

# deepwaterooo deepwateroooMe – I am the same GitHub account person

deepwaterooo

September 8, 2022

## Contents

1 要求	1
2 主要思路	2
3 room 数据库相关的部分: 几个相关可以用作参考的例子	5
3.1 另一个更好的参考例子: dagger + RecyclerView 和相应的 Adapters + BufferKnife View auto-injections	5
3.2 ApiModule.java	6
3.3 ApiComponent.java	6
3.4 NocacheActivity extends Activity	6
3.5 这里有个小例子: <a href="https://github.com/Tom1881/Jet-pack/tree/master/app">https://github.com/Tom1881/Jet-pack/tree/master/app</a>	7
4 Glide 的缓存分为两种, Resource 缓存、Bitmap 缓存。	7
4.1 一、Resource 缓存:	7
4.2 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。	8
5 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	8
6 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	9
7 用 Retrofit+Rxjava 上传图片支持多张图片的上传	11

## 1 要求

- Overview

- 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.
- 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.
- 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.
- 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**).
- **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.
- 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.
- 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.
- 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

- 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](https://blog.csdn.net/qq_38436214/category_11482619.html?spm=1001.2014.3001.5482)
  - 里面有十几篇博文都作了最新的更新，上次自己那个项目，可能就只工作笔记本上还有一个版本吧，其它的两台电脑我都找过了没有了
  - 今天刚到家，只休息了三个小时，脑袋昏昏的，会明天后天再花两三天的时间在这个项目上，把它实现完，理解得再深入透彻一点儿
  - 参考网络上的例子，大部分都能懂，但是因为头昏昏的，实现的时候，现在都不知道是在干什么；今天好好休息一晚，到明天应该就能够好很多了
- 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
- **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
  - 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)
- **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\\_whitelldepth\\_1-utm\\_source=distribute.pc\\_relevant\\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-125687902-blog-51550400.pc\\_relevant\\_multi\\_platform\\_whitellutm\\_relevant\\_index=1](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_whitelldepth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-125687902-blog-51550400.pc_relevant_multi_platform_whitellutm_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

```
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.setCacheMode(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 response 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)
- 应用的 启动优化：重中之重，需要借助这个小应用弄清楚，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 room 数据库相关的部分：几个相关可以用作参考的例子

#### 3.1 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

#### 3.2 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);
}
}

```

### 3.3 ApiComponent.java

```

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

```

### 3.4 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);
    }
}

```



### 3.5 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。

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

### 4.1 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。

### 4.2 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。



## 5 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.getExternalStorageState().equals(Environment.MEDIA_MOUNTED));
```

## 6 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](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 observable = 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);
    observable.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> bitmapObserver = 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(bitmapObserver);
}

```

## 7 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())
            .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"),file));
            }
        }
        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"),file));
            }
        }
        myApiService.Image(murl,headermap,builder.build())
    }
}

```

```

        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(observer);
    return Retrofit.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);
        present.image(Contacts.UploadYourHead, headermap, map,list,Register.class);
    }
}

```

} }