红岩网校春季第八次课

Rxjava，Retrofit，Glide，EventBus使用与分析，Git多人开发

一、Rxjava使用与浅析

[官方文档]

导入依赖：

```
implementation 'io.reactivex.rxjava2:rxjava:2.2.19'
implementation 'io.reactivex.rxjava2:rxandroid:2.1.1'
```

常规使用：

```
Observable.create(new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> emitter) throws
Exception {

                emitter.onNext("abc");
                emitter.onNext("def");
                emitter.onNext("ghi");
                emitter.onComplete();
            }
        }).subscribe(new Observer<String>(){

            @Override
            public void onSubscribe(Disposable d) {

                Log.i("sandyzhang", "准备监听");
            }

            @Override
            public void onNext(String s) {

                Log.i("sandyzhang", s);
            }

            @Override
            public void onError(Throwable e) {

                Log.i("sandyzhang", "error");
            }

            @Override
            public void onComplete() {

                Log.i("sandyzhang", "监听完毕");
            }
        });
```

上述代码可以抽象成：

```
new Observable.subscribe(new Observer)
```

其实流程很简单：

1、创建一个Observable对象。

2、实现它的抽象方法subscribe。

3、调用它的subscribe抽象方法，传入的是Observer观察者。

4、这个时候你的subscribe抽象方法中写的东西，就会回调给Observer观察者（因为你调用的这个方法传入的是观察者鸭），相当于收到了信息。

创建了一个Observable对象，实现了抽象方法subscribe（其实是实现的ObservableOnSubscribe的抽象方法subscribe，Observable对象的抽象方法实际上是subscribeActual，这里为了便于讲述，进行了简化），这个时候调用它的subscribe方法时，实际上调用的就是你覆写的那个Observable的抽象方法（这不是废话嘛，其实这里是用了些设计模式，Observable.create静态方法，方便处理一些应该统一处理的东西，比如判空之类的）。

那么，你可能会说，这有啥好的，不就是普通的调用嘛，为啥要搞观察者-被观察者模式呢？我这样写不行嘛：

```
for(int i = 0; i < 3; i++) {
    onNext(i.toString());
}

void onNext(String s) {
    Log.i("sandyzhang", s);
}
```

怎么样，很像有木有。

那如果要线程切换呢？

```
void onNextOnMainThread(String s) {
    activity.runOnUiThread(new Runnable(){
        @Override
        void run() {
            Log.i("sandyzhang", s);
        }
    }
}
```

这样又不得不修改原来的代码：

```
for(int i = 0; i < 3; i++) {
    onNextOnMainThread(i.toString());
}
```

好难受有木有？。。

这就体现出rxjava的好处了，虽然简单的例子体现不出它的优势，但是代码量多的时候能依然保持高度简洁（上游-下游）。

相比于普通代码，rxjava库可以轻松切换线程（还记得runOnUiThread和handler.post的恐惧吗？），并且 上游-下游 逻辑是分离的（比如图片加载，上游只关心如何产生图片bitmap，下游只关心如何显示图片，一方逻辑变化不会影响另一方），再者，数量繁多的"操作符"，可以实现对数据的加工（比如上游来的图片bitmap，可以经过一些加工变成drawable，这样观察者只需要关心如何使用上游传来的drawable了）。

虽说是 上游-下游 这种模式，但实际上还是主动调用的。我当初刚学习rxjava的时候，没有看源码，惯性地以为

```
emitter.onNext("abc");
emitter.onNext("def");
emitter.onNext("ghi");
```

是将这些字符串，推到一个队列里，然后在有观察者订阅时，从队列里取出。导致在使用时有意料之外的情况发生，不符合直觉。现在想想真是可笑哈哈哈。

实际上subscribe方法是核心，是 上游-下游 这种"流事件" 的动力，根本没有什么队列，全程是通过回调来实现的，不得不佩服想到这种方法的rxjava创始人！脑洞真的太大了。

那运算符到底是啥呢?

其实就是"上游事件到下游之前的 转换"，将一个事件加工成另外的事件，或者线程切换，或者事件合并或分解。

```
Observable.just(1, 2, 3)
                .map(new Function<Integer, String>() {
                    @Override
                    public String apply(@NonNull Integer integer) throws
Exception {
                        return String.valueOf(integer);
                    }
                })
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        Log.i("sandyzhang", s);
                    }
                });
```

这个例子，上游传来的整数经过map操作符，转换为了字符串，传给下游的观察者。

Consumer是简易版的Observer，他有多重重载，可以自定义你需要处理的信息，我这里调用的是只接受onNext消息的方法，他只提供一个回调接口accept，由于没有onError和onCompete，无法再 接受到onError或者onCompete之后，实现函数回调。无法回调，并不代表不接收，他还是会接收到onCompete和onError之后做出默认操作，也就是监听者（Consumer）不在接收Observable发送的消息（一般观察者Observable在收到onCompete和onError之后，不会再收到onNext）。

仔细观察其实还有Function3，4，5...

BiFunction其实就是Function2；

实际上就是一个简单的接口，让你自己定义如何将多个类型的数据合并为一个数据。

有一个zip，将两个类型的数据进行了合并：

```
Observable.zip(Observable.just(1, 2, 3),
                Observable.just("A", "B", "C", "D", "E"),
                new BiFunction<Integer, String, String>(){

                    @Override
                    public String apply(Integer o1, String o2) throws Exception
{

                        return o1 +"_"+ o2;
                    }
                })
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String o) throws Exception {

                        Log.i("sandyzhang", o);
                    }
                });
```

输出为：1_A,2_B,3_C，由于DEF没有配对，于是不输出。

而内部是如何实现数据的“加工”呢？实际上是在内部创建了一个新的可观察者，然后调用它的subscribe方法，有点像代理模式对吧？你以为你在调用它的方法，实际上你在调用外壳的方法，外壳在内部决定自己如何调用里面的方法。

“加工”就不断地创建新的可观察者，用代理模式建立上游与下游的联系，这样有点像一条链子，一个一个可观察者的subscribe方法不断调用上一个可观察者的subscribe方法，这样，你写代码是对下游的Observable调用subscribe方法，实际上就链式地调用到了源Observable的subscribe方法（这个方法的实现是你自己写的）。

再说说线程切换：

```
Observable.just(1, 2, 3)
                .map(new Function<Integer, String>() {
                    @Override
                    public String apply(@NonNull Integer integer) throws
Exception {
                        return String.valueOf(integer);
                    }
                })
                .subscribeOn(Schedulers.io()) // Observable运行在工作线程
                .observeOn(AndroidSchedulers.mainThread()) // Observer运行在主线程
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        Log.i("sandyzhang", s);
                    }
                });
```

细节还是挺多的，比如多个subscribeOn，哪一个有效，还有observeOn可以多次切换吗？情况比较多，可以自己总结加深印象。

> 引用：
>
> **AndroidSchedulers.mainThread()**

> 顾名思义，在Android的主线程执行，一般更新ui都用这个。
>
> ### Schedulers.io()
>
> 这个调度器用于I/O操作，比如：读写文件，数据库，网络交互等等。行为模式和`newThread()`差不多，重点需要注意的是线程池是无限制的，大量的I/O调度操作将创建许多个线程并占用内存。
>
> ### Schedulers.computation()
>
> 计算工作默认的调度器，这个计算指的是 CPU 密集型计算，即不会被 I/O 等操作限制性能的操作，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。
>
> ### Schedulers.immediate()
>
> 这个调度器允许你立即在当前线程执行你指定的工作。这是默认的`Scheduler`。
>
> ### Schedulers.newThread()
>
> 这个调度器正如它所看起来的那样：它为指定任务启动一个新的线程。
>
> ### Schedulers.trampoline()
>
> 当我们想在当前线程执行一个任务时，并不是立即，我们可以用`trampoline()`将它入队。这个调度器将会处理它的队列并且按序运行队列中每一个任务。

如果用的是kotlin，还可以封装一些函数：

```
fun <T> Observable<T>.setSchedulers(
        subscribeOn: Scheduler = Schedulers.io(),
        unsubscribeOn: Scheduler = Schedulers.io(),
        observeOn: Scheduler = AndroidSchedulers.mainThread() // 设置默认在主线程回调
): Observable<T> = subscribeOn(subscribeOn)
        .unsubscribeOn(unsubscribeOn)
        .observeOn(observeOn)

// 使用：
new Observable.setSchedulers().subscribe(new Observer) // 省略了线程切换之类的逻辑
```

相信有了上文的解析，再看运算符，会发现很简单。

关于rxjava的众多运算符，在这里不赘述，有一篇总结值得推荐：[史上最全的Rxjava2讲解（使用篇）(juejin.cn)]

二、Retrofit使用与浅析

[Retrofit官方文档（基于Okhttp）]

[Okhttp官方文档]

导入依赖：

```
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
implementation 'com.squareup.okhttp3:okhttp:4.4.1'
implementation 'com.squareup.okio:okio:2.4.3'
implementation 'com.squareup.retrofit2:converter-gson:2.7.2'
implementation 'com.squareup.retrofit2:adapter-rxjava2:2.7.2'
implementation 'com.squareup.okhttp3:logging-interceptor:4.4.1'
```

常规使用：

先要定义一个接口，“描述每个网络请求的链接、请求方式、参数、请求头、返回的类型”，rxjava会根据接口的这些描述，创建代理对象。

```
public interface ApiService { // 类名随便起
    @GET("users/list") // 跟主链接拼接后，成为完整的链接。
    Call<User> getUsers(@Query String id); // 要明确返回类型，注解表明“是get的参数”

    @FormUrlEncoded // 使用@Field注解时必须要这句
    @POST("users/list")
    Observable<User> getUsersO(@Field("id") String id); // 可以与rxjava连用
}
```

再创建retrofit实例

```
Retrofit retrofit = new Retrofit.Builder()
                // 设置主url，之后接口只需要填剩下的url就可以了
                .baseUrl("https://abc/")
                // 添加反序列化工厂，Gson，可以自定义Converter.Factory
                .addConverterFactory(GsonConverterFactory.create())
                // 添加Rxjava工厂，允许创建Observable对象
                .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                .build();
```

注意，retrofit.create方法传入一个接口，retrofit会根据接口中的方法参数、注解、返回值，生成对应的代理类（典型的代理模式），并且返回这个代理对象。
这个时候就可以使用接口中的方法了，相当于实例化了接口，有木有很神奇？

使用方法1：同步方法（注意，此方法会使线程等待，直到得到网络请求结果）

```
        Call<User> res = retrofit.create(ApiService.class).getUsers();
        User u = res.execute().body(); // 直接根据网络请求返回的json，自动转化为对象
```

使用方法2：异步方法（接口回调）

```
        Call<User> res = retrofit.create(ApiService.class).getUsers();
        User u = res.enqueue(new Callback<User>() {
            @Override
            public void onResponse(Call<User> call, Response<User> response) {
                User u = response.body();
                // do something...
                // 这里已经是主线程啦！
            }

            @Override
            public void onFailure(Call<User> call, Throwable t) {
                // do something...
```

```
            }
        });
```

看得这个回调有木有很像自己封装的网络请求的方法？但是retrofit这里已经做了线程切换，已经是主线程了~非常方便。

但是还有更方便的，retrofit最强大的地方是可以跟rxjava无缝衔接，还记得.addCallAdapterFactory(RxJava2CallAdapterFactory.create())这句吗，实际上就是典型的工厂模式，根据Api接口的返回类型和网络请求返回的数据，自动转换，创建出Observable。

请看用法：

```
retrofit.create(ApiService.class).getUsersO("abc").subscribeOn(Schedulers.io()).
observeOn(AndroidSchedulers.mainThread()).subscribe(new Observer<User>() {
            @Override
            public void onNext(@NonNull User user) {
                // 在这里就可以更新ui啦
            }

            @Override
            public void onError(@NonNull Throwable e) {
                // 网络请求失败
            }

            @Override
            public void onComplete() {
            }
        })
```

如果你学了okhttp，还可以在这里设置默认的OkHttpClient（因为retrofit内部是用okhttp实现的，所有请求都是通过client创建的）

```
Retrofit retrofit = new Retrofit.Builder()
        // 设置主url，之后接口只需要填剩下的url就可以了
        .baseUrl("https://abc/")
        // 添加反序列化工厂，Gson，可以自定义Converter.Factory
        .addConverterFactory(GsonConverterFactory.create())
        // 添加Rxjava工厂，允许创建Observable对象
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
        .client(new OkHttpClient().newBuilder()
                // 设置连接超时
                .connectTimeout(5, TimeUnit.SECONDS)
                // 设置拦截器
                .addInterceptor(myInterceptor)
                // 设置自动打log拦截器
                .addInterceptor(new HttpLoggingInterceptor())
                .build())
        .build();
```

高级用法之：拦截器（其实这里算okhttp的知识了，但感觉有必要提一下）

文档是这么说的：

> Observes, modifies, and potentially short-circuits requests going out and the corresponding responses coming back in. Typically interceptors add, remove, or transform headers on the requestor response.

大意就是：拦截器被用来对请求头进行添加、转换、删除，并且在需要时，使得请求短路。

```
new HttpLoggingInterceptor()
```

设置拦截器，在请求和响应时，均会经过这个拦截器，打出对应的log

这个拦截器其实相当于一个官方写的例子了hhh，因为它包含读取请求头、请求体、响应头、响应体等各种操作，然后生成字符串并且打成Log，所以自己写自定义myInterceptor时可以翻翻HttpLoggingInterceptor()得到些启示（国内关于Interceptor的教程实在少得可怜）。

用拦截器自动重试：

```
    addInterceptor(
        object : Interceptor {
                    override fun intercept(chain: Interceptor.Chain): Response {
                        var retryCount = 0
                        var request =
                            // 这里可以为“每个请求”统一添加header，实际上每一个请求都会走这个拦截器，所以就一并添加上了（可以实现token的自动化管理，过期时自动刷新，特别方便）
                            chain.request().newBuilder().header("token", "你的登录token").build()
                        // 尝试网络连接
                        var response = retry(request, chain)
                        // 如果失败则重试3次
                        while (response?.isSuccessful != true && retryCount < retryNum) {
                            retryCount++
                            response?.close()
                            response = retry(request, chain)
                        }
                        return response
                    }
                }
    ) // end addInterceptor


    // 定义一个不会报错的请求函数
    fun retry(request: Request, chain: Interceptor.Chain): Response? {
        // 注意必须要try-catch，否则有可能不会重试而直接抛异常（因为proceed方法会抛出IOException）
        return try {
            chain.proceed(request)
        } catch (e: Exception) {
            null
        }
    }
```

那么，细心的你肯定注意到，拦截器是add进去的（addInterceptor），执行顺序是什么呢?

为什么直接return也不会影响后续interceptor的执行呢？

这里会有一个惯性思维：以为是你自定义的intercept执行完了，才会调用下一个interceptor的intercept。实际上在你调用“chain.proceed(request)”时，就会请求下一个interceptor的intercept继续对请求进行修饰和处理。

有关拦截器，这里推荐一篇文章：[OkHttp Interceptor 入门到进阶 - 简书 (jianshu.com)]

Glide使用

官方文档

添加依赖：

```
implementation 'com.github.bumptech.glide:glide:4.11.0'
```

基础使用：

```
Glide.with(context) // 此处建议传Activity或者Fragment，会和他们生命周期保持一致，例如：onPaused 时暂停加载，onResume 时又会自动重新加载
    .load(url)
    .into(imageView);
```

使用占位图：

```
Glide.with(context)
    .load(url)
    .placeholder(R.drawable.place_image) // 图片加载出来前，显示的图片
    .error(R.drawable.error_image) // 图片加载失败后，显示的图片
    .into(imageView);
```

使用缩略图：

```
Glide.with(context)
    .load(url)
    .thumbnail(0.2f)
    .into(imageView);
```

其实就是加载一个原图0.2倍的图片，通过采样率。但是网络请求一般是把图片流下载下来后，才设置采样率转换为bitmap，所以这个方法对于大图比较有效。

那么占位图可以是网络上的吗：

```
DrawableRequestBuilder<String> thumbnailRequest =
Glide.with(context).load(url2);
Glide.with(context)
    .load(url1)
    .thumbnail(thumbnailRequest) // 相当于占位图通过url2获得。如果有一个预览图的链接（后端同时给了个低质量的预览图片，则可以设置为url2）
    .into(imageView);
```

裁剪图片：

```
Glide.with(context)
    .load(url)
    .override(width, height)
    .into(imageView);
```

有关其他高级用法，可以参考文档，在此不赘述。

EventBus使用

官方文档

添加依赖：

```
implementation 'org.greenrobot:eventbus:3.1.1'
```

简单使用：

在"要接收消息通知"的activity中：

```
// 在activity的onCreate中，将自己设为订阅者
EventBus.getDefault().register(this)
// 在activity的onDestroy中，取消注册
EventBus.getDefault().unRegister(this)
```

定义一个（或多个）消息类：

```
class Msg(val s: String)
```

发送消息：

```
EventBus.getDefault().post(Msg("666"))
```

这样，任何：

1、注册了的订阅者。

2、中的方法，加了注解的。

3、并且方法的参数类型是你发送的类型。

都会收到事件，事件顺序取决于优先级。

如：

```
@Subscribe(threadMode = ThreadMode.MAIN)
fun log(msg: Msg) {
    Toast.makeText(this, msg.s, Toast.LENGTH_SHORT).show()
}
```

这个方法就能收到事件，threadMode可以设置回调的线程；

> POSTING (默认) 表示事件处理函数的线程跟发布事件的线程在同一个线程。
>
> MAIN 表示事件处理函数的线程在主线程(UI)线程，因此在这里不能进行耗时操作。
>
> BACKGROUND 表示事件处理函数的线程在后台线程，因此不能进行UI操作。如果发布事件的线程是主线程(UI线程)，那么事件处理函数将会开启一个后台线程，如果果发布事件的线程是在后台线程，那么事件处理函数就使用该线程。
>
> ASYNC 表示无论事件发布的线程是哪一个，事件处理函数始终会新建一个子线程运行，同样不能进行UI操作。

黏性事件：

```
@Subscribe(threadMode = ThreadMode.MAIN, sticky = true) // sticky = true
fun log(msg: Msg) {
    Toast.makeText(this, msg.s, Toast.LENGTH_SHORT).show()
}
```

设想一个场景，当你想打开第二个activity，并且立刻执行里面的一个方法，我们会这样：

```
val intent = Intent(this, NextActivity::class.java)
EventBus.getDefault().post(Msg("666"))
```

我们都知道，intent是异步的，不确定什么时候打开第二个activity，这个时候，如果直接post，可能在第二个activity还没有register时就已经把事件分发出去了，这样它就收不到事件了。所以有了黏性事件。

> 所谓粘性事件，就是在发送事件之后再订阅该事件也能收到该事件（最新一个事件）

其实用过MutableLiveData的同学都应该发现了，在调用observe方法时，会调用一次onChange，即变化为初值，这里也非常像。

那么，如何删除黏性事件呢?

```
val msg = EventBus.getDefault().getStickyEvent(Msg::class.java) // 删除有关Msg的黏性事件。每一个事件类，都只有一个“最近状态”
if(msg != null) {
    EventBus.getDefault().removeStickyEvent(msg)
}
```

Git多人开发

口述