过需要说明的是，一个方法或构造函数，只能有一个隐式参数，而且必须放在声明的最后。

implicit 关键字还可以用来修饰定义或声明，例如：

```
implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}
implicit object intMonoid extends Monoid[int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

隐式参数的核心思想在于方法调用时对应的参数可以不用传递，当对应隐式参数的变量不存在时，Scala 编译器会对其进行自动推断。

一个类型标识符能够传递给类型 T 的隐式参数，需要符合隐式参数的类型定义，同时还要满足如下两个条件之一：

1. 该标识符必须在方法调用点可直接访问，无需任何前缀，包括：本地定义、在包含当前位置的作用域中定义、从父类继承而来或在由 import 关键字导入的其他对象中定义。
2. 该标识符定义在一个对象 C 中，其同名类（名为 C 的类）是类型 T 的父类，这种对象叫做 T 的“伴生对象”（companion object）。

上述标准在保证了隐式参数具有一定的局部性，例如：调用者可以通过选择 import 不同的对象来调整传递给隐式参数的标识符定义的范围。

如果同时存在多个与隐式参数类型相匹配的参数值，Scala 编译器会按照标准的静态重载的解析方式找到一个合适的，例如：假定

```
sum(List(1, 2, 3))
```

这条语句的语境中，stringMonoid 和 intMonoid 都是可见的，由于 sum 的类型参数 a 必须是 Int，而适合 Monoid[Int] 类型的隐式参数定义的只有 intMonoid，所以这个对象会被传递给隐式参数。

上述讨论也说明了隐式参数的推断是在所有类型参数推断之后才能完成的。

隐式方法定义自身也可以带有隐式参数，下面这个方法是 scala.List 中定义的一个方法，将列表类型注入（injects）到一个 scala.Ordered 类中，前提是列表的元素也能转化为对应的 Ordered 类型。

```
implicit def list2ordered[a](x: List[a])
                            (implicit elem2ordered: a => Ordered[a]): Ordered[List[a]] =
  ...
```

如果再定义一个方法将一个整数注入到 Ordered 类中：

```
implicit def int2ordered(x: int): Ordered[int]
```

我们就可以对一个可排序列表定义一个 sort 方法：

```
def sort(xs: List[a])(implicit a2ord: a => Ordered[a]) = ...
```

这样一来，我们可以对一个整数列表的列表：yss: List[List[int]] 进行排序了：

```
sort(yss)
```

Scala 编译器会自动推断并传递两个嵌套的隐式参数，完成这一调用

```
sort(yss)((xs: List[int]) => list2ordered[int](xs)(int2ordered))
```

将隐式参数传递给隐式参数的能力存在个问题，就是有可能导致无穷递归。比如，人们可能想要定义这样一个方法，能将任意类型注入到 Ordered 类中：

```
def magic[a](x: a)(implicit a2ordered: a => Ordered[a]): Ordered[a] =
  a2ordered(x)
```

这样的功能未免也太理想了，当然不可能实现。实际上，如果一个 arg 类型本身没有注入到 Ordered 类的另一个具体实现，则对其调用 sort 方法会导致无穷展开：

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

为了避免这种无穷展开，我们要求每一个隐式参数的定义必须是“收缩”的（contractive）：一个方法定义是*收缩的*是指它的每个隐式参数都被一个类型“真包含”(properly contained) [35]，这个类型由该方法去掉所有隐式参数之后剩余的部分转换为一个函数类型所得。例如 list2ordered 方法的类型是：

```
(List[a])(implicit a => Ordered[a]): Ordered[List[a]]
```

这个类型是收缩的，因为其隐式参数的类型 a => Ordered[a] 是被去掉隐式参数的方法 List[a] => Ordered[List[a]] 的类型所真包含的。

上面提到的 magic 方法的类型是

```
(a)(implicit a => Ordered[a]): Ordered[a]
```

这个类型就不是收缩的，因为他的隐式参数类型：a => Ordered[a]，和去掉隐式参数之后的方法是一样的。（*参见节末译注*）

9.2 视图（Views）

视图是类型之间的隐式转换，一般用来为已经存在的类型增加新的功能。例如，下面这个表示一般集合的 trait：

```
trait Set[T] {
  def include(x: T): Set[T]
  def contains(x: T): boolean
}
```

一个从 GenList 到 Set 的视图是通过下面的方法定义的：

```
implicit def listToSet[T](xs: GenList[T]): Set[T] =
  new Set[T] {
    def include(x: T): Set[T] =
      xs prepend x
    def contains(x: T): boolean =
      !isEmpty && (xs.head == x || (xs.tail contains x))
  }
```

如果 xs 是 GenList[T]，那么 listToSet(xs) 将返回一个 Set[T]。

视图与普通方法的唯一区别就是 implicit 关键字，这使视图可以称为隐式参数的候选值，

并且作为隐式转换被自动插入。

如果 e 是一个类型为 T 的表达式，下述两种情况下，e 将被自动进行隐式转换：目标类型不是 T 或者 T 的父类，或者 e 的一个被选中成员不是 T 的成员。例如：一个 GenList[T] 类型的 xs 出现在如下的语句中：

```
val s: Set[T] = xs;
xs contains x
```

编译器会自动将上述定义的 view 插入到这两个语句：

```
val s: Set[T] = listToSet(xs);
listToSet(xs) contains x
```

那么，如果决定使用哪个视图呢？Scala 语言采用与隐式参数的传递同样的规则：一个视图必须可在当前语句被无需前缀的访问，或者在转换的源类型/目标类型之一的伴生对象中进行过定义。视图如果可以映射到一个表达式中合适的类型，或者其中任何类型包含的可选成员，则可以被应用于这个表达式。在所有的可应用视图中，Scala 会挑选一个最明确的，明确性的判定与 Java 和 Scala 中对于重载的解析规则是一样的。如果找不到可应用的视图或者所有可选视图中不存在最明确的，将会产生一个错误。

视图在 Scala 的类库中使用频率很高，主要用于将 Java 的类型进行升级，以支持 Scala 的 traits。例如：Scala 中的 Ordered 这个 trait 定义了一整套用于比较的操作，Scala.Predef 中定义了将所有 Java 基本类型以及 String 转换到这个 trait 的方法。由于任何 Scala 程序都隐含地 import 这个模块的所有成员，因此这些视图是始终可见的。从用户的角度看，基本上可以认为上述 Java 类都通过这个 trait 实现了扩展。

9.3 视图界定（View Bounds）

如前所示，我们看到视图方法在被插入时上必须是静态可见的。实际上，视图如果能够抽象地定义，其作用应该会更大，而通过将 view 也变成一个隐式参数就可以实现这一点了。如下例所示的 maximum 方法，返回任意非空列表的最大元素：

```
def maximum[T](xs: List[T])(implicit t2ordered: T => Ordered[T]): unit = {
  var mx = xs.head
  for (val x <xs.tail) if (mx < x) mx = x
  mx
}
```

这个 maximum 函数可以作用于任何 List[T]，前提是 T 可以通过视图转换为 Ordered[T]，作为特例，这个方法可以作用于任何基本类型的列表，因为标准的 Ordered 视图已经定义好了。

注意到 maximum 方法对类型 T 的两个值 mx 和 x 使用一个比较操作符 (mx < x)，虽然类型 T 本身并没有定义这个操作符 <，由于隐式参数 t2ordered 将 T 映射到一个拥有此操作符的类型，因此这个比较操作被重写为 (t2ordered(mx) < x)。

在 Scala 中，这种将一个泛型参数绑定到隐式视图的场景非常多，以至于 Scala 专门为此设计了相应语法：一个由视图界定（view bounded）的类型参数，形如 [T <% U]，是指参数 T 必须有一个对应的视图，将其映射到类型 U。使用视图界定的方式，maximum 函数的写法可以更加简化：

```
def maximum[T <% Ordered[T]](xs: List[T]): unit = ...
```

这个写法实际上将会被精确地展开成为前面的代码。

本文没有给出收缩的定义，也因此没有上述类型为什么是/不是收缩的解释，有兴趣可以参考 [35]，也就是《The Scala Language Specification》，里面有一段内容和上面非常接近，但增加了 *properly contained* 的解释。

有意思的是，至少在 2.9 及以后版本的 The Scala Language Specification 中，关于 contractive、*properly contained* 等概念直接被去掉了，用一套 Scala 编译器的行为描述所替代，幸好我手里还有旧版本（2.6）才能印证原文的引用。这说明 Scala 在文档完善方面的力度还是很大的，而且方向是脱离过于理论化的色彩，注重对计算机层面构造的阐述。为此，也建议大家如果要了解 Scala 语言最新发展，最少还是从 2.9 文档开始。（我印象里 Scala 语言/规范层面的大规模重构是从 2.8 开始的）

——译注

CHAPTER

TEN

相关研究（RELATED WORK）

Scala 的设计受到多种语言及相关论文的影响，下面列出的是对 Scala 的设计产生主要影响的一些相关研究。

显而易见地，Scala 从 Java [23] 和 C# [15] 中吸收了大量概念和语法约定。Scala 表达属性的方式，大致借鉴了 Sather 语言 [44] 的模型。Scala 的统一对象模型，主要来自于 SmallTalk [22] 的概念。从 Beta [30] 语言中，Scala 借鉴了一切皆可嵌套的理念，包括类。Scala 的混入模型设计，主要来自于面向对象的线性混入 [6]，但是将混入式构成定义成为对称模式，更接近于 [14, 25, 49] 中定义的混入模块，或者 traits [42]。Scala 的抽象类型定义非常接近 ML [24] 和 OCaml [29] 当中模块体系的抽象类型签名，但将其进一步泛化成一等语法组件。For-comprehensions 是基于 Haskell 的 monad comprehensions [46]，尽管语法上更接近 XQuery [3]。视图主要是借鉴 Haskell 的 type classes [47]，可以看做是面向对象版本的类型化 type classes [38]，不过却更具通用性，因为其实例的声明是有范围的/局部的。在动态类型系统里，视图的主要好处来自于 Classbox 的概念，不过 Classbox 比视图更强大，允许本地重绑定，从而可以通过动态指派来选择类的特定扩展。

某种意义上讲，Scala 是 Pizza [37] 相关工作的一个延续（*Pizza 也是 Odersky 发明的语言，接近于是 Scala 前身——译注*）。和 Pizza 一样，Scala 编译后在 JVM 上运行，增加了高阶函数，泛型和模式匹配等函数式编程的构造。区别之处在于 Pizza 还向后兼容 Java，而 Scala 的目标则是只保留互操作性，这样在设计方面就有更大的自由度。

Scala 的目标是为组件的抽象和构成提供先进的语法构造，这一点在很多最近的研究中也都有体现。抽象类型就相当于 gbeta [16, 17] 中虚拟类（virtual classes）的一个保守实现，提供了其带来的大部分好处。类似的还有 FamilyJ [40] 中的代理层，以及 [32] 中提出的 Java 的嵌套继承等。Jiazzi [31] 是 Java 的一个扩展，提出了基于单元的模块化机制，这是一个强大的参数化模块形式的实现。Jazzi 支持类似于 Scala 的语法扩展方式，例如实现 mixin 的能力。

Nice 语言 [4] 也是最近出现的一个类似于 Scala 的语言，但是从 $ML_{<=}$[5] 继承了一些内容。

Nice 包含了多分派（multiple dispatch）、开放类（open classes）以及一个基于抽象接口实现的受限形态的可追溯抽象模式。Nice 不支持模块化的实现阶段类型检查。Nice 和 Scala 虽然都与 Java 有很大差异，但都被设计为与 Java 程序和类库具有互操作性，并且也都编译成 JVM 上执行。

MultiJava [13] 是一个 Java 的保守扩展，也增加了对称多路指派和开放类。他采用了很多不同的解决方案来实现 Scala 解决的问题。例如：多路指派就解决了二元操作方法的问题，而 Scala 是通过抽象类型解决的。又如开放类则解决了外部可扩展性问题，而 Scala 使通过视图来解决的。MultiJava 可以动态为一个类增加新方法，这是因为开放类是基于 Java 的动态装载机制实现的，这一点是 Scala 所没有的。反过来，也只有 Scala 才支持对类的外部扩展限定可见范围。

OCaml 和 Moby [20] 是另外两个将函数式和面向对象模式进行整合的静态类型语言。与 Scala 不同之处在于，这两种语言都是基于一套丰富的函数式编程语言和复杂的模块化系统，然后建立起相对轻量级的对象体系。

CHAPTER

ELEVEN

结论（CONCLUSION）

Scala 应该算是既大又小的一门语言。

大，是指他拥有丰富的语法和类型体系，将面向对象和函数式编程模型结合在一起。因此，从某一种单一的语言类型过来的用户需要学习相应的新构造。Scala 的多样性也主要是基于希望更接近 Java 和 C# 等传统语言的考虑，从而让这些语言的用户更容易采用。

小，则是指 Scala 建立在一套适度的，非常通用的概念基础之上，大部分源代码层面的结构都是语法糖，可以被编码所以换掉。Scala 一体化的对象模型，可以让用户将大量基础类型及相关操作替换成 Scala 的类库中对应的结构，从而实现进一步的抽象。

Scala 提供了一整套强大的语法结构用于组件的构成、抽象和适配，目标是基于这套组件模型，语言的扩展能力足够强，使能够让用户能够很自然地通过类库和框架来实现所在领域的建模。因此，对语言本身的扩展要求就少了，因为大部分结构都能很容易地通过类库来实现。Scala 自身的类库和各种应用就是丰富的例子：Erlang 模式的 actor 类、任意精度的整型、Horn 子句和约束等。这些语法构造都能够像特定语言中实现的那样自然，同时又能与 Scala 无缝整合（通过扩展，进一步与 Java 实现整合）。

Scala 这种实现扩展性的方式，从某种意义上讲把做出良好设计的责任从语言的发明者转移到了使用者身上——设计出糟糕的类库和设计出糟糕的语言效果差不多了。当然，我们还是相信 Scala 的语法构造，相对于目前的主流语言，能够帮助使用者更容易设计出良好的类库。

声明 Scala 的设计和实现，部分来自于瑞士国家基金 NFS 21-61825 项目的支持，瑞士国家竞争力中心 MICS 研究项目，欧洲 Framework 6 PalCom 项目，微软研究院，Hasler 基金会。同时，Gilad Bracha, Craig Chambers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Mejer, Oscar Nierstrasz, Klaus Ostermann, Didier Rémy, Mads Torgersen, 及 Philip Wadler 等人通过活跃的、有启发性的讨论，为语言的成形做出了贡献。Scala 邮件列表的参与者，也为语言和工具的改进提供了非常有益的反馈。

[1] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In Proc. OOPSLA, pages 69-82. ACM Press, Oct 1996.

[2] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In Proc. JMLC 2003, volume 2789 of Springer LNCS, pages 122-131, 2003.

[3] S. Boag, D. Chamberlin, M. F. Fermandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. W3c recommendation, World Wide Web Consortium, November 2003. http://www.w3.org/TR/xquery/.

[4] D. Bonniot and B. Keller. The Nice's user's manual, 2003. http://nice.sourceforge.net/NiceManual.pdf

[5] F. Bourdoncle and S. Merz. Type-checking Higher-Order Polymorphic Multi-Methods. In Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 15-17, Paris, France, 1997.

[6] G. Bracha and W. Cook. Mixin-Based Inheritance. In N. Meyrowitz, editor, Proceedings of ECOOP '90, pages 303-311, Ottawa, Canada, October 1990. ACM Press.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, eds. Extensible Markup Language (XML) 1.0. W3C recommendation, World Wide Web Consortium, February 2004. Available online http://www.w3.org/TR/RECxml20040204/.

[8] K. B. Bruce, M. Odersky, and P. Wadler. A Statically Safe Alternative to Virtual Types. Lecture Notes in Computer Science, 1445, 1998. Proc. ESOP 1998.

[9] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In Pro- ceedings of ECOOP '95, LNCS 952, pages 27-51, Aarhus, Denmark, August 1995. Springer-Verlag.

[10] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-Bounded Quantification for Object-Oriented Programming. In Proc. of 4th Int. Conf. on Functional Programming and Computer Architecture, FPCA'89, London, pages 273- 280, New York, Sep 1989. ACM Pres.

[11] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An Extension of System F with

Subtyping. Information and Computation, 109(1-2):4-56, 1994.

[12] K. Chen, P. Hudak, and M. Odersky. Parametric Type Classes. In Proc. ACM Conf. on Lisp and Functional Pro- gramming, pages 170-181, June 1992.

[13] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Design Rationale, Compiler Implementation, and User Experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, Jan 2004.

[14] D. Duggan. Mixin modules. In Proceedings of the ACM SIG- PLAN International Conference on Functional Program- ming (ICFP '96), volume 31(6), pages 262-273, 1996.

[15] ECMA. C# Language Specification. Technical Report Standard ECMA-334, 2nd Edition, European Computer Manufacturers Association, December 2002.

[16] E. Ernst. Family polymorphism. In Proceedings of the Eu- ropean Conference on Object-Oriented Programming, pages 303-326, Budapest, Hungary, 2001.

[17] E. Ernst. Higher-Order Hierarchies. In L. Cardelli, editor, Proceedings ECOOP 2003, LNCS 2743, pages 303-329, Heidelberg, Germany, July 2003. Springer-Verlag.

[18] M. O. et.al. An introduction to Scala. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online http://scala.epfl.ch.

[19] D. C. Fallside, editor. XML Schema. W3C recommendation, World Wide Web Consortium, May 2001. Available online http://www.w3.org/TR/xmlschema0/.

[20] K. Fisher and J. H. Reppy. The Design of a Class Mechanism for Moby. In SIGPLAN Conference on Programming Language Design and Implementation, pages 37-49, 1999.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.

[22] A. Goldberg and D. Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Lan- guage Specification. Java Series, Sun Microsystems, second edition, 2000.

[24] R. Harper and M. Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In Proc. 21st ACM Symposium on Principles of Programming Languages, January 1994.

[25] T. Hirschowitz and X. Leroy. Mixin Modules in a Call-by- Value Setting. In European

Symposium on Programming, pages 6-20, 2002.

[26] A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. In Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP2002), pages 441-469, June 2002.

[27] M. P. Jones. Using parameterized signatures to express modular structure. In Proceedings of the 23rd ACM Sympo- sium on Principles of Programming Languages, pages 68- 78. ACM Press, 1996.

[28] R. Keller and U. Hölzle. Binary Component Adaptation. In Proceedings ECOOP, Springer LNCS 1445, pages 307-329, 1998.

[29] X. Leroy. Manifest Types, Modules and Separate Compilation. In Proc. 21st ACM Symposium on Principles of Pro- gramming Languages, pages 109-122, January 1994.

[30] O. L. Madsen and B. Moeller-Pedersen. Virtual Classes - A Powerful Mechanism for Object-Oriented Programming. In Proc. OOPSLA'89, pages 397-406, October 1989.

[31] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age Components for Old-Fashioned Java. In Proc. of OOPSLA, October 2001.

[32] N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In Proc. OOPSLA, Oct 2004.

[33] Oasis. RELAX NG. See http://www.oasisopen. org/.

[34] M. Odersky. Scala by example. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online http://scala.epfl.ch.

[35] M. Odersky. The Scala Language Specification. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online http://scala.epfl.ch.

[36] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In Proc. ECOOP'03, Springer LNCS 2743, jul 2003.

[37] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In Proc. 24th ACM Symposium on Principles of Programming Languages, pages 146-159, January 1997.

[38] M. Odersky, P. Wadler, and M. Wehr. A Second Look at Overloading. In Proc. ACM Conf. on Functional Program- ming and Computer Architecture, pages 135-146, June 1995.

[39] M. Odersky, C. Zenger, and M. Zenger. Colored Local Type Inference. In Proceedings of

the 28th ACM Symposium on Principles of Programming Languages, pages 41-53, London, UK, January 2001.

[40] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In Proceedings of the 16th Euro- pean Conference on Object-Oriented Programming, Malaga, Spain, 2002.

[41] B. C. Pierce and D. N. Turner. Local Type Inference. In Proc. 25th ACM Symposium on Principles of Programming Languages, pages 252-265, New York, NY, 1998.

[42] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In Proceedings of the 17th European Conference on Object-Oriented Programming, Darmstadt, Germany, June 2003.

[43] J. Siek and A. Lumsdaine. Essential Language Support for generic programming. In PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language de- sign and implementation, pages 73-84, Jun 2005.

[44] D. Stoutamire and S. M. Omohundro. The Sather 1.0 Specification. Technical Report TR-95-057, International Computer Science Institute, Berkeley, 1995.

[45] M. Torgersen, C. P. Hansen, E. Ernst, P. vod der Ahé, G. Bracha, and N. Gafter. Adding Wildcards to the Java Programming Language. In Proceedings SAC 2004, Nicosia, Cyprus, March 2004.

[46] P. Wadler. The Essence of Functional Programming. In Proc.19th ACM Symposium on Principles of Programming Languages, pages 1-14, January 1992.

[47] P. Wadler and S. Blott. How to make ad-hoc Polymorphism less ad-hoc. In Proc. 16th ACM Symposium on Principles of Programming Languages, pages 60-76, January 1989.

[48] M. Zenger. Type-Safe Prototype-Based Component Evolution. In Proceedings of the European Conference on Object- Oriented Programming, Málaga, Spain, June 2002.

[49] M. Zenger. Programming Language Abstractions for Ex- tensible Software Components. PhD thesis, Department of Computer Science, EPFL, Lausanne, March 2004.