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

**目 录**

[1. 依赖注入框架Dagger2 4](#_Toc3826)

[1.1. 关键字 4](#_Toc17962)

[1.1.1. 依赖注入 4](#_Toc21450)

[1.1.2. 注解 4](#_Toc31297)

[1.1.3. Inject 4](#_Toc7497)

[1.1.4. Component 4](#_Toc9114)

[1.1.5. Module 5](#_Toc27226)

[1.1.6. Qualifier 5](#_Toc20781)

[1.1.7. Scope 6](#_Toc21227)

[1.1.8. 基础总结 7](#_Toc13501)

[1.2. Component划分 7](#_Toc32019)

[1.2.1. 根据什么来划分Component——页面 7](#_Toc12765)

[1.2.2. 为什么以页面为粒度来划分Component 7](#_Toc9355)

[1.2.3. 组织Component 7](#_Toc10764)

[1.3. dagger2的好处 8](#_Toc3802)

[1.4. 如何引入Dagger2 8](#_Toc5809)

[1.5. Dagger2的注入流程 8](#_Toc23003)

[1.5.1. 流程 8](#_Toc18433)

[1.5.2. 简单代码 9](#_Toc16269)

[1.5.3. 解析 10](#_Toc6171)

[1.5.4. 解析二 11](#_Toc335)

[1.6. Dagger2注入原理 12](#_Toc7119)

[1.6.1. 步奏一:MainPresenter 12](#_Toc32234)

[1.6.2. 步奏二:MainModule 12](#_Toc26360)

[1.6.3. 步奏三:Component 13](#_Toc20771)

[1.6.4. 解释关键的这三步和注入代码 15](#_Toc26704)

[1.7. 来个代码 16](#_Toc4268)

[1.7.1. AppModule 16](#_Toc16672)

[1.7.2. HttpModule 16](#_Toc6559)

[1.7.3. ActivityModule 17](#_Toc24383)

[1.7.4. AppComponent 17](#_Toc15094)

[1.7.5. ActivityComponent 17](#_Toc5179)

[1.7.6. App 18](#_Toc11418)

[1.7.7. BasePresenter 18](#_Toc31399)

[1.7.8. RxPresenter 18](#_Toc32757)

[1.7.9. MainPresenter 18](#_Toc17111)

[1.7.10. BaseActivity 19](#_Toc24125)

[1.7.11. MainActivity 19](#_Toc2715)

[2. 依赖、Dagger2讲解资料 20](#_Toc18882)

[2.1. 依赖注入概念 20](#_Toc16312)

[2.2. DI的常见实现方式 20](#_Toc777)

[2.3. Dagger2——DI框架的由来 21](#_Toc19240)

[2.3.1. 不使用DI框架的问题 21](#_Toc30597)

[2.3.2. 解药：Dagger2 22](#_Toc29551)

[2.4. Dagger2各部分作用 22](#_Toc3867)

[2.4.1. Module——生产Dependency的工厂 22](#_Toc5768)

[2.4.2. Component——Dependency工厂管理员 25](#_Toc5856)

[2.4.3. Component给Client提供Dependency的方法 26](#_Toc8239)

[2.4.3.1. 方法一：在Component里面定义一个返回Dependency的方法 26](#_Toc15023)

[2.4.3.1.1. 好处在哪里 27](#_Toc1944)

[2.4.3.2. 方法二：Field Injection 27](#_Toc19595)

[2.4.3.3. 总的来说注入方法： 29](#_Toc14299)

[2.5. Component之间互相有依赖的用法 29](#_Toc20452)

[3. OkHttp的使用 30](#_Toc8333)

[3.1. 基本用法 30](#_Toc12347)

[3.1.1. Http Get 30](#_Toc22627)

[3.1.2. Http Post 携带参数 31](#_Toc1471)

[3.1.3. 基于http的文件上传 31](#_Toc18914)

[3.1.4. 封装（主要是思路） 32](#_Toc8137)

[3.2. 添加统一请求头和参数 34](#_Toc25169)

[3.2.1. 添加统一请求头 34](#_Toc3185)

[3.2.2. 添加统一参数 35](#_Toc15764)

[3.3. 拦截器Interceptor(可用于Retrofit2) 36](#_Toc19856)

[3.3.1. 拦截器的定义： 36](#_Toc23967)

[3.3.2. 记录请求和响应信息的拦截器： 36](#_Toc11997)

[3.3.3. 两类拦截器（Interceptors） 37](#_Toc12946)

[3.3.3.1. 应用拦截器（ApplicationInterceptors） 37](#_Toc3607)

[3.3.3.2. 网络拦截器（NetworkInterceptors） 38](#_Toc8396)

[3.3.3.3. 拦截器选择 39](#_Toc11605)

[3.3.4. 一些拦截器工具类 39](#_Toc23663)

[3.3.4.1. 缓存时间拦截器 39](#_Toc10480)

[3.3.4.2. Logger拦截器 40](#_Toc28207)

[3.3.4.2.1. Logger打印到硬盘中 41](#_Toc13732)

[3.3.4.3. 统一请求头(参数)拦截器 41](#_Toc4955)

[3.4. Client(可用于Retrofit2) 42](#_Toc12511)

[3.4.1. 错误重连 42](#_Toc31626)

[3.4.2. 添加超时 42](#_Toc7250)

[3.4.3. 添加缓存大小 42](#_Toc14417)

[3.4.4. 添加拦截器 42](#_Toc10047)

[4. Retrofit2 42](#_Toc6114)

[4.1. Retrofit入门 42](#_Toc15139)

[4.1.1. 接口列表 43](#_Toc10642)

[4.1.2. 创建Retrofit实例 43](#_Toc28322)

[4.1.3. 接口定义 43](#_Toc25486)

[4.1.4. 接口调用 43](#_Toc6369)

[4.2. Retrofit注解详解 44](#_Toc23377)

[4.2.1. 第一类：HTTP请求方法 44](#_Toc20664)

[4.2.2. 第二类：标记类 45](#_Toc20111)

[4.2.3. 第三类：参数类 45](#_Toc15963)

[4.3. Client（OkHttpClient） 46](#_Toc26800)

[4.4. Converter之GSON 46](#_Toc7026)

[4.5. CallAdapter之RxJava 47](#_Toc20767)

[4.6. 自定义Converter 48](#_Toc23726)

[4.7. 自定义CallAdapter 49](#_Toc12523)

[4.8. 示例源码整合 51](#_Toc6502)

[4.8.1. Example01 51](#_Toc4748)

[4.8.2. Example02 52](#_Toc10435)

[4.8.3. Example03 52](#_Toc21715)

[4.8.4. Example04 54](#_Toc26998)

[4.8.5. Example05 55](#_Toc29255)

[4.8.6. Example06 55](#_Toc4121)

[4.8.7. Example07 56](#_Toc10367)

[4.8.8. Example08 57](#_Toc31148)

[4.8.9. Example09 58](#_Toc10040)

[4.8.10. Example10 59](#_Toc10949)

[4.9. 其它说明 61](#_Toc27966)

[4.9.1. Retrofit.Builder 61](#_Toc10601)

[4.9.2. Retrofit的Url组合规则 61](#_Toc26965)

[4.9.3. Retrofit提供的Converter 61](#_Toc3767)

[4.9.4. Retrofit提供的CallAdapter 62](#_Toc26575)

[5. RxJava 62](#_Toc22207)

[5.1. 引入 62](#_Toc22899)

[5.2. RxJava 到底是什么 62](#_Toc26686)

[5.3. RxJava 好在哪 62](#_Toc9941)

[5.4. API 介绍和原理简析 64](#_Toc23323)

[5.4.1. 概念：扩展的观察者模式 64](#_Toc7004)

[5.4.2. 基本实现 65](#_Toc31695)

[5.4.2.1. 创建 Observer 65](#_Toc3461)

[5.4.2.2. 创建 Observable 66](#_Toc15143)

[5.4.2.3. Subscribe (订阅) 66](#_Toc22049)

[5.4.2.4. subscribe() 还支持不完整定义的回调 67](#_Toc29954)

[5.4.2.5. 场景示例 68](#_Toc30370)

[5.4.2.5.1. 打印字符串数组 68](#_Toc3566)

[5.4.2.5.2. 由 id 取得图片并显示 68](#_Toc3762)

[5.4.3. 线程控制 —— Scheduler (一) 69](#_Toc9949)

[5.4.3.1. Scheduler 的 API (一) 69](#_Toc939)

[5.4.3.2. Scheduler 的原理 (一) 70](#_Toc24386)

[5.4.4. 变换 70](#_Toc4267)

[5.4.4.1. API 70](#_Toc8533)

[5.4.4.1.1. map() 70](#_Toc27731)

[5.4.4.1.2. flatMap() 71](#_Toc4442)

[5.4.4.2. 变换的原理：lift() 73](#_Toc22897)

[5.4.4.3. compose: 对 Observable 整体的变换 75](#_Toc16402)

[5.4.5. 线程控制：Scheduler (二) 76](#_Toc25137)

[5.4.5.1. Scheduler 的 API (二) 76](#_Toc448)

[5.4.5.2. Scheduler 的原理（二） 77](#_Toc14718)

[5.4.5.3. doOnSubscribe() 78](#_Toc5747)

[5.5. CompositeSubscription、addSubscrebe关于控制生命周期 79](#_Toc23737)

[5.6. Retrofit 与RxJava的结合 80](#_Toc30059)

[5.7. RxBinding 84](#_Toc8185)

[5.8. RxBus 84](#_Toc23762)

[5.8.1. RxBus工具类 84](#_Toc24939)

[5.8.2. 事件类 85](#_Toc4135)

[5.8.3. 注册接收 86](#_Toc17546)

[5.8.4. 发送事件 86](#_Toc17161)

[5.9. RxPermissions 86](#_Toc7242)

[5.9.1. RxPermissions使用的注意事项 86](#_Toc23358)

[5.9.2. RxPermissions使用 86](#_Toc9068)

[5.10. 一个简单的应用 88](#_Toc19508)

[6. RxJava2 89](#_Toc23696)

[6.1. Gradle配置 89](#_Toc8017)

[6.2. Observable、Observer、subscribe() 90](#_Toc32708)

[6.3. 线程切换 91](#_Toc6073)

[6.4. 生命周期相关 91](#_Toc18848)

[6.5. 背压BackPress 92](#_Toc5511)

[6.5.1. 水缸满了就OOM 92](#_Toc24625)

[6.5.2. 自己解决 92](#_Toc15238)

[6.5.3. 用Flowable解决 93](#_Toc25660)

[6.5.3.1. 基础用法 93](#_Toc13290)

[6.5.3.2. 大水缸BackpressureStrategy.BUFFER) 95](#_Toc14756)

[6.5.3.3. 数量解决DROP、LATEST 95](#_Toc5018)

[6.6. 配合操作符 96](#_Toc10232)

[6.7. 响应式FlowableEmitter的requested 97](#_Toc6413)

[6.7.1. 同步的情况 98](#_Toc15474)

[6.7.2. 异步的情况 100](#_Toc10951)

[6.7.3. 实践 103](#_Toc24814)

[6.8. 一些操作符 104](#_Toc23282)

[6.8.1. map 104](#_Toc30631)

[6.8.2. FlatMap 105](#_Toc8374)

[6.8.3. concatMap 106](#_Toc2484)

[6.8.4. interval和timer 106](#_Toc15660)

[6.8.5. Zip 107](#_Toc18078)

[6.8.6. Concat 109](#_Toc12587)

[6.8.7. Filter 109](#_Toc17803)

[6.8.8. Single 110](#_Toc4673)

[6.8.9. distinct 110](#_Toc15566)

[6.8.10. debounce 111](#_Toc16522)

[6.8.11. doOnSubscribe()与Rx1.x区别不大 111](#_Toc19161)

[6.8.12. doOnNext与Rx1.x区别不大 112](#_Toc19538)

[6.9. 线程调度相关 112](#_Toc999)

[6.10. Func和Action替换为了Function和Consumer 113](#_Toc4507)

[6.11. 一些常见用法 113](#_Toc20954)

[6.11.1. Map简单的网络请求 113](#_Toc254)

[6.11.2. flatMap多个网络请求依次依赖 115](#_Toc24606)

[6.11.3. 注意Map和FlatMap的区别 116](#_Toc2186)

[6.11.4. Concat先读取缓存，如果缓存没数据再通过网络请求 117](#_Toc23442)

[6.11.5. Zip结合多个接口的数据更新 UI 118](#_Toc25731)

[6.11.6. Interval间隔任务实现心跳 119](#_Toc20164)

[6.12. 常见崩溃 119](#_Toc7986)

[6.12.1. 如果上游发送了onError，下游没有处理，则会报错： 119](#_Toc25862)

# 依赖注入框架Dagger2

## 关键字

### 依赖注入

依赖注入是面向对象编程的一种设计模式，其目的是为了降低程序耦合，这个耦合就是类之间的依赖引起的。

依赖注入就是目标类（目标类需要进行依赖初始化的类，下面都会用目标类一词来指代）中所依赖的其他的类的初始化过程，不是通过手动编码的方式创建，而是通过技术手段可以把其他的类的已经初始化好的实例自动注入到目标类中。一般来说，依赖注入又叫控制反转。控制反转一般分为两种类型(依赖注入与依赖查找)，依赖注入比较常用

我们在写面向对象程序时，往往会用到组合，即在一个类中引用另一个类，从而可以调用引用的类的方法完成某些功能。这个时候就产生了依赖问题，ClassA依赖于ClassB，必须借助ClassB的方法，才能完成一些功能。这样看好像并没有什么问题，但是我们在ClassA的构造方法里面直接创建了ClassB的实例，问题就出现在这，在ClassA里直接创建ClassB实例，违背了单一职责原则，ClassB实例的创建不应由ClassA来完成；其次耦合度增加，扩展性差，如果我们想在实例化ClassB的时候传入参数，那么不得不改动ClassA的构造方法，不符合开闭原则。

因此我们需要一种注入方式，将依赖注入到宿主类（或者叫目标类）中，从而解决上面所述的问题。依赖注入有一下几种方式。

* 通过接口注入
* 通过set方法注入
* 通过构造方法注入
* 通过Java注解

在Dagger2中用的就是最后一种注入方式，通过注解的方式，将依赖注入到宿主类中。

### 注解

注解（Annotation），也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。

### Inject

用注解(Annotation)来标注目标类中所依赖的其他类，同样用注解来标注所依赖的其他类的构造函数，那注解的名字就叫Inject

class B{

@Inject

A a;

}

class A{

private String param1;

@Inject

public A(){

}

@Inject

public A(String param1){

this.param1 = param1

}

}

### Component

Component也是一个注解类，一个类要想是Component，必须用Component注解来标注该类，并且该类是接口或抽象类。我们不讨论具体类的代码，我想从抽象概念的角度来讨论Component。上文中提到Component在目标类中所依赖的其他类与其他类的构造函数之间可以起到一个桥梁的作用。

**Component的工作原理**

Component需要引用到目标类的实例，Component会查找目标类中用Inject注解标注的属性，查找到相应的属性后会接着查找该属性对应的用Inject标注的构造函数（这时候就发生联系了），剩下的工作就是初始化该属性的实例并把实例进行赋值。因此我们也可以给Component叫另外一个名字注入器（Injector）

### Module

问题：项目中使用到了第三方的类库，第三方类库又不能修改，所以根本不可能把Inject注解加入这些类中，这时我们的Inject就失效了。

那我们可以封装第三方的类库，封装的代码怎么管理呢，总不能让这些封装的代码散落在项目中的任何地方，总得有个好的管理机制，那Module就可以担当此任。可以把封装第三方类库的代码放入Module中，像下面的例子：

@Module

public class ModuleClass{

//A是第三方类库中的一个类

@Provides

A provideA(){

return new A();

}

}

### Qualifier

依赖类有多个被Inject标记的构造函数，目标类中注入该依赖类时不知道注入的时哪一个，这时就该使用限定符了。限定符是解决依赖注入迷失的问题。

class A{

private String param1;

@Inject

@NoParmas //Qualifier

public A(){

}

@Inject

@HasParams //Qualifier

public A(String param1){

this.param1 = param1

}

}

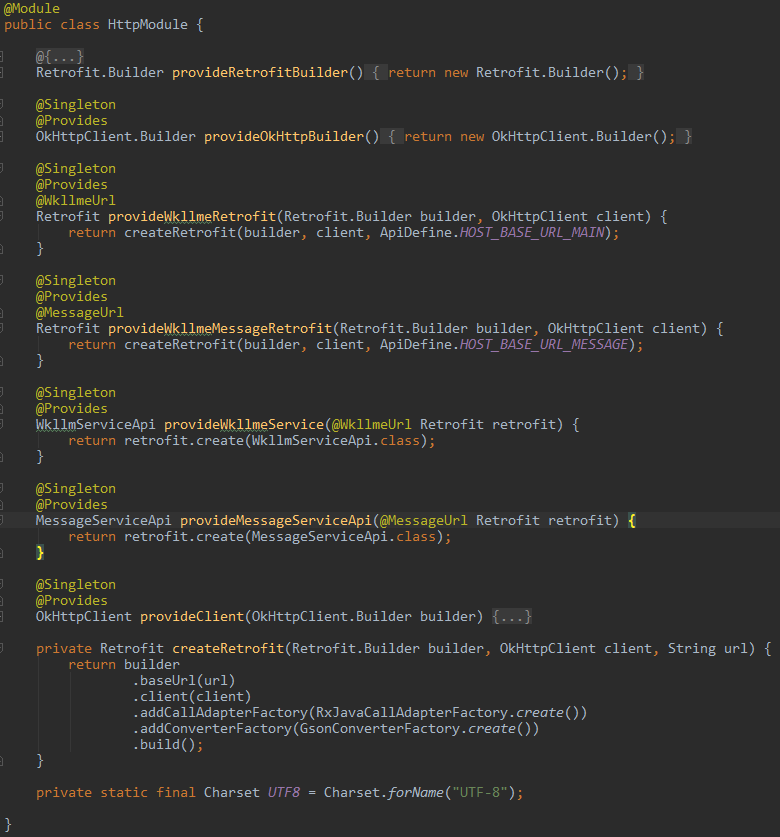
class B{

@Inject

@Noparams

A a;

}



### Scope

Scope的真正用处就在于Component的组织。更好的管理Component之间的组织方式，不管是依赖方式还是包含方式，都有必要用自定义的Scope注解标注这些Component，这些注解最好不要一样了，不一样是为了能更好的体现出Component之间的组织方式。还有编译器检查有依赖关系或包含关系的Component，若发现有Component没有用自定义Scope注解标注，则会报错。

更好的管理Component与Module之间的匹配关系，编译器会检查 Component管理的Modules，若发现标注Component的自定义Scope注解与Modules中的标注创建类实例方法的注解不一样，就会报错。可读性提高，如用Singleton标注全局类，这样让程序猿立马就能明白这类是全局单例类。

**但是Singleton没有创建单例的能力**

为什么要谈到创建单例呢？因为上面谈到一个app要有一个全局的Component（我们暂且叫ApplicationComponent），ApplicationComponent负责管理整个app用到的全局类实例，那不可否认的是这些全局类实例应该都是单例的，那我们怎么才能创建单例？

上一节提到过Module的作用，Module和Provides是为解决第三方类库而生的，Module是一个简单工厂模式,Module可以包含创建类实例的方法，现在Modlule可以创建所以类的实例。同时

Component会首先从Module维度中查找类实例，若找到就用Module维度创建类实例，并停止查找Inject维度。否则才是从Inject维度查找类实例。所以创建类实例级别Module维度要高于Inject维度。

所以我们可以这么创建单例。

* 在Module中定义创建全局类实例的方法
* ApplicationComponent管理Module
* 保证ApplicationComponent只有一个实例（在app的Application中实例化ApplicationComponent）

**Scope可以一定程度上理解为单例**

首先我们自定义一个scope注解，假如名为MyScope。假如一个component B前面加了@Myscope注解，那么对于B管理的Modules，每个module里面有许多provideXXX方法，假设某个module中有一个这样的方法：

ClassA provideClassA(){

return new ClassA();

}   
 下面就是见证奇迹的时刻，如果你为这个方法加上@Myscope注解，那么Dagger2会在第一次使用该方法创建ObjectA时将它缓存起来，下次再需要提供ObjectA时，就直接使用缓存的对象，而不会再次创建新的ObjectA,如果没有加@Myscope注解，则会每次都创建一个新的ObjectA对象。

### 基础总结

Inject，Component，Module，Provides是dagger2中的最基础最核心的知识点。奠定了dagger2的整个依赖注入框架。

* Inject主要是用来标注目标类的依赖和依赖的构造函数
* Component它是一个桥梁，一端是目标类，另一端是目标类所依赖类的实例，它也是注入器（Injector）负责把目标类所依赖类的实例注入到目标类中，同时它也管理Module。
* Module和Provides是为解决第三方类库而生的，Module是一个简单工厂模式，Module可以包含创建类实例的方法，这些方法用Provides来标注

## Component划分

### 根据什么来划分Component——页面

* 假如一个app中只有一个Component，那这个Component是很难维护、并且变化率是很高，很庞大的，就是因为Component的职责太多了导致的。所以就有必要把这个庞大的Component进行划分，划分为粒度小的Component。那划分的规则这样的：
* 要有一个全局的Component(可以叫ApplicationComponent),负责管理整个app的全局类实例（全局类实例整个app都要用到的类的实例，这些类基本都是单例的，后面会用此词代替）

每个页面对应一个Component，比如一个Activity页面定义一个Component，一个Fragment定义一个Component。当然这不是必须的，有些页面之间的依赖的类是一样的，可以公用一个Component。

### 为什么以页面为粒度来划分Component

一个app是由很多个页面组成的，从组成app的角度来看一个页面就是一个完整的最小粒度了。一个页面的实现其实是要依赖各种类的，可以理解成一个页面把各种依赖的类组织起来共同实现一个大的功能，每个页面都组织着自己的需要依赖的类，一个页面就是一堆类的组织者。

### 组织Component

我们已经把一个app按照上面的页面规则划分为不同的Component了，全局类实例也创建了单例模式。问题来了其他的Component想要把全局的类实例注入到目标类中该怎么办呢？这就涉及到类实例共享的问题了，因为Component有管理创建类实例的能力。因此只要能很好的组织Component之间的关系，问题就好办了。具体的组织方式分为以下3种：

**依赖方式**

一个Component是依赖于一个或多个Component，Component中的dependencies属性就是依赖方式的具体实现，例如ActivityComponent依赖Appcomponent，代码如下

@ActivityScope

@Component(

dependencies = AppComponent.class,

modules = {

ActivityModule.class

})

public interface ActivityComponent {

Activity getActivity();

//注入activity

void inject(MainActivity mainActivity);

...

}

**包含方式**

一个Component是包含一个或多个Component的，被包含的Component还可以继续包含其他的Component。这种方式特别像Activity与Fragment的关系。SubComponent就是包含方式的具体实现。

**继承方式**

官网没有提到该方式，具体没有提到的原因我觉得应该是，该方式不是解决类实例共享的问题，而是从更好的管理、维护Component的角度，把一些Component共有的方法抽象到一个父类中，然后子Component继承。

## dagger2的好处

* 增加开发效率、省去重复的简单体力劳动，最明显的变化是不用一个一个的去new实例了。
* 更好的管理类实例，Component，Module，整个app的类实例结构变的很清晰。让我们更好的管理管理项目
* 解耦，因为有了dagger2,一个类的new代码就不会出现在一个项目的任何一个地方，构造函数发生变化，只需要修改module中的代码即可。
* 方便测试

## 如何引入Dagger2

* 配置apt插件(在build.gradle(Project:xxx)中添加如下代码)//Project的Gradle是根，意味着所有的module下都引用了

dependencies {

...

//添加apt插件

classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

}

* 添加依赖(在build.gradle(Module:app)中添加如下代码)

...

//添加如下代码，应用apt插件

apply plugin: 'android-apt'

...

dependencies {

...

compile 'com.google.dagger:dagger:2.4'

apt 'com.google.dagger:dagger-compiler:2.4'

//java注解

compile 'org.glassfish:javax.annotation:10.0-b28'

...

}

## Dagger2的注入流程

### 流程

步骤1：查找Module中是否存在创建该类的方法。

步骤2：若存在创建类方法，查看该方法是否存在参数

步骤2.1：若存在参数，则按从步骤1开始依次初始化每个参数

步骤2.2：若不存在参数，则直接初始化该类实例，依次依赖注入到此结束

步骤3：若不存在创建类方法，则查找Inject注解的构造函数，看构造函数是否存在参数

步骤3.1：若存在参数，则从步骤1开始依次初始化每个参数

步骤3.2：若不存在参数，则直接初始化该类实例，依次依赖注入到此结束

### 简单代码

**直接依赖**

public class MainActivity extends AppCompatActivity implements MainContract.View {

private MainPresenter mainPresenter;

...

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//实例化presenter 将view传递给presenter

mainPresenter = new MainPresenter(this);

//调用Presenter方法加载数据

mainPresenter.loadData();

...

}

}

public class MainPresenter {

//MainContract是个接口，View是他的内部接口，这里看做View接口即可

private MainContract.View mView;

MainPresenter(MainContract.View view) {

mView = view;

}

public void loadData() {

//调用model层方法，加载数据

...

//回调方法成功时

mView.updateUI();

}

**使用Dagger2**

@Module

public class MainModule {

private final MainContract.View mView;

public MainModule(MainContract.View view) {

mView = view;

}

@Provides

MainView provideMainView() {

return mView;

}

}

@Component(

modules = MainModule.class

)

public interface MainComponent {

MainView getMainView();//必须提供出来，这样才将MainModule中的@provider提供给了@Inject中的MainPresenter

void inject(MainActivity activity);

}

public class MainPresenter {

private MainContract.View mView;

@Inject

MainPresenter(MainContract.View view) {

mView = view;

}

public void loadData() {

//调用model层方法，加载数据

...

//回调方法成功时

mView.updateUI();

}

}

public class MainActivity extends AppCompatActivity implements MainContract.View {

@Inject

MainPresenter mainPresenter;

...

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

DaggerMainComponent.builder()

.mainModule(new MainModule(this))

.build()

.inject(this);

//调用Presenter方法加载数据

mainPresenter.loadData();

...

}

}

### 解析

我们先看MainActivity里的代码，之前是直接声明MainPresenter，现在在声明的基础上加了一个注解@Inject，表明MainPresenter是需要注入到MainActivity中，即MainActivity依赖于MainPresenter，这里要注意的是，使用@Inject时，不能用private修饰符修饰类的成员属性。

然后我们在MainPresenter的构造函数上同样加了@Inject注解。这样MainActivity里的mainPresenter与他的构造函数建立了某种联系。这种联系我们可以这样理解，当看到某个类被@Inject标记时，就会到他的构造方法中，如果这个构造方法也被@Inject标记的话，就会自动初始化这个类，从而完成依赖注入。

然后，他们之间并不会凭空建立起联系，因此我们想到，肯定需要一个桥梁，将他们连接起来，也就是下面要介绍的Component。Component是一个接口或者抽象类，用@Component注解标注（这里先不管括号里的modules），我们在这个接口中定义了一个inject()方法,参数是Mainactivity。然后rebuild一下项目，会生成一个以Dagger为前缀的Component类，这里是DaggerMainComponent，然后在MainActivity里完成下面代码.

DaggerMainComponent.builder()

.mainModule(new MainModule(this))

.build()

.inject(this);

此时Component就将@Inject注解的mainPresenter与其构造函数联系了起来。此时，看到这里，如果是初学者的话，一定会非常迷惑，究竟是怎么建立起联系的，实例化过程在哪？别急，后面会讲解这个过程原理的。

此时我们已经完成了presenter的注入过程，但是我们发现还有一个MainModule类，这个类是做什么的？MainModlue是一个注解类，用@Module注解标注，主要用来提供依赖。等等，刚才通过@Inject就可以完成依赖，为什么这里还要用到Module类来提供依赖？之所以有Module类主要是为了提供那些没有构造函数的类的依赖，这些类无法用@Inject标注，比如第三方类库，系统类，以及上面示例的View接口。

我们在MainModule类里声明了MainContract.View成员属性，在构造方法里将外界传进来的view赋值给mView，并通过一个@Provides标注的以provide开头的方法，将这个view返回，这个以provide开头的方法就是提供依赖的，我们可以创建多个方法来提供不同的依赖。那么这个类究竟是怎么作用的？可以想到上面提到的@Component注解括号里的东西了。就是下面这个

@Component(

modules = MainModule.class

)

public interface MainComponent {

void inject(MainActivity activity);

}

所以Module要发挥作用，还是要依靠于Component类，一个Component类可以包含多个Module类，用来提供依赖。我们接着看下面这段代码：

DaggerMainComponent.builder()

.mainModule(new MainModule(this))

.build()

.inject(this);

这里通过new MainModule(this)将view传递到MainModule里，然后MainModule里的provideMainView()方法返回这个View，当去实例化MainPresenter时，发现构造函数有个参数，此时会在Module里查找提供这个依赖的方法，将该View传递进去，这样就完成了presenter里View的注入。

我们来重新理一遍上面的注入过程，首先弄清楚以下几个概念：

* @Inject 带有此注解的属性或构造方法将参与到依赖注入中，Dagger2会实例化有此注解的类
* @Module 带有此注解的类，用来提供依赖，里面定义一些用@Provides注解的以provide开头的方法，这些方法就是所提供的依赖，Dagger2会在该类中寻找实例化某个类所需要的依赖。
* @Component 用来将@Inject和@Module联系起来的桥梁，从@Module中获取依赖并将依赖注入给@Inject

### 解析二

接着我们重新回顾一下上面的注入过程：

* 首先MainActivity需要依赖MainPresenter，因此，我们在里面用@Inject对MainPresenter进行标注，表明这是要注入的类。
* [然后，我们对MainPresenter的构造函数也添加注解@Inject，此时构造函数里有一个参数MainContract.View](mailto:然后，我们对MainPresenter的构造函数也添加注解@Inject，此时构造函数里有一个参数MainContract.View，因为MainPresenter需要依赖MainContract.View，)
* [因为MainPresenter需要依赖MainContract.View](mailto:然后，我们对MainPresenter的构造函数也添加注解@Inject，此时构造函数里有一个参数MainContract.View，因为MainPresenter需要依赖MainContract.View，)，所以我们定义了一个类，叫做MainModule，提供一个方法provideMainView，用来提供这个依赖，这个MainView是通过MainModule的构造函数注入进来的
* 接着我们需要定义Component接口类，并将Module包含进来，即

@Component(

modules = MainModule.class

)

public interface MainComponent {

void inject(MainActivity activity);

}

* 同时里面声明了一个inject方法，方法参数为ManActivity，将presenter注入给MainActivity

DaggerMainComponent.builder()

.mainModule(new MainModule(this))

.build()

.inject(this);

## Dagger2注入原理

Dagger2与其他依赖注入框架不同，它是通过apt插件在编译阶段生成相应的注入代码，下面我们就具体看看Dagger2生成了哪些注入代码？

### 步奏一:MainPresenter

**初始化中的参数MainView由MainModule提供**

我们先看MainPresenter这个类，在这个类中我们对构造方法用了@Inject标注，然后Rebuild Project，Dagger2会在/app/build/generated/apt/debug/目录下生成一个对应的工厂类MainPresenter\_Factory，我们看下面具体代码（为了方便理解，我把MainPresenter也贴了出来）

public class MainPresenter {

MainContract.View mView;

@Inject

MainPresenter(MainContract.View view) {

mView = view;

}

}

public final class MainPresenter\_Factory implements Factory<MainPresenter> {

private final Provider<MainContract.View> viewProvider;

public MainPresenter\_Factory(Provider<MainContract.View> viewProvider) {

assert viewProvider != null;

this.viewProvider = viewProvider;

}

@Override

public MainPresenter get() {

return new MainPresenter(viewProvider.get());

}

public static Factory<MainPresenter> create(Provider<MainContract.View> viewProvider) {

return new MainPresenter\_Factory(viewProvider);

}

}

对比MainPresenter,我们发现在MainPre\_Factory里也生成了对应的代码。首先是viewProvide，这是一个Provider类型，泛型参数就是我们的MainContract.View，接着通过构造方法，对viewProvider进行实例化。

其实这里有个疑惑，上面的成员属性为什么不直接是MainContract.View，而是Provider类型？我们应该想到MainContract.View是一个依赖，而依赖的提供者是MainModule，因此这个viewProvider一定是由MainModul提供的。

我们接着看下面的get()方法，看到这个方法，我想我们有点恍然大悟的感觉，原来MainPresenter的实例化就在这里，构造函数里的参数就是我们依赖的MainContract.View，它是由viewProvider通过get()提供。

接着是一个create()方法，并且有一个参数viewProvider，用来创建这个MainPresenter\_Factory类。

### 步奏二:MainModule

**初始化中的参数MainView由**DaggerMainComponent.builder().mainModule(new MainModule(this))**提供**

上面我们得出，viewProvider是由MainModule提供的，所以我们接着看MainModule所对应的注入类

@Module

public class MainModule {

private final MainContract.View mView;

public MainModule(MainContract.View view) {

mView = view;

}

@Provides

MainContract.View provideMainView() {

return mView;

}

}

public final class MainModule\_ProvideMainViewFactory implements Factory<MainContract.View> {

private final MainModule module;

public MainModule\_ProvideMainViewFactory(MainModule module) {

assert module != null;

this.module = module;

}

@Override

public MainContract.View get() {

return Preconditions.checkNotNull(

module.provideMainView(), "Cannot return null from a non-@Nullable @Provides method");

}

public static Factory<MainContract.View> create(MainModule module) {

return new MainModule\_ProvideMainViewFactory(module);

}

}

看到上面的类名，我们发现了一种对应关系，在MainModule中定义的@Provides修饰的方法会对应的生成一个工厂类，这里是MainModule\_ProvideMainViewFactory。

我们看到这个类里有一个get()方法，其中调用了MainModule里的provideMainView()方法来返回我们所需要的依赖MainContract.View。

还记得在MainPresenter\_Factory里的get()方法中，实例化MainPresenter时候的参数viewProvider.get()吗？到这里我们就明白了，原来那个viewProvider就是生成的MainModule\_ProvideMainViewFactory，然后调用了其get()方法，将我们需要的MainContract.View注入到MainPresenter里。

* 看到这里我们应该明白了MainPresenter的实例化过程。
* MainPresenter会对应的有一个工厂类，在这个类的get()方法中进行MainPresenter创建
* 而MainPresenter所需要的View依赖，是由MainModule里定义的以provide开头的方法所对应的工厂类提供的。
* 虽然我们明白了实例化的创建过程，但是此时还是有点疑惑
* MainPresenter\_Factory的创建是由create()完成的，那么create()是在哪调用的
* 此时创建的MainPresenter实例是怎么跟@Inject注解的MainPresenter关联起来的
* 没错就是Component。前面说过Component是连接@Module和@Inject的桥梁所以上面的疑惑就要到编译后Component所对应的类中寻找答案。

### 步奏三:Component

@Component(

modules = MainModule.class

)

public interface MainComponent {

void inject(MainActivity activity);

}

public final class DaggerMainComponent implements MainComponent {

private Provider<MainContract.View> provideMainViewProvider;

private Provider<MainPresenter> mainPresenterProvider;

private MembersInjector<MainActivity> mainActivityMembersInjector;

private DaggerMainComponent(Builder builder) {

assert builder != null;

initialize(builder);

}

public static Builder builder() {

return new Builder();

}

@SuppressWarnings("unchecked")

private void initialize(final Builder builder) {

//所有的create()方法，都是将方法中的参数传给调用方法的类的全局变量，然后使用变量的get()方法获取依赖

this.provideMainViewProvider = MainModule\_ProvideMainViewFactory.create(builder.mainModule);

this.mainPresenterProvider = MainPresenter\_Factory.create(provideMainViewProvider);

this.mainActivityMembersInjector = MainActivity\_MembersInjector.create(mainPresenterProvider);

}

@Override

public void inject(MainActivity activity) {

mainActivityMembersInjector.injectMembers(activity);//这里就将presenter注入到了activity

}

public static final class Builder {

private MainModule mainModule;

private Builder() {}

public MainComponent build() {

if (mainModule == null) {

throw new IllegalStateException(MainModule.class.getCanonicalName() + " must be set");

}

return new DaggerMainComponent(this);

}

public Builder mainModule(MainModule mainModule) {

this.mainModule = Preconditions.checkNotNull(mainModule);

return this;

}

}

}

从上面代码看到定义的MainComponent会生成一个对应的DaggerMainComponent，并且实现了MainComponent里的方法。我们看到代码中又出现了Provide类型的成员属性，前面说过这个Provide类型就是所提供的依赖，我们在看它们是在哪实例化的。我们看到有一个initialize()方法

@SuppressWarnings("unchecked")

private void initialize(final Builder builder) {

this.provideMainViewProvider = MainModule\_ProvideMainViewFactory.create(builder.mainModule);

this.mainPresenterProvider = MainPresenter\_Factory.create(provideMainViewProvider);

this.mainActivityMembersInjector = MainActivity\_MembersInjector.create(mainPresenterProvider);

}

看到这估计就明白了刚才的疑惑。首先创建了MainModule\_ProvideMainViewFactory实例，用来提供MainContract.View依赖。这里可能有个小疑惑，create()方法返回的是Factory类型，而provideMainViewProvider是个Provider类型，其实看源码就明白了，Factory继承自Provider。

public interface Factory<T> extends Provider<T> {

}

然后将provideMainViewProvider传递到MainPresenter\_Factory里，即就是前面讲到的viewProvider。接着将这个mainPresenterProvider又传递到MainActivity\_MembersInjector中进行实例化，我们看到这个类前面是MainActivity开头，因此可以想到是MainActivity对应得注入类，我们后面再分析这个类。

接着是我们在MainComponent里定义的Inject方法的实现，这里调用了mainActivityMembersInjector.injectMembers(activity)方法，将我们的MainActivity注入到该类中。

接着就是Builder内部类，用来创建我们的module以及自身实例。所以在DaggerMainComponent里主要用来初始化依赖，而真正的将这些依赖于Inject关联起来的就是刚才的MainActivity\_MembersInjector类，我们看看这个类里做了什么。

public final class MainActivity\_MembersInjector implements MembersInjector<MainActivity> {

private final Provider<MainPresenter> mainPresenterProvider;

public MainActivity\_MembersInjector(Provider<MainPresenter> mainPresenterProvider) {

assert mainPresenterProvider != null;

this.mainPresenterProvider = mainPresenterProvider;

}

public static MembersInjector<MainActivity> create(Provider<MainPresenter> mainPresenterProvider) {

return new MainActivity\_MembersInjector(mainPresenterProvider);

}

@Override

public void injectMembers(MainActivity instance) {

if (instance == null) {

throw new NullPointerException("Cannot inject members into a null reference");

}

instance.mainPresenter = mainPresenterProvider.get();

}

public static void injectMainPresenter(

MainActivity instance, Provider<MainPresenter> mainPresenterProvider) {

instance.mainPresenter = mainPresenterProvider.get();

}

}

这个类的关键就是injectMembers()方法，还记得这个方法在哪调用吗?我想你肯定记得，就在刚才提到的DaggerMainComponent类中的inject()方法里,所以这里的instance实例是由DaggerMainComponent提供的，然后我们看到了最关键的一句代码

instance.mainPresenter = mainPresenterProvider.get();

看到这，我想应该一切都明白了，将mainPresenterProvider中创建好的MainPresenter实例赋值给instance(MainActivity)的成员mainPresenter，这样我们用@Inject标注的mainPresenter就得到了实例化，接着就可以在代码中使用了。

到这里，就分析完了Dagger2的注入过程，如果不去看这些生成的类，就很难理解整个过程究竟是怎么发生的，从而导致还是不知道怎么去使用这个依赖注入框架。所以重点去理解这个内部实现原理是非常重要的

### 解释关键的这三步和注入代码

@SuppressWarnings("unchecked")

private void initialize(final Builder builder) {

1. this.provideMainViewProvider = MainModule\_ProvideMainViewFactory.create(builder.mainModule);
2. this.mainPresenterProvider = MainPresenter\_Factory.create(provideMainViewProvider);
3. this.mainActivityMembersInjector = MainActivity\_MembersInjector.create(mainPresenterProvider);

}

所有的create()方法，目的都是创建一个本类，并将方法中的参数传给调用方法的类的全局变量，然后使用变量的get()方法获取依赖

①

调用类ProvideMainViewFactory： 是MainModule生成的类

参数builder.mainModule： 是DaggerMainComponent.builder.mainModule(new MainModule(this))传递进来的new MainModule(this)，它的provideMainView()返回 this、mVIew、MainView、Activity

②

调用类MainPresenter\_Factory： 是MainPresenter生成的类

参数provideMainViewProvider： 是上一步传递进来的MainModule生成的类，它的get返回new MainPresenter(viewProvider.get())

③

调用类MainActivity\_MembersInjector： 是Component生成的类DaggerMainComponent 的参数

参数mainPresenterProvider： 是上一步传进来的MainPresenter生成的类，它的get返回 new MainPresenter(viewProvider.get()),其中viewProvider.get()也就是MainModule的get()方法，返回this、mVIew、MainView、Activity

使用的代码每一步的意义：

DaggerMainComponent

.builder()

.mainModule(new MainModule(this))

.build()

.inject(this);

.builder()-----DaggerMainComponen的静态方法，作用是创建内部类return new Builder()

.new MainModule(this)-----MainModule的构造方法，作用是将activiy注入进MainModule的MainView变量

.mainModule()-----DaggerMainComponent的内部类Builder的mainModule方法，作用是将new MainModule(this)注入进内部类Builder的变量

.build()-----DaggerMainComponent的内部类Builder的build方法,作用是创建return new DaggerMainComponent(this)

.inject(this)-----DaggerMainComponen的inject方法，作用是给这个this指代的activity的presenter赋值

inject()的具体步奏：

inject()------instance.mainPresenter = mainPresenterProvider.get()--------------MainPresenter, retun presenter

mainPresenterProvider.get()------ return new MainPresenter(viewProvider.get())--------------return presenter

viewProvider.get()------ return module.provideMainView()--------------moudle, return mView

module.provideMainView()------ return this、mView、MainView、Activity

## 来个代码

### AppModule

@Module

public class AppModule {

private final App application;

public AppModule(App application) {

this.application = application;

}

@Provides

@Singleton

App provideApplicationContext(){

return application;

}

@Provides

@Singleton

RetrofitHelper provideRetrofitHelper(WkllmServiceApi wkllmService, MessageServiceApi messageServiceApi){

return new RetrofitHelper(wkllmService,messageServiceApi);

}

...

}

### HttpModule

@Module

public class HttpModule {

@Singleton

@Provides

Retrofit.Builder provideRetrofitBuilder() {

return new Retrofit.Builder();

}

@Singleton

@Provides

@WkllmeUrl

Retrofit provideWkllmeRetrofit(Retrofit.Builder builder, OkHttpClient client) {

return createRetrofit(builder, client, ApiDefine.HOST\_BASE\_URL\_MAIN);

}

...

}

### ActivityModule

@Module

public class ActivityModule {

private Activity activity;

public ActivityModule(Activity activity) {

this.activity = activity;

}

@Provides

@ActivityScope

Activity provideActivity(){

return this.activity;

}

}

### AppComponent

@Singleton

@Component(

modules = {

AppModule.class,

HttpModule.class,

PageModule.class

}

)

public interface AppComponent {

App getContext(); //提供App的Context

RetrofitHelper retrofitHelper(); //提供Http的帮助类

RealmHelper realmHelper(); //提供数据库帮助类

SharePreferenceHelper sharePreferenceHelper(); //配置文件帮助类

FileHelper fileHelper(); //文件操作帮助类

MineFragment mineFragment();

WorkbenchFragment workbenchFragment();

GymnasiumFragment gymnasiumFragment();

}

### ActivityComponent

@ActivityScope

@Component(

dependencies = AppComponent.class,

modules = {

ActivityModule.class

})

public interface ActivityComponent {

Activity getActivity();

//注入activity

void inject(MainActivity mainActivity);

...

}

### App

public class App extends Application {

...

public static AppComponent getAppComponent() {

if (appComponent == null) {

appComponent = DaggerAppComponent

.builder()

.appModule(new AppModule(instance))

.httpModule(new HttpModule())

.build();

}

return appComponent;

}

...

}

### BasePresenter

public interface BasePresenter<T extends BaseView>{

void attachView(T view);

void detachView();

}

### RxPresenter

public class RxPresenter<T extends BaseView> implements BasePresenter<T> {

protected T mView;

...

@Override

public void attachView(T view) {

this.mView = view;

}

@Override

public void detachView() {

this.mView = null;

unSubscribe();

}

}

### MainPresenter

public class MainPresenter extends RxPresenter<MainContract.View>{

@Inject

public MainPresenter(RetrofitHelper retrofitHelper,

SharePreferenceHelper sharePreferenceHelper,

RealmHelper realmHelper,

MineFragment mineFragment,

GymnasiumFragment gymnasiumFragment,

WorkbenchFragment workbenchFragment) {

this.retrofitHelper = retrofitHelper;

this.sharePreferenceHelper = sharePreferenceHelper;

this.realmHelper = realmHelper;

this.mineFragment = mineFragment;

this.gymnasiumFragment = gymnasiumFragment;

this.workbenchFragment = workbenchFragment;

this.fragments = new ArrayList<>();

this.mTabEntities = new ArrayList<>();

}

...

}

### BaseActivity

public abstract class BaseActivity<T extends BasePresenter> {

...

@Inject

protected T mPresenter;

...

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

...

mPresenter.attachView(this);

}

@Override

protected void onDestroy() {

super.onDestroy();

...

if (mPresenter != null)

mPresenter.detachView();

}

protected ActivityComponent getActivityComponent() {

return DaggerActivityComponent.builder()

.appComponent(App.getAppComponent())

.activityModule(getActivityModule())

.build();

}

}

### MainActivity

public class MainActivity extends BaseActivity<MainPresenter> {

...

@Override

protected void initInject() {

getActivityComponent().inject(this);

}

...

}

# 依赖、Dagger2讲解资料

## 依赖注入概念

对于依赖注入（Dependency Injection，以下简称DI）基本理念简单描述下：

* 首先这是一种代码模式，这个模式里面有两个概念：Client和Dependency。
* 假如你的代码里面，一个类用到了另外一个类，那么前者叫Client，后者叫Dependency。例如LoginPresenter用到了UserManager，那么LoginPresenter叫Client，UserManager叫Dependency。
* 当然，这是个相对的概念，一个类可以是某个类的Dependency，却是另外一个类的Client。比如说如果UserManager里面用到了Retrofit，那么相对于Retrofit，UserManager又是Client。
* DI的基本思想就是，对于Dependency的创建过程，并不在Client里面进行，而是由外部创建好，然后通过某种方式set到Client里面。这种模式，就叫做依赖注入。

## DI的常见实现方式

实现DI这种模式其实很简单，有多种方式

**setter injection---**通过setter方法，其实就是实现DI的一种方式

public class LoginPresenter {

private UserManager mUserManager;

public void setUserManager(UserManager userManager) { //<==

this.mUserManager = userManager;

}

public void login(String username, String password) {

//... some other code

mUserManager.performLogin(username, password);

}

}

**argument injection---**通过方法的参数传递进去，也是实现DI的一种方式：

public class LoginPresenter {

public void login(UserManager userManager, String username, String password) {

//... some other code

userManager.performLogin(username, password);

}

}

**Constructor Injection---**更常用的方式，将Dependency作为Client的构造方法的参数传递进去：

public class LoginPresenter {

private final UserManager mUserManager;

public LoginPresenter(UserManager userManager) {

this.mUserManager = userManager;

}

public void login(String username, String password) {

//... some other code

mUserManager.performLogin(username, password);

}

}

其实一般来说，提到DI指的都是**Constructor Injection**这种方式。这种方式的好处是，依赖关系非常明显。你必须在创建这个类的时候，就提供必要的dependency。这从某种程度上来说，也是在说明这个类所完成的功能。因此，尽量使用 Constructor injection。

说到这里，你可能会有一个疑问，如果把依赖都声明在Constructor的参数里面，这会不会让这个类的Constructor参数变得非常多？如果真的发生这种情况了，那往往说明这个类的设计是有问题的，需要重构。

为什么呢？我们代码里面的类，一般可以分为两种，一种是Data类，比如说UserInfo，OrderInfo等等。另外一种是Service类，比如UserManager, AudioPlayer等等。所以这个问题就有两种情况了：

1. 如果Constructor里面传入的很多是基本类型的数据或数据类，那么或许你要做的，是创建一个（或者是另一个）数据类把这些数据封装一下，这个过程的价值可是大大滴！而不仅仅是封装一下参数的问题，有了一个类，很多的方法就可以放到这个类里面了。这点请参考Martin Fowler的《重构》第十章“Introduce Parameter Object”。
2. 如果传入的很多是service类，那么这说明这个类做的事情太多了，不符合单一职责的原则（Single Responsibility Principle，SRP），因此，需要重构。

## Dagger2——DI框架的由来

### 不使用DI框架的问题

如果不使用DI框架，而全部采用手工来做DI的话，那么所有的Dependency都需要在最上层的client来生成，这可不是件好事情。继续用我们前面的例子来具体说明一下。

假设有一个登录界面，LoginActivity，他有一个LoginPresenter，LoginPresenter用到了UserManager和PasswordValidator，为了让问题变得更明显一点，我们假设UserManager用到SharedPreference（用来存储一些用户的基本设置等）和UserApiService，而UserApiService又需要由Retrofit创建，而Retrofit又用到OkHttpClient（比如说你要自己控制timeout、cache等东西）。应用DI模式，

**UserManager的设计如下：**

public class UserManager {

private final SharedPreferences mPref;

private final UserApiService mRestAdapter;

public UserManager(SharedPreferences preferences, UserApiService userApiService) {

this.mPref = preferences;

this.mRestAdapter = userApiService;

}

/\*\*Other code\*/

}

**Dependency --- LoginPresenter的设计如下：**

public class LoginPresenter {

private final UserManager mUserManager;

private final PasswordValidator mPasswordValidator;

public LoginPresenter(UserManager userManager, PasswordValidator passwordValidator) {

this.mUserManager = userManager;

this.mPasswordValidator = passwordValidator;

}

/\*\*Other code\*/

}

**Client --- LoginActivity里面要new一个presenter，需要做的事情如下：**

public class LoginActivity extends AppCompatActivity {

private LoginPresenter mLoginPresenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

OkHttpClient okhttpClient = new OkHttpClient.Builder()

.connectTimeout(30, TimeUnit.SECONDS)

.build();

Retrofit retrofit = new Retrofit.Builder()

.client(okhttpClient)

.baseUrl("https://api.github.com")

.build();

UserApiService userApiService = retrofit.create(UserApiService.class);

SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);

UserManager userManager = new UserManager(preferences, userApiService);

PasswordValidator passwordValidator = new PasswordValidator();

mLoginPresenter = new LoginPresenter(userManager, passwordValidator);

}

}

这个也太夸张了，LoginActivity所需要的，不过是一个LoginPresenter而已，然而它却需要知道LoginPresenter的Dependency是什么，LoginPresenter的Dependency的Dependency又是什么，然后new一堆东西出来。而且可以预见的是，这个app的其他地方也需要这里的OkHttpClient、Retrofit、SharedPreference、UserManager等等dependency，因此也需要new这些东西出来，造成大量的代码重复，和不必要的object instance生成。然而如前所述，我们又必须用到DI模式，这个怎么办呢？

想想，如果能达到这样的效果，那该有多好：我们只需要在一个类似于dependency工厂的地方统一生产这些dependency，以及这些dependency的dependency。所有需要用到这些Dependency的client都从这个工厂里面去获取。而且更妙的是，一个client（比如说LoginActivity）只需要知道它直接用到的Dependency（LoginPresenter），而不需要知道它的Dependency（LoginPresenter）又用到哪些Dependency（UserManager和PasswordValidator）。系统自动识别出这个依赖关系，从工厂里面把需要的Dependency找到，然后把这个client所需要的Dependency创建出来。

有这样一个东西，帮我们实现这个效果吗？相信聪明的你已经猜到了，回答是肯定的，它就是我们今天要介绍的dagger2。

### 解药：Dagger2

在dagger2里面，负责生产这些Dependency的统一工厂叫做 Module ，所有的client最终是要从module里面获取Dependency的，然而他们不是直接向module要的，而是有一个专门的“工厂管理员”，负责接收client的要求，然后到Module里面去找到相应的Dependency，提供给client们。这个“工厂管理员”叫做 Component。基本上，这是dagger2里面最重要的两个概念。

## Dagger2各部分作用

### Module——生产Dependency的工厂

首先是Module，一个Module对应到代码里面就是一个类，只不过这个类需要用dagger2里面的一个annotation @Module来标注一下，来表示这是一个Module，而不是一个普通的类。我们说Module是生产Dependency的地方，对应到代码里面就是Module里面有很多方法，这些方法做的事情就是创建Dependency。用上面的例子中的Dependency来说明：

@Module

public class AppModule {

public OkHttpClient provideOkHttpClient() {

OkHttpClient okhttpClient = new OkHttpClient.Builder()

.connectTimeout(30, TimeUnit.SECONDS)

.build();

return okhttpClient;

}

public Retrofit provideRetrofit(OkHttpClient okhttpClient) {

Retrofit retrofit = new Retrofit.Builder()

.client(okhttpClient)

.baseUrl("https://api.github.com")

.build();

return retrofit;

}

}

在上面的Module（AppModule）中，有两个方法provideOkHttpClient()和provideRetrofit(OkHttpClient okhttpClient)，分别创建了两个Dependency，OkHttpClient和Retrofit。但是呢，我们也说了，一个Module就是一个类，这个类有一些生产Dependency的方法，但它也可以有一些正常的，不是用来生产Dependency的方法。那怎么样让管理员知道，一个Module里面哪些方法是用来生产Dependency的，哪些不是呢？为了方便做这个区分，dagger2规定，所有生产Dependency的方法必须用 @Provides这个annotation标注一下。所以，上面的 AppModule正确的写法应该是：

@Module

public class AppModule {

@Provides

public OkHttpClient provideOkHttpClient() {

OkHttpClient okhttpClient = new OkHttpClient.Builder()

.connectTimeout(30, TimeUnit.SECONDS)

.build();

return okhttpClient;

}

@Provides

public Retrofit provideRetrofit(OkHttpClient okhttpClient) {

Retrofit retrofit = new Retrofit.Builder()

.client(okhttpClient)

.baseUrl("https://api.github.com")

.build();

return retrofit;

}

}

这种用来生产Dependency的、用 @Provides修饰过的方法叫Provider方法。这里要注意第二个Provider方法 provideRetrofit(OkHttpClient okhttpClient)，这个方法有一个参数，是OkHttpClient。这是因为创建一个Retrofit对象需要一个OkHttpClient的对象，这里通过参数传递进来。这样做的好处是，当Client向管理员（Component）索要一个Retrofit的时候，Component会自动找到Module里面找到生产Retrofit的这个 provideRetrofit(OkHttpClient okhttpClient)方法，找到以后试图调用这个方法创建一个Retrofit对象，返回给Client。但是调用这个方法需要一个OkHttpClient，于是Component又会去找其他的provider方法，看看有没有哪个会生产OkHttpClient。于是就找到了上面的第一个provider方法： provideOkHttpClient()。找到以后，调用这个方法，创建一个OkHttpClient对象，再调用 provideRetrofit(OkHttpClient okhttpClient)方法，把刚刚创建的OkHttpClient对象传进去，创建出一个Retrofit对象，返回给Client。当然，如果最后找到的 provideOkHttpClient()方法也需要其他参数，那么管理员还会继续递归的找下去，直到所有的Dependency都被满足了，再一个一个创建Dependency，然后把最终Client需要的Dependency呈递给Client。

很好，现在我们把文章开头的例子中的所有Dependency都用这种方式，在 AppModule里面声明一个provider方法：

@Module

public class AppModule {

@Provides

public OkHttpClient provideOkHttpClient() {

OkHttpClient okhttpClient = new OkHttpClient.Builder()

.connectTimeout(30, TimeUnit.SECONDS)

.build();

return okhttpClient;

}

@Provides

public Retrofit provideRetrofit(OkHttpClient okhttpClient) {

Retrofit retrofit = new Retrofit.Builder()

.client(okhttpClient)

.baseUrl("https://api.github.com")

.build();

return retrofit;

}

@Provides

public UserApiService provideUserApiService(Retrofit retrofit) {

return retrofit.create(UserApiService.class);

}

@Provides

public SharedPreferences provideSharedPreferences(Context context) {

return PreferenceManager.getDefaultSharedPreferences(context);

}

@Provides

public UserManager provideUserManager(SharedPreferences preferences, UserApiService service) {

return new UserManager(preferences, service);

}

@Provides

public PasswordValidator providePasswordValidator() {

return new PasswordValidator();

}

@Provides

public LoginPresenter provideLoginPresenter(UserManager userManager, PasswordValidator validator) {

return new LoginPresenter(userManager, validator);

}

}

上面的代码如果你仔细看的话，会发现一个问题，那就是其中的SharedPreference provider方法 provideSharedPreferences(Context context)需要一个context对象，但是 AppModule里面并没有context 的Provider方法，这个怎么办呢？对于这个问题，你可以再创建一个context provider方法，但是context对象从哪来呢？我们可以自定义一个Application，里面提供一个静态方法返回一个context，这种做法相信大家都干过。Application类如下：

public class MyApplication extends Application {

private static Context sContext;

@Override

public void onCreate() {

super.onCreate();

sContext = this;

}

public static Context getContext() {

return sContext;

}

}

provider方法如下：

@Provides

public Context provideContext() {

return MyApplication.getContext();

}

但是这种方法不是很好，为什么呢，因为context的获得相当于是写死了，只能从MyApplication.getContext()，如果测试环境下想把Context换成别的，还要给MyApplication定义一个setter，然后调用MyApplication.setContext(...)，这个就绕的有点远。更好的做法是，把Context作为 AppModule的一个构造参数，从外面传进来（Constructor Injection DI模式，还记得吗？）：

@Module

public class AppModule {

private final Context mContext;

public AppModule(Context context) {

this.mContext = context;

}

@Provides

public Context provideContext() {

return mContext;

}

//其他的provider方法

}

是的，一个Module就是一个正常的类，它也可以有构造方法，以及其他正常类的特性。你可能会想那给构造函数的context对象从哪来呢？别急，这个问题马上解答。

### Component——Dependency工厂管理员

跟Module不同的是，我们在实现Component时，不是定义一个类，而是定义一个接口（interface）：

public interface AppComponent {

}

名字可以随便取，跟Module需要用 @Module修饰一下类似的，一个dagger2的Component需要用 @Component修饰一下，来标注这是一个dagger2的Component，而不是一个普通的interface，所以正确的定义方式是：

@Component

public interface AppComponent {

}

在实际情况中，可能有多个Module，也可能有多个Component，那么当Component接收到一个Client的Dependency请求时，它怎么知道要从哪个Module里面去找这些Dependency呢？它不可能遍历我们的每一个类，然后找出所有的Module，再遍历所有Module的Provider方法，去找Dependency，这样先不说能不能做到，就算做得到，效率也太低了。因此dagger2规定，我们在定义Component的时候，必须指定这个管理员“管理”哪些工厂（Module）。指定的方法是，把需要这个Component管理的Module传给 @Component这个注解的modules属性（或者叫方法？），如下：

@Component(

modules = {

AppModule.class

}

)

public interface AppComponent {

}

modules属性接收一个数组，里面是这个Component管理的所有Module。在上面的例子中，AppComponent只管理AppModule一个。

### Component给Client提供Dependency的方法

前面我们讲了Module和Component的实现，接下来就是Component怎么给Client提供Dependency的问题了。一般来说，有两种，当然总共不止这两种，只不过这两种最常用，也最好理解，一般来说用这两种就够了，因此这里不赘述其他的方法。

#### 方法一：在Component里面定义一个返回Dependency的方法

第一种是在Component里面定义一个返回Dependency的方法，比如LoginActivity需要LoginPresenter，那么我们可以在AppComponent里面定义一个返回LoginPresenter的方法：

@Component(modules = {AppModule.class})

public interface AppComponent {

LoginPresenter loginPresenter();

}

你可能会好奇，为什么Component只需要定义成接口就行了，不是应该定义一个类，然后自己使用Module去做这件事吗？如果是这样的话，那就太low了。

dagger2的工作原理是，在你的java代码编译成字节码的过程中，dagger2会对所有的Component（就是用 @Component修饰过的interface）进行处理，自动生成一个实现了这个interface的类，生成的类名是Component的名字前面加上“Dagger”。比如我们定义的 AppComponent，对应的自动生成的类叫做DaggerAppComponent。我们知道，实现一个interface需要实现里面的所有方法，因此，DaggerAppComponent是实现了 loginPresenter();这个方法的。实现的方式大致就是从 AppComponent管理的 AppModule里面去找LoginPresenter的Provider方法，然后调用这个方法，返回一个LoginPresenter。

因此，使用这种方式，当Client需要Dependency的时候，首先需要用DaggerAppComponent这个类创建一个对象，然后调用这个对象的 loginPresenter()方法，这样Client就能获得一个LoginPresenter了，这个DaggerAppComponent对象的创建及使用方式如下：

public class LoginActivity extends AppCompatActivity {

private LoginPresenter mLoginPresenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

AppComponent appComponent = DaggerAppComponent.builder().appModule(new AppModule(this)).build();

mLoginPresenter = appComponent.loginPresenter();

//从此之后，mLoginPresenter就被实例化了

//mLoginPresenter.isLogin()

}

}

总结一下，我们到现在为止，做了什么：

1. 我们定义了一个 AppModule类，里面定义了一些Provider方法
2. 定义了一个 AppComponent，里面定义了一个返回LoginPresenter的方法loginPresenter()。

就这样，我们便可以使用 DaggerAppComponent.builder().appModule(new AppModule(this)).build().loginPresenter(); 来获取一个LoginPresenter对象了。这简直就是magic，不是吗？

如果不是dagger2，而是我们自己来实现这个AppComponent interface，想想我们需要做哪些事情：

1. 定义一个Constructor，接受一个AppModule对象，保存在field中（mAppModule)
2. 实现loginPresenter()方法，调用mAppModule的provideLoginPresenter(UserManager userManager, PasswordValidator validator)方法，这时候发现这个方法需要两个参数 UserManager和PasswordValidator。
3. 调用provideUserManager(SharedPreferences preferences, UserApiService service)来获取一个UserManager，这时候发现这个方法又需要两个参数 SharedPreferences和 UserApiService。
4. 调用provideSharedPreferences(Context context)来获取一个SharedPreference，这时候发现先要有一个context
5. 。。。
6. 。。。
7. 。。。

说白了，就是把文章开头我们写的那段代码又实现了一遍，而使用dagger2，我们就做了前面描述的两件事而已，这里面错综复杂的Dependency关系dagger2帮我们自动理清了，生成相应的代码，去调用相应的Provider方法，满足这些依赖关系。

##### 好处在哪里

也许这里举得这个例子不足以让你觉得有什么大不了的，但是你要知道，一个正常的App，可不仅仅有一个Login page而已，稍微大点的App，Dependency都有几百甚至上千个，对于服务器程序来说，Dependency则更多。对于这点，大家可以去看Dagger2主要作者的这个视频，他里面提到了Google一个android app有3000行代码专门来管理Dependency，而一个Server app甚至有10万行这样的代码。这个时候要去手动new这些dependency、并且要以正确的顺序new出来，简直会要人命。而且让问题更加棘手的是，随着app的演进需求的变更，Dependency之间的关系也在动态的变化。比如说UserManager不再使用SharedPreference，而是使用database，这个时候UserManager的构造函数里面少了一个SharedPreferences，多了一个DatabaseHelper这样的东西，那么如果使用正常的方式管理Dependency，所有new UserManager的地方都要改。而用dagger2，你只需要在 AppModule里面添加一个DatabaseHelper Provider方法，同时把UserManager的provider方法第一参数从SharedPreferences改成DatabaseHelper就好了，所有用到UserManager的地方不需要做任何更改，LoginPresenter不需要做任何更改，LoginActivity不需要任何更改，这难道不是magic吗？

说点题外话，这种把问题(我们这里是依赖关系)描述出来，而不是把实现过程写出来的编程风格叫Declarative programming（申诉式编程），跟它对应的叫Imperative Programming（命令式编程），相对于后者，前者的优势是：可读性更高，side effect（副作用）更少，可扩展性更高等等。这是一种编程风格，跟语言、框架无关。当然，有的语言或框架天生就能让程序员更容易的使用这种style来编程。这方面最显著的当属Prolog，有兴趣的可以去了解下，绝对mind-blowing！

对于Java或Android开发者来说，想让我们的代码更加declarative，最好的方式是使用dagger2和RxJava。

#### 方法二：Field Injection

话说回来，我们继续介绍dagger2，前面我们介绍了Component给Client提供Dependency的第一种方式，接下来继续介绍第二种方式，这种方式叫 Field injection 。这里我们继续用LoginActivity的例子来说明，LoginActivity需要一个LoginPresenter。那么使用这种方式的做法是，我们就在LoginActivity里面定义一个LoginPresenter的field，这个field需要使用 @Inject修饰一下：

public class LoginActivity extends AppCompatActivity {

@Inject

LoginPresenter mLoginPresenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

然后在onCreate()里面，我们把DaggerAppComponent对象创建出来，调用这个对象的inject方法，把LoginActivity传进去：

public class LoginActivity extends AppCompatActivity {

@Inject

LoginPresenter mLoginPresenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

AppComponent appComponent = DaggerAppComponent.builder().appModule(new AppModule(this)).build();

appComponent.inject(this);

//从此之后，mLoginPresenter就被实例化了

//mLoginPresenter.isLogin()

}

}

当然，我们需要先在AppComponent里面定义一个inject(LoginActivity loginActivity)方法：

@Component(modules = {AppModule.class})

public interface AppComponent {

void inject(LoginActivity loginActivity);

}

DaggerAppComponent实现这个方法的方式是，去LoginActivity里面所有被 @Inject修饰的field，然后调用 AppModule相应的Provider方法，赋值给这个field。这里需要注意的是，@Inject field不能使private，不然dagger2找不到这个field。

通常来说，这种方式比第一种方式更简单，代码也更简洁。假设LoginActivity还需要其他的Dependency，比如需要一个统计打点的Dependency（StatManager），那么你只需要在AppModule里面定义一个Provider方法，然后在LoginActivity里面声明另外一个field就好了：

public class LoginActivity extends AppCompatActivity {

@Inject

LoginPresenter mLoginPresenter;

@Inject

StatManager mStatManager;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

AppComponent appComponent = DaggerAppComponent.builder().appModule(new AppModule(this)).build();

appComponent.inject(this);

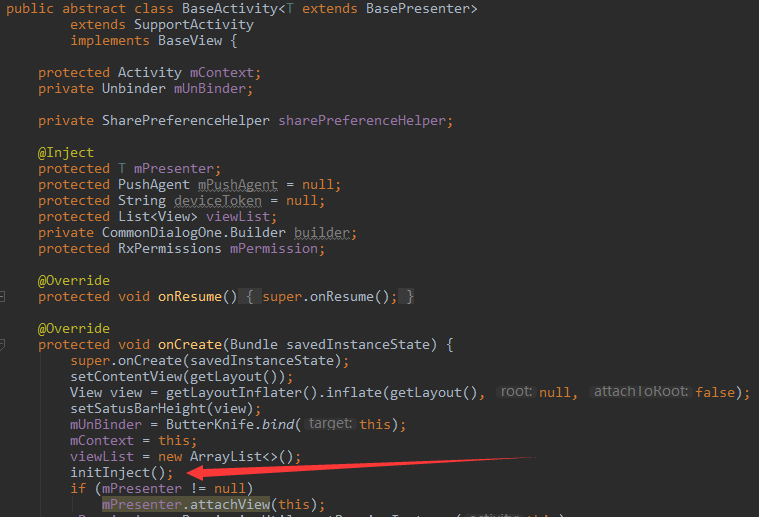
}

}

无论有多少个@Inject field，都只需要调用一次appComponent.inject(this);。用过了你就会觉得，恩，好爽！

不过，需要注意的一点是，这种方式不支持继承，比如说LoginActivity继承自一个 BaseActivity，而@Inject StatManager mStatManager;是放在BaseActivity里面的，那么在LoginActivity里面调用 appComponent.inject(this);并不会让BaseActivity里面的 mStatManager得到实例化，你必须在 BaseActivity里面也调用一次appComponent.inject(this)。

不支持继承，但是可以这么做，因为在BaseActivity中调用了子Activity的InitInject方法，所以@Inject protected T mPresenter成功注入！

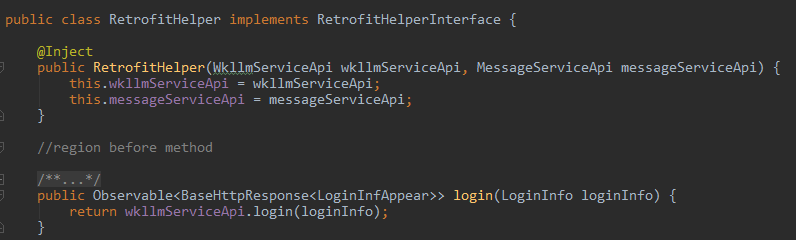


#### 总的来说注入方法：

**通过完整的inject流程**



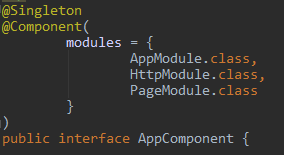
**构造方法**



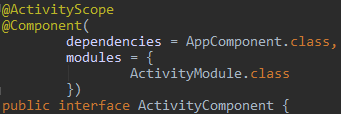
## Component之间互相有依赖的用法

**例子：依赖AppComponent的ActivityComponent**

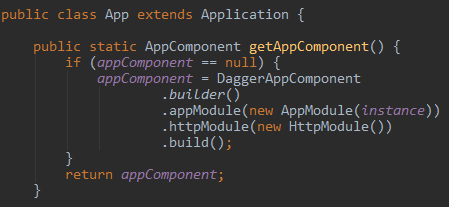
AppComponent



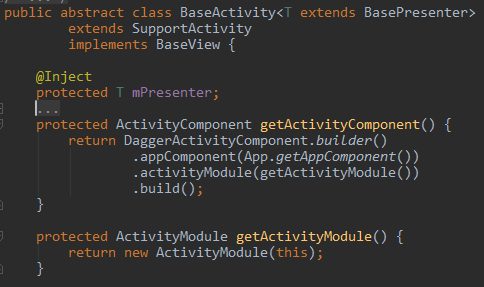
ActivityComponent



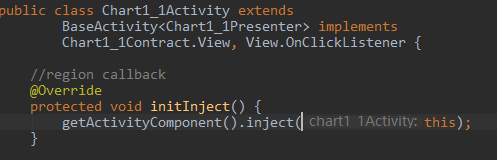
App



BaseActivity



Activity



# OkHttp的使用

## 基本用法

### Http Get

**public** **static** **void** **httpGet**(**String** path){

**OkHttpClient** client = **new** OkHttpClient();**//创建okHttpClient对象**

**Request** request = **new** **Request**.Builder().url(path).build();**//创建一个Request**

**Call** call = client.newCall(request);**//new call**

call.enqueue(**new** Callback() {

@Override

**public** **void** **onResponse**(**Response** arg0) **throws** IOException {

**String** htmlStr = arg0.body().string();

}

@Override

**public** **void** **onFailure**(**Request** arg0, **IOException** arg1) {

}

});

}

/\*\*

\* 以上就是发送一个get请求的步骤，首先构造一个Request对象，参数最起码有个url，当然你可以通过Request.Builder设置更多的参数比如：header、method等。

\* 然后通过request的对象去构造得到一个Call对象，类似于将你的请求封装成了任务，既然是任务，就会有execute()和cancel()等方法

\* 最后，我们希望以异步的方式去执行请求，所以我们调用的是call.enqueue，将call加入调度队列，然后等待任务执行完成，我们在Callback中即可得到结果

\*

\* 需要注意几点：

\* onResponse回调的参数是response，一般情况下，比如我们希望获得返回的字符串，可以通过response.body().string()获取；如果希望获得返回的二进制字节数组，则调用response.body().bytes()；如果你想拿到返回的inputStream，则调用response.body().byteStream()

\* 看到这，你可能会奇怪，竟然还能拿到返回的inputStream，看到这个最起码能意识到一点，这里支持大文件下载，有inputStream我们就可以通过IO的方式写文件。

\* 不过也说明一个问题，这个onResponse执行的线程并不是UI线程。的确是的，如果你希望操作控件，还是需要使用handler等

\* 我们这里是异步的方式去执行，当然也支持阻塞的方式，上面我们也说了Call有一个execute()方法，你也可以直接调用call.execute()通过返回一个Response。

\*/

### Http Post 携带参数

**public** **static** **void** **httpPost**(**String** path){

**FormEncodingBuilder** builder = **new** FormEncodingBuilder();

builder.add("username", "张弘扬");**//键值对，可以多放**

**Request** request = **new** **Request**.Builder().url(path).post(builder.build()).build();

**new** OkHttpClient().newCall(request).enqueue(**new** Callback() {...});

}

### 基于http的文件上传

**public** **static** **void** **httpFile**(**String** path, **File** file){

**RequestBody** fileBody = **RequestBody**.*create*(**MediaType**.*parse*("application/octet-stream"), file);

**RequestBody** requestBody = **new** MultipartBuilder().type(**MultipartBuilder**.**FORM**)

.addPart(**Headers**.*of*("Content-Disposition", "form-data; name=\"username\""), **RequestBody**.*create*(**null**, "陈光宇"))

.addPart(**Headers**.*of*("Content-Disposition", "form-data; name=\"mFile\";filename=\"wjd.mp4\""), fileBody).build();

**Request** request = **new** **Request**.Builder().url(path).post(requestBody).build();

**new** OkHttpClient().newCall(request).enqueue(**new** Callback() {...});

}

### 封装（主要是思路）

**public** **class** **OkHttpUtil** {

**private** **Handler** mHandler;**//回到主线程的handler**

**private** **static** **OkHttpUtil** ***mInstance***;

**private** **OkHttpClient** client;

**//初始化**

**private** **OkHttpUtil**(){

mHandler = **new** Handler(**Looper**.*getMainLooper*());**//直接获取主线程的handler**

client = **new** OkHttpClient();**//全局使用一个client**

}

**//单例**

**public** **static** **OkHttpUtil** **getInstance**(){

**if** (***mInstance*** == **null**) {

**synchronized** (**OkHttpUtil**.**class**) {

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** OkHttpUtil();

}

}

}

**return** ***mInstance***;

}

**//Http Get（内部使用）**

**private** **void** **httpGet**(**String** path, **final** MyCallBack callBack){

**Request** request = **new** **Request**.Builder().url(path).build();

**Call** call = client.newCall(request);**//new call**

asyncGet(call, callBack);

}

**// Http Post 携带参数（内部使用）**

**private** **void** **httpPost**(**String** path, Map<**String**, **String**> map, **final** MyCallBack callBack){

**FormEncodingBuilder** builder = **new** FormEncodingBuilder();

Iterator<Entry<**String**, **String**>> iterator = map.entrySet().iterator();

**while** (iterator.hasNext()) {

Entry<**String**, **String**> entry = iterator.next();

builder.add(entry.getKey(), entry.getValue());

}

**Request** request = **new** **Request**.Builder().url(path).post(builder.build()).build();

asyncGet(client.newCall(request), callBack);

}

**//基于http的文件上传（内部使用）**

**private** **void** **httpFile**(**String** path, **File** file, **final** MyCallBack callBack){

**RequestBody** fileBody = **RequestBody**.*create*(**MediaType**.*parse*("application/octet-stream"), file);

**RequestBody** requestBody = **new** MultipartBuilder().type(**MultipartBuilder**.**FORM**)

.addPart(**Headers**.*of*("Content-Disposition",

"form-data; name=\"username\""), **RequestBody**.*create*(**null**, "陈光宇"))

.addPart(**Headers**.*of*("Content-Disposition",

"form-data; name=\"mFile\";filename=\"wjd.mp4\""), fileBody).build();

**Request** request = **new** **Request**.Builder().url(path).post(requestBody).build();

asyncGet(client.newCall(request), callBack);

}

**//加载图片（内部使用）**

**private** **void** **httpImage**(**final** **String** url, **final** MyImgCallBack callBack){

**final** **Request** request = **new** **Request**.Builder().url(url).build();

**Call** call = client.newCall(request);

call.enqueue(**new** Callback() {

@Override

**public** **void** **onFailure**(**Request** request, **IOException** e) {

callBack.onFailure();

}

@Override

**public** **void** **onResponse**(**Response** response) {

**InputStream** is = **null**;

**try** {

is = response.body().byteStream();

**final** **Bitmap** bm = **BitmapFactory**.*decodeStream*(is);

mHandler.post(**new** Runnable() {

@Override

**public** **void** **run**() {

callBack.onResponse(bm);

}

});

} **catch** (**IOException** e) {

callBack.onFailure();

} **finally** {

**if** (is != **null**) {

**try** {

is.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}

}

});

}

**//回调回调**

**private** **void** **asyncGet**(**Call** call, **final** MyCallBack callBack){

call.enqueue(**new** Callback() {**//这个就是异步了，异步请求**

@Override

**public** **void** **onResponse**(**final** **Response** arg0) **throws** IOException {

**final** **String** arg1 = arg0.body().string();**//这是一个坑，sting方法是请求网络的！！！**

mHandler.post(**new** Runnable() {**//handler.post方法回到主线程**

@Override

**public** **void** **run**() {

callBack.onResponse(arg1)**//回调调用网络请求处的回调函数**

}

});

}

@Override

**public** **void** **onFailure**(**final** **Request** arg0, **final** **IOException** arg1) {

mHandler.post(**new** Runnable() {

@Override

**public** **void** **run**() {

callBack.onFailure(arg0, arg1);

}

});

}

});

}

**//Http Get（外部使用）**

**public** **static** **void** **getRequset**(**final** **String** path, **final** MyCallBack callBack){

*getInstance*().httpGet(path, callBack);**//getInstance方法得到实例，然后就可以调非静态private方法了！！**

}

}

**public** **class** **MainActivity** **extends** **Activity** {

**private** **EditText** edt;

**private** **String** path = "http://www.baidu.com";

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

edt = (**EditText**) findViewById(**R**.**id**.**edt**);

}

**public** **void** **btn**(**View** view){

**OkHttpUtil**.*getRequset*(path, **new** MyCallBack() {

@Override

**public** **void** **onResponse**(**String** arg0) {

edt.setText(arg0);

}

@Override

**public** **void** **onFailure**(**Request** arg0, **IOException** arg1) {

arg1.printStackTrace();

}

});

}

}

## 添加统一请求头和参数

### 添加统一请求头

* HTTP 头是 HTTP 请求和响应中的重要组成部分。在创建 HTTP 请求时需要设置一些 HTTP 头。在得到 HTTP 的响应之后，也会需要对其中包含的 HTTP 头进行解析。
* 从代码的角度来说，HTTP 头的数据结构是 Map<String, List<String>>类型。也就是说，对于每个 HTTP 头，可能有多个值。但是大部分 HTTP 头都只有一个值，只有少部分 HTTP 头允许多个值。
* 在设置 HTTP 头时，使用 header(name, value) 方法来设置 HTTP 头的唯一值。对同一个 HTTP 头，多次调用该方法会覆盖之前设置的值。使用 addHeader(name, value) 方法来为 HTTP 头添加新的值。
* 在读取 HTTP 头时，使用 header(name) 方法来读取 HTTP 头的最近出现的值。如果该 HTTP 头只有单个值，则返回该值；如果有多个值，则返回最后一个值。使用 headers(name) 方法来读取 HTTP 头的所有值。

public class Headers {

public static void main(String[] args) throws IOException {

OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()

.url("http://www.baidu.com")

.header("User-Agent", "My super agent")

.addHeader("Accept", "text/html")

.build();

Response response = client.newCall(request).execute();

if (!response.isSuccessful()) {

throw new IOException("服务器端错误: " + response);

}

System.out.println(response.header("Server"));

System.out.println(response.headers("Set-Cookie"));

}

}

代码中对request使用 header 方法设置了 User-Agent 头的值，并使用 addHeader 方法添加了一个 Accept 头的值。在进行解析 response 时，通过 header 方法来获取 Server 头的单个值，通过 headers 方法来获取 Set-Cookie 头的所有值

### 添加统一参数

**第一种 添加在请求链接尾部**

Request original = chain.request();

HttpUrl originalUrl = original.url();

HttpUrl url = originalUrl.newBuilder() //请求尾部链接

.addQueryParameter(VERSION, "MY\_DEVICE\_ID")

.addQueryParameter(TOKEN, "MY\_TOKEN")

.build();

Request request = original.newBuilder()

.method(original.method(), original.body())

.url(url)//将新的url添加到请求里

.build();

**第二种 添加在请求form表单里**

最麻烦的一种，一般情况下不会用这种方法

//要统一添加的form表单

RequestBody formBody = new FormBody.Builder()

.add(DEVICE\_ID, "MY\_DEVICE\_ID")

.add(TOKEN, "MY\_TOKEN")

.build();

//默认添加formBody后不能添加新的form表单，需要先将RequestBody转成string去拼接

//以前的formBody

String postBodyString = bodyToString(original.body());

//加上现在的formBody

postBodyString += ((postBodyString.length() > 0) ? "&" : "") + bodyToString(formBody);

Request request = original.newBuilder()

.method(original.method(), original.body())

//string转回成RequestBody,添加到请求里

.post(RequestBody.create(MediaType.parse("application/x-www-form-urlencoded"), postBodyString))

.build();

private static String bodyToString(final RequestBody request){

try {

final RequestBody copy = request;

final Buffer buffer = new Buffer();

if(copy != null) {

copy.writeTo(buffer);

} else {

return "";

}

return buffer.readUtf8();

}

catch (final IOException e) {

return "did not work";

}

}

## 拦截器Interceptor(可用于Retrofit2)

### 拦截器的定义：

Interceptors area powerful mechanism that can monitor, rewrite, and retry calls —— 拦截器可以用来转换，重试，重写请求的机制

拦截器是 OkHttp 提供的对 HTTP 请求和响应进行统一处理的强大机制。拦截器在实现和使用上类似于 Servlet 规范中的过滤器。多个拦截器可以链接起来，形成一个链条。拦截器会按照在链条上的顺序依次执行（所以loger拦截器放最后比较好）

* 拦截器在执行时，可以先对请求的 Request 对象进行修改；再得到响应的 Response 之后，可以进行修改之后再返回。
* Interceptor 接口只包含一个方法 intercept，其参数是 Chain 对象，Chain 对象表示的是当前的拦截器链条。
* 通过 Chain 的 request 方法可以获取到当前的 Request 对象进行额外的处理，例如：

request = request.newBuilder()

.addHeader("sign", sign)

.addHeader("accountId", accountId)

.build();

* 通过 Chain 对象的 proceed 方法来继续拦截器链条的执行，可以对得到的 Response 对象进行额外的处理，例如：

response.newBuilder()

.header("Cache-Control", "public, only-if-cached, max-stale=" + maxStale)

.removeHeader("Pragma")

.build();

* Interceptor的典型使用场景，就是对request和response的Headers进行编辑

### 记录请求和响应信息的拦截器：

public class LoggingInterceptor implements Interceptor {

public Response intercept(Chain chain) throws IOException {

Request request = chain.request();

long t1 = System.nanoTime();

System.out.println(String.format("发送请求: [%s] %s%n%s", request.url(), chain.connection(), request.headers()));

Response response = chain.proceed(request);

long t2 = System.nanoTime();

System.out.println(String.format("接收响应: [%s] %.1fms%n%s", response.request().url(), (t2 - t1) / 1e6d, response.headers()));

return response;

}

}

### 两类拦截器（Interceptors）

OkHttp 中的拦截器分成应用和网络拦截器两种。应用拦截器对于每个 HTTP 响应都只会调用一次，可以通过不调用 Chain.proceed 方法来终止请求，也可以通过多次调用 Chain.proceed 方法来进行重试。网络拦截器对于调用执行中的自动重定向和重试所产生的响应也会被调用，而如果响应来自缓存，则不会被调用。

拦截器的接口类：

public interface Interceptor {

Response intercept(Chain chain) throws IOException;

interface Chain {

Request request();

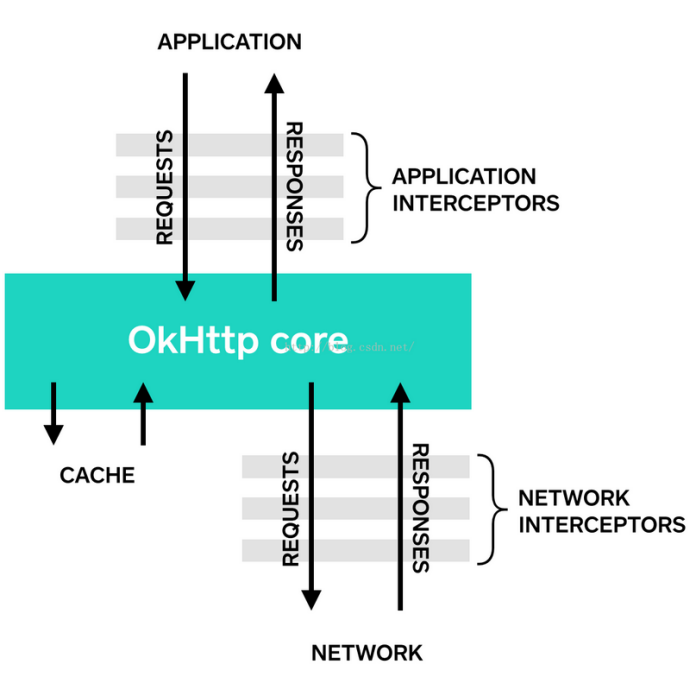
Response proceed(Request request) throws IOException;

Connection connection();

}

}

在注册拦截器时，可以注册成两类拦截器，分别为应用拦截器（Application Interceptors）和网络拦截器（Network Interceptors），如下图：



#### 应用拦截器（ApplicationInterceptors）

主要用于查看请求信息及返回信息，如链接地址、头信息、参数信息等，通过下面两种方式注册的为应用拦截器：

* 方式一：在OkHttpClient.Builder中添加

new OkHttpClient.Builder().addInterceptor(interceptor)

* 方式二：在okHttpClient中直接添加

okHttpClient.interceptors().add(interceptor)

参考下面的log-拦截器定义：

if (BuildConfig.DEBUG) {

HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor();

loggingInterceptor.setLevel(HttpLoggingInterceptor.Level.BODY);

builder.addInterceptor(loggingInterceptor);

}

#### 网络拦截器（NetworkInterceptors）

可以添加、删除或替换请求头信息，还可以改变的请求携带的实体，通过下面两种方式注册的为网络拦截器：

* 方式一：在OkHttpClient.Builder中添加

new OkHttpClient.Builder().addNetworkInterceptor(interceptor)

* 方式二：在okHttpClient中直接添加

okHttpClient. networkInterceptors().add(interceptor)

例如，如果你连接到一个支持压缩的网络服务器你可以使用一个应用拦截器来添加请求实体压缩。

builder.addNetworkInterceptor(new GzipRequestInterceptor());

/\*\*

\* 这个拦截器压缩了请求实体. 很多网络服务器无法处理它

\*/

final class GzipRequestInterceptor implements Interceptor {

@Override

public Response intercept(Interceptor.Chain chain) throws IOException {

Request originalRequest = chain.request();

if (originalRequest.body() == null || originalRequest.header("Content-Encoding") != null) {

return chain.proceed(originalRequest);

}

Request compressedRequest = originalRequest.newBuilder()

.header("Content-Encoding", "gzip")

.method(originalRequest.method(), gzip(originalRequest.body()))

.build();

Response response = chain.proceed(compressedRequest);

return response;

}

private RequestBody gzip(final RequestBody body) {

return new RequestBody() {

@Override

public MediaType contentType() {

return body.contentType();

}

@Override

public long contentLength() {

return -1; // We don't know the compressed length in advance!

}

@Override

public void writeTo(BufferedSink sink) throws IOException {

BufferedSink gzipSink = Okio.buffer(new GzipSink(sink));

body.writeTo(gzipSink);

gzipSink.close();

}

};

}

}

#### 拦截器选择

**应用拦截器**

* 不需要担心中间过程的响应,如重定向和重试.
* 总是只调用一次,即使HTTP响应是从缓存中获取.
* 观察应用程序的初衷. 不关心OkHttp注入的头信息如: If-None-Match.
* 允许短路而不调用 Chain.proceed(),即中止调用.
* 允许重试,使 Chain.proceed()调用多次.

**网络拦截器**

* 能够操作中间过程的响应,如重定向和重试.
* 当网络短路而返回缓存响应时不被调用.
* 只观察在网络上传输的数据.
* 携带请求来访问连接.

### 一些拦截器工具类

#### 缓存时间拦截器

Interceptor cacheInterceptor = new Interceptor() {

@Override

public Response intercept(Chain chain) throws IOException {

Request request = chain.request();

if (!SystemUtil.isNetworkConnected()) {

request = request.newBuilder()

.cacheControl(CacheControl.FORCE\_CACHE)

.build();

}

Response response = chain.proceed(request);

if (SystemUtil.isNetworkConnected()) {

int maxAge = 0;

// 有网络时, 不缓存, 最大保存时长为0

response.newBuilder()

.header("Cache-Control", "public, max-age=" + maxAge)

.removeHeader("Pragma")

.build();

} else {

int maxStale = 0;

if (true) {

maxStale = 0;//没有网络也不走缓存

} else {

maxStale = 60 \* 60 \* 24 \* 28;// 无网络时，设置超时为4周

}

response.newBuilder()

.header("Cache-Control", "public, only-if-cached, max-stale=" + maxStale)

.removeHeader("Pragma")

.build();

}

return response;

}

}

builder.addNetworkInterceptor(cacheInterceptor);

builder.addInterceptor(cacheInterceptor);

#### Logger拦截器

//打印头信息

if (BuildConfig.DEBUG) {

HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor();

loggingInterceptor.setLevel(HttpLoggingInterceptor.Level.HEADERS);

builder.addInterceptor(loggingInterceptor);

}

//打印json信息

if (Constants.isNeedHttpJson) {

Interceptor loggerInterceptor = new Interceptor() {

@Override

public Response intercept(Chain chain) throws IOException {

//请求参数

Request request = chain.request();

try {

RequestBody requestBody = request.body();

Buffer buffer = new Buffer();

requestBody.writeTo(buffer);

Charset charset = UTF8;

MediaType contentType = requestBody.contentType();

if (contentType != null) {

charset = contentType.charset(UTF8);

}

String str = buffer.clone().readString(charset);

Logger.json(str);

} catch (Exception ignored) {

}

//返回参数

Response response = chain.proceed(request);

try {

ResponseBody responseBody = response.body();

BufferedSource source = responseBody.source();

source.request(Long.MAX\_VALUE);

Buffer buffer = source.buffer();

Charset charset = UTF8;

MediaType contentType = responseBody.contentType();

if (contentType != null) {

charset = contentType.charset(UTF8);

}

String str = buffer.clone().readString(charset);

Logger.json(str);

} catch (Exception ignored) {

}

return response;

}

};

builder.addInterceptor(loggerInterceptor);

}

##### Logger打印到硬盘中

FormatStrategy formatStrategy = CsvFormatStrategy.newBuilder()

.tag(TAG)

.build();

Logger.addLogAdapter(new DiskLogAdapter(formatStrategy));

#### 统一请求头(参数)拦截器

Interceptor apikey = new Interceptor() {

@Override

public Response intercept(Chain chain) throws IOException {

Request request = chain.request();

RequestBody requestBody = request.body();

Buffer buffer = new Buffer();

requestBody.writeTo(buffer);

Charset charset = UTF8;

MediaType contentType = requestBody.contentType();

if (contentType != null) {

charset = contentType.charset(UTF8);

}

if (false) {//添加统一参数Demo

HttpUrl newUrl = request.url().newBuilder() //请求尾部链接

.addQueryParameter("VERSION", "MY\_DEVICE\_ID")

.addQueryParameter("TOKEN", "MY\_TOKEN")

.build();

request = request.newBuilder()

.url("newUrl")//将新的url添加到请求里

.build();

}

String userToken = SharedPreferenceUtil.getUserToken();

String accountId = SharedPreferenceUtil.getAccountId();

String orgId = SharedPreferenceUtil.getOrgId();

String urlStr = request.url().toString();

if (urlStr .contains("user/login") || urlStr.contains("register")) {

if (urlStr .contains("user/login")) {

oginInfo loginInfo = JsonUtil.fromJsonToObj(obj, LoginInfo.class);

accountId = loginInfo.getAccountId();

}

request = request.newBuilder()

.addHeader("accountId", accountId)

.build();

} else if (request.url().toString().contains("upload")) {

sign = SecurityUtil.signData("");

request = request.newBuilder()

.addHeader("accountId", accountId)

.addHeader("userToken", userToken)

.build();

} else {

request = request.newBuilder()

.addHeader("accountId", accountId)

.addHeader("userToken", userToken)

.addHeader("chooseOrgId", orgId)

.build();

}

Response response = chain.proceed(request);

return response ;

}

};

builder.addInterceptor(apikey);

## Client(可用于Retrofit2)

* Builder和Client

OkHttpClient.Builder builder = new OkHttpClient.Builder();

OkHttpClient mOkHttpClient = builder.build();

* 提供Builder，构造Client

private OkHttpClient provideClient(OkHttpClient.Builder builder) {

.....

}

### 错误重连

builder.retryOnConnectionFailure(true);

### 添加超时

builder.connectTimeout(10 \* 1000, TimeUnit.SECONDS);

builder.readTimeout(15 \* 1000, TimeUnit.SECONDS);

builder.writeTimeout(15 \* 1000, TimeUnit.SECONDS);

### 添加缓存大小

String PATH\_DATA = App.getInstance().getCacheDir().getAbsolutePath() + File.separator + "data";

String PATH\_CACHE = PATH\_DATA + "/NetCache";

File cacheFile = new File(Constants.PATH\_CACHE);

Cache cache = new Cache(cacheFile, 1024 \* 1024 \* 50);

builder.cache(cache);

### 添加拦截器

builder.addNetworkInterceptor(interceptor);

builder.addInterceptor(interceptor);

# Retrofit2

## Retrofit入门

Retrofit 其实相当简单，简单到源码只有37个文件，其中22个文件是注解还都和HTTP有关，真正暴露给用户的类并不多

### 接口列表



### 创建Retrofit实例

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

创建Retrofit实例时需要通过Retrofit.Builder,并调用baseUrl方法设置URL。注： Retrofit2 的baseUlr 必须以 /（斜线） 结束，不然会抛出一个IllegalArgumentException，所以如果你看到别的教程没有以 / 结束，那么多半是直接从Retrofit 1.X 照搬过来的。

### 接口定义

以获取指定id的Blog为例:

public interface BlogService {

@GET("blog/{id}")//这里的{id} 表示是一个变量等同blog/?id==xxx

Call<ResponseBody> getBlog(@Path("id") int id);

}

注意，这里是interface不是class，所以我们是无法直接调用该方法，我们需要用Retrofit创建一个BlogService的代理对象。

BlogService service = retrofit.create(BlogService.class);

拿到代理对象之后，就可以调用该方法啦

### 接口调用

Call<ResponseBody> call = service.getBlog(2);

// 用法和OkHttp的call如出一辙,

// 不同的是如果是Android系统回调方法执行在主线程

call.enqueue(new Callback<ResponseBody>() {

@Override

public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {

try {

System.out.println(response.body().string());

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void onFailure(Call<ResponseBody> call, Throwable t) {

t.printStackTrace();

}

});

打印结果:

{"code":200,"msg":"OK","data":{"id":2,"date":"2016-04-15 03:17:50","author":"怪盗kidou","title":"Retrofit2 测试2","content":"这里是 Retrofit2 Demo 测试服务器2"},"count":0,"page":0}

**示例源码见Example01 .java**

## Retrofit注解详解

上面提到Retrofit 共22个注解，这节就专门介绍这22个注解，为帮助大家更好理解我将这22个注解分为三类，并用表格的形式展现出来,表格上说得并不完整，具体的见源码上的例子注释。

### 第一类：HTTP请求方法



以上表格中的除HTTP以外都对应了HTTP标准中的请求方法，而HTTP注解则可以代替以上方法中的任意一个注解,有3个属性：method、path,hasBody,下面是用HTTP注解实现上面 Example01.java 的例子。

public interface BlogService {

/\*\*

\* method 表示请求的方法，区分大小写

\* path表示路径

\* hasBody表示是否有请求体

\*/

@HTTP(method = "GET", path = "blog/{id}", hasBody = false)

Call<ResponseBody> getBlog(@Path("id") int id);

}

注：method 的值 retrofit 不会做处理，所以要自行保证其准确性，使用小写也可以是因为示例源码中的服务器不区分大小写，所以希望大家注意

**示例源码见 Example02.java**

### 第二类：标记类



**示例源码见Example03.java**

### 第三类：参数类



* 注1：{占位符}和PATH尽量只用在URL的path部分，url中的参数使用Query和QueryMap 代替，保证接口定义的简洁
* 注2：Query、Field和Part这三者都支持数组和实现了Iterable接口的类型，如List，Set等，方便向后台传递数组。

Call<ResponseBody> foo(@Query("ids[]") List<Integer> ids);

//结果：ids[]=0&ids[]=1&ids[]=2

Path

**示例源码见 Example01.java**

Field、FieldMap、Part和PartMap

**示例源码见 Example03.java**

Header和Headers

**示例源码见 Example04.java**

Query、QueryMap、Url

**示例源码见 Example05.java**

## Client（OkHttpClient）

.client(OkHttpClient)

需要自定义时，则可以使用该方法，如：处理header、Cookie、connectTimeout、readTimeout、writeTimeout、loggingInterceptor、stetho 、charset、调式等

## Converter之GSON

.addConverterFactory(GsonConverterFactory.create())

在默认情况下Retrofit只支持将HTTP的响应体转换换为ResponseBody，这也是什么我在前面的例子接口的返回值都是 Call<ResponseBody>，但如果响应体只是支持转换为ResponseBody的话何必要引用泛型呢，返回值直接用一个Call就行了嘛，既然支持泛型，那说明泛型参数可以是其它类型的，而Converter就是Retrofit为我们提供用于将ResponseBody转换为我们想要的类型，有了Converter之后我们就可以写把我们的第一个例子的接口写成这个样子了：

public interface BlogService {

@GET("blog/{id}")

Call<Result<Blog>> getBlog(@Path("id") int id);

}

当然只改变泛型的类型是不行的，我们在创建Retrofit时需要明确告知用于将ResponseBody转换我们泛型中的类型时需要使用的Converter

引入Gson支持:

compile 'com.squareup.retrofit2:converter-gson:2.0.2'

通过GsonConverterFactory为Retrofit添加Gson支持：

Gson gson = new GsonBuilder()

//配置你的Gson

.setDateFormat("yyyy-MM-dd hh:mm:ss")

.create();

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

//可以接收自定义的Gson，当然也可以不传

.addConverterFactory(GsonConverterFactory.create(gson))

.build();

**示例源码见 Example06.java**

这样Retrofit就会使用Gson将ResponseBody转换我们想要的类型。

这是时候我们终于可以演示如使创建一个Blog了！

@POST("blog")

Call<Result<Blog>> createBlog(@Body Blog blog);

被@Body注解的的Blog将会被Gson转换成RequestBody发送到服务器。

BlogService service = retrofit.create(BlogService.class);

Blog blog = new Blog();

blog.content = "新建的Blog";

blog.title = "测试";

blog.author = "怪盗kidou";

Call<Result<Blog>> call = service.createBlog(blog);

结果：

Result{code=200, msg='OK', data=Blog{id=20, date='2016-04-21 05:29:58', author='怪盗kidou', title='测试', content='新建的Blog'}, count=0, page=0}

**示例源码见 Example07.java**

## CallAdapter之RxJava

.addConverterFactory(GsonConverterFactory.create())

说到Retrofit就不得说到另一个火到不行的库RxJava，网上已经不少文章讲如何与Retrofit结合，但这里还是会有一个RxJava的例子，不过这里主要目的是介绍使用CallAdapter所带来的效果。

第3节介绍的Converter是对于Call<T>中T的转换，而CallAdapter则可以对Call转换，这样的话Call<T>中的Call也是可以被替换的，而返回值的类型就决定你后续的处理程序逻辑，同样Retrofit提供了多个CallAdapter，这里以RxJava的为例，用Observable代替Call：

引入RxJava支持:

compile 'com.squareup.retrofit2:adapter-rxjava:2.0.2'

compile 'com.squareup.retrofit2:adapter-rxjava2:2.3.0'// 针对rxjava2.x（adapter-rxjava2的版本要 >= 2.2.0）

通过RxJavaCallAdapterFactory为Retrofit添加RxJava支持：

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(RxJavaCallAdapterFactory.create())

.addCallAdapterFactory(RxJava2CallAdapterFactory.create()) // 针对rxjava2.x

.build();

接口设计：

public interface BlogService {

@POST("/blog")

Observable<Result<List<Blog>>> getBlogs();

}

使用：

BlogService service = retrofit.create(BlogService.class);

service.getBlogs(1)

.subscribeOn(Schedulers.io())

.subscribe(new Subscriber<Result<List<Blog>>>() {

@Override

public void onCompleted() {

System.out.println("onCompleted");

}

@Override

public void onError(Throwable e) {

System.err.println("onError");

}

@Override

public void onNext(Result<List<Blog>> blogsResult) {

System.out.println(blogsResult);

}

});

结果：

Result{code=200, msg='OK', data=[Blog{id=1, date='2016-04-15 03:17:50', author='怪盗kidou', title='Retrofit2 测试1', content='这里是 Retrofit2 Demo 测试服务器1'},.....], count=20, page=1}

**示例源码见 Example08.java**

像上面的这种情况最后我们无法获取到返回的Header和响应码的，如果我们需要这两者，提供两种方案：

* 用Observable<Response<T>>代替Observable<T> ,这里的Response指retrofit2.Response
* 用Observable<Result<T>> 代替Observable<T>，这里的Result是指retrofit2.adapter.rxjava.Result,这个Result中包含了Response的实例

## 自定义Converter

本节的内容是教大家实现在一简易的Converter，这里以返回格式为Call<String>为例。

在此之前先了解一下Converter接口及其作用：

public interface Converter<F, T> {

// 实现从 F(rom) 到 T(o)的转换

T convert(F value) throws IOException;

// 用于向Retrofit提供相应Converter的工厂

abstract class Factory {

// 这里创建从ResponseBody其它类型的Converter，如果不能处理返回null

// 主要用于对响应体的处理

public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] annotations,

Retrofit retrofit) {

return null;

}

// 在这里创建 从自定类型到ResponseBody 的Converter,不能处理就返回null，

// 主要用于对Part、PartMap、Body注解的处理

public Converter<?, RequestBody> requestBodyConverter(Type type,

Annotation[] parameterAnnotations, Annotation[] methodAnnotations, Retrofit retrofit) {

return null;

}

// 这里用于对Field、FieldMap、Header、Path、Query、QueryMap注解的处理

// Retrfofit对于上面的几个注解默认使用的是调用toString方法

public Converter<?, String> stringConverter(Type type, Annotation[] annotations, Retrofit retrofit) {

return null;

}

}

}

我们要想从Call<ResponseBody> 转换为 Call<String> 那么对应的F和T则分别对应ResponseBody和String，我们定义一个StringConverter并实现Converter接口。

public static class StringConverter implements Converter<ResponseBody, String> {

public static final StringConverter INSTANCE = new StringConverter();

@Override

public String convert(ResponseBody value) throws IOException {

return value.string();

}

}

我们需要一个Fractory来向Retrofit注册StringConverter

public static class StringConverterFactory extends Converter.Factory {

public static final StringConverterFactory INSTANCE = new StringConverterFactory();

public static StringConverterFactory create() {

return INSTANCE;

}

// 我们只实现从ResponseBody 到 String 的转换，所以其它方法可不覆盖

@Override

public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] annotations, Retrofit retrofit) {

if (type == String.class) {

return StringConverter.INSTANCE;

}

//其它类型我们不处理，返回null就行

return null;

}

}

使用Retrofit.Builder.addConverterFactory向Retrofit注册我们StringConverterFactory：

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

// 如是有Gson这类的Converter 一定要放在其它前面

.addConverterFactory(StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.build();

注：addConverterFactory是有先后顺序的，如果有多个ConverterFactory都支持同一种类型，那么就是只有第一个才会被使用，而GsonConverterFactory是不判断是否支持的，所以这里交换了顺序还会有一个异常抛出，原因是类型不匹配。

只要返回值类型的泛型参数就会由我们的StringConverter处理,不管是Call<String>还是Observable<String>。有没有很简单?如果你有其它的需求处理的就自己实现吧。

**示例源码见 Example09.java**

## 自定义CallAdapter

本节将介绍如何自定一个CallAdapter，并验证是否所有的String都会使用我们第5节中自定义的Converter。先看一下CallAdapter接口定义及各方法的作用：

public interface CallAdapter<T> {

// 直正数据的类型 如Call<T> 中的 T

// 这个 T 会作为Converter.Factory.responseBodyConverter 的第一个参数

// 可以参照上面的自定义Converter

Type responseType();

<R> T adapt(Call<R> call);

// 用于向Retrofit提供CallAdapter的工厂类

abstract class Factory {

// 在这个方法中判断是否是我们支持的类型，returnType 即Call<Requestbody>和`Observable<Requestbody>`

// RxJavaCallAdapterFactory 就是判断returnType是不是Observable<?> 类型

// 不支持时返回null

public abstract CallAdapter<?> get(Type returnType, Annotation[] annotations,

Retrofit retrofit);

// 用于获取泛型的参数 如 Call<Requestbody> 中 Requestbody

protected static Type getParameterUpperBound(int index, ParameterizedType type) {

return Utils.getParameterUpperBound(index, type);

}

// 用于获取泛型的原始类型 如 Call<Requestbody> 中的 Call

// 上面的get方法需要使用该方法。

protected static Class<?> getRawType(Type type) {

return Utils.getRawType(type);

}

}

}

了解了CallAdapter的结构和其作用之后，我们就可以开始自定义我们的CallAdapter了，本节以CustomCall<String>为例。在此我们需要定义一个CustomCall，不过这里的CustomCall作为演示只是对Call的一个包装，并没有实际的用途。

public static class CustomCall<R> {

public final Call<R> call;

public CustomCall(Call<R> call) {

this.call = call;

public R get() throws IOException {

return call.execute().body();

}

}

有了CustomCall，我们还需要一个CustomCallAdapter来实现 Call<T> 到 CustomCall<T>的转换，这里需要注意的是最后的泛型，是我们要返回的类型。

public static class CustomCallAdapter implements CallAdapter<CustomCall<?>> {

private final Type responseType;

// 下面的 responseType 方法需要数据的类型

CustomCallAdapter(Type responseType) {

this.responseType = responseType;

}

@Override

public Type responseType() {

return responseType;

}

@Override

public <R> CustomCall<R> adapt(Call<R> call) {

// 由 CustomCall 决定如何使用

return new CustomCall<>(call);

}

}

提供一个CustomCallAdapterFactory用于向Retrofit提供CustomCallAdapter：

public static class CustomCallAdapterFactory extends CallAdapter.Factory {

public static final CustomCallAdapterFactory INSTANCE = new CustomCallAdapterFactory();

@Override

public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {

// 获取原始类型

Class<?> rawType = getRawType(returnType);

// 返回值必须是CustomCall并且带有泛型

if (rawType == CustomCall.class && returnType instanceof ParameterizedType) {

Type callReturnType = getParameterUpperBound(0, (ParameterizedType) returnType);

return new CustomCallAdapter(callReturnType);

}

return null;

}

}

使用addCallAdapterFactory向Retrofit注册CustomCallAdapterFactory

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(Example09.StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(CustomCallAdapterFactory.INSTANCE)

.build();

注： addCallAdapterFactory与addConverterFactory同理，也有先后顺序。

**示例源码见 Example10.java**

## 示例源码整合

### Example01

/\*\*

\* [Retrofit入门]源码

\*/

public class Example01 {

public interface BlogService {

@GET("blog/{id}") //这里的{id} 表示是一个变量等同blog/?id==xxx

Call<ResponseBody> getBlog(/\*\* 这里的id表示的是上面的{id} \*/@Path("id") int id);

}

public static void main(String[] args) throws IOException {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

BlogService service = retrofit.create(BlogService.class);

Call<ResponseBody> call = service.getBlog(2);

// 用法和OkHttp的call如出一辙

// 不同的是如果是Android系统回调方法执行在主线程

call.enqueue(new Callback<ResponseBody>() {

@Override

public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {

try {

System.out.println(response.body().string());

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void onFailure(Call<ResponseBody> call, Throwable t) {

t.printStackTrace();

}

});

}

}

### Example02

/\*\*

\* [Retrofit注解详解 之 HTTP注解]源码

\*/

public class Example02 {

public interface BlogService {

/\*\*

\* method 表示请求的方法，区分大小写，retrofit 不会做处理

\* path表示路径

\* hasBody表示是否有请求体

\*/

@HTTP(method = "GET", path = "blog/{id}", hasBody = false)

Call<ResponseBody> getBlog(@Path("id") int id);

}

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

BlogService service = retrofit.create(BlogService.class);

Call<ResponseBody> call = service.getBlog(2);

ResponseBodyPrinter.printResponseBody(call);

}

}

### Example03

/\*\*

\* [Retrofit注解详解 之 FormUrlEncoded/Field/FieldMap/Multipart/Part/PartMap注解]源码

\*/

public class Example03 {

public interface BlogService {

/\*\*

\* {@link FormUrlEncoded} 表明是一个表单格式的请求（Content-Type:application/x-www-form-urlencoded）

\* <code>Field("username")</code> 表示将后面的 <code>String name</code> 中name的取值作为 username 的值

\*/

@POST("/form")

@FormUrlEncoded

Call<ResponseBody> testFormUrlEncoded1(@Field("username") String name, @Field("age") int age);

/\*\*

\* Map的key作为表单的键

\*/

@POST("/form")

@FormUrlEncoded

Call<ResponseBody> testFormUrlEncoded2(@FieldMap Map<String, Object> map);

//--------------------------------------------------------------------------

/\*\*

\* {@link Part} 后面支持三种类型，{@link RequestBody}、{@link okhttp3.MultipartBody.Part} 、任意类型

\* 除 {@link okhttp3.MultipartBody.Part} 以外，其它类型都必须带上表单字段({@link okhttp3.MultipartBody.Part} 中已经包含了表单字段的信息)，

\*/

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload1(@Part("name") RequestBody name, @Part("age") RequestBody age, @Part MultipartBody.Part file);

/\*\*

\* PartMap 注解支持一个Map作为参数，支持 {@link RequestBody } 类型，

\* 如果有其它的类型，会被{@link retrofit2.Converter}转换，如后面会介绍的 使用{@link com.google.gson.Gson} 的 {@link retrofit2.converter.gson.GsonRequestBodyConverter}

\* 所以{@link MultipartBody.Part} 就不适用了,所以文件只能用<b> @Part MultipartBody.Part </b>

\*/

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload2(@PartMap Map<String, RequestBody> args, @Part MultipartBody.Part file);

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload3(@PartMap Map<String, RequestBody> args);

}

//------------------------------------------------------------------------

//------------------------------------------------------------------------

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

BlogService service = retrofit.create(BlogService.class);

// 演示 @FormUrlEncoded 和 @Field

Call<ResponseBody> call1 = service.testFormUrlEncoded1("怪盗kidou", 24);

ResponseBodyPrinter.printResponseBody(call1);

Map<String, Object> map = new HashMap<>();

map.put("username", "怪盗kidou");

map.put("age", 24);

Call<ResponseBody> call2 = service.testFormUrlEncoded2(map);

ResponseBodyPrinter.printResponseBody(call2);

//--------------------------------------------------------------------------

// 演示 @Multipart 和 @Part

MediaType textType = MediaType.parse("text/plain");

RequestBody name = RequestBody.create(textType, "怪盗kidou");

RequestBody age = RequestBody.create(textType, "24");

RequestBody file = RequestBody.create(MediaType.parse("application/octet-stream"), "这里是模拟文件的内容");

MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", "test.txt", file);

Call<ResponseBody> call3 = service.testFileUpload1(name, age, filePart);

ResponseBodyPrinter.printResponseBody(call3);

Map<String, RequestBody> fileUpload2Args = new HashMap<>();

fileUpload2Args.put("name", name);

fileUpload2Args.put("age", age);

//这里并不会被当成文件，因为没有文件名(包含在Content-Disposition请求头中)，但上面的 filePart 有

//fileUpload2Args.put("file", file);

Call<ResponseBody> call4 = service.testFileUpload2(fileUpload2Args, filePart); //单独处理文件

ResponseBodyPrinter.printResponseBody(call4);

//===================================================

// 还有一种比较hack的方式可以实现文件上传，

// 上面说过被当成文件上传的必要条件就是 Content-Disposition 请求头中必须要有 filename="xxx" 才会被当成文件

// 所有我们在写文件名的时候可以拼把 filename="XXX" 也拼接上去，

// 即文件名变成 表单键名"; filename="文件名 （两端的引号会自动加，所以这里不加）也可以实现，但是不推荐方式

Map<String, RequestBody> fileUpload3Args = new HashMap<>();

fileUpload3Args.put("name",name);

fileUpload3Args.put("age",age);

fileUpload3Args.put("file\"; filename=\"test.txt",file);

Call<ResponseBody> testFileUpload3 = service.testFileUpload3(fileUpload3Args);

ResponseBodyPrinter.printResponseBody(testFileUpload3);

}

}

### Example04

/\*\*

\* [Retrofit注解详解 之 FormUrlEncoded/Field/FieldMap注解]源码

\*/

public class Example04 {

public interface BlogService {

@GET("/headers?showAll=true")

@Headers({"CustomHeader1: customHeaderValue1", "CustomHeader2: customHeaderValue2"})

Call<ResponseBody> testHeader(@Header("CustomHeader3") String customHeaderValue3);

}

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

BlogService service = retrofit.create(BlogService.class);

//演示 @Headers 和 @Header

Call<ResponseBody> call1 = service.testHeader("ikidou");

ResponseBodyPrinter.printResponseBody(call1);

}

}

### Example05

/\*\*

\* [Retrofit注解详解 之 FormUrlEncoded/Field/FieldMap注解]源码

\*/

public class Example05 {

public interface BlogService {

/\*\*

\* 当GET、POST...HTTP等方法中没有设置Url时，则必须使用 {@link Url}提供

\* 对于Query和QueryMap，如果不是String（或Map的第二个泛型参数不是String）时,会被默认会调用toString转换成String类型

\* Url支持的类型有 okhttp3.HttpUrl, String, java.net.URI, android.net.Uri

\* {@link retrofit2.http.QueryMap} 用法和{@link retrofit2.http.FieldMap} 用法一样，不再说明

\*/

@GET //当有URL注解时，这里的URL就省略了

Call<ResponseBody> testUrlAndQuery(@Url String url, @Query("showAll") boolean showAll);

}

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.build();

BlogService service = retrofit.create(BlogService.class);

//演示 @Headers 和 @Header

Call<ResponseBody> call1 = service.testUrlAndQuery("headers",false);

ResponseBodyPrinter.printResponseBody(call1);

}

}

### Example06

/\*\*

\* [Retrofit Converter 反序化]源码

\*/

public class Example06 {

public interface BlogService {

@GET("blog/{id}")

Call<Result<Blog>> getBlog(@Path("id") int id);

}

public static void main(String[] args) {

Gson gson = new GsonBuilder()

//配置你的Gson

.setDateFormat("yyyy-MM-dd hh:mm:ss")

.create();

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

//可以接收自定义的Gson，当然也可以不传

.addConverterFactory(GsonConverterFactory.create(gson))

.build();

BlogService service = retrofit.create(BlogService.class);

Call<Result<Blog>> call = service.getBlog(2);

call.enqueue(new Callback<Result<Blog>>() {

@Override

public void onResponse(Call<Result<Blog>> call, Response<Result<Blog>> response) {

// 已经转换为想要的类型了

Result<Blog> result = response.body();

System.out.println(result);

}

@Override

public void onFailure(Call<Result<Blog>> call, Throwable t) {

t.printStackTrace();

}

});

}

}

### Example07

/\*\*

\* [Retrofit Converter 序列化]源码

\*/

public class Example07 {

public interface BlogService {

@POST("blog")

Call<Result<Blog>> createBlog(@Body Blog blog);

}

public static void main(String[] args) {

Gson gson = new GsonBuilder()

//配置你的Gson

.setDateFormat("yyyy-MM-dd hh:mm:ss")

.create();

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

//可以接收自定义的Gson，当然也可以不传

.addConverterFactory(GsonConverterFactory.create(gson))

.build();

BlogService service = retrofit.create(BlogService.class);

Blog blog = new Blog();

blog.content = "新建的Blog";

blog.title = "测试";

blog.author = "怪盗kidou";

Call<Result<Blog>> call = service.createBlog(blog);

call.enqueue(new Callback<Result<Blog>>() {

@Override

public void onResponse(Call<Result<Blog>> call, Response<Result<Blog>> response) {

// 已经转换为想要的类型了

Result<Blog> result = response.body();

System.out.println(result);

}

@Override

public void onFailure(Call<Result<Blog>> call, Throwable t) {

t.printStackTrace();

}

});

}

}

### Example08

/\*\*

\* [Retrofit CallAdapter rxJava]源码

\*/

public class Example08 {

public interface BlogService {

@GET("/blog")

Observable<Result<List<Blog>>> getBlogs(@Query("page") int page);

/\*

如果需要Header的值，可以把返回值替换为

Observable<Response<Result<List<Blog>>>>

或Observable<retrofit2.adapter.rxjava.Result<Result<List<Blog>>>> //都叫Result，真是失策

\*/

}

public static void main(String[] args) {

Gson gson = new GsonBuilder()

.setDateFormat("yyyy-MM-dd hh:mm:ss")

.create();

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(GsonConverterFactory.create(gson))

.addCallAdapterFactory(RxJavaCallAdapterFactory.create())

.build();

BlogService service = retrofit.create(BlogService.class);

service.getBlogs(1)

.observeOn(Schedulers.io())

.subscribe(new Subscriber<Result<List<Blog>>>() {

@Override

public void onCompleted() {

System.out.println("onCompleted");

}

@Override

public void onError(Throwable e) {

System.err.println("onError");

}

@Override

public void onNext(Result<List<Blog>> blogsResult) {

System.out.println(blogsResult);

}

});

}

}

### Example09

/\*\*

\* [Retrofit 自定义Converter]源码

\*/

public class Example09 {

public interface BlogService {

@GET("blog/{id}")

Call<String> getBlog(@Path("id") int id);

}

/\*\*

\* 自定义Converter实现RequestBody到String的转换

\*/

public static class StringConverter implements Converter<ResponseBody, String> {

public static final StringConverter INSTANCE = new StringConverter();

@Override

public String convert(ResponseBody value) throws IOException {

return value.string();

}

}

/\*\*

\* 用于向Retrofit提供StringConverter

\*/

public static class StringConverterFactory extends Converter.Factory {

public static final StringConverterFactory INSTANCE = new StringConverterFactory();

public static StringConverterFactory create() {

return INSTANCE;

}

// 我们只关实现从ResponseBody 到 String 的转换，所以其它方法可不覆盖

@Override

public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] annotations, Retrofit retrofit) {

if (type == String.class) {

return StringConverter.INSTANCE;

}

//其它类型我们不处理，返回null就行

return null;

}

}

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

// 如是有Gson这类的Converter 一定要放在其它前面

.addConverterFactory(StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.build();

BlogService service = retrofit.create(BlogService.class);

Call<String> call = service.getBlog(2);

call.enqueue(new Callback<String>() {

@Override

public void onResponse(Call<String> call, Response<String> response) {

System.out.println(response.body());

}

@Override

public void onFailure(Call<String> call, Throwable t) {

t.printStackTrace();

}

});

}

}

### Example10

/\*\*

\* [Retrofit 自定义 CallAdapter]源码

\*/

public class Example10 {

public interface BlogService {

@GET("blog/{id}")

CustomCall<String> getBlog(@Path("id") int id);

}

public static class CustomCall<R> {

public final Call<R> call;

public CustomCall(Call<R> call) {

this.call = call;

}

// 提供一个同步获取数据的方法

public R get() throws IOException {

return call.execute().body();

}

}

public static class CustomCallAdapter implements CallAdapter<CustomCall<?>> {

private final Type responseType;

// 下面的 responseType 方法需要数据的类型

CustomCallAdapter(Type responseType) {

this.responseType = responseType;

}

@Override

public Type responseType() {

return responseType;

}

@Override

public <R> CustomCall<R> adapt(Call<R> call) {

// 由 CustomCall 决定如何使用

return new CustomCall<>(call);

}

}

public static class CustomCallAdapterFactory extends CallAdapter.Factory {

public static final CustomCallAdapterFactory INSTANCE = new CustomCallAdapterFactory();

@Override

public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {

// 获取原始类型

Class<?> rawType = getRawType(returnType);

// 返回值必须是CustomCall并且带有泛型

if (rawType == CustomCall.class && returnType instanceof ParameterizedType) {

Type callReturnType = getParameterUpperBound(0, (ParameterizedType) returnType);

return new CustomCallAdapter(callReturnType);

}

return null;

}

}

public static void main(String[] args) {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(Example09.StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(CustomCallAdapterFactory.INSTANCE)

.build();

BlogService service = retrofit.create(BlogService.class);

CustomCall<String> call = service.getBlog(2);

try {

String result = call.get();

System.out.println(result);

} catch (IOException e) {

e.printStackTrace();

}

}

}

## 其它说明

### Retrofit.Builder

前面用到了 Retrofit.Builder 中的baseUrl、addCallAdapterFactory、addConverterFactory、build方法，还有callbackExecutor、callFactory、client、validateEagerly这四个方法没有用到，这里简单的介绍一下。



### Retrofit的Url组合规则



从上面不能难看出以下规则：

* 如果你在注解中提供的url是完整的url，则url将作为请求的url。
* 如果你在注解中提供的url是不完整的url，且不以 / 开头，则请求的url为baseUrl+注解中提供的值
* 如果你在注解中提供的url是不完整的url，且以 / 开头，则请求的url为baseUrl的主机部分+注解中提供的值

### Retrofit提供的Converter

|  |  |
| --- | --- |
| **Converter** | **Gradle依赖** |
| Gson | com.squareup.retrofit2:converter-gson:2.0.2 |
| Jackson | com.squareup.retrofit2:converter-jackson:2.0.2 |
| Moshi | com.squareup.retrofit2:converter-moshi:2.0.2 |
| Protobuf | com.squareup.retrofit2:converter-protobuf:2.0.2 |
| Wire | com.squareup.retrofit2:converter-wire:2.0.2 |
| Simple XML | com.squareup.retrofit2:converter-simplexml:2.0.2 |
| Scalars | com.squareup.retrofit2:converter-scalars:2.0.2 |

### Retrofit提供的CallAdapter

|  |  |
| --- | --- |
| **CallAdapter** | **Gradle依赖** |
| guava | com.squareup.retrofit2:adapter-guava:2.0.2 |
| Java8 | com.squareup.retrofit2:adapter-java8:2.0.2 |
| rxjava | com.squareup.retrofit2:adapter-rxjava:2.0.2 |

# RxJava

## 引入

GitHub 链接和引入依赖的 gradle 代码：

* Github：

https://github.com/ReactiveX/RxJava

https://github.com/ReactiveX/RxAndroid

* 引入依赖：

compile 'io.reactivex:rxjava:1.0.14'

compile 'io.reactivex:rxandroid:1.0.1'

(版本号是文章发布时的最新稳定版)

## RxJava 到底是什么

**一个词：异步。**

RxJava 在 GitHub 主页上的自我介绍是 "a library for composing asynchronous and event-based programs using observable sequences for the Java VM"（一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库）。这就是 RxJava ，概括得非常精准。

然而，对于初学者来说，这太难看懂了。因为它是一个『总结』，而初学者更需要一个『引言』。其实， RxJava 的本质可以压缩为异步这一个词。说到根上，它就是一个实现异步操作的库，而别的定语都是基于这之上的。

## RxJava 好在哪

**一个词：简洁。**

换句话说，『同样是做异步，为什么人们用它，而不用现成的 AsyncTask / Handler / XXX / ... ？』

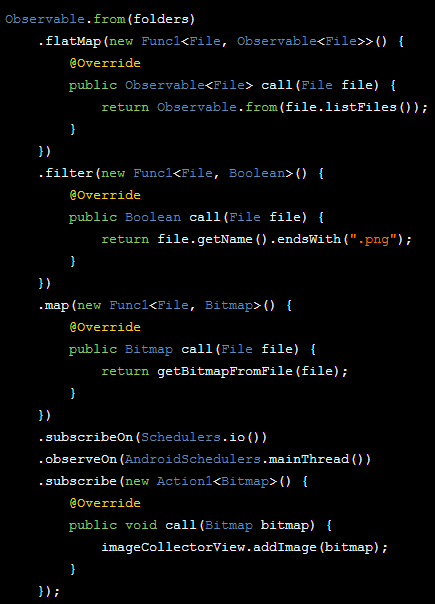
异步操作很关键的一点是程序的简洁性，因为在调度过程比较复杂的情况下，异步代码经常会既难写也难被读懂。 Android 创造的 AsyncTask 和Handler ，其实都是为了让异步代码更加简洁。RxJava 的优势也是简洁，但它的简洁的与众不同之处在于，随着程序逻辑变得越来越复杂，它依然能够保持简洁。

**举个列子**

假设有这样一个需求：界面上有一个自定义的视图 imageCollectorView ，它的作用是显示多张图片，并能使用 addImage(Bitmap) 方法来任意增加显示的图片。现在需要程序将一个给出的目录数组 File[] folders 中每个目录下的 png 图片都加载出来并显示在 imageCollectorView 中。需要注意的是，由于读取图片的这一过程较为耗时，需要放在后台执行，而图片的显示则必须在 UI 线程执行。常用的实现方式有多种，我这里贴出其中一种：



**而如果使用 RxJava ，实现方式是这样的：**



那位说话了：『你这代码明明变多了啊！简洁个毛啊！』大兄弟你消消气，我说的是逻辑的简洁，不是单纯的代码量少（逻辑简洁才是提升读写代码速度的必杀技对不？）。观察一下你会发现， RxJava 的这个实现，是一条从上到下的链式调用，没有任何嵌套，这在逻辑的简洁性上是具有优势的。当需求变得复杂时，这种优势将更加明显（试想如果还要求只选取前 10 张图片，常规方式要怎么办？如果有更多这样那样的要求呢？再试想，在这一大堆需求实现完两个月之后需要改功能，当你翻回这里看到自己当初写下的那一片迷之缩进，你能保证自己将迅速看懂，而不是对着代码重新捋一遍思路？）。

## API 介绍和原理简析

这个我就做不到一个词说明了……因为这一节的主要内容，就是一步步地说明 RxJava 到底怎样做到了异步，怎样做到了简洁。

### 概念：扩展的观察者模式

RxJava 的异步实现，是通过一种扩展的观察者模式来实现的。

**观察者模式**

先简述一下观察者模式，已经熟悉的可以跳过这一段。

观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。举个例子，新闻里喜闻乐见的警察抓小偷，警察需要在小偷伸手作案的时候实施抓捕。在这个例子里，警察是观察者，小偷是被观察者，警察需要时刻盯着小偷的一举一动，才能保证不会漏过任何瞬间。程序的观察者模式和这种真正的『观察』略有不同，观察者不需要时刻盯着被观察者（例如 A 不需要每过 2ms 就检查一次 B 的状态），而是采用注册(Register)或者称为订阅(Subscribe)的方式，告诉被观察者：我需要你的某某状态，你要在它变化的时候通知我。 Android 开发中一个比较典型的例子是点击监听器 OnClickListener 。对设置 OnClickListener 来说， View 是被观察者， OnClickListener 是观察者，二者通过 setOnClickListener() 方法达成订阅关系。订阅之后用户点击按钮的瞬间，Android Framework 就会将点击事件发送给已经注册的 OnClickListener 。采取这样被动的观察方式，既省去了反复检索状态的资源消耗，也能够得到最高的反馈速度。当然，这也得益于我们可以随意定制自己程序中的观察者和被观察者，而警察叔叔明显无法要求小偷『你在作案的时候务必通知我』。

**OnClickListener 的模式大致如下图：**



如图所示，通过 setOnClickListener() 方法，Button 持有 OnClickListener 的引用（这一过程没有在图上画出）；当用户点击时，Button 自动调用 OnClickListener 的 onClick() 方法。另外，如果把这张图中的概念抽象出来（Button -> 被观察者、OnClickListener -> 观察者、setOnClickListener() -> 订阅，onClick() -> 事件），就由专用的观察者模式（例如只用于监听控件点击）转变成了通用的观察者模式。如下图：



而 RxJava 作为一个工具库，使用的就是通用形式的观察者模式。

**RxJava 的观察者模式**

RxJava 有四个基本概念：Observable (可观察者，即被观察者)、 Observer (观察者)、 subscribe (订阅)、事件。

Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()。

* onCompleted(): 事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志。
* onError(): 事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。
* 在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。

RxJava 的观察者模式大致如下图：



### 基本实现

基于以上的概念， RxJava 的基本实现主要有三点：

#### 创建 Observer

Observer 即观察者，它决定事件触发的时候将有怎样的行为。 RxJava 中的 Observer 接口的实现方式：

Observer<String> observer = new Observer<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

};

除了 Observer 接口之外，RxJava 还内置了一个实现了 Observer 的抽象类：Subscriber。 Subscriber 对 Observer 接口进行了一些扩展，但他们的基本使用方式是完全一样的：

Subscriber<String> subscriber = new Subscriber<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

};

不仅基本使用方式一样，实质上，在 RxJava 的 subscribe 过程中，Observer 也总是会先被转换成一个 Subscriber 再使用。所以如果你只想使用基本功能，选择 Observer 和 Subscriber 是完全一样的。它们的区别对于使用者来说主要有两点：

* onStart(): 这是 Subscriber 增加的方法。它会在 subscribe 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对话框，这必须在主线程执行）， onStart() 就不适用了，因为它总是在 subscribe 所发生的线程被调用，而不能指定线程。要在指定的线程来做准备工作，可以使用 doOnSubscribe() 方法，具体可以在后面的文中看到。
* unsubscribe(): 这是 Subscriber 所实现的另一个接口 Subscription 的方法，用于取消订阅。在这个方法被调用后，Subscriber 将不再接收事件。一般在这个方法调用前，可以使用 isUnsubscribed() 先判断一下状态。 unsubscribe() 这个方法很重要，因为在 subscribe() 之后， Observable 会持有 Subscriber 的引用，这个引用如果不能及时被释放，将有内存泄露的风险。所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 onPause() onStop() 等方法中）调用 unsubscribe() 来解除引用关系，以避免内存泄露的发生。

#### 创建 Observable

Observable 即被观察者，它决定什么时候触发事件以及触发怎样的事件。 RxJava 使用 create() 方法来创建一个Observable ，并为它定义事件触发规则：

Observable observable = Observable.create(new Observable.OnSubscribe<String>() {

@Override

public void call(Subscriber<? super String> subscriber) {

subscriber.onNext("Hello");

subscriber.onNext("Hi");

subscriber.onNext("Aloha");

subscriber.onCompleted();

}

});

可以看到，这里传入了一个 OnSubscribe 对象作为参数。OnSubscribe 会被存储在返回的 Observable 对象中，它的作用相当于一个计划表，当 Observable 被订阅的时候，OnSubscribe 的 call() 方法会自动被调用，事件序列就会依照设定依次触发（对于上面的代码，就是观察者Subscriber 将会被调用三次 onNext() 和一次 onCompleted()）。这样，由被观察者调用了观察者的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

create() 方法是 RxJava 最基本的创造事件序列的方法。基于这个方法， RxJava 还提供了一些方法用来快捷创建事件队列，例如：

* just(T...): 将传入的参数依次发送出来。

Observable observable = Observable.just("Hello", "Hi", "Aloha");

// 将会依次调用：

// onNext("Hello");

// onNext("Hi");

// onNext("Aloha");

// onCompleted();

* from(T[]) / from(Iterable<? extends T>) : 将传入的数组或 Iterable 拆分成具体对象后，依次发送出来。

String[] words = {"Hello", "Hi", "Aloha"};

Observable observable = Observable.from(words);

// 将会依次调用：

// onNext("Hello");

// onNext("Hi");

// onNext("Aloha");

// onCompleted();

上面 just(T...) 的例子和 from(T[]) 的例子，都和之前的 create(OnSubscribe) 的例子是等价的。

#### Subscribe (订阅)

创建了 Observable 和 Observer 之后，再用 subscribe() 方法将它们联结起来，整条链子就可以工作了。代码形式很简单：

observable.subscribe(observer);

// 或者：

observable.subscribe(subscriber);

有人可能会注意到， subscribe() 这个方法有点怪：它看起来是『observalbe 订阅了 observer / subscriber』而不是『observer / subscriber 订阅了 observalbe』，这看起来就像『杂志订阅了读者』一样颠倒了对象关系。这让人读起来有点别扭，不过如果把 API 设计成 observer.subscribe(observable) / subscriber.subscribe(observable) ，虽然更加符合思维逻辑，但对流式 API 的设计就造成影响了，比较起来明显是得不偿失的。

Observable.subscribe(Subscriber) 的内部实现是这样的（仅核心代码）：

// 注意：这不是 subscribe() 的源码，而是将源码中与性能、兼容性、扩展性有关的代码剔除后的核心代码。

// 如果需要看源码，可以去 RxJava 的 GitHub 仓库下载。

public Subscription subscribe(Subscriber subscriber) {

subscriber.onStart();

onSubscribe.call(subscriber);

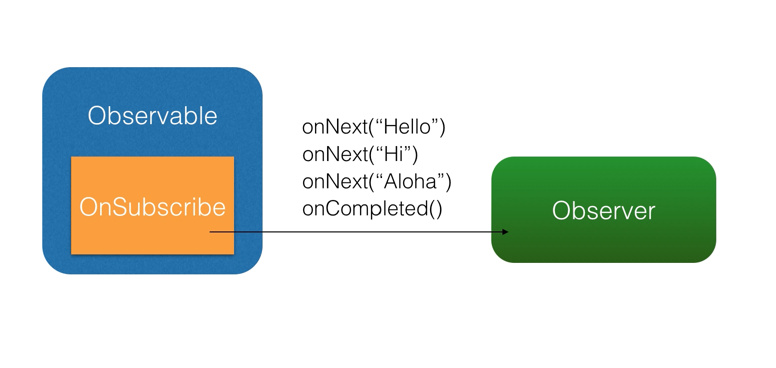
return subscriber;

}

可以看到，subscriber() 做了3件事：

* 调用 Subscriber.onStart() 。这个方法在前面已经介绍过，是一个可选的准备方法。
* 调用 Observable 中的 OnSubscribe.call(Subscriber) 。在这里，事件发送的逻辑开始运行。从这也可以看出，在 RxJava 中， Observable 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，即当 subscribe() 方法执行的时候。
* 将传入的 Subscriber 作为 Subscription 返回。这是为了方便 unsubscribe().

整个过程中对象间的关系如下图：



#### subscribe() 还支持不完整定义的回调

除了 subscribe(Observer) 和 subscribe(Subscriber)，.subscribe() 还支持不完整定义的回调RxJava 会自动根据定义创建出 Subscriber 。形式如下：

Action1<String> onNextAction = new Action1<String>() {

// onNext()

@Override

public void call(String s) {

Log.d(tag, s);

}

};

Action1<Throwable> onErrorAction = new Action1<Throwable>() {

// onError()

@Override

public void call(Throwable throwable) {

// Error handling

}

};

Action0 onCompletedAction = new Action0() {

// onCompleted()

@Override

public void call() {

Log.d(tag, "completed");

}

};

-------------------------------------------------------------------------------------

* 自动创建 Subscriber ，并使用 onNextAction 来定义 onNext()

observable.subscribe(onNextAction);

* 自动创建 Subscriber ，并使用 onNextAction 和 onErrorAction 来定义 onNext() 和 onError()

observable.subscribe(onNextAction, onErrorAction);

* 自动创建 Subscriber ，并使用 onNextAction、 onErrorAction 和 onCompletedAction 来定义 onNext()、 onError() 和 onCompleted()

observable.subscribe(onNextAction, onErrorAction, onCompletedAction);

简单解释一下这段代码中出现的 Action1 和 Action0。 Action0 是 RxJava 的一个接口，它只有一个方法 call()，这个方法是无参无返回值的；由于 onCompleted() 方法也是无参无返回值的，因此 Action0 可以被当成一个包装对象，将 onCompleted() 的内容打包起来将自己作为一个参数传入 subscribe() 以实现不完整定义的回调。这样其实也可以看做将 onCompleted() 方法作为参数传进了 subscribe()，相当于其他某些语言中的『闭包』。 Action1 也是一个接口，它同样只有一个方法 call(T param)，这个方法也无返回值，但有一个参数；与 Action0 同理，由于 onNext(T obj) 和 onError(Throwable error) 也是单参数无返回值的，因此 Action1 可以将 onNext(obj) 和 onError(error) 打包起来传入 subscribe() 以实现不完整定义的回调。事实上，虽然 Action0 和 Action1 在 API 中使用最广泛，但 RxJava 是提供了多个 ActionX 形式的接口 (例如 Action2, Action3) 的，它们可以被用以包装不同的无返回值的方法（0123表示，0个、1个、2个，3个参数）。

注：正如前面所提到的，Observer 和 Subscriber 具有相同的角色，而且 Observer 在 subscribe() 过程中最终会被转换成 Subscriber 对象，因此，从这里开始，后面的描述我将用 Subscriber 来代替 Observer ，这样更加严谨。

#### 场景示例

下面举两个例子：

为了把原理用更清晰的方式表述出来，本文中挑选的都是功能尽可能简单的例子，以至于有些示例代码看起来会有『画蛇添足』『明明不用 RxJava 可以更简便地解决问题』的感觉。当你看到这种情况，不要觉得是因为 RxJava 太啰嗦，而是因为在过早的时候举出真实场景的例子并不利于原理的解析，因此我刻意挑选了简单的情景。

##### 打印字符串数组

将字符串数组 names 中的所有字符串依次打印出来：

String[] names = ...;

Observable.from(names)

.subscribe(new Action1<String>() {

@Override

public void call(String name) {

Log.d(tag, name);

}

});

##### 由 id 取得图片并显示

由指定的一个 drawable 文件 id drawableRes 取得图片，并显示在 ImageView 中，并在出现异常的时候打印 Toast 报错：

int drawableRes = ...;

ImageView imageView = ...;

Observable.create(new OnSubscribe<Drawable>() {

@Override

public void call(Subscriber<? super Drawable> subscriber) {

Drawable drawable = getTheme().getDrawable(drawableRes));

subscriber.onNext(drawable);

subscriber.onCompleted();

}

}).subscribe(new Observer<Drawable>() {

@Override

public void onNext(Drawable drawable) {

imageView.setImageDrawable(drawable);

}

@Override

public void onCompleted() {

Toast.makeText(activity, "Completed!", Toast.LENGTH\_SHORT).show();

}

@Override

public void onError(Throwable e) {

Toast.makeText(activity, "Error!", Toast.LENGTH\_SHORT).show();

}

});

在 RxJava 的默认规则中，事件的发出和消费都是在同一个线程的。也就是说，如果只用上面的方法，实现出来的只是一个同步的观察者模式。观察者模式本身的目的就是『后台处理，前台回调』的异步机制，因此异步对于 RxJava 是至关重要的。而要实现异步，则需要用到 RxJava 的另一个概念： Scheduler 。

### 线程控制 —— Scheduler (一)

在不指定线程的情况下， RxJava 遵循的是线程不变的原则，即：

* 在哪个线程调用 subscribe()，就在哪个线程生产事件；
* 在哪个线程生产事件，就在哪个线程消费事件。
* 如果需要切换线程，就需要用到 Scheduler （调度器）。

#### Scheduler 的 API (一)

在RxJava 中，Scheduler ——调度器，相当于线程控制器，RxJava 通过它来指定每一段代码应该运行在什么样的线程。RxJava 已经内置了几个 Scheduler ，它们已经适合大多数的使用场景：

* 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。
* Schedulers.newThread(): 总是启用新线程，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。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。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。
* 另外， Android 还有一个专用的 AndroidSchedulers.mainThread()，它指定的操作将在 Android 主线程运行。

有了这几个 Scheduler ，就可以使用 subscribeOn() 和 observeOn() 两个方法来对线程进行控制了。

* subscribeOn(): 指定 subscribe() 所发生的线程，即 Observable.OnSubscribe.call()方法被激活调用时所处的线程。或者叫做事件产生的线程。
* observeOn(): 指定 Subscriber 所运行在的线程。或者叫做事件消费的线程。

文字叙述总归难理解，上代码：

Observable.just(1, 2, 3, 4)

.subscribeOn(Schedulers.io()) // 指定 subscribe() 发生在 IO 线程

.observeOn(AndroidSchedulers.mainThread()) // 指定 Subscriber 的回调发生在主线程

.subscribe(new Action1<Integer>() {

@Override

public void call(Integer number) {

Log.d(tag, "number:" + number);

}

});

上面这段代码中，由于 subscribeOn(Schedulers.io()) 的指定，被创建的事件的内容 1、2、3、4 将会在 IO 线程发出；而由于 observeOn(AndroidScheculers.mainThread()) 的指定，因此 subscriber 数字的打印将发生在主线程 。事实上，这种在 subscribe() 之前写上两句 subscribeOn(Scheduler.io()) 和 observeOn(AndroidSchedulers.mainThread()) 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。

而前面提到的由图片 id 取得图片并显示的例子，如果也加上这两句：

int drawableRes = ...;

ImageView imageView = ...;

Observable.create(new OnSubscribe<Drawable>() {

@Override

public void call(Subscriber<? super Drawable> subscriber) {

Drawable drawable = getTheme().getDrawable(drawableRes));

subscriber.onNext(drawable);

subscriber.onCompleted();

}

})

.subscribeOn(Schedulers.io()) // 指定 subscribe() 发生在 IO 线程

.observeOn(AndroidSchedulers.mainThread()) // 指定 Subscriber 的回调发生在主线程

.subscribe(new Observer<Drawable>() {

@Override

public void onNext(Drawable drawable) {

imageView.setImageDrawable(drawable);

。。。。。。。。。。。。。。。。。

那么，加载图片将会发生在 IO 线程，而设置图片则被设定在了主线程。这就意味着，即使加载图片耗费了几十甚至几百毫秒的时间，也不会造成丝毫界面的卡顿。

#### Scheduler 的原理 (一)

RxJava 的 Scheduler API 很方便，也很神奇（加了一句话就把线程切换了，怎么做到的？而且 subscribe() 不是最外层直接调用的方法吗，它竟然也能被指定线程？）。然而 Scheduler 的原理需要放在后面讲，因为它的原理是以下一节《变换》的原理作为基础的。

好吧这一节其实我屁也没说，只是为了让你安心，让你知道我不是忘了讲原理，而是把它放在了更合适的地方。

### 变换

终于要到牛逼的地方了，不管你激动不激动，反正我是激动了。

RxJava 提供了对事件序列进行变换的支持，这是它的核心功能之一，也是大多数人说『RxJava 真是太好用了』的最大原因。所谓变换，就是将事件序列中的对象或整个序列进行加工处理，转换成不同的事件或事件序列。概念说着总是模糊难懂的，来看 API。

#### API

##### map()

map(): 事件对象的直接变换，具体功能上面已经介绍过。它是 RxJava 最常用的变换。

Observable.just("images/logo.png") // 输入类型 String

.map(new Func1<String, Bitmap>() {

@Override

public Bitmap call(String filePath) { // 参数类型 String

return getBitmapFromPath(filePath); // 返回类型 Bitmap

}

})

.subscribe(new Action1<Bitmap>() {

@Override

public void call(Bitmap bitmap) { // 参数类型 Bitmap

showBitmap(bitmap);

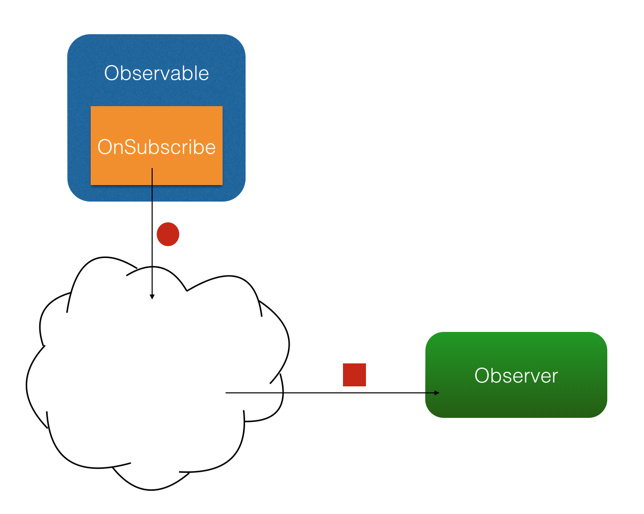
}

});

这里出现了一个叫做 Func1 的类。它和 Action1 非常相似，也是 RxJava 的一个接口，用于包装含有一个参数的方法。 Func1 和 Action 的区别在于， Func1 包装的是有返回值的方法。另外，和 ActionX 一样， FuncX 也有多个，用于不同参数个数的方法。FuncX 和 ActionX 的区别在 FuncX 包装的是有返回值的方法。

可以看到，map() 方法将参数中的 String 对象转换成一个 Bitmap 对象后返回，而在经过 map() 方法后，事件的参数类型也由 String 转为了 Bitmap。这种直接变换对象并返回的，是最常见的也最容易理解的变换。不过 RxJava 的变换远不止这样，它不仅可以针对事件对象，还可以针对整个事件队列，这使得 RxJava 变得非常灵活

map() 的示意图：



##### flatMap()

flatMap(): 这是一个很有用但非常难理解的变换，因此我决定花多些篇幅来介绍它。 首先假设这么一种需求：假设有一个数据结构『学生』，现在需要打印出一组学生的名字。实现方式很简单：

Student[] students = ...;

Subscriber<String> subscriber = new Subscriber<String>() {

@Override

public void onNext(String name) {

Log.d(tag, name);

}

...

};

Observable.from(students)

.map(new Func1<Student, String>() {

@Override

public String call(Student student) {

return student.getName();

}

})

.subscribe(subscriber);

很简单。那么再假设：如果要打印出每个学生所需要修的所有课程的名称呢？（需求的区别在于，每个学生只有一个名字，但却有多个课程。）首先可以这样实现：

Student[] students = ...;

Subscriber<Student> subscriber = new Subscriber<Student>() {

@Override

public void onNext(Student student) {

List<Course> courses = student.getCourses();

for (int i = 0; i < courses.size(); i++) {

Course course = courses.get(i);

Log.d(tag, course.getName());

}

}

...

};

Observable.from(students)

.subscribe(subscriber);

依然很简单。那么如果我不想在 Subscriber 中使用 for 循环，而是希望 Subscriber 中直接传入单个的 Course 对象呢（这对于代码复用很重要）？用 map() 显然是不行的，因为 map() 是一对一的转化，而我现在的要求是一对多的转化。那怎么才能把一个 Student 转化成多个 Course 呢？

这个时候，就需要用 flatMap() 了：

Student[] students = ...;

Subscriber<Course> subscriber = new Subscriber<Course>() {

@Override

public void onNext(Course course) {

Log.d(tag, course.getName());

}

...

};

Observable.from(students)

.flatMap(new Func1<Student, Observable<Course>>() {

@Override

public Observable<Course> call(Student student) {

return Observable.from(student.getCourses());

}

})

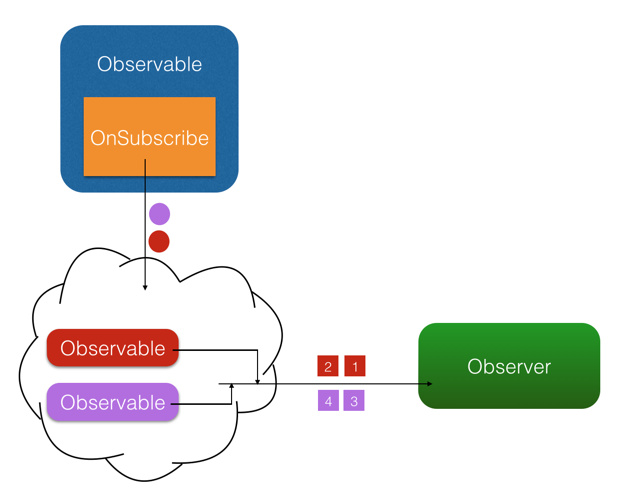
.subscribe(subscriber);

从上面的代码可以看出， flatMap() 和 map() 有一个相同点：它也是把传入的参数转化之后返回另一个对象。但需要注意，和 map() 不同的是， flatMap() 中返回的是个 Observable 对象，并且这个 Observable 对象并不是被直接发送到了 Subscriber 的回调方法中。

flatMap() 的原理是这样的：

1. 使用传入的事件对象创建一个 Observable 对象；
2. 并不发送这个 Observable, 而是将它激活，于是它开始发送事件；
3. 每一个创建出来的 Observable 发送的事件，都被汇入同一个 Observable ，而这个 Observable 负责将这些事件统一交给 Subscriber 的回调方法。

这三个步骤，把事件拆成了两级，通过一组新创建的 Observable 将初始的对象『铺平』之后通过统一路径分发了下去。而这个『铺平』就是 flatMap() 所谓的 flat。flatMap() 示意图：



#### 变换的原理：lift()

这些变换虽然功能各有不同，但实质上都是针对事件序列的处理和再发送。而在 RxJava 的内部，它们是基于同一个基础的变换方法： lift(Operator)。首先看一下 lift() 的内部实现（仅核心代码）：

// 注意：这不是 lift() 的源码，而是将源码中与性能、兼容性、扩展性有关的代码剔除后的核心代码。

// 如果需要看源码，可以去 RxJava 的 GitHub 仓库下载。

public <R> Observable<R> lift(Operator<? extends R, ? super T> operator) {

return Observable.create(new OnSubscribe<R>() {

@Override

public void call(Subscriber subscriber) {

Subscriber newSubscriber = operator.call(subscriber);

newSubscriber.onStart();

onSubscribe.call(newSubscriber);

}

});

}

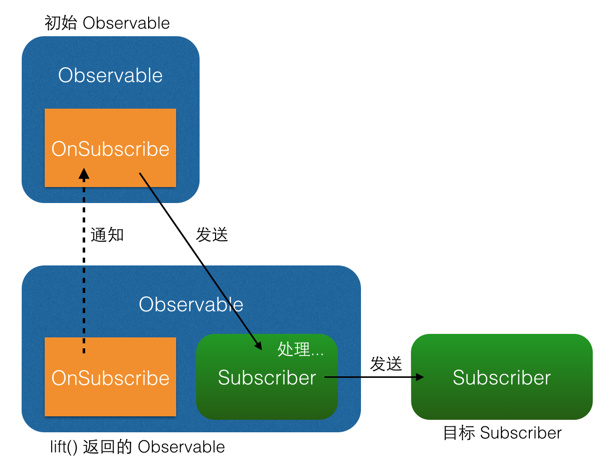
这段代码很有意思：它生成了一个新的 Observable 并返回，而且创建新 Observable 所用的参数 OnSubscribe 的回调方法 call() 中的实现竟然看起来和前面讲过的 Observable.subscribe() 一样！然而它们并不一样哟！不一样的地方关键就在于第二行 onSubscribe.call(subscriber) 中的 onSubscribe 所指代的对象不同（高能预警：接下来的几句话可能会导致身体的严重不适）

subscribe() 中这句话的 onSubscribe 指的是 Observable 中的 onSubscribe 对象，这个没有问题，但是 lift() 之后的情况就复杂了点。

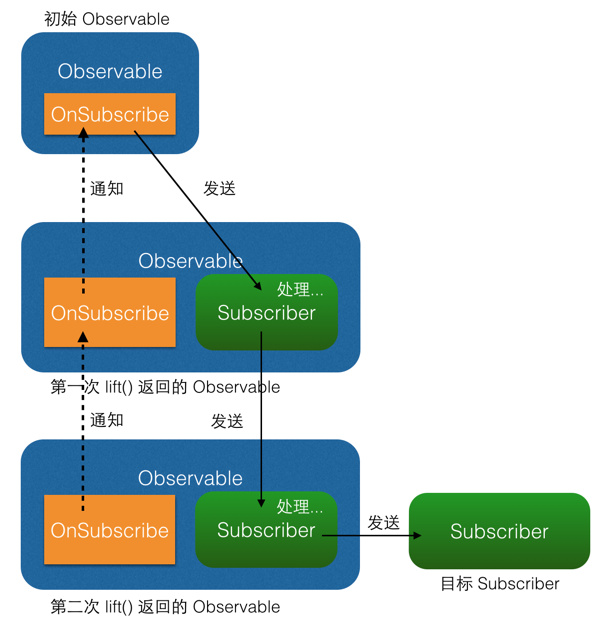
* 当含有 lift() 时：

1. lift() 创建了一个 Observable 后，加上之前的原始 Observable，已经有两个 Observable 了；
2. 而同样地，新 Observable 里的新 OnSubscribe 加上之前的原始 Observable 中的原始 OnSubscribe，也就有了两个 OnSubscribe；
3. 当用户调用经过 lift() 后的 Observable 的 subscribe() 的时候，使用的是 lift() 所返回的新的 Observable ，于是它所触发的 onSubscribe.call(subscriber)，也是用的新 Observable 中的新 OnSubscribe，即在 lift() 中生成的那个 OnSubscribe；
4. 而这个新 OnSubscribe 的 call() 方法中的 onSubscribe ，就是指的原始 Observable 中的原始 OnSubscribe ，在这个 call() 方法里，新 OnSubscribe 利用 operator.call(subscriber) 生成了一个新的 Subscriber（Operator 就是在这里，通过自己的 call() 方法将新 Subscriber 和原始 Subscriber 进行关联，并插入自己的『变换』代码以实现变换），然后利用这个新 Subscriber 向原始 Observable 进行订阅。
5. 这样就实现了 lift() 过程，有点像一种代理机制，通过事件拦截和处理实现事件序列的变换。

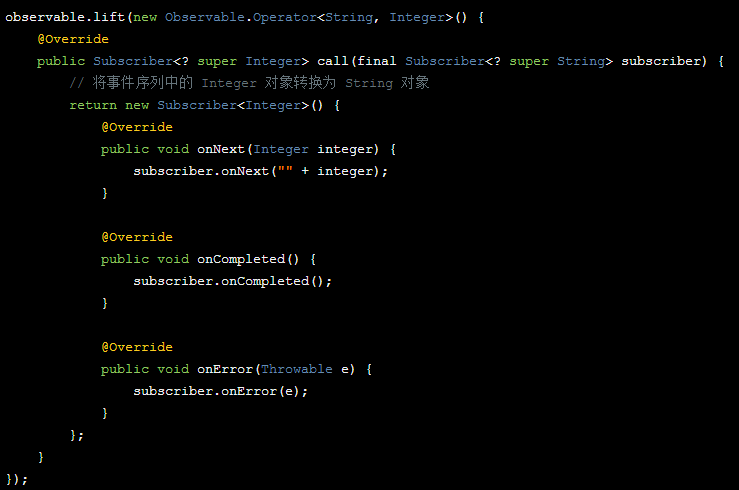
* 精简掉细节的话，也可以这么说：在 Observable 执行了 lift(Operator) 方法之后，会返回一个新的 Observable，这个新的 Observable 会像一个代理一样，负责接收原始的 Observable 发出的事件，并在处理后发送给 Subscriber。
* 如果你更喜欢具象思维，可以看图：



* 两次和多次的 lift() 同理，如下图：

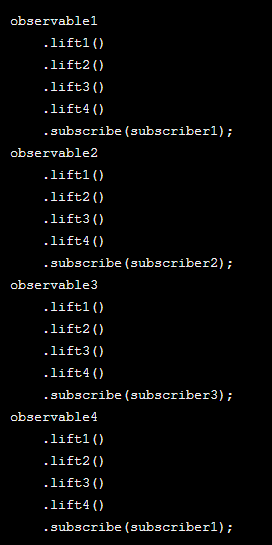


* 举一个具体的 Operator 的实现。下面这是一个将事件中的 Integer 对象转换成 String 的例子，仅供参考：

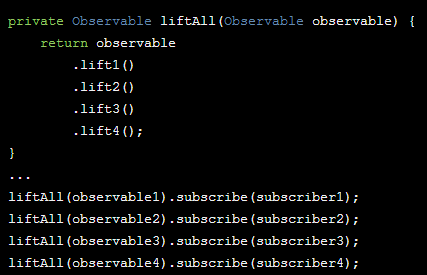


#### compose: 对 Observable 整体的变换

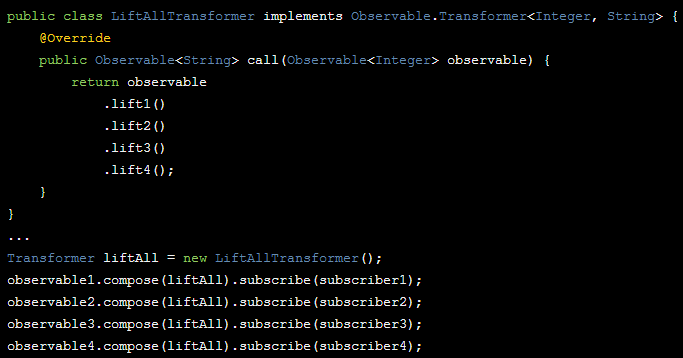
除了 lift() 之外， Observable 还有一个变换方法叫做 compose(Transformer)。它和 lift() 的区别在于，lift() 是针对事件项和事件序列的，而 compose() 是针对 Observable 自身进行变换。举个例子，假设在程序中有多个 Observable ，并且他们都需要应用一组相同的 lift() 变换。你可以这么写：



你觉得这样太不软件工程了，于是你改成了这样：



可读性、可维护性都提高了。可是 Observable 被一个方法包起来，这种方式对于 Observale 的灵活性似乎还是增添了那么点限制。怎么办？这个时候，就应该用 compose() 来解决了



像上面这样，使用 compose() 方法，Observable 可以利用传入的 Transformer 对象的 call 方法直接对自身进行处理，也就不必被包在方法的里面了。

### 线程控制：Scheduler (二)

除了灵活的变换，RxJava 另一个牛逼的地方，就是线程的自由控制。

#### Scheduler 的 API (二)

前面讲到了，可以利用 subscribeOn() 结合 observeOn() 来实现线程控制，让事件的产生和消费发生在不同的线程。可是在了解了 map() flatMap() 等变换方法后，有些好事的（其实就是当初刚接触 RxJava 时的我）就问了：能不能多切换几次线程？

答案是：能。因为 observeOn() 指定的是 Subscriber 的线程，而这个 Subscriber 并不是（严格说应该为『不一定是』，但这里不妨理解为『不是』）subscribe() 参数中的 Subscriber ，而是 observeOn() 执行时的当前 Observable 所对应的 Subscriber ，即它的直接下级 Subscriber 。换句话说，observeOn() 指定的是它之后的操作所在的线程。因此如果有多次切换线程的需求，只要在每个想要切换线程的位置调用一次 observeOn() 即可。上代码：

Observable.just(1, 2, 3, 4) // IO 线程，由 subscribeOn() 指定

.subscribeOn(Schedulers.io())

.observeOn(Schedulers.newThread())

.map(mapOperator) // 新线程，由 observeOn() 指定

.observeOn(Schedulers.io())

.map(mapOperator2) // IO 线程，由 observeOn() 指定

.observeOn(AndroidSchedulers.mainThread)

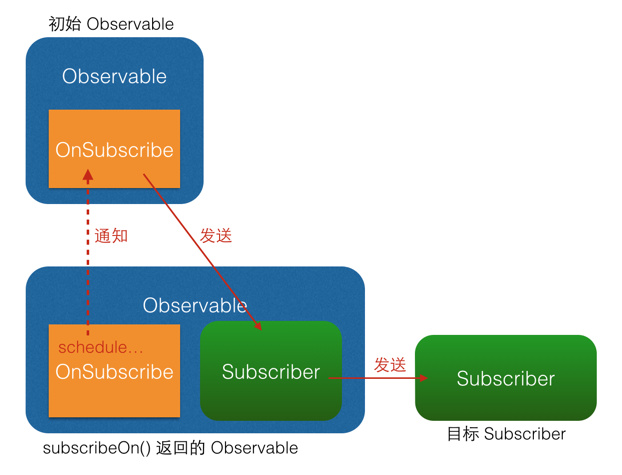
.subscribe(subscriber); // Android 主线程，由 observeOn() 指定

如上，通过 observeOn() 的多次调用，程序实现了线程的多次切换。不过，不同于 observeOn() ， subscribeOn() 的位置放在哪里都可以，但它是只能调用一次的。又有好事的（其实还是当初的我）问了：如果我非要调用多次 subscribeOn() 呢？会有什么效果？这个问题先放着，我们还是从 RxJava 线程控制的原理说起吧。

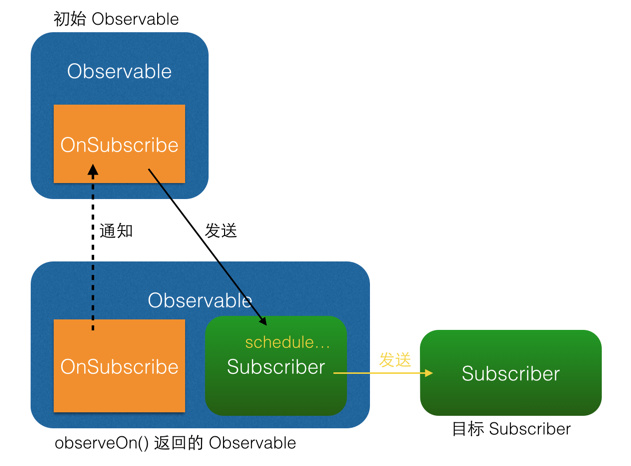
#### Scheduler 的原理（二）

其实， subscribeOn() 和 observeOn() 的内部实现，也是用的 lift()。

* subscribeOn() 原理图：（不同颜色的箭头表示不同的线程）

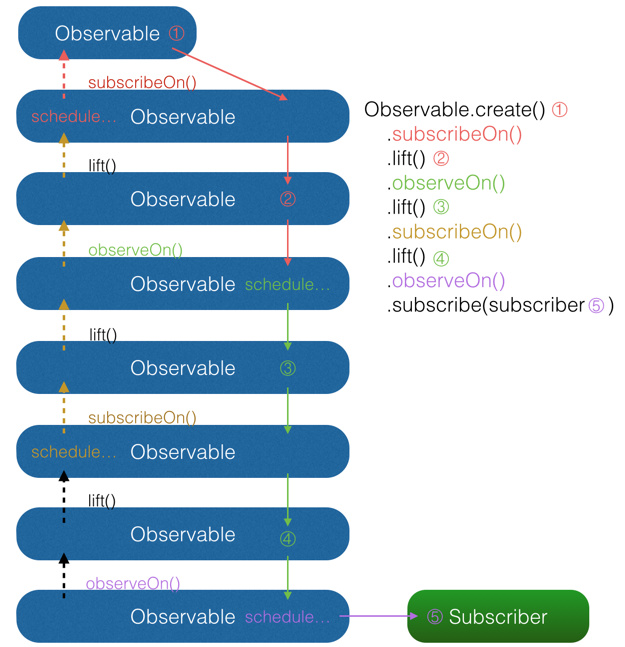


* observeOn() 原理图：



从图中可以看出，subscribeOn() 和 observeOn() 都做了线程切换的工作（图中的 "schedule..." 部位）。不同的是， subscribeOn() 的线程切换发生在 OnSubscribe 中，即在它通知上一级 OnSubscribe 时，这时事件还没有开始发送，因此 subscribeOn() 的线程控制可以从事件发出的开端就造成影响；而 observeOn() 的线程切换则发生在它内建的 Subscriber 中，即发生在它即将给下一级 Subscriber 发送事件时，因此 observeOn() 控制的是它后面的线程。

最后，我用一张图来解释当多个 subscribeOn() 和 observeOn() 混合使用时，线程调度是怎么发生的（由于图中对象较多，相对于上面的图对结构做了一些简化调整）：



图中共有 5 处含有对事件的操作。由图中可以看出，①和②两处受第一个 subscribeOn() 影响，运行在红色线程；③和④处受第一个 observeOn() 的影响，运行在绿色线程；⑤处受第二个 onserveOn() 影响，运行在紫色线程；而第二个 subscribeOn() ，由于在通知过程中线程就被第一个 subscribeOn() 截断，因此对整个流程并没有任何影响。这里也就回答了前面的问题：当使用了多个 subscribeOn() 的时候，只有第一个 subscribeOn() 起作用。

#### doOnSubscribe()

然而，虽然超过一个的 subscribeOn() 对事件处理的流程没有影响，但在流程之前却是可以利用的。在前面讲 Subscriber 的时候，提到过 Subscriber 的 onStart() 可以用作流程开始前的初始化。然而 onStart() 由于在 subscribe() 发生时就被调用了，因此不能指定线程，而是只能执行在 subscribe() 被调用时的线程。这就导致如果 onStart() 中含有对线程有要求的代码（例如在界面上显示一个 ProgressBar，这必须在主线程执行），将会有线程非法的风险，因为有时你无法预测 subscribe() 将会在什么线程执行。

而与Subscriber.onStart()相对应的，有一个方法 Observable.doOnSubscribe() 。它和Subscriber.onStart()同样是在 subscribe()调用后而且在事件发送前执行，但区别在于它可以指定线程。默认情况下，doOnSubscribe() 执行在 subscribe() 发生的线程；而如果在 doOnSubscribe() 之后有 subscribeOn() 的话，它将执行在离它最近的 subscribeOn() 所指定的线程。

示例代码：

Observable.create(onSubscribe)

.subscribeOn(Schedulers.io())

.doOnSubscribe(new Action0() {

@Override

public void call() {

progressBar.setVisibility(View.VISIBLE); // 需要在主线程执行

}

})

.subscribeOn(AndroidSchedulers.mainThread()) // 指定主线程

.observeOn(AndroidSchedulers.mainThread())

.subscribe(subscriber);

如上，在 doOnSubscribe()的后面跟一个 subscribeOn() ，就能指定准备工作的线程了。

## CompositeSubscription、addSubscrebe关于控制生命周期

调用的 Observable.subscribe() 的返回值是一个 Subscription 对象。Subscription 类只有两个方法，unsubscribe() 和 isUnsubscribed()。为了防止可能的内存泄露，在你的 Activity 或 Fragment 的 onDestroy 里，用 Subscription.isUnsubscribed() 检查你的 Subscription 是否是 unsubscribed。一个很常见的模式就是使用CompositeSubscription来持有所有的Subscriptions，然后在onDestroy()或者onDestroyView()里取消所有的订阅。你可以在Activity/Fragment的基类里创建一个CompositeSubscription对象，在子类中使用它。注意! 一旦你调用了 CompositeSubscription.unsubscribe()，这个CompositeSubscription对象就不可用了, 如果你还想使用CompositeSubscription，就必须在创建一个新的对象了。

**RxPresenter**

public class RxPresenter<T extends BaseView> implements BasePresenter<T> {

protected T mView;

protected CompositeSubscription mCompositeSubscription;

protected void unSubscribe() {

if (mCompositeSubscription != null) {

mCompositeSubscription.unsubscribe();

}

}

protected void removeSubscribe(Subscription subscription) {

if (mCompositeSubscription != null) {

mCompositeSubscription.remove(subscription);

}

}

protected void addSubscrebe(Subscription subscription, boolean progress) {

if (progress) {

if (mCompositeSubscription == null) {

mCompositeSubscription = new CompositeSubscription();

}

mCompositeSubscription.add(subscription);

}

}

protected void addSubscrebe(Subscription subscription) {

if (mCompositeSubscription == null) {

mCompositeSubscription = new CompositeSubscription();

}

mCompositeSubscription.add(subscription);

}

protected <U> void addRxBusSubscribe(Class<U> eventType, Action1<U> act) {

if (mCompositeSubscription == null) {

mCompositeSubscription = new CompositeSubscription();

}

mCompositeSubscription.add(RxBus.getDefault().toDefaultObservable(eventType, act));

}

protected <U> void removeRxBusSubscribe(Class<U> eventType, Action1<U> act) {

if (mCompositeSubscription != null) {

mCompositeSubscription.remove(RxBus.getDefault().toDefaultObservable(eventType, act));

}

}

@Override

public void attachView(T view) {

this.mView = view;

}

@Override

public void detachView() {

this.mView = null;

unSubscribe();

}

}

**BaseActivity中**

public class BaseActivity<T extends BasePresenter>{

@Override

protected void onDestroy() {

super.onDestroy();

if (mPresenter != null){

mPresenter.detachView();

}

}

}

## Retrofit 与RxJava的结合

Retrofit 是 Square 的一个著名的网络请求库。没有用过 Retrofit 的可以选择跳过这一小节也没关系，我举的每种场景都只是个例子，而且例子之间并无前后关联，只是个抛砖引玉的作用，所以你跳过这里看别的场景也可以的。

Retrofit 除了提供了传统的 Callback 形式的 API，还有 RxJava 版本的 Observable 形式 API。下面我用对比的方式来介绍 Retrofit 的 RxJava 版 API 和传统版本的区别。

以获取一个 User 对象的接口作为例子。使用Retrofit 的传统 API，你可以用这样的方式来定义请求：

@GET("/user")

public void getUser(@Query("userId") String userId, Callback<User> callback);

在程序的构建过程中， Retrofit 会把自动把方法实现并生成代码，然后开发者就可以利用下面的方法来获取特定用户并处理响应：

getUser(userId, new Callback<User>() {

@Override

public void success(User user) {

userView.setUser(user);

}

@Override

public void failure(RetrofitError error) {

// Error handling

...

}

};

而使用 RxJava 形式的 API，定义同样的请求是这样的：

@GET("/user")

public Observable<User> getUser(@Query("userId") String userId);

使用的时候是这样的：

getUser(userId)

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Observer<User>() {

@Override

public void onNext(User user) {

userView.setUser(user);

}

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable error) {

// Error handling

...

}

});

看到区别了吗？当 RxJava 形式的时候，Retrofit 把请求封装进 Observable ，在请求结束后调用 onNext() 或在请求失败后调用 onError()。

对比来看， Callback 形式和 Observable 形式长得不太一样，但本质都差不多，而且在细节上 Observable 形式似乎还比 Callback 形式要差点。那 Retrofit 为什么还要提供 RxJava 的支持呢？

因为它好用啊！从这个例子看不出来是因为这只是最简单的情况。而一旦情景复杂起来， Callback 形式马上就会开始让人头疼。比如：

假设这么一种情况：你的程序取到的 User 并不应该直接显示，而是需要先与数据库中的数据进行比对和修正后再显示。使用 Callback 方式大概可以这么写：

getUser(userId, new Callback<User>() {

@Override

public void success(User user) {

processUser(user); // 尝试修正 User 数据

userView.setUser(user);

}

@Override

public void failure(RetrofitError error) {

// Error handling

...

}

};

有问题吗？很简便，但不要这样做。为什么？因为这样做会影响性能。数据库的操作很重，一次读写操作花费 10~20ms 是很常见的，这样的耗时很容易造成界面的卡顿。所以通常情况下，如果可以的话一定要避免在主线程中处理数据库。所以为了提升性能，这段代码可以优化一下：

getUser(userId, new Callback<User>() {

@Override

public void success(User user) {

new Thread() {

@Override

public void run() {

processUser(user); // 尝试修正 User 数据

runOnUiThread(new Runnable() { // 切回 UI 线程

@Override

public void run() {

userView.setUser(user);

}

});

}).start();

}

@Override

public void failure(RetrofitError error) {

// Error handling

...

}

};

性能问题解决，但……这代码实在是太乱了，迷之缩进啊！杂乱的代码往往不仅仅是美观问题，因为代码越乱往往就越难读懂，而如果项目中充斥着杂乱的代码，无疑会降低代码的可读性，造成团队开发效率的降低和出错率的升高。

这时候，如果用 RxJava 的形式，就好办多了。 RxJava 形式的代码是这样的：

getUser(userId)

.doOnNext(new Action1<User>() {

@Override

public void call(User user) {

processUser(user);

})

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Observer<User>() {

@Override

public void onNext(User user) {

userView.setUser(user);

}

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable error) {

// Error handling

...

}

});

后台代码和前台代码全都写在一条链中，明显清晰了很多。

再举一个例子：假设 /user 接口并不能直接访问，而需要填入一个在线获取的 token ，代码应该怎么写？Callback 方式，可以使用嵌套的 Callback：

@GET("/token")

public void getToken(Callback<String> callback);

@GET("/user")

public void getUser(@Query("token") String token, @Query("userId") String userId, Callback<User> callback);

...

getToken(new Callback<String>() {

@Override

public void success(String token) {

getUser(token, userId, new Callback<User>() {

@Override

public void success(User user) {

userView.setUser(user);

}

@Override

public void failure(RetrofitError error) {

// Error handling

...

}

};

}

@Override

public void failure(RetrofitError error) {

// Error handling

...

}

});

倒是没有什么性能问题，可是迷之缩进毁一生，你懂我也懂，做过大项目的人应该更懂。而使用 RxJava 的话，代码是这样的：

@GET("/token")

public Observable<String> getToken();

@GET("/user")

public Observable<User> getUser(@Query("token") String token, @Query("userId") String userId);

...

getToken()

.flatMap(new Func1<String, Observable<User>>() {

@Override

public Observable<User> onNext(String token) {

return getUser(token, userId);

})

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Observer<User>() {

@Override

public void onNext(User user) {

userView.setUser(user);

}

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable error) {

// Error handling

...

}

});

用一个 flatMap() 就搞定了逻辑，依然是一条链。看着就很爽，是吧？好，Retrofit 部分就到这里。

## RxBinding

RxBinding 是 Jake Wharton 的一个开源库，它提供了一套在 Android 平台上的基于 RxJava 的 Binding API。所谓 Binding，就是类似设置 OnClickListener 、设置 TextWatcher 这样的注册绑定对象的 API。

举个设置点击监听的例子。使用 RxBinding ，可以把事件监听用这样的方法来设置：

Button button = ...;

RxView.clickEvents(button) // 以 Observable 形式来反馈点击事件

.subscribe(new Action1<ViewClickEvent>() {

@Override

public void call(ViewClickEvent event) {

// Click handling

}

});

看起来除了形式变了没什么区别，实质上也是这样。甚至如果你看一下它的源码，你会发现它连实现都没什么惊喜：它的内部是直接用一个包裹着的 setOnClickListener() 来实现的。然而，仅仅这一个形式的改变，却恰好就是 RxBinding 的目的：扩展性。通过 RxBinding 把点击监听转换成 Observable 之后，就有了对它进行扩展的可能。扩展的方式有很多，根据需求而定。一个例子是前面提到过的 throttleFirst() ，用于去抖动，也就是消除手抖导致的快速连环点击：

RxView.clickEvents(button)

.throttleFirst(500, TimeUnit.MILLISECONDS)

.subscribe(clickAction);

如果想对 RxBinding 有更多了解，可以去它的 GitHub 项目 下面看看。

## RxBus

RxBus 名字看起来像一个库，但它并不是一个库，而是一种模式，它的思想是使用 RxJava 来实现了 EventBus ，而让你不再需要使用 Otto 或者 GreenRobot 的 EventBus。顺便说一句，Flipboard 已经用 RxBus 替换掉了 Otto ，目前为止没有不良反应。

### RxBus工具类

public class RxBus {

private static volatile RxBus defaultInstance;

// 主题

private final Subject<Object, Object> bus;

// PublishSubject只会把在订阅发生的时间点之后来自原始Observable的数据发射给观察者

private RxBus() {

bus = new SerializedSubject<>(PublishSubject.create());

}

//单例RxBus

public static RxBus getDefault() {

if (defaultInstance == null) {

synchronized (RxBus.class) {

if (defaultInstance == null) {

defaultInstance = new RxBus();

}

}

}

return defaultInstance;

}

// 提供了一个新的事件

public void post(Object o) {

try {

bus.onNext(o);

} catch (Exception e) {

bus.onError(new Throwable(e.getMessage()));

}

}

public void error(String message) {

Throwable throwable = new Throwable(message);

bus.onError(throwable);

}

// 根据传递的 eventType 类型返回特定类型(eventType)的 被观察者

public <T> Observable<T> toObservable(Class<T> eventType) {

return bus.ofType(eventType);

}

// 封装默认订阅

public <T> Subscription toDefaultObservable(Class<T> eventType, Action1<T> act) {

return bus.ofType(eventType).compose(RxUtil.<T>rxSchedulerHelper()).subscribe(act);

}

}

1. Subject同时充当了Observer和Observable的角色，Subject是非线程安全的，要避免该问题，需要将 Subject转换为一个 SerializedSubject ，上述RxBus类中把线程非安全的PublishSubject包装成线程安全的Subject。
2. PublishSubject只会把在订阅发生的时间点之后来自原始Observable的数据发射给观察者。
3. ofType操作符只发射指定类型的数据，其内部就是filter+cast

public final <R> Observable<R> ofType(final Class<R> klass) {

return filter(new Func1<T, Boolean>() {

@Override

public final Boolean call(T t) {

return klass.isInstance(t);

}

}).cast(klass);

}

### 事件类

public class RxJavaBusEvent<T> implements Serializable {

private String eventType; //事件类型

private T t; //传输的对象;

public RxJavaBusEvent() {

}

public RxJavaBusEvent(String eventType) {

this.eventType = eventType;

}

public RxJavaBusEvent(String eventType, T t) {

this.eventType = eventType;

this.t = t;

}

public String getEventType() {

return eventType;

}

public void setEventType(String eventType) {

this.eventType = eventType;

}

public T getT() {

return t;

}

public void setT(T t) {

this.t = t;

}

}

### 注册接收

addRxBusSubscribe(RxJavaBusEvent.class, new Action1<RxJavaBusEvent>() {

@Override

public void call(RxJavaBusEvent rxJavaBusEvent) {

if (rxJavaBusEvent.getEventType().equals(Constants.EventBusType.FINISH\_ADD\_APPOINT)) {

mView.finishActivity();

}

}

});

protected <U> void addRxBusSubscribe(Class<U> eventType, Action1<U> act) {

if (mCompositeSubscription == null) {

mCompositeSubscription = new CompositeSubscription();

}

mCompositeSubscription.add(RxBus.getDefault().toDefaultObservable(eventType, act));

}

### 发送事件

RxBus.getDefault().post(new RxJavaBusEvent(Constants.EventBusType.FINISH\_ADD\_APPOINT));

## RxPermissions

RxJava现在有2个大版本，一个RxJava 1.x（通常说RxJava），另一个是RxJava2。所以RxPermissions有2个版本，分别支持RxJava和RxJava2。其中master分支是支持RxJava 1.x的版本，包名为com.tbruyelle.rxpermissions。2.x分支是支持RxJava2的版本，包名为com.tbruyelle.rxpermissions2。 默认看到的是master分支，所以看到的代码是支持RxJava 1.x的RxPermissions。

### RxPermissions使用的注意事项

* minSdkVersion必须 >= 11。
* 使用RxPermissions申请权限，必须在Activity.onCreate()或者View.onFinishInflate()中处理。不能在onResume()中处理，因为onResume()在App的生命周期中可能执行的很频繁。如果在请求权限的时候，App重新启动了（例如屏幕旋转导致的App关闭再重新创建），那么用户的选择（允许 或者 拒绝）将无法发给App。

### RxPermissions使用

**引用：**<https://github.com/tbruyelle/RxPermissions>

compile 'com.tbruyelle.rxpermissions2:rxpermissions:0.9.4@aar'

**代码：**

RxPermissions(mView)  
 .requestEach(Manifest.permission.*WRITE\_EXTERNAL\_STORAGE*)  
 .subscribe **{** permission **->** when {  
 permission.granted -> {  
 // 用户已经同意该权限  
 }  
 permission.shouldShowRequestPermissionRationale -> {  
 // 用户拒绝了该权限，没有选中『不再询问』,那么下次再次启动时，还会提示请求权限的对话框  
 }  
 else -> {  
 // 用户拒绝了该权限，并且选中『不再询问』  
 ToastUtil.shortShow(permission.name + "已被禁止提示，请到设置界面获取")  
 }  
 }  
 **}**

或

RxPermissions rxPermission = new RxPermissions(getActivity());

rxPermission

.requestEach(Manifest.permission.ACCESS\_FINE\_LOCATION,

Manifest.permission.WRITE\_EXTERNAL\_STORAGE,

Manifest.permission.READ\_CALENDAR,

Manifest.permission.READ\_CALL\_LOG,

Manifest.permission.READ\_CONTACTS,

Manifest.permission.READ\_PHONE\_STATE,

Manifest.permission.READ\_SMS,

Manifest.permission.RECORD\_AUDIO,

Manifest.permission.CAMERA)

.subscribe(new Subscriber<Permission>() {

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError", e);

}

@Override

public void onNext(Permission permission) {

if (permission.granted) {

// 用户已经同意该权限

Log.d(TAG, permission.name + " is granted.");

} else if (permission.shouldShowRequestPermissionRationale) {

// 用户拒绝了该权限，没有选中『不再询问』,那么下次再次启动时，还会提示请求权限的对话框

Log.w(TAG, permission.name + " is denied. More info should be provided.");

} else {

// 用户拒绝了该权限，并且选中『不再询问』

ToastUtil.shortShow(permission.name + "已被禁止提示，请到设置界面获取");

Log.e(TAG, permission.name + " is denied.");

}

}

});

## 一个简单的应用

**Course**

public class Course {

private int name;

public Course(int name) {

this.name = name;

}

public int getName() {

return name;

}

public void setName(int name) {

this.name = name;

}

}

**Student**

public class Student {

private List<Course> courses;

public Student(List<Course> courses) {

this.courses = courses;

}

public List<Course> getCourses() {

return courses;

}

public void setCourses(List<Course> courses) {

this.courses = courses;

}

}

public class MainActivity extends Activity {

private ImageView imageView;

private Context mContext;

private Student[] students;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

imageView = (ImageView) findViewById(R.id.img);

mContext = MainActivity.this;

intData();

doSomeThing();

}

private void intData() {

Course c1 = new Course(R.mipmap.devicenewadd1);

Course c2 = new Course(R.mipmap.devicenewadd2);

Course c3 = new Course(R.mipmap.devicenewadd3);

Course c4 = new Course(R.mipmap.devicenewadd4);

List l1 = new ArrayList();

List l2 = new ArrayList();

l1.add(c1);

l1.add(c2);

l2.add(c3);

l2.add(c4);

Student s1 = new Student(l1);

Student s2 = new Student(l2);

students = new Student[]{s1, s2};

}

private void doSomeThing() {

Observable.from(students)

.subscribeOn(Schedulers.io())//指定事件发生在io线程

.observeOn(Schedulers.io())//指定flatMap()发生在io线程

.flatMap(new Func1<Student, Observable<Course>>() {//将学生对象铺平为课程

@Override

public Observable<Course> call(Student student) {

return Observable.from(student.getCourses());

}

})

.observeOn(Schedulers.computation())//指定map()发生在io线程

.map(new Func1<Course, Drawable>() {//将学生对象转为图片对象

@Override

public Drawable call(Course course) {

Log.e("tag", course.getName() + "");

return getDrawable(course.getName());

}

})

.observeOn(AndroidSchedulers.mainThread())//指定事件消费在安卓主线程

.subscribe(new Action1<Drawable>() {//定义订阅者的onnext()

@Override

public void call(Drawable drawable) {

imageView.setImageDrawable(drawable);

}

});

}

}

# RxJava2

## Gradle配置

compile 'io.reactivex.rxjava2:rxjava:2.2.0'

compile 'io.reactivex.rxjava2:rxandroid:2.2.0'

## Observable、Observer、subscribe()

//创建一个上游 Observable：  
Observable<Integer> observable = Observable.*create*(emitter -> {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onComplete();  
});  
//创建一个下游 Observer  
Observer<Integer> observer = new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 Log.*d*(*TAG*, "subscribe");  
 }  
 @Override  
 public void onNext(Integer value) {  
 Log.*d*(*TAG*, "" + value);  
 }  
 @Override  
 public void onError(Throwable e) {  
 Log.*d*(*TAG*, "error");  
 }  
 @Override  
 public void onComplete() {  
 Log.*d*(*TAG*, "complete");  
 }  
};  
//建立连接  
observable.subscribe(observer);

其中两个陌生的玩意：ObservableEmitter和Disposable.

ObservableEmitter

Emitter是发射器的意思，那就很好猜了，这个就是用来发出事件的，它可以发出三种类型的事件，通过调用emitter的onNext(T value)、onComplete()和onError(Throwable error)就可以分别发出next事件、complete事件和error事件。需要满足一定的规则：

* 上游可以发送无限个onNext, 下游也可以接收无限个onNext.
* 当上游发送了一个onComplete后, 上游onComplete之后的事件将会继续发送, 而下游收到onComplete事件之后将不再继续接收事件.
* 当上游发送了一个onError后, 上游onError之后的事件将继续发送, 而下游收到onError事件之后将不再继续接收事件.
* 上游可以不发送onComplete或onError.
* 最为关键的是onComplete和onError必须唯一并且互斥, 即不能发多个onComplete, 也不能发多个onError, 也不能先发一个onComplete, 然后再发一个onError, 反之亦然

Disposable

这个单词的字面意思是一次性用品,用完即可丢弃的. 那么在RxJava中怎么去理解它呢, 对应于上面的水管的例子, 我们可以把它理解成两根管道之间的一个机关, 当调用它的dispose()方法时, 它就会将两根管道切断, 从而导致下游收不到事件.

注意: 调用dispose()并不会导致上游不再继续发送事件, 上游会继续发送剩余的事件.来看个例子, 我们让上游依次发送1,2,3,complete,4,在下游收到第二个事件之后, 切断水管, 看看运行结果:

subscribe()有多个重载的方法:

public final Disposable subscribe() {}

public final Disposable subscribe(Consumer<? super T> onNext) {}

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError) {}

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError,

Action onComplete) {}

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError,

Action onComplete, Consumer<? super Disposable> onSubscribe) {}

public final void subscribe(Observer<? super T> observer) {}

最后一个带有Observer参数的我们已经使用过了,这里对其他几个方法进行说明.

* 不带任何参数的subscribe() 表示下游不关心任何事件,你上游尽管发你的数据去吧, 老子可不管你发什么.
* 带有一个Consumer参数的方法表示下游只关心onNext事件, 其他的事件我假装没看见, 因此我们如果只需要onNext事件可以这么写:

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 }  
}).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*d*(*TAG*, "onNext: " + integer);  
 }  
});

* 其他几个方法同理, 这里就不一一解释了.

## 线程切换

正常情况下, 上游和下游是工作在同一个线程中的, 也就是说上游在哪个线程发事件, 下游就在哪个线程接收事件.这样肯定是满足不了我们的需求的, 我们更多想要的是在子线程中做耗时的操作, 然后回到主线程中来操作UI

要达到这个目的, 我们需要先改变上游发送事件的线程, 让它去子线程中发送事件, 然后再改变下游的线程, 让它去主线程接收事件. 通过RxJava内置的线程调度器可以很轻松的做到这一点. 接下来看一段代码:

observable.subscribeOn(Schedulers.newThread())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(consumer);

* 简单的来说, subscribeOn() 指定的是上游发送事件的线程, observeOn() 指定的是下游接收事件的线程.
* 多次指定上游的线程只有第一次指定的有效, 也就是说多次调用subscribeOn() 只有第一次的有效, 其余的会被忽略
* 多次指定下游的线程是可以的, 也就是说每调用一次observeOn() , 下游的线程就会切换一次.

## 生命周期相关

如果在请求的过程中Activity已经退出了, 这个时候如果回到主线程去更新UI, 那么APP肯定就崩溃了, 怎么办呢, 上一节我们说到了Disposable , 说它是个开关, 调用它的dispose()方法时就会切断水管, 使得下游收不到事件, 既然收不到事件, 那么也就不会再去更新UI了. 因此我们可以在Activity中将这个Disposable 保存起来, 当Activity退出时, 切断它即可.

那如果有多个Disposable 该怎么办呢, RxJava中已经内置了一个容器CompositeDisposable, 每当我们得到一个Disposable时就调用CompositeDisposable.add()将它添加到容器中, 在退出的时候, 调用CompositeDisposable.clear() 即可切断所有的水管.

## 背压BackPress

### 水缸满了就OOM

Observable<Integer> observable1 = Observable.*create*(emitter -> {  
 for (int i = 0; ; i++) { //无限循环发事件  
 emitter.onNext(i);  
 }  
});  
Observable<String> observable2 = Observable.*just*("A");  
Observable  
 .*zip*(observable1, observable2, (integer, s) -> integer + s)  
 .compose(RxUtil.*rxSchedulerHelper*())  
 .subscribe(s -> Log.*d*(*TAG*, s));

我们分别创建了两根水管, 第一根水管用机器指令的执行速度来无限循环发送事件, 第二根水管随便发送点什么, 由于我们没有发送Complete事件, 因此第一根水管会一直发事件到它对应的水缸里去，内存占用以斜率为1的直线迅速上涨, 几秒钟就300多M , 最终报出了OOM

所谓的Backpressure其实就是为了控制流量, 水缸存储的能力毕竟有限, 因此我们还得从源头去解决问题, 既然你发那么快, 数据量那么大, 那我就想办法不让你发那么快呗.

那么这个源头到底在哪里, 究竟什么时候会出现这种情况, 这里只是说的Zip这一个例子, 其他的地方会出现吗? 带着这个问题我们来探究一下.

我们让事情变得简单一点, 从一个单一的Observable说起.

Observable.*create*((ObservableOnSubscribe<Integer>) emitter -> {  
 for (int i = 0; ; i++) { //无限循环发事件   
 emitter.onNext(i);  
 }  
}).subscribe(integer -> {  
 Thread.*sleep*(2000);  
 Log.*d*(*TAG*, "" + integer);  
});

这段代码很简单, 上游同样无限循环的发送事件, 在下游每次接收事件前延时2秒. 上下游工作在同一个线程里, 运行十分平稳，因为每隔两秒发一个就消费了一个。那我们加个线程呢, 改成这样：

.subscribeOn(Schedulers.*io*())  
.observeOn(AndroidSchedulers.*mainThread*())

可以看到, 给上游加了个线程之后, 它就像脱缰的野马一样, 内存又爆掉了

为什么不加线程和加上线程区别这么大呢, 这就涉及了同步和异步的知识了.

* 当上下游工作在同一个线程中时, 这时候是一个同步的订阅关系, 也就是说上游每发送一个事件必须等到下游接收处理完了以后才能接着发送下一个事件.
* 当上下游工作在不同的线程中时, 这时候是一个异步的订阅关系, 这个时候上游发送数据不需要等待下游接收, 为什么呢, 因为两个线程并不能直接进行通信, 因此上游发送的事件并不能直接到下游里去
* 这个时候就需要一个田螺姑娘来帮助它们俩, 这个田螺姑娘就是我们刚才说的水缸 ! 上游把事件发送到水缸里去, 下游从水缸里取出事件来处理,
* 因此, 当上游发事件的速度太快, 下游取事件的速度太慢, 水缸就会迅速装满, 然后溢出来, 最后就OOM了.

### 自己解决

Observable.*create*((ObservableOnSubscribe<Integer>) emitter -> {  
 for (int i = 0; ; i++) { //无限循环发事件  
 emitter.onNext(i);  
 }  
 })  
 .subscribeOn(Schedulers.*io*())  
 .sample(2, TimeUnit.*SECONDS*) //sample取样  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(integer -> {  
 Thread.*sleep*(2000);  
 Log.*d*(*TAG*, "" + integer);  
 });

虽然上游仍然一直在不停的发事件, 但是我们只是每隔一定时间取一个放进水缸里, 并没有全部放进水缸里, 因此这次内存仅仅只占用了5M.

这种方法归根到底其实就是减少放进水缸的事件的数量, 是以数量取胜, 但是这个方法有个缺点, 就是丢失了大部分的事件.那么我们换一个角度来思考, 既然上游发送事件的速度太快, 那我们就适当减慢发送事件的速度, 从速度上取胜, 听上去不错, 我们来试试:

for (int i = 0; ; i++) { //无限循环发事件  
 emitter.onNext(i);  
 Thread.*sleep*(2000); //每次发送完事件延时2秒  
}

上游通过适当的延时, 不但减缓了事件进入水缸的速度, 也可以让下游有充足的时间从水缸里取出事件来处理 , 这样一来, 就不至于导致大量的事件涌进水缸, 也就不会OOM啦.

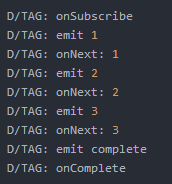
### 用Flowable解决

#### 基础用法

之前我们所的上游和下游分别是Observable和Observer, 这次不一样的是上游变成了Flowable, 下游变成了Subscriber, 但是水管之间的连接还是通过subscribe(), 我们来看看最基本的用法吧:

//创建一个上游 Flowable：  
Flowable<Integer> upstream = Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Log.*d*(*TAG*, "emit complete");  
 emitter.onComplete();  
 }  
}, BackpressureStrategy.*ERROR*); //增加了一个参数

//创建一个下游 Subscriber  
Subscriber<Integer> subscriber = new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 s.request(Long.*MAX\_VALUE*); //注意这句代码  
 }  
 @Override  
 public void onNext(Integer integer) {  
 Log.*d*(*TAG*, "onNext: " + integer);  
 }  
 @Override  
 public void onError(Throwable t) {  
 Log.*w*(*TAG*, "onError: ", t);  
 }  
 @Override  
 public void onComplete() {  
 Log.*d*(*TAG*, "onComplete");  
 }  
};  
//建立连接  
flowable.subscribe(subscriber);



我们注意到这次和Observable有些不同. 首先是创建Flowable的时候增加了一个参数, 这个参数是用来选择背压,也就是出现上下游流速不均衡的时候应该怎么处理的办法, 这里我们直接用BackpressureStrategy.ERROR这种方式, 这种方式会在出现上下游流速不均衡的时候直接抛出一个异常,这个异常就是著名的MissingBackpressureException（注意，只是抛异常，但是不会崩了）. 其余的策略后面再来讲解.

另外的一个区别是在下游的onSubscribe方法中传给我们的不再是Disposable了, 而是Subscription, 它俩有什么区别呢, 首先它们都是上下游中间的一个开关, 之前我们说调用Disposable.dispose()方法可以切断水管, 同样的调用Subscription.cancel()也可以切断水管, 不同的地方在于Subscription增加了一个void request(long n)方法, 这个方法有什么用呢, 在上面的代码中也有这么一句代码:

s.request(Long.*MAX\_VALUE*);

如果不要它，从运行结果中可以看到, 在上游发送第一个事件之后, 下游就抛出了一个著名的MissingBackpressureException异常, 并且下游没有收到任何其余的事件

zlc.season.rxjava2demo D/TAG: onSubscribe

zlc.season.rxjava2demo D/TAG: emit 1

zlc.season.rxjava2demo W/TAG: onError:

io.reactivex.exceptions.MissingBackpressureException: create: could not emit value due to lack of requests

at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncEmitter...

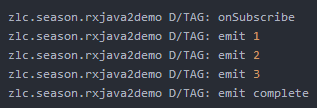
zlc.season.rxjava2demo D/TAG: emit 2

zlc.season.rxjava2demo D/TAG: emit 3

zlc.season.rxjava2demo D/TAG: emit complete

可是这是一个同步的订阅呀, 上下游工作在同一个线程, 上游每发送一个事件应该会等待下游处理完了才会继续发事件啊, 不可能出现上下游流速不均衡的问题呀.带着这个疑问, 我们再来看看异步的情况:

...BackpressureStrategy.*ERROR*)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() { ... }



这次上游正确的发送了所有的事件, 但是下游一个事件也没有收到.这是因为Flowable在设计的时候采用了一种新的思路,也就是响应式拉取的方式来更好的解决上下游流速不均衡的问题, 与我们之前所讲的控制数量和控制速度不太一样

我们把request当做是一种能力, 当成下游每次最多可以处理事件个数的能力, 下游能处理几个就告诉上游我要几个, 这样只要上游根据下游的处理能力来决定发送多少事件, 就不会造成一窝蜂的发出一堆事件来, 从而导致OOM. 这也就完美的解决之前我们所学到的两种方式的缺陷, 过滤事件会导致事件丢失, 减速又可能导致性能损失. 而这种方式既解决了事件丢失的问题, 又解决了速度的问题, 完美 !

但这里需要注意的是, 只有当上游正确的实现了如何根据下游的处理能力来发送事件的时候, 才能达到这种效果, 如果上游根本不管下游的处理能力, 一股脑的瞎他妈发事件, 仍然会产生上下游流速不均衡的问题。那么如何正确的去实现上游呢, 这里先卖个关子, 之后我们再来讲解.

学习了request, 我们就可以解释上面的两段代码了.首先第一个同步的代码, 为什么上游发送第一个事件后下游就抛出了MissingBackpressureException异常, 这是因为下游没有调用request, 上游就认为下游没有处理事件的能力, 而这又是一个同步的订阅, 既然下游处理不了, 那上游不可能一直等待吧, 如果是这样, 万一这两根水管工作在主线程里, 界面不就卡死了吗, 因此只能抛个异常来提醒我们. 那如何解决这种情况呢, 很简单啦, 下游直接调用request(Long.MAX\_VALUE)就行了, 或者根据上游发送事件的数量来request就行了, 比如这里request(3)就可以了.（如果是request(2)，不会崩，但是emit 3收不到了）

然后是异步代码, 为什么上下游没有工作在同一个线程时, 上游却正确的发送了所有的事件呢? 这是因为在Flowable里默认有一个大小为128的水缸, 当上下游工作在不同的线程中时, 上游就会先把事件发送到这个水缸中, 因此, 下游虽然没有调用request, 但是上游在水缸中保存着这些事件, 只有当下游调用request时, 才从水缸里取出事件发给下游.

注意这里我们是把上游发送的事件全部都存进了水缸里, 下游一个也没有消费, 所以就溢出了, 如果下游去消费了事件, 可能就不会导致水缸溢出来了. 这里我们说的是可能不会, 这也很好理解, 比如刚才这个例子上游发了129个事件, 下游只要快速的消费了一个事件, 就不会溢出了, 但如果下游过了十秒钟再来消费一个, 那肯定早就溢出了

#### 大水缸BackpressureStrategy.BUFFER)

这次我们用的策略是BackpressureStrategy.BUFFER, 这就是我们的新水缸啦, 这个水缸就比原来的水缸牛逼多了,它没有大小限制, 因此可以存放许许多多的事件。换了大水缸的FLowable表现出来的特性和Observable一模一样, 因此, 如果像这样单纯的使用FLowable, 同样需要注意OOM的问题, 例如下面这个例子:

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 }  
 }  
}, BackpressureStrategy.*BUFFER*)

同样可以看到, 内存迅速增长, 直到最后抛出OOM.可能有朋友也注意到了, 之前使用Observable测试的时候内存增长非常迅速, 几秒钟就OOM, 但这里增长速度却比较缓慢, 这也看出FLowable相比Observable, 在性能方面有些不足, 毕竟FLowable内部为了实现响应式拉取做了更多的操作, 性能有所丢失也是在所难免, 因此单单只是说因为FLowable是新兴产物就盲目的使用也是不对的, 也要具体分场景

#### 数量解决DROP、LATEST

BackpressureStrategy.DROP和BackpressureStrategy.LATEST这两种策略从名字上就能猜到它俩是干啥的, Drop就是直接把存不下的事件丢弃,Latest就是只保留最新的事件

private void test11() {  
 Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 for (int i = 0; ; i++) {  
 emitter.onNext(i);  
 }  
 }  
 }, BackpressureStrategy.*DROP*).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 }  
 @Override  
 public void onNext(Integer integer) {  
 Log.*d*(*TAG*, "onNext: " + integer);  
 }  
 @Override  
 public void onError(Throwable t) {  
 Log.*w*(*TAG*, "onError: ", t);  
 }  
 @Override  
 public void onComplete() {  
 Log.*d*(*TAG*, "onComplete");  
 }  
 });  
}  
public void request () {  
 mSubscription.request(128);  
}

第一次点击会收到1-127，第二次点击会收到中间的128个

## 配合操作符

Flowable.*interval*(1, 1, TimeUnit.*MICROSECONDS*)  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Long>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(Long.*MAX\_VALUE*);  
 }  
 @Override  
 public void onNext(Long aLong) {  
 Log.*d*(*TAG*, "onNext: " + aLong);  
 try {  
 Thread.*sleep*(1000); //延时1秒  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 @Override  
 public void onError(Throwable t) {  
 Log.*w*(*TAG*, "onError: ", t);  
 }  
 @Override  
 public void onComplete() {  
 Log.*d*(*TAG*, "onComplete");  
 }  
 });

一运行就抛出了MissingBackpressureException异常, 提醒我们发太多了，RxJava给我们提供了其他的方法:

* onBackpressureBuffer()
* onBackpressureDrop()
* onBackpressureLatest()

这跟我们上面学的策略是一样的, 用法也简单, 拿刚才的例子现学现用:

Flowable.*interval*(1, 1, TimeUnit.*MICROSECONDS*)  
 .onBackpressureDrop() //加上背压策略  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Long>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(Long.*MAX\_VALUE*);  
 }  
 @Override  
 public void onNext(Long aLong) {  
 Log.*d*(*TAG*, "onNext: " + aLong);  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ...  
 });

## 响应式FlowableEmitter的requested

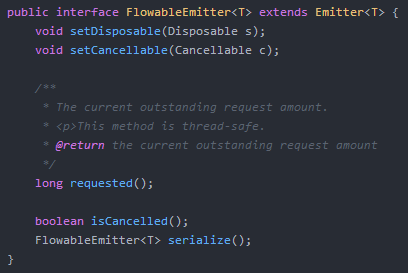
在我们前两节中的例子中，我们口中声称的响应式并没有完全体现出来，比如这个例子：

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Log.*d*(*TAG*, "emit complete");  
 emitter.onComplete();  
 }  
}, BackpressureStrategy.*ERROR*).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(1);  
 }  
 @Override  
 public void onNext(Integer integer) {  
 Log.*d*(*TAG*, "onNext: " + integer);  
 mSubscription.request(1);  
 }  
 ...  
 });

虽然我们在下游中是每次处理掉了一个事件之后才调用request(1)去请求下一个事件，可是上游呢？上游真的是每次当下游请求一个才拿出一个吗？从上上篇文章中我们知道并不是这样的，上游仍然是一开始就发送了所有的事件，也就是说小日本并没有等叶问打死一个才拿出一个，而是一开始就拿出了所有的鬼子，这些鬼子从一开始就在这儿排队等着被打死。

没错，我们前后所说的就是自相矛盾了，这说明了什么呢，说明我们的实现并不是一个完整的实现，那么，究竟怎样的实现才是完整的呢？

我们先自己来想一想，在下游中调用Subscription.request(n)就可以告诉上游，下游能够处理多少个事件，那么上游要根据下游的处理能力正确的去发送事件，那么上游是不是应该知道下游的处理能力是多少啊，对吧，不然，一个巴掌拍不响啊，这种事情得你情我愿才行。那么上游从哪里得知下游的处理能力呢？我们来看看上游最重要的部分，肯定就是FlowableEmitter了啊，我们就是通过它来发送事件的啊，来看看它的源码吧(别紧张，它的代码灰常简单)：



FlowableEmitter是个接口，继承Emitter，Emitter里面就是我们的onNext(),onComplete()和onError()三个方法。我们看到FlowableEmitter中有这么一个方法：

long requested();

### 同步的情况

方法注释的意思就是当前外部请求的数量，哇哦，这好像就是我们要找的答案呢. 我们还是实际验证一下,先来看同步的情况吧：

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "current requested: " + emitter.requested());  
 }  
}, BackpressureStrategy.*ERROR*)  
 .subscribe(new Subscriber<Integer>() { ... });

例子中，我们在上游中打印出当前的request数量，下游什么也不做。看看运行结果：

D/TAG: current requested: 0

下游没有调用request()，说明当前下游的处理能力为0，那么上游得到的requested也应该是0.那下游要是调用了request()呢，来看看：

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "current requested: " + emitter.requested());  
 }  
}, BackpressureStrategy.*ERROR*)  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 mSubscription = s;  
 s.request(10); //我要打十个！  
 }  
 ...  
 });

看看运行结果：

D/TAG: current requested: 10

果然！上游的requested的确是根据下游的请求来决定的，那要是下游多次请求呢？比如这样：

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "current requested: " + emitter.requested());  
 }  
}, BackpressureStrategy.*ERROR*)  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(10); //我要打十个!  
 s.request(100); //再给我一百个！  
 }  
 ...  
 });

下游先调用了request(10), 然后又调用了request(100)，来看看运行结果：

D/TAG: onSubscribe

D/TAG: current requested: 110

看来多次调用也没问题，做了加法。那什么时候做减法呢？当然是发送事件啦！来看个例子吧：

Flowable.*create*((FlowableOnSubscribe<Integer>) emitter -> {  
 Log.*d*(*TAG*, "before emit, requested = " + emitter.requested());  
 Log.*d*(*TAG*, "emit 1");  
 emitter.onNext(1);  
 Log.*d*(*TAG*, "after emit 1, requested = " + emitter.requested());  
 Log.*d*(*TAG*, "emit 2");  
 emitter.onNext(2);  
 Log.*d*(*TAG*, "after emit 2, requested = " + emitter.requested());  
 Log.*d*(*TAG*, "emit 3");  
 emitter.onNext(3);  
 Log.*d*(*TAG*, "after emit 3, requested = " + emitter.requested());  
 Log.*d*(*TAG*, "emit complete");  
 emitter.onComplete();  
 Log.*d*(*TAG*, "after emit complete, requested = " + emitter.requested());  
}, BackpressureStrategy.*ERROR*)  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(10); //request 10  
 }  
 ...  
 });

D/TAG: onSubscribe  
D/TAG: before emit, requested = 10  
D/TAG: emit 1  
D/TAG: onNext: 1  
D/TAG: after emit 1, requested = 9  
D/TAG: emit 2  
D/TAG: onNext: 2  
D/TAG: after emit 2, requested = 8  
D/TAG: emit 3  
D/TAG: onNext: 3  
D/TAG: after emit 3, requested = 7  
D/TAG: emit complete  
D/TAG: onComplete  
D/TAG: after emit complete, requested = 7

大家应该能看出端倪了吧，下游调用request(n) 告诉上游它的处理能力，上游每发送一个next事件之后，requested就减一，注意是next事件，complete和error事件不会消耗requested，当减到0时，则代表下游没有处理能力了，这个时候你如果继续发送事件，会发生什么后果呢？当然是MissingBackpressureException啦

当上下游在同一个线程中的时候，在下游调用request(n)就会直接改变上游中的requested的值，多次调用便会叠加这个值，而上游每发送一个事件之后便会去减少这个值，当这个值减少至0的时候，继续发送事件便会抛异常了。

### 异步的情况

同样的先来看一个基本的例子

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "current requested: " + emitter.requested());  
 }  
 }, BackpressureStrategy.*ERROR*)  
 .subscribeOn(Schedulers.*io*()).observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 mSubscription = s;  
 }  
 ...  
 });

这次是异步的情况，上游啥也不做，下游也啥也不做，来看看运行结果：

D/TAG: current requested: 128

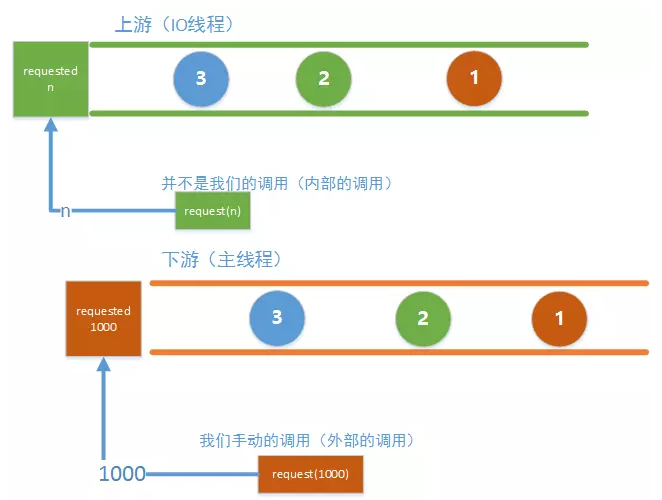
哈哈，又是128，看了我前几篇文章的朋友肯定很熟悉这个数字啊！这个数字为什么和我们之前所说的默认的水缸大小一样啊，带着这个疑问我们继续来研究一下

Flowable.*create*(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {  
 Log.*d*(*TAG*, "current requested: " + emitter.requested());  
 }  
 }, BackpressureStrategy.*ERROR*)  
 .subscribeOn(Schedulers.*io*()).observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 s.request(1000); //我要打1000个！！  
 }  
 ...  
 });

这次我们在下游调用了request（1000）告诉上游我要打1000个，按照之前我们说的，这次的运行结果应该是1000，来看看运行结果：

D/TAG: onSubscribe

D/TAG: current requested: 128



可以看到，当上下游工作在不同的线程里时，每一个线程里都有一个requested，而我们调用request（1000）时，实际上改变的是下游主线程中的requested，而上游中的requested的值是由RxJava内部调用request(n)去设置的，这个调用会在合适的时候自动触发。

现在我们就能理解为什么没有调用request，上游中的值是128了，因为下游在一开始就在内部调用了request(128)去设置了上游中的值，因此即使下游没有调用request()，上游也能发送128个事件，这也可以解释之前我们为什么说Flowable中默认的水缸大小是128，其实就是这里设置的。

刚才同步的时候我们说了，上游每发送一个事件，requested的值便会减一，对于异步来说同样如此，那有人肯定有疑问了，一开始上游的requested的值是128，那这128个事件发送完了不就不能继续发送了吗？

刚刚说了，设置上游requested的值的这个内部调用，会在合适的时候自动触发，那到底什么时候是合适的时候呢？是发完128个事件才去调用吗？还是发送了一半才去调用呢？带着这个疑问我们来看下一段代码：

public void request() {  
 mSubscription.request(96); //请求96个事件  
}  
  
public void demo4() {  
 Flowable.*create*((FlowableOnSubscribe<Integer>) emitter -> {  
 Log.*d*(*TAG*, "First requested = " + emitter.requested());  
 boolean flag;  
 for (int i = 0; ; i++) {  
 flag = false;  
 while (emitter.requested() == 0) {  
 if (!flag) {  
 Log.*d*(*TAG*, "Oh no! I can't emit value!");  
 flag = true;  
 }  
 }  
 emitter.onNext(i);  
 Log.*d*(*TAG*, "emit " + i + " , requested = " + emitter.requested());  
 }  
 }, BackpressureStrategy.*ERROR*)  
 .subscribeOn(Schedulers.*io*()).observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 Log.*d*(*TAG*, "onSubscribe");  
 mSubscription = s;  
 }  
 ...  
 });  
}

这次的上游稍微复杂了一点点，首先仍然是个无限循环发事件，但是是有条件的，只有当上游的requested != 0的时候才会发事件，当emitter.requested() == 0时，会一直陷入while循环中，不往下走了。首先运行之后上游便会发送完128个事件，之后便不做任何事情，从打印的结果中我们也可以看出这一点

D/TAG: onSubscribe

D/TAG: First requested = 128

D/TAG: emit 0 , requested = 127

D/TAG: emit 1 , requested = 126

D/TAG: emit 2 , requested = 125

...

D/TAG: emit 124 , requested = 3

D/TAG: emit 125 , requested = 2

D/TAG: emit 126 , requested = 1

D/TAG: emit 127 , requested = 0

D/TAG: Oh no! I can't emit value!

然后我们调用request(96)，这会让下游去消费96个事件，来看看运行结果吧：

D/TAG: onNext: 0

D/TAG: onNext: 1

...

D/TAG: onNext: 92

D/TAG: onNext: 93

D/TAG: onNext: 94

D/TAG: onNext: 95

D/TAG: emit 128 , requested = 95

D/TAG: emit 129 , requested = 94

D/TAG: emit 130 , requested = 93

D/TAG: emit 131 , requested = 92

...

D/TAG: emit 219 , requested = 4

D/TAG: emit 220 , requested = 3

D/TAG: emit 221 , requested = 2

D/TAG: emit 222 , requested = 1

D/TAG: emit 223 , requested = 0

D/TAG: Oh no! I can't emit value!

可以看到，当下游消费掉第96个事件之后，while循环的条件emitter.requested() == 0不成立，while往下走，上游又开始发事件了，而且可以看到当前上游的requested的值是96(打印出来的95是已经发送了一个事件减一之后的值)，最终发出了第223个事件之后又进入了等待区，而223-127 正好等于 96。

这是不是说明当下游每消费96个事件（注意不是每一个），便会自动触发内部的request()去设置上游的requested的值啊！没错，就是这样，而这个新的值就是96。朋友们可以手动试试请求95个事件，上游是不会继续发送事件的。

### 实践

实现一个完整的响应式拉取，在某一些场景下，可以在发送事件前先判断当前的requested的值是否大于0，若等于0则说明下游处理不过来了，则需要等待，例如下面这个例子。

这个例子是读取一个文本文件，需要一行一行读取，然后处理并输出，如果文本文件很大的时候，比如几十M的时候，全部先读入内存肯定不是明智的做法，因此我们可以一边读取一边处理，实现的代码如下：

Flowable.*create*((FlowableOnSubscribe<String>) emitter -> {  
 try {  
 FileReader reader = new FileReader("test.txt");  
 BufferedReader br = new BufferedReader(reader);  
 String str;  
 //有文字时，一行一行的读  
 while ((str = br.readLine()) != null && !emitter.isCancelled()) {  
 //如果没有requested() == 0就用while循环卡主  
 while (emitter.requested() == 0) {  
 if (emitter.isCancelled()) {  
 break;  
 }  
 }  
 //requested() != 0的时候，就发送事件  
 emitter.onNext(str);  
 }  
 //所有文字的读完了，关闭流，发送onComplete()  
 br.close();  
 reader.close();  
 emitter.onComplete();  
 } catch (Exception e) {  
 emitter.onError(e);  
 }  
}, BackpressureStrategy.*ERROR*)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(Schedulers.*newThread*())  
 .subscribe(new Subscriber<String>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 mSubscription = s;  
 //开始订阅必须先请求一个，才会有next收到消息  
 s.request(1);  
 }  
 @Override  
 public void onNext(String string) {  
 //打印文字  
 System.*out*.println(string);  
 try {  
 Thread.*sleep*(500);  
 //每读完一行，就去请求下一行  
 mSubscription.request(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 @Override  
 public void onError(Throwable t) {  
 System.*out*.println(t);  
 }  
 @Override  
 public void onComplete() {  
 }  
 });

## 一些操作符

### map

之前：

Observable.just("images/logo.png") // 输入类型 String

.map(new Func1<String, Bitmap>() {

@Override

public Bitmap call(String filePath) { // 参数类型 String

return getBitmapFromPath(filePath); // 返回类型 Bitmap

}

})

.subscribe(new Action1<Bitmap>() {

@Override

public void call(Bitmap bitmap) { // 参数类型 Bitmap

showBitmap(bitmap);

}

});

现在：

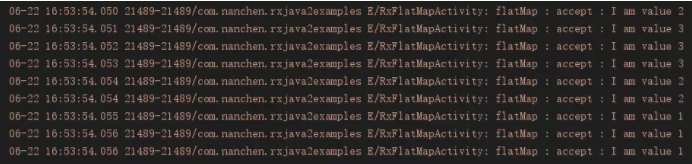
Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {  
 e.onNext(1);  
 e.onNext(2);  
 e.onNext(3);  
 }  
}).map(new Function<Integer, String>() {  
 @Override  
 public String apply(@NonNull Integer integer) throws Exception {  
 return "This is result " + integer;  
 }  
}).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
 mRxOperatorsText.append("accept : " + s +"\n");  
 Log.*e*(*TAG*, "accept : " + s +"\n" );  
 }  
});

### FlatMap

可以把一个发射器 Observable 通过某种方法转换为多个 Observables，然后再把这些分散的 Observables装进一个单一的发射器 Observable。但有个需要注意的是，flatMap 并不能保证事件的顺序，如果需要保证，需要用到我们下面要讲的 ConcatMap

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {  
 e.onNext(1);  
 e.onNext(2);  
 e.onNext(3);  
 }  
}).flatMap(new Function<Integer, ObservableSource<String>>() {  
 @Override  
 public ObservableSource<String> apply(@NonNull Integer integer) throws Exception {  
 List<String> list = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 list.add("I am value " + integer);  
 }  
 int delayTime = (int) (1 + Math.*random*() \* 10);  
 return Observable.*fromIterable*(list).delay(delayTime, TimeUnit.*MILLISECONDS*);  
 }  
}).subscribeOn(Schedulers.*newThread*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
 Log.*e*(*TAG*, "flatMap : accept : " + s + "\n");  
 }  
 });

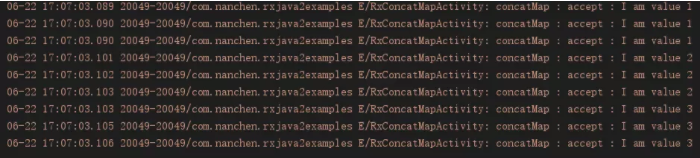
Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {  
 e.onNext(1);  
 e.onNext(2);  
 e.onNext(3);  
 }  
}).flatMap(new Function<Integer, ObservableSource<String>>() {  
 @Override  
 public ObservableSource<String> apply(@NonNull Integer integer) throws Exception {  
 List<String> list = new ArrayList<>();  
 for (int i = 0; i < 3; i++) {  
 list.add("I am value " + integer);  
 }  
 int delayTime = (int) (1 + Math.*random*() \* 10);  
 return Observable.*fromIterable*(list).delay(delayTime, TimeUnit.*MILLISECONDS*);  
 }  
}).subscribeOn(Schedulers.*newThread*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
 Log.*e*(*TAG*, "flatMap : accept : " + s + "\n");  
 }  
 });



一切都如我们预期中的有意思，为了区分 concatMap（下一个会讲），我在代码中特意动了一点小手脚，我采用一个随机数，生成一个时间，然后通过 delay（后面会讲）操作符，做一个小延时操作，而查看 Log 日志也确认验证了我们上面的说法，它是无序的。

### concatMap

上面其实就说了，concatMap 与 FlatMap 的唯一区别就是 concatMap 保证了顺序，所以，我们就直接把 flatMap 替换为 concatMap 验证吧。



### interval和timer

timer 很有意思，相当于一个定时任务。在 1.x 中它还可以执行间隔逻辑，但在 2.x 中此功能被交给了 interval，下一个会介绍。但需要注意的是，timer 和 interval 均默认在新线程。

Observable.*timer*(2, TimeUnit.*SECONDS*)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*()) // timer 默认在新线程，所以需要切换回主线程  
 .subscribe(new Consumer<Long>() {  
 @Override  
 public void accept(@NonNull Long aLong) throws Exception {  
 Log.*e*(*TAG*, "timer :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");  
 }  
 });

interval 操作符用于间隔时间执行某个操作，其接受三个参数，分别是第一次发送延迟，间隔时间，时间单位。由于我们这个是间隔执行，所以当我们的Activity 都销毁的时候，实际上这个操作还依然在进行，所以，我们得花点小心思让我们在不需要它的时候干掉它。查看源码发现，我们subscribe(Cousumer<? super T> onNext)返回的是Disposable

private static Disposable *mDisposable*;

private void doSomeThing() {  
 mDisposable = Observable.*interval*(3, 2, TimeUnit.*SECONDS*)  
 .doOnNext(new Consumer<Long>() {  
 @Override  
 public void accept(Long aLong) throws Exception {  
 Log.*d*(*TAG*, "第 " + aLong + " 次轮询");

//做轮询请求  
 datamanagerHelper  
 .doMemberInfo()  
 .compose(RxUtil.*rxSchedulerHelper*())  
 .compose(RxUtil.*handleMyResult*())  
 .subscribe(new CommonSubscriber...)  
 }  
 })  
 .observeOn(AndroidSchedulers.*mainThread*()) // timer 默认在新线程，所以需要切换回主线程  
 .subscribe(new Consumer<Long>() {  
 @Override  
 public void accept(@NonNull Long aLong) throws Exception {

Log.*e*(*TAG*, "timer :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");  
 }  
 });  
}

@Override  
protected void onDestroy() {  
 super.onDestroy();  
 if (*mDisposable* != null && !*mDisposable*.isDisposed()) {  
 *mDisposable*.dispose();  
 }  
}

### Zip

zip 专用于合并事件

private void zipTest() {  
 Observable.*zip*(getOrderList(), getOrderDetail(), new BiFunction<CommonGetListResp<OrderListRow>, MortageDetailResponse, List<OrderListFraItemBean>>() {  
 @Override  
 public List<OrderListFraItemBean> apply(CommonGetListResp<OrderListRow> orderListRowCommonGetListResp, MortageDetailResponse mortageDetailResponse) throws Exception {  
 orderListRowCommonGetListResp.getRows();  
 mortageDetailResponse.getCoinCode();  
 return new ArrayList<>();  
 }  
 });  
}  
  
private Observable<CommonGetListResp<OrderListRow>> getOrderList() {  
 return datamanagerHelper.getMyOrderList(null)  
 .compose(RxUtil.*rxSchedulerHelper*())  
 .compose(RxUtil.*handleMyResult*());  
}  
  
private Observable<MortageDetailResponse> getOrderDetail() {  
 return datamanagerHelper.doQueryMortageDetail(new MortageDetailReq(""))  
 .compose(RxUtil.*rxSchedulerHelper*())  
 .compose(RxUtil.*handleMyResult*());  
}

Zip合并不是连接（连接操作符后面会说），而是两两配对，也就意味着，最终配对出的 Observable 发射事件数目只和少的那个相同。

private void zipTest() {  
 Observable.*zip*(getStringObservable(), getIntegerObservable(), new BiFunction<String, Integer, String>() {  
 @Override  
 public String apply(@NonNull String s, @NonNull Integer integer) throws Exception {  
 return s + integer;  
 }  
 }).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
 Log.*e*(*TAG*, "zip : accept : " + s + "\n");  
 }  
 });  
}  
  
private Observable<String> getStringObservable() {  
 return Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<String> e) throws Exception {  
 if (!e.isDisposed()) {  
 e.onNext("A");  
 Log.*e*(*TAG*, "String emit : A \n");  
 e.onNext("B");  
 Log.*e*(*TAG*, "String emit : B \n");  
 e.onNext("C");  
 Log.*e*(*TAG*, "String emit : C \n");  
 }  
 }  
 });  
}  
  
private Observable<Integer> getIntegerObservable() {  
 return Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {  
 if (!e.isDisposed()) {  
 e.onNext(1);  
 Log.*e*(*TAG*, "Integer emit : 1 \n");  
 e.onNext(2);  
 Log.*e*(*TAG*, "Integer emit : 2 \n");  
 e.onNext(3);  
 Log.*e*(*TAG*, "Integer emit : 3 \n");  
 e.onNext(4);  
 Log.*e*(*TAG*, "Integer emit : 4 \n");  
 }  
 }  
 });  
}



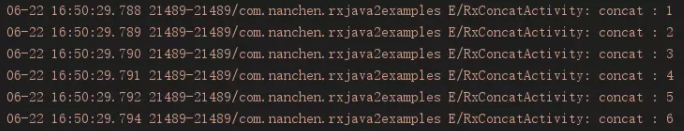
需要注意的是：zip 组合事件的过程就是分别从发射器 A 和发射器 B 各取出一个事件来组合，并且一个事件只能被使用一次，组合的顺序是严格按照事件发送的顺序来进行的，所以上面截图中，可以看到，1 永远是和 A 结合的，2 永远是和 B 结合的。

最终接收器收到的事件数量是和发送器发送事件最少的那个发送器的发送事件数目相同，所以如截图中，4 很孤单，没有人愿意和它交往，孤独终老的单身狗。

### Concat

concat 把两个发射器连接成一个发射器，且一定有顺序，必须是前一个完成后，再发送完成后一个

Observable.*concat*(Observable.*just*(1,2,3), Observable.*just*(4,5,6))  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*, "concat : "+ integer + "\n" );  
 }  
 });

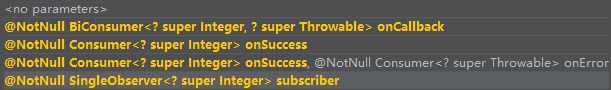


### Filter

Observable.*just*(1, 20, 65, -5, 7, 19)  
 .filter(new Predicate<Integer>() {  
 @Override  
 public boolean test(Integer integer) throws Exception {  
 return integer > 10;  
 }  
 }).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(Integer integer) throws Exception {  
 Log.*e*("", "accept : " + integer +"\n" );  
  
 }  
});

### Single

Single 只会接收一个参数，而 SingleObserver 只会调用 onError() 或者 onSuccess()。



Single.*just*(new Random().nextInt())  
 .subscribe(new SingleObserver<Integer>() {  
 @Override  
 public void onSubscribe(@NonNull Disposable d) {  
  
 }  
  
 @Override  
 public void onSuccess(@NonNull Integer integer) {  
 Log.*e*(*TAG*, "single : onSuccess : " + integer + "\n");  
 }  
  
 @Override  
 public void onError(@NonNull Throwable e) {  
 Log.*e*(*TAG*, "single : onError : " + e.getMessage() + "\n");  
 }  
 });

### distinct

去重操作符，简单的作用就是去重。

Observable.just(1, 1, 1, 2, 2, 3, 4, 5)

.distinct()

.subscribe(new Consumer<Integer>() {

@Override

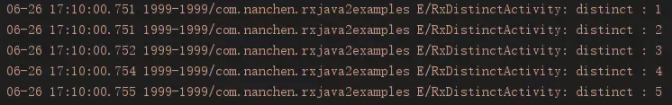
public void accept(@NonNull Integer integer) throws Exception {

mRxOperatorsText.append("distinct : " + integer + "\n");

Log.e(TAG, "distinct : " + integer + "\n");

}

});



### debounce

去除发送频率过快的项，去除的是前一个。。。

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> emitter) throws Exception {  
 emitter.onNext(1); // skip  
 Thread.*sleep*(400);  
 emitter.onNext(2); // deliver  
 Thread.*sleep*(505);  
 emitter.onNext(3); // skip  
 Thread.*sleep*(100);  
 emitter.onNext(4); // deliver  
 Thread.*sleep*(605);  
 emitter.onNext(5); // deliver  
 Thread.*sleep*(510);  
 emitter.onComplete();  
 }  
}).debounce(500, TimeUnit.*MILLISECONDS*)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*,"debounce :" + integer + "\n");  
 }  
 });



### doOnSubscribe()与Rx1.x区别不大

doOnSubscribe()是在subscribe()调用后而且在事件发送前执行，但区别在于它可以指定线程。默认情况下，doOnSubscribe()执行在subscribe()发生的线程（也就是observeOn指定的线程）；而如果在doOnSubscribe()之后有subscribeOn()的话，它将执行在离它最近的subscribeOn()所指定的线程。

datamanagerHelper  
 .doMemberInfo()  
 .compose(RxUtil.rxSchedulerHelper())  
 .compose(RxUtil.handleMyResult())  
 .doOnSubscribe **{** mView.showProgress()  
 **}** .subscribe(CommonSubscriber(mView) **{  
 it**?.*let* **{** msgStatus.set(**it**.hasMsg == "0")  
 idCardStatus.set(**it**.cardAuthentication == "0")  
 bankCardStatus.set(when (**it**.bankAuthentication) {  
 "0" -> "已认证"  
 "2" -> "待核实"  
 else -> "未认证"  
 })  
 emailStatus.set(**it**.emailAuthentication == "0")  
 userName.set(**it**.accountName)  
 headUrl.set(**it**.photoUrl)  
 **}  
 }**)

### doOnNext与Rx1.x区别不大

它的作用是让订阅者在接收到数据之前干点有意思的事情。假如我们在获取到数据之前想先保存一下它

Observable.*just*(1, 2, 3, 4)  
 .doOnNext(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*, "doOnNext 保存 " + integer + "成功" + "\n");  
 }  
 }).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*, "doOnNext :" + integer + "\n");  
 }  
});

## 线程调度相关

这一块基本没什么改动，但细心的小伙伴一定会发现，RxJava 2.x 中已经没有了 Schedulers.immediate() 这个线程环境，还有 Schedulers.test()。

subScribeOn

同 RxJava 1.x 一样，subscribeOn 用于指定 subscribe() 时所发生的线程，从源码角度可以看出，内部线程调度是通过 ObservableSubscribeOn来实现的。

@SchedulerSupport(SchedulerSupport.CUSTOM)

public final Observable<T> subscribeOn(Scheduler scheduler) {

ObjectHelper.requireNonNull(scheduler, "scheduler is null");

return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this, scheduler));

}

ObservableSubscribeOn 的核心源码在 subscribeActual 方法中，通过代理的方式使用 SubscribeOnObserver 包装 Observer 后，设置 Disposable 来将 subscribe 切换到 Scheduler 线程中。

observeOn

observeOn 方法用于指定下游 Observer 回调发生的线程。

@SchedulerSupport(SchedulerSupport.CUSTOM)

public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int bufferSize) {

ObjectHelper.requireNonNull(scheduler, "scheduler is null");

ObjectHelper.verifyPositive(bufferSize, "bufferSize");

return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this, scheduler, delayError, bufferSize));

}

线程切换需要注意的

简单地说，subscribeOn() 指定的就是发射事件的线程，observerOn 指定的就是订阅者接收事件的线程。多次指定发射事件的线程只有第一次指定的有效，也就是说多次调用 subscribeOn() 只有第一次的有效，其余的会被忽略。但多次指定订阅者接收线程是可以的，也就是说每调用一次 observerOn()，下游的线程就会切换一次。

Observable.*create*(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {  
 Log.*e*(*TAG*, "Observable thread is : " + Thread.*currentThread*().getName());  
 e.onNext(1);  
 e.onComplete();  
 }  
}).subscribeOn(Schedulers.*newThread*())  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .doOnNext(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*, "After observeOn(mainThread)，Current thread is " +

Thread.*currentThread*().getName());  
 }  
 })  
 .observeOn(Schedulers.*io*())  
 .subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer integer) throws Exception {  
 Log.*e*(*TAG*, "After observeOn(io)，Current thread is " +

Thread.*currentThread*().getName());  
 }  
 });

RxThreadActivity: Observable thread is : RxNewThreadScheduler-1

RxThreadActivity: After observeOn(mainThread)，Current thread is main

RxThreadActivity: After observeOn(io)，Current thread is RxCachedThreadScheduler-2

实例代码中，分别用 Schedulers.newThread() 和 Schedulers.io() 对发射线程进行切换，并采用 observeOn(AndroidSchedulers.mainThread() 和 Schedulers.io() 进行了接收线程的切换。可以看到输出中发射线程仅仅响应了第一个 newThread，但每调用一次 observeOn() ，线程便会切换一次，因此如果我们有类似的需求时，便知道如何处理了。

RxJava 中，已经内置了很多线程选项供我们选择，例如有：

* Schedulers.io() 代表io操作的线程, 通常用于网络,读写文件等io密集型的操作；
* Schedulers.computation() 代表CPU计算密集型的操作, 例如需要大量计算的操作；
* Schedulers.newThread() 代表一个常规的新线程；
* AndroidSchedulers.mainThread() 代表Android的主线程

这些内置的 Scheduler 已经足够满足我们开发的需求，因此我们应该使用内置的这些选项，而 RxJava 内部使用的是线程池来维护这些线程，所以效率也比较高。

## Func和Action替换为了Function和Consumer

Functions

Func1<T, R>替换为了io.reactivex.functions.Function<T, R> ，Func2<T, R>替换为了BiFunction<T, R>, 把Func3 - Func9 分别改成了 Function3 - Function9 。FuncN被Function<Object[], R>代替。

此外，操作符不再使用Func1<T, Boolean>被替换为为Predicate<T>。

Actions

无参数的Action0 被io.reactivex.functions.Action和Scheduler代替。Action1被重命名为Consumer。Action2 被重命名为BiConsumer。 ActionN 被Consumer<Object[]> 代替。

此外，为了减少组件数量，2.x中没有定义Action3-Action9和ActionN

## 一些常见用法

### Map简单的网络请求

1）通过 Observable.create() 方法，调用 OkHttp 网络请求；

2）通过 map 操作符集合 gson，将 Response 转换为 bean 类；

3）通过 doOnNext() 方法，解析 bean 中的数据，并进行数据库存储等操作；

4）调度线程，在子线程中进行耗时操作任务，在主线程中更新 UI ；

5）通过 subscribe()，根据请求成功或者失败来更新 UI 。

Observable  
 .*create*(new ObservableOnSubscribe<Response>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Response> e) throws Exception {  
 Builder builder = new Builder()  
 .url("http://api.avatardata.cn/LookUp?key=ec47be&mobileNumber=13021671512")  
 .get();  
 Request request = builder.build();  
 Call call = new OkHttpClient().newCall(request);  
 Response response = call.execute();  
 e.onNext(response);  
 }  
 })  
 .map(new Function<Response, MobileAddress>() {  
 @Override  
 public MobileAddress apply(@NonNull Response response) throws Exception {  
 Log.*e*(*TAG*, "map 线程:" + Thread.*currentThread*().getName() + "\n");  
 if (response.isSuccessful()) {  
 ResponseBody body = response.body();  
 if (body != null) {  
 Log.*e*(*TAG*, "map:转换前:" + response.body());  
 return new Gson().fromJson(body.string(), MobileAddress.class);  
 }  
 }  
 return null;  
 }  
 })  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .doOnNext(new Consumer<MobileAddress>() {  
 @Override  
 public void accept(@NonNull MobileAddress s) throws Exception {  
 Log.*e*(*TAG*, "doOnNext 线程:" + Thread.*currentThread*().getName() + "\n");  
 Log.*e*(*TAG*, "doOnNext: 保存成功：" + s.toString() + "\n");  
  
 }  
 })  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<MobileAddress>() {  
 @Override  
 public void accept(@NonNull MobileAddress data) throws Exception {  
 Log.*e*(*TAG*, "subscribe 线程:" + Thread.*currentThread*().getName() + "\n");  
 Log.*e*(*TAG*, "成功:" + data.toString() + "\n");  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(@NonNull Throwable throwable) throws Exception {  
 Log.*e*(*TAG*, "subscribe 线程:" + Thread.*currentThread*().getName() + "\n");  
  
 Log.*e*(*TAG*, "失败：" + throwable.getMessage() + "\n");  
 }  
 });

### flatMap多个网络请求依次依赖

想必这种情况也在实际情况中比比皆是，例如用户注册成功后需要自动登录，我们只需要先通过注册接口注册用户信息，注册成功后马上调用登录接口进行自动登录即可。

我们的 flatMap 恰好解决了这种应用场景

Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")  
 .addQueryParameter("rows", 1 + "")  
 .build()  
 .getObjectObservable(FoodList.class)  
 .subscribeOn(Schedulers.*io*()) // 在io线程进行网络请求  
 .observeOn(AndroidSchedulers.*mainThread*()) // 在主线程处理获取食品列表的请求结果  
 .doOnNext(new Consumer<FoodList>() {  
 @Override  
 public void accept(@NonNull FoodList foodList) throws Exception {  
 // 先根据获取食品列表的响应结果做一些操作  
 Log.*e*(*TAG*, "accept: doOnNext :" + foodList.toString());  
 }  
 })  
 .observeOn(Schedulers.*io*()) // 回到 io 线程去处理获取食品详情的请求  
 .flatMap(new Function<FoodList, ObservableSource<FoodDetail>>() {  
 @Override  
 public ObservableSource<FoodDetail> apply(@NonNull FoodList foodList) throws Exception {  
 if (foodList != null && foodList.getTngou() != null && foodList.getTngou().size() > 0) {  
 return Rx2AndroidNetworking.post("http://www.tngou.net/api/food/show")  
 .addBodyParameter("id", foodList.getTngou().get(0).getId() + "")  
 .build()  
 .getObjectObservable(FoodDetail.class);  
 }  
 return null;  
  
 }  
 })  
 .observeOn(AndroidSchedulers.*mainThread*())// 在主线程处理获取食品详情的请求结果  
 .subscribe(new Consumer<FoodDetail>() {  
 @Override  
 public void accept(@NonNull FoodDetail foodDetail) throws Exception {  
 Log.*e*(*TAG*, "accept: success ：" + foodDetail.toString());  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(@NonNull Throwable throwable) throws Exception {  
 Log.*e*(*TAG*, "accept: error :" + throwable.getMessage());  
 }  
 });

### 注意Map和FlatMap的区别

对比方法可发现：map接收的是R，flatMap接受的是ObservableSource<? extends R>>

public final <R> Observable<R> map(Function<? super T, ? extends R> mapper) { ... }

public final <R> Observable<R> flatMap(Function<? super T, ? extends ObservableSource<? extends R>> mapper) { ... }

.map(new Function<Integer, String>() {//返回值是Observable<String>  
 @Override  
 public String apply(Integer integer) throws Exception {  
 return "This is result " + integer;  
 }  
}

**------------------------------------------------------------------------------------------**

.flatMap(new Function<Integer, ObservableSource<String>>() { //返回值是Observable<String>  
 @Override  
 public ObservableSource<String> apply(Integer integer) throws Exception {  
 return Observable.*create*(new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter) throws Exception {  
 emitter.onNext("I am value " + integer);  
 }  
 });  
 }  
}

所以上面的flatmap例子map不好解决，除非请求详情接口返回FoodDetail而不是Observable<FoodDetail>,这样，通过map转换后，FoodDetail变为Observable<FoodDetail>而不是Observable<Observable<FoodDetail>>

map主要是用来转换结果的，而flatMap主要是用来转换被观察者的

如果强行使用map转：比较复杂，而且还是免不了使用flatMap：

if (true) {  
 datamanagerHelper  
 .doLogin(LoginReq(account.get()!!, passWord.get()!!))  
 .compose(RxUtil.rxSchedulerHelper())  
 .compose(RxUtil.handleMyResult())  
 .doOnNext **{  
 it**.*let* **{** datamanagerHelper.saveAccoutId(**it**.accountId)  
 **}  
 }** .observeOn(Schedulers.io())  
 .flatMap **{** datamanagerHelper.doMemberInfo() **}** .observeOn(AndroidSchedulers.mainThread())  
 .compose(RxUtil.handleMyResult())  
 .subscribe(CommonSubscriber(mView) **{  
 it**.*let* **{**  
 ToastUtil.shortShow("登录成功！")  
 **}  
 }**)  
} else {  
 datamanagerHelper  
 .doLogin(LoginReq(account.get()!!, passWord.get()!!))  
 .compose(RxUtil.rxSchedulerHelper())  
 .compose(RxUtil.handleMyResult())  
 .doOnNext**{  
 it**.*let* **{** datamanagerHelper.saveAccoutId(**it**.accountId)  
 **}  
 }** .subscribeOn(Schedulers.io())  
 .map **{** datamanagerHelper.doMemberInfo().compose(RxUtil.handleMyResult()) **}** .flatMap **{ it }** .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(CommonSubscriber(mView) **{  
 it**.*let* **{** ToastUtil.shortShow("登录成功！")  
 **}  
 }**)  
}

### Concat先读取缓存，如果缓存没数据再通过网络请求

想必在实际应用中，很多时候（对数据操作不敏感时）都需要我们先读取缓存的数据，如果缓存没有数据，再通过网络请求获取，随后在主线程更新我们的 UI。concat 操作符简直就是为我们这种需求量身定做。

concat 可以做到不交错的发射两个甚至多个 Observable 的发射事件，并且只有前一个 Observable 终止( onComplete() ) 后才会定义下一个 Observable。

利用这个特性，我们就可以先读取缓存数据，倘若获取到的缓存数据不是我们想要的，再调用 onComplete() 以执行获取网络数据的 Observable，如果缓存数据能应我们所需，则直接调用 onNext() ，防止过度的网络请求

private Observable<FoodList> getCacheObservable() {  
 return Observable.*create*((ObservableOnSubscribe<FoodList>) e -> {  
 FoodList data = new CacheManager().getFoodListData();  
 if (data != null) { // 如果缓存数据不为空，则直接读取缓存数据，而不读取网络数据  
 isFromNet = false;  
 runOnUiThread(new Runnable() {  
 @Override  
 public void run() {  
 Log.*e*(*TAG*, "\nsubscribe: 读取缓存数据:");  
 }  
 });  
 // 在操作符 concat 中，调用 onNext 之后不会执行下一个Observable  
 e.onNext(data);  
 } else {  
 isFromNet = true;  
 runOnUiThread(new Runnable() {  
 @Override  
 public void run() {  
 Log.*e*(*TAG*, "\nsubscribe: 读取网络数据:");  
 }  
 });  
 // 在操作符 concat 中，只有调用 onComplete 之后才会执行下一个 Observable  
 e.onComplete();  
 }  
 });  
}  
  
private Observable<FoodList> getNetwordObservable() {  
 return Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")  
 .addQueryParameter("rows", 10 + "")  
 .build()  
 .getObjectObservable(FoodList.class);  
}  
  
private void test2() {  
 Observable<FoodList> cache = getCacheObservable();  
 Observable<FoodList> network = getNetwordObservable();  
 // 两个 Observable 的泛型应当保持一致<FoodList>  
 Observable.*concat*(cache, network)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<FoodList>() {  
 @Override  
 public void accept(@NonNull FoodList foodList) throws Exception {  
 if (isFromNet) {  
 Log.*e*(*TAG*, "accept : 网络获取数据设置缓存: \n" + foodList.toString());  
 }  
 Log.*e*(*TAG*, "accept: 读取数据成功:" + foodList.toString());  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(@NonNull Throwable throwable) throws Exception {  
 Log.*e*(*TAG*, "accept: 读取数据失败：" + throwable.getMessage());  
 }  
 });  
}

有时候我们的缓存可能还会分为 memory 和 disk ，实际上都差不多，无非是多写点 Observable ，然后通过 concat 合并即可。

### Zip结合多个接口的数据更新 UI

在实际应用中，我们极有可能会在一个页面显示的数据来源于多个接口，这时候我们的 zip 操作符为我们排忧解难。zip 操作符可以将多个 Observable 的数据结合为一个数据源再发射出去。

Observable<MobileAddress> MobileAddressOb = Rx2AndroidNetworking.get("http://api.avatar1")  
 .build()  
 .getObjectObservable(MobileAddress.class);  
  
Observable<CategoryResult> CategoryResultOb = Rx2AndroidNetworking.get("http://api.avatar2")  
 .build()  
 .getObjectObservable(CategoryResult.class);  
  
Observable.*zip*(MobileAddressOb, CategoryResultOb, new BiFunction<MobileAddress, CategoryResult, String>() {  
 @Override  
 public String apply(@NonNull MobileAddress mobileAddress, @NonNull CategoryResult categoryResult) throws Exception {  
 return "合并后的数据为：手机归属地：" + mobileAddress.getMobilearea() + "人名：" + categoryResult.getName();  
 }  
}).subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*())  
 .subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
 Log.*e*(*TAG*, "accept: 成功：" + s + "\n");  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(@NonNull Throwable throwable) throws Exception {  
 Log.*e*(*TAG*, "accept: 失败：" + throwable + "\n");  
 }  
 });

### Interval间隔任务实现心跳

想必即时通讯等需要轮训的任务在如今的 APP 中已是很常见，而 RxJava 2.x 的 interval 操作符可谓完美地解决了我们的疑惑。这里就简单的意思一下轮训

private static Disposable *mDisposable*;

private static void doSomeThing() {  
 *mDisposable* = Observable.*interval*(3, 2, TimeUnit.*SECONDS*)  
 .subscribeOn(Schedulers.*io*())  
 .observeOn(AndroidSchedulers.*mainThread*()) // timer 默认在新线程，所以需要切换回主线程  
 .subscribe(new Consumer<Long>() {  
 @Override  
 public void accept(@NonNull Long aLong) throws Exception {  
 Log.*e*(*TAG*, "timer :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");  
 }  
 });  
}

@Override  
protected void onDestroy() {  
 super.onDestroy();  
 if (*mDisposable* != null && !*mDisposable*.isDisposed()) {  
 *mDisposable*.dispose();  
 }  
}

## 常见崩溃

### 如果上游发送了onError，下游没有处理，则会报错：

错误信息：at io.reactivex.internal.functions.Functions$OnErrorMissingConsumer.accept(Functions.java:)

Observable.*create*((ObservableOnSubscribe<String>) e -> {  
 e.onNext("1");  
 e.onError(new ApiException("ff"));  
}).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) {  
 Log.*e*("", "accept : " + s + "\n");  
 }  
});

解决办法下游必须处理onError

doOnNext(new Consumer(){ ... }, new Consumer(){ ... })或者doOnNext(new Observer(){ ... })