**1. 代码质量检测工具与代码质量管理机制(QA) -454**

以前：项目代码质量检测管理的手段很单一；通过在wiki文档中定义好具体的代码规则与注释模板，通过checkStyle工具进行代码检测(团队定义了些规则)，还有就是代码合并时进行人工审核；整体实现效果不明显，实施意愿不太高。

现在：重写构建代码检测流程，

1. **Gradle构建优化 -550**

（1）这是之前项目开发中发现的一个问题，Gradle本地构建时间会很长，构建起来会很慢，其实究其原因最核心的问题是我们在项目中集成了太多构建相关的插件，像APM、性能日记采集、神策埋点、注解处理器等等，这些插件都会在我们构建期间生成各种task，然后在构建期间去执行它。还有像注解处理器这种还会在编译期间对代码进行注入； 我的一个解决方案就是生成不同的变体，Gradle有一个变体的概念，就是在不同环境不同变体下对Gradle生成不同的配置；从而删减调一些不必须的插件；像我们开发过程中都会涉及到几个环境，像测试、灰度、正式，每个环境其实对插件的依赖都不同，比如在本地环境构建的时候仅仅想看代码运行效果，那就可把其他不相干的构建插件删减掉，从而提高我们的构建效率；

（2）还有其他的像代码检测、资源文件检测、图片优化等等不需要用的话都可以相应的关闭，也是能够提高本地构建速度的。

（3）还有依赖尽量不要使用动态版本依赖，这样也会每次去远程拉取导致构建变慢。。

gradle的变体可以使用一套代码打出不同的应用程序包，我们可以通过设置一个fastDevelop变体，关闭混淆、删减一些开发时非必须的插件(APM、性能监控、lint静态代码检测、系统图片检测、神策三方构建插件)，从而提高编译速度。但是安全与效率不可兼得，这个变体只在开发过程中使用，在提交给测试人员、发布时还是使用原来的变体。

变体构建主要对build.gradle进行修改，对正常的开发影响不大。但因为增加了一个变体，所以需要开发人员了解各个变体的作用，并在需要时手动切换。

Gradle的apply from 和 apply plugin 区别：

apply from：从本地或URL找到相应Plugin，直接加载进来；

apply plugin：从Maven服务器找到相应的plugin，调用对应的Plugin.apply()； --Plugin插件开发；

自定义Plugin：实际上是在Plugin.apply()上跑各种task(代码扫描、注解处理-代码注入、)；

1. **插件化技术**

插件化的核心是，就是把复杂度很高的业务模块独立成插件，然后根据不同的业务需求对插件模块进行动态替换/动态加载，提供一种实时修复的手段；这有两个概念：一个叫宿主(也就是我们的apk)另一个是插件（也就是我们理解的patch补丁）；像插件化技术最常见的应用就是我们的热更新/热修复，常见的就是我们的tinker热修；

像一般的原理，比如tinker在合并完补丁生成补丁dex后，Tinker是把dex单独存放在本地目录下，在启动时通过反射去获取Application的ClassLoader中的 pathList 字段，然后将 patch 包里面的 dex与基础包的 dexElements(有序数组，存放类信息) 进行合并，最终以此达到类修复手段。 其实就是把补丁包对应的dex插入到原来有序数组dexElements的最前面了。

App启动源码，Application初始化时（ActivityThread.handleBindApplication）会初始化pathClassLoader；

1. **性能分析工具：**

SysTrace：分析方法执行耗时、布局加载耗时；

Layout Inspector：分析布局层级或者布局嵌套情况；

Memory profiler(堆转储:将当前堆内存情况进行快照)：分析当前应用内存状况，检测内存泄漏的情况；

1. **性能优化：**

首页启动速度优化：

布局构建速度优化：

Activity页面启动速度优化：

内存优化(堆转储)：内存泄漏(核心生命周期长的对象持有生命周期短的对象引用，导致短生命对象没被及时回收；静态变量、单例持有；属性动画一直循环执行没有关闭；)

自定义控件优化：避免在频繁调用的方法如onDraw上或动画过程中进行内存分配导致频繁GC卡顿；减少invalidate、requestLayout；保持View整体层级的扁平化，提高构建速度；

1. **属性动画：**

创建属性动画，指定区间数跟动画时间、监听、启动动画、、、计算已完成的动画分数 - 根据插值器改变动画分数、根据这个动画分数计算具体的 Value，监听返回；

1. **Dalvik：**

JIT(just in time)实时编译（运行时）,运行的时候将字节码翻译成机器码,所运行的目标文件与硬件平台无关,app运行效率低

ART:

AOT(Ahead of time)预先编译（安装时）,运行前将字节码翻译成机器码,所运行的目标文件(oat)与硬件平台相关.app运行效率高。但会占用空间。APK安装所需时间增加

**public Class SingleTon{**

**private Static volatile SingleTon mSingleTon;**

**private void SingleTon(){}**

**public Static SingleTon getInstance(){**

**if (mSingleTon == null){**

**synchronized (SingleTon.class){**

**if (mSingleTon == null){**

**mSingleTon = new SingleTon();**

**}**

**}**

**}**

**return mSingleTon;**

**}**

**}**

1. **JetPack：**

**一种概念，可以理解为Google官方推荐的一种开发模式或一种开发的规范，Jetpack集成了各类三方库、AAC组件（LifeCycle、LiveData、ViewModel、Room...）、这些东西都是Google未来大力支持并持续维护的东西。**

1. **Kotlin协程：**

**理解为在主线程与子线程中间加入一个协程管理的管道、消除传统CallBack。**

**GlobalScop。Launch(Dispatcher.Main) {**

**Val value1 = request1(params)**

**Val value2 = asyn {**

**request2(params)**

**}**

**Val value3 = asyn {**

**Request3(params)**

**}**

**updateUI(value2.await(), value3.await())**

**}**

**Suspend fun request1(params :Parameter) {**

**withContext(Dispatchers.io) {**

**repository.getListData()**

**}**

**}**

**10、技术分享分享过什么？**

1. **Kotlin协程的使用和封装。**
2. **Jetpack上AAC组件的介绍、比如LiveData跟ViewModel结合使用对比替换本地项目的Presenter的可行性。ViewModel中Repo数据返回的Model需要进行BaseModel封装逻辑(State、ErrorCode、ErrorMsg、泛型Data)。**
3. **JetPack上AAC组件，Databinding+LiveData+ViewModel绑定UI控件更新，注意一点xml不能做任何逻辑动作，只能绑定具体的view到ViewModel上，然后ViewModel上数据被更新时，通知LiveData然后自动更新所绑定的View。**

**RxJava优势在于它是基于事件流的处理逻辑，比如奇数的图片直角展示、偶数的图片要圆角展示。**

**Kotlin协程优势在于线程切换、异步管理，比如一个页面展示依赖三个接口，Kotlin协程能够通过asyn.await() 并行的处理三个接口而且逻辑清晰。**

****PS：协程是一种并发设计模式，您可以在 Android 平台上使用它来简化异步执行的代码。****

**所以：项目中可以将RxJava的线程切换逻辑更替为Kotlin协程来实现，RxJava专注去处理特定事件流的逻辑。**

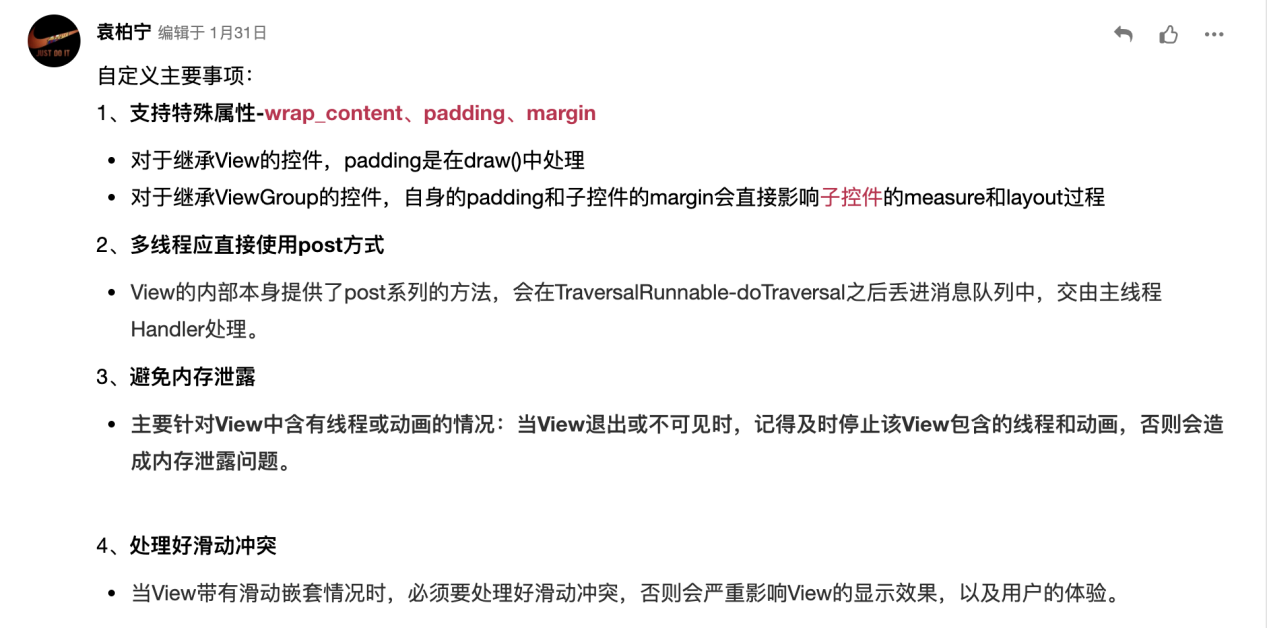
****11、尝试去做一个网络框架（HttpUrlContextion、okHttp）：****

1. **底层通过Socket实现。**
2. **考虑线程创建、线程切换的问题、线程池的使用 (连接池的管理)。**
3. **考虑异常处理机制。**
4. **考虑数据缓存机制（缓存对于移动端是非常重要的存在、减少请求次数，减小服务器压力、本地数据读取速度更快、在无网络的情况下提供数据）。**
5. **数据返回的格式处理Converter。**
6. **完善API、暴露上层更易使用的接口方法。**
7. **Https。**



1. ****自定义View：****

https://tower.im/teams/603160/todos/322/



1. ****Js-Native封装：****

Native 发送消息给Js，先会判断MsgQueue是否已重置为空，未重置则先把Msg存放到MsgQueue上，等待H5页面加载完触发onPageFinished方法时，(注入Js方法)，再把MsgQueue队列上的Msg逐条发送给H5页面。（Js代码库-即WebViewJavascriptBridge.js，只有在页面加载完成中才能注入导致的），H5页面处理完后，若有返回值则通过唤醒Native= CUSTOM\_PROTOCOL\_SCHEME + '://' + QUEUE\_HAS\_MESSAGE，让Native主动loadUrl去刷新，然后H5返回：iframe.src=’yy://return/\_fetchQueue/[{"responseId":"cb\_1\_4321","data":"xxxx"}]’, 数据回传给Native，拿到具体返回值。

JS注入时机：onProgressChanged、onPageFinished

Js 发送消息到本地，不是直接发送，而是将消息存放至Js的sendMessageQueue上，然后唤醒Native（messagingIframe.src = CUSTOM\_PROTOCOL\_SCHEME + '://' + QUEUE\_HAS\_MESSAGE），让Native主动去flushMessageQueue，然后通过loadUrl()去拿消息，H5返回：iframe.src=’yy://return/\_fetchQueue/[{"data":"xxxx","callbackId":"cb\_1\_4321"}]’，然后处理返回值再loadUrl(javascript:WebBiewJavasriptBridge.\_handleMsgForNative(Message={responseId: cb\_1\_4321, responseData:XXX})), 数据回传给H5页面，拿到最终返回值。

前提问题：

1、之前一直都是用JavaScriptInterface+loadUrl的方式处理JsNaive的交互、首先这种方式在低版本是存在漏洞的。

2、而且操作Js直接通过loadUrl或eveluateJavaScript的方式调用H5的方法，并且调用的地方很零散、缺少统一的封装跟维护。

3、还有这种方法本地跟JS调用不能设置返回值，调用完还要另外单独设监听回调。

4、还有 同一套处理逻辑，H5给到iOS跟Android两端的方法名有时还不一样，后期维护就问iOS端些业务都不好问，所以就做了这个JsBridge的统一。

事先：三端统一 约定具体方法名、约定具体协议格式、Data数据格-一般以Json格式来传值。

关键点：

1、MsgQueue队列，未完成H5页面加载前，暂时保存Msg消息。 而Js端的队列用来保存push的Msg，然后唤醒Native来刷新队列获取。 --- Native跟Js的MsgQueue意义是不一样的。

2、Native通过LoadUrl将数据注入H5，给H5页面处理具体逻辑。

3、H5通过iframe.src的值重置来触发Native的WebViewClient的ShouldOverrideUrlConnection()方法。

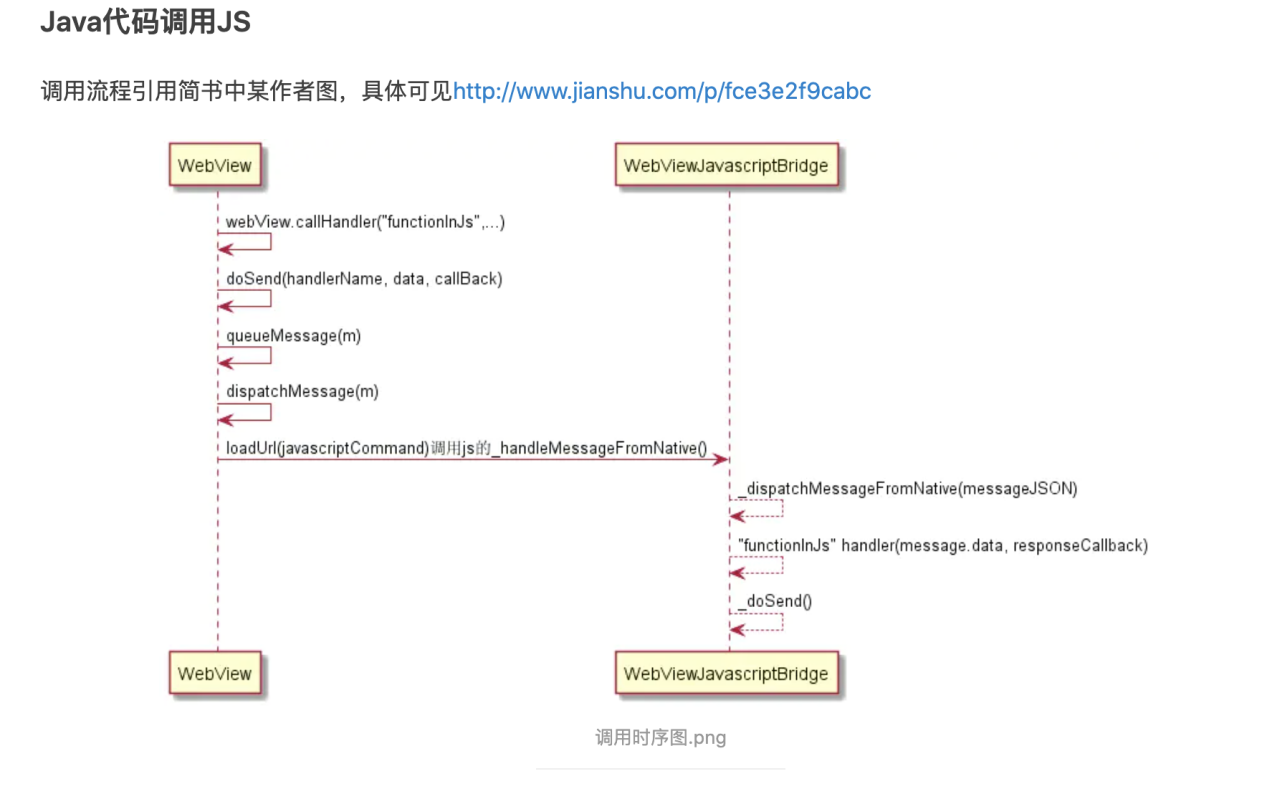
4、H5的任何数据回传给Native都是先唤醒Native再让Native主动刷新获取H5的MsgQueue上的消息。

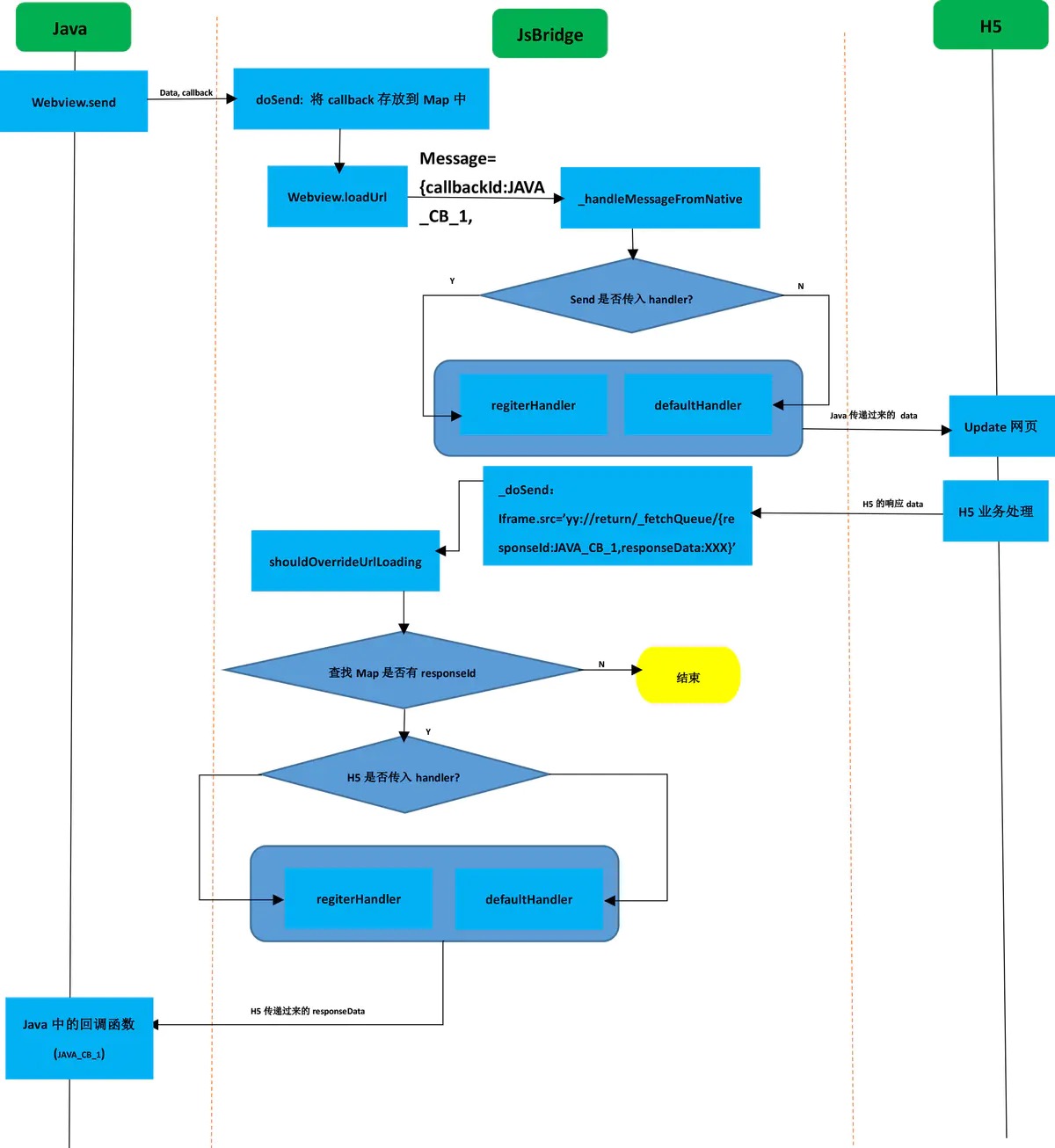
5、几个值很重要：handlerName、callBackId、ResponseId、ResponseDate、defaultDate.

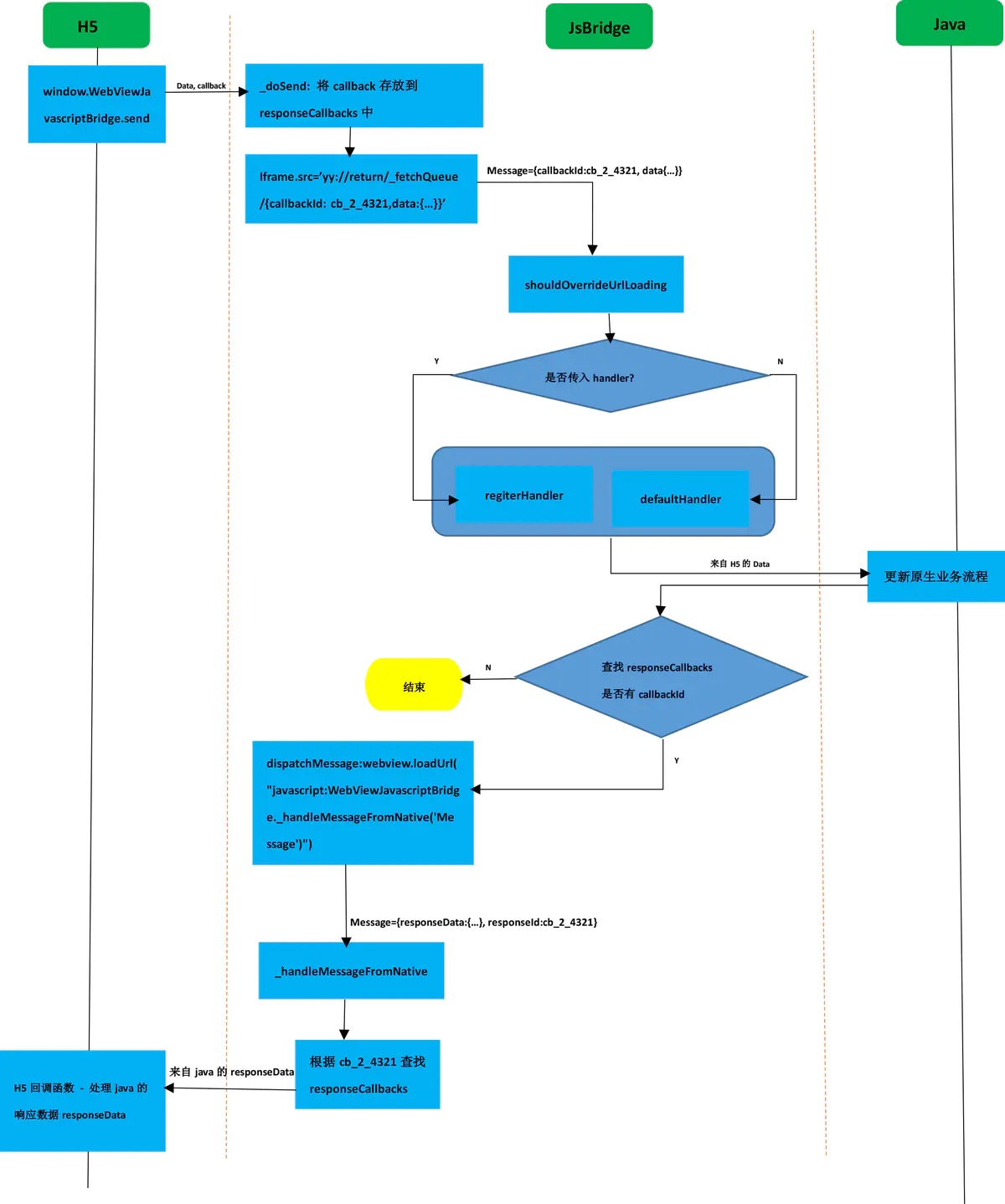
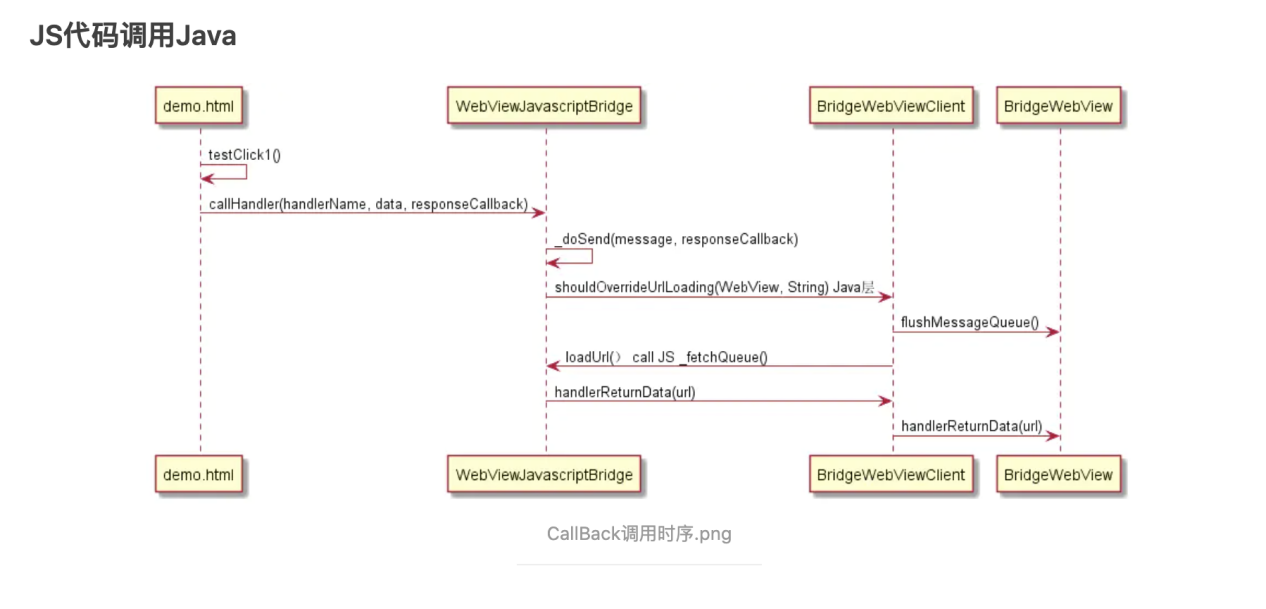
注意：

1、由于webview调用js方法的时候必须在主线程才能生效，所以偶然会出现java调用js失败。

2、另外，Js调用Java偶尔也会失败，因为iframe机制不能保证每次都能触发shouldOverrideUrlLoading回调、这个是之前H5同事反馈，但自测跟测试同事反馈没什么问题、应该是个低频事件.







1. ****Bugly维护：****
2. NullPointException、IndexOutOfBoundsException、IllegalArguementException（非法参数）、ClassCastException。

1.x 部分集合类Key、Value值不能为空。（HashTable、ConcurrentHashMap）

2、注意主线程、子线程的操作，像自定义控件、或者对控件绘制相关的方法必须放主线程。

3、先定位具体的类、然后定位具体的方法或View动作，如果页面是复杂的布局层级的话、还能对比具体的View ID来定位，对比APK包上面的资源ID - 根据包上面的resources.arsc资源索引文件来定位。

4、Bugly：有些比较难复现的问题比如资源、销毁等等，可以尝试开启系统不保留活动。

5、Trace - ANR的问题、不能只盯着看主线程，主线程基本看不出问题、必须去其他子线程上定位问题、先大概过滤下上层业务类信息堆栈再尝试定位。

1. ****ApplicationInterceptor 和 NetworkInterceptor 区别****

如果只关心最初始请求与最终响应的话、不关心中间的请求响应处理过程-如失败重连重定就能在ApplicationInterceptor 拦截处理；

如果想观察失败重连与重定向及原始网络上传输的数据的话就在NetworkInterceptor 拦截处理。

1. ****性能优化：****

1）分析工具原理：

1、启动耗时-Application初始化时通过代理+反射-Hook方式去做Instrumentation的代码注入。

App启动时间：AOP注入 (或adb shell am start -W com.ylmf.androidclient/com.main.partner.user.activity.LogActivity)

2、方法耗时-AOP技术、AspectJ编译器注入代码

Fragment则对fragment.onAttach、performCreate、performCreateView、ActivityCreate、Start

2）绘制流程：

View构建时长：Hook、AOK setContentView、inflate -》 检测布局加载进Decor的时间、

在inflate阶段：LayoutInflaterCompat.setFactory2（CallBack-onCreateView）{ }

View绘制时长：Hook、AOP dispatchDraw-每个view都会触发一次、所以可以累加计算布局层级。

3）流畅度FPS：

Choreographer.postFrameCallBack();;拿回调值计算。

如何保证闲时不回调问题？子线程每30ms(大于一帧16ms略小于两帧32ms)Post一条Msg，如果正常拿到并且时间比callBack时间短、那就意味着callback在闲时状态，取消监听。

要做启动优化和布局构建优化前应该要理解页面启动的流程以及布局构建绘制的流程；

**简单描述流程关键节点：**

1、setContentView或inflate过程读取文件IO、解析布局结构、反射构建View类等耗时动作。

2、测量、布局、绘制每个阶段的耗时。

3、自顶向下的遍历、层级过多、遍历过程耗时越长。

4、多层级布局嵌套或不规则使用RL、LL布局嵌套会多次触发绘制。

4）性能优化：

1、减少布局层级、避免RelativeLayout、LinearLayout层级嵌套、RL会对子View做两次测量。

非必要展示的布局可以进行延后加载、在代码上动态添加。

2、利用Merge、ViewStub标签

-Merge能够减少一层层级(只能在根布局上用)，一般配合Include标签使用。

-ViewStub动态加载，有效避免首次启动时的View构建与绘制(ViewStub只能用于布局)

3、宽高属性尽量少使用Wrap\_Content、WC.会增加布局测量的计算成本(需要将所有子View计算出来大小才能确认)。

4、尽量用Spannable、Html转文字的形式去构造富文本效果。

5、层级过多可以考虑使用constransLayout约束布局。

6、避免过度绘制、尽量少在布局上面使用BackGround属性、自定义View的话避免同一区域被绘制多次。

7、避免非必要的requestLayout和invalidate重绘方法调用。

8、像ListView、RecycleView、可以增加一个滑动监听、等滑动停止后再去加载列表数据。

9、避免使用帧动画。

10、涉及到视图动画的、在页面不显示或不可用时记得关闭资源避免泄露。

11、自定义控件的话尽量减少onDraw上的耗时操作-比如大的循环、创建大的对象跟局部变量。

通过canvas.clipRect(l,t,b,r)指定具体绘制区域。记得canvas.save()、canvas.restore()。

12、引入X2C框架、编译期动态将XML布局转Java布局-针对复杂页面。

13、使用ConsTraintLayout替代RL、LL，减少多层级嵌套、减少页面复杂度。

14、能用FrameLayout替代RL或LL且不增加层级的话，就用布局结构最简单的。

1. ****核心理解-scheduleTraversals-屏障消息：** mChoreographer.postCallback( Choreographer.CALLBACK\_TRAVERSAL, mTraversalRunnable, null);**

**插入一个屏障消息，进入Choreographer等待（FrameLocked）、同时向底层JNI发送一个vsync请求、等16ms时间戳到达后JNI回调回来，再判断当前是否还在执行绘制动作，没有的话往MsgQueue发送一个异步消息，然后等待Looper执行消息， 提取mTraversalRunnable进入PreformTraversals， 进入绘制流程...**

1. ****Okhttp：****
2. ****HttpClient内置配置：**Dispatcher、ConnectionPool、intercepter、Protocols支持Http版本号、Timeout、SocketFactory、Cache、Cookie。**
3. **Request-请求报文类：请求行、请求头、请求体。**
4. **Call-链接请求类：execute、enqueue、同步、异步。**
5. **RealCall-真实请求实现类：**

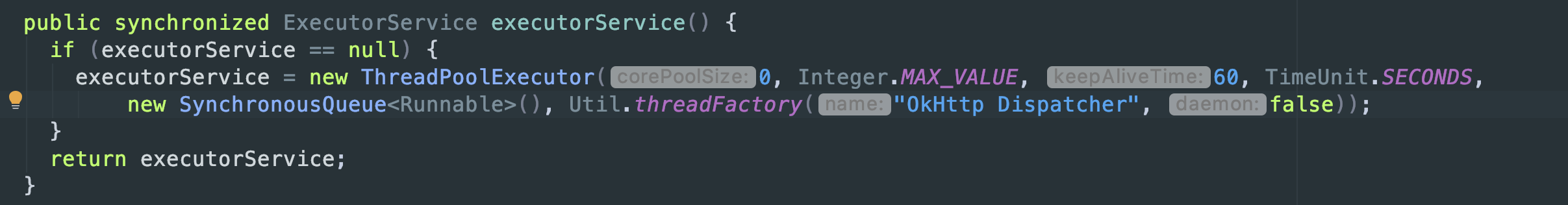
****Execute():** 通过client.getDispatcher().execute(call) -> 放入runningSyncCalls 队列-> Response r = getResponseWithInterceptorChain() 再执行Intercepter责任链的处理逻辑获取Response,最后执行Dispatch.finish(this) 在runningSyncCalls队列中移除当前call。**

****Enqueue()：**client.dispatcher().enqueue(new AsyncCall(responseCallback)) -> 先把Call请求封装成Runnable，然后调用Dispatcher的enqueue方法，放入readyAsyncCalls队列，执行promoteAndExecute()执行调度。**

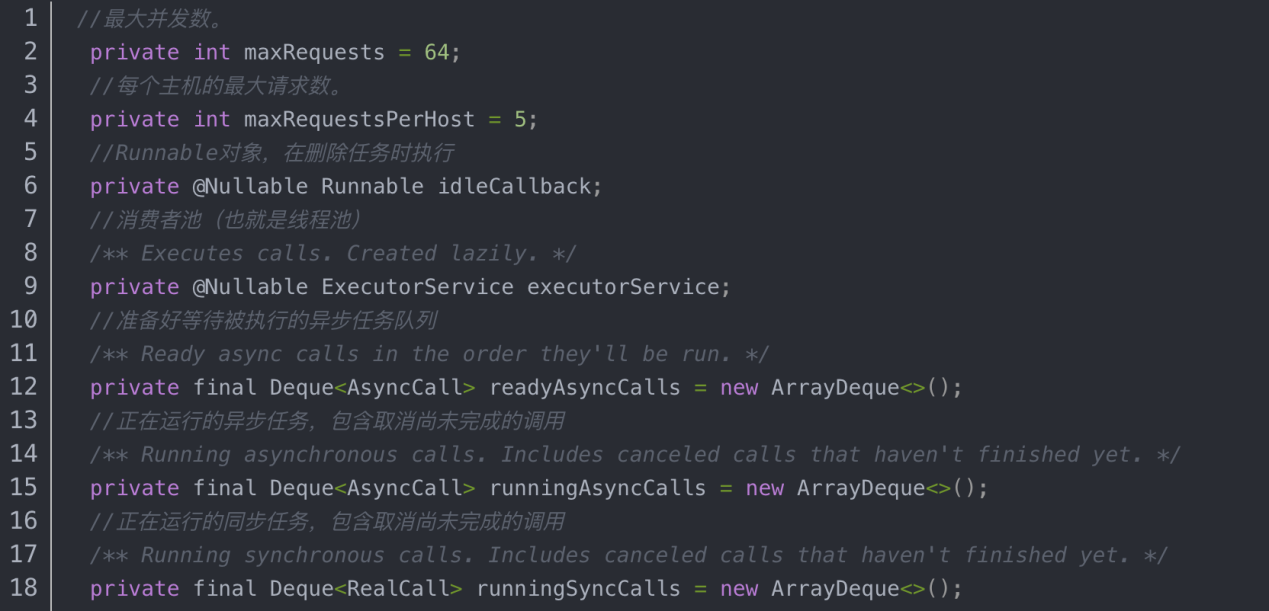


判断是否超最大连接数64、及单主机(域名)数5，然后线程池执行AyncRunnble.execute(),执行完后回调DisPatcher.finish() 重新刷新队列-执行。

OkHttp线程池是缓存线程池，线程数无限制、60秒存活。



****Dispatcher核心：**maxRequests、maxRequestsPerHost、readyAsyncCalls、runningAsyncCalls、runningSyncCalls。**



****五层拦截器重点：****

1. ****retryAndFollowUpInterceptor****

**处理请求的失败重连及重定向问题，例如需要验证、接口重定向等，并创建streamAllocation(ConnectInterceptor被用到)。**

****原理：**第一次执行时，会调用后面的拦截器链获得返回的Response，然后根据Response中的信息，最主要的就是状态码，重试次数，请求体是否允许重复请求，决定是否需要进行重新连接，既然要进行重新请求，那么有可能会对url进行改变，如果改变就不能使用之前建立好的stream,需要重新建立。**

1. ****BridgeInterceptor****

**网桥拦截器，处理请求响应过程中的Rquest与Response，本地格式与网络格式的转换，Response解压缩。**

****原理：**对用户传入的请求做处理，在程序员编写代码时只需要指定Url和请求体，但是事实上，一个完整的请求报文远不止这些信息，BridgeInterceptor帮我们做了这些事，添加各种请求报文所需要的字段（Content-Type、Content-Length**

**、connection、host、cookie、accept-encoding），然后再传递给下一个拦截器去执行。对于得到的返回结果Response，其内容可能经过了Gzip压缩，所以BridgeInterceptor帮我们做了解压缩。**

1. ****CacheInterceptor****

**处理请求与响应过程的缓存的数据（底层基于DiskLruCache），注意只能缓存GET请求的缓存、Http协议规定Post不允许缓存。**

****原理：**先获取缓存策略-CacheStrategy(本地找不到缓存、Https且无握手、不可缓存的方法如Post...则直接请求)，根据头部信息(lastModified与If-Modified-Since)确定缓存是否可用、最后如果不可用进行网络请求返回Response再缓存起来(DiskLruCache、Key为URL的MD5-Hex值)。**

1. ****ConnectInterceptor****

**负责建立Socket物理连接（非阻塞okio），处理ConnectPool连接池复用问题，完成握手，以及创建负责编码解码的HttpCodec。**

****原理：**通过StreamAllocation获取一个HttpCodec与RealConnection。**

1. **当前StreamAllocation的connect是否可用（主要看Host是否匹配、connect的负载数(allocations)超标？），可用直接返回（比如刚重连或重定向）。**
2. **第一次去连接池找connect，找到返回。**
3. **找不到，修改路由（可能多路由合并）,重新去连接池找，找到返回。**
4. **若还第二次还找不到，就直接创建一个RealConnectiong连接，握手...**
5. **将RealConnection放入连接池，清理连接池CleanupPool(计数+标记)。**

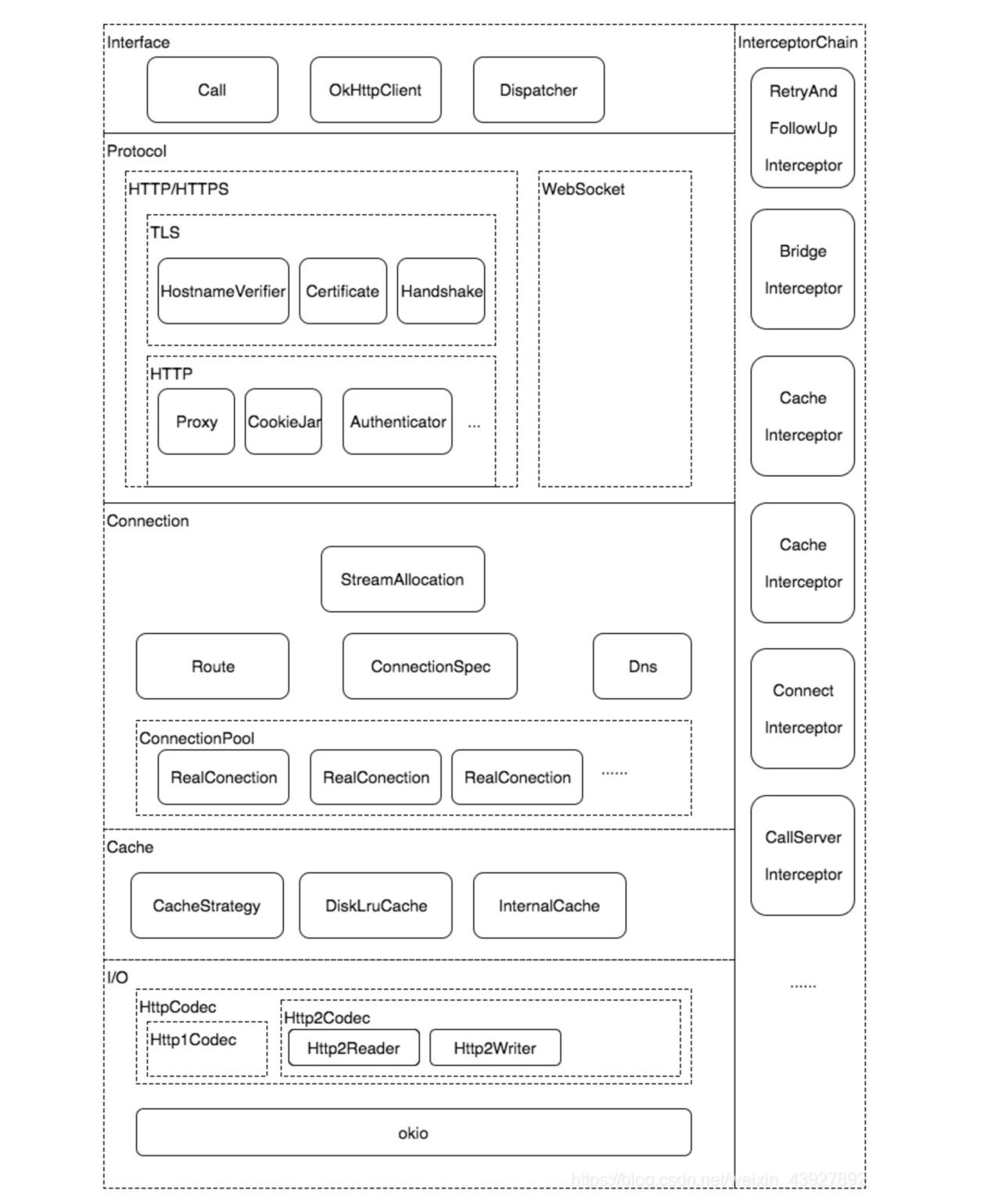
****Ps：ConnectionPool支持5个并发KeepAlive，默认保活5分钟。****

1. ****CallServerInterceptor****

**真正的跟服务器进行通信，进行Socket I/O，读写数据。**

****原理：**通过ConnectInterceptor创建的HttpCodec对请求与响应的数据进行编码、解码操作，最后进行Socket I/O，这个io是用的非阻塞的okio，效率更高。**

****OKHTTP总结：****



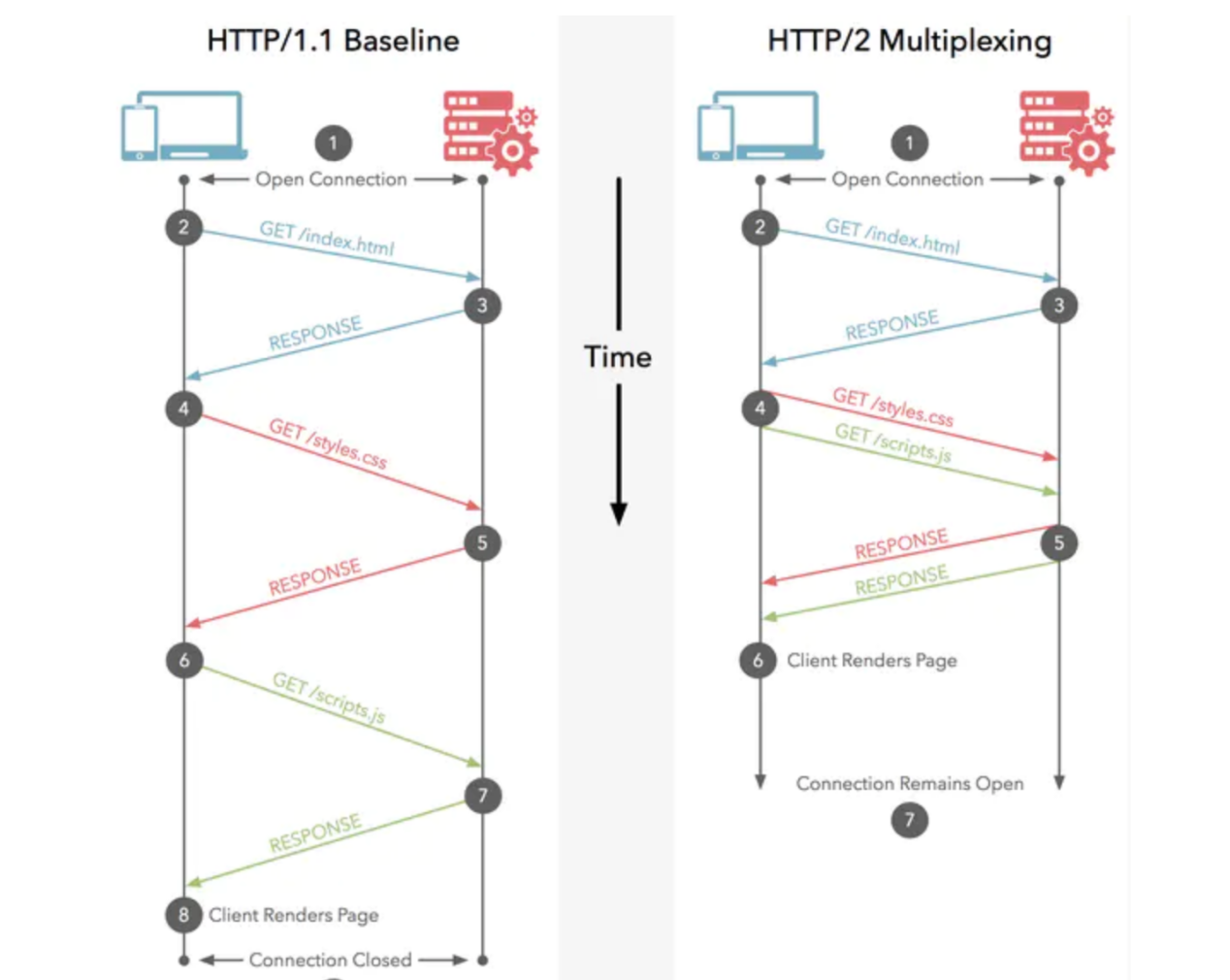
****Okhttp的优势：****

1. **支持Gzip压缩，减少传输流量。**
2. **支持SPDY，对同一主机(域名)的请求共享一个Socket连接。**
3. **支持Http2.0，头部压缩、多路复用。**
4. **内置连接池、高效复用连接，避免连接握手及初始化耗时。**
5. **支持同步、异步。**
6. **DiskLRUCache数据缓存。**
7. **Intercepter基于责任链模式，支持自定义拦截。**

****Http2.0支持多路复用，何为多路复用：****

**HTTP 1.1 中提供了 Keep-Alive，让一次连接能多次传输数据。**

**HTTP2.0 支持多路复用，能够并发的进行数据双向传输。**



****Http缓存相关头部：****

1. ****Cache-control:** private, max-age=0, no-cache -新版相对时间**
2. ****Expires:** Sun, 03 May 2015 23:02:37 -旧版本绝对时间**
3. ****Last-Modified:** Mon, 08 Dec 2014 19:23:51 GMT**
4. ****If-Modified-Since(C端):**Mon, 08 Dec xx GMT -S端返回304就用缓存。**
5. ****Etag:**“5d8c72a5edd6a:3239” **If-None-Match:**“5d8c72a5edd6a:3239”**
6. ****如何维持App的高稳定性：****
7. **Crash监控-Bugly、客服（每周一次汇总，模块划分、针对性处理）**
8. **持续性能监控与优化（启动速度、构建速度、绘制速度、内存、耗电量）**
9. **业务稳定性层面，代码优化、重构、代码Review，坏味道分析。**
10. **APM监控（针对App很多旧逻辑异常处理经常直接Catch也不打Log，这些都是很不合理的，之前专门有同事将整个业务层的Catch捕获逻辑整理了一遍，将这些直接Catch不做任何处理的异常上报至APM系统）**
11. **完善CI层面的监控，比如每次大分支的合并（或者对Application类的修改时）就会触发Jenkins去跑Appium的检测，记录下冷启动的时间，当跑出来启动时间连续几次冷启动都超过某个边界值的时候进行报警。尽量早的暴露、发现问题，让开发人员进行处理，比如针对相关优先级低的功能延后初始化等。**
12. ****内存优化：内存抖动、内存泄露、内存溢出 怎么处理？****
13. **利用AS的Profiler工具进行监控。**
14. **利用CI持续集成、尽早暴露问题、尽早处理。**

****抖动问题：****

1. **抖动是由于程序内短时间大量创建对象导致频繁GC，GC占用ui线程与cpu资源导致App卡顿。**
2. ****几点建议：****
   * 1. **避免代码里边频繁创建对象，尤其是循环里边频繁创建。**
     2. **对于自定义View，避免在onDraw上创建对象，尤其大对象。onDraw会频繁被调用。**
     3. **使用到Bitmap时，避免首次直接加载大图，可先通过option.inJustDecodeBound计算对应的采样率再加载进内存。**
     4. **大量使用Bitmap的时候，可以把它缓存进容器中进行复用。**
3. ****内存泄漏：****
   * 1. **文件IO流、数据库Cursor、等等资源使用后未关闭**
     2. **内部类、匿名内部类持有外部类引用，静态内部类持有外部类强引用，如Handler、AsyncTask。**
     3. **注册的一些监听、观察者、没有及时Remove或注销掉。**
     4. **静态的集合，将一些对象放入静态的集合中，关闭时没有及时清除掉，被静态集合一直持有、也会导致内存泄漏。**
     5. **总的来说，生命周期长的对象持有生命周期短的对象，就会造成内存泄漏。**

****内存泄漏工具：****

**1、线下：AS Profiler、StrictMode严苛模式、LeackCanery。**

**2、线上：不搞、风险大、需要频繁触发GC才能通过引用链检测泄漏。**

1. ****App绘制优化****
2. **Choreographer类能够设置Callback拿到每一秒钟的刷新次数来间接计算FPS。**
3. **对于布局构建与绘制时间的检测我们可以通过反射+代理这种Hook的方式进行，或者通过AOP技术在编译期注入关联代码计算。**
4. **主要用到SysTrace去看绘制流程-每一帧周期的处理细节。高版本的手机可以用Perfetto这个工具来跟踪App的启动流程与绘制流程。**
5. **通过ViewStub标签，将不用立即初始化的View进行延后Inflate，比如可能对于启动页或登陆页，只有当用户点击某个按钮才会加载的布局，就没必要在第一次setContent的时候将所有加载进去。还有，像一些背景图片、默认底图最好能做WebP无损压缩加载起来也会更快。**

**XML读取IO、布局加载反射、层级深遍历耗时越长、嵌套RL、LL导致重绘次数增多。 X2C省去文件IO、创建View的反射耗时。**

****性能优化三方面：启动流程、布局构建流程、绘制流程。****

**最后对**LayoutInflaterCompat.setFactory2做切面监控**，监控没个View绘制。**



针对**AppCompatActivity**，它会在onCreate上进行getDelegate().installViewFactory();

**设置Factory2拦截器**，针对常见View进行布局替换并用New的方式直接创建。



**<https://blog.csdn.net/lwj_zeal/article/details/88728942>**

1. 性能优化主要三方面：**开发阶段**、**CI集成阶段**、**线上监控阶段**。

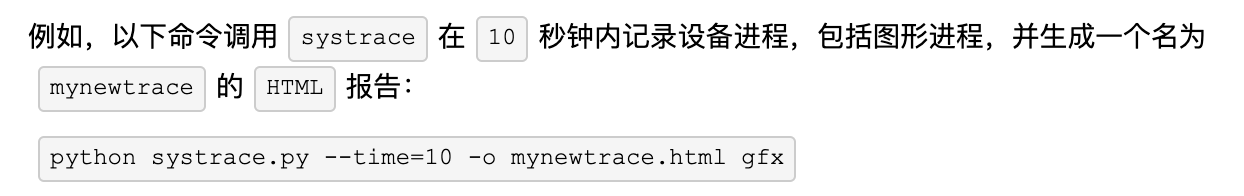
开发阶段：（SyaTrace+[StrictMode](https://copyfuture.com/blogs-details/20191220145541631uw5a1crvdbqc4qt)）主要通过抓取启动页面、核心页面的Trace文件、通过SysTrace、高版本通过perfetto工具去分析、启动流程、布局构建流程、绘制流程的执行周期图与帧率图，找到相关的耗时节点、比如启动流程Create阶段还是Start、Resume阶段耗时、还是布局构建阶段耗时、绘制阶段？一步步分析、比如构建阶段耗时、还可以通过设置LayoutInflateCommp去设置Factory拦截器去监听每一个View的构建时间，还有就是通过StrictMode打开严苛模式，（detectDiskReads、Write、Network、CustomSlowCalls、penaltyDropBoxshushu）记录调试时产生的UI线程操作IO、网络、数据库、内存泄露等问题逐一解决。。

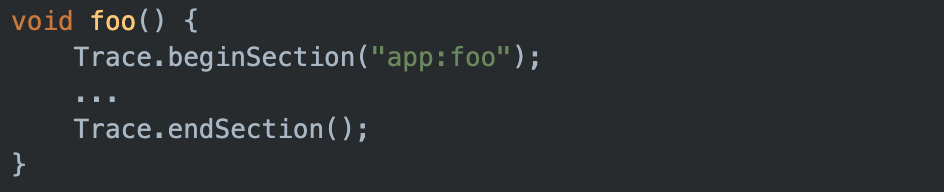
CI阶段：（Appium+STF跑场景生成报告）通过一些持续集成的手段去跑自动化测试脚本、比如Appium+STF云台设备去跑真机场景、生成性能测试报告，根据报告分析启动流程、关键页面的加载流程那部存在耗时、再一步步去分析。

线上阶段：主要依靠Bugly与APM监控平台、因为我们会通过反射+代理Hook的反射或通过AOP切面注入的方式去搜集启动流程、布局构建流程、关键页面关键View的绘制流程的时间，上报APM、还有像一些旧逻辑旧代码中的Catch异常捕获不处理的流程都会进行上报、流量信息TrafficStatic.getUidRxBytes() / Tx()。

**优化：**

**·开发阶段：**通过SysTrace+函数插桩的方式去分析、拿到启动流程跟首页加载的完整周期图或全景时序图后，分析哪些节点存在不理想、比较耗时的动作，然后再针对具体业务进行分析整理和优化。（ 命令行抓取：python systrace.py [options] [categories] ）



 --**自定义SysTrace节点**

**·CI持续集成阶段：**主要就是通过AOP在编译器注入的方式或者反射+动态代理去Hook一些核心生命周期方法、或者通过设置Looper的Printer消息执行的耗时、LayoutInflateCompat的factoryCallBack每个View的构建耗时、或Choreographer的callBack获取刷新率。

**·线上阶段：**主要通过监控平台Bugly、本地APM平台上报信息去做一个监控、或者维护子进程去监控内存、卡顿、ANR等信息。

****ANR本质上是系统(SystemServer)对应用主线程的耗时监控机制！！****

线上ANR监控手段：（线上- FileOberver观察 / Looper设置Printer计算 / 引入ANRWatchDog、BlockCanery 、 线下- Lotcat、拿本地Trace分析）

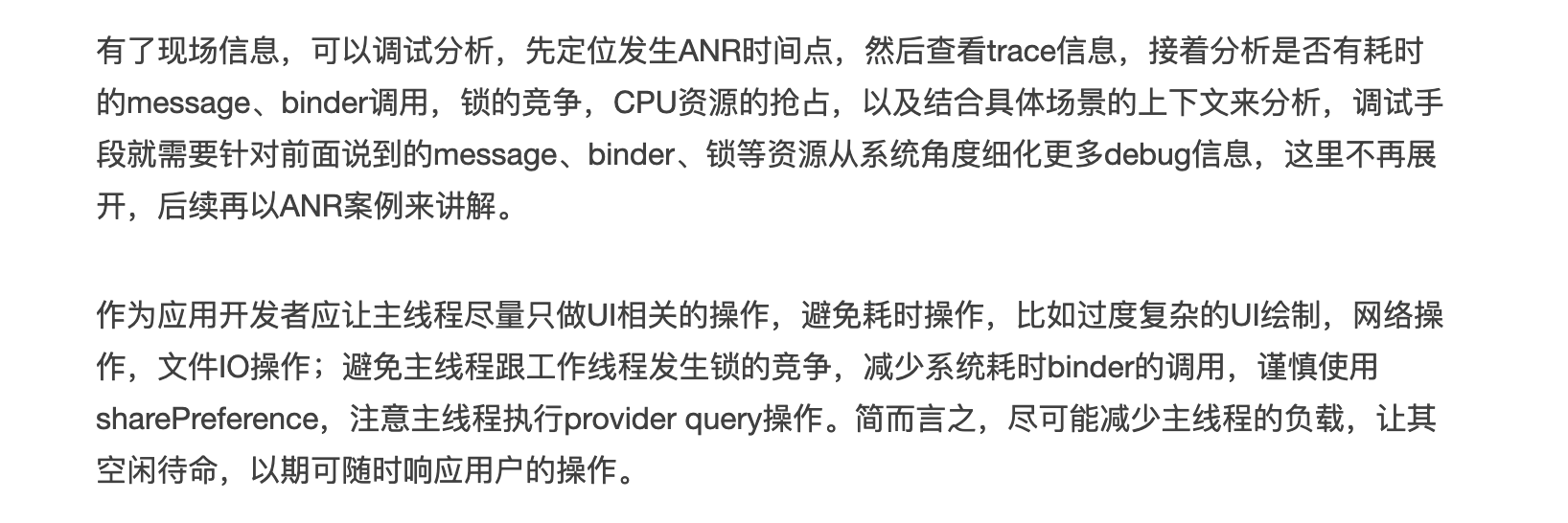
1. 通过FileOberver监听data/anr文件夹目录的Trace文件的新增，发送文件的变化就有ANR发生，然后上传APM平台。
2. 通过监听Looper上的Printer打印来观察Msg执行的耗时，因为主线程Looper在loop()上Dispatch消息也就是处理消息的前后都会进行一次Printer操作。我们可以通过这个日志打印的前后时间来判断ANR，并循环去获取堆栈信息，上报APM。如：BlockCanery，缺陷：不能监听Native层的阻塞，比如设备InputEvent-点击事件。(设置时存在覆盖问题，原方案->动态代理+反射Hook->考虑到反射的影响用静态代理避免覆盖其他SDK的处理)
3. 通过ANR-WatchDog - 本质上是一个Thread，定时间隔向MsgQueue上发送消息，然后睡眠一个interval间隔时间，醒来后如果消息没执行就认定为ANR，发送ANRError。

缺陷：如果间隔期间有非常多Msg处理但每条Msg都没有达到ANR耗时，导致WatchDog 来不及处理，这个也会被WatchDog 认定是ANR。

ANR低端理解就是UI线程上执行耗时操作，高端理解就是组件任务调度超时。ANR类型分4类、分别是Service、Broadcast、Provider、Input。

Input执行耗时消息不一定会引发ANR，关键在他队列之后有没有需要执行的消息，无法调度才会触发ANR。

ANR弹窗是SystemServer所触发的，具体是我们应用进程的所有交互都要与SystemServer进行通信，比如ActivityManagerServer，我们要处理某个时间前都会通知SystemServer，SystemServer就会触发监听机制，启动任务计时，若在规定时间内（service-20s、broadcast-10s、provider-10s、input-5s），就会ANR弹窗，并抓取现场Traces信息。



1. ****ListView与RecyclerView：****
2. ListView缓存对象是View，RecyclerView缓存对象是ViewHolder。
3. ListView有2层缓存（mActiviteViews、mScrapViews），RecyclerView有4层缓存（mAttachedScrap、mCacheViews、mViewCacheExtension、mRecyclerPool(SparseArray<int, arrayList<Holder>>)）-暴露Extension自定义。
4. ListView只有mActiveViews容器的缓存能直接复用不用走getView流程，而RecyclerView的mAttachedScrap容器与mCacheViews容器的两级缓存直接读取ViewHodler复用不用走bindViewHolder流程。

RecyclerView优点：

1. 强制使用ViewHolder，复用机制底层已经封装好，不用像ListView那样自己设置ViewHolder跟Tag。
2. 支持更多的布局样式-线性、网格、瀑布..
3. 更完善的ViewHolder复用机制，通过pos定位Item，一二级缓存(mAttachScrap、mCacheViews)复用是直接复用、无需再走bindView流程。
4. 支持用户自定义extension缓存机制

ListView优化：

1. 设置滑动监听，滑动时不加载网络请求与图片加载
2. 进行convertView的复用，构建ViewHolder复用、通过setTag设置View缓存。
3. 减少ItemLayout的布局层级，尽量避免布局嵌套，减少绘制次数。
4. getView上不进行频繁的IO读写、耗时逻辑、避免创建大对象。
5. 分页加载数据。
6. ****Bitmap优化：****
7. 使用完或不用的时候及时进行recycler与清空引用。
8. 本地资源图片进行WebP压缩。（解码速度慢一点、但图片小加载更快）
9. 加载图片时先计算具体采样率，通过设置Bitmap Options的inJustDecodeBound为true先加载图片宽高信息、计算出采样率(缩放比)再去加载图片。(图片更轻量级)
10. 对于没有透明度的图片优先使用RBG-565格式加载， LRUCache缓存
11. ****glide核心思想****
12. 高效缓存机制，三层缓存：
    * 1. 正在使用的缓存activityResource - Map<Key, WeakRefrence<Resource>>
      2. 本地内存LruCache缓存。
      3. 磁盘缓存DiskLruCache。

取顺序：LruCache - 弱引用缓存activityResource - DiskLruCache

存顺序：网络- 弱引用缓存ActivityResource - LruCache - DiskLruCache

1. Glide默认图片格式是RGB-565。
2. Glide会根据具体ImageView大小进行对于尺寸图片缓存，内存占用更小。
3. Glide支持配置磁盘缓存策略：ALL、NONE、SOURCE、RESULT。
4. ****为什么使用GT？****

测量维度多、比较轻量级、开源暴露SDK、能够自定义-剔除不需要的采集功能、增强核心功能、有具体的Api，接入相关数据，能够生成H5可视化报告、（性能测试点与H5可视化报告）方便引入CI流程里边。（轻量级、功能全、灵活性、可定制、可视化报告）

1. ****启动流程、构建流程、绘制流程各Hook点：****
2. 启动流程：ActivityThread.main()或ActivityThread.attach() -> handleLauncher() / 10前activity.onWindowFocusChanged() / 10后onResume() - 具体看怎么去定义启动。
3. 页面启动耗时：Instrumentation.execAct -> activity.onWindowFocusChanged()
4. 布局构建时间：setContentView或inflate 前后时间。
5. 页面绘制时间：ViewRoot.performTraversal() ->activity.onWindowFocusChanged()
6. 方法耗时：注解 + AspectJ Hook

还可以通过ActivityLifecycle监听Pause->Resume页面启动流程：

application.registerActivityLifecycleCallbacks(new Application.ActivLifecycleCallbacks() {

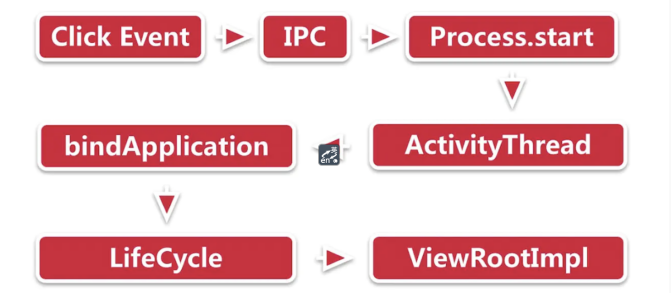
onActivityCreate()....

onActivityPause()....

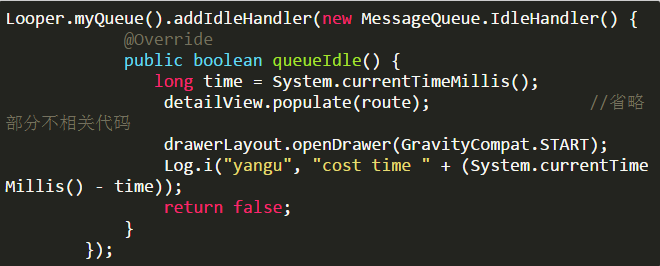
.....

}

# **[Android线上轻量级APM性能监测方案](https://www.jianshu.com/p/be5642884f6b)**



1. ****IdleHandler空闲期消息：****



提供了一种思路，android本身提供的activity框架和fragment框架并没有提供绘制完成的回调（onWindowFocusChanged不十分准确），如果我们自己实现一个框架，就可以使用这个IdleHandler来实现一个onRenderFinished这种回调了。

1. ****Android打印其他进程的Trace堆栈（Process.sendSignal(pid,x)）****

