

---

# Table of Contents

封面	1.1
欢迎	1.2
内容简介	1.3
1 Kotlin是什么？为什么选用它？	1.4
1.1 初尝Kotlin	1.4.1
1.2 Kotlin的主要特性	1.4.2
1.3 Kotlin应用	1.4.3
1.4 Kotlin哲学	1.4.4
1.5 使用Kotlin工具	1.4.5
1.6 总结	1.4.6
2 Kotlin基础	1.5
2.1 基本元素：函数和变量	1.5.1
2.2 类和属性	1.5.2
2.3 表示和处理选择：枚举和`when`	1.5.3
2.4 遍历：while和for循环	1.5.4
2.5 Kotlin异常	1.5.5
2.6 总结	1.5.6
3 函数定义和调用	1.6
3.1 在Kotlin中创建集合	1.6.1
3.2 让函数便于调用	1.6.2
3.3 把方法添加到他人的类中：扩展函数和属性	1.6.3
3.4 使用集合：vararg、infix调用和库支持	1.6.4
3.5 使用字符串和正则表达式	1.6.5
3.6 让你的代码变得整洁：本地函数和扩展	1.6.6
3.7 总结	1.6.7
4 类、对象和接口	1.7
4.1 定义类层级	1.7.1
4.2 声明一个带有精简的构造函数或者属性的类	1.7.2
4.3 编译器生成的方法：数据类和类委托	1.7.3
4.4 用object关键字声明一个类并创建一个它的实例，然后将两者组合	1.7.4

---

4.5 总结	1.7.5
5 用lambda编程	1.8
5.1 lambda表达式和成员引用	1.8.1
5.2 集合的函数式API	1.8.2
5.3 惰性的集合操作: 序列操作	1.8.3
5.4 使用Java的函数式接口	1.8.4
5.5 带有接收器的lambda: with和apply	1.8.5
5.6 总结	1.8.6
6 Kotlin类型系统	1.9
6.1 为空性	1.9.1
6.2 原始的和其他基本的类型	1.9.2
6.3 集合与数组	1.9.3
6.4 总结	1.9.4
7 操作符重载和其他习惯用法	1.10
7.1 重载算术操作符	1.10.1
7.2 重载比较操作符	1.10.2
7.3 集合与范围的习惯用法	1.10.3
7.4 析构声明和组件	1.10.4
7.5 重用属性访问器逻辑: 委托属性	1.10.5
7.6 总结	1.10.6
8 高阶函数: lambda作为参数和返回值	1.11
8.1 声明高阶函数	1.11.1
8.2 内联函数: 移除lambda的天花板	1.11.2
8.3 高阶函数中的控制流	1.11.3
8.4 总结	1.11.4
9 泛型	1.12
9.1 类型参数	1.12.1
9.2 运行时泛型: 类型参数清除与具体化	1.12.2
9.3 变化: 泛型和子类型	1.12.3
9.4 总结	1.12.4
10 标注和反射	1.13
10.1 标注的声明与应用	1.13.1
10.2 反射: 在运行时检测Kotlin对象	1.13.2
10.3 总结	1.13.3

---

---

11 DSL构造	1.14
11.1 从API到DSL	1.14.1
11.2 构建结构化的API: DSL中带有接收器的lambda	1.14.2
11.3 带有invoke的更灵活的块嵌套	1.14.3
11.4 Kotlin DSL实践	1.14.4
11.5 总结	1.14.5
A: 构建Kotlin项目	1.15
B: Kotlin代码文档化	1.16
C: Kotlin生态	1.17

---

# Kotlin

## IN ACTION

Dmitry Jemerov  
Svetlana Isakova

MEAP

 MANNING



# 欢迎

感谢您购买了MEAP计划中的《Kotlin实战》！我们将为您介绍一门实用、安全、简洁和能够与Java或操作的的新型编程语言--Kotlin。通过Kotlin，你能够用更少的代码、更高级的抽象和更少的繁琐的事务来实现你的项目。我们假设你已经很熟悉Java的服务器端或者Android应用开发者。从Java切换到Kotlin是一个很平滑的过程。Kotlin的学习曲线并不陡峭。如果你想尝试Kotlin，你不仅可以在新项目中使用它，而且可以在已有的Java代码中使用。你可以简单的通过交互式控制台或者在线平台进行尝试这门语言的一些特性。

这本书由三部分组成。第一部分将会教授这门语言的基本语法。第二部分将会集中于能够让你构建可重用的抽象、高阶函数、相关的库和完整的领域定制语言等特性。最后，第三部分将会集中于如何运用kotlin到真实项目的细节，比如集成构建系统、Android开发支持和并发编程。

我们两位作者都是已经为Kotlin团队的成员并且工作超过五年。因此，你应该相信你拿到的是最直接、最权威的资料。即便是Kotlin的1.0版本已经发布出来了，它仍然在持续的改进。所以我们会确保MEAP的书籍信息始终和当前的语言状态保持一致。

我们期待你的反馈。如果书中有任何存在争议、不清晰或者确实的情况，请你立即到[作者在线论坛](#)留言。

--Dmitry Jemerov 和 Svetlana Isakova

## 第一部分：介绍

1. [Kotlin是什么？为什么选择它？](#)
2. [Kotlin基础](#)
3. [函数定义和调用](#)
4. [类、对象和接口](#)
5. [使用lambda](#)
6. [Kotlin类型系统](#)

## 第二部分：拥抱Kotlin

7. [操作符重载和其他的习惯用法](#)
8. [高阶函数：lambda作为参数和返回值](#)
9. [泛型](#)
10. [标注和反射](#)
11. [DSL Construction](#)

## 附录：

- A. [构建Kotlin项目](#)
- B. [注释Kotlin代码](#)
- C. [Kotlin的生态](#)

---

鸣谢参与校对、纠错的[GitHub:eggery](#)、[oncereply](#)、[gefangshuai](#)

这个章节将会涵盖以下内容：

- Kotlin 概述
- Kotlin 的主要特性
- Android 开发和服务器开发的可能性
- 和其他语言相比，Kotlin 的脱颖而出之处
- 用 Kotlin 编写和运行代码

首先，Kotlin 是关于什么的？Kotlin 是一门把 Java 平台作为目标的新的编程语言。它简洁、安全、优雅而且专注于和 Java 代码间的互操作性。它几乎可以用于如今 Java 遍及的所有地方：服务器端开发、Android 应用开发和更多其他的方面。Kotlin 能够很好的与现有的 Java 库和框架并存。而且，它运行的性能不亚于 Java。让我们以一个小案例为开始解释 Kotlin 像什么。这个例子定义了一个 Person 类。它创建了一些 Person 类的对象。然后找出其中年纪最大的一个 Person 类对象并将其打印出来。即便是在这么一小段块代码中，你也能够看到 Kotlin 的许多有趣的特性。我们对其中的一些代码做了高亮处理，以便你能够在书中的后续部分快速的找到它们。代码解释的比较粗略，但是如果你有些地方不能立刻明白也不要担心。我们会在后续的任意章节详细的讨论每一个概念。

如果你想尝试运行这个例子，最便捷的方式是使用在线的平台<https://try.kotl.in/>。输入这份示例代码并点击 Run 按钮，代码将会被执行。代码如下：

```
package ch01.ATasteOfKotlin

data class Person(val name: String,           // 1 数据类
                  val age: Int? = null)       // 2 可为空的类型(Int?); 变量声明的
// 默认值

fun main(args: Array<String>) {               // 3 顶层函数
    val persons = listOf(Person("Alice"),
                           Person("Bob", age = 29)) // 4 命名声明

    val oldest = persons.maxBy { it.age ?: 0 }   // 5 lambda表达式; elvis操作符
    println("The oldest is: $oldest")           // 6 字符模板
}

// The oldest is: Person(name=Bob, age=29)      // 7 自动生成`toString`方法
```

你声明一个简单的数据类。它带有两个属性：name 和 age。age 属性默认值为 null (如果它的值没有被指定的话)。当创建人名单时，你忽略了Alice的年龄，因此它的默认值 null 被用上了。之后你使用 maxBy 函数来查找名单中年龄最大的人。传递给函数的lambda表示式接收一个参数，同时，你使用 it 作为默认的参数名。如果 age 是 null 的话，三元操作符 ?: 将返回0。由于Alice的年龄没有指定，三元操作符用0替代。最终Bob很荣幸的成为了年纪最大的人。刚才所看到的一切，你都喜欢吗？阅读更多，学习更多，成为一个Kotlin专家吧！我们希望你能很快能够在你的项目中看到这样的代码，而不仅仅是在这本书中。



对于Kotlin是一种什么样的编程语言，你可能已经有一定的想法了。让我们更详细地看看它的关键属性吧。首先，让我们来看看你能用Kotlin构建什么样的应用。

## 1.2.1 目标平台：服务器端、Android以及运行Java的其他平台

Kotlin的首要目标是为Java提供更加简洁、更具生产效率、更安全的可选方案。它适合当下所有用到Java的场景。Java是极度流行的编程语言。它被用于各种各样的环境。小到智能卡（Java卡片技术），大到谷歌、推特、领英和其他大企业的数据中心。在大多数这些场景，使用Kotlin能够帮助开发者全程以更少的代码和繁杂事务来实现你的目标。

Kotlin的常见领域：

- 构建服务器端代码（通常是web后台应用）
- 构建运行于Android设备的移动应用

但是Kotlin也能在其他场合工作。举个例子，你可以使用Intel Multi-OS 引擎在iOS设备上运行Kotlin代码。你也能够配合使用JavaFX和Kotlin来构建桌面应用。

在Java之外，Kotlin也能编译成Javascript。这将允许你在浏览器端运行Kotlin代码。写这本书时，Kotlin对JavaScript的支持在JetBrains公司内部仍然处于探索和原型阶段。因此，它不在本书的讨论范围内。Kotlin的后续版本也将会考虑其他的平台。

如你所见，Kotlin的目标是如此之广阔。kotlin并不局限于单一领域的问题或者解决现代软件开发所面临的单一类型的挑战。相反，它为开发过程中出现的所有任务带来全面的效率提升。对于特定领域或编程范式的库，Kotlin为你提供了良好的集成。接下来让我们来看看Kotlin作为一门编程语言的关键特性。

## 1.2.2 静态类型

就像 Java 那样，Kotlin 是一种静态类型的编程语言。这意味着在编译时就可以确定程序中每一个表达式的类型。编译器可以验证你所访问的对象中的方法和字段。

这（种特性）跟JVM的具有代表性的动态类型编程语言，比如Groovy和JRuby，形成对比。那些语言允许你定义能够存储或返回任何数据类型的变量和函数，并且可以在运行时解析方法和字段引用。这样的代码更简短，在创建数据结构时也更为灵活。但它也有缺点，诸如无法在编译时检测命名拼写错误，并导致运行时错误这样的问题。另一方面，跟Java相比，Kotlin并不需要你在代码中显式的指定每一个变量的类型。在许多场景中，变量类型能够根据上下文自动推断。这将允许你省略类型声明。

以下是一个最简单的例子：

```
var x = 1
```

你声明了一个变量。由于它以一个整数值进行初始化，Kotlin自动推断这个变量的类型为 `Int`。编译器根据上下文判断变量类型的能力叫做类型推断。

以下是静态类型的一些好处：

- 性能 - 由于不需要再运行时判断需要调用哪个方法，方法调用将会变得更快。
- 可读性 - 编译器校验了程序的准确性，所以在运行时发生崩溃的可能性将会降低。
- 可维护性 - 由于你能看到代码调用了什么类型的对象，使用不熟悉的代码将会变得更加容易。
- 工具支持 - 静态类型使得可靠的重构、精确的代码填充和其他的IDE特性变得可能。

由于Kotlin对类型推断的支持，大部分跟静态类型有关的冗余信息将会消失，因为你不需要显示指定类型。

如果你考究Kotlin类型系统的细节，你将会发现许多熟悉的概念。类、接口和泛型以类似于Java的方式出现。所以你大部分的Java知识应该能够很快速的转变为Kotlin的。尽管Kotlin里面有些东西是新的。

其中最重要的一点是Kotlin支持可以为空的类型(**nullable type**)。通过在编译时检查可能的空指针异常，这一特性将使你编写更可靠的程序。我们将会在这一章的后续部分回到可为空的类型，并在第六章详细的讨论。

Kotlin类型系统中的另一个新事物是它所支持的函数类型(**functional types**)。为了了解这是什么，我们先看看函数式编程的主要思想以及Kotlin是如何支持它的。

### 1.2.3 函数式和面向对象

作为一个Java程序员，毫无疑问，你对面向对象编程的核心概念了如指掌。但是你可能对函数式编程感到陌生。函数式编程的关键概念如下：

- 函数是一等公民 - 你把函数（行为块）看做是一个值。你可以把。存储在变量中，把它们作为一个参数进行传递或者从其他函数中返回它们。
- 不变性 - 你使用的是不可改变的对象。一旦它被创建，它的状态便不可更改。
- 没有副作用 - 你使用的纯函数对于给定的相同输入将会返回相同的结果。同时，它不会修改其他对象的状态或者和外界进行交互。

以函数式风格编写代码，你能从中得到好处呢？首先，它很简洁。跟对应的指令式代码相比，函数式代码会更加优雅和简洁。因为把函数当做一个值将会给你带来更强大的抽象能力，这将避免你的代码出现冗余。想象一下你有两段相似的代码。它们用于完成一个类似的任务（例如：查找集合中的匹配元素），但是细节方面（如何匹配检测到的元素）不一样。你可以简单的把共同的逻辑提取到一个函数中，并将不同的部分当做参数进行传递。参数是函数自身的，但是你可以使用为匿名函数准备的**lambda**表达式来表达：

```
fun findAlice() = findPerson { it.name == "Alice" }    // 1 findPerson()包含了人名查找的  
通用逻辑  
fun findBob() = findPerson { it.name == "Bob" }      // 2 大括号里面的代码块指定了你想查  
找的具体的人
```

函数式代码的第二个好处是多线程安全。多线程程序中的一个最大的错误来源是没有正确的同步的情况下修改了来自多个线程的同一个数据。如果你使用了不可修改的数据结构和纯函数，你可以确保不会出现不安全的修改。同时你不需要去设计复杂的同步方案。

最后，函数式编程意味着容易测试。没有副作用的代码通常更容易测试。函数能够被隔离测试而无需一堆安装代码来构造函数所依赖的完整环境。一般而言，函数式风格能够被用于包括 Java 在内的任何编程语言中。它其中的许多部分也被视为为良好的编程风格。但是并非所有的语言都提供了易于使用的语法和库支持来使用函数式编程。例如：Java8以前的Java版本都没有提供大部分的函数式支持。Kotlin从一开始就有着丰富的特性来支持函数编程。具体如下：

- 函数类型 允许函数接收作为参数的其他函数或者返回其他函数
- **lambda**表达式 让你使用最小的模板分发代码块
- 数据类 为创建不可变值对象提供了精简的语法
- **API** 标准库为以函数式风格使用对象和集合提供了丰富的API

Kotlin允许你使用函数式风格进行编程，但是不强制使用。当你有需要时，你可以毫无障碍的使用可变数据以及编写有副作用的函数。当然，使用基于接口和类继承的框架和Java一样的方便。用Kotlin写代码时，你可以将面向对象和函数式方法混合使用，同时，使用最合适的工具来解决问题。

### 1.2.4 免费、开源

不管是什么用途，Kotlin，包括编译器、函数库和所有相关的库都是完全开源和免费的。

Kotlin使用Apache 2许可。它在GitHub的开发上是开放的。同时我们欢迎社区贡献。你也有三种可选的开源集成开发环境来开发Kotlin应用：IntelliJ IDEA 社区版、Android Studio和Eclipse都是完全支持的（当然IntelliJ IDEA 旗舰版亦可）。

现在你应该理解Kotlin是一种什么样的编程语言了，我们看看Kotlin在具体的应用实践中有什么样的好处。

正如我们之前提到的，Kotlin的两个主要应用领域是服务器端和Android开发。让我们看看依次这些领域以及为什么Kotlin适合它们。

## 1.3.1 服务器端的Kotlin

服务器开发是一个相当宽泛的概念。它包括以下所有类型的应用和更多的其他应用：

- 向浏览器返回HTML页面的web应用
- 通过HTTP协议来向移动应用暴露JSON API的后台应用
- 通过RPC协议进行通信的微服务

多年来，开发者一直使用Java构建这些种类的应用。如今，我们已经积累了一大堆的框架和技术来辅助应用的开发。这些应用通常都不是独立的或者从零开始开发的。世上几乎总是有现存的系统被扩展、升级或被替代。新的代码必须集成到现有系统的一部分。而这部分代码可能在很多年之前就编写好了

在这样的环境下，Kotlin最大的一个优势是它能够与现有Java代码进行无缝的交互。无论你是否编写一个新的组件或者移植已有代码到Kotlin，Kotlin都会非常适合。当你需要在Kotlin中扩展Java类或者以特定的方式标注类方法以及类属性时，你不会遇上问题。这样的好处是系统代码将会更加紧凑、可靠和易于维护。

与此同时，Kotlin为开发这样的系统准备了许多的高新技术。例如，它对建造者模式的支持允许你用精简的语法来创建任意的对象图。它也保留了全套的抽象和语言方面的代码重用工具。

这个特性最简单的一个用例是一个HTML生成库。这个库能够用精简和完全类型安全的方案来替代外部模板语言。示例如下：

```
fun renderPersonList(people: Collection<Person>) =
    createHTML().table { // 1
        for (person in people) { // 2
            tr { // 1
                td { +person.name } // 1
                td { +person.age } // 1
            }
        }
    }

// 1 映射为HTML标签的函数
// 2 一个常规的Kotlin循环
```

如你所见，你可以很容易的把映射HTML标签的函数和常规的Kotlin语言构造组合起来。你不再需要使用单独的模板语言。它独立的语法有待学习。当你生成HTML页面时，你只需要使用一个循环。另一个你能够使用Kotlin的整洁、精简的DSL的用例是持久化框架。举个例子，Exposed框架提供了一种简单的方式来读取描述SQL数据库结构的DSL。而且它完全通过Kotlin代码来执行查询。它有全面的类型检查。这里有一个小案例来向你展示可能的情况：

```

object CountryTable : IdTable() { // 1 描述数据库中的表
    val name = varchar("name", 250).uniqueIndex()
    val iso = varchar("iso", 2).uniqueIndex()
}

class Country(id: EntityID) : Entity(id) { // 2 创建一个数据库实体对应的类
    var name: String by CountryTable.name
    var iso: String by CountryTable.iso
}

val russia = Country.find { // 3 你可以仅使用Kotlin代码查询这个数据库
    CountryTable.iso.eq("ru")
}.first()

println(russia.name)

```

我们将会书在后续部分，委托属性一节和第11章详细的考虑这些技术。

## 1.3.2 Kotlin 在Android方面的应用

一个典型的移动应用和一个典型的企业级应用是非常不一样的。移动应用更加的小。它与现有集成代码的依赖较少。它经常需要快速分发同时要确保在各种设备可靠的运行。Kotlin恰好也能用于那种项目。

Kotlin的语言特性，结合支持Android框架的编译器特殊插件，把Android开发变为一个更具生产效率和舒适的体验。诸如为控件添加侦听器或者绑定布局元素到字段的常见开发任务，能够使用更少的代码来实现。有时候根本不需要代码（编译器会为你生成代码）。Anko库也是由Kotlin团队构建的。即便需要进一步在许多的标准Android API外围添加Kotlin友好的适配器。，它仍然提升了你的体验 这里有一个简单的Anko示例，只是为了让你体验一下使用Kotlin开发Android是一种什么样的感觉。你可以把这份代码放到一个 Activity 中，一个简单Android应用就准备好了！

```

verticalLayout {
    val name = editText() // 1 创建一个简单文本域
    button("Say Hello") { // 2 被点击时，这个按钮会显示文本域的值
        onClick { toast("Hello, ${name.text}!") } // 3 附加侦听器和展示面包条的精简API
    }
}

```

另一个使用Kotlin的巨大优势是能够获得更好的应用可靠性。如果你有更多Android应用开发经验，对于 Unfortunately, Process Has Stopped 对话框，你将不会陌生。当你的应用抛出一个未处理的异常时，通常是 NullPointerException，这个对话框将会出现。Kotlin的类型系统使用它的精确的空值跟踪特性来让空指针异常变得没那么迫切。大部分的代码在Java中会导致 NullPointerException，在Kotlin中会编译失败，这将确保你在应用到达用户手上之前修复错

误。

同时，因为Kotlin是完全兼容Java6的，它的使用不会引入任何新的兼容性问题。你将会受益于Kotlin所有的既酷又新的特性。你的用户将会依然能够在他们的设备上运行你的应用，即使他们不是在最新版的Android上运行。

在性能方面，使用Kotlin也不会带来任何的坏处。Kotlin编译器产生的代码和常规的Java代码在执行时同样高效。Kotlin占用的运行时资源非常小，因此你不会遇到应用安装包大小会大幅增加的问题。当你使用lambda表达式时，许多的Kotlin标准库函数将会以内联的方式出现。内联lambda表达式确保没有新的对象会被创建，同时，应用不会遭遇额外的垃圾回收暂停问题。

考虑Kotlin和Java之间的比较优势，让我们现在看看Kotlin的哲学--使得Kotlin与其他运行在JVM上的现代语言出现区别的主要特性。

当我们讨论Kotlin时，我们喜欢说它是一种专注于交互性的、实用的、精简的和安全的编程语言。我们说的每一个词的精确含义是什么呢？让我们来依次了解它们。

## 1.4.1 实用性

实用意味着对我们来说是简单的东西：Kotlin是一门为解决现实问题的实践性语言。它的设计是基于多年的大规模系统设计的行业经验。它的特性是被选中来解决许多软件开发中所遇到的案例。此外，JetBrains公司内部和社区的开发者已经使用了Kotlin早期版本许多年了。而且，他们的反馈已经在发行版中成型了。这让我们可以自信满满的说：Kotlin能够成功的帮助开发者解决真实项目中的问题。

Kotlin也不是一门研究型语言。我们并非尝试着去提升编程语言设计的艺术境界，也不是在探索计算机科学中的创新构想。相反的，我们尽可能依靠那些已经出现在其他编程语言中并且被证明为成功的特性和解决方案。这降低了语言的复杂性。同时，通过让你沿用熟悉的概念会让语言变得易于上手。同时，Kotlin并不强迫你使用任何特定的编程风格或范式。当你开始学习这门语言的时候，你能够使用你在Java开发经验中熟悉的风格和技术。随后，你会慢慢的发现Kotlin中更强大的特性。学习并将它应用到你的代码中，把你的代码变得更加精简和符合Kotlin的语言习惯吧！

Kotlin的实用主义的另一方面是它对工具的专注。对开发者的生产效率而言，一个智能的开发环境正如一门好语言那般重要。因此，把IDE支持作为一个后来添加的东西并不是一个可选项。就Kotlin来说，IntelliJ IDEA插件的开发紧跟跟编译器开发的步伐。在设计语言特性时，我们一直都考虑到了工具。

在帮助你探索Kotlin特性的过程中，IDE支持也扮演了一个重要角色。在许多场合，工具将会自动检测能够被更加精简的构想所替代的共同的代码模式，并为你提供了代码填充功能。通过研究自动填充功能所使用的语言特性，你可以学习如何在你的代码中也应用这些特性。

## 1.4.2 精简

这是一个常识：开发者花费更多的时间在阅读既有代码上而不是编写新代码。想象一下你是一个大项目的开发团队的一员，你需要添加一个新的特性或者修复一个错误。你第一步会做什么呢？你会查找你需要修改的代码片段的位置。只有在这之后你才能实现修复。你读了许多代码来发现什么是你必须做的事。这份代码可能是由你的同事近期编写的或者是其他不再工作于这个项目的人编写的，又或者是你很久以前写的。只有理解了周边代码以后，你才能做出必要的修改。

代码越简洁，你理解起来越快。当然了，良好的设计和富有表现力的命名也是非常重要的。但是编程语言的选择和它的简洁也同样重要。如果一门语言的语法清晰的表达了你所读代码的意图，并且它不是晦涩难懂，还带有八股代码来解释实现的意图，那么它就是简洁的。在Kotlin中，我们尽力确保你编写的所有代码都是有意义的，而不仅仅是符合约定。大量的标准的Java八股代码，例如getter、setter和将构造器参数值赋给字段的逻辑，在Kotlin中都消失了，而且导致你的代码变得混乱。



代码不必那么长的另一个原因是只需编写显示的代码来完成一般的任务，例如，定位一个集合中的一个元素。和许多其他现代语言一样，Kotlin拥有丰富的标准库。这些标准库能让你通过库函数调用来替代那些冗长、重复的代码段。Kotlin对lambda的支持使得传递一小块代码到库函数变得容易。这让你封装所有的通用部分到函数库，同时仅仅保留用户代码中独特的、业务相关的部分。代码变得莫须有的冗长的另一个原因是，你必须编写显式的代码来执行通用的任务，例如，定位集合中的元素。就像许多其他现代语言那样，Kotlin拥有大量的标准库来让你使用库函数调用替代那些冗长、重复的代码片段。Kotlin对lambda的支持使得向库函数传递小段代码变得容易。这将允许你封装库中所有的通用部分。同时，你可以只保留用户代码中独特的，特定任务的部分。与此同时，Kotlin不会尝试着去把源代码折叠成可能的最少字符。举个例子，尽管Kotlin支持操作符重载，但是用户不能够定义他们自己的操作符。因此，函数库开发者不能用含义模糊的标点符号序列来替代函数名。虽然单词稍微长了一些，但是通常来讲，单词比标点字符更容易阅读和在文档中查找。代码越简洁，花在编写的时间就越少。更重要的是阅读的时间更少。而这将提升你的效率并让你把事情完成的更快。

### 1.4.3 安全

一般而言，当我们说一门编程语言安全时，我们是指它的设计阻止了程序中的某些错误。当然，这不是一个绝对的特性。没有一门编程语言能阻止所有可能的错误。另外，阻止错误往往要付出一些代价。你必须为编译器给出更多的关于程序预定操作的信息，以便编译器能够确认这些信息匹配到程序做了什么。就因为如此，你得到的安全级别和要求提供更多的详细的标注导致的生产效率的丢失之间一直都存在着矛盾。

使用Kotlin，我们尝试着花费较小的代价来达到一个比Java更高级别的安全级别。运行在JVM已经提供了许多的安全性保证：举个例子，内存安全，防止了缓冲区溢出，还有其他动态内存分配的错误使用导致的其他问题。作为JVM平台的一门静态类型语言，Kotlin确保了你的应用的类型安全。这达到了比Java更小的消耗：你不需要指定所有的类型声明，因为在许多场景下编译器能够自动推断类型。

Kotlin也能够超越Java。这意味着通过编译时检查而不是在运行时失败可以阻止更多的错误。最重要的是，Kotlin努从你的程序中移除 `NullPointerException`。Kotlin的类型系统跟踪能或者不能为空的值，并且禁止了在运行时导致 `NullPointerException` 异常的操作。它需要的额外成本是最小的：标记为一个可为空的类型只需要一个单独的字符，在末尾加一个问号：

```
val s: String? = null    // 1 可能为空
val s2: String = ""     // 2 一定不为空
```

另外，Kotlin提供了许多便捷的方式来处理可为空的数据。这在消除应用崩溃方面有着极大的帮助。

Kotlin有助于避免的另一种异常是 `ClassCastException`。当你抛出一个对象为一种类型但没有初次检查它的类型是否正确时，这种异常就会发生。

在Java编程界，开发者经常遗漏了检查，因为类型名称在检查和接下来的转换中会重复。



另一方面，在Kotlin中，检查和转型被合并为一个单独的操作：一旦你检查了类型，你就可以应用该类型的成员而无需额外的类型转换。因此，Kotlin中没有理由跳过检查，也没有机会出错。下面是工作原理展示：

```
if (value is String)           // 1 类型检查
    println(value.toUpperCase()) // 2 调用该方法
```

## 1.4.4 互操作性

考虑到互操作性，你的第一反应可能是：“我能够使用我已有的库吗？”。使用Kotlin，答案是：“是的，一定可以！”。不管函数库要求你使用的API种类，你可以在Kotlin中使用它们。你可以调用Java方法，扩展Java类和实现接口以及为你的Kotlin类应用Java标注等等。

不同于其他的JVM语言，Kotlin在互操作性方面走得更远，也使得你毫不费力的从Java代码中调用Kotlin代码。这并不需要什么技巧：Kotlin类和方法能够像常规的Java类和方法那样被调用。这将给你带来在项目的任何地方混合Java和Kotlin代码的终极灵活性。当你开始在你的Java项目中采用Kotlin时，你可以在你的代码库中的任意单一文件中运行Java-Kotlin转换器，其余的代码将会继续编译和正常工作而无需任何修改。不管你所转换的类是什么角色，这个功能都有效。

Kotlin专注于互操作性的另一个方面是尽最大可能的使用现有的Java库。例如，Kotlin并没有自己的集合库。它依赖于Java标准库的类，通过额外的函数来扩展他们以更加方便的使用Kotlin（我们将会在第XX章讨论这个机制）。这意味着当你调用Java API时，你不需要通过Kotlin包装或者转换对象。Kotlin提供的所有API都不会出现运行时消耗。

Kotlin工具也为多语言项目提供了全面的支持。它能够编译任意一个混合Java和Kotlin的源文件，不论他们之间是如何的相互依赖。这个IDE特性对其他语言也是有效的。它将允许你做以下事情：

- 在Java和Kotlin源文件中自由切换
- 调试混合语言项目并在用不同语言编写的代码中单步跟踪
- 使用Kotlin重构和正确的升级你的Java函数，反之亦然

幸好，到目前为止我们已经说服你尝试Kotlin。现在，你该如何入门呢？在接下来的环节，我们将会讨论从命令行和使用不同工具编译和运行Kotlin代码的步骤。

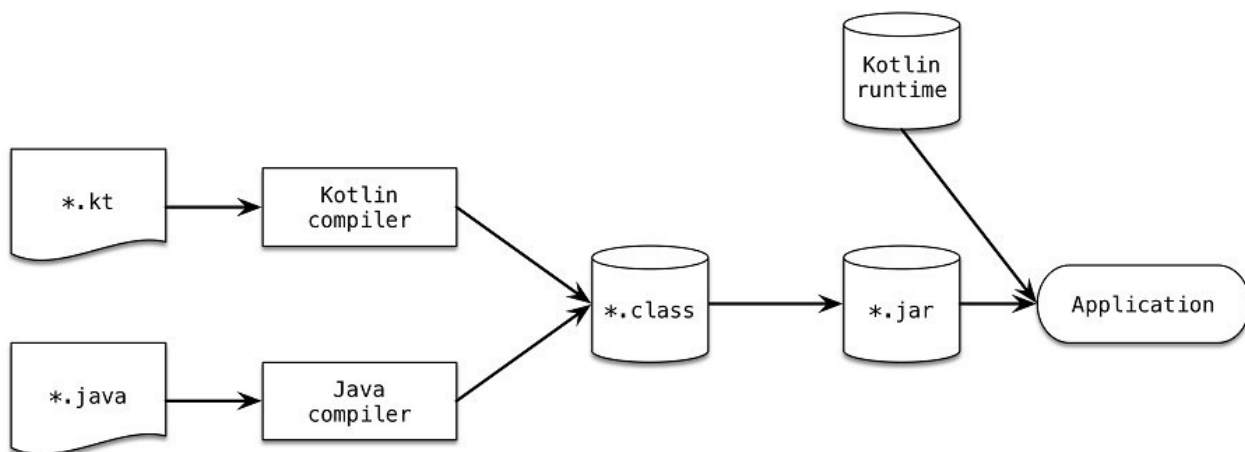
跟Java一样，Kotlin是一个编译语言。这意味着在你运行Kotlin代码之前，你必须编译它。让我们讨论一下编译步骤是如何工作的，然后瞧瞧为你提供生计的各种工具。如果你需要有关环境配置的更多信息，请参考Kotlin官网的[“教程”](#)章节

## 1.5.1 编译Kotlin代码

Kotlin源码通常存放在带有 `.kt` 后缀的文件中。Kotlin编译器分析源码并生成 `.class` 文件，就像Java编译那样。生成的 `.class` 文件会被打包然后使用你使用的应用类型的标准流程执行。在最简单的情景中，你可以使用 `kotlinc` 命令从命令行来编译你的代码，然后使用 `java` 命令来执行你的代码：

```
kotlinc <source file or directory> -include-runtime -d <jar name>
java -jar <jar name>
```

一个简单的Kotlin构建流程描述如图1.1。



使用Kotlin编译器编译的代码依赖于Kotlin运行时库。它包含了Kotlin标准类库的定义和Kotlin添加到标准Java API中的扩展。运行时库需要发放到你的应用中。

在大多数现实生活场景中，你可能会使用像Maven、Gradle或者Ant这样的构建系统来编译你的代码。Kotlin跟所有的这些构建系统都是兼容的。我们将会在附录A中详细讨论。所有的这些构建系统也都支持由Kotlin和Java组成的混合语言项目。Mave和Gradl也会处理包含Kotlin运行库作为你的应用的依赖。

## 1.5.2 IntelliJ IDEA和Android Studio插件

Kotlin的IntelliJ IDEA支持以及跟语言本身并行的开发了。它功能最全的开发环境对Kotlin也是可用的。IntelliJ IDEA成熟、稳定。它为Kotlin开发环境提供了全套的开发工具。

IntelliJ IDEA 15及更新的版本中，Kotlin插件已经包含在内并且是开箱即用的。因此它不需要额外的安装。你也可以使用免费、开源的IntelliJ IDEA社区版或者IntelliJ IDEA旗舰版。在新

项目对话框中选择Kotlin，你就可以开始编码了。

如果你正在使用Android Studio，你可以通过插件管理器来安装Kotlin插件。在设置对话框中，选择插件，然后点击安装JetBrains插件按钮，最后从列表中选择Kotlin。

## 1.5.3 交互式 shell

如果你想快速的试验Kotlin代码小片段，你可以使用交互式shell（又称REPL）。在REPL中，你可以逐行输入Kotlin代码来查看它的执行结果。为了启动REPL，你也可以运行不带参数的 `kotlinc` 命令或者使用IntelliJ IDEA插件中的响应式菜单按钮。

## 1.5.4 Eclipse插件

如果你是一个Eclipse用户，你也可以在你的IDE中使用Kotlin。Kotlin的Eclipse插件提供了必要的IDE功能，例如，导航和代码填充。这个插件在Eclipse商店是可用的。为了安装这个插件，请选择Help > Eclipse Marketplace 菜单项，然后在列表中搜查Kotlin。

## 1.5.5 在线平台

这是体验Kotlin的最简便的方法。它不需要任何的装和配置。在这个网站(<https://try.kotl.in>)，你可以发现一个在线的平台来编写、编译和运行Kotlin程序。这个平台有代码样例来演示Kotlin的特性，也有一系列的练习题来互动学习Kotlin。

## 1.5.6 Java-Kotlin转换器

掌握一门新语言来提升开发效率从来都不是容易的。所幸的是，我们已经构建了一个一些捷径来为你用现有的Java知识来加速学习和理解Kotlin。

当你刚开始学习Kotlin，不记得准确的语法时，转换器能够帮助你表达一些想法。你可以编写对应的Java代码然后将其粘贴到Kotlin源文件。转换器将会自动的帮你把代码翻译成Kotlin。转换结果不会一直都是最符合Kotlin习惯用法的，但是转换后的代码可以正常工作。你可以继续你的任务。

转换器对于把Kotlin引入现有的Java项目是很有帮助的。当你需要编写新的类是，你可以使用Kotlin从零开始。但是如果你需要明显的改变一个现有的类，在这个过程中你可能需要写一些Kotlin代码。以上就是转换器的用处。首先你把类转换成Kotlin，然后你使用现代编程语言的所有优势来添加修改。

在IntelliJ IDEA中使用转换器是相当容易的。你也可以复制Java代码段然后粘贴到Kotlin代码文件中，又或者如果你想转换这个文件的话，调用Java文件到Kotlin的转换动作。Eclipse和在线平台中，转换器也是可用的。



- Kotlin是静态类型的，支持类型推断的，在保持代码精简的同时维持准确性和性能。
- kotlin同时支持面向对象和函数式编程风格，通过把函数放在一等公民的位置实现更高层次的抽象，通过支持不可变值简化了测试和多线程开发。
- Kotlin在服务器端应用运行良好。它能全面支持现有的Java框架并为公共任务提供了新的工具，例如生成HTML和保持一致性。
- Kotlin在Android开发方面也是可用的。由于紧凑的运行时，Android API的特殊编译器支持，丰富的函数库为常见的Android开发任务提供了Kotlin友好的函数支持。
- Kotlin是免费和开源的。它为主流IDE和构建系统提供了全面的支持。
- Kotlin是优雅的、安全的、精简的以及互操作性强的（语言）。这意味着它专注于使用已经被证明的方案来解决常见任务，阻止一般的错误，例如：`NullPointerException`，支持紧凑和易读的代码，松散的Java集成功能。

这一章节将会覆盖以下内容：

- 声明函数、变量、类、枚举类型和属性
- 控制结构
- 智能类型转换
- 抛出和处理异常

在这一章节，你将会了解到如何声明必要的编程要素：变量、函数和类。这一路上，你将会开始了解到Kotlin中的属性概念。

你将会了解到在Kotlin中如何使用不同的控制结构。它们跟你在Java中熟悉的东西非常相似，但是在重要的方面进行了增强。

我们将会介绍智能类型转换这个概念。它把类型检查和转换合并到一个操作。最后我们将会讨论异常处理。在这一章的末尾，你将能够使用基本的Kotlin语言来编写代码，尽管代码可能不是非常Kotlin。

这一章节将会为你介绍每一个Kotlin程序的基本组成元素：函数和变量。你将会看到Kotlin是如何让你省略许多的类型声明以及如果鼓励你使用不可变的，而不是可变数据的。

## 2.1.1 Hello, World!

让我们以一个经典的例子开始吧：一个打印“Hello,world”的程序。在Kotlin中，它只是一个函数：

```
fun main(args: Array<String>) {  
    println("Hello, world!")  
}
```

你在这个简单的代码块中观察到什么样的特性和语法呢？核对下面这个列表：

- `fun` 关键词被用来声明一个函数。事实上，用Kotlin来编程是非常有趣的！
- 参数类型写在参数名后面。正如你在后面看到的那样，这对于变量什么也是适用的。
- 函数可以声明在文件的顶层。你不需要把它放入一个类中。
- 数组只是一个类。不同于Java，Kotlin没有特殊的语法来声明数组类型。
- 你可以写成 `println` 而不是 `System.out.println`。Kotlin标准库使用更加精简的语法来为标准的Java库函数提供了众多的包装器。`println` 函数就是其中之一。
- 你可以省略行末的分号，就像许多其他的现代语言那样。

到目前为止，一切看起来都很好！后续我们将会更详细的讨论其中的一些话题。现在，让我们一起来深入探讨函数声明语法吧！

## 2.1.2 函数

你已经看到了如何声明一个没有任何返回值的函数。但是你应该在何处为拥有有意义的结果的函数添置一个返回类型呢？你可以猜到它应该在参数列表后面的某个位置：

```
fun max(a: Int, b: Int): Int {  
    return if (a > b) a else b  
}  
  
>>> println(max(1, 2))  
2
```

在这个案例中，函数声明以 `fun` 关键字为开始，接着是函数名：`max`。接着是圆括号中的参数列表。返回类型跟在参数列表后面，以冒号分隔。图2.1向你展示了函数的基本结构。注意，在Kotlin中，`if` 是一个带有结果值的表达式。这跟Java中的三元操作符很相似：`(a > b) ? a : b`：

在这个案例中，函数声明以 `fun` 关键字为开始，接着是函数名：`max`。接着是圆括号中的参数列表。返回类型跟在参数列表后面，以冒号分隔。图2.1向你展示了函数的基本结构。注意，在Kotlin中，`if` 是一个带有结果值的表达式。这跟Java中的三元操作符很相似：`(a > b) ? a : b`：

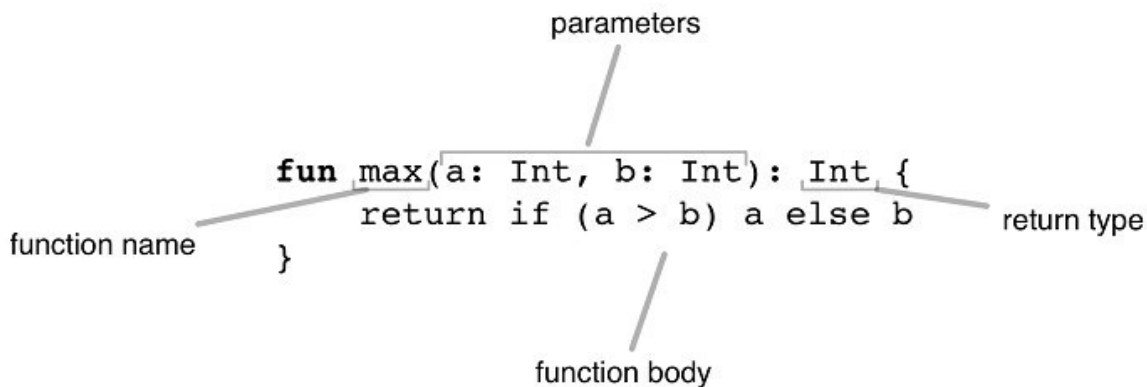


图2.1 Kotlin函数声明

#### NOTE 声明和表达式

在Kotlin中，`if` 是一个表达式，并不是一个声明。两者的区别在于，表达式有值。它可以用作另一个表达式的一部分。然而，一个声明却总是闭合块中的一个顶层元素，而没有自己的值。在Java中，所有的控制结构都属于声明(statement)。而在Kotlin中，循环以外的大多数控制结构都是表达式。正如你在书中后续将会看到的那样，将控制结构和其他表达式结合起来的能力让你更精简地表达许多常见的模式。另一方面，赋值在Java中是表达式，但在Kotlin中却是声明。这有助于避免比较和赋值之间的困惑。而这种困惑是错误的常见源头。

## 表达式主体

你可以进一步简化之前的函数。因为它的内容部分仅仅是有一个表达式组成的，你可以移除大括号和 `return` 声明，使用表达式作为整个函数的主体。

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

如果一个函数拥有大括号，我们说这个函数有一个块主体。如果它直接返回一个表达式，我们说它有一个表达式主体。

#### TIP IntelliJ IDEA Tip

IntelliJ IDEA 提供了意图动作来转换两种类型的函数："转换为表达式主体"和"转换为块主体"

通常有表达式主体的函数能够在Kotlin代码中快速的找出来。这种风格不仅仅用在琐碎的单行函数，也用在单独的、更复杂的求值表达式中，例如 `if`, `when`, 或者 `try`。当我们讨论 `when` 构造时，你可以在这个章节的后续部分看到这样的函数。



为什么这里的函数没有返回类型声明呢？Kotlin，作为一个静态类型的语言，不需要每个表达式在编译时都有一个类型吗？事实上，每一个变量和表达式都有一个类型。每一个函数都有一个返回类型。但是对于表达式主体函数来说，编译器能够分析用做主函数主体的表达式，并使用表达式的类型作为函数返回类型，即使没有拼写出来。这种类型的分析通常叫做类型推断。

注意，只有表达式函数才允许忽略返回值。对于有一个返回值的块主体的函数来说，你必须制定返回类型并且显示的写上 `return` 声明。这是一个明智的选择。一个真实世界的函数经常是很长的。它可能包括多个返回值。有一个显示的返回类型和返回声明有助于你快速的理解函数将会返回什么内容。接下来让我们来看看变量声明的语法。

## 2.1.3 变量

在Java中，你是以一个类型来开始变量声明的。在Kotlin中，这是无效的，因为Kotlin让你从许多的变量声明中忽略类型。因此，在Kotlin中，你是以一个关键词开始的。你可能（也许不会）在变量名后面放置类型。让我们来声明两个变量：

```
val question = "The Ultimate question of Life, the Universe, and Everything"
val answer = 42
```

这个例子忽略了类型声明，但是你也可以按你想的那样显式的声明指定类型：

```
val answer: Int = 42
```

正如有着表达式主体的函数，如果你不指定类型，编译器分析初始化器表达式并使用它的类型作为变量类型。在这个案例中，初始化器,42,有一个 `Int` 类型。因此变量将会有同样的类型。

如果你使用一个浮点类型的常量，变量将会是 `Double` 类型：

```
val yearsTocompute = 7.5e6 // 7.5 * 106 = 7500000.0
```

数字类型在 [XREF ID\\_主要类型](#) 一节中深入探讨。

如果一个变量没有一个初始化器，你需要显式的指定它的类型：

```
val answer: Int
answer = 42
```

如果你没有给出能够推断出这个变量的类型的值，编译器无法推断出它的类型。

## 可变的和不可变的变量

如果你没有给出能够推断出这个变量的类型的值，编译器无法推断出它的类型。

## 可变的和不可变的变量

这里有两个关键词来声明一个变量：

- **val**(从一个值) 不可变的引用。用**val**声明一个变量不能在初始化后重新分配值。它对应于Java中的**final**变量。
- **var**(从一个变量) 可变的引用。它的值是可以改变的，比如变量。这个声明对应于Java中的常规(非**final**)变量。

默认的，在Kotlin中，你应该尽量使用 **val** 声明所有的变量。仅在你必要的时候将 **val** 改为 **var**。使用不可变引用、不可变对象和函数没有副作用，这让你的代码更加接近函数式风格。我已经在第一章简要的接触了它的好处。我们也将第五章回到这个主题。

一个 **val** 变量必须在定义块执行时被初始化而且只能一次。但是如果编译器能够确保唯一的初始化声明能够其中一个被执行，你可以根据情况用不同的值初始化变量：

```
val message: String
if ( canPerformOperation() ) {
    message = "Success"
    // ... perform the operation
}
else {
    message = "Failed"
}
```

注意，尽管一个 **val** 引用本身是不可变的，也不可被改变。但是它指向的对象可能是可变的。例如，下面的代码是完全有效的：

```
val languages = arrayListOf("Java")    // 1 声明一个不可变的引用
languages.add("Kotlin")                // 2 引用指向可变的对象
```

在书中后续部分，第六章，我们将会更详细的讨论可变和不可变对象。

尽管 **var** 关键字允许一个变量改变自己的值，但是他的类型是固定的。例如，下面的代码不能成功编译：

```
var answer = 42
answer = "no answer" // 错误：类型匹配错误
```

这是一个字面量错误。因为它的类型（**String**）不是预期的（**Int**）。编译器仅仅从初始化器推断变量类型。在决定变量类型时，编译器并没有把考虑后面的赋值。

如果你需要把值存储到一个类型不匹配的变量中，你需要手动转换强制把值变为正确的类

让我们回到开篇的“Hello World”的例子。下面是典型习题的下一步已经用Kotlin的方式来通过人名来问候：

```
fun main(args: Array<String>) {  
    val name = if (args.size > 0) args[0] else "Kotlin"  
    println("Hello, $name!")    // 打印"Hello, Kotlin",或者"Hello, Bob"如果你传递"Bob"作为一个声明  
}
```

这个示例介绍了叫做字符串模板的特性。在代码中，你声明了一个 `name` 变量，然后在接下来的字符串值中使用了它。向很多脚本语言一样，Kotlin允许你在字符串值中通过在变量名前面放置 `$` 字符来引用局部变量。这相当于Java的字符串拼接(`"Hello, " + name + "!"`)，但是更加紧凑和高效。当然，表达式是静态检查的。如果你尝试着去应用一个不存在的变量，代码不会通过编译。

如果你需要在一个字符串中包含 `$` 符号，你可以使用转义功能：`println("\$x")`，打印 `$x` 但不会将 `x` 解析为变量引用。

你不会被局限于简单的变量名。你也可以使用更复杂的表达式。表达式所有的内容将放在大括号内：

```
fun main(args: Array<String>) {  
    if (args.size > 0) {  
        println("Hello, ${args[0]}!") // 使用`${}`语法来插入args数组的第一个元素  
    }  
}
```

你也可以在双引号内部嵌套双引号，就像在一个表达式内部一样：

```
fun main(args: Array<String>) {  
    println("Hello, ${if (args.size > 0) args[0] else "someone"}!")  
}
```

在使用字符串一节，我们将会回到字符串这个主题并讨论更多你能用它来做什么。

现在你知道如何声明函数和变量了。让我们走进继承层级并看看类这个主题。这次，你会使用Java到Kotlin转换器来帮助你开始使用新语言的特性。

你可能对面向对象编程不陌生而且非常熟悉类抽象。Kotlin在这方面的概念，你可能会非常熟悉。但是将会发现可以使用更少的代码来实现许多常见的任务。我们将会在第四章详细的讨论类。

首先，让我们来看看JavaBean的 `Person` 类，尽管它只有一个属性， `name`：

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

在Java中，构造器主体包含的代码通常是完全重复的：它把参数值赋给对应名字的字段。在Kotlin中，这个逻辑无需如此多的样板代码来表达。

在1.5.6章节，我们介绍了Java-Kotlin转换器：一个能够自动用同等高效的Kotlin代码替代Java代码的工具。让我们来看看转换器的工作原理，并转化 `Person` 为Kotlin代码：

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

看好了，它是不是你想象的那样？如果你有用过其他的现代JVM语言，你可能看过相似的东西。这种类型的类通常被叫做值对象。许多语言提供了一个精简的语法声明它们。相对于Java版本来说，用Kotlin风格写成的如此简单的类更加容易处理。

注意，代码从Java转换为Kotlin的过程中修饰器 `public` 不见了。在Kotlin中 `public` 是默认的可见性，因此，你也可以忽略它。

## 2.2.1 属性

如你所确信的那样，类的思想是封装数据以及用于将数据转换为单一实体的代码的。在Java中，数据存储通常在私有的字段中。如果你需要让类的客户端访问数据，你可以提供访问方法：获取函数，可能还有设置函数。你已经在 `Person` 类中看到一个这样的例子。设置函数也可以包含额外的逻辑来验证传递的值，发送更改的通知等等。

在Java中，字段和它的访问器组合通常被称作属性。许多的框架大量使用这个概念。在Kotlin中，属性是一个能完全替代字段和访问器函数的一等语言特性。你可以像什么变量那样在类中声明一个属性：使用 `val` 和 `var` 关键词。一个声明为 `val` 的属性是只读的。相反的，`var` 属性是易变的和可被改变的：

```
class Person(
    val name: String,          // 1 只读属性生成和琐碎获取器
    var isMarried: Boolean    // 2 可写属性：一个字段，一个获取函数和一个设置函数
)
```

基本上，当你声明一个属性时，你要声明对应的访问器（只读属性需要获取器，可写属性获取器两者）。默认情况下，访问器的实现都是琐碎的：一个的创建是为值，获取函数和设置函数返回和更新字段值。但是，如果你想的话，你可以声明一个使用不同的逻辑来计算或更新属性值的自定义访问器。

先前的 `Person` 的简洁声明隐藏了跟原始Java代码相同的本质实现：它是一个有着被构造函数初始化并且能够通过相应的获取函数进行访问的私有属性的类。这意味着你能够从Java和Kotlin以同样的方式使用这个类，跟在哪里声明它无关。用法看上去是一样的。以下是你如何通过Java代码使用 `Person` 类（的示例）：

```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

注意，当 `Person` 类在Java和Kotlin中有定义时，结果看起来是相同的。Kotlin的 `name` 属性在Java中暴露为一个叫做`getName()`的获取函数。对于布尔值属性来说，使用到了一个特殊的获取函数命名规则：属性名以 `is` 开头，获取函数没有添加额外的前缀。因此，在Java中你可以调用 `isMarried()`。

如果你把之前的代码转换成Kotlin代码，你将会得到以下结果：

```
>>> val person = Person("Bob", true)    // 1
>>> println(person.name)                // 2
Bob
>>> println(person.isMarried)           // 2
true

// 1 你可以不使用new关键词来调用构造函数
// 2 你可以直接访问属性，但是会调用获取函数
```

现在，你可以直接引用属性而不是调用获取函数。逻辑保持不变，但是代码变得更精简了。易变属性的设置器以同样的方式工作：在Java中，你使用 `person.setMarried(false)` 来表示离婚，而在Kotlin中，你可以写成 `person.isMarried = false`。

**TIP Properties of Java classes**

你也可以为Java中的类使用Kotlin属性语法。Java类中的获取函数可以作为Kotlin中的 `val` 属性进行访问，同时，获取器/设置器对可以作为 `var` 属性进行访问。举个例子，如果一个Java类定义了叫做 `getName()` 和 `setName()` 的方法，你可以把它作为一个叫做 `name` 的属性进行访问。如果它定义了 `isMarried()` 和 `setMarried()` 方法，Kotlin中对应的属性名将会是 `isMarried`。

大多数情况下，属性有一个用来存储属性值的对应的字段。但是如果属性值能够在忙碌状态下被计算出来，举个例子，通过其他属性，你也可以使用自定义的获取函数来表达它。

## 自定义的获取器

这个章节向你展示了如何写一个自定义的属性访问器的实现。假定你声明一个能说出它是否为正方形的矩形类。你不需要独立的字段来存储信息，因为你能够不停地检查高度和宽度是否相等：

```
class Rectangle(val height: Int, val width: Int) {  
    val isSquare: Boolean  
        get() { // 属性的获取函数声明  
            return height == width  
        }  
}
```

`isSquare` 属性不需要一个字段来存储它的值。它只有一个自定义的已经提供实现的获取函数。每一次属性被访问时，它的值都会被重新计算。

注意，你不需要使用大括号中的完整语法。你也可以写成 `get() = height == width`。这个属性的调用保持不变：

```
>>> val rectangle = Rectangle(41, 43)  
>>> println(rectangle.isSquare)  
false
```

如果你需要从Java访问这个属性，你可以像往常那样调用 `isSquare()`。

你可能会问声明一个没有形式参数的函数或者没有自定义访问器的属性是否会更好。这两个选项是相似的：（它们的）实现或者执行是没有差别的，它们只是在可读性方面有所不同。通常的，如果你描述一个类的特征（属性），你应该把它声明为一个（类的）属性。

在第4章，我们将会展示更多的使用类和属性的例子。同时我们也会着眼于显式声明构造



器的语法。如果你此时此刻已经迫不及待了，你可以一直使用 `Java-to-Kotlin` 代码转换器。在我们继续讨论其他语言特性时之前，现在先让我们来简要的检查 `Kotlin` 代码在磁盘上是如何组织的。

## 2.2.3 Kotlin源代码布局：目录和包结构

你（已经）知道 `Java` 是如何将所有的类组织成一个个的包的。`Kotlin` 也有包这个概念。而且跟 `Java` 中的非常相似。每一个 `Kotlin` 文件可以在（文件）开头有一个 `package` 声明。所有定义在文件中的声明（类、函数和属性）都会被放置在这个包里面。如果（它们）在同一个包里面，其他文件中定义的声明也能够被直接使用。如果不在同一个包，它们需要被导入。跟 `Java` 一样，导入声明放在文件的开头，同样也是使用 `import` 关键字。

这有一个源文件示例来展示包声明和导入语法：

```
package geometry.shapes    // 1 包声明

import java.util.Random    // 2 导入标准的Java类库

class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
    get() = height == width
}

fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}
```

`Kotlin` 并没有在导入类和函数之间做区别。它允许你使用 `import` 关键字导入任何类型的声明。你可以通过名字导入顶层的函数：

```
package geometry.example

import geometry.shapes.createRandomRectangle    // 1 通过名字来导入函数

fun main(args: Array<String>) {
    println(createRandomRectangle().isSquare)    // 2 很少概率会打印true
}
```

你也可以通过在包名后面添加 `.*` 来导入定义在某一个的包里所有的声明。注意这个通配符导入不仅会使得包内所有的类可见，也会使得顶层函数可见。在前一个示例中，写上 `import geometry.shapes.*` 而不是显式的导入也能让代码正确编译。

在 `Java` 中，你把你的类放到一个匹配包结构的文件或目录结构下。举个例子，如果你有

一个叫做 `shapes` 的包，这个包有多个类。你需要 每一个类放到一个文件名匹配，同时目存放在一个叫做 `shapes` 的目录中的独立文件里。图2.2展示了 `geometry` 包和它的子包在 `Java` 中是如何组织的。（我们）假设 `createRandomRectangle` 函数位于一个单独的文件 `RectangleUtil` 中。

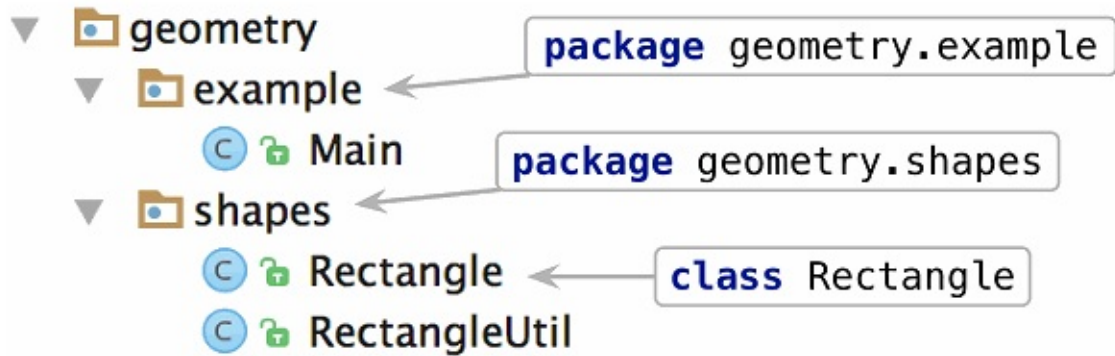


图2.2 在 `Java` 中，目录层级跟包结构层级一样的。

在 `Kotlin` 中，你可以把多个类放到同一个文件中并且可以为文件选择任意的名字。`Kotlin` 并不会对在源文件在磁盘中的布局强加任何的限制。你可以使用任意的目录结构来组织你的文件。举个例子，你可以在 `shapes.kt` 文件中定义 `geometry.shapes` 包所有的内容并把这个文件放在 `geometry` 文件夹中但并不需要创建一个单独的 `shapes` 文件夹（见图2.3）。



图 2.3 你的包层级并不需要跟随目录的层级

然而，在大部分情况下，跟随 `Java` 目录布局和根据包结果将源文件组成到目录中依然是一个很好的实践。在 `Java` 和 `Kotlin` 混合代码中保持这样的结构是相当重要的。因为这样做能让你平滑的实现代码的迁移而不会导致其他的疑惑。但是你应该毫不犹豫的将多个类拉到同一个文件中，尤其是类很小的时候（在 `Kotlin` 中，类经常是很小的）。

现在，你知道程序是如何组织的了。让我们继续学习基本的概念以及 `Kotlin` 中的控制结构吧！



在这个章节里，我们将讨论 `when` 结构。它可以被理解为 Java 中 `switch` 的替代，但事实上，它更加强大而且常用。一路上，我们将会给你一个声明枚举的例子并讨论智能类型转换的概念。

## 2.3.1 声明枚举类

让我们以为这本严肃的书添加一些富有想象的图片并看看颜色枚举值为开始吧：

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

在 Kotlin 中使用的关键词比 Java 对应的要多，这是很少见的一种情况：`enum class` 对应 Java 中的 `enum`。在 Kotlin 中，`enum` 又叫做软关键词(`soft keyword`)：当它出现在 `class` 之前时，它就有了特殊的含义。但是你可以在其他地方把它当做常规名字来使用。另一方面，`class` 依然是一个关键词。你依然需要把变量命名为（`class` 关键字以外的名字）`clazz` 或 `aClazz`。

像 Java 那样，枚举类型并不是一个值的列表：你可以在枚举类中声明属性和方法。以下展示了它是如何工作的：

```
enum class Color(
    val r: Int, val g: Int, val b: Int    // 1 声明枚举常量的属性
) {
    RED(255, 0, 0), ORANGE(255, 265, 0),    // 2 当每个常量被创建时指定属性值
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),

    INDIGO(75, 0, 130), VIOLET(238, 130, 238); // 3 分号 (;) 在这里是必须的

    fun rgb() = (r * 256 + g) * 256 + b    // 4 在枚举类中定义了一个方法
}
>>> println(Color.BLUE.rgb())
255
```

如你所见，枚举常量使用了跟你之前看到的在常规类中声明构造函数和属性的语法是一样的。当你声明每一个枚举常量时，你需要为常量提供属性值。注意，这个示例中展示了 Kotlin 语法中唯一一处需要你使用分号的地方：如果你在枚举类中定义了任何方法，（请使用）分号将枚举常量列表从函数定义中分隔开来。现在，让我们来看看一些酷的方式来处理代码中的枚举常量。

## 2.3.2 使用 `when` 来处理枚举类

你还记得小孩子是如何使用助记词来记忆彩虹的颜色的吗？这一个例子："Richard Of York Gave Battle In Vain!"。想象一下你需要一个为每种颜色给出一个助记词的函数（但是你不把这些信息存储在枚举列表中）。在 Java 中，你可以为此使用一个 `switch` 声明。

在 Kotlin 中对应的（语法）构造是 `when`。

跟 `if` 关键词类似，`when` 是一个返回值的表达式，因此你可以写一个有表达式主体的函数来直接返回 `when` 表达式。当我们在这一章的开头讨论函数时，我们允诺了会有一个带有多个表达式主体的多行函数。这就是那个的一个例子了：

```
fun getMnemonic(color: Color) =           // 1 直接返回一个when表达式
    when (color) {                         // 2 如果颜色等于枚举常量，返回对应的字符串
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
        Color.GREEN -> "Grave"
        Color.BLUE -> "Battle"
        Color.INDIGO -> "In"
        Color.VIOLET -> "Vain"
    }

>>> println(getMnemonic(Color.BLUE))
Battle
```

代码找出传递给 `color` 值对应的分支。不像 Java，你不需要再每一个分支里写 `break` 语句（在 Java 代码中，一个缺失的 `break` 经常导致bug）。如果匹配成功，只有对应的分支会被执行。

你也可以在一个分支中合并多个值，如果你用逗号将它们分隔开：

```
fun getWarmth(color: Color) = when(color) {
    Color.RED, Color.ORANGE, Color.YELLOW -> "warm"
    Color.GREEN -> "neutral"
    Color.BLUE, Color.INDIGO, Color.VIOLET -> "cold"
}

>>> println(getWarmth(Color.ORANGE))
warm
```

这些例子使用了枚举常量的完整名称，指定了 `Color` 枚举类名。你可以通过导入常量值来简化代码：

```
import ch02.colors.Color          // 1 导入在另一个包里声明的Color类
import ch02.colors.Color.*        // 2 显式地导入枚举常量，然后通过名字来使用它们

fun getWarmth(color: Color) = when(color) {

    RED, ORANGE, YELLOW -> "warm"    // 3 通过名字来使用常量
    GREEN -> "neutral"
    BLUE, INDIGO, VIOLET -> "cold"

}
```

在后续的例子中，我们将会使用简短的枚举名字，但是为了简单起见会忽略显式导入。

### 2.3.3 对任意对象使用 `when`

Kotlin 中的 `when` 构造比 Java 中的 `switch` 更为强大。跟要求你使用常量（枚举常量，字符串或者数字字面量）作为分支条件的 `switch` 不同，`when` 允许任意的对象。让我们写一个函数来混合两种颜色，如果它们能够在这个小调色板上能够被混合。你没有太多的选项，同时你也可以便捷的全部枚举它们：

```
fun mix(c1: Color, c2: Color) =
    when (setOf(c1, c2)) {           // 1 一个when表达式的参数可以是任意的对象。它
        // 检查分支的等价性。
        setOf(RED, YELLOW) -> ORANGE // 2 枚举颜色键值对可以是混合的
        setOf(YELLOW, BLUE) -> GREEN
        setOf(BLUE, VIOLET) -> INDIGO

        else -> throw Exception("Dirty color") // 3 如果没有一个分支被匹配，将执行该语句
    }

>>> println(mix(BLUE, YELLOW))
GREEN
```

如果颜色 `c1` 和 `c2` 是 `RED` 和 `YELLOW` (或者相反)，两者混合的结果是 `ORANGE`，以此类推。为了实现这个目的集合比较，你使用了。Kotlin 标准库内置了一个创建包含指定对象作为参数的一个 `setOf` 函数。一个 `set` 是一个元素顺序无关的集合。如果两个集合包含相同的元素，那么它们是等价的。因此，如果集合 `setOf(c1, c2)` 和 `setOf(RED, YELLOW)` 是等价的，这意味着无论 `c1` 是 `RED` 同时 `c2` 是 `YELLOW`，反之亦然。这恰恰就是你想要检查的。

`when` 表达式匹配它的参数直到分支条件是符合，而不是按顺序匹配所有的分支。所以，`setOf(c1, c2)` 检查等价性：首先是 `setOf(RED, YELLOW)`，然后才是其他颜色的集合，一个接着一个。如果没有其他分支条件符合，`else` 分支就会被执行。

允许使用任意的表达式作为一个 `when` 条件让你能够在许多场合下编写精简和出色的代码。在这个例子中，判断条件是一个等价性检查，接下来你将会看到条件是如何变为任意的布尔表达式的。

## 2.3.4 使用不带参数的 `when`

你可能已经注意到前一个例子（的实现）有些效率低下。你每次调用这个函数，它都会创建多个仅仅是用来检查给出的两个颜色是否匹配另外两个颜色的 `Set` 实例。一般情况下这不会有问题，但是如果这个函数是经常被调用的，通过其他方式来重写这段代码以避免创建内存垃圾是值得的。你可以通过使用不带参数的 `when` 语句来达到这个目的。这样代码可读性会降低，但这往往是你为了达到更好性能而必须付出的代价：

```
fun mixOptimized(c1: Color c2: Color) =

    when {      // 不带参数的when
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE

        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN

        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO

        else -> throw Exception("Dirty color")
    }

>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

如果 `when` 表达式没有使用任何参数，分支条件将会是任意的布尔表达式。`mixOptimized` 函数跟之前的 `mix` 函数做的是同样一件事。它的好处是不会创建额外的对象，但（相应的）代价是变得难以阅读。

让我们继续来看 `when` 结构在智能类型转换（**smart casts**）中发挥作用的例子。

## 2.3.5 智能类型转换：合并类型检查和转换

作为这个章节的一个例子，你将会写一个计算像 $(1 + 2) + 4$ 这样的简单数学表达式的函数。这个表达式将会包含一种类型操作：求两个数的和。其他的算术操作（减法、乘法、除法）也能够用相似的方式来实现。你也可以把它作为一个练习。

首先，你如何编码这个表达式？你把它存储在一个类似于树的结构中，其中的每一个节点是一个和（`Sum`）或者是一个数（`Num`）。`Num`一直是一个叶子节点，尽管`Sum`节点有两个孩子：`sum`操作的参数。接下来的代码片段展示了一个用来编码表达式的简单的类结构：一个叫做`Expr`的接口、实现这个接口的两个类`Num`和`Sum`。注意，`Expr`接口并没有声明任何方法。它被用做一个提供不同种类的表达式的公共类型接口记号。为了标记一个类是实现了某个接口的，你（应该）在接口名字的后面使用一个冒号（`:`）：

```
interface Expr
class Num(val value: Int) : Expr           // 1 带有一个属性、值而且实现了Expr接口的简单的值对象类

class Sum(val left: Expr, val right: Expr) : Expr    // 2 求和操作的参数可以是任意的Expr: Num对象或者其他的Sum对象
```

`Sum` 存储了 `Expr` 类型的 `left` 和 `right` 的引用。在这个小案例中，它们可以是 `Num` 或者是 `Sum`。为了存储之前提到的表达式  $(1 + 2) + 4$ ，你（应该）创建一个 `Sum(Sum(Num(1), Num(2)), Num(4))`。图2.4展示了它的树形表示。

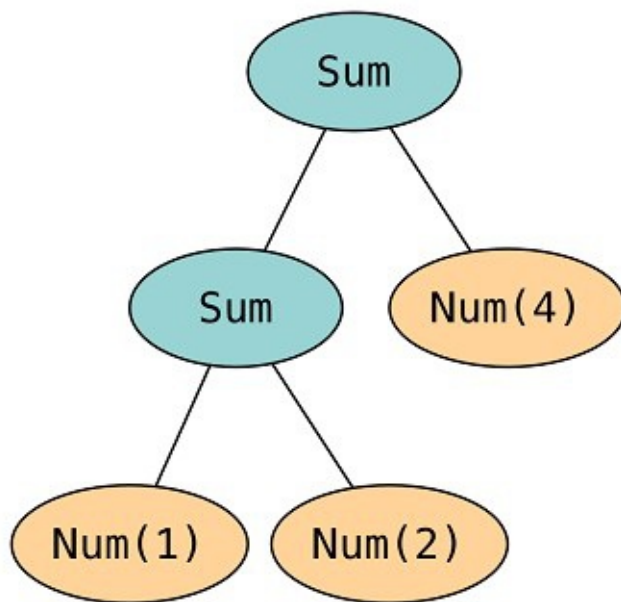


图2.4 `Sum(Sum(Num(1), Num(2)), Num(4))` 表达式的一个展现

让我们看看表达式是如何求值的。计算示例中的表达式应该返回7：

```
>>> eval(Sum(Sum(Num(1), Num(2)), Num(4)))
7
```

`Expr` 接口有两个实现，所以，为了计算一个表达式的值你必须尝试（以下）两种选项：

- 如果一个表达式是一个值，你必须返回对应的值
- 如果它是一个和，你必须由左到右计算表达式并返回它们的和

首先，我们将会着眼于用 Java 的常规方式来写的函数，然后我们会用 Kotlin 的方式来重构这份代码。在 Java 中，你可能已经实用一系列的 `if` 语句来检查选项，因此，让我们在 Kotlin 中使用同样的方法：

```
fun eval(e: Expr): Int {
    if (e is Num) {
        val n = e as Num          // 1 显式的Num类型转换是多余的
        return n.value
    }
    if (e is Sum) {
        return eval(e.right) + eval(e.left)    // 2 变量e是智能类型转换
    }
    throw IllegalArgumentException("Unknown expression")
}
>>> println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))
7
```

在 Kotlin 中，你通过 `is` 检查一个变量是否为一个指定的类型。如果你有 C# 的编程经验，你应该对这个标记很熟悉。`is` 检查跟 Java 中的 `instanceof` 很相似。但是在 Java 中，如果你已经检查一个变量为某个指定的类型，而且必须以指定的类型访问那个成员，那么你得在 `instanceof` 检查后面添加一个显式的类型转换。当初始变量使用超过一次，你往往会将类型转换后的结果保存到另一个变量中。在 Kotlin 中，编译器为你做了这份工作。如果你检查了变量的特定类型，后续你不需要执行类型转换。你可以把它当做你检查的目标类型来使用。实际上，编译器为你执行了类型转换，我们把这叫做智能类型转换(**smart cast**)。

在 `eval` 函数中，你检查变量 `e` 是否为 `Num` 类型之后，编译器会把这个变量当做一个 `Num` 的变量。之后你可以访问 `Num` 的 `value` 属性而不需要显式的类型转换：`e.value`。`Sum` 的 `right` 和 `left` 属性也是同样的道理：你只需要在相应的上下文中写 `e.right` 和 `e.left`。在 IDE 中，这些智能类型转换值会通过一个背景颜色来强调，如图 2.5 所示，因此很容易理解这个值是之前检查的那个。

```
if (e is Sum) {
    return eval(e.right) + eval(e.left)
}
```

图 2.5 IDE 用一个背景颜色对智能类型转换做了高亮处理

当且仅当一个变量在 `is` 检查之后不再改变时，智能类型转换才会起作用。当你对于一个带有属性的类使用智能类型转换时，正如（上面的）这个例子，属性必须是一个 `val`（不可变类型），同时它不能有自定义的访问器。否则，它不能验证每一个属性的访问是否会返回同样

的值。

一个指定类型的显式转换通过 `as` 关键词来表达的：

```
val n = e as Num
```

现在让我们来看看如何把一个 `eval` 函数重构成一个更符合 `Kotlin` 风格的函数。

## 2.3.6 重构：用 `when` 替换 `if`

`Kotlin` 中的 `if` 跟 `Java` 中的 `if` 有何不同？（事实上，）你已经见过（两者间的）差异了。在这一章的开头，你看到了 `if` 表达式用在 `Java` 中用三元操作符的上下文中: `if (a > b)`，就像 `Java` 中的 `a > b ? a : b`。 `Kotlin` 中并没有三元操作符，因为，与 `Java` 不同，`if` 表达式返回一个值。这意味着你可以重写 `eval` 函数来使用表达式主体语法、移除 `return` 语句和闭合的大括号，相反的，使用 `if` 表达式作为函数的主体：

```
fun eval(e: Expr): Int =
    if (e is Num) {
        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }
>>> println(eval(Sum(Num(1), Num(2))))
3
```

如果 `if` 分支里只有一个表达式，闭合的括号是可选的。如果 `if` 分支是一个代码块，最后的一句表达式作为结果返回。

让我们用 `when` 重写这份代码来进一步为它进行润色：

```
fun eval(e: Expr): Int =
    when (e) {
        is Num ->          // 1 用于检查参数类型的when分支
            e.value        // 2 这里使用了智能类型转换
        is Sum ->          // 1 用于检查参数类型的when分支
            eval(e.right) + eval(e.left) // 2 这里使用了智能类型转换
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```



`when` 表达式没有强制要求检查值的等价性。这是你早前看到的。在这里，你使用了一个不同形式的 `when` 分支，允许你检查 `when` 参数值的类型。正如先前的 `if` 例子，类型检查运用了一个智能类型转换，因此你可以访问 `Num` 和 `Sum` 的成员而无需额外的类型转换。

比较以上两个 Kotlin 版的 `eval` 函数，然后思考一下你怎样才能在你的代码中运用 `when` 替换一连串的 `if` 语句。当分支逻辑复杂时，你可以使用表达式块作为分支主体。让我们看看这是如何运作的。

## 2.3.7 代码块作为 `if` 和 `when` 的分支

`if` 和 `when` 都能够使用块作为分支。在这个例子中，代码块的最后一个表达式是结果。如果你想添加一些记录到示例函数中，你可以在返回最后一个值之前做这件事：

```
fun evalWithLogging(e: Expr): Int =
    when (e) {
        is Num -> {
            println("num: ${e.value}")
            e.value           // 1 这是代码块中的最后一个表达式。如果e是Num类型的它将会被返回。
        }
        is Sum -> {
            val left = evalWithLogging(e.left)
            val right = evalWithLogging(e.right)
            println("sum: $left + $right")

            left + right      // 2 如果e是Sum类型的， 这个表达式将会被返回
        }
        else -> throw IllegalArgumentException("Unknown expression")
    }
}
```

现在你能够看到通过 `evalWithLogging` 函数打印的日志和接着的计算顺序：

```
>>> println(evalWithLogging(Sum(Sum(Num(1), Num(2)), Num(4))))
num: 1
num: 2
sum: 1 + 2
num: 4
sum: 3 + 4
7
```

“代码中的最后一个表达式就是结果”这个规则在所有可以使用代码并需要一个（返回）结果的地方都是有效的。正如你将在这一章末尾看到的那样，同样的规则对 `try` 代码块 `catch` 从句有效。第5章讨论了 `lambda` 表达式中它的应用。但是正如我们已经在2.2.1章节中已经提到的那样，这个规则对于常规的函数是无效的。一个函数可以有一个不能为代码块的表达式主体，



或者一个内部有显式 `return` 语句的块主体。

你已经了解了如何用 `Kotlin` 的方式在众多事物中选择你想要的的东西。现在，是时候来看看你如何进行遍历的。

在这个章节讨论的所有特征中，迭代在 Kotlin 中可能是跟 Java 最相似的。while 循环跟 Java 中的那个一样。所以在这个章节的开头简要的提一下。for 循环只有一种形式，它等价于 Java 中的 for-each 循环。跟 C# 中的一样，写成 for <item> in <elements>。这种循环最常见的应用是遍历集合，正如 Java 中的一样，我们将会探索它是如何覆盖其他场景的循环的。

## 2.4.1 while 循环

Kotlin 有 while 和 do-while 循环，同时，它们的语法跟 Java 中对应的循环没什么不同：

```
while (condition) {           // 1 当while条件为真时，执行主体代码
    /*...*/
}

do {
    /*...*/
} while (condition)           // 2 第一次无条件的执行主体代码。在这之后，当条件为真时才执行。
```

Kotlin 并没有为这些简单的循环带来新的东西，所以我们不要在这消耗时间。让我们继续讨论 for 循环的各种用法吧！

## 2.4.2 遍历数字：范围和数列

正如我们所提到的那样，在 Kotlin 中并没有常规的 Java for 循环让你初始化一个变量，在循环的每一步更新它的值，当值到达一个特定的边界时退出循环。为了替代这样一个最常用的循环，Kotlin 使用了 ranges 的概念。

一个范围，在本质上就是两个值之间的一个间隔，通常是成员：一个起始值和一个结束值。你用 .. 操作符来写它：

```
val oneToTen = 1..10
```

注意：Kotlin 中的范围是闭合的或者说是包含的。这意味着第二个值也始终是范围的一部分。

对于整数范围你能做的最基本的是查看所有值。如果你能遍历范围内的所有值，这样的范围叫做数列(progression)。

让我们用整数范围来玩一个Fizz-Buzz游戏。这是一个打发旅途时间和回想起你已经遗忘的除法技能的好方法。玩家随着报数增加轮流玩。玩家用fizz一词替换可以被3整除的数，

用**buzz**一词替换可以被5整除的数字。如果一个数是3和5的乘积，你就说"FizzBuzz"。

下面的代码打印出从数字1到100的正确答案。注意你如何用不带参数的 `when` 表达式来检查可能的情况：

```
fun fizzBuzz(i: Int) = when {
    i % 15 == 0 -> "FizzBuzz " // 1 如果i能被15整除，返回FizzBuzz。就像在Java中，%是求模运算符

    i % 3 == 0 -> "Fizz "      // 2 如果i能被3整除，返回Fizz
    i % 5 == 0 -> "Buzz "      // 3 如果i能被5整除，返回Buzz
    else -> "$i "             // 4 否则返回这个数字的原始值
}
>>> for (i in 1..100) {      // 5 遍历整数返回1..100
...     print(fizzBuzz(i))
... }
1 2 Fizz 4 Buzz Fizz 7 ...
```

假设你在开车一个小时之后对这些规则厌倦了，想一点一点的完成这个游戏。让我们开始从100向后计数，而且只包含偶数：

```
>>> for (i in 100 downTo 1 step 2) {
...     print(fizzBuzz(i))
... }
Buzz 98 Fizz 94 92 FizzBuzz 88 ...
```

现在，你在遍历一个有步进值(**step**)的数列。它允许你跳过一些数字。这个步进值也可以是负的。这种情况下数列是向后遍历而不是向前遍历。在这里案例中，`100 downTo 1` 是一个向后遍历的数列（步进值是-1）。之后 `step` 变量把步进值改成了2同时保持遍历的方向不变（效果上是把步进值设为-2）。

正如我们之前提到的那样，`..` 语法始终产生一个包含终点的范围（`..` 右边的值）。在许多情况下，这让遍历不包含指定终点的半闭合范围更加方便。为了创建这样一个范围，（我们）使用 `util` 函数。举个例子，`for (x in 0 until size)` 循环等价于 `for (x in 0..size-1)`，但无论如何它表达的观点更加清晰。在后续的“使用对：插入调用和解构声明（Working with paris: infix calls and destructuring declarations）”章节，你将会在这些例子中学到更多关于 `downTo`，`step` 和 `until` 的语法。你可以看到如何使用范围和数列来帮助你应对FizzBuzz游戏的高级规则。现在让我们看看其他使用 `for` 循环的例子。

## 2.4.2 遍历映射集

我们已经提到过 `for .. in` 循环最常见的场景是遍历集合。这一点跟 Java 是完全一样的。因此我们不会说太多相关的东西。取而代之的是，让我们看看你能够如何遍历一个映射。

作为示例，我们将会看一些打印字符二进制值的小程序。你把这些二进制表示形式存储在一个映射里（仅仅是为了演示）。你创建一个映射，用一些字母的二进制值填满（这个映射集），然后打印这个映射集的内容：

```
val binaryReps = TreeMap<Char, String>() // 1 使用了TreeMap，因此键值是有序的

for (c in 'A'..'F') { // 2 使用字符范围，从A到F遍历字符
    val binary = Integer.toBinaryString(c.toInt()) // 3 把ASCII编码转换成二进制
    binaryReps[c] = binary // 4 以c为键把数值保存在映射集
}

for ((letter, binary) in binaryReps) { // 5 遍历一个映射集，把映射的键跟值分配
    给两个变量
    println("$letter = $binary")
}
```

`..` 语法创建的一个范围不仅对数字有效，对字符也是适用的。你在这里使用它来遍历从 A 到包含 F 的所有的字符。（上面的）示例展示了一个允许你对正在遍历的集合的元素进行拆箱（unpack）的 `for` 循环。你把拆箱后的结果存储在两个不同的变量中：`letter` 接收键，`binary` 接收值。在后续的析构声明和循环（**Destructuring declarations and loops**）一章，你将会找出更多关于这个拆箱（操作）的语法。

这个示例中用到的另一个新颖的技巧是通过键来获取和更新映射集中的值的简写语法。你可以使用 `map[key]` 来读取值并且通过 `map[key] = value` 来设置值，而不是调用 `get()` 和 `put()`。代码 `binaryReps[c] = binary` 等价于 Java 版中的 `binaryReps.put(c, binary)`。

它的输出跟下面的（示例）是类似的。我们把结果安排成两列而不是一列：

```
A = 1000001    D = 1000100
B = 1000010    E = 1000101
C = 1000011    F = 1000110
```

你可以在记录当前项（`item`）的索引的同时使用同样的拆箱语法来遍历集合。你不必创建一个单独的变量来保存索引和手动增加这个变量的值：

```
val list = arrayListOf("10", "11", "1001")
for ((index, element) in list.withIndex()) { // 通过索引遍历集合
    println("$index: $element")
}
```

（上面的）代码输出了你所期待的（结果）：

```
0: 10
1: 11
2: 1001
```

我们将会在下一章节深入探究 `withIndex` 的去向。

你已经看到你可以如何使用 `in` 关键字来遍历一个返回或者一个集合。你也可以使用 `in` 来检查一个值是否属于（某个）范围或集合。

## 2.4.4 使用 `in` 检查

你可以使用 `in` 操作符来检查一个值是否在某个范围内，或者相反的，`!in`（操作符）来检查一个值是否不再某个范围内。以下（演示了）你可以如何使用 `in` 来检查一个字符是否在某个字符范围内：

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'
>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

（用来）检查一个字符是否为字母的技术看起来很简单。掀开它的面纱，也没有什么花样：你依然是检查字符的编码是在第一个字母的编号与最后一个字母的标号之间的那个位置。但这个逻辑只是隐藏在了标准库中 `range` 类的实现中：

```
c in 'a'..'z' // 1 转换为 a <= c <= z
```

`in` 和 `!in` 操作符在 `when` 表达式中也是有效的：

```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!" // 1 判断值是否在0到9的范围内
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!" // 2 你可以合并多个范围
    else -> "I don't know..."
}
>>> println(recognize('8'))
It's a digit!
```

范围并没有局限于字符。如果你有任何支持（通过实现 `java.lang.Comparable` 接口的）实例比较算法的类，你可以创建一个这种类型的范围对象。如果你有这样一个范围，你不能枚举范围内的所有对象。想象一下这样的场景，举个例子，你能够枚举"Java"和"Kotlin"之间的所有字符串吗？不，你做不到。但是你依然可以使用 `in` 操作符来检查一个对象是否在这个范围内。

```
>>> println("Kotlin" in "Java".."Scala") // "Java".."Kotlin"和"Kotlin".."Scala"是一样的
true
```

注意字符串在这里是可以按字符顺序比较的，因为 `String` 类就是这样实现 `Comparable` 接口的。`in` 检查对集合也是有效的：

```
>>> println("Kotlin" in setOf("Java", "Scala")) // 这个集合并没有包含"Kotlin"字符串。在这个
章节的后续部分，你将会看到如何对你的数据类型使用范围和数列以及如何为那些对象使用in检查。
false
```

在这一章，我们想看到（这样）一组或者多组Java声明：处理异常的声明。

Kotlin中的异常处理跟Java和许多其他语言非常相似。一个函数能够按正常完成，否则当发生错误时抛出一个异常。函数调用者可以捕获这个异常并处理它。如果不这样做异常将会在未来的调用栈中重新抛出。在Kotlin中，异常处理声明的基本形式跟Java非常相似。你以一种不令人迷惑的方式抛出异常：

```
if (percentage !in 0..100) {  
    throw IllegalArgumentException(  
        "A percentage value must be between 0 and 100: $percentage")  
}
```

和其他类一样，你不必使用 `new` 关键词来创建一个异常的实例。跟Java不同，在Kotlin中 `throw` 语法是一个表达式，同时被用作其他表达式的一部分；

```
val percentage =  
    if (number in 0..100)  
        number  
    else  
        throw IllegalArgumentException( // 'throw' 是一个表达式  
            "A percentage value must be between 0 and 100: $number")
```

在这个示例中，如果满足条件，程序能够如期的运行，同时 `percentage` 变量被 `number` 变量初始化。否则，将会抛出一个异常。变量也不会被初始化。我们将会在后续的空类型（The nothing type）一章讨论有关 `throw` 作为另一个表达式的一部分的技术细节。

## 2.5.1 try, catch 和 finally

跟在Java中一样，你使用带有 `catch` 的 `try` 语法和 `finally` 从句来处理异常。你可以在接下来的示例中看到。从一个给定的文件中读取一行，（然后）尝试着去把读入的数据解析为一个数字，并返回数字，或者如果这一行不是有效的数字时返回 `null`：



```

fun readNumber(reader: BufferedReader): Int? { // 1 你不需要显式的声明这个函数会抛出哪些异常
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
    }
    catch (e: NumberFormatException) { // 2 异常的类型放在右边
        return null
    }
    finally { // 3 finally跟Java的中一样
        reader.close()
    }
}

>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239

```

(Kotlin)跟Java最大的不同就是 `throws` 从句并没有在代码中体现出来：如果你用Java写这个函数，你将在这个函数声明的后面显式的写上 `throws IOException`。你不需要这样做是因为 `IOException` 是一个已被检查的异常（*checked exception*）。在Java中，这是一个需要显式处理的异常。你必须声明你的函数可以抛出的所有已检查的异常。如果你是从另一个函数中调用达到，你也需要处理它的已检查的异常或者声明你的函数可以抛出这些异常。就像许多其他现代的JVM语言，Kotlin没有区分已检查和未检查的异常。你没有指定函数抛出的异常，你可能会，也可能不会处理任何异常。这个设计的决定是基于Java使用已检查异常的实践（经验）。经验表明，Java规则通常强制使用一大堆无意义的代码来重新抛出异常或者忽略异常，然而，这些规则并没有一致的使你避免可能发生的错误。例如，在我们刚刚看到的代码中，`NumberFormatException` 是一个没有被检查的异常。因此，Java编译器并不会强制你捕获这个异常，你很容易在运行时看到这个异常的发生。这是很不幸的（事情）。因为无效的输入数据是一种常见的情况，同时这种异常应该被优雅的处理。与此同时，`BufferedReader.close()` 方法可能会抛出一个未检查但是需要被处理的 `IOException`。如果一个流关闭失败了，大部分程序不会采取有意义的行动。所以代码强制要求在 `close()` 中函数捕获异常固定的套路。那对于Java 7的 `try-with-resources`（语法）呢？Kotlin并没有为此准备任何特殊的语法：它是通过一个库函数来实现的。在资源管理和λ（*lambdas for resource management*）一章，你将会看到为什么这是可行的。

## 2.5.2 try 作为一个表达式

为了了解Java和Kotlin之前的另一个明显的不同，让我们来对这个案例做一些小修改。让我们移除 `finally` 代码块（因为你已经明白了它是如何工作对），并添加一些代码来打印你从文件中读取的数字：

```

fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine()) // 1 变成了try表达式的值
    } catch (e: NumberFormatException) {
        return
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader) // 2 什么也不会打印

```

如你所见，Kotlin中的 `try` 关键字就像 `if` 和 `when` 一样，引入一个表达式，你也可以把它的值赋给某个变量。不同于 `if`（的地方），你必须把声明主体放在闭合的大括号里。和其他声明一样，如果主体包含多个表达式，`try` 表达式的值作为一个整体的值是最后一个表达式的值。在这个例子中，我们在 `catch` 块中放了一个 `return` 声明，所以在 `catch` 块后面方法不会继续执行。如果你想它继续执行，`catch` 从句也需要有一个值。而这个值将会是它里面最后一个表达式的值。以下就是它的工作原理：

```

fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine()) // 1 当没有异常发生时使用这个值
    } catch (e: NumberFormatException) {
        null // 2 当发生异常时使用null值
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
null // 3 抛出了一个异常，因此函数打印'null'

```

如果 `try` 代码块正常运行，`try` 代码块中的最后一个表达式就是结果。如果捕获了一个异常，`catch` 代码块中相应的最后一个表达式就是结果。在之前的一个例子，如果捕获了一个 `NumberFormatException`，那么返回值是 `null`。如果此时你已经不耐烦了，你可以开始与你在Java相似的方式来用Kotlin编程。如果你继续读这本书，你将会了解到如何改变你思考的习惯和使用强大的新语言。

- `fun` 关键词用来声明一个函数。`val` 和 `var` 关键词分别用来声明只读和不可变变量。
- 字符串模板帮助你避免繁琐的字符串拼接。在一个变量名之前加 `$` 前缀或者用 `${ }` 表达式包围来把它的值注入到字符串中。
- 值-对象类在Kotlin中以一种非常精简的方式表达。
- 熟悉的 `if` 现在是一个有返回值的表达式。
- `when` 表达式跟Java中的 `switch` 类似，但是更加强大。
- 你不需要在检查一个变量是否为某个类型之后进行显式的变量类型转换：编译器为你自动使用类型声明。
- `for`, `while` 和 `do-while` 循环跟Java非常相似，但是 `for` 循环现在（变得）更加方便，特别是当你需要遍历一个映射集或者一个有索引的集合时。
- 精简的 `1..5` 语法创建了一个集合。范围和数列允许Kotlin使用一个一致的语法和 `for` 循环中的抽象集合，同时，用来检查某个值是否属于一个范围的 `in` 和 `!in` 操作符也是有效的。
- Kotlin中的异常处理跟Java很相似。不同的地方是Kotlin并不要求你什么方法可能会抛出的异常。

这一章将会覆盖以下内容：

- 对集合、字符串和常规表达式有效的函数
- 使用命名参数，默认参数值和中缀调用语法
- 通过扩展函数和属性来适配Java库到Kotlin中
- 使用顶层代码和本地函数跟属性来结构化你的代码

你已经看到了当你把Java转化为Kotlin时，这些概念对你来说是多么熟悉。以及Kotlin是如何使得它们变得更加精简和可读。在这一章中，你将会看到Kotlin如何提升每个程序的关键要素的：声明和调用函数。我们也将会考虑通过使用扩展函数的方式来适配Java库成Kotlin风格。这将让你在混合语言项目中获得Kotlin的全部好处。为了让我们的讨论更加使用而不再变得抽象，我们将会专注于作为Kotlin集合、字符串和常规的表达式作为我们的问题讨论范围。作为一个介绍，我们先看看如何在Kotlin中创建集合。

在你能够用集合来做你感兴趣的事情之前，你需要学习如何创建它们。在 `when` 对象一节，你已经碰到过创建一个新集合的方法：`setOf` 函数。你创建一个颜色的集合，但是现在，让我们用数字来把它变得更加简单：

```
val set = setOf(1, 7, 53)
```

你以一种相似的方式创建一个列表或者一个映射：

```
val list = listOf(1, 7, 53)
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

注意，`to` 并不是特殊的构造，而是一个常规的函数。我们将会在后续的部分回到这个话题。你能够猜到这边（用来）创建的对象类是什么吗？运行下面的这个例子，你可以自己看看：

```
>>> println(set.javaClass) // 1 javaClass在Kotlin中等价于Java中getClass()
class java.util.HashSet
>>> println(list.javaClass)
class java.util.ArrayList
>>> println(map.javaClass)
class java.util.HashMap
```

如你所见，Kotlin使用了标准的Java集合类。这对Java开发者来说是好事情：Kotlin并没有它自己的集合类。你现存所有的关于Java集合的知识在这里都是有用的。为什么没有Kotlin集合呢？因为使用标准的Java集合让Kotlin更容易于Java代码进行交互。当你在Kotlin中调用Java函数时，你并不需要以某种方式转换集合，反之亦然。尽管Kotlin中的集合跟Java中的是完全一样的，但是你可以在Kotlin中用它做更多的事情。举个例子，你可以获得列表中最后一个元素或者找出一个数字集合中的最大值：

```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

在这一章，我们将会深入探讨集合是如何工作的以及所有的这些新方法是来自Java的那些类。在后续的章节，当我们开始讨论 $\lambda$ （表达式）时，你将会看到更多你能够用集合实现（的例子），但我们将依然会使用同样的标准Java集合类。在集合（collections）一节，你将了解到Java集合类是如何用Kotlin类型系统描述的。在讨论对Java集合起作用的神奇`last`和`max`函数之前，让我们了解一些用来声明函数的新概念吧！

现在，你已经知道如何创建带有元素的集合了。让我们直接（用它）来做些事情吧：打印它的内容。如果这看起来过于简单，请不要担心。与此同时，你将会遇到一大堆重要的概念。

Java集合有一个默认的 `toString` 实现。但它的输出格式是固定的，而且并不总是你需要的：

```
>>> val list = listOf(1, 2, 3)
>>> println(list) // 1 调用toString()
[1, 2, 3]
```

想象一下，你需要元素用分号隔开，同时被括号隔开而不是使用默认的实现（就像这样）：`(1; 2; 3)`。为了解决这个问题，Java项目使用了第三方的库，例如Guava和Apache Commons，或者在项目内部重新实现这个逻辑。在Kotlin中，这个函数是标准库的一部分。

在这一章节，你将会自己实现这个函数。你将会没有使用Kotlin语法来简化函数声明，而是以一个直接的实现为开始。同时你将会以一个更具Kotlin味道的方式来重写这份代码。

下面的 `joinToString` 函数把集合元素添加到了 `StringBuilder` 中，在它们之间使用分隔符，在开头使用一个前缀，在末尾添加一个后缀：

```
fun <T> joinToString(
    collection: Collection<T>,
    separator: String,
    prefix: String,
    postfix: String
): String {

    val result = StringBuilder(prefix)

    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator) // 1 在第一个元素之前不添加分隔符
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

这个函数是支持泛型的：它对包含任意类型的集合有效。如你所见，泛型语法跟Java很相似。（更多关于泛型的详细讨论将会成为第9章的主题。）

让我们刻意的使用一下这个函数以验证它（的结果）：

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(" , ")"))
(1; 2; 3)
```



这样的实现是很好的。你很可能留着它就这样子。我们将要集中讨论的是声明：你如何才能以更少的繁琐信息来调用这个函数呢？也许你能避免每次调用这个函数是都传递四个参数。让我们来看看你可以如何实现。

### 3.2.1 有名字的参数

我们即将解决的第一个问题关注的是函数调用的可读性。举个例子，看看接下来的函数调用 `joinToString()`：

```
joinToString(collection, " ", " ", ".")
```

你能够说出所有的这些字符串都对应什么样的参数吗？所有的元素都是用空白符或者逗号分隔开的吗？（如果）不看一眼这个函数的签名，这些问题很难回答。也许你记住它，或者你的IDE能够帮助你。但是调用代码却不明显（的告诉你函数签名）。

这个问题在有布尔标记位是特别的普遍。为了解决这个问题，一些Java编码风格推荐创建一个枚举类型而不是使用布尔类型。更有甚者强制你在注释中显式的指明参数名，正如 `String` 参数的案例：

```
/* Java */
joinToString(collection, /* separator */ " ", /* prefix */ " ",
/* postfix */ ".");
```

但是用Kotlin，你可以做得更好：

```
joinToString(collection, separator = " ", prefix = " ", postfix = ".")
```

当调用Kotlin编写的方法时，你看可以指定一些传递给函数的参数的名字，如果你在某个调用中指定了一个参数，你也应该为后续的所有参数指定名字来避免（造成）困惑。

#### TIP 提示

无需多言，IntelliJ IDEA能保持最新的够显式参数名，如果你重命名了被调函数的参数。你只需确保你使用了重命名或者改变签名动作而不是手动的编辑参数名。

#### WARNING 警告

不幸的是，当你调用Java编写的方法时，你不能使用命名参数，包括来自JDK的方法和Android框架的方法。当且仅当从Java8开始，在.class文件中存储参数名字成为一个可选的特性。Kotlin保持兼容Java6.结果是，编译器无法识别你的调用使用的参数名与方法定义相匹配。

我们将在下文看到，只有当参数带有默认值的时候命名参数才是有效的。

## 3.2.2 默认参数值

另一个常见的Java问题是在某些类中重载方法过多。仅 `java.lang.Thread` 就有八个构造函数！重载可以提供向后兼容，为API用户提供便捷，或者处于其他原因。但是最后的结果是相同的：重复。参数名和类型一次又一次的重复。即便你是一个好公民，你也必须在每一个重载中重复大部分的文档。与此同时，如果你调用了忽略某些参数的重载，那个参数使用了那个值变得不再一直清晰。

### 脚注7 <http://mng.bz/1vKt>

在Kotlin中，你通常可以避免创建重载，因为你可以在函数声明中指定参数的默认值。让我们使用这一特性来提升 `joinToString` 函数。对于大多数情况，字符串可以用逗号分隔而不需任何的前缀或者后缀。所以让我们用这些值当做默认值：

```
fun <T> joinToString(
    collection: Collection<T>,
    separator: String = ", ", // 1 默认的参数值
    prefix: String = "",
    postfix: String = ""
): String
```

现在你也可以使用所有的参数来调用这个函数或者忽略它们：

```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

当使用常规调用语法是，你可以只忽略后面的参数。如果你使用命名参数，你可以忽略列表中的一些（其他）参数而仅仅指定你需要的那些（参数）：

```
>>> joinToString(list, prefix = "# ")
# 1, 2, 3
```

**NOTE** 默认值和Java 考虑到Java并没有参数默认值的概念，当你在Java中调用带有参数默认值值的Kotlin函数时，你必须显式的指定所有参数的值。如果你经常需要从Java中调用一个函数并想让Java调用者易于使用，你可以用 `@JvmOverloads` 对它进行标注。它将指示编译器以从最后一个参数开始逐个忽略每一个参数的方式产生重载的Java函数。举个例子，如果你用 `@JvmOverloads` 标注了 `joinToString()`，以下的重载将会产生：

```
/* Java */
String joinToString(Collection<T> collection, String separator,
    String prefix, String postfix);
String joinToString(Collection<T> collection, String separator,
    String prefix);
String joinToString(Collection<T> collection, String separator);
String joinToString(Collection<T> collection);
```

每一个重载（函数）都为忽略了签名的参数使用了默认值。

到目前为止，你已经编写了你的工具函数而不需要过多的关注上下文。毫无疑问，它必须是某个类的方法，同时你已经忽略了周围的类声明，对吧？事实上，Kotlin使得它不再是必要的。

### 3.2.3 摆脱静态工具类：顶层(top-level)函数和属性

我们都知道Java作为一个面向对象语言，要求代码写成类的方法。一般来说，这种方式可以很好的工作。但事实上，几乎每一个大型的项目都会以带有大量不属于任何单个的类的职责不清的代码结束。有时候，某个操作对两个同样重要的类的对象都有效。有时候，某个对象是主要的，但是你不希望通过把操作添加为实例方法导致它的API变得臃肿。结果，你以类不包含任何状态或作为静态方法的容器的实例方法结束代码。一个完美的例子就是JDK中的 `Collections` 类。为了在你的代码中找出其他例子，查找以 `Util` 作为命名的一部分的类。在Kotlin中，你不需要创建所有这些无意义的类。相反，你可以把这些函数直接放在代码文件的最顶层而不需要在任何的类的内部。这样的函数依然是声明在文件顶部的包的成员。同时如果你想要从其他包里调用它们，你依然需要导入它们。但是不必要的额外的嵌套层级不复存在了。让我们把 `joinToString` 函数直接放进 `strings` 包内。用下面的内容创建一个叫 `join.kt` 的文件：

```
package strings
fun joinToString(...): String { ... }
```

这份代码又如何运行呢？你知道，当你编译文件时，将会产生一些类，因为JVM只能在类中执行的代码。当你只用Kotlin时，这就是你所需要知道的全部（知识）。但是，如果你需要从Java中调用这样的函数，你必须理解它是如何被编译的。为了理解的更清晰一点，

让我们来看看编译出同样的类的Java代码：

```
/* Java */
package strings;
public class JoinKt {    // 对应前一个例子的文件名，join.kt
    public static String joinToString(...) { ... }
}
```

你可以看到Kotlin编译器产生的类的名字跟包含函数的文件的名称一样。文件中所有的顶层函数都会被编译成这个类的静态方法。因此，从Java中调用这个方法跟调用其他静态方法一样简单：

```
/* Java */
import strings.JoinKt;
...
JoinKt.joinToString(list, ", ", "", "");
```

**SIDEBAR** 改变文件的类名 为了改变生成的包含Kotlin顶层函数的类的名字，你可以给文件添加 `@JvmName` 标注。把它放置在文件的开头位于包名之前的地方：

```
@file:JvmName("StringFunctions") // 1 指定类名的标注
package strings // 2 包声明跟在文件标注后面
fun joinToString(...): String { ... }
```

现在函数可以这样调用：

```
/* Java */
import strings.StringFunctions;
StringFunctions.joinToString(list, ", ", "", "");
```

跟标注语法有关的详细讨论将会在后续的第十章进行。

## 顶层属性（TOP-LEVEL PROPERTIES）

就像函数那样，属性可以放在文件的顶层。保存类外部的单独的数据块并不常用，但依然非常有用。举个例子，你可以使用 `var` 属性来计算某个操作被执行的次数：

```
var opCount = 0 // 1 包级别的属性声明
fun performOperation() {
    opCount++ // 2 改变属性的值
    // ...
}
fun reportOperationCount() {
    println("Operation performed $opCount times") // 3 读取属性的值
}
```

这样一个属性的值将会被存储在一个静态字段中。顶层属性（Top-level properties）也允许你在你的代码中定义常量：

```
val UNIX_LINE_SEPARATOR = "\n"
```

默认的，顶层属性就像其他属性一样，以访问器（一个 `val` 属性的读取函数或者一对读取函数/设置函数）的形式暴露给Java代码。如果你想把常量暴露给Java代码组委一个 `public static final` 字段来让它的用法更加自然，你可以用 `const` 修饰器标记它（原始类型属性是允许这样做的，`String` 类型也是）：

```
const val UNIX_LINE_SEPARATOR = "\n"
```

这将得到跟以下等价的Java代码：

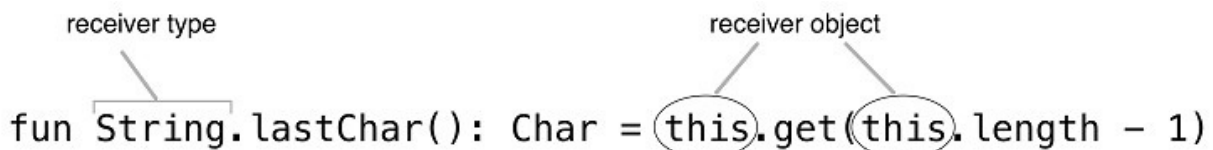
```
/* Java */
public static final String UNIX_LINE_SEPARATOR = "\n";
```

你已经把初始的工具函数 `joinToString` 提升了很多。现在让我们看看如何让它变得更加易于使用。

Kotlin的一大主题就是平滑集成现有的代码。即便纯Kotlin项目是建立在诸如JDK、Android框架和其他第三方框架等Java库之上的。当你把Kotlin整合到Java项目中去时，你也要处理已经或者未转换成Kotlin的现有代码。当使用这些API时可以使用全部Kotlin最有趣的特性而无需重新编写这些代码，这会是一件很好的事情吗？那就是扩展函数能让你做的事情。概念上讲，一个扩展函数是这样东西：它是一个可以作为一个类成员进行调用的函数，但是定义在这个类的外部。为了解释这个概念，让我们添加一个方法来计算一个字符串的最后一个字符：

```
package strings
fun String.lastChar(): Char = this.get(this.length - 1)
```

正如你所见，你所需要做的仅仅是在你添加的函数的名字之前放置你需要扩展的类或者接口的名字。这个类名叫做接收器类型（receiver type），而你调用的扩展函数的值叫做接收器对象（receiver object）。详细解释如图3.1：



```
fun String.lastChar(): Char = this.get(this.length - 1)
```

图3.1 接收器类型是扩展定义的类型，同时，接收器对象是这个类型的实例

你可以使用跟普通类成同样的语法来调用这个函数：

```
>>> println("Kotlin".lastChar())
n
```

在这个例子中，String 是接收器类型，同时"Kotlin"是接收器对象。在某种意义上，你已经添加了你的方法到String类。尽管String并不是你的代码的一部分，你可能甚至没有那个类的远吗，你任然可以用你的项目所需的方法来扩展它。甚至String是否用Java编写的也没无关紧要，Kotlin或者其他JVM语言，例如Groovy。只要它被编译成了Java类，你可以添加到你自己的扩展到那个类中。在一个扩展函数的主体中，你可以使用this（关键字）就像你在一个方法中使用它那样。作为一个常规方法，你可以忽略它：

```
package strings
fun String.lastChar(): Char = get(length - 1) // 1 "this"引用是隐式的
```

在这个扩展函数中，你可以直接访问你扩展的类的函数和属性，就像在定义在这个类中的方法那样。注意扩展函数并不允许你打破封装。跟定义在类中的方法不同，扩展函数并不能访问私有或保护访问属性的类成员。

### 3.3.1 导入和扩展函数

当你定义一个扩展函数是，它并不会自动在你的整个项目中变为可用。相反的，它需要被导入，就像其他的类或者函数那样。这有助于避免意外的命名冲突。**Kotlin**允许你使用你用于类的同样单独语法来导入单独的函数：

```
import strings.lastChar
val c = "Kotlin".lastChar()
```

当然，使用通配符（`*`）导入也是可以的：

```
import strings.*
val c = "Kotlin".lastChar()
```

你可以使用 `as` 关键词来该改变你所导入的类或者函数的名字：

```
import strings.lastChar as last
val c = "Kotlin".last()
```

当你在不同的包中有多个同名的函数并且你想在同一个文件中使用它们时，在导入中改变一个名字是非常有用的，对于常规的类和函数，你在这种情况下有另外的选择：你可以使用完全有效的名字来有用类或者函数。对于扩展函数，这个语法要求你使用缩写名字，因此一个导入声明中的 `as` 关键词是解决冲突的唯一方法。

### 3.3.2 从Java中调用扩展函数

调用一个扩展函数并没有涉及适配器对象的创建或者任何其他运行时开销。在底层方面，一个扩展函数是一个接受接收器对象作为第一个参数的静态方法。这让（我们）从**Java**中使用扩展函数变得非常容易：你调用静态方法并传递接收器对象实例。就像和其他顶层函数，包含这个方法的**java**类的名字由声明这个函数的文件的名字决定。我们可以把代码声明在一个 `StringUtil.kt` 文件中：

```
/* Java */
char c = StringUtilKt.lastChar("Java");
```



这个扩展函数被声明为顶层函数，因此它被编译为一个静态方法。你从Java中静态的可以导入 `lastChar` 方法，用法简化为 `lastChar`。比起Kotlin版本来，这份代码的可读性多少有点下降了，但是从Java的角度来说这是符合语言习惯的。

### 3.3.3

现在你可以编写最终版本的 `joinToString` 函数了。这几乎就是你将在Kotlin标准库中找到的（代码）：

```
fun <T> Collection<T>.joinToString( // 1 用Collection<T>声明一个扩展函数
    separator: String = ", ", // 2 为参数声明默认值
    prefix: String = "", // 2
    postfix: String = "" //2
): String {
    val result = StringBuilder(prefix)
    for ((index, element) in this.withIndex()) { // 3 "this"指向接收器对象:T类型的集合
        if (index > 0) result.append(separator)
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
```

你为一个集合声明了一个扩展，同时为所有的参数提供了默认值。现在你可以像类的一个成员那样调用 `joinToString`：

```
>>> val list = arrayListOf(1, 2, 3)
>>> println(list.joinToString(" "))
1 2 3
```

由于扩展函数只是静态函数调用的高效的语法糖，你可使用一个更加具体的类型作为一个接收器类型而不仅仅是一个类。比如说你想要一个仅仅能被字符串集合调用的 `join` 函数。用其他类型的一系列对象来调用这个函数是无效的：

```
fun Collection<String>.join(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = ""
) = joinToString(separator, prefix, postfix)

>>> println(listOf("one", "two", "eight").join(" "))
one two eight

>!!> listOf(1, 2, 8).join()
Error: Type mismatch: inferred type is List<Int> but Collection<String>
was expected.
```

扩展的静态类型也意味着扩展函数不能被子类覆盖（overridden）。让我们来看一个例子吧！

### 3.3.4 不可覆盖（overriding）的扩展函数

方法覆盖在Kotlin对平常的成员函数是有效的，但是你不能覆盖一个扩展函数。比如说你有两个类，`View` 和它的子类 `Button`，同时 `Button` 类覆盖了来自父类的 `click` 函数：

```
open class View {
    open fun click() = println("View clicked")
}
class Button: View() { // 1 Button类继承自View类
    override fun click() = println("Button clicked")
}
```

如果你声明了一个 `View` 类型的变量，你也可以在这个变量中存储 `Button` 类型的值，因为 `Button` 是 `View` 的子类。如果你用这个变量调用了一个常规方法，比如 `click()`，而且这个方法在 `Button` 类中被覆盖了。这个来自 `Button` 类的覆盖的实现将会被使用：

```
>>> val view: View = Button()
>>> view.click() // 1 Button类的示例的方法被调用。//AU:这个措辞正确吗？好像有点不对。TT
Button clicked
```

但是这种方式对于扩展来说是无效的，正如图3.2所示：

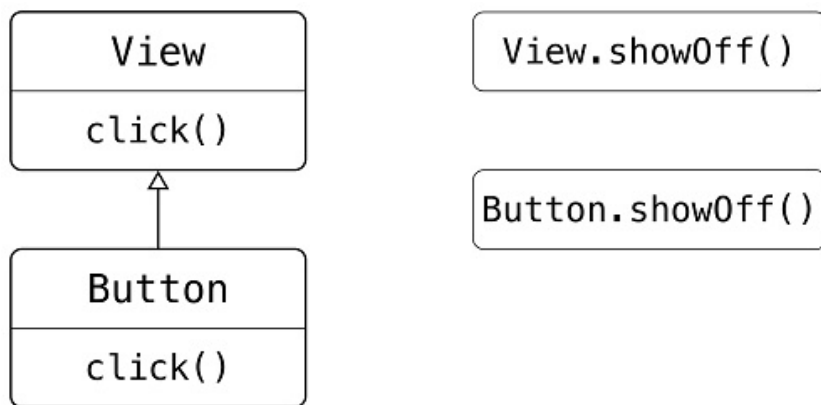


图3.2 扩展函数声明在类的外部

扩展函数并不是类的一部分。它们是声明在类的外部的。尽管你可以为某个基类和它的子类用同样的名字和参数类型来定义扩展函数，被调用的函数依赖于已被声明的静态类型的变量，而不是依赖于变量值的运行时类型。接下来的例子展示了两个声明在 `View` 和 `Button` 类的 `showOff` 扩展函数：

```

fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")

>>> val view: View = Button()
>>> view.showOff() // 1 扩展函数被静态的方式进行解析
I'm a view!
  
```

当你用一个 `View` 类型的变量调用 `showOff` 时，对应的扩展将会被调用，尽管这个值的真实类型是 `Button`。如果你重新调用了用Java编译成静态函数的扩展函数，这个行为对你来说应该是很清晰的，因为Java以同样的方式来选择函数：

```

/* Java */
>>> View view = new Button();
>>> ExtensionsKt.showOff(view); // 1 showOff函数被声明在extensions.kt文件中
I'm a view!
  
```

如你所见，覆盖对于扩展函数来说是不起作用的：Kotlin以静态解析它们。

注意 如果类有一个成员函数跟一个扩展函数有着相同的签名，成员函数总是优先的。当你扩展类的API时，你应该记住这一点：如果你添加了一个跟你已定义类的客户端（调用者）的扩展函数具有同样的签名成员函数，同时客户端随后重新编译了他们的代码，它将会改变它的含义并开始指向一个新的成员函数。

我们已经讨论了如何为外部的类提供额外的方法。现在让我们来看看如何对属性做同样的事。

### 3.3.5 扩展属性（Extension properties）

扩展属性提供了一种方法用能通过属性语法进行访问的API来扩展类。而不是函数的语法。尽管它们被叫做属性，它们不能拥有任何的状态：它不可能添加额外的字段到现有的Java对象实例。但是更简短的语法在某些时候任然是很方便的。在前一个章节，你定义了一个函数 `lastChar`。现在让我们把它转换成一个属性：

```
val String.lastChar: Char
    get() = get(length - 1)
```

你可以看到，就像使用函数一样，一个扩展属性看起来就像一个加了一个接收器类型的常规属性。访问器（getter）必须始终有定义，因为此处没有备用的字段，所以没有默认的属性实现。出于同样的原因初始化器也是不允许（存在）的：这里并没有任何地方来存储指定为初始化器的值。如果你用 `StringBuilder` 定义了同样的属性，你可以让它为 `var` 类型，因为 `StringBuilder` 的内容可以被修改：

```
var StringBuilder.lastChar: Char
    get() = get(length - 1) // 1 属性的访问器
    set(value: Char) { // 2 属性的设置器
        this.setCharAt(length - 1, value)
    }
```

你访问扩展属性（的方式）跟成员属性完全一样：

```
>>> println("Kotlin".lastChar)
n
>>> val sb = StringBuilder("Kotlin?")
>>> sb.lastChar = '!'
>>> println(sb)
Kotlin!
```

注意，当你需要从Java范文一个扩展属性时，你应该显式的调用它的访问器：`StringUtilKt.getLastChar("Java")`。我们已经在普遍意义上讨论了扩展的概念。现在让我们回到集合的主题并看一些有助于你运用它们的库函数，以及在这些函数中涉及的语言特性。

为了使用集合，这一章节展示了一些来自Kotlin标准库的函数。它自始至终描述了一些相关的特性：

- `vararg` 关键字，允许你声明一个带有任意个参数的函数
- 一个让你调用某些仅有一个参数的函数却无需八股代码的中缀标记
- 解构(Deconstructing)声明允许你把一个单独的组合值解压为多个变量

## 3.4.2 扩展Java的集合API

我们以Kotlin中的集合跟Java中的是同样的类但是多了一个扩展API这样的一个想法开始这一章。你看过了获取列表最后一个元素并查找一个数字集中的最大值的例子：

```
>>> val strings: List<String> = listOf("first", "second", "fourteenth")
>>> strings.last()
fourteenth
>>> val numbers: Collection<Int> = setOf(1, 14, 2)
>>> numbers.max()
14
```

我们对于它是如何工作的感兴趣：为什么在Kotlin中使用Java库的类来做如此多的事情是可能的。现在答案应该清晰了：`last` 和 `max` 函数被声明为扩展函数！`last` 函数并没有比 `String` 的 `lastChar` 函数跟复杂，在前一章节中已经讨论的：它是 `List` 类的扩展。对于 `max`，让我们来看一个简化的声明。真实的库函数不仅仅对 `Int` 数字有效，对任意的可比较元素也是有效的：

```
fun <T> List<T>.last(): T { /* returns the last element */ }
fun Collection<Int>.max(): Int { /* finding a maximum in a collection */ }
```

Kotlin标准库中声明许多的扩展函数，我们并不会在这里一一列出。你可能会想知道什么是了解Kotlin标准库的全部只是的最佳途径。在任何时候你都不需要这样做--（因为当）你需要用集合或其他对象做一些事情的时候，IDE的代码补全功能将会为你展示该类型的对象所有的可能的、可用的函数。（IDE提供的）列表包含了常规的方法和扩展函数，你可以选择你需要的函数。在这一章节的开头，你也看到了用来创建集合的函数。这些函数的一个共同的特性是它们可以带任意数量的参数进行调用。在下面的章节中，你将会看到声明这样的函数的语法。

## 3.4.2 可变参数（Varargs）：可以接受任意个参数的函数

当你调用一个函数来创建一个列表时，你可以向它传递任意数量的声明：

```
val list = listOf(2, 3, 5, 7, 11)
```

如果你在查找这个函数在库中是如何被声明的，你将会发现以下代码：

```
fun listOf<T>(vararg values: T): List<T> { ... }
```

你可能对Java的可变参数非常熟悉：一个允许你通过以打包成数组的方式传递任意数量的值给一个方法的特性。Kotlin的可变参数跟Java的很像，但是语法有些不同：不是在类型后面用三个点，而是在参数后面使用 `vararg` 修饰符。Kotlin和Java之间其他的不同点是当你需要传递的参数已经打包在一个数组里面时的函数调用语法。在Java中，你原样传递数组，然而Kotlin却要求你显式的对数组进行拆箱（*unpack the array*）。技术上讲，这个特性称做使用一个展开操作符（*spread operator*），但是在实践中，它跟在对应的参数前面放置 `*` 符号一样简单：

```
fun main(args: Array<String>) {
    val list = listOf("args: ", *args) // 1 展开操作符解压(unpack)数组内容
    println(list)
}
```

展开操作符允许你合并来自数组和一些单一调用中的固定值。Java并不支持（这种特性）。

现在让我们继续讨论映射。我们将会简要的讨论了一种提示Kotlin函数调用的可读性的方法：中缀调用（*infix call*）。

### 3.4.3 使用元组（**pairs**）：中缀调用和析构声明

为了创建映射，你使用 `mapOf()` 函数：

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

这是一个很好的时间点来提供了我们在这一章的开头为你承诺的另一种解释。这行代码中的 `to` 并不是内置的语法，相反的，而是一个方法的特殊调用，叫做中缀调用（*infix call*）。

在一个中缀调用，方法的名字被放在目标对象名和参数之间，而且没有其他的分隔符。下面的两个调用是等价的：

```
1.to("one") // 1 以常规的方式来调用
1 to "one" // 2 使用中缀标记来调用函数
```

中缀调用可以用于带一个参数的常规方法和扩展函数。为了使得函数可以使用中缀标记来调用，你需要用 `infix` 修饰符来标记它。这有一个 `to` 函数的简化版本的声明：

```
infix fun Any.to(other: Any) = Pair(this, other)
```

`to` 函数返回一个 `Pair` 实例。`Pair` 是Kotlin的一个标准库类，毫不意外的，它表示一对元素。`Pair` 和 `to` 的实际声明用的是泛型，但是为了简单起见，我们在此忽略它们。 注意，你可以为两个变量直接分配一对元素：

```
val (number, name) = 1 to "one"
```

这个特性叫做析构声明(*destructuring declaration*)。图3.3阐释了元素的析构声明是如何工作的。

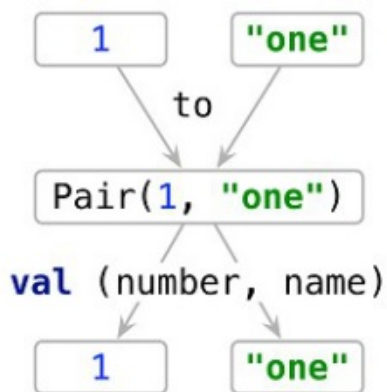


图3.3 你用 `to` 函数创建了一个元组并使用析构声明将它拆箱

析构声明这一特性并不局限于元组。举个例子，你也可以为两个独立的变量 `key` 和 `value` 分配一个映射集合。 这一特性对于循环也是有效的，正如你在 `joinToString()` 的实现中看到的，（它）使用了 `withIndex()` 函数：

```
for ((index, element) in collection.withIndex()) {
    println("$index: $element")
}
```



7.4节将会描述声明时候可以析构一个表达式并将其分配给多个变量。 `to` 函数是一个扩展函数。你可以创建任何元素的一个元组，这意味着它是一个泛型接收器的扩展：你可以写成 `1 to "one"` , `"one" to 1` , `list to list.size()` 等。 让我们来看看 `mapOf` 函数的声明：

```
fun <K, V> mapOf(vararg values: Pair<K, V>): Map<K, V>
```

像 `listOf`, `mapOf` （一类的函数）接受可变数量的参数，但这次参数应该是键值对。 尽管创建一个新的映射集合看起来是Kotlin中的一个特殊语法，但它是一个有着精简语法的常规函数。接下来，我们讨论扩展是如何简化字符串和正则表达式的处理的。

the last slash
the last dot

"/Users/yole/kotlin-book/chapter.adoc"

directory (before the last slash)
file name
extension (after the last dot)

Kotlin字符串跟Java字符串是完全一样的东西。你可以把Kotlin代码中创建的一个字符串传递给任意的Java方法，同时你也可以对你从Java代码中接收的字符串使用Kotlin标准库方法。（这）并不涉及任何的转换，同时也没有创建额外的包装对象。Kotlin通过提供一大堆有用的扩展函数来使得用Java标准库变得更为享受。（同时它）也隐藏了一些令人疑惑的方法，添加更加清晰的扩展。作为我们API差异的第一个例子，让我们来看看Kotlin如何处理拆分的字符串的。

### 3.5.1 拆分字符串

你可以对 `String` 类中的 `split` 方法非常熟悉。每个人都是使用它，但是仍然有人在[Stack Overflow](#)网站上抱怨它：“Java中的 `split` 方法对点号不起作用”。这是一个很常见的陷阱：写成 `"12.345-6.A".split(".")` 同时却期待得到 `[12, 345-6, A]` 数组这样一个结果。但是Java的 `split` 方法返回了空数组！这会发生，因为它把正则表达式当做一个参数，同时它依据（传入的）正则表达式将一个字符串查分成多个字符串。点号 `.` 在此处是一个指代任意字符的正则表达式。Kotlin隐藏了令人困惑的方法并提供了作为替代的多个有着不同参数的叫做 `split` 的扩展。把正则表达式作为参数的(`split` 重载扩展)把 `Regex` 类型的值当做参数而不是 `String` 类型。这确保了传递给函数的一个字符串无论是被解析成普通字符还是正则表达式，它始终是清晰的。

以下是你如何使用点号或者下划线来拆分字符串的：

```
>>> println("12.345-6.A".split("\\.|-".toRegex())) // 1 显式的创建一个正则表达式
[12, 345, 6, A]
```

Kotlin使用跟Java完全一样的正则表达式语法。此处的模式匹配了一个点号或者一个破折号（我们对它进行转义来表明我们想要的是一个字面量，而不是通配符）。使用正则表达式的API也跟Java标准库API相似，但是它们更加好用。举个例子，在Kotlin中，你使用一个扩展函数 `toRegex` 来把一个字符串转换成一个正则表达式。但是对于这样一个简单的例子，你不需要使用正则表达式。Kotlin中其他重载 `split` 的扩展函数接收任意数量的分隔符作为普通的字符串：

```
>>> println("12.345-6.A".split(".", "-")) // 1 指定多个分隔符
[12, 345, 6, A]
```

注意，你可以指定字符参数，写成：`"12.345-6.A".split('.', '-')`，这样得到的结果也是一样的。这个方法隐藏了Java中可以接受一个字符作为分隔符的相似的方法。

## 3.5.2 正则表达式和三引号字符串(triple-quoted strings)

让我们来看看另一个有两种不同实现的例子：第一个将使用扩展，第二个将会使用正则表达式。你的任务将会是吧一个文件的完整路径名解析成它的组件：一个路径、一个文件名、以及一个扩展名。Kotlin标准库包含了获取给定分隔符前后的子字符串的函数。以下是你可以使用它们来解决这个任务的方式（同时看图3.4）：

```
fun parsePath(path: String) {
    val directory = path.substringBeforeLast("/")
    val fullName = path.substringAfterLast("/")
    val fileName = fullName.substringBeforeLast(".")
    val extension = fullName.substringAfterLast(".")
    println("Dir: $directory, name: $fileName, ext: $extension")
}
>>> parsePath("/Users/yole/kotlin-book/chapter.adoc")
Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
```

the last slash      the last dot

`"/Users/yole/kotlin-book/chapter.adoc"`

directory (before the last slash)    file name    extension (after the last dot)

图3.4 通过使用 `substringBeforeLast` 和 `substringAfterLast` 函数把一个路径拆分成一个路径、一个文件名和一个文件扩展名。在文件路径的最后一个斜杠符之前的子字符串是一个闭合的目录路径。在最后一个点号之后的子字符串是一个文件扩展，同时，文件名在两者之间。如你所见，Kotlin使得不适用正则表达式重新排序的情况下解析字符串变得更加方便了。这是非常有用的。但是有时候写下的代码也非常难以理解。如果你想使用正则表达式，Kotlin标准库可以提供帮助。下面演示同样的任务如何使用正则表达式来完成的：

```
fun parsePathRegexp(path: String) {
    val regex = """.+/(.+)\.(.+)""".toRegex()
    val matchResult = regex.matchEntire(path)
    if (matchResult != null) {
        val (directory, filename, extension) = matchResult.destructured
        println("Dir: $directory, name: $filename, ext: $extension")
    }
}
```

在这个例子中，正则表达式被写在一个三引号字符串。在这样一个字符串中，你不需要转义任何字符，包括反斜杠。因此，你可以用 `\.` 来编码点号而不是你在普通字符串中写的 `\\.`（见图3.5）。

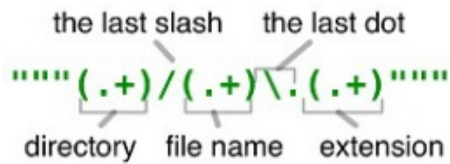


图 3.5 用来将路径分割成一个目录、一个文件名和一个文件扩展名的正则表达式

正则表达式通过斜杠和点号把一个路径分割成三组。`.` 模式匹配开头的任意字符，所以第一组 `(.+)`  包含了最后一个斜杠前的子字符串。这个子字符串包含了之前所有的斜杠，因为他们匹配了“任意字符”模式。相似的，第二组包含了最后一个点号之前的子字符串。第三组包含了其余所有的字符串。现在让我们来讨论前一个例子的 `parsePathRegexp` 函数的实现。你创建一个正则表达式并用它来匹配一个输入的路径。如果匹配结果为成功（不是 `null`），你分配它的 `destructured` 属性值分配给对应的变量。它使用的语法跟你分配一个元组给两个变量时一样：7.4节将会涵盖（一些）细节。

### 3.5.3 多行三引号字符串

三引号字符串的目的不仅仅是避免转义字符。这样的字符串字面量恶意包含任意的字符，包括换行符。这给了你一种便捷的方式在你的程序文本中嵌入包含换行符（的字符串）。正如这个例子，让我们来画一些艺术的ASCII字符：

```
val kotlinLogo = """| //
                    .|//
                    .|/ \"""
>>> println(kotlinLogo.trimMargin("."))
| //
|//
|/ \
```

多行字符串包含了三重引号之间所有的字符，包括用来格式化代码的缩进。如果你想要这种一个字符串的一个更好的表现形式，你可以裁剪缩进（换言之，就是左对齐）。为了达到这个效果，你添加一个前缀到字符串内容，来标记对齐的结束为止。然后调用 `trimMargin()` 来删除每一行前缀之前的文字。之前的例子使用点号作为前缀。尽管三重引号字符串包含了换行符，但是你不能使用像 `\n` 这样的特殊字符另一方面，你不需要转义 `\`，所以，

windows风格的路径 `"C:\\Users\\yole\\kotlin-book"` 可以被写成 `"""C:\Users\yole\kotlin-book"""`。你也可以在多行字符串中使用字符串模板。因为多行字符串并不支持转义（字符）序列，如果你需要在你的字符串中使用一个美元符号的字面量，你必须使用一个嵌套的表达式。比如：`val price = """${'$'}99.9"""`。在你的（除了使用ASCII艺术字的游戏以

外) 程序中, 多行字符串非常有用的一个方面是测试。在测试中, 处理多行文本并比较结果和预期输出是非常常见的操作。多行字符串给了你一个非常完美的解决方案来包含作为你的测试的一部分的预期输出。不需要笨拙的从外部文本进行加载或者转义--只需要添加一些引号标记并在引号之间放置需要的HTML文本或者其他输出。为了得到更好的格式, 使用前面提到的作为又一个扩展函数的例子 `trimMargin` 函数。

**NOTE**      注意    你现在可以看到, 扩展函数式扩展现有函数库的API并将它们融合到你的新语言-- `Pimp my Library` 模式的东西的语法习惯中去的一个很有力的方式。事实上, Kotlin标准库的很大一部分是由标准Java类库的扩展函数组成的。Anko库, 也是由JetBrains开发的, 提供扩展函数来让Android API更加Kotlin友好 (Kotlin-friendly) 的库。你也可以找到许多社区开发的, 用来为主要的第三方库的库, 比如Spring, 提供Kotlin友好的包装的库。

现在你可以看到Kotlin如何为你使用的库提供更好的API, 让我们的注意力回到你的代码中去。你将会看到扩展函数的一些新的用法。我们也将讨论一个新的概念: 本地函数 (local functions)。

许多开发者认为，优秀代码最重要的一个质量是没有冗余。这个原则还有一个别名：不要重复（Don't Repeat Yourself, DRY）。但是当你用Java写代码时，遵循这个原则并非总是无关紧要的。在许多场合，使用IDE的提取函数重构功能来讲冗长的函数分解为更小的代码块并重用它们是可能的。然而，这可能会导致代码变得难以理解，因为你可以走的更远一点并把提取出的方法分组到一个内部类中，这可以让你维护代码结构，但是这个方法会引入一定数量的八股代码。Kotlin给你一个更加清晰的解决方案：你可以在容器函数中嵌套你已经提取的函数。通过这种方式，你保留了你需要的结构而无需额外的代码。让我们来看看如何使用本地函数来解决相当常见的代码冗余问题。在下面的例子中，函数将用户信息保存到数据库中，你需要确保用户对象包含的是有效的数据：

```
class User(val id: Int, val name: String, val address: String)
fun saveUser(user: User) {
    if (user.name.isEmpty()) {
        throw IllegalArgumentException(
            "Cannot save user ${user.id}: Name is empty")
    }

    if (user.address.isEmpty()) { // 1 字段验证是冗余的（代码）
        throw IllegalArgumentException(
            "Cannot save user ${user.id}: Address is empty")
    }

    // Save user to the database
}
>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Cannot save user 1: Name is empty
```

这里的代码冗余比较小，你可能不想在你的类中到处都是个处理用户验证的特殊场景的方法。但是如果你把验证代码放到一个本地函数中，你可以避免冗余，同时保持清晰的代码结构。以下是它的原理：

```
class User(val id: Int, val name: String, val address: String)
    fun saveUser(user: User) {

        fun validate(user: User, // 1 声明一个本地函数来验证任意的字段
            value: String,
            fieldName: String) {
            if (value.isEmpty()) {
                throw IllegalArgumentException(
                    "Cannot save user ${user.id}: $fieldName is empty")
            }
        }

        validate(user, user.name, "Name") // 2 调用本地函数来验证特定的字段
        validate(user, user.address, "Address")

        // Save user to the database
    }
```

这样的代码看起来更加的好。验证逻辑没有重复，随着项目的发展，如果你需要添加其他字段到 `User` 对象，你可以方便的添加更多的验证代码。但是必须将 `User` 对象传递给验证函数是有点丑陋的。好消息是，这种做法是完全不必要的，因为本地函数已经访问了嵌套函数的所有参数和变量。让我们利用这个优势来避免额外的 `User` 参数：

```
class User(val id: Int, val name: String, val address: String)

    fun saveUser(user: User) {

        fun validate(value: String, fieldName: String) { // 1 现在你没有重复saveUser函数的“user”参数

            if (value.isEmpty()) {
                throw IllegalArgumentException(
                    "Can't save user ${user.id}: " + // 2 你可以直接访问外部函数的参数
                    "$fieldName is empty")
            }
        }

        validate(user.name, "Name")
        validate(user.address, "Address")

        // Save user to the database
    }

>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Cannot save user 1: Name is empty
```

为了进一步提升这个例子，你可以把验证逻辑移动到 `User` 类的扩展函数中：

```
class User(val id: Int, val name: String, val address: String)

fun User.validateBeforeSave() {
    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user $id: empty $fieldName") // 1 你可以直接访问User对象的属性
        }
    }

    validate(name, "Name")
    validate(address, "Address")
}

fun saveUser(user: User) {

    user.validateBeforeSave() // 2 调用扩展函数

    // Save user to the database
}
```

抽取一部分代码到一个扩展函数中，结果竟是如此的有用。尽管 `User` 是你的代码的一部分而不是一个库中的类，由于这跟任何其他使用 `User` 的地方不相关，你不想把这个逻辑放到 `User` 的方法中。因此，类的 API 仅仅包含了每个地方都需要用到的方法。所以类依然保持紧凑，同时很便于围绕你的想法（来展开）。另一方面，如前一个例子所示，函数主要处理一个单独的对象并不需要访问它的私有数据，可以访问它的成员却没有额外的限制。扩展函数也可以声明为本地函数，所以你可以进一步把 `User.validateBeforeSave()` 放在 `saveUser()` 中作为本地函数。但是多重嵌套的本地函数往往是难以阅读的。因此，原则上，我们不推荐使用超过一个层级以上的嵌套。看完函数中比较酷的特性，在下一章里，我们来看看你可以用类来做些什么。



- Kotlin并没有定义它自己的集合类，相反的，而是用更加丰富的API来扩展Java的集合类。
- 为函数参数定义默认值在极大程度上减少了定义重载函数的需求，同时命名参数语法使得有许多参数的函数变得更加可读。
- 函数和属性可以直接在文件中声明，而不是仅仅作为类的成员，这为代码结构提供了更多的灵活性。
- 扩展函数和属性允许你扩展任意类的API，包括定义在扩展库中的类，而无需修改它的源代码以及运行时消耗。
- 中缀调用为单一参数调用类操作符函数提供了一个清晰的语法。
- 三重引号字符串提供了一中清晰的方式来编写在Java中需要一大堆无聊的转移和字符拼接操作的表达式
- 本地函数有助于你清晰的组织代码并消除重复。

这一章涵盖了

- 类和接口
- 重要属性和构造函数
- 数据类和类委托
- 使用 `object` 关键词

这一章会在使用Kotlin的类方面给你带来更深入的理解。在第2章，你领略了声明一个类的基本语法。你了解了如何声明方法和属性、使用简单的主构造函数（它们是不是很好用？）以及使用枚举。但这里将会看到更多的干货。Kotlin的类和接口跟Java中对应的（概念）有点不同：比如，接口可以包含属性声明。跟Java不同，Kotlin的声明默认是 `final` 和 `public` 的。另外，嵌套类并不是默认在内部的：它们不包含（所属）外部类的隐式引用。在构造函数方面，简短的主构造器语法可以应对大多数情况，但是Kotlin也有完整的语法来让你用重要的初始化逻辑声明构造函数。对属性也是同样的道理：精简语法很好用，但是你也可以方便的重定义琐碎的访问器实现。Kotlin编译器可以生成有用的方法来避免琐碎的信息。把一个类声明为 `data` 类将指示编译器为这个类生成多个标准的方法。你也可以避免手动声明方法，因为Kotlin原生支持委托模式。这个章节也描述了一个声明类并创建类实例的新的 `object` 关键字。这个关键字用来表示单例对象(singleton objects)、伴生对象（companion objects）和对象表达式(类似于Java的匿名类)。让我们以讨论类和接口以及副标题Kotlin中的类层级定义作为开篇吧！

## 4.1 定义类层级

作为Java的对比,这一节讨论Kotlin中类层级的定义。我们将会着眼于Kotlin中跟Java类似但有些不同的默认可见性和访问修饰符。你也将了解到新的 `sealed` 修饰符。它将限制创建子类的可能性。

### 4.1.1 Kotlin中的接口：带有默认实现的方法

我们将会以接口的定义和实现为开篇。Kotlin接口跟Java 8中的（概念）相似：它们可以包含抽象方法的定义和非抽象方法的实现（类似于Java 8的默认方法），但是它们不能包含任何的状态。为了在Kotlin中声明一个接口，请使用 `interface` 关键字而不是 `class`：

```
interface Clickable {  
    fun click()  
}
```

这将声明一个只带有一个叫做 `click()` 抽象方法的接口。所有实现了这个接口的非抽象类都要提供这个方法的一个实现。以下是你如何实现接口的（演示）：

```
class Button : Clickable {  
    override fun click() = println("I was clicked")  
}
```

Kotlin在类名后面使用一个冒号来替代Java中的 `extends` 和 `implements` 关键词。

`override` 修饰符跟Java中的 `@Override` 标注类似，是用来标记覆盖来自父类或接口的方法和属性的。跟Java不同的是，Kotlin强制使用 `override` 修饰。如果某个方法是在你编写完你的实现以后添加的，这能防止你意外的覆盖了它。你的代码无法编译，除非你显式的将方法标记为 `override` 或者将其重新命名。一个接口方法可以有一个默认的实现。不同于Java 8，要求你用 `default` 关键字标记这样的实现，Kotlin没有用于这些方法的特殊标注：你只需要提供一个方法的主体。让我通过添加一个有默认实现的方法来改变 `Clickable` 接口。

```
interface Clickable {  
    fun click() // 1 常规的方法声明  
    fun showOff() = println("I'm clickable!") // 2 带有默认实现的方法  
}
```

如果你实现了这个接口，你需要为 `click()` 提供一个实现。你可以重写定义 `showOff()` 方法的行为，或者如果你对默认的实现不感冒也可以忽略它。 我们现在假定另一个接口也定义了一个 `showOff` 方法并为它添加了以下的实现：

```
interface Focusable {
    fun setFocus(b: Boolean) =
        println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm focusable!")
}
```

如果你需要在你的类中同时实现这两个接口，会发生什么呢？两个接口都包含带有默认实现的 `showOff` 方法，哪一个实现会胜出呢？两个都不会。相反的，如果你没有显式的实现 `showOff`，你会得到下面的编译错误：

**NOTE** `Button` 类必须覆盖 `public open fun showOff()`，因为它继承了多个 `showOff` 实现。

Kotlin编译器强制你提供自己的实现：

```
class Button : Clickable, Focusable {
    override fun click() = println("I was clicked")

    override fun showOff() { // 1 如果继承的成员（函数）有一个以上的实现，你必须提供一个显式的实现。
        super<Clickable>.showOff() // 2 “super”限制于尖括号中指定的超类名。这是你想调用的。
        super<Focusable>.showOff()
    }
}
```

`Button` 类现在实现了两个接口。通过调用你从超类继承的两个实现，你使 `showOff()` 生效了。为了调用一个继承的实现，你使用跟Java中相同的关键字：`super`。但是选取特定实现的语法是不同的。在Java中，你可以把基类的名字放在 `super` 关键字之前，就像在 `Clickable.super.showOff()` 方法中那样。然而，在Kotlin中，你把基类名放在尖括号中：`super<Clickable>.showOff()`。如果你只需要调用一个继承的实现，你可以写成这样：

**NOTE** `override fun showOff() = super<Clickable>.showOff()`

你可创建这个类的一个实例同时验证：所有继承的方法都可以被调用：

```
fun main(args: Array<String>) {  
    val button = Button()  
    button.showOff() // 1 我是可调用的！我是可获取焦点的！  
    button.setFocus(true) // 2 我获取到焦点了。  
    button.click() // 3 我被点击了  
}
```

`setFocus` 的实现声明在 `Focusable` 接口中，同时在 `Button` 类中自动被继承。

SIDEBAR 在Java中实现有方法体的接口 Kotlin 1.0把不支持接口默认方法的Java 6 作为设计目标。因此，它为常规接口与包含了静态方法的类的组合用默认方法编译了每一个接口。接口只包含了声明，同时类包含了所有作为静态方法的实现。所以，如果你需要在Java类中实现这样一个接口，你必须自定义所有方法的实现，包括Kotlin中有方法体的。Kotlin的未来版本将会支持生成Java 8字节码作为可选项。如果你选择Java 8作为目标平台，Kotlin的方法体将会编译成默认函数，同时你也可以在Java中实现这样一个接口而无需提供这个方法的实现。

现在，你已经看到了Kotlin是如何允许你实现接口中定义的方法的。让我们来看看这个故事的第二部分：覆盖基类中定义的成员。

## 4.1.2 `open` 、 `final` 和 `abstract` 修饰符：`final` 是默认的

如你所知，Java允许你创建任意一个类的子类，并覆盖任意的的方法，除非它已经被显式的用 `final` 关键字标记了。这往往是很方便的，但也有问题。当由于基类的代码改变却没有匹配子类的假设时，基类的修改会导致子类的不正确行为，所谓的脆弱的基类问题就会发生。如果类没有为子类提供精确的规则（那个方法支持覆盖以及如何覆盖），客户端就会面临没有按照基类作者的预期的方式来覆盖方法的风险。由于不可能分析到所有的子类，基类在某种程度上来说是脆弱的，任意的改变都可能导致子类无法预期的行为改变。为了避免这个问题，Java优良编程风格方面最出名的一本书，Joshua Bloch写的《Effective Java》推荐你：“设计并为继承编写文档或者干脆禁止”。这就意味着所有的类和方法并没有特意声明为在子类中可被覆盖的，应该显式的标记为 `final` 。Kotlin遵循同样的哲学。然而Java的类和方法默认是公开的，Kotlin的类和方法默认是 `final` 的。如果你想允许一个类创建子类，你需要用 `open` 修饰符标记这个类。另外，你需要添加 `open` 修饰符到每个可以被覆盖的属性或者方法。

```
open class RichButton : Clickable { // 1 这个类是开放的：其他类可以继承它。

    fun disable() {} // 2 这个类是*不可修改的（final）*：你不可以子类中覆盖它。

    open fun animate() {} // 3 这个函数是开放的：你可以在子类中覆盖它。

    override fun click() {} // 4 这个函数覆盖了一个开放的函数同时它也是开放的。
}
```

注意，如果你覆盖了一个基类或者接口的成员，覆盖的方法将会是默认 `open` 的。如果你想改变这个可继承性并禁止你的类的子类覆盖你的实现，你可以显式的把覆盖后的成员标记为 `final`：

```
open class RichButton : Clickable {
    final override fun click() {} // 1 这里的`final`并不是多余的，因为没有`final`修饰的覆盖将会是开放的。
}
```

**SIDEBAR** 开放类和智能类型转换 类默认为 `final` 的一个很明显的好处是在更广泛的场景开启智能转型。正如我们在 `when` 智能类型转换一节中提到的，一个智能转型仅能被用于 `val` 修饰同时没有自定义访问器的类属性这个要求意味着属性必须是 `final` 型的，因为不这样做的话，一个子类可以重写(`override`)属性并定义一个自定义的访问器，（这就）破坏了智能类型转换的要求了。由于属性默认是 `final` 的，你可以无需考虑并对大部分属性使用智能类型转换。这提升了你的代码的表达能力。 Kotlin中，跟Java一样，你可以把一个类声明为 `abstract`，同时这个类不能被初始化。一个抽象类通常包含抽象成员，其不需要具体实现但必须在子类中重写。抽象成员总是开放的，因此你不需要显式的使用 `open` 修饰符，这有一个例子：

```
abstract class Animated { // 1 这个类是抽象的：你不能创建它的实例

    abstract fun animate() // 2 这个函数是抽象的：它并没有一个实现同时必须在子类中被覆盖。

    open fun stopAnimating() { // 3 抽象类中的非抽象方法默认是不开放的，但是看可以被标记为开放的。
    }

    fun animateTwice() { // 3 抽象类中的非抽象方法默认是不开放的，但是看可以被标记为开放的。
    }
}
```

表格 4.1 列出了Kotlin中的访问修饰符

表格 4.1 类中的访问修饰符的含义

修饰符	对应的成员	备注
<code>final</code>	不能被覆盖	类成员的默认修饰符
<code>open</code>	可以被覆盖	必须显式的指定
<code>abstract</code>	必须被覆盖	只能在抽象类中使用，抽象成员不能有实现
<code>override</code>	在一个子类中覆盖一个成员	如果没有被标记为 <code>final</code> ，覆盖的成员默认是开放的。

表中的备注可用于类中的修饰符。在接口中，你不能使用 `final`、`open` 或者 `abstract`。接口中的成员总是为 `open` 属性。你不能将其声明为 `final`。如果接口没有代码体，那么它为 `abstract`，但是关键词并不是必须的。

**TIP Tip** 就像跟Java中一样，一个类可以实现很多的接口。但是只能继承一个类。

讨论了控制继承的修饰符，让我们现在继续另一种类型的修饰符的讨论：可见性修饰符。

### 4.1.3 可见性修饰符：默认是公开的

可见性修饰符有助于限制访问你的代码中的声明。通过限制类实现细节的可见性，你可以确保你改变实现细节但不会有破坏依赖代码的风险。基本上，Kotlin中的可见性修饰符跟Java中的很相似。你会遇到同样的 `public`、`protected` 和 `private` 修饰符。但是默认的可见性是不同的：如果你省略了修饰符，（默认的）声明将会是 `public`。Java中的默认可见性 `package-private` 并不会在Kotlin中出现。Kotlin把包仅仅作为命名空间中的代码的一种组织方式，并没有用于可见性控制。作为一个可选方案，Kotlin提供了一个新的可见性修饰符：`internal`，它意味着'模块内可见'。一个模块是一组Kotlin文件编译在一起组成的。它可以是一个IntelliJ IDEA模块、一个Eclipse项目、一个Maven或者Gradle项目，又或者Ant任务调用所编译的一组文件。`internal` 可见性的优势是它为你的模块的实现细节提供了实际的封装。使用Java，封装性很容易被破坏。因为外部代码可以在你所使用的同一个包内定义类，并由此获得包内声明的访问权。更多跟Java的差异来自Kotlin在类的外部定义函数和属性的能力。你可以把这样的声明标记为 `private`。这意味着“在包含文件内部可见”。如果一个类应该仅在一个文件中使用，你也可以让它私有。表格4.2列出了所有的可见性修饰符。

表4.2 Kotlin可见性修饰符

修饰符	对应的成员	顶层声明
<code>public</code> (默认可见性)	所有地方可见	所有地方可见
<code>internal</code>	模块内可见	模块内可见
<code>protected</code>	子类内部可见	不可见
<code>private</code>	类内部可见	在文件中可见

让我们来看一个例子。`giveSpeech` 函数的每一行都在尝试着违反可见性的规则，结果编译出错：

```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}

fun TalkativeButton.giveSpeech() { // 1 错误：'public' 成员暴露了它的 'internal' 接收器类型TalkativeButton

    yell() // 2 错误：无法访问'yell'：它在'TalkativeButton'内部是'protected'

    whisper() // 3 错误：无法访问'whisper'：它在'TalkativeButton'内部是'private'

}
```

译注：此处 `yell` 的修饰符是 `'private'`，`whisper` 的修饰符是 `'protected'`。注释与代码不符。已经提出[勘误](#)

Kotlin 禁止你从 `public` 访问性的 `giveSpeech` 函数引用低可见性类型的 `TalkativeButton` (在这个例子中是 `internal`)。这是一个有关要求基础类型列表使用的所有类型和类的类型参数，或者函数签名都要跟类或者方法本身的可见性一致的通用规则的例子。这一规则确保了你可以访问各种你可能需要调用的函数或者扩展的一个类。为了解决这个问题，你也可以让函数为 `internal` 或者让类为 `public`。注意Java和Kotlin的 `protected` 修饰符的表现差异。在Java中，你可以从同一个包中访问一个 `protected` 成员。但是Kotlin并不允许这样做。在Kotlin中，可见性规则是简单的。一个 `protected` 成员仅在类及其子类中可见。也要注意类的扩展函数并不能访问它的私有和保护成员。



**SIDEBAR**

**Java中的可见性** 当被编译为Java字节码时，Kotlin中的 `public`, `protected` 和 `private` 修饰符是保留的。你在Java代码中使用这样的Kotlin声明，好像它们用Java中相同的可见性声明的那样。唯一例外的是 `private` 了：名义上它被编译为 `package-private` 声明（在Java中你不能让一个类私有）。但是，你可能会问，`internal` 修饰符会发生什么？Java中没有直接的类似概念。`package-private` 访问性是完全不同的东西：一个模块通常有多个包组成。不同模块可能包含来自同一个包的声明。因此 `internal` 修饰符在字节码中变成了 `public`。Kotlin声明和Java对应物之间的一致性解释了为什么有时候你可以从Java代码访问却不能从Kotlin中访问。举个例子，你可以从另一个模块的Java代码中访问一个 `internal` 类或者顶层声明。或者从同一个包中的Java代码总访问保护成员（跟你在Java中的方式相似）。但要注意，类的 `internal` 成员的名字是不完整的。技术上讲，`internal` 成员可以用于Java，但是他们在Java代码中看上去很丑陋。当你扩展来自另一个模块的类时，这有助于避免在覆盖时发生意外的冲突。同时它也能阻止你意外的使用内部类。

Kotlin和Java之间的可见性规则的更多不同之处在于，Kotlin中，外部类无法看到内部（或嵌套）类的成员。让我们接下来以一个例子来讨论Kotlin中的内部类和嵌套类。

## 4.1.4 内部类和嵌套类：默认为嵌套

跟Java一样，在Kotlin中你可以在一个类中声明另一个类。这样做对于封装一个助手类或把代码放在更接近于使用它的地方是很有用的。不同之处在于，Kotlin嵌套类并没有访问外部类实例，除非你特别需要它。我们看一个例子来了解为什么特性这是重要的。想象一下，你想要定义一个 `View` 元素，而它的状态是可以被序列化的。序列化一个视图并不容易，但是你可以复制所有的必要数据到另一个助手类。你声明实现了 `Serializable` 接口的 `State` 接口。`View` 接口声明了可以被用于保存一个视图状态的 `getCurrentState` 和 `restoreState` 方法：

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}
```

在 `Button` 类中定义一个保存按钮状态的类是非常方便的。让我们来看看在Java中可以怎么做（稍后展示类似的Kotlin代码）：

```

/* Java */
public class Button implements View {
    @Override
    public State getCurrentState() {
        return new ButtonState();
    }

    @Override
    public void restoreState(State state) { /*...*/ }
    public class ButtonState implements State { /*...*/ }
}

```

你定义了一个实现了 `State` 接口并为 `Button` 类保存特定信息的 `ButtonState` 类。在 `getCurrentState` 方法中，你创建了这个类的一个新的实例。在一个真实的案例中，你已经用所有的必要数据初始化了 `ButtonState`。这份代码有什么错误呢？为什么当你尝试着去序列化声明在按钮中的状态是，你得到一个 `java.io.NotSerializableException: Button` 异常？第一眼看起很奇怪：你序列化的变量是一个 `ButtonState` 类型的 `state`，并不是 `Button` 类型。当你在Java中调用它时，一切都变得清晰了。当你把一个类声明在另一个类里面时，它默认是变为内部类。例子中的 `ButtonState` 类隐式保存了它的外部 `Button` 类的应用。这解释了为什么 `ButtonState` 不能被序列化：`Button` 不能被序列化，同时指向它的应用破坏了 `ButtonState` 的序列化。为了解决这个问题，你需要把 `ButtonState` 类声明为 `static`。把一个嵌套类声明为静态移除了来自外部类的隐式引用。在Kotlin中，内部类的默认行为跟我们刚描述的相反：

```

class Button : View {
    override fun getCurrentState(): State = ButtonState()

    override fun restoreState(state: State) { /*...*/ }

    class ButtonState : State { /*...*/ } // 1 这个类是Java中静态嵌套类的对等物
}

```

Kotlin中，没有显式修饰符的嵌套类跟Java静态嵌套类是一样的。为了把它变成一个内部类，它包含了一个外部类的引用。你使用 `inner` 修饰符。表格4.3描述了这种行为在Java和Kotlin之间的差别。图4.1描述了嵌套类和内部类之间的差别。

表格4.3 Java和Kotlin中嵌套类和内部类的对比

在另一个类B中什么类A	Java	Kotlin
嵌套类（不保存外部类的引用）	静态类A	类A
内部类（保存外部类的引用）	类A	内部类A

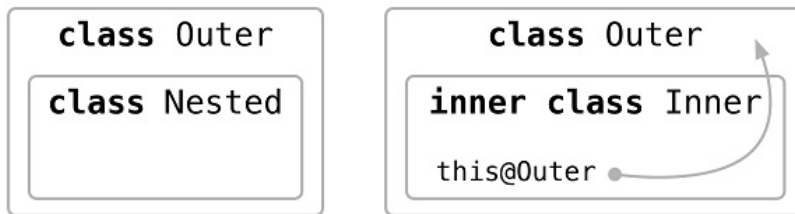


图4.1 嵌套类不会引用它们的外部类，然而内部类会这样做

Kotlin引用外部类实例的语法也跟Java不同。你写 `this@Outer` 来访问 `Inner` 类的 `Outer` 类。

```
class Outer {
    inner class Inner {
        fun getOuterReference(): Outer = this@Outer
    }
}
```

你已经了解到了内部类和嵌套类在Java和Kotlin之间的差别。现在让我们讨论嵌套类在Kotlin可能有用的另一个用例：创建一个包含有限个类的继承层级。

## 4.1.5 密封类：定义受限的类层级

回想一下 `when` 智能类型转换一节得到表达式层级示例。超类 `Expr` 有两个子类：`Num`，表示一个数；`Sum`，表示两个表达式的和。处理 `when` 表达式中所有可能的子类是非常方便的。但是你必须提供 `else` 分支来指定如果没有一个别的分支匹配上时会发生什么：

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)

        else -> // 1 你必须检查else分支
            throw IllegalArgumentException("Unknown expression")
    }
```

当你使用`when`语法计算一个表达式的时候，Kotlin编译器强迫你检查默认选项。在这个例子中，你没能返回有意义的东西，所以你抛出了一个异常。一直要添加默认分支是不方便的。更重要的是，如果你添加了一个新的子类，编译器不会检测到哪些东西发生了改变。

如果你忘了添加一个新的分支，默认分支就会被选中，而这可能会造成一些微妙的bug。

Kotlin为这个问题提供了一个解决方案：密封(sealed)类。你使用sealed修饰符标记一个超类。这限制了创建子类的可能性。所有的直接子类都必须被嵌套在超类中：

```
sealed class Expr { // 1 把一个基类标记为密封类

    class Num(val value: Int) : Expr() // 2 列出所有可能的子类作为嵌套类
    class Sum(val left: Expr, val right: Expr) : Expr()
}

fun eval(e: Expr): Int =

    when (e) { // 3 "when"表达式覆盖了所有可能的情况，所以不需要"else"分支
        is Expr.Num -> e.value
        is Expr.Sum -> eval(e.right) + eval(e.left)
    }
```

如果你在一个 when 语句中处理一个密封类的所有子类，你不需要提供默认分支。注意 sealed 修饰符意味着类是开放的。你不需要一个显式的 open 修饰符。图4.2解释了密封类的行为。



图4.2 密封类不能有定义在外部的继承者

事实上，Expr 类有一个只能在类内部被调用的私有构造函数。你不能声明一个密封接口。为什么呢？如果你可以（这样做）的话，Kotlin编译器不能够保证程序无法在Java代码中实现这个接口。注意，使用这个方法，当你添加一个新的子类时，when 表达式返回一个值并且编译失败，同时指出你的代码哪里需要修改。

**NOTE** 注意 此时，sealed 功能是相当受限的。举个例子，所有的子类都必须被嵌套。同时子类不能是数据类（数据类会在后续章节谈到）。未来版本的Kotlin计划要放松这些限制。

跟你回想的一样，在Kotlin中，你在继承一个类或者实现一个接口时都在使用冒号。让我们窥探一下子类的声明（语法）：

```
class Num(val value: Int) : Expr()
```

除了 `Expr()` 中类名后面的括号的含义以外，这个简单的例子应该很清晰。我们将会在下节讨论它们，也会覆盖Kotlin中初始化类的内容。

如你所知，Java中，一个类可以声明一个或多个构造函数。Kotlin也是相似的，但有一个额外的改变：它在主构造函数（**primary constructor**，通常是主要的，简洁的方式来初始化一个类并且被声明在类的主体外部的）和次要构造函数（**secondary constructor**，声明在类主体的内部）之间制造些差异。它也允许你再初始化器代码块中（**initializer blocks**）放一些额外的初始化逻辑。首先，我们来看看声明主构造函数和初始化器的语法。然后我们将会解释如何声明多个构造函数。之后，我们将会更多的讨论属性。

## 4.2.1 初始化类：主构造器和初始化器

在第二章，你看到了如何声明一个简单的类：

```
class User(val nickname: String)
```

通常，类中所有的声明都会在闭合大括号的内部。你可能很好奇为什么这个类没有闭合的大括号而是仅仅在小括号内有一个声明。圆括号内部的代码叫做主构造函数。它有两个目的：指定构造函数参数，同时定义由这些参数初始化的属性。让我们揭开这里会发生什么并且看看你可以编写的同时实现同样功能的最直接的代码：

```
class User constructor(_nickname: String) { // 1 带有一个参数的主构造器

    val nickname: String

    init { // 2 初始化块
        nickname = _nickname
    }
}
```

在这个例子中，你看到了两个新的Kotlin关键字：`constructor` 和 `init`。`constructor` 关键字开启了一个主构造器或次构造器的声明。`init` 关键字引入了一个 `initializer` 块。这样的块包含了当这个类通过主构造函数创建时执行的初始化代码。由于主构造器有一个限制语法，它不能包含初始化代码。这就是为什么你需要初始化块的原因。如果你想要，你可以在一个类中声明多个初始化块。构造函数参数 `_nickname` 中的下划线是为了区分构造函数参数名的属性名。一个可选的方案是使用同样的名字的同时写上 `this` 来避免歧义，就像Java中经常做的那样：`this.nickname = nickname`。在这个例子中，你不需要在初始化块中放置初始化代码。因为它可以跟 `nickname` 属性声明合并。你也可以省略 `constructor` 关键字，如果主构造函数没有标记或者可见性修饰符。如果你应用了这些改变，你可以得到下面的代码：

```
class User(_nickname: String) { // 1 带有一个参数的主构造函数
    val nickname = _nickname // 2 属性被参数初始化
}
```

这有另一种方法来声明同样的类。注意，你是如何能够引用属性初始化和初始化代码中的主构造函数参数的。前面的两个例子在类主体中使用 `val` 关键字声明的属性。如果属性被初始化为对应的构造函数参数，这份代码可以简化为在参数面前添加 `val` 关键字。这将替代类中的属性定义：

```
class User(val nickname: String) // 1 "val"意味着对应的属性是为构造函数参数生成的
```

`User` 类的以上所有声明都是等价的，但是最后一种使用了最精简的语法。你可以为构造函数参数声明默认值，就像函数参数那样：

```
class User(val nickname: String,
    val isSubscribed: Boolean = true) // 1 为构造函数参数提供默认值
```

为了创建一个类的实例，你可以直接调用构造函数，而无需 `new` 关键字：

```
>>> val alice = User("Alice") // 1 为isSubscribed参数使用默认值"true"
>>> println(alice.isSubscribed)
true

>>> val bob = User("Bob", isSubscribed = false) // 2 你可以为某些构造函数参数显式的指定名字
>>> println(bob.isSubscribed)
false
```

看上去是Alice默认订阅了邮件列表，然而Bob阅读了条款并小心的取消了默认选项。

**NOTE** 注意 如果所有的构造函数参数都有默认值，编译器会额外生成一个使用了所有默认值但没有参数的构造函数。这使得通过无参数构造函数初始化类的库来使用 Kotlin 变得更加容易。

如果你的类有一个超类，主构造函数也需要初始化超类。你可以通过在基类列表中的超类引用后面提供超类构造函数来达到这个目的：

```
open class User(val nickname: String) { ... }
class TwitterUser(nickname: String) : User(nickname) { ... }
```



如果你没有为某个类声明任意的构造函数，编译器会为你生成一个什么也不干的默认构造函数：

```
open class Button // 1 生成一个没有参数的默认构造函数
```

这就是为什么你需要在超类名的后面有一个空括号。注意跟接口的不同：接口没有构造函数，所以如果你实现了一个接口，你绝对不能在它的超类列表名字后面放置圆括号。如果你想确保你的类不能被其他代码初始化，你必须让构造函数私有。以下是你如何让主构造函数私有的：

```
class Secretive private constructor() {} // 1 这个类有一个私有构造函数
```

作为替代方案，你能够在类主体内，一个更加常见的方式来声明它：

```
class Secretive {  
    private constructor()  
}
```

由于 `Secretive` 类只有一个私有构造函数，类外部的代码不能初始化它。在这一章的后续部分，我们将会讨论伴生对象。它可能是放置调用这样的构造函数的好地方。

**SIDEBAR** 可选的私有构造函数 在Java中，你可以使用一个私有构造函数来禁止类实例化以表达更常见的想法：类是一个静态工具成员的容器或者是一个单例。Kotlin为这些意图准备了内置的语言特性。你把顶层函数（你在3.2.3节看到的）作为静态工具。为了表达单例，你使用对象什么，就像你在本章后续的4.4.1节看到的那样。在大部分真实用例中，类的构造函数是直接的：它没有包含参数或者把参数分配给相应的属性。这就是为什么Kotlin的主构造函数会有精简的语法：对于大多数场景它都可以很好的工作。但是生活并不总是容易的，所以Kotlin允许你定义你需要的所有构造函数。让我们来看看这是如何工作的。

### 4.2.2 次构造函数：以不同的方式初始化超类

一般来讲，有多个构造函数的类在Kotlin代码中没有Java那么常见。在Java，你需要重载构造函数主要的情景被Kotlin对默认参数值的支持覆盖了。

**TIP** 提示 不要声明多个次构造函数来重载，同时为参数提供默认值。相反的，直接指定默认值。



但是任然有需要多个构造函数的情景。最常见的一个情形出现了，当你需要扩展一个提供多个以不同方式初始化类的构造函数的框架类。想象一个，声明在Java中，有两个构造函数（如果你是一个Android开发者，你可能认得这个定义）的 `View` 类。Kotlin中的一个相似的声明将会跟下面的代码一样：

```
open class View {  
  
    constructor(ctx: Context) { // 1 次构造函数  
        // some code  
    }  
  
    constructor(ctx: Context, attr: AttributeSet) { // 1 次构造函数  
        // some code  
    }  
}
```

这个类并没有声明一个主构造函数（跟你想说的一样，因为在类头部的名字后面没有圆括号），但是它声明了两个次构造函数。通过使用 `constructor` 关键字引入了一个次构造函数。你可以声明跟你所需的一样多的次构造函数。 如果你想要扩展这个类，你可以声明同样的构造函数：

```
class MyButton : View {  
    constructor(ctx: Context)  
        : super(ctx) { // 1 调用超类构造函数  
        // ...  
    }  
  
    constructor(ctx: Context, attr: AttributeSet)  
        : super(ctx, attr) { // 1 调用超类构造函数  
        // ...  
    }  
}
```

你在这里定义了两个构造函数，每一个都通过使用 `super()` 关键字调用了对应的超类构造函数。图4.3解释了这一点。一个箭头展示委托了那个构造函数。

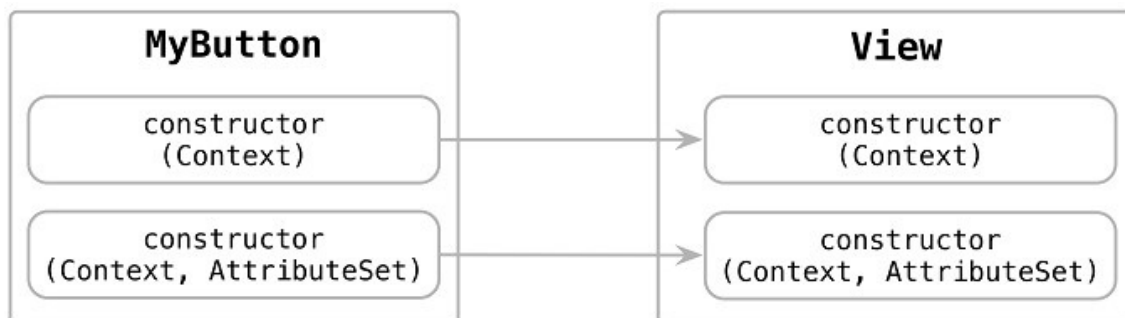


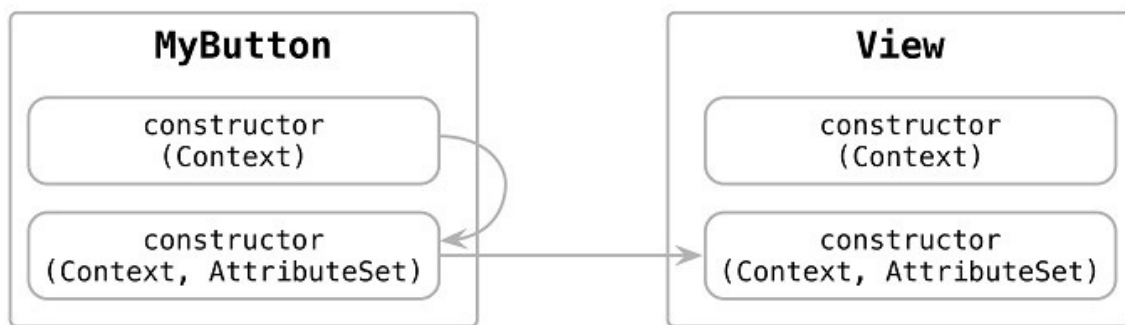
图4.3 使用不同的超类构造函数

就像Java那样，你也有一个可选的权利使用 `this()` 关键字从一个构造函数调用你的类中的其他构造函数。以下是它如何工作的：

```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) { // 1 委托给类中的其他构造函数
        // ...
    }

    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
        // ...
    }
}
```

你可以改变 `MyButton` 类，来将其中一个构造函数委托给类中的其他构造函数，并为参数传递了默认值。如图4.4所示，次构造函数继续调用 `super()`。



如果类没有主构造函数，那么每一个次构造函数必须初始化基类或者委托其他做这些事的构造函数。想一下之前的那张图，每一个次构造函数必须有一个对外的箭头为路径的开始，并在基类的构造函数结束。Java互操作性是你需要使用次构造函数的主要应用场景。但是有一个可能的场景：当你有多种方式来创建带有不同参数列表的类实例的时候。我们将会后续的4.4.2一节讨论一个例子，我们已经讨论了如何定义有意义的构造函数。现在让我们把注意力转移到有意义的属性。

### 4.2.3 实现声明在接口中的属性

在Kotlin中，一个接口可以包含抽象属性声明。这有一个声明这样一个接口定义的例子：

```
interface User {
    val nickname: String
}
```

这意味着实现了 `User` 接口的类需要提供一种方式来和获取 `nickname` 的值。接口并没有指定一个是否应该存储在一个备份字段或者通过一个 `getter` 来获取。因此，接口本身不包含任何状态。如果有需要的话，也只有实现了接口的类能够存储值。让我们来看看接口的一些可能的实现：`PrivateUser`，只填充他们的昵称；`SubscribingUser`，被迫提供一个邮箱来注册；`FacebookUser`，简单的共享了他们的Facebook账号ID。所有的这些类都以不同的方式实现了接口中的抽象属性：

```
class PrivateUser(override val nickname: String) : User // 1 主构造函数属性

class SubscribingUser(val email: String) : User {
    override val nickname: String
    get() = email.substringBefore('@') // 2 自定义getter
    属性初始化器
}

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId) // 3
    属性初始化器
}

>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

对于 `PrivateUser`，你使用精简的语法来直接声明一个主构造函数中的属性。这个属性实现了来自 `User` 类的抽象属性，因此你把它标记为 `override`。对

于 `SubscribingUser`，`nickname` 属性通过一个自定义的`getter`来实现。这个属性并没有支持字段（`backing field`）来存储它的值。它只有一个计算来自每个调用中的邮箱的昵称的`getter`。

对于 `FacebookUser`，你可以在它的初始化器中为 `nickname` 属性分配值。你使用一个支持返回给定IDFacebook用户名称的 `getFacebookName` 函数（假设它在某个地方被定义了）。这个函数是代价昂贵的：他需要跟Facebook建立一个连接来得期望的数据。这就是为什么你决定在初始化阶段调用它一次。注意 `nickname` 在 `SubscribingUser` 和 `FacebookUser` 中的不同实现。尽管它们看上去很相似，第一个属性有一个在每次访问时计算 `substringBefore` 的自定义的`getter`，然而 `FacebookUser` 中的属性有一个存储类初始化时计算的结果的支持字段。除了抽象属性声明之外，接口也可以包含带有多个`getter`和`setter`属性，只要它们不引用支持字段（支持字段要求在一个接口中存储状态，而这是不允许的）。然我们来看一个例子：

```
interface User {
    val email: String
    val nickname: String
    get() = email.substringBefore('@') // 1 属性并没有支持字段：每次调用都要重新计算结果
}
```

这个字段包含了抽象属性 `email`，同时 `nickname` 属性带有自定义的`getter`。第一个属性在子类必须被覆盖，然而第二个可以被继承。跟接口中实现的属性不同，类中实现的属性拥有支持字段的全部访问权限。让我们看看你如何能够通过访问器引用它们。

## 4.2.4 通过`getter`或者`setter`访问支持字段

你已经看到了有关两种属性的一些例子：保存值的属性和带有每次访问都进行计算的自定义访问器的属性。现在让我们来看看你如何能够合并两者同时实现有一个保存值的属性并提供当值被访问或者修时才执行的额外逻辑。为了支持这一点，你需要能够从它的访问器访问属性的支持字段。举个例子，让我们假设你想要记录属性中存储的数据的变化。你声明了一个可变属性并在`setter`的每一次访问时执行了额外的代码：

```
class User(val name: String) {var address: String = "unspecified"
    set(value: String) {
        println("""
            Address was changed for $name:
            "$field" -> "$value".""".trimIndent()) // 1 读取支持字段的值
更新支持字段的值

        field = value // 2 更新支持字段的值
    }
}

>>> val user = User("Alice")
>>> user.address = "Elsenheimerstraße 47, 80687 München"
Address was changed for Alice:
"unspecified" -> "Elsenheimerstraße 47, 80687 München".
```

像往常那样，通过声明 `user.address = "new value"`，这实际上调用了一个`setter`，你改变了属性的值。在这个例子中，`setter`被重新定义了。所以额外的日志代码被执行了（为了简单起见，在这个例子中你仅仅把它打印出来）。在`setter`内部，你使用特殊的标记符 `field` 来访问支持字段的值。在`getter`中，只能读取值。在`setter`中，你既可以读取又可以修改它。注意，你只能为可变属性重定义其中一个访问器。前一个例子中的`getter`是不重要的，它仅仅返回了字段的值。所以你不需要重新定义它。你可能会好奇是什么导致了有支持字段和没有支持字段的属性之间的差异呢？你访问它的方式不依赖于属性是否有支持字段。编译器将会为属性产生支持字段如果你显式的引用它或者使用了默认访问器的实现。如果你提供了一个不使用 `field` 的自定义访问器实现，支持字段不会出现。有时候，你不需要改变访问器的实现，按时你需要改变它的可见性。让我们来看看你如何能做到这一点。

## 4.2.5 改变访问器的可见性

访问器的可见性默认是跟属性的一样的。但是如果你想的话，通过在 `get` 或者 `set` 关键字之间放置可见性修饰符，你可以改变这一点。为了看看你可以如何使用它，让我们来看一个例子：

```
class LengthCounter {
    var counter: Int = 0
    private set // 1 你无法在类的外部改变这个属性

    fun addWord(word: String) {
        counter += word.length
    }
}
```

这个类计算加入到它当中的单词的总长度。由于它是类向客户端提供的API的一部分，保存总长度的属性是公开访问的。但是，你需要确保它只能在类中被修改。因为不这样做的话，外部代码可以改变它并存入一个不正确的值。所以，你让编译器生成一个带有默认可见性的 `getter`。然后你将 `setter` 的可见性改为 `private`。下面的代码演示了你可以如何使用这个类：

```
>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> print(lengthCounter.counter)
3
```

你创建了一个 `LengthCounter` 实例。然后你添加了一个单词"Hi!"的长度3。现在 `counter` 的属性保存的是3。

**SIDEBAR** 更多有关属性的话题 在书的后续部分，我们将会继续属性的讨论。以下是一些参考：

- 非空属性中的 `lateinit` 修饰符指的是，这个属性会在后续时机被初始化。在构造函数被调用之后初始化，在某些框架中是非常常见的。这个特性将会在第6章覆盖到。
- 懒初始化属性，作为更加广泛的委托属性特性的一部分，将会在第7章被覆盖到。
- 出于Java框架的兼容性考虑，你可以在Kotlin中使用模拟Java特性的标注。举个例子，属性的 `@JvmField` 标注无需使用访问器就暴露了一个 `public` 字段。你将会在第10章了解到更多有关标注的知识。
- `const` 修饰符让使用标注更加方便。同时它允许你使用一个原始类型或者 `String` 类型的属性作为标注声明。第10章会给出细节。

以上内容总结了我们有关如何在Kotlin编写重要构造器和属性的讨论。接下来，你将会看到如何使用 `data` 类的概念来让值-对象类变得更加友好。



你只依赖你只依赖你是的是Java平台定义了大量的需要在多个类中出现但往往以一种呆板的方式实现的方法。例如, `equals()`, `hashCode()` 和 `toString()`。所幸的是, Java IDE可以自动生成这些方法。所以你通常不需要手动编写这一类代码。但是这样的话, 你的代码库就会包含许多的八股代码。Kotlin编译器(在这个问题上)向前了一步: 它可以在后台执行机械代码的生成, 而不会使用生成的结果打乱你的代码。

你已经见识过对于重要的类构造函数和属性访问器, 它是如何工作的了。让我们来看看更多关于Kotlin编译器生成对简单的数据类非常有用而且极大的简化了类委托模式的典型方法的例子。

### 4.3.1 通用的对象方法

像Java那样, 所有的Kotlin类有多个你可能想要覆盖的方法:

`toString()`, `equals()` 和 `hashCode()`。让我们来看看这些方法长什么样。还有Kotlin如何能够帮助你自动生成它们的实现。作为一个切入点, 你将会使用一个存储客户端名称和邮编的简单的 `Client` 类:

```
class Client(val name: String, val postalCode: Int)
```

让我们来看看类实例是如何表现为一个字符串的。

字符串表示: **TOSTRING()**

Kotlin中的所有类, 跟Java中的一样, 都提供了获取类对象的字符串表示的一种途径。尽管你也可以在其他环节中使用这个功能, 但这一特性主要是用来调试和输出日志的。默认情况下, 对象的字符串表示看起来像 `Client@5e9f23b4` 这样的形式。但这样的形式并没有多大用处。要想改变它, 你需要覆盖 `toString()` 方法。

```
class Client(val name: String, val postalCode: Int) {
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

现在现在客户端(对象)的表示长这样:

```
>>> val client1 = Client("Alice", 342562)
>>> println(client1)
Client(name=Alice, postalCode=342562)
```

这种方式给出了更多的信息, 不是吗?

对象等价性: `equals()`

`Client` 类所有的计算都在它的外部进行。这个类只保存数据。它注定是简单、透明的。不过，对于这个类的行为，你可能会有一些要求。例如，假设你希望如果它们包含同样的数据，两个对象被认为是相等的：

```
>>> val client1 = Client("Alice", 342562)
>>> val client2 = Client("Alice", 342562)
>>> println(client1 == client2) // 1 Kotlin中，==检查两个对象是否相等，并非它们的引用。它在底层调用的是"equals"。
false
```

你看到对象是不相等的。这意味着你必须为 `Client` 类覆盖 `equals` 方法。

SIDEBAR 用于等价性判断的 `==`

Java中，你可以使用 `==` 来比较原始和引用类型。如果用于原始类型，Java的 `==` 操作符比较两者的值。然而 `==` 操作符用于引用类型时，比较的是引用。所以，在Java中，调用 `equals` 是最佳实践。忘了这样做会导致出名的问题。

Kotlin中，`==` 是比较两个对象的默认方式：它通过在底层调用 `equals` 来比较两者的值。因此，如果 `equals` 方法在你的类中被覆盖的话，你可以使用 `==` 来安全的比较它的实例。对于引用比较，你可以使用 `===` 操作符。它跟Java中的 `==` 完全一样。

让我们来看看修改后的 `Client` 类：

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean { // 1 "Any"类似于java.lang.Object:它是Kotlin中所有类的超类。可为空的类型"Any?"指的是other变量可以为null
        if (other == null || other !is Client) // 2 检查other变量是否为Client
            return false
        return name == other.name && // 3 检查相应的属性是否相等
            postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

这里想提醒一下你，Kotlin中的 `is` 是Java中 `instanceof` 的对应物。它检查一个值是否拥有指定的类型。正如 `in` 检查的否定形式（我们在2.4.4一节讨论过的）：`!in` 操作符，`!is` 操作符表示 `is` 检查的否定形式。这样的操作符让你的代码更容易阅读。在第6章，我们将会详细讨论可为空的类型，以及为什么 `other == null || other !is Client` 条件可以被简化为 `other !is Client`。

由于Kotlin中 `override` 修饰符是强制使用的，（这能）保护你不经意的写成 `fun equals(other: Client)`。这种写法将会添加一个新的方法，而不是覆盖 `equals`。在你覆盖了 `equals` 方法之后，你可能希望拥有相同属性的客户端相等。事实上，前一个例子中的等价



性检查 `client1 == client2` 现在返回 `true`。但是如果你想对客户端做更多复杂的事情，那不会有用了。你可能会说问题是 `hashCode` 缺失了。那是真实的案例。我们即将讨论为什么这是重要的（在你阅读下一个节之前，确保你理解了上面的解释）。

#### 哈希容器：`HASHCODE()`

`hashCode` 方法应该总是跟 `equals` 一同被覆盖。这一章节解释为什么（要这样做）。

让我们创建一个只有一个元素的集合：一个叫做Alice的客户端。当你检查这个集合是否包含拥有相同名字和邮政编码的客户端时，你会认为这样的对象是相等的，但是集合并不包含：

```
>>> val processed = setOf(Client("Alice", 342562))
>>> println(processed.contains(Client("Alice", 342562)))
false
```

原因是 `Client` 类缺失了 `hashCode` 方法。因此，它跟常见的 `hashCode` 约定冲突了：如果两个对象是相等的，他们必须拥有相同的哈希值。`processed` 集合是一个哈希集合。哈希集合中的值以一种优化后的方式进行比较：首先比较两者的哈希值，然后，当且仅当它们相等时，比较真实值。再之前的例子中，两个不同的 `Client` 类实例的哈希值是不同的。所以集合判定不包含第二个对象，尽管 `equals` 返回 `true`。因此，如果不遵循（这个）规则，对于这个对象哈希集合无法正确的工作。

为了解决这个问题，你可以添加 `hashCode()` 实现到类中：

```
class Client(val name: String, val postalCode: Int) {
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
}
```

现在你有一个可以在所有场景下按预期工作的类，但是请注意你必须写多少代码。幸运的是，Kotlin编译器可以通过自动的生成所有的这些方法来帮助你。让我们来看看你可以如何要求编译器去做这件事。

### 4.3.3 数据类：自动生成的通用方法的实现

如果你想要你的类成为你的数据的便捷容器，别忘了覆盖这些方

法：`toString`, `equals` 和 `hashCode`。通常，这些方法的实现是直接的。像IntelliJ IDEA一类的IDE可以帮助你自动生成并验证这些方法是正确的并且一致的实现。

好消息是，在Kotlin中，你不必生成所有这些方法。如果你添加了 `data` 修饰符到你的类，所有必要的方法会为你自动生成：

```
data class Client(val name: String, val postalCode: Int)
```

很简单，对吧？你有一个覆盖了所有Java标准方法的类：

- 用于比较实例的 `equals()`
- 在基于哈希的容器（例如 `HashMap`）中作为键的 `hashCode()`
- 以声明的顺序所有字段的生成字符表示的 `toString()`

`equals` 和 `hashCode` 方法考虑了主构造函数中声明的所有属性。生成的 `equals()` 方法检查所有值相等的属性。`hashCode()` 方法返回一个值。而这个值依赖于所有属性的哈希值。注意，没有在主构造函数中声明的属性没有参与等价性检查和哈希值计算。

这并不是为 `data` 类生成的有用方法的完整列表。下一个章节将会展示更多（有关内容）。7.4节会补充剩下的部分。

数据类和不可变性：**COPY()**方法

注意，尽管数据类的属性并不要求为 `val`，你也可以使用 `var`，但是强烈推荐你使用只读的属性，让数据类的实例不可变。如果你想使用这个实例作为 `HashMap` 或者一个类似的容器的键，这一点是必要的。因为否则的话，容器会进入一个失效的状态，如果对象使用一个添加到容器之后被修改的键。不可变的对象也更容易推断，特别是在多线程代码中：一旦一个对象被创建了，它会保持原始的状态。当你的代码使用它时，你不需要担心其他的线程修改这个对象。

为了让数据类用作不可变对象更容易，Kotlin编译器为它们生成了一个方法：一个允许你复制类实例、改变某些属性的值的方法。在适当的位置创建一个副本通常是修改实例的一个好的可选方案：副本有一个独立的生命周期，不会影响代码中指向原始实例的地方。如果你手动实现 `copy()` 方法，以下是它可能的一个模样：

```
class Client(val name: String, val postalCode: Int) {
    ...
    fun copy(name: String = this.name,
             postalCode: Int = this.postalCode) =
        Client(name, postalCode)
}
```

下面是 `copy()` 方法如何使用：

```
>>> val bob = Client("Bob", 973293)
>>> println(bob.copy(postalCode = 382555))
Client(name=Bob, postalCode=382555)
```

你已经见识了 `data` 修饰符如何使得值-对象类更加容易使用。现在让我们讨论其他的Kotlin特性来让你避开IDE生成的八股代码：类委托。

### 4.3.3 类委托：使用 `by` 关键字

在设计大型面向对象系统时的一个常见问题是继承实现导致的脆弱性。当你扩展一个类并覆盖它当中的方法时，你的代码变得依赖于你所扩展的类的实现细节。当系统进化以及基类的实现改变或者添加新的方法到类基类中时，你所做的有关你的类的行为将变得无效。所以你的代码可能以异常的表现而告终。

Kotlin的设计承认了这个问题并将类默认视为 `final`。这确保了只有为扩展性而设计的类才能被继承。当使用这样的类时，你看到它是开放的，你记住的是修饰需要兼容派生的类。但是，你经常需要添加行为到另一个类，即使它并不是设计用来扩展的。一个常见的实现方法是著名的装饰器模式。这个模式的本质是：创建一个新的类，实现跟原始类中相同的方法并将原始类的实例存储为一个字段。原始类的行为不需要被修改的方法将被转发给原始类实例。这个方法的一个缺点是，它需要大量的模板代码（很多像IntelliJ IDEA的IDE都有专门的特性来为你生成这些代码）。举个例子，这是你为实现一个跟 `Collection` 一样简单的装饰器所需的代码量，即便你没有修改任何的行为：

```
class DelegatingCollection<T> : Collection<T> {
    private val innerList = arrayListOf<T>()
    override val size: Int get() = innerList.size
    override fun isEmpty(): Boolean = innerList.isEmpty()
    override fun contains(element: T): Boolean = innerList.contains(element)
    override fun iterator(): Iterator<T> = innerList.iterator()
    override fun containsAll(elements: Collection<T>): Boolean =
        innerList.containsAll(elements)
}
```

好消息是Kotlin为作为语言特性的委托包含了一流的支持。每当你实现一个接口时，你可以说你通过 `by` 关键字把接口的实现委托给另一个对象。以下是你可以如何使用这个方法重写前一个例子：

```
class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList {}
```

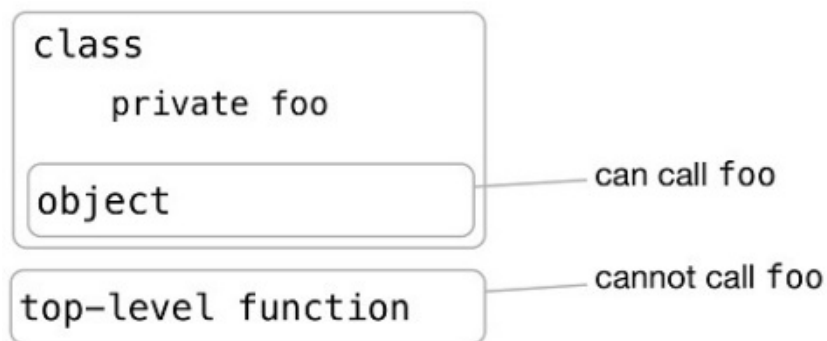
如你所见，这个类中所有方法的实现都消失了。编译器会生成这些实现。它们跟 `DelegatingCollection` 例子中的实现相似。由于代码中没有有趣的内容，当编译器可以自动的为你做同样的工作时手动编写是没有意义的。现在，当你需要改变一些方法的行为时，你可以覆盖它们。你的代码将会被调用而不是（编译器自动）生成的方法。你可以忽略你满

意的委托给底层示例的默认实现。让我们来看看你可以如何使用这个技术来实现一个统计尝试的次数集合，并向其添加元素。举个例子，如果你执行某种去重操作，通过比较尝试次数来添加一个带有结果集合大小的元素，你可以使用这样的集合来测量这个操作的效率如何：

```
class CountingSet<T>(  
    val innerSet: MutableCollection<T> = HashSet<T>()  
) : MutableCollection<T> by innerSet { // 1 委托MutableCollection的实现给innerSet  
  
    var objectsAdded = 0  
  
    override fun add(element: T): Boolean { // 2 没有委托，提供了一个不同的实现  
        objectsAdded++  
        return innerSet.add(element)  
    }  
  
    override fun addAll(c: Collection<T>): Boolean {  
        objectsAdded += c.size  
        return innerSet.addAll(c)  
    }  
}  
  
>>> val cset = CountingSet<Int>()  
>>> cset.addAll(listOf(1, 1, 2))  
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")  
3 objects were added, 2 remain
```

如你所见，你覆盖了 `add` 和 `addAll` 方法来增加计数。同时你把 `MutableCollection` 接口其余的实现委托给了你所包装的容器。重要的部分是你没有引入任何有关底层集合是如何实现的依赖。举个例子，你不会在意实现了 `addAll()` 方法的集合是通过在一个循环中调用 `add()` 方法，还是它使用了为一个特定的场景而优化的一种不同的实现。你完全控制了当客户程序调用你的类时会发生什么。你只依赖于有相关文档的底层集合的API来实现你的操作。所以你可以依赖于它继续工作。我们已经讨论完了有关Kotlin编译器如何为类生成有用的方法。让我们开始Kotlin类故事的最后、最大块的部分：`object` 关键字和适合使用的情形。

\*\*



object 关键字在Kotlin中大量出现。但是它们共用相同的核心理念：这个关键词定义一个类并且同时创建了这个类的一个实例。让我们来看看它使用的不同情况：

- 对象声明是定义一个单例的一种方法
- 伴生对象可以包含工厂方法和其他跟这个类相关但不需要类实例调用的方法。它们的成员可以通过类名访问。
- 使用对象表达式而不是Java的匿名内部类

现在我们将详细的讨论这些Kotlin特性。

### 4.4.1 对象声明：单例变得更容易

在面向对象系统的设计中相当常见的是你需要的一个类只有一个实例。在Java中，这个类通常使用单例模式实现：你定义了一个拥有私有构造函数和一个只保存唯一一个类实例的静态字段。Kotlin为使用对象声明特性提供了一流的语言支持。对象声明组合了类声明以及这个类的一个单例声明。举个例子，你可以使用一个对象声明来表示一个组织的工资单。你可能不会有多个工资单，所以为此使用一个对象听起来更加合理：

```

object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
        for (person in allEmployees) {
            ...
        }
    }
}
  
```

如你所见，对象声明使用 object 关键字引入。一个对象声明有效的用一个单一的声明定义了一个类和这个类的一个变量。就像一个类一样，一个对象声明可以包含属性、方法、初始化块等声明。唯一不允许的是构造函数（主要的或者次要的都不允许）。不同于正常的类实

例，对象声明在定义的时刻立即被创建，不是通过代码中某处的构造函数调用。因此，为一个对象声明定义一个构造函数并没有意义。跟一个类实例一样，一个对象声明允许你通过在 `.` 符号的左边使用对象名来调用方法和访问属性。

```
Payroll.allEmployees.add(Person(...))
Payroll.calculateSalary()
```

对象声明也能继承自类和接口。当你使用的框架要求你实现一个接口，但你的实现没有包含任何声明时，这是非常有用的。让我们拿 `java.util.Comparator` 接口来举个例子。一个 `Comparator` 实现接收了两个对象并返回了表示那个对象更大的一个整数。比较器几乎不保存任何数据，所以你通常只需要为比较对象的一种特定的方式准备一个单独的 `Comparator` 实例。这是对象声明的一个完美用例。作为一个具体的示例，让我们实现以大小写敏感的方式比较文件路径的比较器：

```
object CaseInsensitiveFileComparator : Comparator<File> {
    override fun compare(file1: File, file2: File): Int {
        return file1.getPath().compareTo(file2.getPath(),
            ignoreCase = true)
    }
}
>>> println(CaseInsensitiveFileComparator.compare(
... File("/User"), File("/user")))
0
```

你可以在普通对象可以使用的任何环境使用单例对象。举个例子，你可以把这个对象作为参数传递给接收一个 `Comparator` 的函数：

```
>>> val files = listOf(File("/Z"), File("/a"))
>>> println(files.sortedWith(CaseInsensitiveFileComparator))
[/a, /Z]
```

这里，你使用 `sortedWith` 函数。它根据一个具体的比较器返回一个列表。

**SIDEBAR**    单例和依赖注入    就像单例模式那样，对象声明使用在大型软件系统中并非总是理想的。对于依赖较少或者没有依赖的小块代码，它们是很不错的。但不适合跟系统的许多其他部分交互的大型组件。主要的原因是你没有任何对象实例化的控制，而且你不能为构造函数指定参数。这意味着你不能取代对象自身的实现，或者在单元测试及软件系统的不同配置中，对象所依赖的其他类。如果你需要那样的能力，你应该结合一个诸如Guice的依赖注入框架来使用常规的Kotlin类，就像在Java中那样。



你也可以在类中声明一个对象。这样的对象也有只有一个单例。它们在每个容器类实例中没有单独的实例。举个例子，放置一个比较那个类内部的一个类的对象的比较器是合理的：

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
            p1.name.compareTo(p2.name)
    }
}
>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name="Alice"), Person(name="Bob")]
```

**SIDEBAR** 通过Java使用Kotlin Kotlin中的一个对象被编译为一个带有一个保存它的单例的静态字段 `INSTANCE` 的类。如果你在Java中实现过单例模式，你可能已经手动做过同样的事情了。因此，为了从Java代码使用Kotlin对象，你访问这个静态的 `INSTANCE` 字段：`/* Java */ CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);` 在这个例子中，`INSTANCE` 字段为 `CaseInsensitiveFileComparator` 类型。

现在，让我们来看一个特殊的案例。对象嵌套在一个类的内部：伴生对象。

## 4.4.2 伴生对象：一个放置工厂函数和静态成员的地方

在Kotlin中，类不能有静态成员。Java的 `static` 关键字不是Kotlin语言的一部分。作为一个替代，Kotlin依赖于包级别的函数（在许多情况下可以替代Java的静态函数）和对象声明（替代Java的静态方法，在某些情况下也可以作为静态字段）。在大部分情况，推荐你使用包级别函数。但是包级别函数无法访问类的私有成员。因此，如果你需要写一个可以无需拥有一个类实例时被调用，但需要访问类的内部的函数，你可以在类的内部把它写成一个对象的成员。所有这样的函数的例子都可以是一个工厂方法。

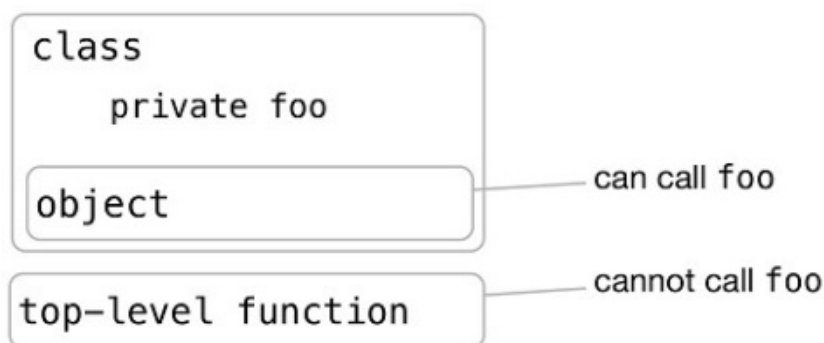


图4.5 私有成员无法使用在顶层，除非这个函数在类的外部

定义在类中的一个对象可以用一个特殊的关键字来标记：`companion`。如果你这样做的话，你就获得了通过所在的类的名字直接访问这个对象的方法和属性的能力。而你无需显式的指定这个对象的名字。最终的语法看上去像极了Java的静态方法调用。这是一个演示语法的基本案例：

```
class A {
    companion object {
        fun bar() {
            println("Companion object called")
        }
    }
}
>>> A.bar()
Companion object called
```

还记得我们承诺过你一个放置调用私有构造函数的地方吗？那就是伴生对象了。伴生对象拥有访问类的所有私有成员的权限。它是工厂模式的一个理想候选方案。来看一个声明两个构造函数并将其改为在伴生对象使用工厂方法的例子。我们将会使用前面的 `FacebookUser` 和 `SubscribingUser` 来构建一个案例。之前，这些实体是实现了公共的 `User` 接口的不同的类实例。现在你决定只使用一个类来管理。但是你将提供不同的方法来创建它：

```
class User {
    val nickname: String

    constructor(email: String) { // 次构造函数
        nickname = email.substringBefore('@')
    }

    constructor(facebookAccountId: Int) { // 次构造函数
        nickname = getFacebookName(facebookAccountId)
    }
}
```

表达同样逻辑的一个可选方法，是使用工厂方法来创建这个类的实例。这也许是出于多个有利因素的考虑。`User` 实例是通过工厂方法创建的，而不是通过多构造函数：



```
class User(val nickname: String) {  
    companion object { // 声明伴生对象  
  
        fun newSubscribingUser(email: String) = // 工厂方法通过email创建一个新的用户  
            User(email.substringBefore('@'))  
  
        fun newFacebookUser(accountId: Int) = // 工厂方法通过Facebook账号ID创建一个新的用户  
            User(getFacebookName(accountId))  
    }  
}
```

你可以通过类名来调用伴生对象的方法：

```
>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")  
>>> val facebookUser = User.newFacebookUser(4)  
  
>>> println(subscribingUser.nickname)  
bob
```

工厂方法是非常有用的。它们可以根据它们的用途来命名，就像例子中展示的那样。另外，当 `SubscribingUser` 和 `FacebookUser` 是类时，正如例子中那样，一个工厂方法可以返回声明工厂方法的类的子类。当这不是必要的时候，你也可以避免创建新的对象。例如，你可以确保每个邮箱地址对应一个不同的 `User` 实例，并且，当使用一个邮箱地址调用一个存在于缓存的工厂方法时，返回一个存在的实例而不是一个新的实例。但是如果你需要扩展这样的类，使用多个构造函数或许是一个更好的方案。因为伴生对象并不能在子类中被覆盖。

### 4.4.3 伴生对象是常规的对象

伴生对象是声明在一个类中的常规对象。它可以被命名、可以实现一个接口，或者拥有扩展函数或熟悉。在这一节，我们来看一个例子。假定你在为一个公司的工资单web服务而工作。你需要把对象序列化和反序列化成JSON。你可以在一个伴生对象中添加序列化逻辑。

```
class Person(val name: String) {
    companion object Loader {
        fun fromJSON(jsonText: String): Person = ...
    }
}

>>> person = Person.Loader.fromJSON("{\"name: 'Dmitry'}") // 你可以使用两种方式来调用fromJSON方法
>>> person.name
Dmitry

>>> person2 = Person.fromJSON("{\"name: 'Brent'}") // 你可以使用两种方式来调用fromJSON方法
>>> person2.name
Brent
```

在大多数情况下，你通过包含它的类的名字来引用伴生对象。所以你不需要担心它的名字。但是你可以指定它的名字，如果有需要的话，就像在例子中那样：`companion object Loader`。如果你忽略了伴生对象的名字，分配给它的默认名称是 `Companion`。当我们讨论伴生对象扩展时，你将会后面看到一些使用这个名字的例子。

在伴生对象中实现接口 就像其他对象声明那样，一个伴生对象那个可以实现接口。正如你将看到的那样，你可以直接使用容器类的名字作为一个实现接口的对象实例。假定在你的系统中你有多种对象，包括 `Person`。你想提供一个普通的方法来创建所有类型的对象。你有一个用于能够从JSON反序列化的对象的 `JSONFactory` 接口。你的系统中所有的对象都能够通过这个工厂来创建。你可以为你的 `Person` 类提供这个接口的一个实现：

```
interface JSONFactory<T> {
    fun fromJSON(jsonText: String): T
}

class Person(val name: String) {
    companion object : JSONFactory<Person> { // 实现了一个接口的伴生对象
        override fun fromJSON(jsonText: String): Person = ...
    }
}
```

如果你有一个使用一个抽象工厂来加载实体的函数，你可以传递 `Person` 对象给它：

```
fun loadFromJSON<T>(factory: JSONFactory<T>): T {
    ...
}

loadFromJSON(Person) // 传递伴生对象给这个函数
```

注意，`Person` 类的名字被用作一个 `JSONFactory` 实例，

**SIDEBAR** Kotlin伴生对象和静态成员 类的伴生对象被编译成一个常规对象：类里面的一个指向自己的实例的静态字段。如果伴生对象没有被命名，它可以从Java代码通过 `Companion` 引用来访问：`/* Java */ Person.Companion.fromJSON("...");` 如果一个伴生对象有一个名字，你应该使用这个名字而不是 `Companion`。但是你可能需要使用要求类的某个成员是静态的Java代码。你可以对相应的成员使用 `@JvmField` 标注来实现这个要求。如果你想声明一个 `static` 字段，对一个顶层属性或者声明在对象中的属性使用 `@JvmField`。这些特性的存在，只是为了达到互操作性的目的。严格来讲，不是语言内核的一部分。我们将在第10章详细覆盖标注的内容。注意，Kotlin可以使用跟Java中一样的语法来访问声明在Java类中静态方法和属性。

### 伴生对象扩展

正如你在3.3一节看到的那样，扩展函数允许你定义能够被定义在代码的任意地方的类实例调用的方法。但是，如果你需要定义可以在类的内部被调用的函数，像伴生对象方法或者Java静态方法，该怎么办呢？如果这个类有一个伴生对象，你可以通过为它定义扩展函数来达到这个目的。更具体些是，如果类 `c` 有一个伴生对象，你在类 `c` 中定义了一个扩展函数 `fun`，那么你可以写成 `c.func()` 来调用它。举个例子，想象一下，你想要为你的 `Person` 类有一个更清晰的关注分离。这个类本身将会成为核心业务逻辑模块的一部分。但是你不想把模块跟任何其他具体的数据格式耦合在一起。因此，需要在这个模块定义反序列化函数来负责客户端/服务端通信。你可以使用这个扩展函数来完成这个任务。注意，你如何使用默认名称（`Companion`）来引用伴生对象是不需要显式名称来声明的：

```
// business logic module
class Person(val firstName: String, val lastName: String) {

    companion object { // 声明一个空的伴生对象
    }
}

// client/server communication module

fun Person.Companion.fromJSON(json: String): Person { // 声明一个扩展函数
    ...
}
val p = Person.fromJSON(json)
```

你调用 `fromJSON` 函数，是因为它被定义在伴生对象内部。但它是伴生对象的一个扩展。由于它经常带有扩展函数，伴生对象看上去像一个成员，但它并不是。为了能够给它定义一个扩展，注意，你必须在类的内部声明一个伴生对象，即使是空的。你已经看到了伴生对象可以是多么有用。现在让我们继续下一个特性，在Kotlin中同样用 `object` 关键字来表达：对象表达式。

### 4.4.4 对象表达式：匿名内部类的另一种表达方式

`object` 关键字不仅可以被用在声明像单例那样的命名对象，也能用来声明匿名对象（anonymous objects）。匿名对象取代了Java中使用的匿名内部类。例如，让我们来看看你可以如何将Java匿名内部类的典型用法--事件侦听器--转换成Kotlin：

```
window.addMouseListener(  
  
    object : MouseAdapter() { // 声明一个扩展MouseAdapter的匿名对象  
        override fun mouseClicked(e: MouseEvent) { // 覆盖MouseAdapter的方法  
            // ...  
        }  
  
        override fun mouseEntered(e: MouseEvent) { // 覆盖MouseAdapter的方法  
            // ...  
        }  
    }  
)
```

如你所见，语法跟对象声明一样。除非你忽略对象的名称。对象表达式声明了一个类并创建了这个类的一个实例。但是它并没有为类或类实例分配一个名字。一般来说，两者都是不必要的，因为你把对象用作函数调用中的一个参数。如果你需要为这个对象分配名称，你可以把它存储在一个变量中：

```
val listener = object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) { ... }  
    override fun mouseEntered(e: MouseEvent) { ... }  
}
```

不同于Java匿名内部类，Kotlin匿名对象可以实现多个接口或不实现（尽管后者不是非常有用）。

**NOTE** Note 不像对象声明，匿名对象不是单例。对象表达式每次执行，都会创建这个对象的一个新的实例。

跟Java的匿名类一样，对象表达式中的代码可以访问创建它的函数中的变量。但跟Java不同的是，这并不局限于 `final` 变量。你也可以从一个对象表达式的内部修改变量的值。举个例子，让我们来看一看你可以如何使用侦听器来统计窗口点击的次数：

```
fun countClicks(window: Window) {  
  
    var clickCount = 0 // 声明一个本地变量  
  
    window.addMouseListener(object : MouseAdapter() {  
        override fun mouseClicked(e: MouseEvent) {  
            clickCount++ // 更新变量的值  
        }  
    })  
    // ...  
}
```

**NOTE** 注意 当你需要在你的匿名对象中覆盖多个方法时，对象表达式是最有用的。如果你只需要实现一个单一方法的接口（比如 `Runnable`），你可以依靠Kotlin对SAM转换（将一个函数字面量转换为带有一个单一抽象方法的接口实现）的支持和编写你的实现作为一个函数字面。我们将会在第5章更加详细的讨论lambda和SAM转换。

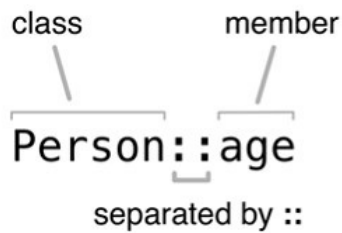
我们已经完成了有关类、接口和对象的讨论。在下一章，我们将会继续Kotlin最有趣的一个领域：`lambda`和函数式编程。

- 在Kotlin中，接口跟Java类似。但是它可以包含默认的实现和属性。
- 所有的声明默认都是 `final` 和 `public` 。
- 为了让一个声明不是 `final` ，把它标记为 `open` 。
- `internal` 声明在同一个模块内是可见的。
- 嵌套类默认不是内部的。使用关键字 `inner` 来为一个外部类保存（它的）一个引用。
- `sealed` 类的子类只能嵌套在它的声明内部。
- 初始化块和次构造函数为初始化类实例提供了灵活性。
- 使用 `field` 标记符来引用一个主体访问器的属性支持字段。
- 数据类提供了编译器生成的 `equals()`, `hashCode()`, `toString()`, `copy()`, 和其他方法。
- 类委托帮你避免了代码中的许多委托方法。
- 对象委托是Kotlin定义一个单例类的方式。
- 伴生对象替代了Java的静态方法和字段定义。
- 伴生对象，跟其他对象一样，可以实现接口或者拥有扩展函数或属性。
- 对象表达式是Kotlin提供的Java匿名内部类的替代品。它增加了有效率的东西，比如实现多个接口的能力以及修改定义在对象创建的地方变量。

这一章覆盖了：

- lambda表达式和成员引用
- 以函数式的风格使用集合
- 序列：延迟执行集合操作
- 在Kotlin中使用Java函数式接口
- 使用带有接收器的lambda

**Lambda表达式(Lambda expressions)**或者**lambda**，本质上是可传递给其他函数的小块代码。使用**lambda**，你可以方便的提取公共的代码结构到库函数中。**Kotlin**标准库大量使用**lambda**。**lambda**最常见的一个用途是跟集合一起使用。在这一章节，你将会看到许多以传递**lambda**给标准库函数（的方式）来取代常见的集合访问模式的例子。你也将会看到**lambda**如何跟**Java**库一同使用--即使这些库最初设计时没有考虑到**lambda**。最后，我们来看一下带有接收器(receiver)的**lambda**。



引入lambda是Java 8这门语言演化历程中期待已久的改变。为什么这是一件大事？在这一章，你将会发现为什么lambda会如此有用，以及lambda表达式的语法在Kotlin中长什么样。

### 5.1.1 Lambda介绍：作为方法参数的代码块

在你的代码中，传递并存储行为块是一个高频的任务。举个例子，你经常需要表达像这样的想法：当某个事件发生时，运行这个句柄。或者是，对某个数据结构中的所有元素应用某个操作。在旧版的Java中，你可以通过匿名内部类来实现。这个技术凑效，但是需要冗余的语法。函数式编程为你解决这个问题提供了一个方法：把函数当做一个值的能力。你可以直接传递一个函数，而不是声明一个类然后把这个类的实例传递给方法。你不需要声明一个函数：相反的，你可以直接有效的传递一个代码块作为一个方法参数。让我们来看一个例子。想象一下你需要定义一个点击一个按钮的行为。你添加了一个侦听器来负责处理点击。侦听器实现了对应的 `OnClickListener` 接口，以及它的 `onClick` 方法：

```
/* Java */
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
        /* actions on click */
    }
});
```

这些冗长的信息要求你声明一个匿名内部类，当它多次重复的时候会让人感到气愤。你想在这表达的只是在点击时应该发生的行为。在Kotlin中，和在Java 8一样，你可以为此使用lambda：

```
button.setOnClickListener { /* actions on click */ }
```

这份Kotlin代码跟Java中的匿名内部类做同样的事情。但是它更加的精简、可读。我们将会在这一节的后续部分讨论这个例子的细节。你已经看到了lambda可以如何只用一个方法来用作匿名对象的一个替代方案。我们现在继续lambda表达式的另一个典型的用法：跟集合一起使用。

### 5.1.2 Lambda与集合



一个好的编程风格的一个主要原则是在你的代码中避免重复。我们使用集合执行的大部分任务都遵循某些通用的模式。所以实现它们的代码应该放入一个库中。但是，没有lambda，为集合提供一个好用、便捷的库就会变得很困难。因此，如果你用Java（Java 8以前的）编写你的代码，你很可能有一个自己实现所有东西的习惯。使用Kotlin，这个习惯必须改掉！让我们来看一个例子。你将会使用一个包含有关一个人的姓名和年龄信息的 `Person` 类：

```
data class Person(val name: String, val age: Int)
```

假定你有一列人，你需要找出其中年纪最大的一个。如果你没有lambda编程经验，你可能会急着去手动实现搜索。你已经引入了两个立即变量--一个保存最大年龄，另一个存储这个年龄对应的人。然后遍历这个列表，更新这两个变量：

```
fun findTheOldest(people: List<Person>) {

    var maxAge = 0                // 存储最大年龄

    var theOldest: Person? = null // 存储最大年龄的人
    for (person in people) {

        if (person.age > maxAge) { // 如果下一个人比当前最年长的人年纪更大，修改最大值
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}

>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

如果有足够的经验，你可以相当快速的实现这样的循环。但是这里的代码有点多。而且也很容易出错。比如，你可能在比较时出错，最后找到的是最小值而不是最大值。Kotlin有更好的方法。你可以使用一个函数库：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age }) // 通过比较年龄找出最大值
Person(name=Bob, age=31)
```

`maxBy` 函数可以在任何集合中被调用。而它只需要一个参数：指定什么样的值应该进行比较来找到最大值元素的函数。大括号中的代码 `{ it.age }` 是一个实现这样的逻辑的lambda。它接受一个集合元素作为一个参数（使用 `it` 引用）。同时返回一个值进行比较。在这个例子中，集合元素是一个 `Person` 对象。比较的值是它存储在 `age` 属性的年龄。如果lambda只是委托一个方法或者属性，它可以被一个成员引用所取代：

```
people.maxBy(Person::age)
```

这段代码表达的东西跟前面的例子相同。我们通常用Java（Java 8之前的版本）集合所做的大部分事情可以使用接受lambda或者成员引用的库函数更好的表达。最后的代码更加简短、更容易理解。为了帮助你开始使用它，让我们来看看lambda表达式的语法。

### 5.1.3 Lambda表达式语法

正如我们已经提及的那样，lambda编码了一小块你可以用作一个值进行传递的行为。它可以被独立的声明并存储在一个变量中。但是更常见的是，当它被传递给一个函数时直接声明。图5.1展示了声明lambda表达式的语法。

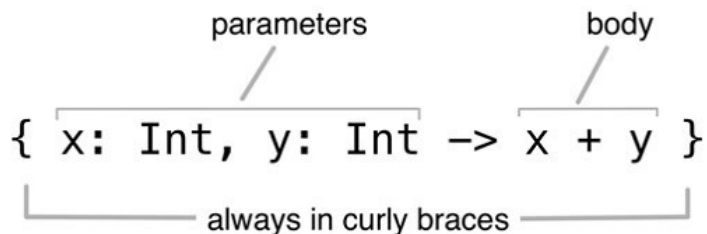


图 5.1 Lambda表达式语法

在Kotlin中，一个lambda表达式往往被一个大括号所包围。注意，参数周围没有圆括号。箭头将参数列表从lambda主体中分隔开来。你可以在一个变量中保存一个lambda表达式。然后把这个变量当做一个正常的函数（使用对应的参数调用它）：

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2)) // 调用存储在一个变量中的lambda
3
```

如果你想的话，你可以直接调用lambda表达式：

```
>>> { println(42) }()
42
```

但是这样的语法并不可读，而且没有多大意义（它等价于立即执行lambda主体）。如果你需要在一个块中包含一块代码，你可以使用执行传递给它的lambda的库函数 `run`：

```
>>> run { println(42) } // 运行Lambda中的代码
42
```

在8.2节，你将会了解为什么这样的调用没有运行时负担，而且跟语言内建的构造有着同样的效率。让我们回到之前查找列表中最年长的人的例子：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

如果你重写了这个例子，而没有使用任何的语法糖，你将会得到下面的代码：

```
people.maxBy({ p: Person -> p.age })
```

这里边发生了什么应该很明显了：大括号里面的代码是lambda表达式。你把它当做一个参数传递给函数。lambda表达式接受一个 `Person` 类型的参数并返回它的年龄。但是这份代码是繁琐的。首先，这里有太多的标点，影响了可读性。第二，类型可以从上下文推断出来，一次可以省略掉。最后，在这种情况下，你不需要为lambda参数分配一个名称。我们来做些提升吧。首先从大括号开始。在Kotlin中，有一个语法约定允许你将lambda表达式移出圆括号外部，如果它是函数调用的最后一个参数。在这个例子中，lambda是唯一的参数，所以它可以放在圆括号外部：

```
people.maxBy() { p: Person -> p.age }
```

当lambda是函数的唯一参数时，你也可以删掉空的圆括号：

```
people.maxBy { p: Person -> p.age }
```

这三种语法形式都表达同样的意思。但是最后一个是最容易阅读的。如果lambda是唯一的参数，你绝对想写成没有圆括号的。当你有多个参数时，你可以通过把它放在圆括号内部来强调lambda是一个参数，或者你把它放在括号外面--两个选项都是有效的。如果你传递两个或更多的lambda，你不能把两个以上的lambda移到圆括号外部。所以通常最好使用常规的语法来传递它们。为了看到看上去带有复杂调用的选项，让我们回到你在第三章大面积使用的 `joinToString` 函数。它也定义在了Kotlin标准库中。不同的是标准库版本接受函数作为一个额外的参数。这个函数可以用来将一个函数转换为一个字符串，但是跟 `toString()` 函数不一样。下面是你可以如何使用它来只打印名字：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> val names = people.joinToString(separator = " ",
...                               transform = { p: Person -> p.name })
>>> println(names)
Alice Bob
```

下面是你可以如何使用圆括号外部的lambda来重写这个调用：

```
people.joinToString(" ") { p: Person -> p.name }
```

第二个变体更加精简地做了同样的事情。但是它没有显式的表达lambda用来做什么。所以对被调用的函数不熟悉的人来说，更加难以理解。

**TIP** IntelliJ IDEA 提示 把一种语法形式转换成其他形式，你可以使用这个动作：“把lambda表达式移到圆括号外面”

我们继续简化成最短的语法，并除去类型参数：

```
people.maxBy { p: Person -> p.age } // 显式的写出参数类型
people.maxBy { p -> p.age } // 参数类型通过推断得出
```

由于带有本地变量，如果lambda参数的类型可以推断出来，你不需要显式的指定。使用 `maxBy` 函数，参数类型总是跟集合元素类型相同。编译器知道你在一个 `Person` 对象集合调用 `maxBy`。所以它可以理解，lambda参数也会是 `Person` 类型。有一些情况下，编译器无法推断lambda参数类型。但是我们不会讨论这些情况。你可以遵循一个简单的规则：总是先不指定类型，如果编译器报错，你就指定类型。你可以仅仅指定其中的某些参数的类型，其他参数只写名字。如果编译器无法推断出某个类型，或者如果显式的类型可以提升可读性，这样做可能会方便一点。在这个案例中，你可以达到的最简形式是使用参数默认名：`it`，来替代一个参数。如果上下文预期衣蛾只有一个参数的lambda，同时它的类型可以推断出来，（编译器）就会生成这个名字：

```
people.maxBy { it.age } // “it”是一个自动生成的参数名
```

只有在你没有显式的指定参数名时，默认名称才会生成。

**NOTE** 注意 `it` 约定对于缩短你的代码是非常有用的。但是你不应该滥用它。特别是，嵌套lambda的情况下，最好是显式的说明每个lambda的参数。否则，难以理解 `it` 指的是哪一个值。如果参数的类型或者含义在上下文中不清晰的话，显式的声明参数也是很有用的。

如果你把lambda保存在一个变量中，那就没有上下文来推断参数类型。所以你必须显式的指定它们：

```
>>> val getAge = { p: Person -> p.age }
>>> people.maxBy(getAge)
```

目前为止，你只看到由一个表达式或语句组成的lambda案例。但是lambda没有局限在这样的一个小规模，而且它也可以包含多个语句。在这个例子中，结果是最后的一个表达式：

```
>>> val sum = { x: Int, y: Int ->
...     println("Computing the sum of $x and $y")
...     x + y
... }
>>> println(sum(1, 2))
Computing the sum of 1 and 2...
3
```

接下来，我们讨论一个经常跟lambda表达式同时出现的概念：从上下文中捕获变量。

## 5.1.4 在作用域内访问变量

你知道，当你在一个方法中声明一个匿名内部类时，你可以在（匿名内部）类的内部通过这个方法引用参数和本地变量。你可以使用lambda做同样的事。如果你在一个函数内使用lambda，你可以和声明在lambda之前的本地变量那样，访问那个函数的参数。为了解释这一点，我们使用标准库函数 `forEach`。它是最基本的集合操作函数之一。它所做的是对集合中的每一个元素调用给定的lambda。 `forEach` 多多少少比常规的 `for` 循环更简洁一些。但是它并没有许多其他的优势。所以你不需要急着把你所有的循环转换为lambda。下面的例子接受一系列消息并打印每一个加了前缀的消息：

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {                // 接受一个指定每个元素做什么的lambda作为参数
        println("$prefix $it")        // 在lambda中访问"prefix"参数
    }
}

>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

Kotlin和Java的一个重要的不同点是：Kotlin没有严格限制你访问 `final` 变量。你也可以从lambda总修改变量。下面的一个案例统计了客户端和服务端在给定响应状态码中的错误个数：

```
fun printProblemCounts(responses: Collection<String>) {

    var clientErrors = 0          // 1

    var serverErrors = 0          // 1
    responses.forEach {
        if (it.startsWith("4")) {
            clientErrors++        // 2
        } else if (it.startsWith("5")) {
            serverErrors++        // 2
        }
    }
    println("$clientErrors client errors, $serverErrors server errors")
}

>>> val responses = listOf("200 OK", "418 I'm a teapot",
...                          "500 Internal Server Error")
>>> printProblemCounts(responses)
1 client errors, 1 server errors
```

// 1 声明通过lambda访问的变量 // 2 修改lambda中的变量

如你所见，跟Java不同，Kotlin允许你访问非不可更改的变量，甚至在lambda中修改它们。通过lambda来访问外部变量，比如案例中的 `prefix`、`clientErrors`、`serverErrors`，可以说被lambda捕获了。注意，本地变量的生命周期默认是限制在声明这个变量的函数内部的。但是如果它被lambda捕获了，使用这个变量的代码可以被存储起来并在后期执行。你可能会问，这是怎么做到的。当你捕获到一个不可修改的变量时，它的值跟用到这个值的lambda代码一同保存。对于不可更改的变量来说，（它的）值依附在一个特殊的包装器内。这个包装器允许你修改它。包装器的引用跟lambda同时存储。

**SIDEBAR** 捕获一个可修改的变量：实现细节 **Java**只允许你捕获不可修改的变量。当你想要捕获可修改变量时，你可以使用以下技巧中的一个：声明一个存储可变值元素的数组，或者创建一个存储可改变引用的包装器类的实例。下面是**Kotlin**中的相似的代码：

```
class Ref<T>(var value: T)    // 用来模拟捕捉一个可修改变量的类
>> val counter = Ref(0)
>> val inc = { counter.value++ } // 形式上，一个不可变变量被捕获了。但是实际的值存储在一个
    字段中而且可以被修改。
```

在**Kotlin**中，你不需要创建这样的包装器。相反的，你可以直接的修改变量：

```
var counter = 0
val inc = { counter++ }
```

这是怎么实现的呢？第一个案例展示了第二个案例的底层工作原理。在任意时刻，你捕获了一个不可修改变量（**val**），它的值就被复制了，就像在**Java**中那样。当你捕获一个可修改变量（**var**）时，它的值被保存为一个 **Ref** 类的实例。**Ref** 变量是不可被修改的。它很容易被捕获。然而真实值保存在一个字段中，并且可以通过**lambda**进行修改。

一个重要的告诫：如果**lambda**用作一个事件句柄，或者异步执行，本地变量的修改只会在**lambda**被执行的时候发生。举个例子，下面的代码不是统计按钮点击次数的正确方式：

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

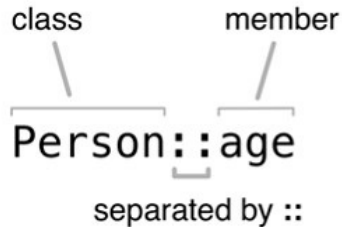
这个函数总会返回0。尽管 **onClick** 句柄会修改 **clicks** 的值，但是你观察不到修改，因为 **onClick** 句柄将会在函数返回之后被调用。这个函数的一个正确的实现，需要保存点击次数到非本地变量，而是在一个依然可以在函数外部进行访问的位置——例如，类的属性。我们已经讨论了声明**lambda**的语法和变量如何在**lambda**中被捕获的。现在，我们讨论成员引用，一个允许你方便的传递引用给现有函数的特性。

## 5.1.5 成员引用

你已经看到了**lambda**是如何允许你向函数传递一块代码作为参数的。但是如果你需要作为参数传递的代码是已经定义了的函数呢？当然，你可以传递一个调用那个函数的**lambda**。但是这样做有点多此一举。你可以直接传递函数吗？跟在**Java 8**一样，如果你把那个函数转换为一个值的话，你可以在**Kotlin**中这样做。你可以为此使用 **::**：

```
val getAge = Person::age
```

这个表达式叫做成员引用。它为创建一个直接调用方法或访问属性的函数值提供了一种简短的语法。双冒号将类名从你需要引用的成员（方法或属性）名中分隔出来。如图5.2所示：



这个更为精简的lambda表达式做了同样的事情：

```
val getAge = { person: Person -> person.age }
```

要注意，不管你引用函数还是属性，你都不应该在函数引用中，为被引用的函数或方法在它的名字后面放一对圆括号。成员引用拥有和调用那个函数的lambda同样的类型。所以你可以互换的使用这两者：

```
people.maxBy(Person::age)
```

你也可以有一个声明在顶层的函数引用（并非类的成员）：

```
fun salute() = println("Salute!")
>>> run(::salute)           // 引用顶层函数
Salute!
```

在这个案例中，你可以忽略类名并以 `::` 为开头。成员引用 `::salute` 作为一个参数传递给库函数 `run`。它将会调用对应的函数。提供一个成员引用非常方便，但是委托给一个接收多个参数的函数却是相反的：

```
val action = { person: Person, message: String -> // 1. 这个lambda委托为sendEmail函数
    sendEmail(person, message)
}

val nextAction = ::sendEmail // 2. 相反，你可以使用一个成员引用
```

你可以使用构造函数引用来保存或者延迟创建类实例的动作。构造函数引用通过在双冒号后面指定类名来构成：



```
data class Person(val name: String, val age: Int)

>>> val createPerson = ::Person          // 创建"Person"类实例的动作被保存为一个值
>>> val p = createPerson("Alice", 29)
>>> println(p)
Person("Alice", 29)
```

注意，你也可以用同样的方式来引用扩展函数：

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

尽管 `isAdult` 并不是 `Person` 类的成员，但是你可以通过引用来访问它，就像你可以访问实例中的成员那样：`person.isAdult()`。

**SIDEBAR** 绑定引用 在Kotlin 1.0中，当你引用类的方法或属性时，在你调用这个引用时，你总是需要提供那个类的实例。Kotlin 1.1计划支持绑定方法引用。这将允许你使用方法引用语法来捕捉特定对象实例中的方法引用。

```
>> val p = Person("Dmitry", 34)
>> val personsAgeFunction = Person::age
>> println(personsAgeFunction(p))
34

>> val dmitrysAgeFunction = p::age // 1 你可以在Kotlin 1.1 使用的绑定方法引用
>> println(dmitrysAgeFunction())
34
```

注意，`personsAgeFunction` 函数只接收一个参数(它返回一个给定的人的年龄)，然而，`dmitrysAgeFunction` 函数没有参数(它返回指定的人的年龄)。在Kotlin 1.1之前的版本，你需要显式的写上 `{ p.age }`，而不是使用绑定成员引用 `p::age`。

在接下来的章节中，我们会看到很多库函数。它们能够跟lambda表达式和成员引用很好的工作。

当你操作集合时，函数式风格提供了许多好处。你可以将库函数用于大部分任务并简化你的代码。在这一章节，我们将会讨论一些用于集合的Kotlin标准库函数。我们主要以`filter`和`map`，以及它们背后的概念为开始。我们也会覆盖其他有用的函数。我们会给你一些有关如何重用它们，以及如何编写更加清晰易读的代码的提示。注意，其中没有一个函数是Kotlin的设计者发明的。这些或类似的函数对于所有支持`lambda`的语言都是可用的。如果你以及对这些概念非常熟悉了，你可以通过下面的例子快速看一眼并跳过其中的解释。

### 5.2.1 必要的:`filter`和`map`

`filter` 和 `map` 函数组成了操作集合的基础。许多的集合请求可以在它们的帮助下进行表达。对每一个函数，我们都将提供一个有数字和熟悉的 `Person` 类的例子：

```
data class Person(val name: String, val age: Int)
```

`filter` 函数变换一个集合，并过滤出不满足给定断言的元素：

```
>>> val list = listOf(1, 2, 3, 4)
>>> list.filter { it % 2 == 0 } // 1 只保留偶数
[2, 4]
```

结果是只包含来自输入集合满足断言的元素的新集合。如图5.3所示。

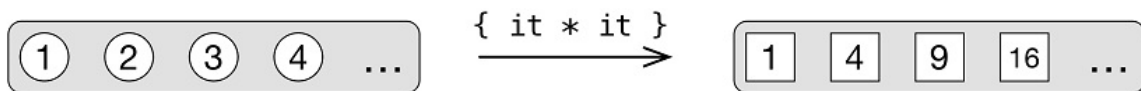


图5.3 过滤器是如何工作的

如果你只想保留年龄大于30的人，你可以使用 `filter`：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> people.filter { it.age > 30 }
[Person("Bob", 31)]
```

`filter` 函数可以从集合中移除不想要的元素。但是它不能修改元素。要修改元素，`map` 就起作用了。`map` 函数对集合中的每个元素应用给定的函数，并且收集结果形成新的集合。例如，你可以把一系列数字变换为它们的立方：

```
>>> val list = listOf(1, 2, 3, 4)
>>> list.map { it * it }
[1, 4, 9, 16]
```

结果是一个包含同样的元素个数，但是每个元素根据给定的断言进行变换的集合（详见图 5.4）。

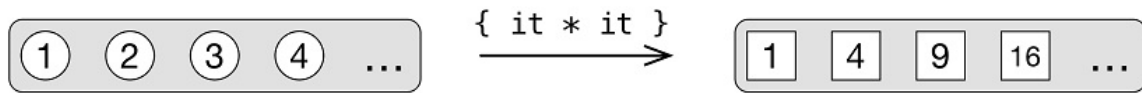


图 5.4 map 函数的工作原理

如果你只是想打印一系列名字，而不是一列人，你可以使用 `map` 来变换列表：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> people.map { it.name }
[Alice, Bob]
```

注意了，这个例子可以使用成员引用完美的重写：

```
people.map(Person::name)
```

你可以方便的将多个调用像这样串成一条链子。例如，我们打印年龄大于30的人的名字：

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

现在，我们假设你需要组中最年长的人的名字。你可以找出组中年纪最大的人。然后返回每一个人的年龄。这样的代码用 `lambda` 写非常简单：

```
people.filter { it.age == people.maxBy(Person::age).age }
```

但是要注意，这份代码为每个人重复了最大年龄的查找步骤。所以，如果集合中有100个人，最大年龄的搜索将会执行100次！下面的解决方案进行了优化。它只计算一次最大年龄：

```
val maxAge = people.maxBy(Person::age).age
people.filter { it.age == maxAge }
```

如果你不需要，就不要重复进行计算。看上去简单的代码，使用 `lambda` 表达式，有时候可以隐藏底层操作的复杂性。所以，你总能记住你写的代码中发生了什么。

你也可以对映射应用过滤和变换函数：

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

Kotlin有函数来分别操作键和值。 `filterKeys` 和 `mapKeys` 过滤并变换映射的键。相应的， `filterValues` 和 `mapValues` 过滤和变换对应的值。

## 5.2.2 `all`, `any`, `count` 和 `find` :对集合应用预言

另一个常见的任务是检查集合中所有的元素是否匹配某个条件。Kotlin中，这可以通过 `all` 和 `any` 函数进行表达。 `count` 函数检查多少个元素满足这个预言。 `find` 函数返回第一个匹配的元素。为了介绍这些函数，我们定义一个语言 `canBeInClub27` 来检查某个人是否为27岁或者更年轻：

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

如果你对所有的元素是否符合这个预言，你可以使用 `all` 函数：

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

如果你需要检查是否至少有一个匹配的元素，使用 `any`：

```
>>> println(people.any(canBeInClub27))
true
```

要注意， `!all` (非全部)可以使用一个否定条件来替代 `any`，反之亦然。为了让你对代码更容易理解，你最好选择一个不需要你在他都前面放置否定符号的函数：

```
>>> val list = listOf(1, 2, 3)
>>> println(!list.all { it == 3 }) // 1 否定符号!并不引人注目。所以，这个案例中使用"any"更加合适

true
>>> println(list.any { it != 3 }) // 2 参数中的条件改成了相反的情况
true
```

第一个检查确保并非所有元素都等于3。它等价于至少有一个不等于3的元素。这就是你在第二行使用 `any` 进行的检查。如果你想知道有多少个元素符合这个预言，使用 `count`：

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

SIDEBAR 为任务使用右边的函数：`count` vs `size` 你很容易忘记了 `count` 的存在。而且你通过过滤集合实现它的功能，得到了它的大小：

```
>>> println(people.filter(canBeInClub27).size)
1
```

在这个案例中，创建了一个临时的集合来保存所有符合预言的元素。另一方面，`count` 方法只追踪匹配的元素个数，而不是元素本身。因此，它会更加高效。作为一个通用的法则，试着去找出满足你的需求的最合适的操作。

为了找出一个符合预言的元素，请使用 `find` 函数：

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

如果有多个元素，函数将返回第一个匹配的元素。如果没有满足的元素，函数返回 `null`。 `find` 的一个同义词是 `firstOrNull`。如果如能够更加清晰的表达你对想法，你可以使用 `firstOrNull` 函数。

### 5.2.3 `groupBy` :把一个列表转换为多组映射

想象一下，你需要根据你某些特性来将所有元素分割成不同的组。例如，你想把年龄相同的人放在一组。把这个特性直接作为一个参数进行传递非常方便！ `groupBy` 函数可以为你做到这一点：

```
>>> val people = listOf(Person("Alice", 31),
...
Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })
```

这个操作的结果是一个映射通过键被分成多个组。见图5.5。

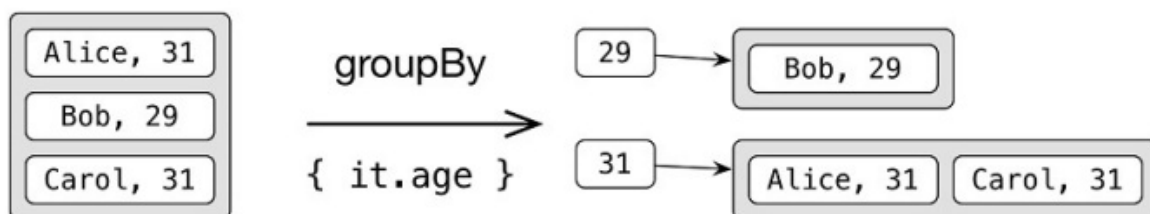


图5.5 应用`groupBy`函数的结果

对于这个案例，输出以下结果：

```
{29=[Person(name=Bob, age=29)],
 31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

每一组都被保存成一个列表。所以结果的类型为 `Map<Int, List<Person>>`。你可以使用像 `mapKeys` 和 `mapValues` 这样的函数对这个映射做更多的修改。

正如另一个例子所示，我们来看看如何使用成员引用通过第一个字符对字符串进行分组：

```
>>> val list = listOf("a", "ab", "b")
>>> println(list.groupBy(String::first))
{a=[a, ab], b=[b]}
```

注意，这里的 `first` 不是 `String` 类的成员。它是一个扩展（函数）。不过你可以把它当做一个成员引用进行访问。

## 5.2.4 flatMap和flatten：处理嵌套集合中的元素

现在，让我们把有关人的讨论放到一边，切换到书的讨论。假定你有一个书库，用 `Book` 类来表示：

```
class Book(val title: String, val authors: List<String>)
```

每一本书有一个或多个作者。你可以计算你的书库中所有作者的集合：

```
books.flatMap { it.authors }.toSet() // 编写书籍的所有作者的集合
```

`flatMap` 函数做了两件事：首先它根据作为参数而给定的函数把每一个元素都变换（或映射）到一个集合中。然后它把多个列表合并为一个。有一个处理字符串的案例很好的解析了这个概念（见图5.6）：

```
>>> val strings = listOf("abc", "def")
>>> println(strings.flatMap { it.toList() })
[a, b, c, d, e, f]
```

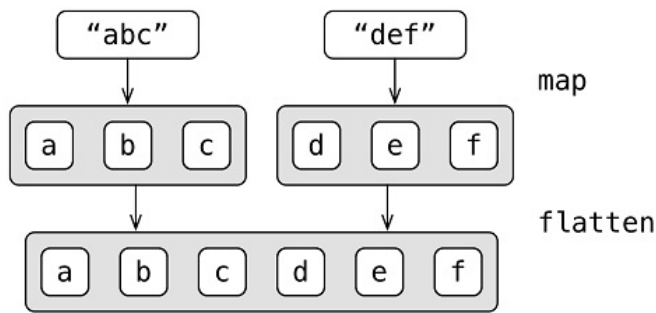


图 5.6 应用flatMap函数后的结果

字符串的 `toList()` 函数将其自身转换为字符的列表。如果你有把 `map` 函数跟 `toList()` 一起使用，如图中的第二行所示，你会得到一个嵌套在列表中的列表的字符串。`flatMap` 函数也做了以下步骤。之后返回一个由所有元素组成的列表。我们来看看返回的作者：

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
Book("Mort", listOf("Terry Pratchett")),
... Book("Good Omens", listOf("Terry Pratchett",
... "Neal Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neal Gaiman]
```

每一本书都由一个或多个作者编写。`book.authors` 属性保存了作者的集合。`flatMap` 函数把所有书的作者都组装进了一个扁平的（没有嵌套的）列表。`toSet` 调用从结果集合中移除了重复的元素。所以，在这个案例中，**Terry Pratchett**只会在打印输出中列举一次。当你陷入必须合并为一个集合的嵌套集合的元素时，你可以考虑 `flatMap`。注意，如果你需要变换任何东西，只是仅仅需要把这样一个集合变得扁平，你可以使用 `flatten` 函

数：`listOfLists.flatten()`。我们重点讲了Kotlin标准库中的一些集合操作函数。但是里边还有更多的函数。处于篇幅的考虑，也因为展示一长串的函数非常的无聊，我们不会面面俱到。我们一般建议，当你编写用到了集合的代码时，考虑如何将操作表达为一个通用的变换。然后查找执行这个变换的库函数。你很可能找到一个这样的函数，并用它来解决你的问题。这往往比你手工实现更加快速。现在，我们来仔细看看集合的链式操作吧！在下一章节，你将会看到操作执行的几种不同方式。

在前面的章节，你看到了多个链式集合函数的例子。比如，`map` 和 `filter`。这些函数会立即创建临时集合。这意味着每一步的临时结果都被保存在一个临时列表中。序列给了你一个可选的方法来完成这样的计算。这样可以避免创建一次性的临时对象。这有一个例子：

```
people.map(Person::name).filter { it.startsWith("A") }
```

Kotlin标准库参考（文档）指出，`filter` 和 `map` 都返回一个列表。这意味着这个链式调用将会创建两个列表：一个保存 `filter` 函数的结果，另一个存储 `map` 函数的结果。当原来的列表只有包含两个元素时，这不会有问题。但是如果你有百万个元素时，这会变得非常低效。为了把它变得更加高效，你可以转换这个操作。请使用序列而不是直接使用集合：

```
people.asSequence()           // 1 把初始集合转换为序列
    .map(Person::name)         // 2 序列支持跟集合同样的API
    .filter { it.startsWith("A") } // 2
    .toList()                  // 3 把结果序列转换为列表
```

应用了这个操作的结果跟之前的案例一样：一个以字母A开头的人名列表。但是在第二个例子中，没有创建保存元素的中间集合。因此，对于大量的元素，性能会有可观的改善。在 Kotlin 中，集合懒操作的入口是 `Sequence` 接口。这个接口仅表示：一系列可以逐个枚举的元素。`Sequence` 只提供了一个方法: `iterator`。你可以用它来从序列中获取元素值。

`Sequence` 接口的长处在于它实现的操作。序列中的元素是延迟计算的。因此，你可以使用序列来高效的执行集合的链式操作，而无需创建集合来保存过程中的中间结果。你可以通过调用扩展函数 `asSequence` 来把任意的集合转换为序列。为什么你需要把序列变回集合呢？如果序列那么好用，使用序列而不是集合岂不是更加方便吗？结果是：有时候是。如果你只需要遍历序列中的元素，你可以直接使用序列。如果你需要使用其他的API方法，例如，通过下标访问元素，那么你需要将序列转换为列表。

**NOTE** 注意 一般来说，无论何时，你在大型集合中有链式操作时，请使用序列。在8.2一节，我们将会讨论为什么在Kotlin中，常规集合的延迟操作是高效的，尽管它会创建中间的集合。但是如果集合包含大量的元素，中间的元素重拍耗时巨大，所以延迟计算更加可取。

由于序列的操作是延迟的，为了执行这些操作，你直接需要遍历序列的元素或者把它转换为集合。下一章节会解释这一点。

### 5.3.1 执行序列操作：中间和最终操作



序列操作分为两类：中间操作和最终操作。中间操作返回另一个序列。这个序列知道如何变换原始序列的集合。最终操作返回一个结果。这个结果可以是集合、集合元素、数字或者任意其他通过初始集合变换得到的序列（见图 5.7）。

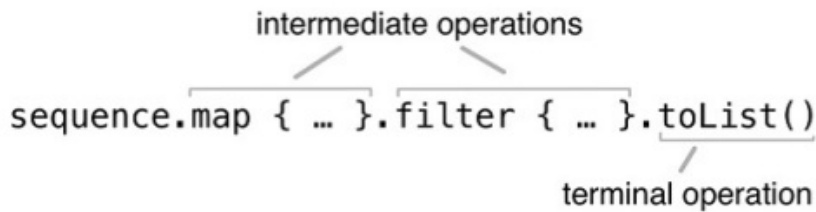


图5.7 集合的中间操作和最终操作

中间操作往往是惰性的。看看这个例子，它没有最终操作：

```
>>> listOf(1, 2, 3, 4).asSequence()
...     .map { print("map($it) "); it * it }
...     .filter { print("filter($it) "); it % 2 == 0 }
```

运行这份代码，控制台里什么也不会打印。这意味着 `map` 和 `filter` 变换延迟了。它们当且仅当获取到结果时执行（当最终操作被调用时）：

```
>>> listOf(1, 2, 3, 4).asSequence()
...         .map { print("map($it) "); it * it }
...         .filter { print("filter($it) "); it % 2 == 0 }
...         .toList()
map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

最终操作导致所有的延迟计算都被执行了。还有一个更重要的事要注意，在这个例子中，计算的执行顺序。原始的方法首先将会对每个元素调用 `map` 函数，然后对结果序列中的每个元素调用 `filter` 函数。这就是 `map` 和 `filter` 在集合上如何工作的。但序列并不是这样的。对于序列来说，所有的操作都会逐个应用于每个元素：处理完第一个元素（映射，然后过滤），然后处理第二个，以此类推。这个方法意味着如果过早获取结果，某些元素根本不会被变换。我们来看一个有 `map` 和 `find` 操作的例子。首先，你把一个数映射为它的平方，之后你查找当中第一个大于3的元素：

[illegible]

如果同样的操作应用于一个集合而不是序列，那么首先会计算 `map` 的结果，变换初始集合中所有的元素。第二步，在中间集合中发现一个满足预言的元素。使用序列，惰性方法意味着你可以提前处理某些元素。图5.8解释了（使用集合）提前和延迟（使用序列）方式执行这份代码的不同点。

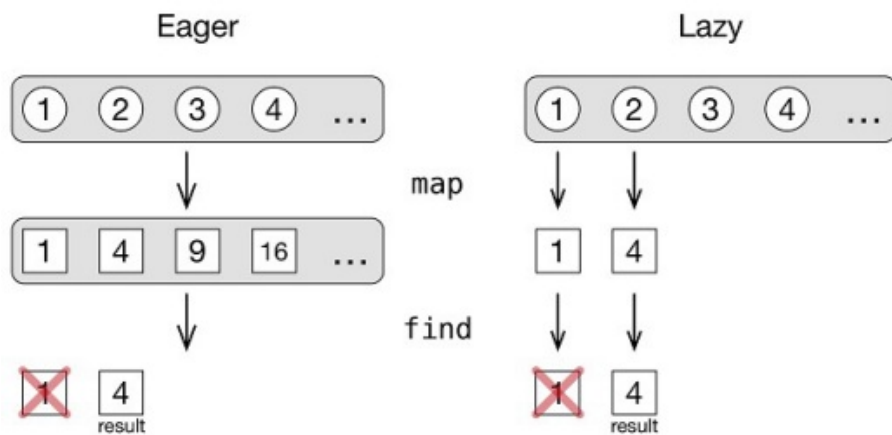


图5.8 提前计算对整个集合运行每一个操作，而惰性求值逐个计算。

在第一个案例中，当你使用集合时，那个列表变换为另一个列表。所以，`map` 变换应用于每一个元素，包括3和4。之后，找到了第一个元素满足预言的元素：2的平方。在第二个案例中，`find()` 调用开始逐个处理元素。你从原始序列中去除一个数字，使用 `map` 对它进行变换。然后检查它是否满足传递给 `find` 的预言。你不需要看3和4了，因为在你到达它们之前，结果已经找到了。你对集合执行的操作顺序也会影响性能。想象一下，你有一个人的集合。你想打印它们的名字，如果它们小于某个值。你需要做两件事：把每个人映射到他们的名字，然后过滤出那些不够短的名字。在这种情形之下，你可以按任意顺序应用 `map` 和 `filter` 操作。两种方法都给出了同样的答案，但是它们需要执行的变换次数不同（见图5.9）：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31),
...      Person("Charles", 31), Person("Dan", 21))
>>> println(people.asSequence().map(Person::name)      // 1 先执行"map", 然后执行"filter"
...      .filter { it.length < 4 }.toList())
[Bob, Dan]
>>> println(people.asSequence().filter { it.name.length < 4 }
...      .map(Person::name).toList())                  // 2 先执行"filter", 然后执行"map"
[Bob, Dan]
```

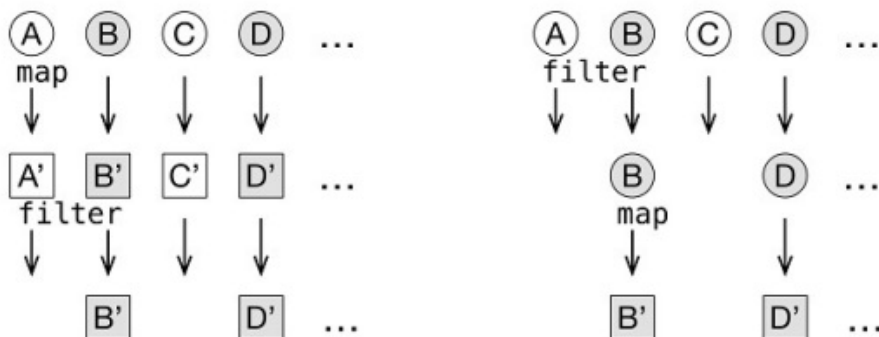


图5.9 先应用`filter`有助于减少变换的总次数

如果先进行 `map`，每个元素都会进行变换。但是如果你先进行 `filter`，不合适的元素会尽快过滤掉，而且不会进行变换。

**SIDEBAR 流 vs 序列** 如果你熟悉Java 8的流，你将会看到，（Kotlin的）序列是完全一样的概念。由于Java 8的流在使用旧版本的Java搭建的平台中无法使用，比如Android，所以Kotlin提供了它自己的轮子。如果你把Java 8作为目标平台，流会给你带来一个很大的特性。而Kotlin的集合与序列并未实现：在多个CPU上并行执行流操作（`map()` 或者 `filter()`）的能力。你可以基于你面向的Java版本和你的具体要求来选择流和序列。

## 5.3.2 创建序列

前面的一个例子使用了同样的方法来创建一个序列：你对集合调用 `asSequence()`。另一个可能是使用 `generateSequence()` 函数。这个函数计算序列中的前一个元素给定的下一个元素。举个例子，这是你可以如何使用 `generateSequence()` 来计算100以内所有自然数的和：

```
>>> val naturalNumbers = generateSequence(0) { it + 1 }
>>> val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }

>>> println(numbersTo100.sum())      // 获取到"sum"结果时，执行所有的延迟操作
5050
```

注意，在这个例子中，`naturalNumbers` 和 `numbersTo100` 都是带有延迟计算的序列。直到你调用最终操作时(在这个例子中是 `sum`)，这些序列中的实数才会进行计算。另一个常见的情形是序列的父类。如果一个元素拥有它的父类（比如人类或者Java文件），也许你会对许多序列的所有祖先感兴趣。在下面的例子中，你通过生成父目录的序列并检查每个目录的熟悉来查询文件是否位于隐藏目录。

```
fun File.isInsideHiddenDirectory() =
    generateSequence(this) { it.parentFile }.any { it.isHidden }

>>> val file = File("/Users/svtk/.HiddenDir/a.txt")
>>> println(file.isInsideHiddenDirectory())
true
```

你又一次通过提供第一个元素和获取每一个子序列元素的方式来创建一个序列。通过使用 `find` 替换 `any`，你将会得到想找的目录。注意，使用序列允许你一旦找到你需要的目录就停止遍历父目录。我们已经详细讨论过lambda表达式的一个常见用途：使用它们来简化集合操作。现在，让我们继续下一个重要的话题：和现有的Java API一起使用lambda。

kotlin与Kotlin库一起使用lambda是非常棒的。但是你用到的大部分API可能是用Java编写的，而不是Kotlin。好消息是，Kotlin的lambda可以完全和Java API进行互操作。在这一章节，你将会完全看到这一点是如何实现的。在这一章的开头，你看到了向Java方法传递lambda的例子：

```
button.setOnClickListener { /* actions on click */ } // 将lambda当做一个参数进行传递
```

Button 类通过接收一个 OnClickListener 类型的 setOnClickListener 方法来为按钮设置了一个新的侦听器：

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
```

OnClickListener 接口声明了一个 onClick 方法：

```
/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

Java 8之前的版本中，你必须创建一个匿名类的实例并把它当做一个参数传递给 setOnClickListener 方法：

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
})
```

在Kotlin中，你可以传递lambda：

```
button.setOnClickListener { view -> ... }
```

用来实现 OnClickListener 的lambda有一个 View 类型的参数，和在 onClick 方法中的一样。图5.10解释了这种对应关系。

```
public interface OnClickListener {
    void onClick(View v);
}
```

→ { view -> ... }

图5.10 lambda的参数对应方法参数

这是有效的，因为 `OnClickListener` 接口只有一个抽象方法。这样的接口叫做函数式接口（*functional interfaces*），或者叫SAM接口。其中SAM表示单一抽象方法（*single abstract method*）。Java API导出都是像 `Runnable` 和 `Callable` 这样的函数式接口，以及接口中的方法。Kotlin允许你在调用接收函数式接口作为参数的Java方法时使用lambda。这确保你的Kotlin代码保持整洁和地道。

**NOTE 注意** 不同于Java，Kotlin拥有真正的函数类型。因此，需要接收lambda作为参数的Kotlin方法应当使用函数类型，而不是函数式接口类型作为这些参数的类型。lambda到实现了Kotlin接口的对象之间的自动转换是不支持的。我们将会在8.1.1节讨论在方法声明中使用函数类型

我们来详细看看当你把lambda传递给一个接收函数式接口类型作为参数的方法时会发生什么。

## 5.4.1 把lambda当做参数传递给Java方法

你可以把lambda传递给任何一个需要一个函数接口的Java方法。例如，思考一下这个接收lambda作为参数的方法：

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

在Kotlin中，你可以调用它并向它传递一个作为参数的lambda。编译器将会自动把它转换为一个 `Runnable` 的实例：

```
postponeComputation(1000) { println(42) }
```

要注意的是，当我们说“一个 `Runnable` 的实例”时，我们的意思是“一个实现了 `Runnable` 接口的匿名类实例”。编译器会为你创建这个实例，并且使用lambda作为唯一的抽象方法--在这个案例中，`run` 方法的主体。你可以通过创建显式实现 `Runnable` 接口的匿名对象来达到这个效果：

```
postponeComputation(1000, object : Runnable // 当你传递一个lambda时，（编译器）会创建匿名类
{ override fun run() {
    println(42)
}
})
```

但是这儿有点不同。当你显式声明一个对象时，每次调用都会创建一个新的实例。使用 `lambda`，情况就不一样了：如果 `lambda` 没有访问定义它的函数的变量，对应的匿名类实例将会在调用之间进行重用：

```
postponeComputation(1000) { println(42) } // 为整个程序创建一个`Runnable`实例
```

因此，下面的代码跟使用 `object` 显式声明是等价的实现。它把 `Runnable` 实例保存在一个变量中，并在每一次调用中使用：

```
val runnable = Runnable { println(42) } // 编译为一个静态变量。程序中的仅有的一个实例。
fun handleComputation() {
    postponeComputation(1000, runnable) // 2 每次`handleComputation`调用都使用同一个对象。
}
```

如果 `lambda` 捕捉来自上下文的变量，那就不再可能为每次调用重用同一个实例了。在这种情况下，编译器会为每次调用创建一个新的对象并保存在该对象中捕获的值。举个例子，下面的函数中，每次调用都使用了一个新的 `Runnable` 实例，并将 `id` 值保存为一个字段：

```
fun handleComputation(id: String) // 在lambda中捕捉“id”变量
{ postponeComputation(1000) // 在每次handleComputation调用中创建一个新的Runnable
实例
    { println(id) }
```

**SIDBAR**    **lambda的实现细节**    从Kotlin1.0开始，每个lambda表达式都会被编译为匿名类，除非它是一个内联lambda。后续的Kotlin版本将会支持生成Java 8字节码。这将会让编译器避免为每个lambda表达式创建一个单独的.class文件。如果lambda进行变量捕捉，匿名类将为每个捕获变量准备一个字段，同时会为每个调用创建这个类的一个新的实例。否则，（编译器）将会创建一个单例。匿名类的名字通过定义lambda的函数的名字添加前缀进行派生，例如：`HandleComputation$1`。如果你反编译了之前的lambda表达式的字节码，你将会看到：

```
class HandleComputation$1(val id: String) : Runnable {
    { override fun run() {
        println(id)
    }
}
fun handleComputation(id: String) {
    postponeComputation(1000, HandleComputation$1(id)) // 底层实现不是一个lambda，而是创建了一个特殊类的实例
}
```

如你所见，编译器为每一个捕获的变量生成一个字段和构造函数参数。

注意了，关于创建匿名类和lambda的类实例的讨论对于函数式接口以外的Java方法是有效的。但是对于使用Kotlin扩展函数的集合是无效的。如果你向标记为 `inline` 的Kotlin函数传递lambda，编译器不会创建匿名类。大部分的库函数都被标记为 `inline`。有关其工作原理的细节，我们会在后续的8.2章节讨论。如你所见，大部分情况下，lambda会自动转换为函数式接口，而无需其他的工作。但是，当你需要显式执行转换时，情况有所不同。我们来看看要如何做。

## 5.4.2 SAM构造函数：lambda变换函数式接口的显式的转换

SAM构造函数是一个由编译器生成的函数。它能让你显式的把lambda转换为函数式接口。当编译器没有自动进行转换时，你可以使用它。举个例子，假如你有一个返回函数式接口的方法，你不能直接返回一个lambda。你必须把它包装成SAM构造函数。下面有一个简单的例子：

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}

>>> createAllDoneRunnable().run()
All done!
```



SAM构造函数的名字跟底层的函数式接口的名字一样。SAM构造函数只接收一个参数--`lambda`。这个`lambda`将被用作函数式接口中的单一抽象方法的主体。同时，它还会返回实现了 `Runnable` 接口的类实例。为了返回一个值，当你需要把从`lambda`生成的函数式接口的实例保存在一个变量时，你会用到SAM构造函数。假定在下面的例子中，你想要为多个按钮重用同一个侦听器（在Android应用中，这份代码会是 `Activity.onCreate` 方法的一部分）：

```
val listener = OnClickListener { view ->

    val text = when (view.id) {           // 1 使用view.id来判断那个按钮被点击了
        R.id.button1 -> "First button"
        R.id.button2 -> "Second button"
        else -> "Unknown button"
    }

    toast(text)                           // 2 向用户展示"text"变量的内容
}
button1.setOnClickListener(listener)
button2.setOnClickListener(listener)
```

`listener` 检查那个按钮是点击的来源并做出相应的行为。你可以通过使用实现了 `OnClickListener` 的对象声明来创建侦听器，但是SAM构造函数给了你一个更精简的选项。

**SIDEBAR**      `lambda`和添加/移除侦听器      注意，由于处在匿名对象内部，`lambda`中没有 `this` 关键字：无法在被转换的`lambda`中引用匿名类实例。从编译器的角度来看，`lambda`是一块代码，不是一个对象。你不能把它引用为一个对象。`lambda`中的 `this` 引用指向包围它的类。如果你的事件侦听器在处理事件时，需要将自身取消订阅，你无法为此使用`lambda`。相反的，你应该使用一个匿名对象来实现侦听器。在匿名对象中，`this` 关键字指向该对象的实例。你可以把它传递给移除侦听器的API。

还有，尽管SAM在方法转换时转换通常会自动执行，但是也有例外。在你把`lambda`当做参数传递给一个重载函数时，编译器无法选择正确的重载函数。在这种情况下，应用SAM构造函数是解决这个编译器错误的好方法。为了完成我们对`lambda`语法和用法的讨论，我们来看看有接收器的`lambda`以及如何用它们来定义看起来想内置语法的好用的库函数。



这一章节介绍Kotlin标准库中的 `with` 和 `apply` 函数。这些函数非常的方便。你将会发现想它们的许多用处，甚至无需理解如何什么它们。在后面的11.2.1节，你将会看到如何为你的需要声明一个相似的函数。但是，这一节的解释却是帮助你熟悉Kotlin的独有特性。这个特性对Java不可用：调用lambda内部的不同对象的方法，而无需额外的修饰符的能力。这样的lambda叫做带有接收器的lambda(*lambdas with receivers*)。我们来看看使用了带有接收器的lambda的 `with` 函数。

## 5.5.1 with函数

许多语言都有这样的一个特殊语句：你可以用来对同一个对象执行多个操作而无需重复这个对象的名字。Kotlin也有这样的特性。但是它作为一个叫做 `with` 的库函数来提供，而不是语句。来看看它是如何个有用法。考虑下面的例子，你将会使用 `with` 重构它：

```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
        result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

在这个例子中，你在 `result` 实例中调用了多个不同的方法。你在每一个调用中重复了 `result`。这还不算太糟。但是，如果你用到的表达式更长或者重复的更加频繁呢？下面是用 `with` 重写的代码：

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()

    return with(stringBuilder) {
        for (letter in 'A'..'Z') {
            this.append(letter)
        }

        append("\nNow I know the alphabet!")

        this.toString()
    }
}
```

// 1 在你调用的方法中制定接收器的值

// 2 通过显式的“this”调用接收器值的方法

// 3 调用方法，忽略“this”

// 4 从lambda中返回值



- `lambda`允许你将代码块当做参数传递给函数
- Kotlin允许你在圆括号外部向方法传递`lambda`。并且可以通过 `it` 来引用单个的`lambda`参数。
- `lambda`中的代码可以访问和修改含有对自身调用的函数中的变量
- 你可以通过为函数名添加前缀 `::` 来创建方法、构造函数和属性的引用。你也可以把这个应用传递给方法而不仅仅是`lambda`。
- 集合最常见的操作无需手动遍历就可以实现，可以使用像 `filter, map, all, any` 等函数。
- 序列允许你合并一个集合上的多个操作而无需创建集合来保存中间结果。
- 你可以把`lambda`当做一个参数传递给Java方法。这个方法接收一个函数式接口（一个带有抽象方法的接口，也就是常说的SAM方法）作为参数。
- 带有接收器的`lambda`，是一种你可以在一个特殊的接收器对象中直接调用方法的`lambda`。
- 标准库函数 `with` 允许你对同一个对象调用多个方法而无趣重复对这个对象的引用。 `apply` 允许你使用建造者风格的API来创建并初始化任意对象。



















