**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. 优秀框架 4](#_Toc17003)

[2. EventBus 5](#_Toc19568)

[2.1. 依赖 5](#_Toc30675)

[2.2. 创建事件实体类 5](#_Toc24258)

[2.3. 注册为订阅者以及注销 5](#_Toc24158)

[2.4. 声明订阅方法 5](#_Toc13124)

[2.5. 发送事件 6](#_Toc9876)

[2.6. 进一步认识@Subscribe注解 6](#_Toc28427)

[2.6.1. ThreadMode 订阅发生的运行线程 7](#_Toc1316)

[2.6.2. priority 优先级 7](#_Toc12365)

[2.6.3. sticky 粘性事件 7](#_Toc29346)

[3. DataBinding 7](#_Toc31651)

[3.1. MVVM 7](#_Toc24144)

[3.1.1. 概述 7](#_Toc13351)

[3.1.2. MVC、MVP、MVVM 7](#_Toc5180)

[3.1.2.1. MVC 7](#_Toc32161)

[3.1.2.2. MVP 7](#_Toc22834)

[3.1.2.3. MVVM 8](#_Toc8497)

[3.2. 初识DataBinding 8](#_Toc4945)

[3.2.1. 简单使用 8](#_Toc4105)

[3.2.2. 解惑 9](#_Toc19531)

[3.2.2.1. 为什么配置了dataBinding{enabled = true}之后就可以使用dataBinding 9](#_Toc15670)

[3.2.2.2. ActivityMainBinding这个类从哪来的 9](#_Toc658)

[3.2.2.3. DataBinding与ButterKnife的区别 10](#_Toc3730)

[3.3. Binding类的其他生成方式 10](#_Toc10499)

[3.4. DataBinding的基本用法 10](#_Toc26189)

[3.4.1. 控件有id时，可以直接使用 10](#_Toc28876)

[3.4.2. 在xml中引入一些基础变量Variables 10](#_Toc4525)

[3.4.3. 引入一些高级变量Variables 11](#_Toc6047)

[3.4.4. xml中引用表达式 12](#_Toc3837)

[3.4.5. 设置别名alias 13](#_Toc1141)

[3.4.6. include中的使用 13](#_Toc27586)

[3.4.7. 事件处理 14](#_Toc3949)

[3.4.7.1. 布局中引入OnClickListener的变量 15](#_Toc30085)

[3.4.7.2. 方法调用 15](#_Toc3005)

[3.4.7.3. Lambda表达式 16](#_Toc16193)

[3.5. 值变化通知UI变化 17](#_Toc19227)

[3.5.1. Observable Objects 17](#_Toc22919)

[3.5.2. Observable Fields 17](#_Toc26825)

[3.5.2.1. 简单的Observable Fields 17](#_Toc23077)

[3.5.2.2. 对象的Observable Fields 17](#_Toc31892)

[3.5.2.3. 注意，尽量给ObservableField创建public final 18](#_Toc30191)

[3.5.3. Obseravble Collection 18](#_Toc23387)

[3.5.3.1. ObservableArrayMap 18](#_Toc7245)

[3.5.3.2. ObservableArrayList 18](#_Toc5316)

[3.6. 双向绑定 18](#_Toc3246)

[3.6.1. xml中的用法 18](#_Toc1907)

[3.6.2. @InverseBindingAdapter和死循环 18](#_Toc7241)

[3.6.3. 不是所有属性都能双向绑定 18](#_Toc2118)

[3.7. 监听属性变更 19](#_Toc22276)

[3.8. 表达式链 19](#_Toc15391)

[3.9. 动画 19](#_Toc15486)

[3.10. RecyclerView中使用DataBinding 19](#_Toc317)

[3.10.1. ViewHolder 19](#_Toc8853)

[3.10.2. onCreateViewHolder 20](#_Toc27712)

[3.10.3. onBindViewHolder 20](#_Toc20695)

[3.10.4. 公用RecycleView的Adapter和ViewHolder 21](#_Toc15932)

[3.11. @BindingAdapter设置自定义属性 22](#_Toc17600)

[3.11.1. 自定义属性 22](#_Toc9360)

[3.11.2. 多个自定义属性 22](#_Toc19640)

[3.11.3. 使用系统的属性，但是逻辑自定义 23](#_Toc22627)

[3.11.4. 多属性 23](#_Toc22572)

[3.12. DataBindingComponent实现数据 23](#_Toc8932)

[3.12.1. 实现测试数据、正式数据分离 23](#_Toc3380)

[3.12.2. 作为参数传给静态的BindingAdapter方法（未实现） 25](#_Toc13515)

[3.12.3. 提取共同逻辑 25](#_Toc30883)

[3.13. @BindingConversion设置转换值 25](#_Toc29359)

[3.14. Databinding插件 25](#_Toc4991)

[3.14.1. Databinding Support 25](#_Toc9509)

[3.14.2. Data Binding Formatter 26](#_Toc28964)

[3.14.3. CreateAdapterPlugin 26](#_Toc13000)

[3.15. DataBinding实用指南 26](#_Toc17318)

[3.15.1. 统一命名的variable 26](#_Toc31645)

[3.15.2. 尽可能少的variable和import 26](#_Toc4147)

[3.15.3. 避免使用复杂的表达式 27](#_Toc26733)

[3.15.4. 点击事件的命名和处理 27](#_Toc31613)

[3.15.5. 处理RecyclerView 列表项的数据及点击事件 29](#_Toc5637)

[3.15.6. 补全自定义的属性 29](#_Toc5748)

[3.15.7. 错误排查 30](#_Toc903)

[3.16. 坑集合 30](#_Toc5593)

[3.16.1. 变量如果引用了没有的成员 30](#_Toc9263)

[3.16.2. include和BindingAdapter中，img和非引用设值的处理 30](#_Toc29560)

[3.16.3. As升级3.2以后，报错 couldn't make a guess XXX 31](#_Toc30767)

[4. Aspect面向切面编程 31](#_Toc11776)

[4.1. 闲谈AOP 31](#_Toc11674)

[4.2. 没有AOP的例子 32](#_Toc17623)

[4.3. AspectJ介绍 33](#_Toc4466)

[4.3.1. AspectJ极简介 33](#_Toc14174)

[4.3.2. AspectJ语法 34](#_Toc6737)

[4.3.2.1. Join Points介绍 34](#_Toc19327)

[4.3.2.2. Pointcuts介绍 37](#_Toc24913)

[4.3.2.2.1. 一个Pointcuts例子 37](#_Toc15738)

[4.3.2.2.2. 语法规则 38](#_Toc17169)

[4.3.2.2.3. 直接针对JoinPoint的选择 38](#_Toc19275)

[4.3.2.2.4. 间接针对JPoint的选择 40](#_Toc29196)

[4.3.3. advice和aspect介绍 41](#_Toc24616)

[4.3.4. 回顾前面三节的内容 42](#_Toc19626)

[4.3.5. 参数传递和JPoint信息 43](#_Toc2264)

[4.3.5.1. 参数传递 43](#_Toc7329)

[4.3.5.2. JoinPoint信息收集 43](#_Toc24575)

[4.4. 使用AOP的例子 44](#_Toc21159)

[4.4.1. 打印Log 44](#_Toc22932)

[4.4.2. 检查权限 45](#_Toc10383)

[4.4.2.1. 使用注解 45](#_Toc20237)

[4.4.2.2. 检查权限 45](#_Toc9521)

[4.4.2.3. 和其他模块交互 46](#_Toc31255)

[4.5. 其他、总结和参考文献 46](#_Toc6367)

[4.5.1. AspectJ编译 46](#_Toc14794)

[4.6. 按钮防颤的例子 48](#_Toc7729)

[4.6.1. project.build.gradle 48](#_Toc7863)

[4.6.2. app.build.gradle 48](#_Toc19045)

[4.6.3. ids.xml 49](#_Toc23627)

[4.6.4. SingleClick 50](#_Toc20917)

[4.6.4.1. kotlin 50](#_Toc29270)

[4.6.4.2. Java 50](#_Toc18244)

[4.6.5. SingleClickAspect 50](#_Toc14264)

[4.6.5.1. kotlin 50](#_Toc7573)

[4.6.5.2. Java 51](#_Toc30591)

[4.6.6. 使用 52](#_Toc29616)

[4.7. 注意：lambda表达式无法拦截 52](#_Toc7168)

[4.7.1. 方案一，不要用lambda表达式 52](#_Toc23638)

[4.7.2. 方案二,mvvm中，没有lambda 53](#_Toc32690)

[4.7.3. 方案三，复写要响应onclick的控件 53](#_Toc30712)

[5. Volley的使用 53](#_Toc31179)

[5.1. 在应用里配置，全局化它 53](#_Toc16206)

[5.2. StringRequest，JsonObj,JsonArray的Get请求方法 54](#_Toc11515)

[5.3. StringRequest的Post请求方法，重点部分： 54](#_Toc3011)

[5.4. JsonObj的Post请求方法，重点部分： 54](#_Toc4273)

[5.5. 与Activity生命周期联动 54](#_Toc5129)

[5.6. 二次封装 55](#_Toc12745)

[5.7. 用ImageRequest请求网络图片 55](#_Toc1689)

[5.8. 用ImageLoader请求带缓存的网络图片 56](#_Toc19881)

[5.9. 用NetworkImageView控件显示网络图片 56](#_Toc30956)

[5.10. GsonVolley构造 57](#_Toc17150)

[6. 表格框架SortableTableView 58](#_Toc22950)

[6.1. 引用 58](#_Toc28100)

[6.2. 初始化控件 59](#_Toc17688)

[6.2.1. Layout中 59](#_Toc28739)

[6.2.2. 代码中 59](#_Toc22203)

[6.3. 表格宽 59](#_Toc14619)

[6.4. 显示数据 59](#_Toc10130)

[6.5. 排序 60](#_Toc22174)

[6.6. 空数据提示（未实现） 60](#_Toc26774)

[6.7. 表头 61](#_Toc27141)

[6.8. 单击监听 61](#_Toc16800)

[6.8.1. Data Click Listening 61](#_Toc24902)

[6.8.2. Long Data Click Listening 61](#_Toc8291)

[6.8.3. Header Click Listening 61](#_Toc5262)

[6.9. 样式Style 61](#_Toc16502)

[6.9.1. Header Styling 61](#_Toc3254)

[6.9.2. 排序图标 62](#_Toc15491)

[6.9.3. 奇偶行背景 62](#_Toc8741)

[6.9.4. 分隔线 62](#_Toc14865)

[7. 数据库的使用 62](#_Toc24576)

[7.1. 存放目录和开启事务 62](#_Toc3211)

[7.2. 来个代码 63](#_Toc13717)

[7.3. 查询方法详解 65](#_Toc20836)

[8. 数据库框架greenDAO3.0 66](#_Toc4196)

[8.1. 引入greenDAO 66](#_Toc22160)

[8.2. 数据库初始化 67](#_Toc2871)

[8.3. 添加数据 67](#_Toc26034)

[8.4. 查询数据 67](#_Toc801)

[8.5. 删除数据 67](#_Toc8610)

[8.6. 修改数据 68](#_Toc10597)

[8.7. 数据库升级 68](#_Toc20795)

[8.7.1. 修改gradle文件 68](#_Toc8587)

[8.7.2. 修改实体类 68](#_Toc2454)

[9. Realm详解 69](#_Toc30001)

[9.1. Realm介绍 69](#_Toc31514)

[9.1.1. 介绍 69](#_Toc13150)

[9.1.2. 优势 69](#_Toc9)

[9.1.3. 条件 69](#_Toc32599)

[9.2. 使用 69](#_Toc16494)

[9.2.1. 加入依赖 69](#_Toc13138)

[9.2.2. 创建model 70](#_Toc7841)

[9.2.3. 初始化 71](#_Toc7537)

[9.2.4. 关闭Realm 71](#_Toc30904)

[9.2.5. 增 72](#_Toc20975)

[9.2.6. 查 74](#_Toc28310)

[9.2.7. 改 76](#_Toc9066)

[9.2.8. 删 76](#_Toc6532)

[9.2.9. 版本升级 77](#_Toc6819)

[9.2.10. 加密 79](#_Toc18801)

[9.2.11. 适配器（Adapter）(官方原文） 79](#_Toc20789)

[9.2.12. Intents（官方原文） 79](#_Toc15692)

[9.2.13. RxJava 80](#_Toc28434)

[10. WebSocket 80](#_Toc14100)

[10.1. 背景 80](#_Toc18623)

[10.2. websocket与http 80](#_Toc4505)

[10.3. Websocket是什么样的协议，具体有什么优点 80](#_Toc19629)

[10.4. 使用 82](#_Toc27899)

[10.4.1. 流程 82](#_Toc8226)

[10.4.1.1. 用户登录流程 82](#_Toc28779)

[10.4.1.2. 客户端发送请求流程 83](#_Toc4230)

[10.4.1.3. 服务端主动推送消息流程 84](#_Toc21185)

[10.4.2. 具体实现 84](#_Toc25627)

[10.4.2.1. 依赖和管理类单例 84](#_Toc10459)

[10.4.2.2. 建立连接 85](#_Toc11618)

[10.4.2.3. 重连 87](#_Toc12262)

[10.4.2.3.1. 收到连接失败或者连接断开事件的时候进行重连，实现了需要重连的情景3 87](#_Toc7870)

[10.4.2.3.2. 网络的切换通过广播来监听实现重连，实现实现情景1 89](#_Toc15103)

[10.4.2.3.3. 应用回到前台情况的重连，实现情景2 90](#_Toc2064)

[10.4.2.4. 客户端发送请求 93](#_Toc26319)

[10.4.2.5. 超时和回调的处理 96](#_Toc13928)

[10.4.2.5.1. 中间层回调定义如下 96](#_Toc2830)

[10.4.2.5.2. Handler中获取的Callback与Data包装类如下 96](#_Toc28154)

[10.4.2.5.3. 超时后的处理 97](#_Toc24846)

[10.4.2.5.4. callback包装类 98](#_Toc14582)

[10.4.2.5.5. 实现添加超时任务代码 99](#_Toc20420)

[10.4.2.6. 服务端响应的处理 100](#_Toc23034)

[10.4.2.7. 授权 104](#_Toc4324)

[10.4.2.8. 心跳和同步 105](#_Toc16392)

[10.4.2.9. 服务端主动通知 107](#_Toc31599)

[11. Jsoup 109](#_Toc11268)

[11.1. 入门 109](#_Toc5272)

[11.1.1. 解析和遍历一个HTML文档 109](#_Toc17357)

[11.2. 输入 110](#_Toc13199)

[11.2.1. 解析一个HTML字符串 110](#_Toc922)

[11.2.2. 解析一个body片断 110](#_Toc12081)

[11.2.3. 从一个URL加载一个Document 110](#_Toc11318)

[11.2.4. 从一个文件加载一个文档 111](#_Toc31999)

[11.3. 数据抽取 111](#_Toc7564)

[11.3.1. 使用DOM方法来遍历一个文档 111](#_Toc986)

[11.3.2. 使用选择器语法来查找元素 112](#_Toc14077)

[11.3.3. 从元素抽取属性，文本和HTML 113](#_Toc29089)

[11.3.4. 处理相对路径URLs 114](#_Toc29440)

[11.4. 数据修改 114](#_Toc4787)

[11.4.1. 设置属性的值 114](#_Toc24853)

[11.4.2. 设置一个元素的HTML内容 114](#_Toc25261)

[11.4.3. 设置元素的文本内容 115](#_Toc30750)

[11.5. HTML清理 115](#_Toc2002)

[11.5.1. 消除不受信任的HTML (来防止XSS攻击) 115](#_Toc19598)

[11.6. 实例 115](#_Toc14083)

[12. XPosed 118](#_Toc20759)

[12.1. Xposed是什么 118](#_Toc11317)

[12.2. 大概简述下Xposed的原理 118](#_Toc15802)

[12.3. 相关文档： 118](#_Toc959)

[12.4. 创建一个Xposed工程 119](#_Toc22630)

[12.5. Xposed API相关介绍 120](#_Toc25339)

[12.5.1. 例子里用到的姿势点 120](#_Toc29582)

[12.5.2. 补充姿势点 122](#_Toc4878)

[12.6. XposeHelpers提供了一些辅助方法 123](#_Toc23126)

[13. 鲁班Luban压缩 124](#_Toc1748)

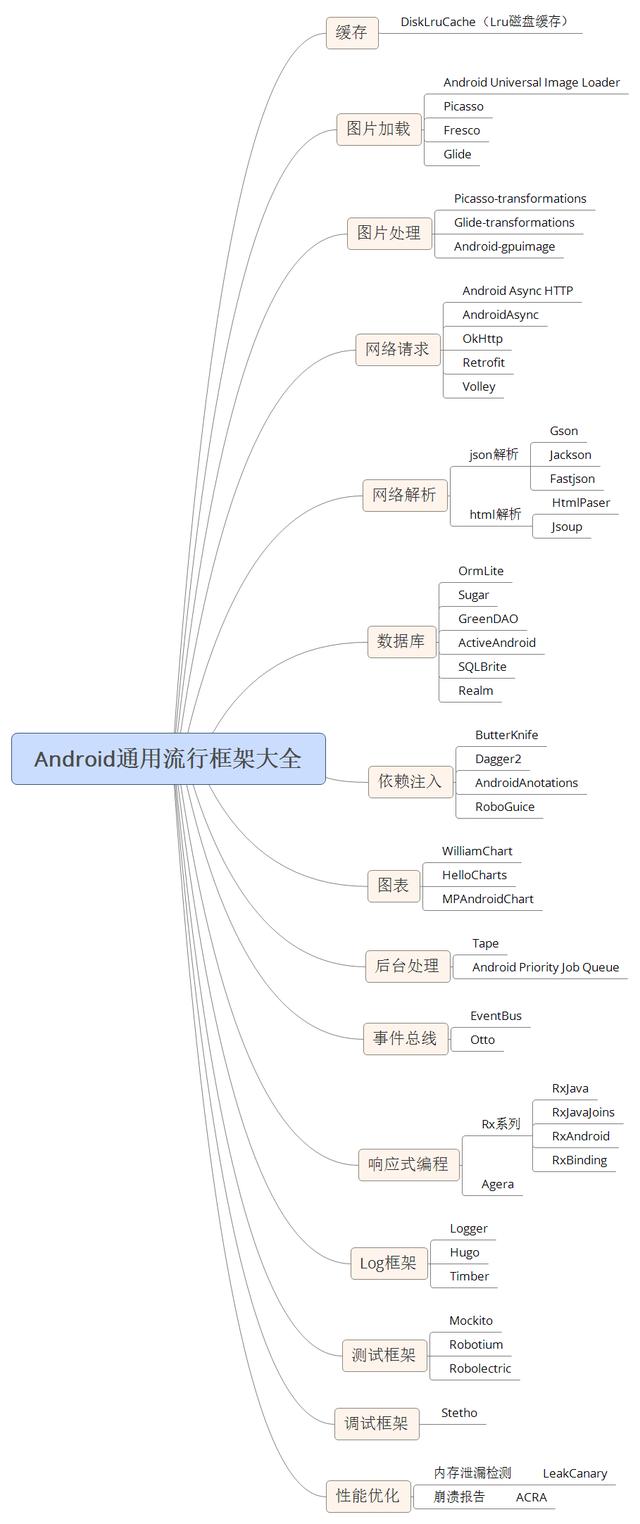
[13.1. 导入 124](#_Toc24394)

[13.2. 方法列表 124](#_Toc10692)

[13.3. 异步调用 124](#_Toc15287)

[13.4. 同步调用 124](#_Toc32605)

# 优秀框架



# EventBus

Eventbus是一款针对Android优化的发布/订阅事件总线，主要功能是替代 Handler、BroadCast在Fragment,、Activity、Service线程之间传递消息将发送者和接收者解耦。一个个事件(event)发送到总线上，然后EventBus根据已注册的订阅者(subscribers)来匹配相应的事件，进而把事件传递给订阅者，这也是观察者模式的一个最佳实践

## 依赖

compile 'org.greenrobot:eventbus:3.1.1'。官网：<https://github.com/greenrobot/EventBus>

## 创建事件实体类

所谓的事件实体类，就是传递的事件，一个组件向另一个组件发送的信息可以储存在一个类中，该类就是一个事件，会被EventBus发送给订阅者

public class MyBusEvent<T> implements Serializable {

private String eventType; //事件类型

private T t; //传输的对象;

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;

}

}

## 注册为订阅者以及注销

按需求在onCreat、onStart中注册，在onStop、onDestroy中注销。

EventBus.getDefault().register(this);即可将当前类注册，成为订阅者，即对应观察者模式的“观察者”，一旦有事件发送过来，该观察者就会接收到匹配的事件。通常，在类的初始化时便进行注册。

EventBus.getDefault().unregister(this);当订阅者不再需要接受事件的时候，我们需要解除注册，释放内存

## 声明订阅方法

在EventBus 3.0中，声明一个订阅方法需要用到@Subscribe注解，因此在订阅者类中添加一个有着@Subscribe注解的方法即可，方法名字可自定义，而且必须是public权限，其方法参数有且只能有一个，如下所示：

@Subscribe

public void onEvent(MyBusEvent event) {

/\* Do something \*/

}

完整的MainActivity.java文件如下所示：

public class MainActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

...

//注册成为订阅者

EventBus.getDefault().register(this);

}

//订阅方法，当接收到事件的时候，会调用该方法

@Subscribe(threadMode = ThreadMode.MAIN)

public void onEvent(MyBusEvent messageEvent){

if (messageEvent.getEventType().equals(Constants.EventBusType.LOGIN)) {

String code = ((LoginActivity.Bean) messageEvent.getT()).getCode();

ToastUtil.shortShow(code);

}

}

@Override

protected void onDestroy() {

super.onDestroy();

//解除注册

EventBus.getDefault().unregister(this);

}

}

## 发送事件

与观察者模式对应的，当有事件发生，需要通知观察者的时候，被观察者会调用notifyObservers()方法来通知所有已经注册的观察者，在EventBus中，对观察者模式底层进行了封装，我们只需要调用以下代码就能把事件发送出去：

EventBus.getDefault().post(MyEventType eventType);

例如：

MyBusEvent messageEvent = new MyBusEvent();

messageEvent.setEventType(Constants.EventBusType.LOGIN);

messageEvent.setT(new LoginActivity.Bean("200" , 1));

EventBus.getDefault().post(messageEvent);

## 进一步认识@Subscribe注解

回头看看上面的订阅方法，添加了@Subscribe注解，该注解标识了当前方法为订阅方法，我们可以看到上面我们还给该注解赋值(threadMode = ThreadMode.MAIN)，那么，这个代表了什么意思呢？首先，看看@Subscribe的类文件：

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

public @interface Subscribe {

ThreadMode threadMode() default ThreadMode.POSTING;

/\*\*

\* If true, delivers the most recent sticky event (posted with

\* {@link EventBus#postSticky(Object)}) to this subscriber (if event available).

\*/

boolean sticky() default false;

/\*\* Subscriber priority to influence the order of event delivery.

\* Within the same delivery thread ({@link ThreadMode}), higher priority subscribers will receive events before

\* others with a lower priority. The default priority is 0. Note: the priority does \*NOT\* affect the order of

\* delivery among subscribers with different {@link ThreadMode}s!

\*/

int priority() default 0;

}

该注解内部有三个成员，分别是threadMode、sticky、priority。threadMode代表订阅方法所运行的线程，sticky代表是否是粘性事件，priority代表优先级。给这个三个成员赋不同的值，能使得订阅方法有着不同的效果。

### ThreadMode 订阅发生的运行线程

ThreadMode是一个枚举类型，默认是使用POSTING的，如果需要更改设置，可以在添加注解的时候同时赋值。有着以下几个类型：

public enum ThreadMode {

POSTING,

MAIN,

BACKGROUND,

ASYNC

}

* POSTING:表示订阅方法运行在：发送事件的线程。
* MAIN：表示订阅方法运行在：UI线程，由于UI线程不能阻塞，因此当使用MAIN的时候，订阅方法不应该耗时过长。
* BACKGROUND：表示订阅方法运行在：后台线程，如果发送的事件线程不是UI线程，那么就使用该线程；如果发送事件的线程是UI线程，那么新建一个后台线程来调用订阅方法。
* ASYNC：订阅方法与发送事件始终不在同一个线程，即订阅方法始终会使用新的线程来运行。

### priority 优先级

设置该优先级的目的是，当一个事件有多个订阅者的时候，优先级高的会优先接收到事件。

### sticky 粘性事件

在注册之前便把事件发生出去，等到注册之后便会收到最近发送的粘性事件（必须匹配）。注意：只会接收到最近发送的一次粘性事件，之前的会接受不到。下列将只收到"粘性事件2"

EventBus.getDefault().postSticky(new MessageEvent("粘性事件1"));

EventBus.getDefault().postSticky(new MessageEvent("粘性事件2"));

@Subscribe(sticky = true)

public void onEvent(MessageEvent messageEvent){}

# DataBinding

## MVVM

### 概述

Databinding 是一种框架，MVVM是一种模式，两者的概念是不一样的。我的理解DataBinding是一个实现数据和UI绑定的框架，只是一个实现MVVM模式的工具。ViewModel和View可以通过DataBinding来实现单向绑定和双向绑定，这套UI和数据之间的动态监听和动态更新的框架Google已经帮我们做好了。在MVVM模式中ViewModel和View是用绑定关系来实现的，所以有了DataBinding 使我们构建Android MVVM 应用程序成为可能

### MVC、MVP、MVVM

首先，我们先大致了解Android开发中常见的模式,以便我们更深入了解MVVM 模式。

#### MVC

* View：对应于xml布局文件
* Model：实体模型
* Controllor：对应于Activity业务逻辑，数据处理和UI处理

从上面看起来各个组件的职责视乎还挺耦合MVC的，但是打开Android的一个Activity文件，一看一言难尽， Android中经常会出现数千行的Activity代码，究其原因，Android中纯粹作为View的各个XML视图功能太弱，Activity基本上都是View和Controller的合体，既要负责视图的显示又要加入控制逻辑，承担的功能过多，代码量大也就不足为奇。所有更贴切的目前常规的开发说应该是View-Model 模式，大部分都是通过Activity的协调，连接，和处理逻辑的。

#### MVP

* View: 对应于Activity和xml，负责View的绘制以及与用户交互
* Model: 依然是实体模型
* Presenter: 负责完成View于Model间的交互和业务逻辑

在Android开发中MVP的设计思想用得比较多，利用MVP的设计模型可以把部分的逻辑的代码从Fragment和Activity业务的逻辑移出来，在Presenter中持有View（Activity或者Fragment）的引用，然后在Presenter调用View暴露的接口对视图进行操作，这样有利于把视图操作和业务逻辑分开来。MVP能够让Activity成为真正的View而不是View和Control的合体，Activity只做UI相关的事。但是这个模式还是存在一些不好的地方，比较如说：

1. Activity需要实现各种跟UI相关的接口，同时要在Activity中编写大量的事件，然后在事件处理中调用presenter的业务处理方法，View和Presenter只是互相持有引用并互相做回调,代码不美观。
2. 这种模式中，程序的主角是UI，通过UI事件的触发对数据进行处理，更新UI就有考虑线程的问题。而且UI改变后牵扯的逻辑耦合度太高，一旦控件更改（比较TextView 替换 EditText等）牵扯的更新UI的接口就必须得换。
3. 复杂的业务同时会导致presenter层太大，代码臃肿的问题。

#### MVVM

* View: 对应于Activity和xml，负责View的绘制以及与用户交互
* Model: 实体模型
* ViewModel: 负责完成View于Model间的交互,负责业务逻辑

MVVM的目标和思想MVP类似，利用数据绑定(Data Binding)、依赖属性(Dependency Property)、命令(Command)、路由事件(Routed Event)等新特性，打造了一个更加灵活高效的架构。

数据驱动

在MVVM中，以前开发模式中必须先处理业务数据，然后根据的数据变化，去获取UI的引用然后更新UI，通过也是通过UI来获取用户输入，而在MVVM中，数据和业务逻辑处于一个独立的View Model中，ViewModel只要关注数据和业务逻辑，不需要和UI或者控件打交道。由数据自动去驱动UI去自动更新UI，UI的改变又同时自动反馈到数据，数据成为主导因素，这样使得在业务逻辑处理只要关心数据，方便而且简单很多。

低耦合度

MVVM模式中，数据是独立于UI的，ViewModel只负责处理和提供数据，UI想怎么处理数据都由UI自己决定，ViewModel 不涉及任何和UI相关的事也不持有UI控件的引用，即使控件改变（TextView 换成 EditText）ViewModel 几乎不需要更改任何代码，专注自己的数据处理就可以了，如果是MVP遇到UI更改，就可能需要改变获取UI的方式，改变更新UI的接口，改变从UI上获取输入的代码，可能还需要更改访问UI对象的属性代码等等。

更新 UI

在MVVM中，我们可以在工作线程中直接修改View Model的数据（只要数据是线程安全的），剩下的数据绑定框架帮你搞定，很多事情都不需要你去关心。

团队协作

MVVM的分工是非常明显的，由于View和View Model之间是松散耦合的。一个是处理业务和数据，一个是专门的UI处理。完全有两个人分工来做，一个做UI（xml 和 Activity）一个写ViewModel，效率更高。

可复用性

一个View Model复用到多个View中，同样的一份数据，用不同的UI去做展示，对于版本迭代频繁的UI改动，只要更换View层就行，对于如果想在UI上的做AbTest 更是方便的多。

单元测试

View Model里面是数据和业务逻辑，View中关注的是UI，这样的做测试是很方便的，完全没有彼此的依赖，不管是UI的单元测试还是业务逻辑的单元测试，都是低耦合的。

通过上面对MVVM的简述和其他两种模式的对比，我们发现MVVM对比MVC和MVP来说还是存在比较大的优势，虽然目前Android开发中可能真正在使用MVVM的很少，但是是值得我们去做一些探讨和调研。

## 初识DataBinding

### 简单使用

在Module的gradle中配置

dataBinding {

enabled = true

}

如果是kotlin项目，还需配置

apply plugin: 'kotlin-kapt'

kapt "com.android.databinding:compiler:$dataBindingCompilerVersion"

创建一个简单的Bean

data class UserBean(val name: String, val age: Int)

使用了DataBinding之后的Activity的布局文件activity\_main.xml

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="user"

type="com.zx.databindingdemo.bean.UserBean" />

</data>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

style="@style/tv\_wrap"

android:text="@{user.name}" />

<!--注意：这里age是int类型，必须转化为String，否则会运行时异常-->

<TextView

style="@style/tv\_wrap"

android:text="@{String.valueOf(user.age)}" />

</LinearLayout>

</layout>

MainActivity使用DataBinding

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

val binding = DataBindingUtil.setContentView<ActivityMainBinding>(this, R.layout.activity\_main)

binding.user = UserBean("cgy", 25)

}

}

这里用DataBindingUtil.setContentView代替了setContentView，然后创建一个 UserBean 对象，通过 binding.setUser(userBean) 与 variable 进行绑定。

### 解惑

#### 为什么配置了dataBinding{enabled = true}之后就可以使用dataBinding

Android Studio中是依靠gradle来管理项目的，在创建一个项目时，从开始创建一直到创建完毕，整个过程是需要执行很多个gradle task的，这些task有很多是系统预先帮我们定义好的，比如build task，clean task等，DataBinding相关的task也是系统预先帮我们定义好的，但是默认情况下，DataBinding相关的task在task列表中是没有的，因为我们没有开启dataBinding,但是一旦我们通过 dataBinding{enabled = true}的方式开启DataBinding之后，DataBinding相关的task就会出现在task列表中，每当我们执行编译之类的操作时，就会执行这些dataBinding Task, 这些task的作用就是检查并生成相关dataBinding代码，比如dataBindingExportBuildInfoDebug这个task就是用来导出debug模式下的build信息的。

#### ActivityMainBinding这个类从哪来的

他是继承ViewDataBinding，是系统帮我们自动生成的，这个类的生成是有规则的，它是根据对应的布局文件的名字生成的，比如：activity\_main-->ActivityMainBinding 、fragment-->FragmentBinding即：第一个单词首字母大写，第二个单词首字母大写，最后都会拼上Binding就是生成的Binding类。

但是如果你在实际编写代码的过程中，你会发现并没有执行编译、运行之类等操作，ActivityMainBinding这个类就直接能用了，竟然还有这种操作？其实是Android Studio 这个IDE自动帮我们做了这一步，在默认情况下，系统会使用Android Studio为我们自动生成databinding相关的代码，但是这种方式生成的代码不能调试，如果你想通过点击ActivityMainBinding跳转到它的源码中，你会发现并不能如你所愿，而是会跳转到对应的布局文件中。那么如果我们确实要查看ActivityMainBinding的源码并且还想调试，我们就需要编译一次代码

#### DataBinding与ButterKnife的区别

ButterKnife很多人都用过，在以前的findViewById的时代，我们厌烦了写这些重复的代码，当有了ButterKnife之后，我们采用注解的方式来查找控件和注册监听，配合ButterKnife的插件，大大提升了我们的开发效率，一度成为开发神器，但是有了DataBinding之后，你会发现使用ButterKnife开发还是不够简洁：比如你需要给很多TextView setText()要获取editText的内容前必须要获取editText对象，给view设置监听前也必须要获取这个view对象等等。但是使用了DataBinding之后，这些冗余的代码统统都可以得到简化，从而真正的只需要专注于你的业务逻辑的处理即可。

## Binding类的其他生成方式

除了DataBindingUtil.setContentView(Context, layoutId)Binding类还有以下四种生成方式

MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater);

MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater, viewGroup, false);

MyLayoutBinding binding = MyLayoutBinding.bind(viewRoot);

MyLayoutBinding binding = DataBindingUtil.inflate(layoutInflater, layoutId, parent, viewGroup, false)

## DataBinding的基本用法

### 控件有id时，可以直接使用

android:id="@+id/recycler\_view\_bind\_btn"

binding.recyclerViewBindBtn.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

startActivity(new Intent(MainActivity.this, RecyclerViewActivity.class));

}

});

### 在xml中引入一些基础变量Variables

data 标签中可以有任意数量的 variable 标签。这些变量可以使Java中的任意数据类型，每个 variable 标签描述了会在 binding 表达式中使用的属性。

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="str"

type="String"/>

<variable

name="age"

type="int" />

</data>

<LinearLayout

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<TextView

style="@style/tv\_wrap"

android:text="@{str}"/>

<TextView

style="@style/tv\_wrap"

android:text="@{String.valueOf(age)}"/>

</LinearLayout>

</layout>

Data Binding和Java一样，java.lang包里的类，我们是可以不用导入包的，也就是说，java基础类型都是不用导包的。注意：android:text设置int类型的值一定要转化为String类型，否则系统会认为是资源文件id，就会出错

### 引入一些高级变量Variables

在上面，我们学会了如何在xml中定义变量，以及在控件中使用。同样，我们可以在data中定义像List、Map，数组等这样的集合变量，只是它的实现稍微有点不同，下面就一起来看看是如何定义以及使用的。

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<import type="java.util.List" />

<import type="java.util.Map" />

<!--泛型的支持会在编译时期报红线，但是是可以直接运行的但是需要通过转义字符才行，如：<号用&lt表示;>号用&gt表示;-->

<variable

name="list"

type="List&lt;String&gt;" />

<variable

name="map"

type="Map&lt;String,Object&gt;" />

<variable

name="array"

type="String[]" />

</data>

<LinearLayout

style="@style/vg\_wrap\_mar\_t15"

android:orientation="vertical">

<TextView

style="@style/tv\_wrap"

android:text="@{list[0]}" />

<TextView

style="@style/tv\_wrap"

android:text="@{list.get(1)}" />

<TextView

style="@style/tv\_wrap\_mar\_t10"

android:text="@{map[`key0`]}" />

<TextView

style="@style/tv\_wrap"

android:text='@{map.get("key0")}' />

<TextView

style="@style/tv\_wrap\_mar\_t10"

android:text="@{array[0]}" />

<TextView

style="@style/tv\_wrap\_mar"

android:text="@{array[1]}" />

</LinearLayout>

</layout>

大家可以看到，list和map这里我没有用List<String>和Map<String,Object>，而是用的List&lt;String&gt;和Map&lt;String,Object&gt;原因是在data中，有些字符是必须用转义字符才能编译通过，上面把<>换成转义字符的写法虽然会在编译时是红色的，但是不用担心，会编译通过的，下面给出常用的转义字符。

附：常用的转义字符



* 关于获取list和map的值，我们有2种写法，[]或者是get()
* 如果是list或者数组，需要设置索引下标，如果是map，还需要为它定义key的变量
* 官方推荐这种集合框架使用[]的写法。
* map的key还有一种单引号的写法。

单引号android:text='@{map["key0"]}' android:text='@{map.get("key0")}'

双引号android:text="@{map[`key0`]}" android:text="@{map.get(`key0`)}"

在Activity中初始化数据，设置这些变量

class TwoActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

val binding = DataBindingUtil.setContentView<ActivityTwoBinding>(this, R.layout.activity\_two)

binding.list = arrayListOf("list1", "list2")

binding.map = mapOf("key0" to "map1", "key1" to "map2")

binding.array = arrayOf("array1", "array2")

}

}

### xml中引用表达式

举几个例子，还有很多，大多数Java表达式都是支持的

android:text="@{String.valueOf(age)}"

android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"

android:text='@{"iname:" +user.name}'

此外还支持null合并操作,它的符号是??，意思是:如果左边的对象非空则选择左边，否则选择右边，这和Java中的三目运算符是一样的，可以算是一个简略吧。

android:text="@{user.displayName ?? user.lastName}"

//等价于

android:text="@{user.displayName != null ? user.displayName : user.lastName}"

支持的表达式语言

表达式语言与 Java 表达式有很多相似之处。下面是相同之处：

* 数学计算 + - / \* %
* 字符串连接 +
* 逻辑 && ||
* 二进制 & | ^
* 一元 + - ! ~
* 位移 >> >>> <<
* 比较 == > < >= <=
* instanceof
* 组 ()
* 字面量 - 字符，字符串，数字， null
* 类型转换
* 函数调用
* 字段存取
* 数组存取 []
* 三元运算符 ?：

不支持的操作符

一些 Java 中的操作符在表达式语法中不能使用。

* this
* super
* new
* 显式泛型调用 <T>

### 设置别名alias

如果我们import了两个不同路径，但名称相同的类，可以借助于别名来解决，别名借助alias字段来标识，例如：

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<import type="com.cgy.databindingdemo.bean.UserBean" />

<!--不同路径下有2个相同名字的类，那么给其中一个起一个别名，用alias表示-->

<import

alias="UserBean2"

type="com.cgy.databindingdemo.bean2.UserBean" />

<!--不同路径下有2个相同名字的类，那么给其中一个起一个别名，用alias表示-->

<variable

name="user"

type="com.cgy.databindingdemo.bean.UserBean" />

<variable

name="user2"

type="UserBean2" />

</data>

<LinearLayout

style="@style/vg\_wmatch\_hwrap\_mar\_t15"

android:orientation="vertical">

<TextView

style="@style/tv\_wrap\_mar\_t10"

android:text='@{"姓名:"+user.name}' />

<TextView

style="@style/tv\_wrap\_mar\_t10"

android:text="@{`user2:`+user2.name}" />

</LinearLayout>

</layout>

### include中的使用

在使用命名空间的布局中，变量可以传递到任何 include 布局中。

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto">

<data>

<variable

name="user"

type="com.cgy.databindingdemo.bean.UserBean" />

<variable

name="user2"

type="com.cgy.databindingdemo.bean2.UserBean" />

</data>

<LinearLayout

style="@style/vg\_wmatch\_hwrap"

android:gravity="center\_horizontal"

android:orientation="vertical">

<include

layout="@layout/in\_name"

app:user="@{user}" />

<include

layout="@layout/in\_content"

app:user2="@{user2}" />

</LinearLayout>

</layout>

注意：在in\_name.xml以及in\_contact.xml两个layout文件中必需要有user和user2 variable

<!--in\_name-->

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="user"

type="com.cgy.databindingdemo.bean.UserBean" />

</data>

<LinearLayout

style="@style/vg\_wmatch\_hwrap"

android:gravity="center\_horizontal"

android:orientation="vertical">

<TextView

style="@style/tv\_wrap\_mar\_t10"

android:text='@{"姓名:"+user.name}' />

</LinearLayout>

</layout>

Data binding不支持直接包含merge 节点。举个例子， 以下的代码不能正常运行 ：

<merge>

<include

layout="@layout/name"

app:user="@{user}"/>

<include

layout="@layout/contact"

app:user="@{user}"/>

</merge>

### 事件处理

大家都知道，我们经常需要处理View的点击事件，在xml中android:onClick 可以指定 Activity 中的函数，Data Binding 也允许处理从视图中发送的事件。下面给出几种实现方式：

* 布局中引入OnClickListener的变量
* 方法调用
* Lambda表达式

#### 布局中引入OnClickListener的变量

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="click"

type="android.view.View.OnClickListener" />

</data>

<LinearLayout

style="@style/vg\_wmatch\_hwrap"

android:gravity="center\_horizontal"

android:orientation="vertical">

<Button

android:id="@+id/click\_btn"

style="@style/tv\_wrap"

android:onClick="@{click}"

android:text="button1" />

<Button

android:id="@+id/click\_btn"

style="@style/tv\_wrap"

android:onClick="@{click}"

android:text="button2" />

</LinearLayout>

</layout>

Activity处理点击事件

class FiveActivity : AppCompatActivity(), View.OnClickListener {

override fun onClick(v: View?) {

when (v!!.id) {

R.id.click\_btn -> toast("按钮1")

R.id.click\_btn2 -> toast("按钮2")

}

}

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

val binding = DataBindingUtil.setContentView<ActivityFiveBinding>(this, R.layout.activity\_five)

binding.click = this

}

}

#### 方法调用

相比较于在android:onClick中指定Activity的一个方法，它的优势在于表达式会在编译时处理，如果方法不存在或者方法签名不对，编译将会报错。以下是个例子：

class OnClickHandler {

fun onClickFriend(v: View) {

Toast.makeText(v.getContext(), "onClickFriend1", Toast.LENGTH\_SHORT).show()

}

fun onClickFriend2(v: View) {

Toast.makeText(v.getContext(), "onClickFriend2", Toast.LENGTH\_SHORT).show()

}

}

布局文件如下

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="handler"

type="com.cgy.databindingdemo.ui.OnClickHandler" />

</data>

<LinearLayout

style="@style/vg\_wmatch\_hwrap"

android:gravity="center\_horizontal"

android:orientation="vertical">

<Button

style="@style/tv\_wrap"

android:onClick="@{handler::onClickFriend}"

android:text="button3" />

<Button

style="@style/tv\_wrap"

android:onClick="@{handler::onClickFriend2}"

android:text="button4" />

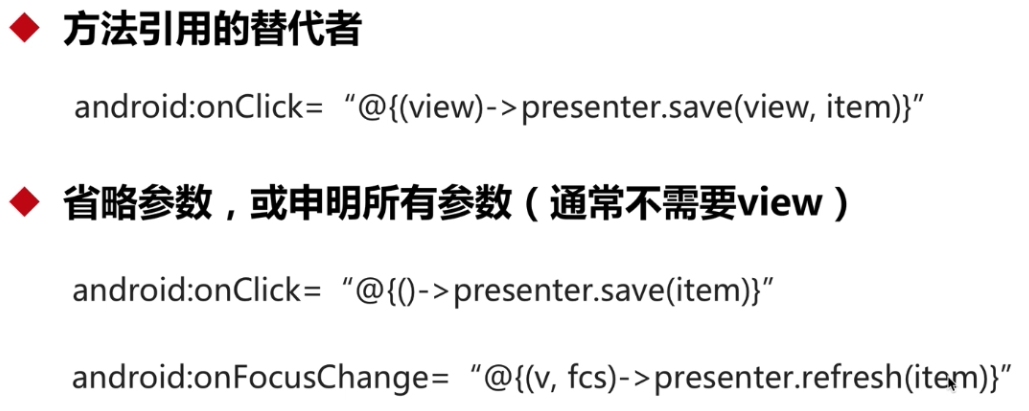
</LinearLayout>

</layout>

别忘了在Activity设置变量

binding.handler = OnClickHandler()

#### Lambda表达式



public class MyClick {

public void doClick(UserBean bean) {

Toast.makeText(MainActivity.this, bean.getName(), Toast.LENGTH\_SHORT).show();

}

}

binding.setUser(new UserBean("", "444", 35));

binding.setMyClick(new MyClick());

<variable

name="user"

type="com.cgy.bean.UserBean" />

<variable

name="myClick"

type="com.cgy.databindingdemojava.MainActivity.MyClick" />

android:onClick="@{() -> myClick.doClick(user)}"

可也以直接传入context，该context是view 的context：

public void doClick(String s, Context context) {

ComRecyclerViewActivity.start(context, s);

}

android:onClick="@{() -> myClick.doClick(user.name, context)}"

## 值变化通知UI变化

任何普通的 java 对象（POJO）可用于数据绑定，但修改一个POJO不会造成UI更新。数据绑定的真正能力可以通过给数据对象在数据变化时通知来使用,有三种不同的数据更改通知机制，Observable 对象，ObservableFilelds 字段和 Observable Cllections 集合,当这些可观察数据对象绑定到 UI 和数据对象的属性更改时，用户界面将自动更新

### Observable Objects

实现 Observable 接口类允许单个监听器连接到绑定对象以监听该对象所有属性的改变Observable 接口有一个机制来添加和删除监听器，但通知是由开发人员管理,为使开发更容易，一个 BaseObservable 基类是为了实现监听器注册机制而创建。Data 类实现者仍然是负责通知时的性能变化。这是通过分配一个 Bindable 注释 getter 和 setter 进行通知来完成

package com.cgy.databindingdemojava;

public class Empl extends BaseObservable{

private String name;

public Empl(String name) {

this.name = name;

}

@Bindable

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

notifyPropertyChanged(com.cgy.databindingdemojava.BR.name);

}

}

### Observable Fields

#### 简单的Observable Fields

一个小工作会涉及到创建 Observable 类，所以开发者们想节省时间或仅有几个特性可以用 observablefield 和它的同类observableboolean，observablebyte，observablechar，observableshort，observableint，observablelong，observablefloat，observabledouble，和observableparcelable。observablefields 完备可观察对象有一个单一的领域。原始版本避免装箱和拆箱过程中访问操作。若要使用，请在数据类中创建 public final 字段：

public final ObservableField<String> addr = new ObservableField("");

android:text="@{emperr.addr}" 或者android:text="@{emperr.addr.get()}"

emperr.addr.set("addsNew");

#### 对象的Observable Fields

public class ObBean extends BaseObservable{

private String str1;

public ObBean(String str1) {

this.str1 = str1;

}

@Bindable

public String getStr1() {

return str1;

}

public void setStr1(String str1) {

this.str1 = str1;

notifyPropertyChanged(com.cgy.databindingdemojava.BR.str1);

}

}

public final ObservableField<ObBean> bean = new ObservableField();

android:text="@{emperr.bean.get().str1}"

private ObBean bean = new ObBean("111");

emperr.bean.set(bean);

bean.setStr1("ssssss");

#### 注意，尽量给ObservableField创建public final

* xml中使用emperr.addr会优先去找getAddr方法，然后找public 的addr，如果没有getAddr和public 的addr，不能实现动态变化
* 定义public String getAddr() {return addr.get();}不能实现动态变化
* 定义public ObservableField<String> getAddr() {return addr;}能实现动态变化
* 所以给ObservableField创建public final后，不用去定义getAddr方法
* 为了看起来更像是直接给addr赋值，可以给addrs设置setter方法：

public void setAddr(String value) {  
 addr.set(value);  
}

emperr.setAddr("addsNew");

### Obseravble Collection

#### ObservableArrayMap

public final ObservableArrayMap<String, String> user = new ObservableArrayMap();

public void putUser(String key, String value) {user.put(key, value);}

android:text="@{emperr.user[`key1`]}"

emperr.putUser("key1", "vvvvv");

#### ObservableArrayList

public final ObservableArrayList<ObBean> beans = new ObservableArrayList<>();

emperr.beans.add(new ObBean("1"));

android:text="@{emperr.beans[0].str1}"

emperr.beans.add(new ObBean("1"));

emperr.beans.get(0).setStr1("2");

## 双向绑定

### xml中的用法

在实现之前的Observable的情况下，xml中将@{}改为@={}即可

android:text="@={item.text}"

### @InverseBindingAdapter和死循环

注意必须有attribute ，也要注意不要陷入互相通知改变的死循环，由于双向绑定的监听UI和Data的数据是否相等，如果相等，就不再通知，所以下面的加上“cgy”会陷入死循环

@InverseBindingAdapter(attribute = "android:text")

public static String doTtt(TextView tv) {

tv.setTextColor(Color.parseColor("#675698"));

return tv.getText().toString() + "cgy";//加了cgy会陷入死循环

}

### 不是所有属性都能双向绑定

双向绑定只有那些带有额外事件的，系统能监听的，比如：text、checked、year、month、progress

## 监听属性变更

public class BR {

...

public static final int text = 9;

public static final int text2 = 10;

}

TextItem item = new TextItem("标题1");  
item.addOnPropertyChangedCallback(new Observable.OnPropertyChangedCallback() {  
 @Override  
 public void onPropertyChanged(Observable sender, int propertyId) {  
 TextItem textItem = (TextItem) sender;  
 if (propertyId == 9) {  
 Log.e("tag", "onPropertyChanged:" + textItem.getText());  
 } else {  
 Log.e("tag", "onPropertyChanged:" + textItem.getText2());  
 }  
 }  
});

## 表达式链

使用有id的控件的属性来设置本控件的属性，注意id使用驼峰（title\_tv -> titleTv）,可以去掉重复的表达式，也可以隐式的更新

<EditText

android:id="@+id/title\_tv"

android:visibility='@{item.text.equals("cgy") ? view.GONE : view.VISIBLE}' />

<EditText

android:visibility="@{titleTv.visibility}"/>

<CheckBox

android:id="@+id/title\_cb" />

<EditText

android:visibility="@{titleCb.checked ? view.GONE : view.VISIBLE}" />

## 动画

简单的实现动画，只需给binding 添加一个监听即可

binding.addOnRebindCallback(new OnRebindCallback() {

@Override

public boolean onPreBind(ViewDataBinding binding) {

ViewGroup root = (ViewGroup) binding.getRoot();

TransitionManager.beginDelayedTransition(root);

return true;

}

});

## RecyclerView中使用DataBinding

利用Data Binding为RecyclerView填充数据。这里的难点和核心点就是RecyclerView中Adapter的实现。主要是下面这三个地方的改动。

### ViewHolder

之前是

class FruitViewHolder extends RecyclerView.ViewHolder {

public FruitViewHolder(View itemView) {

super(itemView);

}

}

现在是：

class FruitViewHolder extends RecyclerView.ViewHolder {

private ItemFruitBinding binding;

public ItemFruitBinding getBinding() {

return binding;

}

public FruitViewHolder(ItemFruitBinding binding) {

super(binding.getRoot());

this.binding = binding;

}

}

### onCreateViewHolder

之前是

@Override

public RecyclerView.ViewHolder onCreateMyViewHolder(ViewGroup parent, int viewType) {

View view = mInflater.inflate(R.layout.item\_fruit, parent, false);

return new MyViewHolder(view);

}

现在是

@Override

public RecyclerView.ViewHolder onCreateMyViewHolder(ViewGroup parent, int viewType) {

ItemFruitBinding binding = DataBindingUtil.inflate(mInflater, R.layout.item\_fruit, parent, false);

return new FruitViewHolder(binding);

}

### onBindViewHolder

之前是

@Override

public void onBindViewHolder(MyViewHolder holder, int position) {

holder.tv\_name.setText(mList.get(position).getName());

}

现在是

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="item"

type="com.cgy.bean.FruitItem" />

</data>

<LinearLayout

...

<TextView

...

android:text="@{item.name}" />

</LinearLayout>

</layout>

@Override

public void onBindMyViewHolder(RecyclerView.ViewHolder holder, int position) {

FruitViewHolder fruitViewHolder = (FruitViewHolder) holder;

ItemFruitBinding itemFruitBinding = fruitViewHolder.getBinding();

FruitItem fruitItem = (FruitItem) mList.get(position);

itemFruitBinding.setItem(fruitItem);

itemFruitBinding.executePendingBindings();

}

### 公用RecycleView的Adapter和ViewHolder

<variable

name="item"

type="com.cgy.bean.TextItem" />

android:hint="@{item.text}"

public class ComBindViewHolder extends RecyclerView.ViewHolder {

private ViewDataBinding binding;

public ViewDataBinding getBinding() {

return binding;

}

public ComBindViewHolder(ViewDataBinding binding) {

super(binding.getRoot());

this.binding = binding;

}

}

public class ComBindRecyAdapter<T> extends RecyclerView.Adapter<ComBindViewHolder> {

private Context mContext;

private List<T> mDatas;

private int layoutId;

private int brId;

private final LayoutInflater mInflater;

public ComBindRecyAdapter(Context context, List<T> mDatas, int layoutId, int brId) {

mContext = context;

this.mDatas = mDatas;

this.layoutId = layoutId;

this.brId = brId;

mInflater = LayoutInflater.from(mContext);

}

@Override

public ComBindViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

ViewDataBinding binding = DataBindingUtil.inflate(mInflater, layoutId, parent, false);

ComBindViewHolder viewHolder = new ComBindViewHolder(binding);

return viewHolder;

}

@Override

public void onBindViewHolder(@NonNull ComBindViewHolder holder, int position) {

ViewDataBinding binding = holder.getBinding();

binding.setVariable(brId, mDatas.get(position));

binding.executePendingBindings();

}

@Override

public int getItemCount() {

return mDatas == null ? 0 : mDatas.size();

}

}

adapter = new ComBindRecyAdapter(this, mList, R.layout.item\_text, BR.item);

LinearLayoutManager layoutManager = new LinearLayoutManager(this, OrientationHelper.VERTICAL, false);

recyclerView.setLayoutManager(layoutManager);

recyclerView.setAdapter(adapter);

## @BindingAdapter设置自定义属性

### 自定义属性

@BindingAdapter对于自定义属性是非常有用的，举个例子，如果你的项目中用到了Glide，或者是其他的图片加载框架，由于这些框架一般通过url给ImageView设置图片的，但是ImageView中并没有设置url的属性，这个时候，BindingAdapter就派上用场了

public class ImageHelper {

/\*\*

\* 1.加载图片,无需手动调用此方法

\* 2.使用@BindingAdapter注解设置自定义属性的名称，imageUrl就是属性的名称，

\* 当ImageView中使用imageUrl属性时，会自动调用loadImage方法，

\* @param imageView ImageView

\* @param url 图片地址

\*/

@BindingAdapter({"imageUrl"})

public static void loadImage(ImageView imageView, String url) {

Glide.with(imageView.getContext()).load(url)

.placeholder(R.mipmap.fruit)

.error(R.mipmap.fruit)

.into(imageView);

}

}

使用@BindingAdapter注解设置自定义属性的名称，如上所示，imageUrl就是属性的名称，当ImageView中使用imageUrl属性时，会自动调用loadImage方法，参数imageView为当前使用imageUrl属性的ImageView控件，参数url为imageUrl属性的值。

app:imageUrl="@{item.picUrl}"

注意：imageUrl属性可以加前缀，使用的时候依然使用imageUrl即可，通常是bind:imageUrl,只要不是android:imageUrl即可

### 多个自定义属性

public class ImageHelper {

/\*\*

\* 多属性的情况，参数url对应imageUrl，errorRes对应imageError

\*/

@BindingAdapter({"bind:imageUrl", "bind:imageError"})

public static void loadImageError(ImageView imageView, String url, String errorRes) {

Glide.with(imageView.getContext()).load(url)

.placeholder(R.mipmap.fruit)

.error(errorRes)

.into(imageView);

}

}

app:imageUrl="@{item.picUrl}"

app:error="@{item.picId}"

注意：自定义属性类型为Drawable时，引用只能app:error="@{@drawable/ic\_launcher}"，且不能用mipmap包下的文件

### 使用系统的属性，但是逻辑自定义

一些系统的属性需要自己去定义并实现逻辑，比如android：hint。我们可以同BindingAdapter注解去自定义个自己的逻辑：

@BindingAdapter({"android:hint"})

public static void myHint(EditText editText, String s) {

editText.setText(s);

editText.setTextColor(Color.parseColor("#ff0000"));

}

android:hint="@{item.text}"

### 多属性

* 注意一定要加可空，因为原始数据可能为null
* requireAll代表是否必须要所有参数，默认是true

app:imageUrl="@{vm.headUrl}"

app:imageError="@{@drawable/bikuaidai\_logo2}"

@BindingAdapter(value = ["imageUrl", "imageError"], requireAll = false)

fun loadImage(imageView: ImageView, url: String?, errorDw: Drawable?) {

if (errorDw != null) {

Glide.with(imageView.context).load(url)

.error(errorDw)

.into(imageView)

} else {

Glide.with(imageView.context).load(url)

.error(R.drawable.bikuaidai\_logo)

.into(imageView)

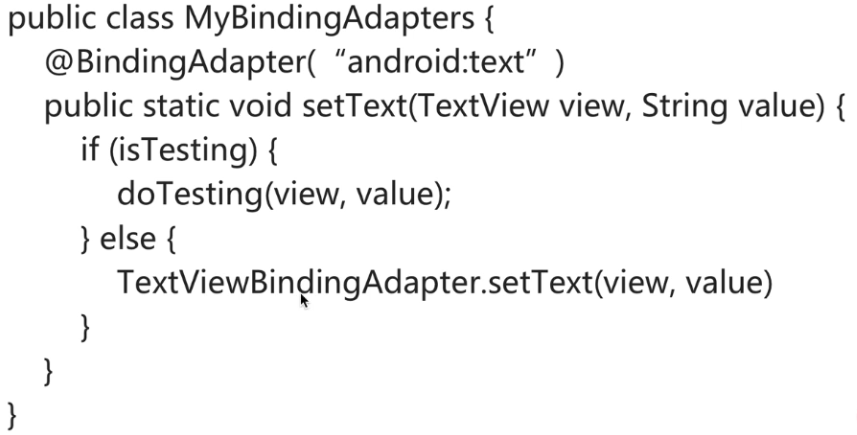
}

}

## DataBindingComponent实现数据

### 实现测试数据、正式数据分离

场景：假设开发的的时候需要显示demo数据，正式环境显示正式数据，传统的可以用ifelse来解决：



通过BindingAdapter是可以增加一些自定义的属性或者是修改Android原生的属性，但是它有一个弊端，就是没有改全局修改所有的相关属性，不过配合上DataBindingComponent就可以解决这个问题。

以一个参数的形式传入这个View中，那么只作用于当前的view。

DataBindingUtil.setContentView(this, R.layout.activity\_component, new FirstComponent());

如果要设置全局的方式，可以通过setDefaultComponent

DataBindingUtil.setDefaultComponent(new FirstComponent());

需要先创建一个拥有@BindingAdapter的类，定义为abstract，因为这样DataBindingComponent就可以找到它：

public abstract class MyAbstractBindAdapter {

@BindingAdapter("android:text")

public abstract void setText(TextView view, String value);

@BindingAdapter("android:textColor")

public abstract void setTextColor(TextView view, int value);

}

继承MyAbstractBindAdapter分别实现两个adapter:

ProductionBindingAdapter.java

public class ProductionBindingAdapter extends MyAbstractBindAdapter {

@Override

public void setText(TextView view, String value) {

TextViewBindingAdapter.setText(view, value);

}

@Override

public void setTextColor(TextView view, int value) {

view.setTextColor(value);

}

}

TestBindingAdapter.java

public class TestBindingAdapter extends MyAbstractBindAdapter {

@Override

public void setText(TextView view, String value) {

view.setText(value + " test");

}

@Override

public void setTextColor(TextView view, int value) {

view.setTextColor(view.getContext().getResources().getColor(R.color.textColorNight));

}

}

}

在写了上面的代码后，再看看DataBindingComponent，会发现里面多了一个接口方法，遂实现之:

ProductionComponent.java

public class ProductionComponent implements DataBindingComponent {

private MyAbstractBindAdapter mAdapter = new ProductionBindingAdapter();

@Override

public MyAbstractBindAdapter getMyAbstractBindAdapter() {

return mAdapter;

}

}

TestComponent.java

public class TestComponent implements DataBindingComponent {

private MyAbstractBindAdapter mAdapter = new TestBindingAdapter();

@Override

public MyAbstractBindAdapter getMyAbstractBindAdapter() {

return mAdapter;

}

}

最后注入Component很简单，我们做全局的注入只需要调用:

if (DemoApplication.isTest) {

DataBindingUtil.setDefaultComponent(new ProductionComponent());

} else {

DataBindingUtil.setDefaultComponent(new TestComponent());

}

### 作为参数传给静态的BindingAdapter方法（未实现）

@BindingAdapter({"bind:imageUrl"})

public static void loadImage(PdComponent component, ImageView imageView, String url) {

Glide.with(imageView.getContext()).load(url)

.placeholder(R.mipmap.fruit)

.error(R.mipmap.ic\_pd)

.into(imageView);

}

@BindingAdapter({"bind:imageUrl"})

public static void loadImage(TestComponent component, ImageView imageView, String url) {

Glide.with(imageView.getContext()).load(url)

.placeholder(R.mipmap.fruit)

.error(R.mipmap.ic\_test)

.into(imageView);

}

在DataBindingUtil.setDefaultComponent后，总有一个出错：TestComponent can not cast to PdComponent 或者PdComponent can not cast to TestComponent，

### 提取共同逻辑

解决办法：如果逻辑是相同的，单独起一个类，写公用的逻辑；如果逻辑不同那么就把bind:imageUrl提取到抽象类MyAbstractBindAdapter 中：

@BindingAdapter({"bind:imageUrl"})

public static void loadImage(ImageView imageView, String url) {

Glide.with(imageView.getContext()).load(url)

.placeholder(R.mipmap.fruit)

.error(R.mipmap.ic\_common)

.into(imageView);

}

@BindingAdapter({"bind:imgUrl"})

public abstract void loadImage(ImageView imageView, String url) ;

## @BindingConversion设置转换值

@BindingConversion

public static Drawable coverInt2Drawable(int i){

return new ColorDrawable(i);

}

app:error="@{item.picId == 3 ? @color/red : @color/green}"

## Databinding插件

### Databinding Support

官方地址：<https://plugins.jetbrains.com/plugin/9271-databinding-support>，功能：

* Convert layout to databinding layout——在原布局根目录的Layout标签上Alt + Enter
* Add data、import、variable tag——在任意地方Alt + Enter
* Wrap @{}、 @={}——在赋值位置Alt + Enter
* 跳转到使用了该ViewModel的Layout里——点击class左边的小图标

### Data Binding Formatter

使Java bean实现Observable，并给所有的属性添加 @Bindable和notify

### CreateAdapterPlugin

自动生成DataBinding框架的RecycleView的Adapter和ViewHolder、item.xml

## DataBinding实用指南

### 统一命名的variable

bad:

<!--user\_activity.xml -->

<variable

name="uservm"

type="io.ditclear.app.viewmodel.UserViewModel"/>

<!--student\_activity.xml -->

<variable

name="studentvm"

type="io.ditclear.app.viewmodel.StudentViewModel"/>

better:

<!--user\_activity.xml -->

<variable

name="vm"

type="io.ditclear.app.viewmodel.UserViewModel"/>

<!--student\_activity.xml -->

<variable

name="vm"

type="io.ditclear.app.viewmodel.StudentViewModel"/>

### 尽可能少的variable和import

bad:

<data>

<import type="android.view.View"/>

<import type="com.ditclear.app.util.StringUtil"/>

<import type="java.math.BigDecimal"/>

<variable

name="isShow"

type="Boolean"/>

<variable

name="time"

type="java.util.Date"/>

<variable

name="date"

type="java.util.Date"/>

<variable

name="signTime"

type="String"/>

</data>

better:

<data>

<variable

name="presenter"

type="com.ditclear.app.helper.presenter.Presenter"/>

<variable

name="vm"

type="com.ditclear.app.view.student.viewmodel.StudentViewModel"/>

</data>

### 避免使用复杂的表达式

bad:

android:text="@{MyStringUtils.capitalize(user.lastName)}"

android:text="@{String.valueOf(index + 1)}"

android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"

android:transitionName='@{"image\_" + id}'

android:text="@{@string/nameFormat(firstName, lastName)}"

android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"

android:text="@{map[`firstName`]}"

它有一个很大的弊端，相信很多刚入门DataBinding的人都遇到过找不到binding文件的错，然后被劝退，原因无外乎就是表达式写错、variable的类路径不对或者没import相应的类（比如View）等等，都与复杂的表达式有关系，而DataBinding报错也不太智能，有时不能准确定位，所以建议不要在xml里进行复杂的数据绑定，这些都尽量放到ViewModel里进行，xml只绑定简单的基础数据类型。

better：

<data>

<variable name="vm" type="com.example.UserViewModel"/>

</data>

<TextView

android:text="@{vm.capitalizeLastName}"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:hint="@{vm.index}"

android:visibility="@{vm.showName}"

android:transitionName='@{vm.transitionName}'/>

### 点击事件的命名和处理

DataBinding支持的写法：

android:onClick="@{presenter.onClick()}" //1.方法引用

android:onClick="@{()->presenter.onClick()}" //2.lamda表达式

android:onClick="@{(view)->presenter.onClick(view)}" //3.lamda表达式

android:onClick="@{()->presenter.onClick(item)}"//4.带参数lamda表达式

android:onClick="@{(view)->presenter.onClick(view, item)}"//5.带参数lamda表达式

选择很多，而且使用方法也是五花八门，有的喜欢直接使用viewmodel里的方法，有的直接将Activity或者fragment作为handler，还有的可能会在activity/fragment里写一个内部类作为presenter，然后由于方法名也可以自定义，所以很可能出现你叫presenter.save()另一个叫(v)->handler.onSave(v)的情况。

bad：

<layout>

<data>

<variable

name="vm"

type="io.ditclear.app.viewmodel.AnimalViewModel"/>

<variable

name="handler"

type="io.ditclear.app.view.AnimalActivity"/>

<variable

name="presenter"

type="io.ditclear.app.view.AnimalActivity.Presenter"

</data>

<LinearLayout

tools:context="io.ditclear.app.view.AnimalActivity">

<Button

android:onClick="@{vm.shoutWhat()}"/>

<Button

android:onClick="@{(v)->handler.shout(v)}"/>

<Button

android:onClick="@{()->presenter.onShout()}"/>

</LinearLayout>

</layout>

better：

推荐的一种处理方式是使用封装过的View.OnClickListener来统一处理点击事件，包裹一层的目的是为了不依赖于具体实现。

public interface Presenter extends View.OnClickListener{

@Override

void onClick(View v);

}

xml布局文件

<layout>

<data>

<variable

name="vm"

type="io.ditclear.app.viewmodel.AnimalViewModel"/>

<variable

name="presenter"

type="io.ditclear.app.helper.Presenter"

</data>

<LinearLayout

tools:context="io.ditclear.app.view.AnimalActivity">

<Button

android:id="@+id/save\_btn"

android:onClick="@{(v)->presenter.onClick(v)}"/>

<Button

android:id="@+id/delete\_btn"

android:onClick="@{(v)->presenter.onClick(v)}"/>

<Button

android:id="@+id/submit\_btn"

android:onClick="@{(v)->presenter.onClick(v)}"/>

</LinearLayout>

</layout>

这里推荐使用(v)->presenter.onClick(v)的写法，原因之一是比较直观一点，其二是需要参数view接着在activity/fragment中来实现Presenter接口，处理点击事件

public class AnimalActivity extends AppCompatActivity implements Presenter {

private AnimalActivityBinding mBinding;

private AnimalViewModel mViewModel;

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mBinding= DataBindingUtil.setContentView(this, R.layout.animal\_activity);

Animal animal=new Animal("dog",1);

mViewModel=new AnimalViewModel(animal);

mBinding.setVm(viewModel);

mBinding.setPresenter(this);

}

@SingleClick

@Override

public void onClick(View v) {

//根据id进行区分

switch (v.getId()){

case R.id.save\_btn:

save();

break;

case R.id.delete\_btn:

delete();

break;

case R.id.submit\_btn:

submit();

break;

}

}

private void submit(){

//调用viewModel的方法

mViewModel.submit();

}

}

### 处理RecyclerView 列表项的数据及点击事件

RecyclerView功能极其强大，能做到的事情很多，网上已经出现很多关于多类型RecyclerView的处理方法，在使用DataBinding这一年多时间里，感受便是使用DataBinding来处理RecyclerView Item再合适不过，充分做到了数据和itemView的完美分离，告别了反复、冗余的自定义Adapter，不需要关心太多无意义的事情。

详情请查看github地址（kotlin版本）：https://github.com/ditclear/BindingListAdapter

### 补全自定义的属性

比如为ImageView，你定义了一个BindingAdapter

@BindingAdapter("url")

public static void bindImgUrl(ImageView imageView,String url){

Glide.with(imageView.getContext()).load(url).into(imageView);

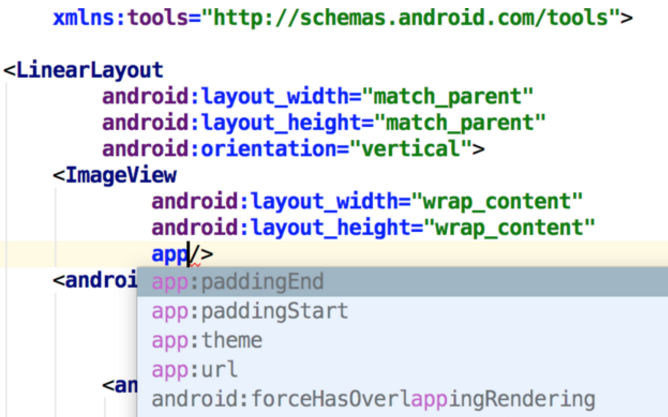
}

实际情况中，ImageView并没有url这个属性， 这时可以在attrs.xml文件中为ImageView添加这一属性，rebuild一下项目，以后就能自动补全属性了

<declare-styleable name="ImageView">

<attr name="url" format="string"/>

</declare-styleable>



### 错误排查

很多开发者放弃DataBinding原因就在于出错了不容易排查错误。只显示出很多XXBinding未找到。如果有一定使用经验的就知道只看最后一条报错信息就够了。这里介绍一种我经常使用来排查错误的方式:

在Android Studio 的terminal 里运行gradlew clean assembleDebug或者gradlew compileDebugJavaWithJavac

因为DataBinding是编译生成代码的，很多错误都是xml中表达式写的有问题导致的，所以运行以上命令容易在terminal中打印出具体错误的信息。这些命令对于需要编译生成代码的框架排查错误十分有用，比如Dagger2。

Cannot find the setter for attribute 'android:layout\_width' with parameter type android.databinding.ObservableField<java.lang.String> on android.widget.LinearLayout.

file://H:\AndroidStudioProjects\DataBindingDemo\app\src\main\res\layout\activity\_second.xml Line:13

## 坑集合

### 变量如果引用了没有的成员

比如user2没有content成员，使用android:text="@{`user2:`+user2.content}" />会报下面的错：e: [kapt] An exception occurred: android.databinding.tool.util.LoggedErrorException: Found data binding errors.

### include和BindingAdapter中，img和非引用设值的处理

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools">  
 <data>  
 <variable name="imgSrcDraw" type="android.graphics.drawable.Drawable" />

<variable name="title" type="String" />  
 <variable name="isShow" type="Boolean" />  
 <variable name="index" type="Integer" />  
 </data>

<LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
 <ImageView  
 style="@style/img\_wrap"

app:title='@{@string/yiyongedu}'

app:imgSrcDraw="@{imgSrcDraw}" />

<TextView  
 style="@style/wrap\_88\_14"  
 android:text="@{title}" />  
 <TextView  
 style="@style/wrap\_88\_14"  
 android:layout\_marginTop="3dp"  
 android:text="@{value}"  
 android:visibility="@{isShow? view.VISIBLE : view.GONE}" />  
 <TextView  
 style="@style/wrap\_88\_14"  
 android:layout\_marginTop="3dp"  
 android:text="@{value}"  
 android:visibility="@{index == 3? view.VISIBLE : view.GONE}" />  
 </LinearLayout>  
</layout>

父layout中

<include  
 layout="@layout/layout\_financial\_product\_tag"

app:title='@{"冻结额度"}'

app:imgSrcDraw="@{@drawable/icon\_dong}"

app:isShow="@{true}"  
app:index="@{2}"/>

BindingAdapter中

*// 根据intRes给ImageView设置src*@BindingAdapter("imgSrcDraw")  
fun bindImgSrcIntRes(img: ImageView, drawable: Drawable) {  
 img.setImageDrawable(drawable)  
}

### As升级3.2以后，报错 couldn't make a guess XXX

包名路径不能为大写，类名可以是大写，但是文件夹名最好不要大写，但是如果依赖的库包名为大写，我们又改不到，此时只能在gradle.properties中添加

android.databinding.enableV2 = false

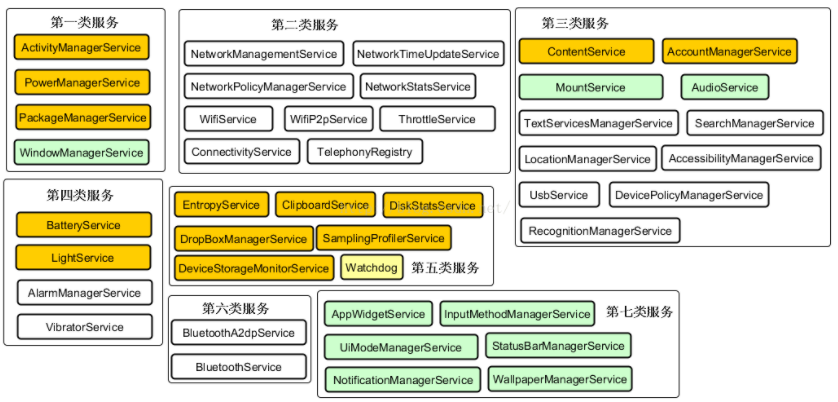
android.enableExperimentalFeatureDatabinding = true

# Aspect面向切面编程

## 闲谈AOP

大家都知道OOP，即ObjectOriented Programming，面向对象编程。而本文要介绍的是AOP。AOP是Aspect Oriented Programming的缩写，中译文为面向切向编程。OOP和AOP是什么关系呢？

* OOP和AOP都是方法论。我记得在刚学习C++的时候，最难学的并不是C++的语法，而是C++所代表的那种看问题的方法，即OOP。同样，今天在AOP中，我发现其难度并不在利用AOP干活，而是从AOP的角度来看待问题，设计解决方法。这就是为什么我特意强调AOP是一种方法论的原因！
* 在OOP的世界中，问题或者功能都被划分到一个一个的模块里边。每个模块专心干自己的事情，模块之间通过设计好的接口交互。从图示来看，OOP世界中，最常见的表示比如：



上图所示为AndroidFramework中的模块。OOP世界中，大家画的模块图基本上是这样的，每个功能都放在一个模块里。非常好理解，而且确实简化了我们所处理问题的难度。

OOP的精髓是把功能或问题模块化，每个模块处理自己的家务事。但在现实世界中，并不是所有问题都能完美得划分到模块中。举个最简单而又常见的例子：现在想为每个模块加上日志功能，要求模块运行时候能输出日志。在不知道AOP的情况下，一般的处理都是：先设计一个日志输出模块，这个模块提供日志输出API，比如Android中的Log类。然后，其他模块需要输出日志的时候调用Log类的几个函数，比如e(TAG,...)，w(TAG,...)，d(TAG,...)，i(TAG,...)等。

在没有接触AOP之前，包括我在内，想到的解决方案就是上面这样的。但是，从OOP角度看，除了日志模块本身，其他模块的家务事绝大部分情况下应该都不会包含日志输出功能。但实际上，软件中的众多模块确实又需要打印日志。这个日志输出功能，从整体来看，都是一个面上的。而这个面的范围，就不局限在单个模块里了，而是横跨多个模块。

AOP的目标就是解决上面提到的不cool的问题。在AOP中：

* + 第一，我们要认识到OOP世界中，有些功能是横跨并嵌入众多模块里的，比如打印日志，比如统计某个模块中某些函数的执行时间等。这些功能在各个模块里分散得很厉害，可能到处都能见到。
  + 第二，AOP的目标是把这些功能集中起来，放到一个统一的地方来控制和管理。如果说，OOP如果是把问题划分到单个模块的话，那么AOP就是把涉及到众多模块的某一类问题进行统一管理。比如我们可以设计两个Aspects，一个是管理某个软件中所有模块的日志输出的功能，另外一个是管理该软件中一些特殊函数调用的权限检查。

讲了这么多，还是先来看个例子。在这个例子中，我们要：

* + Activity的生命周期的几个函数运行时，要输出日志。
  + 几个重要函数调用的时候，要检查有没有权限。

## 没有AOP的例子

先来看没有AOP的情况下，代码怎么写。主要代码都在AopDemoActivity中

public class AopDemoActivity extends Activity {

private static final String TAG = "AopDemoActivity";

onCreate，onStart,onRestart,onPause,onResume,onStop,onDestory返回前，都输出一行日志

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.layout\_main);

Log.e(TAG,"onCreate");

}

protected void onStart() {

super.onStart();

Log.e(TAG, "onStart");

}

protected void onRestart() {

super.onRestart();

Log.e(TAG, "onRestart");

}

protectedvoid onResume() {

super.onResume();

Log.e(TAG, "onResume");

checkPhoneState会检查app是否申明了android.permission.READ\_PHONE\_STATE权限

checkPhoneState();

}

protected void onPause() {

super.onPause();

Log.e(TAG, "onPause");

}

protected void onStop() {

super.onStop();

Log.e(TAG, "onStop");

}

protected void onDestroy() {

super.onDestroy();

Log.e(TAG, "onDestroy");

}

private void checkPhoneState(){

if(checkPermission("android.permission.READ\_PHONE\_STATE")== false){

Log.e(TAG,"have no permission to read phone state");

return;

}

Log.e(TAG,"Read Phone State succeed");

return;

}

private boolean checkPermission(String permissionName){

try{

PackageManager pm = getPackageManager();

//调用PackageMangaer的checkPermission函数，检查自己是否申明使用某权限

int nret = pm.checkPermission(permissionName,getPackageName());

return nret == PackageManager.PERMISSION\_GRANTED;

}......

}

}

代码很简单。但是从这个小例子中，你也会发现要是这个程序比较复杂的话，到处都加Log，或者在某些特殊函数加权限检查的代码，真的是一件挺繁琐的事情。

## AspectJ介绍

### AspectJ极简介

AOP虽然是方法论，但就好像OOP中的Java一样，一些先行者也开发了一套语言来支持AOP。目前用得比较火的就是AspectJ了，它是一种几乎和Java完全一样的语言，而且完全兼容Java（AspectJ应该就是一种扩展Java，但它不是像Groovy[1]那样的拓展。）。当然，除了使用AspectJ特殊的语言外，AspectJ还支持原生的Java，只要加上对应的AspectJ注解就好。所以，使用AspectJ有两种方法：

* 完全使用AspectJ的语言。这语言一点也不难，和Java几乎一样，也能在AspectJ中调用Java的任何类库。AspectJ只是多了一些关键词罢了。
* 或者使用纯Java语言开发，然后使用AspectJ注解，简称@AspectJ。

Anyway，不论哪种方法，最后都需要AspectJ的编译工具ajc来编译。由于AspectJ实际上脱胎于Java，所以ajc工具也能编译java源码。

AspectJ现在托管于Eclipse项目中，官方网站是：

* http://www.eclipse.org/aspectj/ <=AspectJ官方网站
* http://www.eclipse.org/aspectj/doc/released/runtime-api/index.html <=AspectJ类库参考文档，内容很少
* http://www.eclipse.org/aspectj/doc/released/aspectj5rt-api/index.html <=@AspectJ文档，以后我们用Annotation的方式最多。

### AspectJ语法

题外话：AspectJ东西比较多，但是AOP做为方法论，它的学习和体会是需要一步一步，并且一定要结合实际来的。如果一下子讲太多，反而会疲倦。更可怕的是，有些胆肥的同学要是一股脑把所有高级玩法全弄上去，反而得不偿失。这就是是方法论学习和其他知识学习不一样的地方。请大家切记。

#### Join Points介绍

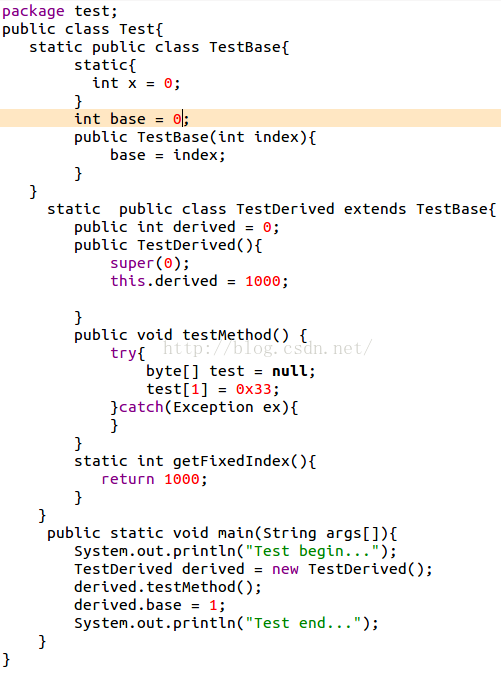
Join Points（以后简称JPoints）是AspectJ中最关键的一个概念。什么是JPoints呢？JPoints就是程序运行时的一些执行点。那么，一个程序中，哪些执行点是JPoints呢？比如：

* 一个函数的调用可以是一个JPoint。比如Log.e()这个函数。e的执行可以是一个JPoint，而调用e的函数也可以认为是一个JPoint。
* 设置一个变量，或者读取一个变量，也可以是一个JPoint。比如Demo类中有一个debug的boolean变量。设置它的地方或者读取它的地方都可以看做是JPoints。
* for循环可以看做是JPoint。

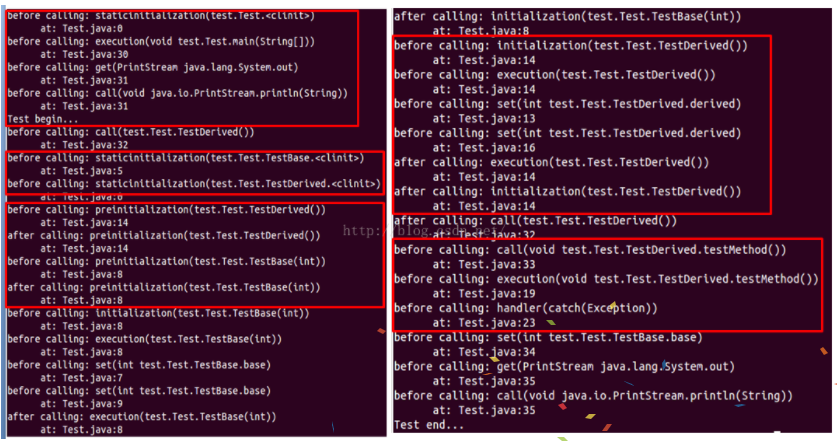
理论上说，程序中很多地方都可以被看做是JPoint，但是AspectJ中，只有如下标所示的几种执行点被认为是JPoints：

|  |  |  |
| --- | --- | --- |
| **Join Points** | **说明** | **示例** |
|
| method call | 函数调用 | 比如调用Log.e()，这是一处JPoint |
| method execution | 函数执行 | 比如Log.e()的执行内部，是一处JPoint。注意它和method call的区别，method call是调用某个函数的地方。而execution是某个函数执行的内部 |
| constructor call | 构造函数调用 | 和method call类似 |
| constructor execution | 构造函数执行 | 和method execution类似 |
| field get | 获取某个变量 | 比如读取DemoActivity.debug成员 |
| field set | 设置某个变量 | 比如设置DemoActivity.debug成员 |
| pre-initialization | Object在构造函数中做得一些工作 | 很少使用，详情见下面的例子 |
| initialization | Object在构造函数中做得工作 | 详情见下面的例子 |
| static initialization | 类初始化 | 比如类的static{} |
| handler | 异常处理 | 比如try catch(xxx)中，对应catch内的执行 |
| advice execution | 这个是AspectJ的内容，稍后再说 |  |

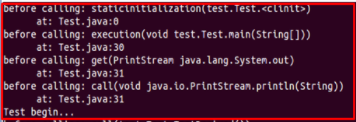
上表列出了AspectJ所认可的JoinPoints的类型。下面我们来看个例子以直观体会一把：



上面是一个Java示例代码，下面我们将打印出其中所有的join points。下图所示为打印出来的join points：



第一个红框：



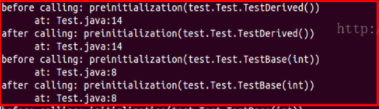
* staticinitialization(test.Test.<clinit>)：表示当前是哪种类型的JPoint，括号中代表目标对象是谁（此处是指Test class的类初始化）。由于Test类没有指定static block，所以后面的at:Test.java:0 表示代码在第0行（其实就是没有找到源代码的意思）。
* Test类初始化完后,就该执行main函数了。所以下一个JPoint就是execution(void test.Test.main(String[]))。括号中表示此JPoint对应的是test.Test.main函数。at:Test.java:30表示这个JPoint在源代码的第30行
* main函数里首先是执行System.out.println。而这一行代码实际包括两个JPoint。一个是get(PrintStream java.lang.System.out)，get表示Field get，它表示从System中获取out对象；另外一个是call(void java.io.PrintStream.println(String))，这是一个call类型的JPoint，表示执行out.println函数。

**第二个红框**



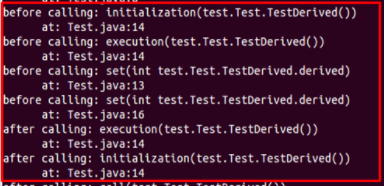
它表示TestBase和TestDerived的类的初始化，由于源码中为TestBase定义了static块，所以这个JPoint清晰指出了源码的位置

第三个红框



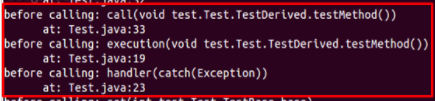
它和对象的初始化有关。在源码中，我们只是构造了一个TestDerived对象。它会先触发TestDerived Preinitialization JPoint，然后触发基类TestBase的PreInitialization JPoint。注意红框中的before和after 。在TestDerived和TestBase所对应的PreInitialization before和after中都没有包含其他JPoint。所以，Pre-Initialization应该是构造函数中一个比较基础的Phase。这个阶段不包括类中成员变量定义时就赋值的操作，也不包括构造函数中对某些成员变量进行的赋值操作。

第四个红框



成员变量的初始化（包括成员变量定义时就赋值的操作，比如源码中的int base = 0，以及在构造函数中所做的赋值操作，比如源码中的this.derived = 1000）都被囊括到initialization阶段

第五个红框



它包括三个JPoint：

* testMethod的call类型JPoint
* testMethod的execution类型JPonint
* 以及对异常捕获的Handler类型JPoint

好了,JPoint的介绍就先到此,简单直白粗暴点说，Join Point就是一个程序中的关键函数（包括构造函数）和代码段。为什么AspectJ首先要定义好JoinPoint呢？大家仔细想想就能明白，以打印log的AopDemo为例，log在哪里打印？自然是在一些关键点去打印。而谁是关键点？AspectJ定义的这些类型的JPoint就能满足我们绝大部分需求。

注意，要是想在一个for循环中打印一些日志，而AspectJ没有这样的JPoint，所以这个需求我们是无法利用AspectJ来实现了。另外，不同的软件框架对表1中的JPoint类型支持也不同。比如Spring中，不是所有AspectJ支持的JPoint都有。

#### Pointcuts介绍

Pointcuts是什么呢？前面介绍的内容可知，一个程序会有很多的JPoints，即使是同一个函数（比如testMethod这个函数），还分为call类型和execution类型的JPoint。显然，不是所有的JPoint，也不是所有类型的JPoint都是我们关注的。再次以AopDemo为例，我们只要求在Activity的几个生命周期函数中打印日志，只有这几个生命周期函数才是我们业务需要的JPoint，而其他的什么JPoint我不需要关注。

怎么从一堆一堆的JPoints中选择自己想要的JPoints呢？恩，这就是Pointcuts的功能。一句话，Pointcuts的目标是提供一种方法使得开发者能够选择自己感兴趣的Join Points。

在上图的例子中，怎么把Test.java中所有的Joinpoint选择出来呢？用到的pointcut格式为：

Pointcut testAll():within(Test)

AspectJ中，pointcut有一套标准语法，涉及的东西很多，还有一些比较高级的玩法。那就一步一步来吧。

##### 一个Pointcuts例子

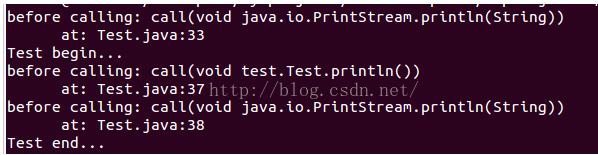
直接来看一个例子，现在我想把上图中的示例代码中，那些调用println的地方找到，该怎么弄？代码该这么写：

public pointcut testAll(): call(public \* \*.println(..)) && !within(TestAspect);

注意，aspectj的语法和Java一样，只不过多了一些关键词，我们来看看上述代码：

* 第一个public ~ 表示这个pointcut是public访问。这主要和aspect的继承关系有关，属于AspectJ的高级玩法，本文不考虑。
* pointcut ~ 关键词，表示这里定义的是一个pointcut。pointcut定义有点像函数定义。总之，在AspectJ中，你得定义一个pointcut。
* testAll() ~ pointcut的名字。在AspectJ中，定义Pointcut可分为有名和匿名两种办法。个人建议使用有名方法。因为在后面，我们要使用一个pointcut的话，就可以直接使用它的名字就好。
* testAll后面的冒号 ~ 这是pointcut定义名字后，必须加上。冒号后面是这个pointcut怎么选择Joinpoint的条件。
* call() ~ 表示我们选择的Joinpoint类型为call类型。
* public \* \*.println(..) ~ 这小段代码使用了通配符。由于我们这里选择的JoinPoint类型为call类型，它对应的目标JPoint一定是某个函数。所以我们要找到这些函数。
  + public 表示目标JPoint的可见性修饰符。
  + 第一个\* ~ 表示返回值的类型是任意类型。
  + 第二个\* ~ 用来指明包名。此处不限定包名。
  + println() ~ 是函数名。这表明我们选择的函数是任何包中定义的名字叫println的函数。
  + 通配符.. ~ 代表名为println的任意个数的参数，任意类型的参数的函数。
* call后面的&& ~ AspectJ可以把几个条件组合起来，目前支持 &&，||，！
* !within(TestAspectJ)：前面的!表示不满足某个条件。within是和call同级的另外一种类型，特别注意，这种类型和前面讲到的join point的那几种类型不同。within的类型是数据类型，而join point的类型更像是动态的，执行时的类型。

上例中的pointcut合起来就是：选择那些调用println，且不考虑println函数的参数是什么的Join point。另外，调用者的类型不要是TestAspect的。执行结果如下：



##### 语法规则

匹配规则

| **表达式** | **含义** |
| --- | --- |
| java.lang.String | 匹配String类型 |
| java.\*.String | 匹配java包下任何“一级子包”下的String类型，如匹配java.lang.String，但不匹配java.lang.ss.String |
| java..\*.String | 匹配java包下任何“X级子包（X包括0）”下的String类型，如匹配java.lang.String，同时匹配java.lang.ss.String |
| java..String | 同上java..\*.String |
| java..\* | 匹配java包及任何子包下的任何类型,如匹配java.lang.String、java.lang.annotation.Annotation |
| java.lang.\*ing | 匹配任何java.lang包下的以ing结尾的类型 |
| java.lang.Number+ | 匹配java.lang包下的任何Number的自类型，如匹配java.lang.Integer，也匹配java.math.BigInteger |

注意

注意：

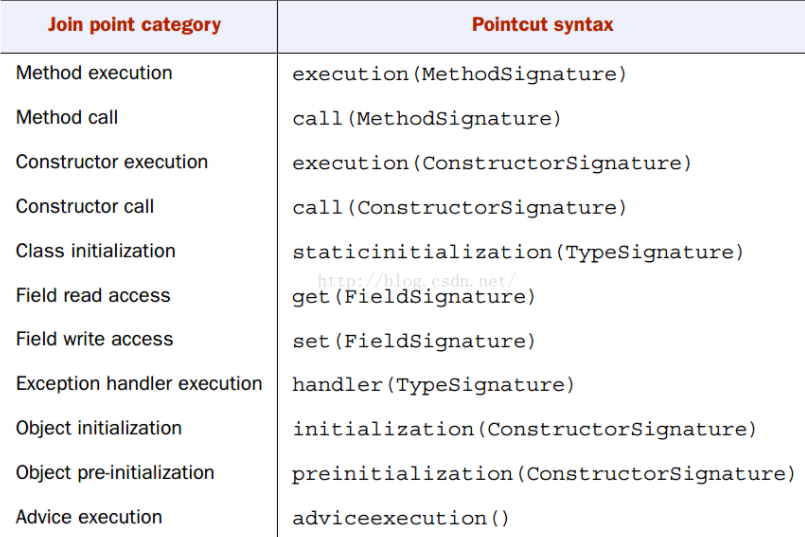
* java包的零级子包下TestActivity类中的String方法，其实就是java包下的一级子包了，因为该String的路径为java.TestActivity.String
* 一个\*和几个\*没有区别

| **参数** | **含义** |
| --- | --- |
| () | 表示方法没有任何参数 |
| (..) | 表示匹配接受任意个参数的方法 |
| (.., java.lang.String) | 表示匹配接受java.lang.String类型的参数结束，且其前边可以接受任意个参数 |
| (java.lang.String,..) | 表示匹配接受java.lang.String类型的参数开始，且其后边可以接受任意个参数 |
| (\*, java.lang.String) | 表示匹配接受java.lang.String类型的参数结束，且其前边接受一个参数 |

Tip：execution(\* com.chejin360.digitalcoins.\*.onSingleClick(android.view.View))表示切入点为:com.chejin360.digitalcoins的子包下的所有publci void onSingleClick(View v)方法的执行点

##### 直接针对JoinPoint的选择

pointcuts中最常用的选择条件和Join point的类型密切相关，比如下图：



如果我们想选择类型为methodexecution的JPoint，那么pointcuts的写法就得包括execution(XXX)来限定。

除了指定JPoint类型外，我们还要更进一步选择目标函数。选择的根据就是上图中列出的什么MethodSignature，ConstructorSignature，TypeSinature，FieldSignature等。名字听起来陌生得很，其实就是指定JPoint对应的函数（包括构造函数），Static block的信息。比如图那个println的Pointcuts例子，首先它的JPoint类型是call，所以它的查询条件是根据MethodSignature来表达。

一个Method Signature的完整表达式为：

public \* \*.println(..) @注解 访问权限 返回值的类型 包名.函数名(参数)

* @注解和访问权限（public/private/protect，以及static/final）属于可选项。如果不设置它们，则默认都会选择。以访问权限为例，如果没有设置访问权限作为条件，那么public，private，protect及static、final的函数都会进行搜索。
* 返回值类型就是普通的函数的返回值类型。如果不限定类型的话，就用\*通配符表示
* 包名.函数名用于查找匹配的函数。可以使用通配符，包括\*和..以及+号。其中\*号用于匹配除.号之外的任意字符，而..则表示任意子package，+号表示子类。比如：
  + java.\*.Date：可以表示java.sql.Date，也可以表示java.util.Date
  + Test\*：可以表示TestBase，也可以表示TestDervied
  + java..\*：表示java任意子类
  + java..\*Model+：表示Java任意package中名字以Model结尾的子类，比如TabelModel，TreeModel等
* 最后来看函数的参数。参数匹配比较简单，主要是参数类型，比如：
* (int, char)：表示参数只有两个，并且第一个参数类型是int，第二个参数类型是char
* (String, ..)：表示至少有一个参数。并且第一个参数类型是String，后面参数类型不限。在参数匹配中，..代表任意参数个数和类型
* (Object ...)：表示不定个数的参数，且类型都是Object，这里的...不是通配符，而是Java中代表不定参数的意思

一个Constructor Signature的完整表达式为：

它和Method Signature类似，只不过构造函数没有返回值，而且函数名必须叫new。比如：

public \*..TestDerived.new(..) @注解 访问权限 包名.函数名(参数)

* public：选择public访问权限
* \*..代表任意包名
* TestDerived.new：代表TestDerived的构造函数
* (..)：代表参数个数和类型都是任意

一个Field Signature的完整表达式为：

它和Method Signature类似，只不过构造函数没有返回值，而且函数名必须叫new。比如：

set(inttest..TestBase.base) @注解 访问权限 类型 类名.成员变量名

* 其中，@注解和访问权限是可选的
* 类型：成员变量类型，\*代表任意类型
* 类名.成员变量名：成员变量名可以是\*，代表任意成员变量
* 比如，set(inttest..TestBase.base)：表示设置TestBase.base变量时的JPoint

一个Type Signature的完整表达式为：

* staticinitialization(test..TestBase)：表示TestBase类的static block
* handler(NullPointerException)：表示catch到NullPointerException的JPoint。注意，上图截获的其实是Exception，其真实类型是NullPointerException。但是由于JPointer的查询匹配是静态的，即编译过程中进行的匹配，所以handler(NullPointerException)在运行时并不能真正被截获。只有改成handler(Exception)，或者把源码的Exception改成NullPointerException才行

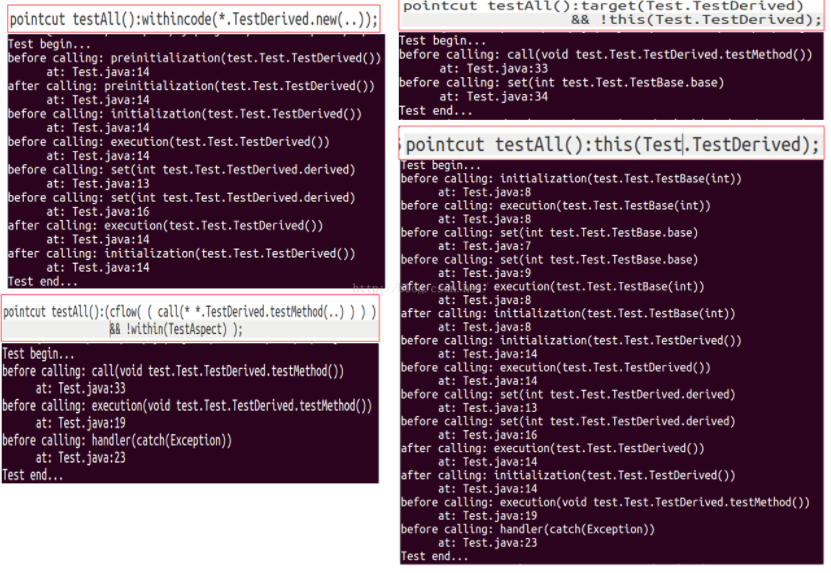
##### 间接针对JPoint的选择

除了根据前面提到的Signature信息来匹配JPoint外，AspectJ还提供其他一些选择方法来选择JPoint。比如某个类中的所有JPoint，每一个函数执行流程中所包含的JPoint。特别强调，不论什么选择方法，最终都是为了找到目标的JPoint。

下表列出了一些常用的非JPoint选择方法：

|  |  |  |
| --- | --- | --- |
| 关键词 | 说明 | 示例 |
| within(TypePattern) | TypePattern标示package或者类。TypePatter可以使用通配符 | 表示某个Package或者类中的所有JPoint。比如within(Test)：Test类中（包括内部类）所有JPoint。上面所示的例子就是用这个方法。 |
| withincode(Constructor Signature|Method Signature) | 表示某个构造函数或其他函数执行过程中涉及到的JPoint | 比如withinCode(\* TestDerived.testMethod(..))。表示testMethod涉及的JPoint。withinCode( \*.Test.new(..))表示Test构造函数涉及的JPoint |
| cflow(pointcuts) | cflow是call flow的意思,cflow的条件是一个pointcut | 比如cflow(call TestDerived.testMethod)：表示调用TestDerived.testMethod函数时所包含的JPoint，包括testMethod的call这个JPoint本身 |
| cflowbelow(pointcuts) | cflow是call flow的意思。 | 比如cflowblow(call TestDerived.testMethod)：表示调用TestDerived.testMethod函数时所包含的JPoint，不包括testMethod的call这个JPoint本身 |
| this(Type) | JPoint的this对象是Type类型。（其实就是判断Type是不是某种类型，即是否满足instanceof Type的条件） | JPoint是代码段（不论是函数，异常处理，static block），从语法上说，它都属于一个类。如果这个类的类型是Type标示的类型，则和它相关的JPoint将全部被选中。上面示例的testMethod是TestDerived类。所以this(TestDerived)将会选中这个testMethod JPoint |
| target(Type) | JPoint的target对象是Type类型 | 和this相对的是target。不过target一般用在call的情况。call一个函数，这个函数可能定义在其他类。比如testMethod是TestDerived类定义的。那么target(TestDerived)就会搜索到调用testMethod的地方。但是不包括testMethod的execution JPoint |
| args(TypeSignature) | 用来对JPoint的参数进行条件搜索的 | 比如args(int,..)，表示第一个参数是int，后面参数个数和类型不限的JPoint。 |

下图为修改示例和输出：注意：this()和target()匹配的时候不能使用通配符。



注意，不是所有的AOP实现都支持本节所说的查询条件。比如Spring就不支持withincode查询条件。

### advice和aspect介绍

现在，我们知道如何通过pointcuts来选择合适的JPoint。那么，下一步工作就很明确了，选择这些JPoint后，我们肯定是需要干一些事情的。比如前面例子中的输出都有before，after之类的。这其实JPoint在执行前，执行后，都执行了一些我们设置的代码。在AspectJ中，这段代码叫advice。简单点说，advice就是一种Hook。

ASpectJ中有好几个Hook，主要是根据JPoint执行时机的不同而不同，比如下面的：testAll()是前面定义的pointcuts，而before()定义了在这个pointcuts选中的JPoint执行前我们要干的事情。

before(): testAll(){

System.out.println("before calling: " + thisJoinPoint);//打印这个JPoint的信息

System.out.println(" at:" + thisJoinPoint.getSourceLocation());//打印这个JPoint对应的源代码位置

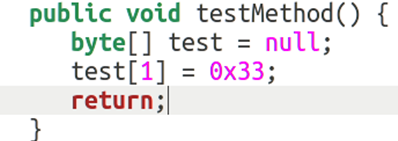
}

下表列出了AspectJ所支持的Advice的类型：

|  |  |  |
| --- | --- | --- |
| 关键词 | 说明 | 示例 |
|
| before() | before advice | 表示在JPoint执行之前，需要干的事情 |
| after() | after advice | 表示JPoint自己执行完了后，需要干的事情。 |
| after():returning(返回值类型) | returning和throwing后面都可以指定具体的类型，如果不指定的话则匹配的时候不限定类型 | 假设JPoint是一个函数调用的话，那么函数调用执行完有两种方式退出，一个是正常的return，另外一个是抛异常。 |
| after():throwing(异常类型) | 注意，after()默认包括returning和throwing两种情况 |
| 返回值类型 around() | before和around是指JPoint执行前或执行后备触发，而around就替代了原JPoint | around是替代了原JPoint，如果要执行原JPoint的话，需要调用proceed |

注意，after和before没有返回值，但是around的目标是替代原JPoint的，所以它一般会有返回值，而且返回值的类型需要匹配被选中的JPoint。我们来看个例子：

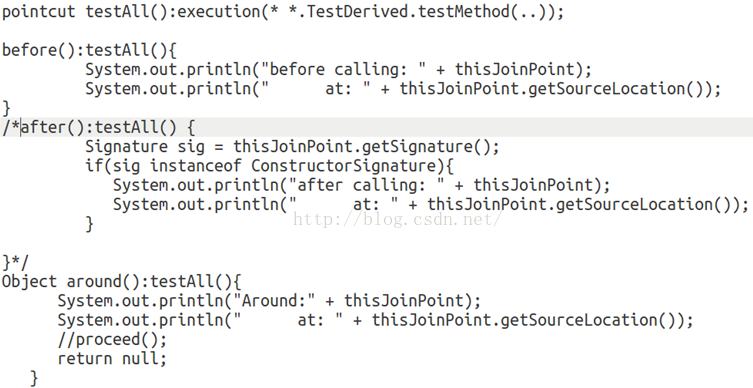
testMethod，在这个testMethod中，肯定会抛出一个空指针异常。

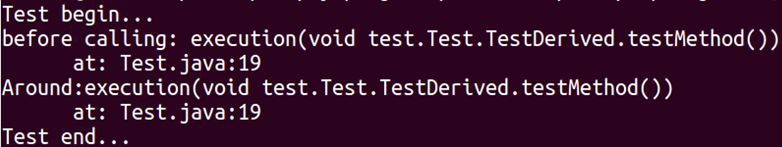


我们配置的advice，除了before以外，还加了一个around。我们重点来看around，它的返回值是Object。虽然匹配的JPoint是testMethod，其定义的返回值是void。但是AspectJ考虑的很周到。在around里，可以设置返回值类型为Object来表示返回任意类型的返回值。AspectJ在真正返回参数的时候，会自动进行转换。比如，假设inttestMethod定义了int作为返回值类型，我们在around里可以返回一个Integer，AspectJ会自动转换成int作为返回值。

再看around中的//proceed()这句话。这代表调用真正的JPoint函数，即testMethod。由于这里我们屏蔽了proceed，所以testMethod真正的内容并未执行，故运行的时候空指针异常就不会抛出来。也就是说，我们完全截获了testMethod的运行，甚至可以任意修改它，让它执行别的函数都没有问题。。

注意：从技术上说，around是完全可以替代before和after的。图7中第二个红框还把after给注释掉了。如果不注释掉，编译时候报错，[error]circular advice precedence: can't determine precedence between two or morepieces of advice that apply to the same join point: method-execution(voidtest.Test$TestDerived.testMethod())我猜测其中的原因是around和after冲突了。around本质上代表了目标JPoint，比如此处的testMethod。而after是testMethod之后执行。那么这个testMethod到底是around还是原testMethod呢？真是傻傻分不清楚！





### 回顾前面三节的内容

* AspectJ中各种类型的Join Point，JPoint是一个程序的关键执行点，也是我们关注的重点。
* pointcuts：提供了一种方法来选择目标JPoint。程序有很多JPoint，但是需要一种方法来让我们选择我们关注的JPoint。这个方法就是利用pointcuts来完成的。
* 通过pointcuts选择了目标JPoint后，我们总得干点什么吧？这就用上了advice。advice包括好几种类型，一般情况下都够我们用了。

上面这些东西都有点像函数定义，在Java中，这些东西都是要放到一个class里的。在AspectJ中，也有类似的数据结构，叫aspect。

public aspect 名字 {//aspect关键字和class的功能一样，文件名以.aj结尾

pointcuts定义...

advice定义...

}

通过这种方式，定义一个aspect类，就把相关的JPoint和advice包含起来，形成了一个“关注面”，比如：

* 我们定义一个LogAspect，在LogAspect中，我们在关键JPoint上设置advice，这些advice就是打印日志
* 再定义一个SecurityCheckAspect，在这个Aspect中，我们在关键JPoint上设置advice，这些advice将检查调用app是否有权限。

通过这种方式，我们在原来的JPoint中，就不需要写log打印的代码，也不需要写权限检查的代码了。所有这些关注点都挪到对应的Aspectj文件中来控制。恩，这就是AOP的精髓。注意，读者在把玩代码时候，一定会碰到AspectJ语法不熟悉的问题。所以请读者记得随时参考官网的文档：

http://www.eclipse.org/aspectj/doc/released/quick5.pdf

<http://www.eclipse.org/aspectj/doc/released/progguide/semantics.html>

### 参数传递和JPoint信息

#### 参数传递

到此，AspectJ最基本的东西其实讲差不多了，但是在实际使用AspectJ的时候，你会发现前面的内容还欠缺一点，尤其是advice的地方：

* 前面介绍的advice都是没有参数信息的，而JPoint肯定是或多或少有参数的。而且advice既然是对JPoint的截获或者hook也好，肯定需要利用传入给JPoint的参数干点什么事情。比方所around advice，我可以对传入的参数进行检查，如果参数不合法，我就直接返回，根本就不需要调用proceed做处理。

往advice传参数比较简单，就是利用前面提到的this(),target(),args()等方法。另外，整个pointcuts和advice编写的语法也有一些区别。具体方法如下：

1. 先在pointcuts定义时候指定参数类型和名字

pointcut testAll(Test.TestDerived derived, int x): call(\*Test.TestDerived.testMethod(..))

&& target(derived) && args(x)

注意上述pointcuts的写法，首先在testAll中定义参数类型和参数名。这一点和定义一个函数完全一样

1. 接着看target和args。此处的target和args括号中用得是参数名。而参数名则是在前面pointcuts中定义好的。这属于target和args的另外一种用法。

注意，增加参数并不会影响pointcuts对JPoint的匹配，上面的pointcuts选择和pointcut testAll():call(\*Test.TestDerived.testMethod(..)) && target(Test.TestDerived) &&args(int)是一样的。只不过我们需要把参数传入advice，才需要改造

1. 接下来是修改advice：

Object around(Test.TestDerived derived,int x):testAll(derived,x){

System.out.println(" arg1=" + derived);

System.out.println(" arg2=" + x);

return proceed(derived, x); *//注意，proceed就必须把所有参数传进去。*

}

* advice的定义现在也和函数定义一样，把参数类型和参数名传进来。
* 接着把参数名传给pointcuts，此处是testAll。注意，advice必须和使用的pointcuts在参数类型和名字上一致。
* 然后在advice的代码中，你就可以引用参数了，比如derived和x，都可以打印出来。

总结，参数传递其实并不复杂，关键是得记住语法：

* pointcuts修改：像定义函数一样定义pointcuts，然后在this,target或args中绑定参数名（注意，不再是参数类型，而是参数名）。
* advice修改：也像定义函数一样定义advice，然后在冒号后面的pointcuts中绑定参数名（注意是参数名）
* 在advice的代码中使用参数名。

#### JoinPoint信息收集

我们前面示例中都打印出了JPoint的信息，比如当前调用的是哪个函数，JPoint位于哪一行代码。这些都属于JPoint的信息。AspectJ为我们提供如下信息：

* thisJoinpoint对象：在advice代码中可直接使用。代表JPoint每次被触发时的一些动态信息，比如参数啊之类的、
* thisJoinpointStatic对象：在advice代码中可直接使用，代表JPoint中那些不变的东西。比如这个JPoint的类型，JPoint所处的代码位置等。
* thisEnclosingJoinPointStaticPart对象：在advice代码中可直接使用。也代表JPoint中不可变的部分，但是它包含的东西和JPoint的类型有关，比如对一个call类型JPoint而言，thisEnclosingJoinPointStaticPart代表包含调用这个JPoint的函数的信息。对一个handler类型的JPoint而言，它代表包含这个try/catch的函数的信息。

thisJoinpoint的API文档地址：http://www.eclipse.org/aspectj/doc/released/runtime-api/index.html

## 使用AOP的例子

现在正式回到我们的AndroidAopDemo这个例子来。我们的目标是为AopDemoActivity的几个Activity生命周期函数加上log，另外为checkPhoneState加上权限检查。一切都用AOP来集中控制。

前面提到说AspectJ需要编写aj文件，然后把AOP代码放到aj文件中。但是在Android开发中，我建议不要使用aj文件。因为aj文件只有AspectJ编译器才认识，而Android编译器不认识这种文件。所以当更新了aj文件后，编译器认为源码没有发生变化，所以不会编译它。

当然，这种问题在其他不认识aj文件的java编译环境中也存在。所以，AspectJ提供了一种基于注解的方法来把AOP实现到一个普通的Java文件中。这样我们就把AOP当做一个普通的Java文件来编写、编译就好。

### 打印Log

马上来看AopDemoActivity对应的DemoAspect.java文件吧。先看输出日志第一版本：

import android.util.Log;  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;  
/\* 必须使用@AspectJ标注，这样class DemoAspect就等同于 aspect DemoAspect了 \*/  
@Aspect  
public class DemoAspect {  
 private static final String *TAG* = "DemoAspect";  
 /\* @Pointcut：pointcut也变成了一个注解，这个注解是针对一个函数的，比如此处的logForActivity()  
 \* 其实它代表了这个pointcut的名字。如果是带参数的pointcut，则把参数类型和名字放到  
 \* 代表pointcut名字的logForActivity中，然后在@Pointcut注解中使用参数名。  
 \* 基本和以前一样，只是写起来比较奇特一点。后面我们会介绍带参数的例子 \*/  
 @Pointcut("execution(\* com.androidaop.demo.AopDemoActivity.onCreate(..)) ||"  
 + "execution(\* com.androidaop.demo.AopDemoActivity.onStart(..))")  
 public void logForActivity() {  
 //注意，这个函数必须要有实现，否则Java编译器会报错  
 }  
 /\* @Before：这就是Before的advice，对于after，after-returning，和after-throwing。对于的注解格式为@After，  
 \* @AfterReturning，@AfterThrowing。Before后面跟的是pointcut名字，然后其代码块由一个函数来实现。比如此处的log \*/  
 @Before("logForActivity()")  
 public void log(JoinPoint joinPoint) {  
 //对于使用Annotation的AspectJ而言，JoinPoint就不能直接在代码里得到多了，而需要通过参数传递进来。  
 Log.*e*(*TAG*, joinPoint.toShortString());  
 }  
}

上面的例子仅仅是列出了onCreate和onStart两个函数的日志，如果想在所有的onXXX这样的函数里加上log，该怎么改呢？

@Pointcut("execution(\* \*..AopDemoActivity.on\*(..))")

public void logForActivity(){};

### 检查权限

#### 使用注解

检查权限这个功能的实现也可以采用刚才打印log那样，但是这样就没有太多意思了。我们玩点高级的。不过这个高级的玩法也是来源于现实需求：

* 权限检查一般是针对API的，比如调用者是否有权限调用某个函数。
* API往往是通过SDK发布的。一般而言，我们会在这个函数的注释里说明需要调用者声明哪些权限。
* 然后我们在API检查调用者是不是申明了文档中列出的权限。

如果我有10个API，10个不同的权限，那么在10个函数的注释里都要写，太麻烦了。怎么办？这个时候我想到了注解。注解的本质是源代码的描述。权限声明，从语义上来说，其实是属于API定义的一部分，二者是一个统一体，而不是分离的。

Java提供了一些默认的注解，不过此处我们要使用自己定义的注解：

import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
/\* 第一个@Target表示这个注解只能给函数使用  
 \* 第二个@Retention表示注解内容需要包含的Class字节码里，属于运行时需要的  
 \* @interface用于定义一个注解 \*/  
@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface SecurityCheckAnnotation {  
 //declarePermission是一个函数，其实代表了注解里的参数  
 String declaredPermission();  
}

#### 检查权限

下面，我们来看看如何在AspectJ中，充分利用这注解信息来帮助我们检查权限。

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;

@Aspect  
public class SecurityCheckAspect {  
 private static final String *TAG* = "DemoAspect";  
 /\* 来看这个Pointcut，首先，它在选择Jpoint的时候，把@SecurityCheckAnnotation使用上了，  
 \* 这表明所有那些public的，并且携带有这个注解的API都是目标JPoint  
 \* 接着，由于我们希望在函数中获取注解的信息，所有这里的poincut函数有一个参数，参数类型是  
 \* SecurityCheckAnnotation，参数名为ann这个参数我们需要在后面的advice里用上，  
 \* 所以pointcut还使用了@annotation(ann)这种方法来告诉AspectJ，这个ann是一个注解 \*/  
 @Pointcut("execution(@SecurityCheckAnnotation public \* \*..\*.\*(..)) && @annotation(ann)")  
 public void checkPermission(SecurityCheckAnnotation ann) {  
 }  
  
 /\* 接下来是advice，advice的真正功能由check函数来实现，这个check函数第二个参数就是我们想要  
 \* 的注解。在实际运行过程中，AspectJ会把这个信息从JPoint中提出出来并传递给check函数 \*/  
 @Before("checkPermission(securityCheckAnnotation)")  
 public void check(JoinPoint joinPoint, SecurityCheckAnnotation securityCheckAnnotation) {  
 //从注解信息中获取声明的权限。  
 String neededPermission = securityCheckAnnotation.declaredPermission();  
 Log.*e*(*TAG*, joinPoint.toShortString());  
 Log.*e*(*TAG*, "\tneeded permission is " + neededPermission);  
 }  
}

#### 和其他模块交互

事情这样就完了？很明显没有。为什么？刚才权限检查只是简单得打出了日志，但是并没有真正去做权限检查。如何处理？这就涉及到AOP如何与一个程序中其他模块交互的问题了。初看起来容易，其实有难度。

比如，SecurityCheckAspect 虽然是一个类，但是没有构造函数。而且，我们也没有在代码中主动去构造它。根据AsepctJ的说明，SecurityCheckAspect 不需要我们自己去构造，AspectJ在编译的时候会把构造函数给你自动加上。具体在程序什么位置加上，其实是有规律的，但是我们并不知道，也不要去知道。

这样的话，SecurityCheckAspect 岂不是除了打打log就没什么作用了？非也！以此例的权限检查为例，我们需要：

* 把真正进行权限检查的地方封装到一个模块里，比如SecurityCheck中。
* SecurityCheck往往在一个程序中只会有一个实例。所以可以为它提供一个函数，比如getInstance以获取SecurityCheck实例对象。
* 我们就可以在SecurityCheckAspect中获取这个对象，然后调用它的check函数，把最终的工作由SecurityCheck来检查了。

恩，这其实是Aspect的真正作用，它负责收集Jpoint，设置advice。一些简单的功能可在Aspect中来完成，而一些复杂的功能，则只是有Aspect来统一收集信息，并交给专业模块来处理。最终代码：

@Before("checkPermssion(securityCheckAnnotation)")  
public void check(JoinPoint joinPoint, SecurityCheckAnnotation securityCheckAnnotation){  
 String neededPermission = securityCheckAnnotation.declaredPermission();  
 Log.*e*(*TAG*, "\tneeded permission is " + neededPermission);  
 SecurityCheckManager manager = SecurityCheckManager.*getInstance*();  
 if(!manager.checkPermission(neededPermission)){  
 throw new SecurityException("Need to declare permission:" + neededPermission);  
 }  
}

## 其他、总结和参考文献

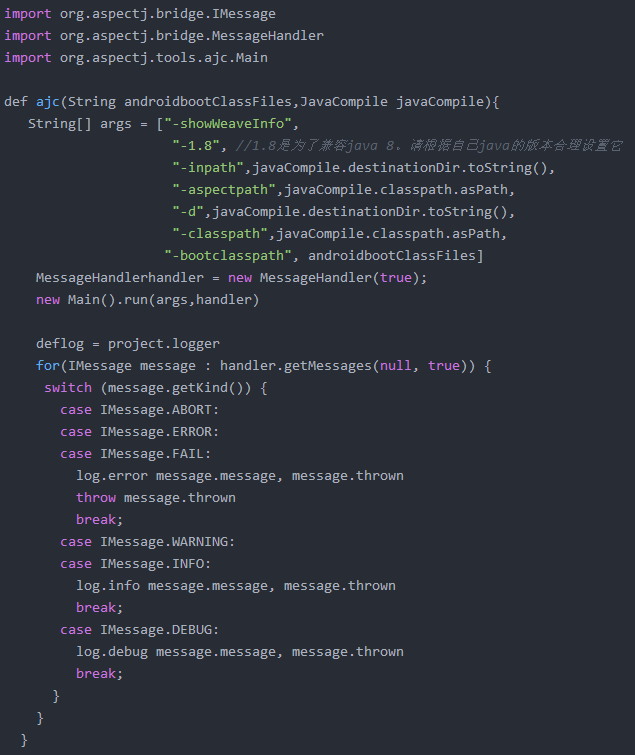
### AspectJ编译

* AspectJ比较强大，除了支持对source文件（即aj文件、或@AspectJ注解的Java文件，或普通java文件）直接进行编译外， 还能对Java字节码（即对class文件）进行处理。
* 有感兴趣的同学可以对aspectj-test小例子的class文件进行反编译，你会发现AspectJ无非是在被选中的JPoint的地方加一些hook函数。当然Before就是在调用JPoint之前加，After就是在JPoint返回之前加。
* 更高级的做法是当class文件被加载到虚拟机后，由虚拟机根据AOP的规则进行hook。

在Android里边，我们用得是第二种方法，即对class文件进行处理。来看看代码：



ajc函数其实和我们手动试玩aspectj-test目标一样，只是我们没有直接调用ajc命令，而是利用AspectJ提供的API做了和ajc命令一样的事情。



## 按钮防颤的例子

### project.build.gradle

buildscript {  
 dependencies {  
 //aspectj  
 classpath 'org.aspectj:aspectjtools:1.8.13'  
 classpath 'org.aspectj:aspectjweaver:1.8.13'  
 }  
}

### app.build.gradle

android {  
 dependencies {  
 //aspectj  
 implementation 'org.aspectj:aspectjrt:1.8.13'  
 }  
}

import org.aspectj.bridge.IMessage  
import org.aspectj.bridge.MessageHandler  
import org.aspectj.tools.ajc.Main  
final def log = project.logger  
final def variants = project.android.applicationVariants  
variants.all { variant ->  
 if (!variant.buildType.isDebuggable()) {  
 log.debug("Skipping non-debuggable build type '${variant.buildType.name}'.")  
 return  
 }  
 JavaCompile javaCompile = variant.javaCompile  
 javaCompile.doLast {  
 String[] args = ["-showWeaveInfo",  
 "-1.5",  
 "-inpath", javaCompile.destinationDir.toString(),  
 "-aspectpath", javaCompile.classpath.asPath,  
 "-d", javaCompile.destinationDir.toString(),  
 "-classpath", javaCompile.classpath.asPath,  
 "-bootclasspath", project.android.bootClasspath.join(File.*pathSeparator*)]  
 log.debug "ajc args: " + Arrays.*toString*(args)  
 MessageHandler handler = new MessageHandler(true)  
 new Main().run(args, handler)  
 for (IMessage message : handler.getMessages(null, true)) {  
 switch (message.getKind()) {  
 case IMessage.*ABORT*:  
 case IMessage.*ERROR*:  
 case IMessage.*FAIL*:  
 log.error message.message, message.thrown  
 break  
 case IMessage.*WARNING*:  
 log.warn message.message, message.thrown  
 break  
 case IMessage.*INFO*:  
 log.info message.message, message.thrown  
 break  
 case IMessage.*DEBUG*:  
 log.debug message.message, message.thrown  
 break  
 }  
 }  
 }  
}

### ids.xml

<?xml version="1.0" encoding="utf-8"?>  
<resources>  
 <item name="click\_time" type="id"/>  
</resources>

### SingleClick

#### kotlin

@Retention(AnnotationRetention.BINARY)  
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.PROPERTY\_GETTER,AnnotationTarget.PROPERTY\_SETTER)  
annotation class SingleClick

#### Java

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface SingleClick {  
}

### SingleClickAspect

#### kotlin

package com.chejin360.digitalcoins.aop  
import android.util.Log  
import android.view.View  
import com.chejin360.digitalcoins.R  
import org.aspectj.lang.ProceedingJoinPoint  
import org.aspectj.lang.annotation.Around  
import org.aspectj.lang.annotation.Aspect  
import org.aspectj.lang.annotation.Pointcut  
import java.util.\*  
@Aspect  
class SingleClickAspect {  
 @Pointcut(*POINT\_CUT\_REG*) //方法切入点  
 fun methodAnnotated() {  
 }  
 @Around("methodAnnotated()")//在连接点进行方法替换  
 @Throws(Throwable::class)  
 fun aroundJoinPoint(joinPoint: ProceedingJoinPoint) {  
 var view: View? = null  
 for (arg in joinPoint.*args*) {  
 if (arg is View) view = arg  
 }  
 view?.*let* **{** //获取当前view的tag中保存的上一次点击时间戳  
 val tag = view.getTag(*TIME\_TAG*)  
 val lastClickTime = if (tag != null) tag as Long else 0  
 if (BuildConfig.*DEBUG*) Log.d(*TAG*, "lastClickTime:$lastClickTime")  
 //获取当前点击时间戳  
 val currentTime = Calendar.getInstance().*timeInMillis* //比较两次点击的间隔时间，大于800毫秒才处理  
 if (currentTime - lastClickTime > *MIN\_CLICK\_DELAY\_TIME*) {  
 //把当前时间保存进view的tag中  
 view.setTag(*TIME\_TAG*, currentTime)  
 if (BuildConfig.*DEBUG*) Log.d(*TAG*, "currentTime:$currentTime")  
 //执行原方法  
 joinPoint.proceed()  
 }  
 **}** }  
}  
const val *TAG* = "singleClick"  
const val *MIN\_CLICK\_DELAY\_TIME* = 800  
const val *TIME\_TAG* = R.id.*click\_time*

const val *POINT\_CUT\_ITEMCLICK* = "execution(\* com.chejin360.digitalcoins..onItemClick(android.view.View, ..))"  
const val *POINT\_CUT\_SINGLECLICK* = "execution(\* com.chejin360.digitalcoins..onSingleClick(android.view.View))"  
const val *POINT\_CUT\_REG* = "$*POINT\_CUT\_ITEMCLICK* || $*POINT\_CUT\_SINGLECLICK*"

const val *POINT\_CUT\_REG2 =* "execution(@com.chejin360.digitalcoins.aop.SingleClick \* \*(..))"

#### Java

@Aspect  
public class SingleClickAspect {  
 public static final String *TAG* = "singleClick";  
 private static final long *MIN\_CLICK\_DELAY\_TIME* = 800L;  
 private static final int *TIME\_TAG* = R.id.*click\_time*;  
 private static final String *POINT\_CUT\_AtSingleClick* = "execution(@com.imlianka.lkapp.aop.SingleClick \* \*(..))";  
 private static final String *POINT\_CUT\_viewOnClick* = "execution(\* android.view.View.OnClickListener.onClick(..))";  
 private static final String *POINT\_CUT\_butterknife\_OnClick* = "execution(@butterknife.OnClick \* \*(..))";  
 private static final String *POINT\_CUT\_ItemClick* = "execution(\* com.chad.library.adapter.base.BaseQuickAdapter.OnItemClickListener..onItemClick(..))";  
 private static final String *POINT\_CUT\_ItemChildClick* = "execution(\* com.chad.library.adapter.base.BaseQuickAdapter.OnItemChildClickListener.onItemChildClick(..))";  
 private static final String *POINT\_CUT\_REG\_DEBUG* = "";  
 private static final String *POINT\_CUT\_REG\_RELEASE* = *POINT\_CUT\_AtSingleClick* + "||" + *POINT\_CUT\_viewOnClick* + "||" + *POINT\_CUT\_butterknife\_OnClick* // 新版的为了效率，不兼容lambda表达式，为了安全这俩可能被lambda的地方，手动写上@SingleClick注解

// + "||" + POINT\_CUT\_ItemChildClick + "||" + POINT\_CUT\_ItemClick  
 ;  
 @Pointcut(*POINT\_CUT\_REG\_DEBUG*) //方法切入点  
 public void methodAnnotated() {  
 }  
 @Around("methodAnnotated()")//在连接点进行方法替换  
 public void aroundJoinPoint(ProceedingJoinPoint joinPoint) {  
 View view = null;  
 for (Object arg : joinPoint.getArgs()) {  
 if (arg instanceof View) view = (View) arg;  
 }  
 if (view != null) {  
 //获取当前view的tag中保存的上一次点击时间戳  
 Object tag = view.getTag(*TIME\_TAG*);  
 long lastClickTime;  
 if (tag != null) {  
 lastClickTime = (long) tag;  
 } else {  
 lastClickTime = 0L;  
 }  
 if (BuildConfig.*DEBUG*) Log.*d*(*TAG*, "lastClickTime:" + lastClickTime);  
 //获取当前点击时间戳  
 long currentTime = Calendar.*getInstance*().getTimeInMillis();  
 //比较两次点击的间隔时间，大于800毫秒才处理  
 if (currentTime - lastClickTime > *MIN\_CLICK\_DELAY\_TIME*) {  
 //把当前时间保存进view的tag中  
 view.setTag(*TIME\_TAG*, currentTime);  
 if (BuildConfig.*DEBUG*) Log.*d*(*TAG*, "currentTime:" + currentTime);  
 //执行原方法  
 try {  
 joinPoint.proceed();  
 } catch (Throwable throwable) {  
 throwable.printStackTrace();  
 }  
 }  
 }  
 }  
}

### 使用

//@SingleClick //我们的PointCut规则是精准定位到了方法名上，如果规则是*POINT\_CUT\_REG2，那么用注解即可*  
override fun onSingleClick(v: View) {

when (v.*id*) {  
 R.id.*iv\_head* -> ToastUtil.shortShow("头像")  
 R.id.*ll\_msg* -> MessageActivity.start(v.*context*)  
 R.id.*ll\_order\_list* -> OrderListActivity.start(v.*context*)   
 }  
}

## 注意：lambda表达式无法拦截

lambda似乎并没有实际实现/覆盖任何具有相应名称的接口方法,而是创建一个匿名方法.因此没有被aspectj拦截到。因此为了解决这个问题，方案有三种

### 方案一，不要用lambda表达式

不使用lambda，并且在方法上加防颤注解，避免不小心lambda

mAdapter.setOnItemClickListener(new BaseQuickAdapter.OnItemClickListener() {  
 @SingleClick  
 @Override  
 public void onItemClick(BaseQuickAdapter adapter, View view, int position) {  
 if (mAdapter.getData().get(position).getPortrait() != null) {  
 String name = mAdapter.getData().get(position).getName();  
 String id = mAdapter.getData().get(position).getUserId();  
 Logger.*d*("asdad" + id);  
 mAdapter.getData().remove(position);  
 SPUtils.*getInstance*().put(AppConstants.*SP\_SAVE\_CHAT\_FIVE*, new Gson().toJson(mAdapter.getData()));  
 finish();  
 RongIM.*getInstance*().startConversation(ConversationActivity.this, Conversation.ConversationType.*PRIVATE*, id, name);  
  
 }  
 }  
});

### 方案二,mvvm中，没有lambda

<TextView  
 android:id="@+id/tv\_submit"  
 style="@style/com\_btn\_314\_44\_orange"  
 android:gravity="center"  
 android:onClick="@{(v)->vm.onSingleClick(v)}"  
 android:text="@{vm.btnName}"  
 tools:text="提交补差" />

override fun onSingleClick(v: View) {

when (v.*id*) {  
 R.id.*iv\_head* -> ToastUtil.shortShow("头像")  
 R.id.*ll\_msg* -> MessageActivity.start(v.*context*)  
 R.id.*ll\_order\_list* -> OrderListActivity.start(v.*context*)   
 }  
}

### 方案三，复写要响应onclick的控件

就算click被lambda了，内部方法中buttonClick方法仍然存在

@Pointcut("execution(public void \*..Button.ClickListener.buttonClick(\*..Button.ClickEvent))") //方法切入点  
 public void methodAnnotated() {  
}

public class ClickButton extends android.support.v7.widget.AppCompatButton {  
 private ClickListener listener;  
 public void addClickListener(ClickListener listener) {  
 this.listener = listener;  
 }  
 public void click() {  
 System.*out*.println("Clicking button");  
 listener.buttonClick(new ClickEvent());  
 }  
 public static class ClickEvent {}  
 public static interface ClickListener {  
 public void buttonClick(ClickEvent clickEvent);  
 }  
}

# Volley的使用

## 在应用里配置，全局化它

**public** **class** **MyApplication** **extends** **Application**{

**public** **static** **RequestQueue** **queue**;

@Override

**public** **void** **onCreate**() {

**super**.onCreate();

**queue** = **Volley**.newRequestQueue(getApplicationContext());

}

**public** **static** **RequestQueue** **getHttpQueues**(){

**return** **queue**;

}

}

## StringRequest，JsonObj,JsonArray的Get请求方法

**private** **void** **volley\_Get\_String**() {

**String** url = "http://apis.juhe.cn/mobile/get?phone=15102391937&key=dbc298515c53f7c16f0a4ad0b1bf9fd5";

/\*\*四个参数，分别是：请求方法，url，成功回调，失败回调\*/

**StringRequest** request = **new** StringRequest(Method.**GET**, url, **new** Listener<**String**>() {

@Override

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

**Toast**.makeText(mContext, response, **Toast**.**LENGTH\_LONG**).show();

}

}, **new** ErrorListener() {

@Override

**public** **void** **onErrorResponse**(**VolleyError** error) {

**Toast**.makeText(mContext, error.toString(), **Toast**.**LENGTH\_LONG**).show();

}

});

request.setTag("abcGet");**//设置标签**

**MyApplication**.getHttpQueues().add(request);**//将这个StringRequest加入volley队列**

}

## StringRequest的Post请求方法，重点部分：

**。。。。。**

**public** **void** **onErrorResponse**(**VolleyError** error) {

**Toast**.makeText(mContext, error.toString(), **Toast**.**LENGTH\_LONG**).show();

}

}){

**//重写StringRequest类里的getParams()方法，放入Post请求参数**

@Override

**protected** Map<**String**, **String**> **getParams**() **throws** AuthFailureError {

Map<**String**, **String**> hasMap = **new** HashMap<**String**, **String**>();

hasMap.put("phone", "15102391937");

hasMap.put("key", "dbc298515c53f7c16f0a4ad0b1bf9fd5");

**return** hasMap;

}

};

request.setTag("abcPost");**//设置标签**

**MyApplication**.getHttpQueues().add(request);**//将这个StringRequest加入volley队列**

## JsonObj的Post请求方法，重点部分：

Map<**String**, **String**> hasMap = **new** HashMap<**String**, **String**>();

hasMap.put("phone", "15102391937");

hasMap.put("key", "dbc298515c53f7c16f0a4ad0b1bf9fd5");

**JSONObject** object = **new** JSONObject(hasMap);

/\*\*五个参数，分别是：请求方法，url，请求参数（jsonObj类型，post方法为object），成功回调，失败回调\*/

**JsonObjectRequest** request = **new** JsonObjectRequest(Method.**GET**, url, object, **new** Listener<**JSONObject**>() {

## 与Activity生命周期联动

@Override

**protected** **void** **onStop**() {

**super**.onStop();

**MyApplication**.getHttpQueues().cancelAll("abcGet");

**MyApplication**.getHttpQueues().cancelAll("abjGet");

**MyApplication**.getHttpQueues().cancelAll("abcPost");

**MyApplication**.getHttpQueues().cancelAll("abjPost");

}

## 二次封装

/\*\*

\* volley\_Post\_OK

\* \*/

**private** **void** **volley\_Post\_OK**() {

**String** url = "http://apis.juhe.cn/mobile/get?phone=15102391937&key=dbc298515c53f7c16f0a4ad0b1bf9fd5";

Map<**String**, **String**> hasMap = **new** HashMap<**String**, **String**>();

hasMap.put("phone", "15102391937");

hasMap.put("key", "dbc298515c53f7c16f0a4ad0b1bf9fd5");

**VolleyRequest**.RequestPost(mContext, url, "abopost", hasMap, **new** VolleyInterface(mContext, **VolleyInterface**.**listener**, **VolleyInterface**.**errorListener**) {

@Override

**public** **void** **onMySuccess**(**String** result) {

tView.setText(result);

}

@Override

**public** **void** **onMyError**(**VolleyError** error) {

**Toast**.makeText(mContext, error.toString(), **Toast**.**LENGTH\_LONG**).show();

}

});

}

## 用ImageRequest请求网络图片

第三第四个参数分别用于指定允许图片最大的宽度和高度，如果指定的网络图片的宽度或高度大于这里的最大值，则会对图片进行压缩，指定成0的话就表示不管图片有多大，都不会进行压缩。第五个参数用于指定图片的颜色属性，Bitmap.Config下的几个常量都可以在这里使用，其中ARGB\_8888可以展示最好的颜色属性，每个图片像素占据4个字节的大小，而RGB\_565则表示每个图片像素占据2个字节大小

**private** **void** **getImg**(){

**String** url = "http://img5.imgtn.bdimg.com/it/u=161888459,1712714238&fm=21&gp=0.jpg";

**ImageRequest** request = **new** ImageRequest(url, **new** Listener<**Bitmap**>() {

@Override

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

img.setImageBitmap(response);

}

}, 0, 0, Config.**RGB\_565**, **new** ErrorListener() {

@Override

**public** **void** **onErrorResponse**(**VolleyError** error) {

img.setBackgroundResource(**R**.**drawable**.**imgerror**);

}

});

request.setTag("abcImg");**//设置标签**

**MyApplication**.*getHttpQueues*().add(request);**//将这个ImageRequest加入volley队列**

}

## 用ImageLoader请求带缓存的网络图片

**private** **void** **getImgCache**(){

**String** url = "http://img5.imgtn.bdimg.com/it/u=161888459,1712714238&fm=21&gp=0.jpg";

**ImageLoader** loader = **new** ImageLoader(**MyApplication**.*getHttpQueues*(), **new** BitmapCache());

ImageListener listener = **ImageLoader**.*getImageListener*(img, **R**.**drawable**.**ic\_launcher**, **R**.**drawable**.**imgerror**);

loader.get(url, listener);

**//loader.get(url, listener, 200, 200);//控制大小**

}

**public** **class** **BitmapCache** **implements** ImageCache {

**public** **LruCache**<**String**, **Bitmap**> cache;

**public** **int** max = 10 \* 1024 \* 1024;

**public** **BitmapCache**() {

cache = **new** LruCache<**String**, **Bitmap**>(max){

@Override

**protected** **int** **sizeOf**(**String** key, **Bitmap** value) {

**return** value.getRowBytes() \* value.getHeight();

}

};

}

@Override

**public** **Bitmap** **getBitmap**(**String** url) {

**return** cache.get(url);

}

@Override

**public** **void** **putBitmap**(**String** url, **Bitmap** bitmap) {

cache.put(url, bitmap);

}

}

## 用NetworkImageView控件显示网络图片

netImageView = (**NetworkImageView**) findViewById(**R**.**id**.**img\_network**);

**private** **void** **getImgNetwork**(){

**String** url = "http://img5.imgtn.bdimg.com/it/u=161888459,1712714238&fm=21&gp=0.jpg";

**ImageLoader** loader = **new** ImageLoader(**MyApplication**.*getHttpQueues*(), **new** BitmapCache());

netImageView.setDefaultImageResId(**R**.**drawable**.**ic\_launcher**);

netImageView.setErrorImageResId(**R**.**drawable**.**imgerror**);

netImageView.setImageUrl(url, loader);

}

<com.android.volley.toolbox.NetworkImageView

android:id=*"@+id/img\_network"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:layout\_centerHorizontal=*"true"*

android:src=*"@drawable/ic\_launcher"* />

使用ImageRequest和ImageLoader这两种方式来加载网络图片，都可以传入一个最大宽度和高度的参数来对图片进行压缩，而NetworkImageView中则完全没有提供设置最大宽度和高度的方法，那么是不是使用NetworkImageView来加载的图片都不会进行压缩呢？

其实并不是这样的，NetworkImageView并不需要提供任何设置最大宽高的方法也能够对加载的图片进行压缩。这是由于NetworkImageView是一个控件，在加载图片的时候它会自动获取自身的宽高，然后对比网络图片的宽度，再决定是否需要对图片进行压缩。也就是说，压缩过程是在内部完全自动化的，并不需要我们关心，NetworkImageView会始终呈现给我们一张大小刚刚好的网络图片，不会多占用任何一点内存，这也是NetworkImageView最简单好用的一点吧。

当然了，如果你不想对图片进行压缩的话，其实也很简单，只需要在布局文件中把NetworkImageView的layout\_width和layout\_height都设置成wrap\_content就可以了，这样NetworkImageView就会将该图片的原始大小展示出来，不会进行任何压缩。

## GsonVolley构造

**public** **class** **GsonRequest**<T> **extends** **Request**<T> {

**private** **Gson** mGson = **new** **Gson**();

**private** **Class**<T> clazz;

**private** **Map**<**String**, **String**> headers;

**private** **Map**<**String**, **String**> params;

**private** **Listener**<T> listener;

**public** **GsonRequest**(**String** url,

**Class**<T> clazz,

**Listener**<T> listener,

**ErrorListener** errorListener) {

**this**(Method.GET, url, clazz, **null**, listener, errorListener);

}

**public** **GsonRequest**(**int** method,

**String** url,

**Class**<T> clazz,

**Map**<**String**, **String**> params,

**Listener**<T> listener,

**ErrorListener** errorListener) {

**super**(method, url, errorListener);

**this**.clazz = clazz;

**this**.params = params;

**this**.listener = listener;

**this**.headers = **null**;

mGson = **new** **Gson**();

}

@Override

**public** **Map**<**String**, **String**> **getHeaders**() **throws** AuthFailureError {

**return** headers != **null** ? headers : **super**.getHeaders();

}

@Override

**protected** **Map**<**String**, **String**> **getParams**() **throws** AuthFailureError {

**return** params;

}

@Override

**protected** **void** **deliverResponse**(T response) {

listener.onResponse(response);

}

@Override

**protected** **Response**<T> **parseNetworkResponse**(**NetworkResponse** response) {

**String** json = **null**;

**try** {

**//如果在服务器的返回数据的header中没有指定字符集那么就会默认使用 ISO-8859-1 字符集,所以自己指定utf-8**

**if** (response.headers.containsKey("charset")) {**//有返回的字符集**

json = **new** String(

response.data, HttpHeaderParser.parseCharset(response.headers));

} **else** {

json = **new** String(response.data, "utf-8");

}

**return** Response.success(mGson.fromJson(json, clazz), HttpHeaderParser.parseCacheHeaders(response));

} **catch** (**UnsupportedEncodingException** e) {

**return** Response.error(**new** ParseError(e));

} **catch** (**JsonSyntaxException** e) {

**return** Response.error(**new** ParseError(e));

}

}

}

**public** **class** **WeatherModeImpl** **implements** **WeatherModel** {

@Override

**public** **void** **getWeather**(**String** countryNum, **final** **OnWeatherListener** listener) {

**//GsonRequest(String url, Class<T> clazz, Listener<T> listener, ErrorListener errorListener) {}**

**GsonRequest** request = **new** GsonRequest("http://www.weather.com.cn/data/sk/" + countryNum + ".html", **Weather**.**class**, **new** **Response**.**Listener**<**Weather**>() {

@Override

**public** **void** **onResponse**(Weather weather) {

listener.onSuccess(weather);

}

}, **new** **Response**.**ErrorListener**() {

@Override

**public** **void** **onErrorResponse**(**VolleyError** volleyError) {

listener.OnError(volleyError);

}

});

MyApplication.getQueue().add(request);

}

}

# 表格框架[SortableTableView](https://github.com/ISchwarz23/SortableTableView)

## 引用

dependencies {

...

compile 'de.codecrafters.tableview:tableview:2.6.0'

...

}

## 初始化控件

### Layout中

<de.codecrafters.tableview.TableView

xmlns:table="http://schemas.android.com/apk/res-auto"

android:id="@+id/tableView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

table:tableView\_columnCount="4" />

### 代码中

TableView tableView = (TableView) findViewById(R.id.tableView);

tableView.setColumnCount(4);

## 表格宽

**Weight**

TableColumnWeightModel columnModel = new TableColumnWeightModel(4);

columnModel.setColumnWeight(0, 2);

columnModel.setColumnWeight(1, 2);

tableView.setColumnModel(columnModel);

**DpWidth**

TableColumnDpWidthModel columnModel = new TableColumnDpWidthModel(context, 4, 200);

columnModel.setColumnWidth(2, 300);

columnModel.setColumnWidth(3, 250);

tableView.setColumnModel(columnModel);

**PxWidth**

TableColumnPxWidthModel columnModel = new TableColumnPxWidthModel(4, 350);

columnModel.setColumnWidth(1, 500);

columnModel.setColumnWidth(3, 600);

tableView.setColumnModel(columnModel);

## 显示数据

**Simple Data**

public class MainActivity extends AppCompatActivity {

private static final String[][] DATA\_TO\_SHOW = { { "This", "is", "a", "test" },

{ "and", "a", "second", "test" } };

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

TableView<String[]> tableView = (TableView<String[]>) findViewById(R.id.tableView);

tableView.setDataAdapter(new SimpleTableDataAdapter(this, DATA\_TO\_SHOW));

}

}

**Custom Data**

// getRowData()

// getContext()

// getLayoutInflater()

// getResources()

public class CarTableDataAdapter extends TableDataAdapter<Car> {

public CarTableDataAdapter(Context context, List<Car> data) {

super(context, data);

}

@Override

public View getCellView(int rowIndex, int columnIndex, ViewGroup parentView) {

Car car = getRowData(rowIndex);

View renderedView = null;

switch (columnIndex) {

case 0:

renderedView = renderProducerLogo(car);

break;

case 1:

renderedView = renderCatName(car);

break;

case 2:

renderedView = renderPower(car);

break;

case 3:

renderedView = renderPrice(car);

break;

}

return renderedView;

}

...

}

## 排序

需要排序时，使用SortableTableView代替TableView

public static class priceComparator implements Comparator<SortTableDemoBean> {

@Override

public int compare(SortTableDemoBean o1, SortTableDemoBean o2) {

return o1.getPrice() - o2.getPrice();

}

}

public static class nameComparator implements Comparator<SortTableDemoBean> {

@Override

public int compare(SortTableDemoBean o1, SortTableDemoBean o2) {

return o1.getName().compareTo(o2.getName());

}

}

tableView.setColumnComparator(0, new SortTableDemoBean.nameComparator());

tableView.setColumnComparator(2, new SortTableDemoBean.priceComparator());

## 空数据提示（未实现）

TextView emptyTv = new TextView(this);

emptyTv.setText("我是空数据Indicator");

emptyTv.setTextColor(0x000000);

tableView.setEmptyDataIndicatorView(emptyTv);

## 表头

将数据设置为头视图，与将数据设置到表单元相同。所需要做的就是扩展TableHeaderAdapter，它还提供了用于TableDataAdapter的简单访问方法。如果想在标题中显示的是列标题为String(在大多数情况下)，SimpleTableHeaderAdapter将满足需求。

tableView.setHeaderAdapter(new SimpleTableHeaderAdapter(this, "名字", "图标", "价格", "标签"));

## 单击监听

### Data Click Listening

tableView.addDataClickListener(new TableDataClickListener<Car>() {

@Override

public void onDataClicked(int rowIndex, Car clickedCar) {

String clickedCarString = clickedCar.getProducer().getName() + " " + clickedCar.getName();

Toast.makeText(getContext(), clickedCarString, Toast.LENGTH\_SHORT).show();

}

});

### Long Data Click Listening

tableView.addDataLongClickListener(new TableDataLongClickListener<Car>() {

@Override

public boolean onDataLongClicked(int rowIndex, Car clickedCar) {

String clickedCarString = clickedCar.getProducer().getName() + " " + clickedCar.getName();

Toast.makeText(getContext(), clickedCarString, Toast.LENGTH\_SHORT).show();

return true;

}

});

### **Header Click Listening**

tableView.addHeaderClickListener(new TableHeaderClickListener() {

@Override

public void onHeaderClicked(int columnIndex) {

String notifyText = "clicked column " + (columnIndex+1);

Toast.makeText(getContext(), notifyText, Toast.LENGTH\_SHORT).show();

}

});

## 样式Style

### Header Styling

<de.codecrafters.tableview.TableView

xmlns:table="http://schemas.android.com/apk/res-auto"

android:id="@+id/tableView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

table:tableView\_headerColor="@color/primary"

table:tableView\_headerElevation="10"/>

Or

tableView.setHeaderBackgroundColor(getResources().getColor(R.color.primary));

tableView.setHeaderElevation(10);

### 排序图标

tableView.setHeaderSortStateViewProvider(new SortStateViewProvider() {

private static final int NO\_IMAGE\_RES = -1;

@Override

public int getSortStateViewResource(SortState state) {

switch (state) {

case SORTABLE:

return R.mipmap.icon\_num0;

case SORTED\_ASC:

return R.mipmap.icon\_num1;

case SORTED\_DESC:

return R.mipmap.icon\_num2;

default:

return NO\_IMAGE\_RES;

}

}

});

### 奇偶行背景

int colorEvenRows = getResources().getColor(R.color.lightYellow);

int colorOddRows = getResources().getColor(R.color.blue\_light);

tableView.setDataRowBackgroundProvider(TableDataRowBackgroundProviders.alternatingRowColors(colorEvenRows, colorOddRows));

### 分隔线

<de.codecrafters.tableview.TableView

android:id="@+id/tableView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

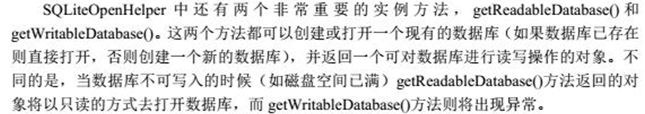
android:divider="@color/black"

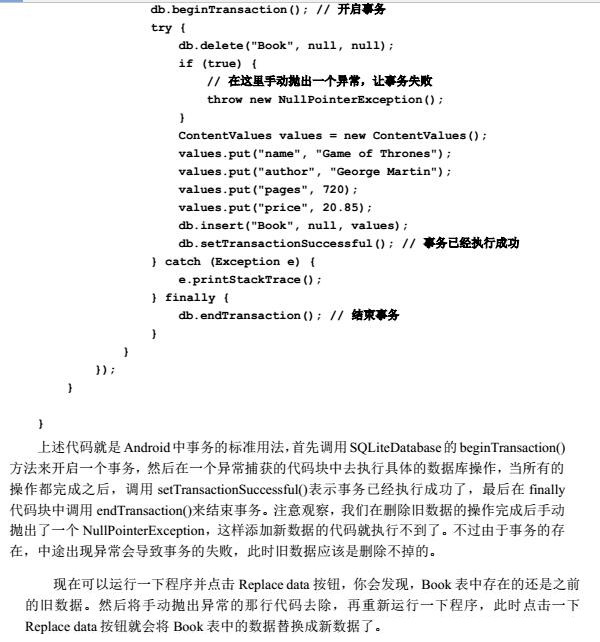
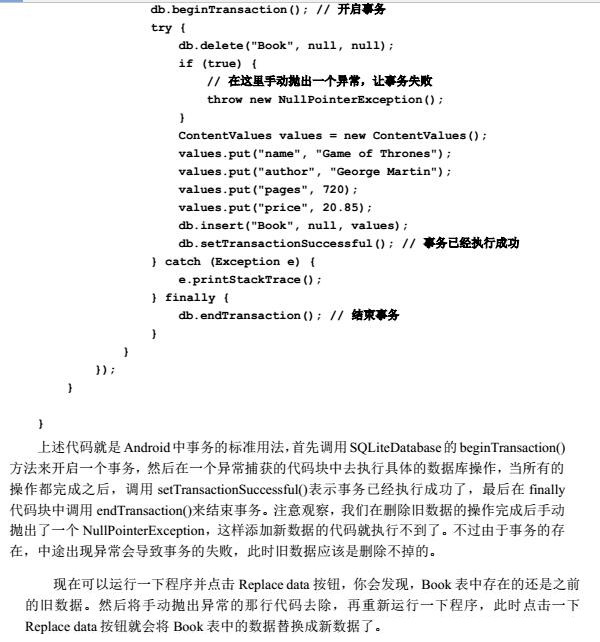
android:dividerHeight="1dip"

... />

# 数据库的使用

## 存放目录和开启事务





## 来个代码

**public** **class** **DBHelper** **extends** **SQLiteOpenHelper** {

**private** **Context** mContext;

**public** **final** **static** **String** **DB\_NAME** = "phones.db";

**private** **final** **static** **String** **TABLE\_NAME\_PHONE** = "phone";

**private** **final** **static** **String** **TABLE\_NAME\_MESSAGE** = "message";

**private** **final** **static** **String** **CREATE\_TBL\_PHONE** = "create table phone(" +

"\_id integer primary key autoincrement, " +

"name text, " +

"sex text, " +

"number text, " +

"desc text)";

**private** **final** **static** **String** **CREATE\_TBL\_MESSAGE** = "create table message(" +

"\_id integer primary key autoincrement, " +

"msg text, " +

"number text)";

**private** **SQLiteDatabase** db;

**public** **DBHelper**(**Context** context, **String** name, CursorFactory factory,**int** version) {

**super**(context, name, factory, version);

mContext = context;

}

**public** **DBHelper**(**Context** context, **int** version){

**this**(context, **DB\_NAME**, **null**, version);

}

// getWritableDatabase和getReadableDatabase时，发现没有这个表时调用

@Override

**public** **void** **onCreate**(**SQLiteDatabase** db) {

db.execSQL(**CREATE\_TBL\_PHONE**);

db.execSQL(**CREATE\_TBL\_MESSAGE**);

}

//初始化时，传入版本号比原有的大时调用

@Override

**public** **void** **onUpgrade**(**SQLiteDatabase** db, **int** oldVersion, **int** newVersion) {

db.execSQL("DROP TABLE IF EXISTS" + **TABLE\_NAME\_PHONE** );

db.execSQL("DROP TABLE IF EXISTS" + **TABLE\_NAME\_MESSAGE** );

onCreate(db);

}

//关闭数据库

**public** **void** **close**(){

db.close();

}

//打开数据库

**public** **void** **open**(){

db = getWritableDatabase();

}

//插入方法----(String table, String nullColumnHack, ContentValues values)

**public** **void** **insert**(**String** nameStr, **String** sexStr, **String** numberStr, **String** descStr){

open();

**ContentValues** values = **new** ContentValues();

values.put("name", nameStr);

values.put("sex", sexStr);

values.put("number", numberStr);

values.put("desc", descStr);

db.insert(**TABLE\_NAME\_PHONE**, **null**, values);

close();

}

//根据唯一标识\_id,来删除数据----(String table, String whereClause, String[] whereArgs)

**public** **void** **delete**(**int** id){

open();

db.delete(**TABLE\_NAME\_PHONE**, "\_id=?", **new** **String**[]{**String**.*valueOf*(id)});

close();

}

//清空表中数据

**public** **void** **deleteAll**(){

open();

db.delete(**TABLE\_NAME\_PHONE**, **null**, **null**);

close();

}

//更新(修改)数据库的内容----(String table, ContentValues values, String whereClause, String[] whereArgs)

**public** **void** **update**(**ContentValues** values, **String** whereClause, **String**[]whereArgs){

open();

db.update(**TABLE\_NAME\_PHONE**, values, whereClause, whereArgs);

close();

}

//查询全部(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)

**public** Cursor **query**(){

db = getReadableDatabase();

Cursor c = db.query(**TABLE\_NAME\_PHONE**, **null**, **null**, **null**, **null**, **null**, **null**);

**return** c;

}

}

## 查询方法详解

**public** Cursor **query**(**String** table,**String**[] columns,**String** selection,**String**[] selectionArgs,**String** groupBy,**String** having,**String** orderBy);

**参数说明：**

table:数据库表的名称

columns:数据库列名称数组 写入后最后返回的Cursor中只能查到这里的列的内容

selection:查询条件

selectionArgs:查询条件参数

groupBy:分组列

having:分组条件

orderBy:排序列

**Cursor的方法：**

Cursor是一个游标接口，每次查询的结果都会保存在Cursor中 可以通过遍历Cursor的方法拿到当前查询到的所有信息。

moveToFirst() //将Curor的游标移动到第一条

moveToNext()//将Curor的游标移动到下一条

getColumnIndex(int columnIndex)//根据列名称获得列索引ID

getString(int columnIndex)//根据索引ID 拿到表中存的String

getInt(int columnIndex)//根据索引ID 拿到表中存的int

moveToLast()///将Curor的游标移动到最后一条

move(int offset)//将Curor的游标移动到指定ID

moveToPrevious()//将Curor的游标移动到上一条

getCount() //得到Cursor 总记录条数

isFirst() //判断当前游标是否为第一条记录

isLast()//判断当前游标是否为最后一条数据

**例子一：（根据用户id查签到天数和日期）**

public Cursor queryById(String userid) {

Cursor cursor = db.query(DATABASE\_TABLE\_TWO,

new String[] {KEY\_SIGNDAY, KEY\_DATE },

KEY\_USERID + "=?",

new String[] { userid },

null, null, null);

return cursor;

}

**例子二：（查询全部）**

public Cursor query(){

SQLiteDatabase db = getReadableDatabase();

Cursor c = db.query(TABLE\_NAME, null, null, null, null, null, null, null);

return c;

}

dbHelper = **new** DBHelper(**this**, 1);

Cursor c = dbHelper.query();

**//if** (c.moveToFirst()) { 不能这样，因为moveTofirst会跳到第一条数据之后，这样会不能得到第一条数据

**while** (c.moveToNext()){

**int** \_id = c.getInt(c.getColumnIndex("\_id"));

**String** name = c.getString(c.getColumnIndex("name"));

**String** sex = c.getString(c.getColumnIndex("sex"))

**String** number = c.getString(c.getColumnIndex("number"));

**String** desc = c.getString(c.getColumnIndex("desc"));

}

//}

# 数据库框架greenDAO3.0

用过ActiveAndroid、玩过ORMLite，穿过千山万水，最终还是发现greenDAO好用，ActiveAndroid我之前有一篇文章介绍过 玩转Android之数据库框架ActiveAndroid的使用，如果小伙伴们有兴趣可以去查看,ActiveAndroid目前存在的问题是没有更新了，GitHub上ActiveAndroid的更新都是两年前的事了。所以我们还是来看看这个目前非常活跃的数据库框架greenDAO，greenDAO在升级到3.0之后使用方式也变得更加简单了，上手更加容易了

## 引入greenDAO

用过3.0之前的greenDAO的小伙伴们应该都知道在引入greenDAO的时候是非常麻烦，但是在3.0之后这个引入过程简化了很多，参考官网给出的指导，我们来看看如何引入greenDAO：

* 首先在project的gradle文件中引入greenDAO插件，引入之后如下：

dependencies {

classpath 'com.android.tools.build:gradle:2.1.0'

classpath 'org.greenrobot:greendao-gradle-plugin:3.1.0'

// NOTE: Do not place your application dependencies here; they belong

// in the individual module build.gradle files

}

* 然后在module的gradle文件中添加greenDAO的插件，并引入相关类库，修改之后如下：

apply plugin: 'com.android.application'

apply plugin: 'org.greenrobot.greendao'

android {

...

...

greendao{

schemaVersion 1

targetGenDir 'src/main/java'

}

}

dependencies {

...

...

compile 'org.greenrobot:greendao:3.1.0'

}

* 小伙伴们注意首先是引入相关插件，然后在dependencies中引入greenDAO的类库。至于greenDAO中，schemaVersion表示数据库版本号，每次数据库升级的时候我们修改这里的版本号即可（修改这里的版本号，greenDAO会自动修改生成到DAOMaster中的版本号），targetGenDir表示greenDAO生成的DAOMaster和DaoSession的位置。OK，做完这一切我们就已经成功将greenDAO引入到我们的项目中了。

## 数据库初始化

* 数据库引入成功后，在使用之前，我们还得先来创建一个实体类：

@Entity

public class User {

@Id

private Long id;

@Property(nameInDb = "USERNAME")

private String username;

@Property(nameInDb = "NICKNAME")

private String nickname;

}

@Entity表示这个实体类会在数据库中生成对应的表，@Id表示该字段是id，小伙伴们注意该字段的数据类型为包装类型Long，为什么要用包装类型呢？我们一会插入数据的时候再说。@Property则表示该属性将作为表的一个字段，其中nameInDb看名字就知道这个属性在数据库中对应的数据名称。OK，写完这些之后将项目进行编译，编译成功之后系统会帮助我们生成相应的构造方法和get/set方法，并且还会在我们的包下生成DaoMaster和DaoSession。那么这里常用的注解除了这几个之外，还有一个较常用的就是@Transient，该注解表示这个属性将不会作为数据表中的一个字段。就是这么简单。另外还有一些比如@NotNull表示该字段不可以为空，@Unique表示该字段唯一。这里的注解还是挺多的，小伙伴们有兴趣可以自行研究。

* OK，这一步做完之后，我们就可以进行数据库的初始化了，如下：

DaoMaster.DevOpenHelper devOpenHelper = new DaoMaster.DevOpenHelper(getApplicationContext(), "lenve.db", null);

DaoMaster daoMaster = new DaoMaster(devOpenHelper.getWritableDb());

DaoSession daoSession = daoMaster.newSession();

首先获取一个DevOpenHelper对象，这个类有点类似于我们使用的SqliteOpenHelper，我们主要在这个类中对数据库的版本进行管理。这样之后，我们对数据库基本的初始化操作就完成了想要操作User实体类，得先有一个UserDao，这个UserDao要怎么获取呢？如下：

userDao = daoSession.getUserDao();

## 添加数据

数据库的增删改查我们都将通过UserDao来进行，插入操作如下：

User user = new User(null, "zhangsan" + random.nextInt(9999),"张三");

userDao.insert(user);

User的第一个参数为id，这里传null的话在插入的过程中id字段会自动增长（现在知道为什么id要为Long类型了吧！）第二个参数是username，每次插入时生成一个随机数，避免重复。

## 查询数据

List<User> list = userDao.queryBuilder().where(UserDao.Properties.Id.between(2, 13)).limit(5).build().list();

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

Log.d("google\_lenve", "search: " + list.get(i).toString());

}

查询里边有许多非常好用的函数，我这里举两个例子between表示查询id介于2到13之间的数据，limit表示查询5条数据。

## 删除数据

删除数据和修改数据的思路一样，都是要先查找到数据：

List<User> userList = (List<User>) userDao.queryBuilder().where(UserDao.Properties.Id.le(10)).build().list();

for (User user : userList) {

userDao.delete(user);

}

where表示查询条件，这里我是查询id小于等于10的数据，where中的参数可以有多个，就是说可以添加多个查询条件。最后的list表示查询结果是一个List集合，如果你只想查询一条数据，最后unique即可。当然，我们也可以根据id来删除数据：

User user = userDao.queryBuilder().where(UserDao.Properties.Id.eq(16)).build().unique();

if (user == null) {

Toast.makeText(MainActivity.this, "用户不存在", Toast.LENGTH\_SHORT).show();

}else{

userDao.deleteByKey(user.getId());

}

也可以将表中所有数据一次删除：

userDao.deleteAll();

## 修改数据

修改数据也是先查询，再修改：

User user = userDao.queryBuilder().where(UserDao.Properties.Id.ge(10), UserDao.Properties.Username.like("%90%")).build().unique();

if (user == null) {

Toast.makeText(MainActivity.this, "用户不存在!", Toast.LENGTH\_SHORT).show();

}else{

user.setUsername("王五");

userDao.update(user);

}

这里我添加了两个查询条件，一个是id要大于等于10，同是还要满足username like %90%，注意最后的unique表示只查询一条数据出来即可。

## 数据库升级

数据库的升级其实就两个步骤我们来看看：

### 修改gradle文件

首先在module的gradle文件中修改版本号：

//这里改为最新的版本号

schemaVersion 2

targetGenDir 'src/main/java'

### 修改实体类

@Entity

public class User {

@Property

private int age;

@Property

private String password;

@Id

private Long id;

@Property(nameInDb = "USERNAME")

private String username;

@Property(nameInDb = "NICKNAME")

private String nickname;

}

重现编译项目运行即可。一般的数据库升级这样就可以了，特殊情况可能需要自己编写数据库迁移脚本，这种时候可以自定义DBHelper，定义方式如下，注意继承类：

public class DBHelper extends DaoMaster.OpenHelper {

public static final String DBNAME = "lenve.db";

public DBHelper(Context context) {

super(context, DBNAME, null);

}

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

super.onUpgrade(db, oldVersion, newVersion);

}

}

可以在onUpgrade方法中进行数据库的迁移，如果自定义了DBHelper，则数据库的初始化变为如下方式：

DBHelper devOpenHelper = new DBHelper(this);

DaoMaster daoMaster = new DaoMaster(devOpenHelper.getWritableDb());

DaoSession daoSession = daoMaster.newSession();

userDao = daoSession.getUserDao();

# Realm详解

## Realm介绍

### 介绍

Realm 是一个 MVCC （多版本并发控制）数据库，由Y Combinator公司在2014年7月发布一款支持运行在手机、平板和可穿戴设备上的嵌入式数据库，目标是取代SQLite。

Realm 本质上是一个嵌入式数据库，他并不是基于SQLite所构建的。它拥有自己的数据库存储引擎，可以高效且快速地完成数据库的构建操作。和SQLite不同，它允许你在持久层直接和数据对象工作。在它之上是一个函数式风格的查询api，众多的努力让它比传统的SQLite 操作更快。

### 优势

* 易用

Ream 不是在SQLite基础上的ORM，它有自己的数据查询引擎。并且十分容易使用。

* 快速

由于它是完全重新开始开发的数据库实现，所以它比任何的ORM速度都快很多，甚至比SLite速度都要快。

* 跨平台

Realm 支持 iOS & OS X (Objective‑C & Swift) & Android。我们可以在这些平台上共享Realm数据库文件，并且上层逻辑可以不用任何改动的情况下实现移植。

* 高级

Ream支持加密，格式化查询，易于移植，支持JSON，流式api，数据变更通知等高级特性

* 可视化

Realm 还提供了一个轻量级的数据库查看工具，在Mac Appstore 可以下载“Realm Browser”这个工具，开发者可以查看数据库当中的内容，执行简单的插入和删除数据的操作。（windows上还不清楚）

### 条件

* 目前不支持Android以外的Java
* Android Studio >= 1.5.1
* 较新的Android SDK版本
* JDK version >=7.
* 支持API 9(Android 2.3)以及之后的版本

## 使用

### 加入依赖

在project的build中加入依赖

buildscript {

repositories {

jcenter()

}

dependencies {

classpath "io.realm:realm-gradle-plugin:2.2.1"

}

}

在module中加入

apply plugin: 'realm-android'

### 创建model

创建一个User类，需要继承RealmObject。支持public, protected和 private的类以及方法

public class User extends RealmObject {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

除了直接继承于RealmObject来声明 Realm 数据模型之外，还可以通过实现 RealmModel接口并添加 @RealmClass修饰符来声明。

@RealmClass

public class User implements RealmModel {

...

}

* 支持的属性

boolean, byte, short,int,long,float, double,String, Date 和，byte[], RealmObject, RealmList<? extends RealmObject>，还支持Boolean, Byte, Short, Integer, Long, Float 和 Double

Tip：整数类型 short、int 和 long 都被映射到 Realm 内的相同类型（实际上为 long ）

* @PrimaryKey——表示该字段是主键

使用过数据库的同学应该看出来了，PrimaryKey就是主键。使用@PrimaryKey来标注，字段类型必须是字符串（String）或整数（byte，short，int或long）以及它们的包装类型（Byte,Short, Integer, 或 Long）。不可以存在多个主键，使用字符串字段作为主键意味着字段被索引（注释@PrimaryKey隐式地设置注释@Index）。

@PrimaryKey

private String id;

* @Required——表示该字段非空

在某些情况下，有一些属性是不能为null的。使用@Required可用于用于强行要求其属性不能为空，只能用于Boolean, Byte, Short, Integer, Long, Float, Double, String, byte[] 和 Date。在其它类型属性上使用 @Required修饰会导致编译失败。

Tip：基本数据类型不需要使用注解 @Required，因为他们本身就不可为空。

@Required

private String name;

* @Ignore——表示忽略该字段

被添加@Ignore标签后，存储数据时会忽略该字段。

@Ignore

private String name;

* @Index——添加搜索索引

为字段添加搜索索引，这样会使得插入的速度变慢，数据量也变得更大。不过在查询速度将变得更快，建议只在优化读取性能的特定情况时添加索引。支持索引：String，byte，short，int，long，boolean和Date字段。

@Index

private String id;

注意：如果你创建Model并运行过，然后修改了Model。那么就需要升级数据库，否则会抛异常。升级方式后面会提到

### 初始化

* 使用默认配置

Realm.init(this);

Realm mRealm = Realm.getDefaultInstance();

这时候会创建一个叫做 default.realm的Realm文件，一般来说，这个文件位于/data/data/包名/files/。通过realm.getPath()来获得该Realm的绝对路径。注意：模拟器上运行时，Realm.getDefaultInstance()抛出异常，真机上没问题（不止何故）

* 当然，我们还可以使用RealmConfiguration来配置Realm

RealmConfiguration config = new RealmConfiguration.Builder()

.name("myrealm.realm") //文件名

.schemaVersion(0) //版本号

.build();

Realm realm = Realm.getInstance(config);

* 创建非持久化的Realm，也就是保持在内存中，应用关闭后就清除了。

RealmConfiguration myConfig = new RealmConfiguration.Builder(context)

.name("myrealm.realm")

.inMemory() //保存在内存中

.build();

* RealmConfiguration支持的方法：
* Builder.name : 指定数据库的名称。如不指定默认名为default。
* Builder.schemaVersion : 指定数据库的版本号。
* Builder.encryptionKey : 指定数据库的密钥。
* Builder.migration : 指定迁移操作的迁移类。
* Builder.deleteRealmIfMigrationNeeded : 声明版本冲突时自动删除原数据库。
* Builder.inMemory : 声明数据库只在内存中持久化。
* build : 完成配置构建。

### 关闭Realm

记得使用完后，在onDestroy中关闭Realm

@Override

protected void onDestroy() {

super.onDestroy();

// Close the Realm instance.

realm.close();

}

### 增

写入操作需要在事务中进行，可以使用executeTransaction方法来开启事务。

* 使用executeTransaction方法插入数据

mRealm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

User user = realm.createObject(User.class);

user.setName("Gavin");

user.setAge(23);

}

});

在execute方法中执行插入操作

注意：如果在UI线程中插入过多的数据，可能会导致主线程拥塞。

* 使用copyToRealmOrUpdate或copyToRealm方法插入数据

当Model中存在主键的时候，推荐使用copyToRealmOrUpdate方法插入数据。如果对象存在，就更新该对象；反之，它会创建一个新的对象。若该Model没有主键，使用copyToRealm方法，否则将抛出异常。

final User user = new User();

user.setName("Jack");

user.setId("2");

mRealm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

realm.copyToRealmOrUpdate(user);

}

});

* 如果User还有其他属性是，比如养了只Dog：

public class Dog extends RealmObject {

private String name;

private int age;

//getters and setters

}

对应的User属性中加入Dog类

public class User extends RealmObject {

private String name;

private int age;

private Dog dog;

//getters and setters

}

如果你养了不止一只，你养了二哈、阿拉撕家...那你就要用到RealmList了。在User中加入该属性

private RealmList<Dog> dogs;

插入数据

@Override

public void execute(Realm realm) {

User user = realm.createObject(User.class);

user.setName("Gain");

user.setAge(23);

Dog dog1 = realm.createObject(Dog.class);

dog1.setAge(1);

dog1.setName("二哈");

user.getDogs().add(dog1);

Dog dog2 = realm.createObject(Dog.class);

dog2.setAge(2);

dog2.setName("阿拉撕家");

user.getDogs().add(dog2);

}

});

* 上面都是用可executeTransaction方法插入数据，还有另一种方法beginTransaction和commitTransaction

mRealm.beginTransaction();//开启事务

User user = mRealm.createObject(User.class);

user.setName("Gavin");

user.setId("3");

mRealm.commitTransaction();//提交事务

在插入前，先调用beginTransaction()，完成后调用commitTransaction()即可。

注意：在UI和后台线程同时开启创建write的事务，可能会导致ANR错误。为了避免该问题，可以使用executeTransactionAsync来实现。

* 使用executeTransactionAsync

该方法会开启一个子线程来执行事务，并且在执行完成后进行结果通知。

RealmAsyncTask transaction = mRealm.executeTransactionAsync(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

User user = realm.createObject(User.class);

user.setName("Eric");

user.setId("4");

}

});

还可以加入监听

RealmAsyncTask transaction = mRealm.executeTransactionAsync(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

User user = realm.createObject(User.class);

user.setName("Eric");

user.setId("4");

}

}, new Realm.Transaction.OnSuccess() {

@Override

public void onSuccess() {

//成功回调

}

}, new Realm.Transaction.OnError() {

@Override

public void onError(Throwable error) {

//失败回调

}

});

注意：如果当Acitivity或Fragment被销毁时，在OnSuccess或OnError中执行UI操作，将导致程序奔溃 。用RealmAsyncTask .cancel();可以取消事务

在onStop中调用，避免crash

public void onStop () {

if (transaction != null && !transaction.isCancelled()) {

transaction.cancel();

}

}

* JSON

Realm还是个很nice的功能就是将Json字符串转化为对象，厉害了我的Realm (直接借用官方的例子）

// 一个city model

public class City extends RealmObject {

private String city;

private int id;

// getters and setters left out ...

}

// 使用Json字符串插入数据

realm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

realm.createObjectFromJson(City.class, "{ city: \"Copenhagen\", id: 1 }");

}

});

// 使用InputStream插入数据

realm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

try {

InputStream is = new FileInputStream(new File("path\_to\_file"));

realm.createAllFromJson(City.class, is);

} catch (IOException e) {

throw new RuntimeException();

}

}

});

Realm 解析 JSON 时遵循如下规则：

* 使用包含空值（null）的 JSON 创建对象：
* 对于非必须（可为空值的属性），设置其值为 null；
* 对于必须（不可为空值的属性），抛出异常；
* 使用包含空值（null）的 JSON 更新对象：
* 对于非必须（可为空值的属性），设置其值为 null；
* 对于必须（不可为空值的属性），抛出异常；
* 使用不包含对应属性的 JSON： \* 该属性保持不变

### 查

查找操作就比插入方便多了，并不需在事务中操作，直接查询即可。

* findAll ——查询

例：查询所有的User

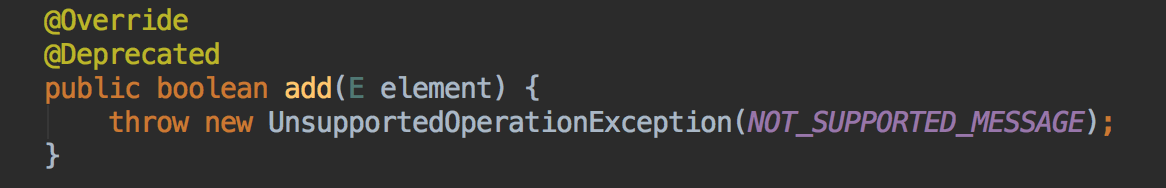
RealmResults<User> userList = mRealm.where(User.class).findAll();

这里使用RealmResults来接受查询到的结果，突然出现的RealmResults可能会让人懵逼。看看他的源码，发现RealmResults继承了AbstractList，而AbstractList又实现了List接口。好吧，原来实现了我们熟悉的List接口。

public final class RealmResults<E extends RealmModel> extends AbstractList<E>

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E>

注意：RealmResults虽然实现了List接口，不过有很多方法是不能用的。比如add、addAll、remove、clear等，调用后会直接抛异常。不过也不用当心误用这些方法，因为它们都被标记为@Deprecated了。



* findAllAsync——异步查询

当数据量较大，可能会引起ANR的时候，就可以使用findAllAsync

RealmResults<User> userList = mRealm.where(User.class)

.equalTo("name", "Gavin")

.findAllAsync();

值得注意的是，这里并不会马上查到数据，是有一定延时的。也就是说，你马上使用userList的时候，里面是没有数据的。可以注册RealmChangeListener监听器，或者使用isLoaded()方法，判断是否查询完成

if (result.isLoaded()) {

// 完成查询

}

* findFirst ——查询第一条数据

例：查询User表中的第一条数据

User user2 = mRealm.where(User.class).findFirst();

* equalTo ——根据条件查询

例：得到name为Gavin的用户列表。

RealmResults<User> userList = mRealm.where(User.class).equalTo("name", "Gavin").findAll();

如果User中还有Dog属性，希望根据Dog的条件来获取用户：

例：查询dogs.name为二哈的User

RealmResults<User> userList = mRealm.where(User.class).equalTo("dogs.name", "二哈").findAll();

* equalTo ——多条件查询

当然，我们还经常要用到多条件的查询的功能。

例：找到用户名为“Gavin”，且dogs.name为“二哈”的User

RealmResults<User> userO = mRealm.where(User.class)

.equalTo("name", "Gavin").findAll();

RealmResults<User> userList = userO.where()

.equalTo("dogs.name", "二哈").findAll();

上面先找到name为“Gavin”的User列表，然后再得到的结果中查询dogs.name为“二哈”,觉得这样写太麻烦？我也是这样想的，所以还可以这样写

RealmResults<User> userList = mRealm.where(User.class)

.equalTo("name", "Gavin")

.equalTo("dogs.name", "二哈")

.findAll();

* 更多查询条件，已知的方法如下：
* sum()：对指定字段求和。
* average()：对指定字段求平均值。
* min(): 对指定字段求最小值。
* max() : 对指定字段求最大值。count : 求结果集的记录数量。
* findAll(): 返回结果集所有字段，返回值为RealmResults队列
* findAllSorted() : 排序返回结果集所有字段，返回值为RealmResults队列
* between(), greaterThan(),lessThan(), greaterThanOrEqualTo() & lessThanOrEqualTo()
* equalTo() & notEqualTo()
* contains(), beginsWith() & endsWith()
* isNull() & isNotNull()
* isEmpty()& isNotEmpty()
* RealmQuery以及or的使用

在使用where()方法时，能得到一个RealmQuery对象，使用方法如下：

例：查询name为“Gavin”和“Eric”的用户

RealmQuery<User> query = mRealm.where(User.class);

query.equalTo("name", "Gavin");

query.or().equalTo("name", "Eric");

RealmResults<User> userList = query.findAll();

这种情况下就要用到or()方法，这么一大串，你又觉得麻烦了？没事，继续简化。

RealmResults<User> userList = mRealm.where(User.class)

.equalTo("name", "Gavin")

.or().equalTo("name", "Eric")

.findAll();

Tip：查询的时候你不用当心得到的RealmResults为null。如果查询的结果为空，那么RealmResults的size为0

* 排序

查询结束后，还可以进行排序，就像这样：

RealmResults<User> userList = mRealm.where(User.class) .findAll();

userList = result.sort("age"); //根据age，正序排列

userList = result.sort("age", Sort.DESCENDING);//逆序排列

* 聚合

RealmResult自带一些聚合方法：

RealmResults<User> results = realm.where(User.class).findAll();

long sum = results.sum("age").longValue();

long min = results.min("age").longValue();

long max = results.max("age").longValue();

double average = results.average("age");

long matches = results.size();

### 改

修改也是需要在事务中操作。使用查询语句得到数据，然后将内容改了即可。

mRealm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

//先查找后得到User对象

User user = mRealm.where(User.class).findFirst();

user.setAge(26);

}

});

### 删

* 使用deleteFromRealm()

//先查找到数据

final RealmResults<User> results= mRealm.where(User.class).findAll();

mRealm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

results.get(0).deleteFromRealm();

}

});

* 使用deleteFromRealm(int index)

mRealm.executeTransaction(new Realm.Transaction() {

@Override

public void execute(Realm realm) {

results.deleteFromRealm(0);

}

});

更多方法：

results.deleteFirstFromRealm(); //删除查到的user表的第一条数据

results.deleteLastFromRealm();//删除查到的user表的最后一条数据

results.deleteAllFromRealm();//删除查到的user表的全部数据

### 版本升级

当数据结构发生变化是，需要升级数据库。对于Realm来说，数据库升级就是迁移操作，把原来的数据库迁移到新结构的数据库。（体验：略麻烦）

例1：User类发生变化，移除age，新增个@Required的id字段。

User版本：version 0

String name;

Int age;

User版本：version 1

@Required

String id;

String name;

创建迁移类CustomMigration，需要实现RealmMigration接口。执行版本升级时的处理：

/\*\*

\* 升级数据库

\*/

class CustomMigration implements RealmMigration {

@Override

public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {

RealmSchema schema = realm.getSchema();

if (oldVersion == 0 && newVersion == 1) {

RealmObjectSchema personSchema = schema.get("User");

//新增@Required的id

personSchema

.addField("id", String.class, FieldAttribute.REQUIRED)

.transform(new RealmObjectSchema.Function() {

@Override

public void apply(DynamicRealmObject obj) {

//为已存在的数据设置id数据

obj.set("id", "1");

}

})

.removeField("age");//移除age属性

oldVersion++;

}

}

}

使用Builder.migration升级数据库，将版本号改为1（原版本号：0）。当Realm发现新旧版本号不一致时，会自动使用该迁移类完成迁移操作。

RealmConfiguration config = new RealmConfiguration.Builder()

.name("myrealm.realm") //文件名

.schemaVersion(1)

.migration(new CustomMigration())//升级数据库

.build();

例2：加入Dog类，User中加入Dog集合。

User版本：version 1

@Required

String id;

String name;

User版本：version 2

@Required

private String id;

private String name;

private RealmList<Dog> dogs;

Dog类

public class Dog extends RealmObject {

private String name;

private int age;

}

在迁移类CustomMigration中，继续添加处理方法。

/\*\*

\* 升级数据库

\*/

class CustomMigration implements RealmMigration {

@Override

public void migrate(DynamicRealm realm, long oldVersion, long newVersion) {

RealmSchema schema = realm.getSchema();

if (oldVersion == 1 && newVersion == 2) {

//创建Dog表

RealmObjectSchema dogSchema = schema.create("Dog");

dogSchema.addField("name", String.class);

dogSchema.addField("age", int.class);

//User中添加dogs属性

schema.get("User")

.addRealmListField("dogs", dogSchema)

.transform(new RealmObjectSchema.Function() {

@Override

public void apply(DynamicRealmObject obj) {

//为已存在的数据设置dogs数据

DynamicRealmObject dog = realm.createObject("Dog");

dog.set("name", "二哈");

dog.set("age", 2);

obj.getList("dogs").add(dog);

}

});

oldVersion++;

}

}

}

* 更多用法——RealmObjectSchema

例：取消id必填：

personSchema.setNullable("id", true):

例：移除id字段

personSchema.removeField("id");

例：重命名

personSchema..renameField("id", "userId");

* 更多用法——DynamicRealmObject

例：获取id

String id = obj.getString("id");

例：为字段设置值

obj.setString("name", "Gavin");

obj.setInt("id", 1);

obj.setLong("id", 1);

### 加密

（官方原文）Realm 文件可以通过传递一个512位（64字节）的密钥参数给 Realm.getInstance().encryptionKey() 来加密存储在磁盘上。

byte[] key = new byte[64];

new SecureRandom().nextBytes(key);

RealmConfiguration config = new RealmConfiguration.Builder()

.encryptionKey(key)

.build();

Realm realm = Realm.getInstance(config);

保证了所有永久性存储在磁盘上的数据都是通过标准 AES-256 加密的。每次创建新的 Realm 实例的时候，都需要提供相同的密钥

### 适配器（Adapter）(官方原文）

Realm提供了一些抽象的工具类来方便地将 OrderedRealmCollection（RealmResults和 RealmList 都实现了这个接口）展示到UI控件上。

RealmBaseAdapter 可以与ListView配合使用。参见

https://github.com/realm/realm-android-adapters/blob/master/example/src/main/java/io/realm/examples/adapters/ui/listview/MyListAdapter.java.

RealmRecyclerViewAdapter可以与 RecyclerView配合使用。参见

https://github.com/realm/realm-android-adapters/blob/master/example/src/main/java/io/realm/examples/adapters/ui/recyclerview/MyRecyclerViewAdapter.java.

你需要在 app 的 build.gradle中添加额外的依赖以使用这些适配器。

dependencies {

compile 'io.realm:android-adapters:1.4.0'

}

### Intents（官方原文）

你不可以直接通过 intent传递 RealmObject，建议你只传递RealmObject的标识符。举个常用例子，假如你的对象拥有一个主键，请通过intent 的 bundle 来传递这个主键的值。

// Assuming we had a person class with a @PrimaryKey on the 'id' field ...

Intent intent = new Intent(getActivity(), ReceivingService.class);

intent.putExtra("person\_id", person.getId());

getActivity().startService(intent);

在接受方（Activity、Service、IntentService、BroadcastReceiver及其它）从 bundle中解析出这个主键然后打开Realm 查询得到这个 RealmObject。

// in onCreate(), onHandleIntent(), etc.

String personId = intent.getStringExtra("person\_id");

Realm realm = Realm.getDefaultInstance();

Person person = realm.where(Person.class).equalTo("id", personId).findFirst();// do something with the person ...

realm.close();

可以参考示例中的 Object Passing部分。该示例展示了在安卓开发中常用的如何传递 id 并且得到对应的 RealmObject。

https://github.com/realm/realm-java/tree/master/examples/threadExample

### RxJava

对于这么火的RxJava，Realm又怎么会放过他。Realm 包含了对 RxJava 的原生支持。如下类可以被暴露为一个 Observable：Realm, RealmResults, RealmObject, DynamicRealm 和 DynamicRealmObject。（直接用一个官方的例子）

Realm realm = Realm.getDefaultInstance();

GitHubService api = retrofit.create(GitHubService.class);

realm.where(Person.class).isNotNull("username").findAllAsync().asObservable()

.filter(persons.isLoaded)

.flatMap(persons -> Observable.from(persons))

.flatMap(person -> api.user(person.getGithubUserName())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(user -> showUser(user));

这里使用asObservable轻松转换成RxJava中的Observable，使用.filter(persons.isLoaded)来判断是否已查询完成。

# WebSocket

## 背景

现在，很多网站为了实现推送技术，所用的技术都是轮询。轮询是在特定的的时间间隔（如每1秒），由浏览器对服务器发出HTTP请求，然后由服务器返回最新的数据给客户端的浏览器。这种传统的模式带来很明显的缺点，即浏览器需要不断的向服务器发出请求，然而HTTP请求可能包含较长的头部，其中真正有效的数据可能只是很小的一部分，显然这样会浪费很多的带宽等资源。

而比较新的技术去做轮询的效果是Comet。这种技术虽然可以双向通信，但依然需要反复发出请求。而且在Comet中，普遍采用的长链接，也会消耗服务器资源。

在这种情况下，HTML5定义了WebSocket协议，能更好的节省服务器资源和带宽，并且能够更实时地进行通讯。

## websocket与http

WebSocket是HTML5出的东西（协议），也就是说HTTP协议没有变化，或者说没关系，但HTTP是不支持持久连接的（长连接，循环连接的不算）。首先HTTP有 1.1 和 1.0 之说，也就是所谓的 keep-alive ，把多个HTTP请求合并为一个，但是 Websocket 其实是一个新协议，跟HTTP协议基本没有关系，只是为了兼容现有浏览器的握手规范而已，也就是说它是HTTP协议上的一种补充

另外Html5是指的一系列新的API，或者说新规范，新技术。Http协议本身只有1.0和1.1，而且跟Html本身没有直接关系。。通俗来说，你可以用HTTP协议传输非Html数据

## Websocket是什么样的协议，具体有什么优点

首先，Websocket是一个持久化的协议，相对于HTTP这种非持久的协议来说。简单的举个例子吧，用目前应用比较广泛的PHP生命周期来解释。

HTTP的生命周期通过 Request 来界定，也就是一个 Request 一个 Response ，那么在 HTTP1.0 中，这次HTTP请求就结束了。

在HTTP1.1中进行了改进，使得有一个keep-alive，也就是说，在一个HTTP连接中，可以发送多个Request，接收多个Response。但是请记住 Request = Response， 在HTTP中永远是这样，也就是说一个request只能有一个response。而且这个response也是被动的，不能主动发起。

而Websocket是基于HTTP协议的，或者说借用了HTTP的协议来完成一部分握手。首先我们来看个典型的 Websocket 握手

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

Origin: http://example.com

熟悉HTTP的童鞋可能发现了，这段类似HTTP协议的握手请求中，多了几个东西。我会顺便讲解下作用。

Upgrade: websocket

Connection: Upgrade

这个就是Websocket的核心了，告诉 Apache 、 Nginx 等服务器：注意啦，我发起的是Websocket协议，快点帮我找到对应的助理处理~不是那个老土的HTTP。

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

首先， Sec-WebSocket-Key 是一个 Base64 encode 的值，这个是浏览器随机生成的，告诉服务器：泥煤，不要忽悠窝，我要验证尼是不是真的是Websocket助理。

然后， Sec\_WebSocket-Protocol 是一个用户定义的字符串，用来区分同URL下，不同的服务所需要的协议。简单理解：今晚我要服务A，别搞错啦~

最后， Sec-WebSocket-Version 是告诉服务器所使用的 Websocket Draft （协议版本），在最初的时候，Websocket协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么Firefox和Chrome用的不是一个版本之类的，当初Websocket协议太多可是一个大难题。。不过现在还好，已经定下来啦~大家都使用的一个东西~ 脱水： 服务员，我要的是13岁的噢→\_→

然后服务器会返回下列东西，表示已经接受到请求， 成功建立Websocket啦！

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Sec-WebSocket-Protocol: chat

这里开始就是HTTP最后负责的区域了，告诉客户，我已经成功切换协议啦~

Upgrade: websocket

Connection: Upgrade

依然是固定的，告诉客户端即将升级的是 Websocket 协议，而不是mozillasocket，lurnarsocket或者shitsocket。

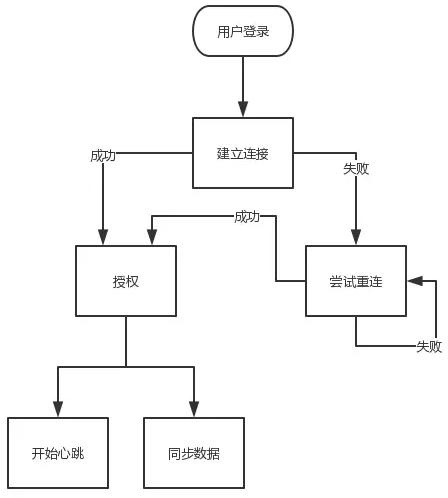
然后， Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key 。 服务器：好啦好啦，知道啦，给你看我的ID CARD来证明行了吧。。

后面的， Sec-WebSocket-Protocol 则是表示最终使用的协议。至此，HTTP已经完成它所有工作了，接下来就是完全按照Websocket协议进行了。具体的协议就不在这阐述了。

## 使用

### 流程

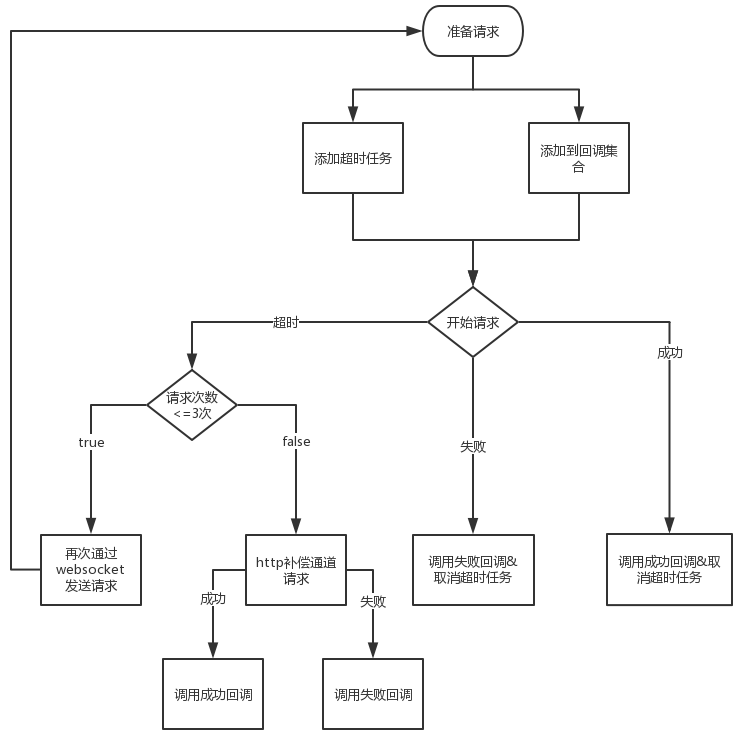
#### 用户登录流程



第一步用户输入账号密码登录成功后,我们将会通过websocket协议建立连接,当连接失败回调的时候我们尝试重连,直到连接成功,当然这个尝试重连的时间间隔我是根据重连失败次数按一定规则写的具体后面再说.

第二步当连接建立成功后,我们需要在后台通过长连接发送请求验证该用户的身份也就是上图的授权,既然前面用户登录都成功了一般情况下授权是不会失败的,所以这里对于授权失败并未处理,授权成功后我们开启心跳,并且发送同步数据请求到服务端获取还未收到的消息

#### **客户端发送请求流程**

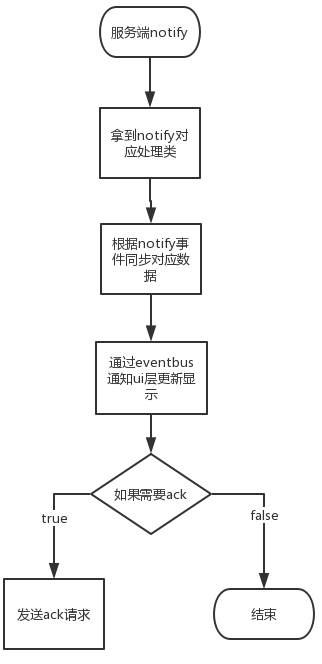


第一步将请求参数封装成请求对象,然后添加超时任务并且将该请求的回调添加到回调集合

这里有点需要说明下,封装请求参数的时候这里额外添加了两个参数seqId和reqCount,这里我们是通过长连接请求当服务端响应的时候为了能够找到对应的回调,所以每个请求我们都需要传给服务端一个唯一标识来标识该请求,这里我用的seqId,请求成功后服务端再把seqId回传,我们再通过这个seqId作为key从回调集合中找到对应的回调.而reqCount的话主要针对请求超时的情况,如果请求超时,第二次请求的时候就把reqCount++在放入request中,我们约定同一个请求次数大于三次时候走http补偿通道,那么当request中的reqCount>3的时候我们就通过http发送该请求,然后根据响应回调对应结果.

第二步开始请求,成功或者失败的话通过seqId找到对应回调执行并从回调集合中移除该回调,然后取消超时任务.如果超时的话根据seqId拿到对应的回调并从回调集合中移除该回调,然后判断请求次数如果小于等于3次再次通过websocket尝试请求,如果大于3次通过http请求,根据请求成功失败情况执行对应回调.

#### 服务端主动推送消息流程



* 这里服务端推送的消息仅仅是个事件,不携带具体消息.
* 第一步根据notify中事件类型找到对应的处理类,一般情况下这里需要同步对应数据.
* 第二步然后用eventbus通知对应的ui界面更新
* 第三步如果需要ack,发送ack请求

### 具体实现

#### 依赖和管理类单例

implementation 'com.neovisionaries:nv-websocket-client:2.+'

public class WsManager {

private static WsManager mInstance;

private WsManager() {}

public static WsManager getInstance(){

if(mInstance == null){

synchronized (WsManager.class){

if(mInstance == null) mInstance = new WsManager();

}

}

return mInstance;

}

}

#### 建立连接

建立连接代码,这里关于WebSocket协议的操作用的都是nv-websocket-client这个库

public class WsManager {

//region singleton

private static volatile WsManager mInstance;

private WsManager() {

}

public static WsManager getInstance(){

if(mInstance == null){

synchronized (WsManager.class){

if(mInstance == null){

mInstance = new WsManager();

}

}

}

return mInstance;

}

//endregion

//region WsStatus and listenerAdapter

/\*\*

\* WebSocket的状态

\*/

public enum WsStatus {

CONNECT\_SUCCESS,//连接成功

CONNECT\_FAIL,//连接失败

CONNECTING;//正在连接

}

private WsStatus mStatus;

private void setStatus(WsStatus status){

this.mStatus = status;

}

private WsStatus getStatus(){

return mStatus;

}

private WsListener mListener;

/\*\*

\* 继承默认的监听空实现WebSocketAdapter,重写我们需要的方法

\* onTextMessage 收到文字信息

\* onConnected 连接成功

\* onConnectError 连接失败

\* onDisconnected 连接关闭

\*/

private class WsListener extends WebSocketAdapter {

@Override

public void onTextMessage(WebSocket websocket, String text) throws Exception {

super.onTextMessage(websocket, text);

Logger.t(TAG).d(text);

}

@Override

public void onConnected(WebSocket websocket, Map<String, List<String>> headers) throws Exception {

super.onConnected(websocket, headers);

Logger.t(TAG).d("连接成功");

setStatus(WsStatus.CONNECT\_SUCCESS);

}

@Override

public void onConnectError(WebSocket websocket, WebSocketException exception) throws Exception {

super.onConnectError(websocket, exception);

Logger.t(TAG).d("连接错误");

setStatus(WsStatus.CONNECT\_FAIL);

}

@Override

public void onDisconnected(WebSocket websocket, WebSocketFrame serverCloseFrame, WebSocketFrame clientCloseFrame, boolean closedByServer) throws Exception {

super.onDisconnected(websocket, serverCloseFrame, clientCloseFrame, closedByServer);

Logger.t(TAG).d("断开连接");

setStatus(WsStatus.CONNECT\_FAIL);

}

}

//endregion

//region declare variable

private final String TAG = this.getClass().getSimpleName();

private static final int FRAME\_QUEUE\_SIZE = 5;

private static final int CONNECT\_TIMEOUT = 5000;

private static final String DEF\_TEST\_URL = "测试服地址";//测试服默认地址

private static final String DEF\_RELEASE\_URL = "正式服地址";//正式服默认地址

private static final String DEF\_URL = BuildConfig.DEBUG ? DEF\_TEST\_URL : DEF\_RELEASE\_URL;

private String url;

private WebSocket ws;

//endregion

//region method

public void init(){

try {

/\*

\* configUrl其实是缓存在本地的连接地址

\* 这个缓存本地连接地址是app启动的时候通过http请求去服务端获取的,

\* 每次app启动的时候会拿当前时间与缓存时间比较,超过6小时就再次去服务端获取新的连接地址更新本地缓存

\*/

String configUrl = "";

url = TextUtils.isEmpty(configUrl) ? DEF\_URL : configUrl;

ws = new WebSocketFactory().createSocket(url, CONNECT\_TIMEOUT)

.setFrameQueueSize(FRAME\_QUEUE\_SIZE)//设置帧队列最大值为5

.setMissingCloseFrameAllowed(false)//不允许服务端关闭连接却未发送关闭帧

.addListener(mListener = new WsListener())//添加回调监听

.connectAsynchronously();//异步连接

setStatus(WsStatus.CONNECTING);

Logger.t(TAG).d("第一次连接");

} catch (IOException e) {

e.printStackTrace();

}

}

public void disconnect(){

if(ws != null) ws.disconnect();

}

//endregion

}

从注释我们可以知道,这里我们是app启动的时候通过http请求获取WebSocket连接地址,如果获取失败就走本地默认的url建立连接.并且内部自己维护了一个websocket状态，后面发送请求和重连的时候会用上.

其实获取连接地址这个地方是可以优化的,就是app启动的时候先比较上次获取的时间如果大于6小时就通过http请求获取websocket的连接地址,这个地址应该是个列表,然后存入本地,连接的时候我们可以先ping下地址,选择耗时最短的地址接入.如果连不上我们在连耗时第二短的地址以此类推.但这里我们就以简单的方式做了.

至于建立连接代码在哪调用的话,我选择的是主界面onCreate()的时候,因为一般能进入主界面了,就代表用户已经登录成功.WsManager.getInstance().init();断开连接的话在主界面onDestroy()的时候调用WsManager.getInstance().disconnect();

#### 重连

建立连接有成功就有失败,对于失败情况我们需要重连,那么下面我们分别说明重连的时机,重连的策略和当前是否应该重连的判断.对于重连的时机有如下几种情况我们需要尝试重连：

1. 应用网络的切换.具体点就是可用网络状态的切换,比如4g切wifi连接会断开我们需要重连.
2. 应用回到前台的时候,判断如果连接断开我们需要重连,这个是尽量保持当应用再前台的时候连接的稳定.
3. 收到连接失败或者连接断开事件的时候,这个没什么好解释.
4. 心跳连续3次失败时候.当然这个连续失败3次是自己定义的,大伙可以根据自己app的情况定制.

等会我们先展示前三种情况,心跳失败这个在后面我们把客户端发送请求讲完再说.上面把需要重连的情景说了,现在讲讲具体的重连策略.

这里我定义了一个最小重连时间间隔min和一个最大重连时间间隔max,当重连次数小于等于3次的时候都以最小重连时间间隔min去尝试重连,当重连次数大于3次的时候我们将重连地址替换成默认地址DEF\_URL,将重连时间间隔按min\*(重连次数-2)递增最大不不超过max.

还有最后一个当前是否应该重连的判断

* 用户是否登录,可以通过本地是否有缓存的用户信息来判断.因为重连成功后我们需要将用户信息通过WebSocket发送到服务器进行身份验证所以这里必须登录成功.
* 当前连接是否可用,这个通过nv-websocket-client库中的api判断ws.isOpen().
* 当前不是正在连接状态,这里我们根据自己维护的状态来判断getStatus() != WsStatus.CONNECTING.
* 当前网络可用

##### 收到连接失败或者连接断开事件的时候进行重连，实现了需要重连的情景3

public class WsManager {

.....省略部分跟之前代码一样.....

private Handler mHandler = new Handler();

private int reconnectCount = 0;//重连次数

private long minInterval = 3000;//重连最小时间间隔

private long maxInterval = 60000;//重连最大时间间隔

private class WsListener extends WebSocketAdapter {

@Override

public void onTextMessage(WebSocket websocket, String text) throws Exception {

super.onTextMessage(websocket, text);

Logger.t(TAG).d(text);

}

@Override

public void onConnected(WebSocket websocket, Map<String, List<String>> headers)

throws Exception {

super.onConnected(websocket, headers);

Logger.t(TAG).d("连接成功");

setStatus(WsStatus.CONNECT\_SUCCESS);

cancelReconnect();//连接成功的时候取消重连,初始化连接次数

}

@Override

public void onConnectError(WebSocket websocket, WebSocketException exception)

throws Exception {

super.onConnectError(websocket, exception);

Logger.t(TAG).d("连接错误");

setStatus(WsStatus.CONNECT\_FAIL);

reconnect();//连接错误的时候调用重连方法

}

@Override

public void onDisconnected(WebSocket websocket, WebSocketFrame serverCloseFrame, WebSocketFrame clientCloseFrame, boolean closedByServer)

throws Exception {

super.onDisconnected(websocket, serverCloseFrame, clientCloseFrame, closedByServer);

Logger.t(TAG).d("断开连接");

setStatus(WsStatus.CONNECT\_FAIL);

reconnect();//连接断开的时候调用重连方法

}

}

public void reconnect() {

if (!isNetConnect()) {

reconnectCount = 0;

Logger.t(TAG).d("重连失败网络不可用");

return;

}

/\*  
 \*这里其实应该还有个用户是否登录了的判断 因为当连接成功后我们需要发送用户信息到服务端进行校验  
 \*由于我们这里是个demo所以省略了  
 \*webSocket不为空 && 当前连接断开了 && 不是正在重连状态  
 \*/

if (ws != null && !ws.isOpen() && getStatus() != WsStatus.CONNECTING) {

reconnectCount++;

setStatus(WsStatus.CONNECTING);

long reconnectTime = minInterval;

if (reconnectCount > 3) {

url = DEF\_URL;

long temp = minInterval \* (reconnectCount - 2);

reconnectTime = temp > maxInterval ? maxInterval : temp;

}

Logger.t(TAG).d("准备开始第%d次重连,重连间隔%d -- url:%s", reconnectCount, reconnectTime, url);

mHandler.postDelayed(mReconnectTask, reconnectTime);

}

}

private Runnable mReconnectTask = new Runnable() {

@Override

public void run() {

try {

ws = new WebSocketFactory().createSocket(url, CONNECT\_TIMEOUT)

.setFrameQueueSize(FRAME\_QUEUE\_SIZE)//设置帧队列最大值为5

.setMissingCloseFrameAllowed(false)//设置不允许服务端关闭连接却未发送关闭帧

.addListener(mListener = new WsListener())//添加回调监听

.connectAsynchronously();//异步连接

} catch (IOException e) {

e.printStackTrace();

}

}

};

private void cancelReconnect() {

reconnectCount = 0;

mHandler.removeCallbacks(mReconnectTask);

}

/\*\*  
 \* 判断当前网络是否可用  
 \*/

private boolean isNetConnect() {

ConnectivityManager connectivity = (ConnectivityManager) App.getInstance().getSystemService(Context.CONNECTIVITY\_SERVICE);

if (connectivity != null) {

NetworkInfo info = connectivity.getActiveNetworkInfo();

if (info != null && info.isConnected()) {

// 当前网络是连接的

if (info.getState() == NetworkInfo.State.CONNECTED) {

return true;

}

}

}

return false;

}

}

上面代码通过handler实现了一定时间间隔的重连,然后我们在WsListener监听中的onConnectError()和onDisconnected()调用了reconnect()实现重连,onConnected()中调用了cancelReconnect()取消重连并初始化重连次数.所以当需要重连的时候我们调用reconnect()方法,如果失败onConnectError()和onDisconnected()回调会再次调用reconnect()实现重连,如果成功onConnected()中会调用cancelReconnect()取消重连并初始化重连次数.

##### 网络的切换通过广播来监听实现重连，实现实现情景1

public class NetStatusReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

String action = intent.getAction();

if (ConnectivityManager.CONNECTIVITY\_ACTION.equals(action)) {

// 获取网络连接管理器

ConnectivityManager connectivityManager

= (ConnectivityManager) WsApplication.getContext()

.getSystemService(Context.CONNECTIVITY\_SERVICE);

// 获取当前网络状态信息

NetworkInfo info = connectivityManager.getActiveNetworkInfo();

if (info != null && info.isAvailable()) {

Logger.t("WsManager").d("监听到可用网络切换,调用重连方法");

WsManager.getInstance().reconnect();//wify 4g切换重连websocket

}

}

}

}

##### 应用回到前台情况的重连，实现情景2

首先创建前台后台ActivityLifecycleCallbacks 监听

public class ForegroundCallbacks implements Application.ActivityLifecycleCallbacks {  
 private static ForegroundCallbacks *instance*;  
 private static final long *CHECK\_DELAY* = 600;//变为后台600毫秒后才执行后台回调  
 public static final String *TAG* = ForegroundCallbacks.class.getName();  
 private boolean foreground = false, paused = true;  
 private Handler handler = new Handler();  
 private List<Listener> listeners = new CopyOnWriteArrayList<>();  
 private Runnable check;  
 public static ForegroundCallbacks init(Application application) {  
 if (*instance* == null) {  
 *instance* = new ForegroundCallbacks();  
 application.registerActivityLifecycleCallbacks(*instance*);  
 }  
 return *instance*;  
 }  
 public static ForegroundCallbacks get(Application application) {  
 if (*instance* == null) {  
 *init*(application);  
 }  
 return *instance*;  
 }  
 public static ForegroundCallbacks get(Context ctx) {  
 if (*instance* == null) {  
 Context appCtx = ctx.getApplicationContext();  
 if (appCtx instanceof Application) {  
 *init*((Application) appCtx);  
 }  
 throw new IllegalStateException("Foreground is not initialised and " + "cannot obtain the Application object");  
 }  
 return *instance*;  
 }  
 public static ForegroundCallbacks get() {  
 return *instance*;  
 }  
 public boolean isForeground() {  
 return foreground;  
 }  
  
 public boolean isBackground() {  
 return !foreground;  
 }  
 public void addListener(Listener listener) {  
 listeners.add(listener);  
 }  
 public void removeListener(Listener listener) {  
 listeners.remove(listener);  
 }  
 @Override  
 public void onActivityResumed(Activity activity) {  
 paused = false;  
 if (check != null) handler.removeCallbacks(check);  
 //如果不在前台  
 if (!foreground) {  
 foreground = true;  
 for (Listener l : listeners) {  
 try {  
 //回到前台了  
 l.onBecameForeground();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }   
 }  
 @Override  
 public void onActivityPaused(Activity activity) {  
 paused = true;  
 if (check != null) handler.removeCallbacks(check);  
 handler.postDelayed(check = () -> {  
 if (foreground && paused) {  
 foreground = false;  
 for (Listener l : listeners) {  
 try {  
 //变为后台了  
 l.onBecameBackground();  
 } catch (Exception e) {  
 e.printStackTrace();  
  
 }  
 }  
 }  
 }, *CHECK\_DELAY*);  
 }  
 @Override  
 public void onActivityCreated(Activity activity, Bundle savedInstanceState) {}  
 @Override  
 public void onActivityStarted(Activity activity) {}  
 @Override  
 public void onActivityStopped(Activity activity) {}  
 @Override  
 public void onActivitySaveInstanceState(Activity activity, Bundle outState) {}  
 @Override  
 public void onActivityDestroyed(Activity activity) {}  
 public interface Listener {  
 void onBecameForeground();  
 void onBecameBackground();  
 }  
}

然后在application中初始化该监听,当应用回到前台的时候尝试重连

public class WsApplication extends Application {

@Override

public void onCreate() {

super.onCreate();

initAppStatusListener();

}

private void initAppStatusListener() {

ForegroundCallbacks.init(this).addListener(new ForegroundCallbacks.Listener() {

@Override

public void onBecameForeground() {

Logger.t("WsManager").d("应用回到前台调用重连方法");

WsManager.getInstance().reconnect();

}

@Override

public void onBecameBackground() {

Logger.t("WsManager").d("应用回到后台调用？？方法");

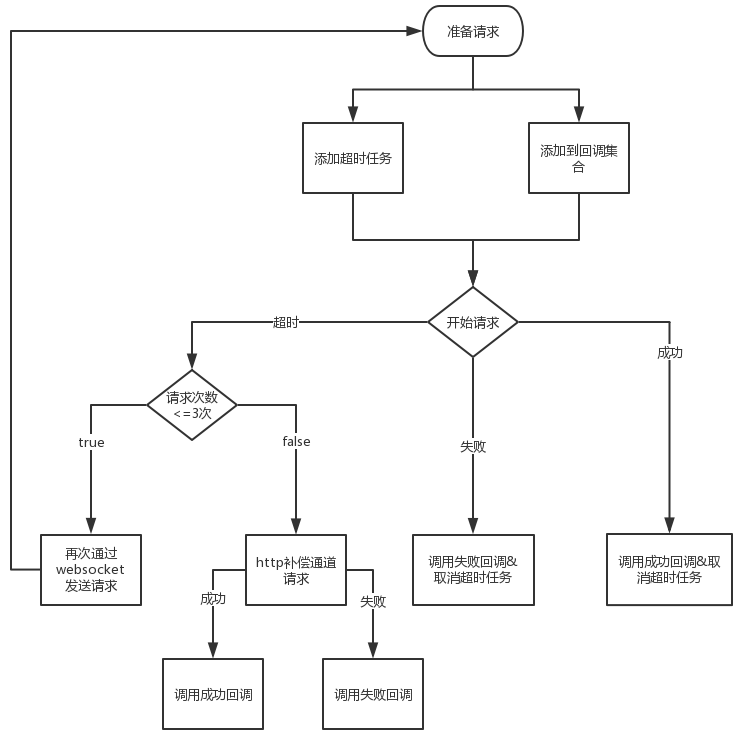
}

});

}

}

#### 客户端发送请求



既然是请求,那么得有一个请求协议.这里我们通过http请求,引导到websocket请求协议.一般http请求我们会需要如下几个参数：url， 请求参数， 回调

而websocket是一个长连接,我们所有的请求都是通过这个连接发送的,也就是说我们没法像http那样通过一个url来区分不同请求,那么我们请求协议中就必须要有一个类似url的东西去区分请求,这里我用action字段来表示url,为了方便协议的扩展,我们在加一个参数req\_event,由action和req\_event共同组成url,下面是一个简单的例子.

{

"action": " login",

"req\_event": 0

}

由于请求是异步的所以当我们请求的时候需要把回调加入集合保存起来,等到服务端响应的时候我们需要从集合中找到对应回调调用对应的方法,那么为了能够找到对应回调我们需要给每个请求增加一个唯一标识,这里我用的seq\_id字段表示,当请求的时候我们把客户端自己生成的seq\_id传给服务端,然后当服务端响应的时候在回传给我们,这样我们就能通过seq\_id作为key找到对应的回调,那么现在协议将变成下面这样.

{

"action": " login",

"req\_event": 0,

"seq\_id":0

}

在加上每个请求都有的请求参数,这里我用req字段表示,最终协议如下.

{

"action": " login",

"req\_event": 0,

"seq\_id":0,

"req":{

"aaa":"aaa",

"bbb":0

}

}

Show Code

Action一个枚举类

把action、req\_event、响应实体统一封装在一起,action和req\_event用来组装url,响应实体在反序列化的时候会用到.调用者只需根据不同请求传入对应Action即可.

public enum Action {  
 *LOGIN*("login", 1, null);  
 private String action;  
 private int reqEvent;  
 private Class respClazz;  
 Action(String action, int reqEvent, Class respClazz) {  
 this.action = action;  
 this.reqEvent = reqEvent;  
 this.respClazz = respClazz;  
 }  
 public String getAction() {  
 return action;  
 }  
 public int getReqEvent() {  
 return reqEvent;  
 }  
 public Class getRespClazz() {  
 return respClazz;  
 }  
}

ui层回调

public interface ICallback<T> {  
 void onSuccess(T t);  
 void onFail(String msg);  
}

协议对应的请求实体类Request和建造者Builder

这里还额外添加了一个没有参加序列化的参数reqCount来表示该请求的请求次数,在后面请求失败情况下对该请求的处理会用上.

public class Request<T> {  
 @SerializedName("action")  
 private String action;  
 @SerializedName("req\_event")  
 private int reqEvent;  
 @SerializedName("seq\_id")  
 private long seqId;  
 @SerializedName("req")  
 private T req;  
 private transient int reqCount;  
 public Request(String action, int reqEvent, long seqId, T req, int reqCount) {  
 this.action = action;  
 this.reqEvent = reqEvent;  
 this.seqId = seqId;  
 this.req = req;  
 this.reqCount = reqCount;  
 }  
 //....这里还有各个参数对应get、set方法,为节省篇幅省略了...  
 public static class Builder<T> {  
 private String action;  
 private int reqEvent;  
 private long seqId;  
 private T req;  
 private int reqCount;  
 public Builder action(String action) {  
 this.action = action;  
 return this;  
 }  
 public Builder reqEvent(int reqEvent) {  
 this.reqEvent = reqEvent;  
 return this;  
 }  
 public Builder seqId(long seqId) {  
 this.seqId = seqId;  
 return this;  
 }  
 public Builder req(T req) {  
 this.req = req;  
 return this;  
 }  
 public Builder reqCount(int reqCount) {  
 this.reqCount = reqCount;  
 return this;  
 }  
 public Request build() {  
 return new Request<T>(action, reqEvent, seqId, req, reqCount);  
 }  
 }  
}

WsManager中发送请求代码

private static final int *REQUEST\_TIMEOUT* = 10000;//请求超时时间  
private AtomicLong seqId = new AtomicLong(SystemClock.*uptimeMillis*());//每个请求的唯一标识  
public void sendReq(Action action, Object req, ICallback callback) {  
 sendReq(action, req, callback, *REQUEST\_TIMEOUT*);  
}  
public void sendReq(Action action, Object req, ICallback callback, long timeout) {  
 sendReq(action, req, callback, timeout, 1);  
}  
*/\*\*  
 \** ***@param*** *action Action  
 \** ***@param*** *req 请求参数  
 \** ***@param*** *callback 回调  
 \** ***@param*** *timeout 超时时间  
 \** ***@param*** *reqCount 请求次数  
 \*/*@SuppressWarnings("unchecked")  
private <T> void sendReq(Action action, T req, ICallback callback, long timeout, int reqCount) {  
 if (!isNetConnect()) {  
 callback.onFail("网络不可用");  
 return;  
 }  
 Request request = new Request.Builder<T>()  
 .action(action.getAction())  
 .reqEvent(action.getReqEvent())  
 .seqId(seqId.getAndIncrement())  
 .reqCount(reqCount)  
 .req(req)  
 .build();  
 Logger.*t*(TAG).d("send text : %s", new Gson().toJson(request));  
 ws.sendText(new Gson().toJson(request));  
}

#### 超时和回调的处理

发送请求ok了接下来需要处理回调的问题了.虽然在方法sendReq(Action action, T req, ICallback callback, long timeout, int reqCount)中我们已经传入了ui层的回调ICallback,但这里还需要在封装一层回调，处理一些通用逻辑,然后再调用ICallback对应方法.

需要处理的通用逻辑有如下三个：

* 请求成功：将服务端返回数据从子线程传到主线程,然后调用ui层回调.
* 请求失败：将失败信息从子线程传到主线程,然后调用ui层回调.
* 请求超时：该请求的reqCount <= 3再次通过websocket发送请求,reqCount > 3通过http通道发送请求,根据结果直接调用对应回调.

##### 中间层回调**定义**如下

onSuccess与普通的成功回调一样,onError和onTimeout回调中有Request与Action是为了方便后续再次请求操作.

public interface IWsCallback<T> {  
 void onSuccess(T t);  
 void onError(String msg, Request request, Action action);  
 void onTimeout(Request request, Action action);  
}

##### Handler中**获取**的Callback与Data包装类如下

public class CallbackDataWrapper<T> {  
 private ICallback<T> callback;  
 private Object data;  
 public CallbackDataWrapper(ICallback<T> callback, Object data) {  
 this.callback = callback;  
 this.data = data;  
 }  
 public ICallback<T> getCallback() {  
 return callback;  
 }  
 public void setCallback(ICallback<T> callback) {  
 this.callback = callback;  
 }  
 public Object getData() {  
 return data;  
 }  
 public void setData(Object data) {  
 this.data = data;  
 }  
}

##### 超时后的处理

private final int SUCCESS\_HANDLE = 0x01;  
private final int ERROR\_HANDLE = 0x02;

//请求回调接收的handler

private Handler mCallbackHandler = new Handler(Looper.*getMainLooper*()) {  
 @Override  
 public void handleMessage(Message msg) {  
 switch (msg.what) {  
 case SUCCESS\_HANDLE:  
 CallbackDataWrapper successWrapper = (CallbackDataWrapper) msg.obj;  
 successWrapper.getCallback().onSuccess(successWrapper.getData());  
 break;  
 case ERROR\_HANDLE:  
 CallbackDataWrapper errorWrapper = (CallbackDataWrapper) msg.obj;  
 errorWrapper.getCallback().onFail((String) errorWrapper.getData());  
 break;  
 }  
 }  
};  
*/\*\*  
 \** ***@param*** *action Action  
 \** ***@param*** *req 请求参数  
 \** ***@param*** *callback 回调  
 \** ***@param*** *timeout 超时时间  
 \** ***@param*** *reqCount 请求次数  
 \*/*@SuppressWarnings("unchecked")  
private <T> void sendReq(Action action, T req, final ICallback callback, final long timeout, int reqCount) {  
 if (!isNetConnect()) {  
 callback.onFail("网络不可用");  
 return;  
 }  
 Request request = new Request.Builder<T>()  
 .action(action.getAction())  
 .reqEvent(action.getReqEvent())  
 .seqId(seqId.getAndIncrement())  
 .reqCount(reqCount)  
 .req(req)  
 .build();  
 IWsCallback temp = new IWsCallback() {  
 @Override  
 public void onSuccess(Object o) {

//发送成功信息到请求回调接收的handler  
 mCallbackHandler.obtainMessage(SUCCESS\_HANDLE, new CallbackDataWrapper(callback, o)).sendToTarget();  
 }  
 @Override  
 public void onError(String msg, Request request, Action action) {  
 mCallbackHandler.obtainMessage(ERROR\_HANDLE, new CallbackDataWrapper(callback, msg)).sendToTarget();  
 }  
 @Override  
 public void onTimeout(Request request, Action action) {  
 timeoutHandle(request, action, callback, timeout);  
 }  
 };  
 Logger.*t*(TAG).d("send text : %s", new Gson().toJson(request));  
 ws.sendText(new Gson().toJson(request));  
}  
*/\*\*  
 \* 超时处理  
 \*/*private void timeoutHandle(Request request, Action action, ICallback callback, long timeout) {  
 if (request.getReqCount() > 3) {  
 Logger.*t*(TAG).d("(action:%s)连续3次请求超时 执行http请求", action.getAction());  
 //走http请求  
 } else {  
 sendReq(action, request.getReq(), callback, timeout, request.getReqCount() + 1);  
 Logger.*t*(TAG).d("(action:%s)发起第%d次请求", action.getAction(), request.getReqCount());  
 }  
}

##### callback包装类

public class CallbackWrapper {  
 private final IWsCallback tempCallback;  
 private final ScheduledFuture timeoutTask;  
 private final Action action;  
 private final Request request;  
 public CallbackWrapper(IWsCallback tempCallback, ScheduledFuture timeoutTask, Action action, Request request) {  
 this.tempCallback = tempCallback;  
 this.timeoutTask = timeoutTask;  
 this.action = action;  
 this.request = request;  
 }  
 public IWsCallback getTempCallback() {  
 return tempCallback;  
 }  
 public ScheduledFuture getTimeoutTask() {  
 return timeoutTask;  
 }  
 public Action getAction() {  
 return action;  
 }  
 public Request getRequest() {  
 return request;  
 }  
}

##### 实现添加超时任务代码

这里我们用的ScheduledThreadPoolExecutor来处理超时任务,通过ScheduledThreadPoolExecutor.schedule()方法间隔一定时长执行超时的处理.当然为了能进行超时的处理我们需要将callback保存起来,又由于超时的时候我们需要进行再次请求所以不仅需要callback还需要request与action,至于执行超时任务返回的ScheduledFuture这个我们会在服务端响应的时候取消超时任务用上.最终通过CallbackWrapper将他们包装了起来.

private ScheduledExecutorService executor = Executors.*newSingleThreadScheduledExecutor*();  
private Map<Long, CallbackWrapper> callbacks = new HashMap<>();  
@SuppressWarnings("unchecked")  
private <T> void sendReq(Action action, T req, final ICallback callback, final long timeout, int reqCount) {  
 if (!isNetConnect()) {  
 callback.onFail("网络不可用");  
 return;  
 }  
 Request request = new Request.Builder<T>()  
 .action(action.getAction())  
 .reqEvent(action.getReqEvent())  
 .seqId(seqId.getAndIncrement())  
 .reqCount(reqCount)  
 .req(req)  
 .build();  
 ScheduledFuture timeoutTask = enqueueTimeout(request.getSeqId(), timeout);//添加超时任务  
 IWsCallback tempCallback = new IWsCallback() {  
 @Override  
 public void onSuccess(Object o) {  
 mCallbackHandler.obtainMessage(SUCCESS\_HANDLE, new CallbackDataWrapper(callback, o)).sendToTarget();  
 }  
 @Override  
 public void onError(String msg, Request request, Action action) {  
 mCallbackHandler.obtainMessage(ERROR\_HANDLE, new CallbackDataWrapper(callback, msg)).sendToTarget();  
 }  
 @Override  
 public void onTimeout(Request request, Action action) {  
 timeoutHandle(request, action, callback, timeout);  
 }  
 };  
 callbacks.put(request.getSeqId(), new CallbackWrapper(tempCallback, timeoutTask, action, request));  
 Logger.*t*(TAG).d("send text : %s", new Gson().toJson(request));  
 ws.sendText(new Gson().toJson(request));  
}  
*/\*\*  
 \* 添加超时任务  
 \*/*private ScheduledFuture enqueueTimeout(final long seqId, long timeout) {  
 return executor.schedule(new Runnable() {  
 @Override  
 public void run() {  
 CallbackWrapper wrapper = callbacks.remove(seqId);  
 if (wrapper != null) {  
 Logger.*t*(TAG).d("(action:%s)第%d次请求超时", wrapper.getAction().getAction(), wrapper.getRequest().getReqCount());  
 wrapper.getTempCallback().onTimeout(wrapper.getRequest(), wrapper.getAction());  
 }  
 }  
 }, timeout, TimeUnit.*MILLISECONDS*);  
}

#### 服务端响应的处理

这里我们还是先从协议入手.首先说明下对于长连接,无论是我们请求的响应还是服务端的主动通知,都是通过同一个回调方法回调给客户端,我们没法区分,所以服务端给我们数据中需要带一个标志,来区分普通请求的响应与服务端主动通知.这里我们用resp\_event来表示,当resp\_event为10的时候是请求的响应,为20的时候是服务端的主动通知.

对于请求的响应我们需要通过seq\_id找到对应的回调,对于主动通知我们需要添加一个action来区分这个通知执行什么操作

最后再加上响应数据体resp协议就定制完成了.当然seq\_id应该在客户端请求的响应的时候才有,action应该在服务端主动通知的时候才有.这里为了解析的方便最外层通用协议如下.

{

"resp\_event": 10,

"action": "",

"seq\_id": 11111111,

"resp": {}

}

对于我们客户端主动请求的响应数据可以按照http请求的格式来定义code,msg,data那么最终数据结构如下

{

"resp\_event": 10,

"action": "",

"seq\_id": 11111111,

"resp": {

"code":0,

"msg":"ok",

"data":{

}

}

}

对于服务端主动的推送不需要code和msg那么数据结构改为如下

{

"resp\_event": 20,

"action": "",

"seq\_id": 11111111,

"resp": {

}

}

可以看出最外层json数据结构一样,从resp开始不同.这里对于客户端主动请求的响应的情况我已经封装好了Codec工具类去解析,服务端主动推送的话后面再说。

最外层bean如下

public class Response {  
 @SerializedName("resp\_event")  
 private int respEvent;  
 @SerializedName("seq\_id")  
 private String seqId;  
 private String action;  
 private String resp;  
 //省略get set方法  
}

第二层bean如下

public class ChildResponse {  
 private int code;  
 private String msg;  
 private String data;  
 public boolean isOK(){  
 return code == 0;  
 }  
 //省略get set方法  
}

对应的解析工具类

public class Codec {  
 public static Response decoder(String text) {  
 Response response = new Response();  
 JsonParser parser = new JsonParser();  
 JsonElement element = parser.parse(text);  
 if (element.isJsonObject()) {  
 JsonObject obj = (JsonObject) element;  
 response.setRespEvent(*decoderInt*(obj, "resp\_event"));  
 response.setAction(*decoderStr*(obj, "action"));  
 response.setSeqId(*decoderStr*(obj, "seq\_id"));  
 response.setResp(*decoderStr*(obj, "resp"));  
 return response;  
 }  
 return response;  
 }  
 private static int decoderInt(JsonObject obj, String name) {  
 int result = -1;  
 JsonElement element = obj.get(name);  
 if (null != element) {  
 try {  
 result = element.getAsInt();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 return result;  
 }  
 private static String decoderStr(JsonObject obj, String name) {  
 String result = "";  
 try {  
 JsonElement element = obj.get(name);  
 if (null != element && element.isJsonPrimitive()) {  
 result = element.getAsString();  
 } else if (null != element && element.isJsonObject()) {  
 result = element.getAsJsonObject().toString();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return result;  
 }  
 public static ChildResponse decoderChildResp(String jsonStr) {  
 ChildResponse childResponse = new ChildResponse();  
 JsonParser parser = new JsonParser();  
 JsonElement element = parser.parse(jsonStr);  
 if (element.isJsonObject()) {  
 JsonObject jsonObject = (JsonObject) element;  
 childResponse.setCode(*decoderInt*(jsonObject, "code"));  
 childResponse.setMsg(*decoderStr*(jsonObject, "msg"));  
 childResponse.setData(*decoderStr*(jsonObject, "data"));  
 }  
 return childResponse;  
 }  
}

错误信息的封装ErrorCode如下

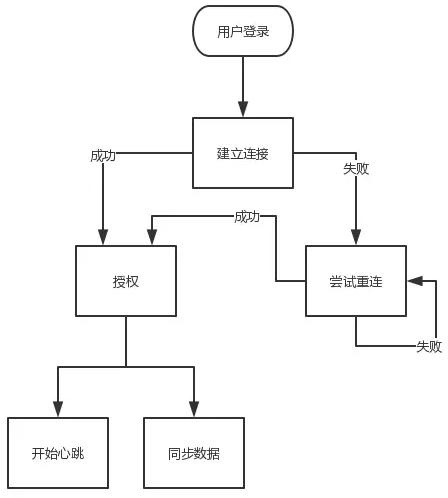
public enum ErrorCode {  
 *BUSINESS\_EXCEPTION*("业务异常"),  
 *PARSE\_EXCEPTION*("数据格式异常"),  
 *DISCONNECT\_EXCEPTION*("连接断开");  
 private String msg;  
 ErrorCode(String msg) {  
 this.msg = msg;  
 }  
 public String getMsg() {  
 return msg;  
 }  
 public void setMsg(String msg) {  
 this.msg = msg;  
 }  
}

接下来就是响应的处理了

private class WsListener extends WebSocketAdapter {  
 @Override  
 public void onTextMessage(WebSocket websocket, String text) throws Exception {  
 super.onTextMessage(websocket, text);  
 Logger.*t*(TAG).d("receiverMsg:%s", text);  
 Response response = Codec.*decoder*(text);//解析出第一层bean  
 if (response.getRespEvent() == 10) {//响应  
 //找到对应callback，并移除  
 CallbackWrapper wrapper = callbacks.remove(Long.*parseLong*(response.getSeqId()));  
 if (wrapper == null) {  
 Logger.*t*(TAG).d("(action:%s) not found callback", response.getAction());  
 return;  
 }  
 try {  
 //取消超时任务  
 wrapper.getTimeoutTask().cancel(true);  
 ChildResponse childResponse = Codec.*decoderChildResp*(response.getResp());//解析第二层bean  
 if (childResponse.isOK()) {  
 Object o = new Gson().fromJson(childResponse.getData(), wrapper.getAction().getRespClazz());  
 wrapper.getTempCallback().onSuccess(o);  
 } else {  
 wrapper.getTempCallback()  
 .onError(ErrorCode.BUSINESS\_EXCEPTION.getMsg(), wrapper.getRequest(), wrapper.getAction());  
 }  
 } catch (JsonSyntaxException e) {  
 e.printStackTrace();  
 wrapper.getTempCallback()  
 .onError(ErrorCode.PARSE\_EXCEPTION.getMsg(), wrapper.getRequest(), wrapper.getAction());  
 }  
  
 } else if (response.getRespEvent() == 20) {//通知  
  
 }  
 }  
 ...

}

#### 授权



我们连接成功后进行授权,授权就是发送一个携带用户信息的请求然后服务端通过这个请求验证用户信息,验证成功后服务端就知道了当前长连接属于哪个用户,由于我们是先通过http请求进行的登录然后在通过websocket进行的授权,既然http登录能成功所以正常情况下通过websocket进行授权不会有问题,所以这里错误回调没有进行处理.

这里我们模拟这个过程,首先是定义请求的action,在action中添加请求对应的action,reqEvent,respClazz这个比较简单我就省略了,然后在连接成功回调进行授权请求doAuth()

*/\*\*  
 \* 登录成功后认证  
 \*/*private void doAuth() {  
 sendReq(Action.*LOGIN*, null, new ICallback() {  
 @Override  
 public void onSuccess(Object o) {  
 Logger.*t*(TAG).d("授权成功");  
 setStatus(WsStatus.AUTH\_SUCCESS);  
 }  
 @Override  
 public void onFail(String msg) {  
  
 }  
 });  
}

private class WsListener extends WebSocketAdapter {  
 @Override  
 public void onConnected(WebSocket websocket, Map<String, List<String>> headers) throws Exception {  
 super.onConnected(websocket, headers);  
 Logger.*t*(TAG).d("连接成功");  
 setStatus(WsStatus.*CONNECT\_SUCCESS*);  
 cancelReconnect();//连接成功的时候取消重连,初始化连接次数  
 doAuth();  
 }

...  
}

#### 心跳和同步

授权成功后接下来是心跳和同步数据,本质跟授权一样也是发送请求.

这里单独说下心跳,心跳其实就是每隔一段时间我们请求服务端然后服务端有响应我们就认为这条连接是稳定的,对于websocket其实已经定义了心跳的格式ping和pong,为了方便这里我就直接使用我们自己定义的协议发送的心跳请求,本质上跟ping和pong是一样的只是协议不同罢了.对于心跳重连策略我这里做的比较简单按一个固定时间执行心跳请求,当心跳连续失败3次就进行重连.

首先还是定义action.这里只是模拟所以respClazz都是null

public enum Action {  
 *LOGIN*("login", 1, null),  
 *HEARTBEAT*("heartbeat", 1, null),  
 *SYNC*("sync", 1, null);

...

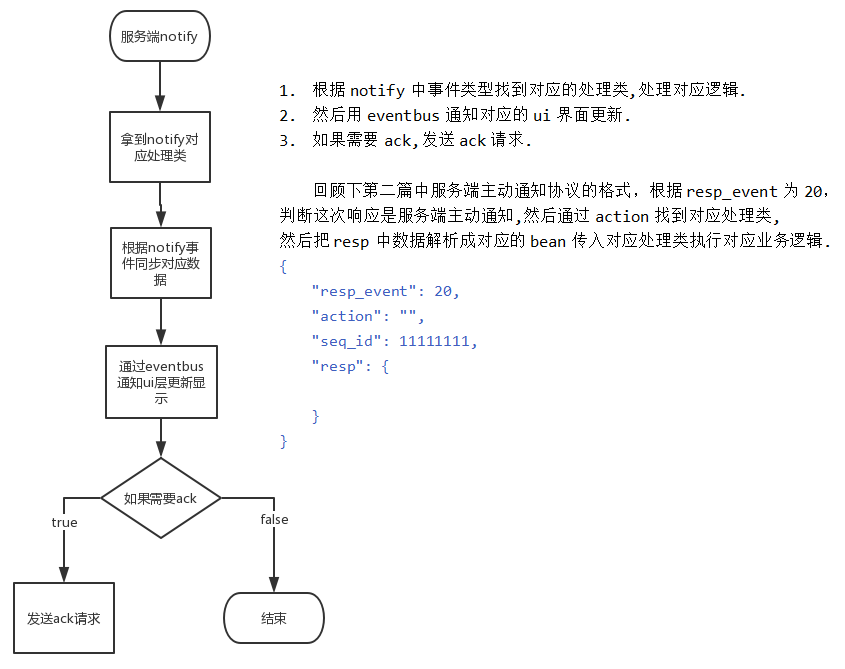
}

心跳和同步

private int heartbeatFailCount = 0;  
private Runnable heartbeatTask = new Runnable() {  
 @Override  
 public void run() {  
 sendReq(Action.*HEARTBEAT*, null, new ICallback() {  
 @Override  
 public void onSuccess(Object o) {  
 heartbeatFailCount = 0;  
 }  
 @Override  
 public void onFail(String msg) {  
 heartbeatFailCount++;  
 if (heartbeatFailCount >= 3) {  
 reconnect();  
 }  
 }  
 });  
 mHandler.postDelayed(this, *HEARTBEAT\_INTERVAL*);  
 }  
};  
private void doAuth() {  
 sendReq(Action.*LOGIN*, null, new ICallback() {  
 @Override  
 public void onSuccess(Object o) {  
 Logger.*t*(TAG).d("授权成功");  
 setStatus(WsStatus.*AUTH\_SUCCESS*);  
 //开始心跳  
 startHeartbeat();  
 //处理同步  
 delaySyncData();  
 }  
 @Override  
 public void onFail(String msg) {  
 }  
 });  
}  
//同步数据  
private void delaySyncData() {  
 mHandler.postDelayed(new Runnable() {  
 @Override  
 public void run() {  
 sendReq(Action.*SYNC*, null, new ICallback() {  
 @Override  
 public void onSuccess(Object o) {  
 }  
 @Override  
 public void onFail(String msg) {  
  
 }  
 });  
 }  
 }, 300);  
}  
private void startHeartbeat() {  
 mHandler.postDelayed(heartbeatTask, *HEARTBEAT\_INTERVAL*);  
}  
private void cancelHeartbeat() {  
 heartbeatFailCount = 0;  
 mHandler.removeCallbacks(heartbeatTask);  
}

授权成功后开始心跳和同步数据,心跳连续失败三次开始重连,在重连时候会取消心跳.当重连成功的时候在连接成功回调会再次进行授权然后授权成功后会再次开启心跳就形成了一个循环.

#### 服务端主动通知



NotifyListenerManager是一个单例的类,在第一次创建的时候在构造方法中执行了regist方法,这是一个变种的观察者模式对于添加观察者这个过程我们直接在regist方法中写好了,如果增加了新的业务逻辑我们只需要在regist方法中put新添加的action与对应处理类.对外暴露的fire方法根据传入的responsse中action找到对应的处理类,拿到处理类对应的注解标记的class,将服务端返回的resp解析成对应的bean丢到对应处理类执行对应逻辑



public class NotifyListenerManager {  
 //region singleton  
 private volatile static NotifyListenerManager *manager*;  
 private NotifyListenerManager() {  
 regist();  
 }  
 public static NotifyListenerManager getInstance() {  
 if (*manager* == null) {  
 synchronized (NotifyListenerManager.class) {  
 if (*manager* == null) {  
 *manager* = new NotifyListenerManager();  
 }  
 }  
 }  
 return *manager*;  
 }  
 //endregion  
 private final String TAG = this.getClass().getSimpleName();  
 private Map<String, INotifyListener> map = new HashMap<>();  
 private void regist() {  
 map.put("notifyAnnounceMsg", new AnnounceMsgListener());  
 }  
 public void fire(Response response) {  
 String action = response.getAction();  
 String resp = response.getResp();  
 INotifyListener listener = map.get(action);  
 if (listener == null) {  
 Logger.*t*(TAG).d("no found notify listener");  
 return;  
 }  
 NotifyClass notifyClass = listener.getClass().getAnnotation(NotifyClass.class);  
 Class<?> clazz = notifyClass.value();  
 Object result = null;  
 try {  
 result = new Gson().fromJson(resp, clazz);  
 } catch (JsonSyntaxException e) {  
 e.printStackTrace();  
 }  
 Logger.*t*(TAG).d(result);  
 listener.fire(result);  
 }  
}

如果新增业务逻辑我们只需要实现新的业务逻辑类,然后在NotifyListenerManager的regist方法中put新增的action与listener映射关系,对外只需要调用NotifyListenerManager.getInstance().fire(response)即可,实现了解耦.

# Jsoup

## 入门

### 解析和遍历一个HTML文档

如何解析一个HTML文档：

String html = "

<html>

<head>

<title>First parse</title>

</head>

<body>

<p>Parsed HTML into a doc.</p>

</body>

</html>

";

Document doc = Jsoup.parse(html);

其解析器能够尽最大可能从你提供的HTML文档来创见一个干净的解析结果，无论HTML的格式是否完整。它可以处理：

* 没有关闭的标签 (比如： <p>Lorem <p>Ipsum parses to <p>Lorem</p> <p>Ipsum</p>)
* 隐式标签 (比如. 它可以自动将 <td>Table data</td>包装成<table><tr><td>?)
* 创建可靠的文档结构（html标签包含head 和 body，在head只出现恰当的元素）

一个文档的对象模型

* 文档由多个Elements和TextNodes组成 (以及其它辅助nodes).
* 其继承结构如下：Document继承Element继承Node. TextNode继承 Node.
* 一个Element包含一个子节点集合，并拥有一个父Element。他们还提供了一个唯一的子元素过滤列表。

## 输入

### 解析一个HTML字符串

问题

来自用户输入，一个文件或一个网站的HTML字符串，你可能需要对它进行解析并取其内容，或校验其格式是否完整，或想修改它

方法

使用静态Jsoup.parse(String html) 方法或 Jsoup.parse(URL url, int timeoutMillis)：

String html = "<html><head><title>First parse</title></head>"

+ "<body><p>Parsed HTML into a doc.</p></body></html>";

Document doc = Jsoup.parse(html);

说明

只要解析的不是空字符串，就能返回一个结构合理的文档，其中包含(至少) 一个head和一个body元素。一旦拥有了一个Document，你就可以使用Document中适当的方法或它父类 Element和Node中的方法来取得相关数据。

### 解析一个body片断

问题

假如你有一个HTML片断 (比如. 一个 div 包含一对 p 标签; 一个不完整的HTML文档) 想对它进行解析。这个HTML片断可以是用户提交的一条评论或在一个CMS页面中编辑body部分。

办法

使用Jsoup.parseBodyFragment(String html)方法.

String html = "<div><p>Lorem ipsum</p>";

Document doc = Jsoup.parseBodyFragment(html);

Element body = doc.body();

说明

parseBodyFragment 方法创建一个空壳的文档，并插入解析过的HTML到body元素中。假如你使用正常的 Jsoup.parse(String html) 方法，通常你也可以得到相同的结果，但是明确将用户输入作为 body片段处理，以确保用户所提供的任何糟糕的HTML都将被补全并解析成body元素。

Document.body() 方法能够取得文档body元素的所有子元素，与 doc.getElementsByTag("body")相同。

保证安全Stay safe

假如你可以让用户输入HTML内容，那么要小心避免跨站脚本攻击。利用基于 Whitelist 的清除器和 clean(String bodyHtml, Whitelist whitelist)方法来清除用户输入的恶意内容。

### 从一个URL加载一个Document

问题

你需要从一个网站获取和解析一个HTML文档，并查找其中的相关数据。

办法

使用 doc.title()方法:

Document doc = Jsoup.connect("http://example.com/").get();

String title = doc.title();

说明

connect(String url) 方法创建一个新的 Connection, 和 get() 取得和解析一个HTML文件。如果从该URL获取HTML时发生错误，便会抛出 IOException，应适当处理。Connection 接口还提供一个方法链来解决特殊请求，具体如下：

Document doc = Jsoup.connect("http://example.com")

.data("query", "Java")

.userAgent("Mozilla")

.cookie("auth", "token")

.timeout(3000)

.post();

这个方法只支持Web URLs (http和https 协议); 假如你需要从一个文件加载，可以使用 parse(File in, String charsetName) 代替。

### 从一个文件加载一个文档

问题

在本机硬盘上有一个HTML文件，需要对它进行解析从中抽取数据或进行修改。

办法

可以使用静态 Jsoup.parse(File in, String charsetName, String baseUri) 方法：

File file = new File("/tmp/input.html");

Document doc = Jsoup.parse(file, "UTF-8", "http://example.com/");

说明

parse(File in, String charsetName, String baseUri) 这个方法用来加载和解析一个HTML文件。如在加载文件的时候发生错误，将抛出IOException，应作适当处理。

baseUri 参数用于解决文件中URLs是相对路径的问题。如果不需要可以传入一个空的字符串。另外还有一个方法parse(File in, String charsetName) ，它使用文件的路径做为 baseUri。 这个方法适用于如果被解析文件位于网站的本地文件系统，且相关链接也指向该文件系统。

## 数据抽取

### 使用DOM方法来遍历一个文档

问题

有一个HTML文档要从中提取数据，并了解这个HTML文档的结构。

方法

将HTML解析成一个Document之后，就可以使用类似于DOM的方法进行操作。示例代码：

File input = new File("/tmp/input.html");

Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Element content = doc.getElementById("content");

Elements links = content.getElementsByTag("a");

for (Element link : links) {

String linkHref = link.attr("href");

String linkText = link.text();

}

说明

Elements这个对象提供了一系列类似于DOM的方法来查找元素，抽取并处理其中的数据。具体如下：

查找元素

* getElementById(String id)
* getElementsByTag(String tag)
* getElementsByClass(String className)
* getElementsByAttribute(String key) (and related methods)
* Element siblings: siblingElements(), firstElementSibling(), lastElementSibling(); nextElementSibling(), previousElementSibling()
* Graph: parent(), children(), child(int index)

元素数据

* attr(String key)获取属性
* attr(String key, String value)设置属性
* attributes()获取所有属性
* id(), className() and classNames()
* text()获取文本内容
* text(String value) 设置文本内容
* html()获取元素内HTML
* html(String value)设置元素内的HTML内容
* outerHtml()获取元素外HTML内容
* data()获取数据内容（例如：script和style标签)
* tag() and tagName()

操作HTML和文本

* append(String html), prepend(String html)
* appendText(String text), prependText(String text)
* appendElement(String tagName), prependElement(String tagName)
* html(String value)

### 使用选择器语法来查找元素

问题

想使用类似于CSS或jQuery的语法来查找和操作元素。

方法

可以使用Element.select(String selector) 和 Elements.select(String selector) 方法实现：

File input = new File("/tmp/input.html");

Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Elements links = doc.select("a[href]"); //带有href属性的a元素

Elements pngs = doc.select("img[src$=.png]"); //扩展名为.png的图片

Elements resultLinks = doc.select("h3.r > a"); //在h3元素之后的a元素

Element masthead = doc.select("div.masthead").first(); //class等于masthead的div标签

说明

jsoup elements对象支持类似于CSS (或jquery)的选择器语法，来实现非常强大和灵活的查找功能。这个select 方法在Document, Element,或Elements对象中都可以使用。且是上下文相关的，因此可实现指定元素的过滤，或者链式选择访问。

Select方法将返回一个Elements集合，并提供一组方法来抽取和处理结果

Selector选择器概述

* tagname: 通过标签查找元素，比如：a
* ns|tag: 通过标签在命名空间查找元素，比如：可以用 fb|name 语法来查找 <fb:name> 元素
* #id: 通过ID查找元素，比如：#logo
* .class: 通过class名称查找元素，比如：.masthead
* [attribute]: 利用属性查找元素，比如：[href]
* [^attr]: 利用属性名前缀来查找元素，比如：可以用[^data-] 来查找带有HTML5 Dataset属性的元素
* [attr=value]: 利用属性值来查找元素，比如：[width=500]
* [attr^=value], [attr$=value], [attr\*=value]: 利用匹配属性值开头、结尾或包含属性值来查找元素，比如：[href\*=/path/]
* [attr~=regex]: 利用属性值匹配正则表达式来查找元素，比如： img[src~=(?i)\.(png|jpe?g)]
* \*: 这个符号将匹配所有元素

Selector选择器组合使用

* el#id: 元素+ID，比如： div#logo
* el.class: 元素+class，比如： div.masthead
* el[attr]: 元素+class，比如： a[href]
* 任意组合，比如：a[href].highlight
* ancestor child: 查找某个元素下子元素，比如：可以用.body p 查找在"body"元素下的所有 p元素
* parent > child: 查找某个父元素下的直接子元素，比如：可以用div.content > p 查找 p 元素，也可以用body > \* 查找body标签下所有直接子元素
* siblingA + siblingB: 查找在A元素之前第一个同级元素B，比如：div.head + div
* siblingA ~ siblingX: 查找A元素之前的同级X元素，比如：h1 ~ p
* el, el, el:多个选择器组合，查找匹配任一选择器的唯一元素，例如：div.masthead, div.logo

伪选择器selectors

* :lt(n): 查找哪些元素的同级索引值（它的位置在DOM树中是相对于它的父节点）小于n，比如：td:lt(3) 表示小于三列的元素
* :gt(n):查找哪些元素的同级索引值大于n，比如： div p:gt(2)表示哪些div中有包含2个以上的p元素
* :eq(n): 查找哪些元素的同级索引值与n相等，比如：form input:eq(1)表示包含一个input标签的Form元素
* :has(seletor): 查找匹配选择器包含元素的元素，比如：div:has(p)表示哪些div包含了p元素
* :not(selector): 查找与选择器不匹配的元素，比如： div:not(.logo) 表示不包含 class=logo 元素的所有 div 列表
* :contains(text): 查找包含给定文本的元素，搜索不区分大不写，比如： p:contains(jsoup)
* :containsOwn(text): 查找直接包含给定文本的元素
* :matches(regex): 查找哪些元素的文本匹配指定的正则表达式，比如：div:matches((?i)login)
* :matchesOwn(regex): 查找自身包含文本匹配指定正则表达式的元素
* 注意：上述伪选择器索引是从0开始的，也就是说第一个元素索引值为0，第二个元素index为1等

### 从元素抽取属性，文本和HTML

问题

在解析获得一个Document实例对象，并查找到一些元素之后，希望取得在这些元素中的数据。

方法

* 要取得一个属性的值，可以使用Node.attr(String key) 方法
* 对于一个元素中的文本，可以使用Element.text()方法
* 对于要取得元素或属性中的HTML内容，可以使用Element.html(), 或 Node.outerHtml()方法

String html = "

<p>An

<a href='http://example.com/'>

<b>example</b>

</a>

link.</p>";

Document doc = Jsoup.parse(html);//解析HTML字符串返回一个Document实现

String text = doc.body().text(); // "An example link"//取得字符串中的文本

Element link = doc.select("a").first();//查找第一个a元素

String linkHref = link.attr("href"); // "http://example.com/"//取得链接地址

String linkText = link.text(); // "example""//取得链接地址中的文本

String linkOuterH = link.outerHtml(); // "<a href="http://example.com"><b>example</b></a>"

String linkInnerH = link.html(); // "<b>example</b>"//取得链接内的html内容

说明

上述方法是元素数据访问的核心办法。此外还其它一些方法可以使用：

Element.id()

Element.tagName()

Element.className() and Element.hasClass(String className)

这些访问器方法都有相应的setter方法来更改数据.

### 处理相对路径URLs

问题

有一个包含相对URLs路径的HTML文档，需要将这些相对路径转换成绝对路径的URLs。

方法

* 在解析文档时确保有指定base URI:parse(URL url, int timeoutMillis)或connect(String url)
* 使用 abs: 属性前缀来取得包含base URI的绝对路径

Document doc = Jsoup.connect("http://www.open-open.com/fasdjlkfjl").get();

Element link = doc.select("a").first();

String relHref = link.attr("href"); // "/download"

String absHref = link.attr("abs:href"); // "http://www.open-open.com/download"

说明

在HTML元素中，URLs经常写成相对于文档位置的相对路径： <a href="/download">...</a>. 当你使用 Node.attr(String key) 方法来取得a元素的href属性时，它将直接返回在HTML源码中指定定的值。

假如你需要取得一个绝对路径，需要在属性名前加 abs: 前缀。这样就可以返回包含根路径的URL地址attr("abs:href")，因此，在解析HTML文档时，定义base URI非常重要。如果你不想使用abs: 前缀，还有一个方法能够实现同样的功能 Node.absUrl(String key)

## 数据修改

### 设置属性的值

问题

在你解析一个Document之后可能想修改其中的某些属性值，然后再保存到磁盘或都输出到前台页面。

方法

可以使用属性设置方法 Element.attr(String key, String value), 和 Elements.attr(String key, String value).

假如你需要修改一个元素的 class 属性，可以使用 Element.addClass(String className) 和 Element.removeClass(String className) 方法。

Elements 提供了批量操作元素属性和class的方法，比如：要为div中的每一个a元素都添加一个 rel="nofollow" 可以使用doc.select("div.comments a").attr("rel", "nofollow");方法：

说明

与Element中的其它方法一样，attr 方法也是返回当 Element (或在使用选择器是返回 Elements 集合)。这样能够很方便使用方法连用的书写方式。比如：

doc.select("div.masthead").attr("title", "jsoup").addClass("round-box");

### 设置一个元素的HTML内容

问题

你需要一个元素中的HTML内容

方法

可以使用Element中的HTML设置方法具体如下：

Element div = doc.select("div").first(); // <div></div>

div.html("<p>lorem ipsum</p>"); // <div><p>lorem ipsum</p></div>

div.prepend("<p>First</p>");//在div前添加html内容

div.append("<p>Last</p>");//在div之后添加html内容

// 添完后的结果: <div><p>First</p><p>lorem ipsum</p><p>Last</p></div>

Element span = doc.select("span").first(); // <span>One</span>

span.wrap("<li><a href='http://example.com/'></a></li>");

// 添完后的结果: <li><a href="http://example.com"><span>One</span></a></li>

说明

* Element.html(String html) 这个方法将先清除元素中的HTML内容，然后用传入的HTML代替。
* Element.prepend(String first) 和 Element.append(String last) 方法用于在分别在元素内部HTML的前面和后面添加HTML内容
* Element.wrap(String around) 对元素包裹一个外部HTML内容。

### 设置元素的文本内容

问题

你需要修改一个HTML文档中的文本内容

方法

可以使用Element的设置方法：:

Element div = doc.select("div").first(); // <div></div>

div.text("five > four"); // <div>five &gt; four</div>

div.prepend("First ");

div.append(" Last"); // now: <div>First five &gt; four Last</div>

说明

文本设置方法与 HTML setter 方法一样：

* Element.text(String text) 将清除一个元素中的内部HTML内容，然后提供的文本进行代替
* Element.prepend(String first) 和 Element.append(String last) 将分别在元素的内部html前后添加文本节点。
* 对于传入的文本如果含有像 <, > 等这样的字符，将以文本处理，而非HTML。

## HTML清理

### 消除不受信任的HTML (来防止XSS攻击)

问题

在做网站的时候，经常会提供用户评论的功能。有些不怀好意的用户，会搞一些脚本到评论内容中，而这些脚本可能会破坏整个页面的行为，更严重的是获取一些机要信息，此时需要清理该HTML，以避免跨站脚本cross-site scripting攻击（XSS）。

方法

使用jsoup HTML Cleaner 方法进行清除，但需要指定一个可配置的 Whitelist（白名单）。

String unsafe = "<p><a href='http://example.com/' onclick='stealCookies()'>Link</a></p>";

String safe = Jsoup.clean(unsafe, Whitelist.basic());

// now: <p><a href="http://example.com/" rel="nofollow">Link</a></p>

说明

XSS又叫CSS (Cross Site Script) ，跨站脚本攻击。它指的是恶意攻击者往Web页面里插入恶意html代码，当用户浏览该页之时，嵌入其中Web里面的html代码会被执行，从而达到恶意攻击用户的特殊目的。XSS属于被动式的攻击，因为其被动且不好利用，所以许多人常忽略其危害性。所以我们经常只让用户输入纯文本的内容，但这样用户体验就比较差了。jsoup提供了一系列的Whitelist基本配置，能够满足大多数要求

## 实例

像qq聊天那样解析网址成title、content、icon

public class ParseUrlActivity extends BaseActivity {  
 private EditText edt;  
 private Button btn;  
 private ImageView iv;  
 private TextView tv\_title;  
 private TextView tv\_content;  
 private LinearLayout ll\_container;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_web\_go*);  
 initView();  
 }  
  
 private void initView() {  
 edt = findViewById(R.id.*edt*);  
 btn = findViewById(R.id.*btn*);  
 iv = findViewById(R.id.*iv*);  
 tv\_title = findViewById(R.id.*tv\_title*);  
 tv\_content = findViewById(R.id.*tv\_content*);  
 ll\_container = findViewById(R.id.*ll\_container*);  
  
 btn.setOnClickListener(v -> {  
 String urlStr = edt.getText().toString().trim();  
 if (TextUtils.*isEmpty*(urlStr)) {  
 Toast.*makeText*(mActivity, "请输入网址！", Toast.*LENGTH\_SHORT*).show();  
 return;  
 }  
 if (!urlStr.startsWith("http")) {  
 Toast.*makeText*(mActivity, "请检查网址是否输入正确！", Toast.*LENGTH\_SHORT*).show();  
 return;  
 }  
 new ParseUrlAsyncTask().execute(urlStr);  
 });  
 }  
  
 private HashMap<String, String> parseUrl(String urlStr) {  
 HashMap<String, String> result = new HashMap<>();  
 try {  
 //这里开始是做一个解析，需要在非UI线程进行，超时设为5000毫秒  
 Document doc = Jsoup.*parse*(new URL(urlStr), 5000);  
 //获取title  
 String title = doc.title();  
 //获取description  
 String description = "";  
 Elements meta\_elements = doc.head().select("meta");  
 for (Element meta\_e : meta\_elements) {  
 if ("description".equalsIgnoreCase(meta\_e.attr("name"))) {  
 description = meta\_e.attr("content");  
 break;  
 } else if ("description".equalsIgnoreCase(meta\_e.attr("itemprop"))) {  
 description = meta\_e.attr("content");  
 break;  
 }  
 }  
 //获取图片  
 String imgStr = "";  
 Elements png\_eles;  
 String reg = ".\*\\.(jpg|png|ico)"; //取href标签以png/ico/jpg结尾的链接;  
 if ((png\_eles = doc.getElementsByAttributeValueMatching("href", reg)) != null && !png\_eles.isEmpty()) {  
 for (Element png\_ele : png\_eles) {  
 if ("shortcut icon".equalsIgnoreCase(png\_ele.attr("rel"))) {  
 imgStr = png\_ele.attr("abs:href");  
 break;  
 }  
 }  
 } else if ((png\_eles = doc.getElementsByAttributeValueMatching("content", reg)) != null && !png\_eles.isEmpty()) {  
 Element png\_ele = png\_eles.first();  
 imgStr = png\_ele.attr("abs:content");  
 } else if ((png\_eles = doc.getElementsByAttributeValueMatching("src", reg)) != null && !png\_eles.isEmpty()) {  
 Element png\_ele = png\_eles.first();  
 imgStr = png\_ele.attr("abs:src");  
 }  
 //放入map  
 result.put("code", "0");  
 result.put("title", title);  
 result.put("description", description);  
 result.put("url", urlStr);  
 result.put("img", imgStr);  
 } catch (IOException e) {  
 result.put("code", "1");  
 e.printStackTrace();  
 }  
 return result;  
 }  
  
 public static void start(Context context) {  
 Intent starter = new Intent(context, ParseUrlActivity.class);  
 context.startActivity(starter);  
 }  
  
 private class ParseUrlAsyncTask extends AsyncTask<String, Void, HashMap<String, String>> {  
  
 @Override  
 protected HashMap<String, String> doInBackground(String... strings) {  
 return parseUrl(strings[0]);  
 }  
  
 @Override  
 protected void onPostExecute(HashMap<String, String> map) {  
 if (map.get("code").equals("0")) {  
 String title = map.get("title");  
 String description = map.get("description");  
 String url = map.get("url");  
 String img = map.get("img");  
 if (!TextUtils.*isEmpty*(img)) {  
 if (!img.startsWith("http")) {  
 img = "http:" + img;  
 }  
 }  
  
 tv\_title.setText(title);  
 tv\_content.setText(description);  
 Glide.*with*(mActivity).load(img).placeholder(R.mipmap.*ic\_launcher*).into(iv);  
 ll\_container.setOnClickListener(v -> {  
 SimpleWebViewActivity.*Companion*.start(mActivity, url);  
 });  
 } else {  
 Toast.*makeText*(mActivity, "解析错误", Toast.*LENGTH\_SHORT*).show();  
 }  
 }  
 }  
}

# XPosed

Xposed并不是什么新东西了，好几年前就有了，其实Xposed并没有想象中那么复杂，原理和相关的API都很 简单，难的是逆向，怎么去实现你要Hook的功能，反编译，调试输出，堆栈 跟踪，抓包等等，在这个过程中你需要去分析很多很多东西，猜测调试，有时候 折腾几天可能毫无进展，不过也会收获更多，比如你自己开发APP的时候也会 慢慢开始考虑安全相关的东西

## Xposed是什么

一个很牛逼的框架，可以在不修改APK的情况下影响程序的运行，比如： 直接把APP的界面改成自己想要的样子，去掉界面里不喜欢的东西， 自动抢红包，消息防撤回，步数修改等等；简直酷得不行，网上有很多插件作者开发出来的优秀插件，不过呢，因为Xposed拥有最高权限，如果不法分子 在插件里植入了恶意代码，比如登录劫持，偷偷采集你的账号密码发送到他们的手里，如果涉及到了金钱，就很恐怖啦，所以在使用Xposed插件的时候， 尽量选那些开源的，并进行代码review，看是否存在恶意代码，再进行安装体验

## 大概简述下Xposed的原理

Android基于Linux，第一个启动的进程自然是init进程，该进程会启动所有Android进程的父进程——Zygote(孵化)进程，该进程的启动配置在/init.rc脚本中，而Zygote进程对应的执行文件是/system/bin/app\_process，该文件完成类库的加载以及一些函数的调用工作。在Zygote进程创建后，再fork出SystemServer进程和其他进程。

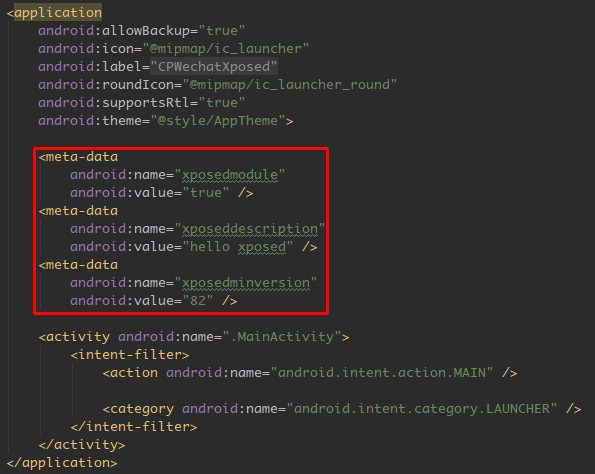
而Xposed Framework呢，就是用自己实现的app\_process替换掉了系统原本提供的app\_process，加载一个额外的jar包，然后入口从原来的：com.android.internal.osZygoteInit.main()被替换成了：de.robv.android.xposed.XposedBridge.main()，然后创建的Zygote进程就变成Hook的Zygote进程了，而后面Fork出来的进程也是被Hook过的。这个Jar包在： /data/data/de.rbov.android.xposed.installer/bin/XposedBridge.jar

## 相关文档：

* 官网：http://repo.xposed.info/
* 作者Github仓库：https://github.com/rovo89
* 官方教程：https://github.com/rovo89/XposedBridge/wiki/Development-tutorial
* XposedBridge.jar下载：https://jcenter.bintray.com/de/robv/android/xposed/api/
* Xposed Installer，由于Android 5.0以上采用ART，而5.0以下默认采用Dalvik， 所以是有两个版本的Xposed：
* Android 4.0.3-4.4：http://repo.xposed.info/module/de.robv.android.xposed.installer
* Android 5.0以上：：https://forum.xda-developers.com/showthread.php?t=3034811

## 创建一个Xposed工程

Step 1：新建一个工程，然后修改下AndroidManifest.xml，增加下面的代码：



Step 2：build.gradle文件添加库依赖

IMG_256

别手贱改成compile，装上打开后会报错的！应该使用provided，只提供编译支持，不会写到apk里

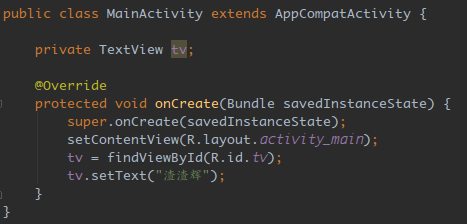
Step 3：res/asserts文件夹创建一个xposed\_init文件

XposedBridge会从assets 目录中的xposed\_init文件中获取入口点，比如我的：

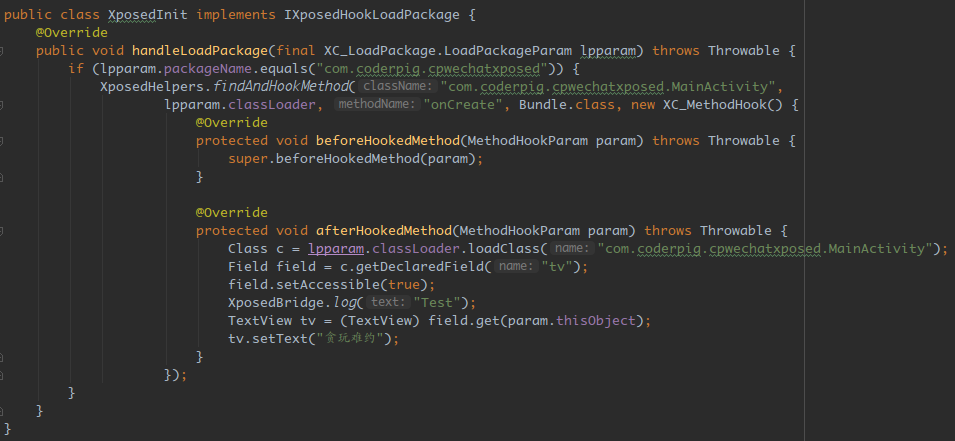
IMG_256

Step 4：编写我们的入口点类

在此之前我们先修改下我们的MainActivity.java，修改下TextView显示的文字：



代码很简单，就是设置成”渣渣辉”而已，接着编写我们的Xposed入口类：XposedInit.java



继承了IXposedHookLoadPackage接口，重写了handleLoadPackage方法， 判断了下包名，如果是的，XposedHelpers.findAndHookMethod()， hook掉onCreate()方法，XC\_MethodHook()重写afterHookedMethod， 当onCreate()执行后会回调这个方法，在这里获得TextView对象， 把文字修改成”贪玩难约”，接着运行，安装后需要重启设备。重启后，打开应用，修改生效了，log也能看到打印出来的日志：

IMG_256

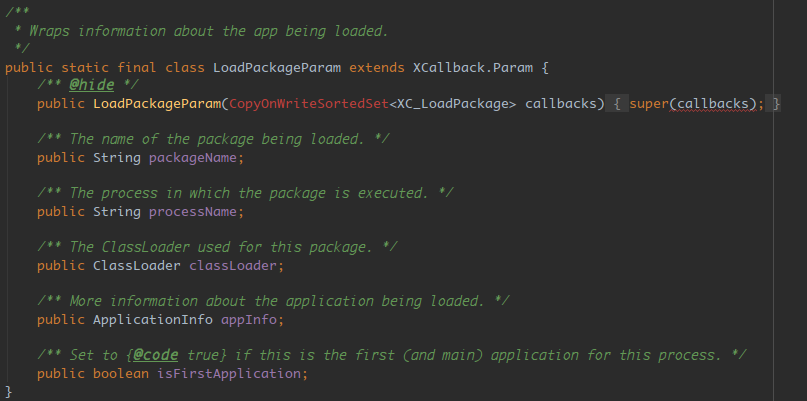
注意事项：在运行Xposed之前，记得把InstallRun的钩钩去掉哦！每次运行都需要重启手机哈

## Xposed API相关介绍

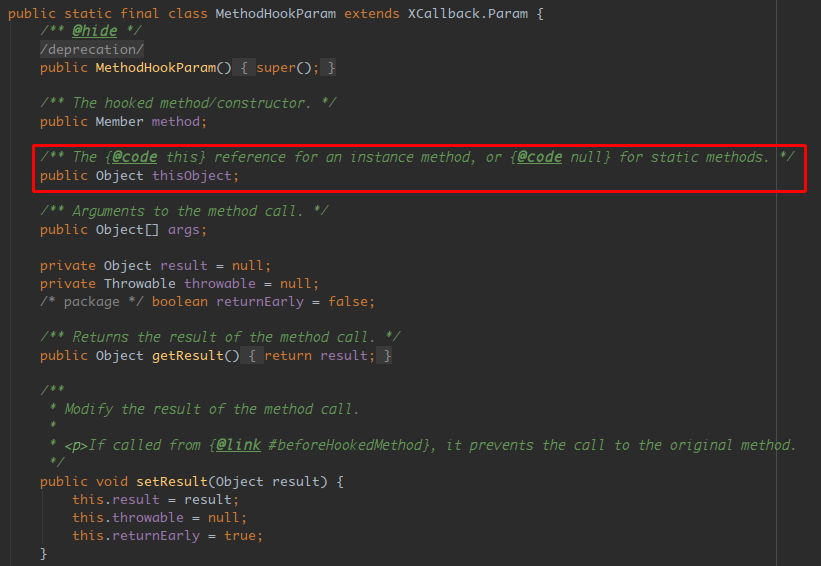
网上没有什么好的Xposed API的文档，如果英语好的，建议直接阅读源码注释！

### 例子里用到的姿势点

* IXposedHookLoadPackage接口：App被加载的时候调用，用于App应用的Hook回调方法是：handleLoadPackage(final XC\_LoadPackage.LoadPackageParam lpparam)
* XC\_LoadPackage.LoadPackageParam：包含与正在加载的应用程序的有关信息。



* XposedHelpers.findAndHookMethod(要Hook的类，classLoader，方法名，参数，回调对象) Hook一个方法的时候使用
* XC\_MethodHook()，回调对象，需重写两个方法
* beforeHookedMethod(MethodHookParam param)：方法调用前执行
* afterHookedMethod(MethodHookParam param) 方法调用后执行
* 注：可以调用param.setResult()设置方法的返回值！
* MethodHookParam：包含与调用方法有关的信息



比较关注的是这个thisObject，代表调用该方法的对象实例，如果是静态方法的话，返回一个Null，比如这里调用onCreate()方法的是MainActivity，获得的自然是MainActivity实例。

接着是获取成员变量，分为私有与非私有变量，非私有直接调用下述方法 即可获得class

Class c = lpparam.classLoader.loadClass("com.coderpig.cpwechatxposed.MainActivity");

Field field = c.getField("tv");

如果是私有，则需要先设置访问权限(setAccessible)

Class c = lpparam.classLoader.loadClass("com.coderpig.cpwechatxposed.MainActivity");

Field field = c.getDeclaredField("tv");

field.setAccessible(true);

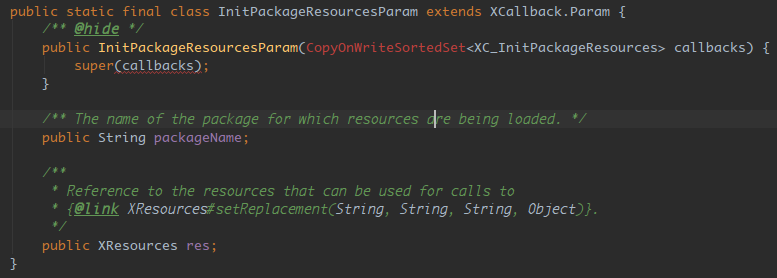
接着调用获得该对象

TextView tv = (TextView) field.get(param.thisObject);

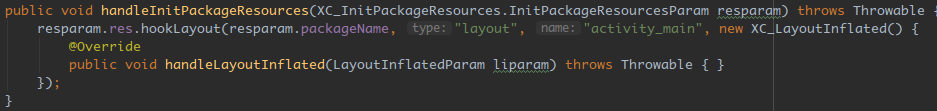
tv.setText("贪玩难约");

### 补充姿势点

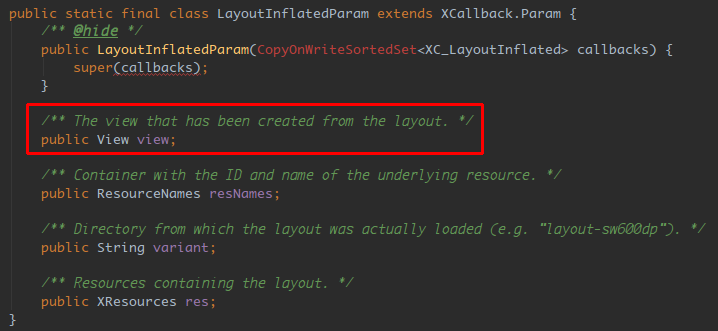
* IXposedHookZygoteInit：在Zygote启动时调用，用于系统服务的Hook回调方法initZygote()
* IXposedHookInitPackageResources：在资源布局初始化时会回被执行(inflate方法) 回调方法：handleInitPackageResources(XC\_InitPackageResources.InitPackageResourcesParam resparam) ，InitPackageResourcesParam包含两个参数，包名和XResource(资源相关)



有了这个XResource对象，就可以拿到布局资源树了，通过重写hookLayout方法



LayoutInflatedParam里面这个view就是布局资源树了，你可以拿到遍历，拿到某个 特定控件，然后做一些骚操作。



## XposeHelpers提供了一些辅助方法

* callMethod(Object obj,String methodName, Object… args)：在APP中调用特定方法

参数依次是：调用方法的所在类，调用方法名，方法参数

* findClass(String className,ClassLoader classLoader)：获取class类实例

参数依次是类名，类加载器

* findMethodExact：通过反射查找类的成员方法(可setAccessible(true)设置非私有)
* findConstructorExact：通过反射查找构造函数(同样可设置可访问下性)
* findAndHookXXX：查找并Hook
* setXxx：通过反射设置对象数据成员的值
* setStaticXxx：通过反射设置静态变量的值
* XposedBridge.log(“日志内容”)：输入日志和写入到/data/xposed/debug.log

Xposed Installer日志那里可以看到

* 内部类：通过$符号链接内部类
* 只能Hook方法与构造方法，不能Hook接口和抽象方法

# 鲁班Luban压缩

## 导入

implementation 'top.zibin:Luban:1.1.8'

## 方法列表

|  |  |
| --- | --- |
| 方法 | 描述 |
| load | 传入原图 |
| filter | 设置开启压缩条件 |
| ignoreBy | 不压缩的阈值，单位为K |
| setFocusAlpha | 设置是否保留透明通道 |
| setTargetDir | 缓存压缩图片路径 |
| setCompressListener | 压缩回调接口 |
| setRenameListener | 压缩前重命名接口 |

## 异步调用

Luban内部采用IO线程进行图片压缩，外部调用只需设置好结果监听即可：

Luban.with(this)

.load(photos) //传入原图

.ignoreBy(100) //不压缩的阈值，单位为K，默认为100K

.setTargetDir(getPath()) //缓存压缩图片路径

.filter(new CompressionPredicate() { //设置开启压缩条件

@Override

public boolean apply(String path) {

return !(TextUtils.isEmpty(path) || path.toLowerCase().endsWith(".gif"));

}

})

.setCompressListener(new OnCompressListener() { //压缩回调接口

@Override

public void onStart() {

// TODO 压缩开始前调用，可以在方法内启动 loading UI

}

@Override

public void onSuccess(File file) {

// TODO 压缩成功后调用，返回压缩后的图片文件

}

@Override

public void onError(Throwable e) {

// TODO 当压缩过程出现问题时调用

}

}).launch();

## 同步调用

同步方法请尽量避免在主线程调用以免阻塞主线程，下面以rxJava调用为例

Flowable.just(photos)

.observeOn(Schedulers.io())

.map(new Function<List<String>, List<File>>() {

@Override public List<File> apply(@NonNull List<String> list) throws Exception {

// 同步方法直接返回压缩后的文件

return Luban.with(MainActivity.this).load(list).get();

}

})

.observeOn(AndroidSchedulers.mainThread())

.subscribe();