

目录

前言	1.1
Kotlin学习资料	1.2
关于Kotlin	1.3
你应该学 Kotlin 吗？	1.3.1
Android 必备技能：最有可能接替Java的语言——Kotlin	1.3.2
你为什么需要 Kotlin	1.3.3
为什么我要改用Kotlin	1.3.4
如何看待 Kotlin 成为 Android 官方支持开发语言？	1.3.5
Kotlin Script 介绍	1.3.6
Kotlin 语言高级安卓开发入门	1.3.7
Kotlin : Java 6 废土中的一线希望	1.3.8
Kotlin语言基础	1.4
package	1.4.1
声明变量和值	1.4.2
变量类型推断	1.4.3
字符串与其模板表达式	1.4.4
流程控制语句	1.4.5
代码注释、语法与标识符	1.4.6
修饰符	1.4.7
函数扩展和属性扩展	1.4.8
空指针安全	1.4.9
Kotlin入门和使用	1.4.10
面向对象	1.5
Kotlin 的类特性(上)	1.5.1
Kotlin 的类特性(下)	1.5.2
面向对象	1.5.3
继承	1.5.4

Kotlin如何优雅的实现多继承	1.5.5
类成员的可见性	1.5.6
接口和抽象类	1.5.7
属性代理	1.5.8
对象表达式和对象声明	1.5.9
伴生对象和静态成员	1.5.10
单例	1.5.11
object单例	1.5.12
密封类	1.5.13
data class	1.5.14
为什么不直接使用 ArrayInt 而是 IntArray?	1.5.15
Kotlin 遇到 MyBatis：到底是 Int 的错，还是 data class 的错？	1.5.16
函数式编程	1.6
一篇文章彻底搞懂 Kotlin 函数	1.6.1
函数	1.6.2
inline 函数	1.6.3
闭包	1.6.4
函数与闭包	1.6.5
细说 Lambda 表达式	1.6.6
高阶函数_1	1.6.7
高阶函数_2	1.6.8
像写文章一样使用 Kotlin	1.6.9
函数复合	1.6.10
函数式编程概述	1.6.11
在Kotlin中使用函数式编程	1.6.12
集合框架	1.7
Iterator	1.7.1
集合框架	1.7.2
集合类是什么	1.7.3
Kotlin 集合类简介	1.7.4

List	1.7.5
List元素操作函数	1.7.6
List集合类的基本运算函数	1.7.7
List过滤操作函数	1.7.8
映射操作函数	1.7.9
分组操作函数	1.7.10
排序操作符	1.7.11
生产操作符	1.7.12
Set	1.7.13
Map	1.7.14
集合泛型与操作符	1.7.15
泛型	1.8
Kotlin 泛型	1.9
协程	1.10
轻量级线程：协程1	1.10.1
轻量级线程：协程2	1.10.2
深入理解 Kotlin Coroutine_1	1.10.3
深入理解 Kotlin Coroutine_2	1.10.4
深入理解 Kotlin Coroutine_3	1.10.5
Kotlin与Java混合开发	1.11
Kotlin 与 Java 共存_1	1.11.1
Kotlin 与 Java 共存_2	1.11.2
勘误：Kotlin 与 Java 共存_2	1.11.3
Kotlin 与 Java 混编	1.11.4
Kotlin 兼容 Java 遇到的最大的坑	1.11.5
Kotlin新特性	1.12
Kotlin 1.0.6	1.12.1
喜大普奔！Kotlin 1.1 Beta 降临	1.12.2
Kotlin 1.1 Beta 2 发布	1.12.3
Kotlin 1.1：我们都路上	1.12.4

Kotlin 1.1	1.12.5
快速上手 Kotlin 11招	1.12.6
Plugin	1.13
用 Kotlin 写 Android 01 难道只有环境搭建这么简单？	1.13.1
用 Kotlin 写 Android 02 说说 Anko	1.13.2
IDE	1.14
高效地使用你的 IntelliJ	1.14.1
如何优雅的在微信公众号中编辑代码	1.14.2

Kotlin 基础教程



十年生死两茫茫，不思量，自难忘，华年短暂，陈辞岁月悠悠伤，

满腔热血已荒荒，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其核心竞争力的体现。

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

<https://github.com/JackChan1999/Kotlin-Tutorials>

GitBook 在线阅读

在线阅读，PDF、ePub、Mobi电子书下载

<https://www.gitbook.com/book/alleniverson/kotlin-tutorials/details>

Kotlin 极简教程

<https://www.gitbook.com/book/alleniverson/easykotlin/details>

目录

- 前言
- Kotlin学习资料
- 关于Kotlin
 - 你应该学 Kotlin 吗？
 - Android 必备技能：最有可能接替Java的语言——Kotlin
 - 你为什么需要 Kotlin
 - 为什么我要改用 Kotlin
 - 如何看待 Kotlin 成为 Android 官方支持开发语言？
 - Kotlin Script 介绍
 - Kotlin 语言高级安卓开发入门
 - Kotlin：Java 6 废土中的一线希望
- Kotlin语言基础
 - package
 - 声明变量和值
 - 变量类型推断
 - 字符串与其模板表达式
 - 流程控制语句
 - 代码注释、语法与标识符
 - 修饰符
 - 函数扩展和属性扩展
 - 空指针安全
 - Kotlin入门和使用
- 面向对象
 - Kotlin 的类特性_上
 - Kotlin 的类特性_下
 - 面向对象

- 继承
 - Kotlin如何优雅的实现多继承
 - 类成员的可见性
 - 接口和抽象类
 - 属性代理
 - 对象表达式和对象声明
 - 伴生对象和静态成员
 - 单例
 - object单例
 - 密封类
 - data class
 - 为什么不直接使用 `ArrayInt` 而是 `IntArray`?
 - Kotlin 遇到 MyBatis：到底是 Int 的错，还是 data class 的错？
- 函数式编程
 - 一篇文章彻底搞懂 Kotlin 函数
 - 函数
 - inline函数
 - 闭包
 - 函数与闭包
 - 细说 Lambda 表达式
 - 高阶函数_1
 - 高阶函数_2
 - 像写文章一样使用 Kotlin
 - 函数复合
 - 函数式编程概述
 - 在Kotlin中使用函数式编程
 - 集合框架
 - Iterator
 - 集合框架
 - 集合类是什么
 - Kotlin 集合类简介
 - List
 - List元素操作函数
 - List集合类的基本运算函数
 - List过滤操作函数
 - 映射操作函数

- 分组操作函数
- 排序操作符
- 生产操作符
- Set
- Map
- 集合泛型与操作符
- 泛型
- Kotlin 泛型
- 协程
 - 轻量级线程：协程1
 - 轻量级线程：协程2
 - 深入理解 Kotlin Coroutine_1
 - 深入理解 Kotlin Coroutine_2
 - 深入理解 Kotlin Coroutine_3
- Kotlin与Java混合开发
 - Kotlin 与 Java 共存_1
 - Kotlin 与 Java 共存_2
 - 勘误：Kotlin 与 Java 共存_2
 - Kotlin 与 Java 混编
 - Kotlin 兼容 Java 遇到的最大的坑
- Kotlin新特性
 - Kotlin 1.0.6
 - 喜大普奔！Kotlin 1.1 Beta 降临
 - Kotlin 1.1 Beta 2 发布
 - Kotlin 1.1：我们都路上
 - Kotlin 1.1
 - 快速上手 Kotlin 11招
- Plugin
 - 用 Kotlin 写 Android 01 难道只有环境搭建这么简单？
 - 用 Kotlin 写 Android 02 说说 Anko
- IDE
 - 高效地使用你的 IntelliJ
 - 如何优雅的在微信公众号中编辑代码

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](#)

1. 概述

Kotlin 成为 Android 官方支持的编程语言，今天一早上各个群都在讨论Kotlin，微信留言也有人问我的观点。

其实我对Kotlin并没有那么强烈的想学习的冲动，看了下语法和Groovy、Javascript 都很类似，可以看一个代码片段：

```
//variables and constants
var currentVersionCode = 1    //变量当前的版本号，类型Int可以根据值推断出来
var currentVersionName : String = "1.0" //显式标明类型
val APPNAME = "droidyue.com" //常量APPNAME 类型(String)可以根据值推断出来

//methods
fun main(args: Array<String>) {
    println(args)
}

// class
class MainActivity : AppCompatActivity() {

}

// data class 自动生成getter, setting, hashCode和equals等方法
data class Book(var name: String, val price: Float, var author: String)

//支持默认参数值，减少方法重载
fun Context.showToast(message: String, duration:Int = Toast.LENGTH_LONG) {
    Toast.makeText(this, message, duration).show()
}
```



代码片段引自：<http://droidyue.com/blog/2017/05/18/why-do-i-turn-to-kotlin/>

我相信不要解释大家也能看懂这些代码，而且可以使用Android Studio作为开发工具，所以大家不用担心，也不是所谓的“Java白学了，Kotlin要替代Java了”。

个人认为：

1. 有着扎实的Java基础，这东西学习起来肯定是可以速成的，所以不要惊慌，如果有原本的学习计划，也不用着急着打乱自己原本的节奏（不反对立即开始学习的哈，为爱学习的点赞）~
2. 一个新的语言想要快速的普及，那么可能只有在运行效率上提升那才是最大的优势，如果说：“XX语言被Android官方支持，运行速度比原本提升50%”，那么不用想，立即去学习（Kotlin并不具备这样的属性，而且其所表现出类似于“简洁”这样的优势，短期内在一个多年Java的开发者面前，是体现不出来的）。

所以，Kotlin肯定时值得学习的，但并没有传的那么夸张。有精力就去学习，有自己的学习计划也可以放一放。

我想只有用得多了，Kotlin的优势应该会慢慢展现出来的，但是应该需要一个较为漫长的过渡期（如果有公司内部支持，那么这个过程肯定很快）

当然，作为一名Android开发者，Google支持的东西，我肯定要无条件支持呀，所以接下来，就是Kotlin学习资料的推荐~耐不住性子的同学可以刚好接下来就是周末，尽情的学习吧

2. Kotlin学习资料

(1) 官方资料

首发的肯定是kotlin的官方github地址了：<https://github.com/JetBrains/kotlin>

晚上看了下trending，稳稳的站在第一。

平时大家没事，可以看看github trending，会有很多的新新优质资源。<https://github.com/trending/java>

Readme中包含官方的tutorials、推荐的图书《Kotlin in Action》、《Kotlin for Android Developers》以及其他资源。

英文不太好的不用担心，有官方中文翻译站点：

<https://www.kotlincn.net/docs/reference/>

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

The screenshot shows the '参考' (Reference) section of the Kotlin Reference website. At the top, there are four tabs: '参考' (selected), '教程' (Tutorial), '书籍' (Books), and '更多资源' (More Resources). On the left, a sidebar lists navigation links: '概述', '开始', '基础', '类和对象', '函数和 Lambda 表达式', '其他', '参考' (selected), 'Java 互操作', and 'JavaScript'. The main content area has a heading '参考' with a 'Edit Page' button. It contains two sections: '提供关于 Kotlin 语言的完整参考以及标准库。' (Provides a complete reference for the Kotlin language and its standard library.) and '从零开始' (Getting Started). Below these is a note: '这个参考是为你很容易地在几个小时内学习 Kotlin 而设计的。先从基本语法开始，然后再学到更高级主题。阅读时你可以在在线 IDE 中尝试代码示例。' (This reference is designed to help you learn Kotlin easily within a few hours. Start from basic syntax and move on to more advanced topics. While reading, you can try code examples in an online IDE.) At the bottom, there is a note: '一旦你认识到 Kotlin 是什么样的，尝试自己解决一些 Kotlin 心印交互式编程练习。如果你不确定如何解决一个心印，或者你正在寻找一个更优雅的解决方案，看看 Kotlin 习惯用法。' (Once you understand what Kotlin is like, try solving some Kotlin Koans interactively. If you're unsure how to solve a koan or are looking for a more elegant solution, see the Kotlin Best Practices.) A note at the bottom also says: '心印: Koan, 佛教用语, 不建议译作“公案”——译者注' (Note: Koan, a Buddhist term, is not recommended to be translated as "public case" —— Translator's note). A watermark '鸿洋' is visible in the bottom right corner.

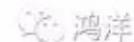
(2) 《Kotlin for android developers》中文版翻译

<https://github.com/wangjiegulu/kotlin-for-android-developers-zh/blob/master/README.md>

119 lines (116 sloc) | 5.76 KB

Summary

- [Introduction](#)
- [写在前面](#)
- [关于本书](#)
- [这本书适合你吗？](#)
- [关于作者](#)
- [介绍
 - \[什么是Kotlin？\]\(#\)
 - \[我们通过Kotlin得到什么\]\(#\)](#)
- [准备工作
 - \[Android Studio\]\(#\)
 - \[安装Kotlin插件\]\(#\)](#)



支持在线阅读和下载pdf

(3) 张涛的开源实验室

之前在推送中推荐过张涛的博客，博客质量都很高，在很早的时候就开始编写Kotlin相关博客，此外还有很多比较新的知识

<https://kymjs.com/column/kotlin.html>

《Kotlin 一门强大的语言》

Kotlin 语言程序设计

2017-04-09 By 张涛

Kotlin Primer·第五章·函数与闭包

久违了，Kotlin 的闭包。函数与闭包的特性可以算是 Kotlin 语言最大的特性了，所以写了很久。

2017-02-26 By 张涛

Kotlin Primer·第四章·Kotlin 的类特性(下)

Kotlin 中有很多非常好的特性，扩展方法、伴生对象、原生支持动态代理、伪多继承

2017-02-12 By 张涛

Kotlin Primer·第四章·Kotlin 的类特性(上)

前面三章的内容是写给希望快速了解 Kotlin 语言的大忙人的。而从本章开始，才会真正讲述 Kotlin 语言的神奇之处。

2017-02-07 By 张涛

Kotlin Primer·第三章·Kotlin 与 Java 混编

前三章的内容是为方便想在短时间内马上用上Kotlin的人，例如作为一个刚入职的新人，公司的代码已经是用Kotlin编写了，你应该如何更快地融入与适应。

2017-02-04 By 张涛

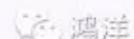
Kotlin Primer·第二章·基本语法

前三章的内容是为方便想在短时间内马上用上Kotlin的人，例如作为一个刚入职的新人，公司的代码已经是用Kotlin编写了，你应该如何更快地融入与适应。

2017-02-03 By 张涛

Kotlin Primer·第一章·启程

其实最初是准备写一本电子书然后免费开放给大家的，可惜啊可惜毅力不够，坚持不下来，所以还是当成博客来写，写好了再出电子书吧。



(4) 大精-wing的地方酒馆

让你的代码量减少3倍！使用kotlin开发Android系列

[让你的代码量减少三倍！使用kotlin开发Android\(一\) 创建Kotlin工程](#)

[让你的代码量减少3倍！使用kotlin开发Android\(二\) --秘笈！扩展函数](#)

[让你的代码量减少3倍！使用kotlin开发Android\(三\) 缩短五倍的Java Bean](#)

[让你的代码量减少3倍！使用kotlin开发Android\(四\) kotlin bean背后的秘密](#)

[使用kotlin开发Android\(五\) 缩短N倍的监听器](#)



<http://androidwing.net/index.php/89>

还有个Kotlin项目：<https://github.com/githubwing/GankClient-Kotlin>

(5) Kotlin 视频教程

竟然还有视频教程

Kotlin从入门到『放弃』系列 视频教程

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。

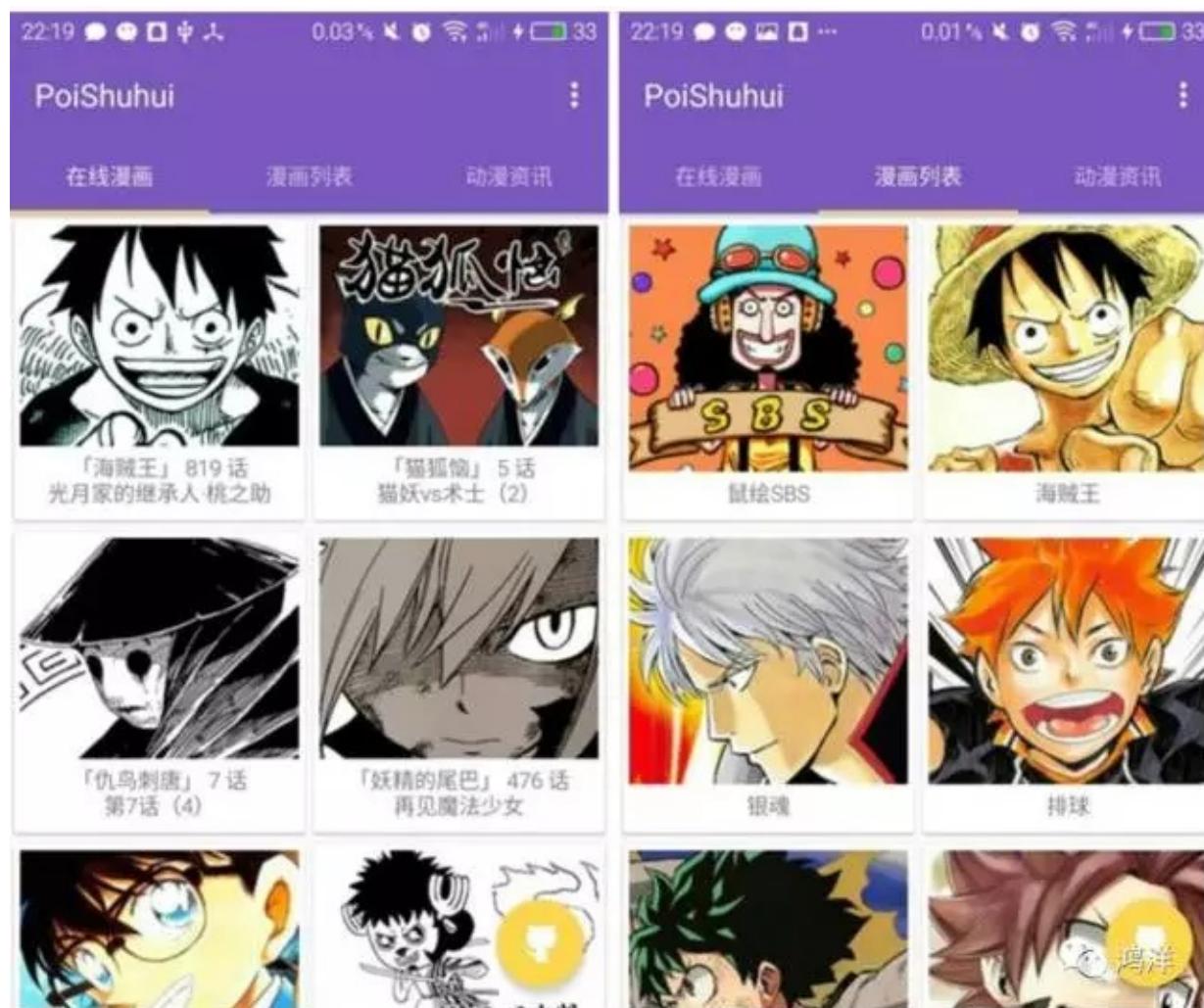
<https://github.com/enbandari/Kotlin-Tutorials>

Kotlin-Tutorials		
① 2016-10-12 22:57	失效时间：永久有效	
返回上一级 全部文件 > Kotlin-Tutorials		
<input type="checkbox"/>	文件名	
<input type="checkbox"/>	Kt00 使用 Kotlin 反返 URL 请求.mp4	1.01M
<input type="checkbox"/>	Kt07 在 RxJava 中使用 Lambda.mp4	195.1M
<input type="checkbox"/>	Kt06 枚举、When表达式.mp4	128.2M
<input type="checkbox"/>	Kt05 扁平化集合 flatMap.mp4	90.5M
<input type="checkbox"/>	Kt04 集合遍历 map.mp4	131.1M
<input type="checkbox"/>	Kt03 基于Gradle的工程示例.mp4	444.7M
<input type="checkbox"/>	Kt02 HelloWorld.mp4	75.2M
<input type="checkbox"/>	Kt01 Kotlin简介.mp4	131M
<input type="checkbox"/>	10 单例 (final) .mp4	106.6M

可以百度云下载或者腾讯视频在线观看。

- [腾讯视频](#)
- [百度云](#)

(6) 开源项目



一个用Kotlin写的简单漫画APP

<https://github.com/wuapnjie/PoiShuhui-Kotlin>

这个是我从俊林的文章中偷来的，如果有推荐可以留言~

(7) 其他文章

为什么我要改用Kotlin

<http://droidyue.com/blog/2017/05/18/why-do-i-turn-to-kotlin/>

by:技术小黑屋

Android开发必备知识：为什么说Kotlin值得一试

[https://mp.weixin.qq.com/s?
__biz=MzA3NTYzODYzMg==&mid=404087761&idx=1&sn=d80625ee52f860a7a2
ed4c238d2151b6](https://mp.weixin.qq.com/s?__biz=MzA3NTYzODYzMg==&mid=404087761&idx=1&sn=d80625ee52f860a7a2ed4c238d2151b6)

by:腾讯Bugly公众号

使用Kotlin进行Android开发

<http://ragnraok.github.io/using-kotlin-to-write-android-app.html>

by:Ragnarok Zhou

最后要非常感谢，今天给我投稿的朋友（汇总资料，就直接放出了链接啦）

使用Kotlin在Android Studio上开发App

http://blog.csdn.net/qq_25867141/article/details/52875330

by:Blincheng

关于Kotlin

- 你应该学 Kotlin 吗？
- Android 必备技能：最有可能接替Java的语言——Kotlin
- 你为什么需要 Kotlin
- 为什么我要改用 Kotlin
- 如何看待 Kotlin 成为 Android 官方支持开发语言？
- Kotlin Script 介绍
- Kotlin 语言高级安卓开发入门
- Kotlin : Java 6 废土中的一线希望

你应该学 **Kotlin** 吗？

作者：bennyhuo 链接：<http://www.imooc.com/article/18195> 来源：慕课网

自从5.18 Google IO 大会以来，关注 Kotlin 或者说想要搞清楚 Kotlin 是什么的人越来越多了。有不少朋友表示在这之前自己都没有听说过 Kotlin，现在突然 Google 空降一个干儿子，你是不是就慌了呢？

下面我列举几个常见的问题，希望能为大家解惑~

Kotlin 是什么？

我刚刚接触 Kotlin 是在两年前，那会儿的 Kotlin 更像是 A Better Java，它能做 Java 能做的任何事情，而且还要更出色。那时候我们还经常称 Kotlin 是一门 Jvm 语言，尽管 Kotlin-js 也在实验当中；而现在，我们只好称 Kotlin 是一门全栈的静态语言了，因为小伙儿长壮实了，不仅通吃 Jvm，Android 和前端，连 Native（注意不是 Jni）也要搞搞，真是前景一片大好。

说白了，它就是一门编程语言而已，在 Android 上被 Google 钦点，就像 Swift 之于 Objective-C，并不是什么洪水猛兽，不应该觉得害怕和恐慌。

没有 Java 基础，我应该学 **Kotlin** 吗？

应该学，作为一个有追求的程序员，你不仅要学 Kotlin，而且要了解各类型的语
言，这跟你什么基础没有关系，大家都有过一个什么都不会的曾经。

当然，现阶段，Kotlin 跟 Java 关系实在密切，只要你希望用 Kotlin 开发 Jvm、
Android 相关的应用程序，Java 你就必须学会，而且要学好，因为它太重要了，除了大量的 Jdk、Sdk 源码都是 Java 的以及大量的资料都是 Java 的之外，Java 更是
Java 系语言的标杆，你需要学习掌握它，了解这一族语言的应用思路，和开发习
惯，甚至了解 Java 的弊病来真正明白为什么会有 Kotlin、Groovy、Scala 这样的语
言诞生。

不学 Java 可以直接搞 Android 开发吗？

可以，做了很多年 Android 开发的我们居然会惊讶的发现，可以用来开发 Android 语言好多，Java 系的 Groovy、Scala 就不提了，连 C# 都可以，真是条条大路通罗马。

当然，主流语言仍然是 Java，这将是在短期内无法动摇的，加之 Kotlin 与 Java 一脉相承，关系密切，随着 Google 的强力推荐，你应该逐渐学习 Kotlin 与 Java 并适应二者共存的状态。

Kotlin 会不会慢慢把 Java 取代了？

短期内不会，但会 Kotlin 的开发会把不会 Kotlin 的开发慢慢取代倒是极有可能。

Java 短期内其地位是无法被撼动的，为什么？一方面自然有积重难返的原因，毕竟 Java 的积累很多，完全清空转向新生的 Kotlin 或者其他语言不是一件容易的事，而且也没有必要，用 Java 写的应用不是仍然好好的在应用商店躺着吗，老板凭什么要为 Google 的钦点额外付费？

那么后面的发展可能是怎样的呢？公司老板慢慢发现 Kotlin 比 Java 的开发效率高，可以节省人力，于是尝试逐渐过渡，进入 Java 和 Kotlin 混合开发阶段，不过他接着发现，掌握 Kotlin 的程序员的成本要高一些，于是他开始盘算到底是招 10 个 Java 合适还是招 7 个 Java & Kotlin 合适。等到掌握 Kotlin 的人越来越多，大家会普遍倾向于认为 Kotlin 是一门必备技能，不会 Kotlin 的话就会被淘汰，这时候老板就只会招 Java & Kotlin 了。

现在用 Kotlin 开发 Android 靠谱吗？

额。。。看到这个问题的时候其实我好难过，然后瞬间又不难过了，因为难过的应该是 Google——你看，人家都不相信你呀。

我从两年前开始接触 Kotlin，大概在一年半以前开始所有个人开发的应用都直接使用 Kotlin 开发，接着在公司项目中小范围尝试了 Kotlin，单从编程语言层面来看，Kotlin 几乎没有任何问题，Google 也通过这次 IO 大会试图在告诉我们这一点。坚定支持 Kotlin 的还有 Square.Inc 的 Jake 大神，如果你不知道他的话，我建议你的领导开除你。国内使用 Kotlin 的公司比较有名的有魅族、沪江、英语流利说等，他们都有了比较长的使用历史，一些创业公司也发现 Kotlin 能为他们的团地带来活力，让兄弟不至于很辛苦，比如北京的快乐迭代。

所以，如果你为 Kotlin 开发 Android 靠谱不靠谱，我告诉你，非常靠谱，只要你的同事不反对你就可以啦 ps: 现在谁反对谁就是政治不正确 :) 逃

我该什么时候学 **Kotlin** ？

这几天在看《那年那兔那些事儿》，用动漫的形式把中国从耻辱的近代到现代再到现在的故事讲述了一遍。其中有一段，讲我们国家的代表去美国参观，见识到美国强大的军事实力时候，感慨，什么时候我们也能有这么大的航母？接着坚定的说：不管怎么样，今天开始学一定比明天开始学早一天学会。

技术在变，身边的人在变，环境在变，整个世界都在变，你不得不集中精力往前冲，你以为你跑到前面了吗？不是的，你胆敢停下来一秒，就会被多少人超过。

这显然不是什么时候学习 Kotlin 的问题，如果你觉得某个东西有用，不要犹豫，现在立刻马上赶紧去学！

Android 必备技能：最有可能接替Java的语言 ——Kotlin

原文链接：<http://bennyhuo.leanote.com/post/Android-A-Powerful-Substitution-Kotlin>

1. Hello, Kotlin

1.1 Kotlin的身世

- 写了许久Java，有没有发现其实你写了太多冗余的代码？
- 后来你体验了一下Python，有没有觉得不写分号的感觉真是超级爽？
- 你虽然勤勤恳恳，可到头来却被NullPointerException折磨的死去活来，难道就没有受够这种日子么？
- 直到有一天你发现自己已经写了好几十万行代码，发现居然全是getter和setter！

哈哈，实际上你完全可以不用这么痛苦，用Kotlin替代Java开发你的程序，无论是Android还是Server，你都能像之前写Java一样思考，同时又能享受到新一代编程语言的特性，说到这里你是不是开始心动了呢？下面我就通过这篇文章来给大家介绍一下Kotlin究竟是何方神圣。

话说，Kotlin是JetBrain公司搞出来的，运行在JVM上的一门静态类型语言，它是用波罗的海的一个小岛的名字命名的。从外观上，乍一看还以为是Scala，我曾经琢磨着把Scala作为我的下一门语言，不过想想用Scala来干嘛呢，我又不做大数据，而它又太复杂了o(╯□╰)o

用Kotlin创建一个数据类

```
data class Mondai(var index: Int = 0,
                  var title: String = "",
                  val ans: ArrayList<String> = ArrayList(),
                  var correct: Int = 0,
                  var comment: String = "",
                  var color: String = "",
                  private var lives: Int = 50)
```

最初是在intelliJ的源码中看到Kotlin的，那时候Kotlin的版本还不太稳定，所以源码总是编译不过，真是要抓狂啊，还骂『什么破玩意儿！为什么又出来新语言了？Groovy还没怎么学会，又来个Kotlin！』话说，Kotlin，难道是『靠它灵』的意思？？

其实经过一年多的发展，Kotlin 1.0已经release，feature基本完善，api也趋于稳定，这时候尝试也不会有那种被坑的感觉了。过年期间也算清闲，于是用Kotlin做了个app，简单来说，就是几个感觉：

- 思路与写Java时一样，不过更简洁清爽
- 少了冗余代码的烦恼，更容易专注于功能的开发，整个过程轻松愉快
- 扩展功能使得代码写起来更有趣
- 空安全和不可变类型使得开发中对变量的定义和初始化倾注了更多关注
- 啊啊，我再也不用写那个findViewById了，真的爽爆有木有！

1.2 第一个Kotlin程序

Kotlin开发当然使用JetBrain系列的IDE，实际上intelliJ idea 15发布时就已经内置了Kotlin插件，更早的版本则需要到插件仓库中下载安装Kotlin插件——在安装时你还会看到有个**Kotlin Extensions for Android**，不要管他，已经过时了。安装好以后，我们就可以使用Kotlin进行开发了。

接下来我们用Android Studio创建一个Android工程，比如叫做HelloKotlin，在app目录下面的build.gradle文件中添加下面的配置：

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
ext.anko_version = '0.8.2'
ext.kotlin_version = '1.0.0'

.....
dependencies{

    .....
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    compile "org.jetbrains.anko:anko-sdk15:$anko_version"
    compile "org.jetbrains.anko:anko-support-v4:$anko_version"
    compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"

    .....
}

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

这里添加了kotlin对android的扩展，同时也添加了kotlin的gradle插件。

接下来就可以编写kotlin代码了——等等，Android Studio会帮我们生成一个MainActivity，你可以直接在菜单

Code -> Convert Java file to Kotlin file

将这个java代码转换为kotlin代码。截止到现在，你什么都不用做，程序就已经可以跑起来了。

2. 完美为Java开发者打造

2.1 通用的集合框架

我们都知道Jvm上面的语言，像什么Java、Groovy、python啥的，都是要编成虚拟机的字节码的，一旦编成字节码，在一定程度上大家就都平等了。

英雄不问出身啊

有人做过一个非常形象的比喻：Java虚拟机语言就是打群架。Kotlin正是充分利用了这一点，它自己的标准库只是基于Java的语言框架做了许多扩展，你在Kotlin当中使用的集合框架仍然跟你在Java当中一样。

举个例子，如果你想要在Kotlin中使用ArrayList，很简单，Java的ArrayList你可以随意使用，这个感觉跟使用Java没有任何区别，请看：

```
//实际上就是创建一个ArrayList
val list = arrayListOf(1, 2, 3, 4)
list.add(5)
list.remove(3)
for(item in list){
    println(item)
}
```

当然，Kotlin标准库也对这些做了扩展，我们在享用Java世界的一切资源的同时，还能比原生Java代码更滋润，真是爽爆有木有：

```
val list = arrayListOf(1, 2, 3, 4, 5)
//doubleList = [2,4,6,8,10]
val doubleList = list.map {
    it * 2
}
//oddList = [1,3,5]
val oddList = list.filter{
    it % 2 == 1
}
//将list挨个打印出来
list.forEach {
    println(it)
}
```

2.2 与Java交互

Kotlin的标准库更多的是对Java库的扩展，基于这个设计思路，你丝毫不需要担心Kotlin对Java代码的引用，你甚至可以在Kotlin当中使用Java反射，反正只要是Java有的，Kotlin都有，于是有人做出这样的评价：

Kotlin就是Java的一个扩展

这样说Kotlin显然是不公平的，但就像微信刚面世那会儿要为QQ接收离线消息一样，总得抱几天大腿嘛。

有关从Kotlin中调用Java的官方文档在此[Calling Java code from Kotlin](#)，其中最常见的就是Getter/Setter方法对应到Kotlin属性的调用，举个例子：

准备一个Java类

```
public class JavaClass {  
    private int anInt = 0;  
    public int getAnInt() {  
        return anInt;  
    }  
    public void setAnInt(int anInt) {  
        this.anInt = anInt;  
    }  
}
```

下面是Kotlin代码

```
val javaClass = JavaClass()  
javaClass.anInt = 5  
print(javaClass.anInt)
```

所以我们在Android开发时，就可以这样：

```
view.background = ...  
textView.text = ...
```

反过来在Java中调用Kotlin也毫无压力，官方文档[Calling Kotlin from Java](#)对于常见的
情况作了比较详细的阐述，这里就不再赘述。

3. 简洁，可靠，有趣

3.1 数据类

最初学Java的时候，学到一个概念叫JavaBean，当时就要被这个概念给折磨死了。明明很简单的一个东西，结果搞得很复杂的样子，而且由于当时对于这些数据类的设计概念不是很清晰，因而也并不懂得去覆写诸如equals和hashcode这样重要的方法，一旦用到HashMap这样的集合框架，总是出了问题都不知道找谁。

Kotlin提供了一种非常简单的方式来创建这样的数据类，例如：

```
data class Coordinate(val x: Double, val y: Double)
```

仅仅一行代码，Kotlin就会创建出一个完整的数据类，并自动生成相应的equals、
hashcode、toString方法。是不是早就受够了getter和setter？反正我是受够了。

3.2 空安全与属性代理

第一次见到空类型安全的设计是在Swift当中，那时候还觉得这个东西有点儿意思哈，一旦要求变量不能为空以后，因它而导致的空指针异常的可能性就直接没有了。想想每次QA提的bug吧，说少了都得有三分之一是空指针吧。

Kotlin的空安全设计，主要是在类型后面加?表示可空，否则就不能为null。

```
val anInt: Int = null // 错误
val anotherInt: Int? = null // 正确
```

使用时，则：

```

val nullable: Int? = 0
val nonNullable: Int = 2
nullable.toFloat() // 编译错误
nullable?.toFloat() // 如果null，什么都不做，否则调用toFloat
nullable!!.toFloat() // 强制转换为非空对象，并调用toFloat；如果nullable为null，抛空指针异常
nonNullable.toFloat() // 正确

```

而对于Java代码，比如我们在覆写Activity的onCreate方法时，有个参数savedInstanceState：

```
override fun onCreate(savedInstanceState: Bundle!)
```

这表示编译器不再强制savedInstanceState是否可null，开发者在覆写时可以自己决定是否可null。当然，对于本例，onCreate的参数是可能为null的，因此覆写以后的方法应为：

```
override fun onCreate(savedInstanceState: Bundle?)
```

通常来讲，教科书式的讲法，到这里就该结束了。然而直到我真正用Kotlin开始写代码时，发现，有些需求实现起来真的有些奇怪。

还是举个例子，我需要在Activity当中创建一个View的引用，通常我们在Java代码中这么写：

```

public class DemoActivity extends Activity{
    private TextView aTextView;
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        aTextView = (TextView) findViewById(R.id.a_textview);
        aTextView.setText("Hello");
        aTextView.setTextSize(20);
        ...
    }
}

```

在Kotlin当中呢？

```
class DemoActivity : Activity(){
    private var aTextView: TextView? = null
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
        //当然有更好用的方式，暂且先这么写
        aTextView = findViewById(R.id.a_textview) as TextView
        aTextView!!.text = "Hello"
        aTextView!!. textSize = 20
        ...
    }
}
```

每次用aTextView都要加俩!，不然编译器不能确定它究竟是不是null，于是不让你使用。。。这尼玛。。。到底是为了方便还是为了麻烦？？

所以后来我又决定这么写：

```
class DemoActivity : Activity(){
    private var aTextView: TextView // 编译错误，必须初始化！！！
    ...
}
```

这可如何是好？？

其实Kotlin肯定是有办法解决这个问题哒！比如上面的场景，我们这么写就可以咯：

```

class DemoActivity : Activity(){
    private val aTextView: TextView by lazy{
        findViewById(R.id.a_textview) as TextView
    }
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
        aTextView.text = "Hello"
        aTextView.setTextSize(20)
        ...
    }
}

```

lazy是Kotlin的属性代理的一个实例，它提供了延迟加载的机制。换句话说，这里的lazy提供了初始化aTextView的方法，不过真正初始化这个动作发生的时机却是在aTextView第一次被使用时了。lazy默认是线程安全的，你当然也可以关掉这个配置，只需要加个参数即可：

```

private val aTextView: TextView by lazy(LazyThreadSafetyMode.NONE){
    findViewById(R.id.a_textview) as TextView
}

```

好，这时候肯定有人要扔西红柿过来了（再扔点儿鸡蛋呗），你这lazy只能初始化val啊，万一我要定义一个var成语，又需要延迟初始化，关键还不为null，怎么办？？

```

class Demo {
    lateinit var anJsonObject: JSONObject
    fun initDemo(){
        anJsonObject = JSONObject("...")  

    }
}

```

lateinit的使用还是有很多限制的，比如只能在不可null的对象上使用，比须为var，不能为primitives（Int、Float之类）等等，不过这样逼迫你一定要初始化这个变量的做法，确实能减少我们在开发中的遗漏，从而提高开发效率。

至于lazy技术，实际上是Delegate Properties的一个应用，也就是属性代理了。在Kotlin当中，声明成员属性，除了直接赋值，还可以用Delegate的方式来声明，这个Delegate需要根据成员的类型（val或者var）来提供相应的getValue和setValue方法，比如一个可读写的Delegate，需要提供下面的方法：

```
public interface ReadWriteProperty<in R, T> {  
    /**  
     * Returns the value of the property for the given object.  
     * @param thisRef the object for which the value is requested.  
     * @param property the metadata for the property.  
     * @return the property value.  
     */  
    public operator fun getValue(thisRef: R, property: KProperty<*>): T  
    /**  
     * Sets the value of the property for the given object.  
     * @param thisRef the object for which the value is requested.  
     * @param property the metadata for the property.  
     * @param value the value to set.  
     */  
    public operator fun setValue(thisRef: R, property: KProperty<*>, value: T)  
}
```

好嘴皮不如来个栗子，下面我们就看一个自定义Delegate，用来访问SharedPreferences：

```
class Preference<T>(val context: Context, val name: String, val default: T) : ReadWriteProperty<Any?, T> {
    val prefs by lazy { context.getSharedPreferences("default",
Context.MODE_PRIVATE) }
    override fun getValue(thisRef: Any?, property: KProperty<*>)
: T {
        return findPreference(name, default)
    }
    override fun setValue(thisRef: Any?, property: KProperty<*>,
value: T) {
        putPreference(name, value)
    }
    private fun <U> findPreference(name: String, default: U): U
= with(prefs) {
        val res: Any = when (default) {
            is Long -> getLong(name, default)
            is String -> getString(name, default)
            is Int -> getInt(name, default)
            is Boolean -> getBoolean(name, default)
            is Float -> getFloat(name, default)
            else -> throw IllegalArgumentException("This type ca
n be saved into Preferences")
        }
        res as U
    }
    private fun <U> putPreference(name: String, value: U) = with
(prefs.edit()) {
        when (value) {
            is Long -> putLong(name, value)
            is String -> putString(name, value)
            is Int ->.putInt(name, value)
            is Boolean -> putBoolean(name, value)
            is Float -> putFloat(name, value)
            else -> throw IllegalArgumentException("This type ca
n be saved into Preferences")
        }.apply()
    }
}
```

需要说明的是，这段代码是我从《Kotlin for Android Developer》的示例中摘出来的。有了这个Delegate类，我们就可以完全不需要关心SharedPreference了，下面给出使用的示例代码：

```
class WhateverActivity : Activity(){
    var aInt: Int by Preference(this, "aInt", 0)
    fun whatever(){
        println(aInt)//会从SharedPreference取这个数据
        aInt = 9 //会将这个数据写入SharedPreference
    }
}
```

于是我们再也不需要重复写那些getSharedPreference，也不用edit、commit，再见那些edit之后忘了commit的日子。有没有觉得非常赞！

3.3 扩展类

扩展类，就是在现有类的基础上，添加一些属性或者方法，当然扩展的这些成员需要导入当前扩展成员所在的包才可以访问到。下面给出一个例子：

```
data class Coordinate(val x: Double, val y: Double)
val Coordinate.theta: Double
    get() {
        return Math.atan(y/x)
    }
fun Coordinate.R():Double{
    return Math.hypot(x, y)
}
```

我们已经介绍过data class，Coordinate有两个成员分别是x和y，我们知道通常表示一个二维平面，有这俩够了；然而我们在图形学当中经常会需要求得其极坐标，所以我们扩展了Coordinate，增加了一个属性theta表示角度（反正切的值域为 $-\pi/2 \sim \pi/2$ ，所以这个式子不适用于二三象限，不过这不是重点了），增加了一个R方法来获得点的半径，于是在main方法中就可以这么用：

```

fun main(args: Array<String>) {
    val coord = Coordinate(3.0, 4.0)
    println(coord.theta)
    println(coord.R())
}

```

那么这个扩展有什么限制呢？

- 在扩展成员当中，只能访问被扩展类在当前作用域内可见的成员，本例中的x和y都是public的（Kotlin默认public，这个我们后面会提到），所以可以在扩展方法和属性中直接访问。
- 扩展成员与被扩展类的内部成员名称相同时，扩展成员将无法被访问到

好的，基本知识就是这些了，下面我们再给出一个实际的例子。

通常我们在Java中会自定义一些LogUtils类来打日志，或者直接用android.util.log来输出日志，不知道大家是什么感受，我反正每次因为要输入Log.d还要输入个tag简直烦的要死，而且有时候恰好这个类还没有tag这个成员，实践中我们通常会把当前类名作为TAG，但每个类都要做这么个工作，是在是没有什么趣味可言（之前我是用LiveTemplates帮我的，即便如此也没有那种流畅的感觉）。

有了Kotlin的这个扩展功能，日子就会好过得多了，下面我创建的一个打日志的方法：

```

package com.benny.utils
import android.util.Log
inline fun <reified T> T.debug(log: Any){
    Log.d(T::class.simpleName, log.toString())
}

```

有了这个方法，你可以在任何类的方法体中直接写：

```
debug(whatever)
```

然后就会输出以这个类名为TAG的日志。

嗯，这里需要简单介绍Kotlin在泛型中的一个比较重要的增强，这个在Java中无论如何也是做不到的：inline、reified。我们再回来回头看一下debug这个方法，我们发现它可以通过泛型参数T来获取到T的具体类型，并且拿到它的类名——当然，如果你愿意，你甚至可以调用它的构造方法来构造一个对象出来——为什么Kotlin可以做到呢？因为这段代码是inline的，最终编译时是要编译到调用它的代码块中，这时候T的类型实际上是确定的，因而Kotlin通过reified这个关键字告诉编译器，T这个参数可不只是个摆设，我要把它当实际类型来用呢。

为了让大家印象深刻，我下面给出类似功能的Java的代码实现：

```
public static void debug(Class<?> clazz, Object log){
    Log.d(clazz.getSimpleName(), log.toString());
}
```

而你如果说希望在Java中也希望像下面这样拿到这个泛型参数的类型，是不可以的：

```
public static <T> void debug(Object log){
    Log.d(T.getSimpleName(), log.toString()); //错误，T是泛型参数，无法直接使用
}
```

就算我们在调用处会写道 `debug < Date >("blabla")`，但这个Date在编译之后还是会被擦除。

3.4 函数式支持 (Lambdas)

Java 8已经开始可以支持Lambda表达式了，这种东西对于Java这样一个『根红苗正』的面向对象编程语言来说还真是显得不自然，不过对于Kotlin来说，就没那么多顾忌了。

通常我们需要执行一段异步的代码，我们会构造一个Runnable对象，然后交给executor，比如这段java代码：

```
executor.submit(new Runnable(){
    @Override
    public void run(){
        //todo
    }
});
```

用Kotlin怎么写呢？

```
executor.submit({
    //todo
})
```

一下子省了很多代码。

那么实际当中我们可能更常见到下面的例子，这是一段很常见的Java代码，在Android的UI初始化会见到：

```
textView.setOnClickListener(new OnClickListener(){
    @Override
    public void onClick(View view){
        //todo
    }
});
handler.post(new Runnable(){
    @Override
    public void run(){
        //todo
    }
});
```

那么我们用Kotlin怎么写呢？

```
textView.setOnClickListener{ /*todo*/ }
handler.post{ /*todo*/ }
```

在Anko这个Android库的帮助下，我们甚至可以继续简化OnClickListener的设置方式：

```
textView.setOnClickListener{ /*todo*/ }
```

当然，好玩的不止这些，如果结合上一节我们提到的扩展方法，我们就很容易看到Kotlin的标准库提供的类似with和apply这样的方法是怎么工作的了：

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R
= receiver.block()
public inline fun <T> T.apply(block: T.() -> Unit): T { block();
return this }
```

我们通常会在某个方法体内创建一个对象并返回它，可我们除了调用它的构造方法之外还需要做一些其他的操作，于是就要创建一个局部变量。。。有了apply这个扩展方法，我们就可以这么写：

```
fun getStringBuilder: StringBuilder{
    return StringBuilder().apply{
        append("whatever")
    }
}
```

这样返回的StringBuilder对象实际上是包含"whatever"这个字符串的。

至于说Kotlin对于RxJava的友好性，使得我突然有点儿相信缘分这种东西了：

```

Observable.create<ArrayList<Dummy>> {
    it.onStart()
    try {
        it.onNext(dummyObjs)
    } catch(e: Exception) {
        it.onError(e)
    } finally {
        it.onCompleted()
    }
}.subscribe(object : Subscriber<ArrayList<Dummy>>() {
    override fun onCompleted() {
    }
    override fun onNext(t: ArrayList<Dummy>?) {
    }
    override fun onError(e: Throwable?) {
    }
})

```

3.5 Pattern Matching

记得之前在浏览 Scala 的特性时，看到：

```

object HelloScala{
    // do something
}

```

觉得很新鲜，这时候有个朋友不屑的说了句，Scala 的模式匹配才真正犀利——Kotlin 当中也有这样的特性，我们下面就来看个例子：

```

val x = 7
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")
}

```

乍一看感觉when表达式就是一个增强版的switch——Java 7以前的switch实际上支持的类型非常有限，Java 7当中增加的对String的支持也是基于int类型的——我们可以看到when不再像switch那样只匹配一个数值，它的子式可以是各种返回Boolean的表达式。

when表达式还有一种写法更革命：

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

只要是返回Boolean的表达式就可以作为when的子式，这样when表达式的灵活性可见一斑。当然，与Scala相比，**Kotlin**还是要保守一些的，下面给出一个**Scala**类似的例子，大家感受一下，这实际上也可以体现出**Kotlin**在增加**Java**的同时也尽量保持简单的设计哲学（大家都知道，毕竟Scala需要智商o(╯□╰)o）。

```
object Hello {
    def main(args: Array[String]) {
        easyMatch((1, 3))
        easyMatch(Array(1, 3, 4))
        easyMatch(Bean(3.0, 4.0))
    }
    def easyMatch(value : Any) = value match {
        case int : Int => {
            println("This is an Int.")
        }
        case (a, b) =>{
            println(s"a tuple with : $a , $b")
        }
        case Bean(x, y)  => {
            println(s"$x, $y")
        }
        case whatever => println(whatever)
    }
    case class Bean(val x: Double, val y: Double)
```

运行结果如下：

```
a tuple with : 1 , 3
[I@2d554825
3.0, 4.0
```

3.6 如果你是一个SDK开发者

我曾经做过一段时间的SDK开发，SDK的内部有很多类其实是需要互相有访问权限的，但一旦类及其成员是public的，那么调用方也就可以看到它们了；而protected或者default这样的可见性对于子包却是不可见的。

用了这么久Java，这简直是我唯一强烈感到不满的地方了，甚至于我突然明白了C++的friend是多么的有用。

Kotlin虽然没有提供对于子包可见的修饰符，不过它提供了internal：即模块内可见。换句话说，internal在模块内相当于public，而对于模块外就是private了——于是乎我们如果开发SDK，那么可以减少api层的编写，那些用户不可见的部分直接用internal岂不更好。当然有人会说我们应当有proguard做混淆，我想说的是，proguard自然是要用到的，不过那是SDK这个产品加工的下一个环节了，我们为什么不能在代码级别把这个事情做好呢？

关于Kotlin的默认可见性究竟是哪个还有人做出过讨论，有兴趣的可以参考这里：[Kotlin's default visibility should be internal](#)。

3.7 DSL

其实我们对DSL肯定不会陌生，gradle的脚本就是基于groovy的DSL，而Kotlin的函数特性显然也是可以支持DSL的。比如，我们最终要生成下面的xml数据：

```
<project version="4">
    <component name="Encoding">
        <file url="PROJECT" charset="UTF-8" />
    </component>
</project>
```

我们可以构建下面的类：

```
class Project {  
    var version: String? = null  
    get() =  
        if (field == null) ""  
        else {  
            " version=\\"${field}\\""  
        }  
    lateinit private var component: Component  
    fun component(op: Component.() -> Unit) {  
        component = Component().apply {  
            op()  
        }  
    }  
    override fun toString(): String {  
        return "<project${version}>${component}<project>"  
    }  
}  
fun project(op: Project.() -> Unit): Project {  
    return Project().apply {  
        op()  
    }  
}  
class Component {  
    var name: String? = null  
    get() =  
        if (field == null) ""  
        else {  
            " name=\\"${field}\\""  
        }  
    lateinit private var file: File  
    fun file(op: File.() -> Unit) {  
        file = File().apply {  
            op()  
        }  
    }  
    override fun toString(): String {  
        return "<component${name}>${file}<component>"  
    }  
}
```

```

}

class File {
    var url: String? = null
        get() =
            if (field == null) ""
            else {
                " url=\">${field}\\""
            }
    var charset: String? = null
        get() =
            if (field == null) ""
            else {
                " charset=\">${field}\\""
            }
    override fun toString(): String {
        return "<file${url}${charset}>"
    }
}

fun main(args: Array<String>) {
    val xml = project {
        version = "4"
        component {
            name = "Encoding"
            file {
                url = "PROJECT"
                charset = "UTF-8"
            }
        }
    }
    println(xml)
}

```

我们看到在main方法当中，我们用kotlin定义的dsl写出了一个Project对象，它有这与xml描述的一致的结构和含义，如果你愿意，可以构造相应的方法来输出这样的xml，运行之后的结果：

```

<project version="4"><component name="Encoding"><file url="PROJE
CT" charset="UTF-8"/><component><project>

```

当然，这个例子做的足够的简陋，如果你有兴趣也可以抽象出”Element”，并为之添加”Attributes”，实际上这也不是很难。

3.7 Kotlin与Android的另一些有趣的东西

写了很多代码，却发现它们干不了多少事情，终究还是会苦恼的。比如我一直比较痛苦的一件事儿就是：

```
Button button = (Button) findViewById(R.id.btn);
```

如果我需要很多个按钮和图片，那么我们要写一大片这样的findViewById。。妈呀。。。这活我干不了啦。。

不过用Kotlin的Android扩展插件，我们就可以这样：

先上布局文件：

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/open_bj"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView"
        android:text="Hello"
        android:textSize="50sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:id="@+id/start"
        android:clickable="false"
        android:layout_gravity="center_horizontal"
        android:background="@drawable/start_selector"
        android:textSize="50sp"
        android:layout_marginTop="20dp"
        android:layout_marginBottom="200dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</RelativeLayout>
```

在Activity中：

```

package com.benny

.....
import kotlinx.android.synthetic.main.load_activity.*
import org.jetbrains.anko.onClick
import org.jetbrains.anko.startActivity
import org.jetbrains.anko.toast

.....
class LoadActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)
        start.onClick {
            toast("开始")
            startActivity<AnotherActivity>()
        }
        textView.text = "你好"
    }
}

```

注意到：

```
import kotlinx.android.synthetic.main.load_activity.*
```

导入这一句之后，我们就可以直接在代码中使用start、textView，他们分别对应于main.xml中的id为start的按钮和id为textView的TextView。

于是你就发现你再也不用findViewById了，多么愉快的一件事！！！当然，你还会发现Toast的调用也变得简单了，那其实就是一个扩展方法toast()；而startActivity呢，其实就是一个inline加reified的应用——这我们前面都提到过了。

还有一个恶心的东西就是UI线程和非UI线程的切换问题。也许你会用handler不断的post，不过说真的，用Handler的时候难道你不颤抖么，那可是一个很容易内存泄露的魔鬼呀~哈哈，好吧其实我不是说这个，主要是用handler写出来的代码实在太丑了！！

原来在java当中，我们这么写：

```

handler.post(new Runnable(){
    @Override
    public void run(){
        //todo
    }
});
MainActivity.this.runOnUiThread(
    public void run(){
        //todo
    }
);

```

而在Kotlin当中呢，我们只需要这么写：

```

async() {
    //do something asynchronously
    uiThread {
        //do something on UI thread
    }
}

```

自己感受一下吧。

下面我们再来提一个有意思的东西，我们从做Android开发一开始就要编写xml，印象中这个对于我来说真的是一件痛苦的事情，因为它的工作机制并不如代码那样直接（以至于我现在很多时候居然喜欢用Java代码直接写布局）——当然，最主要的问题并不是这个，而是解析xml需要耗费CPU。Kotlin有办法可以解决这个问题，那就是DSL了。下面给出一个例子：

```

linearLayout {
    button("Login") {
        textSize = 26f
    }.lparams(width = wrapContent) {
        horizontalMargin = dip(5)
        topMargin = dip(10)
    }
}

```

一个LinearLayout包含了一个Button，这段代码你可以直接写到你的代码中灵活复用，就像这样：

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(linearLayout {  
        button("This is a button") {  
            onClick {  
                toast("clicked!")  
            }  
        }.lparams {  
            width = matchParent  
            verticalMargin = dip(5)  
        }  
    })  
}
```

这样做的好处真是不少：

- 比起xml的繁琐来，这真是要清爽很多
- 布局本身也是代码，可以灵活复用
- 再也不用findViewById了，难道你不觉得在这个上面浪费的生命已经足够多吗
- 事件监听很方便的嵌到布局当中
- DSL方式的布局没有运行时的解析的负担，你的逻辑代码怎么运行它就怎么运行
- Anko还增加了更多好玩的特性，有兴趣的可以参考：[Anko@Github](#)

3.8 方法数之痛

我曾经尝试用Scala写了个Android的HelloWorld，一切都配置好以后，仅仅引入了Scala常见的几个库，加上support-v4以及appcompat这样常见的库，结果还是报错了。是的，65K。。。而且用Scala开发Android的话，基于gradle的构建会让整个app的build过程异常漫长，有时候你会觉得自己悟出了广义相对论的奥义，哦不，你一定是晕了，时间并没有变慢。

相比之下，Kotlin的标准库只有7000个方法，比support-v4还要小，这正反映了Kotlin的设计理念：100% interoperable with Java。其实我们之前就提到，Java有的Kotlin就直接拿来用，而Scala的标准库要有5W多个方法，想想就还是想想算了。

4. 小结

目前Kotlin 1.0已经release，尽管像0xffffffff识别成Long类型这样的bug仍然没有解[详情](#)：

```
val int: Int = 0xffffffff // error
val anotherInt: Int = 0xffffffff.toInt() // correct
```

不过，Kotlin的教学资源和社区建设也已经相对成熟，按照官方的说法，Kotlin可以作为生产工具投入开发，[详情可以参考：Kotlin 1.0 Released: Pragmatic Language for JVM and Android](#)。

敢于吃螃蟹，多少有些浪漫主义色彩，我们这些程序员多少可以有些浪漫主义特质，不过在生成环境中，稳定高于一切仍然是不二法则。追求新技术，一方面会给团队带来开发和维护上的学习成本，另一方面也要承担未来某些情况下因为对新技术不熟悉而产生未知问题的风险——老板们最怕风险了~~

基于这一点，毫无疑问，Kotlin可以作为小工具、测试用例等的开发工具，这是考虑到这些代码通常体量较小，维护人数较少较集中，对项目整体的影响也较小；而对于核心代码，则视情况而定吧。

就我个人而言，长期下去，Kotlin很大可能会成为我的主要语言，短期内则仍然采用温和的改革方式慢慢将Kotlin渗透进来。

一句话，**Kotlin**是用来提升效率的，如果在你的场景中它做不到，甚至成了拖累，请放开它。

你为什么需要 Kotlin

1. 往事

曾经你有段时间研究 IntelliJ 的插件开发，企图编译 IntelliJ Idea Community Edition (ICE) 的源码，结果发现有个奇怪的东西让你的代码无法编译...什么鬼，kt 是什么玩意儿？

file extension .kt ? What kind of file is this? [closed]

▲ I am currently working on a project in which I came across files with extension .kt I am not able to figure out which tool can I use to know the contents of the file. I would be glad to if you can tell what .kt file actually mean and when is it used ? Thanks in advance.
2 share edit flag

asked Jul 6 '13 at 23:50
N mol
116 • 5 • 17

▲ it might be from/for Kotlin: <http://kotlin.jetbrains.org/> it's a source code file for a programming language which "is a statically typed programming language that compiles to JVM byte codes and JavaScript." (according to their website)
8 share edit flag

answered Jul 7 '13 at 0:00
ZCoder
237 • 1 • 7



怎么又有新语言出来啊，简直要疯掉了。这时候，你的脑海里面瞬间浮现出了这句话：

有困难要上，没困难制造困难也要上。

『靠，这尼玛究竟是谁说的，好有道理！』你调侃道。

为了不丢掉社会主义新青年勤奋刻苦的优良传统，你决定学一下 Kotlin，不过说真的，这决定也是坑苦了自己，毕竟那段时间 Kotlin 的 API 还没有趋于稳定，经常从网上找到个 demo，搞到本地就编不过去，哭死。直到 2016 年 2 月，Kotlin 1.0 正式发布，凌乱的 API 也随着曾经躁动的心的平静而稳定下来，你无需再忍受什么，甚至还有了一种『终于看着娃长大了』的感觉。

2. 消失的 Getter 和 Setter

你一直很喜欢 Java，就像你一直喜欢喝黑咖啡一样。原因也很朴实，因为你别的也不怎么会啊。可有一阵子做一个语音聊天的 app，里面各种用户、通话记录等等的数据结构，简直了，写起来长长的一串，光 Getter 和 Setter 就一眼望不到边，每写一个数据结构类，仿佛眼前就是那金黄色的稻田，你吹一口咖啡，它们居然还簌簌作响。最讽刺的是，你发现大家在写有关 Java 的文章的时候，遇到数据结构实体类，经常会这样写：

```
public class Person{
    private int id;
    private String name;
    //瞅啥瞅，省略掉的是 getter 和 setter !
    ...
}
```

额，这么重复的代码居然占用了 80% 的篇幅，也难怪是个 IDE 就有帮忙生成 Getter 和 Setter 模板的功能。

你说你曾经试图不写 Getter 和 Setter，可作为一个写 Java 这么多年的人，没了 Getter 和 Setter 让你感觉就像是。。。



那时候你看到 C# 里面的属性也真是眼馋呐，『怎么 Java 就不能搞这么个特性呢？』

后来，你发现 Kotlin 居然是有属性的：

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

你眼前一亮，放下手中的咖啡，幻想着啥时候去 Kotlin 小岛玩一趟。



3. 又见空指针

你的项目都接入了 Bugly，那赶脚就好像在鸡蛋上跳舞哇，每天去打开崩溃统计都心惊肉跳的。这些 crash 里面绝大多数都是空指针异常，这倒不是说空指针本身有什么问题，空指针只能说明程序有考虑不周的情形出现，出现空指针调用通常都是代码的编写问题，那么为什么 Java 会允许潜在的空指针存在呢？你望着窗外，思索着一定是什么蒙蔽了你的双眼，让你看不透，望不穿，寻不见。

```
person.setName("橘右京");
```

回过神来，突然看到这行代码，你嘟囔着，这哥们名字叫 橘右京，可这哥们是谁？万一他是个 null 呢？



```
Person person = findPersonFromCacheOrCreate();
person.setName("橘右京");
```

你的目光往上移，看到了上面那行赋值，你开始怀疑，开始问自己：『害怕不。。其实我们一直都没有意识到，我们的 Java 代码 处处不让我们放心。』

说来也巧，你最近在梳理项目代码的时候，见到的最多的就是：

```
if(x != null) x.y();
```

你对你的代码充满了不信任，你的代码会回报你什么呢？

『虚拟机空指针全家桶来一份！。。。那一定很好吃吧』你一脸无奈的自嘲着。



你突然发现这代码居然看上去跟窗外的世界很像，只是，给代码用的净化器应该是什么牌子的呢？

『小米的怕是不行了吧。』你哈哈一笑，似乎对此感到很开心。

那开心转瞬即逝，你不得不面对这令人苦恼的现状。你在 Java 当中除了对自己说『我保证 findPersonFromCacheOrCreate() 不会返回空』，还有什么更让人踏实的办法么？当然没有。

『看看 Kotlin 有没有好办法吧！』于是你尝试着用 Kotlin 写下了类似的代码：

```
fun findPersonFromCacheOrCreate(): String{  
    ...  
}
```

当你企图在这个方法中返回 null 时，聪明的 IntelliJ 立即在你的代码上画出红线，告诉你不要这样。

你查了下资料，发现原来在 Kotlin 当中，String 表示一个不可为 null 的字符串类型。这一刻，你的内心感到无比踏实：

```
val person = findPersonFromCacheOrCreate()
person.name = "橘右京"
```

写 `findPersonFromCacheOrCreate` 这个方法的人必须给你保证返回的 `person` 不为 `null`，他在编写这个方法的时候就要百般小心，不然编译器就要削他了。

『找削袄！』你突然想起微信里面的那套桃子一家东北话的表情，想想就好欢乐。



『不过』，你又想，『万一它还是给我返回了 `null` 怎么办呢？』说着你按照 Kotlin 的要求改了下代码：

```
fun findPersonFromCacheOrCreate(): String?{
    ...
}
```

结果发现下面的第二行报错。

『什么情况？』你不明白了。

```
val person = findPersonFromCacheOrCreate()
person.name = "橘右京"
```

Only safe (?) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?

『就是说，如果有人胆敢给我返回个可空类型，我必须做判空处理才可以行呗？』你眼前一亮，旋即惊呼道，『太厉害了！』

紧接着你发现，虽然返回的是可空类型，但这丝毫不会影响你与你的代码谈笑风生，因为 Kotlin 可以给你一千种选择让你的代码看起来犹如行云流水一般，比如你希望拿到 `null` 直接返回，你就这么写：

```
val person = findPersonFromCacheOrCreate()?:return
```

或者这里没办法用 return，你一样可以确保 person 不为空的时候执行你的逻辑：

```
val person = findPersonFromCacheOrCreate()
person?.let{
    //在这里直接用 it 指代 person，绝对不为空
}
person?.setName("安琪拉")//只有person不为空的时候执行
```



厉害了我的哥

4. Smart Cast

『听说风已经到张家口了。』亚瑟说。

『哦是么，那我们可真得拭目以待呀。』你淡定地说。

『待什么，眼前一亮吗，哈哈。』也乐了。

『是啊，不过，眼下我倒更希望 Java 能聪明一点儿。。』

『它怎么了？』

『你看，Java 有一种特别缺心眼的写法：

```
if(view instanceof ViewGroup){  
    ((ViewGroup) view).addView(child);  
}
```

强转加方法调用，两对括号，写到手抽筋啊。可我已经告诉 Java view 是 ViewGroup 了啊，结果还是要强转，这种感觉就像我坐地铁的时候本来刷卡进站，结果到了车上，还有人查票！』说完，你忽然觉得口渴，随手拿起热乎的咖啡喝了一口，你眉头一皱，那味道似乎不太好。

『虽然我们写代码应该尽量避免强转，可你明知道这东西我们无法避免，于是本来想多态的用父类或者接口引用实例，结果强转代码写得多到变态。要是我在判断了 view 的类型之后，在这个类型判断有效的作用域内不用做强转就好了。』你接着说道，一脸的烦恼。

『Kotlin 可以呀，Kotlin 有个特性叫 Smart-Cast，你写的代码就可以像这样：

```
if(view is ViewGroup){  
    view.addView(child) // 现在 Kotlin 已经知道 view 是 ViewGroup  
    类型了！  
}
```

咋样，厉害吧？』亚瑟一脸神气的表情。

『真的呀！』你兴奋的说。

『这你都不知道？』在一旁打游戏的太2真人一如既往地调侃道，他看上去似乎比亚瑟更得意。



5. 打日志

『你们连日志都打，真不要脸。。对了，日志是谁，打的时候叫上我啊！』你露出一脸的坏笑。

原来，你有个函数传入了三个参数，

```
void check(ArrayList<String> list, String tag, int id);
```

你想把他们的值打印一下，于是你不假思索地敲出了一行代码：

```
System.out.println("list: "+list.size()+"; tag="+tag+"; id="+id);
```

这样的语句并没有引起你的任何不适——毕竟，你早已习惯了它的丑陋。你稍稍停顿，活动了一下手指，突然想到那个经久不衰的段子：

女神：你能让这个论坛的人都吵起来，我今晚就跟你走。

程序猿：PHP语言是最好的语言！

论坛炸锅了，各种吵架。

女神：服了你了，我们走吧，你想干啥都行。

程序猿：今天不行，我一定要说服他们，PHP语言是最好的语言。

『难怪 PHP 是最好的语言，没有之一呢！』你自嘲着，『至少人家支持字符串模板呀。』

```
$size = count($list);
echo "list: $size; tag=$tag; id=$id";
```

说来也是好奇，这么好的特性 Kotlin 有没有呢？这家伙最近给你带来的惊喜是在太多了，于是你决定试一试。

```
println("list: ${list.size}; tag=$tag; id=$id")
```

『嗯——』你满意的点了点头。

6. 再见，Utils

你肯定用过 String，不仅如此，你还知道 String 居然连个 empty 方法都没舍得提供，每次都要写 str.equals("")，真是无比丑陋。这还算好的，如果恰好要判断不为空呢？这个应该更常用吧，于是你就会写 !str.equals("")，搞好几个字符敲完了发现，尼玛，还得加个小括号 (!str.equals(""))...

『怎么可以这么反人类！！！』你嘟囔着，一气之下写了个工具类：

StringUtils.java

```
public class StringUtils{
    public static boolean notEmpty(String str){
        return !"\"".equals(str);
    }
    public static boolean isEmpty(String str){
        return "\"".equals(str);
    }
}
```

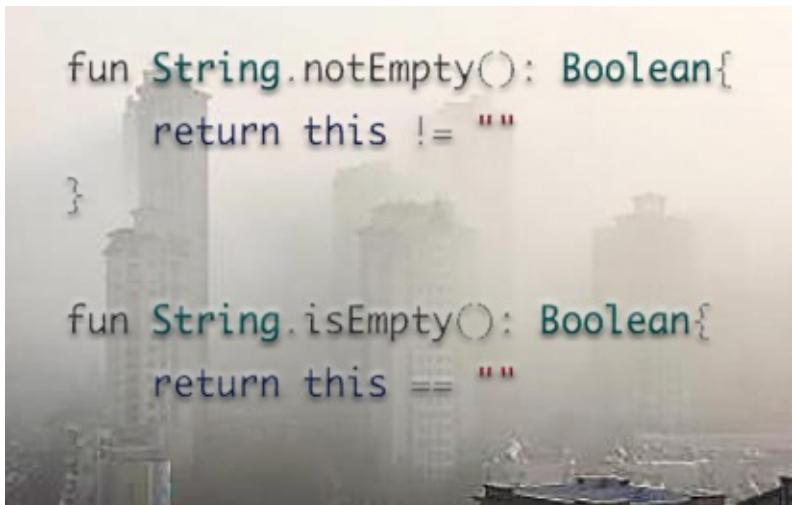
你满意的点点头，觉得这样就再也不用忍受那些丑陋了。

随后的某一天，阳光依旧试图穿过层层雾霾照到你那北向的屋子的楼的正面，无果，于是垂头丧气的点了一支烟。而你呢，正要写下一句叫做 if(StringUtils.isNotEmpty(str)){ ... } 的代码，你很快地写完 if(str. 疲惫的双眼企盼着 IDE 意会你小眼神，可它蒙了，String 并不曾有一个叫做 notEmpty 的方法啊。这时候你迟疑了一下，缓过神来，灵活的把右手小拇指从键盘上的 ; 移到 backspace，删掉 str. 重新写出 if(StringUtils.isNotEmpty(str))。多么令人苦恼的经历啊，于是你惆怅的点了一支烟，屋里的空气净化器也开始飞速的转了起来。



『我要是能重写一下 Java 的 String 类好了，我一定先给它加上这俩方法！！』

这时，只见一道亮光闪过，你的窗户上映出了几行字：



你惊喜的差点儿喊出声来。『这真的是 Kotlin 吗？』你有点儿不敢相信自己的眼睛。是的，有了扩展方法，你再也不需要什么 XXXUtils 了。

7. 晚安，ButterKnife

『晚安。』你轻轻地对 ButterKnife 说到。你知道这也许是最后一次这样说了，毕竟在 Kotlin 的世界里，ButterKnife 开始变得有些不知所措。

『你不需要我了。』ButterKnife 有些疲惫。

『不，你是最棒的。』突如其来的一句话，让你显得有些慌乱。

『kotlin-android-extensions 还不够么？』ButterKnife 不耐烦了。

『可..可是...』你不知道该怎么回答了，毕竟在 Kotlin 出现之后，你很少提及 ButterKnife 了。

是啊，过去你的 View 都是用 ButterKnife 注入的：

```
@BindView(R.id.nameView) TextView nameView;  
...  
nameView.setText("橘右京");  
...
```

而现在呢？你摇摇头，显得有些无奈。『再不需要注入 View 了是么？』你在问自己，尽管对过去百般不舍，可你还是很欣赏你的代码现在的样子：

```
nameView.text = "橘右京"
```

再也不需要 ButterKnife 了，更不需要什么 findViewById 了——这就是命。

8. 请和我的代理谈吧

听说有人把 SharedPreference 先生告上法庭了，说他总是在接受新值之后不去做持久化，SP 先生却觉得很委屈：

『先生们，大家都是绅士，我实在想不出什么能比我的信誉更重要。如果你们在给我们更新值的时候调用 commit，我们一定会按照约定完成持久化的。可你们为什么就不愿意 commit 呢？』 SP 先生大惑不解。

『请问 SP 先生，我是《Dalvik 日报》记者，我想问一下，为什么必须要 commit 呢？』

『您好，这是规定。』 SP 先生慢条斯理的回答道。

『可这有点儿反人类呀！』 记者追问道。

『没关系，基于此次诉讼事件的教训，我们特意引进了高科技人才 Preference< T > 先生，P 先生在这方面可是行家了。』

记者打量着这位 P 先生，试图从他身上发现点儿什么。

『哈哈，P 先生是一位 Delegate 吧，SP 先生就是 Delegated Properties 咯~』 你从人群中走出来，淡定的说道。

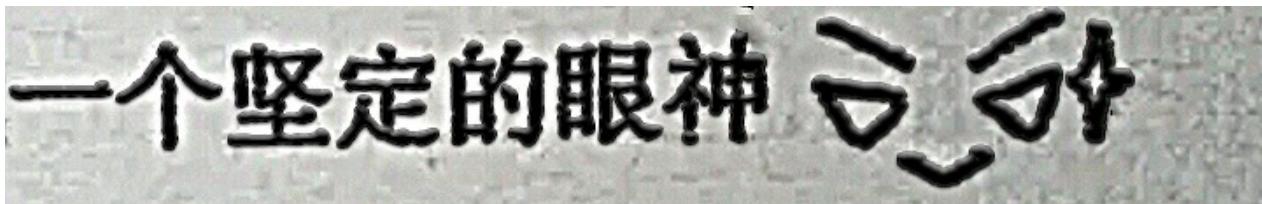
『不错，』 P 先生扶了扶眼镜，『这位先生，SP 先生的操作方案是有些繁琐，我们来看个例子：

```
context.getSharedPreferences("name", Context.MODE_PRIVATE)
    .edit().putString("key", "value").putInt("intKey", intValue)
    .commit();
```

一位绅士需要操作它的时候，总是首先要获取 SharedPreferences 实例，接着 edit 拿到 Editor 实例才能存入值，一方面存入的值存在随意性，key 的值必须约束好才行，否则读取方就无法获取到值，另一方面只有 commit 之后值才可以被存入。这样操作起来确实不是很友好。』

『那 P 先生高见？』 你很好奇。

『从今天起，大家如果有需要 SP 先生持久化数据的需求，只需要在我这里登记一次，剩下的，大家只需要像读写变量一样操作即可生效。』 P 先生停顿了一下，打量了一下四周，大家都在注视着他，而一旁的 SP 先生也给了他一个坚定的眼神。



『那么以后，如果有位绅士需要我们，比如他需要持久化的数据名叫“name”，值叫“橘右京”，当然这个值也是可以修改的，那么他只需要这样操作：

```
var name by Preference(context, "name", "橘右京", "sp_name")  
...  
Log.d(TAG, name) // 第一次读取，只能读取到默认值，那就是 橘右京  
name = "不知火舞"  
Log.d(TAG, name) // 这里输出的就是 不知火舞 啦
```

我们真正做到了读写持久化数据就如同读写内存变量一样简单直接。』

『真的好赞。』你不禁鼓起掌来。『那 P 先生，我能读一下你的源码么？』

.....

突然，你的手机振动了一下，打断了你的思绪。你从沉思中回来，发现你眼前不过仍然是你的 IDE，而屏幕上的这段代码，正是 P 先生的源码。真的是太巧妙了：

```
class Preference<T>(val context: Context, val name: String, val  
default: T, val prefName: String = "default") : ReadWriteProperty  
<Any?, T> {  
    val prefs by lazy { context.getSharedPreferences(prefName, C  
ontext.MODE_PRIVATE) }  
    override fun getValue(thisRef: Any?, property: KProperty<*>)  
: T {  
        return findPreference(name, default)  
    }  
    override fun setValue(thisRef: Any?, property: KProperty<*>,  
value: T) {  
        putPreference(name, value)  
    }  
}
```

```
private fun <U> findPreference(name: String, default: U): U
= with(prefs) {
    val res: Any = when (default) {
        is Long -> getLong(name, default)
        is String -> getString(name, default)
        is Int -> getInt(name, default)
        is Boolean -> getBoolean(name, default)
        is Float -> getFloat(name, default)
        else -> throw IllegalArgumentException("This type ca
n be saved into Preferences")
    }
    res as U
}
private fun <U> putPreference(name: String, value: U) = with
(prefs.edit()) {
    when (value) {
        is Long -> putLong(name, value)
        is String -> putString(name, value)
        is Int ->.putInt(name, value)
        is Boolean -> putBoolean(name, value)
        is Float -> putFloat(name, value)
        else -> throw IllegalArgumentException("This type ca
n be saved into Preferences")
    }.apply()
}
}
```

『Impressive.』你一副恋恋不舍的样子。『SP 先生的全名应该不是 SharedPreferences 吧，我觉得他们一定是搞错了，应该是 SophisticatedPreferences 还差不多。P 先生自然就是简化的它咯。』

9. 壮士一去兮为啥不复还？

这个问题你想了很久。是荆轲的匕首不够快？还是不够长？

『总之是不好用呗。』你嫌弃地说。

Java 里面也有一副利刀，叫做 Dagger，这把利刀可以帮你生成一些代码。

『如果代码可以聪明到自己写代码，那我们不就要失业了吗？』不知道你说这话是在调侃，还是感到有些恐慌。



Kotlin 之前是无法使用这把利刃的，这可能真的打击了不少人的积极性。不过，这已经不是问题了，因为你在前不久读到 Kotlin 1.0.4 的更新说明的时候，就已经发现 kapt 的存在。只要你添加 apply plugin: "kotlin-kapt" 这句配置，你就可以像在 Java 当中一样使用 Dagger 了——你甚至还做了个 demo 试了一下，程序员嘛，总是无法摆脱成功写出一个 Hello World 程序时获得的内心的愉悦感。

『似乎除了 FindBugs 之类与 Java 语法紧密结合的框架不能直接应用到 Kotlin 上，别的都没有什么问题哎。』你似乎发现了什么。

Java 和 Kotlin 的对话 1

『Java 叔叔，我。。。我怕。。。』Kotlin 怯懦的说。

『有叔叔在呢。』Java 拍着胸脯，安慰道。『世界是你们的，也是我们的，但是归根结底是你们的。你们青年人朝气蓬勃，正在兴旺时期，好像早晨八九点钟的太阳。希望寄托在你们身上。孩子，放手去干吧，搞不定的，找叔叔，叔叔虽然老了，但叔叔还是有自信能帮你搞定。』

你突然回过神来，『我脑洞怎么这么大，哈哈哈~』

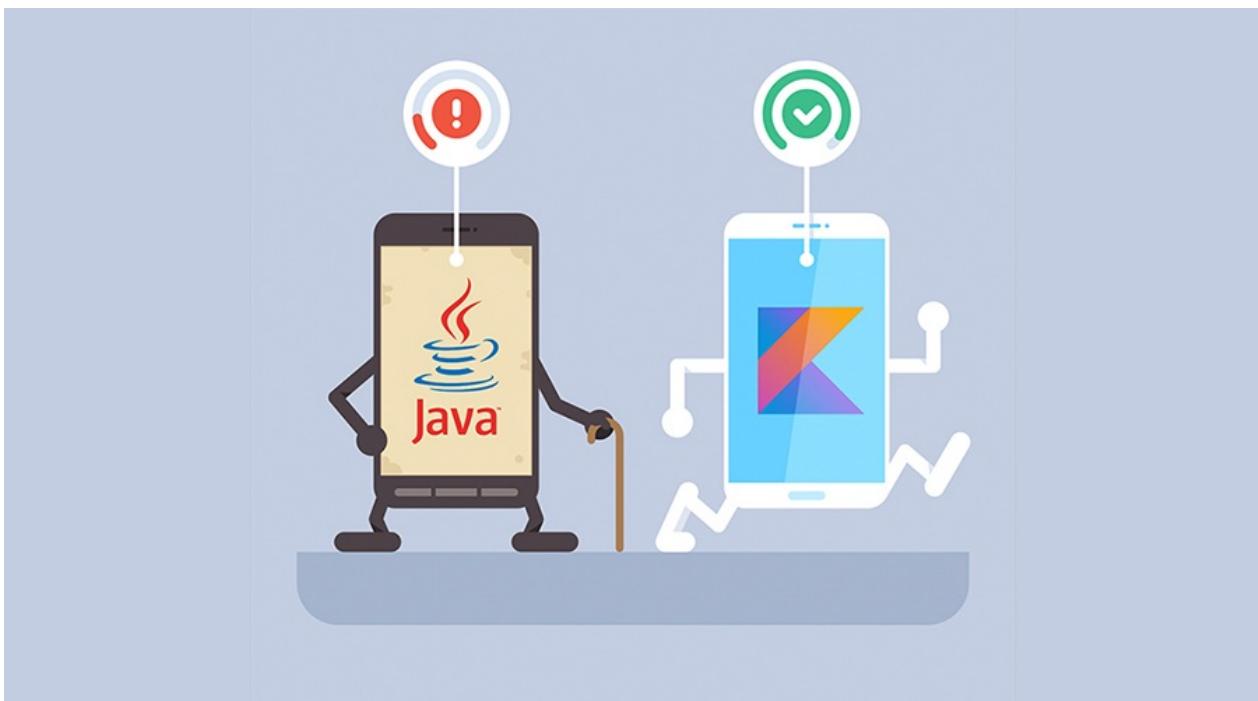
Java 和 Kotlin 的对话 2

『Java 叔叔，我。。。我怕。。。』Kotlin 怯懦的说。

『有叔叔在呢。』Java 拍着胸脯，安慰道。『世界是你们的，也是我们的，但是归根结底是你们的。你们青年人朝气蓬勃，正在兴旺时期，好像早晨八九点钟的太阳。希望寄托在你们身上。孩子，放手去干吧，搞不定的，找叔叔，叔叔虽然老了，但叔叔还是有自信能帮你搞定。』

『好的，叔叔，那我就不客气了！』Kotlin 嬉皮笑脸的样子真得很像个小孩子。

『我去，居然装可怜骗你叔叔，看我不打你！』Java 气急败坏道。



『完了，脑洞合不上了。。。』你已经陶醉了其实。

在你看来，Kotlin 似乎并不是一门新的编程语言，它看上去更像 Java 的语法糖，只不过，这糖放的彻底了些。

『嗯。。。挺甜~』你喝了口咖啡，不过这次不同以往，你放了糖。

为什么我要改用Kotlin

原文链接：<http://droidyue.com/blog/2017/05/18/why-do-i-turn-to-kotlin/>

写在前面的话，作为一个不熬夜的人，一觉醒来发现**Kotlin**成为了**Android**的官方语言，可谓是大喜过望。为了趁热打铁，我决定提前三天放出原定本周日**Release**的文章。希望能及时让大家了解一下**Kotlin**。

相信很多开发人员，尤其是**Android**开发者都会或多或少听说过**Kotlin**，当然如果没有听过或者不熟悉也没有关系。因为本篇文章以及博客后期的内容会涉及到很多关于**Kotlin**的知识分享。

在写这篇文章前的一个多月，**Flipboard**中国的**Android**项目确定了正式将**Kotlin**作为项目开发语言，这就意味着新增的代码文件将以**Kotlin**代码格式出现，而且同时旧的**Java**代码也将会陆陆续续翻译成**Kotlin**代码。在使用**Kotlin**的这段时间，被它的简洁，高效，快捷等等特点震撼，所以有必要写一篇文章来谈一谈**Kotlin**的特性，如若能取得推广**Kotlin**的效果则倍感欣慰。

Kotlin的“简历”

- 来自于著名的IDE IntelliJ IDEA(**Android Studio**基于此开发) 软件开发公司
JetBrains(位于东欧捷克)
- 起源来自JetBrains的圣彼得堡团队，名称取自圣彼得堡附近的一个小岛(Kotlin Island)
- 一种基于JVM的静态类型编程语言

来自知名的工具开发商**JetBrains**，也就决定了**Kotlin**的基因中必然包含实用与高效等特征。那我们接下来看一看**Kotlin**的特点，当然这也是我改用**Kotlin**的重要原因。

语法简单，不啰嗦

```
//variables and constants
var currentVersionCode = 1    //变量当前的版本号，类型Int可以根据值推断出来
var currentVersionName : String = "1.0" //显式标明类型
val APPNAME = "droidyue.com" //常量APPNAME 类型(String)可以根据值推断出来

//methods
fun main(args: Array<String>) {
    println(args)
}

// class
class MainActivity : AppCompatActivity() {

}

// data class 自动生成getter, setting, hashCode和equals等方法
data class Book(var name: String, val price: Float, var author: String)

//支持默认参数值，减少方法重载
fun Context.showToast(message: String, duration:Int = Toast.LENGTH_LONG) {
    Toast.makeText(this, message, duration).show()
}
```

- Kotlin支持类型推断，没有Java那样的啰嗦。
- 另外用 `var` 表示变量，`val` 表示常量更加的简洁
- 方法也很简单，连function都缩写成了fun，平添了几分双关之意。
- 类的继承和实现很简单，使用:即可
- Kotlin每个句子都不需要加分号(;)

空指针安全

空指针（NullPointerException或NPE）是我们使用Java开发程序中最常见的崩溃了。因为在Java中我们不得不写很多防御性的代码，比如这样

```
public void test(String string) {  
    if (string != null) {  
        char[] chars = string.toCharArray();  
        if (chars.length > 10) {  
            System.out.println(((Character)chars[10]).hashCode())  
        };  
    }  
}
```

在Kotlin中空指针异常得到了很好的解决。

- 在类型上的处理，即在类型后面加上?，即表示这个变量或参数以及返回值可以为null，否则不允许为变量参数赋值为null或者返回null
- 对于一个可能是null的变量或者参数，在调用对象方法或者属性之前，需要加上?，否则编译无法通过。

如下面的代码就是Kotlin实现空指针安全的一个例子，而且相对Java实现而言，简直是一行代码搞定的。

```
fun testNullSafeOperator(string: String?) {  
    System.out.println(string?.toCharArray()?.getOrNull(10)?.has  
    hCode())  
}  
  
testNullSafeOperator(null)  
testNullSafeOperator("12345678901")  
testNullSafeOperator("123")  
  
//result  
null  
49  
null
```

关于空指针安全的原理，可以参考这篇文章[研究学习Kotlin的一些方法](#)

支持方法扩展

很多时候，Framework提供给我们的API往往都时比较原子的，调用时需要我们进行组合处理，因为就会产生了一些Util类，一个简单的例子，我们想要更快捷的展示Toast信息，在Java中我们可以这样做。

```
public static void longToast(Context context, String message) {  
    Toast.makeText(context, message, Toast.LENGTH_LONG).show();  
}
```

但是Kotlin的实现却让人惊奇，我们只需要重写扩展方法就可以了，比如这个longToast方法扩展到所有的Context对象中，如果不去追根溯源，可能无法区分是Framework提供的还是自行扩展的。

```
fun Context.longToast(message: String) {  
    Toast.makeText(this, message, Toast.LENGTH_LONG).show()  
}  
applicationContext.longToast("hello world")
```

注意：Kotlin的方法扩展并不是真正修改了对应的类文件，而是在编译器和IDE方面做得处理。使我们看起来像是扩展了方法。

Lambda, 高阶函数，Streams API, 函数式编程支持

所谓的Lambda表达式是匿名函数，这使得我们的代码会更加的简单。比如下面的代码就是lambda的应用。

```
findViewById(R.id.content).setOnClickListener {  
    Log.d("MainActivity", "it was clicked")  
}
```

所谓的高阶函数就是

- 可以接受函数作为参数
- 也可以返回函数作为结果

举一个接受函数作为参数的例子。在Android开发中，我们经常使用SharedPreferences来存储数据，如果忘记调用apply或者commit则数据修改不能应用。利用Kotlin中的高阶函数的功能，我们能更好的解决这个问题

```
fun SharedPreferences.editor(f: (SharedPreferences.Editor) -> Unit) {
    val editor = edit()
    f(editor)
    editor.apply()
}

//实际调用
PreferenceManager.getDefaultSharedPreferences(this).editor {
    it.putBoolean("installed", true)
}
```

当然这上面的例子中我们也同时使用了方法扩展这个特性。

Kotlin支持了Streams API和方法引用，这样函数式编程更加方便。比如下面的代码就是我们结合Jsoup，来抓取某个proxy网站的数据，代码更加简单，实现起来也快速。

```
fun parse(url: String): Unit {
    Jsoup.parse(URL(url), PARSE_URL_TIMEOUT).getElementsByClass(
        "table table-sm")
        .first().children()
        .filter { "tbody".equals(it.tagName().toLowerCase()) }
        .flatMap(Element::children).forEach {
            trElement ->
            ProxyItem().apply {
                trElement.children().forEachIndexed { index, ele
ment ->
                    when (index) {
                        0 -> {
                            host = element.text().split(":")[0]
                            port = element.text().split(":")[1].
                               ToInt()
                        }
                        1 -> protocol = element.text()
                        5 -> country = element.text()
                    }
                }
            }.let(::println)
        }
}
```

字符串模板

无论是Java还是Android开发，我们都会用到字符串拼接，比如进行日志输出等等。在Kotlin中，字符串模板是支持的，我们可以很轻松的完成一个字符串数组的组成

```
val book = Book("Thinking In Java", 59.0f, "Unknown")
val extraValue = "extra"
Log.d("MainActivity", "book.name = ${book.name}; book.price=${bo
ok.price};extraValue=$extraValue")
```

注意：关于字符串拼接可以参考这篇文章[Java细节：字符串的拼接](#)

与Java交互性好

Kotlin和Java都属于基于JVM的编程语言。Kotlin和Java的交互性很好，可以说是无缝连接。这表现在

- Kotlin可以自由的引用Java的代码，反之亦然。
- Kotlin可以现有的全部的Java框架和库
- Java文件可以很轻松的借助IntelliJ的插件转成kotlin

Kotlin应用广泛

Kotlin对Android应用开发支持广泛，诸多工具，比如kotterknife(ButterKnife Kotlin版)，RxKotlin,Anko等等，当然还有已经存在的很多Java的库都是可以使用的。

除此之外，Kotlin也可以编译成Javascript。最近使用Kotlin写了一段抓取proxy的代码，实现起来非常快捷。甚至比纯JavaScript实现起来要快很多。

```
fun handle(): Unit {
    window.onload = {
        document.getElementsByClassName("table table-sm").as
List().first()
            .children.asList().filter { "TBODY".equals(i
t.tagName.toUpperCase()) }
            .flatMap { it.children.asList() }.forEach {
                var proxyItem = ProxyItem()
                it.children.asList().forEachIndexed { index, ele
ment ->
                    when (index) {
                        0 -> {
                            proxyItem.host = element.trimmedTextC
ontent()?.split(":")?.get(0) ?:""
                            proxyItem.port = element.trimmedTextC
ontent()?.split(":")?.get(1)?.trim()?.toInt() ?: -1
                        }
                        1 -> proxyItem.protocol = element.trimed
TextContent() ?:""
                        5 -> proxyItem.country = element.trimedT
extContent() ?:""
                    }
                }.run {
                    console.info("proxyItem $proxyItem")
                }
            }
        }
    }
}
```

关于性能

Kotlin的执行效率和Java代码的执行效率理论上一致的。有时候Kotlin可能会显得高一些，比如Kotlin提供了方法的inline设置，可以设置某些高频方法进行inline操作，减少了运行时的进栈出栈和保存状态的开销。

读到这里，是不是想要尝试一下Kotlin呢，它简洁的语法，汇集诸多特性，高效率实现等等，已经在国外风生水起，国外的Pinterest, Square, Flipboard等公司已经开始应用到生产中。

关于转向Kotlin

其实，我在做决定之前（当时Kotlin还没有被钦定）也曾有过考虑，是不是选择了Kotlin就意味着放弃Java呢，冷静下来想一想，其实并不是那么回事，因为Kotlin与Java语法太相近，以及在Kotlin中无时无刻不在和Java相关的东西打交道，所以这点顾虑不是问题的。

对于个人的项目来转向Kotlin，通常不是很难的选择，毕竟Kotlin是那么优秀的语言，相信很多人还是愿意尝试并使用这个事半功倍的语言的。

而比较难抉择的情况是如果如何让团队转用Kotlin，个人认为团队难以转用的原因有很多，比如学习成本，历史包袱等等。但其实根本原因还是思维方式的问题，歪果仁喜欢用工具来提升开发效率，因为人力成本很高。而国内团队提高效率的办法通常是增加成员。好在Flipboard 美国团队自2015年（可能更早）就引入了Kotlin，因此中国团队这边选用Kotlin也更加顺水推舟。当然更主要的是目前团队规模不大，成员一致认可Kotlin的优点。

关于团队转用Kotlin的方法，一般比较行得通的办法是自上而下的推行。这就意味着要么直接的技术负责人比较开明要么就是需要有人来不断推介来影响团队。

做个比较现实的比拟，Java就像是一趟从我的家乡保定开往北京西的耗时将近2个小时甚至更长的普通列车，而Kotlin则是那趟仅需40分钟就能到达的高铁。通常的人会选择高铁，因为它节省了时间和提高了体验。这个时间和体验对应编程中的，我想应该是高效率和高可读性，可维护性的代码。

现在好了，有了Google的支持，Kotlin转Android相信在不久的将来就会全面展开。篡改Python的一句名言“人生苦短，我用Kotlin”，这样一个高效实用的语言应该会被越来越多的团队所接受，并应用到开发生产中。当然也希望在国内环境下大放异彩。

如何看待 Kotlin 成为 Android 官方支持开发语言？

原文链接：<http://blog.csdn.net/androidyue/article/details/72614805>

Google IO 2017宣布了 Kotlin 会成为 Android 官方开发语言。一时间朋友圈和 Android圈被各种刷屏。当然我也顺势而为发布了一篇文章《[为什么我要改用 Kotlin](#)》，着实狠狠地蹭了一波热度（尽管这样会被鄙视）。眼下Android圈已经躁动了，甚至严重到如果对Kotlin视而不见就显得自己不像一个合格的Android程序员。

本文尝试从一个客观全面一点儿的角度来看待这件事情，尽力为大家提供一个比较理性的观点供参考。

为什么会选用 Kotlin

关于 Google 为什么会选择 Kotlin，我认为有两方面的原因。

1. 为了逐渐摆脱专利流氓Oracle。从去年的转向OpenJDK，到现在的支持Kotlin作为官方语言，某种意义是为了摆脱藉由9行代码敲诈获取天价赔偿的Oracle。
2. 选用Kotlin，实至名归，这个荣誉它值得拥有。Kotlin确实以其实用，高效赢得了海外很多公司和开发者的认可，比如Square的Jake大神一直在推Kotlin。Kotlin在国外至少有将近2年的应用生产环境的实践(非JetBrains内部实践应用)。在移动开发中，相比iOS程序员，Android程序员总是很幸运，因为我们有很多优秀好用的工具（Android Studio等），选用Kotlin，则是Google 为开发者提供高效的开发工具的一贯作风。

成为 Android官方开发语言意味着什么

- 官方：工具支持（Android Studio 3.0附带Kotlin),官方的宣传（教学视频，主题演讲等）
- 对于社区来说，Kotlin版本的库和框架如雨后春笋般涌现
- 对于Java，曾经借助Android这场春风，着实让迟暮的它再度辉煌，现在和将来在Android领域可谓是棋逢敌手，Java的在Android开发语言市场份额会降。但是这也并不一定是坏事，有竞争才能更好进步。

- 对于Android 开发者，我们多了一种开发Android的语言选择，那些对于之前由于前景不明朗却对Kotlin跃跃欲试的人可以放心使用了。有了Kotlin意味着开发效率应该会有所提升。
- 对于团队，这往往带来了一个选择的问题，Use Kotlin or not, That's a question. 团队中总有人想要尝试Kotlin，而另一些人则兴致不那么高。由于历史包袱，团队成员兴趣，对于已有项目采用Kotlin和Java长期并存是实际可行的方案。而新项目则应该鼓励使用Kotlin，但具体还需要结合团队的能力和其他因素。

Kotlin的魅力究竟在哪里

Kotlin的有很多特点，比如简洁，安全实用，开发效率高和提升可读性，更好的函数式编程支持。

1.简洁，Kotlin的代码确实比Java更加简洁，比如类型推断，省去结尾的分号等等，然而这远不能成为我们改用Kotlin的原因。

2.安全，这是Kotlin的一个很重要的特性。Kotlin是空指针安全的，JetBrains做了一件很聪明的事情，它们将运行时才能空指针的检测提前到了编译时，主要方式是增加了Any?这种可为空的类型，使用Kotlin之后，我们程序的空指针会得到明显的改善。

3.实用，高效率。Kotlin的实用具体表现在

- 引入Object，便于我们更好的应用单例模式
- 引入data class，避免了我们手写getter/setter/toString等方法
- 引入参数默认值和具名参数，避免了不必要的方法重载
- 支持扩展方法，让我们可以省去好多必须要的代码

4.Kotlin引入了Lambda，Streams API 和函数式编程支持。

- Lambda表达式可以省去了我们创建很多匿名内部类的代码（注由于目前Kotlin 基于JVM6，Lambda表达式在字节码阶段依然会翻译成内部类形式）
- Streams API 结合Lambda表达式和方法引用，让我们的代码处理一件事情以描述的形式，而不是命令实现的方式。
- Kotlin支持OOP(面向对象编程)和FP(函数式编程)，语言本身并没有限制，给了我们选择的自由，Kotlin对FP的友好支持，便于我们写出更加稳定，易于测试，无副作用的方法和代码

5. 可读性 从客观上，Kotlin语法和特性上让代码更加具有描述性而已。但是不得不指出代码可读性主要依赖编写者的编码素质和能力。

对我个人而言，高阶函数和方法扩展这两个特点着实真心受用。方法扩展会让我有一种创造感，这是Java种的Util方法所无法比拟的。

Kotlin是否会取代Java

这个很难说，因为这个世界上并不是一件事物好，就会必然得到广泛应用的。一件事物的推广开来靠的是一群人，但阻力也往往也来自一群人，只不过和前者不是相同人群。

从个人主观来看这个问题，我更加愿意看到这种现象发生。原因并不是因为我更喜欢Kotlin，而是在于我更愿意看到事物在进步，在变得优秀，所以即便某一天Kotlin被更加优秀的语言取代，我也是很欢迎的。

哪些人适合率先应用 Kotlin

Kotlin适用于多个平台，并没有对学习者做限制。任何有学习意愿的人都可以习得这门语言。

但是考虑到国内 Kotlin 资源不够丰富，网络不够畅通等问题，所以导致了很多人变成了吃瓜群众进行观望。

然而，对于一个项目和团队来说，总需要有第一个人先来推进。而且这个推进过程并非顺利，这其中包括

- 首先你需要足够了解Kotlin
- 你需要说服团队，这期间你会接收到很多challenges,有时候你会很沮丧和生气
- 你需要提供一系列的资料或分享，比如如何配置，sample code, troubleshooting 等等
- 有时候甚至你经常兼职做mentor指导工作，这也就意味着你的手头上的其他工作需要被打断

上述推进 Kotlin 观点部分参考自Life is Great and Everything Will Be Ok, Kotlin is Here (Google I/O '17) 中 Christina Lee(Pinterest Software Engineer,国外 Kotlin 美女布道师之一)的分享内容。

虽然 Kotlin 很优秀，但是推动在项目中推动 Kotlin 应用并非易事，因为这对于新事物来说在正常不过了，就像明治维新一样看起来很光鲜，成功，但是它的变革进程并非顺利，先是血雨腥风的倒幕运动，再到明治六年爆发的标志武士时代结束的西南战争，经过数十年的努力才算取得成功。

因此关于哪些人适合率先应用 Kotlin，我认为需要具备以下几点

- Java 技术和基础要好，这一点很重要
- 英语要好，因为目前 Kotlin 的资料几乎都是英文的，当然也推荐看英文的
- 愿意承担在项目团队推进工作，有耐心，敢挑战，负责任

关于Kotlin 项目应用中的一些顾虑

目前想到了一些关于 Kotlin 应用在项目中的一些顾虑。这些顾虑目前并非全面，但是提出来，希望大家可以规避和改善。

1.写出来的代码并不是 Kotlin style。解决这个问题，还是需要多学习和思考

2.扩展方法的滥用，Kotlin 的扩展方法很好，我们可以扩展很多方法，弥补 Framework的一些不完善，但是扩展时我们需要谨慎，一定要把合适的方法放到合适的类型上，不可为了简单增加不符合某些类不应该具备的职责。具体需要最好以下两点

- 选择在合理范围内的最抽象类增加方法，比如我们想为 Activity 增加一个 longToast，应该想一想是不是放在更加抽象的 Context 会更好一些
- 同时也不能为了便利，增加和当前类不相关的方法，比如我们想为每个 Context 增加显示一个简单 dialog 的扩展方法，这显然不是很合理，因为对于非 UI 的 Context 这是有问题的。

Android 程序员的核心竞争力在哪里

Kotlin出来之后，听到了两种不同的声音：

- 太好了，终于可以有理由改用 Kotlin 了，写代码更加高效了。可以逐渐开始放弃 Java 了。
- WTF，又要学新的语言，感觉好累，会不会以后面试不会 Kotlin 就被 pass 掉呢。

出现以上两种不同的声音，不得不引起我们对于 Android 程序员的核心竞争力的思考。那么到底什么才是 Android 程序员的核心竞争力呢？

Android程序员和其他程序员甚至其他职业并无二致，我认为这种竞争力表现在解决问题的能力。想要具备这种能力，极其依赖我们对问题和技术的准确认识和扎实的基础。

编程语言本质上还是工具，好的工具能带来更好的效果，但是如何运用好，将效率和质量提升到最高，则还是更主要的依赖于开发者的能力。

选用好的工具，更侧重夯实基础和加强对事物本质认识的能力，我想这样才能让我们的竞争力更强。

总结而言，Kotlin是一个更好的工具，没有它，并不影响我们日常的Android开发工作。但是我还是建议开发者和团队去尝试这种语言，抓住这个近在咫尺的小确幸。

额外的话

事情的发展越来越显得不可控了，推介Kotlin和不看好Kotlin的人逐渐分化出来，更准确的说，甚至这件事已经快要演变成了从对事变成了对人。

Kotlin成为Android官方语言的消息一出来，一下子出来了很多人被当做投机蹭热点的Kotlin推介者，当然还出现了一些看不惯这些做法的人，他们认为前者刻意拔高了Kotlin。因而讨论越来越偏向从事情到人的方面。我想要说的是，就像商人追求利润，资本家攫取剩余价值那样，投机者蹭热点，以及招致他人批评，这都是正常的事情，但是我们不能让讨论脱离问题的本质，我们需要回归。

很多人说**Kotlin**无非就是很多语法糖，没什么突破

没错，**Kotlin**是有很多很多的语法糖。有必要简单普及一下语法糖的概念（如下摘自[维基百科](#)）

In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language “sweeter” for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

由定义可知，语法糖的目的就是让代码更简单，更可读。

决定Kotlin使用这么多语法糖的除了简洁，高效可读之外，还有一个原因，是因为kotlin编译生成的class文件是目标到JVM 6（基于JVM 6是一种权衡后的结果），比如我们在Kotlin中使用了Lambda，它是不可能编译成invokedynamic指令的，因为那

样会导致在 JVM 上根本无法识别，所以经常通过翻译成内部类的形式来实现。

使用语法糖又能怎样，它的目的是好的，毕竟它真真实实介绍了开发人员的代码量。

Kotlin 是一门实用语言，这是它的基因，它不是学术语言，它的目标是减轻开发者的负担。它很适合 Android，因为大多数的 Android 的程序员是做工程。

基于 JVM 没有什么不好

Kotlin，是基于 JVM 的编程语言，但是基于 JVM 的编程语言并没有什么不好。(J)VMM 的出现无非也是一种平衡的结果。在牺牲部分执行效率的前提下，提供了一定的抽象，加快了开发者的效率。这种 tradeoff 其实更加有利于人的一侧，这也是编程语言发展的趋势和目标

Kotlin 在国内推广应用的道路可谓漫漫而修远兮。因此更需要真正实践，去踩坑的人开始 on board，去出产更多的真正能推动 Kotlin 应用的文章，分享等这些有实质性意义的行动。

更多关于 Kotlin 和 Android 内容，请访问我的独立博客 [技术小黑屋](#) 或者扫描下方的二维码关注我的公众号获取最及时的内容推荐。

Kotlin Script 及其运行机制简析

1. 认识 kts

打开你的 IntelliJ，随便找个位置，注意我说的，随便找个位置，创建一个文件，命名为 Hello.kts，然后你就会发现 IntelliJ 能够识别这种类型，文件的 icon 与 kt 后缀的 kotlin 文件没啥区别。

那你知道你创建了一个什么东西吗？它究竟与平时我们写的 Kotlin 代码有啥区别呢？其实，从名字我们就可以了解到，这是一个 Kotlin 的脚本文件，我们可以在其中直接写函数调用，逻辑判断，数值计算，干什么都行。

Hello.kts

```
import java.io.File

println("Hello from kts")

val file = File(".")
file.listFiles().forEach(::println)

println("The End.")
```

这段代码能输出什么呢？

```
Hello from kts
./gradle
./idea
./build
./build.gradle
./gradle
./gradlew
./gradlew.bat
./Hello.kts
./settings.gradle
./src
The End.
```

我恰好把这个脚本文件放到了一个工程的目录下面，于是它输出了这个工程根目录的所有文件。

2. 命令行调用 kts

如果只是在 IntelliJ 当中能够运行脚本，那多没意思。脚本就是要放到命令行跑的，就跟 python 一样，当成 shell 的神助攻来帮我们处理一下任务才好。

IntelliJ 的运行方法当然也是可以的，我们不妨把它的命令复制过来给大家看一下：

```
$JAVA_HOME/java -Dfile.encoding=UTF-8 -classpath "$INTELLIJ_HOME
/Kotlin/kotlinc/lib/kotlin-compiler.jar:$INTELLIJ_HOME/Kotlin/ko
tlinc/lib/kotlin-reflect.jar:$INTELLIJ_HOME/Kotlin/kotlinc/lib/k
otlin-runtime.jar:$INTELLIJ_HOME/Kotlin/kotlinc/lib/kotlin-scrip
t-runtime.jar" org.jetbrains.kotlin.cli.jvm.K2JVMCompiler -scrip
t /Users/benny/workspace/temp/Forty/Hello.kts
```

不知道大家看明白没，Kotlin 的编译器或者说脚本运行时环境都是 jar 包，用 Java 直接调用就 OK 了。不过这么复杂的命令我可不想每次都写。

嗨，这你还犹豫什么，赶紧安装 kotlin 的安装包，里面有 kotlinc 和 kotlin 这样的命令，用法几乎与 javac 和 java 一模一样。安装方法[点这里](#)。

啊，你说安装好了？那么这时候你运行 kotlinc，是不是会出现一个响应式终端呢？跟 Python Scala 之类的一样呢？

```
$ kotlinc
Welcome to Kotlin version 1.0.6-release-127 (JRE 1.8.0_60-b27)
Type :help for help, :quit for quit
>>>
```

你可以在里面随便敲个运算式啥的，从今天开始，kotlin 也可以成为你的御用“计算器”啦！

额，扯远了，现在我们该说说怎么运行刚才那个脚本了：

```
$ kotlinc -script Hello.kts
Hello from kts
./.gradle
./.idea
./build
./build.gradle
./gradle
./gradlew
./gradlew.bat
./Hello.class
./Hello.kts
./settings.gradle
./src
The End.
```

我最初觉得应该是类似 python 那样直接运行，可结果却有点儿让人尴尬。。好吧，随便啦。

```
$ kotlin Hello.kts
error: running Kotlin scripts is not yet supported
```

3. 我的 main 函数去哪儿啦？

我们都知道，Java 虚拟机上面的程序入口是 main 函数，嗯，就连 Android dalvik 那个算不上真正意义上的 Java 虚拟机的虚拟机，入口函数也是 main 呢！可是前面的脚本分明就没有 main 函数，还跑得挺欢啊，这简直不能让人相信爱情了（什么？

跟爱情有毛关系？！）。。

好吧，这个事儿我们还是要仔细探查一下，不然毁了三观可不好。Java 系的孩子，还是要有点儿信仰的，嗯，信仰 main 的存在~

且说，我们运行 kotlinc 这个程序，你知道它是什么吗？不知道？没关系，找着它，结果发现丫其实就一 shell 脚本。。我去，搞得这么恐怖，原来就一 shell。。

Mac 版 kotlinc 部分

```
...
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.Main")
else
    [ -n "$KOTLIN_COMPILER" ] || KOTLIN_COMPILER=org.jetbrains.kotlin.cli.jvm.K2JVMCompiler
    java_args=("${java_args[@]}" "-noverify")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-preloader.jar" "org.jetbrains.kotlin.preloading.Preloader" "-cp" "${KOTLIN_HOME}/lib/kotlin-compiler.jar" $KOTLIN_COMPILER)
fi
```

我们看到了什么？org.jetbrains.kotlin.cli.jvm.K2JVMCompiler？小哥，你看起来好生面熟啊，哪儿见过呢？

```
$JAVA_HOME/java -Dfile.encoding=UTF-8 -classpath "$INTELLIJ_HOME/Kotlin/kotlinc/lib/kotlin-compiler.jar:$INTELLIJ_HOME/Kotlin/kotlinc/lib/kotlin-reflect.jar:$INTELLIJ_HOME/Kotlin/kotlinc/lib/kotlin-runtime.jar:$INTELLIJ_HOME/Kotlin/kotlinc/lib/kotlin-script-runtime.jar" org.jetbrains.kotlin.cli.jvm.K2JVMCompiler -script /Users/benny/workspace/temp/Forty/Hello.kts
```

原来，IntelliJ 运行 kts 用到的命令，在 kotlinc 当中也是一样一样的，嗯哈，这就有意思了，我们运行一段脚本的程序入口原来在 K2JVMCompiler 当中，那我们不妨找着它的源码一看究竟~

K2JVMCompiler 在 Kotlin 源码的 compiler/cli 模块下。

它的入口方法倒也直接了当，

```
@JvmStatic fun main(args: Array<String>) {
    CLICompiler.doMain(K2JVMCompiler(), args)
}
```

原来这只是一个壳而已，我们还是一步步往下追查吧。在追查之前呢，我们需要一点儿想象力，猜测一下 kts 是如何运行的。

首先可以确定的是 kts 并没有 main 函数，所以一种可能是 kotlin 编译器在运行时给它生成一个 main 函数，然后调用它。这里有个问题，如果这个调用时普通 Java 虚拟机程序的那样调用的话，这就意味着 kts 执行的过程会有两个进程存在，一个是我们刚才执行的命令，另一个是以动态生成 main 为入口函数的 kts 文件。

还有一种可能，kts 文件直接编译成一个普通的类，直接在 kotlinc 的运行时中类加载并且运行。

另种方式比较下来，显然第二种最为简单，不过我们在 kts 当中写的代码究竟是作为哪部分代码运行的呢？

“元芳，你怎么看？”

“大人所言极是呀！只是小可有一事不明...”

“你哪儿来那么多事儿...”

好，猜测完毕，开始查案~

刚一开始看了两行就给我逗乐了，这个代码跳来跳去的，最后竟然又回到了 K2JVMCompiler.doExecute 方法，接着又到了 KotlinToJVMBytecodeCompiler.compileAndExecuteScript，这里基本上告诉我们 Kotlinc 会直接编译 kts 并且加载运行它。

```

fun compileAndExecuteScript(
    environment: KotlinCoreEnvironment,
    paths: KotlinPaths,
    scriptArgs: List<String>): ExitCode
{
    ...
    val scriptClass = compileScript(environment, paths)
    tryConstructClassFromStringArgs(scriptClass, scriptArgs)
    ...
}

```

我们看下这个方法的内容，省略掉异常处理的代码之后，第一句是编译这个 kts，得到 scriptClass，这实际上就是一个 Java Class，后面的 tryConstructClassFromStringArgs 则是要实例化这个类，scriptArgs 则是我们在运行这个脚本时传入的其他参数，这里作为脚本生成的类 scriptClass 的构造方法的参数传入。

```

fun tryConstructClassFromStringArgs(clazz: Class<*>, args: List<String>): Any? {
    try {
        return clazz.getConstructor(Array<String>::class.java).newInstance(args.toTypedArray())
    } catch (e: NoSuchMethodException) {
        for (ctor in clazz.kotlin.constructors) {
            val mapping = tryCreateCallableMappingFromStringArgs(ctor, args)
            if (mapping != null) {
                return ctor.callBy(mapping)
            }
        }
    }
    return null
}

```

哦，这样我们就明白了，原来 kts 当中的代码其实是被编译成类的构造方法来运行的。这么说来我们还可以给脚本传入参数，在脚本当中引用命令行传入的参数也不难：

```
println("Hello from kts, args below: ")  
args.forEach(::println)  
println("The End.")
```

运行输出：

```
$ kotlinc -script Hello.kts X-Man Wolfrine  
Hello from kts, args below:  
X-Man  
Wolfrine  
The End.
```

注意，如果你需要单步调试上面的过程，可以直接在 IntelliJ 当中右键运行 org.jetbrains.kotlin.cli.jvm.K2JVMCompiler，参数填入 -script [kts文件的路径]即可。如果遇到下面的错误：

```
Class 'xxx' is compiled by a pre-release version of Kotlin and c  
annot be loaded by this version of the compiler
```

确保你的编译环境和 IntelliJ 插件一致的前提下，加入 -Xskip-metadata-version-check 参数来忽略错误即可。

4. 小结

通过这篇文章我们不仅知道了 Kotlin 可以支持脚本方式运行，还知道了其运行的原理：编译成一个类，脚本代码作为其构造方法运行，命令行参数作为构造方法的参数传入。

其实前面这段分析本身没有什么难度，它最有价值的地方在于它为我们提供了一个方便快捷了解 Kotlin 内部运行机制的入口，哪里不会断哪里，妈妈再也不用担心我的 Kotlin~



长按识别二维码，关注Kotlin

原文链接：<https://academy.realm.io/cn/posts/oredev-jake-wharton-kotlin-advancing-android-dev/>

过去一年，使用 Kotlin 来为安卓开发的人越来越多。即使那些现在还没有使用这个语言的开发者，也会对这个语言的精髓产生共鸣，它给现在 Java 开发增加了简单并且强大的范式。Jake Wharton 在他的 Øredev 的讨论中，提到了 Kotlin 是如何通过提升安卓开发的语言特性和设计模式来解决这些严重的问题，通过这些方法你可以清除那些无用的 API 还有无效代码。充分利用扩展特性来解决你的开发中的模板性代码的问题！

Kotlin 代码的格式会有些奇怪，我们非常抱歉。我们的高亮系统还不支持这种语言！我们保证 Jake 的代码事实上是漂亮而且实用的。

为什么要推广这个语言？

好吧，大伙。欢迎来到这里。我们今天的主题是使用 Kotlin 语言的高级安卓开发 – 推广我们开发应用的语言。首先呢，我们需要弄清楚我们为什么要在安卓开发中更进一步？为什么我们需要一些新的东西来推进安卓开发？

许多人都会提到的一个经典的原因就是我们现在被 Java 6-ish 困住了，Java 7 也差不多是同样的状况。这被称作是一片荒地。

javax.time 是 Java 8 才引进的新库，JSR310。这个库我们在安卓上没有，所以我们被老的日期和日历的 API 困住了，它们都非常容易出错。

Java 8 的另外一个功能 streams，安卓也没有。我们还缺少其他的语言特性，比如 lambdas，method references 和 不引用 outer class 的 anonymous classes。最后，我们没有 try-with-resources。Try-with-resources 不在 Java 7 里面，而安卓在 API 19 中大部分都是 Java 7，在 API 20 中又增加了点，但不是全部。所以只有当你的定义最小的 SDK 的时候，你才能用这些相关的功能。

Java 6 不是个问题

尽管上面4个问题都看起来十分令人头疼，但是事实上它们不是。关于 javax.time，为 Java 8 SDK 编写的大部分代码大部分都能够以 ThreeTenBP 的形式向后兼容 Java 6，所以你可以这样使用。

关于 streams，Java 6 上也有一个向后兼容的移植或者一个常用的 Java 库，RxJava 实现了同样的概念，只不过用了一种略微不同的有争议性的更强大的方式。

也有一个工具叫做 [Retrolambda](#)，它事实上也可以运行 Java 8 bytecode 并且向后兼容了 lambdas 和 method references 到 Java 6。

最后，如我所说，如果你的最小的 SDK 是 19 的话，你可以使用 try-with-resources，而且 Retrolambda 也会允许你这样做。

这些关于 Java 6 是个问题的争论都有工具去解决它们。当然，如果我们能使用 Java 8 是最好的，但是这些替代方案都是经过实战测试而且能够很好地工作的方案。所以上面的理由就不太算数了。无论如何，有两个方面我想重点讨论一下，因为这两个方面没有类似的解决方案。

Java 语言的限制和问题

Java 语言的一些限制和所带来的一些问题，这些问题 Java 语言本身是不可避免的。例如，我们不能给不是我们自己写的 types、classes 或者 interfaces 增加新的方法。长时间以来，我们都会采用 util 类，杂乱无章地堆砌着我们代码或者或者揉在同一个 util package 里面。如果这是解决方案的话，它肯定不理想。

Java 语言的类型系统都有 null 的问题，在 android 上更为明显。因为它没有对可能是或不是 null 的类型完成 first-class representation。所以，被称作 “billion dollar mistake” 的 null 指针异常最后会毁了你的应用。

接下来，Java 肯定不是最简洁的语言。这件事本身不是件坏事，但是事实上存在太多的常见的冗余。这会带来潜在的错误和缺陷。在这之前，我们还要处理安卓 API 带来的问题。

Android API 的设计问题

安卓是一个庞大的继承系统；他们为他们的继承系统而感到自豪，对他们来说这个系统工作的也非常的正常，但是这个系统倾向于把问题推向应用开发者。而且，空引用的问题也回来了，这个问题在安卓系统中十分明显，因为他们想让系统更加有效些。null 被用在许多地方来代表值的缺失，而不是封装成更高级的类型，比如一个类型或者可选项。

Receive news and updates from Realm straight to your inbox

订阅 Comments

同时，回到语言的冗余，安卓的 API 有着自己许多的范式。这也许是因为性能的原因。设计者们最后写出的这些 APIs 需要你，开发者，来做许多的事情来提高效率而不是用其他的方法抽象它们。

所以说 Java 语言和安卓 API 是驱动我们转向类似 Kotlin 语言的两个重要的因素。这不是说 Java 6 没有问题，因为 Kotlin 解决了 Java 6 的许多问题，但是我不认为那是 Kotlin 作为一个替代方案的充分原因，因为其他的方案也可以带来同样的好处。

Kotlin 入门

Kotlin 是公司 [JetBrains](#) 研发的语言。市面上为各种语言开发的 IDE 很多，但是 IntelliJ 平台是 Android Studio 的基础。在他们的网站上，他们这样描述 Kotlin：

为 JVM、Android 和浏览器而生的静态编程语言。

它的目标是 JVM、安卓和 Java 6 字节流。他们想在他们的语言里增加这些特性，而且持续支持 Java 6、JVM 和安卓系统市场。他们特别关注和 Java 的相互调用，这点接下来会讨论。

Kotlin 语法速成

网站上有许多的很棒的指导来学习语法，但是我还是会很快地介绍一下，然后再解释为什么这些语法有利于安卓开发。我们以一个这样的方法定义的语法开始。

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

我们有一个“fun”的定义，这代表着函数。函数名和第一个要注意的事情是这和 Java 有明显的不同。参数名的顺序和参数的类型保留了下来 - 名字后面跟着类型。返回值类型在函数末尾声明。没有分号。

另外一个有意思的事情是这个函数还可以有单行描述，我们事实上可以不用大括号和 return，定义函数和表达式一样。

```
fun sum(a: Int, b: Int) = a + b
```

我们接下来还会看到更简洁的语法。这里有另外一个例子，这个看起来像一个 main 函数，如果你写一个普通的 Java 应用的话：

```
fun main(args: Array<String>) {
    println("Args: $args")
}
```

数组的语法不太一样。但是处理得十分自然。编译后的字节码会使用一个字符串的数组，但是在你的代码里却把它处理地像一个普通的数组。它也支持字符串的插入；我们可以写一个串，然后引用它其中的变量，并且可以自动的替换其中的变量。

最后，看看变量：

```
val name = "Jake"
val people: List<String> = ArrayList()
```

这里我用一个叫做“name”的变量给一个字符串命名，而且这里没有类型定义。语言会自动解释类型，因为它只可能是串。它有“val”的前缀而“Val”是它的值，并且是个不可以修改的值。如果我们想修改它，我们就需要用“var”作为前缀定义变量。

这个 `: List<String>` 是一个看起来像在 field 上的类型，它接在名字后面，像一个方法。最后，当我们调用构造函数的时候，我们不需要使用“new”关键字。其他的语法都是一样的，就是不要“new”。

Kotlin 语言特性

让我们看看那些语言本身的特性，看看他们是如何帮助我们构建安卓应用的。我指出过这些 util 类都是反设计模式的，而且它们会在你的应用里越来越不受控制……

函数扩展

Kotlin 有扩展函数的概念。这不是 Kotlin 语言独有的，但是和其他语言里面我们看到的扩展又不太一样。如果我们在纯 Java 语言的环境下添加一个 date 的方法，我们需要写一个 utils 类或者 dates 类，然后增加一个静态方法。它接收一个实例，然后做些事情，可能会返回一个值。

```

static boolean isTuesday(Date date) {
    return date.getDay() == 2;
}

boolean tuesday = DateUtils.isTuesday(date);

```

这里我增加一个十分有用的“isTuesday”函数给我们的 date utils，然后我们用传统的静态方法调用它。在我展现 Kotlin 语法前，我想和 C# 做个比较。这是在 C# 中我们需要在 date 类中添加一个函数的实现， DateTime：

```

static boolean IsTuesday(this DateTime date)
{
    return date.DayOfWeek == DayOfWeek.Tuesday;
}

```

这里我们得到 date 的实例，然后调用这个方法，在任何 .NET 环境下这都能行得通，只要你在某处定义了该扩展方法，你能在你的整个项目中都能引用到 DateTime。我接下来会解释一个有意思的语言特征。这是 Kotlin 定义的方法：

```

fun Date.isTuesday(): Boolean {
    return getDay() == 2
}

val tuesday = date.isTuesday();

```

在 Kotlin 中，我们用我们想增加的函数的方法的类型来给原有的函数名增加了一个前缀。我们现在调用 Date.IsTuesday 而不是 isTuesday。然后你能在最后得到返回值。我们最后能调用“getDay”并且这个扩展的方法能够被调到，尽管我们没有使用实例来调用它。我们的调用方式和该类原来就有有这个方法时调用的方式一样。我们也能够在 Date 上调用其他的方法。

Kotlin 的一个非常好的功能是，它会自动地转换有 getters 和 setters 综合属性的类型。所以我能够替换 getDay() 为 day，因为这个 day 的属性是存在的。它看起来像一个 field，但是实际上是个 property – getter 和 setter 的概念融合在了一起。

前面我指出的单行函数表达式会使得语法变得更简洁，所以我们可以把上面的代码修改成下面的样子，并且隐藏返回值：

```
fun Date.isTuesday() = day == 2
```

现在我们有一个非常漂亮的单行实现的并且在 date 上使用的扩展方法了。

不像 C#，如果我们不在同一个 package 里面的话，扩展函数需要显示引用。如果不是同一个文件里，我们需要非常清楚地描述这个函数从何而来：`import com.example.utils.isTuesday`

这和 Java 的静态方法 import 非常类似。我们需要显示声明我们调用的函数，这样函数的来源就不会模糊。因为当我看到这段代码时，作为一个 Java 开发者，我知道 date 没有“isTuesday”的函数，但是显式的 import 告诉我它来自于公共的某个 util package。而在 C# 中，我们不知道这个扩展函数从何而来。它可能来自一个库，你的源代码，或者其他的地方，你无法静态地知道，除非你到 IDE 里面找到它的定义。

当然，你可以 command B 然后进入这个函数，这看起来像是 date 类型的一个方法。这和我们在 Java 里面编写它，然后产生的二进制代码一模一样。而且，因为它关注 inter-op，你可以从 Java 侧使用一个自动生成类来调用它。

类型系统中的 Null

我提过 null 会是个问题。Kotlin 事实上在它的类型系统里重新表述了 null。对于 get string 可能返回 null 的函数来说，我会返回 `String?` 来表述这可能是个 null 值。然后在 get string 函数中，我使用 double exclamation mark 语法来直接调用这个 null。这基本上是说，“我知道这可能是个 null，所以把它变成一个普通字符串。”当它真是 null 的时候，它会发出一个检查的信号，然后抛出异常。

但是在消费者的代码里面，代码往往是直接调用该函数，类型系统会生成一个带有问号的字符串，并且传递到调用者的代码处。这意味着如果你不首先做 null 检查或者提前的默认处理机制，你就永远不能解引用。这样，它会最终解决消费者代码中导致 null 指针异常的问题。

函数表达式入门

另外，函数表达式也被称作 lambdas 或者 closures。这里有一个最简单的函数表达式：`{ it.toString() }`。它是一段代码在“it”变量上调用了 two-string 函数。“it”是个 built-in 的名字。当你在写这些函数表达式的时候，如果你只有一个参数传入这段代码，你可以用“it”引用，这只是一个你不需要定义参数的方法。

但是当你需要定义参数的时候，或者不止一个参数要定义的时候，语法就是这样 的：`{ x, y -> x + y }`。我们可以创建一段代码，一个函数表达式，输入两个参数，然后把它们相加。如果我们愿意，我们可以显示定义类型。

```
{ x, y -> x + y }

val sum: (Int, Int) -> Int = { x, y -> x + y }

val sum = { x: Int, y: Int -> x + y }
```

然后，在`fields`上存储它们。现在你可以看到在前二个例子中，类型是在`field`本上上定义的。这意味着函数表达式不需要任何类型信息。然后往下一个，类型信息在函数表达式里面自己包含了，所以我们不需要在变量定义时加入类型信息。

在最后一个例子中，返回值是推断出来的。两个整型相加，输出只能是整型。所以你不需要显示定义它们。

Higher-order 函数

这是个新奇的？？？语，它指的是函数可以接收函数，或者函数可以返回函数。这里是个例子：

```
func apply(one: Int, two: Int, func: (Int, Int) -> Int): Int {
    return func(one, two)
}

val sum = apply(1, 2, { x, y -> x + y })
val difference = apply(1, 2, { x, y -> x - y })
```

这里我们定义一个函数，接收两个整型作为参数。然后第三个参数是一个函数。这和我们之前看到的语法一样，那里我们定义了一个函数接收两个整型并返回一个整型。

然后，在函数体内部，我们调用了该函数，并且传入了两个参数。这就是我们如何使用它来计算和或者其他的东西的方法。我们把这段代码应用到了这两个数字上。回到之前所说的地方，我们说这段代码知道如何相加或者相减，而且我们把它运用到我们传入这个方法的数据上。然后这段代码在自己合适的上下文环境中自动运行。

Kotlin 给你提供了一个手动的方式来调整这段代码成为更好的语法表达。如果函数的最后一个参数是个表达式，你一直都不需要使用括号，你可以只用在所有参数初始化后添加一个。

```
val sum = apply(1, 2) { x, y -> x + y }
val difference = apply(1, 2) { x, y -> x - y }
```

这项技术允许你创建非常漂亮的 DSL 和 API。你可以像 Int 的扩展一样来编写它们。

包含 higher-order 函数的应用

```
fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T> {
    val newList = ArrayList<T>()
    for (item in this) {
        if (predicate(item)) {
            newList.add(item)
        }
    }
    return newList
}

val names = listOf("Jake", "Jesse", "Matt", "Alec")
val jakes = names.filter { it == "Jake" }
```

一个常见的操作是你有一个事情的列表，然后你需要基于一些条件来过滤它们。因为我们有 extension methods 和 higher-order functions，我们可以在列表上写一个这样的函数。这和你期望的实现一样，这个函数比较每个列表中的元素然后返回正确的结果。

现在如果你有一个包含数据的列表，只需用一行代码，如此简洁的函数就能过滤出我们想要的数据。谢天谢地，这个功能实际上已经在 Kotlin 标准库里面包含了，这样我们就不用扩展列表类了。

大部分 Kotlin 的库都已经实现了现有的 Java 类型中类似的高级函数来，这样可以统一在这些类型上的操作。

看看这个在 Kotlin 标准库里面不存在的应用：

```
fun Any.lock(func: () -> Unit) {
    synchronized (this) {
        func()
    }
}
```

除了同步阻塞代码块，我们可以用一个表达式来实现同步阻塞。“Any”是 Kotlin 的对象版本。所有的类型都是“Any”的子类型。我们可以对它增加一个方法，该方法输入是一个函数，然后执行这个函数的代码，这段代码会同步该实例。如果我们要同样的锁，或者一些对象需要被锁住，我们可以把这段代码放在函数内部的锁起来。然后我们把它传给每个实例的这个方法中。这个方法简化了调用者的代码，而且清晰很多。

简单清晰的锁

另一个十分酷的应用存在于我们常用的锁的情景中，我们常常忘记在一些操作之前完成锁的操作。我们可以写这样的类，这个类只允许你访问你需要用锁才能操作的资源。

```
data class Lock<T>(private val obj: T) {
    public fun acquire(func: (T) -> Unit) {
        synchronized (obj) {
            func(obj)
        }
    }
}

val readerLock = Lock(JsonReader(stream))

// Later
readerLock.acquire {
    println(it.readString())
}
```

在这个例子里面，我们创建了依赖流的 JsonReader。设想不论什么原因我们需要多线程同时访问它，这样我们就必须使用锁，锁会帮助我们管理同步的问题。

然后在后面的代码中，我们调用了这个 acquire 的方法，它会在这个 JsonReader 的实例上实现同步，然后把它传给我们提供的函数。所以在这种情况下，我们会在这个 JsonReader 里面再次析构，于是我们需要使用锁。但现在我们的代码根本没有处理锁的问题。我们也没有显式的同步，也没有为 JsonReader 创建锁。可是访问 JsonReader 不使用锁是不可能的。

避免 Kotlin 带来的泄露

前面，我提到过关于 data 的代码，定义了一个这样的值：

```
val notEmpty: (String) -> Boolean { !it.isEmpty() }
```

Kotlin 实现函数表达式的方法和在 Java 里面使用类的方法是一样的。好处是 Kotlin 并没有创建对于外部范围的引用，因为没有这样的类。这样避免了可能的上下文泄漏。

我们需要关心的就是传入的数据。它会导致创建一个静态的单例实例而且没有引用。用这些函数表达式根本不可能产生上下文泄漏。但是在最上层，我们两个是一模一样的。

扩展函数表达式的例子

- 扩展函数 – 给一个类型加入函数但是不修改原来的类型。
- 函数表达式 – 未定义的函数体被用作表达式 (i.e., date)
- Higher-Order 函数 – 一个参数是函数或者返回是函数的函数。

扩展函数表达式是上述三个概念的综合体，这是个强大的而且可以创建清晰的 API 的方法。为了说明这点，我将使用一个 databases 的 API 做为例子，把它改造成一个更加整洁的、没有无效代码的 API。

```
db.beginTransaction();
try {
    db.delete("users", "first_name = ?", new String[] { "Jake" });
    db.setTransactionSuccessful();
} finally {
    db.endTransaction();
}
```

如果你想在一个 transaction 里面执行一个 statement 的话，这是你必须写的六行代码。我们开始一个 transaction，我们把它放在“try finally”里面，然后我们标记这个 transaction 为成功。如果它抛出或者不在“finally”里面抛出异常，我们需要结束 transaction.

这是一个容易带来 bug 的代码，容易健忘的代码和应该重构的代码。任何事情都有可能出现错误，在这里你不小心交换了两个事情，然后突然，你就会有一个很难找到原因的 bug。或者它会在运行时 crash。

扩展函数表达式允许我们解决这个问题。我们现在能给 database 自己增加一个方法，这将给现有的代码构建一个防护墙。

```
fun SQLiteDatabase.inTransaction(func: (SQLiteDatabase) -> Unit)
{
    beginTransaction()
    try {
        func(this)
        setTransactionSuccessful()
    } finally {
        endTransaction()
    }
}

db.inTransaction {
    it.db.delete("users", "first_name = ?", arrayOf("Jake"))
}
```

开始一个 transaction，调用“try”代码段，然后设置 transaction 为成功，最后，结束它。我们接收一个函数作为参数，在使用这个小技巧后，我们在“try”代码段里面执行该函数。在我们的消费者代码里，我们会调用这个我们称作 `inTransaction` 的方法，在这个方法里面我们写的代码都会在一个 transaction 里面执行。你没有别的办法来错误地使用它了。

渠道化你的内部 `SQLiteDatabase`

一个有趣的事情是，在这样的实现方法下，你的代码里面还是会需要 database 的引用。这也是可能会出错的地方。如果你有两个数据库，你可能会在 transaction 里面引用了那个错误的数据库。

我们有多种方法来解决这个问题。一个方法就是我们可以让函数的参数包含需要发生 transaction 的数据库的引用。我们传了 this 给函数，然后我们就不需要调用 “db”，而是使用 “it” 了，它是传给表达式的第一参数。

这个方法还是不太好。我们阻止了潜在问题的发生，但是现在我们每次访问 database 的时候都需要调用 “it”。这里有个有意思的事情是输入参数是个函数。我们可以把这个函数改写成 “SQLiteDatabase” 的一个扩展函数。

这将会使你有些迷惑，因为它是个疯狂的强大的概念。通过把 `func(this)` 替换成 `this.func()`，传入的函数参数变成了该对象的扩展函数，而且这个扩展函数已经存在 “SQLiteDatabase” 里了。因为我们并不是真正需要 this，所以我们抛弃了它。

这个函数会像在 `SQLiteDatabase` 里面定义的那样执行。如果我们调用删除方法，我们不需要用任何东西来证明自己有资格调用，因为我们就像是个 `SQLiteDatabase`。现在我们可以不需要 it 了，所以每一个你放在这个代码段里面的表达式都会看起来像 `SQLiteDatabase` 内部的一个私有方法一样。你不需要证明自己是正确的，因为它总是能在正确的对象上执行自己。

这太棒了，它会把你的代码改编成：

```
db.inTransaction {
    it.db.delete("users", "first_name = ?", arrayOf("Jake"))
}
```

避免额外的垃圾回收

这仍然是安卓的一个大问题。等效的 Java 代码中，我们创建了一个函数的新实例，然后把它传给了我们产生的静态方法。这非常不幸，因为在我们顺序执行代码之前，我们并没有真正地分配内存。

如果我们用 Java 来实现这个，我们需要在每次使用这些函数表达式的时候都分配一些极小的函数对象。这不好，因为这会触发许多的垃圾回收。

谢天谢地，我们有办法在 Kotlin 里面来避免这个问题。我们这里的函数仅仅是个定义好的函数表达式。它使用函数表达式作为入参，这会是个问题。那个函数表达式需要转成一个匿名类。

```
inline fun SQLiteDatabase.inTransaction(func: SQLiteDatabase.()<br/>-> Unit) { ... }
```

我们可以把它变成一个 in-line 的函数，这样我们就告诉了 Kotlin 编译器不要把它作为一个静态函数调用，我需要编译器仅仅把我的函数代码替换到需要调用的地方。虽然这样会产生很多的 bytecode，但是这样产生的 Java 类文件会和容易出错的 Java 代码编译出来的类文件一样的。

经验法则：函数表达式加上 in-line 函数和同样实现的 Java 代码一模一样。现在我们就能清理我们想清理的任何 API 了，找到你安卓里面最差的 API，这样的 API 肯定到处都是。这其中有很多是基于 transaction 的，所以我们可以为 fragments 或者为 shared preferences 使用同样的模式。这是一个增加功能的好方法，而且不会对自动产生的代码带来额外开销。

JetBrains 提供的 Anko

JetBrains 的同事把这个思路用到了极致，并且创建了一个库叫做 [Anko](#)。基本想法是 XML 描述性非常好，而且很适合定义 UI，因为它是分级的。当你创建 UI 的时候，它们也是具备分层特性的。

在这些函数表达式的帮助下，我们可以用同样分层式方式来编写代码。当然我们还能够引用其他所有的方法，其他的重构工具，和其他的 Java 代码的静态分析工具来展示 UI。

```
verticalLayout {  
    padding = dip(30)  
    editText {  
        hint = "Name"  
        textSize = 24f  
    }  
    editText {  
        hint = "Password"  
        textSize = 24f  
    }  
    button("Login") {  
        textSize = 26f  
    }  
}
```

这仅仅是扩展函数表达式的一个发展分支。它们会创建这些类的实例，把它们加到它们的父亲那里，设置合适的属性。除了我们在 XML 里面使用的概念以外，比如 layout，你可以调用一个函数来返回这些 layout 和把这些东西组织在一起。

这一定是个有趣的概念。但不一定适合每一个人。它们也有定制的预览插件。使用 XML 的一个好处就是你可以预渲染视图，然后看到它在设备上的样子。他们也写了一个为 Java 代码的工具，这个工具可以解析 Kotlin 代码并且完成渲染。

这也是个 XML 也会令人烦恼的例子。Java 和 XML 这两个分离的系统，会带来一些一般的麻烦。而这个方法会把这两个分离的系统统一到一个 Koltin 的源文件里面。这会导致性能的提升，因为你减少了 XML 解析的开销，也会减少寻找 XML 中定义的类而发生的反射的开销。

所以虽然不是每个人都喜欢这个解决方法，但是这确是个解决他们碰到问题的新的方案。

结论

我想给你们介绍现在解决安卓系统中的问题过程中的最有用的一些概念。Koltin 语言还有许多其他的一般性的改进，但那都是为了 Java 语言的。

这就是今天我想讨论的让你意识到的一些安卓系统开发的问题和可能解决它们的具体途径。[Kotlin 网站](#) 有着非常多的好的资源。那有一个交互性编辑器，使用它你可以在你的浏览器里面创建和运行 Kotlin 代码。

在同样的编辑器里面，它们也有一系列的交互性教程来帮助你一步步学习语法。

语言还处在 1.0 beta 的状态，所以对于那些因为这个原因而持观望态度的人，你们很快就能加入了。我鼓励你们都试试 Kotlin 语言。

原文链接：<https://academy.realm.io/cn/posts/droidcon-michael-pardo-kotlin/>

去年，Java8 发布了，增加了很多新特性和提升，比如lambda，stream。Java 9 的标准也已经在制定了。但是超过半数的 Android 设备仍在运行着 Java 6，我们要怎么才能用上新的现代化语言呢？

在 DroidCon NYC 2015 的这个分享里，Michael Pardo 介绍了 **Kotlin**: 由 JetBrains 开发出的 JVM 静态语言。Kotlin 由很多新的特性，比如 lambdas，类扩展（class extensions），和 null 安全（null-safty）。它简洁明了，同时由很高的互操作性（interoperable）。

Save the date for Droidcon SF in March — a conference with best-in-class presentations from leaders in all parts of the Android ecosystem.

Kotlin

大家好，我是 **Michael Pardo**，今天我要给大家展示一下 **Kotlin** 这门语言，同时看看他如何让你在 Android 开发的时候更开心，更有效率。

Kotlin 是一个基于 JVM 实现的静态语言。Kotlin 是 **JetBrains** 创造并在持续维护这门语言，对，就是那个创造了 Android Studio 和 IntelliJ 的公司。

Kotlin 有几个核心的目标：

1. 简约：帮你减少实现同一个功能的代码量。
2. 易懂：让你的代码更容易阅读，同时易于理解。
3. 安全：移除了你可能会犯错误的功能。
4. 通用：基于 JVM 和 Javascript，你可以在很多地方运行。
5. 互操作性：这就意味着 Kotlin 和 Java 可以相互调用，同时 JetBrains 的目标是让他们 100% 兼容。

为什么不等 Java 8?

Java 和 Android: 一段历史

我们来看看 Java 和 Android 的历史以及他们的关系。在 2006 年，Java 6 发布了。几年之后，Android 1 的 Alpha 版本发布了，四年后，Java 7 发布了。Android 在 2 年后紧随其后的开始支持 Java 7。去年，Java 8 又发布了。

你想想，你什么时候才能用上 Java 8？可能你学的很快，然后就能用上 Java 8。但是 Android 怎么说都得几年后才能开始支持 Java 8，大家适应 Java 8 又需要很长时间。Android 现在的碎片化很严重，Java 7 只支持 API 19 及以上。如果用了 Java 7，那你的 App 用户群一下子就少了一半。即便我们现在有了 Java 8，100% 的覆盖到了所有的用户设备上，但是 Java 本身还是有些问题的。

我们来看看 Java 的一些问题。

Java 有哪些问题？

- 空引用（Null references）：连空引用的发明者都成这是个 billion-dollar 错误（[参见](#)）。不论你费多大的功夫，你都无法避免它。因为 Java 的类型系统就是不安全的。
- 原始类型（Raw types）：我们在开发的时候总是会为了保持兼容性而卡在范型原始类型的问题上，我们都知道要努力避免 raw type 的警告，但是它们毕竟是在语言层面上的存在，这必定会造成误解和不安全因素。
- 协变数组（Covariant arrays）：你可以创建一个 string 类型的数组和一个 object 型的数组，然后把 string 数组分配给 object 数组。这样的代码可以通过编译，但是一旦你尝试在运行时分配一个数给那个数组的时候，他就会在运行时抛出异常。
- Java 8 存在高阶方法（higher-order functions），但是他们是通过 SAM 类型实现的。SAM 是一个单个抽象方法，每个函数类型都需要一个对应的接口。如果你想要创建一个并不存在的 lambda 的时候或者不存在着对应的函数类型的时候，你要自己去创建函数类型作为接口。
- 泛型中的通配符：诡异的泛型总是难以操作，难以阅读，书写，以及理解。对编译器而言，异常检查也变得很困难。

我们来探索下 Kotlin 是如何解决上面的提到的这些问题的。

Kotlin To The Rescue!

刚才我们提到过的这些缺陷，Kotlin 通常直接移除了那些特性。同时它也加了一些新的特性：

- Lambda 表达式
- 数据类 (Data classes)
- 函数字面量和内联函数 (Function literals & inline functions)

- 函数扩展 (Extension functions)
- 空安全 (Null safety)
- 智能转换 (Smart casts)
- 字符串模板 (String templates)
- 主构造函数 (Primary constructors)
- 类委托 (Class delegation)
- 类型推断 (Type inference)
- 单例 (Singletons)
- 声明点变量 (Declaration-site variance)
- 区间表达式 (Range expressions)

我们将在这篇文章里提及以上大多数特性。Kotlin 之所以能跟随者 JVM 的生态系统不断地进步，是因为他没有任何限制。它编译出来的正是 JVM 字节码。在 JVM 看来，它就跟其他语言一样样的。事实上，如果你在 IntelliJ 或者 Android Studio 上用 Kotlin 的插件，它自带里一个字节码查看器，可以显示每个方法生成的 JVM 字节码。

Hello, Kotlin

我们来看看基本语法。下面是一个最简单的，用 Kotlin 书写的 Hello World：

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

只有一个函数和一个 print 语句。不需要包声明和类引用声明。这就是一个 Kotlin Hello World 程序的所有代码。声明函数的关键字是 `fun`，`fun` 后面跟的是函数的名称，然后括号包裹起来的是函数参数，这个跟 Java 类似。

然而，在 Kotlin 里，得把参数名放在前面，参数类型放在后面，用一个冒号隔开。函数的返回类型在最后，这个跟 Java 放在前面形式不太一样。如果一个函数没有返回任何类型，可以返回一个 `Unit` 类型，当然也可以省略。调用 Kotlin 标准库中的函数 `println` 就能打印 Hello World 出来，实际上它最终调用了 Java 的 `System.out.println`。

Get more development news like this

订阅 Comments

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

接下来，我们来从“Hello World”中提取“World”这个词，并把这个词放到一个变量中。`var` 关键字后跟的是变量的名称。Kotlin 支持字符串内插入变量，只用在字符串内用 `$` 符号开头，随后跟上输出变量的变量名即可，就像如下这样：

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

随后，我们来检查我们是否给 `main` 函数传递了参数。先来判断这个字符串数组是不是空，如果不为空，我们把第一个字符串分配给 `name` 变量。Kotlin 里有个 `val` 类型的声明方法，类似 Java 里的 `final`，也就是常量。

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

在我们编译这个程序的时候，我们遇到一个问题：无法重新分配新的值给一个常量。一种解决方法是用内联的 `if-else` 方法。Kotlin 里的多数的代码块都支持返回值。如果语句进入了 `if` 代码块儿，也就是说 `args` 非空，那么就返回 `args[0]`，否则返回“World”。`if-else` 语句结束后，就直接赋值给我们之前声明的 `name` 常量，下面的例子就是条件赋值代码块：

```

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}

```

我们可以把上面的代码用一行来书写，看起来有点像 Java 里的三目运算符。移除掉那些大括号后，看起相当漂亮：

```
val name = if (args.isNotEmpty()) args[0] else "World"
```

类语法

我们来看看类。类的定义要通过 `class` 关键字，跟 Java 里的一样，关键字后是类名。Kotlin 有一个主构造函数，我们可以直接将构造函数参数列表写在类的声明处，还可以直接用 `var` 或者 `val` 关键字将参数声明为成员变量（又称：类属性），如下：

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

继续之前的例子，有了主构造函数以后，我们就不再需要成员变量赋值语句了。在 Kotlin 里创建实例的时候，不必使用 `new` 关键字。你只需要指明创建的类型名就可以创建实例了。

```

class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}

```

很容易发现，字符串插值实际上是错误的，因为 `name` 指向的是一个不存在的变量了。我们可以用刚才提到的 字符串插值表达式 ，即用 `$` 符号和大括号包裹想要插入的变量，来修复这个问题：

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

下面是 `enum` 类。枚举跟 Java 里的枚举很像。定义一个枚举的方法如下：

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}
```

我们来给 `Person` 类增加一个叫 `lang` 的属性，代表一个人的所说的语言。

```
class Person(var name: String, var lang: Language = Language.EN)
```

Kotlin 支持参数默认值，如上：`language` 的默认值就是 `Language.EN`，这样就可以在创建实例的时候忽略这个参数，除非你要改变 `language` 的属性值。我们来把这个例子变得更面向对象一些，给 `person` 增加一个打招呼的方法，简单地输出特定语言打招呼的方法还有人名：

```

enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN)
{
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    person.greet()
}

```

现在在 main 函数里调用 `person.greet()` 方法，看看是不是很酷？！

集合和迭代

```

val people = listOf(
    Person("Michael"),
    Person("Miguel", Language.SP),
    Person("Michelle", Language.FR)
)

```

我们可以用标准库函数 `listOf` 方法创建一个 `person` 列表。遍历这些 `person` 可以用 `for-in` 关键字：

```

for (person in people) {
    person.greet()
}

```

随后，我们可以在每次遍历的时候执行 `person.greet()` 方法，甚至可以更简单，直接调用 `people` 集合的扩展方法 `forEach`，传入一个 `lambda` 表达式，在表达式里用 `it` 代表每次遍历到的 `person` 对象，然后调用它们的 `greet` 方法。

```
people.forEach { it.greet() }
```

我们来创建两个新的类，每个都传入一个默认的语系。我们可以不再像刚才那样重复声明，可以直接用继承的方法来实现。下面是一个扩展版本的 Hello World。展示了很多 Kotlin 的特性：

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Hispanophone("Miguel"),
        Francophone("Michelle")
    ).forEach { it.greet() }
}
```

Kotlin 在 Java 上加了什么特性？

下面，我们来看看 Kotlin 在 Java 之上加了哪些更好用的新特性。

类型推导

你可能在其他语言中看到过类型推导。在 Java 里，我们需要自己声明类型，变量名，以及数值。在 Kotlin 里，顺序有些不一样，你先声明变量名，然后是类型，然后是分配值。很多情况下，你不需要声明类型。一个字符串字面量足以指明这是个字符串类型。字符，整形，长整形，单浮点数，双浮点数，布尔值都是可以无需显性声明类型的。

```

var string: String = ""
var string = ""
var char = ' '

var int = 1
var long = 0L
var float = 0F
var double = 0.0

var boolean = true

var foo = MyFooType()

```

只要 Kotlin 可以推导，这个规则同样适用与其他一些类型。通常，如果是局部变量，当你在声明一个值或者变量的时候你不需要指明类型。在一些无法推导的场景里，你才需要用完整的声明变量语法指明变量类型。

空安全 (null-safety)

Kotlin 一个强大的特性是空安全。我们来看几个例子：

```

String a = null;
System.out.println(a.length());

```

在 Java 里，声明一个 string 类型，赋一个 null 给这个变量。一旦我们要打印这个字符串的时候，会在运行时曝出空指针错误，因为我们在尝试去读一个空值。下面是这个问题的 kotlin 写法，我们定义一个空值，但是在我们尝试操作它之前，Kotlin 的编译器就告诉了我们问题所在：

```

val a:String = null

```

曝出的错误是：我们在尝试着给一个非空类型分配一个 null。在 Kotlin 的类型体系里，有空类型和非空类型。类型系统识别出了 string 是一个非空类型，并且阻止编译器让它以空的状态存在。想要让一个变量为空，我们需要在声明后面加一个 ? 号，同时赋值为 null。

```
val a: String? = null
println(a.length())
```

现在，我们修复了这个问题，继续向下：就像在 Java 里一样，我们尝试打印 stirng 的长度，但是我们遇到了跟 Java 一样的问题，这个字符串有可能为空，不过幸好的是：Kotlin 编译器帮助我们发现了这个问题，而不像 Java 那样，在运行时爆出这个错误。

编译器在长度输出的代码前停止了。想要让编译器编译下去，我们得在调用 length 方法的时候考虑到可能为空的情况，要么赋值给这个 string，要么用一个问号在变量名后，这样，代码执行时在读取变量的时候检查它是否**◆◆◆**空。

```
val a: String? = null
println(a?.length())
```

如果值是空，则会返回空。如果不是空值，就返回真实的值。print 遇到 null 会输出空。

Ternary Null

```
int length = a != null ? a.length() : -1
```

上面的代码你可能在 Java 里见到过。用三目运算符取值，检查是否为空，如果为空则返回真实的长度，否则返回 -1，Kotlin 里又相同的实现：

```
var length = if(a!= null) a.length() else -1
```

如果 `a` 不是 `null`，那么就可以直接读值，否则返回默认值。这里用 `elvis` 操作符实现的简写：

```
var length = a?.length() ?: -1
```

我们用 `?` 号做了一个内联空检查。如果你还记得刚才我说的，如果 `a` 是 `null`，第一个 `?` 表达式就会返回 `null`，如果 `elvis` 操作符 左侧是空，那么他就会返回右侧，否则直接返回左侧的值。

智能转换

Kotlin 支持类型智能转换的特性。如果一个局部对象传入一个类型检查，你可以直接通过这个类型来操作，而不需要再自己做转换，看下面的例子你就明白了：

```
if (x is String) {
    print(x.length())
}
```

我们检查了 `x` 是不是一个字符串，如果是，就打印它的长度。做类型检查，我们需要用到 `is` 关键字，其实跟 Java 里的 `instanceof` 一样。道理很简单，我们既然通过了类型检查，我们就能把它当做这个类型来使用。

```
if (x !is String) {
    return
}

print(x.size())
```

逆操作也没有问题。上面这个例子检查了是否是一个非字符串变量。如果不是，则直接返回。在我们判断了以后就可以认为它是一个字符串了，我们也无需在做显式的类型转换。

```
if (x !is String || x.size() == 0) {
    return
}
```

上面的例子里，我们检查了一个字符串是否是一个非字符串变量，如果左侧的值是 `false`，就会调用右侧的 `or` 判断。`when` 语句也适用上面的规则，`when` 其实是一个增强版的 `switch`。

```
when(x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

我们只要做了类型检查，类型一切都会自动转换。从类型检查到类型自动转换，就是 Kotlin 的智能转换。

字符串模板

很多语言都有字符串模板和字符串插值。下面的例子大概就是你在 Java 里经常用到的：

```
val apples = 4
println("I have " + apples + " apples.")
```

你可以把 `apples` 变量和其他字符串串联起来。用更符合 Kotlin 风格的方式，你可以用插值的方法，在字符串中用 `$` 符号前缀加变量名来代表这个字符串内容。

```
val apples = 4
println("I have $apples apples.")
```

你也可以用下面的表达式：

```
val apples = 4
val bananas = 3

println("I have $apples apples and " + (apples + bananas) + " fruits.") // Java-esque
println("I have $apples apples and ${apples+bananas} fruits.") / Kotlin
```

在 Java 中，可能最常见的方案是在需要显示个数的地方，用加号操作苹果和香蕉的个数，然后将字符串都串起来。但在 Kotlin 中，可以用前缀 `$` 符号加上大括号将操作语句包裹起来代表操作语句的结果。

区间表达式

你可能在其他的语言里见到过这样的表达式。的确，Kotlin 的不少特性是借鉴自其他语言里。下面这个表达式：如果 `i` 大于等于 1，并且小于等于 10，就将其打印出来。我们检测的范围是 1 到 10。

```
if (1 <= i && i <= 10) {
    println(i)
}
```

其实我们可以用 `intRange` 函数来完成这个操作。我们传入 1 和 10，然后调用 `contains` 函数来判断是否在这个范围里。我们打印出 `i` 即可。

```
if (IntRange(1, 10).contains(i)) {
    println(i)
}
```

这个还可以用扩展函数来实现，`1.rangeTo` 创建了一个 1 到 10 的 `intRange`，我们可以用 `contains` 来判断它。

更完美的而简洁的写法，是用下面的操作符：

```
if(i in 1..10) { ... }
```

`..` 是 `rangeTo` 的一个别名，它实际背后工作原理还是 `rangeTo`。

我们还可遍历一个区间，比如：可以用 `step` 关键字来决定每次遍历时候的跳跃幅度：

```
for(i in 1..4 step 2) { ... }
```

也可以逆向迭代，或者逆向遍历并且控制每次的 `step`：

```
for (i in 4 downTo 1 step 2) { ... }
```

在 Kotlin 里，也可以结合不同的函数来实现你想要的区间遍历。可以遍历很多不同的数据类型，比如创建 `strings` 或者你自己的类型。只要符合逻辑就行。

高阶函数

很多语言已经支持了高阶函数，比如 Java 8，但是你并不能用上 Java 8。如果你在用 Java 6 或者 Java 7，下面的例子实现了一个具有过滤功能的函数：

```

public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

我们首先要声明一个函数接口，接受参数类型为 `T`，返回类型为 `R`。我们用接口中的方法遍历操作了目标集合，创建了一个新的列表，把符合条件的过滤了出来。

```

fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T>
{
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}
```

上面的代码是在 Kotlin 下的实现，是不是简单很多？我们调用的时候如下：

```

filter(numbers, { value ->
    value % 2 == 0
})
```

你可能也发现了，我们没有定义任何的函数接口，这是因为在 Kotlin 中，函数也是一种数据类型。看到 `f:(T) -> Boolean` 这个语句了吗？这就是函数类型作为参数的写法，`f` 是函数别名，`T` 是函数接受参数，`Boolean` 是这个函数的返回值。定义完成后，我们随后就能跟调用其他函数一样调用 `f`。调用 `filter` 的时候，我们是用 lambda 表达式来传入过滤函数的，即：`{value -> value % 2 == 0}`。

由于函数类型参数是可以通过函数声明的签名来推导的，所以其实还有下面的一种写法，大括号内就是第二个参数的函数体：

```
filter(numbers) {
    it % 2 == 0
}
```

内联函数

内联函数和高阶函数经常一起见到。在某些场景下，当你用到泛型的时候，你可以给函数加上 `inline` 关键字。在编译时，它会用 lambda 表达式替换掉整个函数，整个函数的代码会成为内联代码。

如果代码是这样的：

```
inline fun <T> filter(items: Collection<T>, f: (T) -> Boolean):
List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers) { it % 2 == 0 }
```

由 `inline` 关键字在编译后会变成如下这样：

```
val filtered = arrayListOf<T>()
for (item in items) if (it % 2 == 0) filtered.add(item)
```

这也意味着我们能实现一些常规函数实现不了的。比如：下面这个函数接受一个 lambda 表达式，但并不能直接返回：

```
fun call(f: () -> Unit) {
    f()
}

call {
    return // Not allowed
}
```

但是如果我们的函数变成内联函数，现在我们就能直接返回了，因为它是内联函数，会自动和其他代码混合在一起：

```
inline fun call(f: () -> Unit) {
    f()
}

call {
    return // Now allowed
}
```

内联函数也允许用 `reified` 类型。下面这个例子就是一个真实场景下的函数，通过一个 View 寻找类型为 `T` 的父元素：

```
inline fun <T : Any> View.findViewParent(): T? {
    var parent = getParent()
    while (parent != null && parent !is T) {
        parent = parent.getParent()
    }
    return parent as T // Cast warning
}
```

这个函数还有些问题。由于泛型类型被擦除了，所以我们无法检测类型，即便我们手工来做检查，依然会出现 warning。

解决方案是：我们给函数参数类型加上 `reified` 关键字。因为函数会被编译成内联代码，所以我们现在就能手工检查类型消除警告了：

```
inline fun <reified T : Any> View.findViewParent(): T? {
    var parent = getParent()
    while (parent != null && parent !is T) {
        parent = parent.getParent()
    }
    return parent as T // Type cast allowed
}
```

函数扩展

函数扩展是 Kotlin 最强大的特性之一。下面是一个工具函数，检测 App 是否运行在 Lollipop 或者更高的 Api 之上，它接受一个整数参数：

```
public fun isLollipopOrGreater(code: Int): Boolean {
    return code >= Build.VERSION_CODES.LOLLIPOP
}
```

通过 `被扩展类型.函数` 的写法，就能将函数变成被扩展类型的一部分，写法如下：

```
public fun Int.isLollipopOrGreater(): Boolean {
    return this >= Build.VERSION_CODES.LOLLIPOP
}
```

我们不在需要参数，想要在函数体内调用整数对象需要用 `this` 关键字。下面就是我们的调用方法，我们可以直接在整数类型上调用这个方法：

`16.isLollipopOrGreater()`

函数扩展可以是任何整形，字面量或者包装类型，也可以在标记为 `final` 的类上做类似操作。因为扩展函数不是真的给类增加代码，任何人都没有办法去修改一个类，它实际上是在创建了一个静态方法，用语法糖来让扩展函数看着像是类自带的方法一样。

Kotlin 在 Java 集合中充分利用了扩展函数，这有一个例子操作集合：

```

final Function<Customer, Order> customerMapper = ...;
final Function<Order, Boolean> orderFilter = ...;
final Function<Order, Float> orderSorter = ...;
final List<Order> vipOrders = sortBy(filter(map(customers,
    customerMapper),
    orderFilter),
    orderSorter);

```

我们对一个 customer 集合，执行了 map，filter，以及 sort 操作。嵌套的写法混乱而且难以阅读。下面是标准库的扩展函数写法，是不是简洁了很多：

```

val vipOrders = customers
    .map { it.lastOrder }
    .filter { it.total >= 500F }
    .sortBy { it.total }

```

属性

Kotlin 把属性也变成了语言特性。

```

class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }

    public void setFirstName(String firstName) { this.firstName =
        firstName }
    public void setLastName(String lastName) { this.lastName = last-
        name }
    public void setEmail(String email) { this.email = email }
}

```

上面是一个典型的 Java bean 类。可看到很多成员变量，和很多 `getter`，`setter` 方法，这可是只有三个属性的时候，就生成了这么多代码。来看看 Kotlin 的写法：

```
class Customer {
    var firstName: String = // ...
    var lastName: String = // ...
    var email: String = // ...
}
```

你只需要将成员变量定义成一个变量即可，默认是 `public` 的。编译器会自动生成 `getter` 和 `setter` 方法。

主构造函数

Kotlin 中，类可以拥有多个构造函数，这一点跟 Java 类似。但你也可以有一个主构造函数。下面的例子是我们从上面的例子里衍生出来的，在函数头里添加了一个主构造函数：

在主构造函数里，可以直接用这些参数变量赋值给类的属性，或者用构造代码块来实现初始化。

```
class Customer(firstName: String, lastName: String, email: String) {
    var firstName: String
    var lastName: String
    var email: String

    init {
        this.firstName = firstName
        this.lastName = lastName
        this.email = email
    }
}
```

当然，更好的方法是：直接在主构造函数里定义这些属性，定义的方法是在参数名前加上 `var` 或者 `val` 关键字，`val` 是代表属性是常量。

```
class Customer(
    var firstName: String,
    var lastName: String,
    var email: String)
```

单例

你可能经常会用到单例设计模式。比如一个 `Logger` 类，在 Java 里，有多种实现单例的写法。

在 Kotlin 里，你只要在 `package` 级别创建一个 `object` 即可！不论你在什么域里，你都可以像单例一样调用这个 `object`。

```
object Singleton
```

比如下面是一个 looger 的写法：

```
object Logger {
    val tag = "TAG"
    fun d(message: String) {
        Log.d(tag, message)
    }
}
```

你可以直接通过 `Logger.D` 的方法来调用 `D` 函数，它在任何地方都是可用的，而且始终只有一个实例。

Companion Objects

Kotlin 移除了 `static` 的概念。通常用 `companion object` 来实现类似功能。你可能时常会看到一个 `Activity` 有一个静态类型的 `String`，名叫 `tag`，和一个启动 `Activity` 的静态方法。Java 中的实现如下：

```

class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();
}

public static void start(Context context) {
    context.startActivity(new Intent(context, LaunchActivity.class));
}
}

```

在 Kotlin 下的实现如下：

```

class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}

Timber.v("Starting activity ${LaunchActivity.TAG}")

LaunchActivity.start(context)

```

有了 `companion object` 后，就跟类多了一个单例的对象和方法一样。

类委托

委托是一个大家都知道的设计模式，Kotlin 把委托视为很重要的语言特性。下面是一个在 Java 中典型的委托写法：

```

public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(List<E> delegate) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}

```

我们有一个自己的 lists 实现，通过构造函数将一个 list 存储起来，存在内部的成员变量里，然后在调用相关方法的时候再委托给这个内部变量。下面是在 Kotlin 里的实现：

```
class MyList<E>(list: List<E>) : List<E> by list
```

用 `by` 关键字，我们实现了一个存储 `E` 类型的 list，在调用 `List` 相关的方法时，会自动委托到 list 上。译者注：参考 [Kotlin 官方文档](#)了解更多。

声明点变型（Declaration-Site Variance）

这个可能是一个比较容易让人迷惑的主题。首先，我们用一个协变数组来开始我们的例子，下面的代码能够很好的编译：

```

String[] strings = { "hello", "world" };
Object[] objects = strings;

```

`string` 数组可以正常的赋值给一个 `object` 数组。但是下面的不行：

```
List<String> strings = Arrays.asList("hello", "world");
List<Object> objects = strings;
```

你不能分配一个 string 类型的 list 给一个 object 类型的 list。因为 list 之间是没有继承关系的。如果你编译这个代码，会得到一个类型不兼容的错误。想要修复这个错误，我们得用到 Java 中的点变型（use-site variance）去声明，所谓的点变型就是在声明 list 可接受类型的时候，用 `extends` 关键字给出参数类型的可接受类型范围，比如类似如下的例子：

译者注：点变型只是一个名字，不要太在意为什么叫这个，简单理解就是类似通配符原理，具体可以查看这个[维基页面#Use-sitevariance_annotations.28Wildcards.29](#)。

```
public interface List<E> extends Collection<E> {
    public boolean addAll(Collection<? extends E> collection);
    public E get(int location);
}
```

`addAll` 方法可以接受一个参数，参数类型为所有继承自 `E` 的类型，这不是一个具体类型，而是一个类型范围。每次调用 `get` 方法时，依然返回类型 `E`。在 Kotlin 中，你可以用 `out` 关键字来实现类似的功能：

```
public interface List<out E> : Collection<E> {
    public fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E>
{
    override fun addAll(c: Collection<E>): Boolean
}
```

上面的一系列被称为声明点变型，即在声明可接受参数的时候，就声明为它是可变的。比如上面例子：我们声明参数是可以允许所有继承自 `E` 类型的，返回类型也为 `E` 的。

现在，我们有了可变和不可变类型的列表。 可变性(variance) 其实很简单，就是取决于我们在声明的时候是动作。

译者注：其实不论声明点变型（Declaration-Site Variance）还是 点变型(Use-site variance) 都是为了实现泛型的类型声明，标注泛型类型可支持的范围，厘清泛型类型上下继承边界。参考 [Generic Types#Generic_types](#)。

操作符重载

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}

var purse = Purse(1.50f)
purse += Coin.QUARTER // 1.75
purse += Coin.DIME // 1.85
purse += Coin.PENNY // 1.86
```

上面的代码中，我们创建了一个硬币枚举，每个硬币枚举都代表一个特定数额的硬币。我们有一个 `Purse`（钱包）类，它拥有一个 `amount` 成员变量，代表钱包里现在有多少钱。我们创建了一个叫做 `plusAssign` 的函数，`plusAssign` 是一个保留关键字。这个函数会重载 `+=` 操作符，也就是说当你在调用 `+=` 符号的时候，就会调用这个函数。

随后，创建一个 `purse` 实例，可以直接用 `+=` 操作来实现给钱包里放钱进去。

让你的项目支持 Kotlin

让你的项目支持 Kotlin 其实非常简单，你需要让你的项目里启用 gradle 插件，Kotlin Android 插件，拷贝代码文件，引入标准库即可。

开始 Kotlin 之路

- [Kotlin 文档](#)
- [快速入门](#)
- [Kotlin 和 Java 的不同](#)

Q&A (43:40);)

Q: Kotlin 有没有什么缺点？

Michael: Kotlin 的少数特性可能有性能问题，比如 annotation 处理速度很慢。除此之外强烈推荐。

Q: 支持 Unit Test 么？

Michael: 支持，我们用 [Robolectric](#) 和 [Mockito](#) 来做测试。

Q: Debug 的时候会很顺利吗？

Michael: Kotlin 生成的只是 JVM 字节码而已，而且 Kotlin 和 IntelliJ 共存的非常好，毕竟都是一家做的。我们之前用的 retrolambda，但是他生成的字节码总是有些小问题。

Q: Kotlin 对 Jack and Jill 的支持如何？

Michael: Jack and Jill 和 Kotlin 不能一起工作。

Q: Kotlin 上用反射怎么样？

Michael: Kotlin 上用反射的一大问题是回引入一个很大的库..... 不过，这些你可以用 Java 来做。

Kotlin语言基础

- package
- 声明变量和值
- 变量类型推断
- 字符串与其模板表达式
- 流程控制语句
- 代码注释、语法与标识符
- 修饰符
- 函数扩展和属性扩展
- 空指针安全
- Kotlin入门和使用

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。

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.jackson` 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.jackson.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
```

```
package com.easy.kotlin

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

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

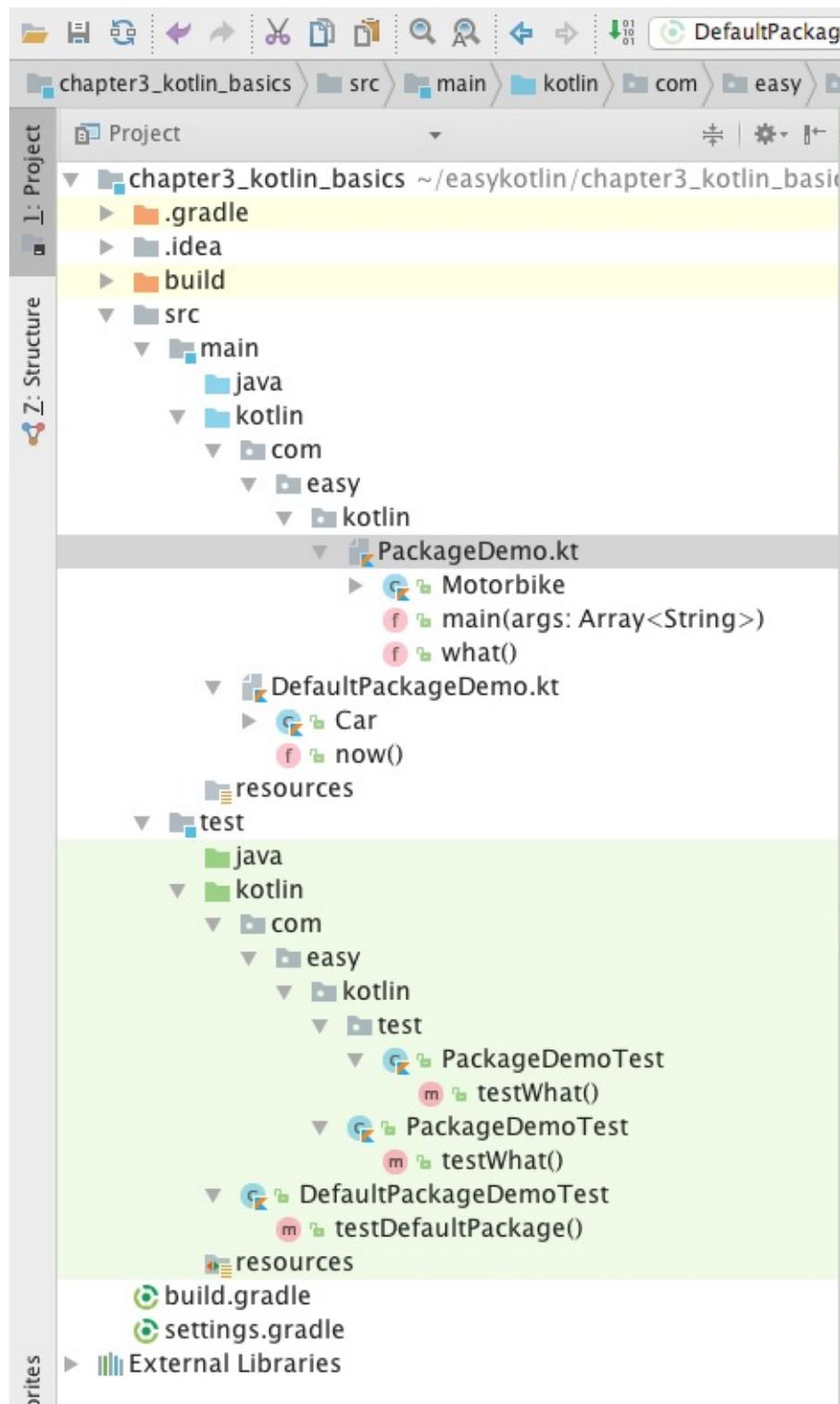
fun main(args:Array<String>){
    println("Hello,World!")
}
```

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

package



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
```

```
package com.easy.kotlin

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()
    }
}
```

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

```
package
```

```
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()
    }
}
```

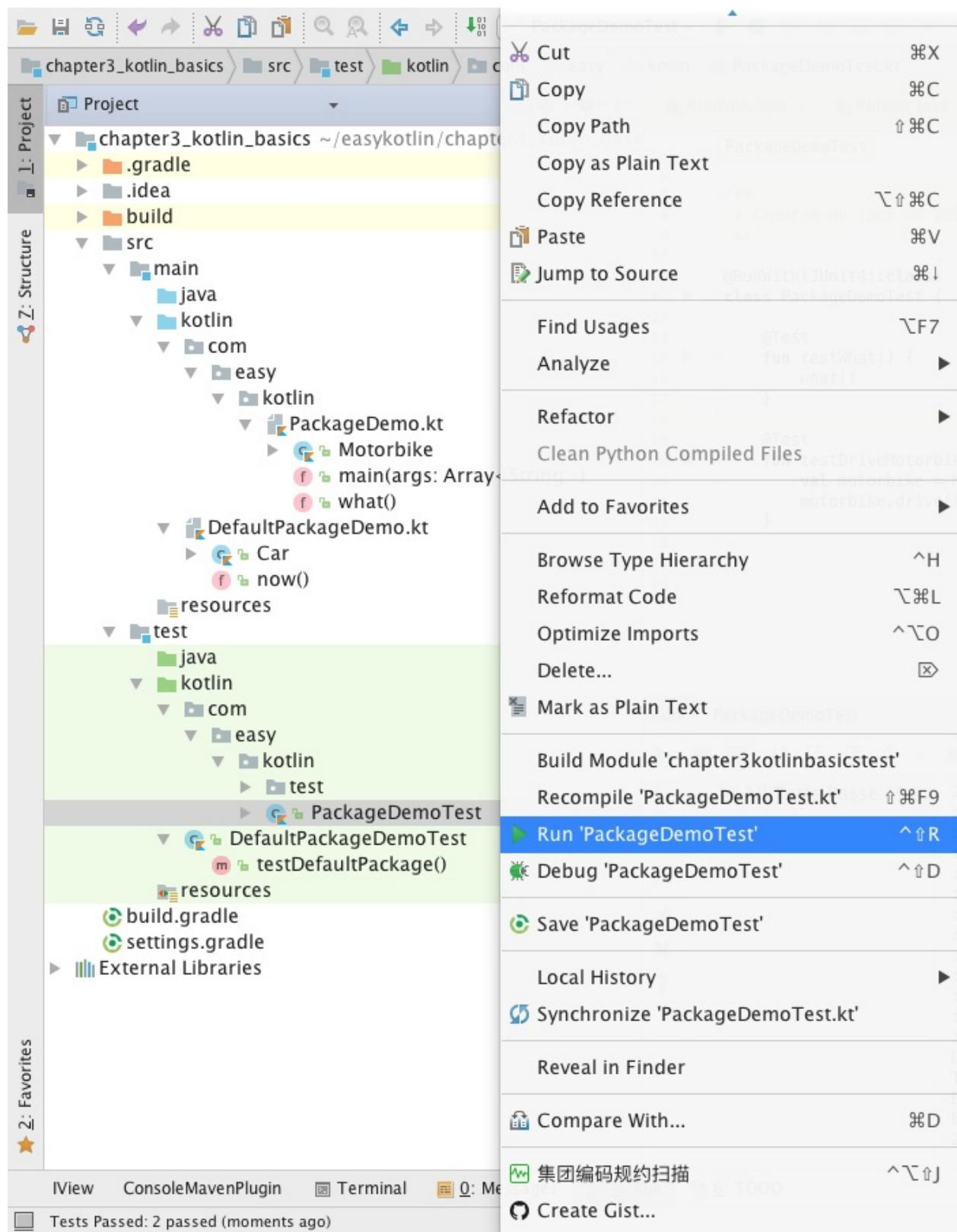
我们使用 `import com.easy.kotlin.Motorbike` 导入类，直接使用 `import com.easy.kotlin.what` 导入包级函数。

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'
```

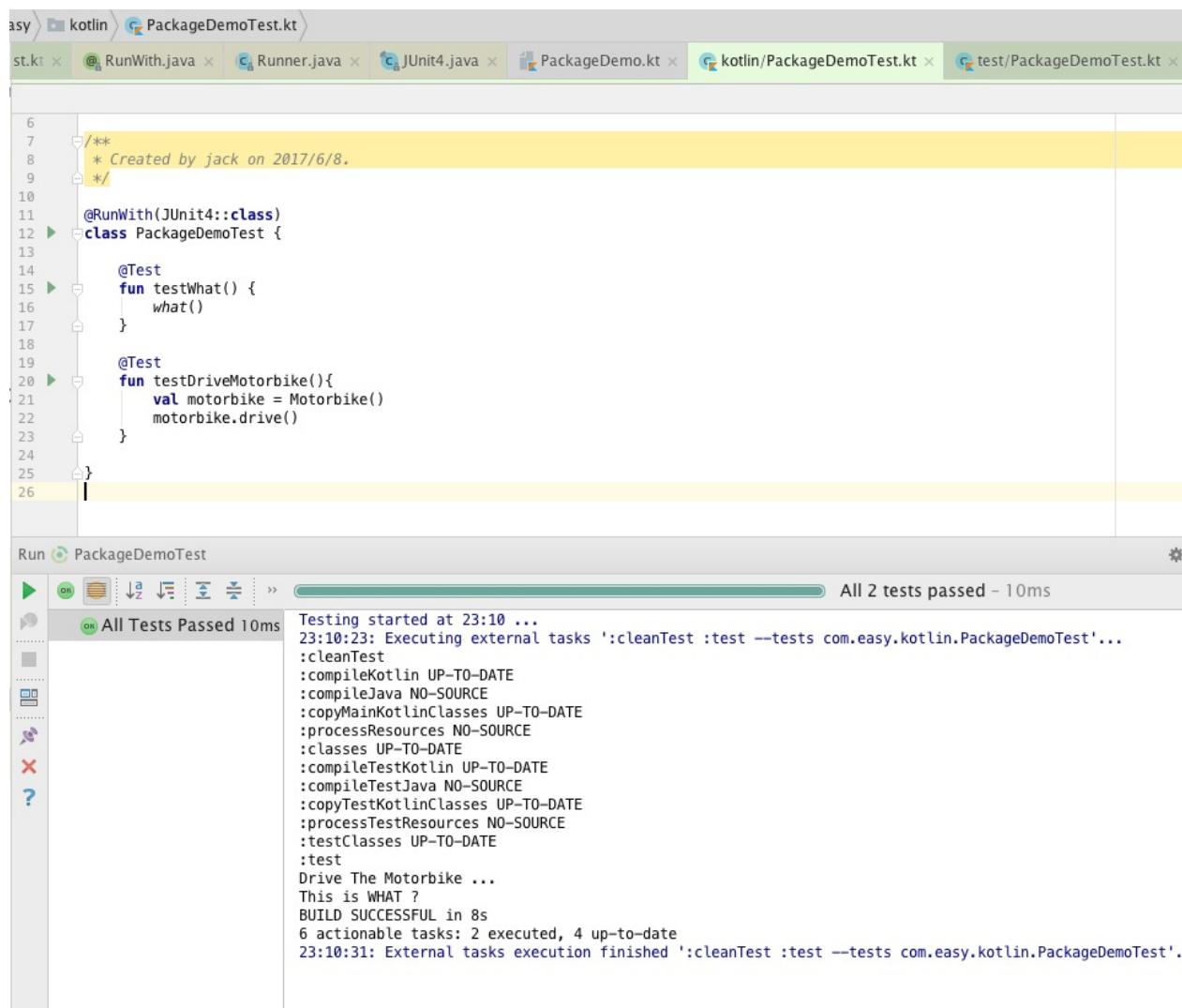
右击测试类，点击执行

package



运行结果

package



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 . . . ")
    }
}
```

```
package
```

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
```

```
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.*

package

- 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

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

```

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。

变量类型推断

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

```
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()
}
```

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)
}
```

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)
```

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。
- 如果有两条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：该标签相當于是记录了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

```

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

>>> val ex:Nothing = throw Exception("YYYYYYYYYY")
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 error: 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
...
```

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
 * /**
 *   ffff
 *   /**
 *     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))
}
```

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。

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, OVERRIDE_KEYWORD, PRIVATE_KEYWORD,
        PUBLIC_KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD,
        OUT_KEYWORD, IN_KEYWORD, FINAL_KEYWORD, VARARG_KEYWORD,
        REIFIED_KEYWORD, COMPANION_KEYWORD, SEALED_KEYWORD,
        LATEINIT_KEYWORD,
        DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEYWORD,
        TAILREC_KEYWORD, EXTERNAL_KEYWORD, ANNOTATION_KEYWORD, CROSS_INLINE_KEYWORD,
        CONST_KEYWORD, OPERATOR_KEYWORD, INFIX_KEYWORD,
        SUSPEND_KEYWORD, HEADER_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, INTERNAL_KEYWORD, PROTECTED_KEYWORD);

```

3.7.2 关键字(保留字)

```

TokenSet KEYWORDS = TokenSet.create(PACKAGE_KEYWORD, AS_KEYWORD,
    TYPE_ALIAS_KEYWORD, CLASS_KEYWORD, INTERFACE_KEYWORD,
        THIS_KEYWORD, SUPER_KEYWORD,
    ORD, VAL_KEYWORD, VAR_KEYWORD, FUN_KEYWORD, FOR_KEYWORD,
        NULL_KEYWORD,
    TRUE_KEYWORD, FALSE_KEYWORD,
    ORD, IS_KEYWORD,
        IN_KEYWORD, THROW_KEYWORD,
    D, RETURN_KEYWORD, BREAK_KEYWORD, CONTINUE_KEYWORD, OBJECT_KEYWORD,
    RD, IF_KEYWORD,
        ELSE_KEYWORD, WHILE_KEYWORD,
    ORD, DO_KEYWORD, TRY_KEYWORD, WHEN_KEYWORD,
        NOT_IN, NOT_IS, AS_SAFE,
    TYPEOF_KEYWORD
);

TokenSet SOFT_KEYWORDS = TokenSet.create(FILE_KEYWORD, IMPORT_KEYWORD,
    WHERE_KEYWORD, BY_KEYWORD, GET_KEYWORD,
        SET_KEYWORD, ABSTRACT_KEYWORD,
    CT_KEYWORD, ENUM_KEYWORD, OPEN_KEYWORD, INNER_KEYWORD,
        OVERRIDE_KEYWORD,
    PRIVATE_KEYWORD, PUBLIC_KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD,
    ORD,
        CATCH_KEYWORD, FINALLY_KEYWORD, OUT_KEYWORD, FINAL_KEYWORD, VARARG_KEYWORD, REIFIED_KEYWORD,
        DYNAMIC_KEYWORD, COMPANION_KEYWORD, CONSTRUCTOR_KEYWORD, INIT_KEYWORD, SEALED_KEYWORD,
        FIELD_KEYWORD, PROPERTY_KEYWORD, RECEIVER_KEYWORD, PARAM_KEYWORD, SETPARAM_KEYWORD,
        DELEGATE_KEYWORD,
        LATEINIT_KEYWORD,
        DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEYWORD, TAILREC_KEYWORD, EXTERNAL_KEYWORD,
        ANNOTATION_KEYWORD,
    CROSSINLINE_KEYWORD, CONST_KEYWORD, OPERATOR_KEYWORD, INFIX_KEYWORD
);

```

```

WORD,
SUSPEND_KEYWORD, HE
ADER_KEYWORD, IMPL_KEYWORD
);

```

其中，对应的关键字如下：

KtKeywordToken PACKAGE_KEYWORD ("package");	= KtKeywordToken.keyword
KtKeywordToken AS_KEYWORD ("as");	= KtKeywordToken.keyword
KtKeywordToken TYPE_ALIAS_KEYWORD ("typealias");	= KtKeywordToken.keyword
KtKeywordToken CLASS_KEYWORD ("class");	= KtKeywordToken.keyword
KtKeywordToken THIS_KEYWORD ("this");	= KtKeywordToken.keyword
KtKeywordToken SUPER_KEYWORD ("super");	= KtKeywordToken.keyword
KtKeywordToken VAL_KEYWORD ("val");	= KtKeywordToken.keyword
KtKeywordToken VAR_KEYWORD ("var");	= KtKeywordToken.keyword
KtKeywordToken FUN_KEYWORD ("fun");	= KtKeywordToken.keyword
KtKeywordToken FOR_KEYWORD ("for");	= KtKeywordToken.keyword
KtKeywordToken NULL_KEYWORD ("null");	= KtKeywordToken.keyword
KtKeywordToken TRUE_KEYWORD ("true");	= KtKeywordToken.keyword
KtKeywordToken FALSE_KEYWORD ("false");	= KtKeywordToken.keyword
KtKeywordToken IS_KEYWORD ("is");	= KtKeywordToken.keyword
KtModifierKeywordToken IN_KEYWORD .keywordModifier("in");	= KtModifierKeywordToken
KtKeywordToken THROW_KEYWORD ("throw");	= KtKeywordToken.keyword

```

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("fil
e");
KtKeywordToken FIELD_KEYWORD           = KtKeywordToken.softKeyword("f
ield");
KtKeywordToken PROPERTY_KEYWORD        = KtKeywordToken.softKeyword
("property");
KtKeywordToken RECEIVER_KEYWORD         = KtKeywordToken.softKeyword
("receiver");
KtKeywordToken PARAM_KEYWORD            = KtKeywordToken.softKeyword("p
aram");
KtKeywordToken SETPARAM_KEYWORD        = KtKeywordToken.softKeyword("s
etparam");
KtKeywordToken DELEGATE_KEYWORD         = KtKeywordToken.softKeyword("d
elegate");

```

```

elegate");
KtKeywordToken IMPORT_KEYWORD      = KtKeywordToken.softKeyword("i
mport");
KtKeywordToken WHERE_KEYWORD       = KtKeywordToken.softKeyword("w
here");
KtKeywordToken BY_KEYWORD          = KtKeywordToken.softKeyword("b
y");
KtKeywordToken GET_KEYWORD         = KtKeywordToken.softKeyword("g
et");
KtKeywordToken SET_KEYWORD         = KtKeywordToken.softKeyword("s
et");
KtKeywordToken CONSTRUCTOR_KEYWORD = KtKeywordToken.softKeyword(
"constructor");
KtKeywordToken INIT_KEYWORD        = KtKeywordToken.softKeyword(
"init");

KtModifierKeywordToken ABSTRACT_KEYWORD = KtModifierKeywordToke
n.softKeywordModifier("abstract");
KtModifierKeywordToken ENUM_KEYWORD    = KtModifierKeywordToke
n.softKeywordModifier("enum");
KtModifierKeywordToken OPEN_KEYWORD     = KtModifierKeywordToke
n.softKeywordModifier("open");
KtModifierKeywordToken INNER_KEYWORD    = KtModifierKeywordToke
n.softKeywordModifier("inner");
KtModifierKeywordToken OVERRIDE_KEYWORD = KtModifierKeywordToke
n.softKeywordModifier("override");
KtModifierKeywordToken PRIVATE_KEYWORD   = KtModifierKeywordToke
n.softKeywordModifier("private");
KtModifierKeywordToken PUBLIC_KEYWORD    = KtModifierKeywordToke
n.softKeywordModifier("public");
KtModifierKeywordToken INTERNAL_KEYWORD  = KtModifierKeywordToke
n.softKeywordModifier("internal");
KtModifierKeywordToken PROTECTED_KEYWORD = KtModifierKeywordToke
n.softKeywordModifier("protected");
KtKeywordToken CATCH_KEYWORD          = KtKeywordToken.softKeyword("c
atch");
KtModifierKeywordToken OUT_KEYWORD     = KtModifierKeywordToke
n.softKeywordModifier("out");
KtModifierKeywordToken VARARG_KEYWORD    = KtModifierKeywordToke
n.softKeywordModifier("vararg");

```

```
KtModifierKeywordToken REIFIED_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("reified");  
KtKeywordToken DYNAMIC_KEYWORD = KtKeywordToken.softKeyword("d  
ynamic");  
KtModifierKeywordToken COMPANION_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("companion");  
KtModifierKeywordToken SEALED_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("sealed");  
  
KtModifierKeywordToken DEFAULT_VISIBILITY_KEYWORD = PUBLIC_KEYWO  
RD;  
  
KtKeywordToken FINALLY_KEYWORD = KtKeywordToken.softKeyword("f  
inally");  
KtModifierKeywordToken FINAL_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("final");  
  
KtModifierKeywordToken LATEINIT_KEYWORD = KtModifierKeywordToken.  
.softKeywordModifier("lateinit");  
  
KtModifierKeywordToken DATA_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("data");  
KtModifierKeywordToken INLINE_KEYWORD = KtModifierKeywordToken.  
softKeywordModifier("inline");  
KtModifierKeywordToken NOINLINE_KEYWORD = KtModifierKeywordTo  
ken.softKeywordModifier("noinline");  
KtModifierKeywordToken TAILREC_KEYWORD = KtModifierKeywordTok  
en.softKeywordModifier("tailrec");  
KtModifierKeywordToken EXTERNAL_KEYWORD = KtModifierKeywordTo  
ken.softKeywordModifier("external");  
KtModifierKeywordToken ANNOTATION_KEYWORD = KtModifierKeyword  
Token.softKeywordModifier("annotation");  
KtModifierKeywordToken CROSSINLINE_KEYWORD = KtModifierKeywor  
dToken.softKeywordModifier("crossinline");  
KtModifierKeywordToken OPERATOR_KEYWORD = KtModifierKeywordToken.  
.softKeywordModifier("operator");  
KtModifierKeywordToken INFIX_KEYWORD = KtModifierKeywordToken.so  
ftKeywordModifier("infix");  
  
KtModifierKeywordToken CONST_KEYWORD = KtModifierKeywordToken.so
```

```

ftKeywordModifier("const");

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

KtModifierKeywordToken HEADER_KEYWORD = KtModifierKeywordToken.s
oftKeywordModifier("header");
KtModifierKeywordToken IMPL_KEYWORD = KtModifierKeywordToken.sof
tKeywordModifier("impl");

```

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 EXCLEQEQQ = new KtSingleValueToken("EXCLEQE
QQ", "!==");
KtSingleValueToken EQEQ      = new KtSingleValueToken("EQEQ",
"==");
KtSingleValueToken EXCLEQ     = new KtSingleValueToken("EXCLEQ"
, "!=");
KtSingleValueToken EXCLEXCL   = new KtSingleValueToken("EXCLEXC
L", "!!");
KtSingleValueToken ANDAND     = new KtSingleValueToken("ANDAND"
, "&&");
KtSingleValueToken OROR       = new KtSingleValueToken("OROR",
"||");
KtSingleValueToken SAFE_ACCESS = new KtSingleValueToken("SAFE_AC
CESS", "?.");
KtSingleValueToken ELVIS      = new KtSingleValueToken("ELVIS",
"?:");
KtSingleValueToken QUEST      = new KtSingleValueToken("QUEST",
"?");
KtSingleValueToken COLONCOLON = new KtSingleValueToken("COLONCO
LON", "::");
KtSingleValueToken COLON      = new KtSingleValueToken("COLON",
":");
KtSingleValueToken SEMICOLON  = new KtSingleValueToken("SEMICOL
ON", ";");
KtSingleValueToken DOUBLE_SEMICOLON = new KtSingleValueToken("D
OUBLE_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)	<code>++</code> , <code>--</code> , <code>.</code> , <code>?.</code> , <code>?</code>
	前缀 (Prefix)	<code>-</code> , <code>+</code> , <code>++</code> , <code>--</code> , <code>!</code> , <code>labelDefinition</code> <code>@</code>
	右手类型运算 (Type RHS, right-hand side class type (RHS))	<code>:</code> , <code>as</code> , <code>as?</code>
	乘除取余 (Multiplicative)	<code>*</code> , <code>/</code> , <code>%</code>
	加减 (Additive)	<code>+</code> , <code>-</code>
	区间范围 (Range)	<code>..</code>
	Infix函数	例如，给 <code>Int</code> 定义扩展 <code>infix fun Int.shl(x: Int): Int {...}</code> , 这样调用 <code>1 shl 2</code> ，等同于 <code>1.shl(2)</code>
	Elvis操作符	<code>?:</code>
	命名检查符 (Named checks)	<code>in</code> , <code>!in</code> , <code>is</code> , <code>!is</code>
	比较大小 (Comparison)	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>
	相等性判断 (Equality)	<code>==</code> , <code>\!=</code>
	与 (Conjunction)	<code>&&</code>
	或 (Disjunction)	<code> </code>
最低	赋值 (Assignment)	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>

注：Markdown表格语法：`||` 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<In
```

```
t> {  
    ...  
  
    /**  
     * Compares this value with the specified value for order.  
     * Returns zero if this value is equal to the specified other  
     * value, a negative number 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 number 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 number 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 number 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 number 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 number 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.WARNING)
    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.WARNING)
    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.WARNING)
    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.WARNING)
    public operator fun mod(other: Long): Long
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other")
)), DeprecationLevel.WARNING)
    public operator fun mod(other: Float): Float
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @Deprecated("Use rem(other) instead", ReplaceWith("rem(other")
)), DeprecationLevel.WARNING)
    public operator fun mod(other: Double): Double

    /** Calculates the remainder of dividing this value by the o
ther value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Byte): Int
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Short): Int
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Int): Int
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Long): Long
    /** Calculates the remainder of dividing this value by the o
ther value. */
    @SinceKotlin("1.1")
    public operator fun rem(other: Float): Float
    /** Calculates the remainder of dividing this value by the o
ther 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
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

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

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

@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
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b) 、 a.mod(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

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代码就显得不太友好)，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

@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
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

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Kotlin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
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

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)
```

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.isNotEmpty(): 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()
}
```

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.isNotEmpty():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.isNotEmpty(): Boolean {
    return !this.isEmpty()
}

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

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

class ExtensionsDemo {

    fun useExtensions() {
        val a = "abc"
        println(a.isNotEmpty())//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。

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`：

The screenshot shows a code editor with the following code:

```
48
49
50
51
52     val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
53     println("Before Swap:")
54     println(mmmList)
55     mmmList.mswap(0, mmmList.lastIndex)
56     println("After Swap:")
57     println(mmmList)
```

A tooltip is displayed over the call to `mmmList.lastIndex`, showing the documentation for the extension function:

`lastIndex for List<T> in com.easy.kotlin`

`(element: Int) Int`

`^↓ and ^↑ will move caret down and up in the editor >>`

空指针安全(Null-safety)

我们写代码的时候知道，在Java中NPE（NullPointerException）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 receiver 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

Kotlin入门和使用

Intro

Java 有哪些问题？

- 空引用（Null references）：连空引用的发明者都承认这是个 billion-dollar 错误（参见）。不论你费多大的功夫，你都无法避免它。因为 Java 的类型系统就是不安全的。
- 原始类型（Raw types）：我们在开发的时候总是会为了保持兼容性而卡在范型原始类型的问题上，我们都知道要努力避免 raw type 的警告，但是它们毕竟是在语言层面上的存在，这必定会造成误解和不安全因素。
- 协变数组（Covariant arrays）：你可以创建一个 string 类型的数组和一个 object 型的数组，然后把 string 数组分配给 object 数组。这样的代码可以通过编译，但是一旦你尝试在运行时分配一个数给那个数组的时候，他就会在运行时抛出异常。
- Java 8 存在高阶方法（higher-order functions），但是他们是通过 SAM 类型实现的。SAM 是一个单个抽象方法，每个函数类型都需要一个对应的接口。如果你想要创建一个并不存在的 lambda 的时候或者不存在着对应的函数类型的时候，你要自己去创建函数类型作为接口。
- 泛型中的通配符：诡异的泛型总是难以操作，难以阅读，书写，以及理解。对编译器而言，异常检查也变得很困难。
- 不够灵活，缺乏扩展能力：我们不能给不是我们自己写的 types、classes 或者 interfaces 增加新的方法。长时间以来，我们都会采用 util 类，杂乱无章地堆砌着我们代码或者或者揉在同一个 util package 里面。如果这是解决方案的话，它肯定不理想。
- 语法繁琐，不够简洁：Java 肯定不是最简洁的语言。这件事本身不是件坏事，但是事实上存在太多的常见的冗余。这会带来潜在的错误和缺陷。在这之前，我们还要处理安卓 API 带来的问题。

Features

- Lambdas
- Data classes
- Function literals
- Extension functions
- Null safety
- Smart casts
- String templates
- Properties
- Class delegation
- Type inference
- Range expressions

Kotlin 有几个核心的目标：

- 简约：帮你减少实现同一个功能的代码量。
- 易懂：让你的代码更容易阅读，同时易于理解。
- 安全：移除了你可能会犯错误的功能。
- 通用：基于 JVM 和 Javascript，你可以在很多地方运行。
- 互操作性：这就意味着 Kotlin 和 Java 可以相互调用，目标是 100% 兼容。

Kotlin 的特性

刚才我们提到过的这些缺陷，Kotlin 通常直接移除了那些特性。同时它也加了一些新的特性：

- Lambda 表达式
- 数据类 (Data classes)
- 函数字面量和内联函数 (Function literals & inline functions)
- 函数扩展 (Extension functions)
- 空安全 (Null safety)
- 智能转换 (Smart casts)
- 字符串模板 (String templates)
- 主构造函数 (Primary constructors)
- 类委托 (Class delegation)

- 类型推断 (Type inference)
- 单例 (Singletons)
- 声明点变量 (Declaration-site variance)
- 区间表达式 (Range expressions)

我们将在这篇文章里提及以上大多数特性。Kotlin之所以能跟随着 JVM 的生态系统不断地进步，是因为他没有任何限制。它编译出来的正是 JVM 字节码。在 JVM 看来，它就跟其他语言一样样的。事实上，如果你在 IntelliJ 或者 Android Studio 上用 Kotlin 的插件，它自带里一个字节码查看器，可以显示每个方法生成的 JVM 字节码。

Syntax

Variables

```
// hello.kt
package com.kotlin.demo

val a: Int = 1
val b = 1
val c: Int
c = 1
// c = 2
var x = 5
x += 1

// val text: String = "Hello, World"
val text = "Hello, World"

// val ints: Array<Int> = arrayOf<Int>(1,2,3,4)
val ints = arrayOf(1,2,3,4)

var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok
```

类型声明

- 包的声明应处于源文件顶部，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。
- 在 Kotlin 里，得把参数名放在前面，参数类型放在后面，用一个冒号隔开。
- 常量（使用 val 关键字声明），相当于 Java 里的 final，如果没有初始值，声明常量时，常量的类型不能省略。
- 变量（使用 var 关键字声明），类型可以省略，自动推断出 Int 类型
- 正如 Java 和 JavaScript，Kotlin 支持行注释及块注释，与 Java 不同的是，Kotlin 的块注释可以嵌套。
- 当某个变量的值可以为 null 的时候，必须在声明处的类型后添加 ? 来标识该引用可为空。

类型推导

你可能在其他语言中看到过类型推导。在 Java 里，我们需要自己声明类型，变量名，以及数值。

在 Kotlin 里，顺序有些不一样，你先声明变量名，然后是类型，然后是分配值。很多情况下，你不需要声明类型。一个字符串字面量足以指明这是个字符串类型。字符，整形，长整形，单浮点数，双浮点数，布尔值都是可以无需显性声明类型的。

只要 Kotlin 可以推导，这个规则同样适用与其他一些类型。通常，如果是局部变量，当你在声明一个值或者变量的时候你不需要指明类型。在一些无法推导的场景里，你才需要用完整的声明变量语法指明变量类型。

Define Function

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}  
  
fun sum(a: Int, b: Int) = a + b  
  
fun printSum(a: Int, b: Int): Unit {  
    print(a + b)  
}  
  
fun printSum(a: Int, b: Int) {  
    print(a + b)  
}  
  
fun main(args: Array<String>) {  
    println("Hello, World!")  
}
```

定义函数

- 声明函数的关键字是 fun，fun 后面跟的是函数的名称，然后括号包裹起来的是函数参数，这个跟 Java 类似。

- 这是带有两个 Int 参数、返回 Int 的函数。可以将表达式作为函数体、返回值类型自动推断的函数。函数返回无意义的值，Unit 返回类型可以省略。Kotlin 的函数和 Java 类似，但不需要定义在类内部。
- 函数的返回类型在最后，这个跟 Java 放在前面形式不太一样。如果一个函数没有返回任何类型，可以返回一个 Unit 类型，当然也可以省略。调用 Kotlin 标准库中的函数 println 就能打印 Hello World 出来，实际上它最终调用了 Java 的 system.out.println。

If Expressions

```

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

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

// As expression
val max = if (a > b) a else b

val max = if (a > b) {
    print("Choose a")
    a
}
else {
    print("Choose b")
    b
}

```

If表达式

- 在 Kotlin 中，if 是一个表达式，即它会返回一个值。因此就不需要三元运算符（条件？然后：否则），因为普通的 if 就能胜任这个角色。

- if的分支可以是代码块，最后的表达式作为该块的值。如果你使用 if 作为表达式而不是语句（例如：返回它的值或者把它赋给变量），该表达式需要有 else 分支。

Loop

```
fun forLoop1(args: Array<String>) {
    for (arg in args) {
        print(arg)
    }
}

fun forLoop2(args: Array<String>) {
    for (i in args.indices) {
        print(args[i])
    }
}

fun whileLoop1(args: Array<String>) {
    var i = 0
    while (i < args.size)
        print(args[i++])
}
```

FOR循环

- for 循环可以对任何提供迭代器（iterator）的对象进行遍历，循环体可以是一个代码块。
- for 可以循环遍历任何提供了迭代器的对象。即：
 - 有一个成员函数或者扩展函数 iterator()，它的返回类型
 - 有一个成员函数或者扩展函数 next()，并且
 - 有一个成员函数或者扩展函数 hasNext() 返回 Boolean
- 对数组的 for 循环会被编译为并不创建迭代器的基于索引的循环，注意这种“在区间上遍历”会编译成优化的实现而不会创建额外对象，如果你想要通过索引遍历一个数组或者一个 list，你可以用 .indices 或 array.withIndex。

- while 和 do..while 照常使用。

When

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    3 -> print("x == 3")
    4 -> print("x == 4")
    else -> print("otherwise")
}

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")
}
```

When语句和表达式

- when 取代了类 C 语言的 switch 操作符
- when 将它的参数和所有的分支条件顺序比较，直到某个分支满足条件
- when 既可以被当做表达式使用也可以被当做语句使用
 - 如果它被当做表达式， 符合条件的分支的值就是整个表达式的值
 - 如果当做语句使用，则忽略个别分支的值
- 如果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
- 我们可以用任意表达式（而不只是常量）作为分支条件
- 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
- 另一种可能性是检测一个值是 (is) 或者不是 (!is) 一个特定类型的值
- when 也可以用来取代 if-else if 链
- 如果不提供参数，所有的分支条件都是简单的布尔表达式，而当一个分支的条件为真时则执行该分支

Returns

```

return.
break.
continue.

loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}

fun foo() {
    ints.forEach lit@ {
        if (it == 0) return@lit
        print(it)
    }
}

```

返回和跳转

- Kotlin 有三种结构化跳转操作符
 - return.默认从最直接包围它的函数或者匿名函数返回。
 - break.终止最直接包围它的循环。
 - continue.继续下一次最直接包围它的循环。
- 在 Kotlin 中任何表达式都可以用标签（label）来标记。
- 我们可以用标签限制 break 或者continue，标签限制的 break 跳转到刚好位于该标签指定的循环后面的执行点。 continue 继续标签指定的循环的下一次迭代。
- Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。 标签限制的 return 允许我们从外层函数返回。最重要的一个用途就是从 lambda 表达式中返回。
- 这个 return 表达式从最直接包围它的函数即 foo 中返回。（注意，这种非局部的返回只支持传给内联函数的 lambda 表达式。）如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return。

Strings

```

for (c in str) {
    println(c)
}

val s = "Hello, world!\n"

val text = """
    for (c in "foo")
        print(c)
"""

val s = "abc"
val str = "$s.length is ${s.length}"

var args = arrayOf("Cat", "Dog", "Rabbit")
print("Hello ${args[0]}")

```

字符串说明

- 字符串用 `String` 类型表示。字符串是不可变的。字符串的元素——字符可以用索引运算符访问: `s[i]`。可以用 `for` 循环迭代字符串。
- Kotlin 有两种类型的字符串字面值: 转义字符串可以有转义字符, 以及原生字符串可以包含换行和任意文本。转义字符串很像 Java 字符串。
- 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。
- 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, 或者用花括号扩起来的任意表达式。
- 原生字符串和转义字符串内部都支持模板。如果你需要在原生字符串中表示字面值 \$ 字符 (它不支持反斜杠转义), 你可以用三引号语法。

- 字符串字面值用单引号括起来: '1'。特殊字符可以用反斜杠转义。支持这几个转义序列：`\t`、`\b`、`\n`、`\r`、`'`、`"`、`\` 和 `$`。编码其他字符要用 Unicode 转义序列语法：`\uFF00`。

Basic Types

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

```
val a: Int = 10000
print(a === a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

数据类型

- 在 Kotlin 中，所有东西都是对象，在这个意义上讲所以我们可以在任何变量上调用成员函数和属性。有些类型是内置的，因为他们的实现是优化过的。但是用户看起来他们就像普通的类。本节我们会描述大多数这些类型：数字、字符、布尔和数组。
- Kotlin 处理数字在某种程度上接近 Java，但是并不完全相同。例如，对于数字没有隐式拓宽转换（如 Java 中 `int` 可以隐式转换为 `long`——译者注），另外有些情况的字面值略有不同。
- 注意在 Kotlin 中字符不是数字。
- 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。

- 由于不同的表示方式，较小类型并不是较大类型的子类型。如果它们是的话，就会出现下述问题。

```
// 假想的代码，实际上并不能编译：
val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)
val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)
print(a == b) // 惊！这将打印 "false" 鉴于 Long 的 equals() 检测其他
部分也是 Long
```

- 因此较小的类型不能隐式转换为较大的类型。这意味着在不进行显式转换的情况下我们不能把 Byte 型值赋给一个 Int 变量。
- 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
```

编码风格

- 使用驼峰不要使用下划线
- 类型名是用大写字母开头
- 方法和属性名小写开头
- 使用四个空格的缩进
- 公开方法应该有文档
- 冒号前后有空格，大括号两边有空格，箭头两边有空格，举例

```
list.filter { it > 10 }.map { element -> element * 2 }
```

Classes and Objects

Classes

```
class Invoice {  
}  
  
class Empty  
  
class Person(name: String) {  
}  
  
class Person(name: String) {  
    val customName = name.toUpperCase()  
}  
  
class Person(val firstName: String, val lastName: String, var age  
: Int) {  
    // ...  
}
```



定义一个类

- 类的定义要通过 `class` 关键字，跟 Java 里的一样，关键字后是类名。Kotlin 有一个主构造函数，我们可以直接将构造函数参数列表写在类的声明处，还可以直接用 `var` 或者 `val` 关键字将参数声明为成员变量（又称：类属性）
- 这个类声明被花括号包围，包括类名、类头(指定其类型参数,主构造 函数等)和这个类的主干。类头和主干都是可选的；如果这个类没有主干，花括号可以被省略。
- 在Kotlin中的类可以有主构造函数和一个或多个二级构造函数。主构造 函数是类头的一部分:它跟在这个类名后面 (和可选的类型参数) 。
- 如果这个主构造函数没有任何注解或者可见的修饰符，这个`constructor` 关键字可以被省略。
- 请注意，主构造的参数可以在初始化模块中使用。它们也可以在 类体内声明初始化的属性。事实上，声明属性和初始化主构造函数,Kotlin有简洁的语法

```
class Person(val firstName: String, val lastName: String, var age: Int) {
    // ...
}
```

Constructors

```
class Person constructor(name: String) {
    val fixName = name.toUpperCase()
}

class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}

class Person(name: String) {
    init {
        logger.info("Person initialized with value ${name}")
    }
}

val invoice = Invoice()
val customer = Customer("Joe Smith")
```

构造函数

Kotlin 中，类可以拥有多个构造函数，这一点跟 Java 类似。但你也可以有一个主构造函数。下面的例子是我们从上面的例子里衍生出来的，在函数头里添加了一个主构造函数

当然，更好的方法是：直接在主构造函数里定义这些属性，定义的方法是在参数名前加上 var 或者 val 关键字，val 是代表属性是常量。

在主构造函数里，可以直接用这些参数变量赋值给类的属性，或者用构造代码块来实现初始化。

- 这个主构造函数不能包含任何的代码。初始化的代码可以被放置在initializer blocks（初始的模块），以init为关键字作为前缀
- 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）或者是只读的（val）
- 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
- 类也可以拥有被称为“二级构造函数”（为了实现Kotlin向Java一样拥有多个构造函数），通常被加上前缀constructor
- 如果类有一个主构造函数，每个二级构造函数需要委托给主构造函数，直接或间接地通过另一个二级函数。委托到另一个使用同一个类的构造函数用this关键字
- 要创建一个类的实例，我们调用构造函数，就好像它是普通的函数

Inheritance

```
class Example // Implicitly inherits from Any

open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

```
open class Base {
    open fun v() {}
    fun nv() {}
}

class Derived() : Base() {
    override fun v() {}
}

open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

类的继承

- 在Kotlin所有的类中都有一个共同的父类Any，这是一个默认的父类且没有父类型声明
- Any不属于java.lang.Object;特别是，它并没有任何其他任何成员，甚至连equals()，hashCode()和toString()都没有。
- 要声明一个明确的父类，我们把类型放到类头冒号之后，父类可以（并且必须）在声明继承的地方，用原始构造函数初始化。
- 如果类没有主构造，那么每个次级构造函数初始化基本类型 使用super{ : .keyword}关键字，或委托给另一个构造函数做到这一点。注意，在这种情况下，不同的二级构造函数可以调用基类型的不同的构造函数。

Overriding Members

```

open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // interface members are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // The compiler requires f() to be overridden:
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}

```

成员覆盖

- 我们之前提到过，Kotlin力求清晰显式。不像Java中，Kotlin需要明确的标注覆盖的成员（我们称之为open）和重写的函数。（继承父类并覆盖父类函数时，Kotlin要求父类必须有open标注，被覆盖的函数必须有open标注，并且子类的函数必须加override标注。）
- Derived.v()函数上必须加上override标注。如果没写，编译器将会报错。如果父类的这个函数没有标注open，则子类中不允许定义同名函数，不论加不加override。在一个final类中（即没有声明open的类），函数上也不允许加open标注。
- 成员标记为override{ : .keyword}的本身是开放的，也就是说，它可以在子类中重写。如果你想禁止重写的，使用final{ : .keyword}关键字
- 在Kotlin中，实现继承的调用通过以下规则：如果一个类继承父类成员的多种实现方法，可以直接在子类中引用，它必须重写这个成员，并提供其自己的实现（当然也可以使用父类的）。为了表示从中继承的实现而采取的父类型，我们使用super{ : .keyword}在尖括号，如规范的父名super
- 类和其中的某些实现可以声明为abstract{ : .keyword}。抽象成员在本类中可以不用实现。
- 需要注意的是，我们并不需要标注一个抽象类或者函数为open - 因为这不言而喻。我们可以重写一个open非抽象成员使之为抽象的。

Properties

```

public class Address {
    public val code:Int = 10015
    public var name: String = ...
    public var city: String = ...
    public var state: String? = ...
    public var zip: String = ...
}

fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}

const val DEPRECATED: String = "deprecated"
const val SOCKET_TIMEOUT = 30*1000L

```

类的属性

- Kotlin的类可以有属性。这些声明是可变的，用关键字var或者使用只读关键字val
- 要使用一个属性，只需要使用名称引用即可，就相当于Java中的公共字段
- 注意公有的API(即public和protected)的属性，类型是不做推导的。~~~这么设计是为了防止改变初始化器时不小心改变了公有API。

常量属性的要求

Properties the value of which is known at compile time can be marked as compile time constants using the const modifier. Such properties need to fulfil the following requirements:

- Top-level or member of an object
- Initialized with a value of type String or a primitive type
- No custom getter

Getters and Setters

```

val isEmpty: Boolean
    get() = this.size == 0

var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value)
    }

var setterVisibility: String = "abc"
// Initializer required, not a nullable type
private set // the setter is private

```

Getter和Setter

声明一个属性的完整语法

```

var <propertyName>: <PropertyType> [= <property_initializer>]
    [<getter>]
    [<setter>]

```

- 上面的定义中，初始器(initializer)、getter 和 setter 都是可选的。属性类型(PropertyType)如果可以从初始器或者父类中推导出来，也可以省略。
- 一个只读属性的语法和一个可变的语法有两方面的不同：1、只读属性的用val开始代替var 2、只读属性不许setter。
- 如果你需要改变一个访问器或注释的可见性,但是不需要改变默认的实现,您可以定义访问器而不定义它的实例。
- 在Kotlin不能有字段。然而,有时有必要有使用一个字段在使用定制的访问器的时候。对于这些目的,Kotlin提供自动支持,在属性名后面使用 field标识符。
- 编译器会查看访问器的内部，如果他们使用了实际字段（或者访问器使用默认实现），那么将会生成一个实际字段，否则不会生成。

Interface

```
interface MyInterface {  
    val property: Int // abstract  
  
    val propertyWithImplementation: String  
        get() = "foo"  
  
    fun bar()  
    fun foo() {  
        // optional body  
    }  
}  
  
class Child : MyInterface {  
    override val property: Int = 29  
  
    override fun bar() {  
        // body  
    }  
}
```

接口定义

- 使用关键字 interface 来定义接口。Kotlin 的接口与 Java 8 类似，既包含抽象方法的声明，也包含实现。与抽象类不同的是，接口无法保存状态。它可以有属性但必须声明为 abstract或提供访问器实现。
- 一个类或者对象可以实现一个或多个接口。
- 实现多个接口时，可能会遇到接口方法名同名的问题。D 可以不用重写 bar()，因为它实现了 A 和 B，因而可以自动继承 B 中 bar() 的实现，但是两个接口都定义了方法 foo()，为了告诉编译器 D 会继承谁的方法，必须在 D 中重写 foo()。

Data Class

```
// equals()/hashCode()/toString()/componentN()/copy()

data class User(val name: String, val age: Int)
data class User(val name: String = "", val age: Int = 0)

fun copy(name: String = this.name, age: Int = this.age)
    = User(name, age)

val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)

val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age")
// prints "Jane, 35 years of age"
```

数据类/POJO

- 我们经常创建一些只是处理数据的类。在这些类里的标准功能经常是衍生自数据。在Kotlin中，这叫做 **数据类** 并标记为**data**。
- 编译器自动从在主构造函数定义的全部特性中得到以下成员：
 - equals()/hashCode() 对，
 - toString() 格式是 "User(name=John, age=42)"，
 - componentN() functions 对应按声明顺序出现的所有属性，
 - copy() 函数（见下面）。
- 如果有某个函数被明确地定义在类里或者被继承，编译器就不会生成这个函数。
- 在JVM中，如果生成的类需要含有一个无参的构造函数，则所有的属性必须有默认值。
- 在很多情况下，我们我们需要对一些属性做修改而其他的不变。这就是**copy()** 这个方法的来源。对于上文的User类，应该是这么实现这个方法。
- 在标准库提供了Pair和Triple。在很多情况下，即使命名数据类是一个更好的设计选择，因为这能让代码可读性更强。

Destructuring Declarations

```

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

val person = Person("John", 20)
val (name, age) = person

val name = person.component1()
val age = person.component2()

for ((a, b) in collection) { ... }
for ((key, value) in map) {
}

val (result, status) = function(...)

operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K,
V>>
    = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()

```

解构声明

- 有时把一个对象解构成很多变量很比较方便，这种语法叫做解构声明。一个解构声明同时创造多个变量。我们申明了两个新变量：name 和 age，并且可以独立使用他们。
- component1() 和 component2() 函数是 principle of conventions widely 在 Kotlin 中的另一个例子。(参考运算符如 +, *, for-loops 等) 任何可以被放在解构声明右边的和组件函数的需求数字都可以调用它。当然，这里可以有更多的如 component3() 和 component4()。
- 变量 a 和 b 从调用从 component1() 和 component2() 返回的集合collection中的对象。
- 让我们从一个函数中返回两个变量。例如，一个结果对象和一些排序的状态。在Kotlin中一个简单的实现方式是申明一个data class并且返回他的实例。

- 可能最好的遍历一个映射的方式就是这样，实现这个接口，于是你可以自由的使用解构声明 for-loops 来操作映射(也可以用在数据类实例的集合等)。

Nested and Inner Classes

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

嵌套类和内部类

- 在类的内部可以嵌套其他的类，相当于Java里的static内部类。
- 为了能被外部类访问一个类可以被标记为内部类 (“inner” 关键词)。内部类会带有一个来自外部类的对象的引用

Enum

```

enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },
    TALKING {
        override fun signal() = WAITING
    };
    abstract fun signal(): ProtocolState
}

```

枚举

- 枚举类的最基本应用是实现类型安全的多项目集合。其中每一个常量（NORTH，SOUTH.....）都是一个对象。每一个常量用逗号“,”分隔。
- 枚举实例也可以被声明为他们自己的匿名类，并同时包含他们相应原本的方法和覆盖基本方法。注意如果枚举类定义了任何成员，你需要像JAVA一样把枚举实例的定义和成员定义用分号分开。
- 像JAVA一样，枚举类在Kotlin中有合成方法。它允许列举枚举实例并且通过名称返回枚举实例。下面是应用实例 (假设枚举实例名称是EnumClass)。
- 枚举常量也可以实现Comparable 接口。他们会依照在枚举类中的定义先后以自然顺序排列。

Object

```

val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}
print(adHoc.x + adHoc.y)

// Singleton
object Resource {
    val name = "Name"
}

open class A(x: Int) {
    public open val y: Int = x
}

interface B {...}

val ab = object : A(1), B {
    override val y = 15
}

```

对象和单例

- 有些时候我们需要创建一个对某些类做了轻微改变的一个对象，而不用为了它显式地定义一个新的子类。Java把这种情况处理为匿名内部类。在Kotlin稍微推广了这个概念，称它们为对象表达式和对象声明。
- 对象表达式：如果父类型有一个构造函数，合适的构造函数参数必须传递给它。多个父类型用逗号隔开，跟在冒号后面。
- 或许，我们需要的仅是无父类的一个对象，那么我们可以简单地写为adHoc这种。
- 就像Java的匿名内部类，在对象表达式里代码可以访问封闭的作用域（但与Java不同的是，它能访问非final修饰的变量）。
- 对象声明：单例模式是一种非常有用的模式，而在Kotlin（在Scala之后）中很容易就能声明一个单例。DataProviderManager被称为对象声明。如果有一个object关键字在名字前面，这不能再被称为一个表达式。我们不能把这样的东

虽然赋值给变量，但我们可以直接通过它的名字来引用它。

对象表达式与对象声明语义上的不同

- 当对象声明被第一次访问的时候，它会被延迟（lazily）初始化
- 当对象表达式被用到的时候，它会被立即执行（并且初始化）

Companion

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

interface Factory<T> {
    fun create(): T
}
class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}

val x = MyClass.Companion
```

伴生对象

Kotlin 移除了 static 的概念。通常用 companion object 来实现类似功能。

- 伴生对象：一个对象声明在一个类里可以加上 companion 这个关键字
- 伴生对象的成员可以使用类名称作为限定符来调用
- 使用 companion 关键字时候，伴生对象的名称可以省略
- 注意，虽然伴生对象的成员在其他语言中看起来像静态成员，但在运行时它们仍然是实体的实例成员，举例来说，我们能用它实现接口
- 然而，在 JVM 中，如果你使用 @JvmStatic 注解，你可以让伴生对象的成员生成为实际存在的静态方法和域

Class Delegation

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print() // prints 10  
}
```

类的委托

委托是一个大家都知道的设计模式，Kotlin 把委托视为很重要的语言特性。

- 委托模式是实现继承的一个有效方式。Kotlin 原生支持它。一个类 Derived 可以从一个接口 Base 继承并且委托所有的共有方法为具体对象。
- 在父类 Derived 中的 by-语句表示 b 将会被储存在 Derived 的内部对象中，并且编译器会生成所有的用于转发给 b 的 Base 的方法。

Delegated Properties

```

class Example {
    var p: String by Delegate()
}

class Delegate {
    operator fun getValue(thisRef: Any?,
                         property: KProperty<*>): String {
        return "$thisRef, thank you for delegating
        '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?,
                         property: KProperty<*>, value: String) {
        println("$value has been assigned to
        '${property.name}' in $thisRef.'")
    }
}

val e = Example()
println(e.p)
// Example@33a17727, thank you for delegating 'p' to me!

e.p = "NEW"
// NEW has been assigned to 'p' in Example@33a17727.

```

委托属性

有一些种类的属性，虽然我们可以在每次需要的时候手动实现它们，但是如果能够把他们只实现一次并放入一个库同时又能够一直使用它们那会更好。例如：

- 延迟属性（lazy properties）：数值只在第一次被访问的时候计算。
- 可观察属性（observable properties）：监听器得到关于这个特性变化的通知，
- 把所有属性储存在一个map中，而不是每个在单独的字段里。为了支持这些(或者其他)例子，Kotlin 采用 委托属性。

当我们读取一个Delegate的委托实例 p , Delegate中的getValue()就被调用, 所以它第一变量就是我们从 p 读取的实例, 第二个变量代表 p 自身的描述。(例如你可以用它的名字)。

类似的，当我们给 p 赋值, setValue() 函数就被调用。前两个参数是一样的，第三个参数保存着将要被赋予的值

属性委托要求

这里我们总结委托对象的要求。

对于一个只读属性(如 val), 一个委托一定会提供一个 getValue 函数来获取下面的参数:

- 接收者 — 必须与属性所有者类型相同或者是其父类(对于扩展属性，类型范围允许扩大),
- 包含数据 — 一定要是 KProperty<*> 的类型或它的父类型,
- 这个函数必须返回同样的类型作为属性 (或者子类型)

对于一个可变属性(如 var), 一个委托需要额外地提供一个函数 setValue 来获取下面的参数:

- 接收者 — 同 getValue(),
- 包含数据 — 同 getValue(),
- 新的值 — 必须和属性同类型或者是他的父类型。

getValue() 或/和 setValue() 函数可能会作为代理类的成员函数或者扩展函数来提供。当你需要代理一个属性给一个不是原来就提供这些函数的对象的时候，后者更为方便。两种函数都需要用 operator 关键字来进行标记

标准委托

标准库中对于一些有用的委托提供了工厂 (factory) 方法。

By Lazy

```
val lazyValue: String by lazy {
    Log.v("Lazy", "Lazy Init")
    "Hello, Lazy!"
}

fun lazyTest() {
    Log.d("Lazy", lazyValue)
    Log.d("Lazy", lazyValue)
}
```

```
V/Lazy: Lazy Init
D/Lazy: Hello, Lazy!
D/Lazy: Hello, Lazy!
```

延迟属性 Lazy

函数 `lazy()` 接受一个 `lambda` 然后返回一个可以作为实现延迟属性的委托 `Lazy` 实例来: 第一次对于 `get()` 的调用会执行 (之前) 传递到 `lazy()` 的 `lambda` 表达式并记录结果, 后面的 `get()` 调用会直接返回记录的结果。

默认地, 对于 `lazy` 属性的计算是同步锁 (`synchronized`) 的: 这个值只在一个线程被计算, 并且所有的线程会看到相同的值。如果初始化代理的同步锁不是必须的, 以至于多个线程可以同步地执行, 那么将 `LazyThreadSafetyMode.PUBLICATION` 作为一个变量传递给 `lazy()` 函数。

而且如果你确定初始化将总是发生在单个线程, 那么你可以使用 `LazyThreadSafetyMode.NONE` 模式, 它不会有任何线程安全的保证和相关的开销。

Observable

```

import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}

// output
// <no name> -> first
// first -> second

```

可观察属性 Observable

Delegates.observable() 需要两个参数：初始值和handler。这个 handler 会在每次我们给赋值的时候被调用 (在工作完成前). 它有三个参数:一个被赋值的属性，旧的值和新的值

这个例子输出：

-> first first -> second 如果你想有能力来截取和“否决”它分派的事件，就使用 vetoable() 取代 observable(). 被传递给 vetoable 的handler会在属性被赋新的值之前执行

By Map

```

class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int      by map
}

val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))

println(user.name) // Prints "John Doe"
println(user.age) // Prints 25

class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int      by map
}

// custom impl by Json

```

把属性储存在map中

一个参加的用例是在一个map里存储属性的值。这经常出现在解析JSON或者做其他的“动态”的事情应用里头。在这样的情况下，你需要使用map的实例本身作为代理用于代理属性

在这个例子中，构造函数会接收一个map参数，委托会从这个map中取值(通过string类型的key，就是属性的名字)，对于var的变量，我们可以把只读的Map换成MutableMap就可以了

Functions and Lambdas

Function Declarations

```
fun double(x: Int): Int {  
}  
val result = double(2)  
  
infix fun Int.shl(x: Int): Int {  
    ...  
}  
1 shl 2  
  
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()): Unit {  
    ...  
}  
  
fun printHello(name: String?): Unit {  
    if (name != null)  
        println("Hello ${name}")  
}  
  
fun printHello(name: String?) {  
    ...  
}
```

函数声明

在Kotlin中，函数声明使用关键字 fun

函数用途

调用函数使用传统的方法，调用成员函数使用点表达式

中缀表示法

函数还可以用中缀表示法，当：1. 他们是成员函数 或者 扩展函数；2. 他们只有一个参数；3. 使用infix关键字声明

单个表达式函数

当一个函数返回单个表达式，花括号可以省略并且主体由 = 符号之后指定。显式地声明返回类型可选时，这可以由编译器推断。

显式地返回类型

函数模块体必须显式地指定返回类型，除非是用于返回Unit，在这种情况下，它是可选的。Kotlin不推断返回类型与函数在模块体的功能，因为这些功能可能在模块体有复杂的控制流程，对于阅读者（有时甚至编译器）来说返回类型将不明显。

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是一种只有一个值 - Unit`。这个值不需要显式地返回。Unit返回类型声明也是可选的，可以省略。

Function Arguments

```
fun reformat(str: String,
            normalizeCase: Boolean = true,
            upperCaseFirstLetter: Boolean = true,
            wordSeparator: Char = ' ')
    ...
}

reformat(str, true, true, false, '_')
reformat(str,
        normalizeCase = true,
        upperCaseFirstLetter = true,
        wordSeparator = '_'
)
reformat(str, wordSeparator = '_')

val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

参数

函数参数是使用Pascal表达式，即 name: type。参数用逗号隔开。每个参数必须有显式类型。

默认参数(缺省参数)

函数参数有默认值，当对应的参数是省略。与其他语言相比可以减少数量的过载。默认值定义使用后 = 类型的值。

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非常方便。使用命名参数我们可以使代码更具有可读性。可以省略部分参数。

Function Usage

```
fun double(x: Int): Int = x * 2
fun double(x: Int) = x * 2

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
}

val list = asList(1, 2, 3)

val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)

class Sample() {
    fun foo() { print("Foo") }
}

Sample().foo()
```

数量可变的参数(可变参数)

函数的（通常最后一个）参数可以使用`vararg`修饰。内部函数vararg类型T是可见的arrayT,即上面的例子中的ts变量是Array类型。当我们调用vararg函数，我们可以一个接一个传递参数，例如 `asList(1, 2, 3)`或者，如果我们已经有了一个数组并希望将其内容传递给函数，我们使用spread操作符（在数组前面加*）

函数作用域(函数范围)

在Kotlin中函数可以在文件顶级声明，这意味着您不需要像一些语言如Java、C#或Scala那样创建一个类来持有一个函数。此外除了顶级函数功能，Kotlin函数也可以在局部声明，作为成员函数和扩展函数。

局部函数

Kotlin提供局部函数,即一个函数在另一个函数中，局部函数可以访问外部函数的局部变量（即闭包），所以在上面的例子，the visited是局部变量.

成员函数

成员函数是一个函数,定义在一个类或对象里，成员函数调用点符号

Higher-order Functions

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

高阶函数

这是个新奇的术语，它指的是函数可以接收函数，或者函数可以返回函数。

高阶函数是一种能用函数作为参数或者返回值为函数的一种函数。lock()是高阶函数中一个比较好的例子，它接收一个lock对象和一个函数，获得锁，运行传入的函数，并释放锁。

我们分析一下上面的代码：函数body拥有函数类型: $() \rightarrow T$ 所以body应该是一个不带参数并且返回T类型的值的函数。它在try代码块中调用，被lock保护的，当lock()函数被调用时返回他的值。

如果我们想调用lock()函数，我们可以把另一个函数传递给它作为参数(详见 函数引用)。

内联函数

使用内联函数有时能提高高阶函数的性能。

Higher-order Functions

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}

val doubled = ints.map { it -> it * 2 }

strings filter {it.length == 5} sortBy {it} map {it.toUpperCase()}

val names = listOf("Jake", "Jesse", "Matt", "Alec")
val jakes = names.filter { it == "Jake" }
```

高阶函数

另一个高阶函数的例子是 map() (MapReduce)

这些约定可以写成 LINQ-风格 的代码：

```
strings filter {it.length == 5} sortBy {it} map {it.toUpperCase() }
```

在Kotlin中，如果函数的最后一个参数是一个函数，那么该参数可以在括号外指定：

```
lock (lock) {
    sharedResource.operation()
}
```

Function Types

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max, it))
            max = it
    return max
}

max(strings, { a, b -> a.length() < b.length() })

fun compare(a: String, b: String): Boolean = a.length() < b.length()
```

函数类型

对于一个接收一个函数作为参数的函数，我们必须为该参数指定一个函数类型。

max函数是一个高阶函数，也就是说他的第二个参数是一个函数。这个参数是一个表达式，但它本身也是一个函数，也就是函数字面量。

参数 less 是一个 (T, T) -> Boolean类型的函数，也就是说less函数接收两个T类型的参数并返回一个Boolean值：如果第一个比第二个小就返回True。

在第四行代码里, less 被用作为一个函数: 它传入两个T类型的参数.

如上所写的是就函数类型, 或者还有命名参数, 如果你想文档化每个参数的含义。

Lambda Expressions

```
val sum = { x: Int, y: Int -> x + y }

val sum: (Int, Int) -> Int = { x, y -> x + y }

ints.filter { it > 0 }

fun(x: Int, y: Int): Int = x + y

fun(x: Int, y: Int): Int {
    return x + y
}

ints.filter(fun(item) = item > 0)

// receiver
val sum = fun Int.(other: Int): Int = this + other
sum : Int.(other: Int) -> Int
1.sum(2)
```

Lambda表达式

另外, 函数表达式也被称作 lambdas 或者 closures。这里有一个最简单的函数表达式: { it.toString() }。它是一段代码在“it”变量上调用了 two-string 函数。“it”是个 built-in 的名字。当你在写这些函数表达式的时候, 如果你只有一个参数传入这段代码, 你可以用“it”引用, 这只是一个你不需要定义参数的方法。

但是当你需要定义参数的时候, 或者不止一个参数要定义的时候, 语法就是这样的: { x, y -> x + y }。我们可以创建一段代码, 一个函数表达式, 输入两个参数, 然后把它们相加。如果我们愿意, 我们可以显示定义类型。

Lambda表达式语法

Lambda 表达式的全部语法形式，也就是函数类型的字面量，譬如上面的sum的声明。

一个 lambda 表达式或匿名函数是一个“函数字面值”，即一个未声明的函数，但却立即写为表达式。

这是非常常见的，一个 lambda 表达式只有一个参数。如果Kotlin能自己计算出自己的数字签名，我们就可以不去声明这个唯一的参数。并且用 it进行隐式声明。

请注意，如果函数取另一个函数作为最后一个参数，该 lambda 表达式参数可以放在括号外的参数列表。语法细则详见 callSuffix.

lambda 表达式

这里有更详细的描述，但是为了继续这一段，让我们看到一个简短的概述：

- 一个 lambda 表达式总是被大括号包围着。
- 其参数（如果说有的话）被声明在->之前（参数类型可以省略）
- 函数体在 -> 后面(如果存在的话).

匿名函数

上述 lambda 表达式的语法还少了一个东西：能够指定函数的返回类型。在大多数情况下，这是不必要的。因为返回类型可以被自动推断出来。然而，如果你需要明确的指定。你需要一个替代语法。匿名函数看起来很像一个普通函数声明，只是名字被省略了。

```
fun(x: Int, y: Int): Int = x + y
```

闭包

一个 lambda 表达式或者匿名函数（以及一个本地函数和一个对象表达式）可以访问他的闭包，即声明在外范围内的变量。与java不同，在闭包中捕获的变量可以被修改。

带接收者得函数字面值

这样的函数字面量的类型是一个带receiver的函数类型

kotlin提供了使用一个特定的 receiver对象 来调用一个函数的能力. 在函数体内部，你可以调用 接受者对象 的方法而不需要任何额外的限定符。这和 扩展函数 有点类似，它允你在函数体内访问接收器对象的成员。

Inline Functions

```
lock(l) { foo() }

// compiled
l.lock()
try {
    foo()
}
finally {
    l.unlock()
}

inline fun lock<T>(lock: Lock, body: () -> T): T {
    // ...
}

inline fun foo(inlined: () -> Unit,
    noinline notInlined: () -> Unit) {
    // ...
}
```

内联函数

内联函数和高阶函数经常一起见到。在某些场景下，当你用到泛型的时候，你可以给函数加上 inline 关键字。在编译时，它会用 lambda 表达式替换掉整个函数，整个函数的代码会成为内联代码。

使用高阶函数会带来一些运行时间效率的损失：每一个函数都是一个对象，并且都会捕获一个闭包。即那些在函数体内会被访问的变量。内存分配(对于函数对象和类)和虚拟调用会引入运行时间开销。

但是在许多情况下通过内联化 lambda 表达式可以消除这类的开销。我们通过下面的示例函数来分析上面这些内容。如，lock() 函数可以被很容易地在调用点被内联。

inline 修饰符会影响函数体本身以及传递过来的 lambdas: 所有的这些会被内联到 调用点。内联本身有时会引起生成的代码数量增加，但是如果我们使用得当(不要内联大的函数)。它将在 性能上有所提升，尤其是在超多态(megamorphic)调用点的循环中。

禁止内联 (**noinline**)

为了预防有时候你只希望被 (作为参数) 传递到一个内联函数的 lambdas 只有一些被内联，你可以用 noinline 修饰符标记你的参数

Extensions

```

fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}

val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'

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

fun Date.isTuesday(): Boolean {
    return getDay() == 2
}

val tuesday = date.isTuesday();

fun Int.biggerThanTen(): Boolean {
    return this > 10
}

```

声明一个扩展函数，我们需要用一个接收者类型也就是被扩展的类型来作为他的前缀。下面是为MutableList<Int>添加一个swap方法，这个this关键字在扩展方法内接受对应的对象（在点符号以前传过来的）现在，我们可以像一个其他方法一样调用MutableList<Int>。

函数扩展

Kotlin和c#、Gosu一样，能够扩展一个类的新功能，而无需继承类或使用任何类型的设计模式，如装饰者。这通过特殊的声明叫做extensions。Kotlin支持extension functions 和 extension properties.

函数扩展是 Kotlin 最强大的特性之一。

函数扩展可以是任何整形，字面量或者包装类型，也可以在标记为 final 的类上做类似操作。因为扩展函数不是真的给类增加代码，任何人都没有办法去修改一个类，它实际上是创建了一个静态方法，用语法糖来让扩展函数看着像是类自带的方法一样。

Kotlin 有扩展函数的概念。这不是 Kotlin 语言独有的，但是和其他语言里面我们看到的扩展又不太一样。如果我们在纯 Java 语言的环境下添加一个 date 的方法，我们需要写一个 utils 类或者 dates 类，然后增加一个静态方法。它接收一个实例，然后做些事情，可能会返回一个值。

Kotlin 的一个非常好的功能是，它会自动地转换有 getters 和 setters 综合属性的类型。所以我能够替换 getDay() 为 day，因为这个 day 的属性是存在的。它看起来像一个 field，但是实际上是个 property – getter 和 setter 的概念融合在了一起。

扩展的静态解析

扩展不能真正的修改他们继承的类。通过定义一个扩展，你不能在类内插入新成员，仅仅是通过该类的实例用点表达式去调用这个新函数。

我们想强调下扩展方法是被静态分发的，即他们不是接收类型的虚方法。

Nullable接收者

注意扩展可被定义为可空的接收类型。这样的扩展可以被对象变量调用，即使他的值是null，你可以在方法体内检查this == null，这也允许你在没有检查null的时候调用Kotlin中的toString()：检查发生在扩展方法的内部的时候

扩展属性

和方法相似，Kotlin支持扩展属性。注意：由于扩展没有实际的将成员插入类中，因此对扩展来说是无效的 属性是有backing field.这就是为什么初始化其不允许有 扩展属性。他们的行为只能显式的使用 getters/setters.

伴生对象的扩展

如果一个类定义有一个伴生对象，你也可以为伴生对象定义 扩展函数和属性，就像伴生对象的其他普通成员，只用用类名作为限定符去调用他们。

Stdlib

Kotlin标准库 The Kotlin Standard Library provides living essentials for everyday work with Kotlin. These include:

- 有用的高阶函数 Higher-order functions implementing idiomatic patterns (let, apply, use, synchronized, etc).
- 集合操作的扩展函数 Extension functions providing querying operations for collections (eager) and sequences (lazy).
- 字符串等工具函数 Various utilities for working with strings and char sequences.
- 对JDK文件/线程/IO等类的扩展 Extensions for JDK classes making it convenient to work with files, IO, and threading.

Types

Java	Kotlin
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.Void	kotlin.Nothing!
java.lang.CharSequence	kotlin(CharSequence)
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

Functions

```

fun assert(value: Boolean, lazyMessage: () -> Any)
fun check(value: Boolean, lazyMessage: () -> Any)
fun require(value: Boolean, lazyMessage: () -> Any)
fun error(message: Any): Nothing

fun <R> run(block: () -> R): R
fun <R> synchronized(lock: Any, block: () -> R): R
fun <T> lazy(initializer: () -> T): Lazy<T>
fun <T> lazyOf(value: T): Lazy<T>
fun <T, R> T.let(block: (T) -> R): R
fun <T> T.apply(block: T.() -> Unit): T
fun repeat(times: Int, action: (Int) -> Unit)
fun <T, R> with(receiver: T, block: T.() -> R): R

```

Functions

```

inline fun <T> T.apply(block: T.() -> Unit)
    : T { block(); return this }

inline fun <T, R> T.let(block: (T) -> R): R = block(this)

inline fun <T, R> with(receiver: T, block: T.() -> R)
    : R = receiver.block()

var u1: User? = null
val u2 = User("Alice", age = 20, desc = "Wonderful!")
u1?.apply { sayHello() }
u2.apply { eat(); sayHello() }
u1?.let {
    u2.show(it)
    u1.sayHello()
}
u2.let { println("user is valid!") }
with(u2) {
    println("do something on user")
    sayHello()
}

```

T.apply

Calls the specified function block with this value as its receiver and returns this value.

T.let

Calls the specified function block with this value as its argument and returns its result.

with

Calls the specified function block with the given receiver as its receiver and returns its result.

Arrays

```
class Array<T> private constructor() {
    val size: Int
    fun get(index: Int): T
    fun set(index: Int, value: T): Unit

    fun iterator(): Iterator<T>
    // ...
}

val asc = Array(5, { i -> (i * i).toString() })

val x: IntArray = arrayOf(1, 2, 3, 4, 5)
x[0] = x[1] + x[2]

val intArray2 = intArrayOf(2, 4, 6, 8, 10)
val boolArray1 = arrayOf(true, false, true, false, true)
val boolArray2 = booleanArrayOf(true, true, false, true)
val strArray1 = arrayOf("Cat", "Dog", "Rabbit")

// byteArrayOf(), charArrayOf(), shortArrayOf(),
// longArrayOf(), floatArrayOf()
```

Array Extensions

```
// create array  
arrayOf()/arrayOfNulls()/emptyArray()/intArrayOf()  
  
// functions  
- get()/set()/iterator()/indices/lastIndex  
  
// extension functions  
- asIterable()/asList()/asSequence()/associate()  
- distinct()/distinctBy()/drop()/dropLast()/binarySearch()  
  
- find()/findLast()  
- fill()/contains()/copyOf()/count()  
  
- elementAt()/elementOrNull()/first()/last()  
- all()/any()/filter()/filterNot()/filterTo()/flatten()  
  
- forEach()/forEachIndexed()/map()/mapIndexed()/groupBy()  
- intersect()()/joinTo()()/joinToString()
```

Collections

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
println(numbers)           // prints "[1, 2, 3]"
numbers.add(4)

val readOnlyView: List<Int> = numbers
readOnlyView.clear()      // -> does not compile

val strings = hashSetOf("a", "b", "c", "c")

val items = listOf(1, 2, 3, 4)
items.last == 4
items.filter { it % 2 == 0 } // Returns [2, 4]

if (rwList.none { it > 6 }) println("No items above 6")
val item = rwList.firstOrNull()

val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(map["foo"])
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

Ranges

```
if (i in 1..10) { // equivalent of 1 <= i && i <= 10
    println(i)
}

for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing

for (i in 4 downTo 1) print(i) // prints "4321"

for (i in 1..4 step 2) print(i) // prints "13"

for (i in 4 downTo 1 step 2) print(i) // prints "42"

// progression with values [1, 3, 5, 7, 9, 11]
(1..12 step 2).last == 11

// progression with values [1, 4, 7, 10]
(1..12 step 3).last == 10

// progression with values [1, 5, 9]
(1..12 step 4).last == 9
```

Collection Extensions

```
// Iterable, Collection, List, Set, Map

- iterator() // Iterable
- size/indices/count()/isEmpty()/contains() // Collection
- lastIndex/get()/indexOf()/listIterator()/subList() // List

// Extensions Functions for List
- toTypedArray()/toMutableList()
- isEmpty()/?orEmpty()/plus()/plusElement()
- asReversed()/binarySearch()/findLast()

- dropLast()/takeLast()/reduceRight()
- getOrNull()/elementAt()/elementAtOrNull()
- single()/slice()/first()/firstOrNull()/last()
- +/plus()

// MutableIterable, MutableCollection, MutableList

- add()/addAll()/remove()/removeAt()
- removeAll()/retainAll()/clear()
- reverse()/sort()/sortBy()/sortByDescending()/sortWith()
- +=/plusAssign() -=/minusAssign()
```

Generics

- Java's wildcards are converted into type projections
 - Foo<? extends Bar> becomes Foo<out Bar!>!
 - Foo<? super Bar> becomes Foo<in Bar!>!
- Java's raw types are converted into star projections
 - List becomes List<*>!, i.e. List<out Any?>!

```
// Array<out Any> -> Java: Array<? extends Object>
fun copy2(from: Array<out Any>, to: Array<Any>) {
    // ...
}

// Array<in String> -> Java: Array<? super String>
fun fill(dest: Array<in String>, value: String) {
    // ...
}

fun <T> T.basicToString(): String { // extension function
    // ...
}
```

Kotlin的泛型

- 与Java相似，Kotlin中的类也具有类型参数，一般而言，创建类的实例时，我们需要声明参数的类型，但当参数类型可以从构造函数参数等途径推测时，在创建的过程中可以忽略类型参数：

```
val box = Box(1) // 1 has type Int, so the compiler figures out
that we are talking about Box<Int>
```

- 首先，我们考虑一下Java中的通配符（wildcards）的意义。该问题在文档 Effective Java, Item 28: Use bounded wildcards to increase API flexibility 中给出了详细的解释。首先，Java中的泛型类型是不变的，即List<String>并不是List<Object>的子类型。原因在于，如果List是可变的，并不会优于Java数组。
- 通配符类型（wildcard）的声明 ? extends T 表明了该方法允许一类对象是 T 的子类型，而非必须得是 T 本身。这意味着我们可以安全地从元素（T 的子类集合中的元素）读取 T，同时由于我们并不知道 T 的子类型，所以不能写元素。反过来，该限制可以让Collection<String>表示为Collection<? extends Object>的子类型。简而言之，带extends限定（上限）的通配符类型（wildcard）使得类型是协变的（covariant）。

- `out`修饰符叫做型变注解，同时由于它在参数类型位置被提供，所以我们讨论声明处型变。与Java的使用处型变相反，类型使用通配符使得类型协变。另外除了`out`，Kotlin又补充了一项型变注释：`in`。它是的变量类型反变：只可以被消费而不可以被生产。反变类的一个很好的例子是`Comparable`

Kotlin 中的 Java 泛型

Kotlin的泛型和Java的有些不同（详见 Generics）。当引入java类型的时候，我们作如下转换：

- Java的通配符转换成类型投射
 - `Foo<? extends Bar>` 转换成 `Foo<out Bar!>!`
 - `Foo<? super Bar>` 转换成 `Foo<in Bar!>!`
- Java的原始类型转换成星号投射
 - `List` 转换成 `List<*>!`, 也就是 `List<out Any?>!`

和Java一样，Kotlin在运行时不保留泛型，即对象不知道传递到他们构造器中的那些参数的实际类型。

Kotlin的范型就像Java一样不会在运行时保留信息，也就是对象不会携带传递到它们构造函数中的类型参数的信息。也就是说，运行时无法区分`ArrayList<Integer>()` 和 `ArrayList<Character>()`.

也就是说，`ArrayList<Integer>()` 和 `ArrayList<Character>()` 是区分不出来的。这意味着，不可能用`is-`来检测泛型。

这就导致，无法使用`is-`检测范型。Kotlin只允许用`is-`来检测星号投射的泛型类型：Kotlin只允许用`is-`检测星号投射的范型类型。

Others

Smart Cast

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}

if (x !is String) return
print(x.length) // x is automatically cast to String

// x is automatically cast to string
// on the right-hand side of `||`
if (x !is String || x.length == 0) return

// x is automatically cast to string
// on the right-hand side of `&&`
if (x is String && x.length > 0)
    // x is automatically cast to String
    print(x.length)
```

is 和 !is 运算符

我们可以使用is 或者它的否定!is运算符检查一个对象在运行中是否符合所给出的类型

智能转换

在很多情况下，在Kotlin有时不用使用明确的转换运算符，因为编译器会在需要的时候自动为了不变的值和输入（安全）而使用is进行监测。这些智能转换在when-expressions 和 while-loops 也一样

Type Cast

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

```
val x: String = y as String

val x: String? = y as String?

val x: String? = y as? String
```

“不安全”的转换运算符

通常，如果转换是不可能的，转换运算符会抛出一个异常。于是，我们称之为不安全的。在Kotlin这种不安全的转换会出现在插入运算符as (see operator precedence)：

```
val x: String = y as String
```

记住null不能被转换为不可为空的String。例如，如果y是空，则这段代码会抛出异常。为了匹配Java的转换语义，我们不得不在右边拥有可空的类型，就像：

```
val x: String? = y as String?
```

“安全的”（可为空的）转换运算符

为了避免异常的抛出，一个可以使用安全的转换运算符——as?，它可以在失败时返回一个null：

```
val x: String? = y as? String
```

记住尽管事实是右边的as?可使一个不为空的String类型的转换结果为可空的。

This

```

class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this

            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver
            }
            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}

```

This的语义

为了记录下当前的接受者我们使用this表达式:

- 在一个类成员中, this指的是当前类对象。
- 在一个扩展函数或者带有接收者字面函数, this表示左边的接收者.

如果 this 没有应用者, 则指向的是最内层的闭合范围。为了在其它范围内返回 this , 需要使用标签。

为了在范围外部访问this(一个类, 或者扩展函数, 或者带标签的带接收者的字面函数 我们使用this@label作为label

Null Safety

```

var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok

val l = a.length
val l = b.length // error: variable 'b' can be null

val l = if (b != null) b.length else -1

val l: Int = if (b != null) b.length else -1
val l = b?.length ?: -1
val l = b{!!}.length() // npe
val aInt: Int? = a as? Int

val files = File("Test").listFiles()
println(files?.size)
println(files?.size ?: "empty")

```

可空（**Nullable**）和不可空（**Non-Null**）类型

在 Kotlin 的类型体系里，有空类型和非空类型。类型系统识别出了 string 是一个非空类型，并且阻止编译器让它以空的状态存在。想要让一个变量为空，我们需要在声明后面加一个？号，同时赋值为 null。

Kotlin 的类型系统致力于消除空引用异常的危险，又称《上亿美元的错误》。

许多编程语言，包括 Java 中最常见的错误就是访问空引用的成员变量，导致空引用异常。在 Java 中，将等同于 NullPointerException 或简称 NPE。

Kotlin 类型系统的目的就是从我们的代码中消除 NullPointerException。NPE 的原因可能是

- 显式调用 throw NullPointerException()
- Usage of the !! operator that is described below
- 外部 Java 代码引起
- 对于初始化，有一些数据不一致（比如一个还没初始化的 this 用于构造函数的某个地方）

在 Kotlin 中，类型系统是要区分一个引用是否可以是 null（nullable references）或者不可以，即不可空引用（non-null references）。例如，常见的 String 就不能够为 null，若是想要允许 null，我们可以声明一个变量为可空字符串，写作 String?。

安全的调用

你的第二个选择是安全的操作符，写作 ?. : b?.length

如果 b 是非空的，就会返回 b.length，否则返回 null，这个表达式的类型就是 Int?.

安全调用在链式调用的时候十分有用。例如，如果 Bob，一个雇员，可被分配给一个部门（或不），这反过来又可以获得 Bob 的部门负责人的名字（如果有的话），我们这么写：

```
bob?.department?.head?.name
```

如果任意一个属性（环节）为空，这个链式调用就会返回 null。

Elvis 操作符

当我们有一个可以为空的变量 r，我们可以说「如果 r 非空，我们使用它；否则使用某个非空的值：

```
val l: Int = if (b != null) b.length else -1
```

对于完整的 if-表达式，可以换成 Elvis 操作符来表达，写作 ?::

```
val l = b?.length ?: -1
```

如果 ?: 的左边表达式是非空的，elvis 操作符就会返回左边的结果，否则返回右边的内容。请注意，仅在左侧为空的时候，右侧表达式才会进行计算。

注意，因为 throw 和 return 在 Kotlin 中都是一种表达式，它们也可以用在 Elvis 操作符的右边。非常方便，例如，检查函数参数

!! 操作符

第三种操作的方式是给 NPE 爱好者的。我们可以写 b!!，这样就会返回一个不可空的 b 的值（例如：在我们例子中的 String）或者如果 b 是空的，就会抛出 NPE 异常：

```
val l = b!!.length()
```

因此，如果你想要一个 NPE，你可以使用它。

安全转型

转型的时候，可能会经常出现 ClassCastException。所以，现在可以使用安全转型，当转型不成功的时候，它会返回 null：

```
val aInt: Int? = a as? Int
```

Exceptions

```
try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}

// try is an expression
val a: Int? = try { parseInt(input) }
    catch (e: NumberFormatException) { null }
```

Reference

```

val c = MyClass::class

fun isOdd(x: Int) = x % 2 != 0

val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // prints [1, 3]

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

fun length(s: String) = s.size

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // Prints "[a, abc]"

```

类引用

最基本的反射特性就是得到运行时的类引用。要获取引用并使之成为静态类可以使用字面类语法:

val c = MyClass::class 引用是KClass类型.你可以使用KClass.properties 和 KClass.extensionProperties来获得类和父类所有属性引用的列表。

注意Kotlin类引用不完全与Java类引用一致.查看Java interop section 详细信息。

函数引用

我们有一个像下面这样的函数声明:

fun isOdd(x: Int) = x % 2 != 0 我们可以直接调用(isOdd(5)),也可以把它作为一个值传给其他函数.我们使用::操作符实现:

val numbers = listOf(1, 2, 3) println(numbers.filter(::isOdd)) // prints [1, 3] 这里 ::isOdd 是一个函数类型的值 (Int) -> Boolean.

注意现在`::`不能被使用来重载函数。将来，我们计划提供一个语法明确参数类型这样就可以使用明确的重载函数了。

如果我们需要使用类成员或者一个扩展方法，它必须是有权访问的，例如`String::toCharArray`带着一个`String: String.() -> CharArray`类型扩展函数。

属性引用

我们同样可以用`::`操作符来访问Kotlin中的顶级类的属性：

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // prints "1"
    ::x.set(2)
    println(x)         // prints "2"
}
```

表达式`::x`推断为`KProperty`类型的属性对象，它允许我们使用`get()`函数来读它的值或者使用`name`属性来得到它的值。

构造函数引用

构造函数可以像属性和方法那样引用。它们可以使用在任何一个函数类型的对象的地方，期望得到相同参数的构造函数，并返回一个适当类型的对象。构造函数使用`::`操作符加类名引用。考虑如下函数，需要一个无参数函数返回类型是`Foo`

Java Interop

Calling Java from Kotlin

```

import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source)
        list.add(item)
    // Operator conventions work as well:
    for (i in 0..source.size() - 1)
        list[i] = source[i] // get and set are called
}

val calendar = Calendar.getInstance()
if (calendar.firstDayOfWeek == Calendar.SUNDAY) {
    calendar.firstDayOfWeek = Calendar.MONDAY
}

```

- 如果一个Java方法返回void，那么在Kotlin中，它会返回Unit。万一有人使用它的返回值，Kotlin的编译器会在调用的地方赋值，因为这个值本身已经提前可以预知了(这个值就是Unit)。

Null安全性和平台类型

Java中的所有引用都可能是null值，这使得Kotlin严格的null控制对来自Java的对象来说变得不切实际。在Kotlin中Java声明类型被特别对待叫做platform types.这种类型的Null检查是不严格的，所以他们还维持着同Java中一样的安全性(更多参见下面)。

平台类型的概念

如上所述，平台类型不能再程序里显式的出现，所以没有针对他们的语法。然而，编译器和IDE有时需要显式他们(如在错误信息，参数信息中)，所以我们用一个好记的标记来表示他们：

- T! 表示 “T 或者 T?”
- (Mutable)Collection! 表示 “T的java集合，可变的或不可变的，可空的或非空的”
- Array<(out) T>! 表示 “T(或T的子类)的java数组，可空的或非空的”

映射类型

Kotlin特殊处理一部分java类型。这些类型不是通过as或is来直接转换，而是映射到了指定的kotlin类型上。映射只发生在编译期间，运行时仍然是原来的类型。java的原生类型映射成如下kotlin类型（记得平台类型）。

Java数组

和Java不同，Kotlin里的数组不是协变的。Kotlin不允许我们把Array<String>赋值给Array<Any>，从而避免了可能的运行时错误。Kotlin也禁止我们把一个子类的数组当做父类的数组传递进Kotlin的方法里。但是对Java方法，这是允许的（考虑这种形式的平台类型platform types Array<(out) String>!）。

Java平台上，原生数据类型的数组被用来避免封箱/开箱的操作开销。由于Kotlin隐藏了这些实现细节，就得有一个变通方法和Java代码交互。每个原生类型的数组都有一个特有类(specialized class)来处理这种问题(IntArray, DoubleArray, CharArray ...)。它们不是Array类，而是被编译成java的原生数组，来获得最好的性能。

Calling Java from Kotlin

```

val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)

val fooclass = foo.javaClass // foo.getClass()
val fooclass = javaClass<Foo>() // Foo.class

if (Character.isLetter(a)) {
    // ...
}

// SAM
val runnable = Runnable { println("This runs in a runnable") }
val executor = ThreadPoolExecutor()
// void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }

```

Java Varargs

Java类也会这样声明方法，表示参数是可变参数。这种情况，你需要用展开操作符 * 来传递 IntArray，目前无法传递 null 给一个变参的方法。

对象方法

当java类型被引入到kotlin里时，所有的java.lang.Object类型引用，会被转换成 Any。因为Any不是平台独有的，它仅声明了三个成员方法：toString(), hashCode() 和 equals()，所以为了能用到java.lang.Object的其他方法，kotlin采用了扩展函数。

Java 反射

Java反射可以用在kotlin类上，反之亦然。前面提过，你可以 instance.javaClass 或者 ClassName::class.java 开始基于 java.lang.Class 的java反射操作。

java类的继承

在kotlin里，超类里最多只能有一个java类(java接口数目不限)。这个java类必须放在超类列表的最前面。

访问静态成员

java类的静态成员就是它们的“同伴对象”。我们无法将这样的“同伴对象”当作数值来传递，但可以显式的访问它们，比如：

SAM(单抽象方法) 转换

就像 Java 8 那样，Kotlin 支持 SAM 转换，这意味着 Kotlin 函数字面量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跟这个 Kotlin 函数的参数类型匹配的上。

如果 Java 类有多个接受函数接口的方法，你可以用一个适配函数来把闭包转成你需要的 SAM 类型。编译器也会在必要时生成这些适配函数。

注意SAM的转换只对接口有效，对抽象类无效，即使它们就只有一个抽象方法。

还要注意这个特性只针对和 Java 的互操作；因为 Kotlin 有合适的函数类型，把函数自动转换成 Kotlin 接口的实现是没有必要的，也就没有支持了。

Calling Kotlin from Java

```
@file:JvmName("DemoUtils")
package demo
class Foo

fun bar() {
}

class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

可以使用 `@JvmName` 注解自定义生成的Java类的类名

属性

属性getters被转换成get-方法，setters转换成set-方法。

包级别的函数

example.kt 文件中 org.foo.bar 包内声明的所有的函数和属性，都会被放到一个叫 org.foo.bar.ExampleKt的java类里。

类名

如果多个文件中生成了相同的Java类名（包名相同，类名相同或者有相同的@JvmName注解）通常会报错，然而，可以在每个文件添加 @JvmMultifileClass注解，可以让编译器生成一个统一的带有特殊名字的类，这个类包含了对应这些文件中所有的声明。

实例字段

如果在 Java 需要像字段一样调用一个 Kotlin 的属性，你需要使用@JvmField注解。这个字段与属性具有相同的可见性。属性符合有实际字段(backing field)、非私有、没有open, override 或者 const修饰符、不是被委托的属性这些条件才可以使用@JvmField注解。

Calling Kotlin from Java

```
class C {
    companion object {
        const val VERSION = 1

        @JvmStatic fun foo() {}
        fun bar() {}

        @JvmField val COMPARATOR: Comparator<Key>
            = compareBy<Key> { it.value }
    }
}
```

```
C.foo(); // works fine
C.bar(); // error: not a static method
C.COMPARATOR.compare(key1, key2);
int v = C.VERSION;
```

静态字段

在一个命名对象或者伴生对象中声明的Koltin属性会持有静态实际字段(backing fields)，这些字段存在于该命名对象或者伴生对象中的。

通常，这些字段都是private的，但是他们可以通过以下方式暴露出来。

- @JvmField 注解;
- lateinit 修饰符;
- const 修饰符.
- 用 @JvmField 注解该属性可以生成一个与该属性相同可见性的静态字段。

使用const注解可以将 Kotlin 属性转换成 Java 中的静态字段。

静态方法

正如上面所说，Kotlin 自动为包级函数生成了静态方法 在Kotlin 中，还可以通过 @JvmStatic注解在命名对象或者伴生对象中定义的函数来生成对应的静态方法。

通过使用 @JvmStatic 注解对象的属性或伴生对象，使对应的getter 和 setter 方法在这个对象或者包含这个伴生对象的类中也成为静态成员。

Calling Kotlin from Java

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>

@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}

@Throws(IOException::class)
fun foo() { throw IOException() }
```

```
// Java
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

用 @JvmName 解决签名冲突

有时我们想让一个 Kotlin 里的命名函数在字节码里有另外一个 JVM 名字。最突出的例子就是 类名型擦除：

```
fun List<String>.filterValid(): List<String> fun List<Int>.filterValid(): List<Int>
```

这两个函数不能同时定义，因为它们的 JVM 签名是一样的：

filterValid(Ljava/util/List;)Ljava/util/List;. 如果我们真的想让它们在 Kotlin 里用同一个名字，我们需要用 @JvmName 去注释它们中的一个（或两个），指定的另外一个名字当参数

Kotlin 里它们可以都用 filterValid 来访问，但是在 Java 里，它们是 filterValid 和 filterValidInt. 同样的技巧也适用于属性 x 和函数 getX() 共存

生成重载

通常，如果你写一个有默认参数值的 Kotlin 方法，在 Java 里，只会有一个有完整参数的签名。如果你要暴露多个重载给 Java 调用者，你可以使用 @JvmOverloads 注解。

构造函数，静态函数等也能用这个标记。但他不能用在抽象方法上，包括 接口中的方法。

注意一下，Secondary Constructors 描述过，如果一个类的所有构造函数参数都有默认值，会生成一个公开的无参构造函数。这就算没有 @JvmOverloads 注解也有效。

Null 安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们传递 null 给一个非空参数。这就是为什么 Kotlin 给所有期望非空参数的公开函数生成运行时检测。这样我们就能在 Java 代码里立即得到 NullPointerException。

Nothing 类型的转换

Nothing 是一种特殊的类型，因为它在 Java 中没有类型相对应。事实上，每个 Java 的引用类型，包括 `java.lang.Void` 都可以接受 null 值，但是 Nothing 不行，因此在 Java 世界中没有什么可以代表这个类型，这就是为什么在 Kotlin 中要生成原始类型需要使用 Nothing。

面向对象

- 面向对象
- 继承
- Kotlin如何优雅的实现多继承
- 类成员的可见性
- 接口和抽象类
- 伴生对象和静态成员
- 属性代理
- 单例
- object单例
- 密封类
- dataclass
- 为什么不直接使用 `ArrayInt` 而是 `IntArray`?
- Kotlin 遇到 MyBatis：到底是 Int 的错，还是 data class 的错？

Kotlin 的类特性(上)

前面三章的内容是写给希望快速了解 Kotlin 语言的大忙人的。

而从本章开始，才会真正讲述 Kotlin 语言的神奇之处。

与 Java 相同，Kotlin 声明类的关键字是 `class`。类声明由类名、类头和类体构成。

其中 `类头` 和 `类体` 都是可选的；如果一个类没有类体，那么花括号也是可以省略的。

1. 构造函数

Kotlin 的构造函数可以写在类头中，跟在类名后面，如果有注解还需要加上关键字 `constructor`。这种写法声明的构造函数，我们称之为 `主构造函数`。例如下面我们将 `Person` 创建带一个 `String` 类型参数的构造函数。

```
class Person(private val name: String) {
    fun sayHello() {
        println("hello $name")
    }
}
```

在构造函数中声明的参数，它们默认属于类的公有字段，可以直接使用，如果你不希望别的类访问到这个变量，可以用 `private` 修饰。

在主构造函数中不能有任何代码实现，如果有额外的代码需要在构造方法中执行，你需要放到 `init` 代码块中执行。同时，在本示例中由于需要更改 `name` 参数的值，我们将 `val` 改为 `var`，表明 `name` 参数是一个可改变的参数。

```
class Person(private var name: String) {  
  
    init {  
        name = "Zhang Tao"  
    }  
  
    internal fun sayHello() {  
        println("hello $name")  
    }  
}
```

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。如果你不希望你的类有一个公有构造函数,你需要声明一个带有非默认可见性的空的主构造函数。

另外，在 JVM 上,如果主构造函数的所有参数都有默认值，编译器会生成一个额外的无参构造函数,它将使用默认值。

2. 次级构造函数

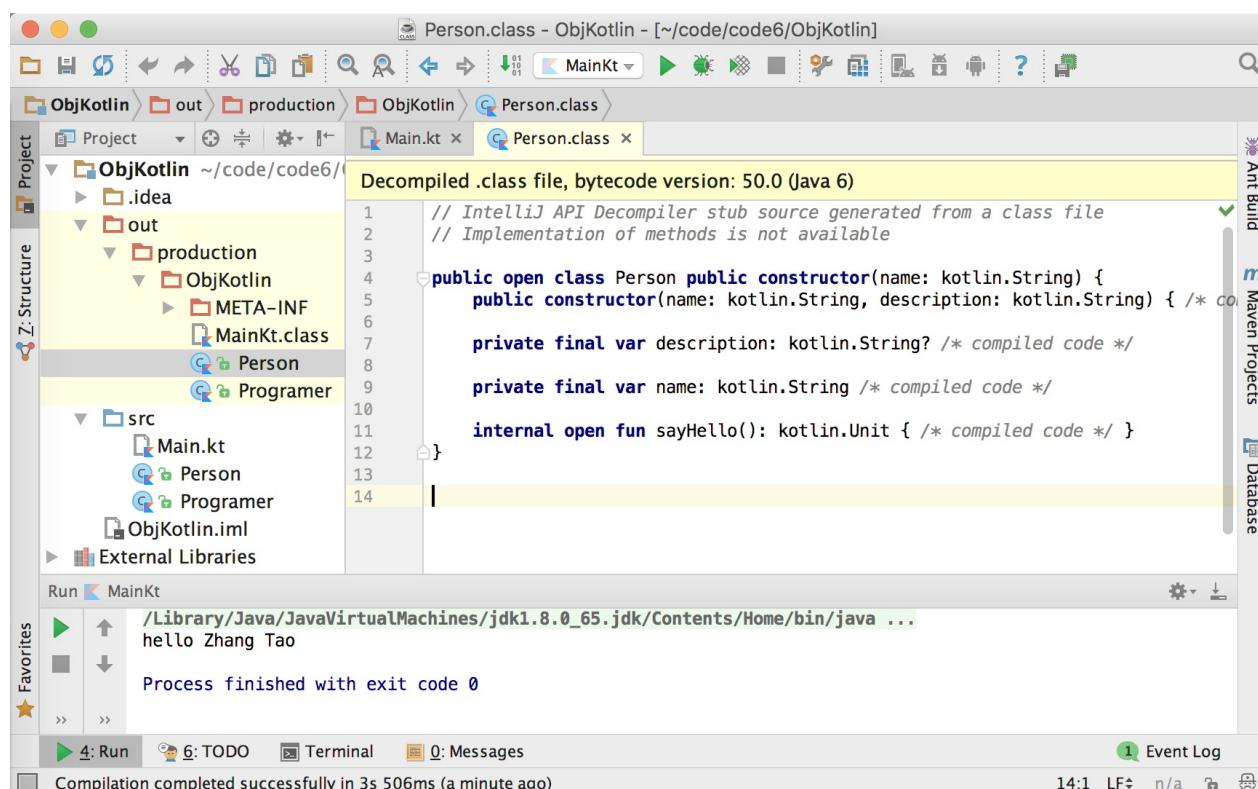
一个类当然会有多个构造函数的可能，只有主构造函数可以写在类头中，其他的次级构造函数(Secondary Constructors)就需要写在类体中了。

```
class Person(private var name: String) {  
  
    private var description: String? = null  
  
    init {  
        name = "Zhang Tao"  
    }  
  
    constructor(name: String, description: String) : this(name)  
    {  
        this.description = description  
    }  
  
    internal fun sayHello() {  
        println("hello $name")  
    }  
}
```

这里我们让次级构造函数调用了主构造函数，完成 name 的赋值。由于次级构造函数不能直接将参数转换为字段，所以需要手动声明一个 description 字段，并为 description 字段赋值。

3. 修饰符

点开 IDEA，工程目录中的 out 列表，看到我们写完的 Person 被编译为 class 文件后的样子。



3.1 open 修饰符

Kotlin 默认会为每个变量和方法添加 final 修饰符。这么做的目的是为了程序运行的性能，其实在 Java 程序中，你也应该尽可能为每个类添加final 修饰符(见 Effective Java 第四章 17 条)。

为每个类加了 final 也就是说，在 Kotlin 中默认每个类都是不可被继承的。如果你确定这个类是会被继承的，那么你需要给这个类添加 open 修饰符。

3.2 internal 修饰符

写过 Java 的同学一定知道，Java 有三种访问修饰符，public/private/protected，还有一个默认的包级别访问权限没有修饰符。

在 Kotlin 中，默认的访问权限是 public，也就是说不加访问权限修饰符的就是 public 的。而多增加了一种访问修饰符叫 internal 。它是模块级别的访问权限。

何为模块(module)，我们称被一起编译的一系列 Kotlin 文件为一个模块。在 IDEA 中可以很明确的看到一个 module 就是一个模块，当跨 module 的时候就无法访问另一个 module 的 internal 变量或方法。

4. 一些特殊的类

4.1 枚举类

在 Kotlin 中，每个枚举常量都是一个对象。枚举常量用逗号分隔。例如我们写一个枚举类 Programer。

```
enum class Programer {  
    JAVA, KOTLIN, C, CPP, ANDROID;  
}
```

当它被编译成 class 后，将转为如下代码实际就是一个私有了构造函数的 `kotlin.Enum` 继承类。

```
public final enum class Programer  
private constructor() : kotlin.Enum<Programer> {  
    JAVA, KOTLIN, C, CPP, ANDROID;  
}
```

接着我们再来看 `kotlin.Enum` 这个类(节选)

```

public abstract class Enum<E : Enum<E>>
(name: String, ordinal: Int): Comparable<E> {
    companion object {}

    /**
     * Returns the name of this enum constant,
     * exactly as declared in its enum declaration.
     */
    public final val name: String

    /**
     * Returns the ordinal of this enumeration
     * constant (its position in its enum
     * declaration, where the initial constant
     * is assigned an ordinal of zero).
     */
    public final val ordinal: Int

    public override final fun compareTo(other: E): Int
}

```

发现，其实在 Kotlin 中，枚举的本质是一个实现了 `Comparable` 的 class，其排序就是按照字段在枚举类中定义的顺序来的。

4.2 sealed 密封类

`sealed` 修饰的类称为密封类，用来表示受限的类层次结构。例如当一个值为有限集中的类型、而不能有任何其他类型时。在某种意义上，他们是枚举类的扩展：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。

4.3 data 数据类

`data` 修饰的类称之为数据类。它通常用在我们写的一些 POJO 类上。

当 `data` 修饰后，会自动将所有成员用 `operator` 声明，即为这些成员生成类似 Java 的 `getter/setter` 方法。

本章就先介绍到这，下一章我们讲继承与组合，伪多继承与接口等内容。

Kotlin 的类特性(下)

前面三章的内容是写给希望快速了解 Kotlin 语言的大忙人的。而从本章开始，才会真正讲述 Kotlin 语言的神奇之处。

1. 类的扩展

在 Java 开发的时候，经常会写一大堆的 Utils 类，甚至经常写一些 common 包，比如著名的 `apache-commons` 系列、`Guava` 等等。

如果每个类在想要用这些工具类的时候，他们自己就已经具备了这些工具方法多好，`Kotlin` 的类扩展方法就是这个作用。

1.1 扩展方法

在之前的文章中我就讲过扩展方法了，这里就不再多赘述，只回顾一下扩展方法的格式：

```
fun Activity.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, duration).show()
}
```

首先是一个 `fun` 关键字，紧接着是要扩展哪个类的类名，点方法名，然后是方法的声明和返回值以及方法体。

1.2 小心有坑

需要注意的是扩展方法是静态解析的，而并不是真正给类添加了这个方法。

举个例子：

```
open class Animal{  
  
}  
class Dog : Animal()  
  
object Main {  
    fun Animal.bark() = "animal"  
  
    fun Dog.bark() = "dog"  
  
    fun Animal.printBark(anim: Animal){  
        println(anim.bark())  
    }  
  
    @JvmStatic fun main(args: Array<String>) {  
        Animal().printBark(Dog())  
    }  
}
```

最终的输出是 `animal`，而不是 `dog`。

因为扩展方法是静态解析的，在添加扩展方法的时候类型为 `Animal`，那么即便运行时传入了子类对象，也依旧会执行参数中声明时类型的方法。

1.3 强转与智能转换

在 `Kotlin` 中，用 `is` 来判断一个对象是否是某个类的实例，用 `as` 来做强转。

`Kotlin` 有一个很好的特性，叫 `智能转换(smart cast)`，在我之前的文章中也提到过。就是当已经确定一个对象的类型后，可以自动识别为这个类的对象，而不用再手动强转。

```

fun main(args: Array<String>) {
    var animal: Animal? = xxx
    if (animal is Dog) {
        //在这里 animal 被当做 Dog 的对象来处理
        animal.bark()
    }
}

```

1.4 总有例外

如果智能转换的对象是一个全局变量，这个变量可能在别的地方被改变赋值，所以你必须手动判断与转换它的类型。

```

open class Animal {

}

class Dog : Animal() {
    fun bark() {
        println("animal")
    }
}

var animal: Animal? = null

fun main(args: Array<String>) {
    if (animal is Dog) {
        //在这里你必须手动强转为Dog的对象
        (animal as Dog).bark()
    }
}

```

2. 伴生对象

在上一篇 **Kotlin 与 Java 互转** 中 我们提到这样一段工具类代码

```

class StringUtils {
    companion object {
        fun isEmpty(str: String): Boolean {
            return "" == str
        }
    }
}

```

由于 Kotlin 没有静态方法。在大多数情况下，官方建议是简单地使用包级函数。如果你需要写一个可以无需用一个类的实例来调用、但需要访问类内部的函数（例如，工厂方法或单利），你可以把它写成一个用 `companion` 修饰的对象内的方法。我们称 `companion` 修饰的对象为伴生对象。

将上面的代码编译后查看，实际上是编译器生成了一个 `public` 的内部对象。

```

public final class StringUtils public constructor() {
    public companion object {
        public final fun isEmpty(str: kotlin.String): kotlin.Bool
lean {
        /* compiled code */
    }
}

```

3. 单例类设计

伴生对象更多的用途是用来创建一个单例类。如果只是简单的写，直接用伴生对象返回一个 `val` 修饰的外部类对象就可以了，但是更多的时候我们希望在类被调用的时候才去初始化他的对象。以下代码将线程安全问题交给虚拟机在静态内部类加载时处理，是一种推荐的写法：

```
class Single private constructor() {  
    companion object {  
        fun get():Single{  
            return Holder.instance  
        }  
    }  
  
    private object Holder {  
        val instance = Single()  
    }  
}
```

4. 动态代理

写多继承还是要根据场景来，正好今天跟朋友聊到他们项目重构的问题，我当时就说了一句：果然还是 Kotlin 好，原生支持动态代理。

朋友的一个 Android 项目，所有网络请求包括回调和参数全部封装在了一个 BaseActivity 中，然后随着项目越来越大，这一些网络请求方法想要抽出来，但又害怕牵连到线上的改动，我就推荐他用个动态代理来做，但是 Java 的动态代理又得要反射，又得要额外多写很多的代码方法，又是一个大改动。

反之看 Kotlin 的动态代理：

```

interface Animal{
    fun bark()
}

class Dog :Animal {
    override fun bark() {
        println("Wang Wang")
    }
}

class Cat(animal: Animal) : Animal by animal {}

fun main(args: Array<String>) {
    Cat(Dog()).bark()
}

```

这样，我们就很成功的让一只猫的叫声用狗去代理掉了，于是上面的 `main` 方法执行完后就变成了 Wang Wang。

5. 伪多继承

Kotlin 的动态代理更多的是用在一种需要多继承的场景。

例如，还是之前我举的我朋友那个项目的例子，他们的问题在于，每个 `BaseActivity` 的子类，都会要请求不同的网络，可能A需要获取用户信息，B需要获取活动列表，C既需要活动列表也需要获取用户信息，D却只需要获取图片列表。

这样一个场景，使用一个代理类实现所有需要获取信息的接口方法。然后让不同的子类去实现所需的接口，请求统一交给代理类完成。这样不仅维护网络请求信息方便，而且每个类不会有额外多出来的方法防止新人接触项目的时候调用错请求方法。

还是用猫狗来举例：

```
interface Animal{
    fun bark()
}

interface Food{
    fun eat()
}

class Delegate : Animal, Food {
    override fun eat() {
        println("mouse")
    }

    override fun bark() {
        println("Miao")
    }
}

class Cat(animal: Animal, food: Food) : Animal by animal, Food by food {
}

@JvmStatic fun main(args: Array<String>) {
    val delegate: Delegate = Delegate()
    Cat(delegate, delegate).bark()
}
```

Kotlin 的类就介绍这么多，下一章我们讲：闭包

面向对象

- Any--Java 中的 Object
- open，默认的类和方法都是 final 的，想要继承需要 open
- 创建对象 classname()，不用 new 关键字
- override 不可省略
- data class 类似于 C 中的 结构体
- is Java 中的 instanceof
- as as? 安全的类型转换，转换失败时不抛异常
- 空类型安全 ?: !!
- package 包，类似于 C++ 中的 namespace
- lateinit 延迟初始化属性
- by lazy
- 运算符重载 infix 中缀表达式
- 接口中包含抽象方法和方法实现，冲突 super<Parent>.fun
- 接口代理 by
- 扩展方法 classname.fun
- 伴生对象 companion object @JvmStatic 静态方法/类方法
- 可见性：internal 模块内可见
- 数据类，再见 javabean，final class，没有默认的无参构造方法，allopen 和 noarg 插件，编译器会自动生成常用方法 toString(), hashCode()...
- sealed class 密封类，子类有限的类，子类只能定义在同一个文件内
- 枚举，实例可数
- 继承冲突 super<A>.x()
- T::class.java

类成员

lateinit 延迟初始化属性

属性代理 by lazy{}

by delegator

- LazyThreadSafetyMode.SYNCHRONIZED
- LazyThreadSafetyMode.PUBLICATION
- LazyThreadSafetyMode.NONE

继承冲突

```
abstract class A{
    open fun x(): Int = 5
}

interface B{
    fun x(): Int = 1
}

interface C{
    fun x(): Int = 0
}

class D(var y: Int = 0): A(), B, C{

    override fun x(): Int {
        println("call x(): Int in D")
        if(y > 0){
            return y
        }else if(y < -200){
            return super<C>.x()
        }else if(y < -100){
            return super<B>.x()
        }else{
            return super<A>.x()
        }
    }
}

fun main(args: Array<String>) {
    println(D(3).x())
    println(D(-10).x())
    println(D(-110).x())
    println(D(-10000).x())
}
```


Hi，Kotliners，尽管视频完结了，不过每周一我还是会给大家推送一些 Kotlin 的有意思的话题，如果大家对视频有兴趣，直接点击阅读原文就可以找到~

[Kotlin 从入门到『放弃』视频教程](#)

这一期给大家讲一个有意思的东西。我们都知道 Java 当年高调的调戏 C++ 的时候，除了最爱说的内存自动回收之外，还有一个著名的单继承，任何 Java 类都是 Object 的子类，任何 Java 类有且只有一个父类，不过，它们可以有多个接口，就像这样：

```
public class Java extends Language implements JVMRunnable{  
    ...  
}  
  
public class Kotlin extends Language implements JVMRunnable, FER  
unnable{  
    ...  
}
```

这样用起来真的比 C++ 要简单得多，不过有时候也会有些麻烦：Java 和 Kotlin 都可以运行在 JVM 上面，我们用一个接口 JVMRunnable 来标识它们的这一身份；现在我们假设这两者对于 JVMRunnable 接口的实现都是一样的，所以我们将会在 Java 和 Kotlin 当中写下两段重复的代码：

```
public class Java extends Language implements JVMRunnable{
    public void runOnJVM(){
        ...
    }

    public class Kotlin extends Language implements JVMRunnable, FER
    unnable{
        public void runOnJVM(){
            ...
        }
        public void runOnFE(){
            ...
        }
    }
}
```

重复代码使我们最不愿意看到的，所以我们决定创建一个 JVMLanguage 作为 Java 和 Kotlin 的父类，它提供默认的 runOnJVM 的实现。看上去挺不错。

```
public abstract class JVMLanguage{
    public void runOnJVM(){
        ...
    }
}

public class Java extends JVMLanguage{
}

public class Kotlin extends JVMLanguage implements FERunnable{
    public void runOnFE(){
        ...
    }
}
```

当然，我们还知道 Kotlin 可以编译成 Js 运行，那我们硬生生的把 Kotlin 称作 JVMLanguage 就有些牵强了，而刚刚我们觉得很完美的写法呢，其实是不合适的。

简单的说，继承和实现接口的区别就是：继承描述的是这个类『是什么』的问题，而实现的接口则描述的是这个类『能做什么』的问题。

Kotlin 与 Java 在能够运行在 JVM 这个问题上是一致的，可 Java 却不能像 Kotlin 那样去运行在前端，Kotlin 和 Java 运行在 JVM 上这个点只能算作一种能力，而不能对其本质定性。

于是我们在 Java 8 当中看到了接口默认实现的 Feature，于是我们的代码可以改改了：

```
public interface JVMRunnable{
    default void runOnJVM(){
        ...
    }
}

public class Java extends Language implements JVMRunnable{

}

public class Kotlin extends Language implements JVMRunnable, FER
unnable{
    public void runOnFE(){
        ...
    }
}
```

这样很好，不过，由于接口无法保存状态，runOnJVM 这个方法的接口级默认实现仍然非常受限制。

那么 Kotlin 给我们带来什么呢？大家请看下面的代码：

```

abstract class Language

interface JVMRunnable{
    fun runOnJVM()
}

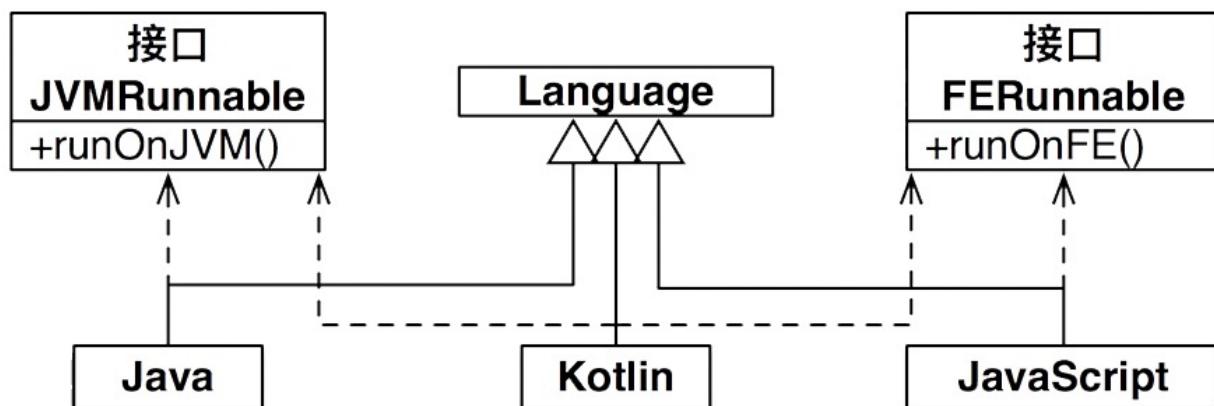
class DefaultJVMRunnable : JVMRunnable {
    override fun runOnJVM() {
        println("running on JVM!")
    }
}

class Java(jvmRunnable: JVMRunnable) : Language(), JVMRunnable by jvmRunnable

class Kotlin(jvmRunnable: JVMRunnable) : Language(), JVMRunnable by jvmRunnable, FERunnable{
    fun runOnFE(){
        ...
    }
}

```

通过接口代理的方式，我们把 `JVMRunnable` 的具体实现代理给了 `jvmRunnable` 这个实例，这个实例当然是可以保存状态的，它一方面可以很好地解决我们前面提到的接口默认实现的问题，另一方面也能在提供能力的同时不影响原有类的『本质』。



internal 模块内可见

成员默认是public

```
class House

class Flower

class Countyard{
    private val house: House = House()
    private val flower: Flower = Flower()
}

class ForbiddenCity{
    internal val houses = arrayOf(House(), House())
    val flowers = arrayOf(Flower(), Flower())
}

fun main(args: Array<String>) {
    val countyard = Countyard()
    val fc = ForbiddenCity()
    println(fc.flowers)
}
```

接口和抽象类

抽象类

```

abstract class Person(open val age: Int){
    abstract fun work()
}

class MaNong(age: Int): Person(age){

    override val age: Int
        get() = 0

    override fun work() {
        println("我是码农，我在写代码")
    }
}

class Doctor(age: Int): Person(age){

    override fun work() {
        println("我是医生，我在给病人看病")
    }
}

fun main(args: Array<String>) {
    val person: Person = MaNong(23)
    person.work()
    println(person.age)

    val person2 : Person = Doctor(24)
    person2.work()
    println(person2.age)
}

```

接口代理

```

class Manager: Driver, Writer {

```

```
override fun write() {  
}  
  
override fun drive() {  
}  
}  
  
class SeniorManager(val driver: Driver, val writer: Writer): Driver by driver, Writer by writer  
  
class CarDriver: Driver {  
    override fun drive() {  
        println("开车呢")  
    }  
}  
  
class PPTWriter: Writer {  
    override fun write() {  
        println("做PPT呢")  
    }  
}  
  
}  
  
interface Driver{  
    fun drive()  
}  
  
interface Writer{  
    fun write()  
}  
  
fun main(args: Array<String>) {  
    val driver = CarDriver()  
    val writer = PPTWriter()  
    val seniorManager = SeniorManager(driver, writer)  
    seniorManager.drive()  
    seniorManager.write()  
}
```


属性代理

```

class Delegates{
    val hello by lazy {
        "HelloWorld"
    }

    val hello2 by X()

    var hello3 by X()
}

class X{
    private var value: String? = null

    operator fun getValue(thisRef: Any?, property: KProperty<*>)
    : String {
        println("getValue: $thisRef -> ${property.name}")
        return value?: ""
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>,
    value: String){
        println("setValue, $thisRef -> ${property.name} = $value")
        this.value = value
    }
}

fun main(args: Array<String> ) {
    val delegates = Delegates()
    println(delegates.hello)
    println(delegates.hello2)
    println(delegates.hello3)
    delegates.hello3 = "value of hello3"
    println(delegates.hello3)
}

```


对象表达式和对象声明

有时候，我们需要创建一个对某个类做了轻微改动的类的对象，而不用为之显式声明新的子类。Java 用匿名内部类处理这种情况。Kotlin 用对象表达式和对象声明对这个概念稍微概括了下。

对象表达式

要创建一个继承自某个（或某些）类型的匿名类的对象，我们会这么写：

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // .....
    }

    override fun mouseEntered(e: MouseEvent) {
        // .....
    }
})
```

如果超类型有一个构造函数，则必须传递适当的构造函数参数给它。多个超类型可以由跟在冒号后面的逗号分隔的列表指定：

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B {.....}

val ab: A = object : A(1), B {
    override val y = 15
}
```

任何时候，如果我们只需要“一个对象而已”，并不需要特殊超类型，那么我们可以简单地写：

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

请注意，匿名对象可以用作只在本地和私有作用域中声明的类型。如果你使用匿名对象作为公有函数的返回类型或者用作公有属性的类型，那么该函数或属性的实际类型会是匿名对象声明的超类型，如果你没有声明任何超类型，就会是 `Any`。在匿名对象中添加的成员将无法访问。

```
class C {
    // 私有函数，所以其返回类型是匿名对象类型
    private fun foo() = object {
        val x: String = "x"
    }

    // 公有函数，所以其返回类型是 Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x           // 没问题
        val x2 = publicFoo().x    // 错误：未能解析的引用“x”
    }
}
```

就像 Java 匿名内部类一样，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，这不仅限于 `final` 变量。）

```

fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // .....
}

```

对象声明

单例模式是一种非常有用的模式，而 Kotlin（继 Scala 之后）使单例声明变得很容易：

```

object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // .....
    }

    val allDataProviders: Collection<DataProvider>
        get() = // .....
}

```

这称为对象声明。并且它总是在 object 关键字后跟一个名称。就像变量声明一样，对象声明不是一个表达式，不能用在赋值语句的右边。

要引用该对象，我们直接使用其名称即可：

```
DataProviderManager.registerDataProvider(.....)
```

这些对象可以有超类型：

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // .....
    }

    override fun mouseEntered(e: MouseEvent) {
        // .....
    }
}
```

注意：对象声明不能在局部作用域（即直接嵌套在函数内部），但是它们可以嵌套到其他对象声明或非内部类中。

伴生对象

类内部的对象声明可以用 companion 关键字标记：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

该伴生对象的成员可通过只使用类名作为限定符来调用：

```
val instance = MyClass.create()
```

可以省略伴生对象的名称，在这种情况下将使用名称 Companion：

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

请注意，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，例如还可以实现接口：

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}
```

当然，在 JVM 平台，如果使用 `@JvmStatic` 注解，你可以将伴生对象的成员生成为真正的静态方法和字段。更详细信息请参见[Java 互操作性](#)一节。

对象表达式和对象声明之间的语义差异

对象表达式和对象声明之间有一个重要的语义差别：

- 对象表达式是在使用他们的地方立即执行（及初始化）的；
- 对象声明是在第一次被访问到时延迟初始化的；
- 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。

伴生对象和静态成员

```

fun main(args: Array<String>) {
    val latitude = Latitude.ofDouble(3.0)
    val latitude2 = Latitude.ofLatitude(latitude)

    println(Latitude.TAG)
}

class Latitude private constructor(val value: Double){
    // companion object 伴生对象
    companion object{
        @JvmStatic // 可在Java代码中调用
        fun ofDouble(double: Double): Latitude{
            return Latitude(double)
        }

        fun ofLatitude(latitude: Latitude): Latitude{
            return Latitude(latitude.value)
        }

        @JvmField // 可在Java代码中访问
        val TAG: String = "Latitude"
    }
}

```

Java中调用

```

public class StaticJava {
    public static void main(String... args) {
        Latitude latitude = Latitude.ofDouble(3.0);
        System.out.println(Latitude.TAG);
    }
}

```

饿汉式单例

java实现

```
public class PlainOldSingleton {  
    private static PlainOldSingleton INSTANCE = new PlainOldSing  
leton();  
  
    private PlainOldSingleton(){  
        System.out.println("PlainOldSingleton");  
    }  
  
    public static PlainOldSingleton getInstance(){  
        return INSTANCE;  
    }  
}
```

kotlin实现

```
object PlainOldSingleton {  
}
```

非线程安全的懒汉式单例

java实现

单例

```
public class LazyNotThreadSafe {  
    private static LazyNotThreadSafe INSTANCE;  
  
    private LazyNotThreadSafe(){}  
  
    public static LazyNotThreadSafe getInstance(){  
        if(INSTANCE == null){  
            INSTANCE = new LazyNotThreadSafe();  
        }  
        return INSTANCE;  
    }  
}
```

kotlin实现

```
class LazyNotThreadSafe {  
  
    companion object{  
        val instance by lazy(LazyThreadSafetyMode.NONE) {  
            LazyNotThreadSafe()  
        }  
  
        //下面是另一种等价的写法， 获取单例使用 get 方法  
        private var instance2: LazyNotThreadSafe? = null  
  
        fun get() : LazyNotThreadSafe {  
            if(instance2 == null){  
                instance2 = LazyNotThreadSafe()  
            }  
            return instance2!!  
        }  
    }  
}
```

线程安全的懒汉式单例

java实现

```
public class LazyThreadSafeSynchronized {  
    private static LazyThreadSafeSynchronized INSTANCE;  
  
    private LazyThreadSafeSynchronized(){}  
  
    public static synchronized LazyThreadSafeSynchronized getInstance(){  
        if(INSTANCE == null){  
            INSTANCE = new LazyThreadSafeSynchronized();  
        }  
        return INSTANCE;  
    }  
}
```

kotlin实现

```
class LazyThreadSafeSynchronized private constructor() {  
    companion object {  
        private var instance: LazyThreadSafeSynchronized? = null  
  
        @Synchronized  
        fun get(): LazyThreadSafeSynchronized{  
            if(instance == null) instance = LazyThreadSafeSynchronized()  
            return instance!!  
        }  
    }  
}
```

DoubleCheck

java实现

```
public class LazyThreadSafeDoubleCheck {  
    private static volatile LazyThreadSafeDoubleCheck INSTANCE;  
  
    private LazyThreadSafeDoubleCheck(){}  
  
    public static LazyThreadSafeDoubleCheck getInstance(){  
        if(INSTANCE == null){  
            synchronized (LazyThreadSafeDoubleCheck.class){  
                if(INSTANCE == null) {  
                    //初始化时分为实例化和赋值两步，尽管我们把这一步写成  
                    //但Java虚拟机并不保证其他线程『眼中』这两步的顺序  
                    //究竟是怎么样的  
                    INSTANCE = new LazyThreadSafeDoubleCheck();  
                }  
            }  
        }  
        return INSTANCE;  
    }  
}
```

kotlin实现

```
class LazyThreadSafeDoubleCheck private constructor(){
    companion object{
        val instance by lazy(mode = LazyThreadSafetyMode.SYNCHRONIZED){
            LazyThreadSafeDoubleCheck()
        }

        private @Volatile var instance2: LazyThreadSafeDoubleCheck? = null

        fun get(): LazyThreadSafeDoubleCheck {
            if(instance2 == null){
                synchronized(this){
                    if(instance2 == null)
                        instance2 = LazyThreadSafeDoubleCheck()
                }
            }
            return instance2!!
        }
    }
}
```

静态内部类单例

java实现

```
public class LazyThreadSafeStaticInnerClass {  
  
    private static class Holder{  
        private static LazyThreadSafeStaticInnerClass INSTANCE =  
new LazyThreadSafeStaticInnerClass();  
    }  
  
    private LazyThreadSafeStaticInnerClass(){}
  
  
    public static LazyThreadSafeStaticInnerClass getInstance(){  
        return Holder.INSTANCE;  
    }  
}
```

kotlin实现

```
class LazyThreadSafeStaticInnerObject private constructor(){  
    companion object{  
        fun getInstance() = Holder.instance  
    }  
  
    private object Holder{  
        val instance = LazyThreadSafeStaticInnerObject()  
    }  
}
```

object单例

object单例

```
public class MusicPlayerJava {  
    public static MusicPlayerJava INSTANCE = new MusicPlayerJava()  
();  
  
    private MusicPlayerJava(){}  
}
```

```
class Driver  
  
interface OnExternalDriverMountListener{  
    fun onMount(driver: Driver)  
  
    fun onUnmount(driver: Driver)  
}  
  
abstract class Player  
  
object MusicPlayer: Player(), OnExternalDriverMountListener{  
    override fun onMount(driver: Driver) {  
  
    }  
  
    override fun onUnmount(driver: Driver) {  
  
    }  
  
    val state : Int = 0  
  
    fun play(url : String){  
  
    }  
  
    fun stop(){  
  
    }  
}
```


seal class

在kotlin1.1前，sealclass只能在类内部定义，在kotlin1.1后，可以定义在同一个文件中

```
sealed class PlayerCmd {
    class Play(val url: String, val position: Long = 0): PlayerCmd()
    class Seek(val position: Long): PlayerCmd()
    object Pause: PlayerCmd()
    object Resume: PlayerCmd()
    object Stop: PlayerCmd()
}

enum class PlayerState{
    IDLE, PAUSE, PLAYING
}
```

data class 数据类

- `toString()`
- `hashCode()`
- `copy()`
- `component1()` 第一个参数
- `component2()` 第二个参数

```
@PoKo
data class Country(val id: Int, val name: String)

class ComponentX{
    operator fun component1(): String{
        return "您好，我是"
    }

    operator fun component2(): Int{
        return 1
    }

    operator fun component3(): Int{
        return 1
    }

    operator fun component4(): Int{
        return 0
    }
}

fun main(args: Array<String>) {
    val china = Country(0, "中国")
    println(china)
    println(china.component1())
    println(china.component2())
    val (id, name) = china
    println(id)
    println(name)

    val componentX = ComponentX()
    val (a, b, c, d) = componentX
    println("$a $b$c$d")
}
```

Kotlin 遇到 MyBatis：到底是 Int 的错，还是 data class 的错？

[TOC]

问题出现

前不久刚刚应小伙伴的要求拉了个 QQ 群：162452394（微信群可以加我微信 enbandari，邀大家进群），上周一的时候在公众号推送了之后一下子就热闹起来了。



话说有个哥們在群里面问了这么一个问题，他用 MyBatis 来接入数据库，有个实体类用 Kotlin 大概是这么写的：

```
data class User (var id: Int, var username: String, var age: Int  
, var passwd: String)
```

它对应的数据库表是这样的：

```
CREATE TABLE userinfo
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    username VARCHAR(45),
    age INT(11),
    passwd VARCHAR(45)
);
```

字段顺序也都能对得上。

然后呢，他就配置了这么一条查询语句：

```
<mapper namespace="net.println.kotlin.mybatis.UserMapper">
    <select id="selectUser" resultType="net.println.kotlin.mybatis.User">
        select * from userinfo where id = #{id}
    </select>
</mapper>
```

对应的 UserMapper 代码如下：

```
public interface UserMapper {
    User selectUser(int id);
}
```

这一切看上去似乎一点儿毛病都没有哇，可一旦他调用 selectUser 方法之后，程序开始抱怨了：

```
No constructor found in net.println.kotlin.mybatis.User matching
[java.lang.Integer, java.lang.String, java.lang.Integer, java.lang.String]
```

啥问题呢？找不到构造方法。当时看到这个问题的时候正好手里有活，没有仔细看，周末特意照着写了个 demo，果然。。恩。。居然找不到构造方法，这就有意思了。

问题探究①——Kotlin 的类型映射

按理说，我们的 data class 是有构造方法的，说找不到构造方法倒也有些不公平，应该确切的说是找不到合适的构造方法。前面那句错误信息告诉我们 MyBatis 想要找的构造方法是下面的签名：

```
init(java.lang.Integer, java.lang.String, java.lang.Integer, java.lang.String)
```

我们的 data class 的构造方法呢？

```
init(kotlin.Int, kotlin.String, kotlin.Int, kotlin.String)
```

嗯，乍一看确实不一样哈，难怪找不到合适的构造方法。这样说对吗？我在之前有篇文章[为什么不直接使用 Array 而是 IntArray](#)？提到过 Kotlin 的类型映射的问题，kotlin.String 编译之后毫无疑问的要映射成 java.lang.String，而 kotlin.Int 则有可能映射成 int 或是 java.lang.Integer，这么说来我们的 User 的构造方法签名可能是下面这样：

```
init(int, java.lang.String, int, java.lang.String)
```

也可能是这样：

```
init(java.lang.Integer, java.lang.String, java.lang.Integer, java.lang.String)
```

现在通过刚才报的错误来看，映射后的签名毫无疑问的应该是前面那种了，毕竟这里 Int 并没有装箱的需求，为了追求效率，映射成 int 是再合适不过的了。也正是这个原因，MyBatis 才无法找到它想要的构造方法，无法构造出 User 对象，最终导致程序运行失败。

问题探究②——JavaBean 的无参构造

JavaBean 是一个很有意思的概念，刚刚接触这个概念的时候都有点儿不敢相信自己的耳朵，一个在 JavaEE 当中举足轻重的概念居然就只是一个有无参构造方法、属性通过 Getter 和 Setter 访问、可序列化和反序列化的 POJO，就这么简单？说实在的，当时真觉得 JavaBean 也没什么了不起的，就像最开始学牛二定律的时候一样，一个只有 4 个字符的定律，料它也不能把酒家怎样——可是实际上呢，它确实把我给怎样了。。

刚刚我们分析错误的时候，很直接的分析了构造方法为什么不匹配的原因，却没有想想为什么要找这个构造方法，试想，如果你用 Java 写这段代码，你肯定会写出类似下面的代码：

```
public class User{  
    private int id;  
    private String username;  
    private int age;  
    private String passwd;  
  
    ... 省略 getter 和 setter  
}
```

如果不纠结序列化的事儿，这个 User 就是个 JavaBean 是吧，你交给 MyBatis 使用的话也不会出现任何问题——MyBatis 压根儿不需要找什么构造方法，因为人家根本不需要费那劲，有无参的默认构造方法的话，构造对象实例岂不是轻而易举？

对咯，MyBatis 其实想要的是一个 JavaBean，一个有默认无参构造方法的类，结果呢，你给人家塞了一个 data class 过去。。

解决方案 ① —— 我就用 Integer 了怎么着吧

这个问题有一个最为直接的解决办法，那就是直接使用 Integer 而不是 kotlin.Int。

```
data class User (var id: Integer, var username: String, var age:  
    Integer, var passwd: String)
```

不过，你一旦这么写了，你就没办法在 Kotlin 当中正常实例化这个类了（在 Java 中可以实例化），所以这种方案堪比七伤拳啊：

```
val user = User(1, "root", 30, "") //error : The integer literal does not conform to the expected type Integer
```

解决方案 ② —— kotlin.Int 什么时候映射为 Integer

如果 kotlin.Int 能够映射成 java.lang.Integer，那么这问题就彻底解了。试想一下，什么情况下 int 不好使，非得用 Integer？

- 整型作为泛型参数的时候
- 可以为 null

这两种情况显而易见的需要 Integer 出马了，比如你想将一堆整数放入 ArrayList 当中，你只能这么搞：

```
ArrayList<Integer> integers = new ArrayList<Integer>();  
...
```

还有一种就是整型值可能为 null 的时候，毕竟作为基本类型的 int 连默认值都是 0，怎么会为 null 呢？

回到我们的问题，如果能让 data class 的 Int 映射为 Integer，那么构造方法应该是妥妥的了：

```
data class User (var id: Int?, var username: String, var age: Int?  
, var passwd: String)
```

我们把构造方法当中的 id 和 age 的类型做了修改，从不可为空的 Int 改为可为空的 Int?，这样编译之后就只好映射为 Integer 了。

问题解决~

这个方案的优点就是几乎没有额外的依赖或者其他什么开销，只是后续编码时，你会总是被迫对 id、age 这几个属性进行是否为空的判断，这样看起来一点儿都不美。

解决方案 ③ —— 默认参数

其实就像我们前面提到的，如果 User 这个类有个无参构造的话，后面查找其他构造方法的事儿就压根儿不会有。也就是说如果我们给 User 类加一个无参构造，这个问题也是可以解决的：

```
class User {  
    var id: Int = 0  
    lateinit var username: String  
    var age: Int = 0  
    lateinit var passwd: String  
}
```

如果这样写的话，我们就无法享受 data class 带来的书写便利了。。不过如果我们能够骗过 MyBatis 说我们这个类有无参构造，那么问题不就解决了？

```
data class User (var id: Int = 0, var username: String = "", var  
age: Int = 0, var passwd: String = "")
```

我们为每一个参数加了默认值，这样编译出来之后，字节码当中就真的会看到有无参构造方法了：

```
public <init>()V  
L0  
  ALOAD 0  
  ICONST_0  
  ACONST_NULL  
  ICONST_0  
  ACONST_NULL  
  BIPUSH 15  
  ACONST_NULL  
  INVOKESPECIAL net/println/kotlin/mybatis/User.<init> (ILjava  
/lang/String;ILjava/lang/String;ILkotlin/jvm/internal/DefaultCon  
structorMarker;)V  
  RETURN  
L1  
  MAXSTACK = 7  
  MAXLOCALS = 1
```

实际上我们也可以通过反射来获得到这个无参的构造方法，也正是因为如此，我们也可以直接用 newInstance 方法来构造 User 实例：

```
User::class.java.newInstance()
```

既然有了无参构造方法，MyBatis 就不需要绞尽脑汁还要找其他的构造方法，于是问题解决~

这个方案的优点是，比较简单，也没有上一个方案那样的副作用；缺点就是，万一某一个属性没有默认值，你该给它设置什么呢？

解决方案 ④ —— 官方也认为有时候我们需要一个无参构造

早在 1.0.6 发版的时候，官方就增加了对无参构造的一种另类支持，即 noarg 插件。Kotlin 原本不需要这么做，但考虑到它与 Java 解不开理还乱的关系，Java 支持的一切代码的写法 Kotlin 也似乎有责任和义务来完全支持了。

这个方法其实是 Kotlin 编译插件在编译器通过字节码织入的方式向 class 文件中写入了一个无参构造方法，这个构造方法由于出现的时间比较晚，我们无法在代码中引用到它，不过却可以通过反射访问到它，这样就即保证了 Kotlin 的初心不变，如果你愿意用 data class 或者类似的实体类，那么你就要按照 Kotlin 的要求妥善处理好它的成员的初始化，也方便了一些框架的“出格”行为，显然一个聪明的框架需要对代码本身有足够的理解，对编码人员的限制对于框架本身来说就显得没有那么的重要了。

如果你遇到了这样的问题，我当然建议你采用官方的这个解决方案，原因很简单，除了要写一个注解之外，几乎没有任何副作用，另外，官方支持的方案自然也比较有保障啦。

拓展延伸 —— 不择手段创建实例

说起来我就要批评一下 MyBatis 了，一点儿都不如 Gson 流氓。我们前面虽然没有细说，不过大家基本上可以知道 MyBatis 是如何创建返回结果的实例的：

```

private Object createResultObject(ResultSetWrapper rsw, Result
Map resultMap, List<Class<?>> constructorArgTypes, List<Object>
constructorArgs, String columnPrefix)
    throws SQLException {
    ...
} else if (resultType.isInterface() || metaType.hasDefaultCo
nstructor()) {
    //有无参构造方法的话走的是这个分支
    return objectFactory.create(resultType);
} else if (shouldApplyAutomaticMappings(resultMap, false)) {
    //在这里查找与表结构匹配的构造方法，我们之前遇到的错误就在这个方法当
中抛出
    return createByConstructorSignature(rsw, resultType, const
ructorArgTypes, constructorArgs, columnPrefix);
}
...
}

```

我们看到如果没有找到匹配的构造方法，也没有无参的构造方法，MyBatis 就叹了一口气，放弃了。这样的事情如果交给 Gson，你就会发现完全不一样。我曾在[12 Json数据引发的血案](#)这一期当中介绍过 Gson 如何创建实例，它甚至可以让 Kotlin 的不可空类型“赋值”为 null，原因很简单，它在实例化对象的时候也跟 MyBatis 一样，先去找无参构造，找不到就用 Unsafe.allocateInstance 来创建对象，主要这个创建方法非常的底层，你可以简单的理解为只为实例化出来的 Java 对象开辟了对象存储需要的空间，而对应地它的成员没有一个会正常初始化。

```

class Test{
    init {
        println("init")
    }

    companion object{
        init {
            println("cinit")
        }
    }
}

```

注意到这段代码，cinit 将在 Test 类加载时打印，init 将在 Test 实例化时打印。

```
val field = Unsafe::class.java.getDeclaredField("theUnsafe")
field.isAccessible = true
val unsafe = field.get(null) as Unsafe
unsafe.allocateInstance(Test::class.java)
```

我们运行这样的程序，结果只有 cinit，难怪人家叫 Unsafe，都告诉你 Unsafe 了你还想要什么。。

不过这在 C++ 当中，简直不叫事儿，不信给你看一段代码：

```
class Hello {
public:
    int getNum();

    int checkNum(int a, int b = 0);
};

...
int Hello::getNum() {
    return 12310;
}

...
using namespace std;

int main() {
    cout << ((Hello*)0)->getNum() << endl;
    return 0;
}
```

我们把一个 0 强转为 Hello 类型的指针，接着还调用人家的函数 getNum，结果你猜怎么着？运行结果还是对的！

如果你经常接触 Jni，你也经常会把 native 的指针传给 Java，Java 拿到的其实就是一个 long 类型的数，等 Java 需要调用 native 代码的时候，你就会发现这个整数传给 native 层会首先被 reinterpret_cast。

这有什么稀罕的，反正你创建的类也好，对象也好，最终都是数，严格的语法限制也不过是编译器给我们盖起的围墙，你通过围墙来保护你自己，同时也让围墙遮挡了你的眼睛。



Kotlin

长按识别二维码，关注Kotlin

函数

- 函数
- inline 函数
- 闭包
- 细说 Lambda 表达式
- 高阶函数_1
- 高阶函数_2
- 像写文章一样使用 Kotlin
- 函数复合
- 函数式编程概述
- 在Kotlin中使用函数式编程

函数

本章节主要介绍Kotlin中的函数相关的概念。

普通函数

声明

即使Kotlin是一门面向对象的编程语言，它也是有函数的概念的——而不像Java那样，仅仅有“方法”。

Java中有静态方法来代替函数的作用，但是Kotlin的函数比Java的静态方法自由度大很多。我们先来看几个例子。

函数声明使用fun保留字，语法比较类似Scala：

```
fun function() {  
    println("This is a function")  
}
```

如果你需要一个有参数的函数：

```
fun functionWithParams(obj: Any?) {  
    println("You have passed an object to this function: $obj")  
}
```

上面的函数有一个Any?类型的参数。类似的例子还有程序的入口main函数：

```
fun main(args: Array<String>) {  
}
```

如果你需要一个返回值的话：

```
fun functionWithReturnValue(): Int {  
    return Random().nextInt()  
}
```

如果一个函数不返回东西，你可以不写返回值。也可以让它显示返回Unit：

```
fun functionReturnsUnit(): Unit {  
}
```

如果一个函数不会返回（也就是说只要调用这个函数，那么在它返回之前程序肯定GG了（比如一定会抛出异常的函数）），因此你也不知道返回值该写啥，那么你可以让它返回Nothing：

```
fun functionReturnsNothing(): Nothing {  
    throw RuntimeException("")  
}
```

一个函数也可以拥有Java风格的泛型参数：

```
fun <T> functionWithGenericsParam(t: T): T {  
    return t  
}
```

Kotlin中的泛型概念和Java基本相同，这里不再展开讨论。关于型变、协变等Java中没有的复杂泛型概念将在下文讨论。

函数参数可以有默认值（关于这个函数中for循环的中缀语法下文会提到，这里可以先忽略）：

```
fun functionWithDefaultParams(limit: Int = 10): Int {  
    for (i in 0..limit step 2) println(i)  
    return limit  
}
```

于是你不再需要像Java那样为了默认参数而写一大串重载函数了。当然，Kotlin也支持重载。

如果一个函数的函数体只需要一个表达式就可以计算出来，比如考虑如下函数：

```
fun functionReturningIncreasedInteger(num: Int): Int {  
    return num + 1  
}
```

你可以直接使用这种语法（expression function body）：

```
fun functionReturningIncreasedInteger(num: Int) = num + 1
```

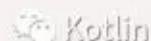
第二种语法省略了返回值类型（编译器可以根据后面的表达式推导出返回类型），以及大括号。

同理，比如说我们有一个求平方的函数：

```
fun square(int: Int) = int * int
```

如果一个函数是空函数，比如Swing的ActionListener强制要求重载但是又不需要使用的函数，可以通过这种方式来表达：

```
fun main(args: Array<String>) {  
    val input = JTextField()  
    input.addKeyListener(object : KeyListener {  
        override fun keyTyped(e: KeyEvent?) = Unit  
        override fun keyReleased(e: KeyEvent?) = Unit  
        override fun keyPressed(e: KeyEvent?) {  
            if (e != null && e.keyCode == KeyEvent.VK_ENTER) {  
                output.append("${input.text}\n", Color(0x467CDA))  
                repl.handle(input.text, sl)  
                input.text = ""  
            }  
        }  
    })  
}
```



其中，`fun xxx() = Unit` 表示这是一个空函数。

如果一个函数不返回Unit或者Nothing，那么尽可能让它成为一个纯函数（即没有副作用的函数）。这是函数式编程的约定。

关于函数式编程：函数式编程是一种学院派的编程范式，与之对应的是“命令式编程”。它常常与数学中的范畴论（category theory）结合，追求编写更易维护、并发性更好的程序。业界对它褒贬不一，笔者倾向于将它与面向对象编程结合。

虽然Hadi Hariri曾经在JetBrains中国开发者日上说过Kotlin不是函数式编程语言，但是Kotlin有大量的（看起来非常）函数式的特性、约定、标准库函数。因此我们可以默认它是追求函数式编程的，那么就应该尽可能遵循函数式法则。

为什么返回Unit或者Nothing就不需要纯了呢？

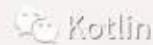
既然一个函数没有返回值，那么对它讨论引用透明也就没有意义了。如果一个返回Unit或者Nothing的函数没有副作用，那么它的存在也将没有意义。

更多关于函数式编程的知识不在本书讨论范围内，读者可以自行查询相关资料。

内部函数

函数里面也可以定义函数：

```
fun functionWithAnotherFunctionInside() {
    val valuesInTheOuterScope = "Kotlin is awesome!"
    fun theFunctionInside(int: Int = 10) {
        println(valuesInTheOuterScope)
        if (int >= 5) theFunctionInside(int - 1)
    }
    theFunctionInside()
}
```



内部函数可以直接访问外部函数的局部变量、常量，Java是做不到这点的哦。

递归也没有任何问题。如上面的代码所示。

方法

方法是一种特殊的函数，它必须通过类的实例调用，也就是说每个方法可以在方法内部拿到这个方法的实例。这是方法和函数的不同之处。

方法和函数几乎一模一样，唯一的区别就是方法必须声明在类里面。下面是一个方法和一个函数：

```
fun thisIsAFunction() = Unit

class ThisIsAClass {
    fun thisIsAMethod() = Unit
}
```

中缀表达式

可能你会好奇上面给出的一个例子中有一个for循环使用的诡异语法，即中缀表达式：

```
fun main(args: Array<String>) {
    for (i in 1..100 step 20) {
        println(i)
    }
}
```

Kotlin

这个step是什么意思呢？

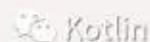
其实是Kotlin方法的一种语法糖，一个方法如果在声明时有一个infix修饰符，那么它可以使用中缀语法调用。

所谓中缀语法就是不需要点和括号的方法调用：

```
class B

class A {
    infix fun infixFunction(b: B) {
        // codes
    }
}

fun main(args: Array<String>) {
    val a = A()
    a infixFunction b
}
```



同理，我们只需要给一个类增加一个中缀的step方法就可以实现上面的语法：

```
class A {
    infix fun step(step: Int) = Unit
}

fun main(args: Array<String>) {
    val a = A()
    a step 2
}
```

这个功能基本就是用于将代码变得“更易阅读”。 Scala所有函数缺省支持中缀表达式，Kotlin需要单独声明。

操作符重载

中缀表达式可以在一定程度上简化函数调用的代码。如果说中缀表达式还不够简洁，那么你一定需要操作符重载了。

Kotlin的操作符重载的规则是：

1. 该方法使用operator修饰符
2. 该方法的方法名必须被声明为特定的名称，以将对应的操作符映射为这个函数的调用
3. 参数必须符合该操作符的规定，比如+的重载就不能有多于一个（不含）的参数，也不能为空参数。

举个例子，我们要重载A类的+运算符。注意三个规定（函数名、参数得符合规矩，加operator修饰）：

```
class A {  
    operator fun plus(a: A) {  
        println("invoking plus")  
    }  
}
```

这就是对+的重载了。我们可以这样调用这个函数：

```
val a = A() + A()
```

此处的+就是对plus方法的调用了。我们可以运行上面的代码，看到输出："invoking plus"。

当然，操作符重载也可以被替换为函数调用：

```
val a = A().plus(A())
```

这段代码和上面那段完全相同。

还有不少操作符的重载，下面给出一系列例子（仅作为函数名的实例，实现全部为空/false/0（因为部分操作符重载要求返回Int/Boolean））：

```
class A {  
    operator fun plus(a: A) = Unit  
    operator fun minus(a: A) = Unit  
    operator fun times(a: A) = Unit  
    operator fun div(a: A) = Unit  
    operator fun rem(a: A) = Unit  
    @Suppress("DEPRECATED_BINARY_MOD")  
    @Deprecated("mod should be replaced with rem", ReplaceWith("re  
m"))  
    operator fun mod(a: A) = Unit  
    operator fun rangeTo(a: A) = Unit  
  
    operator fun get(index: Int) = Unit  
    operator fun get(index1: Int, index2: Int) = Unit  
    operator fun set(index: Int, value: Any?) = Unit  
    operator fun set(index1: Int, index2: Int, value: Any?) = Unit  
  
    operator fun invoke(obj: Any?)  
  
    operator fun inc() = A()  
    operator fun dec() = A()  
    operator fun unaryPlus() = A()  
    operator fun unaryMinus() = A()  
  
    operator fun compareTo(other: Any?) = 0  
    override operator fun equals(other: Any?) = false  
  
    operator fun contains(element: Any?) = false  
  
    operator fun iterator() = object : Iterator<Any> {  
        override operator fun hasNext() = false  
        override operator fun next() = Unit  
    }  
}
```

首先是最基本的操作符，它们对应的操作符和注释中的一一对应：

```
class A {  
    operator fun plus(a: A) = Unit // +  
    operator fun minus(a: A) = Unit // -  
    operator fun times(a: A) = Unit // *  
    operator fun div(a: A) = Unit // /  
    operator fun rem(a: A) = Unit // %  
  
    @Suppress("DEPRECATED_BINARY_MOD")  
    @Deprecated("mod should be replaced with rem", ReplaceWith("rem"))  
    operator fun mod(a: A) = Unit // %, 和rem一样  
  
    operator fun rangeTo(a: A) = Unit // ..  
}
```

倒数第二个mod是特殊的操作符重载——它在Kotlin1.1中被标记为过时(Deprecated)的。在1.1中，使用了rem(remainder)来代替mod，符合java.math.BigInteger的命名。

关于最后那个rangeTo有些不符合人的直觉，下面讲到contains的时候会一起提到，读者可以暂时放下这个问题。

然后是下标访问操作符：

```
class A {  
    operator fun get(index: Int) = Unit  
    operator fun get(index1: Int, index2: Int) = Unit  
    operator fun set(index: Int, value: Any?) = Unit  
    operator fun set(index1: Int, index2: Int, value: Any?) = Unit  
  
}
```

get接收任意数量的int参数，假设它们是index1, index2, index3(以此类推)，那么对应的操作符就是对应维数个下标的访问。比如以下例子，操作符对应的方法调用写在行尾注释里了：

```
fun main(args: Array<String>) {
    val a = A()
    a[1] // a.get(1)
    a[1][2] // a.get(1, 2)
}
```

对应的set接收任意数量个int，以及一个任意类型的对象，表示将下标访问作为左值并赋值：

```
fun main(args: Array<String>) {
    val a = A()
    a[1] = 233 // a.set(1, 233)
    a[1][2] = "666" // a.set(1, 2, "666")
}
```

invoke方法允许你把一个Kotlin对象当作Lambda表达式来使用。关于什么是Lambda表达式下文会专门介绍。

```
class A {
    operator fun invoke(obj: Any?)
}

fun main(args: Array<String>) {
    val a = A()
    a(obj) // 实际上调用了 a.invoke(obj)
    // 还可以
    A()(obj)
}
```

自增自减运算符就更简单了，它们必须返回自己所在类的子类型。这里直接在注释里面写出对应的操作符表达：

```
// 假设a是A的一个实例
class A {
    operator fun inc() = A() // a++
    operator fun dec() = A() // a--
    operator fun unaryPlus() = A() // ++a
    operator fun unaryMinus() = A() // --a
}
```

还有比较运算符，这是一个特殊的操作符重载，一个函数将会生成一组操作符。有一个特殊情况，就是相等的判断。

考虑以下代码：

```
class A {
    operator fun compareTo(other: Any?) = 0
    override operator fun equals(other: Any?) = false
}
```

当一个类只有`compareTo`没有`equals`的时候，所有的六个比较运算符（`<`, `>`, `<=`, `>=`, `==`, `!=`）会被全部代理给`compareTo`函数的返回值和0的大小比较：

```
fun main(args: Array<String>) {
    A() > A() // A().compareTo(A()) > 0
    A() < A() // A().compareTo(A()) > 0
    A() >= A() // A().compareTo(A()) >= 0
    A() <= A() // A().compareTo(A()) <= 0
    A() == A() // A().compareTo(A()) == 0
    A() != A() // A().compareTo(A()) != 0
}
```

如果有`equals`, `==`和`!=`两个操作符会被代理给`equals`方法，其余不变。

```
fun main(args: Array<String>) {
    A() == A() // A().equals(A())
    A() != A() // !A().equals(A())
}
```

有一个操作符，这个操作符不是严格意义上的“操作符”，但它比起其它的操作符重载，它还多了一个特权。它就是in操作符。

```
class A {  
    operator fun contains(element: Any?) = false  
}
```

它要求返回Boolean，传入一个参数。调用的话使用in：

```
fun main(args: Array<String>) {  
    val ls = listOf(12, 233, 43)  
    println(233 in ls)  
}
```

这里的in就是调用了ls的contains方法。这里你可能要问了：既然是使用in这个符合Kotlin函数命名规范的表达，为什么不使用中缀表达式呢？

想清楚了，我们暂且不说in其实是Kotlin保留字这个问题，中缀表达式的语法是：

```
fun main(args: Array<String>) {  
    a.func(b)  
    // 变成  
    a func b  
    // 它不能变成  
    // b func a  
    // 上面那个是错的，会变成b.func(a)  
}
```

但是a in b其实是调用了b的一个方法，因此中缀表达式无法实现这个in。

前面说到contains操作符有一个特权，我们来看看这个特权。还记得when语句吗？它非常灵活，比Java风格的switch不知道高到哪里去了。这里先看一个基本用法：

```
fun main(args: Array<String>) {
    val a = Random(System.currentTimeMillis()).nextInt()
    when (a) {
        1 -> { /* codes */ 233 }
        2 -> 233
        else -> 666
    }
}
```

它可以配合contains方法：

```
fun main(args: Array<String>) {
    val a = Random(System.currentTimeMillis()).nextInt()
    when (a) {
        in 1..100 -> 233
        is Int -> 2333
        else -> 666
    }
}
```

看到了吗？in还可以这样写哦。

还有一组运算符，它们将被编译为相同的一个操作符，也是in，只不过场合不同。

```
class A {
    operator fun iterator() = Unit
    operator fun hasNext() = false
    operator fun next() = Unit
}
```

这个in是用于for-in循环的。上面例子中的for循环使用的是这个in操作符。

一个标准的Kotlin风格的for循环应该是下面这样的： kotlin fun main(args: Array<String>) { for (i in 1..10) println(i) }

它和下面的代码完全等价：

```
fun main(args: Array<String>) {
    val range = 1..10
    val item = range.iterator()
    while (item.hasNext()) {
        println(item.next())
    }
}
```

还记得那个step吗？它其实是对1..100调用了一个中缀的方法step。这下读者对于前面遗留的疑问就全部解决了。

Kotlin关于操作符重载的内容确实比较繁杂，但是比起C++的操作符重载还是要强大那么一点点的（C++没有in这种操作符）。

Lambda表达式

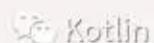
好了这是函数式编程的重头戏——Lambda表达式。

Lambda表达式俗称匿名函数，熟悉Java的大家应该也明白这是个什么概念。Kotlin的Lambda表达式更“纯粹”一点，因为它是真正把Lambda抽象为了一种类型，而Java只是单方法匿名接口实现的语法糖罢了。

Lambda在Java中非常常用，这里不再单独介绍它。

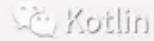
Lambda表达式最常用的地方之一是为GUI控件设置监听器：

```
fun main(args: Array<String>) {
    val button = JButton("Clear screen")
    button.addActionListener {
        output.text = ""
        output.append("Repl.HINT")
    }
}
```



还有调用集合框架抽象出来的高阶函数（Higher Order Function）：

```
fun main(args: Array<String>) {  
    (1..10).forEach {  
        println(it)  
    }  
}
```



上面的代码可能会令读者产生疑惑，那个it是什么鬼？

it是Kotlin为单参数的Lambda钦定的默认参数名。单参数的Lambda可以省去参数的声明，转而使用it这个名字。

首先，我们有无参数的Lambda表达式：

```
fun main(args: Array<String>) {  
    {}  
}
```

你可以直接把它写在外面，并且把它当成对象使用。Lambda对象有缺省的invoke函数的实现，也就是调用这个Lambda：

```
fun main(args: Array<String>) {  
    { println("invoking lambda") }.invoke()  
}
```

你也可以使用操作符重载的方式调用invoke函数：

```
fun main(args: Array<String>) {  
    { println("invoking lambda") }()  
}
```

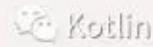
如果一个Lambda有参数，那么需要使用这样的语法声明：

```
fun main(args: Array<String>) {  
    // {参数名:参数类型 -> Lambda体}()  
    { str: String -> println(str) }("the deep dark fantasy")  
}
```



如果有多个参数：

```
fun main(args: Array<String>) {  
    // {参数名:参数类型, 参数名:参数类型 -> Lambda体}()  
    { str: String, num: Int ->  
        println("num = $num, str = $str")  
    }("the deep dark fantasy", 233)  
}
```



Lambda变量的签名不需要写参数名，只需要写类型。Lambda的返回值就是最后一个表达式的返回值。就像这样：

```
fun main(args: Array<String>) {  
    // 只需要写类型，我没骗你吧  
    val value: (String, Int) -> Int  
    value = { str: String, num: Int ->  
        println("num = $num, str = $str")  
        num  
    }  
    value("the deep dark fantasy", 233) + 1  
}
```

如果Lambda的参数、返回值类型不匹配，互相赋值是会报错的哦。

当然，Lambda经常作为函数参数使用。

- 如果一个Lambda是一个函数的最后一个参数，Kotlin为这种情况专门提供了一个语法糖。
- Lambda在作为匿名变量传递给函数时，不需要显式声明参数类型（因为可以根据函数参数那里的类型签名推导）。

上面两句话如何理解呢？比如你有一个自定义UI控件：

```
class KotlinButton : View() {  
    fun setOnClickListener(block: (View) -> Unit) {  
        // set  
    }  
}
```

然后调用的时候就可以使用这个语法糖了：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        val a = KotlinButton()
        // 原本是：
        // a.setOnClickListener { view ->
        //     toast("Clicked")
        // })
        // 语法糖省去了表示参数的括号，直接写Lambda
        a.setOnClickListener { view ->
            // 编译器的类型推导省去了 view: View 的类型声明
            toast("Clicked")
            view.text = "Oh my god"
        }
    }
}
```

上面那段代码是Android的，熟悉的API看出来了吗？

你甚至还可以把那个view参数的声明给省了，因为只有一个参数，可以使用it代替：

```
fun main(args: Array<String>) {
    a.setOnClickListener {
        toast("Hia hia")
        it.text = "clicked"
    }
}
```

如果一个函数不只是有一个参数，但是最后一个参数（依然符合条件），那么可以这样写：

```
fun main(args: Array<String>) {
    a.callSomeMethods(1, 2, 3) {
        // do something
    }
}
```

它完全等价于：

```
fun main(args: Array<String>) {
    a.callSomeMethods(1, 2, 3, {
        // do something else
    })
}
```

如果你不需要用到那个参数，又不想为它命名，那么使用下划线代替它吧。

```
fun main(args: Array<String>) {
    a.setOnClickListener { _ ->
        toast("LOL LOL LOL")
    }
}
```

此时你可能会问：上面那个代码完全可以把整个 `_ ->` 给省掉啊？不是说单参数可以省吗？

因为多参数是不能省的（多参数Lambda不写参数会报错），所以它还是很有用的：

```
fun main(args: Array<String>) {
    listOf(1, 2, 3).forEachIndexed { _, _ ->
        println("Looping!")
    }
}
```

最后说的这个下划线的特性是Kotlin1.1才引入的。

请读者注意不要混淆上面的两个“多参数”，第一处说的是函数多参数，第二处说的是**Lambda**多参数。

我们可以通过Lambda表达式来勉强地实现函数的柯里化：

```
fun plus(a: Int) = { b: Int -> a + b }
```

调用就可以这样：

```
fun main(args: Array<String>) {
    println(plus(233)(666))
}
```

这有什么用呢？可以实现一个函数的部分应用。

```
fun main(args: Array<String>) {
    val a = plus(233)
    // 然后呱啦呱啦呱啦一堆东西
    val pluser2 = getIntFromSomewhere()
    println(a(pluser2))
}
```

和泛型结合之后可以泛化出上面那个函数的抽象：

```
fun <A, B, C> ((a: A, b: B) -> C).curry() =
    { a: A -> { b: B -> invoke(a, b) } }
```

plus就可以这样被抽象出来：

```
fun main(args: Array<String>) {
    val plusOrigin = { a: Int, b: Int ->
        a + b
    }
    val a = plusOrigin.curry()
    a(233)(666) // 和上面那个就是一样的了
}
```

我们还可以做出多参数的柯里化的抽象：

```
fun <A, B, C, D, E, F, G>
    ((a: A, b: B, c: C, d: D, e: E, f: F) -> G).curry() =
    { a: A -> { b: B -> { c: C -> { d: D -> { e: E ->
        { f: F -> invoke(a, b, c, d, e, f) } } } } }
```

这样的抽象可以用于更多参数的函数的柯里化：

```
fun main(args: Array<String>) {
    val makeString = { a: Int,
                      b: Short,
                      c: String,
                      d: File,
                      e: URL,
                      f: Double ->
        "$a $b $c $d $e $f"
    }
    val a = makeString.curry()
    // 调用方法一
    a(233)(666)("Gensokyo")(File("./README.md"))(URL(
        "https://github.com/ice1000"))(1.0)
    // 调用方法二
    a(233)
        .invoke(666)
        .invoke("Gensokyo")
        .invoke(File("./README.md"))
        .invoke(URL("https://github.com/ice1000"))
        .invoke(1.0)
}
```

 Kotlin

就是这样。

调用

函数调用写法和Java几乎完全一样，比如调用上面声明的几个函数：

```
fun main(args: Array<String>) {
    functionWithGenericsParam(233)
    functionReturnsNothing()
}
```

已经说过了Lambda的invoke写法所以这里不再提及。

当你需要调用一个接收了“有返回值的Lambda”的函数时，这样的写法是会报错的：

```
fun main(args: Array<String>) {
    function {
        println("Hi there!")
        // 下面这句会报错！
        // return 233
    }
}
```

因为这样的return语句在被inline之后，编译器不知道你是要return掉main函数还是这个Lambda。

因此你需要显式声明return的作用域：

```
fun main(args: Array<String>) {
    function {
        println("Hi there!")
        return@function 233
    }
}
```

这叫做“Label return”，其中 `return@` 后面的标签（Label）是你要return的Lambda所传递给的函数名（比如上面就是`function`）。

如果你的函数名是`kotlinIsAwesome`，那么你就需要这样写：

```
fun main(args: Array<String>) {
    kotlinIsAwesome {
        return@kotlinIsAwesome 233
    }
}
```

如果你使用IntelliJ IDEA，它会告诉你你需要写什么标签。

内联函数

顾名思义，就是把函数体给内联到调用处。这对程序的逻辑是没有影响的，只是对一些无关逻辑的因素有影响，比如：

- 运行效率
- 目标文件体积
- 重构上的优越性

等等。

比如标准库的一堆函数：

```
/** Prints the given message to the standard output stream. */
@kotlin.internal.InlineOnly
public inline fun print(message: Any?) {
    System.out.print(message)
}
```

内联函数最好的好处就是直接内联Lambda，不产生匿名内部类对象。这是一个非常黑的黑科技，减少Lambda对象的数量可以既保证函数式的优美代码，又不必为Lambda对象的开销买单。

如果一个内联函数没有内联到Lambda表达式，那么Kotlin编译器会给出一个警告——因为inline本身就只是为了inline掉Lambda而准备的特性，标准库会有一些专门用于内联的函数（后面会讲到）。inline一些无关紧要的函数反而会导致无谓的体积增加（JVM本身就会在运行时内联不少函数）。

Kotlin-discussion曾经出现过一个帖子，楼主说他太喜欢inline这个功能了，以至于把大量的大型函数inline的到处都是，最后导致了jar体积的肥大。

crossinline与noinline

crossinline是一种比普通inline更高级的inline方法，它只能修饰Lambda参数，用于处理一些奇怪的Lambda的内部return（有时你只是想return掉Lambda，但是内联后会导致return掉外部context）。

Kotlin编译器使用了一些奇怪的方法来复用内联Lambda产生的参数。

```
fun crossInlineFunction(crossinline f: () -> Unit) {
    f.invoke()
}
```

noinline也只能用于修饰Lambda参数，表示在内联这个函数的情况下，不对这个Lambda进行内联优化（即：依然针对Lambda产生一个匿名内部类对象，开销和Java的就一样了。有些情况编译器无法内联这个Lambda，只能提示你加上noinline。这里不再展开讲解）。

```
fun noInlineFunction(noinline f: () -> Unit) {  
    f.invoke()  
}
```

因此，请读者尽可能遵循标准规定：仅对接收Lambda的函数使用内联。

扩展函数

扩展可谓是Kotlin的“killer feature”，它只是一个语法糖，却是一个（有时）难以理解的语法糖，一般黑带Kotlin程序员会谨慎并大量地使用它。

在IDE的帮助下，扩展对原库的污染已经不再是污染，因为扩展和原方法会被高亮给清晰地区分开。

下文中，有时会把扩展函数称为扩展方法，指的实际上是同一个概念。

普通的扩展函数

说了这么多，扩展到底是什么呢？

就是使用一些语法糖来假装给一些类添加方法，并像真正的方法一样调用它。

为什么要“假装给类添加方法”呢？

你有种给java.io.File类添加一个openOrCreate方法啊？

（以上调侃并不适用于rt.jar开发人员）

你可能还是不能理解这是什么，那么看看下面的例子吧：

```
fun main(args: Array<String>) {
    val ls = java.util.ArrayList<Int>()
    ls.add(233)
    ls.add(666)
    ls.add(555)
    ls.add(1024)
    val sum = ls.fold(0) { sum, value ->
        sum + value
    }
    println(sum)
}
```

以上代码使用Kotlin1.1第一个正式版编译器编译通过。在JRE1.8_101上完美运行。

你肯定知道，java.util.ArrayList是没有fold这个方法的。

那上面的代码是怎么回事？为什么我可以在ls对象上调用一个它本来没有的方法呢？

这完全符合刚才的定义：

像真正的方法一样调用它。

那么我们来看看这个方法怎么实现吧。这里不使用集合框架，而是使用另一个例子：给File类增加一个openOrCreate方法。

首先，我们可以这样：

```
fun openOrCreate(file: java.io.File) {
    if (!file.exists()) file.createNewFile()
}
```

然后这样调用它：

```
fun main(args: Array<String>) {
    openOrCreate(File("./save.txt"))
}
```

但是如果希望一种更优美的方式，我们可以使用这个语法：

```
fun File.openOrCreate() {  
    if (!exists()) createNewFile()  
}
```

卧槽！那个 `if (!exists())` 看起来就像是直接写在 `File` 类内部的方法一样啊！

其实这里是个小小的trick，编译器会这样处理它：

```
fun openOrCreate(receiver: File) {  
    if (!receiver.exists()) receiver.createNewFile()  
}
```

也就是说，如果将 `fun openOrCreate(receiver: File)` 写成 `fun File.openOrCreate()`，那么这个函数可以直接调用 `File` 类的方法，就像它自己也是一个 `File` 类的方法一样。编译过后它会被处理为对 `receiver` 的方法调用。

也就是说，这种扩展方法是不能调用 `private`、`protected` 以及 `internal` 的属性/方法的（因为它事实上就是一个普普通通的函数罢了）。

我们来看看它的调用吧：

```
fun main(args: Array<String>) {  
    val file = File("./save.png")  
    file.openOrCreate()  
}
```

是不是很有趣啊？

而且，在意识到了这只是一个普通方法后，你可能会问：能不能将它当成普通函数而不是方法运行呢？

当然...不可以：

```
fun main(args: Array<String>) {  
    val file = File("./save.png")  
    openOrCreate(file) // error  
}
```

当然，你也可以在一个类里面对另一个类进行扩展：

```
class A(val int: Int)
class B {
    fun A.someFunction() {
        println(int)
    }

    fun anotherFunction() {
        val a = A()
        a.someFunction()
    }
}
```

这也是合法的。

在一个扩展函数内部，this所指向的就是receiver。

```
fun File.openOrCreate() {
    if (!this.exists()) this.createNewFile()
}
```

读者可以通过在扩展函数内部调用 `println(this)` 来验证。

扩展Lambda

其实扩展还有一种用途，就是使一个Lambda成为“扩展Lambda”：

```
fun main(args: Array<String>) {
    val extensionLambda = Int.{ println(this) }
    233.extensionLambda() // OK
    extensionLambda(233) // OK
}
```

看到了吗？Lambda表达式也可以被理解为是“对一个类进行扩展的Lambda”表达式。也就是说，它也可以像扩展方法一样，从内部拿到一个特定的this，然后将它作为一个扩展使用。

扩展Lambda也可以作为一个函数的参数。

不同于扩展函数，扩展Lambda可以被当作一个非扩展Lambda。

```
fun applyToFile(block: File.() -> Unit) {  
    val file = File()  
    file.block() // OK  
    block(file) // OK  
}
```

这么写，没人拦着你，我不会拦着你，编译器也不会拦着你。这个代码和上面的代码都是合法的Kotlin代码。

注意，以上代码仅为演示语法，是没有实际意义的。

我们还可以结合扩展函数和扩展Lambda：将一个扩展Lambda作为一个扩展函数的参数。

```
fun File.run(block: File.() -> Unit) {  
    this.block() // OK  
    block(this) // OK  
}
```

this一般可以省略，所以有：

```
fun File.run(block: File.() -> Unit) {  
    block()  
}
```

还可以结合Kotlin的“expression body”：

```
fun File.run(block: File.() -> Unit) = block()
```

在调用的时候，可以再结合一下Kotlin的Lambda参数语法糖：

```
fun main(args: Array<String>) {
    val file = File("./save.log")
    file.run {
        if (!exists()) createNewFile()
    }
}
```

这相当于是给File类提供了一个run的工具函数，它可以把一个Lambda应用到file对象。

请读者确保自己能读懂以上代码，它用到了很多Kotlin的（比较）高级的特性：

- 扩展函数
- 扩展Lambda
- Lambda作为最后一个参数的简化写法

到这里就结束了吗？

不，你可能已经想到了另一个JVM支持的特性，它和扩展结合起来能做到更优美。

还记得吗？这是JVM很早就引入的一个特性。

泛型。

泛型扩展

首先考虑如下代码：

```
fun <T> runWith(receiver: T, block: T.() -> Unit) {
    receiver.block()
}
```

这个函数用泛型抽象了一个“runWith”的概念，也就是说传入一个对象和一个扩展给该对象的类型的Lambda，然后在这个对象上调用这个Lambda：

```
fun main(args: Array<String>) {
    val file = File("./save.svg")
    runWith(file) {
        if (!exists()) createNewFile()
    }

    val int = 233
    runWith(int) {
        if (this <= 100) println("smaller than 100, or equaled.")
        else println("bigger than 100.")
    }
}
```

Kotlin

我们再结合扩展：

```
fun <T> T.run(block: T.() -> Unit) {
    block() // OK
    this.block() // OK
    block(this) // OK
    block.invoke(this) // OK
}
```

上面展示了四种截然不同但是完全等价的调用block的方法，读者在抄代码的时候请只保留一个。

这个run变得更玄学了，可以直接在任意类型的任意对象上调用run方法了：

```
fun main(args: Array<String>) {
    Random().nextInt(200).run {
        if (this <= 100) println("smaller than 100, or equaled.")
        else println("bigger than 100.")
    }
}
```

还记得吗？接收Lambda对象的函数是被建议写为inline的：

```
inline fun <T> T.run(block: T.() -> Unit) {  
    block()  
}
```

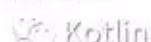
这样省去了Lambda传递的开销，和朴素写法就完全等价了。

所以亲爱的读者，还记得之前提到的集合框架多出来的fold方法吗？这是内置在Kotlin标准库的扩展函数。

Kotlin的标准库，基本由扩展函数组成。它通过扩展，结合Java标准库本身已经有的一个非常强大的集合框架，通过打包后仅仅700kb的jar作为标准库，却非比寻常的强大。

我们来看看其它JVM语言的做法：

语言	做法
Java	做的再烂也是所有JVM语言的爸爸，想打我的人多了去了你算老几
Scala	自己从头造了一个集合框架，通过隐式转换实现和Java标准库的集合框架的无缝衔接，但是开销比较大，不仅仅是运行开销，还有debug开销
Clojure	自己造了一个Lisp风格的集合框架（序列框架），Lisp的语法自带扩展效果
Groovy	相当于是从rt.jar里面抄了一些代码改成Groovy风格的形式，和Java没有互操作
JRuby/Mirah	不存在交互问题，因为这两门语言本身就不适合用于上面几门语言的领域（其实也没有交互）

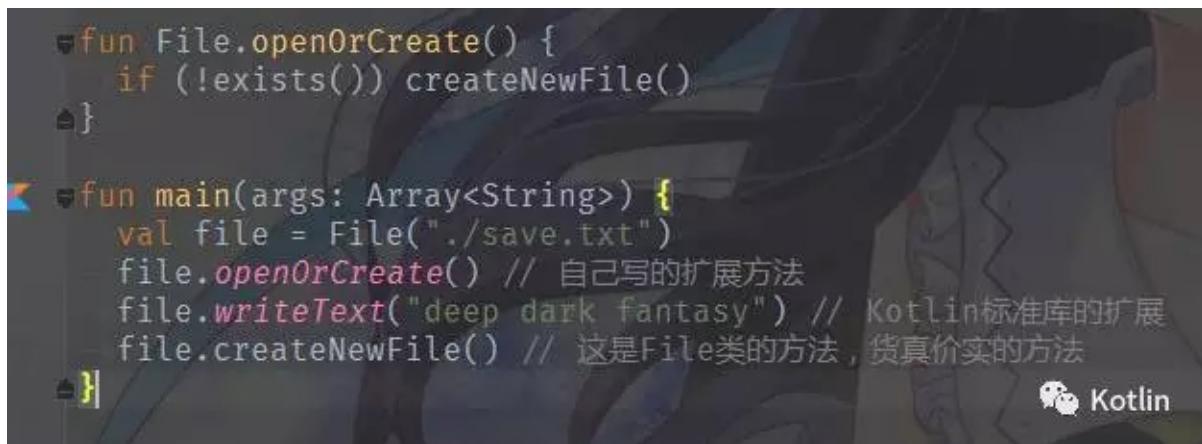


相比之下，Kotlin这种做法是非常友善的（inline零开销，完美利用rt.jar，而且可以随心所欲地扩展）。

关于扩展是否会污染原库的讨论

曾经Kotlin社区有人询问过关于扩展函数是否会污染原库的问题。

JetBrains显然考虑到了这点，他们通过IDE插件将两种方法高亮成了不同颜色，完美区分了普通方法和扩展方法：

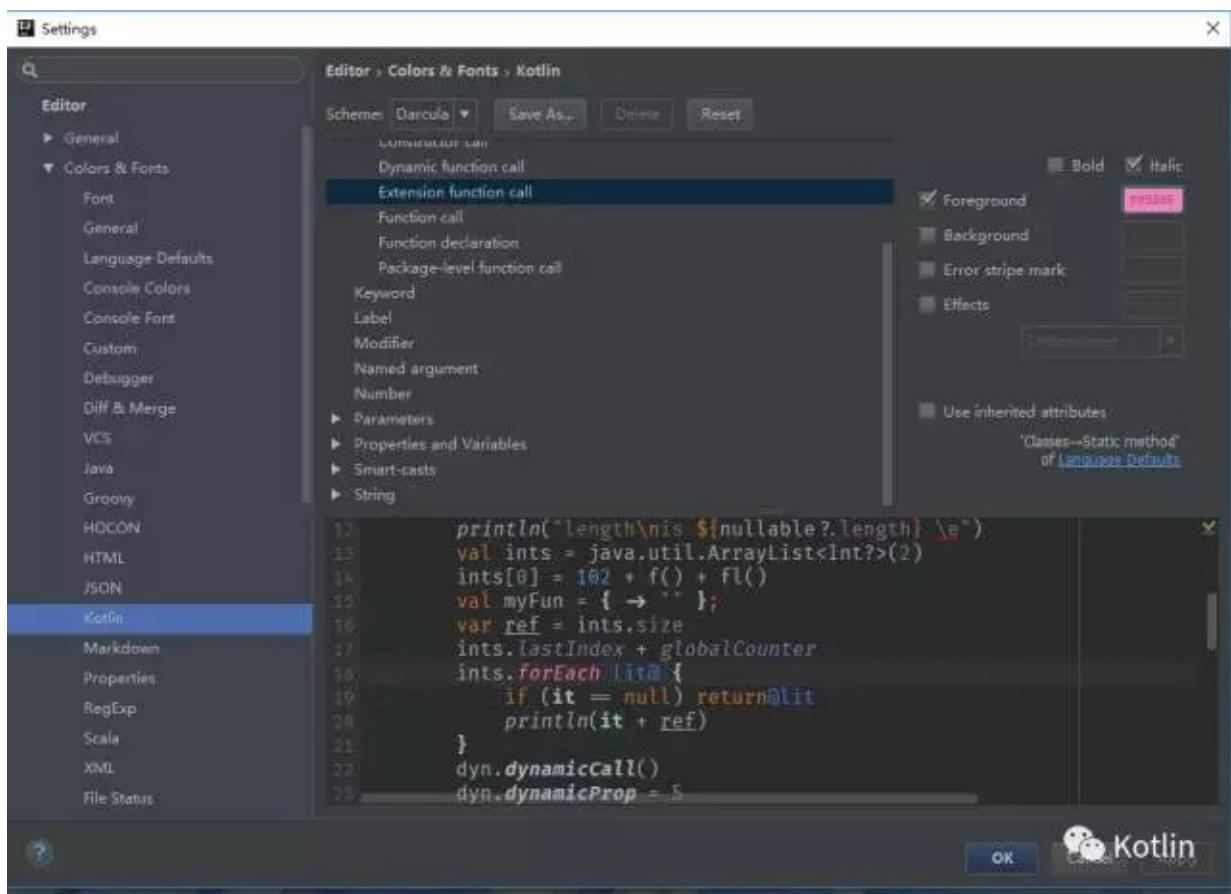


```
fun File.openOrCreate() {
    if (!exists()) createNewFile()
}

fun main(args: Array<String>) {
    val file = File("./save.txt")
    file.openOrCreate() // 自己写的扩展方法
    file.writeText("deep dark fantasy") // Kotlin标准库的扩展
    file.createNewFile() // 这是File类的方法，货真价实的方法
}
```

如果你是IntelliJ IDEA用户，那么应该早就注意到这一点了。

缺省设置是黄色，可以在这里调整：



函数

```
val FINAL_HELLO_CHINA = "HelloChina"

fun main(args: Array<String>) { // (Array<String>) -> Unit
    // checkArgs(args)
    // val arg1 = args[0].toInt()
    // val arg2 = args[1].toInt()
    // println("$arg1 + $arg2 = ${sum(arg1, arg2)}")
    // println(int2Long(3))
    // println(sum(1, 3))
    // println(sum.invoke(1, 3))

    // args.forEach ForEach@{
    //     if(it == "q") return@ForEach
    //     println(it)
    // }
    //
    // println("The End")

    println(sum)
    println(int2Long)
    println(::printUsage is ()-> Unit)
}

fun checkArgs(args: Array<String>) {
    if (args.size != 2) {
        printUsage()
        System.exit(-1)
    }
}

fun printUsage() {
    println("请传入两个整型参数，例如 1 2") // (Any?) -> Unit
} // ()->Unit

val sum = { arg1: Int, arg2: Int ->
    println("$arg1 + $arg2 = ${arg1 + arg2})"
}
```

```

    arg1 + arg2
}

// (Int, Int) -> Int

val printlnHello = {
    println("Hello")
}

// ()-> Unit

val int2Long = fun(x: Int): Long {
    return x.toLong()
}

//Int Long String ABC

// (Int) -> Long

```

- 匿名函数
- 具名函数::fun_name
- 命名参数
- 包级函数
- Lambda表达式：{[参数列表] -> [函数体，最后一行是返回值]}
- 如果函数参数的最后一个参数是Lambda表达式，可以把它移到()的外边,例如()
{}；如果()中什么也没有，可以把()删掉；如果传入的函数跟Lambda表达式是一样的，可使用::fun_name（函数引用）简化;跳出表达式，
- return标签 ForEach@
- 函数类型
- () invoke()
- 遍历数组 for(i in arr) args.forEach()
- 扩展方法，有一个默认参数为调用该方法的实例
- 具名参数
- 默认参数
- 单表达式可以省略花括号，在=号后指定代码体即可
- 函数作用域：顶层函数、局部函数
- 内联函数
- Unit ==> void

```
args.forEach({it -> print(it)})
args.forEach({print(it)})
args.forEach(){print(it)}
args.forEach{print(it)}
args.forEach(::print)
```

闭包

闭包是由函数及其相关的引用环境组合而成的实体(即：闭包=函数+引用环境)。

- 函数是什么

地球人都知道：函数只是一段可执行代码，编译后就“固化”了，每个函数在内存中只有一份实例，得到函数的入口点便可以执行函数了。在函数式编程语言中，函数是一等公民（First class value：第一类对象，我们不需要像命令式语言中那样借助函数指针，委托操作函数），函数可以作为另一个函数的参数或返回值，可以赋给一个变量。函数可以嵌套定义，即在一个函数内部可以定义另一个函数，有了嵌套函数这种结构，便会产生闭包问题。

单表达式函数

```
fun double(x: Int): Int = x * 2
fun double(x: Int) = x * 2
```

匿名函数

```
fun(x: Int, y: Int): Int = x + y

fun(x: Int, y: Int): Int {
    return x + y
}

ints.filter(fun(item) = item > 0)
```

函数类型

FunctionN，N表示函数参数个数，0~22

```
Function0
Function1
Function2<in P1, in P2, out R>:Function<R>
...
Function22
```

中缀表达式 infix

可变参数

可变参数 vararg，可以使用*操作符将可变参数以命名形式传入

```
fun main(vararg args: String) {
    //    for (arg in args){
    //        println(arg)
    //    }

    val list = arrayListOf(1,3,4,5)
    val array = intArrayOf(1,3,4,5)
    hello(3.0, *array) // 伸展 (spread) 操作符*，只能展开数组
}
```



```
fun hello(double: Double, vararg ints: Int, string: String = "Hello") {
    println(double)
    ints.forEach(::println)
    println(string)
}
```

高阶函数

高阶函数是将函数用作参数或返回值的函数。

inline 函数

inline内联函数，最终编译时是要编译到调用它的代码块中，这时候T的类型实际上是确定的，因而Kotlin通过reified这个关键字告诉编译器，T这个参数可不只是个摆设，我要把它当实际类型来用呢。

```
inline fun <reified T: Any> Gson.fromJson(json: String): T{
    return fromJson(json, T::class.java)
}

interface Api {
    fun getSingerFromJson(json: String): BaseResult<Singer>
}

object ApiFactory {
    val api: Api by lazy {
        Proxy.newProxyInstance(ApiFactory.javaClass.classLoader,
        arrayOf(Api::class.java)) {
            proxy, method, args ->
            val responseType = method.genericReturnType
            val adapter = Gson().getAdapter(TypeToken.get(responseType))
            adapter.fromJson(args[0].toString())
        } as Api
    }
}

fun main(args: Array<String>) {
    val json = File("result_singer_field_loss.json").readText()
    val result : BaseResult<Singer> = ApiFactory.api.getSingerFromJson(json)
    println(result.content.name.isEmpty())
}
```

```
package com.benny.utils
import android.util.Log
inline fun <reified T> T.debug(log: Any){
    Log.d(T::class.simpleName, log.toString())
}

debug(whatever)
```

函数编程之闭包漫谈(Closure)

函数是什么

```
>>> def ExFunc(n):
    sum=n
    def InsFunc():
        return sum+1
    return InsFunc

>>> myFunc=ExFunc(10)
>>> myFunc()
>>> myAnotherFunc=ExFunc(20)
>>> myAnotherFunc()
>>> myFunc()
>>> myAnotherFunc()
>>>
```

在这段程序中，函数InsFunc是函数ExFunc的内嵌函数，并且是ExFunc函数的返回值。我们注意到一个问题：内嵌函数InsFunc中引用到外层函数中的局部变量sum，IronPython会这么处理这个问题呢？先让我们来看看这段代码的运行结果。当我们调用分别由不同的参数调用ExFunc函数得到的函数时（myFunc()，myAnotherFunc()），得到的结果是隔离的，也就是说每次调用ExFunc函数后都将生成并保存一个新的局部变量sum。其实这里ExFunc函数返回的就是闭包。

『纯』的函数是没有状态的，加入了闭包以后就变成有状态的了，相对于一个有成员变量的类实例来说，闭包中的状态值不是自己管理，可以认为是『上帝』在管理。

- 动态作用域，词法作用域
- 把数据和作用域绑定到一起就是闭包。
- 将一个上下文的私有变量的生命周期延长的机制。
- 调用了局部变量的函数就是闭包！

引用环境

按照命令式语言的规则，ExFunc函数只是返回了内嵌函数InsFunc的地址，在执行InsFunc函数时将会由于在其作用域内找不到sum变量而出错。而在函数式语言中，当内嵌函数体内引用到体外的变量时，将会把定义时涉及到的引用环境和函数体打包成一个整体（闭包）返回。现在给出引用环境的定义就容易理解了：引用环境是指在程序执行中的某个点所有处于活跃状态的约束（一个变量的名字和其所代表的对象之间的联系）所组成的集合。闭包的使用和正常的函数调用没有区别。

由于闭包把函数和运行时的引用环境打包成为一个新的整体，所以就解决了函数编程中的嵌套所引发的问题。如上述代码段中，当每次调用ExFunc函数时都将返回一个新的闭包实例，这些实例之间是隔离的，分别包含调用时不同的引用环境现场。不同于函数，闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

```

val string = "HelloWorld"
// 闭包：函数的运行环境
// 函数的作用域没有被释放，作用域包含了函数运行的状态、变量、本地类、本地函数，这个作用域就是闭包
fun makeFun(): ()->Unit // ()->Unit 函数是 makeFun() 的返回值，即函数的返回值是一个函数
// makeFun()的函数体
{
    var count = 0
    return fun(){
        println(++count)
    }
}

fun fibonacci(): Iterable<Long>{
    var first = 0L
    var second = 1L
    return Iterable {
        object : LongIterator(){
            override fun nextLong(): Long {
                val result = second
                second += first
                first = second - first
                return result
            }
        }
    }
}

```

```
        override fun hasNext() = true

    }

}

fun main(args: Array<String>) {
    val add5 = add(5)
    println(add5(2))
}

//fun add(x: Int) = fun(y: Int) = x + y

fun add(x: Int): (Int)-> Int{
    data class Person(val name: String, val age: Int)

    return fun(y: Int): Int{
        return x + y
    }
}
```

函数与闭包

函数与闭包的特性可以算是 Kotlin 语言最大的特性了。

1. 函数

即使 Kotlin 是一门面向对象的编程语言，它也是有函数的概念的——而不像 Java 那样，仅仅有“方法”。回顾一下前面第二章讲述的函数声明语法：

```
fun say(str: String): String {  
    return str  
}
```

函数使用关键字 `fun` 声明，如下代码创建了一个名为 `say()` 的函数，它接受一个 `String` 类型的参数，并返回一个 `String` 类型的值。

1.1 Unit

如果一个函数是空函数，比如 Android 开发中的 `TextWatcher` 接口，通常只会用到一个方法，但必须把所有方法都重写一遍，就可以通过这种方式来简写：

```
editText.addTextChangedListener(object : TextWatcher {  
    override fun afterTextChanged(s: Editable?) = Unit  
  
    override fun beforeTextChanged(s: CharSequence?, start: Int,  
        count: Int, after: Int) = Unit  
  
    override fun onTextChanged(s: CharSequence?, start: Int, bef  
        ore: Int, count: Int) = Unit  
})
```

`Unit` 表示的是一个值的类型。这种类型类似于 Java 中的 `void` 类型。

1.2 Nothing

Nothing 也是一个值的类型。如果一个函数不会返回（也就是说只要调用这个函数，那么在它返回之前程序肯定出错了，比如一定会抛出异常的函数），理论上你可以随便给他一个返回值，通常我们会声明为返回 Nothing 类型。我们看到 Nothing 的注释：

```
/**  
 * Nothing has no instances. You can use Nothing to represent "a  
 * value that never exists": for example,  
 * if a function has the return type of Nothing, it means that i  
 t never returns (always throws an exception).  
 */  
public class Nothing private constructor()
```

没有任何实例。你可以使用 Nothing 来表示“永远不存在的值”。

2. 复杂的特性

2.1 嵌套函数

Kotlin 的函数有一些非常有意思的特性，比如函数中再声明函数。

```
fun function() {  
    val str = "hello!"  
  
    fun say(count: Int = 10) {  
        println(str)  
        if (count > 0) {  
            say(count - 1)  
        }  
    }  
    say()  
}
```

与内部类有些类似，内部函数可以直接访问外部函数的局部变量、常量，而外部函数同级的其他函数不能访问到内部函数。这种写法通常使用在 会在某些条件下触发递归的方法内 或者是 不希望外部其他函数访问到的函数，在一般情况下是不推荐使用嵌套函数的。

2.2 运算符重载

```
fun main(args: Array<String>) {
    for (i in 1..100 step 20) {
        print("$i ")
    }
}
```

这段函数将会输出 1 21 41 61 81

这段神奇的循环是怎么执行的？

in关键字，在编译以后，会被翻译为一个迭代器方法，其源码可以在 Progressions 类中查看到

```

override fun iterator(): IntIterator = IntProgressionIterator(first, last, step)

/**
 * An iterator over a progression of values of type `Int`.
 */
internal class IntProgressionIterator(first: Int, last: Int, val
    step: Int) : IntIterator() {
    private var next = first
    private val finalElement = last
    private var hasNext: Boolean = if (step > 0) first <= last e
    lse first >= last

    override fun hasNext(): Boolean = hasNext

    override fun nextInt(): Int {
        val value = next
        if (value == finalElement) {
            hasNext = false
        }
        else {
            next += step
        }
        return value
    }
}

```

in 关键字之后，还有两个点的 .. ，他表示一个封闭区间，其内部实现原理是通过运算符重载来完成的。本质上是一个函数，首先看到他的函数定义，你可以在 Int 类的源码中找到：

```

/** Creates a range from this value to the specified [other] va
lue. */
public operator fun rangeTo(other: Int): IntRange

```

运算符重载需要使用关键字 operator 修饰，其余定义与函数相同。通过源码看到，上面的代码实际 .. 的原理实际上就是对一个 Int 值，调用他的 rangeTo 方法，传入一个 Int 参数，并返回一个区间对象。带入到上面的代码，实际上就是

把`..`看做是方法名，调用`1`的`rangeTo`方法，传入`100`作为参数，会返回一个区间对象。然后再用迭代器`in`便利区间中的每一个值。所以上面那种写法改写为下面这样，依旧是能正常运行的。

```
for (i in 1.rangeTo(100) step 20) {
    print("$i ")
}
```

说道运算符给大家讲个笑话，在C/C++/Java中，其实有一个大家经常使用但是没有人知道的运算符，叫趋近于写法为`-->`，例如下面的代码：`int i = 10; while(i --> 0){ printf("%d", i); }`这个代码运行完后将会依次打印10到0数字。不信你试试

2.3 中缀表达式

运算符的数量毕竟是有限的，有时并不一定有合适的。例如上面代码中的步长这个意义，就没有合适的运算符可以标识。这时候我们可以用一个单词或字母来当运算符用(其本质还是函数调用)，叫做中缀表达式，所谓中缀表达式就是不需要点和括号的方法调用。

你可以在Reangs中看到`step`源码声明：

```
public infix fun IntProgression.step(step: Int): IntProgression {
    checkStepIsPositive(step > 0, step)
    return IntProgression.fromClosedRange(first, last, if (this.
    step > 0) step else -step)
}
```

中缀表达式需要用`infix`修饰，从源码看到，在SDK中定义了一个叫`step`的方法，最终返回一个`IntProgression`对象，这个对象最终会被作用到`in`，也就是迭代器的第三个参数`step`上。

3. 闭包

其实在Kotlin中与其说一等公民是函数，不如说一等公民是闭包。

例如在 Kotlin 中，你可以写出这种怪异的代码

```
fun main(args: Array<String>) {
    test
}
val test = if (5 > 3) {
    print("yes")
} else {
    print("no")
}
```

当然，我们都知道这段代码永远都只会输出yes。

这里只是为了演示，if语句仍旧是一个闭包。而事实上，上文包括前文讲到的所有：函数、Lambda、if语句、for、when，都可以称之为闭包，但通常情况下，我们所说的闭包是Lambda表达式。

2.1 自执行闭包

自执行闭包就是在定义闭包的同时直接执行闭包，一般用于初始化上下文环境。例如：

```
{ x: Int, y: Int ->
    println("${x + y}")
}(1, 3)
```

4. Lambda

4.1 Lambda 表达式

Lambda 表达式俗称匿名函数，熟悉Java的大家应该也明白这是个什么概念。Kotlin 的 Lambda表达式更“纯粹”一点，因为它是真正把Lambda抽象为了一种类型，而 Java 8 的 Lambda 只是单方法匿名接口实现的语法糖罢了。

```

val printMsg = { msg: String ->
    println(msg)
}

fun main(args: Array<String>) {
    printMsg.invoke("hello")
}

```

以上是 Lambda 表达式最简单的实例。

首先声明了一个名为 `printMsg` 的 Lambda，它接受一个 `String` 类型的值作为参数，然后在 `main` 函数中调用它。如果还想省略，你还可以在调用时直接省略 `invoke`，像函数一样使用。

```

fun main(args: Array<String>) {
    printMsg("hello")
}

```

Lambda 表达式还有非常多的语法糖，比如

- 当参数只有一个的时候，声明中可以不用显示声明参数，在使用参数时可以用 `it` 来替代那个唯一的参数。
- 当有多个用不到的参数时，可以用下划线来替代参数名(1.1以后的特性)，但是如果已经用下划线来省略参数时，是不能使用 `it` 来替代当前参数的。
- Lambda 最后一条语句的执行结果表示这个 Lambda 的返回值。

需要注意的是：闭包是不能有变长参数的

例如前面讲过变长参数的函数，但是闭包的参数数量是必须固定的。

```

fun printLog(vararg str: String) {
}

```

4.2 高阶函数

Lambda 表达式最大的特点是可以作为参数传递。当定义一个闭包作为参数的函数，称这个函数为高阶函数。

```

fun main(args: Array<String>) {
    log("world", printMsg)
}

val printMsg = { str: String ->
    println(str)
}

val log = { str: String, printLog: (String) -> Unit ->
    printLog(str)
}

```

这个例子中，log 是一个接受一个 String 和一个以 String 为参数并返回 Unit 的 Lambda 表达式为参数的 Lambda 表达式。

读起来有点绕口，其实就是 log 有两个参数，一个str:String，一个printLog: (String) -> Unit。

4.3 内联函数

在使用高阶函数时，一定要知道内联函数这个东西。它可以大幅提升高阶函数的性能。

官方文档的描述是这样的：使用 高阶函数 在运行时会带来一些不利：每个函数都是一个对象，而且它还要捕获一个闭包，也就是，在函数体内部访问的那些外层变量。内存占用(函数对象和类都会占用内存)以及虚方法调用都会带来运行时的消耗。

但是也不是说所有的函数都要内联，因为一旦添加了 `inline` 修饰，在编译阶段，编译器将会把函数拆分，插入到调用处。如果一个 `inline` 函数是很大的，那他会大幅增加调用它的那个函数的体积。

5. 小结

闭包应该算是 Kotlin 最核心特性之一了。

使用好闭包可以让代码量大大减少，例如 Kotlin 最著名的开源库：`Anko`，使用 Anko 去动态代码布局，比使用 Java 代码配合 xml 要更加简洁。

```
class MyActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?, persistentState: PersistableBundle?) {  
        super.onCreate(savedInstanceState, persistentState)  
        MyActivityUI().setContentView(this)  
    }  
}  
  
class MyActivityUI : AnkoComponent<MyActivity> {  
    override fun createView(ui: AnkoContext<MyActivity>) = ui.aply {  
        verticalLayout {  
            editText()  
            button("Say Hello") {  
                onClick { ctx.toast("Hello!") }  
            }  
        }  
    }.view  
}
```

可以看到，充分运用了闭包的灵活性，省略了很多的临时变量和参数声明。然而，也正是因为闭包的灵活性，造成如果泛滥的话，可能会写出可读性非常差的代码(这里就不举反例了，js 的 lambda 滥用的结果就能想象了)

细说 Lambda 表达式

1. 什么是 Lambda 表达式

Lambda 表达式，其实就是匿名函数。而函数其实也是功能（function），匿名函数，就是匿名的功能代码了。在 Kotlin 当中，函数也是作为类型的一种出现的，尽管在当前的版本中，函数类型的灵活性还不如 Python 这样的语言，不过它也是可以被赋值和传递的，这主要就体现在 Lambda 表达式上。

我们先来看一个 Lambda 表达式的例子：

```
fun main(args: Array<String>) {
    val lambda = {
        left: Int, right: Int
        ->
        left + right
    }
    println(lambda(2, 3))
}
```

大家可以看到我们定义了一个变量 lambda，赋值为一个 Lambda 表达式。Lambda 表达式用一对大括号括起来，后面先依次写下参数及其类型，如果没有就不写，接着写下 `->`，这表明后面的是函数体了，函数体的最后一句的表达式结果就是 Lambda 表达式的返回值，比如这里的返回值就是参数求和的结果。

后面我们用 `()` 的形式调用这个 Lambda 表达式，其实这个 `()` 对应的是 `invoke` 方法，换句话说，我们在这里也可以这么写：

```
println(lambda.invoke(2, 3))
```

这两种调用的写法是完全等价的。

毫无疑问，这段代码的输出应该是 5。

2. 简化 Lambda 表达式

我们再来看个例子：

```
fun main(args: Array<String>) {
    args.forEach {
        if(it == "q") return
        println(it)
    }
    println("The End")
}
```

args 是一个数组，我们已经见过 for 循环迭代数组的例子，不过我们其实有更现代化的手段来迭代一个数组，比如上面这个例子。这没什么可怕的，一旦撕下它的面具，你就会发现你早就认识它了：

```
public inline fun <T> forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

这是一个扩展方法，扩展方法很容易理解，原有类没有这个方法，我们在外部给它扩展一个新的方法，这个新的方法就是扩展方法。大家都把它当做 Array 自己定义的方法就好，我们看到里面其实就是一个 for 循环对吧，for 循环干了什么呢？调用了我们传入的Lambda表达式，并传入了每个元素作为参数。所以我们调用 forEach 方法时应该怎么写呢？

```
args.forEach({
    element -> println(element)
})
```

这相当于什么呢？

```
for(element in args){
    println(element)
}
```

很容易理解吧？

接着，Kotlin 允许我们把函数的最后一个Lambda表达式参数移除括号外，也就是说，我们可以改下上面的 forEach 的写法：

```
args.forEach{  
    element -> println(element)  
}
```

看上去有点儿像函数定义了，不过区别还是很明显的。这时候千万不能晕了，晕了的话我这儿有晕车药吃点儿吧。

事儿还没完呢，如果函数只有这么一个 Lambda 表达式参数，前面那个不就是么，剩下一个小括号也没什么用，干脆也丢掉吧：

```
args.forEach{  
    element -> println(element)  
}
```

大家还好吧？你以为这就结束了？nonono，如果传入的这个Lambda表达式只有一个参数，还是比如上面这位 forEach，参数只有一个 element，于是我们也可以在调用的时候省略他，并且默认它叫 it，说得好有道理，它不就是 it 么，虽然人家其实是 iterator 的意思：

```
args.forEach{  
    println(it)  
}
```

嗯，差不多了。完了没，没有。还有完没啊？就剩这一个了。如果这个 Lambda 表达式里面只有一个函数调用，并且这个函数的参数也是这个Lambda表达式的参数，那么你还可以用函数引用的方式简化上面的代码：

```
args.forEach(::println)
```

这有没有点儿像 C 里面的函数指针？函数也是对象嘛，没什么大惊小怪的，只要实参比如 println 的入参和返回值与形参要求一致，那么就可以这么简化。

总结一下：

1. 最后一个Lambda可以移出去
2. 只有一个Lambda，小括号可省略
3. Lambda 只有一个参数可默认为 it
4. 入参、返回值与形参一致的函数可以用函数引用的方式作为实参传入

这样我们之前给的那个例子就大致能够看懂了吧：

```
fun main(args: Array<String>) {  
    args.forEach {  
        if(it == "q") return  
        println(it)  
    }  
    println("The End")  
}
```

3. 从 Lambda 中返回

真看懂了吗？假设我输入的参数是

```
o p q r s t
```

你知道输出什么吗？

```
o  
p  
The End
```

对吗？

不对，return 会直接结束 main 函数。为啥？Lambda 表达式，是个表达式啊，虽然看上去像函数，功能上也像函数，可它看起来也不过是个代码块罢了。这就像琅琊榜前期，靖王虽然获得了自由进宫拜见母妃的特权，但他当时并不是亲王，而只是一个郡王一样。

那，就没办法 return 了吗？当然不是，兵来将挡水来土掩：

```

fun main(args: Array<String>) {
    args.forEach forEachBlock@{
        if(it == "q") return@forEachBlock
        println(it)
    }
    println("The End")
}

```

定义一个标签就可以了。你还可以在 `return@forEachBlock` 后面加上你的返回值，如果需要的话。

4. Lambda 表达式的类型

好，前面说到 Lambda 表达式其实是函数类型，我们在前面的 `forEach` 方法中传入的 Lambda 表达式其实就是 `forEach` 方法的一个参数，我们再来看下 `forEach` 的定义：

```

public inline fun <T> Array<out T>.forEach(action: (T) -> Unit):
    Unit {
    for (element in this) action(element)
}

```

注意到，`action` 这个形参的类型是 `(T) -> Unit`，这个是 Lambda 表达式的类型，或者说函数的类型，它表示这个函数接受一个 `T` 类型的参数，返回一个 `Unit` 类型的结果。我们再来看几个例子：

```

() -> Int //无参，返回 Int
(Int, Int) -> String //两个整型参数，返回字符串类型
((()) -> Unit, Int) -> Unit //传入了一个 Lambda 表达式和一个整型，返回 Unit

```

我们平时就用这样的形式来表示 Lambda 表达式的类型的。有人可能会说，既然人家都是类型了，怎么就没有个名字呢？或者说，它对应的是哪个类呢？

```
public interface Function<out R>
```

其实所有的 Lambda 表达式都是 Function 的实现，这时候如果你问我，那 invoke 方法呢？在哪儿定义的？说出来你还真别觉得搞笑，Kotlin 的开发人员给我们定义了 23 个 Function 的子接口，其中 FunctionN 表示 invoke 方法有 n 个参数。。

```
public interface Function0<out R> : Function<R> {
    public operator fun invoke(): R
}

public interface Function1<in P1, out R> : Function<R> {
    public operator fun invoke(p1: P1): R
}

...

public interface Function22<in P1, in P2, in P3, in P4, in P5, in P6, in P7, in P8, in P9, in P10, in P11, in P12, in P13, in P14, in P15, in P16, in P17, in P18, in P19, in P20, in P21, in P22, out R> : Function<R> {
    public operator fun invoke(p1: P1, p2: P2, p3: P3, p4: P4, p5: P5, p6: P6, p7: P7, p8: P8, p9: P9, p10: P10, p11: P11, p12: P12, p13: P13, p14: P14, p15: P15, p16: P16, p17: P17, p18: P18, p19: P19, p20: P20, p21: P21, p22: P22): R
}
```

说实在的，第一看到这个的时候，我直接笑喷了，Kotlin 的开发人员还真是黑色幽默啊。

这事儿不能这么完了，万一我真有一个函数，参数超过了 22 个，难道 Kotlin 就不支持了吗？

```
fun hello2(action: (Int, Int, Int) -> Unit) {
    action(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22)
}
```

于是我们定义一个参数有 23 个的 Lambda 表达式，调用方法也比较粗暴：

```

hello2 { i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12,
i13, i14, i15, i16, i17, i18, i19, i20, i21, i22 ->
    println("$i0, $i1, $i2, $i3, $i4, $i5, $i6, $i7, $i8, $i9, $i10, $i11, $i12, $i13, $i14, $i15, $i16, $i17, $i18, $i19, $i20, $i21, $i22,")
}

```

编译运行结果：

```

Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/Function23
    at java.lang.Class.getDeclaredMethods0(Native Method)
    at java.lang.Class.privateGetDeclaredMethods(Class.java:2701)
)
    at java.lang.Class.privateGetMethodRecursive(Class.java:3048)
)
    at java.lang.Class.getMethod0(Class.java:3018)

```

果然，虽然这个参数有 23 个的 Lambda 表达式被映射成 kotlin/Function23 这个类，不过，这个类却不存在，也就是说，对于超过 22 个参数的 Lambda 表达式，Kotlin 代码可以编译通过，但会抛运行时异常。这当然也不是个什么事儿了，毕竟有谁脑残到参数需要 22 个以上呢？

5. SAM 转换

看名字挺高大上，用起来炒鸡简单的东西你估计见了不少，这样的东西你可千万不要回避，多学会一个就能多一样拿出去唬人。

```

val worker = Executors.newCachedThreadPool()

worker.execute {
    println("Hello")
}

```

本来我们应该传入一个 Runnable 的实例的，结果用一个 Lambda 表达式糊弄过去，Java 怎么看？

```
GETSTATIC net/println/MainKt$main$1.INSTANCE : Lnet/println/MainKt$main$1;
CHECKCAST java/lang/Runnable
INVOKEINTERFACE java/util/concurrent/ExecutorService.execute (Ljava/lang/Runnable;)V
```

Java 说介叫嘛事儿，介不就一 Lambda 么，转成 Runnable 在拿过来！

你看上面的这三句字节码，第一句拿到了一个类的实例，这个类一看就是一个匿名内部类：

```
final class net/println/MainKt$main$1 implements java/lang/Runnable {
    ...
}
```

这是这个类定义的字节码部分，实现了 Runnable 接口的一个类！

第二句，拿到这个类的实例以后做强转——还转啥，直接拿来用呗，肯定没问题呀。

那你说 SAM 转换有什么条件呢？

- 首先，调用者在 Kotlin 当中，被调用者是 Java 代码。如果前面的例子当中 worker.execute(...) 是定义在 Kotlin 中方法，那么我们是不能用 SAM 转换的。
- 其次，参数必须是 Java 接口，也就是说，Kotlin 接口和抽象类、Java 抽象类都不可以。
- 再次，参数的 Java 接口必须只有一个方法。

我们再来举个 Android 中常见的例子：

```
view.setOnClickListener{
    view ->
    ...
}
```

view.setOnClickListener(...) 是 Java 方法，参数 OnClickListener 是 Java 接口，并且只有一个方法：

```
public interface OnClickListener {  
    void onClick(View v);  
}
```

6. 小结

Lambda 表达式就是这么简单，简单的让人有点儿害怕。不知道大家有没有过这样的感觉，越是简单的东西用起来越复杂，不相信你回去翻一翻高中物理课本的牛顿第二定律。Lambda 表达式就是这样的东西，它能够极大的简化代码的书写，尽管一旦有了 Lambda 表达式的掺和，代码本身理解起来可就要稍微困难一些了，不过，因噎废食的事情想必大家也是不喜欢做的，对吧？



高阶函数（一）

1 什么是高阶函数

1.1 高阶函数的基本概念

高阶函数其实看着挺吓人，不过就是把函数作为参数或者返回值的一类函数而已。其实这样的函数我们都见过很多了，来看个例子：

```
public inline fun <T> Array<out T>.forEach(action: (T) -> Unit):  
    Unit {  
        for (element in this) action(element)  
    }
```

这是我们的老朋友了，`forEach`，传入了一个 Lambda 表达式，之后在迭代数组的时候调用这个 Lambda。你可千万别把 Lambda 不当函数，人家可是正儿八经的 `FunctionN` 的实例，这个我们在前面一篇文章[细说 Lambda 表达式](#)已经介绍过了

如果使用 `forEach`，我们就这么写：

```
array.forEach{  
    print("$it, ")  
}
```

输出结果就类似：

```
1, 2, 3, 4,
```

`forEach` 其实就是一个把函数当参数传入的高阶函数了。

1.2 函数引用

下面我们要来思考一个问题。为什么 Kotlin 可以有高阶函数？

其实这个问题，我们早就有答案了，因为在 Kotlin 当中，函数是“一等公民”，函数的引用可以自由传递，赋值，并在合适的时候调用。为什么这么说呢？难道仅仅是因为我们可以任性的定义 Lambda 表达式这种匿名函数，并把它赋值为一个变量，然后可以随便传递和调用吗？

当然不能完全是这样了。其实 Kotlin 当中的任何方法、函数都是有其名字和引用的，我们前面其实看到过一个 forEach 的例子，我再给大家拿出来：

```
array.forEach(::println)
```

这个例子当中，我们其实是想要把元素挨个打印一遍，forEach 传入的是一个 $(T) \rightarrow Unit$ ，这并不是说它只能传入一个符合参数和返回值的 Lambda，而是说符合参数和返回值定义的任意函数。println 有很多版本，其中有一个符合上面的条件：

```
public inline fun println(message: Any?) {
    System.out.println(message)
}
```

所以我们可以把它当参数传入。这个意义上讲，`::println` 跟 `Function1` 是什么关系呢？很明显接口实现的关系了，同时 `::println` 因为可以具名引用到一个函数，所以我们也把类似的写法叫做函数引用。

我们再来看一个类成员的例子：

```
class Hello{
    fun world(){
        println("Hello world.")
    }
}

val helloWorld: (Hello) -> Unit = Hello::world
```

我们同样可以用 `<TypeName>::<FunctionName>` 的方式来引用类成员方法，当然扩展方法也是可以的。这个要怎么用呢？

```
fun Int.isOdd(): Boolean = this % 1 == 0

...
val ints = intArrayOf(1, 3, 4, 5, 8)
ints.filter(Int::isOdd)
```

注意到 filter 的参数类型：

```
public inline fun IntArray.filter(predicate: (Int) -> Boolean):
List<Int> {
    return filterTo(ArrayList<Int>(), predicate)
}
```

跟我们前面 Hello::World 的例子是不是一模一样呢？

不过相比包级函数，这种引用在 Kotlin 1.1 以前显得有些苍白，为什么这么说呢？

```
class PdfPrinter{
    fun println(any: Any?){
        println(any)
    }
}

...
array.forEach(PdfPrinter::println) //错误！！
```

请问，这种情况下，我该如何像 `::println` 一样将 `PdfPrinter::println` 传递给 `forEach` 呢？我们知道，所有的类成员方法，它们其实都有一个隐含的参数，即类的实例本身，所以它的类型应该是下面这样：

```
val pdfPrintln: (PdfPrinter, Any?) -> Unit = PdfPrinter::println
```

那么，有人就会说，我干脆构造一个 `PdfPrinter` 的实例，然后这么写看看：

```
array.forEach(PdfPrinter()::println)// Since Kotlin 1.1
```

看着很不错了吧？可惜，这个在 1.1 才支持哦，不过距离 1.1 正式发布应该不久了！

2 常见的内置高阶函数

Kotlin 为我们内置了不少好用的高阶函数，这一节我们就给大家简要介绍一下。

2.1 map

我们经常用 `forEach` 来迭代一个集合，如果我们想要把一个集合映射成另外一个集合的话，通常我们会这么写：

```
val list = listOf(1, 3, 4, 5, 10, 6, 8, 2)

val newlist = ArrayList<Int>()
list.forEach {
    val newElement = it * 2 + 3
    newlist.add(newElement)
}
```

看上去还是挺简单的，不过终究不够简洁，而且还在 Lambda 表达式内部访问了外部变量，这其实都不是很好的编程习惯。

`map` 其实就是对类似的操作做了一点封装，类似的集合映射的操作用 `map` 再合适不过了：

```
val newlist = list.map {
    it * 2 + 3
}
```

Lambda 的参数是原集合的元素，返回值是对应位置的新集合的元素，新集合是 `map` 的返回值。我们再来看个例子：

```
val stringlist = list.map(Int::toString)
```

上面这个例子，我们把一个整型的集合映射成了一个字符串类型的集合。不管你做何种变换，map 的返回值始终是一个大小与原集合相同的集合。

2.2 flatMap

如果我手头有一个整型集合的集合，我想把他们打平，变成一个整型集合，用我们传统的方法就是两层循环。如果我还想要做点儿变换，那么这代码写起来就更丑了。

如果我们要用 flatMap，那么这个故事就直截了当得多：

```
val list = listOf(  
    1..20,  
    2..5,  
    100..232  
)  
  
val flatList = list.flatMap { it }  
println(flatList)
```

flatMap 后面的 Lambda 参数是 list 的元素，也就 1..20、2..5 这些 range，返回的值呢是一个 Iterable，flatMap 会把这些 Lambda 返回的 Iterable 统统添加到它自己的返回值也就是 flatList 当中，这样就相当于把 list 做了一次打平。

结果：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 2, 3, 4, 5, 100, 101, 102, 103, 104, 105, 106, 107, 108,
109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121,
122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134,
135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147,
148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160,
161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173,
174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186,
187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199,
200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212,
213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225,
226, 227, 228, 229, 230, 231, 232]
```

那么这么直白的打平也不见得是我们的目标，比如我们要把这些数都做一些运算再打平，那么这个也简单：

```
val flatList = list.flatMap { iterable ->
    iterable.map { element ->
        element * 2 + 3
    }
}
```

我们只需要对 iterable 做一次 map 即可。

2.3 fold / reduce

其实 fold 就是折叠的意思嘛，把一个集合的元素折叠起来并得到一个最终的结果，这就是 fold 要做的事情。

```
fun main(args: Array<String>) {
    val ints = intArrayOf(1, 2, 3, 4, 5)
    val r = ints.fold(5){
        sum, element ->
        println("$sum, $element")
        sum + element
    }
    println(r)
}
```

结果呢？就是从 5 开始，每次返回的结果又成了 sum，一直这么折叠下去，直到最后输出 20。

```
5, 1
6, 2
8, 3
11, 4
15, 5
20
```

当然，对于fold来说，我们还可以得到其他类型的结果，不一定要与集合的元素类型相同：

```
val r1 = ints.fold(StringBuilder()){  
    sb, element->  
    sb.append(element).append(",")  
}  
  
println(r1)
```

大家看到，我们的初始值实际上是一个 StringBuilder，后续一直在做字符串追加的操作，最后得到的 r1 其实就是一个追加了所有元素的 StringBuilder，我们把它打印出来：

```
1, 2, 3, 4, 5,
```

我们再来看下 reduce。

```
val r2 = ints.reduce { sum, element -> sum + element }
println(r2)
```

输出的最终结果是 15，也即元素之和。显然，reduce 每次求值的结果都作为下一次迭代时传入的 sum，这个看上去跟 fold 极其的类似，只不过 reduce 没有额外的初始值，并且返回值类型也需要保持与集合的元素相同。

如果我们要求一个数的阶乘，那代码其实很容易写：

```
fun factorial(n: Int): Int{
    if(n == 0) return 1
    return (1..n).reduce { factorial, element -> factorial * element }
}
```

2.4 filter / takeWhile

如果我们有一个很大的集合，想要过滤掉其中的一些元素，那么通常的做法也是构造一个新集合来，然后遍历原集合。

显然，我们有更好的写法：

```
val evens = (1..100).filter {
    it % 2 == 0
}
```

找出 1 到 100 之间的所有偶数，我们只需要用 filter，并传入判断条件，那么符合条件的元素就会被保留到返回的集合当中。

类似的，takeWhile 则返回的集合是原集合中从第一个元素开始到第一个不符合条件的元素之前的所有元素。例如：

```
println((1..10).takeWhile { it % 5 != 0 })
```

这表明，从 1..10 中取元素，只要遇到一个是 5 的倍数的元素，那么立即返回，即结果为：

```
[1, 2, 3, 4]
```

2.5 let

let 实际上比较简单，我们先来看下它的定义：

```
public inline fun <T, R> T.let(block: (T) -> R): R = block(this)
```

我们看到 let 实际上传入了一个 Lambda，而这个 Lambda 传入的参数就是 let 的调用者本身，返回值随便你。这个 let 有什么用呢？

```
val person: Person? = findPerson()
```

我们看到 person 这个变量是可空的，我们需要做一些判断才能对其进行操作。通常的写法可能是这样的：

```
person?.name = "张三"  
person?.age = 18  
...
```

不过，这种问号满天飞的写法，看着其实并不是很让人舒服。

我们还可以这么写：

```
if(person != null){  
    person.name = "张三" // person 被智能转换成 Person 类型  
    person.age = 18  
    ...  
}
```

当然，我们还有一种写法就是：

```
person?.let{  
    it.name = "张三"  
    it.age = 18  
    ...  
}
```

let 比较简单，其用法也是很灵活的，大家可以自行发挥。

2.6 apply / with

下面我们来看 apply。

```
public inline fun <T> T.apply(block: T.() -> Unit): T { block();  
    return this }
```

注意到 apply 传入的 Lambda 也是 apply 的调用者的扩展方法，所以，apply 相当于给了我们一个灵活切换上下文的机会，

```
class Options{  
    var scale: Float = 1f  
    var offsetX: Double = 0.0  
    var offsetY: Double = 0.0  
    var rotationX: Float = 0f  
    var rotationY: Float = 0f  
}
```

假设我们有这么一个类，我们在操作一个地图变换的时候需要传入这个东西，告诉地图该怎么变换。

```
mapView.animateChange(Options().apply {  
    //Options 的作用域  
    scale = 2f  
    rotationX = 180f  
})
```

而 with 呢？

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R
= receiver.block()
```

跟 apply 比较类似，不同之处在于与 Lambda 返回值。with 只是单纯的获取 receiver 的上下文，而 apply 则同时也把它本身返回了。

```
val br = BufferedReader(FileReader("hello.txt"))
with(br){
    var line: String?
    while (true){
        line = readLine(): break
    }
    close()
}
```

我们看到在 with 当中，readLine 和 close 方法可以直接调用。

内置高阶函数其实非常多，这几个比较常用，剩下的大家可以自行学习。

3 尾递归优化

递归大家都熟悉，一说递归大家都容易哆嗦：你可别递归的层次太深了啊，不然小心 StackOverflow！没错，StackOverflow 这个异常实在是太常见了，所以你最熟悉的程序员社交网站当中就有一个叫 StackOverflow 的。

其实大家肯定也都知道，递归能实现的，用迭代也基本上能实现，这个感觉就好像做小学数学题，用递归就好比设个 x ，列个方程求解；而用迭代呢，就好比用算式生生的去把结果给算出来。前者思考起来比较直接，编写起来也自然更符合人的思维模式，后者呢往往编写困难，代码可读性差。

如果有一天，我能写出递归程序，编译器呢，却能够按照迭代的方式给我运行，那么我岂不是既能获得递归的简洁性，又不失迭代的运行效率，那该。。。想得美啊。

我们今天就要给大家看一种特定条件下的编译优化措施。其实前面我们的设想并不是完全做不到，对于某些比较简单的场景，编译器是可以直接把我们的递归代码翻译成迭代代码的，而这种场景其实就是“尾递归”。

什么叫“尾递归”？函数在调用自己之后，没有任何操作的情形就是尾递归。

比如：

```
data class ListNode(val value: Int, var next: ListNode?)  
  
fun findListNode(head: ListNode?, value: Int): ListNode?{  
    head?: return null  
    if(head.value == value) return head  
    return findListNode(head.next, value)  
}
```

我们随便定义了一个链表，ListNode 是它的元素，findListNode 目的是找到对应值的元素。我们看到最后一行只有 findListNode 的调用，没有其他任何操作，这就是尾递归。

我们再来看几个例子：

```
fun factorial(n: Long): Long{  
    return n * factorial(n - 1)  
}
```

求阶乘，因为 factorial 调用之后还有乘以 n 的操作，所以这个不是尾递归。

```
data class TreeNode(val value: Int){  
    var left: TreeNode? = null  
    var right: TreeNode? = null  
}  
  
fun findTreeNode(root: TreeNode?, value: Int): TreeNode?{  
    root?: return null  
    if(root.value == value) return root  
    return findTreeNode(root.left, value) ?: findTreeNode  
(root.right, value)  
}
```

这个算不算尾递归呢？好像最后一行的两个 return 都是之调用了 findTreeNode，没有其他操作了啊，这个应该是尾递归吧？答案当然不是。。因为第一个 findTreeNode 的结果拿到之后，我们要看下他是不是为 null，实际上这个判断操作在 findTreeNode 之后，所以不能算尾递归。对于不是尾递归的情况，编译器是没有办法做优化的。

而对于尾递归的情况，我们该如何启用编译器优化呢？

要说告诉编译器需要尾递归优化，其实非常简单，加一个关键字即可：

```
tailrec fun findListNode(head: ListNode?, value: Int): ListNode?  
{  
    head?: return null  
    if(head.value == value) return head  
    return findListNode(head.next, value)  
}
```

这个看起来真的很简单，简单到没有说服力。我们看一段小程序：

```
val MAX_NODE_COUNT = 100000
val head = ListNode(0)
var p = head
for (i in 1..MAX_NODE_COUNT){
    p.next = ListNode(i)
    p = p.next!!
}
//前面先构造了一个链表，节点个数有 10 万个
//后面进行查找，查找值为 MAX_NODE_COUNT - 2 的节点
println(findListNode(head, MAX_NODE_COUNT - 2)?.value)
```

对于没有 tailrec 关键字的版本，结果非常抱歉：

```
Exception in thread "main" java.lang.StackOverflowError
    at net.println.kotlin.RecursiveKt.findListNode(Recursive.kt:34)
    at net.println.kotlin.RecursiveKt.findListNode(Recursive.kt:34)
```

而对于有 tailrec 的版本，结果是：

```
99998
```

显然，对于尾递归优化的版本，即使你递归再多的层次，都不会有 StackOverflow，原因也很简单，编译器其实已经把这种递归编译成迭代来运行了，迭代怎么会有 StackOverflow 呢？

接着我们再来讨论一下非尾递归代码可以改写为尾递归代码的条件。大家仔细观察我们前面给出的两个例子，一个是求阶乘，一个是超找树的节点。二者最后一句：

```
return n * factorial(n - 1)
```

```
return findTreeNode(root.left, value) ?: return findTreeNode(roo
t.right, value)
```

虽然都不是尾递归，但还是有差异的。前者在调用完自己之后进行了跟调用自己无关的运算；后者调用完一次自己之后，还有可能调用一次自己。注意，如果调用完自己，又进行了其他操作，也即没有再次调用自己，那么这种递归其实有希望转换为尾递归代码，下面我们就改写一下求阶乘的代码，让它变成尾递归代码。

```
fun factorial(n: Long): Long{
    class Result(var value: Long)

    tailrec fun factorial0(n: Long, result: Result){
        if(n > 0) {
            result.value *= n
            factorial0(n - 1, result)
        }
    }
    val result = Result(1)
    factorial0(n, result)
    return result.value
}
```

这个例子当中有一些比较有意思的概念哈，我们在一个函数当中定义了一个函数和一个类，它们被称作“本地函数”和“本地类”，由于定义在函数内部，因此在外部无法使用它们。接着我们对内部的 factorial0 加了 tailrec 关键字，由于最后一行只有对自己的调用，因此符合尾递归优化的条件。

我们看到，之前 $n * \dots$ 的这部分操作通过 Result 携带的中间结果被移到了自身调用的前面，这样做让原本的递归代码符合了尾递归优化的条件，却也让代码本身复杂了许多。而对于此类操作，我个人更倾向于直接使用迭代。

```
fun factorial(n: Long): Long{
    var result: Long = 1
    for (i in 1..n){
        result *= i
    }
    return result
}
```

迭代的代码显然也直截了当得多。

总而言之，使用递归是为了让我们的代码更直接，更自然，使用迭代往往是为了追求效率（空间效率）。对于类似查找链表节点这样的场景，它很自然的就是一个尾递归的结构，我们可以使用尾递归优化来提升它的性能；而对于求阶乘这样的场景，它本来就不是尾递归的结构，我们尽管可以通过某种方式改写它，但这样做其实根本没必要；而对于查找树节点这样的场景，尾递归基本上是无能无力了。

4 闭包

对象是要携带状态的。比如：

```
val string = "HelloWorld"
```

string 这个对象它有值，这个值就是它的状态。那么同样作为对象的函数，它有什么状态呢？我们看个例子：

```
fun makeFun(): () -> Unit {
    var count = 0
    return fun() {
        println(++count)
    }
}

...
val x = makeFun()
x()
x()
x()
x()
```

输出的结果会是什么呢？从函数当中返回一个函数，这在 Java 当中简直不能想象，不过这在函数为“一等公民”的 Groovy、JavaScript 当中确实寻常可见。

1
2
3
4

每次调用 x，打印的值都不一样，这说明函数也是可以保存状态的。受到这个启发，我们是不是可以继续写出这样的例子：

```
fun fibonacci(): ()->Long{
    var first = 0L
    var second = 1L
    return fun(): Long{
        val result = second
        second += first
        first = second - first
        return result
    }
}
...
val next = fibonacci()
for (i in 1..10){
    println(next())
}
```

输出结果：

1
1
2
3
5
8
13
21
34
55

我们干脆再进一步吧：

```
fun fibonacciGenerator(): Iterable<Long>{
    var first = 0L
    var second = 1L
    return Iterable {
        object : LongIterator(){
            override fun hasNext() = true

            override fun nextLong(): Long {
                val result = second
                second += first
                first = second - first
                return result
            }
        }
    }
}

...
for(x in fibonacciGenerator()){
    println(x)
    if(x > 100) break
}
```

这个例子我们干得更彻底，通过返回一个 Iterable，我们甚至可以用 for 循环迭代这个结果。

不管我们怎么写，请注意，每次调用同一个函数的结果都不一样，而承载返回结果的 first 和 second 这两个变量是定义在最外层的函数当中的，按说这个函数一旦运行完毕，它所在的作用域就会被回收，如果真是那样，前面的这两段代码一定是我们产生的幻觉。如果不是幻觉，那只能说明一个问题：这个作用域没有被回收。

这个作用域包含了所有函数运行的状态，包括变量、本地类、本地函数等等，那这个作用域其实就是闭包。

我们再来看个好玩的例子：

```
fun add(x: Int) = fun(y: Int) = x + y  
...  
val add5 = add(5)  
println(add5(2))
```

很显然，结果是 7，这个 add 的定义其实写得有些令人迷惑，我把它改写一下给大家看：

```
fun add(x: Int): (Int)->Int{  
    return fun(y: Int): Int{  
        return x + y  
    }  
}
```

很显然，当我们调用 add(5) 返回 add5 这个函数时，它是持有了 add 函数的运行环境的，不然它怎么知道 x 的值是多少呢？

通过这几个小例子，相信大家对闭包有了一定的了解。闭包其实就是函数运行的环境。

下周我们还会继续跟大家讨论函数编程相关的一些话题，谢谢大家的关注~

“

Kotlin, More Than a Better Java.



Kotlin

长按识别二维码，关注Kotlin

上周我已经给大家推送了一篇关于高阶函数的文章，这一期，我们继续探讨一些相关的有意思的话题。

1. 复合函数

大家一定见过下面的数学题吧：

求 $f(g(x))$ 的值。解：设 $m(x) = f(g(x)) \dots$
 m 就是 f 和 g 的复合。

我们在 Kotlin 当中要如何对函数进行复合呢？

```
val add5 = { i: Int -> i + 5 }
val multiplyBy2 = { i: Int -> i * 2 }
```

我们定义了这么两个函数，接着这么调用它：

```
println(multiplyBy2(add5(2))) // (2 + 5) * 2
```

`add5` 相当于我们的 $g(x)$ ，`multiplyBy2` 相当于 $f(x)$ ，那么上面的式子就相当于 $f(g(x))$ 。下面我们提供一个简单的方式来复合这两个函数，得到 $m(x) = f(g(x))$ ：

```
// f andThen g -> g(f(x))
infix fun <P1, P2, R> Function1<P1, P2>.andThen(function: Function1<P2, R>): Function1<P1, R> {
    return fun(p1: P1): R{
        return function.invoke(this.invoke(p1))
    }
}
```

这里面有几个知识点，我请大家一起复习一下。

第一个就是 `infix`，中缀表达式，有了这个关键字，我们的 `add5` 在调用 `andThen` 方法时，就不需要用 `.andThen()` 的形式了，而是像使用操作符一样。

第二个是扩展方法，`andThen` 其实就是 `Function1<P, R>` 的扩展方法。

第三个则是 Lambda 表达式的类型了，我们在前面提到过，Lambda 表达式有 N (N <= 22) 个参数，那么它的类型就是 FunctionN，这里的 add5 只有一个参数，所以对应于 Function1 类型。

第四个是匿名函数，这个我们前面其实已经见到不少了。

我们看一个例子：

```
val add5AndMultiplyBy2 = add5 andThen multiplyBy2
println(add5AndMultiplyBy2(2))
```

这个例子的输出结果其实与前面的相同， $(2 + 5) * 2 = 14$

通过 andThen，我们看到一个全新的函数 add5AndMultiplyBy2 被创造出来，它其实就是 add5 和 multiplyBy2 的复合。

当然，有时候我们其实还需要这样的结果：

```
val multiplyBy2AndAdd5 = add5 compose multiplyBy2
println(multiplyBy2AndAdd5(2))
```

这个相当于 $2 * 2 + 5 = 9$ 。我们简单看下 compose 的实现：

```
// f compose g -> f(g(x))
infix fun <P1, P2, R> Function1<P2, R>.compose(function: Function1<P1, R>): Function1<P1, R> {
    return fun(p1: P1): R{
        return this.invoke(function.invoke(p1))
    }
}
```

compose 与 andThen 的结果是完全相反的， $f \text{ andThen } g \rightarrow g(f(x))$ ，而 $f \text{ compose } g \rightarrow f(g(x))$ 。

这就是函数复合，其实你在初中数学就学过这些东西了。

2. Currying

Curry 也有咖喱的意思，不过这一节可并不是充满咖喱味的。在函数式编程当中，Currying 也经常被翻译成“科里化”，我们从这个名字完全读不出它究竟是要干嘛。为什么？因为，Curry 是个人名——Haskell Curry。

回到我们的程序当中，我们首先必须要搞清楚什么是 Currying？Currying 其实就是由一个多参数的函数构造出一系列只有一个参数的函数的方法。这么说可能还是有些抽象，我们直接上例子：

$$f(x,y,z) = x * y - z$$

我们有一个三元函数，这个没什么复杂的，你在高中数学当中见到过比这个恐怖的多的式子。接着我们给它做个变式：

$$f(x,y,z) = k_{\{yz\}}(x)$$

其中， $(k_{\{yz\}}(x))$ 是关于 (x) 的一个函数， (yz) 可当做常量看待。而一旦传入 (x) 的值以后，例如 $(k_{\{yz\}}(x_0))$ ，那么此时又有变换：

$$f(x_0, y, z) = k_{\{yz\}}(x_0) = m_{\{z, x=x_0\}}(y)$$

类似的，我们还能最终转换成：

$$f(x_0, y_0, z) = m_{\{z, x=x_0\}}(y_0) = n_{\{x=x_0, y=y_0\}}(z)$$

这么一个数学概念，其实就是 Currying。那么它到底想说明怎样一件事情呢？大家看，参数是一个一个传进来的，这就好比我们完成一件事情，也是对其进行肢解，然后一步一步完成的，通过 Currying，我们可以对一个函数的调用细节进行仔细的考量，甚至像流水线一样处理，以实现我们的目标。

用程序的语言描述，假设我们有一个函数：

```
fun hello(x: String, y: Int, z: Double): Boolean{
    ...
}
```

它 Currying 的结果便是：

```
fun curriedHello(x: String): (y: Int) ->(z: Double)-> Boolean{
    ...
}
```

下面我们给出一个 Kotlin 的例子：

```
fun log(tag: String, target: OutputStream, message: Any?) {  
    ...  
}
```

这是一个日志打印的函数，第一个参数 tag 是一个日志的标识，第二个参数是日志的内容，第三个参数是日志打印的目标，这个可以是控制台，也可以是文件，由调用者指定。

显然，我们通常调试时，输出日志都是直接到控制台的，于是我们定义一个新函数：

```
fun consoleLog(tag: String, message: Any?) = log(tag, System.out  
, message)
```

由于我们可能针对某一个问题不断地调试，这些日志的 tag 也是相同的，那么我们又会定义一个新函数：

```
val TAG = ...  
...  
fun consoleLogWithTag(message: Any?) = log(TAG, System.out, message)
```

这样看上去似乎没什么问题，不过你有可能会想，我不过是临时打几行日志，真的有必要定义这么多函数？调试一段代码还好，调试的内容多了呢，而且他们的 tag 都还不一样，难道我要定义 consoleLogWithTag2、consoleLogWithTag3 ... 么？

显然，如果你运用 Currying，问题就简单的多了，只不过是定义一个局部变量嘛：

```
val consoleLogWithTag = (>::log.curried())(TAG)(System.out)  
...  
// 打印日志  
consoleLogWithTag("This may be an error to call here.")
```

其中 log.curried() 这个方法的签名如下：

```
fun <P1, P2, P3, R> Function3<P1, P2, P3, R>.curried(): (p1: P1)
->(p2: P2)->(p3: P3)-> R{
    ...
}
```

注意，由于 log 是函数名，因此我们在获取其对应的函数引用时需要加 ::。

好，说到这里，你可能直接去试前面的代码，然后垂头丧气的告诉我，说我这代码是骗人的，根本不能跑。为啥呢？因为根本没有 curried() 这个方法啊。

对啊，非常遗憾，截止到 1.1RC 版，我们也没有看到这样的 API 出现在标准库当中，所以我们只好自己搞咯：

```
fun <P1, P2, P3, R> Function3<P1, P2, P3, R>.curried()
    = fun(p1: P1) = fun(p2: P2) = fun(p3: P3) = this(p1, p2, p3)
```

当然，这只是 Function3<P1, P2, P3, R> 的 curried() 实现版本，Kotlin 有 0-22 个 Function 版本，因此我们需要使用 Currying 这一特性的话，针对每一个版本都实现一个 curried 方法即可。

当然有 curried，就会有 uncurried，二者是完全相反的过程，我就不多讲了，大家可以自己尝试着实现一下。

3. 偏函数

我们再来看一下上一节这个打日志的例子，对于有三个参数的 log 函数，我们在绝大多数的使用场景下都对前两个参数传入了相同的值：

```
fun log(tag: String, target: OutputStream, message: Any?){
    ...
}

...
val consoleLogWithTag = (::log.curried())(TAG)(System.out)
```

其实，对一个多参数的函数，通过指定其中的一部分参数后得到的仍然是一个函数，那么这个函数就是原函数的一个偏函数了。从这个意义上来说，`consoleLogWithTag` 也可以认为是 `log` 的一个偏函数。

显然，偏函数与 Currying 有一些内在的联系，如果我们需要构造的偏函数的参数恰好处于原函数参数的最前面，那么我们是可以使用 Currying 的方法获得这一偏函数的；当然，如果我们希望得到任意位置的参数被指定后的偏函数，那么我们就有足够的理由使用一些更好的方法。

例如：

```
val makeString = fun(byteArray: ByteArray, charset: Charset): String{
    return String(byteArray, charset)
}

...

val makeStringFromGbkBytes = makeString.partial2(charset("GBK"))
//实际当中这个字节流可以是文件流，也可以是网络数据等等
val gbkByteArray = ...
println(makeStringFromGbkBytes(gbkByteArray))
```

对于第二个参数 `Charset`，我们在国内有不少公司仍在用 GBK 编码，那么在开发的过程中，我们就没有必要每次都指定 GBK 这个编码选项了，下面这一句代码返回了一个 `makeString` 的偏函数，这个函数第二个参数确定为 `charset("GBK")`。

```
val makeStringFromGbkBytes = makeString.partial2(charset("GBK"))
```

接下来，同样我们需要给出 `partial2` 的实现：

```
fun <P1, P2, R> Function2<P1, P2, R>.partial1(p1: P1) = fun(p2: P2) = this(p1, p2)
fun <P1, P2, R> Function2<P1, P2, R>.partial2(p2: P2) = fun(p1: P1) = this(p1, p2)
```

我们看到，我们为 Function2 实现了两个扩展方法 partial1 和 partial2，这两个方法分别用来生成两个参数分别被指定后的偏函数。

目前 Kotlin 标准库尚且没有对此提供支持，如果需要得到 FunctionN ($N > 1$) 的偏函数，那么我们需要把他们对应的 partialN 依次实现。

需要注意的是，makeString 是一个函数引用，可以直接用于调用函数的方法，这与上一节当中的 ::log 本质上是一样的，只是二者的定义方式不同，希望大家不要感到困惑。

```
// log 是函数名
fun log(tag: String, target: OutputStream, message: Any?) {
    ...
}

...
val consoleLogWithTag = (::log.curried())(TAG)(System.out)
```

4. 小结

本文主要给大家介绍了如何基于 Kotlin 的现有标准库来实现一些函数式编程的特性，其实这些特性已经在 Github 的 [funKTionale](#) 当中给出，本文的内容也更多的是在向它致敬。



像写文章一样使用 Kotlin

我把所有文章和视频都放到了 [Github](#) 上，如果你喜欢，请给个 Star，谢谢~

运算符重载

不知道大家有没有看到过下面的函数调用：

```
print "Hello World"
```

这样的感觉就好像在写文章，没有括号，让语言的表现力进一步增强。Groovy 和 Scala 就有这样的特性，那么 Kotlin 呢？

```
for(i in 0..10){  
    ...  
}
```

如果你在 Kotlin 中写下了这样的代码，你可能觉得很自然，不过你有没有想过 0..10 究竟是个什么？

“是个 IntRange 啊，这你都不知道。”你一脸嫌弃的回答道。

是啊，确实是个 IntRange，不过为什么是 0..10 返回个 IntRange，而不是 0->10 呢？

“我靠。。这是出厂设定，懂不懂。。”你开始变得更嫌弃了。。

额，其实我想说的是，你知道这其实是个运算符重载？！

```
public final operator  
    fun rangeTo(other: kotlin.Int)  
        : kotlin.ranges.IntRange {  
        ...  
    }
```

没错，Kotlin 允许我们对运算符进行重载，所以你甚至可以给 String 定义 rangeTo 方法。

去掉方法的括号

“毕竟运算符是有限的吧？如果说我想给 Person 增加个 say 方法，不带括号那种，怎么办？”你不以为然地说。

这个嘛。。。当然也是可以哒！

```
class Person(val name: String){
    infix fun 说(word: String){
        println("\\"$name 说 $word\\\"")
    }
}

fun main(args: Array<String>) {
    val 老张 = Person("老张")
    老张 说 "小伙砸, 你咋又调皮捏!"
}
```

这段代码执行结果是：

```
老张 说 "小伙砸, 你咋又调皮捏!"
```

这个看上去有有点儿意思不？代码跟输出是一样滴！有人会说，中文变量和函数名真的大丈夫？是滴，全然大丈夫，这是因为 Java、Kotlin 的字节码都采用 Unicode，中文确实是可以的——不过，Java 还支持中文包名和文件名，这在 Kotlin 当中还是有些问题的。

接着说，通常我们的方法传参是需要括号的，为什么这里不需要了呢？因为 infix 这个关键字！这里跟 Scala 和 Groovy 不同，Kotlin 只有显示的声明为 infix 的只有一个参数的方法才可以这么玩，如果你不显示声明，想去括号门儿都没有。

聊聊 DSL

好，抛开老张的例子不谈，毕竟真正生产环境下，谁会去用中文呢。什么情况下我们需要 infix？当然是 DSL 中。我们看一段 Gradle 配置：

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

看上去很有表现力是吧，即使你不懂 groovy 语法，也能直接看懂这句话就是使用 kotlin 插件。可它本质上还是句 groovy 代码，所以它的结构是怎样的呢？

```
void apply(Map<String, ?> config);
```

实际上，apply 是 PluginAware 的一个方法，后面的 plugin 呢？那不过是一个 k-v 对而已，在 groovy 当中，K:V 的形式可以表示一个键值对。那既然参数是 Map，那我要是多传几个参数是不是也可以呢？

```
apply ([plugin: 'kotlin', 宝宝: "不开心"])
```

不过这样虽然多传了一个键值对，不过由于 gradle 并不关心“宝宝”，所以“宝宝：不开心”那也没有办法了，说了也白说。哈哈。

前面的 DSL 是 groovy 版本的，再回到 Kotlin 当中，如果我们编写 DSL 代码，据说 Kotlin 版本的 gradle 也已经在开发中了，那么我们猜猜它会长成啥样？

```
apply mapOf("plugin" to "kotlin")
```

用 Kotlin 的 K to V 的方式传入一个 Pair。这里的 apply 的声明就应该是：

```
infix fun apply(config: Map<String, Any>)
```

很丑？没关系，我们为什么不直接搞一个方法叫做 applyPlugin 呢？

```
infix fun applyPlugin(pluginName: String)
```

于是：

```
applyPlugin "kotlin"
```

当然，作为静态语言，与 groovy 当然不能照搬了，所以最理想的实现肯定是强类型约束，比如

```
apply<KotlinPlugin>()
```

不过，这个我们就不谈了，跑题了。

小结

infix 是一个非常有用的关键字，让你的代码看起来像一篇文章一样，你不必再为前后括号匹配着慌，每一个单词其实都是方法调用。

函数复合

```
//f(g(x)) m(x) = f(g(x))
val add5 = {i: Int -> i + 5} // g(x)
val multiplyBy2 = {i : Int -> i * 2} // f(x)

fun main(args: Array<String>) {
    println(multiplyBy2(add5(8))) // (5 + 8) * 2

    val add5AndMultiplyBy2 = add5 andThen multiplyBy2
    val add5ComposeMultiplyBy2 = add5 compose multiplyBy2
        // P1 = 8; P2:add5的返回值
    println(add5AndMultiplyBy2(8)) // m(x) = f(g(x))
    println(add5ComposeMultiplyBy2(8)) // m(x) = g(f(x))
}

// infix 中缀表达式, Function1一个参数
// Function1<P1, P2>.andThen()扩展方法，P1参数类型，P2返回值类型
infix fun <P1, P2, R> Function1<P1, P2>.andThen(function: Function1<P2, R>): Function1<P1, R>{
    return fun(p1: P1): R{
        return function.invoke(this.invoke(p1))
    }
}

infix fun <P1,P2, R> Function1<P2, R>.compose(function: Function1<P1, P2>): Function1<P1, R>{
    return fun(p1: P1): R{
        return this.invoke(function.invoke(p1))
    }
}
```

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。

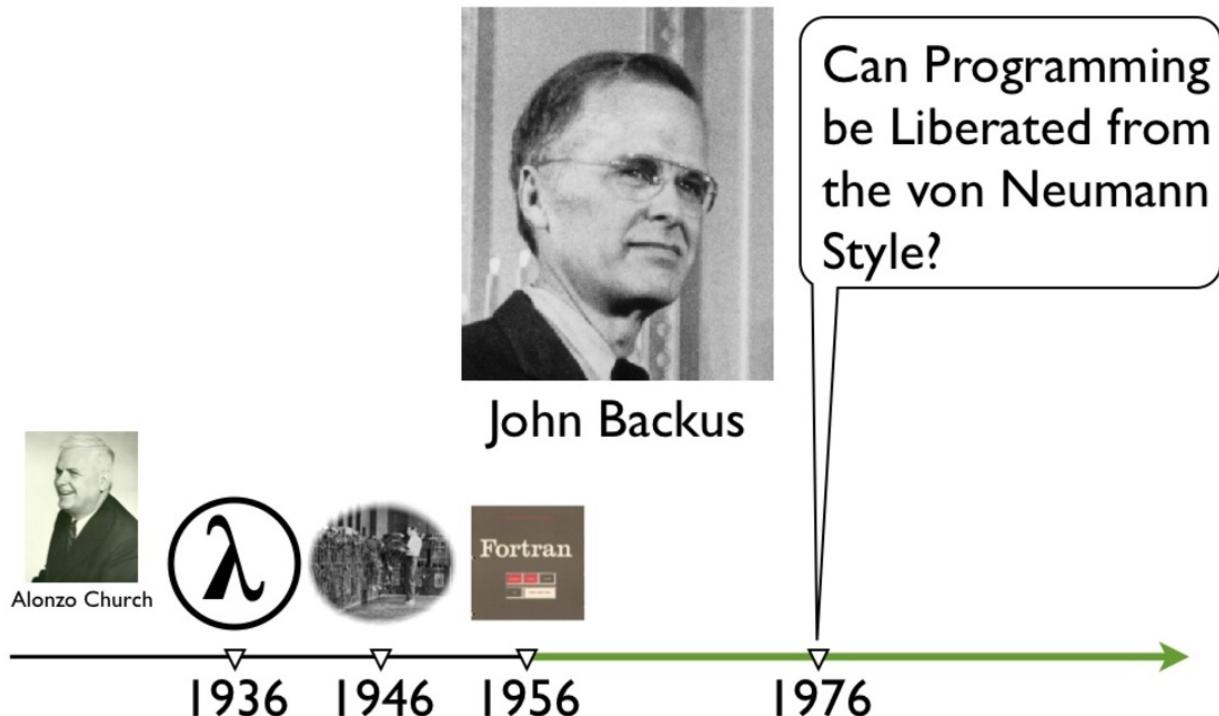
函数式编程中最古老的例子莫过于1958年被创造出来的Lisp了。Lisp由约翰·麦卡锡（John McCarthy，1927-2011）在1958年基于 λ 演算所创造，采用抽象数据列表与递归作符号演算来衍生人工智能。较现代的例子包括Haskell、ML、Erlang等。现代的编程语言对函数式编程都做了不同程度的支持，例如：JavaScript, Coffee Script, PHP, Perl, Python, Ruby, C#, Java 等等（这将是一个不断增长的列表）。

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。

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 [ən'tsaɪdʊŋspʁo̯ble:m]））。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。

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。

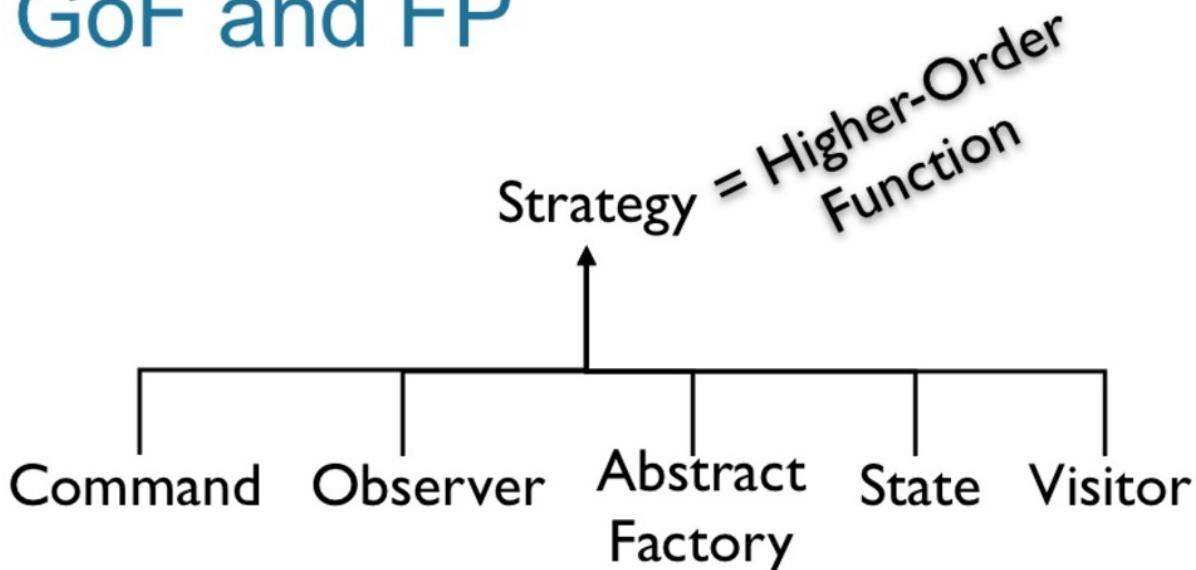
绝大部分设计模式的实现都离不开多态性的思想。换一种说法就是，这些设计模式背后的本质其实就是OOP的多态性，而OOP中的多态本质上又是受约束的函数指针。

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：

GoF and FP



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）。

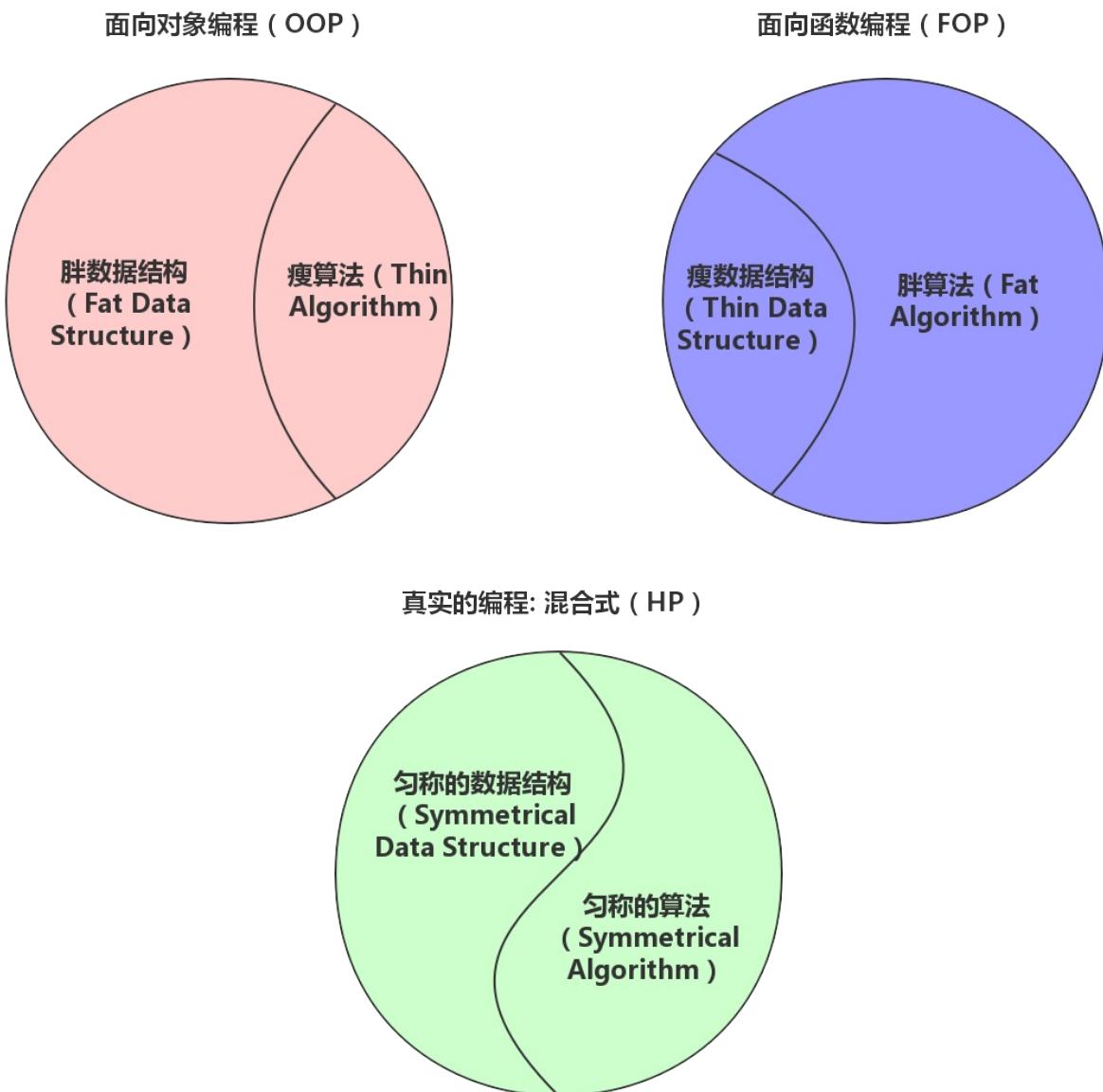
在面向函数式的编程中，一切皆是函数（偏重算法，轻数据结构）。我们把它叫做：瘦数据结构-胖算法（TDS-FA）。

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：



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：

$f: X \rightarrow Y$ $p: Y \rightarrow Z$ $p(f) : X \rightarrow Z$

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))}  
}
```

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。

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 $f: A \rightarrow B, g: B \rightarrow C$ ，则 $g.f: A \rightarrow C$.

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 $I_a: A \rightarrow A$ ，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：

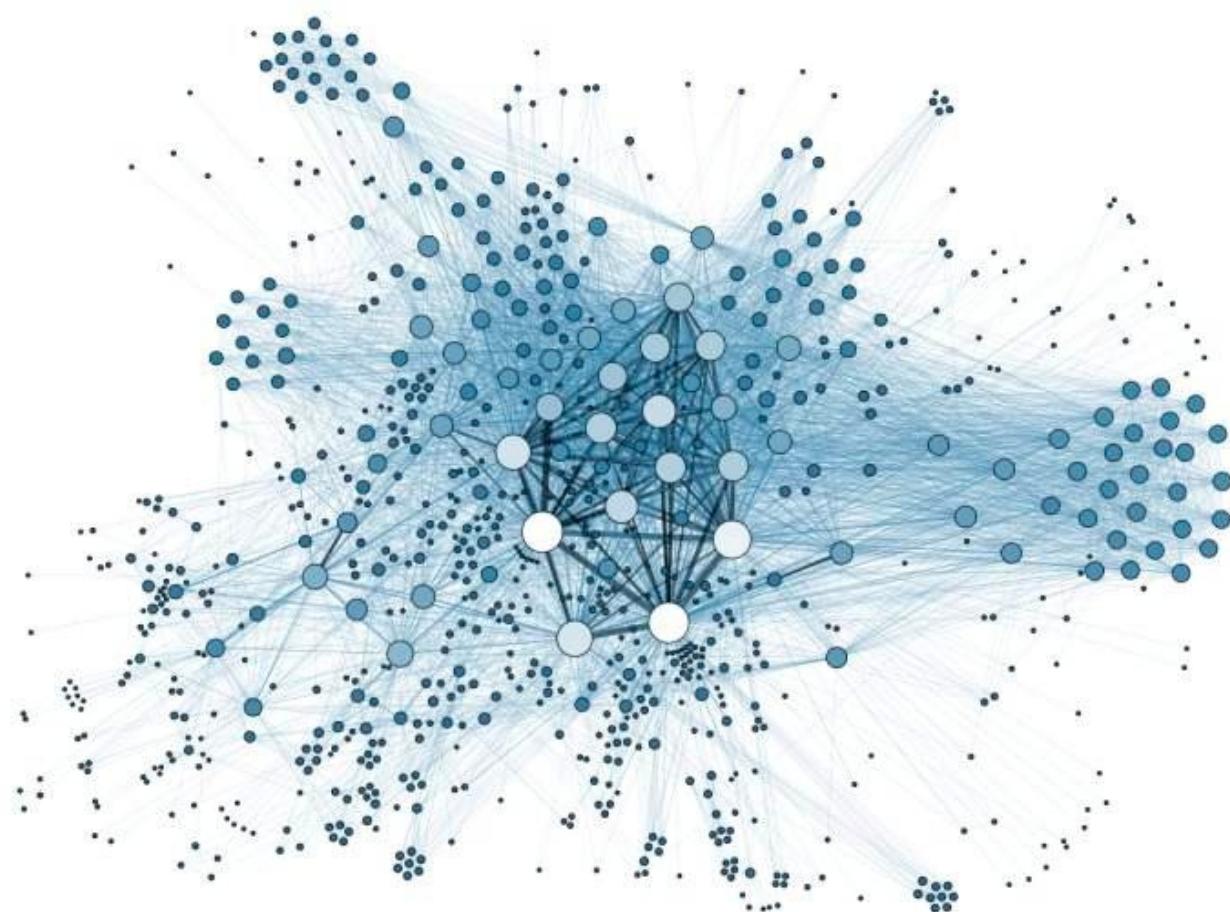
- 将范畴 A 中的对象映射到范畴 B 中的对象。
- 将范畴 A 中的态射映射到范畴 B 中的态射。

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

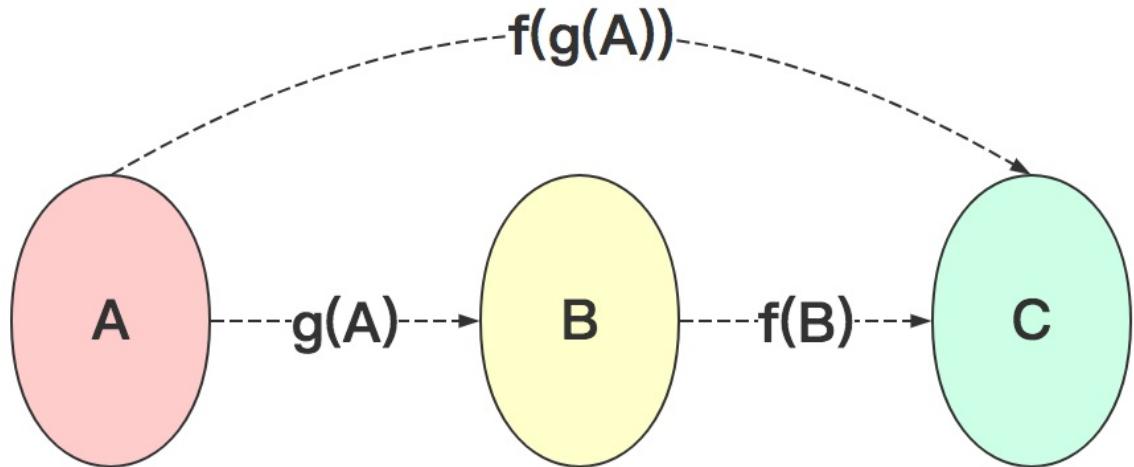
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

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：

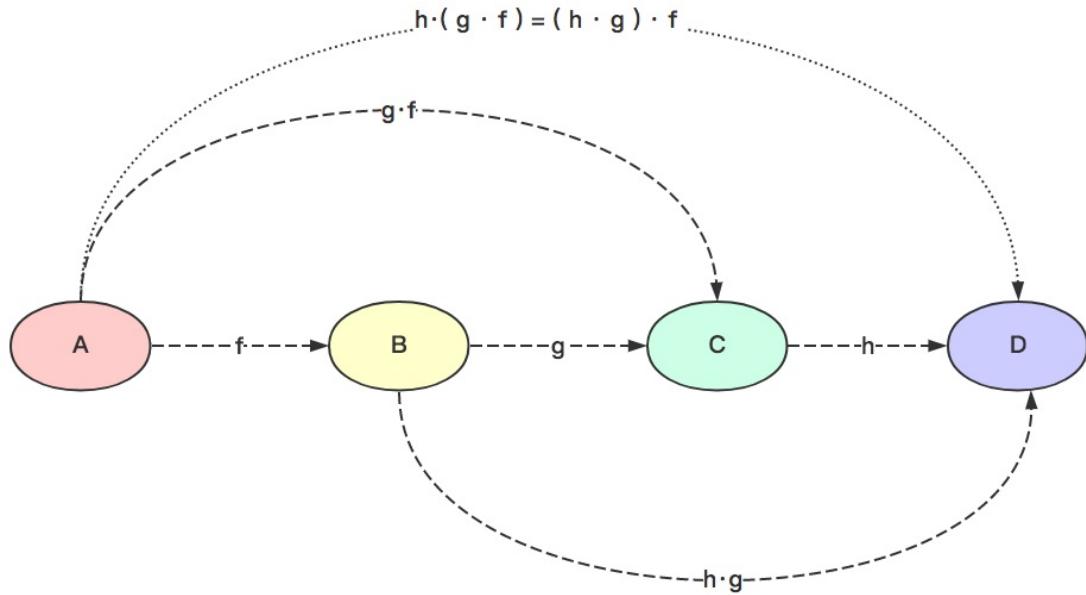
```
fun <String, Int, Boolean> compose(f: (Int) -> Boolean, g: (String) -> Int): (String) -> Boolean
```

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： $f(g(x)) \% 2 \neq 0$ ，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。

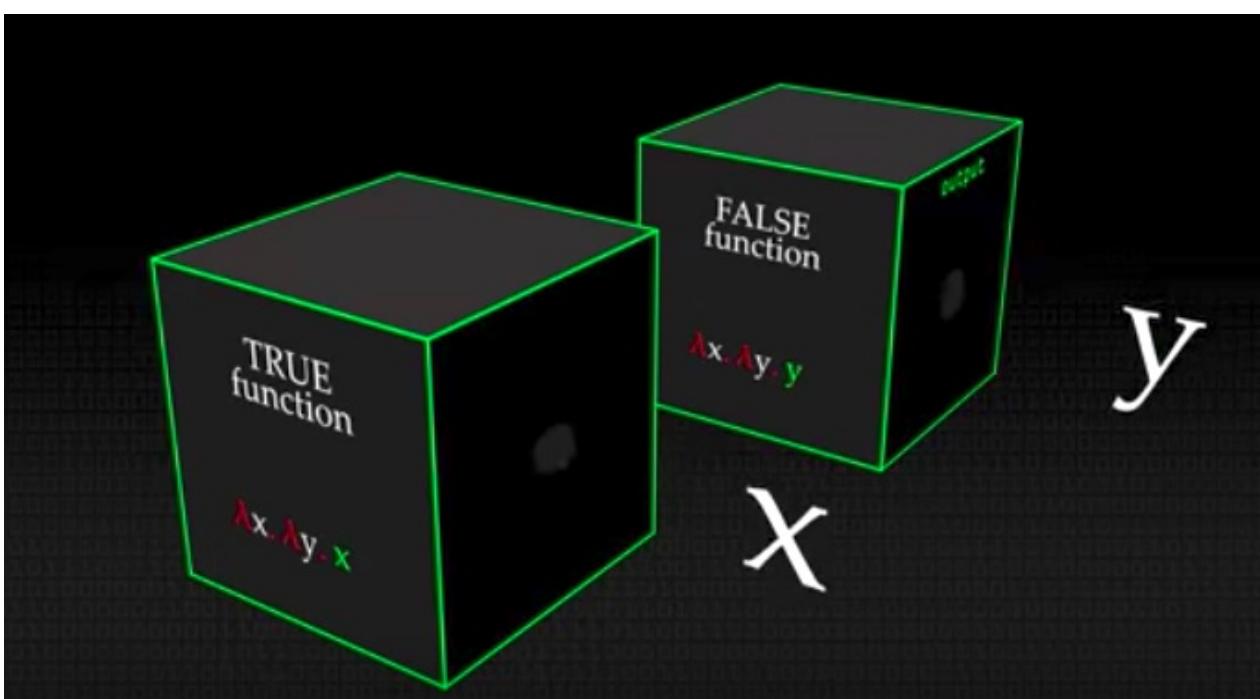
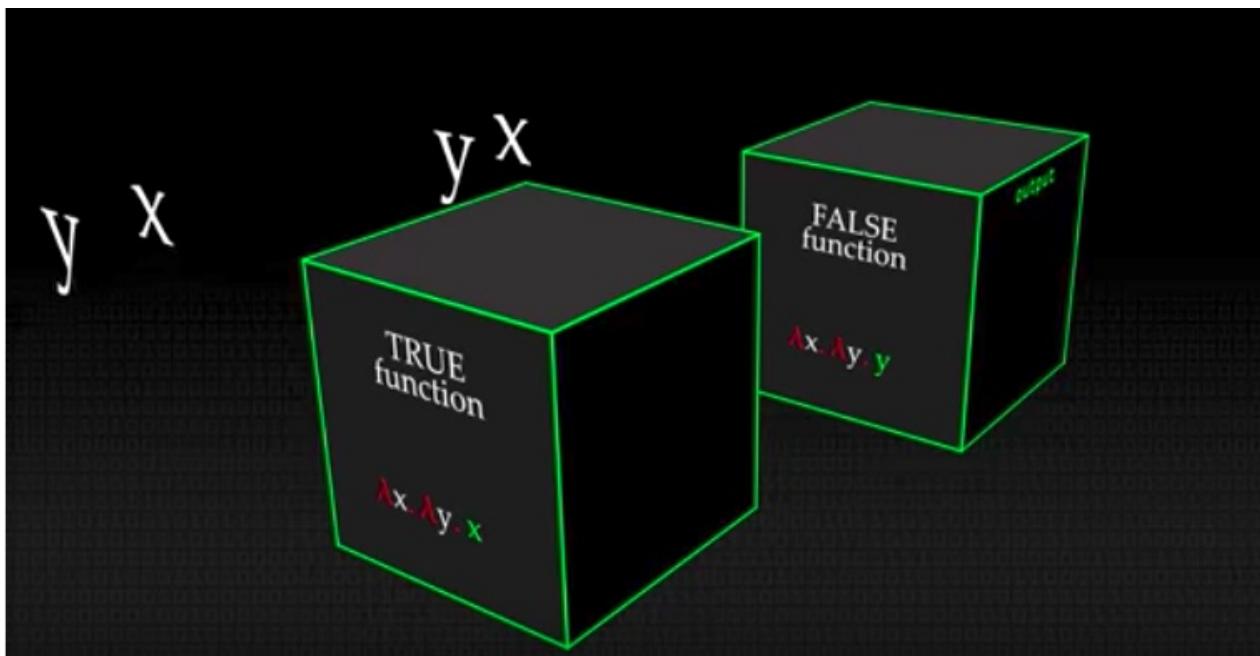
其中的 $\lambda f. \lambda x.$ ， λf 是抽象出来的函数， λx 是输入参数， $.$ 语法用来分割参数表和函数体。为了更简洁，我们简记为 F，那么上面的 Church 整数定义简写为：

```
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 $x+1$ ：

```
 $\lambda x . x + 1$ 
```

测试代码：

```
(  $\lambda x . x + 1$  ) 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 addOneAnonymous = (fun(x: Int): Int {
...     return x + 1
... })
>>> addOneAnonymous(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：

$$\lambda f. \lambda x. f x$$

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 $\text{curryAdd}(1) = 1 + y$ ；然后传入参数 $y = 2$ ，返回最终的值 $\text{curryAdd}(1)(2) = 3$ 。

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：

$\text{fibonacci}(0) = 0 \quad \text{fibonacci}(1) = 1 \quad \text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$

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：

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

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： $Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$ ，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句要简洁许多：

```

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;

/**
 * Created by jack on 2017/7/9.
 */
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;

/*
 * Created by jack on 2017/7/9.
 */
public class YCombinator2 {

    public static Lambda<Lambda> yCombinator2(final Lambda<Lambda>
a> 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...>点击预览；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）。

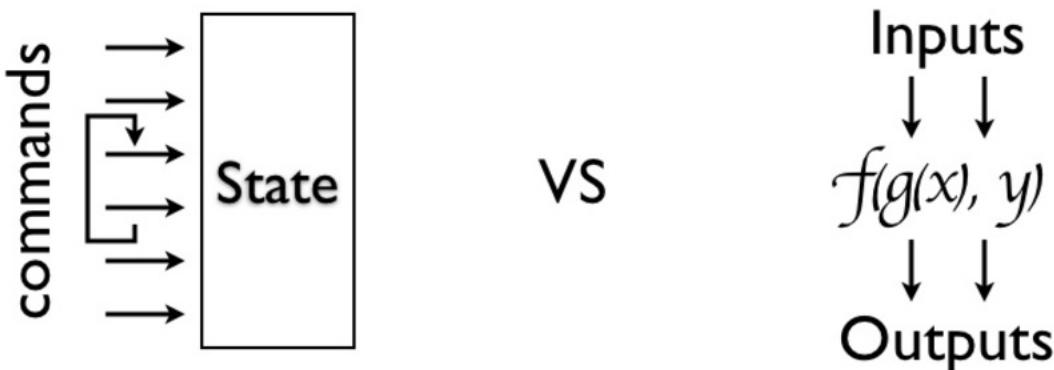
严谨而精巧的lambda演算体系，从最基本的概念“函数”入手，创造出一个绚烂而宏伟的世界，这不能不说这是人类思维的骄傲。

没有“副作用”

Effects (Mutability)



John von Neumann



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。

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。

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。

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`

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)
```

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)
```

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?) {
    ...
}
```

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
```

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
```

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)  
}
```

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。

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
}))
```

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
```

`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]
```

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
```

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
    type
    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。

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。

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

集合框架

- 集合泛型与操作符
- Iterator
- 集合框架
- 集合类是什么
- Kotlin 集合类简介
- List
- List元素操作函数
- List集合类的基本运算函数
- List过滤操作函数
- 映射操作函数
- 分组操作函数
- 排序操作符
- 生产操作符
- Set
- Map
- 集合泛型与操作符

Iterable

```
public interface Iterable<out T> {  
    /**  
     * Returns an iterator over the elements of this object.  
     */  
    public operator fun iterator(): Iterator<T>  
}
```

Iterator

- next()
- hasNext()

```
public interface Iterator<out T> {  
    /**  
     * Returns the next element in the iteration.  
     */  
    public operator fun next(): T  
  
    /**  
     * Returns `true` if the iteration has more elements.  
     */  
    public operator fun hasNext(): Boolean  
}
```

```
/**
 * Given an [iterator] function constructs an [Iterable] instance
 * that returns values through the [Iterator]
 * provided by that function.
 */
@kotlin.internal.InlineOnly
public inline fun <T> Iterable(crossinline iterator: () -> Iterator<T>): Iterable<T> = object : Iterable<T> {
    override fun iterator(): Iterator<T> = iterator()
}
```

```
fun main(args: Array<String>) {

    var x = 5
    while(x > 0){
        println(x)
        x--
    }

    do{
        println(x)
        x--
    }while (x > 0)

    //    for (arg in args){
    //        println(arg)
    //    }
    //
    //    for((index, value) in args.withIndex()){
    //        println("$index -> $value")
    //    }
    //
    //    for(indexedValue in args.withIndex()){
    //        println("${indexedValue.index} -> ${indexedValue.value}")
    //    }
    //
    //    val list = MyIntList()
```

```
//      list.add(1)
//      list.add(2)
//      list.add(3)
//
//      for(i in list){
//          println(i)
//      }
}

class MyIterator(val iterator: Iterator<Int>){
    operator fun next(): Int{
        return iterator.next()
    }

    operator fun hasNext(): Boolean{
        return iterator.hasNext()
    }
}

class MyIntList{
    private val list = ArrayList<Int>()

    fun add(int : Int){
        list.add(int)
    }

    fun remove(int: Int){
        list.remove(int)
    }

    operator fun iterator(): MyIterator{
        return MyIterator(list.iterator())
    }
}
```

Array

- arrayOfNulls() 空数组
- arrayOf()
- xxArrayOf()
- withIndex()

Arrays

- forEach()

```
public inline fun <T> Array<out T>.forEach(action: (T) -> Unit):  
Unit {  
    for (element in this) action(element)  
}
```

Kotlin集合

- Iterable
- MutableIterable
- Collection
- MutableCollection
- List 只读
- MutableList 可读可写
- Set
- MutableSet
- Map
- MutableMap

Collection

方法说明	功能描述
joinToString	

集合和函数操作符

方法说明	功能描述
listOf()	
mutableListOf	
toList	
setOf()	
mutableSetOf	
hashSetOf	
linkedSetOf	
toSet	
mapOf()	
hashMapOf	
linkedMapOf	

总数操作符

方法说明	功能描述
any	至少一个元素符合条件
all	所有元素符合条件
count	符合条件的元素总数
fold	把一个集合的元素折叠起来的并得到一个最终的结果
foldRight	
foreach	遍历
foreachIndexed	
max	
maxBy	
min	
minBy	
none	
reduce	与fold类似，只不过没有初始值，返回值类型也需要保持与集合的元素相同
reduceRight	
sumBy	

过滤操作符

方法声明	功能描述
drop	
dropWhile	
filter	过滤
slice	过滤list中指定index的元素
take	
takeLast	
takeWhile	原集合中从第一个元素开始到第一个不符合条件的元素之前的所有元素

映射操作符

方法声明	功能描述
map	把一个集合映射成另外一个集合
flatMap	打平集合
groupBy	按关键字分组

- map()

```

/**
 * Returns a list containing the results of applying the given [transform]
 * function to each element in the original collection.
 */
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}

/**
 * Applies the given [transform] function to each element of the
 * original collection
 * and appends the results to the given [destination].
 */
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
}

```

元素操作符

方法声明	功能描述
contains	
elementAt	
first	
indexOf	
last	
lastIndexOf	
single	

产生操作符

方法声明	功能描述
merge	合并
partition	
plus	
zip	

顺序操作符

方法声明	功能描述
reverse	
sort	
sortBy	
sortDescending	

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。

集合类是什么

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。

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识识别，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。

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。

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)。

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。

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。

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，

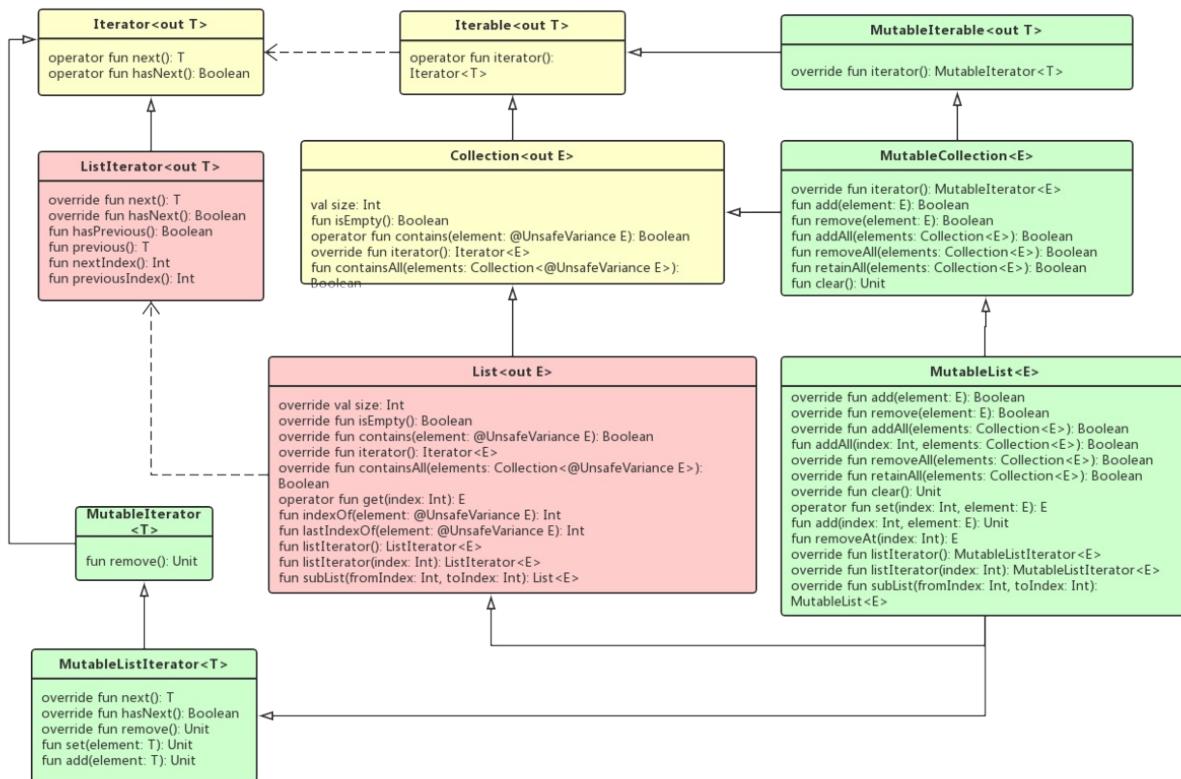
散列码通常用作一个偏置量，该偏置量是相对于分配给映射的内存区域起始位置的，由此确定关键字/对象对的存储位置。理想情况下，散列处理应该产生给定范围内均匀分布的值，而且每个关键字应得到不同的散列码。

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。

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.Ar  
rayList<E>
```

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.Ra
ndomAccess
@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.Ar
rayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.
util.LinkedHashMap<K, V>
@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.H
ashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.uti
l.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.Hash
Set<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)

```

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.NoSuchElementException
```

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。

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：

```
>>> mutableListOf.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`

返回指定元素的下标，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
```

List集合类的基本运算函数

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) {
    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(s
elector: (T) -> Int): Int
cannot be applied to
receiver: List<String> arguments: ((String) -> String)

list.sumBy({it})
^
error: type mismatch: inferred type is (String) -> String but (S
tring) -> Int was expected
list.sumBy({it})
^
```

过滤操作函数

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(_Co
llections.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) -> Bo
olean): 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.nextInt() + 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。

映射操作函数

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]
```

分组操作函数

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 }
```

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 selector: (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, ]
```

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), o
ther.collectionSizeOrDefault(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})
[1, 6, 9]
>>> 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<T>、List<R>中，然后返回一个first为List<T>，second为List<R>的大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>, List<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.Internal
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

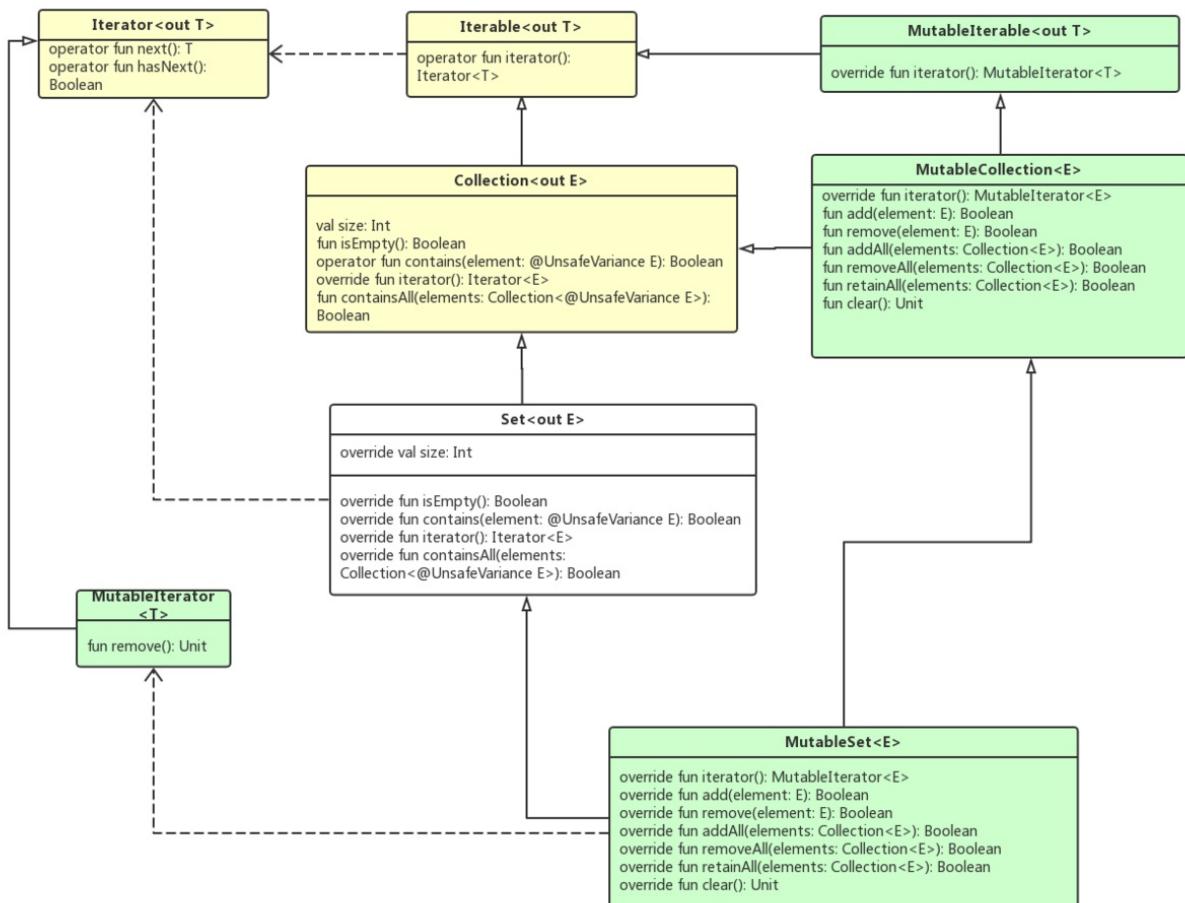
```
>>> list1 + 10
[1, 2, 3, 10]
>>> list1.plusElement(10)
[1, 2, 3, 10]
>>> list1.plus(10)
[1, 2, 3, 10]
```

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：



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 = 34066037743870205
32

    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

```

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 emptySet()
```

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> = LinkedHash
Set()
```

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。

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.Ra
ndomAccess

@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.Ar
rayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.
util.LinkedHashMap<K, V>
@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.H
ashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.uti
l.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.Hash
Set<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, 3, 2, 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。 LinkedHashMap：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。

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]
```

Map

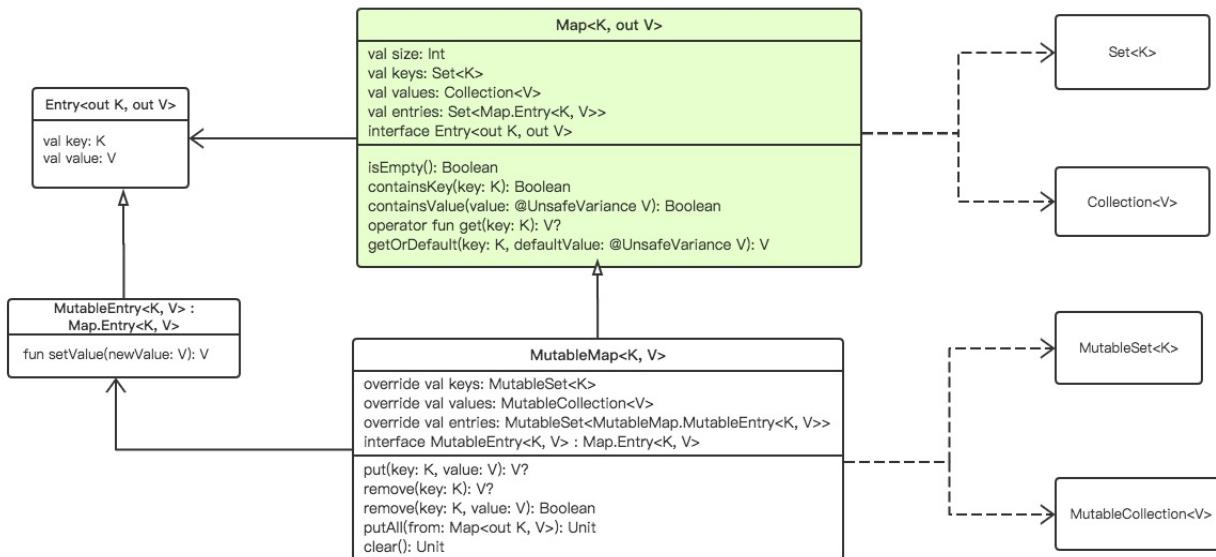
Map概述

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。

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 = 82467148295456882
    74

        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
}
```



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, St
ring>.set(key: Int, value: String): Unit defined in kotlin.colle
ctions
@InlineOnly public operator inline fun kotlin.text.StringBuilder
    /* = java.lang.StringBuilder */.set(index: Int, value: Char): U
nit 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}
```

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
```

如果不想要返回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")
^
```

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.Internal
public inline operator fun <K, V> Map.Entry<K, V>.component1():
K = key

@kotlin.internal.Internal
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)
```

getOrDefault(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.getOrDefault("x", { 1 })
1
>>> map["x"] = 3
>>> map.getOrDefault("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.getOrImplicit
DefaultNullable(MapWithDefault.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(pai
r: Pair<K, V>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pai
rs: Iterable<Pair<K, V>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pai
rs: Array<out Pair<K, V>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pai
rs: 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

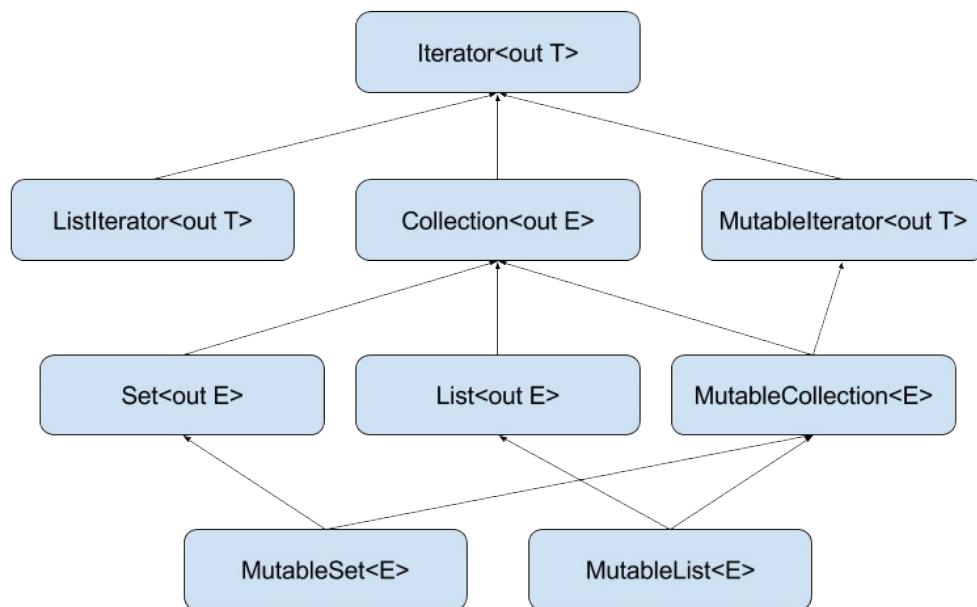
本章我们介绍了Kotlin标准库中的集合类List、Set、Map，以及它们扩展的丰富的操作函数，这些函数使得我们使用这些集合类更加简单容易。

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。

集合泛型与操作符

所谓泛型：就是允许在定义类、接口、方法时指定类型形参，这个类型形参将在声明变量、创建对象、调用方法时动态地指定(即传入实际的类型参数，也可称为类型实参)。Kotlin 泛型定义与 Java 类似，但有着更多特性支持。首先，我们来看看集合类中的泛型声明。

1. Kotlin 中的集合接口



以上是 Kotlin 的集合接口关系，从上到下，依次下面继承(或实现)上面的接口。

所有类声明的泛型尖括号里面如果加入了 `out` 关键字，则说明这个类的对象是只读的，例如他只有：`get()`、`size ()` 等方法，而没有 `set()`、`remove()` 等方法。

相反的，如果没有加 `out` 关键字，或者换一种记法：如果开头是 `MutableXXX` 那就是一个跟 Java 中用法一致的集合类。

2. in 与 out

至此，你应该已经知道 `out` 关键字的含义了，与 `out` 对应的，还有一个 `in`。表示泛型参数是只写的。

泛型的 `in` 关键字一般用在我们定义的代码中。在看下面这个 demo 之前，你需要知道一点：**Kotlin** 的泛型是支持协变的。例如下面这段代码：

```
open class A
open class B : A()
open class C : B()

val mutableListOf: MutableList<B> = mutableListOf(B(), B(), C())
val list: List<A> = mutableListOf;
```

在 Kotlin 中，可以直接把 `List<C>` 的对象赋值给 `List<A>` 知道了上面这一点，我们接下来看 `in` 的用法。

泛型声明中，用 `in` 声明泛型参数，表示这个参数只能是入参，不能对外返回这个参数。

```
open class A
open class B : A()
open class C : B()

class TypeArray<in A> {

    //in 修饰了 A，表示 A 是可以作为参数的。
    fun getValue(a: A): Int? {
        return a?.hashCode()
    }

    //这段代码是非法的，因为A 不能被返回
    fun getA(a: A): A? {
        return a
    }
}
```

3. 集合的初始化

在 Kotlin 中，集合类一般不使用构造方法去初始化，而是使用统一的入口方法，例如初始化一个 `MutableList`，我们使用的是如下代码：

```
val mutableList = mutableListOf(0, 1, 2, 3)
```

类似的初始化集合对象的方法还有

```
// 创建一个 List<> 对象
var list = listOf(0, 1, 2)

// 创建一个 Set<> 对象
val ss = setOf(1, 2, 4)
```

还有很多，可以在 `kotlin.collections.Collections.kt` 文件中查看。

有一点需要知道的是，这些方法的返回值实际上返回的是 Java 的集合对象，目前只支持返回这些对象：

```
java.util.ArrayList
java.util.LinkedHashMap
java.util.HashMap
java.util.LinkedHashSet
java.util.HashSet
java.util.SortedSet
java.util.TreeSet
```

其他的对象，目前还不支持，比如 `LinkedList`，个人觉得这个对象还是很常用的，不知道为什么没有方法提供，所以目前这类对象还是只能用构造方法去创建。

4. 操作符

Kotlin 中，操作符是用来对数据做操作的工具方法。

如果你使用过 RxJava 等一系列库，你一定会对操作符非常了解也对操作符的强大深有感触。

Kotlin 原生支持大量操作符，文尾将会列出常用操作符及含义。这两幅图是我上周在《从 Java 到 Kotlin，当机器人不再喝咖啡》技术分享时 PPT 中的两页，代码运行后是当时会场的 WiFi 密码，大家可以感受一下用 Kotlin 和 Java 实现的对比。

```

final String[] a = new String[]{"0", "1", "2", "6", "7", "h", "j"};
Integer[] indexArray = new Integer[]{5, 9, 6, 8, 2, 7, 0, 1, 4, 0, 3};
List<Integer> list = Arrays.asList(indexArray);

Observable.just(list)
    .flatMap((Func1) (integers) -> { return Observable.from(integers); })
    .filter((Func1) (integer) -> { return integer < a.length; })
    .map((Func1) (integer) -> { return a[integer]; })
    .reduce((s, s2) -> { return s + s2; })
    .subscribe((Action1) (s) -> {
        Log.i( tag: "kymjs", msg: "wifiPassword::" + s);
    });

var wifiPassword = "?"  

val a = arrayOf("0", "1", "2", "6", "7", "h", "j")
wifiPassword = arrayOf(5, 9, 6, 8, 2, 7, 0, 1, 4, 0, 3)
    .filter {
        it in 0 until a.size
    }
    .map {
        a[it]
    }
    .reduce { s1, s2 ->
        "$s1$s2"
    }

print(wifiPassword)

```

5. 操作符实现原理

以最简单也是最常用的一个操作符 `forEach` 举例。

```

list.forEach {  
}
```

在 Kotlin 中，操作符本质上是方法调用。例如 List 的 `forEach` 的实现原理实际上是定义了一个这样的方法。为 Iterable 增加了一个扩展方法，叫 `forEach`，它接收一个 T 作为参数，并返回 Unit 的闭包作为参数。

```
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

而内部，实际上就是 for 循环对每一个对象调用传入的参数方法。

所以，我们也可以很轻松的通过这样的特性实现自己的自定义操作符。例如下面的代码为一个 List 添加一个转换操作符，可以将 `List<A>` 转换成 `List`，其中 A 和 B 可以为任何类型。

```
inline fun <T, E> Iterable<T>.convert(action: (T) -> E): MutableList<E> {
    val list: MutableList<E> = mutableListOf()
    for (element in this) list.add(action(element))
    return list
}

{
    val list: List<String> = listOf("hello", "world")
    list.convert {
        it.hashCode()
    }.forEach {
        print("$it")
    }
}()
```

这样子就可以直接调用 list 的 convert 方法，只需要实现转换逻辑，就可以了。

6. 常用操作符

Kotlin 的操作符跟 RxJava 基本一致，不需要额外记忆。

下标操作类

- contains —— 判断是否有指定元素
- elementAt —— 返回对应的元素，越界会抛IndexOutOfBoundsException
- firstOrNull —— 返回符合条件的第一个元素，没有返回null
- lastOrNull —— 返回符合条件的最后一个元素，没有返回null
- indexOf —— 返回指定元素的下标，没有返回-1
- 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

判断类

- any —— 判断集合中是否有满足条件的元素
- all —— 判断集合中的元素是否都满足条件
- none —— 判断集合中是否都不满足条件，是则返回true
- count —— 查询集合中满足条件的元素个数
- reduce —— 从第一项到最后一项进行累计

过滤类

- filter —— 过滤掉所有满足条件的元素
- filterNot —— 过滤所有不满足条件的元素
- filterNotNull —— 过滤NULL
- take —— 返回前 n 个元素

转换类

- map —— 转换成另一个集合（与上面我们实现的 convert 方法作用一样）；
- mapIndexed —— 除了转换成另一个集合，还可以拿到Index(下标)；
- mapNotNull —— 执行转换前过滤掉为 NULL 的元素
- flatMap —— 自定义逻辑合并两个集合；
- groupBy —— 按照某个条件分组，返回Map；

排序类

- reversed —— 反序
- sorted —— 升序
- sortedBy —— 自定义排序
- sortedDescending —— 降序

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 jdk 5 中，Java 引入了泛型。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 cannot 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 getDefault(key: K, defaultValue: @UnsafeVarianc
e 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<Int, String>` 表示参数化类型 `K, V` 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：

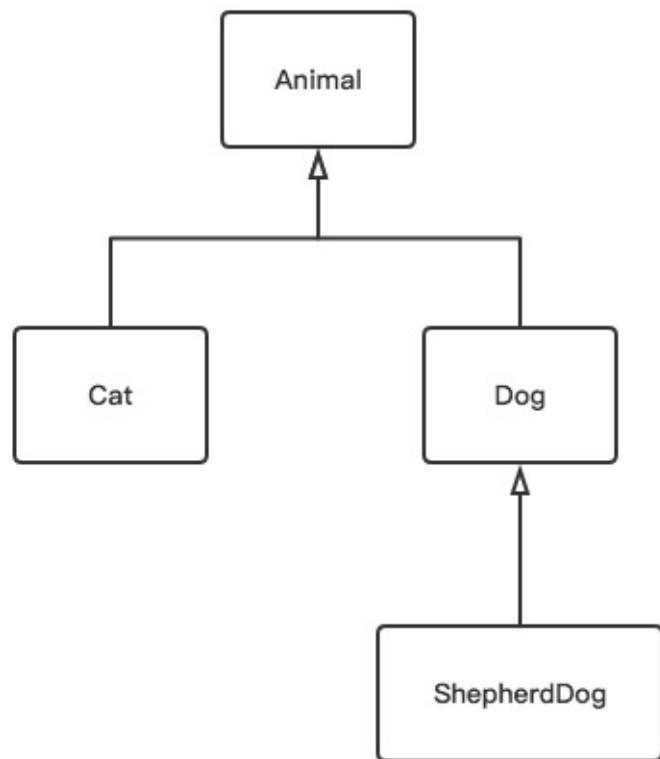
```
List<Animal> list3 = new ArrayList<>();
list3.add(new Dog());
list3.add(new Cat());
animal.act(list3);

List<Dog> list4 = new ArrayList<>();
list4.add(new Dog());
list4.add(new Dog());
animal.act(list4);

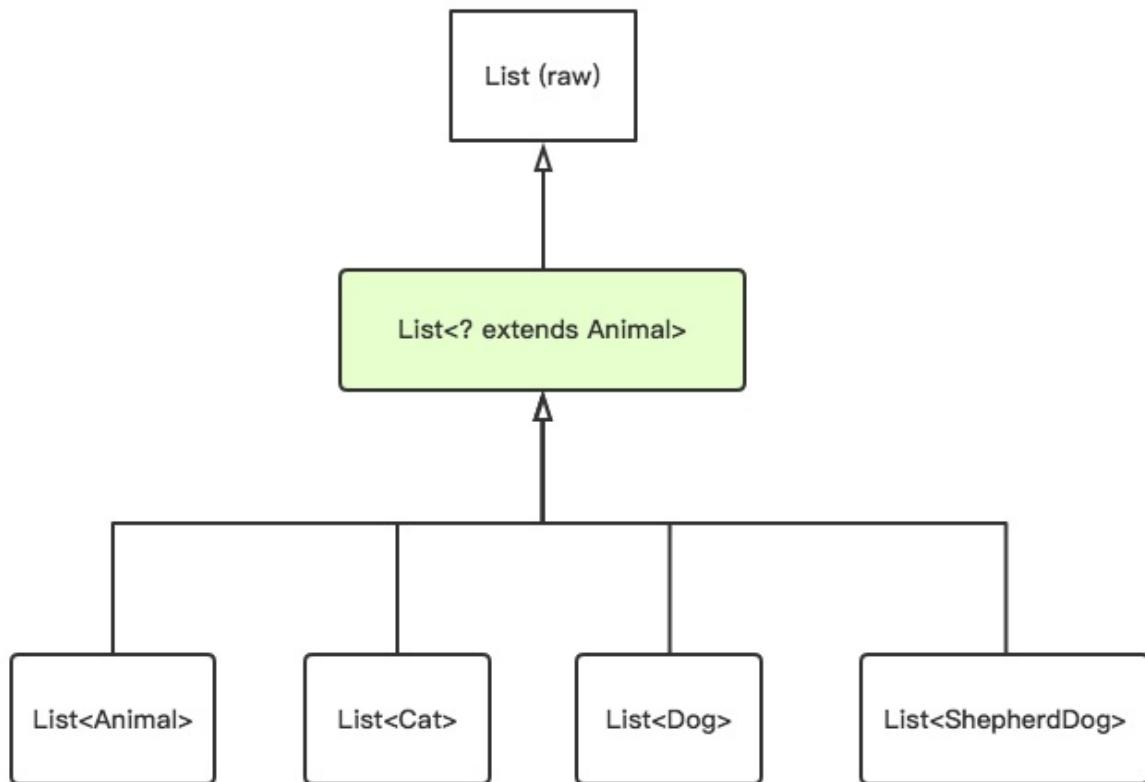
List<Cat> list5 = new ArrayList<>();
list5.add(new Cat());
list5.add(new Cat());
animal.act(list5);
```

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<Dog>`并不是`List<Animal>`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：

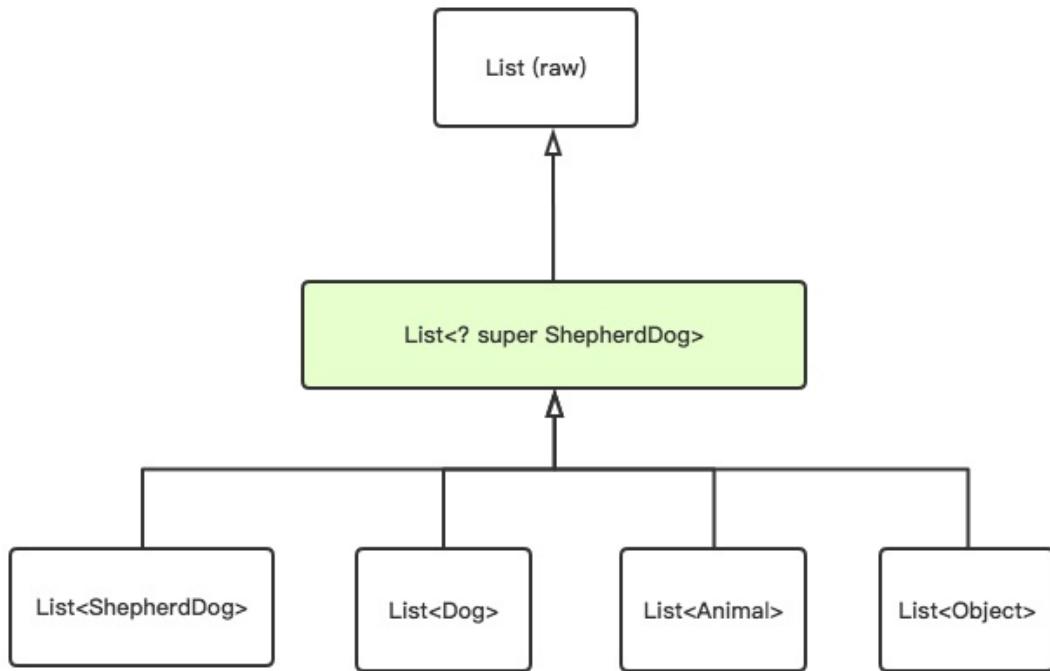
```

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<Integer> 不是 List<Number> 的子类型，试图在要求 List<Number> 的位置提供 List<Integer> 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<Number>的对象却不能由ArrayList<Integer>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<Animal>, List<Dog>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 $F \subset C$ 时，如果有 $f(F) \subset f(C)$ ，那么 f 叫做协变（Convariant）；当 $F \subset C$ 时，如果有 $f(C) \subset f(F)$ ，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`。

`<? 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<Integer>` 可以添加 `Integer` 及其子类，`List<Float>` 可以添加 `Float` 及其子类，`List<Integer>`、`List<Float>` 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<E> 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<Number>的对象stack，src有Iterable<Integer>与Iterable<Float>；

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<Integer>与 Iterable<Float>都不是Iterable<Number>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<E> dst)方法，将Stack中的元素依次取出add到dst中，如果不通用配符实现：

```
// 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<Number>的对象stack，dst为Collection<Object>；

调用popAll方法是会发生type mismatch错误，因为Collection<Object>不是Collection<Number>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 RandomAc
cess)) {
        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(); // ou
t 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(); // i
n T, 写入dest数据
    ListIterator<? extends T> si = src.listIterator(); // o
ut 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。

文章昨天发出之后收到了大家的反馈：

- 感谢 @萌夜雀 指出的关于 Star Projection 的问题，有些疏漏，文中已修正，啊求大佬放过，不要让我退群；
- 也感谢 @冻雨 指出的“到底是型变还是协变”的问题，为了不引起混淆，文中调整了措辞；
- 另外，也感谢 @0u0 @Snk，“搞基函数”明显是个笔误好不啦

成文仓促，略有疏漏，非常抱歉，原文已删，文章重发，谢谢诸位，请不吝赐教！

0. 引子

Kotlin 100% 与 Java 兼容，所以抛开语言表面上的种种特质之外，背后的语言逻辑或者说“灵魂”与 Java 总是相通的。本文只涉及 Kotlin Jvm，Kotlin Js、Kotlin Native 的具体实现可能有差异。

最近一段时间在慕课网上发了一套 Kotlin 的入门视频，涵盖了基础语法、面向对象、高阶函数、DSL、协程等比较有特色的知识点，不过有朋友提出了疑问：这门课为什么不专门讲讲泛型、反射和注解呢？

我最早听到这个问题的时候，反应比较懵逼，因为我居然没有感觉到 Kotlin 的反射、泛型特别是注解有专门学习的必要，因为他们跟 Java 实在是太像了。

实际上，从社区里面学习 Kotlin 的朋友的反应来看，大家大多对于函数式的写法，DSL，协程这些内容比较困惑，或者说不太适应，这与大家的知识结构是密切相关的，面向对象的东西大家很容易理解，因为就那么点儿内容，你懂了 C++ 的面向对象，Java 的也很容易理解，Kotlin 的也就不在话下了；而你没有接触过 Lua 的状态机，没有接触过 Python 的推导式，自然对于协程也就会觉得比较陌生。

所以我想说的是，泛型这东西，只要你对 Java 泛型有一定的认识，Kotlin 的泛型基本可以直接用。那我们这篇文章要干嘛呢？只是做一个简单的介绍啦，都很好理解的。

1. 真·泛型和伪·泛型

Java 的泛型大家肯定都知道了，1.5 之后才加入的，可以为类和方法分别定义泛型参数，就像下面这样：

```
public class Generics<T>{
    private T t;
    ...
    public <R> R getResult(){
        ...
    }
}
```

Kotlin 的写法呢？完全一样：

```
class Generics<T>{
    private val t: T
    ...
    fun <R> getResult(): R{
        ...
    }
}
```

Java/Kotlin 的泛型实现采用了类型擦除的方式，这与 C# 的实现不同，后者是真·泛型，前者是伪·泛型。当然这么说是从运行时的角度来看的，在编译期，Java 的泛型对于语法的约束也是真实存在的，所以你愿意的话，也可以管 Java 的泛型叫做编译期真·泛型。

那么什么是真·泛型呢？我们给大家看一段 C# 的代码：

```

using System;

public class Program{
    public static void Main(String[] args){
        testGeneric<string>();
    }

    public static void testGeneric<T>(){
        Console.WriteLine(typeof(T));
    }
}

```

`testGeneric` 的泛型参数 `string` 可以在运行时获取到，俨然一个真实可用的类型啊。下面是输出的结果：

```
System.String
```

那伪·泛型呢？如果同样的代码放到 Java 或者 Kotlin 当中，结果会怎样呢？

```

public static <T> void testGenerics(){
    System.out.println(T.class);
}

```

这段代码无法编译，因为 `T` 是个泛型参数，你不能用它去获取 `class` 对象。为了更清楚地说明问题，我们看下下面的代码：

```

public static <T> T testGenerics(){
    T t = null;
    return t;
}

```

编译后的字节码：

```

public static testGenerics()Ljava/lang/Object;
L0
    LINENUMBER 13 L0
    ACONST_NULL
    ASTORE 0
L1
    LINENUMBER 14 L1
    ALOAD 0
    ARETURN
L2
    LOCALVARIABLE t Ljava/lang/Object; L1 L2 0
    // signature TT;
    // declaration: T
    MAXSTACK = 1
    MAXLOCALS = 1

```

我们看到，编译之后 T 变成了 Object，简单来说就相当于：

```

public static Object testGenerics(){
    Object t = null;
    return t;
}

```

这就是传说中的类型擦除了。而 Kotlin 在 JVM 之上，编译之后也是字节码，机制与 Java 是一样的。也正是因为这个原因，我们在使用 Gson 反序列化对象的时候除了制定泛型参数，还需要传入一个 class：

```

public <T> T fromJson(String json, Class<T> classOfT) throws JsonSyntaxException {
    ...
}

```

显然 Gson 没有办法根据 T 直接去反序列化。

下面我们说一点儿不太一样的。在 Kotlin 当中有一个关键字叫做 `reified`，还有一个叫做 `inline`，后者可以将函数定义为内联函数，前者可以将内联函数的泛型参数当做真实类型使用，我们先来看例子：

```
inline fun <reified T> Gson.fromJson(json: String): T{
    return fromJson(json, T::class.java)
}
```

这是一个 Gson 的扩展方法，有了这个之后我们就无须在 Kotlin 当中显式的传入一个 class 对象就可以直接反序列化 json 了。

这个会让人感觉到有点儿迷惑，实际上由于是内联的方法调用，T 的类型在编译时就可以确定的：

```
class Person(var id: Int, var name: String)

fun test(){
    val person: Person = Gson().fromJson("""{"id": 0, "name": "Jack"}""")
}
```

反编译之后：

```
public static final void test() {
    Gson $receiver$iv = new Gson();
    String json$iv = "{\"id\": 0, \"name\": \"Jack\"}";
    Person person = (Person)$receiver$iv.fromJson(json$iv, Person.class);
}
```

注意，在这里，**inline** 是必须的。

2. 型变

2.1 Java 的型变

如果 Parent 是 Child 的父类，那么 `List<Parent>` 和 `List<Child>` 的关系是什么呢？对于 Java 来说，没有关系。

也就是说下面的代码是无法编译的：

```
List<Number> numbers = new ArrayList<Integer>(); //ERROR!
```

不过 numbers 中可以添加 Number 类型的对象，所以我添加个 Integer 可以不呢？可以的：

```
numbers.add(1);
```

那么我要想添加一堆 Integer 呢？用 addAll 是吧？注意看下 addAll 的签名：

```
boolean addAll(Collection<? extends E> c);
```

这个泛型参数又是什么鬼？如果我把这个签名写成下面这样：

```
boolean addAll(Collection<E> c);
```

我想要在 numbers 当中 addAll 一个 ArrayList<Integer>，那就不可能了，因为我们说过， ArrayList<Number> 和 ArrayList<Integer> 是两个不同的类型，毛关系都没有。

? extends E 其实就是使用点协变，允许传入的参数可以是泛型参数类型为 Number 子类的任意类型。

当然，也有 ? super E 的用法，这表示元素类型为 E 及其父类，这个通常也叫作逆变。

2.2 Kotlin 的型变

型变包括协变、逆变、不变三种。

下面我们看看 Kotlin 是怎么支持这个特性的。Kotlin 支持声明点型变，我们直接看 Collection 接口的定义：

```
public interface Collection<out E> : Iterable<E> {
    ...
}
```

`out E` 就是型变的定义，表明 `Collection` 的元素类型是协变的，即 `Collection<Number>` 也是 `Collection<Int>` 的父类。

而对于 `MutableList` 来说，它的元素类型就是不变的：

```
public interface MutableCollection<E> : Collection<E>, MutableIterable<E> {
    ...
    public fun addAll(elements: Collection<E>): Boolean
    ...
}
```

换言之，`MutableCollection<Number>` 与 `MutableCollection<Int>` 没有什么关系。

那么请注意看 `addAll` 的声明，参数是 `Collection<E>`，而 `Collection` 是协变的，所以传入的参数可以是任意 `E` 或者其子类的集合。

逆变的写法也简单一些：`Collection<in E>`。

那么 Kotlin 是否支持使用点型变呢？当然支持。

我们刚才说 `MutableCollection` 是不变的，那么如果下面的参数改成这样：

```
public fun addAll(elements: MutableCollection<E>): Boolean
```

结果就是，当 `E` 为 `Number` 时，`addAll` 无法接类似 `ArrayList<Int>` 的参数。而为了接受这样的参数，我们可以修改一下签名：

```
public fun addAll(elements: MutableCollection<out E>): Boolean
```

这其实就与 Java 的型变完全一致了。

2.3 @UnsafeVariance

型变是一个让人费解的话题，很多人接触这东西的时候一开始都会比较晕，我们来看看下面的例子：

```
class MyCollection<out T>{
    fun add(t: T){ // ERROR!
        ...
    }
}
```

为什么会报错呢？因为 T 是协变的，所以外部传入的参数类型如果是 T 的话，会出现问题，不信你看：

```
var myList: MyCollection<Number> = MyCollection<Int>()
myList.add(3.0)
```

上面的代码毫无疑问可以编译，但运行时就会比较尴尬，因为 `MyCollection<Int>` 希望接受的是 Int，没想到来了一个 Double

对于协变的类型，通常我们是不允许将泛型类型作为传入参数的类型的，或者说，对于协变类型，我们通常是不允许其涉及泛型参数的部分被改变的。这也很容易解释为什么 `MutableCollection` 是不变的，而 `Collection` 是协变的，因为在 Kotlin 当中，前者是可被修改的，后者是不可被修改的。

逆变的情形正好相反，即不可以将泛型参数作为方法的返回值。

但实际上有些情况下，我们不得已需要在协变的情况下使用泛型参数类型作为方法参数的类型：

```
public interface Collection<out E> : Iterable<E> {
    ...
    public operator fun contains(element: @UnsafeVariance E): Boolean
    ...
}
```

比如这种情形，为了让编译器放过一马，我们就可以用 `@UnsafeVariance` 来告诉编译器：“我知道我在干嘛，保证不会出错，你不用担心”。

最后再给大家提一个点，现在你们知道为什么 `in` 表示逆变，`out` 表示协变了吗？

3. 通配符

在Java中，当我们不知道泛型具体类型的时候可以用`?`来代替具体的类型来使用，比如下面的写法：

```
Class<?> cls = numbers.getClass();
```

Kotlin也可以有类似的写法：

```
val cls: Class<*> = list.javaClass
val cls2: Class<*> = List::class.java
```

Kotlin可以根据`*`所指代的泛型参数进行相应的映射，下面是官方的说法：

- 对于`Foo <out T>`，其中T是一个具有上界TUpper的协变类型参数，`Foo <*>`等价于`Foo <out TUpper>`。这意味着当T未知时，你可以安全地从`Foo <*>`读取TUpper的值。
- 对于`Foo <in T>`，其中T是一个逆变类型参数，`Foo <*>`等价于`Foo <in Nothing>`。这意味着当T未知时，没有什么可以以安全的方式写入`Foo <*>`。
- 对于`Foo <T>`，其中T是一个具有上界TUpper的不型变类型参数，`Foo<*>`对于读取值时等价于`Foo<out TUpper>`而对于写值时等价于`Foo<in Nothing>`。

那么`*`在哪些场合下可以或者不可以使用呢？

我们来看几个例子：

```
val list = ArrayList<*>() // ERROR!
```

* 不允许作为函数和变量的类型的泛型参数！

```
fun <T> hello(args: Array<T>){
    ...
}

...
hello<*>(args) // ERROR!!
```

* 不允许作为函数和变量的类型的泛型参数！

```
interface Foo<T>

class Bar : Foo<*> // ERROR!
```

* 不能直接作为父类的泛型参数传入！

```
interface Foo<T>

class Bar : Foo<Foo<*>>
```

这是正确的。注意，尽管 * 不能直接作为类的泛型参数， Foo<*> 却可以，按照前面官方给出的说法，它在读时等价于 Foo<out Any> 写时等价于 Foo<in Nothing>

```
fun hello(args: Array<*>){
    ...
}
```

同样，这表示接受的参数的类型在读写时分别等价于 Array<out Any> 和 Array<in Nothing>

4. 其他

4.1 Raw 类型

Raw 类型就是对于定义时有泛型参数要求，但在使用时指定泛型参数的情况，这个只在 Java 中有，显然也是为了前向兼容而已。

例如：

```
List list = new ArrayList();
```

这类用法在 Kotlin 当中是不被允许的。上面的代码大致相当于：

```
val list = ArrayList<Any?>()
```

不过，在 Java 中，raw 类型可以有这种写法：

```
List<Integer> integers = new ArrayList<>();
List list = new ArrayList();
list = integers;
```

但 Kotlin 中，单纯的 `ArrayList<Any?>` 并不是协变的，所以下面的写法是错误的：

```
var list = ArrayList<Any?>()
val integers = ArrayList<Int>()
list = integers // ERROR!
```

Java，你这样做很危险呀。

4.2 泛型边界

在 Java 中，我们同样可以用 `extends` 为泛型参数指定上限：

```
class NumberFormatter<T extends Number>{
    ...
}
```

这表示使用时，泛型参数必须是 `Number` 及其子类的一种。

而在 Kotlin 中，写法与继承类似：

```
class NumberFormatter<T: Number>{  
    ...  
}
```

如果有多个上界，那么：

```
class NumberFormatter<T> where T: Number, T: Cloneable{  
    ...  
}
```

5. 小结

通过上面的讨论，其实大家会发现 Kotlin 的泛型相比 Java 有了更严格的约束，更简洁的表述，更灵活的配置，但背后的思路和具体的实现总体来说是一致的。

协程

- 轻量级线程：协程1
- 轻量级线程：协程2
- 深入理解 Kotlin Coroutine_1
- 深入理解 Kotlin Coroutine_2
- 深入理解 Kotlin Coroutine_2

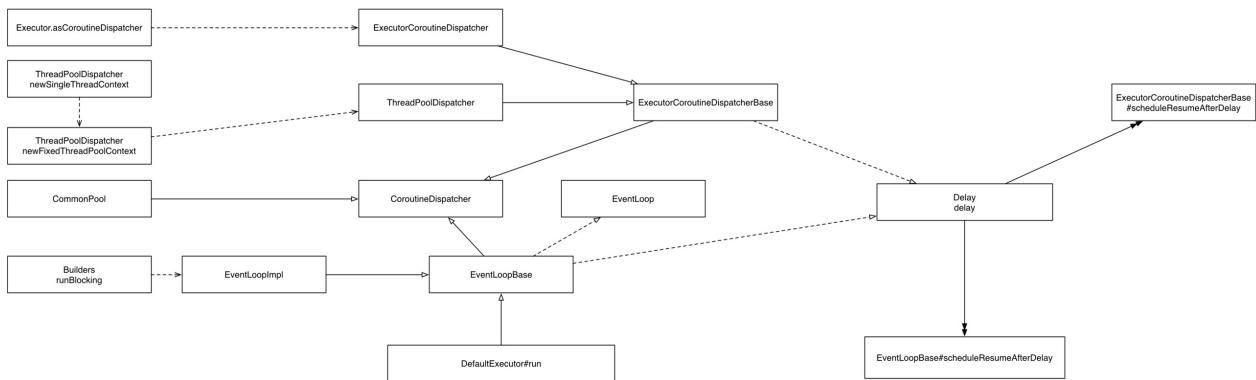
协程提供了一种避免阻塞线程并用更廉价、更可控的操作替代线程阻塞的方法：协程挂起。

Coroutine是编译器级的，Process和Thread是操作系统级的。

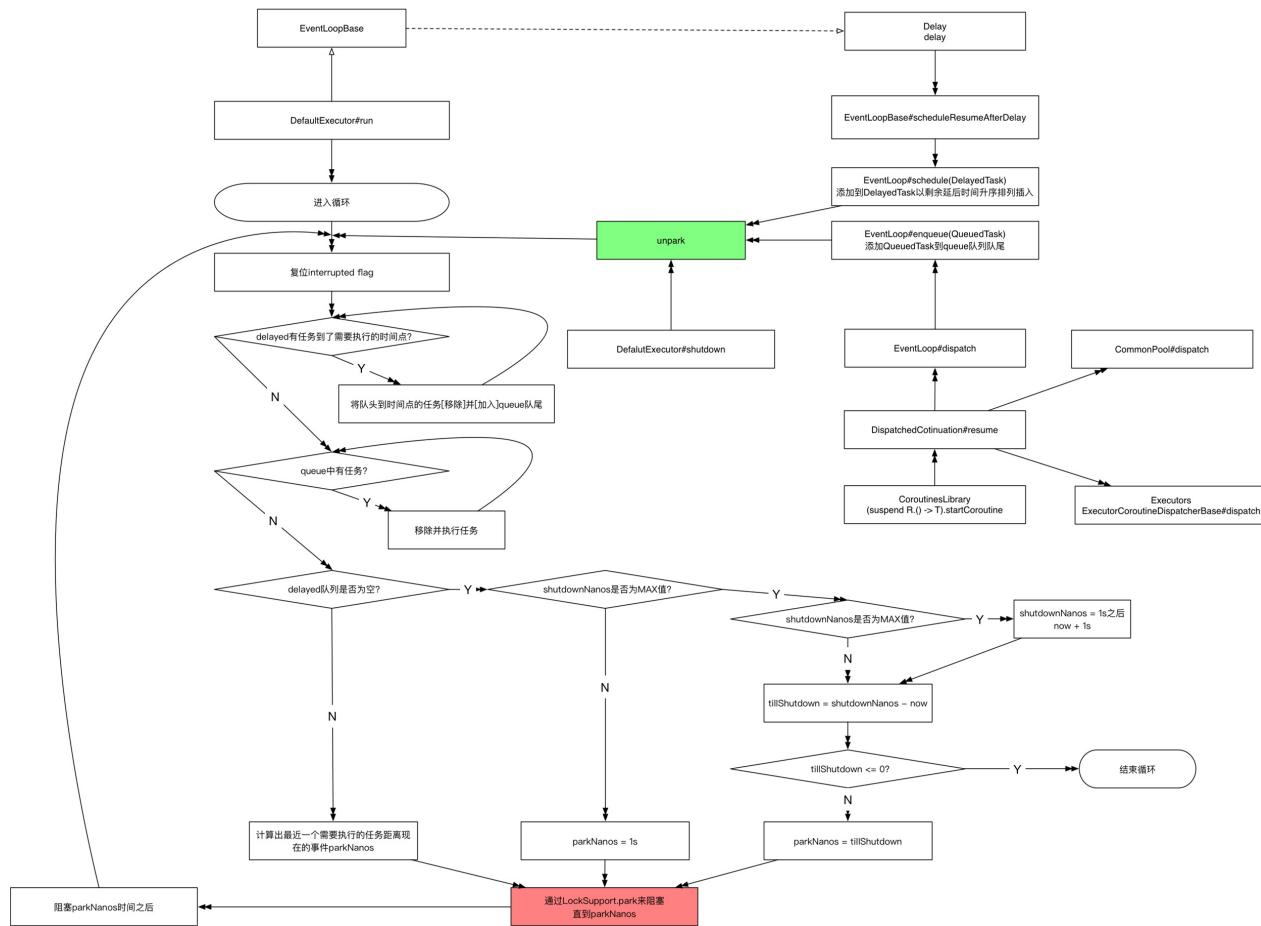
@RestrictsSuspension 注解

kotlinx.coroutines 框架

kotlin协程挂起



kotlin挂起分析



方法声明	功能描述
Job launch()	启动协程。创建运行在CoroutineContext中的coroutine，返回的Job支持取消、启动等操作，不会挂起父coroutine上下文；可以在非coroutine中调用
Deferred async()	启动一个协程，与其他协程并发地执行，异步，带返回值。创建运行在CoroutineContext中的coroutine，并且带回返回值(返回的是Deferred，我们可以通过await等方式获得返回值)
runBlocking()	启动一个协程，由它创建的协程会直接运行在当前线程上。创建一个coroutine并且阻塞当前线程直到代码块执行完毕，这个一般是用于桥接一般的阻塞式编程方式到coroutine编程方式的，不应该在已经是coroutine的地方使用。
run()	创建一个运行在 CoroutineContext 指定线程中的代码块，效果是运行在 CoroutineContext 线程中并且挂起父 coroutine上下文直到代码块执行完毕
delay()	非阻塞的sleep()，只是挂起协程本身
repeat()	执行重复任务
withTimeout()	设置协程超时时间
yield()	一个suspend方法，放弃执行权，并将数据返回
measureTimeMillis()	计算执行时间
produce()	

```
public suspend fun <T> run(
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend () -> T
): T = suspendCoroutineOrReturn sc@ { cont ->
    ...
}
```

- newSingleThreadContext()
- newFixedThreadPoolContext()

挂起函数

挂起 suspend，非阻塞

挂起函数suspend，挂起函数不能在普通函数中调用，必须在协程上下文中调用

CoroutineStart

协程启动选项

- DEFAULT
- LAZY

CoroutineContext

协程上下文，就是代码块执行在哪个场景。

- CoroutineContext.Key
- AbstractCoroutineContextElement
- EmptyCoroutineContext

CoroutineScope

- isActive
- context

CoroutineDispatcher

调度器

- CommonPool 共享线程池（一个有线程池的上下文）
- HandlerContext
- Unconfined
- dispatch(context: CoroutineContext, block: Runnable)

Job

后台任务，持有该协程的引用。非阻塞。

Job launch()

```

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
}

```

Deferred async()

方法声明	功能描述
isActive	
isCompleted	
cancel()	取消协程任务
join()	让主线程一直等到当前协程执行完毕再结束

Deferred(Job的子接口)，携带返回值

- DeferredCoroutine
- LazyDeferredCoroutine

方法声明	功能描述
await()	获取结果

Continuation

挂起的协程可以作为保持其挂起状态与局部变量的对象来存储和传递。这种对象的类型是 Continuation

方法声明	功能描述
context	CoroutineContext上下文
resume()	传给resume的参数会变成suspendCoroutine的返回值
resumeWithException()	

ContinuationInterceptor

拦截器，可以把协程的操作拦截

- interceptContinuation()

协程的基本操作

包级函数

方法声明	功能描述
createCoroutine()	创建协程
startCoroutine(Continuation)	开始协程
suspendCoroutine()	挂起协程

```
public inline suspend fun <T> suspendCoroutine(crossinline block
: (Continuation<T>) -> Unit): T =
    suspendCoroutineOrReturn { c: Continuation<T> ->
        val safe = SafeContinuation(c)
        block(safe)
        safe.getResult()
    }
```

suspendCoroutine的作用就是将当前执行流挂起，在适合的时机再将协程恢复执行，我们可以看到他的参数是一个lambda, lambda的参数是一个Continuation，Continuation了，它表示一段执行流。Continuation实例代表的执行流是从当前的suspension point开始，到下一个suspension point结束，当前的suspension point就是调用suspendCoroutine这一刻。

```

suspendCoroutine { cont ->
    // 如果本lambda在返回前, cont的resume和resumeWithException都没有
    // 调用
    // 那么当前执行流就会挂起, 并且挂起的时机是在suspendCoroutine之前
    // 就是在suspendCoroutine内部return之前就挂起了

    // 如果本lambda在返回前, 调用了cont的resume或resumeWithException
    // 那么当前执行流不会挂起, suspendCoroutine直接返回了,
    // 若调用的是resume, suspendCoroutine就会像普通方法一样返回一个值
    // 若调用的是resumeWithException, suspendCoroutine会抛出一个异常
    // 外面可以通过try-catch来捕获这个异常
}

```

在Kotlin内部, 协程被实现成了一个状态机, 状态的个数就是suspension point的个数+1(初始状态), 当前的状态就是当前的suspension point, 当调用resume时, 就会执行下一个Continuation.

Channel

- SendChannel
- ReceiveChannel
- Channel

方法声明	功能描述
isClosedForSend	
isClosedForReceive	
send()	
receive()	
close()	关闭通道

生成器

方法声明	功能描述
Sequence buildSequence()	
buildIterator()	
yield()	
yieldAll()	

状态机

对具有逻辑顺序或时序规律事件的一种描述方法。

- 有限状态机
- 两种状态机：Moore型和Mealy型

状态机的主要用途

协程与协程的区别

- 线程是抢占式的，协程是非抢占式的
- 线程是操作系统层面的，协程是编译器级的，应用层面，由虚拟机调度
- 线程是在kernel空间，协程是在用户空间
- 线程异步代码有大量的callback，阅读性差；协程代码阅读性更好，更优雅，以同步的代码实现异步功能

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

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。

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()
}
```

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, ")
})
```

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，协程将在池中的空闲线程上恢复。

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，它的主要作用是用来调度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：

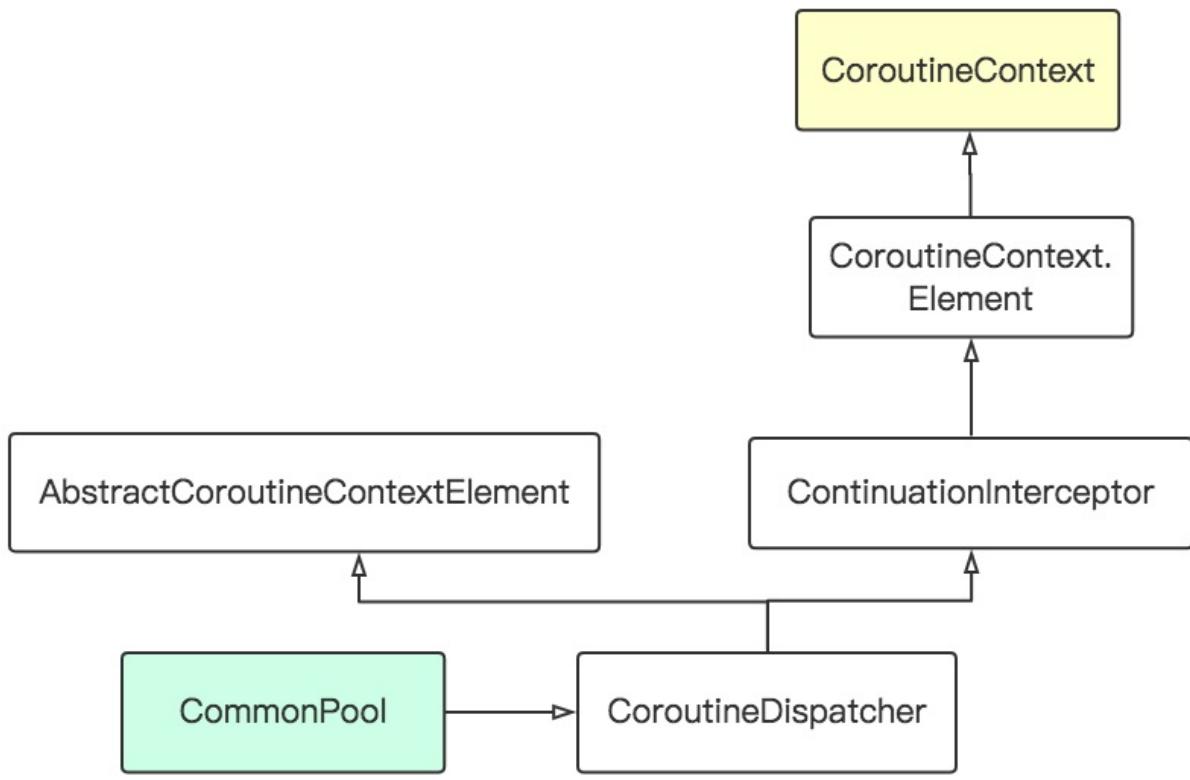
```

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：



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：

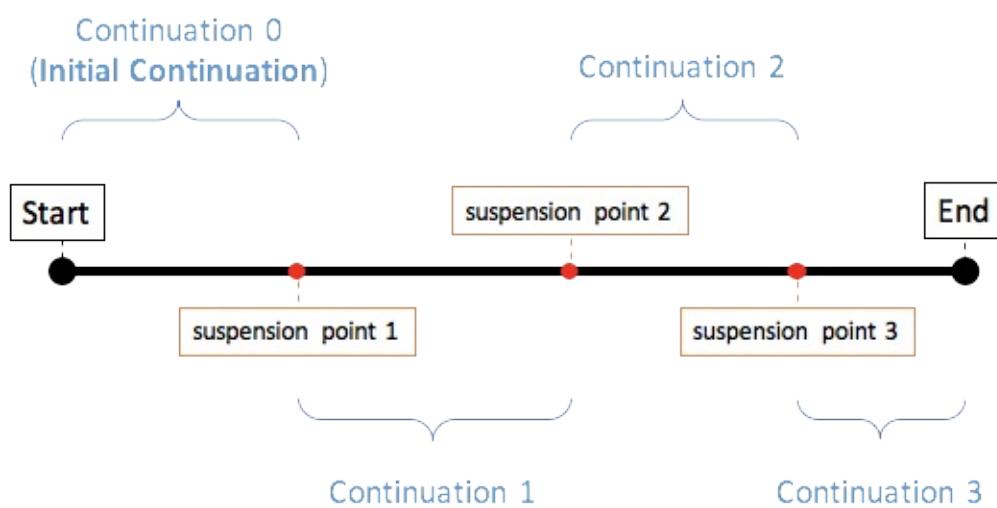
```

    /**
     * 错误反例：用线程调用协程 error
     */
    fun threadCoroutineDemo() {
        Thread({
            delay(3000L, TimeUnit.MILLISECONDS) // error, Suspend functions are only allowed to be called from a coroutine or another suspend function
            println("Hello, ")
        })
        println("World!")
        Thread.sleep(5000L)
    }

```

概念解释：Continuation 与 suspension point

协程的执行其实是断断续续的：执行一段，挂起来，再执行一段，再挂起来，...，每个挂起的地方是一个 `suspension point`，每一小段执行是一个 `Continuation`。协程的执行流被它的 "suspension point" 分割成了很多个 "Continuation"。我们可以用一条画了很多点的线段来表示：



其中的Continuation 0比较特殊，是从起点开始，到第一个suspension point结束，由于它的特殊性，又被称为 Initial Continuation。

协程创建后，并不总是立即执行，要分是怎么创建的协程，通过launch方法的第二个参数是一个枚举类型 `COROUTINESTART`，如果不填，默认值是 `DEFAULT`，那么协程创建后立即启动，如果传入 `LAZY`，创建后就不会立即启动，直到调用Job的 `start` 方法才会启动。

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，同时又使用了阻塞的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）。

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)
    }
}
```

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。

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 thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at com.easy.kotlin.LightWeightCoroutinesDemo.testThread(Ligh
tWeightCoroutinesDemo.kt:30)
    at com.easy.kotlin.LightWeightCoroutinesDemoKt.main(LightWei
ghtCoroutinesDemo.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在创建了太多的线程，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。

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。

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 canceled, 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。

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，或者检查是否有抛出 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 i
s completed: ${job.isCompleted}")

        val b1 = job.cancel() // cancels the job
        println("job cancel1: $b1")
        println("After Cancel, Job is alive: ${job.isActive} Job is
completed: ${job.isCompleted}")

        delay(30000L)

        val b2 = job.cancel() // cancels the job, job already cancel
d, 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.commonPoo
l-worker-1,5,main]
I'm sleeping 19 ... CurrentThread: Thread[ForkJoinPool.commonPoo
l-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。

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.isCompleted}")
        val b1 = job.cancel() // cancels the job
        println("job cancel1: $b1")
        println("After Cancel, Job is alive: ${job.isActive} Job is completed: ${job.isCompleted}")

        delay(3000L)
        val b2 = job.cancel() // cancels the job, job already canceled, 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。

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.isCompleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive} Job is completed: ${job.isCompleted}")
    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.isCompleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive} Job is completed: ${job.isCompleted}")
    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 ?")
}

```

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.
```

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.Timeo
utException: Timed out waiting for 3000 MILLISECONDS
    at kotlinx.coroutines.experimental.TimeoutExceptionCoroutine
    .run(Scheduled.kt:110)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunn
ableTask.invoke(EventLoop.kt:199)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunn
ableTask.invoke(EventLoop.kt:195)
    at kotlinx.coroutines.experimental.EventLoopImpl.processNext
Event(EventLoop.kt:111)
    at kotlinx.coroutines.experimental.BlockingCoroutine.joinBlo
cking(Builders.kt:205)
    at kotlinx.coroutines.experimentalBuildersKt.runBlocking(Bu
ilders.kt:150)
    at kotlinx.coroutines.experimentalBuildersKt.runBlocking$de
fault(Builders.kt:142)
    at com.easy.kotlin.CancellingCoroutineDemo.testTimeouts(Canc
ellingCoroutineDemo.kt:169)
    at com.easy.kotlin.CancellingCoroutineDemoKt.main(Cancelling
CoroutineDemo.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!")
}
```

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。

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，因为协程中的代码（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。

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 = Replace
    ewith("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

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。

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(coroutineContext) {
        // 父协程的上下文 : runBlocking coroutine
        println("coroutineContext: 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
```

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: [Stan
daloneCoroutine{Active}@5b646af2, CommonPool]
job1: 独立的协程上下文!
Thread[ForkJoinPool.commonPool-worker-3,5,main] ContextA2: [Stan
daloneCoroutine{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?

```

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。

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

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.RendezvousCha
nnel@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。

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
```

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。

管道

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 square(numbers: ReceiveChannel<Int>) = produce<Int>(CommonPo
ol) {
    for (x in numbers) send(x * x)
}
```

主代码启动并连接整个管线:

```
fun testPipeline() = runBlocking {
    val numbers = produceNumbers() // produces integers from 1 a
nd 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。

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) = pro
duce<Int>(context) {
    var n = start
    while (true) send(n++) // infinite stream of integers from s
tart
}
```

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
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103

fun numbersProducer(context: CoroutineContext, start: Int) = produce<Int>(context) {
    var n = start
    while (true) send(n++) // infinite stream of integers from start
}

fun filterPrimes(context: CoroutineContext, numbers: ReceiveChannel<Int>, prime: Int) = produce<Int>(context) {
    for (x in numbers) if (x % prime != 0) send(x)
}

fun producePrimesSequences() = runBlocking {
    var producerJob = numbersProducer(context, start: 2)

    while (true) {
        val prime = producerJob.receive()
        print("${prime}\n")
        producerJob = filterPrimes(context, producerJob, prime)
    }
}

```

Run com.easy.kotlin.ChannelsDemoKt

12630/	126311	12631/	126323	12633/	126341	126349	126359	12639/	126421	126433	126443	12645/	126461	126473	126481	12648/	✓
126491	126493	126499	126517	126541	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	129063	129083	✓
129089	129097	129113	129119	129121	129127	129169	129187	129193	129197	129209	129221	129223	129229				

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

```

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]
```

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，而是抽象于线程之上，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。

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。

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。

协程的内部机制

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。

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

深入理解 Kotlin Coroutine (一)

本文主要介绍 Kotlin Coroutine 的基础 API，有关 Koltinx.Coroutine 的内容，我们将在下一期给大家介绍。由于本人水平有限，如果大家有什么异议，欢迎直接抛出来跟我讨论。

1. 什么是 Coroutine

Coroutine 被翻译成了“协程”，意思就是要各个子任务协作运行的意思，所以大家一下就明白了它被创造出来是要解决异步问题的。

我们写 Java 的程序员，对线程更熟悉一些。线程是比进程更小一级的运行单位，它的调度由操作系统来完成，所以我们只管 new Thread 和 start，至于什么时候 run，什么时候 run 完，我们都没办法预见。

```
Thread t = new Thread(task);
t.start();
```

尽管有诸多不可控的因素，不过我们可以肯定的是起了一个新的线程并启动它之后，当前线程并不会受到阻塞。如果大家再往深处想想，CPU 在任意时刻运行什么进程及其线程，是操作系统决定的，但归根结底一个单线程的 CPU 在任一时刻只能运行一个任务。

那么协程呢？协程的调度是应用层完成的，比如我们说 Lua 支持协程，那么各个协程如何运行，这一调度工作实际上是 Lua 自己的虚拟机来完成的。这个调度与线程调度有着比较大的差别，线程调度是抢占式调度，很有可能线程 A 运行得美滋滋的，线程 B 突然把 CPU 抢过来，跟 A 说“你给我下去吧你”，于是线程 A 只能干瞪眼没办法；而协程的调度是非抢占式的，目前常见的各种支持协程的语言实现中都有 yield 关键字，它有“妥协、退让”的意思，如果一个协程执行到一段代码需要歇会儿，那么它将把执行权让出来，如果它不这么做，没人跟它抢。

在接触 Kotlin 的协程之前呢，我们先给大家看一个 Lua 的例子，比较直观：

```

function foo(a)
    print("foo", a)
    return coroutine.yield(2 * a)
end

co = coroutine.create(function (a, b)
    print("co-body", a, b)
    local r = foo(a + 1)
    print("co-body", r)
    local r, s = coroutine.yield(a + b, a - b)
    print("co-body", r, s)
    return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))

```

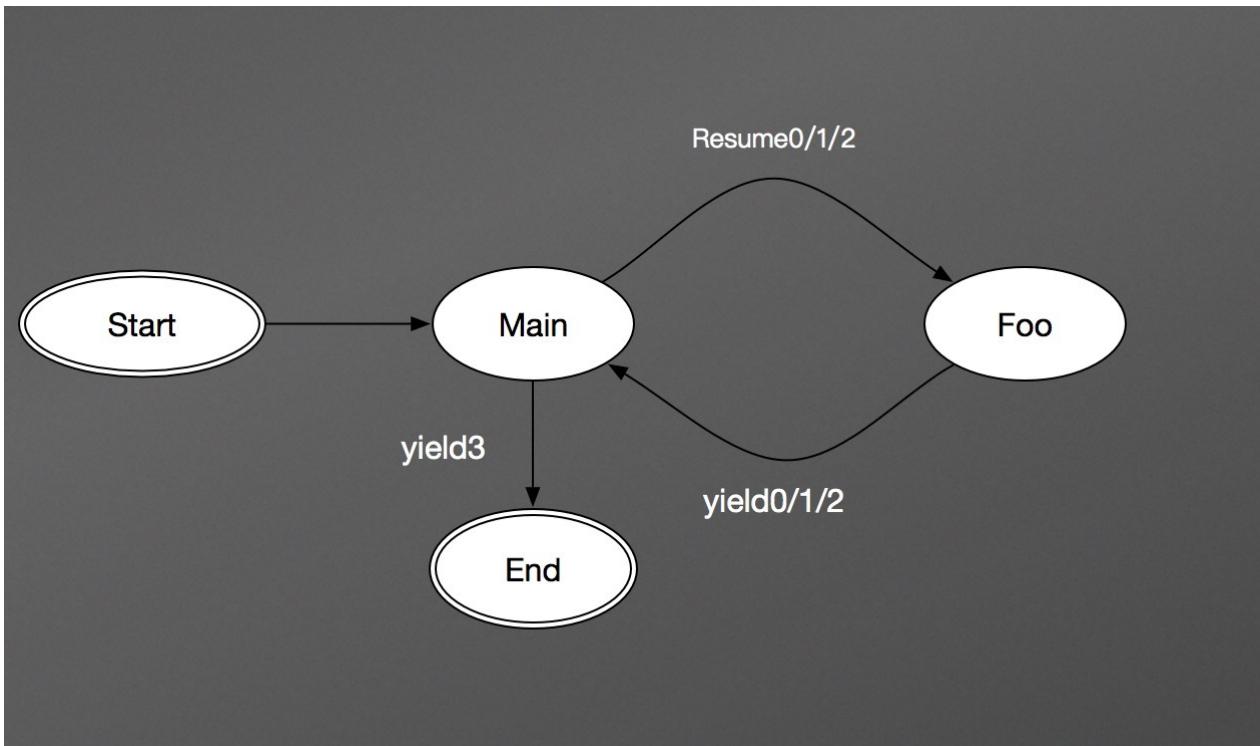
运行结果如下：

```

co-body    1    10
foo      2
main     true   4
co-body    r
main     true   11   -9
co-body    x    y
main     true   10   end
main     false  cannot resume dead coroutine

```

首先定义了一个 foo 函数，然后创建 coroutine，创建了之后还需要调用 resume 才能执行协程，运行过程是谦让的，是交替的：



图中数字表示第n次

协程为我们的程序提供了一种暂停的能力，就好像状态机，只有等到下一次输入，它才做状态转移。显然，用协程来描述一个状态机是再合适不过的了。

也许大家对 lua 的语法不是很熟悉，不过没关系，上面的例子只需要知道大概是在干什么就行：这例子就好像，main 和 Foo 在交替干活，有点儿像 AB 两个人分工协作，A 干一会儿 B 来，B 干一会儿，再让 A 来一样。如果我们用线程来描述这个问题，那么可能会用到很多回调，相信写 Js 的兄弟听到这儿要感到崩溃了，因为 Js 的代码写着写着就容易回调满天飞，业务逻辑的实现越来越抽象，可读性越来越差；而用协程的话，就好像一个很平常的同步操作一样，一点儿异步任务的感觉都没有。

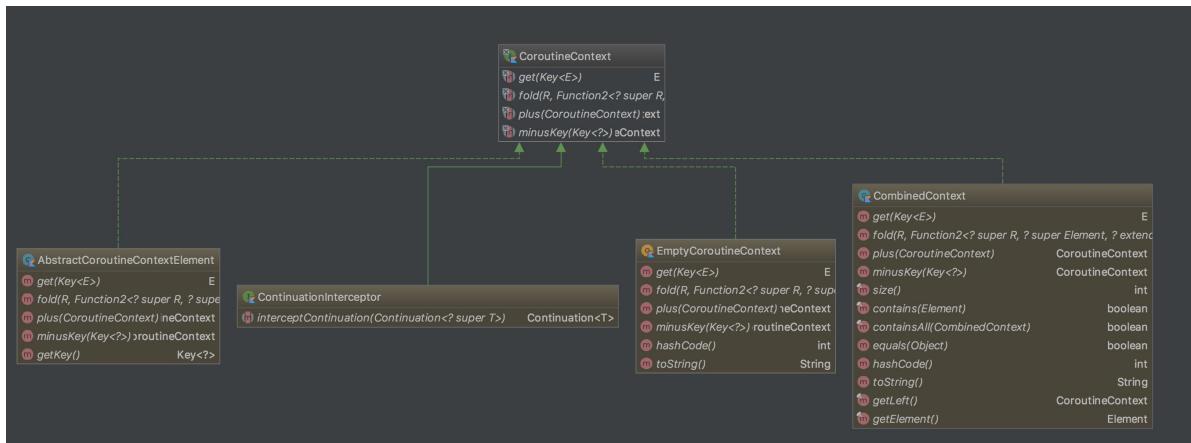
我们前面提到的协程的非抢占调度方式，以及这个交替执行代码的例子，基本上可以说说明协程实际上致力于用同步一样的代码来完成异步任务的运行。

一句话，有了协程，你的异步程序看起来就像同步代码一样。

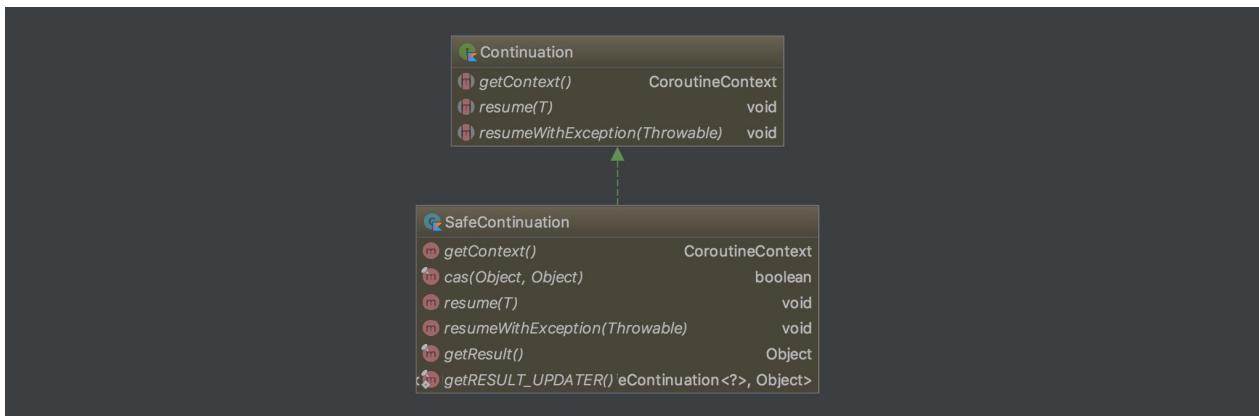
2. Kotlin 协程初体验

Kotlin 1.1 对协程的基本支持都在 Kotlin 标准库当中，主要涉及两个类和几个包级函数和扩展方法：

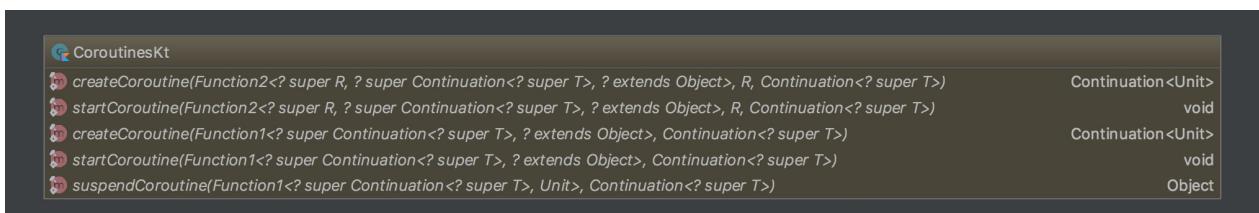
- CoroutineContext，协程的上下文，这个上下文可以是多个的组合，组合的上下文可以通过 key 来获取。EmptyCoroutineContext 是一个空实现，没有任何功能，如果我们在使用协程时不需要上下文，那么我们就用这个对象作为一个占位即可。上下文这个东西，不管大家做什么应用，总是能遇到，比如 Android 里面的 Context，JSP 里面的 PageContext 等等，他们扮演的角色都大同小异：资源管理，数据持有等等，协程的上下文也基本上是如此。



- Continuation，顾名思义，继续、持续的意思。我们前面说过，协程提供了一种暂停的能力，可继续执行才是最终的目的，Continuation 有两个方法，一个是 `resume`，如果我们的程序没有任何异常，那么直接调用这个方法并传入需要返回的值；另一个是 `resumeWithException`，如果我们的程序出了异常，那我们可以通过调用这个方法把异常传递出去。



- 协程的基本操作，包括创建、启动、暂停和继续，继续的操作在 `Continuation` 当中，剩下的三个都是包级函数或扩展方法：



这几个类和函数其实与我们前面提到的 Lua 的协程 API 非常相似，都是协程最基础的 API。

除此之外，Kotlin 还增加了一个关键字：suspend，用作修饰会被暂停的函数，被标记为 suspend 的函数只能运行在协程或者其他 suspend 函数当中。

好，介绍完这些基本概念，让我们来看一个例子：

```

fun main(args: Array<String>) {
    log("before coroutine")
    //启动我们的协程
    asyncCalcMd5("test.zip") {
        log("in coroutine. Before suspend.")
        //暂停我们的线程，并开始执行一段耗时操作
        val result: String = suspendCoroutine {
            continuation ->
            log("in suspend block.")
            continuation.resume(calcMd5(continuation.context[File
ePath]!!.path))
            log("after resume.")
        }
        log("in coroutine. After suspend. result = $result")
    }
    log("after coroutine")
}

/**
 * 上下文，用来存放我们需要的信息，可以灵活的自定义
 */
class FilePath(val path: String): AbstractCoroutineContextElemen
t<FilePath>{
    companion object Key : CoroutineContext.Key<FilePath>
}

fun asyncCalcMd5(path: String, block: suspend () -> Unit) {
    val continuation = object : Continuation<Unit> {
        override val context: CoroutineContext
            get() = FilePath(path)

        override fun resume(value: Unit) {

```

```

        log("resume: $value")
    }

    override fun resumeWithException(exception: Throwable) {
        log(exception.toString())
    }
}

block.startCoroutine(continuation)
}

fun calcMd5(path: String): String{
    log("calc md5 for $path.")
    //暂时用这个模拟耗时
    Thread.sleep(1000)
    //假设这就是我们计算得到的 MD5 值
    return System.currentTimeMillis().toString()
}

```

这段程序在模拟计算文件的 Md5 值。我们知道，文件的 Md5 值计算是一项耗时操作，所以我们希望启动一个协程来处理这个耗时任务，并在任务运行结束时打印出来计算的结果。

我们先来一段一段分析下这个示例：

```

/**
 * 上下文，用来存放我们需要的信息，可以灵活的自定义
 */
class FilePath(val path: String): AbstractCoroutineContextElement<FilePath>{
    companion object Key : CoroutineContext.Key<FilePath>
}

```

我们在计算过程中需要知道计算哪个文件的 Md5，所以我们需要通过上下文把这个路径传入协程当中。如果有多个数据，也可以一并添加进去，在运行当中，我们可以通过 Continuation 的实例拿到上下文，进而获取到这个路径：

```
continuation.context[FilePath]!!.path
```

接着，我们再来看下 Continuation：

```
val continuation = object : Continuation<Unit> {
    override val context: CoroutineContext
        get() = FilePath(path)

    override fun resume(value: Unit) {
        log("resume: $value")
    }

    override fun resumeWithException(exception: Throwable) {
        log(exception.toString())
    }
}
```

我们除了给定了 FilePath 这样一个上下文之外就是简单的打了几行日志，比较简单。这里传入的 Continuation 当中的 resume 和 resumeWithException 只有在协程最终执行完成后才会被调用，这一点需要注意一下，也正是因为如此，startCoroutine 把它叫做 completion：

```
public fun <T> (suspend () -> T).startCoroutine(completion: Continuation<T>
```

那么下面我们看下最关键的这段代码：

```

asyncCalcMd5("test.zip") {
    log("in coroutine. Before suspend.")
    //暂停我们的协程，并开始执行一段耗时操作
    val result: String = suspendCoroutine {
        continuation ->
        log("in suspend block.")
        continuation.resume(calcMd5(continuation.context[FilePath]!!?.path))
        log("after resume.")
    }
    log("in coroutine. After suspend. result = $result")
}

```

suspendCoroutine 这个方法将外部的代码执行权拿走，并转入传入的 Lambda 表达式中，而这个表达式当中的操作就对应异步的耗时操作了，在这里我们“计算”出了 Md5 值，接着调用 `continuation.resume` 将结果传了出去，传给了谁呢？传给了 `suspendCoroutine` 的返回值也即 `result`，这时候协程继续执行，打印 `result` 结束。

下面就是运行结果了：

```

2017-01-30T06:43:52.284Z [main] before coroutine
2017-01-30T06:43:52.422Z [main] in coroutine. Before suspend.
2017-01-30T06:43:52.423Z [main] in suspend block.
2017-01-30T06:43:52.423Z [main] calc md5 for test.zip.
2017-01-30T06:43:53.426Z [main] after resume.
2017-01-30T06:43:53.427Z [main] in coroutine. After suspend. result = 1485758633426
2017-01-30T06:43:53.427Z [main] resume: 1485758633426
2017-01-30T06:43:53.427Z [main] after coroutine

```

细心的读者肯定一看就发现，所谓的异步操作是怎么个异步法？从日志上面看，明明上面这段代码就是顺序执行的嘛，不然 `after coroutine` 这句日志为什么非要等到最后才打印？

还有，整个程序都只运行在了主线程上，我们的日志足以说明这一点了，根本没有异步嘛。难道说协程就是一个大骗子？

3. 实现异步

这一部分我们就要回答上一节留下的问题。不过在此之前，我们再回来回顾一下协程存在的意义：让异步代码看上去像同步代码，直接自然易懂。至于它如何做到这一点，可能各家的语言实现各有不同，但协程给人的感觉更像是底层并发 API（比如线程）的语法糖。当然，如果你愿意，我们通常所谓的线程也可以被称作操作系统级 API 的语法糖了吧，毕竟各家语言对于线程的实现也各有不同，这个就不是我们今天要讨论的内容了。

不管怎么样，你只需要知道，协程的异步需要依赖比它更底层的 API 支持，那么在 Kotlin 当中，这个所谓的底层 API 就非线程莫属了。

知道了这一点，我们就要考虑想办法来把前面的示例完善一下了。

首先我们实例化一个线程池：

```
private val executor = Executors.newSingleThreadScheduledExecutor
    Thread(it, "scheduler")
}
```

接着我们把计算 Md5 的部分交给线程池去运行：

```
asyncCalcMd5("test.zip") {
    log("in coroutine. Before suspend.")
    //暂停我们的线程，并开始执行一段耗时操作
    val result: String = suspendCoroutine {
        continuation ->
        log("in suspend block.")
        executor.submit {
            continuation.resume(calcMd5(continuation.context[File
ePath]!!.path))
            log("after resume.")
        }
    }
    log("in coroutine. After suspend. result = $result")
    executor.shutdown()
}
```

那么结果呢？

```
2017-01-30T07:18:04.496Z [main] before coroutine
2017-01-30T07:18:04.754Z [main] in coroutine. Before suspend.
2017-01-30T07:18:04.757Z [main] in suspend block.
2017-01-30T07:18:04.765Z [main] after coroutine
2017-01-30T07:18:04.765Z [scheduler] calc md5 for test.zip.
2017-01-30T07:18:05.769Z [scheduler] in coroutine. After suspend
    . result = 1485760685768
2017-01-30T07:18:05.769Z [scheduler] resume: 1485760685768
2017-01-30T07:18:05.769Z [scheduler] after resume.
```

我们看到在协程被暂停的那一刻，协程外面的代码被执行了。一段时间之后，协程被继续执行，打印结果。

截止到现在，我们用协程来实现异步操作的功能已经实现。

你可能要问，如果我们想要完成异步操作，直接用线程池加回调岂不更直接简单，为什么要用协程呢，搞得代码这么让人费解不说，也没有变的很简单啊。

说的对，如果我们实际当中把协程的代码都写成这样，肯定会被蛋疼死，我前面展示给大家的，是 Kotlin 标准库当中最为基础的 API，看起来非常的原始也是理所应当的，如果我们对其加以封装，那效果肯定大不一样。

除此之外，在高并发的场景下，多个协程可以共享一个或者多个线程，性能可能会要好一些。举个简单的例子，一台服务器有 1k 用户与之连接，如果我们采用类似于 Tomcat 的实现方式，一个用户开一个线程去处理请求，那么我们将要开 1k 个线程，这算是个不小的数目了；而我们如果使用协程，为每一个用户创建一个协程，考虑到同一时刻并不是所有用户都需要数据传输，因此我们并不需要同时处理所有用户的请求，那么这时候可能只需要几个专门的 IO 线程和少数来承载用户请求对应的协程的线程，只有当用户有数据传输事件到来的时候才去响应，其他时间直接挂起，这种事件驱动的服务器显然对资源的消耗要小得多。

4. 进一步封装

这一节的内容较多的参考了 Kotlin 官方的 [Coroutine Example](#)，里面有更多的例子，大家可以参考学习。

4.1 异步

刚才那个示例让我们感觉到，写个协程调用异步代码实在太原始了，所以我们决定对它做一下封装。如果我们能在调用 `suspendCoroutine` 的时候直接把后面的代码拦截，并切到线程池当中执行，那么我们就不用每次自己搞一个线程池来做事儿了，嗯，让我们研究下有什么办法可以做到这一点。

拦截...怎么拦截呢？

```
public interface ContinuationInterceptor : CoroutineContext.Element {  
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>  
  
    public fun <T> interceptContinuation(continuation: Continuation<T>): Continuation<T>  
}
```

我们发现，Kotlin 的协程 API 当中提供了这么一个拦截器，可以把协程的操作拦截，传入的是原始的 `Continuation`，返回的是我们经过线程切换的 `Continuation`，这样就可以实现我们的目的了。

```

open class Pool(val pool: ForkJoinPool)
    : AbstractCoroutineContextElement(ContinuationInterceptor),

ContinuationInterceptor {

    override fun <T> interceptContinuation(continuation: Continuation<T>)
        : Continuation<T> =
    PoolContinuation(pool,
        //下面这段代码是要查找其他拦截器，并保证能调用它们的拦截方法
        continuation.context.fold(continuation, { cont, elem
ent ->
            if (element != this@Pool && element is ContinuationInterceptor)
                element.interceptContinuation(cont) else con
t
        })))
}

private class PoolContinuation<T>(
    val pool: ForkJoinPool,
    val continuation: Continuation<T>
) : Continuation<T> by continuation {
    override fun resume(value: T) {
        if (isPoolThread()) continuation.resume(value)
        else pool.execute { continuation.resume(value) }
    }

    override fun resumeWithException(exception: Throwable) {
        if (isPoolThread()) continuation.resumeWithException(exc
eption)
        else pool.execute { continuation.resumeWithException(exc
eption) }
    }

    fun isPoolThread(): Boolean = (Thread.currentThread() as? Fo
rkJoinWorkerThread)?.pool == pool
}

```

这个 Pool 是什么鬼？我们让它继承 AbstractCoroutineContextElement 表明它其实就是我们需要的上下文。实际上这个上下文可以给任意协程使用，于是我们再定义一个 object：

```
object CommonPool : Pool(ForkJoinPool.commonPool())
```

有了这个，我们就可以把没加线程池的版本改改了：

```
fun main(args: Array<String>) {
    log("before coroutine")
    //启动我们的协程
    asyncCalcMd5("test.zip") {
        ...
    }
    log("after coroutine")
    //加这句的原因是防止程序在协程运行完之前停止
    CommonPool.pool.awaitTermination(10000, TimeUnit.MILLISECONDS)
}

...
fun asyncCalcMd5(path: String, block: suspend () -> String) {
    val continuation = object : Continuation<String> {
        override val context: CoroutineContext
            //注意这个写法，上下文可以通过 + 来组合使用
            get() = FilePath(path) + CommonPool
        ...
    }
    block.startCoroutine(continuation)
}

...
```

那么运行结果呢？

```

2017-01-30T09:13:11.183Z [main] before coroutine
2017-01-30T09:13:11.334Z [main] after coroutine
2017-01-30T09:13:11.335Z [ForkJoinPool.commonPool-worker-1] in c
oroutine. Before suspend.
2017-01-30T09:13:11.337Z [ForkJoinPool.commonPool-worker-1] in s
uspend block.
2017-01-30T09:13:11.337Z [ForkJoinPool.commonPool-worker-1] calc
md5 for test.zip.
2017-01-30T09:13:12.340Z [ForkJoinPool.commonPool-worker-1] afte
r resume.
2017-01-30T09:13:12.341Z [ForkJoinPool.commonPool-worker-1] in c
oroutine. After suspend. result = 1485767592340
2017-01-30T09:13:12.341Z [ForkJoinPool.commonPool-worker-1] resu
me: 1485767592340

```

我们看到程序已经非常完美的实现异步调用。显然，这种写法要比线程池回调的写法看上去顺理成章得多。

4.2 启动协程

在讨论完异步的封装后，有人肯定还是会提出新问题：启动协程的写法是不是有点儿啰嗦了啊？没错，每次构造一个 Continuation，也没干多少事儿，实在没什么必要，干脆封装一个通用的版本得了：

```

class StandaloneCoroutine(override val context: CoroutineContext
): Continuation<Unit> {
    override fun resume(value: Unit) {}

    override fun resumeWithException(exception: Throwable) {
        //处理异常
        val currentThread = Thread.currentThread()
        currentThread.uncaughtExceptionHandler.uncaughtException(
            currentThread, exception)
    }
}

```

这样就好办了，我们每次启动协程只需要针对当前协程提供特定的上下文即可，那么我们是不是再把启动的那个函数改改呢？

```
fun launch(context: CoroutineContext, block: suspend () -> Unit)
    =
    block.startCoroutine(StandaloneCoroutine(context))
```

有了这个，我们前面的代码就可以进一步修改：

```

fun main(args: Array<String>) {
    log("before coroutine")
    //启动我们的协程
    launch(FilePath("test.zip") + CommonPool) {
        log("in coroutine. Before suspend.")
        //暂停我们的线程，并开始执行一段耗时操作
        val result: String = suspendCoroutine {
            continuation ->
            log("in suspend block.")
            continuation.resume(calcMd5(continuation.context[File
ePath]!!?.path))
            log("after resume.")
        }
        log("in coroutine. After suspend. result = $result")
    }
    log("after coroutine")
    CommonPool.pool.awaitTermination(10000, TimeUnit.MILLISECONDS)
}

/**
 * 上下文，用来存放我们需要的信息，可以灵活的自定义
 */
class FilePath(val path: String) : AbstractCoroutineContextEleme
nt(Key) {
    companion object Key : CoroutineContext.Key<FilePath>
}
fun calcMd5(path: String): String {
    log("calc md5 for $path.")
    //暂时用这个模拟耗时
    Thread.sleep(1000)
    //假设这就是我们计算得到的 MD5 值
    return System.currentTimeMillis().toString()
}

```

运行结果自然也没什么好说的。

4.3 暂停协程

暂停协程这块儿也太乱了，看着莫名其妙的，能不能直白一点儿呢？其实我们的代码不过是想要获取 Md5 的值，所以如果能写成下面这样就好了：

```
val result = calcMd5(continuation.context[FilePath]!!.path).await()
```

毋庸置疑，这肯定是可以的。想一下，有哪个类可以支持我们直接阻塞线程，等到获取到结果之后再返回呢？当然是 Future 了。

```
suspend fun <T> CompletableFuture<T>.await(): T {
    return suspendCoroutine {
        continuation ->
        whenComplete { result, e ->
            if (e == null) continuation.resume(result)
            else continuation.resumeWithException(e)
        }
    }
}
```

我们干脆就直接给 CompletableFuture 定义一个扩展方法，当中只是用来挂起协程，并在结果拿到之后继续执行协程。这样，我们的代码可以进一步修改：

```

fun main(args: Array<String>) {
    log("before coroutine")
    //启动我们的协程
    val coroutineContext = FilePath("test.zip") + CommonPool
    launch(coroutineContext) {
        log("in coroutine. Before suspend.")
        //暂停我们的线程，并开始执行一段耗时操作
        val result: String = calcMd5(coroutineContext[FilePath]!
        !.path).await()
        log("in coroutine. After suspend. result = $result")
    }
    log("after coroutine")
    CommonPool.pool.awaitTermination(10, TimeUnit.SECONDS)
}

fun calcMd5(path: String): CompletableFuture<String> = Completab
leFuture.supplyAsync {
    log("calc md5 for $path.")
    //暂时用这个模拟耗时
    Thread.sleep(1000)
    //假设这就是我们计算得到的 MD5 值
    System.currentTimeMillis().toString()
}

... 省略掉一些没有修改的代码 ...

```

4.4 带有 Receiver 的协程

不知道大家注意到没有，4.3 的代码中有个地方比较别扭：

```

val coroutineContext = FilePath("test.zip") + CommonPool
launch(coroutineContext) {
    ...
    //在协程内部想要访问上下文居然需要用到外部的变量
    val result: String = calcMd5(coroutineContext[FilePath]!!.
    path).await()
    ...
}

```

在协程内部想要访问上下文居然需要用到外部的变量。这个上下文毕竟是协程自己的，自己居然没有办法直接获取到，一点儿都不自然。

其实这也不是没有办法，startCoroutine 其实还有一个带 receiver 的版本：

```
public fun <R, T> (suspend R.() -> T).startCoroutine(  
    receiver: R,  
    completion: Continuation<T>)
```

也就是说，我们不仅可以传入一个独立的函数作为协程的代码块，还可以将一个对象的方法传入，也就是说，我们完全可以在启动协程的时候为它指定一个 receiver：

```
fun <T> launch(  
    receiver: T,  
    context: CoroutineContext,  
    block: suspend T.() -> Unit)  
= block.startCoroutine(receiver, StandaloneCoroutine(context))
```

我们修改了 launch，加入了 receiver，于是我们的代码也可以这么改：

```
val coroutineContext = FilePath("test.zip") + CommonPool  
//需要传入 receiver  
launch(coroutineContext, coroutineContext) {  
    ...  
    //注意下面直接用 this 来获取路径  
    val result: String = calcMd5(this[FilePath]!!.path).await()  
    ...  
}
```

如果你觉得绝大多数情况下 receiver 都会是上下文那么上面的代码还可以接着简化：

```
fun launchWithContext(
    context: CoroutineContext,
    block: suspend CoroutineContext.() -> Unit)
= launch(context, context, block)
```

```
launchWithContext(filePath("test.zip") + CommonPool) {
    log("in coroutine. Before suspend.")
    //暂停我们的线程，并开始执行一段耗时操作
    val result: String = calcMd5(this[filePath]!!.path).await()
    log("in coroutine. After suspend. result = $result")
}
```

截止到现在，我们对最初的代码做了各种封装，这些封装后的代码可以在各种场景下直接使用，于是我们的协程代码也得到了大幅简化。另外，不知道大家有没有注意到，协程当中异常的处理也要比直接用线程写回调的方式容易的多，我们只需要在 Continuation 当中覆写 resumeWithException 方法就可以做到这一点。

5. 拿来主义：Kotlinx.Coroutine

[Kotlinx.Coroutine](#) 是官方单独发出来的一个 Coroutine 的库，这个库为什么没有随着标准库一并发出来，想必大家从其包名就能略窥一

二：`kotlinx.coroutines.experimental`，experimental，还处于试验阶段。不过既然敢随着 1.1 Beta 一并发出来，也说明后面的大方向不会太远，大家可以开始尝试其中的 API 了。

应该说，Kotlinx.Coroutine 做的事情跟我们在上一节做的事情是相同的，只不过它在这个方向上面走的更远。有关它的一些用法和细节，我们将在下一期给大家介绍。

6. 小结

本文主要对 Kotlin 1.1Beta 标准库的 Coroutine API 做了介绍，也给出了相应的示例向大家展示 Coroutine 能为我们带来什么。

协程是干什么的？是用来让异步代码更具表现力的。如果运用得当，它将让我们免于回调嵌套之苦，并发加锁之痛，使我们能够利用我们有限的时间写出更有魅力的程序。

深入理解 Kotlin coroutine (三)

前面有两篇文章介绍过协程，加上这篇，基本上介绍得差不多了。

- 深入理解 Kotlin coroutine (一)
- 深入理解 Kotlin coroutine (二)

上周在北京的活动上给大家分享了一下协程，发现大家对于协程最大的困惑是：我为什么要用它？

正好，前面一直想要再写篇协程的文章，这次让我们再来看看其中的一些问题。

我们为什么要用协程？

这个问题对于新接触协程的朋友来说确实很容易让人困惑，那么我们就来看看协程给我们带来什么吧。

- 协程是一种语法糖 协程的出现是来解决异步问题的，但它本身却不提供异步的能力，额这就很搞笑了，你来自于猴子与逗比吗？当然不是，协程某种意义上更像是一种语法糖，它为我们隐藏了异步调用和回调的细节，让我们更关注于业务逻辑的实现。
- 协程让代码更简洁 协程可以允许我们用同步代码的方式写出异步代码的功能。

```
async{
    val bitmap = await{ loadImage(url) }
}
```

这是一段 Android 的代码示例，请注意这个赋值操作，它实际上是切换到 UI 线程之后运行的，而 await 当中的 loadImage(url) 却是在 IO 线程中运行，所以我们一方面知道协程的异步功能是有线程在后面支持的，另一方面我们也知道异步线程回调可以用协程直接简化为一个简单的赋值。

- 协程让异步异常处理更方便 如果你的异步代码出现异常，通常你会在你的回调中加入一个 onError 来传递这个异常信息：

```
interface Callback{
    fun onError(e: CertainException)
    fun onSuccess(data: Data)
}
```

而在协程的支持下，我们只要按照同步代码的异常捕获方式进行捕获就可以了：

```
async{
    try{
        val bitmap = await{ loadImage(url) }
    }catch(e: CertainException){
        ...
    }
}
```

- 协程更轻量级 通常比较传统的服务器实现，一个用户请求接入后会给他开一个线程来等待和处理它的请求。这样是非常不经济的，因为这个用户有可能就是来逗你玩的，建立连接之后半天来一个字节，就这样你还得付出一个线程的代价来服务他，尴尬。如果用协程，那么就要轻量多了，因为协程只是一块儿内存，不像线程那样还要对应操作系统内核线程（印象中 Hotpot 的实现就是这样）。

用协程的理由我个人感觉也就这些了。一句话，协程是一种轻量级的方便操作异步代码的语法糖，而它本身不提供异步能力。

为什么说它是语法糖？

```
async{
    val bitmap = await{ loadImage(url) }
}
```

这样一句代码其实在编译完之后会生成一些新的类，`async` 后面的 `Lambda` 就会被编译成一个 `CoroutineImpl` 类的子类实例，大家只需要按照我经常提到的查看 `Kotlin` 字节码的方法去看看就知道了。

那么这个类究竟是个什么呢？

```
abstract class CoroutineImpl(
    arity: Int,
    @JvmField
    protected var completion: Continuation<Any?>?
) : Lambda(arity), Continuation<Any?> {

    @JvmField
    protected var label: Int = if (completion != null) 0 else -1

    private val _context: CoroutineContext? = completion?.context

    override val context: CoroutineContext
        get() = _context!!

    private var _facade: Continuation<Any?>? = null

    val facade: Continuation<Any?> get() {
        if (_facade == null) _facade = interceptContinuationIfNeeded(_context!!, this)
        return _facade!!
    }

    override fun resume(value: Any?) {
        processBareContinuationResume(completion!!) {
            doResume(value, null)
        }
    }

    override fun resumeWithException(exception: Throwable) {
        processBareContinuationResume(completion!!) {
            doResume(null, exception)
        }
    }

    protected abstract fun doResume(data: Any?, exception: Throw
able?): Any?
}
```

```

    open fun create(completion: Continuation<*>): Continuation<Unit> {
        throw IllegalStateException("create(Continuation) has not been overridden")
    }

    open fun create(value: Any?, completion: Continuation<*>): Continuation<Unit> {
        throw IllegalStateException("create(Any?;Continuation) has not been overridden")
    }
}

```

它首先是个 Lambda，这没毛病。其次，它是一个 Continuation，了解协程的朋友似乎要知道什么了，没错，这货与我们自己在启动协程时传入的 Continuation 实例是同样的东西，而且我们可以注意到构造方法当中有一个叫做 completion 的字段，不要惊讶，那就是我们传入的 Continuation。

实际上，我们通过编译器编译出来的字节码发现，create 方法当中会通过我们的这个 Lambda 表达式创建一个新的 CoroutineImpl 实例，而 doResume 这个抽象方法其实就是我们的 Lambda 表达式的内容了。

在这里我们还看到了 facade：

```

val facade: Continuation<Any?> get() {
    if (_facade == null) _facade = interceptContinuationIfNeeded(
        _context!!, this)
    return _facade!!
}

```

其中 `interceptContinuationIfNeeded` 当中就会处理各个拦截器，来完成线程调度或者其他操作，也就是说 facade 返回的 Continuation 实例就是经过类似下面这样的拦截器返回的实例了：

```

class UIContext :  
AbstractCoroutineContextElement(ContinuationInterceptor),  
ContinuationInterceptor {  
    override fun <T> interceptContinuation(continuation: Continuation<T>)  
        = ...  
}

```

协程是如何启动的？

我们再来看看协程是如何启动的。我们启动协程的时候通常会调用 `startCoroutine` 或者 `createCoroutine`，它们都会调用到一个方法：

```

public fun <T> (suspend () -> T).createCoroutineUnchecked(  
    completion: Continuation<T>  
) : Continuation<Unit> =  
    if (this !is CoroutineImpl)  
        ...  
    else  
        (this.create(completion) as CoroutineImpl).facade

```

我们已经知道我们的调用者实际上就是一个 `CoroutineImpl` 的实例，我们只需要关注 `else` 分支即可，这时候调用我们前面已经见到的 `create` 方法再创建一个 `CoroutineImpl` 实例，并把 `completion` 这个对象传给它，而 `facade` 实际上就会触发拦截器的操作，最终返回的就是经过拦截器处理之后的 `Continuation` 实例了。

创建完协程之后，就是启动的逻辑：

```

public fun <T> (suspend () -> T).startCoroutine(  
    completion: Continuation<T>  
) {  
    createCoroutineUnchecked(completion).resume(Unit)  
}

```

直接调用 `resume` 方法，结果怎样呢？由于拦截器都是我们自己提供的，比较直观，我们暂且不提，通常情况下，这个 `resume` 方法最终本质上调用的还是 `CoroutineImpl` 的 `resume` 方法：

```
override fun resume(value: Any?) {
    processBareContinuationResume(completion!!) {
        doResume(value, null)
    }
}
```

```
internal inline fun processBareContinuationResume(completion: Continuation<*>, block: () -> Any?) {
    try {
        val result = block()
        if (result !== COROUTINE_SUSPENDED) {
            completion as Continuation<Any?>.resume(result)
        }
    } catch (t: Throwable) {
        completion.resumeWithException(t)
    }
}
```

`processBareContinuationResume` 会首先触发一次 `doResume` 的调用，这个调用也就是我们自己的协程代码了，直到遇到第一个 `suspend` 调用，那么这时候协程就会被挂起，等待异步操作执行完成之后再来调用我们的 `resume/resumeWithException` 方法来通知我们数据回来了或者异常发生了。这个过程直到整个协程执行流程结束。

我们稍微关注一下 `Continuation` 接口：

```
public interface Continuation<in T> {
    public val context: CoroutineContext

    public fun resume(value: T)

    public fun resumeWithException(exception: Throwable)
}
```

再来看看我们通常的回调版本：

```
interface Callback<T>{
    fun onError(e: Exception)
    fun onSuccess(data: T)
}
```

除了协程上下文之外，剩下的两个方法与我们的回调又有什么区别呢？

小结

协程是什么？它就是用来简化你的异步回调代码的语法糖！

Kotlin与Java混合开发

- Kotlin 兼容 Java 遇到的最大的坑
- 勘误：15 Kotlin 与 Java 共存_2

Kotlin 与 Java 共存(一)

大家好，经过前面的课程，相信大家对 Kotlin 已经有了一个初步的认识，那么我们在项目中究竟应该怎么应用 Kotlin 呢？

首先，我们的项目基本上都是使用 Java 编写的，我们没有精力也没有必要去全部用 Kotlin 重写。其次，Java 作为一门历经考验的语言，自然有它存在的道理，Kotlin 作为崭露头角的新秀，自然也有它发力的方向，我们没必要舍弃哪个，而是让他们共存，各取所长。正像 Kotlin 的设计理念一样：

100% interoperable with Java™

比如，在编写 JavaBean 或者说数据结构类时，用 Java 写起来就要繁琐一些，这样的类我比较倾向于用 Kotlin 编写；编写上层代码时，经常会用到一些接口回调，通常这些回调也只有一个方法，于是我也倾向于使用 Kotlin 编写——而这在 Android 的 UI 层代码中体现的尤为明显；编写 TestCase 也是比较倾向于用 Kotlin 的，我最早认识 Kotlin 就是在编译 IntelliJ 的源码的时候，他们在两三年前的源码中就开始大量使用 Kotlin 编写 TestCase 了。

再比如，对于一些较为底层的框架性代码，涉及到较多比较 Tricky 的代码时，我更喜欢用 Java 写，因为 Java 对于变量类型、构造方法、泛型参数的限制要小一些等等。

总体来讲就是，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**

1. 属性

我们在 Kotlin 中编写了一个类，当中有一个属性：

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

我们在 Java 中要怎么使用呢？

```
Person person = new Person("benny");
System.out.println(person.getName());//benny
person.setName("andy");
System.out.println(person.getName());//andy
```

当然，在 Java 看来，name 这个成员是可以为 null 的，不过如果你胆敢设置一个 null 进去，信不信 Kotlin 跟你抱怨：

```
Exception in thread "main" java.lang.IllegalArgumentException: Parameter specified as non-null is null: method net.println.kt14.Person.setName, parameter <set->
    at net.println.kt14.Person.setName(Person.kt)
    at net.println.kt14.PersonMain.main(PersonMain.java:13)
```

其实，对于 null 的检查也没什么特别的，用我之前教大家的办法看下 Kotlin 的字节码：

```
INVOKESTATIC kotlin/jvm/internal/Intrinsics.checkNotNullNonNull (Ljava/lang/Object;Ljava/lang/String;)V
```

原来是调用了 Intrinsics.checkNotNull 方法，这方法很简单的，大家都可以说得出来：

```
public static void checkParameterIsNotNull(Object value, String paramName) {
    if (value == null) {
        throwParameterIsNullException(paramName);
    }
}
```

其实所有空安全的秘密都在这个类里面了，看到 Kotlin 的背后站着强大的 Java，我真是感到欣慰。

那么话说究竟能不能像在 Java 当中那样直接访问 Kotlin 的属性呢？

```
data class Person(var name: String, @JvmField var age: Int)
```

我们看到我们的 Person 有年龄啦，不过它与 name 不太一样，多了一个 @JvmField 的注解。

```
Person person = new Person("benny", 27);
System.out.println(person.getName() + " is " + person.age); //be
nny is 27
person.setName("andy");
person.age = 26;
System.out.println(person.getName() + " is " + person.age); //an
dy is 26
```

看，这就跟在 Java 当中的一样了。所谓有得必有失，用 @JvmField 标注的属性是不可以声明为 private 的，同时也是不可以像其他 Kotlin 属性那样直接自定义 getter 和 setter 的，你只能像 Java 那样自己写：

```
...
fun getAge(): Int = age

fun setAge(value: Int){
    age = value
}
...
```

仔细想想，这实在是多此一举了。

2. object

我们知道在 Kotlin 当中最简单的单例就是 object 了，可是 Java 并没有这样的特性。那我们要怎么访问 object 呢？

```
object Singleton{
    fun printHello(){
        println("Hello")
    }
}
```

如果大家看过之前的单例那一期，我们给大家看了 object 的字节码：

```
public final static Lnet/println/kt14/Singleton; INSTANCE
```

它实际上生成了一个静态实例 INSTANCE，所以我们在 Java 当中访问一个 Kotlin object 也就很简单了：

```
Singleton.INSTANCE.printHello();
```

3. 默认参数的方法

Kotlin 的方法可以有默认参数，这样可以省掉很多方法的重载（我们把重写继承自父类的方法叫做覆盖 override，名字相同参数不同的方法叫做重载 overload），可 Java 是没有这个特性的。Kotlin 的默认参数通常在 Java 当中是被忽略掉的，例如我们定义这样一个 Kotlin 类：

```
class Overloads{

    fun overloaded(a: Int, b: Int = 0, c: Int = 1){
        ...
    }

}
```

在 Java 中访问它的带有默认参数的方法时，必须传入完整的实参列表。

```
new Overloads().overloaded(0, 0, 1);
```

不过，现实也不是如此的残酷，如果我们稍作修改，事情就会好起来：

```
class Overloads{

    @JvmOverloads
    fun overloaded(a: Int, b: Int = 0, c: Int = 1){
        ...
    }

}
```

这样的话，在 Java 看来 overloaded 方法就多了两个小兄弟，分别是：

```
...
fun overloaded(a: Int, b: Int = 0)
fun overloaded(a: Int)
...
```

这样的话，我们在 Java 中也可以愉快地使用默认参数带来的便利了。

4. 包方法

Java 没有包方法，如果说有的话，倒也没那么多事儿了。Kotlin 的包方法会被默认编译到一个名为：包名+KT 的类当中，比如：

Package.kt

```
package net.println.kt14

fun printHello(){
    println("Hello")
}
```

编译完之后的字节码反编译成 Java 之后是：

```

package net.println.kt14;

import kotlin.Metadata;

@Metadata(
    mv = {1, 1, 1},
    bv = {1, 0, 0},
    k = 2,
    d1 = {"\u0000\b\n\u0000\n\u0002\u0010\u0002\n\u0000\u0001a\u0006\u0010\u0000\u001a\u00020\u0001``\u0006\u0002"},
    d2 = {"printHello", "", "production sources for module Kt14_main"},
)
public final class PackageKt {
    public static final void printHello() {
        String var0 = "Hello";
        System.out.println(var0);
    }
}

```

所以如果我们想要调用这样的方法，就可以像普通静态方法一样引用就可以了：

```

public class CallPackageMethod {
    public static void main(String... args) {
        PackageKt.printHello();
    }
}

```

5. 扩展方法

扩展方法实际上更像是一种语法糖，本质上其实是第一个参数为扩展类的实例而已。比如我们为 String 写了一个扩展方法：

ExtensionMethod.kt

```
fun String.isNotEmpty(): Boolean{
    return this != "" //注意 Kotlin 的字符串比较与 Java 的差别
}
```

我们在 Java 当中怎么访问这个方法呢？

```
public class CallExtensionMethod {
    public static void main(String... args) {
        System.out.println(ExtensionMethodKt.isNotEmpty("Hello"));
    }
}
```

6. Internal 的类和成员

我在之前的视频当中一直没有提到过的一个点：Kotlin 其实对访问权限做了调整。除了把默认访问权限改为 public 之外，还提供了一个模块内可见的 internal。一旦某个成员或者类被标记为 internal，那么模块之外的类是无法访问到这个成员或者类的，而对于模块内的其他成员或者类来说，它们则相当于 public——这个特性对于 sdk 开发者来说是相当友好的，举个例子，Android 源码中经常会有被标注为 @Hide 的成员，这些成员在 android.jar 当中不会有，不过他们却存在于 framework 的源码中，如果 Java 有 internal 这样的访问控制能力，那么 Android SDK 的开发者大可不必费尽周折搞出个 @Hide 注解并在打包 android.jar 的时候去掉这些成员。

介绍完 internal 之后，我们来看看模块内的 Java 代码如何访问 Kotlin 中的 internal 成员或者类，首先我们定义一个类用 internal 修饰。

```
internal class InternalClass {
    fun printHello(){
        println("Hello")
    }
}
```

在模块内写一些 Java 方法来访问它：

```

public class CallInternalClass {
    public static void main(String... args) {
        InternalClass internalClass = new InternalClass();
        internalClass.printHello();
    }
}

```

没有问题的，那模块外呢？我们把同样的代码放到了另外一个模块当中，这个模块依赖了我们之前的模块，发现这段 Java 代码仍然可以用，当然，Kotlin 只有在模块内才可以访问到 InternalClass 这个类。那这是不是说 Kotlin 在兼容 Java 方面有缺陷呢？结论不要下的太早，你就算能访问到这个类，又有什么用呢？

如果我们稍微改一下 InternalClass：

```

internal class InternalClass {
    internal fun printHello(){
        println("Hello")
    }
}

```

结果我们发现，Java 代码无论在模块内还是模块外，都无法访问到 printHello 方法（尽管编译器提示有个叫

`printHello$production_sources_for_module_Kt14_Kt14_main`，字节码当中也确实有这个方法，不过我们还是无法编译通过。

结论就是，**internal** 修饰符与 **Java** 的兼容性方法比较差，如果你的项目中有 **Java** 代码依赖 **Kotlin** 代码，那么被依赖的部分需要慎用 **internal**。

7. 小结

总体来讲，Java 依赖 Kotlin 的代码并不是件难事，绝大多数的场景我们并不会觉得二者混用在一起会有什么不舒服，相反，时间久了，你甚至会觉察不到二者的共存。这一期视频就到这里，下一期，我们讲如何在 Kotlin 当中调用 Java，谢谢大家。

Kotlin 与 Java 共存(二)

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。

1. 属性

如果 Java 类存在类似 setXXX 和 getXXX 的方法，Kotlin 会聪明地把他们当做属性来使用，例如：

```
public class DataClass {  
    private int id;  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```

Kotlin 的访问也非常简单：

```
fun main(args: Array<String>) {  
    val dataClass = DataClass()  
    dataClass.id = 0  
    println(dataClass.id)  
}
```

2. 空安全

空安全这个特性使得 Kotlin 对变量的值要求更严格了，由于 Java 变量通常没有这样的信息，因此 Kotlin 在访问 Java 变量或者 Java 方法时，变量、方法的参数和返回值的空与否由我们自己决定，编译器不再进行约束。例如定义这么一个 Java 类：

```
public class NullSafetyJava {  
    public String getData(){  
        return null;  
    }  
}
```

在 Kotlin 当中访问它：

```
fun main(args: Array<String>) {  
    val nullSafetyJava = NullSafetyJava()  
    val data = nullSafetyJava.data  
}
```

这个 data 的类型被称作『平台类型（Platform Type）』，它既可以为可空类型，也可以为不可空类型，这一切取决于我们自己的决定。

```
val dataCanBeNull: String? = data  
val dataCannotBeNull: String = data
```

这样在编译期是允许的，不过如果在运行时由于 data 为 null，那你就会遇到异常：

```
Exception in thread "main" java.lang.IllegalStateException: data  
must not be null  
at net.println.kt15.NullSafetyKt.main(NullSafety.kt:10)
```

如果继承 Java 类，父类的方法参数也是平台类型，也需要我们根据实际情况来判断是否可空。例如：

```
public abstract class NullSafetyAbsClass {  
    public abstract String formatDate(Date date);  
}
```

在 Kotlin 中继承这个类时，`formatDate` 方法的参数和返回值的类型我们根据情况写：

```
class NullSafetySubClass : NullSafetyAbsClass(){
    override fun formatDate(date: Date?): String? {
        return date?.toString()
    }

    fun main(args: Array<String>) {
        val nullSafetySubClass = NullSafetySubClass()
        val formattedDate: String? = nullSafetySubClass.formatDate(D
ate())
        println(formattedDate)
    }
}
```

这样直接写可控类型当然是没有问题的。如果你确信它可以不为空，那么你也可以直接写不可控类型：

```
class NullSafetySubClass : NullSafetyAbsClass(){
    override fun formatDate(date: Date): String {
        return date.toString()
    }

    fun main(args: Array<String>) {
        val nullSafetySubClass = NullSafetySubClass()
        val formattedDate: String = nullSafetySubClass.formatDate(Da
te())
        println(formattedDate)
    }
}
```

当然，对于这种情况，我们在 Java 也已经开始采用一些『曲线救国』的方式来弥补这一不足，比如采用 JetBrains 的 `@Nullable` 和 `@NotNull` 注解以及 Android 当中类似的注解支持，我们可以把我们的 Java 代码改写成这样：

```
public abstract class NullSafetyAbsClass {
    public abstract @NotNull String formatDate(@NotNull Date date);
}
```

3. 泛型

整体上来讲，Kotlin 的泛型与 Java 的没什么大的差别，Java 的？在 Kotlin 中变成了 *，毕竟？在 Kotlin 当中用处大着呢。另外，Java 泛型的上限下限的表示法在 Kotlin 当中变成了 out 和 in。

不过，由于 Java 1.5 之前并没有泛型的概念，为了兼容旧版本，1.5 之后的 Java 仍然允许下面的写法存在：

```
List list = new ArrayList();
list.add("Hello");
list.add(1);
for (Object o : list) {
    System.out.println(o);
}
```

而对应的，在 Kotlin 当中要用 List<*> 来表示 Java 中的 List——这本身没什么问题。那么我们现在再看一段代码：

```
public abstract class View<P extends Presenter>{
    protected P presenter;
}

public abstract class Presenter<V extends View>{
    protected V view;
}
```

这个其实是现在比较流行的 MVP 设计模式的一个简单的接口，我希望通过泛型来绑定 V-P，并且我可以通过泛型参数在运行时直接反射实例化一个 presenter 完成 V-P 的实例注入，这在 Java 当中是没有问题的。

那么在 Kotlin 当中呢？因为我们知道 View 和 Presenter 都有泛型参数的，所以我们在 Kotlin 当中就要这么写：

```
abstract class View<P : Presenter<out View<out Presenter<out Vie  
w<...>>>{  
    protected abstract val presenter: P  
}  
  
abstract class Presenter<out V : View<out Presenter<out View<...  
>>>>{  
    protected abstract val view: V  
}
```

一直写下去，最终陷入了死循环。编译器给出的解释是：

This type parameter violates the Finite Bound Restriction.

在 @zhangdatou 给我发邮件之前，我曾一直对此耿耿于怀，Kotlin 这么优秀的语言怎么还会有做不到的事情呢。原来不是做不到，而是我没有想到：

```
abstract class View<out P: Presenter<View<P>>>  
abstract class Presenter<out V: View<Presenter<V>>>  
  
class MyView: View<MyPresenter>()  
class MyPresenter: Presenter<MyView>()
```

实际上我们需要 View 的泛型参数 P 只要是 Presenter 的子类即可，并且要求这个泛型参数 P 本身对应的泛型参数也需要是 View 的子类，而这个 View 其实就是最初的那个 View，那么这个 View 的泛型参数当然就是 P 了。有点儿绕，但这么写出来明显感觉比 Java 安全靠谱多了。

4. Synchronized 和 volatile

在 Kotlin 当中，这两个关键字被削去了王位，成为平民。也许是设计者们觉得这二位作为关键字出现有点儿太重了，虽然不再身居要职，不过却也是不可或缺。

Synchronized 有两个版本，用于函数的版本是个注解：

```
var num: Int = 0

@synchronized fun count(){
    num++
}
```

用于代码块的则是一个函数：

```
var num: Int = 0

fun count(){
    synchronized(num){
        num++
    }
}
```

```
@kotlin.internal.InlineOnly
public inline fun <R> synchronized(lock: Any, block: () -> R): R
{
    @Suppress("NON_PUBLIC_CALL_FROM_PUBLIC_INLINE", "INVISIBLE_MEMBER")
    monitorEnter(lock)
    try {
        return block()
    }
    finally {
        @Suppress("NON_PUBLIC_CALL_FROM_PUBLIC_INLINE", "INVISIBLE_MEMBER")
        monitorExit(lock)
    }
}
```

volatile 的命运差不多，也变成了一个注解：

```
@Volatile var num: Int = 0
```

5. SAM 转换

看名字一头雾水，其实就是对于只有一个方法的 Java 接口，Kotlin 可以用一个 Lambda 表达式来简化它的书写，例如：

```
fun main(args: Array<String>) {
    val worker = Executors.newCachedThreadPool()
    worker.execute {
        println(System.currentTimeMillis())
    }
}
```

execute 方法传入了一个 Runnable 接口的实例，不过看样子却是一个 Lambda 表达式。不过这里也是有一个需要注意的点的，既然是『转换』，那么 Java 的 execute 方法接受到的实例就一定不是我们看到的这个 Lambda 表达式实例了，换句话说，我们就算每次传入同一个 Lambda 表达式实例，那么 Java 的 execute 方法收到的也并不是同一个对象。举个例子：

```
public class SAMInJava {
    private ArrayList<Runnable> runnables = new ArrayList<Runnable>();

    public void addTask(Runnable task){
        runnables.add(task);
        System.out.println("after add: " + task + ", we " + runnables.size() + " in all.");
    }

    public void removeTask(Runnable task){
        runnables.remove(task);
        System.out.println("after remove: " + task + ", only " + runnables.size() + " left.");
    }
}
```

然后我们在 Kotlin 当中这么写：

```

fun main(args: Array<String>) {
    val samInJava = SAMInJava()
    val lambda = {
        println("Hello")
    }

    samInJava.addTask(lambda)
    samInJava.addTask(lambda)
    samInJava.addTask(lambda)
    samInJava.addTask(lambda)

    samInJava.removeTask(lambda)
    samInJava.removeTask(lambda)
    samInJava.removeTask(lambda)
    samInJava.removeTask(lambda)
}

```

运行结果呢？

```

after add: net.println.kt15.SAMConversionKt$sam$Runnable$9855366
b@6ce253f1, we have 1 in all.
after add: net.println.kt15.SAMConversionKt$sam$Runnable$9855366
b@53d8d10a, we have 2 in all.
after add: net.println.kt15.SAMConversionKt$sam$Runnable$9855366
b@e9e54c2, we have 3 in all.
after add: net.println.kt15.SAMConversionKt$sam$Runnable$9855366
b@65ab7765, we have 4 in all.
after remove: net.println.kt15.SAMConversionKt$sam$Runnable$9855
366b@1b28cdfa, only 4 left.
after remove: net.println.kt15.SAMConversionKt$sam$Runnable$9855
366b@eed1f14, only 4 left.
after remove: net.println.kt15.SAMConversionKt$sam$Runnable$9855
366b@7229724f, only 4 left.
after remove: net.println.kt15.SAMConversionKt$sam$Runnable$9855
366b@4c873330, only 4 left.

```

每次 Java 的 add 和 remove 方法收到的都是不同的实例，所以 remove 方法根本没有起到作用。

6. 小结

除了这些之外还有一些细节，比如异常的捕获，集合类型的映射等等，大家可以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。

最后，作为这一系列视频的最后一集，我还想要告诉大家有关 Android 开发的视频可能会在年后开始筹备，公众号在后续的这段时间内会推送我为大家准备的 Kotlin 的一些有意思的文章，请大家继续关注，并与我互动，谢谢大家！

勘误：Kotlin 与 Java 共存 (2)

在过去推送过的第15期视频中，涉及到一个泛型对比的点，当时文中提到“对于循环引用泛型参数的情况，Kotlin 无法实现”的结论是有问题的。感谢 @zhangdatou 指正，对应部分的内容修改如下：

3. 泛型

整体上来讲，Kotlin 的泛型与 Java 的没什么大的差别，Java 的？在 Kotlin 中变成了 *，毕竟？在 Kotlin 当中用处大着呢。另外，Java 泛型的上限下限的表示法在 Kotlin 当中变成了 out 和 in。

不过，由于 Java 1.5 之前并没有泛型的概念，为了兼容旧版本，1.5 之后的 Java 仍然允许下面的写法存在：

```
List list = new ArrayList();
list.add("Hello");
list.add(1);
for (Object o : list) {
    System.out.println(o);
}
```

而对应的，在 Kotlin 当中要用 List<*> 来表示 Java 中的 List——这本身没什么问题。那么我们现在再看一段代码：

```
public abstract class View<P extends Presenter>{
    protected P presenter;
}

public abstract class Presenter<V extends View>{
    protected V view;
}
```

这个其实是在比较流行的 MVP 设计模式的一个简单的接口，我希望通过泛型来绑定 V-P，并且我可以通过泛型参数在运行时直接反射实例化一个 presenter 完成 V-P 的实例注入，这在 Java 当中是没有问题的。

那么在 Kotlin 当中呢？因为我们知道 View 和 Presenter 都有泛型参数的，所以我们在 Kotlin 当中就要这么写：

```
abstract class View<P : Presenter<out View<out Presenter<out Vi
ew<...>>>{
    protected abstract val presenter: P
}

abstract class Presenter<out V : View<out Presenter<out View<...
>>>>{
    protected abstract val view: V
}
```

一直写下去，最终陷入了死循环。编译器给出的解释是：

This type parameter violates the Finite Bound Restriction.

在 @zhangdatou 给我发邮件之前，我曾一直对此耿耿于怀，Kotlin 这么优秀的语言怎么还会有做不到的事情呢。原来不是做不到，而是我没有想到：

```
abstract class View<out P: Presenter<View<P>>>
abstract class Presenter<out V: View<Presenter<V>>>

class MyView: View<MyPresenter>()
class MyPresenter: Presenter<MyView>()
```

实际上我们需要 View 的泛型参数 P 只要是 Presenter 的子类即可，并且要求这个泛型参数 P 本身对应的泛型参数也需要是 View 的子类，而这个 View 其实就是最初的那个 View，那么这个 View 的泛型参数当然就是 P 了。有点儿绕，但这么写出来明显感觉比 Java 安全靠谱多了。

Kotlin 与 Java 混编

虽然 Kotlin 的开发很方便，但当你与他人协作时，总会碰到 Java 与 Kotlin 代码共存的代码项目。本章就教你如何优雅的实现 Kotlin 与 Java 混合编程。

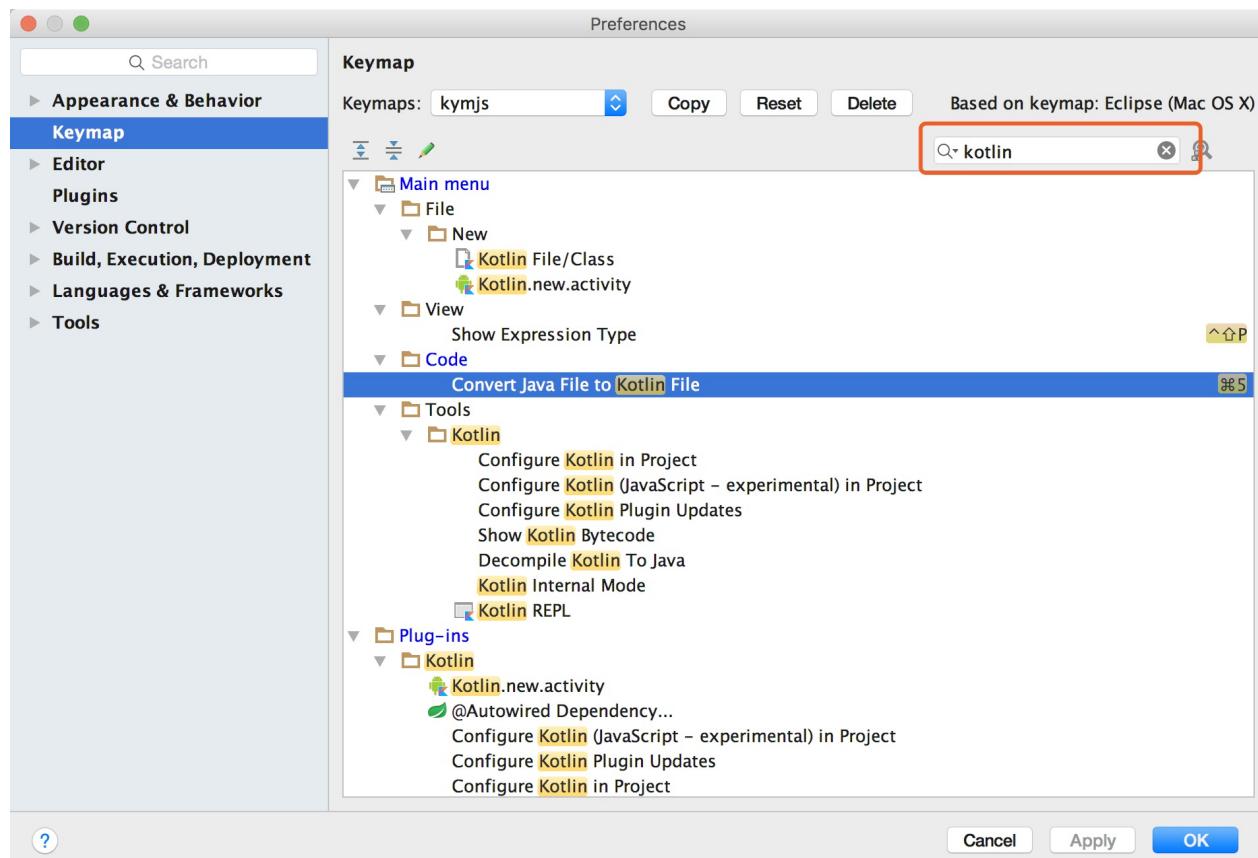
1. 直接转换

1.1 将 Java 转换为 Kotlin

如果你之前使用 Java 语言而没有 Kotlin 开发经验，不用担心，IntelliJ IDEA 会帮你一键转换，将 Java 代码转换成 Kotlin 代码(但是反过来就不行了)。

在 Mac 上，系统默认的快捷键为 `control+shift+command+K`，这个组合键实在有点反人类，建议你自定义一个你觉得舒服的快捷键。

快捷键可以通过你的编译器 keymap 中修改：`command+,` -> 搜索 `keymap` -> 右侧搜索 `kotlin`，可以查看到 `Convert Java File to Kotlin File` 项。



1.2 注意 Class 调用

在 Java 或 Android 开发中，经常会直接调用一个类的 Class 文件。但是当你用上文介绍的转换方法去转换 `XXX.class` 这样的代码时，是无法直接转换的(也许未来会修复这个问题，但目前你仍需要手动修改)。在 M13 之前，Java 中的 `XXX.class` 对应 Kotlin 代码中的 `JavaClass<XXX>` ，而 M13 之后写法已被改为 `XXX::class.java` 。

1.3 Android proguard 的坑

注：我们团队遇到过这样的一个坑，在 Android 开发的时候，如下代码会在混淆以后，发生异常

```
var str = some?.s?.d ?: ""
```

这段代码在正常debug模式编译运行完全正常，但是一旦执行混淆，就会发生所在函数被移除的现象。但是如果改写为以下写法就能正常运行：

```
var str = some?.s?.d ?: String()
```

猜想应该是 proguard 不知道如何处理这段代码，无法识别出最后两个引号是一个 `String`，最后直接将整个函数移除掉了。

同样的代码还有：

```
var list = some?.data?.list:MutableListof()
```

但是如下代码即使混淆后也是可以完全正常执行的

```
var s = some?.s ?: ""
var s = some.d ?: ""
var list = some?.data?.list:klist
var data = some?.data ?: return
```

1.4 开发 Android library 的建议

如果你是开发 Android library 程序，建议你不要使用 Kotlin 代码。因为作为 library，如果使用它的工程是纯 Java 完成的，引入后会额外增大 200k 左右大小，同时它有可能会造成某些情况下编译异常。

2. 在 Kotlin 中调用 Java 代码

2.1 返回 void 的方法

如果一个 Java 方法返回 void，对应的在 Kotlin 代码中它将返回 Unit。关于 Unit，本书将在 第五章 函数 部分着重讲解。现在你只需要知道在 Java 中返回为 void 的函数，在 Kotlin 中可以省略这个返回类型。

2.2 与 Kotlin 关键字冲突的处理

Java 有 static 关键字，在 Kotlin 中没有这个关键字，你需要使用 @JvmStatic 替代这个关键字。同样，在 Kotlin 中也有很多的关键字是 Java 中是没有的。例如 in, is, data 等。如果 Java 中使用了这些关键字，需要加上反引号(`)转义来避免冲突。例如

```
// Java 代码中有个方法叫 is()
public void is(){
    //...
}

// 转换为 Kotlin 代码需要加反引号转义
fun `is`() {
    //...
}
```

3 在 Java 中调用 Kotlin 代码

3.1 static 方法

上文已经提到过，在 Kotlin 中没有 static 关键字，那么如果在 Java 代码中想要通过类名调用一个 Kotlin 类的方法，你需要给这个方法加入 @JvmStatic 注解（这个注解只在 jvm 平台有用）。否则你必须通过对象调用这个方法。

```

StringUtils.isEmpty("hello");
StringUtils.INSTANCE.isEmpty2("hello");

object StringUtils {
    @JvmStatic fun isEmpty(str: String): Boolean {
        return "" == str
    }

    fun isEmpty2(str: String): Boolean {
        return "" == str
    }
}

```

如果你阅读 Kotlin 代码，应该经常看到这样一种写法。

```

class StringUtils {
    companion object {
        fun isEmpty(str: String): Boolean {
            return "" == str
        }
    }
}

```

`companion object` 表示外部类的一个伴生对象，你可以把他理解为外部类自动创建了一个对象作为自己的 `field`。与上面的类似，Java 在调用时，可以这样写：`StringUtils.Companion.isEmpty()`；(1.1以后可以省略中间的 Companion，写作 `StringUtils.isEmpty()`)

关于伴生对象，我们将在下一章 `类与对象` 详细讲解。

3.2 包级别函数

与 Java 不同，Kotlin 允许函数独立存在，而不必依赖于某个类，这类函数我们称之为包级别函数(Package-Level Functions)。

为了兼容 Java，Kotlin 默认会将所有的包级别函数放在一个自动生成的叫 `ExampleKt` 的类中，在 Java 中想要调用包级别函数时，需要通过这个类来调用。

当然，也是可以自定义的，你只需要通过注解 `@file:JvmName("Example")` 即可将当前文件中的所有包级别函数放到一个自动生成的名为 Example 的类中。

3.3 空安全性

在 Java 中，如果你调用的 kotlin 方法参数声明了非空类型，如果你在 Java 代码中传入一个空值，将在运行时抛出 `NullPointerException`。其内部原因在于 Kotlin 为每个非空类型加了断言，如果传入空值则会立刻抛出异常。同样，如果你使用 `null` 对象去调用一个 kotlin 方法，将会立刻抛出 `NullPointerException`（就算是调用普通 java 方法也是一样会抛出 `NullPointerException`）

Kotlin 兼容 Java 遇到的最大的“坑”

前言：上周我发了一篇文章[Kotlin 遇到 MyBatis：到底是 Int 的错，还是 data class 的错？](#)讲如何解决群里面一兄弟遇到的 data class 与 MyBatis 相克的问题，其中提到了几种外门邪道的方法，也提到了官方的解决思路，有些朋友看了之后还是不太明白，甚至紧接着就有小伙伴在使用 Realm 的时候遇到了类似的问题，看来，我还是得再写一篇来进一步告诉大家，这究竟是个什么问题，以及该如何面对它。

本文源码在 [Github : Kotlin-Tutorials](#) 这个项目当中，微信公众号无法添加外链，请大家点击“阅读原文”获取。

一个 Realm 的小例子

Realm 在 2016 年与 RxJava、Retrofit 这样的框架一起，在 Android 开发领域内着实小小的火了一把，如果大家对它不了解，没关系，传送门 biu ~ [Realm](#)

我们先按照官网的说明配置好 gradle 依赖，话说呀，这互联网发展这么快，新时代的框架一出来，逼格果断就体现在完善的构建和开发生态，你发布的东西还只是一个 jar 包，人家呢，早上了 maven 不说，还要搞几个 gradle 任务来方便你开发：

```
buildscript {  
    ext.kotlin_version = '1.1.1'  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:2.3.0'  
        //这里将 realm 的 gradle 插件加入 gradle 构建的运行时  
        classpath "io.realm:realm-gradle-plugin:3.0.0"  
    }  
}  
...
```

```

apply plugin: 'com.android.application'
apply plugin: 'realm-android' //应用插件，Realm 会在这里添加自己的一些构建任务
...

```

其实 gradle 插件开发也是一个很有意思的话题，如果大家有需要，我后面也可以写几篇文章介绍下（悄悄告诉你们，其实我早就想写了，这不是 kotlin 版的 gradle 还没有正式发布么！）。

有了这个我们就可以开始写个 Realm 的 demo 了。

小明：等等！我还有一事不明，你怎么不添加 realm 的依赖就要开始写 demo 了啊！

艾玛，要么说小明人家就是明白人呢，我前面写了一大堆，只不过是添加了 gradle 构建的依赖而已，而我们的程序想要使用 realm，必须依赖 realm 的运行时库才行。那这么说我是不是漏掉了什么？当然没有，怎么会呢，我这么聪（dou）明（bi）的人，我可是一步一步照着官网的步骤抄的！

其实呀，realm 的运行时依赖早在我们 apply plugin 的时候就已经添加进来的，realm-android 这个插件除了添加了一些它需要的 gradle 任务之外，也顺手帮我们把依赖添加了。嗯，就是酱紫，如果有那个同学学（xian）有（de）余（dan）力（teng），可以翻一翻 realm 插件的源码。

来来来，赶紧看 demo，不然有些人该内急了~

首先在 Application 当中初始化它：

```

class App : Application() {
    override fun onCreate() {
        super.onCreate()
        Realm.init(this)
        Realm.setDefaultConfiguration(
            RealmConfiguration.Builder()
                .deleteRealmIfMigrationNeeded()
                .schemaVersion(1)
                .build())
    }
}

```

定义一个 User 类：

```
data class User(@PrimaryKey var id: Int, val name: String) : RealmObject()
```

接着我们开始存数据和查数据啦：

```
add.setOnClickListener {
    Realm.getDefaultInstance().use {
        it.beginTransaction()
        val d = it.createObject(User::class.java, it.where(User::class.java).count())
        d.name = "User ${d.id}"
        it.commitTransaction()
    }
}

query.setOnClickListener {
    Realm.getDefaultInstance().use {
        it.where(User::class.java).findAll().map {
            Log.d(TAG, it.toString())
        }
    }
}
```

想得挺美，结果呢？编译不通过。

```
Error:A default public constructor with no argument must be declared in User if a custom constructor is declared.
```

无参构造方法

这就让我想到上周的文章，那篇文章里面我们其实就发现症结根本不是什么 Int 和 Integer，而是无参构造方法。JavaBean 是 Java 的一个概念，我其实甚至有些觉得 Java 的设计者们通过 JavaBean 这样的概念来弥补语言本身的缺陷——不管怎样，

JavaBean 是不能没有无参构造的，。

Kotlin 呢，语言层面就有类似于 JavaBean 的东西，那就是 data class，这俩孩子实在太像了，以至于大家经常把 data class 当做 JavaBean 来使。嗯，你信不信 Kotlin 的设计者也是这么想的呢？当然，用 data class 这样一个名正言顺的“亲儿子”数据类来替代 JavaBean 这么个语言层面没有任何支持和认可的“野孩子”，应该算是 JavaBean 莫大的荣幸了，可问题又出在 Java 语言本身构造方法滥用的潜在问题上了。在 Java 中，构造方法真心是一个很没有存在感的东西，大家总是根据自己的喜好来随意的定义很多个构造方法的版本，而最终忽视掉它们的内在联系，导致没有正常走完初始化逻辑的实例满天飞，这家伙如果是导弹，我估计也不需要解放军就可以直接把台湾给统一了。

说了这么多，我主要是想吐槽两个点：第一个就是 Java 本身语言设计层面几乎没有任何照顾到数据类的体现（可千万别提 clone 和 Serialize），第二个就是 Java 对其对象的实例化过程的把控太过于儿戏。

这两点呢，Kotlin 都做的很好，我现在写 Kotlin 经常被迫认真思考一个类该如何正确初始化，这显然对于我们的程序结构和逻辑梳理有莫大的好处。可是结果呢？Java 时代的那些框架们受不了了。Kotlin 背靠着 Java 这座大山，Java 就像它的父母一样，父母的观念再老再陈旧，Kotlin 也得做好自己该做的，一方面是向现在看来陈旧但在过去已经非常革命的观念致敬，另一方面嘛，如果 Java 不支持个几十万首付，Kotlin 能买得起房吗？

哇塞，我好能扯啊。

其实想要解决 default public constructor 这样的问题，Kotlin 官方已经想到了，那就是 noarg。嗯，我原以为我提一句 noarg 大家就会知道是什么了，看来是我想的简单了，毕竟这个东西在 1.0.6 才出来，当时我还在介绍这个版本的时候提到了它的使用方法，朋友们可能还没有接触过，没关系，下面我再贴一些写法，大家一看就明白：

首先你要做的就是定义一个注解：

```
annotation class PoKo
```

接着 gradle 配置一下脚本的依赖：

```
buildscript {  
    ...  
    dependencies {  
        ...  
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"  
        ...  
    }  
}
```

加了运行时环境，那么我们就可以使用 noarg 插件了：

```
apply plugin: "kotlin-noarg"  
  
noArg {  
    annotation("net.println.kotlin.realm.PoKo")  
}
```

配置完之后，PoKo 这个注解就有了超能力，所有被它标注的类在编译时都会生成一个无参的构造方法，于是我们给 User 加一个 PoKo 的注解：

```
@PoKo data class User(...) : RealmObject()
```

搞定，果断去编译一下！！

final 还是不 final，这是个问题

本来兴高采烈的以为不就是个无参构造的问题嘛，结果编译的时候又爆出了新的问题：

```
Error:(31, 61) error: cannot inherit from final User
```

好家伙，这究竟发生了什么。。原来 Realm 在编译的时候生成了一个类：

```
public class UserRealmProxy extends net.println.kotlin.realm.User  
    implements RealmObjectProxy, UserRealmProxyInterface
```

这个类要继承我这个 User 类，结果就报错了。

下面是理（che）论（dan）时间。我们说在 C++ 当中给合适的变量、函数参数、函数返回值甚至函数加上 const 是个好习惯，大家没有意见吧？同样的，Java 当中给那些不变的量、不能被继承的类、不能被覆写的方法加上 final 也是个好习惯，大家也没有意见吧？那么问题来了，大家有几个人这么干了？是不是不到万不得已，才懒得写那个 final 呢，五个字母呢，你是想累死宝宝啊？我就知道 Effective Java 这本书看了也白看，因为大家经常明知道什么是好习惯却还是要对着干，这个不是因为大家不喜欢好习惯，而是因为坚持好习惯需要成本！不瞒各位说，我中午为了坚持午休的好习惯，牺牲了跟组里面的小伙伴一起开黑上分的机会，还得装着拥护“人民ri报”关于“小学生打排位太坑”的评论，我容易么我。。

嗯，扯远了。还是说 final 的事儿，Kotlin 就做的很好，它默认所有的类、变量、方法都是 final 的，想要继承？来，过来申请我给你审批。。。你看，这样从根儿解决问题，我们再也不用为了坚持好习惯而发愁了，因为我们根本不需要坚持，难道你想要坚持坏习惯嘛？

可是 Java 及其框架们呢？原来到北京买房有钱就行，现在呢，商住都不让买了啊（什么？你说广州都不让卖了？）。那叫一个不适应，这可不是得闹事儿么。

Kotlin 官方考虑到 Java 帮它出首付买房的事儿，想了想算了，还是出个什么插件，解决下这个问题吧，于是 allopen 闪亮登场！allopen 的原理跟 noarg 极其类似，它是在编译器对指定的类进行去“final”化，你别看你写代码的时候 User 还是个 final 的类，不过编译成字节码之后这天呀可就变了。

关于 allopen 的使用，跟 noarg 简直不要太像，先定义一个注解：

```
annotation class PoKo // How old r U!
```

可以跟 noarg 公用同一个注解，也可以自己另外单独定义一个，这个不要紧。

接着 gradle 配置搞起：

```
buildscript {  
    ...  
    dependencies {  
        ...  
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"  
        ...  
    }  
}
```

接着就是应用插件，配置注解一气呵成：

```
apply plugin: "kotlin-allopen"  
  
allopen {  
    annotation("net.println.kotlin.realm.PoKo")  
}
```

编译运行~

ps：如果加了 allopen 和 noarg 之后编译仍然提示原来的错误，记得狠狠地 clean 一下才行哈。

认真脸：究竟什么是“坑”

前面说了 Kotlin 的两个“坑”，都是关于 data class 的。有人认为这么说 Kotlin 不公平，毕竟人家 Kotlin 也是可以写出下面的代码的：

```
class User{  
    var id: String? = null  
    var name: String? = null  
}
```

尽管你在为 Kotlin 打抱不平，不过如果你真要写这样的代码，我建议你还是用 Java 吧。你不属于 Kotlin。。。

Kotlin 这么美的语言，怎么能写这么丑陋的东西呢？这就好比有人说为什么空类型强转为非空类型一定要两个感叹号呢，用一个不就够了么，两个看起来好丑呀！

```
var user: User? = getUser()  
user!!.name = "小明" //小明，他们说你丑！
```

有人回答说：明明这就是丑陋的东西，为什么要美化？掩盖事物的本质只能让事情变得更糟糕！

我们用 Kotlin 企图兼容 Java 的做法，本来就是权宜之计，兼容必然带来新旧两种观念的冲突以及丑陋的发生，这么说来，我倒是更愿意期待 Kotlin Native 的出现了。

Kotlin 新特性

- Kotlin 1.0.6
- 喜大普奔！Kotlin 1.1 Beta 降临
- Kotlin 1.1 Beta 2 发布
- Kotlin 1.1：我们都上路
- Kotlin 1.1
- 快速上手 Kotlin 11招

Kotlin 1.0.6

我把所有文章和视频都放到了 [Github](#) 上，如果你喜欢，请给个 Star，谢谢

在上周二，Kotlin 1.0.6 发布啦！这次更新主要是工具更新和bug修复。本文的内容主要来自[官方博客](#)。

IDE 插件的更新

- try-finally 转换为 use()

通常我们在进行 IO 操作的时候，我们并不希望异常影响我们程序的执行，所以我们需要对异常进行捕获，但捕获的话我们也没有必要处理，所以写下来的就是下面的形式：

```
try{
    ... do something with "reader" ...
}finally{
    reader.close()
}
```

但这样写起来是不是非常的不流畅？如果用 use() 的话，简直一气呵成：

```
reader.use{
    reader -> ... do something with "reader" ...
}
```

所以，这次更新 Kotlin 的插件为我们带来了这样的自动转换功能：

```
fun File.lineCount(): Int {
    val reader = this.bufferedReader()

    try {
        return reader.lineSequence().count()
    }
    finally {
        reader.close()
    }
}
```

- 补全具名参数

通常我们在编写代码的时候，函数入参都会按照顺序一个一个传入，不过随着代码量的增加，特别是对于参数较多的函数，一长串的代码看上去会让我们感到非常的头疼。所以，这次更新 Kotlin 还为我们带来了自动补全具名参数的功能。

```
val times = (1..9).flatMap { f -> (1..9).map { s -> f to s } }
println(times.joinToString("\n", "Times table:\n") {
    "${it.first} x ${it.second} = " + it.first * it.second
})
```

- 删除空构造方法的声明
- 合并声明和赋值
- inline 函数的问题修复和调试工具的优化
- 提示、KDoc 和 Quick Doc 相关的较多问题的修复

Android 相关更新

- 支持 Android Studio 2.3 beta 1 和 Android Gradle Plugin 2.3.0-alpha3 及更新的版本
- 增加“Create XML resource”的提示
- Android Extensions support 这个功能可以让我们很方便的引用 XML 布局的 View，不过这需要我们主动启用 'kotlin-android-extensions' 才行。在过去，即使不启用这个插件，IDE 也会允许我们直接引用 XML 布局的 View，但这并不能正常编译，所以这次更新修复了这个问题：只有启用了这个插件，IDE 才会允许我们引用对应的 View。

- Android Lint 相关的问题修复。
- 增加 Suppress Lint 提示。

Kapt 优化

尽管还不能完全支持增量编译，相比 1.0.4，这次更新较大的提升了 Kapt 的性能。如果需要启用 Kapt，请在 gradle 当中启动它：`apply plugin: 'kotlin-kapt'`

All-open 插件

我们知道 Kotlin 的所有类及其成员默认情况下都是 final 的，也就是说你想要继承一个类，就要不断得写各种 open。刚开始看到这一特性的时候，觉得很赞，它对培养良好的编码意识非常有帮助，不过它也在某些情况下给我们带来麻烦，比如在一些大量依赖继承和覆写的 Java 框架的使用中。

这一次 Kotlin 提供了一个妥协的办法，主要某个类被某一个特定注解标注，那么这个类就默认所有成员统统 open，省得一个一个写了。有关 allopen 的讨论，大家可以参考[这里 KEEP](#)。

那么 allopen 如何使用呢？

```

buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"

allOpen {
    annotation("com.your.Annotation")
}

buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"

allOpen {
    annotation("com.your.Annotation")
}

```

那么所有被 com.your.Annotation 这个注解标注的类成员都会默认 open。除此之外，它还可以作为元注解使用：

```

@com.your.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open

```

Kotlin 还提供了 "kotlin-spring" 插件，其中包含了 spring 相关的所有注解，这样免得我们一个一个在 allopen 的配置中声明了。

No-arg 插件

如果大家看过我的视频，一定对我之前提到的“毁三观”的实例化有印象吧，附上[视频连接：12 Json数据引发的血案](#)，其中我们提到对于没有无参构造方法的 Kotlin 类，Gson 反序列化它们的时候，不知道如何实例化它们，只好用到了 `Unsafe` 这个类。听说 Java 9 要移除这个略显黑科技的类，如果是这样，Gson 是不是会被削弱呢？Java 的心我们还是不操了，从 Kotlin 1.0.6 开始，这个问题将得到一个比较好的解决。

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

// Or "kotlin-jpa" for the Java Persistence API support
apply plugin: "kotlin-noarg"

noArg {
    annotation("com.your.Annotation")
}
```

类似于 `allopen` 的使用方法，如果某个类需要无参默认构造方法，你只需要用上面声明好的注解 `com.your.Annotation` 标注即可。当然，这个生成的默认构造方法只能通过反射调用。

如何更新

我一直觉得虽然我们出生就被选择了 Hard 模式，但我们没啥感觉啊。可是最近一直访问国外的网站，感觉真的好困难，宽带换成了电信 100M，下载 Kotlin 的插件仍然跟小水管一样，真也是没谁了。为了方便大家把我下载的几个版本的插件放到[百度网盘](#)，供大家使用，请大家点击阅读原文获取下载地址。

小结

目前 Kotlin 1.0.x 的版本更新更侧重于稳定性和易用性，因此语言上的特性基本不会更新，主要集中于 IDE 插件和编译器插件。如果大家期待语言特性的更新，那我们就去关注一下 1.1 吧！

昨天 Kotlin 官方公布了一则消息：1.1 Beta 登场咯~用官方的原话就是说，这意味着正式版已经不再遥远，大家可以踊跃的试一试了~

这么好的消息，我自然不能藏着掖着，要赶紧给大家发一篇文章先介绍介绍。当然啦，我还没有来得及去仔仔细细尝试其中的各种特性，所以以下内容主要来自官网原文内容和 KEEP（也就是 Kotlin Evolution and Enhancement Process）的讨论。

嗯，以后每周我们也可以考虑附加一篇 1.1 的尝鲜日志，有螃蟹就让我先吃吧，大家看着我吃哇咔咔咔！

0. 注意事项

虽然官网没有提到，不过我还是想要强调一下，本文讨论的是 1.1 Beta ! 1.1 Beta ! ! 1.1 Beta ! ! ! 所以如果你装的是稳定版的 IDE 插件，并且在 gradle 当中配置的也是 1.0.6，就不要问『为什么我这个会报错』这样的问题了哦~~另外，如果真的会有这样的问题，我也不太建议大家去尝试 Beta 版，因为。。因为那样会比较打击人，你能忍受除了你自己写出bug之外还会有一些bug是因为语言本身有问题？

1. 特性概览

1.1 协程

不得不说，1.1 还是给我们带来了不少的惊。。喜的，额，虽然我们一直都知道，最重要的就是 协程（Coroutines），话说我最早听说这个概念的时候还是在 Lua 和 Golang 当中，这个 Java 并不支持的特性，一直与我忽远忽近，后来听说 Kotlin 1.1M 还是考虑支持协程，我着实兴奋了一把。好吧，现在终于又近了一步。

协程实际上是一个处理并发调度的概念，非阻塞的异步调用对于有过动态语言开发经历的朋友可能已是司空见惯，而对于我们这些深山老林中的 Java 程序员来说，还真是一個新鲜玩意儿。是不是厌倦了你的 new Thread 和 callback？我们看看异步加载数据的代码在 Kotlin 1.1 下要怎么写：

```

future {
    val original = asyncLoadImage("...original...") // creates a Future
    val overlay = asyncLoadImage("...overlay...") // creates a Future
    ...
    // suspend while awaiting the loading of the images
    // then run `applyOverlay(...)` when they are both loaded
    return applyOverlay(original.await(), overlay.await())
}

```

看上去好像没什么特别的，不过请注意，orginal 和 overlay 可都是异步加载的，applyOverlay 方法的调用会等到二者都准备好之后。

看上去非常符合人的思维，你不用再去过分 care 调用时序的问题，这些都交给虚拟机好了。有关协程的详细介绍，大家也可以看一下[这里Coroutines for Kotlin \(Revision 3\)](#)，我后续的推送当中也会逐渐推送相关内容。

1.2 类型别名

其实很多时候我们都需要一个类型来表示多种不同的含义，比如对于一个音乐播放器来说，在随机模式下，一首歌在播放列表的位置就有两种：实际位置，播放位置。这两种位置其实都是可以用一个 int 来表示的，于是我们的代码就会写成：

```

fun getPosition(): Int{
    ...
}

```

不知道大家发现什么没有，这个方法这么看上去我们根本搞不清楚它要返回的是那种类型的位置。如果有了类型别名呢？

```

typealias ListPosition = Int

fun getPosition(): ListPosition{
    ...
}

```

虽然返回的仍然是整型，不过含义却明确了许多。

当然，除了这个用途之外，对于 Lambda 表达式的表示也会简洁直接许多。

```
typealias Callable = ()->Unit
```

1.3 Bound callable references

说这个之前，我们先看个例子：

```
array.map(::println)
```

这个很显然是遍历 array，并调用函数 println 打印每一个元素。

那么问题来了，假设，现在我不用控制台打印了，我要将元素打印到 PDF 上面，于是：

```
array.map{  
    pdfPrinter.println(it)  
}
```

那么可不可以简化呢？

```
array.map(pdfPrinter::println)
```

也就是说函数引用进一步支持了绑定调用者（或者 receiver）的情况。

1.4 Data class 可以有爸爸啦

以前，data class 是不允许继承其他类的。不要问我为什么，没有父类这种事儿我更建议你去问问猴子。



你丫再瞎说小心我削你啊

不过，1.1 开始，Kotlin 的创造者们终于良心发现，允许 data class 有个爸爸，这种事情嘛，终归是一幅令人感动的画面啦。

```
data class Complex(...) : Something()
```

不过嘛，data class 这个家伙还是有点儿悲剧的，1.1 也没有允许他有子嗣，虽然可以认爹，不过没爹的时候就继续自己孤苦着吧。

1.5 放宽 sealed class 的子类定义限制

sealed class 的子类必须同时也是它的内部类，这即将成为过去。从 1.1 开始，只要跟 sealed class 定义在同一个文件内即可，嗯。。这个功能没有什么逻辑上的变化，不过写起来倒是省力一些。

1.6 万金油下划线

不知道大家有没有注意过 Golang 当中的下划线，

```
_ ,err := func()
```

其实就是一个占位符，如果我们对它不是很关心，那么还用得着费尽心思给它想个名字吗？当然不，Kotlin 1.1 也允许你这么干：

```
val (_ , x , _ , z) = listOf(1, 2, 3, 4)
print(x + z)
```

上面这个例子，我们只关心第2、4个值，1、3就直接忽视好了。

```
var name: String by Delegates.observable("no name") {
    _, oldValue, _ -> println("$oldValue")
}
```

这个例子则是说，三个形参我只关心第二个，那么其他两个退下吧~

哈哈，如果不需要我思考给变量起什么名字的话，我会不会轻松很多！

1.7 provideDelegate

不知道大家有没有想过，一旦给一个成员加了 Delegate，那么我就把这个成员的控制权交给了 Delegate，如果我想要监听一下这个成员的读写，我只能在 Delegate 当中做了。

```

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, property: KProperty<*>)
        : ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, property.name)
        ... // property creation
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}

```

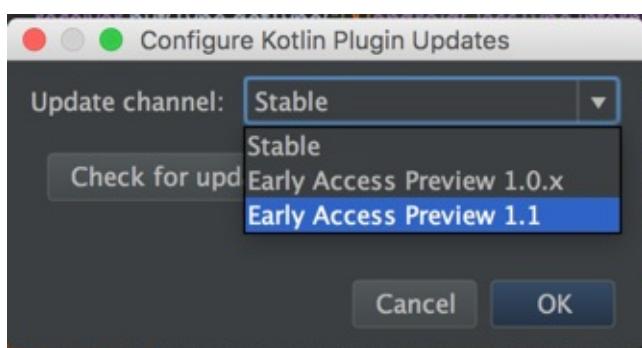
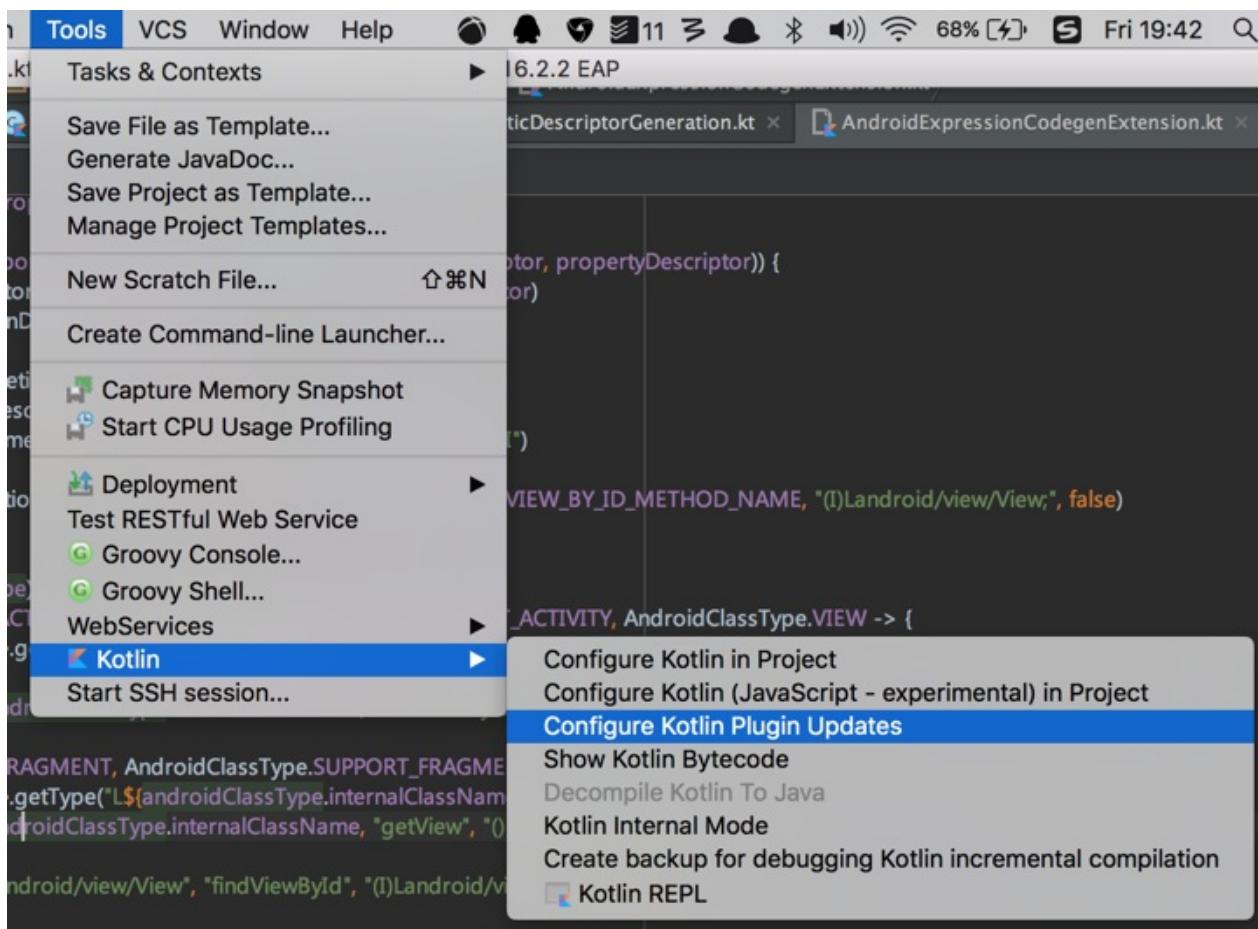
provideDelegate 可以允许我们在配置了 Delegate 之后也可以做一些织入代码的工作，这就为 Delegate 带来了更多的灵活性。

1.8 其他特性

当然，1.1 这么大的版本，特性肯定不止这些了，还有一些特性比如 DSL 作用域控制、为 Kotlin 集合增加 Java 8 方法的支持等等，我这里就不一一介绍了，大家也可以直接到[官博](#)一探究竟。

2. 如何配置 1.1 的环境

如果你也想吃螃蟹，那你必须更新你的 Kotlin 插件版本：



选中 1.1 之后点击 check for update 就可以了。

如果你用 gradle 或者 maven 构建自己的工程，那么你需要添加插件仓库：
<http://dl.bintray.com/kotlin/kotlin-eap-1.1>，选择 Kotlin 的版本为 1.1.0-beta-17 即可。

下面是 gradle 的配置：

```
buildscript {
    ext.kotlin_version = '1.1.0-beta-17'

    repositories {
        jcenter()

        maven{
            url "http://dl.bintray.com/kotlin/kotlin-eap-1.1"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

3. 小结

Kotlin 的迭代速度还是非常快的，稳定版本基本上每两个月更新一次，1.1 版的开发工作从年初正式版发布就逐渐为人所知，终于 Beta 来了，相信 1.1 正式版也不会让我们等太久。

后续我也会陆续推送一些文章介绍 1.1 的新特性，希望大家多多关注。

Kotlin 1.1 Beta 2 重点更新

1. 协程改包名风波

说真的，看到这个标题的时候我还挺兴奋，离 release 又近了一步。不过，看了这篇文章的时候，我就觉得也是醉醉的。发生了啥呢？

原来，协程相关的依赖统统被标记为 experimental 了，以前叫：

```
package kotlin.coroutines
```

现在呢？

```
package kotlin.coroutines.experimental
```

这意味着啥？意味着我们在这次更新之后，还得把原来的协程代码的包重新导入一遍，另外，如果你想使用协程，那么你还需要在配置当中启用它，例如 gradle 配置需要加入：

```
kotlin {  
    experimental {  
       .coroutines 'enable'  
    }  
}
```

你在升级所有的依赖的时候，确保它是兼容 1.1.0-beta-38 的，这一点很重要，不然等着报错吧！

话说，为啥要这么搞呢？按照官方的说法就是，协程这个特性目前已经实现的非常不错了，内置 API 非常少，灵活扩展性也强，不过他们觉得这个东西还有很大的潜力，也不能就这样作为最终版本给大家放出来，而作为实验特性交给大家使用呢，更多地还是希望大家能提提意见啥的。嗯，说实在的，协程这个特性真不是个小特性。

2. 兼容 1.0

话说，1.1 的编译器终于声称兼容 1.0 的源码了，这表明我们再也不用搞两个 IntelliJ 分别装 1.0 稳定版的插件和 1.1 Beta 版的插件了。

是的，就算你不用 1.1 的特性，你装 1.1 的插件，用 1.1 的编译器，写 1.0 的代码毫无压力！！

什么？你问我试了没？当然，我一直用最新的插件，折腾地挺苦的 TT，劝诸君还是装稳定版吧，吃螃蟹要做好心理准备~！

3. 小结

浏览了一下 1.1Beta 2 的主要特性，其实就是改改包名，修几个小 Bug，大的改动基本没有了。如果大家想要尽早上手 1.1 的特性，那么就从现在开始吧。

Kotlin 1.1：我们都路上

► 1.1 RC 来了，Release 还会远吗？



去年 2 月 19 日 1.0 发版，我记得此前看到论坛上面开始讨论 1.0 API 趋于稳定的话题，想想都还挺兴奋呢，一不留神，Kotlin 1.0 在稳定奔跑一年之后，也终于迎来下一个版本：就在前天（17 日），我留意到 Kotlin 官博发出了 1.1 RC(Release Candidate) 的消息，截止到目前，1.1 的所有开发工作都已经进入尾声，剩下的就只有我们期待的眼神了。

► 成长，一路有你



2016 这一年发生了挺多的事情，最重要的就是 Kotlin 终于有一个正式的版本，开始了按部就班的日子：每两个月一个正式版本以及一个 1.1 的里程碑版本。接着，我们也很快的发现我们期待的功能都悄悄的走进了 Kotlin 的更新日志，Kotlin 正一天

一天的成长起来。

不知道各位看官年龄几何，对我来说，C 和 C++ 太老，Java 虽然小了一些，可它诞生的时候我还在小学里面玩 Dos；而当我真正开始认识它们的时候却发现它们实在是太庞大了，有时候甚至觉得可怕，有时候更是没办法理解它们。它们是编程语言的长者，而我，只是一个年轻的程序员。

Kotlin 则不一样了。它出现的不算早，大概 2010 年的样子吧，那个时候我刚刚闲来无事考了个软考中级的“软件设计师”，一个在学校里面随处可见的“小黄书”背后的没有多大用处的考试，那个时候大概是我基础最好的时候吧。后来 Kotlin 逐渐成长起来，而我也逐渐脱离了象牙塔里的天真，冲向了努力干活历练自己的道路中间。

► 还记得，你与 Kotlin 的第一次相遇吗？



最早认识 Kotlin 还是在 15 年初，那时候我正热衷于编写 IntelliJ 的插件，尽管后面成果空空，不过一些个莫名其妙的后缀为 kt 的文件却着实让我头疼不已。尽管照着说明安装好 Kotlin 的插件，可因为版本不一致，始终无法编译过去，真是无可奈何啊。当时我还想，又有新语言啊，这可当真是学海无涯咯。

不过，随着接触的机会逐渐增多，我发现 Kotlin 解决了很多我对 Java 不满的问题，而它呢，却又全力支持着 Java 中的一切，始终让我们感觉不到我用了另外一种语言，语法的那点儿差异说真的比起不同语言背后的编程思维的差异来说，简直不值得一提。我在同时尝试了 groovy 和 scala 之后，觉得 Kotlin 才是我想要的，于是在 Kotlin 1.0 发布之际，我向 Bugly 公众号投稿文章：[Android 必备技能：最有可能接替 Java 的语言——Kotlin](#)，把我眼中的 Kotlin 向大家展现了出来。

► Kotlin 给你带来了什么？

去年上半年有段时间经常奔波在北京和深圳之间，工作节奏“日新月异”，项目似乎进入了一种莫名其妙的状态，而我自己则犹如置身死水，所幸我也是上过王者的人了，也不枉费我那一段时间在上面投入的一个个不眠之夜吧。



那段时间，我对项目的代码有我“自以为是的足够”的祸害的自由，于是我开始肆无忌惮的用 Kotlin 写一些模块，完全没有顾及合作开发的我们组唯一的妹子的感受。以及，这代码后来落入导师手中，我也算是坑他们不浅呐。不过，如果没有这段经历，或许我后来也不会有那么大的底气去在 10 月份斗胆录制视频，也自然不会去开公众号每周发几篇水文了。说来，得好好谢谢他们。



话说，自从摊上这么个事儿，我只好每天早晨 6 点起来或写写东西，或看看书，学点儿东西，每天节奏也极其规律，中午再也没精力跟小伙伴们组团王者了，因为我得睡一个小时。

年前我又在 Bugly 公众号发了一篇文章：[你为什么需要 Kotlin](#)，结果大家都说我是被代码耽误了的段子手，呃。。我想说你们说的很对！



你说的好有道理，我竟无言以对！

承蒙各位朋友厚爱，经常提及后续视频录制的问题，我正在尝试重新录制一套较为细致和基础的视频，目前讲义已经编写完毕，至于发布时间，那得看我啥时候录得完啦。

末了，建议大家有事儿没事儿也都写写，我最开始在公司内部写文章刷积分玩，后来发现自己写的东西经常需要复习；不仅如此，很多时候遇到一个问题，可能最终用某种方式解决了，你以为这个问题你是搞清楚了的，不过，一旦你企图将其形成文字，你就会发现问题的背后将会是更多的细枝末节。俗话说得好，好记性不如咱嘴的机械键盘啊！

“

Kotlin, More Than a Better Java.



Kotlin

长按识别二维码，关注Kotlin

遇见未来：Kotlin 1.1发布

上周一的文章里面提到 Kotlin 1.1 rc 了，还没正式发布，我在周三的时候把文章转到掘金以后，好多小伙伴告诉我，1.1 已经发布了。

tips: 本文有较多外链，公众号阅读时无法跳转，如有需要，请大家点击"阅读原文"。

1. 更新要点

1.1 Coroutine

1.1 最大的更新一定必须毫无疑问的要数 Coroutine 了，尽管在正式发版之前，Kotlin Team 突然虚了，决定把这个特性定为 Experimental，不过这似乎并没有改变什么。不就是改个包名么！！

早在春节放假那几天，我就在公众号连续两周发文介绍 Coroutine，本来还计划有第三篇的，不过开工以后个项目有点儿累，每天翻 Android 系统 C++ 层的代码翻到吐，也没精力去写第三篇文章，真是抱歉，如果大家有兴趣，可以参考前两篇：

[深入理解 Kotlin Coroutine \(一\)](#)

[深入理解 Kotlin Coroutine \(二\)](#)

其中，第一篇文章写于 experimental 之前，不过大家只要在包名当中加上 experimental 就没问题了。

Kotlin 的 Coroutine 实现主要分为两个层面，第一个层面就是标准库以及语言特性的支持，这里面主要包括最基本的 suspend 关键字以及诸如 startCoroutine 这样的方法扩展，上述第一篇文章对此做了详细的介绍。第二层面则主要是基于前面的基础封装的库，目前主要是 kotlinx.coroutines，其中封装了 runBlock、launch 这样方便的操作 Coroutine 的 api，这在第二篇文章做了详细地介绍。所以大家在了解 Coroutine 的时候，可以从这两个角度来入手，以免没有头绪。

我们再来简单说说 Coroutine 的运行机制。Coroutine 是用来解决并发问题的，它甚至有个中文名叫“协程”，它看上去跟线程似乎是并发问题的两种独立的解决方案，其实不然。要并发的执行任务，从根本上说，就是要解决 Cpu 的调度问题，Cpu 究

竟是如何调度，取决于操作系统，我们在应用程序编写的过程中用到的 Thread 也好，Coroutine 也好，本质上也是对操作系统并发 api 的封装。知道了这一点，我们再来想想 Thread 是如何做到两个线程并发执行的呢？Java 虚拟机的实现主要采用了对内核线程映射的方式，换句话说，我们通常用到的 Thread 的真正直接调度者可以理解为是操作系统本身。那我们在 Kotlin 当中支持 Coroutine 是不是也要把每一个 Coroutine 映射到内核呢？显然不能，不然那跟 Thread 还有啥区别呢？再者，Coroutine 的核心在 Co 上，即各个 Coroutine 是协作运行的，有一种“你唱罢来我登场”的感觉，就是说，Coroutine 的调度权是要掌握在程序自己手中的。于是，如果你去了解 kotlinx.coroutines 的实现，你就会发现 CommonPool 这么个东西，它不是别的，它的背后正是线程池。

线程是轻量级进程，而协程则是轻量级线程。

Coroutine 的出现让 Kotlin 如虎添翼，如果你之前在写 Go，Lua，python，或者 C#，这回 Java 虚拟机家族可不会让你失望了。自从有了协程，你也可以写出这样的代码：

```
val fibonacci = buildSequence {
    yield(1) // first Fibonacci number
    var cur = 1
    var next = 1
    while (true) {
        yield(next) // next Fibonacci number
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
...
for (i in fibonacci){
    println(i)
    if(i > 100) break //大于100就停止循环
}
```

序列生成器，记得我刚学 python 那会儿看到这样的语法，简直惊呆了。

```
val imageA = loadImage(urlA)
val imageB = loadImage(urlB)
onImageGet(imageA, imageB)
```

这样的代码也是没有压力的，看上去就如同同步代码一般，殊不知人家做的可是异步的事情呐。

协程的出现，让我们可以用看似同步的代码做着异步的事情。

这篇文章我们主要说说 1.1 的发版，Coroutine 的更多内容，建议大家直接点击前面的链接去读我的另外两篇文章~

1.2 JavaScript 支持

真是媳妇儿终于熬成婆，Js 终于被正式支持了。看官方的意思，他们已经用这一特性做了不少尝试，从 Kotlin 从头到尾写一个站点，似乎毫无压力，尽管类似反射这一的特性还没有支持，不过面包会有的嘛。

从我个人的角度来说，也可能我对前端了解太少吧，我觉得应用在前端比起移动端、服务端来说，Kotlin 的前景相对不明朗。我用 JavaScript 用得好好的，为啥要切换 Kotlin 呢？动态特性玩起来挺爽的，虽然回调写多了容易蛋疼，但这也不是不可以规避的。关于 Kotlin 开发前端这个问题，我需要多了解一下前端开发者的看法，相比他们是否愿意接触 Kotlin，我更关心有几个做前端的人知道这门语言。不瞒各位说，前几天跟一个支付宝客户端的大哥聊了一会儿，他问我这个 k o t 什么的，是干啥的。。。我当时在想，看来阿里人对 Kotlin 还不是很熟悉啊。

Whatever，Kotlin 现在都可以支持 node.js 了，还有什么不可能的呢？作为吃瓜群众，且让我观望一阵子。

1.3 中文支持

你放心，这一段内容你绝对在其他人那里看不到，因为没人会这么蛋疼。我前几天为了做一个案例用中文写了段代码，想着 Java 支持中文标识符，Kotlin 应该也问题不大。没曾想，写的时候一点儿问题的没，可编译的时候却直接狗带了。

```

package 中国.北京.回龙观

class G6出口{
    fun 下高速(){
        println("前方堵死, 请开启飞行模式 :)")
    }
}

fun main(args: Array<String>) {
    val 回龙观出口 = G6出口()
    回龙观出口.下高速()
}

```

注意，包名、代码文件名都是中文的，如果用 1.0.6 版编译，结果就是万里江山一片红哇。

```

Error: Kotlin: [Internal Error] java.io.FileNotFoundException: /Users/benny/temp/testKotlin/out/production/testKotlin/??/?/?/??/
G6???.class (No such file or directory)
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(FileInputStream.java:195)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)

    at kotlin.io.FilesKt__FileReadWriteKt.readBytes(FileReadWriteKt:52)
    at org.jetbrains.kotlin.incremental.LocalFileKotlinClass
    ...

```

注意到，汉字都变成了 ??，瞧瞧编译器那小眼神，真是看得我都醉了。

如果我们用 1.1 的编译器来编译这段代码，结果就可以正常输出：

前方堵死, 请开启飞行模式。

1.4 其他特性

1.1 还新增了不少特性，我在之前的一篇文章就做过介绍：[喜大普奔！Kotlin 1.1 Beta 降临~](#)

- tpyealias
- 绑定调用者的函数引用
- data class 可以继承其他类
- sealed class 子类定义的位置放宽
- _ 作为占位，替代不需要的变量
- provideDelegate

2. Kotlin 元年

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 上面的代码量都破千万了，使用 Kotlin 的公司也逐渐增多，除了 JetBrains 自己以外，我觉得在 Java 界比较有分量的就是 Square 了，如果 Google 能够稍微提一句 Kotlin，显然这个故事就会有另外一个令人兴奋的版本——好啦，不要 yy 啦。

据说，比较著名的主要有 Amazon Web Services, Pinterest, Coursera, Netflix, Uber, Square, Trello, Basecamp 这些公司将 Kotlin 投入了生产实践当中。国内资料较少，估计接触的人也不是很多，像百度、腾讯、阿里巴巴、滴滴、新美大、小米、京东这样的公司可能还没有太多的动力去将 Kotlin 应用到开发中，就算开始尝试，也多

是在 Android 开发上面试水；而敢于尝试 Kotlin 的，更多是没有什么历史包袱且富于创新和挑战精神的创业团队，对于他们而言 Kotlin 为开发带来的效率是非常诱人的。

说到这里，有两个令人兴奋的消息需要同步给大家：

- Gradle 开始尝试用 Kotlin 作为其脚本语言，目前已经发到了 0.4.0。这个真的可以有，groovy 虽然是一门很灵活的语言，不过写配置的时候如果没有 IDE 的提示，实在是太痛苦了。大家有兴趣也可以关注一下这个项目：[gradle-script-kotlin](#)
- Spring 5.0 加入 Kotlin 支持，Spring 的地位可想而知，Spring 为 Kotlin 站台，这分量还是很重的。

不知道 2017 年会发生什么，且让我们准备好爆米花饮料，拭目以待吧。

关于 Kotlin 的资料，英文版的图书已经出版了几本，主要有：

- [Kotlin in Action](#)：这部书已经有了纸质版，是官方自己人写的，算是一本比较权威的参考书了。
- [Kotlin for Android Developers](#)：这本书也算是老资历了，稍微看几眼你就会为 Kotlin 有趣的特性所吸引。另外，它还有一个[中文的翻译版本](#)
- [Modern Web Development with Kotlin](#)：这本书我没有读过，如果你需要用 Kotlin 开发 web 应用，它应该会给予你一些帮助。
- [Programming Kotlin](#)：这本书涉及内容非常全面，内容也算是言简意赅，快速入门 Kotlin 可以选择它。

除了图书以外，Kotlin 的首席布道师 Hadi Hariri 已经在 O'Reilly 上面发布了两套视频教程：

- [Introduction to Kotlin Programming](#)
- [Advanced Kotlin Programming](#)

里面有免费的几段，且不说内容怎么样，反正考验大家英语听力的时候到了，嗯，老爷子讲得还是很清楚的。

国内的资料，很少。除了有个别小伙伴写的一些博客之外，较为系统的学习资料几乎没有。也难怪大家都不知道它呢。也正是为了弥补这一空白，我在 16 年 10 月的时候开始每周 10 分钟的节奏连续录了 15 期视频，如果你有 Java 基础，那么看这些视频基本上可以让你知道 Kotlin 是怎么一回事了。

- [Kotlin 中文视频教程](#)

另外，如果你想要对 Kotlin 持续了解，建议你关注微信公众号 Kotlin，每周一推送的 Kotlin 的相关文章，基本上会覆盖了 Kotlin 的各种最新动态。也欢迎大家跟我交流开发中遇到的问题~

3. Kotlin 时代

1.1 的重要的更新其实就 Coroutine 以及 JavaScript 支持，毕竟 Kotlin 对 Java 的兼容支持已经做得非常不错了（别老提 apt 的事儿，1.0.4 之后的 kapt 不就基本上很好用了么）。别人问我，Kotlin 到底是写啥的，这个问题我通常说很官方的说，Kotlin 是一门运行在 Java 虚拟机、Android、浏览器上的静态语言，可是，Kotlin Team 的节奏已经让这句话显得要过时了。他们用短短几年时间搞出这么个全栈的语言，各方面特性都还很棒，然而他们并不能感到满足，他们已经开始走 C++ 的路线，也许 Kotlin Native 要不了多久就会出现了。

第一次听到这消息的时候，我瞬间就凌乱了，那感觉就好像王者荣耀里面队友选了大乔一样，秒回泉水加满血，秒回战场收人头啊。

前不久，我很荣幸地跟一位创业公司 CEO 坐下来聊理想，他问我的第一句话就是：你觉得 Kotlin 是未来么？我当时就蒙了，不得不说，他对 Kotlin 的期待跟 Kotlin Team 如出一辙呀。我当时实在不知道该怎么回答他，回来仔细想了想，答案其实也是有的。

十几年前，东家缺钱，急需投资，投资人坐下来“拷问”小马哥：“这个东西（指当时的 OICQ）怎么赚钱？”小马哥说自己只知道这个东西大家喜欢，但不知道向谁收钱。对于 Kotlin 来说，我只知道它好用，尽管大家都还看不太懂，不过它的时代正在悄悄的到来。

“

Kotlin, More Than a Better Java.



Kotlin

长按识别二维码，关注Kotlin

[TOC]

最近经常会收到一些“用 Kotlin 怎么写”的问题，作为有经验的程序员，我们已经掌握了一门或者多门语言，那么学 Kotlin 的时候就经常会有类似“‘再见’用日语怎么说？”、“‘你好’用西班牙语怎么说？”的问题，所以我决定把一些常用的语法对照列举出来，如果大家熟悉 Java，那么快速上手 Kotlin 会变得非常地容易。

这篇文章主要是写给需要快速上手 Kotlin 的 Java 程序员看的，这时候他们关注的是如何 Kotlin 写出类似某些 Java 的写法，所以本文基本不涉及 Kotlin 的高级特性。

1. 如何定义变量

Java 定义变量的写法：

```
String string = "Hello";
```

基本等价的 Kotlin 定义变量的写法：

```
var string: String = "Hello"
```

Java 定义 final 变量的写法：

```
final String string = "Hello";
```

注意到前面的是一个编译期常量，Kotlin 当中应该这么写：

```
const val string: String = "Hello"
```

同样是 final 变量，Java 这么写：

```
final String string = getString();
```

注意到，这个不是编译期常量，Kotlin 这么写：

```
val string: String = getString()
```

另外，Kotlin 有类型推导的特性，因此上述变量定义基本上都可以省略掉类型 String。

2. 如何定义函数

Java 中如何定义函数，也就是方法，需要定义到一个类当中：

```
public boolean testString(String name){  
    ...  
}
```

等价的 Kotlin 写法：

```
fun testString(name: String): Boolean {  
    ...  
}
```

注意到返回值的位置放到了参数之后。

3. 如何定义静态变量、方法

Java 的静态方法或者变量只需要加一个 static 即可：

```
public class Singleton{  
    private static Singleton instance = ...;  
  
    public static Singleton getInstance(){  
        ...  
        return instance;  
    }  
}
```

用 Kotlin 直译过来就是：

```

class KotlinSingleton{
    companion object{
        private val kotlinSingleton = KotlinSingleton()

        @JvmStatic
        fun getInstance() = kotlinSingleton
    }
}

```

注意 getInstance 的写法。 JvmStatic 这个注解会将 getInstance 这个方法编译成与 Java 的静态方法一样的签名，如果不加这个注解，Java 当中无法像调用 Java 静态方法那样调用这个方法。

另外，对于静态方法、变量的场景，在 Kotlin 当中建议使用包级函数。

4. 如何定义数组

Java 的数组非常简单，当然也有些抽象，毕竟是编译期生成的类：

```

String[] names = new String[]{"Kyo", "Ryu", "Iory"};
String[] emptyStrings = new String[10];

```

Kotlin 的数组其实更真实一些，看上去更让人容易理解：

```

val names: Array<String> = arrayOf("Kyo", "Ryu", "Iory")
val emptyStrings: Array<String?> = arrayOfNulls(10)

```

注意到，Array T 即数组元素的类型。另外，String? 表示可以为 null 的 String 类型。

数组的使用基本一致。需要注意的是，为了避免装箱和拆箱的开销，Kotlin 对基本类型包括 Int、Short、Byte、Long、Float、Double、Char 等基本类型提供了定制版数组类型，写法为 XArray，例如 Int 的定制版数组为 IntArray，如果我们要定义一个整型数组，写法如下：

```
val ints = intArrayOf(1, 3, 5)
```

5. 如何写变长参数

Java 的变长参数写法如下：

```
void hello(String... names){  
    ...  
}
```

Kotlin 的变长参数写法如下：

```
fun hello(vararg names: String){  
}
```

6. 如何写三元运算符

Java 可以写三元运算符：

```
int code = isSuccessfully? 200: 400;
```

很多人抱怨 Kotlin 为什么没有这个运算符。。。据说是由于 Kotlin 当中：使用的场景比 Java 复杂得多，因此如果加上这个三元运算符的话，会给语法解析器带来较多的麻烦，Scala 也是类似的情况。那么这种情况下，我们用 Kotlin 该怎么写呢？

```
int code = if(isSuccessfully) 200 else 400
```

注意到，if else 这样的语句也是表达式，这一点与 Java 不同。

7. 如何写 main 函数

Java 的写法只有一种：

```
class Main{
    public static void main(String... args){
        ...
    }
}
```

注意到参数可以是变长参数或者数组，这二者都可。

对应 Kotlin，main 函数的写法如下：

```
class KotlinMain{
    companion object{
        @JvmStatic
        fun main(args: Array<String>) {
            ...
        }
    }
}
```

Kotlin 可以有包级函数，因此我们并不需要声明一个类来包装 main 函数：

```
fun main(args: Array<String>){
    ...
}
```

8. 如何实例化类

Java 和 C++ 这样的语言，在构造对象的时候经常需要用到 new 这个关键字，比如：

```
Date date = new Date();
```

Kotlin 构造对象时，不需要 new 这个关键字，所以上述写法等价于：

```
val date = Date()
```

9. 如何写 Getter 和 Setter 方法

Java 的 Getter 和 Setter 是一种约定俗称，而不是语法特性，所以定义起来相对自由：

```
public class GetterAndSetter{  
    private int x = 0;  
  
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

Kotlin 是有属性的：

```
class KotlinGetterAndSetter{  
    var x: Int = 0  
        set(value) { field = value }  
        get() = field  
}
```

注意看到，我们为 x 显式定义了 getter 和 setter，field 是 x 背后真正的变量，所以 setter 当中其实就是为 field 赋值，而 getter 则是返回 field。如果你想要对 x 的访问做控制，那么你就可以通过自定义 getter 和 setter 来实现了：

```

class KotlinGetterAndSetter{
    var x: Int = 0
        set(value) {
            val date = Calendar.getInstance().apply {
                set(2017, 2, 18)
            }
            if(System.currentTimeMillis() < date.timeInMillis){
                println("Cannot be set before 2017.3.18")
            }else{
                field = value
            }
        }
    get(){
        println("Get field x: $field")
        return field
    }
}

```

10. 如何延迟初始化成员变量

Java 定义的类成员变量如果不初始化，那么基本类型被初始化为其默认值，比如 int 初始化为 0，boolean 初始化为 false，非基本类型的成员则会被初始化为 null。

```

public class Hello{
    private String name;
}

```

类似的代码在 Kotlin 当中直译为：

```

class Hello{
    private var name: String? = null
}

```

使用了可空类型，副作用就是后面每次你想要用 name 的时候，都需要判断其是否为 null。如果不使用可控类型，需要加 lateinit 关键字：

```
class Hello{  
    private lateinit var name: String  
}
```

lateinit 是用来告诉编译器，name 这个变量后续会妥善处置的。

对于 final 的成员变量，Java 要求它们必须在构造方法或者构造块当中对他们进行初始化：

```
public class Hello{  
    private final String name = "Peter";  
}
```

也就是说，如果我要想定义一个可以延迟到一定实际再使用并初始化的 final 变量，这在 Java 中是做不到的。

Kotlin 有办法，使用 lazy 这个 delegate 即可：

```
class Hello{  
    private val name by lazy{  
        NameProvider.getName()  
    }  
}
```

只有使用到 name 这个属性的时候，lazy 后面的 Lambda 才会执行，name 的值才会真正计算出来。

11. 如何获得 class 的实例

Java 当中：

```
public class Hello{  
    ...  
}  
  
...  
  
Class<?> clazz = Hello.class;  
  
Hello hello = new Hello();  
Class<?> clazz2 = hello.getClass();
```

前面我们展示了两种获得 class 的途径，一种直接用类名，一种通过类实例。刚刚接触 Kotlin 的时候，获取 Java Class 的方法却是容易让人困惑。

```
class Hello  
  
val clazz = Hello::class.java  
  
val hello = Hello()  
val clazz2 = hello.javaClass
```

同样效果的 Kotlin 代码看上去确实很奇怪，实际上 Hello::class 拿到的是 Kotlin 的 KClass，这个是 Kotlin 的类型，如果想要拿到 Java 的 Class 实例，那么就需要前面的办法了。

Plugin

- 用 Kotlin 写 Android 01 难道只有环境搭建这么简单？
- 用 Kotlin 写 Android 02 说说 Anko

[用 Kotlin 写 Android] 01 难道只有环境搭建这么简单？

从这周开始，每周一的文章推送将连载 Kotlin Android 开发的文章，大家有关心的题目也可以直接反馈给我，这样也可以帮助我提高后续文章的针对性。

亲爱的小伙伴，阅读本文之前，请确保你对 Kotlin 有一定的了解，并且你的 Android Studio 或者 IntelliJ Idea 已经安装了 Kotlin 的插件。如果没有，果断回去先看我的 [Kotlin 视频 第一集](#) :)

1 千里之行，始于 Hello World

话说我们入坑 Kotlin 之后，要怎样才能把它运用到 Android 开发当中呢？我们作为有经验的开发人员，大家都知道 Android 现在基本上都用 gradle 构建，gradle 构建过程中只要加入 Kotlin 代码编译的相关配置，那么 Kotlin 的代码运用到 Android 的问题就解决了。

这个问题有何难呢？Kotlin 团队早就帮我们把这个问题解决了，只要大家在 gradle 配置中加入：

```
apply plugin: 'kotlin-android'
```

就可以了，这与我们在普通 Java 虚拟机的程序的插件不太一样，其他的都差不多，比如我们需要在 buildScript 当中添加的 dependencies 与普通 Java 虚拟机程序毫无二致：

```
buildscript {
    ext.kotlin_version = '1.0.6'//版本号根据实际情况选择
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

当然，我们还要在应用的 dependencies 当中添加 Kotlin 标准库：

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

有了这些，你的 Kotlin 代码就可以跑在 Android 上面了！当然，你的代码写在 src/main/java 或是 src/main/kotlin 下都是可以的。这不重要了，我觉得把 Java 和 Kotlin 代码混着写就可以了，没必要分开，嗯，你最好不要感觉到他们是两个不同的语言，就酱紫。

```

package net.println.kotlinandroiddemo

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

class MainActivity : AppCompatActivity() {

    private lateinit var textView: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        textView = findViewById(R.id.hello) as TextView
        textView.text = "Hello World"
    }
}

```

我们定义一个 TextView 的成员，由于我们只能在 onCreate 当中初始化这个成员，所以我们只好用 lateinit 来修饰它。当然，如果你不怕麻烦，你也可以选择 TextView?，然后给这个成员初始化为 null。

接着我们就用最基本的写法 findViewById、类型强转拿到这个 textView 的引用，然后 setText。

运行自然是没问题的。

不过，不过！我如果就写这么点儿就想糊弄过去这一周的文章，番茄鸡蛋砸过来估计够我吃一年的西红柿炒鸡蛋了吧（我~就~知~道~，我这一年不用愁吃的了！）

2 Anko 已经超神

要说用 Kotlin 写 Android，Anko 谁人不知谁人不晓，简直到了超神的地步。好好，咱们不吹牛了，赶紧把它老人家请出来：

```
compile 'org.jetbrains.anko:anko-sdk15:0.9' // sdk19, sdk21, sdk  
23 are also available  
compile 'org.jetbrains.anko:anko-support-v4:0.9' // In case you  
need support-v4 bindings  
compile 'org.jetbrains.anko:anko-appcompat-v7:0.9' // For appcom  
pat-v7 bindings
```

稍微提一句 anko-sdk 的版本选择：

- org.jetbrains.anko:anko-sdk15 : 15 <= minSdkVersion < 19
- org.jetbrains.anko:anko-sdk19 : 19 <= minSdkVersion < 21
- org.jetbrains.anko:anko-sdk21 : 21 <= minSdkVersion < 23
- org.jetbrains.anko:anko-sdk23 : 23 <= minSdkVersion

当然除了这些之外，anko 还对 cardview、recyclerview 等等做了支持，大家可以按需添加，详细可以参考 [Github - Anko](#)

另外，也建议大家用变量的形式定义 anko 库的版本，比如：

```
ext.anko_version = "0.9"  
...  
  
compile "org.jetbrains.anko:anko-sdk15:$anko_version" // sdk19,  
sdk21, sdk23 are also available  
compile "org.jetbrains.anko:anko-support-v4:$anko_version" // In  
case you need support-v4 bindings  
compile "org.jetbrains.anko:anko-appcompat-v7:$anko_version" //  
For appcompat-v7 bindings
```

好，有了 Anko 我们能干什么呢？

```
textView = find(R.id.hello)
```

还记得 findViewById 吗？变成 find 了，而且强转也没有了，是不是很有趣？你一定有疑问，Anko 究竟干了啥，一下子省了这么多事儿，我们跳进去看看 find 的真面目：

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
```

首先它是个扩展方法，我们暂时只用到了 Activity 的扩展版本，实际上 View、Fragment 都有这个扩展方法；其次，它是个 inline 方法，并且还用到了 reified 泛型参数，我们本来应该这么写：

```
textView = find<TextView>(R.id.hello)
```

由于泛型参数的类型可以很容易的推导出来，所以我们再使用 find 的时候不需要显式的注明。

说到这里，其实还是有问题没有说清楚的，reified 究竟用来做什么？其实我们就算不写 inline 和 reified 泛型，这个方法照样是可以用的：

```
fun <T : View> Activity.myFind(id: Int): T = findViewById(id) as T
```

```
textView = myFind(R.id.hello)
```

不过呢，这地方用 inline 就省了一次函数调用，并且 reified 也可以消除 IDE 的类型检查提示，所以既然可以，为什么不呢？

当然，用 Anko 的好处不可能就这么点儿，我们今天先按住不说，谁好奇的话可以先自己去看看（我~就~知~道~，你们肯定忍不住！！）～

3 不要 findViewById

作为第一篇介绍 Kotlin 写 Android 的文章，绝对不能少的就是 kotlin-android-extensions 插件了。在 gradle 当中加配置：

```
apply plugin: 'kotlin-android-extensions'
```

之后，我们只需要在 Activity 的代码当中直接使用在布局中定义的 id 为 hello 的这个 textView，于是：

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

//这个包会自动导入
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //直接使用 hello，hello 实际上是这个view 在布局当中的id
        hello.text = "Hello World"
    }
}
```

只要布局添加一个 View，在 Activity、View、Fragment 中其实都可以直接用 id 来引用这个 view，超级爽~

所以，你们不准备问下这是为什么吗？为什么可以这样做呢？

其实要回答这个问题也不难，首先 Android Studio 要能够从 IDE 的层面索引到 hello 这个 View，需要 Kotlin 的 IDE 插件的支持（别问我啥是 IDE 插件，你们用 Kotlin 的第一天肯定都装过）；其次，在编译的时候，编译器能够找到 hello 这个变量，那么还需要 Kotlin 的 gradle 插件支持（我们刚刚好像 apply 了个什么 plugin 来着？）。知道了这两点，我们就要有的放矢了~

“啊！”那边的 Kotlin 源码一声惨叫。。。

前方高能。。。我们讨论的源码主要在 plugins 目录下的 android-extensions-compiler 和 android-extensions-idea 两个模块当中。

如果让大家自己实现一套机制来完成上面的功能，大家肯定会想，我首先得解析一下 XML 布局文件吧，并把里面的 View 存起来，这样方便后面的查找。我告诉大家，Kotlin 也是这么干的！

AndroidXmlVisitor.kt

```
override fun visitXmlTag(tag: XmlTag?) {
    ...
    val idAttribute = tag?.getAttribute(AndroidConst.ID_ATTRIBUTE)
    if (idAttribute != null) {
        val idAttributeValue = idAttribute.value
        if (idAttributeValue != null) {
            val xmlType = tag?.getAttribute(AndroidConst.CLASS_ATTRIBUTE_NO_NAMESPACE)?.value ?: localName
            val name = androidIdToName(idAttributeValue)
            if (name != null) elementCallback(name, xmlType, idAttribute)
        }
    }
    tag?.acceptChildren(this)
}
```

这是遍历 XML 标签的代码，典型的访问者模式对吧。如果拿到这个标签，它有 android:id 这个属性，那么小样儿，你别走，老实交代你的 id 是什么！举个例子，如果这个标签是这样的：

```
<Button
    android:id="@+id/login"
    ... />
```

那么，name 就是 login 了，既然 name 不为空，那么调用 elementCallback，其实就是把它记录了下来。

IDEAndroidLayoutXmlFileManager.kt

```
override fun doExtractResources(files: List<PsiFile>, module: ModuleDescriptor): List<AndroidResource> {
    val widgets = arrayListOf<AndroidResource>()

    //注意到这里的 Lambda 表达式就是前面的 elementCallback
    val visitor = AndroidXmlVisitor { id, widgetType, attribute
->
        widgets += parseAndroidResource(id, widgetType, attribute.valueElement)
    }

    files.forEach { it.accept(visitor) }

    //返回所有带 id 的 view
    return widgets
}
```

接着想既然我们找到了所有的布局带有 id 的 view，那么我们总得想办法让 Activity 它们找到这些 view 才行对吧，而我们发现其实在引用它们的时候总是要导入一个包，包名叫做：

```
kotlinx.android.synthetic.main.<布局文件名>.*
```

几个意思？Kotlin 编译器为我们创建了一个包？

AndroidPackageFragmentProviderExtension.kt

```
...
createPackageFragment(packageFqName, false)
createPackageFragment(packageFqName + ".view", true)
...
```

注意到，这里的 packageFqName 其实就是我们前面提到的

```
kotlinx.android.synthetic.main.<布局文件名>
```

不对呀，怎么创建了两个包呢？其实第二个多了个 .view ，我们在 Activity 当中导入的包是第一个，但如果是用父 view 引用子 view 时，用的是第二个：

```
...
import kotlinx.android.synthetic.main.activity_main.view.*

class OverlayManager(context: Context){
    init {
        val view = LayoutInflater.from(context).inflate(R.layout
            .activity_main, null)
        view.hello.text = "HelloWorld"
        ...
    }
    ...
}
```

好，我们现在知道了，IntelliJ 居然已经通过解析 XML 帮我们偷偷搞出了这么两个虚拟的包，这样我们在代码当中能够引用到这个包就很容易解释了。

这时候可能还会有人比较疑惑点击了 Activity 的 hello 之后如何跳转到 XML 的，这个大家阅读一下 `AndroidGotoDeclarationHandler` 的源码就会很容易的看到答案。

费了这么多篇幅，其实我们只是做好了表面文章。上面的一切其实都是障眼法，别管怎么说，这两个包都是虚拟的，编译的时候该怎么办？

其实编译就简单多了，碰到这样的引用，比如前面的 hello ，直接生成 `findViewById` 的字节码就可以了，我们把 `hello.text = "HelloWorld"` 的字节码贴出来给大家看：

```
L2
LINENUMBER 12 L2
ALOAD 0
GETSTATIC net/println/kotlinandroiddemo/R$id.hello : I
INVOKEVIRTUAL net/println/kotlinandroiddemo/MainActivity._$find
CachedViewById (I)Landroid/view/View;
CHECKCAST android/widget/TextView
LDC "Hello World"
CHECKCAST java/lang/CharSequence
INVOKEVIRTUAL android/widget/TextView.setText (Ljava/lang/CharSe
quence;)V
```

这个是怎么做到的？请大家阅读 `AndroidExpressionCodegenExtension.kt`，

```
...
//GETSTATIC net/println/kotlinandroiddemo/R$id.hello : I
v.getstatic(packageName.replace(".", "/") + "/R\\$id", resourceId
.name, "I")

//INVOKEVIRTUAL net/println/kotlinandroiddemo/MainActivity._$fi
ndCachedViewById (I)Landroid/view/View;
v.invokevirtual(declarationDescriptorType.internalName,
    CACHED_FIND_VIEW_BY_ID_METHOD_NAME,
    "(I)Landroid/view/View;", false)
...
```

好，到这里，想必大家才能对 Android 的 HelloWorld 代码有一个彻底的理解。

4 小结

虽然是 HelloWorld，但要想搞清楚其中的所有秘密，并没有那么简单，很多时候，阅读 Kotlin 源码几乎成了唯一的途径。

谢谢大家的关注和支持~如果有任何问题可以联系我~

[用 Kotlin 写 Android] 02 说说 Anko

上周的文章其实我们提到了 Anko 的，不过我们只是给大家展示了一下 `find` 方法。除了这个之外，还有哪些好玩的东西呢？

1. 简化页面操作

我们写 Android 最先做的是什么？当然是设置个 `OnClickListener`，这样自然我的按钮听我的，我的地盘我做主了。

```
hello.setOnClickListener {
    startActivity<AnotherActivity>("from" to "MainActivity")
}
```

哎哟，不错哦。其中 `hello` 是一个 `TextView`，我们通过 `onClick` 为其设置了一个 `OnClickListener`，这样看上去真是简洁不少。

```
fun android.view.View.onClick(l: (v: android.view.View?) -> Unit) {
    setOnClickListener(l)
}
```

也没什么难理解的，`onClick` 是一个扩展方法，传入的 Lambda 表达式通过 SAM 转换成了 `OnClickListener`，一切都是这么的自然。如果你对传入的 `view` 感兴趣，你当然可以直接用 `it` 召唤它：

```
hello.setOnClickListener {
    Log.d(TAG, it.toString())
    startActivity<AnotherActivity>("from" to "MainActivity")
}
```

简单吧。

等等！那个 `startActivity` 是怎么回事？没有 `Intent` 么？

哈哈，这个嘛，且看源码：

```
inline fun <reified T: Activity> Context.startActivity(vararg pa
    rams: Pair<String, Any>) {
    AnkoInternals.internalStartActivity(this, T::class.java, par
    ams)
}

...

fun internalStartActivity(
    ctx: Context,
    activity: Class<out Activity>,
    params: Array<out Pair<String, Any>>
) {
    ctx.startActivity(createIntent(ctx, activity, params))
}
```

其实也没什么，就是对我们之前模板式的跳转写法做了简化而已，至于用到的 reified 和 Pair 也不算什么新鲜的东西，Pair 当中的 K-V 实际上就是我们通常放入 Intent 的 extra，所以我们自然可以在 AnotherActivity 当中取到这个值：

```
class AnotherActivity: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        relativeLayout {
            textView {
                text = if(intent == null)
                    "from nowhere"
                else
                    intent.extras["from"]?.toString()
            }
        }
    }
}
```

取到我们传入的值，在 AnotherActivity 当中显示出来。有木有觉得要比我们用 Java 老大哥写出来的代码简洁易懂呢？

2. 聊聊 DSL 布局

再等等！那个 `relativeLayout{...}` 是几个意思？？

嗯，这个要多说几句了，Anko 这个框架虽然打着简化开发的旗号，不过野心终归还是不小的。它自己搞出一套用 Kotlin 写布局的 DSL，换句话说，有了 Anko 我们布局甚至可以不需要用 XML 了，也不需要像用 Java 硬编码 View 那么繁琐，只需要通过几句 DSL 就可以搞定。我们来多看几个例子：

2.1 水平布局

下面是三个按钮水平等分布局的写法，我们用到的实际上就是线性布局，比较简单，button 的参数是按钮的文字（有较多重载的版本，大家可以酌情选择），lparams 的参数有三个，前两个分别是宽、高，最后一个是一个 Lambda 表达式，我们可以在这个 Lambda 表达式当中详细定义我们需要的布局，比如设置 margin 等等。

```
linearLayout {
    button("1").lparams(wrapContent, wrapContent){
        weight = 1f
    }
    button("2").lparams(wrapContent, wrapContent){
        weight = 1f
    }
    button("3").lparams(wrapContent, wrapContent){
        weight = 1f
    }
}
```

效果图如下：



2.2 纵向布局

还是线性布局，不过换了个方向，你当然可以这么写：

```
linearLayout {  
    orientation = LinearLayout.VERTICAL  
    button("1").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
    button("2").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
    button("3").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
}
```

不过，Anko 更倾向于让我们用这个：

```
verticalLayout {  
    button("1").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
    button("2").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
    button("3").lparams(wrapContent, wrapContent){  
        weight = 1f  
    }  
}
```

我给大家看一下源码，大家就分分钟明白了：

```

val VERTICAL_LAYOUT_FACTORY = { ctx: Context ->
    val view = _LinearLayout(ctx)
    view.orientation = LinearLayout.VERTICAL
    view
}

```

其实我们创建的 verticalLayout 最终是从这个方法当中获取的，没啥新鲜的，就是设置了一下 orientation 罢了。效果图我就不贴了，大家很容易猜得到。

2.3 相对布局

```

relativeLayout {
    relativeLayout {
        textView("周杰伦") {
            id = R.id.extra
            useSecondary()
            .lparams(wrapContent, wrapContent) {
                alignParentRight()
                centerVertically()
                rightMargin = dip(10)
            }

            imageView {
                id = R.id.avatar
                imageResource = R.drawable.jaychow
                scaleType = ImageView.ScaleType.FIT_XY
                .lparams(dip(40), dip(40)){
                    centerVertically()
                    leftMargin = dip(10)
                }

                textView("千里之外") {
                    id = R.id.title
                    usePrimary()
                    .lparams(matchParent, wrapContent) {
                        leftOf(R.id.extra)
                        rightOf(R.id.avatar)
                        margin = dip(5)
                    }
                }
            }
        }
    }
}

```

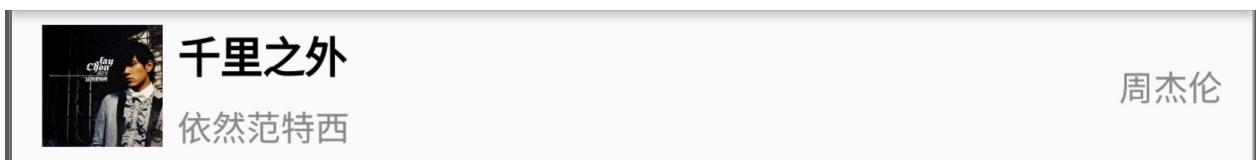
```
}

    textView("依然范特西") {
        id = R.id.subtitle
        useSecondary()
    }.lparams(matchParent, wrapContent) {
        leftOf(R.id.extra)
        rightOf(R.id.avatar)
        below(R.id.title)
        leftMargin = dip(5)
    }
}.lparams(matchParent, dip(50))
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="title" type="id"/>
    <item name="subtitle" type="id"/>
    <item name="extra" type="id"/>
    <item name="avatar" type="id"/>
</resources>
```

这个布局我们看到其实就是一张图片，三个 TextView，难度也不大，不过这种写法可能要适应一下。

效果如下：



注意到我在 TextView 当中用了两个方法：usePrimary() 和 useSecondary()，这其实是我定义的样式：

```
fun TextView.usePrimary(){
    textSize = 15f //注意这里就是 sp 的值
    textColor = Color.BLACK
    typeface = Typeface.DEFAULT_BOLD
}

fun TextView.useSecondary(){
    textSize = 12f //注意这里就是 sp 的值
    textColor = Color.GRAY
}
```

这个算是比较复杂的一个布局了，只要 XML 可以搞定的用 Anko DSL 的方式一样可以搞定，而且写出来的东西都可以直接对应到源码，这一点是非常棒的。我们在使用 XML 布局的时候如果想要知道某一个属性对应 View 的什么成员，还得去找这个 View 解析 XML 的代码，显然这一点 DSL 要方便一些。

2.4 独立的 UI

前面我们说到的都是在 Activity 的 onCreate 方法中使用 DSL 的场景。很多时候我们其实还是希望布局和 Activity 分开的，那么我们就可以用官方推荐的这种方式来给 Activity 设置布局：

```

class MyActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?, persistentState: PersistableBundle?) {
        super.onCreate(savedInstanceState, persistentState)
        MyActivityUI().setContentView(this)
    }
}

class MyActivityUI : AnkoComponent<MyActivity> {
    override fun createView(ui: AnkoContext<MyActivity>) = with(ui) {
        verticalLayout {
            val name = editText()
            button("Say Hello") {
                onClick { ctx.toast("Hello, ${name.text}!") }
            }
        }
    }
}

```

2.5 在任意位置用 DSL 定义 View

前面提到的各种 `relativeLayout {}` 也好，`verticalLayout {}` 也好，都只能在 Activity、ViewManager（ViewGroup 的接口）、Context 这三个类的作用域范围之内使用，换句话说前面的几个布局的方法都是这几个类的扩展方法。

下面这个写法是没有问题的：

```

fun createView(context: Context): View{
    return context.relativeLayout{
        ...
    }
}

```

相应的，我们可以用任意一个 ViewGroup 的子类来调用类似的方法，这与调用 `viewRoot.addView(FrameLayout(viewRoot.context))` 是一样的：

```
fun addViewToParent(viewRoot: ViewGroup){
    viewRoot.frameLayout {
        ...
    }
}
```

如果是在 Fragment 当中，Anko 还非常贴心的定义了一个叫 UI 的方法，这个方法同时也存在于 Context 当中，用法也比较简单：

```
class MainFragment: Fragment(){
    override fun onCreateView(
        inflater: LayoutInflater?,
        container: ViewGroup?,
        savedInstanceState: Bundle?): View {
        return UI {
            tableLayout {
                ...
            }
        }.view
    }
}
```

2.6 扩展 Anko，支持自定义 View

我们在开发中经常继承一个 View 实现一些自己想要的功能，比如我们继承 _RelativeLayout:

```
class CustomLayout(context: Context)
    : _RelativeLayout(context) {
    ...
}
```

注意，如果我们直接继承 RelativeLayout，那么还需要自己定义 layoutParams 方法，这个我就不细说了，大家有需求可以自己详细研究~

为了让 Anko DSL 支持下面的写法：

```
customLayout{
    button("ClickMe"){ ... }
}
```

我们需要定义下面三组扩展方法：

```
inline fun ViewManager.customLayout(theme: Int = 0)
    = customLayout(theme) {}

inline fun ViewManager.customLayout(
    theme: Int = 0,
    init: CustomLayout.() -> Unit)
    = ankoView(::CustomLayout, theme, init)

inline fun Activity.customLayout(theme: Int = 0)
    = customLayout(theme) {}

inline fun Activity.customLayout(
    theme: Int = 0,
    init: CustomLayout.() -> Unit)
    = ankoView(::CustomLayout, theme, init)

inline fun Context.customLayout(theme: Int = 0)
    = customLayout(theme) {}

inline fun Context.customLayout(
    theme: Int = 0,
    init: CustomLayout.() -> Unit)
    = ankoView(::CustomLayout, theme, init)
```

其中，第一组 ViewManager 的是为了在 ViewGroup 当中使用；第二组是为了在 Activity 当中使用，第三组就是为了在所有 Context 当中使用。

扩展也是非常简单的，用起来也丝毫感觉不到这些 View 是自定义的，比起 XML 标签长长的一串确实也要美观得多。

```
<net.println.kotlinandroiddemo.CustomLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    ...
</net.println.kotlinandroiddemo.CustomLayout>
```

2.7 Anko DSL 使用小结

Anko DSL 的方式布局看上去还是比较清爽直观的，而且因为这是 Kotlin 代码，自然所有的 view 都是强类型约束的，不需要我们 findViewById 再强转，除此之外由于是代码，可以直接运行，也就省去了运行时解析 XML 的开销，这一点可以说也是相比于 Android 官方的 XML 布局而言 Anko 主打的性能优势。

它的各方面优势我们在前面已经给大家一一点到，可是它存在哪些问题呢？

- 首先，Anko DSL 布局不能预览。可以说这一点足以让我们放弃它了，不能预览的话很多时候我们只能通过运行结果来判断布局是否准确，这对开发效率的影响是巨大的。当然，这么说可能 Anko 不服，毕竟人家也是发布了一个叫 Anko Preview Plugin 的 IDE 插件的，有了这个插件理论上我们就可以预览 Anko DSL 的布局结果了对吧？可是结果呢，每次做了修改都需要 make 一下才可以看到结果，显然预览速度来看不如 XML 快。而就算这个问题我们可以忍，慢就慢点儿，别慢太多就行了吧，结果呢，人家这个插件存在各种各样的问题，比如对最新版的 Android Studio 2.2 和 IntelliJ 2016.3 不支持（当然其实本质上是对新版本的 Preview 功能不兼容），大家可以参考这个 issue：<https://github.com/Kotlin/anko/issues/202>。也就是说，这个插件现在是不能用的，所以跟没有也没啥区别。
- 其次，对于 id 的定义会比较蛋疼。我们知道我们在布局的时候可以通过 `@+id/xxx` 的方式生成一个 id，并交给 Android 资源管理器统一管理，用 Anko DSL 的话我们就得专门定义一个变量或者在 value 目录下面增加 id 的定义（就像 2.3 的例子那样）去让 view 引用。不用 id 行不行呢？你去问问 RelativeLayout 答应不答应吧。

```
val FROM_TEXT = 0
val CLICK_ME = 1
relativeLayout {
    textView {
        text = ...
        id = FROM_TEXT
    }

    button("clickMe"){
        id = CLICK_ME
    }.lparams {
        below(FROM_TEXT)
    }
}
```

- 再次，我们通常会需要引用一些 view，通过 XML 布局 + kotlin-android-extensions 的方式，我们可以直接引用到这些有 id 的 view，非常方便，不过，如果我们用 Anko DSL 布局的话，我们就享受不到这项福利了（如果你不明白为什么，可以去看下我的[前一篇文章: 用Kotlin写Android 01 难道只有环境搭建这么简单？](#)）。

```

val FROM_TEXT = 0
val CLICK_ME = 1
var fromText: TextView? = null
relativeLayout {
    fromText = textView {
        text = ...
        id = FROM_TEXT
    }

    button("clickMe"){
        id = CLICK_ME
    }.lparams {
        below(FROM_TEXT)
    }
}

...
if(shouldHideText) fromText?.visibility = View.GONE
else fromText?.visibility = View.VISIBLE

```

- 还有就是，如果我们的布局有多个版本，而且需要动态替换外部资源以达到换肤的效果，那么 XML 显然比 Kotlin 代码要来得容易：前者可以编译成一个只有资源的 apk 供应用加载，后者的话就得搞一下动态类加载了。

总之，Anko DSL 布局这个特性我个人觉得还没有达到可以取代 XML 布局的地步，如果大家习惯用 Java 硬编码 View 结构的话，Anko DSL 是个非常不错的选择；相反，如果大家一直用 XML 的话，那请接着用 XML 吧。当然，如果大家有好的使用方式，无论如何要来我这儿跟我嘚瑟一下哈~

3. 简化异步操作

假如你要在点击按钮之后把一个文件（本地或者服务端，也可能比较大，总之读取耗时）当中的文字显示出来，你用 Java 会怎么写呢？

```
button.setOnClickListener(new OnClickListener(){
    @Override public void onClick(View view){
        getExecutor().execute(new Runnable(){
            @Override public void run(){
                ...
                MainActivity.this.runOnUiThread(new Runnable(){
                    ...
                });
            }
        });
    }
});
```

哎呀我去，真是蜜汁缩进啊，我都写晕了。可是有了 Anko 配合，这段代码简直不能更清爽：

```
button.onClick {
    doAsync {
        val text = File("You raise me up.lrc").readText()
        uiThread {
            hello.text = text
        }
    }
}
```

doAsync 当中的代码运行在 Anko 配置的线程池当中，执行完之后还可以转入 uiThread 块来操作 UI，简单明了，还不容易出错。你当然也可以处理异常和自定义线程池：

```
doAsync(  
    exceptionHandler = {  
        Log.e(TAG, "error happened when read file.", it)  
    },  
    task = {  
        val text = File("You raise me up.lrc").readText()  
        uiThread {  
            hello.text = text  
        }  
    },  
    executorService = Executors.newSingleThreadExecutor()  
)
```

其实大家肯定想到了这两个方法的实现逻辑：

```

fun <T> T.doAsync(
    exceptionHandler: ((Throwable) -> Unit)? = null,
    executorService: ExecutorService,
    task: AnkoAsyncContext<T>.() -> Unit
): Future<Unit> {
    val context = AnkoAsyncContext(WeakReference(this))
    return executorService.submit<Unit> {
        try {
            context.task()
        } catch (thr: Throwable) {
            exceptionHandler?.invoke(thr)
        }
    }
}

...

```



```

fun <T> AnkoAsyncContext<T>.uiThread(f: (T) -> Unit): Boolean {
    val ref = weakRef.get() ?: return false
    if (ContextHelper.mainThread == Thread.currentThread()) {
        f(ref)
    } else {
        ContextHelper.handler.post { f(ref) }
    }
    return true
}

```

4. 简化日志打印

不知道大家有没有觉得 `Log.d(TAG, ...)` 这样的代码写起来麻烦，绝大多数情况下，我们打日志都需要多写个 `Log.` 除非静态导入方法，以及 `TAG` 的值通常都是对应的类名，有时候我只是为了临时打印一行日志，还得去定义一个静态常量 `TAG`，简直了，还有就是如果我只是想要打印一下某一个对象，还得显式得调用 `toString` 方法，一点儿都不智能。

```

public class MainActivity extends Activity{
    public static final String TAG = "MainActivity";

    ...
    View view = ...
    Log.d(TAG, view.toString());
    ...

}

```

有了 Anko 就要简单的多了，只要实现 AnkoLogger 这个接口，我们就可以愉快的打印日志了：

```

class SomeActivity : Activity(), AnkoLogger {
    private fun someMethod() {
        info("London is the capital of Great Britain")
        debug(5) // .toString() method will be executed
        warn(null) // "null" will be printed
    }
}

```

日志的 TAG 默认就是类名称，如果你需要自定义，那也没关系，直接覆写这个变量就可以了：

```
override val loggerTag: String = "SomeActivityTag"
```

5. 小结

Anko 这个框架其实没有什么复杂的地方，它更多的是在想办法简化我们的“八股文”代码，让我们的生活更轻松一些而已。DSL 布局是一个很不错的尝试，不过现在看来还是不太完美的，XML 本身也没有太大的问题，想必后续大家完全转向 DSL 的动力也不会很大。

除了前面提到的特性，Anko 还可以简化对话框、toast、sqlite 等操作，相比之下，toast 的用法还是比较常用的，也比较简单，我就不细说了；至于 sqlite，通常我们也不建议去直接操作它，用一些 ORM 框架可能会让你的代码更友好。

IDE

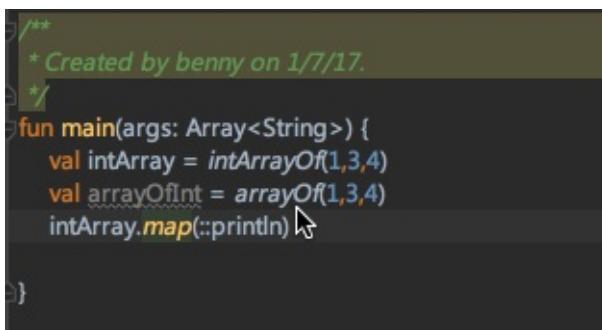
高效地使用你的 IntelliJ

这个世界上，有些人是奥义·真·懒，他们懒得折腾一切，甚至懒得活，所以他们不会看到这篇文章的。还有些人是奥义·嘴上说的·懒，这些人有个特点，懒到什么事儿都要弄个工具，于是人类社会就这么被一点儿推动着前进了。看到这篇文章的大家应该都是后面的这类人，so，给自己鼓个掌吧！

话说，人懒就要用 IDE，IntelliJ 已经超神了，用的人越来越多，不过，它有这么多技能你们都用到过吗？

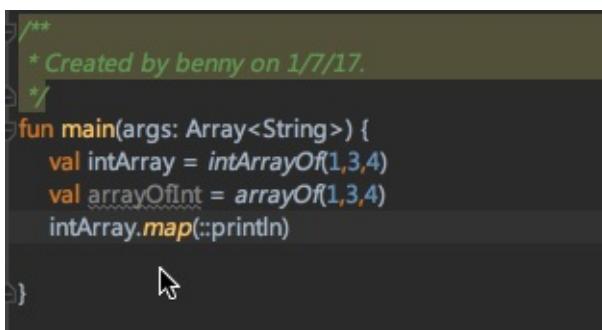
回车 or Tab ?

很多时候我们在召唤出自动补全之后，光标选中了我们想要的那个选项，于是我们按回车。。。。



```
/*
 * Created by benny on 1/7/17.
 */
fun main(args: Array<String>) {
    val intArray = intArrayOf(1,3,4)
    val arrayToInt = arrayOf(1,3,4)
    intArray.map(::println) →
}
```

额。。。这。。。难道就不能直接替换掉那个 map 吗？！当然可以：



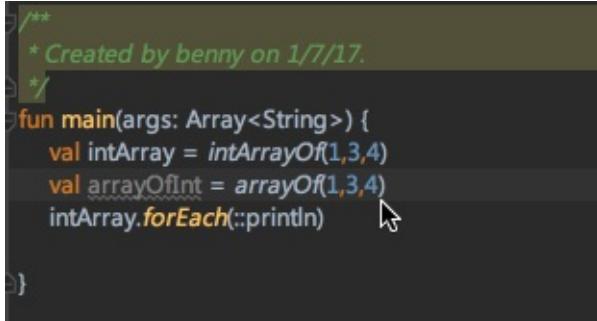
```
/*
 * Created by benny on 1/7/17.
 */
fun main(args: Array<String>) {
    val intArray = intArrayOf(1,3,4)
    val arrayToInt = arrayOf(1,3,4)
    intArray.map::println →
}
```

这操作有什么区别呢？前者按的是回车，后者按的是 Tab。

印象中，微软的东西都比较倾向于按 Tab 补全，比如 VS，再比如 Excel (Whaaaat?)，说这个也没别的意思，就是想让你记住，有时候 Tab 比回车好用。

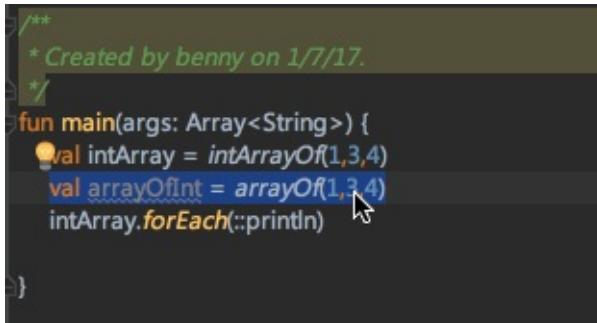
Swap 一下位置？

如果你飞快地写下了两句代码，然后发现他们的顺序是错的，你会怎么办？



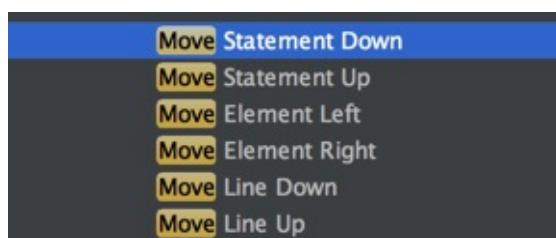
```
/*
 * Created by benny on 1/7/17.
 */
fun main(args: Array<String>) {
    val intArray = intArrayOf(1,3,4)
    val arrayOfInt = arrayOf(1,3,4)
    intArray.forEach(::println)
}
```

选中一整行，剪切，然后粘到上一句的前面？你 Out 了！看我的！

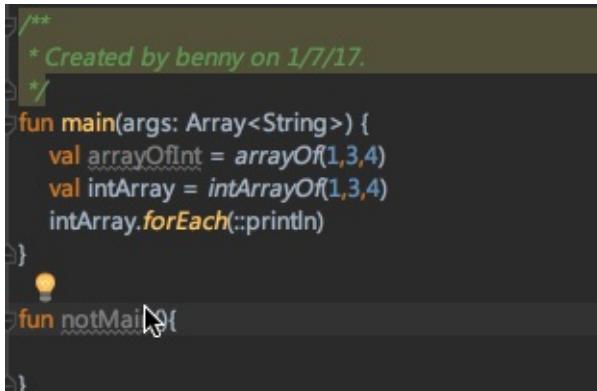


```
/*
 * Created by benny on 1/7/17.
 */
fun main(args: Array<String>) {
    val intArray = intArrayOf(1,3,4)
    val arrayOfInt = arrayOf(1,3,4)
    intArray.forEach(::println)
}
```

怎么做到的？很简单，打开你的设置，找到 keymap，再找到下面的几个 Action：



Statement 和 line 有时候会比较相似，比如我们刚才的情形；不过 Statement 还可以表示一整个函数或者类，于是你完全可以把光标放到一个函数的函数名上，然后 move up and down：

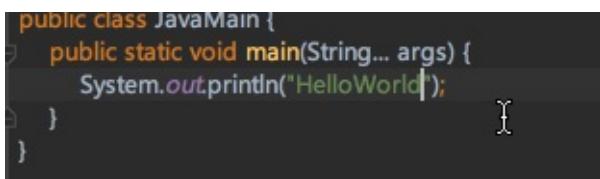


```
/*
 * Created by benny on 1/7/17.
 */
fun main(args: Array<String>) {
    val arrayOfInt = arrayOf(1,3,4)
    val intArray = intArrayOf(1,3,4)
    intArray.forEach(::println)
}
fun notMain()
```

如果你觉得默认的快捷键不好使，自己定义一个，你开心就好。

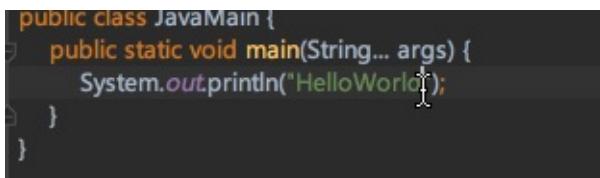
想要不移动光标就换行？

经常遇到这种需求，我的光标在某一行的中间，这时候想要换行，于是。。



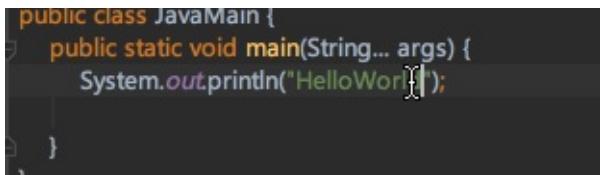
```
public class JavaMain {  
    public static void main(String... args) {  
        System.out.println("HelloWorld");  
    }  
}
```

这明显有点儿尴尬。。当然你也可以直接移到行末，再回车，但总是要操作两次。
其实这个也简单了，按住 shift 再回车，就可以直接换到下一行：



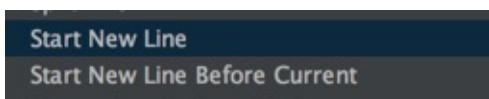
```
public class JavaMain {  
    public static void main(String... args) {  
        System.out.println("HelloWorld");  
    }  
}
```

而按住 cmd+option 再按回车就直接换到上一行：



```
public class JavaMain {  
    public static void main(String... args) {  
        System.out.println("HelloWorld");  
    }  
}
```

windows的快捷键我不知道，不过我还是要告诉大家它的名称，你自己定义就好：



快速迭代一个集合或者数组？

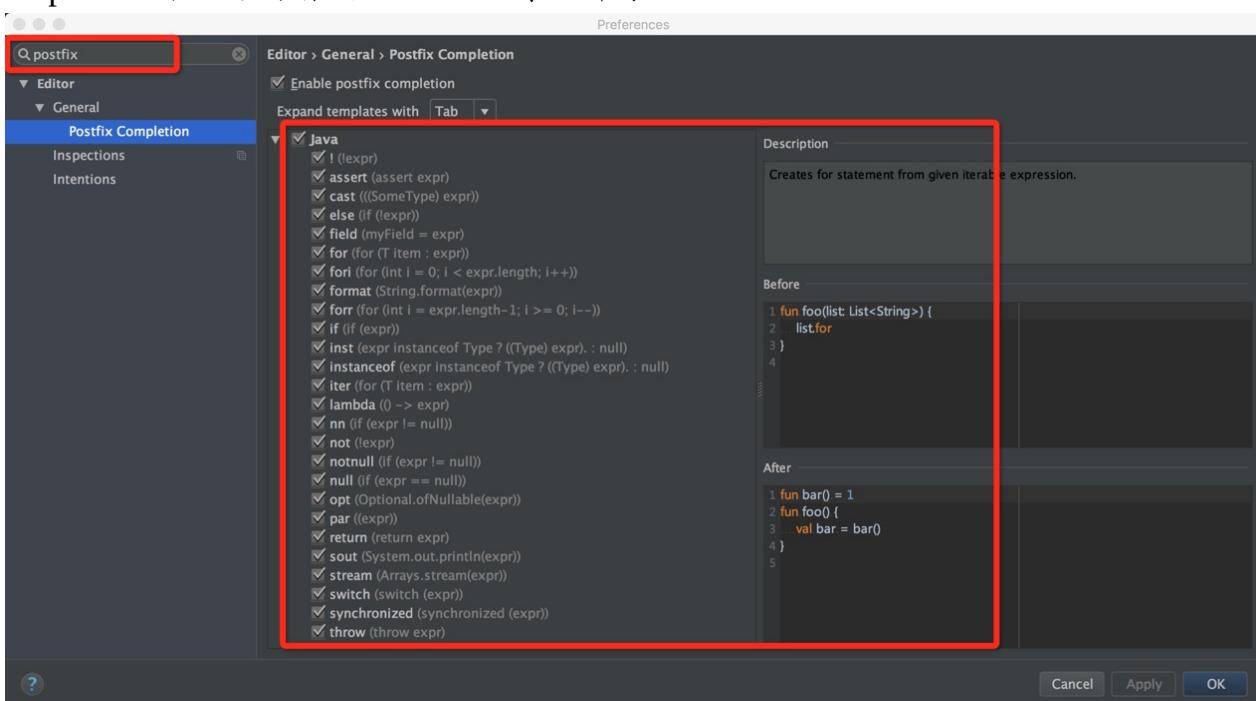
如果你在 Kotlin 里面，可能并不会有这样的困惑，毕竟我们可以写完一个集合，然后 .map 就行了。Java 里面可不能这样，于是。。。。

```
public class JavaMain {
    public static void main(String... args) {
        ArrayList<String> strings = new ArrayList<String>
        for (int i = 0; i < 100; i++) {
            strings.add(String.valueOf(i));
        }
    }
}
```

怎么办，怎么忍？让我们喊出我们的口号：不能忍！

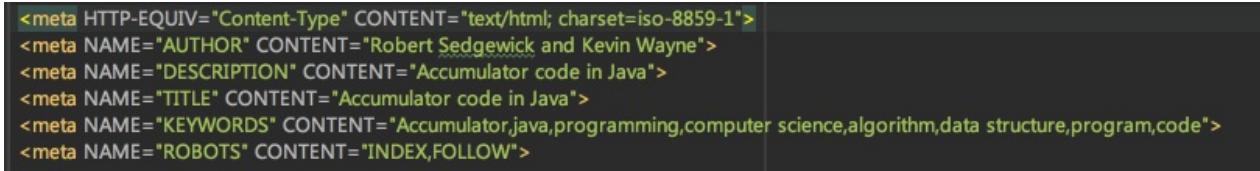
```
public class JavaMain {
    public static void main(String... args) {
        ArrayList<String> strings = new ArrayList<String>
        for (int i = 0; i < 100; i++) {
            strings.add(String.valueOf(i));
        }
    }
}
```

上面只要我们在打出一个集合之后，再打出 for，就能自动生成这段迭代的代码，这个叫做 Postfix。IntelliJ 支持了挺多的 Postfix，我就不一一列出了，在设置里面输入 postfix 那么你将看到 IntelliJ 支持的所有 Postfix：



多行编辑，哎哟小心某些人的钛合金眼

多行编辑没什么神秘的，大家一看就明白了。第一种触发方式比较简单，按住 option或者 alt，用鼠标往下拖就可以触发，例如：



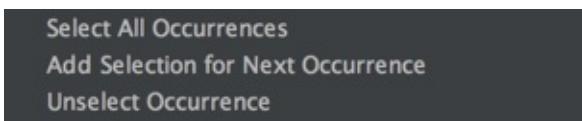
```
<meta HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<meta NAME="AUTHOR" CONTENT="Robert Sedgewick and Kevin Wayne">
<meta NAME="DESCRIPTION" CONTENT="Accumulator code in Java">
<meta NAME="TITLE" CONTENT="Accumulator code in Java">
<meta NAME="KEYWORDS" CONTENT="Accumulator,java,programming,computer science,algorithm,data structure,program,code">
<meta NAME="ROBOTS" CONTENT="INDEX,FOLLOW">
```

第二种要高端一些了，比如我要修改个一个词，又不可以通过 refactor 来统一修改，怎么办，难道有一百个还要让我改一百次吗？当然不需要，先选中第一个，然后如果是在 Mac 上，按 ctrl+G) 选择下一个，或者直接 cmd+ctrl+G 选择全部，就如下图所示，选中之后，多行编辑触发，你就可以一下改掉所有的：



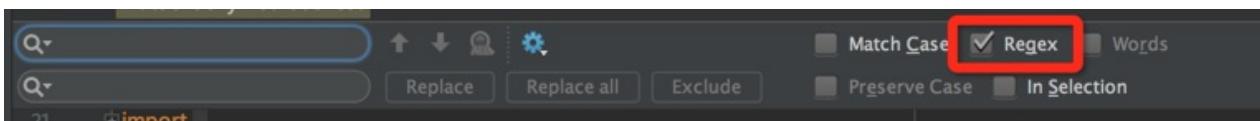
```
<meta HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<meta NAME="AUTHOR" CONTENT="Robert Sedgewick and Kevin Wayne">
<meta NAME="DESCRIPTION" CONTENT="Accumulator code in Java">
<meta NAME="TITLE" CONTENT="Accumulator code in Java">
<meta NAME="KEYWORDS" CONTENT="Accumulator,java,programming,computer science,algorithm,data structure,program,code">
<meta NAME="ROBOTS" CONTENT="INDEX,FOLLOW">
```

这里涉及到添加选择下一个和选择全部的操作，大家可以自己随意定义快捷键：

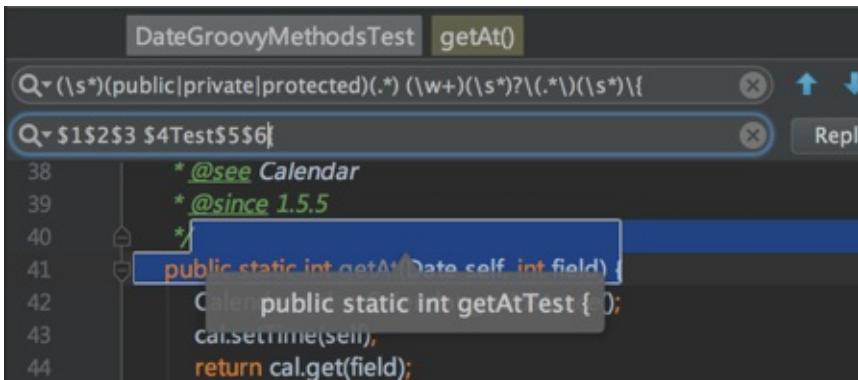


查找替换也有说法？

常规的查找替换肯定没啥可说的。不过不知道你注意到没，查找替换还有个 regex 的选项：



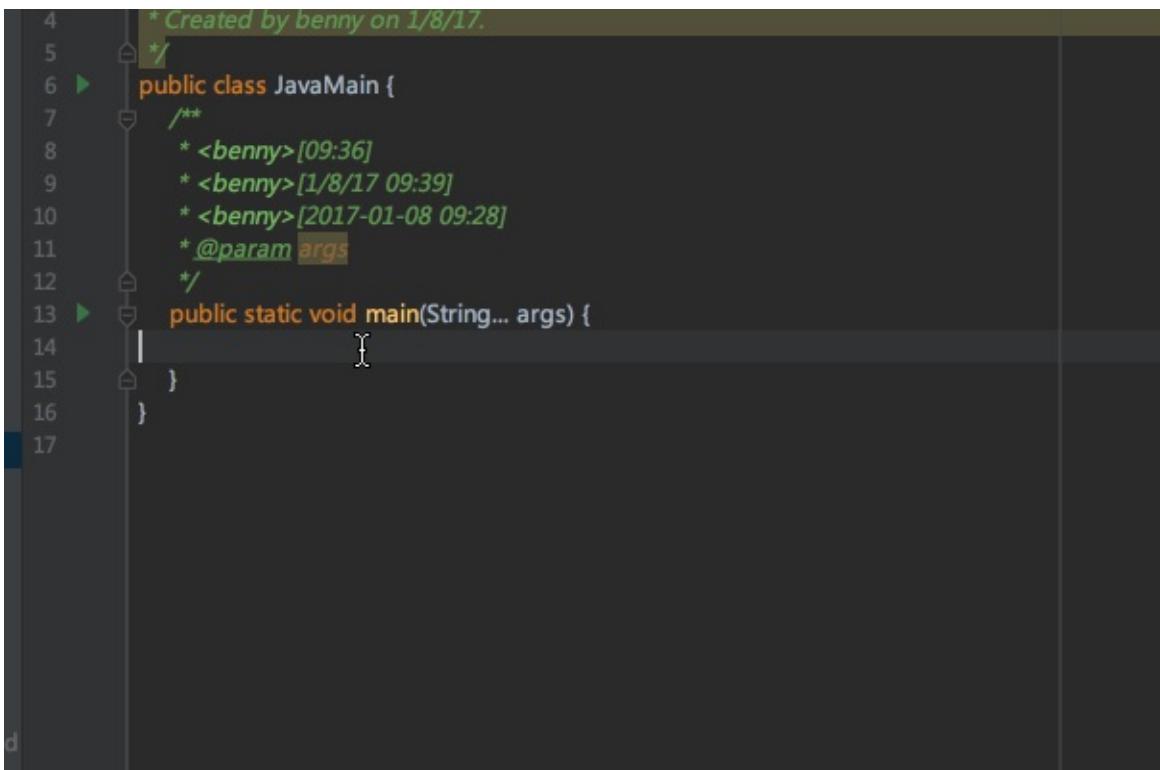
这个咋用呢？很简单，搜索词按照标准的正则表达式写就行了。比如我要定位到这个文件里面的所有函数名称，并在他们后面加一个 Test 方法，这种通常来说多行编辑是不太好做了，所以：



替换结果中，\$n 表示第 n 个匹配到的元组。这个用起来对正则表达式的功底要求比较高了，当然我觉得他更有用的地方在于帮助你熟悉正则比倒是的用法。

最强快捷键，没有之一

我之前看谷歌大会的视频，有一哥們在操作几乎任何操作的时候都用到了一个类似于命令行的快捷键，看上去非常酷：

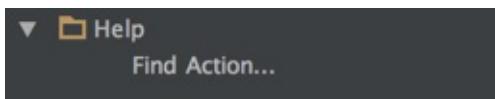


在这里，几乎可以输入任意操作，比如我们想要查看 Kotlin 的字节码，或者想要看下历史记录，甚至 make/run/debug，都毫无压力。

实际上，如果你对 IntelliJ 的插件开发有那么一丁点儿了解（比如我，虽然我自己没有写过==、），你就会知道 IntelliJ 里面的各种操作都是 Action，只要是个 Action，就都可以配置快捷键，而且索引方式也是按照名称，所以前面的输入框就

可以毫无压力的索引到这些 Action 了。

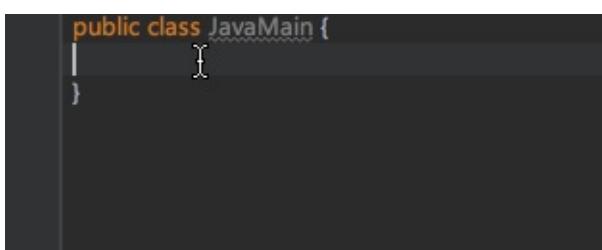
那么这个搜索框本身是个啥呢？它自己也是一个 Action，叫做：



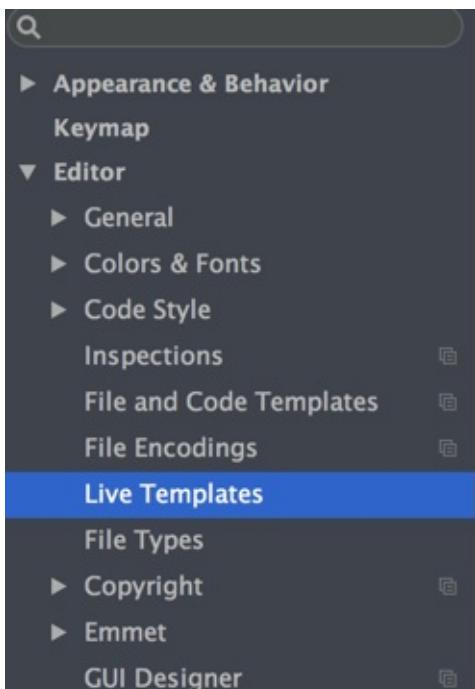
给它配个快捷键，让你的 IntelliJ 起飞吧

为什么输入 **main** 就能打出完整的 **main** 方法？

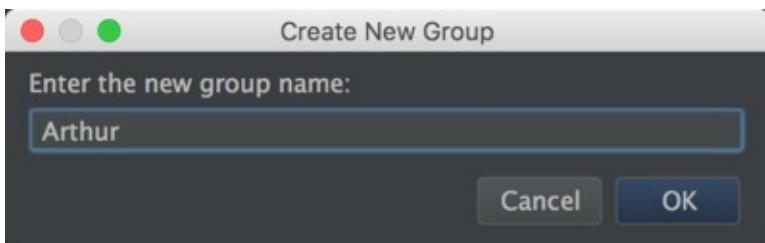
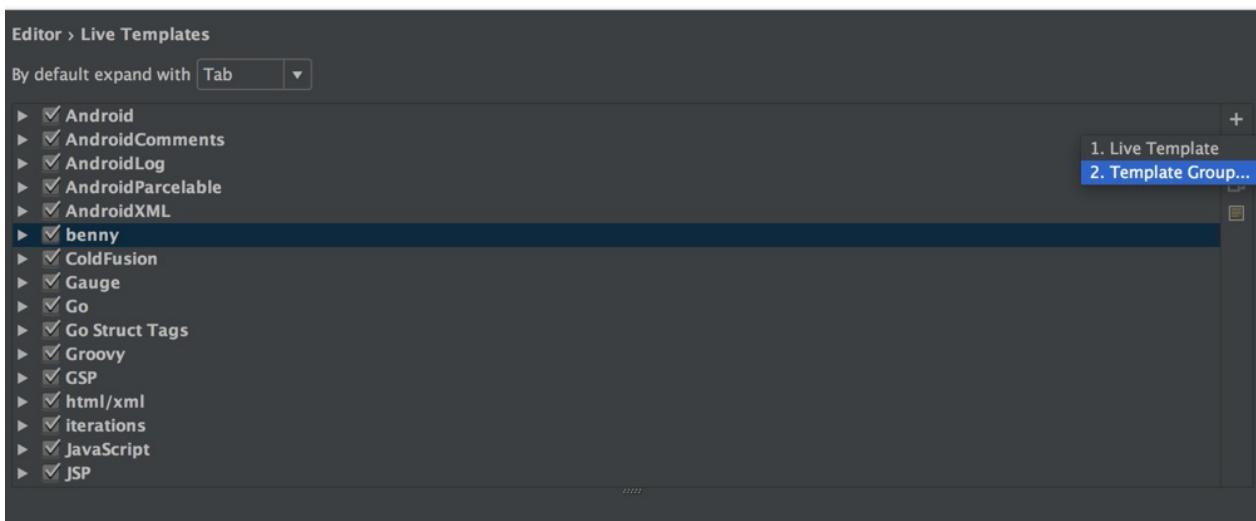
我在录制视频的时候经常会用一个比较快捷的输入方法：



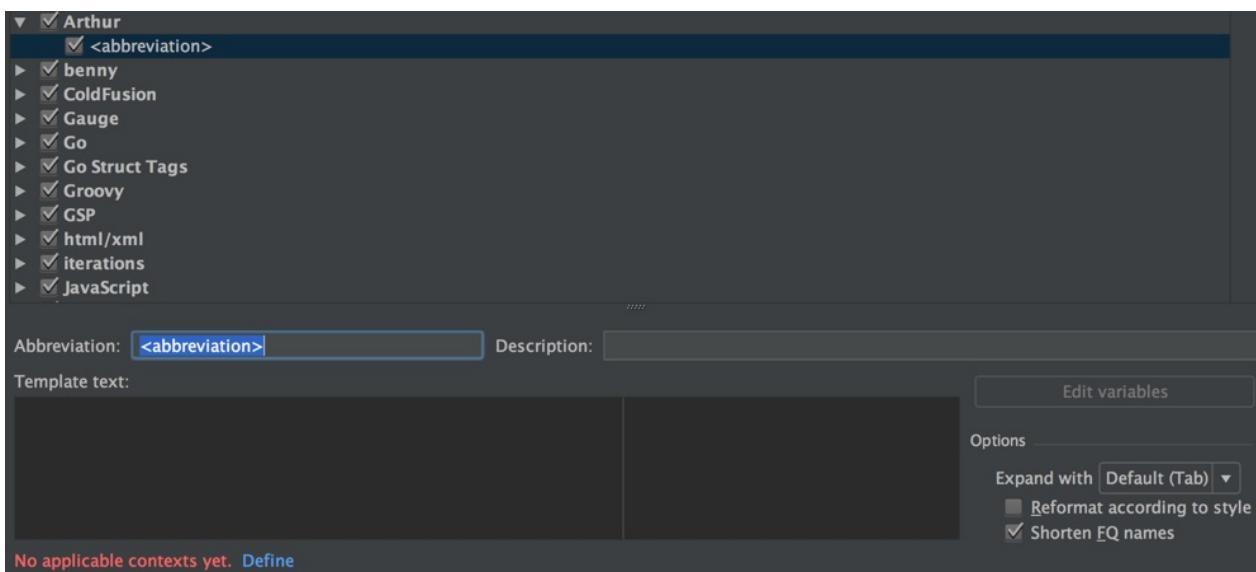
输入main之后就可以打出完整的main方法，这是怎么做到的？打开你的设置，找到Live Templates，



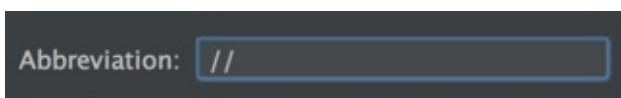
新建一个 group 叫亚瑟：



在这个 group 当中我们新建一个 Live Templates :



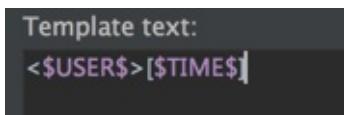
注意看，abbreviation 是缩写的意思，也就是说你将通过输入这段文字来触发模板，比如我们刚才的 main，我们现在想要方便的在注释中写下自己的名字和注释的时间，于是我们给他起个名字叫 "://"



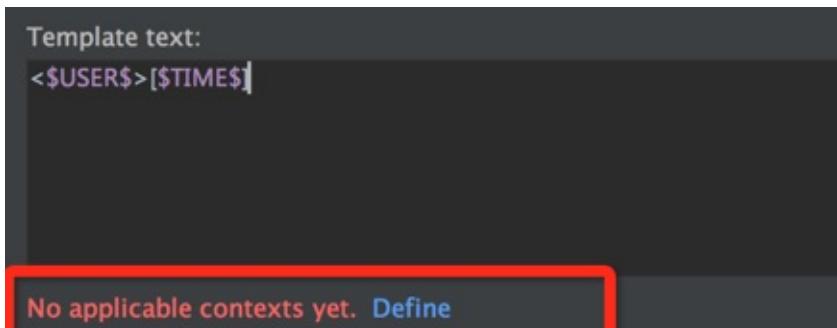
description 就是一个注释的作用，所以不写也可以（虽然这并不是一个好习惯）。接下来我们要写 Template Text了。我们希望输入 // 之后能自动生成下面的代码：

<你的用户名>[当前时间]

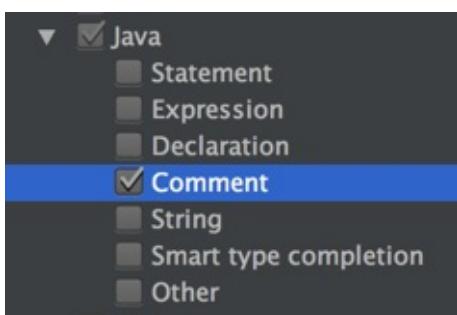
那么我们需要定义两个变量来代表用户名和时间：



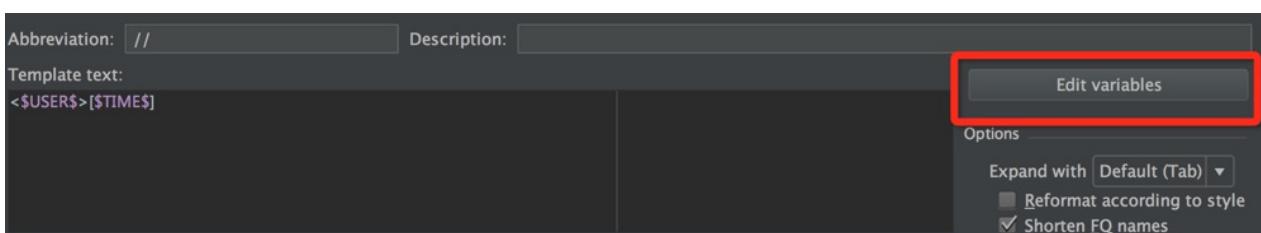
接下来我们要考虑，我们的这个模板用在哪里呢？



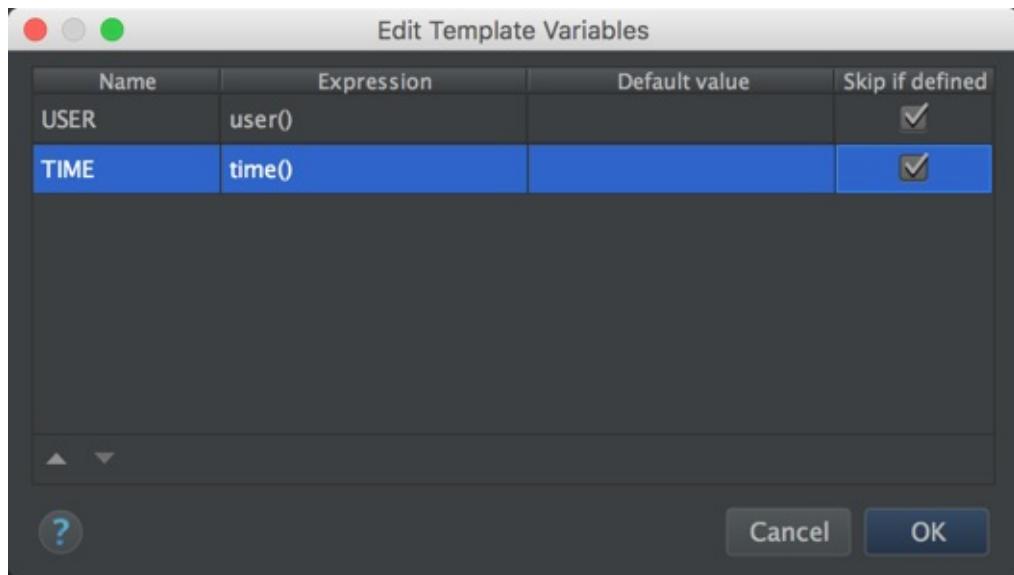
点击 Define，在弹出的菜单中选择 Java -> Comment：



紧接着要给这两个变量赋值了，点击旁边的 Edit Variables：



Expressions 有个下拉菜单，选出 user() 和 time() 赋给我们定义好的两个变量，同时勾上后面的两个勾 Skip if define——如果想知道为什么，你把勾去掉试试。

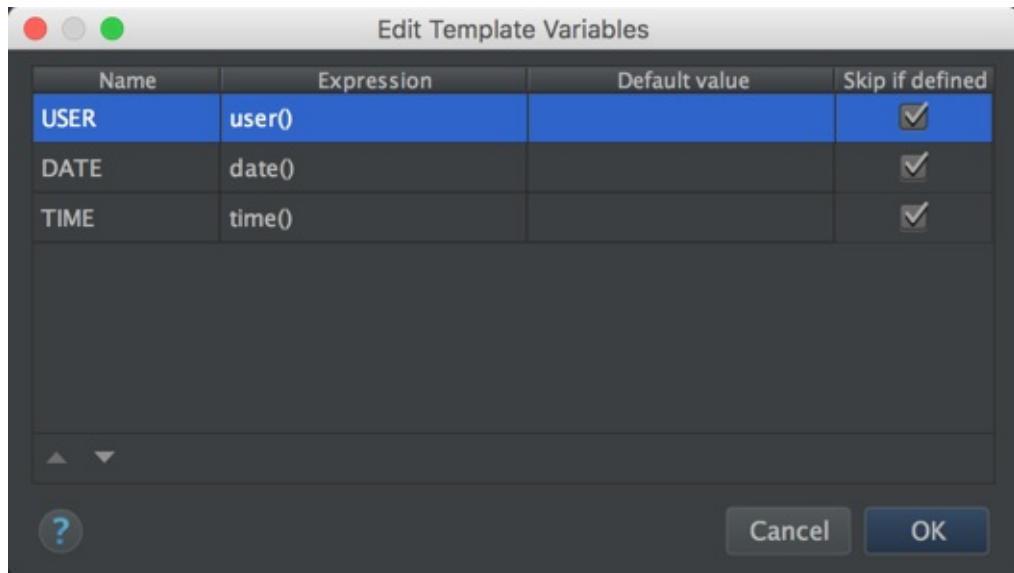


这样我们就创建好了一个新的模板，一路 OK之后，我们就可以试试了：

```
▶ public class JavaMain {  
▶   public static void main(String... args) {  
▶     }  
▶ }
```

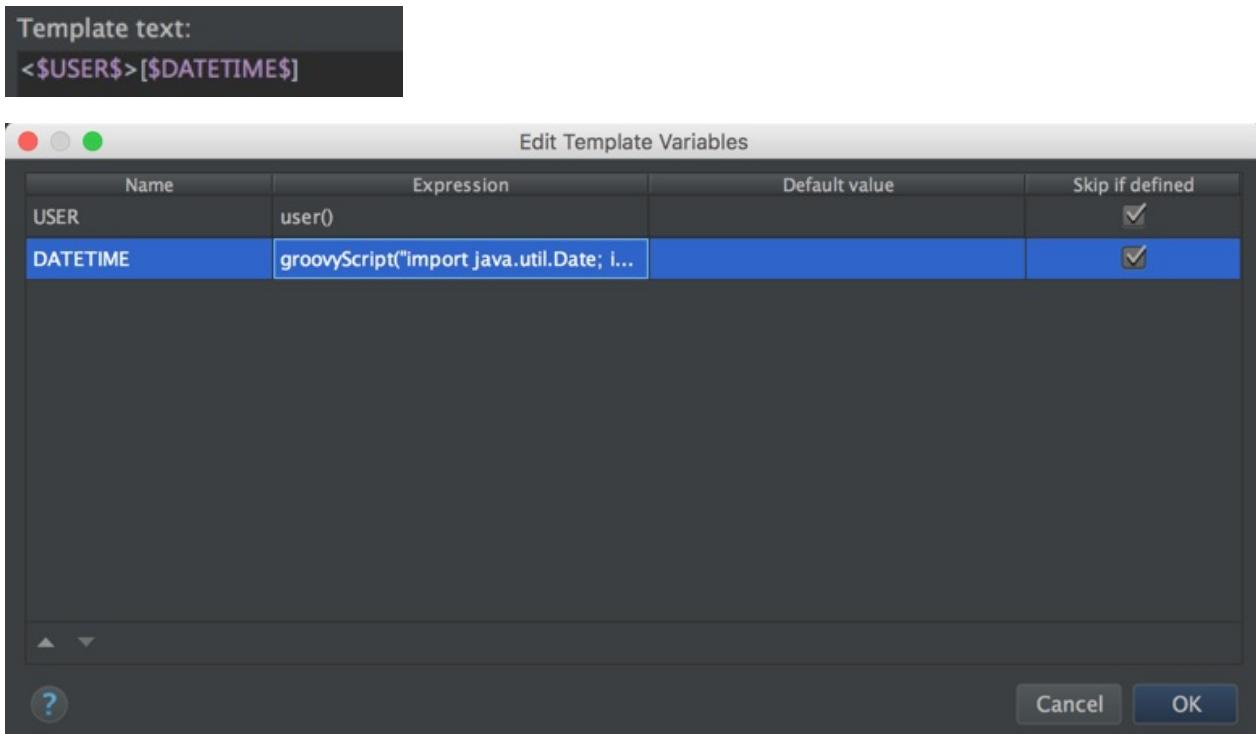
不过，你对时间格式有点儿不太满意，其实你还想要日期对不对？于是：

Template text:
<\$USER\$>[\$DATE\$ \$TIME\$]



```
public class JavaMain {  
    /**  
     * <benny>[09:36]  
     * @param args  
     */  
    public static void main(String... args) {  
    }  
}
```

其实这样基本就满足需求了。如果你偏偏对此感到不满意，为什么时间的格式不能自定义，那。。。我想告诉你的是，谁说不可以的。。



DATETIME 的表达式是：

```
groovyScript("import java.util.Date; import java.text.SimpleDateFormat; new SimpleDateFormat('yyyy-MM-dd HH:ss').format(new Date()));")
```

就是一段普通的 groovy 脚本，不过由于 groovy 完全兼容 Java 语法，所以你可以写一段 Java 代码进入，最后一行的值就是表达式的值。

结果嘛：

```
public class JavaMain {  
    /**  
     * <benny>[09:36]  
     * <benny>[1/8/17 09:39]  
     * @param args  
     */  
    public static void main(String... args) {  
    }  
}
```

小结

当然 IntelliJ 当中好玩的东西不只这些，比如 最常用的 refactor 等等，剩下的就需要大家自己去发现了，实在不行咱也可以写个插件或者去搜个别人做好的插件。IntelliJ 提供了很好的扩展性和定制性，每个人的背景和习惯不一样，对 IDE 的需求也不一样，只要肯折腾，这工具会越来越顺手的。

如何优雅的在微信公众号中编辑代码

这篇文章严格意义上是写给有公众号且公众号文章需要贴代码的朋友们看的。

1. 公众号编辑器真难用

自从入坑公众号以来，被公众号的这个编辑器简直折磨死了。我发的文章基本上是少不了贴代码的，可是每次贴上去的代码总是被公众号的编辑器无厘头的给我过滤掉换行符，简直气死人。

例如我编辑得非常好的代码，

```
Person person = new Person("benny", 27);
System.out.println(person.getName() + " is " + person.age); //b
enny is 27
person.setName("andy");
person.age = 26;
System.out.println(person.getName() + " is " + person.age); //a
ndy is 26
```

贴上去就成了这样：

```
Person person = new Person("benny", 27);System.out.println(pers
on.getName() + " is " + person.age); //benny is 27person.setName
("andy");person.age = 26;System.out.println(person.getName() + "
is " + person.age); //andy is 26
```

每次一段代码一段代码的敲回车，简直敲到手抽筋。醉了。

这还不算，这个格式的代码发到手机上，还不能水平滚动，各种任性的折行，让本来清秀的代码看起来真是一坨一坨的。

2. 分析下原因

查了一下，不少同行都在因为这个而感到苦恼，大家分析得出的结论是微信公众号的编辑器会对我们贴上去的内容进行处理，而处理的过程中又会对一些换行符进行过滤，导致本来排好的代码乱成一团。

3. 几种常见的贴代码的方法

3.1 贴图法

对于这种情况，最直接的方法自然是用工具渲染好，然后截图贴到公众号的编辑器里面。也可以写一个工具把代码自动绘成图片。

- 优点：简单直接，在找到更好的办法之前，我之前的几篇文章都是这么处理的。
- 缺点：操作繁琐，除了生成图片或者截图，还要手动贴到公众号编辑器中，截图时对于代码的文字清晰度控制也不太容易，贴到编辑器之后，还会被压缩，导致代码不容易被看清楚。当然，另一个更大的硬伤就是读者阅读的时候会耗费比较多的流量，如果网速不好还会加载不出来。

3.2 Markdown Here

很多朋友都提到 Markdown Here，这是一个非常棒的 Chrome 插件，大家可以搜索并添加它，安装之后可以在 Options 当中选择自己喜欢的主题，之后只要在选中编辑框，贴入 Markdown 源码，再点击插件的按钮即可。

The screenshot shows a browser-based code editor interface. At the top, there's a navigation bar with links like 'Others', 'jack'n jill', 'IT', 'Stetho', 'Tkinter tutorial', 'Vandal Software', 'Desmos Graphing...', and a search bar. Below the navigation bar, the title '写文章 已保存' is displayed. A toolbar with various icons for bold, italic, underline, etc., is visible. The main area contains Java code:

```
Person person = new Person("benny", 27);
System.out.println(person.getName() + " is " + person.age); //benny is 27
person.setName("andy");
person.age = 26;
System.out.println(person.getName() + " is " + person.age); //andy is 26
```
不过。。这个插件同样不能幸免于换行符的过滤，如果我们在微信公众号编辑器当中贴入 md 源码，按照上述操作渲染出漂亮的格式，保存之后预览，你就会发现这个办法其实并不怎么好用，或者说基本上不能用。
```

- 优点：这插件真的很棒。
- 缺点：因为渲染后的格式还是会惨遭微信公众号的编辑器的毒手，也就是说，你得自己手动处理一下换行的问题。

### 3.3 渲染好的格式直接贴

我们前面说，可以把 md 源码渲染好，然后直接复制粘贴到公众号的编辑器当中，结果其实与 Markdown Here 的效果完全一样。说到底还是换行符被过滤的问题。

我一直用的 md 工具是 MacDown，可以直接先用它编辑好文章，然后复制渲染结果贴到微信公众号当中，自行处理一下换行的问题即可。

- 优点：本地编辑，方便快捷。
- 缺点：一样，还是换行符的问题。

### 3.4 其他什么编辑器

当你在搜索引擎里面搜索这个问题的时候，大多数回答可能是推荐你用一些第三方的编辑器，比如什么 135 编辑器之类的，这些编辑器对于不写代码的朋友来说，真的挺好用的，里面提供了各式各样的模板，大家只要把文字准备好，一套用就立马狂拽酷炫吊炸天。

可是，它们并不是为我们这些码农准备的啊，你想想公众号绝大多数的运营者都是编辑，而不是程序员，他们根本不需要关心什么是代码，更不需要关心怎么把代码排版好（我相信微信公众号编辑器的开发小伙伴大概也是这么想的吧，这么久了这编辑器还是这鬼样子）。

第三方编辑器很多，不过，当你一个一个去试的时候，你会发现这条路根本不通！！

- 优点：适合编辑各种花哨的文字和图片。
- 缺点：不支持代码的格式。

## 4. 目前最优雅的方案诞生记

说实在的这个方案没有很高端。我们的痛点其实就是想个办法不让微信公众号编辑器对贴到里面的代码进行换行符的过滤。

有朋友提出把渲染后的结果每行都用  
替换 \n，这样过滤时，换行自然就不会被干掉了。

```

<code class="hljs language-java"
 style="...">Person person = new Person(<span
 class="hljs-string"
 style="color: rgb(196, 26, 22);">"benny",
 27
);
 System.out.println(person.getName() + " is " +
 person.age); //benny is 27
 person.setName("andy");
 person.age = 26;
 System.out.println(person.getName() + " is " +
 person.age); //andy is 26
</code>

```

这就是 md 渲染后的结果，代码对应于文章开头的例子。我们发现所有的代码被放到 code 这个标签当中，如果我们在其中用  
替换 \n 会发生什么呢？

```

Person person = new Person("benny", 27);
System.out.println(person.getName() + " is " + person.age); //b
enny is 27
person.setName("andy");
person.age = 26;
System.out.println(person.getName() + " is " + person.age); //a
ndy is 26

```

<br> 被直接显示出来了。根本不能用来换行。所以直接从结果入手好像没那么简单。那怎么办，我们干脆修改渲染方式吗？针对每行，用 <p /> 标签包起来不就可以换行了吗？

想法挺好啊。想到这里我就开始准备去修改 Markdown Here 的源码了。。。可，尼玛，我不是搞前端的，一片一片的 js 代码我该修改哪里呢？这下可尴尬了。

此路通，可能成本还是有点儿高了。于是我又回到原点，思考如何欺骗微信公众号编辑器那个换行符不能被过滤的问题。怎么骗呢？关键是，公众号的编辑器通过识别怎样的 pattern 来过滤换行符呢？如果我找到这个 pattern，然后把它破坏掉，不就可以了么？

经过一番尝试发现：

- 每一行代码的前后各加一个空格，那么正常有内容的行末换行符就不会被过滤
- 空行替换成“英文空格+中文空格+英文空格”，这样在微信公众号编辑器看起来似乎不是空行，不过在我们看来其实是空行

于是我动手写了一个简单的 python 脚本，那么后续我只要用 MacDown 编辑好我的文章，再用下面这个脚本自动为所有的代码加上空格、替换空行，那么我再把渲染好的文章贴到公众号的话，大功就告成了。

附上非常简单的脚本代码：

```
#!/usr/local/bin/python
encoding=utf-8

微信公众号编辑器对代码支持的不好，我们在贴代码之前需要在代码前后各加一个空格
并对空行做一下处理，主要随便搞几个空格和制表符
这样做可以防止公众号的编辑器过滤空行和换行符

import sys, re

匹配 md 的代码起始块，我习惯用 ``java 这样的形式，也可以根据需求进行修改
CODE_PATTERN_START = r"^\s*```\s*\w*$"
匹配代码块结束，
```

```
CODE_PATTERN_END = r"\s```\s$" if len(sys.argv) < 2: print "参数不正确." exit()
inputFile = open(sys.argv[1]) output = [] while True: line = inputFile.readline() if line: if re.match(CODE_PATTERN_START, line): output.append(line) while True: line =
```

```
inputFile.readline() if re.match(CODE_PATTERN_END, line): output.append(line) break
if line.strip():
```

```
 # 代码行前后各加一个空格
 line = " " + line
 line = line.replace("\n", " \n")
 print line
else:
 print "空行"
 # 替换空行，下面的空格中有一个是中文空格
 line = " \n"
 output.append(line)
else:
 output.append(line)
else:
 break
```

```
inputFile.close() outputFile = open(sys.argv[1], 'w+') outputFile.writelines(output)
outputFile.close() ````
```

## 5. 小结

这种方法应该算不上一个最终的方案（最终方案应该是公众号编辑器的开发该考虑的事儿），但它成本比较低，输出效果也比较不错，希望能给大家带来帮助。