Scala入门手册

> 本手册由河北工业大学人工智能与数据科学学院刘洪普翻译。
>
> 本书原始链接：Scala Book
>
> 并根据课程内容进行了部分修改，时间仓促难免有错误，如果您发现错误请发邮件告知我。
>
> 我的邮箱地址为：liuji@hebut.edu.cn
>
> 当前文档版本：v 0.1-2020-04-01

1. 前言：SCALA的味道

本书介绍了 Scala 这款优美、现代、而且富于表达的编程语言，在前言中本书首先介绍 Scala 的一些主要特性，接下来才是一个更为传统的“新手入门”章节。

> 本书假设读者之前使用过类似 Java 这样的编程语言，并且打算通过阅读一系列 Scala 的示例来感觉 Scala 的味道。尽管不是必须的，建议首先阅读如何下载和安装 Scala 的内容，以便之后测试代码。当然，也可以使用在线的 Scala 运行网站 https://scalafiddle.io/ 测试示例代码。

1.1 概述

在开始示例之前，首先了解一下 Scala 的的特点：

- Scala 是一款高级语言（相对于汇编、机器语言等低级语言，或者 C 语言这样的中级语言）；
- Scala 是静态类型的（相对于 Python 这种动态类型语言）；
- Scala 的代码风格即简明又不乏可读性，极具表达力；
- Scala 支持面向对象的编程范式；
- Scala 支持函数式编程范式；
- Scala 具有成熟的推理系统；
- Scala 的代码编译为 `.class` 文件，需要在 Java 虚拟机（JVM）上运行。
- 在 Scala 中可以方便的使用 Java 的库。

1.2 Hello, world

丹尼斯·里奇的《C 程序设计语言》一书开创了所有编程书都得以“Hello World”开始的传统，遵循传统让我们从下面的例子开始，在 IDEA 里面新建一个 `Scala Class`，然后选择 `object`，新建一个名为 `Hello` 的对象：

```
object Hello extends App {
    println("Hello, world")
}
```

然后单击右键，选择运行代码，即可看到运行结果。

> 如果要使用命令行进行编译，也可以将上面的代码保存到 `Hello.scala` 中，然后在命令行运行：
>
> ```
> $ scalac Hello.scala
> ```
>
> `scalac` 类似于 Java 中的 `javac` ，Scala 的编译会生成两个文件：
>
> - Hello$.class
> - Hello.class
>
> 生成的 `.class` 文件与 `javac` 生成的 `.class` 文件一样都是字节码文件，可以用运行在 Java 虚拟机上。要运行 `Hello` 应用程序，可以用 `Scala` 命令：
>
> ```
> $ scala Hello
> ```

在本书的后面会提供更多的“Hello，world”的例子。

1.3 Scala 的交互执行环境

Scala 的交互执行环境（Read-Evaluate-Print-Loop，**REPL**）是一个命令行的解释器，你可以像使用 Python 交互式解释器一样使用它。在控制台中输入 `scala` 就可以启动 REPL，启动后的界面如下所示：

```
$ scala
Welcome to Scala 2.13.1 (Java HotSpot(TM) 64-Bit Server VM, Java 13.0.2).
Type in expressions for evaluation. Or try :help.

scala>
```

由于 REPL 是一个命令行解释器，我们可以直接输入 Scala 表达式看看发生了什么：

```
scala> val x = 1
x: Int = 1

scala> val y = x + 1
y: Int = 2
```

如同上面显示的结果，当你在 REPL 中输入表达式后，将在提示行之后显示命令的执行结果。

1.4 两种不同的变量

Scala 中有两种不同类型的变量：

- `val` 是不可变的变量 - 类似于 Java 中的 `final` ，即该变量只能引用固定的对象，我们在 Scala 代码中优先使用 `val` 变量。
- `var` 用于创建可变的变量，使用 `var` 变量必须要有不得不这么做的理由。

示例：

```
val x = 1 // immutable
var y = 0 // mutable
```

1.5 声明变量的类型

在 Scala 中通常在创建变量时不需要声明变量的类型：

```
val x = 1
val s = "a string"
val p = new Person("Regina")
```

这样创建的变量，其数据类型通常由 Scala 进行推理。

```
scala> val x = 1
x: Int = 1

scala> val s = "a string"
s: String = a string
```

这个特性被称为**类型推断**，有助于我们写出简洁的代码。在 Scala 中也可以**显式**的声明变量的数据类型，但是这并不是必要的：

```
val x: Int = 1
val s: String = "a string"
val p: Person = new Person("Regina")
```

正如你看到的，这样写出来的代码比较臃肿。

1.6 控制结构

接下来简单介绍 Scala 的控制语句。

1.6.1 if/else

Scala 的 `if/else` 控制结构与其他的语言非常相似：

```
if (test1) {
    doA()
} else if (test2) {
    doB()
} else if (test3) {
    doC()
} else {
    doD()
}
```

与 Java 以及其他语言不同，`if/else` 结构返回一个值，因此我们也可以用三元运算符版本的 `if/else`：

```scala
val x = if (a < b) a else b
```

1.6.2 match表达式

Scala 中的 `match` 表达式的功能近似于 Java 中的 `switch` 语句：

```scala
val result = i match {
    case 1 => "One"
    case 2 => "two"
    case _ => "not 1 or 2"
}
```

`match` 表达式不仅可以比较整数，可以支持任意的数据类型，例如布尔类型：

```scala
val booleanAsString = bool match {
    case true => "true"
    case false => "false"
}
```

在下面的例子中，`match` 表达式作为方法的方法体，可以匹配各种不同的类型：

```scala
def getClassAsString(x: Any): String = x match {
    case s: String => s + " is a String"
    case i: Int => "Int"
    case f: Float => "Float"
    case l: List[_] => "List"
    case p: Person => "Person"
    case _ => "Unknown"
}
```

强大的 `match` 表达式时 Scala 的一大特性，在本书的后面将会分享更多的例子。

1.6.3 try/catch

Scala 的 `try/catch` 结构语句可以用于捕获异常，其功能与 Java 相似，但是语法格式更接近 `match` 表达式：

```scala
try {
    writeToFile(text)
} catch {
    case fnfe: FileNotFoundException => println(fnfe)
    case ioe: IOException => println(ioe)
}
```

1.6.4 for循环与for表达式

Scala 中的 `for` 循环通常如下例所示：

```
for (arg <- args) println(arg)

// "x to y" syntax
for (i <- 0 to 5) println(i)

// "x to y by" syntax
for (i < 0 to 10 by 2) println(i)
```

我们可以在 `for` 循环中使用 `yield` 关键词来创建 `for` 表达式。下面的例子可以产生一个包含 `2, 4, 6, 8, 10` 的向量：

```
val x = for (i <- 1 to 5) yield i * 2
```

下面的例子展示了如何使用 `for` 表达式来遍历列表：

```
val fruits = List("apple", "banana", "lime", "orange")

val fruitLengths = for {
    f <- fruits
    if f.length > 4
} yield f.length
```

由于 Scala 代码看上去有点像数学描述，因此即使从来没有见到过 Scala 的 `for` 表达式，也能猜出程序的功能。

1.6.5 while 和 do/while

在 Scala 中同样有 `while` 和 `do/while` 循环控制语句，下面是它们的语法格式：

```
// while loop
while (condition) {
    statement(a)
    statement(b)
}

// do-while
do {
    statement(a)
    statement(b)
}
while(condition)
```

1.7 类

首先来看一个 Scala 中类的例子：

```
class Person(var firstName: String, var lastName: String) {
    def printFullName() = println(s"$firstName $lastName")
}
```

下面的代码展示了如何使用这个类：

```
val p = new Person("Julia", "Kern")
println(p.firstName)
p.lastName = "Manes"
p.printFullName()
```

注意在 Scala 中不需要为类的字段创建 `get` 和 `set` 方法。

下面给出一个更为复杂一点 `Pizza` 类：

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {
    def addTopping(t: Topping): Uint = toppings += t
    def removeTopping(t: Topping): Uint = toppings -= t
    def removeAllToppings(): Uint = toppings.clear()
}
```

在上面的例子中， `ArrayBuffer` 类似于 Java 中的 `ArrayList` 。在这个例子中 `CrustSize` 、`CrustType` 和 `Topping` 类的定义省略掉了，但是你可能仍然能够理解这个类是如何工作的。

1.8 Scala 方法

与其他的面向对象语言类似，Scala 的类可以拥有方法，Scala 中方法的语法如下例所示：

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

在定义方法的时候可以省略返回类型，因此上面的方法可以写成下面这种方法，我们推荐省略方法的返回类型：

```
def sum(a: Int, b: Int) = a + b
def concatenate(s1: String, s2: String) = s1 + s2
```

下面的例子是对这些函数的调用方法：

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

对于方法还有更多的使用细节，例如为参数添加默认值，将在本书后面的例子中详细介绍。

1.9 特征（Traits）

Scala 中的特征可以将你的代码分解成更小的模块单元，下面的例子给出了三个特征：

```
trait Speaker {
    def speak(): String // has no body, so it's abstract
}

trait TailWagger {
    def startTail(): Uint = println("tail is wagging")
    def stopTail(): Uint = println("tail is stopped")
}

trait Runner {
    def startRunning(): Uint = println("I'm running")
    def stopRunning(): Uint = println("Stopped running")
}
```

你可以创建一个 `Dog` 类并继承所有这些特征，并覆写 `speak` 方法：

```
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

类似的，我们也可以创建一个 `Cat` 类，并覆写更多的特征方法：

```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
    override def startRunning(): Uint = println("Yeah ... I don't run")
    override def stopRunning(): Uint = println("No need to stop")
}
```

如果你觉得这样的代码言之有理，那就说明你会喜欢特征机制。如果你没有这种感觉，也不用担心，在本书的后面会给出更多的解释。

1.10 集合类

在 Scala 中可以使用 Java 中的集合类，很多人通过这种方式来熟悉 Scala 。但是在本书中强烈建议尽量学习一下 Scala 中的基本集合类型：`List` 、 `ListBuffer` 、 `Vector` 、 `ArrayBuffer` 、 `Map` 和 `Set` 。因为这些 Scala 集合类拥有很多强大的方法可以极大的简化你的代码。

1.10.1 生成列表

在很多时候，通过填充数据来生成列表的功能是非常有用的，Scala 提供了很多方法来生成列表，下面是一些例子：

```
val nums = List.range(0, 10)
val nums = (1 to 10 by 2).toList
val letters = ('a' to 'f').toList
val letters = ('a' to 'f' by 2).toList
```

1.10.2 序列方法

在 Scala 中有很多序列集合类，例如 `Array` 、 `ArrayBuffer` 、 `Vector` 和 `List` 等。首先看一下使用 `List` 的例子：

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

下面是使用 `foreach` 方法的例子：

```
scala> names.foreach(println)
joel
ed
chris
maurice
```

下面是调用 `filter` 方法之后跟上一个 `foreach` 方法：

```
scala> nums.filter(_ < 4).foreach(println)
1
2
3
```

下面是一个使用 `map` 方法的例子：

```
scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

scala> val capNames = names.map(_.capitalize)
capNames: List[String] = List(Joel, Ed, Chris, Maurice)

scala> val lessThanFive = nums.map(_ < 5)
lessThanFive: List[Boolean] = List(true, true, true, true, false, false,
false, false, false, false)
```

即使不做解释，你应该也能看出 `map` 的工作机制：它将一个算法应用到集合的每个元素上，然后返回结果。

如果你已经做好心理准备，那么不妨看一下更为强大的一个方法——`foldLeft`：

```
scala> nums.foldLeft(0)(_ + _)
res0: Int = 55

scala> nums.foldLeft(1)(_ * _)
res1: Int = 3628800
```

`foldLeft` 的第一个参数是一个种子值，你可能能够猜出第一个例子是在计算 `nums` 中所有数字的和，第二个例子是在计算所有数字的乘积。

更多关于 Scala 中集合类的方法，我们将在后面的章节中进行专门的介绍。

1.11 元组

Scala 中的元组允许将不同类型的元素放到同一个集合中，Scala 中的元组可以容纳 2 个到 22 个值，元组中的元素可以拥有不同的类型。

首先定义一个 `Person` 类：

```
class Person(var name: String)
```

接下来可以创建一个包含三个不同类型元素的元组：

```
val t = (11, "Eleven", new Person("Eleven"))
```

可以用下面的方式访问元组中的值：

```
t._1
t._2
t._3
```

或者直接将元组的字段赋给不同的变量：

```
val (num, string, person) = (11, "Eleven", new Person("Eleven"))
```

1.12 尚未涉及的内容

尽管已经介绍了 Scala 中的不少特性，但是以下内容尚未涉及到：

- 字符串和内建数值类型；
- 包与导入；
- 如何在 Scala 中使用 Java 的集合类；
- 如何在 Scala 中使用 Java 的类库；
- 如何构建 Scala 项目；
- 如何在 Scala 中进行单元测试；
- 如何编写 Scala 的 Shell 脚本；
- 映射、集合及其他集合类；
- 面向对象编程范式
- 函数式编程范式
- 基于 `Futures` 的并发程序设计
- 等等……

这些内容将在本书的剩余部分进行介绍。

1.13 一点背景知识

Scala 语言由 Martin Odersky 发明，曾经师从 `Pascal` 语言的发明人 Niklaus Wirth 。奥德斯基先生是通用 Java 语言的设计者之一，被称为 `javac` 编译器之父。

2. 初识 Scala

在本书中我们假设读者熟悉其他的某种编程语言，例如 Java 语言，在本书中不会把大量时间浪费到编程的基础知识上。也就是说本书是面向具有编程经验，但是不熟悉 Scala 的读者。

2.1 准备工作

Scala 的安装可以按照官网的新手入门（https://docs.scala-lang.org/getting-started/index.html）来完成，该入门手册介绍了如何在 IDE 和 命令行中使用 Scala。

2.1.1 注释

首先了解一下 Scala 中注释的语法规则是个不错的选择， Scala 中的注释与 Java 以及很多其他的语言类似：

```
// 这是单行注释

/*
 * 这是多行注释
 */

/**
 * 这也是多行注释
 */
```

2.1.2 集成开发环境

下面的三种主流集成开发环境都支持 Scala：

- IntelliJ IDEA （https://www.jetbrains.com/idea/download）
- Visual Studio Code （https://code.visualstudio.com/）
- Scala IDE for Eclipse （http://scala-ide.org/）

2.1.3 命名惯例

在 Scala 中命名惯例遵循类似于 Java 中的“骆驼命名法”：

- 类名：`Person` 、 `StoreEmployee`
- 变量名： `name` 、 `firstName`
- 方法名： `convertToInt` 、 `toUpper`

2.2 Scala的特性

Scala 的名字来自于单词 `scalable` ，真是实至名归。Scala 被广泛的用于世界上最繁忙的网站，例如推特、网飞、领英、新浪等网站。

下面介绍一些关于 Scala 的重要信息：

- Scala 语言由马丁·奥德斯基（ `javac` 之父）发明，Scala 语言博采 Java、Ruby、Smalltalk、ML、Haskell、Erlang 等语言之长。
- Scala 是高级语言。
- Scala 是静态类型语言。
- Scala 拥有成熟的类型推断系统。
- Scala 的代码精简而又不乏可读性，富于表达。
- Scala 是纯粹的面向对象编程语言，所有的变量都是对象，所有的运算符都是方法。
- Scala 还是函数式编程语言，因此函数也是变量，因此你可以把函数作为参数传递给参数。你既可以写面向对象的代码、也可以写函数式编程的代码，当然也可以将它们混合在一起。
- Scala 的源码会被编译为 `.class` 文件以便在 Java 虚拟机（JVM）上执行。

- Scala 可以无缝的使用开发多年的、成熟的 Java 类库。
- Scala 最重要的特点就是你在上手的第一天就可以开始写生产代码，但是 Scala 是一款有深度的语言，随着使用 Scala 编程并深入学习它，你能够不断发现新大陆，找到更好的方式来编写代码。有些人说 Scala 将会使你重新思考编程的本质。（难道这不是最美好的事情吗？）
- Scala 最重要的好处就是让你可以编写出简洁、可读的代码。通常来说我们花在读代码上的时间是写代码的10倍，因此编写简介和可读的代码将会提高我们的效率。

2.3 HELLO，WORLD

自从 C 语言的发明人丹尼斯·里奇发布了《C程序设计语言》一书，在编程类书籍的开篇给出一个简单的“Hello, world”就成为了惯例。按照传统，下面给出 Scala 版本的 “Hello，world”：

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

用你喜欢的文本编辑器将上述代码保存到 `Hello.scala` 文件中，然后在命令行中用 `scalac` 命令编译该文件（当然要记得把命令行的当前窗口转到你保存文件的目录）：

```
$  scalac .\Hello.scala
```

`scalac` 类似于 `javac` ，该命令会创建两个新文件：

- `Hello$.class`
- `Hello.class`

这些文件与 `javac` 创建的 `.class` 的文件一样，都是字节码文件，可以在 JVM 上运行。

可以用 `scala` 命令来运行 `Hello` 应用程序：

```
$ scala Hello
```

2.3.1 代码解释

我们来简短解释一下这段代码：

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

- 在这段代码中，在一个名为 `Hello` 的 Scala `object` 中定义了一个名为 `main` 的方法。
- Scala 中的 `object` 类似于 `class` ，用于类只有一个实例的情况。
  - 如果你之前使用过 Java 语言，那么 `main` 方法类似于 Java 中的 `static` 方法（本书稍后会讨论这一话题）。
- `main` 函数有一个用于接受字符串数组的参数 `args` 。
- `Array` 类对 Java 的数组进行了封装。

这段 Scala 代码非常类似于下面的 Java 代码：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world")
    }
}
```

2.3.2 深入一点：揭秘 `Hello.class` 文件

使用 `scalac` 命令可以创建可以运行于 JVM 的字节码文件，可以用 `javap` 命令来看一下 `Hello.class` 文件的反编译代码：

> `javap` 是 Java 的反编译程序。

```
$ javap Hello.class
Compiled from "Hello.scala"
public final class Hello {
  public static void main(java.lang.String[]);
}
```

从上面的结果可以看到，`javap` 可以把 Scala 代码编译出来的 `.class` 文件反编译为 Java 代码。因为 Scala 代码可以运行于 JVM ，因此可以调用已有的 Java 库，为开发带来遍历。

2.4 HELLO, WORLD：第二个版本

Scala 提供了更方便的编写应用程序的方式，`object` 可以通过继承 `App` 特征实现同样的功能：

```
object Hello extends App {
  println("Hello, world")
}
```

将上面的代码保存到 `Hello.scala` 文件中，然后用 `scalac` 编译，最后用 `scala` 运行，会得到与上一个版本相同的结果。

在这段代码中，`App` 特征有自己的 `main` 方法，因此你不用再编写 `main` 方法。在本书的后面会介绍用这种方式如何访问命令行的参数。

> Scala 中的特征（`trait`）类似于 Java 中的抽象类。（更准确地说，应该是融合的 Java 中的抽象类和接口）

2.4.1 友情提示

如果想要知道如何在继承了 `App` 特征的 `object` 中访问命令行参数，可以直接访问 `args` ，将下面的代码保存到 `HelloYou.scala`：

```
object HelloYou extends App {
    if (args.size == 0)
        println("Hello, you")
    else
        println("Hello, " + args(0))
}
```

用 `scalac` 进行编译：

```
$ scalac .\HelloYou.scala
```

分别以不带参数和带参数的方式运行程序：

```
$ scala HelloYou
Hello, you

$ scala HelloYou All
Hello, All
```

从这个例子中可以看到：

- 命令行参数会被自动存储在名为 `args` 的变量中。
- 可以用 `args.size` 来获得 `args` 中的元素数量（也可以用 `args.length` ）。
- `args` 是一个 `Array` ，可以用 `args(0)` 、 `args(1)` 的方式访问 `Array` 中的元素。由于 `args` 是一个对象，因此访问数组元素的时候用小括号（而不是 `[]` ）。

2.5 Scala的REPL

Scala 的 REPL（Read-Evaluate-Print-Loop）是一个基于命令行的交互式环境（类似于在命令行中运行 `python` ）。要启动 REPL ，直接在操作系统的命令行中输入 `scala` ：

```
$ scala
Welcome to Scala 2.13.1 (Java HotSpot(TM) 64-Bit Server VM, Java 13.0.2).
Type in expressions for evaluation. Or try :help.

scala> _
```

REPL 是一个命令行的解释器，因此一旦启动之后，就可以在 REPL 中输入 Scala 表达式：

```
scala> val x = 1
x: Int = 1

scala> val y = x + 1
y: Int = 2
```

从上面的例子中可以看到，在 REPL 中输入表达式，会在下一行直接显示每个表达式的结果。

2.5.1 按需创建变量

在 REPL 中如果不把表达式赋给一个变量，那么 REPL 会自动创建以 `res` 开头的变量来存储表达式的值，第一个变量是 `res0` ，第二个变量是 `res1` ，以此类推。

```
scala> 2 + 2
res0: Int = 4

scala> 3 / 3
res1: Int = 1
```

这些变量都是动态创建的，你可以在之后的表达式中使用这些变量：

```
scala> val z = res0 + res1
z: Int = 5
```

在本书中，我们将使用 REPL 来执行代码，下面提供一些表达式，你可以尝试观察它们是如何工作的：

```
val name = "John Doe"
"hello".head
"hello".tail
"Hello, world".take(5)
println(5)
1 + 2 * 3
(1 + 2) * 3
if (2 > 1) println("greater") else println("lesser")
```

除了 REPL 之外，还有一些类似的工具可以使用：

- Scastie（https://scastie.scala-lang.org/）是一个基于 Web 的交互式的 Scala 实验环境。
- IntelliJ IDEA 提供了 Worksheet 插件可以提供近似的功能。
- Eclipse 的 Scala IDE 也提供了 Worksheet 插件。
- scalafiddle.io （https://scalafiddle.io/）提供了可以运行 Scala 代码的 web 页面。

更多关于 Scala REPL 的信息可以阅读： https://docs.scala-lang.org/overviews/repl/overview.html

2.6 两种变量

在 Java 中可以用下面的方式来创建变量：

```
String s = "hello";
int i = 42;
Person p = new Person("Joel Fleischman")
```

在 Scala 中有两种变量：

- `val` 用于创建不可变变量（类似 Java 中的 `final` ）
- `var` 用于创建可变变量

在 Scala 中声明变量如下例所示：

```
val s = "hello" // immutable
var i = 42 // mutable

val p = new Person("Joel Fleischman")
```

从上面的例子可以看出， Scala 的编译器通常可以通过 `=` 的右侧来推断变量的数据类型。我们称之为*由编译器推断变量类型*。也可以在代码中显式的声明变量的类型：

```
val s: String = "hello"
var i: Int = 42
```

大部分情况下，编译器不需要这些显式声明的数据类型，但是可以通过添加这些类型声明来提高代码的可读性。

> 在实践当中，通常在使用一些第三方的类库时显式的声明数据类型，尤其是你不经常使用这些库或者无法通过方法的名字断定其返回类型的时候。

2.6.1 `val` 和 `var` 的区别

`val` 和 `var` 之间的区别在于：`val` 声明的变量是不可变的，类似于 Java 中的 `final` 关键字，而 `var` 则用来声明可变的变量。由于 `val` 字段是不可变的，因此有些人更喜欢将其称之为值 `values` 而不是变量 `variables` 。

在 REPL 中可以观察到试图再次给 `val` 变量赋值会发生什么：

```
scala> val a = 'a'
a: Char = a

scala> a = 'b'
         ^
       error: reassignment to val
```

这会导致一个 `error: reassignment to val` 错误。与之相反，可以给 `var` 重新赋值：

```
scala> var a = 'a'
a: Char = a

scala> a = 'b'
a: Char = b
```

按照 Scala 的惯例，除非有合理的理由使用 `var` ，否则应该一直使用 `val` 。这条简单的规则可以让你的代码更像代数式，并且能够帮助你去理解函数式编程，因为在函数式编程范式中，所有的字段都是不可变的。

2.6.2 带有 `val` 字段的”Hello, world“

下面的的例子展示了一个带有 `val` 字段的 ”Hello, world“ 程序：

```
object Hello3 extends App {
  val hello = "Hello, world"
  println(hello)
}
```

然后：

- 将代码保存至 `Hello3.scala` 文件；
- 在命令行执行 `scalac ./Hello3.scala` 编译代码；
- 在命令行执行 `scala Hello3` 来运行代码。

2.6.3 在 REPL 中使用 `val` 变量的提示

REPL 的工作方式与 IDE 中编写源码的方式并不是完全相同，有些代码可以在 REPL 中运行，但是在真正的项目中是无法运行的。例如，在 REPL 中可以重新定义 `val` 变量：

```
scala> val age = 18
age: Int = 18

scala> val age = 19
age: Int = 19
```

与 REPL 不同，在真正的项目当中，`val` 变量是不能重新定义的。

2.7 可有可无的”类型“

在之前的例子中，可以在创建变量时显式的声明其数据类型：

```
val count: Int = 1
val name: String = "Alvin"
```

然而，在 Scala 中通常省略掉变量的数据类型，并由 Scala 推理它的类型：

```
val count = 1
val name = "Alvin"
```

在大部分情况下，省略数据类型可以提高代码的可读性，因此推荐由 Scala 来自动推断变量的类型。

2.7.1 显式类型声明太啰嗦

例如，下面的例子中很明显数据类型是 `Person`，因此完全没有在表达式的左侧声明类型：

```
val p = new Person("Candy")
```

与之相反，如果在变量后面加上类型声明，代码会显得过于啰嗦：

```
val p: Person = new Person("Leo")
```

即：

```
val p = new Person("Candy")          // 推荐的方式
val p: Person = new Person("Candy") // 不推荐，代码过于啰嗦
```

2.7.2 显式声明数据类型的时机：”模棱两可“

如果你想在代码中清楚的表示变量的数据类型时，那么就显式的声明数据类型。当不显式的声明数据类型，编译器就无法正确的推断数据类型时，就需要显式的声明变量的数据类型。例如，你要把数值型的变量声明为某种特定的数据类型的时候，在下一讲，我们会给出这方面的例子。

2.8 Scala 内建数据类型

Scala 提供了常用的标准数值类型，但是 Scala 中所有的数据类型都是对象（类似于 Java 中的包装类，而不是原生数据类型）

> 原生数据类型通常指的是 CPU 支持的数据类型。

下面的例子展示了如何声明这些基本数据类型：

```
val b: Byte = 1
val x: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 2.0
val f: Float = 3.0f
```

在上面的例子中，如果不显式的指定数据类型，数值 `1` 将被默认设置为 `Int` 类型，因此需要显式的声明数据类型。带小数点的数值（例如 `2.0` ）默认为 `Double` 类型，所以如果要使用 `Float` 类型的数值需要在后面添加 `f` 后缀。

因为 `Int` 和 `Double` 是默认的数值类型，因此创建这两种数据类型的变量不需要显式的声明数据类型：

```
val i = 123 // 默认为 Int 类型
val x = 1.0 // 默认为 Double 类型
```

再 REPL 中会显示上面的例子中变量的数据类型被设置为 `Int` 和 `Double` ：

```
scala> val i = 123
i: Int = 123

scala> val x = 1.0
x: Double = 1.0
```

这些标准数据类型的取值范围如下表所示：

数据类型	取值范围
Boolean	`true` 和 `flase`
Byte	8位带符号补码整数：$[-2^7 \sim 2^7 - 1]$，即 -128 到 127 。
Short	16位带符号补码整数：$[-2^{15} \sim 2^{15} - 1]$，即 -32768 到 32767 。
Int	32位带符号补码整数：$[-2^{31} \sim 2^{31} - 1]$，即 $-2,147,483,648$ 到 $2,147,483,647$ 。
Long	64位带符号补码整数：$[-2^{63} \sim 2^{63} - 1]$
Float	32位 IEEE 754 单精度浮点数， $1.40129846432481707e-45$ 到 $3.40282346638528860e+38$
Double	64位 IEEE 754 双精度浮点数，$4.94065645841246544e-324$ 到 $1.79769313486231570e+308$
Char	16位无符号 `Unicode` 字符：$[0 \sim 2^{16} - 1]$，即 0 到 65535 。
String	`Char` 序列

2.8.1 `BigInt` 和 `BigDecimal`

Scala 中使用 `BigInt` 和 `BigDecimal` 进行大数运算：

```
var b = BigInt(1234567890)
var b = BigDecimal(123456.789)
```

在 Scala 中 `BigInt` 和 `BigDecimal` 支持所有的用于数值计算的运算符：

```
scala> var b = BigInt(123456789)
b: scala.math.BigInt = 123456789

scala> b + b
res0: scala.math.BigInt = 246913578

scala> b * b
res1: scala.math.BigInt = 15241578750190521

scala> b += 1

scala> println(b)
123456790
```

2.8.2 `String` 与 `Char`

在 Scala 中通常隐式声明 `String` 和 `Char` 类型的变量：

```
val name = "Bill"
val c = 'a'
```

同样，如果有必要可以显式声明数据类型：

```
val name: String = "Bill"
val c: Char = 'a'
```

在 Scala 中用双引号表示字符串，用单引号表示字符。

2.9 字符串

Scala 字符串有很多有趣的性质，我们首先介绍两个最常用的特性。

2.9.1 字符串插值

第一个特性就是 Scala 拥有类似于 Ruby 语言的字符串拼接方法，例如有以下三个变量：

```
val firstName = "John"
val mi = "C"
val lastName = "Doe"
```

可以像 Java 那样将它们拼接在一起：

```
val name = firstName + " " + mi + " " + lastName
```

然而，在 Scala 中提供更为方面的形式：

```
val name = s"$firstName $mi $lastName"
```

用这种方式拼接字符串可以写出更具可读性的代码，例如：

```
println(s"Name: $firstName $mi $lastName")
```

如前面代码所示，我们需要在字符串之前加上字母 `s` ，然后在字符串中在变量名之前添加 `$` 符号，这一特性别成为**字符串插值**。

Scala 中的字符串插值还有很多其他特性，例如可以用花括号将变量名包括起来：

```
println(s"Name: ${firstName} ${mi} ${lastName}")
```

对于部分人来说，这样做可以提高代码的可读性，但是这样写最重要的好处是在大括号之间可以直接写表达式：

```
scala> println(s"1+1 = ${1+1}")
1+1 = 2
```

字符串插值还有一些其他特性：

- 用字母 `f` 作为字符串的前导字符，这种字符串允许使用 `printf` 风格的格式字符串（就是 C 语言的 `printf` ）。
- 以 `raw` 作为字符串的前缀可以忽略转义字符（在正则表达式中尤为方便）。
- 你可以自己构造字符串插值。

2.9.2 多行字符串

Scala 字符串的第二个特性就是可以用三个双引号创建多行字符串：

```
val speech = """Four score and
               seven years ago
               our fathers ..."""
```

这对于处理多行文本非常实用，但是这种特性有一个缺点就是第一行之后会有缩进，在 REPL 中我们会观察到：

```
scala> val speech = """ Four score and
     |                  seven years ago
     |                  our fathers ..."""
speech: String =
" Four score and
                      seven years ago
                 our fathers ..."
```

一个解决这个问题的简单方法就是在第一行之外的每一行前面添加一个 `|` 字符，然后在字符串后面调用 `stripMargin` 方法：

```
val speech =  """ Four score and
                |seven years age
                |our fathers ...""".stripMargin
```

在 REPL 中可以看到所有行都是左对齐的。

```
scala> val speech =  """ Four score and
     |                  |seven years age
     |                  |our fathers ...""".stripMargin
speech: String =
" Four score and
seven years age
our fathers ..."
```

通常我们就是需要这样的结果，因此这种创建多行字符串的方法在 Scala 中被广泛使用。

2.10 命令行输入输出

在介绍 `for` 循环、`if` 表达式以及其他 Scala 结构控制语句之前，首先介绍一下在 Scala 中如何处理命令行的输入和输出。

2.10.1 输出

在前面的例子中，我们使用 `println` 将信息输出到标准输出（STDOUT）上。

```
println("Hello, world")
```

该函数在输出字符串后会在后面加上一个换行符，如果不想换行可以使用 `print`：

```
print("Hello without newline")
```

如果需要，还可以将信息输出到标准错误（STDERR）上：

```
System.err.println("yikes, an error happened")
```

> 由于 `println` 使用频繁，因此在 Scala 中不需要导入它，其他一些常用的数据类型也无需导入就能使用，例如 `String`、`Int`、`Float` 等。

2.10.2 输入

有几种不同的方法可以读取命令行的输入，但是最简单的方式是使用 `readLine` 方法，该方法属于 `scala.io.StdIn` 包，要使用该方法要首先导入它：

```
import scala.io.StdIn.readLine
```

为了演示如何使用该方法，我们创建一个名为 `HelloInteractive.scala` 文件：

```scala
import scala.io.StdIn.readLine

object HelloInteractive extends App{

  print("Enter your first name: ")
  val firstName = readLine()

  print("Enter your last name: ")
  val lastName = readLine()

  println(s"Your name is $firstName $lastName")
}
```

然后用 `scalac` 命令编译：

```
$ scalac .\HelloInteractive.scala
```

然后用 `scala` 运行：

```
$ scala HelloInteractive
```

运行程序后，按照提示输入，执行结果如下图所示：

```
$ scala HelloInteractive
Enter your first name: Alvin
Enter your last name: Alexander
Your name is Alvin Alexander
```

和 Java 等语言一样，在 Scala 中需要使用 `import` 语句将类和方法导入：

```scala
import scala.io.StdIn.readLine
```

这条重要的语句将 `readLine` 方法导入，从而在应用程序中调用它。

3. 结构控制语句

Scala 和其他语言相同提供了基本的结构控制语句：

- `if/then/else`
- `for` 循环
- `try/catch/finally`

此外还有一些独有的控制结构：

- `match` 表达式
- `for` 表达式

下面我们将介绍 Scala 的结构控制语句。

3.1 分支结构

Scala 中最基本的 `if` 语句的格式为：

```
if (a == b) doSomething()
```

上面的语句也可以写成下面的形式：

```
if (a == b) {
    doSomething()
}
```

`if` / `else` 结构如下所示：

```
if (a == b) {
    doSomething()
} else {
    doSomethingElse()
}
```

完整的 Scala `if/else-if/else` 表达式的格式如下：

```
if (test1) {
    doX()
} else if (test2) {
    doY()
} else {
    doZ()
}
```

3.1.1 `if` 表达式

在 Scala 中 `if` 结构语句是有返回值的，尽管可以像前面的例子一样忽略这些结果，但是通常的做法是将 `if` 语句的返回值赋给变量，尤其是在函数式编程中：

```
val minValue = if (a < b) a else b
```

这就意味着 Scala 不需要三元运算符（回想一下 C 语言中的 `condition ? expressionA : expressionB`）。

3.1.2 面向表达式编程

通常来说在编程范式中，如果每个表达式都有返回值，这种风格被称为面向表达式编程（`expression-oriented programming, EOP`），如下例所示：

```
val minValue = if (a < b) a else b
```

相反，如果代码中每一行都没有返回值则称为语句，语句通过"副作用"来实现计算。

> 副作用（`side-effects`）：简单的说，副作用就是指函数或者表达式修改基本变量的行为，即内存原始的值在计算后被修改了。

> 例如 C 语言中 `+=` 、`++` 是有副作用的计算，它们会修改操作数的值。而 `+` 等运算符则不会修改操作数的值。

例如，下面的代码每一行都没有返回值，它们依靠“副作用”来实现功能：

```
if (a == b) doSomething()
println("Hello")
```

第一行代码当 `a` 等于 `b` 时运行 `doSomething` 作为“副作用”。第二行代码则利用了将字符串写入到 STDOUT 的“副作用”。随着你深入学习 Scala，你将编写更多的表达式而非语句，然后更深入的理解表达式与语句差别。

3.2 FOR 循环

最简单的 `for` 循环的用法就是迭代访问集合中的每个元素，例如给定一个整数序列：

```
val nums = Seq(1, 2, 3)
```

可以用 `for` 循环遍历其元素并打印：

```
for (n <- nums) println(n)
```

下面是在 REPL 中运行的结果：

```
scala> val nums = Seq(1, 2, 3)
nums: Seq[Int] = List(1, 2, 3)

scala> for (n <- nums) println(n)
1
2
3
```

上面的例子中使用了类型为 `Seq[Int]` 的整数序列。下面的例子中创建了一个 `List[String]` 类型的字符串列表：

```
val people = List(
  "Bill",
  "Candy",
  "Karen",
  "Leo",
  "Regina"
)
```

与输出整数序列的例子相似，可以用 `for` 循环来输出这些字符串：

```
for (p <- people) println(p)
```

> `Seq` 和 `List` 是两种线性集合，在 Scala 中我们更喜欢使用这些集合类而不是 `Array` 。

3.2.1 foreach 方法

如果要迭代集合的所有元素并输出，还可以使用 Scala 中集合类提供的 `foreach` 方法实现这一功能。下面的例子展示了如何用 `foreach` 方法来输出前面的字符串列表：

```
people.foreach(println)
```

Scala 中的序列、映射和集合等大部分的集合类都支持 `foreach` 方法。

3.2.2 用 `for` 和 `foreach` 访问映射（Maps）

在 Scala 中可以使用 `for` 和 `foreach` 来访问 `Map` （类似于 Java 中的 `HashMap` ）。下面的例子给出了一个包含了电影名字和评分的 `Map` ：

```
val ratings = Map(
  "Lady in the Water"  -> 3.0,
  "Snakes on a Plane"  -> 4.0,
  "You, Me and Dupree" -> 3.5
)
```

可以用下面的 `for` 循环来打印电影的名字和评分：

```
for ((name, rating) <- ratings) println(s"Movie: $name, Rating: $rating")
```

下面是在 REPL 中运行的结果：

```
scala> for ((name, rating) <- ratings) println(s"Movie: $name, Rating:
$rating")
Movie: Lady in the Water, Rating: 3.0
Movie: Snakes on a Plane, Rating: 4.0
Movie: You, Me and Dupree, Rating: 3.5
```

在这个例子中， `name` 对应着映射中的 `key` ，而 `rating` 则对应着映射中的 `value` 。

也可以使用 `foreach` 来打印电影的评分：

```
ratings.foreach {
  case(movie, rating) => println(s"key: $movie, value: $rating")
}
```

3.3 `FOR` 表达式

如果你还记得我们之前介绍的面向表达式编程，以及表达式和语句的区别，那么就能发现上一讲介绍的 `for` 关键字和 `foreach` 方法本质是基于副作用（ `Side-effects` ）的。我们使用 `println` 将集合中的值打印到标准输出 STDOUT 。Java 里也有类似的关键字，许多程序员使用这些关键字多年却未曾思考过如何改进它。

使用 Scala 的函数式编程的特性，你会体验到比 `for` 循环更为强大的 `for` 表达式。与 `for` 循环不同， `for` 表达式会依据已有的集合来创建新的集合，而不是像 `for` 循环那样利用 `Side Effects` （例如打印输出）。

> 如果你还记得 Python 的列表生成式，这一小节将不会有任何困扰你的地方。

例如，给定下面的整数列表：

```
val nums = Seq(1, 2, 3)
```

你可以创建一个新列表，列表中的每个元素都是原始列表的两倍：

```
val doubledNums = for (n <- nums) yield n * 2
```

上面的表达式可以解释为："对于列表 `nums` 中的每个数字 `n`，乘以 `2`，然后将所有新得到值赋给一个新的名为 `doubledNums` 的变量"。 在 REPL 中可以看到如下结果：

```
scala> val doubledNums = for (n <- nums) yield n * 2
doubledNums: Seq[Int] = List(2, 4, 6)
```

如 REPL 的计算结果所示，新的列表 `doubledNums` 包含以下的值：

```
List(2, 4, 6)
```

3.3.1 处理字符串列表

可以用类似的方法来处理字符串列表，例如，给定如下由小写字母构成的字符串列表：

```
val names = List("adam", "david", "frank")
```

用 `for` 表达式将所有人名字的首字母大写：

```
val unNames = for (name <- names) yield name.capitalize
```

在 REPL 中运行的结果如下所示：

```
scala> val unNames = for (name <- names) yield name.capitalize
unNames: List[String] = List(Adam, David, Frank)
```

可以看到，新的列表 `unNames` 中每个字符串的首字母都是大写的。

3.3.2 `yield` 关键字

在前面的两个例子中，我们都使用了 `yield` 关键字：

```
val doubledNums = for (n <- nums) yield n * 2

val ucNames = for (name <- names) yield name.capitalize
```

在 `for` 表达式中使用 `yield` 关键字意味着："我想要通过 `yield` 后面的算法对集合进行迭代，从而生成一个新的集合。"

3.3.3 `yield` 代码块

`yield` 表达式可以根据要解决的问题来确定表达式的复杂程度，表达式可能非常长。例如，给定如下字符串列表：

```
val names = List("_adam", "_david", "_frank")
```

假设我们现在要创建一个名字首字母都大写的新列表。为了实现这一目标，首先要移除每个名字开始的下划线，然后再将名字的首字母大写。要移除每个名字前面的下划线，可以在每个字符串后面调用 `drop(1)` 方法，然后再用 `capitalize` 方法将首字母大写。下面的例子用 `for` 表达式解决了这个问题：

```
val capNames = for (name <- names) yield {
  val nameWithoutUnderscore = name.drop(1)
  val capName = nameWithoutUnderscore.capitalize
  capName
}
```

在 REPL 中执行上面的代码可以得到如下结果：

```
capNames: List[String] = List(Adam, David, Frank)
```

> 前面使用多行代码的版本展示了如何在 `yield` 后面写多行代码，但是对于这个例子来说这样的代码太繁冗了，更为 Scala 的写法如下：
>
> ```
> val capNames = for (name <- names) yield name.drop(1).capitalize
> ```
>
> 如果愿意还可以把算法部分用花括号括起来：
>
> ```
> val capNames = for (name <- names) yield { name.drop(1).capitalize }
> ```

3.4 MATCH 表达式

Scala 支持 `match` 表达式，在简单的情况下，你可以像使用 Java 的 `switch` 语句一样来使用 `match` 表达式：

```
// i is an integer
i match {
    case 1  => println("January")
    case 2  => println("February")
    case 3  => println("March")
    case 4  => println("April")
    case 5  => println("May")
    case 6  => println("June")
    case 7  => println("July")
    case 8  => println("August")
    case 9  => println("September")
    case 10 => println("October")
    case 11 => println("November")
    case 12 => println("December")
    // catch the default with a variable so you can print it
    case _  => println("Invalid month")
}
```

如上例所示，一个 `match` 表达式中包含多个 `case` 语句以匹配不同的值。在这个例子中，我们匹配 `1` 到 `12` 之间的整数，其余的值将匹配到 `case _` 上，这条语句将作为默认情况。

`match` 表达式更为有用的特性是可以直接返回值，而不是直接像上面的例子一样打印字符串。可以将 `match` 的返回字符串赋给一个新的变量：

```
val monthName = i match {
    case 1  => "January"
    case 2  => "February"
    case 3  => "March"
    case 4  => "April"
    case 5  => "May"
    case 6  => "June"
    case 7  => "July"
    case 8  => "August"
    case 9  => "September"
    case 10 => "October"
    case 11 => "November"
    case 12 => "December"
    case _  => "Invalid month"
}
```

用 `match` 表达式生成一个结果是最常见的用法。

3.4.1 题外话：Scala 中的方法

在 Scala 中可以用 `match` 表达式作为方法的方法体。由于我们还没有介绍如何编写 Scala 方法，所以首先做一个简单的介绍，下面的例子是一个叫做 `convertBooleanToStringMessage` 的方法，该方法接收一个 `Boolean` 值，返回一个 `String` ：

```
def convertBooleanToStringMessage(bool: Boolean): String = {
  if (bool) "true" else "false"
}
```

希望你能从上面的代码中看出方法是如何工作的，给该方法分别传递 `Boolean` 值 `true` 和 `false` ：

```
scala> val answer = convertBooleanToStringMessage(true)
answer: String = true

scala> val answer = convertBooleanToStringMessage(false)
answer: String = false
```

3.4.2 用 `match` 表达式作为方法体

下面的例子与前面的例子相似，区别在于用 `match` 表达式作为方法体：

```
def convertBooleanToStringMessage(bool: Boolean): String = bool match {
  case true => "you said true"
  case false => "you said false"
}
```

上面的例子中方法体只有两个 `case` 语句，一个匹配 `true` ，另一个匹配 `false` 。因为 `Boolean` 类型的值只有这两种可能，因此不需要默认的 `case` 语句。

下面是调用该方式的例子：

```
val result = convertBooleanToStringMessage(true)
println(result)
```

用 `match` 作为方法体是一种常用的方法。

3.4.3 处理备用情况

`match` 表达式非常强大，下面我们将要介绍它的一些特性。

`match` 表达式允许在一条 `case` 语句中处理多种情况。例如，在 Perl 语言中 `0` 和空字符串都被认为是 `false` ，其他的值被认为是 `true` 。接下来，我们用 `match` 表达式实现这一功能：

```
def isTrue(a: Any) = a match {
  case 0 | "" => false
  case _ => true
}
```

由于输入参数 `a` 的类型是 `Any` 类型 —— 这是所有 Scala 类的基类，类似于 Java 中的 `Object` ，因此该方法可以处理任何类型的参数：

```
scala> isTrue(0)
res1: Boolean = false

scala> isTrue("")
res2: Boolean = false

scala> isTrue(1.1F)
res3: Boolean = true

scala> isTrue(new java.io.File("C:\\windows"))
res4: Boolean = true
```

上面的例子中最核心的部分就是用一条 `case` 语句同时处理 `0` 和空字符串，并返回 `false` ：

```
case 0 | "" => false
```

在我们继续下一个话题之前，在看一些例子：

```
val evenOrOdd = i match {
  case 1 | 3 | 5 | 7 | 9 => "odd"
  case 2 | 4 | 6 | 8 | 10 => "even"
  case _ => "some other number"
}
```

下面的例子展示了如何在多个 `case` 语句中匹配多个字符串：

```
val result = cmd match {
  case "start" | "go" => "starting"
  case "stop" | "quit" | "exit" => "stopping"
  case _ => "doing nothing"
}
```

3.4.4 在 `case` 语句中使用 `if` 表达式

`match` 表达式另外一个重要特性是可以在 `case` 语句中使用 `if` 表达式来增强模式匹配的能力。下面的例子中的第二个和第三个 `case` 语句都使用了 `if` 表达式来匹配数值范围：

```
val result = count match {
  case 1 => "one, a lonely number"
  case x if x == 2 || x == 3 => "two's company, three's a crowd"
  case x if x > 3 => "4+, that's a party"
  case _ => "i'm guessing your number is zero or less"
}
```

在 Scala 中，`if` 表达式不需要用小括号将条件括起来，但是可以用小括号把条件括起来提高可读性：

```
val result = count match {
  case 1 => "one, a lonely number"
  case x if (x == 2 || x == 3) => "two's company, three's a crowd"
  case x if (x > 3) => "4+, that's a party"
  case _ => "i'm guessing your number is zero or less"
}
```

为了提高可读性，可以把 `=>` 右侧写成多行的形式：

```
val result = count match {
  case 1 =>
    "one, a lonely number"
  case x if (x == 2 || x == 3) =>
    "two's company, three's a crowd"
  case x if (x > 3) =>
    "4+, that's a party"
  case _ =>
    "i'm guessing your number is zero or less"
}
```

下面是带有花括号的版本：

```
val result = count match {
  case 1 => {
    "one, a lonely number"
  }
  case x if (x == 2 || x == 3) => {
    "two's company, three's a crowd"
  }
  case x if (x > 3) => {
    "4+, that's a party"
  }
  case _ => {
    "i'm guessing your number is zero or less"
  }
}
```

下面再看两个在 `case` 语句中使用 `if` 表达式的例子。下面的例子展示了如何判断数值区间：

```
i match {
  case a if 0 to 9 contains a => println("0-9 range: " + a)
  case b if 10 to 19 contains b => println("10-19 range: " + b)
  case c if 20 to 29 contains c => println("20-29 range: " + c)
  case _ => println("Hmmm...")
}
```

最后一个例子展示了如何在 `if` 表达式中访问类的字段：

```
stock match {
  case x if (x.symbol == "XYZ" && x.price < 20) => buy(x)
  case x if (x.symbol == "XYZ" && x.price > 50) => sell(x)
  case x => doNothing(x)
}
```

3.5 `TRY/CATCH/FINALLY` 表达式

与 Java 类似， Scala 也提供了 `try/catch/finally` 结构来捕获和处理异常。与 Java 语言的区别在于， Scala 中的 `catch` 子句采用了类似于 `match` 表达式中 `case` 语句的语法结构来匹配不同的异常。

3.5.1 `try/catch`

下面的例子中使用了 Scala 的 `try/catch` 语法结构，在这个例子中， `openAndReadAFile` 方法的功能正如其名字所暗示的那样：打开一个文件，然后读取文件内容：

```
var text = ""
try {
  text = openAndReadAFile(filename)
} catch {
  case e: FileNotFoundException => println("Coundn't find that file.")
  case e: IOException => println("Had an IOException tring to read that
file.")
}
```

Scala 使用 `java.io.*` 中的类来处理文件，因此尝试打开和读取文件可能会导致 `FileNotFoundException` 和 `IOException` 。上面的例子用 `catch` 子句捕获这两种异常。

3.5.2 `try/catch/finally`

Scala 中的 `try/catch` 语句还可以有 `finally` 子句，我们通常用该子句来关闭资源，语法格式如下例所示：

```
try {
  // your scala code here
} catch {
  case foo: FooExcetpion => handleFooExcption(foo)
  case bar: BarException => handleBarExcption(bar)
  case _: Throwable => println("Got some other kind of Throwable excption")
} finally {
  // your scala code here, such as closing a database connection
  // or file handle
}
```

Scala 中的 `try/catch/finally` 语法结构与 `match` 表达式的语法结构具有高度的一致性，这有助于我们在写出具有一致性和可读性的代码的同时，无需记住额外的语法要求。

4. 类

Scala 提供了 `class` 语法以提供面向对象的编程范式，其语法要比 Java 或 C# 这类的语言要更为简洁和易于阅读。

4.1 类的定义

4.1.1 类的基本构造器

在下面的例子中， `Person` 类定义了两个参数： `fisrtName` 和 `lastName` ：

```
class Person(var firstName: String, var lastName: String)
```

在定义了 `Person` 类后，可以用如下的方式创建 `Person` 实例：

```
val p = new Person("Bill", "Panner")
```

Scala 会根据在类的构造器中定义的参数自动创建类的字段。在下面的例子中我们访问了 `firstName` 和 `lastName` 字段：

```
println(p.firstName + " " + p.lastName)
```

在本例中，使用的 `var` 关键字定义的字段，因此它们是可变的，意味着可以改变它们的内容，例如：

```
scala> p.firstName = "William"
p.firstName: String = William

scala> p.lastName = "Bernheim"
p.lastName: String = Bernheim
```

如果你熟悉 Java ，那么下面的代码：

```
class Person(var firstName: String, var lastName: String)
```

近似下面的 Java 代码：

```java
public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return this.firstName;
    }

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

    public String lastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

4.1.2 用 `val` 定义只读字段

在上面的例子中，我们的两个字段的定义都使用的是 `var` 关键字：

```
class Person(var firstName: String, var lastName: String)
```

这样定义的字段都是可变的。也可以使用 `val` 关键字将字段定义为不可变字段：

```
class Person(val firstName: String, val lastName: String)
             ---                    ---
```

依然使用与上例相同的方式来构造实例：

```
val p = new Person("Tom", "Hanks")
```

但是如果视图修改实例中 `firstName` 和 `lastName` 字段，将会导致错误：

```
scala> p.firstName = "Fred"
                   ^
       error: reassignment to val

scala> p.lastName = "Jones"
                  ^
       error: reassignment to val
```

> 如果你用 Scala 编写面向对象的代码，就用 `var` 来定义字段以便修改他们。如果你用 Scala 写函数式编程的代码，通常使用样例类（`case classes`）而不是类（`calsses`）。

4.1.3 类的构造器

在 Scala 中，类的主构造器由以下部分构成：

- 构造器参数
- 在类中定义的方法（`Method`）
- 在类中编写的语句和表达式

在 Scala 的类中定义的字段与 Java 中定义的字段类似，它们在类被初始化的时候进行赋值。下面的例子展示了包含以上元素的 `Person` 类：

```
class Person(var firstName: String, var lastName: String) {
  println("The constructor begins ...")

  // 默认为 public 访问权限
  var age = 0

  // 私有字段
  private val HOME = System.getProperty("user.home")

  // 一些方法
  override def toString: String = s"$firstName $lastName is $age years old"

  def printHome(): Unit = println(s"HOME = $HOME")
  def printFullName(): Unit = println(this)

  printHome()
  printFullName()
  println("you've reached the end of the constructor")
}
```

定义类之后，就可以实例化并使用：

```
scala> val p = new Person("Kim", "Carnes")
The constructor begins ...
HOME = C:\Users\liuii
Kim Carnes is 0 years old
you've reached the end of the constructor
p: Person = Kim Carnes is 0 years old

scala> p.age
res4: Int = 0

scala> p.age = 36
mutated p.age

scala> p
res5: Person = Kim Carnes is 36 years old

scala> p.printHome
HOME = C:\Users\liuii

scala> p.printFullName
```

```
Kim Carnes is 36 years old
```

如果你用过比较“啰嗦”的编程语言（Java），Scala 的构造方法看上去有点“怪”，但是一旦你自己写上几个类并慢慢理解，你就能发现这种语法是具有逻辑性和便利性。

4.1.4 其他的例子

在继续下一节之前，我们再看几个 Scala 类的例子：

```
class Pizza (var crustSize: Int, var crustType: String)

// 股票类，通常上市股票会有一个缩写符号表示该公司，比如“莆田医疗”？
class Stock(var symbol: String, var price: BigDecimal)

// 一个网络套接字，包含了超时和延迟字段
class Socket(val timeout: Int, val linger: Int) {
  override def toString: String = s"timeout: $timeout, linger: $linger"
}

class Address (
              var street1: String,
              var street2: String,
              var city: String,
              var state: String
              )
```

4.2 类的辅助构造器

在 Scala 中可以通过定义名为 `this` 的方法来定义类的辅助构造器，类的辅助构造器需要满足以下规则：

- 每个辅助构造器都必须有一个不同的方法签名（不同的参数列表）。
- 每个辅助构造器必须要调用一个之前定义的构造器。

下面给出一个定义了多个辅助构造器的 `Pizz` 类的示例：

```
val DefaultCrustSize = 12
val DefaultCrustType = "THIN"

// 基本构造器
class Pizza (var crustSize: Int, var curstType: String) {

  // 包含一个参数的辅助构造器
  def this(crustSize: Int) {
    this(crustSize, DefaultCrustType)
  }

  // 包含一个参数的辅助构造器
  def this(crustType: String) {
    this(DefaultCrustSize, crustType)
  }

  // 无参数的辅助构造器
  def this() {
```

```
    this(DefaultCrustSize, DefaultCrustType)
  }

  override def toString: String = s"A $crustSize inch pizza with a
$curstType crust"

}
```

在定义了上面的构造器之后，就可以用以下不同的方式来构造 `Pizza` 的实例：

```
val p1 = new Pizza(DefaultCrustSize, DefaultCrustType)
val p2 = new Pizza(DefaultCrustSize)
val p3 = new Pizza(DefaultCrustType)
val p4 = new Pizza
```

对于这个例子，我们要强调两点：

- 由于尚未介绍 Scala 中的**枚举**，因此 `DefaultCrustSize` 和 `DefaultCrustType` 变量的处理采用了直接赋值的方式，但是在这个例子的应用场景中并不推荐这样做。
- 辅助构造器是一个不错的语言特性，但是由于我们可以给构造器参数设置默认值，因此实际上我们很少需要创建辅助构造器，上面例子中的辅助构造器也可以用下面的方式替代：

```
class Pizza (
            var crustSize: Int = DefaultCrustSize,
            var crustType: String = DefaultCrustType
            )
```

4.3 为基本构造器的参数设置默认值

Scala 允许为类的基本构造器中的参数设置默认值，例如上一小节中，我们定义了一个 `Socket` 类：

```
class Socket(var timeout: Int, var linger: Int) {
  override def toString: String = s"timeout: $timeout, linger: $linger"
}
```

在 Scala 中可以为 `timeout` 和 `linger` 参数设置默认值：

```
class Socket(var timeout: Int = 2000, var linger: Int = 3000) {
  override def toString: String = s"timeout: $timeout, linger: $linger"
}
```

在设置了默认值后可以用多种形式来创建 `Socket` 实例：

```
new Socket()
new Socket(1000)
new Socket(4000, 6000)
```

在 REPL 中运行可以得到如下结果：

```
scala> new Socket()
res0: Socket = timeout: 2000, linger: 3000

scala> new Socket(1000)
res1: Socket = timeout: 1000, linger: 3000

scala> new Socket(4000, 6000)
res2: Socket = timeout: 4000, linger: 6000
```

为构造器中的参数提供默认值至少有两点好处：

- 通过为参数设置默认值，提供推荐值。
- 可以让这些类的使用者根据自己的需要覆盖这些值。

此外，在 Scala 中创建类的实例时，可以使用参数的名字。例如可以用如下方式来创建一个新的 `Socket` 类：

```
val s = new Socket(timeout = 1000, linger = 2000)
```

很多人认为这样编写的代码要比下面的代码更具可读性：

```
val s = new Socket(1000, 2000)
```

4.4 初探 Scala 方法

在 Scala 中方法通常定义在类中（类似于 Java ），但是在 REPL 中出于测试的目的可以直接创建方法。本节将展示一些方法的例子，以便你能了解方法的语法格式。

4.4.1 定义带有一个参数的方法

下面的例子中定义了一个名为 `double` 的方法，包含一个整数类型且名为 `a` 的参数，然后返回整数类型的两倍的 `a` ：

```
def double(a: Int) = a * 2
```

在这个例子中，方法的名字与签名在 `=` 符号的左侧：

```
def double(a: Int) = a * 2
    --------------
```

在这个例子中：

- 使用 `def` 关键字来定义方法；
- 方法的名字为 `double`
- 参数的名字为 `a`
- 参数的类型为 `Int` ，Scala 中的整数类型

函数体写在 `=` 符号的右边，在这个例子中就是简单的返回输入参数 `a` 的两倍：

```
def double(a: Int) = a * 2
                     -----
```

把上面定义的方法拷贝到 REPL 中，然后就可以对其进行调用：

```
scala> double(2)
res0: Int = 4

scala> double(10)
res1: Int = 20
```

4.4.2 方法的返回值类型

在前面的例子中没有给出方法的返回值类型，可以将上面的例子改写为：

```
def double(a: Int): Int = a * 2
                  -----
```

在 Scala 中可以显式的声明方法的类型，部分人认为显式声明方法的类型有助于代码的长期维护。将上面的方法拷贝到 REPL 中，能够看到运行的结果与之前的例子是完全一样的。

4.4.3 带有多个参数的方法

下面看一下稍微复杂一点的例子，这个例子中方法包含两个输入参数：

```
def add(a: Int, b: Int) = a + b
```

下面的例子是显式声明返回类型的版本：

```
def add(a: Int, b: Int): Int = a + b
```

下面是一个带有三个参数的例子：

```
def add(a: Int, b: Int, c: Int): Int = a + b + c
```

4.4.4 多行方法

如果方法只有一行可以使用上面给出的格式，但是如果方法体包含的语句很多，需要用花括号 `{}` 将它们括起来：

```
def addThenDouble(a: Int, b: Int): Int = {
    val sum = a + b
    val doubled = sum * 2
    doubled
}
```

在 REPL 中运行如下的调用可以得到对应的结果：

```
scala> addThenDouble(1, 1)
res0: Int = 4
```

4.5 枚举

本节将介绍枚举的概念，然后给出一个符合面向对象编程范式的 `Pizza` 类。

枚举（`Enumerations`）用于创建一小组常量，例如一周中的星期、一年中的月份、扑克牌的花色等等。枚举适用于创建一组相关的常量。

下面的示例展示了如何创建星期的枚举，因为涉及到之后章节要讨论的内容，在本节不会对其语法进行详细的解释：

```
sealed trait DayOfWeek
case object Sunday extends DayOfWeek
case object Monday extends DayOfWeek
case object Tuesday extends DayOfWeek
case object Wednesday extends DayOfWeek
case object Thursday extends DayOfWeek
case object Friday extends DayOfWeek
case object Saturday extends DayOfWeek
```

在上面的代码中首先声明一个特征（`trait`），然后从这个特征派生出多个我们需要的样例对象（`case object`）。

与之类似，下面的例子从 `Suit` 特征中派生了一组扑克牌的花色：

```
sealed trait Suit
case object Clubs extends Suit
case object Spades extends Suit
case object Diamonds extends Suit
case object Hearts extends Suit
```

这就是在 Scala 中创建枚举的方式，在后面的章节将会讨论特征和样例对象。接下来将创建面向对象版本的 `Pizza` 类。

4.5.1 和披萨相关的枚举

首先给出与 `Pizza` 相关的枚举：配料（`Topping`）、饼皮尺寸（`Crust Size`）和饼皮类型（`Crust Type`）。假设我们的披萨的配料包括：乳酪、意大利辣肉肠、香肠、蘑菇和洋葱，饼皮的尺寸包括大、中、小三个尺寸，饼皮类型包括正常、薄皮和加厚三种类型。

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

这些枚举对我们操作披萨的配料、饼皮尺寸和饼皮类型非常有帮助。

4.5.2 披萨类

有了上面定义的枚举，接下来就可以定义如下的 `Pizza` 类：

```
class Pizza (
            var crustSize: CrustSize = MediumCrustSize,
            var crustType: CrustType = RegularCrustType
            ) {

  // ArrayBuffer 是一个可变的序列（列表）
  val toppings = scala.collection.mutable.ArrayBuffer[Topping]()

  def addTopping(t: Topping): Unit = toppings += t
  def removeTopping(t: Topping): Unit = toppings -= t
  def removeAllTopping(): Unit = toppings.clear()

}
```

如果将包括枚举在内的所有代码保存到 `Pizza.scala` 文件中，然后就可以对其进行编译：

```
$ scalac ./Pizza.scala
```

> 注意：这段代码编译后将产生大量的文件，因此建议将此文件放置在一个单独的文件夹中。

由于没有 `main` 方法，编译好之后的类无法运行。

4.5.3 添加 `main` 方法

下面给出了 `Pizza.scala` 的完整代码：

```
import scala.collection.mutable.ArrayBuffer

sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType

class Pizza (
            var crustSize: CrustSize = MediumCrustSize,
            var crustType: CrustType = RegularCrustType
            ) {

  // ArrayBuffer 是一个可变的序列（列表）
  val toppings = ArrayBuffer[Topping]()
```

```
27
28    def addTopping(t: Topping): Unit = toppings += t
29    def removeTopping(t: Topping): Unit = toppings -= t
30    def removeAllToppings(): Unit = toppings.clear()
31
32    override def toString: String = {
33      s"""
34         |Crust Size: $crustSize
35         |Curst Type: $crustType
36         |Toppints:   $toppings
37         |""".stripMargin
38    }
39  }
40
41  // 代码执行的入口
42  object PizzaTest extends App{
43    val p = new Pizza
44    p.addTopping(Cheese)
45    p.addTopping(Pepperoni)
46    println(p)
47  }
```

将所有的枚举、`Pizza` 类和一个 `PizzaTest` 对象放入到同一个文件中，这是 Scala 的一项非常方便的特性。

接下来对代码进行编译：

```
1  $ scalac ./Pizza.scala
```

然后运行 `PizzaTest` 对象：

```
1  $ scala PizzaTest
```

程序的输出如下：

```
1  $ scala PizzaTest
2
3  Crust Size: MediumCrustSize
4  Curst Type: RegularCrustType
5  Toppints:   ArrayBuffer(Cheese, Pepperoni)
```

这段代码使用了很多不同的概念，其中包括尚未讨论的 `import` 语句和 `ArrayBuffer` 。如果你之前学习过 Java 等其他语言，希望你能够看懂这个例子。

最后，我们建议你对上面的代码尝试进行修改，例如尝试调用 `removeTopping` 和 `removeAllToppings` 方法。

5. 特征和抽象类

特征（`trait`）是 Scala 语言的一个重要特性，Scala 中的特征类似于 Java 中接口的概念，但是又和抽象类近似具有非抽象的方法。 Scala 中的类可以继承自特征，同时也可以实现多个特征。

Scala 中还有抽象类的概念，我们在接下来的内容中会介绍需要使用抽象类的情况。

5.1 像接口一样使用特征

在 Scala 中可以像在 Java 中使用接口一样来使用特征。即当你需要定义一些功能，但是又无需实现该功能的时候，就可以是用特征。

5.1.1 简单的例子

我们用一个例子来开始介绍特征的概念，例如你要实现一些诸如猫、狗等的动物类，而任何动物都有尾巴，那么在 Scala 中可以从定义“摇尾巴”的特征开始建模过程：

```
trait TailWagger {
  def startTail(): Unit
  def stopTail(): Unit
}
```

上面的代码定义了一个名为 `TailWagger` 的特征，意味着任何派生自 `TailWagger` 的类都需要实现 `startTail` 和 `stopTail` 方法。这两个方法都没有输入参数，也没有返回值。上面的代码等价于下面的 Java 代码：

```
public interface TailWagger {
    public void startTail();
    public void stopTail();
}
```

5.1.2 继承特征

给定如下特征：

```
trait TailWagger {
  def startTail(): Unit
  def stopTail(): Unit
}
```

就可以编写一个派生自 `TailWagger` 的类，并实现特征的方法：

```
class Dog extends TailWagger {
  // 实现 TailWagger 的方法
  override def startTail(): Unit = println("tail is wagging")
  override def stopTail(): Unit = println("tail is topped")
}
```

如果愿意，上面的代码也可以省略方法的返回类型声明和 `override` 关键字：

```
class Dog extends TailWagger {
  def startTail() = println("tail is wagging")
  def stopTail() = println("tail is topped")
}
```

注意这两个例子当中我们都使用了 `extends` 关键字用来表示 `Dog` 类继承了 `TailWagger` 特征：

```
class Dog extends TailWagger { ...
          -------
```

在 REPL 中实例化 `Dog` 类，可以尝试如下的代码：

```
scala> val d = new Dog
d: Dog = Dog@2c1ea7be

scala> d.startTail
tail is wagging

scala> d.stopTail
tail is topped
```

上面代码展示了如何在类当中继承一个 Scala 特征。

5.1.3 继承多个特征

Scala 通过提供特征的机制来构建模块化的代码。例如，可以将动物的属性分解成更小、更具有逻辑性的模块化单元：

```
trait Speaker {
  def speak(): String
}

trait TailWagger {
  def startTail(): Unit
  def stopTail(): Unit
}

trait Runner {
  def startRunning(): Unit
  def stopRunning(): Unit
}
```

在定义了这些更小的模块化特征后，就可以通过继承所有的特征来创建 `Dog` 类，然后实现所有必要的方法：

```
class Dog extends Speaker with TailWagger with Runner {

  // Speaker
  def speak(): String = "Woof!"

  // TailWagger
  def startTail(): Unit = println("tail is wagging")
  def stopTail(): Unit = println("tail is topped")

  // Runner
  def startRunning(): Unit = println("I'm running")
  def stopRunning(): Unit = println("Stopped running")
}
```

注意，在 Scala 中用 `extends` 和 `with` 来创建继承多个特征的类：

```
class Dog extends Speaker with TailWagger with Runner {
          -------         ----             ----
```

在继承多个特征的时候：

- 用 `extends` 继承第一个特征；
- 用 `with` 继承剩余的特征。

从上面的例子可以看到 Scala 的特征类似于 Java 中的接口，接下来我们介绍更多关于特征的细节。

5.2 像抽象类一样使用特征

在上一讲介绍了类似于 Java 中接口一样的方式来使用特征，但是 Scala 中的特征还有很多的功能，你可以添加非抽象的方法，就像 Java 中的抽象类一样。更准确地说，我们将特征的这种特性称之为**混入**（`mixins`）。

5.2.1 第一个例子

在下面的例子中， 一个名为 `Pet` 的特征拥有一个名为 `speak` 的非抽象方法，和一个名为 `comeToMaster` 的抽象方法：

```
trait Pet {
  def speak = println("Yo") // 非抽象方法
  def comeToMaster(): Unit        // 抽象方法
}
```

当一个类继承了 `Pet` 特征后，必须实现所有的抽象方法，因此下面的例子继承了 `Pet` 并定义了 `comeToMaster` 方法：

```
class Dog(name: String) extends Pet {
  def comeToMaster(): Unit = println("Woo-hoo, I'm coming!")
}
```

除非有特殊的需要而去覆写 `speak` 方法，不需要去实现特征中的非抽象方法，因此上面的例子的代码是正确的。可以用下面的代码来创建一个新的 `Dog` 实例：

```
val d = new Dog("Zeus")
```

接下来就可以调用 `speak` 和 `comeToMaster` 方法了。下面是在 REPL 中运行的结果：

```
scala> val d = new Dog("Zeus")
d: Dog = Dog@252d8df6

scala> d.speak
Yo

scala> d.comeToMaster()
Woo-hoo, I'm coming!
```

5.2.2 覆写一个非抽象方法

在类中可以覆写特征中的非抽象方法：

```
class Cat extends Pet {
  // 覆写 speak
  override def speak(): Unit = println("meow")
  def comeToMaster(): Unit = println("That's not gonna happen.")
}
```

在 REPL 中执行如下代码：

```
scala> val c = new Cat
c: Cat = Cat@603f4e3e

scala> c.speak
meow

scala> c.comeToMaster
That's not gonna happen.
```

5.2.3 混合多个特征的行为

在 Scala 中可以混合多个特征的行为到类当中，例如组合下面的特征：

```
trait Speak {
  def speak(): String // 抽象方法
}

trait TailWagger {
  def startTail(): Unit = println("tail is wagging")
  def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
  def startRunning(): Unit = println("I'm running")
  def stopRunning(): Unit = println("Stopped running")
}
```

首先创建一个继承上面所有特征，并实现了 `speak` 方法的 `Dog` 类：

```
class Dog extends Speak with TailWagger with Runner {
  def speak(): String = "Woof!"
}
```

然后定义一个 `Cat` 类：

```
class Cat extends Speak with TailWagger with Runner {
  def speak(): String = "Meow"
  override def startRunning(): Unit = println("Yeah ... I don't run")
  override def stopRunning(): Unit = println("No need to stop")
}
```

接下来在 REPL 中测试我们的代码，首先测试 `Dog` 类：

```
scala> val d = new Dog
d: Dog = Dog@76c587ce

scala> d.speak
res0: String = Woof!

scala> d.startRunning
I'm running

scala> d.stopRunning
Stopped running
```

然后测试 `Cat` 类：

```
scala> val c = new Cat
c: Cat = Cat@6c38f4de

scala> c.speak
res3: String = Meow

scala> c.startRunning
Yeah ... I don't run

scala> c.stopRunning
No need to stop
```

5.2.4 在运行时继承特征

Scala 中的特征最有趣的性质就是：可以在运行时动态继承那些**只有非抽象方法的特征**。例如给定如下特征：

```
trait TailWagger {
  def startTail(): Unit = println("tail is wagging")
  def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
  def startRunning(): Unit = println("I'm running")
  def stopRunning(): Unit = println("Stopped running")
}
```

接下来定义一个 `Dog` 类：

```
class Dog(name: String)
```

然后在创建 `Dog` 的实例时混入这些特征：

```
val d = new Dog("Fido") with TailWagger with Runner
                        --------------------------
```

下面是在 REPL 中的测试结果：

```
scala> val d = new Dog("Fido") with TailWagger with Runner
d: Dog with TailWagger with Runner = $anon$1@28b2e4d8

scala> d.startTail
tail is wagging

scala> d.startRunning
I'm running
```

注意，只有那些**只拥有非抽象方法**的特征才能在实例化的时候混入到实例中。

5.3 抽象类

Scala 中也有抽象类的概念，但是由于特征的功能强大，因此很少会使用抽象类。实际上，只有如下情况需要使用抽象类：

- 需要创建一个带有构造器参数的基类；
- Scala 代码要被 Java 代码调用。

5.3.1 特征不支持构造器参数

Scala 中的特征不支持构造器参数：

```
// 编译会出错
trait Animal(name: String)
```

因此，当基类需要构造器参数的时候，我们需要使用抽象类：

```
abstract class Animal(name: String)
```

但是要注意的是，一个类只能继承一个抽象类。

5.3.2 Java 代码调用 Scala 代码

由于 Java 对 Scala 特征一无所知（因为 Java 诞生的早），因此当你想要在 Java 代码中调用 Scala 代码，必须要用抽象类而不是特征。

5.3.3 抽象类的格式

抽象类的语法格式与特征近似，例如下面的例子中定义了一个名为 `Pet` 的抽象类，它的语法格式与上一讲的 `Pet` 特征非常近似：

```
abstract class Pet (name: String) {
  def speak(): Unit = println("Yo") // 非抽象方法
  def comeToMaster(): Unit          // 抽象方法
}
```

在定义了 `Pet` 抽象类后，定义 `Dog` 类：

```
class Dog(name: String) extends Pet(name) {
  override def speak() = println("Woof")
  def comeToMaster() = println("Here I come!")
}
```

下面是在 REPL 中运行的结果：

```
scala> val d = new Dog("Rover")
d: Dog = Dog@e0847a9

scala> d.speak
Woof

scala> d.comeToMaster
Here I come!
```

注意上面的代码中 `name` 是如何传递的。

由于上面的代码和 Java 极为类似，没有什么值得解释的细节，除了 `name` 是如何由 `Dog` 类的构造器传递给 `Pet` 的构造器的：

```
class Dog (name: String) extends Pet(name) { ...
```

注意 `Pet` 在构造器中定义了 `name` 参数：

```
abstract class Pet (name: String) { ...
```

下面的例子展示了如何在 `Dog` 类和 `Pet` 抽象类之间传递构造器参数：

```
abstract class Pet (name: String) {
  def speak: Unit = println(s"My name is $name")
}

class Dog (name: String) extends Pet(name)

val d = new Dog("Fido")
d.speak
```

6. Scala 中的集合类

Scala 的集合类比 Java 的集合类更加灵活，主要的 Scala 集合包括：

类	描述
`ArrayBuffer`	支持索引的可变序列
`List`	线性（链表），不可变序列
`Vector`	支持索引的不可变序列
`Map`	所有映射（键值对）的基类
`Set`	所有集合（无重复值）的基类

其中 `Map` 和 `Set` 有可变和不可变两个版本。

接下来我们介绍这些集合类。

> Scala 的集合中所有不可变（`immutable`）的集合都可以用于函数式编程。对于这些类，我们在应用函数式方法时，将会创建一个新的集合。

6.1 `ArrayBuffer` 类

首先介绍和 Java 中最接近的 `ArrayBuffer` 类，它是一个**可变**序列，这就意味着可以改变它的内容。`ArrayBuffer` 类的方法和 Java 的方法非常相似。

要使用 `ArrayBuffer` 类，首先需要导入它：

```
1 import scala.collection.mutable.ArrayBuffer
```

导入之后就可以创建 `ArrayBuffer` 的实例：

```
1 val ints = ArrayBuffer[Int]()
2 val names = ArrayBuffer[String]()
```

在创建了 `ArrayBuffer` 之后，就可以用多种方法添加元素：

```
1 val ints = ArrayBuffer[Int]()
2 ints += 1
3 ints += 2
```

在 REPL 中执行上述代码，可以观察到 `+=` 是如何工作的：

```
1 scala> ints += 1
2 res0: ints.type = ArrayBuffer(1)
3
4 scala> ints += 2
5 res1: ints.type = ArrayBuffer(1, 2)
```

上面展示了创建 `ArrayBuffer` 然后添加元素的一种方法。也可以在创建 `ArrayBuffer` 时直接初始化元素：

```
1 val nums = ArrayBuffer(1, 2, 3)
```

下面给出几种向 `ArrayBuffer` 中添加元素的方法：

```
// 添加一个元素
nums += 4

// 添加多个元素
nums += 5 += 6

// 从另一个集合中添加多个元素
nums ++= List(7, 8, 9)
```

如果要删除 `ArrayBuffer` 当中的元素可以使用 `-=` 和 `--=` 方法：

```
// 删除一个元素
nums -= 9

// 删除多个元素
nums -= 7 -= 8

// 删除另一个列表提供的多个元素
nums --= Array(5, 6)
```

在 REPL 中运行上面的例子可得以下的结果：

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> val nums = ArrayBuffer(1, 2, 3)
nums: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

scala> nums += 4
res0: nums.type = ArrayBuffer(1, 2, 3, 4)

scala> nums += 5 += 6
res1: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

scala> nums ++= List(7, 8, 9)
res2: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> nums -= 9
res3: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)

scala> nums -= 8 -= 7
res4: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

scala> nums --= Array(5, 6)
res5: nums.type = ArrayBuffer(1, 2, 3, 4)
```

`ArrayBuffer` 还提供了其他的方法来修改其中的内容：

```
scala> val a = ArrayBuffer(1, 2, 3)
a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

scala> a.append(4)
res0: a.type = ArrayBuffer(1, 2, 3, 4)

scala> a.appendAll(Seq(7, 8))
res1: a.type = ArrayBuffer(1, 2, 3, 4, 7, 8)

scala> a.clear()
```

```
scala> val a = ArrayBuffer(9, 10)
a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(9, 10)

scala> a.insert(0, 8)
scala> print(a)
ArrayBuffer(8, 9, 10)

scala> a.insertAll(0, Vector(4, 5, 6, 7))
scala> print(a)
ArrayBuffer(4, 5, 6, 7, 8, 9, 10)

scala> a.prepend(3)
res13: a.type = ArrayBuffer(3, 4, 5, 6, 7, 8, 9, 10)

scala> a.prependAll(Array(0, 1, 2))
res14: a.type = ArrayBuffer(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
scala> val a = ArrayBuffer.range('a', 'h')
a: scala.collection.mutable.ArrayBuffer[Char] = ArrayBuffer(a, b, c, d, e,
f, g)

scala> a.remove(0)
res15: Char = a

scala> println(a)
ArrayBuffer(b, c, d, e, f, g)

scala> a.remove(2, 3)

scala> println(a)
ArrayBuffer(b, c, g)
```

```
scala> val a = ArrayBuffer.range('a', 'h')
a: scala.collection.mutable.ArrayBuffer[Char] = ArrayBuffer(a, b, c, d, e,
f, g)

scala> a.trimStart(2)
scala> println(a)
ArrayBuffer(c, d, e, f, g)

scala> a.trimEnd(2)
scala> println(a)
ArrayBuffer(c, d, e)
```

6.2 `List` 类

`List` 类是一个线性、**不可变**的序列，这就意味着它是一个不能修改的链表。当需要对 `List` 进行添加和删除操作时，需要根据现有的 `List` 创建一个新的 `List` 。

6.2.1 创建 `List`

可以用如下的方法创建并初始化 `List` ：

```
val ints = List(1, 2, 3)
val names = List("Joel", "Chris", "Ed")
```

尽管没有必要，但是依然可以指定 `List` 中元素的数据类型：

```
val ints = List[Int](1, 2, 3)
val names = List[String]("Joel", "Chris", "Ed")
```

6.2.2 向 `List` 中添加元素

因为 `List` 是不可变的，因此无法直接向 `List` 中添加元素。如果要添加元素，需要通过在现有的 `List` 的头部或者尾部拼接上想要插入或追加的元素，来创建新的 `List` 。例如，给定如下的 `List` ：

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

然后在头部插入元素（ `prepend` ）：

```
val b = 0 +: a
```

以及：

```
val b = List(-1, 0) ++: a
```

在 REPL 中的运行结果如下：

```
scala> val b = 0 +: a
b: List[Int] = List(0, 1, 2, 3)

scala> val b = List(-1, 0) ++: a
b: List[Int] = List(-1, 0, 1, 2, 3)
```

由于 `List` 是一个单链表，因此向 `List` 追加元素会导致性能非常低，尤其是在处理庞大序列的时候，这一点尤为明显。

> 如果要对一个不可变序列既要在头部插入，又要在尾部追加，那么应该使用 `Vector` 类。

由于 `List` 是一个链表，因此不要试图用**索引**来访问一个巨型 `List` 中的元素。例如，有一个拥有一百万个元素的 `List` ，然后用 `myList(999999)` 这样的方法来访问元素，这回非常的慢。如果想用下标访问元素，应该使用 `Vector` 或 `ArrayBuffer` 。

6.2.3 如何记忆方法的名字

如今，尽管IDE 可以极大的帮我们从**记忆方法名字的负担**中解放出来，但是仍需要记住形似 `+:` 、`:++` 这样的方法名。有一个方法可以帮我们记住这些方法名：`:` 的一侧是序列，因此当我们使用 `+:` 时就意味着列表需要放在右侧：

```
0 +: a
```

类似的，当使用 `:+` 的时候，需要将列表放在左侧：

```
a :+ 4
```

有很多技巧性的方法来记住这些方法名，但是这可能是最简单的一种。

这些方法名具有一致性，这就意味着在 `Seq` 和 `Vector` 之类的不可变序列类中都支持这些方法名。

6.2.4 如何遍历 `List`

在本书的前面展示了如何遍历 `List` 中的元素，因此没有必要再次介绍它的语法。给定如下的列表：

```
val names = List("Joel", "Chris", "Ed")
```

然后就可以用如下的方法打印每个字符串：

```
for (name <- names) println(name)
```

在 REPL 中运行的结果如下：

```
scala> for (name <- names) println(name)
Joel
Chris
Ed
```

这样的方法可以用于遍历所有的序列类，包括 `ArrayBuffer` 、 `List` 、 `Seq` 、 `Vector` 等等。

6.2.5 历史沿革

如果你对编程语言发展的历史感兴趣， Scala 的 `List` 类与 Lisp 编程语言中的 `List` 类非常的近似。例如，下面创建 `List` 的方式：

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

还可以用下面的方法创建：

```
val list = 1 :: 2 :: 3 :: Nil
```

下面是在 REPL 中的运行结果：

```
scala> val list = 1 :: 2 :: 3 :: Nil
list: List[Int] = List(1, 2, 3)
```

这种方式可以工作的原因是： `List` 是一个以 `Nil` 结尾的单链表。

6.3 `Vector` 类

`Vector` 类是一个可以用**索引**访问元素的**不可变**序列。可以用索引访问意味着通过索引访问 `Vector` 对象的元素速度很快。

除了可以用索引访问元素之外， `Vector` 和 `List` 没有什么差别，两个类的工作方式是一致的，下面来看一些例子。

可以用下面的方式来创建一个 `Vector` ：

```
val nums = Vector(1, 2, 3, 4, 5)

val strings = Vector("one", "two")

val peeps = Vector(
  Person("Bert"),
  Person("Ernie"),
  Person("Grover")
)
```

由于 `Vector` 是不可变的，所以无法向其中添加或者删除元素，因此需要取出现有的 `Vector` 的元素，然后在前面或后面插入元素，来来生成一个新的 `Vector` ：

```
val a = Vector(1, 2, 3)
```

然后可以用如下方式来追加元素：

```
val b = a :+ 4
```

或者：

```
val b = a ++ Vector(4, 5)
```

> 注意在上面的例子中，既可以用 `++` 方法，也可以用 `++:` 方法，还可以用 `:++` 方法。

在 REPL 中运行可得以下结果：

```
scala> val a = Vector(1, 2, 3)
a: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> val b = a :+ 4
b: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4)

scala> val b = a ++ Vector(4, 5)
b: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

还可以用下面的方法在前面添加元素：

```
val b = 0 +: a
```

或者：

```
val b = Vector(-1, 0) ++: a
```

在 REPL 中运行上述例子可以得到如下结果：

```
scala> val b = 0 +: a
b: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2, 3)

scala> val b = Vector(-1, 0) ++: a
b: scala.collection.immutable.Vector[Int] = Vector(-1, 0, 1, 2, 3)
```

由于 `Vector` 不是 `List` 这种链表，因此对于 `Vector` 在前后添加元素的执行速度是相似的。

遍历 `Vector` 的方法和遍历 `ArrayBuffer` 以及 `List` 的方式是一样的：

```
scala> val names = Vector("Joel", "Chris", "Ed")
names: scala.collection.immutable.Vector[String] = Vector(Joel, Chris, Ed)

scala> for (name <- names) println(name)
Joel
Chris
Ed
```

6.4 `Map` 类

`Map` 是一个包含了键值对（`key-value pairs`）的可迭代序列，例如：

```
val states = Map(
  "AK" -> "Alaska",
  "IL" -> "Illinois",
  "KY" -> "Kentucky"
)
```

在 Scala 中有两个 `Map` 类，一个是可变的，而另一个是不可变的，下面我们讨论如何使用可变的 `Map` 类。

6.4.1 创建可变 `Map`

要使用可变的 `Map` 类，首先需要导入：

```
import import scala.collection.mutable.Map
```

然后就可以创建可变 `Map`：

```
val states = collection.mutable.Map("AK" -> "Alaska")
```

> 注意在 Scala 中，类的完整名字和 Java 类似，由包名和类名构成，因此如果要创建不可变 `Map` 可以用下面的方式：
>
> ```
> val states = collection.immutable.Map("AK" -> "Alaska")
> ```

6.4.2 添加元素

如果要向 `Map` 中添加一个键值对，可以用 `+=`：

```
states += ("AL" -> "Alabama")
```

也可以用 `+=` 添加多个键值对：

```
states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
```

> 在最新版本的 Scala 中（ `2.13.1` ）中用 `+=` 添加多个键值对的方法已经被废弃，建议使用下面的方法添加多个键值对。

添加另外一个 `Map` 的所有键值对可以用 `++=`：

```
states ++= Map("CA" -> "California", "CO" -> "Colorado")
```

在 REPL 中运行可得如下结果：

```
scala> states += ("AL" -> "Alabama")
res0: states.type = HashMap(AK -> Alaska, AL -> Alabama)

scala> states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
res1: states.type = HashMap(AR -> Arkansas, AZ -> Arizona, AK -> Alaska, AL
-> Alabama)

scala> states ++= Map("CA" -> "California", "CO" -> "Colorado")
res2: states.type = HashMap(AR -> Arkansas, AZ -> Arizona, AK -> Alaska, AL
-> Alabama, CO -> Colorado, CA -> California)
```

6.4.3 删除元素

可以用 `-=` 和 `--=` 来删除 `Map` 中指定的键值：

```
states -= "AR"
states -= ("AL", "AZ")
states --= List("AL", "AZ")
```

在 REPL 中执行可得如下结果：

```
scala> states -= "AR"
res6: states.type = HashMap(AZ -> Arizona, AK -> Alaska, AL -> Alabama, CO -
> Colorado, CA -> California)

scala> states -= ("AL", "AZ")
res7: states.type = HashMap(AK -> Alaska, CO -> Colorado, CA -> California)

scala> states --= List("AL", "AZ")
res8: states.type = HashMap(AK -> Alaska, CO -> Colorado, CA -> California)
```

6.4.4 更新元素

可以对 `Map` 中的元素重新进行赋值：

```
states("AK") = "Alaska, A Really Bit State"
```

执行上述代码后，在 REPL 中可以观察当前 `Map` 的状态：

```
scala> states("AK") = "Alaska, A Really Bit State"

scala> states
res10: scala.collection.mutable.Map[String,String] = HashMap(AK -> Alaska, A Really Bit State, CO -> Colorado, CA -> California)
```

6.4.5 遍历 `Map`

有多种方式可以用来遍历 `Map` ，下面介绍其中的两种最常用的方式。假设有如下 `Map` ：

```
val ratings = Map(
  "Lady in the Water" -> 3.0,
  "Snakes on a Plane" -> 4.0,
  "You, Me and Dupree" -> 3.5
)
```

最好的方式就是用 `for` 循环来遍历 `Map` 中的元素：

```
for ((k, v) <- ratings) println(s"Key: $k, value: $v")
```

在 `Map` 的 `foreach` 方法中使用 `match` 表达式，也具有很好的可读性：

```
ratings.foreach {
  case (movie, rating) => println(s"key: $movie, value: $ratings")
}
```

6.5 `SET` 类

Scala 中的 `Set` 类是一个可迭代的没有重复元素的集合。

与 `Map` 相同， `Set` 也有**可变**和**不可变**两种版本。本节将讨论如何使用**可变**版本。

6.5.1 添加元素

要使用可变版本的 `Set` 首先需要导入或者使用全名：

```
val set = scala.collection.mutable.Set[Int]()
```

可以使用 `+=` 、 `++=` 和 `add` 方法添加元素：

```
set += 1
set += 2 += 3
set ++= Vector(4, 5)
```

在 REPL 中运行可得下面的结果：

```
scala> val set = scala.collection.mutable.Set[Int]()
set: scala.collection.mutable.Set[Int] = HashSet()

scala> set += 1
res0: set.type = HashSet(1)

scala> set += 2 += 3
res1: set.type = HashSet(1, 2, 3)

scala> set ++= Vector(4, 5)
res2: set.type = HashSet(1, 2, 3, 4, 5)
```

如果试图向 `Set` 中添加一个已经存在的值，会被直接忽略：

```
scala> set += 2
res3: set.type = HashSet(1, 2, 3, 4, 5)
```

`Set` 可以用 `add` 方法添加元素，如果成功添加则返回 `true` ，如果未能添加则返回 `false` ：

```
scala> set.add(6)
res4: Boolean = true

scala> set.add(5)
res5: Boolean = false
```

6.5.2 删除元素

可以用 `-=` 和 `--=` 方法从 `Set` 中删除元素：

```
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4, 5)

// 删除一个元素
scala> set -= 1
res0: set.type = HashSet(2, 3, 4, 5)

// 删除多个元素
scala> set -= (2, 3)
res1: set.type = HashSet(4, 5)

// 删除另一个序列所包含的所有元素
scala> set --= Array(4, 5)
res2: set.type = HashSet()
```

除此之外， `Scala` 还支持 `clear` 和 `remove` 方法：

```
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4, 5)

// 清除集合中的所有元素
scala> set.clear()

scala> set
res1: scala.collection.mutable.Set[Int] = HashSet()
```

```
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4, 5)

// 删除集合中的元素
scala> set.remove(2)
res2: Boolean = true

scala> set
res3: scala.collection.mutable.Set[Int] = HashSet(1, 3, 4, 5)

scala> set.remove(40)
res4: Boolean = false
```

除此之外，Scala 还提供了其他的 `Set` 类，例如 `SortedSet`， `LinkedHashSet` 等，具体细节可以阅读 Scala 的文档。

6.6 匿名函数

在本书前面给出过下面创建整数列表的例子：

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

要创建一个更大的整数列表，还可以使用 `List` 类的 `range` 方法：

```
val ints = List.range(1, 10)
```

这段代码创建了一个包含 `1` 到 `10` 的整数列表，在 REPL 中运行可以看到如下结果：

```
scala> val ints = List.range(1, 10)
ints: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

本节将用创建列表的例子来介绍 Scala 中函数式编程的一个特性——匿名函数（ `Anonymous Functions` ）。通过学习匿名函数，可以帮助深入理解 Scala 集合类的通用方法。

6.6.1 在 `map()` 方法中使用匿名函数

匿名函数有点像一个微型的函数。例如，我们建立如下列表：

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

可以用下面的方法，建立一个新的列表，其中每个元素都是 `ints` 列表的两倍：

```
val doubledInts = ints.map(_ * 2)
```

下面是在 REPL 中运行的结果：

```
scala> val doubledInts = ints.map(_ * 2)
doubledInts: List[Int] = List(2, 4, 6)
```

如上例所示，`doubledInts` 是一个列表：`List(2, 4, 6)`，在上面的例子中，下面的代码是一个匿名函数：

```
_ * 2
```

这是"将一个元素乘以2"的非常简短的表示。

当你习惯 Scala 之后，这是最常用的匿名函数的写法。但是如果愿意，也可以是写成比较长的形式，原有的代码：

```
val doubledInts = ints.map(_ * 2)
```

还可以写成如下形式：

```
val doubledInts = ints.map((i: Int) => i * 2)
val doubledInts = ints.map(i => i * 2)
```

上面的三行代码的功能是完全一样的，都是用 `ints` 中所有元素的两倍的值，来构成一个新的 `doubledInts` 列表。

> 在 Scala 中的 `_` 字符是一种通配符（`Wildcard Character`），可以在多种不同的位置使用。在本例中，`_` 用来表示 `ints` 中的任意一个元素。

在继续下面的内容之前，首先看一下这个例子等价的 Java 代码：

```
List<Integer> ints = new ArrayList<>(Arrays.asList(1, 2, 3));

// map方法的处理过程
List<Integer> doubledInts = ints.stream()
                                .map(i -> i * 2)
                                .collect(Collectors.toList());
```

上面的例子与下面的 Scala 代码也是等价的：

```
val doubledInts = for (i <- ints) yield i * 2
```

6.6.2 在 `filter()` 方法中使用匿名函数

接下来介绍在 `List` 类的 `filter()` 方法中使用匿名函数的例子。给定如下 `List`：

```
val ints = List.range(1, 10)
```

下面展示了如何用 `ints` 中大于 `5` 的元素构造一个新的列表：

```
val x = ints.filter(_ > 5)
```

下面的例子是用 `ints` 中小于 `5` 的元素构造一个新的列表：

```
val x = ints.filter(_ < 5)
```

接下来的例子稍微复杂一点，用 `ints` 中的偶数来创建一个新的列表：

```
val x = ints.filter(_ % 2 == 0)
```

如果这个例子看上去令人费解，不要忘记它也可以写成如下的形式：

```
val x = ints.filter((i: Int) => i % 2 == 0)
val x = ints.filter(i => i % 2 == 0)
```

下面是上面的例子在 REPL 中运行的结果：

```
scala> val x = ints.filter(_ > 5)
x: List[Int] = List(6, 7, 8, 9)

scala> val x = ints.filter(_ < 5)
x: List[Int] = List(1, 2, 3, 4)

scala> val x = ints.filter(_ % 2 == 0)
x: List[Int] = List(2, 4, 6, 8)
```

匿名函数小结：

- 匿名函数就是一段代码片段；
- 可以在 `List` 类的 `map` 、 `filter` 等方法中使用匿名函数；
- 通过这些小的代码片段和功能强大的 `map` 、 `filter` 等方法，可以用非常简短的代码实现非常多的功能。

Scala 的集合类支持许多像 `map` 、 `filter` 这样强大的方法，利用这些方法可以编写富于表达的代码。

6.6.3 深入探讨

在上面的例子中，一个整数列表，更确切的说一个 `List[Int]` 调用 `map` 方法，意味着 `map` 需要接收一个函数，用这个函数去变换列表中的所有值。由于 `map` 期望接收一个函数（或者方法）以便将一个 `Int` 值变换成另外一个 `Int` 值，因此也可以用下面的方法：

```
val ints = List.range(1, 2, 3)
def double(i: Int): Int = i * 2 // 一个将输入值加倍的方法
val doubledInts = ints.map(double)
```

这段代码的最后两行就等价于：

```
val doubledInts = ints.map(_ * 2)
```

> 传递给 `map` 的函数（或方法）也可以将整数转换为其他类型，例如：
>
> ```
> val hi(i: Int): String = "hi" + i
> val hiInts = ints.map(hi)
> ```

与 `map` 类似，当调用 `List[Int]` 的 `filter` 方法时， `filter` 方法需要一个接受一个 `Int` 类型参数并返回一个 `Boolean` 值的函数或方法作为参数。因此，也可以用如下的方式来定义方法：

```
def lessThanFive(i: Int): Boolean = if (i < 5) true else false
```

也可以写成如下更简洁的版本：

```
def lessThanFive(i: Int): Boolean = (i < 5)
```

下面的代码：

```
def lessThanFive(i: Int): Boolean = (i < 5)
val y = ints.filter(lessThanFive)
```

等价于：

```
val y = ints.filter(_ < 5)
```

6.7 通用序列方法

Scala 中的集合类包含了一组内建的方法，可以使得在处理集合时无需使用 `for` 循环。由于 Scala 提供了相当多的方法，本节只介绍其中最常用的方法：

- `map`
- `fileter`
- `foreach`
- `head`
- `tail`
- `take` , `takeWhile`
- `drop` , `dropWhile`
- `find`
- `reduce` , `fold`

包括 `Array` 、 `ArrayBufrer` 、 `List` 、 `Vector` 等集合"序列"类都支持这些方法。本节将以 `List` 作为例子。

> **注意：**所有这些方法都不会改变集合的内容，它们都以一种函数式风格执行，因此他们返回一个存储了修改后结果的新集合。

在下面的例子中将使用下面两个列表：

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

下面是在 REPL 中运行的结果：

```
scala> val nums = (1 to 10).toList
nums: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val names = List("joel", "ed", "chris", "maurice")
names: List[String] = List(joel, ed, chris, maurice)
```

6.7.1 `map`

`map` 方法遍历列表中的每一个元素，每次以一个元素为算法的输入，然后将每次算法运算的结果构成一个新的列表。

下面是调用 `nums` 列表的 `map` 方法的示例：

```
scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

根据上一讲的内容，使用匿名函数的代码也可以写成：

```
scala> val doubles = nums.map(i => i * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

在本节中，我们将使用更为简洁的第一种形式。

下面是一些使用 `map` 方法的例子：

```
scala> val capNames = names.map(_.capitalize)
capNames: List[String] = List(Joel, Ed, Chris, Maurice)

scala> val lessThanFive = nums.map(_ < 5)
lessThanFive: List[Boolean] = List(true, true, true, true, false, false,
false, false, false, false)
```

第二个例子展示了一种将 `List[Int]` 变换为 `List[Boolean]` 的方法。

6.7.2 `filter`

`filter` 方法对现有列表进行过滤，从而生成一个新的列表：

```
scala> val lessThanFive = nums.filter(_ < 5)
lessThanFive: List[Int] = List(1, 2, 3, 4)

scala> val evens = nums.filter(_ % 2 == 0)
evens: List[Int] = List(2, 4, 6, 8, 10)

scala> val shortNames = names.filter(_.length <= 4)
shortNames: List[String] = List(joel, ed)
```

6.7.3 `foreach`

`foreach` 用于循环访问集合中的每一个元素，正如之前介绍的那样，`foreach` 使用打印信息这样的**副作用**（`side-effects`）。例如：

```
scala> names.foreach(println)
joel
ed
chris
maurice
```

在 Scala 中可以链式调用方法，例如打印 `nums` 中小于 `4` 个元素：

```
nums.filter(_ < 4).foreach(println)
```

执行的结果如下：

```
scala> nums.filter(_ < 4).foreach(println)
1
2
3
```

6.7.4 `head`

`head` 方法来自于 Lisp 和函数式编程语言。可以用于获得列表的第一个元素（头元素）：

```
scala> nums.head
res2: Int = 1

scala> names.head
res3: String = joel
```

因为 `String` 是字符序列，因此可以将其看作是一个列表，因此这些方法也适用于 `String`：

```
scala> "foo".head
res5: Char = f

scala> "bar".head
res6: Char = b
```

`head` 方法的用途广泛，但是当集合为空时会抛出异常：

```
scala> val emptyList = List[Int]()
emptyList: List[Int] = List()

scala> emptyList.head
java.util.NoSuchElementException: head of empty list
```

6.7.5 `tail`

`tail` 方法同样来自 Lisp 和函数式编程语言。用于获得除了第一个元素之外的所有元素。例如：

```
scala> nums.tail
res8: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> names.tail
res9: List[String] = List(ed, chris, maurice)
```

与 `head` 一样， `tail` 也可以用于字符串：

```
scala> "foo".tail
res10: String = oo

scala> "bar".tail
res11: String = ar
```

注意，如果在一个空集合上调用 `tail` 方法会抛出异常：

```
scala> emptyList.tail
java.lang.UnsupportedOperationException: tail of empty list
```

6.7.6 `take` , `takeWhile`

`take` 和 `takeWhile` 方法提供了从列表中取出元素来构造新列表的方法。`take` 的使用方法如下：

```
scala> nums.take(1)
res14: List[Int] = List(1)

scala> nums.take(2)
res15: List[Int] = List(1, 2)

scala> names.take(1)
res16: List[String] = List(joel)

scala> names.take(2)
res17: List[String] = List(joel, ed)
```

下面是 `takeWhile` 的使用方法：

```
scala> nums.takeWhile(_ < 5)
res18: List[Int] = List(1, 2, 3, 4)

scala> names.takeWhile(_.length < 5)
res19: List[String] = List(joel, ed)
```

> 注意 `takeWhile` 与 `filter` 的区别，例如：
>
> ```
> scala> List(1, 2, 5, 4, 6, 7, 8).takeWhile(_ < 5)
> res27: List[Int] = List(1, 2)
>
> scala> List(1, 2, 5, 4, 6, 7, 8).filter(_ < 5)
> res28: List[Int] = List(1, 2, 4)
> ```

6.7.7 `drop` , `dropWhile`

`drop` 与 `dropWhile` 方法与 `take` 和 `takeWhile` 方法的功能恰好相反。`drop` 的使用方法：

```
scala> nums.drop(1)
res20: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> nums.drop(5)
res21: List[Int] = List(6, 7, 8, 9, 10)

scala> names.drop(1)
res22: List[String] = List(ed, chris, maurice)

scala> names.drop(2)
res23: List[String] = List(chris, maurice)
```

下面是 `dropwhile` 的使用方法：

```
scala> nums.dropWhile(_ < 5)
res29: List[Int] = List(5, 6, 7, 8, 9, 10)

scala> names.dropWhile(_ != "chris")
res30: List[String] = List(chris, maurice)
```

6.7.8 reduce

在大数据领域有个非常著名的术语：`map-reduce`，其中 `reduce` 部分就是参考的 `reduce` 方法。`reduce` 方法接受一个（匿名）函数作为参数。

> 举个 `map-reduce` 的简单的例子：现在一个学习小组的同学要统计小说《天龙八部》中乔峰的名字出现了多少次。假设学习小组有 `6` 名同学，首先将《天龙八部》（我上学的时候是五卷）分给 `5` 个同学，每个人分别去数自己手头上的这一卷乔峰的名字出现了多少次，这个过程就是 `map`。然后 `5` 个同学把自己计算的结果交给第 `6` 名同学，这位同学把所有人统计的数字求和的过程就叫做 `reduce`。因此 `reduce` 通常用于求和、计数等需要整个列表的数据都参与其中的计算。

为了解释 `reduce` 方法，首先创建一个方法帮助我们来理解 `reduce`。例如，定义一个用于对两个整数进行求和的方法 `add()`，该方法可以给出比较好的调试信息：

```
def add(x: Int, y: Int): Int = {
  val theSum = x + y
  println(s"received $x and $y, their sum is $theSum")
  theSum
}
```

给定如下列表：

```
val a = List(1, 2, 3, 4)
```

下面的输出展示了将 `add` 方法传入到 `reduce` 的输入：

```
scala> a.reduce(add)
received 1 and 2, their sum is 3
received 3 and 3, their sum is 6
received 6 and 4, their sum is 10
res0: Int = 10
```

如结果所示，`reduce` 方法用 `add` 函数将列表 `a` 退化成一个值，在这个例子中，计算了列表所有元素的和。

当你熟悉 `reduce` 的使用之后，就可以编写下面的求和算法：

```
scala> a.reduce(_ + _)
res1: Int = 10
```

类似的，如果要计算所有元素的乘积：

```
scala> a.reduce(_ * _)
res2: Int = 24
```

> 正如 `reduce` 方法的名字暗示的那样，该方法用于将一个集合（**坍塌/退化/降维**）成一个数值。

6.8 通用映射方法

本节将介绍一些常用的 `Map` 方法。本节首先使用不可变的 `Map`，然后给出可变 `Map` 的例子。

6.8.1 不可变映射的示例

给定如下的不可变 `Map`：

```
val m = Map(
  1 -> "a",
  2 -> "b",
  3 -> "c",
  4 -> "d"
)
```

下面是一些 `Map` 方法的使用示例：

```
// 遍历映射中的所有键值对
scala> for ((k, v) <- m) printf("key: %s, value: %s\n", k, v)
key: 1, value: a
key: 2, value: b
key: 3, value: c
key: 4, value: d

// 从映射中获得所有的键
scala> val keys = m.keys
keys: Iterable[Int] = Set(1, 2, 3, 4)

// 从映射中获得所有的值
scala> val values = m.values
values: Iterable[String] = Iterable(a, b, c, d)

// 检测映射中是否包含某个键
scala> val contains3 = m.contains(3)
contains3: Boolean = true

// 修改映射中的值
scala> val ucMap = m.transform((k, v) => v.toUpperCase)
ucMap: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B, 3 -
> C, 4 -> D)

// 按照键对映射进行过滤
scala> val twoAndThree = m.view.filterKeys(Set(2, 3)).toMap
twoAndThree: scala.collection.immutable.Map[Int,String] = Map(2 -> b, 3 ->
c)

// 获得映射的前两个键值对
scala> val firstTwoElements = m.take(2)
firstTwoElements: scala.collection.immutable.Map[Int,String] = Map(1 -> a,
2 -> b)
```

> 注意最后一个例子只能用于有序映射。

6.8.2 可变映射示例

给定如下的可变映射：

```
val states = scala.collection.mutable.Map (
  "AL" -> "Alabama",
  "AK" -> "Alaska"
)
```

下面是一些可变 `Map` 的例子：

```
// 用 += 添加元素
states += ("AZ" -> "Arizona")
states ++= List("CO" -> "Colorado", "KY" -> "Kentucky")

// 用 -= 删除元素
states -= "KY"
states -= ("AZ", "CO")

// 更新映射中的值
states("AK") = "Alaska, The Big State"

// 根据匿名函数的值对映射进行过滤
states.filterInPlace((k, v) => k == "AK")
```

7. 杂项

本章将介绍元组（`Tuples`）和一个用 Scala 编写的面向对象编程范式的例子。

7.1 元组

元组是一个非常简洁的类，提供了在同一个容器中存储异构元素的功能。例如，假设有如下的类：

```
class Person(var name: String)
```

有时需要创建一个临时的类用来存储一些信息，例如：

```
class someThings(i: Int, s: String, p: Person)
```

可以用元组来编写更简洁的代码：

```
val t = (3, "Three", new Person("Al"))
```

如上例所示，只要把元素放到小括号之间，就可以创建一个元组。 Scala 的元组可以包含 `2` 到 `22` 个元素。当需要将不同的元素组合在一起时，使用元组就可以避免去构造这类临时需要的类，以便提高代码的可读性。

> 之所以 Scala 支持 `2` 到 `22` 个元素，这是因为在 Scala 2.x 中构造了 `Tuple2` 、 `Tuple3` ， ..... ， `Tuple22` 类，当然在使用时并不需要知道这些细节，在 Scala 3 中将会对其进行改进。

7.1.1 元组的特性

下面的例子中，创建了一个拥有两个元素的元组：

```
scala> val d = ("Maggie", 30)
d: (String, Int) = (Maggie,30)
```

该元组包含了两个元素，下例展示了如何定义拥有三个元素的元组：

```
scala> case class Person(name: String)
defined class Person

scala> val t = (3, "Three", new Person("David"))
t: (Int, String, Person) = (3,Three,Person(David))
```

有多种方式可以访问元组中的元素。第一种方法就是按照元素的编号进行访问，对应元素的字段名就是在编号之前添加一个下划线 `_` 。

> 注意：编号从 `1` 开始。

```
scala> t._1
res0: Int = 3

scala> t._2
res1: String = Three

scala> t._3
res2: Person = Person(David)
```

开始以用多重赋值的方式访问元组的所有元素：

```
scala> val(x, y, z) = (3, "Three", new Person("David"))
x: Int = 3
y: String = Three
z: Person = Person(David)
```

> 从技术上讲，这种方法实际上是一种模式匹配。

7.1.2 以元组为返回值

另一种适合使用元组的情况是：方法需要返回多个返回值。例如，下面的方法返回的是一个元组：

```
def getStockInfo= {
  // other code here ...
  ("NFLX", 100.00, 101.00) // 这是一个 Tuple3 类
}
```

然后可以用下面的方法来调用方法，并取得元组中的元素：

```
val (symbol, currentPrice, bidPrice) = getStockInfo
```

在 REPL 中运行结果如下：

```
scala> val (symbol, currentPrice, bidPrice) = getStockInfo
symbol: String = NFLX
currentPrice: Double = 100.0
bidPrice: Double = 101.0
```

这种情况下，使用元组可以避免为了返回多个值而创建一个临时的类。

> 注意：从技术的角度来说，Scala 2.x 的元组不是一个集合类，只是一个便于使用的小容器，因此元素不支持 `map` 、 `filter` 等方法。

7.2 面向对象示例

本节介绍一个用 Scala 编写的符合面向对象编程范式的例子——为披萨店构造一个订单输入系统。

首先依据本书之前介绍的内容，创建相关的枚举：

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

Scala 的优点在于，即使我们尚未讨论 `sealed traits` 和 `case object` ，读者可能也能看懂这段代码在做什么工作。

7.2.1 定义类

在定义了枚举之后，就可以开始创建订单输入系统所需要的一些有关披萨的类。首先定义 `Pizza` 类：

```
import scala.collection.mutable.ArrayBuffer

class Pizza (
            var crustSize: CrustSize,
            var crustType: CrustType,
            var toppings: ArrayBuffer[Topping]
            )
```

接下来定义 `Order` 类，该类包含一个用于存储披萨的列表（客户可能会订多个披萨）和一个顾客 `Customer` 字段：

```
class Order (
            var pizzas: ArrayBuffer[Pizza],
            var customer: Customer
            )
```

其中 `Customer` 类的定义如下：

```
class Customer (
               var name: String,
               var phone: String,
               var address: Address
               )
```

其中 `Addresss` 类的定义如下：

```
class Address (
              var street1: String,
              var street2: String,
              var city: String,
              var state: String,
              var zipCode: String
              )
```

到目前为止，我们创建的这些类只包括了数据结构，类似 C 语言中的结构体，接下来为这些类添加行为。

7.2.2 为 `Pizza` 类添加行为

一个具有面向对象编程范式风格的 `Pizza` 类，需要包含一些方法。例如添加或删除配料以及调整饼坯的尺寸和类型。下面的 `Pizza` 类添加了一些用于处理配料和饼坯的行为：

```
class Pizza (
            var crustSize: CrustSize,
            var crustType: CrustType,
            var toppings: ArrayBuffer[Topping]
            ) {
  def addTopping(t: Topping): Unit = toppings += t
  def removeTopping(t: Topping): Unit = toppings -= t
  def removeAllToppings(): Unit = toppings.clear()
}
```

也许我们还需要能够自动计算披萨的价格，因此可以添加如下的方法：

```
def getPrice(
            toppingsPrices: Map[Topping, Int],
            crustSizePrices: Map[CrustSize, Int],
            crustTypePrices: Map[CrustType, Int]
            ): Int = ???
```

注意该方法是一个合法的方法，`???` 语法经常被用作教学工具，通常将其作为一个“打草稿”的工具。`???` 意味着：“这是我的方法的签名，但是我还不想写下方法体。”而且这样编写的代码是可以通过编译的。

> 注意：如果调用该方法，会产生 `NotImplementedError` 的错误，顾名思义。

7.2.3 为 `Order` 类添加行为

需要为 `Order` 类添加以下行为：

- 添加和删除披萨
- 更新客户信息
- 获得订单价格

```
class Order(
             var pizzas: ArrayBuffer[Pizza],
             var customer: Customer
           ) {

  def addPizza(p: Pizza): Unit = pizzas += p
  def removePizza(p: Pizza): Unit = pizzas -= p

  // 待实现方法
  def getBasePrice(): Int = ???
  def getTaxes(): Int = ???
  def getTotalPrice(): Int = ???
}
```

在本例中，我们并不关心订单价格的计算。

7.2.3 测试类

可以通过编写一个测试类来测试这些类，下面的测试类需要在 `Order` 中实现 `printOrder` 方法，在 `Pizza` 类中实现 `toString` 方法，然后就可以运行下面的测试类：

```
import scala.collection.mutable.ArrayBuffer

object MainDriver extends App {

    val p1 = new Pizza (
        MediumCrustSize,
        ThinCrustType,
        ArrayBuffer(Cheese)
    )

    val p2 = new Pizza (
        LargeCrustSize,
        ThinCrustType,
        ArrayBuffer(Cheese, Pepperoni, Sausage)
    )

    val address = new Address (
        "123 Main Street",
        "Apt. 1",
        "Talkeetna",
        "Alaska",
        "99676"
    )

    val customer = new Customer (
```

```
        "Alvin Alexander",
        "907-555-1212",
        address
    )

    val o = new Order(
        ArrayBuffer(p1, p2),
        customer
    )

    o.addPizza(
        new Pizza (
            SmallCrustSize,
            ThinCrustType,
            ArrayBuffer(Cheese, Mushrooms)
        )
    )

    // print the order
    o.printOrder

}
```

本节的完整代码参照：本书的Git仓库

8. 函数式编程

在 Scala 中既可以编写面向对象编程范式的代码，又可以编写函数式编程范式的代码，还可以编写二者混合风格的代码。本书假设读者已经拥有 Java、C++ 或 C# 之类面向对象编程语言的经验，因此前面介绍面向对象编程的章节与其他介绍面向对象编程的书籍没有任何区别。但是，由于对大部分开发者来说函数式编程范式是一个相对较新的概念，因此接下来将简单介绍 Scala 函数式编程方面的内容。

函数式编程（`Functional Programming, FP`）风格是指在编写应用时，只使用**纯函数**和**不可变的值**。正如 Alvin Alexander 在《Functional Programming, Simplified》一书中描述，编写函数式程序的程序员内心渴望将自己的代码写得更具数学风格，将这些组合在一起的函数看作是一系列的代数式。用函数式编程的程序员更喜欢将自己看成是数学家，就是这样的渴望驱动着他们只使用纯函数和不可变的值，这样才能编写出更具数学风格的代码。

函数式编程是一个庞大的主题，本书的篇幅无法容纳完整的主题，因此下面的章节我们将品尝函数式编程的味道，展示 Scala 提供给函数式编程的一些工具。

8.1 纯函数

Scala 提供给函数式编程的第一个特性就是**纯函数**（`Pure Function`）。在《Functional Programming, Simplified》中，Alvin Alexander 给出了纯函数的定义：

- 函数的输出只依赖函数的输入变量；
- 不会改变任何的隐状态；
- 没有任何“后门”：不会从函数外面（控制台、网络服务、数据库、文件等等）读取数据，或将数据写入到函数外面。

根据上面的定义，任何时候调用一个纯函数，只要输入的值不变，总能得到相同的结果。例如，无论如何调用 `double` 函数，只要输入值是 `2` ，那么结果就永远是 `4` 。

> 这一点在并行处理数据非常重要，编写并行程序时——"状态乃万恶之源"。

8.1.1 内置的纯函数

`scala.math._` 包中定义的函数都是纯函数，例如：

- `abs`
- `ceil`
- `max`
- `min`

Scala 中 `String` 的方法中也有部分纯函数，例如：

- `isEmpty`
- `length`
- `substring`

Scala 集合类中的很多方法也是纯函数，例如 `drop` 、 `filter` 和 `map` 等。

8.1.2 非纯函数的例子

下面给出的函数不是纯函数，因为它们不满足纯函数的定义。

集合类的 `foreach` 方法不是纯函数，因为该函数需要使用副作用（ `side-effects` ），例如打印信息到 `STDOUT` 。

> `foreach` 不是纯函数的最大线索就是它的方法签名中返回值的类型是 `Unit` 。因为它没有任何的返回值，因此调用该函数必定是为了修改某种隐状态（副作用）。因此，所有返回类型为 `Unit` 的方法都不是纯函数。

与日期和时间相关的方法，例如 `getDayOfWeek` 、 `getHour` 和 `getMinute` 都不是纯函数，因为这些方法的输出会某些其他因素（时区、地区、当前时间等隐含状态）的影响。

通常，方法只要满足以下一条就不是纯函数：

- 读取隐含的输入，例如访问未曾显式传递给函数的变量和数据。
- 写出隐含的输出。
- 改变输入参数的值。
- 执行 `I/O` 操作。

8.1.3 仅有纯函数是不够的

当然，一个应用如果不能从程序外部读入或写出数据，不会有太多的用处，因此大家通常建议：

> 应用的核心部分用纯函数来编写，然后编写"非纯函数"将纯函数包裹起来，然后与外面的世界进行交互。如果用食物来类比，就好像用一个"**不纯**"的大饼裹住一个"**纯**"的鸡蛋。

有一些方法可以让这些**不纯**的函数看上更**纯**一些，例如你可能听说过用于处理输入、文件、网络和数据库访问的 `IO Monad` 。但是最后，函数式编程的应用依旧包括核心的纯函数和用于与外部世界交互的函数。

8.1.4 编写纯函数

编写纯函数是函数式编程中最简单的内容，只要用编写 Scala 方法的语法就可以编写纯函数。下面是一个将输入值加倍的纯函数：

```scala
def double(i: Int): Int = i * 2
```

尽管本书没有涉及到递归的内容，下面给出一个具有挑战性的例子，对一个整数列表 `List[Int]` 进行求和：

```scala
def sum(list: List[Int]): Int = list match {
  case Nil => 0
  case head :: tail => head + sum(tail)
}
```

尽管我们没有介绍递归，如果你依然能够读懂上面的代码，那么就能发现上面的代码满足纯函数的定义。

8.1.5 小结

函数式编程第一个要点就是定义纯函数：

> 纯函数的输出结果只依赖于输入与函数内部的算法，它不会从"外部世界"（函数定义之外）读取任何值，也不会修改"外部世界"的任何值。

第二个要点就是真实的应用总是由纯函数和不纯的函数构成，通常的建议是应用的核心使用纯函数，用于与外部事件通讯的部分编写不纯的函数。

8.2 传递函数

每一种编程语言都为编写纯函数提供了可能，但是 Scala 提供一个更为强大的功能——创建函数类型的变量。这一特性会带来很多好处，其中最常用的就是把函数作为参数传递给函数。再前面的例子中，已经在介绍 `map` 和 `filter` 的章节中看到过这样的例子：

```scala
val num = (1 to 10).toList

val doubles = num.map(_ * 2)
val lessThanFive = nums.filter(_ < 5)
```

在这些例子中，匿名函数被作为参数传递给 `map` 和 `filter` 方法。在介绍匿名函数的章节中，我们介绍过下面的代码：

```scala
val doubles = nums.map(_ * 2)
```

和把正常函数传递个 `map` 是等价的：

```scala
def double(i: Int): Int = i * 2
val doubles = nums.map(double)
```

如上例所示， Scala 允许将函数以及匿名函数传递给其他方法，这是函数式编程语言提供的一个强大的特性。

> 从技术的角度来说，一个以其他函数作为参数的函数被称之为：高阶函数（ `Higher-Order Function, HOF` ）

8.2.1 方法和函数

尽管 Scala 拥有定义函数的语法，但是从实践的角度来说，出于以下原因推荐使用 `def`：

- 拥有 C/Java/C# 背景的人更熟悉 `def` 格式。
- 可以把 `def` 定义的方法当作用 `val` 定义的函数。

第二个原因意味着如果你用 `def` 定义了如下的方法：

```
def double(i: Int): Int = i * 2
```

你就可以直接把 `double` 看成是变量传递给其他方法：

```
val x = ints.map(double)
            -----------
```

尽管 `double` 被定义为方法，但是 Scala 允许把它当作函数来使用。

把函数当作变量进行传递是函数式编程独有的特性，通过把函数作为参数传递给其他函数，可以帮助我们写出简洁而不失可读性的代码。

> 在 C 语言中也可以把函数作为参数传递给其他参数，但是必须用函数指针的形式。而在函数式编程语言中，函数可以直接用函数名进行传递，也就是说函数也是"一等公民"。我们在 Python 中可能早就习惯了将函数传递给其他函数，这是因为 Python 吸收了函数式编程的特性，包括在 Python 的列表生成式中就使用过 `map` 和 `filter` 方法。在 Java 中我们也可以使用 lambda 方法进行函数式的编程，当然新版的 C++ 标准也支持函数式编程范式的很多特性。现代编程语言通常融合了多种不同编程范式的特性，我们即使今后不使用 Scala 这款编程语言，也可以在别的语言中使用函数式编程范式的特性。

8.2.2 传递函数示例

如果你还不适应把函数作为参数传递给其他参数，可以在 REPL 中实验以下的范例：

```
List("foo", "bar").map(_.toUpperCase)
List("foo", "bar").map(_.capitalize)
List("adam", "scott").map(_.length)
List(1,2,3,4,5).map(_ * 10)
List(1,2,3,4,5).filter(_ > 2)
List(5,1,3,11,7).takeWhile(_ < 6)
```

要记得上面例子中所有的匿名函数都可以写成一个更"正常"的函数，例如：

```
def toUpper(s: String): String = s.toUpperCase
```

然后将其传递给 `map` 方法：

```
List("foo", "bar").map(toUpper)
```

或者是：

```
List("foo", "bar").map(s => toUpper(s))
```

这些方法都是等价的。

8.3 非空值

函数式编程编写代码有点像在书写一系列的代数式，因为在代数不使用 `null` 这样的值，因此在函数式编程中不使用 `null` 值。这就带来一个有趣的问题：如果你在写面向对象编程范式的代码，需要正常使用 `null` 怎么办?

Scala 的解决方法是使用类似 `Option/Some/None` 这样的类，本节将介绍它们如何使用。

8.3.1 示例1

例如要编写一个将字符串转换成整数的方法，如果获得的字符串无法转换成整数会抛出异常，为了能够优雅的处理异常，可能方法会写成下面的样子：

```
def toInt(s: String): Int = {
  try{
    Integer.parseInt(s.trim)
  } catch {
    case e: Exception => 0
  }
}
```

这种实现方法的初衷是将字符串转换为对应的整数，如果转换失败就返回 `0` 。在有些情况下，这样实现可以满足需求，但是这种实现并不完全准确。例如，该方法接收到字符串 `"0"` 和接收到字符串 `"foobar"` 得到的返回结果都是 `0` 。这就会带来一个现实的问题：当方法返回 `0` 的时候，怎么才能知道方法是接受了一个 `"0"` ，还是接收到了其他的字符串。用这种实现方法，我们是无法获得问题的答案的。

8.3.2 使用 `Option/Some/None`

Scala 解决这一问题的方法是使用 `Option` 、 `Some` 和 `None` 三件套。其中 `Some` 和 `None` 是 `Option` 的子类，因此解决方案为：

- 将 `toInt` 声明成 `Option` 类型;
- 如果 `toInt` 接收到一个可以转换的字符串，则把 `Int` 包裹到 `Some` 当中作为返回值;
- 如果 `toInt` 接收到的字符串无法转换则返回 `None` 。

实现的方法如下：

```
def toInt(s: String): Option[Int] = {
  try{
    Some(Integer.parseInt(s.trim))
  } catch {
    case e: Exception => None
  }
}
```

这段代码可以读作："当给定的字符串可以转换为整数，那么就把整数值包裹在 `Some` 中返回，例如 `Some(1)` 。如果字符串无法转换为整数，则返回 `None` 。"

下面是在 REPL 中调用 `toInt` 的例子：

```
scala> val a = toInt("1")
a: Option[Int] = Some(1)

scala> val a = toInt("foo")
a: Option[Int] = None
```

如同上面的例子所示，字符串 `"1"` 被转换为 `Some(1)`，而字符串 `"foo"` 被转换为 `None`。这就是 `Option/Some/None` 方法的本质。这种方法可以用于处理异常，或者其他需要使用 `null` 值的场景。

8.3.3 使用 `toInt` 方法

在调用 `toInt` 方法时，我们知道该方法的返回值时 `Option[Int]` 的子类，那么问题是如何处理这个返回值。

依据需要有两种方法可以处理这种返回值：

- 使用 `match` 表达式；
- 使用 `for` 表达式；

> 还有其他的方法可以处理这类返回值，但是以函数式编程的观点来看，这是最主要的两种方法。

使用 `match` 表达式

可以使用 `match` 表达式来处理 `toInt` 的返回值：

```
toInt(x) match {
  case Some(i) => println(i)
  case None => println("That didn't work.")
}
```

在这个例子中，如果 `x` 可以转成一个 `Int`，那么第一个 `case` 语句被执行；如果 `x` 不能转换成一个 `Int`，那么第二个 `case` 语句被执行。

使用 `for/yield`

另一种处理 `toInt` 方法返回值的方法是使用 `for` 表达式 ——即 `for/yield` 结构。例如，我们想要将三个字符串转换成整数，然后再将它们进行求和，`for/yield` 的实现方法如下：

```
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

当上面的 `for` 表达式运行结束后，`y` 的值有两种可能：

- 如果所有三个字符串都可以转换为整数，`y` 的类型是 `Some[Int]`，即包裹一个整数的 `Some`。
- 如果任意一个字符串无法转换为整数，`y` 的值就是 `None`。

可以在 REPL 中测试这个表达式，首先将下面三个字符串变量粘贴到 REPL：

```
val stringA = "1"
val stringB = "2"
val stringC = "3"
```

然后将 `for` 表达式粘贴到 REPL 中，可以得到如下结果：

```
scala> val y = for {
     |     a <- toInt(stringA)
     |     b <- toInt(stringB)
     |     c <- toInt(stringC)
     | } yield a + b + c
y: Option[Int] = Some(6)
```

如结果所示， `y` 中的值为 `Some(6)` 。

如果要查看转换失败的情况，只要把前面任一字符串改为无法转换为整数的字符串，就可以得到如下的结果：

```
y: Option[Int] = None
```

> `Option` 可以被看作是一个可以容纳 `0` 或 `1` 个元素的容器：
>
> - `Some` 是一个拥有 `1` 个元素的容器；
> - `None` 是一个拥有 `0` 个元素的容器。

8.3.4 使用 `foreach`

由于 `Some` 和 `None` 可被认识是容器，那么很容易就能想到它们本质上是集合类。和集合类相同，`Some` 和 `None` 支持 `map` 、 `filter` 和 `foreach` 等方法。

那么下面的代码会输出什么？

```
toInt("1").foreach(println)
toInt("x").foreach(println)
```

答案就是第一行语句会输出 `1` ，而第二行语句什么也不输出。第一行输出 `1` 是因为：

- `toInt("1")` 计算的结果为 `Some(1)` ；
- 因此该表达式等价于 `Some(1).foreach(println)` ；
- `Some` 类的 `foreach` 方法知道如何获得 `Some` 容器中的值，因此能够取出容器中的 `1` ，将其传递给 `println` 。

类似的，第二行不输出任何内容是因为：

- `toInt("x")` 的计算结果是 `None` ；
- `None` 类的 `foreach` 方法知道 `None` 不包含任何东西，因此就什么也不会做。

使用 `foreach` 和 `for` 表达式，不论结果是 `Some` 还是 `None` ，我们的处理代码都是一样的：

```
toInt("1").foreach(println)
toInt("x").foreach(println)
```

下面是 `for` 表达式的例子：

```
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

只要写一段代码就可以同时处理 `Some` 和 `None` 将简化我们的代码。只有在 `match` 表达式中，我们才需要判断结果到底是 `Some` 还是 `None`：

```
toInt(x) match {
    case Some(i) => println(i)
    case None => println("That didn't work.")
}
```

8.3.5 用 `Option` 代替 `null`

假设现在有一个 `Address` 类：

```
class Address (
                var street1: String,
                var street2: String,
                var city: String,
                var state: String,
                var zip: String
              )
```

所有的地址都有 `street1` 字段，但是 `street2` 则是可选的，因此可能会带来 `null` 值的使用：

```
val santa = new Address(
  "1 Main Street",
  null,        // <-- A null value!
  "North Pole",
  "Alaska",
  "99705"
)
```

为了解决这种问题，开发者可能会使用 `null` 值或空字符串来解决 `street2` 是一个可选字段的问题。在 Scala 中，正确的解决方案是将 `street2` 声明为可选的字段：

```
class Address (
                var street1: String,
                var street2: Option[String],
                var city: String,
                var state: String,
                var zip: String
              )
```

根据定义，开发者就可以编写正确的代码：

```
val santa = new Address(
  "1 Main Street",
  None,
  "North Pole",
  "Alaska",
  "99705"
)
```

或者：

```
val santa = new Address(
  "123 Main Street",
  Some("Apt. 2B"),
  "Talkeetna",
  "Alaska",
  "99676"
)
```

在处理可选字段是，可以使用之前介绍的 `match` 表达式、`for` 表达式和 `foreach` 方法来处理可选字段。

> 除了使用 `Option/Some/None` 三套件之外，Scala 还有一些其他的替代方法。例如 `Try/Success/Failure` 三套件，它们和 `Option/Some/None` 的区别在于：
>
> - 主要用于代码会抛出异常的类；
> - `Failure` 类允许你去访问异常的信息。
>
> 这组三套件通常用于和文件、数据库和互联网服务交互的场景，因为这些处理容易抛出异常。

8.3.6 小结

- 函数式编程范式不使用 `null` 值；
- 最主要的替代方案就是用 `Option/Some/None` 来替代 `null` 值；
- 最常用的处理 `Option` 的方式是使用 `match` 和 `for` 表达式；
- `Option` 可以把认为是包含 `1` 个元素的容器（ `Some` ）和包含 `0` 个元素的容器（ `None` ）；
- 可以用 `Option` 来定义构造器参数。

8.4 伴生对象

Scala 中的伴生对象（ `Companion Object` ）是一个和同名类位于同一个文件的 `object` 。例如，在下面保存于 `Pizza.scala` 文件的代码中， `Pizza` 对象就是 `Pizza` 类的伴生对象。

```
class Pizza {

}

object Pizza {

}
```

伴生对象与它的伴生类可以互相访问对方的私有成员（字段和方法）。这就意味着 `SomeClass` 中的 `printFilename` 方法可以正常工作，因为它可以访问伴生对象的 `HiddenFilename` 字段：

```
class SomeClass {
  def printFilename() = {
    println(SomeClass.HiddenFilename)
  }
}

object SomeClass {
  private val HiddenFilename = "/tmp/foo.bar"
}
```

本节将介绍伴生对象的最重要的一些特性。

8.4.1 创建新的实例无需使用 `new` 关键字

你可能已经注意到，在本书前面的有些例子中，在创建某个类的新实例的时候没有在类名之前使用 `new` 关键字，例如：

```
val zenMasters = List(
  Person("Nansen"),
  Person("Joshu")
)
```

这个功能的实现需要使用伴生对象，并在伴生对象中定义一个名为 `apply` 的方法，这个方法对于 Scala 的编译器有特殊的含义。这种语法糖可以在 Scala 中编写如下的代码：

```
val p = Person("Fred Flinstone")
```

经过编译之后，上面的代码会被自动转换为：

```
val p = Person.apply("Fred Flinstone")
```

伴生对象中的 `apply` 方法是一种**"工厂方法"**，可以使我们在创建新的类实例时无需使用 `new` 关键字。

`apply` 方法的实现

为了实现前面实例代码的功能，需要定义 `Person` 类伴生对象的 `apply` 方法：

```
class Person {
  var name = ""
}

object Person {
  def apply(name: String): Person = {
    var p = new Person
    p.name = name
    p
  }
}
```

要测试这段代码，需要把类和对象同时拷贝到 REPL 当中，这需要一点技巧：

- 首先从命令行启动 Scala 的 REPL；
- 输入 `:paste` 然后按下回车；

- REPL 将会显示以下文本：

```
// Entering paste mode (ctrl-D to finish)
```

- 然后将类与对象都拷贝到 REPL 中；
- 然后按下 `Ctrl + D` 结束 `paste` 过程。

完成上述步骤之后，REPL 的输出如下：

```
defined class Person
defined object Person
```

接下来就可以创建 `Person` 类的实例了：

```
val p = Person.apply("Fred Flinstone")
```

上面的代码调用了伴生对象的 `apply` 方法，更常用的方式是：

```
val p = Person("Fred Flinstone")
```

以及：

```
val zenMasters = List(
  Person("Nansen"),
  Person("Joshu")
)
```

创建类的实例过程是：

- 当你输入 `val p = Person("Fred")` ；
- Scala 编译器检查到在 `Person` 之前没有 `new` 关键字；
- Scala 编译器去模式匹配 `Person` 类的伴生对象中 `apply` 方法的签名；
- 如果匹配到了一个 `apply` 方法，就调用它；如果没有匹配到，就返回一个编译错误。

创建多个构造器

在伴生对象中可以创建多个 `apply` 方法，下面的代码中创建了两个构造器，一个接受一个参数，另一个接受两个参数。在这个例子中，使用了上一节介绍的 `Option` 类：

```
class Person {
  var name: Option[String] = None
  var age: Option[Int] = None
  override def toString: String = s"$name, $age"
}

object Person {

  // 单参数构造器
  def apply(name: Option[String]): Person = {
    var p = new Person
    p.name = name
    p
  }
```

```
15
16    // 双参数构造器
17    def apply(name: Option[String], age: Option[Int]): Person = {
18      var p = new Person
19      p.name = name
20      p.age = age
21      p
22    }
23  }
```

用之前介绍的方法将代码拷贝到 REPL，就可以用如下的方法来创建新的 `Person` 实例：

```
1  val p1 = Person(Some("Fred"))
2  val p2 = Person(None)
3  val p3 = Person(Some("Wilma"), Some(33))
4  val p4 = Person(Some("Wilma"), None)
```

打印这些值的结果，可以得到如下结果：

```
1  p1: Person = Some(Fred), None
2  p2: Person = None, None
3  p3: Person = Some(Wilma), Some(33)
4  p4: Person = Some(Wilma), None
```

> 在测试这个例子之前，最好清空 REPL 的内存，可以使用 `:reset` 来清空 REPL 的内存。

8.4.2 `unapply` 方法

可以通过在伴生对象中创建 `apply` 方法来构造新的对象实例，`unapply` 方法允许提取对象实例中的信息。

下面给出另一个版本的 `Person` 类和伴生对象：

```
1  class Person(var name: String, var age: Int)
2
3  object Person {
4    def unapply(p: Person): String = s"${p.name}, ${p.age}"
5  }
```

在伴生对象中定义了 `unapply` 方法，该方法接受一个 `Person` 类型的参数，然后返回一个 `String`。要手动测试 `unapply` 方法，首先需要创建一个 `Person` 实例：

```
1  val p = new Person("Lori", 29)
```

然后用如下代码测试 `unapply`：

```
1  val result = Person.unapply(p)
```

在 REPL 中可以得到如下的结果：

```
1  scala> val result = Person.unapply(p)
2  result: String = Lori, 29
```

如上例所示，当你将 `unapply` 放到伴生对象中，这就意味着你宣称创建了一个提取实例信息的方法。

`unapply` 可以返回不同的类型

在上面的例子中，`unapply` 返回了 `String` ，但是 `unapply` 方法可以返回任何类型的返回值。例如，返回包含两个字段的元组：

```
class Person(var name: String, var age: Int)

object Person {
  def unapply(p: Person): Tuple2[String, Int] = (p.name, p.age)
}
```

在 REPL 中的测试结果：

```
scala> val result = Person.unapply(p)
result: (String, Int) = (Lori,29)
```

由于这个 `apply` 方法返回的是元组，因此也可以这样处理：

```
scala> val (name, age) = Person.unapply(p)
name: String = Lori
age: Int = 29
```

> 遵从 Scala 的惯例， `unapply` 方法创建了一个提取器，可以更为方便的在 `match` 表达式中进行模式匹配。通常很少需要自己来定义 `unapply` 方法，事实上如果在定义类的时候，使用样例类（ `case class` ）的时候会自动提供 `apply` 和 `apply` 方法。

8.4.3 小结

- 伴生对象与所伴生的类有相同的名字，而且需要存储于同一个文件。
- 伴生对象和所伴生的类可以互相访问私有成员。
- 伴生对象的 `apply` 方法允许不使用 `new` 关键字来创建新的实例。
- 伴生对象的 `unapply` 方法允许提取类的实例中的独立成分。

8.5 样例对象

在介绍样例对象（ `case object` ）之前，首先介绍一下 Scala 中“正常”的对象。在本书前面提到，当你想要创建一个**单例对象**（ `singleton` ）时，可以使用 Scala 的 `object` 。

通常用于创建“工具”对象，例如：

```
object PizzaUtils {
    def addTopping(p: Pizza, t: Topping): Pizza = ...
    def removeTopping(p: Pizza, t: Topping): Pizza = ...
    def removeAllToppings(p: Pizza): Pizza = ...
}
```

或者：

```
object FileUtils {
    def readTextFileAsString(filename: String): Try[String] = ...
    def copyFile(srcFile: File, destFile: File): Try[Boolean] = ...
    def readFileToByteArray(file: File): Try[Array[Byte]] = ...
    def readFileToString(file: File): Try[String] = ...
    def readFileToString(file: File, encoding: String): Try[String] = ...
    def readLines(file: File, encoding: String): Try[List[String]] = ...
}
```

这就是 Scala 中最常用的 `object` 的使用方法。

8.5.1 样例对象

样例对象于对象类似，但是就像样例类比正常类拥有更多的特性一样，样例对象也比正常对象多了很多特性：

- 可序列化；
- 拥有默认的 `hashCode` 实现；
- 拥有加强版的 `toString` 实现；

由于拥有这些特性，样例对象主要用于以下情况：

- 创建枚举
- 其他对象之间传递的信息，可以用样例对象作为消息的容器

8.5.2 用样例对象创建枚举

在前面的例子中，我们创建了如下的枚举：

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

然后用如下的方式使用枚举：

```
case class Pizza (
    crustSize: CrustSize,
    crustType: CrustType,
    toppings: Seq[Topping]
)
```

8.5.3 用样例类传递消息

另一个使用样例类的场合是对"消息"这一概念进行建模的时候。例如，例如在亚马逊的 Alexa 应用中，我们需要能够传递 `speak the enclosed text` 、 `stop speaking` 、 `pause` 和 `resume` 这样的信息。在 Scala 中可以通过为这些信息创建单例对象来对信息进行建模：

```
case class StartSpeakingMessage(textToSpeak: String)
case object StopSpeakingMessage
case object PauseSpeakingMessage
case object ResumeSpeakingMessage
```

注意 `StartSpeakingMessage` 被定义为样例类，而不是样例对象。因为样例对象不能拥有任何的构造器参数。

有了这些信息对象，如果 Alexa 使用 Akka 库实现的，那么你就能发现类似下面的 `Speak` 类：

```
class Speak extends Actor {
  def receive = {
    case StartSpeakingMessage(textToSpeak) =>
    // code to speak the text
    case StopSpeakingMessage =>
    // code to stop speaking
    case PauseSpeakingMessage =>
    // code to pause speaking
    case ResumeSpeakingMessage =>
    // code to resume speaking
  }
}
```

这是一种能够安全的在 Scala 应用中传递消息的方法。

8.6 函数错误处理

由于函数式编程更像代数，因此没有 `null` 值或者异常（ `Exception` ）。但是当我们尝试去访问宕机的服务器或者访问丢失的文件，依然会产生异常。本节介绍在 Scala 中是如何处理功能性错误的。

8.6.1 `Option/Some/None`

我们已经介绍了在 Scala 中用来处理错误的三套件： `Option` 、 `Some` 和 `None` 。通过将方法的返回值定义为 `Option` 类型，来避免抛出异常或者返回 `null` 值，例如：

```
def toInt(s: String): Option[Int] = {
  try {
    Some(Integer.parseInt(s.trim))
  } catch {
    case e: Exception => None
  }
}
```

然后，在接下来的代码中就可以使用 `match` 和 `for` 表达式来处理 `toInt` 的结果：

```
toInt(x) match {
  case Some(i) => println(i)
  case None => println("That didn't work.")
}

val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

8.6.2 `Try/Success/Failure`

另外一组“三件套”：`Try` 、 `Success` 和 `Failure` 相较于 `Option/Some/None` 有以下两个优点：

- `Try` 把捕获异常变得异常简单；
- `Failure` 包含了异常的信息。

写下来我们将重写 `toInt` 方法，首先导入“三件套”：

```
import scala.util.{Try, Success, Failure}
```

使用 `Try` 的 `toInt` 方法：

```
def toInt(s: String): Try[Int] = Try {
  Integer.parseInt(s.trim)
}
```

使用 `Try/Success/Failure` 版本的 `toInt` 方法要比 `Option/Some/None` 版本的方法代码要断，当然上面的代码还可以更短：

```
def toInt(s: String): Try[Int] = Try(Integer.parseInt(s.trim))
```

首先测试能够成功返回信息的情况：

```
scala> val a = toInt("1")
a: scala.util.Try[Int] = Success(1)
```

接下来测试 `Integer.parseInt` 会抛出异常的情况：

```
scala> val b = toInt("boo")
b: scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input
string: "boo")
```

如上所示，`toInt` 返回的 `Failure` 中包含了异常的相关信息，即出错的原因。

有很多方法可以处理 `Try` 的结果，包括从失败中“恢复”的方法，但是最常用的方法还是使用 `match` 和 `for` 表达式：

```scala
toInt(x) match {
  case Success(i) => println(i)
  case Failure(s) => println(s"Failed. Reason: $s")
}

val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

注意，如果使用 `for` 表达式并且所有的字符串都可以正常转换为整数，那么 `for` 表达式的返回值包裹在 `Success` 中：

```scala
scala.util.Try[Int] = Success(6)
```

相反，如果转换失败，则返回一个 `Failure`：

```scala
scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input
string: "a")
```

在 Scala 的标准库和一些其他的第三方库中还有一些其他的类具有类似的行为，例如 `Either/Left/Right` 等。但是 `Option/Some/None` 和 `Try/Success/Failure` 是最常使用的。

通常 `Try/Success/Failure` 用于处理可能会抛出异常的代码，因为我们需要理解异常的原因；而 `Option/Some/None` 通常用于避免使用 `null` 值。

8.7 并行

如果想用 Scala 编写并行或者并发应用程序，可以继续使用原生的 Java `Thread`。但是 Scala 的 `Future` 可以把编写并行/并发程序变得更为简单。

> 并发是在一个处理器上执行多个任务，并行是在多个处理器上执行多个任务。可以简单的记为并发就是一个人同时吃好几个馒头，而并行则相当于好几个人同时吃好几个馒头。

在 Scala 的文档中是如此描述 `Future` 的：

> `Future` 表示一个值现在可能**可用**，也可能不可用，但是终究在某个时刻会变得**可用**，除非确实不可用。

为了便于理解，我们看下面的例子。在单线程编程时，我们通常要把函数调用的结果绑定要一个变量上，例如：

```scala
def aShortRunningTask(): Int = 42
val x = aShortRunningTask
```

在这段代码中，值 `42` 被**立即**绑定到变量 `x` 上。

在使用 `Future` 时，赋值的过程如下所示：

```scala
def aLongRunningTask(): Futrue[Int] = ???
val x = aLontRunningTask
```

但是由于无法确定 `aLongRunningTask` 运行所需的时间，因此 `x` 的值**当前**可能可用，也可能不可用，但是它一定会在某个时刻可用（`in the Future`）。

> 简单地说，`Future` 的思想类似于我们在硬件课程里面学到的中断系统，即当我们在另一个线程运行一个非常慢的计算时，我们希望当该运算结束的时候通知我们，并把结果交给我们。

在本节中，我们将介绍如何使用 `Future`，例如如何在 `for` 表达式中把多个并行运算的结果组合到一起。

8.7.1 源码

本节的源码可以从以下地址下载： github.com/alvinj/HelloScalaFutures

8.7.2 在 REPL 中使用 `Future`

Scala 的 `Future` 用于临时一次性的创建一个临时的并行包，适用于想要调用的算法所需运行的时间不明确的情况，例如调用网络服务或者执行一个运行时间很长的算法。使用 `Future` 可以新建一个线程来运行这些任务。

为了描述 `Future` 如何工作，首先看下面在 REPL 中运行的例子，首先导入以下的类：

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
```

接下来就可以创建一个 `Future`。例如，下面的 `future` 首先休眠 `10` 秒钟，然后返回 `42`：

```
scala> val a = Future { Thread.sleep(10*1000); 42 }
a: scala.concurrent.Future[Int] = Future(<not completed>)
```

上面的例子展示了创建 `Future` 最基本的方法：直接为需要长时间运行的算法构造一个 `Future`。

`Future` 拥有 `map` 函数，因此可以编写如下的代码：

```
scala> val b = a.map(_ * 2)
b: scala.concurrent.Future[Int] = Future(<not completed>)
```

在刚开始执行的时候（`10` 秒之内），上面的计算会显示 `Future(<not completed>)`，但是稍等一会直接检查 `b` 的值就能看到最终其中的值是 `84`：

```
scala> b
res1: scala.concurrent.Future[Int] = Future(Success(84))
```

需要注意的是 `84` 被包裹在 `Success` 中，而 `Success(84)` 则被包裹在 `Future` 中。这就意味着，`Future` 中的值是 `Try` 类型的实例：`Success` 或 `Failure` 类型。因此在处理 `Future` 的返回值时，可以使用 `Try` 的处理方法来处理该返回值，或者使用 `Future` 的回调（`callback`）方法。

最常用的的回调方法是 `onComplete`，该方法接受一个处理 `Sucess` 和 `Failure` 的偏函数（partial function）：

```
a.onComplete {
  case Success(value) => println(s"Got the callback, value = $value")
  case Failure(e) => e.printStackTrace
}
```

在 REPL 中执行将上述代码，在变量 `a` 的值计算完成之后会显示以下结果：

```
Got the callback, value = 42
```

8.7.3 示例应用

接下来的例子中将介绍如何定义多个 `Future`，内容包括：

- 如何创建 `Future`；
- 如何用 `for` 表达式将多个 `Future` 的结果组合到一起；
- 如何处理返回的 `Future`。

首先假设我们有一个访问 Web 服务以获取股票价格的方法，而访问 Web 服务获得返回结果所需的时间是不确定的。因此，需要将该方法定义为 `Future`，该方法以股票代码为输入，返回包裹在 `Future` 中的 `Double` 数值，下面是该方法的签名：

```
def getStockPrice(stockSymbol: String): Future[Double] = ???
```

为保持教程的简洁性，我们不会去真的访问一个真实的 Web 服务，而是用一个运行随机时间然后返回结果的方法作为这一方法的模型：

```
def getStockPrice(stockSymbol: String): Future[Double] = Future {
  val r = scala.util.Random
  val randomSleepTime = r.nextInt(3000)
  println(s"For $stockSymbol, sleep time is $randomSleepTime")
  val randomPrice = r.nextDouble() * 1000
  sleep(randomSleepTime)
  randomPrice
}
```

该方法首先进行不超过 `3000` 毫秒的休眠，然后返回一个随机的股票价格。通过将代码块放到 `Future` 控制结构中，就可以返回一个 `Future` 结果。

假设我们要并行的获得三支股票的价格，然后一次性将三个结果一同返回，可以用下面的代码实现这一功能：

```
package futures

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.{Failure, Success}

object MultipleFutures extends App{

  // 用于为下面的代码计算 delta time
  val startTime = currentTime

  // (a) 创建三个 futures
  val aaplFuture = getStockPrice("AAPL")
  val amznFuture = getStockPrice("AMZN")
  val googFuture = getStockPrice("GOOG")

  // (b) 用 for 表达式将结果组合到一起
  val result: Future[(Double, Double, Double)] = for {
    aapl <- aaplFuture
```

```
    amzn <- amznFuture
    goog <- googFuture
  } yield (aapl, amzn, goog)

  // (c) 处理 for 表达式运算的结果
  result.onComplete {
    case Success(x) => {
      val totalTime = deltaTime(startTime)
      println(s"In Success case, time delta: ${totalTime}")
      println(s"The stock prices are: $x")
    }
    case Failure(e) => e.printStackTrace
  }

  // 下面的代码对于并行时间较短的例子来说非常重要
  // 用于保持主进程出于活动状态
  sleep(5000)

  def sleep(time: Long): Unit = Thread.sleep(time)

  def getStockPrice(stockSymbol: String): Future[Double] = Future {
    val r = scala.util.Random
    val randomSleepTime = r.nextInt(3000)
    println(s"For $stockSymbol, sleep time is $randomSleepTime")
    val randomPrice = r.nextDouble() * 1000
    sleep(randomSleepTime)
    randomPrice
  }

  def currentTime = System.currentTimeMillis()
  def deltaTime(t0: Long) = currentTime - t0
}
```

> **问题**：如果上面的代码确实以并行的方式进行运行，那么 `totalTime` 的最大值是多少？
>
> **答案**：如果三个模拟访问 Web 服务的方法调用确实是并行运行的，那么总的时间就不会超过 `3000` 毫秒。如果他们是串行运行，则不会超过 `9000` 毫秒。

接下来分析一下这段是如何工作的。

创建 `Future`

首先创建了三个 `Future` ：

```
val aaplFuture = getStockPrice("AAPL")
val amznFuture = getStockPrice("AMZN")
val googFuture = getStockPrice("GOOG")
```

其中，`getStockPrice` 的定义如下：

```
def getStockPrice(stockSymbol: String): Future[Double] = Future { ...
```

在 `Future` 之后花括号中的代码被当作参数传递给 `Future` 伴生对象的 `apply` 方法，编译器会把这段代码大致翻译成如下的样子：

```
def getStockPrice ... = Future.apply {method body here}
                                -----
```

Scala 的 `Future` 与 Java 的 `Thread` 有这样一个不同的地方，在调用 `Future` 方法后会**立即**开始执行代码块中的程序，而在 Java 中在创建 `Thread` 进程之后要调用 `start` 方法才能开启线程的运行。

这三行对 `getStockPrice` 方法的调用**最终**会返回模拟的股票价格。之所以用**"最终"**这个词是因为不知道这些调用所需的运行时间，这就意味着我们希望**最终**能够得到返回结果（也有可能会得到一个不成功的结果）。

`for` 表达式

这个应用示例中的 `for` 表达式如下所示：

```
val result: Future[(Double, Double, Double)] = for {
  aapl <- aaplFuture
  amzn <- amznFuture
  goog <- googFuture
} yield (aapl, amzn, goog)
```

这段代码可以读作："一旦 `aapl` 、 `amzn` 和 `goog` 最终返回了它们的值，将它们的值组成一个元组，然后将该元组赋给 `result` 变量。"如代码所示， `result` 的类型为 `Future[(Double, Double, Double)]` ，即一个包裹在 `Future` 中由三个 `Double` 构成的元组。

值得注意的是，在调用 `getStockPrice` 时，应用程序的主线程并不会被停止。事实上如果在 `for` 表达式的前后打印 `System.currentTimeMillis()` ，会发现差别只有几微秒。

`onComplete`

应用程序的最后是 `onComplete` 方法：

```
result.onComplete {
  case Success(x) => {
    val totalTime = deltaTime(startTime)
    println(s"In Success case, time delta: ${totalTime}")
    println(s"The stock prices are: $x")
  }
  case Failure(e) => e.printStackTrace
}
```

`Future` 支持 `onComplete` 方法，可以用该方法以**副作用**的方式来处理 `Future` 的返回结果。与集合类以**副作用**的方式进行处理集合元素的 `foreach` 方法类似， `onComplete` 方法的返回类型也是 `Unit` 。可以用 `onComplete` 方法来处理返回值，以进行更新 GUI 、更新数据库等需要使用**副作用**的场景。

上面的代码可以读作："当 `result` 有了最终值，也就是说 `for` 表达式中所有 `Future` 都返回了结果之后，如果所有的调用都是 `Success` 的，则打印相关的信息，否则执行 `Failure` 分支，打印异常的调用栈信息。"

正如上面的代码所暗示， `Future` 是可能执行失败的，例如你调用一个 Web 服务，然后服务器宕机了。 `Future` 实例可以包含一个异常，因此当调用 `result.onComplete` 这样的方法时，可以去处理 `Failure` 的情况。

注意在执行 `for` 表达式时，JVM 的主线程并没有停止，也没有阻塞，会继续执行后面的代码。 `onComplete` 中的代码本质上是一个回调方法，在 `for` 表达式结果赋给 `result` 之后，这段代码才会被执行。

`sleep`

最后值得讨论的就是在应用示例的最后，需要调用 `sleep` 方法：

```
sleep(5000)
```

这个调用是用来让主线程在 JVM 中保持 `5` 秒钟的活动状态，这会导致在其他线程返回结果之前，主线程就结束了。这会导致在主线程里执行的 `onComplete` 中的代码无法执行。通常在正常的应用中不会出现这种问题，因为正常的应用中主线程会执行很长的时间，尤其是在开发 Web 服务类应用时。

其他代码

在前面介绍的代码中，调用了以下方法来获得运行时间：

```
def currentTime = System.currentTimeMillis()
def deltaTime(t0: Long) = System.currentTimeMillis() - t0
```

8.7.4 其他 `Future` 方法

`Future` 提供的一些常用的回调方法：

- `onComplete`
- `onSuccess`
- `onFailure`

`Future` 支持 Scala 集合的一些方法：

- `filter`
- `foreach`
- `map`

还有一些常用的"顾名思义"的方法：

- `andThen`
- `fallbackTo`
- `recoverWith`

更多关于 `Future` 的细节可以参考阅读：Futures and Promises

8.7.5 小结

- 可以构造 `Future` 在主线程之外的线程中运行任务。
- `Future` 用于处理那些可能运行时间较长的并发性任务。
- `Future` 在构造之后就立即开始运行。
- `Future` 提供不同的回调方法来简化处理并发进程所需的处理异常、线程惯例等工作。
- 除了可以用 `onComplete` 方法处理返回结果之外，还可以使用处理集合的方法 `map` 、 `flatMap` 、 `filter` 、 `andThen` 等来处理返回结果。
- `Future` 的值是一个 `Try` 类型的实例： `Success` 或 `Failure` 。
- 如果要由多个 `future` 生成一个结果，通常可以用 `for` 表达式将它们的结果组合到一起。

> 其他参考代码：GUI App: Future Board

> 对于较为大型的并行应用程序，Akka 为 Scala 提供了 `actor model` 库，该代码库提供了实现大型并行程序的实现方法。

更多关于 Scala 的细节，可以参考 Scala 官网的 Guides and Overviews section。