目录

Kotlin极简教程 1.1
前言 1.2
视频教程 1.3
    Kotlin 系统入门到进阶 1.3.1
    Kotlin 从入门到放弃 1.3.2
    Kotlin 从零基础到进阶 1.3.3
第1章 Kotlin简介 1.4
第2章 快速开始：HelloWorld 1.5
第2章 快速开始：SpringbootRestful 1.6
第3章 Kotlin语言基础 1.7
第4章 基本数据类型与类型系统 1.8
第5章 集合类 1.9
第6章 泛型 1.10
第7章 面向对象编程 1.11
第8章 函数式编程 1.12
第9章 轻量级线程：协程 1.13
第10章 Kotlin与Java互操作 1.14
第11章 使用Kotlin集成SpringBoot开发Web服务端 1.15
第12章 使用Kotlin集成Gradle开发 1.16
第13章 使用 Kotlin 和 Anko 的Android 开发 1.17
第14章 使用 Kotlin DSL 1.18
第15章 Kotlin 文件IO操作与多线程 1.19
第16章 使用 Kotlin Native 1.20

Kotlin极简教程

> 十年生死两茫茫，不思量，自难忘，华年短暂，陈辞岁月悠悠伤，
>
> 满腔热血已芜荒，展未来，后生强，战战兢兢，如履薄冰心彷徨，
>
> 青丝化雪、鬓角成霜，已是英雄迟暮，人生怎慷慨激昂？

对于一个开发者而言，能够胜任系统中任意一个模块的开发是其核心价值的体现。

对于一个架构师而言，掌握各种语言的优势并可以运用到系统中，由此简化系统的开发，是其架构生涯的第一步。

对于一个开发团队而言，能在短期内开发出用户满意的软件系统是起核心竞争力的体现。

每一个程序员都不能固步自封，要多接触新的行业，新的技术领域，突破自我。

GitHub托管主页：https://github.com/JackChan1999/EasyKotlin

GitBook在线阅读、电子书下载：https://www.gitbook.com/book/alleniverson/easykotlin/details

Kotlin 语言中文站

Kotlin 语言官方参考文档 中文版

> https://www.gitbook.com/book/hltj/kotlin-reference-chinese/details

《Kotlin for android developers》中文版翻译

> GitHub：https://github.com/wangjiegulu/kotlin-for-android-developers-zh
>
> 英文原版：https://leanpub.com/kotlin-for-android-developers
>
> GitBook（电子书下载、在线阅读）

目录

前言	1.1
写在前面	1.2
关于本书	1.3
这本书适合你吗？	1.4
关于作者	1.5
介绍	1.6
什么是Kotlin？	1.6.1
我们通过Kotlin得到什么	1.6.2
准备工作	1.7
Android Studio	1.7.1
安装Kotlin插件	1.7.2
创建一个新的项目	1.8
在Android Studio中创建一个项目	1.8.1
配置Gradle	1.8.2
把MainActivity转换成Kotlin代码	1.8.3
测试是否一切就绪	1.8.4
类和函数	1.9
怎么定义一个类	1.9.1
类继承	1.9.2
函数	1.9.3
构造方法和函数参数	1.9.4

关注我

- Email：619888095@qq.com
- CSDN博客：Allen Iverson
- 新浪微博：AndroidDeveloper
- GitHub：JackChan1999
- GitBook：alleniverson
- 个人博客：JackChan

前言

Kotlin是JetBrains团队开发的一门现代的、注重工程实用性的静态类型编程语言，JetBrains团队以开发了世界上最好用的IDE而著称。Kotlin于2010年推出，并在2011年开源。Kotlin充分借鉴汲取了Java、Scala、Groovy、C#、Gosu、JavaScript、Swift等多门杰出语言的优秀特性，语法简单优雅、表现力丰富、抽象扩展方便、代码可重用性好，同时也支持面向对象和函数式编程的多范式编程。Kotlin可以编译成Java字节码运行在JVM平台和Android平台，也可以编译成JavaScript运行在浏览器环境，而且还可以直接编译成机器码的系统级程序，直接运行在嵌入式、iOS、MacOS/Linux/Windows等没有JVM环境的平台。Kotlin源自产业界，它解决了工程实践中程序设计所面临的真实痛点，例如，类型系统可以避免空指针异常的问题。

我最早是被Kotlin的下面这段代码所吸引：

```kotlin
package com.easy.kotlin

fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0
fun length(s: String) = s.length
fun main(args: Array<String>) {
    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")
    println(strings.filter(oddLength))
}
```

13行。

这大约是在三年前，当时我在学习 Java 8 中的函数式编程以及 Lambda 表达式等新特性。那时，我也对Scala、Groovy、Clojure、Haskell等技术很感兴趣，简单学习了部分知识。在这样伴随着兴趣的学习过程中，我无意中看到了上面那段Kotlin代码，第一眼看到这么优雅的函数式编程风格，尤其是compose函数的定义实现，我被深深地吸引了。

Swift 使用func关键字声明函数多个 c 怪怪的，Groovy、Scala 等使用 def 跟函数本义联想不直接，JavaScript 中使用 function 关键字又显得死板了些。而 Kotlin 中的 fun 则简单优雅地恰到好处，关键还让人自然联想到 “乐趣、开心的、使人愉快的” 这样的意思。使用 Kotlin每写一个函数都是充满乐趣的。

我们不妨来看看同样的逻辑实现，如果我们使用 Java 8来写，代码如下所示

```java
package com.easy.kotlin;

import java.util.ArrayList;
import java.util.List;
```

```
interface G<A, B> {
    B apply(A a);
}

interface F<B, C> {
    C apply(B b);
}

interface FG<A, B, C> {
    C apply(A a);
}

public class ComposeFunInJava<A, B, C> {
    public static void main(String[] args) {
        G<String, Integer> g = (s) -> s.length();
        F<Integer, Boolean> f = (x) -> x % 2 != 0;
        FG<String, Integer, Boolean> fg = (x) -> f.apply(g.apply(x));

        List<String> strings = new ArrayList();
        strings.add("a");
        strings.add("ab");
        strings.add("abc");
        List<String> result = new ArrayList();
        for (String s : strings) {
            if (fg.apply(s)) {
                result.add(s);
            }
        }
        System.out.println(result);
    }
}
```

36行，差不多是 Kotlin 的3倍。

我们知道，Java是一门非常优秀的面向对象语言。但是在函数式编程方面，跟其他函数语言相比，还是显得有些笨重与生涩，并且其内在体现出来的思想，依旧是面向对象的思想。

而功能强大的Scala 语言，复杂性却相对较高，学习成本也远高于 Kotlin。 另外，Scala 与 Java 的互操作性没有 Kotlin 好。所以，如果我们既想方便地流畅地使用 Java 的强大且完善的生态库，又想使用更加先进的编程语言的特性，无疑 Kotlin 是个非常不错的选择。

我立马进入了Kotlin的世界。

Kotlin之前一直是默默无闻的，直到今年（2017）5.18 Google IO 大会上，Google宣布正式支持Kotlin为Android的官方开发语言，而且从Android Studio 3.0开始，将直接内置集成Kotlin而无需安装任何的插件。另外，在Spring 5.0 M4 中也引入了对Kotlin专门的支持。

在学习和使用Kotlin中，我发现我越来越喜欢 Kotlin，它是一门非常优秀、优雅有趣、流畅实用的语言，绝对值得一试。感谢Kotlin团队！

本书可以说是我对Kotlin的使用和思考过程的粗浅总结。通过本书的写作，加深了我对 Kotlin语言及其编程的理解，我深刻体会到了学无止境的含义。写书的过程也是我系统学习与思考Kotlin的过程，如果本书能够对你有所帮助，将不胜欣慰。

如何阅读本书

本书共16章。我们希望通过简练的表述，全面介绍Kotlin语言特性以及如何使用Kotlin进行实际项目开发，主要包括如下内容：

- 快速开始 Hello World
- 基础语法
- 基本数据类型和类型系统
- 常用数据结构集合类
- 面向对象编程和函数式编程
- 协程
- Kotlin与 Java 互操作
- 集成 SpringBoot 进行服务端开发
- 使用 Kotlin DSL
- 文件IO操作与多线程
- 使用 Kotlin Native以及与 C 语言互操作

第1章是Kotlin 语言的简介，快速学习Kotlin的环境搭建以及常用工具的使用。该章最后还给出一个编程语言学习的小结。通过该章的学习，能够快速进入Kotlin的世界。

第2章是快速开始 Hello World，分别给出了使用命令行REPL、可执行应用程序、Web RESTFul、Android、Kotlin JavaScript等平台环境上的HelloWorld示例。通过该章的学习，可以快速体验在多平台上使用Kotlin语言进行开发的过程。

第3章介绍Kotlin语言的基础知识，包括Kotlin语言的关键字与标识符等、表达式与流程控制、运算操作符、函数及其扩展等基本内容。

第4章介绍Kotlin语言的基本类型和类型系统。我们会首先简单介绍类型的基本概念，然后具体介绍Kotlin 的内置基本类型：数字、字符串、布尔、数组等。 接着介绍 Kotlin 中引入的特殊的可空类型。最后，简单介绍了Kotlin中的类型推断与类型转换的相关内容。

第5章介绍Kotlin标准库中的集合类：List、Set、Map。Kotlin 中提供了不可变集合类与可变集合类。通过该章的学习，我们将了解到Kotlin 是如何扩展的Java集合库，使得写代码更加简单容易。

第6章介绍Kotlin泛型的基本概念、型变以及类型边界等内容，同时简单介绍了泛型类与泛型函数。

第7章介绍Kotlin面向对象编程的特性： 类与构造函数、抽象类与接口、继承以及多重继承等基础知识，同时介绍了Kotlin中的注解类、枚举类、数据类、密封类、嵌套类、内部类、匿名内部类等特性类。最后我们学习了Kotlin中对单例模式、委托模式的语言层面上的内置支持：object对象、

委托。

第8章介绍Kotlin函数式编程的相关内容，其中重点介绍了Kotlin中的高阶函数、Lambda表达式、闭包等核心语法，并给出相应的实例说明。还探讨了关于Lambda演算、Y组合子与递归等函数式编程思想等相关内容。

第9章介绍Kotlin中的协程。首先引入了协程的基本概念，然后通过一些基础使用实例来学习有关协程的创建、执行、取消等操作的方法。在该章的后半部分，我们主要探讨挂起函数的组合执行、协程上下文与调度器、通道与管道等相关内容。最后，我们对协程与线程进行了简单比较，简要介绍了Kotlin的协程API库。

第10章介绍Kotlin与Java的互操作。

第11章介绍如何使用Kotlin集成Spring Boot、SpringMVC等框架来开发Web服务端应用，给出了一个完整的开发实例。最后，简单介绍了Spring 5.0中对Kotlin的支持特性。

第12章介绍使用 Kotlin 集成Gradle 开发的相关内容。

第13章通过一个具体的Android开发实例，来一起学习使用 Kotlin 开发 Android 应用的具体方法。其中用到了Anko、ButterKnife、Realm 等相关框架。

第14章介绍Kotlin中DSL的相关内容。我们将会看到Kotlin 的扩展函数和高阶函数（Lambda 表达式）特性，为定义Kotlin DSL提供了极大的支持。使用DSL的代码风格，可以让我们的程序更加直观易懂、简洁优雅。如果使用Kotlin来开发项目的话，我们完全可以去尝试一下。

第15章介绍Kotlin文件IO 操作、正则表达式以及多线程相关的内容。

第16章简单介绍了Kotlin Native，该章给出了一个简单的 Hello World 的 Native 实例，同时给出了Kotlin 与 C 语言互操作的完整实例。

谁适合阅读本书

本书适合于所有程序员，不管你是前端开发者、Android/iOS 开发者，还是Java 开发者、C 语言开发者，等等。 如果你目前还不是程序员，但想进入编程世界，那么可以尝试从Kotlin开始学习。虽然本书中的部分内容需要一定的编程基础，但是Kotlin本身的极简特性能激发你对编程的兴趣。

代码下载

基本上在每章末尾，我们都附上了该章示例工程源代码地址。这些源码都在 https://github.com/EasyKotlin 。可以根据需要，自由克隆下载学习。

致谢

在本书的写作出版过程中，得到了很多人的帮助和陪伴。

首先要感谢的是我的妻子和两个可爱的孩子。正是有了你们的陪伴，我的生活才更加有意义。写到这里，我脑海里不禁浮现了我的父亲母亲的音容笑貌，我要感谢我的父母。虽然你们可能不知道我写的东西是什么，但是正是有了你们的辛勤养育，我才能长成今天的我。

我要衷心地感谢我的编辑吴怡女士。当收到她的邀请让我写本关于Kotlin的书时，倍感荣幸之余又感到诚惶诚恐，生怕写出的东西太粗浅。在本书的写作修改过程中，她耐心细致地对稿件进行了详尽、细致的审阅和批注，还提出了很多宝贵的修改建议。同时，在写作过程中也给予了我极大的鼓励，才使我快速完成了这本书。感谢本书出版过程中的所有付出辛勤劳动的华章公司工作人员。

在此，我还要特别感谢我们公司的技术大牛雷卷（陈立兵）。非常感谢他能够抽出宝贵时间审阅本书，同时给出了本书内容的勘误，最后，还为本书写了序。真的非常感谢！

我还要感谢在我的工作学习中认识的所有朋友和同事们，能够认识你们并跟你们一起学习共事，是我的荣幸。

联系我们

虽然在本书写作与修改的过程中，我们竭尽全力追求简单正确、清晰流畅地表达内容，但是限于自身水平和有限的时间，也许仍有错误与疏漏之处，还望各位读者不吝指正。

关于本书的任何问题、意见或者建议都可以通过邮件 universsky@163.com 与我交流。

快乐生活，快乐学习，快乐分享，快乐实践出真知。

最后，祝大家阅读愉快！

陈光剑

2017年8月于杭州

视频教程

- Kotlin 系统入门到进阶
- Kotlin 从入门到放弃
- Kotlin 从零基础到进阶

Kotlin 系统入门到进阶

名称	修改日期
code	2017/6/1 14:37
Kotlin-Tutorials-master	2017/6/7 6:42
第1章 课程介绍	2017/9/12 15:06
第2章 数据类型	2017/9/12 15:06
第3章 程序结构	2017/9/12 17:20
第4章 面向对象	2017/9/12 15:07
第5章 高阶函数	2017/9/12 15:07
第6章 领域特定语言 DSL	2017/9/12 15:07
第7章 协程 Coroutine	2017/9/12 19:27
第8章 Kotlin与 Java 混合开发	2017/9/12 17:22
第9章 Kotlin的应用与展望	2017/9/12 18:49

第1章 课程介绍

本章主要介绍什么是Kotlin，课程安排，以及开发环境的配置。

第2章 数据类型

本章主要讲解 Kotlin 的基本词法，从类型系统入手为大家介绍 Kotlin 中都存在哪些类型，以及相关的特性。（知识点：基本类型、类与对象、空类型、智能类型转换、包、区间、数组）

第3章 程序结构

本章主要讲解 Kotlin 的句法，从程序结构入手为大家介绍 Kotlin 有哪些表达式，有哪些语句，如何定义常量、变量以及函数等等。（知识点：常量与变量 、函数、Lambda表达式、类成员、运算符、分支表达式、循环语句、异常捕获 、函数的具名参数、函数的变长参数、函数的默认参数、案例：一个命令行计算器、导出可执行程...

第4章 面向对象

本章深入探讨 Kotlin 的面向对象的知识，包括抽象、继承，扩展成员、属性代理以及常见类的概念及特性。（知识点：面向对象的基本概念、抽象类与接口、 子承父业的故事、类及其成员的可见性、object、伴生对象与静态成员、方法重载和默认参数、扩展成员、属性代理、数据类、内部类、枚举、密封类）...

第5章 高阶函数

本章深入探讨 Kotlin 的高阶函数的知识，学习常见的内置高阶函数的用法，并对常见高阶函数的相关概念如闭包、复合、科里化等做简要介绍。（知识点：基本概念、常见高阶函数、尾递归优化、闭包、函数复合、科里化 Currying、 偏函数、一个统计字符个数的小例子）...

第6章 领域特定语言 DSL

本章介绍领域特定语言 DSL 的概念，以及如何使用 Kotlin 编写 DSL。（知识点：DSL 的基本概念、案例：HTML DSL、Gradle Kotlin 脚本）

第7章 协程 Coroutine

本章介绍 Kotlin 的协程，主要包括基本 API 的使用，协程执行流程的分析，以及协程相关应用的案例和框架介绍。（知识点：基本概念、案例：异步下载图片、 协程的原理剖析、序列生成器、Kotlinx.coroutines框架简介）

第8章 Kotlin与 Java 混合开发

本章主要对 Kotlin 与 Java 混合开发常见的问题进行了梳理。（知识点：基本互操作、SAM转换、正则表达式、集合框架、IO操作、装箱和拆箱、注解处理器）

第9章 Kotlin的应用与展望

本章主要为大家演示如何编写 Kotlin 脚本，如何开发服务端程序， 如何开发 Android 应用，如何开发前端程序，以及如何使用 Kotlin-Native 将 Kotlin 直接编译为可执行程序并在操作系统上直接运行。（知识点：Kotlin的应用场景、 Kotlin-Script的例子、Kotlin-Android的例子、Kotlin-JavaScript的例子、 Kotlin-springboot的例子、Kotlin-Native 的例子)

Kotlin 从入门到放弃

名称	日期	类型	大小	长度
Kt01 Kotlin简介.mp4	2016/10/17 9:53	媒体文件(.mp4)	134,115 KB	00:06:06
Kt02 Helloworld.mp4	2016/10/12 12:28	媒体文件(.mp4)	77,021 KB	00:08:31
Kt03 基于Gradle的工程示例.mp4	2016/10/12 12:34	媒体文件(.mp4)	455,340 KB	00:26:11
Kt04 集合遍历 map.mp4	2016/10/17 9:55	媒体文件(.mp4)	134,271 KB	00:11:00
Kt05 扁平化集合 flatMap.mp4	2016/10/17 9:55	媒体文件(.mp4)	92,683 KB	00:08:27
Kt06 枚举、When表达式.mp4	2016/10/17 9:52	媒体文件(.mp4)	131,254 KB	00:11:01
Kt07 在 RxJava 中使用 Lambda.mp4	2016/10/26 11:49	媒体文件(.mp4)	199,740 KB	00:13:10
Kt08 使用 Retrofit 发送 GET 请求.mp4	2016/11/2 20:23	媒体文件(.mp4)	110,351 KB	00:11:27
Kt09 尾递归.mp4	2016/11/11 14:46	媒体文件(.mp4)	113,732 KB	00:10:38
Kt10 单例（fixed）.mp4	2017/1/12 9:23	媒体文件(.mp4)	109,144 KB	00:10:06
Kt11 Sealed Class.mp4	2016/11/28 13:12	媒体文件(.mp4)	85,813 KB	00:07:52
Kt12 Json数据引发的血案.mp4	2016/12/5 13:26	媒体文件(.mp4)	112,403 KB	00:14:45
Kt13 kapt 及它的小伙伴们.mp4	2016/12/12 11:14	媒体文件(.mp4)	74,572 KB	00:14:13
Kt14 Kotlin 与 Java 共存 (1).mp4	2016/12/19 14:31	媒体文件(.mp4)	63,225 KB	00:17:35
Kt15 Kotlin 与 Java 共存 (2).mp4	2016/12/19 14:31	媒体文件(.mp4)	60,545 KB	00:19:30

随着Kotlin越来越成熟稳定，我已经开始在生产环境中使用它。考虑到目前国内资料较少，我录制了一套视频教程，希望以此抛砖引玉，让 Kotlin 在国内火起来。

01 Kotlin 简介

简要介绍下什么是 Kotlin，新语言太多了，大家为什么要接触 Kotlin 呢？因为它除了长得与 Java 不太像以外，其他的都差不多

02 Hello World

千里之行，始于Hello World！

03 使用Gradle编写程序简介（可选）

这年头，写 Java 系的代码，不知道 Gradle 怎么行呢？

04 集合遍历 map

放下 i++，你不知道 map 已经占领世界了么？以前我以为 map-reduce 很牛逼，后来才知道就是数据迭代处理嘛。

05 集合扁平化 flatMap

这个可以说是 map 的一个加强版，返回的仍然是开一个可迭代的集合，用哪个您自己看需求~

06 枚举类型与When表达式

Kotlin 丢掉了 switch，却引进了 when，这二者看上去极其相似，不过后者却要强大得多。至于枚举嘛，还是 Java 枚举的老样子。

07 在 RxJava 中使用 Lambda

这一期通过一个统计文章中字符数的小程序进一步给大家呈现 Lambda 的威力，也向大家展示一下如何在 Kotlin 当中优雅地使用 RxJava。我不做教科书，所以如果大家对概念感兴趣，可以直接阅读官方 API

RxJava 是一个非常流行的 Java Reactive 框架，函数式的数据操作使得 Lambda 表达式可以充分体现自己的优势，比起 Java 的冗长，你会看到一段非常漂亮简洁的代码。建议大家先阅读 RxJava 的官方文章以对其有一些基本的认识。

08 使用 Retrofit 发送 GET 请求

Retrofit 是 Square 的 Jake 大神开源的RESTful 网络请求框架，用它发送请求的感觉会让你感觉爽爆的。我这里还有几篇文章，以及一个我 hack 过的分支 HackRetrofit，有兴趣的童鞋可以一起探讨下~

- Android 下午茶：Hack Retrofit 之 增强参数
- Android 下午茶：Hack Retrofit (2) 之 Mock Server
- 深入浅出 Retrofit，这么牛逼的框架你们还不来看看？

09 尾递归优化

尾递归，顾名思义，就是递归中调用自身的部分在函数体的最后一句。我们知道，递归调用对于栈大小的考验是非常大的，也经常会因为这个导致 StackOverflow，所以尾递归优化也是大家比较关注的一个话题。Kotlin 支持语法层面的尾递归优化，这在其他语言里面是不多见的。

10 单例

单例大家一定都不陌生，只要你动手写一个程序，就免不了要设计出一些全局存在且唯一的对象，他们就适合采用单例模式编写。在 Java 里面，单例模式的写法常见的有好几种，虽然简单却也是涉及了一些有意思的话题，那么在 Kotlin 当中我们要怎么设计单例程序呢？

11 Sealed Class

枚举类型可以很好的限制一个类型的实例个数，比如 State 枚举有两种类型 IDLE 和 BUSY 两种状态，用枚举来描述再合适不过。不过，如果你想要设计子类个数有限的数据结构，比如指令，指令的类型通常是确定的，不过对于某些有参数的指令每一次都使用同一个实例反而不合适，这时候就需要 Sealed Class。

12 Json数据引发的血案

Json 数据可真是大红大紫一番，它实在是太容易理解了，随着 Js 的火爆它就更加『肆无忌惮』起来。我们在 Java/Kotlin 当中解析它的时候经常会用到 Gson 这个库，用它来解析数据究竟会遇到哪些问题？本期主要围绕 Json 解析的几个小例子，给大家展示一下 Java/Kotlin 的伪泛型设计的问题，以及不完整的数据的解析对语言本身特性的冲击。

13 kapt 以及它的小伙伴们

首先感谢 @CodingPapi，这一期的内容主要来自于他的建议。

Kotiln 对于注解的支持情况在今年（2016）取得了较大的成果，现在除了对 @Inherited 这个注解的支持还不够之外，试用了一下没有发现太大的问题。关于 kapt，官方的文章罗列下来，其中

- kapt: Annotation Processing for Kotlin 已经过时了，大家可以阅读下了解其中提到的三个方案
- Better Annotation Processing: Supporting Stubs in kapt 提到的实现其实基本上就是现在的正式版
- Kotlin 1.0.4 is here 提到了 kapt 的正式发布，需要注意的是，kapt 的使用方法有些变化，需要 `apply plugin: 'kotlin-kapt'`

本期主要通过一个简单的 Dagger2 实例给大家展示了注解在 Kotlin 当中的使用，看上去其实与在 Java 中使用区别不大，生成的源码也暂时是 Java 代码，不过这都不重要了，反正是要编译成 class 文件的。

后面我们又简单分析了一下 Dagger2 以及 ButterKnife 的源码（有兴趣的话也可以看下我直接对后者进行分析和 Hack 的一篇文章：深入浅出 ButterKnife，听说你还在 findViewById？），其实自己实现一个注解处理器是非常容易的，类似的框架还有androidannotations，它的源码大家可以自行阅读。

通过这个例子，我们其实发现 kapt 还是有一些不完善的地方，主要是：

1. 不支持 @Inherited
2. 生成的源码需要手动添加到 SourceSets 中
3. 编译时有时候需要手动操作一下 gradle 的 build 才能生成源码（这一点大家注意下就行了，我在视频中并没有提到）

不过总体来讲，kapt 的现状还是不错的，相信不久的将来这些问题都将不是问题。

---

14 Kotlin 与 Java 共存 (1)

你想要追求代码简洁、美观、精致，你应该倾向于使用 Kotlin，而如果你想要追求代码的功能强大，甚至有些黑科技的感觉，那 Java 还是当仁不让的。

说了这么多，还是那句话，让他们共存，各取所长。

那么问题来了，怎么共存呢？虽然一说理论我们都知道，跑在 Jvm 上面的语言最终都是要编成 class 文件的，在这个层面大家都是 Java 虚拟机的字节码，可他们在编译之前毕竟还是有不少差异的，这可如何是好？

正所谓兵来将挡水来土掩，有多少差异，就要有多少对策，这一期我们先讲**在 Java 中调用 Kotlin**。

15 Kotlin 与 Java 共存 (2)

上一期我们简单讨论了几个 Java 调用 Kotlin 的场景，这一期我们主要讨论相反的情况：如何在 Kotlin 当中调用 Java 代码。

除了视频中提到的点之外还有一些细节，比如异常的捕获，集合类型的映射等等，大家自行参考官方文档即可。在了解了这些之后，你就可以放心大胆的在你的项目中慢慢渗透 Kotlin，让你的代码逐渐走向简洁与精致了。

Kotlin 从零基础到进阶

名称	日期	类型	大小	长度
01_kotlin课程简介1.mp4	2017/7/26 9:08	媒体文件(.mp4)	10,154 KB	00:06:32
02_kotlin学习方法.mp4	2017/7/24 18:23	媒体文件(.mp4)	11,559 KB	00:06:17
03_kotlin选好教练车.avi	2017/7/24 11:57	媒体文件(.avi)	54,123 KB	00:11:33
04_kotlin你好世界.avi	2017/7/24 11:57	媒体文件(.avi)	30,615 KB	00:06:47
05_kotlin变量与输出.avi	2017/7/24 11:57	媒体文件(.avi)	38,560 KB	00:10:53
06_kotlin二进制基础.avi	2017/7/24 11:57	媒体文件(.avi)	12,969 KB	00:04:36
07_kotlin变量和常量&类型推断.avi	2017/7/24 11:57	媒体文件(.avi)	37,182 KB	00:09:21
08_kotlin变量取值范围.avi	2017/7/24 11:57	媒体文件(.avi)	22,311 KB	00:05:10
09_kotlin函数入门.avi	2017/7/24 11:57	媒体文件(.avi)	28,935 KB	00:07:55
10_kotlin语言boolean.avi	2017/7/24 11:58	媒体文件(.avi)	30,865 KB	00:07:38
11_kotlin命令行交互式终端.avi	2017/7/24 11:58	媒体文件(.avi)	37,070 KB	00:05:42
12_kotlin函数加强.avi	2017/7/24 11:58	媒体文件(.avi)	45,668 KB	00:14:01
13_kotlin函数作业讲解.avi	2017/7/24 11:58	媒体文件(.avi)	24,653 KB	00:05:05
14_kotlin字符串模版.avi	2017/7/24 11:58	媒体文件(.avi)	30,606 KB	00:08:06
15_kotlin条件控制if和else.avi	2017/7/24 11:58	媒体文件(.avi)	22,430 KB	00:06:25
16_kotlin字符串比较.avi	2017/7/24 11:58	媒体文件(.avi)	17,113 KB	00:04:38
17_kotlin空值处理.avi	2017/7/24 11:58	媒体文件(.avi)	20,517 KB	00:06:06
18_kotlin的when表达式.avi	2017/7/24 11:58	媒体文件(.avi)	42,574 KB	00:11:19
19_kotlin的loop和Range.avi	2017/7/24 11:58	媒体文件(.avi)	32,488 KB	00:08:55
20_kotlin的list和map入门.avi	2017/7/24 11:58	媒体文件(.avi)	27,338 KB	00:06:32

名称	修改日期	类型	大小
01_kotlin课程简介.pdf	2017/8/1 11:01	PDF 文件	316 KB
02_kotlin学习方法.pdf	2017/8/1 11:01	PDF 文件	273 KB
03_kotlin选好教练车.pdf	2017/8/1 11:01	PDF 文件	268 KB
04_kotlin你好世界.pdf	2017/8/1 11:01	PDF 文件	292 KB
05_kotlin变量与输出1.pdf	2017/8/1 11:02	PDF 文件	239 KB
05_kotlin变量与输出2.pdf	2017/8/1 11:02	PDF 文件	228 KB
05_kotlin变量与输出3.pdf	2017/8/1 11:02	PDF 文件	104 KB
06_kotlin函数入门.pdf	2017/8/1 11:02	PDF 文件	278 KB
07_kotlin是真的么.pdf	2017/8/1 11:02	PDF 文件	237 KB
08_kotlin函数加强.pdf	2017/8/1 11:02	PDF 文件	347 KB
09_kotlin字符串模版.pdf	2017/8/1 11:02	PDF 文件	175 KB
10_kotlin条件控制.pdf	2017/8/1 11:02	PDF 文件	119 KB
11_kotlin字符串比较.pdf	2017/8/1 11:02	PDF 文件	106 KB
12_kotlin空值处理.pdf	2017/8/1 11:03	PDF 文件	174 KB
13_kotlinwhen表达式.pdf	2017/8/1 11:03	PDF 文件	129 KB
14_kotlin的loop和range.pdf	2017/8/1 11:03	PDF 文件	153 KB
15_kotlin的list和map入门.pdf	2017/8/1 11:03	PDF 文件	117 KB
16_kotlin函数和函数表达式.pdf	2017/8/1 11:03	PDF 文件	106 KB
17_kotlin默认参数和具名参数.pdf	2017/8/1 11:03	PDF 文件	226 KB
22_kotlin字符串转数字.pdf	2017/8/1 11:03	PDF 文件	106 KB
23_kotlin语言人机交互.pdf	2017/8/1 11:03	PDF 文件	288 KB
25_kotlin异常处理.pdf	2017/8/1 11:03	PDF 文件	111 KB

kotlin教程目录

01_kotlin课程简介

02_kotlin学习方法

03_kotlin选好教练车

04_kotlin你好世界

05_kotlin变量与输出

06_kotlin二进制基础

07_kotlin变量和常量&类型推断

08_kotlin变量取值范围

09_kotlin函数入门

10_kotlin语言boolean

11_kotlin命令行交互式终端

12_kotlin函数加强

13_kotlin函数作业讲解

14_kotlin字符串模版

15_kotlin条件控制if和else

16_kotlin字符串比较

17_kotlin空值处理

18_kotlin的when表达式

19_kotlin的loop和Range

20_kotlin的list和map入门

21_kotlin函数和函数式表达式

22_kotlin默认参数和具名参数

23_kotlin字符串和数字之间的转换

24_kotlin人机交互

25_kotlin异常处理

26_kotlin递归

27_kotlin尾递归优化

28_kotlin新的篇章idea使用入门

29_kotlin面向对象入门

30_kotlin静态属性和动态行为

31_kotlin面向对象

32_kotlin面向对象实战-洗衣机

33_kotlin面向对象实战-洗衣机升级

34_kotlin面向对象实战-封装

35_kotlin面向对象-继承(open和override)

36_kotlin抽象类和继承

37_kotlin面向对象-多态

38_kotlin面向对象-抽象类和接口

39_kotlin面向对象-代理和委托

40_kotlin面向对象-单例模式

41_kotlin面向对象-枚举

42_kotlin面向对象-印章类

43_kotlin课程计划

第1章 Kotlin简介

1.1 kotlin简史

1.1.1 Kotlin概述

> 科特林岛（Котлин）是一座俄罗斯的岛屿，位于圣彼得堡以西约30公里处，形状狭长，东西长度约14公里，南北宽度约2公里，面积有16平方公里，扼守俄国进入芬兰湾的水道。科特林岛上建有喀琅施塔得市，为圣彼得堡下辖的城市。

我们这里讲的Kotlin，就是一门以这个Котлин岛命名的现代程序设计语言。它是一门静态类型编程语言，支持JVM平台，Android平台，浏览器JS运行环境，本地机器码等。支持与Java，Android 100% 完全互操作。

其主要设计者是来自 Saint Petersburg, Russia JetBrains团队的布雷斯拉夫(Andrey Breslav , https://www.linkedin.com/in/abreslav/)等人，源码在github上，其实现主要是JetBrains团队成员以及开源贡献者。

> 认识一个事物的最好的方式，首先是取了解它的历史。

我们先来简单看一下来自wikipedia[0]的Kotlin简历：

标题	内容
设计者	JetBrains
实现者	JetBrains与开源贡献者
最新发行时间	Kotlin 1.1.2（2017年4月25日，34天前）
最新测试版发行日期	Kotlin 1.1.3 EAP（2017年5月27日，2天前 [1]）
类型系统	静态类型
系统平台	输出Java虚拟机比特码以及JavaScript源代码

操作系统	任何支持JVM或是JavaScript的解释器
许可证	Apache 2
文件扩展名	.kt
网站	kotlinlang.org
启发语言	Java、Scala、Groovy、C#、Gosu

（注：这里的日期时间，取的是本书当时写作时间）

Kotlin的亲爹是大名鼎鼎的Jetbrains公司。它有一系列耳熟能详的产品，诸如Android程序员们天天用的Android Studio, Java程序员们天天用的IntelliJ IDEA, 还有前端的WebStorm, PhpStorm等等。所以说，使用IntelliJ IDEA了开发Kotlin程序将会非常便捷。

Kotlin这个语言从一开始推出到如今，已经有六年了。官方正式发布首个稳定版本的时间相对比较晚(2016.2)，这是一门比较新的语言。其大致发展简史如下：

2011年7月，JetBrains推出Kotlin项目。

2012年2月，JetBrains以Apache 2许可证开源此项目。

2016年2月15日，Kotlin v1.0（第一个官方稳定版本）发布。

2017 Google I/O 大会，Kotlin “转正”。

Kotlin 具有很多下一代编程语言[1][2]静态语言特性：如类型推断、多范式支持、可空性表达、扩展函数、模式匹配等。

Kotlin的编译器kompiler可以被独立出来并嵌入到 Maven、Ant 或 Gradle 工具链中。这使得在 IDE 中开发的代码能够利用已有的机制来构建，可以在新环境中自由使用。

Kotlin以K字打头的用语，甚至连 contributors 这类词也改成了kontributors。

1.1.2 Kotlin 元年:2016

2016 年是 Kotlin “元年（First year of Kotlin）”，官网给出了这样一幅图来展示它一年来的成绩：

Github 上面的代码量破千万，8000多基于kotlin项目。使用 Kotlin 的人逐渐增多。

Kotlin 是由工程师设计，各种细节设计非常切合工程师的需要。语法近似 Java 和 Scala，且已活跃在 Android 开发领域，被誉为 Android 平台的 Swift。

其主要设计目标：

- 创建一种兼容 Java 的语言
- 让它比 Java 更安全，能够静态检测常见的陷阱。如：引用空指针
- 让它比 Java 更简洁，通过支持 variable type inference，higher-order functions (closures)，extension functions，mixins and first-class delegation 等实现。
- 让它比最成熟的竞争对手 Scala语言更加简单。

Kotlin 的学习曲线极其平缓，学习量相当于一个框架。有经验的程序员阅读了文档即刻上手。

1.2 快速学习工具

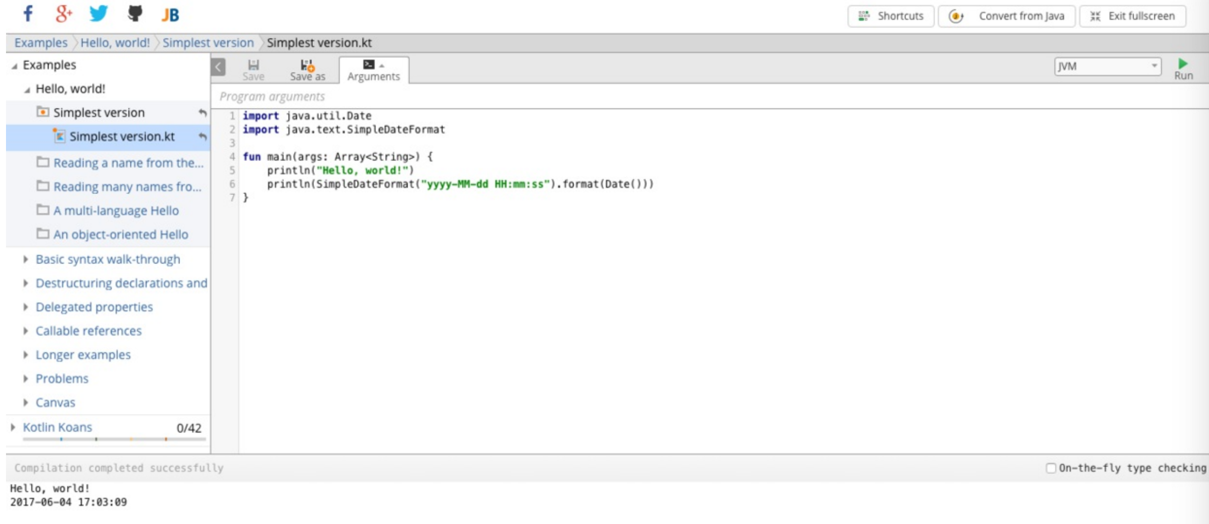
> 工欲善其事必先利其器

1.2.1 云端IDE

未来的是云的世界。不需要搭建本地开发运行环境，直接用浏览器打开

> https://try.kotlinlang.org/

你就可以直接使用云端IDE来即时编写Kotlin代码，并运行之。一个运行示例如下图：



1.2.2 本地命令行环境搭建

Kotlin是运行在JVM环境下的语言。首先我们要有JDK环境。

有时候我们并不需要打开IDE来做一些事情。打开 IDE 是件很麻烦的事情，在某些场景下，我们比较喜欢命令行。

使用命令行环境，我们可以方便地使用Kotlin REPL（Read-Eval-Print-Loop，交互式编程环境）。REPL可以实时编写Kotlin代码，并查看运行结果。通常REPL交互方式可以用于调试、测试以及试验某种想法。

下面我们讲下怎么搭建 Kotlin 命令行环境。

Kotlin 命令行环境主要依赖就是Kotlin Compiler，目前最新版本是 1.1.2-2。其下载链接是：

https://github.com/JetBrains/kotlin/releases/tag/v1.1.2-2

这个zip包里面就是Kotlin Compiler的核心依赖jar包。解压后，目录结构如下：

```
.
├── bin
│   ├── kotlin
│   ├── kotlin.bat
│   ├── kotlinc
│   ├── kotlinc-js
│   ├── kotlinc-js.bat
│   ├── kotlinc-jvm
│   ├── kotlinc-jvm.bat
│   └── kotlinc.bat
├── build.txt
├── lib
│   ├── allopen-compiler-plugin.jar
│   ├── android-extensions-compiler.jar
```

其中，kotlinc，kotlin两个命令就是Kotlin语言的编译.kt文件和运行Kt.class文件命令。

```
│   ├── kotlin-annotation-processing.jar
│   ├── kotlin-ant.jar
│   ├── kotlin-build-common-test.jar
│   ├── kotlin-compiler-client-embeddable.jar
│   ├── kotlin-compiler.jar
│   ├── kotlin-daemon-client.jar
│   ├── kotlin-jslib-sources.jar
│   ├── kotlin-jslib.jar
│   ├── kotlin-preloader.jar
│   ├── kotlin-reflect.jar
│   ├── kotlin-runner.jar
│   ├── kotlin-runtime-sources.jar
│   ├── kotlin-runtime.jar
│   ├── kotlin-script-runtime-sources.jar
│   ├── kotlin-script-runtime.jar
│   ├── kotlin-stdlib-js-sources.jar
│   ├── kotlin-stdlib-js.jar
│   ├── kotlin-stdlib-sources.jar
│   ├── kotlin-stdlib.jar
│   ├── kotlin-test-js.jar
│   ├── kotlin-test.jar
│   ├── noarg-compiler-plugin.jar
│   ├── sam-with-receiver-compiler-plugin.jar
│   └── source-sections-compiler-plugin.jar
└── license
    ├── LICENSE.txt
    ├── NOTICE.txt
    └── third_party
        ├── args4j_LICENSE.txt
        ├── asm_license.txt
        ├── closure-compiler_LICENSE.txt
        ├── dart_LICENSE.txt
        ├── jshashtable_license.txt
        ├── json_LICENSE.txt
        ├── maven_LICENSE.txt
        ├── pcollections_LICENSE.txt
        ├── prototype_license.txt
        ├── rhino_LICENSE.txt
        ├── scala_license.txt
        ├── trove_license.txt
        └── trove_readme_license.txt

4 directories, 50 files
```

其中，kotlinc，kotlin两个命令就是Kotlin语言的编译.kt文件和运行Kt.class文件命令。

我们来看一下kotlinc的命令：

```
#!/usr/bin/env bash
#
##############################################################################
# Copyright 2002-2011, LAMP/EPFL
# Copyright 2011-2015, JetBrains
#
# This is free software; see the distribution for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
##############################################################################

cygwin=false;
case "`uname`" in
    CYGWIN*) cygwin=true ;;
esac

# Based on findScalaHome() from scalac script
findKotlinHome() {
    local source="${BASH_SOURCE[0]}"
    while [ -h "$source" ] ; do
        local linked="$(readlink "$source")"
        local dir="$(cd -P $(dirname "$source") && cd -P $(dirname "$linked") && pwd)"
        source="$dir/$(basename "$linked")"
    done
    (cd -P "$(dirname "$source")/.." && pwd)
}

KOTLIN_HOME="$(findKotlinHome)"

if $cygwin; then
    # Remove spaces from KOTLIN_HOME on windows
    KOTLIN_HOME=`cygpath --windows --short-name "$KOTLIN_HOME"`
fi

[ -n "$JAVA_OPTS" ] || JAVA_OPTS="-Xmx256M -Xms32M"

declare -a java_args
declare -a kotlin_args

while [ $# -gt 0 ]; do
  case "$1" in
    -D*)
      java_args=("${java_args[@]}" "$1")
      shift
      ;;
    -J*)
      java_args=("${java_args[@]}" "${1:2}")
      shift
```

```
        ;;
      *)
        kotlin_args=("${kotlin_args[@]}" "$1")
        shift
        ;;
    esac
done

if [ -z "$JAVACMD" -a -n "$JAVA_HOME" -a -x "$JAVA_HOME/bin/java" ]; then
    JAVACMD="$JAVA_HOME/bin/java"
fi

declare -a kotlin_app

if [ -n "$KOTLIN_RUNNER" ];
then
    java_args=("${java_args[@]}" "-Dkotlin.home=${KOTLIN_HOME}")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-runner.jar" "org.jetbrains.kotlin.runner.Mai
n")
else
    [ -n "$KOTLIN_COMPILER" ] || KOTLIN_COMPILER=org.jetbrains.kotlin.cli.jvm.K2JVMComp
iler
    java_args=("${java_args[@]}" "-noverify")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-preloader.jar" "org.jetbrains.kotlin.preload
ing.Preloader" "-cp" "${KOTLIN_HOME}/lib/kotlin-compiler.jar" $KOTLIN_COMPILER)
fi

"${JAVACMD:=java}" $JAVA_OPTS "${java_args[@]}" -cp "${kotlin_app[@]}" "${kotlin_args[@
]}"
```

我们可以看出，kotlinc是直接依赖java命令的，所以，使用Kotlin Compiler，首先要有JDK环境。

其中kotlin-preloader.jar、kotlin-compiler.jar是其入口依赖jar，入口类是org.jetbrains.kotlin.cli.jvm.K2JVMCompiler。

kotlin命令脚本如下

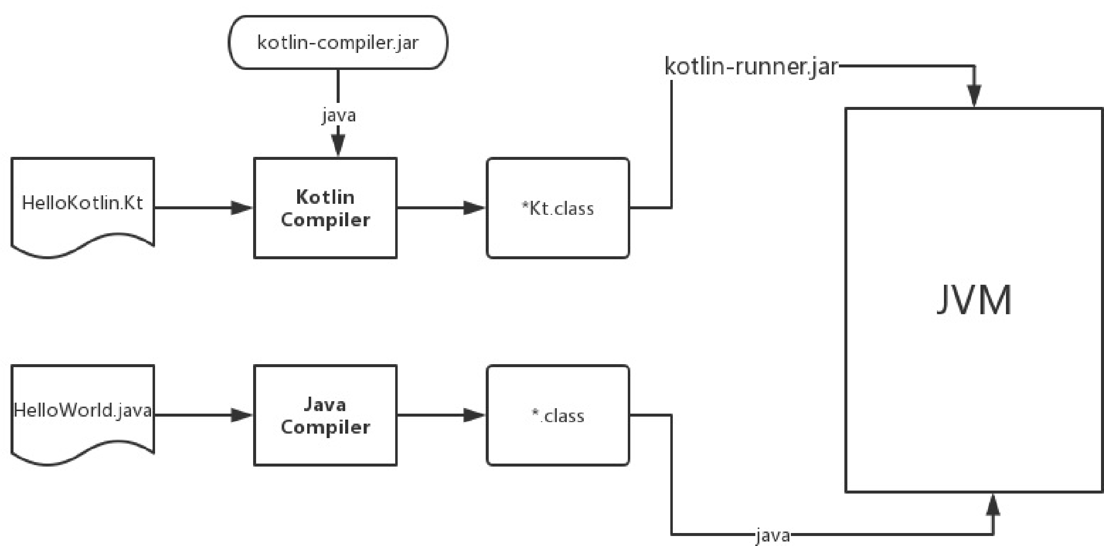
```
export KOTLIN_RUNNER=1

DIR="${BASH_SOURCE[0]%/*}"
: ${DIR:="."}

"${DIR}"/kotlinc "$@"
```

我们可以看出，直接是依赖kotlinc。在if逻辑代码中：

```
if [ -n "$KOTLIN_RUNNER" ];
```

```
then
    java_args=("${java_args[@]}" "-Dkotlin.home=${KOTLIN_HOME}")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-runner.jar" "org.jetbrains.kotlin.runner.Mai
n")
```

从这个逻辑，我们可以看出，Kt.class在java命令执行前，需要从kotlin-runner.jar这个逻辑里走一遍。同时，我们也能知道Kt.class跟Java.class文件有着这个kotlin-runner.jar的逻辑映射上的区别。也就是说，Kotlin的Bytecode跟纯的JVM bytecode存在一个kotlin-runner.jar的映射关系。其大致执行过程如下图所示：



像scala，groovy等基于JVM的语言的compiler,runner，基本都采用这种运行方式。在实现细节上也许会有不同，总的思路是一致的。比如说，scalac的入口类

```
https://github.com/EasyKotlin/scala/blob/2.12.x/src/compiler/scala/tools/nsc/Main.scala
```

对应scalac中的命令行脚本是：

```
...

execCommand \
  "${JAVACMD:=java}" \
  $JAVA_OPTS \
  "${java_args[@]}" \
  "${classpath_args[@]}" \
  -Dscala.home="$SCALA_HOME" \
  $OVERRIDE_USEJAVACP \
  "$EMACS_OPT" \
  $WINDOWS_OPT \
   scala.tools.nsc.Main  "$@"
```

```
...
```

我们解压完kotlin-compiler-1.1.2-2.zip，放到相应的目录下。然后配置系统环境变量：

```
export KOTLIN_HOME=/Users/jack/soft/kotlinc
export PATH=$PATH:$KOTLIN_HOME/bin
```

执行 `source ~/.bashrc` ，命令行输入 `kotlinc` ， 即可REPL环境，我们可以看到如下输出：

```
$ kotlinc
Welcome to Kotlin version 1.1.2-2 (JRE 1.8.0_40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World")
Hello,World
>>>
```

然后，我们就可以像使用python,ruby,scala,groovy的REPL一样去尽情享受Kotlin的编程乐趣了。

1.2.3 使用IntelliJ IDEA

最新版本的IDEA已经默认集成了Kotlin环境。

我们首先去下载安装IntelliJ IDEA。下载页面是：

https://www.jetbrains.com/idea/download/index.html

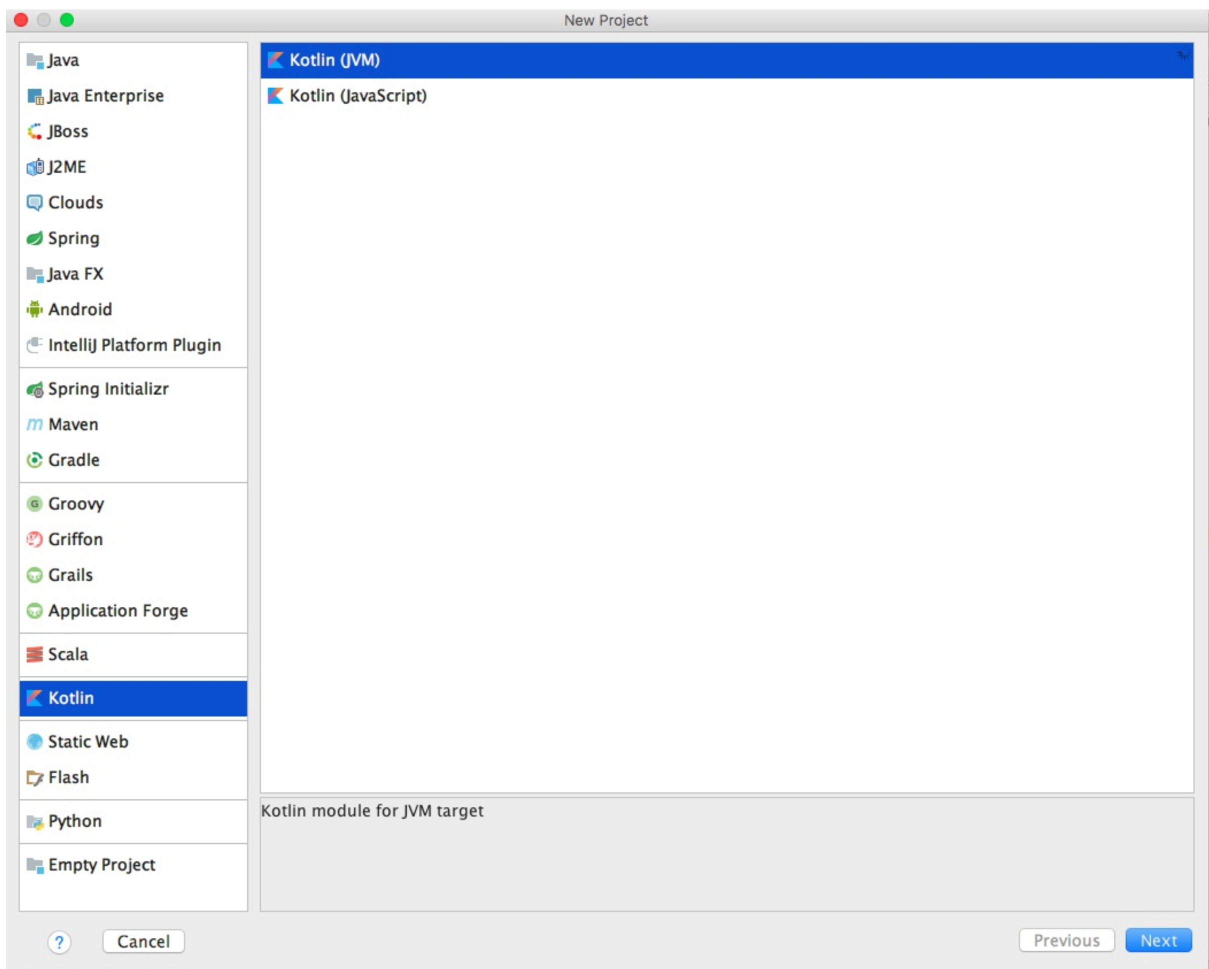
如果您之前没用过IDEA，现在想尝试一下，可以去下面这个页面了解一下：

https://www.jetbrains.com/idea/documentation/

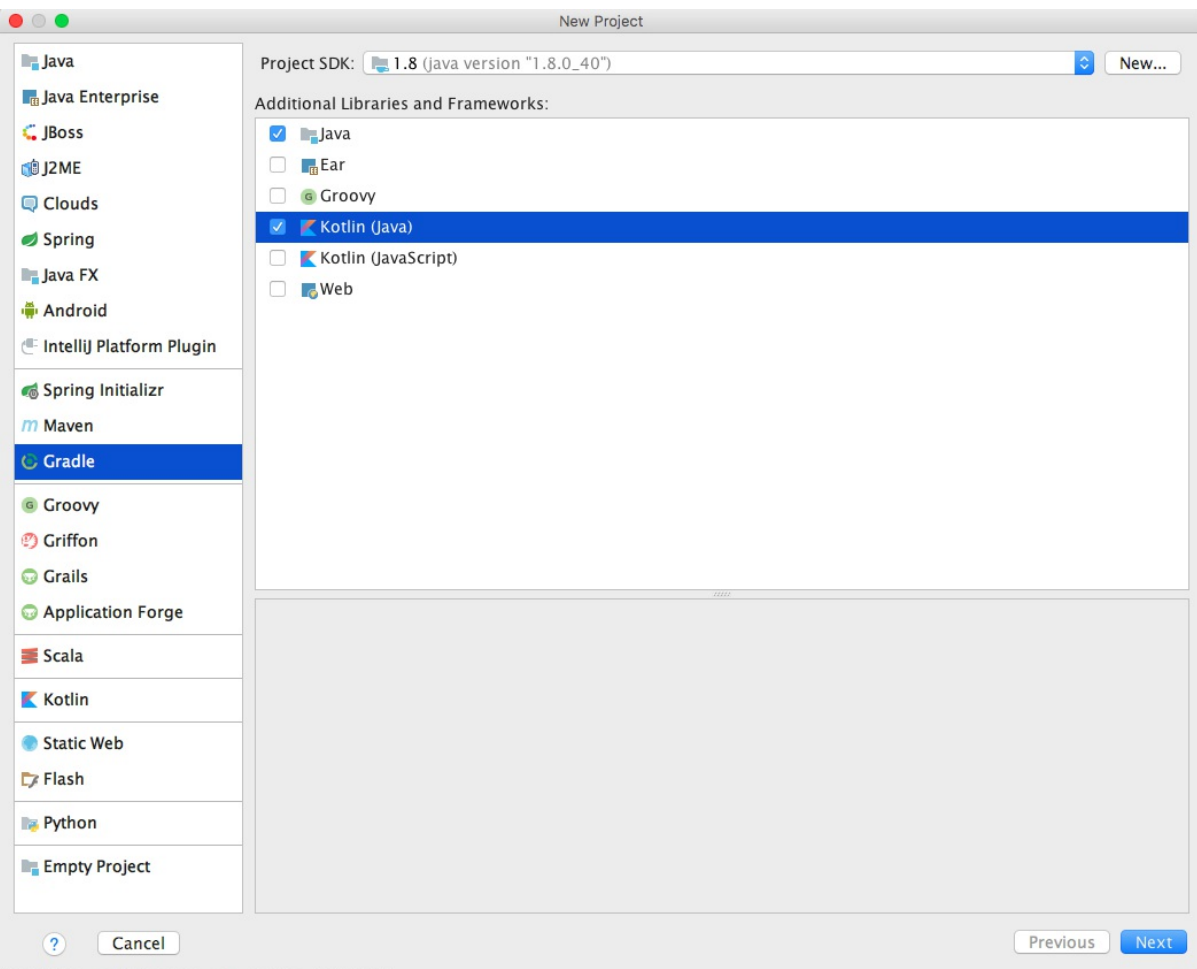
安装完毕，然后点击 `File > New > Project` ，我们可以选择

> Koltin: Kotlin(JVM), Kotlin(JavaScript)

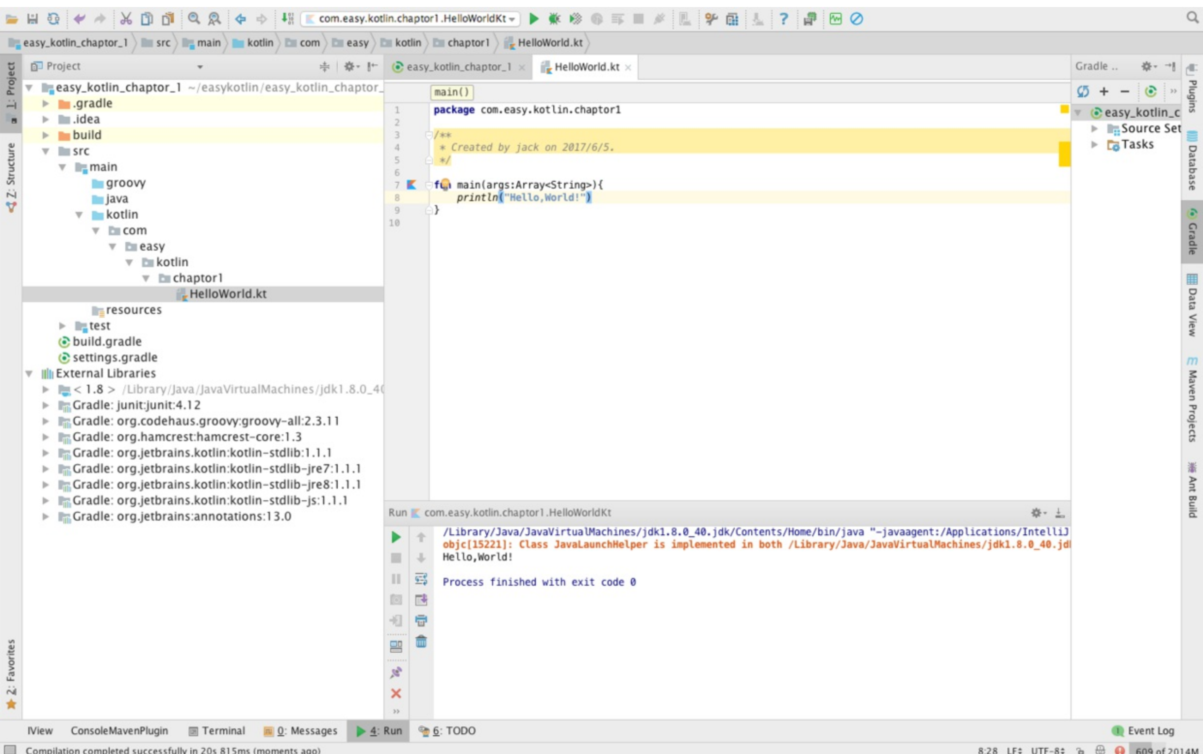
如下图所示



也可以选择Maven，Gradle构建工程。本书采用Gradle来构建工程。如下图所示：



然后按照后续步骤操作，最后等待Gradle下载依赖，完成工程构建。我们将得到一个标准的Gradle工程。

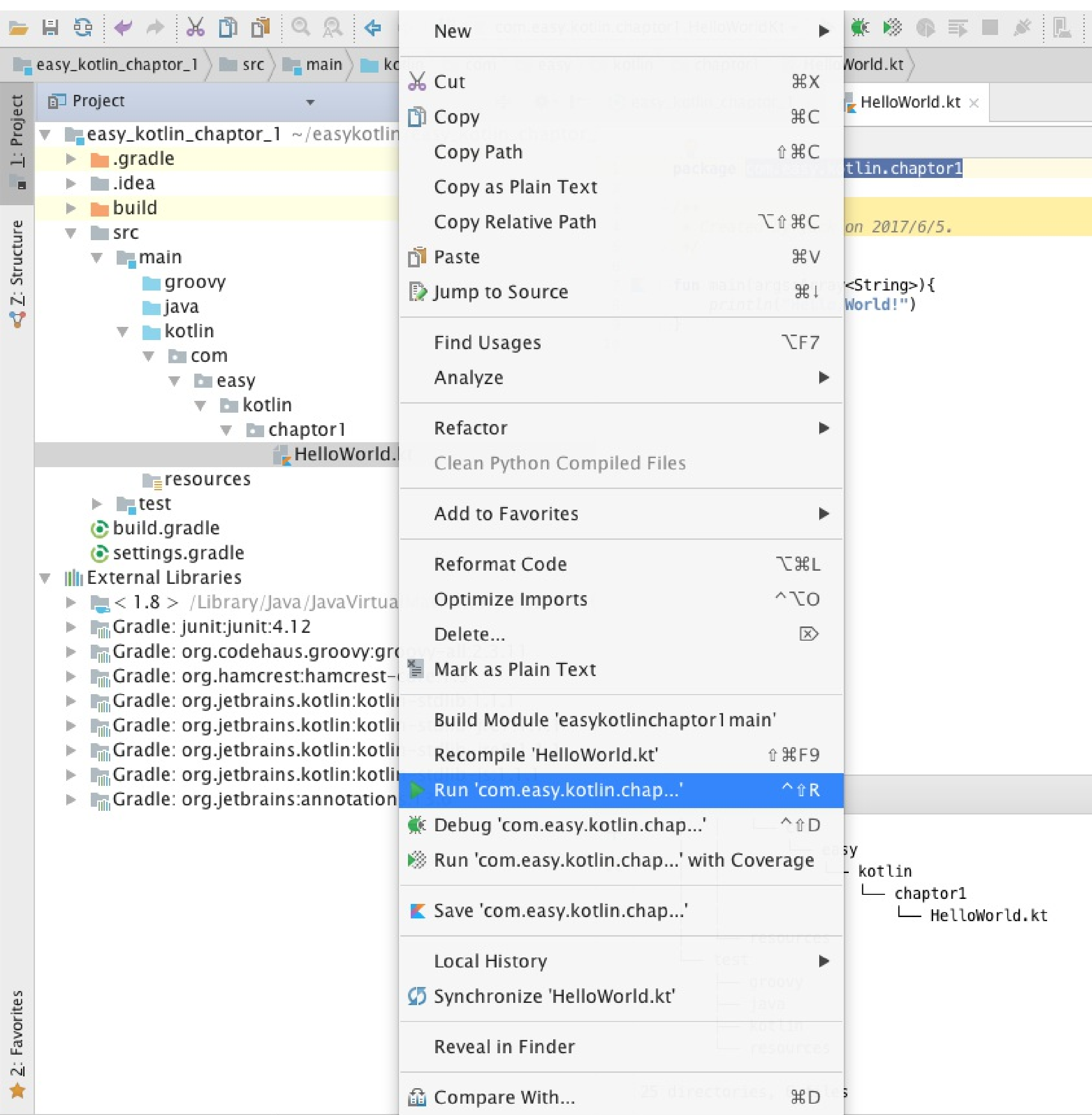


我们在 `src/main/kotlin` 下面新建 `package` :com.easy.kotlin.chapter1

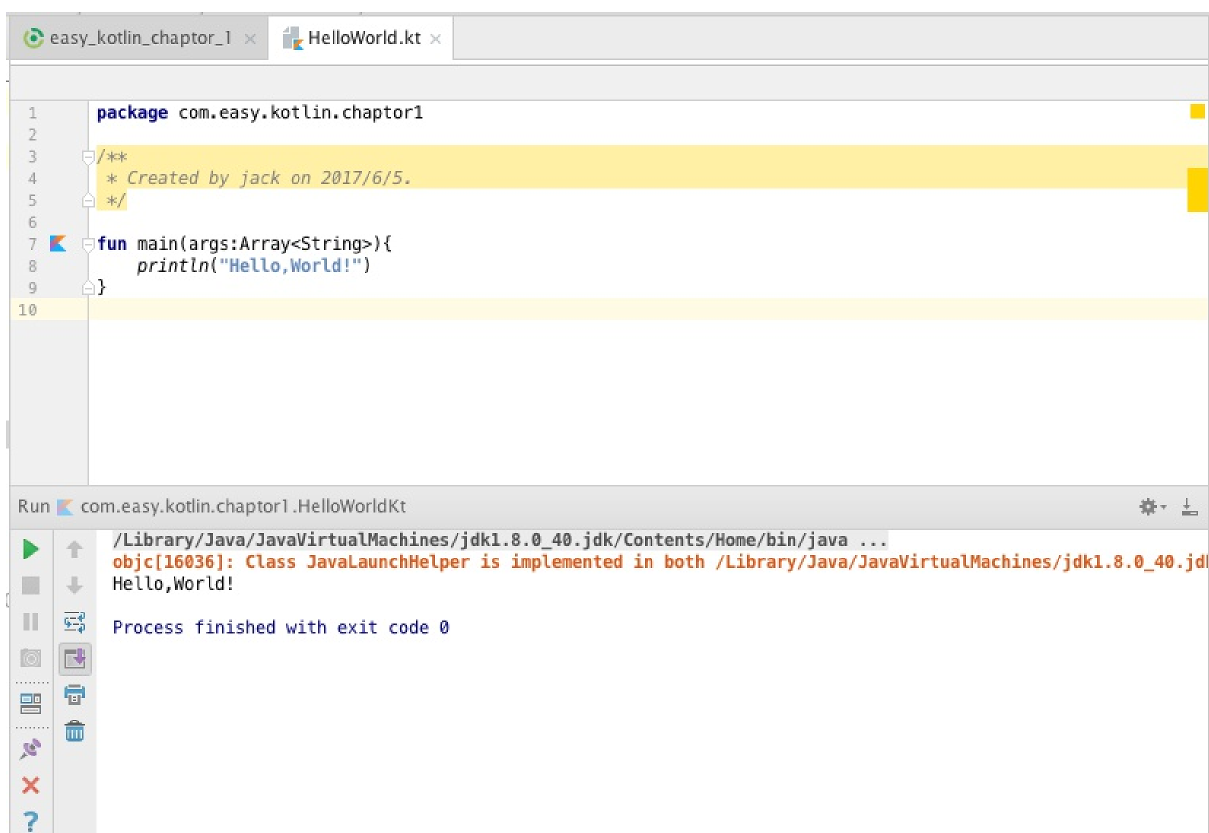
然后新建HelloWorld.kt，编写以下代码

```
package com.easy.kotlin.chapter1

/**
 * Created by jack on 2017/6/5.
 */

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

右击运行该类，如下图



我们将会得到输出



我们观察IDEA控制台输出的执行日志，可以看出IDEA集成Kotlin环境使用的核心依赖jar包：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java "-javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=65404:/Applications/IntelliJ IDEA.app/Contents/bin" -Dfile.encoding=UTF-8 -classpath /Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre/lib/charsets.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre/lib/deploy.jar:...
...

.../kotlin-stdlib-jre8-1.1.1.jar:
.../kotlin-stdlib-jre7-1.1.1.jar:
.../kotlin-stdlib-1.1.1.jar:... com.easy.kotlin.chapter1.HelloWorldKt
...


Hello,World!

Process finished with exit code 0
```

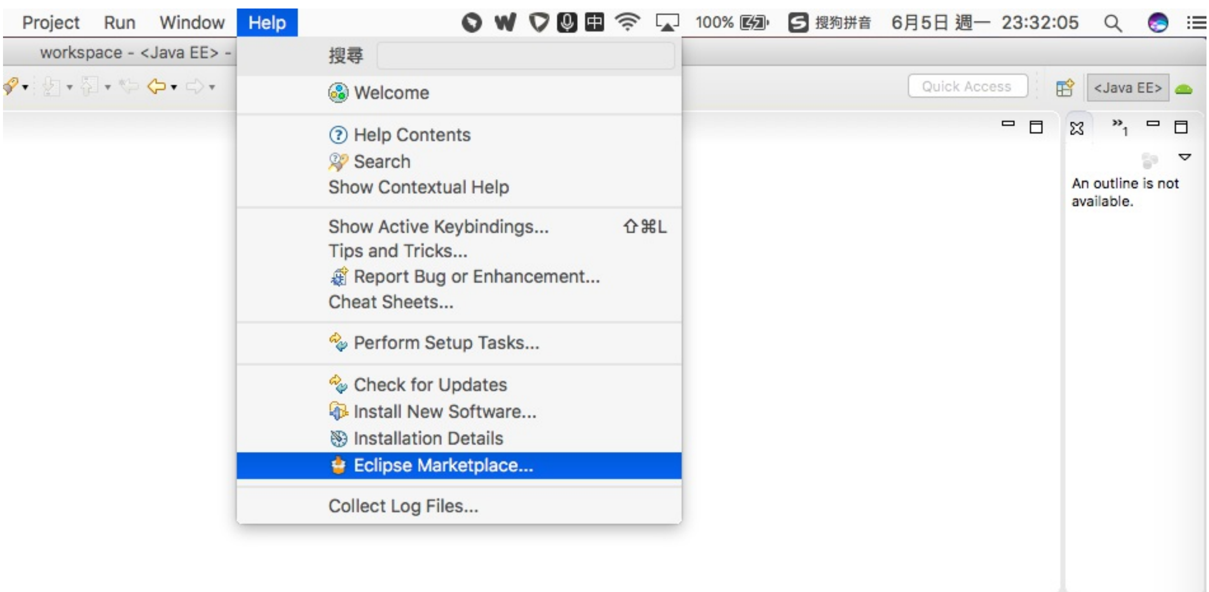
本小节的示例工程代码：https://github.com/EasyKotlin/easy_kotlin_chaptor_1

1.2.4 使用Eclipse

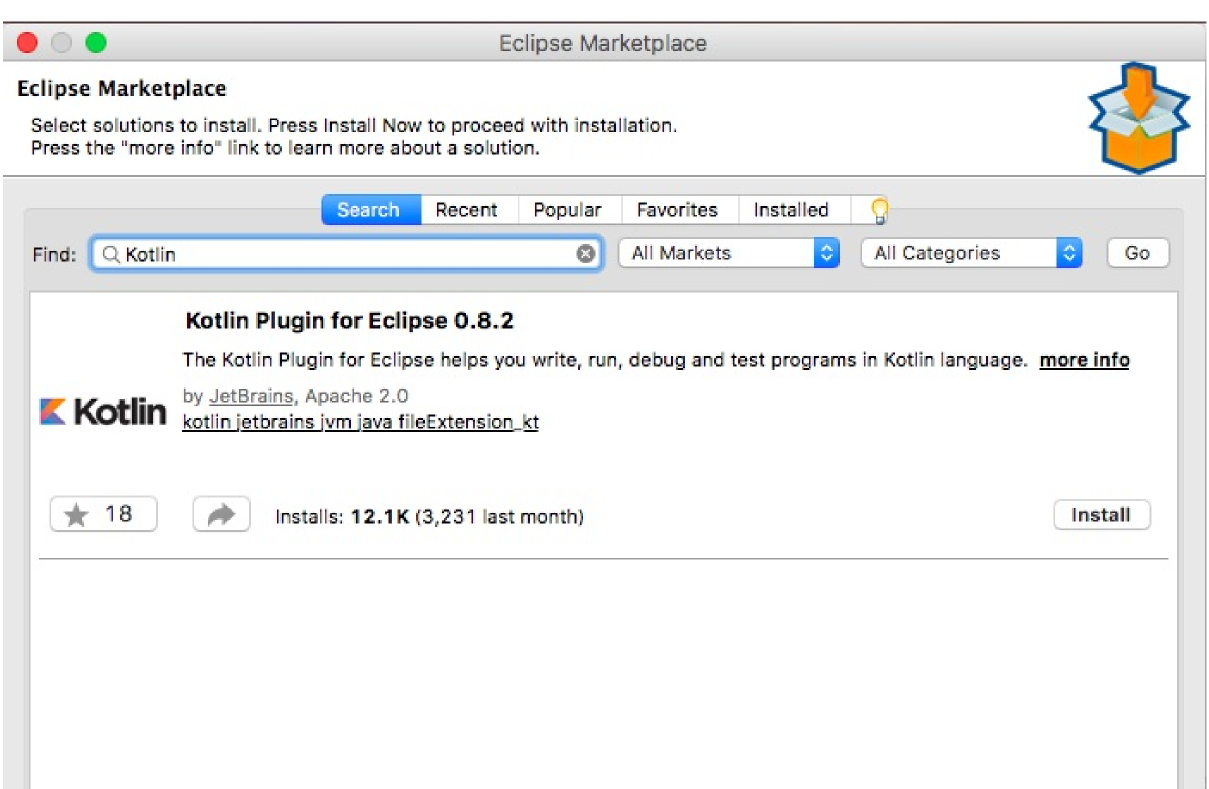
使用Eclipse的开发者们，可以通过安装Kotlin插件来进行Kotlin程序的开发。但是，体验上要比使用IDEA逊色很多。如果您想完美体验Kotlin在IDE中的开发，强烈建议使用IDEA。JetBrains自家的东西，自然是比Eclipse支持的要好很多。

我们下面简单介绍一下在Eclipse中开发Kotlin程序的方法。

首先，打开 `Help > Eclipse Marketplace` ，如下图

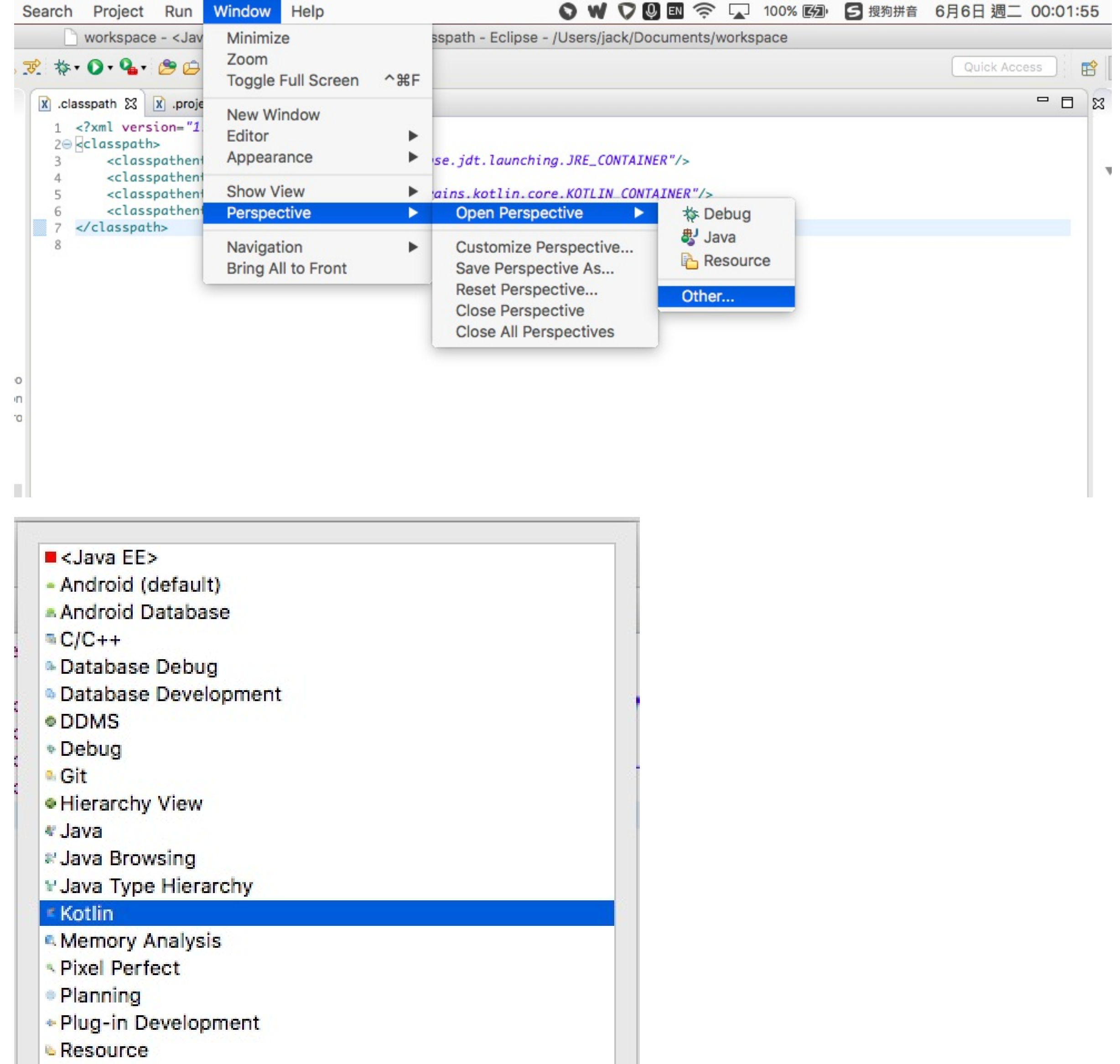


在搜索框里输入 `Kotlin` ，将得到如下结果

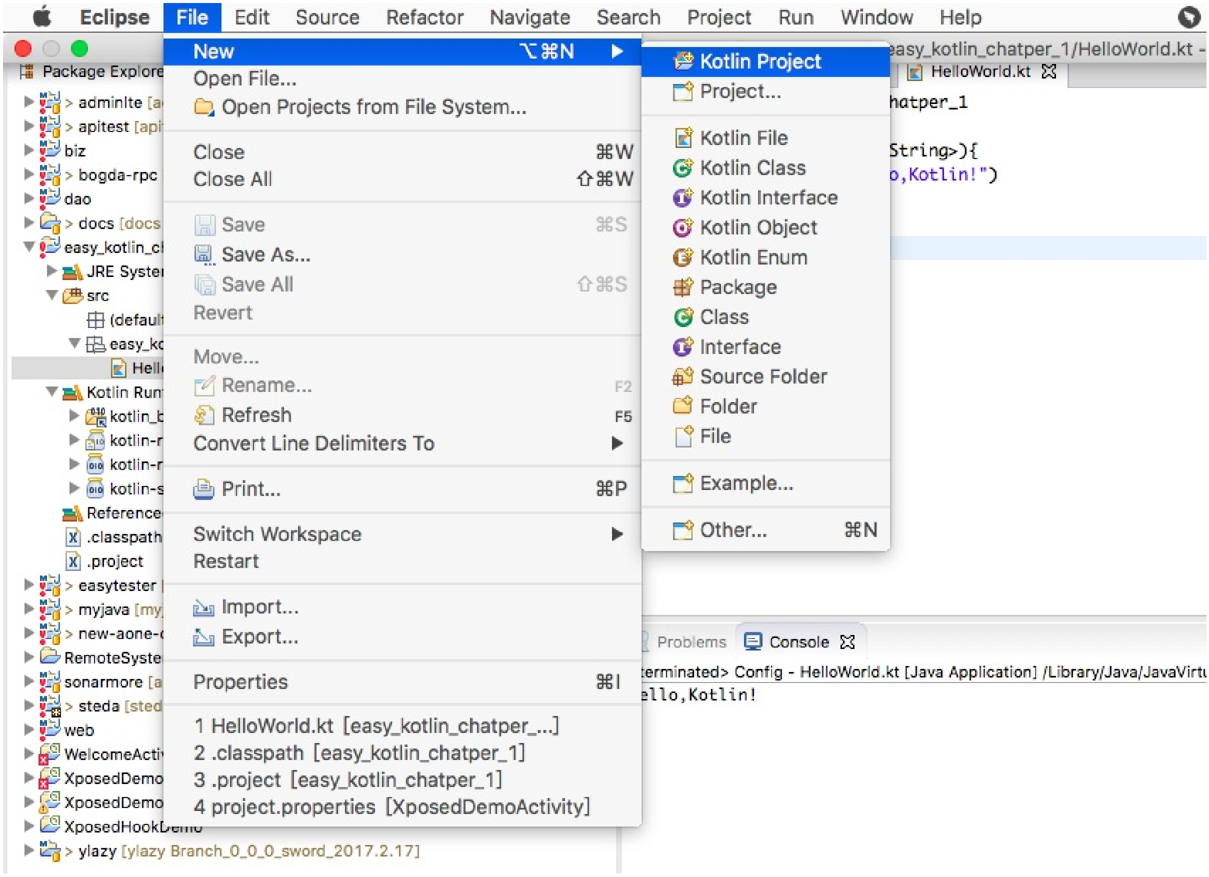


点击 `Install` ，等待完成安装，重启Eclipse。

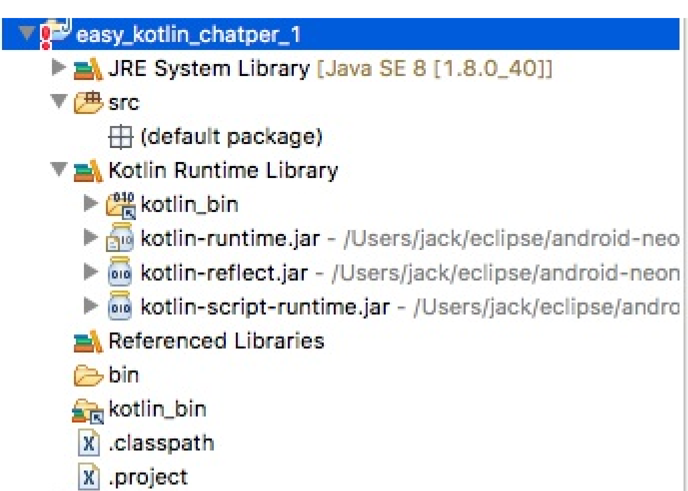
然后，选择 `Kotlin Perspective` ，如下图



点击OK。下面我们就可以新建 `Kotlin` 工程了。如下图



新建完工程，我们将得到如下结构的工程



我们可以看出， `kotlin-runtime.jar` ，`kotlin-reflect.jar` ，`kotlin-script-runtime.jar` 被加到了工程依赖库里。

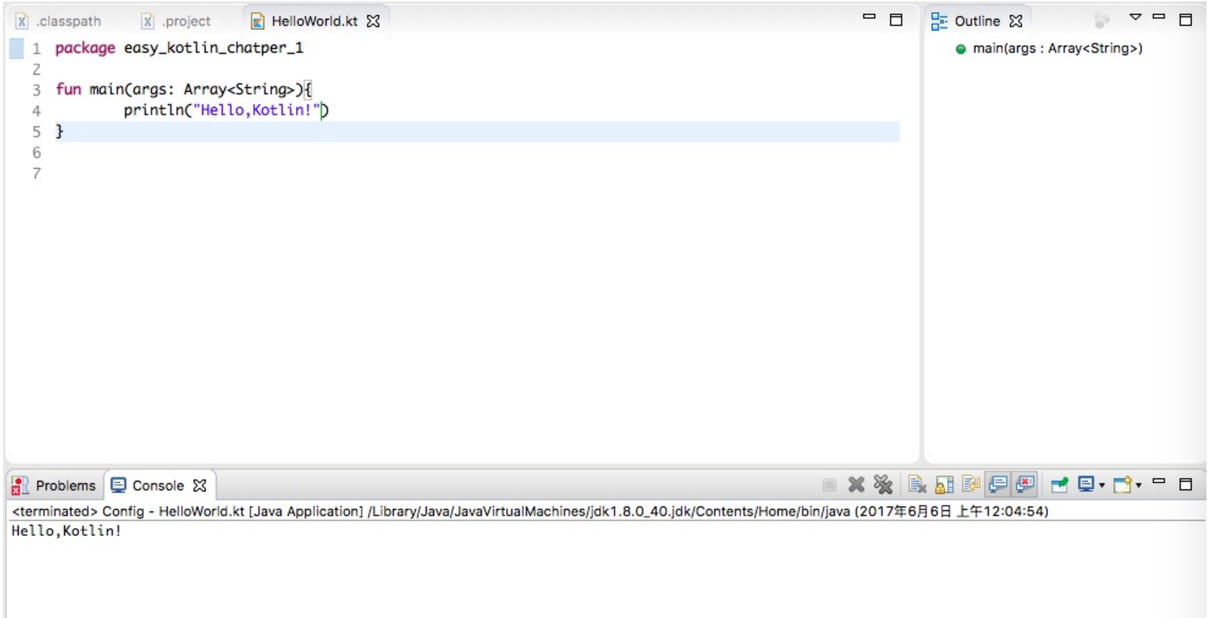
这个配置是在 `.classpath` ， `.project` 配置的。当然这些配置依赖库，执行程序等等的工作是由Eclipse Kotlin插件完成的。

我们在 `src` 目录新建一个 `package` ：easy_kotlin_chatper_1

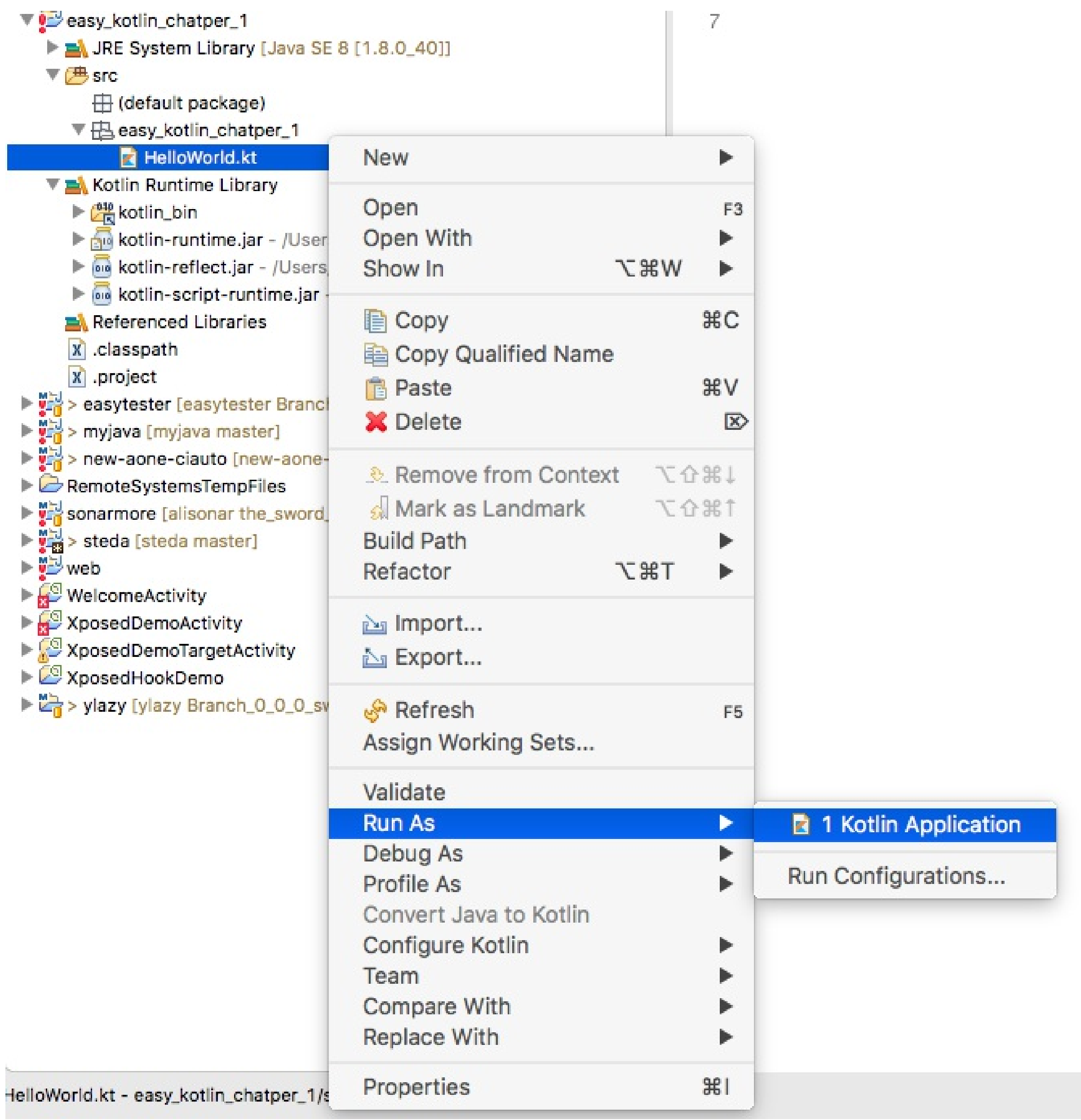
然后在此 `package` 下面新建一个 `HelloWorld.kt` 源码文件，内容如下

```
package easy_kotlin_chatper_1

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

如下图



右击 `HelloWorld.kt` 源码文件，如下图运行



如果一切正常，我们将得到如下输出

```
Hello,Kotlin!
```

本节示例工程源码：https://github.com/EasyKotlin/easy_kotlin_chatper_1

1.2.5 使用Gradle构建Kotlin工程

我们在上面小节中展示了使用IntelliJ IDEA建立一个Kotlin Gradle工程的步骤。我们在本节简单介绍一下使用Gradle构建Kotlin工程的配置。这个配置主要在build.gradle文件中。

其中，构建过程的核心依赖配置如下：

```
buildscript {
```

```
    ext.kotlin_version = '1.1.1'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

kotlin-gradle-plugin完成了Gradle构建Kotlin工程的所有依赖构建执行的相关工作。

然后，使用Gradle java、kotlin插件：

```
apply plugin: 'java'
apply plugin: 'kotlin'
```

当然，如果我们同时想使用Groovy语言，加上

```
apply plugin: 'groovy'
```

源代码JDK兼容性配置兼容1.8往后的版本：

```
sourceCompatibility = 1.8
```

配置Maven仓库：

```
repositories {
    mavenCentral()
}
```

工程依赖：

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
    compile 'org.codehaus.groovy:groovy-all:2.3.11'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

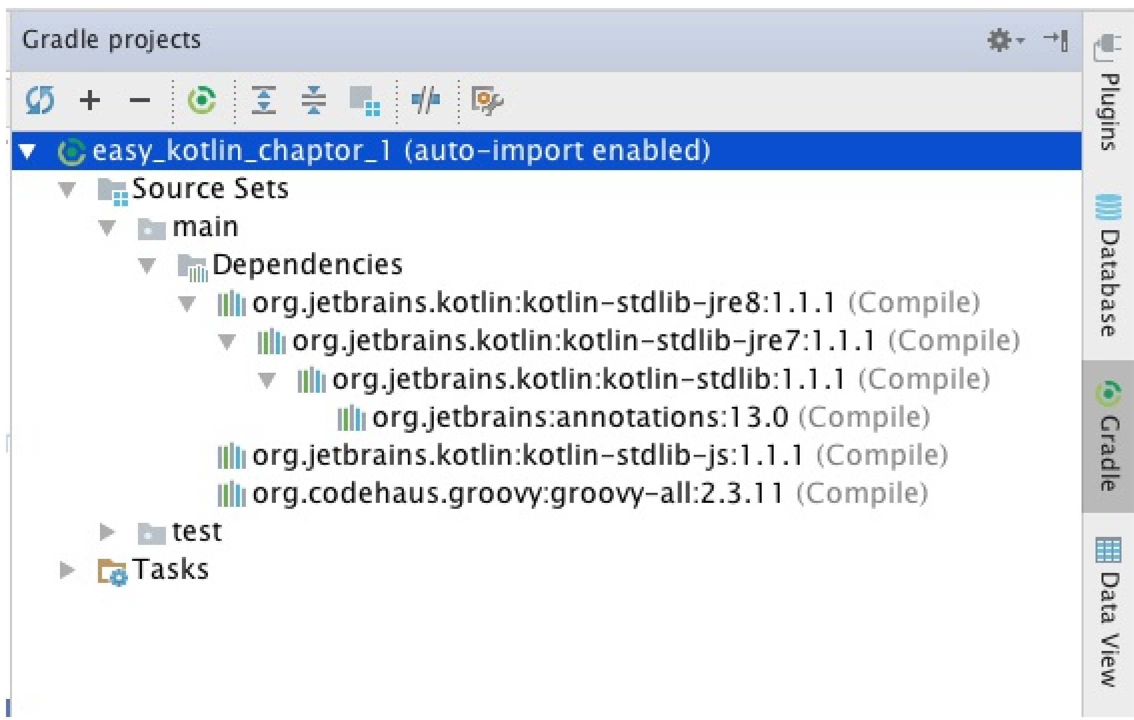
其中，kotlin-stdlib-jre8是Kotlin JVM执行环境依赖。

org.jetbrains.kotlin:kotlin-stdlib-js是Kotlin JS执行环境依赖。

我们可以通过Gradle项目的依赖树看出kotlin-stdlib-jre8依赖

> org.jetbrains.kotlin:kotlin-stdlib:1.1.1

如下图



kotlin-stdlib是Kotlin运行环境的标准库。

1.2.6 编程语言学习小结

学习一门语言大概会经历如下几步。

1.基本语法

学习任何东西，都是一个由表及里的过程。学习一门编程语言也一样。对于一门编程语言来说，“表”就是基本词汇和语法。

对于基础语法的学习，我们可以看一些简短而又系统的教程。

2.编码实践

所谓“纸上得来终觉浅，绝知此事要躬行”是也。此处就不多说。

> 掌握基础，持续练习

每一门编程语言的学习内容都会涉及：

> 运行环境 数据类型（数字、字符串、数组、集合、映射字典等） 表达式 函数 流程控制 类、方法

等等，不同的语言还有一些不同的特性，可以通过对比学习来加深理解。并通过大量实践深入理解，达到熟练使用。后面还要再去深入了解面向对象编程OOP、函数式编程FP、并发、异常、文件IO、网络、标准库等内容，并辅以持续的练习，这些内容才能够让你真正进入编程领域并做出

实际的软件。

> 相信10000小时定律，No Pain, No Gain。

学习一门新的语言的时候，要利用以前所学的语言的功底，但是也要保持开放的心态。这些在认知心理学中有相应的讲述。感兴趣的读者可以去了解一下。

3.技近乎道

基础语法学习，能让你快速上手，应用实践。对技巧和坑的关注，一定程度上拓展了你的知识面。系统学习，一方面会进一步拓展你的知识面。另一方面，也有利于你语言知识结构的形成。

任何一门成熟语言，都有其特有的生态。这个生态包括: 框架，扩展包，解决方案，模式, 规范等。

在不断编码实践过程中，我们逐步熟练使用很多API库、框架，也不断踩坑填坑、看源代码、不断解决问题，不断加深对语言的理解，同时会看一些优秀的框架源代码。

如果还有精力，我们再去学习语言更底层的东西，而不仅仅停留在应用层面。如Java中的集合类实现的算法与数据结构，如JVM是如何执行Java代码的。如Java的线程和操作系统线程的关系。以及一些操作系统方面的知识。

最后，达到游刃有余的境界。这一层级，基本可入武林高手之列了。

> 庖丁释刀对曰：“臣之所好者，道也，进乎技矣。始臣之解牛之时，所见无非牛者。三年之后，未尝见全牛也。方今之时，臣以神遇而不以目视，官知止而神欲行。依乎天理，批大郤，导大窾，因其固然，技经肯綮之未尝，而况大軱乎！

这里的“牛”，可以理解为我们所说的各种编程思想，编程范式，编程方法，编程技巧等等。最后，达到“运用之妙，存乎一心”之境也。

4.创造新世界

> 编程的本质就是创造世界。

达到这个境界的，基本都是世界顶尖大牛了。

例如，编程语言发展史上的杰出人物（下面只是一份不完全名单）：

```
约翰·冯·诺伊曼：操作系统概念的发起者
肯·汤普逊&丹尼斯·里奇：发明了C和Unix
约翰·巴科斯：发明了Fortran
阿兰·库珀：开发了Visual Basic
詹姆斯·高斯林：开发了Oak，即后来的Java
安德斯·海尔斯伯格：开发了Turbo Pascal、Delphi，以及C#
葛丽丝·霍普，开发了Flow-Matic，该语言对COBOL造成了影响
肯尼斯·艾佛森：，开发了APL，并与Roger Hui合作开发了J
比尔·乔伊：发明了vi，BSD，Unix的前期作者，以及SunOS的发起人，该操作系统后来改名为Solaris
艾伦·凯：开创了面向对象编程语言，以及Smalltalk的发起人
Brian Kernighan：与丹尼斯·里奇合著第一本C程序设计语言的书籍，同时也是AWK与AMPL程序设计语言的共同作者
约翰·麦卡锡：发明了LISP
```

```
比雅尼·斯特劳斯特鲁普：开发了C++
尼克劳斯·维尔特：发明了Pascal与Modula
拉里·沃尔：创造了Perl与Perl 6
吉多·范罗苏姆：创造了Python
......
```

这些人，都在创造一个美妙的思维逻辑之塔，创造一个新世界。正是这些各个编程领域的引领者们，才使得我们这个世界更加美好。

小结

本章我们简单介绍了Kotlin语言的发展过程，以及Kotlin开发环境的搭建方法。简单总结了学习一门编程语言的基本过程。我们的这本书基本是按照这个思路组织架构的。

我们将在下一章进入快速开始：Hello,World!

感谢您的阅读！恭喜您已经正式开启Kotlin世界之旅，希望本书能够帮到您的学习，哪怕是一点点启发也倍感欣慰。

本书所涉及到的示例工程代码统一放在这里：https://github.com/EasyKotlin

参考资料

1.http://www.onboard.jetbrains.com/articles/04/10/lop/index.html
2.https://medium.com/@octskyward/why-kotlin-is-my-next-programming-language-c25c001e26e3 3.http://kotlinlang.org/docs/tutorials/command-line.html
4.http://hadihariri.com/2013/12/29/jvm-minimal-survival-guide-for-the-dotnet-developer/

第2章 快速开始：HelloWorld

2.1 命令行的HelloWorld

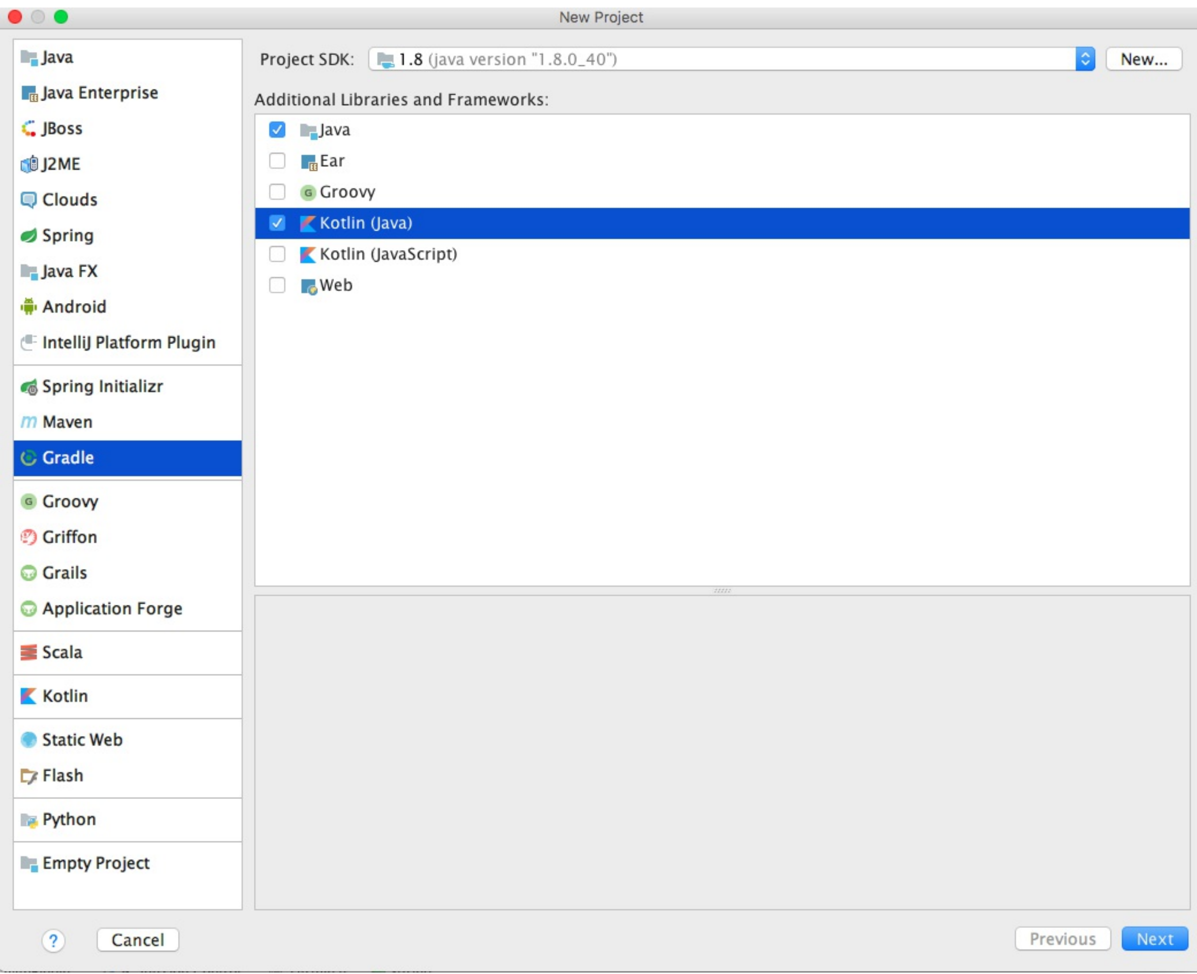
安装配置完Kotlin命令行环境之后，我们直接命令行输入kotlinc, 即可进入 Kotlin REPL界面。

```
$ kotlinc
Welcome to Kotlin version 1.1.2-2 (JRE 1.8.0_40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World!")
Hello,World!

>>> import java.util.Date
>>> Date()
Wed Jun 07 14:19:33 CST 2017
```

2.2 应用程序版HelloWorld

我们如果想拥有学习Kotlin的相对较好的体验，就不建议使用eclipse了。毕竟Kotlin是JetBrains家族的亲儿子，跟Intelli IDEA是血浓于水啊。

我们使用IDEA新建gradle项目，选择Java，Kotlin(Java)框架支持，如下图：



新建完项目，我们写一个HelloWorld.kt类

```
package com.easy.kotlin

/**
 * Created by jack on 2017/5/29.
 */

import java.util.Date
import java.text.SimpleDateFormat

fun main(args: Array<String>) {
    println("Hello, world!")
    println(SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(Date()))
}
```

整体的项目目录结构如下

```
.
├── README.md
├── build
```

```
│   ├── classes
│   │   └── main
│   │       ├── META-INF
│   │       │   └── easykotlin_main.kotlin_module
│   │       └── com
│   │           └── easy
│   │               └── kotlin
│   │                   └── HelloWorldKt.class
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── HelloWorld.kt
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

21 directories, 7 files
```

直接运行HelloWorld.kt，输出结果如下

```
Hello, world!
2017-05-29 01:15:30
```

关于工程的编译、构建、运行，是由gradle协同kotlin-gradle-plugin，在kotlin-stdlib-jre8，kotlin-stdlib核心依赖下完成的。build.gradle配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.1'

    repositories {
        mavenCentral()
    }
    dependencies {
```

```
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

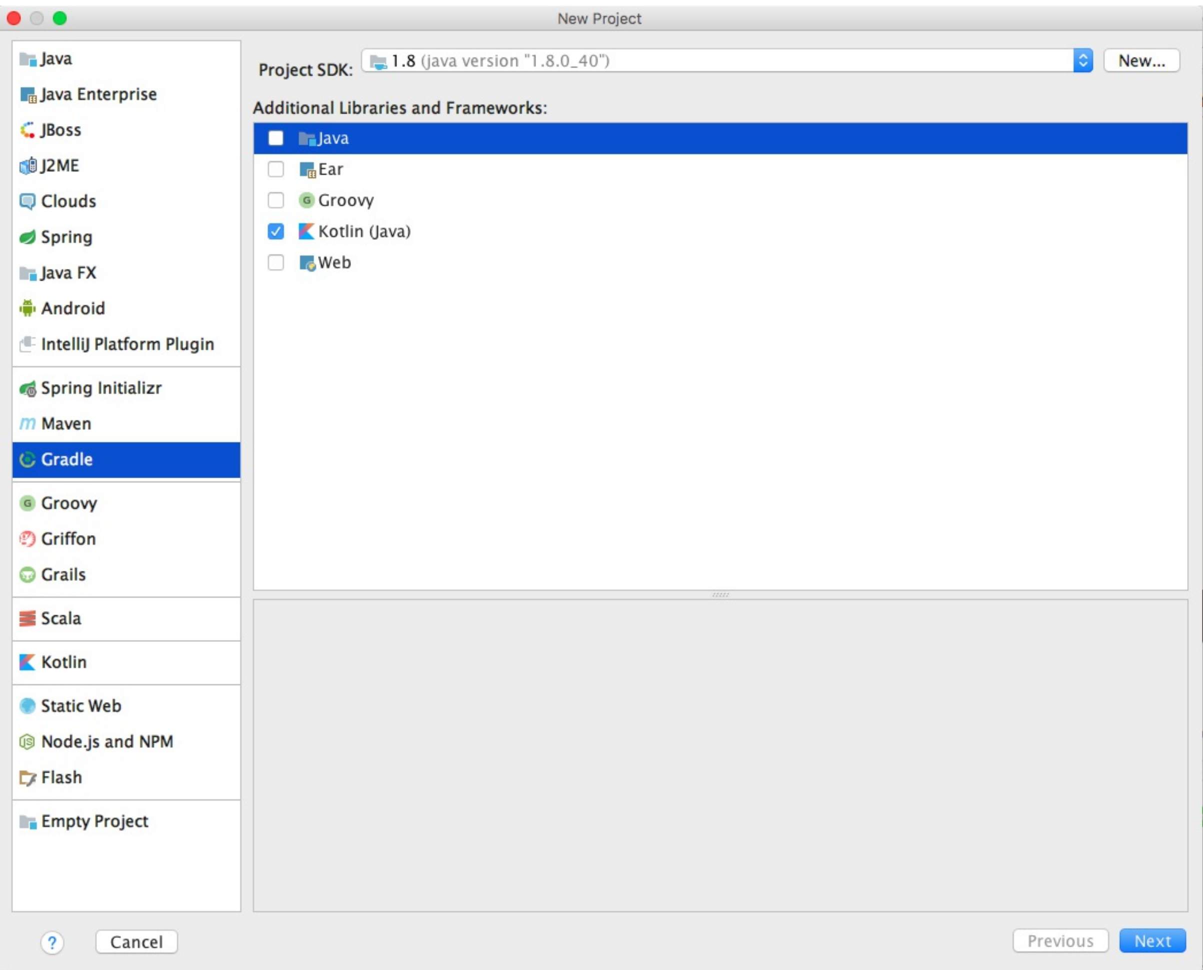
工程源码地址：https://github.com/EasyKotlin/easykotlin/tree/easykotlin_hello_world_20170529

2.3 Web RESTFul HelloWorld

本节介绍使用 `Kotlin` 结合 `SpringBoot` 开发一个RESTFul版本的 `Hello.World` 。

1.新建gradle，kotlin工程：

打开IDEA的 `File > New > Project` ，如下图



按照界面操作，输入相应的工程名等信息，即可新建一个使用Gradle构建的标准Kotlin工程。

2.build.gradle 基本配置

IDEA自动生成的Gradle配置文件如下：

```
group 'com.jason.chen.mini_springboot'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.0'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'kotlin'
apply plugin: 'application'
```

```
mainClassName = 'jason.chen.mini_springboot.HelloWorldKt'

defaultTasks 'run'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

从上面的配置文件我们可以看出，IDEA已经自动把Gradle 构建Kotlin工程插件 kotlin-gradle-plugin，以及Kotlin标准库kotlin-stdlib添加到配置文件中了。

3.配置SpringBoot相关内容

下面我们来配置SpringBoot相关内容。首先在构建脚本里面添加ext变量springBootVersion。

```
ext.kotlin_version = '1.1.2-2'
ext.springboot_version = '1.5.2.RELEASE'
```

然后在构建依赖里添加spring-boot-gradle-plugin

```
buildscript {
...
    dependencies {
//        Kotlin Gradle插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
//        SpringBoot Gradle插件
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$springboot_versi
on")

//        Kotlin整合SpringBoot的默认无参构造函数，默认把所有的类设置open类插件
        classpath("org.jetbrains.kotlin:kotlin-noarg:$kotlin_version")
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlin_version")
    }
}
```

4.配置无参（no-arg）、全开放（allopen）插件

其中，`org.jetbrains.kotlin:kotlin-noarg` 是无参（no-arg）编译器插件，它为具有特定注解的类生成一个额外的零参数构造函数。 这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。 这样我们就可以使用 Java Persistence API（JPA）实例化 data 类。

其中， `org.jetbrains.kotlin:kotlin-allopen` 是全开放编译器插件。我们使用Kotlin 调用Java的Spring AOP框架和库，需要类为 open（可被继承实现），而Kotlin 类和函数都是默认 final 的，这样我们需要为每个类和函数前面加上open修饰符。

这样的代码写起来，可费事了。还好，我们有all-open 编译器插件。它会适配 Kotlin 以满足这些框架的需求，并使用指定的注解标注类而其成员无需显式使用 open 关键字打开。 例如，当我们使用 Spring 时，就不需要打开所有的类，跟我们在Java中写代码一样，只需要用相应的注解标注即可，如 @Configuration 或 @Service。

5.配置application.properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/easykotlin
spring.datasource.username = root
spring.datasource.password = root
#spring.datasource.driverClassName = com.mysql.jdbc.Driver
# Specify the DBMS
spring.jpa.database = MYSQL
# Keep the connection alive if idle for a long time (needed in production)
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
# Show or not log for each sql query
spring.jpa.show-sql = true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto = update
# Naming strategy
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

server.port=8000
```

6.整体工程架构

> UNIX操作系统说， “一切都是文件” 。所以,我们 的所有的源代码、字节码、工程资源文件等等，一切都是文件。文件里面存的是字符串（01也当做是字符） 。各种框架、库、编译器，解释器，都是对这些字符串流进行过滤，最后映射成01机器码（或者CPU微指令码等）， 最终落地到硬件上的高低电平。

整体工程目录如下：

```
.
├── README.md
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
```

```
├── easykotlin.sql
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               ├── Application.kt
    │   │               ├── controller
    │   │               │   ├── HelloWorldController.kt
    │   │               │   └── PeopleController.kt
    │   │               ├── entity
    │   │               │   └── People.kt
    │   │               ├── repository
    │   │               │   └── PeopleRepository.kt
    │   │               └── service
    │   │                   └── PeopleService.kt
    │   └── resources
    │       ├── application.properties
    │       └── banner.txt
    └── test
        ├── java
        ├── kotlin
        └── resources

19 directories, 13 files
```

一切尽在不言中，静静地看工程文件结构。

直接写个HelloWorldController

```kotlin
package com.easy.kotlin.controller

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

/**
 * Created by jack on 2017/6/7.
 */
@RestController
class HelloWorldController {
    @GetMapping(value = *arrayOf("/helloworld", "/"))
    fun helloworld(): Any {
        return "Hello,World!"
    }
}
```

我们再写个访问数据库的标准四层代码

写领域模型类People

```
package com.easy.kotlin.entity

import java.util.*
import javax.persistence.Entity
import javax.persistence.GeneratedValue
import javax.persistence.GenerationType
import javax.persistence.Id

/**
 * Created by jack on 2017/6/6.
 */
@Entity
class People(
        @Id @GeneratedValue(strategy = GenerationType.AUTO)
        val id: Long?,
        val firstName: String?,
        val lastName: String?,
        val gender: String?,
        val age: Int?,
        val gmtCreated: Date,
        val gmtModified: Date
) {
    override fun toString(): String {
        return "People(id=$id, firstName='$firstName', lastName='$lastName', gender='$g
ender', age=$age, gmtCreated=$gmtCreated, gmtModified=$gmtModified)"
    }
}
```

写PeopleRepository

```
package com.easy.kotlin.repository

import com.easy.kotlin.entity.People
import org.springframework.data.repository.CrudRepository

/**
 * Created by jack on 2017/6/7.
 */
interface PeopleRepository : CrudRepository<People, Long> {
    fun findByLastName(lastName: String): List<People>?
}
```

写PeopleService

```
package com.easy.kotlin.service

import com.easy.kotlin.entity.People
import com.easy.kotlin.repository.PeopleRepository
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service

/**
 * Created by jack on 2017/6/7.
 */
@Service
class PeopleService : PeopleRepository {

    @Autowired
    val peopleRepository: PeopleRepository? = null


    override fun findByLastName(lastName: String): List<People>? {
        return peopleRepository?.findByLastName(lastName)
    }

    override fun <S : People?> save(entity: S): S? {
        return peopleRepository?.save(entity)
    }

    override fun <S : People?> save(entities: MutableIterable<S>?): MutableIterable<S>?
 {
        return peopleRepository?.save(entities)
    }

    override fun delete(entities: MutableIterable<People>?) {
    }

    override fun delete(entity: People?) {
    }

    override fun delete(id: Long?) {
    }

    override fun findAll(ids: MutableIterable<Long>?): MutableIterable<People>? {
        return peopleRepository?.findAll(ids)
    }

    override fun findAll(): MutableIterable<People>? {
        return peopleRepository?.findAll()
    }
```

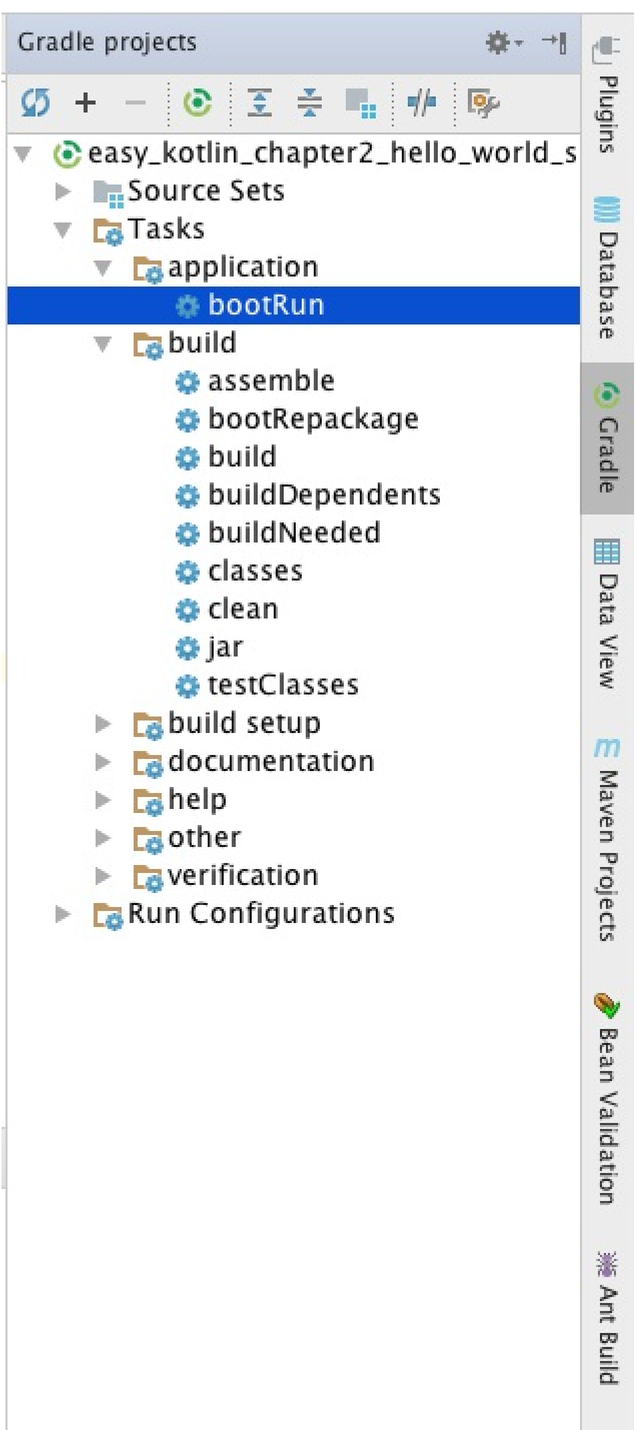
```
    override fun exists(id: Long?): Boolean {
        return peopleRepository?.exists(id)!!
    }

    override fun count(): Long {
        return peopleRepository?.count()!!
    }

    override fun findOne(id: Long?): People? {
        return peopleRepository?.findOne(id)
    }

    override fun deleteAll() {
    }

}
```

写PeopleController

```
package com.easy.kotlin.controller

import com.easy.kotlin.service.PeopleService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody

/**
 * Created by jack on 2017/6/7.
 */
@Controller
class PeopleController {
    @Autowired
    val peopleService: PeopleService? = null

    @GetMapping(value = "/hello")
    @ResponseBody
    fun hello(@RequestParam(value = "lastName") lastName: String): Any {
        val peoples = peopleService?.findByLastName(lastName)
        val map = HashMap<Any, Any>()
        map.put("hello", peoples!!)
        return map
    }

}
```

7.运行测试

点击Gradle的 `bootRun` ，如下图



如果没有异常，启动成功，我们将看到以下输出：

```
Run  easy_kotlin_chapter2_hello_world_springboot_restful [bootRun]
11:47:09: Executing external task 'bootRun'...
:compileKotlin
Using kotlin incremental compilation
:compileJava NO-SOURCE
:copyMainKotlinClasses
:processResources
:classes
:findMainClass
Using a single directory for all classes from a source set. This behaviour has been deprecated and is scheduled to be removed in Gradle
:bootRun
 _  __     _   _ _
| |/ /___ | |_| (_)_ __    _
| ' // _ \| __| | | '_ \ _| |_
| . \ (_) | |_| | | | | |_   _|
|_|\_\___/ \__|_|_|_| |_| |_|

  ____             _             ____              _
 / ___| _ __  _ __(_)_ __   __ _| __ )  ___   ___ | |_
 \___ \| '_ \| '__| | '_ \ / _` |  _ \ / _ \ / _ \| __|
  ___) | |_) | |  | | | | | (_| | |_) | (_) | (_) | |_
 |____/| .__/|_|  |_|_| |_|\__, |____/ \___/ \___/ \__|
       |_|                 |___/

2017-06-07 11:48:04.647  INFO 38248 --- [           main] com.easy.kotlin.ApplicationKt            : Starting ApplicationKt on jacks-Mac
2017-06-07 11:48:04.664  INFO 38248 --- [           main] com.easy.kotlin.ApplicationKt            : No active profile set, falling back
2017-06-07 11:48:04.898  INFO 38248 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot
2017-06-07 11:48:08.886  INFO 38248 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transacti
2017-06-07 11:48:09.872  INFO 38248 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 80
2017-06-07 11:48:09.943  INFO 38248 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
2017-06-07 11:48:09.949  INFO 38248 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tom
2017-06-07 11:48:10.313  INFO 38248 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApp
2017-06-07 11:48:10.314  INFO 38248 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initial
2017-06-07 11:48:10.635  INFO 38248 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet
2017-06-07 11:48:10.647  INFO 38248 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingF
2017-06-07 11:48:10.648  INFO 38248 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFi
2017-06-07 11:48:10.648  INFO 38248 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContent
2017-06-07 11:48:10.648  INFO 38248 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilt
2017-06-07 11:48:11.472  INFO 38248 --- [           main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManage
2017-06-07 11:48:11.534  INFO 38248 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUn
    name: default
    ...]
```

打开浏览器，访问请求：

http://127.0.0.1:8000/

输出响应：

```
Hello,World!
```

访问

http://127.0.0.1:8000/hello?lastName=chen

```
// 20170607115700
// http://127.0.0.1:8000/hello?lastName=chen

{
  "hello": [
    {
      "id": 1,
      "firstName": "Jason",
      "lastName": "Chen",
      "gender": "Male",
      "age": 28,
      "gmtCreated": 1496768497000,
      "gmtModified": 1496768497000
```

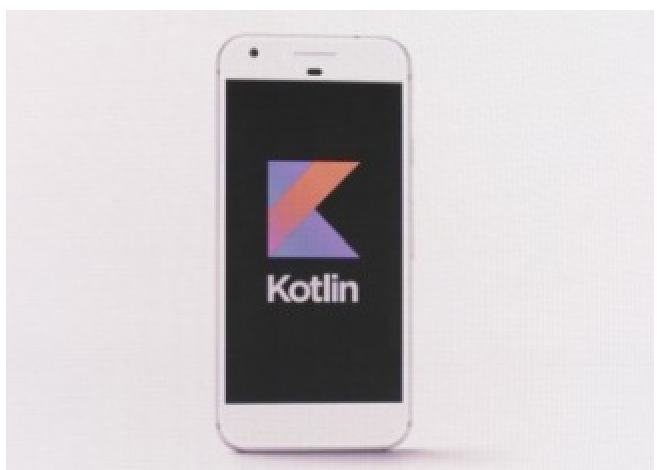
```
    },
    {
      "id": 3,
      "firstName": "Corey",
      "lastName": "Chen",
      "gender": "Female",
      "age": 20,
      "gmtCreated": 1496768497000,
      "gmtModified": 1496768497000
    }
    ...
  ]
}
```

本节示例工程源代码：

https://github.com/EasyKotlin/easy_kotlin_chapter2_hello_world_springboot_restful

2.4 Android版的HelloWorld

2017谷歌I/O大会：宣布 Kotlin 成 Android 开发一级语言。



2017谷歌I/O大会上，谷歌宣布，将Kotlin语言作为安卓开发的一级编程语言。Kotlin由JetBrains公司开发，与Java100%互通，并具备诸多Java尚不支持的新特性。谷歌称还将与JetBrains公司合作，为Kotlin设立一个非盈利基金会。

JetBrains在2010年首次推出Kotlin编程语言，并在次年将之开源。下一版的AndroidStudio（3.0）也将提供支持。

下面我们简要介绍如何在Android上开始一个Kotlin的HelloWorld程序。

对于我们程序员来说，我们正处于一个美好的时代。得益于互联网的发展、工具的进步，我们现在学习一门新技术的成本和难度都比过去低了很多。

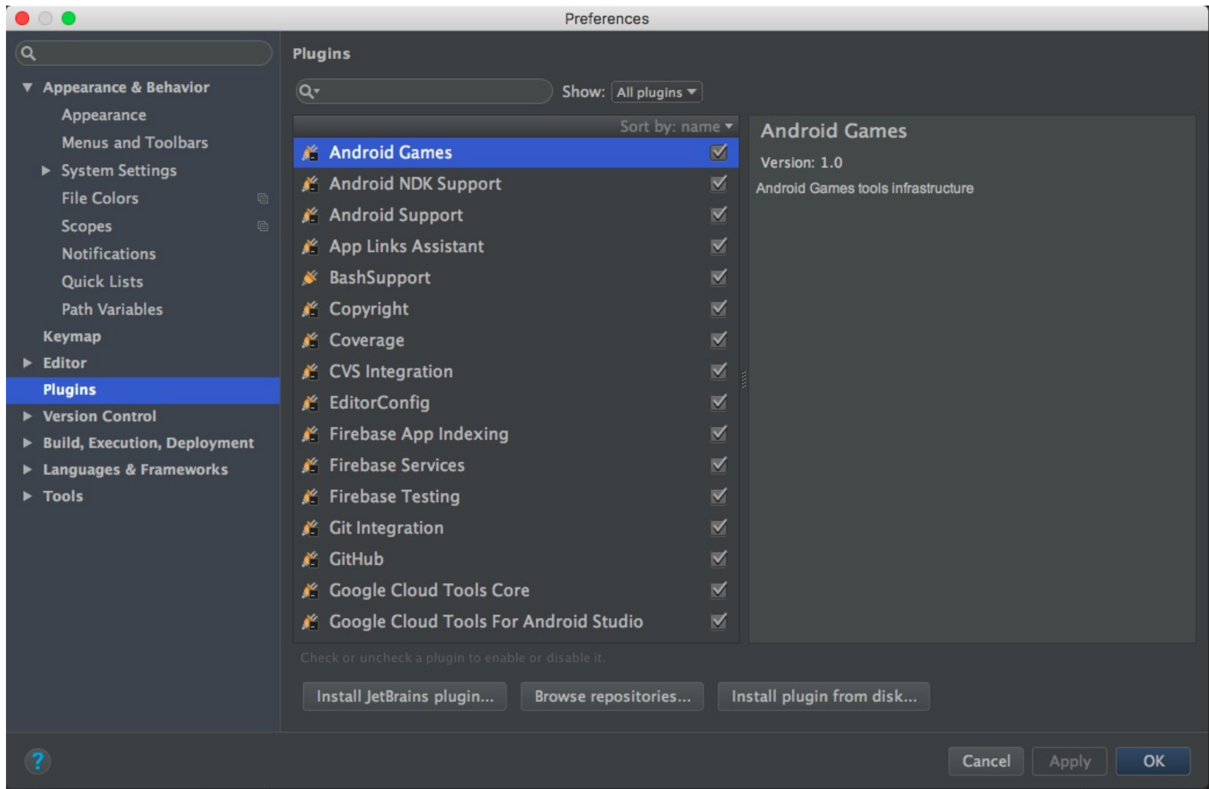
假设你之前没有使用过Kotlin，那么从头开始写一个HelloWorld的app也只需要这么几步：

1.首先，你要有一个Android Studio。 本节中，我们用的是2.2.3版本，其它版本应该也大同小异。

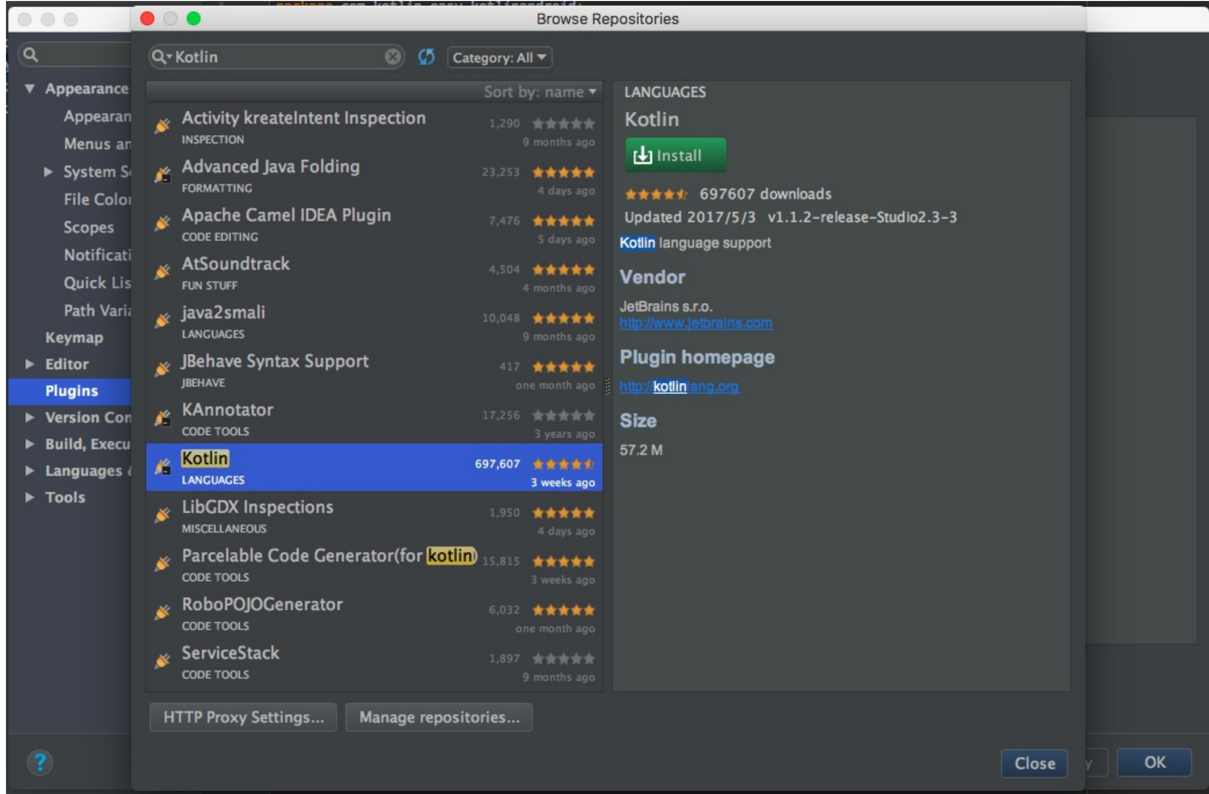
```
Android Studio 2.3.1
Build #AI-162.3871768, built on April 1, 2017
JRE: 1.8.0_112-release-b06 x86_64
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o
```

2.其次，安装一个Kotlin的插件。

依次打开：Android Studio > Preferences > Plugins,



然后选择『Browse repositories』，在搜索框中搜索Kotlin，结果列表中的『Kotlin』插件，如下图



点击安装，安装完成之后，重启Android Studio。

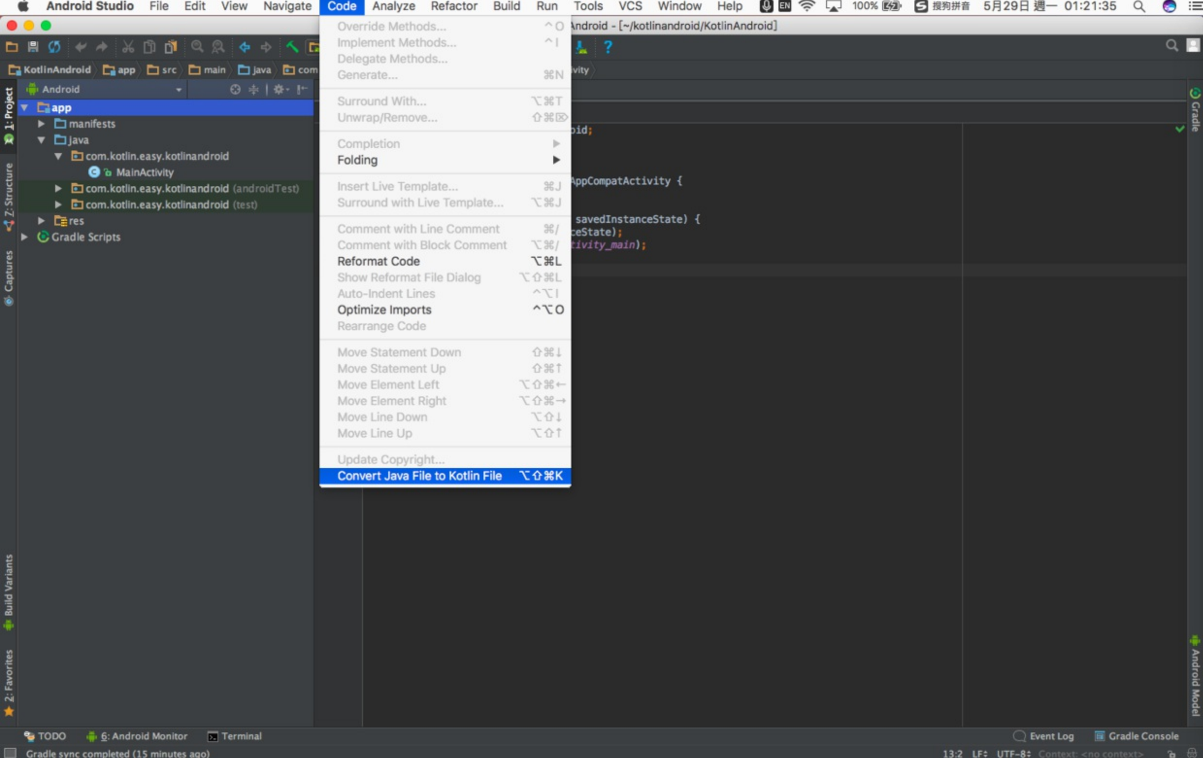
3.新建一个Android项目

重新打开Android Studio，新建一个Android项目吧，添加一个默认的MainActivity——像以前一样即可。

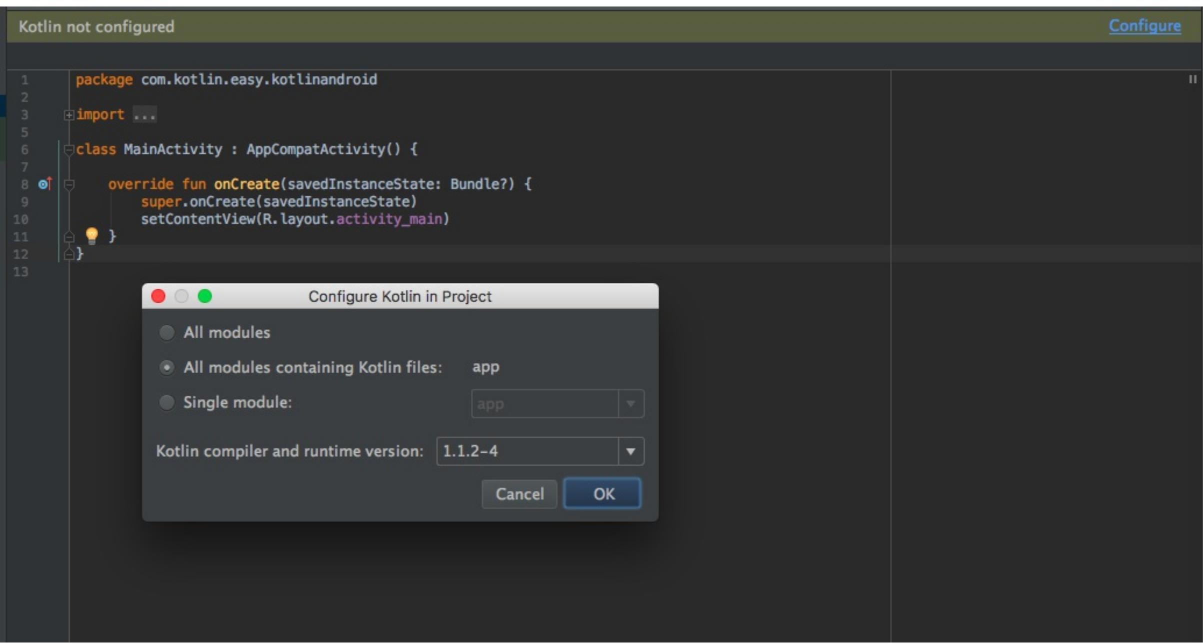
4.转换Java to Kotlin

安装完插件的AndroidStudio现在已经拥有开发Kotlin的功能。我们先来尝试它的转换功能：Java -> Kotlin，可以把现有的java文件翻译成Kotlin文件。

打开MainActivity文件，在Code菜单下面可以看到一个新的功能：Convert Java File to Kotlin File。



点击转换，



可以看到转换后的Kotlin文件：MainActivity.kt

```
package com.kotlin.easy.kotlinandroid

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

这个转换功能，对我们Java程序员在学习Kotlin是十分实用。我们可以基于我们之前的Java编码的经验来迅速学习Kotlin编程。

5.配置gradle文件

MainActivity已经被转换成了Kotlin实现，但是项目目前gradle编译、构建、运行还不能执行，还需要进一步配置一下，让项目支持grade的编译、运行。当然，这一步也不需要我们做太多工作——IDEA都已经帮我们做好了。

在Java代码转换成Kotlin代码之后，打开MainActivity.kt文件，编译器会提示"Kotlin not configured"，点击一下Configure按钮，IDEA就会自动帮我们把配置文件写好了。

我们可以看出，主要的依赖项是：

```
kotlin-gradle-plugin
plugin: 'kotlin-android'
kotlin-stdlib-jre7
```

完整的配置文件如下：

Project build.gradle

```
// Top-level build file where you can add configuration options common to all sub-proje
cts/modules.

buildscript {
    ext.kotlin_version = '1.1.2-4'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
```

```
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Module build.gradle

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.kotlin.easy.kotlinandroid"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rul
es.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
}
repositories {
    mavenCentral()
}
```

所以说使用IDEA来写Kotlin代码，这工具的完美集成会让你用起来如丝般润滑。毕竟Kotlin的亲爸爸JetBrains是专门做工具的，而且Intelli IDEA又是那么敏捷、智能。

配置之后，等Gradle Sync完成，即可运行。

6.运行

运行结果如下

2:40
KotlinAndroid
Hello World!

工程源码：https://github.com/EasyKotlin/KotlinAndroid

2.5 JavaScript版HelloWorld

在Kotlin 1.1中，开始支持JavaScript和协程是引人注目的亮点。本节我们简单介绍Kotlin代码编译转化为JavaScript的方法。

为了极简直观地感受这个过程，我们先在命令行REPL环境体验一下Kotlin源码被编译生成对应的JavaScript代码的过程。

首先，使用编辑器新建一个HelloWord.kt

```
fun helloWorld(){
    println("Hello,World!")
}
```

命令行使用 `kotlinc-js` 编译

```
kotlinc-js -output HelloWorld.js HelloWorld.kt
```

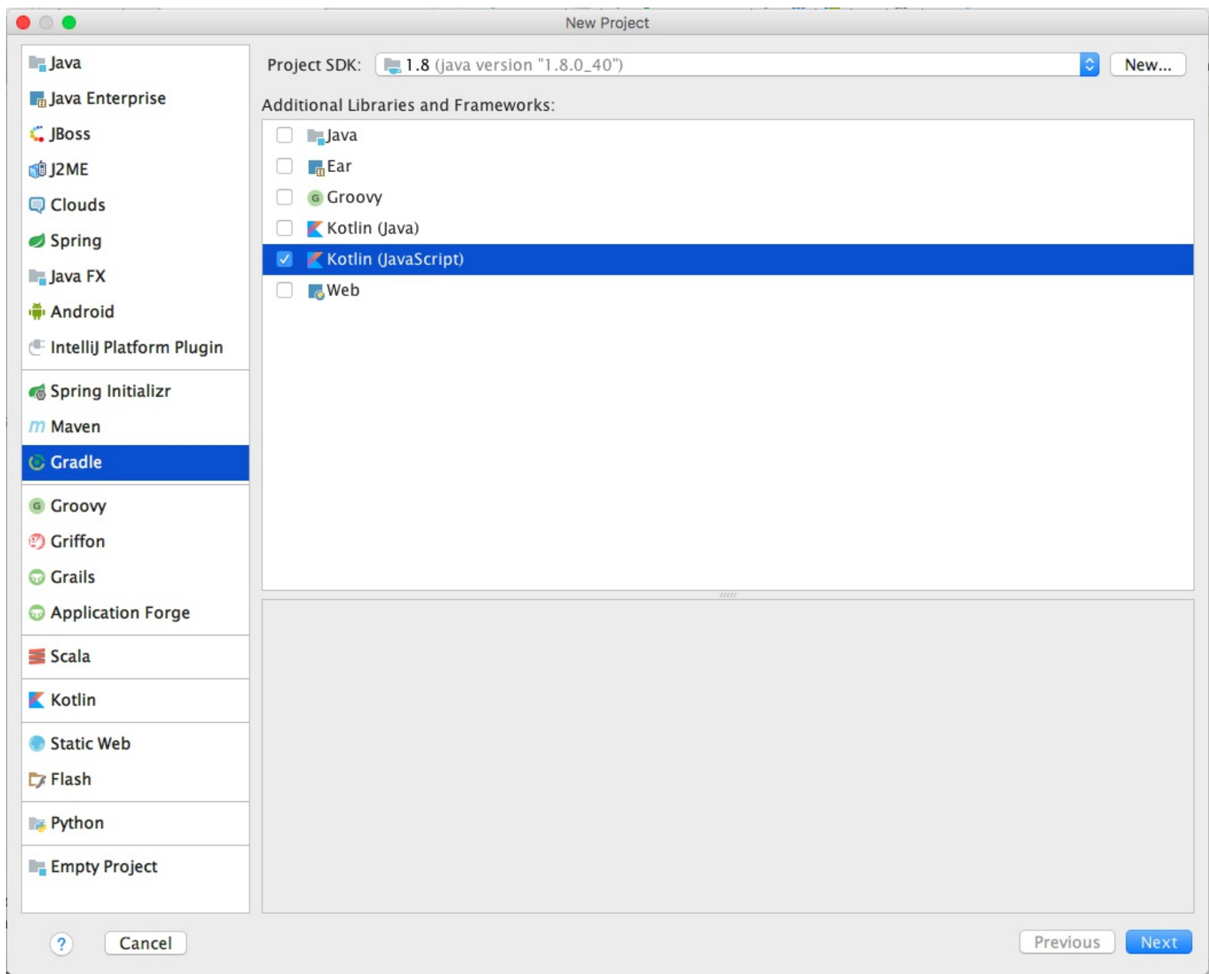
运行完毕，我们会在当前目录下看到 `HelloWorld.js` , 其内容如下

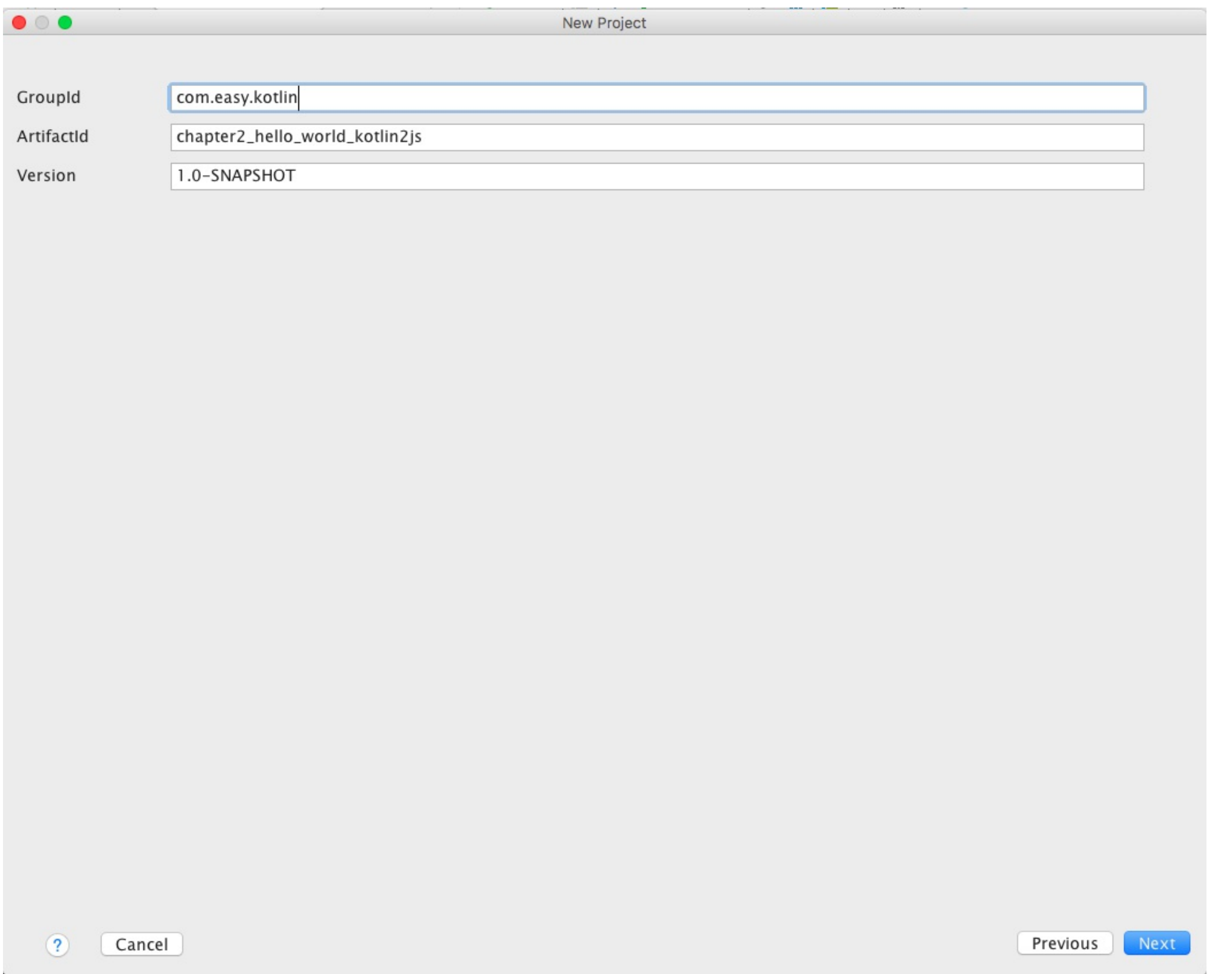
```
if (typeof kotlin === 'undefined') {
  throw new Error("Error loading module 'HelloWorld'. Its dependency 'kotlin' was not found. Please, check whether 'kotlin' is loaded prior to 'HelloWorld'.");
}
var HelloWorld = function (_, Kotlin) {
  'use strict';
  var println = Kotlin.kotlin.io.println_s8jyv4$;
  function helloWorld() {
    println('Hello,World!');
  }
  _.helloWorld = helloWorld;
  Kotlin.defineModule('HelloWorld', _);
  return _;
}(typeof HelloWorld === 'undefined' ? {} : HelloWorld, kotlin);
```

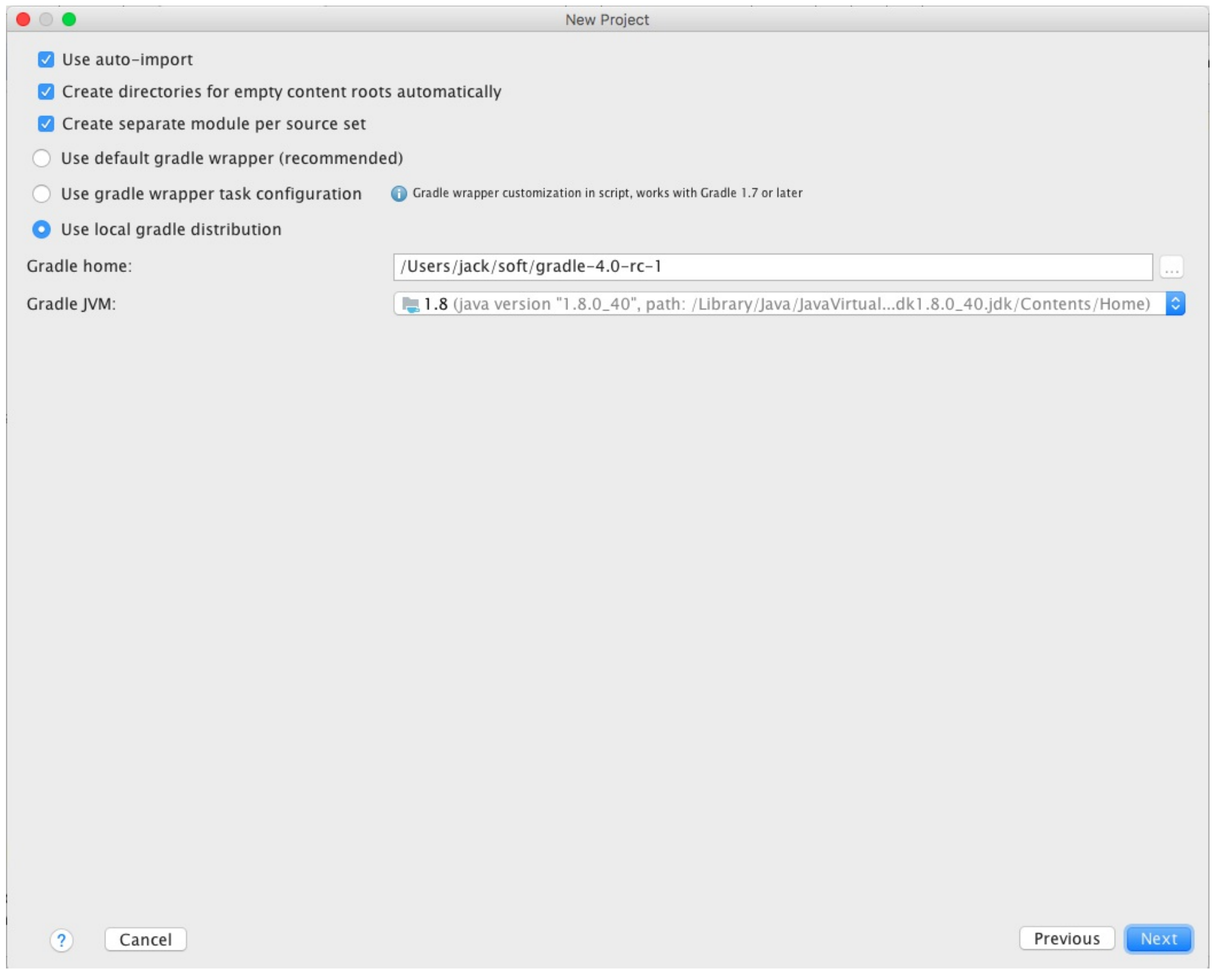
我们看到，使用 `kotlinc-js` 转换成的js代码依赖'kotlin'模块。这个模块是Kotlin支持JavaScript脚本的内部封装模块。也就是说，如果我们想要使用 `HelloWorld.js` ，先要引用 `kotlin.js` 。这个 `kotlin.js` 在kotlin-stdlib-js-1.1.2.jar里面。

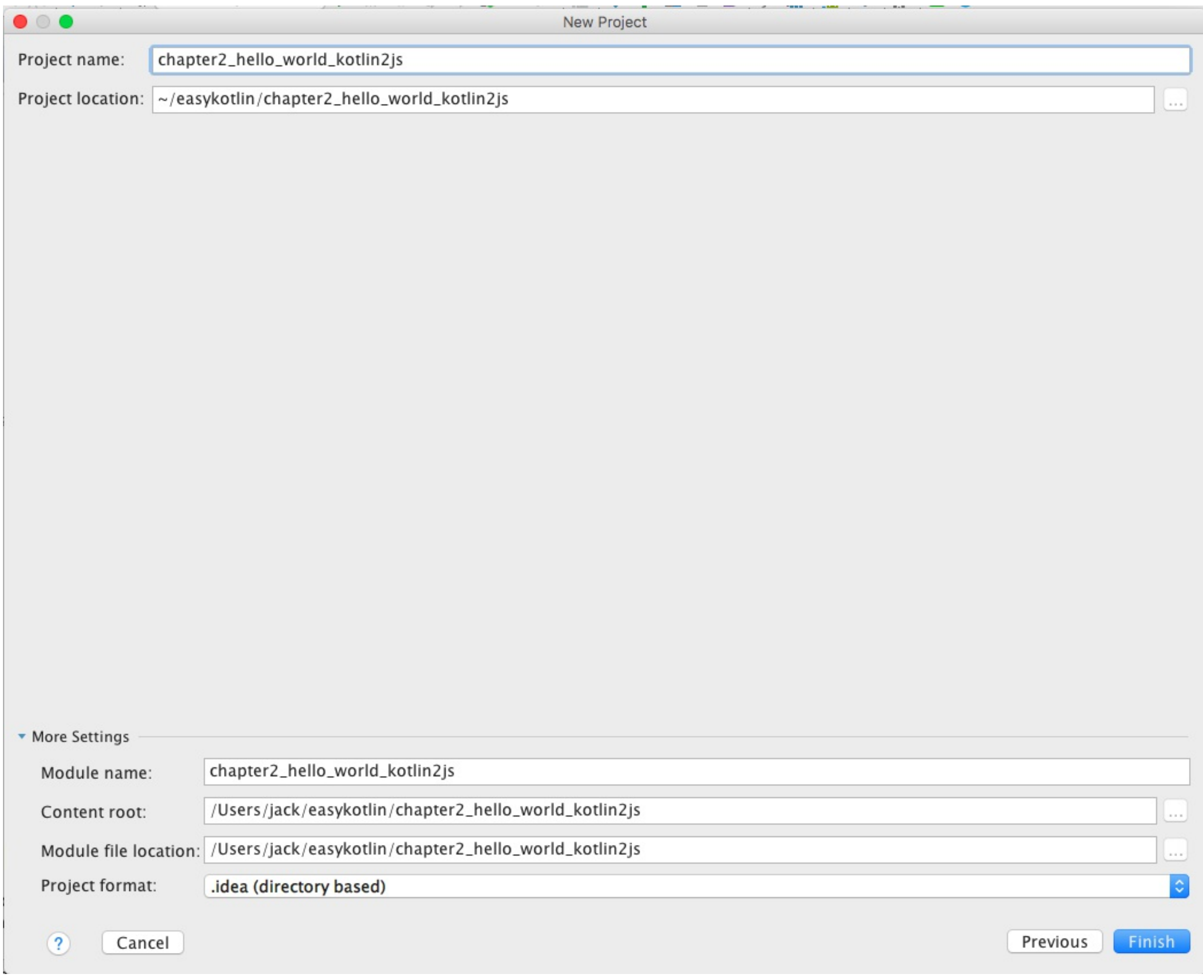
下面我们使用IDEA新建一个Kotlin（JavaScript）工程。在这个过程中，我们将会看到使用Kotlin来开发js的过程。

首先按照以下步骤新建工程


New Project
Java
Java Enterprise
JBoss
J2ME
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Griffon
Grails
Application Forge
Scala
Kotlin
Static Web
Flash
Python
Empty Project
Project SDK:
1.8 (java version "1.8.0_40")
New...
Additional Libraries and Frameworks:
Java
Ear
Groovy
Kotlin (Java)
Kotlin (JavaScript)
Web
Cancel
Previous
Next


New Project
GroupId
com.easy.kotlin
ArtifactId
chapter2_hello_world_kotlin2js
Version
1.0-SNAPSHOT
?
Cancel
Previous
Next


New Project
Use auto-import
Create directories for empty content roots automatically
Create separate module per source set
Use default gradle wrapper (recommended)
Use gradle wrapper task configuration
Gradle wrapper customization in script, works with Gradle 1.7 or later
Use local gradle distribution
Gradle home:
/Users/jack/soft/gradle-4.0-rc-1
Gradle JVM:
1.8 (java version "1.8.0_40", path: /Library/Java/JavaVirtual...dk1.8.0_40.jdk/Contents/Home)
Cancel
Previous
Next



等待Gradle初始化工程完毕，我们将得到一个Gradle KotlinJS 工程，其目录如下

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
```

其中，build.gradle配置文件为

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'kotlin2js'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
}
```

其中，apply plugin: 'kotlin2js' 是Gradle的kotlin编译成js的插件。org.jetbrains.kotlin:kotlin-stdlib-js 是KotlinJS的运行库。

另外，我们需要再配置一下Kotlin代码编译成JS的编译规则，以及文件放置目录等属性，如下所示

```
build.doLast {
    configurations.compile.each { File file ->
        copy {
            includeEmptyDirs = false

            from zipTree(file.absolutePath)
            into "${projectDir}/web"
            include { fileTreeElement ->
                def path = fileTreeElement.path
                path.endsWith(".js") && (path.startsWith("META-INF/resources/") || !pat
h.startsWith("META-INF/"))
            }
        }
    }
}

compileKotlin2Js {
    kotlinOptions.outputFile = "${projectDir}/web/js/app.js"
```

```
    kotlinOptions.moduleKind = "plain" // plain (default),AMD,commonjs,umd
    kotlinOptions.sourceMap = true
    kotlinOptions.verbose = true
    kotlinOptions.suppressWarnings = true
    kotlinOptions.metaInfo = true
}
```

其中， `kotlinOptions.moduleKind` 配置项是Kotlin代码编译成JavaScript代码的类型。 支持普通JS（plain），AMD(Asynchronous Module Definition，异步模块定义)、CommonJS和UMD(Universal Model Definition，通用模型定义)。

AMD通常在浏览器的客户端使用。AMD是异步加载模块，可用性和性能相对会好。

CommonJS是服务器端上使用的模块系统，通常用于nodejs。

UMD是想综合AMD、CommonJS这两种模型，同时支持它们在客户端或服务器端上使用。

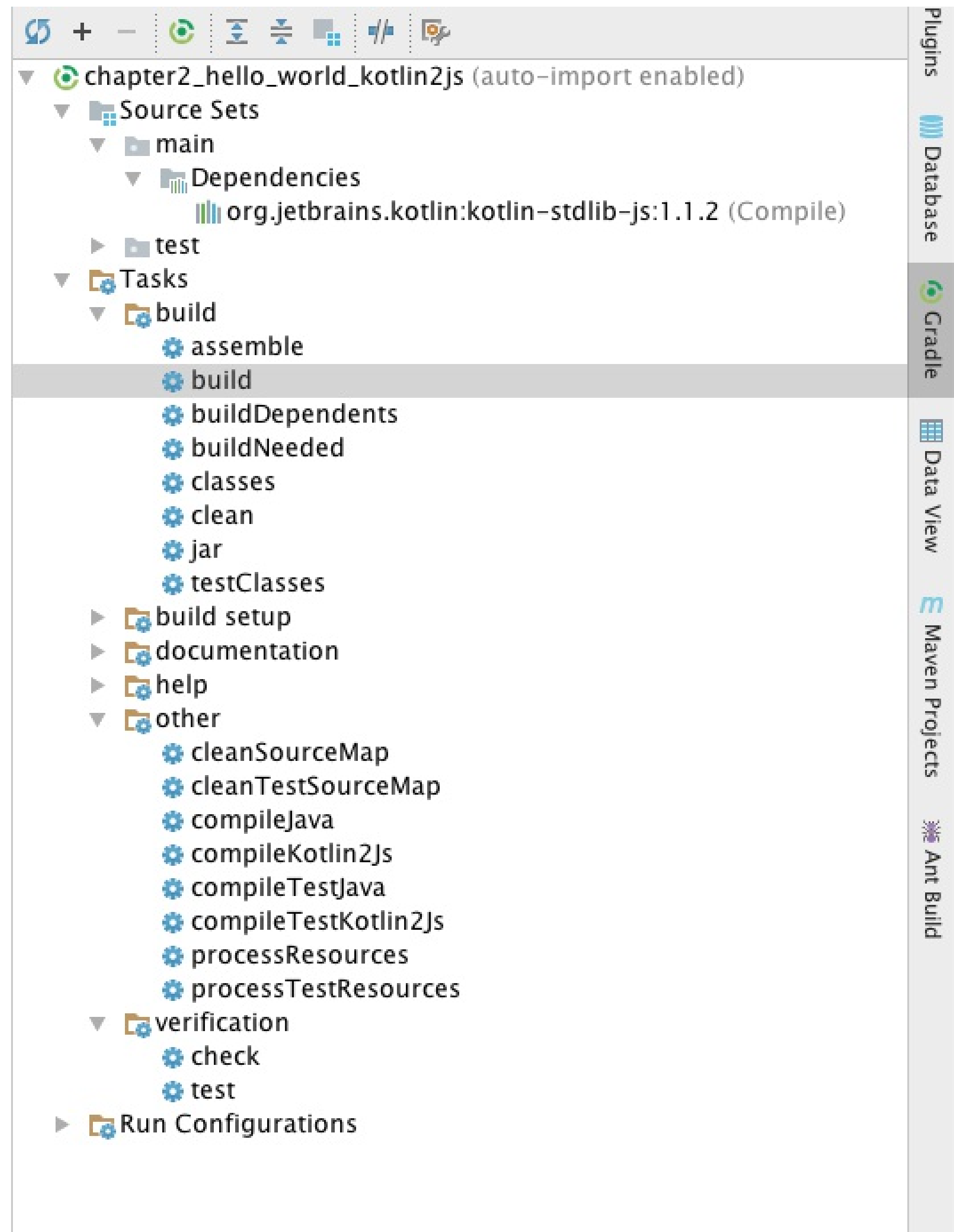
我们这里为了极简化演示，直接采用了普通JS `plain` 类型。

除了输出的 JavaScript 文件，该插件默认会创建一个带二进制描述符的额外 JS 文件。 如果你是构建其他 Kotlin 模块可以依赖的可重用库，那么该文件是必需的，并且应该与转换结果一起分发。 其生成由 kotlinOptions.metaInfo 选项控制。

一切配置完毕，我们来写Kotlin代码App.kt

```kotlin
package com.easy.kotlin

/**
 * Created by jack on 2017/6/7.
 */

fun helloWorld() {
    println("Hello,World!")
}
```

然后，我们直接使用Gradle构建工程，如下图所示



控制台输出

```
23:47:05: Executing external task 'build'...
Using a single directory for all classes from a source set. This behaviour has been dep
recated and is scheduled to be removed in Gradle 5.0
    at build_3e0ikl0qk0r006tvk0olcp2lu.run(/Users/jack/easykotlin/chapter2_hello_world_
kotlin2js/build.gradle:15)
:compileJava NO-SOURCE
:compileKotlin2Js
:processResources NO-SOURCE
:classes
```

```
:jar
:assemble
:compileTestJava NO-SOURCE
:compileTestKotlin2Js NO-SOURCE
:processTestResources NO-SOURCE
:testClasses UP-TO-DATE
:test NO-SOURCE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL in 2s
3 actionable tasks: 3 executed
23:47:08: External task execution finished 'build'.
```

此时，我们可以看到工程目录变为

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
jack@jacks-MacBook-Air:~/easykotlin/chapter2_hello_world_kotlin2js$ tree ..
├── build
│   ├── kotlin
│   │   └── sessions
│   ├── kotlin-build
│   │   └── caches
│   │       └── version.txt
│   ├── libs
│   │   └── chapter2_hello_world_kotlin2js-1.0-SNAPSHOT.jar
│   └── tmp
│       └── jar
│           └── MANIFEST.MF
├── build.gradle
├── settings.gradle
```

```
├── src
│   ├── main
│   │   ├── java
│   │   ├── kotlin
│   │   │   └── com
│   │   │       └── easy
│   │   │           └── kotlin
│   │   │               └── App.kt
│   │   └── resources
│   └── test
│       ├── java
│       ├── kotlin
│       └── resources
└── web
    ├── js
    │   ├── app
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── kotlin.kjsm
    │   ├── app.js
    │   ├── app.js.map
    │   └── app.meta.js
    ├── kotlin.js
    └── kotlin.meta.js

26 directories, 14 files
```

这个web目录就是Kotlin代码通过kotlin-stdlib-js-1.1.2.jar编译的输出结果。

其中，app.js代码如下

```
if (typeof kotlin === 'undefined') {
  throw new Error("Error loading module 'app'. Its dependency 'kotlin' was not found. Please, check whether 'kotlin' is loaded prior to 'app'.");
}
var app = function (_, Kotlin) {
  'use strict';
  var println = Kotlin.kotlin.io.println_s8jyv4$;
  function helloWorld() {
    println('Hello,World!');
  }
  var package$com = _.com || (_.com = {});
  var package$easy = package$com.easy || (package$com.easy = {});
  var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
  package$kotlin.helloWorld = helloWorld;
  Kotlin.defineModule('app', _);
  return _;
```

```
}(typeof app === 'undefined' ? {} : app, kotlin);

//@ sourceMappingURL=app.js.map
```

里面这段自动生成的代码显得有点绕

```
var package$com = _.com || (_.com = {});
  var package$easy = package$com.easy || (package$com.easy = {});
  var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
  package$kotlin.helloWorld = helloWorld;

  Kotlin.defineModule('app', _);
  return _;
```

简化之后的意思表达如下

```

_.com.easy.kotlin.helloWorld = helloWorld;
```

目的是建立Kotlin代码跟JavaScript代码的映射关系。这样我们在前端代码中调用

```
  function helloWorld() {
    println('Hello,World!');
  }
```

这个函数时，只要这样调用即可

```
app.com.easy.kotlin.helloWorld()
```

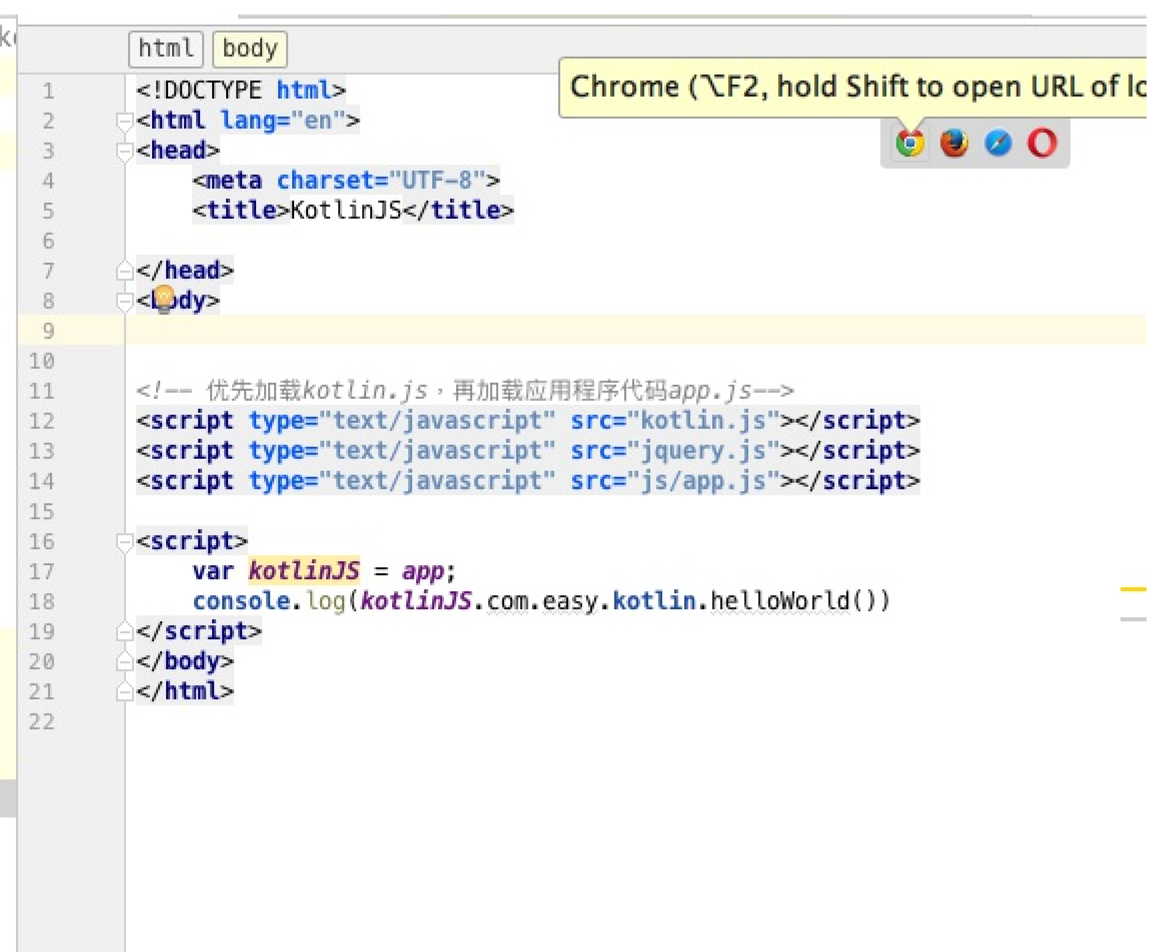
下面我们来新建一个index.html页面，使用我们生成的app.js。代码如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>KotlinJS</title>

</head>
<body>


<!-- 优先加载kotlin.js，再加载应用程序代码app.js-->
<script type="text/javascript" src="kotlin.js"></script>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="js/app.js"></script>
```

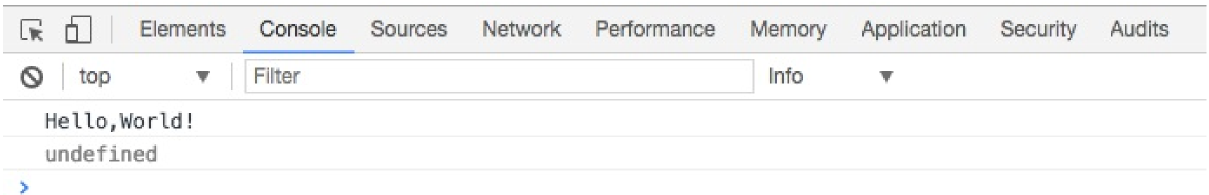
```
<script>
    var kotlinJS = app;
    console.log(kotlinJS.com.easy.kotlin.helloWorld())
</script>
</body>
</html>
```

我们需要优先加载kotlin.js，再加载应用程序代码app.js。当然，我们仍然可以像以前一样使用诸如jquery.js这样的库。

在浏览器中打开index.html



我们可以看到浏览器控制台输出



这个helloWorld() JavaScript函数

```
var println = Kotlin.kotlin.io.println_s8jyv4$;
function helloWorld() {
  println('Hello,World!');
}
```

对应kotlin.js代码中的3755行处的代码：

```
BufferedOutputToConsoleLog.prototype.flush = function() {
    console.log(this.buffer);
    this.buffer = "";
  };
```

参考资料

1.https://kotlinlang.org/docs/reference/compiler-plugins.html

2.http://kotlinlang.org/docs/tutorials/javascript/working-with-modules/working-with-modules.html

第2章 快速开始：HelloWorld

2.1 命令行的HelloWorld

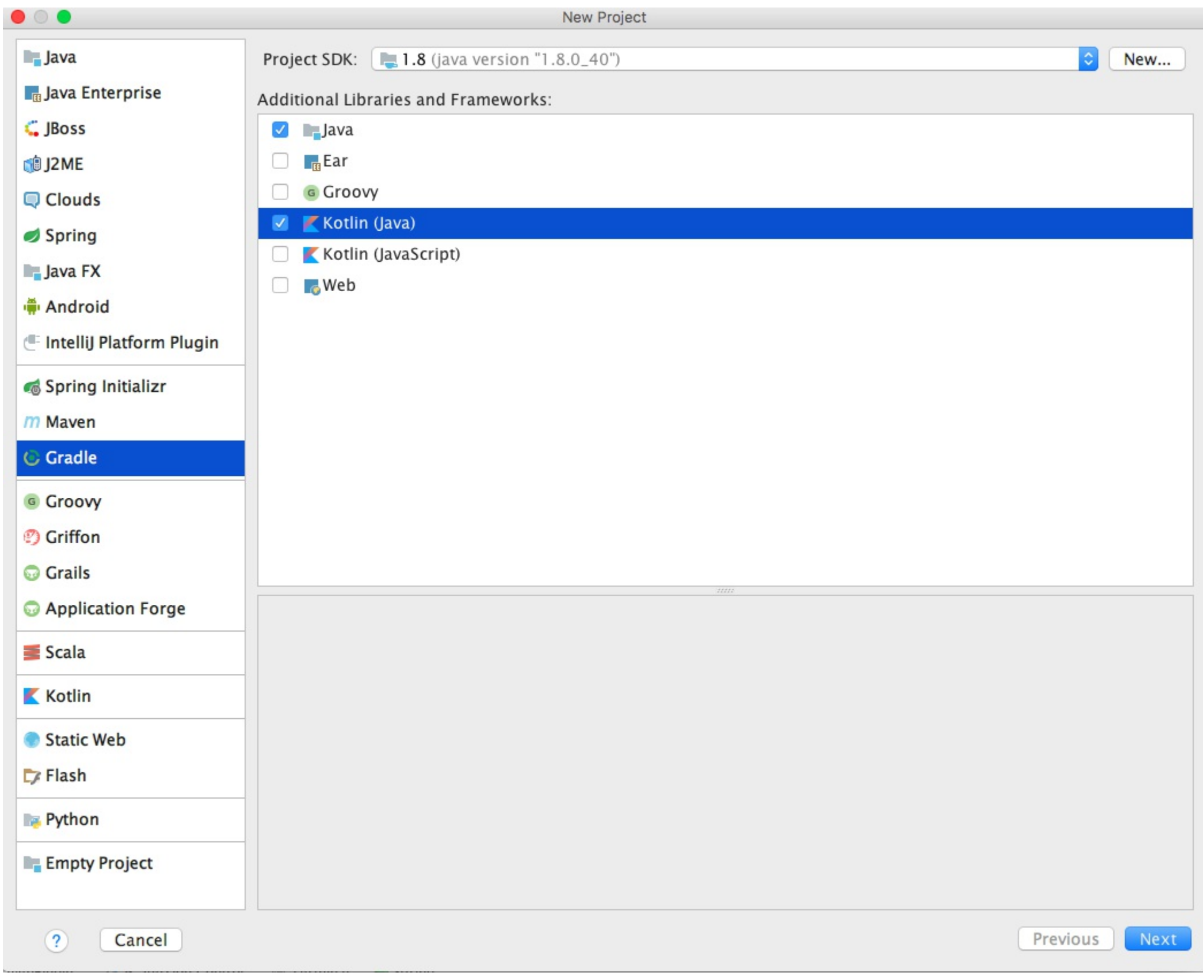
安装配置完Kotlin命令行环境之后，我们直接命令行输入kotlinc, 即可进入 Kotlin REPL界面。

```
$ kotlinc
Welcome to Kotlin version 1.1.2-2 (JRE 1.8.0_40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World!")
Hello,World!

>>> import java.util.Date
>>> Date()
Wed Jun 07 14:19:33 CST 2017
```

2.2 应用程序版HelloWorld

我们如果想拥有学习Kotlin的相对较好的体验，就不建议使用eclipse了。毕竟Kotlin是JetBrains家族的亲儿子，跟Intelli IDEA是血浓于水啊。

我们使用IDEA新建gradle项目，选择Java，Kotlin(Java)框架支持，如下图：



新建完项目，我们写一个HelloWorld.kt类

```
package com.easy.kotlin

/**
 * Created by jack on 2017/5/29.
 */

import java.util.Date
import java.text.SimpleDateFormat

fun main(args: Array<String>) {
    println("Hello, world!")
    println(SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(Date()))
}
```

整体的项目目录结构如下

```
.
├── README.md
├── build
```

```
│   ├── classes
│   │   └── main
│   │       ├── META-INF
│   │       │   └── easykotlin_main.kotlin_module
│   │       └── com
│   │           └── easy
│   │               └── kotlin
│   │                   └── HelloWorldKt.class
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── HelloWorld.kt
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

21 directories, 7 files
```

直接运行HelloWorld.kt，输出结果如下

```
Hello, world!
2017-05-29 01:15:30
```

关于工程的编译、构建、运行，是由gradle协同kotlin-gradle-plugin，在kotlin-stdlib-jre8，kotlin-stdlib核心依赖下完成的。build.gradle配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.1'

    repositories {
        mavenCentral()
    }
    dependencies {
```

```
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

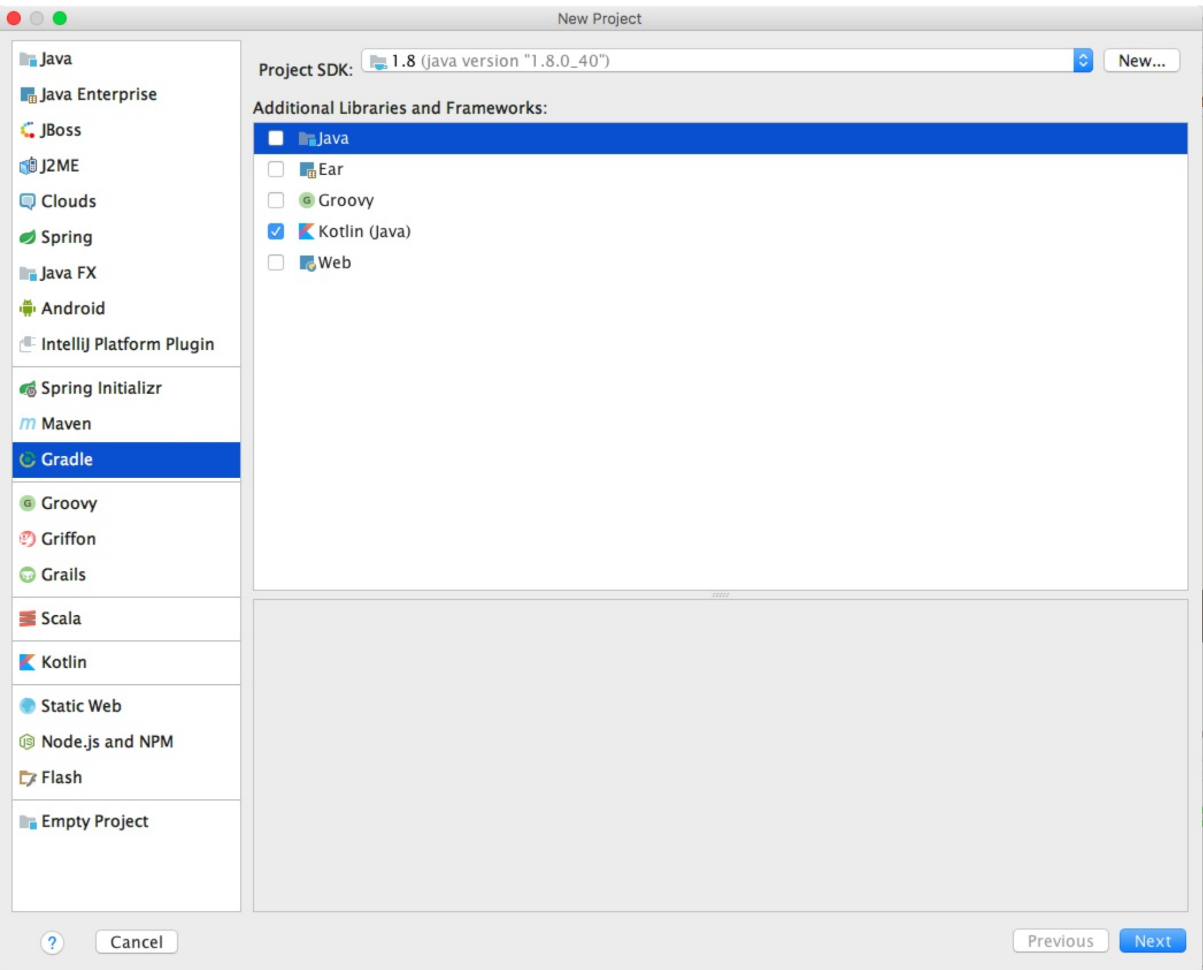
工程源码地址：https://github.com/EasyKotlin/easykotlin/tree/easykotlin_hello_world_20170529

2.3 Web RESTFul HelloWorld

本节介绍使用 `Kotlin` 结合 `SpringBoot` 开发一个RESTFul版本的 `Hello.World` 。

1. 新建gradle，kotlin工程：

打开IDEA的 `File > New > Project` ，如下图



按照界面操作，输入相应的工程名等信息，即可新建一个使用Gradle构建的标准Kotlin工程。

1. build.gradle 基本配置

IDEA自动生成的Gradle配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2-2'

    repositories {
        mavenCentral()
    }
    dependencies {
//         Kotlin Gradle插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

    }
}

apply plugin: 'java'
```

```
apply plugin: 'kotlin'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'

}
```

从上面的配置文件我们可以看出，IDEA已经自动把Gradle 构建Kotlin工程插件 kotlin-gradle-plugin，以及Kotlin标准库kotlin-stdlib添加到配置文件中了。

1. 配置SpringBoot相关内容

下面我们来配置SpringBoot相关内容。首先在构建脚本里面添加ext变量springBootVersion。

```
ext.kotlin_version = '1.1.2-2'
ext.springboot_version = '1.5.2.RELEASE'
```

然后在构建依赖里添加spring-boot-gradle-plugin

```
buildscript {
...
    dependencies {
//        Kotlin Gradle插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
//        SpringBoot Gradle插件
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$springboot_versi
on")

//        Kotlin整合SpringBoot的默认无参构造函数，默认把所有的类设置open类插件
        classpath("org.jetbrains.kotlin:kotlin-noarg:$kotlin_version")
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlin_version")
    }
}
```

1. 配置无参（no-arg）、全开放（allopen）插件

其中，`org.jetbrains.kotlin:kotlin-noarg` 是无参（no-arg）编译器插件，它为具有特定注解的类生成一个额外的零参数构造函数。 这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。 这样我们就可以使用 Java Persistence API（JPA）实例化 data

类。

其中， `org.jetbrains.kotlin:kotlin-allopen` 是全开放编译器插件。我们使用Kotlin 调用Java的Spring AOP框架和库，需要类为 open（可被继承实现），而Kotlin 类和函数都是默认 final 的，这样我们需要为每个类和函数前面加上open修饰符。

这样的代码写起来，可费事了。还好，我们有all-open 编译器插件。它会适配 Kotlin 以满足这些框架的需求，并使用指定的注解标注类而其成员无需显式使用 open 关键字打开。 例如，当我们使用 Spring 时，就不需要打开所有的类，跟我们在Java中写代码一样，只需要用相应的注解标注即可，如 @Configuration 或 @Service。

完整的build.gradle配置文件如下

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2-2'
    ext.springboot_version = '1.5.2.RELEASE'

    repositories {
        mavenCentral()
    }
    dependencies {
//        Kotlin Gradle插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
//        SpringBoot Gradle插件
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$springboot_versi
on")

//        Kotlin整合SpringBoot的默认无参构造函数，默认把所有的类设置open类插件
//        无参（no-arg）编译器插件为具有特定注解的类生成一个额外的零参数构造函数。 这个生成的
构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。 这允许 Java Pers
istence API（JPA）实例化 data 类，虽然它从 Kotlin 或 Java 的角度看没有无参构造函数
        classpath("org.jetbrains.kotlin:kotlin-noarg:$kotlin_version")
//        全开放插件（kotlin-allopen）
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlin_version")
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

//Kotlin整合SpringBoot需要的spring，jpa，org.springframework.boot插件

//Kotlin-spring 编译器插件，它根据 Spring 的要求自动配置全开放插件。
apply plugin: 'kotlin-spring'
//该插件指定 @Entity 和 @Embeddable 注解作为应该为一个类生成无参构造函数的标记。
```

```
apply plugin: 'kotlin-jpa'
apply plugin: 'org.springframework.boot'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'

    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
    compile("org.springframework.boot:spring-boot-starter-data-jpa")
    compile('mysql:mysql-connector-java:5.1.13')

}
```

1. 配置application.properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/easykotlin
spring.datasource.username = root
spring.datasource.password = root
#spring.datasource.driverClassName = com.mysql.jdbc.Driver
# Specify the DBMS
spring.jpa.database = MYSQL
# Keep the connection alive if idle for a long time (needed in production)
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
# Show or not log for each sql query
spring.jpa.show-sql = true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto = update
# Naming strategy
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

server.port=8000
```

1. 整体工程架构

> UNIX操作系统说，“一切都是文件”。所以,我们 的所有的源代码、字节码、工程资源文件等等，一切都是文件。文件里面存的是字符串（01也当做是字符）。各种框架、库、编译器，解释器，都是对这些字符串流进行过滤，最后映射成01机器码（或者CPU微指令码等），最终落地到硬件上的高低电平。

整体工程目录如下：

```
.
├── README.md
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── easykotlin.sql
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               ├── Application.kt
    │   │               ├── controller
    │   │               │   ├── HelloWorldController.kt
    │   │               │   └── PeopleController.kt
    │   │               ├── entity
    │   │               │   └── People.kt
    │   │               ├── repository
    │   │               │   └── PeopleRepository.kt
    │   │               └── service
    │   │                   └── PeopleService.kt
    │   └── resources
    │       ├── application.properties
    │       └── banner.txt
    └── test
        ├── java
        ├── kotlin
        └── resources

19 directories, 13 files
```

一切尽在不言中，静静地看工程文件结构。

直接写个HelloWorldController

```
package com.easy.kotlin.controller

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

/**
 * Created by jack on 2017/6/7.
 */
@RestController
class HelloWorldController {
    @GetMapping(value = *arrayOf("/helloworld", "/"))
    fun helloworld(): Any {
        return "Hello,World!"
    }
}
```

我们再写个访问数据库的标准四层代码

写领域模型类People

```
package com.easy.kotlin.entity

import java.util.*
import javax.persistence.Entity
import javax.persistence.GeneratedValue
import javax.persistence.GenerationType
import javax.persistence.Id

/**
 * Created by jack on 2017/6/6.
 */
@Entity
class People(
        @Id @GeneratedValue(strategy = GenerationType.AUTO)
        val id: Long?,
        val firstName: String?,
        val lastName: String?,
        val gender: String?,
        val age: Int?,
        val gmtCreated: Date,
        val gmtModified: Date
) {
    override fun toString(): String {
        return "People(id=$id, firstName='$firstName', lastName='$lastName', gender='$g
ender', age=$age, gmtCreated=$gmtCreated, gmtModified=$gmtModified)"
    }
}
```

写PeopleRepository

```
package com.easy.kotlin.repository

import com.easy.kotlin.entity.People
import org.springframework.data.repository.CrudRepository

/**
 * Created by jack on 2017/6/7.
 */
interface PeopleRepository : CrudRepository<People, Long> {
    fun findByLastName(lastName: String): List<People>?
}
```

写PeopleService

```
package com.easy.kotlin.service

import com.easy.kotlin.entity.People
import com.easy.kotlin.repository.PeopleRepository
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service

/**
 * Created by jack on 2017/6/7.
 */
@Service
class PeopleService : PeopleRepository {

    @Autowired
    val peopleRepository: PeopleRepository? = null


    override fun findByLastName(lastName: String): List<People>? {
        return peopleRepository?.findByLastName(lastName)
    }

    override fun <S : People?> save(entity: S): S? {
        return peopleRepository?.save(entity)
    }

    override fun <S : People?> save(entities: MutableIterable<S>?): MutableIterable<S>?
 {
        return peopleRepository?.save(entities)
    }

    override fun delete(entities: MutableIterable<People>?) {
```

```
    }

    override fun delete(entity: People?) {
    }

    override fun delete(id: Long?) {
    }

    override fun findAll(ids: MutableIterable<Long>?): MutableIterable<People>? {
        return peopleRepository?.findAll(ids)
    }

    override fun findAll(): MutableIterable<People>? {
        return peopleRepository?.findAll()
    }

    override fun exists(id: Long?): Boolean {
        return peopleRepository?.exists(id)!!
    }

    override fun count(): Long {
        return peopleRepository?.count()!!
    }

    override fun findOne(id: Long?): People? {
        return peopleRepository?.findOne(id)
    }

    override fun deleteAll() {
    }
}
```

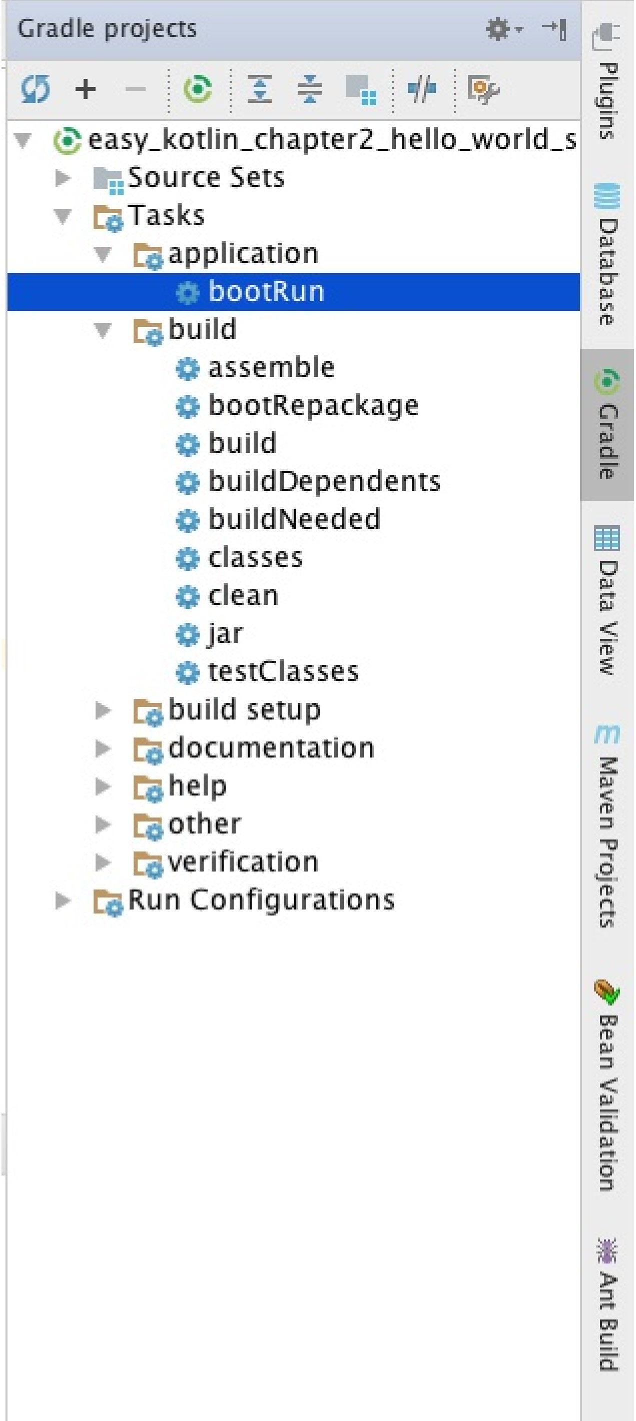
写PeopleController

```
package com.easy.kotlin.controller

import com.easy.kotlin.service.PeopleService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody

/**
 * Created by jack on 2017/6/7.
 */
@Controller
class PeopleController {
```

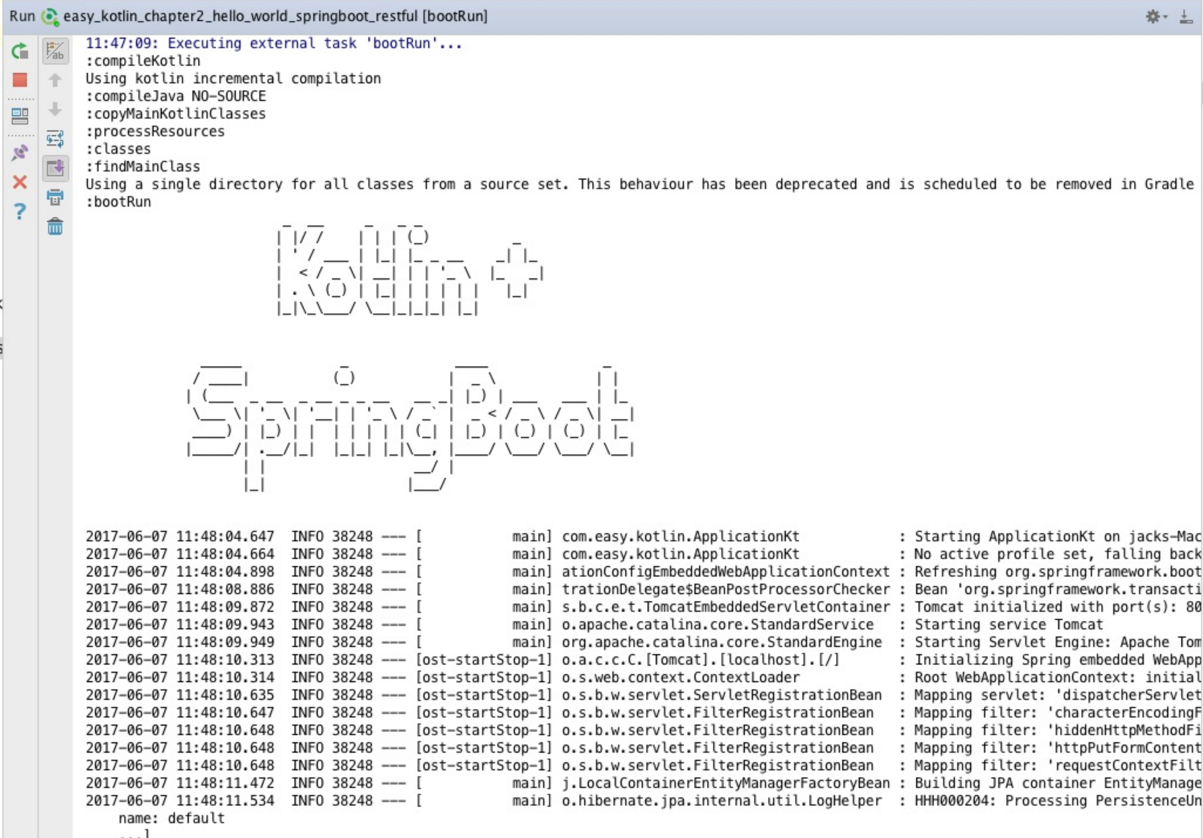
```
    @Autowired
    val peopleService: PeopleService? = null

    @GetMapping(value = "/hello")
    @ResponseBody
    fun hello(@RequestParam(value = "lastName") lastName: String): Any {
        val peoples = peopleService?.findByLastName(lastName)
        val map = HashMap<Any, Any>()
        map.put("hello", peoples!!)
        return map
    }

}
```

1. 运行测试

点击Gradle的 `bootRun` ，如下图



如果没有异常，启动成功，我们将看到以下输出：



打开浏览器，访问请求：

http://127.0.0.1:8000/

输出响应：

```
Hello,World!
```

访问

http://127.0.0.1:8000/hello?lastName=chen

```
// 20170607115700
// http://127.0.0.1:8000/hello?lastName=chen

{
  "hello": [
    {
      "id": 1,
      "firstName": "Jason",
      "lastName": "Chen",
      "gender": "Male",
      "age": 28,
      "gmtCreated": 1496768497000,
      "gmtModified": 1496768497000
```

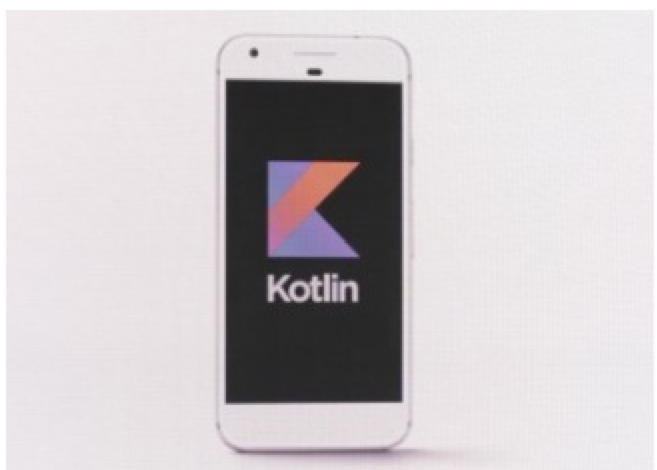
```
    },
    {
      "id": 3,
      "firstName": "Corey",
      "lastName": "Chen",
      "gender": "Female",
      "age": 20,
      "gmtCreated": 1496768497000,
      "gmtModified": 1496768497000
    }
    ...
  ]
}
```

本节示例工程源代码：

https://github.com/EasyKotlin/easy_kotlin_chapter2_hello_world_springboot_restful

2.4 Android版的HelloWorld

2017谷歌I/O大会：宣布 Kotlin 成 Android 开发一级语言。



2017谷歌I/O大会上，谷歌宣布，将Kotlin语言作为安卓开发的一级编程语言。Kotlin由JetBrains公司开发，与Java100%互通，并具备诸多Java尚不支持的新特性。谷歌称还将与JetBrains公司合作，为Kotlin设立一个非盈利基金会。

JetBrains在2010年首次推出Kotlin编程语言，并在次年将之开源。下一版的AndroidStudio（3.0）也将提供支持。

下面我们简要介绍如何在Android上开始一个Kotlin的HelloWorld程序。

对于我们程序员来说，我们正处于一个美好的时代。得益于互联网的发展、工具的进步，我们现在学习一门新技术的成本和难度都比过去低了很多。

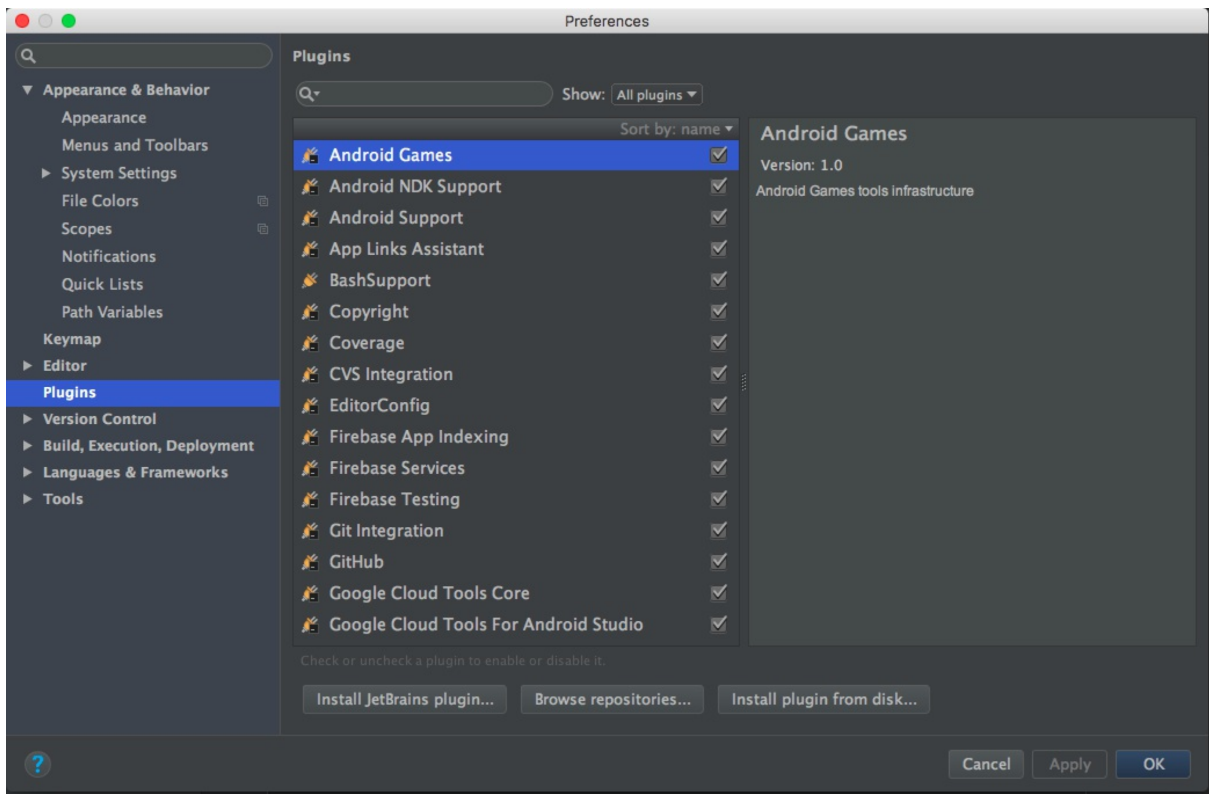
假设你之前没有使用过Kotlin，那么从头开始写一个HelloWorld的app也只需要这么几步：

1. 首先，你要有一个Android Studio。 本节中，我们用的是2.2.3版本，其它版本应该也大同小异。

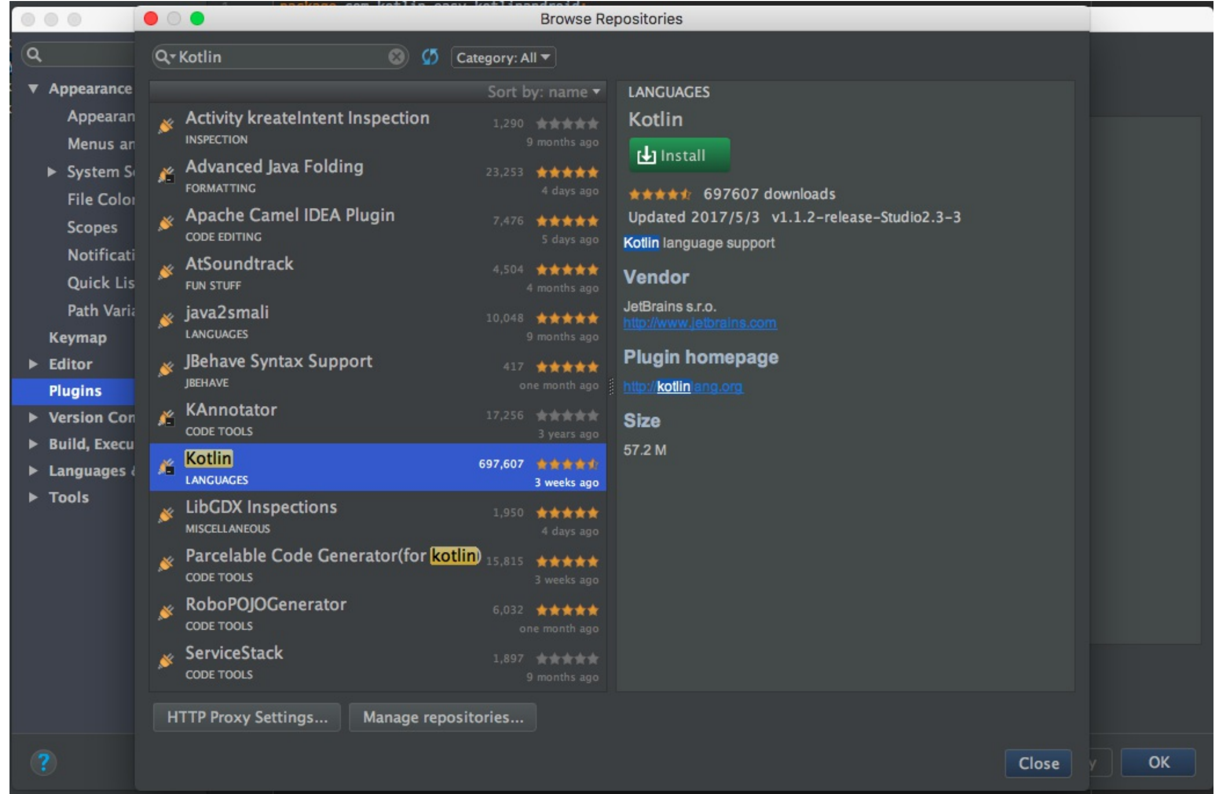
```
Android Studio 2.3.1
Build #AI-162.3871768, built on April 1, 2017
JRE: 1.8.0_112-release-b06 x86_64
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o
```

1. 其次，安装一个Kotlin的插件。

依次打开：Android Studio > Preferences > Plugins,



然后选择『Browse repositories』，在搜索框中搜索Kotlin，结果列表中的『Kotlin』插件，如下图



点击安装，安装完成之后，重启Android Studio。

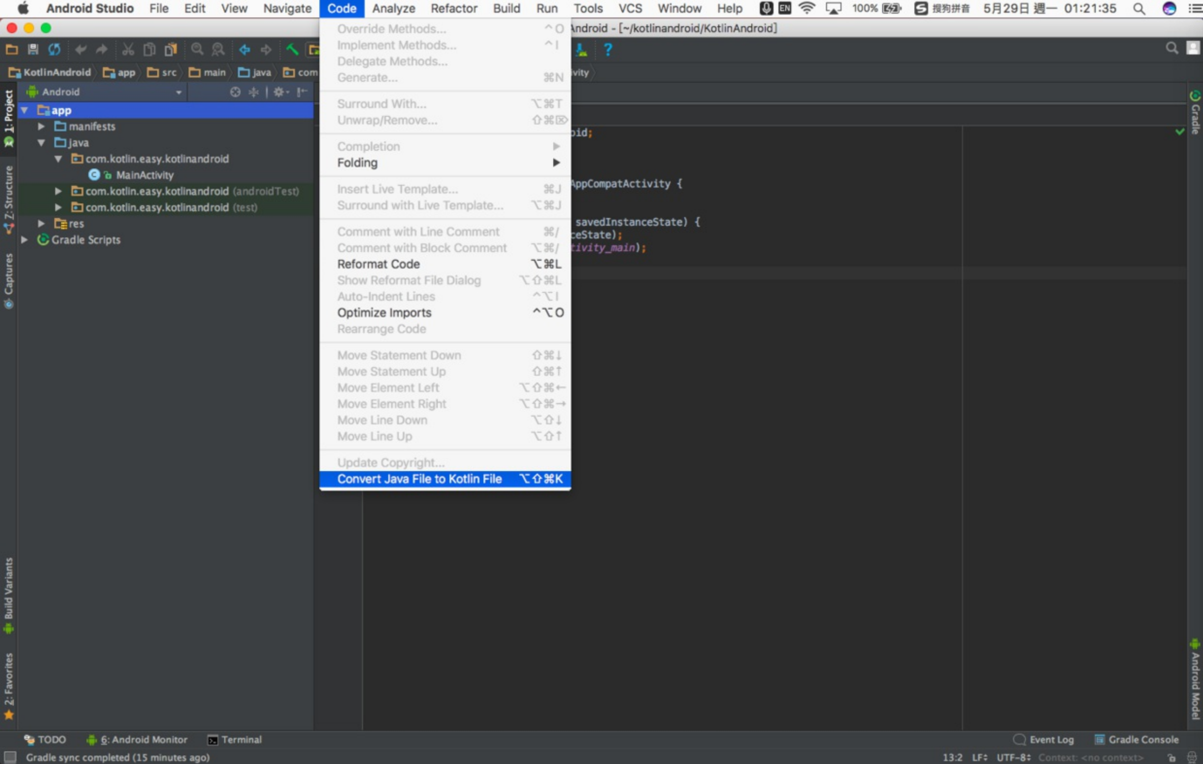
1. 新建一个Android项目

重新打开Android Studio，新建一个Android项目吧，添加一个默认的MainActivity——像以前一样即可。

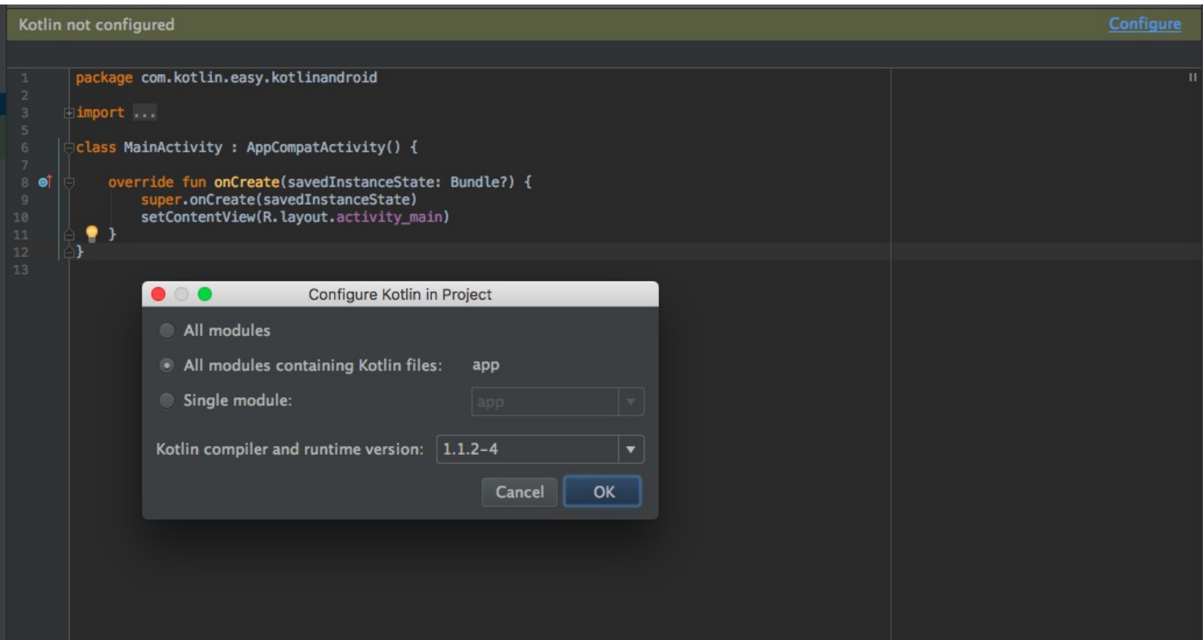
1. 转换Java to Kotlin

安装完插件的AndroidStudio现在已经拥有开发Kotlin的功能。我们先来尝试它的转换功能：Java -> Kotlin，可以把现有的java文件翻译成Kotlin文件。

打开MainActivity文件，在Code菜单下面可以看到一个新的功能：Convert Java File to Kotlin File。



点击转换，



可以看到转换后的Kotlin文件：MainActivity.kt

```
package com.kotlin.easy.kotlinandroid

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

这个转换功能，对我们Java程序员在学习Kotlin是十分实用。我们可以基于我们之前的Java编码的经验来迅速学习Kotlin编程。

1. 配置gradle文件

MainActivity已经被转换成了Kotlin实现，但是项目目前gradle编译、构建、运行还不能执行，还需要进一步配置一下，让项目支持grade的编译、运行。当然，这一步也不需要我们做太多工作——IDEA都已经帮我们做好了。

在Java代码转换成Kotlin代码之后，打开MainActivity.kt文件，编译器会提示"Kotlin not configured"，点击一下Configure按钮，IDEA就会自动帮我们把配置文件写好了。

我们可以看出，主要的依赖项是：

```
kotlin-gradle-plugin
plugin: 'kotlin-android'
kotlin-stdlib-jre7
```

完整的配置文件如下：

Project build.gradle

```
// Top-level build file where you can add configuration options common to all sub-proje
cts/modules.

buildscript {
    ext.kotlin_version = '1.1.2-4'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
```

```
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Module build.gradle

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.kotlin.easy.kotlinandroid"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rul
es.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
}
repositories {
    mavenCentral()
}
```

所以说使用IDEA来写Kotlin代码，这工具的完美集成会让你用起来如丝般润滑。毕竟Kotlin的亲爸爸JetBrains是专门做工具的，而且Intelli IDEA又是那么敏捷、智能。

配置之后，等Gradle Sync完成，即可运行。

6.运行

运行结果如下

LTE
2:40
KotlinAndroid
Hello World!

工程源码：https://github.com/EasyKotlin/KotlinAndroid

2.5 JavaScript版HelloWorld

在Kotlin 1.1中，开始支持JavaScript和协程是引人注目的亮点。本节我们简单介绍Kotlin代码编译转化为JavaScript的方法。

为了极简直观地感受这个过程，我们先在命令行REPL环境体验一下Kotlin源码被编译生成对应的JavaScript代码的过程。

首先，使用编辑器新建一个HelloWord.kt

```
fun helloWorld(){
    println("Hello,World!")
}
```

命令行使用 `kotlinc-js` 编译

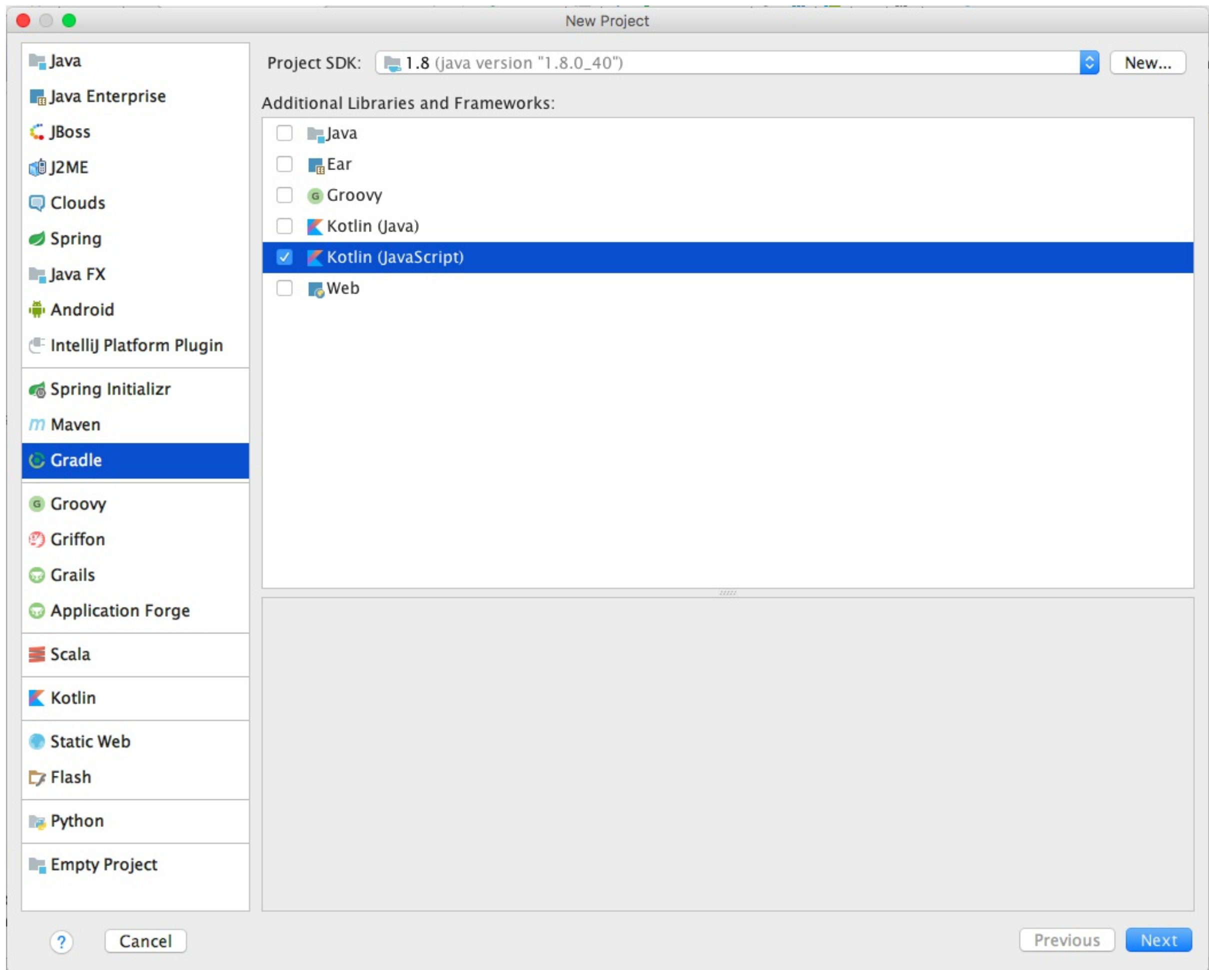
```
kotlinc-js -output HelloWorld.js HelloWorld.kt
```

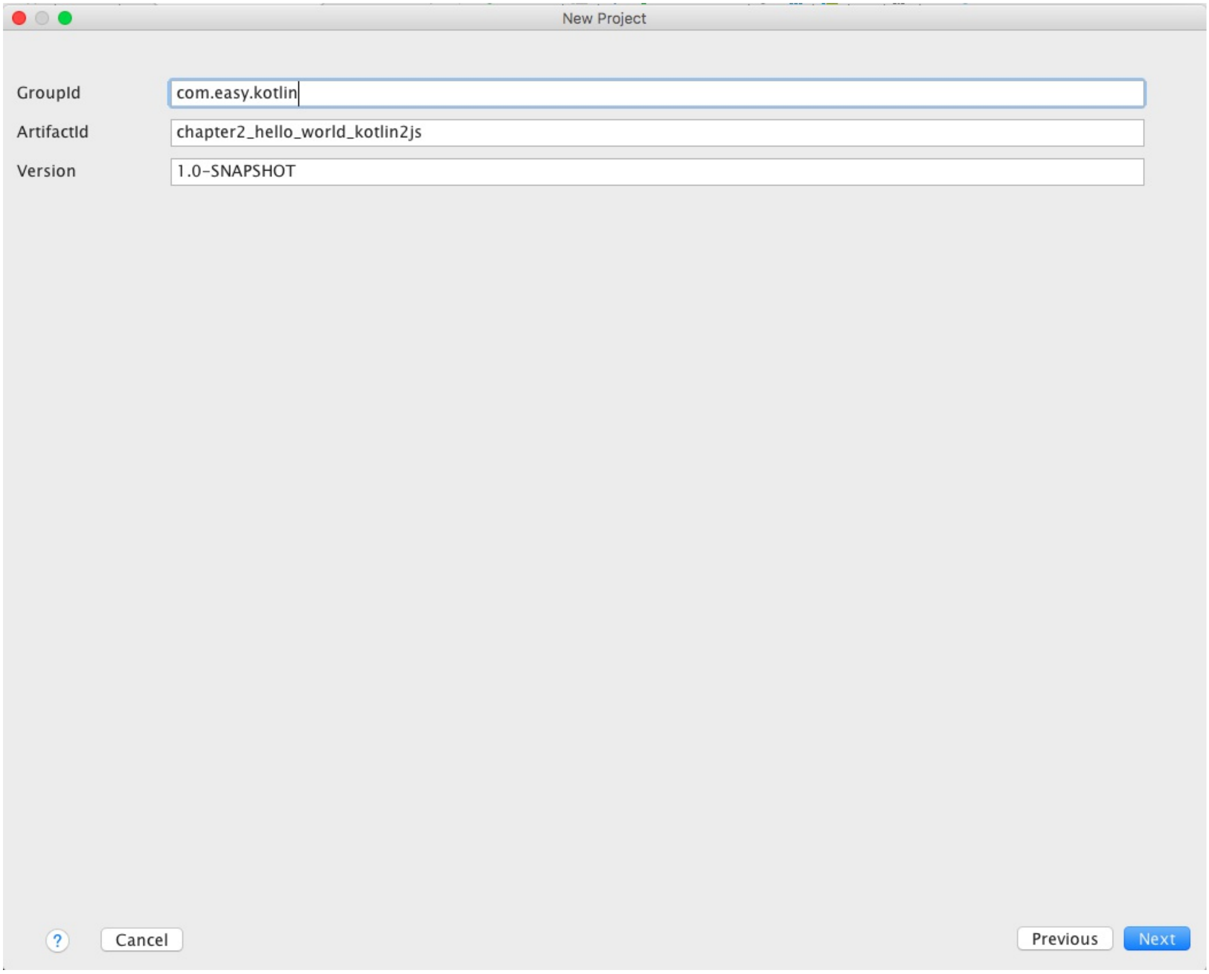
运行完毕，我们会在当前目录下看到 `HelloWorld.js` , 其内容如下

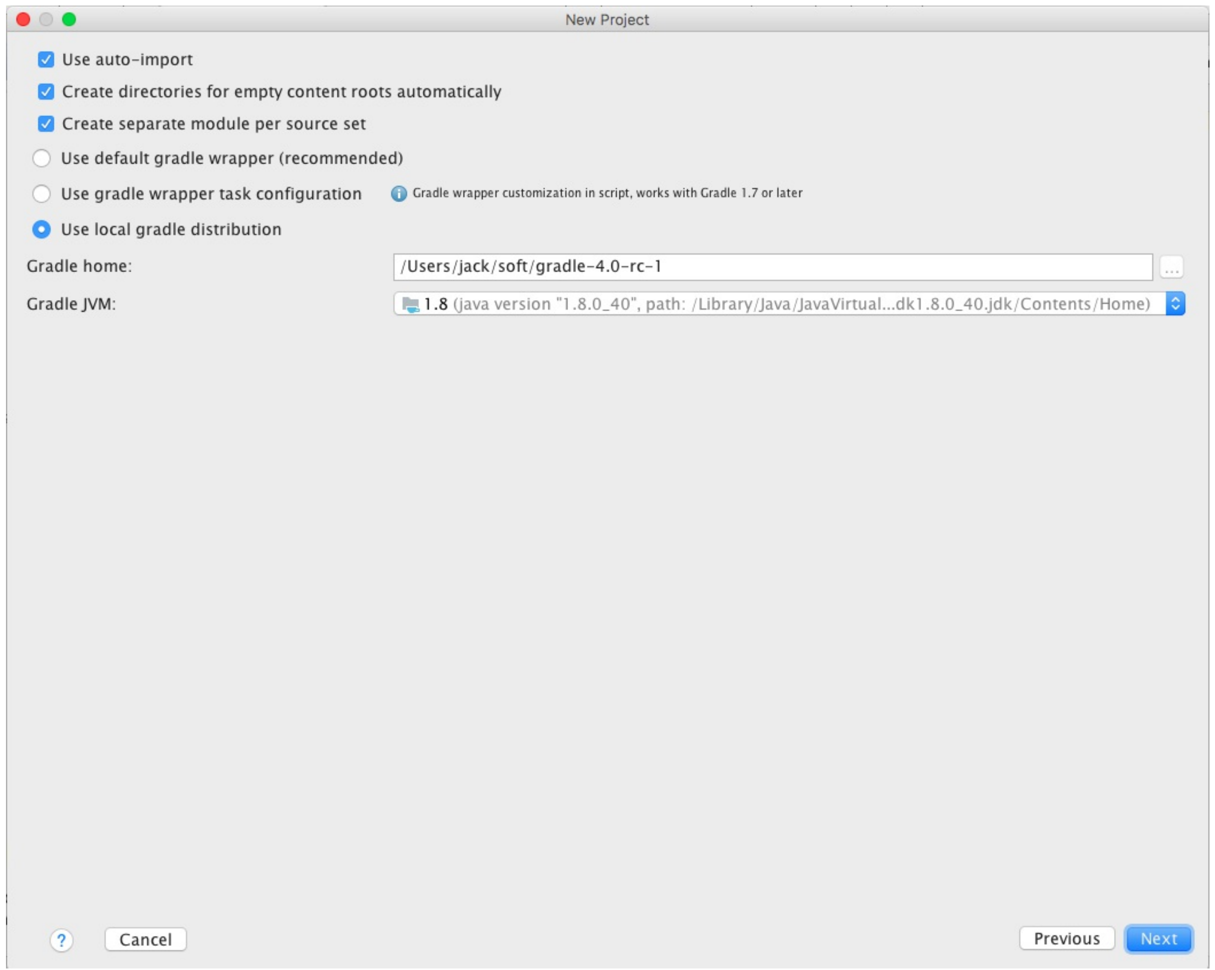
```
if (typeof kotlin === 'undefined') {
  throw new Error("Error loading module 'HelloWorld'. Its dependency 'kotlin' was not found. Please, check whether 'kotlin' is loaded prior to 'HelloWorld'.");
}
var HelloWorld = function (_, Kotlin) {
  'use strict';
  var println = Kotlin.kotlin.io.println_s8jyv4$;
  function helloWorld() {
    println('Hello,World!');
  }
  _.helloWorld = helloWorld;
  Kotlin.defineModule('HelloWorld', _);
  return _;
}(typeof HelloWorld === 'undefined' ? {} : HelloWorld, kotlin);
```

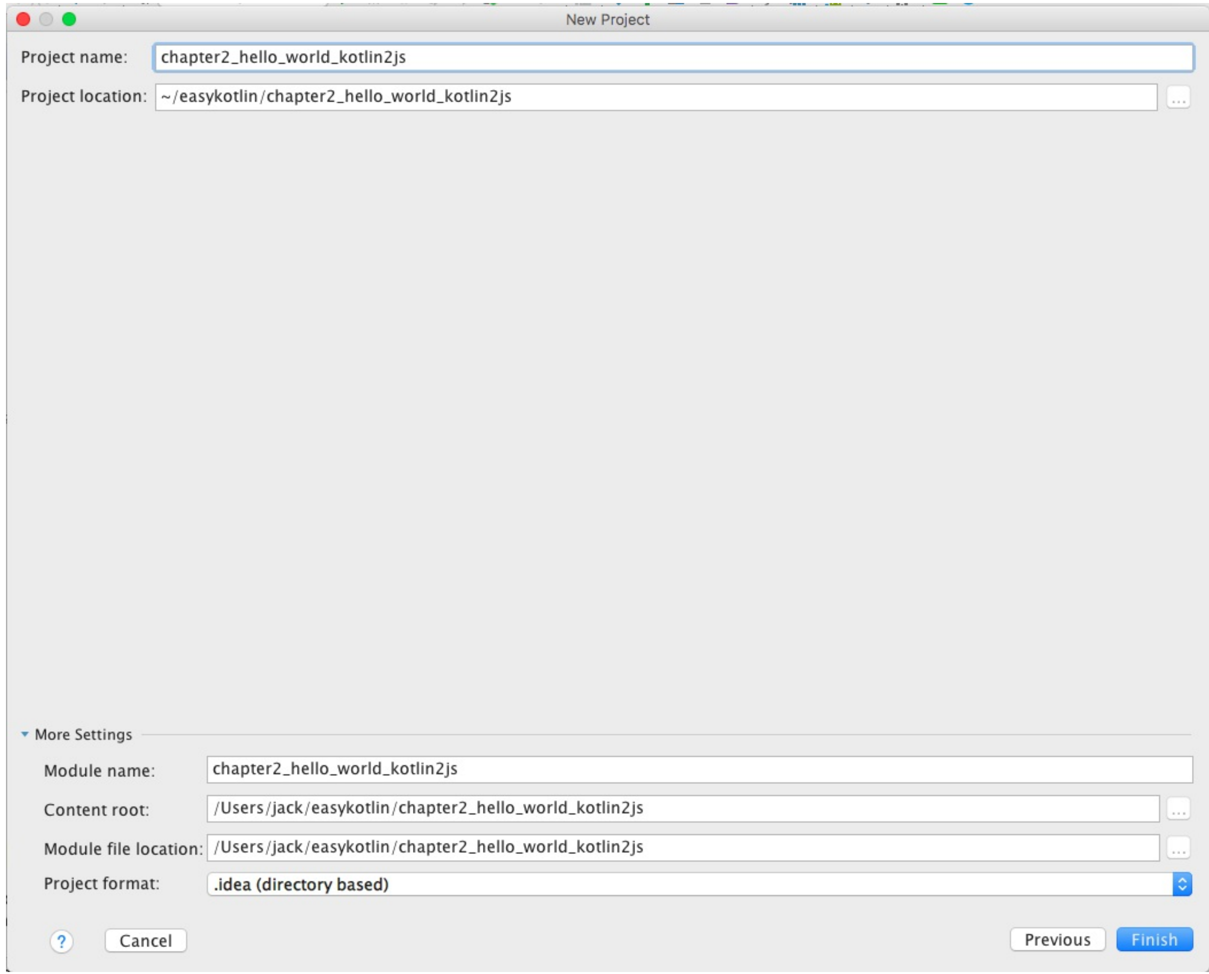
我们看到，使用 `kotlinc-js` 转换成的js代码依赖'kotlin'模块。这个模块是Kotlin支持JavaScript脚本的内部封装模块。也就是说，如果我们想要使用 `HelloWorld.js` ，先要引用 `kotlin.js` 。这个 `kotlin.js` 在kotlin-stdlib-js-1.1.2.jar里面。 下面我们使用IDEA新建一个Kotlin（JavaScript）工程。在这个过程中，我们将会看到使用Kotlin来开发js的过程。

首先按照以下步骤新建工程


New Project
Java
Java Enterprise
JBoss
J2ME
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Griffon
Grails
Application Forge
Scala
Kotlin
Static Web
Flash
Python
Empty Project
Project SDK:
1.8 (java version "1.8.0_40")
New...
Additional Libraries and Frameworks:
Java
Ear
Groovy
Kotlin (Java)
Kotlin (JavaScript)
Web
Cancel
Previous
Next


New Project
GroupId
com.easy.kotlin
ArtifactId
chapter2_hello_world_kotlin2js
Version
1.0-SNAPSHOT
?
Cancel
Previous
Next


New Project
Use auto-import
Create directories for empty content roots automatically
Create separate module per source set
Use default gradle wrapper (recommended)
Use gradle wrapper task configuration
Gradle wrapper customization in script, works with Gradle 1.7 or later
Use local gradle distribution
Gradle home:
/Users/jack/soft/gradle-4.0-rc-1
Gradle JVM:
1.8 (java version "1.8.0_40", path: /Library/Java/JavaVirtual...dk1.8.0_40.jdk/Contents/Home)
Cancel
Previous
Next



等待Gradle初始化工程完毕，我们将得到一个Gradle KotlinJS 工程，其目录如下

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
```

其中，build.gradle配置文件为

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'kotlin2js'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
}
```

其中，apply plugin: 'kotlin2js' 是Gradle的kotlin编译成js的插件。org.jetbrains.kotlin:kotlin-stdlib-js 是KotlinJS的运行库。

另外，我们需要再配置一下Kotlin代码编译成JS的编译规则，以及文件放置目录等属性，如下所示

```
build.doLast {
    configurations.compile.each { File file ->
        copy {
            includeEmptyDirs = false

            from zipTree(file.absolutePath)
            into "${projectDir}/web"
            include { fileTreeElement ->
                def path = fileTreeElement.path
                path.endsWith(".js") && (path.startsWith("META-INF/resources/") || !pat
h.startsWith("META-INF/"))
            }
        }
    }
}

compileKotlin2Js {
    kotlinOptions.outputFile = "${projectDir}/web/js/app.js"
```

```
    kotlinOptions.moduleKind = "plain" // plain (default),AMD,commonjs,umd
    kotlinOptions.sourceMap = true
    kotlinOptions.verbose = true
    kotlinOptions.suppressWarnings = true
    kotlinOptions.metaInfo = true
}
```

其中， `kotlinOptions.moduleKind` 配置项是Kotlin代码编译成JavaScript代码的类型。 支持普通JS（plain），AMD(Asynchronous Module Definition，异步模块定义)、CommonJS和UMD(Universal Model Definition，通用模型定义)。

AMD通常在浏览器的客户端使用。AMD是异步加载模块，可用性和性能相对会好。

CommonJS是服务器端上使用的模块系统，通常用于nodejs。

UMD是想综合AMD、CommonJS这两种模型，同时支持它们在客户端或服务器端上使用。

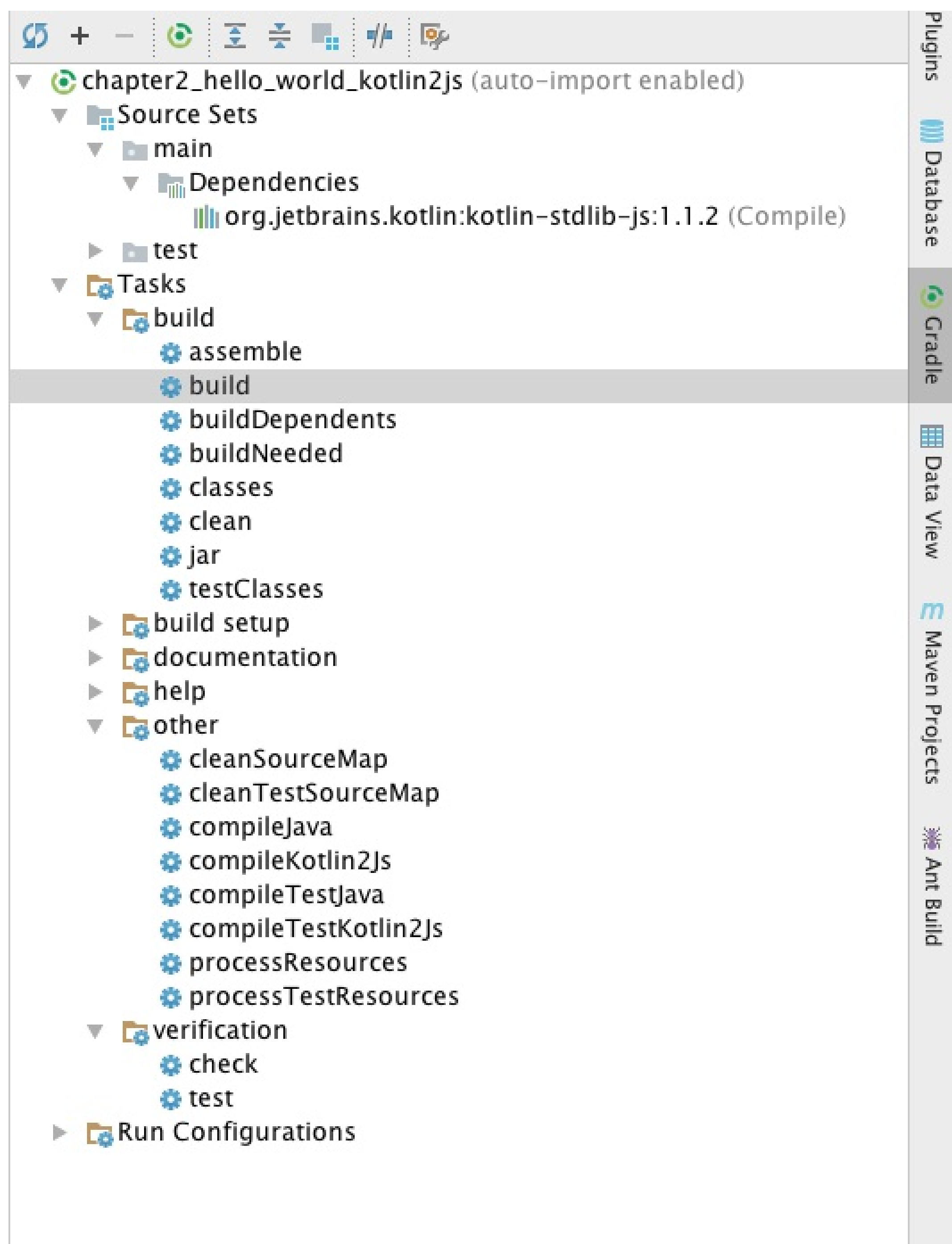
我们这里为了极简化演示，直接采用了普通JS `plain` 类型。

除了输出的 JavaScript 文件，该插件默认会创建一个带二进制描述符的额外 JS 文件。 如果你是构建其他 Kotlin 模块可以依赖的可重用库，那么该文件是必需的，并且应该与转换结果一起分发。 其生成由 kotlinOptions.metaInfo 选项控制。

一切配置完毕，我们来写Kotlin代码App.kt

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/7.
 */

fun helloWorld() {
    println("Hello,World!")
}
```

然后，我们直接使用Gradle构建工程，如下图所示



控制台输出

```
23:47:05: Executing external task 'build'...
Using a single directory for all classes from a source set. This behaviour has been dep
recated and is scheduled to be removed in Gradle 5.0
    at build_3e0ikl0qk0r006tvk0olcp2lu.run(/Users/jack/easykotlin/chapter2_hello_world_
kotlin2js/build.gradle:15)
:compileJava NO-SOURCE
:compileKotlin2Js
:processResources NO-SOURCE
:classes
```

```
:jar
:assemble
:compileTestJava NO-SOURCE
:compileTestKotlin2Js NO-SOURCE
:processTestResources NO-SOURCE
:testClasses UP-TO-DATE
:test NO-SOURCE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL in 2s
3 actionable tasks: 3 executed
23:47:08: External task execution finished 'build'.
```

此时，我们可以看到工程目录变为

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
jack@jacks-MacBook-Air:~/easykotlin/chapter2_hello_world_kotlin2js$ tree ..
├── build
│   ├── kotlin
│   │   └── sessions
│   ├── kotlin-build
│   │   └── caches
│   │       └── version.txt
│   ├── libs
│   │   └── chapter2_hello_world_kotlin2js-1.0-SNAPSHOT.jar
│   └── tmp
│       └── jar
│           └── MANIFEST.MF
├── build.gradle
├── settings.gradle
```

```
├── src
│   ├── main
│   │   ├── java
│   │   ├── kotlin
│   │   │   └── com
│   │   │       └── easy
│   │   │           └── kotlin
│   │   │               └── App.kt
│   │   └── resources
│   └── test
│       ├── java
│       ├── kotlin
│       └── resources
└── web
    ├── js
    │   ├── app
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── kotlin.kjsm
    │   ├── app.js
    │   ├── app.js.map
    │   └── app.meta.js
    ├── kotlin.js
    └── kotlin.meta.js

26 directories, 14 files
```

这个web目录就是Kotlin代码通过kotlin-stdlib-js-1.1.2.jar编译的输出结果。

其中，app.js代码如下

```
if (typeof kotlin === 'undefined') {
  throw new Error("Error loading module 'app'. Its dependency 'kotlin' was not found. P
lease, check whether 'kotlin' is loaded prior to 'app'.");
}
var app = function (_, Kotlin) {
  'use strict';
  var println = Kotlin.kotlin.io.println_s8jyv4$;
  function helloWorld() {
    println('Hello,World!');
  }
  var package$com = _.com || (_.com = {});
  var package$easy = package$com.easy || (package$com.easy = {});
  var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
  package$kotlin.helloWorld = helloWorld;
  Kotlin.defineModule('app', _);
  return _;
```

```
}(typeof app === 'undefined' ? {} : app, kotlin);

//@ sourceMappingURL=app.js.map
```

里面这段自动生成的代码显得有点绕

```
var package$com = _.com || (_.com = {});
  var package$easy = package$com.easy || (package$com.easy = {});
  var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
  package$kotlin.helloWorld = helloWorld;

  Kotlin.defineModule('app', _);
  return _;
```

简化之后的意思表达如下

```

_.com.easy.kotlin.helloWorld = helloWorld;
```

目的是建立Kotlin代码跟JavaScript代码的映射关系。这样我们在前端代码中调用

```
  function helloWorld() {
    println('Hello,World!');
  }
```

这个函数时，只要这样调用即可

```
app.com.easy.kotlin.helloWorld()
```

下面我们来新建一个index.html页面，使用我们生成的app.js。代码如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>KotlinJS</title>

</head>
<body>


<!-- 优先加载kotlin.js，再加载应用程序代码app.js-->
<script type="text/javascript" src="kotlin.js"></script>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="js/app.js"></script>
```

```
<script>
    var kotlinJS = app;
    console.log(kotlinJS.com.easy.kotlin.helloWorld())
</script>
</body>
</html>
```

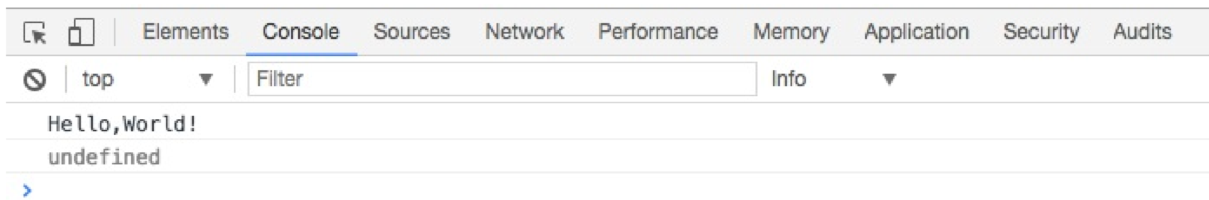
我们需要优先加载kotlin.js，再加载应用程序代码app.js。 当然，我们仍然可以像以前一样使用诸如jquery.js这样的库。

在浏览器中打开index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>KotlinJS</title>
6
7  </head>
8  <body>
9
10
11 <!-- 优先加载kotlin.js，再加载应用程序代码app.js-->
12 <script type="text/javascript" src="kotlin.js"></script>
13 <script type="text/javascript" src="jquery.js"></script>
14 <script type="text/javascript" src="js/app.js"></script>
15
16 <script>
17     var kotlinJS = app;
18     console.log(kotlinJS.com.easy.kotlin.helloWorld())
19 </script>
20 </body>
21 </html>
22
```

Chrome (⌥F2, hold Shift to open URL of lo

我们可以看到浏览器控制台输出



这个helloWorld() JavaScript函数

```
var println = Kotlin.kotlin.io.println_s8jyv4$;
function helloWorld() {
  println('Hello,World!');
}
```

对应kotlin.js代码中的3755行处的代码：

```
BufferedOutputToConsoleLog.prototype.flush = function() {
    console.log(this.buffer);
    this.buffer = "";
  };
```

本节工程源代码：https://github.com/EasyKotlin/chapter2_hello_world_kotlin2js

参考资料

1.https://kotlinlang.org/docs/reference/compiler-plugins.html

2.http://kotlinlang.org/docs/tutorials/javascript/working-with-modules/working-with-modules.html

第3章 Kotlin语言基础

> 掌握基础，持续练习

学习任何东西，都是一个由表及里的过程。学习一门编程语言也一样。对于一门编程语言来说，“表”就是基本词汇（关键字、标识符等）、句子（表达式）和语法。

每一门编程语言的学习内容都会涉及：运行环境、基础数据类型（数字、字符串、数组、集合、映射字典等）、表达式、流程控制、类、方法（函数）等等，不同的语言会借鉴其他的语言特性，同时也会有各自的特性。这样我们就可以通过对比学习来加深理解。另外，我们还通过大量实践深入理解，达到熟练使用。

所谓“纸上得来终觉浅，绝知此事要躬行”是也。下面让我们开始吧。

3.1 包（package）

我们先来举个例子。比如说，程序员A写了一个类叫 JSON , 程序员B也写了一个类叫 JSON。然后，我们在写代码的时候，想要同时使用这两个类，该怎么区分呢？

一个答案是使用目录命名空间。对应在Java中，就是使用 `package` 来组织类，以确保类名的唯一性。上面说的例子，A写的类放到 `package com.abc.fastjson` 中, B写的类就放到 `package com.bbc.jackjson` 中。这样我们在代码中，就可以根据命名空间来分别使用这两个类。调用示例如下

```
com.abc.fastjson.JSON.toJSONString()
com.bbc.jackjson.JSON.parseJSONObject()
```

在Kotlin中也沿袭了Java的 `package` 这个概念，同时做了一些扩展。

我们可以在 `*.kt` 文件开头声明 `package` 命名空间。例如在PackageDemo.kt源代码中，我们按照如下方式声明包

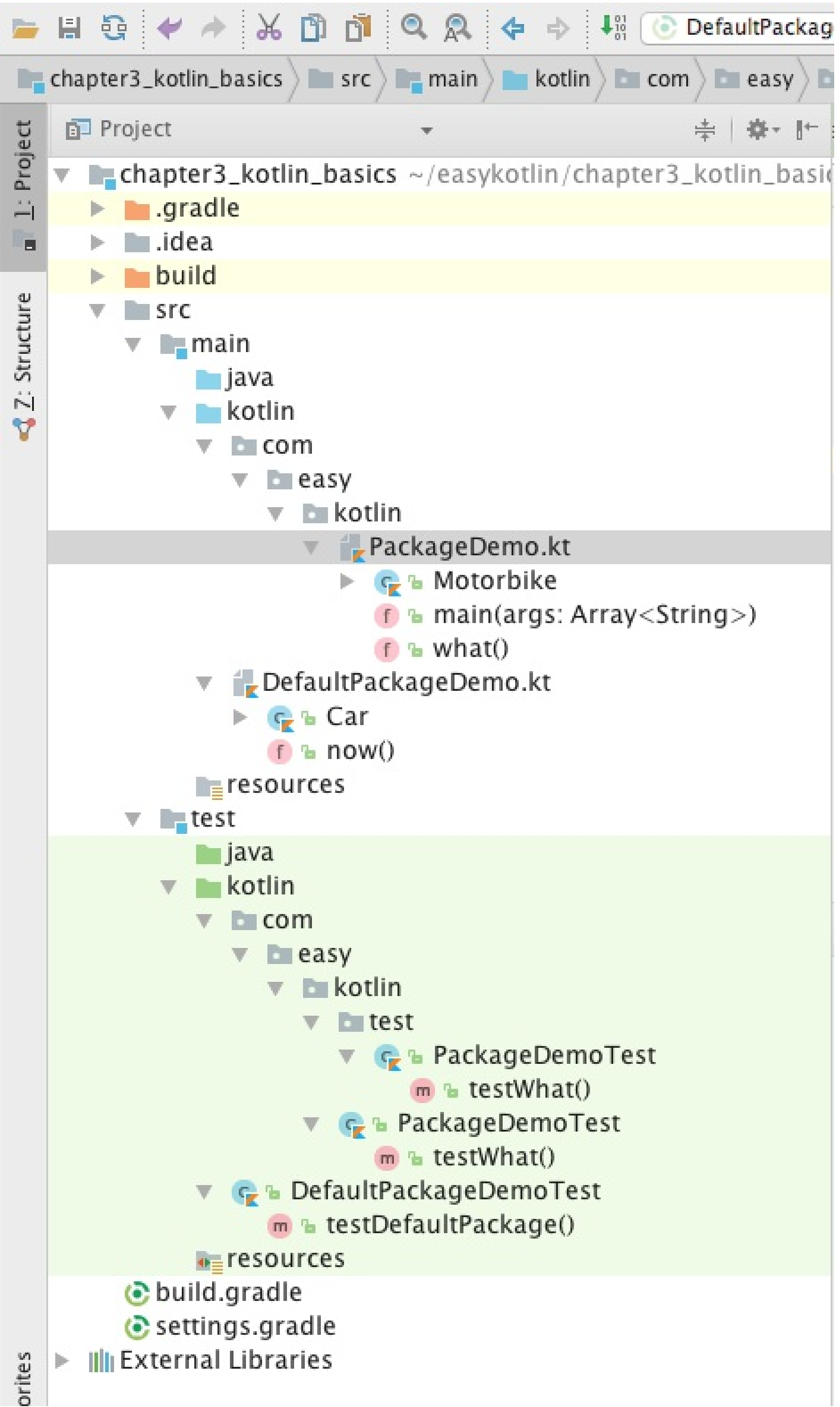
```
package com.easy.kotlin

fun what(){
    println("This is WHAT ?")
}

class Motorbike{
    fun drive(){
        println("Drive The Motorbike ...")
    }
}
```

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

包的声明处于源文件顶部。这里，我们声明了包 `com.easy.kotlin` ，里面定义了包级函数 `what()` ，同时定义了一个类 `Motorbike` 。另外，目录与包的结构无需匹配：源代码可以在文件系统的任意位置。

我们怎么使用这些类和函数呢？我们写一个Junit 测试类来示例说明。

首先，我们使用标准Gradle工程目录，对应的测试代码放在test目录下。具体目录结构如下



我们在测试源代码目录 `src/test/kotlin` 下面新建一个包，跟 `src/main/kotlin` 在同一个 `package com.easy.kotlin` 。然后，在此包下面新建一个测试类PackageDemoTest

```
package com.easy.kotlin

import org.junit.Test
```

```kotlin
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike(){
        val motorbike = Motorbike()
        motorbike.drive()
    }

}
```

其中， `what()` 函数跟 `PackageDemoTest` 类在同一个包命名空间下，可以直接调用，不需要 `import` 。 `Motorbike` 类跟 `PackageDemoTest` 类也是同理分析。

如果不在同一个package下面，我们就需要import对应的类和函数。例如，我们在 `src/test/kotlin` 目录下新建一个 `package com.easy.kotlin.test` , 使用 `package com.easy.kotlin` 下面的类和函数，示例如下

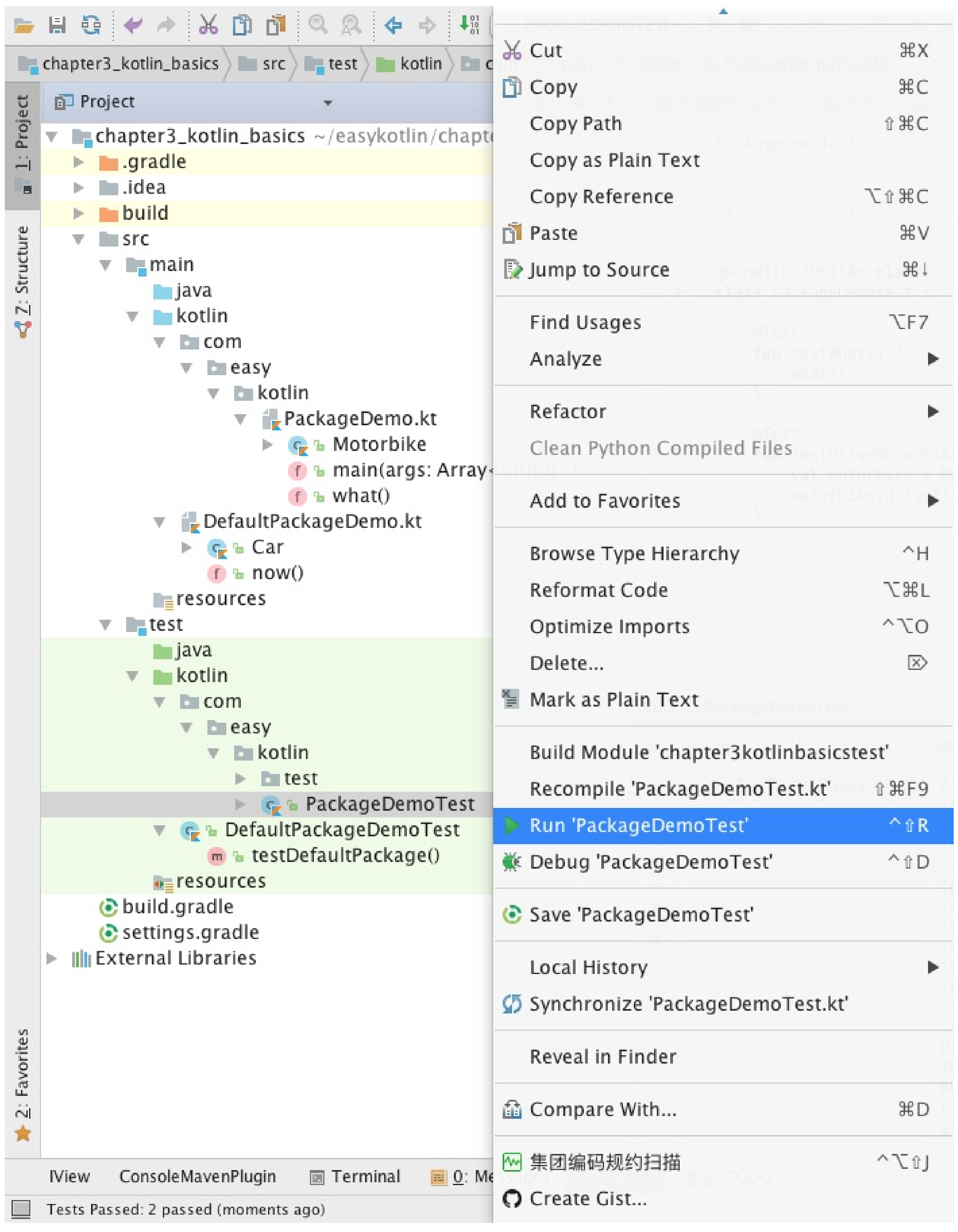
```kotlin
package com.easy.kotlin.test

import com.easy.kotlin.Motorbike
import com.easy.kotlin.what
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike() {
        val motorbike = Motorbike()
        motorbike.drive()
    }
}
```

我们使用 `import com.easy.kotlin.Motorbike` 导入类，直接使用 `import com.easy.kotlin.what` 导入包级函数。

上面我们使用JUnit4测试框架。在 `build.gradle` 中的依赖是

```
testCompile group: 'junit', name: 'junit', version: '4.12'
```

右击测试类，点击执行



运行结果

另外，如果我们不定义package命令空间，则默认在根级目录。例如直接在 `src/main/kotlin` 源代码目录下面新建 DefaultPackageDemo.kt 类

```
import java.util.*

fun now() {
    println("Now Date is: " + Date())
}

class Car{
    fun drive(){
        println("Drive The Car ... ")
    }
}
```

如果，我们同样在 `src/test/kotlin` 目录下面新建测试类DefaultPackageDemoTest

```
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4
```

```
@RunWith(JUnit4::class)
class DefaultPackageDemoTest {

    @Test
    fun testDefaultPackage() {
        now()
        val car = Car()
        car.drive()
    }

}
```

我们不需要import `now()` 函数和 `Car` 类，可以直接调用。如果我们在 `src/test/kotlin/com/easy/kotlin/PackageDemoTest.kt` 测试类里面调用 `now()` 函数和 `Car` 类，我们按照下面的方式import

```
import now
import Car
```

PackageDemoTest.kt完整测试代码如下

```
package com.easy.kotlin

import now
import Car
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike(){
        val motorbike = Motorbike()
        motorbike.drive()
    }

    @Test
    fun testDefaultPackage() {
        now()
```

```
        val car = Car()
        car.drive()
    }

}
```

另外， Kotlin会会默认导入一些基础包到每个 Kotlin 文件中：

> kotlin.*
>
> kotlin.annotation.*
>
> kotlin.collections.*
>
> kotlin.comparisons.* （自 1.1 起）
>
> kotlin.io.*
>
> kotlin.ranges.*
>
> kotlin.sequences.*
>
> kotlin.text.*

根据目标平台还会导入额外的包：

JVM:

> java.lang.*
>
> kotlin.jvm.*

JS:

> kotlin.js.*

本小节示例工程源代码：https://github.com/EasyKotlin/chapter3_kotlin_basics/tree/package_demo

3.2 声明变量和值

首先，在Kotlin中， 一切都是对象。所以，所有变量也都是对象（也就是说，任何变量都是根据引用类型来使用的）。

Kotlin的变量分为 `var` (可变的) 和 `val` (不可变的)。

可以简单理解为：

> `var` 是可写的，在它生命周期中可以被多次赋值；
>
> 而 `val` 是只读的，仅能一次赋值，后面就不能被重新赋值。

代码示例

```kotlin
package com.easy.kotlin

import java.util.*

class VariableVSValue {
    fun declareVar() {
        var a = 1
        a = 2
        println(a)
        println(a::class)
        println(a::class.java)

        var x = 5 // 自动推断出 `Int` 类型
        x += 1

        println("x = $x")
    }

    fun declareVal() {
        val b = "a"
        //b  = "b" //编译器会报错： Val cannot be reassigned
        println(b)
        println(b::class)
        println(b::class.java)

        val c: Int = 1  // 立即赋值
        val d = 2   // 自动推断出 `Int` 类型
        val e: Int  // 如果没有初始值类型不能省略
        e = 3       // 明确赋值
        println("c = $c, d = $d, e = $e")
    }
}
```

我们知道，在Java中也分可变与不可变（final）。在Kotlin中，更简洁的、更常用的场景是：只要可能，尽量在Kotlin中首选使用 `val` 不变值。因为事实上在程序中大部分地方使用不可变的变量，可带来很多益处，如：可预测的行为和线程安全。

3.3 变量类型推断

3.3.1 省去变量类型

在Kotlin中大部分情况你不需要说明你使用对象的类型，编译器可以直接推断出它的类型。代码示例

```kotlin
    fun typeInference(){
```

```
        val str = "abc"
        println(str)
        println(str is String)
        println(str::class)
        println(str::class.java)

//        abc
//        true
//        class java.lang.String (Kotlin reflection is not available)
//        class java.lang.String

        val d = Date()
        println(d)
        println(d is Date)
        println(d::class)
        println(d::class.java)

//        Fri Jun 09 00:06:33 CST 2017
//        true
//        class java.util.Date (Kotlin reflection is not available)
//        class java.util.Date

        val bool = true
        println(bool)
        println(bool::class)
        println(bool::class.java)

//        true
//        boolean (Kotlin reflection is not available)
//        boolean

        val array = arrayOf(1,2,3)
        println(array)
        println(array is Array)
        println(array::class)
        println(array::class.java)

//        [Ljava.lang.Integer;@7b5eadd8
//        true
//        class [Ljava.lang.Integer; (Kotlin reflection is not available)
//        class [Ljava.lang.Integer;
    }
```

所以，我们只需要依据要产生的变量类型填写var或val，其类型通常能够被推断出来。编译器能够检测到其类型，自动地完成类型转换。当然，我们也可以明确地指定变量类型。

但是，类型推断不是所有的。例如，整型变量Int不能赋值Long变量。下面的代码不能通过编译：

```
    fun Int2Long(){
        val x:Int = 10
        val y:Long = x // Type mismatch
    }
```

我们需要显式地调用对应的类型转换函数进行转换：

```
    fun Int2Long(){
        val x:Int = 10
//          val y:Long = x // Type mismatch
        val y: Long = x.toLong()
    }
```

3.3.2 使用 `is` 运算符进行类型检测

`is` 运算符检测一个表达式是否某类型的一个实例。

如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

```
fun getLength(obj: Any): Int? {
    var result = 0
    if (obj is String) {
        // `obj` 在该条件分支内自动转换成 `String`
        println(obj::class) //class java.lang.String
        result = obj.length
        println(result)
    }
    // 在离开类型检测分支后，`obj` 仍然是 `Any` 类型
    println(obj::class) // class java.lang.Object
    return result
}
```

测试类如下

```
@Test
fun testGetLength() {
    val obj = "abcdef"
    val len = variableVSValue.getLength(obj)
    Assert.assertTrue(len == 6)

    val obj2:Any = Any()
    variableVSValue.getLength(obj2)

}
```

3.4 字符串与其模板表达式

原始字符串(raw string)由三重引号（"""）分隔(这个跟python一样)。原始字符串可以包含换行符和任何其他字符。

```
package com.easy.kotlin

fun main(args: Array<String>) {
    val rawString = """
fun helloWorld(val name : String) {
   println("Hello, world!")
}
"""
    println(rawString)

}
```

字符串可以包含模板表达式。模板表达式以美元符号（$）开始。

```
val fooTemplateString = "$rawString has ${rawString.length} characters"
println(fooTemplateString)
```

3.5 流程控制语句

流程控制语句是编程语言中的核心之一。可分为：

> 分支语句(`if` 、 `when`)
>
> 循环语句(`for` 、 `while`)和
>
> 跳转语句 (`return` 、 `break` 、 `continue` 、 `throw`)等。

3.5.1 if表达式

if-else语句是控制程序流程的最基本的形式，其中else是可选的。

在 Kotlin 中，if 是一个表达式，即它会返回一个值(跟Scala一样)。

代码示例：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    println(max(1, 2))
}
```

```
fun max(a: Int, b: Int): Int {

// 作为表达式
    val max = if (a > b) a else b
    return max //  return if (a > b) a else b
}

fun max1(a: Int, b: Int): Int {
    // 传统用法
    var max1 = a
    if (a < b) max1 = b
    return max1

}

fun max2(a: Int, b: Int): Int {

// With else
    var max2: Int
    if (a > b) {
        max2 = a
    } else {
        max2 = b
    }
    return max2
}
```

另外，if 的分支可以是代码块，最后的表达式作为该块的值：

```
fun max3(a: Int, b: Int): Int {
    val max = if (a > b) {
        print("Max is a")
        a
    } else {
        print("Max is b")
        b
    }
    return max
}
```

if作为代码块时，最后一行为其返回值。

另外，在Kotlin中没有类似 `true? 1: 0` 这样的三元表达式。对应的写法是使用 `if else` 语句:

```
if(true) 1 else 0
```

如果 if 表达式只有一个分支， 或者分支的结果是 Unit , 它的值就是 Unit 。

示例代码

```
>>> val x = if(1==1) true
>>> x
kotlin.Unit
>>> val y = if(1==1) true else false
>>> y
true
```

if-else语句规则：

- if后的括号不能省略，括号里表达式的值须是布尔型

代码反例：

```
>>> if("a") 1
error: type mismatch: inferred type is String but Boolean was expected
if("a") 1
   ^

>>> if(1) println(1)
error: the integer literal does not conform to the expected type Boolean
if(1)
   ^
```

- 如果条件体内只有一条语句需要执行，那么if后面的大括号可以省略。良好的编程风格建议加上大括号。

```
>>> if(true) println(1) else println(0)
1
>>> if(true) { println(1)}  else{ println(0)}
1
```

- 对于给定的if，else语句是可选的，else if 语句也是可选的。
- else和else if同时出现时，else必须出现在else if 之后。
- 如果有多条else if语句同时出现，那么如果有一条else if语句的表达式测试成功，那么会忽略掉其他所有else if和else分支。
- 如果出现多个if,只有一个else的情形，else子句归属于最内层的if语句。

以上规则跟Java、C语言基本相同。

3.5.2 when表达式

when表达式类似于 switch-case 表达式。when会对所有的分支进行检查直到有一个条件满足。但相比switch而言，when语句要更加的强大，灵活。

Kotlin的极简语法表达风格，使得我们对分支检查的代码写起来更加简单直接：

```
fun cases(obj: Any) {
    when (obj) {
        1 -> print("第一项")
        "hello" -> print("这个是字符串hello")
        is Long -> print("这是一个Long类型数据")
        !is String -> print("这不是String类型的数据")
        else -> print("else类似于Java中的default")
    }
}
```

像 if 一样，when 的每一个分支也可以是一个代码块，它的值是块中最后的表达式的值。

如果其他分支都不满足条件会到 else 分支（类似default）。

如果我们有很多分支需要用相同的方式处理，则可以把多个分支条件放在一起，用逗号分隔：

```
fun switch(x: Any) {
    when (x) {
        -1, 0 -> print("x == -1 or x == 0")
        1 -> print("x == 1")
        2 -> print("x == 2")
        else -> { // 注意这个块
            print("x is neither 1 nor 2")
        }
    }
}
```

我们可以用任意表达式（而不只是常量）作为分支条件

```
fun switch(x: Int) {
    val s = "123"
    when (x) {
        -1, 0 -> print("x == -1 or x == 0")
        1 -> print("x == 1")
        2 -> print("x == 2")
        8 -> print("x is 8")
        parseInt(s) -> println("x is 123")
        else -> { // 注意这个块
            print("x is neither 1 nor 2")
        }
    }
}
```

我们也可以检测一个值在 in 或者不在 !in 一个区间或者集合中：

```
val x = 1
val validNumbers = arrayOf(1, 2, 3)
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

3.5.3 for循环

Kotlin的for循环跟现代的程序设计语言基本相同。

for 循环可以对任何提供迭代器（iterator）的对象进行遍历，语法如下:

```
for (item in collection) {
    print(item)
}
```

循环体可以是一个代码块。

```
for (i in intArray) {
    ...
}
```

代码示例

```
/**
 * For loop iterates through anything that provides an iterator.
 * See http://kotlinlang.org/docs/reference/control-flow.html#for-loops
 */
fun main(args: Array<String>) {
    for (arg in args)
        println(arg)
    // or
    println()
    for (i in args.indices)
        println(args[i])
}
```

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

```
for (i in array.indices) {
```

```
    print(array[i])
}
```

或者你可以用库函数 `withIndex` ：

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

3.5.4 while循环

while 和 do .. while使用方式跟C、Java语言基本一致。

代码示例

```
package com.easy.kotlin

fun main(args: Array<String>) {
    var x = 10
    while (x > 0) {
        x--
        println(x)
    }

    var y = 10
    do {
        y = y + 1
        println(y)
    } while (y < 20) // y的作用域包含此处
}
```

3.5.5 break 和 continue

`break` 和 `continue` 都是用来控制循环结构的，主要是用来停止循环（中断跳转）。

1.break

我们在写代码的时候，经常会遇到在某种条件出现的时候，就直接提前终止循环。而不是等到循环条件为 `false` 时才终止。这个时候，我们就可以使用 `break` 结束循环。 `break` 用于完全结束一个循环，直接跳出循环体，然后执行循环后面的语句。

问题场景：

> 打印数字1~10，只要遇到偶数，就结束打印。

代码示例：

```
fun breakDemo_1() {
    for (i in 1..10) {
        println(i)
        if (i % 2 == 0) {
            break
        }
    } // break to here
}
```

测试代码：

```
@Test
fun testBreakDemo_1(){
    breakDemo_1()
}
```

输出：

```
1
2
```

2.continue

`continue` 是只终止本轮循环，但是还会继续下一轮循环。可以简单理解为，直接在当前语句处中断，跳转到循环入口，执行下一轮循环。而 `break` 则是完全终止循环，跳转到循环出口。

问题场景：

> 打印数字0~10，但是不打印偶数。

代码示例：

```
fun continueDemo() {
    for (i in 1..10) {
        if (i % 2 == 0) {
            continue
        }
        println(i)
    }
}
```

测试代码

```
    @Test
```

```
    fun testContinueDemo() {
        continueDemo()
    }
```

输出

```
1
3
5
7
9
```

3.5.6 return返回

在Java、C语言中，return语句使我们再常见不过的了。虽然在Scala，Groovy这样的语言中，函数的返回值可以不需要显示用return来指定，但是我们仍然认为，使用return的编码风格更加容易阅读理解。

在Kotlin中，除了表达式的值，有返回值的函数都要求显式使用 `return` 来返回其值。

代码示例

```
fun sum(a: Int,b: Int): Int{
    return a+b
}

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

我们在Kotlin中，可以直接使用 `=` 符号来直接返回一个函数的值。

代码示例

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

>>> sum(1,10)
11

>>> max(1,2)
2

>>> val sum=fun(a:Int, b:Int) = a+b
>>> sum
(kotlin.Int, kotlin.Int) -> kotlin.Int
>>> sum(1,1)
2
```

```
>>> val sumf = fun(a:Int, b:Int) = {a+b}
>>> sumf
(kotlin.Int, kotlin.Int) -> () -> kotlin.Int
>>> sumf(1,1)
() -> kotlin.Int
>>> sumf(1,1).invoke()
2
```

上述代码示例中，我们可以看到，后面的函数体语句有没有大括号 `{}` 意思完全不同。加了大括号，意义就完全不一样了。我们再通过下面的代码示例清晰的看出：

```
>>> fun sumf(a:Int,b:Int) = {a+b}
>>> sumf(1,1)
() -> kotlin.Int
>>> sumf(1,1).invoke
error: function invocation 'invoke()' expected
sumf(1,1).invoke
          ^
>>> sumf(1,1).invoke()
2
>>> fun maxf(a:Int, b:Int) = {if(a>b) a else b}
>>> maxf(1,2)
() -> kotlin.Int
>>> maxf(1,2).invoke()
2
```

可以看出，`sumf`，`maxf` 的返回值是函数类型：

```
() -> kotlin.Int
() -> kotlin.Int
```

这点跟Scala是不同的。在Scala中，带不带大括号 `{}`，意思一样：

```
scala> def maxf(x:Int, y:Int) = { if(x>y) x else y }
maxf: (x: Int, y: Int)Int

scala> def maxv(x:Int, y:Int) = if(x>y) x else y
maxv: (x: Int, y: Int)Int

scala> maxf(1,2)
res4: Int = 2

scala> maxv(1,2)
res6: Int = 2
```

我们可以看出 `maxf: (x: Int, y: Int)Int` 跟 `maxv: (x: Int, y: Int)Int` 签名是一样的。在这里，Kotlin跟Scala在大括号的使用上，是完全不同的。

然后，调用方式是直接调用 `invoke()` 函数。通过REPL的编译错误提示信息，我们也可以看出，在Kotlin中，调用无参函数也是要加上括号 `()` 的。

kotlin 中 `return` 语句会从最近的函数或匿名函数中返回，但是在Lambda表达式中遇到return，则直接返回最近的外层函数。例如下面两个函数是不同的：

```
fun returnDemo_1() {
    println(" START " + ::returnDemo_1.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach {
        if (it == 3) return
        println(it)
    }
    println(" END " + ::returnDemo_2.name)
}

//1
//2

fun returnDemo_2() {
    println(" START " + ::returnDemo_2.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach(fun(a: Int) {
        if (a == 3) return
        println(a)
    })
    println(" END " + ::returnDemo_2.name)
}

//1
//2
//4
//5
```

`returnDemo_1` 在遇到 3 时会直接返回(有点类似循环体中的 `break` 行为)。最后输出

```
1
2
```

`returnDemo_2` 遇到 3 时会跳过它继续执行(有点类似循环体中的 `continue` 行为)。最后输出

```
1
2
4
```

```
5
```

在 `returnDemo_2` 中，我们用一个匿名函数替代 lambda 表达式。 匿名函数内部的 return 语句将从该匿名函数自身返回。

在Kotlin中，这是匿名函数和 lambda 表达式行为不一致的地方。当然，为了显式的指明 `return` 返回的地址，为此 kotlin 还提供了 `@Label` (标签) 来控制返回语句，且看下节分解。

3.5.7 标签（label）

在 Kotlin 中任何表达式都可以用标签（label）来标记。 标签的格式为标识符后跟 `@` 符号，例如： `abc@` 、 `jarOfLove@` 都是有效的标签。我们可以用Label标签来控制 `return` 、 `break` 或 `continue` 的跳转（jump）行为。

Kotlin 的函数是可以被嵌套的。它有函数字面量、局部函数等。 有了标签限制的 return，我们就可以从外层函数返回了。例如，从 lambda 表达式中返回， `returnDemo_2()` 我们可以显示指定 lambda 表达式中的return地址是其入口处。

代码示例：

```
fun returnDemo_3() {
    println(" START " + ::returnDemo_3.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach here@ {
        if (it == 3) return@here // 指令跳转到 lambda 表达式标签 here@ 处。继续下一个it=4
的遍历循环
        println(it)
    }
    println(" END " + ::returnDemo_3.name)
}

//1
//2
//4
//5
```

我们在 lambda 表达式开头处添加了标签 `here@` ， 我们可以这么理解：该标签相当于是记录了Lambda表达式的指令执行入口地址， 然后在表达式内部我们使用 `return@here` 来跳转至Lambda表达式该地址处。

另外，我们也可以使用隐式标签更方便。 该标签与接收该 lambda 的函数同名。

代码示例

```
fun returnDemo_4() {
    println(" START " + ::returnDemo_4.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
```

```
    intArray.forEach {
        if (it == 3) return@forEach // 从 lambda 表达式 @forEach 中返回。
        println(it)
    }

    println(" END " + ::returnDemo_4.name)
}
```

接收该Lambda表达式的函数是forEach, 所以我们可以直接使用 `return@forEach` ，来跳转到此处执行下一轮循环。

通常当我们在循环体中使用break，是跳出最近外层的循环：

```
fun breakDemo_1() {
    println("---------------  breakDemo_1 ---------------")
    for (outer in 1..5) {
        println("outer=" + outer)
        for (inner in 1..10) {
            println("inner=" + inner)
            if (inner % 2 == 0) {
                break
            }
        }
    }
}
```

输出

```
---------------  breakDemo_1 ---------------
outer=1
inner=1
inner=2
outer=2
inner=1
inner=2
outer=3
inner=1
inner=2
outer=4
inner=1
inner=2
outer=5
inner=1
inner=2
```

当我们想直接跳转到外层for循环，这个时候我们就可以使用标签了。

代码示例

```
fun breakDemo_2() {
    println("---------------  breakDemo_2 ---------------")
    outer@ for (outer in 1..5)
        for (inner in 1..10) {
            println("inner=" + inner)
            println("outer=" + outer)
            if (inner % 2 == 0) {
                break@outer
            }
        }
}
```

输出

```
---------------  breakDemo_2 ---------------
inner=1
outer=1
inner=2
outer=1
```

有时候，为了代码可读性，我们可以用标签来显式地指出循环体的跳转地址，比如说在 `breakDemo_1()` 中，我们可以用标签来指明内层循环的跳转地址：

```
fun breakDemo_3() {
    println("---------------  breakDemo_3 ---------------")
    for (outer in 1..5)
        inner@ for (inner in 1..10) {
            println("inner=" + inner)
            println("outer=" + outer)
            if (inner % 2 == 0) {
                break@inner
            }
        }
}
```

3.5.8 throw表达式

在 Kotlin 中 throw 是表达式，它的类型是特殊类型 Nothing。 该类型没有值。跟C、Java中的 `void` 意思一样。

```
>>> Nothing::class
class java.lang.Void
```

我们在代码中，用 Nothing 来标记无返回的函数：

```
>>> fun fail(msg:String):Nothing{ throw IllegalArgumentException(msg) }
>>> fail("XXXX")
java.lang.IllegalArgumentException: XXXX
    at Line57.fail(Unknown Source)
```

另外，如果把一个throw表达式的值赋值给一个变量，需要显式声明类型为 `Nothing` ， 代码示例如下

```
>>> val ex = throw Exception("YYYYYYYY")
error: 'Nothing' property type needs to be specified explicitly
val ex = throw Exception("YYYYYYYY")
    ^

>>> val ex:Nothing = throw Exception("YYYYYYYY")
java.lang.Exception: YYYYYYYY
```

另外，因为ex变量是Nothing类型，没有任何值，所以无法当做参数传给函数：

```
>>> println(ex)
error: overload resolution ambiguity:
@InlineOnly public inline fun println(message: Any?): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Boolean): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Byte): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Char): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: CharArray): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Double): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Float): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Int): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Long): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Short): Unit defined in kotlin.io
println(ex)
^

>>> ex
exception: org.jetbrains.kotlin.codegen.CompilationException: Back-end (JVM) Internal e
rror: Unregistered script: class Line62
Cause: Unregistered script: class Line62
File being compiled and position: (1,1) in /line64.kts
PsiElement: ex
The root cause was thrown at: ScriptContext.java:86
...
```

3.6 代码注释

正如 Java 和 JavaScript，Kotlin 支持行注释及块注释。

```
// 这是一个行注释

/* 这是一个多行的
   块注释。 */
```

与 Java 不同的是，Kotlin 的块注释可以嵌套。就是说，你可以这样注释：

```
/**
 * hhhh
 * /**
 *  fff
 *  /**
 *    ggggg
 *  */
 * */
 *
 * abc
 *
 */
fun main(args:Array<String>){
    val f = Functions()
    println(f.fvoid1())
    println(f.fvoid2())
    println(f.sum1(1,1))
    println(f.sum2(1,1))
}
```

3.7 语法与标识符

我们知道，任何一门编程语言都会有一些自己专用的关键字、符号以及规定的语法规则等等。程序员们使用这些基础词汇和语法规则来表达算法步骤，也就是写代码的过程。

词法分析是编译器对源码进行编译的基础步骤之一。词法分析是将源程序读入的字符序列，按照一定的规则转换成词法单元（Token）序列的过程。词法单元是语言中具有独立意义的最小单元，包括修饰符、关键字、常数、运算符、边界符等等。

3.7.1 修饰符

在Kotlin源码工程中的kotlin/grammar/src/modifiers.grm文件中，描述了Kotlin语言的修饰符，我们在此作简要注释说明：

```
/**
## Modifiers
```

```
*/

modifiers
  : (modifier | annotations)*
  ;

typeModifiers
  : (suspendModifier | annotations)*
  ;

modifier
  : classModifier
  : accessModifier
  : varianceAnnotation
  : memberModifier
  : parameterModifier
  : typeParameterModifier
  : functionModifier
  : propertyModifier
  ;

classModifier 类修饰符
  : "abstract" 抽象类
  : "final" 不可被继承final类
  : "enum" 枚举类
  : "open" 可继承open类
  : "annotation" 注解类
  : "sealed" 密封类
  : "data" 数据类
  ;

memberModifier
  : "override" 重写函数
  : "open" 可被重写
  : "final" 不可被重写
  : "abstract" 抽象函数
  : "lateinit" 后期初始化
  ;

accessModifier 访问权限控制, 默认是public
  : "private"
  : "protected"
  : "public"
  : "internal"  整个模块内（模块(module)是指一起编译的一组 Kotlin 源代码文件: 例如，一个 IntelliJ IDEA 模块，一个 Maven 工程, 或 Gradle 工程，通过 Ant 任务的一次调用编译的一组文件等）可访问
  ;

varianceAnnotation 泛型可变性
```

```
  : "in"
  : "out"
  ;

parameterModifier
  : "noinline"
  : "crossinline"
  : "vararg" 变长参数
  ;

typeParameterModifier
  : "reified"
  ;

functionModifier
  : "tailrec" 尾递归
  : "operator"
  : "infix"
  : "inline"
  : "external"
  : suspendModifier
  ;

propertyModifier
  : "const"
  ;

suspendModifier
  : "suspend"
  ;
```

这些修饰符的完整定义，在kotlin/compiler/frontend/src/org/jetbrains/kotlin/lexer/KtTokens.java源码中：

```
KtModifierKeywordToken[] MODIFIER_KEYWORDS_ARRAY =
            new KtModifierKeywordToken[] {
                    ABSTRACT_KEYWORD, ENUM_KEYWORD, OPEN_KEYWORD, INNER_KEYWORD, OVERRI
DE_KEYWORD, PRIVATE_KEYWORD,
                    PUBLIC_KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD, OUT_KEYWORD, I
N_KEYWORD, FINAL_KEYWORD, VARARG_KEYWORD,
                    REIFIED_KEYWORD, COMPANION_KEYWORD, SEALED_KEYWORD, LATEINIT_KEYWOR
D,
                    DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEYWORD, TAILREC_KEYWORD, EX
TERNAL_KEYWORD, ANNOTATION_KEYWORD, CROSSINLINE_KEYWORD,
                    CONST_KEYWORD, OPERATOR_KEYWORD, INFIX_KEYWORD, SUSPEND_KEYWORD, HE
ADER_KEYWORD, IMPL_KEYWORD
            };
```

```
    TokenSet MODIFIER_KEYWORDS = TokenSet.create(MODIFIER_KEYWORDS_ARRAY);

    TokenSet TYPE_MODIFIER_KEYWORDS = TokenSet.create(SUSPEND_KEYWORD);
    TokenSet TYPE_ARGUMENT_MODIFIER_KEYWORDS = TokenSet.create(IN_KEYWORD, OUT_KEYWORD)
;
    TokenSet RESERVED_VALUE_PARAMETER_MODIFIER_KEYWORDS = TokenSet.create(OUT_KEYWORD,
VARARG_KEYWORD);

    TokenSet VISIBILITY_MODIFIERS = TokenSet.create(PRIVATE_KEYWORD, PUBLIC_KEYWORD, IN
TERNAL_KEYWORD, PROTECTED_KEYWORD);
```

3.7.2 关键字(保留字)

```
TokenSet KEYWORDS = TokenSet.create(PACKAGE_KEYWORD, AS_KEYWORD, TYPE_ALIAS_KEYWORD, CL
ASS_KEYWORD, INTERFACE_KEYWORD,
                                        THIS_KEYWORD, SUPER_KEYWORD, VAL_KEYWORD, VAR_K
EYWORD, FUN_KEYWORD, FOR_KEYWORD,
                                        NULL_KEYWORD,
                                        TRUE_KEYWORD, FALSE_KEYWORD, IS_KEYWORD,
                                        IN_KEYWORD, THROW_KEYWORD, RETURN_KEYWORD, BREA
K_KEYWORD, CONTINUE_KEYWORD, OBJECT_KEYWORD, IF_KEYWORD,
                                        ELSE_KEYWORD, WHILE_KEYWORD, DO_KEYWORD, TRY_KE
YWORD, WHEN_KEYWORD,
                                        NOT_IN, NOT_IS, AS_SAFE,
                                        TYPEOF_KEYWORD
    );

    TokenSet SOFT_KEYWORDS = TokenSet.create(FILE_KEYWORD, IMPORT_KEYWORD, WHERE_KEYWOR
D, BY_KEYWORD, GET_KEYWORD,
                                             SET_KEYWORD, ABSTRACT_KEYWORD, ENUM_KEYWOR
D, OPEN_KEYWORD, INNER_KEYWORD,
                                             OVERRIDE_KEYWORD, PRIVATE_KEYWORD, PUBLIC_
KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD,
                                             CATCH_KEYWORD, FINALLY_KEYWORD, OUT_KEYWOR
D, FINAL_KEYWORD, VARARG_KEYWORD, REIFIED_KEYWORD,
                                             DYNAMIC_KEYWORD, COMPANION_KEYWORD, CONSTR
UCTOR_KEYWORD, INIT_KEYWORD, SEALED_KEYWORD,
                                             FIELD_KEYWORD, PROPERTY_KEYWORD, RECEIVER_
KEYWORD, PARAM_KEYWORD, SETPARAM_KEYWORD,
                                             DELEGATE_KEYWORD,
                                             LATEINIT_KEYWORD,
                                             DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEY
WORD, TAILREC_KEYWORD, EXTERNAL_KEYWORD,
                                             ANNOTATION_KEYWORD, CROSSINLINE_KEYWORD, C
ONST_KEYWORD, OPERATOR_KEYWORD, INFIX_KEYWORD,
                                             SUSPEND_KEYWORD, HEADER_KEYWORD, IMPL_KEYW
```

```
ORD
    );
```

其中，对应的关键字如下：

```
    KtKeywordToken PACKAGE_KEYWORD          = KtKeywordToken.keyword("package");
    KtKeywordToken AS_KEYWORD               = KtKeywordToken.keyword("as");
    KtKeywordToken TYPE_ALIAS_KEYWORD       = KtKeywordToken.keyword("typealias");
    KtKeywordToken CLASS_KEYWORD            = KtKeywordToken.keyword("class");
    KtKeywordToken THIS_KEYWORD             = KtKeywordToken.keyword("this");
    KtKeywordToken SUPER_KEYWORD            = KtKeywordToken.keyword("super");
    KtKeywordToken VAL_KEYWORD              = KtKeywordToken.keyword("val");
    KtKeywordToken VAR_KEYWORD              = KtKeywordToken.keyword("var");
    KtKeywordToken FUN_KEYWORD              = KtKeywordToken.keyword("fun");
    KtKeywordToken FOR_KEYWORD              = KtKeywordToken.keyword("for");
    KtKeywordToken NULL_KEYWORD             = KtKeywordToken.keyword("null");
    KtKeywordToken TRUE_KEYWORD             = KtKeywordToken.keyword("true");
    KtKeywordToken FALSE_KEYWORD            = KtKeywordToken.keyword("false");
    KtKeywordToken IS_KEYWORD               = KtKeywordToken.keyword("is");
    KtModifierKeywordToken IN_KEYWORD       = KtModifierKeywordToken.keywordModifier("i
n");
    KtKeywordToken THROW_KEYWORD            = KtKeywordToken.keyword("throw");
    KtKeywordToken RETURN_KEYWORD           = KtKeywordToken.keyword("return");
    KtKeywordToken BREAK_KEYWORD            = KtKeywordToken.keyword("break");
    KtKeywordToken CONTINUE_KEYWORD         = KtKeywordToken.keyword("continue");
    KtKeywordToken OBJECT_KEYWORD           = KtKeywordToken.keyword("object");
    KtKeywordToken IF_KEYWORD               = KtKeywordToken.keyword("if");
    KtKeywordToken TRY_KEYWORD              = KtKeywordToken.keyword("try");
    KtKeywordToken ELSE_KEYWORD             = KtKeywordToken.keyword("else");
    KtKeywordToken WHILE_KEYWORD            = KtKeywordToken.keyword("while");
    KtKeywordToken DO_KEYWORD               = KtKeywordToken.keyword("do");
    KtKeywordToken WHEN_KEYWORD             = KtKeywordToken.keyword("when");
    KtKeywordToken INTERFACE_KEYWORD        = KtKeywordToken.keyword("interface");

    // Reserved for future use:
    KtKeywordToken TYPEOF_KEYWORD           = KtKeywordToken.keyword("typeof");
...
    KtKeywordToken FILE_KEYWORD    = KtKeywordToken.softKeyword("file");
    KtKeywordToken FIELD_KEYWORD     = KtKeywordToken.softKeyword("field");
    KtKeywordToken PROPERTY_KEYWORD     = KtKeywordToken.softKeyword("property");
    KtKeywordToken RECEIVER_KEYWORD     = KtKeywordToken.softKeyword("receiver");
    KtKeywordToken PARAM_KEYWORD     = KtKeywordToken.softKeyword("param");
    KtKeywordToken SETPARAM_KEYWORD  = KtKeywordToken.softKeyword("setparam");
    KtKeywordToken DELEGATE_KEYWORD  = KtKeywordToken.softKeyword("delegate");
    KtKeywordToken IMPORT_KEYWORD    = KtKeywordToken.softKeyword("import");
    KtKeywordToken WHERE_KEYWORD     = KtKeywordToken.softKeyword("where");
    KtKeywordToken BY_KEYWORD        = KtKeywordToken.softKeyword("by");
    KtKeywordToken GET_KEYWORD       = KtKeywordToken.softKeyword("get");
```

```
    KtKeywordToken SET_KEYWORD       = KtKeywordToken.softKeyword("set");
    KtKeywordToken CONSTRUCTOR_KEYWORD = KtKeywordToken.softKeyword("constructor");
    KtKeywordToken INIT_KEYWORD        = KtKeywordToken.softKeyword("init");

    KtModifierKeywordToken ABSTRACT_KEYWORD  = KtModifierKeywordToken.softKeywordModifi
er("abstract");
    KtModifierKeywordToken ENUM_KEYWORD      = KtModifierKeywordToken.softKeywordModifi
er("enum");
    KtModifierKeywordToken OPEN_KEYWORD      = KtModifierKeywordToken.softKeywordModifi
er("open");
    KtModifierKeywordToken INNER_KEYWORD     = KtModifierKeywordToken.softKeywordModifi
er("inner");
    KtModifierKeywordToken OVERRIDE_KEYWORD  = KtModifierKeywordToken.softKeywordModifi
er("override");
    KtModifierKeywordToken PRIVATE_KEYWORD   = KtModifierKeywordToken.softKeywordModifi
er("private");
    KtModifierKeywordToken PUBLIC_KEYWORD    = KtModifierKeywordToken.softKeywordModifi
er("public");
    KtModifierKeywordToken INTERNAL_KEYWORD  = KtModifierKeywordToken.softKeywordModifi
er("internal");
    KtModifierKeywordToken PROTECTED_KEYWORD = KtModifierKeywordToken.softKeywordModifi
er("protected");
    KtKeywordToken CATCH_KEYWORD     = KtKeywordToken.softKeyword("catch");
    KtModifierKeywordToken OUT_KEYWORD       = KtModifierKeywordToken.softKeywordModifi
er("out");
    KtModifierKeywordToken VARARG_KEYWORD    = KtModifierKeywordToken.softKeywordModifi
er("vararg");
    KtModifierKeywordToken REIFIED_KEYWORD   = KtModifierKeywordToken.softKeywordModifi
er("reified");
    KtKeywordToken DYNAMIC_KEYWORD   = KtKeywordToken.softKeyword("dynamic");
    KtModifierKeywordToken COMPANION_KEYWORD = KtModifierKeywordToken.softKeywordModifi
er("companion");
    KtModifierKeywordToken SEALED_KEYWORD    = KtModifierKeywordToken.softKeywordModifi
er("sealed");

    KtModifierKeywordToken DEFAULT_VISIBILITY_KEYWORD = PUBLIC_KEYWORD;

    KtKeywordToken FINALLY_KEYWORD   = KtKeywordToken.softKeyword("finally");
    KtModifierKeywordToken FINAL_KEYWORD     = KtModifierKeywordToken.softKeywordModifi
er("final");

    KtModifierKeywordToken LATEINIT_KEYWORD = KtModifierKeywordToken.softKeywordModifie
r("lateinit");

    KtModifierKeywordToken DATA_KEYWORD    = KtModifierKeywordToken.softKeywordModifier
("data");
    KtModifierKeywordToken INLINE_KEYWORD    = KtModifierKeywordToken.softKeywordModifi
er("inline");
    KtModifierKeywordToken NOINLINE_KEYWORD    = KtModifierKeywordToken.softKeywordModi
```

```
fier("noinline");
    KtModifierKeywordToken TAILREC_KEYWORD     = KtModifierKeywordToken.softKeywordModif
ier("tailrec");
    KtModifierKeywordToken EXTERNAL_KEYWORD     = KtModifierKeywordToken.softKeywordModi
fier("external");
    KtModifierKeywordToken ANNOTATION_KEYWORD     = KtModifierKeywordToken.softKeywordMo
difier("annotation");
    KtModifierKeywordToken CROSSINLINE_KEYWORD     = KtModifierKeywordToken.softKeywordM
odifier("crossinline");
    KtModifierKeywordToken OPERATOR_KEYWORD = KtModifierKeywordToken.softKeywordModifie
r("operator");
    KtModifierKeywordToken INFIX_KEYWORD = KtModifierKeywordToken.softKeywordModifier("
infix");

    KtModifierKeywordToken CONST_KEYWORD = KtModifierKeywordToken.softKeywordModifier("
const");

    KtModifierKeywordToken SUSPEND_KEYWORD = KtModifierKeywordToken.softKeywordModifier
("suspend");

    KtModifierKeywordToken HEADER_KEYWORD = KtModifierKeywordToken.softKeywordModifier(
"header");
    KtModifierKeywordToken IMPL_KEYWORD = KtModifierKeywordToken.softKeywordModifier("i
mpl");
```

`this` 关键字

`this` 关键字持有当前对象的引用。我们可以使用 `this` 来引用变量或者成员函数，也可以使用 `return this` ，来返回某个类的引用。

代码示例

```
class ThisDemo {
    val thisis = "THIS IS"

    fun whatIsThis(): ThisDemo {
        println(this.thisis) //引用变量
        this.howIsThis()// 引用成员函数
        return this // 返回此类的引用
    }

    fun howIsThis(){
        println("HOW IS THIS ?")
    }
}
```

测试代码

```
    @Test
    fun testThisDemo(){
        val demo = ThisDemo()
        println(demo.whatIsThis())
    }
```

输出

```
THIS IS
HOW IS THIS ?
com.easy.kotlin.ThisDemo@475232fc
```

在类的成员中，this 指向的是该类的当前对象。

在扩展函数或者带接收者的函数字面值中， this 表示在点左侧传递的 接收者参数。

代码示例：

```
>>> val sum = fun Int.(x:Int):Int = this + x
>>> sum
kotlin.Int.(kotlin.Int) -> kotlin.Int
>>> 1.sum(1)
2
>>> val concat = fun String.(x:Any) = this + x
>>> "abc".concat(123)
abc123
>>> "abc".concat(true)
abctrue
```

如果 this 没有限定符，它指的是最内层的包含它的作用域。如果我们想要引用其他作用域中的 this，可以使用 this@label 标签。

代码示例：

```
class Outer {
    val oh = "Oh!"

    inner class Inner {

        fun m() {
            val outer = this@Outer
            val inner = this@Inner
            val pthis = this
            println("outer=" + outer)
```

```
            println("inner=" + inner)
            println("pthis=" + pthis)
            println(this@Outer.oh)

            val fun1 = hello@ fun String.() {
                val d1 = this // fun1 的接收者
                println("d1" + d1)
            }

            val fun2 = { s: String ->
                val d2 = this
                println("d2=" + d2)
            }

            "abc".fun1()

            fun2

        }
    }

}
```

测试代码：

```
    @Test
    fun testThisKeyWord() {
        val outer = Outer()
        outer.Inner().m()
    }
```

输出

```
outer=com.easy.kotlin.Outer@5114e183
inner=com.easy.kotlin.Outer$Inner@5aa8ac7f
pthis=com.easy.kotlin.Outer$Inner@5aa8ac7f
Oh!
d1abc
```

super 关键字

super关键字持有指向其父类的引用。

代码示例：

```
open class Father {
```

```
    open val firstName = "Chen"
    open val lastName = "Jason"

    fun ff() {
        println("FFF")
    }
}

class Son : Father {
    override var firstName = super.firstName
    override var lastName = "Jack"

    constructor(lastName: String) {
        this.lastName = lastName
    }

    fun love() {
        super.ff() // 调用父类方法
        println(super.firstName + " " + super.lastName + " Love " + this.firstName + "
" + this.lastName)
    }
}
```

测试代码

```
    @Test
    fun testSuperKeyWord() {
        val son = Son("Harry")
        son.love()
    }
```

输出

```
FFF
Chen Jason Love Chen Harry
```

3.7.3 操作符和操作符的重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如 `+` 或 `*` ）和固定的优先级。这些操作符的符号定义如下：

```
    KtSingleValueToken LBRACKET    = new KtSingleValueToken("LBRACKET", "[");
    KtSingleValueToken RBRACKET    = new KtSingleValueToken("RBRACKET", "]");
    KtSingleValueToken LBRACE      = new KtSingleValueToken("LBRACE", "{");
    KtSingleValueToken RBRACE      = new KtSingleValueToken("RBRACE", "}");
```

```
    KtSingleValueToken LPAR        = new KtSingleValueToken("LPAR", "(");
    KtSingleValueToken RPAR        = new KtSingleValueToken("RPAR", ")");
    KtSingleValueToken DOT         = new KtSingleValueToken("DOT", ".");
    KtSingleValueToken PLUSPLUS    = new KtSingleValueToken("PLUSPLUS", "++");
    KtSingleValueToken MINUSMINUS  = new KtSingleValueToken("MINUSMINUS", "--");
    KtSingleValueToken MUL         = new KtSingleValueToken("MUL", "*");
    KtSingleValueToken PLUS        = new KtSingleValueToken("PLUS", "+");
    KtSingleValueToken MINUS       = new KtSingleValueToken("MINUS", "-");
    KtSingleValueToken EXCL        = new KtSingleValueToken("EXCL", "!");
    KtSingleValueToken DIV         = new KtSingleValueToken("DIV", "/");
    KtSingleValueToken PERC        = new KtSingleValueToken("PERC", "%");
    KtSingleValueToken LT          = new KtSingleValueToken("LT", "<");
    KtSingleValueToken GT          = new KtSingleValueToken("GT", ">");
    KtSingleValueToken LTEQ        = new KtSingleValueToken("LTEQ", "<=");
    KtSingleValueToken GTEQ        = new KtSingleValueToken("GTEQ", ">=");
    KtSingleValueToken EQEQEQ      = new KtSingleValueToken("EQEQEQ", "===");
    KtSingleValueToken ARROW       = new KtSingleValueToken("ARROW", "->");
    KtSingleValueToken DOUBLE_ARROW       = new KtSingleValueToken("DOUBLE_ARROW", "=>"
);
    KtSingleValueToken EXCLEQEQEQ  = new KtSingleValueToken("EXCLEQEQEQ", "!==");
    KtSingleValueToken EQEQ        = new KtSingleValueToken("EQEQ", "==");
    KtSingleValueToken EXCLEQ      = new KtSingleValueToken("EXCLEQ", "!=");
    KtSingleValueToken EXCLEXCL    = new KtSingleValueToken("EXCLEXCL", "!!");
    KtSingleValueToken ANDAND      = new KtSingleValueToken("ANDAND", "&&");
    KtSingleValueToken OROR        = new KtSingleValueToken("OROR", "||");
    KtSingleValueToken SAFE_ACCESS = new KtSingleValueToken("SAFE_ACCESS", "?.");
    KtSingleValueToken ELVIS       = new KtSingleValueToken("ELVIS", "?:");
    KtSingleValueToken QUEST       = new KtSingleValueToken("QUEST", "?");
    KtSingleValueToken COLONCOLON  = new KtSingleValueToken("COLONCOLON", "::");
    KtSingleValueToken COLON       = new KtSingleValueToken("COLON", ":");
    KtSingleValueToken SEMICOLON   = new KtSingleValueToken("SEMICOLON", ";");
    KtSingleValueToken DOUBLE_SEMICOLON   = new KtSingleValueToken("DOUBLE_SEMICOLON",
";;");
    KtSingleValueToken RANGE       = new KtSingleValueToken("RANGE", "..");
    KtSingleValueToken EQ          = new KtSingleValueToken("EQ", "=");
    KtSingleValueToken MULTEQ      = new KtSingleValueToken("MULTEQ", "*=");
    KtSingleValueToken DIVEQ       = new KtSingleValueToken("DIVEQ", "/=");
    KtSingleValueToken PERCEQ      = new KtSingleValueToken("PERCEQ", "%=");
    KtSingleValueToken PLUSEQ      = new KtSingleValueToken("PLUSEQ", "+=");
    KtSingleValueToken MINUSEQ     = new KtSingleValueToken("MINUSEQ", "-=");
    KtKeywordToken NOT_IN      = KtKeywordToken.keyword("NOT_IN", "!in");
    KtKeywordToken NOT_IS      = KtKeywordToken.keyword("NOT_IS", "!is");
    KtSingleValueToken HASH        = new KtSingleValueToken("HASH", "#");
    KtSingleValueToken AT          = new KtSingleValueToken("AT", "@");

    KtSingleValueToken COMMA       = new KtSingleValueToken("COMMA", ",");
```

3.7.4 操作符优先级（Precedence）

优先级	标题	符号
最高	后缀（Postfix ）	`++` , `--` , `.` , `?.` , `?`
	前缀（Prefix）	`-` , `+` , `++` , `--` , `!` , `labelDefinition` `@`
	右手类型运算（Type RHS, right-hand side class type (RHS) ）	`:` , `as` , `as?`
	乘除取余（Multiplicative）	`*` , `/` , `%`
	加减（Additive ）	`+` , `-`
	区间范围（Range）	`..`
	Infix函数	例如，给 `Int` 定义扩展 `infix fun Int.shl(x: Int): Int {...}` ,这样调用 `1 shl 2` ，等同于 `1.shl(2)`
	Elvis操作符	`?:`
	命名检查符（Named checks）	`in` , `!in` , `is` , `!is`
	比较大小（Comparison）	`<` , `>` , `<=` , `>=`
	相等性判断（Equality）	`==` , `\!==`
	与 （Conjunction）	`&&`
	或 （Disjunction）	`ll`
最低	赋值（Assignment）	`=` , `+=` , `-=` , `*=` , `/=` , `%=`

注：Markdown表格语法： `ll` 是 `||` 。

为实现这些的操作符，Kotlin为二元操作符左侧的类型和一元操作符的参数类型，提供了相应的函数或扩展函数。

例如在kotlin/core/builtins/native/kotlin/Primitives.kt代码中，对基本类型Int的操作符的实现代码如下

```
public class Int private constructor() : Number(), Comparable<Int> {
    ...

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
```

```
     */
    public operator fun compareTo(other: Byte): Int

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public operator fun compareTo(other: Short): Int

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public override operator fun compareTo(other: Int): Int

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public operator fun compareTo(other: Long): Int

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public operator fun compareTo(other: Float): Int

    /**
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public operator fun compareTo(other: Double): Int

    /** Adds the other value to this value. */
    public operator fun plus(other: Byte): Int
    /** Adds the other value to this value. */
    public operator fun plus(other: Short): Int
    /** Adds the other value to this value. */
    public operator fun plus(other: Int): Int
```

```
    /** Adds the other value to this value. */
    public operator fun plus(other: Long): Long
    /** Adds the other value to this value. */
    public operator fun plus(other: Float): Float
    /** Adds the other value to this value. */
    public operator fun plus(other: Double): Double

    /** Subtracts the other value from this value. */
    public operator fun minus(other: Byte): Int
    /** Subtracts the other value from this value. */
    public operator fun minus(other: Short): Int
    /** Subtracts the other value from this value. */
    public operator fun minus(other: Int): Int
    /** Subtracts the other value from this value. */
    public operator fun minus(other: Long): Long
    /** Subtracts the other value from this value. */
    public operator fun minus(other: Float): Float
    /** Subtracts the other value from this value. */
    public operator fun minus(other: Double): Double

    /** Multiplies this value by the other value. */
    public operator fun times(other: Byte): Int
    /** Multiplies this value by the other value. */
    public operator fun times(other: Short): Int
    /** Multiplies this value by the other value. */
    public operator fun times(other: Int): Int
    /** Multiplies this value by the other value. */
    public operator fun times(other: Long): Long
    /** Multiplies this value by the other value. */
    public operator fun times(other: Float): Float
    /** Multiplies this value by the other value. */
    public operator fun times(other: Double): Double

    /** Divides this value by the other value. */
    public operator fun div(other: Byte): Int
    /** Divides this value by the other value. */
    public operator fun div(other: Short): Int
    /** Divides this value by the other value. */
    public operator fun div(other: Int): Int
    /** Divides this value by the other value. */
    public operator fun div(other: Long): Long
    /** Divides this value by the other value. */
    public operator fun div(other: Float): Float
    /** Divides this value by the other value. */
    public operator fun div(other: Double): Double

    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
```

```
    public operator fun mod(other: Byte): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
    public operator fun mod(other: Short): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
    public operator fun mod(other: Int): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
    public operator fun mod(other: Long): Long
    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
    public operator fun mod(other: Float): Float
    /** Calculates the remainder of dividing this value by the other value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other)"), DeprecationLevel.W
ARNING)
    public operator fun mod(other: Double): Double

    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Byte): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Short): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Int): Int
    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Long): Long
    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Float): Float
    /** Calculates the remainder of dividing this value by the other value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Double): Double

    /** Increments this value. */
    public operator fun inc(): Int
    /** Decrements this value. */
    public operator fun dec(): Int
    /** Returns this value. */
    public operator fun unaryPlus(): Int
    /** Returns the negative of this value. */
    public operator fun unaryMinus(): Int
```

```
    /** Creates a range from this value to the specified [other] value. */
    public operator fun rangeTo(other: Byte): IntRange
    /** Creates a range from this value to the specified [other] value. */
    public operator fun rangeTo(other: Short): IntRange
    /** Creates a range from this value to the specified [other] value. */
    public operator fun rangeTo(other: Int): IntRange
    /** Creates a range from this value to the specified [other] value. */
    public operator fun rangeTo(other: Long): LongRange

    /** Shifts this value left by [bits]. */
    public infix fun shl(bitCount: Int): Int
    /** Shifts this value right by [bits], filling the leftmost bits with copies of the
sign bit. */
    public infix fun shr(bitCount: Int): Int
    /** Shifts this value right by [bits], filling the leftmost bits with zeros. */
    public infix fun ushr(bitCount: Int): Int
    /** Performs a bitwise AND operation between the two values. */
    public infix fun and(other: Int): Int
    /** Performs a bitwise OR operation between the two values. */
    public infix fun or(other: Int): Int
    /** Performs a bitwise XOR operation between the two values. */
    public infix fun xor(other: Int): Int
    /** Inverts the bits in this value. */
    public fun inv(): Int

    public override fun toByte(): Byte
    public override fun toChar(): Char
    public override fun toShort(): Short
    public override fun toInt(): Int
    public override fun toLong(): Long
    public override fun toFloat(): Float
    public override fun toDouble(): Double
}
```

从源代码我们可以看出，重载操作符的函数需要用 `operator` 修饰符标记。中缀操作符的函数使用 `infix` 修饰符标记。

3.7.5 一元操作符（unary operation）

前缀操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

例如，当编译器处理表达式 `+a` 时，它将执行以下步骤：

- 确定 `a` 的类型，令其为 `T` 。
- 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus（）` ，即成员函数或扩展函数。
- 如果函数不存在或不明确，则导致编译错误。
- 如果函数存在且其返回类型为 `R` ，那就表达式 `+a` 具有类型 `R` 。

编译器对这些操作以及所有其他操作都针对基本类型做了优化，不会引入函数调用的开销。

以下是如何重载一元减运算符的示例：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/10.
 */

class OperatorDemo {

}

data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)
```

测试代码：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/6/10.
 */
@RunWith(JUnit4::class)
class OperatorDemoTest {

    @Test
    fun testPointUnaryMinus() {
        val p = Point(1, 1)
        val np = -p
        println(np) //Point(x=-1, y=-1)
    }
}
```

递增和递减

表达式	翻译为
`a++`	`a.inc()` 返回值是 `a`
`a--`	`a.dec()` 返回值是 `a`
`++a`	`a.inc()` 返回值是 `a+1`
`--a`	`a.dec()` 返回值是 `a-1`

`inc()` 和 `dec()` 函数必须返回一个值，它用于赋值给使用 `++` 或 `--` 操作的变量。

编译器执行以下步骤来解析后缀形式的操作符，例如 `a++` ：

- 确定 `a` 的类型，令其为 `T` 。
- 查找一个适用于类型为 `T` 的接收者的、带有 `operator` 修饰符的无参数函数 `inc()` 。
- 检查函数的返回类型是 `T` 的子类型。

计算表达式的步骤是：

- 把 `a` 的初始值存储到临时存储 `a_` 中
- 把 `a.inc()` 结果赋值给 `a`
- 把 `a_` 作为表达式的结果返回

（ `a--` 同理分析）。

对于前缀形式 `++a` 和 `--a` 解析步骤类似，但是返回值是取的新值来返回：

- 把 `a.inc()` 结果赋值给 `a`
- 把 `a` 的新值 `a+1` 作为表达式结果返回

（ `--a` 同理分析）。

3.7.6 二元操作符

算术运算符

表达式	翻译为
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)` 、 `a.mod(b)`
`a..b`	`a.rangeTo(b)`

代码示例

```
>>> val a=10
>>> val b=3
>>> a+b
13
>>> a-b
7
>>> a/b
3
>>> a%b
1
>>> a..b
10..3
>>> b..a
3..10
```

字符串的 `+` 运算符重载

先用代码举个例子：

```
>>> ""+1
1
>>> 1+""
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+""
 ^
```

从上面的示例，我们可以看出，在Kotlin中 `1+""` 是不允许的(这地方，相比Scala，写这样的Kotlin代码就显得不大友好)，只能显式调用 `toString` 来相加：

```
>>> 1.toString()+""
1
```

自定义重载的 `+` 运算符

下面我们使用一个计数类 Counter 重载的 `+` 运算符来增加index的计数值。

代码示例

```
data class Counter(var index: Int)

operator fun Counter.plus(increment: Int): Counter {
    return Counter(index + increment)
}
```

测试类

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/6/10.
 */
@RunWith(JUnit4::class)
class OperatorDemoTest
    @Test
    fun testCounterIndexPlus() {
        val c = Counter(1)
        val cplus = c + 10
        println(cplus) //Counter(index=11)
    }
}
```

`in` 操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

索引访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i] = b`	`a.set(i, b)`

方括号转换为调用带有适当数量参数的 `get` 和 `set` 。

调用操作符

表达式	翻译为

`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`

圆括号转换为调用带有适当数量参数的 `invoke` 。

计算并赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.modAssign(b)`

对于赋值操作，例如 `a += b` ，编译器会试着生成 `a = a + b` 的代码（这里包含类型检查： `a + b` 的类型必须是 `a` 的子类型）。

相等与不等操作符

Kotlin 中有两种类型的相等性：

- 引用相等 `===` `!==` （两个引用指向同一对象）
- 结构相等 `==` `!=` （ 使用 `equals()` 判断）

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这个 `==` 操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 `null` 值。

意思是：如果 a 不是 null 则调用 `equals(Any?)` 函数并返回其值；否则（即 `a === null` ）就计算 `b === null` 的值并返回。

当与 null 显式比较时， `a == null` 会被自动转换为 `a=== null`

注意： `===` 和 `!==` 不可重载。

Elvis 操作符 `?:`

在Kotin中，Elvis操作符特定是跟null比较。也就是说

```
y = x?:0
```

等价于

```
val y = if(x!==null) x else 0
```

主要用来作`null`安全性检查。

Elvis操作符 `?:` 是一个二元运算符，如果第一个操作数为真，则返回第一个操作数，否则将计算并返回其第二个操作数。它是三元条件运算符的变体。命名灵感来自猫王的发型风格。

Kotlin中没有这样的三元运算符 `true?1:0` ，取而代之的是`if(true) 1 else 0` 。而Elvis操作符算是精简版的三元运算符。

我们在Java中使用的三元运算符的语法，你通常要重复变量两次， 示例：

```
String name = "Elvis Presley";
String displayName = (name != null) ? name : "Unknown";
```

取而代之，你可以使用Elvis操作符。

```
String name = "Elvis Presley";
String displayName = name?:"Unknown"
```

我们可以看出，用Elvis操作符（?:）可以把带有默认值的if/else结构写的及其短小。用Elvis操作符不用检查null（避免了`NullPointerException` ），也不用重复变量。

这个Elvis操作符功能在Spring 表达式语言 (SpEL)中提供。

在Kotlin中当然就没有理由不支持这个特性。

代码示例：

```
>>> val x = null
>>> val y = x?:0
>>> y
0
>>> val x = false
>>> val y = x?:0
>>> y
false
>>> val x = ""
>>> val y = x?:0
>>> y

>>> val x = "abc"
>>> val y = x?:0
>>> y
abc
```

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较都转换为对 `compareTo` 的调用，这个函数需要返回 `Int` 值

用infix函数自定义中缀操作符

我们可以通过自定义infix函数来实现中缀操作符。

代码示例

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

infix fun Person.grow(years: Int): Person {
    return Person(name, age + years)
}
```

测试代码

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4


@RunWith(JUnit4::class)
class InfixFunctionDemoTest {

    @Test fun testInfixFuntion() {
        val person = Person("Jack", 20)

        println(person.grow(2))

        println(person grow 2)
    }
}
```

输出

```
Person(name=Jack, age=22)
Person(name=Jack, age=22)
```

3.8 函数扩展和属性扩展(Extensions)

Kotlin 支持 扩展函数 和 扩展属性。其能够扩展一个类的新功能而无需继承该类或使用像装饰者这样的设计模式等。

大多数时候我们在顶层定义扩展，即直接在包里：

```
package com.easy.kotlin

val <T> List<T>.lastIndex: Int get() = size - 1

fun String.notEmpty(): Boolean {
    return !this.isEmpty()
}
```

这样我们就可以在整个包里使用这些扩展。

要使用其他包的扩展，我们需要在调用方导入它：

```
package com.example.usage

import foo.bar.goo // 导入所有名为“goo”的扩展
                   // 或者
import foo.bar.*   // 从“foo.bar”导入一切

fun usage(baz: Baz) {
    baz.goo()
}
```

3.8.1 扩展函数

声明一个扩展函数，我们需要用被扩展的类型来作为前缀。

比如说，我们不喜欢类似下面的双重否定式的逻辑判断（绕脑子）：

```
>>> !"123".isEmpty()
true
```

我们就可以为 `String` 类型扩展一个 `notEmpty()` 函数：

```
>>> fun String.notEmpty():Boolean{
```

```
... return !this.isEmpty()
... }

>>> "".notEmpty()
false

>>> "123".notEmpty()
true
```

下面代码为 `MutableList<Int>` 添加一个 `swap` 函数：

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // this对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

这个 `this` 关键字在扩展函数内部对应到接收者对象（传过来的在点 `.` 符号前的对象） 现在，我们对任意 `MutableList<Int>` 调用该函数了。

当然，这个函数对任何 `MutableList<T>` 起作用，我们可以泛化它：

```
    fun <T> MutableList<T>.mswap(index1: Int, index2: Int) {
        val tmp = this[index1] // “this”对应该列表
        this[index1] = this[index2]
        this[index2] = tmp
    }
```

为了在接收者类型表达式中使用泛型，我们要在函数名前声明泛型参数。

完整代码示例

```
package com.easy.kotlin

val <T> List<T>.lastIndex: Int get() = size - 1

fun String.notEmpty(): Boolean {
    return !this.isEmpty()
}

fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // this对应该列表m
    this[index1] = this[index2]
    this[index2] = tmp
}

fun <T> MutableList<T>.mswap(index1: Int, index2: Int) {
```

```
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}

class ExtensionsDemo {

    fun useExtensions() {
        val a = "abc"
        println(a.notEmpty())//true

        val mList = mutableListOf<Int>(1, 2, 3, 4, 5)
        println("Before Swap:")
        println(mList)//[1, 2, 3, 4, 5]
        mList.swap(0, mList.size - 1)
        println("After Swap:")
        println(mList)//[5, 2, 3, 4, 1]

        val mmList = mutableListOf<String>("a12", "b34", "c56", "d78")
        println("Before Swap:")
        println(mmList)//[a12, b34, c56, d78]
        mmList.mswap(1, 2)
        println("After Swap:")
        println(mmList)//[a12, c56, b34, d78]

        val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
        println("Before Swap:")
        println(mmmList)
        mmmList.mswap(0, mmmList.lastIndex)
        println("After Swap:")
        println(mmmList)
    }

    class Inner {
        fun useExtensions() {
            val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
            println(mmmList.lastIndex)
        }
    }
}
```

测试代码

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4
```

```
@RunWith(JUnit4::class)
class ExtensionsDemoTest {
    @Test fun testExtensionsDemo() {
        val demo = ExtensionsDemo()
        demo.useExtensions()
    }
}
```

扩展不是真正的修改他们所扩展的类。我们定义一个扩展，其实并没有在一个类中插入新函数，仅仅是通过该类型的变量，用点 `.` 表达式去调用这个新函数。

3.8.2 扩展属性

和函数类似，Kotlin 支持扩展属性：

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

注意：由于扩展没有实际的将成员插入类中，因此对扩展的属性来说，它的行为只能由显式提供的getters/setters 定义。

代码示例：

```
package com.easy.kotlin

val <T> List<T>.lastIndex: Int get() = size - 1
```

我们可以直接使用包 `com.easy.kotlin` 中扩展的属性 `lastIndex` ：

```
48
49          val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
50          println("Before Swap:")
51          println(mmmList)
52          mmmList.mswap(0, mmmList.lastIndex)
53          println("After Swap:   lastIndex for List<T> in com.easy.kotlin        Int
54          println(mmmList)       lastIndexOf(element: Int)                       Int
55      }                          ^↓ and ^↑ will move caret down and up in the editor >>
56
57
58      class Inner {
59          fun useExtensions() {
60              val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
```

3.9 空指针安全(Null-safety)

我们写代码的时候知道，在Java中NPE（NullPointerExceptions）是一件成程序员几近崩溃的事情。很多时候，虽然费尽体力脑力，仍然防不胜防。

以前，当我们不确定一个DTO类中的字段是否已初始化时，可以使用@Nullable和@NotNull注解来声明，但功能很有限。

现在好了，Kotlin在编译器级别，把你之前在Java中需要写的null check代码完成了。

但是，当我们的代码

- 显式调用 `throw NullPointerException()`
- 使用了 `!!` 操作符
- 调用的外部 Java 代码有NPE
- 对于初始化，有一些数据不一致（如一个未初始化的 `this` 用于构造函数的某个地方）

也可能会发生NPE。

在Kotlin中 `null` 等同于空指针。我们来通过代码来看一下 `null` 的有趣的特性：

首先，一个非空引用不能直接赋值为 `null` ：

```
>>> var a="abc"
>>> a=null
error: null can not be a value of a non-null type String
a=null
  ^

>>> var one=1
>>> one=null
error: null can not be a value of a non-null type Int
one=null
    ^

>>> var arrayInts = intArrayOf(1,2,3)
>>> arrayInts=null
error: null can not be a value of a non-null type IntArray
arrayInts=null
          ^
```

这样，我们就可以放心地调用 `a` 的方法或者访问它的属性，不会导致 `NPE` ：

```
>>> val a="abc"
>>> a.length
3
```

如果要允许为空，我们可以在变量的类型后面加个问号 `?` 声明一个变量为可空的：

```
>>> var a:String?="abc"
>>> a=null
>>> var one:Int?=1
>>> one=null
```

```
>>> var arrayInts:IntArray?=intArrayOf(1,2,3)
>>> arrayInts=null
>>> arrayInts
null
```

如果我们声明了一个可空 `String?` 类型变量 `na` ，然后直接调用 `length` 属性，这将是不安全的。编译器会直接报错：

```
>>> var na:String?="abc"
>>> na=null
>>> na.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiv
er of type String?
na.length
  ^
```

我们使用安全调用 `?.` 和 非空断言调用 `!!.`

```
>>> na?.length
null
>>> na!!.length
kotlin.KotlinNullPointerException
```

我们可以看出，代码返回了 `null` 和 `kotlin.KotlinNullPointerException` 。

安全调用在链式调用中很有用。在调用链中如果任意一个属性（环节）为空，这个链式调用就会安全返回 null。

如果要只对非空值执行某个操作，安全调用操作符可以与 `let` （以调用者的值作为参数来执行指定的函数块，并返回其结果）一起使用：

```
>>> val listWithNulls: List<String?> = listOf("A", "B",null)
>>> listWithNulls
[A, B, null]

>>> listWithNulls.forEach{
... it?.let{println(it)}
... }
A
B
```

本章小结

本章我们学习了Kotlin语言的基本词汇（关键字、标识符等）、句子（流程控制、表达式、操作符等）和一些基础语法。同时，学习了空指针安全、扩展函数与扩展属性等的语言特性。

我们将在下一章节中介绍Kotlin的基本类型和类型系统。

参考资料

1.https://www.kotlincn.net/docs/reference/grammar.html

2.https://en.wikipedia.org/wiki/Elvis_operator

第4章 基本数据类型与类型系统

到目前为止，我们已经了解了Kotlin的基本符号以及基础语法。我们可以看出，使用Kotlin写的代码更简洁、可读性更好、更富有生产力。

本章我们来学习一下Kotlin的基本数据类型与类型系统。

> 道生一，一生二，二生三，三生万物 (老子《道德经》第四十二章)

在计算机科学中，最早的类型系统用来区别数字里面的整数和浮点数。

在20世纪五六十年代，这种分类扩展到了结构化的数据和高阶函数中。

70年代，引入了几个更为丰富的概念，例如：参数化类型，抽象数据类型，模块系统，子类型等等，类型系统作为一个独立的领域形成了。

在每一门编程语言中，都有一个特定的类型系统(Type System)。类型系统是一门编程语言最核心也是最基础的部分。我们这里说的类型系统，可以简单理解为以下两个部分：

- 一组基本类型构成的PTS（Primary Type Set，基本类型集合）；
- PTS上定义的一系列组合、运算、转换规则等。

这一简单优雅而惊人的世界构成观，贯穿了人类现实世界和计算机编程语言所定义的虚拟世界。或许语言的设计者也没有料想到，但是最终的结果确实是有限的设计导出了无限的可能性。

本章我们将学习Kotlin语言的基本类型，以及简单介绍Kotlin的类型系统。

4.1 什么是类型？

> 一切皆是映射

在计算机中，任何数值都是以一组比特（01）组成的，硬件无法区分内存地址、脚本、字符、整数、以及浮点数。这个时候，我们使用类型赋予一组比特以特定的意义。

类型（Type），本质上就是内存中的数值或变量对象的逻辑映射。

《周易》有云：

> 易有太极，是生两仪，两仪生四象，四象生八卦。(《易传·系辞上传》)。

这里所包含的思想，跟我们这里所说的类型系统的思想有着异曲同工之妙。

类型系统用于定义如何将编程语言中的数值和表达式归类为许多不同的类型，如何操作这些类型，这些类型如何互相作用等。

类型系统在各种语言之间有非常大的不同，主要的差异存在于编译时期的语法，以及运行时期的操作实现方式。

类型系统提供的主要功能有：

- 安全性

编译器可以使用类型来检查无意义的，或者是可能无效的代码。例如，在强类型的语言中，如果没有对字符串的 `+` 进行重载，那么表达式

```
"Hello, World" + 3
```

就会被编译器检测出来，因为不能对字符串加上一个整数。强类型提供更多的安全性。

但是，为了让程序员可以写出极简的代码，很多语言都提供了操作符重载的机制。比如说，在Scala中，上面的代码是可以被正确执行的（重载了 `+` 操作符）

```
scala> "Hello,World"+1
res15: String = Hello,World1

scala> 1+"Hello,World"
res16: String = 1Hello,World
```

但是在Kotlin中， 由于 `Int` 类型没有对 `+` 实现重载，所以情况是这样

```
>>> "Hello,World"+1
Hello,World1
>>> 1+"Hello,World"
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+"Hello,World"
 ^
```

- 最优化

静态类型检查可提供有用的信息给编译器。编译器可以使用更有效率的机器指令，实现编译器优化。

- 可读性

- 抽象化（或模块化）

类型本质上是对较低层次的逻辑单元进行高层次的逻辑抽象。这样我们就可以直接使用类型在较高层次的方式思考，而不是繁重的低层次实现。

例如，我们可以将字符串想成一个值，以此取代仅仅是字节的数组。字符串就是一个抽象数据类型。

从01到类型，从类型到接口API，再到软件服务，都可以看做是广义的“类型”范畴。

程序中的变量在程序执行期间，可能会有不同的取值范围，我们可以把变量可取值的最大范围称为这个变量的类型。例如，具有类型Boolean的变量x，在程序执行期间，只能取布尔值。指定变量类型的程序设计语言，称为类型化的语言（typed language）。

如果一个语言，不限制变量的取值，称为无类型语言（untyped language），我们既可以说它不具有类型，也可以说它具有一个通用类型，这个类型的取值范围是程序中所有可能的值。

类型系统是类型化语言的一个组成部分，它用来计算和跟踪程序中所有表达式的类型，从而判断某段程序是否表现良好（well behaved）。

如果程序语言的语法中含有类型标记，就称该语言是显式类型化的（explicitly typed），否则就称为隐式类型化的（implicitly typed）。

像C、C++、Java等语言，都是显式类型化的。而像ML、Haskell、Groovy等可以省略类型声明，它们的类型系统会自动推断出程序的类型。

4.2 编译时类型与运行时类型

> Koltin是一门强类型的、静态类型、支持隐式类型的显式类型语言。

4.2.1 弱类型（Weakly checked language）与强类型（Strongly checked language）

类型系统最主要的作用是，通过检查类型的运算和转换过程，来减少类型错误的发生。如果一个语言的编译器引入越多的类型检查的限制，就可以称这个语言的类型检查越强，反之越弱。根据类型检查的强弱，我们把编程语言分为

- 弱类型语言
- 强类型语言

弱类型语言在运行时会隐式做数据类型转换。 强类型语言在运行时会确保不会发生未经明确转换（显式调用）的类型转换。

但是另一方面，强和弱只是相对的。

Kotlin是强类型语言。

4.2.2 静态类型（Statically checked language）与动态类型（Dynamically

checked language）

类型检查可发生在编译时期（静态检查）或运行时期（动态检查）。这样我们将编程语言分为

- 静态类型语言

- 动态类型语言

静态类型检查是基于编译器来分析源码本身来确保类型安全。静态类型检查能让很多bug在编码早期被捕捉到，并且它也能优化运行。因为如果编译器在编译时已经证明程序是类型安全的，就不用在运行时进行动态的类型检查，编译过后的代码会更优化，运行更快。

动态类型语言是在运行时期进行类型标记的检查，因为变量所约束的值，可经由运行路径获得不同的标记。关于动态类型，有个很形象的说法：

> 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。——詹姆斯·惠特科姆·莱利（James Whitcomb Riley,1849-1916）

Kotlin是静态类型语言。

4.2.3 显式类型（Explicitly typed language）与隐式类型（Implicitly typed language）

还有一种区分方法是，根据变量名是否需要显式给出类型的声明，来将语言分为

- 显式类型语言
- 隐式类型语言

前者需要在定义变量时显式给出变量的类型，而后者可以使用类型推论来确定变量的类型。

大多数静态类型语言，例如 Java、C/C++ 都是显式类型语言。但是有些则不是，如 Haskell、ML 等，它们可以基于变量的操作来推断其类型；

Scala 是静态类型语言，它使用类型推断功能来支持隐式类型。

Kotlin 跟Scala类似，它也使用类型推断支持隐式类型。但是，在一些场景下也需要显式声明变量的类型，所以我们可以说，同时也是显式类型。

4.3 根类型Any

Kotlin 中所有类都有一个共同的超类 Any ，如果类声明时没有指定超类，则默认为 Any 。我们来看一段代码：

```
>>> val any = Any()
>>> any
java.lang.Object@2e377400
>>> any::class
class kotlin.Any
>>> any::class.java
class java.lang.Object
```

也就是说，Any在运行时，其类型自动映射成 `java.lang.Object` 。我们知道，在Java中Object类是所有引用类型的父类。但是不包括基本类型： `byte` `int` `long` 等，基本类型对应的包装类是引用类型，其父类是Object。而在Kotlin中，直接统一——所有类型都是引用类型，统一继承父类 `Any` 。

Any是Java的等价Object类。但是跟Java不同的是，Kotlin中语言内部的类型和用户定义类型之间，并没有像Java那样划清界限。它们是同一类型层次结构的一部分。

Any 只有 equals() 、 hashCode() 和 toString() 三个方法。其源码是

```
public open class Any {
    /**
     * Indicates whether some other object is "equal to" this one. Implementations must
 fulfil the following
     * requirements:
     *
     * * Reflexive: for any non-null reference value x, x.equals(x) should return true.
     * * Symmetric: for any non-null reference values x and y, x.equals(y) should retur
n true if and only if y.equals(x) returns true.
     * * Transitive:  for any non-null reference values x, y, and z, if x.equals(y) ret
urns true and y.equals(z) returns true, then x.equals(z) should return true
     * * Consistent:  for any non-null reference values x and y, multiple invocations o
f x.equals(y) consistently return true or consistently return false, provided no inform
ation used in equals comparisons on the objects is modified.
     *
     * Note that the `==` operator in Kotlin code is translated into a call to [equals]
 when objects on both sides of the
     * operator are not null.
     */
    public open operator fun equals(other: Any?): Boolean

    /**
     * Returns a hash code value for the object.  The general contract of hashCode is:
     *
     * * Whenever it is invoked on the same object more than once, the hashCode method
must consistently return the same integer, provided no information used in equals compa
risons on the object is modified.
     * * If two objects are equal according to the equals() method, then calling the ha
shCode method on each of the two objects must produce the same integer result.
     */
    public open fun hashCode(): Int

    /**
     * Returns a string representation of the object.
     */
    public open fun toString(): String
}
```

4.3.1 对象相等性

从Any的源码注释中，我们可以看到，判断两个对象是否相等，需要满足以下条件：

> 自反性：对于任何非空引用值x，x.equals（x）应返回true。
>
> 对称性：对于任何非空引用值x和y，x.equals（y）应返回true当且仅当y.equals（x）返回true。
>
> 传递性：对于任何非空引用值x，y，z，如果x.equals（y）返回true，y.equals（z）返回true，那么x.equals（z）应返回true
>
> 一致性：对于任何非空引用值x和y，多次调用x.equals（y）始终返回true或者始终返回false。

另外，在Kotlin中，操作符`==`会被编译器翻译成调用`equals()`函数。

4.4 基本类型（Primitive Types）

本节我们来探讨学习：Kotlin的基础类型：数字、字符、布尔和数组等。

我们知道Java的类型分成两种：一种是基本类型，一种是引用类型。它们的本质区别是：

> 基本类型是在堆栈处分配空间存“值”，而引用类型是在堆里面分配空间存“值”。

Java的基本类型有： byte、int、short、long、float、double、char、boolean，这些类都有对应的装箱类（引用类型）。

另外，void也可以算是一种特殊的基本类型，它也有一个装箱类`Void`（跟我们后文讲到的Unit、Nothing相关）。因为，Void是不能new出来的，也就是不能在堆里面分配空间存对应的值。所以，Void是一开始在堆栈处分配好空间。所以，将Void归成基本类型。

在Kotlin中，一切皆是对象。所有类型都是引用类型。没有类似Java中的基本类型。但是，可以把Kotlin中对应的这几种基本数据类型，理解为Java的基本类型的装箱类。

Integer.java

```
public final class Integer extends Number implements Comparable<Integer> {
    /**
     * A constant holding the minimum value an {@code int} can
     * have, -2<sup>31</sup>.
     */
    @Native public static final int   MIN_VALUE = 0x80000000;

    /**
     * A constant holding the maximum value an {@code int} can
     * have, 2<sup>31</sup>-1.
     */
    @Native public static final int   MAX_VALUE = 0x7fffffff;
```

```
    /**
     * The {@code Class} instance representing the primitive type
     * {@code int}.
     *
     * @since   JDK1.1
     */
    @SuppressWarnings("unchecked")
    public static final Class<Integer>  TYPE = (Class<Integer>) Class.getPrimitiveClass
("int");

    ...

}
```

Kotlin中的 `Int` 类型：

```
public class Int private constructor() : Number(), Comparable<Int> {
    companion object {
        /**
         * A constant holding the minimum value an instance of Int can have.
         */
        public const val MIN_VALUE: Int = -2147483648

        /**
         * A constant holding the maximum value an instance of Int can have.
         */
        public const val MAX_VALUE: Int = 2147483647
    }
    ...
}
```

我们通过Java的Integer封装类，跟Kotlin的Int类的定义可以看出两者的思想上的同源性。

Kotlin的基本类型的类图结构如下图所示

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-f5113b7f8d33ff37.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

4.4.1 数字（Number）类型

Kotlin 提供了如下的内置类型来表示数字（与 Java 很相近）：

类型	宽度（Bit）
Double	64
Float	32
Long	64

Int	32
Short	16
Byte	8

从上面的Kotlin的基本类型的类的结构图，我们可以看出这些内置的数据类型，都继承了 `Number` 和 `Comparable` 类。例如， `Byte` 类型的声明：

```
public class Byte private constructor() : Number(), Comparable<Byte> {
    ...
}
```

Kotlin 的数字类型跟 Java基本相同。有一点不同的是，Kotlin对于数字没有隐式拓宽转换（如 Java 中 `int` 可以隐式转换为 `long` ）。

注意在 Kotlin 中字符 `Char` 不是数字。这些基本数据类型，会在运行时自动优化为Java的double、float、long、int、short、byte。

字面常量值（literal constant values）

数值常量字面值有以下几种:

- 十进制: `123`
- Long 类型用大写 `L` 标记: `123L`
- 十六进制: `0x0F`
- 二进制: `0b00001011`

代码示例：

```
>>> 123
123
>>> 123::class
class kotlin.Int
>>> 123::class.java
int
>>> 123L
123
>>> 123L::class
class kotlin.Long
>>> 123L::class.java
long

>>> val b:Byte=128
error: the integer literal does not conform to the expected type Byte
val b:Byte=128
           ^
```

```
>>> val b:Byte=127
>>> b::class
class kotlin.Byte
>>> b::class.java
byte

>>> 0x0f
15
>>> 0x0F
15
>>> 0b1000
8
```

同样的，当我们赋值超过变量的类型的取值范围时，编译器会直接抛错。

注意: 不支持八进制

Kotlin 同样支持浮点数的常规表示方法:

- 默认 double：`123.5` 、`123.5e10`
- Float 用 `f` 或者 `F` 标记: `123.5f`

代码示例：

```
>>> 1234.5
1234.5
>>> 1234.5::class
class kotlin.Double
>>> 1234.5::class.java
double
>>> 12.3e10
1.23E11
>>> 12.3e10::class
class kotlin.Double
>>> 456.7f
456.7
>>> 456.7f::class
class kotlin.Float
>>> 456.7f::class.java
float
```

我们也可以使用数字字面值中的下划线（自 1.1 起），使数字常量更易读：

```
>>> 1_000_000
1000000
>>> 1234_5678_9012_3456L
```

```
1234567890123456
>>> 0xFF_EC_DE_5E
4293713502
>>> 0b11010010_01101001_10010100_10010010
3530134674
```

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 `Int?` ）或泛型。 后者情况下会把数字装箱。

显式转换

由于不同的表示方式，值范围较小类型并不是较大类型的子类型，是不能隐式转换的。

代码示例：

```
>>> val a: Int? = 1
>>> val b: Long? = a
error: type mismatch: inferred type is Int? but Long? was expected
val b: Long? = a
               ^


>>> val b: Byte = 1
>>> val i: Int = b
error: type mismatch: inferred type is Byte but Int was expected
val i: Int = b
             ^
```

这意味着在不进行显式转换的情况下我们不能把 `Int` 型值赋给一个 `Long` 变量。也不能把 `Byte` 型值赋给一个 `Int` 变量。

我们可以显式转换来拓宽数字

```
>>> val i: Int = b.toInt() // OK: 显式拓宽
```

每个数字类型都继承Number抽象类，其中定义了如下的转换函数:

```
toDouble(): Double
toFloat(): Float
toLong(): Long
toInt(): Int
toChar(): Char
toShort(): Short
toByte(): Byte
```

所以，在数字之间的转换，我们直接调用上面的这些转换函数即可。

运算符 `+` 重载

缺乏隐式类型转换并不显著，因为类型会从上下文推断出来，而算术运算会有重载做适当转换，例如：

```
val l = 1L + 3 // Long + Int => Long
```

这个是通过运算符 `+` 重载实现的。我们可以在Long类的源代码中看到这个 `plus` 运算符函数的定义：

```
    public operator fun plus(other: Byte): Long
    public operator fun plus(other: Short): Long
    public operator fun plus(other: Int): Long
    public operator fun plus(other: Long): Long
    public operator fun plus(other: Float): Float
    public operator fun plus(other: Double): Double
```

也就是说，编译器会把 `1L + 3` 翻译成 `1L.plus(3)`，然后这个传入的参数类型必须是Byte、Short、Int、Long、Float、Double中的一种。例如，我们传入一个字符 `Char` 参数，编译器就会直接抛错：

```
>>> 'a'
a
>>> 'a'::class
class kotlin.Char
>>> 'a'::class.java
char
>>> 1L+'a'
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Long defined in kotlin.Long
public final operator fun plus(other: Double): Double defined in kotlin.Long
public final operator fun plus(other: Float): Float defined in kotlin.Long
public final operator fun plus(other: Int): Long defined in kotlin.Long
public final operator fun plus(other: Long): Long defined in kotlin.Long
public final operator fun plus(other: Short): Long defined in kotlin.Long
1L+'a'
  ^
```

运算

Kotlin支持数字运算的标准集，运算被定义为相应的类成员（但编译器会将函数调用优化为相应的指令）。

对于位运算，没有特殊字符来表示，而只可用中缀方式调用命名函数（ `infix fun` ），例如:

```
val x = (1 shl 2) and 0x000FF000
```

这是完整的位运算列表（只用于 `Int` 和 `Long` ）：

- `shl(bits)` – 有符号左移 (Java 的 `<<`)
- `shr(bits)` – 有符号右移 (Java 的 `>>`)
- `ushr(bits)` – 无符号右移 (Java 的 `>>>`)
- `and(bits)` – 位与
- `or(bits)` – 位或
- `xor(bits)` – 位异或
- `inv()` – 位非

4.4.2 Char: 字符(Character)类型与转义符（Escape character）

字符用 `Char` 类型表示。它们不能直接当作数字

```
fun check(c: Char) {
    if (c == 1) { // 错误：类型不兼容
        // ……
    }
}
```

字符字面值用 单引号 括起来: `'1'` 。 特殊字符可以用反斜杠转义。

Kotlin支持如下转义字符：

```
\t
\b
\n
\r
\`
\"
\\
\$
```

编码其他字符要用 Unicode 转义序列语法，例如： `'\uFF00'` 。

Char类的函数接口定义如下：

```
public class Char private constructor() : Comparable<Char> {
    /**
```

```
     * Compares this value with the specified value for order.
     * Returns zero if this value is equal to the specified other value, a negative num
ber if it's less than other,
     * or a positive number if it's greater than other.
     */
    public override fun compareTo(other: Char): Int

    /** Adds the other Int value to this value resulting a Char. */
    public operator fun plus(other: Int): Char

    /** Subtracts the other Char value from this value resulting an Int. */
    public operator fun minus(other: Char): Int
    /** Subtracts the other Int value from this value resulting a Char. */
    public operator fun minus(other: Int): Char

    /** Increments this value. */
    public operator fun inc(): Char
    /** Decrements this value. */
    public operator fun dec(): Char

    /** Creates a range from this value to the specified [other] value. */
    public operator fun rangeTo(other: Char): CharRange

    /** Returns the value of this character as a `Byte`. */
    public fun toByte(): Byte
    /** Returns the value of this character as a `Char`. */
    public fun toChar(): Char
    /** Returns the value of this character as a `Short`. */
    public fun toShort(): Short
    /** Returns the value of this character as a `Int`. */
    public fun toInt(): Int
    /** Returns the value of this character as a `Long`. */
    public fun toLong(): Long
    /** Returns the value of this character as a `Float`. */
    public fun toFloat(): Float
    /** Returns the value of this character as a `Double`. */
    public fun toDouble(): Double


}
```

我们来用代码示例这些函数的使用：

如果两个字符相等：

```
>>> 'a'.compareTo('a')
0
```

如果两个字符不相等：

```
>>> 'a'.compareTo('b')
-1
>>> 'a'.compareTo('c')
-1
>>> 'b'.compareTo('a')
1
>>> 'c'.compareTo('a')
1
```

Char字符只重载了加上 `Int` 类型的数字的 `+` 运算符：

```
>>> 'a'+1
b

>>> 'a'+1L
error: the integer literal does not conform to the expected type Int
'a'+1L
```

所以，当我们把一个 `Char` 类型值和不是 `Int` 类型的值相加，就报错了。

相减：

```
>>> 'a'-1
`
>>> 'c'-'a'
2
```

自增计算：

```
>>> var a='a'
>>> val b=a++
>>> a
b
>>> b
a
>>> val c=++a
>>> c
c
```

我们不能在字符的字面量上直接使用 `++`：

```
>>> 'a'++
error: variable expected
```

```
'a'++
^

>>> ++'a'
error: variable expected
++'a'
  ^
```

范围

```
>>> 'a'.rangeTo('z')
a..z
>>> for(c in 'a'..'z') {print(c)}

abcdefghijklmnopqrstuvwxyz
```

`Char` 的显式类型转换函数如下：

```
    /** Returns the value of this character as a `Byte`. */
    public fun toByte(): Byte
    /** Returns the value of this character as a `Char`. */
    public fun toChar(): Char
    /** Returns the value of this character as a `Short`. */
    public fun toShort(): Short
    /** Returns the value of this character as a `Int`. */
    public fun toInt(): Int
    /** Returns the value of this character as a `Long`. */
    public fun toLong(): Long
    /** Returns the value of this character as a `Float`. */
    public fun toFloat(): Float
    /** Returns the value of this character as a `Double`. */
    public fun toDouble(): Double
```

例如，我们显式把字符转换为 `Int` 数字：

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // 显式转换为数字
}
```

测试代码：

```
>>> decimalDigitValue('a')
java.lang.IllegalArgumentException: Out of range
    at Line24.decimalDigitValue(Unknown Source)

>>> decimalDigitValue('1')
1
```

4.4.3 Boolean: 布尔类型

Kotlin的布尔类型用 `Boolean` 类来表示，它有两个值： `true` 和 `false` 。

```
>>> true::class
class kotlin.Boolean
>>> true::class.java
boolean
```

对应Java中的 `boolean` 类型。

其源码定义如下：

```
package kotlin

/**
 * Represents a value which is either `true` or `false`. On the JVM, non-nullable value
s of this type are
 * represented as values of the primitive type `boolean`.
 */
public class Boolean private constructor() : Comparable<Boolean> {
    /**
     * Returns the inverse of this boolean.
     */
    public operator fun not(): Boolean

    /**
     * Performs a logical `and` operation between this Boolean and the [other] one.
     */
    public infix fun and(other: Boolean): Boolean

    /**
     * Performs a logical `or` operation between this Boolean and the [other] one.
     */
    public infix fun or(other: Boolean): Boolean

    /**
     * Performs a logical `xor` operation between this Boolean and the [other] one.
     */
```

```
    public infix fun xor(other: Boolean): Boolean

    public override fun compareTo(other: Boolean): Int
}
```

从上面我们可以看出，Boolean类的内置的布尔运算有：

- `!` 逻辑非 `not()`
- `&&` 短路逻辑与 `and()`
- `||` 短路逻辑或 `or()`
- `xor` 异或(相同false，不同true)

另外，`Boolean` 还继承实现了 `Comparable` 的 `compareTo()` 函数。

代码示例：

```
>>> !true
false
>>> true.not()
false
>>> true && true
true
>>> true.and(false)
false
>>> true || false
true
>>> false.or(false)
false
>>> true xor true
false
>>> true xor false
true
>>> false xor false
false
>>> true > false
true
>>> true < false
false
>>> true.compareTo(false)
1
>>> true.compareTo(false)
1
>>> true.compareTo(true)
0
>>> false.compareTo(true)
-1
```

4.4.4 String: 字符串类型

Kotlin的字符串用 `String` 类型表示。对应Java中的 `java.lang.String` 。字符串是不可变的。

```
>>> "abc"::class
class kotlin.String
>>> "abc"::class.java
class java.lang.String
```

另外，在Kotlin中，String同样是final不可继承的。

代码示例：

```
>>> class MyString:String
error: this type is final, so it cannot be inherited from
class MyString:String
               ^
```

索引运算符 `s[i]`

字符串的元素——字符可以使用索引运算符 `s[i]` 来访问。

```
>>> val s="abc"
>>> s
abc
>>> s[0]
a
```

当我们下标越界时，会抛越界错误：

```
>>> s[-1]
java.lang.StringIndexOutOfBoundsException: String index out of range: -1
    at java.lang.String.charAt(String.java:646)

>>> s[3]
java.lang.StringIndexOutOfBoundsException: String index out of range: 3
    at java.lang.String.charAt(String.java:646)
```

从出错信息，我们可以看出，索引运算符 `s[i]` 会被翻译成 `java.lang.String.charAt()` , 背后调用的是Java的String类。其调用的方法是：

```
    public char charAt(int index) {
        if ((index < 0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
        }
```

```
        return value[index];
    }
```

`for` 循环迭代字符串

我们可以用 `for` 循环迭代字符串:

```
>>> for(c in "abc") { println(c)  }
a
b
c
```

关于字符串 `String` 类的完整的操作方法，我们可以看下源码：

```
public class String : Comparable<String>, CharSequence {
    companion object {}

    /**
     * Returns a string obtained by concatenating this string with the string represent
ation of the given [other] object.
     */
    public operator fun plus(other: Any?): String

    public override val length: Int

    public override fun get(index: Int): Char

    public override fun subSequence(startIndex: Int, endIndex: Int): CharSequence

    public override fun compareTo(other: String): Int
}
```

类似的，字符串有一个 `length` 属性：

```
>>> "abc".length
3
```

重载 `+` 操作符

字符串类重载了 `+` 操作符，作用对象可以是任何对象，包括空引用：

```
>>> "abc".plus(true)
abctrue
>>> "abc"+false
```

```
abcfalse
>>> "abc"+1
abc1
>>> "abc"+1.20
abc1.2
>>> "abc"+100L
abc100
>>> "abc"+"cdef"
abccdef
>>> "abc"+null
abcnull
>>> "abc"+'z'
abcz
>>> "abc"+arrayOf(1,2,3,4,5)
abc[Ljava.lang.Integer;@3d6f0054
```

截取字符串的子串：

```
>>> "abc".subSequence(0,1)
a
>>> "abc".subSequence(0,2)
ab
>>> "abc".subSequence(0,3)
abc
>>> "abc".subSequence(0,4)
java.lang.StringIndexOutOfBoundsException: String index out of range: 4
    at java.lang.String.substring(String.java:1951)
    at java.lang.String.subSequence(String.java:1991)
```

字符串字面值

字符串的字面值，可以包含原生字符串可以包含换行和任意文本，也可以是带有转义字符（Escape Charactor）的转义字符串。

```
>>> val s = "Hello,World!\n\n\n"
>>> s
Hello,World!


>>>
```

转义采用传统的反斜杠方式。

原生字符串 使用三个引号（ `"""` ）分界符括起来，内部没有转义并且可以包含换行和任何其他字符:

```
>>> val text = """
...     for (c in "abc")
...         print(c)
... """
>>> text

    for (c in "foo")
        print(c)

>>>
```

另外，在 `package kotlin.text` 下面的Indent.kt代码中，Kotlin还定义了 `String` 类的扩展函数：

```
fun String.trimMargin(marginPrefix: String = "|"): String
fun String.trimIndent(): String
```

我们可以使用 `trimMargin()` 、 `trimIndent()` 裁剪函数来去除前导空格。可以看出， `trimMargin()` 函数默认使用 `"|"` 来作为边界字符：

```
>>> val text = """
... |理论是你知道是这样，但它却不好用。
... |实践是它很好用，但你不知道是为什么。
... |程序员将理论和实践结合到一起：
... |既不好用，也不知道是为什么。
...     """
>>> text.trimMargin()
理论是你知道是这样，但它却不好用。
实践是它很好用，但你不知道是为什么。
程序员将理论和实践结合到一起：
既不好用，也不知道是为什么。
```

默认 `|` 用作边界前缀，但你可以选择其他字符并作为参数传入，比如 `trimMargin(">")` 。

`trimIndent()` 函数，则是把字符串行的左边空白对齐切割：

```
>>> val text="""
...             Hello
...                   World!
... """
>>> text.trimIndent()
Hello
      World!
>>> val text="""
...            Hello,
```

```
...         World!
... """
>>> text.trimIndent()
    Hello,
World!
```

字符串模板

字符串可以包含模板表达式，即一些小段代码，会求值并把结果合并到字符串中。模板表达式以美元符（ `$` ）开头，由一个简单的名字构成:

```
>>> val h=100
>>> val str = "A hundred is $h"
>>> str
A hundred is 100
```

或者用花括号扩起来的任意表达式:

```
>>> val s = "abc"
>>> val str = "$s.length is ${s.length}"
>>> str
abc.length is 3
```

原生字符串和转义字符串内部都支持模板。

```
>>> val price=9.9
>>> val str="""Price is $$price"""
>>> str
Price is $9.9
>>> val str="Price is $$price"
>>> str
Price is $9.9

>>> val quantity=100
>>> val str="Quantity is $quantity"
>>> str
Quantity is 100
>>> val str="""Quantity is $quantity"""
>>> str
Quantity is 100
```

4.4.5 Array: 数组类型

数组在 Kotlin 中使用 `Array` 类来表示，它定义了 `get` 和 `set` 函数（映射到重载运算符 `[]` ）和 `size` 属性，以及一个用于变量数组的 `iterator()` 函数：

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit
    operator fun iterator(): Iterator<T>
    // ……
}
```

我们可以使用函数 `arrayOf()` 来创建一个数组并传递元素值给它。这个函数签名如下：

```
public inline fun <reified @PureReifiable T> arrayOf(vararg elements: T): Array<T>
```

其中，vararg表示是一个参数个数是一个变量。

例如， `arrayOf(1, 2, 3)` 创建了 array [1, 2, 3]：

```
>>> arrayOf(1,2,3)
[Ljava.lang.Integer;@4a37191a
>>> arrayOf(1,2,3)::class
class kotlin.Array
>>> arrayOf(1,2,3)::class.java
class [Ljava.lang.Integer;
```

另外，Kotlin还允许不同类型元素放到一个数组中，例如：

```
>>> val arr = arrayOf(1,"2",true)
>>> arr
[Ljava.lang.Object;@61af1510
>>> arr.forEach{ println(it) }
1
2
true
>>> arr.forEach{ println(it::class) }
class kotlin.Int
class kotlin.String
class kotlin.Boolean
```

Kotlin自动把这个数组元素的类型升级为 `java.lang.Object` ， 同时，由于Kotlin拥有的类型推断的功能，我们仍然可以看到每个数组元素对应的各自的类型。

函数 `arrayOfNulls()` 可以用于创建一个指定大小、元素都为空的数组。这个特殊的空数组在创建的时候，我们需要指定元素的类型。如果不指定，直接按照下面这样写，会报错：

```
>>> arrayOfNulls(10)
error: type inference failed: Not enough information to infer parameter T in fun <reifi
ed T> arrayOfNulls(size: Int): Array<T?>
Please specify it explicitly.

arrayOfNulls(10)
^
```

也就是说，我们要指定

```
>>> arrayOfNulls<Int>(10)
[Ljava.lang.Integer;@77c10a5f
>>> arrayOfNulls<Int>(10).forEach{println(it)}
null
null
null
null
null
null
null
null
null
null
```

数组 `Array` 类，还提供了一个构造函数：

```
public inline constructor(size: Int, init: (Int) -> T)
```

第1个参数是数组大小，第2个参数是一个初始化函数类型的参数（关于函数类型，我们将在后面章节介绍）。

代码示例：

```
>>> val square = Array(10, { i -> (i*i)})
>>> square
[Ljava.lang.Integer;@6f9e08d4
>>> square.forEach{ println(it) }
0
1
4
9
```

```
16
25
36
49
64
81
```

如上所述，`[]` 运算符代表调用成员函数 `get()` 和 `set()` 。代码示例：

```
>>> square[3]
9

>>> square[3]=1000
>>> square.forEach{ println(it) }
0
1
4
1000
16
25
36
49
64
81
```

与 Java 不同的是，Kotlin 中数组不是型变的（invariant）。Kotlin中，我们不能把 `Array<String>` 赋值给 `Array<Any>` 。这地方Kotlin类型检查的限制强于Java的数组类型。

代码示例：

```
>>> val arrstr = arrayOf<String>("1","2","3")
>>> arrstr
[Ljava.lang.String;@39374689
>>> var arrany = arrayOf<Any>(Any(),Any(),Any())
>>> arrany
[Ljava.lang.Object;@156324b
>>> arrany = arrstr
error: type mismatch: inferred type is Array<String> but Array<Any> was expected
arrany = arrstr
         ^
```

原生数组类型

Kotlin 也有无装箱开销的专门的类来表示原生类型数组。这些原生数组类如下：

```
BooleanArray
```

```
ByteArray
CharArray
ShortArray
IntArray
LongArray
FloatArray
DoubleArray
BooleanArray
```

这些类和 `Array` 并没有继承关系，但它们有同样的函数和属性集。它们也都有相应的工厂方法:

```
/**
 * Returns an array containing the specified [Double] numbers.
 */
public fun doubleArrayOf(vararg elements: Double): DoubleArray

/**
 * Returns an array containing the specified [Float] numbers.
 */
public fun floatArrayOf(vararg elements: Float): FloatArray

/**
 * Returns an array containing the specified [Long] numbers.
 */
public fun longArrayOf(vararg elements: Long): LongArray

/**
 * Returns an array containing the specified [Int] numbers.
 */
public fun intArrayOf(vararg elements: Int): IntArray

/**
 * Returns an array containing the specified characters.
 */
public fun charArrayOf(vararg elements: Char): CharArray

/**
 * Returns an array containing the specified [Short] numbers.
 */
public fun shortArrayOf(vararg elements: Short): ShortArray

/**
 * Returns an array containing the specified [Byte] numbers.
 */
public fun byteArrayOf(vararg elements: Byte): ByteArray
```

```
/**
 * Returns an array containing the specified boolean values.
 */
public fun booleanArrayOf(vararg elements: Boolean): BooleanArray
```

代码示例：

```
>>> val x: IntArray = intArrayOf(1, 2, 3)
>>> x[0]
1
```

4.5 `Any?` 可空类型（Nullable Types）

可空类型是Kotlin类型系统的一个特性，主要是为了解决Java中的令人头疼的 NullPointerException 问题。

我们知道，在Java中如果一个变量可以是 `null` ，来那么使用它调用一个方法就是不安全的，因为它会导致： `NullPointerException` 。

Kotlin把可空性（nullability）作为类型系统的一部分，Kotlin编译器可以直接在编译过程中发现许多可能的错误，并减少在运行时抛出异常的可能性。

Kotlin的类型系统和Java相比，首要的区别就是Kotlin对可空类型的显式支持。

在本节中，我们将讨论Kotlin中的可空类型。

4.5.1 null 是什么

对于Java程序员来说，null是令人头痛的东西。我们时常会受到空指针异常（NPE）的骚扰。就连Java的发明者都承认这是他的一项巨大失误。Java为什么要保留null呢？null出现有一段时间了，并且我认为Java发明者知道null与它解决的问题相比带来了更多的麻烦，但是null仍然陪伴着Java。

我们通常把 `null` 理解为编程语言中定义特殊的 `0` ，把我们初始化的指针指向它，以防止 "野指针" 的恶果。在Java中， `null` 是任何引用类型的默认值，不严格的说是所有Object类型的默认值。

这里的 `null` 既不是对象也不是一种类型，它仅是一种特殊的值，我们可以将其赋予任何引用类型，也可以将 `null` 转化成任何类型。在编译和运行时期，将 `null` 强制转换成任何引用类型都是可行的，在运行时期都不会抛出空指针异常。注意，这里指的是任何Java的引用类型。在遇到基本类型 `int` `long` `float` `double` `short` `byte` 等的时候，情况就不一样了。而且还是个坑。编译器不会报错，但是运行时会抛NPE。空指针异常。这是Java中的自动拆箱导致的。代码示例：

```
Integer nullInt = null; // this is ok
int anotherInt = nullInt; // 编译器允许这么赋值，但是在运行时抛 NullPointerException
```

所以，我们写Java代码的时候，要时刻注意这一点：Integer的默认值是null而不是0。当把null值传递给一个int型变量的时候，Java的自动装箱将会返回空指针异常。

4.5.2 Kotlin中的 `null`

在Kotlin中，针对Java中的 `null` 的杂乱局面，进行了整顿，作了清晰的界定，并在编译器级别强制规范了可空null变量类型的使用。

我们来看一下Kotlin中关于 `null` 的一些有趣的运算。

`null` 跟 `null` 是相等的：

```
>>> null==null
true
>>> null!=null
false
```

`null` 这个值比较特殊， `null` 不是 `Any` 类型

```
>>> null is Any
false
```

但是， `null` 是 `Any?` 类型：

```
>>> null is Any?
true
```

我们来看看 `null` 对应的类型到底是什么：

```
>>> var a=null
>>> a
null
>>> a=1
error: the integer literal does not conform to the expected type Nothing?
a=1
  ^
```

从报错信息我们可以看出， `null` 的类型是 `Nothing?` 。关于 `Nothing?` 我们将会在下一小节中介绍。

我们可以对 `null` 进行加法运算：

```
>>> "1"+null
1null
```

```
>>> null+20
null20
```

对应的重载运算符的函数定义在 `kotlin/Library.kt` 里面：

```
package kotlin

import kotlin.internal.PureReifiable

/**
 * Returns a string representation of the object. Can be called with a null receiver, i
n which case
 * it returns the string "null".
 */
public fun Any?.toString(): String

/**
 * Concatenates this string with the string representation of the given [other] object.
 If either the receiver
 * or the [other] object are null, they are represented as the string "null".
 */
public operator fun String?.plus(other: Any?): String

...
```

但是，反过来就不行了：

```
>>> 1+null
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+null
 ^
```

这是因为Int没有重载传入 `null` 参数的 `plus()` 函数。

4.5.3 可空类型 `String?` 与安全调用 `?.`

我们来看一个例子。下面是计算字符串长度的简单Java方法：

```
public static int getLength1(String str) {
        return str.length();
    }
```

我们已经习惯了在这样的Java代码中，加上这样的空判断处理：

```
    public static int getLength2(String str) throws Exception {
        if (null == str) {
            throw new Exception("str is null");
        }

        return str.length();
    }
```

而在Kotlin中，当我们同样写一个可能为 `null` 参数的函数时：

```
fun getLength1(str: String): Int {
        return str.length
    }
```

当我们传入一个 `null` 参数时:

```

    @Test fun testGetLength1() {
        val StringUtilKt = StringUtilKt()
        StringUtilKt.getLength1(null)
    }
```

编译器就直接编译失败：

```
e: /Users/jack/easykotlin/chapter4_type_system/src/test/kotlin/com/easy/kotlin/StringUt
ilKtTest.kt: (15, 33): Null can not be a value of a non-null type String
:compileTestKotlin FAILED
FAILURE: Build failed with an exception.
* What went wrong:
Execution failed for task ':compileTestKotlin'.
> Compilation error. See log for more details
```

如果我们使用IDEA，会在编码时就直接提示错误了：

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-6313b8ec066c20c5.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

这样通过编译时强制排除空指针的错误，大大减少了出现NPE的可能。

另外，如果我们确实需要传入一个可空的参数，我们可以使用可空类型 `String?` 来声明一个可以指向空指针的变量。

可空类型可以用来标记任何一个变量，来表明这个变量是可空的（Nullable）。例如： `Char?` ， `Int?` ， `MineType?` （自定义的类型）等等。

我们用示例代码来更加简洁的说明：

```
>>> var x:String="x"
>>> x=null
error: null can not be a value of a non-null type String
x=null
  ^

>>> var y:String?="y"
>>> y=null
>>> y
null
```

我们可以看出：普通 `String` 类型，是不允许指向 `null` 的；而可空 `String?` 类可以指向 `null` 。

下面我们来尝试使用一个可空变量来调用函数：

```
>>> fun getLength2(str: String?): Int? = str.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiv
er of type String?
fun getLength2(str: String?): Int? = str.length
                                        ^
```

编译器直接报错，告诉我们，变量 `str: String?` 是可空的类型，调用只能通过安全调用 `?.` 或者非空断言调用 `!!.` 。

另外，如果不需要捕获异常来处理，我们可以使用Kotlin里面的安全调用符 `?.` 。

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-f285ebad3cf43c11.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

代码示例：

```
    fun getLength2(str: String?): Int? {
        return str?.length
     }
```

测试代码：

```
     @Test fun testGetLength2() {
         val StringUtilKt = StringUtilKt()
```

```
        println(StringUtilKt.getLength2(null)) //null
        Assert.assertTrue(3 == StringUtilKt.getLength2("abc"))
    }
```

我们可以看出，当我们使用安全调用 `?.` ，代码安静的执行输出了 `null` 。

如果，我们确实想写一个出现空指针异常的代码，那就使用可能出现空指针的断言调用符 `!!.` 。

代码示例：

```
    fun getLength3(str: String?): Int? {
        return str!!.length
    }
```

测试代码：

```
    @Test fun testGetLength3() {
        val StringUtilKt = StringUtilKt()
        println(StringUtilKt.getLength3(null))
        Assert.assertTrue(3 == StringUtilKt.getLength3("abc"))
    }
```

上面的代码就跟Java里面差不多了，运行会直接抛出空指针异常：

```
kotlin.KotlinNullPointerException
    at com.easy.kotlin.StringUtilKt.getLength3(StringUtilKt.kt:16)
    at com.easy.kotlin.StringUtilKtTest.testGetLength3(StringUtilKtTest.kt:28)
```

这里的 `KotlinNullPointerException` 是KotlinNullPointerException.java代码，继承了Java中的 `java.lang.NullPointerException` ，它的源代码如下：

```
package kotlin;

public class KotlinNullPointerException extends NullPointerException {
    public KotlinNullPointerException() {
    }

    public KotlinNullPointerException(String message) {
        super(message);
    }
}
```

另外，如果异常需要捕获到进行特殊处理的场景，在Kotlin中仍然使用 `try ... catch` 捕获并处理异常。

4.5.4 可空性的实现原理

我们来看一段Kotlin的可空类型的示例代码如下：

```
fun testNullable1(x: String, y: String?): Int {
    return x.length
}

fun testNullable2(x: String, y: String?): Int? {
    return y?.length
}

fun testNullable3(x: String, y: String?): Int? {
    return y!!.length
}
```

我们来使用IDEA的Kotlin插件来看下可空类型的安全调用的等价Java代码。

打开IDEA的 `Tools > Kotlin > Show Kotlin Bytecode`

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-1e42eef13264b596.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

然后，点击 `Decompile` ，我们可以得到反编译的Java代码

```
public final class NullableTypesKt {
   public static final int testNullable1(@NotNull String x, @Nullable String y) {
      Intrinsics.checkParameterIsNotNull(x, "x");
      return x.length();
   }

   @Nullable
   public static final Integer testNullable2(@NotNull String x, @Nullable String y) {
      Intrinsics.checkParameterIsNotNull(x, "x");
      return y != null?Integer.valueOf(y.length()):null;
   }

   @Nullable
   public static final Integer testNullable3(@NotNull String x, @Nullable String y) {
      Intrinsics.checkParameterIsNotNull(x, "x");
      if(y == null) {
         Intrinsics.throwNpe();
      }

      return Integer.valueOf(y.length());
   }
}
```

在不可空变量调用函数之前，都检查了是否为空, 使用的是 `kotlin.jvm.internal.Intrinsics` 这个Java类里面的 `checkParameterIsNotNull` 方法。如果是 `null` 就抛出异常：

```
    public static void checkParameterIsNotNull(Object value, String paramName) {
        if (value == null) {
            throwParameterIsNullException(paramName);
        }
    }
```

同时，我们可以看出在Kotlin中函数的入参声明

```
fun testNullable(x: String, y: String?)
```

反编译成等价的Java代码是

```
public static final void testNullable(@NotNull String x, @Nullable String y)
```

我们可以看出，这里使用注解 `@NotNull` 标注不可空的变量，使用注解 `@Nullable` 标注一个变量可空。

可空变量的安全调用符 `y?.length` 等价的Java代码就是：

```
y != null?Integer.valueOf(y.length()):null
```

可空变量的断言调用 `y!!.length` 等价的Java代码是：

```
if(y == null) {
    Intrinsics.throwNpe();
}
return Integer.valueOf(y.length());
```

4.5.5 可空类型层次体系

就像 `Any` 是在非空类型层次结构的根， `Any?` 是可空类型层次的根。 由于 `Any?` 是Any的超集，所以， `Any?` 是Kotlin的类型层次结构的最顶端。

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-869c3c563d7348fe.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

代码示例：

```

>>> 1 is Any
```

```
true

>>> 1 is Any?
true

>>> null is Any
false

>>> null is Any?
true



>>> Any() is Any?
true
```

4.6 kotlin.Unit类型

Kotlin也是面向表达式的语言。在Kotlin中所有控制流语句都是表达式（除了变量赋值、异常等）。

Kotlin中的 `Unit` 类型实现了与Java中的 `void` 一样的功能。不同的是，当一个函数没有返回值的时候，我们用 `Unit` 来表示这个特征，而不是 `null` 。

大多数时候，我们并不需要显式地返回 `Unit` ，或者声明一个函数的返回类型为 `Unit` 。编译器会推断出它。

代码示例：

```
>>> fun unitExample(){println("Hello,Unit")}
>>> val helloUnit = unitExample()
Hello,Unit
>>> helloUnit
kotlin.Unit
>>> println(helloUnit)
kotlin.Unit
```

下面几种写法是等价的：

```
@RunWith(JUnit4::class)
class UnitDemoTest {
    @Test fun testUnitDemo() {
        val ur1 = unitReturn1()
        println(ur1) // kotlin.Unit
        val ur2 = unitReturn2()
        println(ur2) // kotlin.Unit
        val ur3 = unitReturn3()
        println(ur3) // kotlin.Unit
    }
```

```
    fun unitReturn1() {

    }

    fun unitReturn2() {
        return Unit
    }

    fun unitReturn3(): Unit {
    }
}
```

总的来说，这个 `Unit` 类型并没有什么特别之处。它的源码是：

```
package kotlin

/**
 * The type with only one value: the Unit object. This type corresponds to the `void` t
ype in Java.
 */
public object Unit {
    override fun toString() = "kotlin.Unit"
}
```

跟任何其他类型一样，它的父类型是 `Any` 。如果是一个可空的 `Unit?` ，它的父类型是 `Any?` 。

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-fc66afa2edb41b58.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

4.7 kotlin.Nothing类型

Kotlin中没有类似Java和C中的函数没有返回值的标记 `void` ，但是拥有一个对应 `Nothing` 。在Java中，返回 `void` 的方法，其返回值 `void` 是无法被访问到的：

```
public class VoidDemo {
    public void voidDemo() {
        System.out.println("Hello,Void");
    }
}
```

测试代码：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class VoidDemoTest {
```

```
    @org.junit.Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        void v = voidDemo.voidDemo(); // 没有void变量类型，无法访问到void返回值
        System.out.println(voidDemo.voidDemo()); // error: 'void' type not allowed here
    }
}
```

在Java中， `void` 不能是变量的类型。也不能被当做值打印输出。但是，在Java中有个包装类 `Void` 是 `void` 的自动装箱类型。如果你想让一个方法返回类型 永远是 null 的话, 可以把返回类型置为这个大写的V的 `Void` 类型。

代码示例：

```
    public Void voidDemo() {
        System.out.println("Hello,Void");
        return null;
    }
```

测试代码：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class VoidDemoTest {
    @org.junit.Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        Void v = voidDemo.voidDemo(); // Hello,Void
        System.out.println(v); // null
    }
}
```

这个 `Void` 就是Kotlin中的 `Nothing?` 。它的唯一可被访问到的返回值也是 `null` 。

在Kotlin类型层次结构的最底层就是类型 `Nothing` 。

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-f98e1c823b289739.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

正如它的名字Nothing所暗示的， `Nothing` 是没有实例的类型。

代码示例：

```
>>> Nothing() is Any
error: cannot access '<init>': it is private in 'Nothing'
Nothing() is Any
^
```

注意：Unit与Nothing之间的区别: Unit类型表达式计算结果的返回类型是Unit。Nothing类型的表达式计算结果是永远不会返回的（跟Java 中的 `void` 相同）。

例如，throw关键字中断的表达式的计算，并抛出堆栈的功能。所以，一个 `throw Exception` 的代码就是返回 `Nothing` 的表达式。代码示例：

```
fun formatCell(value: Double): String =
    if (value.isNaN())
        throw IllegalArgumentException("$value is not a number")  // Nothing
    else
        value.toString()
```

再例如, Kotlin的标准库里面的 `exitProcess` 函数:

```
@file:kotlin.jvm.JvmName("ProcessKt")
@file:kotlin.jvm.JvmVersion
package kotlin.system

/**
 * Terminates the currently running Java Virtual Machine. The
 * argument serves as a status code; by convention, a nonzero status
 * code indicates abnormal termination.
 *
 * This method never returns normally.
 */
@kotlin.internal.InlineOnly
public inline fun exitProcess(status: Int): Nothing {
    System.exit(status)
    throw RuntimeException("System.exit returned normally, while it was supposed to hal
t JVM.")
}
```

Nothing?可以只包含一个值：null。代码示例：

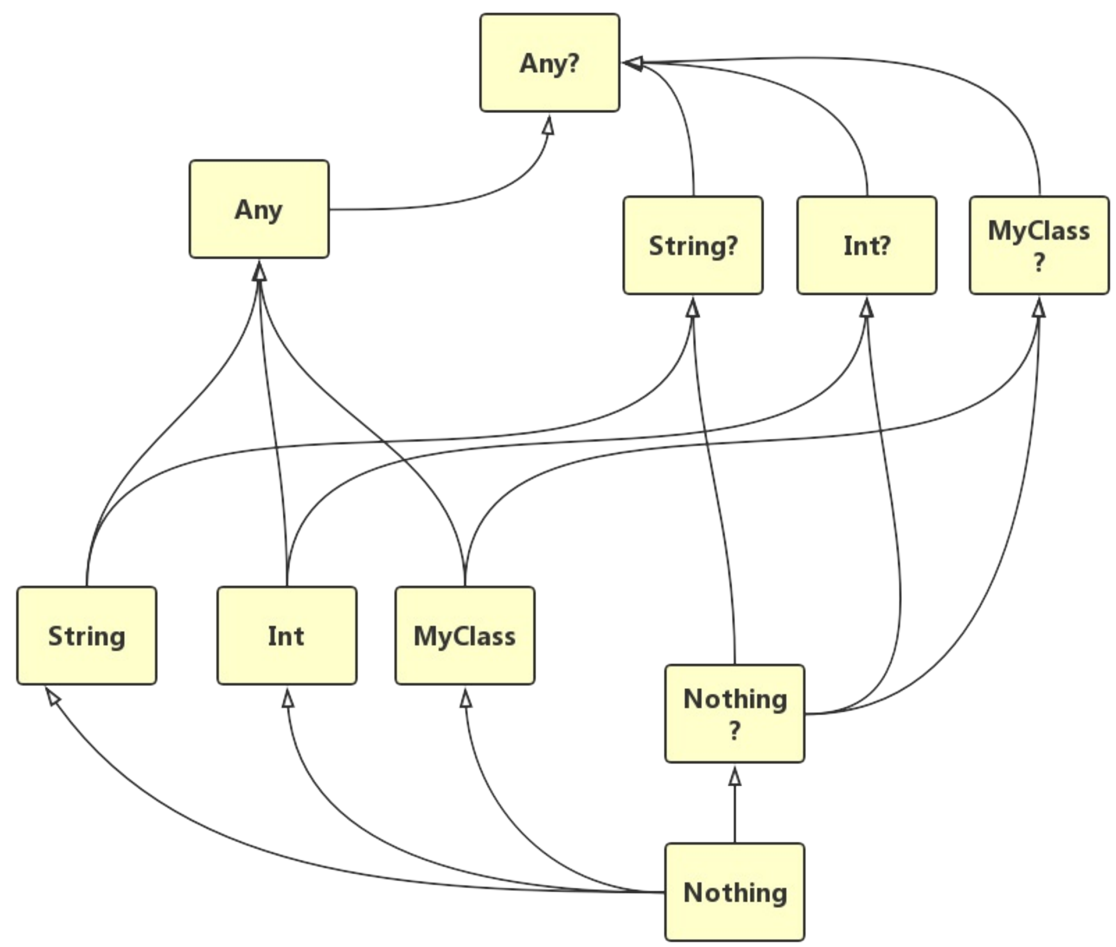
```
>>> var nul:Nothing?=null
>>> nul = 1
error: the integer literal does not conform to the expected type Nothing?
nul = 1
      ^

>>> nul = true
error: the boolean literal does not conform to the expected type Nothing?
nul = true
      ^

>>> nul = null
>>> nul
```

```
null
```

从上面的代码示例，我们可以看出： `Nothing?` 它唯一允许的值是 `null` ，被用作任何可空类型的空引用。

![Kotlin极简教程] (http://upload-images.jianshu.io/upload_images/1233356-5d70a737409a01ce.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

综上所述，我们可以看出Kotlin有一个简单而一致的类型系统。 `Any?` 是整个类型体系的顶部， `Nothing` 是底部。如下图所示：



4.8 类型检测与类型转换

4.8.1 is,!is运算符

is运算符可以检查对象是否与特定的类型兼容（“兼容”的意思是：此对象是该类型，或者派生于该类型）。

is运算符用来检查对象（变量）是否属于某数据类型（如Int、String、Boolean等）。C#里面也有这个运算符。

`is` 运算符类似Java的 `instanceof`：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        if ("abc" instanceof String) {
            println("abc is instanceof String");
        } else {
            println("abc is not instanceof String");
        }
    }

    void println(Object obj) {
        System.out.println(obj);
    }
}
```

在Kotlin中，我们可以在运行时通过使用 `is` 操作符或其否定形式 `!is` 来检查对象是否符合给定类型：

```
>>> "abc" is String
true
>>> "abc" !is String
false

>>> null is Any
false
>>> null is Any?
true
```

代码示例：

```
@RunWith(JUnit4::class)
class ASOperatorTest {
    @Test fun testAS() {
        val foo = Foo()
        val goo = Goo()
        println(foo is Foo) //true 自己
        println(goo is Foo)// 子类 is 父类 = true
        println(foo is Goo)//父类 is 子类 = false
        println(goo is Goo)//true 自己
    }
}
```

```
open class Foo
class Goo : Foo()
```

类型自动转换

在Java代码中，当我们使用 `str instanceof String` 来判断其值为 `true` 的时候，我们想使用str变量，还需要显式的强制转换类型：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        Object str = "abc";
        if (str instanceof String) {
            int len = ((String)str).length();  // 显式的强制转换类型为String
            println(str + " is instanceof String");
            println("Length: " + len);

        } else {
            println(str + " is not instanceof String");
        }

        boolean is = "1" instanceof String;
        println(is);
    }

    void println(Object obj) {
        System.out.println(obj);
    }
}
```

而大多数情况下，我们不需要在 Kotlin 中使用显式转换操作符，因为编译器跟踪不可变值的 is-检查，并在需要时自动插入（安全的）转换：

```
    @Test fun testIS() {
        val len = strlen("abc")
        println(len) // 3
        val lens = strlen(1)
        println(lens) // 1
    }

    fun strlen(ani: Any): Int {
        if (ani is String) {
            return ani.length
        } else if (ani is Number) {
```

```
            return ani.toString().length
        } else if (ani is Char) {
            return 1
        } else if (ani is Boolean) {
            return 1
        }


        print("Not A String")
        return -1
    }
```

4.8.2 as运算符

as运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，使用 `as?` 运算符就会返回值null。

代码示例：

```
>>> open class Foo
>>> class Goo:Foo()
>>> val foo = Foo()
>>> val goo = Goo()


>>> foo as Goo
java.lang.ClassCastException: Line69$Foo cannot be cast to Line71$Goo

>>> foo as? Goo
null

>>> goo as Foo
Line71$Goo@73dce0e6
```

我们可以看出，在Kotlin中，子类是禁止转换为父类型的。

按照Liskov替换原则，父类转换为子类是对OOP的严重违反，不提倡、也不建议。严格来说，父类是不能转换为子类的，子类包含了父类所有的方法和属性，而父类则未必具有和子类同样成员范围，所以这种转换是不被允许的，即便是两个具有父子关系的空类型，也是如此。

本章小结

在本章中，我们停下脚步，仔细深入地去探讨了Kotlin语言中最重要的部分之一的：类型系统。

与Java相比，Kotlin的类型系统更加简单一致，同时引入了一些新的特性，这些特性对于提高代码的安全性、可靠性至关重要。例如：可空类型和只读集合。关于只读集合类，我们将在下一章中介绍。

我们下一章的主题是：Kotlin的集合类和泛型。

本章示例代码工程：https://github.com/EasyKotlin/chapter4_type_system

参考资料

1.https://jetbrains.github.io/kotlin-spec/#_the_kotlin_nothing_type

2.http://natpryce.com/articles/000818.html

第5章 集合类

本章将介绍Kotlin标准库中的集合类，我们将了解到它是如何扩展的Java集合库，使得写代码更加简单容易。如果您熟悉Scala的集合库，您会发现Kotlin跟Scala集合类库的相似之处。

5.1 集合类是什么

5.1.2 集合类是一种数据结构

在讲 Kotlin 的集合类之前，为了更加深刻理解为什么要有集合类，以及集合类到底是怎么一回事，让我们先来简单回顾一下编程的本质：

> 数据结构 + 算法 （信息的逻辑结构及其基本操作）

我们使用计算机编程来解决一个具体问题时，大致需要经过下列几个步骤：

首先要从具体问题中抽象出一个适当的数学模型；然后设计一个解此数学模型的算法（Algorithm）；最后编出程序、进行测试、修改直至得到最终解答。

这里面的寻求数学模型的过程，实质就是分析问题，从中提取操作的对象，并找出这些操作对象之间含有的关系的过程。建立好的模型，我们使用数学语言来表达。

这里的模型对应的就是数据结构。我们用计算机编程来解决问题的关键就是，设计出合适的数据结构（例如，用线性表、树、图等）和性能良好的算法。

算法与数据的结构密切相关，算法无不依附于具体的数据结构，数据结构直接关系到算法的选择和效率。通常情况下，设计良好的数据结构可以大大简化算法的实现复杂度，同时可以提升存储效率。数据结构往往同高效的检索算法和索引技术相关。

我们可以把数据结构理解为是ADT的实现。数据结构就是现实问题模型的表达。

数据结构主要解决以下三个问题：

- 数据元素之间的逻辑关系。

这些逻辑关系有：集合、线性结构、树形结构、图形结构等。

- 数据的物理结构。

数据的逻辑结构在计算机存储空间的存放形式。数据的物理结构是数据结构在计算机中的映射。其具体实现的方法有： 顺序（Sequence）、链接（Link）、索引（Index）、散列（Hash）等形式。

其中，顺序存储结构和链式存储结构是我们常用的两种存储结构。

顺序存储是使用元素在存储器中的相对位置来表示数据元素之间的逻辑关系；

链式存储使用指示元素存储位置的指针（pointer）来表示数据元素之间的逻辑关系。

- 数据的处理运算。

5.1.2 集合类是SDK API

我们现在很少用抽象数据类型ADT（Abstract Data Type）这个概念，其实这个概念是OO范式的前身，也是类的前身。ADT加上继承、重载和多态性就是现代OOP编程范式中的类的概念了。我们简称类为广义ADT的概念。

如果我们更加广义的来理解这里的ADT的思想，其实各种编程语言的SDK API、所有的服务（IaaS，PaaS和SaaS等）都是一种更加广义的ADT。

使用ADT可以让我们更简单地描述现实世界。例如：用线性表描述学生成绩表，用树或图描述遗传关系等。

我们知道类的本质就是，对象及其关系的抽象（abstraction）。一个类通常有属性（数据结构）和行为（算法）。使用OO范式编程的大致过程为：

> 划分对象 → 抽象类 → 将类组织成为层次化结构(继承和合成) → 用类与实例进行设计和实现

等几个阶段。

数据抽象本质上讲就是我们解决现实问题的过程中，进行建立领域模型（Domain Model）的过程。

比如说，在前一章节中，我们介绍的程序设计语言的类型系统，本质上就是一种数据抽象。由于计算机的结构和存储的限制（无法像人类大脑神经系统一样去认知识别，并解决现实问题），人类大脑在解决实际问题过程中，经常要计算整数、小数, 要处理英文字符、中文字符， 要持有对象（被操作的数据），要对这些对象进行诸如：查找、排序、修改、传递等操作。把这些问题解决中最常用的数据结构以及其操作算法抽象成对应的类（例如：String、Array、List、Set、Map等），这样我们就可以极大的复用这些功能。而不需要我们自己来实现诸如：字符串、数组、列表、集合、映射等这些的数据结构。通常这些最通用的数据结构，都是现在编程语言中内置的了。

5.1.3 连续存储和离散存储

内存中的存储形式可以分为连续存储和离散存储两种。因此，数据的物理存储结构就有连续存储和离散存储两种，它们对应了我们通常所说的数组和链表。

由于数组是连续存储的，在操作数组中的数据时就可以根据离首地址的偏移量直接存取相应位置上的数据，但是如果要在数据组中任意位置上插入一个元素，就需要先把后面的元素集体向后移一位为其空出存储空间。与之相反，链表是离散存储的，所以在插入一个数据时只要申请一片新空间，然后将其中的连接关系做一个修改就可以，但是显然在链表上查找一个数据时就要逐个遍历了。

考虑以上的总结可见，数组和链表各有优缺点。在具体使用时要根据具体情况选择。当查找数据操作比较多时最好用数组；当对数据集中的数据进行添加或删除比较多时最好选择链表。

5.2 Kotlin 集合类简介

集合类存放的都是对象的引用，而非对象本身，我们通常说的集合中的对象指的是集合中对象的引用（reference)。

Kotlin的集合类分为：**可变集合类**（Mutable）与**不可变集合类**（Immutable）。

集合类型主要有3种：list(列表）、set(集）、和 map(映射)。

(1)列表

列表的主要特征是其对象以线性方式存储，没有特定顺序，只有一个开头和一个结尾，当然，它与根本没有顺序的集是不同的。

列表在数据结构中可表现为：数组和向量、链表、堆栈、队列等。

(2)集

集（set）是最简单的一种集合，它的对象不按特定方式排序，只是简单的把对象加入集合中，就像往口袋里放东西。

对集中成员的访问和操作是通过集中对象的引用进行的，所以集中不能有重复对象。

集也有多种变体，可以实现排序等功能，如TreeSet，它把对象添加到集中的操作将变为按照某种比较规则将其插入到有序的对象序列中。它实现的是SortedSet接口，也就是加入了对象比较的方法。通过对集中的对象迭代，我们可以得到一个升序的对象集合。

(3)映射

映射与集或列表有明显区别，映射中每个项都是成对的。映射中存储的每个对象都有一个相关的关键字（Key）对象，关键字决定了 对象在映射中的存储位置，检索对象时必须提供相应的关键字，就像在字典中查单词一样。关键字应该是唯一的。

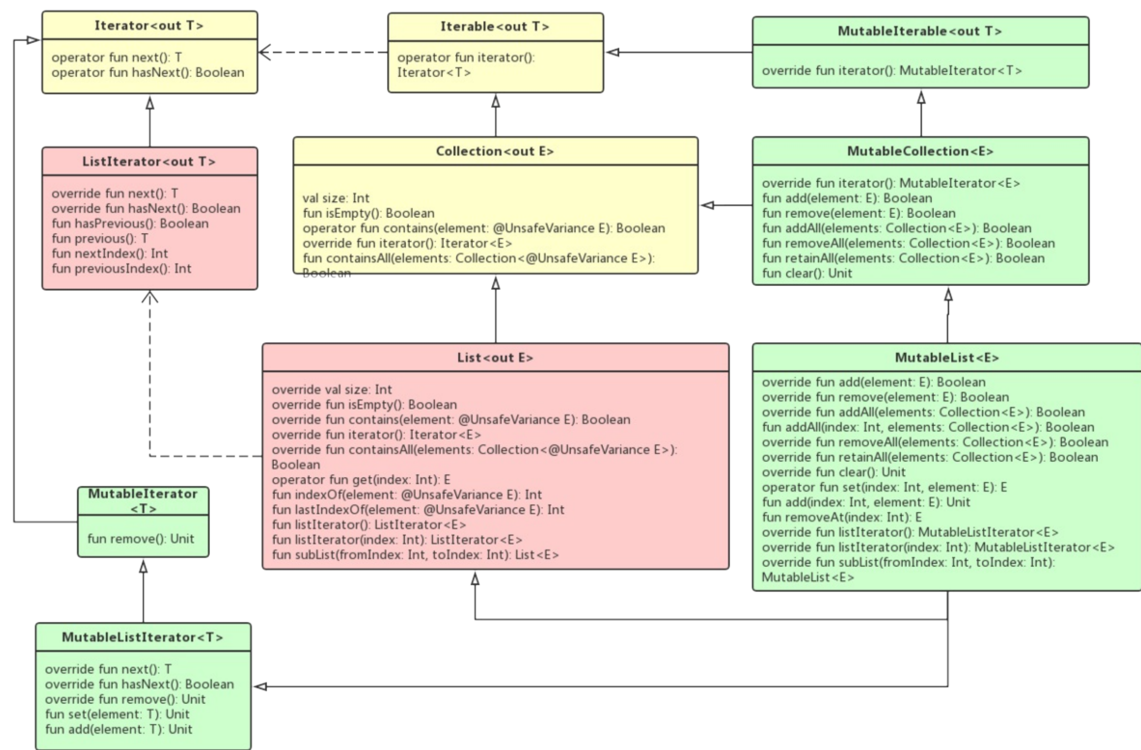
关键字本身并不能决定对象的存储位置，它需要对过一种散列(hashing)技术来处理，产生一个被称作散列码(hash code)的整数值，

散列码通常用作一个偏置量，该偏置量是相对于分配给映射的内存区域起始位置的，由此确定关键字/对象对的存储位置。理想情况 下，散列处理应该产生给定范围内均匀分布的值，而且每个关键字应得到不同的散列码。

5.3 List

List接口继承于Collection接口，元素以线性方式存储，集合中可以存放重复对象。Kotlin的List分为：不可变集合类List（ReadOnly, Immutable）和可变集合类MutableList（Read&Write, Mutable）。

其类图结构如下：



其中， `Iterator` 是所有容器类 `Collection` 的迭代器。迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。 从定义可见，迭代器模式是为容器而生。

5.3.1 创建不可变List

我们可以使用 `listOf` 函数来构建一个不可变的List（read-only，只读的List）。它定义在 `libraries/stdlib/src/kotlin/collections/Collections.kt` 里面。关于 `listOf` 这个构建函数有下面3个重载函数：

```
@kotlin.internal.InlineOnly
public inline fun <T> listOf(): List<T> = emptyList()

public fun <T> listOf(vararg elements: T): List<T> = if (elements.size > 0) elements.asList() else emptyList()

@JvmVersion
public fun <T> listOf(element: T): List<T> = java.util.Collections.singletonList(element)
```

这些函数创建的List都是是只读的（readonly，也就是不可变的immutable ）、可序列化的。

其中，

`listOf()` 用于创建没有元素的空List `listOf(vararg elements: T)` 用于创建只有一个元素的List `listOf(element: T)` 用于创建拥有多个元素的List

我们使用代码示例分别来演示其用法：

首先，我们使用 `listOf()` 来构建一个没有元素的空的List：

```
>>> val list:List<Int> = listOf()
>>> list
[]
>>> list::class
class kotlin.collections.EmptyList
```

注意，这里的变量的类型不能省略，否则会报错：

```
>>> val list = listOf()
error: type inference failed: Not enough information to infer parameter T in inline fun
<T> listOf(): List<T>
Please specify it explicitly.

val list = listOf()
           ^
```

因为这是一个泛型函数。关于泛型，我们将在下一章中介绍。

其中， `EmptyList` 是一个 `internal object EmptyList` , 这是Kotlin内部定义的一个默认空的object List类。

下面，我们再来创建一个只有1个元素的List：

```
>>> val list = listOf(1)
>>> list::class
class java.util.Collections$SingletonList
```

我们可以看出，它实际上是调用Java的 `java.util.Collections` 里面的 `singletonList` 方法：

```
public static <T> List<T> singletonList(T o) {
    return new SingletonList<>(o);
}
```

我们再来创建一个有多个元素的List:

```
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list::class
class java.util.Arrays$ArrayList
>>> list::class.java
```

它调用的是

```
fun <T> listOf(vararg elements: T): List<T> = if (elements.size > 0) elements.asList()
else emptyList()
```

这个函数。其中，`asList`函数是`Array`的扩展函数：

```
public fun <T> Array<out T>.asList(): List<T> {
    return ArraysUtilJVM.asList(this)
}
```

而这个`ArraysUtilJVM`是一个Java类，里面实际上调用的是`java.util.Arrays`和`java.util.List`：

```
package kotlin.collections;

import java.util.Arrays;
import java.util.List;

class ArraysUtilJVM {
    static <T> List<T> asList(T[] array) {
        return Arrays.asList(array);
    }
}
```

另外，我们还可以直接使用`arrayListOf`函数来创建一个Java中的ArrayList对象实例：

```
>>> val list = arrayListOf(0,1,2,3)
>>> list
[0, 1, 2, 3]
>>> list::class
class java.util.ArrayList
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list::class
class java.util.Arrays$ArrayList
```

这个函数定义在`libraries/stdlib/src/kotlin/collections/Collections.kt`类中：

```
@SinceKotlin("1.1")
@kotlin.internal.InlineOnly
```

```
public inline fun <T> arrayListOf(): ArrayList<T> = ArrayList()
```

同样的处理方式，这里的 `ArrayList()` 是Java中的 `java.util.ArrayList` 的类型别名：

```
@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
```

5.3.2 创建可变集合MutableList

在MutableList中，除了继承List中的那些函数外，另外新增了add/addAll、remove/removeAll/removeAt、set、clear、retainAll等更新修改的操作函数。

```
override fun add(element: E): Boolean
override fun remove(element: E): Boolean
override fun addAll(elements: Collection<E>): Boolean
fun addAll(index: Int, elements: Collection<E>): Boolean
override fun removeAll(elements: Collection<E>): Boolean
override fun retainAll(elements: Collection<E>): Boolean
override fun clear(): Unit
operator fun set(index: Int, element: E): E
fun add(index: Int, element: E): Unit
fun removeAt(index: Int): E
override fun listIterator(): MutableListIterator<E>
override fun listIterator(index: Int): MutableListIterator<E>
override fun subList(fromIndex: Int, toIndex: Int): MutableList<E>
```

创建一个MutableList的对象实例跟List类似，前面加上前缀 `mutable` ，代码示例如下：

```
>>> val list = mutableListOf(1, 2, 3)
>>> list
[1, 2, 3]
>>> list::class
class java.util.ArrayList
>>> val list2 = mutableListOf<Int>()
>>> list2
[]
>>> list2::class
class java.util.ArrayList
>>> val list3 = mutableListOf(1)
>>> list3
[1]
>>> list3::class
class java.util.ArrayList
```

我们可以看出，使用 `mutableListOf` 函数创建的可变集合类，实际上背后调用的是 `java.util.ArrayList` 类的相关方法。

另外，我们可以直接使用Kotlin封装的 `arrayListOf` 函数来创建一个ArrayList：

```
>>> val list4 = arrayListOf(1, 2, 3)
>>> list4::class
class java.util.ArrayList
```

关于Kotlin中的 `ArrayList` 类型别名定义在 `kotlin/collections/TypeAliases.kt` 文件中：

```
@file:kotlin.jvm.JvmVersion

package kotlin.collections

@SinceKotlin("1.1") public typealias RandomAccess = java.util.RandomAccess


@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.util.LinkedHashMap<K, V
>
@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.HashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.util.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.HashSet<E>


// also @SinceKotlin("1.1")
internal typealias SortedSet<E> = java.util.SortedSet<E>
internal typealias TreeSet<E> = java.util.TreeSet<E>
```

如果我们已经有了一个不可变的List，而我们现在想把他转换成可变的List，我们可以直接调用转换函数 `toMutableList` ：

```
val list = mutableListOf(1, 2, 3)
val mlist = list.toMutableList()
mlist.add(5)
```

5.3.3 遍历List元素

使用Iterator迭代器

我们以集合 `val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)` 为例，使用Iterator迭代器遍历列表所有元素的操作：

```
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> val iterator = list.iterator()
```

```
>>> iterator
java.util.AbstractList$Itr@438bad7c
>>> while(iterator.hasNext()){
... println(iterator.next())
... }
0
1
2
3
4
5
6
7
8
9
```

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。

Kotlin中的Iterator功能比较简单，并且只能单向移动：

（1）调用iterator()函数，容器返回一个Iterator实例。iterator()函数是 `kotlin.collections.Iterable` 中的函数, 被Collection继承。（2）调用hasNext()函数检查序列中是否还有元素。（3）第一次调用Iterator的next()函数时，它返回序列的第一个元素。依次向后递推，使用next()获得序列中的下一个元素。

当我们调用到最后一个元素，再次调用 `next()` 函数，会抛这个异常 `java.util.NoSuchElementException` 。代码示例：

```
>>> val list = listOf(1,2,3)
>>> val iter = list.iterator()
>>> iter
java.util.AbstractList$Itr@3abfe845
>>> iter.hasNext()
true
>>> iter.next()
1
>>> iter.hasNext()
true
>>> iter.next()
2
>>> iter.hasNext()
true
>>> iter.next()
3
>>> iter.hasNext()
false
>>> iter.next()
```

```
java.util.NoSuchElementException
    at java.util.AbstractList$Itr.next(AbstractList.java:364)
```

我们可以看出，这里的Iterator的实现是在 `AbstractList` 中的内部类 `IteratorImpl` ：

```
private open inner class IteratorImpl : Iterator<E> {
    protected var index = 0
    override fun hasNext(): Boolean = index < size
    override fun next(): E {
        if (!hasNext()) throw NoSuchElementException()
        return get(index++)
    }
}
```

通过这个实现源码，我们可以更加清楚地明白Iterator的工作原理。

其中， `NoSuchElementException()` 这个类是 `java.util.NoSuchElementException` 的类型别名：

```
@kotlin.SinceKotlin public typealias NoSuchElementException = java.util.NoSuchElementEx
ception
```

使用 `forEach` 遍历List元素

这个 `forEach` 函数定义如下：

```
@kotlin.internal.HidesMembers
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

它是 `package kotlin.collections` 包下面的Iterable的扩展内联函数。它的入参是一个函数类型：

```
action: (T) -> Unit
```

关于函数式编程，我们将在后面章节中学习。

这里的 `forEach` 是一个语法糖。实际上 `forEach` 在遍历List对象的时候，仍然使用的是iterator迭代器来进行循环遍历的。

```
>>> val list = listOf(1,2,3)
>>> list
[1, 2, 3]
>>> list.forEach{
... println(it)
... }
```

```
1
2
3
```

当参数只有一个函数的时候，括号可以省略不写。

也就是说，这里面的forEach函数调用的写法，实际上跟下面的写法等价：

```
list.forEach({
    println(it)
})
```

我们甚至还可以直接这样写：

```
>>> list.forEach(::println)
```

其中，`::` 是函数引用符。

5.3.4 List元素操作函数

`add` `remove` `set` `clear`

这两个添加、删除操作函数是MutableList里面的。跟Java中的集合类操作类似。

创建一个可变集合：

```
>>> val mutableList = mutableListOf(1,2,3)
>>> mutableList
[1, 2, 3]
```

向集合中添加一个元素：

```
>>> mutableList.add(4)
true
>>> mutableList
[1, 2, 3, 4]
```

在下标为0的位置添加元素0 ：

```
>>> mutableList.add(0,0)
>>> mutableList
[0, 1, 2, 3, 4]
```

删除元素1 ：

```
>>> mutableList.remove(1)
true
>>> mutableList
[0, 2, 3, 4]
>>> mutableList.remove(1)
false
```

删除下标为1的元素：

```
>>> mutableList.removeAt(1)
2
>>> mutableList
[0, 3, 4]
```

删除子集合：

```
>>> mutableList.removeAll(listOf(3,4))
true
>>> mutableList
[0]
```

添加子集合：

```
>>> mutableList.addAll(listOf(1,2,3))
true
>>> mutableList
[1, 2, 3]
```

更新设置下标0的元素值为100：

```
>>> mutableList.set(0,100)
0
>>> mutableList
[100]
```

清空集合：

```
>>> mutableList.clear()
>>> mutableList
[]
```

把可变集合转为不可变集合：

```
>>> mutableList.toList()
[1, 2, 3]
```

`retainAll`

取两个集合交集：

```
>>> val mlist1 = mutableListOf(1,2,3,4,5,6)
>>> val mlist2 = mutableListOf(3,4,5,6,7,8,9)
>>> mlist1.retainAll(mlist2)
true
>>> mlist1
[3, 4, 5, 6]
```

`contains(element: T): Boolean`

判断集合中是否有指定元素，有就返回true，否则返回false 。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.contains(1)
true
```

`elementAt(index: Int): T`

查找下标对应的元素，如果下标越界会抛IndexOutOfBoundsException。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.elementAt(6)
7
>>> list.elementAt(7)
java.lang.ArrayIndexOutOfBoundsException: 7
    at java.util.Arrays$ArrayList.get(Arrays.java:3841)
```

另外，针对越界的处理，还有下面两个函数：

`elementAtOrElse(index: Int, defaultValue: (Int) -> T): T`：查找下标对应元素，如果越界会根据方法返回默认值。

```
>>> list.elementAtOrElse(7,{0})
0
>>> list.elementAtOrElse(7,{10})
10
```

`elementAtOrNull(index: Int): T?`：查找下标对应元素，如果越界就返回null

```
>>> list.elementAtOrNull(7)
null
```

`first()`

返回集合第1个元素，如果是空集，抛出异常NoSuchElementException。

```
>>> val list = listOf(1,2,3)
>>> list.first()
1
>>> val emptyList = listOf<Int>()
>>> emptyList.first()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.first(_Collections.kt:178)
```

对应的有针对异常处理的函数 `firstOrNull(): T?` ：

```
>>> emptyList.firstOrNull()
null
```

`first(predicate: (T) -> Boolean): T`

返回符合条件的第一个元素，没有则抛异常NoSuchElementException 。

```
>>> val list = listOf(1,2,3)
>>> list.first({it%2==0})
2
>>> list.first({it>100})
java.util.NoSuchElementException: Collection contains no element matching the predicate
.
```

对应的有针对异常处理的函数 `firstOrNull(predicate: (T) -> Boolean): T?` ，返回符合条件的第一个元素，没有就返回null ：

```
>>> list.firstOrNull({it>100})
null
```

`indexOf(element: T): Int`

返回指定下标的元素，没有就返回-1

```
>>> val list = listOf("a","b","c")
>>> list.indexOf("c")
2
```

```
>>> list.indexOf("x")
-1
```

`indexOfFirst(predicate: (T) -> Boolean): Int`

返回第一个符合条件的元素下标，没有就返回-1 。

```
>>> val list = listOf("abc","xyz","xjk","pqk")
>>> list.indexOfFirst({it.contains("x")})
1
>>> list.indexOfFirst({it.contains("k")})
2
>>> list.indexOfFirst({it.contains("e")})
-1
```

`indexOfLast(predicate: (T) -> Boolean): Int`

返回最后一个符合条件的元素下标，没有就返回-1 。

```
>>> val list = listOf("abc","xyz","xjk","pqk")
>>> list.indexOfLast({it.contains("x")})
2
>>> list.indexOfLast({it.contains("k")})
3
>>> list.indexOfLast({it.contains("e")})
-1
```

`last()`

返回集合最后一个元素，空集则抛出异常NoSuchElementException。

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.last()
8
>>> val emptyList = listOf<Int>()
>>> emptyList.last()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.last(_Collections.kt:340)
```

`last(predicate: (T) -> Boolean): T`

返回符合条件的最后一个元素，没有就抛NoSuchElementException

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.last({it==7})
7
```

```
>>> list.last({it>10})
java.util.NoSuchElementException: List contains no element matching the predicate.
```

对应的针对越界处理的 `lastOrNull` 函数：返回符合条件的最后一个元素，没有则返回null：

```
>>> list.lastOrNull({it>10})
null
```

`lastIndexOf(element: T): Int`

返回符合条件的最后一个元素，没有就返回-1

```
>>> val list = listOf("abc","dfg","jkl","abc","bbc","wer")
>>> list.lastIndexOf("abc")
3
```

`single(): T`

该集合如果只有1个元素，则返回该元素。否则，抛异常。

```
>>> val list = listOf(1)
>>> list.single()
1

>>> val list = listOf(1,2)
>>> list.single()
java.lang.IllegalArgumentException: List has more than one element.
    at kotlin.collections.CollectionsKt___CollectionsKt.single(_Collections.kt:471)

>>> val list = listOf<Int>()

>>> list.single()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.single(_Collections.kt:469)
```

`single(predicate: (T) -> Boolean): T`

返回符合条件的单个元素，如有没有符合的抛异常NoSuchElementException，或超过一个的抛异常IllegalArgumentException。

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.single({it==1})
1
>>> list.single({it==7})
java.lang.IllegalArgumentException: Collection contains more than one matching element.
```

```
>>> list.single({it==10})
java.util.NoSuchElementException: Collection contains no element matching the predicate
.
```

对应的针对异常处理的函数 `singleOrNull`：返回符合条件的单个元素，如有没有符合或超过一个，返回null

```
>>> list.singleOrNull({it==7})
null
>>> list.singleOrNull({it==10})
null
```

5.3.5 List集合类的 `any` `all ``none ``count ``reduce ``fold ``max ``min` `sum` 函数算子(operator)

`any()` 判断集合至少有一个元素

这个函数定义如下：

```
public fun <T> Iterable<T>.any(): Boolean {
    for (element in this) return true
    return false
}
```

如果该集合至少有一个元素，返回 `true`，否则返回 `false`。

代码示例：

```
>>> val emptyList = listOf<Int>()
>>> emptyList.any()
false
>>> val list1 = listOf(1)
>>> list1.any()
true
```

`any(predicate: (T) -> Boolean)` 判断集合中是否有满足条件的元素

这个函数定义如下：

```
public inline fun <T> Iterable<T>.any(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (predicate(element)) return true
    return false
}
```

如果该集合中至少有一个元素匹配谓词函数参数 `predicate: (T) -> Boolean` ，返回true，否则返回false。

代码示例：

```
>>> val list = listOf(1, 2, 3)
>>> list.any() // 至少有1个元素
true
>>> list.any({it%2==0}) // 元素2满足{it%2==0}
true
>>> list.any({it>4}) // 没有元素满足{it>4}
false
```

`all(predicate: (T) -> Boolean)` 判断集合中的元素是否都满足条件

函数定义：

```
public inline fun <T> Iterable<T>.all(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (!predicate(element)) return false
    return true
}
```

当且仅当该集合中所有元素都满足条件时，返回 `true` ；否则都返回 `false` 。

代码示例：

```
>>> val list = listOf(0,2,4,6,8)
>>> list.all({it%2==0})
true
>>> list.all({it>2})
false
```

`none()` 判断集合无元素

函数定义：

```
public fun <T> Iterable<T>.none(): Boolean {
    for (element in this) return false
    return true
}
```

如果该集合没有任何元素，返回 `true` ，否则返回 `false` 。

代码示例：

```
>>> val list = listOf<Int>()
```

```
>>> list.none()
true
```

`none(predicate: (T) -> Boolean)` 判断集合中所有元素都不满足条件

函数定义：

```
public inline fun <T> Iterable<T>.none(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (predicate(element)) return false
    return true
}
```

当且仅当集合中所有元素都不满足条件时返回 `true`， 否则返回 `false`。

代码示例：

```
>>> val list = listOf(0,2,4,6,8)
>>> list.none({it%2==1})
true
>>> list.none({it>0})
false
```

`count()` 计算集合中元素的个数

函数定义：

```
public fun <T> Iterable<T>.count(): Int {
    var count = 0
    for (element in this) count++
    return count
}
```

代码示例：

```
>>> val list = listOf(0,2,4,6,8,9)
>>> list.count()
6
```

`count(predicate: (T) -> Boolean)` 计算集合中满足条件的元素的个数

函数定义：

```
public inline fun <T> Iterable<T>.count(predicate: (T) -> Boolean): Int {
    var count = 0
    for (element in this) if (predicate(element)) count++
```

```
    return count
}
```

代码示例：

```
>>> val list = listOf(0,2,4,6,8,9)
>>> list.count()
6
>>> list.count({it%2==0})
5
```

`reduce` 从 第一项到最后一项进行累计运算

函数定义：

```
public inline fun <S, T: S> Iterable<T>.reduce(operation: (acc: S, T) -> S): S {
    val iterator = this.iterator()
    if (!iterator.hasNext()) throw UnsupportedOperationException("Empty collection can'
t be reduced.")
    var accumulator: S = iterator.next()
    while (iterator.hasNext()) {
        accumulator = operation(accumulator, iterator.next())
    }
    return accumulator
}
```

首先把第一个元素赋值给累加子 `accumulator` ，然后逐次向后取元素累加，新值继续赋值给累加子 `accumulator = operation(accumulator, iterator.next())` ，以此类推。最后返回累加子的值。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7,8,9)
>>> list.reduce({sum, next->sum+next})
45
>>> list.reduce({sum, next->sum*next})
362880
>>> val list = listOf("a","b","c")
>>> list.reduce({total, s->total+s})
abc
```

`reduceRight` 从最后一项到第一项进行累计运算

函数定义：

```
public inline fun <S, T: S> List<T>.reduceRight(operation: (T, acc: S) -> S): S {
```

```
    val iterator = listIterator(size)
    if (!iterator.hasPrevious())
        throw UnsupportedOperationException("Empty list can't be reduced.")
    var accumulator: S = iterator.previous()
    while (iterator.hasPrevious()) {
        accumulator = operation(iterator.previous(), accumulator)
    }
    return accumulator
}
```

从函数的定义 `accumulator = operation(iterator.previous(), accumulator)` ，我们可以看出，从右边累计运算的累加子是放在后面的。

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.reduceRight({total, s -> s+total})
cba
```

如果我们位置放错了，会输出下面的结果：

```
>>> list.reduceRight({total, s -> total+s})
abc
```

`fold(initial: R, operation: (acc: R, T) -> R): R` 带初始值的reduce

函数定义：

```
public inline fun <T, R> Iterable<T>.fold(initial: R, operation: (acc: R, T) -> R): R {
    var accumulator = initial
    for (element in this) accumulator = operation(accumulator, element)
    return accumulator
}
```

从函数的定义，我们可以看出，fold函数给累加子赋了初始值 `initial` 。

代码示例：

```
>>> val list=listOf(1,2,3,4)
>>> list.fold(100,{total, next -> next + total})
110
```

`foldRight` 和 `reduceRight` 类似，有初始值。

函数定义：

```
public inline fun <T, R> List<T>.foldRight(initial: R, operation: (T, acc: R) -> R): R {

    var accumulator = initial
    if (!isEmpty()) {
        val iterator = listIterator(size)
        while (iterator.hasPrevious()) {
            accumulator = operation(iterator.previous(), accumulator)
        }
    }
    return accumulator
}
```

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.foldRight("xyz",{s, pre -> pre + s})
xyzcba
```

`forEach(action: (T) -> Unit): Unit` 循环遍历元素，元素是it

我们在前文已经讲述，参看5.3.4。

再写个代码示例：

```
>>> val list = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list.forEach { value -> if (value > 7) println(value) }
8
9
```

`forEachIndexed` 带index(下标) 的元素遍历

函数定义：

```
public inline fun <T> Iterable<T>.forEachIndexed(action: (index: Int, T) -> Unit): Unit
 {
    var index = 0
    for (item in this) action(index++, item)
}
```

代码示例：

```
>>> val list = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list.forEachIndexed { index, value -> if (value > 8) println("value of index $index
 is $value, greater than 8") }
```

```
value of index 9 is 9, greater than 8
```

`max` 、 `min` 查询最大、最小的元素，空集则返回null

`max` 函数定义：

```
public fun <T : Comparable<T>> Iterable<T>.max(): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var max = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (max < e) max = e
    }
    return max
}
```

返回集合中最大的元素。

代码示例：

```
>>> val list = listOf(1,2,3)
>>> list.max()
3
>>> val list = listOf("a","b","c")
>>> list.max()
c
```

`min` 函数定义：

```
public fun <T : Comparable<T>> Iterable<T>.min(): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var min = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (min > e) min = e
    }
    return min
}
```

返回集合中的最小元素。

代码示例：

```
>>> val list = listOf(1,2,3)
```

```
>>> list.min()
1
>>> val list = listOf("a","b","c")
>>> list.min()
a
```

在Kotlin中，字符串的大小比较比较有意思的，我们直接通过代码示例来学习一下：

```
>>> "c" > "a"
true
>>> "abd" > "abc"
true
>>> "abd" > "abcd"
true
>>> "abd" > "abcdefg"
true
```

我们可以看出，字符串的大小比较是按照对应的下标的字符进行比较的。 另外，布尔值的比较是 `true` 大于 `false` ：

```
>>> true > false
true
```

`maxBy(selector: (T) -> R): T?` 、 `minBy(selector: (T) -> R): T?` 获取函数映射结果的最大值、最小值对应的那个元素的值，如果没有则返回null

函数定义：

```
public inline fun <T, R : Comparable<R>> Iterable<T>.maxBy(selector: (T) -> R): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var maxElem = iterator.next()
    var maxValue = selector(maxElem)
    while (iterator.hasNext()) {
        val e = iterator.next()
        val v = selector(e)
        if (maxValue < v) {
            maxElem = e
            maxValue = v
        }
    }
    return maxElem
}
```

也就是说，不直接比较集合元素的大小，而是以集合元素为入参的函数 `selector: (T) -> R` 返回值来比较大小，最后返回此元素的值（注意，不是对应的 `selector` 函数的返回值）。有点像数学里的求函数最值问题：

给定函数 `y = f(x)` ，求 `max f(x)` 的 `x` 的值。

代码示例：

```
>>> val list = listOf(100,-500,300,200)
>>> list.maxBy({it})
300
>>> list.maxBy({it*(1-it)})
100
>>> list.maxBy({it*it})
-500
```

对应的 `minBy` 是获取函数映射后返回结果的最小值所对应那个元素的值，如果没有则返回null。

代码示例：

```
>>> val list = listOf(100,-500,300,200)
>>> list.minBy({it})
-500
>>> list.minBy({it*(1-it)})
-500
>>> list.minBy({it*it})
100
```

`sumBy(selector: (T) -> Int): Int` 获取函数映射值的总和

函数定义：

```
public inline fun <T> Iterable<T>.sumBy(selector: (T) -> Int): Int {
    var sum: Int = 0
    for (element in this) {
        sum += selector(element)
    }
    return sum
}
```

可以看出，这个 `sumBy` 函数算子，累加器 `sum` 初始值为0，返回值是 `Int` 。它的入参 `selector` 是一个函数类型 `(T) -> Int` ，也就是说这个selector也是返回Int类型的函数。

代码示例：

```
>>> val list = listOf(1,2,3,4)
```

```
>>> list.sumBy({it})
10
>>> list.sumBy({it*it})
30
```

类型错误反例：

```
>>> val list = listOf("a","b","c")
>>> list.sumBy({it})
error: type inference failed: inline fun <T> Iterable<T>.sumBy(selector: (T) -> Int): I
nt
cannot be applied to
receiver: List<String>  arguments: ((String) -> String)

list.sumBy({it})
     ^
error: type mismatch: inferred type is (String) -> String but (String) -> Int was expec
ted
list.sumBy({it})
           ^
```

5.3.6 过滤操作函数算子

`take(n: Int): List<T>` 挑出该集合前n个元素的子集合

函数定义：

```
public fun <T> Iterable<T>.take(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return emptyList()
    if (this is Collection<T>) {
        if (n >= size) return toList()
        if (n == 1) return listOf(first())
    }
    var count = 0
    val list = ArrayList<T>(n)
    for (item in this) {
        if (count++ == n)
            break
        list.add(item)
    }
    return list.optimizeReadOnlyList()
}
```

如果n等于0，返回空集；如果n大于集合 `size` ，返回该集合。

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list
[a, b, c]
>>> list.take(2)
[a, b]
>>> list.take(10)
[a, b, c]
>>> list.take(0)
[]
```

`takeWhile(predicate: (T) -> Boolean): List<T>` 挑出满足条件的元素的子集合

函数定义：

```
public inline fun <T> Iterable<T>.takeWhile(predicate: (T) -> Boolean): List<T> {
    val list = ArrayList<T>()
    for (item in this) {
        if (!predicate(item))
            break
        list.add(item)
    }
    return list
}
```

从第一个元素开始，判断是否满足 `predicate` 为true，如果满足条件的元素就丢到返回 `ArrayList` 中。只要遇到任何一个元素不满足条件，就结束循环，返回list 。

代码示例：

```
>>> val list = listOf(1,2,4,6,8,9)
>>> list.takeWhile({it%2==0})
[]
>>> list.takeWhile({it%2==1})
[1]

>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeWhile({it%2==0})
[2, 4, 6, 8]
```

`takeLast` 挑出后n个元素的子集合

函数定义：

```
public fun <T> List<T>.takeLast(n: Int): List<T> {
```

```
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return emptyList()
    val size = size
    if (n >= size) return toList()
    if (n == 1) return listOf(last())
    val list = ArrayList<T>(n)
    if (this is RandomAccess) {
        for (index in size - n .. size - 1)
            list.add(this[index])
    } else {
        for (item in listIterator(n))
            list.add(item)
    }
    return list
}
```

从集合倒数n个元素起，取出到最后一个元素的子集合。如果传入0，返回空集。如果传入n大于集合size，返回整个集合。如果传入负数，直接抛出IllegalArgumentException。

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeLast(0)
[]
>>> list.takeLast(3)
[11, 12, 16]
>>> list.takeLast(100)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.takeLast(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.takeLast(_Collections.kt:734)
```

`takeLastWhile(predicate: (T) -> Boolean)` 从最后开始挑出满足条件元素的子集合

函数定义：

```
public inline fun <T> List<T>.takeLastWhile(predicate: (T) -> Boolean): List<T> {
    if (isEmpty())
        return emptyList()
    val iterator = listIterator(size)
    while (iterator.hasPrevious()) {
        if (!predicate(iterator.previous())) {
            iterator.next()
            val expectedSize = size - iterator.nextIndex()
            if (expectedSize == 0) return emptyList()
            return ArrayList<T>(expectedSize).apply {
```

```
                while (iterator.hasNext())
                    add(iterator.next())
            }
        }
    }
    return toList()
}
```

反方向取满足条件的元素，遇到不满足的元素，直接终止循环，并返回子集合。

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeLastWhile({it%2==0})
[12, 16]
```

`drop(n: Int)` 去除前n个元素返回剩下的元素的子集合

函数定义：

```
public fun <T> Iterable<T>.drop(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return toList()
    val list: ArrayList<T>
    if (this is Collection<*>) {
        val resultSize = size - n
        if (resultSize <= 0)
            return emptyList()
        if (resultSize == 1)
            return listOf(last())
        list = ArrayList<T>(resultSize)
        if (this is List<T>) {
            if (this is RandomAccess) {
                for (index in n..size - 1)
                    list.add(this[index])
            } else {
                for (item in listIterator(n))
                    list.add(item)
            }
            return list
        }
    }
    else {
        list = ArrayList<T>()
    }
    var count = 0
    for (item in this) {
```

```
        if (count++ >= n) list.add(item)
    }
    return list.optimizeReadOnlyList()
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.drop(5)
[11, 12, 16]
>>> list.drop(100)
[]
>>> list.drop(0)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.drop(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.drop(_Collections.kt:538)
```

`dropWhile(predicate: (T) -> Boolean)` 去除满足条件的元素返回剩下的元素的子集合

函数定义：

```
public inline fun <T> Iterable<T>.dropWhile(predicate: (T) -> Boolean): List<T> {
    var yielding = false
    val list = ArrayList<T>()
    for (item in this)
        if (yielding)
            list.add(item)
        else if (!predicate(item)) {
            list.add(item)
            yielding = true
        }
    return list
}
```

去除满足条件的元素，当遇到一个不满足条件的元素时，中止操作，返回剩下的元素子集合。

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropWhile({it%2==0})
[9, 11, 12, 16]
```

`dropLast(n: Int)` 从最后去除n个元素

函数定义：

```
public fun <T> List<T>.dropLast(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    return take((size - n).coerceAtLeast(0))
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropLast(3)
[2, 4, 6, 8, 9]
>>> list.dropLast(100)
[]
>>> list.dropLast(0)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.dropLast(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.dropLast(_Collections.kt:573)
```

`dropLastWhile(predicate: (T) -> Boolean)` 从最后满足条件的元素

函数定义：

```
public inline fun <T> List<T>.dropLastWhile(predicate: (T) -> Boolean): List<T> {
    if (!isEmpty()) {
        val iterator = listIterator(size)
        while (iterator.hasPrevious()) {
            if (!predicate(iterator.previous())) {
                return take(iterator.nextIndex() + 1)
            }
        }
    }
    return emptyList()
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropLastWhile({it%2==0})
[2, 4, 6, 8, 9, 11]
```

`slice(indices: IntRange)` 取开始下标至结束下标元素子集合

函数定义：

```
public fun <T> List<T>.slice(indices: IntRange): List<T> {
    if (indices.isEmpty()) return listOf()
    return this.subList(indices.start, indices.endInclusive + 1).toList()
}
```

代码示例：

```
val list = listOf(2,4,6,8,9,11,12,16)
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(1..3)
[4, 6, 8]
>>> list.slice(2..7)
[6, 8, 9, 11, 12, 16]
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(1..3)
[4, 6, 8]
>>> list.slice(2..7)
[6, 8, 9, 11, 12, 16]
```

`slice(indices: Iterable<Int>)` 返回指定下标的元素子集合

函数定义：

```
public fun <T> List<T>.slice(indices: Iterable<Int>): List<T> {
    val size = indices.collectionSizeOrDefault(10)
    if (size == 0) return emptyList()
    val list = ArrayList<T>(size)
    for (index in indices) {
        list.add(get(index))
    }
    return list
}
```

这个函数从签名上看，不是那么简单直接。从函数的定义看，这里的indices是当做原来集合的下标来使用的。

代码示例：

```
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(listOf(2,4,6))
[6, 9, 12]
```

我们可以看出，这里是取出下标为2，4，6的元素。而不是直观理解上的，去掉元素2，4，6。

`filterTo(destination: C, predicate: (T) -> Boolean)` 过滤出满足条件的元素并赋值给destination

函数定义：

```
public inline fun <T, C : MutableCollection<in T>> Iterable<T>.filterTo(destination: C,
 predicate: (T) -> Boolean): C {
    for (element in this) if (predicate(element)) destination.add(element)
    return destination
}
```

把满足过滤条件的元素组成的子集合赋值给入参destination。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> val dest = mutableListOf<Int>()
>>> list.filterTo(dest,{it>3})
[4, 5, 6, 7]
>>> dest
[4, 5, 6, 7]
```

`filter(predicate: (T) -> Boolean)` 过滤出满足条件的元素组成的子集合

函数定义：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

相对于filterTo函数，filter函数更加简单易用。从源码我们可以看出，filter函数直接调用的 `filterTo(ArrayList<T>(), predicate)` ， 其中入参destination被直接默认赋值为 `ArrayList<T>()` 。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.filter({it>3})
[4, 5, 6, 7]
```

另外，还有下面常用的过滤函数：

`filterNot(predicate: (T) -> Boolean)` ，用来过滤所有不满足条件的元素 ； `filterNotNull()` 过滤掉 `null` 元素。

5.3.7 映射操作符

`map(transform: (T) -> R): List<R>`

将集合中的元素通过转换函数 `transform` 映射后的结果，存到一个集合中返回。

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.map({it})
[1, 2, 3, 4, 5, 6, 7]
>>> list.map({it*it})
[1, 4, 9, 16, 25, 36, 49]
>>> list.map({it+10})
[11, 12, 13, 14, 15, 16, 17]
```

这个函数内部调用的是

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

这里的mapTo函数定义如下：

```
public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo(destination: C,
 transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

我们可以看出，这个map实现的原理是循环遍历原集合中的元素，并把通过transform映射后的结果放到一个新的destination集合中，并返回destination。

`mapIndexed(transform: (kotlin.Int, T) -> R)`

转换函数 `transform` 中带有下标参数。也就是说我们可以同时使用下标和元素的值来进行转换。其中，第一个参数是Int类型的下标。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.mapIndexed({index,it -> index*it})
[0, 2, 6, 12, 20, 30, 42]
```

`mapNotNull(transform: (T) -> R?)`

遍历集合每个元素，得到通过函数算子transform映射之后的值，剔除掉这些值中的null，返回一个无null元素的集合。

代码示例：

```
>>> val list = listOf("a","b",null,"x",null,"z")
>>> list.mapNotNull({it})
[a, b, x, z]
```

这个函数内部实现是调用的 `mapNotNullTo` 函数：

```
public inline fun <T, R : Any, C : MutableCollection<in R>> Iterable<T>.mapNotNullTo(destination: C, transform: (T) -> R?): C {
    forEach { element -> transform(element)?.let { destination.add(it) } }
    return destination
}
```

`flatMap(transform: (T) -> Iterable<R>): List<R>`

在原始集合的每个元素上调用 `transform` 转换函数，得到的映射结果组成的单个列表。为了更简单的理解这个函数，我们跟 `map(transform: (T) -> R): List<R>` 对比下。

首先看函数的各自的实现：

map：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}

public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo(destination: C,
 transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

flatMap:

```
public inline fun <T, R> Iterable<T>.flatMap(transform: (T) -> Iterable<R>): List<R> {
    return flatMapTo(ArrayList<R>(), transform)
}

public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.flatMapTo(destination
: C, transform: (T) -> Iterable<R>): C {
    for (element in this) {
        val list = transform(element)
```

```
        destination.addAll(list)
    }
    return destination
}
```

我们可以看出，这两个函数主要区别在transform函数返回上。

代码示例

```
>>> val list = listOf("a","b","c")
>>> list.map({it->listOf(it+1,it+2,it+3)})
[[a1, a2, a3], [b1, b2, b3], [c1, c2, c3]]
>>> list.flatMap({it->listOf(it+1,it+2,it+3)})
[a1, a2, a3, b1, b2, b3, c1, c2, c3]
```

从代码运行结果我们可以看出，使用 map 是把list中的每一个元素都映射成一个 `List-n` ，然后以这些 `List-n` 为元素，组成一个大的嵌套的 `List` 返回。而使用flatMap则是把list中的第一个元素映射成一个List1，然后把第二个元素映射成的List2跟List1合并： `List1.addAll(List2)` ，以此类推。最终返回一个"扁平的"（flat）List。

其实，这个flatMap的过程是 `map + flatten` 两个操作的组合。这个flatten函数定义如下：

```
public fun <T> Iterable<Iterable<T>>.flatten(): List<T> {
    val result = ArrayList<T>()
    for (element in this) {
        result.addAll(element)
    }
    return result
}
```

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.map({it->listOf(it+1,it+2,it+3)})
[[a1, a2, a3], [b1, b2, b3], [c1, c2, c3]]
>>> list.map({it->listOf(it+1,it+2,it+3)}).flatten()
[a1, a2, a3, b1, b2, b3, c1, c2, c3]
```

5.3.8 分组操作符

`groupBy(keySelector: (T) -> K): Map<K, List<T>>`

将集合中的元素按照条件选择器 `keySelector` （是一个函数）分组，并返回Map。代码示例：

```
>>> val words = listOf("a", "abc", "ab", "def", "abcd")
```

```
>>> val lengthGroup = words.groupBy { it.length }
>>> lengthGroup
{1=[a], 3=[abc, def], 2=[ab], 4=[abcd]}
```

`groupBy(keySelector: (T) -> K, valueTransform: (T) -> V)`

分组函数还有一个是 `groupBy(keySelector: (T) -> K, valueTransform: (T) -> V)` ，根据条件选择器keySelector和转换函数valueTransform分组。

代码示例

```
>>> val programmer = listOf("K&R" to "C", "Bjar" to "C++", "Linus" to "C", "James" to "
Java")
>>> programmer
[(K&R, C), (Bjar, C++), (Linus, C), (James, Java)]
>>> programmer.groupBy({it.second}, {it.first})
{C=[K&R, Linus], C++=[Bjar], Java=[James]}
```

这里涉及到一个 二元组Pair 类，该类是Kotlin提供的用来处理二元数据组的。 可以理解成Map中的一个键值对，比如Pair(“key” ,” value”) 等价于 “key” to “value” 。

我们再通过下面的代码示例，来看一下这两个分组的区别：

```
>>> val words = listOf("a", "abc", "ab", "def", "abcd")
>>> words.groupBy( { it.length })
{1=[a], 3=[abc, def], 2=[ab], 4=[abcd]}
>>> words.groupBy( { it.length },{it.contains("b")})
{1=[false], 3=[true, false], 2=[true], 4=[true]}
```

我们可以看出，后者是在前者的基础上又映射了一次 `{it.contains("b")}` ，把第2次映射的结果放到返回的Map中了。

`groupingBy(crossinline keySelector: (T) -> K): Grouping<T, K>`

另外，我们还可以使用 `groupingBy(crossinline keySelector: (T) -> K): Grouping<T, K>` 函数来创建一个 `Grouping` ，然后调用计数函数 `eachCount` 统计分组：

代码示例

```
>>> val words = "one two three four five six seven eight nine ten".split(' ')
>>> words.groupingBy({it.first()}).eachCount()
{o=1, t=3, f=2, s=2, e=1, n=1}
```

上面的例子是统计words列表的元素单词中首字母出现的频数。

其中， `eachCount` 函数定义如下：

```
@SinceKotlin("1.1")
@JvmVersion
public fun <T, K> Grouping<T, K>.eachCount(): Map<K, Int> =
        // fold(0) { acc, e -> acc + 1 } optimized for boxing
        foldTo( destination = mutableMapOf(),
                initialValueSelector = { _, _ -> kotlin.jvm.internal.Ref.IntRef() },
                operation = { _, acc, _ -> acc.apply { element += 1 } })
        .mapValuesInPlace { it.value.element }
```

5.3.9 排序操作符

`reversed(): List<T>`

倒序排列集合元素。 代码示例

```
>>> val list = listOf(1,2,3)
>>> list.reversed()
[3, 2, 1]
```

这个函数，Kotlin是直接调用的 `java.util.Collections.reverse()` 方法。其相关代码如下：

```
public fun <T> Iterable<T>.reversed(): List<T> {
    if (this is Collection && size <= 1) return toList()
    val list = toMutableList()
    list.reverse()
    return list
}

public fun <T> MutableList<T>.reverse(): Unit {
    java.util.Collections.reverse(this)
}
```

`sorted` 和 `sortedDescending`

升序排序和降序排序。

代码示例

```
>>> val list = listOf(1,3,2)
>>> list.sorted()
[1, 2, 3]
>>> list.sortedDescending()
[3, 2, 1]
```

`sortedBy` 和 `sortedByDescending`

可变集合MutableList的排序操作。根据函数映射的结果进行升序排序和降序排序。 这两个函数定义如下：

```
public inline fun <T, R : Comparable<R>> MutableList<T>.sortBy(crossinline selector: (T)
 -> R?): Unit {
    if (size > 1) sortWith(compareBy(selector))
}
public inline fun <T, R : Comparable<R>> MutableList<T>.sortByDescending(crossinline se
lector: (T) -> R?): Unit {
    if (size > 1) sortWith(compareByDescending(selector))
}
```

代码示例

```
>>> val mlist = mutableListOf("abc","c","bn","opqde","")
>>> mlist.sortBy({it.length})
>>> mlist
[, c, bn, abc, opqde]
>>> mlist.sortByDescending({it.length})
>>> mlist
[opqde, abc, bn, c, ]
```

5.3.10 生产操作符

`zip(other: Iterable<R>): List<Pair<T, R>>`

两个集合按照下标配对，组合成的每个Pair作为新的List集合中的元素，并返回。

如果两个集合长度不一样，取短的长度。

代码示例

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5,6,7)
>>> val list3 = listOf("x","y","z")
>>> list1.zip(list3)
[(1, x), (2, y), (3, z)]
>>> list3.zip(list1)
[(x, 1), (y, 2), (z, 3)]
>>> list2.zip(list3)
[(4, x), (5, y), (6, z)]  // 取短的长度
>>> list3.zip(list2)
[(x, 4), (y, 5), (z, 6)]
>>> list1.zip(listOf<Int>())
[]
```

这个zip函数的定义如下：

```
public infix fun <T, R> Iterable<T>.zip(other: Iterable<R>): List<Pair<T, R>> {
    return zip(other) { t1, t2 -> t1 to t2 }
}
```

我们可以看出，其内部是调用了 `zip(other) { t1, t2 -> t1 to t2 }` 。这个函数定义如下：

```
public inline fun <T, R, V> Iterable<T>.zip(other: Iterable<R>, transform: (a: T, b: R)
 -> V): List<V> {
    val first = iterator()
    val second = other.iterator()
    val list = ArrayList<V>(minOf(collectionSizeOrDefault(10), other.collectionSizeOrDe
fault(10)))
    while (first.hasNext() && second.hasNext()) {
        list.add(transform(first.next(), second.next()))
    }
    return list
}
```

依次取两个集合相同索引的元素，使用提供的转换函数transform得到映射之后的值，作为元素组成一个新的List，并返回该List。列表的长度取两个集合中最短的。

代码示例

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5,6,7)
>>> val list3 = listOf("x","y","z")
>>> list1.zip(list3, {t1,t2 -> t2+t1})
[x1, y2, z3]
>>> list1.zip(list2, {t1,t2 -> t1*t2})
[4, 10, 18]
```

`unzip(): Pair<List<T>, List<R>>`

首先这个函数作用在元素是Pair的集合类上。依次取各个Pair元素的first, second值，分别放到List、List中，然后返回一个first为List，second为List的大的Pair。

函数定义

```
public fun <T, R> Iterable<Pair<T, R>>.unzip(): Pair<List<T>, List<R>> {
    val expectedSize = collectionSizeOrDefault(10)
    val listT = ArrayList<T>(expectedSize)
    val listR = ArrayList<R>(expectedSize)
    for (pair in this) {
        listT.add(pair.first)
```

```
        listR.add(pair.second)
    }
    return listT to listR
}
```

看到这里，仍然有点抽象，我们直接看代码示例：

```
>>> val listPair = listOf(Pair(1,2),Pair(3,4),Pair(5,6))
>>> listPair
[(1, 2), (3, 4), (5, 6)]
>>> listPair.unzip()
([1, 3, 5], [2, 4, 6])
```

`partition(predicate: (T) -> Boolean): Pair<List<T>, List<T>>`

根据判断条件是否成立，将集合拆分成两个子集合组成的 Pair。我们可以直接看函数的定义来更加清晰的理解这个函数的功能：

```
public inline fun <T> Iterable<T>.partition(predicate: (T) -> Boolean): Pair<List<T>, L
ist<T>> {
    val first = ArrayList<T>()
    val second = ArrayList<T>()
    for (element in this) {
        if (predicate(element)) {
            first.add(element)
        } else {
            second.add(element)
        }
    }
    return Pair(first, second)
}
```

我们可以看出，这是一个内联函数。

代码示例

```
>>> val list = listOf(1,2,3,4,5,6,7,8,9)
>>> list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.partition({it>5})
([6, 7, 8, 9], [1, 2, 3, 4, 5])
```

`plus(elements: Iterable<T>): List<T>`

合并两个List。

函数定义

```
public operator fun <T> Iterable<T>.plus(elements: Iterable<T>): List<T> {
    if (this is Collection) return this.plus(elements)
    val result = ArrayList<T>()
    result.addAll(this)
    result.addAll(elements)
    return result
}
```

我们可以看出，这是一个操作符函数。可以用”+”替代 。

代码示例

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5)
>>> list1.plus(list2)
[1, 2, 3, 4, 5]
>>> list1+list2
[1, 2, 3, 4, 5]
```

关于plus函数还有以下的重载函数：

```
plus(element: T): List<T>
plus(elements: Array<out T>): List<T>
plus(elements: Sequence<T>): List<T>
```

等。

`plusElement(element: T): List<T>`

在集合中添加一个元素。 函数定义

```
@kotlin.internal.InlineOnly
public inline fun <T> Iterable<T>.plusElement(element: T): List<T> {
    return plus(element)
}
```

我们可以看出，这个函数内部是直接调用的 `plus(element: T): List<T>` 。

代码示例

```
>>> list1 + 10
[1, 2, 3, 10]
>>> list1.plusElement(10)
[1, 2, 3, 10]
```

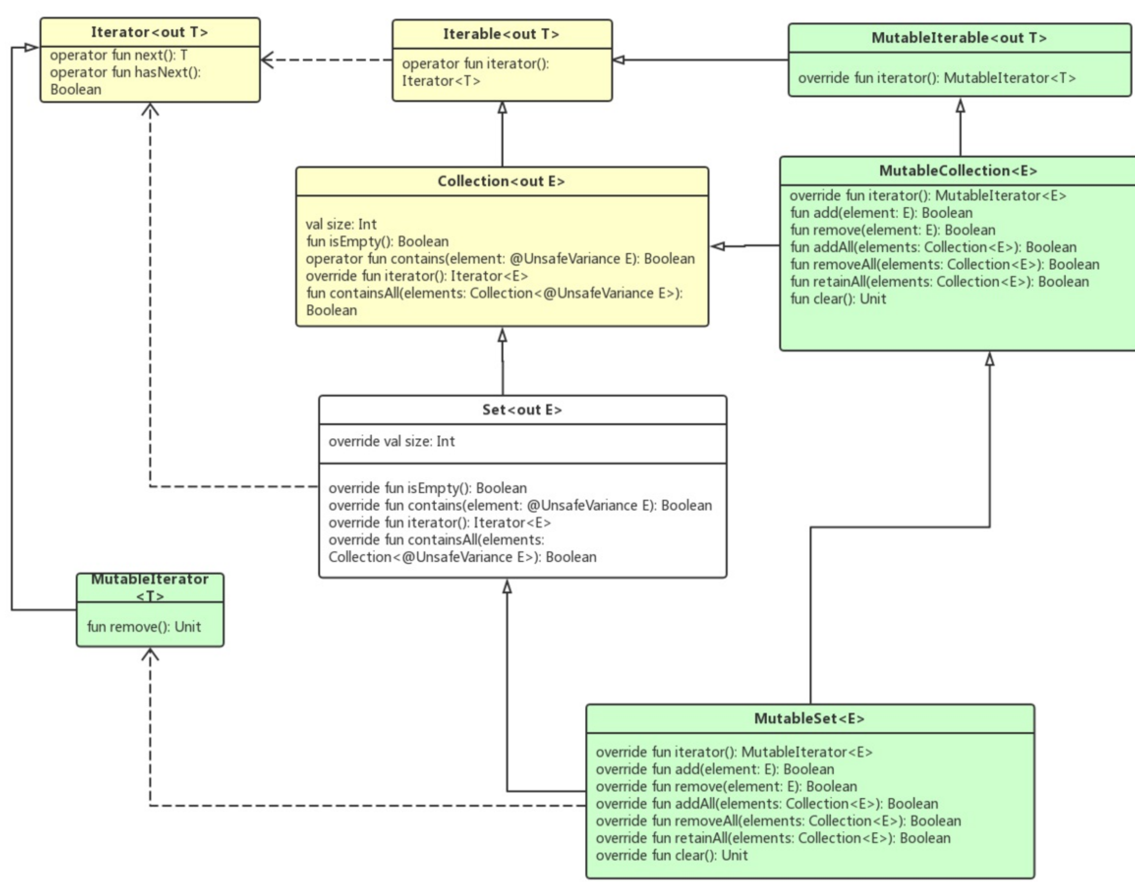
```
>>> list1.plus(10)
[1, 2, 3, 10]
```

5.4 Set

类似的，Kotlin中的Set也分为：不可变Set和支持增加和删除的可变MutableSet。

不可变Set同样是继承了Collection。MutableSet接口继承于Set, MutableCollection，同时对Set进行扩展，添加了对元素添加和删除等操作。

Set的类图结构如下：



5.4.1 空集

万物生于无。我们先来看下Kotlin中的空集：

```
internal object EmptySet : Set<Nothing>, Serializable {
    private const val serialVersionUID: Long = 3406603774387020532

    override fun equals(other: Any?): Boolean = other is Set<*> && other.isEmpty()
    override fun hashCode(): Int = 0
    override fun toString(): String = "[]"
```

```
    override val size: Int get() = 0
    override fun isEmpty(): Boolean = true
    override fun contains(element: Nothing): Boolean = false
    override fun containsAll(elements: Collection<Nothing>): Boolean = elements.isEmpty
()

    override fun iterator(): Iterator<Nothing> = EmptyIterator

    private fun readResolve(): Any = EmptySet
}
```

空集继承了Serializable，表明是可被序列化的。它的size是0， isEmpty()返回true， hashCode()也是0。

下面是创建一个空集的代码示例：

```
>>> val emptySet = emptySet<Int>()
>>> emptySet
[]
>>> emptySet.size
0
>>> emptySet.isEmpty()
true
>>> emptySet.hashCode()
0
```

5.4.2 创建Set

`setOf`

首先，Set中的元素是不可重复的（任意两个元素 x, y 都不相等）。这里的元素 x, y 不相等的意思是：

```
x.hashCode() != y.hashCode()
!x.equals(y)
```

上面两个表达式值都为true 。

代码示例

```
>>> val list = listOf(1,1,2,3,3)
>>> list
[1, 1, 2, 3, 3]
>>> val set = setOf(1,1,2,3,3)
>>> set
```

```
[1, 2, 3]
```

Kotlin跟Java一样的，判断两个对象的是否重复标准是hashCode()和equals()两个参考值，也就是说只有两个对象的hashCode值一样与equals()为真时，才认为是相同的对象。所以自定义的类必须要要重写hashCode()和equals()两个函数。作为Java程序员，这里一般都会注意到。

创建多个元素的Set使用的函数是

```
setOf(vararg elements: T): Set<T> = if (elements.size > 0) elements.toSet() else emptyS
et()
```

这个toSet()函数是Array类的扩展函数，定义如下

```
public fun <T> Array<out T>.toSet(): Set<T> {
    return when (size) {
        0 -> emptySet()
        1 -> setOf(this[0])
        else -> toCollection(LinkedHashSet<T>(mapCapacity(size)))
    }
}
```

我们可以看出，setOf函数背后实际上用的是LinkedHashSet构造函数。关于创建Set的初始容量的算法是：

```
@PublishedApi
internal fun mapCapacity(expectedSize: Int): Int {
    if (expectedSize < 3) {
        return expectedSize + 1
    }
    if (expectedSize < INT_MAX_POWER_OF_TWO) {
        return expectedSize + expectedSize / 3
    }
    return Int.MAX_VALUE // 2147483647, any large value
}
```

也就是说，当元素个数n小于3，初始容量为n+1; 当元素个数n小于 `2147483647 / 2 + 1` ，初始容量为 `n + n/3` ; 否则，初始容量为 `2147483647` 。

如果我们想对一个List去重，可以直接使用下面的方式

```
>>> list.toSet()
[1, 2, 3]
```

上文我们使用 `emptySet<Int>()` 来创建空集，我们也可以使用setOf()来创建空集：

```
>>> val s = setOf<Int>()
>>> s
[]
```

创建1个元素的Set：

```
>>> val s = setOf<Int>(1)
>>> s
[1]
```

这个函数调用的是 `setOf(element: T): Set<T> = java.util.Collections.singleton(element)`，也是Java的Collections类里的方法。

`mutableSetOf(): MutableSet<T>`

创建一个可变Set。

函数定义

```
@SinceKotlin("1.1")
@kotlin.internal.InlineOnly
public inline fun <T> mutableSetOf(): MutableSet<T> = LinkedHashSet()
```

这个 `LinkedHashSet()` 构造函数背后实际上是 `java.util.LinkedHashSet<E>`， 这就是Kotlin中的类型别名。

5.4.3 使用Java中的Set类

包kotlin.collections下面的TypeAliases.kt类中，有一些类型别名的定义如下：

```
@file:kotlin.jvm.JvmVersion

package kotlin.collections

@SinceKotlin("1.1") public typealias RandomAccess = java.util.RandomAccess


@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.util.LinkedHashMap<K, V>

@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.HashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.util.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.HashSet<E>


// also @SinceKotlin("1.1")
```

```
internal typealias SortedSet<E> = java.util.SortedSet<E>
internal typealias TreeSet<E> = java.util.TreeSet<E>
```

从这里，我们可以看出，Kotlin中的 `LinkedHashSet` ，`HashSet`，`SortedSet`，`TreeSet` 就是直接使用的Java中的对应的集合类。

对应的创建的方法是

```
hashSetOf
linkedSetOf
mutableSetOf
sortedSetOf
```

代码示例如下：

```
>>> val hs = hashSetOf(1,3,2,7)
>>> hs
[1, 2, 3, 7]
>>> hs::class
class java.util.HashSet
>>> val ls = linkedSetOf(1,3,2,7)
>>> ls
[1, 3, 2, 7]
>>> ls::class
class java.util.LinkedHashSet
>>> val ms = mutableSetOf(1,3,2,7)
>>> ms
[1, 3, 2, 7]
>>> ms::class
class java.util.LinkedHashSet
>>> val ss = sortedSetOf(1,3,2,7)
>>> ss
[1, 2, 3, 7]
>>> ss::class
class java.util.TreeSet
```

我们知道在Java中，Set接口有两个主要的实现类HashSet和TreeSet：

HashSet : 该类按照哈希算法来存取集合中的对象，存取速度较快。 TreeSet : 该类实现了SortedSet接口，能够对集合中的对象进行排序。 LinkedHashSet：具有HashSet的查询速度，且内部使用链表维护元素的顺序，在对Set元素进行频繁插入、删除的场景中使用。

Kotlin并没有单独去实现一套HashSet、TreeSet和LinkedHashSet。如果我们在实际开发过程中，需要用到这些Set, 就可以直接用上面的方法。

5.4.4 Set元素的加减操作 `plus` `minus`

Kotlin中针对Set做了一些加减运算的扩展函数, 例如：

```
operator fun <T> Set<T>.plus(element: T)
plusElement(element: T)
plus(elements: Iterable<T>)
operator fun <T> Set<T>.minus(element: T)
minusElement(element: T)
minus(elements: Iterable<T>)
```

代码示例：

```
>>> val ms = mutableSetOf(1,3,2,7)
>>> ms+10
[1, 3, 2, 7, 10]
>>> ms-1
[3, 2, 7]
>>>
>>> ms + listOf(8,9)
[1, 3, 2, 7, 8, 9]
>>> ms - listOf(8,9)
[1, 3, 2, 7]
>>> ms - listOf(1,3)
[2, 7]
```

5.5 Map

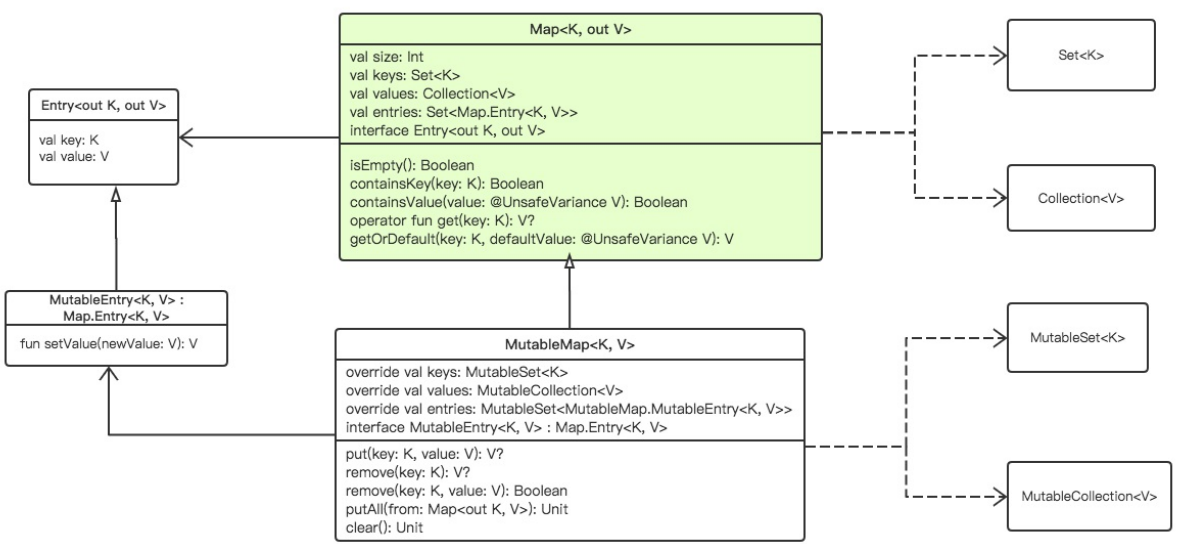
5.5.1 Map概述

Map是一种把键对象Key和值对象Value映射的集合，它的每一个元素都包含一对键对象和值对象（K-V Pair）。 Key可以看成是Value 的索引，作为key的对象在集合中不可重复（uniq）。

如果我们从数据结构的本质上来看，其实List就是Key是Int类型下标的特殊的Map。而Set也是Key为Int，但是Value值不能重复的特殊Map。

Kotlin中的Map与List、Set一样，Map也分为只读Map和可变的MutableMap。

Map没有继承于Collection接口。其类图结构如下：



在接口 `interface Map<K, out V>` 中，K是键值的类型，V是对应的映射值的类型。这里的 `out V` 表示类型为V或V的子类。这是泛型的相关知识，我们将在下一章节中介绍。

其中， `Entry<out K, out V>` 中保存的是Map的键值对。

5.5.2 创建Map

跟Java相比不同的是，在Kotlin中的Map区分了只读的Map和可编辑的Map（MutableMap、HashMap、LinkedHashMap）。

Kotlin没有自己重新去实现一套集合类，而是在Java的集合类基础上做了一些扩展。

我们知道在Java中，根据内部数据结构的不同，Map 接口通常有多种实现类。

其中常用的有：

- HashMap

HashMap是基于哈希表（hash table）的 Map 接口的实现，以key-value的形式存在。在HashMap中，key-value是一个整体，系统会根据hash算法来来计算key-value的存储位置，我们可以通过key快速地存取value。它允许使用 null 值和 null 键。

另外，HashMap中元素的顺序，随着时间的推移会发生变化。

- TreeMap

使用红黑二叉树（red-black tree）的 Map 接口的实现。

- LinkedHashMap

还有继承了HashMap，并使用链表实现的LinkedHashMap。LinkedHashMap保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录是先插入的记录。简单说，LinkedHashMap是有序的，它使用链表维护内部次序。

我们在使用Kotlin创建Map的时候，实际上大部分都是调用Java的Map的方法。

下面我们就来介绍Map的创建以及基本操作函数。

`mapOf()`

创建一个只读空Map。

```
>>> val map1 = mapOf<String, Int>()
>>> map1.size
0
>>> map1.isEmpty()
true
```

我们还可以用另外一个函数创建空Map：

```
>>> val map2 = emptyMap<String, Int>()
>>> map2.size
0
>>> map2.isEmpty()
true
```

空Map都是相等的：

```
>>> map2==map1
true
```

这个空Map是只读的，其属性和函数返回都是预定义好的。其代码如下：

```
private object EmptyMap : Map<Any?, Nothing>, Serializable {
    private const val serialVersionUID: Long = 8246714829545688274

    override fun equals(other: Any?): Boolean = other is Map<*,*> && other.isEmpty()
    override fun hashCode(): Int = 0
    override fun toString(): String = "{}"

    override val size: Int get() = 0
    override fun isEmpty(): Boolean = true

    override fun containsKey(key: Any?): Boolean = false
    override fun containsValue(value: Nothing): Boolean = false
    override fun get(key: Any?): Nothing? = null
    override val entries: Set<Map.Entry<Any?, Nothing>> get() = EmptySet
    override val keys: Set<Any?> get() = EmptySet
    override val values: Collection<Nothing> get() = EmptyList

```

```
    private fun readResolve(): Any = EmptyMap
}
```

`mapOf(pair: Pair<K, V>): Map<K, V>`

使用二元组Pair创建一个只读Map。

```
>>> val map = mapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
>>> map.get(1)
x
>>> map.get(3)
z
>>> map.size
3
>>> map.entries
[1=x, 2=y, 3=z]
```

这个创建函数内部是调用的LinkedHashMap构造函数，其相关代码如下：

```
pairs.toMap(LinkedHashMap(mapCapacity(pairs.size)))
```

如果我们想编辑这个Map，编译器会直接报错

```
>>> map[1]="a"
error: unresolved reference. None of the following candidates is applicable because of
receiver type mismatch:
@InlineOnly public operator inline fun <K, V> MutableMap<Int, String>.set(key: Int, val
ue: String): Unit defined in kotlin.collections
@InlineOnly public operator inline fun kotlin.text.StringBuilder /* = java.lang.StringB
uilder */.set(index: Int, value: Char): Unit defined in kotlin.text
map[1]="a"
^
error: no set method providing array access
map[1]="a"
   ^
```

因为在不可变（Immutable）Map中，根本就没有提供set函数。

`mutableMapOf()`

创建一个空的可变的Map。

```
>>> val map = mutableMapOf<Int, Any?>()
>>> map.isEmpty()
```

```
true
>>> map[1] = "x"
>>> map[2] = 1
>>> map
{1=x, 2=1}
```

该函数直接是调用的LinkedHashMap()构造函数。

`mutableMapOf(vararg pairs: Pair<K, V>): MutableMap<K, V>`

创建一个可编辑的MutableMap对象。

```
>>> val map = mutableMapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
>>> map[1]="a"
>>> map
{1=a, 2=y, 3=z}
```

另外，如果Map中有重复的key键，后面的会直接覆盖掉前面的：

```
>>> val map = mutableMapOf(1 to "x", 2 to "y", 1 to "z")
>>> map
{1=z, 2=y}
```

后面的 `1 to "z"` 直接把前面的 `1 to "x"` 覆盖掉了。

`hashMapOf(): HashMap<K, V>`

创建HashMap对象。Kotlin直接使用的是Java的HashMap。

```
>>> val map: HashMap<Int, String> = hashMapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
```

`linkedMapOf(): LinkedHashMap<K, V>`

创建空对象LinkedHashMap。直接使用的是Java中的LinkedHashMap。

```
>>> val map: LinkedHashMap<Int, String> = linkedMapOf()
>>> map
{}
>>> map[1]="x"
>>> map
{1=x}
```

`linkedMapOf(vararg pairs: Pair<K, V>): LinkedHashMap<K, V>`

创建带二元组Pair元素的LinkedHashMap对象。直接使用的是Java中的LinkedHashMap。

```
>>> val map: LinkedHashMap<Int, String> = linkedMapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
>>> map[1]="a"
>>> map
{1=a, 2=y, 3=z}
```

`sortedMapOf(vararg pairs: Pair<K, V>): SortedMap<K, V>`

创建一个根据Key升序排序好的TreeMap。对应的是使用Java中的SortedMap。

```
>>> val map = sortedMapOf(Pair("c", 3), Pair("b", 2), Pair("d", 1))
>>> map
{b=2, c=3, d=1}
```

5.5.3 访问Map的元素

entries属性

我们可以直接访问entries属性

```
val entries: Set<Entry<K, V>>
```

获取该Map中的所有键/值对的Set。这个Entry类型定义如下：

```
 public interface Entry<out K, out V> {
        public val key: K
        public val value: V
    }
```

代码示例

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map
{x=1, y=2, z=3}
>>> map.entries
[x=1, y=2, z=3]
```

这样，我们就可以遍历这个Entry的Set了：

```
>>> map.entries.forEach({println("key="+ it.key + " value=" + it.value)})
key=x value=1
key=y value=2
key=z value=3
```

keys属性

访问keys属性：

```
val keys: Set<K>
```

获取Map中的所有键的Set。

```
>>> map.keys
[x, y, z]
```

values属性

访问 `val values: Collection<V>` 获取Map中的所有值的Collection。这个值的集合可能包含重复值。

```
>>> map.values
[1, 2, 3]
```

size属性

访问 `val size: Int` 获取map键/值对的数目。

```
>>> map.size
3
```

get(key: K)

我们使用get函数来通过key来获取value的值。

```
operator fun get(key: K): V?
```

对应的操作符是 `[]` ：

```
>>> map["x"]
1
>>> map.get("x")
1
```

如果这个key不在Map中，就返回null。

```
>>> map["k"]
null
```

如果不想返回null，可以使用getOrDefault函数

```
getOrDefault(key: K, defaultValue: @UnsafeVariance V): V
```

当为null时，不返回null，而是返回设置的一个默认值：

```
>>> map.getOrDefault("k",0)
0
```

这个默认值的类型，要和V对应。类型不匹配会报错：

```
>>> map.getOrDefault("k","a")
error: type mismatch: inferred type is String but Int was expected
map.getOrDefault("k","a")
                      ^
```

5.5.4 Map操作符函数

`containsKey(key: K): Boolean`

是否包含该key。

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.containsKey("x")
true
>>> map.containsKey("j")
false
```

`containsValue(value: V): Boolean`

是否包含该value。

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.containsValue(2)
true
>>> map.containsValue(20)
false
```

`component1()` `component2()`

`Map.Entry<K, V>` 的操作符函数，分别用来直接访问key和value。

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.entries.forEach({println("key="+ it.component1() + " value=" + it.component2())
})
key=x value=1
key=y value=2
key=z value=3
```

这两个函数的定义如下：

```
@kotlin.internal.InlineOnly
public inline operator fun <K, V> Map.Entry<K, V>.component1(): K = key

@kotlin.internal.InlineOnly
public inline operator fun <K, V> Map.Entry<K, V>.component2(): V = value
```

`Map.Entry<K, V>.toPair(): Pair<K, V>`

把Map的Entry转换为Pair。

```
>>> map.entries
[x=1, y=2, z=3]
>>> map.entries.forEach({println(it.toPair())})
(x, 1)
(y, 2)
(z, 3)
```

`getOrElse(key: K, defaultValue: () -> V): V`

通过key获取值，当没有值可以设置默认值。

```
>>> val map = mutableMapOf<String, Int?>()
>>> map.getOrElse("x", { 1 })
1
>>> map["x"] = 3
>>> map.getOrElse("x", { 1 })
3
```

`getValue(key: K): V`

当Map中不存在这个key，调用get函数，如果不想返回null，直接抛出异常，可调用此方法。

```
val map = mutableMapOf<String, Int?>()
```

```
>>> map.get("v")
null
>>> map.getValue("v")
java.util.NoSuchElementException: Key v is missing in the map.
    at kotlin.collections.MapsKt__MapWithDefaultKt.getOrImplicitDefaultNullable(MapWith
Default.kt:19)
    at kotlin.collections.MapsKt__MapsKt.getValue(Maps.kt:252)
```

`getOrPut(key: K, defaultValue: () -> V): V`

如果不存在这个key，就添加这个key到Map中，对应的value是defaultValue。

```
>>> val map = mutableMapOf<String, Int?>()
>>> map.getOrPut("x", { 2 })
2
>>> map
{x=2}
```

`iterator(): Iterator<Map.Entry<K, V>>`

这个函数返回的是 `entries.iterator()` 。这样我们就可以像下面这样使用for循环来遍历Map：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3 )
>>> for((k,v) in map){println("key=$k, value=$v")}
key=x, value=1
key=y, value=2
key=z, value=3
```

`mapKeys(transform: (Map.Entry<K, V>) -> R): Map<R, V>`

把Map的Key设置为通过转换函数transform映射之后的值。

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val mmap = map.mapKeys{it.key * 10}
>>> mmap
{10=a, 20=b, 30=c, -10=z}
```

注意，这里的it是Map的Entry。 如果不巧，有任意两个key通过映射之后相等了，那么后面的key将会覆盖掉前面的key。

```
>>> val mmap = map.mapKeys{it.key * it.key}
>>> mmap
{1=z, 4=b, 9=c}
```

我们可以看出， `1 to "a"` 被 `-1 to "z"` 覆盖掉了。

`mapValues(transform: (Map.Entry<K, V>) -> R): Map<K, R>`

对应的这个函数是把Map的value设置为通过转换函数transform转换之后的新值。

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val mmap = map.mapValues({it.value + "$"})
>>> mmap
{1=a$, 2=b$, 3=c$, -1=z$}
```

`filterKeys(predicate: (K) -> Boolean): Map<K, V>`

返回过滤出满足key判断条件的元素组成的新Map。

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filterKeys({it>0})
{1=a, 2=b, 3=c}
```

注意，这里的it元素是Key。

`filterValues(predicate: (V) -> Boolean): Map<K, V>`

返回过滤出满足value判断条件的元素组成的新Map。

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filterValues({it>"b"})
{3=c, -1=z}
```

注意，这里的it元素是value。

`filter(predicate: (Map.Entry<K, V>) -> Boolean): Map<K, V>`

返回过滤出满足Entry判断条件的元素组成的新Map。

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filter({it.key>0 && it.value > "b"})
{3=c}
```

`Iterable<Pair<K, V>>.toMap(destination: M): M`

把持有Pair的Iterable集合转换为Map。

```
>>> val pairList = listOf(Pair(1,"a"),Pair(2,"b"),Pair(3,"c"))
>>> pairList
[(1, a), (2, b), (3, c)]
>>> pairList.toMap()
{1=a, 2=b, 3=c}
```

`Map<out K, V>.toMutableMap(): MutableMap<K, V>`

把一个只读的Map转换为可编辑的MutableMap。

```
>>> val map = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map[1]="x"
error: unresolved reference. None of the following candidates is applicable ...
error: no set method providing array access
map[1]="x"
   ^

>>> val mutableMap = map.toMutableMap()
>>> mutableMap
{1=a, 2=b, 3=c, -1=z}
>>> mutableMap[1]="x"
>>> mutableMap
{1=x, 2=b, 3=c, -1=z}
```

`plus` `minus`

Map的加法运算符函数如下：

```
operator fun <K, V> Map<out K, V>.plus(pair: Pair<K, V>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Iterable<Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Array<out Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Sequence<Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(map: Map<out K, V>): Map<K, V>
```

代码示例：

```
>>> val map = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map+Pair(10,"g")
{1=a, 2=b, 3=c, -1=z, 10=g}
>>> map + listOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + arrayOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + sequenceOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + mapOf(9 to "s", 10 to "w")
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
```

加并赋值函数：

```
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pair: Pair<K, V>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Iterable<Pair<K, V>
```

```
>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Array<out Pair<K, V
>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Sequence<Pair<K, V>
>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(map: Map<K, V>)
```

代码示例：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map+=Pair(10,"g")
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g}
>>> map += listOf(Pair(9,"s"),Pair(11,"w"))
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g, 9=s, 11=w}
>>> map += mapOf(20 to "qq", 30 to "tt")
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g, 9=s, 11=w, 20=qq, 30=tt}
```

减法跟加法类似。

`put(key: K, value: V): V?`

根据key设置元素的value。如果该key存在就更新value；不存在就添加，但是put的返回值是null。

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map
{1=a, 2=b, 3=c, -1=z}
>>> map.put(10,"q")
null
>>> map
{1=a, 2=b, 3=c, -1=z, 10=q}
>>> map.put(1,"f")
a
>>> map
{1=f, 2=b, 3=c, -1=z, 10=q}
```

`putAll(from: Map<out K, V>): Unit`

把一个Map全部添加到一个MutableMap中。

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val map2 = mapOf(99 to "aa", 100 to "bb")
>>> map.putAll(map2)
>>> map
{1=a, 2=b, 3=c, -1=z, 99=aa, 100=bb}
```

如果有key重复的，后面的值会覆盖掉前面的值：

```
>>> map
{1=a, 2=b, 3=c, -1=z, 99=aa, 100=bb}
>>> map.putAll(mapOf(1 to "www",2 to "tttt"))
>>> map
{1=www, 2=tttt, 3=c, -1=z, 99=aa, 100=bb}
```

`MutableMap<out K, V>.remove(key: K): V?`

根据键值key来删除元素。

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.remove(-1)
z
>>> map
{1=a, 2=b, 3=c}
>>> map.remove(100)
null
>>> map
{1=a, 2=b, 3=c}
```

`MutableMap<K, V>.clear(): Unit`

清空MutableMap。

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map
{1=a, 2=b, 3=c, -1=z}
>>> map.clear()
>>> map
{}
```

本章小结

本章我们介绍了Kotlin标准库中的集合类List、Set、Map，以及它们扩展的丰富的操作函数，这些函数使得我们使用这些集合类更加简单容易。

集合类持有的是对象，而怎样的放入正确的对象类型则是我们写代码过程中需要注意的。下一章节中我们将学习泛型。

第6章 泛型

6.1 泛型（Generic Type）简介

通常情况的类和函数，我们只需要使用具体的类型即可：要么是基本类型，要么是自定义的类。

但是尤其在集合类的场景下，我们需要编写可以应用于多种类型的代码，我们最简单原始的做法是，针对每一种类型，写一套刻板的代码。

这样做，代码复用率会很低，抽象也没有做好。

在SE 5种，Java引用了泛型。泛型，即“参数化类型”（Parameterized Type）。顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式，我们称之为类型参数，然后在使用时传入具体的类型（类型实参）。

我们知道，在数学中泛函是以函数为自变量的函数。类比的来理解，编程中的泛型就是以类型为变量的类型，即参数化类型。这样的变量参数就叫类型参数(Type Parameters)。

本章我们来一起学习一下Kotlin泛型的相关知识。

6.1.1 为什么要有类型参数

我们先来看下没有泛型之前，我们的集合类是怎样持有对象的。

在Java中，Object类是所有类的根类。为了集合类的通用性。我们把元素的类型定义为Object，当放入具体的类型的时候，再作强制类型转换。

这是一个示例代码：

```
class RawArrayList {
    public int length = 0;
    private Object[] elements;

    public RawArrayList(int length) {
        this.length = length;
        this.elements = new Object[length];
    }

    public Object get(int index) {
        return elements[index];
    }

    public void add(int index, Object element) {
        elements[index] = element;
    }
```

```
    @Override
    public String toString() {
        return "RawArrayList{" +
            "length=" + length +
            ", elements=" + Arrays.toString(elements) +
            '}';
    }
}
```

一个简单的测试代码如下：

```
public class RawTypeDemo {

    public static void main(String[] args) {
        RawArrayList rawArrayList = new RawArrayList(4);
        rawArrayList.add(0, "a");
        rawArrayList.add(1, "b");
        System.out.println(rawArrayList);

        String a = (String)rawArrayList.get(0);
        System.out.println(a);

        String b = (String)rawArrayList.get(1);
        System.out.println(b);

        rawArrayList.add(2, 200);
        rawArrayList.add(3, 300);
        System.out.println(rawArrayList);

        int c = (int)rawArrayList.get(2);
        int d = (int)rawArrayList.get(3);
        System.out.println(c);
        System.out.println(d);

        //Exception in thread "main" java.lang.ClassCastException: java.lang.Integer ca
nnot be cast to java.lang.String
        String x = (String)rawArrayList.get(2);
        System.out.println(x);

    }

}
```

我们可以看出，在使用原生态类型（raw type）实现的集合类中，我们使用的是Object[]数组。这种实现方式，存在的问题有两个：

1. 向集合中添加对象元素的时候，没有对元素的类型进行检查，也就是说，我们往集合中添加任意对象，编译器都不会报错。
2. 当我们从集合中获取一个值的时候，我们不能都使用Object类型，需要进行强制类型转换。而这个转换过程由于在添加元素的时候没有作任何的类型的限制跟检查，所以容易出错。例如上面代码中的：

```
String a = (String)rawArrayList.get(0);
```

对于这行代码，编译时不会报错，但是运行时会抛出类型转换错误。

由于我们不能笼统地把集合类中所有的对象是视作Object，然后在使用的时候各自作强制类型转换。因此，我们引入了类型参数来解决这个类型安全使用的问题。

Java 中的泛型是在1.5 之后加入的，我们可以为类和方法分别定义泛型参数，比如说Java中的Map接口的定义：

```
public interface Map<K,V> {
    ...
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
    default V getOrDefault(Object key, V defaultValue) {
        V v;
        return (((v = get(key)) != null) || containsKey(key))
            ? v
            : defaultValue;
    }
}
```

我们在Kotlin 中的写法基本一样：

```
public interface Map<K, out V> {
    ...
    public fun containsKey(key: K): Boolean
    public fun containsValue(value: @UnsafeVariance V): Boolean
    public operator fun get(key: K): V?
    @SinceKotlin("1.1")
    @PlatformDependent
    public fun getOrDefault(key: K, defaultValue: @UnsafeVariance V): V {
        // See default implementation in JDK sources
```

```
        return null as V
    }
    public val keys: Set<K>
    public val values: Collection<V>
    public val entries: Set<Map.Entry<K, V>>

}

public interface MutableMap<K, V> : Map<K, V> {
    public fun put(key: K, value: V): V?
    public fun remove(key: K): V?
    public fun putAll(from: Map<out K, V>): Unit
    ...

}
```

比如，在实例化一个Map时，我们使用这个函数：

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

类型参数K，V是一个占位符，当泛型类型被实例化和使用时，它将被一个实际的类型参数所替代。

代码示例

```
>>> val map = mutableMapOf<Int,String>(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
>>> map.put(4,"c")
null
>>> map
{1=a, 2=b, 3=c, 4=c}
```

mutableMapOf表示参数化类型分别是Int 和 String，这是泛型类型集合的实例化，在这里，放置K， V 的位置被具体的Int 和 String 类型所替代。

泛型主要是用来限制集合类持有的对象类型，这样使得类型更加安全。当我们在一个集合类里面放入了错误类型的对象，编译器就会报错：

```
>>> map.put("5","e")
error: type mismatch: inferred type is String but Int was expected
map.put("5","e")
        ^
```

Kotlin中有类型推断的功能，有些类型参数可以直接省略不写。上面的mapOf后面的类型参数可以省掉不写：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
```

Java和Kotlin 的泛型实现，都是采用了运行时类型擦除的方式。也就是说，在运行时，这些类型参数的信息将会被擦除。Java 和Kotlin 的泛型对于语法的约束是在编译期。

6.2 型变（Variance）

6.2.1 Java的类型通配符

Java 泛型的通配符有两种形式。我们使用

- 子类型上界限定符 `? extends T` 指定类型参数的上限（该类型必须是类型T或者它的子类型）
- 超类型下界限定符 `? super T` 指定类型参数的下限（该类型必须是类型T或者它的父类型）

我们称之为类型通配符(Type Wildcard)。默认的上界（如果没有声明）是 `Any?` ，下界是Nothing。

代码示例：

```
class Animal {

    public void act(List<? extends Animal> list) {
        for (Animal animal : list) {
            animal.eat();
        }
    }

    public void aboutShepherdDog(List<? super ShepherdDog> list) {
        System.out.println("About ShepherdDog");
    }

    public void eat() {
        System.out.println("Eating");
    }

}

class Dog extends Animal {}

class Cat extends Animal {}
```

```
class ShepherdDog extends Dog {}
```

我们在方法 `act(List<? extends Animal> list)` 中, 这个list可以传入以下类型的参数：

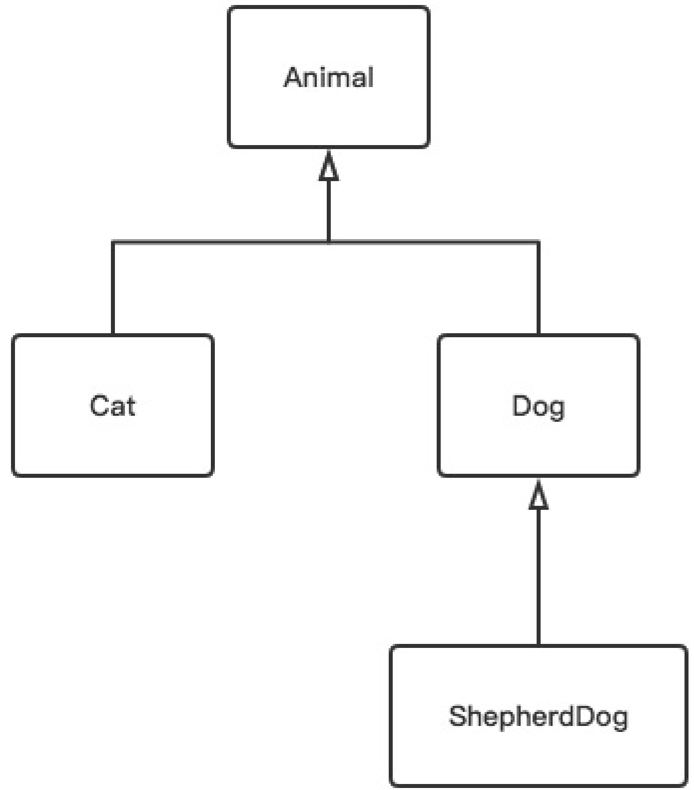
```
List<Animal>
List<Dog>
List<ShepherdDog>
List<Cat>
```

测试代码：

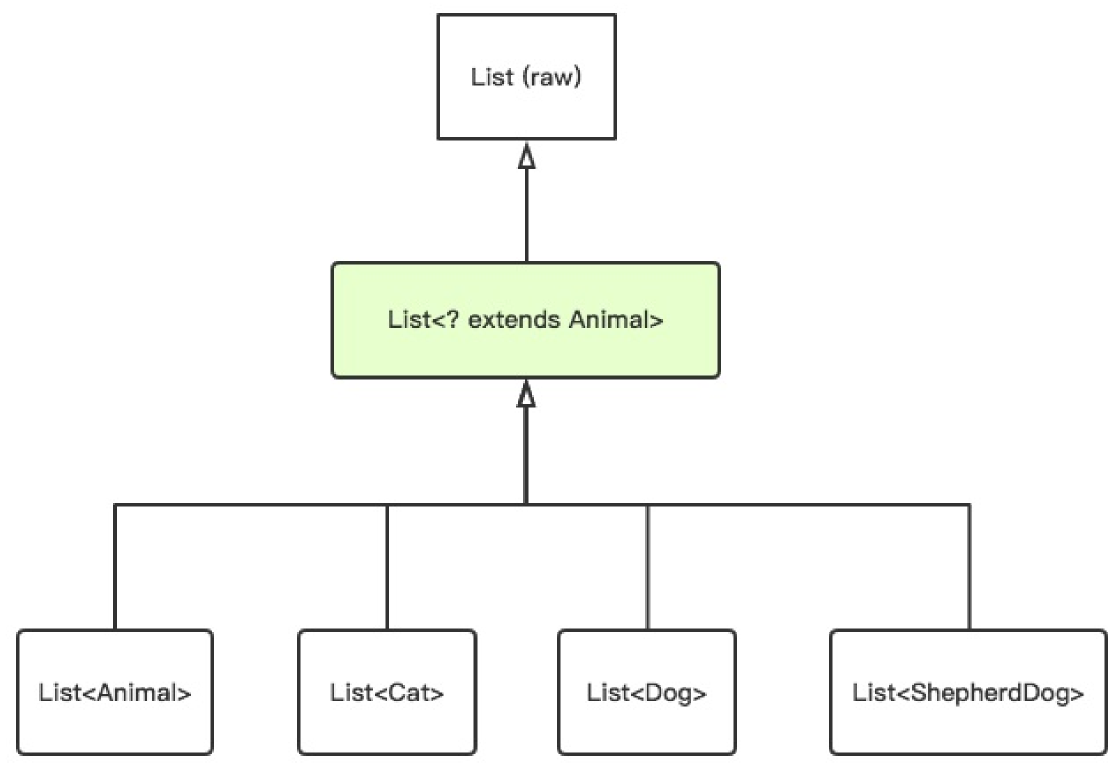
```
List<Animal> list3 = new ArrayList<>();
list3.add(new Dog());
list3.add(new Cat());
animal.act(list3);

List<Dog> list4 = new ArrayList<>();
list4.add(new Dog());
list4.add(new Dog());
animal.act(list4);

List<Cat> list5 = new ArrayList<>();
list5.add(new Cat());
list5.add(new Cat());
animal.act(list5);
```

为了更加简单明了说明这些类型的层次关系，我们图示如下：

对象层次类图：



集合类泛型层次类图：



也就是说，List并不是List的子类型，而是两种不存在父子关系的类型。

而 `List<? extends Animal>` 是 `List<Animal>` ， `List<Dog>` 等的父类型，对于任何的 `List<X>` 这里的 `X` 只要是Animal的子类型，那么 `List<? extends Animal>` 就是 `List<X>` 的父类型。

使用通配符 `List<? extends Animal>` 的引用, 我们不可以往这个List中添加Animal类型以及其子类型的元素：

```
List<? extends Animal> list1 = new ArrayList<>();

list1.add(new Dog());
list1.add(new Animal());
```

这样的写法，Java编译器是不允许的。

```
17
18        List<? extends Animal> list1 = new ArrayList<>();
19
20        list1.add(new Dog());
21        list1.add(new Animal());
```

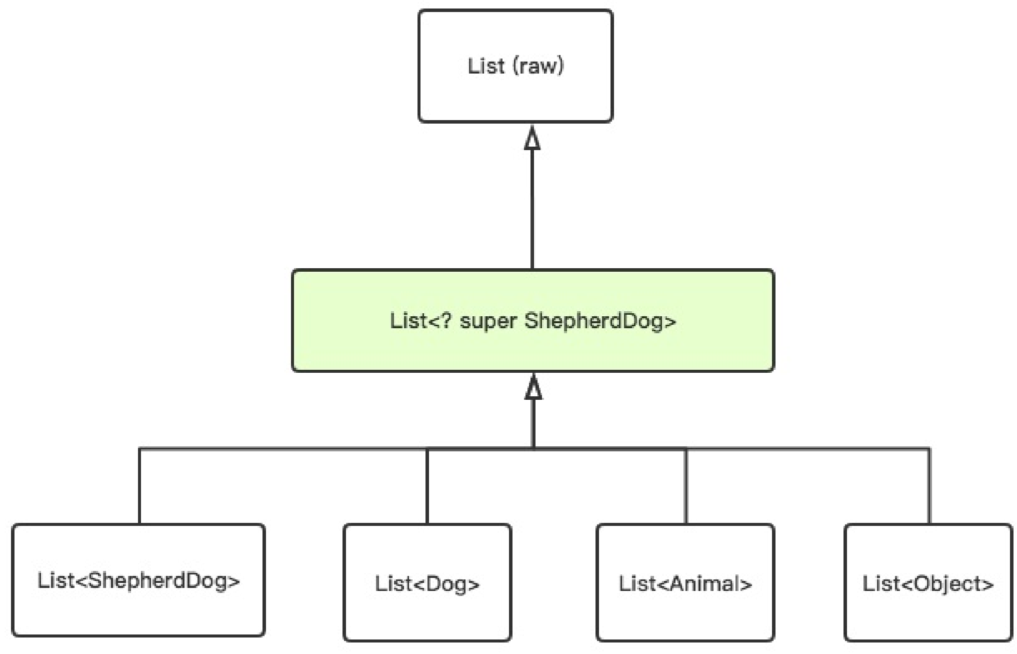
add (capture<? extends com.easy.kotlin.Animal>) in List cannot be applied to (com.easy.kotlin.Animal)

因为对于set方法，编译器无法知道具体的类型，所以会拒绝这个调用。但是，如果是get方法形式的调用，则是允许的：

```
List<? extends Animal> list1 = new ArrayList<>();
List<Dog> list4 = new ArrayList<>();
list4.add(new Dog());
list4.add(new Dog());
animal.act(list4);
list1 = list4;
animal.act(list1);
```

我们这里把引用变量 `List<? extends Animal> list1` 直接赋值 `List<Dog> list4`，因为编译器知道可以把返回对象转换为一个Animal类型。

相应的，`? super T` 超类型限定符的变量类型 `List<? super ShepherdDog>` 的层次结构如下：



在Java中，还有一个无界通配符，即单独一个 `?`。如 `List<?>`，`?` 可以代表任意类型，“任意”是未知类型。例如:

```
Pair<?>
```

参数替换后的Pair类有如下方法:

```
? getFirst()
```

```
void setFirst(?)
```

我们可以调用getFirst方法，因为编译器可以把返回值转换为Object。 但是不能调用setFirst方法，因为编译器无法确定参数类型。

通配符在类型系统中具有重要的意义，它们为一个泛型类所指定的类型集合提供了一个有用的类型范围。泛型参数表明的是在类、接口、方法的创建中，要使用一个数据类型参数来代表将来可能会用到的一种具体的数据类型。它可以是Integer类型，也可以是String类型。我们通常把它的类型定义成 E、T 、K 、V等等。

当我们在实例化对象的时候，必须声明T具体是一个什么类型。所以当我们把T定义成一个确定的泛型数据类型，参数就只能是这种数据类型。此时，我们就用到了通配符代替指定的泛型数据类型。

如果把一个对象分为声明、使用两部分的话。泛型主要是侧重于类型的声明的代码复用，通配符则侧重于使用上的代码复用。泛型用于定义内部数据类型的参数化，通配符则用于定义使用的对象类型的参数化。

使用泛型、通配符提高了代码的复用性。同时对象的类型得到了类型安全的检查，减少了类型转换过程中的错误。

6.2.2 协变（covariant）与逆变（contravariant）

在Java中数组是协变的，下面的代码是可以正确编译运行的：

```
Integer[] ints = new Integer[3];
ints[0] = 0;
ints[1] = 1;
ints[2] = 2;
Number[] numbers = new Number[3];
numbers = ints;
for (Number n : numbers) {
    System.out.println(n);
}
```

在Java中，因为 Integer 是 Number 的子类型，数组类型 Integer[] 也是 Number[] 的子类型，因此在任何需要 Number[] 值的地方都可以提供一个 Integer[] 值。

而另一方面，泛型不是协变的。也就是说， List 不是 List 的子类型，试图在要求 List 的位置提供 List 是一个类型错误。下面的代码，编译器是会直接报错的：

```
List<Integer> integerList = new ArrayList<>();
integerList.add(0);
integerList.add(1);
integerList.add(2);
List<Number> numberList = new ArrayList<>();
```

```
numberList = integerList;
```

编译器报错提示如下：

```
List<Integer> integerList = new ArrayList<>();
integerList.add(0);
integerList.add(1);
integerList.add(2);
List<Number> numberList = new ArrayList<>();
numberList = integerList;
```

Incompatible types.
Required: List <java.lang.Number>
Found: List <java.lang.Integer>

Java中泛型和数组的不同行为，的确引起了许多混乱。

就算我们使用通配符，这样写：

```
List<? extends Number> list = new ArrayList<Number>();
list.add(new Integer(1)); //error
```

仍然是报错的：

```
List<? extends Number> list = new ArrayList<Number>();
list.add(new Integer( value: 1)); //error
```

add (capture<? extends java.lang.Number>) in List cannot be applied to (java.lang.Integer)

为什么Number的对象可以由Integer实例化，而ArrayList的对象却不能由ArrayList实例化？list中的<? extends Number>声明其元素是Number或Number的派生类，为什么不能add Integer?为了解决这些问题，需要了解Java中的逆变和协变以及泛型中通配符用法。

逆变与协变

Animal类型（简记为F, Father）是Dog类型（简记为C, Child）的父类型，我们把这种父子类型关系简记为F <| C。

而List, List的类型，我们分别简记为f(F), f(C)。

那么我们可以这么来描述协变和逆变：

> 当F <| C 时, 如果有f(F) <| f(C),那么f叫做协变（Convariant）；当F <| C 时, 如果有f(C) <| f(F), 那么f叫做逆变（Contravariance）。如果上面两种关系都不成立则叫做不可变。

协变和逆协变都是类型安全的。

Java中泛型是不变的，可有时需要实现逆变与协变，怎么办呢？这时就需要使用我们上面讲的通配符 `?` 。

`<? extends T>` 实现了泛型的协变

```
List<? extends Number> list = new ArrayList<>();
```

这里的 `? extends Number` 表示的是Number类或其子类，我们简记为C。

这里 `C <| Number` ，这个关系成立： `List<C> <| List< Number >` 。即有：

```
List<? extends Number> list1 = new ArrayList<Integer>();
List<? extends Number> list2 = new ArrayList<Float>();
```

但是这里不能向list1、list2添加除null以外的任意对象。

```
list1.add(null);
list2.add(null);

list1.add(new Integer(1)); // error
list2.add(new Float(1.1f)); // error
```

因为，List可以添加Interger及其子类，List可以添加Float及其子类，List、List都是 `List<? extends Number>` 的子类型，如果能将Float的子类添加到 `List<? extends Number>` 中，那么也能将Integer的子类添加到 `List<? extends Number>` 中，那么这时候 `List<? extends Number>` 里面将会持有各种Number子类型的对象（Byte，Integer，Float，Double等等）。Java为了保护其类型一致，禁止向List<? extends Number>添加任意对象，不过可以添加null。

```
List<? extends Number> list1 = new ArrayList<Integer>();
List<? extends Number> list2 = new ArrayList<Float>();

list1.add(null);
list2.add(null);

list1.add(new Integer( value: 1));
list2.add(new Float( value: 1.1f));
```

add (capture<? extends java.lang.Number>) in List cannot be applied to (java.lang.Float)

`<? super T>` 实现了泛型的逆变

```
List<? super Number> list = new ArrayList<>();
```

`? super Number` 通配符则表示的类型下界为Number。即这里的父类型F是 `? super Number` ，子类型C是Number。即当F <| C，有f(C) <| f(F)， 这就是逆变。代码示例：

```
List<? super Number> list3 = new ArrayList<Number>();
List<? super Number> list4 = new ArrayList<Object>();
list3.add(new Integer(3));
list4.add(new Integer(4));
```

也就是说，我们不能往 `List<? super Number >` 中添加Number的任意父类对象。但是可以向List<? super Number >添加Number及其子类对象。

PECS

现在问题来了：我们什么时候用extends什么时候用super呢？《Effective Java》给出了答案：

> PECS: producer-extends, consumer-super

比如，一个简单的Stack API：

```
public class Stack<E>{
    public Stack();
    public void push(E e):
    public E pop();
    public boolean isEmpty();
}
```

要实现pushAll(Iterable src)方法，将src的元素逐一入栈：

```
public void pushAll(Iterable<E> src){
    for(E e : src)
        push(e)
}
```

假设有一个实例化Stack的对象stack，src有Iterable与 Iterable；

在调用pushAll方法时会发生type mismatch错误，因为Java中泛型是不可变的，Iterable与 Iterable都不是Iterable的子类型。

因此，应改为

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)   // out T, 从src中读取数据，producer-extends
        push(e);
}
```

要实现popAll(Collection dst)方法，将Stack中的元素依次取出add到dst中，如果不用通配符实现：

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

同样地，假设有一个实例化Stack的对象stack，dst为Collection；

调用popAll方法是会发生type mismatch错误，因为Collection不是Collection的子类型。

因而，应改为：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());   // in T, 向dst中写入数据， consumer-super
}
```

Naftalin与Wadler将PECS称为 **Get and Put Principle**。

在 `java.util.Collections` 的 `copy` 方法中(JDK1.7)完美地诠释了PECS：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");

    if (srcSize < COPY_THRESHOLD ||
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
            dest.set(i, src.get(i));
    } else {
        ListIterator<? super T> di=dest.listIterator();   // in T, 写入dest数据
        ListIterator<? extends T> si=src.listIterator();   // out T， 读取src数据
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

6.3 Kotlin的泛型特色

正如上文所讲的，在 Java 泛型里，有通配符这种东西，我们要用 `? extends T` 指定类型参数的上限，用 `? super T` 指定类型参数的下限。

而Kotlin 抛弃了这个东西，引用了生产者和消费者的概念。也就是我们前面讲到的PECS。生产者就是我们去读取数据的对象，消费者则是我们要写入数据的对象。这两个概念理解起来有点绕。

我们用代码示例简单讲解一下：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        ...
        ListIterator<? super T> di=dest.listIterator();   // in T, 写入dest数据
        ListIterator<? extends T> si=src.listIterator();   // out T,  读取src数据
         ...
}
```

`List<? super T> dest` 是消费数据的对象，这些数据会写入到该对象中，这些数据该对象被“吃掉”了（Kotlin中叫 `in T`）。

`List<? extends T> src` 是生产提供数据的对象。这些数据哪里来的呢？就是通过src读取获得的（Kotlin中叫 `out T`）。

6.3.1 `out T` 与 `in T`

在Kotlin中，我们把那些只能保证读取数据时类型安全的对象叫做生产者，用 `out T` 标记；把那些只能保证写入数据安全时类型安全的对象叫做消费者，用 `in T` 标记。

如果你觉得太晦涩难懂，就这么记吧：

> `out T` 等价于 `? extends T` `in T` 等价于 `? super T` 此外, 还有 `*` 等价于 `?`

6.3.2 声明处型变

Kotlin 泛型中添加了声明处型变。看下面的例子：

```
interface Source<out T> {
    fun <T> nextT();
}
```

我们在接口的声明处用 `out T` 做了生产者声明以实现安全的类型协变：

```
fun demo(str: Source<String>) {
    val obj: Source<Any> = str // 合法的类型协变
}
```

Kotlin 中有大量的声明处协变，比如 Iterable 接口的声明：

```
public interface Iterable<out T> {
    public operator fun iterator(): Iterator<T>
}
```

因为 Collection 接口和 Map 接口都继承了 Iterable 接口，而 Iterable 接口被声明为生产者接口，所以所有的 Collection 和 Map 对象都可以实现安全的类型协变：

```
val c: List<Number> = listOf(1, 2, 3)
```

这里的 listOf() 函数返回 `List<Int>` 类型，因为 List 接口实现了安全的类型协变，所以可以安全地把 `List<Int>` 类型赋给 `List<Number>` 类型变量。

6.3.3 类型投影

将类型参数 T 声明为 out 非常方便，并且能避免使用处子类型化的麻烦，但是有些类实际上**不能**限制为只返回 `T`。

一个很好的例子是 Array：

```
class Array<T>(val size: Int) {
    fun get(index: Int): T {  }
    fun set(index: Int, value: T) {  }
}
```

该类在 `T` 上既不能是协变的也不能是逆变的。这造成了一些不灵活性。考虑下述函数：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

这个函数应该将项目从一个数组复制到另一个数组。如果我们采用如下方式使用这个函数：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any) // 错误：期望 (Array<Any>, Array<Any>)
```

这里我们将遇到同样的问题：`Array <T>` 在 `T` 上是**不型变的**，因此 `Array <Int>` 和 `Array <Any>` 都不是另一个的子类型。

那么，我们唯一要确保的是 `copy()` 不会做任何坏事。我们阻止它写到 `from`，我们可以：

```
fun copy(from: Array<out Any>, to: Array<Any>) {}
```

现在这个 `from` 是一个受 `Array<out Any>` 限制的（**投影的**）数组。在Kotlin中，称为**类型投影**(type projection)。其主要作用是参数作限定，避免不安全操作。

类似的，我们也可以使用 **in** 投影一个类型：

```
fun fill(dest: Array<in String>, value: String) {}
```

`Array<in String>` 对应于 Java 的 `Array<? super String>`，也就是说，我们可以传递一个 `CharSequence` 数组或一个 `Object` 数组给 `fill()` 函数。

类似Java中的无界类型通配符 `?`，Kotlin 也有对应的**星投影**语法 `*`。

例如，如果类型被声明为 `interface Function <in T, out U>`，我们有以下星投影：

- `Function<*, String>` 表示 `Function<in Nothing, String>`；
- `Function<Int, *>` 表示 `Function<Int, out Any?>`；
- `Function<*, *>` 表示 `Function<in Nothing, out Any?>`。

`*` 投影跟 Java 的原始类型类似，不过是安全的。

6.6 泛型类

声明一个泛型类

```
class Box<T>(t: T) {
    var value = t
}
```

通常，要创建这样一个类的实例，我们需要指定类型参数:

```
val box: Box<Int> = Box<Int>(1)
```

但是，如果类型参数可以通过推断得到，比如，通过构造器参数类型，或通过其他手段推断得到，此时允许省略类型参数:

```
val box = Box(1) // 1 的类型为 Int, 因此编译器知道我们创建的实例是 Box<Int> 类型
```

6.5 泛型函数

类可以有类型参数。函数也有。类型参数要放在函数名称之前：

```
fun <T> singletonList(item: T): List<T> {}
fun <T> T.basicToString() : String {  // 扩展函数
```

```
}
```

要调用泛型函数，在函数名后指定类型参数即可：

```
val l = singletonList<Int>(1)
```

泛型函数与其所在的类是否是泛型没有关系。泛型函数独立于其所在的类。我们应该尽量使用泛型方法，也就是说如果使用泛型方法可以取代将整个类泛型化，那么就应该只使用泛型方法，因为它可以使事情更明白。

本章小结

泛型是一个非常有用的东西。尤其在集合类中。我们可以发现大量的泛型代码。

本章我们通过对Java泛型的回顾，对比介绍了Kotlin泛型的特色功能，尤其是协变、逆变、`in`、`out`等概念，需要我们深入去理解。只有深入理解了这些概念，我们才能更好理解并用好Kotlin的集合类，进而写出高质量的泛型代码。

泛型实现是依赖OOP中的类型多态机制的。Kotlin是一门支持面向对象编程（OOP）跟函数式编程（FP）强大的语言。我们已经学习了Kotlin的语言基础知识、类型系统、集合类、泛型等相关知识了，相信您已经对Kotlin有了一个初步的了解。

在下一章节中，我们将一起来学习Kotlin的面向对象编程相关的知识。

第7章 面向对象编程（OOP）

在前面的章节中，我们学习了Kotlin的语言基础知识、类型系统、集合类以及泛型相关的知识。在本章节以及下一章中，我们将一起来学习Kotlin对面向对象编程以及函数式编程的支持。

7.1 面向对象编程思想

7.1.1 一切皆是映射

《易传·系辞上传》：“易有太极，是生两仪，两仪生四象，四象生八卦。” 如今的互联网世界，其基石却是01（阴阳），不得不佩服我华夏先祖的博大精深的智慧。

> 一切皆是映射

计算机领域中的所有问题,都可以通过向上一层进行抽象封装来解决.这里的封装的本质概念，其实就是“映射”。

就好比通过的电子电路中的电平进行01逻辑映射，于是有了布尔代数，数字逻辑电路系统；

对01逻辑的进一步封装抽象成CPU指令集映射，诞生了汇编语言；

通过汇编语言的向上抽象一层编译解释器，于是有了pascal，fortran，C语言；再对核心函数api进行封装形成开发包（Development Kit), 于是有了Java，C++ 。

从面向过程到面向对象，再到设计模式，架构设计，面向服务，Sass/Pass/Iass等等的思想，各种软件理论思想五花八门，但万变不离其宗——

- 你要解决一个怎样的问题？
- 你的问题领域是怎样的？
- 你的模型（数据结构）是什么？
- 你的算法是什么？
- 你对这个世界的本质认知是怎样的？
- 你的业务领域的逻辑问题，流程是什么？等等。

> Grady Booch：我对OO编程的目标从来就不是复用。相反，对我来说，对象提供了一种处理复杂性的方式。这个问题可以追溯到亚里士多德：您把这个世界视为过程还是对象？在OO兴起运动之前，编程以过程为中心--例如结构化设计方法。然而，系统已经到达了超越其处理能力的复杂性极点。有了对象，我们能够通过提升抽象级别来构建更大的、更复杂的系统--我认为，这才是面向对象编程运动的真正胜利。

最初， 人们使用物理的或逻辑的二进制机器指令来编写程序， 尝试着表达思想中的逻辑， 控制硬件计算和显示， 发现是可行的；

接着， 创造了助记符 —— 汇编语言， 比机器指令更容易记忆；

再接着，创造了编译器、解释器和计算机高级语言，能够以人类友好自然的方式去编写程序，在牺牲少量性能的情况下，获得比汇编语言更强且更容易使用的语句控制能力：条件、分支、循环，以及更多的语言特性：指针、结构体、联合体、枚举等，还创造了函数，能够将一系列指令封装成一个独立的逻辑块反复使用；

逐渐地，产生了面向过程的编程方法；

后来，人们发现将数据和逻辑封装成对象，更接近于现实世界，且更容易维护大型软件，又出现了面向对象的编程语言和编程方法学，增加了新的语言特性：继承、多态、模板、异常错误。

为了不必重复开发常见工具和任务，人们创造和封装了容器及算法、SDK，垃圾回收器，甚至是并发库；

为了让计算机语言更有力更有效率地表达各种现实逻辑，消解软件开发中遇到的冲突，还在语言中支持了元编程、高阶函数，闭包 等有用特性。

为了更高效率地开发可靠的软件和应用程序，人们逐渐构建了代码编辑器、IDE、代码版本管理工具、公共库、应用框架、可复用组件、系统规范、网络协议、语言标准等，针对遇到的问题提出了许多不同的思路和解决方案，并总结提炼成特定的技术和设计模式，还探讨和形成了不少软件开发过程，用来保证最终发布的软件质量。尽管编写的这些软件和工具还存在不少 BUG，但是它们都“奇迹般地存活”，并共同构建了今天蔚为壮观的互联网时代的电商，互联网金融，云计算，大数据，物联网，机器智能等等的“虚拟世界”。

7.1.2 二进制01与易经阴阳

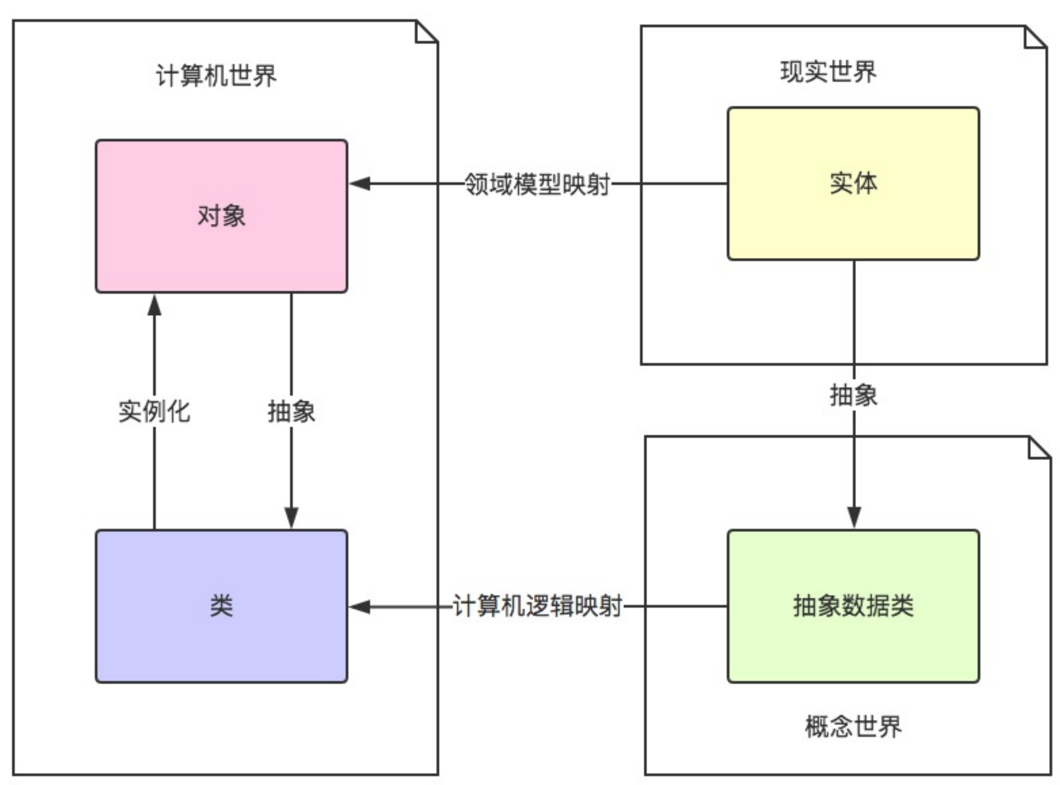
二进制数是用0和1两个数码来表示的数。它的基数为2，进位规则是“逢二进一”，借位规则是“借一当二”，由18世纪德国数理哲学大师莱布尼兹发现。当前的计算机系统使用的基本上是二进制系统。

19世纪爱尔兰逻辑学家B对逻辑命题的思考过程转化为对符号0，1的某种代数演算，二进制是逢2进位的进位制。0、1是基本算符。因为它只使用0、1两个数字符号，非常简单方便，易于用电子方式实现。

二进制的发现直接导致了电子计算器和计算机的发明，并让计算机得到了迅速的普及，进入各行各业，成为人类生活和生产的重要工具。

二进制的实质是通过两个数字“0”和“1”来描述事件。在人类的生产、生活等许多领域，我们可以通过计算机来虚拟地描述现实中存在的事件，并能通过给定的条件和参数模拟事件变化的规律。二进制的计算机几乎是万能的，能将我们生活的现实世界完美复制，并且还能根据我们人类给定的条件模拟在现实世界难以实现的各种实验。

但是，不论计算机能给我们如何多变、如何完美、如何复杂的画面，其本源只是简单的“0”和“1”。“0”和“1”在计算机中通过不同的组合与再组合，模拟出一个纷繁复杂、包罗万象的虚拟世界。我们简单图示如下：



二进制的“0”和“1”通过计算机里能够创造出一个虚拟的、纷繁的世界。自然界中的阴阳形成了现实世界的万事万物。

所以自然世界的“阴”“阳”作为基础切实地造就了复杂的现实世界，计算机的“0”和“1”形象地模拟现实世界的一切现象，易学中的“卦”和“阴阳爻”抽象地揭示了自然界存在的事件和其变化规律。

所以说，编程的本质跟大自然创造万物的本质是一样的。

7.1.3 从面向过程到面向对象

从IBM公司的约翰·巴库斯在1957年开发出世界上第一个高级程序设计语言Fortran至今，高级程序设计语言的发展已经经历了整整半个世纪。在这期间，程序设计语言主要经历了从面向过程（如C和Pascal语言）到面向对象（如C++和Java语言），再到面向组件编程（如.NET平台下的C#语言），以及面向服务架构技术（如SOA、Service以及最近很火的微服务架构）等。

面向过程编程

> 结构化编程思想的核心：功能分解（自顶向下，逐层细化）。

1971年4月份的 Communications of ACM上，尼古拉斯·沃斯（Niklaus Wirth，1934年2月15日—, 结构化编程思想的创始人。因发明了Euler、Alogo-W、Modula和Pascal等一系列优秀的编程语言并提出了结构化编程思想而在1984年获得了图灵奖。）发表了论文“通过逐步求精方式开发程

序’ （Program Development by Stepwise Refinement）， 首次提出 “结构化程序设计” （structure programming）的概念。

不要求一步就编制成可执行的程序，而是分若干步进行，逐步求精。

第一步编出的程序抽象度最高，第二步编出的程序抽象度有所降低…… 最后一步编出的程序即为可执行的程序。

用这种方法编程，似乎复杂，实际上优点很多，可使程序易读、易写、易调试、易维护、易保证其正确性及验证其正确性。

结构化程序设计方法又称为 “自顶向下” 或 “逐步求精” 法，在程序设计领域引发了一场革命，成为程序开发的一个标准方法，尤其是在后来发展起来的软件工程中获得广泛应用。有人评价说Wirth的结构化程序设计概念 “完全改变了人们对程序设计的思维方式” ，这是一点也不夸张的。

尼古拉斯· 沃思教授在编程界提出了一个著名的公式：

> 程序 = 数据结构 + 算法

面向对象编程

> 面向对象编程思想的核心：应对变化，提高复用。

阿伦·凯（Alan Kay）：面向对象编程思想的创始人。2003年因在面向对象编程上所做的巨大贡献而获得图灵奖。

> The best way to predict the future is to invent it，预测未来最好的方法是创造它！（Alan Kay）

阿伦·凯是Smalltalk面向对象编程语言的发明人之一，也是面向对象编程思想的创始人之一，同时，他还是笔记本电脑最早的构想者和现代Windows GUI的建筑师。最早提出PC概念和互联网的也是阿伦·凯，所以人们都尊称他为 “预言大师” 。他是当今IT界屈指可数的技术天才级人物。

面向对象编程思想主要是复用性和灵活性（弹性）。复用性是面向对象编程的一个主要机制。灵活性主要是应对变化的特性，因为客户的需求是不断改变的，怎样适应客户需求的变化，这是软件设计灵活性或者说是弹性的问题。

Java是一种面向对象编程语言，它基于Smalltalk语言，作为OOP语言，它具有以下五个基本特性：

1.万物皆对象，每一个对象都会存储数据，并且可以对自身执行操作。因此，每一个对象包含两部分：成员变量和成员方法。在成员方法中可以改变成员变量的值。

2.程序是对象的集合，他们通过发送消息来告知彼此所要做的事情，也就是调用相应的成员函数。

3.每一个对象都有自己的由其他对象所构成的存储，也就是说在创建新对象的时候可以在成员变量中使用已存在的对象。

4.每个对象都拥有其类型，每个对象都是某个类的一个实例，每一个类区别于其它类的特性就是可以向它发送什么类型的消息，也就是它定义了哪些成员函数。

5.某一个特定类型的所有对象都可以接受同样的消息。另一种对对象的描述为：对象具有状态(数据，成员变量)、行为(操作，成员方法)和标识(成员名，内存地址)。

面向对象语言其实是对现实生活中的实物的抽象。

每个对象能够接受的请求(消息)由对象的接口所定义，而在程序中必须由满足这些请求的代码，这段代码称之为这个接口的实现。当向某个对象发送消息(请求)时，这个对象便知道该消息的目的(该方法的实现已定义)，然后执行相应的代码。

我们经常说一些代码片段是优雅的或美观的，实际上意味着它们更容易被人类有限的思维所处理。

对于程序的复合而言，好的代码是它的表面积要比体积增长的慢。

代码块的 “表面积” 是是我们复合代码块时所需要的信息（接口API协议定义）。代码块的 “体积” 就是接口内部的实现逻辑（API背后的实现代码）。

在面向对象编程中，一个理想的对象应该是只暴露它的抽象接口（纯表面， 无体积），其方法则扮演箭头的角色。如果为了理解一个对象如何与其他对象进行复合，当你发现不得不深入挖掘对象的实现之时，此时你所用的编程范式的原本优势就荡然无存了。

面向组件和面向服务

- 面向组件

我们知道面向对象支持重用，但是重用的单元很小，一般是类；而面向组件则不同，它可以重用多个类甚至一个程序。也就是说面向组件支持更大范围内的重用，开发效率更高。如果把面向对象比作重用零件，那么面向组件则是重用部件。

- 面向服务

将系统进行功能化，每个功能提供一种服务。现在非常流行微服务MicroService技术以及SOA（面向服务架构）技术。

> 面向过程（Procedure）→面向对象（Object）→ 面向组件（Component） →面向服务（Service）

正如解决数学问题通常我们会谈 “思想” ，诸如反证法、化繁为简等，解决计算机问题也有很多非常出色的思想。思想之所以称为思想，是因为 “思想” 有拓展性与引导性，可以解决一系列问题。

解决问题的复杂程度直接取决于抽象的种类及质量。过将结构、性质不同的底层实现进行封装，向上提供统一的API接口，让使用者觉得就是在使用一个统一的资源，或者让使用者觉得自己在使用一个本来底层不直接提供、 “虚拟” 出来的资源。

计算机中的所有问题 , 都可以通过向上抽象封装一层来解决。同样的,任何复杂的问题, 最终总能够回归最本质,最简单。

面向对象编程是一种自顶向下的程序设计方法。万事万物都是对象,对象有其行为(方法),状态(成员变量,属性)。OOP是一种编程思想，而不是针对某个语言而言的。当然，语言影响思维方式，思维依赖语言的表达，这也是辩证的来看。

所谓“面向对象语言”，其实经典的“过程式语言”（比如Pascal，C），也能体现面向对象的思想。所谓“类”和“对象”，就是C语言里面的抽象数据类型结构体（struct）。

而面向对象的多态是唯一相比struct多付出的代价，也是最重要的特性。这就是SmallTalk、Java这样的面向对象语言所提供的特性。

回到一个古老的话题：程序是什么？

在面向对象的编程世界里，下面的这个公式

> 程序 = 算法 + 数据结构

可以简单重构成：

> 程序 = 基于对象操作的算法 + 以对象为最小单位的数据结构

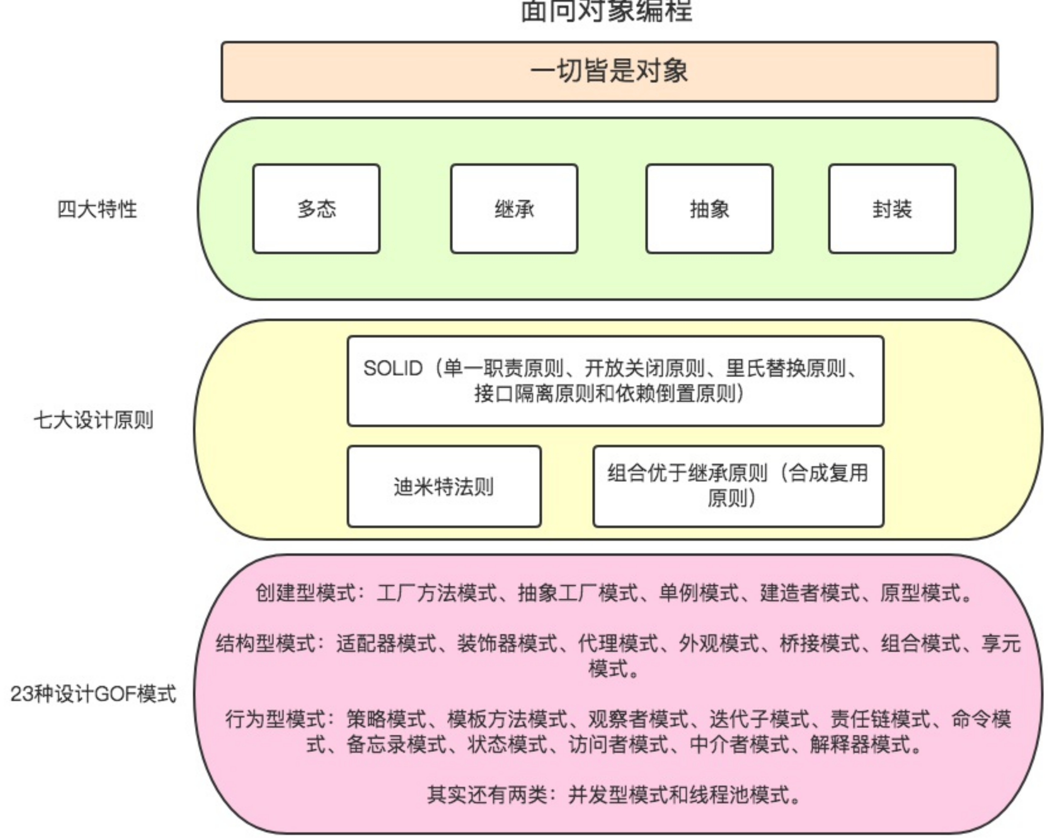
封装总是为了减少操作粒度，数据结构上的封装导致了数据的减少，自然减少了问题求解的复杂度；对代码的封装使得代码得以复用，减少了代码的体积，同样使问题简化。这个时候，算法操作的就是一个抽象概念的集合。

在面向对象的程序设计中，我们便少不了集合类容器。容器就用来存放一类有共同抽象概念的东西。这里说有共同概念的东西（而没有说对象），其实，就是我们上一个章节中讲到的泛型。这样对于一个通用的算法，我们就可以最大化的实现复用，作用于的集合。

面向对象的本质就是让对象有多态性，把不同对象以同一特性来归组，统一处理。至于所谓继承、虚表、等等概念，只是其实现的细节。

在遵循这些面向对象设计原则基础上，前辈们总结出一些解决不同问题场景的设计模式，以GOF的23中设计模式最为知名。

我们用一幅图简单概括一下面向对象编程的知识框架：



讲了这么多思考性的思想层面的东西，我们下面来开始Kotlin的面向对象编程的学习。Kotlin对面向对象编程是完全支持的。

7.2 类与构造函数

Kotlin和Java很相似，也是一种面向对象的语言。下面我们来一起学习Kotlin的面向对象的特性。如果您熟悉Java或者C++、C#中的类，您可以很快上手。同时，您也将看到Kotlin与Java中的面向对象编程的一些不同的特性。

Kotlin中的类和接口跟Java中对应的概念有些不同，比如接口可以包含属性声明；Kotlin的类声明，默认是final和public的。

另外，嵌套类并不是默认在内部的。它们不包含外部类的隐式引用。

在构造函数方面，Kotlin简短的主构造函数在大多数情况下都可以满足使用，当然如果有稍微复杂的初始化逻辑，我们也可以声明次级构造函数来完成。

我们还可以使用 data 修饰符来声明一个数据类，使用 object 关键字来表示单例对象、伴生对象等。

Kotlin类的成员可以包含：

- 构造函数和初始化块
- 属性

- 函数
- 嵌套类和内部类
- 对象声明

等。

7.2.1 声明类

和大部分语言类似，Kotlin使用class作为类的关键字，当我们声明一个类时，直接通过class加类名的方式来实现：

```
class World
```

这样我们就声明了一个World类。

7.2.2 构造函数

在 Kotlin 中，一个类可以有一个

- 主构造函数（primary constructor）和一个或多个
- 次构造函数（secondary constructor）。

主构造函数

主构造函数是类头的一部分，直接放在类名后面：

```
open class Student constructor(var name: String, var age: Int) : Any() {
...
}
```

如果主构造函数没有任何注解或者可见性修饰符，可以省略这个 constructor 关键字。如果构造函数有注解或可见性修饰符，这个 constructor 关键字是必需的，并且这些修饰符在它前面：

```
annotation class MyAutowired

class ElementaryStudent public @MyAutowired constructor(name: String, age: Int) : Stude
nt(name, age) {
...
}
```

与普通属性一样，主构造函数中声明的属性可以是可变的（var）或只读的（val）。

主构造函数不能包含任何的代码。初始化的代码可以放到以 init 关键字作为前缀的初始化块（initializer blocks）中：

```
open class Student constructor(var name: String, var age: Int) : Any() {

    init {
        println("Student{name=$name, age=$age} created!")
    }
   ...

}
```

主构造的参数可以在初始化块中使用,也可以在类体内声明的属性初始化器中使用。

次构造函数

在类体中，我们也可以声明前缀有 constructor的次构造函数，次构造函数不能有声明 val 或 var ：

```
class MiddleSchoolStudent {
    constructor(name: String, age: Int) {
    }
}
```

如果类有一个主构造函数，那么每个次构造函数需要委托给主构造函数， 委托到同一个类的另一个构造函数用 this 关键字即可：

```
class ElementarySchoolStudent public @MyAutowired constructor(name: String, age: Int) :
 Student(name, age) {
    override var weight: Float = 80.0f

    constructor(name: String, age: Int, weight: Float) : this(name, age) {
        this.weight = weight
    }

    ...
}
```

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造函数。构造函数的可见性是 public。

私有主构造函数

我们如果希望这个构造函数是私有的，我们可以如下声明：

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

这样我们在代码中，就无法直接使用主构造函数来实例化这个类，下面的写法是不允许的：

```
val dontCreateMe = DontCreateMe() // cannot access it
```

但是，我们可以通过次构造函数引用这个私有主构造函数来实例化对象：

7.2.2 类的属性

我们再给这个World类加入两个属性。我们可能直接简单地写成：

```
class World1 {
    val yin: Int
    val yang: Int
}
```

在Kotlin中，直接这样写语法上是会报错的：

```
class World1 {
    val yin: Int
    val yang: Int
Property must be initialized or be abstract
```

意思很明显，是说这个类的属性必须要初始化，或者如果不初始化那就得是抽象的abstract属性。

我们把这两个属性都给初始化如下：

```
class World1 {
    val yin: Int = 0
    val yang: Int = 1
}
```

我们再来使用测试代码来看下访问这两个属性的方式：

```
>>> class World1 {
...     val yin: Int = 0
...     val yang: Int = 1
... }
>>> val w1 = World1()
>>> w1.yin
0
>>> w1.yang
1
```

上面的World1类的代码，在Java中等价的写法是：

```
public final class World1 {
   private final int yin;
```

```
    private final int yang = 1;

    public final int getYin() {
        return this.yin;
    }

    public final int getYang() {
        return this.yang;
    }
}
```

我们可以看出，Kotlin中的类的字段自动带有getter方法和setter方法。而且写起来比Java要简洁的多。

7.2.3 函数（方法）

我们再来给这个World1类中加上一个函数：

```
class World2 {
    val yin: Int = 0
    val yang: Int = 1

    fun plus(): Int {
        return yin + yang
    }
}


val w2 = World2()
println(w2.plus()) // 输出 1
```

7.3 抽象类

7.3.1 抽象类的定义

含有抽象函数的类(这样的类需要使用abstract修饰符来声明)，称为抽象类。

下面是一个抽象类的例子：

```
abstract class Person(var name: String, var age: Int) : Any() {

    abstract var addr: String
    abstract val weight: Float

    abstract fun doEat()
```

```
    abstract fun doWalk()

    fun doSwim() {
        println("I am Swimming ... ")
    }

    open fun doSleep() {
        println("I am Sleeping ... ")
    }
}
```

7.3.2 抽象函数

在上面的这个抽象类中，不仅可以有抽象函数 `abstract fun doEat()` `abstract fun doWalk()` ，同时可以有具体实现的函数 `fun doSwim()` ， 这个函数默认是final的。也就是说，我们不能重写这个doSwim函数：

```
56  class Teacher(name: String, age: Int) : Person(name, age) {
57      override var addr: String = "HangZhou"
58      override val weight: Float = 100.0f
59
60      override fun doEat() {
61          println("Teacher is Eating ... ")
62      }
63
64      override fun doWalk() {
65          println("Teacher is Walking ... ")
66      }
67
68      override fun doSleep() {
69          super.doSleep()
70          println("Teacher is Sleeping ... ")
71      }
72
73      override fun doSwim() {
74          println("Teacher is Swimming ... ")
'doSwim' in 'Person' is final and cannot be overridden
```

如果一个函数想要设计成能被重写，例如 `fun doSleep()` ,我们给它加上open关键字即可。然后，我们就可以在子类中重写这个 `open fun doSleep()` ：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String = "HangZhou"
    override val weight: Float = 100.0f

    override fun doEat() {
        println("Teacher is Eating ... ")
    }

    override fun doWalk() {
        println("Teacher is Walking ... ")
```

```
    }

    override fun doSleep() {
        super.doSleep()
        println("Teacher is Sleeping ... ")
    }

//    override fun doSwim() { // cannot be overriden
//        println("Teacher is Swimming ... ")
//    }
}
```

抽象函数是一种特殊的函数：它只有声明，而没有具体的实现。抽象函数的声明格式为：

```
abstract fun doEat()
```

关于抽象函数的特征，我们简单总结如下：

- 抽象函数必须用abstract关键字进行修饰
- 抽象函数不用手动添加open，默认被open修饰
- 抽象函数没有具体的实现
- 含有抽象函数的类成为抽象类，必须由abtract关键字修饰。抽象类中可以有具体实现的函数，这样的函数默认是final（不能被覆盖重写），如果想要重写这个函数，给这个函数加上open关键字。

7.3.3 抽象属性

抽象属性就是在var或val前被abstract修饰，抽象属性的声明格式为：

```
abstract var addr : String
abstract val weight : Float
```

关于抽象属性，需要注意的是：

1. 抽象属相在抽象类中不能被初始化
2. 如果在子类中没有主构造函数，要对抽象属性手动初始化。如果子类中有主构造函数，抽象属性可以在主构造函数中声明。

综上所述，抽象类和普通类的区别有：

1.抽象函数必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public。

也就是说，这三个函数

```
abstract fun doEat()
```

```
abstract fun doWalk()

fun doSwim() {
    println("I am Swimming ... ")
}
```

默认的都是public的。

另外抽象类中的具体实现的函数，默认是final的。上面的三个函数，等价的Java的代码如下：

```
public abstract void doEat();

public abstract void doWalk();

public final void doSwim() {
   String var1 = "I am Swimming ... ";
   System.out.println(var1);
}
```

2.抽象类不能用来创建对象实例。也就是说，下面的写法编译器是不允许的：

```
val p = Person("Jack", "28")
```

Cannot create an instance of an abstract class

3.如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。实现父类抽象函数，我们使用override关键字来表明是重写函数：

```
class Programmer(override var addr: String, override val weight: Float, name: String, a
ge: Int) : Person(name, age) {
    override fun doEat() {
        println("Programmer is Eating ... ")
    }

    override fun doWalk() {
        println("Programmer is Walking ... ")
    }
}
```

如果子类没有实现父类的抽象函数，则必须将子类也定义为为abstract类。例如：

```
abstract class Writer(override var addr: String, override val weight: Float, name: Stri
ng, age: Int) : Person(name, age) {
    override fun doEat() {
        println("Programmer is Eating ... ")
    }
```

```
    abstract override fun doWalk();
}
```

doWalk函数没有实现父类的抽象函数，那么我们在子类中把它依然定义为抽象函数。相应地这个子类，也成为了抽象子类，需要使用abstract关键字来声明。

如果抽象类中含有抽象属性，再实现子类中必须将抽象属性初始化，除非子类也为抽象类。例如我们声明一个Teacher类继承Person类：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String //  error， 需要初始化，或者声明为abstract
    override val weight: Float //  error， 需要初始化，或者声明为abstract
    ...
}
```

这样写，编译器会直接报错：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String
    override val weight: Float
Property must be initialized or be abstract
    }

    override fun doWalk() {
        println("Teacher is Walking ... ")
    }
}
```

解决方法是，在实现的子类中，我们将抽象属性初始化即可：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String = "HangZhou"
    override val weight: Float = 100.0f

    override fun doEat() {
        println("Teacher is Eating ... ")
    }

    override fun doWalk() {
        println("Teacher is Walking ... ")
    }
}
```

7.4 接口

7.4.1 接口定义

和Java类似，Kotlin使用interface作为接口的关键词：

```
interface ProjectService
```

Kotlin 的接口与 Java 8 的接口类似。与抽象类相比，他们都可以包含抽象的方法以及方法的实现:

```
interface ProjectService {
    val name: String
    val owner: String
    fun save(project: Project)
    fun print() {
        println("I am project")
    }
}
```

7.4.2 实现接口

接口是没有构造函数的。我们使用冒号 `:` 语法来实现一个接口，如果有多个用 `,` 逗号隔开：

```
class ProjectServiceImpl : ProjectService
class ProjectMilestoneServiceImpl : ProjectService, MilestoneService
```

我们也可以实现多个接口：

```
class Project

class Milestone

interface ProjectService {
    val name: String
    val owner: String
    fun save(project: Project)
    fun print() {
        println("I am project")
    }
}

interface MilestoneService {
    val name: String
    fun save(milestone: Milestone)
    fun print() {
        println("I am Milestone")
    }
}
```

```
class ProjectMilestoneServiceImpl : ProjectService, MilestoneService {
    override val name: String
        get() = "ProjectMilestone"
    override val owner: String
        get() = "Jack"

    override fun save(project: Project) {
        println("Save Project")
    }

    override fun print() {
//          super.print()
        super<ProjectService>.print()
        super<MilestoneService>.print()
    }

    override fun save(milestone: Milestone) {
        println("Save Milestone")
    }
}
```

当子类继承了某个类之后，便可以使用父类中的成员变量，但是并不是完全继承父类的所有成员变量。具体的原则如下：

1.能够继承父类的public和protected成员变量；不能够继承父类的private成员变量；

2.对于父类的包访问权限成员变量，如果子类和父类在同一个包下，则子类能够继承；否则，子类不能够继承；

3.对于子类可以继承的父类成员变量，如果在子类中出现了同名称的成员变量，则会发生隐藏现象，即子类的成员变量会屏蔽掉父类的同名成员变量。如果要在子类中访问父类中同名成员变量，需要使用super关键字来进行引用。

7.4.3 覆盖冲突

在kotlin中， 实现继承通常遵循如下规则：如果一个类从它的直接父类继承了同一个函数的多个实现，那么它必须重写这个函数并且提供自己的实现(或许只是直接用了继承来的实现) 为表示使用父类中提供的方法我们用 super 表示。

在重写 `print()` 时，因为我们实现的ProjectService、MilestoneService都有一个 `print()` 函数，当我们直接使用 `super.print()` 时，编译器是无法知道我们想要调用的是那个里面的print函数的，这个我们叫做覆盖冲突：

```
59    override fun print() {
60        super.print()
```

Many supertypes available, please specify the one you mean in angle brackets, e.g. 'super<Foo>'

这个时候，我们可以使用下面的语法来调用：

```
super<ProjectService>.print()
super<MilestoneService>.print()
```

7.4.4 接口中的属性

在接口中声明的属性，可以是抽象的，或者是提供访问器的实现。

在企业应用中，大多数的类型都是无状态的，如：Controller、ApplicationService、DomainService、Repository等。

因为接口没有状态， 所以它的属性是无状态的。

```
interface MilestoneService {
    val name: String // 抽象的
    val owner: String get() = "Jack" // 访问器

    fun save(milestone: Milestone)
    fun print() {
        println("I am Milestone")
    }
}


class MilestoneServiceImpl : MilestoneService {
    override val name: String
        get() = "MilestoneServiceImpl name"


    override fun save(milestone: Milestone) {
        println("save Milestone")
    }
}
```

7.5 抽象类和接口的差异

概念上的区别

接口主要是对动作的抽象，定义了行为特性的规约。 抽象类是对根源的抽象。当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

语法层面上的区别

接口不能保存状态，可以有属性但必须是抽象的。 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

类如果要实现一个接口，它必须要实现接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

设计层面上的区别

抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

继承是 `is a` 的关系，而 接口实现则是 `has a` 的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现就不需要有这层类型关系。

设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。也就是说：

- 对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的实现，子类可以不进行变更；
- 而对于接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。

实际应用上的差异

在实际使用中，使用抽象类(也就是继承)，是一种强耦合的设计，用来描述 `A is a B` 的关系，即如果说A继承于B，那么在代码中将A当做B去使用应该完全没有问题。比如在Android中，各种控件都可以被当做View去处理。

如果在你设计中有两个类型的关系并不是 `is a` ，而是 `is like a` ，那就必须慎重考虑继承。因为一旦我们使用了继承，就要小心处理好子类跟父类的耦合依赖关系。组合优于继承。

7.6 继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

在Kotlin中，所有的类会默认继承Any这个父类，但Any并不完全等同于java中的Object类，因为它只有equals(),hashCode()和toString()这三个方法。

7.6.1 open类

除了抽象类、接口默认可以被继承（实现）外，我们也可以把一个类声明为open的，这样我们就可以继承这个open类。

当我们想定义一个父类时，需要使用open关键字:

```
open class Base{
}
```

当然，抽象类是默认open的。

然后在子类中使用冒号 `：` 进行继承

```
class SubClass : Base(){
}
```

如果父类有构造函数，那么必须在子类的主构造函数中进行继承，没有的话则可以选择主构造函数或二级构造函数

```
//父类
open class Base(type:String){

}

//子类
class SubClass(type:String) : Base(type){

}
```

Kotlin中的 `override` 重写和java中也有所不同，因为Kotlin提倡所有的操作都是明确的，因此需要将希望被重写的函数设为open:

```
open fun doSomething() {}
```

然后通过override标记实现重写

```
override fun doSomething() {
 super.doSomething()
 }
```

同样的，抽象函数以及接口中定义的函数默认都是open的。

override重写的函数也是open的，如果希望它不被重写，可以在前面增加final :

```
open class SubClass : Base{
 constructor(type:String) : super(type){
 }

 final override fun doSomething() {
 super.doSomething()
 }
}
```

7.6.2 多重继承

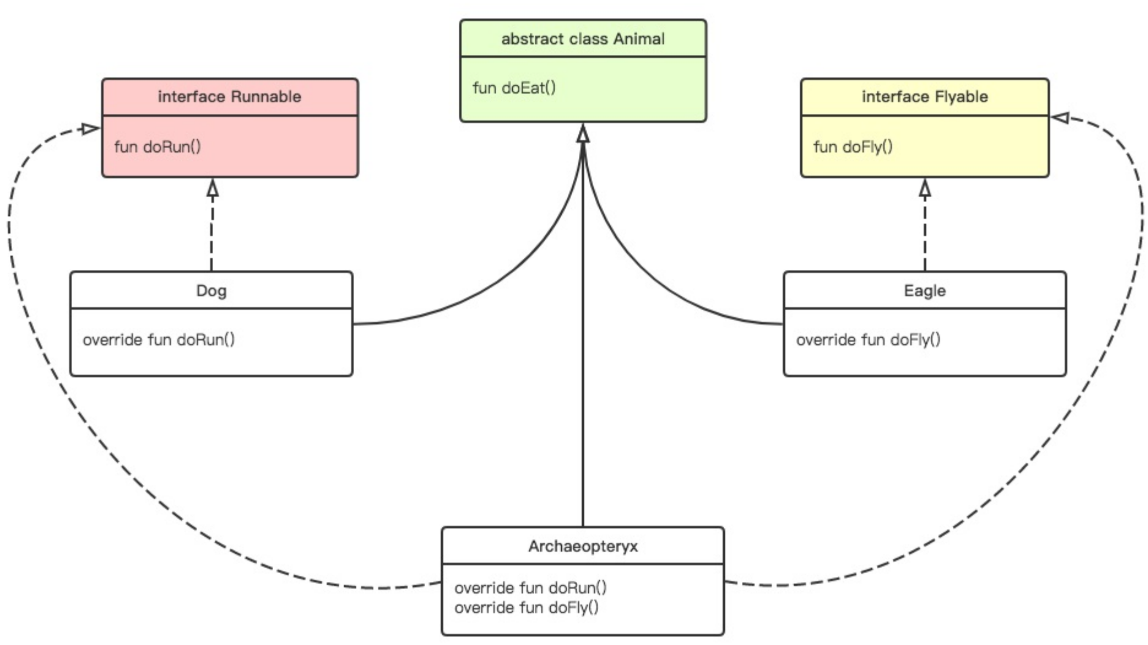
有些编程语言支持一个类拥有多个父类，例如C++。 我们将这个特性称之为多重继承（multiple inheritance）。多重继承会有二义性和钻石型继承树（DOD：Diamond Of Death）的复杂性问题。Kotlin跟Java一样，没有采用多继承，任何一个子类仅允许一个父类存在，而在多继承的问题场景下，使用实现多个interface 组合的方式来实现多继承的功能。

代码示例：

```
package com.easy.kotlin

abstract class Animal {
    fun doEat() {
        println("Animal Eating")
    }
}


abstract class Plant {
    fun doEat() {
        println("Plant Eating")
    }
}


interface Runnable {
    fun doRun()
}

interface Flyable {
    fun doFly()
}

class Dog : Animal(), Runnable {
    override fun doRun() {
        println("Dog Running")
    }
}

class Eagle : Animal(), Flyable {
    override fun doFly() {
        println("Eagle Flying")
    }
}

// 始祖鸟, 能飞也能跑
class Archaeopteryx : Animal(), Runnable, Flyable {
    override fun doRun() {
```

```
        println("Archaeopteryx Running")
    }

    override fun doFly() {
        println("Archaeopteryx Flying")
    }

}

fun main(args: Array<String>) {
    val d = Dog()
    d.doEat()
    d.doRun()

    val e = Eagle()
    e.doEat()
    e.doFly()

    val a = Archaeopteryx()
    a.doEat()
    a.doFly()
    a.doRun()
}
```

上述代码类之间的关系，我们用图示如下：



我们可以看出，Archaeopteryx继承了Animal类，用了父类doEat()函数功能；实现了Runnable接口，拥有了doRun()函数规范；实现了Flyable接口，拥有了doFly()函数规范。

在这里，我们通过实现多个接口，组合完成了的多个功能，而不是设计多个层次的复杂的继承关系。

7.7 枚举类

Kotlin的枚举类定义如下：

```
public abstract class Enum<E : Enum<E>>(name: String, ordinal: Int): Comparable<E> {
    companion object {}

    public final val name: String
    public final val ordinal: Int

    public override final fun compareTo(other: E): Int
    protected final fun clone(): Any

    public override final fun equals(other: Any?): Boolean
    public override final fun hashCode(): Int
    public override fun toString(): String
}
```

我们可以看出，这个枚举类有两个属性：

```
public final val name: String
public final val ordinal: Int
```

分别表示的是枚举对象的值跟下标位置。

同时，我们可以看出枚举类还实现了Comparable接口。

7.7.1 枚举类基本用法

枚举类的最基本的用法是实现类型安全的枚举：

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

>>> val north = Direction.NORTH
>>> north.name
NORTH
>>> north.ordinal
0
>>> north is Direction
true
```

每个枚举常量都是一个对象。枚举常量用逗号分隔。

7.7.2 初始化枚举值

我们可以如下初始化枚举类的值：

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}

>>> val red = Color.RED
>>> red.rgb
16711680
```

另外，枚举常量也可以声明自己的匿名类:

```
enum class ActivtyLifeState {
    onCreate {
        override fun signal() = onStart
    },

    onStart {
        override fun signal() = onStop
    },

    onStop {
        override fun signal() = onStart
    },

    onDestroy {
        override fun signal() = onDestroy
    };

    abstract fun signal(): ActivtyLifeState
}

>>> val s = ActivtyLifeState.onCreate
>>> println(s.signal())
onStart
```

7.7.3 使用枚举常量

我们使用enumValues()函数来列出枚举的所有值：

```
@SinceKotlin("1.1")
public inline fun <reified T : Enum<T>> enumValues(): Array<T>
```

每个枚举常量，默认都 `name` 名称和 `ordinal` 位置的属性(这个跟Java的Enum类里面的类似)：

```
val name: String
val ordinal: Int
```

代码示例：

```
enum class RGB { RED, GREEN, BLUE }

>>> val rgbs = enumValues<RGB>().joinToString { "${it.name} : ${it.ordinal} " }
>>> rgbs
RED : 0 , GREEN : 1 , BLUE : 2
```

我们直接声明了一个简单枚举类，我们使用遍历函数 `enumValues<RGB>()` 列出了RGB枚举类的所有枚举值。使用 `it.name` `it.ordinal` 直接访问各个枚举值的名称和位置。

另外，我们也可以自定义枚举属性值：

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

>>> val colors = enumValues<Color>().joinToString { "${it.rgb} : ${it.name} : ${it.ordi
nal} " }
>>> colors
16711680 : RED : 0 , 65280 : GREEN : 1 , 255 : BLUE : 2
```

然后，我们可以直接使用 `it.rgb` 访问属性名来得到对应的属性值。

7.8 注解类

Kotlin 的注解与 Java 的注解完全兼容。

7.8.1 声明注解

```
annotation class 注解名
```

代码示例：

```
@Target(AnnotationTarget.CLASS,
        AnnotationTarget.FUNCTION,
        AnnotationTarget.EXPRESSION,
        AnnotationTarget.FIELD,
        AnnotationTarget.LOCAL_VARIABLE,
        AnnotationTarget.TYPE,
        AnnotationTarget.TYPEALIAS,
        AnnotationTarget.TYPE_PARAMETER,
        AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicClass


@Target(AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicFunction


@Target(AnnotationTarget.CONSTRUCTOR)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicConstructor
```

在上面的代码中，我们通过向注解类添加元注解(meta-annotation)的方法来指定其他属性：

- @Target ：指定这个注解可被用于哪些元素(类, 函数, 属性, 表达式, 等等.);
- @Retention ：指定这个注解的信息是否被保存到编译后的 class 文件中, 以及在运行时是否可以通过反 射访问到它；
- @Repeatable：允许在单个元素上多次使用同一个注解；
- @MustBeDocumented ： 表示这个注解是公开 API 的一部分, 在自动产生的 API 文档的类或者函数签名中, 应该包含这个注解的信息。

这几个注解定义在 `kotlin/annotation/Annotations.kt` 类中。

7.8.2 使用注解

注解可以用在类、函数、参数、变量（成员变量、局部变量）、表达式、类型上等。这个由该注解的元注解@Target定义。

```
@MagicClass class Foo @MagicConstructor constructor() {
```

```
    constructor(index: Int) : this() {
        this.index = index
    }

    @MagicClass var index: Int = 0
    @MagicFunction fun magic(@MagicClass name: String) {

    }
}
```

注解在主构造器上，主构造器必须加上关键字 “constructor”

```
@MagicClass class Foo @MagicConstructor constructor() {
...
}
```

7.9 单例模式(Singleton)与伴生对象(companion object)

7.9.1 单例模式(Singleton)

单例模式很常用。它是一种常用的软件设计模式。例如，Spring中的Bean默认就是单例。通过单例模式可以保证系统中一个类只有一个实例。即一个类只有一个对象实例。

我们用Java实现一个简单的单例类的代码如下：

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

测试代码：

```
Singleton singleton1 = Singleton.getInstance();
```

可以看出，我们先在单例类中声明了一个私有静态的 `Singleton instance` 变量，然后声明一个私有构造函数 `private Singleton() {}` ，这个私有构造函数使得外部无法直接通过new的方式来构建对象：

```
Singleton singleton2 = new Singleton(); //error, cannot private access
```

最后提供一个public的获取当前类的唯一实例的静态方法 `getInstance()` 。我们这里给出的是一个简单的单例类，是线程不安全的。

7.9.2 object对象

Kotlin中没有 **静态属性和方法**，但是也提供了实现**类似于单例的功能**，我们可以使用关键字 `object` 声明一个object对象：

```
object AdminUser {
    val username: String = "admin"
    val password: String = "admin"
    fun getTimestamp() = SimpleDateFormat("yyyyMMddHHmmss").format(Date())
    fun md5Password() = EncoderByMd5(password + getTimestamp())
}
```

测试代码：

```
val adminUser = AdminUser.username
val adminPassword = AdminUser.md5Password()
println(adminUser)  // admin
println(adminPassword)  // g+0yLfaPVYxUf6TMIdXFXw==，这个值具体运行时会变化
```

为了方便在REPL中演示说明，我们再写一个示例代码：

```
>>> object User {
...     val username: String = "admin"
...     val password: String = "admin"
... }
```

object对象只能通过对象名字来访问：

```
>>> User.username
admin
>>> User.password
admin
```

不能像下面这样使用构造函数：

```
>>> val u = User()
error: expression 'User' of type 'Line130.User' cannot be invoked as a function. The fu
nction 'invoke()' is not found
val u = User()
        ^
```

为了更加直观的了解object对象的概念，我们把上面的 `object User` 的代码反编译成Java代码：

```
public final class User {
   @NotNull
   private static final String username = "admin";
   @NotNull
   private static final String password = "admin";
   public static final User INSTANCE;

   @NotNull
   public final String getUsername() {
      return username;
   }

   @NotNull
   public final String getPassword() {
      return password;
   }

   private User() {
      INSTANCE = (User)this;
      username = "admin";
      password = "admin";
   }

   static {
      new User();
   }
}
```

从上面的反编译代码，我们可以直观了解Kotlin的object背后的一些原理。

7.9.3 嵌套（Nested）object对象

这个object对象还可以放到一个类里面：

```
class DataProcessor {
    fun process() {
        println("Process Data")
    }
```

```
    object FileUtils {
        val userHome = "/Users/jack/"

        fun getFileContent(file: String): String {
            var content = ""
            val f = File(file)
            f.forEachLine { content = content + it + "\n" }
            return content
        }

    }
}
```

测试代码：

```
DataProcessor.FileUtils.userHome // /Users/jack/
DataProcessor.FileUtils.getFileContent("test.data") // 输出文件的内容
```

同样的，我们只能通过类的名称来直接访问object，不能使用对象实例引用。下面的写法是错误的：

```
val dp = DataProcessor()
dp.FileUtils.userHome // error, Nested object FileUtils cannot access object via refere
nce
```

我们在Java中通常会写一些Utils类，这样的类我们在Kotlin中就可以直接使用object对象：

```
object HttpUtils {
    val client = OkHttpClient()

    @Throws(Exception::class)
    fun getSync(url: String): String? {
        val request = Request.Builder()
                .url(url)
                .build()

        val response = client.newCall(request).execute()
        if (!response.isSuccessful()) throw IOException("Unexpected code " + response)

        val responseHeaders = response.headers()
        for (i in 0..responseHeaders.size() - 1) {
            println(responseHeaders.name(i) + ": " + responseHeaders.value(i))
        }
        return response.body()?.string()
```

```
    }

    @Throws(Exception::class)
    fun getAsync(url: String) {
        var result: String? = ""

        val request = Request.Builder()
                .url(url)
                .build()
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException?) {
                e?.printStackTrace()
            }

            @Throws(IOException::class)
            override fun onResponse(call: Call, response: Response) {
                if (!response.isSuccessful()) throw IOException("Unexpected code " + re
sponse)

                val responseHeaders = response.headers()
                for (i in 0..responseHeaders.size() - 1) {
                    println(responseHeaders.name(i) + ": " + responseHeaders.value(i))
                }
                result = response.body()?.string()
                println(result)

            }
        })
    }
}
```

测试代码：

```
val url = "http://www.baidu.com"
val html1 = HttpUtils.getSync(url) // 同步get
println("html1=${html1}")
HttpUtils.getAsync(url) // 异步get
```

7.9.4 匿名object

还有，在代码行内，有时候我们需要的仅仅是一个简单的对象，我们这个时候就可以使用下面的匿名object的方式：

```
fun distance(x: Double, y: Double): Double {
    val porigin = object {
        var x = 0.0
```

```
        var y = 0.0
    }
    return Math.sqrt((x - porigin.x) * (x - porigin.x) + (y - porigin.y) * (y - porigin
.y))
}
```

测试代码：

```
distance(3.0, 4.0)
```

需要注意的是，匿名对象只可以用在本地和私有作用域中声明的类型。代码示例：

```
class AnonymousObjectType {
    // 私有函数，返回的是匿名object类型
    private fun privateFoo() = object {
        val x: String = "x"
    }

    // 公有函数，返回的类型是 Any
    fun publicFoo() = object {
        val x: String = "x" // 无法访问到
    }

    fun test() {
        val x1 = privateFoo().x        // Works
        //val x2 = publicFoo().x  // ERROR: Unresolved reference 'x'
    }
}


fun main(args: Array<String>) {
    AnonymousObjectType().publicFoo().x // Unresolved reference 'x'
}
```

跟 Java 匿名内部类类似，object对象表达式中的代码可以访问来自包含它的作用域的变量（与 Java 不同的是，这不限于 final 变量）：

```
fun countCompare() {
    var list = mutableListOf(1, 4, 3, 7, 11, 9, 10, 20)
    var countCompare = 0
    Collections.sort(list, object : Comparator<Int> {
        override fun compare(o1: Int, o2: Int): Int {
            countCompare++
            println("countCompare=$countCompare")
            println(list)
            return o1.compareTo(o2)
```

```
        }
    })
}
```

测试代码：

```
countCompare()

countCompare=1
[1, 4, 3, 7, 11, 9, 10, 20]
...
countCompare=17
[1, 3, 4, 7, 9, 10, 11, 20]
```

7.9.5 伴生对象(companion object)

Kotlin中还提供了 **伴生对象** ，用 `companion object` 关键字声明:

```
class DataProcessor {
    fun process() {
        println("Process Data")
    }


    object FileUtils {
        val userHome = "/Users/jack/"

        fun getFileContent(file: String): String {
            var content = ""
            val f = File(file)
            f.forEachLine { content = content + it + "\n" }
            return content
        }

    }

    companion object StringUtils {
        fun isEmpty(s: String): Boolean {
            return s.isEmpty()
        }
    }

}
```

一个类只能有1个伴生对象。也就是是下面的写法是错误的：

```
class ClassA {
    companion object Factory {
        fun create(): ClassA = ClassA()
    }

    companion object Factory2 { // error, only 1 companion object is allowed per class
        fun create(): MyClass = MyClass()
    }
}
```

一个类的伴生对象默认引用名是Companion:

```
class ClassB {
    companion object {
        fun create(): ClassB = ClassB()
        fun get() = "Hi, I am CompanyB"
    }
}
```

我们可以直接像在Java静态类中使用静态方法一样使用一个类的伴生对象的函数，属性(但是在运行时，它们依旧是实体的实例成员)：

```
ClassB.Companion.index
ClassB.Companion.create()
ClassB.Companion.get()
```

其中， Companion可以省略不写：

```
ClassB.index
ClassB.create()
ClassB.get()
```

当然，我们也可以指定伴生对象的名称：

```
class ClassC {
    var index = 0
    fun get(index: Int): Int {
        return 0
    }

    companion object CompanyC {
        fun create(): ClassC = ClassC()
        fun get() = "Hi, I am CompanyC"
    }
}
```

测试代码：

```
ClassC.index
ClassC.create()// com.easy.kotli.ClassC@7440e464，具体运行值会变化
ClassC.get() // Hi, I am CompanyC
ClassC.CompanyC.index
ClassC.CompanyC.create()
ClassC.CompanyC.get()
```

伴生对象的初始化是在相应的类被加载解析时，与 Java 静态初始化器的语义相匹配。

即使伴生对象的成员看起来像其他语言的静态成员，在运行时他们仍然是真实对象的实例成员。而且，还可以实现接口：

```
interface BeanFactory<T> {
    fun create(): T
}


class MyClass {
    companion object : BeanFactory<MyClass> {
        override fun create(): MyClass {
            println("MyClass Created!")
            return MyClass()
        }
    }
}
```

测试代码：

```
    MyClass.create()  // "MyClass Created!"
    MyClass.Companion.create() // "MyClass Created!"
```

另外，如果想使用Java中的静态成员和静态方法的话，我们可以用：

@JvmField注解：生成与该属性相同的静态字段 **@JvmStatic注解**：在单例对象和伴生对象中生成对应的静态方法

7.10 sealed 密封类

7.10.1 为什么使用密封类

就像我们为什么要用enum类型一样，比如你有一个enum类型 MoneyUnit，定义了元、角、分这些单位。枚举就是为了控制住你所有要的情况是正确的，而不是用硬编码方式写成字符串“元”，“角”，“分”。

同样，sealed的目的类似，一个类之所以设计成sealed，就是为了限制类的继承结构，将一个值限制在有限集中的类型中，而不能有任何其他的类型。

在某种意义上，sealed类是枚举类的扩展：枚举类型的值集合也是受限的，但每个枚举常量只存在一个实例，而密封类的一个子类可以有可包含状态的多个实例。

7.10.1 声明密封类

要声明一个密封类，需要在类名前面添加 sealed 修饰符。密封类的所有子类都必须与密封类在同一个文件中声明（在 Kotlin 1.1 之前，该规则更加严格：子类必须嵌套在密封类声明的内部）：

```
sealed class Expression

class Unit : Expression()
data class Const(val number: Double) : Expression()
data class Sum(val e1: Expression, val e2: Expression) : Expression()
data class Multiply(val e1: Expression, val e2: Expression) : Expression()
object NaN : Expression()
```

使用密封类的主要场景是在使用 when 表达式的时候，能够验证语句覆盖了所有情况，而无需再添加一个 else 子句：

```
fun eval(expr: Expression): Double = when (expr) {
    is Unit -> 1.0
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    is Multiply -> eval(expr.e1) * eval(expr.e2)
    NaN -> Double.NaN
// 不再需要 `else` 子句，因为我们已经覆盖了所有的情况
}
```

测试代码：

```
fun main(args: Array<String>) {
    val u = eval(Unit())
    val a = eval(Const(1.1))
    val b = eval(Sum(Const(1.0), Const(9.0)))
    val c = eval(Multiply(Const(10.0), Const(10.0)))
    println(u)
    println(a)
    println(b)
    println(c)
```

```
}
```

输出：

```
1.0
1.1
10.0
100.0
```

7.11 data 数据类

7.11.1 构造函数中的 `val/var`

在开始讲数据类之前，我们先来看一下几种类声明的写法。

写法一：

```
class Aook(name: String)
```

这样写，这个name变量是无法被外部访问到的。它对应的反编译之后的Java代码如下：

```
public final class Aook {
   public Aook(@NotNull String name) {
      Intrinsics.checkParameterIsNotNull(name, "name");
      super();
   }
}
```

写法二：要想这个name变量被访问到，我们可以在类体中再声明一个变量，然后把这个构造函数中的参数赋值给它：

```
class Cook(name: String) {
    val name = name
}
```

测试代码：

```
    val cook = Cook("Cook")
    cook.name
```

对应的Java实现代码是：

```
public final class Cook {
```

```
    @NotNull
    private final String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public Cook(@NotNull String name) {
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
        this.name = name;
    }
}
```

写法三：

```
class Dook(val name: String)
class Eook(var name: String)
```

构造函数中带var、val修饰的变量，Kotlin编译器会自动为它们生成getter、setter函数。

上面的写法对应的Java代码就是：

```
public final class Dook {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public Dook(@NotNull String name) {
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
        this.name = name;
    }
}

public final class Eook {
    @NotNull
    private String name;

    @NotNull
    public final String getName() {
        return this.name;
```

```
    }

    public final void setName(@NotNull String var1) {
       Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
       this.name = var1;
    }

    public Eook(@NotNull String name) {
       Intrinsics.checkParameterIsNotNull(name, "name");
       super();
       this.name = name;
    }
 }
```

测试代码：

```
      val dook = Dook("Dook")
      dook.name
      val eook = Eook("Eook")
      eook.name
```

下面我们来学习一下Kotlin中的数据类： `data class` 。

7.11.2 领域实体类

我们写Java代码的时候，会经常创建一些只保存数据的类。比如说：

- POJO类：POJO全称是Plain Ordinary Java Object / Pure Old Java Object，中文可以翻译成：普通Java类，具有一部分getter/setter方法的那种类就可以称作POJO。
- DTO类：Data Transfer Object，数据传输对象类，泛指用于展示层与服务层之间的数据传输对象。
- VO类：VO有两种说法,一个是ViewObject,一个是ValueObject。
- PO类：Persisent Object，持久对象。它们是由一组属性和属性的get和set方法组成。PO是在持久层所使用，用来封装原始数据。
- BO类：Business Object，业务对象层，表示应用程序领域内 “事物” 的所有实体类。
- DO类：Domain Object，领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。

等等。

这些我们统称为领域模型中的实体类。最简单的实体类是POJO类，含有属性及属性对应的set和get方法，实体类常见的方法还有用于输出自身数据的toString方法。

7.11.3 数据类 `data class` 的概念

在 Kotlin 中，也有对应这样的领域实体类的概念，并在语言层面上做了支持，叫做数据类 ：

```
data class Book(val name: String)
data class Fook(var name: String)
data class User(
        val name: String,
        val gender: String,
        val age: Int
) {
    fun validate(): Boolean {
        return true
    }
}
```

这里的var/val是必须要带上的。因为编译器要把主构造函数中声明的所有属性，自动生成以下函数：

```
equals()/hashCode()
toString() : 格式是 User(name=Jacky, gender=Male, age=10)
componentN() 函数 : 按声明顺序对应于所有属性component1()、component2() ...
copy() 函数
```

如果我们自定义了这些函数，或者继承父类重写了这些函数，编译器就不会再去生成。

测试代码：

```
val book = Book("Book")
book.name
book.copy("Book2")

val jack = User("Jack", "Male", 1)
jack.name
jack.gender
jack.age
jack.toString()
jack.validate()


val olderJack = jack.copy(age = 2)
val anotherJack = jack.copy(name = "Jacky", age = 10)
```

在一些场景下，我们需要复制一个对象来改变它的部分属性，而其余部分保持不变。 copy() 函数就是为此而生成。例如上面的的 User 类的copy函数的使用：

```
val olderJack = jack.copy(age = 2)
val anotherJack = jack.copy(name = "Jacky", age = 10)
```

7.11.4 数据类的限制

数据类有以下的限制要求：

1.主构造函数需要至少有一个参数。下面的写法是错误的：

```
data class Gook // error, data class must have at least one primary constructor paramet
er
```

2.主构造函数的所有参数需要标记为 val 或 var；

```
data class Hook(name: String)// error, data class must have only var/val property
```

跟普通类一样，数据类也可以有次级构造函数：

```
data class LoginUser(val name: String = "", val password: String = "") : DBase(), IBase
A, IBaseB {

    var isActive = true

    constructor(name: String, password: String, isActive: Boolean) : this(name, passwor
d) {
        this.isActive = isActive
    }
    ...
}
```

3.数据类不能是抽象、开放、密封或者内部的。也就是说，下面的写法都是错误的：

```
abstract data class Iook(val name: String) // modifier abstract is incompatible with da
ta
open data class Jook(val name: String) // modifier abstract is incompatible with data
sealed data class Kook(val name: String)// modifier sealed is incompatible with data
inner data class Look(val name: String)// modifier inner is incompatible with data
```

数据类只能是final的：

```
final data class Mook(val name: String) // modifier abstract is incompatible with data
```

4.在1.1之前数据类只能实现接口。自 1.1 起，数据类可以扩展其他类。代码示例：

```
open class DBase
interface IBaseA
```

```
interface IBaseB

data class LoginUser(val name: String, val password: String) : DBase(), IBaseA, IBaseB
{

    override fun equals(other: Any?): Boolean {
        return super.equals(other)
    }

    override fun hashCode(): Int {
        return super.hashCode()
    }

    override fun toString(): String {
        return super.toString()
    }

    fun validate(): Boolean {
        return true
    }
}
```

测试代码：

```
val loginUser1 = LoginUser("Admin", "admin")
println(loginUser1.component1())
println(loginUser1.component2())
println(loginUser1.name)
println(loginUser1.password)
println(loginUser1.toString())
```

输出：

```
Admin
admin
Admin
admin
com.easy.kotlin.LoginUser@7440e464
```

可以看出，由于我们重写了 `override fun toString(): String`，对应的输出使我们熟悉的类的输出格式。

如果我们不重写这个toString函数，则会默认输出：

```
LoginUser(name=Admin, password=admin)
```

上面的类声明的构造函数，要求我们每次必须初始化name、password的值，如果我们想拥有一个无参的构造函数，我们只要对所有的属性指定默认值即可：

```
data class LoginUser(val name: String = "", val password: String = "") : DBase(), IBase
A, IBaseB {
...
}
```

这样我们在创建对象的时候，就可以直接使用：

```
val loginUser3 = LoginUser()
loginUser3.name
loginUser3.password
```

7.11.5 数据类的解构

解构相当于 Component 函数的逆向映射：

```
val helen = User("Helen", "Female", 15)
val (name, gender, age) = helen
println("$name, $gender, $age years of age")
```

输出：

Helen, Female, 15 years of age

7.11.6 标准数据类 `Pair` 和 `Triple`

标准库中的二元组 Pair类就是一个数据类：

```
public data class Pair<out A, out B>(
        public val first: A,
        public val second: B) : Serializable {
    public override fun toString(): String = "($first, $second)"
}
```

Kotlin标准库中，对Pair类还增加了转换成List的扩展函数：

```
public fun <T> Pair<T, T>.toList(): List<T> = listOf(first, second)
```

还有三元组Triple类：

```
public data class Triple<out A, out B, out C>(
```

```
        public val first: A,
        public val second: B,
        public val third: C) : Serializable {
    public override fun toString(): String = "($first, $second, $third)"
}
 fun <T> Triple<T, T, T>.toList(): List<T> = listOf(first, second, third)
```

7.12 嵌套类（Nested Class）

7.12.1 嵌套类：类中的类

类可以嵌套在其他类中，可以嵌套多层：

```
class NestedClassesDemo {
    class Outer {
        private val zero: Int = 0
        val one: Int = 1

        class Nested {
            fun getTwo() = 2
            class Nested1 {
                val three = 3
                fun getFour() = 4
            }
        }
    }
}
```

测试代码：

```
val one = NestedClassesDemo.Outer().one
val two = NestedClassesDemo.Outer.Nested().getTwo()
val three = NestedClassesDemo.Outer.Nested.Nested1().three
val four = NestedClassesDemo.Outer.Nested.Nested1().getFour()
println(one)
println(two)
println(three)
println(four)
```

我们可以看出，访问嵌套类的方式是直接使用 `类名.`， 有多少层嵌套，就用多少层类名来访问。

普通的嵌套类，没有持有外部类的引用，所以是无法访问外部类的变量的：

```
class NestedClassesDemo {
class Outer {
```

```
        private val zero: Int = 0
        val one: Int = 1


        class Nested {
            fun getTwo() = 2

            fun accessOuter() = {
                println(zero) // error, cannot access outer class
                println(one)  // error, cannot access outer class
            }
        }
    }
}
```

我们在Nested类中，访问不到Outer类中的变量zero，one。 如果想要访问到，我们只需要在Nested类前面加上 `inner` 关键字修饰，表明这是一个嵌套的内部类。

7.12.2 内部类（Inner Class）

类可以标记为 inner 以便能够访问外部类的成员。内部类会带有一个对外部类的对象的引用：

```
class NestedClassesDemo {
class Outer {
        private val zero: Int = 0
        val one: Int = 1

        inner class Inner {
            fun accessOuter() = {
                println(zero) // works
                println(one) // works
            }

        }
}
```

测试代码：

```
val innerClass = NestedClassesDemo.Outer().Inner().accessOuter()
```

我们可以看到，当访问 `inner class Inner` 的时候，我们使用的是 `Outer().Inner()`，这是持有了Outer的对象引用。跟普通嵌套类直接使用类名访问的方式区分。

7.12.3 匿名内部类（Annonymous Inner Class）

匿名内部类，就是没有名字的内部类。既然是内部类，那么它自然也是可以访问外部类的变量的。

我们使用对象表达式创建一个匿名内部类实例：

```
class NestedClassesDemo {
class AnonymousInnerClassDemo {
            var isRunning = false
            fun doRun() {
                Thread(object : Runnable {
                    override fun run() {
                        isRunning = true
                        println("doRun : i am running, isRunning = $isRunning")
                    }
                }).start()
            }
}
}
```

如果对象是函数式 Java 接口，即具有单个抽象方法的 Java 接口的实例，例如上面的例子中的Runnable接口：

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

我们可以使用lambda表达式创建它，下面的几种写法都是可以的：

```
fun doStop() {
    var isRunning = true
    Thread({
        isRunning = false
        println("doStop: i am not running, isRunning = $isRunning")
    }).start()
}

fun doWait() {
    var isRunning = true

    val wait = Runnable {
        isRunning = false
        println("doWait: i am waiting, isRunning = $isRunning")
    }

    Thread(wait).start()
}
```

```
fun doNotify() {
    var isRunning = true

    val wait = {
        isRunning = false
        println("doNotify: i notify, isRunning = $isRunning")
    }

    Thread(wait).start()
}
```

测试代码：

```
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doRun()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doStop()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doWait()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doNotify()
```

输出：

```
doRun : i am running, isRunning = true
doStop: i am not running, isRunning = false
doWait: i am waiting, isRunning = false
doNotify: i notify, isRunning = false
```

关于lambda表达式以及函数式编程，我们将在下一章中学习。

7.13 委托(Delegation)

7.13.1 代理模式（Proxy Pattern）

代理模式，也称委托模式。

在代理模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理。代理模式是一项基本技巧，许多其他的模式，如状态模式、策略模式、访问者模式本质上是在特殊的场合采用了代理模式。

代理模式使得我们可以用聚合来替代继承，它还使我们可以模拟mixin（混合类型）。委托模式的作用是将委托者与实际实现代码分离出来，以达成解耦的目的。

一个代理模式的Java代码示例：

```
package com.easy.kotlin;

interface JSubject {
```

```java
    public void request();
}

class JRealSubject implements JSubject {
    @Override
    public void request() {
        System.out.println("JRealSubject Requesting");
    }
}

class JProxy implements JSubject {
    private JSubject subject = null;

    //通过构造函数传递代理者
    public JProxy(JSubject sub) {
        this.subject = sub;
    }

    @Override
    public void request() { //实现接口中定义的方法
        this.before();
        this.subject.request();
        this.after();
    }

    private void before() {
        System.out.println("JProxy Before Requesting ");
    }

    private void after() {
        System.out.println("JProxy After Requesting ");
    }
}

public class DelegateDemo {
    public static void main(String[] args) {
        JRealSubject jRealSubject = new JRealSubject();
        JProxy jProxy = new JProxy(jRealSubject);
        jProxy.request();
    }
}
```

输出：

```
JProxy Before Requesting
JRealSubject Requesting
JProxy After Requesting
```

7.13.2 类的委托(Class Delegation)

就像支持单例模式的object对象一样，Kotlin 在语言层面原生支持委托模式。

代码示例：

```
package com.easy.kotlin

import java.util.*

interface Subject {
    fun hello()
}

class RealSubject(val name: String) : Subject {
    override fun hello() {
        val now = Date()
        println("Hello, REAL $name! Now is $now")
    }
}

class ProxySubject(val sb: Subject) : Subject by sb {
    override fun hello() {
        println("Before ! Now is ${Date()}")
        sb.hello()
        println("After ! Now is ${Date()}")
    }
}

fun main(args: Array<String>) {
    val subject = RealSubject("World")
    subject.hello()
    println("-------------------------")
    val proxySubject = ProxySubject(subject)
    proxySubject.hello()
}
```

在这个例子中，委托代理类 ProxySubject 继承接口 Subject，并将其所有共有的方法委托给一个指定的对象sb：

```
class ProxySubject(val sb: Subject) : Subject by sb
```

ProxySubject 的超类型Subject中的 `by sb` 表示sb 将会在 ProxySubject 中内部存储。

另外，我们在覆盖重写了函数 `override fun hello()` 。

测试代码：

```
fun main(args: Array<String>) {
    val subject = RealSubject("World")
    subject.hello()
    println("-------------------------")
    val proxySubject = ProxySubject(subject)
    proxySubject.hello()
}
```

输出：

```
Hello, REAL World! Now is Wed Jul 05 02:45:42 CST 2017
-------------------------
Before ! Now is Wed Jul 05 02:45:42 CST 2017
Hello, REAL World! Now is Wed Jul 05 02:45:42 CST 2017
After ! Now is Wed Jul 05 02:45:42 CST 2017
```

7.13.3 委托属性 (Delegated Properties)

通常对于属性类型，我们是在每次需要的时候手动声明它们：

```
class NormalPropertiesDemo {
    var content: String = "NormalProperties init content"
}
```

那么这个content属性将会很 “呆板” 。属性委托赋予了属性富有变化的活力。

例如：

- 延迟属性（lazy properties）：其值只在首次访问时计算
- 可观察属性（observable properties）：监听器会收到有关此属性变更的通知
- 把多个属性储存在一个映射（map）中，而不是每个存在单独的字段中。

委托属性

Kotlin 支持 委托属性:

```
class DelegatePropertiesDemo {
    var content: String by Content()

    override fun toString(): String {
        return "DelegatePropertiesDemo Class"
    }
}

class Content {
    operator fun getValue(delegatePropertiesDemo: DelegatePropertiesDemo, property: KPr
```

```
operty<*>): String {
        return "${delegatePropertiesDemo} property '${property.name}' = 'Balalala ... '
 "
    }

    operator fun setValue(delegatePropertiesDemo: DelegatePropertiesDemo, property: KPr
operty<*>, value: String) {
        println("${delegatePropertiesDemo} property '${property.name}' is setting value
: '$value'")
    }
}
```

在 `var content: String by Content()` 中, `by` 后面的表达式的 `Content()` 就是该属性 委托的对象。content属性对应的 `get()` （和 `set()` ）会被委托给 `Content()` 的 `operator fun getValue()` 和 `operator fun setValue()` 函数，这两个函数是必须的，而且得是操作符函数。

测试代码：

```
val n = NormalPropertiesDemo()
println(n.content)
n.content = "Lao tze"
println(n.content)

val e = DelegatePropertiesDemo()
println(e.content) // call Content.getValue
e.content = "Confucius" // call Content.setValue
println(e.content) // call Content.getValue
```

输出：

```
NormalProperties init content
Lao tze
DelegatePropertiesDemo Class property 'content' = 'Balalala ... '
DelegatePropertiesDemo Class property 'content' is setting value: 'Confucius'
DelegatePropertiesDemo Class property 'content' = 'Balalala ...
```

懒加载属性委托 lazy

lazy() 函数定义如下：

```
@kotlin.jvm.JvmVersion
public fun <T> lazy(initializer: () -> T): Lazy<T> = SynchronizedLazyImpl(initializer)
```

它接受一个 lambda 并返回一个 Lazy 实例的函数，返回的实例可以作为实现懒加载属性的委托：

第一次调用 get() 会执行已传递给 lazy() 的 lamda 表达式并记录下结果， 后续调用 get() 只是返回之前记录的结果。

代码示例：

```
val synchronizedLazyImpl = lazy({
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
    "Hello, lazyValueSynchronized1 ! "
})

val lazyValueSynchronized1: String by synchronizedLazyImpl
println(lazyValueSynchronized1)
println(lazyValueSynchronized1)

val lazyValueSynchronized2: String by lazy {
    println("lazyValueSynchronized2  3!")
    println("lazyValueSynchronized2  2!")
    println("lazyValueSynchronized2  1!")
    "Hello, lazyValueSynchronized2 ! "
}

println(lazyValueSynchronized2)
println(lazyValueSynchronized2)
```

输出：

```
lazyValueSynchronized1  3!
lazyValueSynchronized1  2!
lazyValueSynchronized1  1!
Hello, lazyValueSynchronized1 !
Hello, lazyValueSynchronized1 !


lazyValueSynchronized2  3!
lazyValueSynchronized2  2!
lazyValueSynchronized2  1!
Hello, lazyValueSynchronized2 !
Hello, lazyValueSynchronized2 !
```

默认情况下，对于 lazy 属性的求值是同步的（synchronized），下面两种写法是等价的：

```
val synchronizedLazyImpl = lazy({
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
```

```
    "Hello, lazyValueSynchronized1 ! "
})

val synchronizedLazyImpl2 = lazy(LazyThreadSafetyMode.SYNCHRONIZED, {
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
    "Hello, lazyValueSynchronized1 ! "
})
```

该值是线程安全的。所有线程会看到相同的值。

如果初始化委托多个线程可以同时执行，不需要同步锁，使用 `LazyThreadSafetyMode.PUBLICATION` ：

```
val lazyValuePublication: String by lazy(LazyThreadSafetyMode.PUBLICATION, {
    println("lazyValuePublication 3!")
    println("lazyValuePublication 2!")
    println("lazyValuePublication 1!")
    "Hello, lazyValuePublication ! "
})
```

而如果属性的初始化是单线程的，那么我们使用 LazyThreadSafetyMode.NONE 模式(性能最高)：

```
val lazyValueNone: String by lazy(LazyThreadSafetyMode.NONE, {
    println("lazyValueNone 3!")
    println("lazyValueNone 2!")
    println("lazyValueNone 1!")
    "Hello, lazyValueNone ! "
})
```

Delegates.observable 可观察属性委托

我们把属性委托给 `Delegates.observable` 函数，当属性值被重新赋值的时候， 触发其中的回调函数 onChange。

该函数定义如下：

```
public inline fun <T> observable(initialValue: T, crossinline onChange: (property: KPro
perty<*>, oldValue: T, newValue: T) -> Unit):
        ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue) {
            override fun afterChange(property: KProperty<*>, oldValue: T, newValue: T)
= onChange(property, oldValue, newValue)
        }
```

代码示例：

```
class PostHierarchy {
    var level: String by Delegates.observable("P0",
            { property: KProperty<*>,
              oldValue: String,
              newValue: String ->
                println("$oldValue -> $newValue")
            })
}
```

测试代码：

```
val ph = PostHierarchy()
ph.level = "P1"
ph.level = "P2"
ph.level = "P3"
println(ph.level) // P3
```

输出：

```
P0 -> P1
P1 -> P2
P2 -> P3
P3
```

我们可以看出，属性 `level` 每次赋值，都回调了 `Delegates.observable` 中的lambda表达式所写的 `onChange` 函数。

Delegates.vetoable 可否决属性委托

这个函数定义如下：

```
public inline fun <T> vetoable(initialValue: T, crossinline onChange: (property: KPrope
rty<*>, oldValue: T, newValue: T) -> Boolean):
        ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue) {
            override fun beforeChange(property: KProperty<*>, oldValue: T, newValue: T)
: Boolean = onChange(property, oldValue, newValue)
        }
```

当我们把属性委托给这个函数时，我们可以通过 `onChange` 函数的返回值是否为true， 来选择属性的值是否需要改变。

代码示例:

```
class PostHierarchy {
    var grade: String by Delegates.vetoable("T0", {
```

```
        property, oldValue, newValue ->
        true
    })

    var notChangeGrade: String by Delegates.vetoable("T0", {
        property, oldValue, newValue ->
        false
    })
}
```

测试代码：

```
ph.grade = "T1"
ph.grade = "T2"
ph.grade = "T3"
println(ph.grade) // T3

ph.notChangeGrade = "T1"
ph.notChangeGrade = "T2"
ph.notChangeGrade = "T3"
println(ph.notChangeGrade) // T0
```

我们可以看出，当onChange函数返回值是false的时候，对属性notChangeGrade的赋值都没有生效，依然是原来的默认值T0 。

Delegates.notNull 非空属性委托

我们也可以使用委托来实现属性的非空限制：

```
var name: String by Delegates.notNull()
```

这样name属性就被限制为不能为null，如果被赋值null，编译器直接报错：

```
ph.name = null // error
Null can not be a value of a non-null type String
```

属性委托给Map映射

我们也可以把属性委托给Map：

```
class Account(val map: Map<String, Any?>) {
    val name: String by map
    val password: String by map
}
```

测试代码：

```
val account = Account(mapOf(
            "name" to "admin",
            "password" to "admin"
    ))

println("Account(name=${account.name}, password = ${account.password})")
```

输出：

```
Account(name=admin, password = admin)
```

如果是可变属性，这里也可以把只读的 Map 换成 MutableMap ：

```
class MutableAccount(val map: MutableMap<String, Any?>) {
    var name: String by map
    var password: String by map
}
```

测试代码：

```
val maccount = MutableAccount(mutableMapOf(
            "name" to "admin",
            "password" to "admin"
))

maccount.password = "root"
println("MutableAccount(name=${maccount.name}, password = ${maccount.password})")
```

输出：

```
MutableAccount(name=admin, password = root)
```

本章小结

本章我们介绍了Kotlin面向对象编程的特性： 类与构造函数、抽象类与接口、继承以及多重继承等基础知识，同时介绍了Kotlin中的注解类、枚举类、数据类、密封类、嵌套类、内部类、匿名内部类等特性类。最后我们学习了Kotlin中对单例模式、委托模式的语言层面上的内置支持：object对象、委托。

总的来说，Kotlin相比于Java的面向对象编程，增加不少有趣的功能与特性支持，这使得我们代码写起来更加方便快捷了。

我们知道，在Java 8 中，引进了对函数式编程的支持：Lambda表达式、Function接口、stream API等，而在Kotlin中，对函数式编程的支持更加全面丰富，代码写起来也更加简洁优雅。下一章中，我们来一起学习Kotlin的函数式编程。

本章示例代码工程：https://github.com/EasyKotlin/chatper7_oop

第8章 函数式编程（FP）

> 值就是函数，函数就是值。所有函数都消费函数，所有函数都生产函数。

"函数式编程", 又称泛函编程, 是一种"编程范式"（programming paradigm），也就是如何编写程序的方法论。它的基础是 λ 演算（lambda calculus）。λ演算可以接受函数当作输入（参数）和输出（返回值）。

和指令式编程相比，函数式编程的思维方式更加注重函数的计算。它的主要思想是把问题的解决方案写成一系列嵌套的函数调用。

就像在OOP中，一切皆是对象，编程的是由对象交合创造的世界；在FP中，一切皆是函数，编程的世界是由函数交合创造的世界。

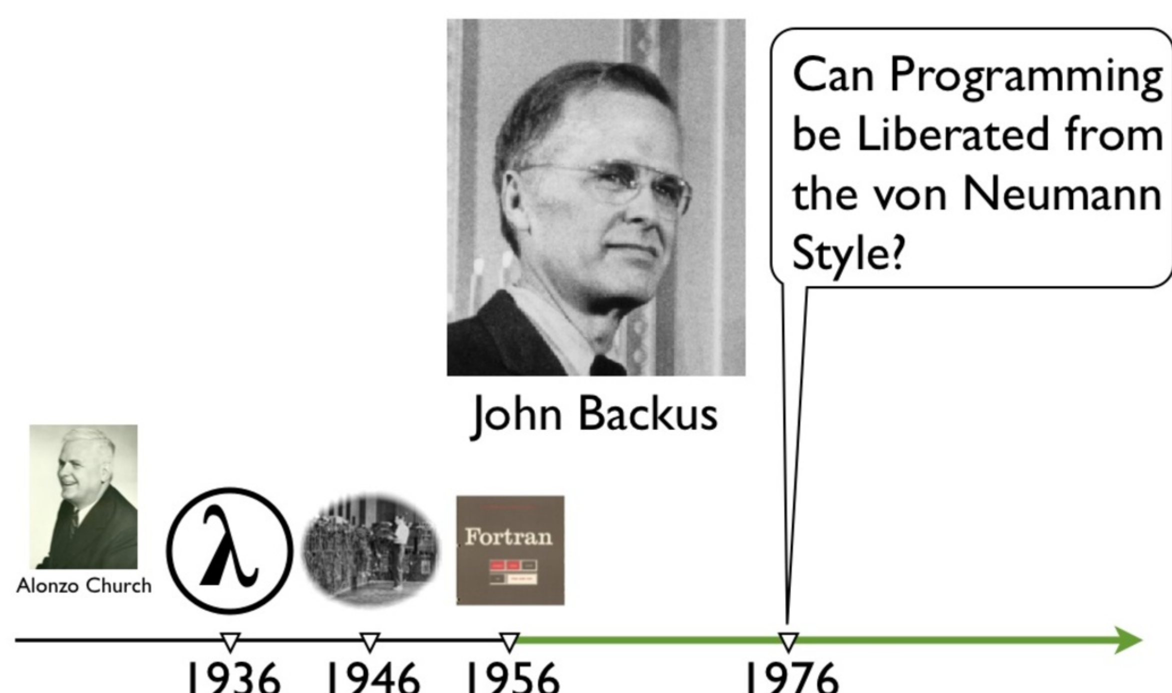
函数式编程中最古老的例子莫过于1958年被创造出来的Lisp了。Lisp由约翰·麦卡锡（John McCarthy，1927-2011）在1958年基于λ演算所创造，采用抽象数据列表与递归作符号演算来衍生人工智能。较现代的例子包括Haskell、ML、Erlang等。现代的编程语言对函数式编程都做了不同程度的支持，例如：JavaScript, Coffee Script，PHP，Perl，Python, Ruby, C# , Java 等等（这将是一个不断增长的列表）。

函数式语言在Java 虚拟机（JVM）平台上也迅速地崭露头角，例如Scala 、Clojure ； .NET 平台也不例外，例如：F# 。

函数作为Kotlin中的一等公民，可以像其他对象一样作为函数的输入与输出。关于对函数式编程的支持，相对于Scala的学院派风格，Kotlin则是纯的的工程派：实用性、简洁性上都要比Scala要好。

本章我们来一起学习函数式编程以及在Kotlin中使用函数式编程的相关内容。

8.1 函数式编程概述



函数式编程思想是一个非常古老的思想。我们简述如下：

- 我们就从1900 年 David Hilbert 的第 10 问题（能否通过有限步骤来判定不定方程是否存在有理整数解？） 开始说起吧。
- 1920，Schönfinkel，组合子逻辑(combinatory logic)。直到 Curry Haskell 1927 在普林斯顿大学当讲师时重新发现了 Moses Schönfinkel 关于组合子逻辑的成果。Moses Schönfinkel的成果预言了很多 Curry 在做的研究，于是他就跑去哥廷根大学与熟悉Moses Schönfinkel工作的Heinrich Behmann、Paul Bernays两人一起工作，并于 1930 年以一篇组合子逻辑的论文拿到了博士学位。Curry Brooks Haskell 整个职业生涯都在研究组合子，实际开创了这个研究领域，λ演算中用单参数函数来表示多个参数函数的方法被称为 Currying (柯里化)，虽然 Curry 同学多次指出这个其实是 Schönfinkel 已经搞出来的，不过其他人都是因为他用了才知道，所以这名字就这定下来了；并且有三门编程语言以他的名字命名，分别是：Curry, Brooks, Haskell。Curry 在 1928 开始开发类型系统，他搞的是基于组合子的 polymorphic，Church 则建立了基于函数的简单类型系统。
- 1929, 哥德尔(Kurt Gödel)完备性定理。Gödel 首先证明了一个形式系统中的所有公式都可以表示为自然数，并可以从一自然数反过来得出相应的公式。这对于今天的程序员都来说，数字编码、程序即数据计算机原理最核心、最基本的常识，在那个时代却脑洞大开的创见。
- 1933，λ 演算。 Church 在 1933 年搞出来一套以纯λ演算为基础的逻辑，以期对数学进行形式化描述。 λ 演算和递归函数理论就是函数式编程的基础。
- 1936，确定性问题（decision problem，德文 Entscheidungsproblem (发音 [ɛntˈʃaɪ̯dʊŋspʁoˌbleːm]） 。 Alan Turing 和 Alonzo Church，两人在同在1936年独立给出了否定答案。

- 1935-1936这个时间段上，我们有了三个有效计算模型：通用图灵机、通用递归函数、λ可定义。Rosser 1939 年正式确认这三个模型是等效的。
- 1953-1957，FORTRAN (FORmula TRANslating)，John Backus。1952 年 Halcombe Laning 提出了直接输入数学公式的设想，并制作了 GEORGE编译器演示该想法。受这个想法启发，1953 年 IBM 的 John Backus 团队给 IBM 704 主机研发数学公式翻译系统。第一个 FORTRAN (FORmula TRANslating 的缩写)编译器 1957.4 正式发行。FORTRAN 程序的代码行数比汇编少20倍。FORTRAN 的成功，让很多人认识到直接把代数公式输入进电脑是可行的，并开始渴望能用某种形式语言直接把自己的研究内容输入到电脑里进行运算。John Backus 在1970年代搞了 FP 语言，1977 年发表。虽然这门语言并不是最早的函数式编程语言，但他是 Functional Programming 这个词儿的创造者， 1977 年他的图灵奖演讲题为[“Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs”]
- 1956， LISP， John McCarthy。John McCarthy 1956年在 Dartmouth一台 IBM 704 上搞人工智能研究时，就想到要一个代数列表处理(algebraic list processing)语言。他的项目需要用某种形式语言来编写语句，以记录关于世界的信息，而他感觉列表结构这种形式挺合适，既方便编写，也方便推演。于是就创造了LISP。正因为是在 IBM 704 上开搞的，所以 LISP 的表处理函数才会有奇葩的名字： car/cdr 什么的。其实是取 IBM704 机器字的不同部分，c=content of, r=register number, a=address part, d=decrement part 。

8.1.1 面向对象编程（OOP）与面向函数编程（FOP）

面向对象编程（OOP）

在OOP中，一切皆是对象。

在面向对象的命令式（imperative）编程语言里面，构建整个世界的基础是类和类之间沟通用的消息，这些都可以用类图（class diagram）来表述。《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software，作者ErichGamma、Richard Helm、Ralph Johnson、John Vlissides）一书中，在每一个模式的说明里都附上了至少一幅类图。

OOP 的世界提倡开发者针对具体问题建立专门的数据结构，相关的专门操作行为以“方法”的形式附加在数据结构上，自顶向下地来构建其编程世界。

OOP追求的是万事万物皆对象的理念，自然地弱化了函数。例如：函数无法作为普通数据那样来传递（OOP在函数指针上的约束），所以在OOP中有各种各样的、五花八门的设计模式。

GoF所著的《设计模式-可复用面向对象软件的基础》从面向对象设计的角度出发的，通过对封装、继承、多态、组合等技术的反复使用，提炼出一些可重复使用的面向对象设计技巧。而多态在其中又是重中之重。

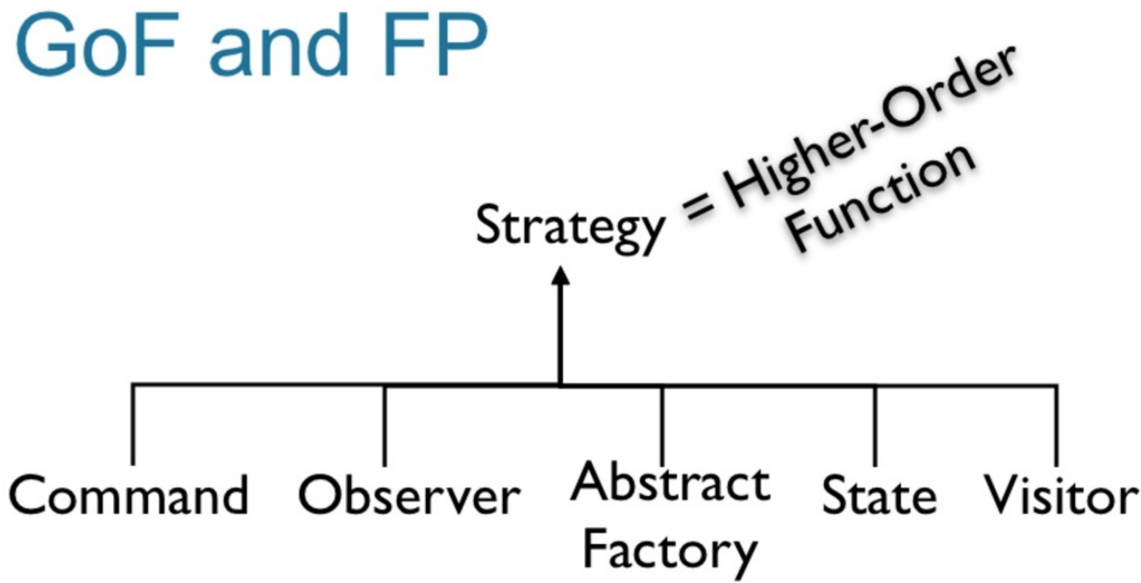
多态、面向接口编程、依赖反转等术语，描述的思想其实是相同的。这种反转模式实现了模块与模块之间的解耦。这样的架构是健壮的, 而为了实现这样的健壮系统，在系统架构中基本都需要使用多态性。

绝大部分设计模式的实现都离不开多态性的思想。换一种说法就是，这些设计模式背后的本质其实就是OOP的多态性，而OOP中的多态本质上又是受约束的函数指针。

引用Charlie Calverts对多态的描述："多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。"

简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。而我们在OOP中的那么多的设计模式，其实就是在OOP的多态性的约束规则下，对这些函数指针的调用模式的总结。

很多设计模式，在函数式编程中都可以用高阶函数来代替实现：



面向函数编程（FOP）

在FP中，一切皆是函数。

函数式编程（FP）是关于不变性和函数组合的一种编程范式。

函数式编程语言实现重用的思路很不一样。函数式语言提倡在有限的几种关键数据结构（如list、set、map）上， 运用函数的组合 (高阶函数) 操作，自底向上地来构建世界。

当然，我们在工程实践中，是不能极端地追求纯函数式的编程的。一个简单的原因就是：性能和效率。例如：对于有状态的操作，命令式操作通常会比声明式操作更有效率。纯函数式编程是解决某些问题的伟大工具，但是在另外的一些问题场景中，并不适用。因为副作用总是真实存在。

OOP喜欢自顶向下架构层层分解（解构），FP喜欢自底向上层层组合（复合）。 而实际上，编程的本质就是次化分解与复合的过程。通过这样的过程，创造一个美妙的逻辑之塔世界。

我们经常说一些代码片段是优雅的或美观的，实际上意味着它们更容易被人类有限的思维所处理。

对于程序的复合而言，好的代码是它的表面积要比体积增长的慢。 代码块的 "表面积" 是是我们复合代码块时所需要的信息（接口API协议定义）。代码块的 "体积" 就是接口内部的实现逻辑（API内部的实现代码）。

在OOP中，一个理想的对象应该是只暴露它的抽象接口（纯表面， 无体积），其方法则扮演箭头的角色。如果为了理解一个对象如何与其他对象进行复合，当你发现不得不深入挖掘对象的实现之时，此时你所用的编程范式的原本优势就荡然无存了。

FP通过函数组合来构造其逻辑系统。FP倾向于把软件分解为其需要执行的行为或操作,而且通常采用自底向上的方法。函数式编程也提供了非常强大的对事物进行抽象和组合的能力。

在FP里面，函数是 “一类公民” （first-class）。它们可以像1, 2, "hello"，true，对象…… 之类的 “值” 一样，在任意位置诞生，通过变量，参数和数据结构传递到其它地方，可以在任何位置被调用。

而在OOP中，很多所谓面向对象设计模式（design pattern）， 都是因为面向对象语言没有first-class function（对应的是多态性），所以导致了每个函数必须被包在一个对象里面（受约束的函数指针）才能传递到其它地方。

匀称的数据结构 + 匀称的算法

在面向对象式的编程中，一切皆是对象（偏重数据结构、数据抽象，轻算法）。我们把它叫做：胖数据结构-瘦算法（FDS-TA）。

在面向函数式的编程中，一切皆是函数（偏重算法，轻数据结构）。我们把它叫做：瘦数据结构-胖算法（TDS-FA）。

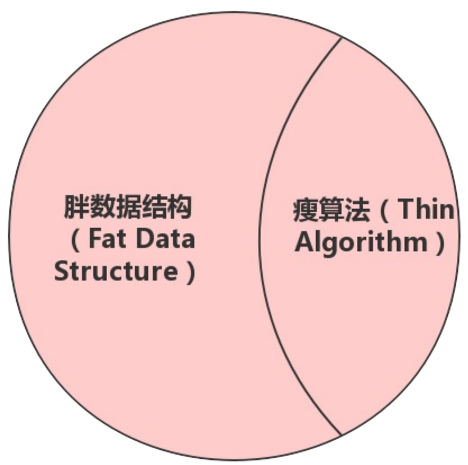
可是，这个世界很复杂，你怎么能说一切皆是啥呢？真实的编程世界，自然是匀称的数据结构结合匀称的算法（SDS-SA）来创造的。

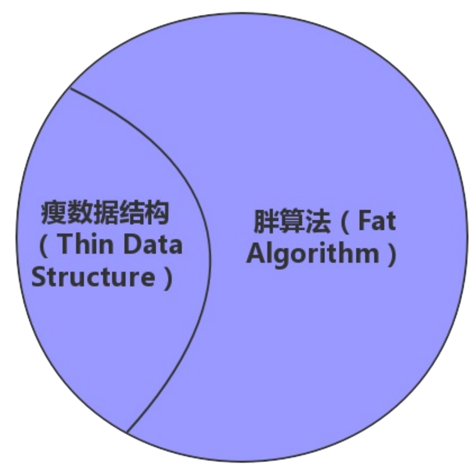
我们在编程中，不可能使用纯的对象（对象的行为方法其实就是函数），或者纯的函数（调用函数的对象、函数操作的数据其实就是数据结构）来创造一个完整的世界。如果 `数据结构` 是 `阴` ， `算法` 是 `阳` ，那么在解决实际问题中，往往是阴阳交合而成世界。还是那句经典的：

> 程序 = 匀称的数据结构 + 匀称的算法

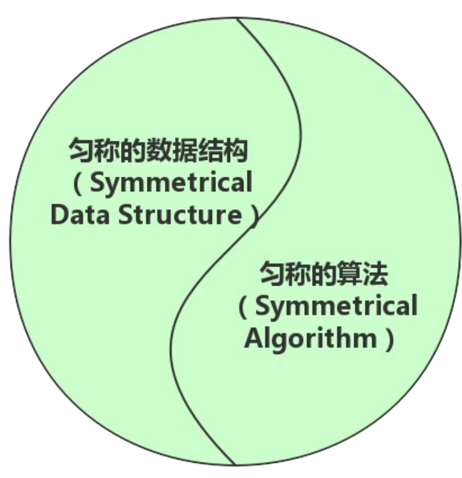
我们用一幅图来简单说明：

面向对象编程（OOP）





真实的编程: 混合式（HP）



函数与映射

一切皆是映射。函数式编程的代码主要就是“对映射的描述”。我们说组合是编程的本质，其实，组合就是建立映射关系。

一个函数无非就是从输入到输出的映射，写成数学表达式就是：

f: X -> Y p:Y -> Z p(f) : X ->Z

用编程语言表达就是：

```
fun f(x:X) : Y{}
fun p(y:Y) : Z{}
fun fp(f: (X)->Y, p: (Y)->Z) : Z {
    return {x -> p(f(x))}
}
```

8.1.2 函数式编程基本特性

在经常被引用的论文 “Why Functional Programming Matters” 中，作者 John Hughes 说明了模块化是成功编程的关键，而函数编程可以极大地改进模块化。

在函数编程中，我们有一个内置的框架来开发更小的、更简单的和更一般化的模块， 然后将它们组合在一起。

函数编程的一些基本特点包括：

- 函数是"第一等公民"。
- 闭包（Closure）和高阶函数（Higher Order Function）。
- Lambda演算与函数柯里化（Currying）。
- 懒惰计算（lazy evaluation）。
- 使用递归作为控制流程的机制。
- 引用透明性。
- 没有副作用。

8.1.3 组合与范畴

函数式编程的本质是函数的组合，组合的本质是范畴（Category）。

和搞编程的一样，数学家喜欢将问题不断加以抽象从而将本质问题抽取出来加以论证解决，范畴论就是这样一门以抽象的方法来处理数学概念的学科，主要用于研究一些数学结构之间的映射关系（函数）。

在范畴论里，一个范畴(category)由三部分组成：

- 对象(object).
- 态射(morphism).
- 组合(composition)操作符,

范畴的对象

这里的对象可以看成是一类东西，例如数学上的群，环，以及有理数，无理数等都可以归为一个对象。对应到编程语言里，可以理解为一个类型，比如说整型，布尔型等。

态射

态射指的是一种映射关系，简单理解，态射的作用就是把一个对象 A 里的值 a 映射为 另一个对象 B 里的值 b = f(a)，这就是映射的概念。

态射的存在反映了对象内部的结构，这是范畴论用来研究对象的主要手法：对象内部的结构特性是通过与别的对象的映射关系反映出来的，动静是相对的，范畴论通过研究映射关系来达到探知对象的内部结构的目的。

组合操作符

组合操作符，用点(.)表示，用于将态射进行组合。组合操作符的作用是将两个态射进行组合，例如，假设存在态射 f: A -> B, g: B -> C， 则 g.f : A -> C.

一个结构要想成为一个范畴, 除了必须包含上述三样东西，它还要满足以下三个限制:

- 结合律： f.(g.h) = (f.g).h 。
- 封闭律：如果存在态射 f, g，则必然存在 h = f.g 。
- 同一律：对结构中的每一个对象 A, 必须存在一个单位态射 Ia: A -> A， 对于单位态射，显然，对任意其它态射 f, 有 f.I = f。

在范畴论里另外研究的重点是范畴与范畴之间的关系，就正如对象与对象之间有态射一样，范畴与范畴之间也存在映射关系，从而可以将一个范畴映射为另一个范畴，这种映射在范畴论中叫作函子(functor），具体来说，对于给定的两个范畴 A 和 B, 函子的作用有两个:

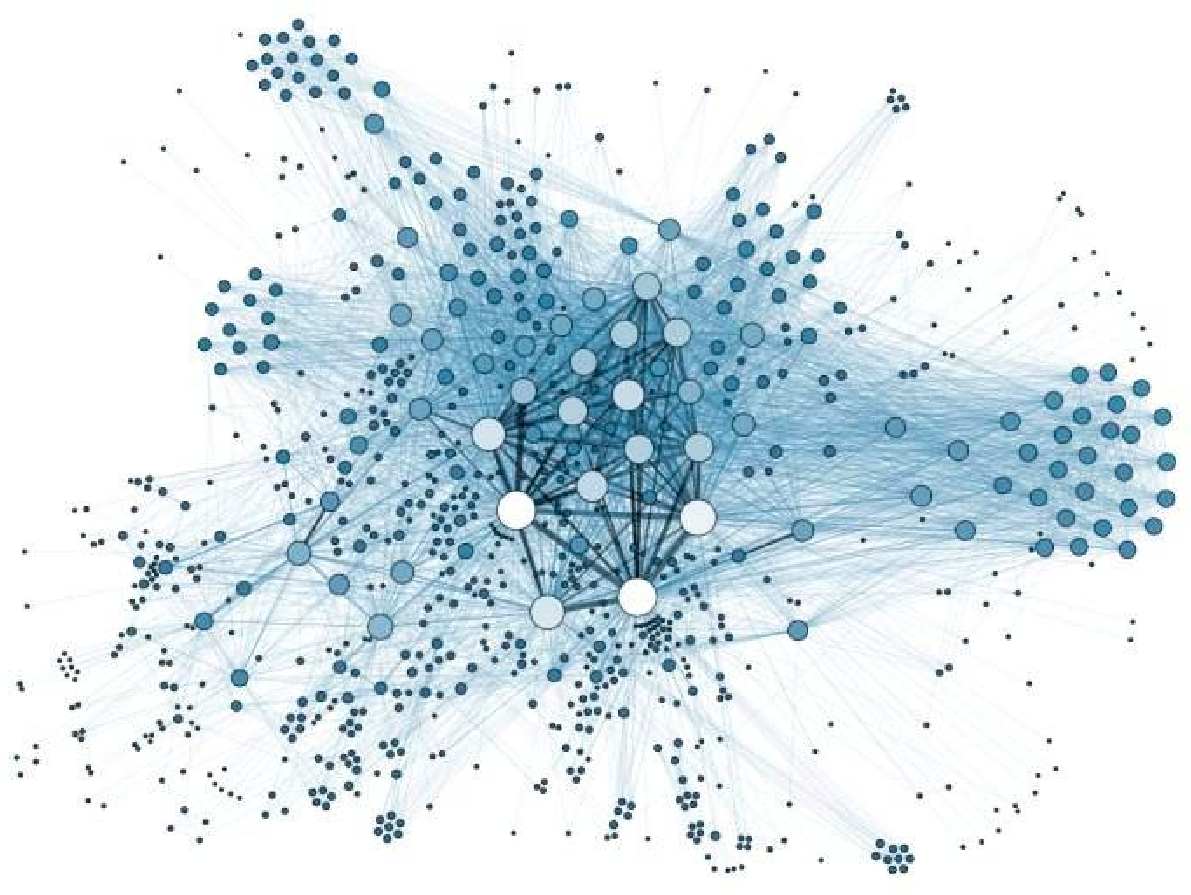
- 将范畴 A 中的对象映射到范畴 B 中的对象。
- 将范畴 A 中的态射映射到范畴 B 中的态射。

显然，函子反映了不同的范畴之间的内在联系。跟函数和泛函数的思想是相同的。

而我们的函数式编程探究的问题与思想理念可以说是跟范畴论完全吻合。如果把函数式编程的整个的世界看做一个对象，那么FP真正搞的事情就是建立通过函数之间的映射关系，来构建这样一个美丽的编程世界。

很多问题的解决（证明）其实都不涉及具体的（数据）结构，而完全可以只依赖映射之间的组合运算(composition)来搞定。这就是函数式编程的核心思想。

如果我们把 `程序` 看做图论里面的一张图G， `数据结构` 当作是图G的节点Node（数据结构，存储状态）， 而 `算法` 逻辑就是这些节点Node之间的Edge (数据映射，Mapping)， 那么这整幅图 `G(N,E)` 就是一幅美妙的抽象逻辑之塔的 `映射图` ， 也就是我们编程创造的世界：



函数是"第一等公民"

函数式编程（FP）中，函数是"第一等公民"。

所谓"第一等公民"（first class），有时称为 闭包或者 仿函数（functor）对象， 指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。这个以函数为参数的概念，跟C语言中的函数指针类似。

举例来说，下面代码中的print变量就是一个函数（没有函数名），可以作为另一个函数的参数:

```
>>> val print = fun(x:Any){println(x)}
>>> listOf(1,2,3).forEach(print)
1
2
3
```

高阶函数（Higher order Function）

FP 语言支持高阶函数，高阶函数就是多阶映射。高阶函数用另一个函数作为其输入参数，也可以返回一个函数作为输出。

代码示例：

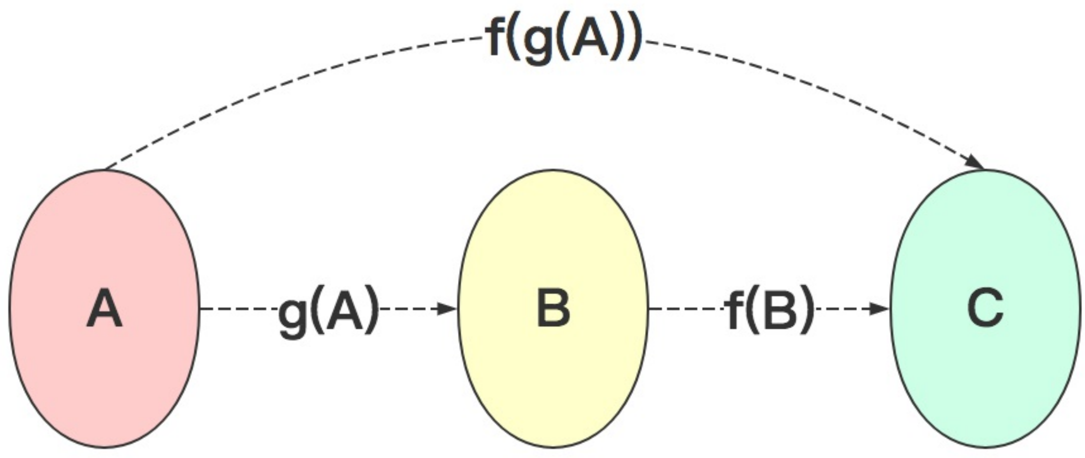
```
fun isOdd(x: Int) = x % 2 != 0
fun length(s: String) = s.length

fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

测试代码：

```
fun main(args: Array<String>) {
    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")
    println(strings.filter(oddLength)) // [a, abc]
}
```

这个compose函数，其实就是数学中的复合函数的概念，这是一个高阶函数的例子：传入的两个参数f , g都是函数，其返回值也是函数。

图示如下：



这里的

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C
```

中类型参数对应：

```
fun <String, Int, Boolean> compose(f: (Int) -> Boolean, g: (String) -> Int): (String) -
```

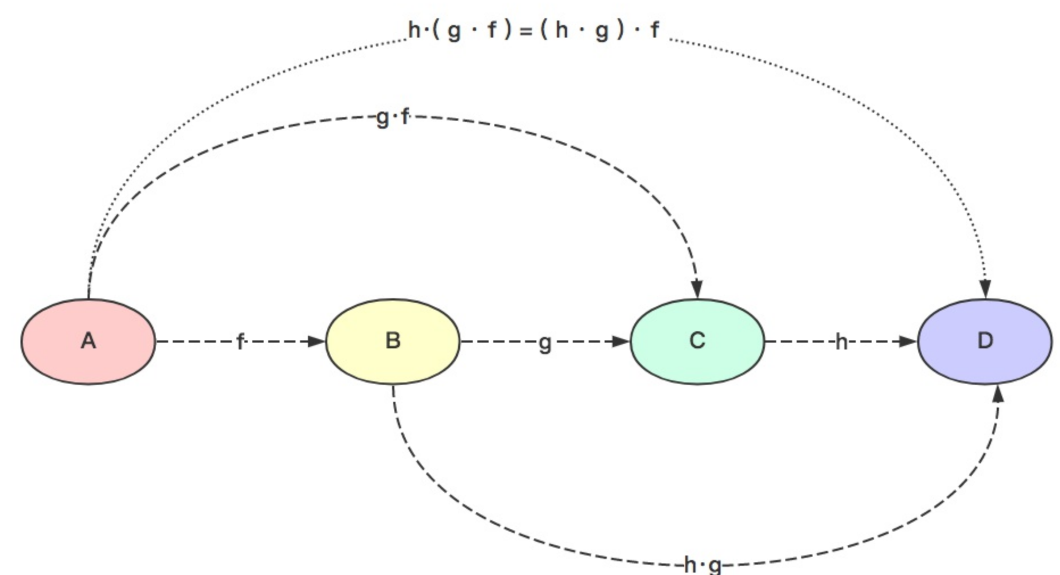
```
> Boolean
```

这里的 `(Int) -> Boolean` 、 `(String) -> Int` 、 `(String) -> Boolean` 都是函数类型。

其实，从映射的角度看，就是二阶映射。对[a, ab, abc] 中每个元素 x 先映射成长度g(x) = 1, 2, 3，再进行第二次映射：f(g(x)) %2 != 0，长度是奇数？返回值是true的被过滤出来。

有了高阶函数，我们可以用优雅的方式进行模块化编程。

另外，高阶函数满足结合律：



λ演算 （Lambda calculus 或者 λ-calculus）

λ 演算是函数式语言的基础。在λ-演算的基础上，发展起来的π-演算、χ-演算，成为近年来的并发程序的理论工具之一，许多经典的并发程序模型就是以π-演算为框架的。λ 演算神奇之处在于，通过最基本的函数抽象和函数应用法则，配套以适当的技巧，便能够构造出任意复杂的可计算函数。

λ演算是一套用于研究函数定义、函数应用和递归的形式系统。它由 阿隆佐·丘奇（Alonzo Church，1903~1995）和 Stephen Cole Kleene 在 20 世纪三十年代引入。当时的背景是解决函数可计算的本质性问题，初期λ演算成功的解决了在可计算理论中的判定性问题，后来根据Church–Turing thesis，证明了λ演算与图灵机是等价的。

λ 演算可以被称为最小的通用程序设计语言。它包括一条变换规则 (变量替换) 和一条函数定义方式，λ演算之通用在于，任何一个可计算函数都能用这种形式来表达和求值。

λ演算强调的是变换规则的运用，这里的变换规则本质上就是函数映射。 Lambda 表达式（Lambda Expression） 是 λ演算 的一部分。

λ演算中一切皆函数，全体λ表达式构成Λ空间，λ表达式为Λ空间到Λ空间的函数。

例如，在 lambda 演算中有许多方式都可以定义自然数，最常见的是Church 整数，定义如下：

```
0 = λ f. λ x. x
1 = λ f. λ x. f x
2 = λ f. λ x. f (f x)
3 = λ f. λ x. f (f (f x))
...
```

数学家们都崇尚简洁，只用一个关键字 'λ' 来表示对函数的抽象。

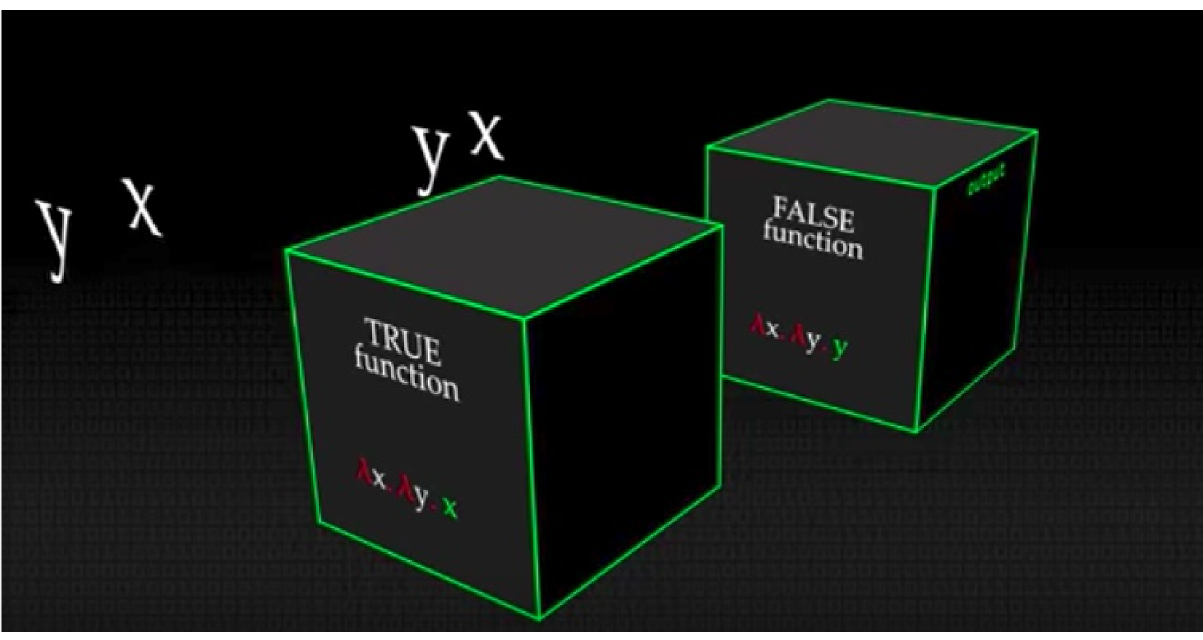
其中的 `λ f. λ x.` ，`λ f` 是抽象出来的函数，`λ x` 是输入参数，`.` 语法用来分割参数表和函数体。 为了更简洁，我们简记为F, 那么上面的Church 整数定义简写为：

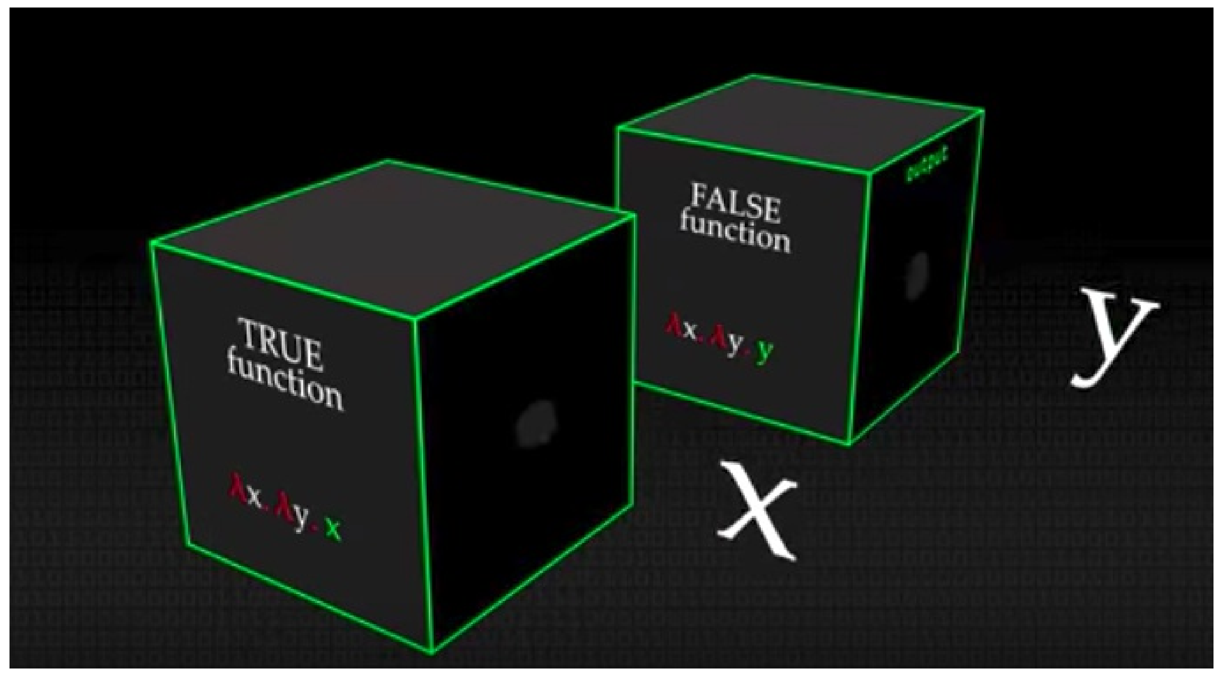
```
0 = F x
1 = F f x
2 = F f (f x)
3 = F f (f (f x))
...
```

使用λ演算定义布尔值：

```
TRUE  = λ x. λ y. x
FALSE = λ x. λ y. y
```

用图示如下：





在λ演算中只有函数，一门编程语言中的数据类型，比如boolean、number、list等，都可以使用纯λ演算来实现。我们不用去关心数据的值是什么，重点是我们能对这个值做什么操作（apply function）。

使用λ演算定义一个恒等函数I ：

```
I = λ x . x
```

使用Kotlin代码来写，如下：

```
>>> val I = {x:Int -> x}
>>> I(0)
0
>>> I(1)
1
>>> I(100)
100
```

对 I 而言任何一个 x 都是它的不动点(即对某个函数 f(x) 存在这样的一个输入 x，使得函数的输出仍旧等于输入的 x 。形式化的表示即为 f(x) = x)。

再例如，下面的 λ 表达式表示将x映射为 x+1 :

```
λ x . x + 1
```

测试代码：

```
( λ x . x + 1) 5
```

将输出6 。

这样的表达式，在Kotlin中, 如果使用Lambda表达式我们这样写：

```
>>> val addOneLambda = {
...         x: Int ->
...         x + 1
...     }
>>> addOneLambda(1)
2
```

如果使用匿名函数，这样写：

```
>>> val addOneAnonymouse = (fun(x: Int): Int {
...         return x + 1
...     })
>>> addOneAnonymouse(1)
2
```

在一些古老的编程语言中，lambda表达式还是比较接近lambda演算的表达式的。在现代程序语言中的lambda表达式，只是取名自lambda演算，已经与原始的lambda演算有很大差别了。例如：

	λf. λx. f x
Lisp	(lambda (f) (lambda (x) (f x)))
Clojure	(fn [f] (fn [x] (f x)))
Ruby	lambda { |f| lambda { |x| f[x] } }
CoffeeScript	->(f) { -> (x) { f.(x) } }
	(f) -> (x) -> f(x)
Javascript	function(f) { return function(x) { return f(x) } }

在Javascript里没有任何语法专门代表lambda, 只写成这样的嵌套函数 `function{ return function{...} }` 。

函数柯里化（Currying）

很多基于 lambda calculus 的程序语言，比如 ML 和 Haskell，都习惯用currying 的手法来表示函数。比如，如果你在 Haskell 里面这样写一个函数：

```
f x y = x + y
```

然后你就可以这样把链表里的每个元素加上 2：

```
map (f 2) [1, 2, 3]
```

它会输出 [3, 4, 5]。

Currying 用一元函数，来组合成多元函数。比如，上面的函数 f 的定义在 Scheme 里面相当于：

```
(define f (lambda (x) (lambda (y) (+ x y))))
```

它是说，函数 f，接受一个参数 x，返回另一个函数（没有名字）。这个匿名函数，如果再接受一个参数 y，就会返回 x + y。所以上面的例子里面，(f 2) 返回的是一个匿名函数，它会把 2 加到自己的参数上面返回。所以把它 map 到 [1, 2, 3]，我们就得到了 [3, 4, 5]。

我们再使用Kotlin中的函数式编程来举例说明。

首先，我们看下普通的二元函数的写法：

```
fun add(x: Int, y: Int): Int {
    return x + y
}

add(1, 2) // 输出3
```

这种写法最简单，只有一层映射。

柯里化的写法：

```
fun curryAdd(x: Int): (Int) -> Int {
    return { y -> x + y }
}

curryAdd(1)(2)// 输出3
```

我们先传入参数x = 1， 返回函数 curryAdd(1) = 1 + y； 然后传入参数 y = 2, 返回最终的值 curryAdd(1)(2) = 3。

当然，我们也有 λ 表达式的写法：

```
val lambdaCurryAdd = {
        x: Int ->
        {
            y: Int ->
            x + y
        }
    }

lambdaCurryAdd(1)(2)  // 输出 3
```

这个做法其实来源于最早的 lambda calculus 的设计。因为 lambda calculus 的函数都只有一个参数，所以为了能够表示多参数的函数， Haskell Curry （数学家和逻辑学家），发明了这个方法。

不过在编码实践中，Currying 的工程实用性、简洁性上不是那么的友好。大量使用 Currying，会导致代码可读性降低，复杂性增加，并且还可能因此引起意想不到的错误。 所以在我们的讲求工程实践性能的Kotlin语言中，

古老而美丽的理论，也许能够给我带来思想的启迪，但是在工程实践中未必那么理想。

闭包（Closure）

闭包简单讲就是一个代码块，用 `{ }` 包起来。此时，程序代码也就成了数据，可以被一个变量所引用（与C语言的函数指针比较类似）。闭包的最典型的应用是实现回调函数（callback）。

闭包包含以下两个组成部分：

- 要执行的代码块（由于自由变量被包含在代码块中，这些自由变量以及它们引用的对象没有被释放）
- 自由变量的作用域

在PHP、Scala、Scheme、Common Lisp、Smalltalk、Groovy、JavaScript、Ruby、 Python、Go、Lua、objective c、swift 以及Java（Java8及以上）等语言中都能找到对闭包不同程度的支持。

Lambda表达式可以表示闭包。

惰性计算

除了高阶函数、闭包、Lambda表达式的概念，FP 还引入了惰性计算的概念。惰性计算（尽可能延迟表达式求值）是许多函数式编程语言的特性。惰性集合在需要时提供其元素，无需预先计算它们，这带来了一些好处。首先，您可以将耗时的计算推迟到绝对需要的时候。其次，您可以创造无限个集合，只要它们继续收到请求，就会继续提供元素。第三，map 和 filter 等函数的惰性使用让您能够得到更高效的代码（请参阅 参考资料 中的链接，加入由 Brian Goetz 组织的相关讨论）。

在惰性计算中，表达式不是在绑定到变量时立即计算，而是在求值程序需要产生表达式的值时进行计算。

一个惰性计算的例子是生成无穷 Fibonacci 列表的函数，但是对 第 n 个Fibonacci 数的计算相当于只是从可能的无穷列表中提取一项。

递归函数

递归指的是一个函数在其定义中直接或间接调用自身的一种方法, 它通常把一个大型的复杂的问题转化为一个与原问题相似的规模较小的问题来解决（复用函数自身）, 这样可以极大的减少代码量。递归分为两个阶段:

1.递推:把复杂的问题的求解推到比原问题简单一些的问题的求解; 2.回归:当获得最简单的情况后,逐步返回,依次得到复杂的解。

递归的能力在于用有限的语句来定义对象的无限集合。

使用递归要注意的有两点:

（1）递归就是在过程或函数里面调用自身;

（2）在使用递归时,必须有一个明确的递归结束条件,称为递归出口。

下面我们举例说明。

阶乘函数 fact(n) 一般这样递归地定义：

fact(n) = if n=0 then 1 else n * fact(n-1)

我们使用Kotlin代码实现这个函数如下：

```
fun factorial(n: Int): Int {
    println("factorial() called!  n=$n")
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

测试代码：

```
@Test
fun testFactorial() {
    Assert.assertTrue(factorial(0) == 1)
    Assert.assertTrue(factorial(1) == 1)
    Assert.assertTrue(factorial(3) == 6)
    Assert.assertTrue(factorial(10) == 3628800)
}
```

输出：

```
factorial() called!  n=0
factorial() called!  n=1
factorial() called!  n=0
factorial() called!  n=3
factorial() called!  n=2
factorial() called!  n=1
factorial() called!  n=0
factorial() called!  n=10
factorial() called!  n=9
factorial() called!  n=8
factorial() called!  n=7
factorial() called!  n=6
factorial() called!  n=5
factorial() called!  n=4
factorial() called!  n=3
```

```
factorial() called!  n=2
factorial() called!  n=1
factorial() called!  n=0
BUILD SUCCESSFUL in 24s
6 actionable tasks: 5 executed, 1 up-to-date
```

我们可以看到在factorial计算的过程中，函数不断的调用自身，然后不断的展开，直到最后到达了终止的n==0，这是递归的原则之一，就是在递归的过程中，传递的参数一定要不断的接近终止条件，在上面的例子中就是n的值不断减少，直至最后为0。

再举个Fibonacci数列的例子。

Fibonacci数列用数学中的数列的递归表达式定义如下：

fibonacci (0) = 0 fibonacci (1) = 1 fibonacci (n) = fibonacci (n - 1) + fibonacci (n - 2)

我们使用Kotlin代码实现它：

```
fun fibonacci(n: Int): Int {
    if (n == 1 || n == 2) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

测试代码：

```
@Test
fun testFibonacci() {
    Assert.assertTrue(fibonacci(1) == 1)
    Assert.assertTrue(fibonacci(2) == 1)
    Assert.assertTrue(fibonacci(3) == 2)
    Assert.assertTrue(fibonacci(4) == 3)
    Assert.assertTrue(fibonacci(5) == 5)
    Assert.assertTrue(fibonacci(6) == 8)
}
```

外篇： Scheme中的递归写法

因为Scheme 程序中充满了一对对嵌套的小括号，这些嵌套的符号体现了最基本的数学思想——递归。所以，为了多维度的来理解递归，我们给出Scheme中的递归写法：

```
  (define factorial
    (lambda (n)
      (if (= n 0)
          1
          (* n (factorial (- n 1))))))
```

```
  (define fibonacci
    (lambda (n)
      (cond ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))
```

其中关键字lambda, 表明我们定义的(即任何封闭的开括号立即离开λ及其相应的关闭括号)是一个函数。

Lambda演算和函数式语言的计算模型天生较为接近，Lambda表达式一般是这些语言必备的基本特性。

Scheme是Lisp方言，遵循极简主义哲学，有着独特的魅力。Scheme的一个主要特性是可以像操作数据一样操作函数调用。

Y组合子(Y - Combinator)

在现代编程语言中，函数都是具名的，而在传统的Lambda Calculus中，函数都是没有名字的。这样就出现了一个问题 —— 如何在Lambda Calculus中实现递归函数，即匿名递归函数。Haskell B. Curry （编程语言 Haskell 就是以此人命名的）发现了一种不动点组合子 —— Y Combinator，用于解决匿名递归函数实现的问题。Y 组合子(Y Combinator)，其定义是：

```
Y = λf.(λx.f (x x)) (λx.f (x x))
```

对于任意函数 g，可以通过推导得到 `Y g = g (Y g)` （（高阶）函数的不动点 ），从而证明 λ演算 是 图灵完备 的。 Y 组合子 的重要性由此可见一斑。

她让人绞尽脑汁，也琢磨不定！她让人心力憔悴，又百般回味！ 她，看似平淡，却深藏玄机！她，貌不惊人，却天下无敌！ 她是谁？她就是 Y 组合子：Y = λf.(λx.f (x x)) (λx.f (x x))，不动点组合子中最著名的一个。

Y 组合子让我们可以定义匿名的递归函数。Y组合子是Lambda演算的一部分，也是函数式编程的理论基础。仅仅通过Lambda表达式这个最基本的 原子 实现循环迭代。Y 组合子本身是函数，其输入也是函数（在 Lisp 中连程序都是函数）。

颇有道生一、一生二、二生三、三生万物的韵味。

举个例子说明： 我们先使用类C语言中较为熟悉的JavaScript来实现一个Y组合子函数， 因为JavaScript语言的动态特性，使得该实现相比许多需要声明各种类型的语言要简洁许多：

```
function Y(f) {
    return (function (g) {
        return g(g);
    })(function (g) {
        return f(function (x) {
            return g(g)(x);
        });
```

```
    });
}

var fact = Y(function (rec) {
    return function (n) {
        return n == 0 ? 1 : n * rec(n - 1);
    };
});
```

我们使用了Y函数组合一段匿名函数代码，实现了一个匿名的递归阶乘函数。

直接将这两个函数放到浏览器的Console中去执行，我们将看到如下输出：

```
fact(10)
3628800
```

```
> function Y(f) {
      return (function (g) {
          return g(g);
      })(function (g) {
          return f(function (x) {
              return g(g)(x);
          });
      });
  }
< undefined
> var fact = Y(function (rec) {
      return function (n) {
          return n == 0 ? 1 : n * rec(n - 1);
      };
  });
< undefined
> fact(10)
< 3628800
```

这个Y函数相当绕脑。要是在Clojure（JVM上的Lisp方言）中，这个Y函数实现如下：

```
(defn Y [r]
 ((fn [f] (f f))
 (fn [f]
 (r (fn [x] ((f f) x))))))
```

使用Scheme语言来表达：

```
  (define Y
    (lambda (f)
      ((lambda (x) (f (lambda (y) ((x x) y))))
       (lambda (x) (f (lambda (y) ((x x) y)))))))
```

我们可以看出，使用Scheme语言表达的Y组合子跟 原生的 λ演算 表达式基本一样。

用CoffeeScript实现一个 Y combinator就长这样：

```
coffee> Y = (f) -> ((x) -> (x x)) ((x) -> (f ((y) -> ((x x) y))))
[Function]
```

这个看起就相当简洁优雅了。我们使用这个 Y combinator 实现一个匿名递归的Fibonacci函数：

```
coffee> fib = Y (f) -> (n) ->  if n < 2 then n else f(n-1) + f(n-2)
[Function]
coffee> index = [0..10]
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
coffee> index.map(fib)
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

实现一个匿名递归阶乘函数：

```
coffee> fact = Y (f)  ->(n) -> if n==0 then 1 else n*f(n-1)
[Function]
coffee> fact(10)
3628800
```

上面的Coffee代码的命令行REPL运行环境搭建非常简单：

```
$ npm install -g coffee-script
$ coffee
coffee>
```

对CoffeeScript感兴趣的读者，可以参考：http://coffee-script.org/。

但是，这个Y组合子 要是 使用 OOP 语言编程范式， 就要显得复杂许多。为了更加深刻地认识OOP 与 FP编程范式，我们使用Java 8 以及 Kotlin 的实例来说明。这里使用Java给出示例的原因，是为了给出Kotlin与Java语言上的对比，在下一章节中，我们将要学习Kotlin与Java的互操作。

首先我们使用Java的匿名内部类实现Y组合子 ：

```
package com.easy.kotlin;

public class YCombinator {
    public static Lambda<Lambda> yCombinator(final Lambda<Lambda> f) {
        return new Lambda<Lambda>() {
            @Override
            public Lambda call(Object input) {
                final Lambda<Lambda> u = (Lambda<Lambda>)input;
                return u.call(u);
```

```
            }
        }.call(new Lambda<Lambda>() {
            @Override
            public Lambda call(Object input) {
                final Lambda<Lambda> x = (Lambda<Lambda>)input;
                return f.call(new Lambda<Object>() {
                    @Override
                    public Object call(Object input) {
                        return x.call(x).call(input);
                    }
                });
            }
        });
    }

    public static void main(String[] args) {
        Lambda<Lambda> y = yCombinator(new Lambda<Lambda>() {
            @Override
            public Lambda call(Object input) {
                final Lambda<Integer> fab = (Lambda<Integer>)input;
                return new Lambda<Integer>() {
                    @Override
                    public Integer call(Object input) {
                        Integer n = Integer.parseInt(input.toString());
                        if (n < 2) {
                            return Integer.valueOf(1);
                        } else {
                            return n * fab.call(n - 1);
                        }
                    }
                };
            }
        });
        System.out.println(y.call(10));//输出: 3628800
    }

    interface Lambda<E> {
        E call(Object input);
    }
}
```

这里定义了一个 `Lambda<E>` 类型， 然后通过 `E call(Object input)` 方法实现自调用，方法实现里有多处转型以及嵌套调用。逻辑比较绕，代码可读性也比较差。当然，这个问题本身也比较复杂。

我们使用Java 8的Lambda表达式来改写下匿名内部类：

```
package com.easy.kotlin;
```

```
public class YCombinator2 {

    public static Lambda<Lambda> yCombinator2(final Lambda<Lambda> f) {
        return ((Lambda<Lambda>)(Object input) -> {
            final Lambda<Lambda> u = (Lambda<Lambda>)input;
            return u.call(u);
        }).call(
            ((Lambda<Lambda>)(Object input) -> {
                final Lambda<Lambda> v = (Lambda<Lambda>)input;
                return f.call((Lambda<Object>)(Object p) -> {
                    return v.call(v).call(p);
                });
            })
        );

    }

    public static void main(String[] args) {
        Lambda<Lambda> y2 = yCombinator2(
            (Lambda<Lambda>)(Object input) -> {
                Lambda<Integer> fab = (Lambda<Integer>)input;
                return (Lambda<Integer>)(Object p) -> {
                    Integer n = Integer.parseInt(p.toString());
                    if (n < 2) {
                        return Integer.valueOf(1);
                    } else {
                        return n * fab.call(n - 1);
                    }
                };
            });

        System.out.println(y2.call(10));//输出： 3628800
    }

    interface Lambda<E> {
        E call(Object input);
    }

}
```

最后，我们使用Kotlin的对象表达式（顺便复习回顾一下上一章节的相关内容）实现Y组合子：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/7/9.
 *
 * lambda f. (lambda x. (f(x x)) lambda x. (f(x x)))
```

```
 *
 * OOP YCombinator
 *
 */

object YCombinatorKt {

    fun yCombinator(f: Lambda<Lambda<*>>): Lambda<Lambda<*>> {

        return object : Lambda<Lambda<*>> {

            override fun call(n: Any): Lambda<*> {
                val u = n as Lambda<Lambda<*>>
                return u.call(u)
            }
        }.call(object : Lambda<Lambda<*>> {

            override fun call(n: Any): Lambda<*> {
                val x = n as Lambda<Lambda<*>>

                return f.call(object : Lambda<Any> {
                    override fun call(n: Any): Any {
                        return x.call(x).call(n)!!
                    }
                })
            }

        }) as Lambda<Lambda<*>>
    }

    @JvmStatic fun main(args: Array<String>) {

        val y = yCombinator(object : Lambda<Lambda<*>> {

            override fun call(n: Any): Lambda<*> {
                val fab = n as Lambda<Int>

                return object : Lambda<Int> {

                    override fun call(n: Any): Int {
                        val n = Integer.parseInt(n.toString())
                        if (n < 2) {
                            return Integer.valueOf(1)
                        } else {
                            return n * fab.call(n - 1)
                        }
                    }
                }
            }
```

```
        })

        println(y.call(10)) //输出： 3628800
    }

    interface Lambda<E> {
        fun call(n: Any): E
    }
}
```

关于Y combinator的更多实现，可以参考：https://gist.github.com/Jason-Chen-2017/88e13b63fa5b7c612fddf999739964b0 ；另外，关于Y combinator的原理介绍，推荐看《The Little Schemer》这本书。

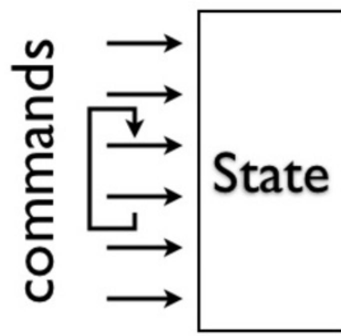
从上面的例子，我们可以看出OOP中的对接口以及多态类型，跟FP中的函数的思想表达的，本质上是一个东西，这个东西到底是什么呢？我们姑且称之为"编程之道"罢！

Y combinator 给我们提供了一种方法，让我们在一个只支持first-class函数，但是没有内建递归的编程语言里完成递归。所以Y combinator给我们展示了一个语言完全可以定义递归函数，即使这个语言的定义一点也没提到递归。它给我们展示了一件美妙的事：仅仅函数式编程自己，就可以让我们做到我们从来不认为可以做到的事（而且还不止这一个例子）。

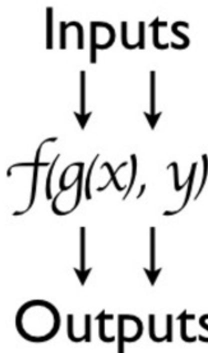
严谨而精巧的lambda演算体系，从最基本的概念"函数"入手，创造出一个绚烂而宏伟的世界，这不能不说是人类思维的骄傲。

没有"副作用"

Effects (Mutability)



VS



所谓"副作用"（side effect），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

函数式编程的动机，一开始就是为了处理运算（computation），不考虑系统的读写（I/O）。"语句"属于对系统的读写操作，所以就被排斥在外。

当然，实际应用中，不做I/O是不可能的。因此，编程过程中，函数式编程只要求把I/O限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。

在其他类型的语言中，变量往往用来保存"状态"（state）。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归。

引用透明性

函数程序通常还加强引用透明性，即如果提供同样的输入，那么函数总是返回同样的结果。就是说，表达式的值不依赖于可以改变值的全局状态。这样我们就可以从形式上逻辑推断程序行为。因为表达式的意义只取决于其子表达式而不是计算顺序或者其他表达式的副作用。这有助于我们来验证代码正确性、简化算法，有助于找出优化它的方法。

8.2 在Kotlin中使用函数式编程

好了亲，前文中我们在函数式编程的世界里遨游了一番，现在我们把思绪收回来，放到在Kotlin中的函数式编程中来。

严格的面向对象的观点，使得很多问题的解决方案变得较为笨拙。为了将一行有用的代码包装到Runnable或者Callable 这两个Java中最流行的函数式示例中，我们不得不去写五六行模板范例代码。为了让事情简单化（在Java 8中，增加Lambda表达式的支持），我们在Kotlin中使用普通的函数来替代函数式接口。事实上，函数式编程中的函数，比C语言中的函数或者Java中的方法都要强大的多。

在Kotlin中，支持函数作为一等公民。它支持高阶函数、Lambda表达式等。我们不仅可以把函数当做普通变量一样传递、返回，还可以把它分配给变量、放进数据结构或者进行一般性的操作。它们可以是未经命名的，也就是匿名函数。我们也可以直接把一段代码丢到 `{}` 中，这就是闭包。

在前面的章节中，其实我们已经涉及到一些关于函数的地方，我们将在这里系统地学习一下Kotlin的函数式编程。

8.2.1 Kotlin中的函数

首先，我们来看下Kotlin中函数的概念。

函数声明

Kotlin 中的函数使用 fun 关键字声明

```
fun double(x: Int): Int {
    return 2*x
}
```

函数用法

调用函数使用传统的方法

```
fun test() {
        val doubleTwo = double(2)
        println("double(2) = $doubleTwo")
    }
```

输出：double(2) = 4

调用成员函数使用点表示法

```
object FPBasics {

    fun double(x: Int): Int {
```

```
        return 2 * x
    }

    fun test() {
        val doubleTwo = double(2)
        println("double(2) = $doubleTwo")
    }
}

fun main(args: Array<String>) {
    FPBasics.test()
}
```

我们这里直接用object对象FPBasics来演示。

8.2.2 扩展函数

通过 扩展 声明完成一个类的新功能 扩展， 而无需继承该类或使用设计模式(例如，装饰者模式)。

一个扩展String类的swap函数的例子：

```
fun String.swap(index1: Int, index2: Int): String {
    val charArray = this.toCharArray()
    val tmp = charArray[index1]
    charArray[index1] = charArray[index2]
    charArray[index2] = tmp

    return charArrayToString(charArray)
}

fun charArrayToString(charArray: CharArray): String {
    var result = ""
    charArray.forEach { it -> result = result + it }
    return result
}
```

这个 this 关键字在扩展函数内部对应到接收者对象（传过来的在点符号前的对象）。 现在，我们对任意 String 调用该函数了：

```
val str = "abcd"
val swapStr = str.swap(0, str.lastIndex)
println("str.swap(0, str.lastIndex) = $swapStr")
```

输出： str.swap(0, str.lastIndex) = dbca

8.2.3 中缀函数

在以下场景中，函数还可以用中缀表示法调用：

- 成员函数或扩展函数
- 只有一个参数
- 用 `infix` 关键字标注

例如，给 Int 定义扩展

```
infix fun Int.shl(x: Int): Int {
 ...
}
```

用中缀表示法调用扩展函数：

```
1 shl 2
```

等同于这样

```
1.shl(2)
```

8.2.4 函数参数

函数参数使用 Pascal 表示法定义，即 name: type。参数用逗号隔开。每个参数必须显式指定其类型。

```
fun powerOf(number: Int, exponent: Int): Int {
    return Math.pow(number.toDouble(), exponent.toDouble()).toInt()
}
```

测试代码：

```
val eight = powerOf(2, 3)
println("powerOf(2,3) = $eight")
```

输出：powerOf(2,3) = 8

默认参数

函数参数可以有默认值，当省略相应的参数时使用默认值。这可以减少重载数量。

```
fun add(x: Int = 0, y: Int = 0): Int {
    return x + y
```

```
}
```

默认值通过类型后面的 = 及给出的值来定义。

测试代码：

```
val zero = add()
val one = add(1)
val two = add(1, 1)
println("add() = $zero")
println("add(1) = $one")
println("add(1, 1) = $two")
```

输出：

```
add() = 0
add(1) = 1
add(1, 1) = 2
```

另外，覆盖带默认参数的函数时，总是使用与基类型方法相同的默认参数值。 当覆盖一个带有默认参数值的方法时，签名中不带默认参数值：

```
open class DefaultParamBase {
    open fun add(x: Int = 0, y: Int = 0): Int {
        return x + y
    }
}

class DefaultParam : DefaultParamBase() {
    override fun add(x: Int, y: Int): Int { // 不能有默认值
        return super.add(x, y)
    }
}
```

命名参数

可以在调用函数时使用命名的函数参数。当一个函数有大量的参数或默认参数时这会非常方便。

给定以下函数

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
```

```
}
```

我们可以使用默认参数来调用它

```
reformat(str)
```

然而，当使用非默认参数调用它时，该调用看起来就像

```
reformat(str, true, true, false, '_')
```

使用命名参数我们可以使代码更具有可读性

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

并且如果我们不需要所有的参数

```
reformat(str, wordSeparator = '_')
```

可变数量的参数（Varargs）

函数的参数（通常是最后一个）可以用 `vararg` 修饰符标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

允许将可变数量的参数传递给函数：

```
val list = asList(1, 2, 3)
```

8.2.5 函数返回类型

函数返回类型需要显式声明

具有块代码体的函数必须始终显式指定返回类型，除非他们旨在返回 `Unit` 。

Kotlin 不推断具有块代码体的函数的返回类型，因为这样的函数在代码体中可能有复杂的控制流，并且返回类型对于读者（有时对于编译器）也是不明显的。

返回 Unit 的函数

如果一个函数不返回任何有用的值，它的返回类型是 `Unit` 。 `Unit` 是一种只有一个 `Unit` 值的类型。这个值不需要显式返回:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` 或者 `return` 是可选的
}
```

`Unit` 返回类型声明也是可选的。上面的代码等同于

```
fun printHello(name: String?) {
    .....
}
```

8.2.6 单表达式函数

当函数返回单个表达式时，可以省略花括号并且在 = 符号之后指定代码体即可

```
fun double(x: Int): Int = x * 2
```

当返回值类型可由编译器推断时，显式声明返回类型是可选的:

```
fun double(x: Int) = x * 2
```

8.2.7 函数作用域

在 Kotlin 中函数可以在文件顶层声明，这意味着你不需要像一些语言如 Java、C# 或 Scala 那样创建一个类来保存一个函数。此外除了顶层函数，Kotlin 中函数也可以声明在局部作用域、作为成员函数以及扩展函数。

局部函数（嵌套函数）

Kotlin 支持局部函数，即一个函数在另一个函数内部

```
fun sum(x: Int, y: Int, z: Int): Int {
```

```
    val delta = 0;
    fun add(a: Int, b: Int): Int {
        return a + b + delta
    }
    return add(x + add(y, z))
}
```

局部函数可以访问外部函数（即闭包）中的局部变量delta。

```
println("sum(1,2,3) = ${sum(0, 1, 2, 3)}")
```

输出：

sum(1,2,3) = 6

成员函数

成员函数是在类或对象内部定义的函数

```
class Sample() {
    fun foo() { print("Foo") }
}
```

成员函数以点表示法调用

```
Sample().foo() // 创建类 Sample 实例并调用 foo
```

8.2.8 泛型函数

函数可以有泛型参数，通过在函数名前使用尖括号指定。

例如Iterable的map函数：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

8.2.9 高阶函数

高阶函数是将函数用作参数或返回值的函数。例如，Iterable的filter函数：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

它的输入参数 `predicate: (T) -> Boolean` 就是一个函数。其中，函数类型声明的语法是：

```
(X)->Y
```

表示这个函数是从类型X到类型Y的映射。即这个函数输入X类型，输出Y类型。

这个函数我们这样调用：

```
fun isOdd(x: Int): Boolean {
    return x % 2 == 1
}

val list = listOf(1, 2, 3, 4, 5)
list.filter(::isOdd)
```

其中，`::` 用来引用一个函数。

8.2.10 匿名函数

我们也可以使用匿名函数来实现这个predicate函数：

```
list.filter((fun(x: Int): Boolean {
    return x % 2 == 1
}))
```

8.2.11 Lambda 表达式

我们也可以直接使用更简单的Lambda表达式来实现一个predicate函数：

```
list.filter {
    it % 2 == 1
}
```

- lambda 表达式总是被大括号 `{}` 括着
- 其参数（如果有的话）在 `->` 之前声明（参数类型可以省略）
- 函数体（如果存在的话）在 `->` 后面

上面的写法跟：

```
list.filter({
    it % 2 == 1
})
```

等价，如果 lambda 是该调用的唯一参数，则调用中的圆括号可以省略。

使用Lambda表达式定义一个函数字面值：

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> sum(1,1)
2
```

我们在使用嵌套的Lambda表达式来定义一个柯里化的sum函数：

```
>>> val sum = {x:Int ->  {y:Int -> x+y }}
>>> sum
(kotlin.Int) -> (kotlin.Int) -> kotlin.Int
>>> sum(1)(1)
2
```

8.2.11 `it` ：单个参数的隐式名称

Kotlin中另一个有用的约定是，如果函数字面值只有一个参数， 那么它的声明可以省略（连同 `->`），其名称是 `it` 。

代码示例：

```
>>> val list = listOf(1,2,3,4,5)
>>> list.map { it * 2 }
[2, 4, 6, 8, 10]
```

8.2.12 闭包（Closure）

Lambda 表达式或者匿名函数，以及局部函数和对象表达式（object declarations）可以访问其 闭包， 即在外部作用域中声明的变量。 与 Java 不同的是可以修改闭包中捕获的变量：

```
fun sumGTZero(c: Iterable<Int>): Int {
    var sum = 0
    c.filter { it > 0 }.forEach {
        sum += it
    }
    return sum
}

val list = listOf(1, 2, 3, 4, 5)
sumGTZero(list) // 输出 15
```

我们再使用闭包来写一个使用Java中的Thread接口的例子：

```
fun closureDemo() {
    Thread({
        for (i in 1..10) {
            println("I = $i")
            Thread.sleep(1000)
        }

    }).start()

    Thread({
        for (j in 10..20) {
            println("J = $j")
            Thread.sleep(2000)
        }
        Thread.sleep(1000)
    }).start()
}
```

一个输出：

```
I = 1
J = 10
I = 2
I = 3
...
J = 20
```

8.2.13 带接收者的函数字面值

Kotlin 提供了使用指定的 接收者对象 调用函数字面值的功能。

使用匿名函数的语法，我们可以直接指定函数字面值的接收者类型。

下面我们使用带接收者的函数类型声明一个变量，并在之后使用它。代码示例：

```
>>> val sum = fun Int.(other: Int): Int = this + other
>>> 1.sum(1)
2
```

当接收者类型可以从上下文推断时，lambda 表达式可以用作带接收者的函数字面值。

```
class HTML {
    fun body() {
        println("HTML BODY")
    }
}
```

```
fun html(init: HTML.() -> Unit): HTML { // HTML.()中的HTML是接受者类型
    val html = HTML()  // 创建接收者对象
    html.init()        // 将该接收者对象传给该 lambda
    return html
}
```

测试代码：

```
html {
    body()
}
```

输出：HTML BODY

使用这个特性，我们可以构建一个HTML的DSL语言。

8.2.14 具体化的类型参数

有时候我们需要访问一个参数类型：

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

在这里我们向上遍历一棵树并且检查每个节点是不是特定的类型。 这都没有问题，但是调用处不是很优雅：

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

我们真正想要的只是传一个类型给该函数，即像这样调用它：

```
treeNode.findParentOfType<MyTreeNode>()
```

为能够这么做，内联函数支持具体化的类型参数，于是我们可以这样写：

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
```

```
        p = p.parent
    }
    return p as T?
}
```

我们使用 `reified` 修饰符来限定类型参数，现在可以在函数内部访问它了， 几乎就像是一个普通的类一样。由于函数是内联的，不需要反射，正常的操作符如 `!is` 和 `as` 现在都能用了。

虽然在许多情况下可能不需要反射，但我们仍然可以对一个具体化的类型参数使用它：

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

普通的函数（未标记为内联函数的）没有具体化参数。

8.2.10 尾递归tailrec

Kotlin 支持一种称为尾递归的函数式编程风格。 这允许一些通常用循环写的算法改用递归函数来写，而无堆栈溢出的风险。 当一个函数用 tailrec 修饰符标记并满足所需的形式时，编译器会优化该递归，生成一个快速而高效的基于循环的版本。

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x)) // 函数必须将其自身调用作为它执行的最后一个操作
```

这段代码计算余弦的不动点（fixpoint of cosine），这是一个数学常数。 它只是重复地从 1.0 开始调用 Math.cos，直到结果不再改变，产生0.7390851332151607的结果。最终代码相当于这种更传统风格的代码：

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

要符合 tailrec 修饰符的条件的话，函数必须将其自身调用作为它执行的最后一个操作。在递归调用后有更多代码时，不能使用尾递归，并且不能用在 try/catch/finally 块中。尾部递归在 JVM 后端中支持。

Kotlin 还为集合类引入了许多扩展函数。例如，使用 map() 和 filter() 函数可以流畅地操纵数据，具体的函数的使用以及示例我们已经在 集合类 章节中介绍。

本章小结

本章我们一起学习了函数式编程的简史、Lambda演算、Y组合子与递归等核心函数式的编程思想等相关内容。然后重点介绍了在Kotlin中如何使用函数式风格编程，其中重点介绍了Kotlin中函数的相关知识，以及高阶函数、Lambda表达式、闭包等核心语法，并给出相应的实例说明。

我们将在下一章 中介绍Kotlin的 轻量级线程：协程（Coroutines）的相关知识，我们将看到在Kotlin中，程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。

本章示例代码工程：https://github.com/EasyKotlin/chapter8_fp

第9章 轻量级线程：协程

在常用的并发模型中，多进程、多线程、分布式是最普遍的，不过近些年来逐渐有一些语言以first-class或者library的形式提供对基于协程的并发模型的支持。其中比较典型的有Scheme、Lua、Python、Perl、Go等以first-class的方式提供对协程的支持。

同样地，Kotlin也支持协程。

本章我们主要介绍：

- 什么是协程
- 协程的用法实例
- 挂起函数
- 通道与管道
- 协程的实现原理
- coroutine库等

9.1 协程简介

从硬件发展来看，从最初的单核单CPU，到单核多CPU，多核多CPU，似乎已经到了极限了，但是单核CPU性能却还在不断提升。如果将程序分为IO密集型应用和CPU密集型应用，二者的发展历程大致如下：

> IO密集型应用: 多进程->多线程->事件驱动->协程
>
> CPU密集型应用:多进程-->多线程

如果说多进程对于多CPU，多线程对应多核CPU，那么事件驱动和协程则是在充分挖掘不断提高性能的单核CPU的潜力。

常见的有性能瓶颈的API (例如网络 IO、文件 IO、CPU 或 GPU 密集型任务等)，要求调用者阻塞（blocking）直到它们完成才能进行下一步。后来，我们又使用异步回调的方式来实现非阻塞，但是异步回调代码写起来并不简单。

协程提供了一种避免阻塞线程并用更简单、更可控的操作替代线程阻塞的方法：协程挂起。

协程主要是让原来要使用 “异步+回调方式” 写出来的复杂代码, 简化成可以用看似同步的方式写出来（对线程的操作进一步抽象）。这样我们就可以按串行的思维模型去组织原本分散在不同上下文中的代码逻辑，而不需要去处理复杂的状态同步问题。

协程最早的描述是由Melvin Conway于1958年给出： “subroutines who act as the master program”（与主程序行为类似的子例程)。此后他又在博士论文中给出了如下定义：

> - 数据在后续调用中始终保持（ The values of data local to a coroutine persist between successive calls 协程的局部）

> - 当控制流程离开时，协程的执行被挂起，此后控制流程再次进入这个协程时，这个协程只应从上次离开挂起的地方继续（The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage）。

协程的实现要维护一组局部状态，在重新进入协程前，保证这些状态不被改变，从而能顺利定位到之前的位置。

协程可以用来解决很多问题，比如nodejs的嵌套回调，Erlang以及Golang的并发模型实现等。

实质上，协程（coroutine）是一种用户态的轻量级线程。它由协程构建器（launch coroutine builder）启动。

下面我们通过代码实践来学习协程的相关内容。

9.1.1 搭建协程代码工程

首先，我们来新建一个Kotlin Gradle工程。生成标准gradle工程后，在配置文件build.gradle中，配置kotlinx-coroutines-core依赖：

添加 dependencies :

```
compile 'org.jetbrains.kotlinx:kotlinx-coroutines-core:0.16'
```

kotlinx-coroutines还提供了下面的模块：

```
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-jdk8', version: '0.16
'
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-nio', version: '0.16'
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-reactive', version: '
0.16'
```

我们使用Kotlin最新的1.1.3-2 版本:

```
buildscript {
    ext.kotlin_version = '1.1.3-2'
    ...
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

其中，kotlin-gradle-plugin是Kotlin集成Gradle的插件。

另外，配置一下JCenter 的仓库:

```
repositories {
    jcenter()
}
```

9.1.2 简单协程示例

下面我们先来看一个简单的协程示例。

运行下面的代码：

```
fun firstCoroutineDemo0() {
    launch(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("Hello,")
    }
    println("World!")
    Thread.sleep(5000L)
}
```

你将会发现输出：

```
World!
Hello,
```

上面的这段代码：

```
launch(CommonPool) {
            delay(3000L, TimeUnit.MILLISECONDS)
            println("Hello,")
}
```

等价于：

```
launch(CommonPool, CoroutineStart.DEFAULT, {
            delay(3000L, TimeUnit.MILLISECONDS)
            println("Hello, ")
})
```

9.1.3 launch函数

这个launch函数定义在kotlinx.coroutines.experimental下面。

```
public fun launch(
```

```
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.initParentJob(context[Job])
    start(block, coroutine, coroutine)
    return coroutine
}
```

launch函数有3个入参：context、start、block，这些函数参数分别说明如下：

参数	说明
context	协程上下文
start	协程启动选项
block	协程真正要执行的代码块，必须是suspend修饰的挂起函数

这个launch函数返回一个Job类型，Job是协程创建的后台任务的概念，它持有该协程的引用。Job接口实际上继承自CoroutineContext类型。一个Job有如下三种状态：

State	isActive	isCompleted
New (optional initial state) 新建 （可选的初始状态）	`false`	`false`
Active (default initial state) 活动中（默认初始状态）	`true`	`false`
Completed (final state) 已结束（最终状态）	`false`	`true`

也就是说，launch函数它以非阻塞（non-blocking）当前线程的方式，启动一个新的协程后台任务，并返回一个Job类型的对象作为当前协程的引用。

另外，这里的delay()函数类似Thread.sleep()的功能，但更好的是：它不会阻塞线程，而只是挂起协程本身。当协程在等待时，线程将返回到池中, 当等待完成时, 协同将在池中的空闲线程上恢复。

9.1.4 CommonPool：共享线程池

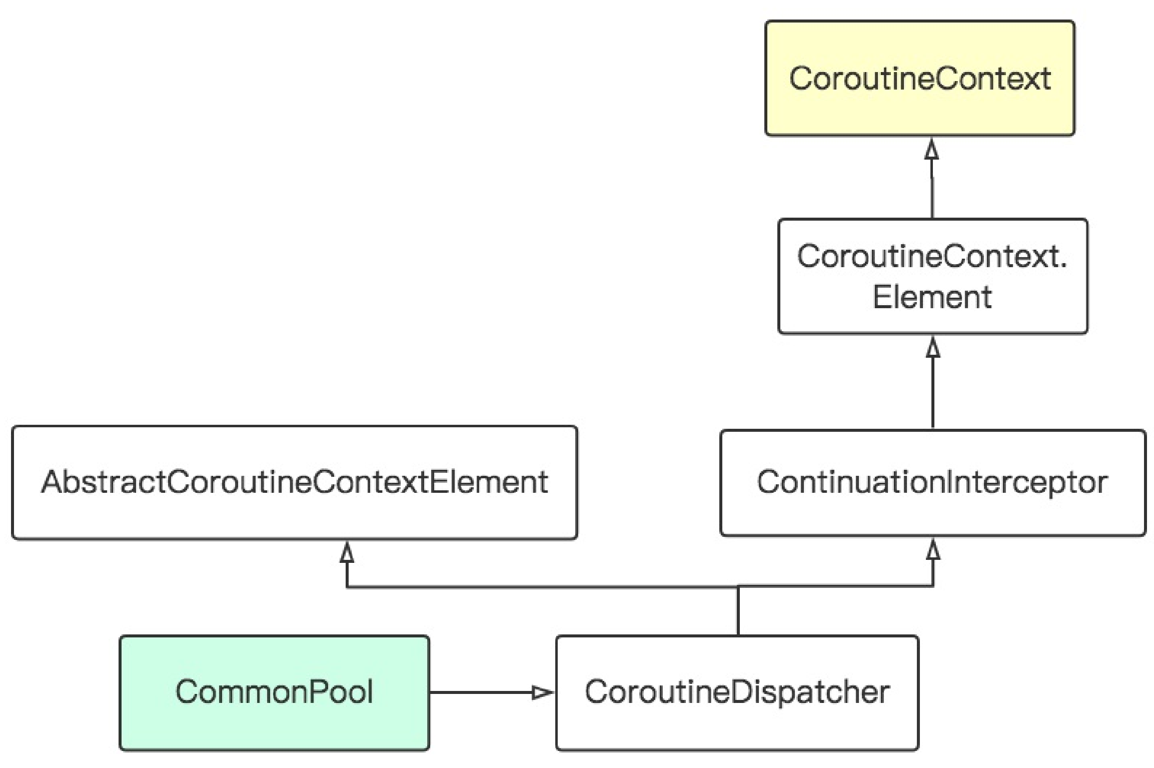
我们再来看一下 `launch(CommonPool) {...}` 这段代码。

首先，这个CommonPool是代表共享线程池，它的主要作用是来调度计算密集型任务的协程的执行。它的实现使用的是java.util.concurrent包下面的API。它首先尝试创建一个 `java.util.concurrent.ForkJoinPool` （ForkJoinPool是一个可以执行ForkJoinTask的ExcuteService，它采用了work-stealing模式：所有在池中的线程尝试去执行其他线程创建的子任务，这样很少有线程处于空闲状态，更加高效）；如果不可用，就使

用 `java.util.concurrent.Executors` 来创建一个普通的线程池： `Executors.newFixedThreadPool` 。相关代码在kotlinx/coroutines/experimental/CommonPool.kt中：

```
private fun createPool(): ExecutorService {
    val fjpClass = Try { Class.forName("java.util.concurrent.ForkJoinPool") }
        ?: return createPlainPool()
    if (!usePrivatePool) {
        Try { fjpClass.getMethod("commonPool")?.invoke(null) as? ExecutorService }
            ?.let { return it }
    }
    Try { fjpClass.getConstructor(Int::class.java).newInstance(defaultParallelism()) as
? ExecutorService }
        ?. let { return it }
    return createPlainPool()
}

private fun createPlainPool(): ExecutorService {
    val threadId = AtomicInteger()
    return Executors.newFixedThreadPool(defaultParallelism()) {
        Thread(it, "CommonPool-worker-${threadId.incrementAndGet()}").apply { isDaemon
= true }
    }
}
```

这个CommonPool对象类是CoroutineContext的子类型。它们的类型集成层次结构如下：



9.1.5 挂起函数

代码块中的 `delay(3000L, TimeUnit.MILLISECONDS)` 函数，是一个用suspend关键字修饰的函数，我们称之为挂起函数。挂起函数只能从协程代码内部调用，普通的非协程的代码不能调用。

挂起函数只允许由协程或者另外一个挂起函数里面调用, 例如我们在协程代码中调用一个挂起函数，代码示例如下：

```
suspend fun runCoroutineDemo() {
    run(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("suspend,")
    }
    println("runCoroutineDemo!")
    Thread.sleep(5000L)
}

fun callSuspendFun() {
    launch(CommonPool) {
        runCoroutineDemo()
    }
}
```

如果我们用Java中的Thread类来写类似功能的代码，上面的代码可以写成这样：

```
fun threadDemo0() {
    Thread({
        Thread.sleep(3000L)
        println("Hello,")
    }).start()

    println("World!")
    Thread.sleep(5000L)
}
```

输出结果也是：

World! Hello,

另外， 我们不能使用Thread来启动协程代码。例如下面的写法编译器会报错：

```
/**
 * 错误反例：用线程调用协程 error
 */
fun threadCoroutineDemo() {
    Thread({
        delay(3000L, TimeUnit.MILLISECONDS) // error, Suspend functions are only allowe
d to be called from a coroutine or another suspend function
        println("Hello,")
    })
    println("World!")
    Thread.sleep(5000L)
}
```

9.2 桥接 阻塞和非阻塞

上面的例子中，我们给出的是使用非阻塞的delay函数，同时有使用了阻塞的Thread.sleep函数，这样代码写在一起可读性不是那么地好。让我们来使用纯的Kotlin的协程代码来实现上面的 阻塞+非阻塞 的例子（不用Thread）。

9.2.1 runBlocking函数

Kotlin中提供了runBlocking函数来实现类似主协程的功能：

```
fun main(args: Array<String>) = runBlocking<Unit> {
    // 主协程
    println("${format(Date())}: T0")

    // 启动主协程
    launch(CommonPool) {
```

```
        //在common thread pool中创建协程
        println("${format(Date())}: T1")
        delay(3000L)
        println("${format(Date())}: T2 Hello,")
    }
    println("${format(Date())}: T3 World!") //  当子协程被delay，主协程仍然继续运行

    delay(5000L)

    println("${format(Date())}: T4")
}
```

运行结果：

```
14:37:59.640: T0
14:37:59.721: T1
14:37:59.721: T3 World!
14:38:02.763: T2 Hello,
14:38:04.738: T4
```

可以发现，运行结果跟之前的是一样的，但是我们没有使用Thread.sleep，我们只使用了非阻塞的delay函数。如果main函数不加 `= runBlocking<Unit>` ，那么我们是不能在main函数体内调用delay(5000L)的。

如果这个阻塞的线程被中断，runBlocking抛出InterruptedException异常。

该runBlocking函数不是用来当做普通协程函数使用的，它的设计主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码的, 例如用在 `main` 函数中，或者用于测试用例代码中。

```
@RunWith(JUnit4::class)
class RunBlockingTest {

    @Test fun testRunBlocking() = runBlocking<Unit> {
        // 这样我们就可以在这里调用任何suspend fun了
        launch(CommonPool) {
            delay(3000L)
        }
        delay(5000L)
    }
}
```

9.3 等待一个任务执行完毕

我们先来看一段代码：

```
fun firstCoroutineDemo() {
    launch(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("[firstCoroutineDemo] Hello, 1")
    }

    launch(CommonPool, CoroutineStart.DEFAULT, {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("[firstCoroutineDemo] Hello, 2")
    })
    println("[firstCoroutineDemo] World!")
}
```

运行这段代码，我们会发现只输出：

```
[firstCoroutineDemo] World!
```

这是为什么？

为了弄清上面的代码执行的内部过程，我们打印一些日志看下：

```
fun testJoinCoroutine() = runBlocking<Unit> {
     // Start a coroutine
     val c1 = launch(CommonPool) {
         println("C1 Thread: ${Thread.currentThread()}")
         println("C1 Start")
         delay(3000L)
         println("C1 World! 1")
     }

     val c2 = launch(CommonPool) {
         println("C2 Thread: ${Thread.currentThread()}")
         println("C2 Start")
         delay(5000L)
         println("C2 World! 2")
     }

     println("Main Thread: ${Thread.currentThread()}")
     println("Hello,")
     println("Hi,")
     println("c1 is active: ${c1.isActive}  ${c1.isCompleted}")
     println("c2 is active: ${c2.isActive}  ${c2.isCompleted}")
}
```

再次运行：

```
C1 Thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
C1 Start
C2 Thread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
C2 Start
Main Thread: Thread[main,5,main]
Hello,
Hi,
c1 is active: true  false
c2 is active: true  false
```

我们可以看到，这里的C1、C2代码也开始执行了，使用的是 `ForkJoinPool.commonPool-worker` 线程池中的worker线程。但是，我们在代码执行到最后打印出这两个协程的状态isCompleted都是false，这表明我们的C1、C2的代码，在Main Thread结束的时刻（此时的运行main函数的Java进程也退出了），还没有执行完毕，然后就跟着主线程一起退出结束了。

所以我们可以得出结论：运行 main () 函数的主线程， 必须要等到我们的协程完成之前结束，否则我们的程序在 打印Hello, 1和Hello, 2之前就直接结束掉了。

我们怎样让这两个协程参与到主线程的时间顺序里呢？我们可以使用 `join` , 让主线程一直等到当前协程执行完毕再结束, 例如下面的这段代码

```
fun testJoinCoroutine() = runBlocking<Unit> {
    // Start a coroutine
    val c1 = launch(CommonPool) {
        println("C1 Thread: ${Thread.currentThread()}")
        println("C1 Start")
        delay(3000L)
        println("C1 World! 1")
    }

    val c2 = launch(CommonPool) {
        println("C2 Thread: ${Thread.currentThread()}")
        println("C2 Start")
        delay(5000L)
        println("C2 World! 2")
    }

    println("Main Thread: ${Thread.currentThread()}")
    println("Hello,")

    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")

    c1.join() // the main thread will wait until child coroutine completes
    println("Hi,")
    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")
```

```
    c2.join() // the main thread will wait until child coroutine completes
    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")
}
```

将会输出：

```
C1 Thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
C1 Start
C2 Thread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
C2 Start
Main Thread: Thread[main,5,main]
Hello,
c1 is active: true  isCompleted: false
c2 is active: true  isCompleted: false
C1 World! 1
Hi,
c1 is active: false  isCompleted: true
c2 is active: true  isCompleted: false
C2 World! 2
c1 is active: false  isCompleted: true
c2 is active: false  isCompleted: true
```

通常，良好的代码风格我们会把一个单独的逻辑放到一个独立的函数中，我们可以重构上面的代码如下：

```
fun testJoinCoroutine2() = runBlocking<Unit> {
    // Start a coroutine
    val c1 = launch(CommonPool) {
        fc1()
    }

    val c2 = launch(CommonPool) {
        fc2()
    }
    ...
}

private suspend fun fc2() {
    println("C2 Thread: ${Thread.currentThread()}")
    println("C2 Start")
    delay(5000L)
    println("C2 World! 2")
}

private suspend fun fc1() {
    println("C1 Thread: ${Thread.currentThread()}")
```

```
        println("C1 Start")
        delay(3000L)
        println("C1 World! 1")
    }
```

可以看出，我们这里的fc1, fc2函数是suspend fun。

9.4 协程是轻量级的

直接运行下面的代码：

```
fun testThread() {
    val jobs = List(100_1000) {
        Thread({
            Thread.sleep(1000L)
            print(".")
        })
    }
    jobs.forEach { it.start() }
    jobs.forEach { it.join() }
}
```

我们应该会看到输出报错：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thre
ad
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at com.easy.kotlin.LightWeightCoroutinesDemo.testThread(LightWeightCoroutinesDemo.k
t:30)
    at com.easy.kotlin.LightWeightCoroutinesDemoKt.main(LightWeightCoroutinesDemo.kt:40
)
........................................................................................
....
```

我们这里直接启动了100，000个线程，并join到一起打印".", 不出意外的我们收到了 `java.lang.OutOfMemoryError` 。

这个异常问题本质原因是我们创建了太多的线程，而能创建的线程数是有限制的，导致了异常的发生。在Java中， 当我们创建一个线程的时候，虚拟机会在JVM内存创建一个Thread对象同时创建一个操作系统线程，而这个系统线程的内存用的不是JVMMemory，而是系统中剩下的内存(MaxProcessMemory - JVMMemory - ReservedOsMemory)。 能创建的线程数的具体计算公式如下：

> Number of Threads = (MaxProcessMemory - JVMMemory - ReservedOsMemory) / (ThreadStackSize)

其中，参数说明如下：

参数	说明
MaxProcessMemory	指的是一个进程的最大内存
JVMMemory	JVM内存
ReservedOsMemory	保留的操作系统内存
ThreadStackSize	线程栈的大小

我们通常在优化这种问题的时候，要么是采用减小thread stack的大小的方法，要么是采用减小heap或permgen初始分配的大小方法等方式来临时解决问题。

在协程中，情况完全就不一样了。我们看一下实现上面的逻辑的协程代码：

```
fun testLightWeightCoroutine() = runBlocking {
    val jobs = List(100_000) {
        // create a lot of coroutines and list their jobs
        launch(CommonPool) {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() } // wait for all jobs to complete
}
```

运行上面的代码，我们将看到输出：

```
START: 21:22:28.913
.....................
.....................(100000个)
.....END: 21:22:30.956
```

上面的程序在2s左右的时间内正确执行完毕。

9.5 协程 vs 守护线程

在Java中有两类线程：用户线程 (User Thread)、守护线程 (Daemon Thread)。

所谓守护 线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。

我们来看一段Thread的守护线程的代码：

```
fun testDaemon2() {
    val t = Thread({
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            Thread.sleep(500L)
        }
    })
    t.isDaemon = true // 必须在启动线程前调用,否则会报错: Exception in thread "main" java.lang.IllegalThreadStateException
    t.start()
    Thread.sleep(2000L) // just quit after delay
}
```

这段代码启动一个线程，并设置为守护线程。线程内部是间隔500ms 重复打印100次输出。外部主线程睡眠2s。

运行这段代码，将会输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
```

协程跟守护线程很像，用协程来写上面的逻辑，代码如下：

```
fun testDaemon1() = runBlocking {
    launch(CommonPool) {
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(2000L) // just quit after delay
}
```

运行这段代码，我们发现也输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
```

我们可以看出，活动的协程不会使进程保持活动状态。它们的行为就像守护程序线程。

9.6 协程执行的取消

我们知道，启动函数launch返回一个Job引用当前协程，该Job引用可用于取消正在运行协程:

```
fun testCancellation() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        repeat(1000) { i ->
            println("I'm sleeping $i ... CurrentThread: ${Thread.currentThread()}")
            delay(500L)
        }
    }
    delay(1300L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Job is alive: ${job.isActive}  Job is completed: ${job.isCompleted}")
    val b1 = job.cancel() // cancels the job
    println("job cancel: $b1")
    delay(1300L)
    println("Job is alive: ${job.isActive}  Job is completed: ${job.isCompleted}")

    val b2 = job.cancel() // cancels the job, job already canceld, return false
    println("job cancel: $b2")

    println("main: Now I can quit.")
}
```

运行上面的代码，将会输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Job is alive: true  Job is completed: false
job cancel: true
Job is alive: false  Job is completed: true
job cancel: false
main: Now I can quit.
```

我们可以看出，当job还在运行时，isAlive是true，isCompleted是false。当调用job.cancel取消该协程任务，cancel函数本身返回true, 此时协程的打印动作就停止了。此时，job的状态是isAlive是false，isCompleted是true。 如果，再次调用job.cancel函数，我们将会看到cancel函数返回的是false。

9.6.1 计算代码的协程取消失效

kotlinx 协程的所有suspend函数都是可以取消的。我们可以通过job的isActive状态来判断协程的状态，或者检查手否有抛出 CancellationException 时取消。

例如，协程正工作在循环计算中，并且不检查协程当前的状态, 那么调用cancel来取消协程将无法停止协程的运行, 如下面的示例所示:

```
fun testCooperativeCancellation1() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        var nextPrintTime = 0L
        var i = 0
        while (i < 20) { // computation loop
            val currentTime = System.currentTimeMillis()
            if (currentTime >= nextPrintTime) {
                println("I'm sleeping ${i++} ... CurrentThread: ${Thread.currentThread(
)}")
                nextPrintTime = currentTime + 500L
            }
        }
    }
    delay(3000L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCo
mpleted}")

    val b1 = job.cancel() // cancels the job
    println("job cancel1: $b1")
    println("After Cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCom
pleted}")

    delay(30000L)

    val b2 = job.cancel() // cancels the job, job already canceld, return false
    println("job cancel2: $b2")

    println("main: Now I can quit.")
}
```

运行上面的代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
...
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
```

```
I'm sleeping 7 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
...
I'm sleeping 18 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 19 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
job cancel2: false
main: Now I can quit.
```

我们可以看出，即使我们调用了cancel函数，当前的job状态isAlive是false了，但是协程的代码依然一直在运行，并没有停止。

9.6.2 计算代码协程的有效取消

有两种方法可以使计算代码取消成功。

方法一：显式检查取消状态isActive

我们直接给出实现的代码：

```
fun testCooperativeCancellation2() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        var nextPrintTime = 0L
        var i = 0
        while (i < 20) { // computation loop

            if (!isActive) {
                return@launch
            }

            val currentTime = System.currentTimeMillis()
            if (currentTime >= nextPrintTime) {
                println("I'm sleeping ${i++} ... CurrentThread: ${Thread.currentThread(
)}")
                nextPrintTime = currentTime + 500L
            }
        }
    }
    delay(3000L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCo
mpleted}")
    val b1 = job.cancel() // cancels the job
    println("job cancel1: $b1")
    println("After Cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCom
pleted}")

    delay(3000L)
    val b2 = job.cancel() // cancels the job, job already canceld, return false
```

```
    println("job cancel2: $b2")

    println("main: Now I can quit.")
}
```

运行这段代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 3 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 4 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 5 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
job cancel2: false
main: Now I can quit.
```

正如您所看到的, 现在这个循环可以被取消了。这里的isActive属性是CoroutineScope中的属性。这个接口的定义是：

```
public interface CoroutineScope {
    public val isActive: Boolean
    public val context: CoroutineContext
}
```

该接口用于通用协程构建器的接收器，以便协程中的代码可以方便的访问其isActive状态值（取消状态），以及其上下文CoroutineContext信息。

方法二：循环调用一个挂起函数yield()

该方法实质上是通过job的isCompleted状态值来捕获CancellationException完成取消功能。

我们只需要在while循环体中循环调用yield()来检查该job的取消状态，如果已经被取消，那么isCompleted值将会是true，yield函数就直接抛出CancellationException异常，从而完成取消的功能：

```
val job = launch(CommonPool) {
    var nextPrintTime = 0L
    var i = 0
    while (i < 20) { // computation loop

        yield()
```

```
            val currentTime = System.currentTimeMillis()
            if (currentTime >= nextPrintTime) {
                println("I'm sleeping ${i++} ... CurrentThread: ${Thread.currentThread()}")
                nextPrintTime = currentTime + 500L
            }
        }
    }
```

运行上面的代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
I'm sleeping 3 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 4 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 5 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
job cancel2: false
main: Now I can quit.
```

如果我们想看看yield函数抛出的异常，我们可以加上try catch打印出日志：

```
try {
    yield()
} catch (e: Exception) {
    println("$i ${e.message}")
}
```

我们可以看到类似：Job was cancelled 这样的信息。

这个yield函数的实现是：

```
suspend fun yield(): Unit = suspendCoroutineOrReturn sc@ { cont ->
    val context = cont.context
    val job = context[Job]
    if (job != null && job.isCompleted) throw job.getCompletionException()
    if (cont !is DispatchedContinuation<Unit>) return@sc Unit
    if (!cont.dispatcher.isDispatchNeeded(context)) return@sc Unit
    cont.dispatchYield(job, Unit)
    COROUTINE_SUSPENDED
}
```

如果调用此挂起函数时，当前协程的Job已经完成 (isActive = false, isCompleted = true)，当前协程将以CancellationException取消。

9.6.3 在finally中的协程代码

当我们取消一个协程任务时，如果有 `try {...} finally {...}` 代码块，那么finally {...}中的代码会被正常执行完毕：

```
fun finallyCancelDemo() = runBlocking {
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("I'm running finally")
        }
    }
    delay(2000L)
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCo
mpleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCom
pleted}")
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行这段代码，输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
Before cancel, Job is alive: true  Job is completed: false
I'm running finally
After cancel, Job is alive: false  Job is completed: true
main: Now I can quit.
```

我们可以看出，在调用cancel之后，就算当前协程任务Job已经结束了， `finally{...}` 中的代码依然被正常执行。

但是，如果我们在 `finally{...}` 中放入挂起函数：

```
fun finallyCancelDemo() = runBlocking {
```

```
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("I'm running finally")
            delay(1000L)
            println("And I've delayed for 1 sec ?")
        }
    }
    delay(2000L)
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCo
mpleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive}  Job is completed: ${job.isCom
pleted}")
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行上述代码，我们将会发现只输出了一句：I'm running finally。因为主线程在挂起函数 `delay(1000L)` 以及后面的打印逻辑还没执行完，就已经结束退出。

```
} finally {
    println("I'm running finally")
    delay(1000L)
    println("And I've delayed for 1 sec ?")
}
```

9.6.4 协程执行不可取消的代码块

如果我们想要上面的例子中的 `finally{...}` 完整执行，不被取消函数操作所影响，我们可以使用 run 函数和 NonCancellable 上下文将相应的代码包装在 run (NonCancellable) {...} 中, 如下面的示例所示:

```
fun testNonCancellable() = runBlocking {
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            run(NonCancellable) {
```

```
                println("I'm running finally")
                delay(1000L)
                println("And I've just delayed for 1 sec because I'm non-cancellable")
            }
        }
    }
    delay(2000L)
    println("main: I'm tired of waiting!")
    job.cancel()
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
main: I'm tired of waiting!
I'm running finally
And I've just delayed for 1 sec because I'm non-cancellable
main: Now I can quit.
```

9.7 设置协程超时时间

我们通常取消协同执行的原因给协程的执行时间设定一个执行时间上限。我们也可以使用 withTimeout 函数来给一个协程任务的执行设定最大执行时间，超出这个时间，就直接终止掉。代码示例如下:

```
fun testTimeouts() = runBlocking {
    withTimeout(3000L) {
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
}
```

运行上述代码，我们将会看到如下输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
```

```
I'm sleeping 4 ...
I'm sleeping 5 ...
Exception in thread "main" kotlinx.coroutines.experimental.TimeoutException: Timed out
waiting for 3000 MILLISECONDS
    at kotlinx.coroutines.experimental.TimeoutExceptionCoroutine.run(Scheduled.kt:110)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunnableTask.invoke(EventLo
op.kt:199)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunnableTask.invoke(EventLo
op.kt:195)
    at kotlinx.coroutines.experimental.EventLoopImpl.processNextEvent(EventLoop.kt:111)
    at kotlinx.coroutines.experimental.BlockingCoroutine.joinBlocking(Builders.kt:205)
    at kotlinx.coroutines.experimental.BuildersKt.runBlocking(Builders.kt:150)
    at kotlinx.coroutines.experimental.BuildersKt.runBlocking$default(Builders.kt:142)
    at com.easy.kotlin.CancellingCoroutineDemo.testTimeouts(CancellingCoroutineDemo.kt:
169)
    at com.easy.kotlin.CancellingCoroutineDemoKt.main(CancellingCoroutineDemo.kt:193)
```

由 withTimeout 抛出的 TimeoutException 是 CancellationException 的一个子类。这个TimeoutException类型定义如下：

```
private class TimeoutException(
    time: Long,
    unit: TimeUnit,
    @JvmField val coroutine: Job
) : CancellationException("Timed out waiting for $time $unit")
```

如果您需要在超时时执行一些附加操作, 则可以把逻辑放在 try {...} catch (e: CancellationException) {...} 代码块中。例如：

```
try {
    ccd.testTimeouts()
} catch (e: CancellationException) {
    println("I am timed out!")
}
```

9.8 挂起函数的组合执行

本节我们介绍挂起函数组合的各种方法。

9.8.1 按默认顺序执行

假设我们有两个在别处定义的挂起函数：

```
suspend fun doJob1(): Int {
```

```
    println("Doing Job1 ...")
    delay(1000L) // 此处模拟我们的工作代码
    println("Job1 Done")
    return 10
}

suspend fun doJob2(): Int {
    println("Doing Job2 ...")
    delay(1000L) // 此处模拟我们的工作代码
    println("Job2 Done")
    return 20
}
```

如果需要依次调用它们, 我们只需要使用正常的顺序调用, 因为协同中的代码 (就像在常规代码中一样) 是默认的顺序执行。下面的示例通过测量执行两个挂起函数所需的总时间来演示:

```
fun testSequential() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = doJob1()
        val two = doJob2()
        println("[testSequential] 最终结果： ${one + two}")
    }
    println("[testSequential] Completed in $time ms")
}
```

执行上面的代码，我们将得到输出：

```
Doing Job1 ...
Job1 Done
Doing Job2 ...
Job2 Done
[testSequential] 最终结果： 30
[testSequential] Completed in 6023 ms
```

可以看出，我们的代码是跟普通的代码一样顺序执行下去。

9.8.2 使用async异步并发执行

上面的例子中，如果在调用 doJob1 和 doJob2 之间没有时序上的依赖关系, 并且我们希望通过同时并发地执行这两个函数来更快地得到答案, 那该怎么办呢？这个时候，我们就可以使用async来实现异步。代码示例如下：

```
fun testAsync() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = async(CommonPool) { doJob1() }
```

```
        val two = async(CommonPool) { doJob2() }
        println("最终结果： ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}
```

如果跟上面同步的代码一起执行对比，我们可以看到如下输出：

```
Doing Job1 ...
Job1 Done
Doing Job2 ...
Job2 Done
[testSequential] 最终结果： 30
[testSequential] Completed in 6023 ms
Doing Job1 ...
Doing Job2 ...
Job1 Done
Job2 Done
[testAsync] 最终结果： 30
[testAsync] Completed in 3032 ms
```

我们可以看出，使用async函数，我们的两个Job并发的执行了，并发花的时间要比顺序的执行的要快将近两倍。因为，我们有两个任务在并发的执行。

从概念上讲, async跟launch类似, 它启动一个协程, 它与其他协程并发地执行。

不同之处在于, launch返回一个任务Job对象, 不带任何结果值；而async返回一个延迟任务对象Deferred，一种轻量级的非阻塞性future, 它表示后面会提供结果。

在上面的示例代码中，我们使用Deferred调用 await() 函数来获得其最终结果。另外，延迟任务Deferred也是Job类型, 它继承自Job，所以它也有isActive、isCompleted属性，也有join()、cancel()函数，因此我们也可以在需要时取消它。Deferred接口定义如下：

```
public interface Deferred<out T> : Job {
    val isCompletedExceptionally: Boolean
    val isCancelled: Boolean
    public suspend fun await(): T
    public fun <R> registerSelectAwait(select: SelectInstance<R>, block: suspend (T) ->
 R)
    public fun getCompleted(): T
    @Deprecated(message = "Use `isActive`", replaceWith = ReplaceWith("isActive"))
    public val isComputing: Boolean get() = isActive
}
```

其中，常用的属性和函数说明如下：

名称	说明

isCompletedExceptionally	当协程在计算过程中有异常failed 或被取消，返回true。 这也意味着 `isActive` 等于 `false` ，同时 `isCompleted` 等于 `true`
isCancelled	如果当前延迟任务被取消，返回true
suspend fun await()	等待此延迟任务完成，而不阻塞线程；如果延迟任务完成, 则返回结果值或引发相应的异常。

延迟任务对象Deferred的状态与对应的属性值如下表所示：

状态	isActive	isCompleted	isCompletedExceptionally	isCancelled
New (可选初始状态)	`false`	`false`	`false`	`false`
Active (默认初始状态)	`true`	`false`	`false`	`false`
Resolved (最终状态)	`false`	`true`	`false`	`false`
Failed (最终状态)	`false`	`true`	`true`	`false`
Cancelled (最终状态)	`false`	`true`	`true`	`true`

9.9 协程上下文与调度器

到这里，我们已经看到了下面这些启动协程的方式：

```
launch(CommonPool) {...}
async(CommonPool) {...}
run(NonCancellable) {...}
```

这里的CommonPool 和 NonCancellable 是协程上下文（coroutine contexts）。本小节我们简单介绍一下自定义协程上下文。

9.9.1 调度和线程

协程上下文包括一个协程调度程序, 它可以指定由哪个线程来执行协程。调度器可以将协程的执行调度到一个线程池，限制在特定的线程中；也可以不作任何限制，让它无约束地运行。请看下面的示例:

```
fun testDispatchersAndThreads() = runBlocking {
    val jobs = arrayListOf<Job>()
    jobs += launch(Unconfined) {
        // 未作限制 -- 将会在 main thread 中执行
        println("Unconfined: I'm working in thread ${Thread.currentThread()}")
    }
```

```
    jobs += launch(context) {
        // 父协程的上下文 ： runBlocking coroutine
        println("context: I'm working in thread ${Thread.currentThread()}")
    }
    jobs += launch(CommonPool) {
        // 调度指派给 ForkJoinPool.commonPool
        println("CommonPool: I'm working in thread ${Thread.currentThread()}")
    }
    jobs += launch(newSingleThreadContext("MyOwnThread")) {
        // 将会在这个协程自己的新线程中执行
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread()
}")
    }
    jobs.forEach { it.join() }
}
```

运行上面的代码，我们将得到以下输出 (可能按不同的顺序):

```
Unconfined: I'm working in thread Thread[main,5,main]
CommonPool: I'm working in thread Thread[ForkJoinPool.commonPool-worker-1,5,main]
newSingleThreadContext: I'm working in thread Thread[MyOwnThread,5,main]
context: I'm working in thread Thread[main,5,main]
```

从上面的结果，我们可以看出： 使用无限制的Unconfined上下文的协程运行在主线程中； 继承了runBlocking {...} 的context的协程继续在主线程中执行； 而CommonPool在ForkJoinPool.commonPool中； 我们使用newSingleThreadContext函数新建的协程上下文，该协程运行在自己的新线程Thread[MyOwnThread,5,main]中。

另外，我们还可以在使用 runBlocking的时候显式指定上下文, 同时使用 run 函数来更改协程的上下文：

```
fun log(msg: String) = println("${Thread.currentThread()} $msg")

fun testRunBlockingWithSpecifiedContext() = runBlocking {
    log("$context")
    log("${context[Job]}")
    log("开始")

    val ctx1 = newSingleThreadContext("线程A")
    val ctx2 = newSingleThreadContext("线程B")
    runBlocking(ctx1) {
        log("Started in Context1")
        run(ctx2) {
            log("Working in Context2")
        }
        log("Back to Context1")
    }
```

```
    log("结束")
}
```

运行输出：

```
Thread[main,5,main] [BlockingCoroutine{Active}@b1bc7ed, EventLoopImpl@7cd84586]
Thread[main,5,main] BlockingCoroutine{Active}@b1bc7ed
Thread[main,5,main] 开始
Thread[线程A,5,main] Started in Context1
Thread[线程B,5,main] Working in Context2
Thread[线程A,5,main] Back to Context1
Thread[main,5,main] 结束
```

9.9.2 父子协程

当我们使用协程A的上下文启动另一个协程B时, B将成为A的子协程。当父协程A任务被取消时, B以及它的所有子协程都会被递归地取消。代码示例如下：

```
fun testChildrenCoroutine()= runBlocking<Unit> {
    val request = launch(CommonPool) {
        log("ContextA1: ${context}")

        val job1 = launch(CommonPool) {
            println("job1: 独立的协程上下文!")
            delay(1000)
            println("job1: 不会受到request.cancel()的影响")
        }
        // 继承父上下文：request的context
        val job2 = launch(context) {
            log("ContextA2: ${context}")
            println("job2: 是request coroutine的子协程")
            delay(1000)
            println("job2: 当request.cancel()，job2也会被取消")
        }
        job1.join()
        job2.join()
    }
    delay(500)
    request.cancel()
    delay(1000)
    println("main: Who has survived request cancellation?")
}
```

运行输出：

```
Thread[ForkJoinPool.commonPool-worker-1,5,main] ContextA1: [StandaloneCoroutine{Active}
@5b646af2, CommonPool]
job1: 独立的协程上下文!
Thread[ForkJoinPool.commonPool-worker-3,5,main] ContextA2: [StandaloneCoroutine{Active}
@75152aa4, CommonPool]
job2: 是request coroutine的子协程
job1: 不会受到request.cancel()的影响
main: Who has survived request cancellation?
```

9.10 通道

延迟对象提供了一种在协程之间传输单个值的方法。而通道（Channel）提供了一种传输数据流的方法。通道是使用 SendChannel 和使用 ReceiveChannel 之间的非阻塞通信。

9.10.1 通道 vs 阻塞队列

通道的概念类似于 阻塞队列（BlockingQueue）。在Java的Concurrent包中，BlockingQueue很好的解决了多线程中如何高效安全“传输”数据的问题。它有两个常用的方法如下：

- E take(): 取走BlockingQueue里排在首位的对象,若BlockingQueue为空, 阻塞进入等待状态直到BlockingQueue有新的数据被加入;
- put(E e): 把对象 e 加到BlockingQueue里, 如果BlockQueue没有空间,则调用此方法的线程被阻塞，直到BlockingQueue里面有空间再继续。

通道跟阻塞队列一个关键的区别是：通道有挂起的操作, 而不是阻塞的, 同时它可以关闭。

代码示例：

```
package com.easy.kotlin

import kotlinx.coroutines.experimental.CommonPool
import kotlinx.coroutines.experimental.channels.Channel
import kotlinx.coroutines.experimental.launch
import kotlinx.coroutines.experimental.runBlocking

/**
 * Created by jack on 2017/7/13.
 */
class ChannelsDemo {
    fun testChannel() = runBlocking<Unit> {
        val channel = Channel<Int>()
        launch(CommonPool) {
            for (x in 1..10) channel.send(x * x)
        }
        println("channel = ${channel}")
```

```
        // here we print five received integers:
        repeat(10) { println(channel.receive()) }
        println("Done!")
    }
}

fun main(args: Array<String>) {
    val cd = ChannelsDemo()
    cd.testChannel()
}
```

运行输出：

```
channel = kotlinx.coroutines.experimental.channels.RendezvousChannel@2e817b38
1
4
9
16
25
36
49
64
81
100
Done!
```

我们可以看出使用 `Channel<Int>()` 背后调用的是会合通道 `RendezvousChannel()` ，会合通道中没有任何缓冲区。send函数被挂起直到另外一个协程调用receive函数， 然后receive函数挂起直到另外一个协程调用send函数。它是一个完全无锁的实现。

9.10.2 关闭通道和迭代遍历元素

与队列不同, 通道可以关闭, 以指示没有更多的元素。在接收端, 可以使用 for 循环从通道接收元素。代码示例：

```
fun testClosingAndIterationChannels() = runBlocking {
    val channel = Channel<Int>()
    launch(CommonPool) {
        for (x in 1..5) channel.send(x * x)
        channel.close() // 我们结束 sending
    }
    // 打印通道中的值，直到通道关闭
    for (x in channel) println(x)
    println("Done!")
}
```

其中， close函数在这个通道上发送一个特殊的 "关闭令牌"。这是一个幂等运算：对此函数的重复调用不起作用, 并返回 "false"。此函数执行后， `isClosedForSend` 返回 "true"。但是, `ReceiveChannel` 的 `isClosedForReceive` 在所有之前发送的元素收到之后才返回 "true"。

我们把上面的代码加入打印日志：

```
fun testClosingAndIterationChannels() = runBlocking {
    val channel = Channel<Int>()
    launch(CommonPool) {
        for (x in 1..5) {
            channel.send(x * x)
        }
        println("Before Close => isClosedForSend = ${channel.isClosedForSend}")
        channel.close() // 我们结束 sending
        println("After Close => isClosedForSend = ${channel.isClosedForSend}")
    }
    // 打印通道中的值，直到通道关闭
    for (x in channel) {
        println("${x} => isClosedForReceive = ${channel.isClosedForReceive}")
    }
    println("Done!  => isClosedForReceive = ${channel.isClosedForReceive}")
}
```

运行输出：

```
1 => isClosedForReceive = false
4 => isClosedForReceive = false
9 => isClosedForReceive = false
16 => isClosedForReceive = false
25 => isClosedForReceive = false
Before Close => isClosedForSend = false
After Close => isClosedForSend = true
Done!  => isClosedForReceive = true
```

9.10.3 生产者-消费者模式

使用协程生成元素序列的模式非常常见。这是在并发代码中经常有的生产者-消费者模式。代码示例：

```
fun produceSquares() = produce<Int>(CommonPool) {
    for (x in 1..7) send(x * x)
}

fun consumeSquares() = runBlocking{
    val squares = produceSquares()
    squares.consumeEach { println(it) }
```

```
    println("Done!")
}
```

这里的produce函数定义如下：

```
public fun <E> produce(
    context: CoroutineContext,
    capacity: Int = 0,
    block: suspend ProducerScope<E>.() -> Unit
): ProducerJob<E> {
    val channel = Channel<E>(capacity)
    return ProducerCoroutine(newCoroutineContext(context), channel).apply {
        initParentJob(context[Job])
        block.startCoroutine(this, this)
    }
}
```

其中，参数说明如下：

参数名	说明
context	协程上下文
capacity	通道缓存容量大小 (默认没有缓存)
block	协程代码块

produce函数会启动一个新的协程, 协程中发送数据到通道来生成数据流，并以 ProducerJob对象返回对协程的引用。ProducerJob继承了Job, ReceiveChannel类型。

9.11 管道

9.11.1 生产无限序列

管道（Pipeline）是一种模式, 我们可以用一个协程生产无限序列:

```
fun produceNumbers() = produce<Long>(CommonPool) {
    var x = 1L
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

我们的消费序列的函数如下：

```
fun produceNumbers() = produce<Long>(CommonPool) {
    var x = 1L
    while (true) send(x++) // infinite stream of integers starting from 1
```

```
}
```

主代码启动并连接整个管线:

```
fun testPipeline() = runBlocking {
    val numbers = produceNumbers() // produces integers from 1 and on
    val squares = consumeNumbers(numbers) // squares integers
    //for (i in 1..6) println(squares.receive())
    while (true) {
        println(squares.receive())
    }
    println("Done!")
    squares.cancel()
    numbers.cancel()
}
```

运行上面的代码，我们将会发现控制台在打印一个无限序列，完全没有停止的意思。

9.11.2 管道与无穷质数序列

我们使用协程管道来生成一个无穷质数序列。

我们从无穷大的自然数序列开始：

```
fun numbersProducer(context: CoroutineContext, start: Int) = produce<Int>(context) {
    var n = start
    while (true) send(n++) // infinite stream of integers from start
}
```

这次我们引入一个显式上下文参数context, 以便调用方可以控制我们的协程运行的位置。

下面的管道将筛选传入的数字流, 过滤掉可以被当前质数整除的所有数字：

```
fun filterPrimes(context: CoroutineContext, numbers: ReceiveChannel<Int>, prime: Int) =
 produce<Int>(context) {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

现在我们通过从2开始, 从当前通道中取一个质数, 并为找到的每个质数启动新的管道阶段, 从而构建出我们的管道:

```
numbersFrom(2) -> filterPrimes(2) -> filterPrimes(3) -> filterPrimes(5) -> filterPrimes
(7) ...
```

测试无穷质数序列：

```
fun producePrimesSequences() = runBlocking {
    var producerJob = numbersProducer(context, 2)

    while (true) {
        val prime = producerJob.receive()
        print("${prime} \t")
        producerJob = filterPrimes(context, producerJob, prime)
    }
}
```

运行上面的代码，我们将会看到控制台一直在无限打印出质数序列：

```
84
85  fun numbersProducer(context: CoroutineContext, start: Int) = produce<Int>(context) {
86      var n = start
87      while (true) send(n++) // infinite stream of integers from start
88  }
89
90  fun filterPrimes(context: CoroutineContext, numbers: ReceiveChannel<Int>, prime: Int) = produce<Int>(context) {
91      for (x in numbers) if (x % prime != 0) send(x)
92  }
93
94  fun producePrimesSequences() = runBlocking {
95      var producerJob = numbersProducer(context, start: 2)
96
97      while (true) {
98          val prime = producerJob.receive()
99          print("${prime} \t")
100         producerJob = filterPrimes(context, producerJob, prime)
101     }
102 }
103 }
```

```
Run  com.easy.kotlin.ChannelsDemoKt
126307 126311 126317 126323 126337 126341 126349 126359 126397 126421 126433 126443 126457 126461 126473 126481 126487
126491 126493 126499 126517 126541 126547 126551 126583 126601 126611 126613 126631 126641 126653 126683 126691 126703
126713 126719 126733 126739 126743 126751 126757 126761 126781 126823 126827 126839 126851 126857 126859 126913 126923
126943 126949 126961 126967 126989 127031 127033 127037 127051 127079 127081 127103 127123 127133 127139 127157 127163
127189 127207 127217 127219 127241 127247 127249 127261 127271 127277 127289 127291 127297 127301 127321 127331 127343
127363 127373 127399 127403 127423 127447 127453 127481 127487 127493 127507 127529 127541 127549 127579 127583 127591
127597 127601 127607 127609 127637 127643 127649 127657 127663 127669 127679 127681 127691 127703 127709 127711 127717
127727 127733 127739 127747 127763 127781 127807 127817 127819 127837 127843 127849 127859 127867 127873 127877 127913
127921 127931 127951 127973 127979 127997 128021 128033 128047 128053 128099 128111 128113 128119 128147 128153 128159
128173 128189 128201 128203 128213 128221 128237 128239 128257 128273 128287 128291 128311 128321 128327 128339 128341
128347 128351 128377 128389 128393 128399 128411 128413 128431 128437 128449 128461 128467 128473 128477 128483 128489
128509 128519 128521 128549 128551 128563 128591 128599 128603 128621 128629 128657 128659 128663 128669 128677 128683
128693 128717 128747 128749 128761 128767 128813 128819 128831 128833 128837 128857 128861 128873 128879 128903 128923
128939 128941 128951 128959 128969 128971 128981 128983 128987 128993 129001 129011 129023 129037 129049 129061 129083
129089 129097 129113 129119 129121 129127 129169 129187 129193 129197 129209 129221 129223 129229
```

9.11.3 通道缓冲区

我们可以给通道设置一个缓冲区：

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val channel = Channel<Int>(4) // 创建一个缓冲区容量为4的通道
    launch(context) {
        repeat(10) {
            println("Sending $it")
            channel.send(it) // 当缓冲区已满的时候， send将会挂起
        }
    }
    delay(1000)
}
```

输出：

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

9.12 构建无穷惰性序列

我们可以使用 buildSequence 序列生成器， 构建一个无穷惰性序列。

```
val fibonacci = buildSequence {
    yield(1L)
    var current = 1L
    var next = 1L
    while (true) {
        yield(next)
        val tmp = current + next
        current = next
        next = tmp
    }
}
```

我们通过buildSequence创建一个协程，生成一个惰性的无穷斐波那契数列。该协程通过调用 yield() 函数来产生连续的斐波纳契数。

我们可以从该序列中取出任何有限的数字列表，例如

```
println(fibonacci.take(16).toList())
```

的结果是：

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

9.13 协程与线程比较

直接先说区别，协程是编译器级的，而线程是操作系统级的。

协程通常是由编译器来实现的机制。线程看起来也在语言层次，但是内在原理却是操作系统先有这个东西，然后通过一定的API暴露给用户使用，两者在这里有不同。

协程就是用户空间下的线程。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。

线程是抢占式，而协程是非抢占式的，所以需要用户自己释放使用权来切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。

协程并不是取代线程, 而且抽象于线程之上, 线程是被分割的CPU资源, 协程是组织好的代码流程, 协程需要线程来承载运行, 线程是协程的资源, 但协程不会直接使用线程, 协程直接利用的是执行器(Interceptor), 执行器可以关联任意线程或线程池, 可以使当前线程, UI线程, 或新建新程.。

线程是协程的资源。协程通过Interceptor来间接使用线程这个资源。

9.14 协程的好处

与多线程、多进程等并发模型不同，协程依靠user-space调度，而线程、进程则是依靠kernel来进行调度。线程、进程间切换都需要从用户态进入内核态，而协程的切换完全是在用户态完成，且不像线程进行抢占式调度，协程是非抢占式的调度。

通常多个运行在同一调度器中的协程运行在一个线程内，这也消除掉了多线程同步等带来的编程复杂性。同一时刻同一调度器中的协程只有一个会处于运行状态。

我们使用协程，程序只在用户空间内切换上下文，不再陷入内核来做线程切换，这样可以避免大量的用户空间和内核空间之间的数据拷贝，降低了CPU的消耗，从而大大减缓高并发场景时CPU瓶颈的窘境。

另外，使用协程，我们不再需要像异步编程时写那么一堆callback函数，代码结构不再支离破碎，整个代码逻辑上看上去和同步代码没什么区别，简单，易理解，优雅。

我们使用协程，我们可以很简单地实现一个可以随时中断随时恢复的函数。

一些 API 启动长时间运行的操作(例如网络 IO、文件 IO、CPU 或 GPU 密集型任务等)，并要求调用者阻塞直到它们完成。协程提供了一种避免阻塞线程并用更廉价、更可控的操作替代线程阻塞的方法：协程挂起。

协程通过将复杂性放入库来简化异步编程。程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。该库可以将用户代码的相关部分包装为回调、订阅相关事件、在不同线程(甚至不同机器)上调度执行，而代码则保持如同顺序执行一样简单。

9.14.1 阻塞 vs 挂起

协程可以被挂起而无需阻塞线程。而线程阻塞的代价通常是昂贵的，尤其在高负载时，阻塞其中一个会导致一些重要的任务被延迟。

另外，协程挂起几乎是无代价的。不需要上下文切换或者 OS 的任何其他干预。

最重要的是，挂起可以在很大程度上由用户来控制，我们可以决定挂起时做些，并根据需求优化、记日志、拦截处理等。

9.15 协程的内部机制

9.15.1 基本原理

协程完全通过编译技术实现(不需要来自 VM 或 OS 端的支持)，挂起机制是通过状态机来实现，其中的状态对应于挂起调用。

在挂起时，对应的协程状态与局部变量等一起被存储在编译器生成的类的字段中。在恢复该协程时，恢复局部变量并且状态机从挂起点接着后面的状态往后执行。

挂起的协程，是作为Continuation对象来存储和传递，Continuation中持有协程挂起状态与局部变量。

关于协程工作原理的更多细节可以在这个设计文档中找到：https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md。

9.15.2 标准 API

协程有三个主要组成部分：

- 语言支持(即如上所述的挂起功能)，
- Kotlin 标准库中的底层核心 API，
- 可以直接在用户代码中使用的高级 API。
- 底层 API：kotlin.coroutines

底层 API 相对较小，并且除了创建更高级的库之外，不应该使用它。 它由两个主要包组成：

kotlin.coroutines.experimental 带有主要类型与下述原语：

- createCoroutine()
- startCoroutine()
- suspendCoroutine()

kotlin.coroutines.experimental.intrinsics 带有甚至更底层的内在函数如 ：

suspendCoroutineOrReturn

大多数基于协程的应用程序级API都作为单独的库发布：kotlinx.coroutines。这个库主要包括下面几大模块：

- 使用 kotlinx-coroutines-core 的平台无关异步编程
- 基于 JDK 8 中的 CompletableFuture 的 API：kotlinx-coroutines-jdk8
- 基于 JDK 7 及更高版本 API 的非阻塞 IO(NIO)：kotlinx-coroutines-nio
- 支持 Swing (kotlinx-coroutines-swing) 和 JavaFx (kotlinx-coroutines-javafx)
- 支持 RxJava：kotlinx-coroutines-rx

这些库既作为使通用任务易用的便利的 API，也作为如何构建基于协程的库的端到端示例。关于这些 API 用法的更多细节可以参考相关文档。

本章小结

本章我通过大量实例学习了协程的用法；同时了解了作为轻量级线程的协程是怎样简化的我们的多线程并发编程的。我们看到协程通过挂起机制实现非阻塞的特性大大提升了我们并发性能。

最后，我们还简单介绍了协程的实现的原理以及标准API库。Kotlin的协程的实现大量地调用了Java中的多线程API。所以在Kotlin中，我们仍然完全可以使用Java中的多线程编程。

下一章我们来一起学习Kotlin与Java代码之间的互相调用。

本章示例代码工程：https://github.com/EasyKotlin/chapter9_coroutines

第10章 Kotlin与Java互操作

> Kotlin is 100% interoperable with Java™ and Android™

在前面的章节中，我们已经学习了Kotlin的基础语法、类型系统、泛型与集合类、面向对象与函数式编程等主题，在上一章中我们还看到了Kotlin提供的轻量级并发编程模型：协程的相关内容。

从本章开始到在后面的章节中，我们将进入工程代码的实战。我们将在后面分别介绍Kotlin集成SpringBoot开发服务端Web项目、使用Kotlin开发Android项目，以及使用Kotlin来写前端JavaScript代码的等主题。

Kotlin 的竞争优势在于它并不是完全隔离于 Java 语言。它基本上是可与 Java 100% 互操作的。这样，Kotlin就可以站在整个Java生态巨人的肩上，向着更远大的前程前进。

本章我们就让我们一起来学习下Kotlin与Java的互操作。

Kotlin 调用 Java示例

Kotlin 很像 Java。它长得不像 Clojure 或者 Scala 那么奇怪（承认现实把，这两种语言就是挺奇怪的）。所以我们学 Kotlin 应该很快。这门语言显然就是写给 Java 开发者来用的。

Kotlin 在设计之初就考虑了与 Java 的互操作性。我们可以从 Kotlin 中自然地调用现存的 Java 代码。例如，下面是一个Kotlin调用Java中的Okhttp库的代码：

```
package com.easy.kotlin

import okhttp3.*
import java.io.File
import java.io.IOException
import java.util.concurrent.TimeUnit

object OkhttpUtils {
    fun get(url: String): String? {
        var result: String? = ""
        val okhttp = OkHttpClient.Builder()
                .connectTimeout(1, TimeUnit.HOURS)
                .readTimeout(1, TimeUnit.HOURS)
                .writeTimeout(1, TimeUnit.HOURS)
                .build()

        val request = Request.Builder()
                .url(url)
                .build()

        val call = okhttp.newCall(request)
```

```
        try {
            val response = call.execute()
            result = response.body()?.string()
            val f = File("run.log")
            f.appendText(result!!)
            f.appendText("\n")

        } catch (e: IOException) {
            e.printStackTrace()
        }

        return result
    }
}
```

Kotlin调用Java代码跟Groovy一样流畅自如（但是不像Groovy那样 “怎么写都对，但是一运行就报错”， 因为Groovy是一门动态类型语言，而Kotlin则是一门强类型的静态类型语言）。我们基本不需要改变什么就可以直接使用Java中的API库。

并且在 Java 代码中也可以很顺利地调用 Kotlin 代码：

```
package com.easy.kotlin;

import com.alibaba.fastjson.JSON;

public class JSONUtils {
    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }

    public static void main(String[] args) {
        String url = "http://www.baidu.com";
        String result = OkhttpUtils.INSTANCE.get(url);
        System.out.println(result);
    }
}
```

因为Kotlin跟Java本是两门语言，所以在互相调用的时候，会有一些特殊的语法。这里的使用Java调用Kotlin的object对象函数的语法就是 `OkhttpUtils.INSTANCE.get(url)` ，我们看到这里多了个 INSTANCE 。

我们甚至也可以在一个项目中同时使用Kotlin和Java两 种语言混合编程。我们可以在下一章中看到，我们在一个SpringBoot工程中同时使用了Kotlin和Java两种语言进行混合开发。

下面我们来继续介绍 Kotlin 调用 Java 代码的一些细节。

Kotlin使用Java的集合类

Kotlin的集合类API很多就是直接使用的Java的API来实现的。我们在使用的时候，毫无违和感，自然天成：

```
@RunWith(JUnit4::class)
class KotlinUsingJavaTest {
    @Test fun testArrayList() {
        val source = listOf<Int>(1, 2, 3, 4, 5)
        // 使用Java的ArrayList
        val list = ArrayList<Int>()
        for (item in source) {
            list.add(item) // ArrayList.add()
        }
        for (i in 0..source.size - 1) {
            list[i] = source[i] // 调用 get 和 set
        }
    }
}
```

Kotlin调用Java中的Getter 和 Setter

在Java中遵循这样的约定： getter 方法无参数并以 `get` 开头，setter 方法单参数并以 `set` 开头。在 Kotlin 中我们可以直接表示为属性。 例如，我们写一个带setter和getter的Java类：

```
package com.easy.kotlin;

import java.util.Date;

public class Product {
    Long id;
    String name;
    String category;
    Date gmtCreated;
    Date gmtModified;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

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

```
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public Date getGmtCreated() {
        return gmtCreated;
    }

    public void setGmtCreated(Date gmtCreated) {
        this.gmtCreated = gmtCreated;
    }

    public Date getGmtModified() {
        return gmtModified;
    }

    public void setGmtModified(Date gmtModified) {
        this.gmtModified = gmtModified;
    }
}
```

然后，我们在Kotlin可以直接使用属性名字进行get和set操作：

```
@RunWith(JUnit4::class)
class ProductTest {
    @Test fun testGetterSetter() {
        val product = Product()
        product.name = "账务系统"
        product.category = "金融财务类"
        product.gmtCreated = Date()
        product.gmtModified = Date()
        println(JSONUtils.toJsonString(product))
        Assert.assertTrue(product.getName() == "账务系统")
        Assert.assertTrue(product.name == "账务系统")
        Assert.assertTrue(product.getCategory() == "金融财务类")
        Assert.assertTrue(product.category == "金融财务类")
    }
}
```

当然，我们还可以像在Java中一样，直接调用像product.getName()、product.setName("Kotlin")这样的getter、setter方法。

调用Java中返回 void 的方法

如果一个 Java 方法返回 void，那么从 Kotlin 调用时中返回 `Unit` 。

```
public class Admin {
    String name;

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Admin{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

我们这样调用

```
val setReturn = admin.setName("root")
println(setReturn)
```

将输出：kotlin.Unit

空安全和平台类型

我们知道Java 中的任何引用都可能是null，这样我们在使用 Kotlin调用来自 Java 的对象的时候就有可能会出现空安全的问题。

Java 声明的类型在 Kotlin 中会被特别对待并称为平台类型（platform types ）。对这种类型的空检查会放宽，因此它们的安全保证与在 Java 中相同。

请看以下示例：

```
@RunWith(JUnit4::class)
class CallingJavaNullSafe {
    @Test fun testCallingJavaNullSafe() {
        val product = Product()
        // product.name = null
```

```
        product.category = "金融财务类"
        product.gmtCreated = Date()
        product.gmtModified = Date()
        println(JSONUtils.toJsonString(product))

        val name = product.name
        println("product name is ${name}")

        val eqName = name == "账务系统"
        println(eqName)

        name.substring(1)
    }
}
```

上面的代码可以正确编译通过。Kotlin编译器对来自Java的空值name（平台类型）放宽了空检查 `name.substring(1)` 。但是这样的空指针异常仍然会在运行时抛出来。

运行上面的代码，我们可以看到输出：

```
{"category":"金融财务类","gmtCreated":1500050426817,"gmtModified":1500050426817}
product name is null
false

null cannot be cast to non-null type java.lang.String
kotlin.TypeCastException: null cannot be cast to non-null type java.lang.String
    at com.easy.kotlin.CallingJavaNullSafe.testCallingJavaNullSafe(CallingJavaNullSafe.
kt:27)
```

我们没有设置name的值，在Java它就是null。我们在Kotlin代码中使用了这个name进行计算，我们可以看出：

```
        val eqName = name == "账务系统"
        println(eqName)
```

可以正确输出false。这表明Kotlin的判断字符串是否相等已经对null的情况作了判断处理，这样的代码如果在Java中调用 `name.equals("账务系统")` 就该抛空指针异常了。

但是当我们直接使用name这个值来调用 `name.substring(1)` 的时候，Kotlin编译器不会检查这个空异常，但是运行时还是要报错的： `null cannot be cast to non-null type java.lang.String` 。

如果我们不想看到这样的异常，而是当name是null的时候，安静的输出null，直接使用Kotlin中的空安全的调用 `.?` ：

```
name?.substring(1)
```

这样，运行的时候不会抛出异常，直接安静的返回null。

平台类型

平台类型不能在程序中显式表述，因此在语言中没有相应语法。 然而，编译器和 IDE 有时需要（在错误信息中、参数信息中等）显示他们，所以我们用一个助记符来表示他们：

- `T!` ：表示 `T` 或者 `T?`
- `(Mutable) Collection<T>!` ：表示 “可以可变或不可变、可空或不可空的 T 的 Java 集合”
- `Array<(out) T>!` ：表示 “可空或者不可空的 T（或 T 的子类型）的 Java 数组”

Kotlin与Java中的类型映射

Kotlin 特殊处理一部分 Java 类型。这样的类型不是 “按原样” 从 Java 加载，而是 映射 到相应的 Kotlin 类型。

映射只发生在编译期间，运行时表示保持不变。

Java 的原生类型映射到相应的 Kotlin 类型：

Java 类型	Kotlin 类型
`byte`	`kotlin.Byte`
`short`	`kotlin.Short`
`int`	`kotlin.Int`
`long`	`kotlin.Long`
`char`	`kotlin.Char`
`float`	`kotlin.Float`
`double`	`kotlin.Double`
`boolean`	`kotlin.Boolean`

Java中的一些内置类型也会作相应的映射：

Java 类型	Kotlin 类型
`java.lang.Object`	`kotlin.Any!`
`java.lang.Cloneable`	`kotlin.Cloneable!`
`java.lang.Comparable`	`kotlin.Comparable!`
`java.lang.Enum`	`kotlin.Enum!`
`java.lang.Annotation`	`kotlin.Annotation!`
`java.lang.Deprecated`	`kotlin.Deprecated!`
`java.lang.CharSequence`	`kotlin.CharSequence!`

java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java 的装箱原始类型映射到对应的可空Kotlin 类型：

Java 类型	Kotlin 类型
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

另外，用作类型参数的Java类型映射到Kotlin中的平台类型：例如，List<java.lang.Integer> 在 Kotlin 中会成为 List<Int!> 。

集合类型在 Kotlin 中可以是只读的或可变的，因此 Java 集合类型作如下映射：（下表中的所有 Kotlin 类型都在 kotlin.collections 包中）：

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加载的
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Ite
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Ite
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Col
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set
List<T>	List<T>	MutableList<T>	(Mutable)Lis
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)Lis
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map (Mutable)Entr

Java 的数组映射：

Java 类型	Kotlin 类型
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

Kotlin 中使用 Java 的泛型

Kotlin 的泛型与 Java 有点不同。当将 Java 类型导入 Kotlin 时，我们会执行一些转换：

Kotlin 的泛型	Java 的泛型	说明
`Foo<out Bar!>!`	`Foo<? extends Bar>`	Java 的通配符转换成类型投影
`Foo<? super Bar>`	`Foo<in Bar!>!`	同上
`List<*>!`	`List`	Java的原始类型转换成星投影

和 Java 一样，Kotlin 在运行时不保留泛型，即对象不携带传递到他们构造器中的那些类型参数的实际类型。

即 `ArrayList<Integer>()` 和 `ArrayList<Character>()` 是不能区分的。

Kotlin与Java 中的数组

与 Java 不同，Kotlin 中的数组是非型变的，即 Kotlin 不允许我们把一个 `Array<String>` 赋值给一个 `Array<Any>` 。

Java 平台上，持有原生数据类型的数组避免了装箱/拆箱操作的开销。 在Kotlin中，对于每种原生类型的数组都有一个特化的类（ `IntArray` 、 `DoubleArray` 、 `CharArray` 等）来实现同样的功能。它们与 `Array` 类无关，并且会编译成 Java 原生类型数组以获得最佳性能。

Java 可变参数

Java 类有时声明一个具有可变数量参数（varargs）的方法来使用索引。

```
public class VarArgsDemo<T> {
    static VarArgsDemo vad = new VarArgsDemo();

    public static void main(String... agrs) {
        System.out.println(vad.append("a", "b", "c"));
        System.out.println(vad.append(1, 2, 3));
        System.out.println(vad.append(1, 2, "3"));
    }

    public String append(T... element) {
        StringBuilder result = new StringBuilder();
        for (T e : element) {
            result.append(e);
        }
        return result.toString();
    }
}
```

在Kotlin中，我们使用展开运算符 `*` 来传递这个varargs：

```kotlin
@RunWith(JUnit4::class)
class VarArgsDemoTest {
    @Test fun testVarArgsDemo() {
        val varArgsDemo = VarArgsDemo<Any?>()
        val array = arrayOf(0, 1, 2, 3)
        val result = varArgsDemo.append(*array)
        println(result)
    }
}
```

运行输出：0123

非受检异常

在 Kotlin 中，所有异常都是非受检的（Non-Checked Exceptions），这意味着编译器不会强迫你捕获其中的任何一个。而在Java中会要求我们捕获异常，例如下面的代码：

```java
public class JSONUtils {

    static JSONUtils jsonUtils = new JSONUtils();

    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }

    public static void main(String[] args) {
        String url = "http://www.baidu.com";
        String result = OkhttpUtils.INSTANCE.get(url);
        System.out.println(result);

        jsonUtils.parseObject( jsonText: "{}");
    }

    public Object parseObject(String jsonText) throws Exception {
        throw new Exception("Test Exception");
    }
}
```

Unhandled exception: java.lang.Exception

也就是说，我们需要写类似下面的 `try catch` 代码块：

```java
        try {
            jsonUtils.parseObject("{}");
        } catch (Exception e) {
            e.printStackTrace();
        }
```

然而在Kotlin中情况就不是这样子了：当我们调用一个声明受检异常的 Java 方法时，Kotlin 不会强迫你做任何事情：

```
    @Test fun testNonCheckedExceptions() {
        val jsonUtils = JSONUtils()
        jsonUtils.parseObject("{}")
    }
```

但是，我们在运行的时候，还是会抛异常：

```
com.easy.kotlin.CallingJavaNullSafe > testNonCheckedExceptions FAILED
    java.lang.Exception at CallingJavaNullSafe.kt:34
```

Kotlin的不受检异常，这样也会导致运行时抛出异常。关于异常的处理，该处理的终归还是要处理的。

对象方法

Java中的java.lang.Object定义如下：

```
public class Object {
    private static native void registerNatives();
    static {
        registerNatives();
    }
    public final native Class<?> getClass();
    public native int hashCode();
    public boolean equals(Object obj) {
        return (this == obj);
    }
    protected native Object clone() throws CloneNotSupportedException;
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public final void wait(long timeout, int nanos) throws InterruptedException {...}
    public final void wait() throws InterruptedException {
        wait(0);
    }
    protected void finalize() throws Throwable { }
```

当 Java 类型导入到 Kotlin 中时，类型 `java.lang.Object` 的所有引用都成了 `Any` 。 `Any` 只声明了 `toString()` 、 `hashCode()` 和 `equals()` 函数。怎样才能用到 `java.lang.Object` 的其他成员方法呢？下面我们来看下。

wait()/notify()

《Effective Java》 第 69 条中建议优先使用并发工具（concurrency utilities）而不是 `wait()` 和 `notify()` 。因此，类型 `Any` 的引用没有提供这两个方法。

如果我们真的需要调用它们的话，可以将其转换为 `java.lang.Object` 来使用：

```
(foo as java.lang.Object).wait()
```

getClass()

要取得对象的 Java 类，我们可以在类引用上使用 `java` 扩展属性，它是Kotlin的反射类kotlin.reflect.KClass的扩展属性。

```
val fooClass = foo::class.java
```

上面的代码使用了自 Kotlin 1.1 起支持的绑定类引用。我们也可以使用 `javaClass` 扩展属性。

```
val fooClass = foo.javaClass
```

clone()

要覆盖 `clone()` ，需要继承 `kotlin.Cloneable` ：

```

class Example : Cloneable {
    override fun clone(): Any { …… }
}
```

要谨慎地改写clone方法。

finalize()

要覆盖 `finalize()` ，我们只需要声明它即可，不用再写 override关键字：

```
class C {
    protected fun finalize() {
        // 终止化逻辑
    }
```

```
}
```

访问静态成员

Java 类的静态成员会形成该类的“伴生对象”。我们可以直接显式访问其成员。例如：

一个带静态方法的Java类

```
public class JSONUtils {
    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }
}
```

我们在Kotlin代码可以直接这样调用:

```
@RunWith(JUnit4::class)
class JSONUtilsTest {
    @Test fun testJSONUtils() {
        val userService = UserServiceImpl()
        val user = userService.findByName("admin")
        Assert.assertTrue(user.name == "admin")

        val userJson = JSONUtils.toJsonString(user)
        println(userJson)
        Assert.assertTrue(userJson == "{\"name\":\"admin\",\"password\":\"admin\"}")
    }
}
```

上面我们提到过，如果是反过来调用，Java调用Kotlin中的object对象类中的函数，需要使用object的 `对象名.INSTANCE` 来调用函数。

Kotlin与Java 的反射

我们可以使用 `instance::class.java` 、 `ClassName::class.java` 或者 `instance.javaClass` 通过 `java.lang.Class` 来进入 Java 的反射类 `java.lang.Class` ， 之后我们就可以使用Java中的反射的功能特性了。

代码示例：

```
@RunWith(JUnit4::class)
class RefectClassTest {
    @Test fun testGetterSetter() {
        val product = Product()
        val pClz = product::class.java
```

```
        println(pClz.canonicalName)
        pClz.declaredFields.forEach { println(it) }
        pClz.declaredMethods.forEach {
            println(it.name);
            it.parameters.forEach { println(it) }
        }
    }
}
```

运行上面的代码输出：

```
com.easy.kotlin.Product
java.lang.Long com.easy.kotlin.Product.id
java.lang.String com.easy.kotlin.Product.name
java.lang.String com.easy.kotlin.Product.category
java.util.Date com.easy.kotlin.Product.gmtCreated
java.util.Date com.easy.kotlin.Product.gmtModified
getName
setName
java.lang.String arg0
getId
setId
java.lang.Long arg0
setCategory
java.lang.String arg0
getGmtCreated
setGmtCreated
java.util.Date arg0
getGmtModified
setGmtModified
java.util.Date arg0
getCategory
```

SAM 转换

我们在Kotlin中，要某个函数做某件事时，会传一个函数参数给它。 而在Java中，并不支持传送函数参数。通常Java的实现方式是将动作放在一个实现某接口的类中，然后将该类的一个实例传递给另一个方法。在大多数情况下，这些接口只有单个抽象方法（single abstract method），在Java中被称为SAM类型。

例如：Runnable接口：

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

在 Java 8中我们也通常称之为函数式接口。

Kotlin 支持 SAM 转换。Kotlin 的函数字面值可以被自动的转换成只有一个非默认方法的 Java 接口的实现，只要这个方法的参数类型能够与这个 Kotlin 函数的参数类型相匹配。

我们可以这样创建 SAM 接口的实例：

```
val runnable = Runnable { println("执行测试") } // Kotlin 调用Java的SAM接口方法
```

测试代码：

```
@RunWith(JUnit4::class)
class SAMFunctionalInterfaceTest {
    @Test fun testSAMFunctionalInterface() {
        val runnable = Runnable { println("执行测试") }
        val thread = Thread(runnable)
        thread.start()
    }
}
```

要注意的是，SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类也只有一个抽象方法。

还要注意，此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换为 Kotlin 接口的实现。

Java使用了Kotlin的关键字

一些 Kotlin 关键字在 Java 中是有效标识符：in、 object、 is等等。

如果一个 Java 库使用了 Kotlin 关键字作为方法，我们可以通过反引号（`）字符转义它来调用该方法。例如我们有个Java类，其中有个is方法：

```
public class MathTools {

    public boolean is(Object o) {
        return true;
    }

}
```

那么我们在Kotlin代码这样调用这个is方法：

```
@RunWith(JUnit4::class)
```

```
class MathToolsTest {
    @Test fun testISKeyWord(){
        val b = MathTools().`is`(1)
    }
}
```

Java 调用 Kotlin

Java 同样也可以调用 Kotlin 代码。但是要多用一些注解语法。

Java访问Kotlin属性

Kotlin 属性会编译成以下 Java 元素：

- 一个 getter 方法，名称通过加前缀 `get` 算出；
- 一个 setter 方法，名称通过加前缀 `set` 算出（只适用于 `var` 属性）；
- 一个与属性名称相同的私有字段。

例如，下面的Kotlin类：

```
class Department {
    var id: Long = -1L
    var name: String = "Dept"
}
```

会被编译成对应的 Java 代码：

```
public final class Department {
   private long id = -1L;
   @NotNull
   private String name = "Dept";

   public final long getId() {
      return this.id;
   }

   public final void setId(long var1) {
      this.id = var1;
   }

   @NotNull
   public final String getName() {
      return this.name;
   }
```

```
    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }
}
```

我们可以看出，在Kotlin中的Long类型被编译成Java中的原生的long了。 我们在Java代码这样调用：

```
@RunWith(JUnit4.class)
public class JavaCallingKotlinCodeTest {
    @Test
    public void testProperty() {
        Department d = new Department();
        d.setId(1);
        d.setName("技术部");

        Assert.assertTrue(1 == d.getId());
        Assert.assertTrue("技术部".equals(d.getName()));

    }
}
```

另外，如果Kotlin的属性名以 `is` 开头，则使用不同的名称映射规则：

- getter 的名称直接使用属性名称
- setter 的名称是通过将 `is` 替换为 `set` 获得。

例如，对于属性 `isOpen` ，其 getter 会称做 `isOpen()` ，而其 setter 会称做 `setOpen()` 。

这一规则适用于任何类型的属性，并不仅限于 `Boolean` 。

代码示例：

Kotlin代码

```
class Department {
    var id: Long = -1L
    var name: String = "Dept"
    var isOpen:Boolean = true
    var isBig:String = "Y"
}
```

Java调用Kotlin的测试代码：

```
@Test
public void testProperty() {
```

```
    Department d = new Department();
    d.setId(1);
    d.setName("技术部");
    d.setBig("Y");
    d.setOpen(true);

    Assert.assertTrue(1 == d.getId());
    Assert.assertTrue("技术部".equals(d.getName()));
    Assert.assertTrue("Y".equals(d.isBig()));
    Assert.assertTrue(d.isOpen());

}
```

Java调用Kotlin的包级函数

在 `package com.easy.kotlin` 包内的 `KotlinExample.kt` 源文件中声明的所有的函数和属性，包括扩展函数，都将编译成一个名为 `com.easy.kotlin.KotlinExampleKt` 的 Java 类中的静态方法。

代码示例：

Kotlin的包级属性、函数代码：

```
package com.easy.kotlin

fun f1() {
    println("I am f1")
}

fun f2() {
    println("I am f2")
}

val p: String = "PPP"

fun String.swap(index1: Int, index2: Int): String {
    val strArray = this.toCharArray()
    val tmp = strArray[index1]
    strArray[index1] = strArray[index2]
    strArray[index2] = tmp

    var result = ""
    strArray.forEach { result += it }
    return result
}

fun main(args: Array<String>) {
    println("abc".swap(0, 2))
```

```
}
```

编译成对应的Java的代码：

```
public final class KotlinExampleKt {
   @NotNull
   private static final String p = "PPP";

   public static final void f1() {
      String var0 = "I am f1";
      System.out.println(var0);
   }

   public static final void f2() {
      String var0 = "I am f2";
      System.out.println(var0);
   }

   @NotNull
   public static final String getP() {
      return p;
   }

   @NotNull
   public static final String swap(@NotNull String $receiver, int index1, int index2) {
      Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
      char[] var10000 = $receiver.toCharArray();
      Intrinsics.checkExpressionValueIsNotNull(var10000, "(this as java.lang.String).to
CharArray()");
      char[] strArray = var10000;
      char tmp = strArray[index1];
      strArray[index1] = strArray[index2];
      strArray[index2] = tmp;
      Object result = "";
      char[] $receiver$iv = strArray;

      for(int var7 = 0; var7 < $receiver$iv.length; ++var7) {
         char element$iv = $receiver$iv[var7];
         result = result + element$iv;
      }

      return result;
   }

   public static final void main(@NotNull String[] args) {
      Intrinsics.checkParameterIsNotNull(args, "args");
      String var1 = swap("abc", 0, 2);
      System.out.println(var1);
```

```
    }
}
```

我们可以看到，Kotlin中的扩展函数

```
fun String.swap(index1: Int, index2: Int): String
```

被编译成

```
public static final String swap(@NotNull String $receiver, int index1, int index2)
```

Kotlin中的 `String.` 接收者被当做Java方法中的第一个参数传入。

Java调用Kotlin包级属性、函数的测试代码：

```
@Test
public void testPackageFun() {
    KotlinExampleKt.f1();
    KotlinExampleKt.f2();
    System.out.println(KotlinExampleKt.getP());
    KotlinExampleKt.swap("abc",0,1);
}
```

运行输出：

```
I am f1
I am f2
PPP
bac
```

另外，要注意的这里生成的类KotlinExampleKt，我们不能使用new来创建实例对象：

```
KotlinExampleKt example = new KotlinExampleKt();// 报错
```

报如下错误：

```
error: cannot find symbol
        KotlinExampleKt example = new KotlinExampleKt();
                                  ^
  symbol:   constructor KotlinExampleKt()
  location: class KotlinExampleKt
1 error
```

在编程中，我们推荐使用Kotlin默认的命名生成规则。如果确实有特殊场景需要自定义Kotlin包级函数对应的生成Java类的名字，我们可以使用 `@JvmName` 注解修改生成的 Java 类的类名：

```
@file:JvmName("MyKotlinExample")

package com.easy.kotlin


fun f3() {
    println("I am f3")
}

fun f4() {
    println("I am f4")
}

val p2: String = "PPP"
```

测试代码：

```
MyKotlinExample.f3();
MyKotlinExample.f4();
```

实例字段

我们使用 `@JvmField` 注解对Kotlin中的属性字段标注，表示这是一个实例字段（Instance Fields），Kotlin编译器在处理的时候，将不会给这个字段生成getters/setters方法。

```
class Department {
    var id: Long = -1L
    var name: String = "Dept"
    var isOpen: Boolean = true
    var isBig: String = "Y"

    @JvmField var NO = 0
}
```

映射成Java的代码就是：

```
public final class Department {
   private long id = -1L;
   @NotNull
   private String name = "Dept";
   private boolean isOpen = true;
   @NotNull
```

```
    private String isBig = "Y";
    @JvmField
    public int NO;

    public final long getId() {
        return this.id;
    }

    public final void setId(long var1) {
        this.id = var1;
    }

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }

    public final boolean isOpen() {
        return this.isOpen;
    }

    public final void setOpen(boolean var1) {
        this.isOpen = var1;
    }

    @NotNull
    public final String isBig() {
        return this.isBig;
    }

    public final void setBig(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.isBig = var1;
    }
}
```

我们在Java中调用的时候，就直接使用这个属性实例字段 `NO` ：

```
System.out.println(d.NO = 10);
```

静态字段

Kotlin中在命名对象或伴生对象中声明的 属性:

```
class Department {
    ...
    companion object {
        var innerID = "X001"
        @JvmField
        var innerName = "DEP"
    }
}
```

innerID、innerName这两个字段的区别在于可见性上：

```
@NotNull
private static String innerID = "X001";
@JvmField
@NotNull
public static String innerName = "DEP";
```

这个私有的innerID通过Companion对象来封装，提供出public的getInnerID()、setInnerID来访问：

```
public static final class Companion {
   @NotNull
   public final String getInnerID() {
      return Department.innerID;
   }

   public final void setInnerID(@NotNull String var1) {
      Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
      Department.innerID = var1;
   }

   private Companion() {
   }

   // $FF: synthetic method
   public Companion(DefaultConstructorMarker $constructor_marker) {
      this();
   }
}
```

我们在Java访问的 `innerID` 时候，是通过Companion来访问：

```
Department.Companion.getInnerID()
```

而我们使用 `@JvmField` 注解的字段 `innerName` ，Kotlin编译器会把它的访问权限设置是public的，这样我们就可以这样访问这个属性字段了：

```
Department.innerName
```

静态方法

Kotlin 中，我还可以将命名对象或伴生对象中定义的函数标注为 `@JvmStatic` ，这样编译器既会在相应对象的类中生成静态方法，也会在对象自身中生成实例方法。

跟静态属性类似的，我们看下面的代码示例：

```
class Department {
    ...
    companion object {
        var innerID = "X001"
        @JvmField
        var innerName = "DEP"

        fun getObjectName() = "ONAME"
        @JvmStatic
        fun getObjectID() = "OID"
    }
}
```

编译器编译之后，反编译成的对应的Java代码：

```
public final class Department {
   ...
   @JvmStatic
   @NotNull
   public static final String getObjectID() {
      return Companion.getObjectID();
   }

   public static final class Companion {
      ...
      @NotNull
      public final String getObjectName() {
         return "ONAME";
      }

      @JvmStatic
      @NotNull
      public final String getObjectID() {
         return "OID";
```

```
        }
       ...
    }
}
```

在Java中调用的代码如下：

```
Department.Companion.getObjectID(); // OK
Department.Companion.getObjectName(); // OK, 唯一的工作方式
Department.getObjectID(); // ALSO OK
Department.getObjectName(); // ERROR
```

这些注解语法是编译器为了更加方便Java调用Kotlin代码提供的一些简便技巧。这样可使得Java中调用Kotlin代码更加自然优雅些。

可见性

Kotlin 的可见性与Java的可见性的映射关系如下表所示：

Kotlin中的声明	Java中的声明
private	private
protected	protected
internal	public
public	public

例如下面的Kotlin代码：

```
class ProgrammingBook {
    private var isbn: String = "978-7-111-44250-9"
    protected var author: String = "Cay"
    public var name: String = "Core Java"
    internal var pages: Int = 300

    private fun findISBN(): String = "978-7-111-44250-9"
    protected fun findAuthor(): String = "Cay"
    public fun findName(): String = "Core Java"
    internal fun findPages(): Int = 300
}
```

对应的Java的代码是：

```
public final class ProgrammingBook {
   private String isbn = "978-7-111-44250-9";
```

```
    @NotNull
    private String author = "Cay";
    @NotNull
    private String name = "Core Java";
    private int pages = 300;

    @NotNull
    protected final String getAuthor() {
        return this.author;
    }

    protected final void setAuthor(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.author = var1;
    }

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }

    public final int getPages$production_sources_for_module_chapter10_interoperability_m
ain() {
        return this.pages;
    }

    public final void setPages$production_sources_for_module_chapter10_interoperability_
main(int var1) {
        this.pages = var1;
    }

    private final String findISBN() {
        return "978-7-111-44250-9";
    }

    @NotNull
    protected final String findAuthor() {
        return "Cay";
    }

    @NotNull
    public final String findName() {
        return "Core Java";
    }
```

```
    public final int findPages$production_sources_for_module_chapter10_interoperability_
main() {
        return 300;
    }
}
```

我们可以看到Kotlin中的可见性跟Java中的基本相同。

生成默认参数值函数的重载

我们在Kotlin中写一个有默认参数值的 Kotlin 方法，它会对每一个有默认值的参数都生成一个重载函数。这样的Kotlin函数，在 Java 中调用的话，只会有一个所有参数都存在的完整参数签名方法可见。如果我们希望Java像Kotlin中一样可以调用多个重载，可以使用 `@JvmOverloads` 注解。

下面我们来通过一个实例对比两者的区别：

这是一段Kotlin代码：

```
class OverridesFunWithDefaultParams {
    fun f1(a: Int = 0, b: String = "B") {

    }

    @JvmOverloads fun f2(a: Int = 0, b: String = "B") {

    }
}
```

函数f1 和 f2 都带有默认参数。测试代码如下：

```
    @Test
    public void testOverridesFunWithDefaultParams() {
        OverridesFunWithDefaultParams ofdp = new OverridesFunWithDefaultParams();
        ofdp.f1(1, "a");
        ofdp.f2();
        ofdp.f2(2);
        ofdp.f2(2, "b");
    }
```

这就是 `@JvmOverloads` 注解的作用，编译器会处理这个注解所标注的函数，并为之生成额外的重载函数给Java调用。

检查Kotlin中异常

如上所述，Kotlin 没有受检异常。即像下面像这样的 Kotlin 函数：

```
class CheckKotlinException {
    fun thisIsAFunWithException() {
        throw Exception("I am an exception in kotlin")
    }
}
```

在Java中调用，编译器是不会检查这个异常的：

```
    @Test
    public void testCheckKotlinException() {
        CheckKotlinException cke = new CheckKotlinException();
        cke.thisIsAFunWithException();// Java编译器不检查这个Kotlin中的异常
    }
```

当然，在运行时，这个异常还是会抛出来。然后，如果我们想要在 Java 中调用它并捕捉这个异常，我们可以给Kotlin中的函数加上注解 `@Throws(Exception::class)`，就像下面这样：

```
@Throws(Exception::class)
fun thisIsAnotherFunWithException() {
    throw Exception("I am Another exception in kotlin")
}
```

然后，我们在Java中调用的时候，Java编译器就会检查这个异常：

```
@RunWith(JUnit4.class)
public class CheckKotlinExceptionTest {
    @Test
    public void testCheckKotlinException() {
        CheckKotlinException cke = new CheckKotlinException();
        cke.thisIsAFunWithException();// Java编译器不检查这个Kotlin中的异常

        // Kotlin中显示声明了异常，Java编译器会检查这个异常
        cke.thisIsAnotherFunWithException();
                                      Unhandled exception: java.lang.Exception
        try {
            cke.thisIsAnotherFunWithException();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

最后，我们的代码就需要捕获该异常并处理它。

完整的示例代码如下：

```
package com.easy.kotlin
```

```
class CheckKotlinException {
    fun thisIsAFunWithException() {
        throw Exception("I am an exception in kotlin")
    }

    @Throws(Exception::class)
    fun thisIsAnotherFunWithException() {
        throw Exception("I am Another exception in kotlin")
    }
}
```

测试代码：

```
package com.easy.kotlin;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)
public class CheckKotlinExceptionTest {
    @Test
    public void testCheckKotlinException() {
        CheckKotlinException cke = new CheckKotlinException();
        cke.thisIsAFunWithException();// Java编译器不检查这个Kotlin中的异常

        // Kotlin中显示声明了异常，Java编译器会检查这个异常
        // cke.thisIsAnotherFunWithException();
        try {
            cke.thisIsAnotherFunWithException();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Nothing 类型

在Kotlin中 `Nothing` 类型是一个特殊的类型，它在 Java 中没有的对应的类型。在使用 `Nothing` 参数的地方会生成一个原始类型。

例如下面的Kotlin代码：

```
fun emptyList(): List<Nothing> = listOf()
```

对应到Java代码中是：

```
@NotNull
public final List emptyList() {
   return CollectionsKt.emptyList();
}
```

Kotlin中的 `List<Nothing>` 映射为原生类型 `List` 。

Kotlin与Java对比

在前面的内容里，我们已经看到了Java与Kotlin的互操作的基本方式。为了更好的认识Java与Kotlin这两门语言，我们在这里给出一些基本功能，同时使用Java与Kotlin来实现的代码实例。通过横向对比，从中我们可以看出它们的异同。

（此处可整理成表格形式）

打印日志

- Java

```
System.out.print("Java");
System.out.println("Java");
```

- Kotlin

```
print("Kotlin")
println("Kotlin")
```

其实，Kotlin中的println函数是一个内联函数，它其实就是通过封装 `java.lang.System` 类的System.out.println来实现的。

```
@kotlin.internal.InlineOnly
public inline fun print(message: Any?) {
    System.out.print(message)
}
```

常量与变量

- Java

```
String name = "KotlinVSJava";
final String name = "KotlinVSJava";
```

- Kotlin

```
var name = "KotlinVSJava"
val name = "KotlinVSJava"
```

null声明

- Java

```
String otherName;
otherName = null;
```

- Kotlin

```
var otherName : String?
otherName = null
```

空判断

- Java

```
if (text != null) {
    int length = text.length();
}
```

- Kotlin

```
text?.let {
    val length = text.length
}
// 或者
val length = text?.length
```

在Kotlin中，我们只使用一个问号安全调用符号就省去了Java中烦人的 `if - null` 判断。

字符串拼接

- Java

```
String firstName = "Jack";
String lastName = "Chen";
String message = "My name is: " + firstName + " " + lastName;
```

- Kotlin

```
val firstName = "Jack"
val lastName = "Chen"
val message = "My name is: $firstName $lastName"
```

Kotlin中使用 `$` 和 `${}` （花括号里面是表达式的时候）占位符来实现字符串的拼接，这个比在Java中每次使用加号来拼接要方便许多。

换行

- Java

```
String text = "First Line\n" +
              "Second Line\n" +
              "Third Line";
```

- Kotlin

```
val text = """
        |First Line
        |Second Line
        |Third Line
        """.trimMargin()
```

三元表达式

- Java

```
String text = x > 5 ? "x > 5" : "x <= 5";
```

- Kotlin

```
val text = if (x > 5)
              "x > 5"
           else "x <= 5"
```

操作符

- java

```java
final int andResult  = a & b;
final int orResult   = a | b;
final int xorResult  = a ^ b;
final int rightShift = a >> 2;
final int leftShift  = a << 2;
```

- Kotlin

```kotlin
val andResult  = a and b
val orResult   = a or b
val xorResult  = a xor b
val rightShift = a shr 2
val leftShift  = a shl 2
```

类型判断和转换（显式）

- Java

```java
if (object instanceof Car) {
}
Car car = (Car) object;
```

- Kotlin

```kotlin
if (object is Car) {
}
var car = object as Car
```

类型判断和转换 (隐式)

- Java

```java
if (object instanceof Car) {
   Car car = (Car) object;
}
```

- Kotlin

```kotlin
if (object is Car) {
   var car = object // Kotlin智能转换
}
```

Kotlin的类型系统具备一定的类型推断能力，这样也省去了不少在Java中类型转换的样板式代码。

Range区间

- Java

```
if (score >= 0 && score <= 300) { }
```

- Kotlin

```
if (score in 0..300) { }
```

更灵活的case语句

- Java

```
public String getGrade(int score) {
    String grade;
    switch (score) {
        case 10:
        case 9:
            grade = "A";
            break;
        case 8:
        case 7:
        case 6:
            grade = "B";
            break;
        case 5:
        case 4:
            grade = "C";
            break;
        case 3:
        case 2:
        case 1:
            grade = "D";
            break;
        default:
            grade = "E";
    }
    return grade;
}
```

- Kotlin

```kotlin
fun getGrade(score: Int): String {
    var grade = when (score) {
        9, 10 -> "A"
        in 6..8 -> "B"
        4, 5 -> "C"
        in 1..3 -> "D"
        else -> "E"
    }
    return grade
}
```

for循环

- Java

```java
for (int i = 1; i <= 10 ; i++) { }

for (int i = 1; i < 10 ; i++) { }

for (int i = 10; i >= 0 ; i--) { }

for (int i = 1; i <= 10 ; i+=2) { }

for (int i = 10; i >= 0 ; i-=2) { }

for (String item : collection) { }

for (Map.Entry<String, String> entry: map.entrySet()) { }
```

- Kotlin

```kotlin
for (i in 1..10) { }

for (i in 1 until 10) { }

for (i in 10 downTo 0) { }

for (i in 1..10 step 2) { }

for (i in 10 downTo 1 step 2) { }

for (item in collection) { }

for ((key, value) in map) { }
```

更方便的集合操作

- Java

```
final List<Integer> listOfNumber = Arrays.asList(1, 2, 3, 4);
final Map<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "Jack");
map.put(2, "Ali");
map.put(3, "Mindorks");
```

- Kotlin

```
val listOfNumber = listOf(1, 2, 3, 4)
val map = mapOf(1 to "Jack", 2 to "Ali", 3 to "Mindorks")
```

遍历

- Java

```
// Java 7
for (Car car : cars) {
  System.out.println(car.speed);
}

// Java 8+
cars.forEach(car -> System.out.println(car.speed));

// Java 7
for (Car car : cars) {
  if (car.speed > 100) {
    System.out.println(car.speed);
  }
}

// Java 8+
cars.stream().filter(car -> car.speed > 100).forEach(car -> System.out.println(car.spee
d));
```

- Kotlin

```
cars.forEach {
    println(it.speed)
}

cars.filter { it.speed > 100 }
```

```
        .forEach { println(it.speed)}
```

方法(函数)定义

- Java

```
void doSomething() {
    // 实现
}

void doSomething(int... numbers) {
    // 实现
}
```

- Kotlin

```
fun doSomething() {
    // 实现
}

fun doSomething(vararg numbers: Int) {
    // 实现
}
```

带返回值的方法（函数）

- Java

```
int getScore() {
    // logic here
    return score;
}
```

- Kotlin

```
fun getScore(): Int {
    // logic here
    return score
}

// 单表达式函数
fun getScore(): Int = score
```

另外，Kotlin中的函数是可以直接传入函数参数，同时可以返回一个函数类型的。

constructor 构造器

- Java

```
public class Utils {

    private Utils() {
      // 外部无法来调用实例化
    }

    public static int getScore(int value) {
        return 2 * value;
    }

}
```

- Kotlin

```
class Utils private constructor() {

    companion object {

        fun getScore(value: Int): Int {
            return 2 * value
        }

    }
}

// 或者直接声明一个object对象
object Utils {

    fun getScore(value: Int): Int {
        return 2 * value
    }

}
```

JavaBean与Kotlin数据类

这段Kotlin中的数据类的代码：

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

对应下面这段Java实体类的代码：

- Java

```
public final class Developer {
   @NotNull
   private final String name;
   private final int age;

   @NotNull
   public final String getName() {
      return this.name;
   }

   public final int getAge() {
      return this.age;
   }

   public Developer(@NotNull String name, int age) {
      Intrinsics.checkParameterIsNotNull(name, "name");
      super();
      this.name = name;
      this.age = age;
   }

   @NotNull
   public final String component1() {
      return this.name;
   }

   public final int component2() {
      return this.age;
   }

   @NotNull
   public final Developer copy(@NotNull String name, int age) {
      Intrinsics.checkParameterIsNotNull(name, "name");
      return new Developer(name, age);
   }

   // $FF: synthetic method
   // $FF: bridge method
   @NotNull
   public static Developer copy$default(Developer var0, String var1, int var2, int var3
, Object var4) {
      if((var3 & 1) != 0) {
         var1 = var0.name;
      }

      if((var3 & 2) != 0) {
```

```
            var2 = var0.age;
        }

        return var0.copy(var1, var2);
    }

    public String toString() {
        return "Developer(name=" + this.name + ", age=" + this.age + ")";
    }

    public int hashCode() {
        return (this.name != null?this.name.hashCode():0) * 31 + this.age;
    }

    public boolean equals(Object var1) {
        if(this != var1) {
            if(var1 instanceof Developer) {
                Developer var2 = (Developer)var1;
                if(Intrinsics.areEqual(this.name, var2.name) && this.age == var2.age) {
                    return true;
                }
            }

            return false;
        } else {
            return true;
        }
    }
}
```

本章小结

本章我们一起学习了Kotlin与Java的互操作，同时我们用一些简单的示例对比了它们的异同。在这之中，我们能感受到Kotlin的简洁、优雅。我们可以用更少的代码来实现更多的功能。另外，在IDEA中，我们可以直接使用Kotlin插件来直接进行Java代码与Kotlin代码之间的转换（虽然，有些情况下需要我们手工再去稍作改动）。

Kotlin的定位本身之一就是官网首页重点强调的:100% interoperable with Java™。它并不是scala那样另起炉灶, 将类库（例如，集合类）都自己实现了一遍。kotlin是对现有Java的增强,通过扩展方法给java提供了很多诸如fp之类的特性, 但同时始终保持对java的兼容。

而在Java生态领域最为人知的Spring框架，在最新的Spring 5中对Kotlin也作了支持（参看：https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0）。当前，作为Spring大家族中最引人注目的非Spring Boot莫属了。我们即将在下一章中介绍Kotlin集成Spring Boot来开发服务端Web项目。

本章示例代码：https://github.com/EasyKotlin/chapter10_interoperability

第11章 使用Kotlin集成SpringBoot开发Web服务端

我们在前面第2章 “ 2.3 Web RESTFul HelloWorld ” 一节中，已经介绍了使用 Kotlin 结合 SpringBoot 开发一个RESTFul版本的 Hello World。当然，Kotlin与Spring家族的关系不止如此。在 Spring 5.0 M4 中引入了一个专门针对Kotlin的支持。

本章我们就一起来学习怎样使用Kotlin集成SpringBoot、SpringMVC等框架来开发Web服务端应用，同时简单介绍Spring 5.0对Kotlin的支持特性。

11.1 Spring Boot简介

SpringBoot是伴随着Spring4.0诞生的。从字面理解，Boot是引导的意思，SpringBoot帮助开发者快速搭建Spring框架、快速启动一个Web容器等，使得基于Spring的开发过程更加简易。 大部分 Spring Boot Application只要一些极简的配置，即可 “一键运行” 。

SpringBoot的特性如下：

- 创建独立的Spring applications
- 能够使用内嵌的Tomcat, Jetty or Undertow，不需要部署war
- 提供定制化的starter poms来简化maven配置（gradle相同）
- 追求极致的自动配置Spring
- 提供一些生产环境的特性，比如特征指标，健康检查和外部配置。
- 零代码生成和零XML配置

Spring由于其繁琐的配置，一度被人认为 “配置地狱” ，各种XML文件的配置，让人眼花缭乱，而且如果出错了也很难找出原因。而Spring Boot更多的是采用Java Config的方式对Spring进行配置。

11.2 统架构技术栈

本节我们介绍使用 Kotlin 集成 Spring Boot 开发一个完整的博客站点的服务端Web 应用, 它支持 Markdown 写文章, 文章列表分页、搜索查询等功能。

其系统架构技术栈如下表所示:

编程语言	Java，Kotlin
数据库	MySQL , mysql-jdbc-driver, Spring data JPA,
J2EE框架	Spring Boot, Spring MVC
视图模板引擎	Freemarker
前端组件库	jquery， bootstrap, flat UI , Mditor , DataTables
工程构建工具	Gradle

11.3 环境准备

11.3.1 创建工程

首先，我们使用SPRING INITIALIZR来创建一个模板工程。

第一步：访问 http://start.spring.io/， 选择生成一个Gradle项目，使用Kotlin语言，使用的Spring Boot版本是2.0.0 M2。

第二步： 配置项目基本信息。 Group： com.easy.kotlin Artifact：chapter11_kotlin_springboot 以及项目名称、项目描述、包名称等其他的选项。选择jar包方式打包，使用JDK1.8 。

第三步：选择项目依赖。我们这里分别选择了：Web、DevTools、JPA、MySQL、Actuator、Freemarker。

以上三步如下图所示：

点击生成项目，下载zip包，解压后导入IDEA中，我们可以看到一个如下目录结构的工程：

```
.
├── build
│   └── kotlin-build
│       └── caches
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
```

```
├── gradlew
├── gradlew.bat
└── src
    ├── main
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── chapter11_kotlin_springboot
    │   │                   └── Chapter11KotlinSpringbootApplication.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── kotlin
            └── com
                └── easy
                    └── kotlin
                        └── chapter11_kotlin_springboot
                            └── Chapter11KotlinSpringbootApplicationTests.kt

21 directories, 8 files
```

其中，Chapter11KotlinSpringbootApplication.kt是SpringBoot应用的入口启动类。

11.3.2 Gradle配置文件说明

整个工程的Gradle构建配置文件build.gradle的内容如下：

```
buildscript {
    ext {
        kotlinVersion = '1.1.3-2'
        springBootVersion = '2.0.0.M2'
    }
    repositories {
        mavenCentral()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersi
on}")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
        classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
    }
}
```

```
apply plugin: 'kotlin'
apply plugin: 'kotlin-spring'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}


dependencies {
    compile('org.springframework.boot:spring-boot-starter-actuator')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.springframework.boot:spring-boot-starter-freemarker')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
    runtime('org.springframework.boot:spring-boot-devtools')
    runtime('mysql:mysql-connector-java')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

其中主要的配置项如下表说明：

配置项	功能说明
spring-boot-gradle-plugin	SpringBoot集成Gradle的插件
kotlin-gradle-plugin	Kotlin集成Gradle的插件
kotlin-allopen	Kotlin全开放插件。Kotlin 里类默认都是final的,如果声明的类需要被继承则需要使用open 关键字来描述类，这个插件就是把Kotlin中的所有类都open打开，可被继承

spring-boot-starter-actuator	SpringBoot的健康检查监控组件启动器
spring-boot-starter-data-jpa	JPA启动器
spring-boot-starter-freemarker	模板引擎freemarker启动器
kotlin-stdlib-jre8	Kotlin基于JRE8的标准库
kotlin-reflect	Kotlin反射库
spring-boot-devtools	SpringBoot开发者工具，例如：热部署等
mysql-connector-java	Java的MySQL连接器库
spring-boot-starter-test	测试启动器

11.4 数据库层配置

上面的模板工程，我们来直接运行main函数，会发现启动失败，控制台会输出如下报错信息：

```
BeanCreationException: Error creating bean with name 'dataSource' defined in class path
 resource [org/springframework/boot/autoconfigure/jdbc/DataSourceConfiguration$Hikari.c
lass]
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Cannot determine embedded database driver class for database type NONE

Action:

If you want an embedded database please put a supported one on the classpath. If you ha
ve database settings to be loaded from a particular profile you may need to active it (
no profiles are currently active).
```

因为我们还没有配置数据源。我们先在MySQL中新建一个schema：

```
CREATE SCHEMA `blog` DEFAULT CHARACTER SET utf8 ;
```

11.4.1 配置数据源

接着，我们在配置文件application.properties中配置MySQL数据源：

```
# datasource
# datasource: unicode编码的支持，设定为utf-8
spring.datasource.url=jdbc:mysql://localhost:3306/blog?zeroDateTimeBehavior=convertToNu
ll&characterEncoding=utf8&characterSetResults=utf8
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.testWhileIdle=true
spring.datasource.validationQuery=SELECT 1
```

其中，spring.datasource.url 中，为了支持中文的正确显示（防止出现中文乱码），我们需要设置一下编码。

数据库ORM（对象关系映射）层，我们使用spring-data-jpa ：

```
spring.jpa.database=MYSQL
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto=update
# Naming strategy
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

再次运行启动类，控制台输出启动日志：

```
...
                        _  __     _   _ _
                       | |/ /    | | | (_)          _
                       | ' / ___ | |_| |_ _ __    _| |_
                       |  < / _ \| __| | | '_ \  |_   _|
                       | . \ (_) | |_| | | | | |   |_|
                       |_|\_\___/ \__|_|_|_| |_|



              _____           _            ____            _
             / ____|          (_)           |  _ \            | |
            | (___  _ __  _ __ _ _ __   __ _| |_) | ___   ___ | |_
             \___ \| '_ \| '__| | '_ \ / _` |  _ < / _ \ / _ \| __|
```

```
             ____) | |_) | |  | | | | | | (_| | |_) | (_) | (_) | |_
            |_____/| .__/|_|  |_|_| |_|\__, |____/ \___/ \___/ \__|
                   | |                  __/ |
                   |_|                 |___/


2017-07-17 21:10:48.741  INFO 5062 --- [  restartedMain] c.Chapter11KotlinSpringbootApp
licationKt : Starting Chapter11KotlinSpringbootApplicationKt on 192.168.1.6 with PID 50
62 ...
...
2017-07-17 21:19:40.548  INFO 5329 --- [  restartedMain] j.LocalContainerEntityManagerF
actoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2017-07-17 21:11:03.017  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/env || /application/env.json],methods=[GET],produce
s=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto public java
.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()
...
2017-07-17 21:11:03.026  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/beans || /application/beans.json],methods=[GET],pro
duces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto public
java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invok
e()
2017-07-17 21:11:03.028  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/health || /application/health.json],methods=[GET],p
roduces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto publi
c java.lang.Object org.springframework.boot.actuate.endpoint.mvc.HealthMvcEndpoint.invo
ke(javax.servlet.http.HttpServletRequest,java.security.Principal)
...
2017-07-17 21:11:03.060  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/autoconfig || /application/autoconfig.json],methods
=[GET],produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" on
to public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAda
pter.invoke()
...
2017-07-17 21:11:03.478  INFO 5062 --- [  restartedMain] o.s.ui.freemarker.SpringTempla
teLoader   : SpringTemplateLoader for FreeMarker: using resource loader [org.springfram
ework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@6aab8
872: startup date [Mon Jul 17 21:10:48 CST 2017]; root of context hierarchy] and templa
te loader path [classpath:/templates/]
2017-07-17 21:11:03.479  INFO 5062 --- [  restartedMain] o.s.w.s.v.f.FreeMarkerConfigur
er         : ClassTemplateLoader for Spring macros added to FreeMarker configuration
2017-07-17 21:11:03.520  WARN 5062 --- [  restartedMain] o.s.b.a.f.FreeMarkerAutoConfig
uration    : Cannot find template location(s): [classpath:/templates/] (please add some
 templates, check your FreeMarker configuration, or set spring.freemarker.checkTemplate
Location=false)
2017-07-17 21:11:03.713  INFO 5062 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadSe
rver       : LiveReload server is running on port 35729
2017-07-17 21:11:03.871  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter        : Registering beans for JMX exposure on startup
```

```
2017-07-17 21:11:03.874  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter         : Bean with name 'dataSource' has been autodetected for JMX exposure
2017-07-17 21:11:03.886  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter         : Located MBean 'dataSource': registering with JMX server as MBean [com.zaxx
er.hikari:name=dataSource,type=HikariDataSource]
2017-07-17 21:11:03.901  INFO 5062 --- [  restartedMain] o.s.c.support.DefaultLifecycle
Processor   : Starting beans in phase 0
2017-07-17 21:11:04.232  INFO 5062 --- [  restartedMain] o.s.b.w.embedded.tomcat.Tomcat
WebServer   : Tomcat started on port(s): 8000 (http)
2017-07-17 21:11:04.240  INFO 5062 --- [  restartedMain] c.Chapter11KotlinSpringbootApp
licationKt : Started Chapter11KotlinSpringbootApplicationKt in 16.316 seconds (JVM runn
ing for 17.68)
```

关于上面的日志，我们通过下面的表格作简要说明：

日志内容	简要说明
LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory	初始化JAP实体管理器工厂
EndpointHandlerMapping : Mapped "{[/application/beans ...等	SpringBoot健康监控Endpoint 等REST接口
FreeMarkerAutoConfiguration	Freemarker模板引擎自动配置，默认视图文件目录是classpath:/templates/
AnnotationMBeanExporter : Bean with name 'dataSource' has been autodetected for JMX exposure ... Located MBean 'dataSource': registering with JMX server as MBean [com.zaxxer.hikari:name=dataSource,type=HikariDataSource]	数据源Bean通过annotation注解注册MBean到JMX实现监控其运行状态
TomcatWebServer : Tomcat started on port(s): 8000 (http)	SpringBoot默认内嵌了Tomcat，端口我们可以在application.properties中配置
Started Chapter11KotlinSpringbootApplicationKt in 16.316 seconds (JVM running for 17.68)	SpringBoot应用启动成功

11.5 Endpoint监控接口

我们来尝试访问：http://127.0.0.1:8000/application/beans ，浏览器显示如下信息：

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallba
ck.

Mon Jul 17 21:30:35 CST 2017
There was an unexpected error (type=Unauthorized, status=401).
```

```
Full authentication is required to access this resource.
```

提示没有权限访问。我们去控制台日志可以看到下面这行输出：

```
s.b.a.e.m.MvcEndpointSecurityInterceptor : Full authentication is required to access actuator endpoints. Consider adding Spring Security or set 'management.security.enabled' to false.
```

意思是说，要访问这些Endpoints需要权限，可以通过Spring Security来实现权限控制，或者把权限限制去掉：把 `management.security.enabled` 设置为 `false` 。

我们在application.properties里面添加配置：

```
management.security.enabled=false
```

重启应用，再次访问，我们可以看到如下输出：


```
[
  {
    "context": "application:8000",
    "parent": null,
    "beans": [
      {
        "bean": "chapter11KotlinSpringbootApplication",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "com.easy.kotlin.chapter11_kotlin_springboot.Chapter11KotlinSpringbootApplication$$EnhancerBySpringCGLIB$$353fd63e",
        "resource": "null",
        "dependencies": [

        ]
      },
      ...
      {
        "bean": "org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration$$EnhancerBySpringCGLIB$$24d31c25",
        "resource": "null",
```

```
        "dependencies": [
          "dataSource",
          "spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties"
        ]
      },
      {
        "bean": "transactionManager",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.orm.jpa.JpaTransactionManager",
        "resource": "class path resource [org/springframework/boot/autoconfigure/orm/jp
a/HibernateJpaAutoConfiguration.class]",
        "dependencies": [

        ]
      },
      ...
      {
        "bean": "spring.devtools-org.springframework.boot.devtools.autoconfigure.DevToo
lsProperties",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.devtools.autoconfigure.DevToolsProperties",
        "resource": "null",
        "dependencies": [

        ]
      },
      {
        "bean": "org.springframework.orm.jpa.SharedEntityManagerCreator#0",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "com.sun.proxy.$Proxy86",
        "resource": "null",
        "dependencies": [
          "entityManagerFactory"
        ]
      }
    ]
  }
```

可以看出，我们一行代码还没写，只是加了几行配置，SpringBoot已经自动配置初始化了这么多的Bean。我们再访问 http://127.0.0.1:8000/application/health

```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "total": 120108089344,
    "free": 1724157952,
    "threshold": 10485760
  },
  "db": {
    "status": "UP",
    "database": "MySQL",
    "hello": 1
  }
}
```

从上面我们可以看到一些应用的健康状态信息，例如：应用状态、磁盘空间、数据库状态等信息。

11.6 数据库实体类

我们在上面已经完成了MySQL数据源的配置，下面我们来写一个实体类。新建 `package com.easy.kotlin.chapter11_kotlin_springboot.entity` ，然后新建 `Article` 实体类：

```
package com.easy.kotlin.chapter11_kotlin_springboot.entity

import java.util.*
import javax.persistence.*

@Entity
class Article {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Long = 0
    var title: String = ""
    var content: String = ""
    var author: String = ""
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0 //1 Yes 0 No
    var deletedDate: Date = Date()
```

```
    override fun toString(): String {
        return "Article(id=$id, version=$version, title='$title', content='$content', a
uthor='$author', gmtCreated=$gmtCreated, gmtModified=$gmtModified, isDeleted=$isDeleted
, deletedDate=$deletedDate)"
    }

}
```

类似的实体类，我们在Java中需要生成一堆getter/setter方法；如果我们用Scala写还需要加个 注解 @BeanProperty, 例如

```
package com.springboot.in.action.entity

import java.util.Date
import javax.persistence.{ Entity, GeneratedValue, GenerationType, Id }
import scala.language.implicitConversions
import scala.beans.BeanProperty

@Entity
class HttpApi {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @BeanProperty
  var id: Integer = _

  @BeanProperty
  var httpSuiteId: Integer = _
  //用例名称
  @BeanProperty
  var name: String = _

  //用例状态: -1未执行 0失败 1成功
  @BeanProperty
  var state: Integer = _
  ...

}
```

我们这个是一个博客文章的简单实体类。再次重启运行应用，我们去MySQL的Schema: blog 里面去看，发现数据库自动生成了 Table: article , 它的表字段信息如下：

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
author	varchar(255)	YES		NULL	
content	varchar(255)	YES		NULL	

deleted_date	datetime	YES		NULL	
gmt_created	datetime	YES		NULL	
gmt_modified	datetime	YES		NULL	
is_deleted	int(11)	NO		NULL	
title	varchar(255)	YES		NULL	
version	bigint(20)	NO		NULL	-

11.7 数据访问层代码

在Spring Data JPA中，我们只需要实现接口 `CrudRepository<T, ID>` 即可获得一个拥有基本CRUD操作的接口实现了：

```
interface ArticleRepository : CrudRepository<Article, Long>
```

JPA会自动实现ArticleRepository接口中的方法，不需要我们写基本的CRUD操作代码。它的常用的基本CRUD操作方法的简单说明如下表：

方法	功能说明
S save(S entity)	保存给定的实体对象，我们可以使用这个保存之后返回的实例进行进一步操作（保存操作可能会更改实体实例）
findById(ID id)	根据主键id查询
existsById(ID id)	判断是否存在该主键id的记录
findAll()	返回所有记录
findAllById(Iterable ids)	根据主键id集合批量查询
count()	返回总记录条数
deleteById(ID id)	根据主键id删除
deleteAll()	全部删除

当然，如果我们需要自己去实现SQL查询逻辑，我们可以直接使用@Query注解。

```
interface ArticleRepository : CrudRepository<Article, Long> {
    override fun findAll(): MutableList<Article>

    @Query(value = "SELECT * FROM blog.article where title like %?1%", nativeQuery = true)
    fun findByTitle(title: String): MutableList<Article>

    @Query("SELECT a FROM #{#entityName} a where a.content like %:content%")
    fun findByContent(@Param("content") content: String): MutableList<Article>
```

```
    @Query(value = "SELECT * FROM blog.article  where author = ?1", nativeQuery = true)
    fun findByAuthor(author: String): MutableList<Article>
}
```

11.7.1 原生SQL查询

其中，@Query注解里面的value的值就是我们要写的 JP QL语句。另外，JPA的EntityManager API 还提供了创建 Query 实例以执行原生 SQL 语句的createNativeQuery方法。

默认是非原生的JP QL查询模式。如果我们想指定原生SQL查询，只需要设置 `nativeQuery=true` 即可。

11.7.2 模糊查询like写法

另外，我们原生SQL模糊查询like语法,我们在写sql的时候是这样写的

```
like '%?%'
```

但是在JP QL中, 这样写

```
like %?1%
```

11.7.3 参数占位符

其中，查询语句中的 `?1` 是函数参数的占位符，1代表的是参数的位置。

11.7.4 JP QL中的SpEL

另外我们使用JPA的标准查询（Criteria Query）：

```
SELECT a FROM #{#entityName} a where a.content like %:content%
```

其中的 `#{#entityName}` 是SpEL（Spring表达式语言），用来代替本来实体的名称，而Spring data jpa会自动根据Article实体上对应的默认的 `@Entity class Article` ，或者指定 `@Entity(name = "Article") class Article` 自动将实体名称填入 JP QL语句中。

通过把实体类名称抽象出来成为参数，帮助我们解决了项目中很多dao接口的方法除了实体类名称不同，其他操作都相同的问题。

11.7.5 注解参数

我们使用 `@Param("content")` 来指定参数名绑定，然后在JP QL语句中这样引用：

```
:content
```

JP QL 语句中通过": 变量"的格式来指定参数，同时在方法的参数前面使用 @Param 将方法参数与 JP QL 中的命名参数对应。

11.8 控制器层

我们新建子目录controller，然后在下面新建控制器类：

```
@Controller
class ArticleController {

}
```

我们首先，装配数据访问层的接口Bean：

```
@Autowired val articleRepository: ArticleRepository? = null
```

这个接口Bean的实例化由Spring data jpa完成。如果我们去 http://127.0.0.1:8000/application/beans 中查看这个Bean，我们可以看到信息如下：

```
        {
          "bean": "articleRepository",
          "aliases": [

          ],
          "scope": "singleton",
          "type": "com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository",
          "resource": "null",
          "dependencies": [
            "(inner bean)#39c36d98",
            "(inner bean)#19d60142",
            "(inner bean)#1757cb01",
            "(inner bean)#6dd045f0",
            "jpaMappingContext"
          ]
        }
```

我们先来实现一个简单的查询所有记录的REST接口。我们在ArticleRepository中重写了findAll方法：

```
override fun findAll(): MutableList<Article>
```

然后，我们在控制器代码中直接调用这个接口方法：

```
    @GetMapping("listAllArticle")
    @ResponseBody
    fun listAllArticle(): MutableList<Article>? {
        return articleRepository?.findAll()
    }
```

其中，注解@ResponseBody表示把方法返回值直接绑定到响应体（response body）。

11.9 启动初始化CommandLineRunner

为了方便测试用，我们在SpringBoot应用启动的时候初始化几条数据到数据库里。Spring Boot 为我们提供了一个方法，通过实现接口 CommandLineRunner 来实现。这是一个函数式接口：

```
@FunctionalInterface
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

我们只需要创建一个实现接口 CommandLineRunner 的类。很简单，只需要一个类就可以，无需其他配置。 这里我们使用Kotlin的Lambda表达式来写：

```
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
```

11.10 应用启动类

我们在main函数中调用SpringApplication类的静态run方法，我们的SpringBootApplication主类代码如下：

```
package com.easy.kotlin.chapter11_kotlin_springboot

import com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository
import com.easy.kotlin.chapter11_kotlin_springboot.entity.Article
import org.springframework.boot.CommandLineRunner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
```

```kotlin
import org.springframework.context.annotation.Bean
import java.util.*

@SpringBootApplication
class Chapter11KotlinSpringbootApplication {
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
}

fun main(args: Array<String>) {
    SpringApplication.run(Chapter11KotlinSpringbootApplication::class.java, *args)
}
```

这里我们主要关注的是@SpringBootApplication注解，它包括三个注解，简单说明如下表：

注解	功能说明
@SpringBootConfiguration（它包括@Configuration）	表示将该类作用springboot配置文件类。
@EnableAutoConfiguration	表示SpringBoot程序启动时，启动Spring Boot默认的自动配置。
@ComponentScan	表示程序启动时自动扫描当前包及子包下所有类。

11.10.1 启动运行

如果是在IDEA中运行，可以直接点击main函数运行，如下图所示：

```
@SpringBootApplication
class Chapter11KotlinSpringbootApplication {
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
}

fun main(args: Array<String>) {
    ngbootApplication::class.java, *args)
```

Run 'com.easy.kotlin.chap...' ^⇧R
Debug 'com.easy.kotlin.chap...' ^⇧D
Run 'com.easy.kotlin.chap...' with Coverage

如果想在命令行运行，直接在项目根目录下运行命令：

```
$ gradle bootRun
```

我们可以看到控制台的日志输出：

```
2017-07-18 17:42:53.689  INFO 21239 --- [  restartedMain] o.s.b.w.embedded.tomcat.Tomca
tWebServer  : Tomcat started on port(s): 8000 (http)
Hibernate: insert into article (author, content, deleted_date, gmt_created, gmt_modifie
d, is_deleted, title, version) values (?, ?, ?, ?, ?, ?, ?, ?)
2017-07-18 17:42:53.974  INFO 21239 --- [  restartedMain] c.Chapter11KotlinSpringbootAp
plicationKt : Started Chapter11KotlinSpringbootApplicationKt in 16.183 seconds (JVM run
ning for 17.663)
<===========---> 83% EXECUTING [1m 43s]
> :bootRun
```

我们在浏览器中直接访问： http://127.0.0.1:8000/listAllArticle ， 可以看到类似如下输出：

```
[
  {
    "id": 1,
    "version": 0,
    "title": "极简Kotlin教程",
    "content": "Easy Kotlin ",
    "author": "Kotlin",
    "gmtCreated": 1500306475000,
    "gmtModified": 1500306475000,
    "deletedDate": 1500306475000
  },
  {
    "id": 2,
```

```
    "version": 0,
    "title": "极简Kotlin教程",
    "content": "Easy Kotlin ",
    "author": "Kotlin",
    "gmtCreated": 1500306764000,
    "gmtModified": 1500306764000,
    "deletedDate": 1500306764000
  }
]
```

至此，我们已经完成了一个简单的REST接口从数据库到后端的开发。

下面我们继续来写一个前端的文章列表页面。

11.11 Model数据绑定

我们写一个返回ModelAndView对象控制器类，其中数据模型Model中放入文章列表数据，代码如下：

```
    @GetMapping("listAllArticleView")
    fun listAllArticleView(model: Model): ModelAndView {
        model.addAttribute("articles", articleRepository?.findAll())
        return ModelAndView("list")
    }
```

其中，`ModelAndView("list")` 中的"list"表示视图文件的所在目录的相对路径。SpringBoot的默认的视图文件放在src/main/resources/templates目录。

11.12 模板引擎视图页面

我们使用Freemarker模板引擎。我们在templates目录下新建一个list.ftl文件，内容如下：

```
<html>
<head>
    <title>Blog!!!</title>
</head>
<body>
<table>
    <thead>
    <th>序号</th>
    <th>标题</th>
    <th>作者</th>
    <th>发表时间</th>
    <th>操作</th>
    </thead>
```

```
    <tbody>
    <#-- 使用FTL指令 -->
    <#list articles as article>
    <tr>
        <td>${article.id}</td>
        <td>${article.title}</td>
        <td>${article.author}</td>
        <td>${article.gmtModified}</td>
        <td><a href="#" target="_blank">编辑</a></td>
    </tr>
    </#list>
    </tbody>
</table>
</body>
</html>
```

其中，<#list articles as article>是Freemarker的循环指令，${}是 Freemarker引用变量的方式。

提示：关于Freemarker的详细语法可参考 http://freemarker.org/ 。

11.13 运行测试

重启应用，浏览器访问 ： http://127.0.0.1:8000/listAllArticleView ，我们可以看到页面输出：

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑
15	极简Kotlin教程 Tue Jul 18 11:00:50 CST 2017	Kotlin	2017-7-18 11:00:50	编辑
16	极简Kotlin教程 Tue Jul 18 15:44:54 CST 2017	Kotlin	2017-7-18 15:44:54	编辑
17	极简Kotlin教程 Tue Jul 18 17:42:53 CST 2017	Kotlin	2017-7-18 17:42:54	编辑
18	极简Kotlin教程 Tue Jul 18 17:49:57 CST 2017	Kotlin	2017-7-18 17:49:58	编辑
19	极简Kotlin教程 Tue Jul 18 23:27:51 CST 2017	Kotlin	2017-7-18 23:27:52	编辑

到这里，我们已经完成了一个从数据库到前端页面的完整的一个极简的Web应用。

当然，这样的UI样式未免太简陋了一些。下面我们加入前端UI组件美化一下。

11.14 引入前端组件

我们使用基于Bootstrap的前端UI库Flat UI。首先去Flat UI的首页：http://www.bootcss.com/p/flat-ui/ 下载zip包，加压后，放到我们的工程里，放置的目录是：src/main/resources/static 。如下图所示：

```
chapter11_kotlin_springboot ~/kotlin/chapter11_ko
  .gradle
  .idea
  build
  classes
  gradle
  src
    main
      java
      kotlin
      resources
        static
          blog
          flatui
            dist
              css
              fonts
              img
              js
                vendor
                  html5shiv.js
                  jquery.min.js
                  respond.min.js
                  video.js
                  video-js.swf
                flat-ui.js
              index.html
            docs
            fonts
            img
            js
            less
            bower.json
            CHANGELOG.md
            Gruntfile.js
            index.html
            index.js
            package.json
            README.md
          mditor-master
```

我们在list.ftl头部引入静态资源文件：

```
<head>
    <meta charset="utf-8">
    <title>Blog</title>
    <meta name="description"
          content="Blog, using Flat UI Kit Free is a Twitter Bootstrap Framework design
```

```
 and Theme, this responsive framework includes a PSD and HTML version."/>

    <meta name="viewport" content="width=1000, initial-scale=1.0, maximum-scale=1.0">

    <!-- Loading Bootstrap -->
    <link href="/flatui/dist/css/vendor/bootstrap/css/bootstrap.min.css" rel="styleshee
t">

    <!-- Loading Flat UI -->
    <link href="/flatui/dist/css/flat-ui.css" rel="stylesheet">
    <link href="/flatui/docs/assets/css/demo.css" rel="stylesheet">

    <link rel="shortcut icon" href="/flatui/img/favicon.ico">

    <script src="/flatui/dist/js/vendor/jquery.min.js"></script>
    <script src="/flatui/dist/js/flat-ui.js"></script>
    <script src="/flatui/dist/js/vendor/html5shiv.js"></script>
    <script src="/flatui/dist/js/vendor/respond.min.js"></script>

    <link rel="stylesheet" href="/blog/blog.css">
    <script src="/blog/blog.js"></script>
</head>
```

其中，我们的这个SpringBoot应用中默认的静态资源的跟路径是src/main/resources/static，然后我们的HTML代码中引用的路径是在此根目录下的相对路径。

提示：更多的关于Spring Boot静态资源处理内容可以参考文章：http://www.jianshu.com/p/d127c4f78bb8

然后，我们再把我们的文章列表布局优化一下：

```
<div class="container">
    <h1>我的博客</h1>
    <table class="table table-responsive table-bordered">
        <thead>
        <th>序号</th>
        <th>标题</th>
        <th>作者</th>
        <th>发表时间</th>
        <th>操作</th>
        </thead>
        <tbody>
        <#-- 使用FTL指令 -->
        <#list articles as article>
        <tr>
            <td>${article.id}</td>
            <td>${article.title}</td>
            <td>${article.author}</td>
```

```
            <td>${article.gmtModified}</td>
            <td><a href="#" target="_blank">编辑</a></td>
        </tr>
        </#list>
        </tbody>
    </table>
</div>
```

重新build工程，在此访问文章列表页，我们将看到一个比刚才漂亮多了的页面：

我的博客

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑

考虑到头部的静态资源文件基本都是公共的代码，我们单独抽取到一个head.ftl文件中，然后在list.ftl中直接这样引用：

```
<#include "head.ftl">
```

11.15 实现写文章模块

我们在列表页上面添加一个“写文章”的入口：

```
<a href="addArticleView" target="_blank" class="btn btn-primary pull-right add-article"
>写文章</a>
```

其中，btn btn-primary pull-right 这三个css样式类是Flat UI组件的。add-article是我们自定义的样式类：

```
.add-article {
    margin: 20px;
}
```

下面我们来写新建文章的页面。我们写文章的跳转页面路径是 `<a href="addArticleView">`，我们先来新建一个写文章页面addArticleView.ftl：

```
<!DOCTYPE html>
<html>
<#include "head.ftl">
<body>
<div class="container">
    <h2>写文章</h2>

    <form id="addArticleForm" class="form-horizontal">
        <div class="form-group">
            <input type="text" name="title" class="form-control" placeholder="文章标题"
>
        </div>

        <div class="form-group">
            <textarea id="articleContentEditor" type="text" name="content" class="form-
control" rows="20"
                      placeholder=""></textarea>
        </div>

        <div class="form-group save-article">
            <div class="col-sm-offset-2 col-sm-10">
                <button type="submit" class="btn btn-primary" id="addArticleBtn">保存并
发表</button>
            </div>
        </div>
    </form>
</div>
</body>
</html>
```

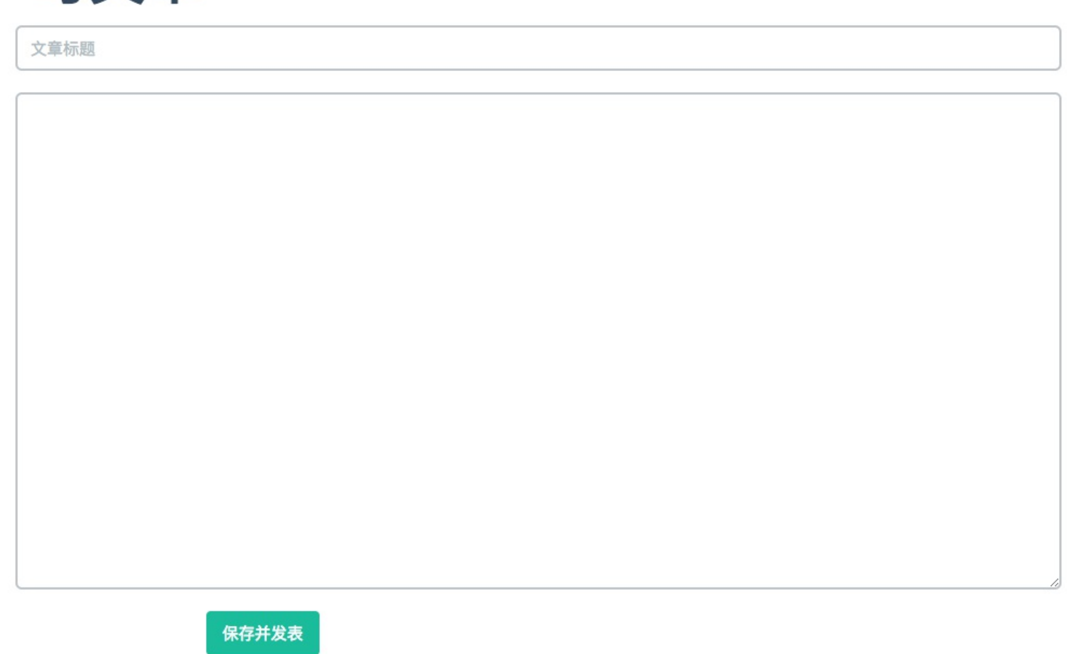
然后，再添加控制器请求转发。这里我们使用集成WebMvcConfigurerAdapter类，重写实现addViewControllers方法的方式来添加一个不带数据传输的，单纯的请求转发的跳转View的RequestMapping Controller：

```
@Configuration
class WebMvcConfig : WebMvcConfigurerAdapter() {
    // 注册简单请求转发跳转View的RequestMapping Controller
    override fun addViewControllers(registry: ViewControllerRegistry?) {
        //写文章的RequestMapping
```

```
        registry?.addViewController("addArticleView")?.setViewName("addArticleView")
    }
}
```

这样前端浏览器来的请求addArticle会直接映射转发到视图addArticle.ftl文件渲染解析。

重启应用，进入到我们的写文章的页面，如下图：



11.15.1 加上导航栏

为了方便页面之间的跳转，我们给我们的博客站点加上导航栏，我们新建一个navbar.ftl文件，内容如下：

```
<nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse"
                    data-target="#example-navbar-collapse">
                <span class="sr-only">切换导航</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">我的博客</a>
        </div>
        <div class="collapse navbar-collapse" id="example-navbar-collapse">
```

```
            <ul class="nav navbar-nav">
                <li class=""><a href="listAllArticleView">文章列表</a></li>
                <li class="active"><a href="addArticleView">写文章</a></li>
                <li><a href="#">关于</a></li>
                <li class="dropdown">
                    <a href="http://www.jianshu.com/nb/12976878" class="dropdown-toggle
" data-toggle="dropdown">
                        Kotlin <b class="caret"></b>
                    </a>
                    <ul class="dropdown-menu">
                        <li><a href="http://www.jianshu.com/nb/12976878" target="_blank
">Kotlin极简教程</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Java</a></li>
                        <li><a href="#">Scala</a></li>
                        <li><a href="#">Groovy</a></li>
                        <li class="divider"></li>
                        <li><a href="http://www.jianshu.com/nb/12066555" target="_blank
">SpringBoot极简教程</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

11.15.2 抽取公共模板文件

考虑到head.ftl、navbar.ftl都是公共的文件，我们把他们单独放到一个common目录下。

然后，我们分别在addArticleView.ftl、listAllArticleView.ftl引用如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
```

加入了导航栏之后，我们的页面现在更加美观了：

我的博客 文章列表 写文章 关于 Kotlin ▾

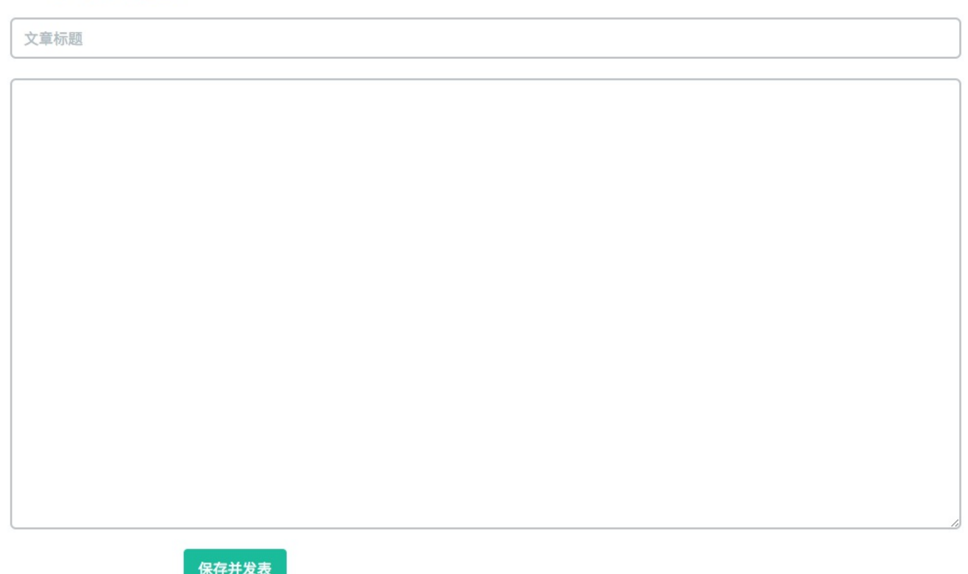
我的博客

写文章

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑

我的博客 文章列表 写文章 关于 Kotlin ▾

写文章



11.15.3 写文章的控制器层接口

控制器层保存接口：

```
@PostMapping("saveArticle")
@ResponseBody
fun saveArticle(article: Article): Article? {
    article.gmtCreated = Date()
    article.gmtModified = Date()
    article.version = 0
```

```
        return articleRepository?.save(article)
    }
```

另外，为了支持较多内容的文章，我们把文章内容字段设置成LONGTEXT:

```
ALTER TABLE `blog`.`article`
CHANGE COLUMN `content` `content` LONGTEXT NULL DEFAULT NULL ;
```

11.15.4 前端 ajax 请求

我们在blog.js中加上ajax POST请求后端接口的逻辑：

```
$(function () {

    $('#addArticleBtn').on('click', function () {
        saveArticle()
    })

    function saveArticle() {
        $.ajax({
            url: "saveArticle",
            data: $('#addArticleForm').serialize(),
            type: "POST",
            async: false,
            success: function (resp) {
                if (resp) {
                    saveArticleSuccess(resp)
                } else {
                    saveArticleFail()
                }
            },
            error: function () {
                saveArticleFail()
            }
        })
    }

    function saveArticleSuccess(resp) {
        alert('保存成功: ' + JSON.stringify(resp))
        window.open('detailArticleView?id=' + resp.id)
    }

    function saveArticleFail() {
        alert("保存失败！")
    }
```

```
})
```

11.15.5 文章详情页

保存成功后，我们默认跳转该文章详情页。 后端控制器代码：

```
    @GetMapping("detailArticleView")
    fun detailArticleView(id: Long, model: Model): ModelAndView {
        model.addAttribute("article", articleRepository?.findById(id)?.get())
        return ModelAndView("detailArticleView")
    }
```

注意，这里的articleRepository?.findById(id) 方法返回的是Optional

，我们调用其get()方法，返回真正的Article实体对象。

前端视图detailArticleView.ftl代码：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
    <h3>${article.title}</h3>
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <div>${article.content}</div>
</div>
</body>
</html>
```

我们在文章列表页中，给每篇文章标题加上跳转文章详情的超链接：

```
<td><a target="_blank" href="detailArticleView?id=${article.id}">${article.title}</a></
td>
```

现在我们的文章列表页面如下：

我的博客 文章列表 写文章 关于 Kotlin

我的博客

写文章

序号	标题	作者	发表时间	操作
52	Kotlin 简介	K	2017-7-19 3:32:23	编辑
51	极简Kotlin教程 Wed Jul 19 03:23:36 CST 2017	Kotlin	2017-7-19 3:23:36	编辑
50	极简Kotlin教程 Wed Jul 19 03:21:01 CST 2017	Kotlin	2017-7-19 3:21:02	编辑
49	极简Kotlin教程 Wed Jul 19 03:07:36 CST 2017	Kotlin	2017-7-19 3:07:36	编辑
48	Kotlin 极简教程	Z	2017-7-19 3:04:05	编辑
47	极简Kotlin教程 Wed Jul 19 03:03:31 CST 2017	Kotlin	2017-7-19 3:03:32	编辑
46	M	M	2017-7-19 2:57:05	编辑
45	B	b	2017-7-19 2:56:16	编辑
44	A	S	2017-7-19 2:55:34	编辑

点击一篇文章标题，即可进入详情页：

我的博客 文章列表 写文章 关于 Kotlin

Kotlin 简介

K

2017-7-19 3:32:23

Kotlin Kotlin 是一个基于 JVM 的新的编程语言，由 JetBrains 开发。 Kotlin可以编译成Java字节码，也可以编译成JavaScript，方便在没有JVM的设备上运行。 JetBrains，作为目前广受欢迎的Java IDE IntelliJ 的提供商，在 Apache 许可下已经开源其Kotlin 编程语言。 Kotlin已正式成为Android官方支持开发语言。

11.16 添加Markdown支持

我们写技术博客文章，最常用的就是使用Markdown了。我们来为我们的博客添加Markdown的支持。我们使用前端js组件Mditor来支持Markdown的编辑。 Mditor是一个简洁、易于集成、方便扩展、期望舒服的编写 markdown 的编辑器。

11.16.1 引入静态资源

```
<link href="/mditor-master/dist/css/mditor.css" rel="stylesheet">
<script src="/mditor-master/dist/js/mditor.js"></script>
```

11.16.2 初始化Mditor

我们在写文章的页面addArticleView.ftl，初始化Mditor如下：

```
<script>
    $(function () {
        //写文章 mditor
        var mditor = Mditor.fromTextarea(document.getElementById('articleContentEditor'
));

        //是否打开分屏
        mditor.split = true;    //打开
        //是否打开预览
        mditor.preivew = true;    //打开
        //是否全屏
        mditor.fullscreen = false;    //关闭
        //获取或设置编辑器的值
        mditor.on('ready', function () {
            mditor.value = '# ';
        });
        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

另外，我们还使用了代码高亮插件highlight.js 。

这样，写文章的页面对应的textarea区域就变成了支持Markdown在线编辑+预览的功能了：

11.16.3 文章详情显示Markdown渲染

下面我们来使我们的详情页也能支持Markdown的渲染显示。

详情页的视图文件detailArticleView.ftl如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
    <h3>${article.title}</h3>
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <textarea id="articleContentShow" placeholder="<#escape  x as x?html>${article.cont
ent}</#escape>" style="display:
              none"></textarea>
    <div id="article-content" class="markdown-body"></div>

</div>
</body>
</html>
```

这里我们把文章内容放到一个隐藏的textarea的placeholder属性中：

```
<textarea id="articleContentShow" placeholder="<#escape  x as x?html>${article.content}
</#escape>" style="display:
              none"></textarea>
```

注意，这里我们作了字符的转义escape，防止有特殊字符导致页面显示错乱。 然后，我们在js中获取这个内容：

```
<script>
    $(function () {
        // 文章详情 mditor
        var parser = new Mditor.Parser();
        var articleContent = document.getElementById('articleContentShow').placeholder
//直接取原本的字符串。不会被转译，默认html页面中textarea区域text需要escape编码
        articleContent = unescape(articleContent);//unescape解码
        var html = parser.parse(articleContent);

        $('#article-content').append(html);

        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

其中，我们是直接调用的Mditor.Parser()函数来解析Markdown字符文本的。

这样我们的详情页也支持了Markdown的渲染显示了：

JP QL中的SpEL

Koltin

2017-7-19 4:56:46

JP QL中的SpEL

另外我们使用JPA的标准查询（Criteria Query）：

```
SELECT a FROM #{#entityName} a where a.content like %:content%
```

其中的 `#{#entityName}` 是SpEL（Spring表达式语言），用来代替本来实体的名称，而Spring data jpa会自动根据Article实体上对应的默认的 `@Entity class Article` ，或者指定 `@Entity(name = "Article") class Article` 自动将实体名称填入 JP QL语句中。

通过把实体类名称抽象出来成为参数，帮助我们解决了项目中很多dao接口的方法除了实体类名称不同，其他操作都相同的问题。

11.17 文章列表分页搜索

为了方便检索我们的博客文章，我们再来给文章列表页面添加分页、搜索、排序等功能。我们使用前端组件DataTables来实现。

提示：更多关于DataTables，可参考： http://www.datatables.club/

11.17.1 引入静态资源文件

```
<link href="/datatables/media/css/jquery.dataTables.css" rel="stylesheet">
<script src="/datatables/media/js/jquery.dataTables.js"></script>
```

11.17.2 给表格加上id

我们给表格加个属性id="articlesDataTable" ：

```
<table id="articlesDataTable" class="table table-responsive table-bordered">
    <thead>
    <th>序号</th>
    <th>标题</th>
    <th>作者</th>
    <th>发表时间</th>
    <th>操作</th>
    </thead>
    <tbody>
    <#-- 使用FTL指令 -->
```

```
        <#list articles as article>
        <tr>
            <td>${article.id}</td>
            <td><a target="_blank" href="detailArticleView?id=${article.id}">${article.
title}</a></td>
            <td>${article.author}</td>
            <td>${article.gmtModified}</td>
            <td><a href="#" target="_blank">编辑</a></td>
        </tr>
        </#list>
        </tbody>
    </table>
```

11.17.3 调用DataTable函数

首先，我们配置一下DataTable的选项：

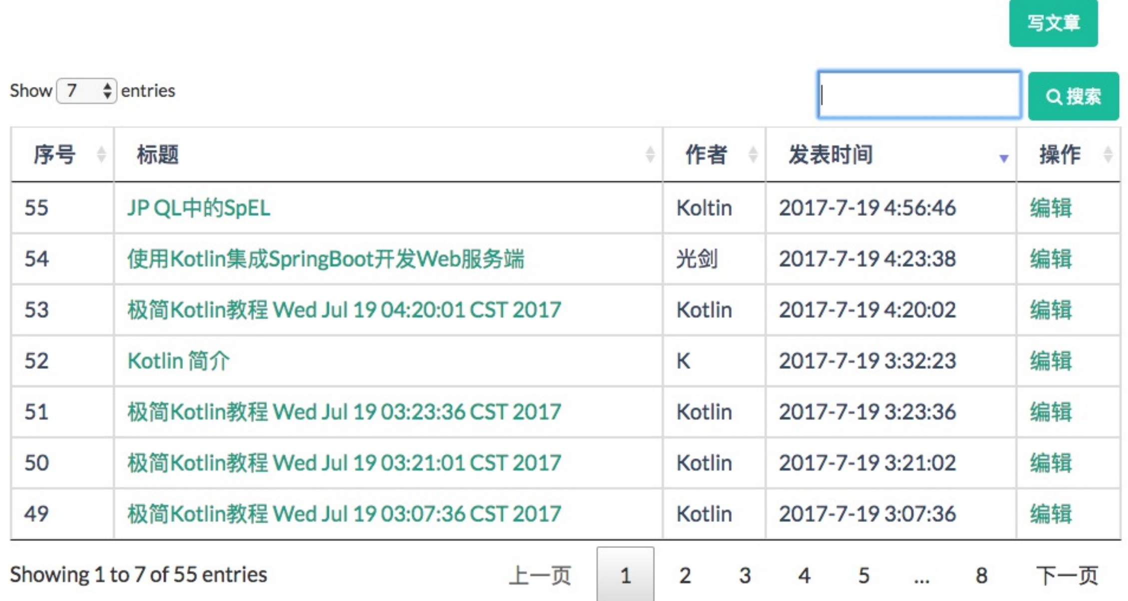
```
    var aLengthMenu = [7, 10, 20, 50, 100, 200]
    var dataTableOptions = {
        "bDestroy": true,
        dom: 'lfrtip',
        "paging": true,
        "lengthChange": true,
        "searching": true,
        "ordering": true,
        "info": true,
        "autoWidth": true,
        "processing": true,
        "stateSave": true,
        responsive: true,
        fixedHeader: false,
        order: [[1, "desc"]],
        "aLengthMenu": aLengthMenu,
        language: {
            "search": "<div style='border-radius:10px;margin-left:auto;margin-right:2px
;width:760px;'>_INPUT_  <span class='btn btn-primary'><span class='fa fa-search'></span
> 搜索</span></div>",

            paginate: {//分页的样式内容
                previous: "上一页",
                next: "下一页",
                first: "第一页",
                last: "最后"
            }
        },
        zeroRecords: "没有内容",//table tbody内容为空时，tbody的内容。
        //下面三者构成了总体的左下角的内容。
```

```
        info: "总计 _TOTAL_ 条,共 _PAGES_ 页，_START_ - _END_ ",//左下角的信息显示，大写的词为关键字。
        infoEmpty: "0条记录",//筛选为空时左下角的显示。
        infoFiltered: ""//筛选之后的左下角筛选提示
    }
```

然后把我们刚才添加了id的表格使用JQuery选择器获取对象，然后直接调用：

```
$('#articlesDataTable').DataTable(dataTableOptions)
```

再次看我们的文章列表页：



写文章

Show 7 entries

搜索

序号	标题	作者	发表时间	操作
55	JP QL中的SpEL	Koltin	2017-7-19 4:56:46	编辑
54	使用Kotlin集成SpringBoot开发Web服务端	光剑	2017-7-19 4:23:38	编辑
53	极简Kotlin教程 Wed Jul 19 04:20:01 CST 2017	Kotlin	2017-7-19 4:20:02	编辑
52	Kotlin 简介	K	2017-7-19 3:32:23	编辑
51	极简Kotlin教程 Wed Jul 19 03:23:36 CST 2017	Kotlin	2017-7-19 3:23:36	编辑
50	极简Kotlin教程 Wed Jul 19 03:21:01 CST 2017	Kotlin	2017-7-19 3:21:02	编辑
49	极简Kotlin教程 Wed Jul 19 03:07:36 CST 2017	Kotlin	2017-7-19 3:07:36	编辑

Showing 1 to 7 of 55 entries　上一页 1 2 3 4 5 ... 8 下一页

已经具备了分页、搜索、排序等功能了。

到这里，我们的这个较为完整的极简博客站点应用基本就开发完成了。

11.18 Spring 5.0对Kotlin的支持

Kotlin 关键性能之一就是能与 Java 库很好地互用。但要在 Spring 中编写惯用的 Kotlin 代码，还需要一段时间的发展。 Spring 对 Java 8 的新支持：函数式 Web 编程、bean 注册 API， 这同样可以在 Kotlin 中使用。

Kotlin 扩展是Kotlin 的编程利器。它能对现有的 API 实现非侵入式的扩展，从而向 Spring中加入 Kotlin 的专有的功能特性。

11.18.1 一种注册 Bean 的新方法

Spring Framework 5.0 引入了一种注册 Bean 的新方法，作为利用 XML 或者 JavaConfig 的 @Configuration 或者 @Bean 的替代方案。简言之就是 Lambda 表达式。

例如用 Java 代码我们会这样写：

```
GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new
    Bar(context.getBean(Foo.class))
);
```

而使用 Kotlin 我们可以将代码写成这样:

```
val context = GenericApplicationContext {
    registerBean<foo>()
    registerBean { Bar(it.getBean<foo>()) }
}
```

11.18.2 Spring Web 函数式 API

Spring 5.0 中的 RouterFunctionDsl 可以让我们使用干净且优雅的 Kotlin 代码来使用崭新的 Spring Web 函数式 API：

```
fun route(request: ServerRequest) = RouterFunctionDsl {
    accept(TEXT_HTML).apply {
            (GET("/user/") or GET("/users/")) { findAllView() }
            GET("/user/{login}") { findViewById() }
    }
    accept(APPLICATION_JSON).apply {
            (GET("/api/user/") or GET("/api/users/")) { findAll() }
            POST("/api/user/") { create() }
            POST("/api/user/{login}") { findOne() }
    }
 } (request)
```

11.18.3 Reactor Kotlin 扩展

Reactor 是 Spring 5.0 中提供的响应式框架。而 reactor-kotlin 项目则是对 Reactor 中使用Kotlin 的支持。目前该项目正在早期阶段。

11.18.4 基于 Kotlin脚本的 Gradle 构建配置

之前我们的 Gradle 构建配置文件都是用Groovy 来编写的，这导致我们基于 Gradle 的 Kotlin 工程还要配置 Groovy 的语法的构建配置文件。

在gradle-script-kotlin 项目中，我们可以直接用 Kotlin 脚本来编写 Gradle 的构建配置文件了。而且 IDE 还为我们提供了在编写配置文件过程中的自动完成功能和重构功能的支持。

11.18.5 基于模板的 Kotlin 脚本

从 4.3 版本开始，Spring 提供了一个 ScriptTemplateView，用于利用支持 JSR-223 的脚本引擎来渲染模板。 Kotlin 1.1-M04 提供了这样的支持，并支持渲染基于 Kotlin 的模板，类似下面这样：

```
import io.spring.demo.User
import io.spring.demo.joinToLine

"""
${include("header", bindings)}
<h1>Title : $title</h1>
<ul>
    ${(users as List<User>).joinToLine{ "<li>User ${it.firstname} ${it.lastname}</li>"
}}
</ul>
${include("footer")}
"""
```

本章小结

本章我们较为细致完整地介绍了使用Kotlin集成SpringBoot进行服务后端开发，并结合简单的前端开发，完成了一个极简的技术博客Web站点。我们可以看到，使用Kotlin结合Spring Boot、Spring MVC、JPA等Java框架的无缝集成，关键是大大简化了我们的代码。同时，在本章最后我们简单介绍了Spring 5.0中对Kotlin的支持诸多新特性，这些新特性令人惊喜。

使用Kotlin编写Spring Boot应用程序越多，我们越觉得这两种技术有着共同的目标，让我们广大程序员可以使用——

- 富有表达性
- 简洁优雅
- 可读

的代码来更高效地编写应用程序，而Spring Framework 5 Kotlin支持将这些技术以更加自然，简单和强大的方式来展现给我们。

未来Spring Framework 5.0 和 Kotlin 结合的开发实践更加值得我们期待。

在下一章中我们将一起学习Kotlin 集成 Gradle 开发的相关内容。

本章项目源码: https://github.com/EasyKotlin/chapter11_kotlin_springboot

第11章 使用Kotlin集成SpringBoot开发Web服务端

我们在前面第2章 “ 2.3 Web RESTFul HelloWorld ” 一节中，已经介绍了使用 Kotlin 结合 SpringBoot 开发一个RESTFul版本的 Hello World。当然，Kotlin与Spring家族的关系不止如此。在 Spring 5.0 M4 中引入了一个专门针对Kotlin的支持。

本章我们就一起来学习怎样使用Kotlin集成SpringBoot、SpringMVC等框架来开发Web服务端应用，同时简单介绍Spring 5.0对Kotlin的支持特性。

11.1 Spring Boot简介

SpringBoot是伴随着Spring4.0诞生的。从字面理解，Boot是引导的意思，SpringBoot帮助开发者快速搭建Spring框架、快速启动一个Web容器等，使得基于Spring的开发过程更加简易。 大部分Spring Boot Application只要一些极简的配置，即可 “一键运行” 。

SpringBoot的特性如下：

- 创建独立的Spring applications
- 能够使用内嵌的Tomcat, Jetty or Undertow，不需要部署war
- 提供定制化的starter poms来简化maven配置（gradle相同）
- 追求极致的自动配置Spring
- 提供一些生产环境的特性，比如特征指标，健康检查和外部配置。
- 零代码生成和零XML配置

Spring由于其繁琐的配置，一度被人认为 “配置地狱” ，各种XML文件的配置，让人眼花缭乱，而且如果出错了也很难找出原因。而Spring Boot更多的是采用Java Config的方式对Spring进行配置。

11.2 统架构技术栈

本节我们介绍使用 Kotlin 集成 Spring Boot 开发一个完整的博客站点的服务端Web 应用, 它支持 Markdown 写文章, 文章列表分页、搜索查询等功能。

其系统架构技术栈如下表所示:

编程语言	Java，Kotlin
数据库	MySQL , mysql-jdbc-driver, Spring data JPA,
J2EE框架	Spring Boot, Spring MVC
视图模板引擎	Freemarker
前端组件库	jquery， bootstrap, flat UI , Mditor , DataTables
工程构建工具	Gradle

11.3 环境准备

11.3.1 创建工程

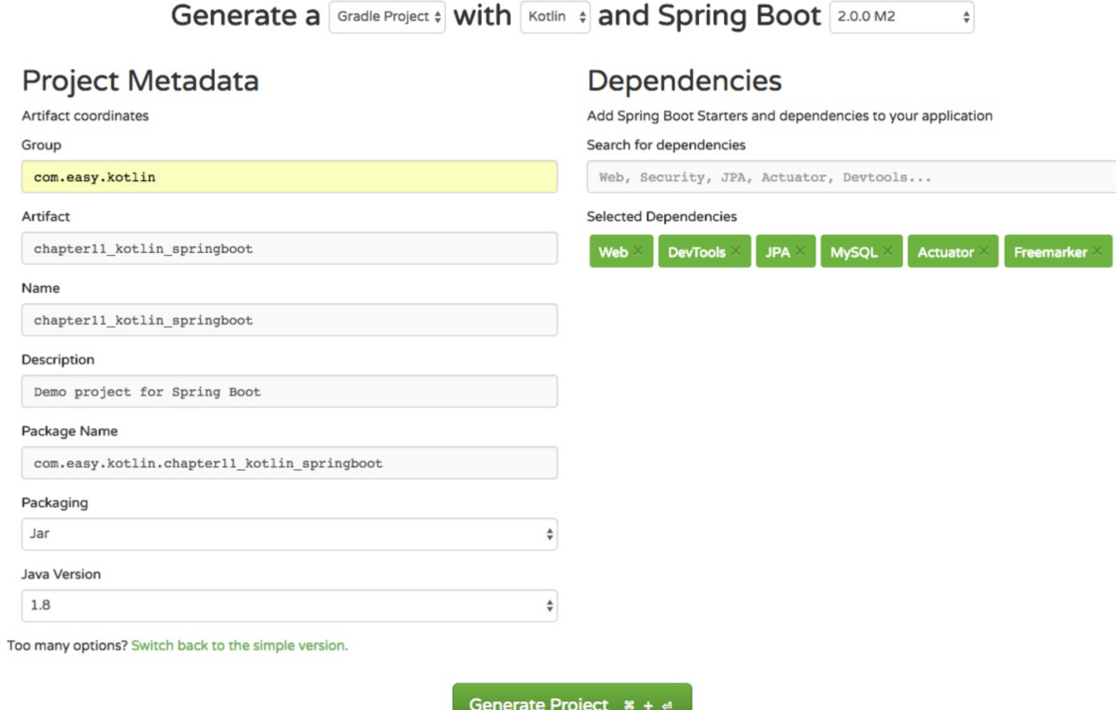
首先，我们使用SPRING INITIALIZR来创建一个模板工程。

第一步：访问 http://start.spring.io/， 选择生成一个Gradle项目，使用Kotlin语言，使用的Spring Boot版本是2.0.0 M2。

第二步： 配置项目基本信息。 Group： com.easy.kotlin Artifact：chapter11_kotlin_springboot 以及项目名称、项目描述、包名称等其他的选项。选择jar包方式打包，使用JDK1.8 。

第三步：选择项目依赖。我们这里分别选择了：Web、DevTools、JPA、MySQL、Actuator、Freemarker。

以上三步如下图所示：



点击生成项目，下载zip包，解压后导入IDEA中，我们可以看到一个如下目录结构的工程：

```
.
├── build
│   └── kotlin-build
│       └── caches
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
```

```
├── gradlew
├── gradlew.bat
└── src
    ├── main
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── chapter11_kotlin_springboot
    │   │                   └── Chapter11KotlinSpringbootApplication.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── kotlin
            └── com
                └── easy
                    └── kotlin
                        └── chapter11_kotlin_springboot
                            └── Chapter11KotlinSpringbootApplicationTests.kt

21 directories, 8 files
```

其中，Chapter11KotlinSpringbootApplication.kt是SpringBoot应用的入口启动类。

11.3.2 Gradle配置文件说明

整个工程的Gradle构建配置文件build.gradle的内容如下：

```
buildscript {
    ext {
        kotlinVersion = '1.1.3-2'
        springBootVersion = '2.0.0.M2'
    }
    repositories {
        mavenCentral()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersi
on}")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
        classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
    }
}
```

```
apply plugin: 'kotlin'
apply plugin: 'kotlin-spring'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}


dependencies {
    compile('org.springframework.boot:spring-boot-starter-actuator')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.springframework.boot:spring-boot-starter-freemarker')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
    runtime('org.springframework.boot:spring-boot-devtools')
    runtime('mysql:mysql-connector-java')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

其中主要的配置项如下表说明：

配置项	功能说明
spring-boot-gradle-plugin	SpringBoot集成Gradle的插件
kotlin-gradle-plugin	Kotlin集成Gradle的插件
kotlin-allopen	Kotlin全开放插件。Kotlin 里类默认都是final的,如果声明的类需要被继承则需要使用open 关键字来描述类，这个插件就是把Kotlin中的所有类都open打开，可被继承

spring-boot-starter-actuator	SpringBoot的健康检查监控组件启动器
spring-boot-starter-data-jpa	JPA启动器
spring-boot-starter-freemarker	模板引擎freemarker启动器
kotlin-stdlib-jre8	Kotlin基于JRE8的标准库
kotlin-reflect	Kotlin反射库
spring-boot-devtools	SpringBoot开发者工具，例如：热部署等
mysql-connector-java	Java的MySQL连接器库
spring-boot-starter-test	测试启动器

11.4 数据库层配置

上面的模板工程，我们来直接运行main函数，会发现启动失败，控制台会输出如下报错信息：

```
BeanCreationException: Error creating bean with name 'dataSource' defined in class path
 resource [org/springframework/boot/autoconfigure/jdbc/DataSourceConfiguration$Hikari.c
lass]
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Cannot determine embedded database driver class for database type NONE

Action:

If you want an embedded database please put a supported one on the classpath. If you ha
ve database settings to be loaded from a particular profile you may need to active it (
no profiles are currently active).
```

因为我们还没有配置数据源。我们先在MySQL中新建一个schema：

```
CREATE SCHEMA `blog` DEFAULT CHARACTER SET utf8 ;
```

11.4.1 配置数据源

接着，我们在配置文件application.properties中配置MySQL数据源：

```
# datasource
# datasource: unicode编码的支持，设定为utf-8
spring.datasource.url=jdbc:mysql://localhost:3306/blog?zeroDateTimeBehavior=convertToNu
ll&characterEncoding=utf8&characterSetResults=utf8
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.testWhileIdle=true
spring.datasource.validationQuery=SELECT 1
```

其中，spring.datasource.url 中，为了支持中文的正确显示（防止出现中文乱码），我们需要设置一下编码。

数据库ORM（对象关系映射）层，我们使用spring-data-jpa ：

```
spring.jpa.database=MYSQL
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto=update
# Naming strategy
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

再次运行启动类，控制台输出启动日志：

```
...
                        _  __     _   _ _
                       | |/ /    | | | (_)        _
                       | ' / ___ | |_| |_ _ __   _| |_
                       |  < / _ \| __| | | '_ \ |_   _|
                       | . \ (_) | |_| | | | | |  |_|
                       |_|\_\___/ \__|_|_|_| |_|



              _____           _           ____             _
             / ____|         (_)          |  _ \            | |
            | (___  _ __  _ __ _ _ __   __ _| |_) | ___   ___ | |_
             \___ \| '_ \| '__| | '_ \ / _` |  _ < / _ \ / _ \| __|
```

```
           ____) | |_) | |  | | | | | | (_| | |_) | (_) | (_) | |_
          |_____/| .__/|_|  |_|_| |_|\__, |____/ \___/ \___/ \__|
                 | |                  __/ |
                 |_|                 |___/


2017-07-17 21:10:48.741  INFO 5062 --- [  restartedMain] c.Chapter11KotlinSpringbootApp
licationKt : Starting Chapter11KotlinSpringbootApplicationKt on 192.168.1.6 with PID 50
62 ...
...
2017-07-17 21:19:40.548  INFO 5329 --- [  restartedMain] j.LocalContainerEntityManagerF
actoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2017-07-17 21:11:03.017  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/env || /application/env.json],methods=[GET],produce
s=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto public java
.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()
...
2017-07-17 21:11:03.026  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/beans || /application/beans.json],methods=[GET],pro
duces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto public
java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invok
e()
2017-07-17 21:11:03.028  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/health || /application/health.json],methods=[GET],p
roduces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" onto publi
c java.lang.Object org.springframework.boot.actuate.endpoint.mvc.HealthMvcEndpoint.invo
ke(javax.servlet.http.HttpServletRequest,java.security.Principal)
...
2017-07-17 21:11:03.060  INFO 5062 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerM
apping     : Mapped "{[/application/autoconfig || /application/autoconfig.json],methods
=[GET],produces=[application/vnd.spring-boot.actuator.v2+json || application/json]}" on
to public java.lang.Object org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAda
pter.invoke()
...
2017-07-17 21:11:03.478  INFO 5062 --- [  restartedMain] o.s.ui.freemarker.SpringTempla
teLoader   : SpringTemplateLoader for FreeMarker: using resource loader [org.springfram
ework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@6aab8
872: startup date [Mon Jul 17 21:10:48 CST 2017]; root of context hierarchy] and templa
te loader path [classpath:/templates/]
2017-07-17 21:11:03.479  INFO 5062 --- [  restartedMain] o.s.w.s.v.f.FreeMarkerConfigur
er         : ClassTemplateLoader for Spring macros added to FreeMarker configuration
2017-07-17 21:11:03.520  WARN 5062 --- [  restartedMain] o.s.b.a.f.FreeMarkerAutoConfig
uration    : Cannot find template location(s): [classpath:/templates/] (please add some
 templates, check your FreeMarker configuration, or set spring.freemarker.checkTemplate
Location=false)
2017-07-17 21:11:03.713  INFO 5062 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadSe
rver       : LiveReload server is running on port 35729
2017-07-17 21:11:03.871  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter        : Registering beans for JMX exposure on startup
```

```
2017-07-17 21:11:03.874  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter        : Bean with name 'dataSource' has been autodetected for JMX exposure
2017-07-17 21:11:03.886  INFO 5062 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExpor
ter        : Located MBean 'dataSource': registering with JMX server as MBean [com.zaxx
er.hikari:name=dataSource,type=HikariDataSource]
2017-07-17 21:11:03.901  INFO 5062 --- [  restartedMain] o.s.c.support.DefaultLifecycle
Processor  : Starting beans in phase 0
2017-07-17 21:11:04.232  INFO 5062 --- [  restartedMain] o.s.b.w.embedded.tomcat.Tomcat
WebServer  : Tomcat started on port(s): 8000 (http)
2017-07-17 21:11:04.240  INFO 5062 --- [  restartedMain] c.Chapter11KotlinSpringbootApp
licationKt : Started Chapter11KotlinSpringbootApplicationKt in 16.316 seconds (JVM runn
ing for 17.68)
```

关于上面的日志，我们通过下面的表格作简要说明：

日志内容	简要说明
LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory	初始化JAP实体管理器工厂
EndpointHandlerMapping : Mapped "{[/application/beans ...等	SpringBoot健康监控Endpoint 等REST接口
FreeMarkerAutoConfiguration	Freemarker模板引擎自动配置，默认视图文件目录是classpath:/templates/
AnnotationMBeanExporter : Bean with name 'dataSource' has been autodetected for JMX exposure ... Located MBean 'dataSource': registering with JMX server as MBean [com.zaxxer.hikari:name=dataSource,type=HikariDataSource]	数据源Bean通过annotation注解注册MBean到JMX实现监控其运行状态
TomcatWebServer : Tomcat started on port(s): 8000 (http)	SpringBoot默认内嵌了Tomcat，端口我们可以在application.properties中配置
Started Chapter11KotlinSpringbootApplicationKt in 16.316 seconds (JVM running for 17.68)	SpringBoot应用启动成功

11.5 Endpoint监控接口

我们来尝试访问：http://127.0.0.1:8000/application/beans ，浏览器显示如下信息：

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallba
ck.

Mon Jul 17 21:30:35 CST 2017
There was an unexpected error (type=Unauthorized, status=401).
```

```
Full authentication is required to access this resource.
```

提示没有权限访问。我们去控制台日志可以看到下面这行输出：

```
s.b.a.e.m.MvcEndpointSecurityInterceptor : Full authentication is required to access ac
tuator endpoints. Consider adding Spring Security or set 'management.security.enabled'
to false.
```

意思是说，要访问这些Endpoints需要权限，可以通过Spring Security来实现权限控制，或者把权限限制去掉：把 `management.security.enabled` 设置为 `false` 。

我们在application.properties里面添加配置：

```
management.security.enabled=false
```

重启应用，再次访问，我们可以看到如下输出：

```
[
  {
    "context": "application:8000",
    "parent": null,
    "beans": [
      {
        "bean": "chapter11KotlinSpringbootApplication",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "com.easy.kotlin.chapter11_kotlin_springboot.Chapter11KotlinSpringbootA
pplication$$EnhancerBySpringCGLIB$$353fd63e",
        "resource": "null",
        "dependencies": [

        ]
      },
      ...
      {
        "bean": "org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfigu
ration",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfigu
ration$$EnhancerBySpringCGLIB$$24d31c25",
        "resource": "null",
```

```
        "dependencies": [
          "dataSource",
          "spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties"
        ]
      },
      {
        "bean": "transactionManager",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.orm.jpa.JpaTransactionManager",
        "resource": "class path resource [org/springframework/boot/autoconfigure/orm/jp
a/HibernateJpaAutoConfiguration.class]",
        "dependencies": [

        ]
      },
      ...
      {
        "bean": "spring.devtools-org.springframework.boot.devtools.autoconfigure.DevToo
lsProperties",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.devtools.autoconfigure.DevToolsProperties",
        "resource": "null",
        "dependencies": [

        ]
      },
      {
        "bean": "org.springframework.orm.jpa.SharedEntityManagerCreator#0",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "com.sun.proxy.$Proxy86",
        "resource": "null",
        "dependencies": [
          "entityManagerFactory"
        ]
      }
    ]
  }
```

可以看出，我们一行代码还没写，只是加了几行配置，SpringBoot已经自动配置初始化了这么多的Bean。我们再访问 http://127.0.0.1:8000/application/health

```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "total": 120108089344,
    "free": 1724157952,
    "threshold": 10485760
  },
  "db": {
    "status": "UP",
    "database": "MySQL",
    "hello": 1
  }
}
```

从上面我们可以看到一些应用的健康状态信息，例如：应用状态、磁盘空间、数据库状态等信息。

11.6 数据库实体类

我们在上面已经完成了MySQL数据源的配置，下面我们来写一个实体类。新建 `package com.easy.kotlin.chapter11_kotlin_springboot.entity` ，然后新建 `Article` 实体类：

```
package com.easy.kotlin.chapter11_kotlin_springboot.entity

import java.util.*
import javax.persistence.*

@Entity
class Article {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Long = 0
    var title: String = ""
    var content: String = ""
    var author: String = ""
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0 //1 Yes 0 No
    var deletedDate: Date = Date()
```

```
    override fun toString(): String {
        return "Article(id=$id, version=$version, title='$title', content='$content', a
uthor='$author', gmtCreated=$gmtCreated, gmtModified=$gmtModified, isDeleted=$isDeleted
, deletedDate=$deletedDate)"
    }

}
```

类似的实体类，我们在Java中需要生成一堆getter/setter方法；如果我们用Scala写还需要加个 注解 @BeanProperty, 例如

```
package com.springboot.in.action.entity

import java.util.Date
import javax.persistence.{ Entity, GeneratedValue, GenerationType, Id }
import scala.language.implicitConversions
import scala.beans.BeanProperty

@Entity
class HttpApi {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @BeanProperty
  var id: Integer = _

  @BeanProperty
  var httpSuiteId: Integer = _
  //用例名称
  @BeanProperty
  var name: String = _

  //用例状态: -1未执行 0失败 1成功
  @BeanProperty
  var state: Integer = _
  ...

}
```

我们这个是一个博客文章的简单实体类。再次重启运行应用，我们去MySQL的Schema: blog 里面去看，发现数据库自动生成了 Table: article , 它的表字段信息如下：

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
author	varchar(255)	YES		NULL	
content	varchar(255)	YES		NULL	

deleted_date	datetime	YES		NULL	
gmt_created	datetime	YES		NULL	
gmt_modified	datetime	YES		NULL	
is_deleted	int(11)	NO		NULL	
title	varchar(255)	YES		NULL	
version	bigint(20)	NO		NULL	-

11.7 数据访问层代码

在Spring Data JPA中，我们只需要实现接口 `CrudRepository<T, ID>` 即可获得一个拥有基本CRUD操作的接口实现了：

```
interface ArticleRepository : CrudRepository<Article, Long>
```

JPA会自动实现ArticleRepository接口中的方法，不需要我们写基本的CRUD操作代码。它的常用的基本CRUD操作方法的简单说明如下表：

方法	功能说明
S save(S entity)	保存给定的实体对象，我们可以使用这个保存之后返回的实例进行进一步操作（保存操作可能会更改实体实例）
findById(ID id)	根据主键id查询
existsById(ID id)	判断是否存在该主键id的记录
findAll()	返回所有记录
findAllById(Iterable ids)	根据主键id集合批量查询
count()	返回总记录条数
deleteById(ID id)	根据主键id删除
deleteAll()	全部删除

当然，如果我们需要自己去实现SQL查询逻辑，我们可以直接使用@Query注解。

```
interface ArticleRepository : CrudRepository<Article, Long> {
    override fun findAll(): MutableList<Article>

    @Query(value = "SELECT * FROM blog.article where title like %?1%", nativeQuery = tr
ue)
    fun findByTitle(title: String): MutableList<Article>

    @Query("SELECT a FROM #{#entityName} a where a.content like %:content%")
    fun findByContent(@Param("content") content: String): MutableList<Article>
```

```
    @Query(value = "SELECT * FROM blog.article  where author = ?1", nativeQuery = true)
    fun findByAuthor(author: String): MutableList<Article>
}
```

11.7.1 原生SQL查询

其中，@Query注解里面的value的值就是我们要写的 JP QL语句。另外，JPA的EntityManager API 还提供了创建 Query 实例以执行原生 SQL 语句的createNativeQuery方法。

默认是非原生的JP QL查询模式。如果我们想指定原生SQL查询，只需要设置 `nativeQuery=true` 即可。

11.7.2 模糊查询like写法

另外，我们原生SQL模糊查询like语法,我们在写sql的时候是这样写的

```
like '%?%'
```

但是在JP QL中, 这样写

```
like %?1%
```

11.7.3 参数占位符

其中，查询语句中的 `?1` 是函数参数的占位符，1代表的是参数的位置。

11.7.4 JP QL中的SpEL

另外我们使用JPA的标准查询（Criteria Query）：

```
SELECT a FROM #{#entityName} a where a.content like %:content%
```

其中的 `#{#entityName}` 是SpEL（Spring表达式语言），用来代替本来实体的名称，而Spring data jpa会自动根据Article实体上对应的默认的 `@Entity class Article` ，或者指定 `@Entity(name = "Article") class Article` 自动将实体名称填入 JP QL语句中。

通过把实体类名称抽象出来成为参数，帮助我们解决了项目中很多dao接口的方法除了实体类名称不同，其他操作都相同的问题。

11.7.5 注解参数

我们使用 `@Param("content")` 来指定参数名绑定，然后在JP QL语句中这样引用：

```
:content
```

JP QL 语句中通过": 变量"的格式来指定参数，同时在方法的参数前面使用 @Param 将方法参数与 JP QL 中的命名参数对应。

11.8 控制器层

我们新建子目录controller，然后在下面新建控制器类：

```
@Controller
class ArticleController {

}
```

我们首先，装配数据访问层的接口Bean：

```
@Autowired val articleRepository: ArticleRepository? = null
```

这个接口Bean的实例化由Spring data jpa完成。如果我们去 http://127.0.0.1:8000/application/beans 中查看这个Bean，我们可以看到信息如下：

```
    {
      "bean": "articleRepository",
      "aliases": [

      ],
      "scope": "singleton",
      "type": "com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository",
      "resource": "null",
      "dependencies": [
        "(inner bean)#39c36d98",
        "(inner bean)#19d60142",
        "(inner bean)#1757cb01",
        "(inner bean)#6dd045f0",
        "jpaMappingContext"
      ]
    }
```

我们先来实现一个简单的查询所有记录的REST接口。我们在ArticleRepository中重写了findAll方法：

```
override fun findAll(): MutableList<Article>
```

然后，我们在控制器代码中直接调用这个接口方法：

```
    @GetMapping("listAllArticle")
    @ResponseBody
    fun listAllArticle(): MutableList<Article>? {
        return articleRepository?.findAll()
    }
```

其中，注解@ResponseBody表示把方法返回值直接绑定到响应体（response body）。

11.9 启动初始化CommandLineRunner

为了方便测试用，我们在SpringBoot应用启动的时候初始化几条数据到数据库里。Spring Boot 为我们提供了一个方法，通过实现接口 CommandLineRunner 来实现。这是一个函数式接口：

```
@FunctionalInterface
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

我们只需要创建一个实现接口 CommandLineRunner 的类。很简单，只需要一个类就可以，无需其他配置。 这里我们使用Kotlin的Lambda表达式来写：

```
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
```

11.10 应用启动类

我们在main函数中调用SpringApplication类的静态run方法，我们的SpringBootApplication主类代码如下：

```
package com.easy.kotlin.chapter11_kotlin_springboot

import com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository
import com.easy.kotlin.chapter11_kotlin_springboot.entity.Article
import org.springframework.boot.CommandLineRunner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
```

```
import org.springframework.context.annotation.Bean
import java.util.*

@SpringBootApplication
class Chapter11KotlinSpringbootApplication {
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
}

fun main(args: Array<String>) {
    SpringApplication.run(Chapter11KotlinSpringbootApplication::class.java, *args)
}
```

这里我们主要关注的是@SpringBootApplication注解，它包括三个注解，简单说明如下表：

注解	功能说明
@SpringBootConfiguration（它包括@Configuration）	表示将该类作用springboot配置文件类。
@EnableAutoConfiguration	表示SpringBoot程序启动时，启动Spring Boot默认的自动配置。
@ComponentScan	表示程序启动时自动扫描当前包及子包下所有类。

11.10.1 启动运行

如果是在IDEA中运行，可以直接点击main函数运行，如下图所示：

```
@SpringBootApplication
class Chapter11KotlinSpringbootApplication {
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简Kotlin教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
}

fun main(args: Array<String>) {
                                                  ngbootApplication::class.java, *args)
```

```
Run 'com.easy.kotlin.chap...'                 ^⇧R
Debug 'com.easy.kotlin.chap...'               ^⇧D
Run 'com.easy.kotlin.chap...' with Coverage
```

如果想在命令行运行，直接在项目根目录下运行命令：

```
$ gradle bootRun
```

我们可以看到控制台的日志输出：

```
2017-07-18 17:42:53.689  INFO 21239 --- [  restartedMain] o.s.b.w.embedded.tomcat.Tomca
tWebServer  : Tomcat started on port(s): 8000 (http)
Hibernate: insert into article (author, content, deleted_date, gmt_created, gmt_modifie
d, is_deleted, title, version) values (?, ?, ?, ?, ?, ?, ?, ?)
2017-07-18 17:42:53.974  INFO 21239 --- [  restartedMain] c.Chapter11KotlinSpringbootAp
plicationKt : Started Chapter11KotlinSpringbootApplicationKt in 16.183 seconds (JVM run
ning for 17.663)
<===========---> 83% EXECUTING [1m 43s]
> :bootRun
```

我们在浏览器中直接访问： http://127.0.0.1:8000/listAllArticle ， 可以看到类似如下输出：

```
[
  {
    "id": 1,
    "version": 0,
    "title": "极简Kotlin教程",
    "content": "Easy Kotlin ",
    "author": "Kotlin",
    "gmtCreated": 1500306475000,
    "gmtModified": 1500306475000,
    "deletedDate": 1500306475000
  },
  {
    "id": 2,
```

```
    "version": 0,
    "title": "极简Kotlin教程",
    "content": "Easy Kotlin ",
    "author": "Kotlin",
    "gmtCreated": 1500306764000,
    "gmtModified": 1500306764000,
    "deletedDate": 1500306764000
  }
]
```

至此，我们已经完成了一个简单的REST接口从数据库到后端的开发。

下面我们继续来写一个前端的文章列表页面。

11.11 Model数据绑定

我们写一个返回ModelAndView对象控制器类，其中数据模型Model中放入文章列表数据，代码如下：

```
    @GetMapping("listAllArticleView")
    fun listAllArticleView(model: Model): ModelAndView {
        model.addAttribute("articles", articleRepository?.findAll())
        return ModelAndView("list")
    }
```

其中，`ModelAndView("list")` 中的"list"表示视图文件的所在目录的相对路径。SpringBoot的默认的视图文件放在src/main/resources/templates目录。

11.12 模板引擎视图页面

我们使用Freemarker模板引擎。我们在templates目录下新建一个list.ftl文件，内容如下：

```
<html>
<head>
    <title>Blog!!!</title>
</head>
<body>
<table>
    <thead>
    <th>序号</th>
    <th>标题</th>
    <th>作者</th>
    <th>发表时间</th>
    <th>操作</th>
    </thead>
```

```
    <tbody>
    <#-- 使用FTL指令 -->
    <#list articles as article>
    <tr>
        <td>${article.id}</td>
        <td>${article.title}</td>
        <td>${article.author}</td>
        <td>${article.gmtModified}</td>
        <td><a href="#" target="_blank">编辑</a></td>
    </tr>
    </#list>
    </tbody>
</table>
</body>
</html>
```

其中，<#list articles as article>是Freemarker的循环指令，${}是 Freemarker引用变量的方式。

提示：关于Freemarker的详细语法可参考 http://freemarker.org/ 。

11.13 运行测试

重启应用，浏览器访问 ： http://127.0.0.1:8000/listAllArticleView ， 我们可以看到页面输出：

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑
15	极简Kotlin教程 Tue Jul 18 11:00:50 CST 2017	Kotlin	2017-7-18 11:00:50	编辑
16	极简Kotlin教程 Tue Jul 18 15:44:54 CST 2017	Kotlin	2017-7-18 15:44:54	编辑
17	极简Kotlin教程 Tue Jul 18 17:42:53 CST 2017	Kotlin	2017-7-18 17:42:54	编辑
18	极简Kotlin教程 Tue Jul 18 17:49:57 CST 2017	Kotlin	2017-7-18 17:49:58	编辑
19	极简Kotlin教程 Tue Jul 18 23:27:51 CST 2017	Kotlin	2017-7-18 23:27:52	编辑

到这里，我们已经完成了一个从数据库到前端页面的完整的一个极简的Web应用。

当然，这样的UI样式未免太简陋了一些。下面我们加入前端UI组件美化一下。

11.14 引入前端组件

我们使用基于Bootstrap的前端UI库Flat UI。首先去Flat UI的首页：http://www.bootcss.com/p/flat-ui/ 下载zip包，加压后，放到我们的工程里，放置的目录是：src/main/resources/static 。如下图所示：

```
chapter11_kotlin_springboot ~/kotlin/chapter11_ko
  .gradle
  .idea
  build
  classes
  gradle
  src
    main
      java
      kotlin
      resources
        static
          blog
          flatui
            dist
              css
              fonts
              img
              js
                vendor
                  html5shiv.js
                  jquery.min.js
                  respond.min.js
                  video.js
                  video-js.swf
                flat-ui.js
              index.html
            docs
            fonts
            img
            js
            less
            bower.json
            CHANGELOG.md
            Gruntfile.js
            index.html
            index.js
            package.json
            README.md
          mditor-master
```

我们在list.ftl头部引入静态资源文件：

```
<head>
    <meta charset="utf-8">
    <title>Blog</title>
    <meta name="description"
          content="Blog, using Flat UI Kit Free is a Twitter Bootstrap Framework design
```

```
 and Theme, this responsive framework includes a PSD and HTML version."/>

    <meta name="viewport" content="width=1000, initial-scale=1.0, maximum-scale=1.0">

    <!-- Loading Bootstrap -->
    <link href="/flatui/dist/css/vendor/bootstrap/css/bootstrap.min.css" rel="styleshee
t">

    <!-- Loading Flat UI -->
    <link href="/flatui/dist/css/flat-ui.css" rel="stylesheet">
    <link href="/flatui/docs/assets/css/demo.css" rel="stylesheet">

    <link rel="shortcut icon" href="/flatui/img/favicon.ico">

    <script src="/flatui/dist/js/vendor/jquery.min.js"></script>
    <script src="/flatui/dist/js/flat-ui.js"></script>
    <script src="/flatui/dist/js/vendor/html5shiv.js"></script>
    <script src="/flatui/dist/js/vendor/respond.min.js"></script>

    <link rel="stylesheet" href="/blog/blog.css">
    <script src="/blog/blog.js"></script>
</head>
```

其中，我们的这个SpringBoot应用中默认的静态资源的跟路径是src/main/resources/static，然后我们的HTML代码中引用的路径是在此根目录下的相对路径。

提示：更多的关于Spring Boot静态资源处理内容可以参考文章：http://www.jianshu.com/p/d127c4f78bb8

然后，我们再把我们的文章列表布局优化一下：

```
<div class="container">
    <h1>我的博客</h1>
    <table class="table table-responsive table-bordered">
        <thead>
        <th>序号</th>
        <th>标题</th>
        <th>作者</th>
        <th>发表时间</th>
        <th>操作</th>
        </thead>
        <tbody>
        <#-- 使用FTL指令 -->
        <#list articles as article>
        <tr>
            <td>${article.id}</td>
            <td>${article.title}</td>
            <td>${article.author}</td>
```

```
            <td>${article.gmtModified}</td>
            <td><a href="#" target="_blank">编辑</a></td>
        </tr>
        </#list>
        </tbody>
    </table>
</div>
```

重新build工程，在此访问文章列表页，我们将看到一个比刚才漂亮多了的页面：

我的博客

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑

考虑到头部的静态资源文件基本都是公共的代码，我们单独抽取到一个head.ftl文件中，然后在list.ftl中直接这样引用：

```
<#include "head.ftl">
```

11.15 实现写文章模块

我们在列表页上面添加一个“写文章”的入口：

```
<a href="addArticleView" target="_blank" class="btn btn-primary pull-right add-article"
>写文章</a>
```

其中，btn btn-primary pull-right 这三个css样式类是Flat UI组件的。add-article是我们自定义的样式类：

```
.add-article {
    margin: 20px;
}
```

下面我们来写新建文章的页面。我们写文章的跳转页面路径是 `<a href="addArticleView">`, 我们先来新建一个写文章页面addArticleView.ftl：

```
<!DOCTYPE html>
<html>
<#include "head.ftl">
<body>
<div class="container">
    <h2>写文章</h2>

    <form id="addArticleForm" class="form-horizontal">
        <div class="form-group">
            <input type="text" name="title" class="form-control" placeholder="文章标题"
>
        </div>

        <div class="form-group">
            <textarea id="articleContentEditor" type="text" name="content" class="form-
control" rows="20"
                      placeholder=""></textarea>
        </div>

        <div class="form-group save-article">
            <div class="col-sm-offset-2 col-sm-10">
                <button type="submit" class="btn btn-primary" id="addArticleBtn">保存并
发表</button>
            </div>
        </div>
    </form>
</div>
</body>
</html>
```

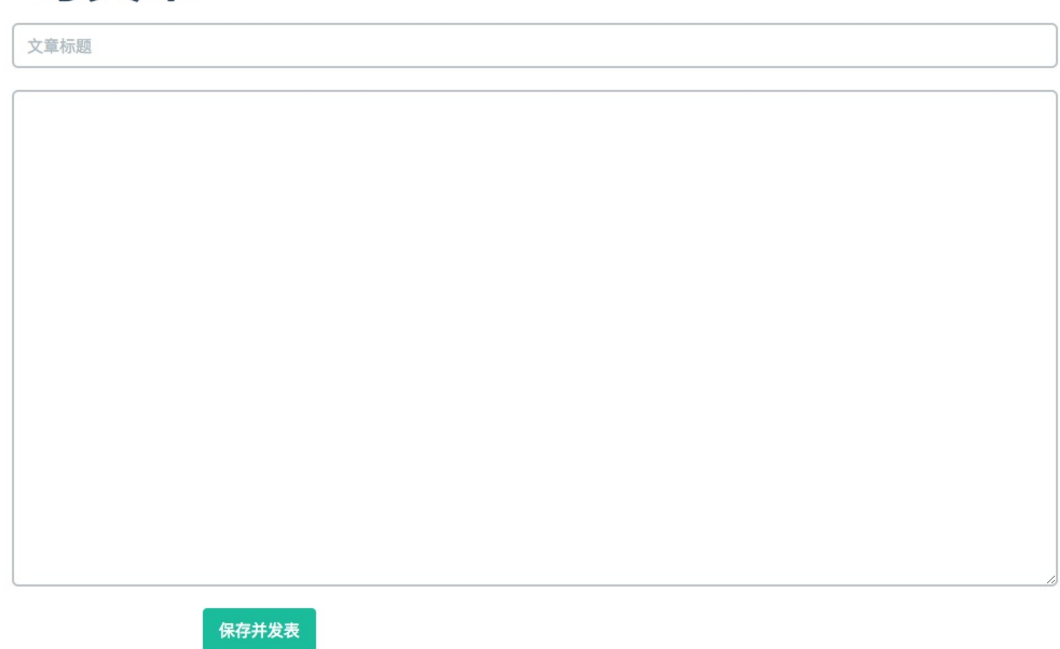
然后，再添加控制器请求转发。这里我们使用集成WebMvcConfigurerAdapter类，重写实现addViewControllers方法的方式来添加一个不带数据传输的，单纯的请求转发的跳转View的RequestMapping Controller：

```
@Configuration
class WebMvcConfig : WebMvcConfigurerAdapter() {
    // 注册简单请求转发跳转View的RequestMapping Controller
    override fun addViewControllers(registry: ViewControllerRegistry?) {
        //写文章的RequestMapping
```

```
        registry?.addViewController("addArticleView")?.setViewName("addArticleView")
    }
}
```

这样前端浏览器来的请求addArticle会直接映射转发到视图addArticle.ftl文件渲染解析。

重启应用，进入到我们的写文章的页面，如下图：



11.15.1 加上导航栏

为了方便页面之间的跳转，我们给我们的博客站点加上导航栏，我们新建一个navbar.ftl文件，内容如下：

```
<nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse"
                    data-target="#example-navbar-collapse">
                <span class="sr-only">切换导航</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">我的博客</a>
        </div>
        <div class="collapse navbar-collapse" id="example-navbar-collapse">
```

```
            <ul class="nav navbar-nav">
                <li class=""><a href="listAllArticleView">文章列表</a></li>
                <li class="active"><a href="addArticleView">写文章</a></li>
                <li><a href="#">关于</a></li>
                <li class="dropdown">
                    <a href="http://www.jianshu.com/nb/12976878" class="dropdown-toggle
" data-toggle="dropdown">
                        Kotlin <b class="caret"></b>
                    </a>
                    <ul class="dropdown-menu">
                        <li><a href="http://www.jianshu.com/nb/12976878" target="_blank
">Kotlin极简教程</a></li>
                        <li class="divider"></li>
                        <li><a href="#">Java</a></li>
                        <li><a href="#">Scala</a></li>
                        <li><a href="#">Groovy</a></li>
                        <li class="divider"></li>
                        <li><a href="http://www.jianshu.com/nb/12066555" target="_blank
">SpringBoot极简教程</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

11.15.2 抽取公共模板文件

考虑到head.ftl、navbar.ftl都是公共的文件，我们把他们单独放到一个common目录下。

然后，我们分别在addArticleView.ftl、listAllArticleView.ftl引用如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
```

加入了导航栏之后，我们的页面现在更加美观了：

我的博客 文章列表 写文章 关于 Kotlin ▾

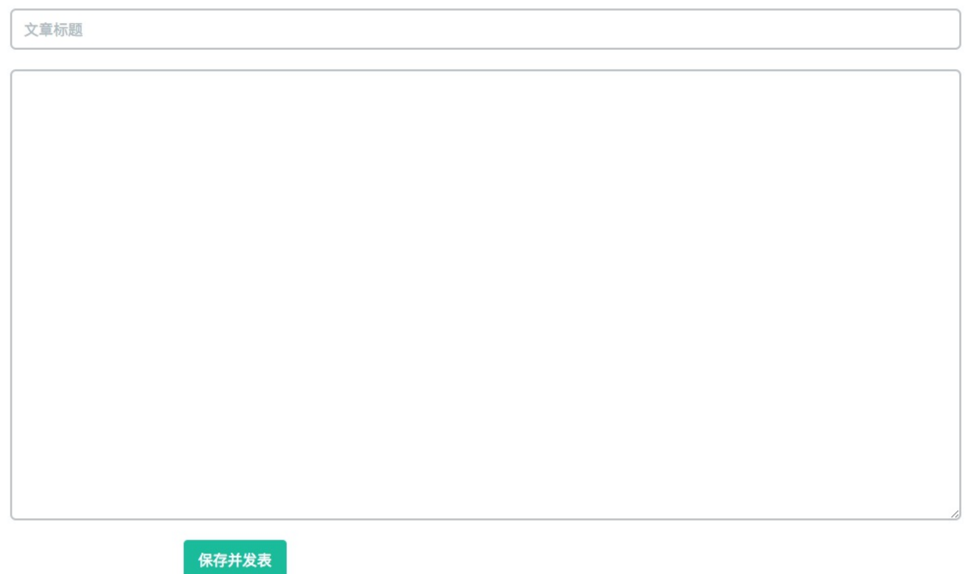
我的博客

写文章

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑

我的博客 文章列表 写文章 关于 Kotlin ▾

写文章



11.15.3 写文章的控制器层接口

控制器层保存接口：

```
    @PostMapping("saveArticle")
    @ResponseBody
    fun saveArticle(article: Article): Article? {
        article.gmtCreated = Date()
        article.gmtModified = Date()
        article.version = 0
```

```
        return articleRepository?.save(article)
    }
```

另外，为了支持较多内容的文章，我们把文章内容字段设置成LONGTEXT:

```
ALTER TABLE `blog`.`article`
CHANGE COLUMN `content` `content` LONGTEXT NULL DEFAULT NULL ;
```

11.15.4 前端 ajax 请求

我们在blog.js中加上ajax POST请求后端接口的逻辑：

```
$(function () {

    $('#addArticleBtn').on('click', function () {
        saveArticle()
    })

    function saveArticle() {
        $.ajax({
            url: "saveArticle",
            data: $('#addArticleForm').serialize(),
            type: "POST",
            async: false,
            success: function (resp) {
                if (resp) {
                    saveArticleSuccess(resp)
                } else {
                    saveArticleFail()
                }
            },
            error: function () {
                saveArticleFail()
            }
        })
    }

    function saveArticleSuccess(resp) {
        alert('保存成功: ' + JSON.stringify(resp))
        window.open('detailArticleView?id=' + resp.id)
    }

    function saveArticleFail() {
        alert("保存失败！")
    }

```

```
})
```

11.15.5 文章详情页

保存成功后，我们默认跳转该文章详情页。 后端控制器代码：

```
    @GetMapping("detailArticleView")
    fun detailArticleView(id: Long, model: Model): ModelAndView {
        model.addAttribute("article", articleRepository?.findById(id)?.get())
        return ModelAndView("detailArticleView")
    }
```

注意，这里的articleRepository?.findById(id) 方法返回的是Optional

， 我们调用其get()方法，返回真正的Article实体对象。

前端视图detailArticleView.ftl代码：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
    <h3>${article.title}</h3>
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <div>${article.content}</div>
</div>
</body>
</html>
```

我们在文章列表页中，给每篇文章标题加上跳转文章详情的超链接：

```
<td><a target="_blank" href="detailArticleView?id=${article.id}">${article.title}</a></
td>
```

现在我们的文章列表页面如下：

我的博客 文章列表 写文章 关于 Kotlin ▾

我的博客

写文章

序号	标题	作者	发表时间	操作
52	Kotlin 简介	K	2017-7-19 3:32:23	编辑
51	极简Kotlin教程 Wed Jul 19 03:23:36 CST 2017	Kotlin	2017-7-19 3:23:36	编辑
50	极简Kotlin教程 Wed Jul 19 03:21:01 CST 2017	Kotlin	2017-7-19 3:21:02	编辑
49	极简Kotlin教程 Wed Jul 19 03:07:36 CST 2017	Kotlin	2017-7-19 3:07:36	编辑
48	Kotlin 极简教程	Z	2017-7-19 3:04:05	编辑
47	极简Kotlin教程 Wed Jul 19 03:03:31 CST 2017	Kotlin	2017-7-19 3:03:32	编辑
46	M	M	2017-7-19 2:57:05	编辑
45	B	b	2017-7-19 2:56:16	编辑
44	A	S	2017-7-19 2:55:34	编辑

点击一篇文章标题，即可进入详情页：

我的博客 文章列表 写文章 关于 Kotlin ▾

Kotlin 简介

K

2017-7-19 3:32:23

Kotlin Kotlin 是一个基于 JVM 的新的编程语言，由 JetBrains 开发。 Kotlin可以编译成Java字节码，也可以编译成JavaScript，方便在没有JVM的设备上运行。 JetBrains，作为目前广受欢迎的Java IDE IntelliJ 的提供商，在 Apache 许可下已经开源其Kotlin 编程语言。 Kotlin已正式成为Android官方支持开发语言。

11.16 添加Markdown支持

我们写技术博客文章，最常用的就是使用Markdown了。我们来为我们的博客添加Markdown的支持。我们使用前端js组件Mditor来支持Markdown的编辑。 Mditor是一个简洁、易于集成、方便扩展、期望舒服的编写 markdown 的编辑器。

11.16.1 引入静态资源

```
<link href="/mditor-master/dist/css/mditor.css" rel="stylesheet">
<script src="/mditor-master/dist/js/mditor.js"></script>
```

11.16.2 初始化Mditor

我们在写文章的页面addArticleView.ftl，初始化Mditor如下：

```
<script>
    $(function () {
        //写文章 mditor
        var mditor = Mditor.fromTextarea(document.getElementById('articleContentEditor'
));

        //是否打开分屏
        mditor.split = true;    //打开
        //是否打开预览
        mditor.preivew = true;    //打开
        //是否全屏
        mditor.fullscreen = false;    //关闭
        //获取或设置编辑器的值
        mditor.on('ready', function () {
            mditor.value = '# ';
        });
        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

另外，我们还使用了代码高亮插件highlight.js 。

这样，写文章的页面对应的textarea区域就变成了支持Markdown在线编辑+预览的功能了：

11.16.3 文章详情显示Markdown渲染

下面我们来使我们的详情页也能支持Markdown的渲染显示。

详情页的视图文件detailArticleView.ftl如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
    <h3>${article.title}</h3>
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <textarea id="articleContentShow" placeholder="<#escape  x as x?html>${article.cont
ent}</#escape>" style="display:
              none"></textarea>
    <div id="article-content" class="markdown-body"></div>

</div>
</body>
</html>
```

这里我们把文章内容放到一个隐藏的textarea的placeholder属性中：

```
<textarea id="articleContentShow" placeholder="<#escape  x as x?html>${article.content}
</#escape>" style="display:
              none"></textarea>
```

注意，这里我们作了字符的转义escape，防止有特殊字符导致页面显示错乱。 然后，我们在js中获取这个内容：

```
<script>
    $(function () {
        // 文章详情 mditor
        var parser = new Mditor.Parser();
        var articleContent = document.getElementById('articleContentShow').placeholder
//直接取原本的字符串。不会被转译，默认html页面中textarea区域text需要escape编码
        articleContent = unescape(articleContent);//unescape解码
        var html = parser.parse(articleContent);

        $('#article-content').append(html);

        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

其中，我们是直接调用的Mditor.Parser()函数来解析Markdown字符文本的。

这样我们的详情页也支持了Markdown的渲染显示了：

JP QL中的SpEL

Koltin

2017-7-19 4:56:46

JP QL中的SpEL

另外我们使用JPA的标准查询（Criteria Query）：

```
SELECT a FROM #{#entityName} a where a.content like %:content%
```

其中的 `#{#entityName}` 是SpEL（Spring表达式语言），用来代替本来实体的名称，而Spring data jpa会自动根据Article实体上对应的默认的 `@Entity class Article`，或者指定 `@Entity(name = "Article") class Article` 自动将实体名称填入 JP QL语句中。

通过把实体类名称抽象出来成为参数，帮助我们解决了项目中很多dao接口的方法除了实体类名称不同，其他操作都相同的问题。

11.17 文章列表分页搜索

为了方便检索我们的博客文章，我们再来给文章列表页面添加分页、搜索、排序等功能。我们使用前端组件DataTables来实现。

提示：更多关于DataTables，可参考： http://www.datatables.club/

11.17.1 引入静态资源文件

```
<link href="/datatables/media/css/jquery.dataTables.css" rel="stylesheet">
<script src="/datatables/media/js/jquery.dataTables.js"></script>
```

11.17.2 给表格加上id

我们给表格加个属性id="articlesDataTable" ：

```
<table id="articlesDataTable" class="table table-responsive table-bordered">
    <thead>
    <th>序号</th>
    <th>标题</th>
    <th>作者</th>
    <th>发表时间</th>
    <th>操作</th>
    </thead>
    <tbody>
    <#-- 使用FTL指令 -->
```

```
        <#list articles as article>
        <tr>
            <td>${article.id}</td>
            <td><a target="_blank" href="detailArticleView?id=${article.id}">${article.
title}</a></td>
            <td>${article.author}</td>
            <td>${article.gmtModified}</td>
            <td><a href="#" target="_blank">编辑</a></td>
        </tr>
        </#list>
        </tbody>
    </table>
```

11.17.3 调用DataTable函数

首先，我们配置一下DataTable的选项：

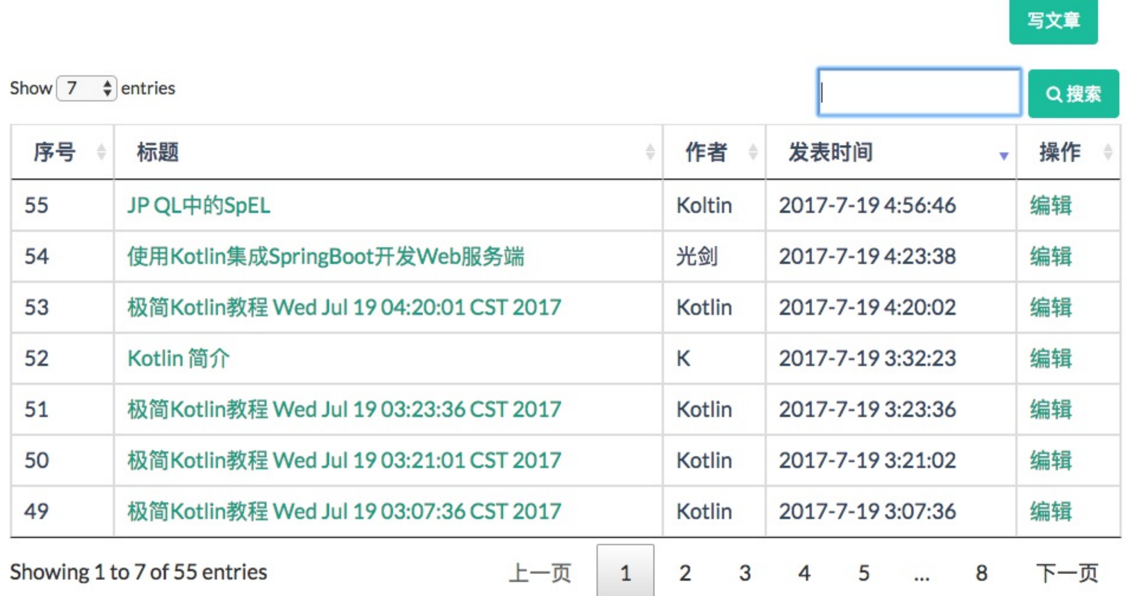
```
    var aLengthMenu = [7, 10, 20, 50, 100, 200]
    var dataTableOptions = {
        "bDestroy": true,
        dom: 'lfrtip',
        "paging": true,
        "lengthChange": true,
        "searching": true,
        "ordering": true,
        "info": true,
        "autoWidth": true,
        "processing": true,
        "stateSave": true,
        responsive: true,
        fixedHeader: false,
        order: [[1, "desc"]],
        "aLengthMenu": aLengthMenu,
        language: {
            "search": "<div style='border-radius:10px;margin-left:auto;margin-right:2px
;width:760px;'>_INPUT_  <span class='btn btn-primary'><span class='fa fa-search'></span
> 搜索</span></div>",

            paginate: {//分页的样式内容
                previous: "上一页",
                next: "下一页",
                first: "第一页",
                last: "最后"
            }
        },
        zeroRecords: "没有内容",//table tbody内容为空时，tbody的内容。
        //下面三者构成了总体的左下角的内容。
```

```
        info: "总计 _TOTAL_ 条,共 _PAGES_ 页，_START_ - _END_ ",//左下角的信息显示，大写的词为关键字。
        infoEmpty: "0条记录",//筛选为空时左下角的显示。
        infoFiltered: ""//筛选之后的左下角筛选提示
    }
```

然后把我们刚才添加了id的表格使用JQuery选择器获取对象，然后直接调用：

```
$('#articlesDataTable').DataTable(dataTableOptions)
```

再次看我们的文章列表页：



已经具备了分页、搜索、排序等功能了。

到这里，我们的这个较为完整的极简博客站点应用基本就开发完成了。

11.18 Spring 5.0对Kotlin的支持

Kotlin 关键性能之一就是能与 Java 库很好地互用。但要在 Spring 中编写惯用的 Kotlin 代码，还需要一段时间的发展。 Spring 对 Java 8 的新支持：函数式 Web 编程、bean 注册 API， 这同样可以在 Kotlin 中使用。

Kotlin 扩展是Kotlin 的编程利器。它能对现有的 API 实现非侵入式的扩展，从而向 Spring中加入 Kotlin 的专有的功能特性。

11.18.1 一种注册 Bean 的新方法

Spring Framework 5.0 引入了一种注册 Bean 的新方法，作为利用 XML 或者 JavaConfig 的 @Configuration 或者 @Bean 的替代方案。简言之就是 Lambda 表达式。

例如用 Java 代码我们会这样写：

```
GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new
    Bar(context.getBean(Foo.class))
);
```

而使用 Kotlin 我们可以将代码写成这样:

```
val context = GenericApplicationContext {
    registerBean<foo>()
    registerBean { Bar(it.getBean<foo>()) }
}
```

11.18.2 Spring Web 函数式 API

Spring 5.0 中的 RouterFunctionDsl 可以让我们使用干净且优雅的 Kotlin 代码来使用崭新的 Spring Web 函数式 API：

```
fun route(request: ServerRequest) = RouterFunctionDsl {
    accept(TEXT_HTML).apply {
            (GET("/user/") or GET("/users/")) { findAllView() }
            GET("/user/{login}") { findViewById() }
    }
    accept(APPLICATION_JSON).apply {
            (GET("/api/user/") or GET("/api/users/")) { findAll() }
            POST("/api/user/") { create() }
            POST("/api/user/{login}") { findOne() }
    }
 } (request)
```

11.18.3 Reactor Kotlin 扩展

Reactor 是 Spring 5.0 中提供的响应式框架。而 reactor-kotlin 项目则是对 Reactor 中使用Kotlin 的支持。目前该项目正在早期阶段。

11.18.4 基于 Kotlin脚本的 Gradle 构建配置

之前我们的 Gradle 构建配置文件都是用Groovy 来编写的，这导致我们基于 Gradle 的 Kotlin 工程还要配置 Groovy 的语法的构建配置文件。

在gradle-script-kotlin 项目中，我们可以直接用 Kotlin 脚本来编写 Gradle 的构建配置文件了。而且 IDE 还为我们提供了在编写配置文件过程中的自动完成功能和重构功能的支持。

11.18.5 基于模板的 Kotlin 脚本

从 4.3 版本开始，Spring 提供了一个 ScriptTemplateView，用于利用支持 JSR-223 的脚本引擎来渲染模板。 Kotlin 1.1-M04 提供了这样的支持，并支持渲染基于 Kotlin 的模板，类似下面这样：

```
import io.spring.demo.User
import io.spring.demo.joinToLine

"""
${include("header", bindings)}
<h1>Title : $title</h1>
<ul>
    ${(users as List<User>).joinToLine{ "<li>User ${it.firstname} ${it.lastname}</li>"
}}
</ul>
${include("footer")}
"""
```

本章小结

本章我们较为细致完整地介绍了使用Kotlin集成SpringBoot进行服务后端开发，并结合简单的前端开发，完成了一个极简的技术博客Web站点。我们可以看到，使用Kotlin结合Spring Boot、Spring MVC、JPA等Java框架的无缝集成，关键是大大简化了我们的代码。同时，在本章最后我们简单介绍了Spring 5.0中对Kotlin的支持诸多新特性，这些新特性令人惊喜。

使用Kotlin编写Spring Boot应用程序越多，我们越觉得这两种技术有着共同的目标，让我们广大程序员可以使用——

- 富有表达性
- 简洁优雅
- 可读

的代码来更高效地编写应用程序，而Spring Framework 5 Kotlin支持将这些技术以更加自然，简单和强大的方式来展现给我们。

未来Spring Framework 5.0 和 Kotlin 结合的开发实践更加值得我们期待。

在下一章中我们将一起学习Kotlin 集成 Gradle 开发的相关内容。

本章项目源码: https://github.com/EasyKotlin/chapter11_kotlin_springboot

第13章 使用 Kotlin 和 Anko 的Android 开发

13.1 什么是 Anko？

Anko (https://github.com/Kotlin/anko) 是一个用 Kotlin 写的Android DSL (Domain-Specific Language)。长久以来，Android视图都是用 XML 来完成布局的。这些 XML可重用性比较差。同时在运行的时候，XML 要转换成 Java 表述，这在一定程度上占用了 CPU 和耗费了电量。

Anko是一个 Kotlin 库, 它使 android 应用程序的开发变得更快、更容易。它使您的代码干净, 易于阅读, 并让您忘记了粗糙的边缘 android sdk 为 java。

Anko由几个部分组成:

模块	功能说明
Anko Commons	使得对 intents, dialogs, logging等操作更加简单的轻量级库
Anko Layouts	快速和类型安全的动态的 android 布局库
Anko SQLite	用于 android sqlite 的查询 dsl 和分析库
Anko Coroutines	基于 kotlinx 协程库

有了Anko 我们就能直接用 Kotlin 在任何的 Activity 、 Fragment 或者 AnkoComponent里来编写视图。

13.2 一个简单Anko视图

这里是一个转换成 Anko 的简单 XML 文件。

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <EditText
        android:id="@+id/todo_title"
        android:layout_width="match_parent"
        android:layout_heigh="wrap_content"
        android:hint="@string/title_hint" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/add_todo" />
</LinearLayout>
```

用 Anko 描述的同样的视图

```
verticalLayout {
    var title = editText {
        id = R.id.todo_title
        hintResource = R.string.title_hint
    }
    button {
        textResource = R.string.add_todo
        onClick { view -> {
                // 可以在这里添加一些处理逻辑
                title.text = "Foo"
            }
        }
    }
}
```

可以看到在button布局中的onClick监听函数中，因为我们是使用 Kotlin代码来设计视图，所以可以直接使用title变量（editText视图对象）。

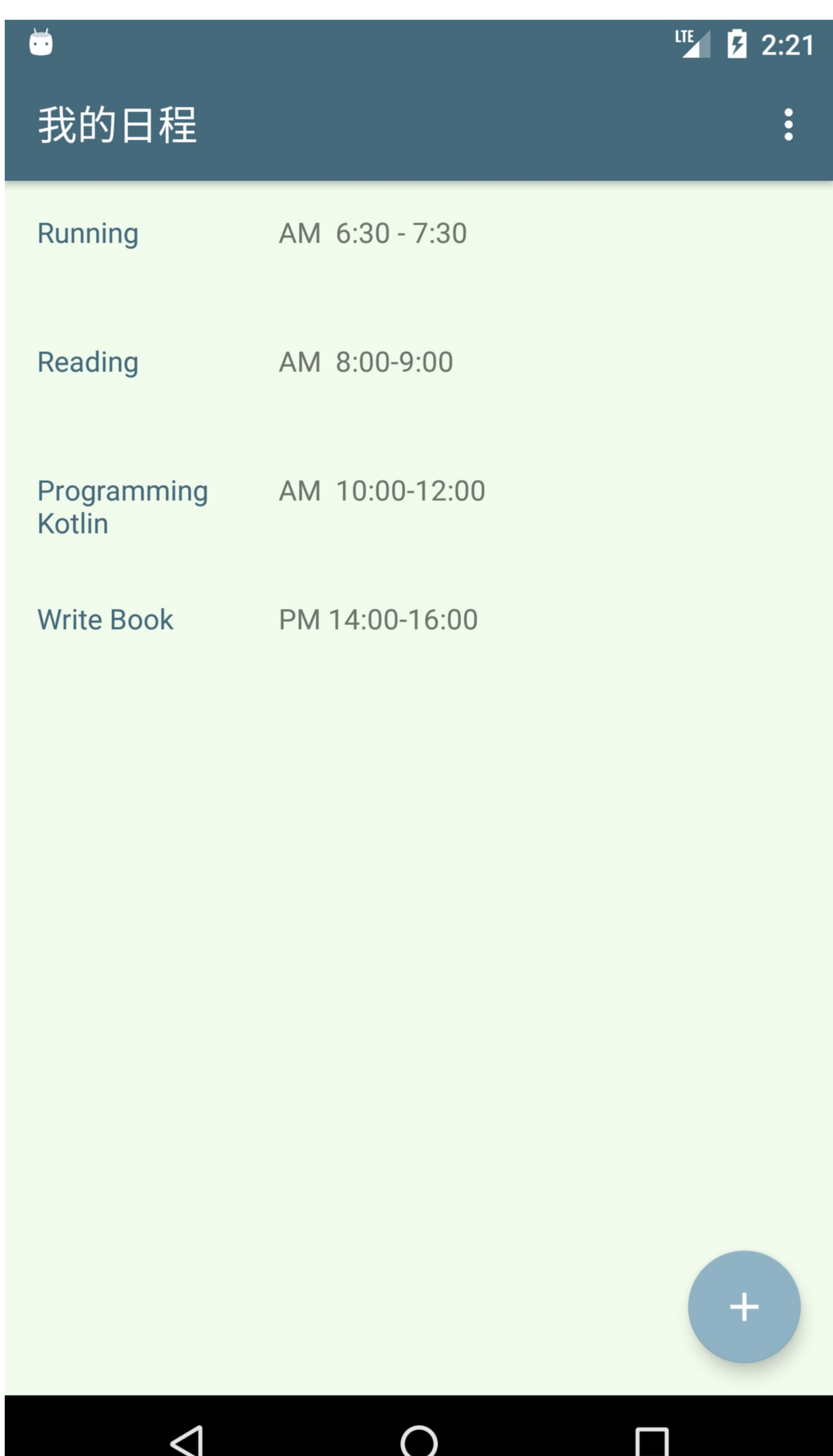
13.3 快速入门实例

下面我们通过一个“我的日程”待办事项应用，来详细介绍使用 Kotlin 混合 Java，使用 Anko 开发的Android 应用的方法。移动端数据库引擎我们使用 Realm，视图绑定使用Butter Knife。

这个应用程序界面如下所示：

2:22
我的日程
标题
待办内容
添加

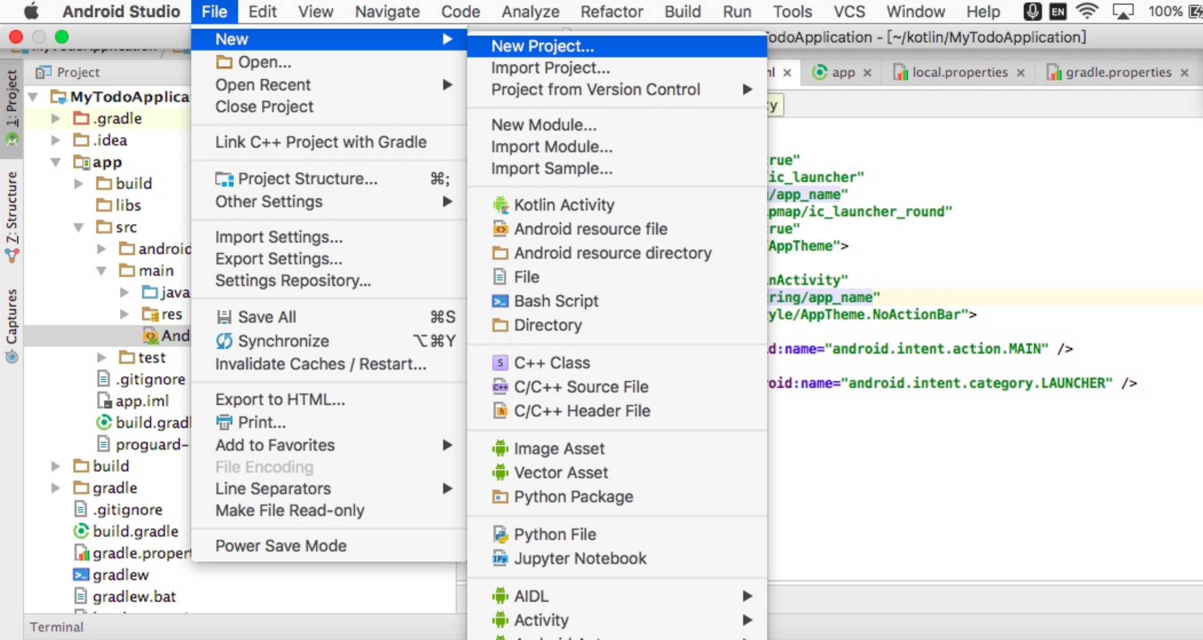
2:22
我的日程
Running
AM 6:30 - 7:30
保存


2:21
我的日程
Running
AM 6:30 - 7:30
Reading
AM 8:00-9:00
Programming Kotlin
AM 10:00-12:00
Write Book
PM 14:00-16:00

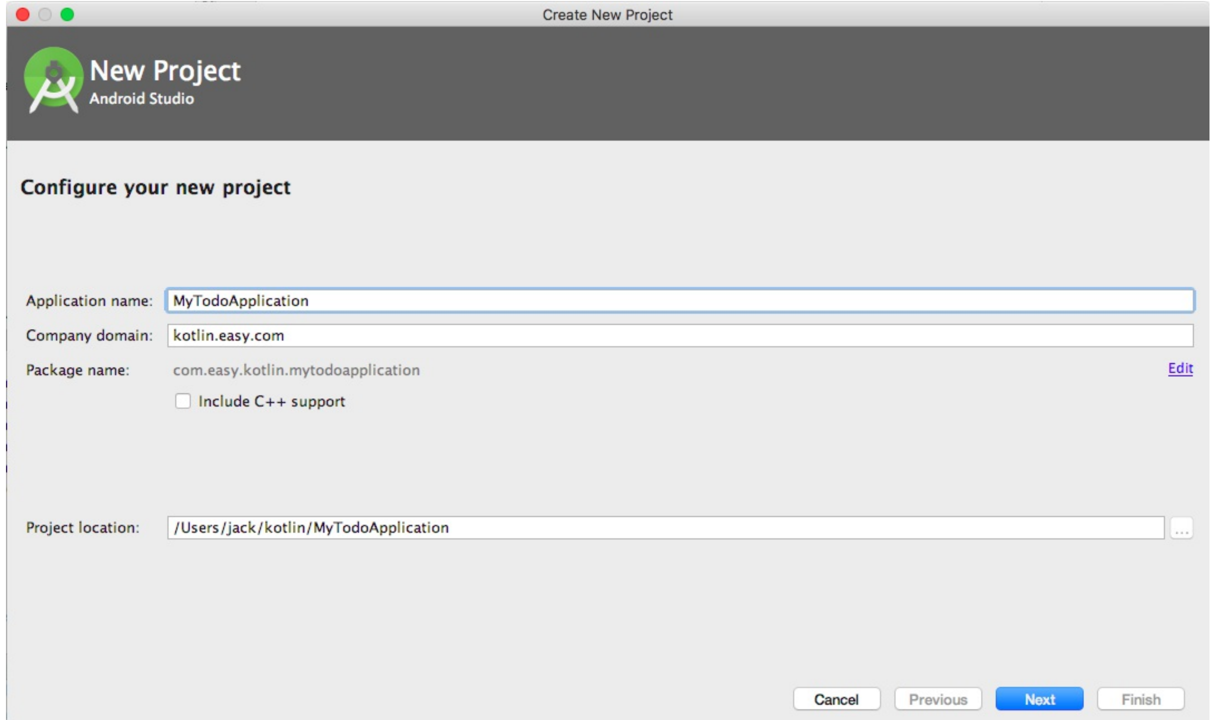
13.4 使用 Android Studio 新建工程

我们首先在 Android Studio 中新建工程，步骤如下：

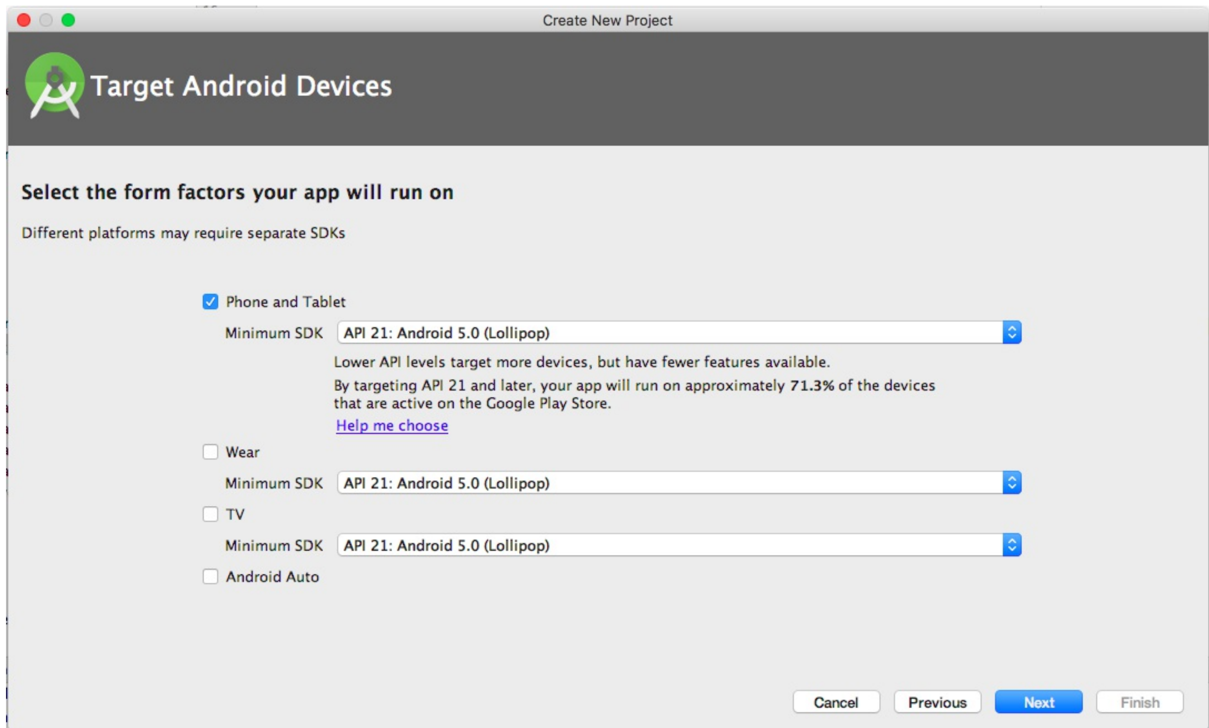
第一步，新建项目



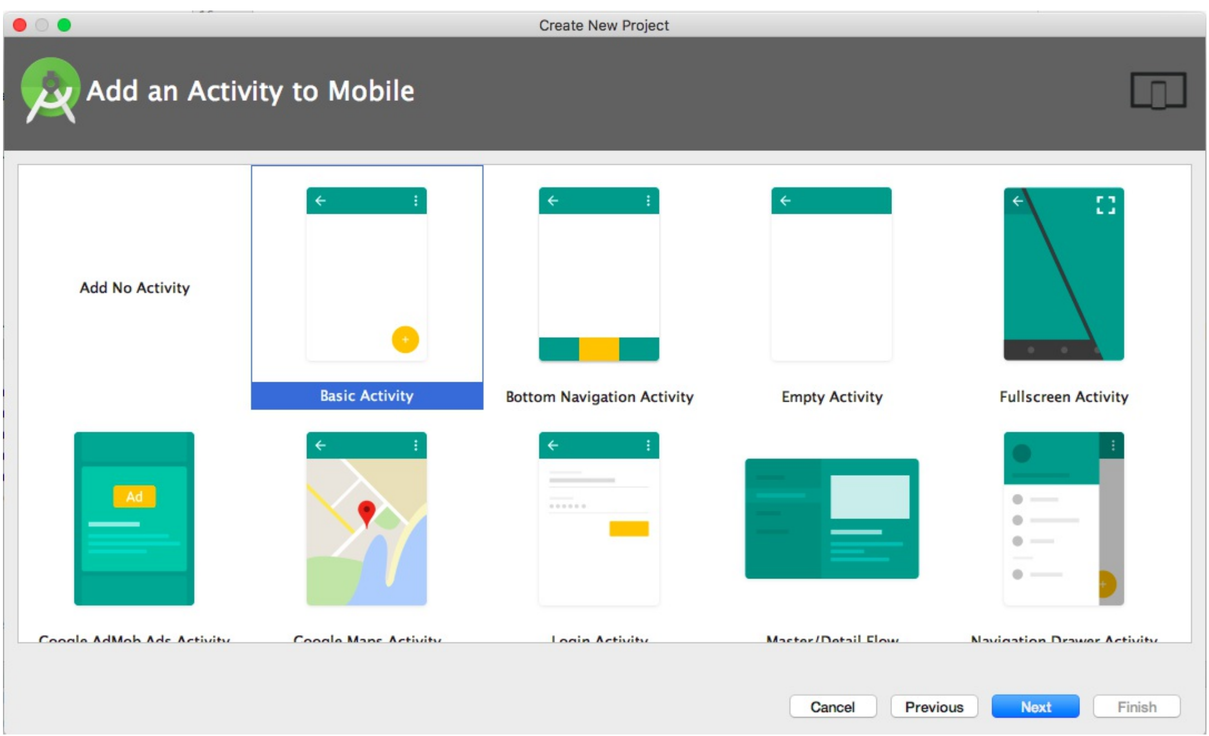
第二步，配置项目基本信息



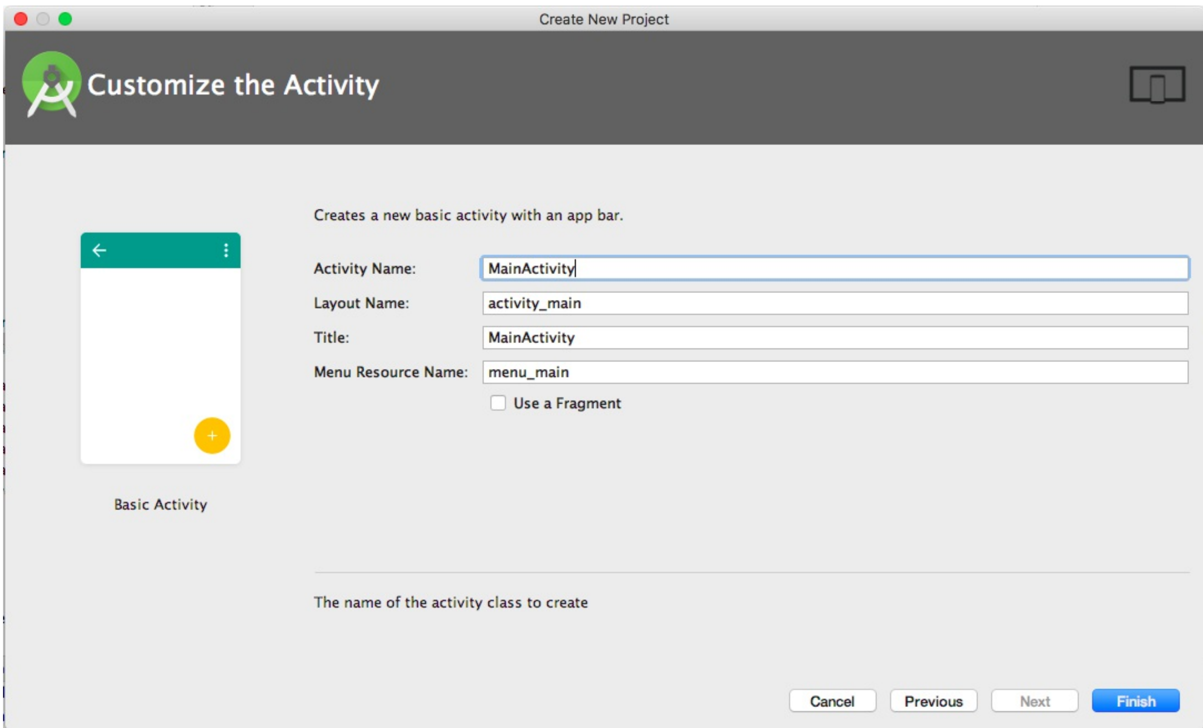
第三步，设置支持设备以及 SDK 版本



第四步，选择 Basic Activity



第五步，使用默认的Activity命名



我们将得到一个标准的 Gradle Android 工程：

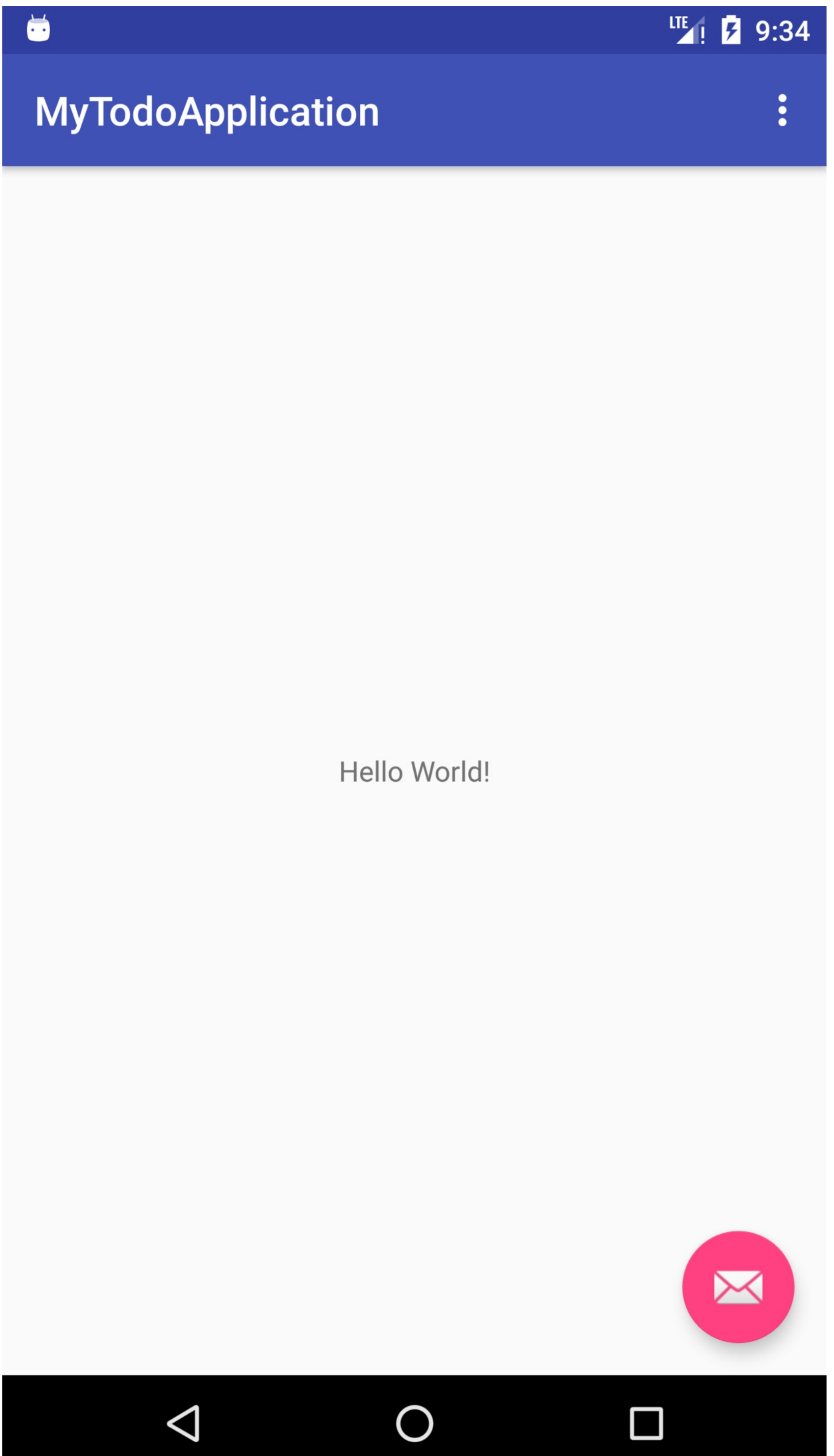
```
MyTodoApplication ~/kotlin/MyTodoApplication
    .gradle
    .idea
    app
        build
        libs
        src
            androidTest
            main
                java
                    com.easy.kotlin.mytodoapplication
                        MainActivity
                res
                    drawable
                    layout
                        activity_main.xml
                        content_main.xml
                    menu
                    mipmap-hdpi
                    mipmap-mdpi
                    mipmap-xhdpi
                    mipmap-xxhdpi
                    mipmap-xxxhdpi
                    values
                AndroidManifest.xml
            test
        .gitignore
        app.iml
        build.gradle
        proguard-rules.pro
    build
    gradle
    .gitignore
    build.gradle
    gradle.properties
    gradlew
    gradlew.bat
    local.properties
    MyTodoApplication.iml
    settings.gradle
External Libraries
```

其中，app 工程 src 目录如下：

```
.
├── androidTest
│   └── java
│       └── com
```

```
│           └── easy
│               └── kotlin
│                   └── mytodoapplication
│                       └── ExampleInstrumentedTest.java
├── main
│   ├── AndroidManifest.xml
│   ├── java
│   │   └── com
│   │       └── easy
│   │           └── kotlin
│   │               └── mytodoapplication
│   │                   └── MainActivity.java
│   └── res
│       ├── drawable
│       ├── layout
│       │   ├── activity_main.xml
│       │   └── content_main.xml
│       ├── menu
│       │   └── menu_main.xml
│       ├── mipmap-hdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-mdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xxhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xxxhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       └── values
│           ├── colors.xml
│           ├── dimens.xml
│           ├── strings.xml
│           └── styles.xml
└── test
    └── java
        └── com
            └── easy
                └── kotlin
                    └── mytodoapplication
                        └── ExampleUnitTest.java

28 directories, 21 files
```

我们直接在Android 模拟器中（也可以选择用真机）运行它，可以看到如下效果：


9:34
MyTodoApplication
Hello World!

13.5 设计UI 界面主题颜色

我们首先把应用名称改成“我的日程” 。在文件MyTodoApplication/app/src/main/res/values/strings.xml中：

```
<resources>
    <string name="app_name">MyTodoApplication</string>
    <string name="action_settings">Settings</string>
</resources>
```

改写成

```
<resources>
    <string name="app_name">我的日程</string>
    <string name="action_settings">设置</string>
</resources>
```

再去colors.xml中，设计主题颜色为：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#f2fced</color>
    <color name="colorPrimaryDark">#456a7c</color>
    <color name="colorAccent">#8fb3c4</color>
</resources>
```

然后到文件MyTodoApplication/app/src/main/res/layout/activity_main.xml中，设置android.support.v7.widget.Toolbar的背景色为

```
android:background="?attr/colorPrimaryDark"
```

配置android.support.design.widget.FloatingActionButton的图标为：

```
app:srcCompat="drawable/ic_content_add"
```

其中，ic_content_add.png图片是我们添加按钮中间的加号 icon。

13.6 配置 Kotlin 与 Anko 依赖

我们默认生成的 app 项目的 Gradle 配置文件build.gradle如下：

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.3"
    defaultConfig {
        applicationId "com.easy.kotlin.mytodoapplication"
        minSdkVersion 21
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rul
es.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    compile 'com.android.support:design:25.3.1'
    testCompile 'junit:junit:4.12'
}
```

下面我们在 app 项目的build.gradle里面加上Kotlin 、Anko 、Realm、Butter Knife 等依赖。

13.6.1 Kotlin依赖

首先，启用插件 `kotlin-android` ：

```
apply plugin: 'kotlin-android'
```

然后，添加构建脚本

```
buildscript {

}
```

我们使用 Kotlin 1.1.3版本。在构建脚本中添加kotlin-gradle-plugin依赖，使用 Kotlin 对应的版本号。

```
buildscript {
    ext.kotlin_version = '1.1.3'
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

在项目依赖里添加 Kotlin 标准库：

```
// Kotlin
compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

13.6.2 添加 Kotlin 源代码目录

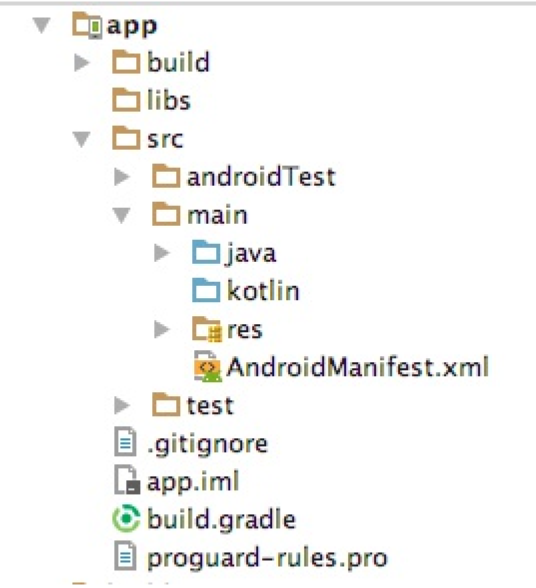
首先，我们在 src/main/下面新建一个 kotlin 目录，来存放 Kotlin源码。然后在 build.gradle 文件里的 `android {}` 配置里面添加Java的编译路径：

```
android {
    ...
    sourceSets {
        // += , 在main中创建kotlin文件夹，用于存放kotlin代码
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

刚添加完毕，src/main/kotlin 还没有变成源码目录的蓝色，这个时候点击下图右上角的 Sync Now :

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.3"

    defaultConfig {
        applicationId "com.easy.kotlin.mytodoapplication"
        minSdkVersion 21
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }

    sourceSets {
        // += 添加Java的编译路径, 在main中创建kotlin文件夹, 用于存放kotlin代码
        main.java.srcDirs += 'src/main/kotlin'
    }


}
```

Gradle 同步完毕，即可看到kotlin 目录已经变成蓝色的源码目录了：



13.6.3 Anko依赖

在项目依赖里添加

```
    // Anko
    compile 'org.jetbrains.anko:anko-sdk15:0.8.2' // sdk19, sdk21, sdk23 are also avail
able
    compile 'org.jetbrains.anko:anko-support-v4:0.8.2' // In case you need support-v4 b
indings
    compile 'org.jetbrains.anko:anko-appcompat-v7:0.8.2' // For appcompat-v7 bindings
```

13.6.4 Realm依赖

```
    compile 'io.realm:realm-android:0.87.1'
    compile 'com.github.thorbenprimke:realm-recyclerview:0.9.12' // 在jitpack.io上
```

其中，Realm是一个轻量级的跨平台移动数据库引。Realm 简单易用，model 设计在代码中，更加易于维护，同时其性能也不错。在Android开发中，它可以替代 SQLite 和 ORM 框架。相比SQLite，Realm更快并且具有很多现代数据库的特性，比如支持JSON，流式api，数据变更通知，以及加密支持。

RecyclerView用于在有限的窗口展现大量的数据，相比ListView、GridView，RecyclerView标准化了ViewHolder，而且更加灵活，可以轻松实现ListView实现不了的样式和功能。我们使用的 `com.github.thorbenprimke:realm-recyclerview` 依赖包在在jitpack.io上， 所以我们还需要配置一下仓库地址：

```
repositories {
    mavenCentral()
    maven { url "https://jitpack.io" }
}
```

提示：realm-recyclerview的 Github 地址是 https://github.com/thorbenprimke/realm-recyclerview

另外， Kotlin使用 Realm 还要加上注解处理的依赖库：

```
    // kotlin使用realm的注解处理依赖库
    kapt "io.realm:realm-annotations:0.87.1"
    kapt "io.realm:realm-annotations-processor:0.87.1"
```

13.6.5 Butter Knife依赖

Butter Knife是基于注解处理方式工作：通过对代码注解自动生成模板代码。我们添加其依赖如下：

```
    // Butter Knife，专门为Android View设计的绑定注解，专业解决各种findViewById
    compile 'com.jakewharton:butterknife:8.7.0'
    annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'
```

Butter Knife主要是用来做Android视图的成员变量和属性的数据绑定。在开发过程中，我们通常要写大量的findViewById和点击事件，像初始view、设置view监听这样简单而重复的操作会显得比较繁琐。而我们有了 Butter Knife，就可以通过使用注解直接生成样板代码。例如，在 Java 中我们可以通过在字段上使用 @BindView 来替代 findViewById 的调用。上面的配置中的 `annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'` 就是来处理这些注解从而生成样板代码的。

```
@Bind(R.id.todo_item_todo_title)
public TextView todoTitle;

@Bind(R.id.todo_item_todo_content)
public TextView todoContent;
```

而在 Kotlin 中使用Butter Knife情况有些不同，需要作额外的配置。

如果在Kotlin中直接使用ButterKnife的注解方式的话，会出现空指针的异常，导致绑定失败。例如

```
@Bind(R.id.todos_recycler_view)
var realmRecyclerView: RealmRecyclerView? = null
```

运行会报错：

```
Caused by: kotlin.KotlinNullPointerException
at com.easy.kotlin.mytodoapplication.TodoListFragment.onResume(TodoListFragment.kt:43)
```

一般情况下，我们使用Kotlin集成 Java 生态的一些框架的时候，像 Spring Boot，JPA，Butter Knife，Realm等，都需要一些额外的插件或者依赖来 “填充缝隙” （例如：all-open, kotterknife, realm-annotations等）， 所谓Kotlin 与 Java 的无缝集成，很多时候并非Java 中怎么用，Kotlin就直接拿过来就怎么用，往往是要再添加一些插件或者额外的配置等。

那么要如何才能在Kotlin的环境中使用ButterKnife呢？

在早些时候，ButterKnife的作者已经帮我们想好解决方案了，那就是——KotterKnife，见名知意。KotterKnife的GitHub地址是：https://github.com/JakeWharton/kotterknife 。这个插件是建立在ButterKnife 7的基础上的。

下面我们配置一下在 Kotlin 中使用 Butter Knife 的依赖库 KotterKnife。

首先在repositories中添加KotterKnife的仓库地址（KotterKnife不在 Maven Center 仓库中，而是在oss.sonatype.org仓库中。这么多仓库，要是哪天能统一用一个就方便多了）。

```
repositories {
    ...
    maven { url 'https://oss.sonatype.org/content/repositories/snapshots/' }
}
```

然后在dependencies里面添加依赖

```
dependencies {
    ...
    compile 'com.jakewharton:butterknife:7.0.1'
    compile 'com.jakewharton:kotterknife:0.1.0-SNAPSHOT'
```

```
}
```

采用这种方式的配置，我们的视图注入代码如下

```
val todoTitle: TextView by bindView(R.id.todo_item_todo_title)
val todoContent: TextView by bindView(R.id.todo_item_todo_content)
```

这样的代码看起来不是那么的优雅，还没有在 Java 中直接使用注解来的简单好看。同时要注意的是，如果使用 kotterknife 0.1.0 + butterknife:7.0.1 ， 同时使用 Java 跟 Kotlin 混合编程的场景中使用 Butter Knife，发现配了KotterKnife 之后的 Java 的注解式写法就失效了。也就是说，如果我们上面添加了KotterKnife的依赖，那么 Java 代码中同时使用 Butter Knife 注解的地方会绑定失败。不过这个问题，在后面的新版本中已经解决。例如在butterknife 8.7.0中，我们可以直接添加下面的依赖项：

```
    compile 'com.jakewharton:butterknife:8.7.0'
    annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'
    kapt 'com.jakewharton:butterknife-compiler:8.7.0'
```

其中， `annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'` 是 Java 的 butterknife注解处理器。 `kapt 'com.jakewharton:butterknife-compiler:8.7.0'` 是 Kotlin 的 butterknife注解处理器（Kotlin Annotation processing tool，kapt）。

这样我们的代码就继续优雅简洁下去了：

```
@BindView(R.id.todo_item_todo_title)
lateinit var todoTitle: TextView
@BindView(R.id.todo_item_todo_content)
lateinit var todoContent: TextView
```

其中，lateinit 修饰符允许声明非空类型，并在对象创建后(构造函数调用后)初始化。 不使用 lateinit 则需要声明可空类型并且有额外的空安全检测操作。

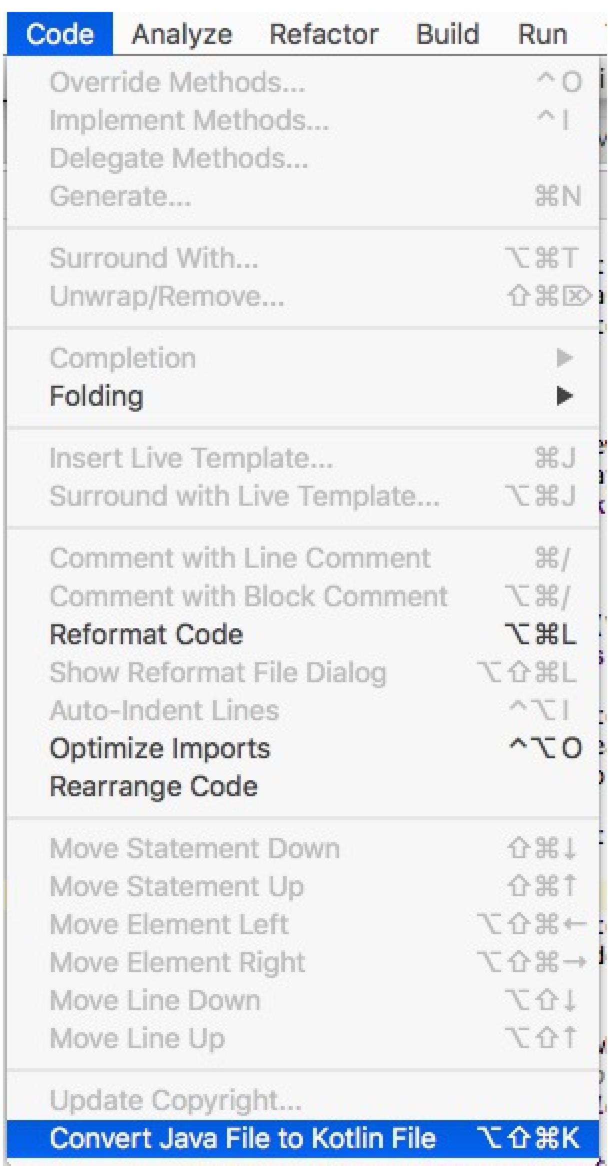
当然，我们使用 Butter Knife 的同时，仍然可以使用原生的 findViewById：

```
class MainActivity : AppCompatActivity() {
    var fab: FloatingActionButton? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val toolbar = findViewById(R.id.toolbar) as Toolbar
        setSupportActionBar(toolbar)

        fab = findViewById(R.id.fab) as FloatingActionButton
```

```
        // 添加日程事件
        fab?.setOnClickListener { _ ->
            ...
            hideFab()
        }
    }

    fun hideFab() {
        fab?.visibility = View.GONE
    }

    fun showFab() {
        fab?.visibility = View.VISIBLE
    }

}
```

13.7 将MainActivity.java 转成 Kotlin 代码

选中默认生成的MainActivity.java， 我们使用 IDEA 的 Code > Convert Java File to Kotlin File :



点击转换，即可看到转换成 Kotlin 的代码：

```
package com.easy.kotlin.mytodoapplication

import android.os.Bundle
import android.support.design.widget.FloatingActionButton
import android.support.design.widget.Snackbar
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.Toolbar
import android.view.Menu
import android.view.MenuItem

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

```
        val toolbar = findViewById(R.id.toolbar) as Toolbar
        setSupportActionBar(toolbar)

        val fab = findViewById(R.id.fab) as FloatingActionButton
        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show()
        }
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        val id = item.itemId

        if (id == R.id.action_settings) {
            return true
        }

        return super.onOptionsItemSelected(item)
    }
}
```

看，这就是Android 开发者，从 Java无缝转到 Kotlin 的过程。

我们把这个MainActivity.kt放到对应的 src/main/kotlin 目录下。首先新建 `package com.easy.kotlin.mytodoapplication` ，直接在 IDEA 中把这个MainActivity.kt 拖到这个package 下面即可。现在我们的工程目录是下面这个样子

```
▼ MyTodoApplication  ~/kotlin/MyTodoApplication
  ▶ .gradle
  ▶ .idea
  ▼ app
    ▶ build
      libs
    ▼ src
      ▶ androidTest
      ▼ main
        ▼ java
          ▼ com
            ▼ easy
              ▼ kotlin
                  mytodoapplication
        ▼ kotlin
          ▼ com
            ▼ easy
              ▼ kotlin
                ▼ mytodoapplication
                    MainActivity
        ▶ res
          AndroidManifest.xml
      ▶ test
      .gitignore
      app.iml
      build.gradle
      proguard-rules.pro
  ▶ build
  ▶ gradle
    .gitignore
    build.gradle
    gradle.properties
    gradlew
    gradlew.bat
    local.properties
    MyTodoApplication.iml
    settings.gradle
▶ External Libraries
```

13.8 在 Kotlin 中使用 Realm

我们需要添加针对 Kotlin 的realm注解处理的库：

```
kapt "io.realm:realm-annotations:0.87.1"
kapt "io.realm:realm-annotations-processor:0.87.1"
```

13.9 添加日程实体类

我们先从领域模型的建立开始。首先我们需要设计一个极简的待办事项的实体类 Todo, 它有主键 id、标题、内容三个字段。

```
@RealmClass
open class Todo : RealmObject() {
    @PrimaryKey
    open var id: String = "-1"
    open var title: String = "日程"
    open var content: String = "事项"
}
```

然后，我们写一个应用程序入口类 `MyTodoApplication` 继承 `android.app.Application` , 在 onCreate() 里面初始化 Realm 数据库的配置。代码如下：

```
class MyTodoApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        val config = RealmConfiguration.Builder(this)
                .name("realm.my_todos")// 库文件名
                .encryptionKey(getKey())  // 加密
                .schemaVersion(1)  // 版本号
                .deleteRealmIfMigrationNeeded()
                .build()

        Realm.setDefaultConfiguration(config)// 设置默认 RealmConfiguration

    }

    /**
     * 64 bits
     * @return
     */
    private fun getKey(): ByteArray {
        return byteArrayOf(0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3,
4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0,
1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1)
    }
}
```

RealmConfiguration.Builder里面如果没有deleteRealmIfMigrationNeeded()的话，会如下报错误：

```
Caused by: io.realm.exceptions.RealmMigrationNeededException:
RealmMigration must be provided ...
```

```
at com.easy.kotlin.mytodoapplication.TodoListFragment.onActivityCreated(TodoListFragmen
t.kt:36)
```

提示： 更多关于 realm 数据库的相关内容可参考 https://realm.io/docs/

13.10 添加日程事件

现在我们点击添加日程的浮层按钮中，添加切换到 “日程添加编辑” `TodoEditFragment` 的逻辑。

```
// 添加日程事件
fab?.setOnClickListener { _ ->
    // Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG).setAct
ion("Action", null).show()
    val todoEditFragment = TodoEditFragment()
    getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.content_main, todoEditFragment, todoEditFragment.javaClass.ge
tSimpleName())
            .addToBackStack(todoEditFragment.javaClass.getSimpleName())
            .commit()

    hideFab()
}
```

13.11 添加日程界面

下面我们来完成这个添加日程的界面。

3:41
我的日程
标题
待办内容
添加

我们采用Fragment来实现。首先新建一个TodoEditFragment继承Fragment() ：

```
class TodoEditFragment : Fragment() {
    val realm: Realm = Realm.getDefaultInstance()
    var todo: Todo? = null

    companion object {
        val TODO_ID_KEY: String = "todo_id_key"

        fun newInstance(id: String): TodoEditFragment {
            var args: Bundle = Bundle()
            args.putString(TODO_ID_KEY, id)
            var todoEditFragment: TodoEditFragment = newInstance()
            todoEditFragment.arguments = args
            return todoEditFragment
        }

        fun newInstance(): TodoEditFragment {
            return TodoEditFragment()
        }
    }

    override fun onCreateView(inflater: LayoutInflater?, container: ViewGroup?, savedIn
stanceState: Bundle?): View? {
        return UI {
            // AnkoContext

            verticalLayout {
                padding = dip(30)
                var title = editText {
                    // editText 视图
                    id = R.id.todo_title
                    hintResource = R.string.title_hint
                }

                var content = editText {
                    id = R.id.todo_content
                    height = 400
                    hintResource = R.string.content_hint
                }
                button {
                    // button 视图
                    id = R.id.todo_add
                    textResource = R.string.add_todo
                    textColor = Color.WHITE
                    setBackgroundColor(Color.DKGRAY)
                    onClick { _ -> createTodoFrom(title, content) }
                }
```

```
            }
        }.view
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)

        if (arguments != null && arguments.containsKey(TODO_ID_KEY)) {

            val todoId = arguments.getString(TODO_ID_KEY)
            todo = realm.where(Todo::class.java).equalTo("id", todoId).findFirst()

            val todoTitle = find<EditText>(R.id.todo_title)
            todoTitle.setText(todo?.title)

            val todoContent = find<EditText>(R.id.todo_content)
            todoContent.setText(todo?.content)

            val add = find<Button>(R.id.todo_add)
            add.setText(R.string.save)
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        realm.close()
    }

    /**
     *  新增待办事项，存入Realm数据库
     *
     *  @param title the title edit text.
     *  @param todoContent the content edit text.
     */
    private fun createTodoFrom(title: EditText, todoContent: EditText) {

        realm.beginTransaction()
        // Either update the edited object or create a new one.
        var t = todo ?: realm.createObject(Todo::class.java)
        t.id = todo?.id ?: UUID.randomUUID().toString()
        t.title = title.text.toString()
        t.content = todoContent.text.toString()

        realm.commitTransaction()
        activity.supportFragmentManager.popBackStack()
    }

}
```

其中，我们重点讲下 Anko 的 UI 布局部分的代码。

```
return UI {
    // AnkoContext

    verticalLayout {
        padding = dip(30)
        var title = editText {
            // editText 视图
            id = R.id.todo_title
            hintResource = R.string.title_hint
        }

        var content = editText {
            id = R.id.todo_content
            height = 400
            hintResource = R.string.content_hint
        }
        button {
            // button 视图
            id = R.id.todo_add
            textResource = R.string.add_todo
            textColor = Color.WHITE
            setBackgroundColor(Color.DKGRAY)
            onClick { _ -> createTodoFrom(title, content) }
        }
    }
}.view
```

我们使用 Kotlin 的代码 Anko DSL 创建了一个垂直方向的线性布局(用代码写配置写布局要比 XML 灵活方便多了)。 其中 UI 函数

```
fun Fragment.UI(init: AnkoContext<Fragment>.() -> Unit) = createAnkoContext(activity, i
nit)
```

是Fragment的扩展函数，它接收一个函数

```
init: AnkoContext<Fragment>.() -> Unit
```

init 的入参是AnkoContext类型。

而verticalLayout函数则是ViewManager的内联扩展函数。

```
inline fun ViewManager.verticalLayout(init: _LinearLayout.() -> Unit): LinearLayout {
    return ankoView(`$$Anko$Factories$CustomViews`.VERTICAL_LAYOUT_FACTORY, init)
}
```

从这些例子我们可以看出 Kotlin 的函数扩展功能相当实用，尤其在 DSL 中用的非常广泛。

在 verticalLayout 代码段内部，创建了三个Android的控件 - 两个 editText 视图和一个 button 视图。这里视图的属性都在一行里面设置好了。

```
padding = dip(30)
var title = editText {
    // editText 视图
    id = R.id.todo_title
    hintResource = R.string.title_hint
}

var content = editText {
    id = R.id.todo_content
    height = 400
    hintResource = R.string.content_hint
}
button {
    // button 视图
    id = R.id.todo_add
    textResource = R.string.add_todo
    textColor = Color.WHITE
    setBackgroundColor(Color.DKGRAY)
    onClick { _ -> createTodoFrom(title, content) }
}
```

这样的视图文件要比 XML 优雅许多了，XML 的配置有时候让人看了就心生烦恼。

我们可以看下按钮控件定义的地方。按钮有一个点击监听函数是定义在视图定义文件里面的。在定义按钮之前，有两个参数 title 和 content 的方法 createTodoFrom 已经被调用了。最后，通过在 AnkoContext （UI 类）上调用 view 属性 `UI {...}.view` 来返回视图。

这里的 ids 被设置为 R.id.。这些 ids 需要手工在一个加做 ids.xml 的文件里创建，这个文件放在 app/src/main/res/values/ids.xml。如果这个文件不存在就创建它。文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="todo_title" type="id" />
    <item name="todo_content" type="id" />
    <item name="todo_add" type="id" />
</resources>
```

这个 ids.xml 文件定义了所有能够被代码引用到的各种视图的 ids。

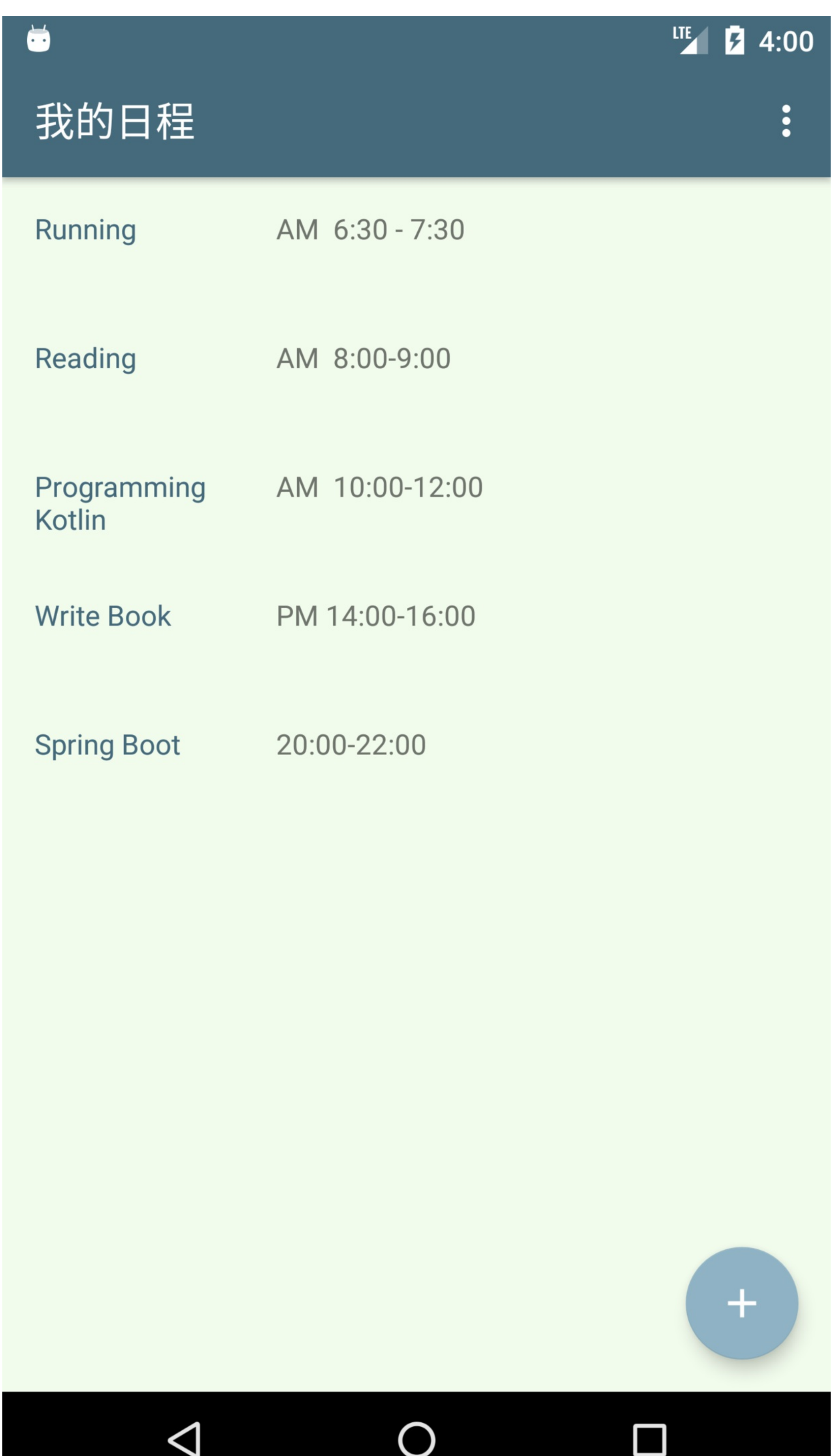
13.12 保存到 Realm 中

新增待办事项，存入Realm数据库：

```
private fun createTodoFrom(title: EditText, todoContent: EditText) {

    realm.beginTransaction()
    // Either update the edited object or create a new one.
    var t = todo ?: realm.createObject(Todo::class.java)
    t.id = todo?.id ?: UUID.randomUUID().toString()
    t.title = title.text.toString()
    t.content = todoContent.text.toString()

    realm.commitTransaction()

    activity.supportFragmentManager.popBackStack()
}
```

13.13 用RecyclerView 来展示待办事项

下面我们来实现这个页面。


4:00
我的日程
Running
AM 6:30 - 7:30
Reading
AM 8:00-9:00
Programming Kotlin
AM 10:00-12:00
Write Book
PM 14:00-16:00
Spring Boot
20:00-22:00

首先，这个是主页面，对应 activity_main.xml 视图, 文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.mytodoapplication.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimaryDark"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@drawable/ic_content_add" />

</android.support.design.widget.CoordinatorLayout>
```

我们的待办事项列表视图是fragment_todos.xml， 文件内容如下：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
```

```
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".TodosFragment"
    tools:showIn="@layout/activity_main">

    <co.moonmonkeylabs.realmrecyclerview.RealmRecyclerView
        android:id="@+id/todos_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:rrvEmptyLayoutId="@layout/empty_view"
        app:rrvIsRefreshable="false"
        app:rrvLayoutType="LinearLayout" />

</RelativeLayout>
```

我们看下 `RealmRecyclerView` 的配置：

配置项	功能说明
app:rrvEmptyLayoutId	当列表为空的时候的显示页面
app:rrvIsRefreshable	是否支持下拉刷新，通过setOnRefreshListener 或 setRefreshing来进行事件处理
app:rrvLayoutType	配置LayoutManager，可选项是：LinearLayout，Grid，LinearLayoutWithHeaders等

下面我们来实现这个TodosFragment 。

首先新建TodosFragment类，继承如下面代码所示：

```
class TodosFragment : Fragment(), TodoAdapter.TodoItemClickListener {
    @BindView(R.id.todos_recycler_view)
    lateinit var realmRecyclerView: RealmRecyclerView

    private var realm: Realm? = null
    ...
}
```

其中，TodoAdapter是继承了RealmBasedRecyclerViewAdapter的适配器类。我们在 TodoAdapter里面定义了一个视图持有类：

```
    inner class ViewHolder(view: View, private val clickListener: TodoItemClickListener
?) :
            RealmViewHolder(view), View.OnClickListener {

        // Bind a field to the view for the specified ID. The view will automatically b
e cast to the field type
```

```
        @BindView(R.id.todo_item_todo_title)
        lateinit var todoTitle: TextView
        // val todoTitle: TextView by bindView(R.id.todo_item_todo_title)
        @BindView(R.id.todo_item_todo_content)
        lateinit var todoContent: TextView
        // val todoContent: TextView by bindView(R.id.todo_item_todo_content)

        init {
            // Bind annotated fields and methods
            ButterKnife.bind(this, view)
            view.setOnClickListener(this)
        }

        override fun onClick(v: View) {
            clickListener?.onClick(v, realmResults[adapterPosition])
        }
    }
```

在ViewHolder初始化 View 的时候，我们使用ButterKnife进行了绑定

```
init {
            // Bind annotated fields and methods
            ButterKnife.bind(this, view)
            view.setOnClickListener(this)
        }
```

待办事项监听器类：

```
    interface TodoItemClickListener {
        fun onClick(caller: View, todo: Todo)
    }
```

我们在TodosFragment中实现这个方法：

```
    override fun onClick(caller: View, todo: Todo) {
        (activity as MainActivity).hideFab()

        val todoEditFragment = TodoEditFragment.newInstance(todo.id)
        activity.supportFragmentManager
                .beginTransaction()
                .replace(R.id.content_main, todoEditFragment, todoEditFragment.javaClas
s.getSimpleName())
                .addToBackStack(todoEditFragment.javaClass.getSimpleName())
                .commit()
    }
```

点击待办事项，把当前的content_main切换成编辑事项 EditFragment的视图。

然后我们在TodoAdapter中重写RealmBasedRecyclerViewAdapter的onCreateRealmViewHolder和onBindRealmViewHolder方法。

```
    override fun onCreateRealmViewHolder(viewGroup: ViewGroup, viewType: Int): ViewHold
er {
        val v = inflater.inflate(R.layout.todo_item, viewGroup, false)
        return ViewHolder(v, clickListener)
    }

    override fun onBindRealmViewHolder(viewHolder: ViewHolder, position: Int) {
        val todo = realmResults[position]

        viewHolder.todoTitle.setText(todo.title)
        viewHolder.todoTitle.fontFeatureSettings = "font-size:12px"
        viewHolder.todoTitle.setTextColor(Color.argb(255, 69, 106, 124))

        viewHolder.todoContent.setText(todo.content)
    }
```

我们在添加（保存）完事项的时候，回到之前的列表页面：

```
private fun createTodoFrom(title: EditText, todoContent: EditText) {
        realm.beginTransaction()
        // Either update the edited object or create a new one.
        var t = todo ?: realm.createObject(Todo::class.java)
        t.id = todo?.id ?: UUID.randomUUID().toString()
        t.title = title.text.toString()
        t.content = todoContent.text.toString()

        realm.commitTransaction()

        activity.supportFragmentManager.popBackStack()
    }
```

当回退到待办事项列表的时候，我们在TodosFragment中的 `onResume()` 函数中来实现数据的更新展示：

```
override fun onResume() {
        super.onResume()
        val todos = realm!!.where(Todo::class.java).findAll()
        Log.i(MY_TAG, "onResume: ${todos}")
        Log.i(MY_TAG, "onResume: realmRecyclerView = ${realmRecyclerView} ")
        val adapter = TodoAdapter(activity, todos, true, true, this)
        realmRecyclerView.setAdapter(adapter)
    }
```

其中， `val todos = realm!!.where(Todo::class.java).findAll()` 是去 Realm 数据库中查询出所有Todo对应的实体记录。

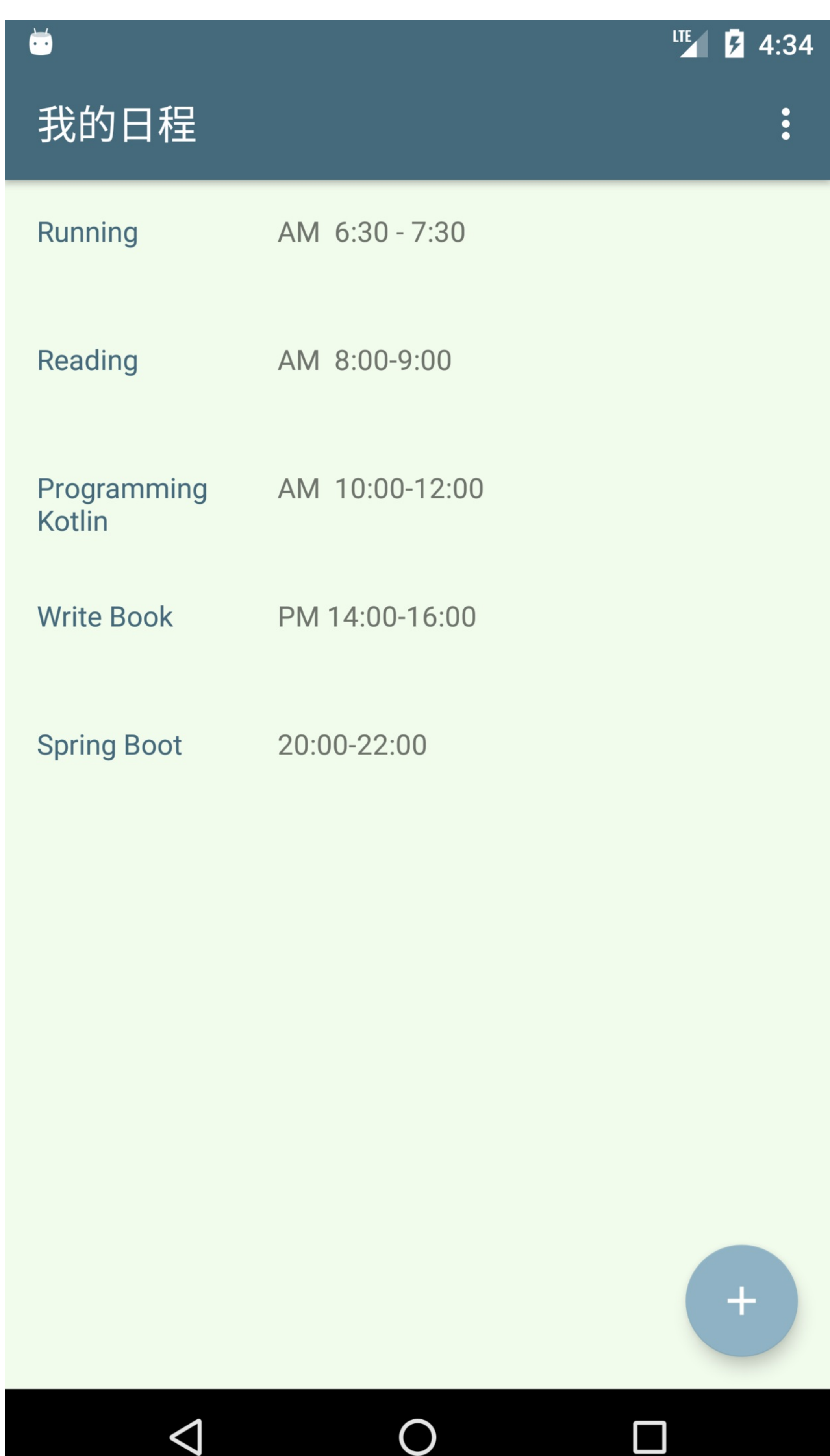
然后，通过适配器 `val adapter = TodoAdapter(activity, todos, true, true, this)` 把数据装配到RecyclerView中 `realmRecyclerView.setAdapter(adapter)` 。

13.14 运行测试

编译安装应用，我们就可以看到如下的界面了，我们可以在里面添加编辑我们的待办事项。

4:34
我的日程
标题
待办内容
添加

4:34
我的日程
Programming Kotlin
AM 10:00-12:00
保存


4:34
我的日程
Running
AM 6:30 - 7:30
Reading
AM 8:00-9:00
Programming Kotlin
AM 10:00-12:00
Write Book
PM 14:00-16:00
Spring Boot
20:00-22:00

本章小结

Android 中经常出现的空引用、API的冗余样板式代码等都是是驱动我们转向 Kotlin 语言的动力。另外，Kotlin 的 Android 视图 DSL Anko帮我们从繁杂的 XML 视图配置文件中解放出来。我们可以像在 Java 中一样方便的使用 Android 开发的流行的库诸如 Butter Knife、Realm、RecyclerView等。当然，我们使用 Kotlin 集成这些库来进行 Andorid 开发，既能够直接使用我们之前的开发库，又能够从 Java 语言、Android API 的限制中出来。这不得不说是一件好事。

下一章我们介绍使用 Kotlin 创建 DSL。

本章工程源码：

https://github.com/EasyKotlin/chapter13_kotlin_android

《Kotlin极简教程》正式上架：

> **点击这里 > 去京东商城购买阅读**
>
> **点击这里 > 去天猫商城购买阅读**

非常感谢您亲爱的读者，大家请多支持！！！有任何问题，欢迎随时与我交流~

第14章 使用 Kotlin DSL

我们在前面的章节中，已经看到了 Kotlin DSL 的强大功能。例如Gradle 的配置文件 build.gradle （Groovy），以及前面我们涉及到的Gradle Script Kotlin（Kotlin）、Anko（Kotlin）等，都是DSL。我们可以看出，使用DSL的编程风格，可以让程序更加简单干净、直观简洁。当然，我们也可以创建自己的 DSL。

本章就让我们一起来学习一下 使用 Kotlin 创建 DSL的相关内容。

我们在上一章中已经看到了在 Android 中使用下面这样的 嵌套DSL 风格的代码来替代 XML 式风格的视图文件

```
UI {
    // AnkoContext

    verticalLayout {
        padding = dip(30)
        var title = editText {
            // editText 视图
            id = R.id.todo_title
            hintResource = R.string.title_hint
        }

        var content = editText {
            id = R.id.todo_content
            height = 400
            hintResource = R.string.content_hint
        }
        button {
            // button 视图
            id = R.id.todo_add
            textResource = R.string.add_todo
            textColor = Color.WHITE
            setBackgroundColor(Color.DKGRAY)
            onClick { _ -> createTodoFrom(title, content) }
        }
    }
}
```

相比 XML 风格的 DSL（XML 本质上讲也是一种 DSL），明显使用原生的编程语言（例如Kotlin）DSL 风格更加简单干净，也更加自由灵活。

Kotlin DSL 的编程风格是怎样的呢？以及其背后实现的原理是怎样的呢？下面就让我一起来探讨一下。

DSL 是什么

DSL（Domain-Specific Language，领域特定语言）指的是专注于特定问题领域的计算机语言（领域专用语言）。不同于通用的计算机语言(GPL)，领域特定语言只用在某些特定的领域。 比如用来显示网页的HTML语言，以及Emacs所使用的Emac LISP语言等。更加典型的例子是Gradle， 它基于Ant 和 Maven，使用基于Groovy的DSL 来声明项目设置，而不是传统的XML。

DSL 简单讲就是对一个特定问题 (受限的表达能力) 的方案模型的更高层次的抽象表达（领域语言），使其更加简单易懂 (容易理解的语义以及清晰的语义模型)。

DSL 只是问题解决方案模型的外部封装，这个模型可能是一个 API 库，也可能是一个完整的框架等等。DSL 提供了思考特定领域问题的模型语言，这使得我们可以更加简单高效地来解决问题。DSL 聚焦一个特定的领域，简单易懂，功能极简但完备。DSL 让我们理解和使用模型更加简易。

提示：关于 DSL 的详细介绍可以参考：《领域特定语言》（Martin Fowler）这本书。

Kotlin 的 DSL 特性支持

扩展（eXtension）特性。

实现一个极简的 DSL

OkHttp是一个成熟且强大的网络库，在Android源码中已经使用OkHttp替代原先的HttpURLConnection。很多著名的框架例如Picasso、Retrofit也使用OkHttp作为底层框架。在这里我对OkHttp做一下简单的封装，其实封装得有点粗暴只是为了演示如何实现dsl。

```
import io.reactivex.BackpressureStrategy
import io.reactivex.Flowable
import io.reactivex.schedulers.Schedulers
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.RequestBody
import okhttp3.Response
import java.util.concurrent.TimeUnit

/**
 * Created by Tony Shen on 2017/6/1.
 */
class RequestWrapper {

    var url:String? = null

    var method:String? = null

    var body: RequestBody? = null
```

```
    var timeout:Long = 10

    internal var _success: (String) -> Unit = { }
    internal var _fail: (Throwable) -> Unit = {}

    fun onSuccess(onSuccess: (String) -> Unit) {
        _success = onSuccess
    }

    fun onFail(onError: (Throwable) -> Unit) {
        _fail = onError
    }
}

fun http(init: RequestWrapper.() -> Unit) {
    val wrap = RequestWrapper()

    wrap.init()

    executeForResult(wrap)
}

private fun executeForResult(wrap:RequestWrapper) {

    Flowable.create<Response>({
        e -> e.onNext(onExecute(wrap))
    }, BackpressureStrategy.BUFFER)
            .subscribeOn(Schedulers.io())
            .subscribe(
                    { resp ->
                        wrap._success(resp.body()!!.string())
                    },

                    { e -> wrap._fail(e) })
}

private fun onExecute(wrap:RequestWrapper): Response? {

    var req:Request? = null
    when(wrap.method) {

        "get","Get","GET" -> req =Request.Builder().url(wrap.url).build()
        "post","Post","POST" -> req = Request.Builder().url(wrap.url).post(wrap.body).b
uild()
        "put","Put","PUT" -> req = Request.Builder().url(wrap.url).put(wrap.body).build
()
        "delete","Delete","DELETE" -> req = Request.Builder().url(wrap.url).delete(wrap
.body).build()
    }
```

```
    val http = OkHttpClient.Builder().connectTimeout(wrap.timeout, TimeUnit.SECONDS).bu
ild()
    val resp = http.newCall(req).execute()
    return resp
}
```

封装完OkHttp之后，看看如何来编写get请求

```
        http {

            url = "http://www.163.com/"

            method = "get"

            onSuccess {
                string -> L.i(string)
            }

            onFail {
                e -> L.i(e.message)
            }
        }
```

是不是很像以前用jquery来写ajax？

post请求也是类似的，只不过多了body

```
        var json = JSONObject()
        json.put("xxx","yyyy")
        ....

        val postBody = RequestBody.create(MediaType.parse("application/json; charset=ut
f-8"),json.toString())

        http {

            url = "https://......"

            method = "post"

            body = postBody

            onSuccess {
                string -> L.json(string)
            }

```

```
            onFail {
                e -> L.i(e.message)
            }
        }
```

使用kotlinx.html DSL 写前端代码

kotlinx.html是可在 Web 应用程序中用于构建 HTML 的 DSL。 它可以作为传统模板系统（例如JSP、FreeMarker等）的替代品。

kotlinx. html 分别提供了kotlinx-html-jvm 和 kotlinx-html-js库的DSL , 用于在 JVM 和浏览器 (或其他 javascript 引擎) 中直接使用 Kotlin 代码来构建 html, 直接解放了原有的 HTML 标签式的前端代码。这样，我们 也可以使用 Kotlin来先传统意义上的 HTML 页面了。 Kotlin Web 编程将会更加简单纯净。

提示： 更多关于kotlinx.html的相关内容可以参考它的 Github 地址：https://github.com/Kotlin/kotlinx.html

要使用 kotlinx.html 首先添加依赖

```
dependencies {
    def kotlinx_html_version = "0.6.3"
    compile "org.jetbrains.kotlinx:kotlinx-html-jvm:${kotlinx_html_version}"
    compile "org.jetbrains.kotlinx:kotlinx-html-js:${kotlinx_html_version}"
    ...
}
```

kotlinx.html 最新版本发布在 https://jcenter.bintray.com/ 仓库上，所以我们添加一下仓库的配置

```
repositories {
    maven { url 'https://jitpack.io' }
    mavenCentral()
    jcenter() // https://jcenter.bintray.com/ 仓库
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}
```

我们来写一个极简百度首页示例。这个页面界面如下图所示

前端 HTML 代码：

```
<!DOCTYPE html>
<html lang=zh-CN>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name=viewport content="width=device-width,initial-scale=1">
    <title>百度一下</title>
    <link href="https://cdn.bootcss.com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
rel="stylesheet">
    <script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>
    <link href="dsl.css" rel="stylesheet">
    <script src="dsl.js"></script>
</head>
<body>
<div class="container">
    <div class="ipad center">
        ![](http://upload-images.jianshu.io/upload_images/1233356-49a0fecdc8bfa9cf.png?
imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)
    </div>

    <form class="form">
        <input id="wd" class="form-control ipad">
        <button id="baiduBtn" type="submit" class="btn btn-primary form-control ipad">
百度一下</button>
    </form>
</div>
</body>
</html>
```

其中，dsl.css文件内容如下

```
.ipad {
    margin: 10px
```

```
}

.center {
    text-align: center;
}
```

dsl.js 文件内容如下

```
$(function () {
    $('#baiduBtn').on('click', function () {
        var wd = $('#wd').val()
        window.open("https://www.baidu.com/s?wd=" + wd)
    })
})
```

上面我们是通常使用的 HTML+JS+CSS 的方式来写前端页面的方法。现在我们把 HTML 部分的代码用Kotlin 的 DSL kotlinx.html 来重新实现一遍。

我们首先新建 Kotlin + Spring Boot 工程，然后直接来写 Kotlin 视图类HelloDSLView，代码如下：

```
package com.easy.kotlin.chapter14_kotlin_dsl.view

import kotlinx.html.*
import kotlinx.html.stream.createHTML
import org.springframework.stereotype.Service

/**
 * Created by jack on 2017/7/22.
 */
@Service
class HelloDSLView {
    fun html(): String {
        return createHTML().html {
            head {
                meta {
                    charset = "utf-8"
                    httpEquiv = "X-UA-Compatible"
                    content = "IE=edge"
                }
                title("百度一下")
                link {
                    href = "https://cdn.bootcss.com/bootstrap/4.0.0-alpha.6/css/bootstr
ap.min.css"
                    rel = "stylesheet"
                }
                script {
                    src = "https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"
```

```
                }
                link {
                    href = "dsl.css"
                    rel = "stylesheet"
                }
                script {
                    src = "dsl.js"
                }
            }

            body {
                div(classes = "container") {
                    div(classes = "ipad center") {
                        img {
                            src = "https://www.baidu.com/img/bd_logo1.png"
                            width = "270"
                            height = "129"
                        }
                    }

                    form(classes = "form") {
                        input(InputType.text, classes = "form-control ipad") {
                            id = "wd"
                        }
                        button(classes = "btn btn-primary form-control ipad") {
                            id = "baiduBtn"
                            type = ButtonType.submit
                            text("百度一下")

                        }
                    }

                }
            }
        }
    }
}
```

相比之下，我们使用 DSL 的风格要比原生 HTML 要简洁优雅。关键是，我们的这个 HTML 是用 Kotlin 写的，这也就意味着，我们的 HTML 代码不再是简单的静态的前端代码了。我们完全可以直接使用后端的接口返回数据来给 HTML 元素赋值，我们也完全具备了（当然是完全超越了）诸如 JSP、Freemarker 这样的视图模板引擎的各种判断、循环等的语法功能，因为我们直接使用的是一门强大的编程语言 Kotlin 来写的 HTML 代码 。

然后，我们就可以直接在控制器层的代码里直接调用我们的 Kotlin 视图代码了：

```
@Controller
```

```
class HelloDSLController {
    @Autowired
    var helloDSLView: HelloDSLView? = null

    @GetMapping("hello")
    fun helloDSL(model: Model): ModelAndView {
        model.addAttribute("hello", helloDSLView?.html())
        return ModelAndView("hello")
    }
}
```

为了简单起见，我们借用一下 Freemarker 来做视图解析引擎，但是它只负责原封不动地来传输我们的 Kotlin 视图代码。hello.ftl 代码如下：

```
${hello}
```

我们的源码目录如下

```
── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── chapter14_kotlin_dsl
    │   │                   ├── Chapter14KotlinDslApplication.kt
    │   │                   ├── controller
    │   │                   │   └── HelloDSLController.kt
    │   │                   └── view
    │   │                       └── HelloDSLView.kt
    │   └── resources
    │       ├── application.properties
    │       ├── banner.txt
    │       ├── static
    │       │   ├── dsl.css
    │       │   ├── dsl.js
    │       │   └── hello.html
    │       └── templates
    │           └── hello.ftl
    └── test
        ├── java
        ├── kotlin
        │   └── com
        │       └── easy
        │           └── kotlin
        │               └── chapter14_kotlin_dsl
```

```
        │                   └── Chapter14KotlinDslApplicationTests.kt
        └── resources
```

然后，启动运行 SpringBoot 应用，浏览器访问 http://127.0.0.1:8888/hello ， 我们可以看到如下输出界面：

这就是 DSL 的精妙之处。我们后面可以尝试使用 kotlinx.html 来写Kotlin 语言的前端代码了。在做 Web 开发的时候，我们通常是使用 HTML + 模板引擎（Velocity、JSP、Freemarker 等）来集成前后端的代码，这让我们有时候感到很尴尬，要学习模板引擎的语法，还得应对 前端HTML代码中凌乱的模板引擎标签、变量等片段代码。

使用 Kotlin DSL 来写 HTML 代码的情况将完全不一样了，我们将重拾前后端集成编码的乐趣（不再是模板引擎套前端 HTML，各种奇怪的 #、<#>、${} 模板语言标签），我们直接把 更加优雅简单的 DSL 风格的HTML 代码搬到了后端，同时HTML中的元素将直接跟后端的数据无缝交互，而完成这些的只是 Kotlin（当然，相应领域的 DSL 基本语义模型还是要学习一下）。

提示：本节项目源码：

https://github.com/EasyKotlin/chapter14_kotlin_dsl

使用KotlinTest写测试代码

本章小结

本章工程源码：

第15章 Kotlin 文件IO操作与多线程

我们在使用 Groovy 的文件 IO 操作的时候，感觉非常便利。同样的Kotlin也有好用的文件 IO 操作的 API。同样的在 Kotlin 中对 Java 的正则表达式功能做了一些实用的扩展。还有 Kotlin 中的多线程主要也是对 Java 的多线程 API 作了一些封装。因为这些 Java 已经有了很多的基础 API，Kotlin 并没有自己再去重复实现，而是在 Java 的基础上进行了实用的功能扩展。

本章我们就来介绍Kotlin 文件 IO 操作、正则表达式以及多线程相关的内容。

15.1 Kotlin IO 简介

Kotlin的IO操作都在kotlin.io包下。Kotlin的原则就是Java已经有的，好用的就直接使用，没有的或者不好用的，就在原有类的基础上进行封装扩展，例如Kotlin 就给 File 类写了扩展函数。这跟Groovy的扩展API 的思想是一样的。

15.2 终端 IO

Java 超长的输出语句 System.out.println() 居然延续到了现在！同样的工作在C++里面只需要简单的cout<< 就可以完成。当然，如果需要的话，我们可以在工程中直接封装 System.out.println() 为简单的打印方法。

在Kotlin里面很简单，只需要使用println或者print这两个全局函数即可，我们不再需要冗长的前缀。当然如果我们很怀旧，就是想用 System.out.println()，Kotlin 依然支持直接这么使用（与 Java 无缝互操作）。

```
>>> System.out.println("K")
K
>>> println("K")
K
```

这里的 println 函数Kotlin实现如下

```
@kotlin.internal.InlineOnly
public inline fun println(message: Any?) {
    System.out.println(message)
}
```

当然，Kotlin 也只是在 System.out.println() 的基础上进行了封装。

从终端读取数据也很简单，最基本的方法就是全局函数readLine，它直接从终端读取一行作为字符串。如果需要更进一步的处理，可以使用Kotlin提供的各种字符串处理函数来处理和转换字符串。

Kotlin 的封装终端IO 的类在 stdlib/src/kotlin/io/Console.kt 源文件中。

15.3 文件 IO 操作

Kotlin为java.io.File提供了大量好用的扩展函数，这些扩展函数主要在下面三个源文件中：

kotlin/io/files/FileTreeWalk.kt
kotlin/io/files/Utils.kt
kotlin/io/FileReadWrite.kt

同时，Kotlin 也针对InputStream、OutputStream和 Reader 等都做了简单的扩展。它们主要在下面的两个源文件中：

kotlin/io/IOStreams.kt
kotlin/io/ReadWrite.kt

Koltin 的序列化直接采用的 Java 的序列化类的类型别名：

```
internal typealias Serializable = java.io.Serializable
```

下面我们来简单介绍一下 Kotlin 文件读写操作。

15.3.1 读文件

读取文件全部内容

我们如果简单读取一个文件，可以使用readText()方法，它直接返回整个文件内容。代码示例如下

```
/**
 * 获取文件全部内容字符串
 * @param filename
 */
fun getFileContent(filename: String): String {
    val f = File(filename)
    return f.readText(Charset.forName("UTF-8"))
}
```

我们直接使用 File 对象来调用 readText 函数即可获得该文件的全部内容，它返回一个字符串。如果指定字符编码，可以通过传入参数Charset来指定，默认是UTF-8编码。

如果我们想要获得文件每行的内容，可以简单通过 `split("\n")` 来获得一个每行内容的数组。

获取文件每行的内容

我们也可以直接调用 Kotlin 封装好的readLines函数，获得文件每行的内容。readLines函数返回一个持有每行内容的 List。

```
/**
 * 获取文件每一行内容，存入一个 List 中
 * @param filename
 */
fun getFileLines(filename: String): List<String> {
    return File(filename).readLines(Charset.forName("UTF-8"))
}
```

直接操作字节数组

我们如果希望直接操作文件的字节数组，可以使用readBytes()。如果想使用传统的Java方式，在Kotlin 中你也可以像 Groovy 一样自如使用。

```
    //读取为bytes数组
    val bytes: ByteArray = f.readBytes()
    println(bytes.joinToString(separator = " "))

    //直接像 Java 中的那样处理Reader或InputStream
    val reader: Reader = f.reader()
    val inputStream: InputStream = f.inputStream()
    val bufferedReader: BufferedReader = f.bufferedReader()
}
```

15.3.2 写文件

和读文件类似，写入文件也很简单。我们可以写入字符串，也可以写入字节流。还可以直接使用Java的 Writer 或者 OutputStream。

覆盖写文件

```
fun writeFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.writeText(text, Charset.defaultCharset())
}
```

末尾追加写文件

```
fun appendFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
```

```
    }
    f.appendText(text, Charset.defaultCharset())
}
```

15.4 遍历文件树

和Groovy一样，Kotlin也提供了方便的功能来遍历文件树。遍历文件树需要调用扩展方法walk()。它会返回一个FileTreeWalk对象，它有一些方法用于设置遍历方向和深度，详情参见FileTreeWalk API 文档说明。

提示：FileTreeWalk API 文档链接 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/-file-tree-walk/

下面的例子遍历了指定文件夹下的所有文件。

```
fun traverseFileTree(filename: String) {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    fileTreeWalk.iterator().forEach { println(it.absolutePath) }
}
```

测试代码：

```
@Test fun testTraverseFileTree() {
    KFileUtil.traverseFileTree(".")
}
```

运行上面的测试代码，它将输出当前目录下的所有子目录及其文件。

我们还可以遍历当前文件下面所有子目录文件，存入一个 Iterator 中

```
fun getFileIterator(filename: String): Iterator<File> {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    return fileTreeWalk.iterator()
}
```

我们遍历当前文件下面所有子目录文件，还可以根据条件过滤，并把结果存入一个 Sequence 中

```
fun getFileSequenceBy(filename: String, p: (File) -> Boolean): Sequence<File> {
    val f = File(filename)
    return f.walk().filter(p)
}
```

测试代码：

```
    @Test fun testGetFileSequenceBy() {
        val fileSequence1 = KFileUtil.getFileSequenceBy(".", {
            it.isDirectory
        })
        fileSequence1.forEach { println("fileSequence1: ${it.absoluteFile} ") }

        val fileSequence2 = KFileUtil.getFileSequenceBy(".", {
            it.isFile
        })
        fileSequence2.forEach { println("fileSequence2: ${it.absoluteFile} ") }

        val fileSequence3 = KFileUtil.getFileSequenceBy(".", {
            it.extension == "kt"
        })
        fileSequence3.forEach { println("fileSequence3: ${it.absoluteFile} ") }
    }
```

在工程中运行上面的测试代码，它将会有类似下面的输出：

```
...
...

fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/easy/kotlin/f
ileio/KFileUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/easy/kotlin/f
ileio/KNetUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/easy/kotlin/f
ileio/KShellUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/test/kotlin/com/easy/kotlin/f
ileio/KFileUtilTest.kt
```

15.5 网络IO操作

Kotlin为java.net.URL增加了两个扩展方法，readBytes和readText。我们可以方便的使用这两个方法配合正则表达式实现网络爬虫的功能。

下面我们简单写几个函数实例。

根据 url 获取该 url 的响应 HTML函数

```

fun getUrlContent(url: String): String {
    return URL(url).readText(Charset.defaultCharset())
}
```

根据 url 获取该 url 响应比特数组函数

```
fun getUrlBytes(url: String): ByteArray {
    return URL(url).readBytes()
}
```

把 url 响应字节数组写入文件

```
fun writeUrlBytesTo(filename: String, url: String) {
    val bytes = URL(url).readBytes()
    File(filename).writeBytes(bytes)
}
```

下面这个例子简单的获取了百度首页的源代码。

```
getUrlContent("https://www.baidu.com")
```

下面这个例子根据 url 来获取一张图片的比特流，然后调用readBytes()方法读取到字节流并写入文件。

```
writeUrlBytesTo("图片.jpg", "http://n.sinaimg.cn/default/4_img/uplaod/3933d981/20170622
/2fIE-fyhfxph6601959.jpg")
```

在项目相应文件夹下我们可以看到下载好的 “图片.jpg” 。

15.6 kotlin.io标准库

Kotlin 的 io 库主要是扩展 Java 的 io 库。下面我们简单举几个例子。

appendBytes

追加字节数组到该文件中

方法签名：

```
fun File.appendBytes(array: ByteArray)
```

appendText

追加文本到该文件中

方法签名：

```

```

```
fun File.appendText(
    text: String,
    charset: Charset = Charsets.UTF_8)
```

`bufferedReader`

获取该文件的BufferedReader

方法签名：

```
fun File.bufferedReader(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedReader
```

`bufferedWriter`

获取该文件的BufferedWriter

方法签名：

```
fun File.bufferedWriter(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedWriter
```

`copyRecursively`

复制该文件或者递归复制该目录及其所有子文件到指定路径，如果指定路径下的文件不存在，会自动创建。

方法签名：

```
fun File.copyRecursively(
    target: File,
    overwrite: Boolean = false, // 是否覆盖。true：覆盖之前先删除原来的文件
    onError: (File, IOException) -> OnErrorAction = { _, exception -> throw exception }
): Boolean
```

提示： Kotlin 对 File 的扩展函数 API 文档https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/java.io.-file/index.html
关于 kotlin.io 下面的API文档在这里 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/index.html

15.7 执行Shell命令行

我们使用 Groovy 的文件 IO 操作感觉非常好用，例如

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4)
class ShellExecuteDemoTest {
    @Test
    def void testShellExecute() {
        def p = "ls -R".execute()
        def output = p.inputStream.text
        println(output)
        def fname = "我图.url"
        def f = new File(fname)
        def lines = f.readLines()
        lines.forEach({
            println(it)
        })
        println(f.text)
    }
}
```

Kotlin 中的文件 IO，网络 IO 操作跟 Groovy一样简单。

另外，从上面的代码中我们看到使用 Groovy 执行终端命令非常简单：

```
def p = "ls -R".execute()
def output = p.inputStream.text
```

在 Kotlin 中，目前还没有对 String 类和 Process 扩展这样的函数。其实扩展这样的函数非常简单。我们完全可以自己扩展。

首先，我们来扩展 String 的 execute() 函数。

```
fun String.execute(): Process {
    val runtime = Runtime.getRuntime()
    return runtime.exec(this)
}
```

然后，我们来给 Process 类扩展一个 text函数。

```
fun Process.text(): String {
    var output = ""
```

```
    //    输出 Shell 执行的结果
    val inputStream = this.inputStream
    val isr = InputStreamReader(inputStream)
    val reader = BufferedReader(isr)
    var line: String? = ""
    while (line != null) {
        line = reader.readLine()
        output += line + "\n"
    }
    return output
}
```

完成了上面两个简单的扩展函数之后，我们就可以在下面的测试代码中，可以像 Groovy 一样执行终端命令了：

```
val p = "ls -al".execute()

val exitCode = p.waitFor()
val text = p.text()

println(exitCode)
println(text)
```

实际上，通过之前的很多实例的学习，我们可以看出 Kotlin 的扩展函数相当实用。Kotlin 语言本身 API 也大量使用了扩展功能。

15.8 正则表达式

我们在 Kotlin 中除了仍然可以使用 Java中的 Pattern，Matcher 等类之外，Kotlin 还提供了一个正则表达式类 kotlin/text/regex/Regex.kt ，我们通过 Regex 的构造函数来创建一个正则表达式。

15.8.1 构造 Regex 表达式

使用Regex构造函数

```
>>> val r1 = Regex("[a-z]+")
>>> val r2 = Regex("[a-z]+", RegexOption.IGNORE_CASE)
```

其中的匹配选项 RegexOption 是直接使用的 Java 类 Pattern中的正则匹配选项。

使用 String 的 toRegex 扩展函数

```
>>> val r3 = "[A-Z]+".toRegex()
```

15.8.2 Regex 函数

Regex 里面提供了丰富的简单而实用的函数，如下表所示

函数名称	功能说明
matches(input: CharSequence): Boolean	输入字符串全部匹配
containsMatchIn(input: CharSequence): Boolean	输入字符串至少有一个匹配
matchEntire(input: CharSequence): MatchResult?	输入字符串全部匹配，返回一个匹配结果对象
replace(input: CharSequence, replacement: String): String	把输入字符串中匹配的部分替换成replacement的内容
replace(input: CharSequence, transform: (MatchResult) -> CharSequence): String	把输入字符串中匹配到的值，用函数transform映射之后的新值替换
find(input: CharSequence, startIndex: Int = 0): MatchResult?	返回输入字符串中第一个匹配的值
findAll(input: CharSequence, startIndex: Int = 0): Sequence	返回输入字符串中所有匹配的值MatchResult的序列

下面我们分别就上面的函数给出简单实例。

`matches`

输入字符串全部匹配正则表达式返回 true , 否则返回 false。

```
>>> val r1 = Regex("[a-z]+")
>>> r1.matches("ABCzxc")
false
>>>

>>> val r2 = Regex("[a-z]+", RegexOption.IGNORE_CASE)
>>> r2.matches("ABCzxc")
true

>>> val r3 = "[A-Z]+".toRegex()
>>> r3.matches("GGMM")
true
```

`containsMatchIn`

输入字符串中至少有一个匹配就返回true，没有一个匹配就返回false。

```
>>> val re = Regex("[0-9]+")
>>> re.containsMatchIn("012Abc")
true
>>> re.containsMatchIn("Abc")
false
```

`matchEntire`

输入字符串全部匹配正则表达式返回 一个MatcherMatchResult对象，否则返回 null。

```
>>> val re = Regex("[0-9]+")
>>> re.matchEntire("1234567890")
kotlin.text.MatcherMatchResult@34d713a2
>>> re.matchEntire("1234567890!")
null
```

我们可以访问MatcherMatchResult的value熟悉来获得匹配的值。

```
>>> re.matchEntire("1234567890")?.value
1234567890
```

由于 matchEntire 函数的返回是MatchResult? 可空对象，所以这里我们使用了安全调用符号 `?.` 。

`replace(input: CharSequence, replacement: String): String`

把输入字符串中匹配的部分替换成replacement的内容。

```
>>> val re = Regex("[0-9]+")
>>> re.replace("12345XYZ","abcd")
abcdXYZ
```

我们可以看到，"12345XYZ"中 `12345` 是匹配正则表达式 `[0-9]+` 的内容，它被替换成了 `abcd` 。

`replace(input: CharSequence, transform: (MatchResult) -> CharSequence): String`

把输入字符串中匹配到的值，用函数 transform映射之后的新值替换。

```
>>> val re = Regex("[0-9]+")
>>> re.replace("9XYZ8", { (it.value.toInt() * it.value.toInt()).toString() })
81XYZ64
```

我们可以看到， `9XYZ8` 中数字9和8是匹配正则表达式 `[0-9]+` 的内容，它们分别被transform函数映射 `(it.value.toInt() * it.value.toInt()).toString()` 的新值 81 和 64 替换。

`find`

返回输入字符串中第一个匹配的MatcherMatchResult对象。

```
>>> val re = Regex("[0-9]+")
>>> re.find("123XYZ987abcd7777")
kotlin.text.MatcherMatchResult@4d4436d0
>>> re.find("123XYZ987abcd7777")?.value
123
```

`findAll`

返回输入字符串中所有匹配的值的MatchResult的序列。

```
>>> val re = Regex("[0-9]+")
>>> re.findAll("123XYZ987abcd7777")
kotlin.sequences.GeneratorSequence@f245bdd
```

我们可以通过 forEach 循环遍历所以匹配的值

```
>>> re.findAll("123XYZ987abcd7777").forEach{println(it.value)}
123
987
7777
```

15.8.3 使用 Java 正则表达式类

除了上面 Kotlin 提供的函数之外，我们在 Kotlin 中仍然可以使用 Java 的正则表达式的 API。

```
val re = Regex("[0-9]+")
val p = re.toPattern()
val m = p.matcher("888ABC999")
while (m.find()) {
    val d = m.group()
    println(d)
}
```

上面的代码运行输出：

```
888
999
```

15.9 Kotlin 的多线程

Kotlin中没有synchronized关键字。 Kotlin中没有volatile关键字。 Kotlin的Any类似于Java的Object，但是没有wait()，notify()和notifyAll() 方法。

那么并发如何在Kotlin中工作呢？放心，Kotlin 既然是站在 Java 的肩膀上，当然少不了对多线程编程的支持——Kotlin通过封装 Java 中的线程类，简化了我们的编码。同时我们也可以使用一些特定的注解，直接使用 Java 中的同步关键字等。下面我们简单介绍一下使用Kotlin 进行多线程编程的相关内容。

15.9.1 创建线程

我们在 Java中通常有两种方法在Java中创建线程：

- 扩展Thread类
- 或者实例化它并通过构造函数传递一个Runnable

因为我们可以很容易地在Kotlin中使用Java类，这两个方式都可以使用。

使用对象表达式创建

```
object : Thread() {
    override fun run() {
        Thread.sleep(3000)
        println("A 使用 Thread 对象表达式: ${Thread.currentThread()}")
    }
}.start()
```

此代码使用Kotlin的对象表达式创建一个匿名类并覆盖run()方法。

使用 Lambda 表达式

下面是如何将一个Runnable传递给一个新创建的Thread实例：

```
Thread({
    Thread.sleep(2000)
    println("B 使用 Lambda 表达式: ${Thread.currentThread()}")
}).start()
```

我们在这里看不到Runnable，在Kotlin中可以很方便的直接使用上面的Lambda表达式来表达。

还有更简单的方法吗？ 且看下文解说。

使用 Kotlin 封装的 thread 函数

例如，我们写了下面一段线程的代码

```
val t = Thread({
    Thread.sleep(2000)
    println("C 使用 Lambda 表达式:${Thread.currentThread()}")
})
```

```
    t.isDaemon = false
    t.name = "CThread"
    t.priority = 3
    t.start()
```

后面的四行可以说是样板化的代码。在 Kotlin 中把这样的操作封装简化了。

```
    thread(start = true, isDaemon = false, name = "DThread", priority = 3) {
        Thread.sleep(1000)
        println("D 使用 Kotlin 封装的函数 thread(): ${Thread.currentThread()}")
    }
```

这样的代码显得更加精简整洁了。事实上，thread()函数就是对我们编程实践中经常用到的样板化的代码进行了抽象封装，它的实现如下：

```
public fun thread(start: Boolean = true, isDaemon: Boolean = false, contextClassLoader:
 ClassLoader? = null, name: String? = null, priority: Int = -1, block: () -> Unit): Thr
ead {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon)
        thread.isDaemon = true
    if (priority > 0)
        thread.priority = priority
    if (name != null)
        thread.name = name
    if (contextClassLoader != null)
        thread.contextClassLoader = contextClassLoader
    if (start)
        thread.start()
    return thread
}
```

这只是一个非常方便的包装函数，简单实用。从上面的例子我们可以看出，Kotlin 通过扩展 Java 的线程 API，简化了样板代码。

15.9.2 同步方法和块

synchronized不是Kotlin中的关键字，它替换为@Synchronized 注解。 Kotlin中的同步方法的声明将如下所示：

```
    @Synchronized fun appendFile(text: String, destFile: String) {
```

```
        val f = File(destFile)
        if (!f.exists()) {
            f.createNewFile()
        }
        f.appendText(text, Charset.defaultCharset())
    }
```

@Synchronized 注解与 Java中的 synchronized 具有相同的效果：它会将JVM方法标记为同步。 对于同步块，我们使用synchronized() 函数，它使用锁作为参数：

```
    fun appendFileSync(text: String, destFile: String) {
        val f = File(destFile)
        if (!f.exists()) {
            f.createNewFile()
        }

        synchronized(this){
            f.appendText(text, Charset.defaultCharset())
        }
    }
```

跟 Java 基本一样。

15.9.3 可变字段

同样的，Kotlin没有 volatile 关键字，但是有@Volatile注解。

```
@Volatile private var running = false
fun start() {
    running = true
    thread(start = true) {
        while (running) {
            println("Still running: ${Thread.currentThread()}")
        }
    }
}

fun stop() {
    running = false
    println("Stopped: ${Thread.currentThread()}")
}
```

@Volatile会将JVM备份字段标记为volatile。

当然，在 Kotlin 中我们有更好用的协程并发库。在代码工程实践中，我们可以根据实际情况自由选择。

本章小结

Kotlin 是一门工程实践性很强的语言，从本章介绍的文件IO、正则表达式以及多线程等内容中，我们可以领会到 Kotlin 的基本原则：充分使用已有的 Java 生态库，在此基础之上进行更加简单实用的扩展，大大提升程序员们的生产力。从中我们也体会到了Kotlin 编程中的极简理念——不断地抽象、封装、扩展，使之更加简单实用。

本章示例代码：https://github.com/EasyKotlin/chapter15_file_io

另外，笔者综合了本章的内容，使用 SpringBoot + Kotlin 写了一个简单的图片爬虫 Web 应用，感兴趣的读者可参考源码：https://github.com/EasyKotlin/chatper15_net_io_img_crawler

在下一章，也我们的最后一章中，让我们脱离 JVM，直接使用 Kotlin Native 来开发一个直接编译成机器码运行的 Kotlin 应用程序。

《Kotlin极简教程》正式上架：

> **点击这里 > 去京东商城购买阅读**
>
> **点击这里 > 去天猫商城购买阅读**

非常感谢您亲爱的读者，大家请多支持！！！有任何问题，欢迎随时与我交流~

第16章 使用 Kotlin Native

不得不说 JetBrains 是一家务实的公司，各种IDE让人赞不绝口，用起来也是相当溜。同样的，诞生自 JetBrains 的 Kotlin 也是一门务实的编程语言，Kotlin以工程实用性为导向，充分借鉴了Java, Scala, Groovy, C#, Gosu, JavaScript, Swift等等语言的精华，让我们写起代码来可谓是相当优雅却又不失工程质量与效率。Kotlin Native能把 Kotlin代码直接编译成机器码，也就是站在了跟 C/C++、Go和Rust的同一个层次，于是这个领域又添一位竞争对手。

在前面的所有章节中，我们使用的 Kotlin 都是基于 JVM 的运行环境。本章我们将从 JVM 的运行环境中离开，走向直接编译生成原生机器码的系统编程的生态系统：Kotlin Native 。

16.1 Kotlin Native 简介

Kotlin Native利用LLVM来编译到机器码。Kotlin Native 主要是基于 LLVM后端编译器（Backend Compiler）来生成本地机器码。

Kotlin Native 的设计初衷是为了支持在非JVM虚拟机平台环境的编程，如 ios、嵌入式平台等。同时支持与 C 互操作。

16.1.1 LLVM

LLVM最初是Low Level Virtual Machine的缩写，定位是一个虚拟机，但是是比较底层的虚拟机。LLVM是构架编译器(compiler)的框架系统，以C++编写而成，用于优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)，对开发者保持开放，并兼容已有脚本。

LLVM的出现正是为了解决编译器代码重用的问题，LLVM一上来就站在比较高的角度，制定了LLVM IR这一中间代码表示语言。LLVM IR充分考虑了各种应用场景，例如在IDE中调用LLVM进行实时的代码语法检查，对静态语言、动态语言的编译、优化等。

16.1.2 支持平台

Kotlin Native现在已支持以下平台：

平台名称	target 配置
Linux	linux
Mac OS	macbook
Windows	mingw
Android arm32	android_arm32
Android arm64	android_arm64
iOS	iphone

Raspberry Pi	raspberrypi

这意味着我们可以在这些平台上愉快地开始体验了！目前Kotlin Native 已经发布的最新预发布版本是 v0.3 。

16.1.3 解释型语言与编译型语言

编译型语言，是在程序执行之前有一个单独的编译过程，将程序翻译成机器语言，以后执行这个程序的时候，就不用再进行翻译了。例如，C/C++ 等都是编译型语言。

解释型语言，是在运行的时候将程序翻译成机器语言，所以运行速度相对于编译型语言要慢。例如，Java，C#等都是解释型语言。

虽然Java程序在运行之前也有一个编译过程，但是并不是将程序编译成机器语言，而是将它编译成字节码（可以理解为一个中间语言）。在运行的时候，由JVM将字节码再翻译成机器语言。

16.2 快速开始 Hello World

16.2.1 运行环境准备

我们直接去 Github上面去下载 kotlin-native 编译器的软件包。下载地址是：https://github.com/JetBrains/kotlin-native/releases 。

Downloads

- kotlin-native-linux-0.3.tar.gz
- kotlin-native-macos-0.3.tar.gz
- kotlin-native-windows-0.3.zip

下载解压之后，我们可以看到 Kotlin Native 编译器 konan 的目录如下：

```
-rw-r--r--@  1 jack  staff   6828  6 20 22:47 GRADLE_PLUGIN.md
-rw-r--r--@  1 jack  staff  16286  6 20 22:47 INTEROP.md
-rw-r--r--@  1 jack  staff   1957  6 21 01:03 README.md
-rw-r--r--@  1 jack  staff   4606  6 20 22:47 RELEASE_NOTES.md
drwxr-xr-x@  8 jack  staff    272  6 20 23:04 bin
drwxr-xr-x   6 jack  staff    204  7 28 17:08 dependencies
drwxr-xr-x@  3 jack  staff    102  6 20 23:01 klib
drwxr-xr-x@  5 jack  staff    170  5 12 00:02 konan
drwxr-xr-x@  4 jack  staff    136  5 12 00:02 lib
drwxr-xr-x@ 22 jack  staff    748  6 22 19:04 samples
```

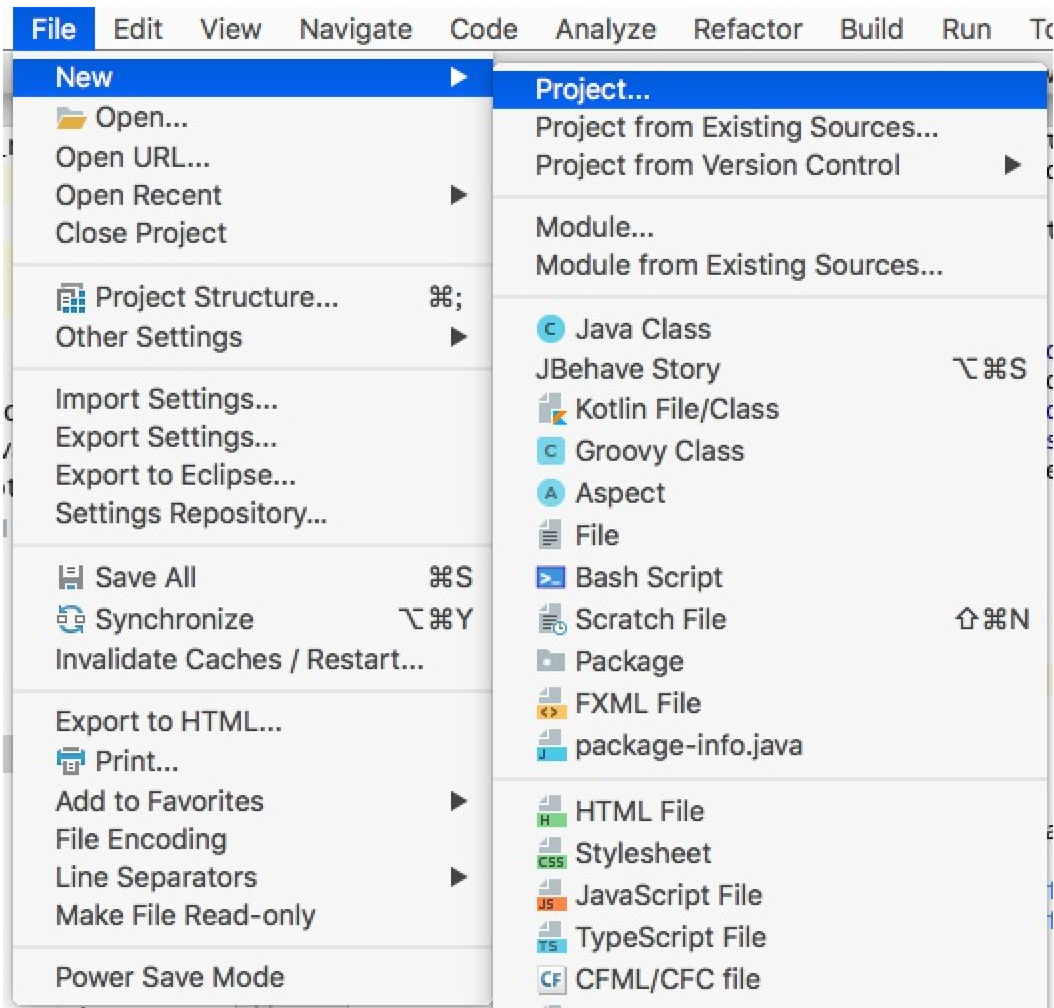
关于这个目录里面的内容我们在后面小节中介绍。

另外，我们也可以自己下载源码编译，这里就不多说了。

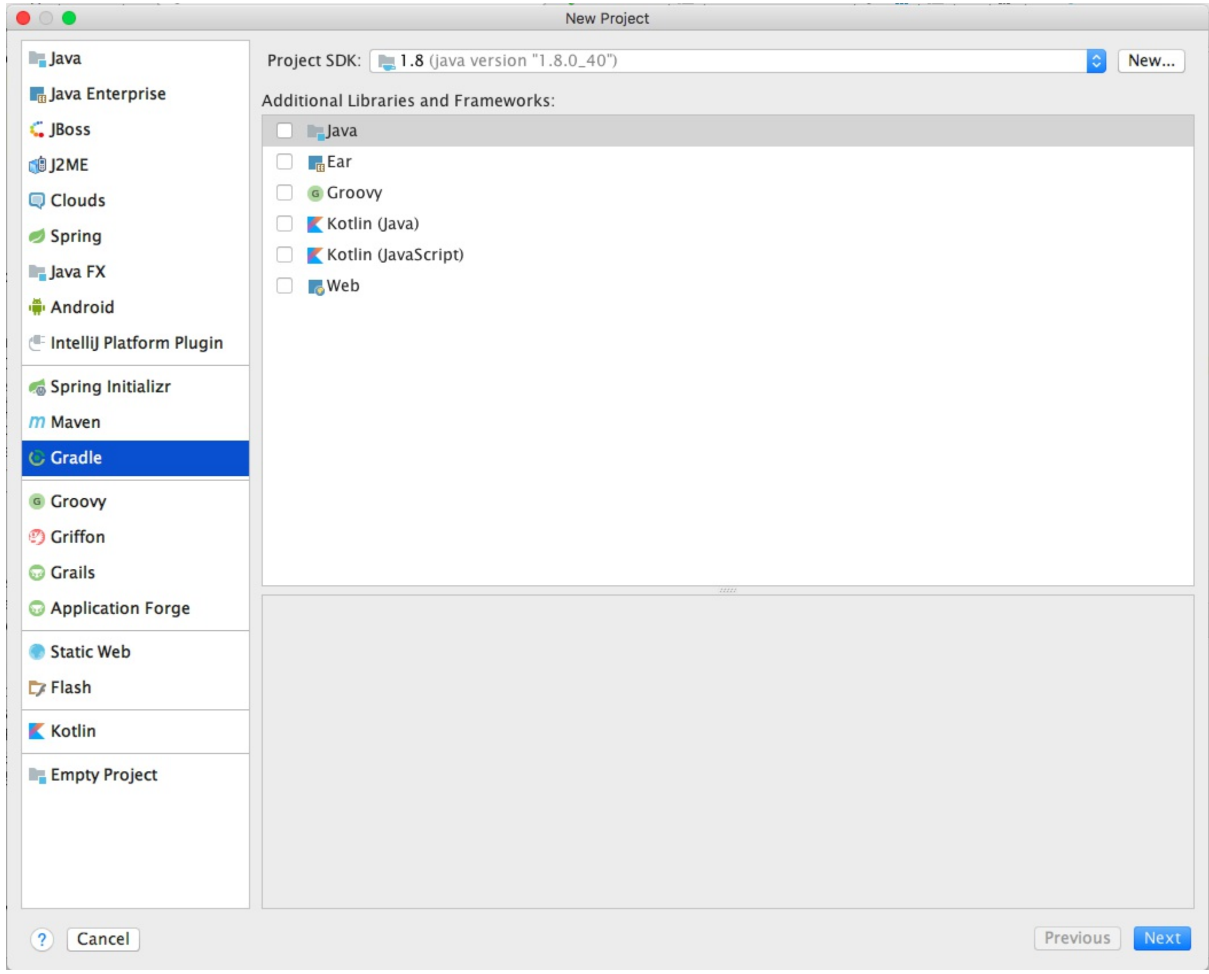
16.2.2新建 Gradle 工程

在本小节中，我们先来使用IDEA 来创建一个普通的 Gradle 工程。

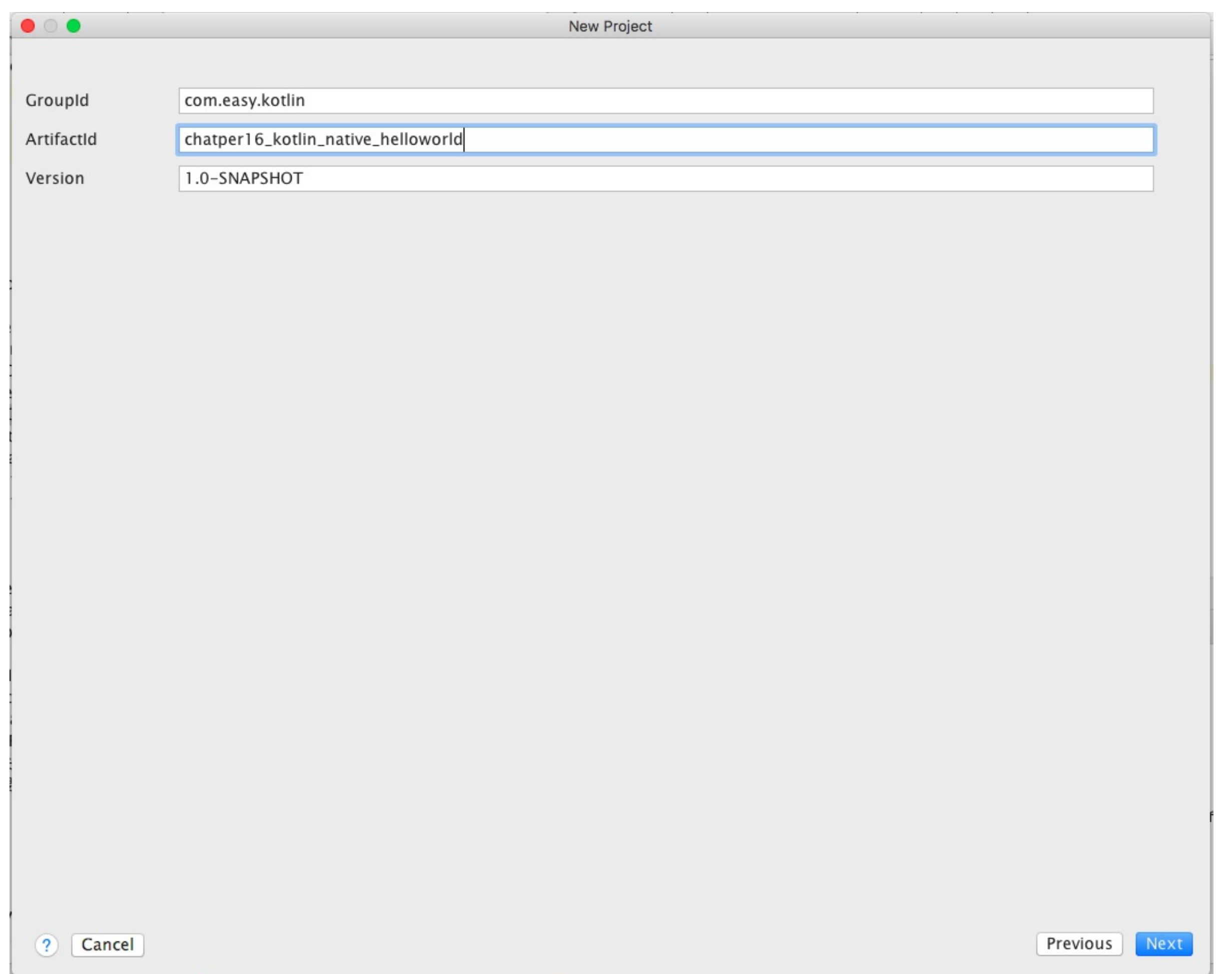
第1步，打开 File -> New -> Project ，如下图所示



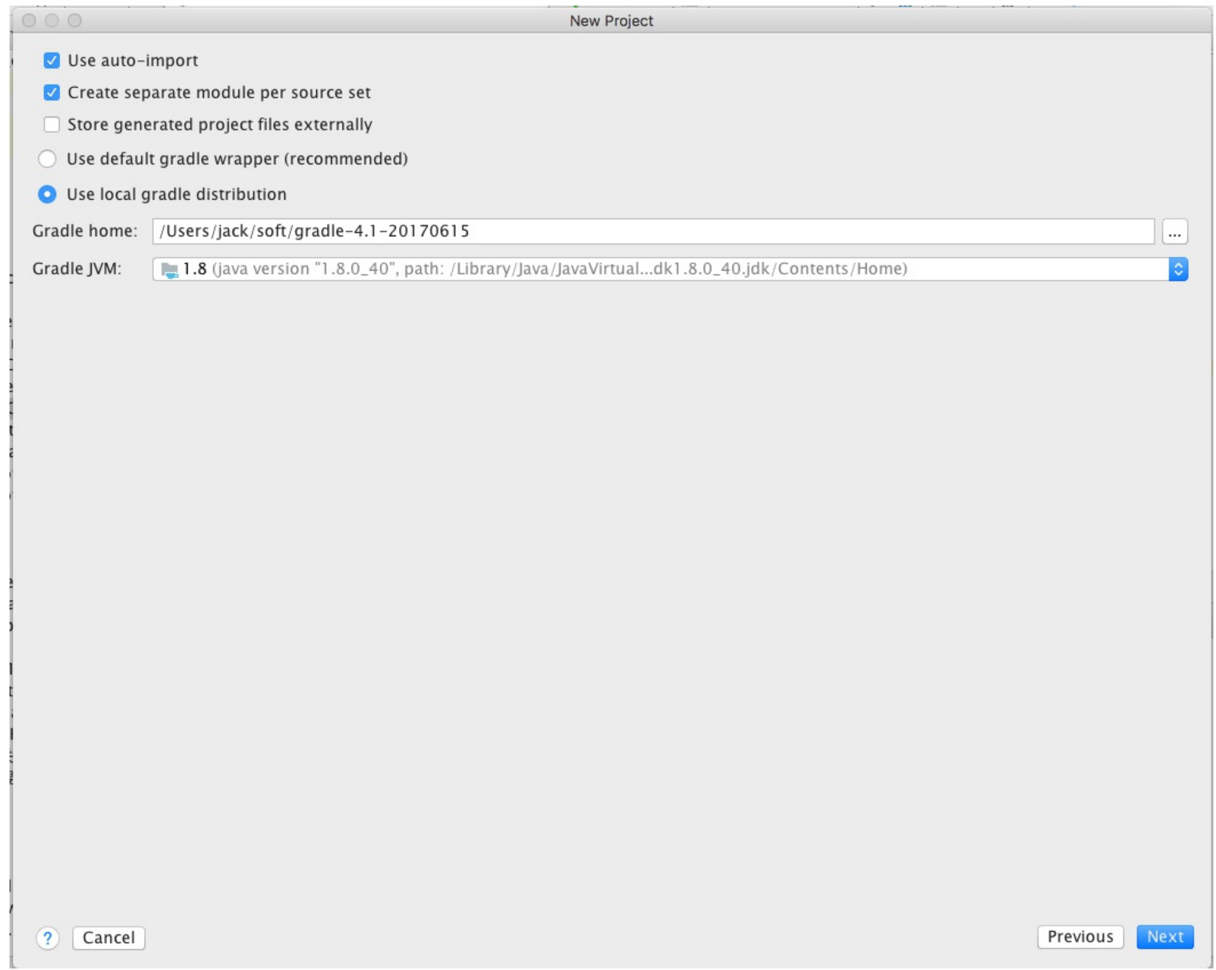
第2步，新建Gradle项目。我们直接在左侧栏中选择 Gradle，点击 Next



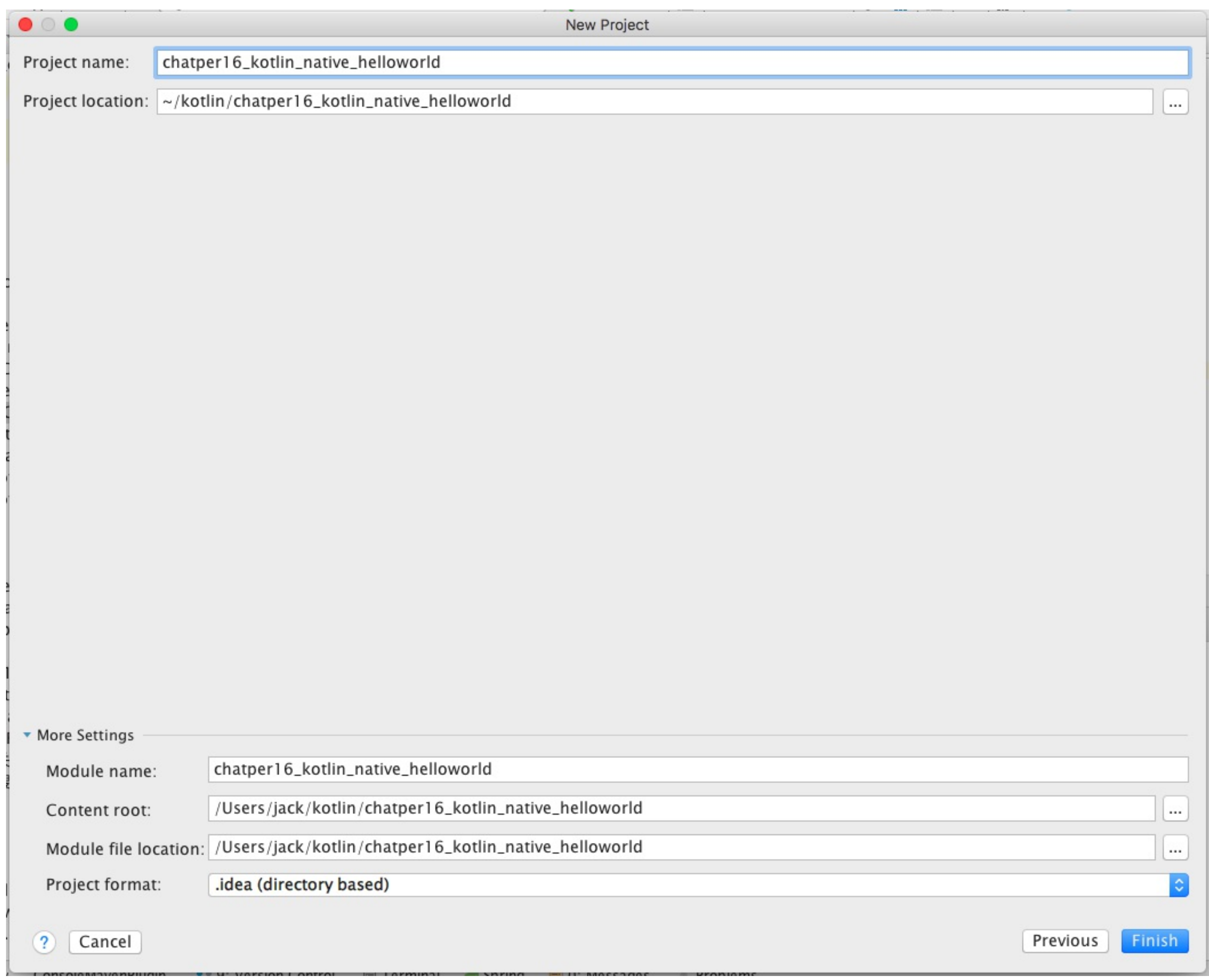
第3步，设置项目的 GroupId、ArtifactId、Version 信息



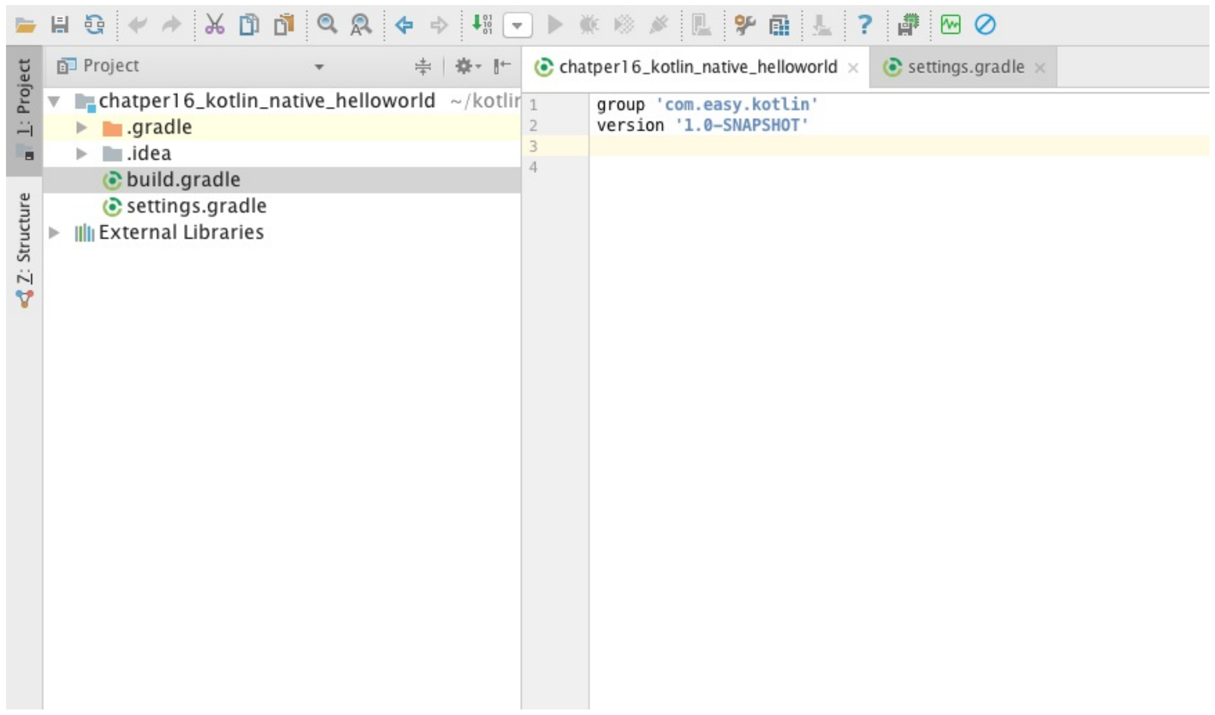
第4步，配置 Gradle 项目的基本设置。我们直接选择本地的 Gradle 环境目录，省去下载的时间（有时候网络不好，要下载半天），具体配置如下图所示



第5步，配置项目名称和项目存放目录，点击 Finish



第6步，等待 IDEA 创建完毕，我们将得到一个如下的Gradle 工程



现在这个工程里面什么都没有。下面我们就来开始原始的手工新建文件编码。

16.2.3 源代码目录

首先我们在工程根目录下面新建 src 目录，用来存放源代码。在 src 下面新建 c 目录存放 C 代码，新建 kotlin 目录存放 Kotlin 代码。我们的源代码组织结构设计如下

```
src
├── c
│   ├── cn_kotlinor.c
│   ├── cn_kotlinor.h
└── kotlin
    └── main.kt
```

16.2.4 C 代码文件

cn_kotlinor.h

C头文件中声明如下

```
#ifndef CN_KOTLINOR_H
#define CN_KOTLINOR_H
void printHello();
int factorial(int n);
int fib(int n);
#endif
```

我们简单声明了3个函数。

cn_kotlinor.c

C 源代码文件内容如下

```
#include "cn_kotlinor.h"
#include <stdio.h>

void printHello(){
    printf("[C]HelloWorld\n");
}

int factorial(int n){
    printf("[C]calc factorial: %d\n", n);
    if(n == 0) return 1;
    return n * factorial(n - 1);
}

int fib(int n){
    printf("[C]calc fibonacci: %d\n", n);
```

```
    if(n==1||n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```

这就是我们熟悉的 C 语言代码。

16.2.5 Kotlin 代码文件

main.kt 文件内容如下

```
import ckotlinor.*

fun main(args: Array<String>) {
    printHello()
    (1..7).map(::factorial).forEach(::println)
    (1..7).map(::fib).forEach(::println)
}
```

其中， `import kotlinor.*` 是 C 语言代码经过 clang 编译之后的C 的接口包路径，我们将在下面的 build.gradle 配置文件中的konanInterop中配置这个路径。

16.2.6 konan插件配置

首先，我们在 build.gradle 里面添加构建脚本 buildscript 闭包

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:0.3"
    }
}
```

这里我们添加了Gradle 构建 Kotlin Native 工程的 DSL 插件 kotlin-native-gradle-plugin:0.3 。这里的版本号，对应我们下载的 konan 编译器的版本号，我们使用的是 v0.3，所以这里我们也使用0.3版本的插件。这个插件发布在https://dl.bintray.com/jetbrains/kotlin-native-dependencies仓库里，所以我们在repositories里面添加了这个仓库。

然后，我们应用插件 konan

```
apply plugin: 'konan'
```

konan 就是用来编译 Kotlin 为 native 代码的插件。

16.2.7 konanInterop 互操作配置

```
konanInterop {
    ckotlinor {
        defFile 'kotlinor.def' // interop 的配置文件
        includeDirs "src/c" // C 头文件目录，可以传入多个
    }
}
```

konanInterop 主要用来配置 Kotlin 调用 C 的接口。konanInterop 的配置是由konan 插件API中的 KonanInteropTask.kt来处理的（这个类的源码在： https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanInteropTask.kt）。

这里我们声明的 ckotlinor 是插件中的KonanInteropConfig 对象。我们在下面的konanArtifacts里面会引用这个 ckotlinor 。

关于konanInterop的配置选项有

```
  konanInterop {
       pkgName {
           defFile <def-file>
           pkg <package with stubs>
           target <target: linux/macbook/iphone/iphone_sim>
           compilerOpts <Options for native stubs compilation>
           linkerOpts <Options for native stubs >
           headers <headers to process>
           includeDirs <directories where headers are located>
           linkFiles <files which will be linked with native stubs>
           dumpParameters <Option to print parameters of task before execution>
       }

       // TODO: add configuration for konan compiler
 }
```

我们简要说明如下表所示

配置项	功能说明
defFile	互操作映射关系配置文件
pkg	C 头文件编译后映射为 Kotlin 的包名
target	编译目标平台：linux/macbook/iphone/iphone_sim等

compilerOpts	编译选项
linkerOpts	链接选项
headers	要处理的头文件
includeDirs	包括的头文件目录
linkFiles	与native stubs 链接的文件
dumpParameters	打印 Gradle 任务参数选项配置

其中，kotlinor.def 是Kotlin Native 与 C 语言互操作的配置文件，我们在kotlinor.def 里面配置 C 源码到 kotlin 的映射关系。这个文件内容如下

kotlinor.def

```
headers=cn_kotlinor.h
compilerOpts=-Isrc/c
```

同样的配置，如果我们写在 build.gradle 文件中的konanInterop配置里如下

```
konanInterop {
    ckotlinor {
        // defFile 'kotlinor.def' // interop 的配置文件
        compilerOpts '-Isrc/c'
        headers 'src/c/cn_kotlinor.h' // interop 的配置文件
        includeDirs "src/c" // C 头文件存放目录，可以传入多个
    }
}
```

关于这个配置文件的解析原理可以参考 KonanPlugin.kt 文件的源码（https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanPlugin.kt）。

16.2.8 konanArtifacts 配置

在 konan 插件中，我们使用konanArtifacts来配置编译任务执行。

```
konanArtifacts {
    KotlinorApp { // (1)
        inputFiles fileTree("src/kotlin") // (2)
        useInterop 'ckotlinor' // (3)
        nativeLibrary fileTree('src/c/cn_kotlinor.bc') // (4)
        target 'macbook' // (5)
    }
}
```

其中，(1)处的KotlinorApp名称，在 build 之后会生成以这个名称命名的 KotlinorApp.kexe 可执行程序。 (2)处的inputFiles配置的是 kotlin 代码目录，程序执行的入口 main 定义在这里。

(3)处的useInterop 配置的是使用哪个互操作配置。我们使用的是前面的 konanInterop 里面的配置 ckotlinor 。

(4) 处的nativeLibrary配置的是本地库文件。关于'src/c/cn_kotlinor.bc'文件的编译生成我们在下面讲。

(5) 处的target 配置的是编译的目标平台，这里我们配置为 'macbook' 。

关于konanArtifacts可选的配置如下所示

```
konanArtifacts {

      artifactName1 {

          inputFiles "files" "to" "be" "compiled"

          outputDir "path/to/output/dir"

          library "path/to/library"
          library File("Library")

          nativeLibrary "path/to/library"
          nativeLibrary File("Library")

          noStdLib
          produce "library"|"program"|"bitcode"
          enableOptimization

          linkerOpts "linker" "args"
          target "target"

          languageVersion "version"
          apiVersion "version"

    }
     artifactName2 {

          extends artifactName1

          inputDir "someDir"
          outputDir "someDir"
     }

  }
```

konan 编译任务配置处理类是KonanCompileTask.kt （https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanCompileTask.kt） 。

16.2.9 完整的 build.gradle 配置

完整的 build.gradle 配置文件内容如下

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'


buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:0.3"
    }
}

apply plugin: 'konan' // konan 就是用来编译 Kotlin 为 native 代码的插件


konanInterop { // konanInterop 主要用来配置 Kotlin 调用 C 的接口
    ckotlinor {
        defFile 'kotlinor.def' // interop 的配置文件
        includeDirs "src/c" // C 头文件目录，可以传入多个
    }
}

konanArtifacts { //konanArtifacts 配置我们的项目
    KotlinorApp { // build 之后会生成 KotlinorApp.kexe 可执行程序
        inputFiles fileTree("src/kotlin") //kotlin 代码配置，项目入口 main 需要定义在这里
        useInterop 'ckotlinor' //使用前面的 konanInterop 里面的配置  kotlinor{ ... }
        nativeLibrary fileTree('src/c/cn_kotlinor.bc') //自己编译的 llvm 字节格式的依赖
        target 'macbook' // 编译的目标平台
    }
}
```

提示：关于konan 插件详细配置文档：Gradle DSL https://github.com/JetBrains/kotlin-native/blob/master/GRADLE_PLUGIN.md

16.2.10 使用 clang 编译 C 代码

为了实用性，我们新建一个 shell 脚本 kclang.sh 来简化 clang 编译的命令行输入参数

```
#!/usr/bin/env bash
clang -std=c99 -c $1 -o $2 -emit-llvm
```

这样，我们把 kclang.sh 放到 C 代码目录下，然后直接使用脚本来编译：

```
 kclang.sh cn_kotlinor.c cn_kotlinor.bc
```

我们将得到一个 cn_kotlinor.bc 库文件。

提示：clang是一个C++编写、基于LLVM、发布于LLVM BSD许可证下的C/C++/Objective-C/Objective-C++编译器。它与GNU C语言规范几乎完全兼容。更多关于 clang 的内容可参考 ：http://clang.llvm.org/docs/index.html 。

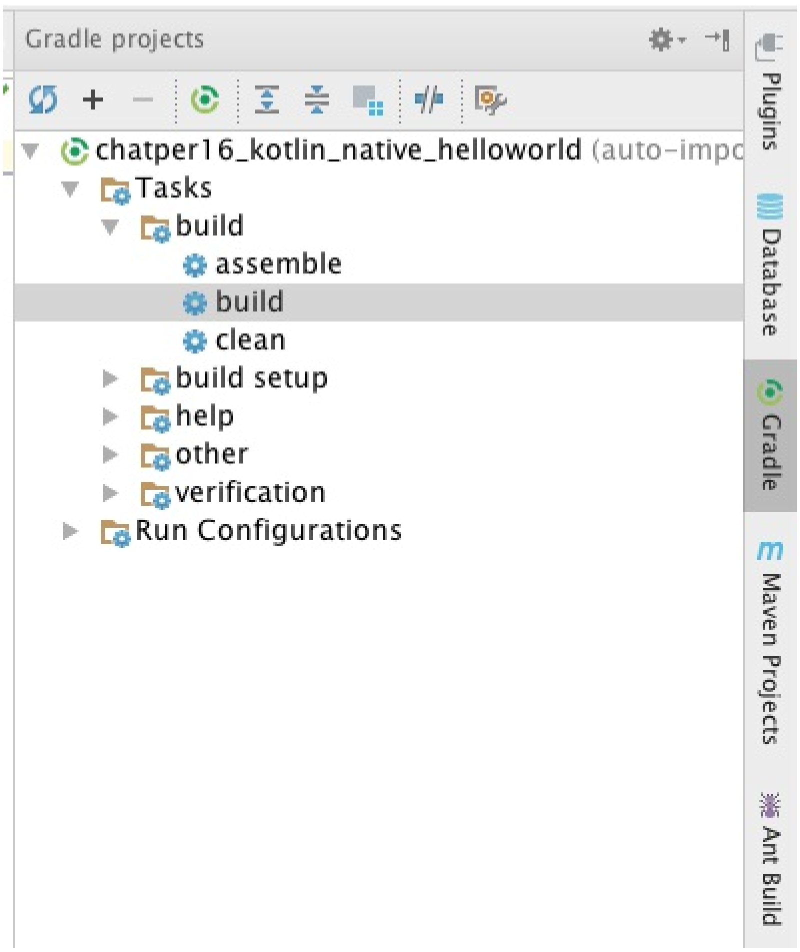
16.2.11 配置 konan 编译器主目录

最后，在执行 Gradle 构建之前，我们还需要指定konan 编译器主目录。我们在工程根目录下面新建 gradle.properties 这个属性配置文件，内容如下

```
konan.home=/Users/jack/soft/kotlin-native-macos-0.3
```

16.2.12 执行构建操作

我们直接在 IDEA 右侧的 Gradle 工具栏点击Tasks ->build -> build 命令执行构建操作



我们会看到终端输出

```
15:12:02: Executing external task 'build'...
:assemble UP-TO-DATE
:check UP-TO-DATE
:downloadKonanCompiler
:genKotlinerInteropStubs
:compileKotlinerInteropStubs
KtFile: kotliner.kt
:compileKonanKotliner
KtFile: main.kt
ld: warning: object file (/var/folders/q5/kvt7_nsd6ngdw5qry4d99xv00000gn/T/combined6977
50051437954502.o) was built for newer OSX version (10.12) than being linked (10.11)
:compileKonan
:build

BUILD SUCCESSFUL in 29s
4 actionable tasks: 4 executed
15:12:31: External task execution finished 'build'.
```

构建完成之后，会在build/konan/bin/目录下面生成一个KotlinorApp.kexe可执行程序，它直接在Mac OS 上运行，不再依赖JVM 环境了。我们得到的完整的构建输出目录树如下

```
build
└── konan
    ├── bin
    │   ├── KotlinorApp.kexe
    │   └── KotlinorApp.kt.bc
    ├── interopCompiledStubs
    │   └── ckotlinorInteropStubs
    │       ├── ckotlinorInteropStubs
    │       │   ├── linkdata
    │       │   │   ├── module
    │       │   │   ├── package_ckotlinor
    │       │   │   └── root_package
    │       │   ├── manifest
    │       │   ├── resources
    │       │   └── targets
    │       │       └── macbook
    │       │           ├── kotlin
    │       │           │   └── program.kt.bc
    │       │           └── native
    │       └── ckotlinorInteropStubs.klib
    ├── interopStubs
    │   └── genCkotlinorInteropStubs
    │       └── ckotlinor
    │           └── ckotlinor.kt
    └── nativelibs
        └── genCkotlinorInteropStubs
            └── ckotlinorstubs.bc

16 directories, 10 files
```

其中在 ckotlinor.kt中，我们可以看出 konan 编译器还为我们生成了 C 代码对应的 Kotlin 的接口

```
@file:Suppress("UNUSED_EXPRESSION", "UNUSED_VARIABLE")
package ckotlinor

import konan.SymbolName
import kotlinx.cinterop.*

fun printHello(): Unit {
    val res = kni_printHello()
    return res
}

@SymbolName("ckotlinor_kni_printHello")
private external fun kni_printHello(): Unit

fun factorial(n: Int): Int {
```

```
    val _n = n
    val res = kni_factorial(_n)
    return res
}

@SymbolName("ckotlinor_kni_factorial")
private external fun kni_factorial(n: Int): Int

fun fib(n: Int): Int {
    val _n = n
    val res = kni_fib(_n)
    return res
}

@SymbolName("ckotlinor_kni_fib")
private external fun kni_fib(n: Int): Int
```

我们在Kotlin 代码中，调用的就是这些映射到 C 中的函数接口。

16.2.12 执行 kexe 应用程序

我们直接在命令行中执行 KotlinorApp.kexe 如下

```
chatper16_kotlin_native_helloworld$ build/konan/bin/KotlinorApp.kexe
```

我们可以看到如下输出：

```
[C]HelloWorld
[C]calc factorial: 1
[C]calc factorial: 0
[C]calc factorial: 2
...
[C]calc factorial: 2
[C]calc factorial: 1
[C]calc factorial: 0
1
2
6
24
120
720
5040
[C]calc fibonacci: 1
[C]calc fibonacci: 2
[C]calc fibonacci: 3
...
```

```
[C]calc fibonacci: 3
[C]calc fibonacci: 2
[C]calc fibonacci: 1
1
1
2
3
5
8
13
```

至此，我们完成了一次简单的Kotlin Native 与 C 语言互操作在系统级编程的体验之旅。

我们看到，Kotlin Native仍然看重互操作性(Interoperability)。它能高效地调用C函数，甚至还能从C头文件自动生成了对应的Kotlin接口，发扬了JetBrains为开发者服务的良好传统！

但是，在体验的过程中我们也发现整个过程比较手工化，显得比较繁琐（例如手工新建各种配置文件、手工使用 clang 编译C 代码等）。

不过，Kotlin Native 的 Gradle 插件用起来还是相当不错的。相信未来 IDEA 会对 Kotlin Native 开发进行智能的集成，以方便系统编程的开发者更好更快的完成项目的配置以及开发编码工作。

16.3 Kotlin Native 编译器 konan 简介

本小节我们简单介绍一下Kotlin Native 编译器的相关内容（主要以 Mac OS 平台示例）。

bin目录

bin目录下面是执行命令行

```
cinterop       klib           konanc         kotlinc        kotlinc-native  run_konan
```

`run_konan` 是真正的入口 shell，它的执行逻辑是

```
TOOL_NAME="$1"
shift

if [ -z "$JAVACMD" -a -n "$JAVA_HOME" -a -x "$JAVA_HOME/bin/java" ]; then
    JAVACMD="$JAVA_HOME/bin/java"
else
    JAVACMD=java
fi
[ -n "$JAVACMD" ] || JAVACMD=java
...
java_opts=(-ea \
            -Xmx3G \
```

```
              "-Djava.library.path=${NATIVE_LIB}" \
              "-Dkonan.home=${KONAN_HOME}" \
             -Dfile.encoding=UTF-8)

KONAN_JAR="${KONAN_HOME}/konan/lib/backend.native.jar"
KOTLIN_JAR="${KONAN_HOME}/konan/lib/kotlin-compiler.jar"
STUB_GENERATOR_JAR="${KONAN_HOME}/konan/lib/StubGenerator.jar"
INTEROP_INDEXER_JAR="${KONAN_HOME}/konan/lib/Indexer.jar"
INTEROP_JAR="${KONAN_HOME}/konan/lib/Runtime.jar"
HELPERS_JAR="${KONAN_HOME}/konan/lib/helpers.jar"
KLIB_JAR="${KONAN_HOME}/konan/lib/klib.jar"
UTILITIES_JAR="${KONAN_HOME}/konan/lib/utilities.jar"
KONAN_CLASSPATH="$KOTLIN_JAR:$INTEROP_JAR:$STUB_GENERATOR_JAR:$INTEROP_INDEXER_JAR:$KON
AN_JAR:$HELPERS_JAR:$KLIB_JAR:$UTILITIES_JAR"
TOOL_CLASS=org.jetbrains.kotlin.cli.utilities.MainKt

LIBCLANG_DISABLE_CRASH_RECOVERY=1 \
$TIMECMD "$JAVACMD" "${java_opts[@]}" "${java_args[@]}" -cp "$KONAN_CLASSPATH" "$TOOL_C
LASS" "$TOOL_NAME" "${konan_args[@]}"
```

我们可以看出，Kotlin Native 编译器 konan 的运行环境还是在 JVM 上，但是它生成的机器码的可执行程序是直接运行在对应的平台系统上（直接编译成机器语言）。

konan目录

konan目录是 Kotlin Native 编译器的核心实现部分。目录结构如下：

```
kotlin-native-macos-0.3$ tree konan
konan/
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── libcallbacks.dylib
│   ├── clangstubs
│   │   └── shared
│   │       └── libclangstubs.dylib
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
    ├── libcallbacks.dylib
```

```
    ├── libclangstubs.dylib
    ├── libllvmstubs.dylib
    └── liborgjetbrainskotlinbackendkonanhashstubs.dylib

6 directories, 16 files
```

我们可以看到在 `run_konan` 命令行 shell 中依赖了上面的这些 jar 包。上面的目录文件是 Mac OS 平台上的。

对应的 Linux 平台上的konan目录文件如下

```
kotlin-native-linux-0.3$ tree konan
konan
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── libcallbacks.so
│   ├── clangstubs
│   │   └── shared
│   │       └── libclangstubs.so
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
    ├── libcallbacks.so
    ├── libclangstubs.so
    ├── libllvmstubs.so
    └── liborgjetbrainskotlinbackendkonanhashstubs.so

6 directories, 16 files
```

Windows 平台上的 konan 目录文件如下

```
kotlin-native-windows-0.3$ tree konan
konan
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
```

```
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── callbacks.dll
│   ├── clangstubs
│   │   └── shared
│   │       └── clangstubs.dll
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
    ├── callbacks.dll
    ├── clangstubs.dll
    ├── llvmstubs.dll
    └── orgjetbrainskotlinbackendkonanhashstubs.dll

6 directories, 16 files
```

klib 目录

klib 目录下是 Kotlin 的标准库的关联元数据文件以及 Kotlin Native 针对各个目标平台的 bc 文件

```
kotlin-native-macos-0.3$ tree klib
klib/
└── stdlib
    ├── linkdata
    │   ├── module
    │   ├── package_konan
    │   ├── package_konan.internal
    │   ├── package_kotlin
    │   ├── package_kotlin.annotation
    │   ├── package_kotlin.collections
    │   ├── package_kotlin.comparisons
    │   ├── package_kotlin.coroutines
    │   ├── package_kotlin.coroutines.experimental
    │   ├── package_kotlin.coroutines.experimental.intrinsics
    │   ├── package_kotlin.experimental
    │   ├── package_kotlin.internal
    │   ├── package_kotlin.io
    │   ├── package_kotlin.properties
    │   ├── package_kotlin.ranges
    │   ├── package_kotlin.reflect
    │   ├── package_kotlin.sequences
    │   ├── package_kotlin.text
    │   ├── package_kotlin.text.regex
    │   ├── package_kotlin.util
```

```
│   ├── package_kotlinx
│   ├── package_kotlinx.cinterop
│   └── root_package
├── manifest
├── resources
└── targets
    ├── android_arm32
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── android_arm64
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── iphone
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       ├── start.bc
    │       ├── start.kt.bc
    │       └── stdlib.kt.bc
    └── macbook
        ├── kotlin
        │   └── program.kt.bc
        └── native
            ├── launcher.bc
            ├── runtime.bc
            └── start.bc

16 directories, 42 files
```

上面的目录是 kotlin-native-macos-0.3 平台的版本。我们可以看出，在Mac OS上，我们可以使用 Kotlin Native 编译android_arm32、android_arm64、iphone、macbook等目标平台的机器码可执行的程序。

另外，对应的 Linux 平台的目录文件如下

```
kotlin-native-linux-0.3$ tree klib
klib/
└── stdlib
```

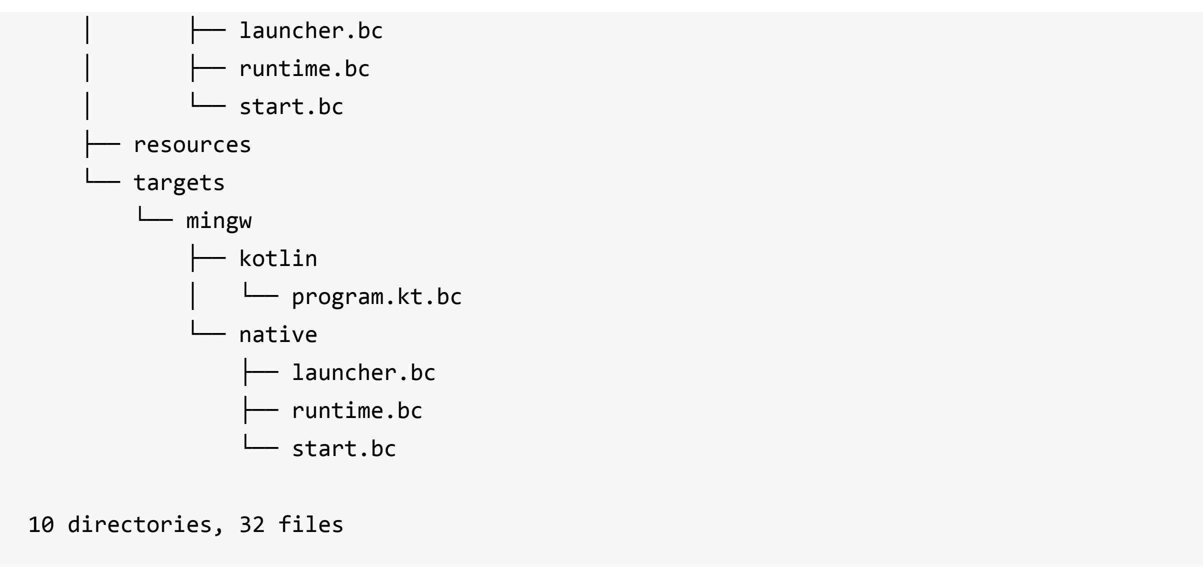
```
├── linkdata
│   ├── module
│   ├── package_konan
│   ├── package_konan.internal
│   ├── package_kotlin
│   ├── package_kotlin.annotation
│   ├── package_kotlin.collections
│   ├── package_kotlin.comparisons
│   ├── package_kotlin.coroutines
│   ├── package_kotlin.coroutines.experimental
│   ├── package_kotlin.coroutines.experimental.intrinsics
│   ├── package_kotlin.experimental
│   ├── package_kotlin.internal
│   ├── package_kotlin.io
│   ├── package_kotlin.properties
│   ├── package_kotlin.ranges
│   ├── package_kotlin.reflect
│   ├── package_kotlin.sequences
│   ├── package_kotlin.text
│   ├── package_kotlin.text.regex
│   ├── package_kotlin.util
│   ├── package_kotlinx
│   ├── package_kotlinx.cinterop
│   └── root_package
├── manifest
├── resources
└── targets
    ├── android_arm32
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── android_arm64
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── linux
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    └── raspberrypi
```

```
            ├── kotlin
            │   └── program.kt.bc
            └── native
                ├── launcher.bc
                ├── runtime.bc
                ├── start.bc
                ├── start.kt.bc
                └── stdlib.kt.bc

16 directories, 42 files
```

也就是说我们可以在 Linux 平台上编译android_arm32、android_arm64、linux、raspberrypi等平台上的目标程序。

对应Windows 平台的如下

```
kotlin-native-windows-0.3$ tree klib
klib/
└── stdlib
    ├── linkdata
    │   ├── module
    │   ├── package_konan
    │   ├── package_konan.internal
    │   ├── package_kotlin
    │   ├── package_kotlin.annotation
    │   ├── package_kotlin.collections
    │   ├── package_kotlin.comparisons
    │   ├── package_kotlin.coroutines
    │   ├── package_kotlin.coroutines.experimental
    │   ├── package_kotlin.coroutines.experimental.intrinsics
    │   ├── package_kotlin.experimental
    │   ├── package_kotlin.internal
    │   ├── package_kotlin.io
    │   ├── package_kotlin.properties
    │   ├── package_kotlin.ranges
    │   ├── package_kotlin.reflect
    │   ├── package_kotlin.sequences
    │   ├── package_kotlin.text
    │   ├── package_kotlin.text.regex
    │   ├── package_kotlin.util
    │   ├── package_kotlinx
    │   ├── package_kotlinx.cinterop
    │   └── root_package
    ├── manifest
    ├── mingw
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
```



```
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── resources
    └── targets
        └── mingw
            ├── kotlin
            │   └── program.kt.bc
            └── native
                ├── launcher.bc
                ├── runtime.bc
                └── start.bc

10 directories, 32 files
```

在 Windows 平台中，Kotlin Native 使用的是 mingw 库来实现的。目前，在 V0.3预发布版本，我们在 Windows 平台上可以体验的东西比较少，像 Android，iOS，Raspberrypi都还不支持。

提示：MinGW，是Minimalist GNUfor Windows的缩写。它是一个可自由使用和自由发布的Windows特定头文件和使用GNU工具集导入库的集合，允许你在GNU/Linux和Windows平台生成本地的Windows程序而不需要第三方C运行时（C Runtime）库。MinGW 是一组包含文件和端口库，其功能是允许控制台模式的程序使用微软的标准C运行时（C Runtime）库（MSVCRT.DLL）,该库在所有的 NT OS 上有效，在所有的 Windows 95发行版以上的 Windows OS 有效，使用基本运行时，你可以使用 GCC 写控制台模式的符合美国标准化组织（ANSI）程序，可以使用微软提供的 C 运行时（C Runtime）扩展，与基本运行时相结合，就可以有充分的权利既使用 CRT（C Runtime）又使用 WindowsAPI功能。

samples目录

samples目录下面是官方给出的一些实例。关于这些实例的文档介绍以及源码工程是：https://github.com/JetBrains/kotlin-native/tree/master/samples 。想更加深入了解学习的同学可以参考。

本章小结

本章工程源码： https://github.com/EasyKotlin/chatper16_kotlin_native_helloworld

现在我们可以把 Kotlin 像C 一样地直接编译成的机器码来运行，这样在 C 语言出现的地方（例如应用于嵌入式等对性能要求比较高的场景），Kotlin 也来了。Kotlin 将会在嵌入式系统和物联网、数据分析和科学计算、游戏开发、服务端开发和微服务等领域持续发力。

Kotlin 整个语言的架构不可谓不宏大：上的了云端（服务端程序），下的了手机端（ Kotlin / Native ），写的了前端（JS，HTML DSL 等），嵌的了冰箱（Kotlin Native）。Kotlin 俨然已成为一门擅长多个领域的语言了。

在互联网领域，目前在Web服务端应用开发、Android移动端开发是Kotlin 最活跃的领域。 Kotlin 将会越来越多地进入 Java 程序员们的视野， Java 程序员们会逐渐爱上 Kotlin 。

未来的可能性有很多。但是真正的未来还是要我们去创造。