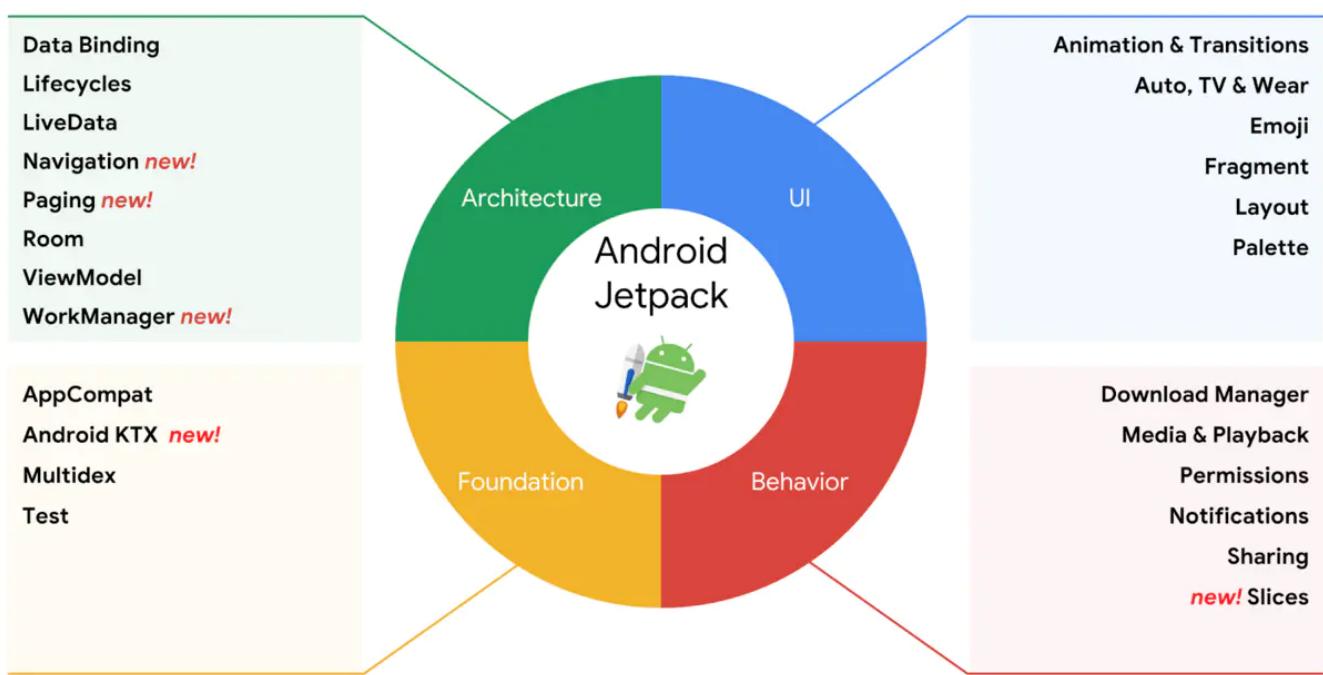


Jetpack架构组件从入门到精通

1. 什么是Jetpack



1.1 简介

Jetpack是一套库、工具和指南，可以帮助开发者更轻松地编写优质应用。这些组件可以帮助开发者遵循最佳做法、让开发者摆脱编写样板代码的工作并简化复杂任务，以便开发者将精力集中放在所需的代码上。

1.2 特性

- 加速开发** 组件可以单独采用（不过这些组件是为协同工作而构建的），同时利用Kotlin语言功能帮助开发者提高工作效率。
- 消除样板代码** Android Jetpack可管理繁琐的Activity（如后台任务、导航和生命周期管理），以便开发者可以专注于如何让自己的应用出类拔萃。
- 构建高质量的强大应用** Android Jetpack组件围绕现代化设计实践构建而成，具有向后兼容性，可以减少崩溃和内存泄漏。

1.3 分类

Architecture（架构组件）可帮助您设计稳健、可测试且易维护的应用。

1. **Data Binding**：是一种支持库，借助该库，可以以声明方式将可观察数据绑定到界面元素。

2. **Lifecycles**：管理Activity 和 Fragment的生命周期，能够帮助开发者轻松的应对Activity/Fragment的生命周期变化问题，帮助开发者生成更易于维护的轻量级代码。
3. **LiveData**：在底层数据库更改时通知视图，是可观察的数据持有者类。与常规的可观察对象不同，LiveData具有生命周期感知功能（例如Activity，Fragment或Service的生命周期）。
4. **Navigation**：处理应用内导航所需的一切。
5. **Paging**：逐步从您的数据源按需加载信息，帮助开发者一次加载和显示小块数据。按需加载部分数据可减少网络带宽和系统资源的使用。
6. **Room**：流畅地访问 SQLite 数据库。在SQLite上提供了一个抽象层，以在利用SQLite的全部功能的同时允许更健壮的数据库访问。
7. **ViewModel**：以注重生命周期的方式管理界面相关的数据。ViewModel类允许数据幸免于配置更改（例如屏幕旋转）。通常和DataBinding配合使用，为开发者实现MVVM架构提供了强有力的支持。
8. **WorkManager**：管理 Android 的后台作业，即使应用程序退出或设备重新启动，也可以轻松地调度预期将要运行的可延迟异步任务。

Foundation (基础组件) 可提供横向功能，例如向后兼容性、测试和 Kotlin 语言支持。

1. **Android KTX**：编写更简洁、惯用的 Kotlin 代码，是一组Kotlin扩展程序。优化了供Kotlin使用的Jetpack和Android平台API。旨在让开发者利用 Kotlin 语言功能（例如扩展函数/属性、lambda、命名参数和参数默认值），以更简洁、更愉悦、更惯用的方式使用 Kotlin 进行 Android 开发。Android KTX 不会向现有的 Android API 添加任何新功能。
2. **AppCompat**：帮助较低版本的Android系统进行兼容。
3. **Auto**：有助于开发 Android Auto 应用的组件。是 Google推出的专为汽车所设计之 Android 功能，旨在取代汽车制造商之原生车载系统来执行 Android应用与服务并访问与存取Android手机内容。
4. **Benchmark**：从 Android Studio 中快速检测基于 Kotlin 或 Java 的代码。
5. **Multidex**：为具有多个 DEX 文件的应用提供支持。
6. **Security**：按照安全最佳做法读写加密文件和共享偏好设置。
7. **Test**：用于单元和运行时界面测试的 Android 测试框架。
8. **TV**：有助于开发 Android TV 应用的组件。
9. **Wear OS by Google**：有助于开发 Wear 应用的组件。

Behavior (行为组件) 可帮助您的应用与标准 Android 服务（如通知、权限、分享和 Google 助理）相集成。

1. **CameraX**：简化相机应用的开发工作。它提供一致且易于使用的 API 界面，适用于大多数 Android 设备，并可向后兼容至 Android 5.0 (API 级别 21)。
2. **DownloadManager**：是一项系统服务，可处理长时间运行的HTTP下载。客户端可以请求将URI下载到特定的目标文件。下载管理器将在后台进行下载，处理HTTP交互，并在出现故障或在连接更改和系统重新启动后重试下载。
3. **Media & playback**：用于媒体播放和路由（包括 Google Cast ）的向后兼容 API。
4. **Notifications**：提供向后兼容的通知 API，支持 Wear 和 Auto。
5. **Permissions**：用于检查和请求应用权限的兼容性 API。
6. **Preferences**：创建交互式设置屏幕，建议使用 AndroidX Preference Library 将用户可配置设置集成至应用中。
7. **Sharing**：提供适合应用操作栏的共享操作。
8. **Slices**：是UI模板，可以通过启用全屏应用程序之外的互动来帮助用户更快地执行任务，即可以创建在应用外部显示应用数据的灵活界面。

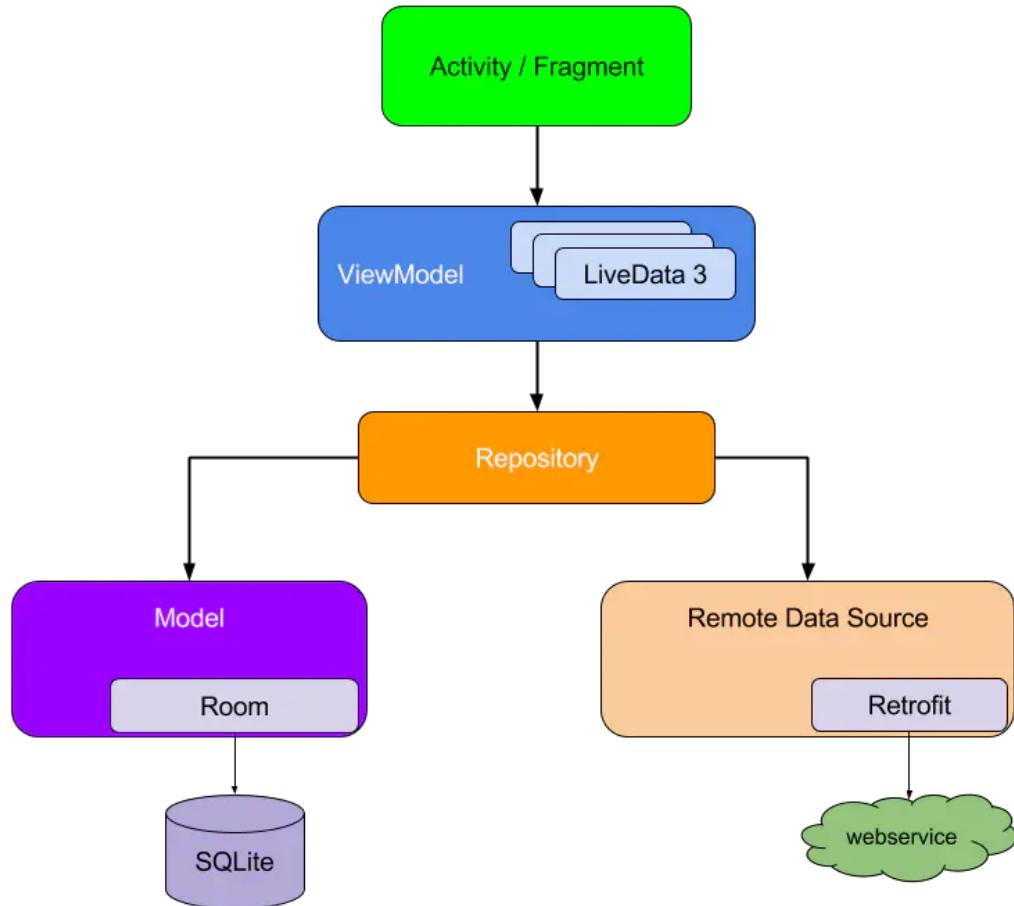
UI (界面组件) 可提供微件和辅助程序，让您的应用不仅简单易用，还能带来愉悦体验。了解有助于简化界面开发的 Jetpack Compose。

1. **Animation & transitions**：使开发者可以轻松地为两个视图层次结构之间的变化设置动画。该框架通过随时更改其某些属性值在运行时为视图设置动画。该框架包括用于常见效果的内置动画，并允许开发者创建自定

义动画和过渡生命周期回调。

2. **Emoji**：使Android设备保持最新的最新emoji表情，开发者的应用程序用户无需等待Android OS更新即可获取最新的表情符号。
3. **Fragment**：Activity的模块化组成部分。
4. **Layout**：定义应用中的界面结构。可以在xml中声明界面元素，也可以在运行时实例化布局元素。
5. **Palette**：是一个支持库，可从图像中提取突出的颜色，以帮助开发者创建视觉上引人入胜的应用程序。开发者可以使用调色板库设计布局主题，并将自定义颜色应用于应用程序中的视觉元素。

1.4 应用架构

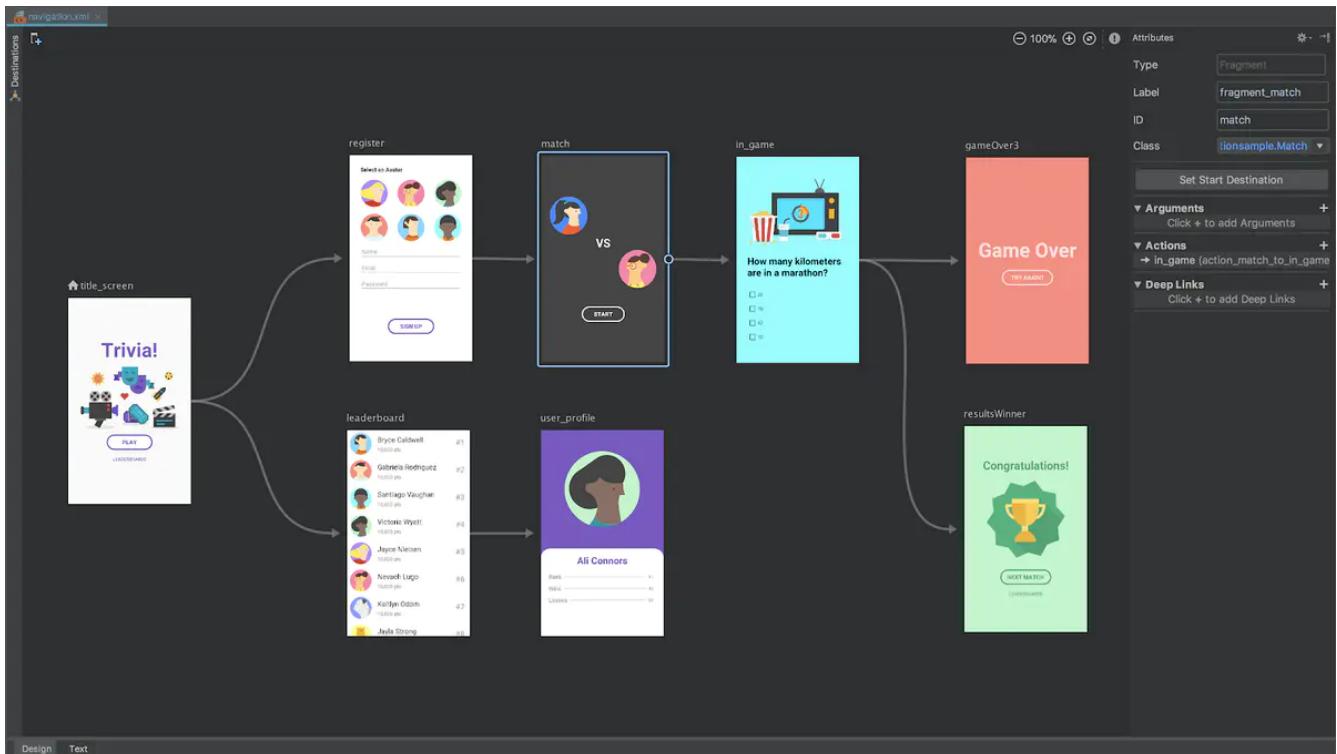


2. Android Jetpack - Navigation

2.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第一篇。

记得去年第一次参加谷歌开发者大会的时候，就被 `Navigation` 的图形导航界面给迷住了，一句卧槽就代表了小王的全部心情~，我们可以看一下来自网络的一张图片：



所以，Android Jetpack学习之旅就开始了。



Android Jetpack

本人打算每周学习一个组件（上图的左上区域），最后将所学的组件组成一个简单的Demo。同时，刚刚过去的2019年谷歌开发者大会宣布亲儿子

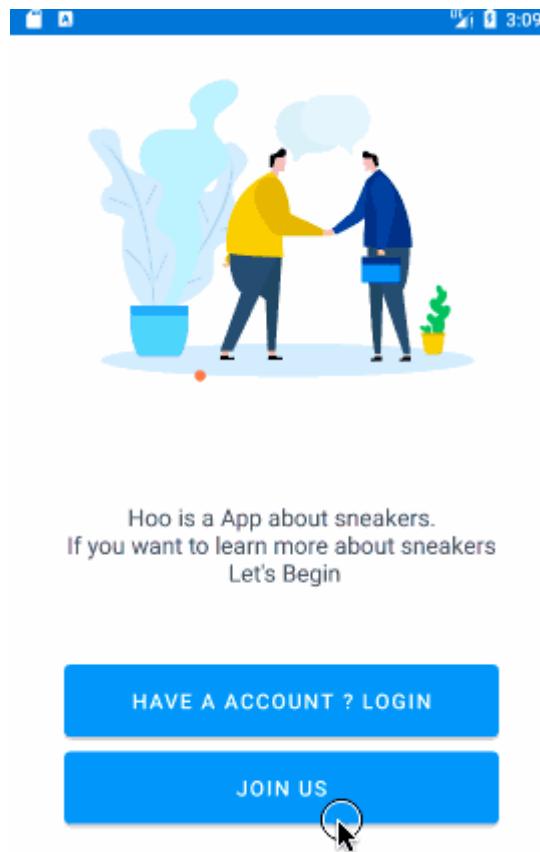
Kotlin

成为开发Android的首选语言，所以本文的Demo也将都会采用

Kotlin

编写。

本章结束后登录部分完成效果：



语言：

Kotlin

Demo地址：

<https://github.com/mCyp/Hoo>

2.2 简介

2.2.1 定义

`Navigation`是什么呢？谷歌的介绍视频上说：

`Navigation`是一个可简化Android导航的库和插件

更确切的来说，`Navigation`是用来管理`Fragment`的切换，并且可以通过可视化的方式，看见App的交互流程。这完美的契合了Jake Wharton大神单Activity的建议。

2.2.2 优点

- 处理`Fragment`的切换（上文已说过）
- 默认情况下正确处理`Fragment`的前进和后退
- 为过渡和动画提供标准化的资源
- 实现和处理深层连接
- 可以绑定`Toolbar`、`BottomNavigationView`和`ActionBar`等
- `SafeArgs`（Gradle插件）数据传递时提供类型安全性
- `ViewModel`支持

2.2.3 准备

如果想要进行下面的学习，你需要3.2或者更高的`Android studio`。

2.2.4 学习方式

最好的学习方式仍然是通过官方文档，下面是官方的学习地址：谷歌官方教程：[Navigation Codelab](#) 谷歌官方文档：[Navigation](#) 官方Demo：[Demo地址](#)

2.3 实战

在实战之前，我们先来了解一下`Navigation`中最关键的三要素，他们是：

名词	解释
<code>Navigation Graph</code> (New XML resource)	如我们的第一张图所示，这是一个新的资源文件，用户在可视化界面可以看出他能够到达的 <code>Destination</code> （用户能够到达的屏幕界面），以及流程关系。
<code>NavHostFragment</code> (Layout XML view)	当前 <code>Fragment</code> 的容器
<code>NavController</code> (Kotlin/Java object)	导航的控制者

可能我这么解释还是有点抽象，做一个不是那么恰当的比喻，我们可以将`Navigation Graph`看作一个地图，`NavHostFragment`看作一个车，以及把`NavController`看作车中的方向盘，`Navigation Graph`中可以看出各个地点（`Destination`）和通往各个地点的路径，`NavHostFragment`可以到达地图中的各个目的地，但是决定到什么目的地还是方向盘`NavController`，虽然它取决于开车人（用户）。

2.3.1 第一步 添加依赖

模块层的`build.gradle`文件需要添加：

```
ext.navigationVersion = "2.0.0"
dependencies {
    //...
    implementation "androidx.navigation:navigation-fragment-
ktx:$rootProject.navigationVersion"
    implementation "androidx.navigation:navigation-ui-ktx:$rootProject.navigationVersion"
}
```

如果你要使用 safeArgs 插件，还要在项目目录下的 `build.gradle` 文件添加：

```
buildscript {
    ext.navigationVersion = "2.0.0"
    dependencies {
        classpath "androidx.navigation:navigation-safe-args-gradle-
plugin:$navigationVersion"
    }
}
```

以及模块下面的 `build.gradle` 文件添加：

```
apply plugin: 'kotlin-android-extensions'
apply plugin: 'androidx.navigation.safeargs'
```

2.3.2 第二步 创建navigation导航

1. 创建基础目录：资源文件 `res` 目录下创建 `navigation` 目录 -> 右击 `navigation` 目录 New一个 `Navigation resource file`

2. 创建一个

`Destination`

，如果说

`navigation`

是我们的导航工具，

`Destination`

是我们的目的地，在此之前，我已经写好了一个

`WelcomeFragment`

、

`LoginFragment`

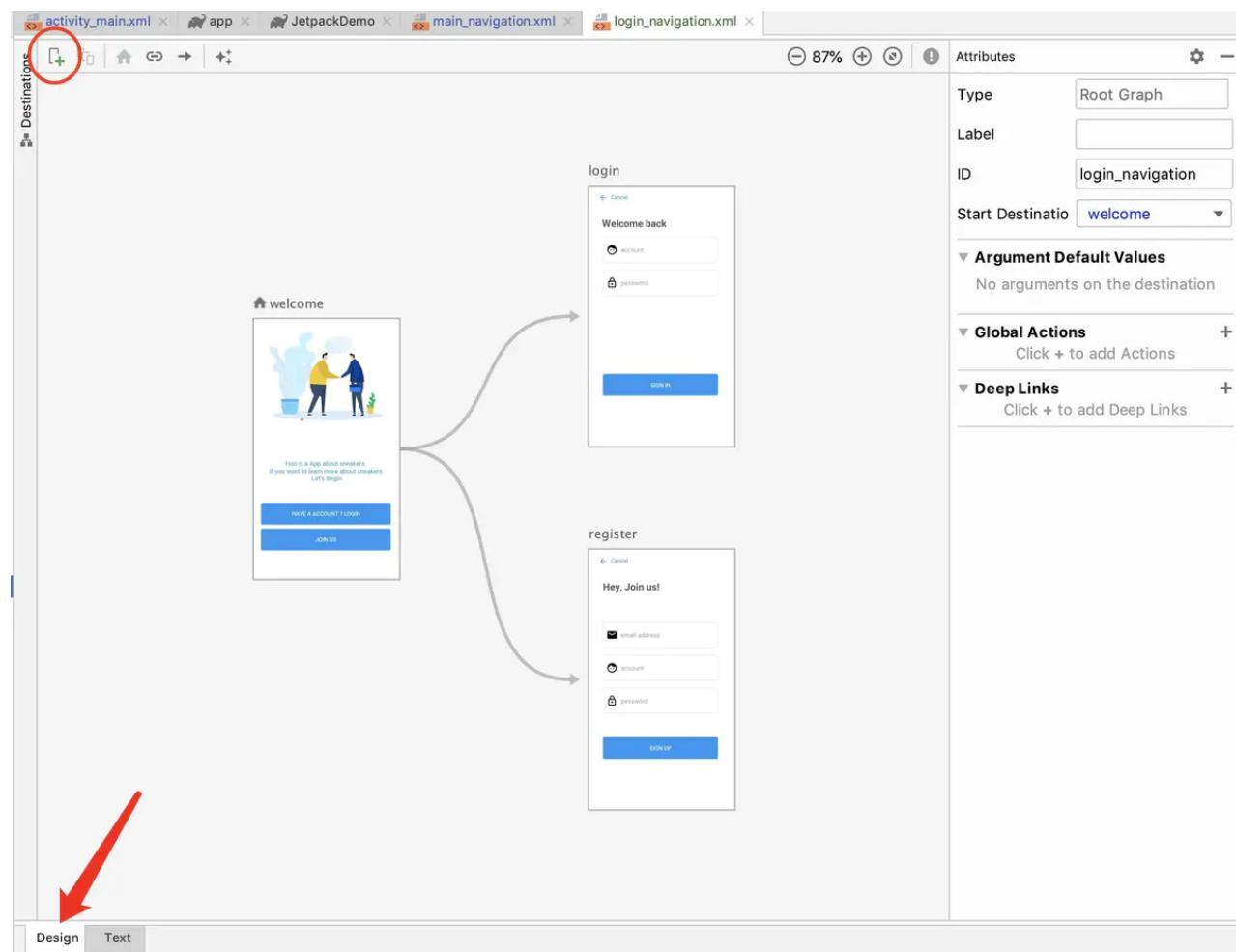
和

RegisterFragment

, 添加

Destination

的操作完成后如下所示：



添加Destination

除了可视化界面之外，我们仍然有必要看一下里面的内容组成，`login_navigation.xml`：

```
<navigation>
    ...
    android:id="@+id/login_navigation"
    app:startDestination="@+id/welcome">

    <fragment
        android:id="@+id/login"
        android:name="com.joe.jetpackdemo.ui.fragment.login.LoginFragment"
        android:label="LoginFragment"
```

```

        tools:layout="@layout/fragment_login"
    />

<fragment
    android:id="@+id/welcome"
    android:name="com.joe.jetpackdemo.ui.fragment.login.WelcomeFragment"
    android:label="LoginFragment"
    tools:layout="@layout/fragment_welcome">
    <action
        .../>
    <action
        .../>
</fragment>

<fragment
    android:id="@+id/register"
    android:name="com.joe.jetpackdemo.ui.fragment.login.RegisterFragment"
    android:label="LoginFragment"
    tools:layout="@layout/fragment_register"
    >

    <argument
        .../>
</fragment>
</navigation>

```

我在这里省略了一些不必要的代码。让我们看一下 `navigation` 标签的属性：

属性	解释
<code>app:startDestination</code>	默认的起始位置

2.3.3 第三步 建立 NavHostFragment

我们创建一个新的 `LoginActivity`，在 `activity_login.xml` 文件中：

```

<androidx.constraintlayout.widget.ConstraintLayout
    ...>

    <fragment
        android:id="@+id/my_nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        app:navGraph="@navigation/login_navigation"
        app:defaultNavHost="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>

```

有几个属性需要解释一下：

属性	解释
android:name	值必须是 <code>androidx.navigation.fragment.NavHostFragment</code> , 声明这是一个 <code>NavHostFragment</code>
app:navGraph	存放的是第二步建好导航的资源文件，也就是确定了 <code>Navigation Graph</code>
app:defaultNavHost="true"	与系统的返回按钮相关联

2.3.4 第四步 界面跳转、参数传递和动画

在 `WelcomeFragment` 中，点击登录和注册按钮可以分别跳转到 `LoginFragment` 和 `RegisterFragment` 中。



Hoo is a App about sneakers.
If you want to learn more about sneakers
Let's Begin



WelcomeFragment.

这里我使用了两种方式实现：

方式一 利用ID导航

目标：`welcomeFragment` 携带 key 为 `name` 的数据跳转到 `LoginFragment`，`LoginFragment` 接收后显示。 Have a account ? Login 按钮的点击事件如下：

```
btnLogin.setOnClickListener {
    // 设置动画参数
    val navOption = navOptions {
        anim {
            enter = R.anim.slide_in_right
            exit = R.anim.slide_out_left
            popEnter = R.anim.slide_in_left
            popExit = R.anim.slide_out_right
        }
    }
    // 参数设置
    val bundle = Bundle()
    bundle.putString("name", "Teaof")
    findNavController().navigate(R.id.login, bundle, navOption)
}
```

后续 `LoginFragment` 的接收代码比较简单，直接获取 `Fragment` 中的 `Bundle` 即可，这里不再出示代码。最后的效果：



LoginFragment

方式二 利用 Safe Args

目标：`WelcomeFragment` 通过 `Safe Args` 将数据传到 `RegisterFragment`，`RegisterFragment` 接收后显示。再看一下已经展示过的 `login_navigation.xml`：

```
<navigation
    ...
    <fragment
        ...
        />

    <fragment
        android:id="@+id/welcome"
        >
        <action
            android:id="@+id/action_welcome_to_login"
```

```

        app:destination="@+id/login"/>
    <action
        android:id="@+id/action_welcome_to_register"
        app:enterAnim="@anim/slide_in_right"
        app:exitAnim="@anim/slide_out_left"
        app:popEnterAnim="@anim/slide_in_left"
        app:popExitAnim="@anim/slide_out_right"
        app:destination="@+id/register"/>
    </fragment>

    <fragment
        android:id="@+id/register"
        ...
        >

        <argument
            android:name="EMAIL"
            android:defaultValue="2005@qq.com"
            app:argType="string"/>
    </fragment>
</navigation>

```

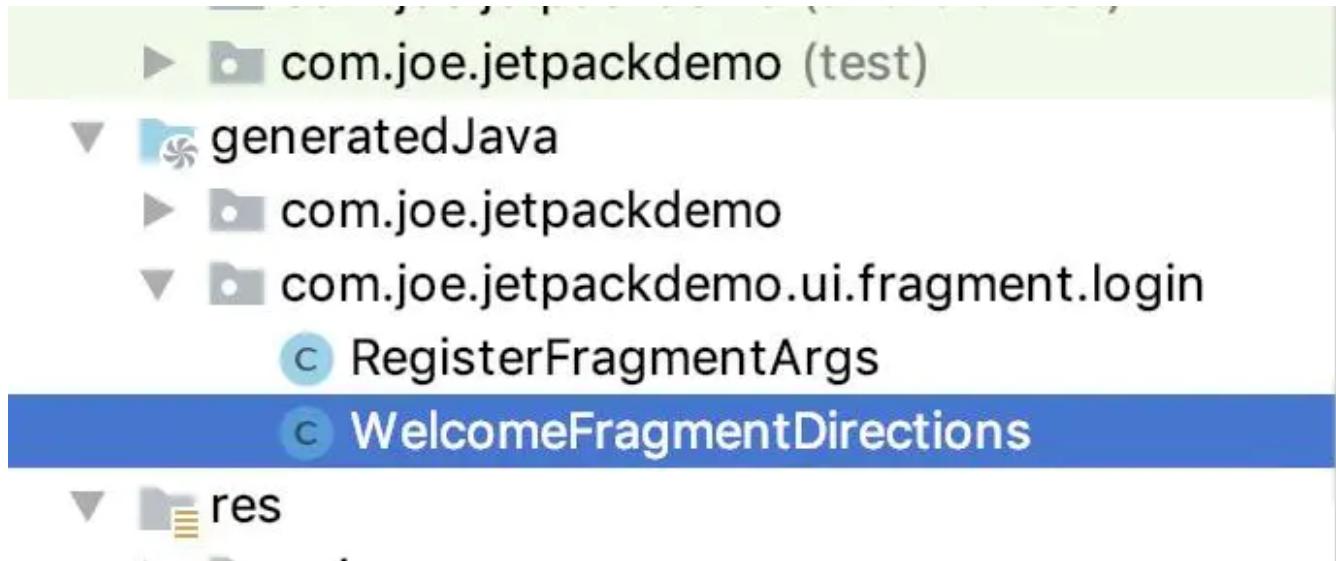
细心的同学可能已经观察到 `navigation` 目录下的 `login_navigation.xml` 资源文件中的 `action` 标签和 `argument` 标签，这里需要解释一下：**action标签**

属性	作用
<code>app:destination</code>	跳转完成到达的 <code>fragment</code> 的Id
<code>app:popUpTo</code>	将 <code>fragment</code> 从 栈 中弹出，直到某个Id的 <code>fragment</code>

argument标签

属性	作用
<code>android:name</code>	标签名字
<code>app:argType</code>	标签的类型
<code>android:defaultValue</code>	默认值

点击Android studio中的Make Project按钮，可以发现系统为我们生成了两个类：



系统生成的类

WelcomeFragment

中的

JOIN US

按钮点击事件：

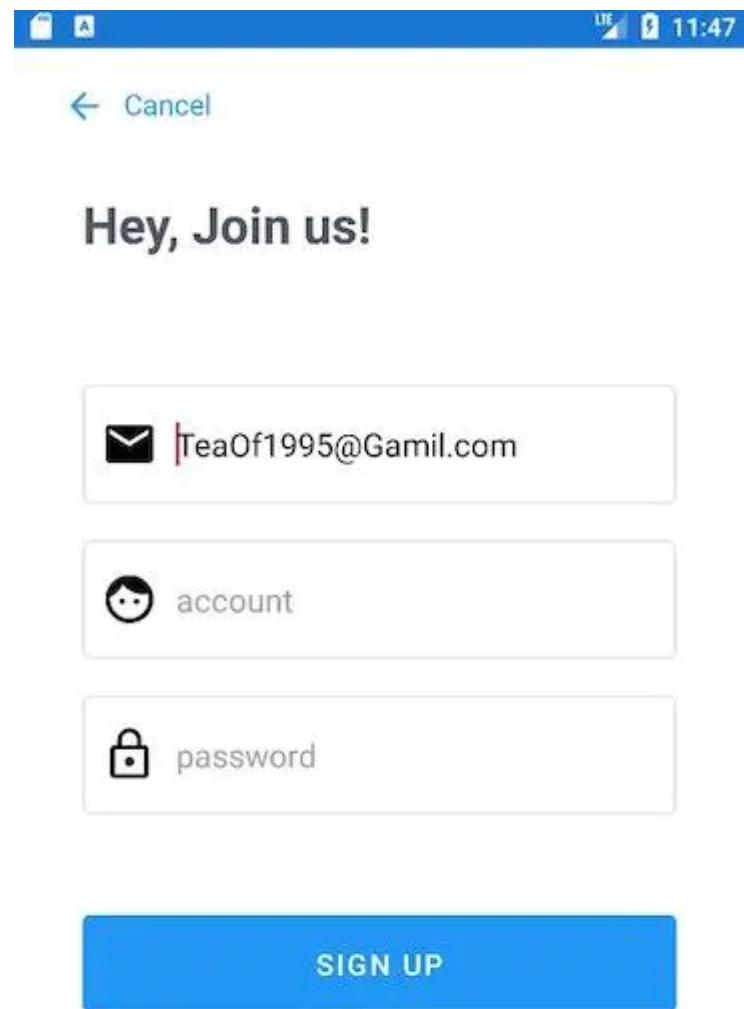
```
btnRegister.setOnClickListener {
    val action = WelcomeFragmentDirections
        .actionWelcomeToRegister()
        .setEMAIL("Teaoft1995@Gamil.com")
    findNavController().navigate(action)
}
```

RegisterFragment 中的接收：

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    // ...
    val safeArgs: RegisterFragmentArgs by navArgs()
    val email = safeArgs.email
    mEmailEt.setText(email)
}
```

以及效果：



RegisterFragment

需要提及的是，如果不

Safe Args

,

action

可以由

```
Navigation.createNavigateOnClickListener(R.id.next_action, null)
```

方式生成，感兴趣的同学可以自行编写。

2.4 更多

Navigation 可以绑定 menus、drawers 和 bottom navigation，这里我们以 bottom navigation 为例，我先在 navigation 目录下新创建了 main_navigation.xml，接着新建了 MainActivity，下面则是 activity_main.xml：

```
<LinearLayout
    ...
    <fragment
        android:id="@+id/my_nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        app:navGraph="@navigation/main_navigation"
        app:defaultNavHost="true"
        android:layout_height="0dp"
        android:layout_weight="1"/>

    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/navigation_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/white"
        app:itemIconTint="@color/colorAccent"
        app:itemTextColor="@color/colorPrimary"
        app:menu="@menu/menu_main"/>

</LinearLayout>
```

MainActivity 中的处理也十分简单：

```
class MainActivity : AppCompatActivity() {

    lateinit var bottomNavigationView: BottomNavigationView

    override fun onCreate(savedInstanceState: Bundle?) {
        //...
        val host: NavHostFragment =
            supportFragmentManager.findFragmentById(R.id.my_nav_host_fragment) as NavHostFragment
        val navController = host.navController
        initWidget()
        initBottomNavigationView(bottomNavigationView, navController)
    }

    private fun initBottomNavigationView(bottomNavigationView: BottomNavigationView,
                                       navController: NavController) {
        bottomNavigationView.setupWithNavController(navController)
    }

    private fun initWidget() {
        bottomNavigationView = findViewById(R.id.navigation_view)
```

```
    }  
}
```

效果：



Info



主页



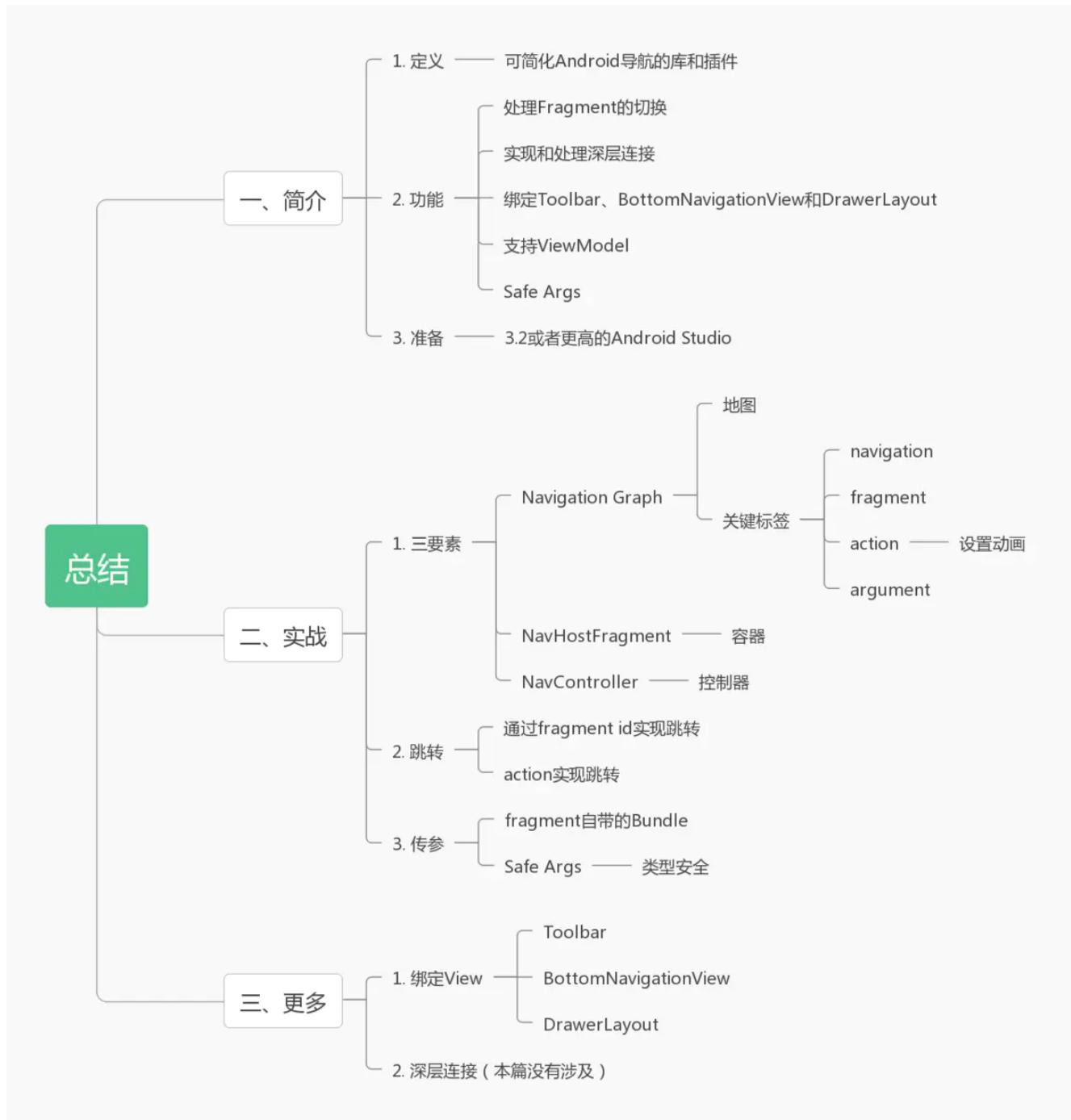
市场



我

MainActivity

2.5 总结



上图概括了本文的一些知识点，当然还有一些知识点没有涉及，比如

深层连接

等，其次，本文只是一篇入门型博客，关于更深层次的学习，本人会逐步进行。本人水平有限，文章难免有误，欢迎指正。

3. Android Jetpack - Data Binding

3.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第二篇。

Google在2018年推出Android Jetpack，本人最近在学习Android Jetpack，如果你有研究过Android Jetpack，你会发现Livedata，ViewModel和Livecycles等一系列Android Jetpack组件非常适用于实现MVVM，因此，在进行Android Jetpack的下一步研究之前，我们有必要学习一下MVVM设计模式以及Android中实现MVVM的Data Binding组件。

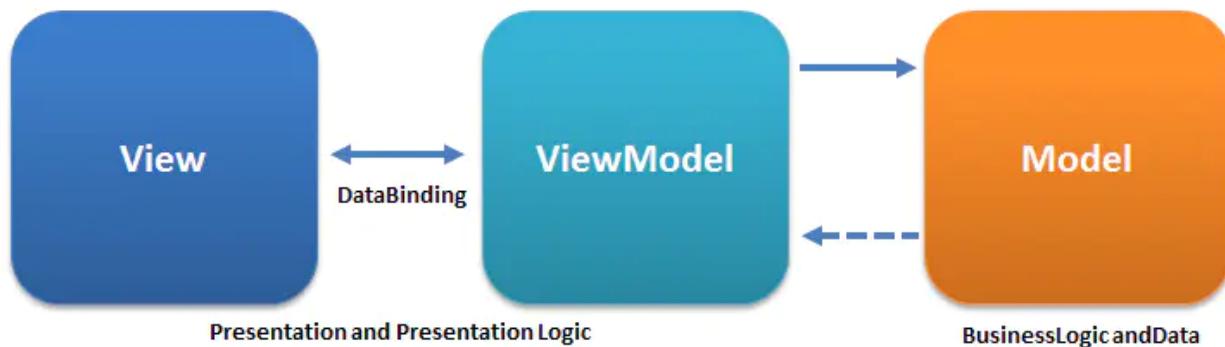
语言：kotlin 我的Demo：<https://github.com/mCyp/Hoo>

3.2 介绍

3.2.1 MVVM介绍

MVVM（全称Model-View-ViewModel）同MVC和MVP一样，是逻辑分层解偶的模式（如果你还不了解MVC和MVP，建议还是提前了解一下）。

结构图



MVVM结构图

从上图我们可以了解到MVVM的三要素，他们分别是：

- View层：xml、Activity、Fragment、Adapter和View等
- Model层：数据源（本地数据和网络数据等）
- ViewModel层：View层处理数据以及逻辑处理

3.2.2 Data Binding介绍

Data Binding不算特别新的东西，2015年Google就推出了，但即便是现在，很多人都没有学习过它，我就是这些工程师中的一位，因为我觉得MVP已经足够帮我处理日常的业务，Android Jetpack的出现，是我研究Data Binding的一个契机。

在进行下文之前，我有必要声明一下，MVVM和Data Binding是两个不同的概念，MVVM是一种架构模式，而Data Binding是一个实现数据和UI绑定的框架，是构建MVVM模式的一个工具。

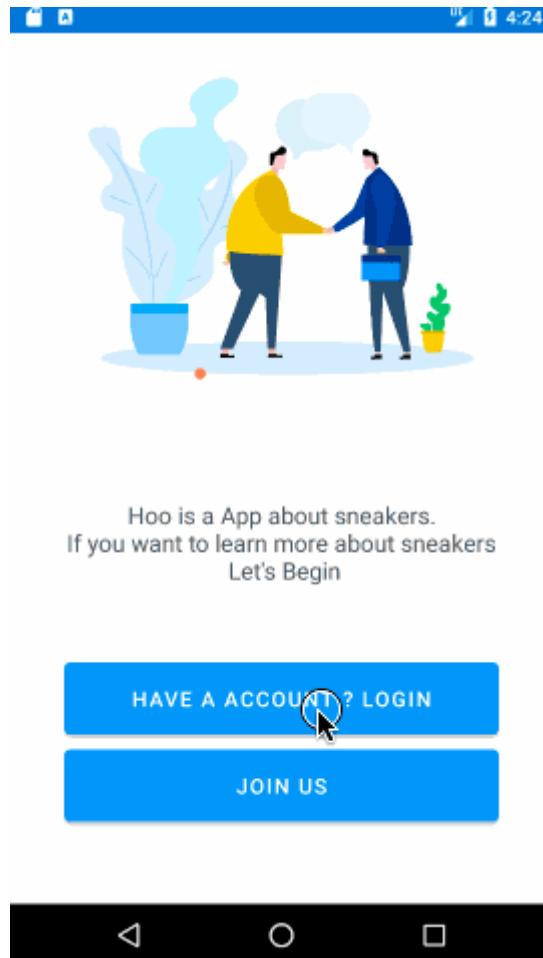
学习姿势

我依然认为官方文档是最好的学习途径：官方文档：[Data Binding Library](#) 谷歌实验室：[官方教程](#) 官方Demo地址：<https://github.com/googlecodelabs/android-databinding>

3.3 实战

在这里，我打算先在上一节[即学即用Android Jetpack - Navigation](#)的基础代码上进行拓展（如有涉及到 Navigation 的代码，我会注明），本文会在登录和注册模块的基础上进行讲解，后期如有需要，会拓展到其他模块。

效果图，和之前的有点不一样：



Data Binding

3.3.1 第一步 在app模块下的 build.gradle 文件添加内容

```
android {  
    ...  
    dataBinding {  
        enabled true  
    }  
}
```

3.3.2 第二步 构建LoginModel

创建登录的 `LoginModel`，`LoginModel` 主要负责登录逻辑的处理以及两个输入框内容改变的时候数据更新的处理：

```
class LoginModel constructor(name: String, pwd: String, context: Context) {  
    val n = ObservableField<String>(name)
```

```

val p = ObservableField<String>(pwd)
var context: Context = context

/**
 * 用户名改变回调的函数
 */
fun onNameChanged(s: CharSequence) {
    n.set(s.toString())
}

/**
 * 密码改变的回调函数
 */
fun onPwdChanged(s: CharSequence, start: Int, before: Int, count: Int) {
    p.set(s.toString())
}

fun login() {
    if (n.get().equals(BaseConstant.USER_NAME)
        && p.get().equals(BaseConstant.USER_PWD))
    ) {
        Toast.makeText(context, "账号密码正确", Toast.LENGTH_SHORT).show()
        val intent = Intent(context, MainActivity::class.java)
        context.startActivity(intent)
    }
}
}

```

我相信同学们可能会对 `ObservableField` 存在疑惑，那么 `ObservableField` 是什么呢？它其实是一个可观察的域，通过泛型来使用，可以使用的方法也就三个：

方法	作用
<code>ObservableField(T value)</code>	构造函数，设置可观察的域
<code>T get()</code>	获取可观察的域的内容，可以使用UI控件监测它的值
<code>set(T value)</code>	设置可观察的域，设置成功之后，会通知UI控件进行更新

不过，除了使用 `ObservableField` 之外，`Data Binding` 为我们提供了基本类型的 `Observablexxx` (如 `ObservableInt`) 和存放容器的 `Observablexxx` (如 `ObservableList<T>`) 等，同样，如果你想让你自定义的类变成可观察状态，需要实现 `Observable` 接口。

我们再回头看看 `LoginModel` 这个类，它其实只有分别用来观察 `name` 和 `pwd` 的成员变量 `n` 和 `p`，外加一个处理登录逻辑的方法，非常简单。

3.3.3 第三步 创建布局文件

引入 `Data Binding` 之后的布局文件的使用方式会和以前的布局使用方式有很大的不同，且听我一一解释：

标签名	作用
layout	用作布局的根节点，只能包裹一个View标签，且不能包裹merge标签。
data	Data Binding的数据，只能存在一个data标签。
variable	data 中使用，数据的变量标签，type 属性指明变量的类，如 com.joe.jetpackdemo.viewmodel.LoginModel。name 属性指明变量的名字，方便布局中使用。
import	data 中使用，需要使用静态方法和静态常量，如需要使用View.Visible属性的时候，则需导入<import type="android.view.View"/>。type 属性指明类的路径，如果两个 import 标签导入的类名相同，则可以使用 alias 属性声明别名，使用的时候直接使用别名即可。
include	View标签中使用，作用同普通布局中的 include一样，需要使用 bind:<参数名> 传递参数

我们再看一下 LoginFragment 下的 fragment_login.xml 布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <!--需要的viewModel，通过mBinding.vm=mViewModel注入-->
        <variable
            name="model"
            type="com.joe.jetpackdemo.viewmodel.LoginModel"/>

        <variable
            name="activity"
            type="androidx.fragment.app.FragmentActivity"/>
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/txt_cancel"
            android:onClick="@{() -> activity.onBackPressed()}"
            />

        <TextView
            android:id="@+id/txt_title"
            app:layout_constraintTop_toTopOf="parent"
            .../>

        <EditText
            android:id="@+id/et_account"
            android:text="@{model.n.get()}"
            android:onTextChanged="@{(text, start, before, count) ->
                model.n.set(text.toString())
            }"/>
    

```

```

>model.onNameChanged(text)"  

    ...  

    />  
  

<EditText  

    android:id="@+id/et_pwd"  

    android:text="@{model.p.get()}"  

    android:onTextChanged="@{model::onPwdChanged}"  

    ...  

    />  
  

<Button  

    android:id="@+id	btn_login"  

    android:text="Sign in"  

    android:onClick="@{() -> model.login()}"  

    android:enabled="@{(model.p.get().isEmpty() || model.n.get().isEmpty()) ? false :  

true}"  

    .../>  
  

</androidx.constraintlayout.widget.ConstraintLayout>  

</layout>

```

`variable` 有两个：

- `model`：类型为 `com.joe.jetpackdemo.viewmodel.LoginModel`，绑定用户名详见 `et_account` `EditText` 中的 `android:text="@{model.n.get()}"`，当 `EditText` 输入框内容变化的时候有如下处理 `android:onTextChanged="@{(text, start, before, count)->model.onNameChanged(text)}"`，以及登录按钮处理 `android:onClick="@{() -> model.login()}"`。
- `activity`：类型为 `androidx.fragment.app.FragmentActivity`，主要用来返回按钮的事件处理，详见 `txt_cancel` `TextView` 的 `android:onClick="@{() -> activity.onBackPressed()}"`。

对于以上的内容，我仍然有知识点需要讲解：

1. 属性的引用

如果想使用 `ViewModel` 中成员变量，如直接使用 `model.p`。

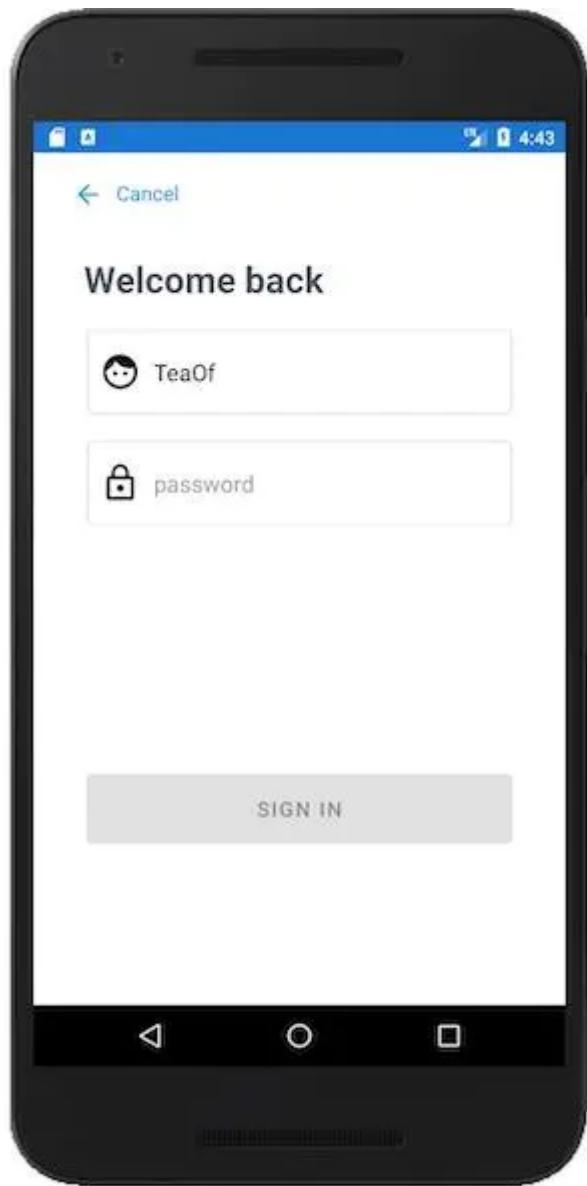
2. 事件绑定

事件绑定包括 `方法引用` 和 `监听绑定`：

- `方法引用`：参数类型和返回类型要一致，参考 `et_pwd` `EditText` 的 `android:onTextChanged` 引用。
- `监听绑定`：相比较于 `方法引用`，`监听绑定` 的要求就没那么高了，我们可以使用自行定义的函数，参考 `et_account` `EditText` 的 `android:onTextChanged` 引用。

3. 表达式

如果你注意到了 `btn_login` `Button` 在密码没有内容的时候是灰色的：



LoginFragment

是因为它在 `android:enabled` 使用了表达式：`@{model.p.get().isEmpty() || model.n.get().isEmpty()} ? false : true}`，它的意思是用户名和密码为空的时候登录的 `enable` 属性为 `false`，这是普通的三元表达式，除了上述的 `||` 和三元表达式之外，`Data Binding` 还支持：

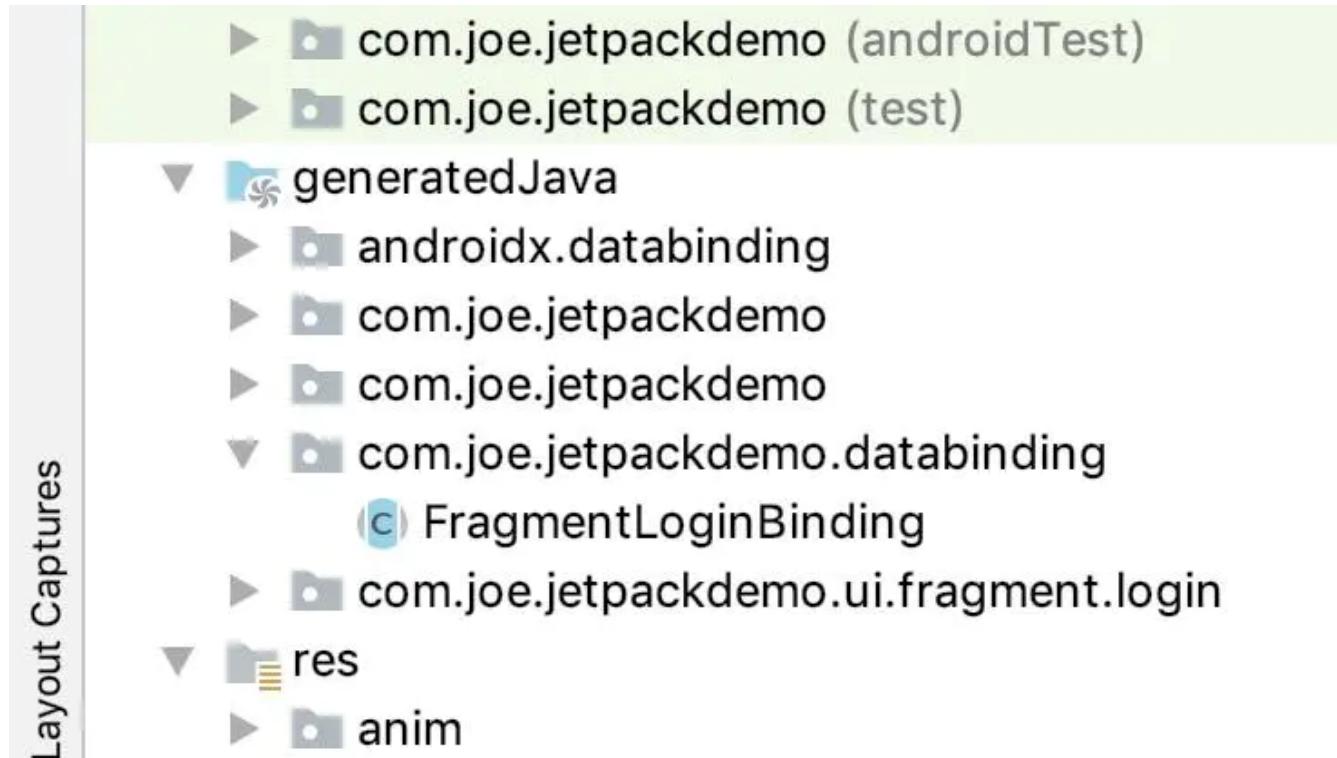
- 运算符 + - * %
- 字符串连接 +
- 逻辑与或 && ||
- 二进制 & | ^
- 一元 + - ! ~
- 移位 >> >>> <<
- 比较 == > < >= <= (Note that < needs to be escaped as <)
- instanceof
- Grouping ()
- Literals - character, String, numeric, null
- Cast

- 方法调用
- 域访问
- 数组访问
- 三元操作符

除了上述之外，Data Binding 新增了空合并操作符 ??，例如 `android:text="@{user.displayName ?? user.lastName}"`，它等价于 `android:text="@{user.displayName != null ? user.displayName : user.lastName}"`。

3.3.4 第四步 生成绑定类

我们的布局文件创建完毕之后，点击 Build 下面的 Make Project，让系统帮我生成绑定类，生成绑定的类如下：



生成的绑定类

下面我们只需在

LoginFragment

完成绑定即可，绑定操作既可以使用上述生成的

FragmentLoginBinding

也可以使用自带的

DataBindingUtil

完成：

1. 使用 DataBindingUtil

我们可以看一下 `DataBindingUtil` 的一些常用 API :

函数名	作用
<code>setContentView</code>	用来进行 Activity 下面的绑定
<code>inflate</code>	用来进行 Fragment 下面的绑定
<code>bind</code>	用来进行 View 的绑定

`LoginFragment` 绑定代码如下：

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val binding: FragmentLoginBinding = DataBindingUtil.inflate(
        inflater
        , R.layout.fragment_login
        , container
        , false
    )
    loginModel = LoginModel("", "", context!!)
    binding.model = loginModel
    binding.activity = activity
    return binding.root
}
```

2. 使用生成的 `FragmentLoginBinding`

使用方法与第一种类似，仅需将生成方式改成 `val binding = FragmentLoginBinding.inflate(inflater, container, false)` 即可

运行一下代码，开始图的效果就出现了。

3.4 更多

`Data Binding` 还有一些有趣的功能，为了让同学们了解到更多的知识，我们在这里有必要探讨一下：

3.4.1 布局中属性的设置

有属性有setter的情况

如果 `XXXView` 类有成员变量 `borderColor`，并且 `XXXView` 类有 `setBoderColor(int color)` 方法，那么在布局中我们就可以借助 `Data Binding` 直接使用 `app:borderColor` 这个属性，不太明白？没关系，以 `DrawerLayout` 为例，`DrawerLayout` 没有声明 `app:scrimColor`、`app:drawerListener`，但是 `DrawerLayout` 有 `mScrimColor:int`、`mListener:DrawerListener` 这两个成员变量并且具有这两个属性的 `setter` 的方法，他就可以直接使用 `app:scrimColor`、`app:drawerListener` 这两个属性，代码如下：

```
<android.support.v4.widget.DrawerLayout  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:scrimColor="@{@color/scrim}"  
    app:drawerListener="@{fragment.drawerListener}">
```

没有setter但是有相关方法

还用XXXView为例，它有成员变量 bordercolor，这次设置 bordercolor 的方法是 setBColor(总有程序员乱写方法名~)，强行用 app:bordercolor 显然是行不通的，可以这样用的前提是必须有 setBoderColor(int color) 方法，显然 setBColor 不匹配，但我们可以使用 BindingMethods 注解实现 app:bordercolor 的使用，代码如下：

```
@BindingMethods(value = [  
    BindingMethod(  
        type = 包名.XXXView::class,  
        attribute = "app:borderColor",  
        method = "setBColor")])
```

自定义属性

这次不仅没 setter 方法，甚至连成员变量都需要自带（条件越来越刻苦~），这次我们的目标就是给EditText添加文本监听器，先在 LoginModel 中自定义一个监听器并使用 @BindingAdapter 注解：

```
// Simplewatcher 是简化了的TextWatcher  
val namewatcher = object : Simplewatcher() {  
    override fun afterTextChanged(s: Editable) {  
        super.afterTextChanged(s)  
  
        n.set(s.toString())  
    }  
}  
  
@BindingAdapter("addTextChangedListener")  
fun addTextChangedListener(editText: EditText, simplewatcher: Simplewatcher) {  
    editText.addTextChangedListener(simplewatcher)  
}
```

这样我们就可以在布局文件中对EditText愉快的使用 app:addTextChangedListener 属性了：

```
<EditText  
    android:id="@+id/et_account"  
    android:text="@{model.n.get()}"  
    app:addTextChangedListener="@{model.namewatcher}"  
    ...  
/>
```

效果与我们之前使用的时候一样

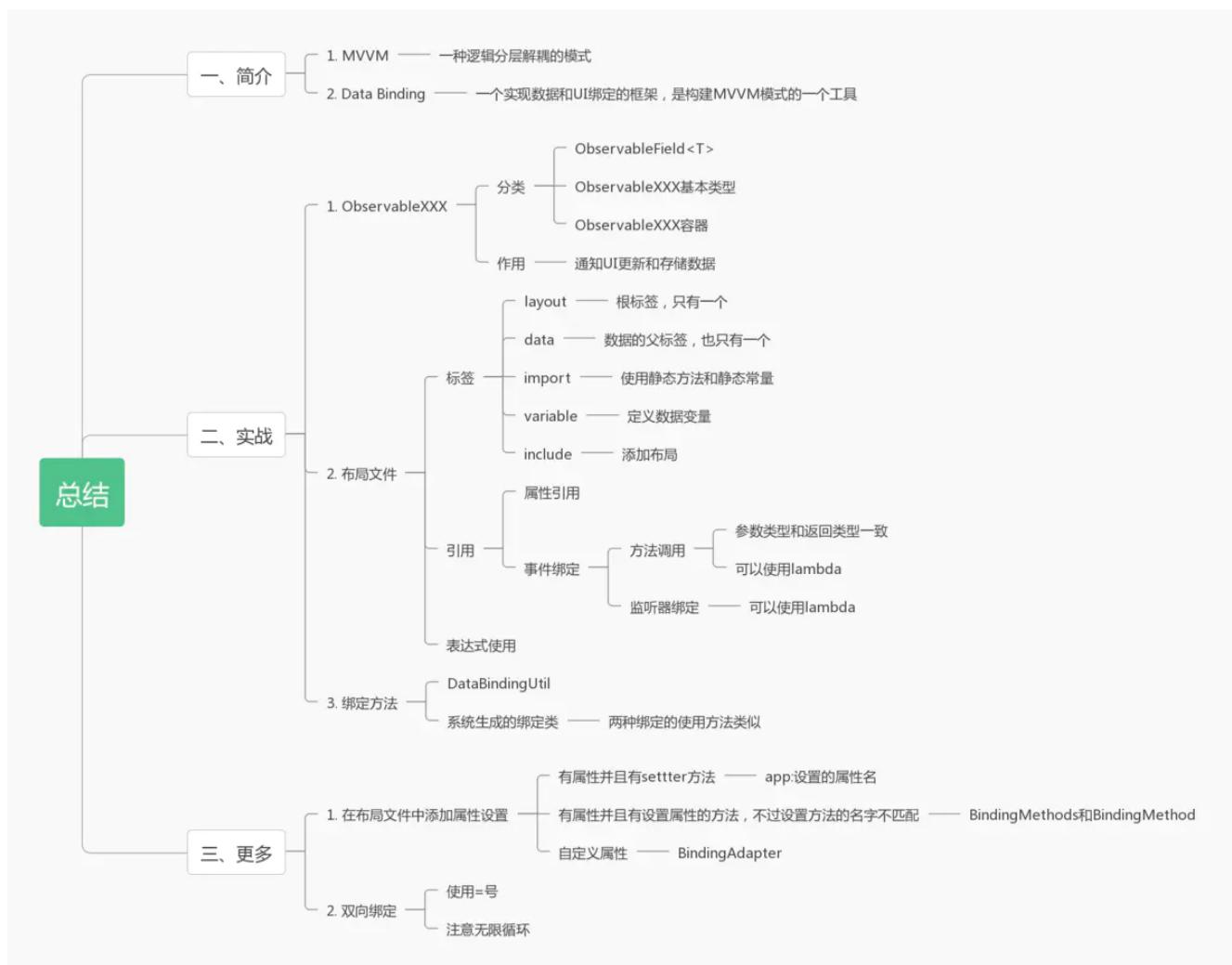
3.4.2 双向绑定

使用双向绑定可以简化我们的代码，比如我们上面的EditText在实现双向绑定之后既不需要添加 simplewatcher 也不需要用方法调用，怎么实现呢？代码如下：

```
<EditText  
    android:id="@+id/et_account"  
    android:text="@={model.n.get()}"  
    ...  
/>
```

仅仅在将 @={model.n.get()} 替换为 @={model.n.get()}，多了一个=号而已，需要注意的是，属性必须是可观察的，可以使用上面提到的 observableField，也可以自定义实现 BaseObservable 接口，双向绑定的时候需要注意无限循环，更多关于双向绑定还请查看官方文档。

3.5 总结



Data Binding总结

Data Binding

的介绍可能没有那么全面，基本使用没什么问题了，想要了解更多可以查看官方文档呦~，本人水平有限，难免理解有误差，欢迎指正。

4. Android Jetpack - ViewModel & LiveData

4.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第三篇。

在第二篇《[即学即用Android Jetpack - Data Binding](#)》，我们讨论了MVVM模式和Data Binding组件，所以这一章，我们继续学习跟MVVM模式相关的Android Jetpack组件ViewModel和LiveData。由于ViewModel和LiveData关联性比较强且使用简单（其实LiveData可以和很多组件一起使用），故打算一次性介绍这两个Android Jetpack组件。

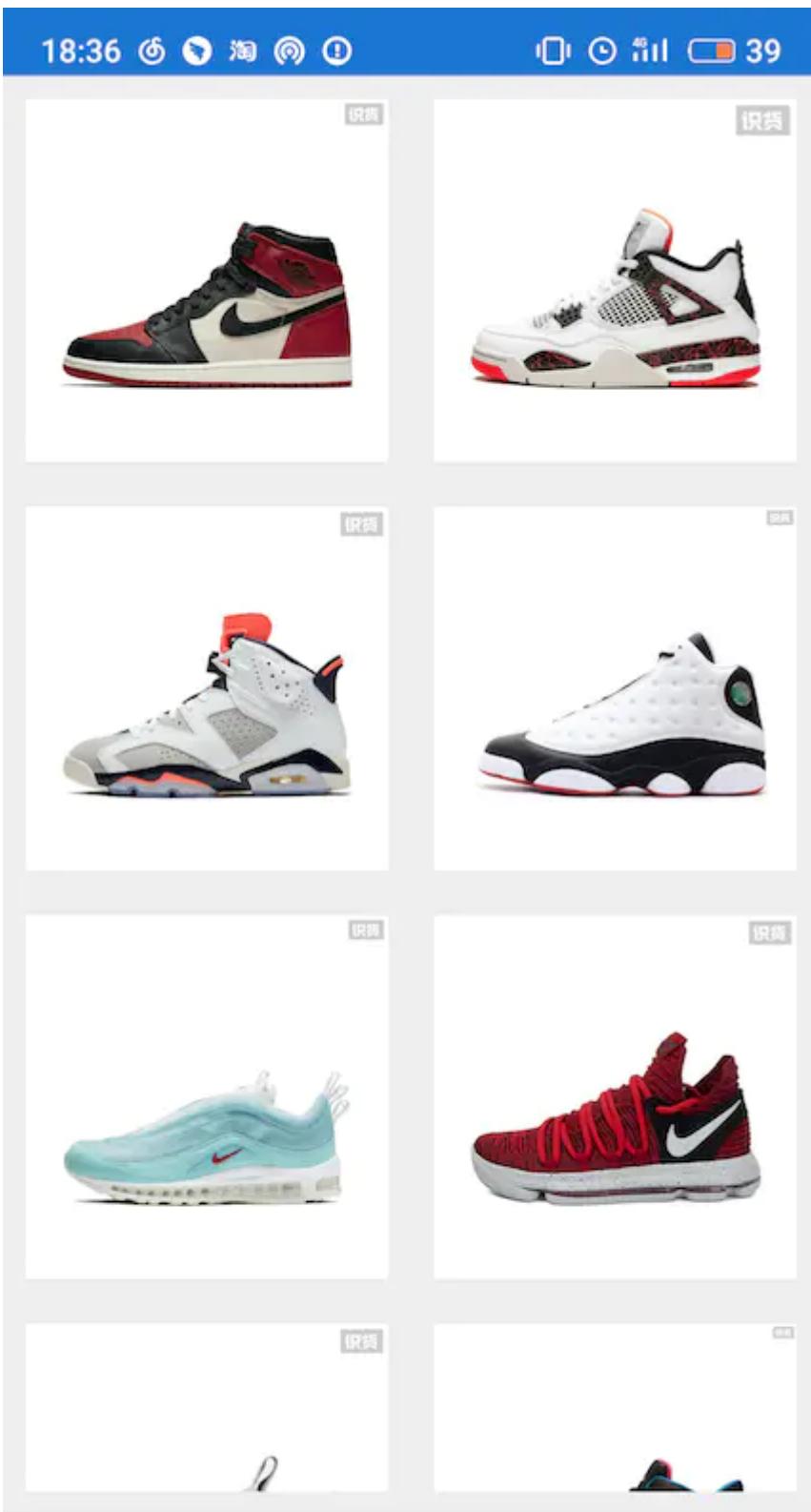
学习本文需要掌握Data Binding，如果还没掌握，建议学习：

 | [《即学即用Android Jetpack - Data Binding》](#)

本文实现的效果：

18:36 ⌂ 淘 ⌂ 39

⌚ ⌂ ⌂ 39



主页



喜欢



我

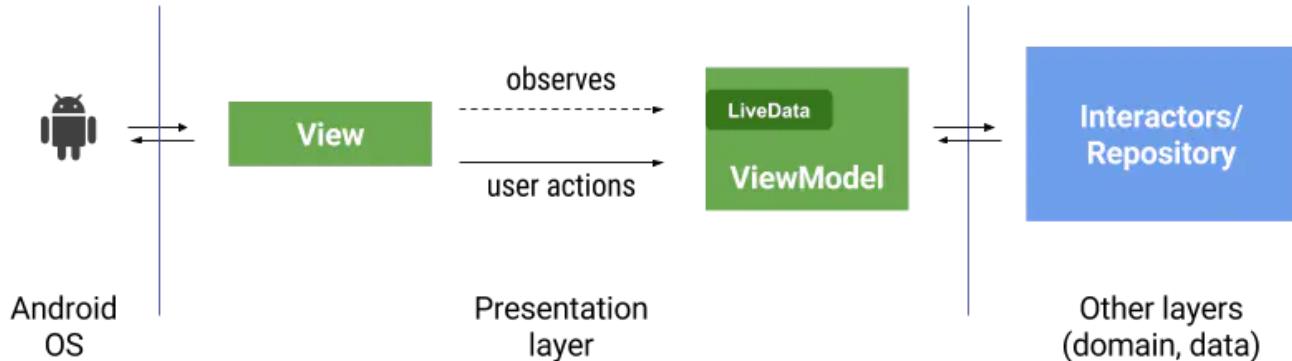
效果

语言 : `Kotlin` 我的Demo : <https://github.com/mCyp/Hoo>

4.1 LiveData

友情提醒：官方文档：[LiveData](#)

在讲 LiveData 之前，我们先看看 LiveData 和 ViewModel 的作用：



LiveData和ViewModel的作用

从这一张图，我们可以看出

ViewModel

和

LiveData

在整个MVVM架构中担当数据驱动的职责，这也是

MVVM

模式中

ViewModel层

的作用。

4.1.1 介绍

看了上面的图，对于 LiveData 我们还是感到疑惑，那么我们看看官网是如何定义的：

[LiveData](#) is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services.

从官网的介绍可以看到，LiveData 作用跟RxJava类似，是观察数据的类，相比RxJava，它能够在Activity、Fragment和Service之中正确的处理生命周期。那么 LiveData 有什么优点呢？

- 数据变更的时候更新UI

- 没有内存泄漏
- 不会因为停止Activity崩溃
- 无需手动处理生命周期
- 共享资源

乍看之下 `LiveData` 挺鸡肋的，事实也确实如此，因为 `LiveData` 能够实现的功能 `RxJava` 也可以实现，而且与 `LiveData` 相比，`RxJava` 拥有着更加丰富的生态，当然，谷歌的官方架构仍然值得我们去学习。

4.1.2 使用方式

`LiveData` 常用的方法也就如下几个：

方法名	作用
<code>observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super T> observer)</code>	最常用的方法，需要提供 <code>Observer</code> 处理数据变更后的处理。 <code>LifecycleOwner</code> 则是我们能够正确处理声明周期的关键！
<code>setValue(T value)</code>	设置数据
<code>getValue():T</code>	获取数据
<code>postValue(T value)</code>	在主线程中更新数据

4.1.3 使用场景

我看绝大部分的 `LiveData` 都是配合其他 `Android Jetpack` 组件使用的，具体情况具体分析。

- `ViewModel`：见下文。
- `Room`：先参考Demo，文章后续推出。

4.2 ViewModel

友情提醒： 官方文档：[ViewModel](#) 谷歌实验室：[教程](#) 谷歌官方Demo地址：
<https://github.com/googlegooglesamples/android-lifecycles>

众所周知，`MVVM` 层中 `ViewModel` 层用来作逻辑处理的，那么我们 `Android Jetpack` 组件中 `ViewModel` 的作用是否也一致呢？

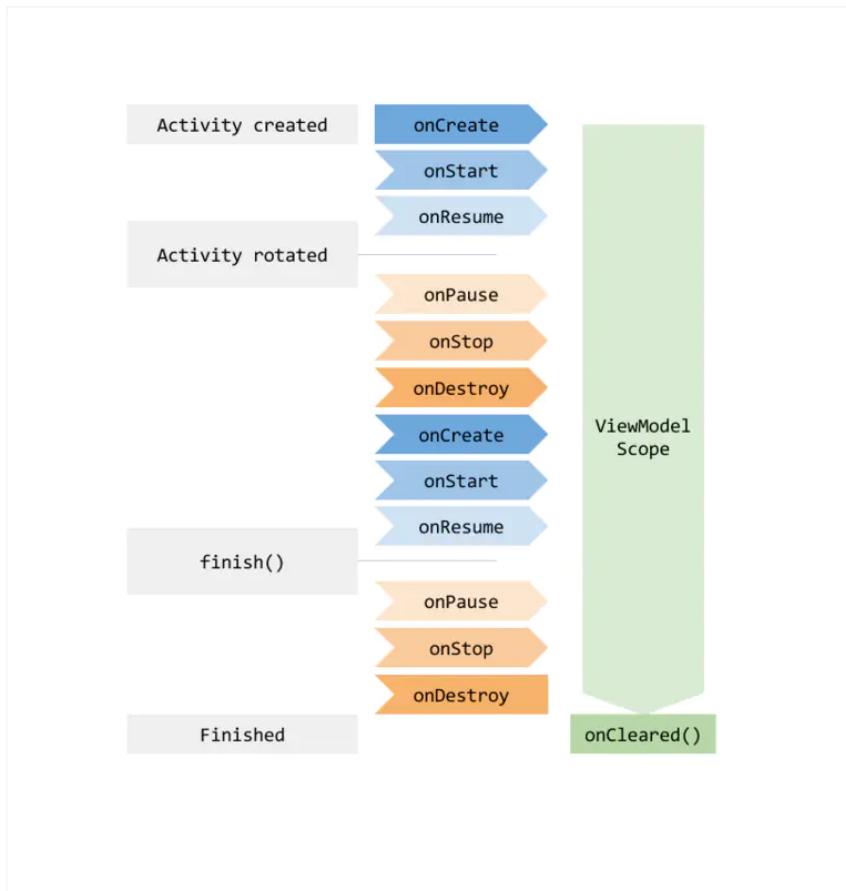
4.2.1 介绍

我们先来看官网的介绍：

The `ViewModel` class is designed to store and manage UI-related data in a lifecycle conscious way. The `ViewModel` class allows data to survive configuration changes such as screen rotations.

`ViewModel` 同样具有生命周期意识的处理跟UI相关的数据，并且，当设备的一些配置信息改变（例如屏幕旋转）它的数据不会消失。

通常情况下，如果我们不做特殊处理，当屏幕旋转的时候，数据会消失，那 `ViewModel` 管理的数据为什么不会消失呢，是因为 `ViewModel` 的生命周期：



ViewModel的生命周期

ViewModel 的另一个特点就是同一个 Activity 的 Fragment 之间可以使用ViewModel实现共享数据。

4.2.2 使用方法

继承 `ViewModel` 即可。

4.2.3 实战

第一步：添加依赖

添加进 module 下面的 `build.gradle`：

```
ext.lifecycleVersion = '2.2.0-alpha01'
dependencies {
    //...

    // LiveData
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$rootProject.lifecycleVersion"
    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-
ktx:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-extensions:$rootProject.lifecycleVersion"
}
```

第二步：创建 ShoeModel

继承 `ViewModel` 类，分别创建对品牌名的观察对象 `brand:MutableLiveData<String>` 和对鞋子集合的观察对象 `shoes: LiveData<List<Shoe>>`：

```
class ShoeModel constructor(shoeRepository: ShoeRepository) : ViewModel() {

    // 品牌的观察对象 默认观察所有的品牌
    private val brand = MutableLiveData<String>().apply {
        value = ALL
    }

    // 鞋子集合的观察类
    val shoes: LiveData<List<Shoe>> = brand.switchMap {
        // Room数据库查询，只要知道返回的是LiveData<List<Shoe>>即可
        if (it == ALL) {
            shoeRepository.getAllShoes()
        } else {
            shoeRepository.getShoesByBrand(it)
        }
    }

    //... 不重要的函数省略

    companion object {
        private const val ALL = "所有"
    }
}
```

第三步：获取ViewModel

无构造参数获取：构造函数没有参数的情况下，获取 `ShoeModel` 很简单，

`ViewModelProviders.of(this).get(ShoeModel::class.java)` 这样就可以返回一个我们需要的 `ShoeModel` 了。

有构造参数获取 不过，上面的 `ShoeModel` 中我们在构造函数中需要一个 `ShoeRepository` 参数，上述方法是显然行不通的，这种情况下我们需要自定义实现 `Factory`：

```
class ShoeModelFactory(
    private val repository: ShoeRepository
) : ViewModelProvider.NewInstanceFactory() {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return ShoeModel(repository) as T
    }
}
```

为了使用方便，又写了一个工具类 `CustomViewModelProvider`：

```
object CustomViewModelProvider {

    // ...省略无关代码

    fun providerShoeModel(context: Context): ShoeModelFactory{
        val repository: ShoeRepository = RepositoryProvider.providershoeRepository(context)
        return ShoeModelFactory(repository)
    }
}
```

最后在 `shoeFragment` 中获取：

```
// by viewModels 需要依赖 "androidx.navigation:navigation-ui-
ktx:$rootProject.navigationVersion"
private val viewModel: ShoeModel by viewModels {
    CustomViewModelProvider.providerShoeModel(requireContext())
}
```

第四步：使用ViewModel

`ViewModel` 的使用需要结合具体的业务，比如我这里的 `shoeModel`，因为 `shoeFragment` 的代码不多，我直接贴出来：

```
/**
 * 鞋子集合的Fragment
 */
class ShoeFragment : Fragment() {

    // by viewModels 需要依赖 "androidx.navigation:navigation-ui-
    ktx:$rootProject.navigationVersion"
    private val viewModel: ShoeModel by viewModels {
        CustomViewModelProvider.providerShoeModel(requireContext())
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val binding: FragmentShoeBinding = FragmentShoeBinding.inflate(inflater, container,
false)
```

```

        context ?: return binding.root
        ViewModelProviders.of(this).get(ShoeModel::class.java)
        // RecyclerView 的适配器 ShoeAdapter
        val adapter = ShoeAdapter()
        binding.recycler.adapter = adapter
        onSubscribeUi(adapter)
        return binding.root
    }

    /**
     * 鞋子数据更新的通知
     */
    private fun onSubscribeUi(adapter: ShoeAdapter) {
        viewModel.shoes.observe(viewLifecycleOwner, Observer {
            if (it != null) {
                adapter.submitList(it)
            }
        })
    }
}

```

在 `onSubscribeUi` 方法中，我们使用 `shoeModel` 的 `LiveData` 进行了观察通知，当鞋子集合更新的时候，会更新到当前 `Recyclerview` 中的适配器。

布局文件 `fragment_shoe.xml` 很简单，虽使用了 `Data Binding`，但是没有变量，且只有一个 `Recyclerview`，这里不再赘述。`ShoeAdapter` 的实现同样简单，感兴趣的可以查看源码，这里同样不再赘述。

这样写完之后，本文一开始的图的效果就出现了~

4.3 更多

一个例子并不能展现所有的关于 `LiveData` 和 `ViewModel` 的内容。`LiveData` 和 `ViewModel` 仍有一些知识需要我们注意。

4.3.1 LiveData数据变换

`LiveData` 中数据变换方法有 `map()` 和 `switchMap()`，关于 `switchMap()`，我在上面实战的 `shoeModel` 已经实践过了：

```

// 本地数据仓库
class ShoeRepository private constructor(private val shoeDao: ShoeDao) {

    fun getAllShoes() = shoeDao.getAllShoes()

    /**
     * 通过品牌查询鞋子 返回 LiveData<List<Shoe>>
     */
    fun getShoesByBrand(brand:String) = shoeDao.findShoeByBrand(brand)

    /**
     * 插入鞋子的集合 返回 LiveData<List<Shoe>>
     */
    fun insertShoes(shoes: List<Shoe>) = shoeDao.insertShoes(shoes)
}

```

```

    // ... 单例省略
}

class ShoeModel constructor(shoeRepository: ShoeRepository) : ViewModel() {

    // 品牌的观察对象 默认观察所有的品牌
    private val brand = MutableLiveData<String>().apply {
        value = ALL
    }

    // 鞋子集合的观察类
    val shoes: LiveData<List<Shoe>> = brand.switchMap {
        // Room数据库查询，只要知道返回的是LiveData<List<Shoe>>即可
        if (it == ALL) {
            shoeRepository.getAllShoes()
        } else {
            shoeRepository.getShoesByBrand(it)
        }
    }
}

```

`map()` 的使用我们借用官方的例子：

```

val userLiveData: LiveData<User> = UserLiveData()
val userName: LiveData<String> = Transformations.map(userLiveData) {
    user -> "${user.name} ${user.lastName}"
}

```

可以看到，`map()` 同样可以实现将A变成B，那么`switchMap()` 和 `map()` 的区别是什么？`map()` 中只有一个 `LiveData<A>`，他是在 `LiveData<A>` 发送数据的时候把A变成B，而 `switchMap()` 中同时存在 `LiveData<A>` 和 `LiveData`，`LiveData<A>` 更新之后通知 `LiveData` 更新。

4.3.2 LiveData如何共享数据

假设我们有这样的需求：注册页需要记录信息，注册完成跳转到登录页，并将账号和密码显示在登录页。这种情况下，我们可以定义一个类然后继承 `LiveData`，并使用单例模式即可：

```

// 登录信息
data class LoginInfo constructor(val account:String, val pwd:String, val email:String)

/**
 * 自定义单例LiveData
 */
class LoginLiveData:LiveData<LoginInfo>() {

    companion object {
        private lateinit var sInstance: LoginLiveData

        @MainThread
        fun get(): LoginLiveData {
            sInstance = if (::sInstance.isInitialized) sInstance else LoginLiveData()
        }
    }
}

```

```
        return sInstance
    }
}
}
```

需要实例的时候用单例创建即可。

4.3.3 使用ViewModel在同一个Activity中的Fragment之间共享数据

想要利用 `ViewModel` 实现Fragment之间数据共享，前提是 `Fragment` 中的 `FragmentActivity` 得相同，这里直接贴上官方的代码：

```
class SharedViewModel : ViewModel() {
    val selected = MutableLiveData<Item>()

    fun select(item: Item) {
        selected.value = item
    }
}

class MasterFragment : Fragment() {

    private lateinit var itemSelector: Selector

    private lateinit var model: SharedViewModel

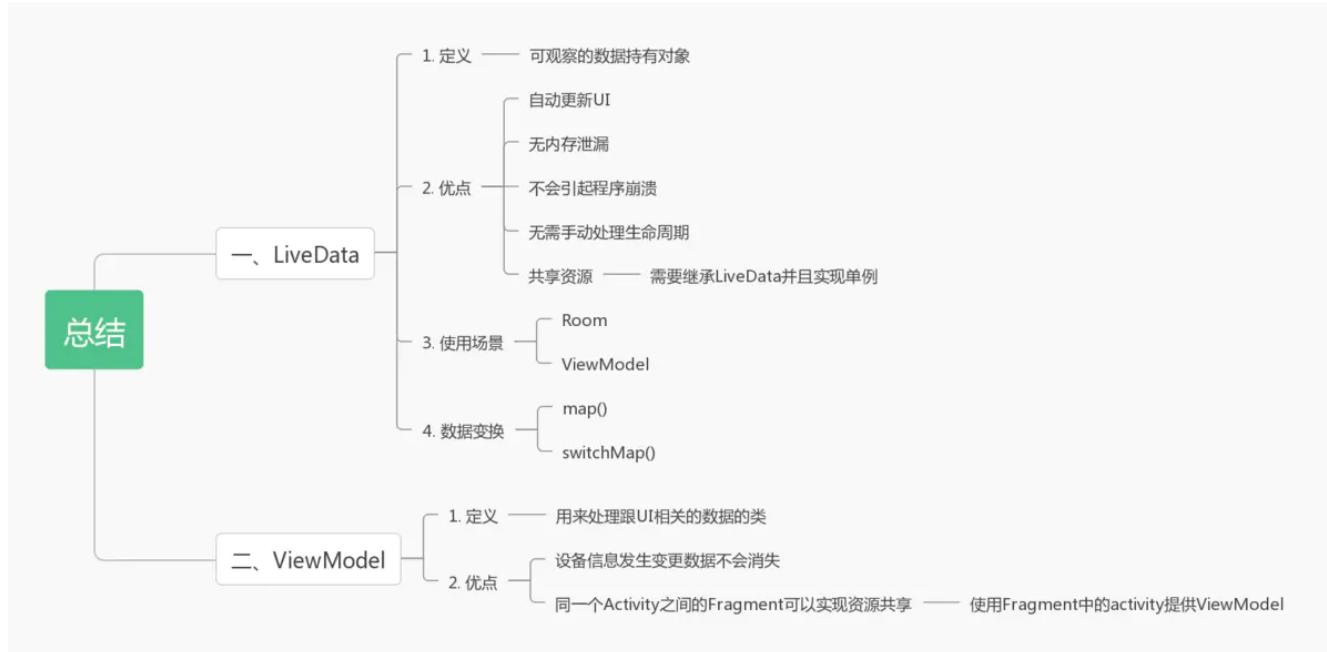
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        } ?: throw Exception("Invalid Activity")
        itemSelector.setOnClickListener { item ->
            // Update the UI
        }
    }
}

class DetailFragment : Fragment() {

    private lateinit var model: SharedViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        } ?: throw Exception("Invalid Activity")
        model.selected.observe(this, Observer<Item> { item ->
            // Update the UI
        })
    }
}
```

4.4 总结



总结

本文到此就结束了，本人水平有限，难免有误，欢迎指正哟~

5. Android Jetpack - Room

5.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第四篇。

我们在日常的工作中，免不了和数据打交道，因此，存储数据便是一项很重要的工作，在此之前，我使用过[GreenDao](#)、[DBFlow](#)等优秀的ORM数据库框架，但是，这些框架都不是谷歌官方的，现在，我们有了谷歌官方的Room数据库框架，看看它能够给我们带来什么？

语言：`Kotlin` Demo地址：<https://github.com/mCyp/Hoo>

5.2 介绍

友情提示 官方文档：[Room](#) 谷歌实验室：[官方教程](#) SQL语法：[SQLite教程](#)

谷歌官方的介绍：

The [Room](#) persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

简单来说：Room是一个基于SQLite的强大数据库框架。

5.2.1 Room优点

可是它强大在哪里呢？

- 使用编译时注解，能够对 `@Query` 和 `@Entity` 里面的SQL语句等进行验证。
- 与SQL语句的使用更加贴近，能够降低学习成本。
- 对 `RxJava 2` 的支持（大部分都Android数据库框架都支持），对 `LiveData` 的支持。
- `@Embedded` 能够减少表的创建。

5.3 实战

我们的目标结构：



目标ER图

我们的目标挺简单的，三张表，

用户表

、

鞋表

和

收藏记录表

用户表

和

鞋表

存在多对多的关系，确定好目标之后，正式开始我们的实战之旅了。

5.3.1 第一步 添加依赖

模块层的 build.gradle 添加：

```
apply plugin: 'kotlin-kapt'

dependencies {
    // ... 省略无关

    // room
    implementation "androidx.room:room-runtime:$rootProject.roomVersion"
    implementation "androidx.room:room-ktx:$rootProject.roomVersion"
    kapt "androidx.room:room-compiler:$rootProject.roomVersion"
    androidTestImplementation "androidx.room:room-testing:$rootProject.roomVersion"
}
```

项目下的 build.gradle 添加：

```
ext {
    roomVersion = '2.1.0-alpha06'
    //... 省略无关
}
```

5.3.2 第二步 创建表（实体）

这里我们以 用户表 和 收藏记录表 为例，`用户表`：

```
/**
 * 用户表
 */
@Entity(tableName = "user")
data class User(
    @ColumnInfo(name = "user_account") val account: String // 账号
    , @ColumnInfo(name = "user_pwd") val pwd: String // 密码
    , @ColumnInfo(name = "user_name") val name: String
    , @Embedded val address: Address // 地址
    , @Ignore val state: Int // 状态只是临时用，所以不需要存储在数据库中
) {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    var id: Long = 0
}
```

`收藏记录表`：

```
/**
 * 喜欢的球鞋
 */
@Entity(
```

```

        tableName = "fav_shoe"
        , foreignKeys = [ForeignKey(entity = Shoe::class, parentColumns = ["id"], childColumns =
        ["shoe_id"])
            , ForeignKey(entity = User::class, parentColumns = ["id"], childColumns =
        ["user_id"])]
        , indices = [Index("shoe_id")]
    )
data class FavouriteShoe(
    @ColumnInfo(name = "shoe_id") val shoeId: Long // 外键 鞋子的id
    , @ColumnInfo(name = "user_id") val userId: Long // 外键 用户的id
    , @ColumnInfo(name = "fav_date") val date: Date // 创建日期
) {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    var id: Long = 0
}

```

对于其中的一些注解，你可能不是很明白，解释如下：

注解	说明
@Entity	声明这是一个表（实体），主要参数： <code>tableName</code> -表名、 <code>foreignKeys</code> -外键、 <code>indices</code> -索引。
@ColumnInfo	主要用来修改在数据库中的字段名。
@PrimaryKey	声明该字段主键并可以声明是否自动创建。
@Ignore	声明某个字段只是临时用，不存储在数据库中。
@Embedded	用于嵌套，里面的字段同样会存储在数据库中。

最后一个可能解释的不明，我们直接看例子就好，如我们的 `用户表`，里面有一个变量 `address`，它是一个 `Address` 类：

```

/**
 * 地址
 */
data class Address(
    val street:String,val state:String,val city:String,val postCode:String
)

```

通常情况下，如果我们想这些字段存储在数据库中，有两种方法：

- 重新创建一个表进行一对一关联，但是多创建一个表显得麻烦。
- 在用户表中增加字段，可是使用第二种方式映射出来的对象又显得不那么面向对象。

`@Embedded` 解决了第二种方式中问题，既不需要多创建一个表，又能将数据库中映射的对象看上去面向对象。

放上 `shoe` 表，后面会用到：

```
/**
 * 鞋表
 */
@Entity(tableName = "shoe")
data class Shoe(
    @ColumnInfo(name = "shoe_name") val name: String // 鞋名
    , @ColumnInfo(name = "shoe_description") val description: String// 描述
    , @ColumnInfo(name = "shoe_price") val price: Float // 价格
    , @ColumnInfo(name = "shoe_brand") val brand: String // 品牌
    , @ColumnInfo(name = "shoe_imgUrl") val imageUrl: String // 图片地址
) {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "id")
    var id: Long = 0
}
```

5.3.3 第三步 创建Dao

有了数据库，我们现在需要建立数据处理的方法，就是数据的增删查改。如果想声明一个 `Dao`，只要在抽象类或者接口加一个 `@Dao` 注解就行。

增

`@Insert` 注解声明当前的方法为新增的方法，并且可以设置当新增冲突的时候处理的方法。

用到增的地方有很多，Demo中本地用户的注册、鞋子集合的新增和收藏的新增，这里我们选择具有代表性的 `shoeDao`：

```
/**
 * 鞋子的方法
 */
@Dao
interface ShoeDao {
    // 省略...
    // 增加一双鞋子
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertShoe(shoe: Shoe)

    // 增加多双鞋子
    // 除了List之外，也可以使用数组
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertShoes(shoes: List<Shoe>)
}
```

删

`@Delete` 注解声明当前的方法是一个删除方法。

用法与 `@Insert` 类似，同样以 `shoeDao` 为例：

```
/**
 * 鞋子的方法
 */
```

```
@Dao
interface ShoeDao {
    // 省略...
    // 删除一双鞋子
    @Delete
    fun deleteShoe(shoe: Shoe)

    // 删除多个鞋子
    // 参数也可以使用数组
    @Delete
    fun deleteShoes(shoes: List<Shoe>)
}
```

改

`@Update` 注解声明当前方法是一个更新方法

用法同样与 `@Insert` 类似：

```
/**
 * 鞋子的方法
 */
@Dao
interface ShoeDao {
    // 省略...
    // 更新一双鞋
    @Update
    fun updateShoe(shoe: Shoe)

    // 更新多双鞋
    // 参数也可以是集合
    @Update
    fun updateShoes(shoes: Array<Shoe>)
}
```

查

增删改是如此的简单，查是否也是如此的简单呢？答案是否定的，`Room` 的查很接近原生的SQL语句。`@Query` 注解不仅可以声明这是一个查询语句，也可以用来删除和修改，不可以用来新增。

简单查询 除了简单查询，这里还有如何配合 `LiveData` 和 `RxJava 2`。

```
@Dao
interface ShoeDao {

    // 查询一个
    @Query("SELECT * FROM shoe WHERE id=:id")
    fun findShoeById(id: Long): Shoe?

    // 查询多个 通过品牌查询多款鞋
    @Query("SELECT * FROM shoe WHERE shoe_brand=:brand")
    fun findShoesByBrand(brand: String): List<Shoe>
}
```

```

// 模糊查询 排序 同名鞋名查询鞋
@Query("SELECT * FROM shoe WHERE shoe_name LIKE :name ORDER BY shoe_brand ASC")
fun findShoesByName(name:String):List<Shoe>

// 配合LiveData 返回所有的鞋子
@Query("SELECT * FROM shoe")
fun getAllShoesLD(): LiveData<List<Shoe>>

// 配合LiveData 通过Id查询单款鞋子
@Query("SELECT * FROM shoe WHERE id=:id")
fun findShoeByIdLD(id: Long): LiveData<Shoe>

// 配合RxJava 通过Id查询单款鞋子
@Query("SELECT * FROM shoe WHERE id=:id")
fun findShoeByIdRx(id: Long): Flowable<Shoe>

// 省略...
}

```

查询多个的时候，可以返回 `List` 和 `数组`，还可以配合 `LiveData` 和 `RxJava 2`。当然，更多的查询可以参考SQL语法。

复合查询 因为本Demo并没有引入 `RxJava 2`，所以本文基本以 `LiveData` 为例。

```

@Dao
interface ShoeDao {
    // 省略...
    // 根据收藏结合 查询用户喜欢的鞋的集合 内联查询
    @Query(
        "SELECT
shoe.id, shoe.shoe_name, shoe.shoe_description, shoe.shoe_price, shoe.shoe_brand, shoe.shoe_imgur
1 " +
        "FROM shoe " +
        "INNER JOIN fav_shoe ON fav_shoe.shoe_id = shoe.id " +
        "WHERE fav_shoe.user_id = :userId"
    )
    fun findShoesByUserId(userId: Long): LiveData<List<Shoe>>
}

```

5.3.4 第四步 创建数据库

创建一个数据库对象是一件非常消耗资源，使用单例可以避免过多的资源消耗。

```

/**
 * 数据库文件
 */
@Database(entities = [User::class, Shoe::class, FavouriteShoe::class], version = 1, exportSchema = false)
abstract class AppDataBase: RoomDatabase() {
    // 得到UserDao
    abstract fun userDao(): UserDao
    // 得到ShoeDao
}

```

```

abstract fun shoeDao():ShoeDao
// 得到FavouriteShoeDao
abstract fun favouriteShoeDao():FavouriteShoeDao

companion object{
    @Volatile
    private var instance:AppDataBase? = null

    fun getInstance(context:Context):AppDataBase{
        return instance?: synchronized(this){
            instance?:buildDataBase(context)
                .also {
                    instance = it
                }
        }
    }

    private fun buildDataBase(context: Context):AppDataBase{
        return Room
            .databaseBuilder(context,AppDataBase::class.java,"jetPackDemo-database")
            .addCallback(object :RoomDatabase.Callback(){
                override fun onCreate(db: SupportSQLiteDatabase) {
                    super.onCreate(db)

                    // 读取鞋的集合
                    val request = OneTimeworkRequestBuilder<Shoeworker>().build()
                    workManager.getInstance(context).enqueue(request)
                }
            })
            .build()
    }
}

```

`@Database` 注解声明当前是一个数据库文件，注解中 `entities` 变量声明数据库中的表（实体），以及其他例如版本等变量。同时，获取的 Dao 也必须在数据库类中。完成之后，点击 `build` 目录下的 `make project`，系统就会自动帮我创建 `AppDataBase` 和 `xxxDao` 的实现类。

5.3.5 第五步 简要封装

这里有必要提醒一下，在不使用 `LiveData` 和 `RxJava` 的前提下，`Room` 的操作是不可以放在主线程中的。这里选择比较有示范性的 `UserRepository`：

```

/**
 * 用户处理仓库
 */
class UserRepository private constructor(private val userDao: UserDao) {
    //...

    /**
     * 登录用户 本地数据库的查询
     */

```

```

    fun login(account: String, pwd: String): LiveData<User?>
        = userDao.login(account, pwd)

    /**
     * 注册一个用户 本地数据库的新增
     */
    suspend fun register(email: String, account: String, pwd: String): Long {
        return withContext(IO) {
            userDao.insertUser(User(account, pwd, email))
        }
    }

    companion object {
        @Volatile
        private var instance: UserRepository? = null
        fun getInstance(userDao: UserDao): UserRepository =
            // ...
    }
}

```

`register()` 方法是一个普通方法，所以它需要在子线程使用，如代码所见，通过协程实现。`login()` 是配合 `LiveData` 使用的，不需要额外创建子线程，但是他的核心数据库操作还是在子线程中实现的。

现在，你就可以愉快的操作本地数据库了。

5.4 更多

除了上面的基本使用技巧之外，还有一些不常用的知识需要我们了解。

5.4.1 类型转换器

我们都已经知道，SQLite支持的类型有：NULL、INTEGER、REAL、TEXT和BLOB，对于 `Data` 类，SQLite还可以将其转化为TEXT、REAL或者INTEGER，如果是 `Calendar` 类呢？`Room` 为你提供了解决方法，使用 `@TypeConverter` 注解，我们使用谷歌官方Demo-[SunFlower](#)例子：

```

class Converters {
    @TypeConverter fun calendarToDatestamp(calendar: Calendar): Long = calendar.timeInMillis

    @TypeConverter fun datestampToCalendar(value: Long): Calendar =
        Calendar.getInstance().apply { timeInMillis = value }
}

```

然后在数据库声明的时候，加上 `@TypeConverters(Converters::class)` 就行了：

```

@Database(...)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    //...
}

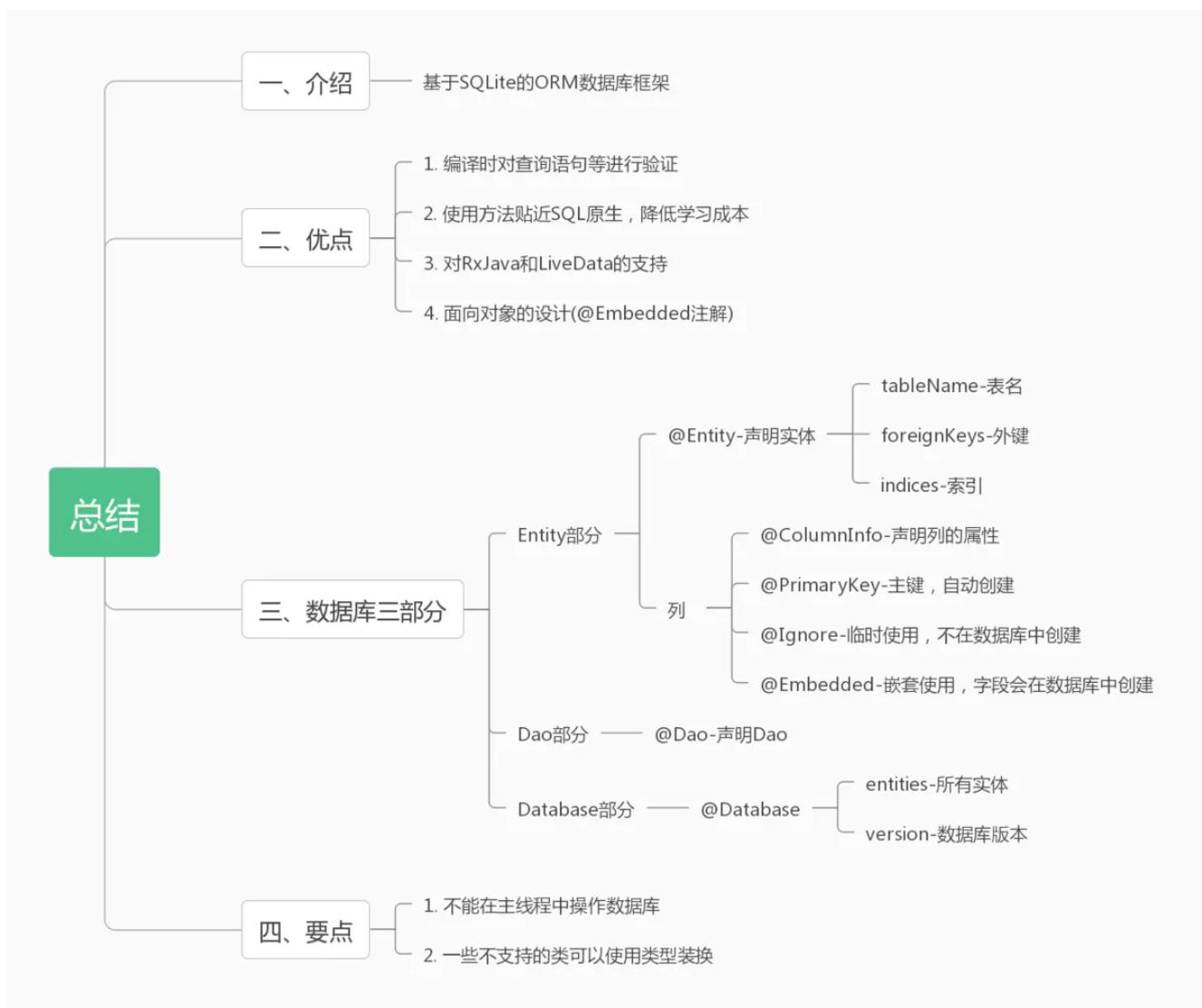
```

5.4.2 数据库迁移

Room的数据库迁移实在是麻烦，同查询一样，需要使用到SQL语句，但比查询麻烦的多。感兴趣的各位可以参考下面的文章：

[《Understanding migrations with Room》](#) 谷歌工程师写的 [《Android Room 框架学习》](#)

5.5 总结



总结

Room

作为谷歌的官方数据库框架，优点和缺点都十分明显。至此，

Room

的学习就此就结束了。本人水平有限，难免有误，欢迎指正。

Over~

参考文章：

[《Android Room 框架学习》](#) [《7 Steps To Room》](#)

6. Android Jetpack - Paging

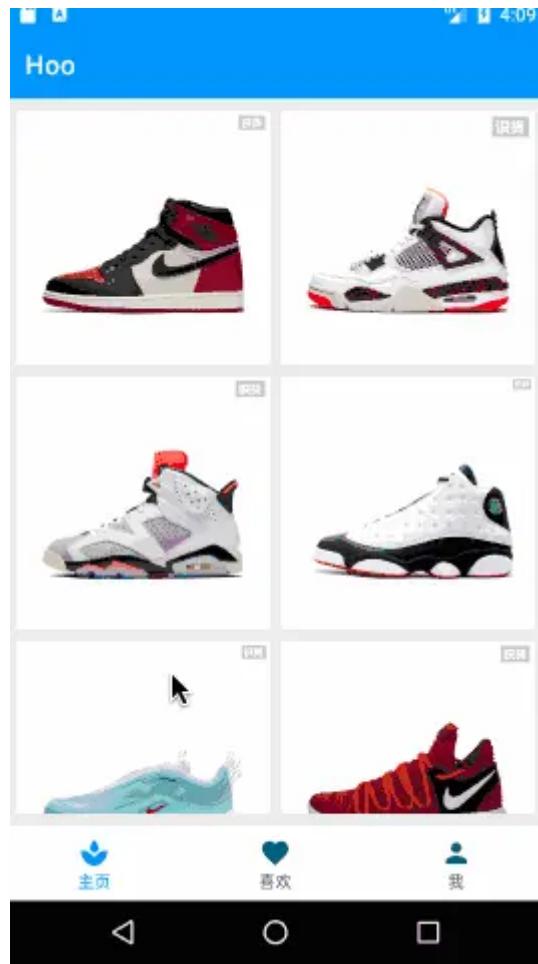
6.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第五篇。

我相信几乎所有的Android开发者都会遇到在 RecyclerView 加载大量数据的情况，如果是在数据库请求，需要消耗数据库资源并且需要花费较多的时间，同样的，如果是发送网络请求，则需要消耗带宽和更多的时间，无论处于哪一种情形，对于用户的体验都是糟糕的。在这两种情形中，如果采用分段加载则缩短了时间，给用户带来了良好的体验，目前，对于加载大量数据的处理方法有两种：

1. 借助刷新控件实现用户手动请求数据。
2. 数据到达边界自动请求加载。

谷歌架构组件 Android Jetpack 也实现了自己的分页库 Paging，以下是我使用Paging实现的效果：



效果

语言 : `Kotlin` Demo地址 : <https://github.com/mCyp/Hoo>

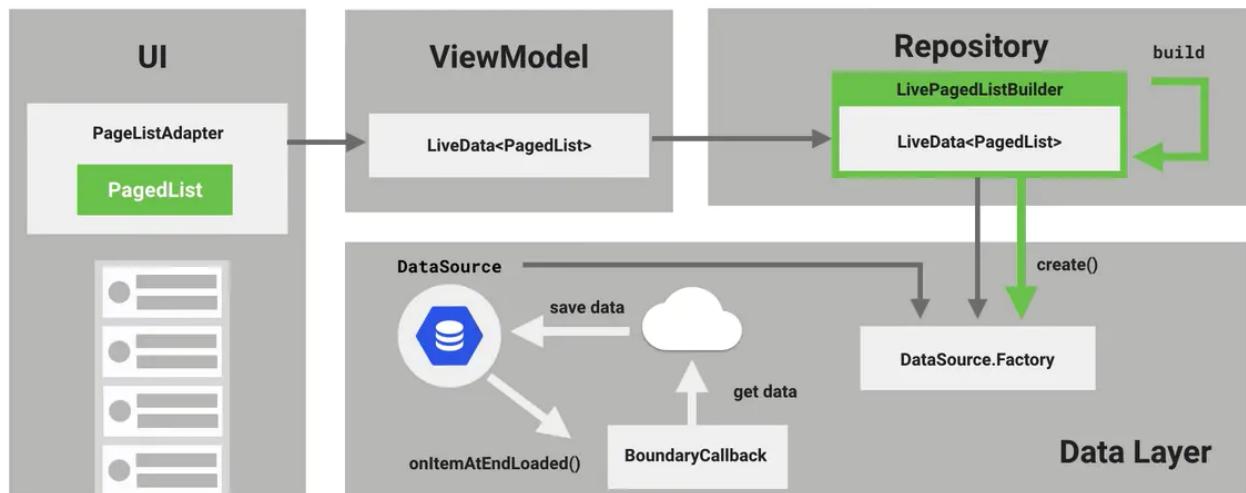
6.2 介绍

友情提示 官方文档 : [Paging](#) 谷歌实验室 : [官方教程](#) 官方Demo : [网络方式](#), [数据库方式](#)

我们在前言中已经简要的介绍了 `Paging` , 让我们看看谷歌官方如何介绍 :

The Paging Library helps you load and display small chunks of data at a time. Loading partial data on demand reduces usage of network bandwidth and system resources.

可以看到 , 官方的介绍就是我们上面提及的内容 , 我们再来看看 `Paging` 是如何运作的 :



Paging架构

当然 , 我需要做更具体的介绍 , 至于

`ViewModel`

和

`LiveData`

, 可以翻阅我的前几期博客 , 关键元素如下 :

名称	作用
PagedList	一个可以以分页形式异步加载数据的容器，可以跟 RecyclerView 很好的结合
DataSource 或 DataSource.Factory	数据源，DataSource 将数据转变成 PagedList，DataSource.Factory 则用来创建 DataSource
LivePagedListBuilder	用来生成 LiveData<PagedList>，需要 DataSource.Factory 参数
BoundaryCallback	数据到达边界的回调
PagedListAdapter	一种 RecyclerView 的适配器

6.2.1 优点

网上的分页解决方法挺多的，与他们相比，Paging有什么优点呢？

- RxJava 2 以及 Android Jetpack 的支持，如 LiveData、Room。
- 自定义分页策略。
- 异步处理数据。
- 结合RecyclerView等

6.3 实战

因为本文是 Android Jetpack 系列文章，所以主要介绍配合 LiveData 使用，对于 RxJava 的配合使用，本文会一笔带过。

6.3.1 第一步 添加依赖

```
ext.pagingVersion = '2.1.0-alpha01'
dependencies {
    // ... 省略
    // paging
    implementation "androidx.paging:paging-runtime:$pagingVersion"
}
```

6.3.2 第二步 创建数据源

1. 非Room数据库

如果没有使用 Room 数据库，我们需要自定义实现 DataSource，通常实现 DataSource 有三种方式，分别继承三种抽象类，它们分别是：

名称	使用场景
<code>PageKeyedDataSource<Key, value></code>	分页请求数据的场景
<code>ItemKeyedDataSource<Key, value></code>	以表的某个列为key，加载其后的N个数据（个人理解以某个字段进行排序，然后分段加载数据）
<code>PositionalDataSource<T></code>	当数据源总数特定，根据指定位置请求数据的场景

这里我们以 `PageKeyedDataSource<Key, value>` 为例，虽然这里的数据库使用的是 Room，但我们查询数据以返回 `List<Shoe>` 代表着通常数据库的使用方式：

```
// 因为代表着不同方式，所以不需要看Dao层
class ShoeRepository private constructor(private val shoeDao: ShoeDao) {

    /**
     * 通过id的范围寻找鞋子
     */
    fun getPageShoes(startIndex: Long, endIndex: Long): List<Shoe> =
        shoeDao.findShoesByIndexRange(startIndex, endIndex)

    //... 省略
}

/**
 * 自定义PageKeyedDataSource
 * 演示Page库的时候使用
 */
class CustomPageDataSource(private val shoeRepository: ShoeRepository) :
    PageKeyedDataSource<Int, Shoe>() {

    private val TAG: String by lazy {
        this::class.java.simpleName
    }

    // 第一次加载的时候调用
    override fun loadInitial(params: LoadInitialParams<Int>, callback:
    LoadInitialCallback<Int, Shoe>) {
        val startIndex = 0L
        val endIndex: Long = 0L + params.requestedLoadSize
        val shoes = shoeRepository.getPageShoes(startIndex, endIndex)

        callback.onResult(shoes, null, 2)
    }

    // 每次分页加载的时候调用
    override fun loadAfter(params: LoadParams<Int>, callback: LoadCallback<Int, Shoe>) {
        Log.e(TAG, "startPage:${params.key},size:${params.requestedLoadSize}")

        val startPage = params.key
        val startIndex = ((startPage - 1) * BaseConstant.SINGLE_PAGE_SIZE).toLong() + 1
    }
}
```

```

        val endIndex = startIndex + params.requestedLoadSize - 1
        val shoes = shoeRepository.getPageShoes(startIndex, endIndex)

        callback.onResult(shoes, params.key + 1)
    }

    override fun loadBefore(params: LoadParams<Int>, callback: LoadCallback<Int, Shoe>) {
        // ... 省略类似loadAfter
    }
}

```

DataSource 创建好了，再创建一个 `DataSource.Factory`，这个比较简单，返回上面创建的 `CustomPageDataSource` 实例：

```

/**
 * 构建CustomPageDataSource的工厂
 */
class CustomPageDataSourceFactory(val shoeRepository:
ShoeRepository):DataSource.Factory<Int,Shoe>() {
    override fun create(): DataSource<Int, Shoe> {
        return CustomPageDataSource(shoeRepository)
    }
}

```

2. Room数据库

如果是使用 Room 与 Paging 结合的方式呢？直接在 Room 的 Dao 层中这样使用：

```

/**
 * 鞋子的方法
 */
@Dao
interface ShoeDao {
    //... 省略

    // 配合LiveData 返回所有的鞋子
    @Query("SELECT * FROM shoe")
    fun getAllShoesLD(): DataSource.Factory<Int, Shoe>
}

```

不止简单了一个档次~

6.3.3 第三步 构建LiveData

想要获得 `LiveData<PagedList>` 则需要先创建 `LivePagedListBuilder`，`LivePagedListBuilder` 有设分页数量和配置参数两种构造方法，设置分页数量比较简单，直接查看 `Api` 就可以使用，我们看看如何配置参数使用：

```

class ShoeModel constructor(shoeRepository: ShoeRepository) : ViewModel() {
    // 鞋子集合的观察类
    val shoes: LiveData<PagedList<Shoe>> = LivePagedListBuilder<Int, Shoe>(
        CustomPageDataSourceFactory(shoeRepository) // DataSourceFactory
        , PagedList.Config.Builder()
            .setPageSize(10) // 分页加载的数量
            .setEnablePlaceholders(false) // 当item为null是否使用PlaceHolder展示
            .setInitialLoadSizeHint(10) // 预加载的数量
            .build())
        .build()
}

```

6.3.4 第四步 创建PagedListAdapter

PagedListAdapter 就是特殊的 RecyclerView 的 RecyclerAdapter，跟 RecyclerAdapter 一样，需要继承并实现其方法，这里使用了 Data Binding：

```

/**
 * 鞋子的适配器 配合Data Binding使用
 */
class ShoeAdapter constructor(val context: Context) :
    PagedListAdapter<Shoe, ShoeAdapter.ViewHolder>(shoeDiffCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder(
            RecyclerItemShoeBinding.inflate(
                LayoutInflater.from(parent.context)
                , parent
                , false
            )
        )
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val shoe = getItem(position)
        holder.apply {
            bind(onCreateListener(shoe!!.id), shoe)
            itemView.tag = shoe
        }
    }

    /**
     * Holder的点击事件
     */
    private fun onCreateListener(id: Long): View.OnClickListener {
        return View.OnClickListener {
            val intent = Intent(context, DetailActivity::class.java)
            intent.putExtra(BaseConstant.DETAIL_SHOE_ID, id)
            context.startActivity(intent)
        }
    }
}

```

```
class ViewHolder(private val binding: RecyclerItemShoeBinding) :  
RecyclerView.ViewHolder(binding.root) {  
  
    fun bind(listener: View.OnClickListener, item: Shoe) {  
        binding.apply {  
            this.listener = listener  
            this.shoe = item  
            executePendingBindings()  
        }  
    }  
}  
}
```

上面出现的 `ShoeDiffCallback`:

```
class ShoeDiffCallback: DiffUtil.ItemCallback<Shoe>() {  
    override fun areItemsTheSame(oldItem: Shoe, newItem: Shoe): Boolean {  
        return oldItem.id == newItem.id  
    }  
  
    override fun areContentsTheSame(oldItem: Shoe, newItem: Shoe): Boolean {  
        return oldItem == newItem  
    }  
}
```

布局文件只有一个 `ImageView`，不再赘述。

6.3.5 第五步 监听数据

同样使用了 `Data Binding`，`FragmentShoe` 的布局仅仅只用了一个 `RecyclerView`，比较简单，也不再赘述。

```
/**  
 * 鞋子页面  
 */  
class ShoeFragment : Fragment() {  
  
    // ... 省略  
  
    override fun onCreateView(  
        inflater: LayoutInflater, container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
        val binding: FragmentShoeBinding = FragmentShoeBinding.inflate(inflater, container,  
false)  
        context ?: return binding.root  
        val adapter = ShoeAdapter(context!!)  
        binding.recycler.adapter = adapter  
        onSubscribeUi(adapter)  
        return binding.root  
    }  
}
```

```
/**
 * 鞋子数据更新的通知
 */
private fun onSubscribeUi(adapter: ShoeAdapter) {
    viewModel.shoes.observe(viewLifecycleOwner, Observer {
        if (it != null) {
            adapter.submitList(it)
        }
    })
}
```

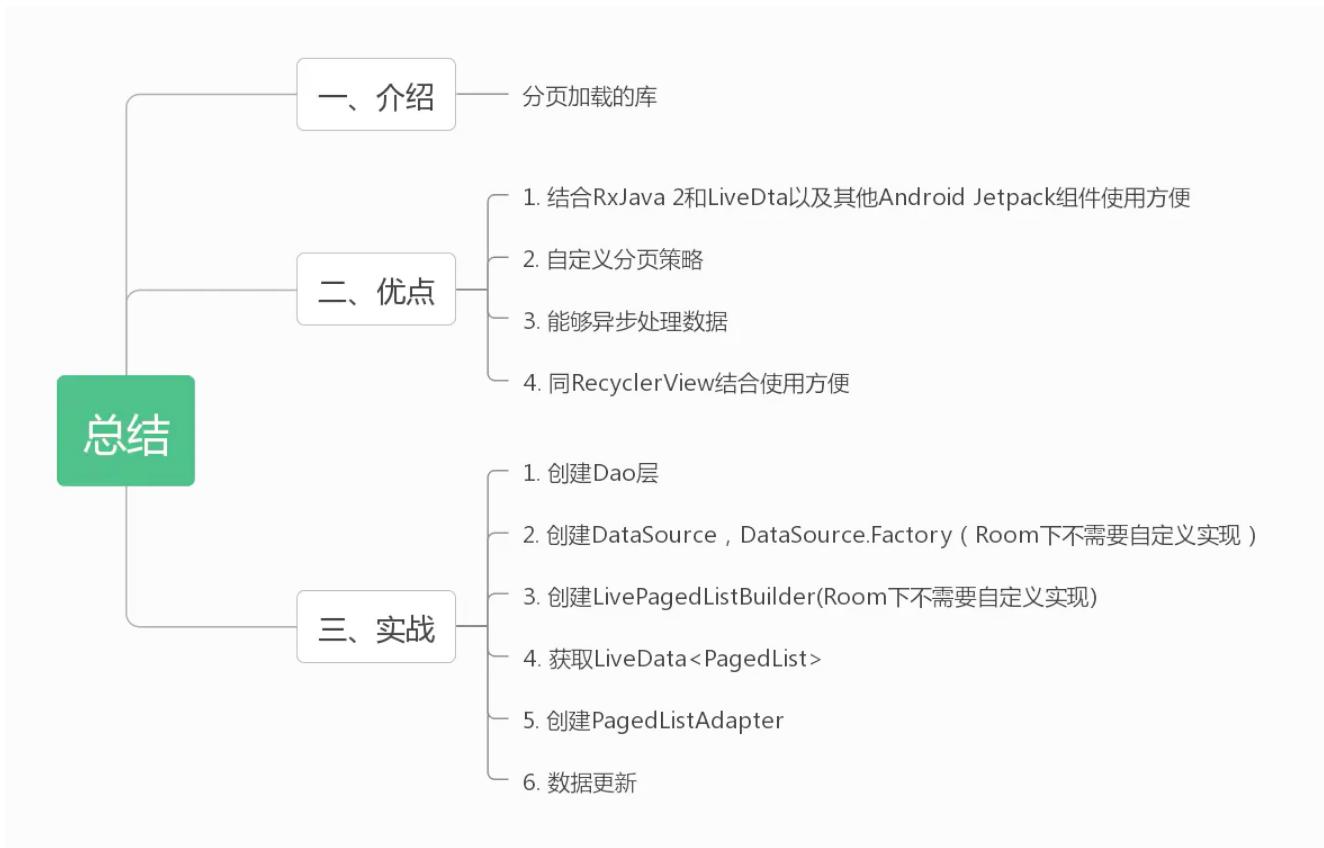
这样，我们的程序就可以分段加载数据了，感兴趣的可以打一下日志：

```
2019-06-30 17:15:00.564 32051-32117/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:2,size:10
2019-06-30 17:15:02.836 32051-32112/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:3,size:10
2019-06-30 17:15:13.705 32051-32113/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:4,size:10
2019-06-30 17:15:15.869 32051-32116/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:5,size:10
2019-06-30 17:15:19.986 32051-32117/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:6,size:10
2019-06-30 17:15:22.102 32051-32112/com.joe.jetpackdemo E/CustomPageDataSource:
startPage:7,size:10
```

6.4 更多

RxJava 2 如此强大，怎么能少了对 RxJava 2 的支持呢？对于上述的代码，我们只要将数据观测的 LiveData 修改成 RxJava 2 就行了，因为 Room 对 RxJava 2 也提供了支持，并且 RxJava 2 和 LiveData 做的本质工作是相同的，这里不写代码了，感兴趣的稍微查看一下官方文档就行了。

6.5 总结



总结

以上内容是本篇博客的全部，本人知识水平有限，难免有误，欢迎指正。Over~

参考内容：

[《Android Jetpack之Paging初探》](#) [《官方文档》](#)

7. Android Jetpack - WorkManger

7.1 前言

即学即用Android Jetpack系列Blog的目的是通过学习Android Jetpack完成一个简单的Demo，本文是即学即用Android Jetpack系列Blog的第六篇。

经过前面几篇博客的学习，我们的Demo已经基本成型，先上图：

20:46 ⌂

□ ○ □ ○ □ 37

Hoo

识货

识货



识货

识货



识货

识货



主页



喜欢



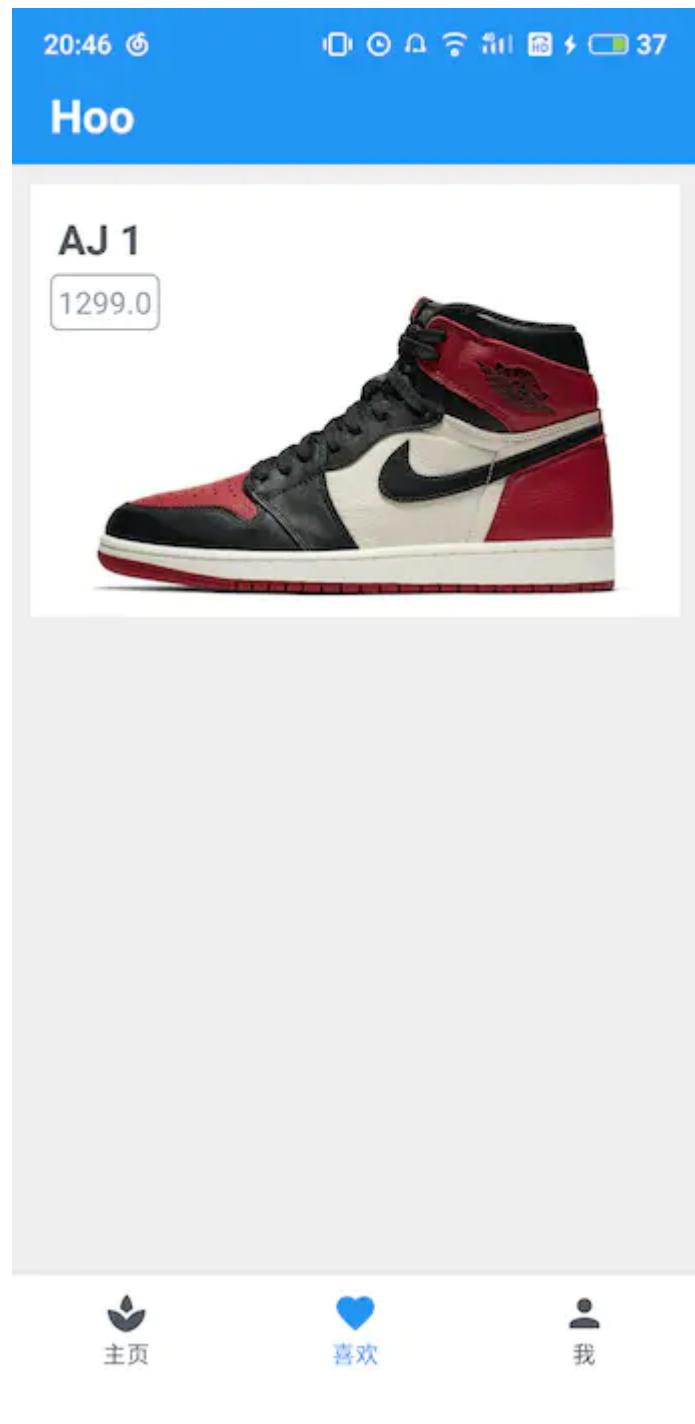
我

列表页



AIR MAX '97 男子运动鞋以流线型鞋身和卓越科技为热爱运动的你造就全方位防护，强大的功能为你的双脚带来更高境界的舒适、轻质、缓震和响应力，打造AIR MAX系列的不朽传说，更助你不断创造属于你的运动巅峰纪录。

详情页



主页



喜欢



我

喜欢页

这里我得提一下，鞋子的数据不是从网络请求中获取的，这个时候小王就举手了，那鞋子的数据是哪里来的呢？其实很简单，数据是从

assets

目录下的

json

读取出来的，通常情况下，从文件读取数据都不会放在主线程中执行，所以呢，我们Demo中的数据初始化当然也没有在主线程执行了，这时，就得请出我们今天的主角——

，它是我们能够在后台执行数据初始化的原因。

语言：`Kotlin` 我的Demo：<https://github.com/mCyp/Hoo>

7.2 介绍

友情提示 官方文档：[WorkManager](#) 谷歌实验室：[官方教程](#) 官方案例：[android-workmanager](#) 以及强力安利：
WorkManger介绍视频：[中文官方介绍视频](#) (主要是小姐姐好看~)

7.2.1 定义

通过一开始粗略的介绍，我们已经了解到，`workManager` 是用来执行后台任务的，正如官方介绍：

[WorkManager](#), a compatible, flexible and simple library for deferrable background work. WorkManger是一个可兼容、灵活和简单的延迟后台任务。

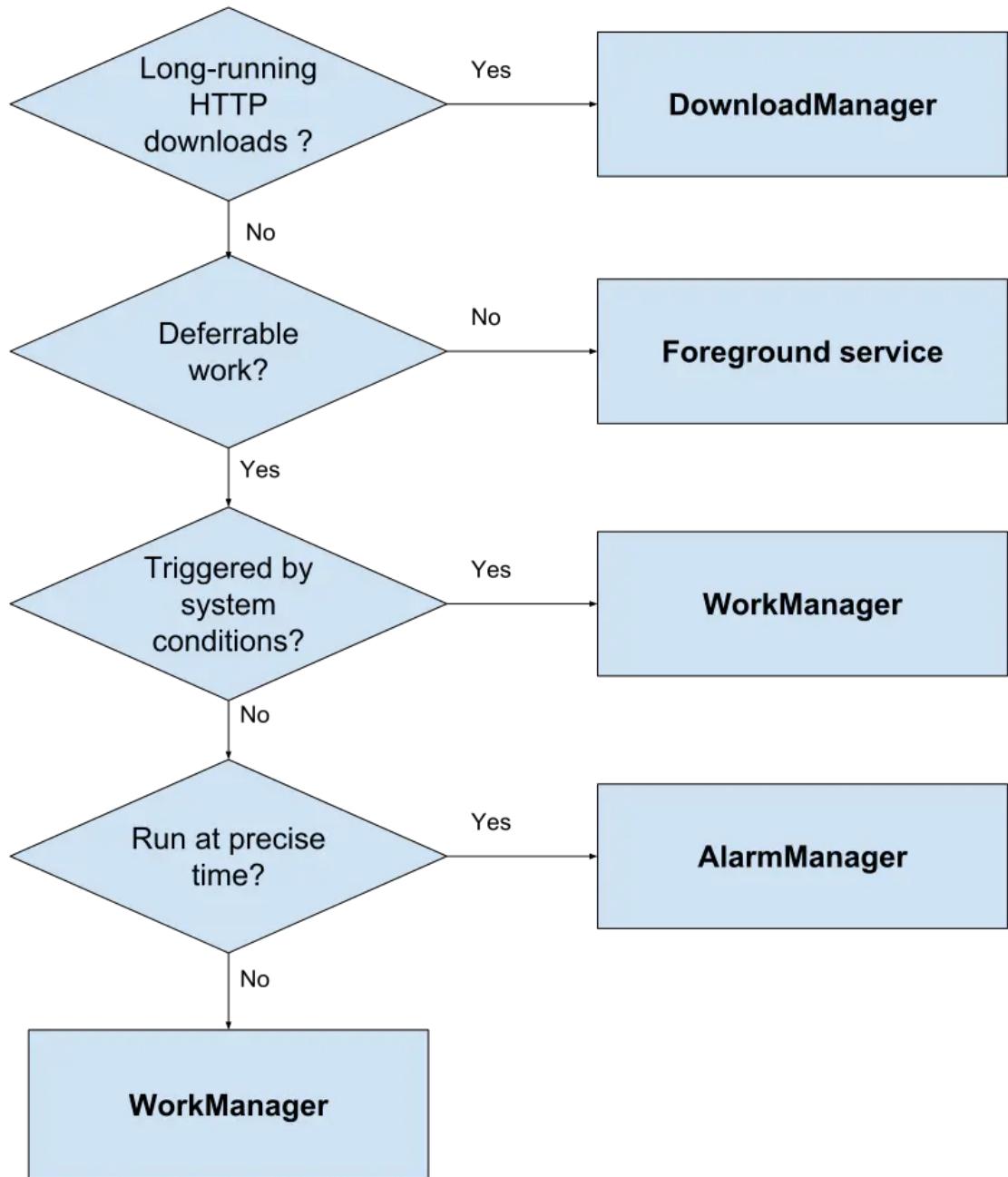
7.2.2 选择WorkManager的理由

Android中处理后台任务的选择挺多的，比如`service`、`DownloadManager`、`AlarmManager`、`Jobscheduler`等，那么选择`workManager`的理由是什么呢？

1. 版本兼容性强，向后兼容至API 14。
2. 可以指定约束条件，比如可以选择必须在有网络的条件下执行。
3. 可定时执行也可单次执行。
4. 监听和管理任务状态。
5. 多个任务可使用任务链。
6. 保证任务执行，如当前执行条件不满足或者App进程被杀死，它会等到下次条件满足或者App进程打开后执行。
7. 支持省电模式。

7.2.3 多线程任务如何选择？

后台任务会消耗设备的系统资源，如若处理不当，可能会造成设备电量的急剧消耗，给用户带来糟糕的体验。所以，选择正确的后台处理方式是每个开发者应当注意的，如下是官方给的选择方式：



选择方式

图片来自：

官方文档

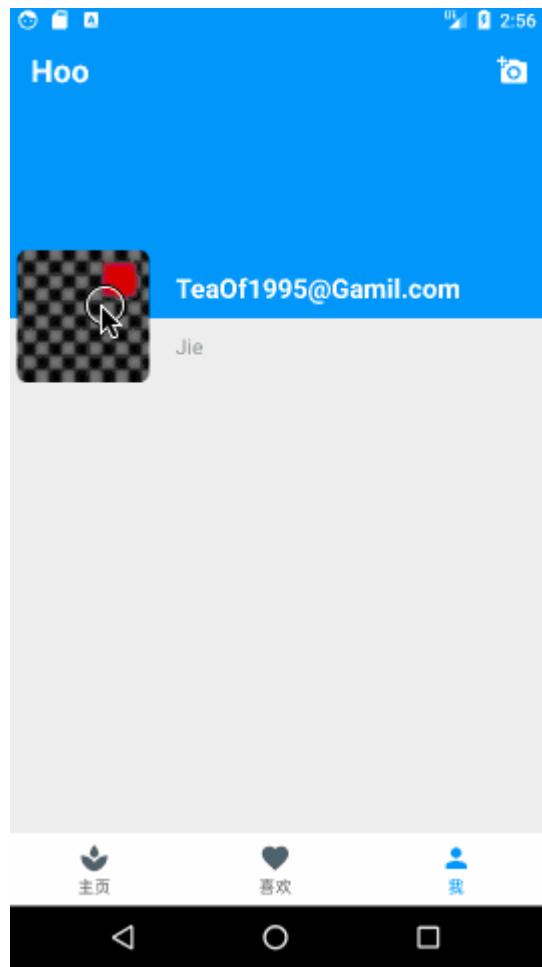
关于一些后台任务的知识，我推荐你阅读：

[译] 从Service到WorkManager

，很好的一篇文章。

7.3 实战

本次的实战来自于我上面的介绍的官方例子，最终我将它添加进我的Demo里面：



效果

如图所见，我们要做的就是选取一张图片，将图片做模糊处理，之后显示在我们的头像上。

7.3.1 第一步 添加依赖

```
ext.workVersion = "2.0.1"
dependencies {
    // ...省略

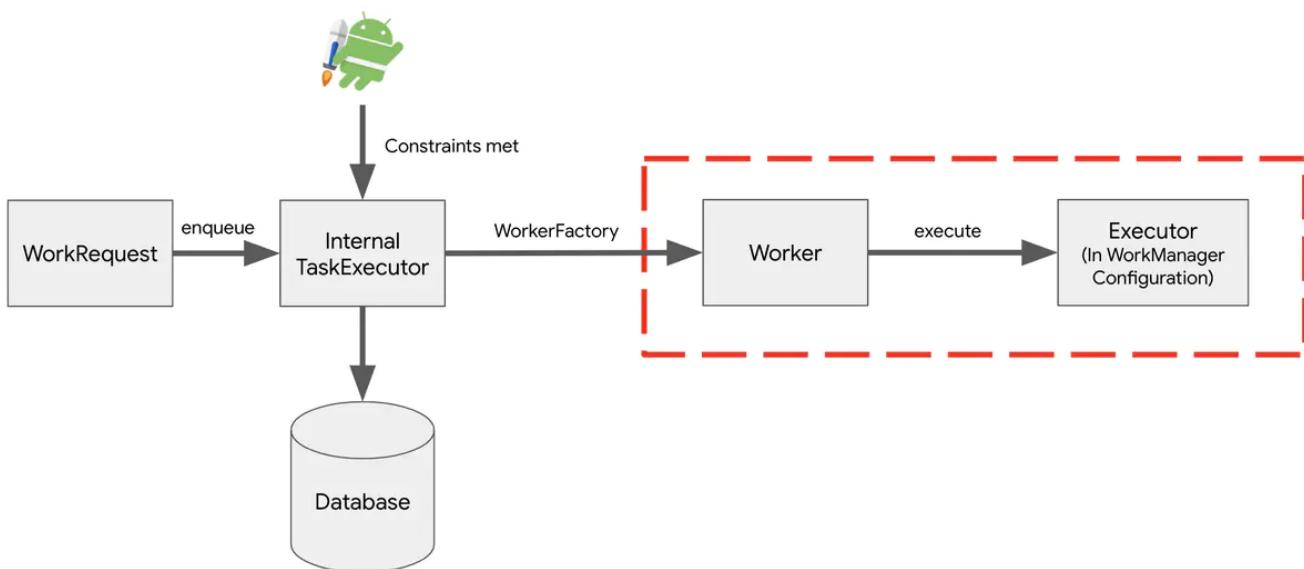
    implementation "androidx.work:work-runtime-ktx:$rootProject.workVersion"
}
```

7.3.2 第二步 自定义Worker

构建 `worker` 之前，我们有必要了解一下 `WorkManger` 中重要的几个类：

类	作用
worker	需要继承 worker，并复写 dowork() 方法，在 dowork() 方法中放入你需要在后台执行的代码。
workRequest	指后台工作的请求，你可以在后台工作的请求中添加约束条件
workManager	真正让 worker 在后台执行的类

除了这几个重要的类，我们仍需了解 workManger 的执行流程，以便于我们能够更好的使用：



WorkerManger

图片来自：

谷歌工程师的博客

主要分为三步：

1. workRequest 生成以后， Internal TaskExecutor 将它存入 workManger 的数据库中，这也是为什么即使在程序退出之后， workManger 也能保证后台任务在下次启动后条件满足的情况下执行。
2. 当约束条件满足的情况下， Internal TaskExecutor 告诉 workFactory 生成 worker 。
3. 后台任务 worker 执行。

下面开始我们的**构建Worker**，为了生成一张模糊图片，我们需要：清除之前的缓存路径、图片模糊的处理和图片的生成。我们可以将这三个步骤分为三个后台任务，三个后台任务又分别涉及到无变量情况、往外传参和读取参数这三种情况：

通常情况

```
/** 
 * 清理临时文件的worker
 */
```

```
class CleanUpWorker(ctx: Context, params: WorkerParameters) : worker(ctx, params) {
    private val TAG by lazy {
        this::class.java.simpleName
    }

    override fun dowork(): Result {
        // ... 省略

        return try {
            // 删除逻辑
            // ...代码省略
            // 成功时返回
            Result.success()
        } catch (exception: Exception) {
            // 失败时返回
            Result.failure()
        }
    }
}
```

输出参数

```
/**
 * 模糊处理的worker
 */
class BlurWorker(context: Context, params: WorkerParameters) : worker(context, params) {

    override fun dowork(): Result {
        //...
        return try {
            // 图片处理逻辑
            // 图片处理逻辑省略...

            // 将路径输出
            val outPutData = workDataOf(KEY_IMAGE_URI to outputUri.toString())
            makeStatusNotification("Output is $outputUri", context)
            Result.success(outPutData)
        }catch (throwable: Throwable){
            Result.failure()
        }
    }
}
```

读取参数

```
/**
 * 存储照片的worker
 */
class SaveImageToFileWorker(ctx:Context,parameters: WorkerParameters):worker(ctx,parameters)
{
    //...
```

```

override fun dowork(): Result {
    //...
    return try {
        // 获取从外部传入的参数
        val resourceuri = inputData.getString(KEY_IMAGE_URI)
        //... 存储逻辑
        val imageUrl = MediaStore.Images.Media.insertImage(
            resolver, bitmap, Title, dateFormatter.format(date()))
        if (!imageUrl.isNullOrEmpty()) {
            val output = workDataOf(KEY_IMAGE_URI to imageUrl)
            Result.success(output)
        } else {
            // 失败时返回
            Result.failure()
        }
    } catch (exception: Exception) {
        // 异常时返回
        Result.failure()
    }
}
}

```

7.3.3 第三步 创建WorkManger

这一步还是挺简单的，`MeModel` 中单例获取：

```

class MeModel(val userRepository: UserRepository) : viewModel() {
    //...
    private val workManager = WorkManager.getInstance()
    // ...
}

```

7.3.4 第四步 构建WorkRequest

`WorkRequest` 可以分为两类：

名称	作用
<code>PeriodicworkRequest</code>	多次、定时执行的任务请求，不支持任务链
<code>OneTimeworkRequest</code>	只执行一次的任务请求，支持任务链

1. 执行一个任务

我们以 `oneTimeworkRequest` 为例，如果我们只有一个任务请求，这样写就行：

```

val request = OneTimeworkRequest.from(CleanUpWorker::class.java)
workManager.enqueue(request)

```

2. 执行多个任务

但是，这样写显然不适合我们当前的业务需求，因为我们有三个 `worker`，并且三个 `worker` 有先后顺序，因此我们可以使用任务链：

```
// 多任务按顺序执行
workManager.beginWith(
    mutableListOf(
        OneTimeWorkRequest.from(CleanUpWorker::class.java)
    )
    .then(OneTimeWorkRequestBuilder<BlurWorker>
() .setInputData(createInputDataForUri()).build())
    .then(OneTimeWorkRequestBuilder<SaveImageToFileWorker>().build())
    .enqueue()
```

等等，假设我多次点击图片更换头像，提交多次请求，由于网络等原因（虽然我们的Demo没有网络数据请求部分），最后返回的很有可能不是我们最后一次请求的图片，这显然是糟糕的，不过，`workManger` 能够满足你的需求，保证任务的唯一性：

```
// 多任务按顺序执行
workManager.beginUniqueWork(
    IMAGE_MANIPULATION_WORK_NAME, // 任务名称
    ExistingWorkPolicy.REPLACE, // 任务相同的执行策略 分为REPLACE, KEEP, APPEND
    mutableListOf(
        OneTimeWorkRequest.from(CleanUpWorker::class.java)
    )
    .then(OneTimeWorkRequestBuilder<BlurWorker>
() .setInputData(createInputDataForUri()).build())
    .then(OneTimeWorkRequestBuilder<SaveImageToFileWorker>().build())
    .enqueue()
```

无顺序多任务 这里有必要提一下，如果并行执行没有顺序的多个任务，无论是 `beginUniqueWork` 还是 `beginWith` 方法都可以接受一个 `List<OneTimeWorkRequest>`。

3. 使用约束

假设我们需要将生成的图片上传到服务端，并且需要将图片存储到本地，显然，我们需要设备网络条件良好并且有存储空间，这时候，我们可以给 `workRequest` 指明约束条件：

```
// 构建约束条件
val constraints = Constraints.Builder()
    .setRequiresBatteryNotLow(true) // 非电池低电量
    .setRequiredNetworkType(NetworkType.CONNECTED) // 网络连接的情况
    .setRequiresStorageNotLow(true) // 存储空间足
    .build()

// 储存照片
val save = OneTimeWorkRequestBuilder<SaveImageToFileWorker>()
    .setConstraints(constraints)
    .addTag(TAG_OUTPUT)
    .build()
continuation = continuation.then(save)
```

可以指明的约束条件有：电池电量、充电、网络条件、存储和延迟等，具体的可以使用的时候查看接口。

以下则是我们Demo中的具体使用：

```
class MeModel(val userRepository: UserRepository) : ViewModel() {
    //...
    private val workManager = WorkManager.getInstance()
    val user = userRepository.findUserById(AppPrefsUtils.getLong(BaseConstant.SP_USER_ID))

    internal fun applyBlur(blurLevel: Int) {
        //... 创建任务链

        var continuation = workManager
            .beginUniqueWork(
                IMAGE_MANIPULATION_WORK_NAME,
                ExistingWorkPolicy.REPLACE,
                OneTimeWorkRequest.from(CleanUpWorker::class.java)
            )

        for (i in 0 until blurLevel) {
            val builder = OneTimeWorkRequestBuilder<BlurWorker>()
            if (i == 0) {
                builder.setInputData(createInputDataForUri())
            }
            continuation = continuation.then(builder.build())
        }

        // 构建约束条件
        val constraints = Constraints.Builder()
            .setRequiresBatteryNotLow(true) // 非电池低电量
            .setRequiredNetworkType(NetworkType.CONNECTED) // 网络连接的情况
            .setRequiresStorageNotLow(true) // 存储空间足
            .build()

        // 储存照片
        val save = OneTimeWorkRequestBuilder<SaveImageToFileWorker>()
            .setConstraints(constraints)
            .addTag(TAG_OUTPUT)
            .build()
        continuation = continuation.then(save)

        continuation.enqueue()
    }

    private fun createInputDataForUri(): Data {
        val builder = Data.Builder()
        imageUri?.let {
            builder.putString(KEY_IMAGE_URI, imageUri.toString())
        }
        return builder.build()
    }
}
```

```
//... 省略  
}
```

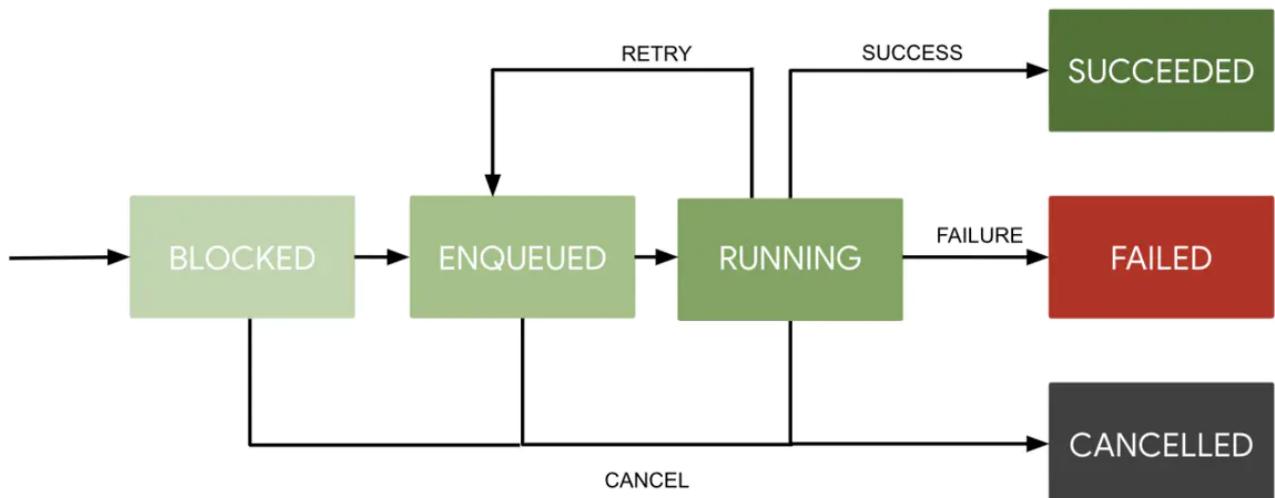
7.3.5 第五步 取消任务

如果想取消所有的任务 `workManager.cancelAllWork()`，当然如果想取消我们上面执行的唯一任务，需要我们上面的唯一任务名：

```
class MeModel(val userRepository: UserRepository) : ViewModel() {  
    fun cancelWork() {  
        workManager.cancelUniqueWork(IMAGE_MANIPULATION_WORK_NAME)  
    }  
}
```

7.3.6 第六步 观察任务状态

任务状态的变化过程：



状态观测

图片来自于：

How to use WorkManager with RxJava

其中，

```
SUCCEEDED
```

```
FAILED
```

和

```
CANCELLED
```

都属于任务已经完成。观察任务状态需要使用到

LiveData

:

```
class MeModel(val userRepository: UserRepository) : ViewModel() {  
    //... 省略  
    private val workManager = WorkManager.getInstance()  
    val user = userRepository.findUserById(AppPrefsUtils.getLong(BaseConstant.SP_USER_ID))  
  
    init {  
        outputWorkInfos = workManager.getWorkInfosByTagLiveData(TAG_OUTPUT)  
    }  
  
    // ...省略  
}
```

当图片处理的时候，程序弹出加载框，图片处理完成，程序会将图片路径保存到 user 里的 headImage 并存储到数据库中，任务状态观测参见 MeFragment 中的 onSubscribeUi 方法：

```
class MeFragment : Fragment() {  
    private val TAG by lazy { MeFragment::class.java.simpleName }  
    // 选择图片的标识  
    private val REQUEST_CODE_IMAGE = 100  
    // 加载框  
    private val sweetAlertDialog: SweetAlertDialog by lazy {  
        SweetAlertDialog(requireContext(), SweetAlertDialog.PROGRESS_TYPE)  
            .setTitleText("头像")  
            .setContentText("更新中...")  
            /*  
             *  
             .setCancelButton("取消") {  
                 model.cancelWork()  
                 sweetAlertDialog.dismiss()  
             }*/  
    }  
  
    // MeModel懒加载  
    private val model: MeModel by viewModels {  
        CustomViewModelProvider.providerMeModel(requireContext())  
    }  
  
    override fun onCreateView(  
        inflater: LayoutInflater, container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
        // Data Binding  
        val binding: FragmentMeBinding = FragmentMeBinding.inflate(inflater, container,  
        false)  
        initListener(binding)  
        onSubscribeUi(binding)  
    }  
}
```

```
        return binding.root
    }

    /**
     * 初始化监听器
     */
    private fun initListener(binding: FragmentMeBinding) {
        binding.ivHead.setOnClickListener {
            // 选择处理的图片
            val chooseIntent = Intent(
                Intent.ACTION_PICK,
                MediaStore.Images.Media.EXTERNAL_CONTENT_URI
            )
            startActivityForResult(chooseIntent, REQUEST_CODE_IMAGE)
        }
    }

    /**
     * Binding绑定
     */
    private fun onSubscribeUi(binding: FragmentMeBinding) {
        model.user.observe(this, Observer {
            binding.user = it
        })

        // 任务状态的观测
        model.outPutWorkInfos.observe(this, Observer {
            if (it.isNullOrEmpty())
                return@Observer

            val state = it[0]
            if (state.state.isFinished) {
                // 更新头像
                val outputImageUri = state.outputData.getString(KEY_IMAGE_URI)
                if (!outputImageUri.isNullOrEmpty()) {
                    model.setOutputUri(outputImageUri)
                }
                sweetAlertDialog.dismiss()
            }
        })
    }

    /**
     * 图片选择完成的回调
     */
    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        if (resultCode == Activity.RESULT_OK) {
            when (requestCode) {
                REQUEST_CODE_IMAGE -> data?.let { handleImageRequestResult(data) }
                else -> Log.d(TAG, "Unknown request code.")
            }
        } else {
            Log.e(TAG, String.format("Unexpected Result code %s", resultCode))
        }
    }
}
```

```

        }

    }

    /**
     * 图片选择完成的处理
     */
    private fun handleImageRequestResult(intent: Intent) {
        // If clipdata is available, we use it, otherwise we use data
        val imageUri: Uri? = intent.clipData?.let {
            it.getItemAt(0).uri
        } ?: intent.data

        if (imageUri == null) {
            Log.e(TAG, "Invalid input image uri.")
            return
        }

        sweetAlertDialog.show()
        // 图片模糊处理
        model.setImageUri(imageUri.toString())
        model.applyBlur(3)
    }
}

```

写完以后，动图的效果就会出现了。

7.4 更多

选择适合自己的Worker

谷歌提供了四种 `worker` 给我们使用，分别为：自动运行在后台线程的 `worker`、结合协程的 `coroutineworker`、结合 `RxJava2` 的 `Rxworker` 和以上三个类的基类的 `Listenableworker`。

由于本文使用的 `Kotlin`，故打算简单的介绍 `Coroutineworker`，其他的可以自行探索。

我们使用 `Shoeworker` 来从文件中读取鞋子的数据并完成数据库的插入工作，使用方式基本与 `worker` 一致：

```

class Shoeworker(
    context: Context,
    workerParams: WorkerParameters
) : Coroutineworker(context, workerParams) {

    private val TAG by lazy {
        Shoeworker::class.java.simpleName
    }

    // 指定Dispatchers
    override val coroutineContext: CoroutineDispatcher
        get() = Dispatchers.IO

    override suspend fun dowork(): Result = coroutineScope {
        try {
            applicationContext.assets.open("shoes.json").use {

```

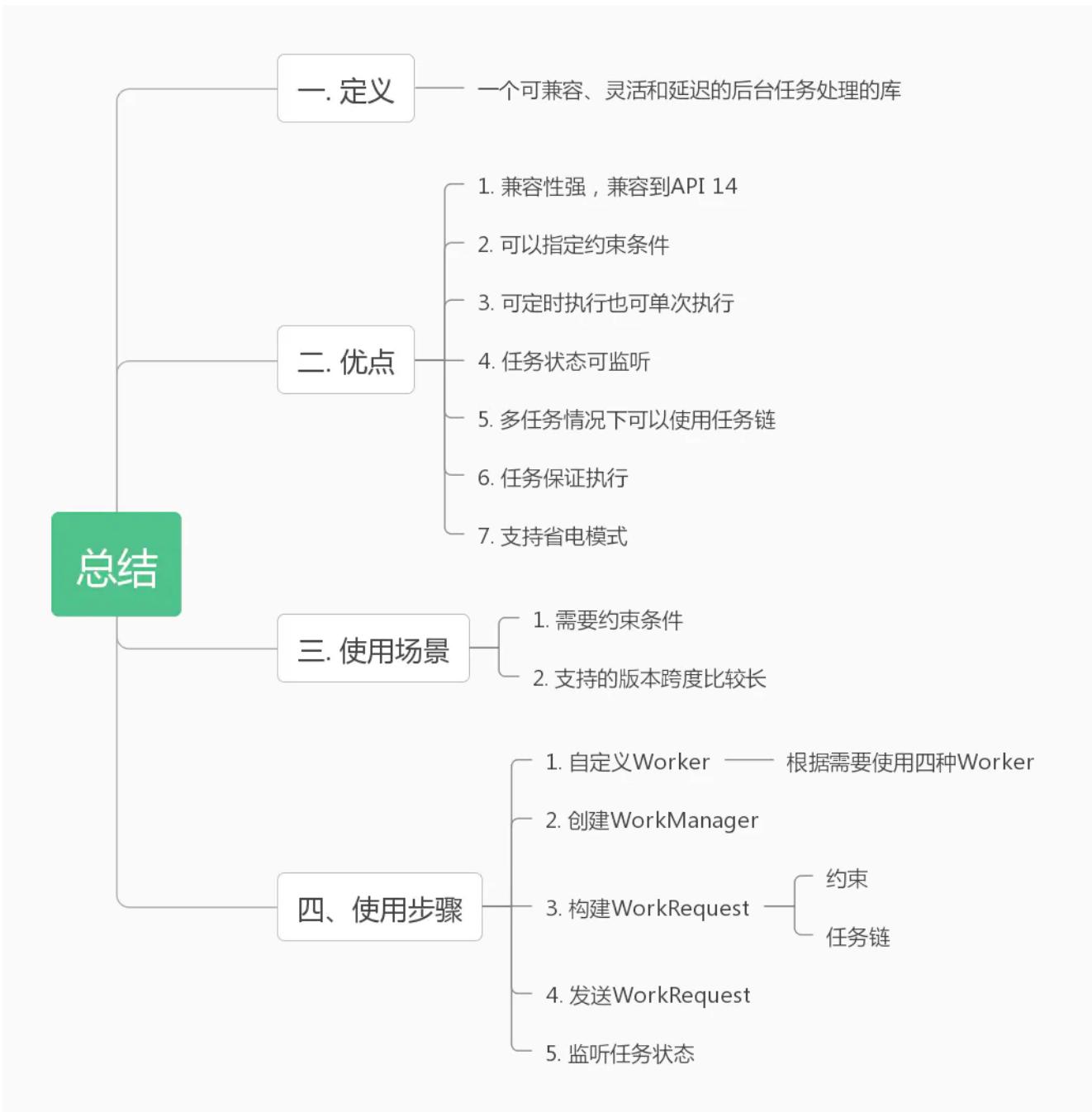
```
JsonReader(it.reader()).use {
    val shoeType = object : TypeToken<List<Shoe>>() {}.type
    val shoeList: List<Shoe> = Gson().fromJson(it, shoeType)

    val shoeDao =
RepositoryProvider.providerShoeRepository(applicationContext)
    shoeDao.insertShoes(shoeList)
    for (i in 0..2) {
        for (shoe in shoeList) {
            shoe.id += shoeList.size
        }
        shoeDao.insertShoes(shoeList)
    }
    Result.success()
}

}

} catch (ex: Exception) {
    Log.e(TAG, "Error seeding database", ex)
    Result.failure()
}
}
}
```

7.5 总结



总结

可以发现，大部分的后台任务处理，

WorkManager

都可以胜任，这也是我们需要学习

WorkManager

的原因。本次WorkManager学习完毕，本人水平有限，难免有误，欢迎指正。

Over~

参考文章：

[《Android Jetpack - 使用 WorkManager 管理后台任务》](#) [《[从Service到WorkManager](#)》] [《官方文档：Guide to background processing》](#) [《谷歌实验室》](#) [《官方文档：WorkManager》](#) [《WorkManager Basics》](#)

8. Android Jetpack架构组件之Lifecycle

8.1 Lifecycle简介

一直以来，解耦都是软件开发永恒的话题。在Android开发中，解耦很大程度上表现为系统组件的生命周期与普通组件之间的解耦，因为普通组件在使用过程中需要依赖系统组件的生命周期。

举个例子，我们经常需要在页面的onCreate()方法中对组件进行初始化，然后在onStop()中停止组件，或者在onDestory()方法中对其进行销毁。事实上，这样的工作非常繁琐，会让页面和页面耦合度变高，但又不得不做，因为如果不即时的释放资源，有可能会导致内存泄露。例如，下面是一个在Activity的不同生命周期方法中监听调用的例子，代码如下。

```
public class MainActivity extends AppCompatActivity {
    private MyListener myListener;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        myListener = new MyListener(MainActivity.this);
    }

    @Override
    protected void onStart() {
        super.onStart();
        myListener.start();
    }

    @Override
    protected void onStop() {
        super.onStop();
        myListener.stop();
    }
}

class MyListener {
    public MyListener(Context context) {
    ...
}
    void start() {
    ...
}
```

```
void stop() {  
    ...  
}  
}
```

虽然，代码看起来没什么问题，但在实际开发中可能会有多个组件在Activity的生命周期中进行回调，这样Activity的生命周期的方法中可能就需要编写大量的代码，这就使得它们难以维护。我们希望在对组件进行管理不依赖页面的生命周期的回调方法，同时当页面生命周期发生改变时，也能够即时的收到通知。这在Android组件化和架构设计的时候表现的尤为明显。

那纠结什么是Lifecycle组件呢？总的来说，Lifecycle 就是具有生命周期感知能力的组件。简单的理解就是，当Activity/Fragment的生命周期产生变化时，Lifecycle组件会感应相应的生命周期变化，当然我们还可以通过使用Lifecycle组件来在自定义的类中管理Activity/fragment的生命周期。

目前，Lifecycle生命周期组件主要由Lifecycle、LifecycleOwner、LifecycleObserver三个对象构成。

- **Lifecycle**：是一个持有组件生命周期状态与事件（如Activity或Fragment）的信息的类。
- **LifecycleOwner**：Lifecycle的提供者，通过实现LifecycleOwner接口来访问Lifecycle生命周期对象。
Fragment和FragmentActivity类实现了LifecycleOwner接口，它具有访问生命周期的getLifecycle方法，使用时需要在自己的类中实现LifecycleOwner。
- **LifecycleObserver**：Lifecycle观察者，可以使用LifecycleOwner类的addObserver()方法进行注册，被注册后LifecycleObserver便可以观察到LifecycleOwner的生命周期事件。

8.2 Lifecycle使用

使用Lifecycle进行应用开发之前，需要先在app的build.gradle文件中添加如下依赖代码。

```
dependencies {  
    def lifecycle_version = "2.2.0"  
    def arch_version = "2.1.0"  
  
    // ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle_version"  
    // LiveData  
    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle_version"  
    // Lifecycles only (without ViewModel or LiveData)  
    implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"  
  
    // Saved state module for ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lifecycle_version"  
  
    // Annotation processor  
    annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"  
    // alternately - if using Java8, use the following instead of lifecycle-compiler  
    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"  
  
    // optional - helpers for implementing LifecycleOwner in a Service  
    implementation "androidx.lifecycle:lifecycle-service:$lifecycle_version"  
  
    // optional - ProcessLifecycleOwner provides a lifecycle for the whole application  
    process  
    implementation "androidx.lifecycle:lifecycle-process:$lifecycle_version"
```

```
// optional - ReactiveStreams support for LiveData  
implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle_version"  
  
// optional - Test helpers for LiveData  
testImplementation "androidx.arch.core:core-testing:$arch_version"  
}
```

官网用的是AndroidX，因为使用AndroidX可能会产生一些迁移的问题，这里的例子就不使用AndroidX，使用lifecycleandroid.arch.lifecycle库即可，如下所示。

```
dependencies {  
    implementation fileTree(dir: "libs", include: ["*.jar"])  
    implementation 'androidx.appcompat:appcompat:1.2.0'  
    implementation 'androidx.constraintlayout:constraintlayout:2.0.2'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'  
  
    def lifecycle_version = "2.2.0"  
  
    // 包含ViewModel和LiveData  
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"  
    // 仅仅包含ViewModel  
    implementation "android.arch.lifecycle:viewmodel:$lifecycle_version" // For Kotlin use  
viewmodel-ktx  
    // 仅仅包含LiveData  
    implementation "android.arch.lifecycle:livedata:$lifecycle_version"  
    // 仅仅包含Lifecycles  
    implementation "android.arch.lifecycle:runtime:$lifecycle_version"  
    //noinspection LifecycleAnnotationProcessorwithJava8  
    annotationProcessor "android.arch.lifecycle:compiler:$lifecycle_version" // For Kotlin  
use kapt instead of annotationProcessor  
    // 如果用Java8，用于替代compiler  
    implementation "android.arch.lifecycle:common-java8:$lifecycle_version"  
    // 可选，ReactiveStreams对LiveData的支持  
    implementation "android.arch.lifecycle:reactivestreams:$lifecycle_version"  
    // 可选，LiveData的测试  
    testImplementation "android.arch.core:core-testing:$lifecycle_version"  
}
```

按照Lifecycle的使用流程，需要先定义观察者，并重写对应的生命周期，代码如下。

```
public class MyObserver implements LifecycleObserver {  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
    void onResume(){  
        Log.d(TAG, "Lifecycle call onResume");  
    }  
    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)  
    void onPause(){  
        Log.d(TAG, "Lifecycle call onPause");  
    }  
}
```

然后，我们在onCreate()方法中添加观察者，代码如下。

```
getLifecycle().addObserver(new MyObserver());
```

完整的代码如下所示。

```
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "MainActivity";  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        getLifecycle().addObserver(new MyObserver());  
    }  
  
    @Override  
    protected void onResume() {  
        super.onResume();  
        Log.d(TAG, "onResume");  
    }  
  
    @Override  
    protected void onPause() {  
        super.onPause();  
        Log.d(TAG, "onPause");  
    }  
  
    //自定义观察者  
    public class MyObserver implements LifecycleObserver {  
  
        @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
        void onResume(){  
            Log.d(TAG, "Lifecycle call onResume");  
        }  
        @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)  
        void onPause(){  
            Log.d(TAG, "Lifecycle call onPause");  
        }  
    }  
}
```

```
    }  
}
```

经过上面的处理后，MyObserver就可以观察MainActivity的生命周期变化了。在上面的示例中，LifecycleOwner可以理解为被观察者，MainActivity默认实现了LifecycleOwner接口，也就是说MainActivity是被观察者。运行上面的代码，得到如下的日志。

```
com.xzh.androidx D/MainActivity: MainActivity onResume  
com.xzh.androidx D/MainActivity: Lifecycle call onResume  
com.xzh.androidx D/MainActivity: Lifecycle call onPause  
com.xzh.androidx D/MainActivity: MainActivity onPause
```

当然，在被观察者中进行注册时，我们还可以对代码进行拆解，写成下面的方式。

```
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "MainActivity";  
    private LifecycleRegistry registry;  
    private MyObserver myObserver = new MyObserver();  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        init();  
    }  
  
    @Override  
    protected void onResume() {  
        super.onResume();  
        registry.setCurrentState(Lifecycle.State.RESUMED);  
    }  
  
    @NonNull  
    @Override  
    public Lifecycle getLifecycle() {  
        return registry;  
    }  
  
    private void init() {  
        registry = new LifecycleRegistry(this);  
        registry.addObserver(myObserver);  
    }  
  
    ... //省略Myobserver代码  
}
```

在自定义的Activity或Fragment中实现LifeCycleOwner时，可以实现LifecycleRegistryOwner接口，如下所示。

```
public class MyFragment extends Fragment implements LifecycleRegistryOwner {
    LifecycleRegistry lifecycleRegistry = new LifecycleRegistry(this);

    @Override
    public LifecycleRegistry getLifecycle() {
        return lifecycleRegistry;
    }
}
```

通过示例的分析可以发现，Android的Lifecycle组件需要先创建一个观察者，当组件生命周期发生变化时，通知观察者LifeCycle注解的方法做出响应。

8.3 Lifecycle源码分析

8.3.1 Lifecycle注册流程

Lifecycle使用两个枚举来跟踪其关联组件的生命周期状态，这两个枚举分别是Event和State。

- **State** : Lifecycle的生命周期所处的状态。
- **Event** : Lifecycle生命周期对应的事件，这些事件会映射到Activity和Fragment中的回调事件中。

打开lifecycle:common库下的Lifecycle类，

```
public abstract class Lifecycle {

    @RestrictTo(RestrictTo.Scope.LIBRARY_GROUP)
    @NonNull
    AtomicReference<Object> mInternalScopeRef = new AtomicReference<>();

    @MainThread
    public abstract void addObserver(@NonNull LifecycleObserver observer);

    @MainThread
    public abstract void removeObserver(@NonNull LifecycleObserver observer);

    @MainThread
    @NonNull
    public abstract State getCurrentState();

    @SuppressWarnings("WeakerAccess")
    public enum Event {
        ON_CREATE,
        ON_START,
        ON_RESUME,
        ON_PAUSE,
        ON_STOP,
        ON_DESTROY,
        ON_ANY
    }
}
```

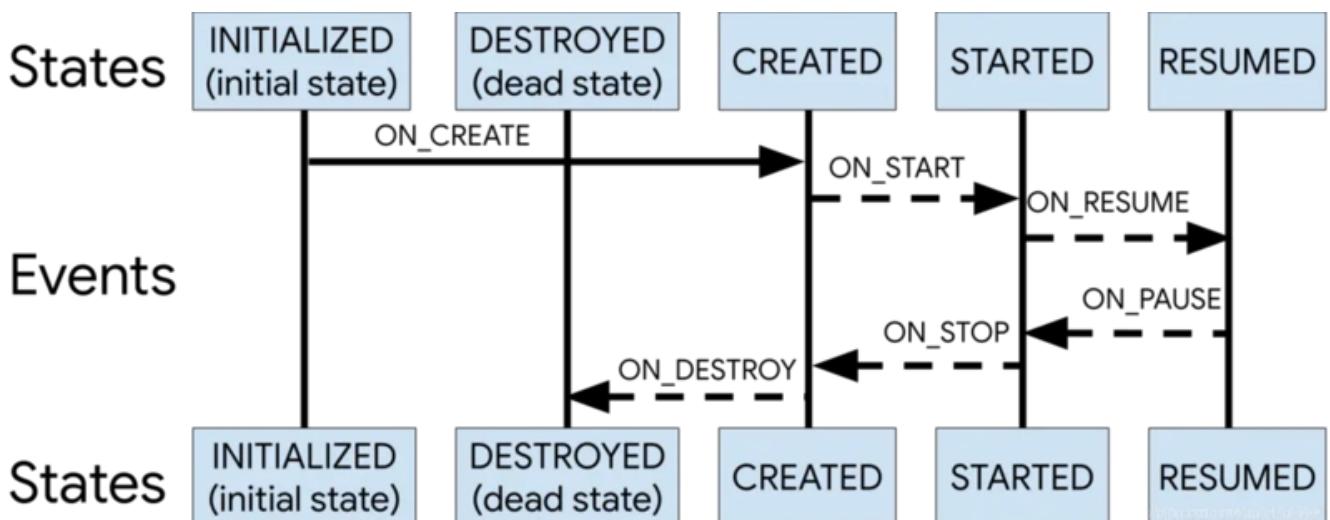
```

@SuppressWarnings("WeakerAccess")
public enum State {
    DESTROYED,
    INITIALIZED,
    CREATED,
    STARTED,
    RESUMED;

    public boolean isAtLeast(@NonNull State state) {
        return compareTo(state) >= 0;
    }
}

```

可以发现，Lifecycle是一个抽象类，其内部不仅包括了添加和移除观察者的方法，还包括了Event和State两个枚举。并且，Event中的事件和Activity的生命周期几乎是对应的，除了ON_ANY，它可用于匹配所有事件。State与Event的生命周期关系的时序图如下图所示。



在Lifecycle抽象类中，enum枚举定义了所有State，各个状态都是按照固定的顺序来变化的，所以State具备了生命周期的概念。Lifecycle是抽象类，唯一的具体实现类为 LifecycleRegistry，源码如下。

```

public class LifecycleRegistry extends Lifecycle {
    /**
     * 一个列表，并且可以在遍历期间添加或者删除元素
     * 新观察者的状态一定是小于等于之前的观察者的
     */
    private FastSafeIterableMap<LifecycleObserver, ObserverWithState> mObserverMap =
        new FastSafeIterableMap<>();

    /**
     * 当前状态
     */
    private State mState;

    /**
     * 以弱引用保存LifecycleOwner，防止内存泄漏
     */
    private final WeakReference<LifecycleOwner> mLifecycleOwner;
}

```

LifecycleRegistry将事件通知给所有观察者之前，存在一个同步的过程。这个同步的过程中，前面的观察者已经通知到了，后面的观察者还没被通知，于是所有观察者之间的状态就不一致了，各观察者状态之间便产生了差异，只有第一个观察者的状态等于最后一个观察者的状态，并且等于LifecycleRegistry中的当前状态mState，才说明状态同步整个完成了。

加下来，我们来看一下Lifecycle的注册流程，addObserver()方法是注册观察者的入口，源码如下。

```
@Override  
public void addObserver(@NonNull LifecycleObserver observer) {  
    State initialState = mState == DESTROYED ? DESTROYED : INITIALIZED;  
    ObserverWithState statefulObserver = new ObserverWithState(observer, initialState);  
    ObserverWithState previous = mObserverMap.putIfAbsent(observer, statefulObserver);  
    if (previous != null) {  
        return;  
    }  
    LifecycleOwner lifecycleOwner = mLifecycleOwner.get();  
    if (lifecycleOwner == null) {  
        return;  
    }  
  
    boolean isReentrance = mAddingObserverCounter != 0 || mHandlingEvent;  
    State targetState = calculateTargetState(observer);  
    mAddingObserverCounter++;  
  
    while ((statefulObserver.mState.compareTo(targetState) < 0  
            && mObserverMap.contains(observer))) {  
        pushParentState(statefulObserver.mState);  
        statefulObserver.dispatchEvent(lifecycleOwner,  
upEvent(statefulObserver.mState));  
        popParentState();  
        // 重新计算状态，用于循环退出条件：直到observer的状态从INITIALIZED的状态递进到当前  
LifecycleOwner的状态  
        targetState = calculateTargetState(observer);  
    }  
  
    if (!isReentrance) {  
        sync();  
    }  
    mAddingObserverCounter--;  
}
```

在上面的源码中，代码的来前几行是将 state与observer 包装成ObserverWithState类型，state 的初始值为 INITIALIZED，然后存入集合，如果observer之前已经存在的话，就认定重复添加，直接返回。当添加的observer为新的时候，执行循环流程。接着判断了一下isReentrance的值，表示是否重入，即是否需要同时执行添加 addObserver()的流程或者同时有其他Event事件正在分发。然后，在while循环中，执行事件的分发逻辑。while循环中有两个比较重要的方法：`dispatchEvent()` 和 `upEvent()`。

首先，我们来看一下 `dispatchEvent()` 方法的源码，如下所示。

```
static class ObserverWithState {  
    State mState;  
    LifecycleEventObserver mLifecycleObserver;
```

```

    observerWithState(LifecycleObserver observer, State initialState) {
        mLifecycleObserver = Lifecycling.lifecycleEventObserver(observer);
        mState = initialState;
    }

    void dispatchEvent(LifecycleOwner owner, Event event) {
        State newState = getStateAfter(event);
        mState = min(mState, newState);
        // observer的回调函数
        mLifecycleObserver.onStateChanged(owner, event);
        mState = newState;
    }
}

```

然后再调用了 observer的回调方法onStateChanged()更新组件的状态mState。

8.3.2 通知观察者

前面我们分析了Lifecycle的注册观察者的流程，接下来我们看一下Lifecycle又是如何通知Activity或Fragment的生命周期改变的呢？在Android 8.0时，FragmentActivity继承自SupportActivity，而在Android 9.0，FragmentActivity继承自ComponentActivity。SupportActivity和ComponentActivity的代码区别不大，以ComponentActivity来说，源码如下。

```

public class ComponentActivity extends androidx.core.app.ComponentActivity implements
    LifecycleOwner,
    ViewModelStoreOwner,

    ... //省略其他代码

private final LifecycleRegistry mLifecycleRegistry = new LifecycleRegistry(this);

@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ... //省略其他代码

    ReportFragment.injectIfNeededIn(this);
    if (mContentLayoutId != 0) {
        setContentView(mContentLayoutId);
    }
}
...
... //省略其他代码
}

```

接下来，我们看一下ReportFragment的源码。

```

public class ReportFragment extends Fragment {

    public static void injectIfNeededIn(Activity activity) {

```

```

        android.app.FragmentManager manager = activity.getFragmentManager();
        if (manager.findFragmentByTag(REPORT_FRAGMENT_TAG) == null) {
            manager.beginTransaction().add(new ReportFragment(),
REPORT_FRAGMENT_TAG).commit();
            manager.executePendingTransactions();
        }
    }

    ...

@Override
public void onActivityCreated(Bundle savedInstanceState) {
    dispatch(Lifecycle.Event.ON_CREATE);
}

@Override
public void onStart() {
    dispatch(Lifecycle.Event.ON_START);
}

... //省略其他代码

private void dispatch(Lifecycle.Event event) {
    Activity activity = getActivity();
    if (activity instanceof LifecycleRegistryOwner) {
        ((LifecycleRegistryOwner) activity).getLifecycle().handleLifecycleEvent(event);
        return;
    }

    if (activity instanceof LifecycleOwner) {
        Lifecycle lifecycle = ((LifecycleOwner) activity).getLifecycle();
        if (lifecycle instanceof LifecycleRegistry) {
            ((LifecycleRegistry) lifecycle).handleLifecycleEvent(event);
        }
    }
}

```

在上面的ReportFragment类中，Activity 通过注入一个没有UI的 ReportFragment ，然后在 ReportFragment的的生命周期回调中调用dispatch() 方法分发生命周期状态的改变。因为Fragment依赖于创建它的Activity，所以 Fragment的生命周期和Activity生命周期同步，这样就间接实现了 Lifecycle 监听Activity生命周期的功能。接下来，看一下是dispatch()方法是如何分发Event的，源码如下。

```

private void dispatch(Lifecycle.Event event) {
    Activity activity = getActivity();
    if (activity instanceof LifecycleRegistryOwner) {
        ((LifecycleRegistryOwner) activity).getLifecycle().handleLifecycleEvent(event);
        return;
    }

    if (activity instanceof LifecycleOwner) {
        Lifecycle lifecycle = ((LifecycleOwner) activity).getLifecycle();

```

```

        if (lifecycle instanceof LifecycleRegistry) {
            ((LifecycleRegistry) lifecycle).handleLifecycleEvent(event);
        }
    }
}

```

可以发现，在调用getActivity()方法后向上转型强制转换为LifecycleOwner，然后调用了LifecycleRegistry类的handleLifecycleEvent()方法，然后逻辑又回到了LifecycleRegistry类中，进而将事件Event分发交由LifecycleRegistry进行处理。其中，handleLifecycleEvent(event)的实现如下所示。

```

public void handleLifecycleEvent(@NonNull Lifecycle.Event event) {
    State next = getStateAfter(event);
    moveToState(next);
}

private void moveToState(State next) {

    // 将 mState 更新为当前的 state
    mState = next;
    ...

    mHandlingEvent = true;
    sync();
    mHandlingEvent = false;
}

```

更新了mState的值之后，就调用sync()方法，sync()方法就是根据 mState 的改变做出同步操作，并执行分发事件，sync()方法的源码如下。

```

private void sync() {
    LifecycleOwner lifecycleOwner = mLifecycleOwner.get();

    // 判断是否需要同步，没有同步则一直进行
    while (!isSynced()) {
        mNewEventOccurred = false;
        if (mState.compareTo(mObserverMap.eldest().getValue().mState) < 0) {
            //同步并分发事件
            backwardPass(lifecycleOwner);
        }
        Entry<LifecycleObserver, ObserverWithState> newest = mObserverMap.newest();
        if (!mNewEventOccurred && newest != null
            && mState.compareTo(newest.getValue().mState) > 0) {
            //同步并分发事件
            forwardPass(lifecycleOwner);
        }
    }
    mNewEventOccurred = false;
}

private boolean isSynced() {
    if (mObserverMap.size() == 0) {
        return true;
    }
}

```

```

        State eldestObserverState = mObserverMap.eldest().getValue().mState;
        State newestObserverState = mObserverMap.newest().getValue().mState;
        return eldestObserverState == newestObserverState && mState == newestObserverState;
    }
}

```

首先，通过调用isSynced()方法来判断是否需要同步。isSynced()方法比较简单，主要通过比较第一个observer和最后一个observer，他们的mState值是否相等，相等的话则说明同步完毕，不相等的话继续同步，直到相等为止。

同时，sync()方法中会根据当前状态和mObserverMap中的eldest和newest的状态做对比，判断当前状态是向前还是向后，以向后为例。

```

private void forwardPass(LifecycleOwner lifecycleowner) {
    Iterator<Entry<LifecycleObserver, ObserverWithState>> ascendingIterator =
        mObserverMap.iteratorWithAdditions();
    while (ascendingIterator.hasNext() && !mNewEventOccurred) {
        Entry<LifecycleObserver, ObserverWithState> entry = ascendingIterator.next();
        ObserverWithState observer = entry.getValue(); //1
        while ((observer.mState.compareTo(mState) < 0 && !mNewEventOccurred
                && mObserverMap.contains(entry.getKey()))) {
            pushParentState(observer.mState);
            observer.dispatchEvent(lifecycleowner, upEvent(observer.mState)); //2
            popParentState();
        }
    }
}

```

forwardPass()方法最核心的就是获取ObserverWithState状态，ObserverWithState的代码如下。

```

static class ObserverWithState {
    State mState;
    GenericLifecycleObserver mLifecycleObserver;

    ObserverWithState(LifecycleObserver observer, State initialState) {
        mLifecycleObserver = Lifecycling.getCallback(observer); //1
        mState = initialState;
    }

    void dispatchEvent(LifecycleOwner owner, Event event) {
        State newState = getStateAfter(event);
        mState = min(mState, newState);
        mLifecycleObserver.onStateChanged(owner, event);
        mState = newState;
    }
}

```

ObserverWithState类包括了State和GenericLifecycleObserver成员变量，GenericLifecycleObserver是一个接口，它继承了LifecycleObserver接口。ReflectiveGenericLifecycleObserver和CompositeGeneratedAdaptersObserver是GenericLifecycleObserver的实现类，这里主要查看ReflectiveGenericLifecycleObserver的onStateChanged方法是如何实现的，源码如下。

```

class ReflectiveGenericLifecycleobserver implements GenericLifecycleobserver {
    private final Object mWrapped;
    private final CallbackInfo mInfo;

    ReflectiveGenericLifecycleobserver(Object wrapped) {
        mWrapped = wrapped;
        mInfo = ClassesInfoCache.sInstance.getInfo(mWrapped.getClass());
    }

    @Override
    public void onStateChanged(LifecycleOwner source, Event event) {
        mInfo.invokeCallbacks(source, event, mWrapped); //1
    }
}

```

onStateChanged()方法会调用CallbackInfo的invokeCallbacks()方法，这里会用到CallbackInfo类，代码如下。

```

static class CallbackInfo {
    final Map<Lifecycle.Event, List<MethodReference>> mEventToHandlers;
    final Map<MethodReference, Lifecycle.Event> mHandlerToEvent;
    CallbackInfo(Map<MethodReference, Lifecycle.Event> handlerToEvent) {
        mHandlerToEvent = handlerToEvent;
        mEventToHandlers = new HashMap<>();
        for (Map.Entry<MethodReference, Lifecycle.Event> entry :
    handlerToEvent.entrySet()) { //1
            Lifecycle.Event event = entry.getValue();
            List<MethodReference> methodReferences = mEventToHandlers.get(event);
            if (methodReferences == null) {
                methodReferences = new ArrayList<>();
                mEventToHandlers.put(event, methodReferences);
            }
            methodReferences.add(entry.getKey());
        }
    }
    @SuppressWarnings("ConstantConditions")
    void invokeCallbacks(LifecycleOwner source, Lifecycle.Event event, Object target) {
        invokeMethodsForEvent(mEventToHandlers.get(event), source, event, target); //2
        invokeMethodsForEvent(mEventToHandlers.get(Lifecycle.Event.ON_ANY), source,
event,
                                target);
    }

    private static void invokeMethodsForEvent(List<MethodReference> handlers,
                                              LifecycleOwner source, Lifecycle.Event event, Object mWrapped) {
        if (handlers != null) {
            for (int i = handlers.size() - 1; i >= 0; i--) {
                handlers.get(i).invokeCallback(source, event, mWrapped); //1
            }
        }
    }
}

```

在CallbackInfo代码中，首先使用循环将handlerToEvent进行数据类型转换，转化为一个HashMap，key的值为事件，value的值为MethodReference。而invokeMethodsForEvent()方法会传入mEventToHandlers.get(event)，也就是事件对应的MethodReference的集合。然后，invokeMethodsForEvent方法中会遍历MethodReference的集合，调用MethodReference的invokeCallback方法。其中，MethodReference类的代码如下。

```
@SuppressWarnings("WeakerAccess")
static class MethodReference {
    final int mCallType;
    final Method mMethod;
    MethodReference(int callType, Method method) {
        mCallType = callType;
        mMethod = method;
        mMethod.setAccessible(true);
    }
    void invokeCallback(LifecycleOwner source, Lifecycle.Event event, Object target) {
        try {
            switch (mCallType) {
                case CALL_TYPE_NO_ARG:
                    mMethod.invoke(target);
                    break;
                case CALL_TYPE_PROVIDER:
                    mMethod.invoke(target, source);
                    break;
                case CALL_TYPE_PROVIDER_WITH_EVENT:
                    mMethod.invoke(target, source, event);
                    break;
            }
        } catch (InvocationTargetException e) {
            throw new RuntimeException("Failed to call observer method", e.getCause());
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
}
...
}
```

MethodReference类中有两个变量，一个是callType，它代表调用方法的类型，另一个是Method，它代表方法，并最终通过invoke对方法进行反射，通过反射对事件的对应方法进行调用。

9. Android Jetpack Compose 最全上手指南

9.1 声明式 UI 的前世今生

其实声明式 UI 并不是什么新技术，早在 2006 年，微软就已经发布了其新一代界面开发框架 WPF，其采用了 XAML 标记语言，支持双向数据绑定、可复用模板等特性。

2010 年，由诺基亚领导的 Qt 团队也正式发布了其下一代界面解决方案 Qt Quick，同样也是声明式，甚至 Qt Quick 起初的名字就是 Qt Declarative。QML 语言同样支持数据绑定、模块化等特性，此外还支持内置 JavaScript，开发者只用 QML 就可以开发出简单的带交互的原型应用。

声明式 UI 框架近年来飞速发展，并且被 Web 开发带向高潮。React 更是为声明式 UI 奠定了坚实基础并一直引领其未来的发展。随后 Flutter 的发布也将声明式 UI 的思想成功带到移动端开发领域...

声明式UI的意思就是，描述你想要一个什么样的UI界面，状态变化时，界面按照先前描述的重新“渲染”即可得到状态绝对正确的界面，而不用像命令一样，告诉程序一步一步该干什么，维护各种状态。扯远了，这个并不是今天文章的重点，稍微了解一下就好，其他的就不在本文延伸。关于声明式的更多介绍，建议看看这篇文章：<https://zhuanlan.zhihu.com/p/68275232>

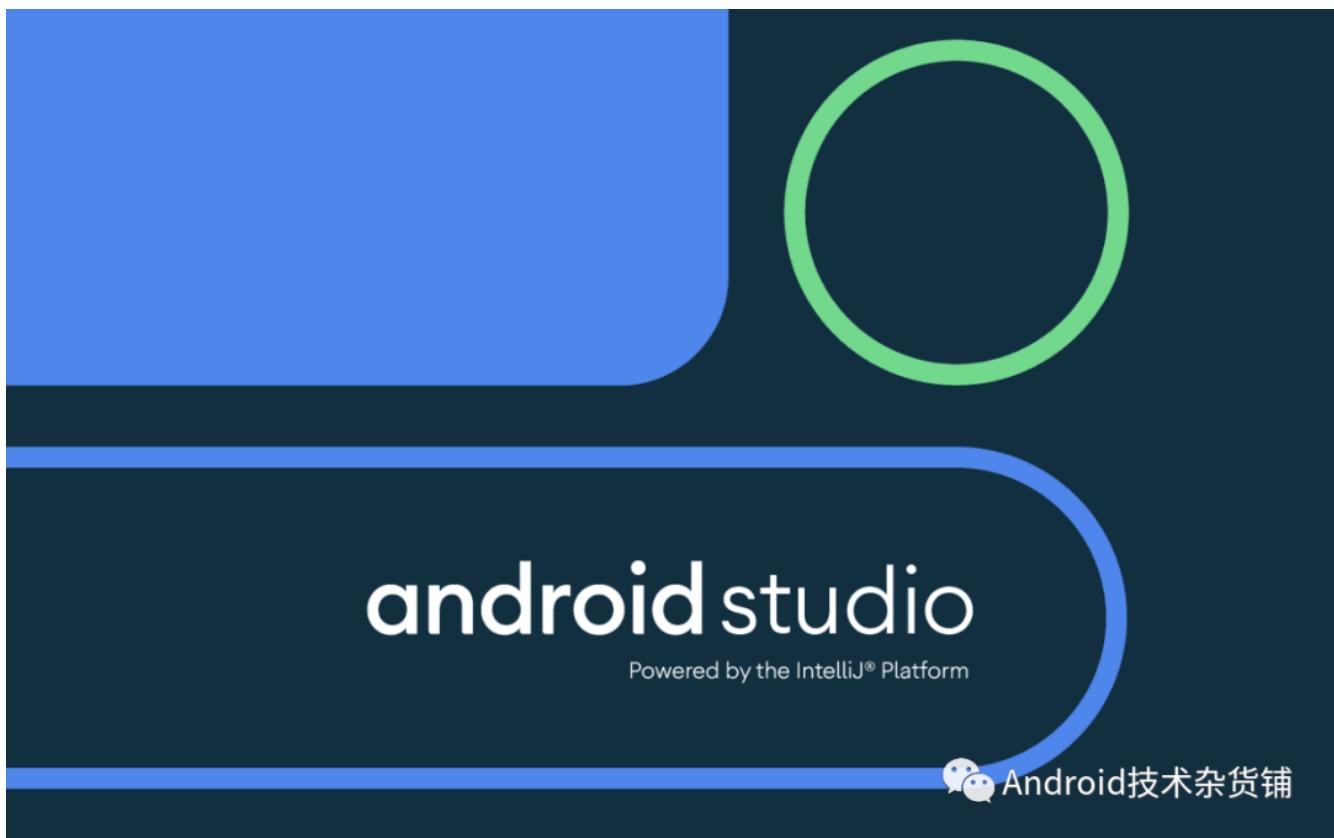
我们回到本文的重点Jetpack Compose。

9.2 Jetpack Compose 介绍

Jetpack Compose 是一个用于构建原生Android UI 的现代化工具包，它基于声明式的编程模型，因此你可以简单地描述UI的外观，而Compose则负责其余的工作-当状态发生改变时，你的UI将自动更新。由于Compose基于Kotlin构建，因此可以与Java编程语言完全互操作，并且可以直接访问所有Android和Jetpack API。它与现有的UI工具包也是完全兼容的，因此你可以混合原来的View和现在新的View，并且从一开始就使用Material和动画进行设计。

9.3 Jetpack Compose 环境准备和Hello World

每当我们学习一门新的语言，我们都是从一个 `hello world` 开始，今天我们也从一个 `hello world` 来开始Jetpack Compose 吧！要想获得Jetpack Compose 的最佳体验，我们需要下载最新版本的Android Studio 预览版本（即 **Android Studio 4.0**）。因为Android Studio 4.0 添加了对Jetpack Compose 的支持，如新的Compose 模版和Compose 及时预览。



Android Studio 4.0.png

使用Jetpack Compose 来开始你的开发工作有2种方式：

- 将Jetpack Compose 添加到现有项目
- 创建一个支持Jetpack Compose的新应用

接下来分别介绍一下这两种方式。

1. 将Jetpack Compose 添加到现有项目

如果你想在现有的项目中使用Jetpack Compose，你需要配置一些必须的设置和依赖：

(1) gradle 配置

在app目录下的 `build.gradle` 中将app支持的最低API 版本设置为21或更高，同时开启Jetpack Compose `enable` 开关，代码如下：

```
android {  
    defaultConfig {  
        ...  
        minSdkVersion 21  
    }  
  
    buildFeatures {  
        // Enables Jetpack Compose for this module  
        compose true  
    }  
    ...  
  
    // Set both the Java and Kotlin compilers to target Java 8.  
  
    compileOptions {  
        sourceCompatibility JavaVersion.VERSION_1_8  
        targetCompatibility JavaVersion.VERSION_1_8  
    }  
  
    kotlinOptions {  
        jvmTarget = "1.8"  
    }  
}
```

(2) 使用试验版 Kotlin-Gradle 插件

Jetpack Compose 需要试验版的 `Kotlin-Gradle` 插件，在根目录下的 `build.gradle` 添加如下代码：

```
buildscript {  
    repositories {  
        google()  
        jcenter()  
        // To download the required version of the Kotlin-Gradle plugin,  
        // add the following repository.  
        maven { url 'https://dl.bintray.com/kotlin/kotlin-eap' }  
        ...  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:4.0.0-alpha01'  
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.60-eap-25'  
    }  
}  
  
allprojects {
```

```
repositories {  
    google()  
    jcenter()  
    maven { url 'https://dl.bintray.com/kotlin/kotlin-eap' }  
}  
}
```

(3) 添加Jetpack Compose工具包依赖项

在app目录下的 `build.gradle` 添加Jetpack Compose 工具包依赖项，代码如下：

```
dependencies {  
    // You also need to include the following Compose toolkit dependencies.  
    implementation 'androidx.ui:ui-tooling:0.1.0-dev02'  
    implementation 'androidx.ui:ui-layout:0.1.0-dev02'  
    implementation 'androidx.ui:ui-material:0.1.0-dev02'  
    ...  
}
```

ok，到这儿准备工作就完毕，就可以开始写代码了，但是前面说了，还有一种方式接入Jetpack Compose，我们来一起看看。

2. 创建一个支持Jetpack Compose的新应用

比起在现有应用中接入Jetpack Compose，创建一个支持Jetpack Compose 的新项目则简单了许多，因为Android Studio 4.0 提供了一个新的Compose 模版，只要选择这个模版创建应用，则所有上面的那些配置项都自动帮我们完成了。

创建一个支持Jetpack Compose 的应用，如下几个步骤就可以了：

- 1.如果你在Android Studio的欢迎窗口，点击 `Start a new Android studio project`，如果你已经打开了Android Studio 项目，则在顶部菜单栏选择 `File > New > New Project`
- \2. 在 `Select a Project Template` 窗口，选择 `Empty Compose Activity` 并且点击下一步
- \3. 在 `Configure your project` 窗口，做如下几步：
 - a. 设置项目名称，包名 和 保存位置
- b. 注意，在语言下来菜单中，**Kotlin** 是唯一一个可选项，因为Jetpack Compose 只能用Kotlin来写的才能运行。
- c. `Minimum API level` 下拉菜单中，选择21或者更高
- 4点击 `Finish`
- ``

现在，你就可以使用Jetpack Compose 来编写你的应用了。

3. Hello world

在 `MainActivity.kt` 中添加如下代码：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello, Android技术杂货铺")
        }
    }
}
```

hello.png

Jetpack Compose是围绕 composable 函数来构建的。这些函数使你可以通过描述应用程序的形状和数据依赖，以编程方式定义应用程序的UI，而不是着眼于UI的构建过程。要创建 composable 函数，只需要在函数名前面加上一个 @composable 注解即可，上面的 Text 就是一个 composable 函数。

4. 定义一个 composable 函数

一个 composable 函数只能在另一个 composable 函数的作用域里被调用，要使一个函数变为 composable 函数，只需在函数名前加上 @composable 注解，我们把上面的代码中， setContent 中的部分移到外面，抽取到一个 composable 函数中，然后传递一个参数 name 给 text 元素。

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Greeting("Android技术杂货铺")
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```



9.4 布局

UI元素是分层级的，元素包含在其他元素中。在Jetpack Compose中，你可以通过从其他 `composable` 函数中调用 `composable` 函数来构建UI层次结构。

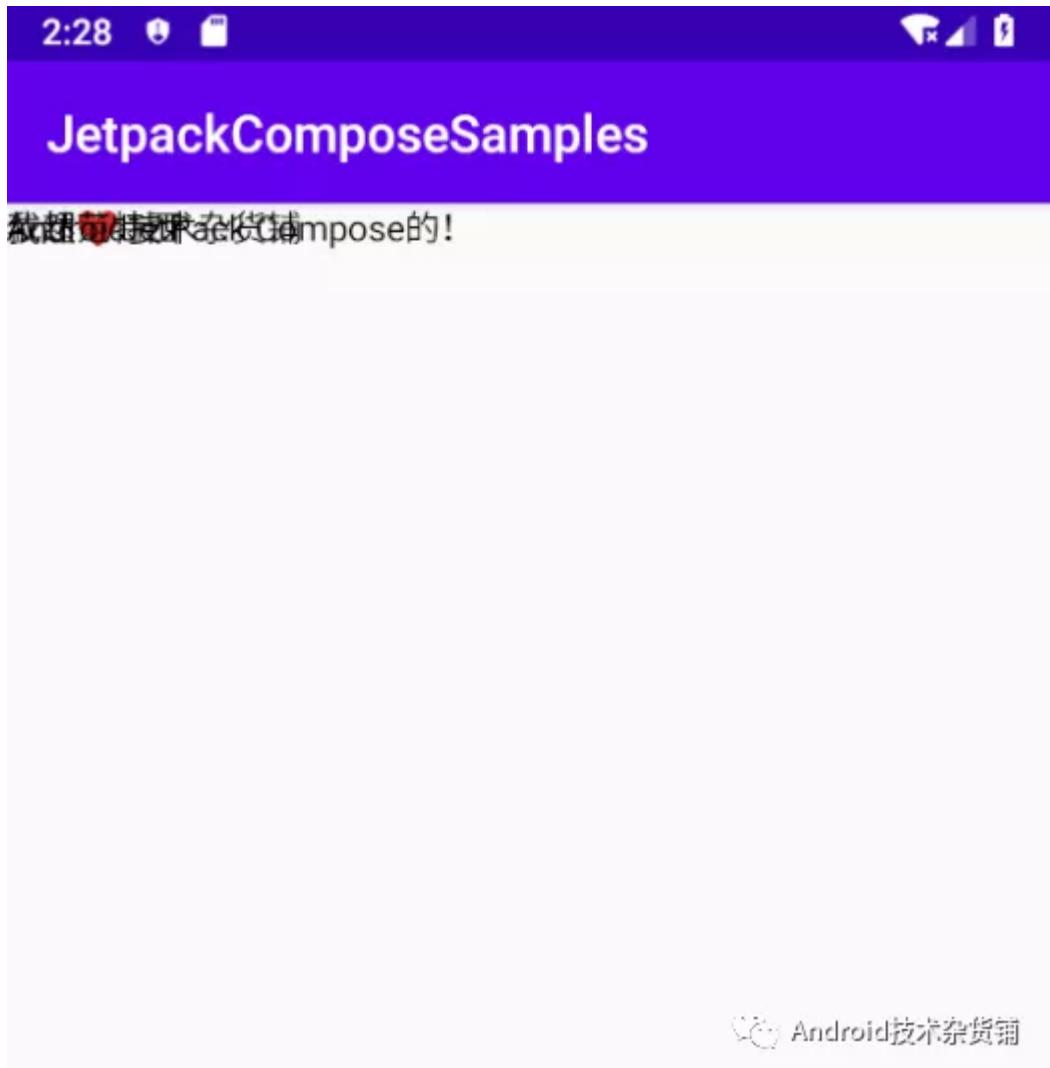
在Android的 `xml` 布局中，如果要显示一个垂直结构的布局，最常用的就是 `LinearLayout`，设置 `android:orientation` 值为 `vertical`，子元素就会垂直排列，那么，在Jetpack Compose 中，如何来实现垂直布局呢？先添加几个 `Text` 来看一下。

1. 添加多个Text

在上面的例子中，我们添加了一个 `Text` 显示文本，现在我们添加三个文本，代码如下：

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView {
            NewsStory()
        }
    }
}

@Composable
fun NewsStory() {
    Text("我超❤️JetPack Compose的！")
    Text("Android技术杂货铺")
    Text("依然范特西")
}
```



从上图可以看到，我们添加了3个文本，但是，由于我们还没有提供有关如何排列它们的任何信息，因此三个文本元素相互重叠绘制，使得文本不可读。

2. 使用Column

要使重叠绘制的 `Text` 文本能够垂直排列，我们就需要使用到 `Column` 函数，**写过flutter的同学看起来是不是很眼熟？是的，跟flutter里面的Column Widget 名字和功能完全一样，甚至连他们的属性都一摸一样。**

```
@Composable
fun NewsStory() {
    Column { // 添加Column，使布局垂直排列
        Text("我超❤️JetPack Compose的!")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

效果如下：



可以看到，前面重叠的布局，现在已经垂直排列了，但是，默认情况下，从左上角开始，一个接一个的排列，没有任何间距。接下来，我们给 `Column` 设置一些样式。

3. 给 `Column` 添加样式

在调用 `Column()` 时，可以传递参数给 `Column()` 来配置 `Column` 的大小、位置以及设置子元素的排列方式。

```
@Composable
fun NewsStory() {
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

如上图所示，我们填充了padding，其他效果几乎一摸一样，上面代码中的设置属性解释如下：

- `crossAxisSize`: 指定 `Column` 组件（注：Compose中，所有的组件都是 `composable` 函数，文中的组件都是指代 `composable` 函数）在水平方向的大小，设置 `crossAxisSize` 为 `LayoutSize.Expand` 即表示 `Column` 宽度应为其父组件允许的最大宽度，相当于传统布局中的 `match_parent`，还有一个值为 `Layoutsize.Wrap`，看名字就知道，包裹内容，相当于传统布局中的 `wrap_content`。
- `modifier`: 使你可以进行其他格式更改。在这种情况下，我们将应用一个 `spacing` 修改器，该设置将 `Column` 与周围的视图产生间距。

4. 如何显示一张图片？

在原来的安卓原生布局中，显示图片有相应的控件 `ImageView`,设置本地图片地址或者 `Bitmap` 就能展示，在Jetpack Compose 中该如何显示图片呢？



Android技术杂货铺

我们先下载这张图片到本地，添加到资源管理器中，命名为 `header.png`，我们更改一下上面的 `NewsStory()` 方法，先从资源文件夹获取图片 `image`，然后通过 `DrawImage()` 将图片绘制出来：

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 显示图片
        DrawImage(image)

        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

3:57



JetpackComposeSamples

我超❤️JetPack Compose的！

Android技术杂货铺

依然范特西



Android技术杂货铺

可以看到，图片不会按正确的比例显示，接下来，我们来修复它。

图片已添加到布局中，但会展开以填充整个视图，并和文本是拼叠排列，文字显示在上层。要设置图形样式，请将其放入 `Container`(又一个和flutter中一样的控件)

- `Container`: 一个通用的内容对象，用于保存和安排其他UI元素。然后，你可以将大小和位置的设置应用于容器。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 放在容器中，设置大小
        Container(expanded = true, height = 180.dp) {
            // 显示图片
            DrawImage(image)
        }
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

image.png

- `expanded` : 指定`Container`的大小，默认是 `false` (`Container`的大小是子组件的大小，相当于 `wrap_content`)，如果将它设置为 `true`，就指定`Container`的大小为父控件所允许的最大size, 相当于 `match_parent`。
- `height` : 设置`Container`容器的高度，`height`属性的优先级高于 `expanded`，因此会覆盖 `expanded`，如上面的例子，设置 `height` 为 `180dp`, 也就是容器宽为父控件宽度，高为 `180dp`

5. 添加间距 `Spacer`

我们看到，图片和文本之间没有间距，传统布局中，我们可以添加 `Margin` 属性，设置间距，在Jetpack Compose 中，我们可以使用 `HeightSpacer()` 和 `widthSpacer()` 来设置垂直和水平间距

```
HeightSpacer(height = 20.dp) //设置垂直间距20dp
widthSpacer(width = 20.dp) // 设置水平间距20dp
```

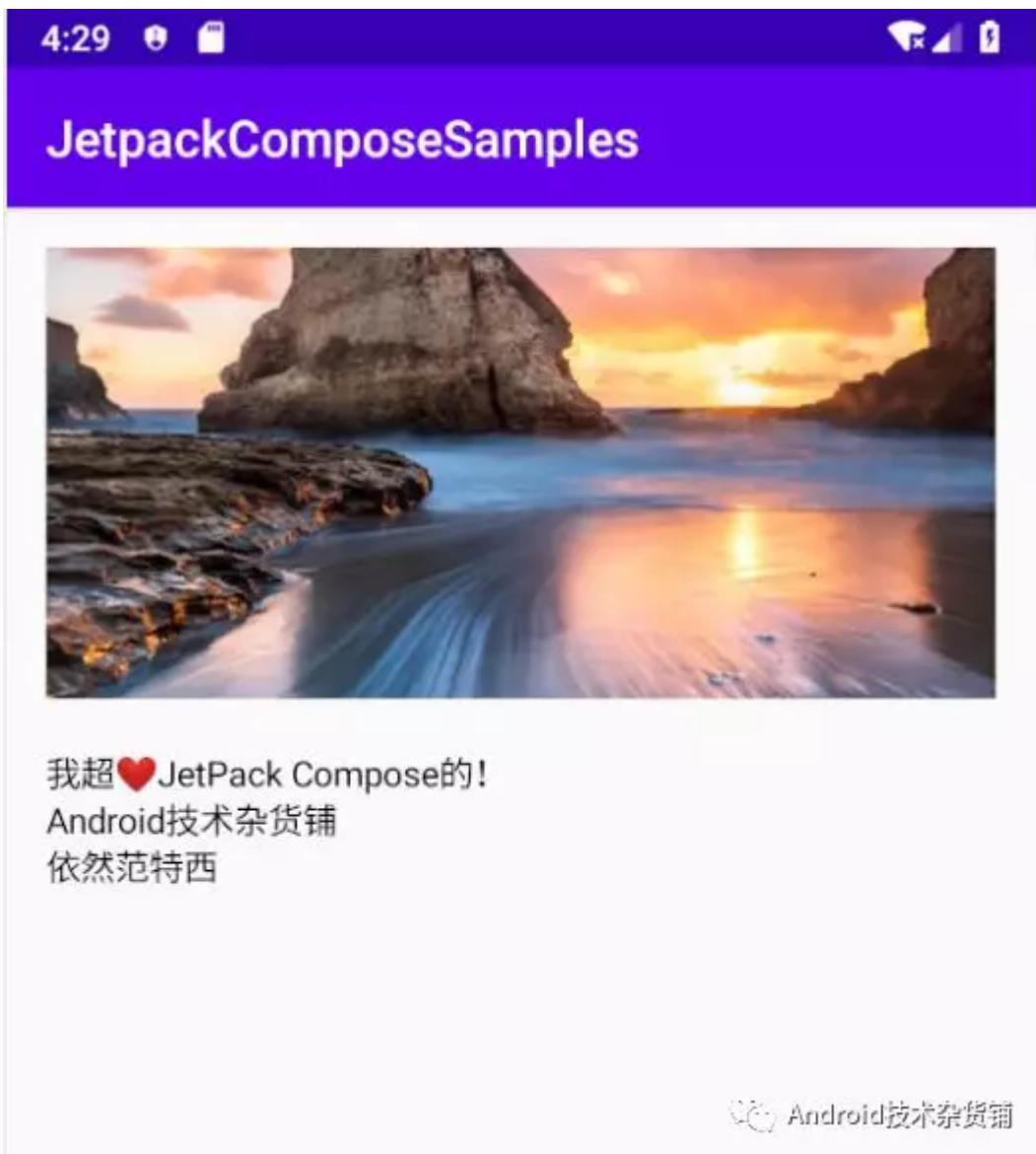
在上面的例子中，我们来为图片和文本之间添加 `20dp` 的间距：

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
```

```
// 放在容器中，设置大小
Container(expanded = true, height = 180.dp) {
    // 显示图片
    DrawImage(image)
}

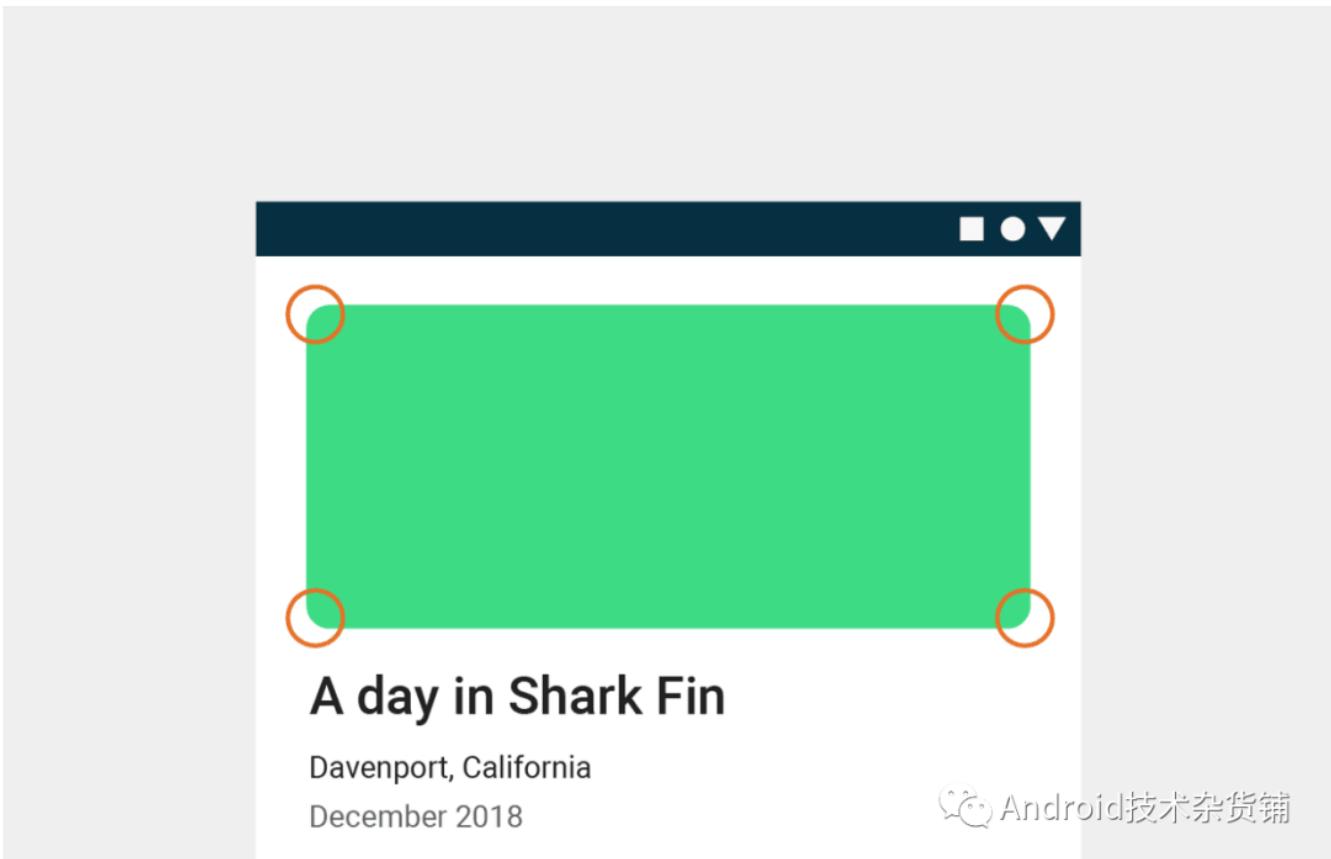
HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

Text("我超❤JetPack Compose的！")
Text("Android技术杂货铺")
Text("依然范特西")
}
```



9.5 使用Material design 设计

Compose 旨在支持Material Design 设计原则，许多组件都实现了Material Design 设计，可以开箱即用，在这一节中，将使用一些Material小组件来对app进行样式设置



1. 添加 Shape 样式

Shape 是Material Design 系统中的支柱之一，我们来用 clip 函数对图片进行圆角裁剪。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    Column (
        crossAxisSize = LayoutSize.Expand,
        modifier = Spacing(16.dp)
    ) { // 添加Column，使布局垂直排列
        // 放在容器中，设置大小
        Container(expanded = true, height = 180.dp) {
            clip(shape = RoundedCornerShape(10.dp)) {
                // 显示图片
                DrawImage(image)
            }
        }
        HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp
        Text("我超❤️JetPack Compose的！")
        Text("Android技术杂货铺")
        Text("依然范特西")
    }
}
```

形状是不可见的，但是我们的图片已经被裁剪成了设置的形状样式，因此如上图，图片已经有圆角了。

2. 给 Text 添加一些样式

通过Compose，可以轻松利用Material Design原则。将 `MaterialTheme()` 应用于创建的组件

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    // 使用Material Design 设计
    MaterialTheme() {
        Column (
            crossAxisSize = LayoutSize.Expand,
            modifier = Spacing(16.dp)
        ) { // 添加Column，使布局垂直排列
            // 放在容器中，设置大小
            Container(expanded = true, height = 180.dp) {
                Clip(shape = RoundedCornerShape(10.dp)) {
                    // 显示图片
                    DrawImage(image)
                }
            }

            HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

            Text("我超❤️JetPack Compose的！")
            Text("Android技术杂货铺")
            Text("依然范特西")
        }
    }
}
```

如上面的代码，添加了 `MaterialTheme` 后，重新运行，效果没有任何变化，文本现在使用了 `MaterialTheme` 的默认文本样式。接下来，我们将特定的段落样式应用于每个文本元素。

```
@Composable
fun NewsStory() {
    // 获取图片
    val image = +imageResource(R.drawable.header)
    // 使用Material Design 设计
    MaterialTheme() {
        Column (
            crossAxisSize = LayoutSize.Expand,
            modifier = Spacing(16.dp)
        ) { // 添加Column，使布局垂直排列
            // 放在容器中，设置大小
            Container(expanded = true, height = 180.dp) {
                Clip(shape = RoundedCornerShape(10.dp)) {
                    // 显示图片
                    DrawImage(image)
                }
            }

            HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

            Text("我超❤️JetPack Compose的！")
            Text("Android技术杂货铺")
            Text("依然范特西")
        }
    }
}
```

```
        }

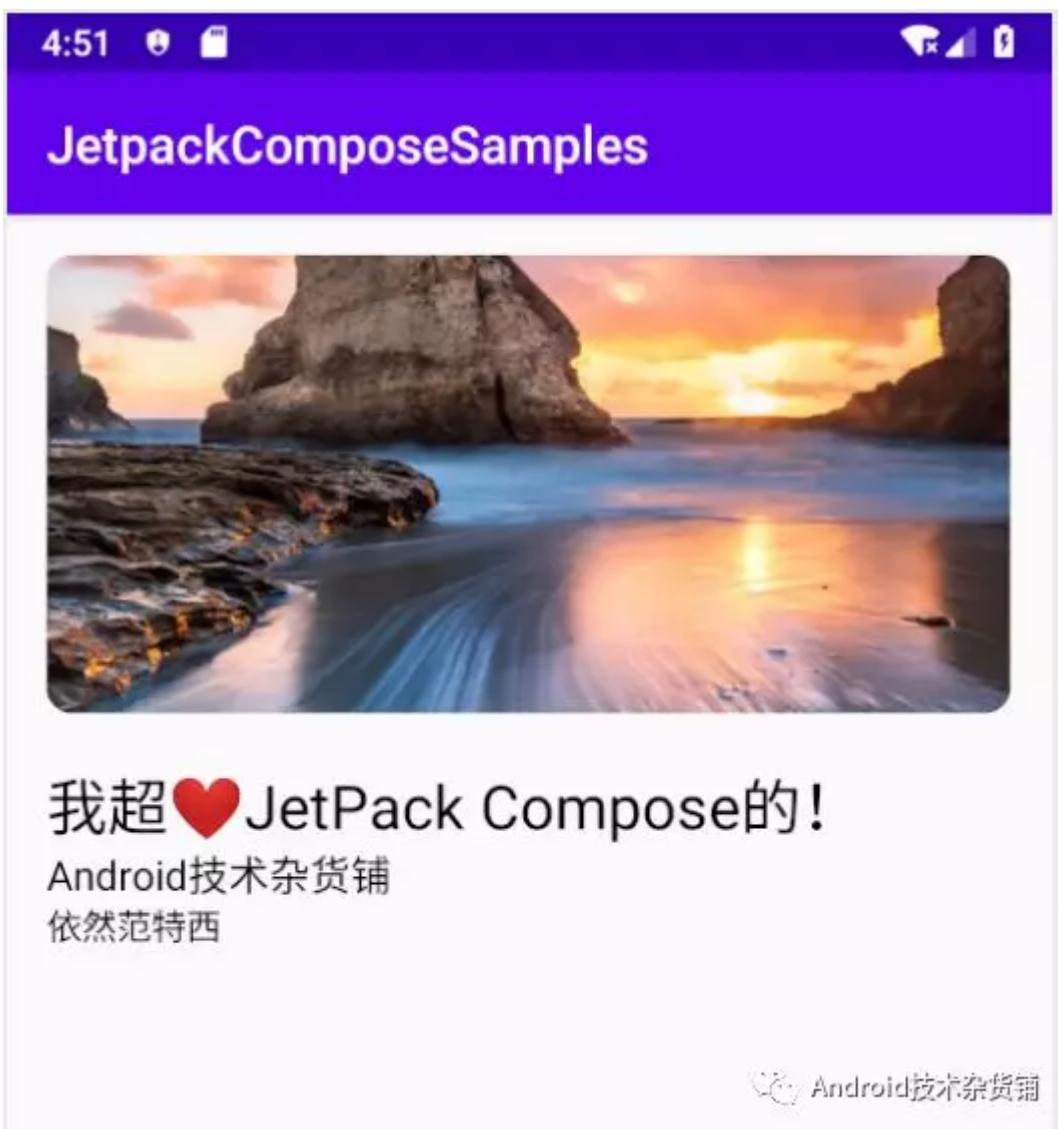
    }

    HeightSpacer(height = 20.dp) // 添加垂直方向间距20dp

    Text("我超❤JetPack Compose的！", style = +themeTextStyle { h5 }) // 注意添加了style
    Text("Android技术杂货铺", style = +themeTextStyle { body1 }) // 注意添加了style
    Text("依然范特西", style = +themeTextStyle { body2 }) // 注意添加了style
}

}

}
```



现在看看，我们的文本样式已经有变化了，标题有6中样式 `h1-h6`，其实 HTML 中的样式很像，内容文本有 `body1` 和 `body2` 2中样式。

Material 调色版使用了一些基本颜色，如果要强调文本，可以调整文本的不透明度：

```
Text("我超❤JetPack Compose的！", style = (+themeTextStyle { h5 }).withOpacity(0.87f))
Text("Android技术杂货铺", style = (+themeTextStyle { body1 }).withOpacity(0.87f))
Text("依然范特西", style = (+themeTextStyle { body2 }).withOpacity(0.6f))
```

5:05



JetpackComposeSamples



我超❤️JetPack Compose的！

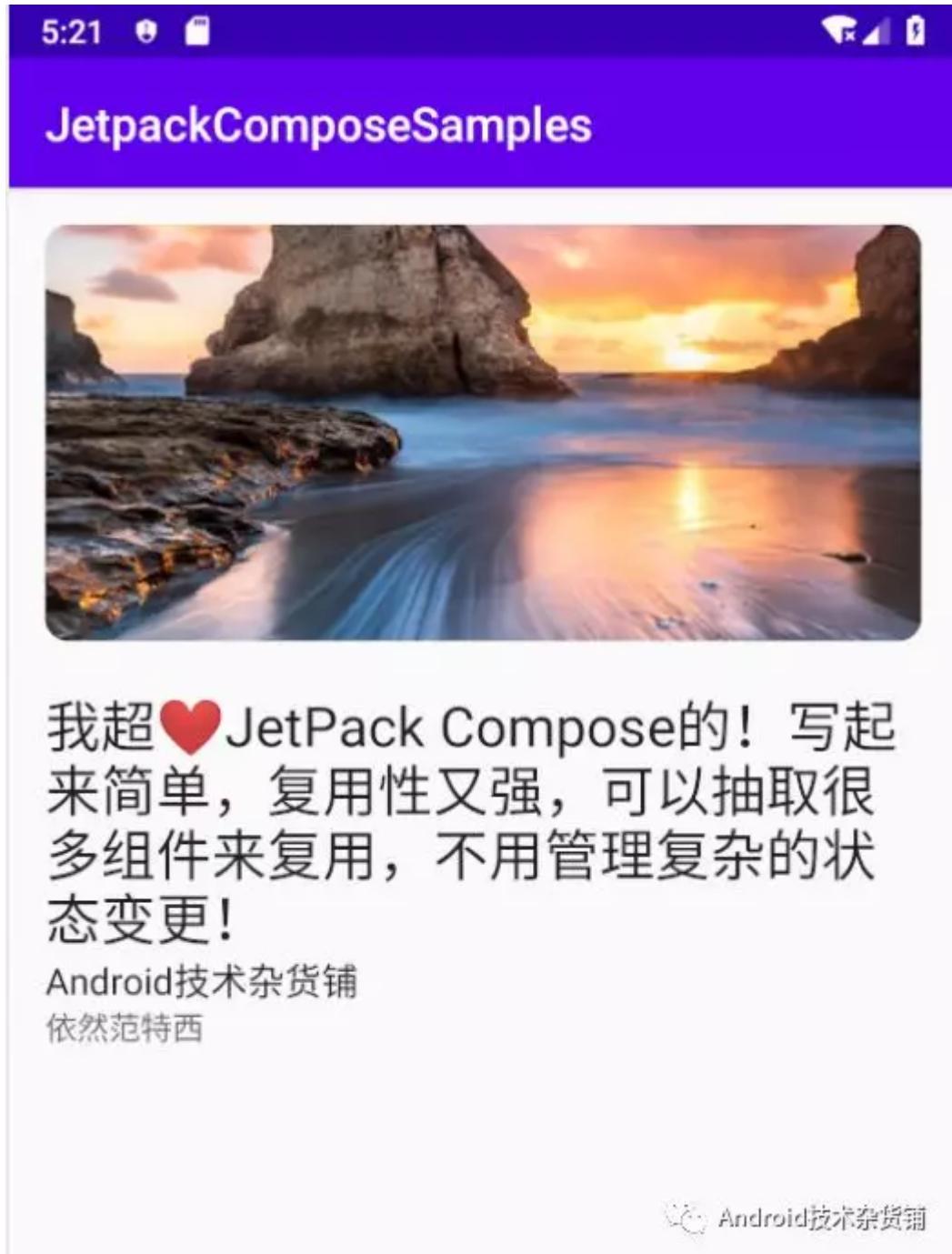
Android技术杂货铺

依然范特西



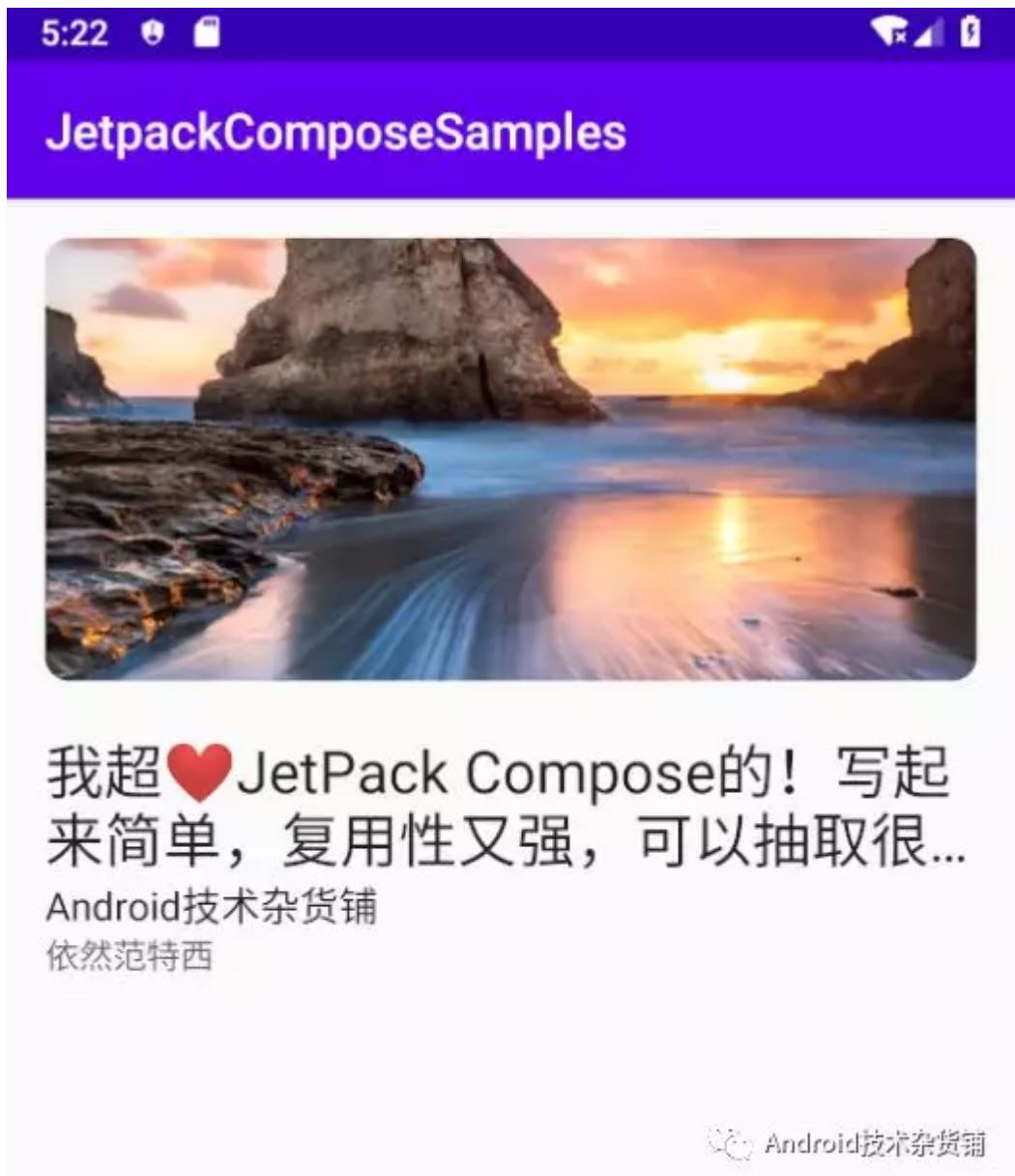
Android技术杂货铺

有些时候，标题很长，但是我们又不想长标题换行从而影响我们的app UI，因此，我们可以设置文本的最大显示行数，



如本例所示，我们设置显示最大行数为 2，多于的部分截断处理：

```
Text("我超❤️JetPack Compose的！写起来简单，复用性又强，可以抽取很多组件来复用，不用管理复杂的状态变更！",  
    maxLines = 2, overflow = TextOverflow.Ellipsis,  
    style = (+themeTextStyle { h5 }).withOpacity(0.87f))
```



可以看到，设置了 `maxLines` 和 `overflow` 之后，超出部分就截断处理了，不会影响到整个布局样式。

9.6 Compose 布局实时预览

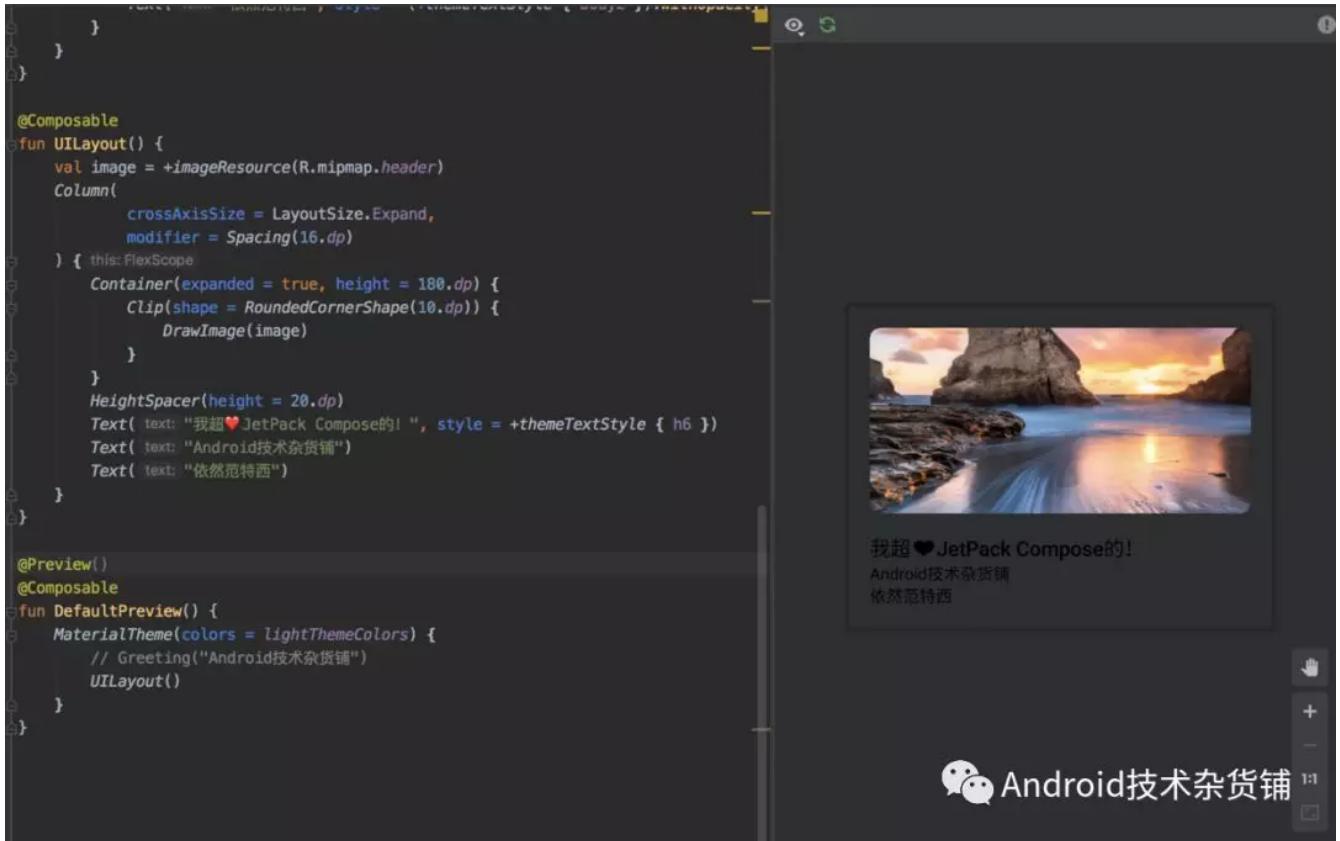
从Android Studio 4.0 开始，提供了在IDE中预览 `composable` 函数的功能，不用像以前那样，要先下载一个模拟器，然后将app状态模拟器上，运行app才能看到效果。

但是有一个限制，那就是`composable`函数不能有参数

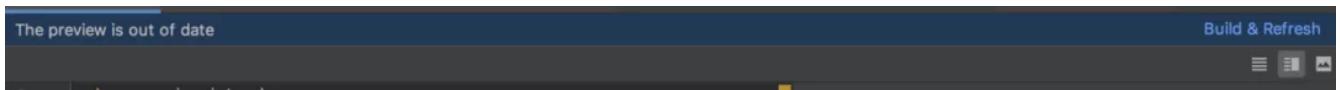
满足下面两个条件：

- 函数没有参数
- 在函数前面添加 `@Preview` 注解

预览效果图如下：



当布局改变了之后，顶部会出现一个导航条，显示预览已经过期，点击 build&Refresh 就可以刷新预览



这真的是一个非常棒的功能，像其他声明式布局，如React、flutter 是没有这个功能的，布局了之后，要重新运行才能看到效果，虽然可以热启动，但是还是没有这个预览来得直接。

还有一个非常牛逼的地方是，compose 的预览可以同时预览多个 composable 函数。

效果如下：

The screenshot shows the Android Studio code editor on the left and three preview cards on the right. The code editor contains Java code for Jetpack Compose previews:

```
76 // Preview section
77
78 @Preview( name: "Default colors")
79 @Composable
80 fun TutorialPreview() {
81     TutorialPreviewTemplate()
82 }
83
84
85 @Preview( name: "Dark colors")
86 @Composable
87 fun TutorialPreviewDark() {
88     TutorialPreviewTemplate(colors = darkThemeColors)
89 }
90
91 @Preview( name: "Font scaling 1.5", fontScale = 1.5f)
92 @Composable
93 fun TutorialPreviewFontscale() {
94     TutorialPreviewTemplate()
95 }
96
97 @Composable
98 fun TutorialPreviewTemplate(
99     colors: MaterialColors = lightThemeColors,
100    typography: MaterialTypography = themeTypography
101 ) {
102     val context = +ambient(ContextAmbient)
103     val previewPosts = getPostsWithImagesLoaded(posts.subList(1, 2), context)
104     val post = previewPosts[0]
105     MaterialTheme(colors = colors, typography = typography) {
106         Surface {
107             PostCardTop(post)
108         }
109     }
110 }
111
```

The three preview cards are:

- Default colors**: Shows a light-themed interface with a blue header and a white body.
- Dark colors**: Shows a dark-themed interface with a black header and a white body.
- Font scaling 1.5**: Shows a light-themed interface with a larger font size.

9.7 总结

Jetpack Compose 目前还是试验版，所以还存在很多问题，还不能现在将其用于商业项目中，但是这并不能妨碍我们学习和体验它，声明式 UI 框架近年来飞速发展，React 为声明式 UI 奠定了坚实基础并。Flutter 的发布将声明式 UI 的思想成功带到移动端开发领域，Apple 和 Google 分别先后发布了自己的声明式 UI 框架 SwiftUI 和 Jetpack Compose，以后，原生 UI 布局，声明式可能将会是主流。

10. Android Jetpack 架构组件--App Startup

10.1 解决的问题

一般需要初始化的 sdk 都会对外提供一个初始化方法供外界调用，如：

```
public class App extends Application {  
  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        Sdk1.init(this);  
    }  
}
```

对调用者很不友好。另一种做法是使用ContentProvider初始化，如下：

```
public class Sdk1InitializeProvider extends ContentProvider {  
    @Override  
    public boolean onCreate() {  
        Sdk1.init(getContext());  
        return true;  
    }  
    ...  
}
```

然后在AndroidManifest.xml文件中注册这个privoder，如下：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="cn.zhengzhengxiaogege.sdk1">  
    <application>  
        <provider  
            android:authorities="${applicationId}.init-provider"  
            android:name=".Sdk1InitializeProvider"  
            android:exported="false"/>  
    </application>  
</manifest>
```

这样初始化的逻辑就由Sdk开发者在内部完成了。

但是，如果一个app依赖了很多需要初始化的sdk，如果都放在一个ContentProvider中会导致此ContentProvider代码数量增加。而且每增加一个需要初始化的sdk都要对该ContentProvider文件做改动，不方便合作开发。而如果每个sdk都采用同样的方式将会带来性能问题。App Startup library可以有效解决这个问题。

10.2 使用App Startup

添加依赖

在App模块的build.gradle文件中添加依赖：

```
dependencies {  
    implementation "androidx.startup:startup-runtime:1.0.0-alpha01"  
}
```

实现Initializer接口

app通过Initializer接口接入App Startup，需要实现两个方法

```
public interface Initializer<T> {  
  
    @NonNull  
    T create(@NonNull Context context);  
  
    @NonNull  
    List<Class<? extends Initializer<?>>> dependencies();  
}
```

例如有一个Sdk1如下：

```
public class Sdk1 {  
  
    private static final String TAG = "Sdk1";  
  
    private static Context sApplicationContext;  
  
    private static volatile Sdk1 sInstance;  
  
    public static void init(Context applicationContext){  
        sApplicationContext = applicationContext;  
        Log.e(TAG, "Sdk1 is initialized");  
    }  
  
    public static Sdk1 getInstance(){  
        if (sInstance == null) {  
            synchronized (Sdk1.class){  
                if (sInstance == null) {  
                    sInstance = new Sdk1();  
                }  
            }  
        }  
        return sInstance;  
    }  
  
    private Sdk1(){  
    }  
  
    public void printApplicationName(){  
        Log.e(TAG, sApplicationContext.getPackageName());  
    }  
}
```

Sdk1对外提供Sdk1类，包含初始化方法init(Context)，实例获取方法getInstance()和对外的服务方法printApplicationName()。为了使用App Startup，需要提供一个初始化器如下：

```
public class Sdk1Initializer implements Initializer<Sdk1> {  
    @NonNull  
    @Override  
    public Sdk1 create(@NonNull Context context) {
```

```

        Sdk1.init(context);
        return Sdk1.getInstance();
    }

    @NonNull
    @Override
    public List<Class<? extends Initializer<?>>> dependencies() {
        return Collections.emptyList();
    }
}

```

泛型T为待初始化的Sdk对外提供的对象类型；create(Context)方法是该Sdk初始化逻辑写入的地方，其参数context为Application Context，同时需要返回一个Sdk对外提供的对象实例。dependencies()方法则需要返回一个列表，这个列表需要给出一个该Sdk依赖的其它Sdk的初始化器，也就是这个列表决定了哪些sdk会在这个sdk之前初始化，如果这个sdk是独立的没有依赖与其它的sdk，可以将该方法返回一个空列表(如Sdk1Initializer的实现)。但是如果这个sdk依赖于其它的sdk，必须在其它sdk初始化之后才能初始化，则需要在dependencies()方法中指明。例如现在有一个sdk2也需要初始化，且它必须在sdk1初始化之后才能初始化，那么sdk2的初始化器的实现如下：

```

public class Sdk2Initializer implements Initializer<Sdk2> {
    @NonNull
    @Override
    public Sdk2 create(@NonNull Context context) {
        Sdk2.init(context);
        return Sdk2.getInstance();
    }

    @NonNull
    @Override
    public List<Class<? extends Initializer<?>>> dependencies() {
        List<Class<? extends Initializer<?>>> dependencies = new ArrayList<>();
        dependencies.add(Sdk1Initializer.class);
        return dependencies;
    }
}

```

在dependencies()方法中指明了Sdk2的依赖项，因此App Startup会在初始化sdk2之前先初始化sdk1。

注册Provider和Initializer<?>

我们需要告诉App Startup我们实现了哪些Sdk初始化器(Sdk1Initializer、Sdk2Initializer)。同时App Startup并没有提供AndroidManifest.xml文件，因此App Startup用到的provider同样需要注册。在app的AndroidManifest.xml文件添加如下代码：

```
<provider
    android:authorities="${applicationId}.androidx-startup"
    android:name="androidx.startup.InitializationProvider"
    android:exported="false"
    tools:node="merge">
    <meta-data
        android:name="cn.zhengzhengxiaogege.appstartupstudy.Sdk1Initializer"
        android:value="@string/androidx_startup"/>
    <meta-data
        android:name="cn.zhengzhengxiaogege.appstartupstudy.Sdk2Initializer"
        android:value="@string/androidx_startup"/>
</provider>
```

通常每一个初始化器对应一个标签，但是如果有些初始化器已经被一个已经注册的初始化器依赖(比如Sdk1Initializer已经被Sdk2Initializer依赖)，那么

可以不用在AndroidManifest.xml文件中显式地指明，因为App Startup已经通过
注册的Sdk2Initializer找到它了。

这里的标签的value属性必须指定为字符串androidx_startup的值，

也就是("androidx.startup")，否则将不生效。

如果有一个sdk3内部通过App Startup帮助使用者处理了初始化，那么sdk3的AndroidManifest.xml文件中已经存在了InitializationProvider的provider标签，此时会与app模块中的冲突，因此在app模块的provider标签中指明
tools:node="merge"，通过AndroidManifest.xml文件的合并机制。

10.3 App Startup实现懒加载

为了减少app启动时间，对于一些非必须的初始化应该在app启动后、sdk使用前完成初始化，使用在provider中注册的方法不能达到这个目的。App StartUp提供了AppInitializer来解决这个问题。如下，在需要初始化的位置使用
AppInitializer:

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Log.d(TAG, "MainActivity Created");

        AppInitializer.getInstance(getApplicationContext())
            .initializeComponent(Sdk2Initializer.class);

        Sdk1.getInstance().printApplicationName();
    }
}
```

同时需要修改AndroidManifest.xml文件中的对应初始化器的，如下：

```

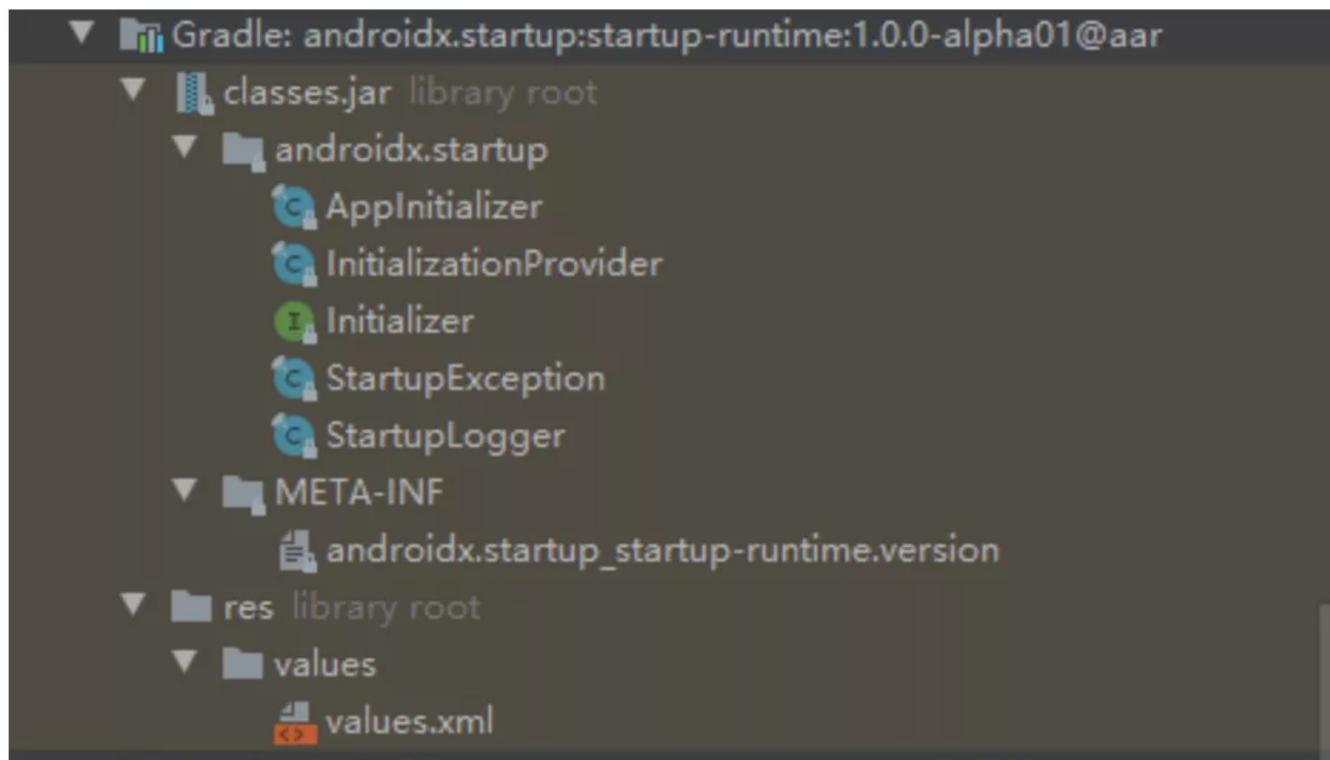
<provider
    android:authorities="${applicationId}.androidx-startup"
    android:name="androidx.startup.InitializationProvider"
    android:exported="false"
    tools:node="merge">
    <meta-data
        android:name="cn.zhengzhengxiaogege.appstartupstudy.Sdk2Initializer"
        android:value="@string/androidx_startup"
        tools:node="remove"/>
</provider>

```

通过tools:node="remove"来标记该初始化器。这样会在AndroidManifest.xml文件合并时将这个移除掉，否则该初始化器仍会在Application中被初始化并标记为已经初始化，后面的懒加载将不执行任何初始化操作，相当于使懒加载失效了。

10.4 剖析App StartUp

App StartUp的设计思路比较简单，就是将多个需要初始化的Sdk在一个provider中完成，从而减少多个provider带来的性能问题和繁杂的AndroidManifest.xml文件声明。目前App StartUp为1.0.0-alpha01版本，其代码结构非常简单。



只有五个类文件，除去StartupException和StartupLogger，其核心类只有三个。

Initializer.java的作用很简单，就是为lib的使用者提供了接入的方法，因此不再赘述。

InitializationProvider.java是App StartUp中使用的单一的provider，所有注册的初始化将在这个provider中完成。App StartUp只重写了onCreate()方法：

```

public final class InitializationProvider extends ContentProvider {
    @Override
    public boolean onCreate() {
        Context context = getContext();
        if (context != null) {
            AppInitializer.getInstance(context).discoverAndInitialize();
        } else {
            throw new StartupException("Context cannot be null");
        }
        return true;
    }
    ...
}

```

可以看到，在onCreate()方法中执行了扫描和初始化注册的组件。其实现方法在AppInitializer.java中。AppInitializer内部是一个单例实现，它的getInstance(Context)方法传入的是Application级别的Context，并将其传递给注册的各Initializer，首先看discoverAndInitialize()方法：

```

@SuppressWarnings("unchecked")
void discoverAndInitialize() {
    try {
        Trace.beginSection(SECTION_NAME);

        // 扫描并获取Manifest文件中的 `InitializationProvider` 这个组件中注册的<meta-data>
        // 信息
        ComponentName provider = new ComponentName(mContext.getPackageName(),
            InitializationProvider.class.getName());
        ProviderInfo providerInfo = mContext.getPackageManager()
            .getProviderInfo(provider, GET_META_DATA);
        Bundle metadata = providerInfo.metaData;

        // 然后遍历<meta-data>标签，获取到每个标签的 `Initializer` 并对其进行初始化
        String startup = mContext.getString(R.string.androidx_startup);
        if (metadata != null) {
            Set<Class<?>> initializing = new HashSet<>();
            Set<String> keys = metadata.keySet();
            for (String key : keys) {

                // 注意这里会用<meta-data>标签的value属性的值和`@string/androidx_startup`
                // 对比，只有是这个值的<meta-data>标签才会被初始化。
                String value = metadata.getString(key, null);
                if (startup.equals(value)) {
                    Class<?> clazz = Class.forName(key);
                    if (Initializer.class.isAssignableFrom(clazz)) {
                        Class<? extends Initializer<?>> component =
                            (Class<? extends Initializer<?>>) clazz;
                        if (StartupLogger.DEBUG) {
                            StartupLogger.i(String.format("Discovered %s", key));
                        }
                        doInitialize(component, initializing);
                    }
                }
            }
        }
    }
}

```

```

        }
    } catch (PackageManager.NameNotFoundException | ClassNotFoundException exception) {
        throw new StartupException(exception);
    } finally {
        Trace.endSection();
    }
}

```

discoverAndInitialize()方法首先扫描清单文件获取到需要初始化的初始化器Initializer，然后执行初始化操作，即调用doInitialize(Class<?>, Set<Class<?>>)方法，如下：

```

@NonNull
@SuppressWarnings({"unchecked", "TypeParameterUnusedInFormals"})
<T> T doInitialize(
    @NonNull Class<? extends Initializer<?>> component,
    @NonNull Set<Class<?>> initializing) {
    synchronized (sLock) {
        boolean isTracingEnabled = Trace.isEnabled();
        try {
            if (isTracingEnabled) {
                Trace.beginSection(component.getSimpleName());
            }

            // `initializing` 存储着正在初始化的初始化器。
            // 这个判断是要解决循环依赖的问题，比如 `Sdk1Initializer` 依赖了它本身，或者是
            // `Sdk1Initializer` 依赖了 `Sdk2Initializer`，同时 `Sdk2Initializer` 又
            // 依赖了 `Sdk1Initializer`，这是存在逻辑错误的，因此需要排除。
            if (initializing.contains(component)) {
                String message = String.format(
                    "Cannot initialize %. Cycle detected.", component.getName()
                );
                throw new IllegalStateException(message);
            }

            // `mInitialized` 是一个 `Map<Class<?>, Object>`，它缓存了已经执行过初始化的
            // `Initializer` 的 `class` 对象和初始化的结果，通过这种方法来避免重复初始化。
            Object result;
            if (!mInitialized.containsKey(component)) {

                // 这是这个初始化器还没被初始化的情况。
                initializing.add(component);
                try {

                    // 首先构造一个该初始化器的实例
                    Object instance = component.getDeclaredConstructor().newInstance();
                    Initializer<?> initializer = (Initializer<?>) instance;

                    // 读取它的依赖关系，如果有依赖的初始化器，要先对他们做初始化。
                    List<Class<? extends Initializer<?>>> dependencies =
                        initializer.dependencies();
                    if (!dependencies.isEmpty()) {
                        for (Class<? extends Initializer<?>> clazz : dependencies) {
                            if (!mInitialized.containsKey(clazz)) {

```

```

        doInitialize(clazz, initializing);
    }
}

if (StartupLogger.DEBUG) {
    StartupLogger.i(String.format("Initializing %s",
component.getName()));
}

// 调用初始化器的 `create(Context)` 方法，执行具体的初始化逻辑。
result = initializer.create(mContext);

if (StartupLogger.DEBUG) {
    StartupLogger.i(String.format("Initialized %s",
component.getName()));
}

// 最后把这个初始化器标为已初始化并缓存结果。
initializing.remove(component);
mInitialized.put(component, result);
} catch (Throwable throwable) {
    throw new StartupException(throwable);
}
} else {
    // 已经初始化过了，就直接从缓存中取走结果即可。
    result = mInitialized.get(component);
}
return (T) result;
} finally {
    Trace.endSection();
}
}
}
}

```

doInitialize(Class<?>, Set<Class<?>>)方法首先会实例化一个初始化器，然后通过dependencies()方法找到它依赖的初始化器做递归初始化，这个过程中如果遇到诸如依赖自身、循环依赖等逻辑错误问题将抛出异常。处理完依赖后调用它的create(Context)方法执行具体的初始化逻辑。最后初始化完成，将状态和结果缓存，防止多次初始化。

用来做懒加载的initializeComponent(Class<?>)的方法就比较简单了，它直接调用doInitialize(Class<?>, Set<Class<?>>)方法对指定的初始化器做初始化，如下：

```

@NonNull
@SuppressWarnings("unused")
public <T> T initializeComponent(@NonNull Class<? extends Initializer<T>> component) {
    return doInitialize(component, new HashSet<Class<?>>());
}

```

10.5 App Startup利弊

优点：

- 解决了多个sdk初始化导致Application文件和Mainfest文件需要频繁改动的问题，同时也减少了Application文件和Mainfest文件的代码量，更方便维护了
- 方便了sdk开发者在内部处理sdk的初始化问题，并且可以和调用者共享一个ContentProvider，减少性能损耗。
- 提供了所有sdk使用同一个ContentProvider做初始化的能力，并精简了sdk的使用流程。
- 符合面向对象中类的单一职责原则
- 有效解耦，方便协同开发

缺点：

- 会通过反射实例化Initializer<>的实现类，在低版本系统中会有一定的性能损耗。
- 必须给Initializer<>的实现类提供一个无参构造器，当然也不能算是缺点，如果缺少的话新版的android studio会通过lint检查给出提醒。

The screenshot shows a Java code editor with the following code:

```
import cn.zhengzhengxiaogege.sdk2.Sdk2;

public class Sdk2Initializer implements Initializer<Sdk2> {

    public Sdk2Initializer(int id) {
    }

    @NotNull
    @Override
    public Sdk2 create(@NotNull Context context) {
        Sdk2.init(context);
        return Sdk2.getInstance();
    }

    @NotNull
    @Override
    public List<Class<? extends Initializer<?>>> dependencies() {
        Sdk2Initializer > Sdk2Initializer()
    }
}
```

A red arrow points from the text "必须给Initializer<>的实现类提供一个无参构造器" to a warning dialog box. The dialog box contains the text "Missing Initializer no-arg constructor" and has three buttons: "Provide feedback on this warning Alt+Shift+Enter", "More actions... Alt+Enter", and a three-dot menu icon.

- 导致类文件增多，特别是有大量需要初始化的sdk存在时。
- 版本较低，还没有发行正式版。

11. Android Jetpack最新更新组件介绍

11.1 Hilt - Jetpack 推荐的依赖注入类库

译者注：前几天掘金有一篇介绍 Hilt 的文章 神一样的存在，Dagger Hilt !!，看评论区很多读者仍然把它当成 Dagger。其实官方也知道 Dagger2 难用，学习曲线陡峭，所以有了 Hilt，一个基于 Dagger2 的为 Android 准备的依赖注入类库。

Hilt 是一个帮助你简化 依赖注入 操作的 Android 类库，它让你可以专注于定义和注入的重要部分，而无需担心管理所有的 DI 设置。

基于 Dagger 之上 , Hilt 继承了它的编译期正确性 , 也提升了运行时性能和可扩展性。 Hilt 增加了对 Jetpack 类库和 Android Framework 类的集成。例如 , 要注入 `ViewModel` 的参数的话 , 你可以在 `ViewModel` 的构造函数上添加 `@ViewModelInject` 注解 , 并在 `Fragment` 上添加 `@AndroidEntryPoint` 注解。

在我们发布的博客 Dependency Injection on Android with Hilt 中 , 可以了解 Hilt 的更多信息。

11.2 Paging3 - 逐步加载和显示数据

Paging 是一个帮助你逐步分块加载和显示数据的类库。今天我们发布了 **Paging3** , 使用 Kotlin Coroutines 完全重写。这个版本添加了呼声很高的新特性 , 例如分隔符 , header , footer , 列表转换 , 用于重试和刷新的观察列表加载状态的 API 。

通过 Paging3 , 数据源可以继承 `PagingSource` , 并实现 `suspend load` 方法 , 在其中可以直接调用其他挂起函数。

关于 Paging3 的更新信息 , 请查看 文档 和 codelab 。

译者注 : 视频里有提到 , Paging3 是兼容 Paging2 的 , 大家可以放心升级 (我不负责。。。) 。

11.3 App Startup - 在应用启动时初始化组件

App Startup 类库提供了一种简单高效的方法在应用启动时初始化组件 , 而不是为每个需要初始化的组件定义单独的 ContentProvider 。 App Startup 允许你定义共享同一个 ContentProvider 的组件初始化器。这可以显著优化应用启动时间。

关于 AppStart 的更多信息 , 请查看 官方文档 。

11.4 Auto-fill IME

Android 11 引入了键盘相关的平台 API , 用于展示自动填充建议 , 例如密码管理。 Jetpack 的 AutoFill API 通过 `InlineSuggestionUi` 使得键盘和自动填充服务更简单的使用这一特性。 AutoFill 服务通过它可以提供可靠的建议 , 键盘通过它可以自定义建议的样式。

12:30



← Payment

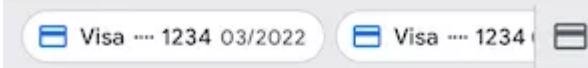
Name on card
Marian Ryan

Card number

MM/YY

Security code

Done


1 2 3 -
4 5 6 —
7 8 9
, 0 .
—— 

11.5 更简单的动画 — core-animation 和 SeekableAnimatedVectorDrawable

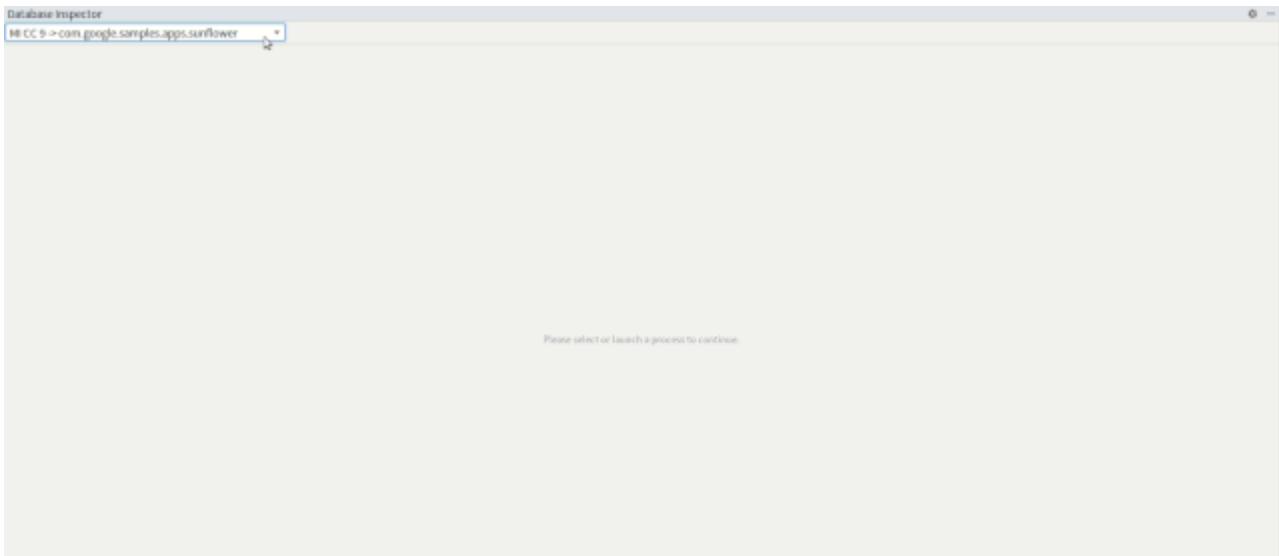
为了更简单的实现和测试动画，我们添加了两个新类库：`androidx.core:core-animation` 和 `androidx.core:core-animation-testing`。

我们还在 `androidx.vectordrawable` 库中引入了新的 API `SeekableAnimatedVectorDrawable`。

`core-animation` 移植了 `Animator` 自 Ice Cream Sandwich 依赖的所有特性，例如 暂停/恢复，拖动。

`SeekableAnimatedVectorDrawable` 是一个基于 `core-animation` 的全新的，可拖动的，
`AnimatedVectorDrawable(AVD)` 的替代方案。它和 AVD 使用同样的格式，并添加了 拖动，暂停，恢复的功能。

11.6 使用 Database Inspector 调试数据库



这块我就不翻译了，我两个月前的文章就介绍过了，可以查看一下 数据库还能这么看？

官方博客也有相关介绍：Database Inspector

11.7 WindowManager - 更好的设备支持

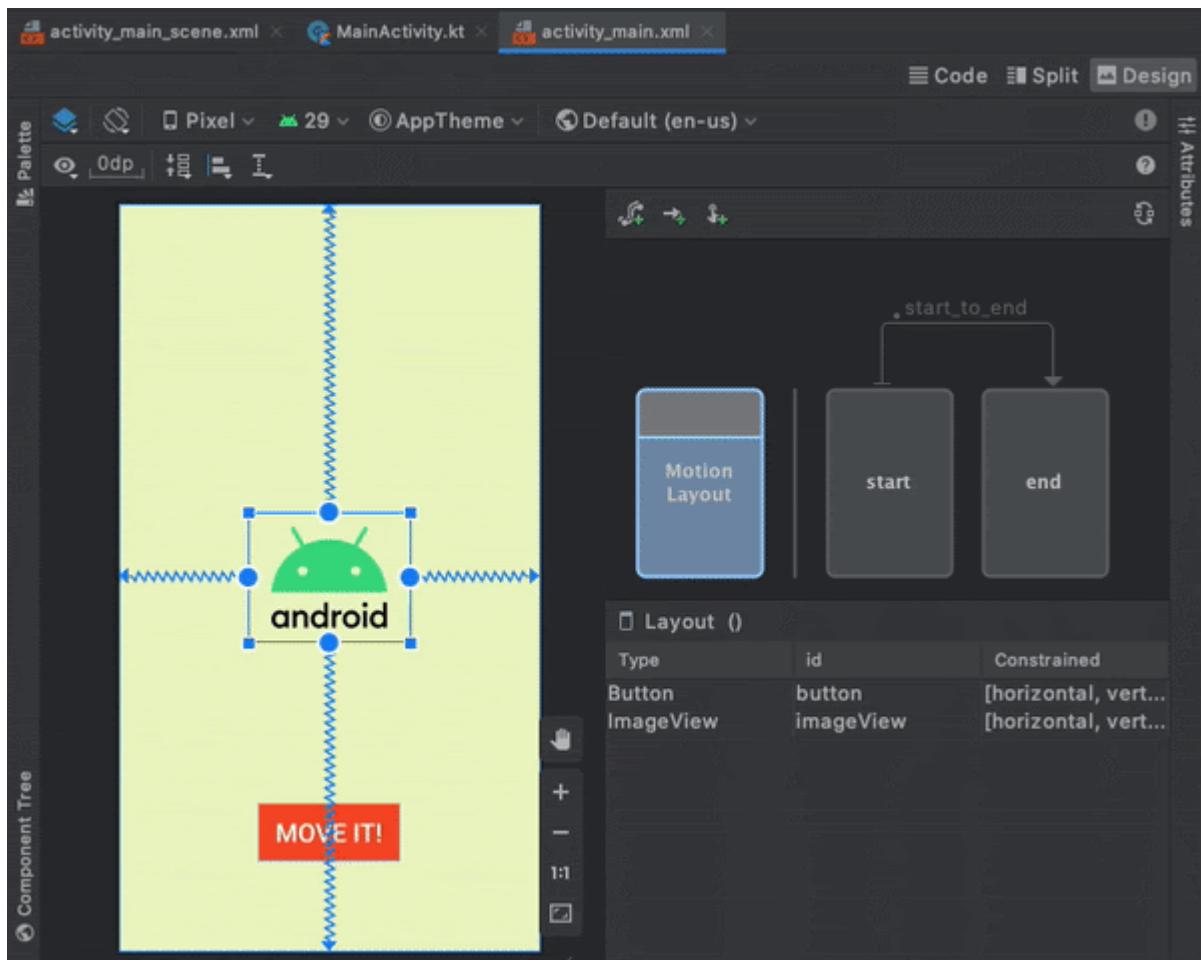
WindowManager 是 Jetpack 新增加的类库，旨在帮助开发者适配各种不同的设备，例如折叠屏。它为新旧平台版本的 WindowManager 特性提供了统一的 API 支持。

支持不同类型的可折叠设备的初始版本已经投入市场，所以开发者可以进行适配了。更多信息可以查看这篇博客 Support New Form Factors with the new Jetpack WindowManager Library，示例代码：<https://github.com/android/user-interface-samples/tree/master/WindowManager>

译者注：掘金上也有一篇相关文章，可以参考：Jetpack WindowManager，Android 折叠屏官方适配方案！

11.8 MotionLayout，构建流畅的交互式动画

MotionLayout 继承了 ConstraintLayout 的丰富特性，帮助 Android 开发者管理复杂的运动和窗口组件动画。通过 MotionLayout，你可以在 ConstraintSets 之间构造过渡动画，并且可以轻易的集成通用 View 的动画，像 RecyclerView 和 ViewPager。Android Studio 4.0 支持了 Motion Editor，用于创建和预览 MotionLayout 动画的图形工具。



12. Android Jetpack项目实战(从0搭建Jetpack版的WanAndroid客户端)

12.1 项目目的

在接触Android Jetpack组件时，就深深被其巧妙的设计和强大的功能所吸引，暗自告诉自己一定要学会这些组件，而网上并不能找到系统的学习资料，于是利用每天的时间访问Google Developer网站，把Jetpack的每个组件从使用到源码进行了系统的学习和总结。

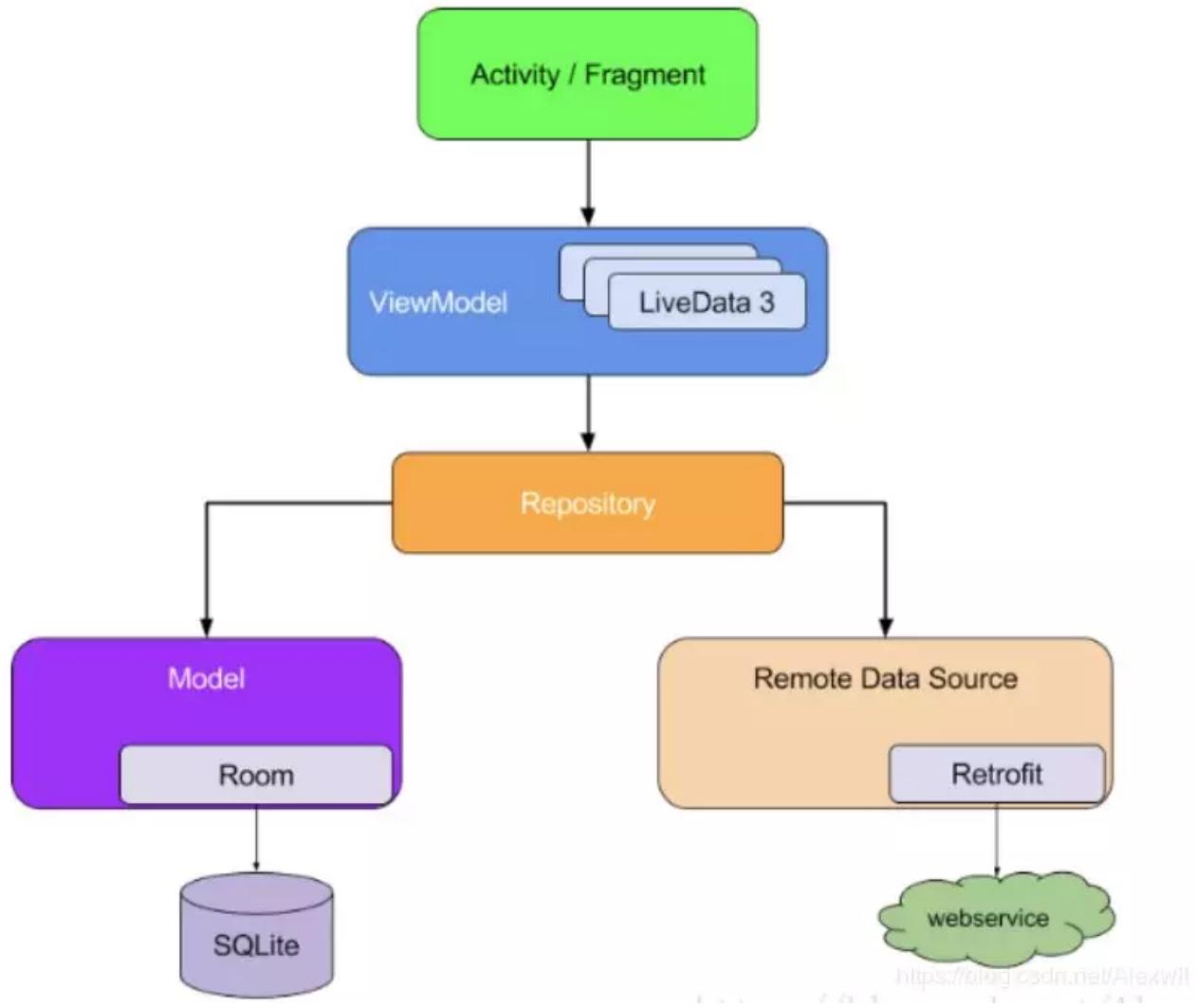
于是就有了[带你领略Android Jetpack组件的魅力](#)系列文章，希望在总结自己学习的同时，也能帮助需要这一些资料的同学，在写完这些文章后，想在项目中使用这些强大组件的想法就更加强烈了。

但又担心直接在公司项目中使用会又踩坑的危险，而且公司的项目又一时难以全部替换，好在WanAndroid提供了完整的应用接口，才有了这一个Jetpack版的WanAndroid客户端，项目功能比较简单，作为Jetpack组件的实战项目，旨在抛砖引玉和大家一起真正的使用Jetpack组件。

12.2 项目简介

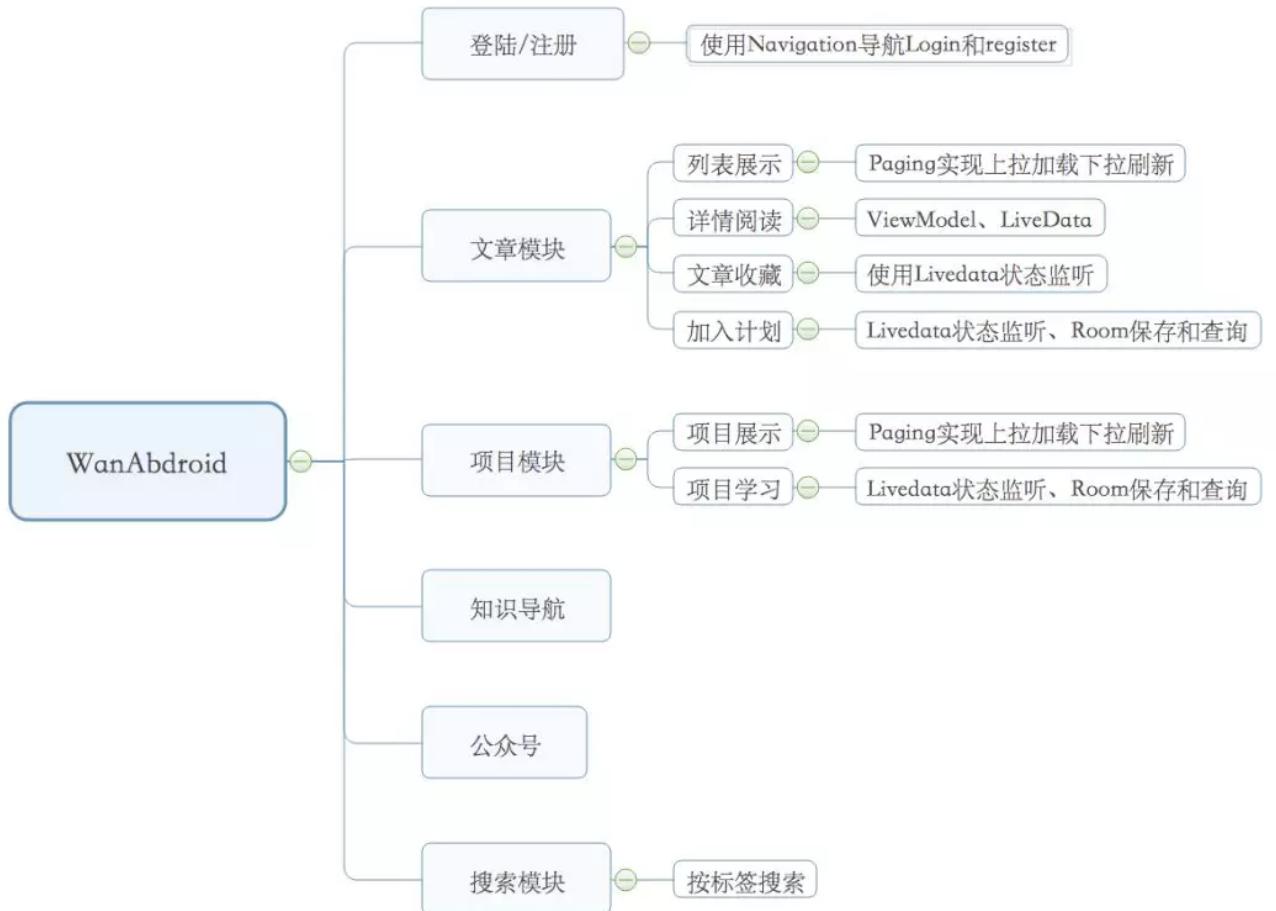
项目架构

既然本篇是对Android Jetpack组件的实战，那么就按照官方推荐的项目架构进行开发，架构内容见下图：



上面架构大家应是很熟悉的，基本原则和平时使用的MVC、MVP等一样，都是使界面、数据、和处理的逻辑进行解耦，打造稳定的、易测试、易扩展的项目架构，只是在这个过程中使用了全新的组件，如：ViewModel、LiveData等，使整个项目架构更加简单和灵活，关于使用的新组件不了解的可以点击文章开头的链接，学习相关组件的使用，本文默认读者已经了解组件的简单使用。

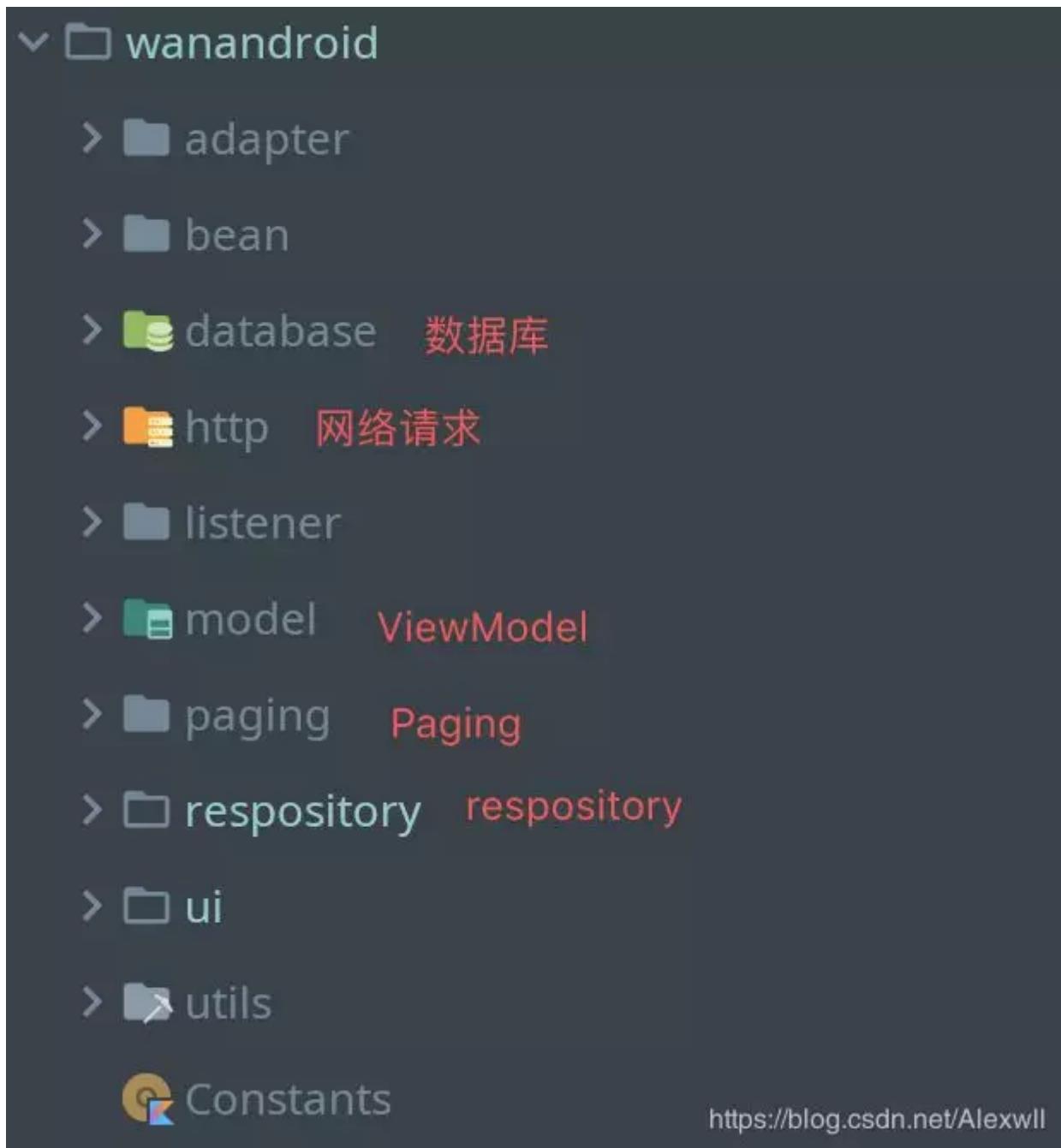
项目内容



<https://blog.csdn.net/Alenwill>

项目结构

本项目按照前面项目架构的知识，根据各个模块的功能进行分包管理，如下图：



12.3 项目实战

12.3.1 登陆模块

登陆模块遵循着一个Activity多Fragment的实现，提供注册（RegisterFragment）和登陆（LoginFragment）功能，相信这样的实现和写法对所有开发者来说都是So easy，甚至心里已将想好了如何像Activity添加Fragment，如何实现两个Fragment间的交互，我想说兄弟先停下脑子中的代码，来看看下面Loginactivity中的实现：

```
class LoginActivity : BaseCompatActivity() {

    override fun onErrorViewClick(v: View?) {}

    override fun initView(savedInstanceState: Bundle?) {}

    override fun getLayoutId() = R.layout.activity_login

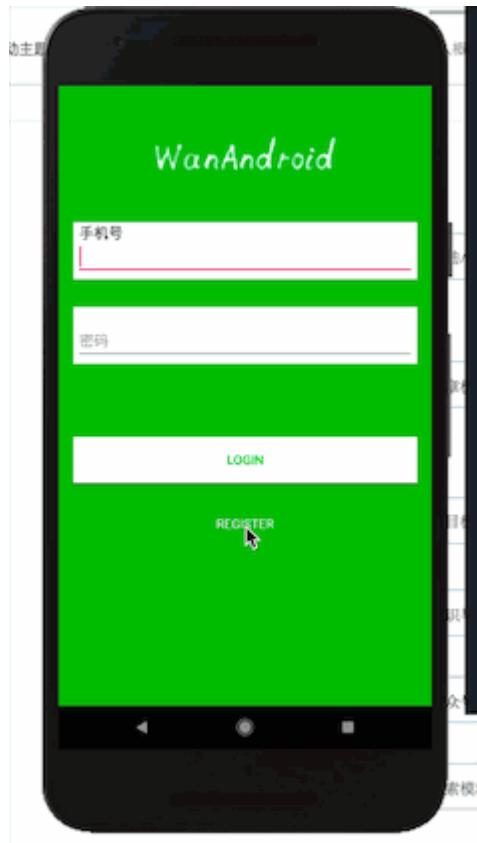
    override fun onSupportNavigateUp() = Navigation.findNavController(this,
R.id.fragment_navigation_login).navigateUp()}
```

onErrorViewClick()、initView()、getLayoutId()是在BaseCompatActivity中的抽象方法，用于加载布局和初始化控件，忽略这些方法后，真正实现像Activity中添加Fragment和Fragment的导航的代码就只有一行。

之所以这么简单完全得力于Navigation的使用，我们只需按规定的设置Navigation的xml文件，并将其加载到布局中，其他的操作都在Navigation中自动完成，下面看一下navigation.xml文件：

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto" android:id="@+id/login_navigation"
    app:startDestination="@+id/loginFragment">
    <fragment
        android:id="@+id/loginFragment"
        android:name="com.example.administrator.wanandroid.ui.fragment.LoginFragment"
        android:label="LoginFragment" >
        <action
            android:id="@+id/action_loginFragment_to_registerFragment"
            app:destination="@+id/registerFragment" />
    </fragment>
    <fragment
        android:id="@+id/registerFragment"
        android:name="com.example.administrator.wanandroid.ui.fragment.RegisterFragment"
        android:label="RegisterFragment" />
</navigation>
```

效果展示



12.3.2 文章模块

3.2.1、文章列表展示

对于常规的内容展示，使用RecyclerView并实现上拉加载和下拉刷新即可，此处使用Paging组件实现这些功能，对于Paging的下拉加载之前文章已经介绍了，通过自定义DataSource控制数据的加载和分页，本文不再进行介绍，这里只介绍对Paging组件进行了简单的封装，代码结构如下：



除了DataBase、Factory和Adaoter之外，上述封装中主要的类是三个类：

- Listing：用于封装需要监听的对象和执行的操作，用于系统交互
- BaseRepository：配置并实例化LivePagedListBuilder（）对象，根据设定的监听状态和数据，封装List对象
- BasePagingViewModel：保存所有的可观察的数据和所有的操作方法

Listing代码如下，属性和作用见代码注释：

```
/*
 * 用于封装需要监听的对象和执行的操作，用于系统交互
 * pagedList：观察获取数据列表
 * networkStatus：观察网络状态
 * refreshState：观察刷新状态
 * refresh：执行刷新操作
 * retry：重试操作
 * @author：Alex
 * @date：2018/08/21
 * @version：v 2.0.0
 */
data class Listing<T>(
    val pagedList: LiveData<PagedList<T>>,
    val networkStatus: LiveData<Resource<String>>,
    val refreshState: LiveData<Resource<String>>,
```

```
    val refresh: () -> Unit,
    val retry: () -> Unit)
```

BaseRepositroy

```
abstract class BaseRepository<T, M> : Repository<M> {

    /**
     * 配置PagedList.Config实例化List<M>对象，初始化加载的数量默认为{@link #pagesize} 的两倍
     * @param pageSize : 每次加载的数量
     */
    override fun getDataList(pageSize: Int): Listing<M> {

        val pageConfig = PagedList.Config.Builder()
            .setPageSize(pageSize)
            .setPrefetchDistance(pagesize)
            .setInitialLoadSizeHint(pagesize * 2)
            .setEnablePlaceholders(false)
            .build()

        val stuDataSourceFactory = createDataBaseFactory()
        val pagedList = LivePagedListBuilder(stuDataSourceFactory, pageConfig)
        val refreshState = Transformations.switchMap(stuDataSourceFactory.sourceLivaData) {
            it.refreshStatus }
        val networkStatus = Transformations.switchMap(stuDataSourceFactory.sourceLivaData) {
            it.networkStatus }

        return Listing<M>(
            pagedList.build(),
            networkStatus,
            refreshState,
            refresh = {
                stuDataSourceFactory.sourceLivaData.value?.invalidate()
            },
            retry = {
                stuDataSourceFactory.sourceLivaData.value?.retryFailed()
            }
        )
    }

    /**
     * 创建DataSourceFactory
     */
    abstract fun createDataBaseFactory(): BaseDataSourceFactory<T, M>
}
```

上述代码中做了以下事情：

1. 创建BaseDataSourceFactory实例
2. 初始化并配置Paging组件
3. 转换并监听BaseDataSourceFactory中保存的可观察的DataSource状态的变化
4. 将所有的监听状态封装到Listing的实例中

对于上拉加载之前的文章有介绍，可对于下拉刷新的实现并没有直接介绍，不过从上面的代码可以看出，此处的refresh（）调用DataSource的invalidate（）方法，通知数据失效，此时数据会从新加载。

BasePagingViewModel

BasePagingViewModel的作用就是ViewModel的基本作用，不过这里进行了相关状态的转换和监听，没错就是前面生成和封装的Listing实例中的操作，

```
open class BasePagingViewModel<T>(repository: Repository<T>) : ViewModel() {
    //开始时建立DataSource和LiveData<List<StudentBean>>的连接
    val data = MutableLiveData<Int>()
    // map的数据修改时，会执行studentRespository 重新创建 LiveData<List<StudentBean>>
    private val repoResult = Transformations.map(data) {
        repository.getDataList(it)
    }
    // 从List对象中获取要观察的数据，调用switchMap当repoResult 修改时会自动更新 生成的LiveData
    // 监听加载的数据
    val pagedList = Transformations.switchMap(repoResult) {
        it.pagedList
    }!!
    // 网络状况
    val networkStatus = Transformations.switchMap(repoResult) { it.networkStatus }!!
    // 刷新和加载更多的状态
    val refreshState = Transformations.switchMap(repoResult) { it.refreshState }!!

    /**
     * 执行刷新操作
     */
    fun refresh() {
        repoResult.value?.refresh?.invoke()
    }

    /**
     * 设置每次加载次数，初始化 data 和 repoResult
     * @param int 加载个数
     */
    fun setPageSize(int: Int = 10): Boolean {
        if (data.value == int)
            return false
        data.value = int
        return true
    }

    /**
     * 执行点击重试操作
     */
    fun retry() {
        repoResult.value?.retry?.invoke()
    }
}
```

ViewModel中储存和执行的方法见上面的注释，所有的监听状态都是转换Listing实例，而Listing实例的创建有时装欢DataSource，所以用户执行的操作和DataSource就联系起来了，当你使用了Paging组件的时候，你真的会有牵一发动全身的感觉，简单来说只要DataSource的数据、请求状态、请求结果任意一个发生改变，相应的ViewModel中的数据就会改变，那在Fragment中监听的Observer就会执行相应的方法，响应用户的操作。

使用效果



3.3.2 文章阅读

这个部分的实现比较简单，也是组件的经典结构，详情页主要是根据文章的Url和Title决定，换句话说只要Url和Title改变文章的内容就会改变，所以只要在ViewModel中保存Title和Url的可观察类，在Activity中监听二者并在其改变时执行相应的操作。

```
val contentTitle = MutableLiveData<String>()
val contentUrl = MutableLiveData<String>()
```

Activity中观察数据：

```
model.contentTitle.observe(this, Observer {
    supportActionBar?.title = it
})
model.contentTitle.value = mTitle

private fun initwebView() {
    model.contenturl.observe(this, observer {
        createwebview(it)
    })
    model.contenturl.value = mUrl
}
```

效果展示



3.3.3、文章收藏和加入阅读计划

这部分和上面文章展示大致相似，只不过比它多了初始化收藏状态、收藏后上传服务器和保存数据库的操作，也就是多了ArticleDetailRespository中的调度操作，执行逻辑大致如下：

1. 在显示详情时，初始化本篇文章的收藏状态和加入计划状态
2. 点击收藏或计划后响应操作
3. 执行逻辑后响应界面修改

实现过程如下：

在ArticleDetailRepository中创建LiveData标记收藏和阅读的状态

```
class ArticleDetailRepository(val api: Api, val context: Context) {  
    val articleIsCollected = MutableLiveData<Boolean>()  
    val articleIsReadLater = MutableLiveData<Boolean>()  
}
```

在ViewModel中转换ArticleDetailRepository中的LiveData

```
/**  
 * 是否收藏  
 */  
val collected = Transformations.map(articleDetailRespository.articleIsCollected) { it }!!  
  
/**  
 * 是否加入阅读计划  
 */  
val readPlan = Transformations.map(articleDetailRespository.articleIsReadLater) { it }!!
```

在UI界面中观察数据

```
//如果文章已收藏则显示“取消收藏”，否则显示“文章收藏”  
model.collected.observe(this, Observer {  
    if (it!!) collectButton.setText(R.string.cancel_collect_article) else  
    collectButton.setText(R.string.collect_article)  
})  
//如果文章已加入计划则显示“取消阅读计划”，否则显示“加入阅读计划”  
model.readPlan.observe(this, Observer {  
    if (it!!) readPlanButton.setText(R.string.delete_read_plan) else  
    readPlanButton.setText(R.string.add_read_plan)  
})
```

到这里实现了监听文章的操作状态，根据文章收藏和计入计划的状态，改变相应的UI控件，那么剩下的是执行相应的操作，然后去改变ArticleDetailRepository中可观察数据的状态。

此处文章的收藏和阅读计划相同，都是根据本地数据的存储进行或服务端数据初始化，操作成功后再修改数据库的操作，关于网络的请求本文不做介绍了，只是在请求收藏链接成功后修改ArticleDetailRepository中状态即可，本文主要介绍“加入”和“取消”阅读计划，此部分是保留在本地的数据库中，所以接下来就看看阅读计划的数据库创建。

DataBase：本项目后面的几个关于数据库的操作，如：项目学习等，不一一介绍都以此阅读计划为例

```
@Database(entities =  
[CollectArticle::class, ReadPlanArticle::class, StudyProject::class, RecentSearch::class], versi  
on = 1, exportSchema = false)  
abstract class AndroidDataBase : RoomDatabase() {  
  
    abstract fun getCollectDao() : CollectedDao // 用于收藏文章操作  
    abstract fun getReadPlanDao() : ReadPlanDao // 用于阅读计划操作  
    abstract fun getStudyProjectDao() : StudyProjectDao // 用于项目学习操作  
    abstract fun getRecentSearchDao() : RecentSearchDao // 用于最近搜索操作
```

```

companion object {
    @Volatile
    private var instance : AndroidDataBase? = null
    fun getInstence(context: Context) : AndroidDataBase{
        if (instance == null){
            synchronized(AndroidDataBase::class){
                if (instance == null){
                    instance =
                    Room.databaseBuilder(context.applicationContext,AndroidDataBase::class.java,"wanAndroid")
                        .build()
                }
            }
        }
        return instance!!
    }
}

```

Entity

```

@Entity(tableName = "read_plan")
data class ReadPlanArticle(var author: String? = null,
                           var chapterName: String? = null,
                           var link: String? = null,
                           var articleId: Int = 0,
                           var title: String? = null
){}
@PrimaryKey(autoGenerate = true)
var id: Int = 0
}

```

Dao

```

@Dao
interface ReadPlanDao {
    @Insert
    fun insert(readPlanArticle: ReadPlanArticle)
    @Delete
    fun remove(readPlanArticle: ReadPlanArticle)
    @Query("SELECT * from read_plan")
    fun getArticleList():DataSource.Factory<Int,ReadPlanArticle>
    @Query("SELECT * from read_plan WHERE articleId = :id")
    fun getArticle(id :Int):ReadPlanArticle
}

```

数据库的创建和要执行的操作已在上述配置完成，关于Room的使用这里不再介绍，结下来看看 ArticleDetailRepository中是如何使用数据库，响应和修改LiveData的数据，我们依次看看初始化、加入计划和取消计划的操作

初始化：主要查询数据库中是否保存此文章，并更新界面UI

```
fun isRaedPlan(context: Context, id: Int) {
    runOnIoThread {
        val liva = AndroidDataBase.getInstence(context).getReadPlanDao().getArticle(id)
        if (liva != null) {
            articleIsReadLater.postValue(true)
        } else {
            articleIsReadLater.postValue(false)
        }
    }
}
```

上述代码执行操作：根据文章Id从数据库查询此文章，如果存在将articleIsReadLater设置为true，否则设置为false，那么ViewModel和Activity中的观察者都会执行响应改变。

注意：数据库的所有操作都不嫩放在主线程中

加入阅读计划：向数据库添加一条记录，并在添加成功后修改articleIsReadLater值

```
fun addStudyProject(readPlanArticle: StudyProject) {
    runOnIoThread {

        AndroidDataBase.getInstence(context).getStudyProjectDao().insert(readPlanArticle)
        articleIsReadLater.postValue(true)
    }
}
```

取消阅读计划：删除数据库记录，并修改articleIsReadLater值

```
fun removeReadLater(id: Int) {
    runOnIoThread {
        val readPlanArticle =
        AndroidDataBase.getInstence(context).getReadPlanDao().getArticle(id)
        AndroidDataBase.getInstence(context).getReadPlanDao().remove(readPlanArticle)
        articleIsReadLater.postValue(false)
    }
}
```

效果展示



3.3.4、阅读计划的展示

阅读计划的内容是储存在本地数据库中，所以对文章的展示自然是Room的数据库的查询，而查询后数据的展示又是RecyclerView的使用，提到RecyclerView就会想到Paging组件，没错我们想到的Google已经想到了，他们对Room和Paging进行了额外的支持，即可以实现对数据库的监听，当数据库改变时直接显示在RecyclerView中，首先在Room中设置数据库和查询数据库，此步骤前面已经完成，看一下这个方法：

```
@Query("SELECT * from read_plan")
fun getArticleList(): DataSource.Factory<Int, ReadPlanArticle>
```

这里Room查询直接返回了DataSource.Factory的实例，也就是说Room已经在查询的时候就直接初始化了DataSource，简化了我们的操作，接下来看看ViewModel中如何处理数据：

```
class PlanArticleModel(application: Application) : AndroidViewModel(application) {

    val dao = AndroidDataBase.getInstance(application).getReadPlanDao()

    val livePagingList : LiveData<PagedList<ReadPlanArticle>> =
        LivePagedListBuilder(dao.getArticleList(), PagedList.Config.Builder()
            .setPageSize(5)
            .build()).build()
}
```

上面代码执行如下操作：

1. 继承AndroidViewModel

2. 初始化数据库查询的ReadPlanDao实例
3. 初始化并配置LivePagingList

在Ui中监听ViewModel中的LiveData：

```
model.livePagingList.observe(this, observer {  
    adapter.submitList(it)  
})
```

此时你在添加和移除数据库操作时，Room返回的DataSource中的数据会发生改变，进而RecyclerView自动实现数据刷新，效果如下：



12.3.3 其余模块

1. **项目模块**：实现代码和文章模块相似，Paging展示项目列表，Room保存数据，只是所有的操作都针对于玩安卓中的学习项目；
2. **导航模块**：根据Tag导航响应的文章
3. **公众号**：在Fragment中使用Paging展示各个公众号中的文章
4. **搜索模块**：SearchView搜索文章，Room保存最近搜索

12.4 总结

以上是本项目各个模块的实现和分析，实现的项目比较简单，主要是展示以下组件的使用以及组件间的配合使用，本计划加入WorkManger做定时提醒的功能，并加上一些完整的功能，但由于各种原因（具体大家都懂。。。），后面有机会会继续完善，那本文到此结束，希望对大家学习和了解Jetpack组件，以及灵活应用组件有所帮助，让大家一起更好的学安卓、玩安卓！

点击查看源码

<https://github.com/AlexTiti/WanAndroid>