**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. 介绍 3](#_Toc3890)

[2. 准备工作 3](#_Toc15835)

[3. 创建一个新的项目 3](#_Toc1260)

[3.1. 配置Gradle 3](#_Toc22917)

[3.2. 把MainActivity转换成Kotlin代码 4](#_Toc13703)

[3.3. 测试是否一切就绪 4](#_Toc20112)

[4. 类和函数 4](#_Toc3327)

[5. 编写你的第一个类 4](#_Toc17152)

[5.1. 创建一个layout 4](#_Toc25406)

[5.2. The Recycler Adapter 5](#_Toc14643)

[6. 变量和属性 5](#_Toc23289)

[6.1. 基本类型 5](#_Toc29608)

[6.2. 属性 6](#_Toc13516)

[7. Anko和扩展函数 7](#_Toc59)

[7.1. Anko是什么 7](#_Toc8202)

[7.2. 开始使用Anko 7](#_Toc28216)

[7.3. 扩展函数 7](#_Toc7949)

[8. 从API中获取数据 7](#_Toc3555)

[8.1. 执行一个请求 7](#_Toc783)

[8.2. 在主线程以外执行请求 8](#_Toc19822)

[9. 数据类 8](#_Toc25561)

[9.1. 复制一个数据类 8](#_Toc9843)

[9.2. 自动创建构析声明 8](#_Toc22978)

[10. 解析数据 9](#_Toc26733)

[10.1. 转换json到数据类 9](#_Toc13529)

[10.2. 构建domain层 9](#_Toc20418)

[10.3. 在UI中绘制数据 10](#_Toc558)

[11. 操作符重载 11](#_Toc5728)

[11.1. 操作符表 11](#_Toc30872)

[11.2. 例子 11](#_Toc20985)

[11.3. 扩展函数中的操作符 12](#_Toc28775)

[12. 使Forecast list可点击 12](#_Toc21856)

[13. Lambda 12](#_Toc26077)

[13.1. 扩展语言 12](#_Toc13721)

[14. 可见性修饰符 12](#_Toc17527)

[15. Kotlin Android Extensions 13](#_Toc22680)

[15.1. 怎么去使用Kotlin Android Extensions 13](#_Toc26937)

[15.1.1. Activities或者Fragments的Android Extensions 13](#_Toc25280)

[15.1.2. Views的Android Extensions 13](#_Toc31899)

[16. Application单例化和属性的Delegated 13](#_Toc16806)

[16.1. Applicaton单例化 13](#_Toc2268)

[16.2. 委托属性 14](#_Toc17832)

[16.3. 标准委托 14](#_Toc6064)

[16.3.1. Lazy 14](#_Toc2373)

[16.3.2. Observable 15](#_Toc15020)

[16.3.3. Vetoable 15](#_Toc7280)

[16.3.4. Not Null 15](#_Toc15336)

[16.3.5. 从Map中映射值 16](#_Toc23337)

[16.4. 怎么去创建一个自定义的委托 16](#_Toc24729)

[16.5. 重新实现Application单例化 17](#_Toc21023)

[17. 创建一个SQLiteOpenHelper 17](#_Toc13761)

[17.1. ManagedSqliteOpenHelper 18](#_Toc26923)

[17.2. 定义表 18](#_Toc24362)

[17.3. 实现SqliteOpenHelper 19](#_Toc2790)

[17.4. 依赖注入 20](#_Toc25400)

[18. 集合和函数操作符 20](#_Toc31142)

[18.1. 总数操作符 21](#_Toc604)

[18.2. 过滤操作符 22](#_Toc17829)

[18.3. 映射操作符 23](#_Toc24377)

[18.4. 元素操作符 23](#_Toc13601)

[18.5. 生产操作符 24](#_Toc9389)

[18.6. 顺序操作符 25](#_Toc14001)

[19. 从数据库中保存或查询数据 26](#_Toc347)

[19.1. 创建数据库model类 26](#_Toc30510)

[19.2. 写入和查询数据库 27](#_Toc19572)

[19.2.1. 创建ForecastDb 27](#_Toc16341)

[19.2.2. 查询一个forecast 27](#_Toc1482)

[19.2.3. 保存一个forecast 28](#_Toc9873)

[20. Kotlin中的null安全 30](#_Toc13171)

[20.1. 可null类型怎么工作 30](#_Toc10646)

[21. 创建业务逻辑来访问数据 31](#_Toc29060)

[22. Flow control 和ranges 33](#_Toc22693)

[23. 创建一个详情界面 33](#_Toc27823)

[23.1. 获取color 33](#_Toc443)

[23.2. 启动一个activity：reified函数 33](#_Toc1737)

[24. 接口和委托 34](#_Toc17404)

[24.1. 接口 34](#_Toc7296)

[24.2. 委托 35](#_Toc17165)

[24.3. 在我们的App中实现一个例子 35](#_Toc24012)

[25. 泛型 38](#_Toc17990)

[25.1. 基础 38](#_Toc30436)

[26. 设置界面 39](#_Toc10844)

[26.1. 创建一个设置activity 39](#_Toc30127)

[26.2. 访问Shared Preferences 39](#_Toc372)

[26.3. 泛型preference委托 41](#_Toc30744)

[27. 测试你的app 42](#_Toc23746)

[27.1. Unit testing 42](#_Toc20910)

[27.2. Instrumentation tests 44](#_Toc21533)

[28. 其他的概念 46](#_Toc2922)

[28.1. 内部类 46](#_Toc17196)

[28.2. 枚举 46](#_Toc30363)

[28.3. 密封（Sealed）类 47](#_Toc12576)

[28.4. 异常（Exceptions） 47](#_Toc5193)

[29. 高阶函数、表达式函数、块主体函数的返回值 50](#_Toc16149)

[30. kotlin使用Parcelize注解简化Parcelable 50](#_Toc23690)

[31. let,apply,with,run等函数的区别 50](#_Toc6795)

[31.1. Standard.Kt代码 50](#_Toc3486)

[31.2. 前提介绍 52](#_Toc7317)

[31.3. with 53](#_Toc10445)

[31.4. let 53](#_Toc17339)

[31.5. apply 54](#_Toc13151)

[31.6. run 55](#_Toc26011)

[31.7. 另一个run（非扩展函数） 56](#_Toc17443)

[31.8. repeat 56](#_Toc7877)

[31.9. also 56](#_Toc27232)

[31.10. takeIf 57](#_Toc31350)

[31.11. takeUnless 57](#_Toc8492)

[31.12. 举例一，赋值： 58](#_Toc22805)

[31.13. 举例二，response： 58](#_Toc23149)

[31.14. 总结 59](#_Toc5019)

[31.14.1. let函数使用场景 60](#_Toc18208)

[31.14.2. with函数使用场景 60](#_Toc2901)

[31.14.3. run函数的使用场景 60](#_Toc28311)

[31.14.4. apply函数的使用场景 60](#_Toc4258)

[31.14.5. also函数的使用场景 61](#_Toc29889)

[32. 与Java对比运算符 61](#_Toc8337)

[32.1. Kotlin与Java相同的运算符 61](#_Toc12235)

[32.2. Kotlin与Java使用有差异的运算符 62](#_Toc14481)

[32.2.1. ==和!= 62](#_Toc28558)

[32.2.2. [] 62](#_Toc14488)

[32.3. Java中有而Kotlin中没有的运算符 62](#_Toc11689)

[32.4. Kotlin中有而Java中没有的运算符 63](#_Toc20317)

[32.4.1. is和!is 63](#_Toc21306)

[32.4.2. in和!in 63](#_Toc2469)

[32.4.3. .. 63](#_Toc27026)

[33. 写给Android开发者的Kotlin入门 63](#_Toc9390)

[33.1. Kotlin的特色 63](#_Toc30044)

[33.1.1. 简化findViewById 63](#_Toc1342)

[33.2. Anko 63](#_Toc30723)

[33.3. Kotlin语法特点 65](#_Toc32739)

[33.4. 函数式编程 70](#_Toc12648)

[33.4.1. 什么是编程 70](#_Toc17501)

[33.4.2. 纯函数 72](#_Toc19490)

[33.4.3. 不变性 73](#_Toc3414)

[33.4.4. 高阶函数 74](#_Toc12510)

[33.4.5. 闭包 74](#_Toc32462)

[33.4.6. Lambda表达式 75](#_Toc24983)

[33.4.7. 流式（Stream）API 75](#_Toc12513)

[33.5. Kotlin的潜在问题 76](#_Toc961)

[33.5.1. 思维方式的问题 76](#_Toc4568)

[33.5.2. Kotlin->Java的转换 76](#_Toc14493)

[33.5.3. 团队开发的问题 76](#_Toc24993)

[34. 协程Coroutine 76](#_Toc7499)

# 介绍

# 准备工作

# 创建一个新的项目

## 配置Gradle

Kotlin插件包括一个让我们配置Gradle的工具。但是我还是倾向于保持我对Gradle文件读写的控制权，否则它只会变得混乱而不会变得简单。所以这次，我们将手动去做。

首先，你需要如下修改project的build.gradle：

buildscript {

ext.support\_version = '23.1.1'

ext.kotlin\_version = '1.0.0'

ext.anko\_version = '0.8.2'

repositories {

jcenter()

dependencies {

classpath 'com.android.tools.build:gradle:1.5.0'

classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin\_version"

}

}

}

allprojects {

repositories {

jcenter()

}

}

我们创建了一个变量来存储当前的Kotlin版本，对于support library也是如此，Anko库也是同样的做法。用这个方式可以更方便地在一个地方修改所有的版本号。并且使用相同的版本号，更新的时候也不需要每个地方都修改。

我们会增加Kotlin标准库，Anko库，以及Kotlin和Kotlin Android Extensions plugin插件到dependencies。

apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {

...

}

dependencies {

compile "com.android.support:appcompat-v7:$support\_version"

compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin\_version"

compile "org.jetbrains.anko:anko-common:$anko\_version"

}

buildscript {

repositories {

jcenter()

}

dependencies {

classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin\_version"

}

}

Anko是一个用来简化一些Android任务的很强大的Kotlin库。我们之后将会学习部分anko，但是现在来说仅仅增加anko-common就足够了。这个库被分割成了一系列小的部分以至于我们不会把没用到的部分加进来。

## 把MainActivity转换成Kotlin代码

Kotlin plugin包含了一个有趣的特性，它能把Java代码转成Kotlin代码。正如任何自动化那样，结果不会很完美，所以我们在MainActivity.java类中使用它。打开文件，然后选择Code -> Convert Java File to Kotlin File

## 测试是否一切就绪

首先，打开activity\_main.xml，然后设置TextView的id：

<TextView

android:id="@+id/message"

android:text="@string/hello\_world"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

然后，手动在Activity中增加一个import语句（不要担心你现在对这个还不太理解）。

import kotlinx.android.synthetic.main.activity\_main.\*

在onCreate中，你现在可以直接得到并访问这个TextView了。

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

message.text = "Hello Kotlin!"

}

现在运行这个app，并且它是正常运行的。检查TextView是否是显示的新的内容。如果你有疑问或者想查看代码，请在<https://github.com/antoniolg/Kotlin-for-Android-Developers>查看

# 类和函数

# 编写你的第一个类

## 创建一个layout

显示天气预报的列表我们使用RecyclerView，所以你需要在build.gradle中增加一个新的依赖：

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

compile "com.android.support:appcompat-v7:$support\_version"

compile "com.android.support:recyclerview-v7:$support\_version"

}

然后，activity\_main.xml如下：

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<android.support.v7.widget.RecyclerView

android:id="@+id/forecast\_list"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"/>

</FrameLayout>

暂且我们使用老的findViewByid()的方式：

val forecastList = findViewById(R.id.forecast\_list) as RecyclerView

forecastList.layoutManager = LinearLayoutManager(this)

## The Recycler Adapter

我们同样需要一个RecyclerView的Adapter。RecyclerView中所使用到的布局现在只需要一个TextView，我会手动去创建这个简单的文本列表。增加一个名为ForecastListAdapter.kt的Kotlin文件，包括如下代码：

class ForecastListAdapter(val items: List<String>) :

RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {

return ViewHolder(TextView(parent.context))

}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {

holder.textView.text = items[position]

}

override fun getItemCount(): Int = items.size

class ViewHolder(val textView: TextView) : RecyclerView.ViewHolder(textView)

}

回到MainActivity，现在简单地创建一系列的String放入List中，然后使用创建分配Adapter实例。

private val items = listOf(

"Mon 6/23 - Sunny - 31/17",

"Tue 6/24 - Foggy - 21/8",

"Wed 6/25 - Cloudy - 22/17",

"Thurs 6/26 - Rainy - 18/11",

"Fri 6/27 - Foggy - 21/10",

"Sat 6/28 - TRAPPED IN WEATHERSTATION - 23/18",

"Sun 6/29 - Sunny - 20/7"

)

override fun onCreate(savedInstanceState: Bundle?) {

...

forecastList.adapter = ForecastListAdapter(items)

}

我们在上面很简短的代码中看到了很多新的东西，所以我会在下一章讲到它们。我们现在必须要学习一些比如基本类型，变量，属性等比较重要的概念才能继续下去。

# 变量和属性

在Kotlin中，一切都是对象。没有像Java中那样的原始基本类型。这个是非常有帮助的，因为我们可以使用一致的方式来处理所有的可用的类型

## 基本类型

像integer，float或者boolean等类型仍然存在，但是它们全部都会作为对象存在的。基本类型的名字和它们工作方式都是与Java非常相似的，但是有一些不同之处你可能需要考虑到：

数字类型中不会自动转型

举个例子，你不能给Double变量分配一个Int。必须要做一个明确的类型转换，可以使用众多的函数之一：

val i:Int=7

val d: Double = i.toDouble()

字符（Char）不能直接作为一个数字来处理

在需要时我们需要把他们转换为一个数字：

val c:Char='c'

val i: Int = c.toInt()

位运算也有一点不同

// Java

int bitwiseOr = FLAG1 | FLAG2;

int bitwiseAnd = FLAG1 & FLAG2;

// Kotlin

val bitwiseOr = FLAG1 or FLAG2

val bitwiseAnd = FLAG1 and FLAG2

类型推断

val i = 12 // An Int

val iHex = 0x0f // 一个十六进制的Int类型

val l = 3L // A Long

val d = 3.5 // A Double

val f = 3.5F // A Float

一个String可以像数组那样访问，并且被迭代：

val s = "Example"

val c = s[2] // 这是一个字符'a'

// 迭代String

val s = "Example"

for(c in s){

print(c)

}

## 属性

属性与Java中的字段是相同的，但是更加强大。我们通过一个例子来比较他们的不同之处

/\*Java\*/

public class Person {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Person person = new Person();

person.setName("name");

String name = person.getName();

/\*在Kotlin中，只需要一个属性就可以了\*/

public class Person {

var name: String = ""

}

val person = Person()

person.name = "name"

val name = person.name

如果没有任何指定，属性会默认使用getter和setter。当然它也可以修改为你自定义的代码，并且不修改存在的代码：

public class Person {

var name: String = ""

get() = field.toUpperCase()

set(value){

field = "Name: $value"

}

}

如果需要在getter和setter中访问这个属性自身的值，它需要创建一个field。可以使用field这个预留字段来访问，它会被编译器找到正在使用的并自动创建。

# Anko和扩展函数

## Anko是什么

Anko是JetBrains开发的一个强大的库。它主要的目的是用来替代以前XML的方式来使用代码生成UI布局。这是一个很有趣的特性，我推荐你可以尝试下，但是我在这个项目中暂时不使用它。对于我（可能是由于多年的UI绘制经验）来说使用XML更容易一些，但是你会喜欢那种方式的。

然而，这个不是我们能在这个库中得到的唯一一个功能。Anko包含了很多的非常有帮助的函数和属性来避免让你写很多的模版代码。建议你要理解这个背后到底做了什么。你可以在任何时候使用ctrl + 点击跳转到Anko的源代码。

## 开始使用Anko

在之前，让我们来使用Anko来简化一些代码。就像你将看到的，任何时候你使用了Anko库中的某些东西，它们都会以属性名、方法等方式被导入。这是因为Anko使用了扩展函数在Android框架中增加了一些新的功能

在MainActivity:onCreate，Anko扩展函数find可以被用来简化获取一个RecyclerView：

inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T

val forecastList: RecyclerView = find(R.id.forecast\_list)

我们现在还不能使用库中更多的东西，但是Anko能帮助我们简化代码，比如，实例化Intent，Activity之间的跳转，Fragment的创建，数据库的访问，Alert的创建等，我们将会在实现这个App的过程中学习到很多有趣的例子。

## 扩展函数

Anko已经给Context和Fragment扩展了toast和longToast函数,参数可以传CharSequence和Int：

fun Context.toast(textResource: Int) = Toast.makeText(this, textResource, Toast.LENGTH\_SHORT).show()

inline fun Fragment.toast(text: CharSequence): Unit = activity.toast(text)//activity实际是getActivity()

# 从API中获取数据

## 执行一个请求

我们将会使用OpenWeatherMap API来获取数据，还有一些普通类来现实这个请求。多亏Kotlin非常强大的互操作性，你可以使用任何你想使用的库，比如用Retrofit来执行服务器请求。

当只是执行一个简单的API请求，我们可以不使用任何第三方库来简单地实现。 Kotlin提供了一些扩展函数来让请求变得更简单。首先，我们要创建一个新的Request类：

public inline fun URL.readText(charset: Charset = Charsets.UTF\_8): String = readBytes().toString(charset)

public fun URL.readBytes(): ByteArray = openStream().use { it.readBytes() }

public final InputStream openStream() throws java.io.IOException {

return openConnection().getInputStream();

}

public class Request(val url: String) {

public fun run() {

val forecastJsonStr = URL(url).readText()

Log.d(javaClass.simpleName, forecastJsonStr)

}

}

readText函数，这是Kotlin标准库中的扩展函数。这个方法不推荐结果很大的响应，但是在我们这个例子中已经足够好了。如果你用这些代码去比较Java，你会发现我们仅使用标准库就节省了大量的代码。比如HttpURLConnection、BufferedReader和需要达到相同效果所必要的迭代结果，管理连接状态、reader等部分的代码。很明显，这些就是场景背后函数所作的事情，但是我们却不用关心。

为了可以执行请求，App必须要有Internet权限。所以需要在AndroidManifest.xml中添加：：

<uses-permission android:name="android.permission.INTERNET" />

## 在主线程以外执行请求

如你所知，HTTP请求不被允许在主线程中执行，否则它会抛出异常。这是因为阻塞住UI线程是一个非常差的体验。Android中通用的做法是使用AsyncTask，但是这些类是非常丑陋的，并且使用它们无任何副作用地实现功能也是非常困难的。如果你使用不小心，AsyncTasks会非常危险，因为当运行到postExecute时，如果Activity已经被销毁了，这里就会崩溃。

Anko提供了非常简单的DSL来处理异步任务，它满足大部分的需求。它提供了一个基本的async函数用于在其它线程执行代码，也可以选择通过调用uiThread的方式回到主线程。在子线程中执行请求如下这么简单：

fun <T> T.async(task: AnkoAsyncContext<T>.() -> Unit): Future<Unit> {

val context = AnkoAsyncContext(WeakReference(this))

return BackgroundExecutor.submit { context.task() }

}

fun <T> AnkoAsyncContext<T>.uiThread(f: (T) -> Unit) {

val ref = weakRef.get() ?: return

if (ContextHelper.mainThread == Thread.currentThread()) {

f(ref)

} else {

ContextHelper.handler.post { f(ref) }

}

}

async() {

Request(url).run()

uiThread { longToast("Request performed") }

}

uiThread有一个很不错的一点就是可以依赖于调用者。如果它是被一个Activity调用的，那么如果activity.isFinishing()返回true，则uiThread不会执行，这样就不会在Activity销毁的时候遇到崩溃的情况了。

# 数据类

通常只提供了用于访问它们属性的简单的getter和setter。定义一个新的数据类非常简单：

data class Forecast(val date: Date, val temperature: Float, val details: String)

## 复制一个数据类

如果我们使用不可修改的对象，就像我们之前讲过的，假如我们需要修改这个对象状态，必须要创建一个新的一个或者多个属性被修改的实例。这个任务是非常重复且不简洁的。举个例子，如果我们需要修改Forecast中的temperature（温度），我们可以这么做：

val f1 = Forecast(Date(), 27.5f, "Shiny day")

val f2 = f1.copy(temperature = 30f)

## 自动创建构析声明

映射对象的每一个属性到一个变量中，这个过程就是我们知道的解构声明。数据类的componentX函数被自动创建。使用上面的Forecast类举个例子：

val f1 = Forecast(Date(), 27.5f, "Shiny day")

val (date, temperature, details) = f1

上面这个多声明会被编译成下面的代码：

val date = f1.component1()

val temperature = f1.component2()

val details = f1.component3()

这个特性背后的逻辑是非常强大的，它可以在很多情况下帮助我们简化代码。举个例子，Map类含有一些扩展函数的实现，允许它在迭代时使用key和value：

for ((key, value) in map) {

Log.d("map", "key:$key, value:$value")

}

# 解析数据

## 转换json到数据类

在date包中，创建一个名为ResponseClasses.kt新的文件

data class ForecastResult(val city: City, val list: List<Forecast>)

data class City(val id: Long, val name: String, val coord: Coordinates, val country: String, val population: Int)

data class Coordinates(val lon: Float, val lat: Float)

data class Forecast(val dt: Long, val temp: Temperature, val pressure: Float,

val humidity: Int, val weather: List<Weather>,

val speed: Float, val deg: Int, val clouds: Int,

val rain: Float)

data class Temperature(val day: Float, val min: Float, val max: Float, val night: Float, val eve: Float, val morn: Float)

data class Weather(val id: Long, val main: String, val description: String, val icon: String)

把这个静态的url放在一个companion object（伴随对象）中，以下是最后的代码

public class ForecastRequest(val zipCode: String) {

companion object {

private val APP\_ID = "15646a06818f61f7b8d7823ca833e1ce"

private val URL = "http://api.openweathermap.org/data/2.5/" +

"forecast/daily?mode=json&units=metric&cnt=7"

private val COMPLETE\_URL = "$URL&APPID=$APP\_ID&q="

}

fun execute(): ForecastResult {

val forecastJsonStr = URL(COMPLETE\_URL + zipCode).readText()

return Gson().fromJson(forecastJsonStr, ForecastResult::class.java)

}

}

记得在build.gradle中增加你需要的Gson依赖：

compile "com.google.code.gson:gson:2.8.2"

## 构建domain层

我们现在创建一个新的包作为domain层。这一层中会包含一些Commands的实现来为app执行任务。首先，必须要定义一个Command：

public interface Command<T> {

fun execute(): T

}

这个command会执行一个操作并且返回某种类型的对象，这个类型可以通过范型被指定

第一个command需要去请求天气预报结构然后转换结果为domain类。下面这些类会在domain类中被定义：

data class ForecastList(val city: String, val country: String, val dailyForecast:List<Forecast>)

data class Forecast(val date: String, val description: String, val high: Int, val low: Int)

当更多的功能被增加，这些类可能会需要在以后被审查。但是现在这些类对我们来说已经足够了。这些类必须从数据映射到我们的domain类，所以我接下来需要创建一个DataMapper：

public class ForecastDataMapper {  
  
 fun convertFromDataModel(forecast: ForecastResult): ForecastList {  
 return ForecastList(forecast.city.name, forecast.city.country, convertForecastListToDomain(forecast.list))  
 }  
  
 private fun convertForecastListToDomain(list: List<Forecast>): List<ModelForecast> {  
 return list.*map* **{** convertForecastItemToDomain(**it**) **}** }  
  
 private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {  
 return ModelForecast(convertDate(forecast.dt),  
 forecast.weather[0].description, forecast.temp.max.toInt(),  
 forecast.temp.min.toInt())  
 }  
  
 private fun convertDate(date: Long): String {  
 val df = DateFormat.getDateInstance(DateFormat.*MEDIUM*, Locale.getDefault())  
 return df.format(date \* 1000)  
 }  
}

当我们使用了两个相同名字的类，我们可以给其中一个指定一个别名，这样我们就不需要写完整的包名了：

import com.cgy.kotlinforandroidtest.date.Forecast

import com.cgy.kotlinforandroidtest.domain.Forecast as ModelForecast

现在，编写命令前的准备就绪：

class RequestForecastCommand(val zipCode: String) : Command<ForecastList> {

override fun execute(): ForecastList {

val forecastRequest = ForecastRequest(zipCode)

return ForecastDataMapper().convertFromDataModel(

forecastRequest.execute())

}

}

## 在UI中绘制数据

MainActivity中的代码有些小的改动，因为现在有真实的数据需要填充到adapter中。异步调用需要被重写成：

async() {

val result = RequestForecastCommand("94043").execute()

uiThread{

forecastList.adapter = ForecastListAdapter(result)

}

}

Adapter也需要被修改：

class ForecastListAdapter(val weekForecast: ForecastList) : RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {

return ViewHolder(TextView(parent.getContext()))

}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {

with(weekForecast.dailyForecast[position]) {

holder.textView.text = "$date - $description - $high/$low"

}

}

override fun getItemCount(): Int = weekForecast.dailyForecast.size

class ViewHolder(val textView: TextView) : RecyclerView.ViewHolder(textView)

}

# 操作符重载

## 操作符表



## 例子

Kotlin List是实现了数组操作符的，所以我们可以像Java中的数组一样访问List的每一项。除此之外：在可修改的List中，每一项也可以用一个简单的方式被直接设置：

val x = myList[2]

myList[2] = 4

我们有一个叫ForecastList的数据类，它是由很多其他额外的信息组成的。我们可以重载操作符直接访问它的每一项而不是请求内部的list得到某一项。

做一个完全不相关的事情，我要去实现一个size()方法，它能稍微能简化一点当前的Adapter：

data class ForecastList(val city: String, val country: String, val dailyForecast: List<Forecast>) {

operator fun get(position: Int): Forecast = dailyForecast[position]

fun size(): Int = dailyForecast.size

}

它会使我们的onBindViewHolder更简单一点：

override fun onBindViewHolder(holder: ViewHolder, position: Int) {

with(weekForecast[position]) {

holder.textView.text = "$date - $description - $high/$low"

}

}

当然还有getItemCount()方法：

override fun getItemCount(): Int = weekForecast.size()

## 扩展函数中的操作符

我们不需要去扩展我们自己的类，但是我需要去使用扩展函数扩展我们已经存在的类来让第三方的库能提供更多的操作。几个例子，我们可以去像访问List的方式去访问ViewGroup的view：

operator fun ViewGroup.get(position: Int): View = getChildAt(position)

现在真的可以非常简单地从一个ViewGroup中通过position得到一个view：

val container: ViewGroup = find(R.id.container)

val view = container[2]

# 使Forecast list可点击

作为一个真正的app，当前列表的每一个item布局应该做一些工作。第一件事就是创建一个合适的XML，能符合我们的需要就行。让我们创建一个名为item\_forecast.xml的layout：

Adapter也需要一个大的改动了。还需要一个click listener，我们来定义它：

public interface OnItemClickListener {

operator fun invoke(forecast: Forecast)

}

当被调用时invoke方法可以被省略。所以我们来使用它来简化。listener可以被以下两种方式调用：

itemClick.invoke(forecast)

itemClick(forecast)

在Adapter中

itemView.setOnClickListener { itemClick(forecast) }

Anko提供了大量的扩展函数来让Android编程更简单。举个例子，activitys、fragments以及其它包含了ctx这个属性，通过ctx这个属性来返回context，但是在View中缺少这个属性。所以我们要创建一个新的名叫ViewExtensions.kt文件来代替ui.utils，然后增加这个扩展属性：

inline val Fragment.ctx: Context

get() = activity

inline val Context.ctx: Context

get() = this

inline val Activity.act: Activity

get() = this

val View.ctx: Context

get() = context

# Lambda

## 扩展语言

我们可以创建代码块只提供Lollipop或者更高版本来执行：

inline fun supportsLollipop(code: () -> Unit) {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.LOLLIPOP) {

code()

}

}

它只是检查版本，然后如果满足条件则去执行。现在我们可以这么做：

supportsLollipop {

window.setStatusBarColor(Color.BLACK)

}

# 可见性修饰符

# Kotlin Android Extensions

另一个Kotlin团队研发的可以让开发更简单的插件是Kotlin Android Extensions。当前仅仅包括了view的绑定。这个插件自动创建了很多的属性来让我们直接访问XML中的view。这种方式不需要你在开始使用之前明确地从布局中去找到这些views。

这些属性的名字就是来自对应view的id，所以我们取id的时候要十分小心，因为它们将会是我们类中非常重要的一部分。这些属性的类型也是来自XML中的，所以我们不需要去进行额外的类型转换。

Kotlin Android Extensions的一个优点是它不需要在我们的代码中依赖其它额外的库。它仅仅由插件组层，需要时用于生成工作所需的代码，只需要依赖于Kotlin的标准库。

那它背后是怎么工作的？该插件会代替任何属性调用函数，比如获取到view并具有缓存功能，以免每次属性被调用都会去重新获取这个view。需要注意的是这个缓存装置只会在Activity或者Fragment中才有效。如果它是在一个扩展函数中增加的，这个缓存就会被跳过，因为它可以被用在Activity中但是插件不能被修改，所以不需要再去增加一个缓存功能。

## 怎么去使用Kotlin Android Extensions

我们有两个方法来使用它：

### Activities或者Fragments的Android Extensions

这是最典型的使用方式。它们可以作为activity或fragment的属性是可以被访问的。属性的名字就是XML中对应view的id。

我们需要使用的import语句以kotlin.android.synthetic开头，然后加上我们要绑定到Activity的布局XML的名字：

import kotlinx.android.synthetic.activity\_main.\*

此后，我们就可以在setContentView被调用后访问这些view。新的Android Studio版本中可以通过使用include标签在Activity默认布局中增加内嵌的布局。很重要的一点是，针对这些布局，我们也需要增加手工的import：

import kotlinx.android.synthetic.activity\_main.\*

import kotlinx.android.synthetic.content\_main.\*

### Views的Android Extensions

前面说的使用还是有局限性的，因为可能有很多代码需要访问XML中的view。比如，一个自定义view或者一个adapter。举个例子，绑定一个xml中的view到另一个view。唯一不同的就是需要import：

import kotlinx.android.synthetic.view\_item.view.\*

如果我们需要一个adapter，比如，我们现在要从inflater的View中访问属性：

val view = LayoutInflater.from(parent.ctx).inflate(R.layout.view\_item, parent, false)

view.textView.text = "Hello"

# Application单例化和属性的Delegated

我们很快要去实现一个数据库，如果我们想要保持我们代码的简洁性和层次性（而不是把所有代码添加到Activity中），我们就要需要有一个更简单的访问application context的方式。

## Applicaton单例化

按照我们在Java中一样创建一个单例最简单的方式：

class App : Application() {

companion object {

private var instance: Application? = null

fun instance() = instance!!

}

override fun onCreate() {

super.onCreate()

instance = this

}

}

Android有一个问题，就是我们不能去控制很多类的构造函数。比如，我们不能初始化一个非null属性，因为它的值需要在构造函数中去定义。所以我们需要一个可null的变量，和一个返回非null值的函数。我们知道我们一直都有一个App实例，但是在它调用onCreate之前我们不能去操作任何事情，所以我们为了安全性，我们假设instance()函数将会总是返回一个非null的app实例。

但是这个方案看起来有点不自然。我们需要定义个一个属性（已经有了getter和setter），然后通过一个函数来返回那个属性。我们有其他方法去达到相似的效果么？是的，我们可以通过委托这个属性的值给另外一个类。这个就是我们知道的委托属性。

## 委托属性

我们可能需要一个属性具有一些相同的行为，使用lazy或者observable可以被很有趣地实现重用。而不是一次又一次地去声明那些相同的代码，Kotlin提供了一个委托属性到一个类的方法。这就是我们知道的委托属性。

当我们使用属性的get或者set的时候，属性委托的getValue和setValue就会被调用。

属性委托的结构如下：

class Delegate<T> : ReadWriteProperty<Any?, T> {

override fun getValue(thisRef: Any?, property: KProperty<\*>): T {

return ...

}

override fun setValue(thisRef: Any?, property: KProperty<\*>, value: T) {

...

}

}

或者直接写

public interface ReadOnlyProperty<in R, out T> {

public operator fun getValue(thisRef: R, property: KProperty<\*>): T

}

public interface ReadWriteProperty<in R, T> {

public operator fun getValue(thisRef: R, property: KProperty<\*>): T

public operator fun setValue(thisRef: R, property: KProperty<\*>, value: T)

}

class Delegate<T> {

operator fun getValue(thisRef: Any?, property: KProperty<\*>): T {

return ...

}

operator fun setValue(thisRef: Any?, property: KProperty<\*>, value: T) {

...

}

}

这个T是委托属性的类型。getValue函数接收一个类的引用和一个属性的元数据。setValue函数又接收了一个被设置的值。如果这个属性是不可修改（val），就会只有一个getValue函数。

下面展示属性委托是怎么设置的，它使用了by这个关键字来指定一个委托对象：

class Example {

var p: String by Delegate()

}

## 标准委托

在Kotlin的标准库中有一系列的标准委托。它们包括了大部分有用的委托，但是我们也可以创建我们自己的委托。

### Lazy

public fun <T> lazy(initializer: () -> T): Lazy<T> = SynchronizedLazyImpl(initializer)

它包含一个lambda，当第一次执行getValue的时候这个lambda会被调用，所以这个属性可以被延迟初始化。之后的调用都只会返回同一个值。这是非常有趣的特性， 当我们在它们第一次真正调用之前不是必须需要它们的时候。我们可以节省内存，在这些属性真正需要前不进行初始化。

class App : Application() {

val database: SQLiteOpenHelper by lazy {

MyDatabaseHelper(applicationContext)

}

override fun onCreate() {

super.onCreate()

val db = database.writableDatabase

}

}

在这个例子中，database并没有被真正初始化，直到第一次调用onCreate时。在那之后，我们才确保applicationContext存在，并且已经准备好可以被使用了。lazy操作符是线程安全的。

如果你不担心多线程问题或者想提高更多的性能，你也可以使用lazy(LazyThreadSafeMode.NONE){ ... }。

### Observable

public inline fun <T> observable(initialValue: T, crossinline onChange: (property: KProperty<\*>, oldValue: T,

newValue: T) -> Unit): ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue) {...}

这个委托会帮我们监测我们希望观察的属性的变化。当被观察属性的set方法被调用的时候，它就会自动执行我们指定的lambda表达式。所以一旦该属性被赋了新的值，我们就会接收到被委托的属性、旧值和新值。

class ViewModel(val db: MyDatabase) {

var myProperty by Delegates.observable("") {

d, old, new ->

db.saveChanges(this, new)

}

}

这个例子展示了，一些我们需要关心的ViewMode，每次值被修改了，就会保存它们到数据库。

### Vetoable

public inline fun <T> vetoable(initialValue: T, crossinline onChange: (property: KProperty<\*>, oldValue: T,

newValue: T) -> Boolean): ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue){...}

这是一个特殊的observable，它让你决定是否这个值需要被保存。它可以被用于在真正保存之前进行一些条件判断。

var positiveNumber = Delegates.vetoable(0) {

d, old, new ->

new >= 0

}

上面这个委托只允许在新的值是正数的时候执行保存。在lambda中，最后一行表示返回值。你不需要使用return关键字（实质上不能被编译）。

### Not Null

有时候我们需要在某些地方初始化这个属性，但是我们不能在构造函数中确定，或者我们不能在构造函数中做任何事情。第二种情况在Android中很常见：在Activity、fragment、service、receivers……无论如何，一个非抽象的属性在构造函数执行完之前需要被赋值。为了给这些属性赋值，我们无法让它一直等待到我们希望给它赋值的时候。我们至少有两种选择方案。

第一种就是使用可null类型并且赋值为null，直到我们有了真正想赋的值。但是我们就需要在每个地方不管是否是null都要去检查。如果我们确定这个属性在任何我们使用的时候都不会是null，这可能会使得我们要编写一些必要的代码了。

第二种选择是使用notNull委托。它会含有一个可null的变量并会在我们设置这个属性的时候分配一个真实的值。如果这个值在被获取之前没有被分配，它就会抛出一个异常。

public fun <T: Any> notNull(): ReadWriteProperty<Any?, T> = NotNullVar()

private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> {

private var value: T? = null

public override fun getValue(thisRef: Any?, property: KProperty<\*>): T {

return value ?: throw IllegalStateException("Property ${property.name} should be initialized before get.")

}

public override fun setValue(thisRef: Any?, property: KProperty<\*>, value: T) {

this.value = value

}

}

这个在单例App这个例子中很有用：

class App : Application() {

companion object {

var instance: App by Delegates.notNull()

}

override fun onCreate() {

super.onCreate()

instance = this

}

}

### 从Map中映射值

另外一种属性委托方式就是，属性的值会从一个map中获取value，属性的名字对应这个map中的key。如果我们import kotlin.properties.getValue，我们可以从构造函数的参数（一个不可修改的map）映射到val属性。如果我们想去修改map和属性，我们也可以import kotlin.properties.setValue。类需要一个MutableMap作为构造函数的参数。

想象我们从一个Json中加载了一个配置类，然后分配它们的key和value到一个map中。我们可以仅仅通过传入一个map的构造函数来创建一个实例：

import kotlin.properties.getValue

class Configuration(map: Map<String, Any?>) {

val width: Int by map

val height: Int by map

val dp: Int by map

val deviceName: String by map

}

作为一个参考，这里我展示下对于这个类怎么去创建一个必须要的map：

conf = Configuration(mapOf(

"width" to 1080,

"height" to 720,

"dp" to 240,

"deviceName" to "mydevice"

))

## 怎么去创建一个自定义的委托

举个例子，我们创建一个notNull的委托，它只能被赋值一次，如果第二次赋值，它就会抛异常。

Kotlin库提供了几个接口，我们自己的委托必须要实现：ReadOnlyProperty和ReadWriteProperty。具体取决于我们被委托的对象是val还是var。

我们要做的第一件事就是创建一个类然后继承ReadWriteProperty：

private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {

override fun getValue(thisRef: Any?, property: KProperty<\*>): T {

throw UnsupportedOperationException()

}

override fun setValue(thisRef: Any?, property: KProperty<\*>, value: T) {

}

}

这个委托可以作用在任何非null的类型。它接收任何类型的引用，然后像getter和setter那样使用T。现在我们需要去实现这些函数。

* Getter函数 如果已经被初始化，则会返回一个值，否则会抛异常。
* Setter函数 如果仍然是null，则赋值，否则会抛异常。

private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {

private var value: T? = null

override fun getValue(thisRef: Any?, property: KProperty<\*>): T {

return value ?: throw IllegalStateException("${property.name} " +

"not initialized")

}

override fun setValue(thisRef: Any?, property: KProperty<\*>, value: T) {

this.value = if (this.value == null) value

else throw IllegalStateException("${property.name} already initialized")

}

}

现在你可以创建一个对象，然后添加函数使用你的委托：

object DelegatesExt {

fun notNullSingleValue<T>(): ReadWriteProperty<Any?, T> = NotNullSingleValueVar()

}

## 重新实现Application单例化

在这个情景下，委托就可以帮助我们了。我们直到我们的单例不会是null，但是我们不能使用构造函数去初始化属性。所以我们可以使用notNull委托：

class App : Application() {

companion object {

var instance: App by Delegates.notNull()

}

override fun onCreate() {

super.onCreate()

instance = this

}

}

这种情况下有个问题，我们可以在app的任何地方去修改这个值，因为如果我们使用Delegates.notNull()，属性必须是var的。但是我们可以使用刚刚创建的委托，这样可以多一点保护。我们只能修改这个值一次：

companion object {

var instance: App by DelegatesExt.notNullSingleValue()

}

# 创建一个SQLiteOpenHelper

如你所知，Android使用SQLite作为它的数据库管理系统。SQLite是一个嵌入app的一个数据库，它的确是非常轻量的。这就是为什么这是手机app的不错的选择。

尽管如此，它的操作数据库的API在Android中是非常原生的。你将会需要编写很多SQL语句和你的对象与ContentValues或者Cursors之间的解析过程。很感激的，联合使用Kotlin和Anko，我们可以大量简化这些。

当然，有很多Android中可以使用的关于数据库的库，多亏Kotlin的互操作性，所有这些库都可以正常使用。但是针对一个简单的数据库来说可以不使用任何它们，之后的一分钟之内你就可以看到。

## ManagedSqliteOpenHelper

Anko提供了很多强大的SqliteOpenHelper来可以大量简化代码。当我们使用一个一般的SqliteOpenHelper，我们需要去调用getReadableDatabase()或者getWritableDatabase()，然后我们可以执行我们的搜索并拿到结果。在这之后，我们不能忘记调用close()。使用ManagedSqliteOpenHelper我们只需要：

forecastDbHelper.use {

...

}

在lambda里面，我们可以直接使用SqliteDatabase中的函数。它是怎么工作的？阅读Anko函数的实现方式真是一件有趣的事情，你可以从这里学到Kotlin的很多知识：

abstract class ManagedSQLiteOpenHelper(

ctx: Context,

name: String?,

factory: SQLiteDatabase.CursorFactory? = null,

version: Int = 1

): SQLiteOpenHelper(ctx, name, factory, version) {

...

private var db: SQLiteDatabase? = null

fun <T> use(f: SQLiteDatabase.() -> T): T {

try {

return openDatabase().f()

} finally {

closeDatabase()

}

}

}

首先，use接收一个SQLiteDatabase的扩展函数。这表示，我们可以使用this在大括号中，并且处于SQLiteDatabase对象中。这个函数扩展可以返回一个值，所以我们可以像这么做：

val result = forecastDbHelper.use {

val queriedObject = ...

queriedObject

}

要记住，在一个函数中，最后一行表示返回值。因为T没有任何的限制，所以我们可以返回任何对象。甚至如果我们不想返回任何值就使用Unit。

由于使用了try-finally，use方法会确保不管在数据库操作执行成功还是失败都会去关闭数据库。

而且，在sqliteDatabase中还有很多有用的扩展函数，我们会在之后使用到他们。但是现在让我们先定义我们的表和实现SqliteOpenHelper。

## 定义表

创建几个objects可以让我们避免表名列名拼写错误、重复等。我们需要两个表：一个用来保存城市的信息，另一个用来保存某天的天气预报。第二张表会有一个关联到第一张表的字段。

CityForecastTable提供了表的名字还有需要列：一个id（这个城市的zipCode），城市的名称和所在国家。

object CityForecastTable {

val NAME = "CityForecast"

val ID = "\_id"

val CITY = "city"

val COUNTRY = "country"

}

DayForecast有更多的信息，就如你下面看到的有很多的列。最后一列cityId，用来保持属于的城市id。

object DayForecastTable {

val NAME = "DayForecast"

val ID = "\_id"

val DATE = "date"

val DESCRIPTION = "description"

val HIGH = "high"

val LOW = "low"

val ICON\_URL = "iconUrl"

val CITY\_ID = "cityId"

}

## 实现SqliteOpenHelper

我们SqliteOpenHelper的基本组成是数据库的创建和更新，并提供了一个SqliteDatebase，使得我们可以用它来工作。查询可以被抽取出来放在其它的类中：

class ForecastDbHelper() : ManagedSQLiteOpenHelper(App.instance,

ForecastDbHelper.DB\_NAME, null, ForecastDbHelper.DB\_VERSION) {

...

}

我们在前面的章节中使用过我们创建的App.instance，这次我们同样的包括数据库名称和版本。这些值我们都会与SqliteOpenHelper一起定义在companion object中：

companion object {

val DB\_NAME = "forecast.db"

val DB\_VERSION = 1

val instance: ForecastDbHelper by lazy { ForecastDbHelper() }

}

instance这个属性使用了lazy委托，它表示直到它真的被调用才会被创建。用这种方法，如果数据库从来没有被使用，我们没有必要去创建这个对象。一般lazy委托的代码块可以阻止在多个不同的线程中创建多个对象。这个只会发生在两个线程在同事时间访问这个instance对象，它很难发生。但是发生具体还有看app的实现。无论如何，lazy委托是线程安全的。

为了去创建这些定义的表，我们需要去提供一个onCreate函数的实现。Anko提供了一个简单的扩展函数，它接收一个表的名字和一系列由列名和类型构建的Pair对象：

fun SQLiteDatabase.createTable(tableName:String, ifNotExists: Boolean=false, vararg columns:Pair<String, SqlType>) {

val escapedTableName = tableName.replace("`", "``")

val ifNotExistsText = if (ifNotExists) "IF NOT EXISTS" else ""

execSQL(

columns.map { col ->

"${col.first} ${col.second.render()}"

}.joinToString(", ", prefix = "CREATE TABLE $ifNotExistsText `$escapedTableName`(", postfix = ");")

)

}

db.createTable(CityForecastTable.NAME, true,

Pair(CityForecastTable.ID, INTEGER + PRIMARY\_KEY),

Pair(CityForecastTable.CITY, TEXT),

Pair(CityForecastTable.COUNTRY, TEXT))

* 第一个参数是表的名称
* 第二个参数，当是true的时候，创建之前会检查这个表是否存在。
* 第三个参数是一个Pair类型的vararg参数。vararg也存在在Java中，这是一种在一个函数中传入联系很多相同类型的参数。这个函数也接收一个对象数组。

Anko中有一种叫做SqlType的特殊类型，它可以与SqlTypeModifiers混合，比如PRIMARY\_KEY。+操作符像之前那样被重写了。这个plus函数会把两者通过合适的方式结合起来，然后返回一个新的SqlType：

fun SqlType.plus(m: SqlTypeModifier) : SqlType {

return SqlTypeImpl(name, if (modifier == null) m.toString()

else "$modifier $m")

}

如你所见，她会把多个修饰符组合起来。

但是回到我们的代码，我们可以修改得更好。Kotlin标准库中包含了一个叫to的函数，又一次，让我们来展示Kotlin的强大之处。它作为第一参数的扩展函数，接收另外一个对象作为参数，把两者组装并返回一个Pair。

public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)

因为带有函数to被中缀修饰了，所以结果非常清晰：

val pair = object1 to object2

然后，把他们应用到表的创建中，这就是整个函数看起来的样子：

override fun onCreate(db: SQLiteDatabase) {

db.createTable(CityForecastTable.NAME, true,

CityForecastTable.ID to INTEGER + PRIMARY\_KEY,

CityForecastTable.CITY to TEXT,

CityForecastTable.COUNTRY to TEXT)

db.createTable(DayForecastTable.NAME, true,

DayForecastTable.ID to INTEGER + PRIMARY\_KEY + AUTOINCREMENT,

DayForecastTable.DATE to INTEGER,

DayForecastTable.DESCRIPTION to TEXT,

DayForecastTable.HIGH to INTEGER,

DayForecastTable.LOW to INTEGER,

DayForecastTable.ICON\_URL to TEXT,

DayForecastTable.CITY\_ID to INTEGER)

}

我们有一个相似的函数用于删除表。onUpgrade将只是删除表，然后重建它们。我们只是把我们数据库作为一个缓存，所以这是一个简单安全的方法保证我们的表会如我们所期望的那样被重建。如果我有很重要的数据需要保留，我们就需要优化onUpgrade的代码，让它根据数据库版本来做相应的数据转移。

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {

db.dropTable(CityForecastTable.NAME, true)

db.dropTable(DayForecastTable.NAME, true)

onCreate(db)

}

## 依赖注入

Dagger可能是Android上最流行的依赖注入器。当然当我们提供依赖有一定复杂性的时候是个很好的替代品。但是最小的替代是可以在这个构造函数中使用默认值。我们可以给构造函数的参数通过分配默认值的方式提供一个依赖，然后在不同的情况下提供不同的实例。比如，在我们的ForecastDbHelper我们可以用更智能的方式提供一个context：

class ForecastDbHelper(ctx: Context = App.instance) :

ManagedSQLiteOpenHelper(ctx, ForecastDbHelper.DB\_NAME, null,

ForecastDbHelper.DB\_VERSION) {

...

}

现在我们有两种方式来创建这个类：

val dbHelper1 = ForecastDbHelper() // 它会使用 App.instance

val dbHelper2 = ForecastDbHelper(mockedContext) // 比如，提供给测试tests

# 集合和函数操作符

## 总数操作符

any

如果至少有一个元素符合给出的判断条件，则返回true。

val list = listOf(1, 2, 3, 4, 5, 6)

assertTrue(list.any { it % 2 == 0 })

assertFalse(list.any { it > 10 })

all

如果全部的元素符合给出的判断条件，则返回true。

assertTrue(list.all { it < 10 })

assertFalse(list.all { it % 2 == 0 })

count

返回符合给出判断条件的元素总数。

assertEquals(3, list.count { it % 2 == 0 })

fold

在一个初始值的基础上从第一项到最后一项通过一个函数累计所有的元素。

assertEquals(25, list.fold(4) { total, next -> total + next })

foldRight

与fold一样，但是顺序是从最后一项到第一项。

assertEquals(25, list.foldRight(4) { total, next -> total + next })

forEach

遍历所有元素，并执行给定的操作。

list.forEach { println(it) }

forEachIndexed

与forEach，但是我们同时可以得到元素的index。

list.forEachIndexed { index, value

-> println("position $index contains a $value") }

max

返回最大的一项，如果没有则返回null。

assertEquals(6, list.max())

maxBy

根据给定的函数返回最大的一项，如果没有则返回null。

// The element whose negative is greater

assertEquals(1, list.maxBy { -it })

min

返回最小的一项，如果没有则返回null。

assertEquals(1, list.min())

minBy

根据给定的函数返回最小的一项，如果没有则返回null。

// The element whose negative is smaller

assertEquals(6, list.minBy { -it })

none

如果没有任何元素与给定的函数匹配，则返回true。

// No elements are divisible by 7

assertTrue(list.none { it % 7 == 0 })

reduce

与fold一样，但是没有一个初始值。通过一个函数从第一项到最后一项进行累计。

assertEquals(21, list.reduce { total, next -> total + next })

reduceRight

与reduce一样，但是顺序是从最后一项到第一项。

assertEquals(21, list.reduceRight { total, next -> total + next })

sumBy

返回所有每一项通过函数转换之后的数据的总和。

assertEquals(3, list.sumBy { it % 2 })

## 过滤操作符

drop

返回包含去掉前n个元素的所有元素的列表。

assertEquals(listOf(5, 6), list.drop(4))

dropWhile

返回根据给定函数从第一项开始去掉指定元素的列表。

assertEquals(listOf(3, 4, 5, 6), list.dropWhile { it < 3 })

dropLastWhile

返回根据给定函数从最后一项开始去掉指定元素的列表。

assertEquals(listOf(1, 2, 3, 4), list.dropLastWhile { it > 4 })

filter

过滤所有符合给定函数条件的元素。

assertEquals(listOf(2, 4, 6), list.filter { it % 2 == 0 })

filterNot

过滤所有不符合给定函数条件的元素。

assertEquals(listOf(1, 3, 5), list.filterNot { it % 2 == 0 })

filterNotNull

过滤所有元素中不是null的元素。

assertEquals(listOf(1, 2, 3, 4), listWithNull.filterNotNull())

slice

过滤一个list中指定index的元素。

assertEquals(listOf(2, 4, 5), list.slice(listOf(1, 3, 4)))

take

返回从第一个开始的n个元素。

assertEquals(listOf(1, 2), list.take(2))

takeLast

返回从最后一个开始的n个元素

assertEquals(listOf(5, 6), list.takeLast(2))

takeWhile

返回从第一个开始符合给定函数条件的元素。

assertEquals(listOf(1, 2), list.takeWhile { it < 3 })

## 映射操作符

flatMap

遍历所有的元素，为每一个创建一个集合，最后把所有的集合合并放在一个集合中。

assertEquals(listOf(1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7),

list.flatMap { listOf(it, it + 1) })

groupBy

返回一个根据给定函数分组后的map。

assertEquals(mapOf("odd" to listOf(1, 3, 5), "even" to listOf(2, 4, 6)), list.groupBy { if (it % 2 == 0) "even" else "odd" })

map

返回一个每一个元素根据给定的函数转换所组成的List。

assertEquals(listOf(2, 4, 6, 8, 10, 12), list.map { it \* 2 })

mapIndexed

返回一个每一个元素根据给定的包含元素index的函数转换所组成的List。

assertEquals(listOf (0, 2, 6, 12, 20, 30), list.mapIndexed { index, it -> index \* it })

mapNotNull

返回一个每一个非null元素根据给定的函数转换所组成的List。

assertEquals(listOf(2, 4, 6, 8), listWithNull.mapNotNull { it \* 2 })

## 元素操作符

contains

如果指定元素可以在集合中找到，则返回true。

assertTrue(list.contains(2))

elementAt

返回给定index对应的元素，如果index数组越界则会抛出IndexOutOfBoundsException。

assertEquals(2, list.elementAt(1))

elementAtOrElse

返回给定index对应的元素，如果index数组越界则会根据给定函数返回默认值。

assertEquals(20, list.elementAtOrElse(10, { 2 \* it }))

elementAtOrNull

返回给定index对应的元素，如果index数组越界则会返回null。

assertNull(list.elementAtOrNull(10))

first

返回符合给定函数条件的第一个元素。

assertEquals(2, list.first { it % 2 == 0 })

firstOrNull

返回符合给定函数条件的第一个元素，如果没有符合则返回null。

assertNull(list.firstOrNull { it % 7 == 0 })

indexOf

返回指定元素的第一个index，如果不存在，则返回-1。

assertEquals(3, list.indexOf(4))

indexOfFirst

返回第一个符合给定函数条件的元素的index，如果没有符合则返回-1。

assertEquals(1, list.indexOfFirst { it % 2 == 0 })

indexOfLast

返回最后一个符合给定函数条件的元素的index，如果没有符合则返回-1。

assertEquals(5, list.indexOfLast { it % 2 == 0 })

last

返回符合给定函数条件的最后一个元素。

assertEquals(6, list.last { it % 2 == 0 })

lastIndexOf

返回指定元素的最后一个index，如果不存在，则返回-1。

lastOrNull

返回符合给定函数条件的最后一个元素，如果没有符合则返回null。

val list = listOf(1, 2, 3, 4, 5, 6)

assertNull(list.lastOrNull { it % 7 == 0 })

single

返回符合给定函数的单个元素，如果没有符合或者超过一个，则抛出异常。

assertEquals(5, list.single { it % 5 == 0 })

singleOrNull

返回符合给定函数的单个元素，如果没有符合或者超过一个，则返回null。

assertNull(list.singleOrNull { it % 7 == 0 })

## 生产操作符

merge

把两个集合合并成一个新的，相同index的元素通过给定的函数进行合并成新的元素作为新的集合的一个元素，返回这个新的集合。新的集合的大小由最小的那个集合大小决定。

val list = listOf(1, 2, 3, 4, 5, 6)

val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)

assertEquals(listOf(3, 4, 6, 8, 10, 11), list.merge(listRepeated) { it1, it2 -> it1 + it2 })

partition

把一个给定的集合分割成两个，第一个集合是由原集合每一项元素匹配给定函数条件返回true的元素组成，第二个集合是由原集合每一项元素匹配给定函数条件返回false的元素组成。

assertEquals(

Pair(listOf(2, 4, 6), listOf(1, 3, 5)),

list.partition { it % 2 == 0 }

)

plus

返回一个包含原集合和给定集合中所有元素的集合，因为函数的名字原因，我们可以使用+操作符。

assertEquals(

listOf(1, 2, 3, 4, 5, 6, 7, 8),

list + listOf(7, 8)

)

zip

返回由pair组成的List，每个pair由两个集合中相同index的元素组成。这个返回的List的大小由最小的那个集合决定。

assertEquals(

listOf(Pair(1, 7), Pair(2, 8)),

list.zip(listOf(7, 8))

)

unzip

从包含pair的List中生成包含List的Pair。

assertEquals(

Pair(listOf(5, 6), listOf(7, 8)),

listOf(Pair(5, 7), Pair(6, 8)).unzip()

)

## 顺序操作符

reverse

返回一个与指定list相反顺序的list。

val unsortedList = listOf(3, 2, 7, 5)

assertEquals(listOf(5, 7, 2, 3), unsortedList.reverse())

sort

返回一个自然排序后的list。

assertEquals(listOf(2, 3, 5, 7), unsortedList.sort())

sortBy

返回一个根据指定函数排序后的list。

assertEquals(listOf(3, 7, 2, 5), unsortedList.sortBy { it % 3 })

sortDescending

返回一个降序排序后的List。

assertEquals(listOf(7, 5, 3, 2), unsortedList.sortDescending())

sortDescendingBy

返回一个根据指定函数降序排序后的list。

assertEquals(listOf(2, 5, 7, 3), unsortedList.sortDescendingBy { it % 3 })

# 从数据库中保存或查询数据

前面一个章节中我们讲了关于SQLiteOpenHelper的创建，但是我们需要在必要的时候有方法去保存我们的数据到数据库，或者从我们的数据库中查询数据。另外一个叫ForecastDb类就会做这件事。

## 创建数据库model类

首先，我们要去为数据库创建model类。你还记得我们之前所见的map委托的方式？我们要把这些属性直接映射到数据库中，反过来也一样。

我们先来看下CityForecast类：

class CityForecast(val map: MutableMap<String, Any?>, val dailyForecast: List<DayForecast>) {

var \_id: Long by map

var city: String by map

var country: String by map

constructor(id: Long, city: String, country: String, dailyForecast: List<DayForecast>)

: this(HashMap(), dailyForecast) {

this.\_id = id

this.city = city

this.country = country

}

}

默认的构造函数会得到一个含有属性和对应的值的map，和一个dailyForecast。多亏了委托，这些值会根据key的名字会映射到相应的属性中去。如果我们希望映射的过程运行完美，那么属性的名字必须要和数据库中对应的名字一模一样

但是，第二个构造函数也是必要的。这是因为我们需要从domain映射到数据库类中，所以不能使用map，从属性中设置值也是方便的。我们传入一个空的map，但是又一次，多亏了委托，当我们设置值到属性的时候，它会自动增加所有的值到map中。用这种方式，我们就准备好map来保存到数据库中了。使用了这些有用的代码，我将会看见它运行起来就像魔法一样神奇。

现在我们需要第二个类，DayForecast，它会是第二个表。它包括表中的每一列作为它的属性，它也有第二个构造函数。唯一不同之处就是不需要设置id，因为它将通过SQLite自增长。

class DayForecast(var map: MutableMap<String, Any?>) {

var \_id: Long by map

var date: Long by map

var description: String by map

var high: Int by map

var low: Int by map

var iconUrl: String by map

var cityId: Long by map

constructor(date: Long, description: String, high: Int, low: Int,

iconUrl: String, cityId: Long)

: this(HashMap()) {

this.date = date

this.description = description

this.high = high

this.low = low

this.iconUrl = iconUrl

this.cityId = cityId

}

}

这些类将会帮助我们SQLite表与对象之间的互相映射。

## 写入和查询数据库

### 创建ForecastDb

SqliteOpenHelper只是一个工具，是SQL世界和OOP之间的一个通道。我们要新建几个类来请求已经保存在数据库中的数据，和保存新的数据。被定义的类会使用ForecastDbHelper和DataMapper来转换数据库中的数据到domain models。我仍旧使用默认值的方式来实现简单的依赖注入：

class ForecastDb(

val forecastDbHelper: ForecastDbHelper = ForecastDbHelper.instance,

val dataMapper: DbDataMapper = DbDataMapper()) {

...

}

所有的函数使用前面章节讲到过的use()函数。lambda返回的值也会被作为这个函数的返回值。所以让我们定义一个使用zip code和date来查询一个forecast的函数：

fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {

...

}

这么没有什么解释的：我们使用use函数返回的结果作为这个函数返回的结果。

### 查询一个forecast

第一个要做的查询就是每日的天气预报，因为我们需要这个列表来创建一个city对象。Anko提供了一个简单的请求构建器，所以我们来利用下这个有利条件：

val dailyRequest = "${DayForecastTable.CITY\_ID} = ? " + "AND ${DayForecastTable.DATE} >= ?"

val dailyForecast = select(DayForecastTable.NAME)

.whereSimple(dailyRequest, zipCode.toString(), date.toString())

.parseList { DayForecast(HashMap(it)) }

第一行，dailyRequest是查询语句中where的一部分。它是whereSimple函数需要的第一个参数，这与我们用一般的helper做的方式很相似。这里有另外一个简化的where函数，它需要一些tags和values来进行匹配。我不太喜欢这个方式，因为我觉得这个增加了代码的模版化，虽然这个对我们把values解析成String很有利。最后它看起来会是这样：

val dailyRequest = "${DayForecastTable.CITY\_ID} = {id}" + "AND ${DayForecastTable.DATE} >= {date}"

val dailyForecast = select(DayForecastTable.NAME)

.where(dailyRequest, "id" to zipCode, "date" to date)

.parseList { DayForecast(HashMap(it)) }

你可以选择你喜欢的一种方式。select函数是很简单的，它仅仅是需要一个被查询表的名字。parse函数的时候会有一些魔法在里面。在这个例子中我们假设请求结果是一个list，使用了parseList函数。它使用了RowParser或MapRowParser函数去把cursor转换成一个对象的集合。这两个不同之处就是RowParser是依赖列的顺序的，而MapRowParser是从map中拿到作为column的key名的。

在它们之间有两个重载的冲突，所以我们不能直接使用简化的方式准确地创建需要的对象。但是没有什么是不能通过扩展函数来解决的。我创建了一个接收一个lambda函数返回一个MapRowParser的函数。解析器会调用这个lambda来创建这个对象：

fun <T : Any> SelectQueryBuilder.parseList(parser: (Map<String, Any>) -> T): List<T> =

parseList(object : MapRowParser<T> {

override fun parseRow(columns: Map<String, Any>): T = parser(columns)

})

这个函数可以帮助我们简单地去parseList查询的结果：

parseList { DayForecast(HashMap(it)) }

解析器接收的immutable map被我们转化成了一个mutable map（我们需要在database model中是可以修改的）通过使用相应的HashMap构造函数。在DayForecast中的构造函数中会使用到这个HashMap。

所以，这个查询返回了一个Cursor，要理解这个场景的背后到底发生了什么。parseList中会迭代它，然后得到Cursor的每一行直到最后一个。对于每一行，它会创建一个包含这列的key和给对应的key赋值后的map。然后把这个map返回给这个解析器。如果查询没有任何结果，parseList会返回一个空的list。

下一步查询城市也是一样的方法：

val city = select(CityForecastTable.NAME)

.whereSimple("${CityForecastTable.ID} = ?", zipCode.toString())

.parseOpt { CityForecast(HashMap(it), dailyForecast) }

不同之处是：我们使用的是parseOpt。这个函数返回一个可null的对象。结果可以使一个null或者单个的对象，这取决于请求是否能在数据库中查询到数据。这里有另外一个叫parseSingle的函数，本质上是一样的，但是它返回的事一个不可null的对象。所以如果没有在数据库中找到这一条数据，它会抛出一个异常。在我们的例子中，第一次查询一个城市的时候，肯定是不存在的，所以使用parseOpt会更安全。我又创建了一个好用的函数来阻止我们需要的对象的创建：

public fun <T : Any> SelectQueryBuilder.parseOpt(

parser: (Map<String, Any>) -> T): T? =

parseOpt(object : MapRowParser<T> {

override fun parseRow(columns: Map<String, Any>): T = parser(columns)

})

最后如果返回的city不是null，我们使用dataMapper把它转换成domain object再返回它。否则，我们直接返回null。你应该记得，lambda的最后一行表示返回值。所以这里将会返回一个CityForecast?类型的对象：

if (city != null) dataMapper.convertToDomain(city) else null

DataMapper函数很简单：

fun convertToDomain(forecast: CityForecast) = with(forecast) {

val daily = dailyForecast.map { convertDayToDomain(it) }

ForecastList(\_id, city, country, daily)

}

private fun convertDayToDomain(dayForecast: DayForecast) = with(dayForecast) {

Forecast(date, description, high, low, iconUrl)

}

最后完整的函数如下：

fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {

val dailyRequest = "${DayForecastTable.CITY\_ID} = ? AND " + "${DayForecastTable.DATE} >= ?"

val dailyForecast = select(DayForecastTable.NAME)

.whereSimple(dailyRequest, zipCode.toString(), date.toString())

.parseList { DayForecast(HashMap(it)) }

val city = select(CityForecastTable.NAME)

.whereSimple("${CityForecastTable.ID} = ?", zipCode.toString())

.parseOpt { CityForecast(HashMap(it), dailyForecast) }

if (city != null) dataMapper.convertToDomain(city) else null

}

另外一个Anko中好玩的功能我们在这里展示，那就是你可以使用classParser()来替代我们用的MapRowParser，它是基于列名通过反射的方式去生成对象的。我喜欢另一种方法因为我不需要使用反射并且还有控制权进行转换操作，但是在有时候可能对你有用。

### 保存一个forecast

saveForecast函数只是从数据库中清除数据，然后转换domain model为数据库model，然后插入每一天的forecast和city forecast。这个结构比之前的更简单：它通过use函数从database helper中返回数据。在这个例子中我们不需要返回值，所以它将返回Unit。

fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

...

}

首先，我们清空这两个表。Anko没有提供比较漂亮的方式来做这个，但这并不意味着我们不行。所以我们创建了一个SQLiteDatabase的扩展函数来让我们可以像SQL查询一样来执行它：

fun SQLiteDatabase.clear(tableName: String) {

execSQL("delete from $tableName")

}

清空这两个表：

clear(CityForecastTable.NAME)

clear(DayForecastTable.NAME)

现在，是时候去转换执行insert后返回的数据了。

with(dataMapper.convertFromDomain(forecast)) {

...

}

从domain model转换的方式也是很直接的：

fun convertFromDomain(forecast: ForecastList) = with(forecast) {

val daily = dailyForecast.map { convertDayFromDomain(id, it) }

CityForecast(id, city, country, daily)

}

private fun convertDayFromDomain(cityId: Long, forecast: Forecast) =

with(forecast) {

DayForecast(date, description, high, low, iconUrl, cityId)

}

在代码块，我们可以在不使用引用和变量的情况下使用dailyForecast和map，只是像我们在这个类内部一样就可以了。

针对插入我们使用另外一个Anko函数，它需要一个表名和一个vararg修饰的Pair<String, Any>作为参数。这个函数会把vararg转换成Android SDK需要的ContentValues对象。所以我们的任务组成是把map转换成一个vararg数组。我们为MutableMap创建了一个扩展函数：

public inline fun <K, V, R> Map<out K, V>.map(transform: (Map.Entry<K, V>) -> R): List<R> {

return mapTo(ArrayList<R>(size), transform)

}

@Suppress("UNCHECKED\_CAST")

public inline fun <reified T> Collection<T>.toTypedArray(): Array<T> {

@Suppress("PLATFORM\_CLASS\_MAPPED\_TO\_KOTLIN")

val thisCollection = this as java.util.Collection<T>

return thisCollection.toArray(arrayOfNulls<T>(0)) as Array<T>

}

fun <K, V : Any> MutableMap<K, V?>.toVarargArray(): Array<out Pair<K, V>> =

map({ Pair(it.key, it.value!!) }).toTypedArray()

它是支持可null的值的（这是map delegate的条件），把它转换为非null值（select函数需要）的Array所组成的Pairs。不用担心就算你不完全理解这个函数，我很快就会讲到可空性。

所以，这个新的函数我们可以这么使用：

insert(CityForecastTable.NAME, \*map.toVarargArray())

它在CityForecast中插入了一个一行新的数据。在toVarargArray函数结果前面使用\*表示这个array会被分解成为一个vararg参数。这个在Java中是自动处理的，但是我们需要在Kotlin中明确指明。

每天的天气预报也是一样了：

dailyForecast.forEach { insert(DayForecastTable.NAME, \*it.map.toVarargArray()) }

所以，通过map的使用，我们可以用很简单的方式把类转换为数据表，反之亦然。因为我们已经新建了扩展函数，我们可以在别的项目中使用，这个才是真正可贵的地方。

这个函数的完整代码如下：

fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

clear(CityForecastTable.NAME)

clear(DayForecastTable.NAME)

with(dataMapper.convertFromDomain(forecast)) {

insert(CityForecastTable.NAME, \*map.toVarargArray())

dailyForecast forEach {

insert(DayForecastTable.NAME, \*it.map.toVarargArray())

}

}

}

在这一章中有很多代码被需要，所以你可以到代码库中查看检出。

# Kotlin中的null安全

## 可null类型怎么工作

指定一个变量是可null是通过在类型的最后增加一个问号。因为在Kotlin中一切都是对象（甚至是Java中原始数据类型），一切都是可null的。所以，当然我们可以有一个可null的integer：

val a: Int? = null

一个可null类型，你在没有进行检查之前你是不能直接使用它。这个代码不能被编译：

val a: Int? = null

a.toString()

前一行代码标记为可null，然后编译器就会知道它，所以在你null检查之前你不能去使用它。还有一个特性是当我们检查了一个对象的可null性，之后这个对象就会自动转型成不可null类型，这就是Kotlin编译器的智能转换：

vala:Int?=null

...

if(a!=null){

a.toString()

}

在if中，a从Int?变成了Int，所以我们可以不需要再检查可null性而直接使用它。if代码之外，当然我们又得检查处理。这仅仅在变量当前不能被改变的时候才有效，因为否则这个value可能被另外的线程修改，这时前面的检查会返回false。val属性或者本地（val or var）变量。

这听起来会让事情变得更多。难道我们不得不去编写大量代码去进行可null性的检查？当然不是，首先，因为大多数时候你不需要使用null类型。Null引用没有我们想象中的有用，当你想弄清楚一个变量是否可以为null时你就会发现这一点。但是Kotlin也有它自己的使处理更简洁的方案。举个例子，我们如下简化代码：

val a: Int? = null

...

a?.toString()

这里我们使用了安全访问操作符(?)。只有这个变量不是null的时候才会去执行前面的那行代码。否则，它不会做任何事情。并且我们甚至可以使用Elvis operator(?:)：

val a:Int? = null

val myString = a?.toString() ?: ""

因为在Kotlin中throw和return都是表达式，他们可以用在Elvis operator操作符的右边：

val myString = a?.toString() ?: return false

val myString = a?.toString() ?: throw IllegalStateException()

然后，我们可能会遇到这种情景，我们确定我们是在用一个非null变量，但是他的类型却是可null的。我们可以使用!!操作符来强制编译器执行可null类型时跳过限制检查：

val a: Int? = null

a!!.toString()

上面的代码将会被编译，但是很显然会奔溃。所以我们要确保只能在特定的情况下使用。通常我们可以自己选择作为解决方案。如果一份代码满篇都是!!，那就有股代码没有被正确处理的气味了。

# 创建业务逻辑来访问数据

在实现访问服务器和与本地数据库交互之后，是时候把事情整合起来了。逻辑步骤如下：

* 从数据库获取数据
* 检查是否存在对应星期的数据
* 如果有，返回UI并且渲染
* 如果没有，请求服务器获取数据
* 结果被保存在数据库中并且返回UI渲染

但是我们的commands不应该去处理所有这些逻辑。数据源应该是一个具体的实现，这样就可以被容易地修改，所以增加一些额外的代码，然后把command从数据访问中抽象出来听起来是个不错的方式。在我们的实现中，它会遍历整个list直到结果被找到。

所以我们先来给接口定义一些我们实现provider需要使用到的数据源：

interface ForecastDataSource {

fun requestForecastByZipCode(zipCode: Long, date: Long): ForecastList?

}

provider需要一个接收zip code和一个date，然后它应该根据那一天返回一周的天气预报。

class ForecastProvider(val sources: List<ForecastDataSource> = ForecastProvider.SOURCES) {

companion object {

val DAY\_IN\_MILLIS = 1000 \* 60 \* 60 \* 24

val SOURCES = listOf(ForecastDb(), ForecastServer())

}

...

}

forecast provider接收一个数据源列表，通过构造函数传入（比如用于测试），但是我设置了source的默认值为被定义在companion object中的SOURCES List。我将使用数据库的数据源和服务端数据源。顺序是很重要的，因为它会根据顺序去遍历这个sources，然后一旦获取到有效的返回值就会停止查询。逻辑顺序是先在本地查询（本地数据库中），然后再通过API查询。

所以主函数的代码如下：

fun requestByZipCode(zipCode: Long, days: Int): ForecastList

= sources.firstResult { requestSource(it, days, zipCode) }

它会得到第一个不是null的结果然后返回。当我在第18章中讲到的大量的函数操作符中搜索后，我没有找到完全符合我想要的。所以当我去查看Kotlin的源码时，我直接拷贝了first函数然后修改它们来达到我想要的目的：

inline fun <T, R : Any> Iterable<T>.firstResult(predicate: (T) -> R?) : R {

for (element in this){

val result = predicate(element)

if (result != null) return result

}

throw NoSuchElementException("No element matching predicate was found.")

}

该函数接收一个断言函数，它接收一个T类型的对象然后返回一个R?类型的值。这表示predicate可以返回null类型，但是我们的firstResult不能返回null。这就是为什么返回R的原因。

它怎么工作呢？它将遍历集合中的每一个元素然后执行这个断言函数。当这个断言函数的结果返回不是null时，这个结果就会被返回。

如果我们可以允许sources返回null，那我们就可以使用firstOrNull函数来代替。不同之处就是最后一行的返回null和抛异常。但是我现在不在代码里面去处理这些细节了。

在我们的例子中T = ForecastDataSource，R = ForecastList。但是记住在ForecastDataSource中指定的函数返回一个ForecastList?，也就是R?，所以一切都是匹配得这么完美。requestSource让前面的函数看起来更有可读性：

fun requestSource(source: ForecastDataSource, days: Int, zipCode: Long): ForecastList? {

val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())

return if (res != null && res.size() >= days) res else null

}

如果结果不是null并且数量也参数匹配，那这个查询被执行且只会返回一个数据。否则，数据源没有足够的数据来返回一个成功的结果。

函数todayTimeSpan()计算今天毫秒级的时间，并排除掉“时差”。其中一些数据源（我们例子中的数据库）可能会需要它。因为如果我们没有指定更多的信息，服务端默认就是今天，所以我们不需要设置它。

private fun todayTimeSpan() = System.currentTimeMillis() / DAY\_IN\_MILLIS \* DAY\_IN\_MILLIS

这个类完整的代码如下：

class ForecastProvider(val sources: List<ForecastDataSource> = ForecastProvider.SOURCES) {

companion object {

val DAY\_IN\_MILLIS = 1000 \* 60 \* 60 \* 24;

val SOURCES = listOf(ForecastDb(), ForecastServer())

}

fun requestByZipCode(zipCode: Long, days: Int): ForecastList

= sources.firstResult { requestSource(it, days, zipCode) }

private fun requestSource(source: RepositorySource, days: Int,

zipCode: Long): ForecastList? {

val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())

return if (res != null && res.size() >= days) res else null

}

private fun todayTimeSpan() = System.currentTimeMillis() /

DAY\_IN\_MILLIS \* DAY\_IN\_MILLIS

}

我们已经定义了一个ForecastDb。现在我们需要去实现ForcastDataSource：

class ForecastDb(val forecastDbHelper: ForecastDbHelper =

ForecastDbHelper.instance, val dataMapper: DbDataMapper = DbDataMapper())

: ForecastDataSource {

override fun requestForecastByZipCode(zipCode: Long, date: Long) =

forecastDbHelper.use {

...

}

...

}

ForecastServer还没有还被实现，但是这是非常简单的。它在从服务端接收到数据之后就会使用ForecastDb去保存到数据库。用这种方式，我们就可以缓存这些数据到数据库中，提供给以后的查询。

class ForecastServer(val dataMapper: ServerDataMapper = ServerDataMapper(),

val forecastDb: ForecastDb = ForecastDb()) : ForecastDataSource {

override fun requestForecastByZipCode(zipCode: Long, date: Long):

ForecastList? {

val result = ForecastByZipCodeRequest(zipCode).execute()

val converted = dataMapper.convertToDomain(zipCode, result)

forecastDb.saveForecast(converted)

return forecastDb.requestForecastByZipCode(zipCode, date)

}

}

它也是使用了之前我们创建的data mapper，最然我们修改一些函数的名字来让它更加与我们之前用在database model的mapper更相似。你可以查看provider来查看细节。

被重写的方法用来请求服务器，转换结果到domain objects并保存它们到数据库。它最后查询数据库返回数据，这是因为我们需要使用到插入到数据库中的字增长id。

这就是provider被实现的最后的一步了。现在我们需要开始使用它。ForecastCommand不会再直接与服务端交互，也不会转换数据到domain model。

RequestForecastCommand(val zipCode: Long,

val forecastProvider: ForecastProvider = ForecastProvider()) :

Command<ForecastList> {

companion object {

val DAYS = 7

}

override fun execute(): ForecastList {

return forecastProvider.requestByZipCode(zipCode, DAYS)

}

}

# Flow control 和ranges

# 创建一个详情界面

## 获取color

fun dealTv(t: Temperature) = with(tv) {

text = "${t.toString()}"

textColor = color(when (t.num) {

in -50..0 -> android.R.color.holo\_red\_dark

in 0..15 -> android.R.color.holo\_orange\_dark

else -> android.R.color.holo\_green\_dark

})

}

}

color是Anko中的一个扩展函数，它可以很简洁的方式从resources中获取一个color，类似于我们在其它地方使用到的dimen。由于没有导入这个包，手动写该扩展，当前support library依赖ContextCompat来从不同的Android版本中获取一个color：

public fun Context.color(res: Int): Int = ContextCompat.getColor(this, res)

## 启动一个activity：reified函数

启动一个activity

val intent = Intent(MainActivity@this, javaClass<DetailActivity>())

intent.putExtra(DetailActivity.ID, id)

intent.putExtra(DetailActivity.CITY\_NAME, city)

startActivity(intent)

但是这是非常冗长的。一如既往地，Anko提供了简单得多的方式通过reified function来启动一个activity：

startActivity<DetailActivity>(DetailActivity.ID to it.id, DetailActivity.CITY\_NAME to result.city)

reified function背后到底有什么魔法呢？就像你可能知道的那样，当我们在Java中创建一个范型函数，我们没有办法得到范型类型的Class。一个流行的变通方法是作为参数传入一个Class。在Kotlin中，一个内联（inline）函数可以被具体化（reified），这意味着我们可以在函数中得到并使用范型类型的Class。Anko真正使用它的一个简单的例子接下来会讲到（在这个例子中我只使用了String）：

public inline fun <reified T: Activity> Context.startActivity(vararg params: Pair<String, String>) {

val intent = Intent(this, T::class.javaClass)

params forEach { intent.putExtra(it.first, it.second) }

startActivity(intent)

}

真正的实现要更加复杂一点因为它使用了一个很长的令人讨厌的when表达式来增加由类型决定的额外信息，但是在概念上来说它没有增加其它更有用的知识。

Reified函数是有一个可以简化代码和提高理解性的语法糖。在这个例子中，它通过获取到了范型类型的javaClass来创建了一个intent，迭代所有参数并增加到intent，然后使用Intent来启动activity。reified限制于activity的子类。

# 接口和委托

## 接口

Kotlin中的接口比Java 7中要强大得多。如果你使用Java 8，它们非常相似。在Kotlin中，我们可以像Java中那样使用接口。

想象我们有一些动物，它们的其中一些可以飞行。这个是我们针对飞行动物的接口：

interface FlyingAnimal {

fun fly()

}

鸟和蝙蝠都可以通过扇动翅膀的方式飞行。所以我们为它们创建两个类：

class Bird : FlyingAnimal {

val wings: Wings = Wings()

override fun fly() = wings.move()

}

class Bat : FlyingAnimal {

val wings: Wings = Wings()

override fun fly() = wings.move()

}

当两个类继承自一个接口，非常典型的是它们两者共享相同的实现。但是Java 7中的接口只能定义行为，但是不能去实现它。

Kotlin接口在某一方面它可以实现函数。它们与类唯一的不同之处是它们是无状态（stateless）的，所以属性需要子类去重写。类需要去负责保存接口属性的状态。

我们可以让接口实现fly函数：

interface FlyingAnimal {

val wings: Wings

fun fly() = wings.move()

}

就像提到的那样，类需要去重写属性：

class Bird : FlyingAnimal {

override val wings: Wings = Wings()

}

class Bat : FlyingAnimal {

override val wings: Wings = Wings()

}

现在鸟和蝙蝠都可以飞行了：

val bird = Bird()

val bat = Bat()

bird.fly()

bat.fly()

## 委托

委托模式是一个很有用的模式，它可以用来从类中抽取出主要负责的部分。委托模式是Kotlin原生支持的，所以它避免了我们需要去调用委托对象。委托者只需要指定实现的接口的实例。

在我们前面的例子中，我们可以通过构造函数指定动物怎么飞行，而不是实现它。比如，一个使用翅膀飞行的动物可以用这种方式指定：

interface CanFly {

fun fly()

}

class Bird(f: CanFly) : CanFly by f

我们可以使用接口来指示鸟可以飞行，但是鸟的飞行方式被定义在一个委托中，这个委托定义在构造函数中，所以我们可以针对不同的鸟使用不同的飞行方式。动物使用翅膀飞行的方式被定义在另一个类中：

class AnimalWithWings : CanFly {

val wings: Wings = Wings()

override fun fly() = wings.move()

}

动物扇动翅膀来飞行。所以我们可以创建一个鸟，它使用翅膀飞行：

val birdWithWings = Bird(AnimalWithWings())

birdWithWings.fly()

但是现在翅膀可以被别的不是鸟类的动物使用。如果我们假设蝙蝠使用翅膀，我们可以直接指定委托来实例化对象：

class Bat : CanFly by AnimalWithWings()

...

val bat = Bat()

bat.fly()

## 在我们的App中实现一个例子

接口可以被用来从类中提取出相似行为的通用代码。比如，我们可以创建一个接口用于处理app的toolbar。MainActivity和DetailActivity在处理toolbar时会共享这些相似的代码。

但是受限，我们需要做出一些改变，使用被定义在布局中toolbar，而不是标准的ActionBar。第一件事是继承NoActionBar主题。这样toolbar不会自动被包含进来：

<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">

<item name="colorPrimary">#ff212121</item>

<item name="colorPrimaryDark">@android:color/black</item>

</style>

我们使用light主题。然后我们创建一个toolbar的布局，我们稍后会在其它的布局中使用到它：

<android.support.v7.widget.Toolbar

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/toolbar"

android:layout\_width="match\_parent"

android:layout\_height="?attr/actionBarSize"

android:background="?attr/colorPrimary"

app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"

app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

toolbar指定了它自己的背景，一个针对自己的dark主题和一个针对生成的弹出框的light主题（overflow menu实例）。我们现在已经有了相同的主题：light主题和dark主题的Action Bar。

下一步我们将修改MainActivity的布局，增加一个toolbar：

<FrameLayout

xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<android.support.v7.widget.RecyclerView

android:id="@+id/forecastList"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:clipToPadding="false"

android:paddingTop="?attr/actionBarSize"/>

<include layout="@layout/toolbar"/>

</FrameLayout>

现在toolbar被增加到布局中，我们可以开始使用它。我们创建了一个接口，它可以让我们：

* 改变title
* 指定是否显示上一步的导航动作
* 滚动时的toolbar动画
* 给所有的activity设置相同的菜单，甚至行为

然后让我们定义ToolbarManager：

interface ToolbarManager {

val toolbar: Toolbar

...

}

它将需要一个toolbar属性。接口是无状态的，所以属性可以被定义，但是不能赋值。子类会实现这个接口并重写这个属性。

另一方面，我们可以不使用重写来实现无状态的属性。也就是说属性不需要维护一个backup field。一个处理toolbar title属性的例子：

var toolbarTitle: String

get() = toolbar.title.toString()

set(value) {

toolbar.title = value

}

因为属性仅仅使用了toolbar，它不需要保存任何新的状态。

我们现在创建了一个新的函数用来初始化toolbar，inflate一个menu并且设置一个listener：

fun initToolbar(){

toolbar.inflateMenu(R.menu.menu\_main)

toolbar.setOnMenuItemClickListener {

when (it.itemId) {

R.id.action\_settings -> App.instance.toast("Settings")

else -> App.instance.toast("Unknown option")

}

true

}

}

我们可以增加一个函数用来开启toolbar上面导航icon，设置一个箭头的icon并设置当icon被按压时触发的事件：

fun enableHomeAsUp(up: () -> Unit) {

toolbar.navigationIcon = createUpDrawable()

toolbar.setNavigationOnClickListener { up() }

}

private fun createUpDrawable() = with (DrawerArrowDrawable(toolbar.ctx)){

progress = 1f

this

}

这个函数接收一个listener，使用DrawerArrowDrawable来创建一个最后状态（当箭头已经显示时）的drawable，然后把listener设置给toolbar。

最后，接口将会提供一个函数，它允许toolbar可以attached到一个scroll上面，并且根据scroll的方向来执行动画。当往下滚动时toolbar会消失 ，往上滚动toolbar会再次显示：

fun attachToScroll(recyclerView: RecyclerView) {

recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {

override fun onScrolled(recyclerView: RecyclerView?, dx: Int, dy: Int) {

if (dy > 0) toolbar.slideExit() else toolbar.slideEnter()

}

})

}

我们会创建两个用于view从屏幕中显示或者消失动画的扩展函数。我们会检查是否动画之前没有执行过。这种方式可以避免每次不同的滚动view都会执行动画：

fun View.slideExit() {

if (translationY == 0f) animate().translationY(-height.toFlat())

}

fun View.slideEnter() {

if (translationY < 0f) animate().translationY(0f)

}

在toobar manager实现之后，是时候在MainActivity中使用它了。我们首先指定toolbar属性。我们可以使用lazy委托实现，这样会在我们第一次使用它的时候才会inflate：

override val toolbar by lazy { find<Toolbar>(R.id.toolbar) }

MainActivity将会仅仅初始化toolbar并attach到RecyclerView的滚动并修改toolbar的title：

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

initToolbar()

forecastList.layoutManager = LinearLayoutManager(this)

attachToScroll(forecastList)

async {

val result = RequestForecastCommand(94043).execute()

uiThread {

val adapter = ForecastListAdapter(result) {

startActivity<DetailActivity>(DetailActivity.ID to it.id,

DetailActivity.CITY\_NAME to result.city)

}

forecastList.adapter = adapter

toolbarTitle = "${result.city} (${result.country})"

}

}

}

DetailActivity使用相同的方式去指定toolbar属性。DetailActivity也会初始化toolbar，设置title并且开启导航返回icon：

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_detail)

initToolbar()

toolbarTitle = intent.getStringExtra(CITY\_NAME)

enableHomeAsUp { onBackPressed() }

...

}

ToolbarManager 接口，由于可以有默认实现，相当于绕过了多继承，将通用方法提到了接口

interface ToolbarManager {

val toolbar: Toolbar

var toolbarTitle: String

get() = toolbar.title.toString()

set(value) {

toolbar.title = value

}

fun initToolbar() {

toolbar.inflateMenu(R.menu.menu\_main)

toolbar.setOnMenuItemClickListener {

when (it.itemId) {

R.id.action\_settings -> toolbar.ctx.startActivity<SettingsActivity>()

else -> App.instance.toast("Unknown option")

}

true

}

}

fun enableHomeAsUp(up: () -> Unit) {

toolbar.navigationIcon = createUpDrawable()

toolbar.setNavigationOnClickListener { up() }

}

private fun createUpDrawable() = DrawerArrowDrawable(toolbar.ctx).apply { progress = 1f }

fun attachToScroll(recyclerView: RecyclerView) {

recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {

override fun onScrolled(recyclerView: RecyclerView?, dx: Int, dy: Int) {

if (dy > 0) toolbar.slideExit() else toolbar.slideEnter()

}

})

}

}

# 泛型

## 基础

举个例子，我们可以创建一个指定泛型类：

class TypedClass<T>(parameter: T) {

val value: T = parameter

}

这个类现在可以使用任何的类型初始化，并且参数也会使用定义的类型，我们可以这么做：

val t1 = TypedClass<String>("Hello World!")

val t2 = TypedClass<Int>(25)

但是Kotlin很简单并且缩减了模版代码，所以如果编译器能够推断参数的类型，我们甚至也就不需要去指定它：

val t1 = TypedClass("Hello World!")

val t2 = TypedClass(25)

val t3 = TypedClass<String?>(null)

如第三个对象接收一个null引用，那仍然还是需要指定它的类型，因为它不能去推断出来。

我们可以像Java中那样在定义中指定的方式来增加类型限制。比如，如果我们想限制上一个类中为非null类型，我们只需要这么做：

class TypedClass<T : Any>(parameter: T) {

val value: T = parameter

}

如果你再去编译前面的代码，你将看到t3现在会抛出一个错误。可null类型不再被允许了。但是限制明显可以更加严厉。如果我们只希望Context的子类该怎么做？很简单：

class TypedClass<T : Context>(parameter: T) {

val value: T = parameter

}

现在所有继承Context的类都可以在我们这个类中使用。其它的类型是不被允许的。

当然，可以使用函数中。我们可以相当简单地构建泛型函数：

fun <T> typedFunction(item: T): List<T> {

...

}

# 设置界面

## 创建一个设置activity

当toolbar上溢出菜单（overflow menu）的settings选项被点击时，需要打开一个新的Activity。所以首先要做的事情时需要一个新的SettingActivity：

class SettingsActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_settings)

setSupportActionBar(toolbar)

supportActionBar.setDisplayHomeAsUpEnabled(true)

}

override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {

android.R.id.home -> { onBackPressed(); true }

else -> false

}

}

## 访问Shared Preferences

从Android 6.0（Marshmallow），shared preferences可以自动被云存储，所以当一个用户在一个新的设备上面恢复App的时候，它们的preferences也会被恢复。

多亏使用了属性委托，我们可以使用非常简单的方式来处理preferences。我们可以创建一个委托，当get被调用时去查询，当set被调用时去执行保存操作。

因为我们想去保存zip code，它是一个long型，所以让我们创建一个Long属性的委托吧。在DelegatesExtensions.kt中，实现一个新的LongPreference类：

class LongPreference(val context: Context, val name: String, val default: Long)

: ReadWriteProperty<Any?, Long> {

val prefs by lazy {

context.getSharedPreferences("default", Context.MODE\_PRIVATE)

}

override fun getValue(thisRef: Any?, property: KProperty<\*>): Long {

return prefs.getLong(name, default)

}

override fun setValue(thisRef: Any?, property: KProperty<\*>, value: Long) {

prefs.edit().putLong(name, value).apply()

}

}

首先，我们使用lazy委托的方式创建一个preferences。这样的话，如果我们没有使用这个属性，这个委托就不会请求这个SharedPreferences对象。

当get被调用，它的实现是使用preferences实例去获取一个委托声明中指定名字的long属性，如果没有找到这个属性，则默认使用default。当一个值被set，拿到preferences editor并使用属性名保存。

我们可以在DelegatesExt中定义一个新的委托，这样我们访问时就简单很多：

object DelegatesExt {

....

fun longPreference(context: Context, name: String, default: Long) =

LongPreference(context, name, default)

}

在SettingActivity，现在可以定义一个属性去处理zip code偏好。我创建了两个常量用来作为名字和属性的默认值。这种方式可以在App其他地方使用：

companion object {

val ZIP\_CODE = "zipCode"

val DEFAULT\_ZIP = 94043L

}

var zipCode: Long by DelegatesExt.longPreference(this, ZIP\_CODE, DEFAULT\_ZIP)

现在preference工作起来就非常简单了，我们可以从属性中得到并赋值给EditText：

override fun onCreate(savedInstanceState: Bundle?) {

...

cityCode.setText(zipCode.toString())

}

我们不能使用自动生成的属性text，因为EditText在getText中返回的是Editable，所以该属性默认为该值。如果我尝试去分配一个String，编译器会报错，使用setText()就足够了。

现在具备了所有要实现onBackPressed的东西。这里，一个属性的新值会被储存：

override fun onBackPressed() {

super.onBackPressed()

zipCode = cityCode.text.toString().toLong()

}

MainActivity需要一些小的改变。首先，它也需要一个zip code属性。

val zipCode: Long by DelegatesExt.longPreference(this, SettingsActivity.ZIP\_CODE,

SettingsActivity.DEFAULT\_ZIP)

然后，我把forecast load的代码移动到了onResume，这样每次activity resumed，它都会刷新数据，以防code zip被修改。当然这里有更加复杂一点的方式去做，比如通过在请求forecast之前检查是否zip code真的改变了。但是我像保持这个例子的简单性，而且因为请求的数据已经保存在本地数据库中了，所以这个解决方案也不算太坏：

override fun onResume() {

super.onResume()

loadForecast()

}

private fun loadForecast() = async {

val result = RequestForecastCommand(zipCode).execute()

uiThread {

val adapter = ForecastListAdapter(result) {

startActivity<DetailActivity>(DetailActivity.ID to it.id,

DetailActivity.CITY\_NAME to result.city)

}

forecastList.adapter = adapter

toolbarTitle = "${result.city} (${result.country})"

}

}

RequestForecastCommand现在使用zipCode而不是之前的是一个固定值。

这里还有意见我们必须要做的事情：当溢出菜单的settings点击时启动这个setting activity。在ToolbarManager中的initToolbar函数需要有一些小的修改：

when (it.itemId) {

R.id.action\_settings -> toolbar.ctx.startActivity<SettingsActivity>()

else -> App.instance.toast("Unknown option")

}

## 泛型preference委托

扩展LongPreference为支持所有Shared Preferences支持的类型呢？我们来创建一个Preference委托：

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)

}

...

}

这个preference与我们之前使用的非常相似。我们仅仅替换了Long为泛型类型T，然后调用了两个函数来做具体重要的工作。这些函数非常简单，尽管有些重复。它们会检查类型然后使用指定的方式来操作。比如，findPrefernce函数如下：

private fun <T> findPreference(name: String, default: T): T = 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 can be saved into Preferences")

}

res as T

}

putPreference函数也是一样，但是在when最后通过apply，使用preferences editor保存结果：

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 can be saved into Preferences")

}.apply()

}

现在修改DelegateExt：

object DelegatesExt {

...

fun preference<T : Any>(context: Context, name: String, default: T) = Preference(context, name, default)

}

# 测试你的app

## Unit testing

我不会对unit testing（单元测试）是什么的话题展开讨论。存在很多定义，但是都有一些细微的不同。一个普通的观点可能是unit testing验证一个单位（unit）的源代码的测试。一个单位（unit）包含什么就留给读者了。在我们的例子中，我仅仅去定义了一个unit test作为一个不需要设备运行的测试。IDE将会运行这些测试然后显示最后的结果分辩哪些测试成功哪些测试失败了。

Unit testing通常使用JUnit库。所以让我们增加这个依赖到build.gradle。因为这个依赖只会在跑测试的时候才会用到，所以我们可以使用testImplementation而不是implementation。用这种方式，这个库会在正式编译时忽略掉，可以减少APK的大小：

dependencies {

...

testImplementation 'junit:junit:4.12'

}

现在同步gradle来获取该库并加入到你的项目中。为了开启unit testing，打开Build Variantstab（你可能可以在IDE的左边找到它），点击Test Artifact下拉，你应该选择Unit Tests。

另一件你需要做的事情是创建一个新的文件夹。在src下面，你可能已经有androidTest和main了。创建另一个名为test的文件夹，再在它下面创建一个java文件夹。所以现在你应该有一个名为src/test/java绿色的文件夹。这是IDE发现我们在使用Unit Test模式好的迹象，这个文件夹中将会包括一些测试文件。

我们来写一个非常简单的测试来看看一切是不是正常运行了。使用合适的包名（我的是com.antonioleiva.weatherapp，但是你需要使用你app中的主包名）创建一个新的名为SimpleTest的Kotlin类。当你创建完，编写如下简单的测试：

import org.junit.Test

import kotlin.test.assertTrue

class SimpleTest {

@Test fun unitTestingWorks() {

assertTrue(true)

}

}

使用@Test注解来辨别该函数为是一个测试。确认是org.unit.Test。然后增加一个简单的断言。它只是判断了true是否是true，它显然会成功。这个测试只是用开确认一切配置正确。

执行测试，只需要在你在test下创建的新的java文件夹上右击，然后选择Run All Tests。当编译完成后，它会运行测试并会看见结果简介的显示。你应该可以看见我们的测试通过了。

现在是时候创建一个真正的测试了。所有使用Android框架来处理的测试可能都需要一个instrumentation test或者使用更复杂的像Robolectric库。所以在这些例子中我会不使用框架的任何东西。举个例子，我将测试从Long转String的扩展函数。

创建一个新的名为ExtensionTests的文件，然后增加如下测试：

class ExtensionsTest {

@Test fun testLongToDateString() {

assertEquals("Oct 19, 2015", 1445275635000L.toDateString())

}

@Test fun testDateStringFullFormat() {

assertEquals("Monday, October 19, 2015",

1445275635000L.toDateString(DateFormat.FULL))

}

}

这些测试检测Long实例是否可以转换成一个String。第一个测试默认行为（使用DateFormat.MEDIUM)），而第二个指定一个不同的格式。运行这些测试然后你会看到它们都通过了。我建议你修改它们然后看看它们失败是怎么样的。

如果你在Java中使用过测试，你将会发现这并没有什么太多的不同。我会演示一个简单的例子，我们可以对ForecastProvider进行一些测试。我们可以使用Mockito库来模拟其它的类然后独立测试provider：

dependencies {

...

testImplementation "junit:junit:4.12"

testImplementation "org.mockito:mockito-core:1.10.19"

}

现在创建了一个ForecastProviderTest。我们要去测试ForecastProvider，使用DataSource来返回结果，看它结果是否为null。所以首先我们需要模拟一个ForecastDataSource：

val ds = mock(ForecastDataSource::class.java)

`when`(ds.requestDayForecast(0)).then {

Forecast(0, 0, "desc", 20, 0, "url")

}

如你所见，我们需要在when上加反引号。因为when在Kotlin中是一个保留关键字，所以如果我们在一些Java代码中使用到它我们需要避免它。现在我们用这个数据源创建了一个provider，然后检测调用那个方法之后的结果是否为null：

val provider = ForecastProvider(listOf(ds))

assertNotNull(provider.requestForecast(0))

这是完整的测试函数：

@Test fun testDataSourceReturnsValue() {

val ds = mock(ForecastDataSource::class.java)

`when`(ds.requestDayForecast(0)).then {

Forecast(0, 0, "desc", 20, 0, "url")

}

val provider = ForecastProvider(listOf(ds))

assertNotNull(provider.requestForecast(0))

}

如果你运行它，你将会看见它会出错。多亏这个测试，我们在自己的代码中发现了某些错误。测试失败是因为ForecastProvider在使用之前正在它的companion object中初始化。我们可以通过构造函数的方式在ForecastProvider中增加一些数据源，这个静态的List就永远不会被使用，所以它应该是使用lazy加载：

companion object {

val DAY\_IN\_MILLIS = 1000 \* 60 \* 60 \* 24

val SOURCES by lazy { listOf(ForecastDb(), ForecastServer()) }

}

如果你现在再次去运行，你会发现现在会通过所有的测试。 我们也可以测试一些比如当数据源返回null的时候，它会便利下一个数据源来得到结果：

@Test fun emptyDatabaseReturnsServerValue() {

val db = mock(ForecastDataSource::class.java)

val server = mock(ForecastDataSource::class.java)

`when`(server.requestForecastByZipCode(

any(Long::class.java), any(Long::class.java)))

.then {

ForecastList(0, "city", "country", listOf())

val provider = ForecastProvider(listOf(db, server))

assertNotNull(provider.requestByZipCode(0, 0))

}

如你所见，通过使用参数的默认值这种简单的依赖倒置足够让我们实现一些简单的unit tests。对于这个provider还有很多我们可以测试的东西，但是这个例子足够让我们学会使用unit testing工具了。

## Instrumentation tests

Instrumentation tests有一点不同。它们通常被使用在UI交互上，我们需要一个应用程序实例跑的同时执行测试。达到这个，我们就需要在设备上部署并运行。

这类的测试必须要放在androidTest文件夹中，我们必须要修改Build Variants区域的Test Artifact为Android Instrumentation Tests。实现instrumentation的官方库是Espresso，它通过Actions、filter以及检测结果的ViewMatchers和Matchers可以帮助我们更简单地使用。

配置比之前更加难一点。我们需要下载额外的库和Gradle的配置。好事是Kotlin的测试不需要添加额外的东西，所以如果你已经知道怎么去配置Espresso，它将对你来说是很简单的。

首先，在defaultConfig中指定test runner：

defaultConfig {

...

testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

}

当你处理完该runner，然后增加其它的依赖，这次是用androidTestCompile。这种方式，这些库只会再编译运行instrumentation tests的时候才被增加：

dependencies {

...

androidTestCompile "com.android.support:support-annotations:$support\_version"

androidTestCompile "com.android.support.test:runner:0.4.1"

androidTestCompile "com.android.support.test:rules:0.4.1"

androidTestCompile "com.android.support.test.espresso:espresso-core:2.2.1"

androidTestCompile ("com.android.support.test.espresso:espresso-contrib:2.2.1"){

exclude group: 'com.android.support', module: 'appcompat'

exclude group: 'com.android.support', module: 'support-v4'

exclude module: 'recyclerview-v7'

}

我不想花大量的时间去讲这些，但是这里有为什么需要这些库的简短原因：

* support-annotations：其它库中需要使用到。
* runner：这是test runner，就是我们再defaultConfig中指定的那个。
* rules：包括一些测试inflate启动activity的规则。我们将会在我们的例子中使用这些规则。
* espresso-core：Espresso的基本实现，它让instrument tests更加容易。
* espresso-contrib：它增加了其它额外的功能，比如支持RecyclerView测试。我们不得不排除掉一些它的依赖，因为我们已经在这个项目中使用到了，否则测试会出错。

我们现在来创建一个简单的例子。测试将会点击forecast列表的第一行，然后它会判断是否能找到一个id为R.id.weatherDescription的view。这个view是在DetailActivity中的，这表示我们在测试在RecyclerView里面点击后是否可以成功地导航到详情页面。

class SimpleInstrumentationTest {

@get:Rule

val activityRule = ActivityTestRule(MainActivity::class.java)

...

}

首先我们需要指定它运行时使用AndroidJUnit4。然后，创建一个activity规则，它会实例化一个测试需要的activity。在Java中，你可以使用@Rule。但是如你所知，字段和属性是不一样的，所以如果你像那样去使用的话，执行会失败因为访问属性中的字段是不是public的。你需要加注解的是在getter上面。Kotlin允许指定get或者set在Rule的名字前面。在这个例子中，只要些@get:Rule。

之后，我们已经准备好创建第一个测试了：

@Test fun itemClick\_navigatesToDetail() {

onView(withId(R.id.forecastList)).perform(

RecyclerViewActions

.actionOnItemAtPosition<RecyclerView.ViewHolder>(0, click()))

onView(withId(R.id.weatherDescription))

.check(matches(isAssignableFrom(TextView::class.java)))

}

函数加上了@Test注解，这根我们使用unit test的方式一样。我们可以开始在测试体中使用Espresso。它首先在RecyclerView的第一个position中执行了一个点击。然后它检测是否可以找到一个指定id的view且这个view是一个TextView。

要运行这个测试，点击顶部的Run configurations下拉选择Edit Configurations...按下+图标，选择Android Tests，然后选择app模块。现在，在target device中选择你喜欢的target。点击OK然后运行。你应该可以看到App是怎样在你的设备中开始的，它会测试第一个position，打开详情页面然后再次关闭app。

现在我们要做一个更加复杂一点的事情。测试会从toolbar众打开一个溢出菜单，点击settings栏，改变城市的code，然后检测toolbar的标题是否改变成了对应的标题。

@Test fun modifyZipCode\_changesToolbarTitle() {

openActionBarOverflowOrOptionsMenu(activityRule.activity)

onView(withText(R.string.settings)).perform(click())

onView(withId(R.id.cityCode)).perform(replaceText("28830"))

pressBack()

onView(isAssignableFrom(Toolbar::class.java))

.check(matches(

withToolbarTitle(`is`("San Fernando de Henares (ES)"))))

}

这个测试实际做的事情：

* 它首先使用openActionBarOverflowOrOptionsMenu打开溢出菜单。
* 然后它根据Settings文本查找一个view，然后点击这个它。
* 之后，设置界面就会被打开，所以它会查找一个EditText并且替换成一个新的code。
* 它会点击返回按钮。它会把新的值保存在preferences中，然后关闭Activity。
* 因为MainActivity的onResume会调用，请求会再调用一次。这时它会获取到新城市的forecast。
* 最后一行将会检测Toolbar我们看到的title是否是新的城市的title。

这不是一个toolbar的title的默认匹配器，但是Espresso是很容易扩展的，所以我们可以创建一个新的matcher来实现检测：

private fun withToolbarTitle(textMatcher: Matcher<CharSequence>): Matcher<Any> =

object : BoundedMatcher<Any, Toolbar>(Toolbar::class.java) {

override fun matchesSafely(toolbar: Toolbar): Boolean {

return textMatcher.matches(toolbar.title)

}

override fun describeTo(description: Description) {

description.appendText("with toolbar title: ")

textMatcher.describeTo(description)

}

}

matchSafely函数是我们检测的地方，而describeTo函数为matcher增加了一些新的信息。

这章特别有趣，因为我们看到了在Kotlin中怎么样去完美和谐地整合测试，它们可以没有任何问题地整合测试。查看代码然后你自己运行一下吧。

# 其他的概念

## 内部类

在Java中，我们可以在类的里面再定义类。如果它是一个通常的类，它不能去访问外部类的成员（就如Java中的static）：

class Outer {

private val bar: Int = 1

class Nested {

fun foo() = 2

}

}

val demo = Outer.Nested().foo() // == 2

如果需要去访问外部类的成员，我们需要用inner声明这个类：

class Outer {

private val bar: Int = 1

inner class Inner{

fun foo() = bar

}

}

val demo = Outer().Inner().foo() // == 1

## 枚举

Kotlin也提供了枚举（enums）的实现：

enum class Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY

}

枚举可以带有参数：

enum class Icon(val res: Int) {

UP(R.drawable.ic\_up),

SEARCH(R.drawable.ic\_search),

CAST(R.drawable.ic\_cast)

}

val searchIconRes = Icon.SEARCH.res

枚举可以通过String匹配名字来获取，我们也可以获取包含所有枚举的Array，所以我们可以遍历它。

val search: Icon = Icon.valueOf("SEARCH")

val iconList: Array<Icon> = Icon.values()

而且每一个枚举都有一些函数来获取它的名字、声明的位置：

val searchName: String = Icon.SEARCH.name()

val searchPosition: Int = Icon.SEARCH.ordinal()

枚举根据它的顺序实现了 Comparable接口，所以可以很方便地把它们进行排序。

## 密封（Sealed）类

密封类用来限制类的继承关系，这意味着密封类的子类数量是固定的。看起来就像是枚举那样，当你想在一个密封类的子类中寻找一个指定的类的时候，你可以事先知道所有的子类。不同之处在于枚举的实例是唯一的，而密封类可以有很多实例，它们可以有不同的状态。

我们可以实现，比如类似Scala中的Option类：这种类型可以防止null的使用，当对象包含一个值时返回Some类，当对象为空时则返回None：

sealed class Option<out T> {

class Some<out T> : Option<T>()

object None : Option<Nothing>()

}

有一件关于密封类很不错的事情是当我们使用when表达式时，我们可以匹配所有选项而不使用else分支：

val result = when (option) {

is Option.Some<\*> -> "Contains a value"

is Option.None -> "Empty"

}

## 异常（Exceptions）

在Kotlin中，所有的Exception都是实现了Throwable，含有一个message且未经检查。这表示我们不会强迫我们在任何地方使用try/catch。这与Java中不太一样，比如在抛出IOException的方法，我们需要使用try-catch包围代码块。通过检查exception来处理显示并不是一个好的方法。像Bruce Eckel、Rod Waldhoff或Anders Hejlsberg等人可以给你关于这个更好的观点。

抛出异常的方式与Java很类似：

throw MyException("Exception message")

try表达式也是相同的：

try{

// 一些代码

}

catch (e: SomeException) {

// 处理

}

finally {

// 可选的finally块

}

在Kotlin中，throw和try都是表达式，这意味着它们可以被赋值给一个变量。这个在处理一些边界问题的时候确实非常有用：

val s = when(x){

is Int -> "Int instance"

is String -> "String instance"

else -> throw UnsupportedOperationException("Not valid type")

}

或者

val s = try { x as String } catch(e: ClassCastException) { null }

**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 一10一"

**目 录**

[1. 介绍 3](#_Toc28085)

[2. 准备工作 3](#_Toc20835)

[3. 创建一个新的项目 3](#_Toc7958)

[3.1. 配置Gradle 3](#_Toc9974)

[3.2. 把MainActivity转换成Kotlin代码 4](#_Toc27355)

[3.3. 测试是否一切就绪 4](#_Toc16718)

[4. 类和函数 4](#_Toc31109)

[5. 编写你的第一个类 4](#_Toc30813)

[5.1. 创建一个layout 4](#_Toc14028)

[5.2. The Recycler Adapter 5](#_Toc30282)

[6. 变量和属性 5](#_Toc1121)

[6.1. 基本类型 5](#_Toc32155)

[6.2. 属性 6](#_Toc19425)

[7. Anko和扩展函数 7](#_Toc1504)

[7.1. Anko是什么 7](#_Toc12640)

[7.2. 开始使用Anko 7](#_Toc2715)

[7.3. 扩展函数 7](#_Toc29481)

[8. 从API中获取数据 7](#_Toc1650)

[8.1. 执行一个请求 7](#_Toc10411)

[8.2. 在主线程以外执行请求 8](#_Toc9460)

[9. 数据类 8](#_Toc25006)

[9.1. 复制一个数据类 8](#_Toc18289)

[9.2. 自动创建构析声明 8](#_Toc28736)

[10. 解析数据 9](#_Toc7613)

[10.1. 转换json到数据类 9](#_Toc15098)

[10.2. 构建domain层 9](#_Toc22993)

[10.3. 在UI中绘制数据 10](#_Toc8852)

[11. 操作符重载 11](#_Toc32164)

[11.1. 操作符表 11](#_Toc7386)

[11.2. 例子 11](#_Toc31930)

[11.3. 扩展函数中的操作符 12](#_Toc30845)

[12. 使Forecast list可点击 12](#_Toc30522)

[13. Lambda 12](#_Toc6242)

[13.1. 扩展语言 12](#_Toc17025)

[14. 可见性修饰符 12](#_Toc10832)

[15. Kotlin Android Extensions 13](#_Toc21393)

[15.1. 怎么去使用Kotlin Android Extensions 13](#_Toc15409)

[15.1.1. Activities或者Fragments的Android Extensions 13](#_Toc470)

[15.1.2. Views的Android Extensions 13](#_Toc27718)

[16. Application单例化和属性的Delegated 13](#_Toc3337)

[16.1. Applicaton单例化 13](#_Toc5111)

[16.2. 委托属性 14](#_Toc26431)

[16.3. 标准委托 14](#_Toc2597)

[16.3.1. Lazy 14](#_Toc3175)

[16.3.2. Observable 15](#_Toc16692)

[16.3.3. Vetoable 15](#_Toc15045)

[16.3.4. Not Null 15](#_Toc26140)

[16.3.5. 从Map中映射值 16](#_Toc15399)

[16.4. 怎么去创建一个自定义的委托 16](#_Toc4418)

[16.5. 重新实现Application单例化 17](#_Toc11756)

[17. 创建一个SQLiteOpenHelper 17](#_Toc24302)

[17.1. ManagedSqliteOpenHelper 18](#_Toc26132)

[17.2. 定义表 18](#_Toc13461)

[17.3. 实现SqliteOpenHelper 19](#_Toc17897)

[17.4. 依赖注入 20](#_Toc24915)

[18. 集合和函数操作符 20](#_Toc3861)

[18.1. 总数操作符 21](#_Toc31554)

[18.2. 过滤操作符 22](#_Toc13840)

[18.3. 映射操作符 23](#_Toc10395)

[18.4. 元素操作符 23](#_Toc17738)

[18.5. 生产操作符 24](#_Toc15038)

[18.6. 顺序操作符 25](#_Toc16012)

[19. 从数据库中保存或查询数据 26](#_Toc29332)

[19.1. 创建数据库model类 26](#_Toc5207)

[19.2. 写入和查询数据库 27](#_Toc16616)

[19.2.1. 创建ForecastDb 27](#_Toc25426)

[19.2.2. 查询一个forecast 27](#_Toc10926)

[19.2.3. 保存一个forecast 28](#_Toc5120)

[20. Kotlin中的null安全 30](#_Toc30794)

[20.1. 可null类型怎么工作 30](#_Toc3928)

[21. 创建业务逻辑来访问数据 31](#_Toc20573)

[22. Flow control 和ranges 33](#_Toc17655)

[23. 创建一个详情界面 33](#_Toc8604)

[23.1. 获取color 33](#_Toc30721)

[23.2. 启动一个activity：reified函数 33](#_Toc7359)

[24. 接口和委托 34](#_Toc12273)

[24.1. 接口 34](#_Toc18058)

[24.2. 委托 35](#_Toc21724)

[24.3. 在我们的App中实现一个例子 35](#_Toc11135)

[25. 泛型 38](#_Toc31487)

[25.1. 基础 38](#_Toc18872)

[26. 设置界面 39](#_Toc16271)

[26.1. 创建一个设置activity 39](#_Toc11001)

[26.2. 访问Shared Preferences 39](#_Toc26908)

[26.3. 泛型preference委托 41](#_Toc14338)

[27. 测试你的app 42](#_Toc20494)

[27.1. Unit testing 42](#_Toc22430)

[27.2. Instrumentation tests 44](#_Toc30722)

[28. 其他的概念 46](#_Toc16136)

[28.1. 内部类 46](#_Toc18705)

[28.2. 枚举 46](#_Toc18074)

[28.3. 密封（Sealed）类 47](#_Toc4163)

[28.4. 异常（Exceptions） 47](#_Toc27882)

[29. 高阶函数、表达式函数、块主体函数的返回值 50](#_Toc26820)

[30. kotlin使用Parcelize注解简化Parcelable 50](#_Toc17333)

[31. let,apply,with,run等函数的区别 50](#_Toc31636)

[31.1. Standard.Kt代码 50](#_Toc7504)

[31.2. 前提介绍 52](#_Toc12736)

[31.3. with 53](#_Toc18039)

[31.4. let 53](#_Toc5484)

[31.5. apply 54](#_Toc14254)

[31.6. run 55](#_Toc16442)

[31.7. 另一个run（非扩展函数） 56](#_Toc17040)

[31.8. repeat 56](#_Toc27086)

[31.9. also 56](#_Toc8902)

[31.10. takeIf 57](#_Toc32634)

[31.11. takeUnless 57](#_Toc22391)

[31.12. 举例一，赋值： 58](#_Toc13922)

[31.13. 举例二，response： 58](#_Toc4462)

[31.14. 总结 59](#_Toc21391)

[31.14.1. let函数使用场景 60](#_Toc9040)

[31.14.2. with函数使用场景 60](#_Toc459)

[31.14.3. run函数的使用场景 60](#_Toc3219)

[31.14.4. apply函数的使用场景 60](#_Toc2882)

[31.14.5. also函数的使用场景 61](#_Toc15635)

[32. 与Java对比运算符 61](#_Toc15669)

[32.1. Kotlin与Java相同的运算符 61](#_Toc24190)

[32.2. Kotlin与Java使用有差异的运算符 62](#_Toc11848)

[32.2.1. ==和!= 62](#_Toc6635)

[32.2.2. [] 62](#_Toc9261)

[32.3. Java中有而Kotlin中没有的运算符 62](#_Toc17238)

[32.4. Kotlin中有而Java中没有的运算符 63](#_Toc31676)

[32.4.1. is和!is 63](#_Toc25052)

[32.4.2. in和!in 63](#_Toc20150)

[32.4.3. .. 63](#_Toc742)

[33. 写给Android开发者的Kotlin入门 63](#_Toc11468)

[33.1. Kotlin的特色 63](#_Toc30633)

[33.1.1. 简化findViewById 63](#_Toc10649)

[33.2. Anko 63](#_Toc22433)

[33.3. Kotlin语法特点 65](#_Toc20402)

[33.4. 函数式编程 70](#_Toc14325)

[33.4.1. 什么是编程 70](#_Toc18684)

[33.4.2. 纯函数 72](#_Toc18147)

[33.4.3. 不变性 73](#_Toc20198)

[33.4.4. 高阶函数 74](#_Toc13224)

[33.4.5. 闭包 74](#_Toc18742)

[33.4.6. Lambda表达式 75](#_Toc20090)

[33.4.7. 流式（Stream）API 75](#_Toc30865)

[33.5. Kotlin的潜在问题 76](#_Toc27028)

[33.5.1. 思维方式的问题 76](#_Toc12770)

[33.5.2. Kotlin->Java的转换 76](#_Toc2917)

[33.5.3. 团队开发的问题 76](#_Toc6725)

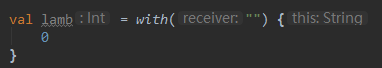
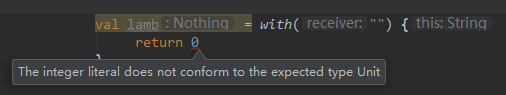
[34. 协程Coroutine 76](#_Toc17241)

# 高阶函数、表达式函数、块主体函数的返回值

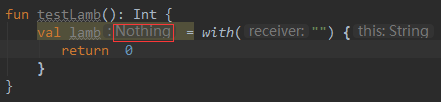
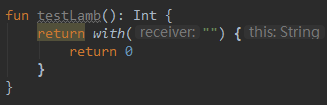
* 每一个函数都有一个返回类型,每一个变量和表达式都有一个类型
* 表达式主体函数，编译器使用表达式的类型作为函数返回类型，这种类型的分析通常叫做类型推断。
* 块主体的函数，除返回值类型是Unit，其他返回值类型必须显式指定。编译器不会自动推断，因为块主体函数在代码体中可能有复杂的控制流，并且返回类型对于读者甚至是编译器是很不明显的。
* 高阶函数(lambda表达式),lambda 主体中的最后一个表达式（或指定return表达式）会视为返回值。

对于lambda的返回值是否用return的场景

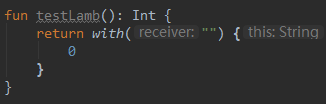
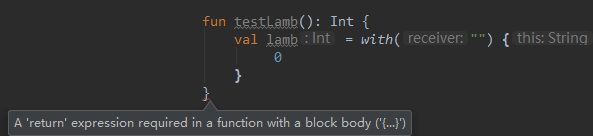
lambda是一个变量时（变量的类型是函数类型）,不能使用return，lambda中最后一个表达式是lambda的返回值

lambda是块函数的代码体，在lambda中使用return，将退回到fun关键字处，也就是fun有了返回值

lambda是块函数的代码体，如果不在lambda中使用return,fun需要返回值时，则必须使用return

# kotlin使用Parcelize注解简化Parcelable

在build.gradle中添加支持

apply plugin: ‘kotlin-android-extensions’

....

androidExtensions {

experimental = true

}

实体类

@Parcelize

data class Student(val id: String, val name: String, val grade: String) : Parcelable

# let,apply,with,run等函数的区别

## Standard.Kt代码

/\*Standard.Kt\*/

@file:kotlin.jvm.JvmMultifileClass

@file:kotlin.jvm.JvmName("StandardKt")

package kotlin

import kotlin.internal.contracts.\*

public class NotImplementedError(message: String = "An operation is not implemented.") : Error(message)

@kotlin.internal.InlineOnly

public inline fun TODO(): Nothing = throw NotImplementedError()

@kotlin.internal.InlineOnly

public inline fun TODO(reason: String): Nothing = throw NotImplementedError("An operation is not implemented: $reason")

@kotlin.internal.InlineOnly

public inline fun <R> run(block: () -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

}

@kotlin.internal.InlineOnly

public inline fun <T, R> T.run(block: T.() -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

}

@kotlin.internal.InlineOnly

public inline fun <T, R> with(receiver: T, block: T.() -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return receiver.block()

}

@kotlin.internal.InlineOnly

public inline fun <T> T.apply(block: T.() -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block()

return this

}

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.also(block: (T) -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block(this)

return this

}

@kotlin.internal.InlineOnly

public inline fun <T, R> T.let(block: (T) -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block(this)

}

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (predicate(this)) this else null

}

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (!predicate(this)) this else null

}

@kotlin.internal.InlineOnly

public inline fun repeat(times: Int, action: (Int) -> Unit) {

contract { callsInPlace(action) }

for (index in 0 until times) {

action(index)

}

}

## 前提介绍

Kotlin和Groovy等语言一样，支持闭包(block)，如果函数中最后一个参数为闭包，那么最后一个参可以不写在括号中，而写在括号后面，如果只有一个参数，括号也可以去掉。如下所示：

button.setOnClickListener({

println("test")

})

=>

button.setOnClickListener {

println("test")

}

## with

with函数是一个扩展函数，并不是Kotlin中的表达式(extension)

@kotlin.internal.InlineOnly

public inline fun <T, R> with(receiver: T, block: T.() -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return receiver.block()

}

代码示例

fun testWith() {

with(ArrayList<String>()) {

add("testWith")

add("testWith")

println("this = " + this)

}.let { println(it) }

}

>>

this = [testWith, testWith]

kotlin.Unit //with函数的最后一句是返回值，由于是println返回的是Unit，所以使用let后，打印it的结果是kotlin.Unit

class文件

public static final void testWith() {

Object localObject = new ArrayList();

ArrayList localArrayList1 = (ArrayList)localObject;

int $i$a$1$with;

ArrayList $receiver;

$receiver.add("testWith");

$receiver.add("testWith");

String str = "this = " + $receiver;

System.out.println(str);

localObject = Unit.INSTANCE;

Unit it = (Unit)localObject;

int $i$a$2$let;

System.out.println(it);

}

## let

let函数是一个扩展函数，扩展对象实例作为闭包的参数it

@kotlin.internal.InlineOnly

public inline fun <T, R> T.let(block: (T) -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block(this)

}

简单示例

fun testLet(): Int {

"testLet".let {

println(it)

return 1

}

}

>>

testLet

class文件

public static final int testLet() {

String str1 = "testLet";

String it = (String)str1;

int $i$a$1$let;

System.out.println(it);

return 1;

}

复杂示例

fun testLet(): Int {

"testLet".let {

if (Random().nextBoolean()) {

println(it)

return 1

} else {

println(it)

return 2

}

}

}

class文件

public static final int testLet() {

String str1 = "testLet";

String it = (String)str1;

int $i$a$1$let;

if (new Random().nextBoolean()) {

System.out.println(it);

return 1;

}

System.out.println(it);

return 2;

}

## apply

apply函数是一个扩展函数，扩展对象实例作为apply函数的返回值

@kotlin.internal.InlineOnly

public inline fun <T> T.apply(block: T.() -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block()

return this

}

代码示例

fun testApply() {

ArrayList<String>().apply {

add("testApply")

add("testApply")

println("this = " + this)

}.let { println(it) }

}

>>

this = [testApply, testApply]

[testApply, testApply]//applay函数返回接收者对象本身，所以使用let后，打印it的结果是[testApply, testApply]

class文件

public static final void testApply(){

ArrayList localArrayList1 = new ArrayList();

ArrayList localArrayList2 = (ArrayList)localArrayList1;

int $i$a$1$apply;

ArrayList $receiver;

$receiver.add("testApply");

$receiver.add("testApply");

String str = "this = " + $receiver;

System.out.println(str);

localArrayList1 = localArrayList1;

ArrayList it = (ArrayList)localArrayList1;

int $i$a$2$let;

System.out.println(it);

}

## run

run函数是一个扩展函数

@kotlin.internal.InlineOnly

public inline fun <T, R> T.run(block: T.() -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

}

代码示例

fun testRun() {

"testRun".run {

println("this = " + this)

}.let { println(it) }

}

>>

this = testRun

kotlin.Unit//run函数的最后一句是返回值，由于是println返回的是Unit，所以使用run后，打印it的结果是kotlin.Unit

class文件

public static final void testRun() {

Object localObject = "testRun";

String str1 = (String)localObject;

int $i$a$1$run;

String $receiver;

String str2 = "this = " + $receiver;

System.out.println(str2);

localObject = Unit.INSTANCE;

Unit it = (Unit)localObject;

int $i$a$2$let;

System.out.println(it);

}

## 另一个run（非扩展函数）

run是一个高阶函数

@kotlin.internal.InlineOnly

public inline fun <R> run(block: () -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

}

代码示例

fun main(args: Array<String>) {

val date = run {

Date()

}

println("date = $date")

}

>>

date = Thu Jan 04 19:31:09 CST 2018

## repeat

repeat是一个高阶函数，循环执行多少次block中内容，闭包的参数为递增的Int，repeat函数的返回值为Unit。

@kotlin.internal.InlineOnly

public inline fun repeat(times: Int, action: (Int) -> Unit) {

contract { callsInPlace(action) }

for (index in 0..times - 1) {

action(index)

}

}

代码示例

fun main(args: Array<String>) {

repeat(2) {

println("this is $it times - Hello world")

}

}

>>

this is 0 times - Hello world

this is 1 times - Hello world

## also

also函数是一个扩展函数，扩展对象实例作为闭包的参数it，扩展对象实例作为also函数的返回值

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.also(block: (T) -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block(this)

return this

}

代码示例

fun main(args: Array<String>) {

val also = Date().also {

println("in also time = " + it.time)

}

println("also = $also")

}

>>

in also time = 1515065830740

also = Thu Jan 04 19:37:10 CST 2018

## takeIf

takeIf函数是一个扩展函数，扩展对象实例作为闭包的参数it，满足闭包的条件，则返回扩展对象实例，否则返回null

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (predicate(this)) this else null

}

代码示例

fun main(args: Array<String>) {

val date = Date().takeIf {

it.after(Date(2018 - 1900, 0, 1))// 是否在2018年元旦后

}

println("date = $date")

}

>>

date = Thu Jan 04 19:42:09 CST 2018

## takeUnless

takeIf函数是一个扩展函数,逻辑和takeIf相反

@kotlin.internal.InlineOnly

@SinceKotlin("1.1")

public inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (!predicate(this)) this else null

}

代码示例

fun main(args: Array<String>) {

val date = Date().takeIf {

it.after(Date(2018 - 1900, 0, 1))// 是否在2018年元旦后

}

println("date = $date")

}

>>

date = null

## 举例一，赋值：

class User {

var id: Int = 0

var hobbies: List<String>? = null

}

普通赋值

var user = User()

user.id = 1

user.hobbies = listOf("aa", "bb", "cc")

使用库函数

user.let {

it.id = 2

it.hobbies = listOf("aa", "bb", "cc")

}

user.also {

it.id = 3

it.hobbies = listOf("aa", "bb", "cc")

}

user.apply {

id = 2

hobbies = listOf("aa", "bb", "cc")

Date()

}

user.run {

id = 3

hobbies = listOf("aa", "bb", "cc")

Date()

}

with(user) {

id = 4

hobbies = listOf("aa", "bb", "cc")

Date()

}

## 举例二，response：

class Resp<T> {

var code: Int = 0

var body: T? = null

var errorMessage: String? = null

fun isSuccess(): Boolean = code == 0

}

普通使用

var resp: Resp<String>? = Resp()

if (resp != null) {

if (resp.isSuccess()) {

println(resp.body)

} else {

println(resp.errorMessage)

}

}

使用库函数

resp?.let {

if (it.isSuccess()) {

println(it.body)

} else {

println(it.errorMessage)

}

}

resp?.also {

if (it.isSuccess()) {

println(it.body)

} else {

println(it.errorMessage)

}

}

resp?.run {

if (isSuccess()) {

println(body)

} else {

println(errorMessage)

}

}

resp?.apply {

if (isSuccess()) {

println(body)

} else {

println(errorMessage)

}

}

## 总结

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 函数名 | 定义 | block参数 | 闭包返回值 | 函数返回值 | extension | 其他 |
| repeat | fun repeat(times: Int, action: (Int) -> Unit) | 无 | Unit | Unit | 否 | 普通函数 |
| with | fun with(receiver: T, f: T.() -> R): R = receiver.f() | 无，可以使用this | Any | 闭包返回 | 否 | 普通函数 |
| run | run(block: () -> R): R | 无 | Any | 闭包返回 | 否 | 普通函数 |
| let | fun T.let(f: (T) -> R): R | it | Any | 闭包返回 | 是 | 扩展函数 |
| apply | fun T.apply(f: T.() -> Unit): T | 无，可以使用this | Unit | this | 是 | 扩展函数 |
| run | fun T.run(f: T.() -> R): R | 无，可以使用this | Any | 闭包返回 | 是 | 扩展函数 |
| also | fun T.also(block: (T) -> Unit): T | it | Unit | 闭包返回 | 是 | 扩展函数 |
| takeIF | fun T.takeIf(predicate: (T) -> Boolean): T? | it | Boolean | this 或 null | 是 | 扩展函数 |
| takeUnless | fun T.takeUnless(predicate: (T) -> Boolean): T? | it | Boolean | this 或 null | 是 | 扩展函数 |

### let函数使用场景

fun T.let(f: (T) -> R): R

场景一：最常用的场景，对一个可null的对象统一做判空处理，object不为null的条件下，才会去执行let函数体

object?.let {

it.todo()

}

场景二：在函数体内使用it替代object对象去访问其公有的属性和方法

object.let {

it.todo()

}

### with函数使用场景

fun with(receiver: T, f: T.() -> R): R = receiver.f()

调用同一个类的多个方法时，可以省去类名重复，直接调用类的方法即可，经常用于ViewHolder中

val item = getItem(position)?: return

with(item) {

holder.tvNewsTitle.text = StringUtils.trimToEmpty(titleEn)

holder.tvNewsSummary.text = StringUtils.trimToEmpty(summary)

}

### run函数的使用场景

fun T.run(f: T.() -> R): R

适用于let,with函数任何场景。因为run函数是let,with两个函数结合体，在run函数中可以像with函数一样可以省略接收者实例，直接访问实例的公有属性和方法；另一方面它弥补了with函数传入对象判空问题，在run函数中可以像let函数一样做判空处理

getItem(position)?.run{

holder.tvNewsTitle.text = StringUtils.trimToEmpty(titleEn)

holder.tvNewsSummary.text = StringUtils.trimToEmpty(summary)

}

### apply函数的使用场景

fun T.apply(f: T.() -> Unit): T

apply和run很像，不同点就是apply返回值是对象本身，run返回值是闭包的最后一行。

场景一：用于一个对象实例初始化的时候，需要对对象中的属性进行赋值。或者动态inflate出一个XML的View的时候需要给View绑定数据。

mSheetDialogView = View.inflate(activity, R.layout\_sheet\_inner, null).apply{

tv\_label.paint.isFakeBoldText = true

seek\_bar.max = 10

}

场景二：多层级判空问题

mSectionMetaData?.apply{ //mSectionMetaData不为空的时候操作mSectionMetaData

}?.questionnaire?.apply{ //questionnaire不为空的时候操作questionnaire

}?.sectionArticle?.apply{//sectionArticle不为空的时候操作sectionArticle

}

### also函数的使用场景

fun T.also(block: (T) -> Unit): T

also和let很像，不同点就also返回值是对象本身，let返回值是闭包的最后一行

适用于let函数的任何场景，一般可用于多个扩展函数链式调用

object.also{

it.todo()

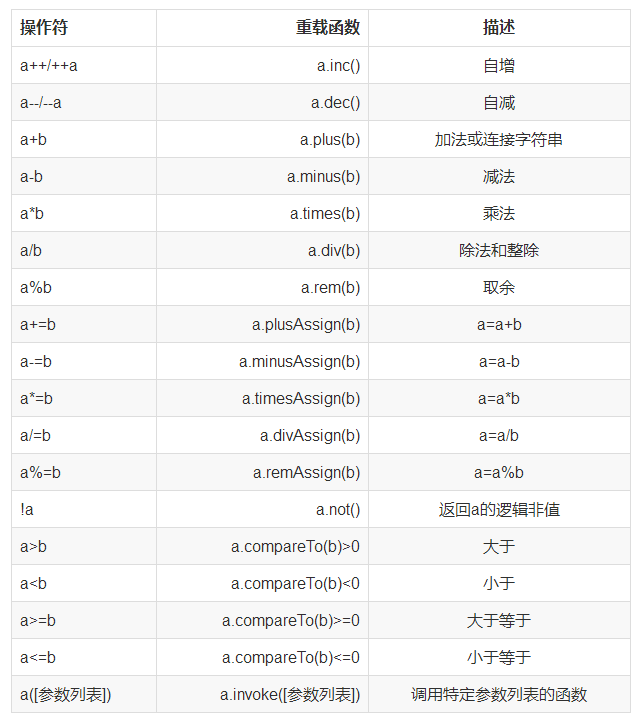
}.also{

it.todo2()

}

# 与Java对比运算符

## Kotlin与Java相同的运算符



## Kotlin与Java使用有差异的运算符

### ==和!=



a?.equals(b)?:b===null用Java写出来如下：

if(a != null){

return a.equals(b);

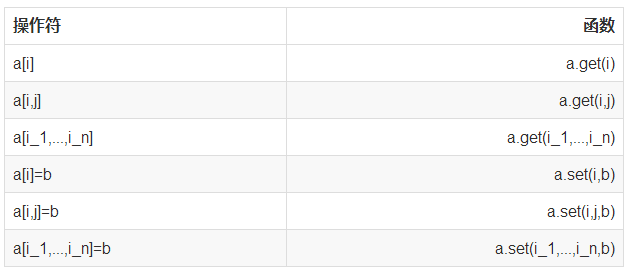
}else{

return b == null;

}

Kotlin调用的是equals方法来比较两个元素是否对象相等，而不是Java中的比较引用相同。如果要比较两个对象的引用是否相同，可以使用===和!==操作符

### []



与Java不同，Kotlin的这个运算符不仅可以用在数组变量后，也可以用在List和Map变量后，可以方便地调用和操作数组、List和Map中的元素，中缀表示法是Kotlin定义的一种特殊的函数调用方式，标记为infix，且该函数只有一个参数。

## Java中有而Kotlin中没有的运算符

Kotlin中抛弃了类C风格的位运算符，只是定义了几个函数，对于<<=、>>=、>>>=这三个赋值操作符，Kotlin中没有与之对应的函数：



setSystemUiVisibility(View.SYSTEM\_UI\_FLAG\_LAYOUT\_FULLSCREEN or View.SYSTEM\_UI\_FLAG\_LIGHT\_STATUS\_BAR)

## Kotlin中有而Java中没有的运算符

### is和!is

用于检查对象是否是某个类的实例，相当于Java中的instanceof运算符。

### in和!in

a in b重载为b.contains(a)，用于检查a是否是b内的元素，b可以是数组、集合和区间。此运算符也可以用在for循环中，用来遍历b中的元素：

val b = arrayOf(1, 2, 3)

for(a in b) println(a)

### ..

1. .b重载为a.rangeTo(b)，返回一个从a到b的区间，a和b也包含在区间内（闭区间）。要求a和b都是整数，并且a<=b。

# 写给Android开发者的Kotlin入门

## Kotlin的特色

Kotlin作为Java的改良，在Android开发中有很多优势，我们先从相对直观的界面绘制开始了解，然后看看Kotlin的语法特点，再慢慢去接触更深层次的编程思想。

### 简化findViewById

我们知道，Android的架构里，xml布局文件和Activity是松耦合的，Activity中要使用界面元素，必须借助R文件对xml控件的记录，用findViewById找到这个元素。

在Kotlin中我们可继续使用findViewById去绑定xml布局中的控件：findViewById(R.id.hello) as TextView；

进一步引用Anko之后，可以使用find函数去绑定控件：find(R.id.hello)，不需要类型转换

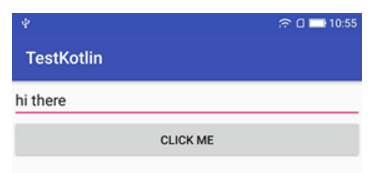
再进一步Kotlin还提供一种更激进的方法，通过在gradule中引用apply plugin:'kotlin-android-extensions'，彻底取消findViewById这个函数

## Anko

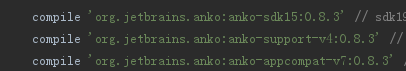
Anko其实是一种DSL，是专门用代码方式来写界面和布局的。上一节针对findViewById，最激进的方式是取消这个函数，这一节更加激进，我们可以连XML布局文件也取消掉。

在XML中定义界面布局当然是有好处的，分层清晰，代码易读，现在AS中预览效果也不错。但是它渲染过程复杂，难以重用（虽然有including），而如果我们用Java代码去替换xml，代码会更加复杂和晦涩。

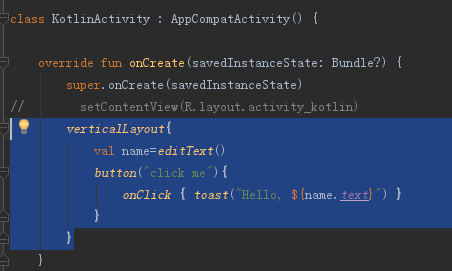
Anko却实现了在代码中简洁优雅地定义界面和布局，而且由于不需要读取和解析XML布局文件，Anko的性能表现更佳。Anko在Github上的代码示例，用6行代码就做出了一个有输入框、按钮、点击事件和Toast的界面和功能



我们自己写一下这6行代码，首先需要在gradle中添加引用，主要是sdk和v4/v7包



然后参照Anko在Github中的示例，实现这6行代码

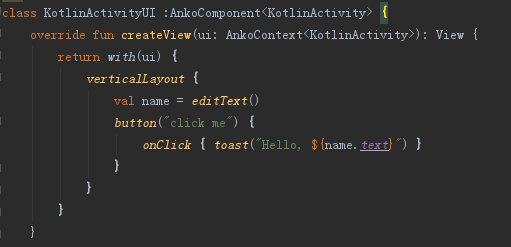


Activity本来会在加载时在onCreate函数里用setContentView函数来寻找布局文件，并加载为自己的界面，在这里，Anko代码替代了setContentView，直接告诉Activity应该如何绘制界面。

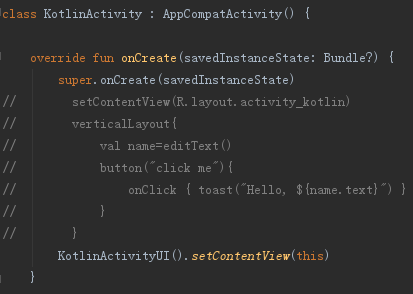
在Fragment里不可以这样直接写verticalLayout，因为加载机制不一样，Fragment需要在onCreateView函数里inflate并返回一个View对象，所以对应的Anko代码也需要写在onCreateView函数里并返回一个View，可以用return with(context){verticalLayout[...]}或者return UI{verticalLayout[...]}.view）

可以看到，代码非常简洁干练，不像以往的Android代码那样拖沓，这既与Kotlin的语法有关，也与Anko能用代码实现界面和布局有关。

这段代码虽然简洁，可是却失去了MVC分层的好处，因为V直接写在业务代码里了，这个问题好解决，我们可以把Anko布局代码放到一个专门的类文件里



然后在Activity引用这个布局类来绘制界面



虽然Anko效率很高，代码简洁，清爽直观，但是目前还有很多坑，主要包括：

1. AS并不支持直接预览Anko界面，虽然有个Anko DSL Preview插件，但是需要make才能刷新，而且和现在的AS不兼容。
2. 如果要在多版本中动态替换外部资源，需要用动态类加载才能实现，无法借用资源apk实现。
3. 不方便根据view的id去即时引用view控件（R文件和inflate这时反而更加灵活）。

另外，Anko还在异步、日志、Toast、对话框、数据库等方面提供优化服务，是否采用就看自身需要了。

## Kotlin语法特点

1. 没有“;”

在Kotlin语法里，代码行不需要用“;”结尾，什么都不写就好

1. 重要的“:”

在Java里，“:”主要在运算符里出现（for/switch/三元运算符等）。在Kotlin里，“:”的地位大大提升了，它的用途非常广泛，包括：

定义变量类型

var name: String = "my name" //变量name为String类型

定义参数的类型

fun makeTool(id: Int) { ... } //参数id为Int类型

定义函数的返回值

fun getAddr(id: Int): String { ... } //返回值为String类型

声明类/接口的继承

class KotlinActivityUI : AnkoComponent<KotlinActivity> { ... }//继承AnkoComponent接口

使用Java类

val intent = Intent(this, MainActivity::class.java) //需要用::来使用Java类

1. 没有“new”

Kotlin实例化一个对象时不需要new关键字

var list = ArrayList()

1. 变量、常量、类型推断

用var定义变量（像js）

var name: String = "my name"

用val定义常量（相当于final）

val TAG: String = "ClassName"

上面两个例子用:String来定义了数据类型，这个是可以省略的，Kotlin支持类型推断，这两句话你可以写成

var name="my name"

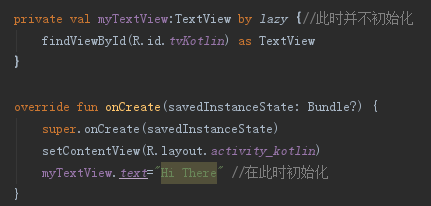
val TAG="ClassName"

1. 初始化和延迟加载

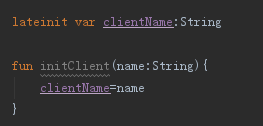
在Java里，我们可以定义一个变量，但是并不赋值（int和boolean会有默认值）但是Kotlin里必须为变量赋值，如果只写一个变量，却不赋值，像下面这样：

var name // 编译器会报错，提示你未初始化，你必须赋值为0或者null，或者别的什么值。

不过，我们有时候就是不能在定义变量时就初始化它，比如在Android中我们经常预定义一个View控件而不初始化，但是直到onCreate或onCreateView时才初始化它。针对这种情况，Kotlin提供了懒加载lazy机制来解决这个问题，在懒加载机制里，变量只有在第一次被调用时，才会初始化，代码需要这样写



lazy只适用于val对象，对于var对象，需要使用lateinit，原理是类似的，只是代码需要这样写



1. 空指针安全

在Kotlin里，可以用“?”表示可以为空，也可以用“!!”表示不可以为空。空指针安全并不是不需要处理空指针，你需要用“?”声明某个变量是允许空指针的，例如：

var num:Int?=null

声明允许为空时，不能使用类型推断，必须声明其数据类型。空指针虽然安全了，但对空指针的处理还是要视情况而定，有时候不处理，有时候做数据检查，有时候还需要抛出异常，这三种情况可以这样写：

val v1 =num?.toInt() //不做处理返回 null

val v2 =num?.toInt() ?:0 //判断为空时返回0

val v3 =num!!.toInt() //抛出空指针异常（用“!!”表示不能为空）

1. 定义函数

在Kotlin语法里，定义函数的格式是这样的

fun 方法名(参数名:类型,参数名:类型...) :返回类型{ ... }

所以，一般来说，函数是这样写的

fun getAddress(id:Int,name:String):String{ return"got it" }

由于Kotlin可以对函数的返回值进行类型推断，所以经常用“=”代替返回类型和“return”关键字，上面这段代码也可以写成

fun getAddress(id:Int,name:String)={ //用“=”代替return，返回String类型则交给类型推断

"got it" //return被“=”代替了

}

如果函数内代码只有一行，我们甚至可以去掉{}

fun getAddress(id:Int,name:String)="got it" //去掉了{}

函数也允许空指针安全，在返回类型后面增加“?”即可

fun getAddress(id:Int,name:String) :String?="got it"

有时候，函数的返回类型是个Unit，这其实就是Java中的void，表示没有返回

fun addAddress(id:Int,name:String):Unit{ ... }//相当于java的void

不过，在函数无返回时，一般省略不写Unit

fun addAddress(id:Int,name:String){ ... }//相当于java的void

1. 用is取代了instance of

代码很简单

if(obj is String){ ... }

1. in、区间和集合

Kotlin里有区间的概念，例如1..5表示的就是1-5的整数区间。可以用in判断数字是否在某个区间

if(x in 1..5){ ... }//检查x数值是否在1到5区间

可以用in判断集合中是否存在某个元素

if(name in list){ ... }//检查list中是否有某个元素（比Java简洁多了）

可以用in遍历整个集合

for(i in 1..5){ ... }//遍历1到5

for(item in list){ ... }//遍历list中的每个元素（相当于Java的for(String item : list)）

另外，in在遍历集合时功能相当强大：

在遍历集合时，可以从第N项开始遍历

for(i in 3..list.size-2){ ... }相当于for (int i = 3; i <= list.size()-2; i++)

可以倒序遍历

for(i in list.size downTo 0) { ... }相当于for (int i = list.size(); i >= 0; i--)

可以反转列表

for(i in (1..5).reversed())

可以指定步长

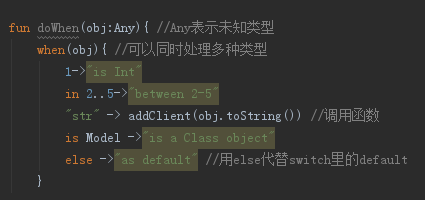
for(i in 1.0..2.0 step 0.3) //步长0.3

Kotlin里的集合还都自带foreach函数

list.forEach { ... }

1. 用when取代了switch

switch在Java里一直不怎么给力，在稍老一些的版本里，甚至不支持String。Kotlin干脆用强大的when取代了switch，具体用法如下



代码中的参数类型Any，相当于Java中的Obejct，是Kotlin中所有类的基类，至于object关键字，在Kotlin中另有用处...

1. 字符串模板

在Java里使用字符串模板没有难度，但是可读性较差，代码一般是

MessageFormat.format("{0}xivehribuher{1}xhvihuehewogweg",para0,para2);

在字符串较长时，你就很难读出字符串想表达什么。在kotlin里，字符串模板可读性更好

"${para0}xivehribuher${para1}xhvihuehewogweg"

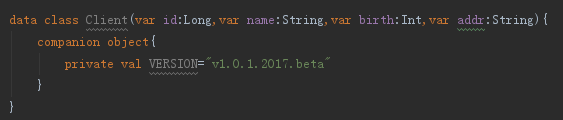
1. 数据类

数据类是Kotlin相对Java的一项重大改进，我们在Java里定义一个数据Model时，要做的事情有很多，例如需要定义getter/setter（虽然有插件代写），需要自己写equals()，hashCode()，copy()等函数（部分需要手写）

但是在Kotlin里，你只需要用data修饰class的一行代码，Kotlin会自动帮你实现前面说的那些特性

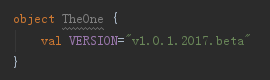
data class Client(var id:Long,var name:String,var birth:Int,var addr:String)

数据模型里经常需要一些静态属性或方法，Kotlin可以在数据类里添加一个companion object（伴随对象），让这个类的所有对象共享这个伴随对象（object在Kotlin中用来表示单例，Kotlin用Any来表示所有类的基类）



1. 单例模式

如果你在扩展了Kotlin的IDE里输入singleton，IDE也会自动帮你生成一个伴随对象，也就是一个单例



在Java中，单例模式：定义一个拥有私有构造函数和一个只保存唯一一个类实例的静态字段。Kotlin为使用对象声明特性提供了一流的语言支持。对象声明组合了类声明以及这个类的一个单例声明。

举个例子，你可以使用一个对象声明来表示一个组织的工资单。你可能不会有多个工资单，所以为此使用一个对象听起来更加合理：

object Payroll {

val allEmployees = arrayListOf<Person>()

fun calculateSalary() {

for (person in allEmployees) {

...

}

}

}

对象声明使用object关键字引入。一个对象声明定义了一个类和这个类的一个单例对象。就像一个类一样，一个对象声明可以包含属性、方法、初始化块等声明。唯一不允许的是构造函数。不同于正常的类实例，对象声明在定义类的时刻立即创建这个类的单例对象，因此，为一个对象声明定义一个构造函数并没有意义。跟一个类实例一样，一个对象声明允许你使用对象名.xxx来调用方法和访问属性

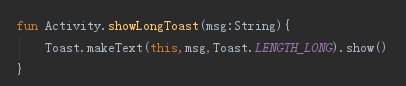
Payroll.allEmployees.add(Person(...))

Payroll.calculateSalary()

1. 为已存在的类扩展方法和属性

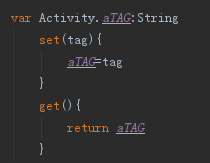
为了满足开放封闭原则，类是允许扩展，同时严禁修改的，但是实现扩展并不轻松，在Java里，我们需要先再造一个新的类，在新类里继承或者引用旧类，然后才能在新类里扩展方法和属性，实际上Java里层层嵌套的类也非常多。

在Kotlin里，这就简洁优雅地多，她允许直接在一个旧的类上做扩展，即使这是一个final类。例如，Android中常见的Toast，在Kotlin里，我们只需要这样写，就完成了Activity类的函数扩展，我们可以在Activity及其子类里随意调用了



需要注意的是，你无法用扩展去覆盖已存在的方法，例如，Activity里已经有一个onBackPressed方法，那么你再扩展一个Activity.onBackPressed方法是无用的，当你调用Activity().onBackPressed()时，它只会指向Activity本身的那个onBackPressed方法。

我们还可以用类似的方式去扩展属性



不过，Kotlin的扩展其实是伪装的，我们并没有真正给Activity类扩展出新的函数或属性，你在A类里为Activity扩展了函数，换到B类里，你就找不到这个函数了。这是因为，Kotlin为类扩展函数时，并没有真的去修改对应的类文件，只是借助IDE和编译器，使他看起来像扩展而已。（所以一般扩展函数都是携程顶层函数）

如果想为类永久性地添加某些新的特性，还是要利用继承或者装饰模式（decorator）。

1. 类的家族结构

Kotlin关于类的家族结构的设计，和Java基本相似，但是略有不同：

* Object:取消，在Java里Object是所有类的基类，但在Kotlin里，基类改成了Any
* Any:新增，Kotlin里所有类的基类
* object:新增，Kotlin是区分大小写的，object是Kotlin中的单例类
* new:取消，Kotlin不需要new关键字
* private: 仍然表示私有
* protected: 类似private，在子类中也可见
* internal: 模块内可见
* inner:内部类
* public: 仍然表示共有，但是Kotlin的内部类和参数默认为public
* abstract:仍然表示抽象类
* interface:仍然表示接口
* final:取消，Kotlin的继承和Java不同，Java的类默认可继承，只有final修饰的类不能继承；Kotlin的类默认不能继承，只有为open修饰的类能继承
* open:新增，作用见上一条
* static:取消！Java用static去共享同一块内存空间，这是一个非常实用的设计，不过Kotlin移除了static，用伴随对象（前面提到过的compaion object）的概念替换了static，伴随对象其实是个单例的实体，所以伴随对象比static更加灵活一些，能去继承和扩展。
* 继承:在Kotlin里，继承关系统一用“:”，不需要向java那样区分implement和extend，在继承多个类/接口时，中间用“,”区分即可，另外，在继承类时，类后面要跟()。所以在Kotlin里，继承类和接口的代码一般是这样的：

class MainActivity : Activity(), onClickListener{ ... }

1. 构造函数

在Java里，类的构造函数是这样的

public 类名作为函数名 (参数) {...}

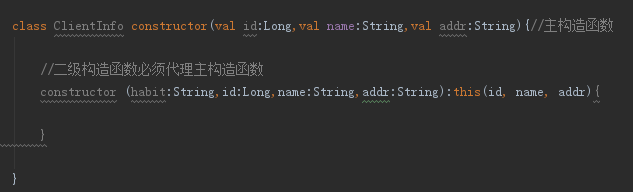
Java里有时会重载多个构造函数，这些构造函数都是并列的。在Kotlin里，类也可以有多个构造函数（constructor），但是分成了1个主构造函数和N个二级构造函数，二级构造函数必须直接或间接代理主构造函数，也就是说，在Kotlin里，主构造函数有核心地位。主构造函数一般直接写在类名后面，像这么写

class ClientInfo(id:Long,name:String,addr:String){...}

这其实是个缩写，完全版本应该是

class ClientInfo constructor(id:Long,name:String,addr:String){...}

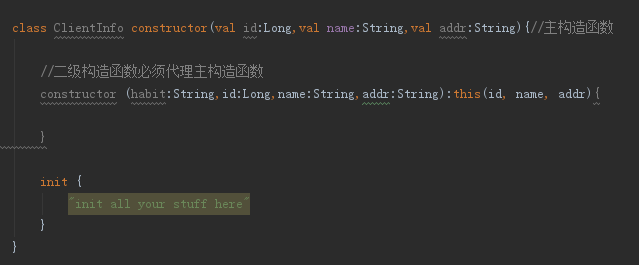
主构造函数的这个结构，基本决定了，在这个主构造函数里，没法写初始化代码,而二级构造函数必须代理主构造函数，写出来的效果是这样的



* 如果一个类没有声明任何构造函数，编译器自动生成一个没有参数的默认构造函数：
* 如果一个类有主构造函数，那么每一个次构造函数必须调用自己的主构造函数
* 如果一个子类有主构造函数，那么必须在类定义冒号后面引用父类的构造函数
* 如果一个子类没有主构造函数，那么每一个次构造函数必须调用父类的构造函数或者调用自己的其他次构造函数

1. 初始化模块init

上一节提到过，主构造函数里不能写代码，这就很麻烦了，不过还好，Kotlin提供了初始化模块，基本上就是用init修饰符修饰一个{}，在类初始化时执行这段儿代码，代码像这样写就行



## 函数式编程

### 什么是编程

但语法只是让我们能用Kotlin，要想用好Kotlin，就必须理解Kotlin背后的函数式编程理念。一个用惯了锤子的人，看什么都像是钉子，我们必须先扔掉锤子，再去理解函数式编程。我们先重新理解一下什么是计算机，什么是编程：

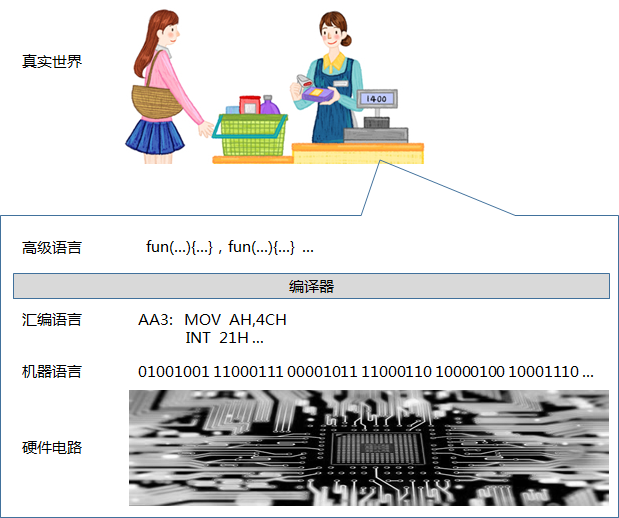
1.计算机：人发明计算机是为了计算数据（二战期间为了把炮弹打得更准，需要解大量的微积分，就造了台计算机帮忙，我们知道第一台通用计算机叫做ENIAC，这名字不是它的昵称绰号，就是它的功能，ENIAC的全称为Electronic Numerical Integrator And Computer，即电子数字积分计算机），直到现在，计算机程序在底层硬件电路上仍然是0和1的计算问题。

2.计算：计算机很笨，它其实只会计算0和1；但是人很聪明，人发现只要能把问题转换成0和1的运算，就可以丢给计算机去处理了，然后，几乎所有的问题，都可以设法转换成0和1的计算问题。

3.程序：一次或几次0和1的计算，几乎不能解决任何问题，需要很多次，步骤很复杂，过程很详细的0和1的计算才行，这种专为计算机提供的复杂而详细的计算步骤，就是计算机程序（为了向计算机传递程序，早期用打孔的纸带，后来用磁带，再后来用软盘，再后来是硬盘、光盘、闪存什么的...）。

4.编程：编程就是编写计算机程序，目的是把具体问题转换成0和1的运算问题，然后交给计算机去处理。

5.语言：编写计算机程序是给计算机用的，所以早期用的都是机器语言（全是0和1）。这样写出来的程序全是0和1，人自己反而看不懂，所以就抽象出汇编语言，就像把英文翻译成中文一样，这样人比较容易看懂。但是汇编语言描述的是底层电路的运算过程（把数据从内存的这里搬到那里，寄存器里的一个数据减去1，另一个数据乘以2），具体的输入、输出以及运算的目的都很难识别出来，所以又抽象出高级语言（C、BASIC等），不用再写底层电路如何操作（高级语言需要先经过编译器生成对应的汇编语言，再交给计算机去操作底层电路），只关心如何实现真实世界的业务逻辑。

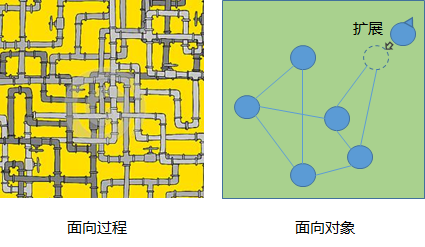


6.抽象：编程的目的是把具体问题转成0和1的计算问题，在高级语言里不用再考虑0和1了，我们可以更自由地把真实世界抽象为某种模型以便编写代码，这种抽象建模的过程，就是我们编程的核心能力

7.流派：关于如何对真实世界进行抽象，是有不同流派的，面向对象是和面向过程对应的，函数式编程是和命令式编程对应的

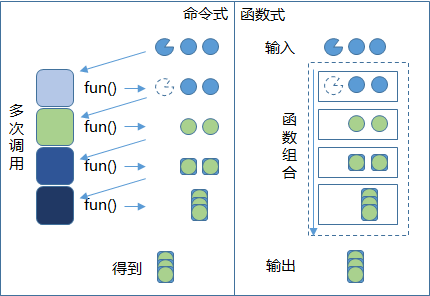
8.面向过程和面向对象：计算机的使命是用来计算，所有的计算都是有具体过程的，这样就会很自然地把真实世界映射为计算的过程，对真实世界的建模就是直接建出一个个业务的流程，然后去运转而已。但是日益复杂的流程会变成一团乱麻，难以理解，难以修复，难以扩展；

在面向对象中，不再纠结于流程本身，而是抽象出了对象的概念，把业务中的相关要素抽象为互相独立又互相调用的对象，对象和对象之间的关系（继承、封装、多态）成为核心，由于对象的概念更贴近人对于真实世界的理解，而且对象之间的关系也比整条复杂的流程简单，修改或者扩展起来的波及范围也小，容易理解/分解/修改/组合/扩展，所以面向对象非常适合大型的软件工程



9.命令式编程和函数式编程：换个角度来看，在计算机中实现业务逻辑有两种书写方式，一种是像输入命令一样，一步一步告诉计算机如何处理业务逻辑（还记得吗，计算机很笨，只会做它懂的事情），这就是命令式编程。如果命令有误，就是处理失败，如果要修改业务，就要把整个业务相关的命令都去检查和修改一遍。

另一种是告诉计算机，我需要什么，不去详细地告诉它要怎么做，由于计算机不可能理解我们的需求，所以我们把函数拼接到一起，让数据按照我们设想的方式流动，我们只要在数据流的最前面输入参数，等数据自己流完整个处理过程，就能得到我们需要的数据。如果数据有误或者需要修改业务，我们就去调整这个数据流，将它里面的数据流动调整为我们需要的方式。我们看到，函数式编程的运算过程是高度抽象的，能节省大量运算细节的代码编写和debug工作。



10.区别：面向对象和函数式编程是有区别的，面向对象把真实世界抽象为类和对象，函数式编程则把真实世界抽象为函数；面向对象关心的是对象的行为，以及对象之间的关系，而函数式编程关心的是函数的行为，以及对函数的组合运用；面向对象只要对象不出错，对象关系不出错就可以，函数式编程只要奔涌在函数组合里的数据流按照预期进行转换就可以。

11.选择：在抽象建模的概念里，面向对象因为贴近真实世界，相对简单容易理解，工程上还容易扩展维护，所以很长一段时间以来，面向对象在软件工程领域备受欢迎。

12.现实：从时间上来看，函数式编程其实并不新潮，但是过去主要活跃在大学和实验室里，这几年突然变得火热，背后一定有现实的原因。

13.硬件和并行：这些年来，对计算机的应用越来越广泛，丢给计算机处理的问题越来越多，计算量越来越大，所以计算机CPU就越来越快，一开始还能每18个月翻一番（摩尔定律），到了这几年单核CPU逼近物理极限，提升有限，就开始着重搞多核，并行计算也越来越重要。

14.数据的问题：计算机的本质在于计算数据，而软件最大的问题则是计算错误（出bug），不巧的是，面向对象编程在并行计算里就特别容易出现bug，因为她的核心是各种独立而又互相调用的对象，当多个对象同时处理数据时，就很容易导致数据修改的不确定性，从而引发bug。

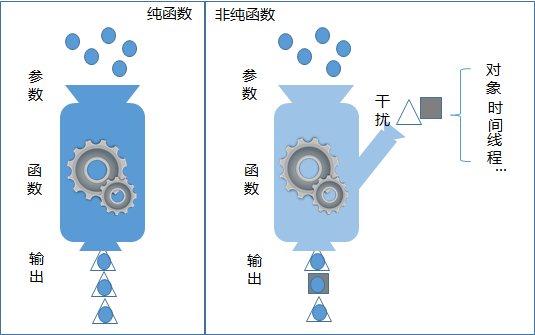
15.混合：编程的本质是把真实世界抽象映射到计算机的电路上，采用的抽象模式只是工具而已，我们没有必要排斥函数式编程，也不需要放弃面向对象，Kotlin也同时支持这两种方式，我们需要的是根据需要选用工具，用锤子，用扳手，或者两者都用。

要更深入地理解函数式编程，有一篇[So You Want to be a Functional Programmer](https://www.jianshu.com/p/bb53cba6c8f4)，写的非常好，在函数式编程里，我们需要用到纯函数、不变性、高阶函数、闭包等概念。

### 纯函数

开发者在学习编程之前，其实都学过数学，在数学的范畴里，函数的运算是不受干扰的，比如你算一个数字的平方根，只要参数确定，计算的过程永远是一致的，算出来的结果永远是一样的。

但是在学习编程（命令式编程）之后，函数就变了，变得“不纯洁”了，函数的运算会受到干扰，而且干扰无处不在，例如，我们可以在函数里使用一个会变化的全局变量，只要在任何位置/时间/线程里修改这个全局变量，函数就会输出不同的结果。



如果这种变化是开发者故意设计的，开发者就把它称为业务逻辑；如果这种变化不符合开发者的预期，开发者就把它称为——bug，悲剧的是，在命令式编程里，有无数的对象、时间点、线程可能对函数造成干扰。

在函数式编程里，重心是函数组合和数据流，更加不允许有干扰，所以要求我们编写纯函数。

不过，纯函数就像是编码规范，Kotlin鼓励而不是强制写出函数，毕竟，编程是为了与真实世界交互的，有时候必须使用一些“不纯洁”的函数，所以我们不要求彻底的纯函数化，只要求尽量写出纯函数

### 不变性

函数式编程不仅要求纯函数，还要求不变性（Kotlin用val和集合表示不变性，集合默认是不可变的），还是先回到数学上，在数学里，不允许这样的表达（我在刚学编程时，看到这个式子也是颠覆三观的）

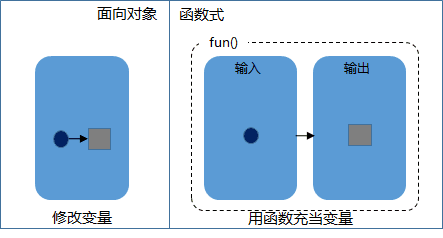
x = x + 1

在函数式编程里，这种表达也是非法的，也就是说，在函数式编程里，没有可变变量，变量一旦被赋值，就不可更改。

不变性有很多好处，这意味着程序运行的整个流程是固定可重现的，如果出了问题，只要跟着数据流走一遍就能找到出错点，再也不会有稀奇古怪的变化来为难我们。

不过，不变性最大的好处在于多线程安全，它可以完美地规避多个线程同时修改一个数据时的同步问题（变量不再允许修改，每个线程需要各自生成变量），这一点对于目前大量应用多线程的工程现状来说，特别有实际价值。

可是，如果变量不可变，我们还要怎样去做业务逻辑呢，函数式编程给出的方式就是——用函数去返回一个复制的新对象，在这个新的对象里，改掉你想改的那个值。更彻底地说，函数式编程里，没有变量，一切都是函数（就像面向对象编程里，一切都是对象），变量实际上被函数取代了



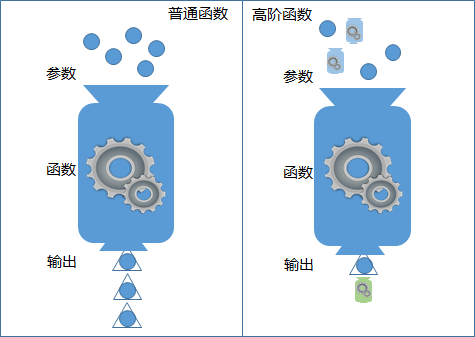
所以，函数式编程里只能新增变量，不能修改变量，所以函数式编程可能会非常耗内存（生成的变量太多了，而且业务不走完，变量不释放）

另外，在函数式编程里还有一个特点——没有循环，因为for(int i; i<9; i++)是非法的（当然，在Kotlin里你还可以这样写，因为Kotlin既支持函数式编程，又支持面向对象

### 高阶函数

既然变量已经被函数取代了，那么函数里的参数和返回值呢？这些对象是不是也可以被替换成为函数呢？

在面向函数编程里，有个重要的概念，叫做“函数是一等公民”，核心就是，函数拥有和数据一样的地位，都可以作为参数和返回值，相应的就出现了高阶函数的概念，简单理解，高阶函数就是参数为函数，或者返回值为函数的函数。



我们知道，在开发过程中，复用是非常重要的优化手段，说白了，能用1个函数就别用多个函数，不容易出错，出错也容易检查和修改。那么我们看下面这两个函数，要怎么优化？

fun getA(){

doA()

}

fun getB(){

doB()

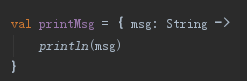
}

在面向对象编程里，我们第一反应是用接口和类来解决问题，当然，那样就得好几个类和接口，然后层层嵌套。有了高阶函数的话，开头那段代码就可以这样优化了

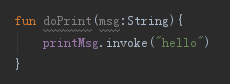
fun getAB(doA()){

}

在Kotlin里不能直接这么写，需要用Lambda表达式才行，lambda还可以作为一种类型，可以被定义为val



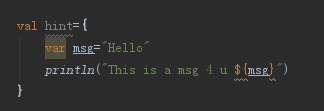
调用这个lambda类型的“对象”，与调用函数无异



lambda除非是内联的,都是被编译成实现了函数式接口(Function1等)的类,而这些接口定义了具有对应数量参数的invoke方法，由于invoke约定，所以也可以直接调用printMsg("hello")

### 闭包

前面说过，函数式编程里的函数是第一等公民，所以，一个val可以是一段代码，这就是一个闭包



不过，闭包不是函数，闭包在逻辑上是封闭的，它使用自己内部的数据，用自己内部的逻辑进行处理，外部只能得到闭包的输出，无法输入，也无法干扰。在系统资源上，闭包是持久使用的，它会一直在系统里，不像函数那样会被系统注销掉。闭包在函数式编程里可以简化参数量、减少变量，会更加方便我们的开发。

另外，函数式编程还有柯里化、inline、with、apply、let、run、it等概念，我们以后可以慢慢了解。

### Lambda表达式

为了写高阶函数和闭包，Kotlin支持我们使用Lambda表达式。Lambda看起来就是对匿名方法（如：回调、事件响应、Runnable等）的简化写法，目的是为了更贴近函数式编程把函数作为参数的思想。

Lambda表达式包括最外面的“{}”，用“()”来定义的参数列表，箭头->，以及一个表达式或语句块。

事件响应的简化：

textView.setOnClickListener(newOnClickListener(){

   @Override

   public void onClick(View view){/\*todo\*/}

});

简化为:

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

Runnable的简化：

executor.submit(new Runnable(){

@Override

public void run(){/\*todo\*/}

});

简化为：

executor.submit{/\*todo\*/}

使用lambda表达式，我们就可以编写高阶函数，传递一个函数（或者一段代码）作为参数。

### 流式（Stream）API

前面提过，函数式编程以数据流为中心，通过组合函数来整理一个数据流，通过调整这个函数组合得出需要的数据。要让数据流在组合函数里流动起来，就需要使用流式API，流式API使我们更容易把函数组合起来，而且使整个数据流动过程更加直观。

如果接触过Java8或者RxAndroid，应该很容易理解流式API，流式API写出来的代码风格如下:



## Kotlin的潜在问题

Kotlin也有一些潜在的问题是我们需要注意的，主要是开发时容易遇到的一些问题。

### 思维方式的问题

我们已经知道Kotlin的核心在于函数式编程，问题在于函数式编程的核心不是语法的问题，而是思维方式的问题，语法容易转变，思维却很难，所以没有函数式编程经验的话，切换到Kotlin其实会相当困难。

### Kotlin->Java的转换

我们应该注意，AS中只提供了从Java文件转换为Kotlin文件的工具，并没有逆向转换的工具，就是说目前你还不能很轻松地把Kotlin代码转换为Java代码，一件事情如果不能回退，就必须小心谨慎。

### 团队开发的问题

一般来说，鉴于Kotlin和Java兼容良好，可以一边维持旧的Java代码，一边开发新的Kotlin代码和新的Java代码，但是团队开发不仅是兼容性的问题，Kotlin语法糖背后的很多思维方式也许会对团队造成冲击，例如，一旦某个模块采用了流式API的话，其他团队成员在调用这个模块时，也需要理解并且能够编写流式API才能完成工作衔接，这就可能带来额外的成本和意外的延期。

# 协程Coroutine

说到Coroutine，必须提到两个更远的东西。在操作系统（os）级别，有进程（process）和线程（thread）两个东西。它们都是用来模拟"并行"的，写操作系统的程序员通过用一定的策略给不同的进程和线程分配CPU计算资源，来让用户"以为"几个不同的事情在"同时"进行"。

在单CPU上，是os代码强制把一个进程或者线程挂起，换成另外一个来计算，所以，实际上是串行的，只是"概念上的并行"。在现在的多核的cpu上，线程才可能是"真正并行的"。

Coroutine，翻译成"协程"，Coroutine是编译器级的，Process和Thread是操作系统级的。Coroutine的实现，通常是对某个语言做相应的提议，然后通过后成编译器标准，然后编译器厂商来实现该机制。Process和Thread内生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

Process和Thread是os通过调度算法，保存当前的上下文，然后从上次暂停的地方再次开始计算，重新开始的地方不可预期，每次CPU计算的指令数量和代码跑过的CPU时间是相关的，跑到os分配的cpu时间到达后就会被os强制挂起。Coroutine是编译器的魔术，通过插入相关的代码使得代码段能够实现分段式的执行，重新开始的地方是yield关键字指定的，一次一定会跑到一个yield对应的地方。