# Android方面

1. **Okhttp源码：**
2. 同步请求：执行call.excute()，把任务放到同步请求队列中，**任务执行完成后**，就把任务从同步请求队列中移除。（最后执行空闲回调）

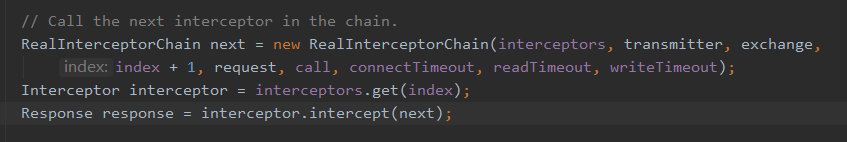


1. 异步请求：执行call.enqueue()，判断任务数，小于最大请求数就放到运行中的异步请求队列中，否则放到等待中的异步请求中；任务在AsyncCall中执行，**任务执行完成后**，就把任务从异步请求队列中移除。



1. **拦截器（Interceptor）**





构建责任链之后，RealInterceptorChain中再创建一个新的责任链，然后责任链的索引加1，再调用intercept()方法，因为拦截器中依然有proceed()，因此，拦截器会一直向下传；执行完之后，再从下到上返回Response。

* 1. **重试与重定向拦截器（RetryAndFollowUpInterceptor）**





（1）发生 Route 或 IO 异常，则进行重试；

（2）如果出现以下情况则不会重试：

·客户端配置了出错不再重试

·无法再次发送 request body

·发生 ProtocolException（协议异常）、InterruptedIOException（中断异常）、SSLHandshakeException（SSL握手异常）、SSLPeerUnverifiedException（SSL握手未授权异常）中的任意一个异常

·没有更多的路线可供尝试

1. 检查是否需要重定向，如果不需要则返回之前的 response，需要则进行重定向，继续进行循环。

**3.2 桥拦截器（BridgeInterceptor）**

（1）添加一些默认的请求头；

（2）解析服务器返回的 header，进行 gzip 解压。

**3.3 缓存拦截器（CacheInterceptor）**

（1）通过 Request 得到缓存；

（2）通过缓存策略获取是使用缓存还是使用网络请求，都不使用直接返回失败。如果使用缓存，则直接返回缓存。如果使用网络，则执行下一个拦截器；

（3）对请求结果进行缓存；

**3.4 连接拦截器（ConnectInterceptor）**

主要用来打开与目标服务器的连接，然后继续执行下一个拦截器。

findConnection()方法流程：

（1）判断当前连接是否可用，如果不可用，尝试从连接池中获取可用连接；如果获取的连接不可用，切换路由再次获取可用连接；再不可用，只能重新创建新的连接；

（2）进行 TCP 和 TLS 握手；

（3）最后将新创建的连接放进连接池中；

result.connect()方法流程：

（1）okhttp 底层是通过 socket 进行连接的；

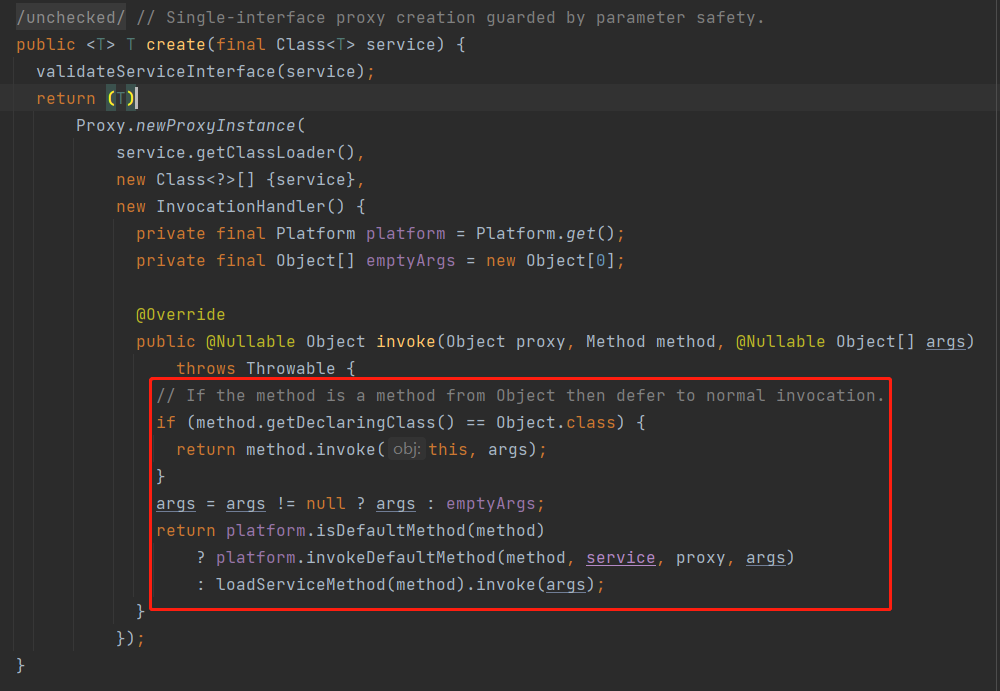
**3.5 服务器请求拦截器（CallServerInterceptor）**

1. **Retrofit源码**

retrofit.build() 配置参数，在create()中使用；

Retrofit.create()：

动态代理：



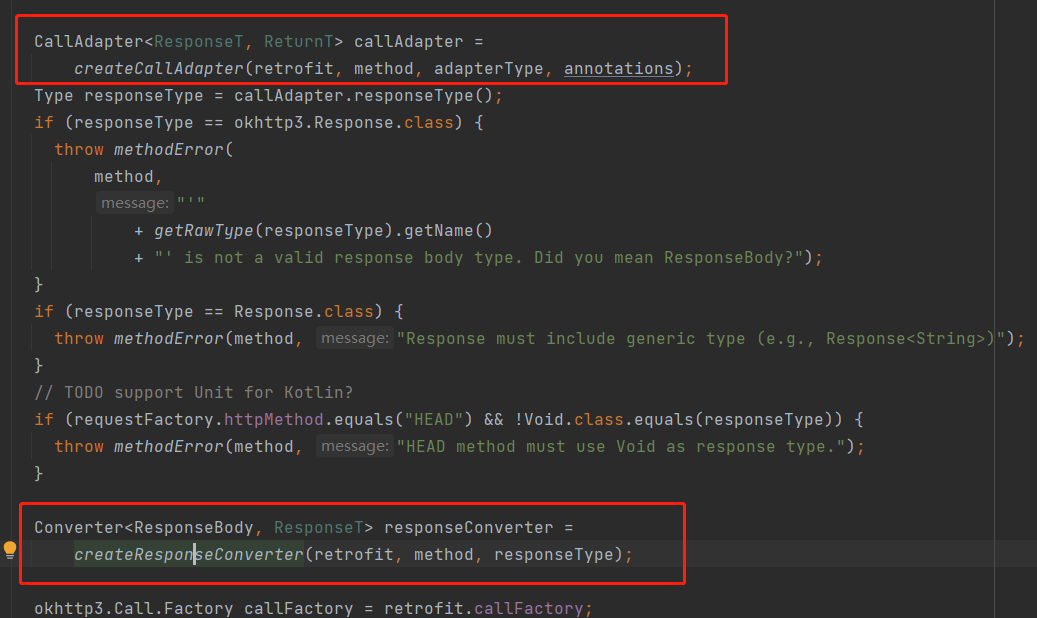
利用动态代理，当调用service.xxx()时，就会执行invoke方法里面的逻辑。

动态代理：



流程：

1. 判断平台；
2. 通过HttpServiceMethod的parseAnnotitions解析请求参数；
3. 执行CallAdatper和Converter；



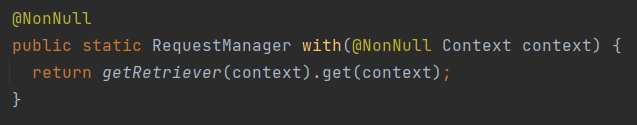
1. 通过RxJava3CallAdapterFactory执行OkHttp请求（Retrofit本身不发起网络请求）

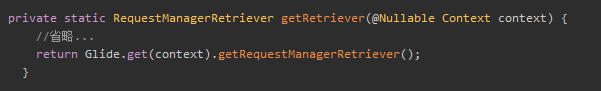
总结：

* 使用构建模式创建Retrofit实例对象
* 使用动态代理模式，通过Retrofit实例对象的create方法动态创建网络请求接口的代理类
* 使用代理类调用方法发起网络请求时，会通过CallAdapterFactory创建的CallAdapter对象的adapt方法调用OkHttpCall的enqueue方法发起网络请求
* OkHttpCall的enqueue方法先根据注解、请求参数等信息构建出OkHttpClient的网络请求Call对象，接着使用Call对象发起网络请求
* 网络请求回来时，使用ServiceMethod的toResponse方法中调用响应内容适配器的具体实现类的convert发放进行解析

Retrofit只是对网络请求进行封装的一个框架，它本身并不发起网络请求，为的是方便我们使用、优化我们网络请求的代码、对网络请求进行扩展（可以自定义请求适配器、响应内容解析器等）

1. **Glide源码**
2. **Glide.with(context)**



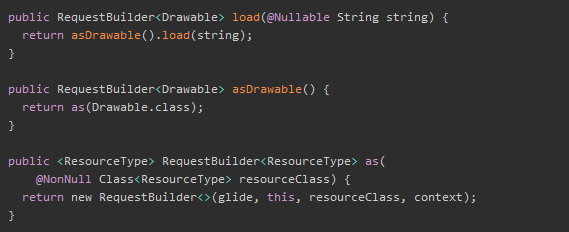


Glide.get(context)实现glide的初始化；

getRequestManagerRetriever()是用来感知组件的生命周期的；

getRetriever(context).get(context)根据不同生命周期对象创建RequestManagerFragment,包括context,activity,fragment,view，用于感知各大组件的生命周期。

1. **Glide.load(url)**



主要是创建一个RequestBuilder，Glide里面的load,error,placeholder,fitCenter等都是RequestBuilder里面的方法。

1. **RequestBuilder**
2. load、transition、apply、error、listener都为保存配置的数据，用于调用into(imageView)后生效；
3. Into(imageView)：

处理图片的加载，包括缓存策略

总结：

· 运行期通过GlideApp的with作为入口返回RequestManager

· 使用RequestManager创建一个图片加载请求构造器RequestBuilder

· 通过RequestBuilder设置图片加载和解析时的一些配置信息：比如playholder、error、apply、transform、listener等

· 通过RequestBuilder的into构建具体图片加载请求对象默认是SingleRequest

· SingleRequest等待ImageView测量得到具体大小之后，在通过图片加载引擎Engine来加载图片

· Engine图片加载引擎根据四级缓存策略：优先从获取资源缓存列表加载；接着从缓存列表加载；再从磁盘缓存加载；最后是从网络加载

· 最后加载完成先回调RequestListener通知图片加载完成；再回调TargetListener通知图片加载完成；最后才回调Target(ImageView的包裹类)将图片设置给ImageView

1. **Glide的生命周期绑定**

· 如果是在子线程，这Glide使用的RequestManager与Application一样的生命周期

· 如果是fragment、activity,则使用FragmentManagerFragment与RequestManager进行关联，通过FragmentManagerFragment的生命周期变化来调度RequestManager对图片加载请求Request采取暂停、重新开始、停止等操作。  
问题：Glide的get操作有哪些优化点？

· 在UI线程中调用，可以避免RequestManager生命周期与Application的一制

· get尽量传递fragment或者activity，这样可以减少通过view找到具体的fragment或者activity的步骤

1. ****ImageViewTarget****

· 通过ViewTreeObserver实现View的大小测量，测量到大小之后回到监听者的onSizeReady告知view的大小已经测量ok

· 通过监听View与Window的绑定关系发起加载图片的请求或者取消加载图片

· 设置加载中的显示图片

· 设置加载失败时显示的图片

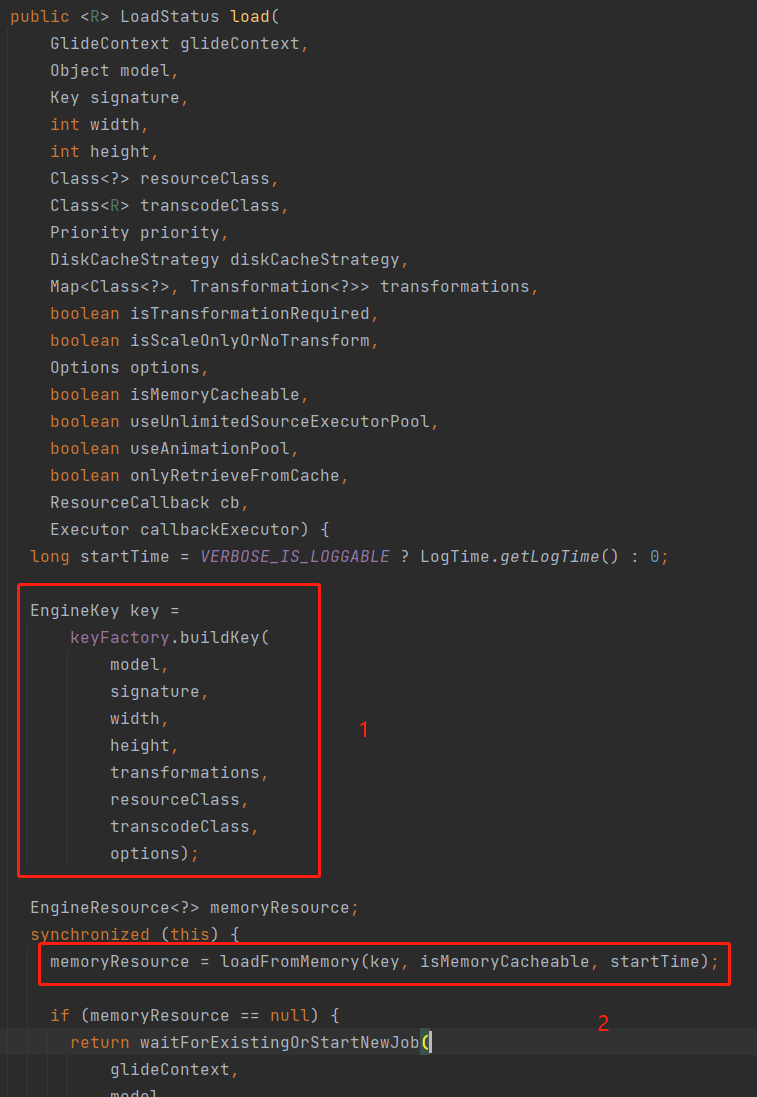
· 设置加载成功时的图片、动效

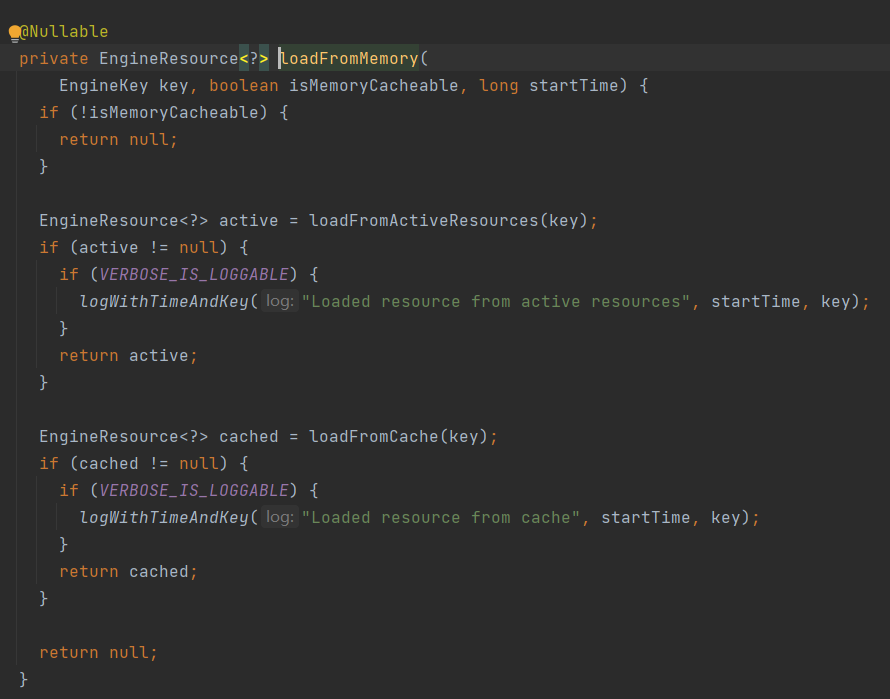
1. **Glide缓存机制**

glide缓存机制分为3级缓存机制，其顺序为WeakReference->LruCache->DiskLrucache->网络。

****内存缓存**的主要作用是防止应用重复将图片数据读取到内存当中；而**硬盘缓存**的主要作用是防止应用重复从网络或其他地方下载和读取数据。**

**（1）内存缓存（弱引用+LruCache）**





1. 获取key
2. 判断是否有活跃缓存，没有则去LruCache读取缓存；
3. 通过弱引用的hashmap来存储资源，Key是缓存key，ResourceWeakReference代表资源，它继承WeakReference。首先从弱引用的map获取图片资源，然后通过弱引用的get()方法获取最终需要的对象；



（4）**通过lrucache获取图片资源，如果获取到的话就会从LruCache中删除这张图片，然后会调用**acquire()**方法和**activate()**方法,其中**activate()**是把取到的数据会存到弱引用中，说白了就是把图片从LruCache转移到弱引用**。

这个acquired变量是用来记录图片被引用的次数，调用acquire()方法会让变量加1，调用release()方法会让变量减1。当调用loadFromActiveResources()、loadFromCache()、EngineJob#handleResultOnMainThread()获取图片的时候都会执行acquire()方法；当暂停请求或者加载完毕或者清除资源时会调用release()方法。  
  
注意：从弱引用取缓存，拿到的话，引用计数+1；从LruCache中拿缓存，拿到的话，引用计数也是+1，同时把LruCache缓存转移到弱应用缓存池中；从EngineJob去加载图片，拿到的话，引用计数也是+1，会把图片放到弱引用。反过来，一旦没有地方正在使用这个资源，就会将其从弱引用中转移到LruCache缓存池中。这也说明了正在使用中的图片使用弱引用来进行缓存，暂时不用的图片使用LruCache来进行缓存的功能。

1. **磁盘缓存**

**Glide5大磁盘缓存策略**  
DiskCacheStrategy.DATA: 只缓存原始图片；  
DiskCacheStrategy.RESOURCE:只缓存转换过后的图片；  
DiskCacheStrategy.ALL:既缓存原始图片，也缓存转换过后的图片；对于远程图片，缓存 DATA和 RESOURCE；对于本地图片，只缓存 RESOURCE；  
DiskCacheStrategy.NONE：不缓存任何内容；  
DiskCacheStrategy.AUTOMATIC：默认策略，尝试对本地和远程图片使用最佳的策略。当下载网络图片时，使用DATA；对于本地图片，使用RESOURCE

磁盘缓存就是通过DiskLruCache实现的，根据缓存策略的不同会获取到不同类型的缓存图片。它的逻辑是：先从转换后的缓存中取；没有的话再从原始的（没有转换过的）缓存中拿数据；再没有的话就从网络加载图片数据，获取到数据之后，**再依次缓存到磁盘和弱引用**。

总结：

Glide缓存分为弱引用+ LruCache+ DiskLruCache，其中读取数据的顺序是：弱引用 > LruCache > DiskLruCache>网络；写入缓存的顺序是：网络 --> DiskLruCache--> 弱引用-->LruCache

内存缓存分为弱引用的和 LruCache ，其中正在使用的图片使用弱引用缓存，暂时不使用的图片用 LruCache缓存，这一点是通过 图片引用计数器（acquired变量）来实现的，详情可以看内存缓存的小结。

磁盘缓存就是通过DiskLruCache实现的，根据缓存策略的不同会获取到不同类型的缓存图片。它的逻辑是：先从转换后的缓存中取；没有的话再从原始的（没有转换过的）缓存中拿数据；再没有的话就从网络加载图片数据，获取到数据之后，再依次缓存到磁盘和弱引用。

1. **LruCache原理**

LruCache 是个泛型类，主要原理是：把最近使用的对象用强引用存储在 LinkedHashMap 中，当缓存满时，把最近最少使用的对象从内存中移除，并提供 get/put 方法完成缓存的获取和添加LruCache 是线程安全的，因为使用了 synchronized 关键字。

当调用 put()方法，将元素加到链表头，如果链表中没有该元素，大小不变，如果没有，需调用 trimToSize 方法判断是否超过最大缓存量，trimToSize()方法中有一个 while(true)死循环，如果缓存大小大于最大的缓存值,会不断删除 LinkedHashMap 中队尾的元素，即最少访问的，直到缓存大小小于最大缓存值。当调用 LruCache 的 get 方法时，LinkedHashMap 会调用recordAccess 方法将此元素加到链表头部

1. **其它**

对于一般App来说，Glide完全够用，而对于图片需求比较大的App，为了防止加载大量图片导致OOM，Fresco 会更合适一些。并不是说用Glide会导致OOM，Glide默认用的内存缓存是LruCache，内存不会一直往上涨。

假如让你自己写个图片加载框架，你会考虑哪些问题：

· 异步加载：线程池

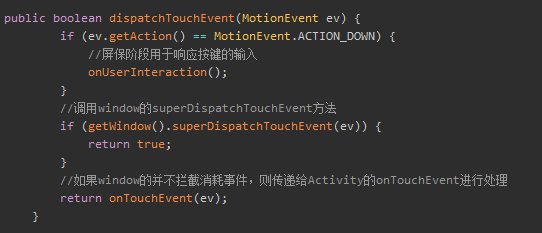
· 切换线程：Handler，没有争议吧

· 缓存：LruCache、DiskLruCache

· 防止OOM：软引用、LruCache、图片压缩、Bitmap像素存储位置

· 内存泄露：注意ImageView的正确引用，生命周期管理

1. **触摸事件分发机制**
2. Activity事件分发



Activity dispatchTouchEvent的核心思想是：将事件分发给window，如果window拦截消耗事件，则完成事件的分发；如果window不拦截消耗事件，则将事件传递给Activity的onTouchEvent进行处理。

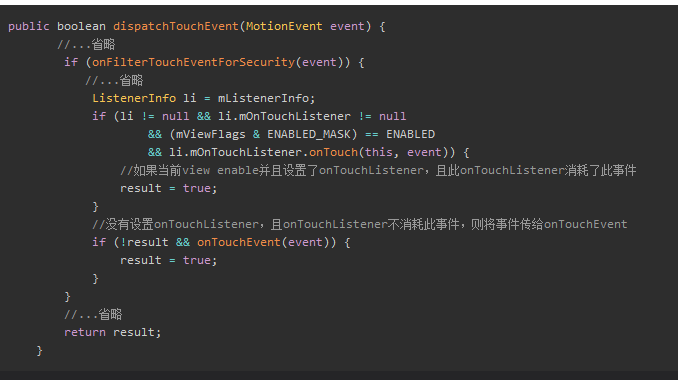
1. ViewGroup事件分发



如果没有子View，则调用ViewGroup的父类View来实现dispatchTouchEvent。

ViewGroup进行进行事件分发时，其核心思想是：  
1、 调用onInterceptTouchEvent判断自己是否要拦截消耗此事件  
2、 如果自己不拦截消耗事件，则遍历子view列表，根据事件坐标找到对应的子view，然后将事件分发给此view，如果此子view是ViewGroup，那么又走ViewGroup的dispatchTouchEvent逻辑；如果此子view是View，那么进入第四阶段  
3、如果自己拦截消耗事件那么调用父类的dispatchTouchEvent，其父类是View，而View的事件分发dispathcTouchEvent请看第四阶段

1. View事件分发



View的dispatchTouchEvent优先将事件传递给onTouchListener，如果onTouchListener消耗此事件，则直接放回true；如果没有设置onTouchListener或者onTouchListener没有消耗此事件，则将事件传递给onTouchEvent

onTouchEvent的核心工作是：  
1、先判断view是否enable，如果不enable则不响应事件，并返回clickable的值  
2、如果设置了touch的代理，并且touch代理消耗了此事件则返回true代表消耗了事件  
3、如果view不可点击，则返回false，代表不消耗此事件  
4、如果view可点击，则返回true，代表消耗此事件。如果设置了长按事件，则在按下事件中延迟500ms响应长按事件；响应点击事件是在松开事件时响应的

1. 如何拦截触摸事件？怎么不响应触摸事件？

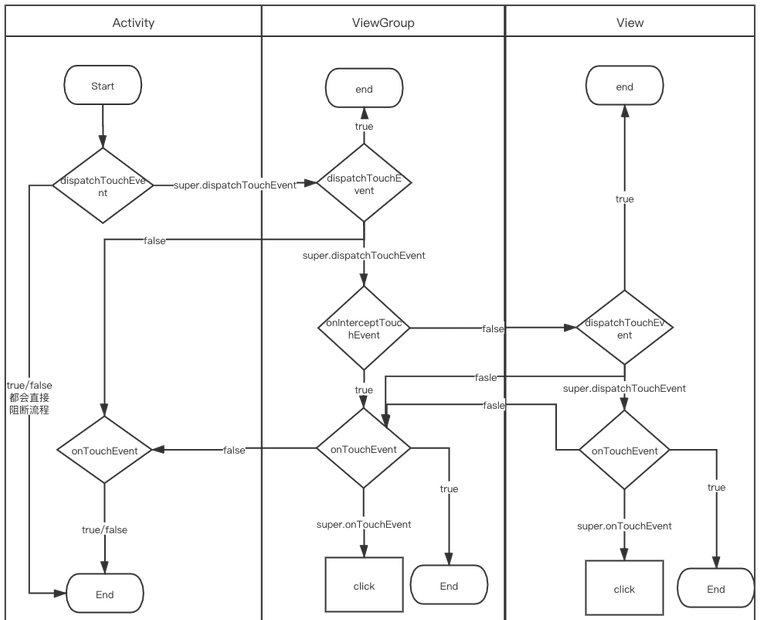
要响应事件，根据View的事件分发流程可以有几种解决方案：  
a、父控件拦截事件，将事件传递给view进行处理  
b、自己拦截事件：

1.可以通过dispatchTouchEvent直接返回true，事件处理再dispatchTouchEvent中处理；

2.onInterceptTouchEvent返回true，并在onTouch或者onTouchEvent中消耗此事件，并返回true

3.不响应最简单的是clickable为false或者enable为false、或者不可见；

1. 嵌套的view，父级要求响应垂直滑动，子级要求响应水平滑动，如何实现？
2. 父级控件拦截事件，滑动时判断水平滚动距离大于垂直滑动距离，则将事件传递给子view响应水平滑动；否则父级控件自己响应垂直滑动  
   b、子级在onInterceptTouchEvent中判断水平滚动距离大于垂直滑动距离，则拦截消耗此事件
3. 流程图



1. **LeakCanary解析**

AppWatcher.manualInstall()在主进程中被自动调用

AppWatcherInstall继承自ContentProvider并在AndroidManifest.xml中注册，利用ContentProvider无需显示初始化

内存泄漏检测主要过程：



Activity内存泄漏检测过程：

(1)注册监听Activity生命周期onDestroy事件

(2)在Activity onDestroy事件回调中创建KeyedWeakReference对象，并关联ReferenceQueue

(3)延时5秒检查目标对象是否回收

(4)未回收则开启服务，dump heap获取内存快照hprof文件

(5)解析hprof文件根据KeyedWeakReference类型过滤找到内存泄漏对象

(6)计算对象到GC roots的最短路径，并合并所有最短路径为一棵树

(7)输出分析结果，并根据分析结果展示到可视化页面

# JAVA方面

1. JAVA锁