deepwaterooo deepwaterooMe – I am the same GitHub account person

deepwaterooo

September 13, 2022

Contents

1 要求 **1**

2 主要思路 **2**

3 OkHttp 网络请求的缓存 **4**

4 封装：OkHttp + EventBus **6**
4.1 BaseEvent.java . . . 7
4.2 HttpEvent . . . 7
4.3 HttpSuccessEvent . . . 7
4.4 HttpErrorEvent . . . 7
4.5 AppEvent . . . 8
4.6 RequestTag . . . 8
4.7 MainRequest . . . 8
4.8 UserRequest . . . 10
4.9 BaseActivity . . . 10
4.10 使用封装 . . . 11
4.11 怎么设置网络请求的缓存? . . . 11

5 Retrofit **12**

6 room 数据库相关的部分: 几个相关可以用作参考的例子 **12**
6.1 另一个更好的参考例子：dagger + RecyclerView 和相应的 Adapters + BufferKnife View auto-injections . . . 12
6.2 ApiModule.java . . . 12
6.3 ApiComponent.java . . . 13
6.4 NocacheActivity extends Activity . . . 13
6.5 这里有个小例子： . . . 13

7 Glide 的缓存分为两种，Resource 缓存、Bitmap 缓存。 **14**
7.1 一、Resource 缓存： . . . 14
7.2 二、Bitmap 缓存：Bitmap 所占的内存大小由其三部分组成：图片宽，高和 Bitmap 质量参数。 . . . 14

8 下载图片并保存到本地：rxjava 2.x+retrofit 通过动态 url 保存网络图片到本地 **14**

9 关于图片的处理：不仅要下载，下载后还需要自动保存到数据库 **16**

10 用 Retrofit+Rxjava 上传图片支持多张图片的上传 **17**

11 Android onSaveInstanceState()、onRestoreInstanceState() 保存和恢复被系统销毁的数据 **19**
11.1 一、onSaveInstanceState(Bundle outState) 在什么时机会被调用呢? 20
11.2 二、onRestoreInstanceState 什么时机被调用? 21
11.3 三、onCreate() 里也有 Bundle 参数,可以用来恢复数据,它和 onRestoreInstanceState 有什么区别? 21

12 Library Hours **22**

13 OkHttp Call 实现的简单案例: 最简单的小例子 **22**
13.1 简单的异步 Get 请求 22
13.2 简单的异步 Post 请求 23
13.3 OkHttp 的封装 23
13.3.1 CallBack 的创建 23
13.3.2 对 Get、Post 方法的简单封装 24

14 转屏等几种情况下 activity 的销毁与否，销毁与重建，数据保存 **27**
14.1 activity 被系统超脱用户期望被销毁时数据的保存与恢复 27
14.2 对图片的保存 27

15 Retrofit + RxJava + Json 数据解析 **29**

16 RxJava 线程调度 **29**
16.1 RxJava 内置的 5 个 Scheduler 29
16.2 线程的控制 30

17 其它相对比较有参考价值的链接 **30**

18 Retrofit + RxJava 的进一步深入理解：一个详尽的封装讲解与归纳 **30**
18.1 BaseResponse 就是对基础数据进行封装处理。 30
18.1.11、根据基础数据定义 BaseResponse 30
18.1.22、修改 API 接口返回数据类型 31
18.1.33、对基础数据统一处理 31
18.1.4 请求网络数据 32
18.2 RxHelper 调度类封装 32
18.3 Retrofit 初始化封装 33
18.4 细节完善 33
18.4.11、服务错误信息处理 33
18.4.22、添加“正在加载”弹窗 34
18.4.33、Retorfit 请求方法汇总 35
18.4.44、提交参数方式汇总（可忽略） 36
18.5 总结 37
18.6 RxJava 观察者模式，与公用接口回调类的设计实现 38

19 RxJava: 防止 Rxjava 引发的内存泄漏: 需要把这部分理解清楚并整合到自己的项目中去 **38**
19.1 哪些场景会出现 38
19.2 解决办法 39
19.3 （1）添加依赖 39
19.4 （2）基本使用 39

20 RxLifecycle 第三方库：解决内存泄露，原理详解 **40**
20.1 1.bindToLifecycle 40
20.2 ActivityEvent FragmentEvent 40
20.3 2.bindUntilEvent 41
20.4 3.LifecycleProvider 41
20.5 4. 总结 42

1 要求

- Build an **employee directory app** that shows a list of employees from the provided endpoint.
- The app should display a list (or any kind of **collection view**!) which shows all the employees returned from the JSON endpoint described below.
- Each item in the view should contain a **summary of the employee**, including their **photo, name, and team** at minimum. You may add more information to the summary if you want, or **sort employees in any fashion** you'd like –sort and group by name, team, etc.
  - 这个很好实现：加一个 ListView 的下拉列表框以提供可选择的排序标准；因为是 MVVM 设计，由数据来驱动 UI, 用户选择排序后，只需要操作 viewModel 里的链表按标准排序，并刷新 UI 就可以了，因为简单，不作这个项目的重点。
- There should be some UX to reload the employee list from within the app at any time. The UX can be done in any way you want: **a button, pull-to-refresh**, etc.
- If there is any additional UI/UX you would like to add, feel free to do so! We only ask that you please **do not build any more screens** than this list. Do not worry about building custom controls or UI elements –using **system-provided, standard elements** is totally fine.
- Be sure to **appropriately handle the normal variety of errors when querying an endpoint**. The app should **display useful loading, empty, and error states** where appropriate. **If images fail to load, displaying a placeholder** is fine.
  - **网络请求的异常处理**：这个我想要把它作为项目的重中之重来处理
- One extra thing we ask is that you please ensure you **do not use more network bandwidth than necessary –load expensive resources such as photos on-demand only**.
  - **网络请求处理**：在且仅在必要的时候网络请求
- The **employee list should not be persisted to disk**. You can reload it from the network **on each app launch and when refresh is requested** —but no more often than that unintentionally.
- Android developers in particular should take care **not to make redundant network calls** when the **phone is rotated, or when memory is low**.
  - 旋转手机屏的时候，或手机内存低的时候，不能重新申请网络请求。就是 **转手机屏的时候，activity 不能重建**，只更改视图就可以了；(这个很容易就可以实现)
  - 当系统内存少，需要销毁 acitivty 并在必要时重建时 (activity 销毁前的数据保存)：之前没有搞懂数据库是起什么作用的，我把它删除了；但是现在看来，我还是需要使用数据库来在必要的时候保存链表数据的 (另对图片的保存处理是否达到标准要求，还要再证实一下)

- **Images**, however, should **be cached on disk** so as to not waste device bandwidth. You may use an **open source image caching solution**, or write your own caching. Do not rely upon HTTP caching for image caching.
  - 这里 **需要测试一下，再想想要不要保存**
  - 我不担心这个，觉得 Glide 已经最大限度地优化了这个整个过程。只需要选对这个库就可以了
- Note that photos at a given URL will never change. Once one is loaded, you do not need to reload the photo. If an employee's photo changes, they will be given a new photo URL.
  - **不懂这句话说的是什么意思，有什么影响？**
- **Tests should be provided for the app**. We do not expect 100% code coverage, so please use your best judgment for what should be tested. We're also interested only in **unit tests**. Feel free to skip snapshot or app tests.
  - 不喜欢测试，但是这里可能 **需要加一两个测试用例**
- MVVM: 需要数据驱动，viewModel 里定义一个状态变量，来标记当前的活动状态
- If any employee is malformed, it is fine to invalidate the entire list of employees in the response - there is no need to exclude only malformed employees.
- If there are no employees to show, the app should present an **empty state** view instead of an empty list.

2 主要思路

- **对照它提示过/读得懂的要求，把自己可能会 miss 掉的小细节全捡回来**
- 现在重新再次整合 Room 数据库，把手机内存产资源不足时销毁 activity 保存数据的部分补充完整；
- 现在终于改掉了自己网址写错的小 bug 了，可以再往前迈进一步，深入深解拦截器与 OkHttp 底层的原理了。。。
- 顺着这个例子https://github.com/xitu/gold-miner/blob/master/TODO/getting-started-wi
  md 把进阶的部分看完，弄懂
- 终于找到了先前 **今年二三月份参考过的一个很好的案例日志序列**：MVVM https://blog.
  csdn.net/qq_38436214/category_11482619.html?spm=1001.2014.3001.5482
  - 下午忘记带一根测试用的线了，就暂时理一下思路，晚上回家后再在大电脑上测试
- 这是一个 **看似要求极其简单，实则考验的知识点和深度有着相当的跨度的小项目**。
- 它们一定挑都要挑我出差到 WSU 的一个星期里来考验我，因为他们就是想要去打败一个人。呵呵，真正想要打败一个人，谈何容易，就凭这???
- **Retrofit + RxJava**: 好像是更合适的，可以用注解，并且用得更为广泛
  - 搜索关键字：Retrofit + OkHttp +RxJava 网络库构建
  - **OkHttp**: 网络请求处理，主要是在应用启动的时候，什么时机开始发布和调用网络请求。所以这个可以不用了，大家都喜欢新的更好用的库
- **网络数据解析**：我这里得到了网络数据，可是好像我并没有解析数据出来，这整个过程我可能还少了这比较关键的一个步骤

  - 当对这类框架和 OOP 设计有了更好的了解，就可以自定义解析类来在获得数据前自动解析为自己想要的类型
    * 注意这里提供的 API 其实并没有 code msg 之类的信息，而是直接的结果；再想一下：为什么 OkHttp 的拦截器能够得到 code 200 呢？这里 Response 返回基本信息还有点儿糊涂
    * 可以参考这个例子：https://developer.aliyun.com/article/609862

- **RxJava 基本原理**：RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.(一个通过使用可观察序列来组成异步的、基于事件的程序的库。)
  - 从介绍中我们可以提取出一个关键词：异步，但安卓中已经有很多解决异步操作的方法了，比如 Handler 和 AsyncTask 等，**为什么还选择 RxJava 呢，其实目的就是为了让代码更简洁，而且它的简洁是与众不同的，因为 RxJava 的使用方式是基于事件流的链式调用，这就保证了随着程序复杂性的提高，RxJava 依然能保持代码的简洁和优雅**。

- **图片本地缓存**: 第三方库找一个，还是用 AndroidX 的 Room
  - 上面可能想错了，就是使用的第三方图片库 Glide 本身已经具备了缓存图片到本地的功能，所有我大可不必再多此一举，再来一个 AndroidX Jetpack 的 Room 数据库
  - 现将所有的 room 数据库相送的源码全部删除，**现只负责将 RecyclerView 的所有相送逻辑连通就可以了**

- 小问题：根据返回来数据的 url 链接来加载员工头像图片，可能并不需要我来实现什么网络请求的串接执行，更多应该是 **Recyclerview 的双向数据绑定就可以了**

- 头像图片加载：现有两套思路，一套 Mitch 的 MVVM 但是非数据绑定的版本，一套双向数据绑定的自动化绑定的版本。
  - 按照项目的要求与缓存机制的要求，我觉得用 Mitch 的版本更为简洁，主要由第三方库 Glide 帮助缓存处理。自己需要必理的逻辑比较少，更简洁方便好用。
  - 那么下面的这些关于缓存的问题都可以暂时不思考了，先运行起一个可以执行运行不出错的应用再说再优化。
  - 我 **现在数据库的问题**是：我 **缓存保存了员工数据进数据库**，但是这里说得很清楚了，**不用保存员工数据，只保存每个员工 id 所对应的图片就可以了**
  - 说到网络缓存，肯定都不陌生，多多少少使用过不同的缓存方案。使用网络缓存有什么作用:
    * 减少服务器请求次数
    * 减少用户等待时间
    * 增加应用流畅度
    * 节省用户流量（虽然现在流量也不怎么值钱了）

- 应用的 **启动优化**：重中之重，需要借助这个小应用弄懂弄清楚，**不知道如何拆解网络请求的步骤，什么时候加载，初始化之类的？**以达到较好的启动优化

- **MVVM 设计**：只有一个页面，相对就简单方便多了。工作中的案例是使用 MVVM 但自己编辑逻辑处理信号下发，与数据驱动的 UI 更新，没有实现双向数据绑定的；可是这里感觉 **双向数据绑定**更简单，会有哪些可能的问题呢？这里基本可以当作不需要双向，因为一个 UI 按钮要求刷新是唯一的 UI 需求；更多的只是需要时候的数据往 UI 加载更新；所以 **可以简单使用观察者模式，UI 观察数据的变化**就可以了

- **图片的加载与处理**：用样可以使用么第三方库 **glide**

- **图片的加载与处理**：用样可以使用么第三方库 **CircularImageView**

- **AndroidX RecyclerView** 的使用：选择相对更为高效和方便管理的库和数据结构来使用
- **Constraint Layout vs Coordinate Layout**: 暂时先用任何简单的 layout 先能运行起一个大致的框架来，再进一步优化
- 我丢掉了的文件呀，我写过的项目呀，不是在进 Lucid 之前写得好好的一个项目，现在源码全丢了。。。。。该死的 GitHub. . . ..

3 OkHttp 网络请求的缓存

- **OkHttpClient/Retrofit 里在网络请求的时候 (根据不同的 url 链接，或是不是请求接口？基于拦截器来做缓存) 来动态使用不同的缓存策略 (适用于自己只缓存图片，而不缓存员工链表)**，这个思路应该用在这个项目的设计与实现里。原理参考这个思路：
  - https://blog.csdn.net/c10WTiybQ1Ye3/article/details/125687902?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-125687902-blog-51550400.pc_relevant_multi_platform_whitelidepth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-125687902-blog-51550400.pc_relevant_multi_platform_whitelutm_relevant_index=1
- 笔记本电脑上的 kotlin-mode 还没有配置好，改天配置好后再把这个部分的代码好好整理一下。早上时间紧张，暂时没时间来处理这个了

```kotlin
private fun buildCacheKey(request: Request): String {
    val requestBody = request.body ?: return request.url.toString()
    val buffer = Buffer()
    requestBody.writeTo(buffer)

    val contentType = requestBody.contentType()
    val charset = contentType?.charset(Charsets.UTF_8) ?: Charsets.UTF_8

    if (isProbablyUtf8(buffer)) {
        val questParam = buffer.readString(charset)
        buffer.close()
        if (questParam.isBlank()) return request.url.toString()
        val builder = request.url.newBuilder()
        kotlin.runCatching {
            builder.addQueryParameter("${request.method.lowercase()}param", questParam)
            return builder.build().toString()
        }.onFailure {
            return ""
        }
    }
    return request.url.toString()
}

// 拦截器
// 我们在拦截器里做缓存，每次请求可能会是不同的策略，所以首先要拿到的就是缓存模式，
// 拿到缓存模式之后再根据不同的模式去读取或者写入操作，核心代码也就下边这几行：
override fun intercept(chain: Interceptor.Chain): Response {
    val initialRequest = chain.request()
    val strategy = CacheUtil.getCacheStrategy(initialRequest)
    val newRequest = initialRequest.rmCacheHeader()

    if (strategy == null) return chain.proceed(newRequest)// 策略为空，直接返回网络结果

    // ONLY_NETWORK 直接请求网络
    if (strategy.cacheMode == CacheMode.ONLY_NETWORK) return chain.proceed(newRequest)

    // ONLY_CACHE 只读取缓存
    if (strategy.cacheMode == CacheMode.ONLY_CACHE) {
        // 只读缓存模式，缓存为空，返回错误响应
        return (if (CacheManager.useExpiredData) mCache.getCache(strategy.cacheKey, newRequest)
                else redCache(strategy, newRequest)) ?: Response.Builder()
            .request(chain.request())
            .protocol(Protocol.HTTP_1_1)
            .code(HttpURLConnection.HTTP_GATEWAY_TIMEOUT)
            .message("no cached data")
```

```
            .body(EMPTY_RESPONSE)
            .sentRequestAtMillis(-1L)
            .receivedResponseAtMillis(System.currentTimeMillis())
            .build()
    }

    //先取缓存再取网络
    if (strategy.cacheMode == CacheMode.READ_CACHE_NETWORK_PUT) {
        val cacheResponse = redCache(strategy, newRequest)
        if (cacheResponse != null) return cacheResponse
    }

    try {
        // 开始请求网络
        val response = chain.proceed(newRequest)
        // 成功后写入缓存
        if (response.isSuccessful) {
            return cacheWritingResponse(mCache.putCache(strategy.cacheKey, response), response)
        }
        if (strategy.cacheMode == CacheMode.NETWORK_PUT_READ_CACHE) {
            return redCache(strategy, newRequest) ?: response
        }
        return response
    } catch (e: Throwable) {
        //请求失败尝试读取缓存，缓存没有或者失效，抛异常
        if (strategy.cacheMode == CacheMode.NETWORK_PUT_READ_CACHE) {
            return redCache(strategy, newRequest) ?: throw e
        }
        throw e
    }
}

// 设置缓存
// 这里不得不佩服 Retrofit 在解耦方面做的是真的强啊。我何时能有那样的思路跟想法呢。眼里只有崇拜 ~~~
// 言归正传 Retrofit 的请求头是在 Service 里边添加的，所以添加缓存策略，直接写在 Service 里。
// Retrofit 两种添加请求头的方式 @Headers 是方法注解，@Header 是参数注解。
// 再结合 Kotlin 语法可以指定默认参数，如有不同缓存模式就可以在请求的时候，去动态使用不同缓存模式。
/**
 * 使用 Header 参数注解
 */
@FormUrlEncoded
@POST("user/login")
suspend fun login(
    @Field("username") username: String,
    @Field("password") password: String,
    @Header(CacheStrategy.CACHE_MODE) cacheMode: String = CacheMode.READ_CACHE_NETWORK_PUT,
    @Header(CacheStrategy.CACHE_TIME) cacheTime: String = "10"// 过期时间，10 秒 不过期
): BaseResponse<Any>

/**
 * 使用 Headers 方法注解
 */
@Headers(
    "${CacheStrategy.CACHE_TIME}:-1", // 过期时间，-1 不过期
    "${CacheStrategy.CACHE_MODE}:${CacheMode.READ_CACHE_NETWORK_PUT}"
)
@GET("article/list/{page}/json")
suspend fun getPage(@Path("page") page: Any): BaseResponse<Page<ArticleBean>>

// 缓存的读写
// 读写操作还是用的 OkHttp 的 DiskLruCache 类。
// Okhttp 4.0.0 版本以后 就用 Kotlin 重构了。DiskLruCache 的构造函数被 internal 修饰了。
// 重构后的前几个版本还提供了 静态方法来创建。后边版本直接静态方法都移除了，这是要搞事情啊，不准备给我们用的样子。
// 不过如果用 Java 写的话就可以直接创建，Java 会忽视 internal 关键字直接过编译期。但是 Kotlin 就不行了，会报错。
// 又不想用 Java 写。还是直接用反射创建吧，没有反射干不了的事情。
internal fun getDiskLruCache(
    fileSystem: FileSystem?,
    directory: File?,
    appVersion: Int,
    valueCount: Int,
    maxSize: Long
): DiskLruCache {
    val cls = DiskLruCache::class.java
    return try {
        val runnerClass = Class.forName("okhttp3.internal.concurrent.TaskRunner")
        val constructor = cls.getConstructor(
            FileSystem::class.java,
            File::class.java,
            Int::class.java,
```

```
            Int::class.java,
            Long::class.java,
            runnerClass
        )
        constructor.newInstance(
            fileSystem,
            directory,
            appVersion,
            valueCount,
            maxSize,
            TaskRunner.INSTANCE
        )
    } catch (e: Exception) {
        try {
            val constructor = cls.getConstructor(
                FileSystem::class.java,
                File::class.java,
                Int::class.java,
                Int::class.java,
                Long::class.java,
                Executor::class.java
            )
            val executor = ThreadPoolExecutor(
                0, 1, 60L, TimeUnit.SECONDS,
                LinkedBlockingQueue(), threadFactory("OkHttp DiskLruCache", true)
            )
            constructor.newInstance(
                fileSystem,
                directory,
                appVersion,
                valueCount,
                maxSize,
                executor
            )
        } catch (e: Exception) {
            throw IllegalArgumentException("Please use okhttp 4.0.0 or later")
        }
    }
}
// 刚好 4.0.0 之后的几个版本，构造函数要提供一个线程池，4.3.0 后的版本成了 TaskRunner 了。可以都兼容一下。
// 具体的读写 IO 操作在 CacheManager.kt 这个类中，这个是根据 Okhttp 的 Cache 修改而来的。
// 全局参数
// 增加了全局 设置缓存模式、缓存时间。优先级还是 Service 中声明出来的高。
CacheManager.setCacheModel(CacheMode.READ_CACHE_NETWORK_PUT)// 设置全局缓存模式
    .setCacheTime(15 * 1000) // 设置全局 过期时间 (毫秒)
    .useExpiredData(true)// 缓存过期时是否继续使用，仅对 ONLY_CACHE 生效
// 具体使用方式：详见 Demo NetCache：  https://github.com/AleynP/net-cache
```

- 现在的难点：不知道怎么定义图片数据库，同时以 OkHTTP respnose 回来的连接起来 (可以参考下面的一个例子，虽然 MVVM 的分工可能还不是很明确，但至少是一个可以运行的版本)

4 封装：OkHttp + EventBus

- EventBut 是自己知识点面上的欠缺。借助这个极小的包装，打开一个通向真正理解这个 OkHttp 底层 EventBus 的道路。。。。
- event 有 5 个类：**BaseEvent + HttpEvent + HttpSuccessEvent + HttpErrorEvent + AppEvent**
- **RequestTag**: 请求 tag
- **MainReqeust**: 封装了 OkHttp 的回调，onResponse(...) onFailure(...) 中用 EventBus 发送数据
- **UserRequest**： 请求网络数据的方法全部在里面，把 OkHttp 的前 3 步写在这里面，第 4 布封装在了 MainRequest 中
- **BaseActivity**: 订阅事件总线，接收 EventBus 发送 (post) 的数据

4.1 BaseEvent.java

```
public class BaseEvent {
    private int id;
    private String message;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

4.2 HttpEvent

```
public class HttpEvent extends BaseEvent {
    @NonNull
        private RequestTag requestTag;
    public RequestTag getRequestTag() {
        return requestTag;
    }
    public void setRequestTag(@NonNull RequestTag requestTag) {
        this.requestTag = requestTag;
    }
}
```

4.3 HttpSuccessEvent

```
public class HttpSuccessEvent extends HttpEvent {

    private String json;
    public String getJson() {
        return json;
    }
    public void setJson(String json) {
        this.json = json;
    }
}
```

4.4 HttpErrorEvent

```
public class HttpErrorEvent extends HttpEvent {
    private int errorCode;
    private String errorMessage;
    public int getErrorCode() {
        return errorCode;
    }
    public void setErrorCode(int errorCode) {
        this.errorCode = errorCode;
    }
    public String getErrorMessage() {
        return errorMessage;
    }
    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }
}
```

4.5 AppEvent

```
public class AppEvent extends BaseEvent {

    private Object obj1;
    private Object obj2;
    private String extraInfo = null;
    private String tag;
    private int code;
```

```
    public String getExtraInfo() {
        return extraInfo;
    }
    public void setExtraInfo(String extraInfo) {
        this.extraInfo = extraInfo;
    }
    public Object getObj1() {
        return obj1;
    }
    public void setObj1(Object obj) {
        this.obj1 = obj;
    }
    public Object getObj2() {
        return obj2;
    }
    public void setObj2(Object obj2) {
        this.obj2 = obj2;
    }
    public String getTag() {
        return tag;
    }
    public void setTag(String tag) {
        this.tag = tag;
    }
    public int getCode() {
        return code;
    }
    public void setCode(int code) {
        this.code = code;
    }
}
```

4.6 RequestTag

```
public enum RequestTag {
    GET1,
    GET2,
    POST1,
    POST2,
}
```

4.7 MainRequest

```
public class MainRequest {
    private static MainRequest mainRequest;
    private MainRequest() {
        super();
    }
    public static MainRequest getInstance() {
        if (mainRequest == null) {
            mainRequest = new MainRequest();
        }
        return mainRequest;
    }

    // 异步 get
    public void makeAsyncGetRequest(Call call, final RequestTag tag) {
        call.enqueue(new Callback() {
                @Override
                public void onFailure(Call call, IOException e) {
                    httpErrorEvent(e, tag);
                }
                @Override
                public void onResponse(Call call, Response response) throws IOException {
                    httpSuccessEvent(response.body().string(), tag);
                }
            });
    }
    // 同步 get
    public void makeSyncGetRequest(final Call call, final RequestTag tag) {
        new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Response response = call.execute();
                        if (response.isSuccessful()) {
```

```
                        httpSuccessEvent(response.body().string(), tag);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                    httpErrorEvent(e, tag);
                }
            }
        }).start();
    }
    // 异步 post
    public void makeSyncPostRequest(Call call, final RequestTag tag) {
        call.enqueue(new Callback() {
            @Override
            public void onFailure(Call call, IOException e) {
                httpErrorEvent(e, tag);
            }
            @Override
            public void onResponse(Call call, Response response) throws IOException {
                httpSuccessEvent(response.body().string(), tag);
            }
        });
    }
    // 同步 post
    public void makeAsyncPostRequest(final Call call, final RequestTag tag) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Response response = call.execute();
                    if (response.isSuccessful()) {
                        httpSuccessEvent(response.body().string(), tag);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                    httpErrorEvent(e, tag);
                }
            }
        }).start();
    }
    private void httpErrorEvent(IOException e, RequestTag tag) {
        Log.d("error", "error=" + e.getMessage().toString());
        HttpErrorEvent event = new HttpErrorEvent();
        event.setErrorMessage("" + e.getMessage().toString());
        event.setRequestTag(tag);
        EventBus.getDefault().post(event);
    }
    private void httpSuccessEvent(String json, RequestTag tag) {
        Log.d("response", "response=" + json);
        HttpSuccessEvent event = new HttpSuccessEvent();
        event.setJson(json);
        event.setRequestTag(tag);
        EventBus.getDefault().post(event);
    }
}
```

4.8 UserRequest

- app 中所有的请求都放在这个类中，类名比较随意，可以自己修改成 AppRequest，比较好理解。

```
public class UserRequest {
    private OkHttpClient http;

    private UserRequest() {
        super();
        http = new OkHttpClient();
    }
    private static UserRequest userRequest;
    public static UserRequest getInstance() {
        if (userRequest == null)
            userRequest = new UserRequest();
        return userRequest;
    }

    // get 请求 不带参数
    //  同步 get
```

```
    public void syncGet(String name, String pwd) {
        String url = "http:// 192.168.1.11:8080/okhttp/json1";
        RequestTag tag = RequestTag.GET1;
        Request request = new Request.Builder().url(url).get().build();
        Call call = http.newCall(request);
        MainRequest.getInstance().makeSyncGetRequest(call, tag);
    }
    // 异步 get
    public void AsyncGet(String name, String pwd) {
        String url = "http:// 192.168.1.11:8080/okhttp/json2";
        RequestTag tag = RequestTag.GET2;
        Request request = new Request.Builder().url(url).get().build();
        Call call = http.newCall(request);
        MainRequest.getInstance().makeAsyncGetRequest(call, tag);
    }
    // 同步 post
    public void syncPost(String name, String pwd) {
        String url = "http:// 192.168.1.11:8080/okhttp/json3";
        RequestTag tag = RequestTag.POST1;
        FormBody formBody = new FormBody.Builder().add("name", name).add("pwd", pwd).build();
        Request request = new Request.Builder().post(formBody).url(url).build();
        Call call = http.newCall(request);
        MainRequest.getInstance().makeSyncPostRequest(call, tag);
    }
    // 异步 post
    public void AsyncPost(String name, String pwd) {
        String url = "http:// 192.168.1.11:8080/okhttp/json4";
        RequestTag tag = RequestTag.POST2;
        FormBody formBody = new FormBody.Builder().add("name", name).add("pwd", pwd).build();
        Request request = new Request.Builder().url(url).post(formBody).build();
        Call call = http.newCall(request);
        MainRequest.getInstance().makeAsyncPostRequest(call, tag);
    }
}
```

4.9 BaseActivity

- 订阅事件，其余 activity 只需要继承即可

```
public class BaseActivity extends AppCompatActivity {
    private ProgressDialogUtil progressDialogUtil;
    @Override
        protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EventBus.getDefault().register(this);
        progressDialogUtil = new ProgressDialogUtil(this);
    }
    @Override
        protected void onDestroy() {
        super.onDestroy();
        EventBus.getDefault().unregister(this);
    }

    @Subscribe(threadMode = ThreadMode.MAIN)
        public final void onEventBack(BaseEvent event) {
        if (event instanceof HttpErrorEvent) {
            // mark error
            httpErrorEvent((HttpErrorEvent) event);
        } else if (event instanceof HttpSuccessEvent) {
            httpSuccessEvent((HttpSuccessEvent) event);
        } else {
            applicationEvent((AppEvent) event);
        }
    }
    /**
     * 处理网络失败/错误请求
     * <p> 直接判断 HttpEvent 的 RequestTag 即可
     * @param event 错误事件
     */
    public void httpErrorEvent(HttpErrorEvent event) {}
    /**
     * 处理网络成功请求
     * <p> 直接判断 HttpEvent 的 RequestTag 即可
     * @param event 成功事件
     */
    public void httpSuccessEvent(HttpSuccessEvent event) {}
    /**
```

```
         * 处理 app 内部事件
         * @param event app 内部事件
         */
        public void applicationEvent(AppEvent event) {}
        public void showToast(String message) {
            Toast.makeText(this, message, Toast.LENGTH_SHORT).show();
        }
        public void showProgressDialog() {
            progressDialogUtil.showDialog();
        }
        public void dismissProgressDialog() {
            progressDialogUtil.dismissDialog();
        }
    }
```

4.10 使用封装

- 这样我们只需要调用一行代码就可以实现请求数据，提高了代码的简洁性。

```
UserRequest.getInstance().AsyncPost("cui", "123456");
```

- 重写这 3 个方法用于处理请求的数据

```
    @Override
    public void httpSuccessEvent(HttpSuccessEvent event) {
        super.httpSuccessEvent(event);
        if (event.getRequestTag() == RequestTag.GET1 || event.getRequestTag() == RequestTag.GET2
            || event.getRequestTag() == RequestTag.POST1 || event.getRequestTag() == RequestTag.POST2) {
            String json = event.getJson();
            tv.setText(json);
            //  TODO:  解析数据可以再写一个类 JsonParser, 将解析结果用 EventBus 发送过来, EventBus.getDefault().post(event);
        }
    }

    @Override
    public void httpErrorEvent(HttpErrorEvent event) {
        super.httpErrorEvent(event);
        if (event.getRequestTag() == RequestTag.GET1 || event.getRequestTag() == RequestTag.GET2
            || event.getRequestTag() == RequestTag.POST1 || event.getRequestTag() == RequestTag.POST2) {
            String json = event.getErrorMessage();
            tv.setText(json);
        }
    }
    @Override
    public void applicationEvent(AppEvent event) {
        super.applicationEvent(event);
        //  TODO: 接收 httpSuccessEvent(...) 中 JsonParser 成功后发送的结果
    }
```

4.11 怎么设置网络请求的缓存？

```
OkHttpClient client = new OkHttpClient.Builder()
    .connectTimeout(5, TimeUnit.SECONDS)
    .cache(new Cache(new File(this.getExternalCacheDir(), "okhttpcache"), 10 * 1024 * 1024))
    .build();
```

5 Retrofit

- 首先来了解下 Retrofit 是什么，在官网中对于 Retrofit 的描述是这样的：
  - A type-safe HTTP client for Android and Java.
  - 适用于 Android 和 Java 的类型安全的 HTTP 客户端。
  - 可以理解成一个封装好的网络请求库。

6 room 数据库相关的部分：几个相关可以用作参考的例子

6.1 另一个更好的参考例子：dagger + RecyclerView 和相应的 Adapters + BufferKnife View auto-injections

- https://github.com/SpikeKing/wcl-rx-cache-demo
- 这个 **设计思路可能显得相对过时了一点儿，四年前的仓库**，应该还有很多更好的设计与实现，但仍然是一个非常值得自己参考与学习的仓库
- **没有使用 room，而是直接操作安卓 SQLiteDatabase** ，具有上传数据的逻辑处理。所有弄懂了可以理解 Room 装填更为底层一点儿的原理
  - 如果最后时间不够用，又找不到更上层使用 Room 封装的案例用来参考学习，就可以回退到按照这个版本来参考实现
- **使用 SwipeRefreshLayout 来代替显示的刷新按钮**，相比于我加上一个 Button, 显得更为方便好用 elegant，可能会改变这个实现吧
- 这里一开始有个思想：是走本地有存储的路线，还是走本地没有存储的路线。所以，需要搞清楚，两个不同的路线之间是如何才能够动态切换的。另，这里是否涉及启动优化。Dagger 的设计思想在这里的应用与主要作用是什么 (Application layer ApiComponent 原理目的等)?
  - 这里是无关设计思路，而是在两个按钮的点击回调里，分别指向本地有缓存或是本地无缓存的两条不同的路线逻辑，所以不用把问题想复杂或是把 dagger 想得太聪明了
- 这里网络数据的刷新与获取是在 activity 的 onResume() 里自动刷新并更新 UI 数据，MVVM 用了吗分工明确了吗？仍然感觉不是很好

6.2 ApiModule.java

```
/**
 * 模块
  */
@Module
public class ApiModule {
    private Application mApplication;

    public ApiModule(Application application) {
        mApplication = application;
    }

    @Provides @Singleton
    public Application provideApplication() {
        return mApplication;
    }

    @Provides @Singleton
    GitHubClient provideGitHubClient() {
        return new GitHubClient();
    }

    @Provides ObservableRepoDb provideObservableRepoDb() {
        return new ObservableRepoDb(mApplication);
    }
}
```

6.3 ApiComponent.java

```
/**
 * 组件
 */
@Singleton @Component(modules = ApiModule.class)
public interface ApiComponent {
    void inject(NocacheActivity activity);
    void inject(CacheActivity activity);
}
```

6.4 NocacheActivity extends Activity

```
/**
 * 无缓存 Activity
 * Created by wangchenlong on 16/1/18.
 */
public class NocacheActivity extends Activity {
    @Bind(R.id.nocache_rv_list) RecyclerView mRvList;
    @Bind(R.id.nocache_pb_progress) ProgressBar mPbProgress;
    @Inject Application mApplication;
    @Inject GitHubClient mGitHubClient;
    private ListAdapter mListAdapter;

    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_nocache);
        ButterKnife.bind(this);
        ((RcApplication) getApplication()).getApiComponent().inject(this);
        LinearLayoutManager layoutManager = new LinearLayoutManager(mApplication);
        mRvList.setLayoutManager(layoutManager);
        mListAdapter = new ListAdapter();
        mRvList.setAdapter(mListAdapter);
    }

    @Override protected void onResume() {
        super.onResume();
        // 延迟 3 秒，模拟网络较差的效果
        mGitHubClient.getRepos("SpikeKing")
            .delay(3, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::onSuccess, this::onError);
        mPbProgress.setVisibility(View.VISIBLE);
    }
    private void onSuccess(ArrayList<Repo> repos) {
        mListAdapter.setRepos(repos);
        mPbProgress.setVisibility(View.INVISIBLE);
    }
    private void onError(Throwable throwable) {
        mPbProgress.setVisibility(View.INVISIBLE);
    }
}
```

6.5 这里有个小例子：

- https://github.com/Tom1881/Jet-pack/tree/master/app
- 但是我感觉上面的例子中，关于 MVVM 中的 M, V, VM 的分工逻辑处理得不好，dao 不应该出现在 view (activity/fragment) 中，应该是在 ViewModel 或是 Model 中，应用是在数据的管理中，而不是 View/Ui 中。

7 Glide 的缓存分为两种，Resource 缓存、Bitmap 缓存。

7.1 一、Resource 缓存：

- 首先 Resource 缓存就是缓存整体的图片资源文件，缓存它是为了当首次从服务器端下载下来之后，缓存到本地，如果再次使用这个图片，不用去跑网络请求，直接从本地读取，节省流量也提高访问速度。它使用的是三级缓存原理：
  - 一级缓存：内存缓存，缓存被回收的资源，使用 LRU 算法（Least Frequently Used，最近最少使用算法），当需要再次使用到被回收的资源时，直接从内存中读取；
  - 二级缓存：使用弱引用缓存正在使用的资源，当系统执行 GC 操作时，会回收没有强引用的资源。使用弱引用缓存，既可以缓存当前正在强引用使用的资源，又不阻碍系统回收无引用的资源
  - 三级缓存：磁盘缓存，网络图片下载成功后，以文件的形式缓存到磁盘中

- 1 和 2 都是内存缓存，只不过功能不一样，1 是使用 LRU 算法缓存被 GC 回收的资源，2 是用弱引用缓存正在使用的资源。在复用图片资源的时候首先从回收的内存缓存集合中查找，内存缓存的集合中没有的时候，去弱引用集合查找是否是当前正在使用，没有的话，去磁盘中查找，再没有的时候去网络中查找。

7.2 二、Bitmap 缓存：Bitmap 所占的内存大小由其三部分组成：图片宽，高和 Bitmap 质量参数。

- bitmap 内存大小 = 宽 * 高 * 质量参数所占的位数，单位是字节 b
  - ALPHA—8 就是 Alpha 是由 8 位组成的（1B）
  - ARGB_4444，4 个 4 位组成 16 位（2B）
  - ARGB_8888，4 个 8 位组成 32 位（4B）
  - RGB_565，R 是 5 位，G 是 6 位，B 是 5 位组成 16 位（2B），Glide 默认 bitmap 压缩参数就是这个 RGB_565，但是它不能显示透明度
- 先说一下为什么要进行 bitmap 压缩，比如在 recycleView 中加载大量的图片，频繁的创建和回收 Bitmap 会导致内存波动影响性能，既然这样，我们能不能缓存 Bitmap，不要让它老是 new 和销毁，这应该是 Glide 去做 Bitmap 缓存的原因，
- Bitmap 缓存算法：在 Glide 中使用 BitmapPool 来缓存 Bitmap，使用的也是 LRU 算法（最近最少使用算法），当需要使用 Bitmap 时，先从 Bitmap 的池子中选取，如果找不到合适的 Bitmap，再去创建，当使用完毕后，不再直接调用 Bitmap.recycle() 释放内存，而是缓存到 Bitmap 池子里。
- Bitmap 的缓存是以键值对的方式进行缓存的，Resource 和 Bitmap 都作为 Value，而这些值是需要一个 key 来标识缓存的内容，根据 key 可以查找和移除对应的缓存。

8 下载图片并保存到本地：rxjava 2.x+retrofit 通过动态 url 保存网络图片到本地

```
// HttpManager 类：就是一个通过单例模式实现的类，获取 retrofit 的一个实例来调用 NetApi 接口内声明的方法，此处只写关键的一部分，别的
public <T> T getHttpApi(Class<T> service) {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(BASE_URl)
        .client(getClient())
        .addConverterFactory(GsonConverterFactory.create())
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
        .build();

    return retrofit.create(service);
}
// BASE_URl 是你定义的域名比如：http://www.xxxx.com:8080 之类的

// NetApi 接口：
@GET
@Streaming
Observable<ResponseBody> downloadImg(@Url String  imgUrl);

// 注意注解：
// @GET 后面不加任何东西，平时的都是 @GET("api/getuserinfo") 之类的和上面的那个 BASE_URl 拼接起来生成 url：
// http://www.xxxx.com:8080/api/getuserinfo? 请求条件 =xx
// 然后去请求，这里采用 @Url 注解的方式就不用那么麻烦了
// @Url 此处是动态 url 即网络图片的 url，需要从外部传入，如度娘图标 url：
// https://www.baidu.com/img/superlogo_c4d7df0a003d3db9b65e9ef0fe6da1ec.png
// 用字符串的形式传入即可

// Presenter 类 ： 发起网络请求把得到的图片二进制流转化为 bitmap 对象，再通过 bitmap 对象保存到本地指定目录下
/**
 * 指定线程下载文件 (异步)，非阻塞式下载
 * @param url       图片 url
 * @param savePatch 下载文件保存目录
```

```
 * @param fileName  文件名称 (不带后缀)
 */
public void downloadFile(String url, final String savePatch, final String fileName) {
    HttpManager.getInstance().getHttpApi(NetApi.class)
        .downloadImg(url)
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.newThread())
        .subscribe(new DisposableObserver<ResponseBody>() {
                @Override
                    public void onNext(ResponseBody responseBody) {
                    Bitmap bitmap = null;
                    byte[] bys;
                    try {
                        bys = responseBody.bytes();
                        bitmap = BitmapFactory.decodeByteArray(bys, 0, bys.length);

                        try {
                            FileUtils.saveImg(bitmap, savePatch, fileName);
                            String savePath = savePatch + File.separator + fileName + ".jpg";
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }

                    if (bitmap != null) {
                        bitmap.recycle();
                    }
                }
                @Override
                    public void onError(Throwable e) {
                    //你的处理
                }
                @Override
                    public void onComplete() {
                    //你的处理
                }
            });
}
// decodeByteArray 是 BitmapFactory 内的方法，把二进制流转化为 bitmap，需要导入系统包：
// import android.graphics.BitmapFactory;

// FileUtils 类：IO 操作，把图片保存到本地：
/**
 * 保存图片到 SD 卡
 * @param bm         图片 bitmap 对象
 * @param floderPath 下载文件保存目录
 * @param fileName   文件名称 (不带后缀)
 */
public static void saveImg(Bitmap bm, String floderPath, String fileName) throws IOException {
    //如果不保存在 sd 下面下面这几行可以不加
    if (!Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)) {
        Log.e("SD 卡异常");
        return;
    }
    File folder = new File(floderPath);
    if (!folder.exists()) {
        folder.mkdirs();
    }
    String savePath = folder.getPath() + File.separator + fileName + ".jpg";
    File file = new File(savePath);
    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file));
    bm.compress(Bitmap.CompressFormat.JPEG, 80, bos);
    Log.d(savePath + " 保存成功");
    bos.flush();
    bos.close();
}
// 在你的 service 或者 activity 中调用：
mPresenter.downloadFile("https://www.baidu.com/img/superlogo_c4d7df0a003d3db9b65e9ef0fe6da1ec.png", Environment.getExternal
```

9 关于图片的处理：不仅要下载，下载后还需要自动保存到数据库

- https://blog.csdn.net/ANDROID_WangWeiDa/article/details/62284675

- 主要源码参考如下：

```
/**
 * 观察者
 */
Observer<String> observer = new Observer<String>() {
    @Override
    public void onCompleted() {
        Log.e("TAG", "oncompleted()");
    }
    @Override
    public void onError(Throwable e) {
        Log.e("TAG", "onError()");
    }
    @Override
    public void onNext(String s) {
        Log.e("TAG", "onNext()" + s);
    }
};
// 或者创建观察者的实现类：Subscriber
/**
 * 观察者（观察者的实现类）
 */
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onCompleted() {
        Log.e("TAG", "oncompleted()");
    }
    @Override
    public void onError(Throwable e) {
        Log.e("TAG", "onError()");
    }
    @Override
    public void onNext(String s) {
        Log.e("TAG", "onNext()" + s);
    }
};
// 可以说，两者的效果是一样的。
// 接着创建可观察者（被观察者）Observable

/**
 * 可观察者（被观察者）
 */
Observable observale = Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            subscriber.onNext("Hello");
            subscriber.onNext("My name is Avater!");
            subscriber.onCompleted();
        }
    });
// 好了，到此已经创建完毕，接着在 onCreate 方法中进行简单的调用：
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    observale.subscribeOn(Schedulers.io())  //订阅在 io 线程（非主线程），不会阻塞主线程
        .observeOn(AndroidSchedulers.mainThread())  //在主线程中观察
        .subscribe(observer);   //进行订阅关系
}
// Log:
// 03-15 12:06:45.837 2952-2952/com.avater.myapplication E/TAG: onNext()Hello
// 03-15 12:06:45.847 2952-2952/com.avater.myapplication E/TAG: onNext()My name is Avater!
// 03-15 12:06:45.847 2952-2952/com.avater.myapplication E/TAG: oncompleted()
// 是不是很快？是不是很懵逼？哈哈，这就对了，毕竟入门嘛，多实战，多理解！
// 下面附上一个使用 Rxjava 下载图片的例子：

private ImageView imageView;
private String url = "https://ss0.bdstatic.com/5aV1bjqh_Q23odCf/static/superman/img/logo/bd_logo1_31bdc765.png";

/**
 * 图片观察者
 */
Observer<Bitmap> bitmapOberver = new Observer<Bitmap>() {
    @Override
    public void onCompleted() {

    }
    @Override
```

```
    public void onError(Throwable e) {
        Toast.makeText(MainActivity.this, " 图片下载失败", Toast.LENGTH_SHORT).show();
    }
    @Override
    public void onNext(Bitmap bitmap) {
        imageView.setImageBitmap(bitmap);
    }
};

/**
 * 可观察者（被观察者）
 */
Observable<Bitmap> bitmapObservable = Observable.create(new Observable.OnSubscribe<Bitmap>() {
        @Override
        public void call(Subscriber<? super Bitmap> subscriber) {
            URL net;
            HttpURLConnection conn = null;
            InputStream inputStream = null;
            Bitmap bitmap = null;
            try {
                net = new URL(url);
                conn = (HttpURLConnection) net.openConnection();
                inputStream = conn.getInputStream();
                bitmap = BitmapFactory.decodeStream(inputStream);
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                conn.disconnect();
                try {
                    inputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            subscriber.onNext(bitmap);
        }
    });
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    imageView = (ImageView) findViewById(R.id.imageview);

    bitmapObservable.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(bitmapOberver);
}
```

10 用 Retrofit+Rxjava 上传图片支持多张图片的上传

```
// 1. 这是一个接口
@POST
Observable<ResponseBody> Image(@Url String url, @HeaderMap Map<String,Object> headermap,@Body MultipartBody body);
// 第一个是上传一个 第二个是上传多个

// 下面这个是一个 Retrofit 封装好的工具类
public class Retrofits{
    private MyApiService myApiService;
    public Retrofits() {
        HttpLoggingInterceptor loggingInterceptor =new HttpLoggingInterceptor();
        loggingInterceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
        OkHttpClient okHttpClient =new OkHttpClient.Builder()
            .readTimeout(20,TimeUnit.SECONDS)
            .connectTimeout(20,TimeUnit.SECONDS)
            .writeTimeout(20,TimeUnit.SECONDS)
            .addInterceptor(loggingInterceptor)
            .retryOnConnectionFailure(true)
            .build();
        Retrofit retrofit =new Retrofit.Builder()
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
//                存放的头文件
            .baseUrl(Contacts.BASE_URL)
            .client(okHttpClient)
```

```
            .build();
        myApiService =retrofit.create(MyApiService.class);
    }
    public static  Retrofits getInstance(){
        return RetroHolder.OK_UTIL;
    }
    static class RetroHolder{
        private static final Retrofits OK_UTIL =new Retrofits ();
    }
    /**
     * 封装一个上传图片
     */
    public OkUtil image(String murl,Map<String,Object> headermap,Map<String,Object> map,List<Object> list){
        MultipartBody.Builder builder = new MultipartBody.Builder().setType(MultipartBody.FORM);
        if (list.size()==1) {
            for (int i = 0; i < list.size(); i++) {
                File file = new File((String) list.get(i));
                builder.addFormDataPart("image", file.getName(),RequestBody.create(MediaType.parse("multipart/octet-stream"
            }
        }
        myApiService.Image(murl,headermap,builder.build())
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(observer);
        return Retrofits.getInstance();
    }
    /**
     * 多个图片的上传
     */
    public OkUtil pinglun(String murl,Map<String,Object> headermap,Map<String,Object> map,List<Object> list){
        MultipartBody.Builder builder = new MultipartBody.Builder().setType(MultipartBody.FORM);
        builder.addFormDataPart("commodityId",String.valueOf(map.get("commodityId")));
        if(!String.valueOf(map.get("orderId")).equals("")){
            builder.addFormDataPart("orderId",String.valueOf(map.get("orderId")));
        }
        builder.addFormDataPart("content",String.valueOf(map.get("content")));
        if (list.size()!=0) {
            for (int i = 1; i < list.size(); i++) {
                File file = new File((String) list.get(i));
                builder.addFormDataPart("image", file.getName(),RequestBody.create(MediaType.parse("multipart/octet-stream"
            }
        }
        myApiService.Image(murl,headermap,builder.build())
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(observer);
        return Retrofits.getInstance();
    }
//    重写一个观察者模式
    private Observer observer =new Observer<ResponseBody>(){
        @Override
        public void onCompleted() {
        }
        @Override
        public void onError(Throwable e) {
            if(httpListener!=null){
                httpListener.onError(e.getMessage());
            }
        }
        @Override
        public void onNext(ResponseBody responseBody) {
            if(httpListener !=null){
                try {
                    httpListener.onSuccess(responseBody.string());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    };
    public interface HttpListener{
        void onSuccess(String gsonstr);
        void onError(String error);
    }
    private HttpListener httpListener;
    public void setHttpListener(HttpListener listener){
        this.httpListener =listener;
    }
}
```

```
// 一个方法把得到的图片路径 变为 String 类型
public String getFilePath(String fileName, int requestCode, Intent data) {
    if (requestCode == 1) {
        return fileName;
    } else if (requestCode == 0) {
        Uri uri = data.getData();
        String[] proj = {MediaStore.Images.Media.DATA};
        Cursor actualimagecursor = managedQuery(uri, proj, null, null, null);
        int actual_image_column_index = actualimagecursor
            .getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
        actualimagecursor.moveToFirst();
        String img_path = actualimagecursor
            .getString(actual_image_column_index);
        // 4.0 以上平台会自动关闭 cursor, 所以加上版本判断,OK
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.ICE_CREAM_SANDWICH)
            actualimagecursor.close();
        return img_path;
    }
    return null;
}

// 一个打开图库的方法
Intent intent1 = new Intent(Intent.ACTION_PICK);
intent1.setType("image/*");
startActivityForResult(intent1,0);

// 重写一个回调方法
@Override
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if(data==null){
        return;
    }
    if(requestCode==0){
        String filePath = getFilePath(null,requestCode,data);
        /**
         * 这里是用的一个图片的上传
         */
        Map<String, Object> map = new HashMap<>();
        List<Object> list =new ArrayList<>();
        list.add(filePath);
        pressent.image(Contacts.UploadYourHead, headermap, map,list,Register.class);
    }
}
```

11 Android onSaveInstanceState()、onRestoreInstanceState() 保存和恢复被系统销毁的数据

- Android 系统的回收机制会在未经用户主动操作的情况下销毁 activity，而为了避免系统回收 activity 导致数据丢失，Android 为我们提供了 onSaveInstanceState(Bundle outState) 和 onRestoreInstanceState(Bundle savedInstanceState) 用于保存和恢复数据。

11.1 一、onSaveInstanceState(Bundle outState) 在什么时机会被调用呢?

- 答案是 **当 activity 有可能被系统回收的情况下，而且是在 onStop() 之前（之前，确定吗？好像也有可能是之后呀，比如按 HOME 键后又立即从最近任务列表启动应用时，要不要再主证实一下？）**。注意是 **有可能**，如果是已经确定会被销毁，比如 **用户按下了返回键，或者调用了 finish() 方法销毁 activity，则 onSaveInstanceState 不会被调用**。或者也可以说，**此方法只有在 activity 有可能被异常终止的情况下会被调用**。

- onSaveInstanceState 方法,onSave 方法的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的 activity，则 onSaveInstanceState 会被系统调用，这是系统的责任，因为它必须要提供一个机会让你保存你的数据。

- Activity 的销毁一般分为两种情况：

  - 当用户按返回按钮或你的 Activity 通过调用 finish() 销毁时，这属于正常销毁，此时是不需要恢复状态的，因为下次回来又是重新创建新的实例。
  - 如果 Activity 当前被停止或长期未使用，或者前台 Activity 需要更多资源以致系统必须关闭后台进程恢复内存，系统也可能会销毁 Activity，这属于非正常销毁，尽管 Activity 实例被销毁，但系统会保存其状态，这样，如果用户导航回该 Activity，系统会使用保存了该 Activity 被销毁时的状态数据来创建 Activity 的新实例。
- 屏幕旋转、键盘可用性改变、语言改变都可以归结为第二种情况；
  - 值得一提的是，如果需要模拟这种情况的 Activity 销毁，可以打开开发者选项，选择不保留活动（英文为 Do not keep activities），即可模拟内存不足时的系统行为。
  - 如何模仿 Android 系统在内存紧张的情况下把我的 activity 给 kill 掉了呢?
    * 虽然上面用到的横竖屏切换可以解决“不是用户主动调用的情况下，进程 destory 了”，但是我还是想解决这个问题
    * 开个模拟器，内存给小点（比较 lower）
    * adb shell am kill 包名（注意：一定要将当前进程弄后台后，再运行命令）
    * adb（Android Debug Bridge），am（Android Manege）有什么命令我一并贴过来
    * http://blog.csdn.net/soslinken/article/details/50245865
- 原文链接：https://blog.csdn.net/yyd_Diablo/article/details/53489840
- 总结下，**onSaveInstanceState** (Bundle outState) 会在以下情况被调用：
  - 1、当用户按下 HOME 键时。
  - 2、从最近应用中选择运行其他的程序时。
  - 3、按下电源按键（关闭屏幕显示）时。
  - 4、从当前 activity 启动一个新的 activity 时。
  - 5、屏幕方向切换时 (无论竖屏切横屏还是横屏切竖屏都会调用)。
- 在前 4 种情况下，当前 activity 的生命周期为：
  - onPause -> onSaveInstanceState -> onStop。
- 这个是我测试的结果，但是 **根据《Android 开发艺术探索》，说 onPause 和 onSaveInstanceState 的顺序是不一定的**

11.2 二、onRestoreInstanceState 什么时机被调用?

- onRestoreInstanceState(BundlesavedInstanceState) **只有在 activity 确实是被系统回收，重新创建 activity 的情况下才会被调用**。
- 比如第 5 种情况屏幕方向切换时，activity 生命周期如下：onPause -> onSaveInstanceState -> onStop -> onDestroy -> onCreate -> onStart -> onRestoreInstanceState -> onResume 在这里 onRestoreInstanceState 被调用，是因为屏幕切换时原来的 activity 确实被系统回收了，又重新创建了一个新的 activity。（顺便吐槽一下网上的那些文章说横屏切竖屏和竖屏切横屏时 activity 生命周期方法执行不一样，经自己实践证明是一样的。）
- 而按 HOME 键返回桌面，又马上点击应用图标回到原来页面时，activity 生命周期如下：onPause -> onSaveInstanceState -> onStop -> onRestart -> onStart -> onResume 因为 activity 没有被系统回收，因此 onRestoreInstanceState 没有被调用。
  - 上面我自己测的 onSaveInstanceState 是在 onStop 之后：onPause() ==> onStop() ==> onSaveInstanceState() ==> onRestart() ==> onStart() ==> onResume()
- **如果 onRestoreInstanceState 被调用了，则页面必然被回收过，则 onSaveInstanceState 必然被调用过**。

11.3 三、onCreate() 里也有 Bundle 参数，可以用来恢复数据，它和 onRestoreInstanceState 有什么区别?

- 因为 onSaveInstanceState 不一定会被调用，所以 **onCreate() 里的 Bundle 参数可能为空，如果使用 onCreate() 来恢复数据，一定要做非空判断**。
- 而 **onRestoreInstanceState 的 Bundle 参数一定不会是空值**，因为 **它只有在上次 activity 被回收了才会调用**。
- 而且 onRestoreInstanceState 是在 onStart() 之后被调用的。有时候我们需要 onCreate() 中做的一些初始化完成之后再恢复数据，用 onRestoreInstanceState 会比较方便。下面是官方文档对 onRestoreInstanceState 的说明：
  - This method is called after onStart() when the activity is being re-initialized from a previously saved state, given here in savedInstanceState. Most implementations will simply use onCreate(Bundle) to restore their state, but it is sometimes convenient to do it here after all of the initialization has been done or to allow subclasses to decide whether to use your default implementation.
  - 注意这个说明的最后一句是什么意思?
  - to allow subclasses to decide whether to use your default implementation.
- 它是说，用 onRestoreInstanceState 方法恢复数据，你可以决定是否在方法里调用父类的 onRestoreInstanceState 方法，即是否调用 super.onRestoreInstanceState(savedInstanceState);
  - 可是上面也有警告说：注意：您应始终调用 onRestoreInstanceState() 的父类实现，以便默认实现可以恢复视图层次结构的状态。所以这里要再搜索好好想一下
- 而用 onCreate() 恢复数据，你必须调用 super.onCreate(savedInstanceState);

```
//保存和恢复数据
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
//可以把要保存的静态全局变量先转成 Json
        savedInstanceState.putBoolean("MyBoolean", true);
        savedInstanceState.putDouble("myDouble", 1.9);
        savedInstanceState.putInt("MyInt", 1);
        savedInstanceState.putString("MyString", "Welcome back to Android");
        super.onSaveInstanceState(savedInstanceState);
}
  @Override
  public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
// 从 savedInstanceState 中恢复数据, 如果没有需要恢复数据 savedInstanceState 为 nul
      if (savedInstanceState != null) { // <<<<<<<<<<<<<<<<<<<<  非空判断
        boolean myBoolean = savedInstanceState.getBoolean("MyBoolean");
        double myDouble = savedInstanceState.getDouble("myDouble");
        int myInt = savedInstanceState.getInt("MyInt");
        String myString = savedInstanceState.getString("MyString");
      }
     }
//或在 onRestoreInstanceState 恢复数据
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState); // <<<<<<<<<< 这个可以调用，可以不用调用，随用户喜好
        boolean myBoolean = savedInstanceState.getBoolean("MyBoolean");
        double myDouble = savedInstanceState.getDouble("myDouble");
        int myInt = savedInstanceState.getInt("MyInt");
        String myString = savedInstanceState.getString("MyString");
}
```

12 Library Hours

1. 34007 Alvarado-Niles Rd, **Union City**, CA 94587 (4.3 miles)
2. 37055 Newark Blvd, **Newark**, CA 94560 (5-7 miles)

3. 2400 Stevenson Blvd, **Fremont**, CA 94538 (太远了)

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
1	Closed	**2–8PM**	10AM–6PM	**10AM–6PM**	**2–8PM**	**2–6PM**	**10AM–5PM**
2	Closed	12–8PM	**12–8PM**	10AM–6PM	10AM-6PM	Closed	10AM–5PM
3	1–5PM	12–8PM	12–8PM	11AM–6PM	11AM–6PM	10AM–5PM	10AM–5PM

13 OkHttp Call 实现的简单案例: 最简单的小例子

- https://www.cnblogs.com/wjtaigwh/p/6210534.html
- 继这个最简单浅显的例子之后，可以借助https://blog.51cto.com/u_15456329/4799618 再深入理解一下，并按照别人的例子再实现一遍

13.1 简单的异步 Get 请求

```
// okHttp 的基本使用 --- get 方法
String url = "https:// api.douban.com/v2/movie/top250?start=0&count=10";
// 1, 创建 OKHttpClient 对象
OkHttpClient mOkHttpClient = new OkHttpClient();
// 2, 创建一个 Request
Request request = new Request.Builder().url(url).build();
// 3, 创建一个 call 对象
Call call = mOkHttpClient.newCall(request);
// 4, 将请求添加到调度中
call.enqueue(new Callback() {
        @Override
        public void onFailure(Request request, IOException e) {
         }
         @Override
        public void onResponse(Response response) throws IOException {
            if (response.isSuccessful()) {
                final String message = response.body().string();
//  由于我们调用的 enqueue () 方法，是运行在网络线程中的，
//  所以当我们得到 json 数据后想要获取更新 UI 的话，可以开使用 handle.post() 方法在 run 方法里面更新 UI。
                handler.post(new Runnable() { //  <<<<<<<<<<<<<<<<<<<<  将从网络请求的线程结果传到主线程上
                        @Override
                        public void run() {
                            tv_message.setText(message);
                            progressBar.setVisibility(View.GONE);
                        }
                    });

            }
        }

    });
```

13.2 简单的异步 Post 请求

- 这里的 Post 请求我们以最常见的注册登录来举例。post 请求的步骤和 get 是相似的只是在创建 Request 的时候将服务器需要的参数传递进去.

```
String url = "http:// 192.168.1.123:8081/api/login";
// 1, 创建 OKhttpClient 对象
OkHttpClient mOkHttpClient = new OkHttpClient();
// 2, 创建 Request
RequestBody formBody = new FormEncodingBuilder() //  <<<<<<<<<<
    .add("username", "superadmin")
    .add("pwd", "ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87621b6c233b548baeae6956df346ec8c17f5ea10f35ee3cbc5
    .build();

Request request = new Request.Builder().url(url).post(formBody).build(); //  <<<<<<<<<<
// 3, 创建 call 对象并将请求对象添加到调度中
mOkHttpClient.newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(Request request, IOException e) {
```

```
            }
            @Override
            public void onResponse(Response response) throws IOException {
                Log.i("wangjitao", response.body().string());
            }
        });
```

13.3 OkHttp 的封装

13.3.1 CallBack 的创建

- 首选我们知道，当接口请求成功或者失败的时候我们需要将这个信息通知给用户，那么我们就需要创建一个抽象类 RequestCallBack，请求前、成功、失败、请求后这几个方法，创建 OnBefore ()、OnAfter ()、OnError ()、OnResponse () 对应

```
    // 在请求之前的方法，一般用于加载框展示
    // @param request
    public void onBefore(Request request) {}

    // 在请求之后的方法，一般用于加载框隐藏
    public void onAfter() {}

    // 请求失败的时候
    // @param request
    // @param e
    public abstract void onError(Request request, Exception e);

    // @param response
    public abstract void onResponse(T response);
```

- 由于我们每次想要的数据不一定，所以这里我们用 <T> 来接收想要装成的数据格式，并通过反射得到想要的数据类型（一般是 Bean、List）之类 ，所以 RequestCallBack 的整体代码如下：

```
// import com.google.gson.internal.$Gson$Types;
import com.squareup.okhttp.Request;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

/**
 * Created by wangjitao on 15/10/16.
 * 抽象类，用于请求成功后的回调
 */
public abstract class ResultCallback<T> {
    //这是请求数据的返回类型，包含常见的（Bean，List 等）
    Type mType;

    public ResultCallback() {
        mType = getSuperclassTypeParameter(getClass());
    }

    /**
     * 通过反射想要的返回类型
     * @param subclass
     * @return
     */
    static Type getSuperclassTypeParameter(Class<?> subclass) {
        Type superclass = subclass.getGenericSuperclass();
        if (superclass instanceof Class) {
            throw new RuntimeException("Missing type parameter.");
        }
        ParameterizedType parameterized = (ParameterizedType) superclass;
        return $Gson$Types.canonicalize(parameterized.getActualTypeArguments()[0]);
    }

    /**
     * 在请求之前的方法，一般用于加载框展示
     * @param request
     */
    public void onBefore(Request request) {}

    /**
     * 在请求之后的方法，一般用于加载框隐藏
     */
```

```
    public void onAfter() {}

    /**
     * 请求失败的时候
     * @param request
     * @param e
     */
    public abstract void onError(Request request, Exception e);

    /**
     * @param response
     */
    public abstract void onResponse(T response);
}
```

13.3.2 对 Get、Post 方法的简单封装

- 首先我们创建一个 OkHttpClientManager 类，由于是管理类，所以，单例加静态对象搞起

```
private static OkHttpClientManager mInstance;
public static OkHttpClientManager getInstance() {
    if (mInstance == null){
        synchronized (OkHttpClientManager.class) {
            if (mInstance == null)
                mInstance = new OkHttpClientManager();
        }
    }
    return mInstance;
}
```

- 在创建 Manager 对象的时候我们要把 OkHttp 的一些参数配置一下，顺便一提一下，由于我们我们异步 get、post 方法是运行在子线程中，所以这里我们添加了分发的 Handler mDelivery;，重写的 OkHttpClientManager 构造方法如下：

```
private OkHttpClientManager() {
    mOkHttpClient = new OkHttpClient();
    mOkHttpClient.setConnectTimeout(10, TimeUnit.SECONDS);
    mOkHttpClient.setWriteTimeout(10, TimeUnit.SECONDS);
    mOkHttpClient.setReadTimeout(30, TimeUnit.SECONDS);
    //cookie enabled
    mOkHttpClient.setCookieHandler(new CookieManager(null, CookiePolicy.ACCEPT_ORIGINAL_SERVER));
    mDelivery = new Handler(Looper.getMainLooper());
    mGson = new Gson();
}
```

- 前面的外部调用对象封装好了，这里我们开始来封装 Get 或 Post 方法，我这里以 Post 方法为例子，首先分析一下，post 方法会有几个参数，参数一 url，参数二参数 params，参数三 Callback（及我们上面的 RequestCallBack）参数四 flag（用于取消请求操作，可为空），基础代码如下：

```
    /**
     * 通用基础的异步的 post 请求
     * @param url
     * @param callback
     * @param tag
     */
    public void postAsyn(String url, Param[] params, final ResultCallback callback, Object tag) {
        Request request = buildPostFormRequest(url, params, tag);
        deliveryResult(callback, request);
    }
```

- 那么我们再看一下 deliveryResult 方法到底是干什么的

```
/**
 * 请求回调处理方法并传递返回值
 * @param callback Map 类型请求参数
 * @param request Request 请求
 */
private void deliveryResult(ResultCallback callback, Request request) {
    if (callback == null)
        callback = DEFAULT_RESULT_CALLBACK;
    final ResultCallback resCallBack = callback;
```

```
    // UI thread
    callback.onBefore(request);
    mOkHttpClient.newCall(request).enqueue(new Callback() {
            @Override
            public void onFailure(final Request request, final IOException e) {
                sendFailedStringCallback(request, e, resCallBack);
            }
            @Override
            public void onResponse(final Response response) {
                try {
                    final String responseMessage=response.message();
                    final String responseBody = response.body().string();
                    if(response.code()==200){
                        if (resCallBack.mType == String.class) {
                            sendSuccessResultCallback(responseBody, resCallBack);
                        } else {
                            Object o = mGson.fromJson(responseBody, resCallBack.mType);
                            sendSuccessResultCallback(o, resCallBack);
                        }
                    }else{
                        Exception exception=new Exception(response.code()+":"+responseMessage);
                        sendFailedStringCallback(response.request(), exception, resCallBack);
                    }
                } catch (IOException e) {
                    sendFailedStringCallback(response.request(), e, resCallBack);
                } catch (com.google.gson.JsonParseException e) {//Json 解析的错误
                    sendFailedStringCallback(response.request(), e, resCallBack);
                }
            }
        });
}
```

- 可以看到，这个方法主要是发出请求并对请求后的数据开始回调，这样我们就基本上封装好了一个 post 方法了，把代码这一部分的代码贴出来看看

```
public class OkHttpClientManager {
    private static final String TAG = "com.qianmo.httprequest.http.OkHttpClientManager";

    private static OkHttpClientManager mInstance;
    public static OkHttpClientManager getInstance() {
        if (mInstance == null) {
            synchronized (OkHttpClientManager.class) {
                if (mInstance == null)
                    mInstance = new OkHttpClientManager();
            }
        }
        return mInstance;
    }

    // 默认的请求回调类
    private final ResultCallback<String> DEFAULT_RESULT_CALLBACK = new ResultCallback<String>(){
        @Override
        public void onError(Request request, Exception e) {}
        @Override
        public void onResponse(String response) {}
    };

    private OkHttpClient mOkHttpClient;
    private Handler mDelivery;
    private Gson mGson;
    private GetDelegate mGetDelegate = new GetDelegate();
    private PostDelegate mPostDelegate = new PostDelegate();
    private DownloadDelegate mDownloadDelegate = new DownloadDelegate();

    private OkHttpClientManager() {
        mOkHttpClient = new OkHttpClient();
        mOkHttpClient.setConnectTimeout(10, TimeUnit.SECONDS);
        mOkHttpClient.setWriteTimeout(10, TimeUnit.SECONDS);
        mOkHttpClient.setReadTimeout(30, TimeUnit.SECONDS);
        // cookie enabled
        mOkHttpClient.setCookieHandler(new CookieManager(null, CookiePolicy.ACCEPT_ORIGINAL_SERVER));
        mDelivery = new Handler(Looper.getMainLooper());
        mGson = new Gson();
    }

    /**
     * 外部可调用的 Post 异步请求方法
     * @param url 请求 url
```

```
     * @param params
     * @param callback 请求完成后回调类
     */
    public static void postAsyn(String url, Map<String, String> params, final ResultCallback callback) {
        getInstance().getPostDelegate().postAsyn(url, params, callback, null);
    }

    /**
     * 异步的 post 请求
     * @param url
     * @param params
     * @param callback
     * @param tag
     */
    public void postAsyn(String url, Map<String, String> params, final ResultCallback callback, Object tag) {
        Param[] paramsArr = map2Params(params);
        postAsyn(url, paramsArr, callback, tag);
    }
    /**
     * 通用基础的异步的 post 请求
     * @param url
     * @param callback
     * @param tag
     */
    public void postAsyn(String url, Param[] params, final ResultCallback callback, Object tag) {
        Request request = buildPostFormRequest(url, params, tag);
        deliveryResult(callback, request);
    }

    /**
     * 请求回调处理方法并传递返回值
     * @param callback Map 类型请求参数
     * @param request Request 请求
     */
    private void deliveryResult(ResultCallback callback, Request request) {
        if (callback == null)
            callback = DEFAULT_RESULT_CALLBACK;
        final ResultCallback resCallBack = callback;
        // UI thread
        callback.onBefore(request);
        mOkHttpClient.newCall(request).enqueue(new Callback() {
                @Override
                public void onFailure(final Request request, final IOException e) {
                    sendFailedStringCallback(request, e, resCallBack);
                }
                @Override
                public void onResponse(final Response response) {
                    try {
                        final String responseMessage=response.message();
                        final String responseBody = response.body().string();
                        if (response.code()==200){
                            if (resCallBack.mType == String.class) {
                                sendSuccessResultCallback(responseBody, resCallBack);
                            } else {
                                Object o = mGson.fromJson(responseBody, resCallBack.mType);
                                sendSuccessResultCallback(o, resCallBack);
                            }
                        } else{
                            Exception exception=new Exception(response.code()+":"+responseMessage);
                            sendFailedStringCallback(response.request(), exception, resCallBack);
                        }
                    } catch (IOException e) {
                        sendFailedStringCallback(response.request(), e, resCallBack);
                    } catch (com.google.gson.JsonParseException e) {// Json 解析的错误
                        sendFailedStringCallback(response.request(), e, resCallBack);
                    }
                }
            });
    }
    /**
     * 处理请求成功的回调信息方法
     * @param object 服务器响应信息
     * @param callback 回调类
     */
    private void sendSuccessResultCallback(final Object object, final      ResultCallback callback) {
        mDelivery.post(() -> {
                callback.onResponse(object);
                callback.onAfter();
            });
```

```
    }
}
```

14 转屏等几种情况下 activity 的销毁与否，销毁与重建，数据保存

14.1 activity 被系统超脱用户期望被销毁时数据的保存与恢复

- 缺省状态下，**系统会把每一个 View 对象保存起来（比如 EditText 对象中的文本，ListView 中的滚动条位置等）**，即如果 activity 实例被销毁和重建，那么不需要你编码，layout 状态会恢复到前次状态。
- 默认的实现中存储了 activity 的 view 系列的状态，比如文本和滚动条位置等。要存储额外的信息，必须自己实现 onSaveInstanceState()，并且给 Bundle object 加上键值对。
- 但是如果你的 activity 需要恢复更多的信息，比如成员变量信息，则需要自己动手写了
- ListView RecyclerView 位点数据的保存与恢复 (当前选择或是活动位点 index position)
- List<Object> 的保存：数据库

14.2 对图片的保存

- https://blog.csdn.net/candy_rainbow/article/details/54381949?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ETopNSimi
7Edefault-1-54381949-blog-100525959.topnsimilarv1&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ETopNSimilar%7Edefault-1-54381949-blog
topnsimilarv1&utm_relevant_index=1
- **onRetainNonConfigurationInstance() 方法：**
- 这个方法也会在活动倍销毁钱调用，我是在做头像上传的时候遇到了这样的一个问题：每次选择好照片显示在 Activity 中的时候，如果横竖屏切换了，那么会重新加载布局，也就是说重新调用 onCreate 方法，之前的活动中的图片当然也就没有了. 那么问题来了，如何保存已经加载好的图片呢，我使用的就是 Activity 提供的 onRetainNonConfigurationInstance() 方法，这个方法与 getLastNonConfigurationInstance()（也是 Activity 提供的）是一起用的，我们可以重写这个方法对图片 Uri 进行保存：

```
public Object onRetainNonConfigurationInstance() {
    return imageUri;
}
```

然后，在 onCreate 方法里面通过 getLastNonConfigurationInstance 方法进行接收：

```
imageUri = (Uri)getLastNonConfigurationInstance();
```

- 接受到了之后，进行判断，如果是 null 则不加载图片，如果不是 null，则直接加载
- 横竖屏切换时候 activity 的生命周期：这个就打印日志再验证一遍吧。。。。。

```
1、不设置 Activity 的 android:configChanges 时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次
2、设置 Activity 的 android:configChanges="orientation" 时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次
3、设置 Activity 的 android:configChanges="orientation|keyboardHidden" 时，切屏不会重新调用各个生命周期，只会执行 onConfigurati
```

- 一般情况下 Configuration 的改变会导致 Activity 被销毁重建，但也有办法让指定的 Configuration 改变时不重建 Activity，方法是在 AndroidManifest.xml 里通过 android:configChanges 属性指定需要忽略的 Configuration 名字，例如下面这样：

```
<activity
    android:name=".MyActivity"
     android:configChanges="orientation|keyboardHidden|navigation|screenSize"
    android:label="@string/app_name"/>
```

- 这样设置以后，当屏幕旋转时 Activity 对象不会被销毁——作为替代，Activity 的 onConfigurationChanged() 方法被触发，在这里开发者可以获取到当前的屏幕方向以便做必要的更新。既然这种情况下的 Activity 不会被销毁，旋转后 Activity 里正显示的信息（例如文本框中的文字）也就不会丢失了。
- 假如你的应用里，横屏和竖屏使用同一个 layout 资源文件，onConfigurationChanged() 里甚至可以什么都不做。但如果横屏与竖屏使用不同的 layout 资源文件，例如横屏用 res/layout-land/main.xml，竖屏用 res/layout-port/main.xml，则必须在 onConfigurationChanged() 里重新调用 setContentView() 方法以便新的 layout 能够生效，这时虽然 Activity 对象没有销毁，但界面上的各种控件都被销毁重建了，你需要写额外的代码来恢复界面信息。

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
     super.onConfigurationChanged(newConfig);
     if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE)
         Toast.makeText(this, " 横屏模式", Toast.LENGTH_SHORT).show();
     else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT)
         Toast.makeText(this, " 竖屏模式", Toast.LENGTH_SHORT).show();
}
```

- （注：官方不推荐使用这种方法。）
- 销毁当前的 Activity：
  - 如果销毁当前的 Activity，那么就要重写 Activity 的两个方法 onSaveInstanceState（）和 onRestoreInstanceState（），显然从方法名字可以看出一个是保存
- 另一种保存-恢复现场的方法
  - 实现 onRetainNonConfigurationInstance() 方法保存数据，使用方法和前面的 onSaveInstanceState(Bundle) 差不多。

```
/* 保存 */
 @Override
public Object onRetainNonConfigurationInstance() {
     final MyDataObject data = collectMyLoadedData();
    return data;
}
/* 重建 */
@Override
public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);
     final MyDataObject data = (MyDataObject) getLastNonConfigurationInstance();
    if (data == null) // 表示不是由于 Configuration 改变触发的 onCreate()
         data = loadMyData();
}
```

  - activity 的销毁和重建有时候不一定是由屏幕旋转引起的，所以还是建议使用保存-恢复现场的方法。

15 Retrofit + RxJava + Json 数据解析

- **网络数据解析**：我这里得到了网络数据，可是好像我并没有解析数据出来，这整个过程我可能还少了这比较关键的一个步骤
  - 当对这类框架和 OOP 设计有了更好的了解，就可以自定义解析类来在获得数据前自动解析为自己想要的类型

    - 注意这里提供的 API 其实并没有 code msg 之类的信息，而是直接的结果；再想一下：为什么 OkHttp 的拦截器能够得到 code 200 呢？这里 Response 返回基本信息还有点儿糊涂
    - 可以参考这个例子：https://developer.aliyun.com/article/609862
  - 需要一个如下的步骤来解析从网络上拿到返回回来的数据

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://www.kuaidi100.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();

RetrofitService service = retrofit.create(RetrofitService.class);
Call<PostInfo> call = service.getPostInfo("yuantong", "11111111111");
call.enqueue(new Callback<PostInfo>() {
        @Override
            public void onResponse(Call<PostInfo> call, Response<PostInfo> response) {
            Log.i("http 返回：", response.body().toString() + "");
        }
        @Override
            public void onFailure(Call<PostInfo> call, Throwable t) {
        }
    });
```

16 RxJava 线程调度

- 在 RxJava 中，我们可以自行指定事件产生和事件消费的线程，可以通过 RxJava 中的 Scheduler 来实现。Scheduler

16.1 RxJava 内置的 5 个 Scheduler

- Schedulers. **immediate()**: 直接在当前线程运行,相当于不指定线程。这是默认的 Scheduler，但是为了防止被错误使用，在 RxJava2 中已经被移除了。
- Schedulers. **newThread()**: 开启新线程，并在新线程执行操作。
- Schedulers. **io()**: I/O 操作（**读写文件、读写数据库、网络信息交互等**）所使用的 Scheduler。行为模式和 newThread() 差不多，区别在于 io() 的内部实现是是用一个无数量上限的线程池，可以重用空闲的线程，因此 **多数情况下 io() 比 newThread() 更有效率。不要把计算工作放在 io() 中，可以避免创建不必要的线程。**
- Schedulers. **computation()**: 计算所使用的 Scheduler，例如图形的计算。这个 Scheduler 使用的固定的线程池，大小为 CPU 核数。不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU。
- Schedulers. **trampoline()**: 主要用于延迟工作任务的执行。当我们想在当前线程执行一个任务时，但并不是立即，我们可以用.trampoline() 将它入队，trampoline 将会处理它的队列并且按序运行队列中每一个任务。
- Android 特有的 Scheduler
  - AndroidSchedulers. **mainThread()**: 指定的操作将在 Android 的主线程中进行，如 UI 界面的更新操作。

16.2 线程的控制

- **subscribeOn(): 指定事件产生的线程**，例如 subscribeOn(Schedulers.io()) 可以指定被观察者的网络请求、文件读写等操作放置在 io 线程。
- **observeOn(): 指定事件消费的线程**，例如 observeOn(AndroidSchedulers.mainThread()) 指定 Subscriber 中的方法在主线程中运行。

- 在 subscribe() 之前写上两句 subscribeOn(Scheduler.io()) 和 observeOn(AndroidSchedulers.mainT 的使用方式非常常见，它适用于多数的 < 后台线程取数据，主线程显示 > 的程序策略。

17 其它相对比较有参考价值的链接

- https://blog.51cto.com/u_15456329/4799618

18 Retrofit + RxJava 的进一步深入理解：一个详尽的封装讲解与归纳

- 封装特点：
  - https://blog.51cto.com/jun5753/4925616 居然复制不下来，死破网太慢了。。。。。

18.1 BaseResponse 就是对基础数据进行封装处理。

```
实现步骤：
1、根据基础数据定义 BaseResponse
2、修改 API 接口返回数据类型
3、对基础数据统一处理
```

18.1.1 1、根据基础数据定义 BaseResponse

```
public class BaseResponse<T> {
    private int res_code;
    private String err_msg;
    private T demo;
    public int getRes_code() {
        return res_code;
    }
    public void setRes_code(int res_code) {
        this.res_code = res_code;
    }
    public String getErr_msg() {
        return err_msg;
    }
    public void setErr_msg(String err_msg) {
        this.err_msg = err_msg;
    }
    public T getDemo() {
        return demo;
    }
    public void setDemo(T demo) {
        this.demo = demo;
    }
}
```

- 当然我们需求数据也需重新定义

```
public class Demo {
    @Override
    public String toString() {
        return "Demo{" + "id='" + id + '\'' +
                ", appid='" + appid + '\'' +
                ", name='" + name + '\'' +
                ", showtype='" + showtype + '\'' +
                '}';
    }
    private String id;
    private String appid;
    private String name;
    private String showtype;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
```

```
    }
    public String getAppid() {
        return appid;
    }
    public void setAppid(String appid) {
        this.appid = appid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getShowtype() {
        return showtype;
    }
    public void setShowtype(String showtype) {
        this.showtype = showtype;
    }
}
```

18.1.2 2、修改 API 接口返回数据类型

```
// @GET(Constans.retrofit)
// Observable<Bean> getRetrofit1();
// 把原先的 Bean 类分成 BaseResponse 基础数据与 Demo 需求数据两个类
@GET(Constans.retrofit)
Observable<BaseResponse<Demo>> getRetrofit2();
```

18.1.3 3、对基础数据统一处理

```
/**
 * 创建 Base 抽象类实现 Observer
 */
public abstract class BaseObserver<T> implements Observer<BaseResponse<T>> {
    private static final String TAG = "BaseObserver";
    @Override
    public void onSubscribe(Disposable d) {
        Log.e(TAG, "onSubscribe: " );
    }
    @Override
    public void onNext(BaseResponse<T> response) {
        //在这边对 基础数据 进行统一处理  举个例子:
        if(response.getRes_code()==200){
            onSuccess(response.getDemo());
        }else{
            onFailure(null,response.getErr_msg());
        }
    }
    @Override
    public void onError(Throwable e) {
        Log.e(TAG, "Throwable: " + e.getMessage());
    }
    @Override
    public void onComplete() {
        Log.e(TAG, "onComplete: " );
    }
    public abstract void onSuccess(T demo);
    public abstract void onFailure(Throwable e,String errorMsg);
}
```

18.1.4 请求网络数据

```
 retrofit.create(ApiUrl.class)
                .getRetrofit2()
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                // 绑定生命周期 // <<<<<<<<<<<<<<<<<<<
                .compose(bindUntilEvent(ActivityEvent.DESTROY)) // <<<<<<<<<<<<<<<<<<<
                .subscribe(new BaseObserver<Demo>(){
                    @Override
                    public void onSuccess(Demo demo) {
                    }
                    @Override
                    public void onFailure(Throwable e, String errorMsg) {
```

```
                }
            });
// 打印结果:  onSuccess: Demo{id='1001', appid='1021', name='sss', showtype='text'}
```

18.2 RxHelper 调度类封装

- RxHelper 主要是对执行线程和绑定生命周期几个方法进行封装，

```
/**
 * 调度类: 通过这个调度类封闭订阅线程 IO 与观察者线程主线程的全自动切换
 */
public class RxHelper {
    public static <T> ObservableTransformer<T, T> observableIO2Main(final Context context) {
        return upstream -> {
            Observable<T> observable = upstream.subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThread());
            return composeContext(context, observable);
        };
    }
    public static <T> ObservableTransformer<T, T> observableIO2Main(final RxFragment fragment) {
        return upstream -> upstream.subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread()).compose(fragment.<T>bindToLifecycle());
    }
    public static <T> FlowableTransformer<T, T> flowableIO2Main() {
        return upstream -> upstream
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
    private static <T> ObservableSource<T> composeContext(Context context, Observable<T> observable) {
        if(context instanceof RxActivity) {
            return observable.compose(((RxActivity) context).bindUntilEvent(ActivityEvent.DESTROY));
        } else if(context instanceof RxFragmentActivity){
            return observable.compose(((RxFragmentActivity) context).bindUntilEvent(ActivityEvent.DESTROY));
        }else if(context instanceof RxAppCompatActivity){
            return observable.compose(((RxAppCompatActivity) context).bindUntilEvent(ActivityEvent.DESTROY));
        }else {
            return observable;
        }
    }
}
```

18.3 Retrofit 初始化封装

- 我们不可能每次要请求网络就重复去创建初始化 Retrofit。所以我们需要对 Retrofit 进行单例封装。

```
import android.support.annotation.NonNull;
import com.jakewharton.retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory;
import java.util.concurrent.TimeUnit;
import okhttp3.OkHttpClient;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;
/**
 * Retrofit 封装
 */
public class RetrofitUtils {
    private static final String TAG = "RetrofitUtils";
    private static ApiUrl mApiUrl;
     // 单例模式
    public static ApiUrl getApiUrl() {
        if (mApiUrl == null) { // <<<<<<<<<<<<<<<<<<<<
            synchronized (RetrofitUtils.class) {
                if (mApiUrl == null) // <<<<<<<<<<<<<<<<<<<<
                    mApiUrl = new RetrofitUtils().getRetrofit();
            }
        }
        return mApiUrl;
    }
    private RetrofitUtils(){ }

    public ApiUrl getRetrofit() {
        // 初始化 Retrofit
        ApiUrl apiUrl = initRetrofit(initOkHttp()) .create(ApiUrl.class);
        return apiUrl;
```

```
    }
     // 初始化 Retrofit
    @NonNull
    private Retrofit initRetrofit(OkHttpClient client) {
        return new Retrofit.Builder()
            .client(client)
            .baseUrl(Constans.BaseUrl)
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .addConverterFactory(GsonConverterFactory.create())
            .build();
    }
    // 初始化 okhttp
    @NonNull
    private OkHttpClient initOkHttp() {
        return new OkHttpClient().newBuilder()
            .readTimeout(Constans.DEFAULT_TIME, TimeUnit.SECONDS)   // 设置读取超时时间
            .connectTimeout(Constans.DEFAULT_TIME, TimeUnit.SECONDS)// 设置请求超时时间
            .writeTimeout(Constans.DEFAULT_TIME,TimeUnit.SECONDS)   // 设置写入超时时间
            .addInterceptor(new LogInterceptor()) // 添加打印拦截器
            .retryOnConnectionFailure(true)       // 设置出现错误进行重新连接。
            .build();
    }
}
```

18.4 细节完善

18.4.1 1、服务错误信息处理

- BaseObserver 对请求成功数进行了统一处理，但并未对服务器返回错误进行处理。
- 这里从某个大神 Copy 了个工具类 RxExceptionUtils 来对错误信息进行处理。
- 具体代码如下：

```
import org.json.JSONException;
import java.net.SocketTimeoutException;
import java.net.UnknownHostException;
import java.text.ParseException;
import retrofit2.HttpException;
/**
 * 异常处理
 */
public class RxExceptionUtil {
    public static String exceptionHandler(Throwable e){
        String errorMsg = " 未知错误";
        if (e instanceof UnknownHostException) {
            errorMsg = " 网络不可用";
        } else if (e instanceof SocketTimeoutException) {
            errorMsg = " 请求网络超时";
        } else if (e instanceof HttpException) {
            HttpException httpException = (HttpException) e;
            errorMsg = convertStatusCode(httpException);
        } else if (e instanceof ParseException || e instanceof JSONException
                    || e instanceof JSONException) {
            errorMsg = " 数据解析错误";
        }
        return errorMsg;
    }
    private static String convertStatusCode(HttpException httpException) {
        String msg;
        if (httpException.code() >= 500 && httpException.code() < 600) {
            msg = " 服务器处理请求出错";
        } else if (httpException.code() >= 400 && httpException.code() < 500) {
            msg = " 服务器无法处理请求";
        } else if (httpException.code() >= 300 && httpException.code() < 400) {
            msg = " 请求被重定向到其他页面";
        } else {
            msg = httpException.message();
        }
        return msg;
    }
}
```

18.4.2 2、添加“正在加载”弹窗

```
import android.app.ProgressDialog;
import android.content.Context;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.widget.Toast;
import io.reactivex.disposables.Disposable;
/**
 * Observer 加入加载框: 就是把加载框封装到抽象蕨类里去
 * @param <T>
 */
public abstract class MyObserver<T> extends BaseObserver<T> {
    private Context mContext;

    private boolean mShowDialog;
    private ProgressDialog dialog;
    private Disposable d; // <<<<<<<<<<<<<<<<<<< 这个需要再好好理解一下

    public MyObserver(Context context, Boolean showDialog) {
        mContext = context;
        mShowDialog = showDialog;
    }
    public MyObserver(Context context) {
        this(context,true);
    }

    @Override public void onSubscribe(Disposable d) {
        this.d = d;
        if (!isConnected(mContext)) {
            Toast.makeText(mContext," 未连接网络",Toast.LENGTH_SHORT).show();
            if (d.isDisposed())
                d.dispose();
        } else {
            if (dialog == null && mShowDialog == true) {
                dialog = new ProgressDialog(mContext);
                dialog.setMessage(" 正在加载中");
                dialog.show();
            }
        }
    }
    @Override public void onError(Throwable e) {
        if (d.isDisposed())
            d.dispose();
        hidDialog();
        super.onError(e);
    }
    @Override public void onComplete() {
        if (d.isDisposed())
            d.dispose();
        hidDialog();
        super.onComplete();
    }
    public void hidDialog() {
        if (dialog != null && mShowDialog == true)
            dialog.dismiss();
        dialog = null;
    }
    /**
     * 是否有网络连接, 不管是 wifi 还是数据流量
     * @param context
     */
    public static boolean isConnected(Context context) {
        ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo info = cm.getActiveNetworkInfo();
        if (info == null) return false;
        boolean available = info.isAvailable();
        return available;
    }
}
```

18.4.3 3、Retorfit 请求方法汇总

- ApiUrl 类

```
import io.reactivex.Observable;
```

```
import okhttp3.MultipartBody;
import okhttp3.RequestBody;
import okhttp3.ResponseBody;

import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.GET;
import retrofit2.http.POST;
import retrofit2.http.PUT;
import retrofit2.http.DELETE;
import retrofit2.http.Field;
import retrofit2.http.FieldMap;
import retrofit2.http.FormUrlEncoded;
import retrofit2.http.Header;
import retrofit2.http.HeaderMap;
import retrofit2.http.Headers;
import retrofit2.http.Multipart;
import retrofit2.http.Part;
import retrofit2.http.PartMap;
import retrofit2.http.Path;
import retrofit2.http.Query;
import retrofit2.http.QueryMap;
import retrofit2.http.Streaming;
import retrofit2.http.Url;

public interface ApiUrl {

    @GET(Constans.retrofit)
    Call<Bean> getRetrofit();

    @GET(Constans.retrofit)
    Observable<BaseResponse<Demo>> getDemo();

    /**
     * TODO Get 请求：带参数 和 不带参数
     */
    // 第一种方式：GET 不带参数
    @GET("retrofit.txt")
    Observable<BaseResponse<Demo>> getUser();
    @GET
    Observable<Demo> getUser(@Url String url);
    @GET
    Observable<Demo> getUser1(@Url String url); // 简洁方式，直接获取所需数据

    // 第二种方式：GET 带参数
    @GET("api/data/{type}/{count}/{page}")
    Observable<Demo> getUser(@Path("type") String type, @Path("count") int count, @Path("page") int page);
    // 第三种方式：GET 带请求参数：https://api.github.com/users/whatever?client_id=xxxx&client_secret=yyyy
    @GET("users/whatever")
    Observable<Demo> getUser(@Query("client_id") String id, @Query("client_secret") String secret);
    @GET("users/whatever")
    Observable<Demo> getUser(@QueryMap Map<String, String> info);

     // TODO POST 请求
    // 第一种方式：@Body
    @Headers("Accept:application/json")
    @POST("login")
    Observable<Demo> postUser(@Body RequestBody body);
    // 第二种方式：@Field
    @Headers("Accept:application/json")
    @POST("auth/login")
    @FormUrlEncoded
    Observable<Demo> postUser(@Field("username") String username, @Field("password") String password);
    // 多个参数
    Observable<Demo> postUser(@FieldMap Map<String, String> map);

     // TODO DELETE
    @DELETE("member_follow_member/{id}")
    Observable<Demo> delete(@Header("Authorization") String auth, @Path("id") int id);

     // TODO PUT
    @PUT("member")
    Observable<Demo> put(@HeaderMap Map<String, String> headers,
                         @Query("nickname") String nickname);

     // TODO 文件上传
    @Multipart
    @POST("upload")
    Observable<ResponseBody> upload(@Part("description") RequestBody description, @Part MultipartBody.Part file);
```

```
    // 亲测可用
    @Multipart
    @POST("member/avatar")
    Observable<Demo> uploadImage(@HeaderMap Map<String, String> headers, @Part MultipartBody.Part file);

     // 多文件上传
    @Multipart
    @POST("register")
    Observable<ResponseBody> upload(@PartMap Map<String, RequestBody> params, @Part("description") RequestBody description)
    // Observable<ResponseBody> upload(@Part() List<MultipartBody.Part> parts);

    @Multipart
    @POST("member/avatar")
    Observable<Demo> uploadImage1(@HeaderMap Map<String, String> headers, @Part List<MultipartBody.Part> file);

     // 来自 https://blog.csdn.net/impure/article/details/79658098
     // @Streaming 这个注解必须添加，否则文件全部写入内存，文件过大会造成内存溢出
    @Streaming
    @GET
    Observable<ResponseBody> download(@Header("RANGE") String start, @Url String url);
}
```

18.4.4 4、提交参数方式汇总（可忽略）

```
/**
 * 提交参数方式
 */
public class RequestUtils {

    // Get 请求 demo
    // @param context
    // @param observer
    public static void getDemo(RxAppCompatActivity context, MyObserver<Demo> observer){
        RetrofitUtils.getApiUrl()
            .getDemo().compose(RxHelper.observableIO2Main(context))
            .subscribe(observer);
    }

    // Post 请求 demo
    // @param context
    // @param consumer
    public static void postDemo(RxAppCompatActivity context, String name, String password, Observer<Demo> consumer){
        RetrofitUtils.getApiUrl()
            .postUser(name,password).compose(RxHelper.observableIO2Main(context))
            .subscribe(consumer);
    }

    // Put 请求 demo
    // @param context
    // @param consumer
    public static void putDemo(RxFragment context, String access_token,Observer<Demo> consumer){
        Map<String, String> headers = new HashMap<String, String>();
        headers.put("Accept","application/json");
        headers.put("Authorization",access_token);
        RetrofitUtils.getApiUrl()
            .put(headers," 厦门").compose(RxHelper.observableIO2Main(context))
            .subscribe(consumer);
    }

    // Delete 请求 demo
    // @param context
    // @param consumer
    public static void deleteDemo(RxFragment context, String access_token,Observer<Demo> consumer){
        RetrofitUtils.getApiUrl()
            .delete(access_token,1).compose(RxHelper.observableIO2Main(context))
            .subscribe(consumer);
    }

    // 上传图片
    // @param context
    // @param observer
    public static void upImagView(RxFragment context, String  access_token,String str, Observer<Demo>  observer){
        File file = new File(str);
//        File file = new File(imgPath);
        Map<String,String> header = new HashMap<String, String>();
        header.put("Accept","application/json");
```

```
        header.put("Authorization",access_token);
//        File file =new File(filePath);
        RequestBody reqFile = RequestBody.create(MediaType.parse("image/*"), file);
//        RequestBody requestFile =
//                RequestBody.create(MediaType.parse("multipart/form-data"), file);
        MultipartBody.Part body =
            MultipartBody.Part.createFormData("file", file.getName(), reqFile);
        RetrofitUtils.getApiUrl().uploadImage(header,body).compose(RxHelper.observableIO2Main(context))
            .subscribe(observer);
    }

    // 上传多张图片
    // @param files
    public static void upLoadImg(RxFragment context,String access_token,List<File> files, Observer<Demo>  observer1){
        Map<String,String> header = new HashMap<String, String>();
        header.put("Accept","application/json");
        header.put("Authorization",access_token);
        MultipartBody.Builder builder = new MultipartBody.Builder()
            .setType(MultipartBody.FORM);//表单类型
        for (int i = 0; i < files.size(); i++) {
            File file = files.get(i);
            RequestBody photoRequestBody = RequestBody.create(MediaType.parse("image/*"), file);
            builder.addFormDataPart("file", file.getName(), photoRequestBody);
        }
        List<MultipartBody.Part> parts = builder.build().parts();
        RetrofitUtils.getApiUrl().uploadImage1(header,parts).compose(RxHelper.observableIO2Main(context))
            .subscribe(observer1);
    }
}
```

18.5 总结

- 如若加上 RequestUtils 则代码中请求网络方式如下：

```
RequestUtils.getDemo(this, new MyObserver<Demo>(this) {
    @Override
    public void onSuccess(Demo result) {
        tv_retrofit.setText(result.toString());
    }
    @Override
    public void onFailure(Throwable e, String errorMsg) {
        tv_retrofit.setText(errorMsg);
    }
});
```

- 若后台返回的 Demo 不是个对象而是数组咋办不用慌
- 其他相关的地方也要加上 List<>

```
RequestUtils.getDemoList(
    this,
    new MyObserver<List<Demo>>(this) {
        @Override
            public void onSuccess(List<Demo> result) {
            for (Demo demo : result){
                Log.e(TAG, "onSuccess: "+demo.toString() );
            }
            tv_retrofit.setText(result.toString());
        }
        @Override
            public void onFailure(Throwable e, String errorMsg) {
            tv_retrofit.setText(errorMsg);
        }
    });
```

- 附上 Url 链接:

```
public final static  String BaseUrl = "http://120.78.186.81/api/";
public final static  String retrofit = "values/5";
public final static  String retrofitList = "values";
```

- 项目的 GitHub 地址：https://github.com/DayorNight/RxjavaRetrofit2

18.6 RxJava 观察者模式，与公用接口回调类的设计实现

19 RxJava: 防止 Rxjava 引发的内存泄漏: 需要把这部分理解清楚并整合到自己的项目中去

19.1 哪些场景会出现

- 使用 RxJava 发布一个订阅后，当页面被 finish，此时订阅逻辑还未完成，如果没有及时取消订阅，就会导致 Activity/Fragment 无法被回收，从而引发内存泄漏. 场景还比较多：Activity 中执行异步任务、Presenter 或 ViewModel 中执行异步任务、Manager、Adapter 相关类中执行异步任务。
- 如果代码如下：

```
Observable.create(new ObservableOnSubscribe<XXX>() {
        @Override
            public void subscribe(final ObservableEmitter<XXX> emitter) throws Exception {
            Observable.zip(.....)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<ImpressionBean>() {
                        @Override
                            public void accept(XXX impressionBean) throws Exception {
                            emitter.onNext(impressionBean);
                        }
                    }, new Consumer<Throwable>() {
                        @Override
                            public void accept(Throwable throwable) throws Exception {
                            emitter.onNext(new XXX());
                        }
                    });
        }
    });
```

- 当使用 RxJava 订阅并执行耗时任务后，当 Activity 被 finish 时，如果耗时任务还未完成，没有及时取消订阅，就会导致 Activity 无法被回收，从而引发内存泄漏。为了解决这个问题，就产生了 RxLifecycle，让 RxJava 变得有生命周期感知，使得其能及时取消订阅，避免出现内存泄漏问题。

19.2 解决办法

- 把整个 Observable.create() . . . 执行 subscribe（）方法后赋值为 Disposable，然后再销毁页面调用下面方法

```
if (disposable != null && !disposable.isDisposed()) {
    disposable.dispose();
}
```

- Presenter 或 ViewModel 中执行异步任务，自定义控件，Manager 相关类，自定义 Adapter 中执行异步任务应该如何防止内存泄露?
- 在 BaseActivity 中实现代码:

```
private CompositeDisposable mCompositeDisposable = new CompositeDisposable();
public void addDisposable(Disposable disposable) {
    mCompositeDisposable.add(disposable);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    mCompositeDisposable.dispose();
}
```

- 每次 Rxjava 异步任务时，把相关的 disposable 对象传入，onDestroy 中统一解绑

- 在 Activity 的 onDestory() 生命周期时，自动解除订阅，以防止因生命周期组件的生命周期而导致的 RxJava 内存泄漏事件。
- 其他的解决方式
  - Rxlife 解决内存泄露
  - autoDisposable 解决内存泄露
  - RxLifecycle 解决内存泄露

19.3 （1）添加依赖

19.4 （2）基本使用

- Activity 需继承自 RxAppCompatActivity 或 RxFragmentActivity 或 RxActivity
- Fragment 需继承 RxFragment

```
retrofit.create(ApiUrl.class)
    .getRetrofit1()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    // 绑定生命周期
.compose(bindUntilEvent(ActivityEvent.DESTROY)) // <<<<<<<<<<<<<<<<<<<<
.subscribe(new Observer<Bean>() {
        @Override
            public void onSubscribe(Disposable d) {
            Log.e(TAG, "onSubscribe: " );
        }
        @Override
            public void onNext(Bean demo) {
            Log.e(TAG, "onNext: " +demo.toString());
        }
        @Override
            public void onError(Throwable e) {
            Log.e(TAG, "Throwable: " + e.getMessage());
        }
        @Override
            public void onComplete() {
            Log.e(TAG, "onComplete: " );
        }
    });
```

20 RxLifecycle 第三方库：解决内存泄露，原理详解

```
compile 'com.trello.rxlifecycle2:rxlifecycle-components:2.2.2'
compile 'com.trello.rxlifecycle2:rxlifecycle-navi:2.2.2'
```

- 主要有下面几种使用方式：

```
1 bindToLifecycle
2 bindUntilEvent
3 LifecycleProvider
```

20.1 1.bindToLifecycle

- 这种方式可以自动根据 Activity 或者 Fragment 的生命周期进行解绑，用起来也很方便,Avtivity 需要继承 RxActivity，Fragment 则需要继承 RxFragment

```
public class MainActivity extends BaseActivity{
    @Override
        protected void onStart() {
        super.onStart();
        Observable.interval(1, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .compose(this.<Long>bindToLifecycle())
            .observeOn(AndroidSchedulers.mainThread())
```

```
            .subscribe(new Consumer<Long>() {
                    @Override
                        public void accept(Long num) throws Exception {
                        Timber.tag(TAG).d("onStart, running num : " + num);
                    }
                });
    }
}
```

- 这里在 onStart 中进行绑定，如果 Activity 进入 onStop 生命周期的时候就会停止 Observable，看一下日志：
- bindToLifecycle 就是会自动绑定生命周期，我们看下 Activity 生命周期，很明显在 onCreate 中 bindToLifecycle 就会在 onDestroy 中进行解绑，其他的一一对应就是。

20.2 ActivityEvent FragmentEvent

```
/**
 * Lifecycle events that can be emitted by Activities.
 */
public enum ActivityEvent {
    CREATE,
    START,
    RESUME,
    PAUSE,
    STOP,
    DESTROY
}
/**
 * Lifecycle events that can be emitted by Fragments.
 */
public enum FragmentEvent {
    ATTACH,
    CREATE,
    CREATE_VIEW,
    START,
    RESUME,
    PAUSE,
    STOP,
    DESTROY_VIEW,
    DESTROY,
    DETACH
}
```

20.3 2.bindUntilEvent

- 见名知意，就是可以和制定的生命周期进行绑定，这种方式比上面的灵活，比如可以在一个按钮中绑定 onStart 的事件，而不必要一定要卸载 onStart 中。
- 我在上面的 Activity 中添加一个按钮，点击事件在 getData(View view) 中，看下代码：

```
public void getData(View view) {
    Observable observable = Observable.interval(1, TimeUnit.SECONDS).
        subscribeOn(Schedulers.io())
        .compose(this.bindUntilEvent(ActivityEvent.PAUSE));
    observable.observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Long>() {
                @Override
                    public void accept(Long num) throws Exception {
                    Timber.tag(TAG).d("getData, running until num : " + num);
                }
            });
}
```

- 按我们的预期，就是在 Activity 进入 onPause 时，Observable 会停止发送数据，看下打印的日志是不是这样的：bindUntilEvent.PNG
- 基本上面这两种方式就够用了，下面的方式 LifecycleProvider 在 MVP 的模式中用处就比较大了。我们接着往下看。

20.4 3.LifecycleProvider

- 使用方式就是，首先继承 NaviActivity，然后在 Activity 中加上这句话
- LifecycleProvider<ActivityEvent> provider = NaviLifecycle.createActivityLifecycleProvider(this)
- 这样就可以通过 provider 监听生命周期了。我这里在初始化 presenter 的时候传递过去

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //初始化 Presenter
    presenter = new Presenter(provider);
}
```

- 在 Activity 中添加一个按钮，通过延时 3 秒模拟网络请求，请求成功后更新 UI。这里通过 provider 发送生命周期事件，然后在 onNext 中判断事件类型，如果已经 Activity 已经进入 onPause onStop onDestroy 中的其中一个，就不再更新 UI 了，并且通过 Disposable 断开连接。

```
FactoryModel.getModel(Token.STRING_MODEL).params(params)
    .execute(new CallBack<String>() {
        @Override
            public void onSuccess(final String data) {
            provider.lifecycle().subscribe(new Observer<ActivityEvent>() {
                    @Override
                        public void onSubscribe(Disposable d) {
                        disposable = d;
                    }
                    @Override
                        public void onNext(ActivityEvent activityEvent) {
                        Timber.tag("Presenter_TAG").i("received activityEvent, activityEvent = %s" , activityEvent.name());
                        if (null != disposable && disposable.isDisposed()){
                            Timber.tag("Presenter_TAG").i("disposable isDisposed");
                            return;
                        }
                        if (activityEvent == ActivityEvent.PAUSE || activityEvent == ActivityEvent.STOP
                            || activityEvent == ActivityEvent.DESTROY) {
                            Timber.tag("Presenter_TAG").e("do not refresh UI, activityEvent = %s", activityEvent.name());
                            onComplete();
                            return;
                        }
                        if (isViewAttached()) {
                            Timber.tag("Presenter_TAG").i("refresh UI, activityEvent = %s" , activityEvent.name());
                            view.showData(data);
                        }
                    }
                    @Override
                        public void onError(Throwable e) {
                    }
                    @Override
                        public void onComplete() {
                        if (null != disposable && !disposable.isDisposed()){
                            disposable.dispose();
                            Timber.tag("Presenter_TAG").d("LifecycleProvider disposed");
                        }
                    }
                });
        }
    });
```

- 看一下正常请求成功的日志，和预期一样，正常更新 UI 了。请求成功.PNG
- 下面我们来搞一下破坏，在刚发起网络请求后，按 home 按键将 APP 切到后台，这样 Activity 就进入 onStop 的生命周期，那么即使网络请求成功也应该再更新 UI 了，看下日志是不是这样的：破坏.PNG 最终 UI 也是没有更新的。我们在下面的逻辑中切断了订阅关系，那么下次再次点击按钮发起网络请求是还能正常使用 provider 吗？

```
if (activityEvent == ActivityEvent.PAUSE
    || activityEvent == ActivityEvent.STOP
    || activityEvent == ActivityEvent.DESTROY) {
    Timber.tag("Presenter_TAG").e("do not refresh UI, activityEvent = %s", activityEvent.name());
    onComplete();
    return;
}
```

- 其实是可以的，看一下 Demo 的日志：切断订阅关系后再次监听生命周期.PNG
- 那么是怎么回事？其实就在下面这个代码中：

```
provider.lifecycle()
```

- 再跟进去 BehaviorSubject, 其中以后一句注释 Hides the identity of this Observable and its Disposable，最后会返回一个新的 Observable：

```
/**
 * Hides the identity of this Observable and its Disposable.
 * <p>Allows hiding extra features such as {@link io.reactivex.subjects.Subject}'s
 * {@link Observer} methods or preventing certain identity-based
 * optimizations (fusion).
 * <dl>
 *  <dt><b>Scheduler:</b></dt>
 *  <dd>{@code hide} does not operate by default on a particular {@link Scheduler}.</dd>
 * </dl>
 * @return the new Observable instance
 *
 * @since 2.0
 */
@CheckReturnValue
@SchedulerSupport(SchedulerSupport.NONE)
public final Observable<T> hide() {
    return RxJavaPlugins.onAssembly(new ObservableHide<T>(this));
}
```

20.5 4. 总结

- RxLifecycle 的侵入性还是比较低的，基本不需要改动原来的代码就可以实现生命周期的监听，也提供了防止 RxJava 订阅关系内存泄漏的另外一种解决方案，还是很不错的。
- 今天的分享就到这了，后面有机会和大家分享下 RxLifecycle 的源码。
- 平时工作也比较忙，写博客真的是需要耐力，如果对大家有帮助欢迎关注和点赞哈。