面试官你好，我叫郑频文。

之前的工作经历主要是从事android开发。使用的开发语言主要是Java、Kotlin，少量使用C++。

涉及的工作内容主要是项目功能开发、基础架构搭建和性能调优。对于一些常用的开源框架、自定义View、APK打包流程、内存泄漏、IO优化、APK瘦身比较熟悉。

最近一份工作是在美的集团里面负责APP的功能迭代、基础架构搭建和性能调优工作。他们的APP是国内的美的美居，是一款日活百万的智能设备控制APP，主要功能包括设备绑定，设备控制和内容发现模块。用户购买美的产品后可以通过这款APP进行设备的状态查看和功能操作，同时提供了内容发现功能用于推广新产品与用户交互与反馈。性能优化方面主要工作有：内存泄漏排查、APK瘦身、启动优化、gradle 构建优化。

我今天想要面试公司的这个职位，我觉得以我的能力可以胜任这个工作，而且也符合我个人的职业发展方向。

希望可以通过面试进一步了解这个工作的内容。

设计模式

创建模式：建造者模式、工厂模式、抽象工厂模式、单例模式、原型模式

建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。（日志上报工具中有关字段的构建。）

工厂模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。（）

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。（）

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。（配网网络请求参数的全局单例。）

原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。（）

行为模式：观察者模式、策略模式、责任链模式、中介者模式、模版模式；

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。（配网过程中，多种设备监听底层SDK配网进度。）

策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。（不同设备配网流程只有中间个别部分不同，定义配网总流程，然后抽象出其中不同的部分。）

责任链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它或者每个对象都处理过为止。（OKHttp网络请求的流程就是一个责任链模式、路由组建也可以在请求的过程中执行各种拦截操作。）

中介者模式：用一个中介对象来封装一系列的对象交互。中介者使各个对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。（设备管理中涉及设备列表管理的操作，全选、反选、取消等操作。）

模版模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。（与策略模式有点相同。）

结构模式：适配器模式、装饰者模式、代理模式、享元模式

适配器模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。（Java中的文件流API架构就是基于适配器模式设计。）

装饰者模式：动态地给一个对象添加一些额外的职责。就增加功能来说，相比生成子类更为灵活。（Java中的文件压缩与缓冲流设计就是基于装饰者模式设计。）

代理模式：为其他对象提供一种代理以控制对这个对象的访问。（Retrofit基于动态代理对网络请求进行拦截。）

享元模式：运用共享技术有效地支持大量细粒度的对象。（OkHttp中将链接进行缓存，相同请求使用缓存的链接。）

常用的开源框架：ARouter、okhttp、MMKV、okio、EventBus、Glide、Retrofit

ARouter：路由工具

0、ARouter使用了APK打包过程重点APT功能，在这个流程中会遍历所有的Java文件进行语法解析，包括注解的解析。通过获取对应Java文件的注解来判断是否是路由标志类。将路由标志类的路由信息进行解析，获取对应的路由信息，通过类生成工具将对应的类生成到对应的包下。

1. 调用跳转API的时候，首先根据跳转的目标路径解析出path和group（一定要以反斜杠开头，至少有两个反斜杠，第二个反斜杠开始是group），构造PostCard实例。可以通过PathReplaceService 这个类对路径进行预处理。
2. 然后进行导航逻辑处理，如果有自定义PretreatmentService这个类，就会走里面的预处理逻辑判断是否需要拦截，没有就走默认逻辑。在LogisticsCenter中根据path和group在WireHouse.routes获取具体路径信息RouteMeta完善PostCard。如果不是绿色通道，然后有拦截器InterceptorServiceImpl 则执行拦截器里的逻辑。然后继续导航。
3. 补全Postcard信息的操作主要在LogisticsCenter实现。路由信息一开始保存在Warehouse.routes里面，然后通过IRouteGroup.loadInto加载出来，IRouteGroup 的信息都保存在Warehouse.groupsIndex中。有两种方式可以初始化路由信息。如果有ARouter-auto-register 插件，则通过在APK打包过程中获取并解析有关注解，动态生成代码，在初始化的时候加载这些路由信息代码。如果没有插件，则动态扫描所有dex文件，获取所有包名com.alibaba.android.ARouter.routes的文件，初始化路由信息。初始化的过程就是填充Warehouse中的各种路由表。初始化结束后会将路由信息保存在SP中，如果APP有升级才会从新检测一次。

3.1 如果使用了ARouter-auto-register 插件，会在插件初始化的时候获取对应模块的moduleName 参数，所以我们需要在对应模块的gradle文件进行属性AROUTER\_MODULE\_NAME配置。插件中会拿到所有@Route注解标注的类，在com.alibaba.android.arouter.routes包下生成ARouter$$Group$$[GroupName]包含路由组的所有信息。然后生成ARouter$$Root$$[moduleName]包含所有组的信息。

3.2 在获取到了所有组信息之后，需要将这些信息插入到LogisticsCenter实现类中，进行路由信息的完善。ARouter主要是通过注册Transform来进行字节码的插入。让ASM扫描对应的jar文件和class文件。并将对应route包下扫描到的类保存起来，

1. 针对各种类型的路由进行处理。如果是Activity，则新建Intent，通过postcard信息，完善Intent走context.startActivity或者context.startActivityForResult。如果是Provider，从postcard获取实例返回。如果是broadcast、contentProvider、fragment则通过routeMeta.getConstructor().newInstance()获取实例并设置参数，然后返回。
2. ARouter需要在Application进行初始化，这一步主要是将生成的路由信息加载进内存。

Glide：图片加载工具

with()：with传入的context决定了Glide的生命周期，如果传入的是activity，那么当activity被销毁后，Glide就也会被销毁。如果传入的ApplicationContext，那么它的生命周期就跟APP相同。如果传入的是activity，glide会在Activity上面创建一个无UI的Fragment（RequestManagerFragment），这样可以把glide加载的图片与生命周期同步。不需要把对应的回调暴露在开发者管理。RequestManagerFragment.class中持有一个ActivityFragmentLifecycle，在Fragment进入关键生命周期时会主动通知lifecycle执行相关方法。ActivityFragmentLifecycle.class中持有一个lifecycleListeners，在Fragment进入关键生命周期时Lifecycle会通知他的所有Listener。然后将通知传递给RequestManger.class，执行关键生命周期中处理加载任务。

1. Glide为什么不能在子线程中with？子线程中不会去添加 生命周期管理机制，主线程才会添加一个 空白的Fragment去监听 Activity Fragment的变化。
2. getRetriever(...).get(...)做了什么事情？不论传入Activity、FragmentActivity、Fragment最终都会调用supportFragmentGet()方法，而这两个方法最终流程都是一致的就是那就是会向当前的Activity当中添加一个隐藏的Fragment。
3. 为什么要添加一个隐藏的Fragment呢？因为Glide需要知道加载的生命周期。很简单的一个道理，如果你在某个Activity上正在加载着一张图片，结果图片还没加载出来，这时候Activity被用户关掉了，那么图片就应该取消加载，可是Glide并不知道Activity的生命周期，怎么办呢？
4. 于是Glide就使用了添加隐藏Fragment的这种小技巧，因为Fragment的生命周期和Activity是同步的，如果Activity被销毁了，Fragment是可以监听到的，这样Glide就可以捕获这个事件并停止图片加载了。
5. 既然有了Application，为什么不用registerActivityLifecycleCallbacks而是用隐藏的Fragment？registerActivityLifecycleCallbacks是可以实现，并且我的小伙伴在自己的某些工程中也在使用，但是个人理解是这样的：
6. registerActivityLifecycleCallbacks监控所有的Activity生命周期，然而当你使用Glide加载图片时，并不是所有的Activity都会用到Glide加载图片（大多数情况），所以呢，使用registerActivityLifecycleCallbacks存在资源浪费的现象。不仅如此，你监控了所有的activity怎么和Glide想要监控的Activity关联到一块去，虽然可以实现，但是这个办法真心不实用。

load()：RequestManager的load()方法中，fromString()方法会返回一个DrawableTypeRequest对象，接下来会调用这个对象的load()方法，把图片的URL地址传进去。DrawableTypeRequest的父类是DrawableRequestBuilder，DrawableRequestBuilder中有很多个方法，这些方法其实就是Glide绝大多数的API了。比如说placeholder()方法、error()方法、diskCacheStrategy()方法、override()方法等。

into()：into()方法的具体逻辑都是在DrawableRequestBuilder的父类当中了。代码先是调用了glide.buildImageViewTarget()方法，这个方法会构建出一个Target对象，Target对象则是用来最终展示图片用的。这里其实又是调用了ImageViewTargetFactory的buildTarget()方法，也就是说，通过glide.buildImageViewTarget()方法，我们构建出了一个GlideDrawableImageViewTarget对象。那现在回到刚才into()方法的最后一行，可以看到，这里又将这个参数传入到了GenericRequestBuilder另一个接收Target对象的into()方法当中了。调用buildRequest()方法构建出了一个Request对象，还有使用runRequest()方法来执行这个Request。调用了obtainRequest()方法来获取一个Request对象，而obtainRequest()方法中又去调用了GenericRequest的obtain()方法。runRequest()方法先判断Glide当前是不是处理暂停状态，如果不是暂停状态就调用Request的begin()方法来执行Request，否则的话就先将Request添加到待执行队列里面，等暂停状态解除了之后再执行。执行过程中，调用了一个target.onLoadStarted()方法，并传入了一个loading占位图，在也就说，在图片请求开始之前，会先使用这张占位图代替最终的图片显示。如果model等于null，model也就是我们在第二步load()方法中传入的图片URL地址，这个时候会调用onException()方法。如果你跟到onException()方法里面去看看，你会发现它最终会调用到一个setErrorPlaceholder()当中，这个方法中会先去获取一个error的占位图，如果获取不到的话会再去获取一个loading占位图，然后调用target.onLoadFailed()方法并将占位图传入。其实就是将这张error占位图显示到ImageView上而已，因为现在出现了异常，没办法展示正常的图片了。调用target.getSize()方法。请求过程中，这个target.getSize()方法的内部会根据ImageView的layout\_width和layout\_height值做一系列的计算。

glide在发起图片请求的时候会生成一个key来判断是否已经存在相同的请求。将id连同着signature、width、height等等10个参数一起传入到EngineKeyFactory的buildKey()方法当中，从而构建出了一个EngineKey对象，这个EngineKey也就是Glide中的缓存Key了。

EngineKey类的源码大家有兴趣可以自己去看一下，其实主要就是重写了equals()和hashCode()方法，保证只有传入EngineKey的所有参数都相同的情况下才认为是同一个EngineKey对象。

Glide的缓存设计可以说是非常先进的，考虑的场景也很周全。在缓存这一功能上，Glide又将它分成了两个模块，一个是内存缓存，一个是硬盘缓存。

其中内存缓存又包括活动缓存（WeakReference Cache）和 内存缓存（LruCache）。硬盘缓存就是DiskLruCache。

这两个缓存模块的作用各不相同:

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，

硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据。

内存缓存读取：先获取LruResourceCache（Lru算法缓存），然后放入activeResources（是一个弱引用的HashMap）来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

内存缓存写入：回调过来的EngineResource先被put到了activeResources当中，也就是在这里写入的缓存。如果图片正在使用中，也就应该放到activeResources弱引用缓存当中。如果图片不再被使用了，首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

进一步总结：正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

磁盘缓存读取：这个diskCacheStrategy()方法基本上就是Glide硬盘缓存功能的一切，它可以接收四种参数：

DiskCacheStrategy.NONE： 表示不缓存任何内容。

DiskCacheStrategy.SOURCE： 表示只缓存原始图片。

DiskCacheStrategy.RESULT： 表示只缓存转换过后的图片（默认选项）。

DiskCacheStrategy.ALL ： 表示既缓存原始图片，也缓存转换过后的图片。

首先，和内存缓存类似，硬盘缓存的实现也是使用的LruCache算法，而且Google还提供了一个现成的工具类DiskLruCache。

发起请求后会将我们传递的图片地址转换为输入流，获取到输入流之后会将其转换为bitmap，然后通过mainHandler转换到主线程进行图片的展示。

1. glide比Picasso加载速度慢？glide读取数据的步骤：1、读取图片的exif信息；2、获取图片的宽高，同时设置options.inJustDecodeBounds = true，此时图片并没有存在内存中，仅仅获取了一些options；3、根据宽高创建出对应的bitmap；4、在读取图片的信息，判断图片类型；5、根据读出来的方向信息再次修改bitmap。
2. glide比Picasso内存占用少、质量差？glide的缓冲区是循环使用的：1、在解析图片的时候设置了options.inBitmap = recycled；设置了一个复用的bitmap作为返回的结果使用；2、这个bitmap复用了LruBitmapPool 中规格相同的bitmap；3、在计算SimpleSize的时候，SimpleSize不为2的次幂时候会取成最近的2次幂的采样率。
3. glide缓存策略。分为两种：内存和磁盘。glide在创建Request的时候会先判断是否已经存在这个请求，有的话撤销请求，重新创建一个请求。在load方法的时候，会先判断一次memoryCache是否存在对应request产生出来的key对应的文件，有就读出直接返回。这里的处理策略是，先从LruResourceCache读取缓存中的资源，如果有就从LruResourceCache中移除，并且加入到activeResources中。接下来再尝试从activeResources取出资源文件。这么做的好处是什么，首先即使是使用了LruCache最近最少用算法，也无法避免OOM的结果，毕竟加载图片很消耗内存。但是如果把正在使用的资源放在弱引用里面结果就不同了。弱引用相当于打上一个标记，当gc来的时候就会回收掉。一来我正在使用这个资源，即使gc来了，经过分析对象可达性，如果没有使用者也会尝试把这个列表里面的资源全部回收掉。这样就尽量保证了不会出现OOM的情况。如果都没有资源，则从网络尝试获取资源文件。在联网之前会先去磁盘缓存里面查看是否还存在key对应的资源，有就从磁盘里面获取。磁盘缓存分为下面四种策略:source和result两种缓存策略。默认 RESULT(false, true);是只存储经过glide导正和采样之后的图片，而source网上说是没有变形之前的图片，其实还不准确，实际上是没有解析图片之前的从网络获取下来的数据流。当然如果获取不到图片会调用LoadFailed，就会修改Stage状态，尝试这区联网去读流。
4. Glide加载一个100x100的图片，是否会压缩后再加载？放到一个300x300的view上会怎样？Glide会为每个不同尺寸的ImageView缓存一张图片，也就是说不管你的这张图片有没有被加载过，只要ImageView的尺寸不一样，那么GLide就会重新加载一次，这时候，他会在加载ImageView之前从网络上重新下载，然后再缓存。举个例子，如果一个页面的ImageView是300 \* 300像素，而另一个页面中的ImageView是100 \* 100像素，这时候想要让两个ImageView是同一张图片，那么Glide需要下载两次图片，并且缓存两张图片。
5. 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。
6. 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；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；列表滑动加载的问题：加载错乱、队满任务过多问题；

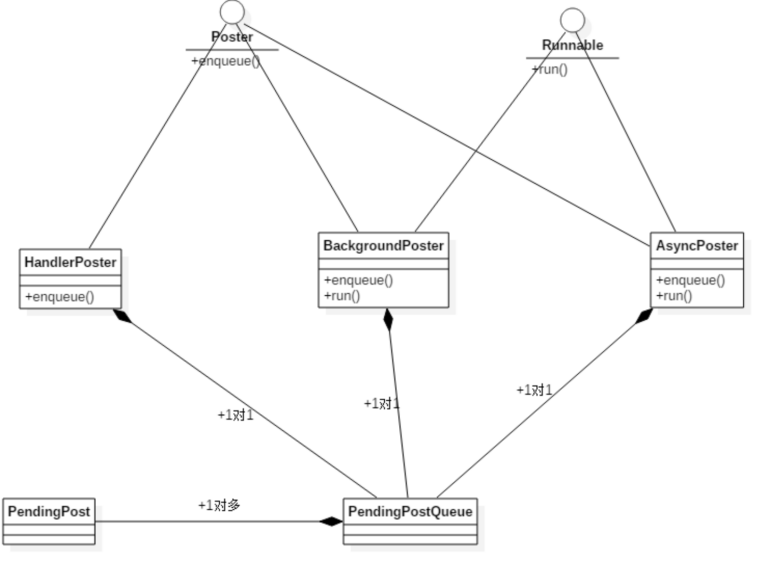
EventBus：消息通知

EventBus本质是基于观察者模式设计的总线事件处理。通过扫描类中加了注解的方法作为被观察者接收器，以及获取方法中的参数类型作为这种接收器接收的事件。在post的时候，筛选出来，反射方法，调用存在于全局的缓存方法信息。

1、EventBus注册监听：EventBus是通过一个全局单例来管理注册信息。首先扫描注册类中标记了@Subscriber的方法，生成包含方法、类名信息的队列保存在全局缓存中。扫描过程中会缓存对应的类名，下次直接从缓冲中获取。所谓的黏着事件是在每一次register的时候，发送一次那些通过postSticky保存在stickyEvents全局缓存中的事件。

2、EventBus发送事件：通过ThreadLocal保存当前线程的副本数据，主要包含消息队列等信息。通过消息队列中的事件类型检索出HashMap中保存的需要监听该事件的被观察者列表。主要分为五种：1.POSTING 代表着什么线程发送，就在什么线程处理；2.MAIN 是最常用的，发送到主线程中处理；3.MAIN\_ORDERED 不管在什么线程发送，都是先切换到主线程发送。除非主线程工作区没有设置，就发送到当前线程；4.BACKGROUND 发送另一个线程后台中循环处理；5.ASYNC 每一次都从缓存线程池中，获取一个线程的处理。在对应的线程把对应类中的方法通过反射调用，并把事件传递到被观察者的方法中。

3、EventBus线程切换：EventBus的机制和Android的handler机制很相似。



我们比较一下Handler机制。

1. 在Handler中Message作为消息的承载对象，同理在Poster中存在着PendingPost作为消息的承载对象。
2. 在每一个Handler中都有自己的MessageQueue。同理在Poster中存在PendingPostQueue作为消息队列。
3. 在Handler中存在着Looper作为驱动(因为有naitve的下层的管道会唤醒Looper)，而Poster中承载这种角色的就有不少不同，首先HandlerPost 中以Handler作为核心驱动，在BackgroundPoster和AsyncPoster中以ExecutorService最为驱动。
4. EventBus和Handler的定位就很相似。都是这个设计中的处理器。

Okio：IO工具

Okio本质上是对NIO进行的扩展，并且做了内存优化。IO是面向流的，NIO是面向缓冲区的。NIO是非阻塞式IO操作，IO是阻塞式操作。NIO通过对Channel+Buffer的读写方式本质上是对缓存区进行读写操作。当我们把缓冲区中的写满时候，再进行一次write的写入，就能避免频繁的调用系统调用。这也是Android性能优化中IO优化的核心思想之一，为了避免过多频繁的调用读写操作，我们必须适当的设置读写大小，避免过度调用系统调用，或者一口气写入过多的内容导致一口气申请过多的pageCache，导致内存骤降，可能会触发脏数据的写到磁盘中，导致系统cpu过于繁忙。

Okio包含三个主要对象：1、Source 数据读取对象（相当于 InputStream）；2、Sink 数据写入对象（相当于 OutputStream）；3、Buffer 读写缓冲对象。

Okio的读写操作对象是Source和Sink，Okio提供了Okio.source()方法来获取Source,提供了Okio.sink()方法来获Sink；他们都是继承自Channel类，所以可以联合Selector进行nio的读写操作。

Okio内部的所有操作都要经过buffer缓冲区处理，缓冲区内部管理更细Segment，是通过一个Segment链表环加上一个SegmentPoll缓冲池处管理，每次写数据都会申请一个Segment对象，在对象中将数据转换成byte数组，当达到数组上限的时候就不允许写入，通过调用stream的读写方法调用系统的读写，这样就可以更大的利用内存，同时避免更多的缓存对象生成。

Segment的shared字段表示当前片段是共享的片段。就是多个片段的字节数组data 是同一个引用。owner字段表示当前片段拥有字节数组data的所有权，owner = true ,表示当前片段可以追加数据。共享片段的好处就是不用进行字节数组的复制，共享同一个字节数组，在某些只需要读的场景非常有效。

SegmentPool实例池的作用就是回收使用过的对象实例，当需要新对象时，可以从池中获取，而不用新的创建。针对大对象的时候，会使用这个方式，避免频繁 GC 。

Buffer通过Segment链表来动态调整存储数据大小，通过SegmentPool 来重复使用片段Segment。

Okhttp：网络请求

okhttp将整个网络请求拆分成7个拦截器，设计成责任链模式。

1.RetryAndFollowUpInterceptor 重试拦截器

2.BridgeInterceptor 建立网络桥梁的拦截器，主要是为了给网络请求时候，添加各种各种必要参数。如Cookie，Content-type

3.CacheInterceptor 缓存拦截器，主要是为了在网络请求时候，根据返回码处理缓存。

4.ConnectInterceptor 链接拦截器，主要是为了从链接池子中查找可以复用的socket链接。

5.CallServerInterceptor 真正执行网络请求的逻辑。

6.Interceptor 用户定义的拦截器，在重试拦截器之前执行

7.NetworkInterceptors 用户定义的网络拦截器，在CallServerInterceptor(执行网络请求拦截器)之前运行。

每个拦截器的逻辑都可以拆分成请求部分和应答部分。

okhttp请求流程：

当我们构建好一个请求时，我们需要生成一个RealCall对象，然后通过队列的方式顺序消费执行。

RealCall的执行方法将我们传入的监听回调结果的接口封装成AsyncCall传入Dispatch队列执行。

AsyncCall是一个Runnable对象，在里面会执行所有OkHttp的拦截器，然后返回Response。调用我们传入的监听回调结果的接口将Response传递出去。遇到异常则调用Response的失败接口，最后结束调用Dispatch的finish方法。

Dispatch维护了三个队列，readyAsyncCalls、runningAsyncCalls、runningSyncCalls。首先将AsyncCall添加到readyAsyncCalls队列中，然后在readyAsyncCalls、runningAsyncCalls中查找是否是否有相同的AsyncCall，有的话就复用请求配置。没有的话就在线程池中执行队列中的AsyncCall。‘

Dispatch中维护了一个线程池和SynchronousQueue队列，里面不断轮询readyAsyncCalls队列中的请求。而在一次请求中一次性消费AsyncCall最大的数量默认为64，且这个AysncCall复用Host的次数要小于5次。最后就会添加到executableCalls和runningAsyncCalls 加入到执行队列中。最后遍历一次executableCalls中AsyncCall的executeOn，执行其中的run方法。回到第3步。

数据返回后，会执行所有OkHttp的拦截器。这里是用了RealInterceptorChain包裹所有的拦截器。每当一个拦截器走完一个Request的处理流程就会生成一个新的RealInterceptorChain并且下标+1，时候下一个拦截器的intercept方法。不断的迭代下去。RetryAndFollowUpInterceptor 主要处理了如下几个方向的问题：1.异常，或者协议重试(408客户端超时，权限问题，503服务暂时不处理，retry-after为0)；2.重定向；3.重试的次数不能超过20次。BridgeInterceptor：主要是把Cookie，Content-type设置到头部中，如果需要自己设置，则需要在NetworkInterceptors 重新设置，不然会被覆盖。CacheInterceptor：主要是处理304等响应体的缓存。通过DiskLruCache缓存起来。ConnectInterceptor：1.尝试从ConnectionPool 中获取可以进行多路复用的socket链接(当然需要http 2.0协议的请求)；2.从ProxySelector 中获取直连或者代理的资源路径；3.当拿到了该资源路径对应所有的ip地址，(不是一对一，是因为可能是408等情况存在了负载均衡实际关联不同的服务器)，并开始尝试链接；4.establishProtocol 尝试这处理具体的协议，这个方法主要处理了Http 1.0和Http 2.0。

Retrofit：网络请求

一开始动态代理中调用loadServiceMethod方法，解析接口方法中的注解，参数，头部信息等；

依据接口方法的返回类型，从适配器工厂集合里进行查询，生成相应的适配器CallAdapter，区分是RxJava的Observable、Single还是Call或者其他类型，（适配器工厂集合的数据是由构建Retrofit时addCallAdapterFactory()添加，如无自定义，则添加Android平台默认适配器）。以相同的方式取出数据转换器Converter；

利用上面生成的CallAdapter，调用invoke方法，创建OkHttpCall对象，即针对请求信息，利用OkHttp进行异步或者同步网络请求，并且对响应结果进行实体类转换；

创建好OkHttpCall后，又利用上面查询到的适配器CallAdapter调用adapt，返回RxJava的Observable、Single或者Call对象。

在添加默认适配器工厂defaultCallAdapterFactories时，将callbackExecutor作为了一个参数，callbackExecutor即Executor，一个线程调度器。在Call的enqueue实现里执行了一个异步网络请求delegate.enqueue，在请求的响应onResponse、onFailure中 Executor也同样执行了一个线程。默认callbackExecutor的创建在Retrofit的初始化中，callbackExecutor = platform.defaultCallbackExecutor();defaultCallbackExecutor 内部其实调用的是 new MainThreadExecutor() ，很清楚的看到， handler.post(r) 内部使用Handler将响应抛到了主线程。

Retrofit为什么要使用动态代理？OkHttp在使用的时候，请求参数的配置是不是很繁琐，尤其当有一些表单提交时，又臭又长，而Retrofit就是弥补了这个缺点，利用@GET、@POST、@Path、@Body等注解以及一些参数很简便的就构造出了请求。当Retrofit创建了不同的接口，动态代理就发挥出了作用。每当不同接口方法执行时，动态代理都会拦截该请求，对接口中的注解，参数进行解析，构建出不同的Request，最后则交给OkHttp去真正执行。Retrofit结合动态代理，不用关心真正的接口方法，对符合规范的接口进行统一化的管理，以统一的方式解析注解和参数，拼接成request。

MMKV：SharedPreferences的缺点：1、跨进程不安全；2、加载缓慢；3、全量写入；4、异步落盘卡顿。根据官方性能图来看，MMKV相比于SharedPreferences性能高出不少。MMKV是一个存储键值对的组件，不仅采用了mmap将文件映射到内存上，绕开了普通IO每次读写都要进入内核再写到磁盘的过程，还将对象序列化的方式改为Protocol Buffers，标准PB不支持增量更新，每次写入都是全量写入，优化方式是每次都将kv进行append到内存末尾，这样同一个key就会存在多份value，最新的数据在最后，获取的时候不断用后面的value更新前面获取的value即可。MMKV在append数据的时候，会根据先前申请的pagesize进行数据存放，当到达文件末尾时会进行key排重，如果此时还是不够用的话，就会扩大文件的pagesize。进一步增加了性能。MMKV还使用了Ashmem匿名内存进行更加快速的大对象传输。

1. mmap：Linux通过将文件描述符fd映射到虚拟空间中初始化这部分内存容，成为内存映射。创建对应关系后，就可以通过指针操作这段内容，系统会自动回写到对应的文件磁盘上，所以能够实现进程间的通信、数据存取。
2. Protocol Buffers：protocol buffers 是一种语言无关、平台无关、可扩展的序列化结构数据的方法，它可用于（数据）通信协议、数据存储等。
3. Ashmem：共享内存是Linux自带的一种IPC机制，Android直接使用了该模型，不过做出了自己的改进，进而形成了Android的匿名共享内存（Anonymous Shared Memory-Ashmem）
4. 单/多进程共享：
5. 单进程：当只有一个进程时，第一次加读锁，m\_sharedLockCount+1，加读锁成功；后续加读锁纯粹计数。后面要加写锁时，m\_exclusiveLockCount为1，判断得出unLockFirstIfNeeded为true，此时锁升级，先加写锁，如果成功，返回true。这里没有把读次数减一，因为锁降级时，还要保留读锁。当加写锁失败时，需要解读锁，防止两个进程同时加写锁，导致死锁，这里有一方先退出即先解掉占有的读锁，这样另一方必定会加写锁成功。当解锁时，如果解读锁，m\_sharedLockCount为零表示一种错误并返回；先对读计数减一，如果有读写计数大于零，表示进行了递归加锁，不释放锁处理，否则解锁(不再占有)。当解写锁时，如果m\_exclusiveLockCount为零表示一种错误并返回，否则对m\_exclusiveLockCount减一，如果大于零则表示之前进行的是递归加写锁；否则此时不再成为写锁，那么尝试解锁或者进行锁降级为读锁，取决于m\_sharedLockCount是否大于零，最后根据  
   m\_lockInfo.l\_type = static\_cast<short>(unlockToSharedLock ? F\_RDLCK : F\_UNLCK);判断是加读锁还是完全解锁。这个过程是unlock的过程。
6. 多进程：在MMKV中使用排他锁作为写锁，共享锁作为读锁。并实现了自己的文件锁，实现了可重入锁，升降级锁。要达到这两个目标需要增加读锁、写锁计数器。加写锁时，如果当前已经持有读锁，那么先尝试加写锁，try\_lock 失败说明其他进程持有了读锁，我们需要先将自己的读锁释放掉，再进行加写锁操作，以避免死锁的发生。解写锁时，假如之前曾经持有读锁，那么我们不能直接释放掉写锁，这样会导致读锁也解了。我们应该加一个读锁，再将锁降级。那么进程间如何同步写入的内容：1、写指针同步：每个进程都缓存了自己的写指针，MMKV在文件头部保存了写指针的内存偏移量，写入的时候可以使用这个值来校对写指针；2、内存重整感知：使用一个单调递增的序列号，发生重整时序列号加一，每个进程都缓存一份，只要对比序列号是否一致，就知道是否其他进程触发了内存重整；3、通过查询文件大小得知内存增长。

死锁：1.互斥条件： 一段时间内资源只允许一个任务持有；2.不可剥夺条件： 任务所获的的资源只能由自己释放；3.请求与保持条件：任务已经持有了一个资源，但是又提出了新的资源请求，而该资源被另一个任务持有，此时请求的行为被阻塞，不释放自己的资源；4.循环等待 若干任务形成首尾相接的循环等待资源。

kotlin特性：协程

协程：协程是一种非抢占式或协作式的计算机程序并发调度的实现，程序可以主动挂起或者恢复执行。避免在异步编程中使用大量的回调，同时相比于传统的多线程技术，更容易提升系统的高并发处理能力。Kotlin 协程的核心竞争力在于：它能简化异步并发任务，以同步的方式完成异步任务。

线程：线程和协程属于一对多关系，一个线程上允许存在多个协程。线程大多数的实现是映射到内核的线程，当线程中的代码逻辑在线程抢到CPU的时间片才可以执行，否则只能等待。而协程之所以轻量级，协程并不会映射到内核线程，调度可以在用户态搞定，任务之间调度并非抢占式。

suspend原理：Continuation作为挂起函数调用时的一个隐式参数传入，封装了协程恢复后的执行代码逻辑。

创建 $continuation 对象, 它是 ContinuationImpl 类的实例。当调用 $continuation 对象的 resumeWith 方法会调用 invokeSuspend 方法，就会再次调用本协程体。

$continuation储存了方法参数的 $completion 实例，作用是当本协程体执行完毕之后，会回调 $completion 实例的 resumeWith 方法，唤醒调用方的协程。

当在协程函数内部调用其他协程函数时，都会生成一个挂起点；

调用其他协程函数的时候，都会将本协程函数的协程体 $continuation 传递给被调用的协程函数，以便被调用的协程函数可以回调恢复本协程函数。

当被调用的协程函数返回 COROUTINE\_SUSPENDED，即表明被调用的协程函数是一个异步操作，希望本协程函数挂起，等待被调用的协程函数执行完成回调它。本协程函数就会从这个挂起点直接返回，不再执行下面的代码，直到被调用的协程函数通过 $continuation 的 resumeWith 方法来恢复本协程函数，并继续执行下面代码，直到遇到下一个挂起点。

文件下载框架：PRDownload、

PRDownload：基于线程池ThreadPoolExecutor构建的并发文件下载框架。可用于下载任何类型的文件，如图像、视频、pdf、apk 等。通过在下载文件反复检测DownloadRequest的状态支持在下载文件时暂停和恢复。在下载文件时获取文件长度和下载长度为 onProgress提供进度回调、onCancel取消回调、onStart开始下载回调、onError下载提供回调，并通过Handler跨线程通知主线程进行UI提示。通过在数据库中保存下载进度结合RandomAccessFile和OutputStream的seek()方法实现文件的增量下载。

Plugin插件：ASM、防抖动、大图检测、方法自动打印Log、

ASM：ASM是一个操作Java字节码的类库。ASM处理字节码（ByteCode）数据的思路是这样的：第一步，将.class文件拆分成多个部分；第二步，对某一个部分的信息进行修改；第三步，将多个部分重新组织成一个新的.class文件。ASM经过generation操作，可以生成一个新的.class文件；原来有一个.class文件，经过transformation操作，可以生成一个新的.class文件（优化或混淆程序，将调试或性能监控代码插入应用程序，用于面向切面的编程等。）；原来有一个.class文件，经过analysis操作，可以获得分析的结果（发现应用程序中的潜在错误，检测未使用的代码，逆向工程代码等。）；

asm.jar：

ClassReader类，负责读取.class文件里的内容，然后拆分成各个不同的部分。

ClassVisitor类，负责对.class文件中某一部分里的信息进行修改。

ClassWriter类，负责将各个不同的部分重新组合成一个完整的.class文件。

以Check开头的类，主要负责检查（Check），也就是检查生成的.class文件内容是否正确。

以Trace开头的类，主要负责追踪（Trace），也就是将.class文件的内容打印成文字输出，根据输出的文字信息，可以探索或追踪（Trace）.class文件的内部信息。

在JVM当中，每一个方法的调用都会分配一个Stack Frame内存空间；在Stack Frame内存空间当中，有local variables和operand stack两个重要结构；在Java文件进行编译的时候，方法对应的local variables和operand stack的大小就决定了。

在方法刚开始的时候，Stack Frame中的operand stack是空的，而只需要计算local variables的初始状态；而计算local variables的初始状态，则需要考虑当前方法是否为static方法（this参数）、是否接收方法参数、方法参数中是否有long和double类型。

从Frame的角度来讲，实现方法的调用，需要先将this变量和方法接收的参数放到operand stack上。

从Instruction的角度来讲，调用static方法是使用invokestatic指令，调用non-static方法一般使用invokevirtual指令。

防抖动：通过字节码插桩的方式来为整个项目中所有使用了 OnClickListener 的回调方法中都插入一段逻辑代码，该段逻辑代码会对前后两次点击的时间进行判断，如果判断到时间小于某个阈值的话就直接 return。

此时就至少需要考虑以下几种场景：

1、通过代码直接为 View 设置了 OnClickListener

是否使用了 lambda 表达式分为两种

是：Java平台中，在编译阶段，lambda 表达式不会生成相应的实现类，而是使用invokedynamic指令。invokedynamic 指令在常量池中并没有包含其目标方法的具体符号信息，存储的是 BootstapMethod 信息，在运行时再来通过引导方法机制动态确定方法的所属者和类型。使用 lambda 表达式并不是不会生成对应的实现类，而是将生成时机改成了运行时，当第一次执行到 lambda 表达式时，JVM 就会在内存中动态生成对应的实现类，当后续再次执行到该 lambda 表达式时也可以直接复用该类。

Android平台中，为了能够支持 Java 8，目前 Android Gradle 插件是通过在 D8/R8 将 class 文件编译成 dex 文件的过程中插入字节码转换的操作来实现的，这个转换过程称为 desugar，也即 脱糖。desugar 操作就用于将某些目前 Android 系统还不支持的语法糖还原为简单的基础语法结构。例如，lambda 表达式经过 desugar 之后就会被转换为具体的实现类，并将生成的实现类直接写入到 dex 文件中，从而保证了 lambda 表达式也能够在低版本系统上正常运行，从而也就不存在兼容性问题了。

由于 Transform 是在 desugar 之前执行的，所以此时还未生成各个 lambda 表达式的具体实现类，导致我们没法直接根据 View.OnClickListener 的签名信息来定位到所有实现类。

这种情况下，通过 ASM 框架，在字节码层面上我们能够获取到的信息有：

该方式包含 lambda 表达式，也即包含了一条 invokedynamic 指令，对应 ASM 中的 InvokeDynamicInsnNode

invokedynamic 指令中包含了要生成的接口实例的签名信息，即 invokedynamic 指令中标明了要生成的是 OnClickListener 对象，且包含一个 onClick 方法，所以此时就可以通过遍历项目全局的 InvokeDynamicInsnNode 的 name 和 desc 两个属性，来查找到和 OnClickListener lambda 表达式关联的 InvokeDynamicInsnNode

invokedynamic 指令会指向 BootstapMethod 区域，而 BootstapMethod 中已经标明了三个入参参数，第二个参数是编译期间就自动生成的私有静态方法，当中就包含了 onClick 方法应该执行的代码块。这三个参数就对应 InvokeDynamicInsnNode 的 bsmArgs 属性，所以通过 bsmArgs 我们就能够知道 onClick 方法最终要调用的方法的签名信息，通过向该方法插入需要的逻辑就可以实现 hook 了。

每当拿到到一份字节码，就遍历其所有方法，判断方法内部是否包含和 OnClickListener 相关的 lambda 表达式，有的话则其指向的静态方法的签名信息保存起来。当拿到所有需要 hook 的方法后，再来向其插入 ViewDoubleClickCheck 防抖指令。

否：如果不使用 lambda 表达式的话，OnClickListener 在编译阶段就已经生成具体的接口实现类了，所以当判断到当前遍历的 Class 对象实现了 OnClickListener 接口的话，就拿到其 onClick 方法即可。

1. 在 XML 中为 View 声明了 onClick 属性

在 XML 中声明的 onClick 属性会在 View 类解析 AttributeSet 时进行读取，View 类会为其设置一个自定义的 OnClickListener，通过反射 handlerName 方法的形式来实现回调。

由于 Transform 无法作用于系统源码，所以我们没法对 DeclaredOnClickListener 进行 hook，有其它两种解决思路

第一种方案是为 XML 指向的 onClick 方法添加一个自定义注解，通过该注解来标明此方法需要进行 hook，也是本文采用的方法，采用此方案后使用场景就不只是局限于 XML onClick 了，也可以作用于任何符合签名信息的方法。

第二种方案是对 AppCompat 包中的 AppCompatViewInflater 类进行 hook。目前大多数情况下我们使用的 Activity 都会继承于 AppCompatActivity， 而 AppCompatActivity 会通过 AppCompatViewInflater 来解析生成 View 对象，内部会尝试代理 onClick 属性并为 View 设置一个自定义的 OnClickListener。由于 AppCompatViewInflater 属于依赖库，Transform 可以通过扩大作用范围从而扫描到 AppCompatViewInflater 类，所以我们对 checkOnClickListener 方法或者是 DeclaredOnClickListener 类进行 hook 都可以。

3、对第三方框架的支持程度。例如，

如果项目中使用了 ButterKnife 的话，要照顾声明了 @OnClick 注解的方法。

ButterKnife 会为每个使用了 @OnClick 注解的类自动生成一个辅助文件，就像以下的 MainActivity\_ViewBinding，而 ButterKnife 为每个 View 设置的 OnClickListener 都是框架内部自定义的实现类 DebouncingOnClickListener。

对 ButterKnife 进行 hook 的方案也有多种：

直接根据 @OnClick 注解来进行定位，只要解析到包含该注解的方法就对其进行 hook

根据 DebouncingOnClickListener 的匿名内部类来进行定位。ButterKnife 在为 View 设置 DebouncingOnClickListener 时没有使用 lambda 表达式，所以只要解析到其实现类然后对其 doClick 方法进行 hook 即可

将 Transform 的作用域扩大到所有依赖库。前两种方式仅需要扫描项目自有代码和通过 APT 生成的代码即可，而如果将 Transform 范围扩大到所有依赖库的话，Transform 阶段就可以看到 DebouncingOnClickListener 类了，此时就相当于在处理匿名内部类的情况。

1. 如果使用了 BaseRecyclerViewAdapterHelper 的话，要照顾为每个 Adapter 设置的 onItemClickListener 或者是 onItemChildClickListener。

onItemClickListener 和 onItemChildClickListener 方法内部也一样是通过为 View 设置 OnClickListener 来实现的，此时一样可以通过扩大 Transform 的作用范围来处理，但为了减小影响范围，我 hook 的是这两个 onItem 方法。

两个 onItem 方法的总体处理流程和 OnClickListener 一样，根据是否使用了 lambda 表达式走不同的判断逻辑，所以此时只要为 extraHookPoints 多添加两个需要 hook 的节点即可，具体的签名信息就需要查看该开源库的源码来得到了。

大图检测：本地图片检测通过Project对象获取发布快照的Task。通过给Task设置dependsOn依赖，在项目发布快照之前执行资源检测Task。通过获取资源检测Task对应的环境变量，设置文件大小阀值。获取App中的资源集sourceSets，从中过滤res、assets资源目录。从中查找类型为so、png、jpg、jpeg、svg、webp、gif、json的文件，通过对应的阀值进行文件过滤并重定向文件名到指定的临时文件中。云平台图片检测通过监测加载后的图片内存来判断。目前加载图片一般都使用第三方框架，所以可以对常用的图片加载框架进行Hook,这里主要对主流的四种图片加载框架进行Hook操作。Glide、Picasso、Fresco、Image Loader。以从网络加载一张图片举例，当使用图片框架加载一张网络图片时，会使用OkHttp或者是HttpUrlconnection去下载该图片,这时候我们就能得到图片文件的大小。当图片框架将图片文件构造成Bitmap对象以后，我们又能得到其所占用的内存，这样我们就同时的得到了图片的文件大小和所占用的内存。那么这里我们也必须对OkHttp和HttpUrlconnection进行Hook。

重复文件检测：通过Project对象获取发布快照的Task。通过给Task设置dependsOn依赖，在项目发布快照之前执行资源检测Task。获取App中的资源集sourceSets，从中过滤res、assets资源目录。遍历目录下的所有文件，将文件转换成FileInputStream获取MD5。根据文件的MD5进行分组，过滤大于1的分组数，并将对应的文件名输出到临时文件中显示。

方法自动打印Log：创建注解MethodLog，用来标记需要打印日志的类和方法。扩展属性MethodLogExtension，用于设置开关，enable：是否打开插件、openLog：是否打印、enableModule：模块全开启。在打包过程中的Tramsform流程过滤对应Class文件（.class结尾，非R$、非R.class、非BuildConfig.class）。遍历当前Class文件对应类的所有变量，如果当前类包含MethodLog，获取类名与方法名映射的参数值和参数名。获取方法上的MethodLog，解析MethodLog的各个属性。在方法入口根据类名与方法名获取到映射的参数值和参数名。（构造函数和静态方法除外）。在ASM修改对应Class文件输出日志，覆盖原有文件。

模块依赖监测：从 configurations 中拿到继承自 implements 的 CompileClassPath configuration，然后通过 asPath 方法拿到所有依赖缓存到本地的路径，然后解析依赖拿到文件和内容进行分析。

性能优化：ANR、

ANR：AMS 在执⾏特定⽅法时出现的超时错误，触发点有四个：InputDispatching Timeout、BroadcastQueue Timeout、Service Timeout、ContentProvider Timeout。同时 AMS 还会发出⼀个 SIGQUIT 信号： SignalCatcher 线程会接收到这个信号，并且处理后续的

dump 逻辑；市⾯上的 ANR 错误收集 SDK⼤部分都依赖于这个原理。常见ANR问题有：

1. MessageQueue.nativePollOnce 问题

1.1 主线程卡死情况

Android 应⽤启动过程中有这样的⼀段逻辑：zygote 初始化 →RuntimeInit 初始化。在 RuntimeInit 初始化过程中会注册⼀个默认的错误处理器来响应异常。默认的异常处理机制会在线程发⽣ Crash 时同步给 ActivityThread、ActivityManagerService 之后再“kill”掉⾃身。那么如果当主线程发⽣异常的情况下，不使⽤系统的处理链路或异常处理过程中耗时过久就会发⽣ ANR。当主线程发⽣了崩溃后其实已处于终⽌状态。此时主线程 Looper 的 MessageQueue 组件⽆法继续添加新的消息，⽽ Android 应⽤的运⾏恰恰依赖的就是主线程的消息轮询 -- 线上这个错误堆栈也是指向了 MessageQueue 组件在等待新消息的到来。因此，当主线程发⽣异常并⽆法及时 kill 掉进程时，系统就会触发 ANR 超时机制。可以通过埋点找出了线上异常处理链的各个⽅法耗时数据。

1.2 卡顿堆栈漂移

在这种情况下，错误上报中的卡顿堆栈已经失真，⽆法反映出当时现场的真实的情况。因此，增加线上的慢函数监控可以更准确地分析此问题。线上慢函数监控的原理：查看 Looper 的源码得知：主线程所有执⾏的任务都在 Looper.loop() ⽅法中的 msg.target.dispatchMessage 中派发执⾏。在这⾥有个 Printer 组件分别在消息的执⾏前、后会 有⼀个打印的⽅法调⽤。可通过 Looper.setMessageLogging ⽅法设置⼀个 Printer ，来监控每个 Message 的执⾏时间。

1. QueuedWork.processPendingWork 问题

SharedPreferencesImpl.apply() ⽅法中调⽤ QueuedWork.add() 将 SharedPreferences 的写 ⼊任务添加进 QueuedWork 的任务队列中，之后 ActivityThread 在⼀些组件⽣命周期⽅法中执⾏了 QueuedWork.waitToFinish → QueuedWork.processPendingWork 这⼀流程。这⼏个⽣命周期方法有：handleStopService、handlePauseActivity、handleStopActivity、handleSleeping。以上的⽣命周期⽅法都会先等待 QueuedWork 中的异步队列执⾏完成，再执⾏后续的流程。很容易得出这样的结论：SharedPreferences 的 apply ⽅法本身设计为异步写⼊，⽽Android 系统为了保证数据有效性会在特定的⽣命周期⽅法中等待异步写⼊任务的完成。如果这个任务处理耗时过⻓，就会产⽣ ANR 问题。通过AOP来拦截 SharedPreferences 的创建，并返回带⽇志输出功能的 SharedPreferencesWrapper 。通过 SharedPreferencesWrapper 的⽇志对SharedPreferences 写⼊频率进⾏分析。

1. ⾸⻚创建问题处理

Runtime.loadlibrary() 是⼀个 synchronize 修饰的函数，多线程调⽤必然会存在锁竞争情况。如果在启动过程中，存在着⼤量“异步”加载 so 的操作。如果⼦线程先于主线程进⼊ Runtime.loadLibrary() ⽅法，拿不到锁的主线程就会等待⼦线程释放锁之后再继续执⾏。也就是说，如果某个⼦线程中存在着耗时较久的 so 加载⾏为，就会阻塞主线程的 so 加载。通过拦截 System.loadLibrary() ⽅法并输出其耗时，找出异步加载的地方。延迟执⾏异步任务中的 so 加载、⾼通平台的 BoostFramewok 加速在合理的时间启动。

1. BinderProxy 问题

线上 Binder 问题的堆栈表现多种多样，不过其最终调⽤点都是android.os.BinderProxy.transactNative() 方法。Binder 问题最终完全发⽣在系统层，⽆法通过 AOP 进⾏拦截。因此，换⼀种⽅ 式：通过 NativeHook 可以拦截所有的 Binder 调⽤，从⽽获取到 Binder 调⽤运⾏期的数据。此处选 择了 BinderProxy.transact() 作为拦截点。以下情况频繁获取本身 package 信息：如 versionName，versionCode

、频繁获取设备信息：如 IMEI，IMSI，⽹卡地址、频繁⽹络类型检测，⽹络状态检测、频繁访问本地⽬录、频繁权限检测。这些⽅法的调⽤频率都出现了异常（有些甚⾄超过了 View 绘制）。

Lint：Lint插件

Lint：Android Lint本身就是Java、Kotlin的AST抽象语法树，通过语法树找到所有代码的节点，然后对其逻辑进行判断，如果不符合规范就会抛出ISSUE。Android Lint现在使用的是UAST规范，并使用UastScanner进行代码扫描。虽然我们通过lint将项目内的代码进行了一次约束，但是已经编译成.class并不会被这段UastScanner所识别。

使用的时候通过自定义IssueRegistry作为需要检测的所有ISSUE的仓库，再将各个ISSUE注册进去。将需要检测的规则继承Detector，并实现对应的规则接口，主要是需要检测的范围。然后再getApplicableUastTypes方法中返回要检测的语法树，然后在createUastHandler定义UElementHandler实现类，并重写对应的方法来判断逻辑。