目录

[一、四大组件 6](#_Toc118893044)

[1.1 Activity 6](#_Toc118893045)

[1.1.1 生命周期 6](#_Toc118893046)

[1.1.2 启动模式 7](#_Toc118893047)

[1.1.3 启动过程 8](#_Toc118893048)

[1.2 BroadcastReceiver 17](#_Toc118893049)

[1.2.1 使用场景 17](#_Toc118893050)

[1.2.2 种类 17](#_Toc118893051)

[1.2.3 广播接收器 17](#_Toc118893052)

[1.3 ContentProvider 17](#_Toc118893053)

[1.3.1 基本使用 17](#_Toc118893054)

[1.3.2 创建应用的内容提供者 18](#_Toc118893055)

[1.3.3 工作原理 19](#_Toc118893056)

[1.4 Service 19](#_Toc118893057)

[1.4.1 与Thread的区别 19](#_Toc118893058)

[1.4.2 启动方式 19](#_Toc118893059)

[1.4.3 生命周期 20](#_Toc118893060)

[1.4.4 Service保活 20](#_Toc118893061)

[1.4.5 IntentService 20](#_Toc118893062)

[1.4.6 独立进程Servic 21](#_Toc118893063)

[二、Fragment 21](#_Toc118893064)

[2.1 生命周期 21](#_Toc118893065)

[2.1.1 简单情形 22](#_Toc118893066)

[2.1.2 结合Activity 22](#_Toc118893067)

[2.1.3结合ViewPager 23](#_Toc118893068)

[2.2 懒加载 25](#_Toc118893069)

[2.3 使用方法 25](#_Toc118893070)

[2.3.1 单独使用 25](#_Toc118893071)

[2.3.2 结合ViewPager 26](#_Toc118893072)

[三、Handler 26](#_Toc118893073)

[3.1 实现原理 26](#_Toc118893074)

[3.2 内存泄漏 26](#_Toc118893075)

[3.2.1 原因 26](#_Toc118893076)

[3.2.2 解决方法 26](#_Toc118893077)

[3.3 Handler框架 27](#_Toc118893078)

[3.3.1 AsyncTask 27](#_Toc118893079)

[3.3.2 HandlerThread 28](#_Toc118893080)

[3.3.3 IntentService 28](#_Toc118893081)

[四、数据持久化 28](#_Toc118893082)

[4.1 Android中数据持久化的方式与使用场景 28](#_Toc118893083)

[4.1.1 Shared Preferences 28](#_Toc118893084)

[4.1.2 Internal Storage 30](#_Toc118893085)

[4.1.3 External Storage 30](#_Toc118893086)

[4.1.4 SQLite数据库 31](#_Toc118893087)

[4.1.5 三级缓存 31](#_Toc118893088)

[4.2 MMKV 31](#_Toc118893089)

[五、布局 31](#_Toc118893090)

[5.1 常用布局及特点 31](#_Toc118893091)

[5.1.1 相对布局RelativeLayout 31](#_Toc118893092)

[5.1.2 线性布局LinearLayout 32](#_Toc118893093)

[5.1.3约束布局ConstraintLayout 32](#_Toc118893094)

[5.1.4 表格布局TableLayout 32](#_Toc118893095)

[5.1.5 帧布局FrameLayout 32](#_Toc118893096)

[5.1.6 绝对布局Absolute Layout 32](#_Toc118893097)

[5.2 性能对比 32](#_Toc118893098)

[5.2.1 RelativeLayout和LinearLayout的onMeasure()方法 32](#_Toc118893099)

[5.2.2 RelativeLayout另一个性能问题 33](#_Toc118893100)

[5.3 布局文件到控件对象的过程 33](#_Toc118893101)

[5.3.1 33](#_Toc118893102)

[六、IPC 33](#_Toc118893103)

[6.1 多进程概述 33](#_Toc118893104)

[6.1.1 多进程的定义 33](#_Toc118893105)

[6.1.2 多进程的等级 34](#_Toc118893106)

[6.1.3 多进程的创建 35](#_Toc118893107)

[6.2 多进程间通信IPC 35](#_Toc118893108)

[6.2.1 Bundle/Intent 35](#_Toc118893109)

[6.2.2 文件共享 35](#_Toc118893110)

[6.2.3 Messenger 35](#_Toc118893111)

[6.2.4 AIDL 35](#_Toc118893112)

[6.2.5 ContentProvider 35](#_Toc118893113)

[6.2.6 Socket 35](#_Toc118893114)

[6.3 Binder 35](#_Toc118893115)

[6.3.1 Linux和Android的IPC机制种类 36](#_Toc118893116)

[6.3.2 Linux的IPC通信原理 36](#_Toc118893117)

[6.3.3 Binder的通信原理 39](#_Toc118893118)

[6.3.4 Binder的特点 40](#_Toc118893119)

[6.3.5 binder源码解析 41](#_Toc118893120)

[七、图片编程 42](#_Toc118893121)

[7.1 Bitmap基础 42](#_Toc118893122)

[7.2 Bitmap导致的OOM 42](#_Toc118893123)

[7.3 图片压缩 42](#_Toc118893124)

[7.4 图片缓存 42](#_Toc118893125)

[八、View 42](#_Toc118893126)

[8.1 View基础知识 42](#_Toc118893127)

[8.1.1 View的位置参数 42](#_Toc118893128)

[8.1.2 MotionEvent 42](#_Toc118893129)

[8.1.3 TouchSlop 43](#_Toc118893130)

[8.1.4 VelocityTracker 44](#_Toc118893131)

[8.1.5 GestureDetector 44](#_Toc118893132)

[8.1.6 Scroller 44](#_Toc118893133)

[8.2 View的滑动 44](#_Toc118893134)

[8.2.1 scrollTo/scrollBy 44](#_Toc118893135)

[8.2.2 使用动画 44](#_Toc118893136)

[8.2.3 改变布局参数 44](#_Toc118893137)

[8.3 弹性滑动 45](#_Toc118893138)

[8.3.1 使用Scroller 45](#_Toc118893139)

[8.3.2 使用动画 45](#_Toc118893140)

[8.3.3 使用延时策略 45](#_Toc118893141)

[8.4 事件分发机制 45](#_Toc118893142)

[8.4.1事件分发的本质 45](#_Toc118893143)

[8.4.2 三个重要方法 45](#_Toc118893144)

[8.4.3 事件分发流程&源码分析 48](#_Toc118893145)

[8.5 滑动冲突 51](#_Toc118893146)

[8.5.1 常见的滑动冲突场景 51](#_Toc118893147)

[8.5.2 滑动冲突的解决方式 52](#_Toc118893148)

[8.6 View的工作原理 52](#_Toc118893149)

[8.6.1 ViewRoot和DecorView 52](#_Toc118893150)

[8.6.2 MeasureSpec 52](#_Toc118893151)

[8.6.3 View的工作流程 52](#_Toc118893152)

[8.7 自定义View 53](#_Toc118893153)

[8.7.1 自定义View的分类 53](#_Toc118893154)

[8.7.2 自定义View须知 53](#_Toc118893155)

[8.8 ListView和RecyclerView 53](#_Toc118893156)

[8.8.1 ListView和RecyclerView的区别 53](#_Toc118893157)

[8.8.2 RecyclerView的使用 54](#_Toc118893158)

[8.8.3 RecyclerView的缓存机制 54](#_Toc118893159)

[8.8.4上拉刷新 & 下拉加载 55](#_Toc118893160)

[8.8.5 ListView和RecyclerView的优化 55](#_Toc118893161)

[九、动画 56](#_Toc118893162)

[9.1 56](#_Toc118893163)

[十、Android开源库 56](#_Toc118893164)

[10.1 Rxjava 56](#_Toc118893165)

[10.1.1 设计思想 56](#_Toc118893166)

[10.1.2 使用方法 56](#_Toc118893167)

[10.1.3 线程切换原理 57](#_Toc118893168)

[10.2 OKHttp 57](#_Toc118893169)

[10.2.1 OKHttp的基本使用 57](#_Toc118893170)

[10.2.2 请求的整体流程 57](#_Toc118893171)

[10.2.3 分发器Dispatcher 58](#_Toc118893172)

[10.2.3 拦截器Interceptor 58](#_Toc118893173)

[10.2.4 缓存机制 59](#_Toc118893174)

[10.2.4 ConnectionPool 59](#_Toc118893175)

[10.3 Retrofit 60](#_Toc118893176)

[10.3.1 Retrofit要解决的OKHttp的问题 60](#_Toc118893177)

[10.3.2 Retrofit的基本使用 60](#_Toc118893178)

[10.3.3 Retrofit中的设计模式 60](#_Toc118893179)

[10.4 Glide 61](#_Toc118893180)

[10.4.1 61](#_Toc118893181)

[10.4.2 61](#_Toc118893182)

[10.4.3 61](#_Toc118893183)

[10.4.4 61](#_Toc118893184)

[10.5 GreenDao 62](#_Toc118893185)

[10.5.1 配置 62](#_Toc118893186)

[10.5.2 使用方法 62](#_Toc118893187)

[10.6 Dagger2 62](#_Toc118893188)

[10.6.1 使用方法 63](#_Toc118893189)

[10.6.2 实现原理 66](#_Toc118893190)

[10.7 ButterKnife 66](#_Toc118893191)

[10.7.1 配置 66](#_Toc118893192)

[10.7.2基本使用 66](#_Toc118893193)

[十一、性能优化 68](#_Toc118893194)

[11.1 68](#_Toc118893195)

[十二、JNI 68](#_Toc118893196)

[12.1 68](#_Toc118893197)

[十三、音视频 68](#_Toc118893198)

[13.1 68](#_Toc118893199)

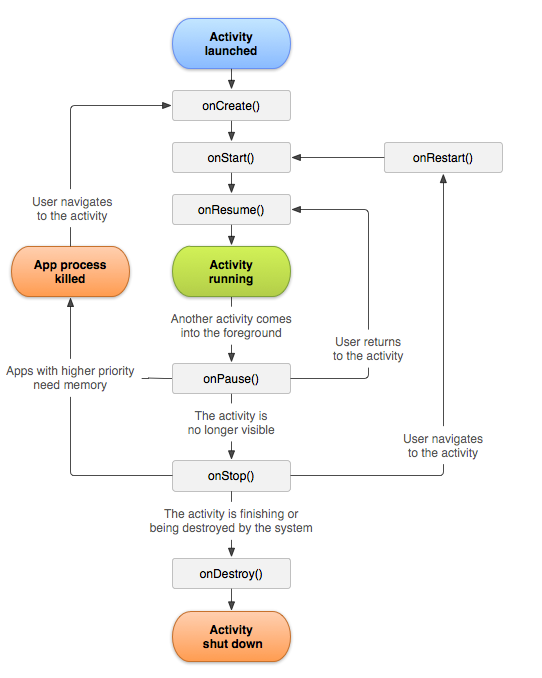
# 一、四大组件

## 1.1 Activity

### 1.1.1 生命周期

Activity实际上只是一个与用户交互的接口而已。

1. 正常情况



（2）异常情况

系统会调用onSaveInstanceState来保存当前Activity状态。调用onSaveInstanceState的时机总会发生在onStop之前，。当Activity被重新创建后，系统会调用onRestoreInstanceState,并且把Actiivty销毁时onSaveInstanceState方法所保存的Bundle对象作为参数传递给onRestoreInstanceState和onCreate方法。所以可以通过onRestoreInstanceState和onCreate方法来判断Actiivty是否被重建了，如果被重建了，那么我们就可以取出之前保存的数据并恢复，从时序上来看，onRestoreInstanceState的调用时机发生在onStart之后。

同时，在onSaveInstanceState和onRestoreInstanceState方法中，系统自动为我们做了一定的恢复工作。当Activity在异常情况下需要重新创建时，系统会默认为我们保存当前Activity的视图结构。当Activity在异常情况下需要重新创建时，系统会默认为我们保存当前Activity的视图结构，并且在Activity重启后为我们恢复这些数据，比如：文本框中用户输入的数据,ListView滚动的位置等，这些View相关的状态系统都能够默认为我们恢复。具体针对某一个特定的View系统能为我们恢复哪些数据，我们可以查看View的源码。和Activity一样，每个View都有onSaveInstanceState和onRestoreInstanceState这两个方法，看一下它们的具体实现，就能知道系统能够自动为每个View恢复哪些数据。

关于保存和恢复View层次结构，系统的工作流程是这样的：

首先Activity被意外终止时，Activity会调用onSaveInstanceState去保存数据，然后Activity会委托Window去保存数据，接着Window在委托它上面的顶级容器去保存数据。顶级容器是一个ViewGroup，一般来说它很可能是DecorView。最后顶层容器再去一一通知它的子元素来保存数据，这样整个数据保存过程就完成了。

（3）特殊情况

横竖屏切换：不设置Activity的android:configChanges，或设置Activity的android:configChanges="orientation"，或设置Activity的android:configChanges="orientation|keyboardHidden"，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行一次；配置 android:configChanges="orientation|keyboardHidden|screenSize"，不会销毁 activity，且只调用 onConfigurationChanged方法。

### 1.1.2 启动模式

启动一个Activity后，这个Activity实例就会被放入任务栈中，当点击返回键的时候，位于任务栈顶层的Activity就会被清理出去，当任务栈中不存在任何Activity实例后，系统就回去回收这个任务栈，也就是程序退出了。启动模式是为了解决频繁启动Activity而生成多个实例的问题。

（1）standard

标准模式，这也是系统的默认模式。每次启动一个Activity都会重新创建一个新的实例，不管这个实例是否存在。被创建的实例的生命周期符合典型情况下的Activity的生命周期。

（2）singleTop

栈顶复用模式，在这种模式下，如果新的Activity已经位于任务栈的栈顶，那么此Activity不会被重新创建，同时它的onNewIntent方法被回调，通过此方法的参数我们可以取出当前请求的信息。

（3）singleTask

栈内复用模式，这是一种单例实例模式，在这种模式下，只要Activity在一个栈中存在，那么多次启动此Activity都不会重新创建实例，和singleTop一样，系统也会回调其onNewIntent。同时由于singleTask默认具有clearTop的效果，启动会导致栈内所有在此上面的Activity全部出栈。

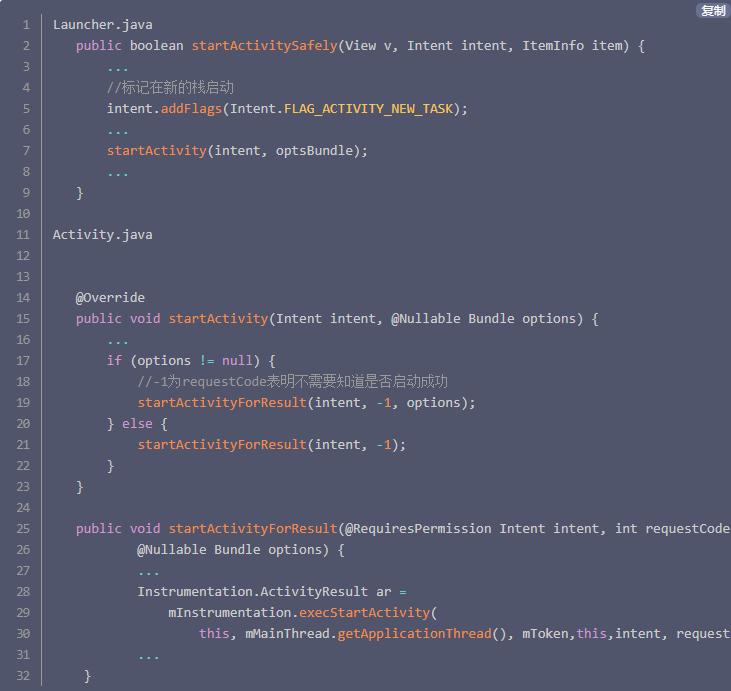
（4）singleInstance

这是一种加强的singleTask模式，它除了具有singleTask模式所有的特性外，还加强了一点，那就是具有此种模式的Activity只能单独位于一个任务栈中。

### 1.1.3 启动过程

1. Launcher向AMS发送启动Activity

在用户点击应用图标时，Launcher这个Activity会调用startActivitySafely方法，最后调用到Activity.startActivity方法，方法调用流程如下：



每个Activity都持有Instrumentation对象，通过它的execStartActivity方法来继续完成启动Activity的流程，要注意的是这个方法中传入了mMainThread.getApplicationThread()，它获取到的是ActivityThread的内部类ApplicationThread实例，这是一个Binder对象，之后AMS通过此对象与App通信。



这一步Launcher开始向AMS通信，由于在不同的进程所以需要通过Binder来通信，IActivityTaskManager是一个代理AMS端Binder的对象，之后AMS开始startActivity。

到这里Launcher向AMS请求启动一个Activity的流程就结束了。

1. AMS启动Activity并通知Launcher进入Paused状态

接下来的流程我们切换到AMS所在的进程继续分析，上一步通过代理调用到AMS的startActivity方法，接下来的调用如下：



上面几步主要是做权限检查，接着进入execute()方法。



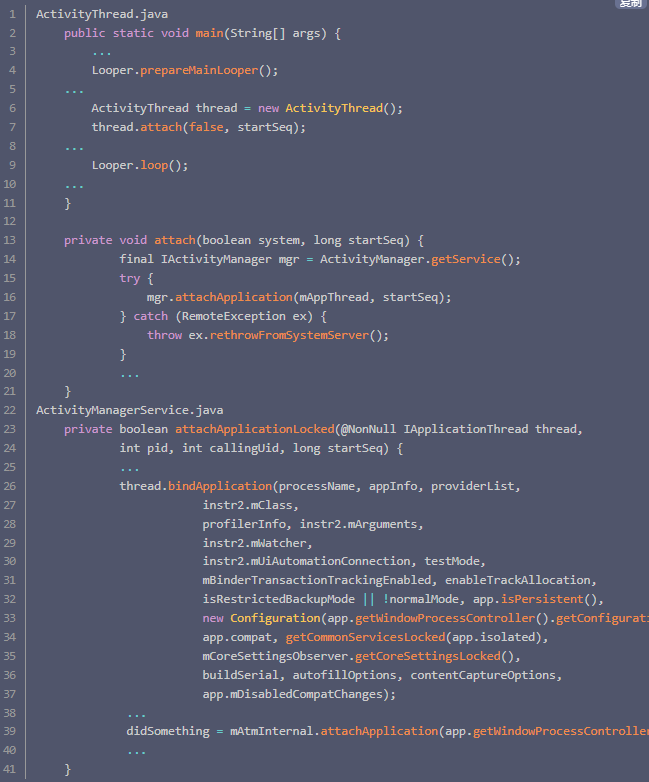
startPausingLocked方法主要是通知Launcher进入Paused状态，在它进入这个状态后，再通过ActivityStackSupervisor.startSpecificActivity方法判断要启动的Activity所属进程的状态做出不同响应。



截止到这里完成了Launcher和AMS的通信，以及AMS和Zygote进程的通信，接下来我们要创建要启动的App的线程，即ActivityThread。

1. 新的进程启动，ActivityThread的main函数入口

上一部分的最后Zygote启动新的进程时会标记ActivityThread.main函数，在Zygote创建好新进程后通过反射调用此方法，之后的流程处于新的App进程中，当然还是会与AMS进行一些通信的。



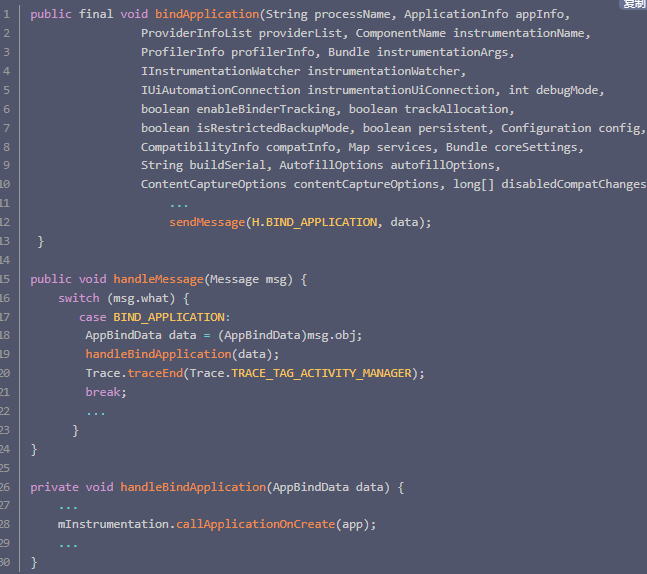
这里主要是创建了Looper和ActivityThread对象，然后将当前应用ApplicationThread注册到AMS中，ApplicationThread是ActivityThread的内部类实现了IApplicationThread.Stub用此对象可跨进程通信。

上面的代码逻辑分两步：

第一步，在AMS绑定ApplicationThread时，发送了一个H.BIND\_APPLICATION的Message，在Handler中处理该消息时调用了Application的onCreate方法。

第二步，在mAtmInternal的attachApplication层层调用到ActivityStackSupervisor.realStartActivityLocked方法，进入准备Activity事务阶段。

整体如下：



到这里为止，新的App线程已经启动并且绑定了Application。

1. 创建Activity



ClientTransaction管理了Activity的启动信息，LaunchActivityItem继承ActivityLifecycleItem，具象化了Activity的生命周期，并可ClientLifecycleManager由执行，接下来scheduleTransaction方法中发送了EXECUTE\_TRANSACTION的消息给ActivityThread的H类处理，然后再执行TransactionExecutor.execute()，最后执行handleLaunchActivity方法，如下:



接下来由ActivityThread来处理后续操作。



performLaunchActivity方法中主要做了以下几件事：

* 1. 创建要启动activity的上下文环境；
  2. 通过Instrumentation的newActivity方法，以反射形式创建activity实例；
  3. 如果Application不存在的话会创建Application并调用Application的onCreate方法；
  4. 初始化Activity，创建Window对象（PhoneWindow）并实现Activity和Window相关联；
  5. 通过Instrumentation调用Activity的onCreate方法。

1. 总结Activity的整体启动流程
   1. 点击图标，Launcher向AMS请求启动该App；
   2. AMS反馈收到启动请求,并告知Launcher进入pause状态；
   3. Launcher进入Paused状态并告知AMS；
   4. AMS检测新的App是否已启动，否则通知Zygote创建新的进程并调用ActivityThread.main方法；
   5. 应用进程启动ActivityThread；
   6. ActivityThread中H类处理需要启动Activity的请求消息。

## 1.2 BroadcastReceiver

### 1.2.1 使用场景

它是一种广泛运用在应用程序之间传输信息的机制。

（1）同一app内有多个进程的不同组件之间的消息通信。

（2）不同app之间的组件之间消息的通信。

### 1.2.2 种类

1、无序广播 context.sendBroadcast(Intent)

不可被拦截，当然发送的数据，接收者是不能进行修改的。

2、有序广播 context.sendOrderBroadcast(Intent)

可被拦截，而且接收者是可以修改其中要发送的数据，修改和添加都是可以的，这就意味着优先接收者对数据修改之后，下一个接收者接受的数据是上一个接收者已经修改了的。

3、本地广播 localBroadcastManager.sendBroadcast(Intent)

只在app内传播，保证安全性。

### 1.2.3 广播接收器

1、静态注册

新建一个类，让它继承自BroadcastReceiver,并重写父类的onReceive()方法，接着在AndroidManifest.xml文件中注册。Android8.0之后intent要添加setComponent()，发送显示intent。

2、动态注册

IntentFilter注册Action，用registerReceiver()绑定receiver和intentFilter。必须在onDestory()中用unregisterReceiver()取消注册。

3、优缺点

动态注册的广播接收器可以自由地控制注册与注销，在灵活性方面有很大优势，但是它也存在着一个缺点，即必须要在程序启动之后才能接收到广播，因为注册的逻辑是写在onCreate()方法中的。而静态注册的广播接收器则可以在程序未启动的情况下就能接收到广播。

## 1.3 ContentProvider

### 1.3.1 基本使用

ContentProvider用于在不同的应用程序之间实现数据共享的功能。

Cursor cursor = getContentResolver().query(…, …, …)

（1）uri，指定查询某一个程序下的某一张表；

（2）projection，指定查询的列名；

（3）selection，指定查询条件，相当于sql语句中where后面的条件；

（4）selectionArgs，给selection中的占位符提供具体的值；

（5）orderBy，指定查询结果排序方式；

（6）cancellationSignal，取消正在进行操作的信号量。

在api30中，需要在使用provider的应用程序的AndroidManifest中配置好如下queries才能使用：

<queries>

<provider android:authroities="com.example.databasestore.provider"/>

</queries>

### 1.3.2 创建应用的内容提供者

继承ContentProvider类，实现以下六个方法：

（1）onCreate()

  初始化内容提供器的时候调用。通常会在这里完成对数据库的创建和升级等操作。返回true表示内容提供器初始化成功，返回false则表示失败。注意，只有当存在ContentResolver尝试访问我们的程序中的数据时，内容提供器才会被初始化。

  （2）query()

  从内容提供器中查询数据。使用uri参数来确定查询的哪张表，projection参数用于确定查询的哪一列，selection和selectionArgs参数用于约束查询哪些行，sortOrder参数用于对结果进行排序，查询的结果存放在Cursor对象中返回。

  （3）insert()

  向内容提供器中添加一条数据。使用uri参数来确定要添加的表，待添加的数据保存在values参数中。添加完成后，返回一个用于表示这条新纪录的URI。

  （4）update()

  更新内容提供器中已有的数据。使用uri参数来确定更新哪一张表中的数据，新数据保存着values参数当中，selection和selectionArgs参数用于约束更新哪些行，受影响的行数将作为返回值返回。

  （5）delete()

  从内容提供器中删除数据。使用uri参数来确定删除哪一张表中的数据，selection和selectionArgs参数用于约束删除哪些行，被删除的行数将作为返回值返回。

  （6）getType()

  根据传入的内容URI来返回相应的MIME类型。一个内容URI所对应的MIME字符串主要由3部分组成，Android对这3个部分做了如下格式规定：

* 必须以vnd开头；
* 如果内容URI以路径结尾，则后接android.cursor.dir/,如果内容URI以id结尾，则后接android.cursor.item/；
* 最后接vnd.< authority >.< path >

借助UriMatcher这个类匹配内容URI。UriMatcher中提供了一个addURI()方法，分别把authority，path和一个自定义代码传进去，这个自定义代码其实就是一个final的int类型的具值。当调用UriMatcher的match()方法时，将一个Uri对象传入，返回值是某个能够匹配这个Uri对象所对应的自定义代码，利用这个代码判断调用方期望访问哪张表中的数据。

### 1.3.3 工作原理

ContentProvider的onCreate的调用时机介于Application的attachBaseContext和onCreate之间，Provider的onCreate优先于Application的onCreate执行。

进程通过AMS获取ContentProvider的IContenProvider这个binder对象，再通过binder机制实现IPC。

## 1.4 Service

### 1.4.1 与Thread的区别

Service和Thread没有任何关系。Service运行在主线程，完全不依赖UI/Activity，只要进程还在就可继续运行。所有Activity都可与Service关联，获得Binder实例，操作其中的方法。若要处理耗时操作，可在Service中创建Thread子线程执行。

### 1.4.2 启动方式

（1）startService()

a. 定义一个类继承Service。

   b. 在AndroidManifest.xml文件中配置该Service。

   c. 使用Context的startService(Intent)方法启动该Service。

   d. 不再使用该Service时，调用Context的stopService(Intent)方法停止该Service。

android 5.0以上不支持隐式启动服务。

（2）bindService（Activity与Service绑定）

a. 创建BinderService服务端，继承自Service并在类中创建一个实现IBinder接口的实现实例对象并提供公共方法给客户端调用。

   b. 从onBind()回调方法返回此Binder实例。

   c. 在客户端中，从onServiceConnected回调方法接收Binder,并使用提供的方法调用绑定服务。

### 1.4.3 生命周期

### 1.4.4 Service保活

（1）提升service进程优先级，将Service设置为前台进程；

（2）onDestroy方法里重启service；

（3）保证Service在开机后自动启动；

（4）保证息屏后不被释放资源杀死（WakeLock的使用）

### 1.4.5 IntentService

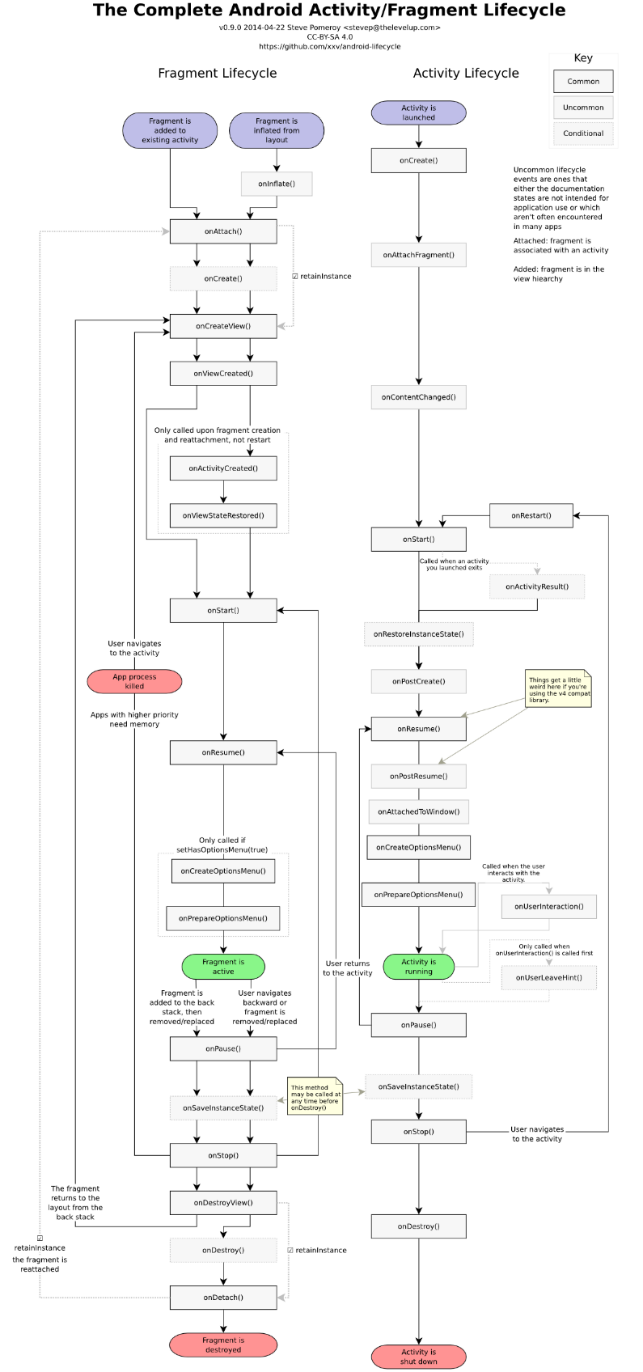
IntentService是继承处理异步请求的一个类，在IntentService内有一个工作线程来处理耗时操作，启动IntentServiced的方式和启动传统的Service一样，同时，当任务执行完成后，IntentService会自动停止，而不需要我们手动去控制或stopSelf()。另外，可以启动IntentService多次，而每一个耗时操作会以工作队列的方式在IntentService的onHandlerIntent回调方法中执行，并且，每次只执行一个工作线程，执行完第一个在执行第二个。

  它本质是一种特殊的Service,继承自Service并且本身就是一个抽象类。内部是由HandlerThread和Handler实现异步操作。它的优先级高于Service。

### 1.4.6 独立进程Servic

# 二、Fragment

## 2.1 生命周期



### 2.1.1 简单情形

（1）界面打开

onCreate()→onCreateView()→onViewCreated()→onStart()→onResume()

（2）按下主屏幕键/锁屏

onPause()→onStop()

（3）重新打开

onStart()→onResume()

（4）按下后退键

onPause()→onStop()→onDestroyView()→onDestroy()→onDetach()

### 2.1.2 结合Activity

（1）界面打开

Fragment onAttach()

Fragment onCreate()

Fragment onCreateView()

Fragment onViewCreated()

Activity onCreate()

Activity onStart()

Fragment onStart()

Activity onResume()

Fragment onResume()

（2）按下主屏幕键/锁屏

Fragment onPause()

Activity onPause()

Fragment onStop()

Activity onStop()

（3）再次打开

Activity onRestart()

Activity onStart()

Fragment onStart()

Activity onResume()

Fragment onResume()

（4）按下后退键

Fragment onPause()

Activity onPause()

Fragment onStop()

Activity onStop()

Fragment onDestroyView()

Fragment onDestroy()

Fragment onDetach()

Activity onDestroy()

### 2.1.3结合ViewPager

以四页的ViewPage为例，OffscreenPageLimitr设置为1（注意FragmentPagerAdapter和FragmentStatePagerAdapter的onDetach的区别）。

（1）打开界面

MainActivity\_LifeCricle: onCreate

MainActivity\_LifeCricle: onStart

MainActivity\_LifeCricle: onResume

FragmentFunOne: onAttach

FragmentFunOne: onCreate

FragmentFunOne: onCreateView

FragmentFunOne: onViewCreated

FragmentFunOne: onStart

FragmentFunTwo: onAttach

FragmentFunTwo: onCreate

FragmentFunTwo: onCreateView

FragmentFunTwo: onViewCreated

FragmentFunTwo: onStart

FragmentFunOne: onResume

（2）左划一页

FragmentFunThree: onAttach

FragmentFunThree: onCreate

FragmentFunThree: onCreateView

FragmentFunThree: onViewCreated

FragmentFunThree: onStart

FragmentFunOne: onPause

FragmentFunTwo: onResume

（3）按下主屏幕键/锁屏

FragmentFunTwo: onPause

MainActivity\_LifeCricle: onPause

FragmentFunOne: onStop

FragmentFunTwo: onStop

FragmentFunThree: onStop

MainActivity\_LifeCricle: onStop

（4）再次打开

MainActivity\_LifeCricle: onRestart

FragmentFunOne: onStart

FragmentFunTwo: onStart

FragmentFunThree: onStart

MainActivity\_LifeCricle: onStart

MainActivity\_LifeCricle: onResume

FragmentFunTwo: onResume

（5）再左划一页

FragmentFunOne: onStop

FragmentFunFour: onAttach

FragmentFunFour: onCreate

FragmentFunFour: onCreateView

FragmentFunFour: onViewCreated

FragmentFunFour: onStart

FragmentFunTwo: onPause

FragmentFunThree: onResume

FragmentFunOne: onDestroyView

（6）按下后退键

FragmentFunThree: onPause

MainActivity\_LifeCricle: onPause

FragmentFunTwo: onStop

FragmentFunThree: onStop

FragmentFunFour: onStop

MainActivity\_LifeCricle: onStop

FragmentFunOne: onDestroy

FragmentFunOne: onDetach

FragmentFunTwo: onDestroyView

FragmentFunTwo: onDestroy

FragmentFunTwo: onDetach

FragmentFunThree: onDestroyView

FragmentFunThree: onDestroy

FragmentFunThree: onDetach

FragmentFunFour: onDestroyView

FragmentFunFour: onDestroy

FragmentFunFour: onDetach

MainActivity\_LifeCricle: onDestroy

## 2.2 懒加载

Support库中的ViewPager使用setUserVisibleHint()控制fragment的显示。利用重写setUserVisibleHint()的方式，将加载数据放在里面实现懒加载。

AndroidX中FragmentPagerAdapter增加了BEHAVIOR参数，通过setMaxLifecycle()将不可见的fragment的最大生命周期设置为STARTED。只需要将加载数据放在onResume()中即可实现懒加载。

## 2.3 使用方法

### 2.3.1 单独使用

（1）静态加载

直接在Activity布局文件中指定Fragment。使用指定属性name即可。

（2）动态加载

利用getSupportFragmentManager()获取FragmentTransaction。

① add()&remove()

添加和移除，replace()只是其实是先调用了remove()然后再调用add()。

② hide()&show()

隐藏和显示，这种方式防止Fragment多次创建实例对象，所以正确的切换方式是add()，切换时hide()，add()另一个Fragment；再次切换时，只需hide()当前，show()另一个，这样就能做到多个Fragment切换不重新实例化。

③ detach()&attach()

使用detach()会将view从ViewTree中删除,和remove()不同,此时Fragment的状态依然保持着,在使用attach()时会再次调用onCreateView()来重绘视图,注意使用detach()后Fragment.isAdded()方法将返回false,在使用attach()还原Fragment后isAdded()会依然返回false(需要再次确认)执行detach()和replace()后要还原视图的话, 可以在相应的Fragment中保持相应的view,并在onCreateView()方法中通过view的parent的removeView()方法将view和parent的关联删除后返回,这种方式极少使用。

### 2.3.2 结合ViewPager

（1）ViewPager

（2）ViewPager2

# 三、Handler

Handler 是一个消息分发对象。Handler是Android给我们提供用来更新UI的一套机制，也是一套消息处理机制，我们可以发消息，也可以通过它处理消息。

## 3.1 实现原理

（1）ActivityThread执行Looper.prepareMainLooper()，创建了一个Looper，并把它放在了ThreadLocal中；

（2）Looper.loop()将消息从MessageQueue中取出分发；

（3）Handler.sendMessage()发送消息，通过enqueueMessage()加入MessageQueue。

## 3.2 内存泄漏

### 3.2.1 原因

非静态内部类，或者匿名内部类。使得Handler默认持有外部类的引用。在Activity销毁时，由于Handler可能有未执行完/正在执行的Message，导致Handler持有Activity的引用。进而导致GC无法回收Activity。

### 3.2.2 解决方法

使用静态内部类并继承Handler。因为静态的内部类不会持有外部类的引用，所以不会导致外部类实例的内存泄露。当你需要在静态内部类中调用外部的Activity时，我们可以使用弱引用来处理。另外关于同样也需要将Runnable设置为静态的成员属性。

## 3.3 Handler框架

### 3.3.1 AsyncTask

本质上是一个封装了线程池和Handler的异步框架。

（1）AsyncTask<Params, Progress, Result>

① Params: 这个泛型指定的是我们传递给异步任务执行时的参数的类型；

② Progress: 这个泛型指定的是我们的异步任务在执行的时候将执行的进度返回给UI线程的参数的类型；

③ Result: 这个泛型指定的异步任务执行完后返回给UI线程的结果的类型。

我们在定义一个类继承AsyncTask类的时候，必须要指定好这三个泛型的类型，如果都不指定的话，则都将其写成Void，例如：

AsyncTask <Void, Void, Void>

（2）重写方法

① onPreExecute(): 这个方法是在执行异步任务之前的时候执行，并且是在UI Thread当中执行的，通常我们在这个方法里做一些UI控件的初始化的操作，例如弹出要给ProgressDialog；

② doInBackground(Params… params): 在onPreExecute()方法执行完之后，会马上执行这个方法，这个方法就是来处理异步任务的方法，Android操作系统会在后台的线程池当中开启一个worker thread来执行我们的这个方法，所以这个方法是在worker thread当中执行的，这个方法执行完之后就可以将我们的执行结果发送给我们的最后一个 onPostExecute 方法，在这个方法里，我们可以从网络当中获取数据等一些耗时的操作；

③ onProgressUpdate(Progess… values): 这个方法也是在UI Thread当中执行的，我们在异步任务执行的时候，有时候需要将执行的进度返回给我们的UI界面，例如下载一张网络图片，我们需要时刻显示其下载的进度，就可以使用这个方法来更新我们的进度。这个方法在调用之前，我们需要在 doInBackground 方法中调用一个 publishProgress(Progress) 的方法来将我们的进度时时刻刻传递给 onProgressUpdate 方法来更新；

④ onPostExecute(Result… result): 当我们的异步任务执行完之后，就会将结果返回给这个方法，这个方法也是在UI Thread当中调用的，我们可以将返回的结果显示在UI控件上。

（3）实现原理

① 它本质上是一个静态的线程池，AsyncTask派生出的子类可以实现不同的异步任务，这些任务都是提交到静态的线程池中执行。

② 线程池中的工作线程执行doInBackground(mParams)方法执行异步的任务。

③ 当任务状态改变后，工作线程向UI线程发送消息，AsyncTask内部的InternalHandler响应这些消息，并调用相关的回调函数。

### 3.3.2 HandlerThread

由Handler + Thread + Looper组成，是一个Thread内部有Looper。

本质上是一个它继承了Thread的线程类。HandlerThread有自己内部的Looper对象，可以进行Looper循环。通过获取HandlerThread的Looper对象传递给Handler对象，可以在handlerMessage方法中执行异步任务。优点是不会有堵塞，减少对性能的消耗，缺点是不能进行多任务的处理，需要等待进行处理，处理效率较低。

与线程池注重并发不同，HandlerThread是一个串行队列，HandlerThread背后只有一个线程。

### 3.3.3 IntentService

继承Service处理异步请求的一个类，优先级高于Service。在IntentService内有一个工作线程来处理耗时操作，启动IntentServiced的方式和启动传统的Service一样。当任务执行完成后，IntentService会自动停止，而不需要我们手动去控制或stopSelf()。另外，可以启动IntentService多次，而每一个耗时操作会以工作队列的方式在IntentService的onHandlerIntent回调方法中执行，并且，每次只执行一个工作线程，执行完第一个在执行第二个。内部是由HandlerThread和Handler实现异步操作。创建IntentService时，只需要实现onHandlerIntent和构造方法，onHandlerIntent为异步方法，可以执行耗时操作。

# 四、数据持久化

## 4.1 Android中数据持久化的方式与使用场景

### 4.1.1 Shared Preferences

1. 概述

有些时候，应用程序有少量的数据需要保存，并且这些数据的格式很简单。比如：软件设置、用户账户设置，用户习惯设置等，这个时候就可以用到SharedPreferences。SharedPreferences创建的文件总是保存在根目录的data/data/<package name>/shared\_prefs目录下。

SharedPreferences本身是一个接口，程序无法直接创建SharedPreferences的实例，只能通过Context提供的getSharedPreferences(String name,int mode)方法来获取SharedPreferences的实例。

第一个参数name为创建的文件名称，如设定了name为"SilverDemo"，则生成的文件名为：SilverDemo.xml。第二个参数mode为int类型，此参数描述了创建的文件的可见性。有如下值：

1. MODE\_PRIVATE：指定此SharedPreferences只有本程序可以访问。
2. MODE\_WORLD\_READABLE：指定此SharedPreferences对其他程序只读且无法改。
3. MODE\_WORLD\_WRITEABLE：指定此SharedPreferences能被其他程序读写。
4. MODE\_MULTI\_PROCESS：Android2.3之后已经弃之不用了。
5. SharedPreferences的常用方法：
6. boolean contains(String Key)：

判断SharedPreferences是否包含特定Key的数据。

1. abstract Map<String,?> getAll()：

获取SharedPreferences数据里全部的Key-Value对。

1. boolean getXxx(String key,Xxx defValue)：

获取SharedPreferences数据里指定Key对应的value。如果该Key不存在，返回默认值defValue。

SharedPreferences本身为接口类型，并没有提供写入数据的能力，而是通过SharedPreferences的内部接口实现写入数据的能力。调用edit()方法即可获取它所对应的Editor对象。

1. SharedPreferences.Editor的常用方法：
2. clear()：

清空SharedPreferences里所有的数据。

1. putXxx(String key,xxx value)：

向SharedPreferences中插入指定的Key-Value对。

1. remove(String key)：

从SharedPreferences中移除指定Key的数据。

1. boolean commit()：

当Editor编辑完成后，调用该方法提交修改。

1. SharedPreferences是线程安全的
2. SharedPreference的apply和commit的区别
   * 1. apply没有返回值而commit返回boolean表明修改是否提交成功；
     2. apply是将修改数据原子提交到内存, 而后异步真正提交到硬件磁盘, 而commit是同步的提交到硬件磁盘，因此，在多个并发的提交commit的时候，他们会等待正在处理的commit保存到磁盘后在操作，从而降低了效率。而apply只是原子的提交到内容，后面有调用apply的函数的将会直接覆盖前面的内存数据，这样从一定程度上提高了很多效率；
     3. apply方法不会提示任何失败的提示。 由于在一个进程中，sharedPreference是单实例，一般不会出现并发冲突，如果对提交的结果不关心的话，建议使用apply，当然需要确保提交成功且有后续操作的话，还是需要用commit的。

### 4.1.2 Internal Storage

内部存储，在Android中，开发者可以直接使用设备的内部存储器中保存文件，默认情况下，以这种方式保存的和数据是只能被当前程序访问，在其他程序中是无法访问到的，而当用户卸载该程序的时候，这些文件也会随之被删除。使用内部存储的方式进行数据持久化，文件的地址将保存在/data/data/<package\_name>/files/路径下。

缓存文件cache files的操作与操作内部存储中的文件方式基本一致，只是获取文件的路径有说不同。如果需要使用缓存的方式进行数据持久话，那么需要使用Context.getCacheDir()方法获取文件保存的路径。对于缓存文件而言，当设备内部内存存储空间不足的时候，Android会有自动删除的机制删除这些缓存文件，用来恢复可用空间，所以对于缓存文件而言，内容一般最好控制在1MB之下，并且也不要存放重要的数据，因为很可能下次去取数据的时候，已经被Android系统自动清理了。

### 4.1.3 External Storage

使用外部存储实现数据持久化，这里的外部存储一般就是指的是sdcard。使用sdcard存储的数据，不限制只有本应用访问，任何可以有访问Sdcard权限的应用均可以访问，而Sdcard相对于设备的内部存储空间而言，会大很多，所以一般比较大的数据，均会存放在外部存储中。

　　使用Sdcard存储数据的方式与内部存储的方式基本一致，但是有三点需要注意的：

1. 需要首先判断是否存在可用的Sdcard，这个可以使用一个访问设备环境变量的类Environment进行判断，这个类提供了一系列的静态方法，用于获取当前设备的状态，在这里获取是否存在有效的Sdcard，使用的是Environment.getExternalStorageState()方法，返回的是一个字符串数据，Environment封装好了一些final对象进行匹配，除了Environment.MEDIA\_MOUNTED外，其他均为有问题，所以只需要判断是否是Environment.MEDIA\_MOUNTED状态即可。
2. 既然转向了Sdcard，那么存储的文件路径就需要相对变更，这里可以使用Envir.getExternalStorageDirectory()方法获取当Sdcard的根目录，可以通过它访问到相应的文件。
3. 需要赋予应用程序访问Sdcard的权限，Android的权限控制尤为重点，在Android程序中，如果需要做一些越界的操作，均需要对其进行授权才可以访问。在AndroidManifest.xml中添加代码：<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/>

### 4.1.4 SQLite数据库

SQLiteOpenHelper，SQLiteDatabase，execSQL

### 4.1.5 三级缓存

三级缓存指的是：内存缓存、本地缓存、网络缓存。其各自的特点是内存缓存速度快, 优先读取，本地缓存速度其次, 内存没有,读本地，网络缓存速度最慢, 本地也没有,才访问网络。对于网络缓存理解起来较为容易直接从网络中获取资源，本地缓存可以存在SD卡中，内存缓存一般存在数组或集合中。需要在注意的是，数组和集合的生命周期依赖于它存在的activity中，因此当程序退出，一般情况下数组和集合中的资源会被释放。

## 4.2 MMKV

# 五、布局

## 5.1 常用布局及特点

### 5.1.1 相对布局RelativeLayout

RelativeLayout按照子元素之间的位置关系完成布局，作为 Android布局中最灵活也是最常用的一种布局方式，非常适合于一些比较复杂的界面设计。

### 5.1.2 线性布局LinearLayout

LinearLayout是按照水平或垂直的顺序将子元素依次按照顺序排列，每一个元素都位于前面一个元素之后。线性布局分为水平方向和垂直方向两种。

### 5.1.3约束布局ConstraintLayout

ConstraintLayout 使用约束的方式来指定各个控件的位置和关系的，它有点类似于 RelativeLayout，但远比 RelativeLayout 要更强大，它可以有效地解决布局嵌套过多的问题。

### 5.1.4 表格布局TableLayout

表格布局，适用于多行多列的布局格式，每个 TableLayout 是由多个 TableRow 组成，一个 TableRow 就表示 TableLayout 中的每一行，这一行可以由多个子元素组成。实际上 TableLayout 和 TableRow 都是 LineLayout 线性布局的子类。但是 TableRow 的参数 android:orientation 属性值固定为 horizontal，且 android:layout\_width=MATCH\_PARENT，android:layout\_height=WRAP\_CONTENT。所以 TableRow 实际是一个横向的线性布局，且所以子元素宽度和高度一致。

### 5.1.5 帧布局FrameLayout

FrameLayout 是最简单的布局。所有放在布局里的控件，都按照层次堆叠在屏幕的左上角。后加进来的控件覆盖前面的控件。在 FrameLayout 布局里，定义任何空间的位置相关的属性都毫无意义。控件自动的堆放在左上角，根本不听你的控制。但是控件本身是可以控制自己内部的布局的。

### 5.1.6 绝对布局Absolute Layout

绝对布局中将所有的子元素通过设置 android:layout\_x 和 android:layout\_y 属性，将子元素的坐标位置固定下来，即坐标 (android:layout\_x, android:layout\_y) ，layout\_x 用来表示横坐标，layout\_y 用来表示纵坐标。屏幕左上角为坐标 (0,0)，横向往右为正方，纵向往下为正方。实际应用中，这种布局用的比较少，因为 Android 终端一般机型比较多，各自的屏幕大小。分辨率等可能都不一样，如果用绝对布局，可能导致在有的终端上显示不全等。

## 5.2 性能对比

### 5.2.1 RelativeLayout和LinearLayout的onMeasure()方法

由源码可知RelativeLayout会对子View做两次measure。首先RelativeLayout中子View的排列方式是基于彼此的依赖关系，而这个依赖关系可能和布局中View的顺序并不相同，在确定每个子View的位置的时候，就需要先给所有的子View排序一下。又因为RelativeLayout允许A，B 2个子View，横向上B依赖A，纵向上A依赖B。所以需要横向纵向分别进行一次排序测量。

而LinearLayout的measure就简单明了的多了，先判断线性规则，然后执行对应方向上的测量。如果不使用weight属性，LinearLayout会在当前方向上进行一次measure的过程，如果使用weight属性，LinearLayout会避开设置过weight属性的view做第一次measure，完了再对设置过weight属性的view做第二次measure。因此weight属性对性能是有影响的。

### 5.2.2 RelativeLayout另一个性能问题

View的measure方法里对绘制过程做了一个优化，如果我们或者我们的子View没有要求强制刷新，而父View给子View的传入值也没有变化（也就是说子View的位置没变化），就不会做无谓的measure。但是RelativeLayout要做两次measure，而在做横向的测量时，纵向的测量结果尚未完成，只好暂时使用myHeight传入子View系统，假如子View的Height不等于（设置了margin）myHeight的高度，那么measure中上面代码所做得优化将不起作用，这一过程将进一步影响RelativeLayout的绘制性能。而LinearLayout则无这方面的担忧。解决这个问题也很好办，如果可以，尽量使用padding代替margin。

## 5.3 布局文件到控件对象的过程

### 5.3.1

# 六、IPC

## 6.1 多进程概述

### 6.1.1 多进程的定义

当一个应用在开始运行时，系统会为它创建一个进程，一个应用默认只有一个进程，这个进程（主进程）的名称就是应用的包名。

进程的特点：

1. 进程是系统资源和分配的基本单位，而线程是调度的基本单位；
2. 每个进程都有自己独立的资源和内存空间；
3. 其它进程不能任意访问当前进程的内存和资源；
4. 系统给每个进程分配的内存会有限制。

使用多进程的场景为：需要使apk所使用的内存限制扩大。

### 6.1.2 多进程的等级

1. 前台进程：

该进程包含正在与用户进行交互的界面组件，比如一个Activity。在接收关键生命周期方法时会让一个进程临时提升为前台进程，包括任何服务的生命周期方法onCreate()和onDestroy()和任何广播接收器onReceive()方法。这样做确保了这些组件的操作是有效的原子操作，每个组件都能执行完成而不被杀掉。

1. 可见进程：

该进程中的组件虽然没有和用户交互，但是仍然可以被看到。activity可见的时候不一定在前台。一个简单的例子是前台的 activity 使用对话框启动了一个新的 activity 或者一个透明 activity 。另一个例子是当调用运行时权限对话框时（事实上它就是一个 activity！）。

1. 服务进程：

该进程包含在执行后台操作的服务组件，比如播放音乐的Service。对于许多在后台做处理（如加载数据）而没有立即成为前台服务的应用都属于这种情况。

请特别注意从onStartCommand()返回的常量，如果服务由于内存压力被杀掉，它表示控制什么发生什么：

START\_STICKY表示希望系统可用的时候自动重启服务，但不关心是否能获得最后一次的 Intent （例如，可以重建自己的状态或者控制自己的 start/stop 生命周期）。

START\_REDELIVER\_INTENT是为那些在被杀死之后重启时重新获得 Intent 的服务的，直到用传递给 onStartCommand() 方法的 startId 参数调用stopSelf()为止。这里会使用 Intent 和 startId 作为队列完成工作。

START\_NOT\_STICKY用于那些杀掉也没关系的服务。这适合那些管理周期性任务的服务，它们只是等待下一个时间窗口工作。

1. 后台进程：

该进程包含的组件没有与用户交互，用户也看不到 Service。在一般操作场景下，设备上的许多内存就是用在这上面的，使可以重新回到之前打开过的某个 activity 。

1. 空进程：

没有任何界面组件、服务组件，或触发器组件，只是出于缓存的目的而被保留（为了更加有效地使用内存而不是完全释放掉），只要 Android 需要可以随时杀掉它们。

### 6.1.3 多进程的创建

Android多进程创建很简单，只需要在AndroidManifest.xml的声明四大组件的标签中增加“android:process”属性即可。命名之后，就成了一个单独的进程。process分为私有进程和全局进程：

私有进程的名称前面有冒号，例如：

<service android:name=".MusicService"

android:process=":musicservice"/>

全局进程的名称前面没有冒号，例如：

<service android:name=".MusicService"

android:process="com.trampcr.musicdemo.service"/>

## 6.2 多进程间通信IPC

### 6.2.1 Bundle/Intent

### 6.2.2 文件共享

### 6.2.3 Messenger

### 6.2.4 AIDL

### 6.2.5 ContentProvider

### 6.2.6 Socket

## 6.3 Binder

https://blog.csdn.net/tkwxty/article/details/112325376

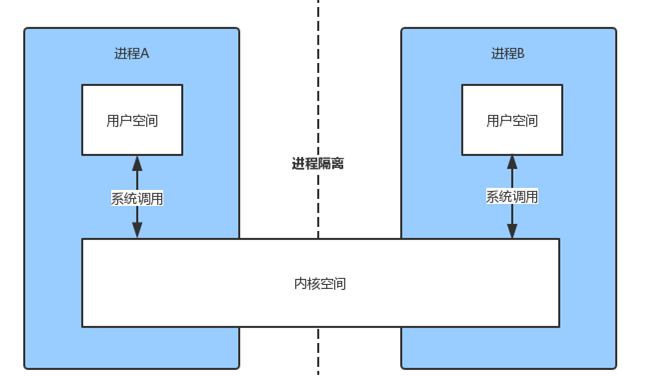
### 6.3.1 Linux和Android的IPC机制种类

Linux中提供了很多进程间通信机制，主要有管道（pipe）、信号（sinal）、信号量（semophore）、消息队列（Message）、共享内存（Share Memory)、套接字（Socket）等。

Android系统是基于Linux内核的，在Linux内核基础上，又拓展出了一些IPC机制。Android系统除了支持套接字，还支持序列化、Messenger、AIDL、Bundle、文件共享、ContentProvider、Binder等。

### 6.3.2 Linux的IPC通信原理

首先理解Liunx中的几个概念：



1. 内核空间和用户空间

为了保护用户进程不能直接操作内核，保证内核的安全，操作系统从逻辑上将虚拟空间划分为用户空间和内核空间。Linux 操作系统将最高的1GB字节供内核使用，称为内核空间，较低的3GB 字节供各进程使用，称为用户空间。内核空间是Linux内核的运行空间，用户空间是用户程序的运行空间。为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不会受到影响。内核空间的数据是可以进程间共享的，而用户空间则不可以。

1. 进程隔离

进程隔离指的是，一个进程不能直接操作或者访问另一个进程，也就是进程A不可以直接访问进程B的数据。

1. 系统调用

用户空间需要访问内核空间，就需要借助系统调用来实现。系统调用是用户空间访问内核空间的唯一方式，保证了所有的资源访问都是在内核的控制下进行的，避免了用户程序对系统资源的越权访问，提升了系统安全性和稳定性。

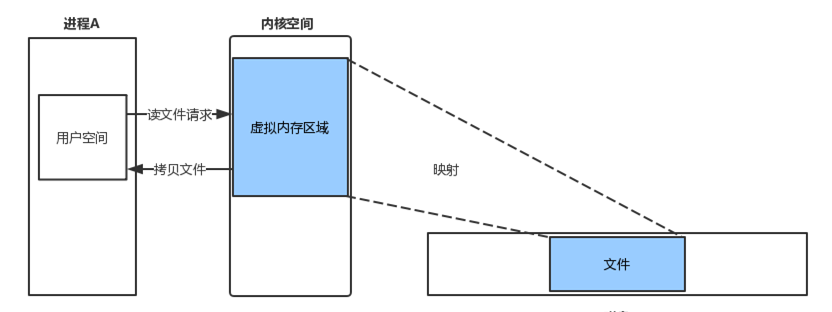
进程A和进程B的用户空间可以通过如下系统函数和内核空间进行交互：

copy\_from\_user：将用户空间的数据拷贝到内核空间。

copy\_to\_user：将内核空间的数据拷贝到用户空间。

1. 内存映射

由于应用程序不能直接操作设备硬件地址，所以操作系统提供了一种机制：内存映射，把设备地址映射到进程虚拟内存区。比如用户空间需要读取磁盘的文件，如果不采用内存映射，那么就需要在内核空间建立一个页缓存，页缓存去拷贝磁盘上的文件，然后用户空间拷贝页缓存的文件，这就需要两次拷贝。采用内存映射，如下图所示。



由于新建了虚拟内存区域，那么磁盘文件和虚拟内存区域就可以直接映射，少了一次拷贝。

内存映射全名为Memory Map，在Linux中通过系统调用函数mmap来实现内存映射。将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间，反之亦然。内存映射能减少数据拷贝次数，实现用户空间和内核空间的高效互动。

MMAP的内存映射原理：

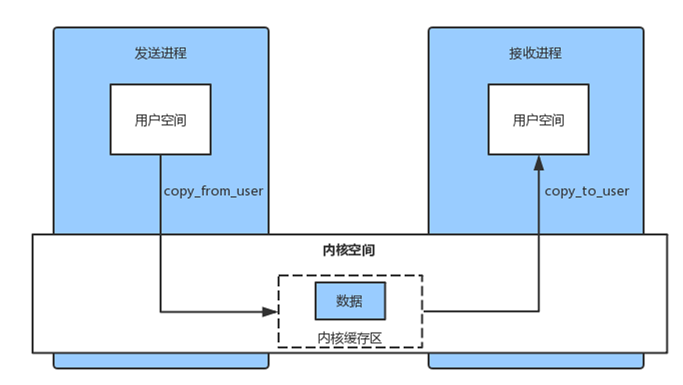
MMAP内存映射的实现过程，总的来说可以分为三个阶段：

1. 进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域：
   * 1. 进程在用户空间调用库函数mmap，原型：void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);
     2. 在当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续的虚拟地址；
     3. 为此虚拟区分配一个vm\_area\_struct结构，接着对这个结构的各个域进行了初始化；
     4. 将新建的虚拟区结构（vm\_area\_struct）插入进程的虚拟地址区域链表或树中；
2. 调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系：
   * 1. 为映射分配了新的虚拟地址区域后，通过待映射的文件指针，在文件描述符表中找到对应的文件描述符，通过文件描述符，链接到内核“已打开文件集”中该文件的文件结构体（struct file），每个文件结构体维护着和这个已打开文件相关各项信息；
     2. 通过该文件的文件结构体，链接到file\_operations模块，调用内核函数mmap，其原型为：int mmap(struct file \*filp, struct vm\_area\_struct \*vma)，不同于用户空间库函数；
     3. 内核mmap函数通过虚拟文件系统inode模块定位到文件磁盘物理地址；
     4. 通过remap\_pfn\_range函数建立页表，即实现了文件地址和虚拟地址区域的映射关系。此时，这片虚拟地址并没有任何数据关联到主存中；
3. 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝

注：前两个阶段仅在于创建虚拟区间并完成地址映射，但是并没有将任何文件数据的拷贝至主存。真正的文件读取是当进程发起读或写操作时。进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上。因为目前只建立了地址映射，真正的硬盘数据还没有拷贝到内存中，因此引发缺页异常。

* + 1. 缺页异常进行一系列判断，确定无非法操作后，内核发起请求调页过程；
    2. 调页过程先在交换缓存空间（swap cache）中寻找需要访问的内存页，如果没有则调用nopage函数把所缺的页从磁盘装入到主存中；
    3. 之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。

Linux的IPC通信原理如下图所示。



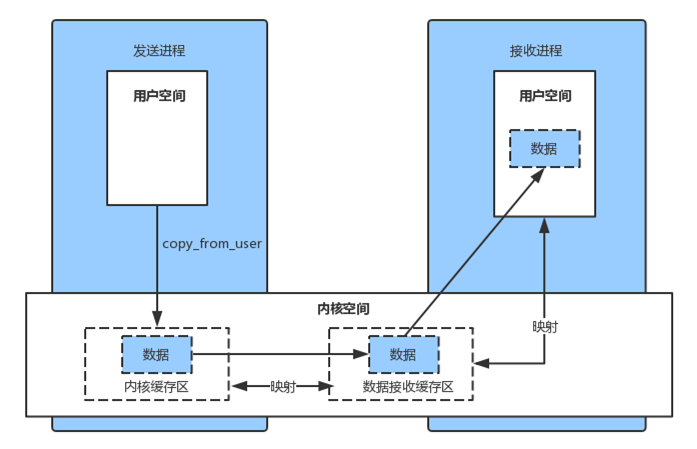
内核程序在内核空间分配内存并开辟一块内核缓存区，发送进程通过copy\_from\_user函数将数据拷贝到到内核空间的缓冲区中。同样的，接收进程在接收数据时在自己的用户空间开辟一块内存缓存区，然后内核程序调用 copy\_to\_user() 函数将数据从内核缓存区拷贝到接收进程。这样数据发送进程和数据接收进程完成了一次数据传输，也就是一次进程间通信。

Linux的IPC通信原理有两个问题：

1. 一次数据传递需要经历：用户空间 --> 内核缓存区 --> 用户空间，需要2次数据拷贝，这样效率不高。
2. 接收数据的缓存区由数据接收进程提供，但是接收进程并不知道需要多大的空间来存放将要传递过来的数据，因此只能开辟尽可能大的内存空间或者先调用API接收消息头来获取消息体的大小，浪费了空间或者时间。

### 6.3.3 Binder的通信原理

Binder通信的步骤如下所示。



* 1. Binder驱动在内核空间创建一个数据接收缓存区。
  2. 在内核空间开辟一块内核缓存区，建立内核缓存区和数据接收缓存区之间的映射关系，以及数据接收缓存区和接收进程用户空间地址的映射关系。
  3. 发送方进程通过copy\_from\_user()函数将数据拷贝到内核中的内核缓存区，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

整个过程只使用了1次拷贝，不会因为不知道数据的大小而浪费空间或者时间，效率更高。

### 6.3.4 Binder的特点

1. 性能方面

主要影响的因素是拷贝次数，管道、消息队列、Socket的拷贝次书都是两次，性能不是很好，共享内存不需要拷贝，性能最好，Binder的拷贝次书为1次，性能仅次于内存拷贝。

1. 稳定性方面

Binder是基于C/S架构的，这个架构通常采用两层结构，在技术上已经很成熟了，稳定性是没有问题的。共享内存没有分层，难以控制，并发同步访问临界资源时，可能还会产生死锁。从稳定性的角度讲，Binder是优于共享内存的。

1. 安全方面

Android是一个开源的系统，并且拥有开放性的平台，市场上应用来源很广，因此安全性对于Android 平台而言极其重要。传统的IPC接收方无法获得对方可靠的进程用户ID/进程ID（UID/PID），无法鉴别对方身份。Android 为每个安装好的APP分配了自己的UID，通过进程的UID来鉴别进程身份。另外，Android系统中的Server端会判断UID/PID是否满足访问权限，而对外只暴露Client端，加强了系统的安全性。

1. 语言方面

Linux是基于C语言，C语言是面向过程的，Android应用层和Java Framework是基于Java语言，Java语言是面向对象的。Binder本身符合面向对象的思想。

### 6.3.5 binder源码解析

三层结构

java binder(stub&prox)

native binder(bpbinder&bbinder)

kernel binder(open,mmap,icol)

# 七、图片编程

## 7.1 Bitmap基础

## 7.2 Bitmap导致的OOM

## 7.3 图片压缩

## 7.4 图片缓存

# 八、View

## 8.1 View基础知识

### 8.1.1 View的位置参数

View的位置主要由它的四个顶点来决定，分别对应View的四个属性：top，left，right，bottom。这些坐标是相对父容器来说的，因此它是一种相对坐标。从Android3.0开始，View增加了几个额外参数：x，y，translationX和translationY。其中x和y是View左上角的坐标，而translationX和translationY是View左上角相对于父容器的偏移量。

### 8.1.2 MotionEvent

1. 触摸事件的基本类型有三种：
2. ACTION\_DOWN: 表示手指按下屏幕
3. ACTION\_MOVE: 手指在屏幕上滑动时，会产生一系列的MOVE事件
4. ACTION\_UP: 手指抬起，离开屏幕
5. ACTION\_CANCEL：当出现异常情况事件序列被中断，会产生该类型事件

完整的事件序列是：从ACTION\_DOWN开始，到ACTION\_UP或者ACTION\_CANCEL结束 。当然，这是我们一个手指的情况，那么在多指操作的情况是怎么样的呢？这里需要引入另外的事件类型：

1. ACTION\_POINTER\_DOWN: 当已经有一个手指按下的情况下，另一个手指按下会产生该事件
2. ACTION\_POINTER\_UP: 多个手指同时按下的情况下，抬起其中一个手指会产生该事件

区别于ACTION\_DOWN和ACTION\_UP，使用另外两个事件类型来表示手指的按下与抬起，使得ACTION\_DOWN和ACTION\_UP可以作为一个完整的事件序列的边界 。同时，一个手指的事件序列，是从ACTION\_DOWN/ACTION\_POINTER\_DOWN开始，到ACTION\_UP/ACTION\_POINTER\_UP/ACTION\_CANCEL结束。

1. MotionEvent携带的信息

在MotionEvent对象内部，维护有一个数组。这个数组中的每一项对应不同的触摸点的信息，数组下标称为触控点的索引，每个节点，拥有一个触控点的完整信息。

这里要注意的是，一个触控点的索引并不是一成不变的，而是会随着触控点的数目变化而变化。所以跟踪一个触控点必须是依靠一个触控点的id，而不是他的索引。

判断事件对应哪个触控点需要使用MotionEvent的getAction()方法 。这个方法返回一个整型变量，他的低1-8位表示该事件的类型，高9-16位表示触控点索引。我们只需要将这16位进行分离，就可以知道触控点的类型和所对应的触控点。同时，MotionEvent有两个获取触控点坐标的方法：getX()/getY() ，他们都需要传入一个触控点索引来表示获取哪个触控点的坐标信息。

### 8.1.3 TouchSlop

系统所能识别出的被认为是滑动的最小距离。

ViewConfiguration.get(getContext()).getScaledTouchSlop()

### 8.1.4 VelocityTracker

速度追踪，用于追踪手指在滑动过程中的速度，包括水平和竖直方向的速度。

VelocityTracker velocityTracker = VelocityTracker.obtain();

velocityTracker.addMovement(event);

velocityTracker.computeCurrentVelocity(1000);

int xVelocity = (int) velocityTracker.getXVelocity;

int YVelocity = (int) velocityTracker.getYVelocity;

computeCurrentVelocity这个方法的参数表示的是一个时间单元或者说时间间隔，它的单位是ms毫秒，得到的速度是在这个时间间隔内手指在水平或竖直方向上所滑动的像素数。最后，当不需要使用VelocityTracker时，需要调用clear方法来重置并回收内存。

velocityTracker.clear();

velocityTracker.recycle();

### 8.1.5 GestureDetector

手势检测，用于辅助检测用户的单击，滑动，长按，双击等行为。

### 8.1.6 Scroller

弹性滑动对象，用于实现View的弹性滑动。Scroller本身无法让View弹性滑动，需要和View的computeScroll方法配合使用。

## 8.2 View的滑动

### 8.2.1 scrollTo/scrollBy

只能改变View内容的位置而不能改变View在布局中的位置。mScrollX和mScrollY的单位为像素，内容左边缘在左边缘左侧、内容上边缘在上边缘上侧时值为正。适合用于对View内容的滑动。

### 8.2.2 使用动画

View动画是对View的影像做操作，它并不能真正改变View的位置参数。使用属性动画并不会存在这个问题，但是Android3.0以下无法使用属性动画，此时需要使用动画兼容库nineoldandroids来实现动画。适用于实现复杂的动画效果。

### 8.2.3 改变布局参数

即改变LayoutParams。还有一种情形，可以在View-A旁边放置一个空的View-B，默认宽度为0，需要移动View-A时重新设置View-B的参数即可。操作稍微复杂，适用于有交互的View。

## 8.3 弹性滑动

### 8.3.1 使用Scroller

Scroller本身并不能实现View的滑动，它需要配合View的computeScroll方法才能完成弹性滑动的效果，它不断地让View重绘，而每一次重绘距滑动起始时间会有一个时间间隔，通过这个时间间隔Scroller就可以得出View当前的滑动位置，知道了滑动位置就可以通过scrollTo方法来完成View的滑动。

### 8.3.2 使用动画

可以利用动画的特性来实现一些动画不能实现的效果。通过利用在动画每一帧到来时获取动画完成的比例，根据这个比例来获得需要的结果。

### 8.3.3 使用延时策略

通过发送一系列延时消息达到一种渐进式的效果，可以使用Handler或View的postDelayed，也可以使用线程的sleep方法。

## 8.4 事件分发机制

### 8.4.1事件分发的本质

事件分发的本质将点击屏幕产生的MotionEvent对象传递到某个具体的View然后处理消耗这个事件的过程。

### 8.4.2 三个重要方法

1. dispatchTouchEvent

用来进行事件的分发。如果事件能够传递给当前View,那么此方法一定会被调用，返回结果受当前View的onTouchEvent和下级的dispatchTouchEvent方法影响，表示是否消耗此事件。

1. onInterceptTouchEvent

在dispatchTouchEvent内部调用，用来判断是否拦截某个事件，如果当前View拦截了某个事件，那么同一个事件序列当中，此方法不会被再次调用，返回结果表示是否拦截当前事件。

1. onTouchEvent

同样在dispatchTouchEvent内部调用，用来处理点击事件，返回结果表示是否消耗当前事件，如果不消耗，则在同一个事件序列中，当前View无法再次接收到事件。

1. 三者关系

通过上述伪代码可知点击事件的传递规则：对于一个根ViewGroup而言，点击事件产生后，首先会传递给它，这时它的dispatchTouch就会被调用，如果这个ViewGroup的onInterceptTouchEvent方法返回true就表示它要拦截当前的事件，接着事件就会交给这个ViewGroup处理，即它的onTouch方法就会被调用；如果这个ViewGroup的onInterceptTouchEvent方法返回false就表示它不拦截当前事件，这时当前事件就会继续传递给它的子元素，接着子元素的dispatchTouchEvent方法就会被调用，如此直到事件被最终处理。

当一个View需要处理事件时，如果它设置了OnTouchListener,那么OnTouchListener中的onTouch方法会被回调。这时事件处理还要看onTouch的返回值，如果返回false,则当前View的onTouchEvent方法会被调用;如果返回true,那么当前View的onTouchEvent方法不会被调用。由此可见，给View设置的onTouchListener的优先级比onTouchEvent要高。在onTouchEvent方法中，如果当前设置的有onClickListener，那么它的onClick方法会被调用。可以看出，平时我们常用的OnClickListener,其优先级最低，即处于事件传递的尾端。

当一个点击事件产生后，它的传递过程遵循如下顺序：Activity–>Window–>View,即事件总是先传递给Activity，Activity再传递给Window，最后Window再传递给顶级View，顶级View接收到事件后，就会按照事件分发机制去分发事件。考虑一种情况，如果一个View的onTouchEvent返回false,那么它的父容器的onTouchEvent将会被调用，依次类推。如果所有的元素都不处理这个事件，那么这个事件将会最终传递给Activity处理， 即Activity的onTouchEvent方法会被调用。这个过程其实很好理解，我们可以换一种思路，假设点击事件是一个难题，这个难题最终被上级领导分给了一个程序员去处理（这是事件分发过程），结果这个程序员搞不定（onTouchEvent返回了false）,现在该怎么办呢？难题必须要解决，那就只能交给水平更高的上级解决（上级的onTouchEvent被调用），如果上级再搞不定，那就只能交给上级的上级去解决，就这样难题一层层地向上抛，这是公司内部一种常见的处理问题的过程。

1. 结论
2. 同一个事件序列是从手指接触屏幕的那一刻起，到手指离开屏幕的那一刻结束，在这个过程中所产生的一系列事件，这个事件的序列以down开始，中间含有数量不定的move事件，最终以up事件结束。
3. 正常情况下，一个事件序列只能被一个View拦截且消耗。这一条的原因可以参考（3），因为一旦一个元素拦截了某个事件，那么同一个事件序列的所有事件都会直接交给它处理，因此同一个事件序列中的事件不能分别由两个View同时处理，但是通过特殊手段可以做到，比如一个View将本该自己处理的事件通过onTouchEvent强行传递给其他View处理。
4. 某个View一旦决定拦截，那么这个事件序列都只能由它来处理（如果事件序列能够传递给它的话），并且它的onInterceptTouchEvent不会被调用。这条也很好理解，就是说当一个View决定拦截一个事件后，那么系统会把同一个事件序列内的其他方法都直接交给它来处理，因此就不用再调用这个View的onInterceptTouchEvent去询问它是否拦截了。
5. 某个View一旦开始处理事件，如果它不消耗ACTION\_DOWN事件（onTouchEvent返回了false），那么同一件序列中的其他事件都不会再交给它处理，并且事件将重新交由它的父元素去处理，即父元素的onTouchEvent会被调用。意思就是事件一旦交给一个View处理，那么它就必须消耗掉，否则同一事件序列中剩下的事件就不再交给它处理了，这就好比上级交给程序员一件事，如果这件事没有处理好，短时间内上级就不敢再把事件交给这个程序员做了，二者是类似的道理。
6. 如果View不消耗ACTION\_DOWN以外的事件，那么这个点击事件会消失，此时父元素的onTouchEvent并不会调用，并且当前View可以持续收到后续的事件，最终这些消失的点击事件会传递给Activity处理。
7. ViewGroup默认不拦截任何事件。Android源码中ViewGroup的onInterceptTouchEvent方法默认返回false。
8. View没有onInterceptTouchEvent方法，一旦点击事件传递给它，那么它的onTouchEvent方法就会被调用。
9. View的onTouchEvent默认都会消耗事件（返回true）,除非它是不可点击的（clickable和longClickable同时为false）。View的longClickable属性默认为false，clickable属性要分情况，比如Button的clickable属性默认为true,而TextView的clickable属性默认为false。
10. View的enable属性不影响onTouchEvent的默认返回值。哪怕一个View是disable状态的，只要它的clickable或者longClickable有一个为true,那么它的onTouchEvent就返回true。
11. onClick会发生的前提是当前View是可点击的，并且它接收到了down和up事件。
12. 事件传递过程是由外向内的，即事件总是先传递给父元素，然后再由父元素分发给子View,通过requestDisallowInterTouchEvent方法可以在子元素中干预父元素的事件分发过程，但是ACTION\_DOWN事件除外。

### 8.4.3 事件分发流程&源码分析

1. 触摸事件的产生

当手指触摸屏幕时，即产生了触摸信息。这个触摸信息由屏幕这个硬件产生，被系统底层驱动获取，交给Android的输入系统服务：InputManagerService，也就是IMS。

IMS会对这个触摸信息进行处理，通过WMS找到要分发的window，随后发送给对应的viewRootImpl。所以发送触摸信息的并不是WMS，WMS提供的是window的相关信息。

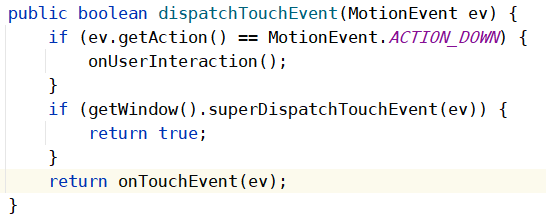
1. ViewRootImpl分发事件

ViewRootImpl管理一棵View树，View树的最外层是ViewGroup。顶层的ViewGroup为DecorView，因此会调用DecorView的 dispatchTouchEvent 方法进行分发。DecorView重写了该方法，逻辑比较简单，仅仅做了一个判断：如果window callBack对象不为空，则调用callBack对象的分发方法进行分发；如果window callBack对象为空，则调用父类ViewGroup的事件分发方法进行分发。

Activity实现了Window.CallBack接口，并在创建布局的时候，把自己设置给了DecorView，因此在Activity的布局界面中，DecorView会把事件分发给Activity进行处理。同理，在Dialog的布局界面中，会分发给实现了callBack接口的Dialog。

1. callBack控件对事件的分发
   1. Acvitity

Activity对于callBack接口方法的实现：



可以看到Activity直接调用PhoneWindow的方法进行分发。如果事件没有被处理，那么自己处理这个事件。PhoneWindow直接调用DecorView的方法进行分发。DecorView对于事件也是没有做任何处理，直接调用父类的方法进行分发。DecorView继承自FrameLayout，但是FrameLayout并没有重写 dispatchTouchEvent 方法，所以调用的就是viewGroup类的方法。

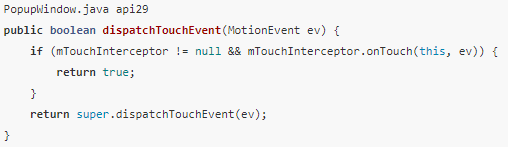
* 1. Dialog



mWindow是Dialog内部的PhoneWindow，后面的逻辑与Activity相同。

* 1. PopupWindow

PopupWindow他的根View是 PopupDecorView ，而不是 DecorView 。虽然他的名字带有DecorView，但是却和DecorView一点关系都没有，他是直接继承于FrameLayout。我们看到他的事件分发方法：



mTouchInterceptor 是一个拦截器，我们可以手动给PopupWindow设置拦截器。时间会优先交给拦截器处理，如果没有拦截器或拦截器没有消费事件，那么才会交给viewGroup去进行分发。

1. ViewGroup的事件分发https://juejin.cn/post/6920883974952714247

一个触控点的序列一般情况下只给一个view处理，当一个view消费了一个触控点的down事件后，该触控点的事件序列后续事件都会交给他处理。对于viewGroup来说，他有很多个子view，如果不同的子view接受了不同的触控点的down事件，那么ViewGroup如何记录这些信息并精准把事件发送给对应的子view呢？答案就是：TouchTarget。

TouchTarget中维护了每个子view以及所对应的触控点id，这里的id可以不止一个。TouchTarget本身是个链表，每个节点记录了子view所对应的触控点id。在viewGroup中，该链表的链表头是mFirstTouchTarget，如果他为null，表示没有任何子view接收了down事件。

TouchTarget有个非常神奇的设计，他只使用一个整型变量来记录所有的触控id。整型变量中哪一个二进制位为1，则对应绑定该id的触控点。

例如 00000000 00000000 00000000 10001000，则表示绑定了id为3和id为7的两个触控点，因为第3位和第7位的二进制位是1。这里可以间接说明系统支持的最大多点触控数是32，当然实际上一般是8比较多。当要判断一个TouchTarget绑定了哪些id时，只需要通过一定的位操作即可，既提高了速度，也优化了空间占用。

当一个down事件来临时，viewGroup会为这个down事件寻找适合的子view，并为他们创建一个TouchTarget加入到链表中。而当一个up事件来临时，viewGroup会把对应的TouchTarget节点信息删除。

总结：

* 1. 每一个触控点的事件序列，只能给一个view消费；如果一个view消费了一个触控点的down事件，那么该触控点的后续事件都会给他处理。
  2. 每一个事件到达viewGroup，如果需要分发到子view，那么viewGroup会新判断是否要拦截。
     1. 当viewGroup的touchTarget!=null || 事件的类型为down 需要进行判断是否拦截；
     2. 判断是否拦截受两个因素影响：onInterceptTouchEvent和FLAG\_DISALLOW\_INTERCEPT标志
  3. 如果该事件是down类型，那么需要遍历所有的子控件判断是否有子控件消费该down事件
     1. 当有新的down事件被消费时，viewGroup会把该view和对应的触控点id绑定起来存储到touchTarget中
  4. 根据前面的处理情况，将事件派发到viewGroup自身或touchTarget中
     1. 如果touchTarget==null，说明没有子控件消费了down事件，那么viewGroup自己处理事件
     2. 否则将事件分离成多个MotionEvent，每个MotionEvent只包含对应view感兴趣的触控点的信息，并派发给对应的子view

1. View的事件分发
2. 首先判断是否被其他非全屏view覆盖。这和上面viewGroup的安全性检查是一样的
3. 经过检查之后先检查是否有onTouchListener监听器，如果有则调用它
4. 如果第2步没有消费事件，那么会调用onTouchEvent方法来处理事件，里面包含了点击、双击、长按等逻辑的处理。

## 8.5 滑动冲突

### 8.5.1 常见的滑动冲突场景

1. 外部滑动方向和内部滑动方向不一致

主要是将ViewPager和Fragment配合使用所组成的页面滑动效果

1. 外部滑动方向和内部滑动方向一致

主要是指内外两层同时能上下滑动或左右滑动

1. 上述两者的嵌套

### 8.5.2 滑动冲突的解决方式

1. 外部拦截法

指点击事件都先经过父容器的拦截处理，如果父容器需要此事件就拦截，如果不需要此事件就不拦截。

1. 内部拦截法

## 8.6 View的工作原理

### 8.6.1 ViewRoot和DecorView

ViewRoot对应ViewRootImpl类，它是连接WindowManager和DecorView的纽带。在ActivityThread中，当Activity对象被创建完毕后，会将DecorView添加到Window中，同时创建ViewRootImpl对象，并将ViewRootImpl对象和DecorView建立关联。

View的三大流程均是通过ViewRoot来完成的。View的绘制流程是从ViewRoot的performTraversals方法开始的，经过measure、layout、draw三个过程将View绘制出来。其中measure用来测量View的宽和高，layout用来确定View在父容器中的位置，draw负责将View绘制在屏幕上。

### 8.6.2 MeasureSpec

MeasureSpec代表一个32位int值，高2位代表SpecMode，指测量模式，低30位代表SpecSize，指在某种测量模式下的规格大小。

SpecMode有三类：

1. UNSPECIFIED
2. EXACTLY
3. AT\_MOST

### 8.6.3 View的工作流程

1. measure过程

setMeasuredDimension()

1. layout过程
2. draw过程
3. 绘制背景background.draw(canvas)
4. 绘制自己onDraw
5. 绘制children（dispatchDraw）
6. 绘制装饰（onDrawScrollBars）

## 8.7 自定义View

### 8.7.1 自定义View的分类

1. 继承View重写onDraw方法
2. 继承ViewGroup派生特殊的Layout
3. 继承特定的View
4. 继承特定的ViewGroup

### 8.7.2 自定义View须知

1. 让View支持wrap\_content
2. 如果有必要，让View支持padding
3. 尽量不要在View中使用Handler
4. View中如果有线程或动画，需要及时停止
5. View带有滑动嵌套情形时，需要处理滑动冲突

## 8.8 ListView和RecyclerView

### 8.8.1 ListView和RecyclerView的区别

（1）ListView：

优点：

1. 对item的点击事件能够很快相应，实现简单；
2. CursorAdapter游标适配器能很方便与item数据绑定；
3. 能够在xml中轻松定义divide。

缺点：item的动画，decoration与触摸交互实现起来困难。

（2）RecyclerView：

优点：

1. 高度自定义化的onClick事件，能够在viewHolder创建的时候为view设置监听器，通过view.setTag的方式，把dataList里面的值或者position传入view中绑定，再在监听器实现中，获取参数并回调相应的接口，并通过自定义的接口回传给外部的activity；
2. layoutManager的灵活使用，支持水平线性滑动，以及瀑布流形式；
3. ItemAnimator支持多种item添加以及删除动画，更好的交互效果。

缺点：实现相较listview会较复杂。

（3）总结：RecyclerView自定义强，如果listview布局复杂，也能在RecyclerView.Adapter中灵活处理。

### 8.8.2 RecyclerView的使用

1. setLayoutManager需要一个LayoutManager，这个类有三个实现，分别是：1. LinearLayoutManager 线性管理器，支持横向、纵向

2. GridLayoutManager 网格布局管理器

3. StaggeredGridLayoutManager 瀑布就式布局管理器

1. RecyclerView包含了一种新型适配器，它也需要使用ViewHolder，使用时需要重写两个主要方法：一个用来展现视图和它的持有者的onCreateViewHolder(ViewGroup parent,int viewType)，一个用来把数据绑定到视图上的onBindViewHolder(ViewHolder holder,int position)。这么做的好处是，onCreateViewHolder只有当我们真正需要创建一个新视图时才被调用，不需要检查它是否已经被回收。
2. 多Item布局，利用getItemViewType(int position)获取item的类型，再根据viewType判断去创建不同item的ViewHolder。

### 8.8.3 RecyclerView的缓存机制

RecyclerView的缓存分为四级：Scrap，Cache，ViewCacheExtension，RecycledViewPool。

Scrap，对应ListView 的Active View，就是屏幕内的缓存数据，就是相当于换了个名字，可以直接拿来复用。

Cache，刚刚移出屏幕的缓存数据，默认大小是2个，当其容量被充满同时又有新的数据添加的时候，会根据FIFO原则，把优先进入的缓存数据移出并放到下一级缓存中，然后再把新的数据添加进来。Cache里面的数据是干净的，也就是携带了原来的ViewHolder的所有数据信息，数据可以直接来拿来复用。需要注意的是，cache是根据position来寻找数据的，这个postion是根据第一个或者最后一个可见的item的position以及用户操作行为（上拉还是下拉）。

举个栗子：当前屏幕内第一个可见的item的position是1，用户进行了一个下拉操作，那么当前预测的position就相当于（1-1=0），也就是position=0的那个item要被拉回到屏幕，此时RecyclerView就从Cache里面找position=0的数据，如果找到了就直接拿来复用。

ViewCacheExtension是google留给开发者自己来自定义缓存的，慎用。

RecycledViewPool，由于Cache默认的缓存数量是2个，当Cache缓存满了以后会根据FIFO（先进先出）的规则把Cache先缓存进去的ViewHolder移出并缓存到RecycledViewPool中，RecycledViewPool默认的缓存数量是5个。RecycledViewPool与Cache相比不同的是，从Cache里面移出的ViewHolder再存入RecycledViewPool之前ViewHolder的数据会被全部重置，相当于一个新的ViewHolder，而且Cache是根据position来获取ViewHolder，而RecycledViewPool是根据itemType获取的，如果没有重写getItemType（）方法，itemType就是默认的。因为RecycledViewPool缓存的ViewHolder是全新的，所以取出来的时候需要走onBindViewHolder（）方法。

### 8.8.4上拉刷新 & 下拉加载

SwipeRefreshlayout

### 8.8.5 ListView和RecyclerView的优化

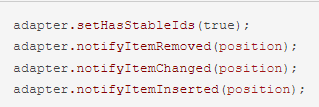
1. ListView
   1. 在adapter中的getView方法中尽量少使用逻辑
   2. 尽最大可能避免GC
   3. 滑动的时候不载入图片
   4. 将ListView的scrollingCache和animateCache设置为false

scrollingCache: scrollingCache本质上是drawing cache，你可以让一个View将他自己的drawing保存在cache中（保存为一个bitmap），这样下次再显示View的时候就不用重画了，而是从cache中取出。默认情况下drawing cahce是禁用的，因为它太耗内存了，但是它确实比重画来的更加平滑。而在ListView中，scrollingCache是默认开启的，我们可以手动将它关闭

animateCache: ListView默认开启了animateCache，这会消耗大量的内存，因此会频繁调用GC，我们可以手动将它关闭掉

* 1. item的布局层级越少越好
  2. 使用ViewHolder

1. RecyclerView
   1. 使用ConstraintLayout减少布局层级；
   2. 如果item的高度固定，可以设置RecyclerView.setHasFixedSize(true)这样可以让每次绘制Item时不再重新计算Item高度；
   3. 根据需求修改RecyclerView默认的绘制缓存选项；
   4. 在onBindViewHolder方法中，减少逻辑判断，减少临时对象创建。例如：复用事件监听，在其方法外部创建监听，可以避免生成大量的临时变量；
   5. 避免整个列表的数据更新，只更新受影响的布局。例如，加载更多时，不使用notifyDataSetChanged()，而是使用notifyItemRangeInserted(rangeStart, rangeEnd)；
   6. 对于RecyclerView，持有item具有的独特id,可以很容易地确定具体item并单独更新，当变化发生时，可以按照如下方式更新，从而避免整体刷新。



# 九、动画

## 9.1

# 十、Android开源库

## 10.1 Rxjava

### 10.1.1 设计思想

ReactiveX是异步编程的响应式扩展。RxJava是在Java上的响应式扩展，通过使用可观察序列，用于组成异步和基于事件编程的类库。响应式编程是面向数据流的编程思想，在响应式编程思想下，一切皆数据流。响应式编程所侧重的是数据流的流动。

在响应式编程的思想下，我们所关心的是数据流的流动。如网络请求这一数据流发起后，数据流流向服务器接口，经过服务器接口处理后流出返回的数据流，返回的数据流再经过若干步的处理，最后流出结果为我们所需要的数据流。

### 10.1.2 使用方法

RxJava基于观察者模式，而且是链式调用的。RxJava 有以下三个基本的元素：被观察者Observable，观察者Observer，订阅，subscribe。

RxJava的常用操作符有：

（1）创建操作符

（2）转换操作符

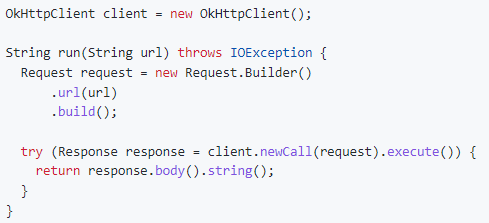
（3）组合操作符

### 10.1.3 线程切换原理

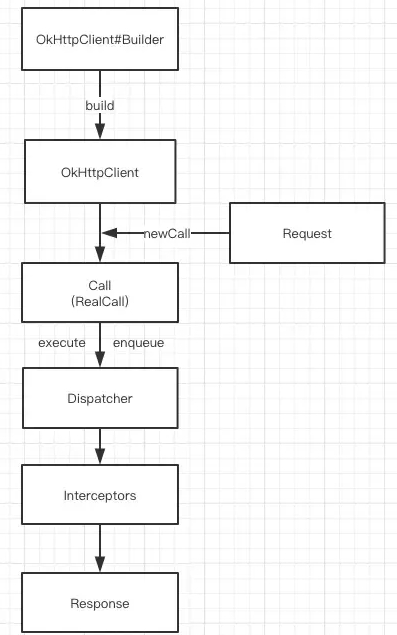
https://blog.csdn.net/zhujiangtaotaise/article/details/112579137

## 10.2 OKHttp

### 10.2.1 OKHttp的基本使用



### 10.2.2 请求的整体流程



1. 通过建造者模式构建OKHttpClient与Request；
2. OKHttpClient.newCall()创建RealCall对象发出请求；
3. 分发器维护线程池与三个请求队列，执行execute()或enqueue()完成请求调配；
4. 拦截器通过责任链完成请求重试，缓存处理，建立连接等一系列操作；
5. 得到网络请求结果；

### 10.2.3 分发器Dispatcher

1. RealCall的Dispatcher内部维护有一个线程池，一个同步队列runningSyncCalls，两个异步队列readyAsyncCalls与runningAsyncCalls。execute()将call放入runningSyncCalls，enqueue()会判断正在执行的异步请求个数是否小于64个，同个域名的请求是否不大于5个这两个条件选择放入runningAsyncCalls或readyAsyncCalls。当有请求结束（无论成功与否）时，Dispatcher会调用finish方法，将其从队列中移除，并把等待中的请求放入runningSyncCalls。
2. 线程池使用SynchronousQueue，该容器没有容量，无等待，高并发。

### 10.2.3 拦截器Interceptor

RealCall的getResponseWithInterceptorChain()方法构建了责任链。共有七种拦截器，按责任链顺序依次为：

* 1. 应用拦截器client. interceptors

拿到的是原始请求，可以添加一些自定义header、通用参数、参数加密、网关接入等等。通常用于统计客户端的网络请求发起情况。

* 1. 重试和重定向拦截器RetryAndFollowUpInterceptor

处理错误重试和重定向。

* 1. 桥接拦截器BridgeInterceptor

为请求添加cookie、添加固定的header，比如Host、Content-Length、Content-Type、User-Agent等等，然后保存响应结果的cookie，如果响应使用gzip压缩过，则还需要进行解压。

* 1. 缓存拦截器CacheInterceptor

如果命中缓存则不会发起网络请求。

* 1. 连接拦截器ConnectInterceptor

内部会维护一个连接池，负责连接复用、创建连接（三次握手等等）、释放连接以及创建连接上的socket流。

* 1. 网络拦截器NetworkInterceptors

用户自定义拦截器，通常用于监控和统计网络链路上传输的数据。

* 1. 请求连接器CallServerInterceptor

在前置准备工作完成后，真正发起了网络请求。

### 10.2.4 缓存机制

Http缓存分为强制缓存和协商缓存两种，通过不同的响应头标识区分。CacheInterceptor第一次拿到响应后根据头信息决定是否缓存。下次请求时判断是否存在本地缓存，是否需要使用对比缓存、封装请求头信息等等。如果缓存失效或者需要对比缓存则发出网络请求，否则使用本地缓存。OKHttp只支持get缓存。

### 10.2.4 ConnectionPool

1. 连接复用
   1. 首先会尝试使用已给请求分配的连接（已分配连接的情况例如重定向时的再次请求，说明上次已经有了连接）；
   2. 若没有已分配的可用连接，就尝试从连接池中 匹配获取。因为此时没有路由信息，所以匹配条件：address一致—— host、port、代理等一致，且匹配的连接可以接受新的请求；
   3. 若从连接池没有获取到，则传routes再次尝试获取，这主要是针对 Http2.0 的一个操作, Http2.0可以复用square.com与square.ca的连接；
   4. 若第二次也没有获取到，就创建RealConnection实例，进行TCP + TLS握手，与服务端建立连接；
   5. 此时为了确保Http2.0连接的多路复用性，会第三次从连接池匹配。因为新建立的连接的握手过程是非线程安全的，所以此时可能连接池新存入了相同的连接。
   6. 第三次若匹配到，就使用已有连接，释放刚刚新建的连接；若未匹配到，则把新连接存入连接池并返回。
2. 连接管理

连接管理分成两个步骤，一边当我们创建了一个新的连接的时候，我们要把它放进缓存里面；另一边，我们还要来对缓存进行清理。在 ConnectionPool中，当我们向连接池中缓存一个连接的时候，只要调用双端队列的add()方法，将其加入到双端队列即可，而清理连接缓存的操作则交给线程池来定时执行。

从源码可知，首先会对缓存中的连接进行遍历，以寻找一个闲置时间最长的连接，然后根据该连接的闲置时长和最大允许的连接数量等参数来决定是否应该清理该连接。同时注意上面的方法的返回值是一个时间，如果闲置时间最长的连接仍然需要一段时间才能被清理的时候，会返回这段时间的时间差，然后会在这段时间之后再次对连接池进行清理。

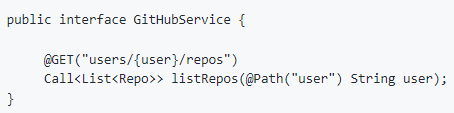
## 10.3 Retrofit

### 10.3.1 Retrofit解决的OKHttp的问题

1. 网络请求的接口配置繁琐；
2. 需要手动对responsbody进行解析；
3. 无法适配自动进行线程的切换；
4. 容易陷入嵌套网络请求的回调陷阱；

### 10.3.2 基本使用

1. 定义API，用于描述请求



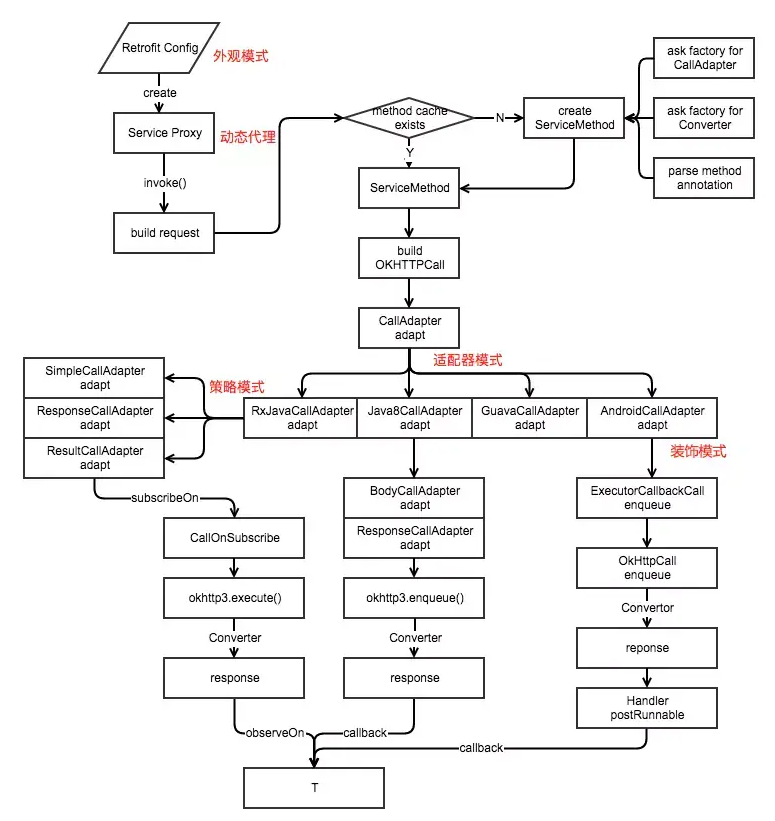
1. 创建JavaBean对象
2. 创建Retrofit并生成API的实现



1. 调用API方法，生成Call，执行请求



### 10.3.3 整体流程



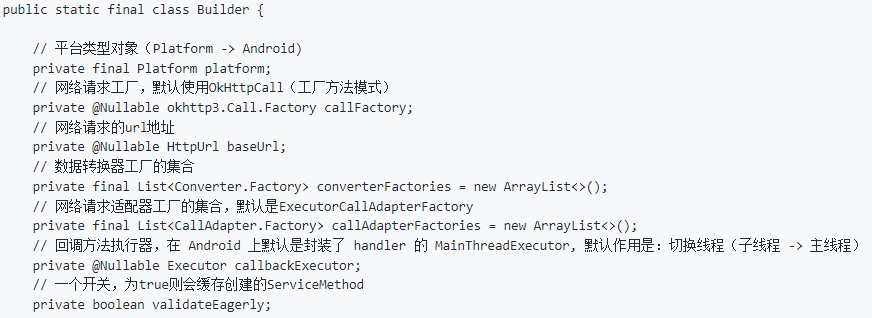
### 10.3.4 Retrofit构建过程

1. 核心对象

首先Retrofit中有一个全局变量非常关键，在V2.5之前的版本，使用的是LinkedHashMap()，它是一个网络请求配置对象，是由网络请求接口中方法注解进行解析后得到的。



Retrofit使用了建造者模式通过内部类Builder类建立一个Retrofit实例，如下：



1. Builder内部构造



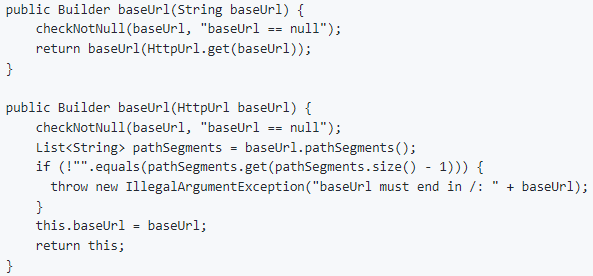




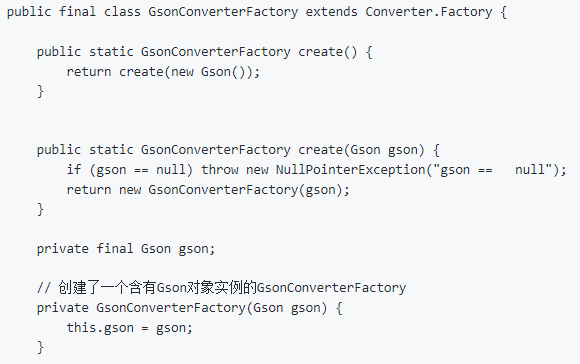
Builder内部构造时设置了默认Platform、callAdapterFactories和callbackExecutor。

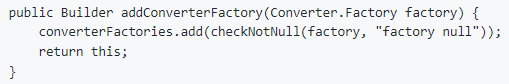
1. 添加baseUrl

将String类型的url转换为OkHttp的HttpUrl过程如下：



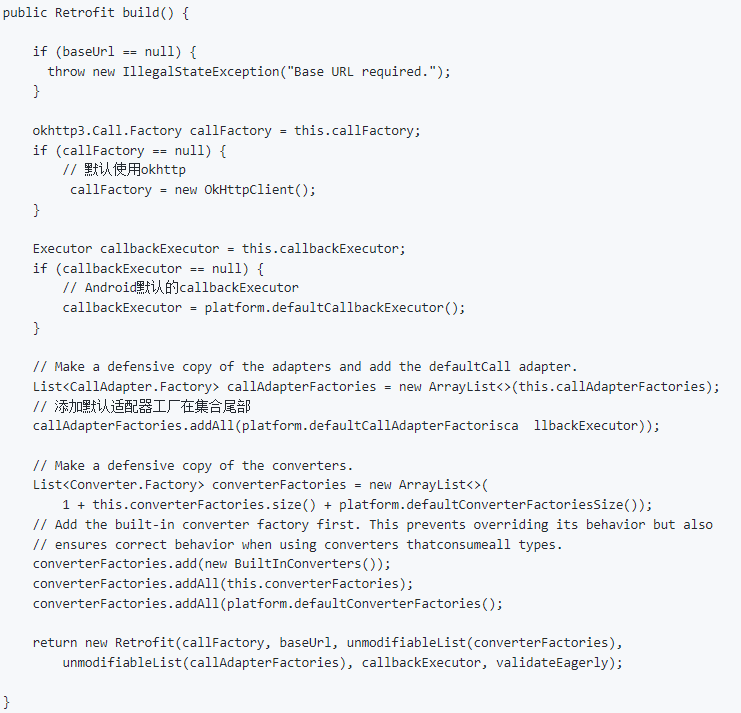
1. 添加GsonConverterFactory





将一个含有Gson对象实例的GsonConverterFactory放入到了数据转换器工厂converterFactories里。

1. build过程



最终在Builder类中6大核心对象都配置到Retrofit对象中了。

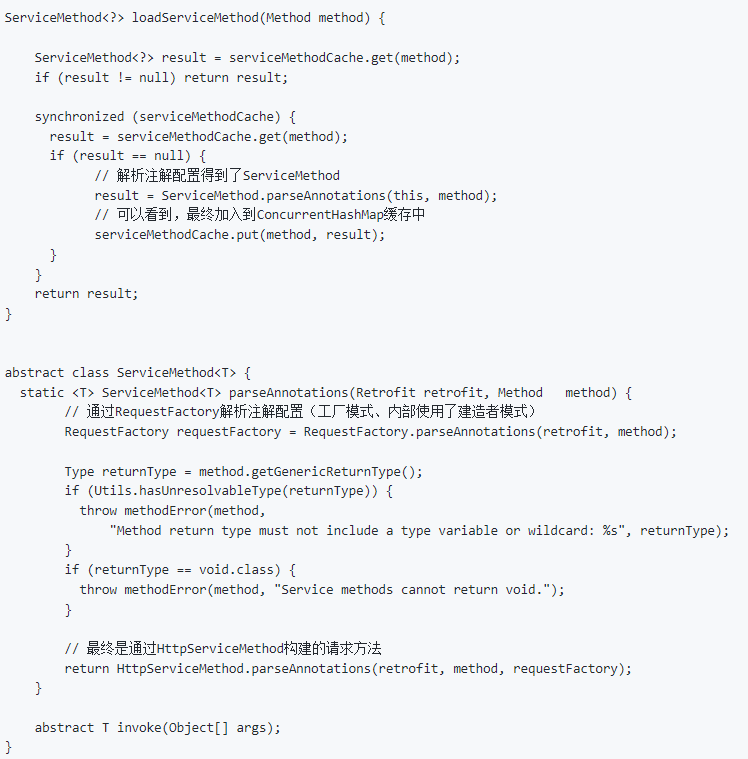
### 10.3.5 创建网络请求接口实例过程

1. retrofit.create()

retrofit.create()使用了外观模式和代理模式创建了网络请求的接口实例，create方法如下：

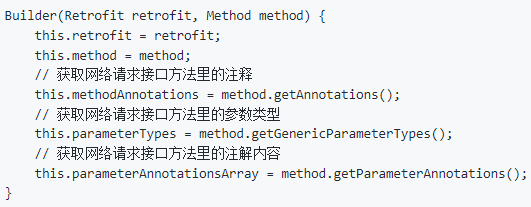


1. loadServiceMethod的内部流程：



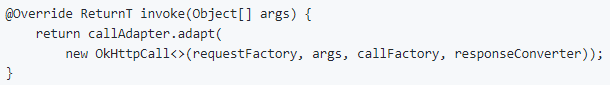
1. 请求构造核心流程

根据RequestFactory#Builder构造方法和parseAnnotations方法的源码，可知的它的作用就是用来解析注解配置的。





执行HttpServiceMethod的invoke方法：



最终在adapt中创建了一个ExecutorCallbackCall对象，它是一个装饰者，而在它内部真正去执行网络请求的还是OkHttpCall。

### 10.3.6 生成Call并执行请求过程

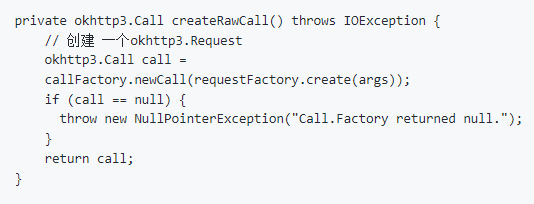
1. service.<>

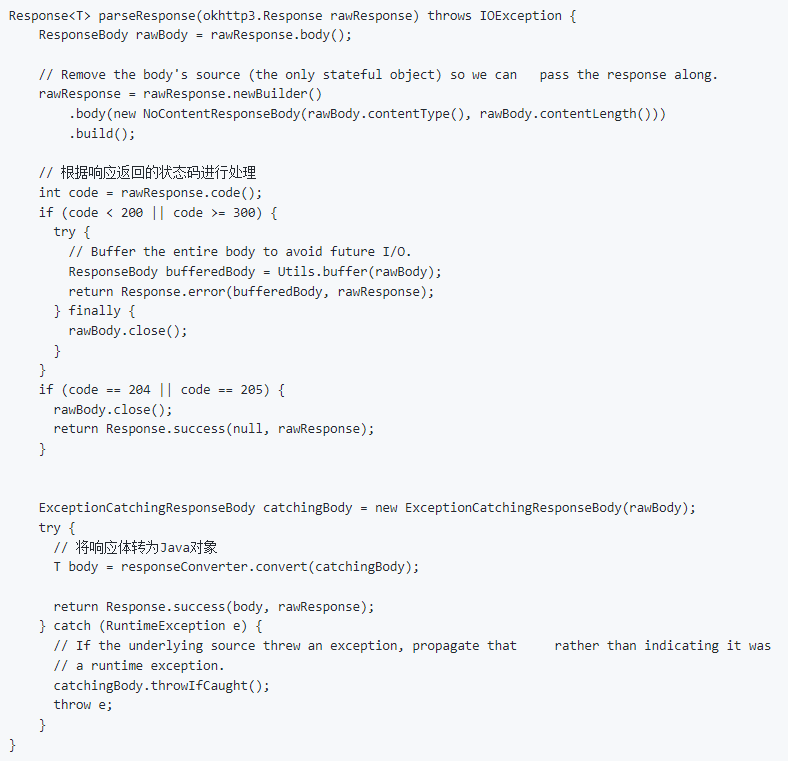


service对象是动态代理对象Proxy.newProxyInstance()，当调用getCall()时会被它拦截，然后调用自身的InvocationHandler#invoke()，得到最终的Call对象。

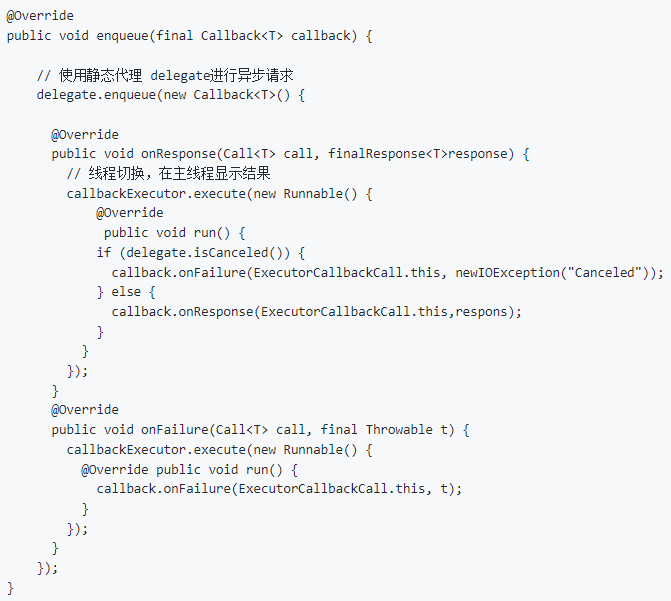
1. 同步执行流程call.execute()



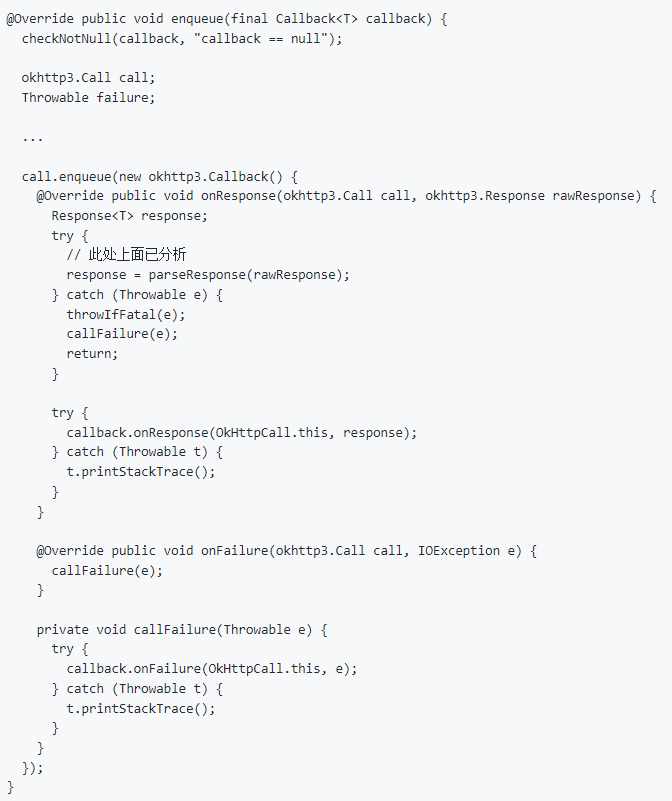




1. 异步请求流程call.enqueue()



delegate.enqueue内部流程：



### 10.3.7 Retrofit中的设计模式

建造者模式，工厂模式，外观模式，代理模式，装饰器模式，适配器模式，策略模式。

## 10.4 Glide

### 10.4.1

### 10.4.2

### 10.4.3

### 10.4.4

## 10.5 GreenDao

### 10.5.1 配置

### 10.5.2 使用方法

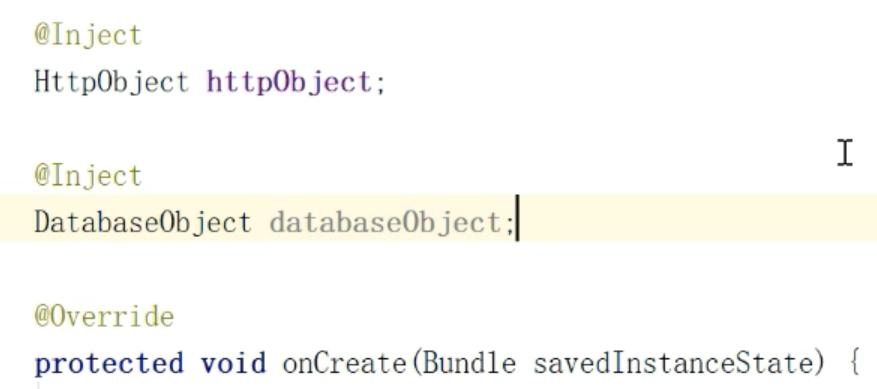
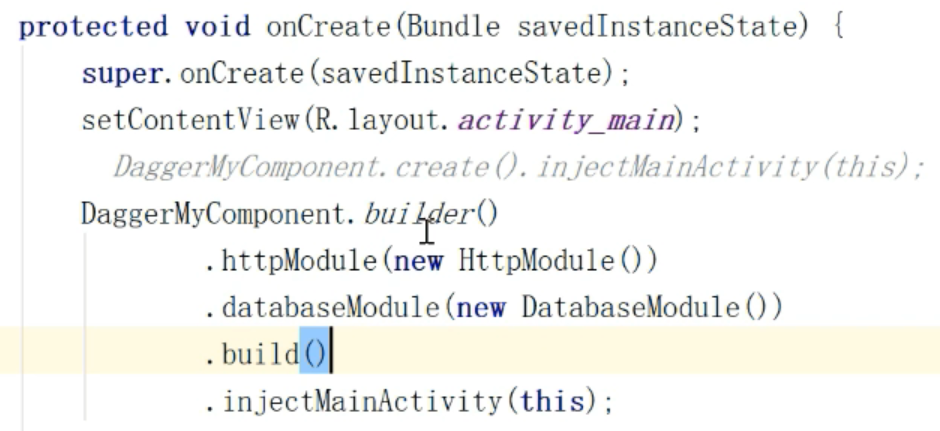
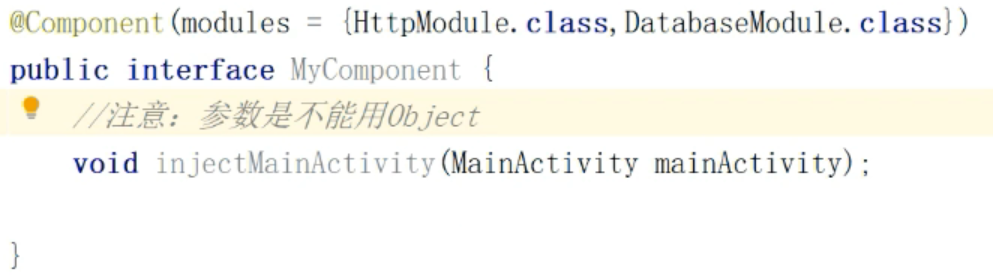
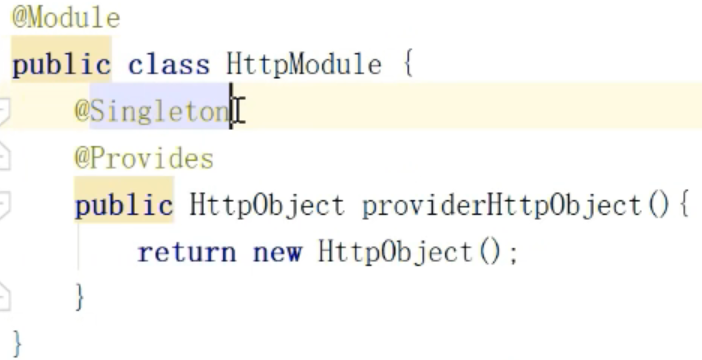
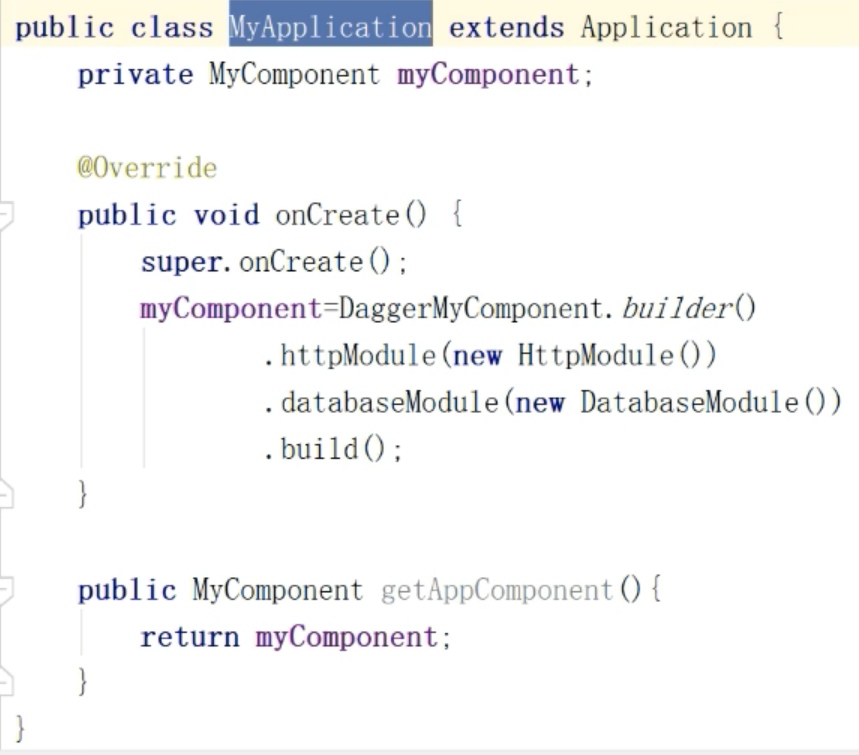
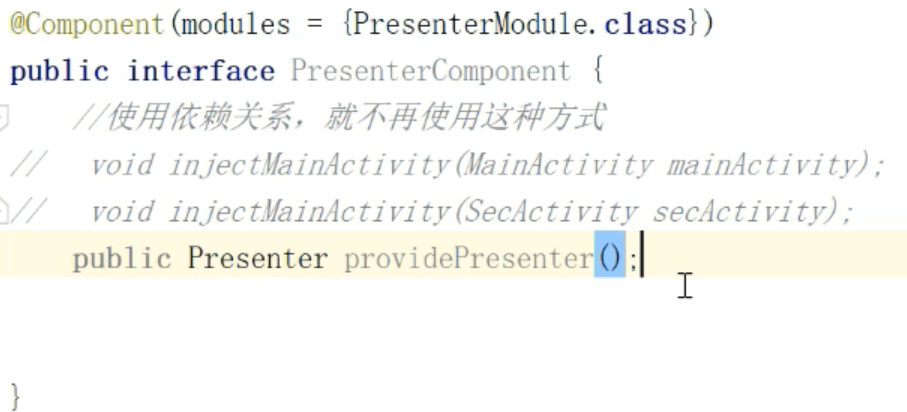
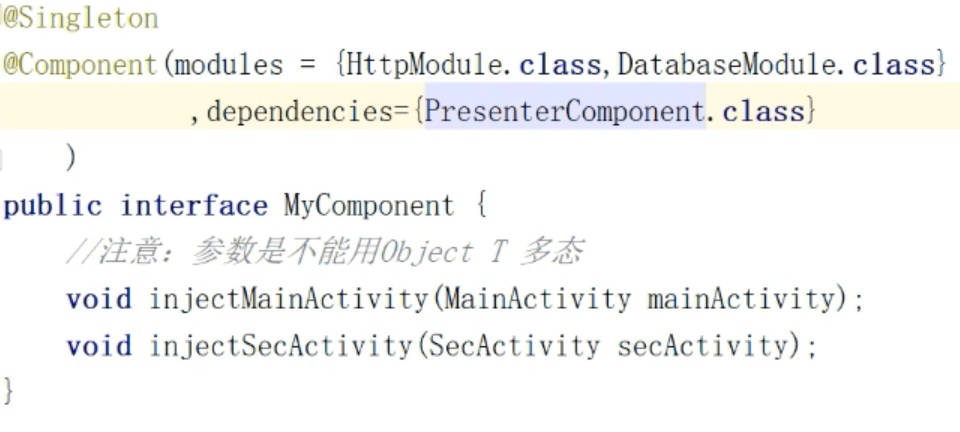
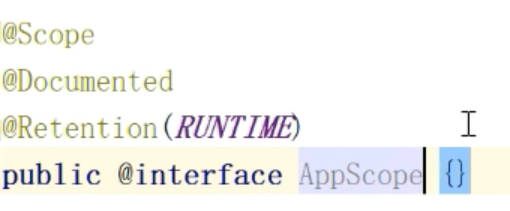
1. 新建实体类
2. 编写DaoManager类，单例，创建，DaoSession，关闭
3. 编写DBHelper类，处理增删改查

## 10.6 Dagger2

Dagger2起源于Dagger，是一款基于Java注解来实现的完全在编译阶段完成依赖注入的开源库，主要用于模块间解耦、提高代码的健壮性和可维护性。Dagger2在编译阶段通过apt利用Java注解自动生成Java代码，然后结合手写的代码来自动帮我们完成依赖注入的工作。

### 10.6.1 使用方法

<https://www.jianshu.com/p/39d1df6c877d>

1. Inject
2. Module
3. Component
4. 单例
5. 全局单例
6. 多Component依赖
7. scope

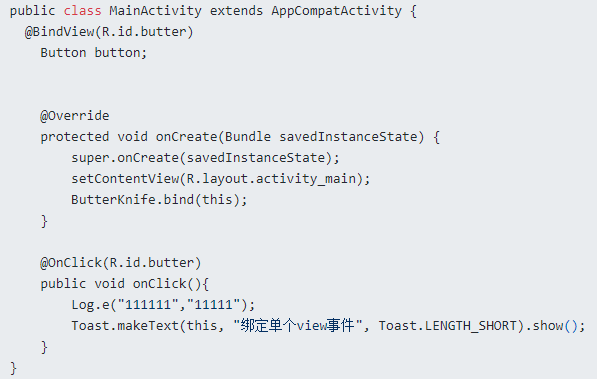
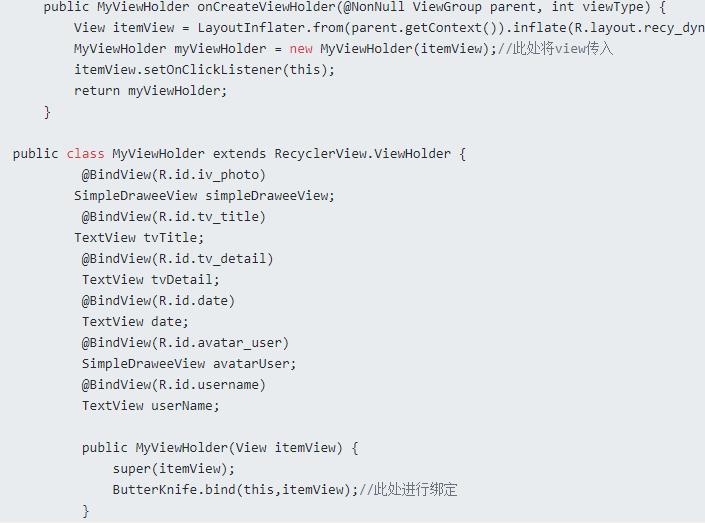
### 10.6.2 实现原理

<https://www.jianshu.com/p/eef7fa8136e7>

## 10.7 ButterKnife

### 10.7.1 配置

### 10.7.2基本使用

1. 在Activity中绑定ButterKnife
2. 在Fragment中绑定ButterKnife
3. 在Adapter的ViewHolder中绑定ButterKnife

# 十一、性能优化

## 11.1

# 十二、JNI

## 12.1

# 十三、音视频

## 13.1