**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. UML类图 6](#_Toc30661)

[1.1. 概念 6](#_Toc6531)

[1.2. 6](#_Toc30150)

[1.3. 类 6](#_Toc28602)

[1.4. 类的UML图示 6](#_Toc7185)

[1.5. 类与类之间的关系（4类9种） 7](#_Toc13603)

[1.5.1. 关联关系 7](#_Toc2461)

[1.5.1.1. 双向关联 7](#_Toc32595)

[1.5.1.2. 单向关联 7](#_Toc3638)

[1.5.1.3. 自关联 8](#_Toc29997)

[1.5.1.4. 多重性关联 8](#_Toc6742)

[1.5.1.5. 聚合关系 8](#_Toc7648)

[1.5.1.6. 组合关系 9](#_Toc30373)

[1.5.2. 依赖关系 9](#_Toc19676)

[1.5.3. 泛化关系（继承关系） 9](#_Toc18579)

[1.5.4. 接口与实现关系 10](#_Toc14990)

[1.6. 实例分析1——登录模块 10](#_Toc21673)

[1.7. 实例分析2——注册模块 11](#_Toc6115)

[1.8. 实例分析3——售票机控制程序 12](#_Toc13446)

[2. 重构列表 14](#_Toc6872)

[2.1. 重构组织函数 14](#_Toc9637)

[2.2. 在对象之间搬移特性 15](#_Toc13479)

[2.3. 重新组织数据 15](#_Toc32598)

[2.4. 简化条件表达式 16](#_Toc26197)

[2.5. 简化函数调用 16](#_Toc4509)

[2.6. 处理概括关系 16](#_Toc3755)

[2.7. 大型重构 17](#_Toc31685)

[3. 坏味道列表 17](#_Toc9429)

[4. Java内存模型 17](#_Toc16631)

[4.1. 共享内存和消息传递 17](#_Toc11595)

[4.2. Java内存模型的抽象 18](#_Toc21009)

[4.3. 从源代码到指令序列的重排序 18](#_Toc25261)

[4.4. happens-before简介 18](#_Toc16259)

[4.4.1. JMM的设计意图 19](#_Toc6830)

[4.4.2. happens-before的定义与规则 19](#_Toc6935)

[4.5. 顺序一致性 19](#_Toc28495)

[4.5.1. 数据竞争与顺序一致性 19](#_Toc4282)

[4.5.2. 顺序一致性模型 19](#_Toc14070)

[4.5.3. 同步程序的顺序一致性 20](#_Toc14014)

[4.5.4. 未同步程序的顺序一致性 21](#_Toc27166)

[5. 方法中参数按值与按引用传递 22](#_Toc22851)

[5.1. 按值传递是什么 22](#_Toc19385)

[5.2. 按引用传递 22](#_Toc24979)

[5.2.1. 按引用传递是什么 22](#_Toc4480)

[5.2.2. 对按引用传递例子的改变 23](#_Toc16828)

[5.3. 总结说明 23](#_Toc29621)

[6. volatile关键字介绍 23](#_Toc19889)

[6.1. 资料一： 23](#_Toc2100)

[6.1.1. 前言 23](#_Toc30390)

[6.1.2. java内存模型与原子性，可见性和有序性 23](#_Toc10943)

[6.1.2.1. 原子性 24](#_Toc17124)

[6.1.2.2. 可见性 24](#_Toc1254)

[6.1.2.3. 有序性 24](#_Toc27137)

[6.1.3. volatile关键字 24](#_Toc6653)

[6.1.3.1. volatile保证可见性吗？ 24](#_Toc2574)

[6.1.3.2. volatile保证原子性吗？ 25](#_Toc8192)

[6.1.3.3. volatile能保证有序性吗？ 25](#_Toc14635)

[6.1.4. 正确使用volatile关键字 25](#_Toc29781)

[6.1.4.1. 状态标志 26](#_Toc10185)

[6.1.4.2. 双重检查模式 （DCL） 26](#_Toc13252)

[6.1.5. 总结 27](#_Toc23875)

[6.2. 资料二： 27](#_Toc30501)

[6.2.1. C/C++中的volatile关键字作用 27](#_Toc10829)

[6.2.2. java中volatile关键字作用 27](#_Toc664)

[6.2.3. 适配 28](#_Toc2018)

[7. 回调函数 28](#_Toc25397)

[7.1. 1.定义接口 28](#_Toc3733)

[7.2. 2.调用接口 28](#_Toc6048)

[7.3. 3.注册接口 28](#_Toc20845)

[8. 内存泄漏 29](#_Toc490)

[8.1. 概述 29](#_Toc189)

[8.2. Activity对象未被回收 29](#_Toc5522)

[8.2.1. 静态变量引用Activity对象 29](#_Toc7316)

[8.2.2. 静态View 29](#_Toc2748)

[8.2.3. 内部类 30](#_Toc31663)

[8.2.4. 匿名类 30](#_Toc26566)

[8.2.5. Handler 30](#_Toc21278)

[8.2.6. Threads和TimerTask 31](#_Toc31977)

[8.2.7. 监听器 31](#_Toc3070)

[8.3. 集合对象造成的泄漏 31](#_Toc2445)

[8.4. 资源对象没关闭造成内存泄漏 31](#_Toc24553)

[8.5. 使用对象池避免频繁创建对象 31](#_Toc16747)

[9. 四种内存引用 32](#_Toc14866)

[10. 内存溢出 32](#_Toc22468)

[10.1.1. 释放强引用 32](#_Toc2604)

[10.1.2. 使用软引用 32](#_Toc10536)

[10.1.3. 使用弱引用 33](#_Toc24454)

[10.2. 图像处理 33](#_Toc8361)

[10.2.1. 在内存中压缩图片 33](#_Toc5632)

[10.2.2. 使用完图片后回收图片所占内存 33](#_Toc7018)

[10.2.3. 降低要显示的图片色彩质量 33](#_Toc27598)

[10.2.4. 查询图片信息时不把图片加载到内存中 33](#_Toc26093)

[11. 图片OOM 34](#_Toc19999)

[11.1. 前言 34](#_Toc2190)

[11.2. 计算Bitmap的内存占用 34](#_Toc23581)

[11.3. 图像后门inPurgeable 36](#_Toc1290)

[11.4. 被堵上的后门你还在用？ 38](#_Toc23275)

[12. 避免创建不必要的对象 39](#_Toc29952)

[12.1. 使用单例 39](#_Toc27269)

[12.2. 避免进行隐式装箱 40](#_Toc14806)

[12.3. 谨慎选用容器 40](#_Toc2738)

[12.4. 用好LaunchMode 40](#_Toc12391)

[12.5. Activity处理onConfigurationChanged 41](#_Toc14433)

[12.6. 注意字符串拼接 41](#_Toc12120)

[12.7. 减少布局层级 41](#_Toc28464)

[12.8. 提前检查，减少不必要的异常 41](#_Toc3356)

[12.9. 不要过多创建线程 42](#_Toc30704)

[12.10. 使用注解替代枚举 43](#_Toc4827)

[12.11. 选用对象池 43](#_Toc11233)

[12.12. 谨慎初始化Application 43](#_Toc22850)

[13. 在谈MVP之前，你真的懂MVC吗 44](#_Toc23874)

[13.1. 什么才是MVC？ 44](#_Toc21577)

[13.2. 经典MVC模式 44](#_Toc12240)

[13.3. 现代MVC模式 45](#_Toc15960)

[14. MVP模式是你的救命稻草吗？ 45](#_Toc18075)

[14.1. 什么是MVP 45](#_Toc3246)

[14.2. MVP到底是在解决什么问题的： 45](#_Toc28305)

[14.3. Talk is cheap, show me the code 46](#_Toc31367)

[15. 使用MVP模式重构代码 48](#_Toc6135)

[15.1. 一般的MVC里的 Controller 需要做的事情： 48](#_Toc14629)

[15.2. Activity作为一个 Controller 到底都负责干什么了 48](#_Toc28880)

[15.3. Presenter是来给activity减负的吗？ 48](#_Toc8477)

[15.4. 一个LoginActivity的重构 49](#_Toc20344)

[15.5. 从1个类变为3个类 50](#_Toc30483)

[15.6. 缺点 51](#_Toc22254)

[15.7. 那么为什么要用MVP？ 51](#_Toc10055)

[16. MVC模式 51](#_Toc19115)

[16.1. MVC (Model-View-Controller)： 51](#_Toc13989)

[16.2. MVC好处： 51](#_Toc30199)

[16.3. 什么时候适合使用MVC设计模式？ 52](#_Toc23858)

[16.4. 在MVC模式中我们发现 52](#_Toc27416)

[16.5. 在Android中： 52](#_Toc7441)

[16.6. MVC for Android 52](#_Toc16597)

[17. MVP模式 54](#_Toc625)

[17.1. MVP 54](#_Toc26904)

[17.2. MVC和MVP的区别？ 54](#_Toc23583)

[17.3. MVP的优点 55](#_Toc20931)

[17.4. MVP的缺点 55](#_Toc31135)

[18. 这个VP（注意是VP）泛型有点懵 55](#_Toc18663)

[18.1. View 55](#_Toc8937)

[18.2. Presenter 56](#_Toc10244)

[19. 如何向妻子解释OOD 57](#_Toc24389)

[19.1. OOD简介 57](#_Toc9722)

[19.2. 为什么要OOD？ 58](#_Toc3038)

[19.2.1. 单一职责原则 58](#_Toc28116)

[19.2.2. 开放闭合原则 60](#_Toc2045)

[19.2.3. Liskov's 替换原则 61](#_Toc11277)

[19.2.4. 接口分离原则 62](#_Toc23199)

[19.2.5. 依赖倒置原则 63](#_Toc10712)

[19.2.6. 总结 64](#_Toc14738)

[20. 进程保活大全 65](#_Toc10145)

[20.1. 进程的优先级 65](#_Toc1393)

[20.1.1. 前台进程 —— Foreground process 65](#_Toc13566)

[20.1.2. 可见进程 —— Visible process 65](#_Toc18574)

[20.1.3. 服务进程 —— Service process 65](#_Toc28583)

[20.1.4. 后台进程 —— Background process 66](#_Toc15278)

[20.1.5. 空进程 —— Empty process 66](#_Toc8084)

[20.2. Android 进程回收策略 66](#_Toc14372)

[20.3. 提升进程优先级的方案 67](#_Toc3352)

[20.3.1. 利用 Activity 提升权限 67](#_Toc12291)

[20.3.1.1. 方案设计思想 67](#_Toc10467)

[20.3.1.2. 方案适用范围 67](#_Toc1167)

[20.3.1.3. 方案具体实现 67](#_Toc7443)

[20.3.2. 利用 Notification 提升权限 68](#_Toc5532)

[20.3.2.1. 方案设计思想 68](#_Toc22454)

[20.3.2.2. 方案实现挑战 68](#_Toc30841)

[20.3.2.3. 方案挑战应对措施 68](#_Toc25194)

[20.3.2.4. 方案适用范围 68](#_Toc31639)

[20.3.2.5. 方案具体实现 69](#_Toc14339)

[20.4. 进程死后拉活的方案 69](#_Toc1041)

[20.4.1. 利用系统广播拉活 69](#_Toc4192)

[20.4.1.1. 方案设计思想 69](#_Toc31929)

[20.4.1.2. 方案适用范围 69](#_Toc7118)

[20.4.2. 利用第三方应用广播拉活 70](#_Toc2654)

[20.4.2.1. 方案设计思想 70](#_Toc12977)

[20.4.2.2. 方案适用范围 70](#_Toc5984)

[20.4.3. 利用系统Service机制拉活 70](#_Toc25419)

[20.4.3.1. 方案设计思想 70](#_Toc6355)

[20.4.3.2. 方案适用范围 70](#_Toc22227)

[20.4.4. 利用Native进程拉活 70](#_Toc16825)

[20.4.4.1. 方案设计思想 70](#_Toc4163)

[20.4.4.2. 方案实现挑战 70](#_Toc15651)

[20.4.4.2.1. 挑战一：在 Native 进程中如何感知主进程死亡。 70](#_Toc23164)

[20.4.4.2.2. 挑战二：在 Native 进程中如何拉活主进程。 71](#_Toc3635)

[20.4.4.2.3. 挑战三：如何保证 Native 进程的唯一 72](#_Toc10424)

[20.4.4.3. 方案适用范围 72](#_Toc27991)

[20.4.5. 利用 JobScheduler 机制拉活 72](#_Toc868)

[20.4.5.1. 方案设计思想 72](#_Toc29748)

[20.4.5.2. 方案适用范围 73](#_Toc11468)

[20.4.6. 利用账号同步机制拉活 73](#_Toc13853)

[20.4.6.1. 方案设计思想 73](#_Toc17484)

[20.4.6.2. 方案适用范围 74](#_Toc4362)

[20.4.7. 其他有效拉活方案 74](#_Toc9183)

[21. 硬件加速 74](#_Toc9059)

[21.1. 概述 74](#_Toc21311)

[21.2. 硬件加速对App开发的意义 74](#_Toc29988)

[21.3. 页面渲染背景知识 75](#_Toc1272)

[21.4. CPU与GPU结构介绍 75](#_Toc5924)

[21.5. Android中的硬件加速 75](#_Toc29976)

[21.5.1. DisplayList 75](#_Toc20866)

[21.5.2. RenderNode 75](#_Toc32358)

[21.5.3. Android绘制流程（Android 6.0） 76](#_Toc642)

[21.6. 纯软件绘制 VS 硬件加速 78](#_Toc8772)

[21.7. 软件绘制刷新逻辑简介 78](#_Toc6506)

[21.8. 总结 78](#_Toc24743)

[22. Android 权限机制 79](#_Toc13590)

[22.1. Android权限机制 79](#_Toc26825)

[22.1.1. 开始适配和如何兼容 79](#_Toc27741)

[22.1.2. 危险权限与普通权限 79](#_Toc12424)

[22.2. 权限相关API说明 80](#_Toc13956)

[22.2.1. 检查权限是否授予 80](#_Toc12167)

[22.2.2. 申请权限 80](#_Toc23625)

[22.2.3. 权限回调 80](#_Toc9358)

[22.2.4. 权限说明 80](#_Toc24146)

[22.2.5. 获取照相权限举例说明 80](#_Toc16062)

[22.3. QQ音乐的权限适配经验 81](#_Toc25600)

[22.3.1. 不同权限，申请的时机不同 81](#_Toc4501)

[22.3.2. 应用启动授权，需要一个壳 81](#_Toc7674)

[22.4. Android权限机制“乱象” 81](#_Toc12753)

[22.4.1. 读取运动数据权限 82](#_Toc8681)

[22.4.2. 无法添加快捷方式 82](#_Toc2743)

[22.4.3. 消失的桌面歌词，悬浮窗权限 82](#_Toc12605)

[23. FragmentActivity报IllegalStateException错 82](#_Toc13898)

[23.1. 原因 82](#_Toc10997)

[23.2. 解决方案 83](#_Toc14447)

[23.2.1. 关于commit 方法的调用异常处理方法 83](#_Toc10427)

[23.2.2. 针对onbackpress 导致的异常，也有几种解决手段 84](#_Toc15389)

[24. NullPointerException空指针 85](#_Toc2517)

[24.1. 定义 85](#_Toc11006)

[24.2. 简单例子分析 85](#_Toc705)

[24.3. 一连串exceptions的例子 86](#_Toc31537)

[24.4. 具体示例 86](#_Toc15979)

[24.4.1. 方法中对象参数为空导致的NullPointerException 86](#_Toc11278)

[24.4.2. 创建对象数组时候抛出空指针 87](#_Toc18018)

[24.4.3. 在Fragment中调用onCreate方法并找控件造成NullPointerException 88](#_Toc29212)

[24.4.4. 使用RecyclerView造成的NullPointerException 89](#_Toc20862)

[24.4.5. findViewById方法引发的NullPointerException 89](#_Toc21398)

[24.5. 总结 90](#_Toc21705)

[25. NoClassDefFoundError、ClassNotFoundException 90](#_Toc16943)

[25.1. NoClassDefFoundError错误发生的原因 90](#_Toc20657)

[25.2. NoClassDefFoundError和ClassNotFoundException区别 90](#_Toc25597)

[25.3. 常见的NoClassDefFoundError 91](#_Toc25805)

[26. Lambda表达式 91](#_Toc3250)

[26.1. 基本写法 91](#_Toc18102)

[26.2. 自定义接口然后使用Lambda表达式 92](#_Toc12674)

[26.3. Android中使用Lambda 92](#_Toc23750)

[27. 编译型、解释型、动态、静态、动态类型、静态类型、强类型、弱类型，语言概念与区别 92](#_Toc13680)

[27.1. 编译型语言和解释型语言 92](#_Toc959)

[27.1.1. 编译型语言 92](#_Toc6363)

[27.1.2. 解释型语言 93](#_Toc32415)

[27.1.3. 混合型语言 93](#_Toc11895)

[27.2. 动态语言和静态语言 93](#_Toc18997)

[27.2.1. 动态语言 93](#_Toc18716)

[27.2.2. 静态语言 93](#_Toc6701)

[27.2.3. 注意： 93](#_Toc12569)

[27.3. 动态类型语言和静态类型语言 93](#_Toc13229)

[27.3.1. 动态类型语言 93](#_Toc25819)

[27.3.2. 静态类型语言 93](#_Toc14783)

[27.3.3. 注意： 93](#_Toc2102)

[27.4. 强类型语言和弱类型语言 94](#_Toc25112)

[27.4.1. 强类型语言： 94](#_Toc16018)

[27.4.2. 弱类型语言： 94](#_Toc31991)

[27.4.3. 注意： 94](#_Toc20414)

[27.4.4. Java语言的总结 94](#_Toc20793)

[28. 闭包 94](#_Toc13380)

[28.1. 什么是闭包 94](#_Toc5592)

[28.2. 闭包的作用 95](#_Toc19825)

[29. 跨进程通信的四种方式 97](#_Toc23645)

[29.1. 概述 97](#_Toc28260)

[29.2. 方式一：访问其他应用程序的Activity 97](#_Toc7199)

[29.3. 方式二：Content Provider 98](#_Toc18153)

[29.4. 方式三：广播（Broadcast） 98](#_Toc19526)

[29.5. 方式四：AIDL服务 98](#_Toc1522)

[30. AIDL 98](#_Toc10967)

[30.1. 介绍 98](#_Toc30939)

[30.2. 支持的数据类型共 4 种： 99](#_Toc360)

[30.3. 编写主要为以下三部分： 99](#_Toc21991)

[30.4. AIDL 实例 99](#_Toc31784)

[30.4.1. 创建 AIDL 99](#_Toc5018)

[30.4.1.1. 创建要操作的实体类，实现 Parcelable 接口，以便序列化/反序列化 99](#_Toc3672)

[30.4.1.2. 新建 aidl 文件夹，在其中创建接口 aidl 文件以及实体类的映射 aidl 文件 100](#_Toc31699)

[30.4.1.3. Make Project ，生成 Binder 的 Java 文件 100](#_Toc30576)

[30.4.2. 编写服务端代码 101](#_Toc24796)

[30.4.3. 编写客户端代码 102](#_Toc22145)

[30.4.3.1. 实现 ServiceConnection 接口，在其中拿到 AIDL 类 102](#_Toc19397)

[30.4.3.2. 接着绑定服务 102](#_Toc26005)

[30.4.3.3. 拿到 AIDL 类后，就可以调用 AIDL 类中定义好的操作，进行跨进程请求 102](#_Toc30401)

[31. Binder 102](#_Toc1305)

[31.1. IBinder 103](#_Toc15211)

[31.2. Binder 104](#_Toc16281)

[31.2.1. 实现 IBinder 的 transact() 方法： 104](#_Toc28755)

[31.2.2. 接着看看 onTransact() 方法： 105](#_Toc2170)

[31.2.3. 另外一个关键的方法 attachInterface： 106](#_Toc19310)

[31.3. Binder 通信机制 107](#_Toc16837)

[31.3.1. Binder Driver 108](#_Toc15574)

[31.3.2. Service Manager 109](#_Toc7760)

[31.3.3. Binder 机制跨进程通信流程 111](#_Toc31500)

[31.3.3.1. Binder实体 112](#_Toc13770)

[31.3.3.2. Binder 引用 112](#_Toc32473)

[31.3.3.3. 远程服务 112](#_Toc16834)

[31.3.4. Binder 跨进程通讯流程主要为如下 4 步： 113](#_Toc29441)

[31.4. Binder 机制的优点 113](#_Toc2818)

[31.5. 总结 114](#_Toc21449)

[32. AIDL的再次解析 114](#_Toc12938)

[32.1. AIDL文件 114](#_Toc28871)

[32.2. AIDL 生成文件分析 115](#_Toc1157)

[32.2.1. Stub 116](#_Toc1055)

[32.2.2. Proxy 117](#_Toc8770)

[32.2.3. AIDL 生成的内容小结 118](#_Toc18610)

[32.3. AIDL 的使用回顾 119](#_Toc3013)

[32.3.1. 服务端 119](#_Toc13458)

[32.3.2. 客户端 119](#_Toc25738)

[32.3.3. 小结 120](#_Toc5171)

[33. Messenger  121](#_Toc19064)

[33.1. Messenger 简介 121](#_Toc3099)

[33.2. Messenger 的使用 122](#_Toc15691)

[33.2.1. 服务端 123](#_Toc7456)

[33.2.2. 客户端 123](#_Toc25191)

[33.3. 使用小结 125](#_Toc26478)

[33.3.1. 客户端的操作主要有 3 步： 125](#_Toc4151)

[33.3.2. 服务端的操作主要有4步 125](#_Toc2800)

[33.4. 总结 125](#_Toc9599)

[34. 控制反转、依赖注入 125](#_Toc14287)

[34.1. 通过Dagger2来看看他们之间的关系 125](#_Toc5201)

[34.2. 通过《墨攻》电影来看看他们之间的关系 126](#_Toc22569)

[35. 模块化、组件化、插件化 129](#_Toc1402)

[35.1. 单工程模式 129](#_Toc17447)

[35.2. 模块化 129](#_Toc24751)

[35.3. 组件化 129](#_Toc15387)

[35.4. 插件化 129](#_Toc9916)

[36. 如何实现组件化 129](#_Toc87)

[36.1. 为什么要项目组件化 130](#_Toc13950)

[36.2. 如何组件化 131](#_Toc23759)

[36.3. 组件化实施流程 132](#_Toc6779)

[36.3.1. 组件模式和集成模式的转换 132](#_Toc28364)

[36.3.2. AndroidManifest合并问题 132](#_Toc11336)

[36.3.3. 全局Context的获取及组件数据初始化 134](#_Toc23867)

[36.3.4. library依赖问题 135](#_Toc1059)

[36.3.5. 调用和通信 136](#_Toc16468)

[36.3.6. 资源名冲突 136](#_Toc7794)

[36.4. 组件化项目的工程类型 137](#_Toc7095)

[36.4.1. app壳工程 137](#_Toc11769)

[36.4.2. 功能组件和Common组件 138](#_Toc9776)

[36.4.2.1. 功能组件 138](#_Toc2313)

[36.4.2.2. Common功能组件 139](#_Toc13493)

[36.4.3. 业务组件和Main组件 139](#_Toc20734)

[36.4.3.1. 业务组件 139](#_Toc2666)

[36.4.3.2. Main业务组件 140](#_Toc11966)

[36.5. 组件化项目的混淆方案 140](#_Toc7596)

[36.6. 工程的buildgradle和gradleproperties文件 140](#_Toc5142)

[36.6.1. 组件化工程的build.gradle文件 140](#_Toc8809)

[36.6.2. 组件化工程的gradle.properties文件 141](#_Toc12590)

[36.7. 结束语 141](#_Toc27733)

[37. ARouter 141](#_Toc1641)

[37.1. 功能介绍 141](#_Toc4034)

[37.2. 典型应用 142](#_Toc32359)

[37.3. 基础功能 142](#_Toc7706)

[37.3.1. 添加依赖和配置 142](#_Toc6410)

[37.3.2. 添加注解 142](#_Toc849)

[37.3.3. 初始化SDK 143](#_Toc14248)

[37.3.4. 发起路由操作 143](#_Toc29644)

[37.3.5. 添加混淆规则 144](#_Toc23511)

[37.3.6. 使用 Gradle 插件实现路由表的自动加载 (可选) 144](#_Toc5769)

[37.3.7. 使用 IDE 插件导航到目标类 (可选) 145](#_Toc30057)

[37.4. 进阶用法 145](#_Toc13716)

[37.4.1. 坑 - IProviders实现类的坑 145](#_Toc416)

[37.4.2. 通过URL跳转 145](#_Toc20232)

[37.4.3. 解析URL中的参数 146](#_Toc3227)

[37.4.4. 声明拦截器(拦截跳转过程，面向切面编程) 146](#_Toc29484)

[37.4.5. 处理跳转结果 147](#_Toc28555)

[37.4.6. 降级策略 147](#_Toc110)

[37.4.7. 发现服务和Fragment 148](#_Toc14641)

[37.4.8. 通过依赖注入解耦 148](#_Toc5198)

[37.5. 其他api 149](#_Toc23478)

[37.6. 重写跳转URL(!!疑问!!) 150](#_Toc112)

[37.7. 生成路由文档 150](#_Toc20959)

[37.8. 其他 150](#_Toc19821)

[38. ARouter解析 151](#_Toc16159)

[38.1. 名词解释 151](#_Toc24872)

[38.1.1. APT技术 151](#_Toc26914)

[38.1.2. SPI技术 151](#_Toc19620)

[38.2. 框架架构 151](#_Toc7873)

[38.3. 三个主要注解 152](#_Toc9647)

[38.4. 页面注册源码分析 152](#_Toc32640)

# UML类图

## 概念

**UML**

UML(Unified Modeling Language,统一建模语言或[标准建模语言](http://baike.baidu.com/view/3458435.htm" \t "http://baike.baidu.com/_blank))，是始于1997年一个[OMG](http://baike.baidu.com/subview/138917/5088704.htm" \t "http://baike.baidu.com/_blank)标准，它是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持，包括由需求分析到规格，到构造和配置。 面向对象的分析与设计(OOA&D，OOAD)方法的发展在80年代末至90年代中出现了一个高潮，[UML](http://baike.baidu.com/view/23396.htm" \t "http://baike.baidu.com/_blank)是这个高潮的产物。它不仅统一了Booch、Rumbaugh和Jacobson的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的[标准建模语言](http://baike.baidu.com/view/3458435.htm" \t "http://baike.baidu.com/_blank)。

**UML类图**

在UML的静态机制中类图是一个重点，它不但是设计人员关心的核心，更是实现人员关注的核心。建模工具也主要根据类图来产生代码。类图在UML的9个图中占据了一个相当重要的地位。James Rumbaugh对类的定义是：类是具有相似结构、行为和关系的一组对象的描述符。类是面向对象系统中最重要的构造块。类图显示了一组类、接口、协作以及他们之间的关系。在UML中问题域最终要被逐步转化，通过类来建模，通过编程语言构建这些类从而实现系统。类加上他们之间的关系就构成了类图，类图中还可以包含接口、包等元素，也可以包括对象、链等实例

## 

## 类

类(Class)封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称。在系统中，每个类都具有一定的职责，职责指的是类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类一般只有一种职责。在定义类的时候，将类的职责分解成为类的属性和操作（即方法）。类的属性即类的数据职责，类的操作即类的行为职责。设计类是面向对象设计中最重要的组成部分，也是最复杂和最耗时的部分。

在软件系统运行时，类将被实例化成对象(Object)，对象对应于某个具体的事物，是类的实例(Instance)。

类图(Class Diagram)使用出现在系统中的不同类来描述系统的静态结构，它用来描述不同的类以及它们之间的关系。

在系统分析与设计阶段，类通常可以分为三种，实体类(Entity Class)、控制类(Control Class)和边界类(Boundary Class)：

**实体类：**

实体类对应系统需求中的每个实体，它们通常需要保存在永久存储体中，一般使用数据库表或文件来记录，实体类既包括存储和传递数据的类，还包括操作数据的类。实体类来源于需求说明中的名词，如学生、商品等。

**控制类：**

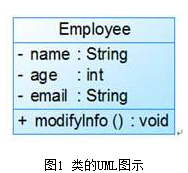
控制类用于体现应用程序的执行逻辑，提供相应的业务操作，将控制类抽象出来可以降低界面和数据库之间的耦合度。控制类一般是由动宾结构的短语（动词+名词）转化来的名词，如增加商品对应有一个商品增加类，注册对应有一个用户注册类等

**边界类：**

边界类用于对外部用户与系统之间的交互对象进行抽象，主要包括界面类，如对话框、窗口、菜单等。

在面向对象分析和设计的初级阶段，通常首先识别出实体类，绘制初始类图，此时的类图也可称为领域模型，包括实体类及其它们之间的相互关系。

## 类的UML图示

在UML类图中，类一般由三部分组成：

(1) 第一部分是类名：每个类都必须有一个名字，类名是一个字符串。

(2) 第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性

UML规定属性的表示方式为： 可见性 名称:类型 [ = 缺省值 ]

“可见性”表示该属性对于类外的元素而言是否可见，包括公有(public) +；私有(private) - ；受保护(protected)#三种

“名称”表示属性名，用一个字符串表示。

“类型”表示属性的数据类型，可以是基本数据类型，也可以是用户自定义类型。

“缺省值”是一个可选项，即属性的初始值。

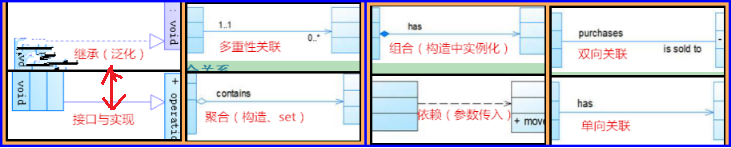
(3) 第三部分是类的操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，是类的成员方法。

UML规定操作的表示方式为： 可见性 名称(参数列表) [ : 返回类型]

“返回类型”是一个可选项，表示方法的返回值类型，依赖于具体的编程语言，可以是基本数据类型，也可以是用户自定义类型，还可以是空类型(void)，如果是构造方法，则无返回类型。

## 类与类之间的关系（4类9种）

在软件系统中，类并不是孤立存在的，类与类之间存在各种关系，对于不同类型的关系，UML提供了不同的表示方式。



### 关联关系

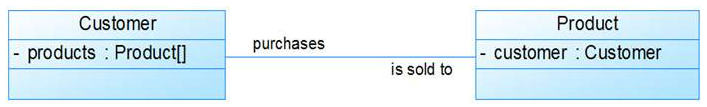
关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系，如汽车和轮胎、师傅和徒弟、班级和学生等等。在UML类图中，用实线连接有关联关系的对象所对应的类，在使用Java、C#和C++等编程语言实现关联关系时，通常将一个类的对象作为另一个类的成员变量。在使用类图表示关联关系时可以在关联线上标注角色名，一般使用一个表示两者之间关系的动词或者名词表示角色名（有时该名词为实例对象名），关系的两端代表两种不同的角色，因此在一个关联关系中可以包含两个角色名，角色名不是必须的，可以根据需要增加，其目的是使类之间的关系更加明确。



**在UML中，关联关系通常又包含如下几种形式：**

#### 双向关联

默认情况下，关联是双向的。例如：顾客(Customer)购买商品(Product)并拥有商品，反之，卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系



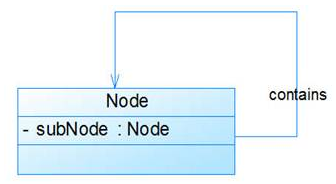
#### 单向关联

类的关联关系也可以是单向的，单向关联用带箭头的实线表示。例如：顾客(Customer)拥有地址(Address)，则Customer类与Address类具有单向关联关系



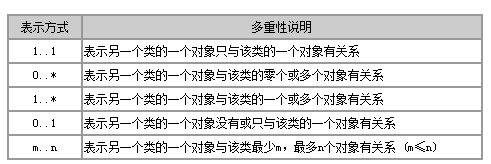
#### 自关联

在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联。例如：一个节点类(Node)的成员又是节点Node类型的对象



#### 多重性关联

多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示。对象之间可以存在多种多重性关联关系，常见的多重性表示方式如表所示：

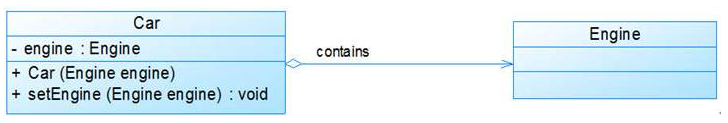


例如：一个界面(Form)可以拥有零个或多个按钮(Button)，但是一个按钮只能属于一个界面，因此，一个Form类的对象可以与零个或多个Button类的对象相关联，但一个Button类的对象只能与一个Form类的对象关联



#### 聚合关系

聚合(Aggregation)关系表示整体与部分的关系。在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在UML中，聚合关系用带空心菱形的直线表示。例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系



在代码实现聚合关系时，成员对象通常作为构造方法、Setter方法或业务方法的参数注入到整体对象中

public class Car {

private Engine engine;

//构造注入

public Car(Engine engine) {

this.engine = engine;

}

//设值注入

public void setEngine(Engine engine) {

this.engine = engine;

}

……

}

public class Engine {

……

}

#### 组合关系

组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。在UML中，组合关系用带实心菱形的直线表示。例如：人的头(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系：



在代码实现组合关系时，通常在整体类的构造方法中直接实例化成员类

public class Head {

private Mouth mouth;

public Head() {

mouth = new Mouth(); //实例化成员类

}

……

}

public class Mouth {

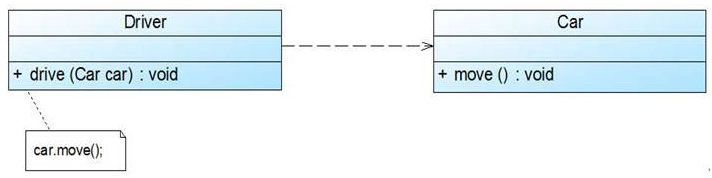
……

}

### 依赖关系

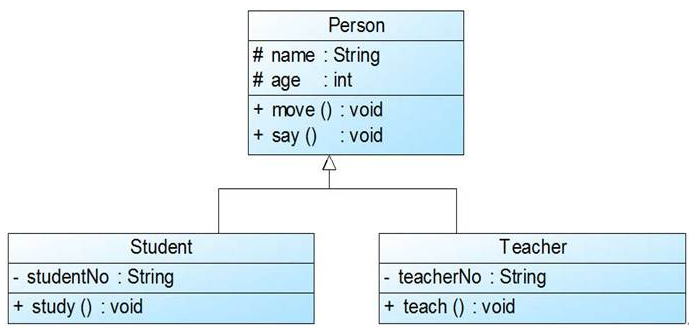
依赖(Dependency)关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。例如：驾驶员开车，在Driver类的drive()方法中将Car类型的对象car作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的drive()方法依赖车的move()方法，因此类Driver依赖类Car。在系统实施阶段，依赖关系通常通过三种方式来实现：

* 第一种也是最常用的一种方式是如图1所示的将一个类的对象作为另一个类中方法的参数；
* 第二种方式是在一个类的方法中将另一个类的对象作为其局部变量；
* 第三种方式是在一个类的方法中调用另一个类的静态方法



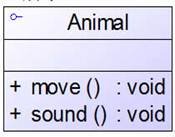
### 泛化关系（继承关系）

泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。在代码实现时，我们使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键字、在C++/C#中使用冒号“：”来实现。例如：Student类和Teacher类都是Person类的子类，Student类和Teacher类继承了Person类的属性和方法，Person类的属性包含姓名(name)和年龄(age)，每一个Student和Teacher也都具有这两个属性，另外Student类增加了属性学号(studentNo)，Teacher类增加了属性教师编号(teacherNo)，Person类的方法包括行走move()和说话say()，Student类和Teacher类继承了这两个方法，而且Student类还新增方法study()，Teacher类还新增方法teach()

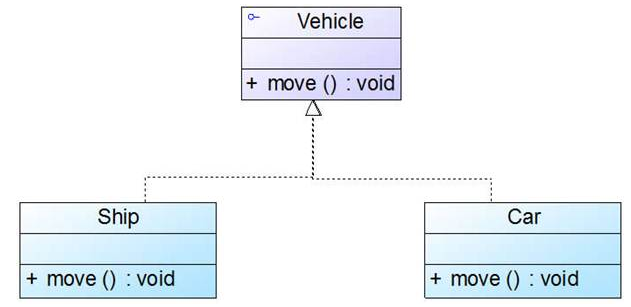


### 接口与实现关系

在很多面向对象语言中都引入了接口的概念，如Java、C#等，在接口中，通常没有属性，而且所有的操作都是抽象的，只有操作的声明，没有操作的实现。UML中用与类的表示法类似的方式表示接口



接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现(Realization)关系，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。例如：定义了一个交通工具接口Vehicle，包含一个抽象操作move()，在类Ship和类Car中都实现了该move()操作，不过具体的实现细节将会不一样



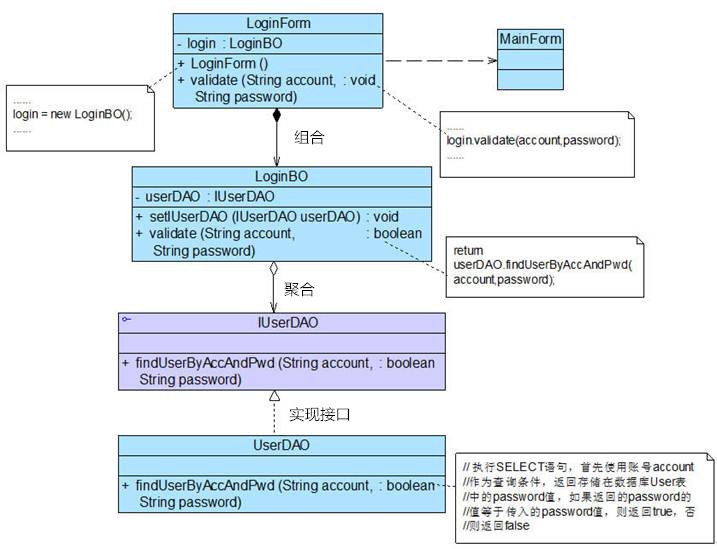
## 实例分析1——登录模块

某基于C/S的即时聊天系统登录模块功能描述如下：用户通过登录界面(LoginForm)输入账号和密码，系统将输入的账号和密码与存储在数据库(User)表中的用户信息进行比较，验证用户输入是否正确，如果输入正确则进入主界面(MainForm)，否则提示“输入错误”。

**根据以上描述绘制初始类图。**

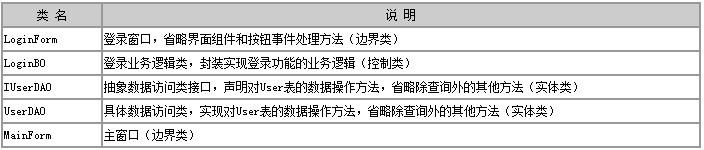
**参考解决方案：**

**参考类图如下：**



考虑到系统扩展性，在本实例中引入了抽象数据访问接口IUserDAO，再将具体数据访问对象注入到业务逻辑对象中，可通过配置文件（如XML文件）等方式来实现，将具体的数据访问类类名存储在配置文件中，如果需要更换新的具体数据访问对象，只需修改配置文件即可，原有程序代码无须做任何修改。

**类说明：**



**方法说明：**



## 实例分析2——注册模块

某基于Java语言的C/S软件需要提供注册功能，该功能简要描述如下：用户通过注册界面(RegisterForm)输入个人信息，用户点击“注册”按钮后将输入的信息通过一个封装用户输入数据的对象(UserDTO)传递给操作数据库的数据访问类，为了提高系统的扩展性，针对不同的数据库可能需要提供不同的数据访问类，因此提供了数据访问类接口，如IUserDAO，每一个具体数据访问类都是某一个数据访问类接口的实现类，如OracleUserDAO就是一个专门用于访问Oracle数据库的数据访问类。

**根据以上描述绘制类图**

为了简化类图，个人信息仅包括账号(userAccount)和密码(userPassword)，且界面类无需涉及界面细节元素。

**参考解决方案：**

在以上功能说明中，可以分析出该系统包括三个类和一个接口，这三个类分别是注册界面类RegisterForm、用户数据传输类UserDTO、Oracle用户数据访问类OracleUserDAO，接口是抽象用户数据访问接口IUserDAO。它们之间的关系如下：

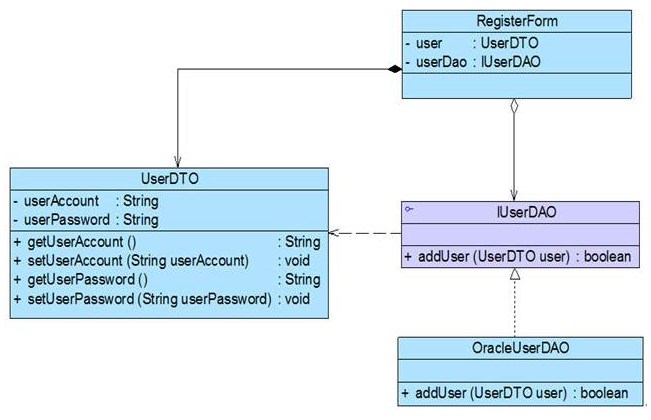
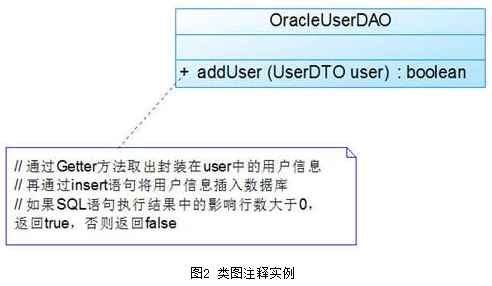
(1) 在RegisterForm中需要使用UserDTO类传输数据且需要使用数据访问类来操作数据库，因此RegisterForm与UserDTO和IUserDAO之间存在关联关系，在RegisterForm中可以直接实例化UserDTO，因此它们之间可以使用组合关联。

(2) 由于数据库类型需要灵活更换，因此在RegisterForm中不能直接实例化IUserDAO的子类，可以针对接口IUserDAO编程，再通过注入的方式传入一个IUserDAO接口的子类对象，因此RegisterForm和IUserDAO之间具有聚合关联关系。

(3) OracleUserDAO是实现了IUserDAO接口的子类，因此它们之间具有类与接口的实现关系。

(4) 在声明IUserDAO接口的增加用户信息方法addUser()时，需要将在界面类中实例化的UserDTO对象作为参数传递进来，然后取出封装在UserDTO对象中的数据插入数据库，因此addUser()方法的函数原型可以定义为：public boolean addUser(UserDTO user)，在IUserDAO的方法addUser()中将UserDTO类型的对象作为参数，故IUserDAO与UserDTO存在依赖关系。

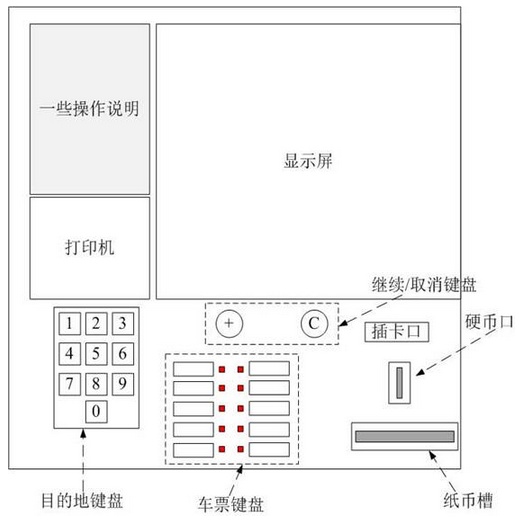
**通过以上分析，该实例参考类图如图所示：**

注意：在绘制类图或其他UML图形时，可以通过注释(Comment)来对图中的符号或元素进行一些附加说明，如果需要详细说明类图中的某一方法的功能或者实现过程，可以使用如图2所示表示方式：

## 实例分析3——售票机控制程序

某运输公司决定为新的售票机开发车票销售的控制软件。图给出了售票机的面板示意图以及相关的控制部件。



**售票机相关部件的作用如下所述：**

(1) 目的地键盘用来输入行程目的地的代码（例如，200表示总站）。

(2) 乘客可以通过车票键盘选择车票种类（单程票、多次往返票和座席种类）。

(3) 继续/取消键盘上的取消按钮用于取消购票过程，继续按钮允许乘客连续购买多张票。

(4) 显示屏显示所有的系统输出和用户提示信息。

(5) 插卡口接受MCard（现金卡），硬币口和纸币槽接受现金。

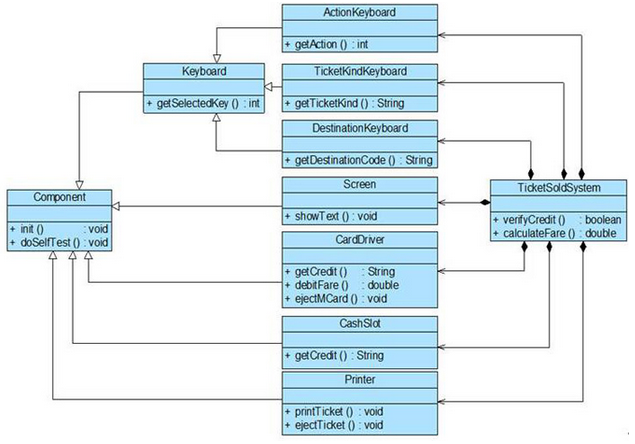
(6) 打印机用于输出车票。

(7) 所有部件均可实现自检并恢复到初始状态。

**现采用面向对象方法开发该系统，使用UML进行建模，绘制该系统的初始类图。**

**参考解决方案：**

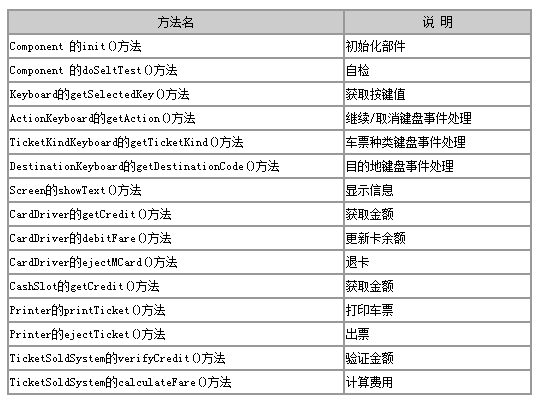
**参考类图如下：**



**类说明：**

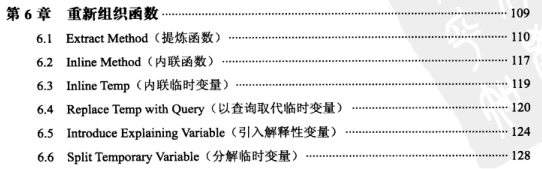


**方法说明：**



# 重构列表

## 重构组织函数



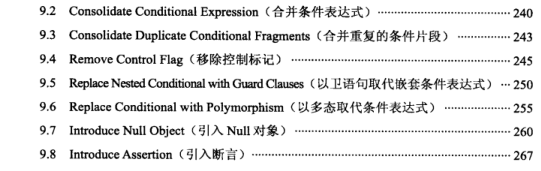
## 在对象之间搬移特性



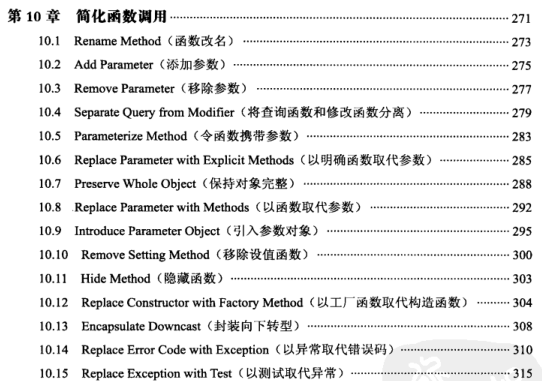
## 重新组织数据



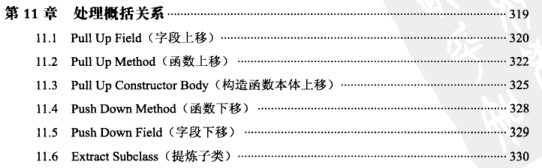
## 简化条件表达式

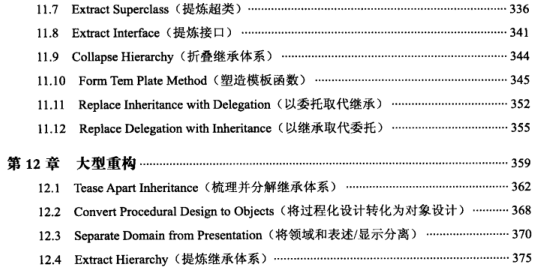


## 简化函数调用

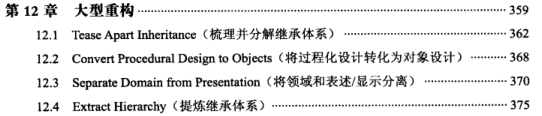


## 处理概括关系

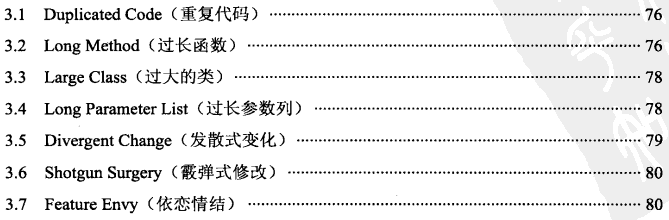


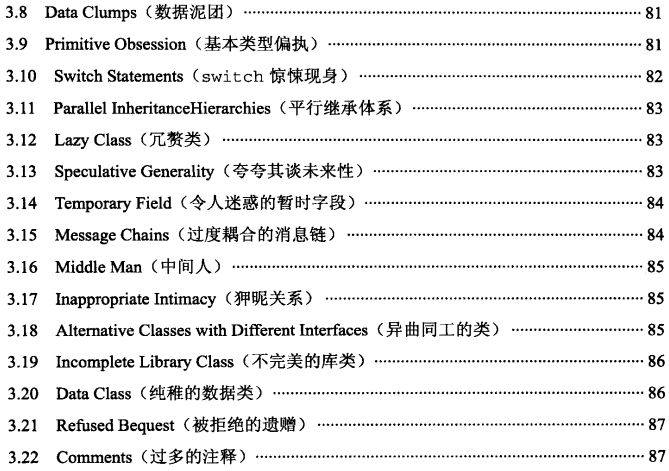


## 大型重构



# 坏味道列表





# Java内存模型

## 共享内存和消息传递

线程之间的通信机制有两种：共享内存和消息传递；在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。

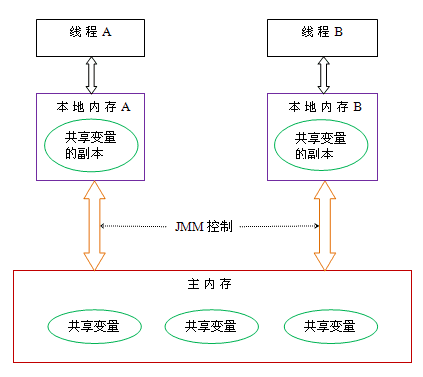
同步是指程序用于控制不同线程之间操作发生相对顺序的机制。在共享内存并发模型里，同步是显式进行的。工程师必须显式指定某个方法或某段代码需要在线程之间互斥执行。在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

Java的并发采用的是共享内存模型，Java线程之间的通信总是隐式进行，整个通信过程对工程师完全透明。

## Java内存模型的抽象

在java中，所有实例域、静态域和数组元素存储在堆内存中，堆内存在线程之间共享（本文使用“共享变量”这个术语代指实例域，静态域和数组元素）。局部变量，方法定义参数和异常处理器参数不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响。

Java线程之间的通信由Java内存模型（本文简称为JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

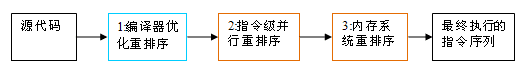
* 线程A把本地内存A中更新过的共享变量刷新到主内存中去。
* 线程B到主内存中去读取线程A之前已更新过的共享变量。

## 从源代码到指令序列的重排序

在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序。重排序分三种类型：

* 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
* 指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
* 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从java源代码到最终实际执行的指令序列，会分别经历下面三种重排序：



上述的1属于编译器重排序，2和3属于处理器重排序。这些重排序都可能会导致多线程程序出现内存可见性问题。对于编译器，JMM的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。对于处理器重排序，JMM的处理器重排序规则会要求java编译器在生成指令序列时，插入特定类型的内存屏障指令，通过内存屏障指令来禁止特定类型的处理器重排序（不是所有的处理器重排序都要禁止）。

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

## happens-before简介

happens-before是JMM最核心的概念，对于Java工程师来说，理解happens-before是理解JMM的关键。

### JMM的设计意图

在设计JMM需要考虑两个关键因素：

* 工程师对内存模型的使用，希望内存模型易于理解和编程，工程师希望基于一个强内存模型来编写代码。
* 编译器和处理器对内存的实现，希望内存模型对他们的束缚越少越好，编译器和处理器希望实现一个弱内存模型。

这两个因素是互相矛盾的，所以JSR-133专家组设计时需要考虑到一个好的平衡点：一方面为工程师提供足够强的内存可见性，另一方面要对编译器和处理器的限制要尽量松些。我们来举了例子：

int a=10; //A

int b=20; //B

int c=a\*b; //C

上面是一个简单的乘法运算，并存在3个happens-before关系：

1、A happens-before B

2、B happens-before C

3、A happens-before C

这三个happens-before关系中，2和3是必须的，但1是不必要的。因此，JMM把happens-before要求禁止的重排序分为两类：

* 会改变程序执行结果的重排序。
* 不会改变程序执行结果的重排序。

JMM对这两种不同性质的重排序，采取了不同的策略：

* 对于会改变程序执行结果的重排序，JMM要求编译器和处理器必须禁止这种重排序。
* 对于不会改变程序执行结果的重排序，JMM要求编译器和处理器不做要求，可以允许这种重排序。

### happens-before的定义与规则

JSR-133使用happens-before的概念来指定两个操作之间的执行顺序，由于这两个操作可以在一个线程内，也可以在不同的线程之间。因此，JMM可以通过happens-before关系向工程师提供跨线程的内存可见性保证。

happens-before规则如下：

* 程序顺序规则：一个线程中的每个操作，happens- before 于该线程中的任意后续操作。
* 监视器锁规则：对一个监视器锁的解锁，happens- before 于随后对这个监视器锁的加锁。
* volatile变量规则：对一个volatile域的写，happens- before 于任意后续对这个volatile域的读。
* 传递性：如果A happens- before B，且B happens- before C，那么A happens- before C。

## 顺序一致性

顺序一致性内存模型是一个理论参考模型，在设计的时候，处理器的内存模型和编程语言的内存模型都会以顺序一致性内存模型为参考。

### 数据竞争与顺序一致性

当程序未正确同步时，就会存在数据竞争。数据竞争指的是：在一个线程中写一个变量，在另一个线程读同一个变量，而且写和读没有通过同步来排序。当代码中包含数据竞争时，程序的执行往往产生违反直觉的结果。如果一个多线程程序能正确同步，这个程序将是一个没有数据竞争的程序。

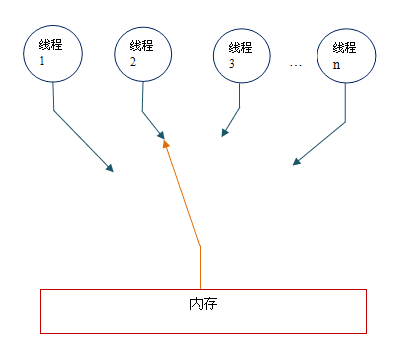
JMM对正确同步的多线程程序的内存一致性做了如下保证：如果程序是正确同步的，程序的执行将具有顺序一致性（sequentially consistent），即程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同。这里的同步是指广义上的同步，包括对常用同步原语（synchronized，volatile和final）的正确使用。

### 顺序一致性模型

顺序一致性内存模型是一个被计算机科学家理想化了的理论参考模型，它为程序员提供了极强的内存可见性保证。顺序一致性内存模型有两大特性：

* 一个线程中的所有操作必须按照程序的顺序来执行。
* 不管程序是否同步，所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见。

顺序一致性内存模型为程序员提供的视图如下：



在概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关可以连接到任意一个线程。同时，每一个线程必须按程序的顺序来执行内存读/写操作。从上图我们可以看出，在任意时间点最多只能有一个线程可以连接到内存。当多个线程并发执行时，图中的开关装置能把所有线程的所有内存读/写操作串行化。

顺序一致性内存模型中的每个操作必须立即对任意线程可见，但是在JMM中就没有这个保证。未同步程序在JMM中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。比如，在当前线程把写过的数据缓存在本地内存中，且还没有刷新到主内存之前，这个写操作仅对当前线程可见；从其他线程的角度来观察，会认为这个写操作根本还没有被当前线程执行。只有当前线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，当前线程和其它线程看到的操作执行顺序将不一致。

### 同步程序的顺序一致性

我们接下来看看正确同步的程序如何具有顺序一致性。

class SynchronizedExample {

int a = 0;

boolean flag = false;

public synchronized void writer() {

a = 1;

flag = true;

}

public synchronized void reader() {

if (flag) {

int i = a;

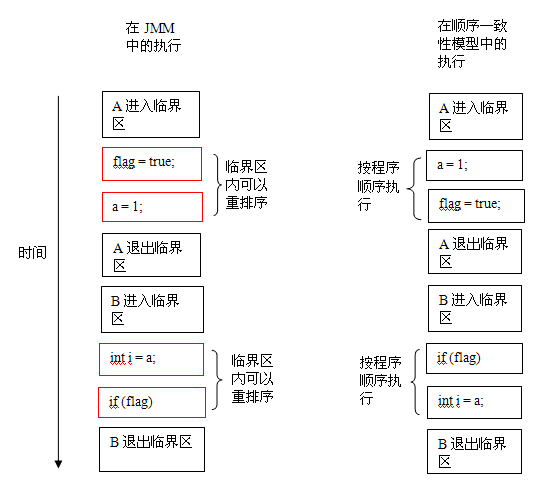
……

}

}

}

上面示例代码中，假设A线程执行writer()方法后，B线程执行reader()方法。这是一个正确同步的多线程程序。根据JMM规范，该程序的执行结果将与该程序在顺序一致性模型中的执行结果相同。下面是该程序两个内存模型中的执行时序对比图：



在顺序一致性模型中，所有操作完全按程序的顺序串行执行。而在JMM中，临界区内的代码可以重排序（但JMM不允许临界区内的代码“逸出”到临界区之外，那样会破坏监视器的语义）。JMM会在退出监视器和进入监视器这两个关键时间点做一些特别处理，使得线程在这两个时间点具有与顺序一致性模型相同的内存视图。虽然线程A在临界区内做了重排序，但由于监视器的互斥执行的特性，这里的线程B根本无法“观察”到线程A在临界区内的重排序。这种重排序既提高了执行效率，又没有改变程序的执行结果。

从这里我们可以看到JMM在具体实现上的基本方针：在不改变（正确同步的）程序执行结果的前提下，尽可能的为编译器和处理器的优化打开方便之门。

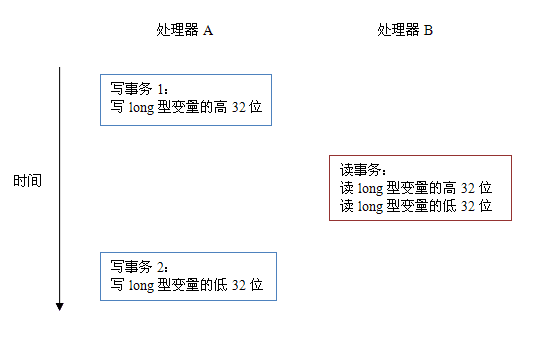
### 未同步程序的顺序一致性

JMM不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致。因为未同步程序在顺序一致性模型中执行时，整体上是无序的，其执行结果无法预知。保证未同步程序在两个模型中的执行结果一致毫无意义。和顺序一致性模型一样，未同步程序在JMM中的执行时，整体上也是无序的，其执行结果也无法预知。

同时，未同步程序在这两个模型中的执行特性有下面几个差异：

* 顺序一致性模型保证单线程内的操作会按程序的顺序执行，而JMM不保证单线程内的操作会按程序的顺序执行（比如上面正确同步的多线程程序在临界区内的重排序）。
* 顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而JMM不保证所有线程能看到一致的操作执行顺序。
* JMM不保证对64位的long型和double型变量的读/写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子性。

对于第三个差异：在一些32位的处理器上，如果要求对64位数据的读/写操作具有原子性，会有比较大的开销。为了照顾这种处理器，java语言规范鼓励但不强求JVM对64位的long型变量和double型变量的读/写具有原子性。当JVM在这种处理器上运行时，会把一个64位long/ double型变量的读/写操作拆分为两个32位的读/写操作来执行。这两个32位的读/写操作可能会被分配到不同的总线事务中执行，此时对这个64位变量的读/写将不具有原子性。当单个内存操作不具有原子性，将可能会产生意想不到后果。请看下面示意图：



如上图所示，假设处理器A写一个long型变量，同时处理器B要读这个long型变量。处理器A中64位的写操作被拆分为两个32位的写操作，且这两个32位的写操作被分配到不同的写事务中执行。同时处理器B中64位的读操作被拆分为两个32位的读操作，且这两个32位的读操作被分配到同一个的读事务中执行。当处理器A和B按上图的时序来执行时，处理器B将看到仅仅被处理器A“写了一半“的无效值。

# 方法中参数按值与按引用传递

## 按值传递是什么

指的是在方法调用时，传递的参数是按值的拷贝传递，也就是说传递后就互不相关了，示例如下：

public class TempTest {

private void change(int a){

a = 5;

}

public static void main(String[] args) {

TempTest t = new TempTest();

int a = 3;

t.change(a);//传递后，test1方法对变量值的改变不影响这里的a

System.out.println(”a=”+a);

}

}

运行结果是：a=3

## 按引用传递

### 按引用传递是什么

指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址，传递前和传递后都指向同一个引用（也就是同一个内存空间），示例如下：

public class TempTest {

private void change(A a){

a.age = 20;

}

public static void main(String[] args) {

TempTest t = new TempTest();

A a = new A();

a.age = 10;

t.change(a);

System.out.println(”age=”+a.age);

}

}

class A{

public int age = 0;

}

运行结果如下：age=20

### 对按引用传递例子的改变

理解了上面的例子，可能有人会问，那么能不能让按照引用传递的值，相互不影响呢？就是change方法里面的修改不影响到main方法里面呢？方法是在change方法里面新new一个实例就可以了，改变成下面的例子：

public class TempTest {

private void test1(A a){

**a = new A();//新加的一行**

a.age = 20;

System.out.println("test1方法中的age="+a.age);

}

public static void main(String[] args) {

TempTest t = new TempTest();

A a = new A();

a.age = 10;

t.test1(a);

System.out.println(”main方法中的age=”+a.age);

}

}

class A{

public int age = 0;

}

运行结果为：

test1方法中的age=20

main方法中的age=10

## 总结说明

“在Java里面参数传递都是按值传递”这句话的意思是：按值传递是传递的值的拷贝，按引用传递其实传递的是引用的地址值，所以统称按值传递。

在Java里面只有基本类型和按照使用双引号定义（String str = “Java私塾”）的String是按值传递，其它的都是按引用传递。

但在实际中，String由于是不可变的，replace、toUperCase、trim之类的都是返回一个String，并没有改变原String的值，所以，可以把String当成按值传递

# volatile关键字介绍

## 资料一：

### 前言

有时仅仅为了读写一个或者两个实例域就使用同步的话，显得开销过大，volatile关键字为实例域的同步访问提供了免锁的机制。如果声明一个域为volatile,那么编译器和虚拟机就知道该域是可能被另一个线程并发更新的。再讲到volatile关键字之前我们需要了解一下内存模型的相关概念以及并发编程中的三个特性：原子性，可见性和有序性。

### java内存模型与原子性，可见性和有序性

Java内存模型规定所有的变量都是存在主存当中，每个线程都有自己的工作内存。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。在java中，执行下面这个语句：

int a = 3;

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值3写入主存当中。那么Java语言本身对原子性、可见性以及有序性提供了哪些保证呢？

#### 原子性

对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。来看一下下面的代码：

x = 10; //语句1

y = x; //语句2

x++; //语句3

x = x + 1; //语句4

只有语句1是原子性操作，其他三个语句都不是原子性操作。语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。也就是说，只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

java.util.concurrent.atomic包中有很多类使用了很高效的机器级指令（而不是使用锁）来保证其他操作的原子性。例如AtomicInteger类提供了方法incrementAndGet和decrementAndGet，它们分别以原子方式将一个整数自增和自减。可以安全地使用AtomicInteger类作为共享计数器而无需同步。另外这个包还包含AtomicBoolean，AtomicLong和AtomicReference这些原子类仅供开发并发工具的系统程序员使用，应用程序员不应该使用这些类。

#### 可见性

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，所以对其他线程是可见的，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

#### 有序性

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。可以通过volatile关键字来保证一定的“有序性”。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

### volatile关键字

#### volatile保证可见性吗？

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
* 禁止进行指令重排序。

先看一段代码，假如线程1先执行，线程2后执行：

//线程1

boolean stop = false;

while(!stop){

doSomething();

}

//线程2

stop = true;

很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

为何有可能导致无法中断线程？每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

但是用volatile修饰之后就变得不一样了：

* 使用volatile关键字会强制将修改的值立即写入主存；
* 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效；
* 由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

#### volatile保证原子性吗？

我们知道volatile关键字保证了操作的可见性，但是volatile能保证对变量的操作是原子性吗？

public class Test {

public volatile int inc = 0;

public void increase() {

inc++;

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

//保证前面的线程都执行完

while(Thread.activeCount()>1)

Thread.yield();

System.out.println(test.inc);

}

}

这段代码每次运行结果都不一致，都是一个小于10000的数字，在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加1操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行。假如某个时刻变量inc的值为10，线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。那么两个线程分别进行了一次自增操作后，inc只增加了1。自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

#### volatile能保证有序性吗？

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。volatile关键字禁止指令重排序有两层意思：

* 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
* 在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

### 正确使用volatile关键字

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

* 对变量的写操作不依赖于当前值
* 该变量没有包含在具有其他变量的不变式中

第一个条件就是不能是自增自减等操作，上文已经提到volatile不保证原子性。

第二个条件我们来举个例子它包含了一个不变式 ：下界总是小于或等于上界

public class NumberRange {

private volatile int lower, upper;

public int getLower() { return lower; }

public int getUpper() { return upper; }

public void setLower(int value) {

if (value > upper)

throw new IllegalArgumentException(...);

lower = value;

}

public void setUpper(int value) {

if (value < lower)

throw new IllegalArgumentException(...);

upper = value;

}

}

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全，从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态。例如，如果初始状态是 (0, 5)，同一时间内，线程 A 调用 setLower(4) 并且线程 B 调用 setUpper(3)，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是 (4, 3)，这显然是不对的。其实就是要保证操作的原子性就可以使用volatile，使用volatile主要有两个场景:

#### 状态标志

volatile boolean shutdownRequested;

...

public void shutdown() {

shutdownRequested = true;

}

public void doWork() {

while (!shutdownRequested) {

// do stuff

}

}

很可能会从循环外部调用 shutdown() 方法 —— 即在另一个线程中 —— 因此，需要执行某种同步来确保正确实现 shutdownRequested 变量的可见性。然而，使用 synchronized 块编写循环要比使用volatile 状态标志编写麻烦很多。由于 volatile 简化了编码，并且状态标志并不依赖于程序内任何其他状态，因此此处非常适合使用 volatile。

#### 双重检查模式 （DCL）

public class Singleton {

private volatile static Singleton instance = null;

public static Singleton getInstance() {

if (instance == null) {

synchronized(this) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

在这里使用volatile会或多或少的影响性能，但考虑到程序的正确性，牺牲这点性能还是值得的。DCL优点是资源利用率高，第一次执行getInstance时单例对象才被实例化，效率高。缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷，虽然发生的概率很小。DCL虽然在一定程度解决了资源的消耗和多余的同步，线程安全等问题，但是他还是在某些情况会出现失效的问题，也就是DCL失效，在《java并发编程实践》一书建议用以下的代码(静态内部类单例模式）来替代DCL：

public class Singleton {

private Singleton(){

}

public static Singleton getInstance(){

return SingletonHolder.sInstance;

}

private static class SingletonHolder {

private static final Singleton sInstance = new Singleton();

}

}

### 总结

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件即变量真正独立于其他变量和自己以前的值 ，在某些情况下可以使用 volatile 代替 synchronized 来简化代码。然而，使用 volatile 的代码往往比使用锁的代码更加容易出错。本文介绍了可以使用 volatile 代替 synchronized 的最常见的两种用例，其他的情况我们最好还是去使用synchronized 。

## 资料二：

在C/C++中，volatile关键字的作用和java中是不一样的，总结一下：

### C/C++中的volatile关键字作用

**可见性**

“可见性”指的是在一个线程中对该变量的修改会马上由工作内存（Work Memory）写回主内存（Main Memory），所以会马上反应在其它线程的读取操作中。顺便一提，工作内存和主内存可以近似理解为实际电脑中的高速缓存和主存，工作内存是线程独享的，主存是线程共享的。

**不可优化性**

“不可优化”特性，volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

**顺序性**

“顺序性”，能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。Volatile变量与非Volatile变量的顺序，编译器不保证顺序，可能会进行乱序优化。同时，C/C++ Volatile关键词，并不能用于构建happens-before语义，因此在进行多线程程序设计时，要小心使用volatile，不要掉入volatile变量的使用陷阱之中。

### java中volatile关键字作用

**介绍**

Java也支持volatile关键字，但它被用于其他不同的用途。当volatile用于一个作用域时，Java保证如下：

volatile让变量每次在使用的时候，都从主存中取。而不是从各个线程的“工作内存”。

volatile具有synchronized关键字的“可见性”，但是没有synchronized关键字的“并发正确性”，也就是说不保证线程执行的有序性。

也就是说，volatile变量对于每次使用，线程都能得到当前volatile变量的最新值。但是volatile变量并不保证并发的正确性。

**适用于Java所有版本**

读和写一个volatile变量有全局的排序。也就是说每个线程访问一个volatile作用域时会在继续执行之前读取它的当前值，而不是（可能）使用一个缓存的值。(但是并不保证经常读写volatile作用域时读和写的相对顺序，也就是说通常这并不是有用的线程构建)。

**适用于Java5及其之后的版本**

volatile的读和写建立了一个happens-before关系，类似于申请和释放一个互斥锁。使用volatile会比使用锁更快，但是在一些情况下它不能工作。volatile使用范围在Java5中得到了扩展，特别是双重检查锁定现在能够正确工作。

### 适配

上面有一个细节，java 5版本之后volatile的读与写才建立了一个happens-before的关系，之前的版本会出现一个问题：Why is volatile used in this example of double checked locking，这个答案写的很清楚了，线程 A 在完全构造完 instance 对象之前就会给 instance 分配内存，线程B在看到 instance 已经分配了内存不为空就回去使用它，所以这就造成了B线程使用了部分初始化的 instance 对象，最后就会出问题了。

Double-checked locking里面有一句话：As of J2SE 5.0, this problem has been fixed. The volatile keyword now ensures that multiple threads handle the singleton instance correctly. This new idiom is described in [2] and [3].所以对于 Android 来说，使用 volatile关键字是一点问题都没有的了。

# 回调函数

## 1.定义接口

public class ListImageDirPopupWindow extends PopupWindow {

public interface onDirSelectedListener{

void onDirSeleted(FolderBean folderBean);

}

private onDirSelectedListener mListener;

public void setOnDirSelectedListener(onDirSelectedListener mListener){

this.mListener = mListener;

}

}

## 2.调用接口

if (mListener != null) {

mListener.onDirSeleted(mDatas.get(position));

}

## 3.注册接口

mDirPopupWindow.setOnDirSelectedListener(new onDirSelectedListener() {

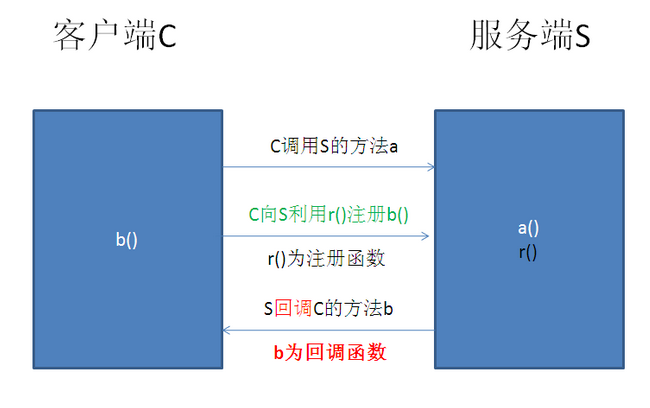
@Override

public void onDirSeleted(FolderBean folderBean) {

mCurrentDir = new File(folderBean.getDir());

}

});



# 内存泄漏

## 概述

内存泄漏也称作“存储渗漏”，用动态存储分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元。直到程序结束。即所谓内存泄漏。

内存泄漏简单地说就是申请了一块内存空间，使用完毕后没有释放掉。它的一般表现方式是程序运行时间越长，占用内存越多，最终用尽全部内存，整个系统崩溃。由程序申请的一块内存，且没有任何一个指针指向它，那么这块内存就泄露了。

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到。

**Android应用内存泄漏的的原因有以下几个：**

* 查询数据库后没有关闭游标cursor
* 构造Adapter时，没有使用 convertView 重用
* Bitmap对象不在使用时调用recycle()释放内存
* 对象被生命周期长的对象引用，如activity被静态集合引用导致activity不能释放

**内存泄漏的发现:**

* 通过DDMS中的heap工具，去发现是否有内存溢出。
* 内存泄漏如何解决：
* 通过内存分析工具 MAT(Memory Analyzer Tool)，找到内存泄露的对象

## Activity对象未被回收

### 静态变量引用Activity对象

通过静态变量引用Activty对象时，会导致Activty对象所占内存内漏。主要是因为，静态变量是驻扎在JVM的方法区，因此，静态变量引用的对象是不会被GC回收的，因为它们所引用的对象本身就是GC ROOT。即最终导致Activity对象不被回收，从而也就造成内存泄漏。

看个简单例子，比如说，你应用启动Activty的场景很多，你希望定义一个工具类Util.java，在这个类中，定义一个启动Activty的方法startActivity(Class nextActivity);以此来简化启动Activty的代码。另外，加入你当前的Activty启动另一个Activty的代码使用率也特别高。为了使得参数尽可能的少，你提供setFirstActivty，保存当前的Activty。代码如下：

public class Util {

private static Activity sActivity;

public static void setActivity(Activity activity) {

sActivity = activity;

}

public static void startActivity(Class nextActivity) {

Intent intent = new Intent(sActivity, nextActivity);

sActivity.startActivity(intent);

}

}

在当前的Activty中，只需在onCreate中调用Util.setFirstActivity(this);，在需要启动另一个Activty处调用Util.startActivity(SecondActivity.class);。在上面代码中，如果当前的Activty不再使用且Util中的sActivity对象没有更改，会导致当前Activty一直驻留在内存中。

### 静态View

有时，当一个Activity经常启动，但是对应的View读取非常耗时，我们可以通过静态View变量来保持对该Activity的rootView引用。这样就可以不用每次启动Activity都去读取并渲染View了。这确实是一个提高Activity启动速度的好方法！但是要注意，一旦View attach到我们的Window上，就会持有一个Context(即Activity)的引用。而我们的View又是一个静态变量，所以导致Activity不被回收。当然了，也不是说不能使用静态View，但是在使用静态View时，需要确保在资源回收时，将静态View detach掉。

### 内部类

我们知道，非静态内部类持有外部类的一个引用。因此，如果我们在一个外部类中定义一个静态变量，这个静态变量是引用内部类对象。将会导致内存泄漏！因为这相当于间接导致静态引用外部类。

static InnerClass innerClass;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

innerClass = this.new InnerClass();

}

class InnerClass {

}

### 匿名类

与内部类一样，匿名类也会持有外部类的引用。

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

new AsyncTask<Void, Void, Void>() {

@Override

protected Void doInBackground(Void... params) {

//另一个线程中持有Activity的引用，并且不释放

while (true) ;

}

}.execute();

}

### Handler

我们知道，主线程的Looper对象不断从消息队列中取出消息，然后再交给Handler处理。如果在Activity中定义Handler对象，那么Handler肯定是持有Activty的引用。而每个Message对象是持有Handler的引用的（Message对象的target属性持有Handler引用），从而导致Message间接引用到了Activity。如果在Activty destroy之后，消息队列中还有Message对象，Activty是不会被回收的。当然了，如果消息正在准备（处于延时入队期间）放入到消息队列中也是一样的。

private final Handler handler = new Handler() {

@Override

public void handleMessage(Message msg) {

}

};

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

handler.postDelayed(new Runnable() {

@Override

public void run() { /\* ... \*/ }

}, Integer.MAX\_VALUE);

}

解决办法就是，将Handler放入单独的类或者将Handler放入到静态内部类中（静态内部类不会持有外部类的引用）。如果想要在handler内部去调用所在的外部类Activity，可以在handler内部使用弱引用的方式指向所在Activity，这样不会导致内存泄漏。

### Threads和TimerTask

Threads和Timer导致内存泄漏的原因跟内部类一样。虽然在新的线程中创建匿名类，但是只要是匿名类/内部类，它都会持有外部类引用。

void spawnThread() {

new Thread() {

@Override public void run() {

while(true);

}

}.start();

}

void scheduleTimer() {

new Timer().schedule(new TimerTask() {

@Override

public void run() {

while(true);

}

}, Long.MAX\_VALUE >> 1);

}

### 监听器

当我们需要使用系统服务时，比如执行某些后台任务、为硬件访问提供接口等等系统服务。我们需要把自己注册到服务的监听器中。然而，这会让服务持有 activity 的引用，如果程序员忘记在 activity 销毁时取消注册，那就会导致 activity 泄漏了。

## 集合对象造成的泄漏

当我们定义一个静态的集合类时，请注意，这可能会导致内存泄漏！前面我们提到过，静态变量所引用的对象是不会被回收掉的。而我的静态集合类中，包含有大量的对象，这些对象不会被回收。另外，如果集合中保存的对象又引用到了其他的大对象，如超长字符串、Bitmap、大数组等，很容易造成OOM。

## 资源对象没关闭造成内存泄漏

当我们打开资源时，一般都会使用缓存。比如读写文件资源、打开数据库资源、使用Bitmap资源等等。当我们不再使用时，应该关闭它们，使得缓存内存区域及时回收。虽然有些对象，如果我们不去关闭，它自己在finalize()函数中会自行关闭。但是这得等到GC回收时才关闭，这样会导致缓存驻留一段时间。如果我们频繁的打开资源，内存泄漏带来的影响就比较明显了。

## 使用对象池避免频繁创建对象

在我们需要频繁创建使用某个类时，或者是在for循环里面创建新的对象时，导致JVM不断创建同一个类。我们知道，在使用Message对象时，不是直接new出来的，而是通过obtain方法获取，以及recycle方法回收。这是典型的享元模式。我们可以通过使用对象池来实现.

public class MyObject {

private static final Pools.SynchronizedPool<MyObject> MY\_POOLS = new Pools.SynchronizedPool<>(10);

public static MyObject obtain() {

MyObject object = MY\_POOLS.acquire();

if (object == null)

object = new MyObject();

return object;

}

public void recycle() {

MY\_POOLS.release(this);

}

}

# 四种内存引用

在处理内存引用之前，我们先来复习下什么是强引用、软引用、弱引用、虚引用

**强引用：**

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。 当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**软引用：**

如果一个对象只具有软引用，但内存空间足够时，垃圾回收器就不会回收它；直到虚拟机报告内存不够时才会回收， 只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。 软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

**弱引用：**

只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间是否足够，都会回收它的内存。 不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

**虚引用：**

虚引用可以理解为虚设的引用，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。 虚引用主要用来跟踪对象被垃圾回收器回收的活动。 虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。 当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。 程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。 如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

# 内存溢出

内存溢出（out of memory）通俗理解就是内存不够，通常在运行大型软件或游戏时，软件或游戏所需要的内存远远超出了你主机内安装的内存所承受大小，就叫内存溢出。此时软件或游戏就运行不了，系统会提示内存溢出，有时候会自动关闭软件，重启或者软件后释放掉一部分内存又可以正常运行该软件

避免内存溢出的方法，主要是对以下三个方面对程序进行优化

### 释放强引用

一般我们在声明对象变量时，使用完后就不管了，认为垃圾回收器会帮助我们回收这些对象所指向的内存空间，实际上如果这个对象的内存空间还处在被引用状态的话，垃圾回收器是永远不会回收它的内存空间的，只有当这个内存空间不被任何对象引用的时候，垃圾回收器才会去回收。

所以我们在使用完对象后，可以把对象置为空，这样我的垃圾回收器gc就会在合适的时候释放掉为该对象分配的内存空间

Object obj = new Object();

obj = null;

当然，在置为空前要确认是否不再需要使用该对象了，如果需要随时使用这个对象，则不能这么做

### 使用软引用

在jvm报告内存不足之前会清除所有的软引用，这样的话gc就可以收集到很多软引用释放出来的内存空间，从而解决内存吃紧的问题，避免内存溢出，什么时候被回收取决于gc的算法和gc运行时可用的内存大小。

我们可以用SoftReference来封装强引用的对象

String str = "zhuwentao"; // 强引用

SoftReference<String> strSoft = new SoftReference<String>(str); // 使用软引用封装强引用

### 使用弱引用

gc收集弱引用对象的执行过程和软引用一样，只是gc不会根据内存情况来决定是否回收弱引用的对象。

String str = "zhuwentao"; // 强引用

WeakReference<String> strWeak = new WeakReference<String>(str); // 使用弱引用封装强强引用

如果你希望能够随时取得某个对象的信息，但又不希望影响该对象的垃圾回收，则应该使用WeakReference来记住该对象，而不是使用一般的Reference。

## 图像处理

* 大部分的OOM都是发生在图片加载上的，当我们加载大图时，需要特别注意避免OOM的发生。
* 处理大图片时，不管你的手机内存有多大，如果不对图片进行处理，都有可能会发生内存溢出问题。
* 因为Android系统会为每一个应用分配一定大小的内存，并不会把整个系统内存全部分给应用，所以不管你手机内存多大，对每个App来说，它能使用的内存都是有限的。
* 这和PC端是有很大的不同，PC端如果内存不够了还可以请求使用虚拟内存，而Android系统可没这个机制。

### 在内存中压缩图片

装载大图片时需要对图片进行压缩，使用等比例压缩的方法直接在内存中处理图片

Options options = new BitmapFactory.Options();

options.inSampleSize = 4; // 原图的四分之一，设置为2则为二分之一

BitmapFactory.decodeFile(myImage.getAbsolutePath(), options);

这样做要注意的是，图片质量会变差，inSampleSize设置的值越大，图片质量就越差，不同的手机厂商缩放的比例可能不同。

### 使用完图片后回收图片所占内存

由于Android外层是使用java而底层使用的是c语言在里层为图片对象分配的内存空间。所以我们的外部虽然看起来释放了，但里层却并不一定完全释放了，我们使用完图片后最好再释放掉里层的内存空间。

if (!bitmapObject.isRecyled()) { // Bitmap对象没有被回收

bitmapObject.recycle(); // 释放

System.gc(); // 提醒系统及时回收

}

### 降低要显示的图片色彩质量

* Android中Bitmap有四种图片色彩模式：
* ALPHA\_8：每个像素需要占用内存中的1byte
* RGB\_565：每个像素需要占用内存中的2byte
* ARGB\_4444：每个像素需要占用内存中的2byte
* ARGB\_8888：每个像素需要占用内存中的4byte

我们创建Bitmap时，默认的色彩模式是ARGB\_8888的，这种色彩模式是质量最高的，当然这样的模式占用的内存也最大。

而ARGB\_4444每个像素只占用2byte，所以使用ARGB\_4444的模式也能降低图片占用的内存大小。

BitmapFactory.Options options = new BitmapFactory.Options();

options.inPreferredConfig = Bitmap.Config.ARGB\_4444;

Bitmap btimapObject = BitmapFactory.decodeFile(myImage.getAbsolutePath(), options);

* 其实大多数图片设置成ARGB\_4444模式后，在显示上是看不出与ARGB\_8888模式有什么差别的，只是在具有渐变色效果的图片时，可能会让渐变色呈现色彩条样的效果。
* 这种降低色彩质量的方法对内存的降低效果不如方法1明显。

### 查询图片信息时不把图片加载到内存中

有时候我们取得一张图片，也许只是为了获得这个图片的一些信息，比如图片的width、height等信息，不需要显示到界面上，这个时候我们可以不把图片加载到内存中。

BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true; // 不把图片加载到内存中

Bitmap btimapObject = BitmapFactory.decodeFile(myImage.getAbsolutePath(), options);

inJustDecodeBounds属性，如果值为true，那么将不返回实际的Bitmap对象，也不给其分配内存空间，但允许我们查询图片宽、高、大小等基本信息。

（获取原始宽高：options.outWidth，options.outHeight）

# 图片OOM

## 前言

开源图片加载库能为我们解决绝大部分有关图片的问题，然而并不是所有！首先，图片从来源上可以分成三大类：网络图片、手机图片、APK资源图片。网络图片和手机图片都在图片加载库功能的覆盖范围内，基本上不用开发者太操心，但是APK资源图片却不在此范围！

关于APK资源图片有3个特征：

* 资源图片基本都是在xml中引用 ，在Java中也是通过资源ID查找 。
* 资源图片一般不使用异步记载，不会出现loading图这些中间状态。
* 资源图片不会加载失败，如果失败了那么APP也挂掉了。

正是由于这3点特征，所以图片加载库实在鞭长莫及。那么就很容易出现一个问题：图片过大导致OOM！很多APP为了追求酷炫的效果，热衷于设计绚丽全屏背景页面。既然是为了炫酷，考虑到用户体验，这些全屏背景图自然不能使用网络图片了，所以，这些图片都被放在apk包中作为资源文件直接引用。使用这些资源图片的方式一般都是：

android:background="@drawable/xxx"

正常情况下，这样使用自然不会出现问题，但是如果APP内存紧张，很容易就出现OOM，尤其是5.0版本以下的手机，经常跑着跑着就Crash了，始作俑者就是这个。为了解决这种问题，最常用的方式是找设计师压缩图片。而压缩图片有两种方式：缩小尺寸和降低质量

* 缩小尺寸： 压缩图片的宽度和高度。由于图片的内存占用与宽高成正比，这种方式确实有效，但是图片显示时会被拉伸导致变形，从而失却美感。
* 降低质量： 降低图片的色彩度，像素颜色密度。这其实是一个误区，很多人认为图片的存储占用空间小，图片的内存占用就会小，其实是错误的观点。这是方式并不会影响图片的内存占用，反而由于质量降低（下文具体分析），使得页面缺乏质感。必须记住：图片的内存占用与图片质量毫无干系！

为了寻求一个合理的解决方案，必须知彼知己。下面，我们来详细分析下资源图片的内存占用的情况！

## 计算Bitmap的内存占用

以一张标准720p的全屏图片为例，宽高比为720x1280，对应的资源文件夹为drawable-xhdpi。同样，设备以标准720p的小米2S手机为例，density=320。

首先，android设备上图片都被处理成Bitmap对象。生成Bitmap有一个非常重要的参数Config，属性值有ALPHA\_8、RGB\_565、ARGB\_4444、ARGB\_8888四种。不同的属性值对应的图片每个像素点占用内存大小不同，ALPHA\_8每个像素占用1byte，RGB\_565和ARGB\_4444占用2byte，ARGB\_8888占用4byte，其中ARGB\_4444在高版本中已经废弃。那么，资源图片被decode成Bitmap的时候，Config参数值是哪个呢？来看几段代码。

**Resources.java**

private Drawable loadDrawableForCookie(TypedValue value, int id, Theme theme) {

...

final String file = value.string.toString();

...

final Drawable dr;

if (file.endsWith(".xml")) {

final XmlResourceParser rp = loadXmlResourceParser(file, id, value.assetCookie, "drawable");

dr = Drawable.createFromXml(this, rp, theme);

rp.close();

} else {

final InputStream is = mAssets.openNonAsset(value.assetCookie, file, AssetManager.ACCESS\_STREAMING);

dr = Drawable.createFromResourceStream(this, value, is, file, null);

is.close();

}

...

}

**Drawable.java**

public static Drawable createFromResourceStream(Resources res, TypedValue value,

InputStream is, String srcName, BitmapFactory.Options opts) {

...

if (opts == null) opts = new BitmapFactory.Options();

opts.inScreenDensity = res != null ? res.getDisplayMetrics().noncompatDensityDpi : DisplayMetrics.DENSITY\_DEVICE;

Bitmap bm = BitmapFactory.decodeResourceStream(res, value, is, pad, opts);

...

return null;

}

**BitmapFactory.java**

public static class Options {

...

/\*\*

\* Image are loaded with the {@link Bitmap.Config#ARGB\_8888} config by default.

\*/

public Bitmap.Config inPreferredConfig = Bitmap.Config.ARGB\_8888;

...

}

我们图片的尺寸是720x1280，也就是说像素点个数是720x1280=921600，所有像素点占用内存=720x1280x4=3686400 byte=3.515625M，这个大小是图片不做任何处理时占用的内存大小。

另外，不管图片的内容是什么样子，体现在内存中的也仅仅是每个像素点对应的字节的值不同，大小是一样的，即一张720x1280的空白图和一张720x1280的彩色绚图占用内存大小是一致的。所以说想要降低占用内存，唯有减小宽高尺寸。

刚刚说过，计算出来的3.515625M大小是图片未作任何处理时的大小，但是系统在将图片处理成Drawable对象的时候是否未作处理呢？答案是：不！

**BitmapFactory.java**

public static Bitmap decodeResourceStream(Resources res, TypedValue value, InputStream is, Rect pad, Options opts) {

if (opts == null) {

opts = new Options();

}

if (opts.inDensity == 0 && value != null) {

final int density = value.density;

if (density == TypedValue.DENSITY\_DEFAULT) {

opts.inDensity = DisplayMetrics.DENSITY\_DEFAULT;

} else if (density != TypedValue.DENSITY\_NONE) {

opts.inDensity = density;

}

}

if (opts.inTargetDensity == 0 && res != null) {

opts.inTargetDensity = res.getDisplayMetrics().densityDpi;

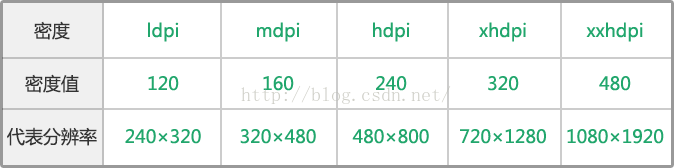
}

return decodeStream(is, pad, opts);

}

代码中Options有两个非常重要的参数，inDensity和inTargetDensity，先来解释一下这俩参数的作用。

inDensity表示被设定的图像密度，决定这个值的是图片所放置的文件目录，比如drawable-hdpi、drawable-xhdpi等等，其对应的density如下表：



代码中opts.inDensity 被赋值为 value.density，也就是资源维度对应的密度值。如果图片放在drawable-hdpi下，inDensity=240，如果放在drawable-xhdpi下，inDensity=320。

inTargetDensity表示最终需要适配到的图片密度，这个值由手机设备来决定，上面代码中其值为DisplayMetrics的densityDpi，手机屏幕越高清这个值越大，而我们例子中720p的小米2S对应的densityDpi=320。

如果inDensity的值和inTargetDensity的值不相等，那么图片尺寸就被会缩放，缩放的比例为 inTargetDensity / inDensity。当然，宽高是需要同时等比缩放的，不然图片就变形了。

前面说过图片占用内存与图片的尺寸有关，如果被尺寸缩放了，内存大小就变了。前面未作任何缩放处理的720x1280图占用内存是3.515625M，假设放在drawable-ldpi目录下inDensity=120，设备inTargetDensity=320，那么最终的占用内存大小将是3.515625Mx(320/120)x(320/120)=25M。一张图片占用25M大小，很恐怖的一个值，这种情况下，app估计直接挂了，如果放在drawable-hdpi下，占用就是6.25M，drawable-xhdpi下占用是3.515625M。由此可见，图片放置的目录一定要慎重。

最终我们得出一个公式：

资源图片内存大小 = 宽 x 高 x 4 x （设备密度 / 资源维度密度）x（设备密度 / 资源维度密度）

## 图像后门inPurgeable

前面说到，资源图片放置的目录不对会导致内存占用翻倍，但也不是放的密度维度越高越好，毕竟还是要做适配，不然小尺寸图片显示在高清大屏幕上就不好看了。而即使图片放对位置，占用内存大小也是相当惊人的，来个十张大图应用内存就蹭蹭上去了，冷不丁还来个OOM。相信很多人都找到过解决方案：inPurgeable

public static Bitmap readBitmap(Context context, int resId) {

BitmapFactory.Options opt = new BitmapFactory.Options();

opt.inPurgeable = true;

opt.inInputShareable = true;

InputStream is =context.getResources().openRawResource(resId);

return BitmapFactory.decodeStream(is, null, opt);

}

那么，这段代码是否起效果呢？答案是肯定的，以前经常报OOM的现在都好了，而且用AS的内存监视器一看，加载图片时基本上不占内存。不管有没有其它问题，姑且把这个称之为图像后门吧。下面，我们来看这个后门为什么能起效果！

**BitmapFactory.java**

public static Bitmap decodeStream(InputStream is, Rect outPadding, Options opts) {

if (is == null) {

return null;

}

Bitmap bm = null;

...

if (is instanceof AssetManager.AssetInputStream) {

...

} else {

bm = decodeStreamInternal(is, outPadding, opts);

}

}

private static Bitmap decodeStreamInternal(InputStream is, Rect outPadding, Options opts) {

byte [] tempStorage = null;

if (opts != null) tempStorage = opts.inTempStorage;

if (tempStorage == null) tempStorage = new byte[DECODE\_BUFFER\_SIZE];

return nativeDecodeStream(is, tempStorage, outPadding, opts);

}

private static native Bitmap nativeDecodeStream(InputStream is, byte[] storage, Rect padding, Options opts);

很明显，decodeStream这段代码最终调用的是native层的类库，我们追踪下去查看（下面以JellyBean源码为例）。

**BitmapFactory.cpp**

static jobject doDecode(JNIEnv\* env, SkStream\* stream, jobject padding,

jobject options, bool allowPurgeable, bool forcePurgeable = false,

bool applyScale = false, float scale = 1.0f) {

...

if (!isPurgeable) {

decoder->setAllocator(&javaAllocator);

}

...

if (isPurgeable) {

decodeMode = SkImageDecoder::kDecodeBounds\_Mode;

}

...

if (isPurgeable) {

pr = installPixelRef(bitmap, stream, sampleSize, doDither);

} else {

pr = bitmap->pixelRef();

}

...

}

static SkPixelRef\* installPixelRef(SkBitmap\* bitmap, SkStream\* stream, int sampleSize, bool ditherImage) {

SkImageRef\* pr;

// only use ashmem for large images, since mmaps come at a price

if (bitmap->getSize() >= 32 \* 1024) {

pr = new SkImageRef\_ashmem(stream, bitmap->config(), sampleSize);

} else {

pr = new SkImageRef\_GlobalPool(stream, bitmap->config(), sampleSize);

}

...

return pr;

}

相关isPurgeable的代码就这么多，最终关于图片的decode逻辑都在installPixelRef中，有一段逻辑值得玩味。如果图片大小（占用内存）大于32x1024=32K，那么就使用Ashmem，否则就就放入一个引用池中。这个做法也很容易理解，如果图片不大，直接放到native层内存中，读取方便且迅速。如果图片过大，放到native层内存也就不合理了，不然图片一多，native层内存很难管理。但是如果使用Ashmem匿名共享内存方式，写入到设备文件中，需要时再读取就能避免很大的内存消耗了，另外，这块内存是由Linux系统的内存管理来管理的，系统内存不足可以直接回收。而且，由于Ashmem跨进程的特性，同一张图片内存是可以跨进程共享的，这也是inInputShareable属性的由来。由此可见，如果inPurgeable=true，图片所占用的内存就完全与Java Heap无关了，自然就不会有OOM这种烦恼了。

但是，万事有利有弊，一件事情的成功往往是用牺牲其它方面换来的。前面说过，使用Resources获取图片Drawable的时候，会默认使用inDensity和inTargetDensity属性缩放图片来达到适配不同分辨率屏幕的目的。但是如果设置了inPurgeable=true来避免在Java Heap中分配内存，inDensity和inTargetDensity这两个属性就不能再使用了，否则即使inPurgeable=true，图片仍然会在Java Heap中分配内存。关于这一点，从以下代码中可以验证：

**BitmapFactory.cpp**

static jobject doDecode(JNIEnv\* env, SkStream\* stream, jobject padding,

jobject options, bool allowPurgeable, bool forcePurgeable = false,

bool applyScale = false, float scale = 1.0f) {

...

bool willScale = applyScale && scale != 1.0f;

bool isPurgeable = !willScale &&

(forcePurgeable || (allowPurgeable && optionsPurgeable(env, options)));

...

}

在doDecode方法中，isPurgeable会重新赋值，首决条件是图片不会缩放（willScale），其次才会判断Options中的isPurgeable属性。很明显，如果inDensity和inTargetDensity两个属性断定图片需要缩放，isPurgeable会被强制设定成false。这么做的原因很简单，Ashmem不可能维护多套不同尺寸的相同图片。

如果要解决这种适配问题，唯一的解决方案就是图片切成不同的尺寸，放到不同维度的drawable目录下。这样虽然不能做到精准适配，但是可以做到大体适配。原理就是，不同分辨率的屏幕decode相匹配密度维度目录下的对应尺寸图片。

说完适配的问题，你以为坑就到此结束了？其实不然，真正的大问题不是这个！我们来看inPurgeable属性的一段官方注释：While inPurgeable can help avoid big Dalvik heap allocations (from API level 11 onward), it sacrifices performance predictability since any image that the view system tries to draw may incur a decode delay which can lead to dropped frames。意思就是：虽然inPurgeable能避免在Heap中分配一大段内存，但这个是以牺牲性能为代价的，如果图片要绘制到View上可能出现延时导致掉帧。

前面说过，inPurgeable=true的情况下，大图使用Ashmem共享内存存储图片，但是这部分内存存储的仅仅是解码前的图片数据，我们获取的Bitmap只是一个空包弹，不含任何像素信息。当图片需要渲染的时候，先要对一个个像素点进行解码，这个过程是比较耗时的，而偏偏又发生在UI线程中，必须等图像解码完成，UI线程才能继续渲染。如果图片像素点过多，计算量大，很容易就导致卡帧。最坑爹的是，Ashmem内存是由Linux系统统一管理的，如果系统内存紧张，这块儿图片内存很容易被回收，当图片再次被渲染时，Ashmem设备文件就需要重新映射内存再重新解码。

综上所诉，虽然inPurgeable既能导致适配问题，又可能导致性能问题，那么我们为什么还要使用呢？理由很简单:相对于出现OOM导致Crash，这两点牺牲仍然是值得的！Facebook出品的大名鼎鼎的图片加载库Fresco中对图片的处理都使用了inPurgeable=true，代码如下 ：

BitmapFactory.Options = new BitmapFactory.Options();

options.inPurgeable = true;

Bitmap bitmap = BitmapFactory.decodeByteArray(jpeg, 0, jpeg.length, options);

虽然，Fresco和我们所说的资源图片干系并不大，但是很多思想还是值得我们借鉴。

## 被堵上的后门你还在用？

很多时候，知其然而不知其所以然，很容易出问题，如果又不关注版本变化，就肯定会出问题，inPurgeable就是一个很典型的例子。在5.0版本及以上，inPurgeable这个属性已经被标志为过时了！即使inPurgeable=true，也不会再使用Ashmem内存存放图片，而是直接放到了Java Heap中，简而言之就是inPurgeable属性被忽略了。

因为Android系统从5.0开始对Java Heap内存管理做了大幅的优化。和以往不同的是，对象不再统一管理和回收，而是在Java Heap中单独开辟了一块区域用来存放大型对象，比如Bitmap这种，同时这块内存区域的垃圾回收机制也是和其它区域完全分开的，这样就使得OOM的概率大幅降低，而且读取效率更高。所以，用Ashmem来存储图片就完全没有必要了，何况后者还会导致性能问题。既然这样，我们就需要考虑继续使用inPurgeable方式处理资源图片是否合理了。

如果仔细阅读过Resources的源码，会发现对于Drawable对象有一套缓存机制，比如当一张图片被解码成BitmapDrawable对象后，会被存储到缓存中，下次再使用这张图片将优先从缓存中获取，既避免了图片重复decode的耗时，又做到了内存的复用，大体代码如下：

**Resources.java**

Drawable loadDrawable(TypedValue value, int id, Theme theme) throws NotFoundException {

...

// First, check whether we have a cached version of this drawable

// that was inflated against the specified theme.

if (!mPreloading) {

final Drawable cachedDrawable = caches.getInstance(key, theme);

if (cachedDrawable != null) {

return cachedDrawable;

}

}

// Next, check preloaded drawables. These may contain unresolved theme

// attributes.

final ConstantState cs;

if (isColorDrawable) {

cs = sPreloadedColorDrawables.get(key);

} else {

cs = sPreloadedDrawables[mConfiguration.getLayoutDirection()].get(key);

}

...

// If we were able to obtain a drawable, store it in the appropriate

// cache: preload, not themed, null theme, or theme-specific.

if (dr != null) {

dr.setChangingConfigurations(value.changingConfigurations);

cacheDrawable(value, isColorDrawable, caches, theme, canApplyTheme, key, dr);

}

...

}

在inPurgeable后门被堵上之后，如果我们仍然通过BitmapFactory.decodeStream的方式获取资源图片的Bitmap，就会导致相同的图片重复decode，且多次在Java Heap中开辟内存

同样的方式，原本在5.0以下可以节省Java Heap内存占用，在5.0及以上反而成了真正内存杀手！所以在真正使用inPurgeable时是需要区分版本的，最简单的解决方案如下：

public static Drawable decodeLargeResourceImage(Resources resources, int resId) {

Drawable drawable;

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.LOLLIPOP) {

drawable = resources.getDrawable(resId, null);

} else {

try {

BitmapFactory.Options opt = new BitmapFactory.Options();

opt.inPurgeable = true;

opt.inInputShareable = true;

InputStream is = resources.openRawResource(resId);

drawable = new BitmapDrawable(resources, BitmapFactory.decodeStream(is, null, opt));

} catch (OutOfMemoryError e) {

drawable = null;

}

}

return drawable;

}

之前看过一篇文章，阿里手机淘宝客户端对存到Ashmem的图片解码做了优化，在工作线程中对图片真正解码，从而避免在UI线程渲染图片时解码，同时锁住Ashmem内存，防止在系统内存紧张时回收出现二次解码。再者，针对资源图片，目前出现了SVG矢量图代替常规PNG图片的解决方案，但也仅仅限于线条简易的Icon图。对于图片处理这一块，需要学习和研究的方面太多，路漫漫其修远兮，吾将上下而求索！

# 避免创建不必要的对象

## 使用单例

单例是我们常用的设计模式，使用这种模式，我们可以只提供一个对象供全局调用。因此单例是避免创建不必要的对象的一种方式。单例模式上手容易，但是需要注意很多问题，最重要的就是多线程并发的情况下保证单例的唯一性。当然方式很多，比如饿汉式，懒汉式double-check等

## 避免进行隐式装箱

自动装箱是Java 5 引入的一个特性，即自动将原始类型的数据转换成对应的引用类型，比如将int转为Integer等。这种特性，极大的减少了编码时的琐碎工作，但是稍有不注意就可能创建了不必要的对象了。比如下面的代码

Integer sum = 0;

for(int i=1000; i<5000; i++){

sum+=i;

}

上面的代码sum+=i可以看成sum = sum + i，但是+这个操作符不适用于Integer对象，首先sum进行自动拆箱操作，进行数值相加操作，最后发生自动装箱操作转换成Integer对象。其内部变化如下

int result = sum.intValue() + i;

Integer sum = new Integer(result);

由于我们这里声明的sum为Integer类型，在上面的循环中会创建将近4000个无用的Integer对象，在这样庞大的循环中，会降低程序的性能并且加重了垃圾回收的工作量。因此在我们编程时，需要注意到这一点，正确地声明变量类型，避免因为自动装箱引起的性能问题。

另外，当将原始数据类型的值加入集合中时，也会发生自动装箱，所以这个过程中也是有对象创建的。如有需要避免这种情况，可以选择SparseArray,SparseBooleanArray,SparseLongArray等容器

## 谨慎选用容器

Java和Android提供了很多编辑的容器集合来组织对象。比如ArrayList,ContentValues,HashMap等。然而，这样容器虽然使用起来方便，但也存在一些问题，就是他们会自动扩容，这其中不是创建新的对象，而是创建一个更大的容器对象。这就意味这将占用更大的内存空间。以HashMap为例，当我们put key和value时，会检测是否需要扩容，如需要则双倍扩容

@Override

public V put(K key, V value) {

if (key == null) {

return putValueForNullKey(value);

}

//some code here

// No entry for (non-null) key is present; create one

modCount++;

if (size++ > threshold) {

tab = doubleCapacity();

index = hash & (tab.length - 1);

}

addNewEntry(key, value, hash, index);

return null;

}

关于扩容的问题，通常有如下几种方法

* 预估一个较大的容量值，避免多次扩容
* 寻找替代的数据结构，确保做到时间和空间的平衡

## 用好LaunchMode

提到LaunchMode必然和Activity有关系。正常情况下我们在manifest中声明Activity，如果不设置LaunchMode就使用默认的standard模式。一旦设置成standard，每当有一次Intent请求，就会创建一个新的Activity实例。举个例子，如果有10个撰写邮件的Intent，那么就会创建10个ComposeMailActivity的实例来处理这些Intent。结果很明显，这种模式会创建某个Activity的多个实例，如果对于一个搜索功能的Activity，实际上保持一个Activity示例就可以了，使用standard模式会造成Activity实例的过多创建，因而不好。确保符合常理的情况下，合理的使用LaunchMode，减少Activity的创建。

## Activity处理onConfigurationChanged

这又是一个关于Activity对象创建相关的，因为Activity创建的成本相对其他对象要高很多。默认情况下，当我们进行屏幕旋转时，原Activity会销毁，一个新的Activity被创建，之所以这样做是为了处理布局适应。当然这是系统默认的做法，在我们开发可控的情况下，我们可以避免重新创建Activity。以屏幕切换为例，在Activity声明时，加上

<activity

android:name=".MainActivity"

android:label="@string/app\_name"

android:theme="@style/AppTheme.NoActionBar"

android:configChanges="orientation"

>

然后重写Activity的onConfigurationChanged方法

public void onConfigurationChanged(Configuration newConfig) {

super.onConfigurationChanged(newConfig);

if (newConfig.orientation == Configuration.ORIENTATION\_PORTRAIT) {

setContentView(R.layout.portrait\_layout);

} else if (newConfig.orientation == Configuration.ORIENTATION\_LANDSCAPE) {

setContentView(R.layout.landscape\_layout);

}

}

## 注意字符串拼接

字符串这个或许是最不起眼的一项了。这里主要讲的是字符串的拼接Log.i(LOGTAG, "onCreate bundle=" + savedInstanceState);这应该是我们最常见的打log的方式了，然而字符串的拼接内部实际是生成StringBuilder对象，然后挨个进行append，直至最后调用toString方法的过程。下面是一段代码循环的代码，这明显是很不好的，因为这其中创建了很多的StringBuilder对象。

public void implicitUseStringBuilder(String[] values) {

String result = "";

for (int i = 0 ; i < values.length; i ++) {

result += values[i];

}

System.out.println(result);

}

降低字符串拼接的方法有

* 使用String.format替换
* 如果是循环拼接，建议显式在循环外部创建StringBuilder使用

## 减少布局层级

布局层级过多，不仅导致inflate过程耗时，还多创建了多余的辅助布局。所以减少辅助布局还是很有必要的。可以尝试其他布局方式或者自定义视图来解决这类的问题

## 提前检查，减少不必要的异常

异常对于程序来说，在平常不过了，然后其实异常的代码很高的，因为它需要收集现场数据stacktrace。但是还是有一些避免异常抛出的措施的，那就是做一些提前检查。

比如，我们想要打印一个文件的每一行字符串，没做检查的代码如下，是存在FileNotFoundException抛出可能的。

private void printFileByLine(String filePath) {

try {

FileInputStream inputStream = new FileInputStream("textfile.txt");

BufferedReader br = new BufferedReader(new InputStreamReader(inputStream));

String strLine;

//Read File Line By Line

while ((strLine = br.readLine()) != null) {

// Print the content on the console

System.out.println (strLine);

}

br.close();

} catch(FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

如果我们进行文件是否存在的检查，抛出FileNotFoundException的概率会减少很多，

private void printFileByLine(String filePath) {

if (!new File(filePath).exists()) {

return;

}

try {

FileInputStream inputStream = new FileInputStream("textfile.txt");

BufferedReader br = new BufferedReader(new InputStreamReader(inputStream));

String strLine;

//Read File Line By Line

while ((strLine = br.readLine()) != null) {

// Print the content on the console

System.out.println (strLine);

}

br.close();

} catch(FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

上述的检查是一个不错的编码技巧，建议采纳。

## 不要过多创建线程

在android中，我们应该尽量避免在主线程中执行耗时的操作，因而需要使用其他线程。

private void testThread() {

new Thread() {

@Override

public void run() {

super.run();

//do some io work

}

}.start();

}

虽然这些能工作，但是创建线程的代价远比普通对象要 高的多，建议使用HandlerThread或者ThreadPool做替换。

## 使用注解替代枚举

枚举是我们经常使用的一种用作值限定的手段，使用枚举比单纯的常量约定要靠谱。然后枚举的实质还是创建对象。好在Android提供了相关的注解，使得值限定在编译时进行，进而减少了运行时的压力。相关的注解为IntDef和StringDef。

如下以IntDef为例，介绍如何使用

在一个文件中如下声明

public class AppConstants {

public static final int STATE\_OPEN = 0;

public static final int STATE\_CLOSE = 1;

public static final int STATE\_BROKEN = 2;

@IntDef({STATE\_OPEN, STATE\_CLOSE, STATE\_BROKEN})

public @interface DoorState {}

}

然后设置书写这样的方法

private void setDoorState(@AppConstants.DoorState int state) {

//some code

}

当调用方法时只能使用STATE\_OPEN，STATE\_CLOSE和STATE\_BROKEN。使用其他值会导致编译提醒和警告。

## 选用对象池

在Android中有很多池的概念，如线程池，连接池。包括我们很长用的Handler.Message就是使用了池的技术。比如，我们想要使用Handler发送消息，可以使用Message msg = new Message()，也可以使用Message msg = handler.obtainMessage()。使用池并不会每一次都创建新的对象，而是优先从池中取对象。

使用对象池需要需要注意几点

* 将对象放回池中，注意初始化对象的数据，防止存在脏数据
* 合理控制池的增长，避免过大，导致很多对象处于闲置状态

## 谨慎初始化Application

Android应用可以支持开启多个进程。 通常的做法是这样

<service android:name=".NetworkService"

android:process=":network"

/>

通常我们在Application的onCreate方法中会做很多初始化操作,但是每个进程启动都需要执行到这个onCreate方法,为了避免不必要的初始化,建议按照进程(通过判断当前进程名)对应初始化.

public class MyApplication extends Application {

private static final String LOGTAG = "MyApplication";

@Override

public void onCreate() {

String currentProcessName = getCurrentProcessName();

Log.i(LOGTAG, "onCreate currentProcessName=" + currentProcessName);

super.onCreate();

if (getPackageName().equals(currentProcessName)) {

//init for default process

} else if (currentProcessName.endsWith(":network")) {

//init for netowrk process

}

}

private String getCurrentProcessName() {

String currentProcessName = "";

int pid = android.os.Process.myPid();

ActivityManager manager = (ActivityManager) this.getSystemService(Context.ACTIVITY\_SERVICE);

for (ActivityManager.RunningAppProcessInfo processInfo : manager.getRunningAppProcesses()) {

if (processInfo.pid == pid) {

currentProcessName = processInfo.processName;

break;

}

}

return currentProcessName;

}

}

# 在谈MVP之前，你真的懂MVC吗

最近看到很多文章在谈论MVP或者MVVM模式的，但其实无论MVP还是MVVM都只是MVC模式的一种变种。而如果你对MVC的设计理念都还没有理解透彻，那么即使换成MVP亦或MVVM也不可能让你杂乱不堪的代码突然变得清晰明了起来，模式绝不是救命的稻草，它只是一种表现形式，真正要学的其蕴含的思维方式。

## 什么才是MVC？

这是一个非新手都就会嗤之以鼻的问题，试问哪个程序猿不知道什么是MVC，但在此我希望大家先忘记之前对MVC的所有知识，很多时候学习的第一步就是承认自己的无知，这是一个多么重要的步骤，又是一个多么容易遗忘的步骤啊。包括我自己也是如此，经常因为固有思维而变得傲慢而不自知，今天我们就一起重头来学习一下MVC的历史。

对于MVC的概念我想没人不知，但是大部分人其实并不知道MVC的概念其实不止一个，从纵向来看，它经过了历史上很多的演进和变种；从横向来看，它也有许许多多不同的细微差异。即使包括后来的MVP和MVVM也都只能算它的一个变种而已。

## 经典MVC模式

大家坐稳了，老司机要带大家穿越时空回到1978年，来听听MVC模式的创始人挪威教授Trygve Reenskaug是怎么定义MVC模式的。

这是最早期的MVC模式，其中三者的定义可以简单的理解为：

* Model，负责的是数据，这里的“数据”不仅限于数据本身，还包括处理数据的逻辑。
* View，负责数据的表现形式，将数据及数据的变化呈现给用户。
* Controller，负责用户的输入，将用户的命令转化成消息传递给model或view，是一个翻译者。

在早期的MVC模式中，Controller的设计目的其实并不是为了隔离Model和View的，而后来这点才发现了变化。

**Model & View**

**Model和View的关系，也可以分为两种形式**：

**push model：**

View在model上将自己注册为数据的监听者，model的数据发生变化时会发送通知，view接到通知后用新数据更新自己。

**pull model：**

View负责在它需要的时候调用model来获取当前最新的数据。

但不管是那种模式，model对于view都是没有感知的，push model是在数据变化的时候简单的发送广播，告之所有对该数据变化感兴趣的监听者，而pull model是view对model进行调用。因此在任何一种模式下，model都是不能直接操作view。

这种model-view模式也就是俗称的观察者模式，model是被观察者，view是观察者，当被观察者发送变化的时候，通知注册在它上面的所有观察者。这样设计带来的另一个好处就在于多个view可以监听同一个model的变化。

**Controller & View**

关于Controller和View的关系，Controller是绑定在View上面的，意思是用户任何在View上的操作（例如点击按钮等）都会调用Controller上的一个回调方法。其实也意味着View是持有Controller的引用的，当用户做相应操作时，是由View来调用合适的Controller方法的，而Controller对View的操作在早期概念中则不太明确。

**Controller & Model**

Controller是可以向Model直接通信的。例如用户点击了删除按钮，那么Controller将用户的这个操作翻译成“用户需要删除这条数据”的消息传递给Model，Model负责删除数据，然后通知View来更新页面以告知用户数据的变化。

## 现代MVC模式

与经典MVC模式不同，很多现代系统设计中，如Apple Cocoa框架，最大的改变在于将Controller的位置放在了Model和View之间。

主要区别就在于Controller的位置变了：Controller将消息传递给Model，M处理完数据后是先将数据的变化通知给C，再由C来通知View来变化视图的。也就是说Controller变成了在Model和View之间双向传递数据的中间协调者，关系变成了： View <-> Controller <-> Model 。

**Model & View**

Model和View之间没有了任何关系，所有通信都是通过Controller传递。

**Model & Controller & View**

View通过Controller向model传递用户操作的消息，而model在处理完数据后通过Controller来向View来传递结果。Controller从经典MVC模式中的单向翻译官变成了双向的中间人。

**为什么要这样变？**

其实大家都看出来了，这样变的主要目的就是为了让Model和View之间不再直接联系。从而使得三者的关系理得更清楚了。

**这货不就是MVP吗？**

了解MVP概念的同学可能读到这里可以会产生巨大的问号：这货不就是MVP吗？MVP里面的Presenter不就是充当Controller和View之间的中间人吗？

可以说Apple Cocoa这类框架使用的这种进化版的MVC确实离MVP相差很小了，可以说只差一步而已，MVP只是把三者之间的关系解藕得更厉害而已。

**最后说两句**

不管是传统的MVC，还是进化的MVC，亦或者MVP和MVVM模式，你会发现其实它们的设计理念都是一致的，逐步进化也只是为了更进一步的达到这个理念的目标，那就是：

上帝的归上帝，凯撒的归凯撒！

如果你没有理解这一点，那么用什么模式也是混乱的；而如果你理解了这一点，什么模式不用也是清晰的。

# MVP模式是你的救命稻草吗？

## 什么是MVP

MVP与MVC最大的区别就在与将Model和View通过Presenter隔开了，不再允许其互相直接通信，而所有的消息都是通过Presenter这个中间人来传递。而这样做的目的主要是为了将数据和展示划出更明确的界限。

## MVP到底是在解决什么问题的：

**数据管理**

什么是数据？(Model)

如何筛选数据？(Selections)

如何改变数据？(Commands)

**用户界面**

如何展示数据？(View)

如何通过事件来改变数据？(Interactor)

如何将这些放在一起？(Presenter)

**可以看出一个GUI程序的核心**

还是在于用户和数据之间的关系。而架构的引入也是为了解决这几个核心的问题。那么MVP又是如何解决这几个问题的，Model，View，Presenter三者又各自负责什么呢？谁来负责请求网络数据，访问和存储本地数据，谁来负责处理数据，谁来负责显示数据，谁又来负责和用户交互呢？具体表现在代码上，也可以说：网络请求应该放在哪，数据库访问应该放在哪，业务逻辑应该放在哪里，处理用户输入应该放在哪，谁又来控制显示或隐藏View的等具体的细节问题。

**如何拆分**

Model，负责定义数据（解决什么是数据）

Presenter, 负责在Model和View之间，从model里取出数据，格式化后在View上展示（解决如何把数据和用户界面放在一起）。

View，负责担任一个被动界面，用于展示数据。（解决如何展示数据）

和MVC比较而已，这里出现一个最大的疑问就是：那么谁来负责和用户的操作交互呢？答案是，用户在View上的所有操作（事件）都由View路由给Presenter，由Presenter来与其交互。而和MVC最大的区别在于Model和View完全被Presenter隔开了，Presenter作为它们之间的中间人去传递所有数据。

**如何组合**

很显然Presenter作为中间者，它是同时拥有View和Model的引用的，为了在它们之间起到桥梁作用，即Presenter会主动和View和Model进行通信。而Model和View必须是完全隔离的，不允许两者之间互相通信，保持对彼此的不感知，这样的好处是你彻底将数据和展示分离来开，并且可以独立的为Model去做测试。

**Model在三者中是独立性最高的**

**Model不应该拥有对View的引用**，而且Model**也不需要保存对Presenter的引用**，对于Presenter而已，M**odel只需要提供接口，等着Presenter来调用时返回相应数据即可**，这和经典MVC模式中是非常不同的，在MVC中Model在数据发送变化后，是需要发送广播来告之View去更新用户界面的，而在MVP中，Model是不应该去通知View，而是通知Presenter来间接的更新View的。

**Presenter和Model的关系**

也应该是基于接口来通信，这样才能把Model和Presenter的耦合度也降到最低，那么在需要改变Model内部实现，甚至彻底替换Model的时候，Presenter则是无需随之改变的。这样做带来的另一个好处就是你可以通过Mock一个Model来对Presenter以及View做模拟测试了，从而提高了可测试性。

**Presenter和View的关系**

View是需要拥有对Presenter的引用，但仅仅是为了将用户的操作和事件立即传递给Presenter，为了让View和Presenter耦合较低，View也只应该通过接口与Presenter通信，从而保证View是完全被动的，一方面它由用户的操作触发来和Presenter通信，另一方面它完全受Presenter控制，唯一需要做的事情就是如何展示数据。

**简要总结三者之间的关系**

View和Model之间没有联系，View通过接口向Presenter来传递用户操作，Model不主动和Presenter联系，被动的等着Presenter来调用其接口，Presenter通过接口和View/Model来联系。

View <- 接口 <- Presenter ->接口 -> Model

View -> 接口 -> Presenter <- 接口 <- Model

## Talk is cheap, show me the code

**View**

**interface** IUserView {

**void** setPresenter(presenter);

**void** showUsers(users);

**void** **showDeleteUserComplete**();

**void** **showDeleteUserError**();

}

**class** **UserView** **implements** IUserView {

**UserPresenter** presenter;

**// 保持对Presenter的引用，用于路由用户操作**

**void** setPresenter(presenter) {

**this**.presenter = presenter;

}

**// 将Presenter传递来的数据展示出来**

**void** showUsers(users) {

draw(users);

}

**// Model操作数据成功后，通过Presenter来告之View需要更新用户界面**

**void** **showDeleteUserComplete**() {

alert("Delete User Complete");

}

**// Model操作数据失败后，也是通过Presenter来告之View需要更新用户界面**

**void** **showDeleteUserError**() {

alert("Delete User Fail");

}

**// 当用户点击某个按钮时，将用户操作路由给presenter，由presenter去处理**

**void** onDeleteButtonClicked(event) {

presenter.deleteUser(event);

}

}

**Model**

**interface** IUserModel {

**List**<**User**> **getUsers**();

**boolean** **deleteUserById**();

}

**class** **UserModel** **implements** IUserModel {

**// 在数据库里查找数据，并将数据返回给presenter**

**List**<**User**> **getUsers**() {

**return** getUsersInDatabase(id);

}

**// 在数据库里删除数据，并将结果返回给presenter**

User deleteUserById(id) {

**return** deleteUserByIdInDatabase(id);

}

}

**Presenter**

**interface** IUserUserPresenter {

**void** deleteUser(event);

}

**class** **UserUserPresenter** **implements** **IUserPresenter** {

**// 保持对View的引用**

IUserView view;

**// 保持对Model的引用**

IUserModel model;

**UserUserPresenter**(IUserView view, IUserModel model) {

**this**.view = view;

**this**.model = model;

**this**.view.setPresenter(**this**);

}

**void** start() {

**// 从Model中取出数据**

List<User> users = model.getUsers();

**// 将数据发送给View，让其展示到用户界面**

view.showUsers(users);

}

**void** deleteUser(event) {

**// View将用户操作路由过来，由Presenter来处理**

**long** uid = whichUserNeedToDeleteBy(event);

**// 将用户操作翻译成命令或消息传递给model，以改变数据**

**boolean** success = model.deleteUserById(uid);

**// 将Model操作数据后的结果通知View来改变用户界面**

**if** (success) {

view.onDeleteUserSuccess();

} **else** {

view.onDeleteUserFail();

}

}

}

# 使用MVP模式重构代码

## 一般的MVC里的 Controller 需要做的事情：

* 负责获取和处理用户的输入。
* 负责将输入传给负责业务逻辑层去做数据上的操作（如增删改查）。
* 负责将业务逻辑层对于数据操作的结果，传给View层去做展示。

因此如果完全按照这种定义的话，你应该很难看到一个非常臃肿的Controller，因为Controller在MVC模式中，本来就应该是很轻的，而不是很重的部分，重的应该是M层，甚至在前端交互复杂的时候，V层都应该比C层要重。我认为对于Controller的理解，就是一个站在M和V两者之间的一个翻译家，M来自地球，V来自火星。而如果站在中间的这个翻译者，话比他两的话还多，老是抢话，自言自语，这样显然是不合适的。

## Activity作为一个 Controller 到底都负责干什么了

1.首先activity必须去操作View的控件，设置它们的回调函数，有时也需要用代码去控制它们如何展示的属性。

2.然后activity一定需要去处理用户的输入，例如输入的值，以及点击事件等用户行为。

3.而几乎大部分异步网络请求都从activity发起，以及服务器返回数据的处理。

4.activity一般还需要根据数据的操作结果，负责在页面上将结果告之用户，例如Toast或者其他View的操作。

5.除此之外，activity还需要管理其生命周期相关的所有事务，例如在页面退出的时候处理一下View控件和其他与生命期相关的逻辑。

但如果你仔细的分析，你会发现activity不仅仅需要承担Controller的责任，还需要处理大量View的逻辑，例如控件的监听的属性，如何展示数据的职责也往往落到了它的肩上。更何况你很容易在activity写操作数据和网络请求的代码，也就是让它又承担了Model的责任，那么请问这样的Activity能不臃肿吗？

## Presenter是来给activity减负的吗？

很多人会认为MVP中引入Presenter的概念，是为了给日益臃肿的activity来减负的，而我不这样认为，我认为Presenter和Controller的责任是差不多的，它们后期承担的目的都其实很简单，就是用来隔离Model和View的，也就是常说的展示层和业务层的解藕。

那么该如何解决activity的问题呢？目前常见的MVP在Android里的实践有两种解决方案：

1.直接将Activity看作View，让它只承担View的责任。

2.将Activity看作一个MVP三者以外的一个Controller，只控制生命周期。

在Google推出的[官方MVP实例](https://github.com/googlesamples/android-architecture" \t "_blank)里，使用的就是第2种思路，它让每一个Activity都拥有一个Fragment来作为View，然后每个Activity也对应一个Presenter，在Activity里只处理与生命周期有关的内容，并跳出MVP之外，负责实例化Model，View，Presenter，并负责将三者合理的建立联系，承担的就是一个上帝视角。

在实践中，也有很多观点会简化掉Fragment，直接将Activity视为View，这个也是我比较赞同的，更简便一些，而且这样观念上也容易理解一些，你就把activity看作View的一部分，永远只让它处理展示的逻辑，不允许它去处理数据，和拥有业务逻辑。但是这样也有一个缺点，就是V和P的依赖关系不太规范了，理论上你是不应该在View里面去实例化Presenter和Model的，这其实是不合理的，正确的依赖关系，确实是应该在一个独立的更上层去实例化Model，View，Presenter的，这样依赖才是较为合理的关系，这点来看Google的架构模式确实更合理，但实操上也会麻烦一点，必须让每个activity拥有一个独立的fragment，这个我是觉得可以自由取舍，你是要概念上的合理，还是现实中的方便，其实都可以。

因为重点还是在于如何分离展示层和业务层，activity具体承担什么责任都可以，但只能承担一个责任。

## 一个LoginActivity的重构

**这里将Activity被视为View， 仅负责数据的展示，并且将用户的操作事件路由给P去做处理。**

**public** **interface** UserView {

**void** **showSuccesss**(**UserBean** userBean);

**void** **showError**(**String** errorString);

}

**public** **interface** UserPresenter {

**void** **login**(**String** name, **String** password);

}

**public** **interface** UserModel {

**void** **login**(**String** name, **String** password, MyCallBack callBack);

}

**public** **interface** MyCallBack {

**void** **success**(**UserBean** userBean);

**void** **error**(**String** errorString);

}

**public** **class** **MainActivity** **extends** **Activity** **implements** UserView{

**private** UserPresenter presenter;

@Override

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

**super**.onCreate(savedInstanceState);

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

presenter = **new** UserPresenterImpl(**new** UserModelImpl(), **this**);

findViewById(**R**.**id**.**tv**).setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

presenter.login("fasfd", "fasdfda");

}

});

}

@Override

**public** **void** **showSuccesss**(**UserBean** userBean) {

**Toast**.*makeText*(**MainActivity**.**this**, userBean.toString(), **Toast**.**LENGTH\_SHORT**).show();

}

@Override

**public** **void** **showError**(**String** errorString) {

**Toast**.*makeText*(**MainActivity**.**this**, errorString, **Toast**.**LENGTH\_SHORT**).show();

}

}

**public** **class** **UserPresenterImpl** **implements** UserPresenter{

**private** UserModel userModel;

**private** UserView userView;

**public** **UserPresenterImpl**(UserModel userModel, UserView userView) {

**this**.userModel = userModel;

**this**.userView = userView;

}

@Override

**public** **void** **login**(**String** name, **String** password) {

userModel.login(name, password, **new** MyCallBack() {

@Override

**public** **void** **success**(**UserBean** userBean) {

userView.showSuccesss(userBean);

}

@Override

**public** **void** **error**(**String** errorString) {

userView.showError(errorString);

}

});

}

}

**public** **class** **UserModelImpl** **implements** UserModel{

@Override

**public** **void** **login**(**String** name, **String** password, **final** MyCallBack callBack) {

**StringRequest** request = **new** StringRequest("www.baidu.com" + name + password, **new** Listener<**String**>() {

@Override

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

**UserBean** userBean = **new** UserBean(response, 1, response);

callBack.success(userBean);

}

}, **new** ErrorListener() {

@Override

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

callBack.error(error.toString());

}

});

}

}

## 从1个类变为3个类

在MVP的实践中，很明显的结构变化就是很多页面从1个类变成了3个甚至更多的类。

例如，原来只有一个 LoginActivity ，而现在会变成至少3个类：

LoginActivity(View)

LoginPresenterImpl (Presenter)

LoginModelImpl (Model)

而你以为这些就够了，就太天真的，在MVP里，为了解藕三者之间的关系，还需要通过接口来通信，P层是通过接口来和M层通信的，P层和V层之间也是通过接口来互相通信的（但V层对P层的通信被视为被动通信，而非主动通信）

接口列表：

LoginView (interface for View)

LoginPresenter (interface for Presenter)

LoginMode (interface for Model)

这里插一句题外话，在Google官方的MVP实例里的，有一个契约类的概念，这个契约类的概念引入我觉得真的很赞，其实它只是将View和Presenter的接口写到了一个类里面，但这样写则会使得读代码的人一目了然就可以了解这个页面需要展示些什么，有什么操作。

如果你以为MVP各一个接口这样就应该够了，我只能说你还是太年轻太天真。要知道很多消息在MVP三者之间传递，不仅仅是同步消息，还有很多异步消息，例如用户点击了一个按钮，View将该事件传递给Presenter，Presenter异步的向Model请求数据，Model异步的返回数据给Presenter，Presenter再将Model处理的结果异步的传递给View，让其向用户作出回应。

可想而之，这样异步操作，自然少不了一些Callback的接口类，虽然可以用内部类来解决，但如果不用范型的话，这些Callback的接口类数目还是很多的。

## 缺点

从直来直去变成跳来跳去

并且我认为这样的代码结构，甚至加大了调试的困难，过去直来直去，你很容易判断出数据是断在哪里，而现在你很难判断出数据断在哪一个层面，例如用户点击了刷新，需要从服务器拉回数据刷新到列表。但当页面没有正常展示数据的时候，你必须知道在哪个环节出错了，而我告诉你，因为分成了三个层，并且消息和数据在三个层之间传递，那么出错的可能性也变多了：

1、可能是View层没有把用户的事件传递给Presenter层。

2、Presenter层可能接受到View层事件，但没有将操作传递给Model层。

3、Model层可能接受到Presenter的请求，但没有将数据传回给Presenter层。

4、Presenter层可能接受到Model的返回值了，但没有正确的将数据传回给View层。

5、View层可能接受到了Presenter返回值，只是没有正确的将数据显示到页面而已。

在调试的时候，你会发现，跟踪一个问题变复杂了，消息在MVP之间传来传去的，你很难一下定位到问题出在哪个层面。

## 那么为什么要用MVP？

说了这么MVP带来的麻烦，例如多写了很多类，思维跳来跳去，消息传来传去，层层回调把人转晕，那回头去思考：我们为什么要用MVP，为什么要这样拆分代码，不是说这样代码更清晰，更容易理解了吗，为什么我看不懂我的代码了，为什么调试起来如何麻烦？

其实，这样我要反复说的，如果你只是学会怎么使用MVP，那么你只是换了一个架构而已，这就和你换了一个IDE写代码，却期望换了IDE就可以让代码突然变的更好一样。而你真正需要做的，依然是我之前说过的：

你需要换的是脑子，而不是架构。

如果你还在每次修改一行代码，就整体去测试你的系统，那么你把代码写在一个文件里，还是拆分到几个文件里，其实是没有区别的。你只是把代码拆开在放，而这样的拆注定只是形式的，最终我相信写着写着，你会在View里面写Model的逻辑，在Model里面写View的逻辑，并且和过去一样，Presenter越来越臃肿。

为什么要把架构里的各个层次分得清清楚楚，每个层面负责什么，不应该负责负责，如何组合起来都需要严格的定义起来，你要知道，每一种架构都不是编码规范，也不是组织代码的规范，它们都是一种思维方式。

之前说过，良好的架构都是在解决几个问题：低藕合，高复用，易测试，好维护。

如果你还在你的类和类之间new来new去，你引用我，我引用你，互相依赖，层层依赖，那么你把它们写在一个文件里，和把它们几个文件里有区别吗？

如果你的一个类还承担多个职责，明明这是个叫 Car 的类，却又在承担轮子，又在承担引擎的责任，那么你抽象和不抽像，封装不封装真的有区别吗？

如果你的一个方法还在做两件甚至三件事情，甚至把一整套事情都做完了，动辄超过几屏的函数，那么你真的觉得用不用架构真的有区别吗？

# MVC模式

## MVC (Model-View-Controller)：

MVC是一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。其中M层处理数据，业务逻辑等；V层处理界面的显示结果；C层起到桥梁的作用，来控制V层和M层通信以此来达到分离视图显示和业务逻辑层使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式，而C存在的目的则是确保M和V的同步，一旦M改变，V应该同步更新，这与《设计模式》中的观察者模式是完全一样。

## MVC好处：

1. 耦合性低。所谓耦合性就是模块代码之间的关联程度。利用MVC框架使得View（视图）层和Model（模型）层可以很好的分离，这样就达到了解耦的目的，所以耦合性低，减少模块代码之间的相互影响。
2. 可扩展性好。由于耦合性低，添加需求，扩展代码就可以减少修改之前的代码，降低bug的出现率。
3. 模块职责划分明确。主要划分层M,V,C三个模块，利于代码的维护。
4. MVC把应用程序的逻辑层与界面是完全分开的界面设计人员可以直接参与到界面开发，程序员就可以把精力放在逻辑层上。

## 什么时候适合使用MVC设计模式？

当然一个小的项目且无需频繁修改需求就不用MVC框架来设计了，那样反而觉得代码过度设计，代码臃肿。一般在大的项目中，且业务逻辑处理复杂，页面显示比较多，需要模块化设计的项目使用MVC就有足够的优势了。

## 在MVC模式中我们发现

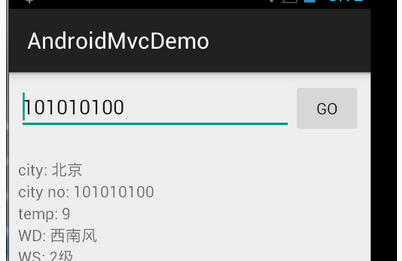
其实控制器Activity主要是起到解耦作用，将View视图和Model模型分离，虽然Activity起到交互作用，但是找Activity中有很多关于视图UI的显示代码，因此View视图和Activity控制器并不是完全分离的，也就是说一部分View视图和Contronller控制器Activity是绑定在一个类中的。

## 在Android中：

1) 视图层（View）：一般采用XML文件进行界面的描述，使用的时候可以非常方便的引入。当然，如何你对Android了解的比较的多了话，就一定可以想到在Android中也可以使用JavaScript+HTML等的方式作为View层，当然这里需要进行Java和JavaScript之间的通信，幸运的是，Android提供了它们之间非常方便的通信实现。

2) 控制层（Controller）：Android的控制层的重任通常落在了众多的Acitvity的肩上，这句话也就暗含了不要在Acitivity中写代码，要通过Activity交割Model业务逻辑层处理，这样做的另外一个原因是Android中的Acitivity的响应时间是5s，如果耗时的操作放在这里，程序就很容易被回收掉。

3) 模型层（Model）：对数据库的操作、对网络等的操作都应该在Model里面处理，当然对业务计算等操作也是必须放在的该层的。就是应用程序中二进制的数据。

## MVC for Android

M层：适合做一些业务逻辑处理，比如数据库存取操作，网络操作，复杂的算法，耗时的任务等都在model层处理。

V层：应用层中处理数据显示的部分，XML布局可以视为V层，显示Model层的数据结果。

C层：在Android中，Activity处理用户交互问题，因此可以认为Activity是控制器。

Activity读取V视图层的数据（eg.读取当前EditText控件的数据），控制用户输入（eg.EditText控件数据的输入），并向Model发送数据请求（eg.发起网络请求等）。接下来我们通过一个获取天气预报数据的小项目来解读 MVC for Android。上面是界面图：

**public** **class** **MainActivity** **extends** **AppCompatActivity** **implements** View.**OnClickListener** {

**private** **WeatherModel** weatherModel;

**private** **Dialog** loadingDialog;

**private** **EditText** cityNOInput;

@Override

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

**super**.onCreate(savedInstanceState);

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

weatherModel = **new** **WeatherModeImpl**();

initView();

}

**private** **void** **initView**() {

time = (**TextView**) findViewById(**R**.**id**.tv\_time);

njd = (**TextView**) findViewById(**R**.**id**.tv\_njd);

findViewById(**R**.**id**.btn\_go).setOnClickListener(**this**);

loadingDialog = **new** **ProgressDialog**(**this**);

loadingDialog.setTitle("加载天气中...");

}

@Override

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

loadingDialog.show();

**if** (!TextUtils.isEmpty(cityNOInput.getText().toString().trim())) {

weatherModel.getWeather(cityNOInput.getText().toString().trim(), **new** **OnWeatherListener**() {

@Override

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

hideDialog();

disPlayWeather(weather);

}

@Override

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

hideDialog();

Log.e("tag", volleyError.toString());

Toast.makeText(**MainActivity**.**this**, "请求失败", Toast.LENGTH\_SHORT).show();

}

});

}

}

**private** **void** **disPlayWeather**(**Weather** weather) {

**Weather**.**weatherinfo** Info = weather.getWeatherinfo();

**if** (Info!= **null** && Info.getCity() != **null**) {

city.setText(Info.getCity());

cityNO.setText(Info.getCityid());

temp.setText(Info.getTemp());

}

}

**private** **void** **hideDialog**() {

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

loadingDialog.dismiss();

}

}

}

从上面代码可以看到，Activity持有了WeatherModel模型的对象，当用户有点击Button交互的时候，Activity作为Controller控制层读取View视图层EditTextView的数据，然后向Model模型发起数据请求，也就是调用WeatherModel对象的方法 getWeathre（）方法。当Model模型处理数据结束后，通过接口OnWeatherListener通知View视图层数据处理完毕，View视图层该更新界面UI了。然后View视图层调用displayResult（）方法更新UI。至此，整个MVC框架流程就在Activity中体现出来了。

**public** **interface** OnWeatherListener {

**void** **onSuccess**(**Weather** weather);

**void** **OnError**(**VolleyError** volleyError);

}

**public** **interface** WeatherModel {

**void** **getWeather**(**String** countryNum, **OnWeatherListener** listener);

}

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

@Override

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

**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);

}

}

以上代码看出，这里设计了一个WeatherModel模型接口，然后实现了接口WeatherModelImpl类。controller控制器activity调用WeatherModelImpl类中的方法发起网络请求，然后通过实现OnWeatherListener接口来获得网络请求的结果通知View视图层更新UI 。至此，Activity就将View视图显示和Model模型数据处理隔离开了。activity担当contronller完成了model和view之间的协调作用。

至于这里为什么不直接设计成类里面的一个getWeather（）方法直接请求网络数据？你考虑下这种情况：现在代码中的网络请求是使用Volley框架来实现的，如果哪天老板非要你使用Afinal框架实现网络请求，你怎么解决问题？难道是修改 getWeather（）方法的实现？ no no no，这样修改不仅破坏了以前的代码，而且还不利于维护， 考虑到以后代码的扩展和维护性，我们选择设计接口的方式来解决着一个问题，我们实现另外一个WeatherModelWithAfinalImpl类，继承自WeatherModel，重写里面的方法，这样不仅保留了以前的WeatherModelImpl类请求网络方式，还增加了WeatherModelWithAfinalImpl类的请求方式。Activity调用代码无需要任何修改。

# MVP模式

## MVP

MVP 是从经典的模式MVC演变而来，它们的基本思想有相通的地方：Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。

## MVC和MVP的区别？

在MVP中View并不直接使用Model，它们之间的通信是通过Presenter (MVC中的Controller)来进行的，所有的交互都发生在Presenter内部，而在MVC中View会直接从Model中读取数据而不是通过 Controller。

MVC模型里，更关注的Model的不变，而同时有多个对Model的不同显示，即View。所以，在MVC模型里，Model不依赖于View，但是View是依赖于Model的。不仅如此，因为有一些业务逻辑在View里实现了，导致要更改View也是比较困难的，至少那些业务逻辑是无法重用的。

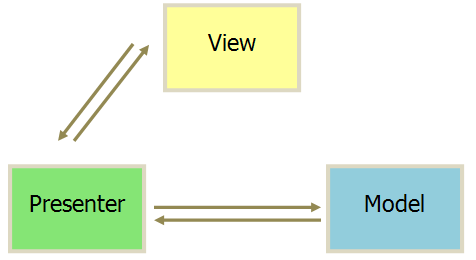
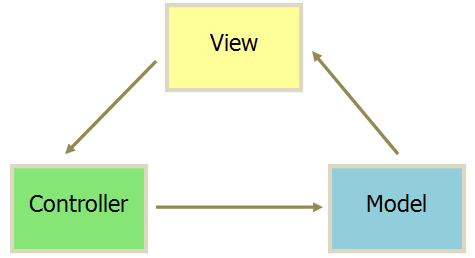
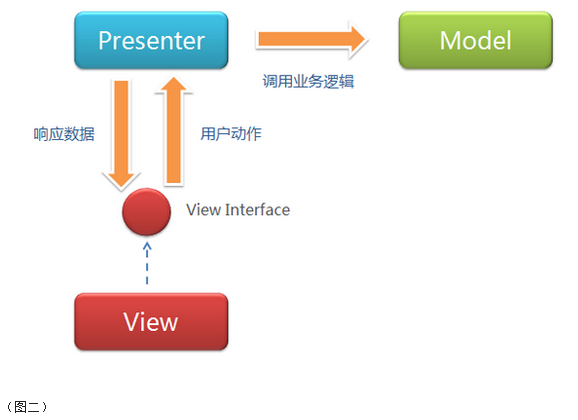
在MVP里，Presenter完全把Model和View进行了分离，主要的程序逻辑在Presenter里实现。而且，Presenter与具体的View是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更View时候可以保持Presenter的不变，即重用！ 在MVP模式里，View只应该有简单的Set/Get的方法，用户输入和设置界面显示的内容，除此就不应该有更多的内容，绝不容许直接访问Model--这就是与MVC很大的不同之处。

## MVP的优点

1. 模型与视图完全分离，我们可以修改视图而不影响模型
2. 可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter内部
3. 我们可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁。
4. 如果我们把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）

## MVP的缺点

由于对视图的渲染放在了Presenter中，所以视图和Presenter的交互会过于频繁。还有一点需要明白，如果Presenter过多地渲染了视图，往往会使得它与特定的视图的联系过于紧密。一旦视图需要变更，那么Presenter也需要变更了。比如说，原本用来呈现Html的Presenter现在也需要用于呈现Pdf了，那么视图很有可能也需要变更。

# 这个VP（注意是VP）泛型有点懵

## View

**public** **interface** MvpView {

}

**public** **interface** MainMvpView **extends** MvpView {

**void** **showImages**(List<**String**> imgs);

**void** **updateLinkView**(**boolean** isConnect);

**void** **updateDeviceId**();

}

**public** **class** **MainActivity implements** MainMvpView{

**private** **MainPresenter** mPresenter = **new** MainPresenter();

@Override

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

**super**.onCreate(savedInstanceState);

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

mPresenter.attachView(**this**);

mPresenter.setConnectListener();

mPresenter.loadImages();

}

@Override

**public** **void** **showImages**(List<**String**> imgs) {

img.setImageResource(imgs.get(i));

}

@Override

**public** **void** **updateLinkView**(**boolean** isConnect) {

**NetworkUtil** util = **new** NetworkUtil(**this**);

**if**(!util.isNetworkConnected()){

offLineAutoRefresh = **true**;

}**else**{

offLineAutoRefresh = **false**;

}

}

@Override

**public** **void** **updateDeviceId**() {

**AgentUtil** mAgentUtil = **AgentUtil**.*getInstance*();

mDeviceInfo.setText("设备：" + mId);

}

}

## Presenter

**public** **interface** Presenter<V **extends** MvpView> {

**void** **attachView**(V mvpView);

**void** **detachView**();

}

**public** **class** **BasePresenter**<T **extends** MvpView> **implements** Presenter<T> {

**private** T mMvpView;

@Override

**public** **void** **attachView**(T mvpView) {

mMvpView = mvpView;

}

@Override

**public** **void** **detachView**() {

mMvpView = **null**;

}

**public** **boolean** **isViewAttached**() {

**return** mMvpView != **null**;

}

**public** T **getMvpView**() {

**return** mMvpView;

}

**public** **void** **checkViewAttached**() {

**if** (!isViewAttached()) **throw** **new** MvpViewNotAttachedException();

}

**public** **static** **class** **MvpViewNotAttachedException** **extends** **RuntimeException** {

**public** **MvpViewNotAttachedException**() {

**super**("Please call Presenter.attachView(MvpView) before" + " requesting data to the Presenter");

}

}

}

**public** **class** **MainPresenter** **extends** **BasePresenter**<MainMvpView> {

@Override

**public** **void** **attachView**(MainMvpView mvpView) {

**super**.attachView(mvpView);

}

@Override

**public** **void** **detachView**() {

**super**.detachView();

}

**public** **void** **loadImages**() {

List<**String**> pathList = **new** ArrayList<**String**>();

pathList.add("/sdcard/filedemo/ss\_pic\_2.jpg");

getMvpView().showImages(pathList);

}

**private** **void** **updateLinkView**(){

**new** Handler(**Looper**.*getMainLooper*()).post(**new** Runnable() {

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

getMvpView().updateLinkView(isConnect);

}

});

}

**public** **void** **setConnectListener**() {

**final** ConnectListener listener = **new** ConnectListener() {

@Override

**public** **void** **onConnectState**(**boolean** isSuccese) {

isConnect = isSuccese;

updateLinkView();

}

};

}

}

# 如何向妻子解释OOD

## OOD简介

Shubho:亲爱的，让我们开始学习OOD吧。你了解面向对象原则吗？

Farhana:你是说封装，继承，多态对吗？我知道的。

Shubho:好，我希望你已了解如何使用类和对象。今天我们学习OOD。

Farhana:等一下。面向对象原则对面向对象编程(OOP)来说不够吗？我的意思是我会定义类，并封装属性和方法。我也能根据类的关系定义它们之间的层次。如果是，那么还有什么？

Shubho:问得好。面向对象原则和OOD实际上是两个不同的方面。让我给你举个实际生活中的例子帮你弄明白。再你小时候你首先学会字母表，对吗？

Farhana:嗯

Shubho:好。你也学了单词，并学会如何根据字母表造词。后来你学会了一些造句的语法。例如时态，介词，连词和其他一些让你能造出语法正确的句子。例如："I" (代词) "want" (动词) "to" (介词) "learn" (动词) "OOD"(名词)。看，你按照某些规则组合了单词，并且你选择了有某些意义的正确的单词结束了句子。

Farhana:OK，这意味着什么呢？

Shubho:面向对象原则与这类似。OOP指的是面向对象编程的基本原则和核心思路。在这里，OOP可以比作英语基础语法，这些语法教你如何用单词构造有意义且正确的句子，OOP教你在代 码中构造类，并在类里封装属性和方法，同时构造他们之间的层次关系。

Farhana:嗯..我有点感觉了，这里有OOD吗？

Shubho:马上就有答案。现在假定你需要就某些主题写几篇文章或随笔。你也希望就几个你擅长主体写几本书。对写好文章/随笔或书来说，知道如何造句是不够的，对吗？为了使读者能更轻 松的明白你讲的内容，你需要写更多的内容，学习以更好的方式解释它。

Farhana:看起来有点意思...继续。

Shubho:现在，如果你想就某个主题写一本书，如学习OOD，你知道如何把一个主题分为几个子主题。你需要为这些题目写几章内容，也需要在这些章节中写前言，简介，例子和其他段落。 你需要为写个整体框架，并学习一些很好的写作技巧以便读者能更容易明白你要说的内容。这就是整体规划。在软件开发中，OOD是整体思路。在某种程度上，设计软件时，你的类和代码需能达到模块化，可复用，且灵活，这些很不错的指导原则不用你重新发明创造。确实有些原则你已经在你的类和对象中已经用到了，对吗？

Farhana:嗯...有个大概的印象了，但需要继续深入。

Shubho:别担心，你马上就会学到。我们继续讨论下去。

## 为什么要OOD？

Shubho:这是一个非常重要的问题。当我们能很快地设计一些类，完成开发并发布时，为什么我们需要关心OOD？那样子还不够吗？

Farhana:嗯，我早先并不知道OOD，我一直就是开发并发布项目。那么关键是什么？

Shubho:好的，我先给你一句名言：走在结冰的河边不会湿鞋，开发需求不变的项目畅通无阻(Walking on water and developing software from a specification are easy if both are frozen)——Edward V. Berard

Farhana:你的意思是软件开发说明书会不断变化？

Shubho:非常正确！软件开发唯一的真理是“软件一定会变化”。为什么?因为你的软件解决的是现实生活中的业务问题，而现实生活中得业务流程总是在不停的变化。假设你的软件在今天工作的很好。但它能灵活的支持“变化”吗？如果不能，那么你就没有一个设计敏捷的软件。

Farhana:好，那么请解释一下“设计敏捷的软件”。

Shubho:"一个设计敏捷的软件能轻松应对变化，能被扩展，并且能被复用。"并且应用好"面向对象设计"是做到敏捷设计的关键。那么，你什么时候能说你在代码中很好的应用了OOD？

Farhana:这正是我的问题。

Shubho:如果你代码能做到以下几点，那么你就正在OOD：面向对象、复用、能以最小的代价满足变化、不用改变现有代码满足扩展

Farhana：还有？

Shubho:我们并不是孤立的。很多人在这个问题上思考了很多，也花费了很大努力，他们试图做好OOD，并为OOD指出几条基本的原则(那些灵感你能用之于你的OOD)。他们最终也确实总结出了一些通用的设计模式(基于基本的原则)。

Farhana:你能说几个吗？

Shubho:当然。这里有很多涉及原则，但最基本的是叫做SOLID的5原则(感谢Uncle Bob,伟大OOD导师)。

S = 单一职责原则 Single Responsibility Principle

O = 开放闭合原则 Opened Closed Principle

L = Liscov替换原则 Liscov Substitution Principle

I = 接口隔离原则 Interface Segregation Principle

D = 依赖倒置原则 Dependency Inversion Principle

接下去，我们会仔细探讨每一个原则。

### 单一职责原则

Shubho:我先给你展示一张海报。我们应当谢谢做这张海报的人，它非常有意思。



单一职责原则海报

它说："并不是因为你能，你就应该做"。为什么？因为长远来看它会带来很多管理问题。

从面向对象角度解释为："引起类变化的因素永远不要多于一个。"

或者说"一个类有且只有一个职责"。

Farhana：能解释一下吗？

Shubho:当然，这个原则是说，如果你的类有多于一个原因会导致它变化(或者多于一个职责)，你需要一句它们的职责把这个类拆分为多个类。

Farhana:嗯...这是不是意味着在一个类里不能有多个方法？

Shubho:不。你当然可以在一个类中包含多个方法。问题是，他们都是为了一个目的。如今为什么拆分是重要的？

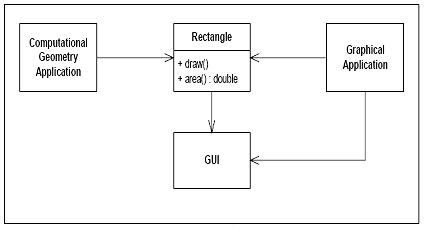
那是因为：

每个职责是轴向变化的；

如果类包含多个职责，代码会变得耦合；

Farhana:能给我一个例子吗？

Shubho:当然，看一下下面的类层次。当然这个例子是从Uncle Bob那里得来，再谢谢他。



违反单一职责原则的类结构图

这里，Rectangle类做了下面两件事：

计算矩形面积；

在界面上绘制矩形；

并且，有两个应用使用了Rectangle类：

计算几何应用程序用这个类计算面积；

图形程序用这个类在界面上绘制矩形；

这违反了SRP(单一职责原则);

Farhana:如何违反的？

Shubho:你看，Rectangle类做了两件事。在一个方法里它计算了面积，在另外一个方法了它返回一个表示矩形的GUI。这会带来一些有趣的问题：

在计算几何应用程序中我们必须包含GUI。也就是在开发几何应用时，我们必须引用GUI库；

图形应用中Rectangle类的变化可能导致计算几何应用变化，编译和测试，反之亦然；

Farhana:有点意思。那么我猜我们应该依据职责拆分这个类，对吗？

Shubho:非常对，你猜我们应该做些什么？

Farhana:当然，我试试。下面是我们可能要做的：

拆分职责到两个不同的类中，如：

Rectangle:这个类应该定义Area()方法；

RectangleUI:这个类应继承Rectangle类，并定义Draw()方法。

Shubho:非常好。在这里，Rectangle类被计算几何应用使用，而RectangleUI被图形应用使用。我们甚至可以分离这些类到两个独立的DLL中，那会允许我们在变化时不需要关心另一个就可以实现它。

Farhana:谢谢，我想我明白SRP了。SRP看起来是把事物分离成分子部分，以便于能被复用和集中管理。我们也不能把SRP用到方法级别吗？我的意思是，我们可以写一些方法，它们包含做很多事的代码。这些方法可能违反SRP，对吗？

Shubho:你理解了。你应当分解你的方法，让每个方法只做某一项工作。那样允许你复用方法，并且一旦出现变化，你能够以修改最少的代码满足变化。

### 开放闭合原则

Shubho:这里是开放闭合原则的海报



开放闭合原则海报

从面向对象设计角度看,它可以这么说:"软件实体(类,模块,函数等等)应当对扩展开放，对修改闭合。"

通俗来讲，它意味着你应当能在不修改类的前提下扩展一个类的行为。就好像我不需要改变我的身体而可以穿上衣服。

Farhana:有趣。你能够按照你意愿穿上不同的衣服来改变面貌，而从不用改造身体。你对扩展开放了，对不？

Shubho:是的。在OOD里，对扩展开发意味着类或模块的行为能够改变，在需求变化时我们能以新的，不同的方式让模块改变，或者在新的应用中满足需求。

Farhana:并且你的身体对修改是闭合的。我喜欢这个例子。当需要变化时，核心类或模块的源代码不应当改动。你能用些例子解释一下吗？

Shubho:当然，看下面这个例子。它不支持"开放闭合"原则。

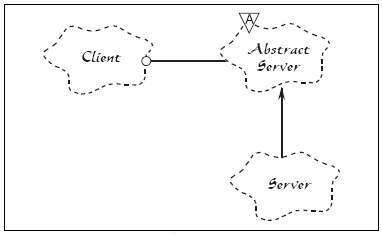


违反开发闭合原则的类结构

你看，客户端和服务段都耦合在一起。那么，只要出现任何变化，服务端变化了，客户端一样需要改变。

Farhana:理解。如果一个浏览器以紧耦合的方式按照指定的服务器(比如IIS)实现，那么如果服务器因为某些原因被其他服务器(如Apache)替换了，那么浏览器也需要修改或替换。这确实很可怕！

Shubho:对的。下面是正确的设计。



遵循开放闭合原则的类结构

在这个例子中，添加了一个抽象的服务器类，客户端包含一个抽象类的引用，具体的服务类实现了抽象服务类。那么，因任何原因引起服务实现发生变化时，客户端都不需要任何改变。

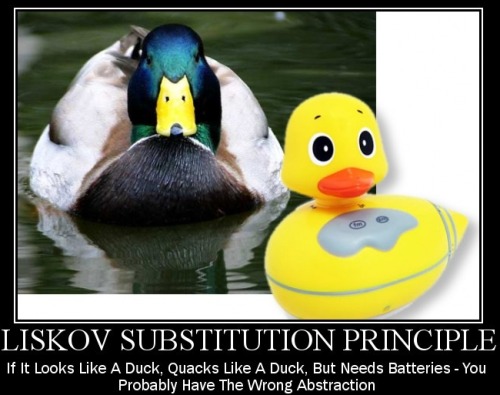
这里抽象服务类对修改是闭合的，实体类的实现对扩展是开放的。

Farhana:我明白了，抽象是关键，对吗？

Shubho:是的，基本上，你抽象的东西是你系统的核心内容，如果你抽象的好，很可能在扩展功能时它不需要任何修改(就像服务是一个抽象概念)。如果在实现里定义了抽象的东西(比如IIS服务器实现的服务)，代码要尽可能以抽象(服务)为依据。这会允许你扩展抽象事物，定义一个新的实现(如Apache服务器)而不需要修改任何客户端代码。

### Liskov's 替换原则

Shubho:"Liskov's替换原则(LSP)"听起来很难，却是很有用的基本概念。看下这幅有趣的海报：



Liskov替换原则海报

这个原则意思是："子类型必须能够替换它们基类型。"

或者换个说法："使用基类引用的函数必须能使用继承类的对象而不必知道它。"

Farhana:不好意思，听起来有点困惑。我认为这个OOP的基本原则之一。也就是多态，对吗？为什么一个面向对象原则需要这么说呢？

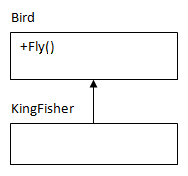
Shubho:问的好。这就是你的答案：

在基本的面向对象原则里，"继承"通常是"is a"的关系。如果"Developer" 是一个"SoftwareProfessional",那么"Developer"类应当继承"SoftwareProfessional"类。在类设计中"Is a"关系非常重要，但它容易冲昏头脑，结果使用错误的继承造成错误设计。

"Liskov替换原则"正是保证继承能够被正确使用的方法。

Farhana:我明白了。有意思。

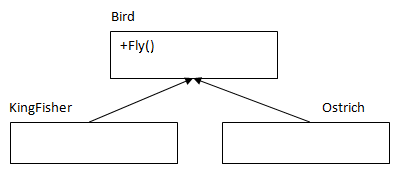
Shubho:是的，亲爱的，确实。我们看个例子：



Liskov替换原则类结构图

这里，KingFisher类扩展了Bird基类，并继承了Fly()方法，这看起来没问题。

现在看下面的例子：



违反Liskov替换原则类结构图

Ostrich(鸵鸟)是一种鸟(显然是),并从Bird类继承。它能飞吗？不能，这个设计就违反了LSP。

所以，即使在现实中看起来没问题，在类设计中，Ostrich不应该从Bird类继承，这里应该从Bird中分离一个不会飞的类，Ostrich应该继承与它。

Farhana:好，明白了。那么让我来试着指出为什么LSP这么重要：

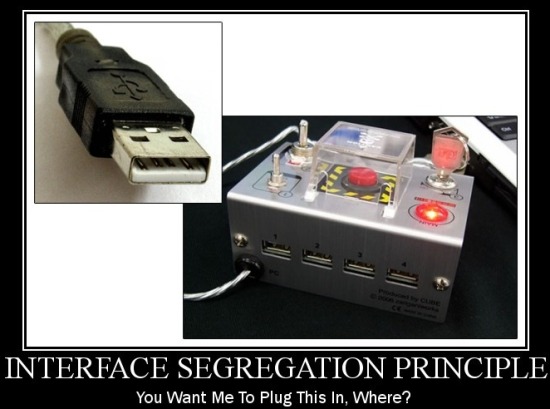
如果没有LSP，类继承就会混乱；如果子类作为一个参数传递给方法，将会出现未知行为；

如果没有LSP，适用与基类的单元测试将不能成功用于测试子类；对吗？

Shubho:非常正确。你能设计对象，使用LSP做为一个检查工作来测试继承是否正确。

### 接口分离原则

Shubho:今天我们学习"接口分离原则",这是海报：



接口分离原则海报

Farhana:这是什么意思？

Shubho:它的意思是："客户端不应该被迫依赖于它们不用的接口。"

Farhana:请解释一下。

Shubho:当然，这是解释：假设你想买个电视机，你有两个选择。一个有很多开关和按钮，它们看起来很混乱，且好像对你来说没必要。另一个只有几个开关和按钮，它们很友好，且适合你使用。假定两个电视机提供同样的功能，你会选哪一个？

Farhana:当然是只有几个开关和按钮的第二个。

Shubho:对，但为什么？

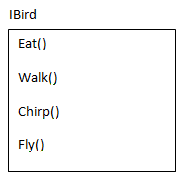
Farhana:因为我不需要那些看起来混乱又对我没用的开关和按钮。

Shubho:以便外部能够知道这些类有哪些可用的功能,客户端代码也能根据接口来设计.现在,如果接口太大,包含很多暴露的方法,在外界看来会很混乱.接口包含太多的方法也使其可用性降低,像这种包含了无用方法的"胖接口"会增加类之间的耦合.你通过接口暴露类的功能,对.同样地,假设你有一些类,这也引起了其他问题.如果一个类想实现该接口,那么它需要实现所有的方法,尽管有些对它来说可能完全没用.所以说这么做会在系统中引入不必要的复杂度,降低可维护性或鲁棒性。接口隔离原则确保实现的接口有他们共同的职责,它们是明确的,易理解的,可复用的.

Farhana:你的意思是接口应该仅包含必要的方法,而不该包含其它的.我明白了.

Shubho:非常正确.一起看个例子.

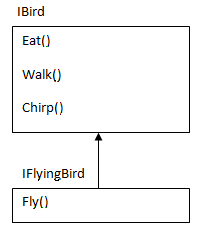
下面是违反接口隔离原则的一个胖接口



注意到IBird接口包含很多鸟类的行为,包括Fly()行为.现在如果一个Bird类(如Ostrich)实现了这个接口，那么它需要实现不必要的Fly()行为(Ostrich不会飞).

Farhana:确实如此。那么这个接口必须拆分了？

Shubho:是的。这个"胖接口"应该拆分未两个不同的接口，IBird和IFlyingBird,IFlyingBird继承自IBird.



这里如果一种鸟不会飞(如Ostrich)，那它实现IBird接口。如果一种鸟会飞(如KingFisher)，那么它实现IFlyingBird.

Farhana:所以回头看包含了很多开关和按钮的电视机的例子，电视机制造商应该有一个电视机的图纸，开关和按钮都在这个方案里。不论任何时候，当他们向制造一种新款电视机时，如果他们想复用这个图纸，他们将需要在这个方案里添加更多的开关和按钮。那么他们将没法复用这个方案，对吗？

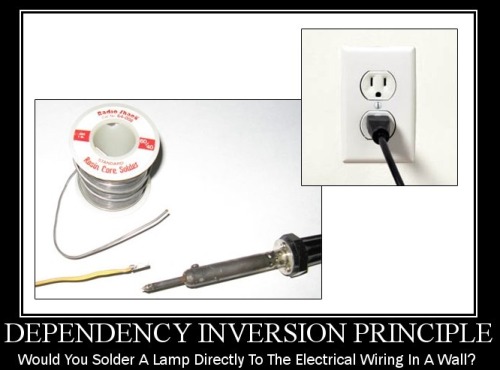
Shubho:对的。

Farhana:如果他们确实需要复用方案，它们应当把电视机的图纸份为更小部分，以便在任何需要造新款电视机的时候复用这点小部分。

Shubho:你理解了。

### 依赖倒置原则

Shubho:这是SOLID原则里最后一个原则。这是海报



它的意思是：高层模块不应该依赖底层模块，两者都应该依赖其抽象

Shubho:考虑一个现实中的例子。你的汽车是由很多如引擎，车轮，空调和其它等部件组成，对吗？

Farhana:是的

Shubho:好，它们没有一个是严格的构建在一个单一单元里；换句话说，它们都是可插拔的，因此当引擎或车轮出问题时，你可以修理它(而不需要修理其它部件),甚至可以换一个。在替换时，你仅需要确保引擎或车轮符合汽车的设计(如汽车能使用任何1500CC的引擎或任何18寸的车轮)。当然，汽车也可能允许你在1500CC引擎的地方安装一个2000CC的引擎，事实上对某些制造商(如丰田汽车)是一样的。现在，如果你的汽车的零部件不具备可插拔性会有什么不同？

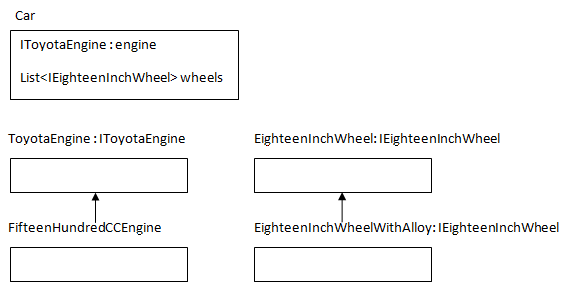
Farhana:那会很可怕！因为如果汽车的引擎出故障了，你可能修理整部车或者需要买一个新的。

Shubho:是的，那么该如何做到"可插拔性"呢？

Farhana:这里抽象是关键，对吗？

Shubho:是的，在现实中，汽车是高级模块或实体，它依赖于低级模块或实体，如引擎或车轮。相比直接依赖于引擎或车轮，汽车应依赖于某些抽象的有规格的引擎或车轮，以便于如果任何引擎或车轮符合抽象，那么它们都能组合到汽车中，汽车也能跑动。

一起看下面的类图



Shubho:注意到上面Car类有两个属性，它们都是抽象类型(接口)。引擎和车轮是可插拔的，因为汽车能接受任何实现了声明接口的对象，并且Car类不需要做任何改动。

Farhana:所以，如果代码中不用依赖倒置，我们将面临如下风险：

使用低级类会破环高级代码；

当低级类变化时需要很多时间和代价来修改高级代码；

产生低复用的代码；

Shubho:你完全掌握了，亲爱的！

### 总结

Shubho：除SOLID原则外还有很多其它的面向对象原则。如：

"组合替代继承":这是说相对于继承，要更倾向于使用组合；

"笛米特法则"：这是说"你的类对其它类知道的越少越好"；

"共同封闭原则"：这是说"相关类应该打包在一起"；

"稳定抽象原则"：这是说"类越稳定，越应该由抽象类组成";

Farhana:我应该学习那些原则吗？

Shubho:当然可以。你可以从整个网上学习。仅仅需要Google一下那些原则，然后尝试理解它。当然如果有需要，尽管问我。

Farhana:在那些设计原则之上我听说过很多设计模式。

Shubho:对的。设计模式只是对一些经常出现的场景的一些通用设计建议。这些灵感主要来自于面向对象原则。你可以把设计模式看作"框架",把OOD原则看作"规范".

Farhana:那么接下去我将学习设计模式吗？

Shubho:是的，亲爱的。

Farhana:那会很有意思，对吗？

Shubho:是，那确实令人兴奋。

# 进程保活大全

Android 进程拉活包括两个层面：

A. 提供进程优先级，降低进程被杀死的概率

B. 在进程被杀死后，进行拉活

## 进程的优先级

Android 系统将尽量长时间地保持应用进程，但为了新建进程或运行更重要的进程，最终需要清除旧进程来回收内存。 为了确定保留或终止哪些进程，系统会根据进程中正在运行的组件以及这些组件的状态，将每个进程放入“重要性层次结构”中。必要时，系统会首先消除重要性最低的进程，然后是清除重要性稍低一级的进程，依此类推，以回收系统资源。

**进程的重要性，划分5级：**

1. 前台进程(Foreground process)

2. 可见进程(Visible process)

3. 服务进程(Service process)

4. 后台进程(Background process)

5. 空进程(Empty process)

**前台进程的重要性最高，依次递减，空进程的重要性最低，下面分别来阐述每种级别的进程**

### 前台进程 —— Foreground process

用户当前操作所必需的进程。通常在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。

A. 拥有用户正在交互的 Activity（已调用 onResume()）

B. 拥有某个 Service，后者绑定到用户正在交互的 Activity

C. 拥有正在“前台”运行的 Service（服务已调用 startForeground()）

D. 拥有正执行一个生命周期回调的 Service（onCreate()、onStart() 或 onDestroy()）

E. 拥有正执行其 onReceive() 方法的 BroadcastReceiver

### 可见进程 —— Visible process

没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。可见进程被视为是极其重要的进程，除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

A. 拥有不在前台、但仍对用户可见的 Activity（已调用 onPause()）。

B. 拥有绑定到可见（或前台）Activity 的 Service

### 服务进程 —— Service process

尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

A. 正在运行 startService() 方法启动的服务，且不属于上述两个更高类别进程的进程。

### 后台进程 —— Background process

后台进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。 通常会有很多后台进程在运行，因此它们会保存在 LRU 列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。如果某个 Activity 正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该 Activity 时，Activity 会恢复其所有可见状态。

A. 对用户不可见的 Activity 的进程（已调用 Activity的onStop() 方法）

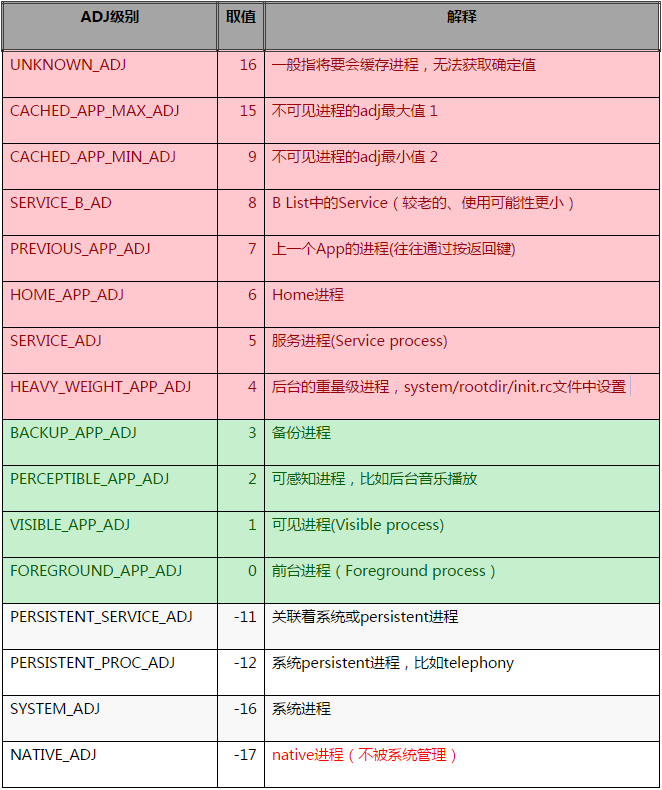
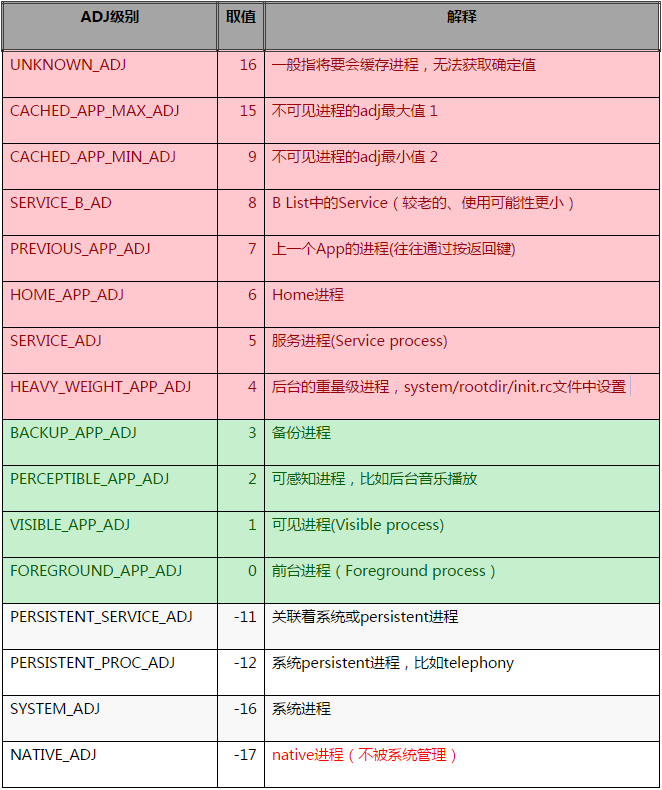
### 空进程 —— Empty process

保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

1. 不含任何活动应用组件的进程

## Android 进程回收策略

Android 中对于内存的回收，主要依靠 Lowmemorykiller 来完成，是一种根据 OOM\_ADJ 阈值级别触发相应力度的内存回收的机制。关于 OOM\_ADJ 的说明如下：

其中红色部分代表比较容易被杀死的 Android 进程（OOM\_ADJ>=4）,绿色部分表示不容易被杀死的 Android 进程，其他表示非 Android 进程（纯 Linux 进程）。在 Lowmemorykiller 回收内存时会根据进程的级别优先杀死 OOM\_ADJ 比较大的进程，对于优先级相同的进程则进一步受到进程所占内存和进程存活时间的影响。Android 手机中进程被杀死可能有如下情况：



综上，可以得出减少进程被杀死概率无非就是想办法提高进程优先级，减少进程在内存不足等情况下被杀死的概率。

## 提升进程优先级的方案

### 利用 Activity 提升权限

#### 方案设计思想

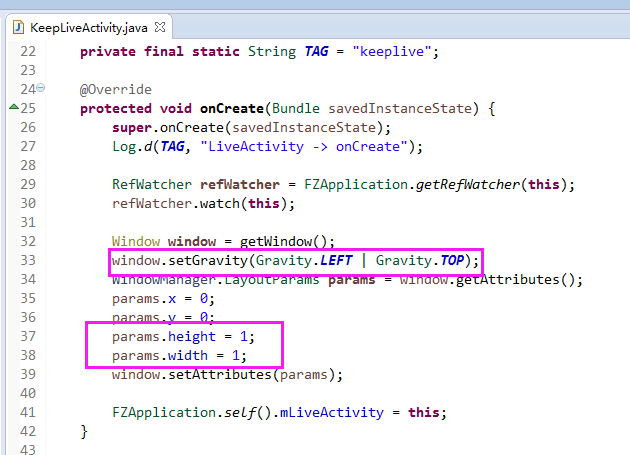
监控手机锁屏解锁事件，在屏幕锁屏时启动1个像素的 Activity，在用户解锁时将 Activity 销毁掉。注意该 Activity 需设计成用户无感知。通过该方案，可以使进程的优先级在屏幕锁屏时间由4提升为最高优先级1。

#### 方案适用范围

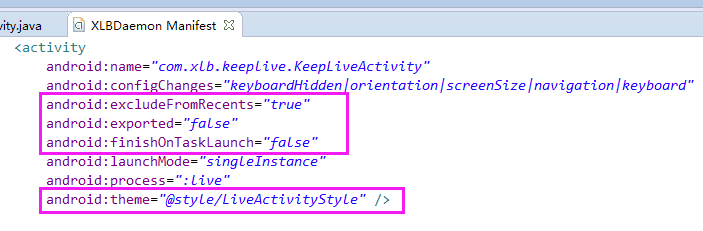
适用场景： 本方案主要解决第三方应用及系统管理工具在检测到锁屏事件后一段时间（一般为5分钟以内）内会杀死后台进程，已达到省电的目的问题。适用版本： 适用于所有的 Android 版本。

#### 方案具体实现

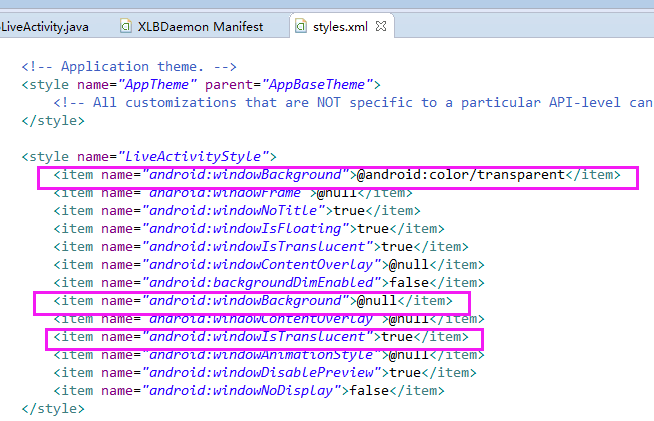
首先定义 Activity，并设置 Activity 的大小为1像素：



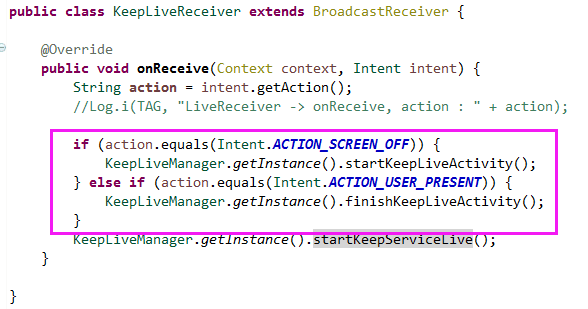
其次，从 AndroidManifest 中通过如下属性，排除 Activity 在 RecentTask 中的显示：



最后，控制 Activity 为透明：



Activity 启动与销毁时机的控制：



### 利用 Notification 提升权限

#### 方案设计思想

Android 中 Service 的优先级为4，通过 setForeground 接口可以将后台 Service 设置为前台 Service，使进程的优先级由4提升为2，从而使进程的优先级仅仅低于用户当前正在交互的进程，与可见进程优先级一致，使进程被杀死的概率大大降低。

#### 方案实现挑战

从 Android2.3 开始调用 setForeground 将后台 Service 设置为前台 Service 时，必须在系统的通知栏发送一条通知，也就是前台 Service 与一条可见的通知时绑定在一起的。对于不需要常驻通知栏的应用来说，该方案虽好，但却是用户感知的，无法直接使用。

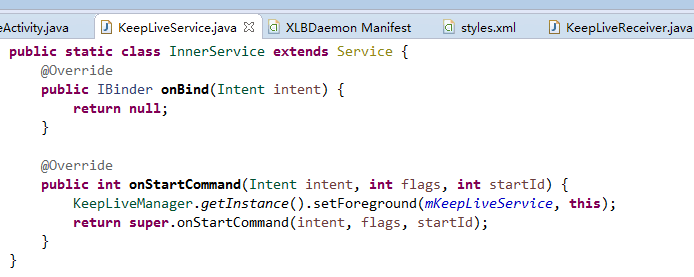
#### 方案挑战应对措施

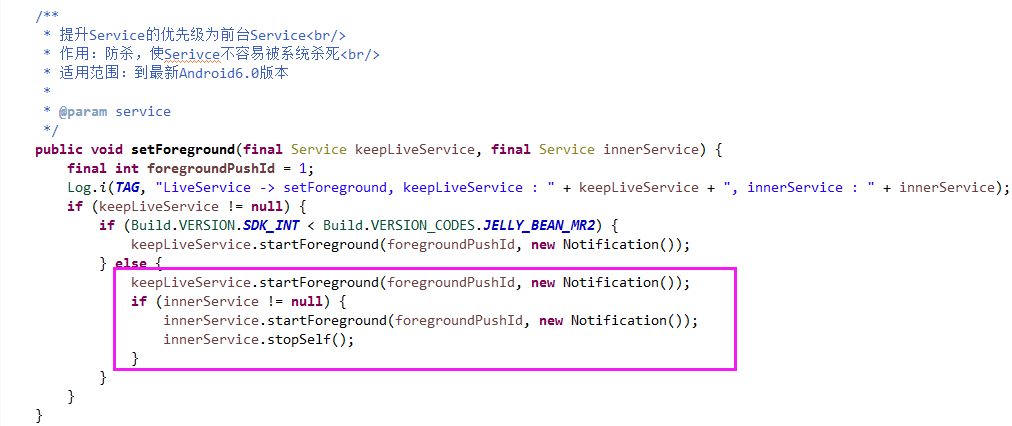
通过实现一个内部 Service，在 LiveService 和其内部 Service 中同时发送具有相同 ID 的 Notification，然后将内部 Service 结束掉。随着内部 Service 的结束，Notification 将会消失，但系统优先级依然保持为2。

#### 方案适用范围

适用于目前已知所有版本。

#### 方案具体实现



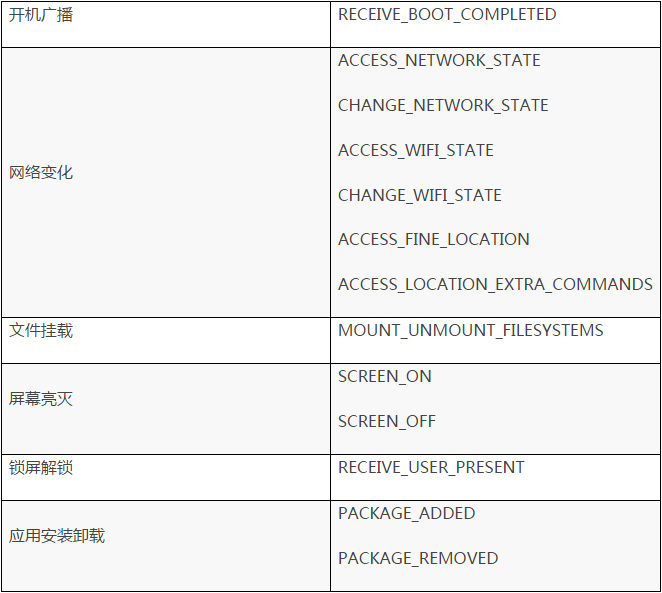


## 进程死后拉活的方案

### 利用系统广播拉活

#### 方案设计思想

在发生特定系统事件时，系统会发出响应的广播，通过在 AndroidManifest 中“静态”注册对应的广播监听器，即可在发生响应事件时拉活。常用的用于拉活的广播事件包括：



#### 方案适用范围

适用于全部 Android 平台。但存在如下几个缺点，因此，该方案主要作为备用手段。

* 广播接收器被管理软件、系统软件通过“自启管理”等功能禁用的场景无法接收到广播，从而无法自启。
* 系统广播事件不可控，只能保证发生事件时拉活进程，但无法保证进程挂掉后立即拉活。

### 利用第三方应用广播拉活

#### 方案设计思想

该方案总的设计思想与接收系统广播类似，不同的是该方案为接收第三方 Top 应用广播。通过反编译第三方 Top 应用，如：手机QQ、微信、支付宝、UC浏览器等，以及友盟、信鸽、个推等 SDK，找出它们外发的广播，在应用中进行监听，这样当这些应用发出广播时，就会将我们的应用拉活。

#### 方案适用范围

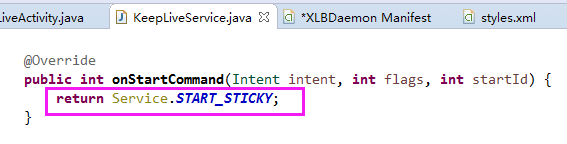
该方案的有效程度除与系统广播一样的因素外，主要受如下因素限制，这些因素都影响了拉活的效果。

* 反编译分析过的第三方应用的多少
* 第三方应用的广播属于应用私有，当前版本中有效的广播，在后续版本随时就可能被移除或被改为不外发。

### 利用系统Service机制拉活

#### 方案设计思想

将 Service 设置为 START\_STICKY，利用系统机制在 Service 挂掉后自动拉活：



#### 方案适用范围

如下两种情况无法拉活：

* Service 第一次被异常杀死后会在5秒内重启，第二次被杀死会在10秒内重启，第三次会在20秒内重启，一旦在短时间内 Service 被杀死达到5次，则系统不再拉起。
* 进程被取得 Root 权限的管理工具或系统工具通过 forestop 停止掉，无法重启。

### 利用Native进程拉活

#### 方案设计思想

主要思想：利用 Linux 中的 fork 机制创建 Native 进程，在 Native 进程中监控主进程的存活，当主进程挂掉后，在 Native 进程中立即对主进程进行拉活。

主要原理：在 Android 中所有进程和系统组件的生命周期受 ActivityManagerService 的统一管理。而且，通过 Linux 的 fork 机制创建的进程为纯 Linux 进程，其生命周期不受 Android 的管理。

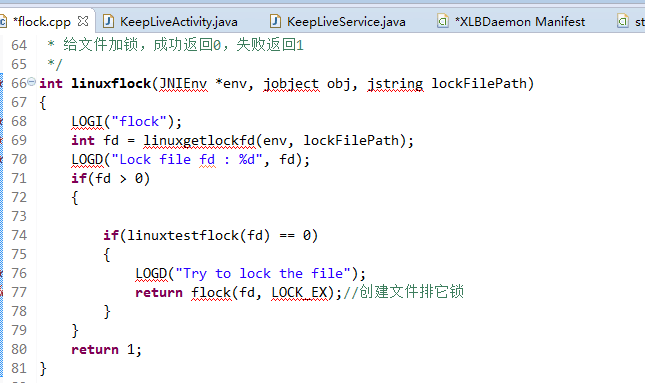
#### 方案实现挑战

##### 挑战一：在 Native 进程中如何感知主进程死亡。

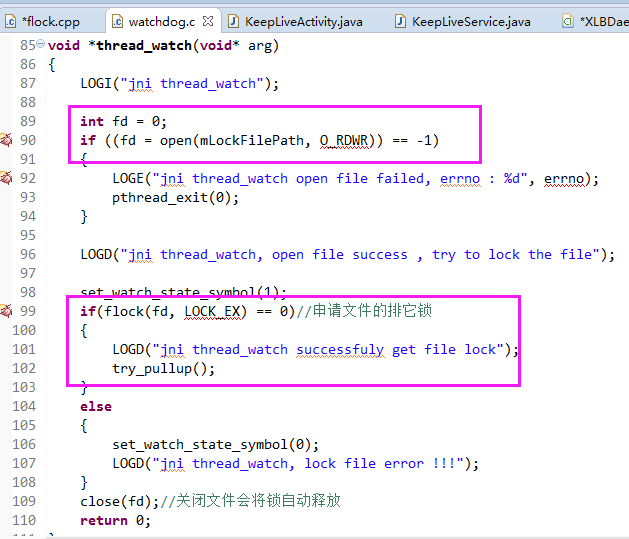
要在 Native 进程中感知主进程是否存活有两种实现方式：

* 在 Native 进程中通过死循环或定时器，轮训判断主进程是否存活，档主进程不存活时进行拉活。该方案的很大缺点是不停的轮询执行判断逻辑，非常耗电。
* 在主进程中创建一个监控文件，并且在主进程中持有文件锁。在拉活进程启动后申请文件锁将会被堵塞，一旦可以成功获取到锁，说明主进程挂掉，即可进行拉活。由于 Android 中的应用都运行于虚拟机之上，Java 层的文件锁与 Linux 层的文件锁是不同的，要实现该功能需要封装 Linux 层的文件锁供上层调用。

封装 Linux 文件锁的代码如下：

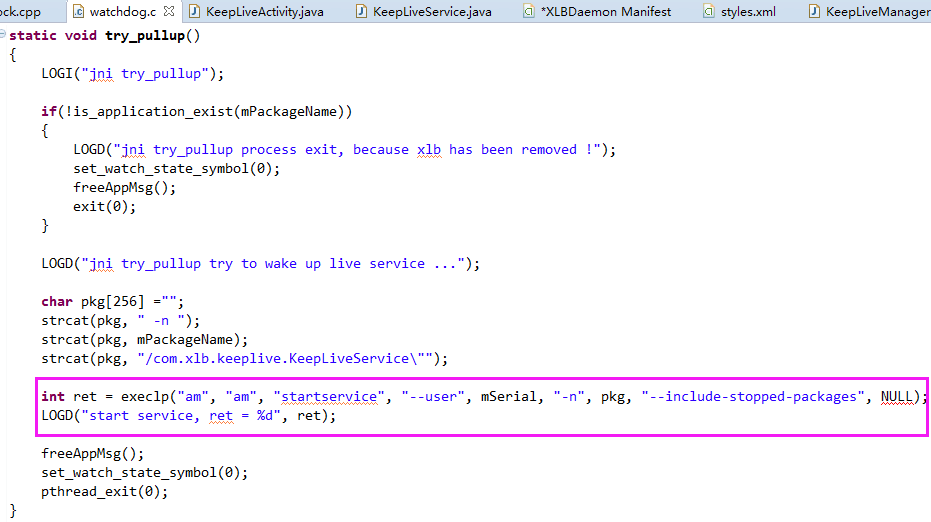


Native 层中堵塞申请文件锁的部分代码：



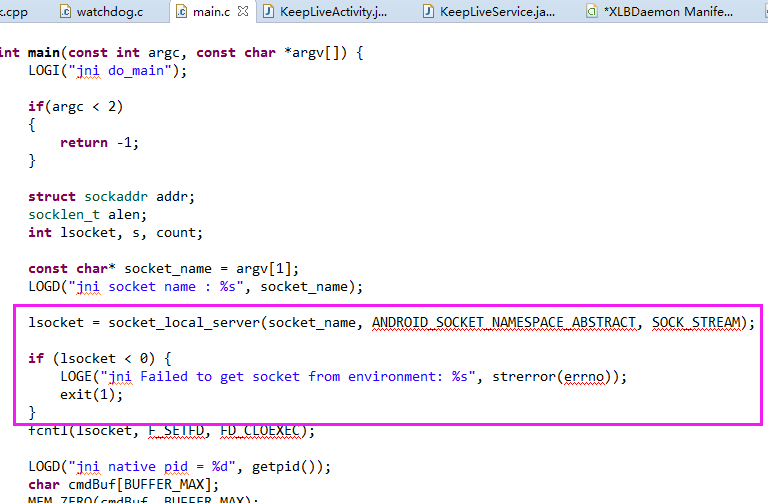
##### 挑战二：在 Native 进程中如何拉活主进程。

通过 Native 进程拉活主进程的部分代码如下，即通过 am 命令进行拉活。通过指定“–include-stopped-packages”参数来拉活主进程处于 forestop 状态的情况。



##### 挑战三：如何保证 Native 进程的唯一

从可扩展性和进程唯一等多方面考虑，将 Native 进程设计层 C/S 结构模式，主进程与 Native 进程通过 Localsocket 进行通信。在Native进程中利用 Localsocket 保证 Native 进程的唯一性，不至于出现创建多个 Native 进程以及 Native 进程变成僵尸进程等问题。



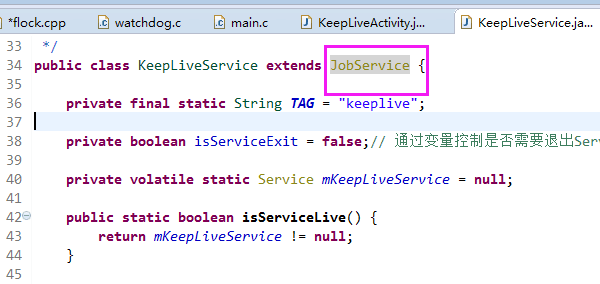
#### 方案适用范围

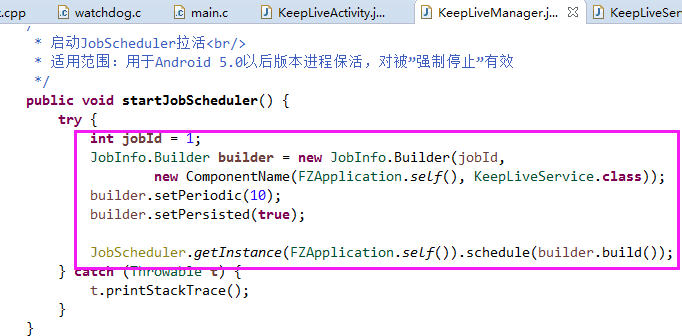
该方案主要适用于 Android5.0 以下版本手机。该方案不受 forcestop 影响，被强制停止的应用依然可以被拉活，在 Android5.0 以下版本拉活效果非常好。对于 Android5.0 以上手机，系统虽然会将native进程内的所有进程都杀死，这里其实就是系统“依次”杀死进程时间与拉活逻辑执行时间赛跑的问题，如果可以跑的比系统逻辑快，依然可以有效拉起。记得网上有人做过实验，该结论是成立的，在某些 Android 5.0 以上机型有效。

### 利用 JobScheduler 机制拉活

#### 方案设计思想

Android5.0 以后系统对 Native 进程等加强了管理，Native 拉活方式失效。系统在 Android5.0 以上版本提供了 JobScheduler 接口，系统会定时调用该进程以使应用进行一些逻辑操作。在本项目中，我对 JobScheduler 进行了进一步封装，兼容 Android5.0 以下版本。封装后 JobScheduler 接口的使用如下：





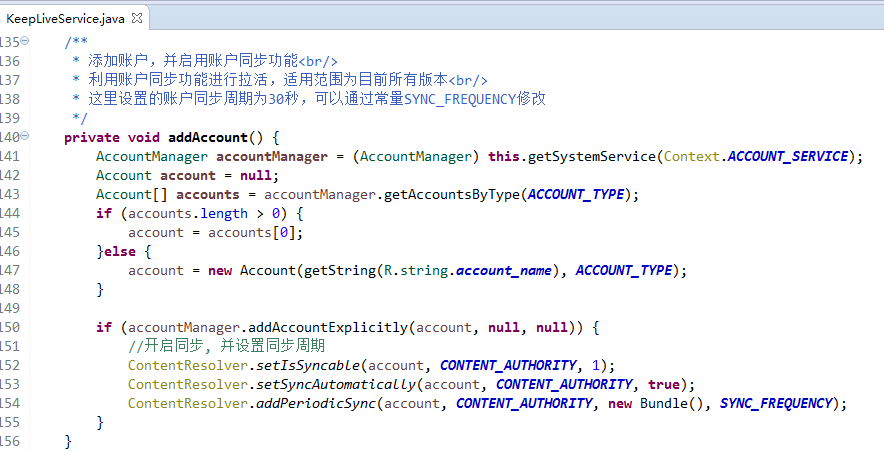
#### 方案适用范围

该方案主要适用于 Android5.0 以上版本手机。该方案在 Android5.0 以上版本中不受 forcestop 影响，被强制停止的应用依然可以被拉活，在 Android5.0 以上版本拉活效果非常好。仅在小米手机可能会出现有时无法拉活的问题。

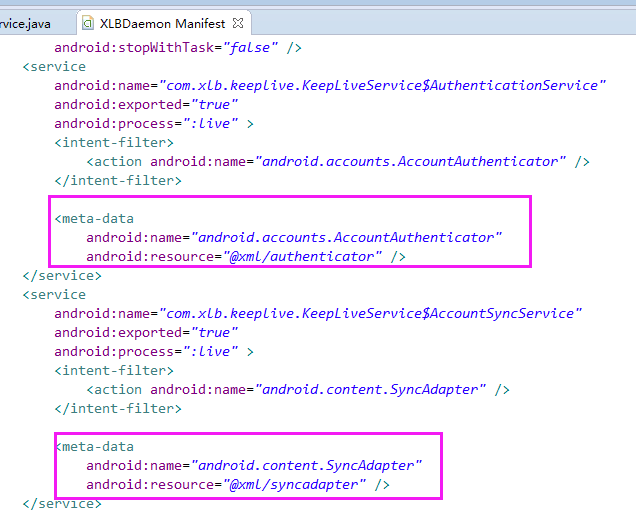
### 利用账号同步机制拉活

#### 方案设计思想

Android 系统的账号同步机制会定期同步账号进行，该方案目的在于利用同步机制进行进程的拉活。添加账号和设置同步周期的代码如下：



该方案需要在 AndroidManifest 中定义账号授权与同步服务。



#### 方案适用范围

该方案适用于所有的 Android 版本，包括被 forestop 掉的进程也可以进行拉活。最新 Android 版本（Android N）中系统好像对账户同步这里做了变动，该方法不再有效。

### 其他有效拉活方案

经研究发现还有其他一些系统拉活措施可以使用，但在使用时需要用户授权，用户感知比较强烈。

这些方案包括：

* 利用系统通知管理权限进行拉活
* 利用辅助功能拉活，将应用加入厂商或管理软件白名单。

这些方案需要结合具体产品特性来搞。上面所有解释这些方案都是考虑的无 Root 的情况。其他还有一些技术之外的措施，比如说应用内 Push 通道的选择：

* 国外版应用：接入 Google 的 GCM。
* 国内版应用：根据终端不同，在小米手机（包括 MIUI）接入小米推送、华为手机接入华为推送；其他手机可以考虑接入腾讯信鸽或极光推送与小米推送做 A/B Test。

# 硬件加速

## 概述

在手机客户端尤其是Android应用的开发过程中，我们经常会接触到“硬件加速”这个词。由于操作系统对底层软硬件封装非常完善，上层软件开发者往往对硬件加速的底层原理了解很少，也不清楚了解底层原理的意义，因此常会有一些误解，如硬件加速是不是通过特殊算法实现页面渲染加速，或是通过硬件提高CPU/GPU运算速率实现渲染加速。

本文尝试从底层硬件原理，一直到上层代码实现，对硬件加速技术进行简单介绍，其中上层实现基于Android 6.0。

## 硬件加速对App开发的意义

对于App开发者，简单了解硬件加速原理及上层API实现，开发时就可以充分利用硬件加速提高页面的性能。以Android举例，实现一个圆角矩形按钮通常有两种方案：使用PNG图片；使用代码（XML/Java）实现。简单对比两种方案如下：



## 页面渲染背景知识

* 页面渲染时，被绘制的元素最终要转换成矩阵像素点（即多维数组形式，类似安卓中的Bitmap），才能被显示器显示。
* 页面由各种基本元素组成，例如圆形、圆角矩形、线段、文字、矢量图（常用贝塞尔曲线组成）、Bitmap等。
* 元素绘制时尤其是动画绘制过程中，经常涉及插值、缩放、旋转、透明度变化、动画过渡、毛玻璃模糊，甚至包括3D变换、物理运动（例如游戏中常见的抛物线运动）、多媒体文件解码（主要在桌面机中有应用，移动设备一般不用GPU做解码）等运算。
* 绘制过程经常需要进行逻辑较简单、但数据量庞大的浮点运算。

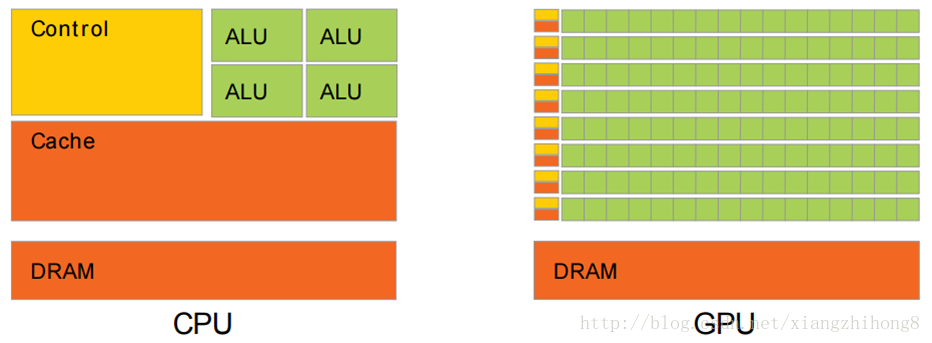
## CPU与GPU结构介绍

CPU（Central Processing Unit，中央处理器）是计算机设备核心器件，用于执行程序代码，软件开发者对此都很熟悉；

GPU（Graphics Processing Unit，图形处理器）主要用于处理图形运算，通常所说“显卡”的核心部件就是GPU。

下面是CPU和GPU的结构对比图。其中：

* 黄色的Control为控制器，用于协调控制整个CPU的运行，包括取出指令、控制其他模块的运行等；
* 绿色的ALU（Arithmetic Logic Unit）是算术逻辑单元，用于进行数学、逻辑运算；
* 橙色的Cache和DRAM分别为缓存和RAM，用于存储信息。



说明：从结构图可以看出，CPU的控制器较为复杂，而ALU数量较少。因此CPU擅长各种复杂的逻辑运算，但不擅长数学尤其是浮点运算。

* 以8086为例，一百多条汇编指令大部分都是逻辑指令，数学计算相关的主要是16位加减乘除和移位运算。一次整型和逻辑运算一般需要1~3个机器周期，而浮点运算要转换成整数计算，一次运算可能消耗上百个机器周期。
* 更简单的CPU甚至只有加法指令，减法用补码加法实现，乘法用累加实现，除法用减法循环实现。
* 现代CPU一般都带有硬件浮点运算器（FPU），但主要适用于数据量不大的情况。
* CPU是串行结构。以计算100个数字为例，对于CPU的一个核，每次只能计算两个数的和，结果逐步累加。
* 和CPU不同的是，GPU就是为实现大量数学运算设计的。从结构图中可以看到，GPU的控制器比较简单，但包含了大量ALU。GPU中的ALU使用了并行设计，且具有较多浮点运算单元。
* 硬件加速的主要原理，就是通过底层软件代码，将CPU不擅长的图形计算转换成GPU专用指令，由GPU完成。

扩展：很多计算机中的GPU有自己独立的显存；没有独立显存则使用共享内存的形式，从内存中划分一块区域作为显存。显存可以保存GPU指令等信息。

## Android中的硬件加速

在Android中，大多数应用的界面都是利用常规的View来构建的（除了游戏、视频、图像等应用可能直接使用OpenGL ES）。下面根据Android 6.0原生系统的Java层代码，对View的软件和硬件加速渲染做一些分析和对比。

### DisplayList

DisplayList是一个基本绘制元素，包含元素原始属性（位置、尺寸、角度、透明度等），对应Canvas的drawXxx()方法。信息传递流程：Canvas(Java API) —> OpenGL(C/C++ Lib) —> 驱动程序 —> GPU。

在Android 4.1及以上版本，DisplayList支持属性，如果View的一些属性发生变化（比如Scale、Alpha、Translate），只需把属性更新给GPU，不需要生成新的DisplayList。

### RenderNode

一个RenderNode包含若干个DisplayList，通常一个RenderNode对应一个View，包含View自身及其子View的所有DisplayList。



### Android绘制流程（Android 6.0）

下面是安卓View完整的绘制流程图，主要通过阅读源码和调试得出，虚线箭头表示递归调用。下面是绘制的简单流程：

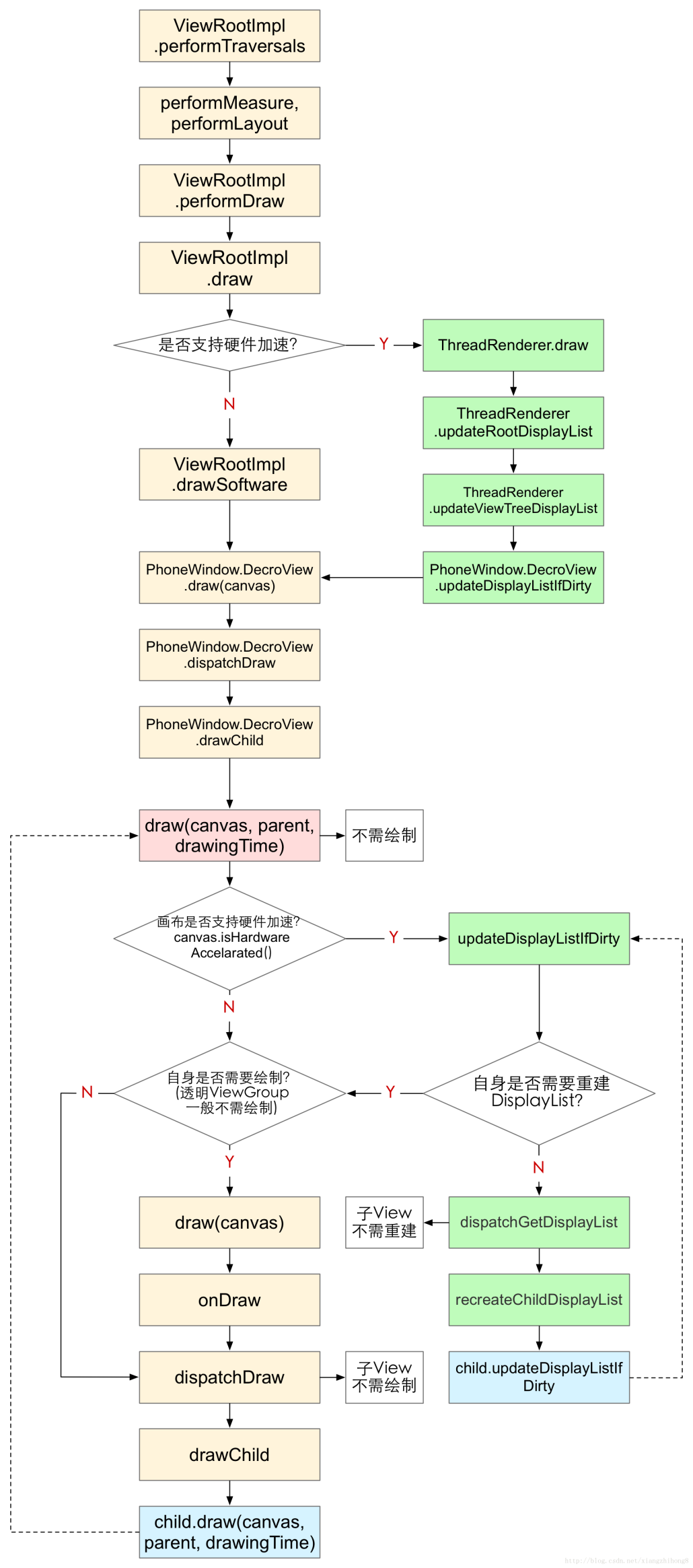
从ViewRootImpl.performTraversals到PhoneWindow.DecroView.drawChild是每次遍历View树的固定流程，首先根据标志位判断是否需要重新布局并执行布局；然后进行Canvas的创建等操作开始绘制。

* 如果硬件加速不支持或者被关闭，则使用软件绘制，生成的Canvas即Canvas.class的对象；
* 如果支持硬件加速，则生成的是DisplayListCanvas.class的对象；

两者的isHardwareAccelerated()方法返回的值分别为false、true，View根据这个值判断是否使用硬件加速。

1. View中的draw(canvas,parent,drawingTime) - draw(canvas) - onDraw -dispachDraw -drawChild这条递归路径（下文简称Draw路径），调用了Canvas.drawXxx()方法，在软件渲染时用于实际绘制；在硬件加速时，用于构建DisplayList。
2. View中的updateDisplayListIfDirty - dispatchGetDisplayList -
3. recreateChildDisplayList这条递归路径（下文简称DisplayList路径），仅在硬件加速时会经过，用于在遍历View树绘制的过程中更新DisplayList属性，并快速跳过不需要重建DisplayList的View。
4. Android 6.0中，和DisplayList相关的API目前仍被标记为“@hide”不可访问，表示还不成熟，后续版本可能开放。硬件加速情况下，draw流程执行结束后DisplayList构建完成，然后通过ThreadedRenderer.nSyncAndDrawFrame()利用GPU绘制DisplayList到屏幕上。

完整的绘制流程如下：



## 纯软件绘制 VS 硬件加速



说明：

1. 场景1中，无论是否加速，遍历View树并都会走Draw路径。硬件加速后Draw路径不做实际绘制工作，只是构建DisplayList，复杂的绘制计算任务被GPU分担，已经有了较大的加速效果。
2. 场景2中，绘制时，TextView所在区域即为脏区。由于TextView有透明区域，遍历View树的过程中，和脏区重叠的多数View都要重绘，包括与之重叠的兄弟节点和他们的父节点（详见后面的介绍），不需要绘制的View在draw(canvas,parent,drawingTime)方法中判断直接返回。

* 硬件TextView设置前后尺寸位置不变，不会触发重新Layout。
* 软件加速后，也需要遍历View树，但只有TextView及其每一层父节点需要重建DisplayList，走的是Draw路径，其他View直接走了DisplayList路径，剩下的工作都交给GPU处理。页面越复杂，两者性能差距越明显。

1. 场景3中，软件绘制每一帧都要做大量绘制工作，很容易导致动画卡顿。硬件加速后，动画过程直接走DisplayList路径更新DisplayList的属性，动画流畅度能得到极大提高。
2. 场景4中，两者的性能差距更明显。简单修改透明度，软件绘制仍然要做很多工作；硬件加速后一般直接更新RenderNode的属性，不需要触发invalidate，也不会遍历View树（除了少数View可能要对Alpha做特殊响应并在onSetAlpha()返回true）

## 软件绘制刷新逻辑简介

实际阅读源码并实验，得出通常情况下的软件绘制刷新逻辑：

1，默认情况下，View的clipChildren属性为true，即每个View绘制区域不能超出其父View的范围。如果设置一个页面根布局的clipChildren属性为false，则子View可以超出父View的绘制区域。

2，当一个View触发invalidate，且没有播放动画、没有触发layout的情况下：

* 对于全不透明的View，其自身会设置标志位PFLAG\_DIRTY，其父View会设置标志位PFLAG\_DIRTY\_OPAQUE。在draw(canvas)方法中，只有这个View自身重绘。
* 对于可能有透明区域的View，其自身和父View都会设置标志位PFLAG\_DIRTY。
* clipChildren为true时，脏区会被转换成ViewRoot中的Rect，刷新时层层向下判断，当View与脏区有重叠则重绘。如果一个View超出父View范围且与脏区重叠，但其父View不与脏区重叠，这个子View不会重绘。
* clipChildren为false时，ViewGroup.invalidateChildInParent()中会把脏区扩大到自身整个区域，于是与这个区域重叠的所有View都会重绘。

## 总结

对于硬件加速我们总结一下：

* CPU更擅长复杂逻辑控制，而GPU得益于大量ALU和并行结构设计，更擅长数学运算。
* 页面由各种基础元素（DisplayList）构成，渲染时需要进行大量浮点运算。
* 硬件加速条件下，CPU用于控制复杂绘制逻辑、构建或更新DisplayList；GPU用于完成图形计算、渲染DisplayList。
* 硬件加速条件下，刷新界面尤其是播放动画时，CPU只重建或更新必要的DisplayList，进一步提高渲染效率。
* 实现同样效果，应尽量使用更简单的DisplayList，从而达到更好的性能（Shape代替Bitmap等）。

# NullPointerException空指针

## 定义

定义：NullPointerException是java.lang.NullPointerException的简称，是Java语言中的一个异常类，位于java.lang包中，父类是java.lang.RuntimeException，该异常在源程序中可以不进行捕获和处理。

发生频率： ★★★★★

发生原因：当应用程序试图在需要对象的地方使用时，抛出该异常。这种情况包括：

* 调用 null 对象的实例方法。
* 访问或修改 null 对象的字段。
* 将 null 作为一个数组，获得其长度。
* 将 null 作为一个数组，访问或修改其时间片。
* 将 null 作为 Throwable 值抛出。
* Android 中View没有初始化

应用程序应该抛出该类的实例，指示其他对 null 对象的非法使用。

## 简单例子分析

**具体事例**

现在看一个我们经常在log中出现的事例，下面我们看一下这个Log

Exception in thread "main" java.lang.NullPointerException

at com.example.myproject.Book.getTitle(Book.java:16)

at com.example.myproject.Author.getBookTitles(Author.java:25)

at com.example.myproject.Bootstrap.main(Bootstrap.java:14)

**分析**

我们简单分析一下，这个是我们所遇到的问题。最上面一行表示我们遇到的是java.lang.NullPointerException问题，然后我们需要特别关注at这个单词，它可以告诉我们问题发生的具体位置。然后我们找到相应的位置并解决问题。我们最上面的at提示的是at com.example.myproject.Book.getTitle(Book.java:16)，我们通过debug，首先找到Book.java类，然后看一下第16行

15 public String getTitle() {

16 System.out.println(title.toString());

17 return title;

18 }

通过上面的代码，因为我们报错的方法是getTitle,然后这个方法返回的String值，如果为空的化，只有可能title 为空。所以可以了解到基本上就是title 为null

## 一连串exceptions的例子

**具体事例**

有时候APP catch的是一个Exception，然而实际抛出的是另一个Exception，像这样的：

34 public void getBookIds(int id) {

35 try {

36 book.getId(id); // 这个方法抛出NullPointerException在22行

37 } catch (NullPointerException e) {

38 throw new IllegalStateException("A book has a null property", e)

39 }

40 }

然后这个异常的Log是这样的：

Exception in thread "main" java.lang.IllegalStateException: A book has a null property

at com.example.myproject.Author.getBookIds(Author.java:38)

at com.example.myproject.Bootstrap.main(Bootstrap.java:14)

Caused by: java.lang.NullPointerException

at com.example.myproject.Book.getId(Book.java:22)

at com.example.myproject.Author.getBookIds(Author.java:36)

... 1 more

**分析**

这个异常中的Caused by有什么不同吗，有时候Log中有很多Caused by。我们希望在这些错误中找到最主要的问题，一般主要的问题出现在最后的Caused by。在这个异常中的主要问题是：

Caused by: java.lang.NullPointerException <-- 主要导致的Exception

at com.example.myproject.Book.getId(Book.java:22) <-- 最重要的一行

然后我们根据这个异常Log，我们可以定位到Book.java类中22行的getId方法。确定是这行导致的NullPointerException 。然后判断是否有值为null 。

## 具体示例

### 方法中对象参数为空导致的NullPointerException

public void doSomething(SomeObject obj){

//一些对obj的操作

}

**分析**

正常看上去没有什么问题，但是这个会出现隐藏的bug，平时我们在编码过程中很容易出现这种细节的问题。当我们使用下列代码的时候。

doSomething(null);

当你传递的参数对象SomeObject为null的时候，例如：

SomeObject obj = null;

doSomething(obj);

// 等同于

doSomething(null);

最后实现的就是这样的。当方法中使用SomeObject对象时。会发生NullPointerException异常。所以在我们操作的过程中。我们需要对传递过来的SomeObject对象进行非空判断，然后在对象不为空的时候进行你需要的操作，然后在为空的时候进行提示或者其他操作。所以最后我们的处理操作是：

/\*\*

\* @param obj 参数有可能为空

\*

\*/

public void doSomething(SomeObject obj){

if(obj != null){

//do something

} else {

//do something else

}

}

### 创建对象数组时候抛出空指针

public class ResultList {

public String name;

public Object value;

public ResultList() {}

}

public class Test {

public static void main(String[] args){

ResultList[] boll = new ResultList[5];

boll[0].name = "iiii";

}

}

**分析**

首先看一下main方法中的这一行代码

ResultList[] boll = new ResultList[5];

对象数组已经初始化了，但是数据中的每一个对象并没有初始化。所以我们调用boll[0].name = "iiii";这行代码会报空指针。 我们应该在调用之前进行初始化对象操作。

ResultList[] boll = new ResultList[5];

// 调用之前进行初始化

boll[0] = new ResultList();

boll[0].name = "iiii";

或者另外一种方式初始化

ResultList[] boll = {new ResultList(),new ResultList(),new ResultList(),new ResultList(),new ResultList()}；

// 这种数据的定义并对每一个对象进行初始化

boll[0].name = "iiii";

这种方式就是数据的定义和初始化的两种方式。

### 在Fragment中调用onCreate方法并找控件造成NullPointerException

Fragment的onCreate()方法

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

View something = findViewById(R.id.something);

something.setOnClickListener(new View.OnClickListener() { ... }); // NPE HERE

if (savedInstanceState == null) {

getSupportFragmentManager().beginTransaction()

.add(R.id.container, new PlaceholderFragment()).commit();

}

}

Fragment的布局

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

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<View

android:id="@+id/something"

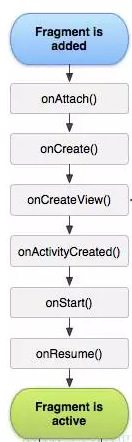
android:layout\_width="100dp"

android:layout\_height="100dp"/>

</RelativeLayout>

**分析**

首先我们看一张Fragment生命周期的图片



可以知道onCreate方法在onCreateView方法之前执行。然后返回null。然后如果调用方法就造成了NullPointerException 可以使用以下的方法进行

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

View rootView = inflater.inflate(R.layout.fragment\_main, container, false);

View something = rootView.findViewById(R.id.something); // not activity findViewById()

something.setOnClickListener(new View.OnClickListener() { ... });

return rootView;

}

我们等到布局已经创建之后进行findViewById就不会造成NullPointerException

### 使用RecyclerView造成的NullPointerException

报错日志信息

java.lang.NullPointerException: Attempt to invoke virtual method 'void android.support.v7.widget.RecyclerView$LayoutManager.onMeasure(android.support.v7.widget.RecyclerView$Recycler, android.support.v7.widget.RecyclerView$State, int, int)' on a null object reference

at android.support.v7.widget.RecyclerView.onMeasure(RecyclerView.java:1764)

at android.view.View.measure(View.java:17430)

**分析**

根据我们上面使用的方法，找到报错行文件。找到有可能发生NullPointerException的位置

void android.support.v7.widget.RecyclerView$LayoutManager.onMeasure

是没有提供LayoutManager造成的，所以我们需要做以下处理

// 还需要找到相应的recyclerView

final LinearLayoutManager layoutManager = new LinearLayoutManager(context);

layoutManager.setOrientation(LinearLayoutManager.VERTICAL);

recyclerView.setLayoutManager(layoutManager);

### findViewById方法引发的NullPointerException

MainActivity

public class MainActivity extends Activity {

ViewPager pager;

MyPagerAdapter adapter;

LinearLayout layout1, layout2, layout3;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

layout1 = (LinearLayout) findViewById(R.id.first\_View);

}

}

activity\_main layout

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

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

android:orientation="vertical"

android:layout\_width="fill\_parent"

android:layout\_height="fill\_parent"

android:background="#a4c639">

<android.support.v4.view.ViewPager

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:id="@+id/main\_pager"/>

</LinearLayout>

activity\_first layout

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

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:id="@+id/first\_View">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@string/hello\_world" />

<Button

android:id="@+id/button1"

style="?android:attr/buttonStyleSmall"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Button" />

</LinearLayout>

**分析**

findViewById在setContentView方法调用完之后调用，并且在setContentView中找需要的控件，如果没有则返回为null。当使用setContentView(R.layout.activity\_main)，如果你调用layout1 = (LinearLayout) findViewById(R.id.first\_View)，会返回null（因为在activity\_main中找不到这个控件的id，它在activity\_first中）。所以在我们使用控件的时候，如果是findViewById造成的NullPointerException,你就需要在setContentView的布局文件中寻找是否有该控件和id。

## 总结

NullPointerException发生的原因有很多种，我们也有相应的解决方法。主要的方式有：

1. 首先查看报NullPointerException的日志，然后找到具体的位置(1.找到at的位置，2.定位你个人的包名，（如at com.example.myproject.Author.getBookIds），然后进行非空判断或者其他操作
2. 如果日志中没有找到具体的位置，然后需要找到具体报错的外部类名和方法（如android.support.v7.widget.RecyclerView$LayoutManager.onMeasure），然后进行非空操作或者其他操作
3. Android Activity和Fragment中出现NullPointerException,如果以上方法没有问题，了解他们的生命周期，然后看看执行顺序有没有问题。
4. 数组，对象，集合等调用方法之前需要初始化，List<String> mList = new ArrayList<>();这种类似的操作必不可少。

# NoClassDefFoundError、ClassNotFoundException

## NoClassDefFoundError错误发生的原因

NoClassDefFoundError错误的发生，是因为Java虚拟机在编译时能找到合适的类，而在运行时不能找到合适的类导致的错误。例如在运行时我们想调用某个类的方法或者访问这个类的静态成员的时候，发现这个类不可用，此时Java虚拟机就会抛出NoClassDefFoundError错误。简单总结就是，NoClassDefFoundError发生在编译时对应的类可用，而运行时在Java的classpath路径中，对应的类不可用导致的错误。

## NoClassDefFoundError和ClassNotFoundException区别

我们经常被java.lang.ClassNotFoundException和java.lang.NoClassDefFoundError这两个错误迷惑不清，尽管他们都与Java classpath有关，但是他们完全不同。

* NoClassDefFoundError发生在JVM在动态运行时，根据你提供的类名，在classpath中找到对应的类进行加载，但当它找不到这个类时，就发生了java.lang.NoClassDefFoundError的错误
* ClassNotFoundException是在编译的时候在classpath中找不到对应的类而发生的错误

## 常见的NoClassDefFoundError

情况一：

编译版本设置的表较高，而最低安装版本较低,。比如compileSdkVersion 25，minSdkVersion 15，scrollview使用方法setOnScrollChangeListener()，而该方法是23以后引用的，没有做版本判断的话，在低于23 的手机上使用，由于没有该方法，所以就报NoClassDefFoundError

情况二：

当你的项目出现Found 2 versions of android-support-v4.jar in the dependency list,but not all the versions are identical (check is based on SHA-1 only at this time).提示的时候，要小心了， 因为很可能因为这个错误而导致解释不通的异常出现。这个错误是因为导入的某个jar包版本跟第三方库的jar版本不一致造成的。

最常见的是android-support-v4.jar这个jar文件，一般我们自己的项目会用到，而第三方库往往也会用到，如果这两个jar的版本不一致，就会出现问题（往往编译能通过）。以下是我运行之后遇到的异常：java.lang.NoClassDefFoundError: com.jcodecraeer.devandroid.MainActivity，但是我很确信这个类是绝对ok的。如果报出这个异常你按照常理去找错误会浪费你很多时间，而且解决不了问题。其实我们只要将两个android-support-v4.jar都用一个就行了，删除自己项目中的，把第三方库中的jar copy进来，或者反过来也行。

# Lambda表达式

Lambda表达式在 Android 中最低兼容到 Android2.3 系统，本质上是一种匿名方法，它既没有方法名，也没有访问修饰符和返回值类型，使用它编写的代码将更加简洁易读。

在Java 8里面，所有的Lambda的类型都是一个接口，而Lambda表达式本身，也就是”那段代码“，需要是这个接口的实现。这是我认为理解Lambda的一个关键所在，简而言之就是，Lambda表达式本身就是一个接口的实现

## 基本写法

如果想要在 Android 项目中使用 Lambda表达式 或者 Java8 的其他新特性，首先我们需要安装Java8版本的JDK，然后在 app/build.gradle 中添加以下配置：

android {

...

defaultConfig {

jackOptions.enabled = true

}

compileOptions {

sourceCompatibility JavaVersion.VERSION\_1\_8

targetCompatibility JavaVersion.VERSION\_1\_8

}

}

比如开启子线程的写法：

// 传统方式

new Thread(new Runnable() {

@Override

public void run() {

doSomeThing();

}

}).start();

// 使用Lambda表达式

new Thread(() -> {

doSomeThing();

}).start();

不管是从代码行数上还是缩进结构上看，Lambda表达式的写法更加精简。为什么可以这么写呢？我们看一下 Runnable 接口的源码：

public interface Runnable {

void run();

}

凡是这种只有一个待实现方法的接口，都可以使用 Lambda表达式的写法。

## 自定义接口然后使用Lambda表达式

新建一个 MyListener 接口，接口中只有一个待实现方法，和前面唯一不同的是带有参数和返回值:

public interface MyListener {

String run(String str1, String str2);

}

那么使用 Lambda表达式 创建 MyListener 接口的匿名实现方法写法如下：

MyListener listener = (String str1, String str2) -> {

String result = "result:" + str1 + str2;

return result;

};

另外Java还可以通过上下文自动推断出 Lambda表达式 中的参数类型，所以还可以进一步简化：

MyListener listener = (str1, str2) -> {

String result = "result:" + str1 + str2;

return result;

};

如果只有一行语句还可省略掉括号（有返回值的连return也能省掉）

MyListener listener = (str1, str2) -> "result:" + str1 + str2;

另外，当接口只有一个参数时，我们还可以进一步简化，将参数外面的括号去掉：

MyListener listener = str1 -> "result:" + str1;

## Android中使用Lambda

button.setOnClickListener(v -> {

doSomeThing();

});

new Thread(() -> {

doSomeThing();

}).start();

# Lambda表达式究竟是不是语法糖

## 什么是语法糖

代码的写法更加简便，但其实底层的原理并没有改变，例如：

1. 方法当中的可变参数，底层仍然是一个数组
2. 增强for循环用于java.lang.Iterable实现类型时，底层仍然是迭代器
3. 自动装箱、自动拆箱

## lambda表达式和匿名内部类区别

所需的类型不一样

如果是匿名内部类，那么可以用接口，也可以用抽象类，甚至可以是普通的实体类

如果是lambda表达式，就只能用接口

使用的限制不一样

如果接口当中有且只有一个抽象方法时，那么可以使用lambda表达式，也可以使用匿名内部类

但是如果接口当中抽象方法不唯一时，那么就只能使用匿名内部类，不能使用lambda表达式

实现的原理不一样

匿名内部类：其实本身就是一个类，编译之后，直接产生一个单独的.class字节码文件

lambda表达式：编译之后，没有单独的.calss字节码文件，对应的字节码文件会在运行的时候生成

## 分析

分别使用lambda、匿名内部类，内部类，并编译

public class Lambda {

private static void hello(String name, Consumer<String> printer) {

printer.accept(name);

}

public static void main(String[] args) {

hello("lambda", (name) -> System.out.println("Hello " + name));

hello("匿名类", new Consumer<String> () {

@Override

public void accept(String name) {

System.out.println("Hello " + name);

}

});

hello("内部类", new SupplierImpl());

}

static class SupplierImpl implements Consumer<String> {

@Override

public void accept(String name) {

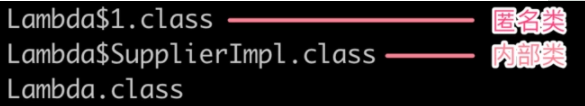
System.out.println("Hello " + name);

}

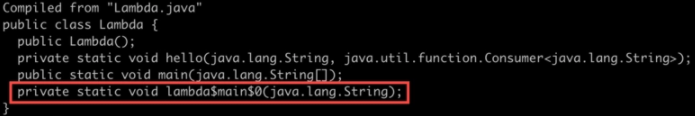
}

}

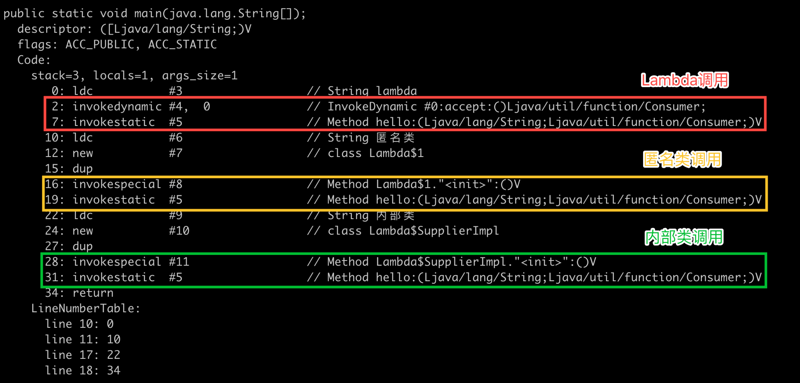
编译后会产生三个文件



简单查看Lambda.Java文件的字节码javap -p Lambda



Java编译器自动帮我们生成了方法lambda$main$0，我们有理由相信，Lambda表达式内的逻辑就封装在此函数内，生成的方法lambda$main$0又是如何被调用的呢？



Lambda的调用使用了invokedynamic指令

分析...分析...分析...<https://segmentfault.com/a/1190000019577184>

## 总结

* lambda表达式会被编译为invokedynamic指令
* 每一个lambda表达式的实现逻辑均会被封装为一个静态私有方法
* 只要存在lambda表达式调用，便会生成一个内部类
* 内部类中每一个方法（启动方法 BoostrapMethod）对应一个lambda表达式所生成的静态私有方法，内部类中的方法用以生成对应的调用点绑定到相应的invokedynamic指令上，这也解释了为什么lambda中的this指针指向的是周围的类 (定义该Lambda表达式时所处的类) (Lambda表达式里的"陷阱")
* 所以，lambda表达式确实是语法糖，但并不是匿名类/内部类的语法糖

# 编译型、解释型、动态、静态、动态类型、静态类型、强类型、弱类型，语言概念与区别

## 编译型语言和解释型语言

### 编译型语言

需通过编译器（compiler）将源代码编译成机器码，之后才能执行的语言。一般需经过编译（compile）、链接（linker）这两个步骤。编译是把源代码编译成机器码，链接是把各个模块的机器码和依赖库串连起来生成可执行文件。

优点：编译器一般会有预编译的过程对代码进行优化。因为编译只做一次，运行时不需要编译，所以编译型语言的程序执行效率高。可以脱离语言环境独立运行。

缺点：编译之后如果需要修改就需要整个模块重新编译。编译的时候根据对应的运行环境生成机器码，不同的操作系统之间移植就会有问题，需要根据运行的操作系统环境编译不同的可执行文件。

代表语言：C、C++、Pascal、Object-C以及最近很火的苹果新语言swift

### 解释型语言

解释性语言的程序不需要编译，相比编译型语言省了道工序，解释性语言在运行程序的时候才逐行翻译。

优点：有良好的平台兼容性，在任何环境中都可以运行，前提是安装了解释器（虚拟机）。灵活，修改代码的时候直接修改就可以，可以快速部署，不用停机维护。

缺点：每次运行的时候都要解释一遍，性能上不如编译型语言。

代表语言：JavaScript、Python、Erlang、PHP、Perl、Ruby

### 混合型语言

既然编译型和解释型各有缺点就会有人想到把两种类型整合起来，取其精华去其糟粕。就出现了半编译型语言。比如C#,C#在编译的时候不是直接编译成机器码而是中间码，.NET平台提供了中间语言运行库运行中间码，中间语言运行库类似于Java虚拟机。.net在编译成IL代码后，保存在dll中，首次运行时由JIT在编译成机器码缓存在内存中，下次直接执行（博友回复指出）。我个人认为抛开一切的偏见C#是这个星球上最好的编程语言。可惜微软的政策限制了C#的推广。

Java先生成字节码再在Java虚拟机中解释执行。严格来说混合型语言属于解释型语言。C#更接近编译型语言。

## 动态语言和静态语言

### 动态语言

是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。通俗点说就是在运行时代码可以根据某些条件改变自身结构。

主要动态语言：Object-C、C#、JavaScript、PHP、Python、Erlang。

### 静态语言

与动态语言相对应的，运行时结构不可变的语言就是静态语言。如Java、C、C++。

### 注意：

很多人认为解释型语言都是动态语言，这个观点是错的！Java是解释型语言但是不是动态语言，Java不能在运行的时候改变自己结构。反之成立吗？动态语言都是解释型语言。也是错的！Object-C是编译型语言，但是他是动态语言。得益于特有的run time机制（准确说run time不是语法特性是运行时环境，这里不展开）OC代码是可以在运行的时候插入、替换方法的。C#也是动态语言，通过C#的反射机制可以动态的插入一段代码执行。所以我说C#是这个星球最好的编程语言。

## 动态类型语言和静态类型语言

### 动态类型语言

很多网上资料把动态类型语言和动态语言混为一谈，简直是误人子弟。动态类型语言和动态语言是完全不同的两个概念。动态类型语言是指在运行期间才去做数据类型检查的语言，说的是数据类型，动态语言说的是运行是改变结构，说的是代码结构。

动态类型语言的数据类型不是在编译阶段决定的，而是把类型绑定延后到了运行阶段。

主要语言：Python、Ruby、Erlang、JavaScript、swift、PHP、Perl。

### 静态类型语言

静态语言的数据类型是在编译其间确定的或者说运行之前确定的，编写代码的时候要明确确定变量的数据类型。

主要语言：C、C++、C#、Java、Object-C。

### 注意：

相当一部分程序员，也包括曾经的我，认为解释型语言都是动态类型语言，编译型语言都是静态类型语言。这个也是错的。swift是编译型语言但是它也是动态类型语言。C#和Java是解释型语言也是静态类型语言。

## 强类型语言和弱类型语言

### 强类型语言：

强类型语言，一旦一个变量被指定了某个数据类型，如果不经过强制类型转换，那么它就永远是这个数据类型。你不能把一个整形变量当成一个字符串来处理。

主要语言：Java、C#、Python、Object-C、Ruby

### 弱类型语言：

数据类型可以被忽略，一个变量可以赋不同数据类型的值。一旦给一个整型变量a赋一个字符串值，那么a就变成字符类型。

主要语言：JavaScript、PHP、C、C++（C和C++有争议，但是确实可以给一个字符变量赋整形值，可能初衷是强类型，形态上接近弱类型）

### 注意：

一个语言是不是强类型语言和是不是动态类型语言也没有必然联系。Python是动态类型语言，是强类型语言。JavaScript是动态类型语言，是弱类型语言。Java是静态类型语言，是强类型语言。

### Java语言的总结

* Java是混合型语言并且属于解释型语言，因为严格来说它时先生成字节码再在Java虚拟机中解释执行
* Java是静态类型语言
* Java是强类型语言

# 闭包

## 什么是闭包

什么是闭包? 对于我们这些习惯了命令式编程, 尤其是java这种完全面向对象的语言的人, 闭包可能是一个很陌生的概念, 闭包时常在函数式语言中被提及, 那闭包到底是一个什么概念? 下面对闭包的解释来自维基百科.

在计算机科学中，闭包（Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。

闭包更多强调的是引用环境, 函数在定义时因为使用到了自由变量, 函数的调用和定义时的引用环境不同, 所以它必须要清楚的了解到自己定义时的引用环境, 在调用的时候需要切换到它定义的引用环境中执行, 这样就形成了闭包.

更通俗的讲, 因为函数引用了自由变量, 当它执行的环境和定义的环境不同时, 在执行它的地方就不仅仅是拿到这个函数这么简单了, 因为它必须要清楚它引用的自由变量, 所以最好的方式是把此时的引用环境和函数本身一并返回, 这样就是闭包.所以, 闭包 = 函数 + 引用环境。分解一下就是闭包形成的必要条件：

1. 函数引用自由变量
2. 函数的执行环境和自由变量的声明环境不同

只有符合了以上两点, 函数的执行点才仅仅需要函数本身, 而是函数+引用环境.两个重点是自由变量, 引用环境, 但就这几个字可能对于一些对闭包一知半解的人来说确是很难理解, 因为在大部分人印象里闭包应该是这样的.

// kotlin

list.map { println it}

这样的一个实例中没有任何地方提及到自由变量. 其实这压根和闭包没有任何关系, 这仅仅是一种语法-lambda表达式, 很多人将lambda表达式想当然的认为是闭包. 再来看看下面语法.

// groovy

android {

compileSdkVersion 24

}

对于搞android的人来说这行代码并不陌生, 很多人把这个也理解成闭包, 和上面kotlin的代码一样, 其实这和闭包也没有任何关系.

那什么样的才算闭包? 上面提到了, 引用了自由变量的函数, 这句话里有一个名词-自由变量, 那什么又是自由变量呢? 理解了这个名词后才能继续理解这句话.

在某个作用域内使用了其他作用域声明的变量, 那该变量就是自由变量

上面的概念是是可能有点难理解, 来看看代码

// javascript

var a = 10;

function add5() {

return a + 5;

}

上面的变量a即为自由变量, a变量不是在函数add5的作用域中声明的, 却在函数add5中被使用.好了, 知道了什么是自由变量, 那下面我们来看看什么是闭包.

// javascript

function add(a) {

return function(b) {

return a + b;

};

}

var add10 = add(10);

var result = add10(5);

上面的例子是最典型的闭包, add函数返回了一个函数, 这个匿名函数引用了add函数的一个a变量, 所以a变量是一个自由变量. 而且, 这个匿名函数的执行点不在变量a声明的作用域内, 根据上面的概念, 这样就形成了闭包.

来看看kotlin官方文档上闭包的实例

// kotlin

var sum = 0

ints.filter { it > 0 }.forEach {

sum += it

}

print(sum)

这是不是形成闭包了? 把它普通化来看看, 毕竟去除语法糖才更适合人理解.

// kotlin

var sum = 0

var filter: (Int) -> Bool = { a -> a > 0 }

var each: (Int) -> Int = { a -> sum + a }

ints.filter(filter).forEach(each)

主要看这个each函数, 很明显, 这里有对自由变量a的引用, 而且each函数的定义和执行时的引用环境不同, 所以上面的实例确实形成了闭包. 不过我还是要吐槽一下kotlin的文档, 它将闭包放在了Lambda章节里, 虽然实例代码确实是形成了闭包, 但肯定会去一些人产生误导作用, 毕竟是两个毫无关系的概念.

## 闭包的作用

通过概念和实例代码, 很明显闭包的存在改变了变量的生命周期, 大部分情况下它可以将自由变量的生命周期延迟到闭包函数的执行, 而函数式中最重要的一个思想是尽可能多使用纯函数(纯函数是指对于相同的输入必定有相同的输入的函数), 在纯函数中如果想要保持一个变量, 那闭包肯定是最佳选择. 来看一下实例.

// javascript

function nameBy(lastName) {

return function(firstName) {

retrn firstName + " " + lastName;

}

}

var group = nameBy("Jordan")

var michael = group("Michael")

var susan = group("Susan")

另外一个函数式中重要的概念-柯里化大部分情况下也离不开闭包的支持, 什么是柯里化?

柯里化（英语：Currying），是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

其实上面的大部分例子中已经实现了柯里化, 柯里化的好处就是大大的提高了函数的灵活性.

// javascript

var add = (x, y) => x + y

// user

alert(add(10, 20))

alert(add(10, 30))

alert(add(10, 40))

上面对add函数的调用其实都是10+y的形式, 很多时候我们为了封装, 又对10+y这样的式子进行封装.

// javascript

var add = (x, y) => x + y

var add10 = (y) => 10 + y

// user

alert(add10(20))

alert(add10(30))

alert(add10(40))

这里有一个不好的地方就是add10这个函数的封装只能适用于10+y, 虽然实现了柯里化, 但是对于使用者来说灵活性不够, 其实这里我们可以利用闭包对add函数稍加改造, 既方便使用又不失灵活性

// javascript

var add = (x) => (y) => x + y

// user

var add10 = add(10)

alert(add10(20))

alert(add10(30))

alert(add10(40))

再来看, 上面实例中的函数add的定义其实不太符合我们人类的思想, 这方面, 其实很多函数式编程语言中已经实现了自动柯里化, 来看个实例.

// elm

add x y =

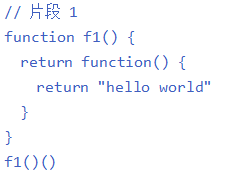
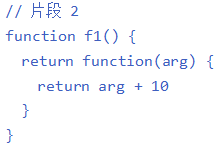
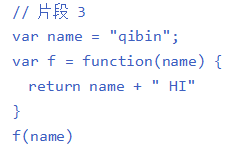
x + y

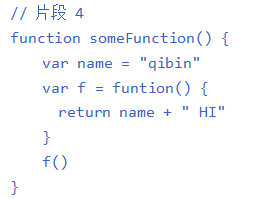
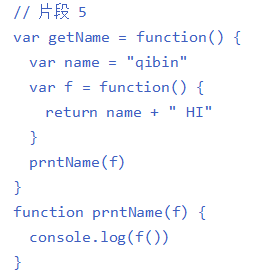
add 10 20 // result is 30

add10 = add 10

add10 20 // result is 30

这些支持自动柯里化的语言可以根据实参的个数推断出返回的类型, 所以可以很轻松的实现柯里化, 而不必在我们定义函数的时候过多考虑好了, 虽然说了这么多关于柯里化的东西, 其实都是对闭包的一些应用, 可以帮助大家去理解闭包. 最后留几个代码片段, 大家判断一下是否实现了闭包.

# 跨进程通信的四种方式

## 概述

这4种方式正好对应于android系统中4种应用程序组件：Activity、Content Provider、Broadcast和Service。

Activity可以跨进程调用其他应用程序的Activity；

Content Provider可以跨进程访问其他应用程序中的数据（以Cursor对象形式返回），当然，也可以对其他应用程序的数据进行增、删、改操 作；

Broadcast可以向android系统中所有应用程序发送广播，而需要跨进程通讯的应用程序可以监听这些广播；

Service和Content Provider类似，也可以访问其他应用程序中的数据，但不同的是，Content Provider返回的是Cursor对象，而Service返回的是Java对象，这种可以跨进程通讯的服务叫AIDL服务。

## 方式一：访问其他应用程序的Activity

Activity既可以在进程内（同一个应用程序）访问，也可以跨进程访问。如果想在同一个应用程序中访问Activity，需要指定Context对象和Activity的Class对象，代码如下：

Intent intent = new Intent(this , Test.class );

startActivity(intent);

Activity的跨进程访问与进程内访问略有不同。虽然它们都需要Intent对象，但跨进程访问并不需要指定Context对象和Activity的 Class对象，而需要指定的是要访问的Activity所对应的Action（一个字符串）。有些Activity还需要指定一个Uri（通过 Intent构造方法的第2个参数指定）。

在android系统中有很多应用程序提供了可以跨进程访问的Activity，例如，下面的代码可以直接调用拨打电话的Activity。

Intent callIntent = new Intent(Intent.ACTION\_CALL, Uri.parse("tel:12345678"));

startActivity(callIntent);

执行上面的代码后，系统会自动拨号。

在调用拨号程序的代码中使用了一个Intent.ACTION\_CALL常量，该常量的定义如下：

Public static final String ACTION\_CALL = "android.intent.action.CALL" ;

这个常量是一个字符串常量，也是我们在这节要介绍的跨进程调用Activity的关键。如果在应用程序中要共享某个Activity，需要为这个 Activity指定一个字符串ID，也就是Action。也可以将这个Action看做这个Activity的key。在其他的应用程序中只要通过这个 Action就可以找到与Action对应的Activity，并通过startActivity方法来启动这个Activity。

下面先来看一下如何将应用程序的Activity共享出来，读者可按如下几步来共享Activity：

1. 在AndroidManifest.xml文件中指定Action。指定Action要使用<action>标签，并在该标签的android:name属性中指定Action

2. 在AndroidManifest.xml文件中指定访问协议。在指定Uri（Intent类的第2个参数）时需要访问协议。访问协议需要使用<data>标签的android:scheme属性来指定。如果该属性的值是“abc”，那么Uri就应该是“abc://Uri的主体 部分”，也就是说，访问协议是Uri的开头部分。

3. 通过getIntent().getData().getHost()方法获得协议后的Uri的主体部分。这个Host只是个称谓，并不一定是主机名。读者可以将其看成是任意的字符串。

4. 从Bundle对象中获得其他应用程序传递过来的数据。

<!-- 重新配置Main -->

<activity android:name=".Main" android:label="@string/app\_name">

<intent-filter>

<action android:name="net.blogjava.mobile.MYACTION" />

<data android:scheme="info" />

<category android:name="android.intent.category.DEFAULT" />

</intent-filter>

</activity>

// 获得其他应用程序传递过来的数据

if (getIntent().getData() != null) {

// 获得Host，也就是info://后面的内容

String host = getIntent().getData().getHost();

Bundle bundle = getIntent().getExtras();

// 其他的应用程序会传递过来一个value值，在该应用程序中需要获得这个值

String value = bundle.getString("value");

// 将Host和Value组合在一下显示在EditText组件中

editText.setText(host + ":" + value);

}

public void onClick(View view) {

// 需要使用Intent类的第2个参数指定Uri

Intent intent = new Intent("net.blogjava.mobile.MYACTION", Uri.parse("info://调用其他应用程序的Activity"));

// 设置value属性值

intent.putExtra("value", "调用成功");

// 调用ActionActivity中的Main

startActivity(intent);

}

## 方式二：Content Provider

Android应用程序可以使用文件或SqlLite数据库来存储数据。Content Provider提供了一种在多个应用程序之间数据共享的方式（跨进程共享数据）。应用程序可以利用Content Provider完成下面的工作1. 查询数据;2. 修改数据;3. 添加数据;4. 删除数据

## 方式三：广播（Broadcast）

广播是一种被动跨进程通讯的方式。当某个程序向系统发送广播时，其他的应用程序只能被动地接收广播数据。这就象电台进行广播一样，听众只能被动地收听，而不能主动与电台进行沟通。

## 方式四：AIDL服务

服务（Service）是android系统中非常重要的组件。Service可以脱离应用程序运行。也就是说，应用程序只起到一个启动Service的作用。一但Service被启动，就算应用程序关闭，Service仍然会在后台运行。

android系统中的Service主要有两个作用：后台运行和跨进程通讯。后台运行就不用说了，当Service启动后，就可以在Service对象中 运行相应的业务代码，而这一切用户并不会察觉。而跨进程通讯是这一节的主题。如果想让应用程序可以跨进程通讯，就要使用我们这节讲的AIDL服 务，AIDL的全称是Android Interface Definition Language，也就是说，AIDL实际上是一种接口定义语言。通过这种语言定义接口后，Eclipse插件（ODT）会自动生成相应的Java代码接口代码

# AIDL

## 介绍

在Android中，每个应用（Application）执行在它自己的进程中，无法直接调用到其他应用的资源，这也符合“沙箱”的理念。所谓沙箱原理，一般来说用在移动电话业务中，简单地说旨在部分地或全部地隔离应用程序。关于沙箱技术我们这里就不多做介绍了，因此，在Android中，当一个应用被执行时，一些操作是被限制的，比如访问内存，访问传感器，等等。这样做可以最大化地保护系统，免得应用程序“为所欲为”。那我们有时需要在应用间交互，怎么办呢？于是，Android需要实现IPC协议。然而，这个协议还是有点复杂，主要因为需要实现数据管理系统（在进程或线程间传递数据）Android为我们实现了自己的IPC，也就是AIDL(Android Interface Definition Language)

## 支持的数据类型共 4 种：

* Java 的基本数据类型
* List 和 Map :元素必须是 AIDL 支持的数据类型；Server 端具体的类里则必须是 ArrayList 或者 HashMap
* 其他 AIDL 生成的接口
* 实现 Parcelable 的实体

## 编写主要为以下三部分：

1. 创建 AIDL

* 创建要操作的实体类，实现 Parcelable 接口，以便序列化/反序列化
* 新建 aidl 文件夹，在其中创建接口 aidl 文件以及实体类的映射 aidl 文件
* Make project ，生成 Binder 的 Java 文件

1. 服务端

* 创建 Service，在其中创建上面生成的 Binder 对象实例，实现接口定义的方法
* 在 onBind() 中返回

1. 客户端

* 实现 ServiceConnection 接口，在其中拿到 AIDL 类
* bindService()
* 调用 AIDL 类中定义好的操作请求

## AIDL 实例

### 创建 AIDL

#### 创建要操作的实体类，实现 Parcelable 接口，以便序列化/反序列化

package net.sxkeji.shixinandroiddemo2.bean;

import android.os.Parcel;

import android.os.Parcelable;

public class Person implements Parcelable {

private String mName;

public Person(String name) {

mName = name;

}

protected Person(Parcel in) {

mName = in.readString();

}

public static final Creator<Person> CREATOR = new Creator<Person>() {

@Override

public Person createFromParcel(Parcel in) {

return new Person(in);

}

@Override

public Person[] newArray(int size) {

return new Person[size];

}

};

@Override

public int describeContents() {

return 0;

}

@Override

public void writeToParcel(Parcel dest, int flags) {

dest.writeString(mName);

}

@Override

public String toString() {

return "Person{" +

"mName='" + mName + '\'' +

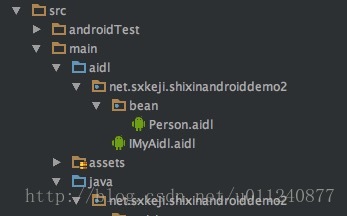
'}';

}

}

#### 新建 aidl 文件夹，在其中创建接口 aidl 文件以及实体类的映射 aidl 文件

在 main 文件夹下新建 aidl 文件夹，使用的包名要和 java 文件夹的包名一致：



先创建实体类的映射aidl文件Person.aidl，要和声明的实体类在一个包名里，在其中声明映射的实体类名称与类型

/\* Person.aidl \*/

package net.sxkeji.shixinandroiddemo2.bean;

parcelable Person;

然后创建接口 aidl 文件，IMyAidl.aidl：

/\* IMyAidl.aidl \*/

package net.sxkeji.shixinandroiddemo2;

import net.sxkeji.shixinandroiddemo2.bean.Person;

interface IMyAidl {

//除了基本数据类型，其他类型的参数都需要标上方向类型：in(输入), out(输出), inout(输入输出)

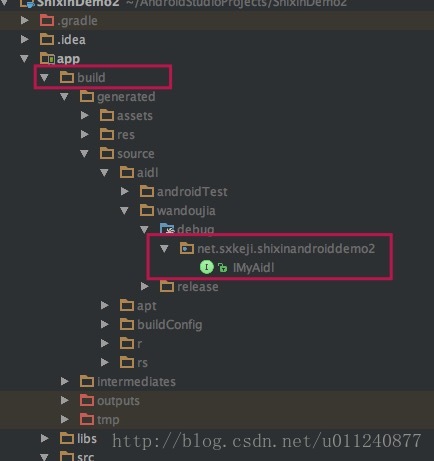
void addPerson(in Person person);

List<Person> getPersonList();

}

#### Make Project ，生成 Binder 的 Java 文件

AIDL 真正的强大之处就在这里，通过简单的定义 aidl 接口，然后编译，就会为我们生成复杂的 Java 文件。点击 Build -> Make Project，然后等待构建完成。然后就会在 build/generated/source/aidl/你的 flavor/ 下生成一个 Java 文件：



### 编写服务端代码

现在我们有了跨进程 Client 和 Server 的通信媒介，接着就可以编写客户端和服务端代码了。

创建 Service，在其中创建上面生成的 Binder 对象实例，实现接口定义的方法；然后在 onBind() 中返回

创建将来要运行在另一个进程的 Service，在其中实现了 AIDL 接口中定义的方法:

public class MyAidlService extends Service {

private final String TAG = this.getClass().getSimpleName();

private ArrayList<Person> mPersons;

/\*\*

\* 创建生成的本地 Binder 对象，实现 AIDL 制定的方法

\*/

private IBinder mIBinder = new IMyAidl.Stub() {

@Override

public void addPerson(Person person) throws RemoteException {

mPersons.add(person);

}

@Override

public List<Person> getPersonList() throws RemoteException {

return mPersons;

}

};

/\*\*

\* 客户端与服务端绑定时的回调，返回 mIBinder 后客户端就可以通过它远程调用服务端的方法，即实现了通讯

\*/

@Nullable

@Override

public IBinder onBind(Intent intent) {

mPersons = new ArrayList<>();

LogUtils.d(TAG, "MyAidlService onBind");

return mIBinder;

}

}

别忘记在 Manifest 文件中声明：

<service

android:name="net.sxkeji.shixinandroiddemo2.service.MyAidlService"

android:enabled="true"

android:exported="true"

android:process=":aidl"/>

### 编写客户端代码

服务端实现了接口，在 onBind() 中返回这个 Binder，客户端拿到就可以操作数据了。这里我们以一个 Activity 为客户端。

#### 实现 ServiceConnection 接口，在其中拿到 AIDL 类

private IMyAidl mAidl;

private ServiceConnection mConnection = new ServiceConnection() {

@Override

public void onServiceConnected(ComponentName name, IBinder service) {

//连接后拿到 Binder，转换成 AIDL，在不同进程会返回个代理

mAidl = IMyAidl.Stub.asInterface(service);

}

@Override

public void onServiceDisconnected(ComponentName name) {

mAidl = null;

}

};

在 Activity 中创建一个服务连接，在其中调用 IMyAidl.Stub.asInterface() 方法将 Binder 转为 AIDL 类。

#### 接着绑定服务

要执行 IPC，必须使用 bindService() 将应用绑定到服务上。5.0之后只能用ComponentName的方法绑定服务

Intent intent = new Intent();

intent.setComponent(new ComponentName(

"net.sxkeji.shixinandroiddemo2.service", "net.sxkeji.shixinandroiddemo2.service.MyAidlService"

));

bindService(intent, serviceConnection, Context.BIND\_AUTO\_CREATE);//绑定远程服务

#### 拿到 AIDL 类后，就可以调用 AIDL 类中定义好的操作，进行跨进程请求

@OnClick(R.id.btn\_add\_person)

public void addPerson() {

Random random = new Random();

Person person = new Person("shixin" + random.nextInt(10));

try {

mAidl.addPerson(person);

List<Person> personList = mAidl.getPersonList();

mTvResult.setText(personList.toString());

} catch (RemoteException e) {

e.printStackTrace();

}

}

# Binder

上篇文章进程通信之 AIDL 中我们虽然跨进程通信成功，但是还是有很多疑问的，比如：

* AIDL 帮我们做了什么？
* 为什么要这么写？
* 什么是 Binder？

知其然还要知其所以然，一切都要从 Binder 讲起。

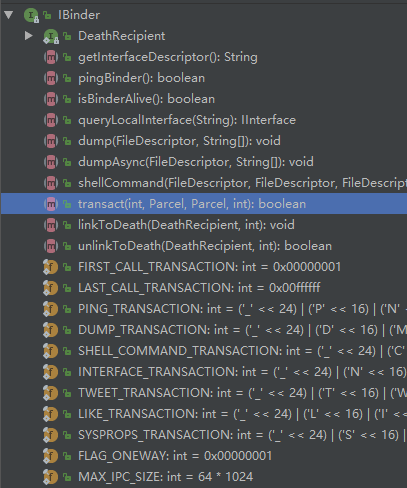
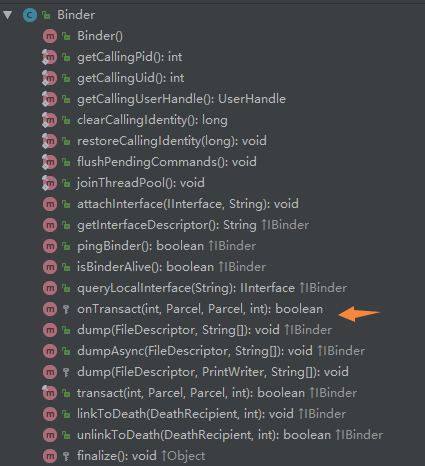
## IBinder

Binder 继承自 IBinder，所以我们先来了解下它。

public interface IBinder {...}

public class Binder implements IBinder {...}

IBinder 是一个接口，它代表了一种跨进程传输的能力。只要实现了这个接口，就能将这个对象进行跨进程传递。IBinder 是高性能、轻量级远程调用机制的核心部分，它定义了远程操作对象的基本接口。

这些方法中最关键的一个是 transact():

public boolean transact(int code, Parcel data, Parcel reply, int flags) throws RemoteException;

与它对应的是 Binder.onTransact()

protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException{...}

可以看到这两个方法非常相似，介绍一下方法中的各个参数：

code：要执行的动作，类似 Handler 的 msg.what，IBinder 中定义了以下几个 code

* PING\_TRANSACTION，表示要调用 pingBinder() 方法
* DUMP\_TRANSACTION，表示要获取 Binder 内部状态
* SHELL\_COMMAND\_TRANSACTION，执行一个 shell 命令
* INTERFACE\_TRANSACTION，询问被调用方的接口描述符号
* TWEET\_TRANSACTION
* LIKE\_TRANSACTION
* 如果我们需要自定义 code，code 的值范围需要在 FIRST\_CALL\_TRANSACTION(0x00000001) 和 LAST\_CALL\_TRANSACTION(0x00ffffff) 之间

data, reply：传入的参数和返回的值

flags：表示是否需要阻塞等待返回值，有两个值

* 0
* FLAG\_ONEWAY (0x00000001)，表示 Client 的 transact() 是单向调用，执行后立即返回

1. 经常的场景是，我们在客户端调用 IBinder.transact() 给一个服务端IBinder 对象发送请求，然后经过服务端Binder.onTransact() 得到调用，接着远程操作的客户端目标得到对应的调用。这个过程不仅在同一进程中可以进行，在跨进程（IPC）间也可以完成。IBinder.transact() 方法是同步的，它被调用后一直到 Binder.onTransact() 调用完成后才返回。
2. 通过 IBinder.transact() 方法传输的数据被保存为一个 Parcel 对象，Parcel 中保存了数据以及描述数据的元数据，元数据在缓存区中保持了 IBinder 对象的引用，这样不同进程都可以访问同一个数据。

因此在一个 IBinder 对象写入 Parcel 对象然后发送到另一个进程时，另外那进程将这个 IBinder 对象发送回去时，原本进程接收到的 IBinder 对象和开始发送出去的是同一个引用。(缓存区中的IBinder)

在跨进程传输后引用没有改变，这是非常关键的一点，这就使得 IBinder/Binder 对象在跨进程通信时可以作为唯一的标识（比如作为 token 什么的）。

1. 系统在每个进程中都有一个处理事物的线程池，这些线程用于调度其他进程对当前进程的跨进程访问。比如说进程 A 对进程 B 发起 IPC 时，A 中调用 IBinder.transact() 的线程会阻塞。B 中的事物线程池收到 A 的 IPC，调用目标对象的 Binder.onTransact() 方法，然后返回带结果的 Parcel。一旦接收到结果，A 中阻塞的线程得以继续执行。这个过程和线程通信非常相似吧。
2. Binder 机制还支持进程间的递归调用。

比如，进程 A 向进程 B 发起 IPC，而进程 B 在其 Binder.onTransact() 中又用 transact() 向进程 A 发起 IPC，那么进程 A 在等待它发出的调用返回的同时，也会响应 B 的调用，对调用的对象执行 Binder.onTransact() 方法。这种机制可以让我们觉得到跨进程的调用与进程内的调用没什么区别，这是非常重要的。

1. 在跨进程通信时，我们常常想要知道另外进程是否可用，IBinder 提供了三个检查的方法：

* transact() 当你调用的 IBinder 所在进程不存在时，会抛出 RemoteException 异常
* pingBinder()当远程进程不存在时该方法返回 false
* linkToDeath()这个方法可以向 IBinder 中注册一个 IBinder.DeathRecipient，它将在 IBinder 所在的进程退出时被调用

/\*\*

\* 检查远程 Binder 对象是否存在, 当不存在时返回 false

\*/

public boolean pingBinder();

/\*\*

\* 注册一个 Binder 销毁的监听

\* 如果一个 Binder 被销毁（通常是它所在的进程被关闭），会回调 DeathRecipient 的 BinderDied 方法

\* 注意，只会监听远程的 Binder，本地 Binder 一般不会销毁，除非当前进程退出

\* 如果要注册的 Binder 进程已经销毁，就抛出 RemoteException

\*/

public void linkToDeath(DeathRecipient recipient, int flags) throws RemoteException;

/\*\*

\* linkToDeath 注册监听回调的接口

\*/

public interface DeathRecipient {

public void binderDied();

}

## Binder

官方文档中建议：日常开发中一般不需要我们再实现 IBinder，直接使用系统提供的 Binder 即可。

Binder 实现了 IBinder 定义的操作，它是 Android IPC 的基础，平常接触到的各种 Manager（ActivityManager, ServiceManager 等），以及绑定 Service 时都在使用它进行跨进程操作。它的存在不会影响一个应用的生命周期，只要创建它的进程在运行它就一直可用。

通常我们需要在顶级的组件（Service, Activity, ContentProvider）中使用它，这样系统才知道你的进程应该一直被保留。下面介绍 Binder 的几个关键方法：

### 实现 IBinder 的 transact() 方法：

public final boolean transact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

if (false) Log.v("Binder", "Transact: " + code + " to " + this);

if (data != null) {

data.setDataPosition(0);

}

boolean r = onTransact(code, data, reply, flags);

if (reply != null) {

reply.setDataPosition(0);

}

return r;

}

可以看到，这个方法就是调用 onTransact() ，然后将返回的结果再返回回去。

### 接着看看 onTransact() 方法：

protected boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {

if (code == INTERFACE\_TRANSACTION) { //获取接口描述

reply.writeString(getInterfaceDescriptor());

return true;

} else if (code == DUMP\_TRANSACTION) { //获取当前状态

ParcelFileDescriptor fd = data.readFileDescriptor();

String[] args = data.readStringArray();

if (fd != null) {

try {

dump(fd.getFileDescriptor(), args);

} finally {

IoUtils.closeQuietly(fd);

}

}

// Write the StrictMode header.

if (reply != null) {

reply.writeNoException();

} else {

StrictMode.clearGatheredViolations();

}

return true;

} else if (code == SHELL\_COMMAND\_TRANSACTION) { //执行 shell 脚本

ParcelFileDescriptor in = data.readFileDescriptor();

ParcelFileDescriptor out = data.readFileDescriptor();

ParcelFileDescriptor err = data.readFileDescriptor();

String[] args = data.readStringArray();

ResultReceiver resultReceiver = ResultReceiver.CREATOR.createFromParcel(data);

try {

if (out != null) {

shellCommand(in != null ? in.getFileDescriptor() : null,

out.getFileDescriptor(),

err != null ? err.getFileDescriptor() : out.getFileDescriptor(),

args, resultReceiver);

}

} finally {

IoUtils.closeQuietly(in);

IoUtils.closeQuietly(out);

IoUtils.closeQuietly(err);

// Write the StrictMode header.

if (reply != null) {

reply.writeNoException();

} else {

StrictMode.clearGatheredViolations();

}

}

return true;

}

return false;

}

也没看出什么特别的，系统的 Binder.onTransact() 方法只定义了系统要进行的操作，我们如果创建自己的 Binder 时，就需要重写这个方法，根据 code 对传入的参数 data 做相应的处理，然后写入 reply，这样就能返回操作后的数据。

### 另外一个关键的方法 attachInterface：

public void attachInterface(IInterface owner, String descriptor) {

mOwner = owner;

mDescriptor = descriptor;

}

/\* mObject is used by native code, do not remove or rename \*/

private long mObject;

private IInterface mOwner;

private String mDescriptor;

这个方法的作用是将一个描述符、特定的IInterface 与当前 Binder 绑定起来，这样后续调用 queryLocalInterface 就可以拿到这个 IInterface，那 IInterface 又是什么呢？

public interface IInterface {

/\*\*

\* Retrieve the Binder object associated with this interface.

\* You must use this instead of a plain cast, so that proxy objects

\* can return the correct result.

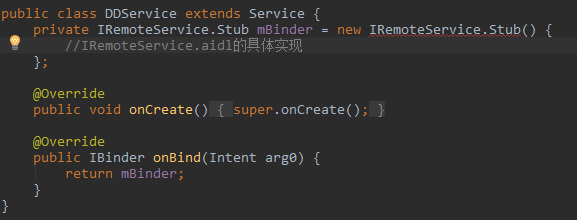
\*/

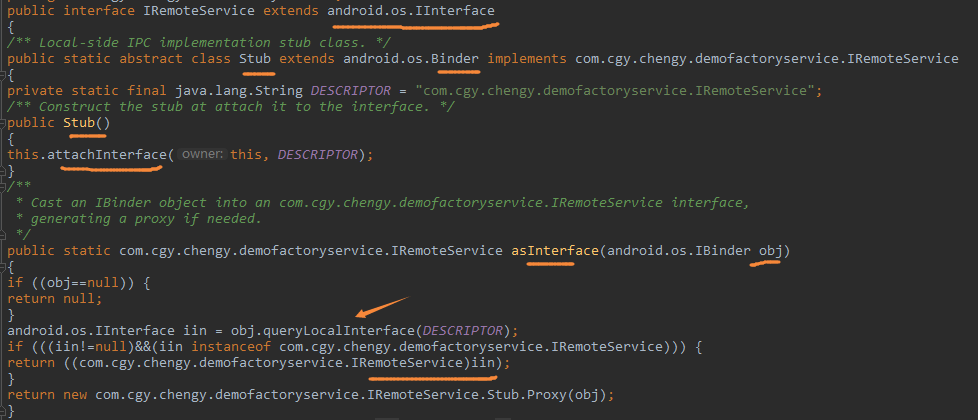
public IBinder asBinder();

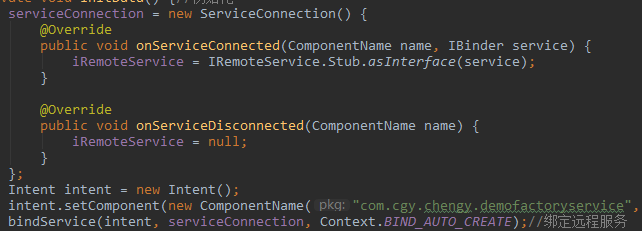
}

其实看名字就可以大概猜出来，IInterface 应该就是进程间通信定义的通用接口，我们通过定义接口，然后再服务端实现接口、客户端调用接口，就可以实现跨进程通信。IInterface 里只定义了一个 asBinder() 方法，这个方法可以返回当前接口关联的 Binder 对象。

定义IRemoteService.aidl后，系统自动生成的java文件，在客户端调用IRemoteService iRemoteService = IRemoteService.Stub.asInterface(service);后，就获取到了实现了IInterface 的IRemoteService







## Binder 通信机制

上面介绍了 Binder 类以及相关的方法，但是这只是 Binder 机制的最基础部分。我们平常看的文章或者面试时，说的 Binder 其实是范围更大的整个“Binder 消息通信机制”。借用老罗的 Android进程间通信（IPC）机制Binder简要介绍和学习计划 中对 Binder 通信机制的介绍：



在 Android 系统的 Binder 机制中，由四个组件组成，分别是：

* Client
* Server
* ServiceManager：提供辅助管理 Server 的功能
* Binder Driver：整个机制的核心

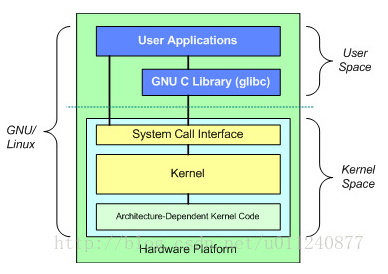
### Binder Driver

驱动程序一般指的是设备驱动程序（Device Driver），是一种可以使计算机和设备通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作。

我们知道，在 Linux 系统中，内存空间分为两部分：

用户空间：运行着应用程序

内核空间：运行着系统内核和驱动



用户空间中的进程无法直接访问内核空间，需要通过上图中的 System Call Interface （系统调用接口），通过这个统一入口，所有资源访问都是在内核的控制下执行，这样可以避免用户程序对系统资源的越权访问，从而保障了系统的安全和稳定。

同样的，用户空间中的进程直接也不可以直接访问数据，需要通过内核空间进行中转。

在 Binder 机制中，由 Binder 驱动负责完成这个中转操作，主要过程如下：

* 当 Client 向 Server 发起 IPC 请求时，Client 会先将请求数据从用户空间拷贝到内核空间
* 数据被拷贝到内核空间之后，驱动程序将内核空间中的数据拷贝到 Server 位于用户空间的缓存中

这样，就成功的将 Client 进程中的请求数据传递到了 Server 进程中。

实际上，Binder 驱动是整个 Binder 机制的核心。除了实现数据传输之外，Binder 驱动还是实现线程控制(通过中断等待队列实现线程的等待/唤醒)，以及 UID/PID 等安全机制的保证。

### Service Manager

ServiceManager 运行在用户空间，它负责管理 Service 注册与查询。

public final class ServiceManager {

private static final String TAG = "ServiceManager";

private static IServiceManager sServiceManager;

private static HashMap<String, IBinder> sCache = new HashMap<String, IBinder>();

private static IServiceManager getIServiceManager() {

if (sServiceManager != null) {

return sServiceManager;

}

// Find the service manager

sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());

return sServiceManager;

}

/\*\*

\* 根据 Service 名称获取 Service

\*/

public static IBinder getService(String name) {

try {

IBinder service = sCache.get(name);

if (service != null) {

return service;

} else {

//如果不存在就去 IServiceManager 中找，这时可能会阻塞

return getIServiceManager().getService(name);

}

} catch (RemoteException e) {

Log.e(TAG, "error in getService", e);

}

return null;

}

/\*\*

\* 添加一个 Service 到 Manager 中

\*/

public static void addService(String name, IBinder service) {

try {

getIServiceManager().addService(name, service, false);

} catch (RemoteException e) {

Log.e(TAG, "error in addService", e);

}

}

/\*\*

\* 添加一个 Service 到 Manager 中,如果 allowIsolated 为 true 表示运行在沙盒中的进程也可以访问这个 Service

\*/

public static void addService(String name, IBinder service, boolean allowIsolated) {

try {

getIServiceManager().addService(name, service, allowIsolated);

} catch (RemoteException e) {

Log.e(TAG, "error in addService", e);

}

}

/\*\*

\* Retrieve an existing service called @a name from the

\* service manager. Non-blocking.

\*/

public static IBinder checkService(String name) {

try {

IBinder service = sCache.get(name);

if (service != null) {

return service;

} else {

return getIServiceManager().checkService(name);

}

} catch (RemoteException e) {

Log.e(TAG, "error in checkService", e);

return null;

}

}

/\*\*

\* 获取 Service 列表

\*/

public static String[] listServices() {

try {

return getIServiceManager().listServices();

} catch (RemoteException e) {

Log.e(TAG, "error in listServices", e);

return null;

}

}

/\*\*

\* 当前进程首次被 activity manager 创建时调用这个方法

\*/

public static void initServiceCache(Map<String, IBinder> cache) {

if (sCache.size() != 0) {

throw new IllegalStateException("setServiceCache may only be called once");

}

sCache.putAll(cache);

}

}

可以看到 ServiceManager 提供了 Service 的添加和查询，其中操作都是通过 IServiceManager，它是何方神圣？

public interface IServiceManager extends IInterface {

/\*\*

\* 获取一个 Service，不存在就会阻塞几秒

\*/

public IBinder getService(String name) throws RemoteException;

/\*\*

\* 不阻塞的获取 Service

\*/

public IBinder checkService(String name) throws RemoteException;

public void addService(String name, IBinder service, boolean allowIsolated) throws RemoteException;

public String[] listServices() throws RemoteException;

/\*\*

\* 为 Service Manager 添加权限，具体作用暂不追究

\*/

public void setPermissionController(IPermissionController controller) throws RemoteException;

static final String descriptor = "android.os.IServiceManager";

//定义了一些用于调用 transact() 方法的 code

int GET\_SERVICE\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION;

int CHECK\_SERVICE\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION+1;

int ADD\_SERVICE\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION+2;

int LIST\_SERVICES\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION+3;

int CHECK\_SERVICES\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION+4;

int SET\_PERMISSION\_CONTROLLER\_TRANSACTION = IBinder.FIRST\_CALL\_TRANSACTION+5;

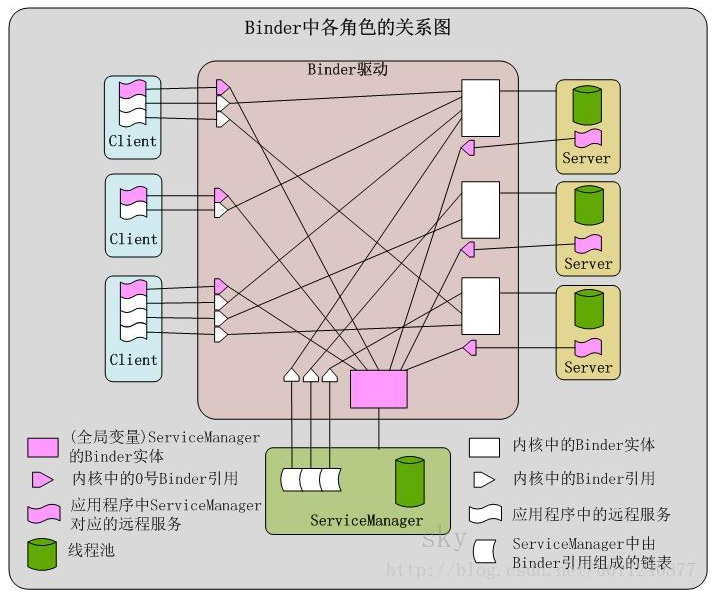
}

可以看到 IServiceManager 是一个接口，它定义了管理 Service 的一些方法，同时继承了 IInterface。最常见的实现是 BnServiceManager.getDefault()。

### Binder 机制跨进程通信流程

上面两节简单介绍了 Binder 机制中非常重要的两部分，ServiceManager 和 Binder 驱动。

在 Binder 机制的四个部分中， Client、Server 和 ServiceManager 运行在用户空间，Binder 驱动程序运行内核空间，Binder 就是一种把这四个组件粘合在一起的粘结剂。这个流程是如何进行的呢？借用 Android Binder机制(一) Binder的设计和框架 的图片：



#### Binder实体

* Binder 实体实际上是内核中 binder\_node 结构体的对象，它的作用是在内核中保存 Server 和ServiceManager 的信息(例如，Binder 实体中保存了 Server 对象在用户空间的地址)
* Binder 实体是 Server 在 Binder 驱动中的存在形式，内核通过 Binder 实体可以找到用户空间的Server对象
* 在上图中，Server 和 ServiceManager 在 Binder 驱动中都对应的存在一个 Binder 实体

#### Binder 引用

* Binder引用实际上是内核中 binder\_ref 结构体的对象，是某一个 Binder 实体的引用，通过Binder 引用可以在内核中找到对应的 Binder 实体
* 如果将 Server 看作是 Binder 实体的话，那么 Client 就好比 Binder 引用，Client 通过保存一个Server 对象的 Binder 引用，再通过该 Binder 引用在内核中找到对应的 Binder 实体，进而找到Server 对象，然后将通信内容发送给 Server 对象

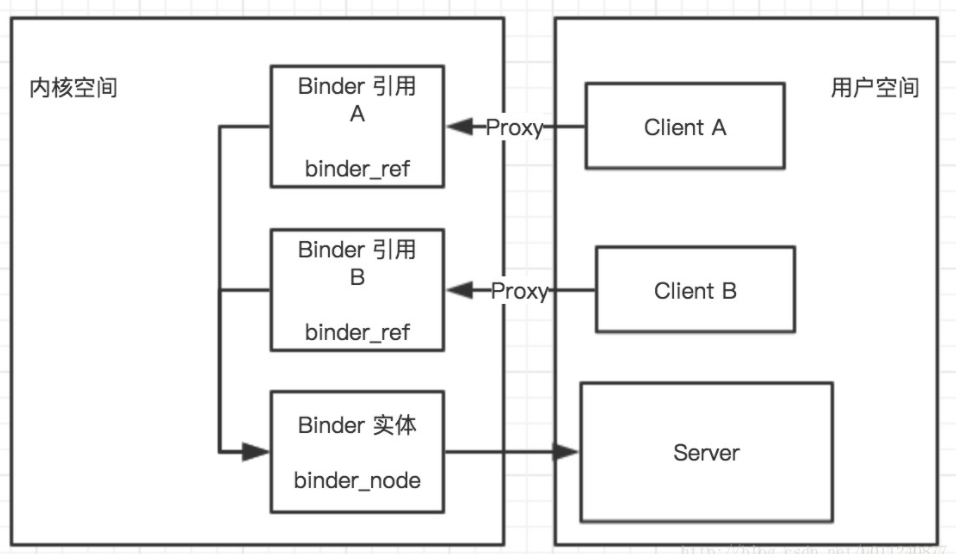
#### 远程服务

* 本地服务的代理，通过调用远程服务可以间接调用本地服务

1.Client、Server 和 ServiceManager 处于用户空间的不同进程。

2.Binder 实体和 Binder 引用都是内核(即 Binder 驱动)中的数据结构。

它们的关系如下：



### Binder 跨进程通讯流程主要为如下 4 步：

1. ServiceManager 初始化

* 当该应用程序启动时，ServiceManager 会和 Binder 驱动进行通信，告诉 Binder 驱动它是服务管理者
* Binder 驱动新建 ServiceManager 对应的 Binder 实体

1. Server 向 ServiceManager 注册自己

* Server 向 Binder 驱动发起注册请求，Binder 为它创建 Binder 实体
* 然后如果 ServiceManager 中没有这个 Server 时就添加 Server 名称与 Binder 引用到它的 Binder 引用表

1. Client 获取远程服务

* Client 首先会向 Binder 驱动发起获取服务的请求，传递要获取的服务名称
* Binder 驱动将该请求转发给 ServiceManager 进程
* ServiceManager 查找到 Client 需要的 Server 对应的 Binder 实体的 Binder 引用信息，然后通过 Binder 驱动反馈给 Client
* Client 收到 Server 对应的 Binder 引用后，会创建一个 Server 对应的远程服务（即 Server 在当前进程的代理）

1. Client 通过代理调用 Server

* Client 调用远程服务，远程服务收到 Client 请求之后，会和 Binder 驱动通信
* 因为远程服务中有 Server 的 Binder 引用信息，因此驱动就能轻易的找到对应的 Server，进而将Client 的请求内容发送 Server

## Binder 机制的优点

对比 Linux 上的其他进程通信方式（管道/消息队列/共享内存/信号量/Socket），Binder 机制的优点有以下几点：

高效简单

通过驱动在内核空间拷贝数据，不需要额外的同步处理。对比 Socket 等传输效率高

安全

Binder 机制为每个进程分配了 UID/PID 来作为鉴别身份的标示，并且在 Binder 通信时会根据UID/PID 进行有效性检测

Client/Server 架构

这种架构使得通讯更为简单

## 总结

借用《Android 开发艺术探索》对 Binder 的概括：

* 从代码角度来看，Binder 是一个类，实现了 IBinder 接口；
* 从来源看，Binder 来自于 OpenBinder,是 Android IPC 机制中的一种，Binder 还可以理解成一个虚拟物理设备，设备驱动是dev/binder；
* 从 Framework 层看，Binder 是 Service Manager 连接各种Manager(ActivityManager,PackageManager…) 和相应Service (ActivityManagerService, PackageManagerService…) 的桥梁；
* 从客户端看，Binder 是客户端服务器通讯的媒介

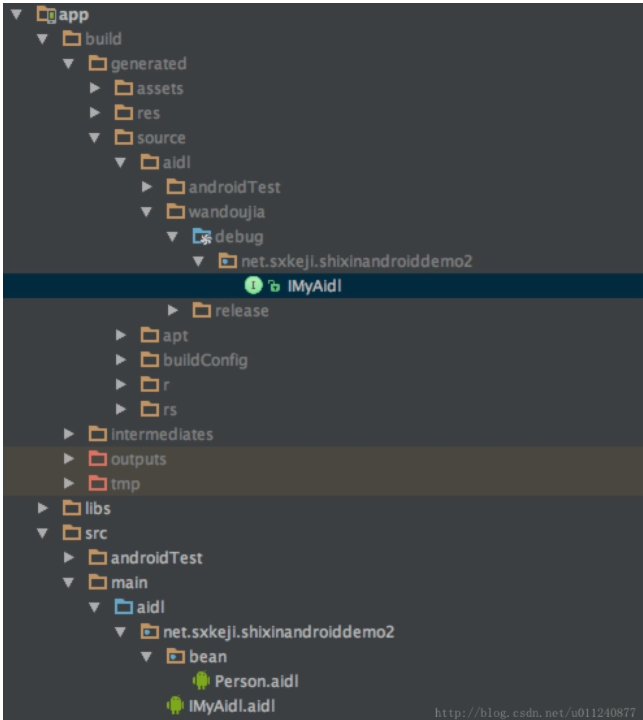
# AIDL的再次解析

## AIDL文件

前面我们讲了，在使用 AIDL 编写 IPC 代码时，们只需要编写简单的 接口 aidl 文件：

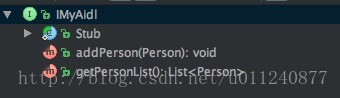


Make Project 后系统就会帮我们生成 Java 文件：



## AIDL 生成文件分析

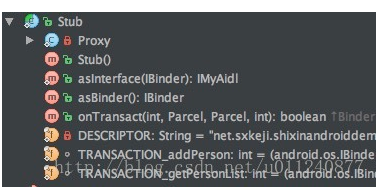
AIDL 帮我们生成内容：





可以看到，生成的接口 IMyAidl 继承了 IInterface，IInterface 是进程间通信定义的通用接口。同时 IMyAidl 中也包含了我们在 aidl 文件中声明的两个方法。除此外，IMyAidl 中还包括一个抽象类 Stub，它是一个 Binder，实现了 IMyAidl 接口：

### Stub



public static abstract class Stub extends android.os.Binder implements net.sxkeji.shixinandroiddemo2.IMyAidl {

//唯一标识，一般为完整路径

private static final java.lang.String DESCRIPTOR = "net.sxkeji.shixinandroiddemo2.IMyAidl";

/\*\*

\* 将当前接口与 Binder 绑定

\*/

public Stub() {

this.attachInterface(this, DESCRIPTOR);

}

/\*\*

\* 将一个 IBinder 转换为 IMyAidl，如果不在一个进程就创建一个代理

\*/

public static net.sxkeji.shixinandroiddemo2.IMyAidl asInterface(android.os.IBinder obj) {

if ((obj == null)) {

return null;

}

//拿着标识从本地查询接口

android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);

if (((iin != null) && (iin instanceof net.sxkeji.shixinandroiddemo2.IMyAidl))) {

return ((net.sxkeji.shixinandroiddemo2.IMyAidl) iin);

}

//查不到就返回代理

return new net.sxkeji.shixinandroiddemo2.IMyAidl.Stub.Proxy(obj);

}

//覆盖 IInterface 的方法，获取当前接口对应的 Binder 对象

@Override

public android.os.IBinder asBinder() {

return this;

}

//关键的方法，处理操作，返回

@Override

public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException {

switch (code) {

case INTERFACE\_TRANSACTION: { //获取当前接口的描述符

reply.writeString(DESCRIPTOR);

return true;

}

case TRANSACTION\_addPerson: { //执行 addPerson 方法

data.enforceInterface(DESCRIPTOR);

net.sxkeji.shixinandroiddemo2.bean.Person \_arg0;

if ((0 != data.readInt())) { //反序列化传入的数据

\_arg0 = net.sxkeji.shixinandroiddemo2.bean.Person.CREATOR.createFromParcel(data);

} else {

\_arg0 = null;

}

//调用 addPerson 方法，这个方法的实现是在服务端

this.addPerson(\_arg0);

reply.writeNoException();

return true;

}

case TRANSACTION\_getPersonList: {

data.enforceInterface(DESCRIPTOR);

java.util.List<net.sxkeji.shixinandroiddemo2.bean.Person> \_result = this.getPersonList();

reply.writeNoException();

reply.writeTypedList(\_result);

return true;

}

}

return super.onTransact(code, data, reply, flags);

}

//不在一个进程时返回的代理

private static class Proxy implements net.sxkeji.shixinandroiddemo2.IMyAidl {...}

//用于 onTransact 方法的两个 code，分别标识要进行的操作

static final int TRANSACTION\_addPerson = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);

static final int TRANSACTION\_getPersonList = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 1);

}

Stub 的几个关键内容介绍：

构造函数

* 调用了 attachInterface() 方法
* 将一个描述符、特定的 IInterface 与当前 Binder 绑定起来，这样后续调用 queryLocalInterface 就可以拿到这个IInterface
* 需要创建一个 DESCRIPTOR，一般是类的具体路径名，用于唯一表示这个 IInterface

asInterface()

* 将 IBinder 转换为 IMyAidl ，这用于返回给客户端
* 不在一个进程的话，客户端持有的是一个代理

onTransact()

* Binder 关键的处理事物方法
* 根据传入的 code，调用本地/服务端的不同方法

### Proxy

private static class Proxy implements net.sxkeji.shixinandroiddemo2.IMyAidl {

private android.os.IBinder mRemote; //代理的远端 IBinder

Proxy(android.os.IBinder remote) {

mRemote = remote;

}

//获取代理的 Binder

@Override

public android.os.IBinder asBinder() {

return mRemote;

}

public java.lang.String getInterfaceDescriptor() {

return DESCRIPTOR;

}

/\*\*

\* 代理嘛，处理下数据后直接调用实际 Binder 来处理

\*/

@Override

public void addPerson(net.sxkeji.shixinandroiddemo2.bean.Person person) throws android.os.RemoteException {

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

try {

\_data.writeInterfaceToken(DESCRIPTOR);

if ((person != null)) {

\_data.writeInt(1);

person.writeToParcel(\_data, 0);

} else {

\_data.writeInt(0);

}

//调用远端

mRemote.transact(Stub.TRANSACTION\_addPerson, \_data, \_reply, 0);

\_reply.readException();

} finally {

\_reply.recycle();

\_data.recycle();

}

}

@Override

public java.util.List<net.sxkeji.shixinandroiddemo2.bean.Person> getPersonList() throws android.os.RemoteException {

android.os.Parcel \_data = android.os.Parcel.obtain();

android.os.Parcel \_reply = android.os.Parcel.obtain();

java.util.List<net.sxkeji.shixinandroiddemo2.bean.Person> \_result;

try {

\_data.writeInterfaceToken(DESCRIPTOR);

mRemote.transact(Stub.TRANSACTION\_getPersonList, \_data, \_reply, 0);

\_reply.readException();

\_result = \_reply.createTypedArrayList(net.sxkeji.shixinandroiddemo2.bean.Person.CREATOR);

} finally {

\_reply.recycle();

\_data.recycle();

}

return \_result;

}

}

### AIDL 生成的内容小结

IInterface 类型的接口，里面有：

* Stub 抽象类
* aidl 接口定义的操作方法

Stub ，是一个 Binder，同时也是一个 IInterface，里面有：

* 将 Binder 转成 IInterface 的 asInterface() 方法
* 处理调度的 onTransact() 方法
* 用于在 onTransact() 中标识要进行的操作的两个标志
* 一个 IInterface 类型的代理

Proxy， 是IInterface 类型的代理，里面有：

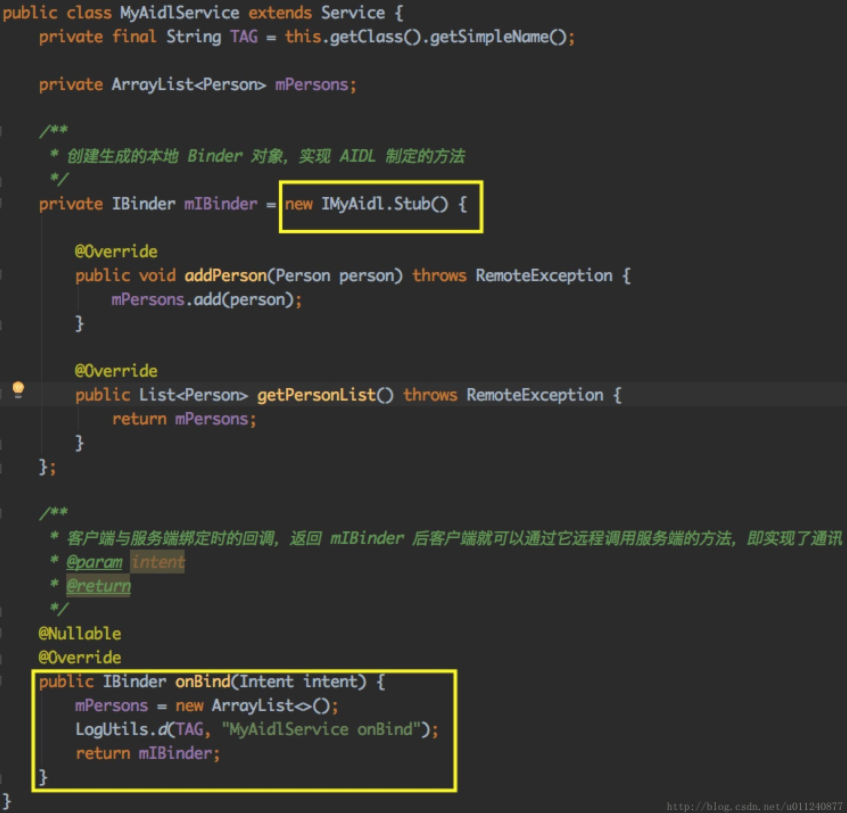
* 接口定义方法的伪实现，实际调用的是真正的 Binder 的方法

一句话总结：AIDL 帮我们生成了 Binder 和 跨平台接口的转换类 Stub，以及在不同进程时，客户端拿到的代理 Proxy。

## AIDL 的使用回顾

### 服务端

使用时先在另一个进程的 Service 中实现 AIDL 生成文件中的 Stub 类，然后在 onBind() 中返回：

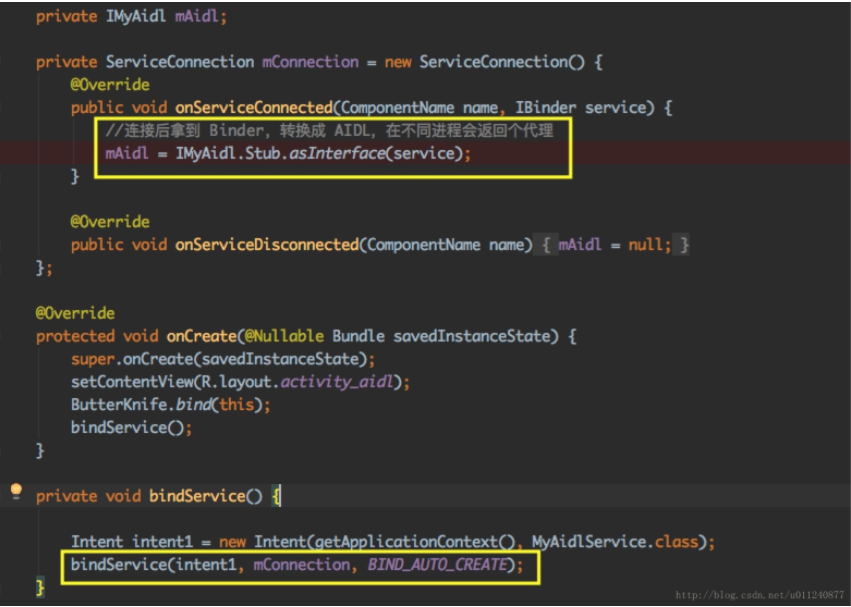


我们在服务端实例化的是 Stub 的实体，它既是 Binder 也是 IInterface。在其中实现了接口定义的方法，然后在 onBind() 中返回自己。

### 客户端

在 Activity 中使用 bindService() 绑定服务，然后再回调中，调用 Stub.asInterface() 将拿到的远端 Binder 转换为定义的接口，跨进程的话这里拿到的实际是代理接口（也就是上边的IMyAidl.Stab extend Binder）

return new net.sxkeji.shixinandroiddemo2.IMyAidl.Stub.Proxy(obj);



### 小结

根据上面的分析，我们可以看到，AIDL 帮我们做了以下几件事：

* 根据定好的接口生成不同进程都可以共同访问的接口类
* 在接口类中提供了 Binder 和接口的共同载体 Stub
* 在 Stub 中创建了代理类，用于映射调用实际接口实现

有了 AIDL，我们编写跨进程操作就变得十分简单，我们只需要关注业务接口的实现即可。

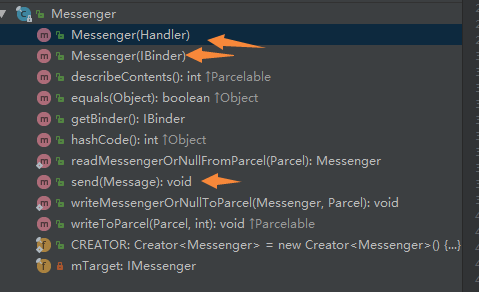


需要注意的是，客户端在发起远程请求时，当前线程会被挂起直到服务端返回，因此尽量不要在 UI 线程发起远程请求。

而在服务端，Binder 方法是运行在 Binder 线程池中的，因此可以直接使用同步的方式实现。

# Messenger

## Messenger 简介



前面我们介绍了 AIDL 的使用与原理，这篇文章来介绍下 Android 中另一种 IPC 方式：Messenger。Messenger “信使”，顾名思义，它的作用就是传递信息。Messenger 有两个构造函数：

* 以 Handler 为参数
* 以 Binder 为参数

public final class Messenger implements Parcelable { ... }

public IBinder getBinder() {  
 return mTarget.asBinder();  
}

private final IMessenger mTarget;

public Messenger(Handler target) {

mTarget = target.getIMessenger();

}

public Messenger(IBinder target) {

mTarget = IMessenger.Stub.asInterface(target); //和前面的 AIDL 很相似吧

}

看下 Handler.getIMessenger() 源码：

final IMessenger getIMessenger() {

synchronized (mQueue) {

if (mMessenger != null) {

return mMessenger;

}

mMessenger = new MessengerImpl();

return mMessenger;

}

}

这个 IMessanger 应该也是个 AIDL 生成的类吧，看下源码，果然是：

public interface IMessenger extends android.os.IInterface {

/\*\* Local-side IPC implementation stub class. \*/

public static abstract class Stub extends android.os.Binder implements android.os.IMessenger {

private static final java.lang.String DESCRIPTOR = "android.os.IMessenger";

public Stub() {

this.attachInterface(this, DESCRIPTOR);

}

public static android.os.IMessenger asInterface(...}

public android.os.IBinder asBinder() {

return this;

}

@Override

public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)

throws android.os.RemoteException {...}

private static class Proxy implements android.os.IMessenger {...}

public void send(android.os.Message msg) throws android.os.RemoteException;

}

}

IMessenger 是 AIDL 生成的跨进程接口，里面定义了一个发送消息的方法：

public void send(android.os.Message msg) throws android.os.RemoteException;

Handler 中 MessengerImpl 实现了这个方法，就是使用 Handler 将消息发出去：

private final class MessengerImpl extends IMessenger.Stub {

public void send(Message msg) {

msg.sendingUid = Binder.getCallingUid();

Handler.this.sendMessage(msg);

}

}

这就解释了为什么我们的消息来得时候会出现在 Handler.handlerMessage（） 中接着再看下 Messenger 另一个重要的方法，send()：Messenger 中持有一个 IMessenger 的引用（mTarget），在构造函数中可以通过 Handler 或者 Binder 的形式获得最终的 IMessenger 实现，然后调用它的 send() 方法。

public void send(Message message) throws RemoteException {

mTarget.send(message);//最后调用的也就是MessengerImpl的send方法

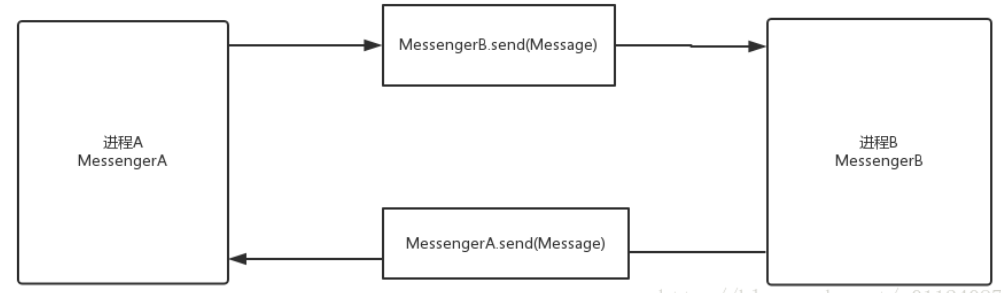
}

Messenger 其实就是 AIDL 的简化版，它把接口都封装好，我们只需在一个进程创建一个Handler 传递给 Messenger，Messenger 帮我们把消息跨进程传递到另一个进程，我们在另一个进程的 Handler 在处理消息就可以了。

## Messenger 的使用

Messenger 的使用需要结合 Handler, Message, Bundle 。

下面我们将写一个客户端跨进程发送消息到服务端的例子，服务端在收到消息后会回复，由于在 Messenger 中一个对象对应一个 Handler，所以我们需要在客户端、服务端分别创建一个 Messenger：



服务端在收到消息后会使用 Message.replyTo 对应的信使回复消息。

### 服务端

<service  
 android:name=".service.MessengerService"  
 android:exported="true"  
 android:process=":messenger"/>

服务端只需要创建一个 Messenger 对象，然后给它传递一个 Handler，在 Handler 中处理消息：

public class MessengerService extends BaseService {

private final String TAG = this.getClass().getSimpleName();

Messenger mMessenger = new Messenger(new Handler() {

@Override

public void handleMessage(final Message msg) {

if (msg != null && msg.arg1 == ConfigHelper.MSG\_ID\_CLIENT) {

if (msg.getData() == null) return;

String content = (String) msg.getData().get(ConfigHelper.MSG\_CONTENT); //接收客户端的消息

LogUtils.d(TAG, "Message from client: " + content);

//回复消息给客户端

Message replyMsg = Message.obtain();

replyMsg.arg1 = ConfigHelper.MSG\_ID\_SERVER;

Bundle bundle = new Bundle();

bundle.putString(ConfigHelper.MSG\_CONTENT, "听到你的消息了，请说点正经的");

replyMsg.setData(bundle);

try {

msg.replyTo.send(replyMsg); //回信

} catch (RemoteException e) {

e.printStackTrace();

}

}

}

});

@Nullable

@Override

public IBinder onBind(final Intent intent) {

return mMessenger.getBinder();// mTarget.asBinder() -> IMessenger.Stub.this

}

}

### 客户端

public class IPCTestActivity extends BaseActivity {

private final String TAG = this.getClass().getSimpleName();

@BindView(R.id.tv\_result)

TextView mTvResult;

@BindView(R.id.btn\_add\_person)

Button mBtnAddPerson;

@BindView(R.id.et\_msg\_content)

EditText mEtMsgContent;

@BindView(R.id.btn\_send\_msg)

Button mBtnSendMsg;

/\*\*

\* 客户端的 Messenger

\*/

Messenger mClientMessenger = new Messenger(new Handler() {

@Override

public void handleMessage(final Message msg) {

if (msg != null && msg.arg1 == ConfigHelper.MSG\_ID\_SERVER){

if (msg.getData() == null){

return;

}

String content = (String) msg.getData().get(ConfigHelper.MSG\_CONTENT);

LogUtils.d(TAG, "Message from server: " + content);

}

}

});

//服务端的 Messenger

private Messenger mServerMessenger;

private ServiceConnection mMessengerConnection = new ServiceConnection() {

@Override

public void onServiceConnected(final ComponentName name, final IBinder service) {

mServerMessenger = new Messenger(service);

}

@Override

public void onServiceDisconnected(final ComponentName name) {

mServerMessenger = null;

}

};

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_aidl);

ButterKnife.bind(this);

bindAIDLService();

bindMessengerService();

}

private void bindMessengerService() {

Intent intent = new Intent(this, MessengerService.class);

bindService(intent, mMessengerConnection, BIND\_AUTO\_CREATE);

}

@OnClick(R.id.btn\_send\_msg)

public void sendMsg() {

String msgContent = mEtMsgContent.getText().toString();

msgContent = TextUtils.isEmpty(msgContent) ? "默认消息" : msgContent;

Message message = Message.obtain();

message.arg1 = ConfigHelper.MSG\_ID\_CLIENT;

Bundle bundle = new Bundle();

bundle.putString(ConfigHelper.MSG\_CONTENT, msgContent);

message.setData(bundle);

message.replyTo = mClientMessenger; //指定回信人是客户端定义的

try {

mServerMessenger.send(message);

} catch (RemoteException e) {

e.printStackTrace();

}

}

@Override

protected void onDestroy() {

super.onDestroy();

unbindService(mMessengerConnection);

}

}

## 使用小结

### 客户端的操作主要有 3 步：

* 创建客户端的 Messenger，传递一个 Handler 处理消息
* bindService，在 ServiceConnection 回调中拿到服务端的 Messenger
* 发送消息

1. Message.obtain() 消息池里获取一个空闲消息对象
2. 给Message设置数据
3. 指定回信的Messenger message.replyTo = mClientMessenger
4. 调用服务端Messenger，发射！mServerMessenger.send(message)

### 服务端的操作主要有4步

* 创建一个Service
* 创建服务端的Messenger，传递一个Handler处理消息
* 在Service的onBind方法中返回Messenger.getBinder()
* 接收和发送消息

1. 在handlerMessage中接收消息
2. Message.obtain() 消息池里获取一个空闲消息对象
3. 给Message设置数据
4. 通过message.replyTo获取客户端Messenger
5. 调用客户端端Messenger，发射！mServerMessenger.send(message)

## 总结

Messenger 对 AIDL 进行了封装，也就是对 Binder 的封装，我们可以使用它的实现来完成基于消息的跨进程通信，就和使用 Handler 一样简单。使用时和 Binder 一样，建议在四大组件中使用，那样可以提高优先级，让系统不随便关闭当前进程。

# 控制反转、依赖注入

控制反转（IOC: Inverse Of Control），依赖注入（DI: Dependency Injection）

## 通过Dagger2来看看他们之间的关系

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

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

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

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

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

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

## 通过《墨攻》电影来看看他们之间的关系

其中有一个场景：当刘德华所饰演的墨者革离到达梁国都城下，城上梁国守军问到：“来者何人？”刘德华回答：“墨者革离！”我们不妨通过一个Java类为这个“城门叩问”的场景进行编剧，并借此理解IoC的概念：

public class MoAttack {

public void cityGateAsk() {

//①演员直接侵入剧本

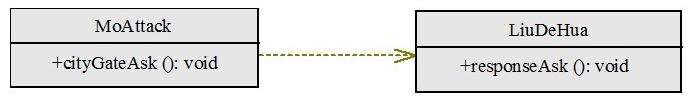
LiuDeHua ldh = new LiuDeHua();

ldh.responseAsk("墨者革离！");

}

}

我们会发现以上剧本在①处，作为具体角色饰演者的刘德华直接侵入到剧本中，使剧本和演员直接耦合在一起



通过分析，我们知道需要为该剧本主人公革离定义一个接口，这样才能换人之后也能演出

public class MoAttack {

public void cityGateAsk() {

//①引入革离角色接口

GeLi geli = new LiuDeHua();

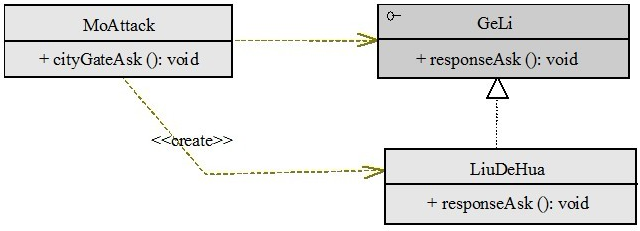
//②通过接口开展剧情

geli.responseAsk("墨者革离！");

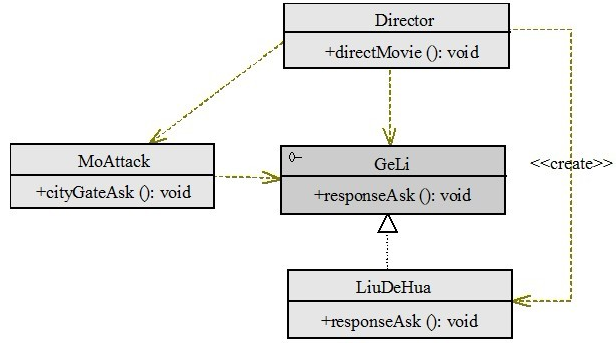
}

}

在①处引入了《墨攻》剧本的角色——革离，剧本的情节通过角色展开。在②处拍摄时革离由演员刘德华饰演。因此墨攻、革离、刘德华三者的类图关系如图



可以看出MoAttack同时依赖于GeLi接口和LiuDeHua类，并没有达到我们所期望的剧本仅依赖于角色的目的。但是角色最终必须通过具体的演员才能完成拍摄，如何让LiuDeHua和剧本无关而又能完成GeLi的具体动作呢？当然是在影片投拍时，导演将LiuDeHua安排在GeLi的角色上，导演将剧本、角色、饰演者装配起来



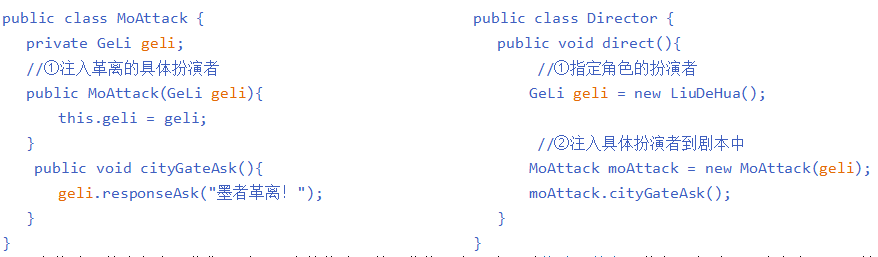
通过引入导演，使剧本和具体饰演者解耦了。对应到软件中，导演像是一个装配器，安排演员表演具体的角色。 现在我们可以反过来讲解IoC的概念了。IoC（Inverse of Control）的字面意思是控制反转，它包括两个内容： 其一是控制；其二是反转

那到底是什么东西的“控制”被“反转”了呢？对应到前面的例子，“控制”是指选择GeLi角色扮演者的控制权；“反转”是指这种控制权从《墨攻》剧本中移除，转交到导演的手中。对于软件来说，即是某一接口具体实现类的选择控制权从调用类中移除，转交给第三方决定。

因为IoC确实不够开门见山，因此业界曾进行了广泛的讨论，最终软件界的泰斗级人物Martin Fowler提出了DI（依赖注入：Dependency Injection）的概念用以代替IoC，即让调用类对某一接口实现类的依赖关系由第三方（容器或协作类）注入，以移除调用类对某一接口实现类的依赖。“依赖注入”这个名词显然比“控制反转”直接明了、易于理解。依赖注入主要是下面四种类型:

* 通过接口注入
* 通过set方法注入
* 通过构造方法注入
* 通过Java注解（Dagger2中使用该方式）

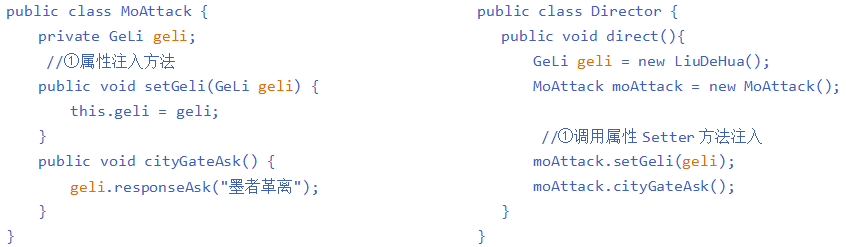
构造函数注入



在构造函数注入中，我们通过调用类的构造函数，将接口实现类通过构造函数变量传入，有时，导演会发现，虽然革离是影片《墨攻》的第一主角，但并非每个场景都需要革离的出现，在这种情况下通过构造函数注入相当于每时每刻都在革离的饰演者在场，可见并不妥当，这时可以考虑使用属性注

属性注入

属性注入可以有选择地通过Setter方法完成调用类所需依赖的注入，更加灵活方便



和通过构造函数注入革离扮演者不同，在实例化MoAttack剧本时，并未指定任何扮演者，而是在实例化MoAttack后，在需要革离出场时，才调用其setGeli()方法注入扮演者。按照类似的方式，我们还可以分别为剧本中其他诸如梁王、巷淹中等角色提供注入的Setter方法，这样，导演就可以根据所拍剧段的不同，注入相应的角色了

接口注入

将调用类所有依赖注入的方法抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。为了采取接口注入的方式，必须先声明一个ActorArrangable接口

public interface ActorArrangable {

void injectGeli(GeLi geli);

}

public class MoAttack implements ActorArrangable {

private GeLi geli;

//①实现接口方法

public void injectGeli (GeLi geli) {

this.geli = geli;

}

public void cityGateAsk() {

geli.responseAsk("墨者革离");

}

}

public class Director {

public void direct(){

GeLi geli = new LiuDeHua();

MoAttack moAttack = new MoAttack();

moAttack.injectGeli (geli);

moAttack.cityGateAsk();

}

}

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此我们不提倡采用这种方式

注解注入-通过Dagger2容器完成

虽然MoAttack和LiuDeHua实现了解耦，MoAttack无须关注角色实现类的实例化工作，但这些工作在代码中依然存在，只是转移到Director类中而已。假设某一制片人想改变这一局面，在选择某个剧本后，希望通过一个“海选”或者第三中介机构来选择导演、演员，让他们各司其职，那剧本、导演、演员就都实现解耦了。

所谓媒体“海选”和第三方中介机构在程序领域即是一个第三方的容器，它帮助完成类的初始化与装配工作，让开发者从这些底层实现类的实例化、依赖关系装配等工作中脱离出来，专注于更有意义的业务逻辑开发工作。这无疑是一件令人向往的事情，Dagger2就是这样的一个容器，它通过配置文件或注解描述类和类之间的依赖关系，自动完成类的初始化和依赖注入的工作

# 模块化、组件化、插件化

## 单工程模式

移动开发诞生，我们开发移动项目，我相信大多用的是单工程单任务的开发模式，二话不说，直接就开始写起，是不是这样呢？ new Project -> 分包 -> 写起。我相信都经历过，也写的比较爽，为什么呢？ 这种模式不涉及乱七八糟的处理方式， 上手快，开发快，足够敏捷。那么原因是什么呢？Mobile Project 刚起步，项目都偏小，一些附加业务还没绑到App上

## 模块化

Android Studio出来了，多出来了一个新的概念， Project, Module... 模块；当时以包的形式分离的公共包common,现在成了AS中的Module。大家都知道，Module包含两种格式: application和library。也就是说，一个Module就是一个小的项目，也是AS概念中的模块。因此我们开始设计common模块， common\_business模块，甚至db模块。模块的好处是什么？ 相比于包来讲，模块更灵活，耦合更低，随意插拔，想引入哪个就引入哪个。根据不同的关注点，将一个项目的可以共享的部分抽取出来，形成独立的Module，就是模块化。模块化不只包含公共部分，当然也可以是业务模块。

## 组件化

模块化和组件化有什么区别？他们的区别很小，但并不是完全相同的概念。组件化是建立在模块化思想上的一次演进，一个变种。模块化粒度更小,更侧重于重用,而组件化粒度稍大,更侧重于业务解耦

组件化本来就是模块化的概念。但是组件化的核心是什么？ 是模块角色的可转换性。在打包时是library; 在调试时是application。

将一个app分成多个模块，每个模块都是一个组件（Module），开发的过程中我们可以让这些组件相互依赖或者单独调试部分组件等，但是最终发布的时候是将这些组件合并统一成一个apk，这就是组件化开发。

正常一个App中可以有多个module，但是一般只会有一个module是设置为application的，其他均设置为library，组件化开发就是要每个module都可以运行起来，因此在开发期间(Debug版本)每个module均设置为application，发布时(Release版本)设置为libs再进行合并。

## 插件化

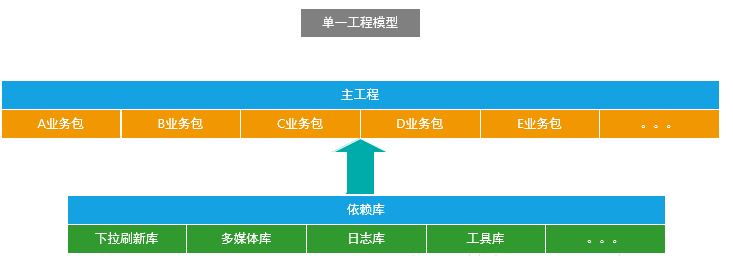
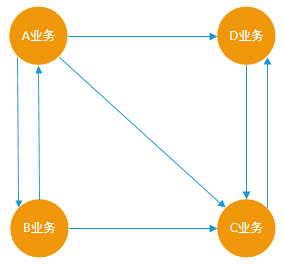
插件化严格意义来讲，其实也算是模块化的观念。将一个完整的工程，按业务划分为不同的插件，都是分治法的一种体现。化整为零，相互配合，越小的模块越容易维护。但插件化和组件化就不一个概念了。那么，到底有什么区别呢？

* 组件化的单位是组件（module）。
* 插件化的单位是apk(一个完整的应用)。
* 组件化实现的是解耦与加快编译， 隔离不需要关注的部分。
* 插件化实现的也是解耦与加快编译，同时实现热插拔也就是热更新。
* 组件化的灵活性在于按加载时机切换，分离出独立的业务组件，比如微信的朋友圈
* 插件化的灵活性在于是加载apk, 完全可以动态下载，动态更新，比组件化更灵活。
* 组件化能做的只是，朋友圈已经有了，我想单独调试，维护，和别人不耦合。但是和整个项目还是有关联的。
* 插件化可以说朋友圈就是一个app, 我需要整合了，把它整合进微信这个大的app里面
* 组本来就是一个系统，你把微信分为朋友圈，聊天， 通讯录按意义上划为独立模块，但并不是真正意义上的独立模块。
* 插本来就是不同的apk， 你把微信的朋友圈，聊天，通讯录单独做一个完全独立的app, 需要微信的时候插在一起，就是一个大型的app了。
* 其实从框架名称就可以看出： 组 和 插。
* 插件化的加载是动态的，这点很重要，也是灵活的根源

# 如何实现组件化

## 为什么要项目组件化

随着APP版本不断的迭代，新功能的不断增加，业务也会变的越来越复杂，APP业务模块的数量有可能还会继续增加，而且每个模块的代码也变的越来越多，这样发展下去单一工程下的APP架构势必会影响开发效率，增加项目的维护成本，每个工程师都要熟悉如此之多的代码，将很难进行多人协作开发，而且Android项目在编译代码的时候电脑会非常卡，又因为单一工程下代码耦合严重，每修改一处代码后都要重新编译打包测试，导致非常耗时，最重要的是这样的代码想要做单元测试根本无从下手，所以必须要有更灵活的架构代替过去单一的工程架构。

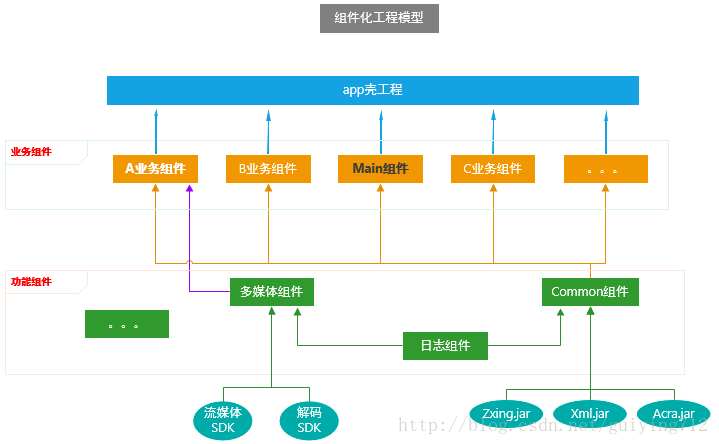
上图是目前比较普遍使用的Android APP技术架构，往往是在一个界面中存在大量的业务逻辑，而业务逻辑中充斥着各种网络请求、数据操作等行为，整个项目中也没有模块的概念，只有简单的以业务逻辑划分的文件夹，并且业务之间也是直接相互调用、高度耦合在一起的；

上图单一工程模型下的业务关系，总的来说就是：你中有我，我中有你，相互依赖，无法分离。 然而随着产品的迭代，业务越来越复杂，随之带来的是项目结构复杂度的极度增加，此时我们会面临如下几个问题：

* 实际业务变化非常快，但是单一工程的业务模块耦合度太高，牵一发而动全身；
* 对工程所做的任何修改都必须要编译整个工程；
* 功能测试和系统测试每次都要进行；
* 团队协同开发存在较多的冲突.不得不花费更多的时间去沟通和协调，并且在开发过程中，任何一位成员没办法专注于自己的功能点，影响开发效率；
* 不能灵活的对业务模块进行配置和组装；

为了满足各个业务模块的迭代而彼此不受影响，更好的解决上面这种让人头疼的依赖关系，就需要整改App的架构。

## 如何组件化

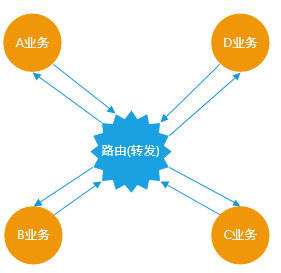


上图是组件化工程模型，为了方便理解这张架构图，下面会列举一些组件化工程中用到的名词的含义：

|  |  |
| --- | --- |
| **名词** | **含义** |
| 集成模式 | 所有的业务组件被“app壳工程”依赖，组成一个完整的APP； |
| 组件模式 | 可以独立开发业务组件，每一个业务组件就是一个APP； |
| app壳工程 | 负责管理各个业务组件，和打包apk，没有具体的业务功能； |
| 业务组件 | 根据公司具体业务而独立形成一个的工程； |
| 功能组件 | 提供开发APP的某些基础功能，例如打印日志、树状图等； |
| Main组件 | 属于业务组件，指定APP启动页面、主界面； |
| Common组件 | 属于功能组件，支撑业务组件的基础，提供多数业务组件需要的功能，例如提供网络请求功能； |

Android APP组件化架构的目标是告别结构臃肿，让各个业务变得相对独立，业务组件在组件模式下可以独立开发，而在集成模式下又可以变为arr包集成到“app壳工程”中，组成一个完整功能的APP；

从组件化工程模型中可以看到，业务组件之间是独立的，没有关联的，这些业务组件在集成模式下是一个个library，被app壳工程所依赖，组成一个具有完整业务功能的APP应用，但是在组件开发模式下，业务组件又变成了一个个application，它们可以独立开发和调试，由于在组件开发模式下，业务组件们的代码量相比于完整的项目差了很远，因此在运行时可以显著减少编译时间



这是组件化工程模型下的业务关系，业务之间将不再直接引用和依赖，而是通过“路由”这样一个中转站间接产生联系，而Android中的路由实际就是对URL Scheme的封装； 如此规模大的架构整改需要付出更高的成本，还会涉及一些潜在的风险，但是整改后的架构能够带来很多好处：

* 加快业务迭代速度，各个业务模块组件更加独立，不再出现业务耦合情况；
* 稳定的公共模块采用依赖库方式，提供给各个业务线使用，减少重复开发和维护工作量；
* 迭代频繁的业务模块采用组件方式，各业务研发可以互不干扰、提升协作效率，并控制产品质量；
* 为新业务随时集成提供了基础，所有业务可上可下，灵活多变；
* 降低团队成员熟悉项目的成本，降低项目的维护难度；
* 加快编译速度，提高开发效率；
* 控制代码权限，将代码的权限细分到更小的粒度

## 组件化实施流程

### 组件模式和集成模式的转换

Android Studio中的Module主要有两种属性，分别为：

* application属性，可以独立运行的Android程序，也就是我们的APP；

apply plugin: 'com.android.application'

* library属性，不可以独立运行，一般是Android程序依赖的库文件；

apply plugin: 'com.android.library'

Module的属性是在每个组件的 build.gradle 文件中配置的，当我们在组件模式开发时，业务组件应处于application属性，这时的业务组件就是一个 Android App，可以独立开发和调试；而当我们转换到集成模式开发时，业务组件应该处于 library 属性，这样才能被我们的“app壳工程”所依赖，组成一个具有完整功能的APP；

Gradle自动构建工具有一个重要属性，可以帮助我们完成组件在这两种模式之间自动转换。每当我们用AndroidStudio创建一个Android项目后，就会在项目的根目录中生成一个文件 gradle.properties，我们将使用这个文件的一个重要属性：在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来。

首先我们在gradle.properties中定义一个常量值 isModule（是否是组件开发模式，true为是，false为否）：

# 每次更改“isModule”的值后，需要点击 "Sync Project" 按钮  
isModule=false

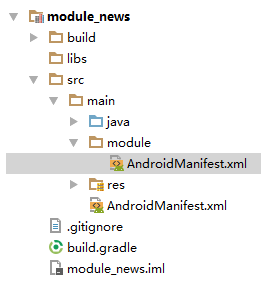
然后我们在业务组件的build.gradle中读取 isModule，但是 gradle.properties 还有一个重要属性： gradle.properties 中的数据类型都是String类型，使用其他数据类型需要自行转换；也就是说我们读到 isModule 是个String类型的值，而我们需要的是Boolean值，代码如下：

if (isModule.toBoolean()) {  
 apply plugin: 'com.android.application'  
} else {  
 apply plugin: 'com.android.library'  
}

### AndroidManifest合并问题

在 AndroidStudio 中每一个组件都会有对应的 AndroidManifest.xml，用于声明需要的权限、Application、Activity、Service、Broadcast等，当项目处于组件模式时，业务组件的 AndroidManifest.xml 应该具有一个 Android APP 所具有的的所有属性，尤其是声明 Application 和要 launch的Activity，但是当项目处于集成模式的时候，每一个业务组件的 AndroidManifest.xml 都要合并到“app壳工程”中，那么合并的时候肯定会冲突，试想一个APP怎么可能会有多个 Application 和 launch 的Activity呢？

因此我们可以为组件开发模式下的业务组件再创建一个 AndroidManifest.xml，然后根据isModule指定AndroidManifest.xml的文件路径，让业务组件在集成模式和组件模式下使用不同的AndroidManifest.xml，这样表单冲突的问题就可以规避了。



上图是组件化项目中一个标准的业务组件目录结构，首先我们在main文件夹下创建一个module文件夹用于存放组件开发模式下业务组件的 AndroidManifest.xml，而 AndroidStudio 生成的 AndroidManifest.xml 则依然保留，并用于集成开发模式下业务组件的表单；然后我们需要在业务组件的 build.gradle 中指定表单的路径，代码如下

sourceSets {  
 *main* {  
 if (isModule.toBoolean()) {  
 manifest.srcFile 'src/main/module/AndroidManifest.xml'  
 } else {  
 manifest.srcFile 'src/main/AndroidManifest.xml'  
 }  
 }  
}

这样在不同的开发模式下就会读取到不同的 AndroidManifest.xml ，然后我们需要修改这两个表单的内容以为我们不同的开发模式服务。

首先是集成开发模式下的 AndroidManifest.xml，前面我们说过集成模式下，业务组件的表单是绝对不能拥有自己的 Application 和 launch 的 Activity的，也不能声明APP名称、图标等属性，总之app壳工程有的属性，业务组件都不能有

组件模式下业务组件的 AndroidManifest.xml：

<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="cgy.com.mouone">  
 <application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:roundIcon="@mipmap/ic\_launcher\_round"  
 android:supportsRtl="true"  
 android:theme="@style/AppTheme">  
 <activity android:name=".MainActivity">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
 </application>  
</manifest>

集成模式下业务组件的 AndroidManifest.xml：

<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="cgy.com.mouone">  
 <application android:theme="@style/AppTheme">  
 <activity android:name=".GirlsActivity"/>  
 </application>  
</manifest>

这个表单中只声明了应用的主题，而且这个主题还是跟app壳工程中的主题是一致的，在这里声明主题是为了方便这个业务组件中有使用默认主题的Activity时就不用再给Activity单独声明theme了。

### 全局Context的获取及组件数据初始化

当Android程序启动时，Android系统会为每个程序创建一个 Application 类的对象，并且只创建一个，application对象的生命周期是整个程序中最长的，它的生命周期就等于这个程序的生命周期。在默认情况下应用系统会自动生成 Application 对象，但是如果我们自定义了 Application，那就需要在 AndroidManifest.xml 中声明告知系统，实例化的时候，是实例化我们自定义的，而非默认的。

但是我们在组件化开发的时候，可能为了数据的问题每一个组件都会自定义一个Application类，当所有组件要打包合并在一起的时候就会出现问题，因为最后程序只有一个 Application，我们组件中自己定义的 Application 肯定是没法使用的，因此我们需要想办法在任何一个业务组件中都能获取到全局的 Context，而且这个 Context 不管是在组件开发模式还是在集成开发模式都是生效的。

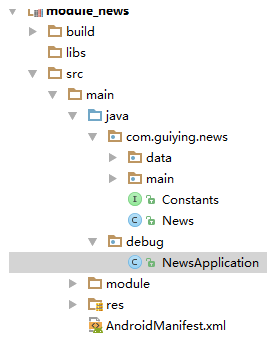
在组件化工程模型图中，功能组件集合中有一个 Common 组件， Common 有公共、公用、共同的意思，所以这个组件中主要封装了项目中需要的基础功能，并且每一个业务组件都要依赖Common组件，Common 组件就像是万丈高楼的地基，而业务组件就是在 Common 组件这个地基上搭建起来我们的APP的，Common 组件会专门在一个章节中讲解，这里只讲 Common组件中的一个功能，在Common组件中我们封装了项目中用到的各种Base类，这些基类中就有BaseApplication 类。

BaseApplication 主要用于各个业务组件和app壳工程中声明的 Application 类继承用的，只要各个业务组件和app壳工程中声明的Application类继承了 BaseApplication，当应用启动时 BaseApplication 就会被动实例化，这样从 BaseApplication 获取的 Context 就会生效，也就从根本上解决了我们不能直接从各个组件获取全局 Context 的问题；

这时候大家肯定都会有个疑问？不是说了业务组件不能有自己的 Application 吗，怎么还让他们继承 BaseApplication 呢？其实我前面说的是业务组件不能在集成模式下拥有自己的 Application，但是其实业务组件在组件开发模式下必须要有自己的 Application 类

一方面是为了让 BaseApplication 被实例化从而获取 Context，还有一个作用是，业务组件自己的 Application 可以在组件开发模式下初始化一些数据，例如在组件开发模式下，A组件没有登录页面也没法登录，因此就无法获取到 Token，这样请求网络就无法成功，因此我们需要在A组件这个 APP 启动后就应该已经登录了，这时候组件自己的 Application 类就有了用武之地，我们在组件的 Application的 onCreate 方法中模拟一个登陆接口，在登陆成功后将数据保存到本地，这样就可以处理A组件中的数据业务了；另外我们也可以在组件Application中初始化一些第三方库。

但是，实际上业务组件中的Application在最终的集成项目中是没有什么实际作用的，组件自己的 Application 仅限于在组件模式下发挥功能，因此我们需要在项目为集成模式后将组件自己的Application剔除出项目；



我们在Java文件夹下创建一个 debug 文件夹，用于存放不会在集成模式中存在的类，例如上图中的 NewsApplication ，你甚至可以在 debug 文件夹中创建一个Activity，然后组件表单中声明启动这个Activity，在这个Activity中不用setContentView，只需要在启动你的目标Activity的时候传递参数就行，这样就就可以解决组件模式下某些Activity需要getIntent数据而没有办法拿到的情况，代码如下；

class LauncherActivity: AppCompatActivity(){  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 val intent = Intent(this, GirlsActivity::class.*java*).*apply* **{** putExtra("name", "avcd");  
 putExtra("syscode", "023e2e12ed");  
 **}** startActivity(intent)  
 finish()  
 }  
}

在业务组件的 build.gradle 中，根据 isModule 是否是集成模式将 debug 这个 Java代码文件夹排除

sourceSets {  
 *main* {  
 if (isModule.toBoolean()) {  
 manifest.srcFile 'src/main/module/AndroidManifest.xml'  
 } else {  
 manifest.srcFile 'src/main/AndroidManifest.xml'  
 //集成开发模式下排除debug文件夹中的所有Java文件  
 java {  
 exclude 'debug/\*\*'  
 }  
 }  
 }  
}

### library依赖问题

在介绍这一节的时候，先说一个问题，在组件化工程模型图中，多媒体组件和Common组件都依赖了日志组件，而A业务组件有同时依赖了多媒体组件和Common组件，这时候就会有人问，你这样搞岂不是日志组件要被重复依赖了，而且Common组件也被每一个业务组件依赖了，这样不出问题吗？

其实大家完全没有必要担心这个问题，如果真有重复依赖的问题，在你编译打包的时候就会报错，如果你还是不相信的话可以反编译下最后打包出来的APP，看看里面的代码你就知道了。组件只是我们在代码开发阶段中为了方便叫的一个术语，在组件被打包进APP的时候是没有这个概念的，这些组件最后都会被打包成arr包，然后被app壳工程所依赖，在构建APP的过程中Gradle会自动将重复的arr包排除，APP中也就不会存在相同的代码了；

但是虽然组件是不会重复了，但是我们还是要考虑另一个情况，我们在build.gradle中compile的第三方库，例如Android Support库经常会被一些开源的控件所依赖，而我们自己一定也会compile Android Support库 ，这就会造成第三方包和我们自己的包存在重复加载，解决办法就是找出那个多出来的库，并将多出来的库给排除掉，而且Gradle也是支持这样做的，分别有两种方式：根据组件名排除或者根据包名排除，下面以排除support-v4库为例：

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

compile("com.jude:easyrecyclerview:$rootProject.easyRecyclerVersion") {

exclude module: 'support-v4'//根据组件名排除

exclude group: 'android.support.v4'//根据包名排除

}

}

library重复依赖的问题算是都解决了，但是我们在开发项目的时候会依赖很多开源库，而这些库可能多个组件都需要用到，要是每个组件都去依赖一遍也是很麻烦的，尤其是给这些库升级的时候，为了方便我们统一管理第三方库，我们将给给整个工程提供统一的依赖第三方库的入口，前面介绍的Common组件的作用之一就是统一依赖开源库，因为其他业务组件都依赖了Common库，所以这些业务组件也就间接依赖了Common所依赖的开源库

注意：common中的implement最好使用api代替

Android Gradle plugin 3.0 插件需要你明确的指出你是否要将某 module 的接口暴露出去。基于此它可以在判断某 module 是否需要重编译时做出正确的选择。之前的compile 已被弃用，并由两个新的替代：

api

某module通过api引用到的库，将被暴露出去供引用该 module的模块使用，含义上实际与 compile 一致。比如：B引用 A 时使用了 api ，那么 C也就可以使用到 A 的接口

implementation

某module通过api引用到的库，仅在该 module 内部使用，而不将其接口暴露给外部。比如：B引用 A 时使用了 implementation，那么 C无法使用 A 的接口



同理：testApi代替testImplement；androidTestApi代替androidTestImplement

### 调用和通信

在组件化开发的时候，组件之间是没有依赖关系，我们不能在使用显示调用来跳转页面了，因为我们组件化的目的之一就是解决模块间的强依赖问题，假如现在要从A业务组件跳转到业务B组件，并且要携带参数跳转，这时候怎么办呢？而且组件这么多怎么管理也是个问题，这时候就需要引入“路由”的概念了，由本文开始的组件化模型下的业务关系图可知路由就是起到一个转发的作用。

路由库的知识查看下一章ARouter

组件之间的调用解决后，另外需要解决的就是组件之间的通信，例如A业务组件中有消息列表，而用户在B组件中操作某个事件后会产生一条新消息，需要通知A组件刷新消息列表，这样业务场景需求可以使用Android广播来解决，也可以使用第三方的事件总线来实现，比如EventBus

### 资源名冲突

因为我们拆分出了很多业务组件和功能组件，在把这些组件合并到“app壳工程”时候就有可能会出现资源名冲突问题，例如A组件和B组件都有一张叫做“ic\_back”的图标，这时候在集成模式下打包APP就会编译出错，解决这个问题最简单的办法就是在项目中约定资源文件命名规约，比如强制使每个资源文件的名称以组件名开始，这个可以根据实际情况和开发人员制定规则。当然了万能的Gradle构建工具也提供了解决方法，通过在在组件的build.gradle中添加如下的代码：

//设置了resourcePrefix值后，所有的资源名必须以指定的字符串做前缀，否则会报错  
//但是resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名  
resourcePrefix "one\_"

## 组件化项目的工程类型

在组件化工程模型中主要有：app壳工程、业务组件和功能组件,这三种类型，而业务组件中的Main组件和功能组件中的Common组件比较特殊，下面将分别介绍。

### app壳工程

app壳工程是从名称来解释就是一个空壳工程，没有任何的业务代码，也不能有Activity，但它又必须被单独划分成一个组件，而不能融合到其他组件中，是因为它有如下几点重要功能：

* app壳工程中声明了我们Android应用的 Application，这个 Application 必须继承自 Common组件中的 BaseApplication或者直接在表单声明BaseApplication，因为只有这样，在打包应用后才能让BaseApplication中的Context生效，当然你还可以在这个 Application中初始化我们工程中使用到的库文件，还可以在这里解决Android引用方法数不能超过 65535 的限制，对崩溃事件的捕获和发送也可以在这里声明。
* app壳工程的 AndroidManifest.xml 是Android应用的根表单，应用的名称、图标以及是否支持备份等等属性都是在这份表单中配置的，其他组件中的表单最终在集成开发模式下都被合并到这份 AndroidManifest.xml 中。
* app壳工程的 build.gradle 是比较特殊的，app壳不管是在集成开发模式还是组件开发模式，它的属性始终都是：com.android.application，因为最终app壳工程要依赖其他的组件，全被打包进app壳工程中，这一点从组件化工程模型图中就能体现出来，所以app壳工程不需要单独调试单独开发。另外Android应用的打包签名，以及buildTypes和defaultConfig都需要在这里配置，而它的dependencies则需要根据isModule的值分别依赖不同的组件，在组件开发模式下app壳工程只需要依赖Common组件，或者为了防止报错也可以根据实际情况依赖其他功能组件，而在集成模式下app壳工程必须依赖所有在应用Application中声明的业务组件，并且不需要再依赖任何功能组件。

apply plugin: 'com.android.application'  
android {  
 signingConfigs {  
 release {  
 keyAlias 'guiying712'  
 keyPassword 'guiying712'  
 storeFile file('/mykey.jks')  
 storePassword 'guiying712'  
 }  
 }  
 compileSdkVersion rootProject.ext.compileSdkVersion  
 buildToolsVersion rootProject.ext.buildToolsVersion  
 defaultConfig {  
 applicationId "com.guiying.androidmodulepattern"  
 minSdkVersion rootProject.ext.minSdkVersion  
 targetSdkVersion rootProject.ext.targetSdkVersion  
 versionCode rootProject.ext.versionCode  
 versionName rootProject.ext.versionName  
 multiDexEnabled false  
 }  
 buildTypes {  
 release {  
 // 不显示Log  
 buildConfigField "boolean", "LEO\_DEBUG", "false"  
 //是否zip对齐  
 zipAlignEnabled true  
 // 缩减resource文件  
 shrinkResources true  
 //Proguard  
 minifyEnabled true  
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  
 //签名  
 signingConfig signingConfigs.release  
 }  
 debug {  
 //给applicationId添加后缀“.debug”  
 applicationIdSuffix ".debug"  
 buildConfigField "boolean", "LOG\_DEBUG", "true"  
 zipAlignEnabled false  
 shrinkResources false  
 minifyEnabled false  
 debuggable true  
 }  
 }  
}  
dependencies {  
 compile fileTree(dir: 'libs', include: ['\*.jar'])  
 if (isModule.toBoolean()) {  
 compile project(':lib\_common')  
 } else {  
 compile project(':module\_main')  
 compile project(':module\_girls')  
 compile project(':module\_news')  
 }  
}

### 功能组件和Common组件

#### 功能组件

功能组件是为了支撑业务组件的某些功能而独立划分出来的组件，功能实质上跟项目中引入的第三方库是一样的，功能组件的特征如下：

* 功能组件的 AndroidManifest.xml 是一张空表，这张表中只有功能组件的包名；
* 功能组件不管是在集成开发模式下还是组件开发模式下属性始终是：com.android.library。另外功能组件的 build.gradle 也无需设置 buildTypes ，只需要 dependencies 这个功能组件需要的jar包和开源库。

apply plugin: 'com.android.library'  
android {  
 compileSdkVersion rootProject.ext.compileSdkVersion  
 buildToolsVersion rootProject.ext.buildToolsVersion  
 defaultConfig {  
 minSdkVersion rootProject.ext.minSdkVersion  
 targetSdkVersion rootProject.ext.targetSdkVersion  
 versionCode rootProject.ext.versionCode  
 versionName rootProject.ext.versionName  
 }  
}  
dependencies {  
 compile fileTree(dir: 'libs', include: ['\*.jar'])  
}

#### Common功能组件

Common组件除了有功能组件的普遍属性外，还具有其他功能：

* Common组件的 AndroidManifest.xml 不是一张空表，这张表中声明了我们 Android应用用到的所有使用权限 uses-permission 和 uses-feature，放到这里是因为在组件开发模式下，所有业务组件就无需在自己的 AndroidManifest.xml 声明自己要用到的权限了。
* Common组件的 build.gradle 需要统一依赖业务组件中用到的第三方依赖库和jar包，例如我们用到的ActivityRouter、Okhttp等等。
* 应用的 Base类（包括BaseApplication）和网络请求工具、图片加载工具等等，公用的 widget控件也应该放在Common 组件中
* 业务组件中都用到的数据也应放于Common组件中，例如保存到 SharedPreferences 和 DataBase 中的登陆数据；
* Common组件的资源文件中需要放置项目公用的 Drawable、layout、sting、dimen、color和style 等等，另外项目中的 Activity 主题必须定义在 Common中，方便和 BaseActivity 配合保持整个Android应用的界面风格统一。

### 业务组件和Main组件

#### 业务组件

业务组件就是根据业务逻辑的不同拆分出来的组件，业务组件的特征如下：

* 业务组件中要有两张AndroidManifest.xml，分别对应组件开发模式和集成开发模式
* 业务组件在集成模式下是不能有自己的Application的，但在组件开发模式下又必须实现自己的Application并且要继承自Common组件的BaseApplication，并且这个Application不能被业务组件中的代码引用，因为它的功能就是为了使业务组件从BaseApplication中获取的全局Context生效，还有初始化数据（跳转的Intent数据）之用。
* 业务组件有debug文件夹，这个文件夹在集成模式下会从业务组件的代码中排除掉，所以debug文件夹中的类不能被业务组件引用，例如组件模式下的 Application 就是置于这个文件夹中，还有组件模式下开发给目标 Activity 传递参数的用的 LaunchActivity 也应该置于 debug 文件夹中；
* 业务组件必须在自己的 Java文件夹中创建业务组件声明类，以使 app壳工程 中的 应用Application能够引用，实现组件跳转；（！！疑问！！）
* 业务组件必须在自己的 build.gradle 中根据 isModule 值的不同改变自己的属性，在组件模式下是：com.android.application，而在集成模式下com.android.library；同时还需要在build.gradle配置资源文件，如 指定不同开发模式下的AndroidManifest.xml文件路径，排除debug文件夹等；业务组件还必须在dependencies中依赖Common组件，以及依赖其他用到的功能组件。

if (isModule.toBoolean()) {  
 apply plugin: 'com.android.application'  
} else {  
 apply plugin: 'com.android.library'  
}  
  
android {  
 compileSdkVersion rootProject.ext.compileSdkVersion  
 buildToolsVersion rootProject.ext.buildToolsVersion  
 //设置了resourcePrefix值后，所有的资源名必须以指定的字符串做前缀，否则会报错。  
 //但是resourcePrefix这个值只能限定xml里面的资源，并不能限定图片资源，所有图片资源仍然需要手动去修改资源名。  
 resourcePrefix "girls\_"  
 defaultConfig {  
 minSdkVersion rootProject.ext.minSdkVersion  
 targetSdkVersion rootProject.ext.targetSdkVersion  
 versionCode rootProject.ext.versionCode  
 versionName rootProject.ext.versionName  
 }  
 sourceSets {  
 *main* {  
 if (isModule.toBoolean()) {  
 manifest.srcFile 'src/main/module/AndroidManifest.xml'  
 } else {  
 manifest.srcFile 'src/main/AndroidManifest.xml'  
 //集成开发模式下排除debug文件夹中的所有Java文件  
 java {  
 exclude 'debug/\*\*'  
 }  
 }  
 }  
 }  
}  
dependencies {  
 compile fileTree(dir: 'libs', include: ['\*.jar'])  
 compile project(':lib\_common')  
}

#### Main业务组件

Main组件除了有业务组件的普遍属性外，还有一项重要功能：

* Main组件集成模式下的AndroidManifest.xml是跟其他业务组件不一样的，Main组件的表单中声明了我们整个Android应用的LaunchActivity，这就是Main组件的独特之处；所以我建议SplashActivity、LoginActivity以及主界面都应属于Main组件，也就是说Android应用启动后要调用的页面应置于Main组件。

## 组件化项目的混淆方案

组件化项目的Java代码混淆方案采用在集成模式下集中在app壳工程中混淆，各个业务组件不配置混淆文件。集成开发模式下在app壳工程中build.gradle文件的release构建类型中开启混淆属性，其他buildTypes配置方案跟普通项目保持一致

之所以不采用在每个业务组件中开启混淆的方案，是因为组件在集成模式下都被 Gradle 构建成了 release 类型的arr包，一旦业务组件的代码被混淆，而这时候代码中又出现了bug，将很难根据日志找出导致bug的原因；另外每个业务组件中都保留一份混淆配置文件非常不便于修改和管理，这也是不推荐在业务组件的 build.gradle 文件中配置 buildTypes （构建类型）的原因。

因此，除了App壳需要配置buildTypes，功能组件和业务组件都不用配置，直接删除buildTypes或使用默认的

buildTypes {  
 release {  
 minifyEnabled false  
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  
 }  
}

## 工程的buildgradle和gradleproperties文件

### 组件化工程的build.gradle文件

因为每个组件的 build.gradle 都需要配置 compileSdkVersion、buildToolsVersion和defaultConfig 等的版本号，为了能够保持统一，也为了方便修改版本号，统一在Android工程根目录下的build.gradle中定义这些版本号。当然为了方便管理Common组件中的第三方开源库的版本号，最好也在这里定义这些开源库的版本号

//时间：2019.1.10；每次修改版本号都要添加修改时间  
ext {  
 // Sdk and tools  
 buildToolsVersion = 28  
 compileSdkVersion = 25  
 minSdkVersion = 16  
 kotlinVersion = '1.2.51'  
 //dependencies version  
 yMengVersion = "1.5.4"  
 yMengAnalVersion = "7.5.4"

}

使用：

...

compileSdkVersion rootProject.compileSdkVersion  
 buildToolsVersion rootProject.ext.buildToolsVersion  
 defaultConfig {  
 minSdkVersion rootProject.minSdkVersion  
 targetSdkVersion rootProject.ext.targetSdkVersion

...  
 }

...  
 implementation "com.umeng.umsdk:common:$yMengVersion"  
 implementation "com.umeng.umsdk:analytics:$rootProject.yMengAnalVersion"  
 implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$rootProject.ext.kotlinVersion"

### 组件化工程的gradle.properties文件

在组件化实施流程中我们了解到gradle.properties有两个属性对我们非常有用：

* 在Android项目中的任何一个build.gradle文件中都可以把gradle.properties中的常量读取出来，不管这个build.gradle是组件的还是整个项目工程的build.gradle；
* gradle.properties中的数据类型都是String类型，使用其他数据类型需要自行转换；

利用gradle.properties的属性不仅可以解决集成开发模式和组件开发模式的转换

## 结束语

组件化相比于单一工程优势是显而易见的：

* 组件模式下可以加快编译速度，提高开发效率；
* 自由选择开发框架（MVC /MVP / MVVM /）；
* 方便做单元测试；
* 代码架构更加清晰，降低项目的维护难度；
* 适合于团队开发；

# ARouter

## 功能介绍

* 支持直接解析标准URL进行跳转，并自动注入参数到目标页面中
* 支持多模块工程使用
* 支持添加多个拦截器，自定义拦截顺序
* 支持依赖注入，可单独作为依赖注入框架使用
* 支持InstantRun
* 支持MultiDex(Google方案)
* 映射关系按组分类、多级管理，按需初始化
* 支持用户指定全局降级与局部降级策略
* 页面、拦截器、服务等组件均自动注册到框架
* 支持多种方式配置转场动画
* 支持获取Fragment
* 完全支持Kotlin以及混编(配置见文末 其他#5)
* 支持第三方 App 加固(使用 arouter-register 实现自动注册)
* 支持生成路由文档
* 提供 IDE 插件便捷的关联路径和目标类

## 典型应用

* 从外部URL映射到内部页面，以及参数传递与解析
* 跨模块页面跳转，模块间解耦
* 拦截跳转过程，处理登陆、埋点等逻辑
* 跨模块API调用，通过控制反转来做组件解耦

## 基础功能

### 添加依赖和配置

非kotlin项目配置方式

android {

defaultConfig {

...

javaCompileOptions {

annotationProcessorOptions {

arguments = [AROUTER\_MODULE\_NAME: project.getName()]

}

}

}

}

dependencies {

// 替换成最新版本, 需要注意的是api要与compiler匹配使用，均使用最新版可以保证兼容

compile 'com.alibaba:arouter-api:x.x.x'

annotationProcessor 'com.alibaba:arouter-compiler:x.x.x'

...

}

Kotlin项目中的配置方式

apply plugin: 'kotlin-kapt'

kapt {

arguments {

arg("AROUTER\_MODULE\_NAME", project.getName())

}

}

dependencies {

// 替换成最新版本, 需要注意的是api要与compiler匹配使用，均使用最新版可以保证兼容

implementation 'com.alibaba:arouter-api:x.x.x'

kapt 'com.alibaba:arouter-compiler:x.x.x'

...

}

### 添加注解

// 在支持路由的页面上添加注解(必选)

// 这里的路径需要注意的是至少需要有两级，/xx/xx

@Route(path = "/test/one")

public class YourActivity extend Activity {

...

}

### 初始化SDK

if (isDebug()) { // 这两行必须写在init之前，否则这些配置在init过程中将无效

ARouter.openLog(); // 打印日志

ARouter.openDebug(); // 开启调试模式(如果在InstantRun模式，必须开启！线上版本需要关闭,否则有安全风险)

}

ARouter.init(this); // 尽可能早，推荐在Application中初始化

### 发起路由操作

// 1. 应用内简单的跳转

ARouter.getInstance().build("/test/activity").navigation();

// 2. 跳转ForResult

ARouter.getInstance().build("/test/activity").navigation(this, 1);//第一个Context，第二个requestCode

// 3. 跳转并携带参数

ARouter.getInstance().build("/test/one")

.withLong("key1", 666L)

.withString("key3", "888")

.withParcelable("key4", new Test("Jack", "Rose"))

.navigation();

// 4. 跳转并携带Bundle

ARouter.getInstance().build("/test/one")

.with(params) //with(Bundle bundle) 方法是直接将bundle赋值进去，所以之前放的参数将会无效

.navigation();

获取值时用普通的getIntent().getStringExtra()等方法就行

注意：如果要传递object，需要在使用withObject的module中添加一个类，并且必须用注解@Autowired

ARouter.getInstance().build("/test/one")

.withObject("key4", new Test("Jack", "Rose"))

.navigation();

传递的object必须有空构造函数

public class OneBean {

public String name;

public String id;

public OneBean() {

}

public OneBean(String name, String id) {

this.name = name;

this.id = id;

}

}

/\*如果需要传递自定义对象，新建一个类实现 SerializationService,并使用@Route注解标注(方便用户自行选择序列化方式)

注意：test可以换成任何字段，这个SerializationService 类必须在使用withObjct()的module中\*/  
@Route(path = "/test/json")  
class JsonServiceOneImpl : SerializationService {  
 override fun init(context: Context) {  
 }  
 override fun <T> json2Object(text: String, clazz: Class<T>): T? {  
 return JSON.parseObject(text, clazz)  
 }  
 override fun <T : Any?> parseObject(input: String?, clazz: Type?): T {  
 return JSON.parseObject(input, clazz)  
 }  
 override fun object2Json(instance: Any): String {  
 return JSON.toJSONString(instance)  
 }  
}

// 为每一个参数声明一个字段，并使用 @Autowired 标注

// URL中不能传递Parcelable类型数据，但通过ARouter api可以传递Parcelable对象

@Route(path = "/test/activity")

public class Test1Activity extends Activity {

@Autowired

public String name; //必须是public ，如果是kotlin，需要写成 @JvmField var name: String? = null

@Autowired

int age;

@Autowired(name = "isBoy") // 通过name来映射URL中的不同参数

boolean boy;

@Autowired

OneBean obj; // 支持解析自定义对象，URL中使用json传递

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

/\* 如果需要 根据URL中的参数赋值，需要在 Activity 调用自动注入

可以不写 ARouter.getInstance().inject(this)，但是需要取值的字段仍然需要标上 @Autowired 注解，

这样 才能在 intent 中获取到对应的参数\*/

ARouter.getInstance().inject(this);

// ARouter会自动对字段进行赋值，无需主动获取

Log.d("param", name + age + boy + obj);

}

}

### 添加混淆规则

################################## ARouter

-keep public class com.alibaba.android.arouter.routes.\*\*{\*;}

-keep public class com.alibaba.android.arouter.facade.\*\*{\*;}

-keep class \* implements com.alibaba.android.arouter.facade.template.ISyringe{\*;}

# 如果使用了 byType 的方式获取 Service，需添加下面规则，保护接口

-keep interface \* implements com.alibaba.android.arouter.facade.template.IProvider

# 如果使用了 单类注入，即不定义接口实现 IProvider，需添加下面规则，保护实现

# -keep class \* implements com.alibaba.android.arouter.facade.template.IProvider

### 使用 Gradle 插件实现路由表的自动加载 (可选)

apply plugin: 'com.alibaba.arouter'

buildscript {

repositories {

jcenter()

}

dependencies {

classpath "com.alibaba:arouter-register:?"

}

}

通过 ARouter 提供的注册插件进行路由表的自动加载(<https://github.com/luckybilly/AutoRegister>)， 默认通过扫描 dex 的方式 进行加载通过 gradle 插件进行自动注册可以缩短初始化时间解决应用加固导致无法直接访问 dex 文件，初始化失败的问题

### 使用 IDE 插件导航到目标类 (可选)

在Android Studio -> setting -> Plugins -> Borwse Repositories中搜索 ARouter Helper, 或者直接下载文档上方 最新版本 中列出的 arouter-idea-plugin zip 安装包手动安装，安装后可以在跳转代码的行首找到一个图标, 点击该图标，即可跳转到标识了代码中路径的目标类

## 进阶用法

### 坑 - IProviders实现类的坑

拦截器类、json解析服务、全局降级策略类共同点

* 放在app、common中，不起作用
* 放在任一业务module中，所有业务都会起作用
* 拦截器类在app初始化时，就会执行init方法
* 拦截器priority越小，优先级越高
* json解析服务在Arouter第一次跳转时，才会执行init方法
* json解析服务的path，组名可以随便写：/lala/json
* 全局降级策略类在第一次执行到全局降级时，才会执行init方法
* 全局降级策略类的path，可以随便写：/xxx/xxx
* 拦截器类和json解析服务最好全部写在main module中
* json解析服务的转换库，最好选Gson
* 传递Object时，接受类必须用@AutoWire注解，最好inject自动赋值
* 通过Url传递Object时，也需要声明json解析服务

### 通过URL跳转

第三方或者自身App的webview中通过url跳转app，传统做法是shouldOverrideUrlLoading在硬编码跳转，或者在AndroidManifest.xml中给每个要跳转的Activity的配置协议，这种方法会导致如果需要通过url跳转的界面过多，会配置很多协议

而ARouter只需配置一个协议，用一个专门的协议中转SchameFilterActivity ，然后解析url的后缀，实现跳转

// 新建一个Activity用于监听Schame事件,之后直接把url传递给ARouter即可

public class SchameFilterActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

Uri uri = getIntent().getData();

ARouter.getInstance().build(uri).navigation();

finish();

}

}

<activity android:name=".activity.SchameFilterActivity">

<!-- Schame -->

<intent-filter>

<data

android:host="m.aliyun.com"

android:scheme="arouter"/>

<action android:name="android.intent.action.VIEW"/>

<category android:name="android.intent.category.DEFAULT"/>

<category android:name="android.intent.category.BROWSABLE"/>

</intent-filter>

</activity>

<p><a href="arouter://m.aliyun.com/main/main">MainActivity</a></p>在SchameFilterActivity 中转后，跳转到了标记有@Route(path = "/main/main")的MainActivity上

### 解析URL中的参数

如果需要 根据URL中的参数赋值，需要在 Activity 调用自动注入ARouter.getInstance().inject(this)，也可以不写 ，但是需要取值的字段仍然需要标上 @Autowired 注解，这样 才能在 intent 中获取到对应的参数

如果需要传递自定义对象，和上面withObject一样，要新建一个类实现 SerializationService

<p><a href="arouter://m.aliyun.com/test/activity?name=alex&age=18&isBoy=true&obj=%7b%22name%22%3a%22jack%22%2c%22id%22%3a666%7d">name=alex&age=18&boy=true&high=180&obj={"name":"jack","id":"666"}</a></p>将被Test1Activity解析成功

### 声明拦截器(拦截跳转过程，面向切面编程)

// 比较经典的应用就是在跳转过程中处理登陆事件，这样就不需要在目标页重复做登陆检查

// 拦截器会在跳转之间执行，多个拦截器会按优先级顺序从小到大依次执行  
@Interceptor(priority = 1, name = "拦截器1")  
public class TestInterceptorTwo implements IInterceptor {  
 @Override  
 public void process(Postcard postcard, InterceptorCallback callback) {  
 if (StrNumUtil.*has\_intterupt* && StrNumUtil.*getRandomBoolean*()) {  
 callback.onInterrupt(new RuntimeException("我觉得有点异常")); // 觉得有问题，中断路由流程  
 Log.*e*("tag", "process: 1 interrupt");  
 } else {  
 callback.onContinue(postcard); // 处理完成，交还控制权  
 Log.*e*("tag", "process: 1 pass");  
 }  
 }  
  
 @Override  
 public void init(Context context) {  
 // 拦截器的初始化，会在sdk初始化的时候调用该方法，仅会调用一次  
 Log.*e*("tag", "位于 main 中的拦截器1初始化了");  
 }  
}

为目标页面声明更多信息

// 我们经常需要在目标页面中配置一些属性，比方说"是否需要登陆"之类的，可以通过 Route 注解中的 extras 属性进行扩展

// 通过开关标记目标页面的一些属性，在拦截器中可以拿到这个标记进行判断，拦截跳往目标页面的请求

@Route(path = "/one/one", extras = ARouterConstants.*EXTRA\_NEED\_LOGIN*)  
class OneActivity : BaseActivity() { ... }

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

@Interceptor(priority = 9, name = "Extras拦截器")  
public class ExtrasInterceptor implements IInterceptor {  
 @Override  
 public void process(Postcard postcard, InterceptorCallback callback) {  
 switch (postcard.getExtra()) {  
 case ARouterConstants.*EXTRA\_NEED\_LOGIN*:  
 callback.onInterrupt(new RuntimeException("EXTRA\_NEED\_LOGIN")); // 中断路由流程  
 Log.*e*("tag", "ExtrasInterceptor interrupt");  
 break;  
 default:  
 callback.onContinue(postcard); // 处理完成，交还控制权  
 Log.*e*("tag", "ExtrasInterceptor pass");  
 break;  
 }  
 }  
 @Override  
 public void init(Context context) {  
 // 拦截器的初始化，会在sdk初始化的时候调用该方法，仅会调用一次  
 Log.*e*("tag", "位于 main 中的拦截器 extras 初始化了");  
 }  
}

### 处理跳转结果

首先会回调onLost或者onFound，如果Found了，拦截器拦住了会走onInterrupt，否则走onArrival

tv\_main.setOnClickListener **{** ARouter.getInstance().build("/one/one")  
 .withObject("obj", CommonBean("main", 1))  
 .navigation(this, 11, object : NavigationCallback {  
 override fun onLost(postcard: Postcard?) {  
 Log.e("ARouter", "找不到了")  
 }  
 override fun onFound(postcard: Postcard?) {  
 Log.e("ARouter", "找到了")  
 }  
 override fun onInterrupt(postcard: Postcard?) {  
 Log.e("ARouter", "被拦截了")  
 }  
 override fun onArrival(postcard: Postcard?) {  
 Log.e("ARouter", "跳转完了")  
 }  
 })  
**}**

### 降级策略

使用系统自带的StartActivity()启动后就无法插手其中任何环节了，只能交给系统管理，这就导致了在跳转失败的情况下无法降级，而是会直接抛出运营级的异常，甚至导致崩溃，这个给用户的感觉就不是很好。ARouter的降级策略就允许我们在自定义降级服务，在跳转失败的时候可以自行处理，比如可以加载H5页面来处理这种错误情况。

ARouter提供的降级策略主要有两种方式，一种是通过回调的方式；一种是提供服务接口的方式

单独降级-回调的方式

这种方式在跳转失败的时候会回调NavCallback接口的onLost方法

tv\_main.setOnClickListener **{** ARouter.getInstance().build("/one/one")  
 .withObject("obj", CommonBean("main", 1))  
 .navigation(this, 11, object : NavigationCallback {  
 override fun onLost(postcard: Postcard?) {  
 Log.e("ARouter", "找不到了")  
 }  
 ...  
 })  
**}**

全局降级-服务接口的方式：

这种降级策略主要是实现服务接口DegradeService

// 实现DegradeService接口，并加上一个Path内容任意的注解即可  
@Route(path = "/xxx/xxx")  
class DegradeServiceImpl : DegradeService {  
 override fun onLost(context: Context?, postcard: Postcard?) {  
 Log.e("tag", "main的全局降级")  
 }  
 override fun init(context: Context?) {  
 Log.e("tag", "位于 main 中的degrade初始化了")  
 }  
}

同时**使**用

如果同时使用两种方式，单独降级的方式优先于全局降级，也就是说调用完单独降级策略后就不会再调用全局降级

### 发现服务和Fragment

发现服务

这里说到的服务不是Android四大组件中的Service，而是服务端开发的概念，就是将一部分功能和组件封装起来成为接口，以接口的形式对外提供能力，所以在这部分就可以将每个功能作为一个服务，而服务的实现就是具体的业务功能

ARouter发现服务主要有两种方式，ByName和ByType。ByName就是需要传递path路径来进行发现，ByType就是通过服务class来进行查找。

case R.id.navByName:

((HelloService) ARouter.getInstance().build("/service/hello").navigation()).sayHello("mike");

break;

case R.id.navByType:

ARouter.getInstance().navigation(HelloService.class).sayHello("mike");

break;

在Java中接口是可以有多个实现类的，如果有多实现的时候就需要使用ByName。没有多个实现类的大多数情况是使用ByType

发现服务还可以通过下一节的依赖注入

发现Fragment

Fragment获取实例的使用也是很简单，一行代码搞定，和跳转的写法很基本就是一样的。

Fragment fragment = (Fragment) ARouter.getInstance().build("/test/fragment").navigation();

### 通过依赖注入解耦

之前传递参数和对象时通过@Autowired 注解和ARouter.getInstance().inject(this)自动注入，而实现了IProvider的服务类也可以按此方法实现依赖注入

暴露服务

// 写在其他module中找不到，因此只能写在使用该服务的module中

// 实现声明好的服务接口，可通过byType或byName查找到  
@Route(path = "/service/hello", name = "测试服务hello")  
class HelloServiceImpl : HelloService {  
 private var context: Context? = null  
 override fun sayHellow(name: String): String {  
 context?.*let* **{** Toast.makeText(context, "HelloServiceImpl:$name", Toast.*LENGTH\_SHORT*).show()  
 **}** return "HelloServiceImpl:$name"  
 }  
 override fun init(context: Context?) {  
 Log.e("ARouter", "位于 one 中的 HelloServiceImpl 初始化了")  
 this.context = context  
 }  
}  
// 声明继承与OProvider的接口,其他组件通过byType或byName查找到该接口的具体实现类后，来调用具体服务  
interface HelloService : IProvider {  
 fun sayHellow(name: String): String  
}

发现服务

public class Test {  
 @Autowired  
 HelloService helloService;  
 @Autowired(name = "/service/hello")  
 HelloService helloService2;  
 HelloService helloService3;  
 HelloService helloService4;  
 public Test() {  
 ARouter.*getInstance*().inject(this);  
 }  
 public void testService() {  
 // 1. 使用依赖注入的方式发现服务,通过注解标注字段,即可使用，无需主动获取  
 // @Autowired注解中标注name之后，将会使用byName，不设置name属性，会默认使用byType  
 helloService.sayHello("cgy");  
 helloService2.sayHello("cgy");  
 // 2. 使用依赖查找的方式发现服务，主动去发现服务并使用，下面两种方式分别是byName和byType  
 helloService4 = (HelloService) ARouter.*getInstance*().build("/service/hello").navigation();  
 helloService3 = ARouter.*getInstance*().navigation(HelloService.class);  
 helloService3.sayHello("cgy");  
 helloService4.sayHello("cgy");  
 }  
}

## 其他api

// 指定Flag

ARouter.getInstance()

.build("/home/main")

.withFlags();

.navigation();

// 觉得接口不够多，可以直接拿出Bundle赋值

ARouter.getInstance()

.build("/home/main")

.getExtra();

// 转场动画(常规方式)

ARouter.getInstance()

.build("/test/activity2")

.withTransition(R.anim.slide\_in\_bottom, R.anim.slide\_out\_bottom)

.navigation(this);

// 转场动画(API16+)

// ps. makeSceneTransitionAnimation 使用共享元素的时候，需要在navigation方法中传入当前Activity

ARouter.getInstance()

.build("/test/activity2")

.withOptionsCompat(ActivityOptionsCompat.makeScaleUpAnimation(v, v.getWidth() / 2, v.getHeight() / 2, 0, 0))

.navigation();

// 使用绿色通道(跳过所有的拦截器)

ARouter.getInstance().build("/home/main").greenChannel().navigation();

// 使用自己的日志工具打印日志

ARouter.setLogger();

// 使用自己提供的线程池

ARouter.setExecutor();

// 获取原始的URI（通过uri跳转的才有，通过path跳转的没有）

String uriStr = getIntent().getStringExtra(ARouter.RAW\_URI);

## 重写跳转URL(!!疑问!!)

// 实现PathReplaceService接口，并加上一个Path内容任意的注解即可

@Route(path = "/xxx/xxx") // 必须标明注解  
class PathReplaceServiceImpl : PathReplaceService {  
 override fun forString(path: String?): String {  
 return "/one/one"  
 }  
 override fun forUri(uri: Uri?): Uri {  
 return Uri.fromFile(File("arouter://m.aliyun.com/one/one"))  
 }  
  
 override fun init(context: Context?) {  
 Log.e("ARouter", "位于 main 中的 PathReplaceServiceImpl 初始化了")  
 }  
}

## 生成路由文档

要生成哪个module的路由文档，就在哪个module的gradle中添加

// 生成的文档路径 : build/generated/source/apt/(debug or release)/com/alibaba/android/arouter/docs/arouter-map-of-${moduleName}.json

android {

defaultConfig {

...

javaCompileOptions {

annotationProcessorOptions {

arguments = [AROUTER\_MODULE\_NAME: project.getName(), AROUTER\_GENERATE\_DOC: "enable"]

}

}

}

}

## 其他

路由中的分组概念

* SDK中针对所有的路径(/test/1 /test/2)进行分组，分组只有在分组中的某一个路径第一次被访问的时候，该分组才会被初始化
* 可以通过 @Route 注解主动指定分组，否则使用路径中第一段字符串(/\*/)作为分组@Route(path = "/test/1", group = "app")
* 注意：一旦主动指定分组之后，应用内路由需要使用 ARouter.getInstance().build(path, group) 进行跳转，手动指定分组，否则无法找到

拦截器和服务的异同

* 拦截器和服务所需要实现的接口不同，但是结构类似，都存在 init(Context context) 方法，但是两者的调用时机不同
* 拦截器因为其特殊性，会被任何一次路由所触发，拦截器会在ARouter初始化的时候异步初始化，如果第一次路由的时候拦截器还没有初始化结束，路由会等待，直到初始化完成。
* 服务没有该限制，某一服务可能在App整个生命周期中都不会用到，所以服务只有被调用的时候才会触发初始化操作

# ARouter解析

## 名词解释

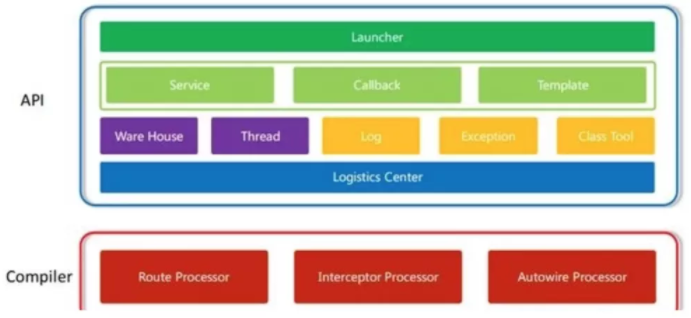
### APT技术

APT，就是Annotation Processing Tool 的简称，就是可以在代码编译期间对注解进行处理，并且生成Java文件，减少手动的代码输入。这里我们就不多做介绍，这个不太清楚的可以参考我之前的分享Android模块开发之APT技术。

### SPI技术

Java提供的SPI全名就是Service Provider Interface,其实就是为某个接口寻找服务的机制，有点类似IOC的思想，将装配的控制权移交给ServiceLoader。SPI在平时我们用到的会比较少，但是在Android模块开发中就会比较有用，不同的模块可以基于接口编程，每个模块有不同的实现service provider,然后通过SPI机制自动注册到一个配置文件中，就可以实现在程序运行时扫描加载同一接口的不同service provider。不太清楚的可以参考之前的分析Android模块开发之SPI。

## 框架架构



* 最基础的就是Compiler这个SDK，主要是用来在编译期间处理注解Router/Interceptor/Autowire三个注解，在编译期间自动注册注解标注的类，成员变量等。
* API的SDK是用户在运行期使用。Launcher这一层只有ARouter和\_ ARouter, ARouter就是我们需要打交道的接口，这里使用了代理模式，实际ARouter是通过\_ ARouter进行工作。
* 第二层也是绿色的，用户也是可以调用到，这里主要是提供接口，我们可以扩展服务，比如拦截器服务，成员变量装配服务等，Template主要是提供模版接口，这个SDK会在编译期生成一些映射文件，而这些映射文件会按照Template组件中提供的模板来生成，这样按照一定的规则和约束生成映射文件也方便Route在运行的时候进行读取。
* Ware House类似一个仓库，主要存储ARouter在运行期间加载的一些配置文件以及映射关系。
* Thread则是提供了线程池，因为存在多个拦截器的时候以及跳转过程中都是需要异步执行的。
* 再下一层就是Logistics Center，从名字上翻译就是物流中心，整个SDK的流转以及内部调用最终都会下沉到这一层。

## 三个主要注解

@Target({ElementType.*TYPE*})  
@Retention(RetentionPolicy.*CLASS*)  
public @interface Route {  
 */\*\*  
 \* Path of route  
 \*/* String path();  
 ……  
}  
  
@Target({ElementType.*TYPE*})  
@Retention(RetentionPolicy.*CLASS*)  
public @interface Interceptor {  
 */\*\*  
 \* The priority of interceptor, ARouter will be excute them follow the priority.  
 \*/* int priority();  
 */\*\*  
 \* The name of interceptor, may be used to generate javadoc.  
 \*/* String name() default "Default";  
}  
  
@Target({ElementType.*FIELD*})  
@Retention(RetentionPolicy.*CLASS*)  
public @interface Autowired {  
 // Mark param's name or service name.  
 String name() default "";  
 // If required, app will be crash when value is null.  
 // Primitive type wont be check!  
 boolean required() default false;  
 // Description of the field  
 String desc() default "No desc.";  
}

ARouter会在编译的时候将Router注解标注的activity扫描出来，并且按照定义好的模版生成装配关系的代码。这个其实就是APT技术。简单说就是要自定义注解的处理器，继承自AbstractProcessor，需要自己实现process方法来处理注解，然后需要用到SPI技术来注册我们自己定义的注解处理器，之后JVM就能在编译期间通过serviceLoader来找到我们的注解处理器balabala。

## 页面注册源码分析

@Route(path = "/test/activity2")public class Test2Activity extends AppCompatActivity{}

ARouter.getInstance().build("/test/activity2").withString("key1", "value1").navigation();

ARouter的process方法比较长，我们就看下生成的文件放在哪里，包名就是public static final String PACKAGE\_OF\_GENERATE\_FILE = "com.alibaba.android.arouter.routes";

那么文件名呢？跳转Test2Activity的path是@Route(path = "/test/activity2")其中test就是group，所以我们文件名就是groupFileName = ARouter$$Group$$test

public static final String SEPARATOR = "$$";

public static final String PROJECT = "ARouter";

public static final String NAME\_OF\_GROUP = PROJECT + SEPARATOR + "Group" + SEPARATOR;

public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv){

String groupFileName = NAME\_OF\_GROUP + groupName;

JavaFile.builder(PACKAGE\_OF\_GENERATE\_FILE,

TypeSpec.classBuilder(groupFileName)

.addJavadoc(WARNING\_TIPS)

.addSuperinterface(ClassName.get(type\_IRouteGroup))

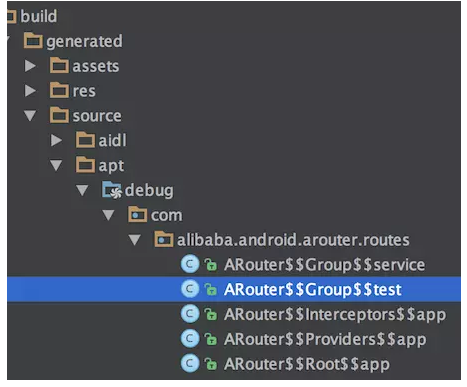
.addModifiers(PUBLIC)

.addMethod(loadIntoMethodOfGroupBuilder.build())

.build()

).build().writeTo(mFiler);

}



那么我们来看看ARouter自动生成的activity装配关系是怎么样的。

public class ARouter$$Group$$test implements IRouteGroup {

@Override

public void loadInto(Map<String, RouteMeta> atlas) {

atlas.put("/test/activity2", RouteMeta.build(RouteType.ACTIVITY, Test2Activity.class, "/test/activity2",

"test", new java.util.HashMap<String, Integer>(){{put("key1", 8); }}, -1, -2147483648));

}

}

有一个需要注意的地方就是put("key1", 8)，还记得我们上面参数跳转应该是withString("key1", "value1"), 为什么这里是8？其实在编译的时候做了处理，8代表类型String，页面自动注册的源码基本就是上面这些内容，我们最后引用一张官方的图片。



* 首先通过注解处理器扫出被标注的类文件；然后按照不同种类的源文件进行分类，不仅仅提供了跳转功能，它也能够实现模块之间的解耦，其实ARouter中的所有组件都是自动注册的
* 在按照不同种类的源文件进行分类完成之后，就能够按照固定的命名格式生成映射文件，这部分完成之后就意味着编译期的部分已经结束了；
* 而最后一步的初始化其实是发生在运行期的，在运行期只需要通过固定的包名来加载映射文件就可以了，这就是页面自动注册的整个流程。