A Scala Tutorial
for Java programmers

Version 1.3
March 15, 2009

Michel Schinz, Philipp Haller
靳雄飞（中文翻译）

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 介绍 － Introduction

本文档是 Scala 语言和编译器的快速入门介绍，适合已经有一定编程经验，且希望了解 Scala 可以做什么的读者。我们假定本文的读者具有面向对象编程（Object-oriented programming，尤其是 java 相关）的基础知识。

This document gives a quick introduction to the Scala language and compiler. It is intended for people who already have some programming experience and want an overview of what they can do with Scala. A basic knowledge of object-oriented programming, especially in Java, is assumed.

2 第一个例子 － A first example

我们使用最经典的“Hello world”作为第一个例子，这个例子虽然并不是特别炫（fascinating），但它可以很好的展示 Scala 的用法，且无须涉及太多的语言特性。示例代码如下：

As a first example, we will use the standard Hello world program. It is not very fascinatingbut makes it easy to demonstrate the use of the Scala tools without knowing too much about the language. Here is how it looks:

```
object HelloWorld { def
  main(args: Array[String])
  { println("Hello,
  world!") } }
```

Java 程序员应该对示例代码的结构感到很熟悉：它包含一个 `main` 方法，其参数是一个字符串数组，用来接收命令行参数；`main` 的方法体只有一句话，调用预定义的 `println` 方法输出“Hello world！”问候语。`main` 方法不返回值（这是一个过程方法 procedure method），因此，该方法不必声明返回值类型。

The structure of this program should be familiar to Java programmers: it consists of one method called `main` which takes the command line arguments, an array of strings, as parameter; the body of this method consists of a single call to the predefined method `println` with the friendly greeting as argument. The `main` method does not return a value (it is a procedure method). Therefore, it is not necessary to declare a return type.

对于包含 `main` 方法的 **`object`** 声明，Java 程序员可能要相对陌生一些。这种声明方式引入了一个通常被称为单例对象（singleton object）的概念，也就是有且仅有一个实例的类。因此，上例中的声明，在定义了一个名为的 `HelloWorld` 类的同时，还声明了该类的一个实例，实例的名字也叫 `HelloWorld`。该实例在第一次被用到的时候即时（on demand）创建。

What is less familiar to Java programmers is the **`object`** declaration containing the `main` method. Such a declaration introduces what is commonly known as a *singleton object*, that is a class with a single instance. The declaration above thus declares both

a class called `HelloWorld` and an instance of that class, also called `HelloWorld`. This instance is created on demand, the first time it is used.

细心(astute，机敏的，聪明的)的读者可能会注意到，`main` 方法并没有声明为 `static`。这是因为 Scala 中不存在静态成员（无论方法还是属性，methods or fields）这一概念，Scala 使用前述的单例对象（singleton objects）中的成员来代替静态成员。

The astute reader might have noticed that the `main` method is not declared as `static` here. This is because static members (methods or fields) do not exist in Scala. Rather than defining static members, the Scala programmer declares these members in singleton objects.

2.1 编译该示例 — Compiling the example

要编译上面写的例子，要 `scalac` 命令，这就是 Scala 的编译器。`scalac` 的工作流程和多数编译器类似：从命令行上接收待编译的源文件名（source file）以及编译参数，生成一个或者多个目标文件（object files，或者叫对象文件）。Scala 生成的目标文件是标准的 java class 文件。

To compile the example, we use `scalac`, the Scala compiler. `scalac` works like most compilers: it takes a source file as argument, maybe some options, and produces one or several object files. The object files it produces are standard Java class files.

假如我们将 `HelloWorld` 示例程序存放到 `HelloWorld.scala` 文件中，则可以用以下指令进行编译（大于号'>'表示 shell/命令行 的提示符，不需要人工键入）：
If we save the above program in a file called `HelloWorld.scala`, we can compile it by issuing the following command (the greater-than sign '>' represents the shell prompt and should not be typed):

```
> scalac HelloWorld.scala
```

该指令执行后，会在当前目录下生成几个 class 文件，其中一个是 HelloWorld.class，该文件中包含一个可以直接被 `scala` 指令执行的类(class)，具体操作参见后续章节。

This will generate a few class files in the current directory. One of them will be called `HelloWorld.class`, and contains a class which can be directly executed using the `scala` command, as the following section shows.

2.2 运行该示例 — Running the example

代码编译通过以后，可以使用 `scala` 指令运行程序，`scala` 指令和 `java` 指令的用法非常相似，甚至它们接受的命令行参数都是一样的。前面编译好的例子，可以用如下指令运行，并输出预期的问候语：

Once compiled, a Scala program can be run using the `scala` command. Its usage is very similar to the `java` command used to run Java programs, and accepts the same options. The above example can be executed using the following command, which produces the expected output:

```
> scala -classpath . HelloWorld
```

```
Hello, world!
```

3 和 Java 进行交互 — Interaction with Java

和 Java 代码的交互能力，是 Scala 语言的强项之一。在 Scala 程序中，`java.lang` 包下的类是默认全部引入的，其它包下的类则需要显式(explicitly)引入。

One of Scala's strengths is that it makes it very easy to interact with Java code. All classes from the `java.lang` package are imported by default, while others need to be imported explicitly.

我们可以通过一个例子来展示Scala与Java的交互能力。假如，我们想获取系统当前时间，并按照某个国家（比如法国）的显示习惯进行格式化[1]。

Let's look at an example that demonstrates this. We want to obtain and format the current date according to the conventions used in a specific country, say France[1] .

我们知道，在 Java 的类库中已经实现了 Date、DateFormat 等功能强大的工具类，且 Scala 可以和 Java 进行无缝(seemlessly)的互操作，所以，想在 Scala 程序中使用这些功能，只需要引入这些 Java 类即可，无须从头重复实现相同的功能。

Java's class libraries define powerful utility classes, such as `Date` and `DateFormat`. Since Scala interoperates seemlessly with Java, there is no need to implement equivalent classes in the Scala class library–we can simply import the classes of the corresponding Java packages:

```
import java.util.{Date, Locale}

import java.text.DateFormat

import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Scala 的 import 语句和 Java 中的 import 很想象，但 Scala 的语法更强大一些。比如，要想引入一个包中的多个类，在 Scala 中可以写在一行上，只需要把多个类名放到一个大括号中(curly braces, {})即可。此外，如果要引入一个包或者类中的所有名字，Scala 使用下划线(underscore，_)而不是星号（asterisk，*），这是因为，在 Scala 中，星号是一个合法的标识符（比如：方法名），后面我们会遇到这种情况。

[1] 其它说法语的国家和地区，比如瑞士的部分地区通常也使用相同的格式。

Scala's import statement looks very similar to Java's equivalent, however, it is more powerful. Multiple classes can be imported from the same package by enclosing them in curly braces as on the first line. Another difference is that when importing all the names of a package or class, one uses the underscore character (_) instead of the asterisk (*). That's because the asterisk is a valid Scala identifier (e.g. method name), as we will see later.

因此，上例中第三行的 import 语句，引入了 `DateFormat` 类的所有成员，这样该类的静态方法 `getDateInstance` 和静态属性 `LONG` 对 `FrenchDate` 类来说，是直接可见的（directly visible）。

The import statement on the third line therefore imports all members of the `DateFormat` class. This makes the static method `getDateInstance` and the static field `LONG` directly visible.

在 `main` 方法中，我们首先创建一个 Java 的 `Date` 实例，该实例默认取得系统当前时间；接下来，我们使用从 `DateFormat` 类中引入的静态方法 `getDateInstance` 创建一个负责日期格式化的对象 df，创建过程中，通过参数指定了本地化区域（`Locale.FRANCE`）；最后，使用 df 将当前时间进行格式化并打印输出到控制台。这个方法的最后一行，体现了 Scala 语法中一种很有意思的特性（interesting property）：如果一个方法只接受一个参数，那么可以使用 infix 语法，也就是说，下面的表达式：

```
df format now
```

和 `df.format(now)`的语义完全相同，只是前者更加简洁。

Inside the `main` method we first create an instance of Java's `Date` class which by default contains the current date. Next, we define a date format using the static `getDateInstance` method that we imported previously. Finally, we print the current date formatted according to the localized `DateFormat` instance. This last line shows an interesting property of Scala's syntax. Methods taking one argument can be used with an infix syntax. That is, the expression

```
df format now
```

is just another, slightly less verbose way of writing the expression

```
df.format(now)
```

这虽然是一个很小的语法细节，但它具有深远的影响，本文后续的章节中将会有进一步的论述。

This might seem like a minor syntactic detail, but it has important consequences, one of which will be explored in the next section.

最后，从 Java 与 Scala 整合的角度看，值得一提的是，Scala 中可以直接继承 Java 的类或者实现 Java 的接口。

To conclude this section about integration with Java, it should be noted that it is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

4 一切皆对象 — Everything is an object

Scala 中的一切都是对象，从这个意义上说，Scala 是纯粹的面向对象（pure object-oriented）的语言。在这一点上，Scala 与 Java 不同，因为 Java 中，原子类型（primitive types）和引用类型是有区别的，而且 Java 中不能把函数（function）当做值（value）来对待。

Scala is a pure object-oriented language in the sense that *everything* is an object, including numbers or functions. It differs from Java in that respect, since Java distinguishes primitive types (such as `boolean` and `int`) from reference types, and does not enable one to manipulate functions as values.

4.1 数字是对象 — Numbers are objects

因为数字是对象，所以数字也拥有自己的方法，如下的算术表达式：

```
1+2*3/x
```

实际上完全是由方法调用（method calls）构成的。前面章节已经提到过“单参数方法”的简化写法，所以，上述表达式实际上是下面这个表达式的等价简化写法：

```
(1).+(((2).*(3))./(x))
```

Since numbers are objects, they also have methods. And in fact, an arithmetic expression like the following:

```
1+2 3/x*
```

consists exclusively of method calls, because it is equivalent to the following expression, as we saw in the previous section:

```
(1).+(((2).*(3))./(x))
```

由此我们还可以看到：+, *等符号在 Scala 中是合法的标识符（和前面进行印证）。

This also means that +, *, etc. are valid identifiers in Scala.

在第二种写法中，加在数字上的括号是必须的，因为 Scala 的词法分析器使用贪婪算法（longest match，最长匹配）来匹配符号，所以，表达式：

```
1.+(2)
```

将被解释成：1.、+和 2 三个符号。虽然“1.”和“1”都是合法的符号，但“1.”的长度更长，所以被 Scala 的词法分析器优先选择，而“1.”等价于字面值(literal)“1.0”，

这将产生一个 Double 浮点数而不是我们期望的 Int 整数。所以必须在数字上面加括号，就像这样：

```
(1).+(2)
```

以避免整数 1 被解释成 Double 类型。

The parentheses around the numbers in the second version are necessary because Scala's lexer uses a longest match rule for tokens. Therefore, it would break the following expression:

```
1.+(2)
```

into the tokens `1.`, `+`, and `2`. The reason that this tokenization is chosen is because `1.` is a longer valid match than `1`. The token `1.` is interpreted as the literal `1.0`, making it a `Double` rather than an `Int`. Writing the expression as:

```
(1).+(2)
```

prevents `1` from being interpreted as a `Double`.

4.2 函数是对象 － Functions are objects

在 Scala 中，函数(functions)也是对象(objects)，所以，函数可以当做参数进行传递，可以把函数存储在变量中，也可以把函数作为其他函数的返回值，Java 程序员可能会觉得这是一项非常神奇的特性。这种将函数当做值进行操作的能力，是函数式编程（functional programming）最重要的特性（cornerstone，基石）之一。

Perhaps more surprising for the Java programmer, functions are also objects in Scala. It is therefore possible to pass functions as arguments, to store them in variables, and to return them from other functions. This ability to manipulate functions as values is one of the cornerstone of a very interesting programming paradigm called *functional programming*.

举一个简单的例子，就可以说明把函数当做值来操作的意义何在。假如我们要开发一个定时器，该定时器每秒钟执行一定的动作，我们如何把要执行的动作传给定时器？最直观的回答是：传一个实现动作的函数（function）。许多程序员，对这种函数传递模式并不陌生：在用户界面（user-interface）相关代码中，当事件被触发时，会调用预先注册的回调函数。

As a very simple example of why it can be useful to use functions as values, let's consider a timer function whose aim is to perform some action every second. How do we pass it the action to perform? Quite logically, as a function. This very simple kind of function passing should be familiar to many programmers: it is often used in user-interface code, to register call-back functions which get called when some event occurs.

下面的程序将实现简单定时器的功能，负责定时的函数（function）名为：`oncePerSecond`，它接受一个回调函数作为参数，该回调函数的类型记为：`() => Unit`，代表任何无参数、无返回值的函数（`Unit` 和 C/C++中的 `void` 类似）。程序的 main 方法调用定时函数，

作为实参传进去的回调函数 timeFlies，仅仅向终端打印一句话，所以，该程序的实际功能是：每秒钟在屏幕上打印一条信息：time flies like an arrow。

In the following program, the timer function is called `oncePerSecond`, and it gets a call-back function as argument. The type of this function is written `() => Unit` and is the type of all functions which take no arguments and return nothing (the type `Unit` is similar to `void` in C/C++). The main function of this program simply calls this timer function with a call-back which prints a sentence on the terminal. In other words, this program endlessly prints the sentence “time flies like an arrow” every second.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

注意，程序中使用 Scala 预定义方法 `println` 实现字符串显示，而没有用 `System.out` 上的方法。

Note that in order to print the string, we used the predefined method `println` instead of using the one from `System.out`.

4.2.1 匿名函数 — Anonymous functions

定时器的示例程序还可以做一些改进。首先，`timeFlies` 函数只被用过一次，也就是当做回调函数传给 `oncePerSecond` 的时候，对于这种函数，在用到的时候即时构造更合理，因为可以省去定义和命名的麻烦，在 Scala 中，这样的函数称为匿名函数（anonymous functions），也就是没有名字的函数。使用匿名函数代替 `timeFlies` 函数后的程序代码如下：

While this program is easy to understand, it can be refined a bit. First of all, notice that the function `timeFlies` is only defined in order to be passed later to the `oncePerSecond` function. Having to name that function, which is only used once, might seem unnecessary, and it would in fact be nice to be able to construct this function just as it is passed to `oncePerSecond`. This is possible in Scala using *anonymous functions*, which are exactly that: functions without a name. The revised version of our timer program using an anonymous function instead of `timeFlies` looks like that:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
```

```
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

代码中的右箭头'=>'表明程序中存在一个匿名函数，箭头左边是匿名函数的参数列表，右边是函数体。在本例中，参数列表为空(箭头左边是一对空括号)，而函数体和改造前定义的 timeFlies 函数保持一致。

The presence of an anonymous function in this example is revealed by the right arrow '=>' which separates the function's argument list from its body. In this example, the argument list is empty, as witnessed by the empty pair of parenthesis on the left of the arrow. The body of the function is the same as the one of timeFlies above.

5 类 — Classes

前面已经说过，Scala是面向对象的语言，所以它有类（class）的概念[2]。Scala中声明类的语法和Java类似，但有一点重要的差异，那就是Scala中的类定义可以带参数（parameters），下面定义的复数类（complex number）可以很清晰的展示这一特性：

As we have seen above, Scala is an object-oriented language, and as such it has a concept of class.[2] Classes in Scala are declared using a syntax which is close to Java's syntax. One important difference is that classes in Scala can have parameters. This is illustrated in the following definition of complex numbers.

```
class Complex(real: Double, imaginary: Double) {

  def re() = real
  def im() = imaginary
}
```

该复数类可以接受两个参数，分别代表复数的实部和虚部，如果要创建 `Complex` 类的实例，则必须提供这两个参数，比如：`new Complex(1.5, 2.3)`。该类有两个方法：`re` 和 `im`，分别用于访问复数的实部和虚部。

This complex class takes two arguments, which are the real and imaginary part of the complex. These arguments must be passed when creating an instance of class `Complex`, as follows: `new Complex(1.5, 2.3)`. The class contains two methods, called `re` and `im`, which give access to these two parts.

需要注意的是，这两个方法的返回值都没有显式定义。在编译过程中，编译器可以根据函数定义的右部（right-hand ），推断(infer，deduce)出两个函数的返回值都是 `Double` 类型。

[2] 严格的说，的确有些面向对象的语言没有类（class）的概念，但Scala并非其中之一。

It should be noted that the return type of these two methods is not given explicitly. It will be inferred automatically by the compiler, which looks at the right-hand side of these methods and deduces that both return a value of type `Double`.

但编译器并非在任何情况下都能准确推导出数据类型，而且，很难用一套简单的规则来定义什么情况下可以，什么情况下不可以。不过，对于没有显式指定类型，且无法推导出类型的表达式，编译器会给出提示信息，所以，推导规则的复杂性，对实际编程的影响不大。对于初学者，可以遵循这样一条原则：当上下文看起来比较容易推导出数据类型时，就应该忽略类型声明，并尝试是否能够编译通过，不行就修改。这样用上一段时间，程序员会积累足够的经验，从而可以比较自如的决定何时应该省略类型声明，而何时应该显式声明类型。

The compiler is not always able to infer types like it does here, and there is unfortunately no simple rule to know exactly when it will be, and when not. In practice, this is usually not a problem since the compiler complains when it is not able to infer a type which was not given explicitly. As a simple rule, beginner Scala programmers should try to omit type declarations which seem to be easy to deduce from the context, and see if the compiler agrees. After some time, the programmer should get a good feeling about when to omit types, and when to specify them explicitly.

5.1 无参方法－ Methods without arguments

`Complex` 类中的 `re` 和 `im` 方法有个小问题，那就是调用这两个方法时，需要在方法名后面跟上一对空括号，就像下面的例子一样：

A small problem of the methods `re` and `im` is that, in order to call them, one has to put an empty pair of parenthesis after their name, as the following example shows:

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

如果能够省掉这些方法后面的空括号，就像访问类属性（fields）一样访问类的方法，则程序会更加简洁。这在 Scala 中是可行的，只需将方法显式定义为没有参数（without arguments）即可。无参方法和零参方法（methods with zero arguments）的差异在于：无参方法在声明和调用时，均无须在方法名后面加括号。所以，前面的 `Complex` 类可以重写如下：

It would be nicer to be able to access the real and imaginary parts like if they were fields, without putting the empty pair of parenthesis. This is perfectly doable in Scala, simply by defining them as methods *without arguments*. Such methods differ from methods with zero arguments in that they don't have parenthesis after their name, neither in their definition nor in their use. Our `Complex` class can be rewritten as follows:

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

5.2 继承和方法重写 — Inheritance and overriding

Scala 中的所有类都继承自某一个父类（或者说超类，super-class），若没有显式指定父类（比如前面的 `Complex` 类），则默认继承自 `scala.AnyRef`。

All classes in Scala inherit from a super-class. When no super-class is specified, as in the `Complex` example of previous section, `scala.AnyRef` is implicitly used.

在 Scala 中可以重写（overriding）从父类继承的方法，但必须使用 **`override`** 修饰符来显式声明，这样可以避免无意间的方法覆盖（accidental overriding）。例如，前面定义的 `Complex` 类中，我们可以重写从 `Object` 类中继承的 `toString` 方法，代码如下：

It is possible to override methods inherited from a super-class in Scala. It is however mandatory to explicitly specify that a method overrides another one using the **`override`** modifier, in order to avoid accidental overriding. As an example, our `Complex` class can be augmented with a redefinition of the `toString` method inherited from `Object`.

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

6 条件类和模式匹配 — Case classes and pattern matching

树是软件开发中使用频率很高的一种数据结构，例如：解释器和编译器内部使用树来表示代码结构；XML 文档是树形结构；还有一些容器（集合，containers）也是基于树的，比如：红黑树（red-black tree，一种自平衡二叉查找树）。

A kind of data structure that often appears in programs is the tree. For example, interpreters and compilers usually represent programs internally as trees; XML documents are trees; and several kinds of containers are based on trees, like red-black trees.

接下来，我们通过一个示例程序，了解在 Scala 中如何表示和操作树形结构，这个示例将实现非常简单的计算器功能，该计算器可以处理包含加法、变量和整数常量的算术表达式，比如：1 + 2、(*x* + *x*) + (7 + *y*)等。

We will now examine how such trees are represented and manipulated in Scala through a small calculator program. The aim of this program is to manipulate very simple arithmetic expressions composed of sums, integer constants and variables. Two examples of such expressions are 1 + 2 and (*x* + *x*) + (7 + *y*).

首先，我们要决定如何表示这样的表达式。最自然的选择是树形结构，用非叶子节点表示操作符（具体到这个例子，只有 加法操作），用叶子节点表示操作数（具体到这个例子是常量和变量）。

We first have to decide on a representation for such expressions. The most natural one is the tree, where nodes are operations (here, the addition) and leaves are values (here constants or variables).

如果是在 Java 中，建立树形结构最常见的做法是：创建一个表示树的抽象类，然后每种类型的节点用一个继承自抽象类的子类来表示。而在函数式编程语言中，则可以使用代数数据类型（algebraic data-type）来达到同样的目的。Scala 则提供了一种介于两者之间（类继承和代数数据类型），被称为条件类（case classes）的概念，下面就是用条件类定义树的示例代码：

In Java, such a tree would be represented using an abstract super-class for the trees, and one concrete sub-class per node or leaf. In a functional programming language, one would use an algebraic data-type for the same purpose. Scala provides the concept of *case classes* which is somewhat in between the two. Here is how they can be used to define the type of the trees for our example:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

上例中的 Sum, Var 和 Const 就是条件类，它们与普通类的差异主要体现在如下几个方面：

The fact that classes `Sum`, `Var` and `Const` are declared as case classes means that they differ from standard classes in several respects:

- 新建条件类的实例，无须使用 **new** 关键字（比如，可以直接用 `Const(5)` 代替 **new** `Const(5)`来创建实例）。
- the **new** keyword is not mandatory to create instances of these classes (i.e. one can write `Const(5)` instead of **new** `Const(5)`),
- 自动为构造函数所带的参数创建对应的 getter 方法（也就是说，如果 c 是 Const 的实例，通过 c.v 即可访问构造函数中的同名参数 v 的值）
- getter functions are automatically defined for the constructor parameters (i.e. it is possible to get the value of the `v` constructor parameter of some instance `c` of class `Const` just by writing `c.v`),
- 条件类都默认实现 equals 和 hashCode 两个方法，不过这两个方法都是基于实例的结构本身（structure of instance），而不是基于实例中可用于区分的值（identity），这一点和 java 中 Object 提供的同名方法的默认实现是基本一致的。
- default definitions for methods `equals` and `hashCode` are provided, which work on the *structure* of the instances and not on their identity,
- 条件类还提供了一个默认的 toString 方法，能够以源码形式（source form）打印实例的值（比如，表达式 *x*+1 会被打印成

`Sum(Var(x),Const(1))`，这个打印结果，和源代码中创建表达式结构树的那段代码完全一致)。

- a default definition for method `toString` is provided, and prints the value in a "source form" (e.g. the tree for expression *x*+1 prints as `Sum(Var(x),Const(1))`),
- 条件类的实例可以通过模式匹配（pattern matching）进行分解（decompose），接下来会详细介绍。
- instances of these classes can be decomposed through *pattern matching* as we will see below.

既然我们已经定义了用于表示算术表达式的数据结构，接下来我们可以定义作用在这些数据结构上的操作。首先，我们定义一个在特定环境（environment，上下文）中对表达式进行求值的函数，其中环境的作用是为了确定表达式中的变量的取值。例如：有一个环境，对变量 x 的赋值为 5，我们记为：{*x* → 5}，那么，在这个环境上求 *x* + 1 的值，得到的结果为 6。

Now that we have defined the data-type to represent our arithmetic expressions, we can start defining operations to manipulate them. We will start with a function to evaluate an expression in some *environment*. The aim of the environment is to give values to variables. For example, the expression *x* + 1 evaluated in an environment which associates the value 5 to variable *x*, written {*x* → 5}, gives 6 as result.

在程序中，环境也需要一种合理的方式来表示。可以使用哈希表（hash table）之类的数据结构，也可以直接使用函数（functions）！实际上，环境就是一个给变量赋予特定值的函数。上面提到的环境：{*x* → 5}，在 Scala 中可以写成：

We therefore have to find a way to represent environments. We could of course use some associative data-structure like a hash table, but we can also directly use functions! An environment is really nothing more than a function which associates a value to a (variable) name. The environment {*x* → 5} given above can simply be written as follows in Scala:

```
{ case "x" => 5 }
```

上面这一行代码定义了一个函数，如果给该函数传入一个字符串`"x"`作为参数，则函数返回整数 5，否则，将抛出异常。

This notation defines a function which, when given the string `"x"` as argument, returns the integer 5, and fails with an exception otherwise.

在写表达式求值函数之前，我们还要对环境的类型（type of the environments）进行命名。虽然在程序中全都使用 `String => Int` 这种写法也可以的，但给环境起名后，可以简化代码，并使得将来的修改更加方便（这里说的环境命名，简单的理解就是宏，或者说是自定义类型）。在 Scala 中，使用如下代码来完成命名：

Before writing the evaluation function, let us give a name to the type of the environments. We could of course always use the type `String => Int` for environments, but it simplifies the program if we introduce a name for this type, and

makes future changes easier. This is accomplished in Scala with the following notation:

```
type Environment = String => Int
```

此后，类型名 Environment 可以作为“从 String 转成 Int”这一类函数的别名。

From then on, the type Environment can be used as an alias of the type of functions from String to Int.

现在，我们来写求值函数。求值函数的实现思路很直观：两个表达式之和（sum），等于分别对两个表达式求值然后求和；变量的值直接从环境中获取；常量的值等于常量本身。在 Scala 中描述这个概念并不困难：

We can now give the definition of the evaluation function. Conceptually, it is very simple: the value of a sum of two expressions is simply the sum of the value of these expressions; the value of a variable is obtained directly from the environment; and the value of a constant is the constant itself. Expressing this in Scala is not more difficult:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n) => env(n)
  case Const(v) => v
}
```

求值函数的工作原理是对树 t 上的结点进行模式匹配，下面是对匹配过程的详细描述（实际上是递归）：

This evaluation function works by performing *pattern matching* on the tree t. Intuitively, the meaning of the above definition should be clear:

1. 求值函数首先检查树 t 是不是一个求和（Sum），如果是，则把 t 的左子树和右子树分别绑定到两个新的变量 l 和 r 上，然后对箭头右边的表达式进行运算（实际上就是分别求左右子树的值然后相加，这是一个递归）。箭头右边的表达式可以使用箭头左边绑定的变量，也就是 l 和 r。it first checks if the tree t is a Sum, and if it is, it binds the left sub-tree to a new variable called l and the right sub-tree to a variable called r, and then proceeds with the evaluation of the expression following the arrow; this expression can (and does) make use of the variables bound by the pattern appearing on the left of the arrow, i.e. l and r,
2. 如果第一个检查不满足，也就是说，树 t 不是 Sum，接下来就要检查 t 是不是一个变量 Var；如果是，则 Var 中包含的名字被绑定到变量 n 上，然后继续执行箭头右边的逻辑。if the first check does not succeed, that is if the tree is not a Sum, it goes on and checks if t is a Var; if it is, it binds the name contained in the Var node to a variable n and proceeds with the right-hand expression,

3. 如果第二个检查也不满足，那意味着树 `t` 既不是 `Sum`，也不是 `Var`，那就进一步检查 `t` 是不是常量 `Const`。如果是，则将常量所包含的值赋给变量 `v`，然后继续执行箭头右边的逻辑。if the second check also fails, that is if `t` is neither a `Sum` nor a `Var`, it checks if it is a `Const`, and if it is, it binds the value contained in the `Const` node to a variable `v` and proceeds with the right-hand side,

4. 最后，如果以上所有的检查都不满足，程序会抛出异常，表明对表达式做模式匹配时产生了错误。这种情况，在本例中，只有声明了更多 `Tree` 的子类，却没有增加对应的模式匹配条件时，才会出现。finally, if all checks fail, an exception is raised to signal the failure of the pattern matching expression; this could happen here only if more sub-classes of `Tree` were declared.

通过上例，我们可以看到，模式匹配的过程，实际上就是把一个值（value）和一系列的模式进行比对，如果能够匹配上，则从值（value）中取出有用的部件（parts）进行命名，然后用这些命名的部件（作为参数）来驱动另一段代码的执行。

We see that the basic idea of pattern matching is to attempt to match a value to a series of patterns, and as soon as a pattern matches, extract and name various parts of the value, to finally evaluate some code which typically makes use of these named parts.

一个有经验（seasoned，老练的）的面向对象程序员可能会问：为什么不把 `eval` 定义成类 `Tree` 的成员方法？事实上，这么做也行，因为在 Scala 中，条件类和普通类一样，都可以定义方法。不过，“模式匹配”和“类方法”除了编程风格的差异，也各有利弊，决策者需要根据程序的扩展性需求做出权衡和选择：

A seasoned object-oriented programmer might wonder why we did not define `eval` as a *method* of class `Tree` and its subclasses. We could have done it actually, since Scala allows method definitions in case classes just like in normal classes. Deciding whether to use pattern matching or methods is therefore a matter of taste, but it also has important implications on extensibility:

- 使用类方法，添加一种新的节点类型比较简单，因为只需要增加一个 `Tree` 的子类即可。但是，要在树上增加一种新的操作则比较麻烦，因为这需要修改 `Tree` 的所有子类。
- when using methods, it is easy to add a new kind of node as this can be done just by defining the sub-class of `Tree` for it; on the other hand, adding a new operation to manipulate the tree is tedious, as it requires modifications to all sub-classes of `Tree`,
- 使用模式匹配，情况则刚好相反：增加一种新的节点类型需要修改所有作用在树上的模式匹配函数；而增加新的操作则比较简单，只需要增加一个新的函数即可。
- when using pattern matching, the situation is reversed: adding a new kind of node requires the modification of all functions which do pattern

matching on the tree, to take the new node into account; on the other hand, adding a new operation is easy, by just defining it as an independent function.

为了更深入的探索模式匹配，我们要在算术表达式上定义一个新的操作：对符号求导（symbolic derivation，导数）。该操作的规则如下：

To explore pattern matching further, let us define another operation on arithmetic expressions: symbolic derivation. The reader might remember the following rules regarding this operation:

1. 对和求导，等于分别求导的和。the derivative of a sum is the sum of the derivatives,
2. 对变量 *v* 求导，有两种情况：如果变量 *v* 正好是用于求导的符号，则返回 1，否则返回 0。the derivative of some variable *v* is one if *v* is the variable relative to which the derivation takes place, and zero otherwise,
3. 常量求导恒为 0。the derivative of a constant is zero.

这几条规则几乎可以直接翻译成 Scala 的代码：

These rules can be translated almost literally into Scala code, to obtain the following definition:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

通过求导函数的定义，又引出了两个跟模式匹配相关的知识点。第一、**`case`** 语句可以带一个 *guard*，它由关键字 **`if`** 和紧随其后的表达式组成。*guard* 的作用是对 **`case`** 匹配的模式进行二次限定，只有 **`if`** 后面的表达式为 true 时，才允许匹配成功。在本例中，*guard* 保证当且仅当被求导的变量名 `n` 等于当前求导符号 `v` 时，才返回常量 1。第二、模式匹配中可以使用通配符（记为_，下划线）来匹配任意值（相当于 java 中 switch 语句的 default 分支）。

This function introduces two new concepts related to pattern matching. First of all, the **`case`** expression for variables has a *guard*, an expression following the **`if`** keyword. This guard prevents pattern matching from succeeding unless its expression is true. Here it is used to make sure that we return the constant 1 only if the name of the variable being derived is the same as the derivation variable `v`. The second new feature of pattern matching used here is the *wild-card*, written _, which is a pattern matching any value, without giving it a name.

模式匹配的功能非常强大，但限于本文的长度和定位，我们将不再做太多深入的讨论，接下来，我们还是通过一个实例，来看看前面定义的两个函数如何使用吧。为此，我们编写一个的 `main` 函数，在函数中，先创建一个表达式：(*x* + *x*) + (7 + *y*)，然后在环境 {*x* → 5, *y* → 7}上求表达式的值，最后分别求表达式相对于 *x* 和 *y* 的导数。

We did not explore the whole power of pattern matching yet, but we will stop here in order to keep this document short. We still want to see how the two functions above perform on a real example. For that purpose, let's write a simple `main` function which performs several operations on the expression $(x + x) + (7 + y)$: it first computes its value in the environment $\{x \rightarrow 5, y \rightarrow 7\}$, then computes its derivative relative to *x* and then *y*.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

执行这段程序，我们得到的输出如下：

Executing this program, we get the expected output:

```
Expression:     Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

仔细观察程序的输出，我们发现，如果把求导结果化简（simplification）后再展示给用户会更好。使用模式匹配来定义一个化简函数是很有意思（同时也很棘手）的事情，读者可以自己做做练习。

By examining the output, we see that the result of the derivative should be simplified before being presented to the user. Defining a basic simplification function using pattern matching is an interesting (but surprisingly tricky) problem, left as an exercise for the reader.

7 Traits（特征、特性）

Scala 中的类不但可以从父类继承代码（code），还可以从一个或者多个 traits 引入代码。

Apart from inheriting code from a super-class, a Scala class can also import code from one or several *traits*.

对于 Java 程序员来说，理解 traits 最简单的方法，是把它当作可以包含代码逻辑的接口（interface）。在 Scala 中，如果一个类继承自某个 trait，则该类实现了 trait 的接口，并继承了 trait 的所有代码（code）。

Maybe the easiest way for a Java programmer to understand what traitss are is to view them as interfaces which can also contain code. In Scala, when a class inherits from a trait, it implements that traits's interface, and inherits all the code contained in the trait.

我们用一个经典的例子：有序对象（ordered objects）来展示 trait 的作用。有很多应用场景，需要在同类对象之间比较大小，比如排序算法。在 Java 中，可以实现 `Comparable` 接口，而在 Scala 中，有更好的办法，那就是定义一个和 `Comparable` 对等的 trait，名为：`Ord`。

To see the usefulness of traits, let's look at a classical example: ordered objects. It is often useful to be able to compare objects of a given class among themselves, for example to sort them. In Java, objects which are comparable implement the `Comparable` interface. In Scala, we can do a bit better than in Java by defining our equivalent of `Comparable` as a trait, which we will call `Ord`.

对象之间做比较，需要六种断言（predicate，谓词）：小于，小于等于，等于，不等于，大于等于，大于。不过，这六种断言中的四种，可以用另外两种进行表述，比如，只要确定了等于和小于两种断言，其它四种就可以推导出来，所以，并不是每种断言都需要由具体类来实现（实现在 trait 上即可，相当于抽象类）。基于以上的分析，我们用下面的代码定义一个 trait：

When comparing objects, six different predicates can be useful: smaller, smaller or equal, equal, not equal, greater or equal, and greater. However, defining all of them is fastidious, especially since four out of these six can be expressed using the remaining two. That is, given the equal and smaller predicates (for example), one can express the other ones. In Scala, all these observations can be nicely captured by the following trait declaration:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

以上代码，定义了和 java 中 `Comparable` 接口对等的 trait：`Ord`，同时，默认实现了三个断言，这三个断言依赖的第四个是抽象的（留给具体类实现）。等于和不等于默认存在于所有对象上，因此这里不需要显式定义。

This definition both creates a new type called `Ord`, which plays the same role as Java's `Comparable` interface, and default implementations of three predicates in terms of a fourth, abstract one. The predicates for equality and inequality do not appear here since they are by default present in all objects.

代码中用到的类型 `Any` 是 Scala 中所有类型的超类。它比 java 中的 `Object` 类型更加通用，因为基本类型如：`Int`, `Float` 也是继承自该类的。

The type `Any` which is used above is the type which is a super-type of all other types in Scala. It can be seen as a more general version of Java's `Object` type, since it is also a super-type of basic types like `Int`, `Float`, etc.

要想让一个类的实例可比，只需要输入（mix in，实际上就是继承）前面定义的 `Ord` trait（原文是 class，个人感觉可能是笔误），并实现相等和小于（inferiority，劣等，劣势）两个断言即可。接下来还是用例子说话，我们定义一个 `Date` 类，这个类使用三个整数分别表示公历的年、月、日，该类继承自 `Ord`，代码如下：

To make objects of a class comparable, it is therefore sufficient to define the predicates which test equality and inferiority, and mix in the `Ord` class above. As an example, let's define a `Date` class representing dates in the Gregorian calendar. Such dates are composed of a day, a month and a year, which we will all represent as integers. We therefore start the definition of the `Date` class as follows:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

请重点关注代码中紧跟在类名和参数后面的 **`extends`** `Ord`，这是 `Date` 类声明继承自 `Ord` trait 的语法。

The important part here is the **`extends`** `Ord` declaration which follows the class name and parameters. It declares that the `Date` class inherits from the `Ord` trait.

接下来，我们要重写（redefine）从 `Object` 上继承的 `equals` 方法，该方法的默认实现是比较对象的天然特性（比如内存地址），而 `Date` 类需要比较年、月、日字段的值才能确定大小。

Then, we redefine the `equals` method, inherited from `Object`, so that it correctly compares dates by comparing their individual fields. The default implementation of `equals` is not usable, because as in Java it compares objects physically. We arrive at the following definition:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

以上代码用到了两个预定义的方法：`isInstanceOf` 和 `asInstanceOf`。其中 `isInstanceOf` 方法对应 java 中的 `instanceof` 操作符，当且仅当一个对象的类型和方法参数所指定类型匹配时，才返回 true；`asInstanceOf` 方法对应 java 中的 cast 强制类型转换操作：如果当前对象是特定类的实例，转换成功，否则抛出 `ClassCastException` 异常。

This method makes use of the predefined methods `isInstanceOf` and `asInstanceOf`. The first one, `isInstanceOf`, corresponds to Java's `instanceof` operator, and returns true if and only if the object on which it is applied is an instance of the given type. The second one, `asInstanceOf`, corresponds to Java's cast operator: if the object is an instance of the given type, it is viewed as such, otherwise a `ClassCastException` is thrown.

最后，还需要定义一个判断小于（inferiority）的函数，该函数又用到了一个预定义方法：`error`，作用是抛出异常，并附带指定的错误信息。代码如下：

Finally, the last method to define is the predicate which tests for inferiority, as follows. It makes use of another predefined method, `error`, which throws an exception with the given error message.

```
    def <(that: Any): Boolean = { if
      (!that.isInstanceOf[Date])
      error("cannot compare " + that + "
      and a Date")

    val o = that.asInstanceOf[Date]
    (year < o.year) ||
    (year == o.year && (month < o.month ||
                   (month == o.month && day < o.day)))
  }
```

至此，`Date` 类就写完了，该类的实例既可以被看作是一个日期（dates），也可以被看作一个可比的对象，并且，无论那种看法，他们都有六个用作比较的断言，其中，`equals` 和<直接定义在 `Date` 类上，而其它四个则继承自 `Ord` trait。

This completes the definition of the `Date` class. Instances of this class can be seen either as dates or as comparable objects. Moreover, they all define the six comparison predicates mentioned above: `equals` and < because they appear directly in the definition of the `Date` class, and the others because they are inherited from the `Ord` trait.

Traits 的作用当然远不止这些，但深入的探讨超出了本文的范围。

Traits are useful in other situations than the one shown here, of course, but discussing their applications in length is outside the scope of this document.

8 泛型 — Genericity

最后，我们再来看看 Scala 中的泛型。Java 在 1.5 才引入泛型，在此之前，Java 程序员对语言缺少泛型支持所引发的种种问题，应该是深有体会的。

The last characteristic of Scala we will explore in this tutorial is genericity. Java programmers should be well aware of the problems posed by the lack of genericity in their language, a shortcoming which is addressed in Java 1.5.

所谓泛型，就是代码中可以使用参数化类型的能力。例如，有一个负责类库开发的程序员，他想提供一个链表（linked list）结构，他必须决定，在链表中可以存放什么类型的元素。由于链表可以被广泛使用，所以预先限定链表中元素的类型是不现实的，即便勉强作出武断的决定，势必会给类库的应用带来极大的局限性。

Genericity is the ability to write code parametrized by types. For example, a programmer writing a library for linked lists faces the problem of deciding which type to give to the elements of the list. Since this list is meant to be used in many different contexts, it is not possible to decide that the type of the elements has to be, say, `Int`. This would be completely arbitrary and overly restrictive.

在这种情况下，Java 程序员选择 `Object` 类来降低局限性，但这种解决方案很不理想，一方面，java 中的基本类型，比如 `int`, `long`, `float` 等等不是对象，也就无法加入链表，另一方面，使用 Object 这样的最顶级类，意味着程序员要在代码中大量的使用动态类型转换。

Java programmers resort to using `Object`, which is the super-type of all objects. This solution is however far from being ideal, since it doesn't work for basic types (`int`, `long`, `float`, etc.) and it implies that a lot of dynamic type casts have to be inserted by the programmer.

Scala 引入泛型类（和泛型方法）来解决此这个问题。让我们以引用（reference）为例来了解泛型，引用是最简单的容器，可以指向某种类型的对象，或者为空（什么都不指向）。

Scala makes it possible to define generic classes (and methods) to solve this problem. Let us examine this with an example of the simplest container class possible: a reference, which can either be empty or point to an object of some type.

```
class Reference[T] { private var contents: T = _

 def set(value: T) { contents = value }
 def get: T = contents }
```

以上代码中，`Reference` 是参数化类，`T` 是类型参数。类型参数在类中用于定义变量 `contents` 的类型、用作 `set` 方法的参数以及 `get` 方法的返回值。

The class `Reference` is parametrized by a type, called `T`, which is the type of its element. This type is used in the body of the class as the type of the `contents` variable, the argument of the `set` method, and the return type of the `get` method.

例子中声明了一个变量，变量本身没有什么可讨论的，但赋给变量的初始值是‘_’，这一点比较有意思。_表示各种类型的默认值，其中，数字类型的默认值是 `0`，`Boolean` 型的默认值是 **`false`**，`Unit` 类型是`()`，而所有的对象类型（`object type`）的默认值为 **`null`**。

The above code sample introduces variables in Scala, which should not require further explanations. It is however interesting to see that the initial value given to that variable is _, which represents a default value. This default value is 0 for numeric types, **`false`** for the `Boolean` type, `()` for the `Unit` type and **`null`** for all object types.

要使用 `Reference` 类，需要指定类型参数 `T` 的具体类型，也就是被引用的对象的类型。例如，下面的代码创建一个指向整数的引用：

To use this `Reference` class, one needs to specify which type to use for the type parameter `T`, that is the type of the element contained by the cell. For example, to create and use a cell holding an integer, one could write the following:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get*2))
   }
 }
```

从代码中我们可以看到，`get` 方法返回的值不需要做类型转换，就可以当作整数使用，并且，该引用无法指向整数之外的任何对象。

As can be seen in that example, it is not necessary to cast the value returned by the `get` method before using it as an integer. It is also not possible to store anything but an integer in that particular cell, since it was declared as holding an integer.

9 结语 — Conclusion

本文档提供 Scala 语言的简单介绍和一些基本的示例。对 Scala 感兴趣的读者，可以进一步阅读 *Scala By Example*，并在必要的时候参考 *Scala Language Specification*。

This document gave a quick overview of the Scala language and presented some basic examples. The interested reader can go on by reading the companion document *Scala By Example*, which contains much more advanced examples, and consult the *Scala Language Specification* when needed.