深入探索Android卡顿优化（下）

转载自：**深入探索Android卡顿优化（下）**

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

在上篇文章中，笔者带领大家学习了卡顿优化分析方法与工具、自动化卡顿检测方案及优化这两块内容。如果对这块内容还不了解的同学建议先看看《深入探索Android卡顿优化（上）》。本篇，为深入探索Android卡顿优化的下篇。这篇文章包含的主要内容如下所示：

- 1、ANR分析与实战
- 2、卡顿单点问题检测方案
- 3、高效实现界面秒开
- 4、优雅监控耗时盲区
- 5、卡顿优化技巧总结
- 6、常见卡顿问题解决方案总结
- 7、卡顿优化的常见问题

卡顿时间过长，一定会造成应用发生ANR。下面，我们就来从应用的ANR分析与实战来开始今天的探索之旅。

一、ANR分析与实战

1、ANR介绍与实战

首先，我们再来回顾一下ANR的几种常见的类型，如下所示：

- 1、KeyDispatchTimeout：按键事件在**5s**的时间内没有处理完成。
- 2、BroadcastTimeout：广播接收器在**前台10s，后台60s**的时间内没有响应完成。
- 3、ServiceTimeout：服务在**前台20s，后台200s**的时间内没有处理完成。

具体的时间定义我们可以在AMS（ActivityManagerService）中找到：

```
1 // How long we allow a receiver to run before giving up on it.
```

```
static final int BROADCAST_FG_TIMEOUT = 10*1000;
static final int BROADCAST_BG_TIMEOUT = 60*1000;
// How long we wait until we timeout on key dispatching.
static final int KEY_DISPATCHING_TIMEOUT = 5*1000;
```

接下来，我们来看一下ANR的执行流程。

ANR执行流程

- 1、首先，我们的应用发生了ANR。
- 2、然后，我们的进程就会接受到异常终止信息，并开始写入进程ANR信息，也就是当时应用的场景信息，它包含了应用所有的堆栈信息、CPU、IO等使用的情况等待。
- 3、最后，会弹出一个ANR提示框，看你是要选择继续等待还是退出应用，需要注意这个ANR提示框不一定会弹出，根据不同ROM，它的表现情况也不同。因为有些手机厂商它会默认去掉这个提示框，以避免带来不好的用户体验。

分析完ANR的执行流程之后，我们来分析下怎样去解决ANR，究竟哪里可以作为我们的一个突破点。

在上面我们说过，当应用发生ANR时，会写入当时发生ANR的场景信息到文件中，**那么，我们可不可以通过这个文件来判断是否发生了ANR呢?**

关于根据ANR log进行ANR问题的排查与解决的方式笔者已经在深入探索Android稳定性优化的第三节ANR优化中讲解过了，这里就不多赘述了。

线上ANR监控方式

在深入探索Android稳定性优化的第三节ANR优化中我说到了使用FileObserver可以监听/data/anr/traces.txt的变化，利用它可以实现线上ANR的监控，但是它有一个致命的缺点，就是高版本ROM需要root权限，解决方案是只能通过海外Google Play服务、国内Hardcoder的方式去规避。但是，这在国内显然是不现实的，那么，有没有更好的实现方式呢?

那就是ANR-WatchDog，下面我就来详细地介绍一下它。

ANR-WatchDog项目地址

ANR-WatchDog是一种非侵入式的ANR监控组件，可以用于线上ANR的监控，接下来，我们就使用ANR-WatchDog来监控ANR。

首先，在我们项目的app/build.gradle中添加如下依赖：

```
implementation 'com.github.anrwatchdog:anrwatchdog:1.4.0'
```

然后，在应用的Application的onCreate方法中添加如下代码启动ANR-WatchDog：

```
new ANRWatchDog().start();
```

可以看到，它的初始化方式非常地简单，同时，它内部的实现也非常简单，整个库只有两个类，一个是ANRWatchDog，另一个是ANRError。

接下来我们来看一下ANRWatchDog的实现方式。

```
/**
```

```
 * A watchdog timer thread that detects when the UI thread has frozen.
 */
public class ANRWatchDog extends Thread {
```

可以看到，ANRWatchDog实际上是继承了Thread类，也就是它是一个线程，对于线程来说，最重要的就是其run方法，如下所示：

```
private static final int DEFAULT_ANR_TIMEOUT = 5000;
private volatile long _tick = 0;
private volatile boolean _reported = false;
private final Runnable _ticker = new Runnable() {
    @Override public void run() {
        _tick = 0;
        _reported = false;
    }
};
@Override
public void run() {
    // 1、首先，将线程命名为|ANR-WatchDog|。
    setName("|ANR-WatchDog|");
    // 2、接着，声明了一个默认的超时间隔时间，默认的值为5000ms。
    long interval = _timeoutInterval;
    // 3、然后，在while循环中通过_uiHandler去post一个_ticker Runnable。
    while (!isInterrupted()) {
        // 3.1 这里的_tick默认是0，所以needPost即为true。
        boolean needPost = _tick == 0;
        // 这里的_tick加上了默认的5000ms
        _tick += interval;
        if (needPost) {
            _uiHandler.post(_ticker);
        }
        // 接下来，线程会sleep一段时间，默认值为5000ms。
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
            _interruptionListener.onInterrupted(e);
            return ;
        }
        // 4、如果主线程没有处理Runnable，即_tick的值没有被赋值为0，则说明发生了ANR，第二个_reported标志位是为了避免重复报道已经处理过的ANR。
```

```
33          if (_tick != 0 && !_reported) {
34              //noinspection ConstantConditions
35              if (!_ignoreDebugger && (Debug.isDebuggerConnected() ||
     Debug.waitingForDebugger())) {
36                  Log.w("ANRWatchdog", "An ANR was detected but ignore
    d because the debugger is connected (you can prevent this with setIg
    noreDebugger(true))");
37                  _reported = true;
38                  continue ;
39              }
40              interval = _anrInterceptor.intercept(_tick);
41              if (interval > 0) {
42                  continue;
43              }
44              final ANRError error;
45              if (_namePrefix != null) {
46                  error = ANRError.New(_tick, _namePrefix, _logThreads
    WithoutStackTrace);
47              } else {
48                  // 5、如果没有主动给ANR_Watchdog设置线程名，则会默认会使用AN
    RError的NewMainOnly方法去处理ANR。
49                  error = ANRError.NewMainOnly(_tick);
50              }
51             // 6、最后会通过ANRListener调用它的onAppNotResponding方法，其默
    认的处理会直接抛出当前的ANRError，导致程序崩溃。 _anrListener.onAppNotRespon
    ding(error);
52              interval = _timeoutInterval;
53              _reported = true;
54          }
55      }
56  }
```

首先，在注释1处，我们将线程命名为了|ANR-WatchDog|。接着，在注释2处，声明了一个默认的超时间隔时间，默认的值为5000ms。然后，注释3处，在while循环中通过_uiHandler去post一个_tickerRunnable。注意这里的_tick默认是0，所以needPost即为true。接下来，线程会sleep一段时间，默认值为5000ms。在注释4处，如果主线程没有处理Runnable，即_tick的值没有被赋值为0，则说明发生了ANR，第二个_reported标志位是为了避免重复报道已经处理过的ANR。如果发生了ANR，就会调用接下来的代码，开始会处理debug的情况，然后，我们看到注释5处，如果没有主动给ANR_Watchdog设置线程名，则会默认会使用ANRError的NewMainOnly方法去处理ANR。ANRError的NewMainOnly方法如下所示：

```
/**
 * The minimum duration, in ms, for which the main thread has been b
locked. May be more.
 */
public final long duration;
static ANRError NewMainOnly(long duration) {
    // 1、获取主线程的堆栈信息
    final Thread mainThread = Looper.getMainLooper().getThread();
    final StackTraceElement[] mainStackTrace = mainThread.getStackTr
ace();
    // 2、返回一个包含主线程名、主线程堆栈信息以及发生ANR的最小时间值的实例。
    return new ANRError(new $(getThreadTitle(mainThread), mainStackT
race).new _Thread(null), duration);
}
```

可以看到，在注释1处，首先获了主线程的堆栈信息，然后返回了一个包含主线程名、主线程堆栈信息以及发生ANR的最小时间值的实例。（我们可以改造其源码在此时添加更多的卡顿现场信息，如CPU 使用率和调度信息、内存相关信息、I/O 和网络相关的信息等等）

接下来，我们再回到ANRWatchDog的run方法中的注释6处，最后这里会通过ANRListener调用它的onAppNotResponding方法，其默认的处理会直接抛出当前的ANRError，导致程序崩溃。对应的代码如下所示：

```
private static final ANRListener DEFAULT_ANR_LISTENER = new ANRListen
er() {
    @Override public void onAppNotResponding(ANRError error) {
        throw error;
    }
};
```

了解了ANRWatchDog的实现原理之后，我们试一试它的效果如何。首先，我们给MainActivity中的悬浮按钮添加主线程休眠10s的代码，如下所示：

```
@OnClick({R.id.main_floating_action_btn})
void onClick(View view) {
    switch (view.getId()) {
        case R.id.main_floating_action_btn:
            try {
                // 对应项目中的第170行
                Thread.sleep(10000);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            jumpToTheTop();
            break;
        default:
            break;
    }
}
```

然后，我们重新安装运行项目，点击悬浮按钮，发现在10s内都不能触发屏幕点击和触摸事件，并且在10s之后，应用直接发生了崩溃。接着，我们在Logcat过滤栏中输入**fatal**关键字，找出致命的错误，log如下所示：

```
2020-01-18 09:55:53.459 29924-29969/? E/AndroidRuntime: FATAL EXCEPTION: |ANR-WatchDog|
Process: json.chao.com.wanandroid, PID: 29924
com.github.anrwatchdog.ANRError: Application Not Responding for at least 5000 ms.
Caused by: com.github.anrwatchdog.ANRError$$$_Thread: main (state = TIMED_WAITING)
    at java.lang.Thread.sleep(Native Method)
    at java.lang.Thread.sleep(Thread.java:373)
    at java.lang.Thread.sleep(Thread.java:314)
    // 1
    at json.chao.com.wanandroid.ui.main.activity.MainActivity.onClick(MainActivity.java:170)
    at json.chao.com.wanandroid.ui.main.activity.MainActivity_ViewBinding$1.doClick(MainActivity_ViewBinding.java:45)
    at butterknife.internal.DebouncingOnClickListener.onClick(DebouncingOnClickListener.java:22)
    at android.view.View.performClick(View.java:6311)
    at android.view.View$PerformClick.run(View.java:24833)
    at android.os.Handler.handleCallback(Handler.java:794)
    at android.os.Handler.dispatchMessage(Handler.java:99)
    at android.os.Looper.loop(Looper.java:173)
    at android.app.ActivityThread.main(ActivityThread.java:6653)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:547)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:821)
```

```
21  Caused by: com.github.anrwatchdog.ANRError$$$_Thread: AndroidFileLo
    gger./storage/emulated/0/Android/data/json.chao.com.wanandroid/log/
     (state = RUNNABLE)
```

可以看到，发生崩溃的线程正是|ANR-WatchDog|。我们重点关注注释1，这里发生崩溃的位置是在MainActivity的onClick方法，对应的行数为170行，从前可知，这里正是线程休眠的地方。
接下来，我们来分析一下ANR-WatchDog的实现原理。

2、ANR-WatchDog原理

- 首先，我们调用了ANR-WatchDog的start方法，然后这个线程就会开始工作。
- 然后，我们通过主线程的Handler post一个消息将主线程的某个值进行一个加值的操作。
- post完成之后呢，我们这个线程就sleep一段时间。
- 在sleep之后呢，它就会来检测我们这个值有没有被修改，如果这个值被修改了，那就说明我们在主线程中执行了这个message，即表明主线程没有发生卡顿，否则，则说明主线程发生了卡顿。
- 最后，ANR-WatchDog就会判断发生了ANR，抛出一个异常给我们。

最后，ANR-WatchDog的工作流程简图如下所示：

上面我们最后说到，如果检测到主线程发生了卡顿，则会抛出一个ANR异常，这将会导致应用崩溃，这显然不能将这种方案带到线上，那么，**有什么方式能够自定义最后发生卡顿时的处理过程吗?**
其实ANR-WatchDog自身就实现了一个我们自身也可以去实现的**ANRListener，通过它，我们就可以对ANR事件去做一个自定义的处理**，比如将堆栈信息压缩后保存到本地，并在适当的时间上传到APM后台。

3、小结

ANR-WatchDog是一种非侵入式的ANR监控方案，它能够弥补我们在高版本中没有权限去读取traces.txt文件的问题，需要注意的是，在线上这两种方案我们是结合使用的。
在之前，我们还讲到了**AndroidPerformanceMonitor，那么它和ANR-WatchDog有什么区别呢?**
对于AndroidPerformanceMonitor来说，它是监控我们主线程中每一个message的执行，它会在主线程的每一个message的前后打印一个时间戳，然后，我们就可以据此计算每一个message的具体执行时间，但是我们需要注意的是一个message的执行时间通常是非常短暂的，也就是很难达到ANR这个级别。然后我们来看看ANR-WatchDog的原理，它是不管应用是如何执行的，它只会看最终的结果，即sleep 5s之后，我就看主线程的这个值有没有被更改。如果说被改过，就说明没有发生ANR，否则，就表明发生了ANR。

根据这两个库的原理，我们便可以判断出它们分别的适用场景，对于AndroidPerformanceMonitor来说，它适合监控卡顿，因为每一个message它执行的时间并不长。对于ANR-WatchDog来说，它更加适合于ANR监控的补充。

此外，虽然ANR-WatchDog解决了在高版本系统没有权限读取 /data/anr/traces.txt 文件的问题，但是在Java层去获取所有线程堆栈以及各种信息非常耗时，对于卡顿场景不一定合适，它可能会进一步加剧用户的卡顿。如果是对性能要求比较高的应用,可以通过Hook Native层的方式去获得所有线程的堆栈信息，具体为如下两个步骤：

- 通过libart.so、dlsym调用ThreadList::ForEach方法，拿到所有的 Native 线程对象。
- 遍历线程对象列表，调用Thread::DumpState方法。

通过这种方式就大致模拟了系统打印 ANR 日志的流程，但是由于采用的是Hook方式，所以可能会产生一些异常甚至崩溃的情况，这个时候就需要通过 fork 子进程方式去避免这种问题，而且使用 子进程去获取堆栈信息的方式可以做到完全不卡住我们主进程。

但是需要注意的是，fork 进程会导致进程号发生改变，此时需要通过指定 /proc/[父进程 id]的方式重新获取应用主进程的堆栈信息。

通过 Native Hook 的 方式我们实现了一套“无损”获取所有 Java 线程堆栈与详细信息的卡顿监控体系。为了降低上报数据量，建议只有主线程的 Java 线程状态是 WAITING、TIME_WAITING 或者 BLOCKED 的时候，才去使用这套方案。

二、卡顿单点问题检测方案

除了自动化的卡顿与ANR监控之外，我们还需要进行卡顿单点问题的检测，因为上述两种检测方案的并不能满足所有场景的检测要求，这里我举一个小栗子：

```
比如我有很多的message要执行，但是每一个message的执行时间
都不到卡顿的阈值，那自动化卡顿检测方案也就不能够检测出卡
顿，但是对用户来说，用户就觉得你的App就是有些卡顿。
```

除此之外，为了建立体系化的监控解决方案，我们就必须在上线之前将问题尽可能地暴露出来。

1、IPC单点问题检测方案

常见的单点问题有主线程IPC、DB操作等等，这里我就拿主线程IPC来说，因为IPC其实是一个很耗时的操作，但是在实际开发过程中，我们可能对IPC操作没有足够的重视，所以，我们经常在主程序中去做频繁IPC操作，所以说，这种耗时它可能并不到你设定卡顿的一个阈值，接下来，我们看一下，对于IPC问题，我们应该去监测哪些指标。

- 1、IPC调用类型：如PackageManager、TelephoneManager的调用。
- 2、每一个的调用次数与耗时。
- 3、IPC的调用堆栈（表明哪行代码调用的）、发生线程。

常规方案

常规方案就是在IPC的前后加上埋点。但是，这种方式不够优雅，而且，在平常开发过程中我们经常忘记某个埋点的真正用处，同时它的维护成本也非常大。

接下来，我们讲解一下IPC问题监测的技巧。

IPC问题监测技巧

在线下，我们可以通过adb命令的方式来进行监测，如下所示：

```
// 1、首先，对IPC操作开始进行监控
adb shell am trace-ipc start
// 2、然后，结束IPC操作的监控，同时，将监控到的信息存放到指定的文件当中
adb shell am trace-ipc stop -dump-file /data/local/tmp/ipc-trace.txt
// 3、最后，将监控到的ipc-trace导出到电脑查看
adb pull /data/local/tmp/ipc-trace.txt
```

然后，这里我们介绍一种优雅的实现方案，看过深入探索Android布局优化（上）的同学可能知道这里的实现方案无非就是ARTHook或AspectJ这两种方案，这里我们需要去监控IPC操作，那么，我们应该选用哪种方式会更好一些呢?

要回答这个问题，就需要我们对ARTHook和AspectJ这两者的思想有足够的认识，对应ARTHook来说，其实我们可以用它来去Hook系统的一些方法，因为对于系统代码来说，我们无法对它进行更改，但是我们可以Hook住它的一个方法，在它的方法体里面去加上自己的一些代码。但是，对于AspectJ来说，它只能针对于哪些非系统方法，也就是我们App自己的源码，或者是我们所引用到的一些jar、aar包。因为AspectJ实际上是往我们的具体方法里面插入相对于的代码，所以说，他不能够针对于我们的系统方法去做操作，在这里，我们就需要采用ARTHook的方式去进行IPC操作的监控。

在使用ARTHook去监控IPC操作之前，我们首先思考一下，哪些操作是IPC操作呢?

比如说，我们通过PackageManager去拿到我们应用的一些信息，或者去拿到设备的DeviceId这样的信息以及AMS相关的信息等等，这些其实都涉及到了IPC的操作，而这些操作都会通过固定的方式进行IPC，并最终会调用到**android.os.BinderProxy**，接下来，我们来看看它的transact方法，如下所示：

```
public boolean transact(int code, Parcel data, Parcel reply, int flag
s) throws RemoteException {
```

这里我们仅仅关注transact方法的参数即可，第一个参数是一个行动编码，为int类型，它是在FIRST_CALL_TRANSACTION与LAST_CALL_TRANSACTION之间的某个值，第二、三个参数都是Parcel类型的参数，用于获取和回复相应的数据，第四个参数为一个int类型的标记值，为0表示一个正常的IPC调用，否则表明是一个单向的IPC调用。然后，我们在项目中的Application的onCreate方法中使用ARTHook对android.os.BinderProxy类的transact方法进行Hook，代码如下所示：

```
try {
        DexposedBridge.findAndHookMethod(Class.forName("android.os.B
inderProxy"), "transact",
                int.class, Parcel.class, Parcel.class, int.class, ne
w XC_MethodHook() {
                    @Override
                    protected void beforeHookedMethod(MethodHookPara
```

```
m param) throws Throwable {
                        LogHelper.i( "BinderProxy beforeHookedMethod
" + param.thisObject.getClass().getSimpleName()
                                + "\n" + Log.getStackTraceString(new
Throwable()));
                        super.beforeHookedMethod(param);
                    }
                });
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
```

重新安装应用，即可看到如下的Log信息：

```
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: | WanAndroidApp$1.beforeHookedMethod  (WanAndroidApp.jav
a:160)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |    LogHelper.i  (LogHelper.java:37)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: ├──────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: | [WanAndroidApp.java | 160 | beforeHookedMethod] BinderP
roxy beforeHookedMethod BinderProxy
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: | java.lang.Throwable
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at json.chao.com.wanandroid.app.WanAndroidApp$1.bef
oreHookedMethod(WanAndroidApp.java:160)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at com.taobao.android.dexposed.DexposedBridge.handl
eHookedArtMethod(DexposedBridge.java:237)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at me.weishu.epic.art.entry.Entry64.onHookBoolean(E
ntry64.java:72)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at me.weishu.epic.art.entry.Entry64.referenceBridge
(Entry64.java:237)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
```

```
droid-LOG: |     at me.weishu.epic.art.entry.Entry64.booleanBridge(E
ntry64.java:86)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.os.ServiceManagerProxy.getService(Servic
eManagerNative.java:123)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.os.ServiceManager.getService(ServiceMana
ger.java:56)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.os.ServiceManager.getServiceOrThrow(Serv
iceManager.java:71)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.UiModeManager.<init>(UiModeManager.j
ava:127)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.SystemServiceRegistry$42.createServi
ce(SystemServiceRegistry.java:511)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.SystemServiceRegistry$42.createServi
ce(SystemServiceRegistry.java:509)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.SystemServiceRegistry$CachedServiceF
etcher.getService(SystemServiceRegistry.java:970)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.SystemServiceRegistry.getSystemServi
ce(SystemServiceRegistry.java:920)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.ContextImpl.getSystemService(Context
Impl.java:1677)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.view.ContextThemeWrapper.getSystemServic
e(ContextThemeWrapper.java:171)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.app.Activity.getSystemService(Activity.j
ava:6003)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.support.v7.app.AppCompatDelegateImplV23.
<init>(AppCompatDelegateImplV23.java:33)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAn
droid-LOG: |     at android.support.v7.app.AppCompatDelegateImplN.<i
```

```
nit>(AppCompatDelegateImplN.java:31)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.support.v7.app.AppCompatDelegate.create(AppCompatDelegate.java:198)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.support.v7.app.AppCompatDelegate.create(AppCompatDelegate.java:183)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.support.v7.app.AppCompatActivity.getDelegate(AppCompatActivity.java:519)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.support.v7.app.AppCompatActivity.onCreate(AppCompatActivity.java:70)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at me.yokeyword.fragmentation.SupportActivity.onCreate(SupportActivity.java:38)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at json.chao.com.wanandroid.base.activity.AbstractSimpleActivity.onCreate(AbstractSimpleActivity.java:29)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at json.chao.com.wanandroid.base.activity.BaseActivity.onCreate(BaseActivity.java:37)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.Activity.performCreate(Activity.java:7098)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.Activity.performCreate(Activity.java:7089)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1215)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2770)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2895)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.ActivityThread.-wrap11(Unknown Source:0)
```

```
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1616)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.os.Handler.dispatchMessage(Handler.java:106)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.os.Looper.loop(Looper.java:173)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at android.app.ActivityThread.main(ActivityThread.java:6653)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at java.lang.reflect.Method.invoke(Native Method)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:547)
2020-01-22 19:52:47.657 10683-10683/json.chao.com.wanandroid I/WanAndroid-LOG: |     at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:821)
```

可以看出，这里弹出了应用中某一个IPC调用的所有堆栈信息。在这里，具体是在AbstractSimpleActivity的onCreate方法中调用了ServiceManager的getService方法，它是一个IPC调用的方法。这样，应用的IPC调用我们就能很方便地捕获到了。

大家可以看到，通过这种方式我们可以很方便地拿到应用中所有的IPC操作，并可以获得到IPC调用的类型、调用耗时、发生次数、调用的堆栈等等一系列信息。当然，除了IPC调用的问题之外，还有IO、DB、View绘制等一系列单点问题需要去建立与之对应的检测方案。

2、卡顿问题检测方案

对于卡顿问题检测方案的建设，主要是利用ARTHook去完善线下的检测工具，尽可能地去Hook相对应的操作，以暴露、分析问题。这样，才能更好地实现卡顿的体系还解决方案。

三、如何实现界面秒开?

界面的打开速度对用户体验来说是至关重要的，那么如何实现界面秒开呢?

其实界面秒开就是一个小的启动优化，其优化的思想可以借鉴启动速度优化与布局优化的一些实现思路。

1、界面秒开实现

首先，我们可以通过Systrace来观察CPU的运行状况，比如有没有跑满CPU；然后，我们在启动优化中学习到的优雅异步以及优雅延迟初始化等等一些方案；其次，针对于我们的界面布局，我们可以使用异步Inflate、X2C、其它的绘制优化措施等等；最后，我们可以使用预加载的方式去提前获取页面的数据，以避免网络或磁盘IO速度的影响，或者也可以将获取数据的方法放到onCreate方法的第一行。

那么我们如何去衡量界面的打开速度呢?

通常，我们是通过界面秒开率去统计页面的打开速度的，具体就是计算onCreate到onWindowFocusChanged的时间。当然，在某些特定的场景下，把onWindowFocusChanged作为页面打开的结束点并不是特别的精确，那我们可以去实现一个特定的接口来适配我们的Activity或Fragment，我们可以把那个接口方法作为页面打开的结束点。

那么，除了以上说到的一些界面秒开的实现方式之外，还没有更好的方式呢?

那就是Lancet。

2、Lancet

Lancet是一个轻量级的Android AOP框架，它具有如下优势：

- 1、编译速度快，支持增量编译。
- 2、API简单，没有任何多余代码插入apk。（这一点对应包体积优化时至关重要的）

然后，我来简单地讲解下Lancet的用法。Lancet自身提供了一些注解用于Hook，如下所示：

- @Prxoy：通常是用于对系统API调用的Hook。
- @Insert：经常用于操作App或者是Library当中的一些类。

接下来，我们就是使用Lancet来进行一下实战演练。

首先，我们需要在项目根目录的 build.gradle 添加如下依赖:

```
dependencies{
    classpath 'me.ele:lancet-plugin:1.0.5'
}
```

然后，在 app 目录的’build.gradle’ 添加：

```
apply plugin: 'me.ele.lancet'
dependencies {
    compileOnly 'me.ele:lancet-base:1.0.5'
}
```

接下来，我们就可以使用Lancet了，这里我们需要先新建一个类去进行专门的Hook操作，如下所示：

```
public class ActivityHooker {
    @Proxy("i")
    @TargetClass("android.util.Log")
    public static int i(String tag, String msg) {
        msg = msg + "JsonChao";
        return (int) Origin.call();
    }
}
```

上述的方法就是对android.util.Log的i方法进行Hook，并在所有的msg后面加上”JsonChao”字符串，注意这里的i方法我们需要从android.util.Log里面将它的i方法复制过来，确保方法名和对应的参数信息一致；然后，方法上面的@TargetClass与@Proxy分别是指定对应的全路径类名与方法名；最后，我们需要通过Lancet提供的Origin类去调用它的call方法来实现返回原来的调用信息。完成之后，我们重新运行项目，会出现如下log信息：

```
2020-01-23 13:13:34.124 7277-7277/json.chao.com.wanandroid I/MultiDex: VM with version 2.1.0 has multidex supportJsonChao
2020-01-23 13:13:34.124 7277-7277/json.chao.com.wanandroid I/MultiDex: Installing applicationJsonChao
```

可以看到，log后面都加上了我们预先添加的字符串，说明Hook成功了。下面，我们就可以用Lancet来统计一下项目界面的秒开率了，代码如下所示：

```
public static ActivityRecord sActivityRecord;
static {
    sActivityRecord = new ActivityRecord();
}
@Insert(value = "onCreate",mayCreateSuper = true)
@TargetClass(value = "android.support.v7.app.AppCompatActivity",scope = Scope.ALL)
protected void onCreate(Bundle savedInstanceState) {
    sActivityRecord.mOnCreateTime = System.currentTimeMillis();
    // 调用当前Hook类方法中原先的逻辑
    Origin.callVoid();
}
@Insert(value = "onWindowFocusChanged",mayCreateSuper = true)
@TargetClass(value = "android.support.v7.app.AppCompatActivity",scope = Scope.ALL)
public void onWindowFocusChanged(boolean hasFocus) {
    sActivityRecord.mOnWindowsFocusChangedTime = System.currentTimeMillis();
    LogHelper.i(getClass().getCanonicalName() + " onWindowFocusChanged cost "+(sActivityRecord.mOnWindowsFocusChangedTime - sActivityRecord.mOnCreateTime));
    Origin.callVoid();
}
```

上面，我们通过@TargetClass和@Insert两个注解实现Hook了android.support.v7.app.AppCompatActivity的onCreate与onWindowFocusChanged方法。我们注意到，这里@Insert注解可以指定两个参数，其源码如下所示：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Insert {
    String value();
    boolean mayCreateSuper() default false;
}
```

第二个参数mayCreateSuper设定为true则表明如果没有重写父类的方法，则会默认去重写这个方法。对应到我们ActivityHooker里面实现的@Insert注解方法就是如果当前的Activity没有重写父类的onCreate和onWindowFocusChanged方法，则此时默认回去重写父类的这个方法，以避免因某些Activity不存在该方法而Hook失败的情况。

然后，我们注意到@TargetClass也可以指定两个参数，其源码如下所示：

```
@Retention(RetentionPolicy.RUNTIME)
@java.lang.annotation.Target({ElementType.TYPE, ElementType.METHOD})
public @interface TargetClass {
    String value();
    Scope scope() default Scope.SELF;
}
```

第二个参数scope指定的值是一个枚举，可选的值如下所示：

```
public enum Scope {
    SELF,
    DIRECT,
    ALL,
    LEAF
}
```

对于Scope.SELF，它代表仅匹配目标value所指定的一个匹配类；对于DIRECT，它代表匹配value所指定的类的一个直接子类；如果是Scope.ALL，它就表明会去匹配value所指定的类的所有子类，而我们上面指定的value值为android.support.v7.app.AppCompatActivity，因为scope指定为了Scope.ALL，则说明会去匹配AppCompatActivity的所有子类。而最后的Scope.LEAF 代表匹配 value 指定类的最终子类，因为java是单继承，所以继承关系是树形结构，所以这里代表了指定类为顶点的继承树的所有叶子节点。

最后，我们设定了一个ActivityRecord类去记录onCreate与onWindowFocusChanged的时间戳，如下所示：

```
public class ActivityRecord {
    /**
    * 避免没有仅执行onResume就去统计界面打开速度的情况，如息屏、亮屏等等
    */
```

```
    public boolean isNewCreate;
    public long mOnCreateTime;
    public long mOnWindowsFocusChangedTime;
}
```

通过sActivityRecord.mOnWindowsFocusChangedTime – sActivityRecord.mOnCreateTime得到的时间即为界面的打开速度，最后，重新运行项目，会得到如下log信息：

```
2020-01-23 14:12:16.406 15098-15098/json.chao.com.wanandroid I/WanAndroid-LOG: | [null | 57 | json_chao_com_wanandroid_aop_ActivityHooker_onWindowFocusChanged] json.chao.com.wanandroid.ui.main.activity.SplashActivity onWindowFocusChanged cost 257
2020-01-23 14:12:18.930 15098-15098/json.chao.com.wanandroid I/WanAndroid-LOG: | [null | 57 | json_chao_com_wanandroid_aop_ActivityHooker_onWindowFocusChanged] json.chao.com.wanandroid.ui.main.activity.MainActivity onWindowFocusChanged cost 608
```

从上面的log信息，我们就可以知道 SplashActivity 和 MainActivity 的界面打开速度分别是257ms和608ms。

最后，我们来看下界面秒开的监控纬度。

3、界面秒开监控纬度

对于界面秒开的监控纬度，主要分为以下三个方面：

- 总体耗时
- 生命周期耗时
- 生命周期间隔耗时

首先，我们会监控界面打开的整体耗时，也就是onCreate到onWindowFocusChanged这个方法的耗时；当然，如果我们是在一个特殊的界面，我们需要更精确的知道界面打开的一个时间，这个我们可以用自定义的接口去实现。其次，我们也需要去监控生命周期的一个耗时，如onCreate、onStart、onResume等等。最后，我们也需要去做生命周期间隔的耗时监控，这点经常被我们所忽略，比如onCreate的结束到onStart开始的这一段时间，也是有时间损耗的，我们可以监控它是不是在一个合理的范围之内。通过这三个方面的监控纬度，我们就能够非常细粒度地去检测页面秒开各个方面的情况。

四、优雅监控耗时盲区

尽管我们在应用中监控了很多的耗时区间，但是还是有一些耗时区间我们还没有捕捉到，如onResume到列表展示的间隔时间，这些时间在我们的统计过程中很容易被忽视，这里我们举一个小栗子：

```
我们在Activity的生命周期中post了一个message，那这个message很可能其中执行了一段耗时操作，那你知道这个message它的具体执行时间吗？这个message其实很有可能在列表展示之前就执行了，如果这个message耗时1s，那么列表的展示时间就会延迟1s，如果是200ms，那么我们设定的自动化卡顿检测就无法发现它，那么列表的展示时间就会延迟200ms。
```

其实这种场景非常常见，接下来，我们就在项目中来进行实战演练。
首先，我们在MainActivity的onCreate中加上post消息的一段代码，其中模拟了延迟1000ms的耗时操作，代码如下所示：

```
// 以下代码是为了演示Msg导致的主线程卡顿
    new Handler().post(() -> {
        LogHelper.i("Msg 执行");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
```

接着，我们在RecyclerView对应的Adapter中将列表展示的时间打印出来，如下所示：

```
if (helper.getLayoutPosition() == 1 && !mHasRecorded) {
         mHasRecorded = true;
         helper.getView(R.id.item_search_pager_group).getViewTreeObse
rver().addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener()
{
             @Override
             public boolean onPreDraw() {
                 helper.getView(R.id.item_search_pager_group).getView
TreeObserver().removeOnPreDrawListener(this);
                 LogHelper.i("FeedShow");
                 return true;
             }
         });
     }
```

最后，我们重新运行下项目，看看两者的执行时间，log信息如下：

```
2020-01-23 15:21:55.076 19091-19091/json.chao.com.wanandroid I/WanAnd
roid-LOG: ｜ [MainActivity.java | 108 | lambda$initEventAndData$1$Main
Activity] Msg 执行
2020-01-23 15:21:56.264 19091-19091/json.chao.com.wanandroid I/WanAnd
roid-LOG: ｜ [null | 57 | json_chao_com_wanandroid_aop_ActivityHooker_
onWindowFocusChanged] json.chao.com.wanandroid.ui.main.activity.MainA
ctivity onWindowFocusChanged cost 1585
2020-01-23 15:21:57.207 19091-19091/json.chao.com.wanandroid I/WanAnd
```

```
roid-LOG: │ ArticleListAdapter$1.onPreDraw  (ArticleListAdapter.java:
93)
4 2020-01-23 15:21:57.208 19091-19091/json.chao.com.wanandroid I/WanAnd
roid-LOG: │    LogHelper.i  (LogHelper.java:37)
5 2020-01-23 15:21:57.208 19091-19091/json.chao.com.wanandroid I/WanAnd
roid-LOG: ├──────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────
6 2020-01-23 15:21:57.208 19091-19091/json.chao.com.wanandroid I/WanAnd
roid-LOG: │ [ArticleListAdapter.java | 93 | onPreDraw] FeedShow
```

从log信息中可以看到，MAinActivity的onWindowFocusChanged方法延迟了1000ms才被调用，与此同时，列表页时延迟了1000ms才展示出来。也就是说，post的这个message消息是执行在界面、列表展示之前的。因为任何一个开发都有可能在某一个生命周期或者是某一个阶段以及一些第三方的SDK里面，回去做一些handler post的相关操作，这样，他的handler post的message的执行，很有可能在我们的界面或列表展示之前就被执行，所以说，出现这种耗时的盲区是非常普遍的，而且也不好排查，下面，我们分析下耗时盲区存在的难点。

1、耗时盲区监控难点

首先，我们可以通过细化监控的方式去获取耗时的一些盲区，但是我们却不知道在这个盲区中它执行了什么操作。其次，对于线上的一些耗时盲区，我们是无法进行排查的。
这里，我们先来看看如何建立耗时盲区监控的线下方案。

2、耗时盲区监控线下方案

这里我们直接使用TraceView去检测即可，因为它能够清晰地记录线程在具体的时间内到底做了什么操作，特别适合一段时间内的盲区监控。
然后，我们来看下如何建立耗时盲区监控的线上方案。

3、耗时盲区监控线上方案

我们知道主线程的所有方法都是通过message来执行的，还记得在之前我们学习了一个库：AndroidPerformanceMonitor，我们是否可以通过这个mLogging来做盲区检测呢？通过这个mLogging确实可以知道我们主线程发生的message，但是通过mLogging无法获取具体的调用栈信息，因为它所获取的调用栈信息都是系统回调回来的，它并不知道当前的message是被谁抛出来的，所以说，这个方案并不够完美。
那么，我们是否可以通过AOP的方式去切Handler方法呢？比如sendMessage、sendMessageDeleayd方法等等，这样我们就可以知道发生message的一个堆栈，但是这种方案也存在着一个问题，就是它不清楚准确的执行时间，我们切了这个handler的方法，仅仅只知道它具体是在那个地方被发的和它所对应的堆栈信息，但是无法获取准确的执行时间。如果我们想知道在onResume到列表展示之间执行了哪些message，那么通过AOP的方式也无法实现。
那么，最终的耗时盲区监控的一个线上方案就是使用一个统一的Handler，定制了它的两个方法，一个是sendMessageAtTime，另外一个是dispatchMessage方法。因为对于发送message，不管调用哪个方法最终都会调用到一个是sendMessageAtTime这个方法，而处理message呢，它最终会调用

dispatchMessage方法。然后，我们需要定制一个gradle插件，来实现自动化的接入我们定制好的handler，通过这种方式，我们就能在编译期间去动态地替换所有使用Handler的父类为我们定制好的这个handler。这样，在整个项目中，所有的sendMessage和handleMessage都会经过我们的回调方法。接下来，我们来进行一下实战演练。

首先，我这里给出定制好的全局Handler类，如下所示：

```
public class GlobalHandler extends Handler {
    private long mStartTime = System.currentTimeMillis();
    public GlobalHandler() {
        super(Looper.myLooper(), null);
    }
    public GlobalHandler(Callback callback) {
        super(Looper.myLooper(), callback);
    }
    public GlobalHandler(Looper looper, Callback callback) {
        super(looper, callback);
    }
    public GlobalHandler(Looper looper) {
        super(looper);
    }
    @Override
    public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
        boolean send = super.sendMessageAtTime(msg, uptimeMillis);
        // 1
        if (send) {
            GetDetailHandlerHelper.getMsgDetail().put(msg, Log.getSt
ackTraceString(new Throwable()).replace("java.lang.Throwable", ""));
        }
        return send;
    }
    @Override
    public void dispatchMessage(Message msg) {
        mStartTime = System.currentTimeMillis();
        super.dispatchMessage(msg);
        if (GetDetailHandlerHelper.getMsgDetail().containsKey(msg)
            && Looper.myLooper() == Looper.getMainLooper()) {
            JSONObject jsonObject = new JSONObject();
            try {
                // 2
```

```
33                 jsonObject.put("Msg_Cost", System.currentTimeMillis
   () - mStartTime);
34                 jsonObject.put("MsgTrace", msg.getTarget() + " " + G
   etDetailHandlerHelper.getMsgDetail().get(msg));
35                 // 3
36                 LogHelper.i("MsgDetail " + jsonObject.toString());
37                 GetDetailHandlerHelper.getMsgDetail().remove(msg);
38             } catch (Exception e) {
39             }
40         }
41     }
42 }
```

上面的GlobalHandler将会是我们项目中所有Handler的一个父类。在注释1处，我们在sendMessageAtTime这个方法里面判断如果message发送成功，将会把当前message对象对应的调用栈信息都保存到一个ConcurrentHashMap中，GetDetailHandlerHelper类的代码如下所示：

```
1 public class GetDetailHandlerHelper {
2     private static ConcurrentHashMap<Message, String> sMsgDetail = ne
  w ConcurrentHashMap<>();
3     public static ConcurrentHashMap<Message, String> getMsgDetail() {
4         return sMsgDetail;
5     }
6 }
```

这样，我们就能够知道这个message它是被谁发送过来的。然后，在dispatchMessage方法里面，我们可以计算拿到其处理消息的一个耗时，并在注释2处将这个耗时保存到一个jsonObject对象中，同时，我们也可以通过GetDetailHandlerHelper类的ConcurrentHashMap对象拿到这个message对应的堆栈信息，并在注释3处将它们输出到log控制台上，当然，如果是线上监控，则会把这些信息保存到本地，然后选择合适的时间去上传。最后，我们还可以在方法体里面做一个判断，我们设置一个阈值，比如阈值为20ms，超过了20ms就把这些保存好的信息上报到APM后台。

在前面的实战演练中，我们使用了handler post的方式去发送一个消息，通过gradle插件将所有handler的父类替换为我们定制好的GlobalHandler之后，我们就可以优雅地去监控应用中的耗时盲区了。

对于实现全局替换handler的gradle插件，除了使用AspectJ实现之外，这里推荐一个已有的项目：DroidAssist。

然后，重新运行项目，关键的log信息如下所示：

```
1 MsgDetail {"Msg_Cost":1001,"MsgTrace":"Handler (com.json.chao.com.wan
  android.performance.handler.GlobalHandler) {b0d4d48} \n\tat
2 com.json.chao.com.wanandroid.performance.handler.GlobalHandler.sendMe
  ssageAtTime(GlobalHandler.java:36)\n\tat
```

```
3 json.chao.com.wanandroid.ui.main.activity.MainActivity.initEventAndDa
  ta$__twin__(MainActivity.java:107)\n\tat"
```

从以上信息我们不仅可以知道message执行的时间，还可以从对应的堆栈信息中得到发送message的位置，这里的位置是MainActivity的107行，也就是new Handler().post()这一行代码。使用这种方式我们就可以知道在列表展示之前到底执行了哪些自定义的message，我们一眼就可以知道哪些message其实是不符合我们预期的，比如说message的执行时间过长，或者说这个message其实可以延后执行，这个我们都可以根据实际的项目和业务需求进行相应地修改。

4、耗时盲区监控方案总结

耗时盲区监控是我们卡顿监控中不可或缺的一个环节，也是卡顿监控全面性的一个重要保障。而需要注意的是，TraceView仅仅适用于线下的一个场景，同时对于TraceView来说，它可以用于监控我们系统的message。而最后介绍的动态替换的方式其实是适合于线上的，同时，它只有应用自身的一个message。

五、卡顿优化技巧总结

1、卡顿优化实践经验

如果应用出现了卡顿现象，那么可以考虑以下方式进行优化：

- 首先，对于耗时的操作，我们可以考虑异步或延迟初始化的方式，这样可以解决大多数的问题。但是，大家一定要注意代码的优雅性。
- 对于布局加载优化，可以采用AsyncLayoutInflater或者是X2C的方式来优化主线程IO以及反射导致的消耗，同时，需要注意，对于重绘问题，要给与一定的重视。
- 此外，内存问题也可能会导致应用界面的卡顿，我们可以通过降低内存占用的方式来减少GC的次数以及时间，而GC的次数和时间我们可以通过log查看。

然后，我们来看看卡顿优化的工具建设。

2、卡顿优化工具建设

工具建设这块经常容易被大家所忽视，但是它的收益却非常大，也是卡顿优化的一个重点。首先，对于系统工具而言，我们要有一个认识，同时一定要学会使用它，这里我们再回顾一下。

- 对于Systrace来说，我们可以很方便地看出来它的CPU使用情况。另外，它的开销也比较小。
- 对于TraceView来说，我们可以很方便地看出来每一个线程它在特定的时间内做了什么操作，但是TraceView它的开销相对比较大，有时候可能会被带偏优化方向。
- 同时，需要注意，StrictMode也是一个非常强大的工具。

然后，我们介绍了自动化工具建设以及优化方案。我们介绍了两个工具，AndroidPerformanceMonitor以及ANR-WatchDog。同时针对于AndroidPerformanceMonitor的问题，我们采用了高频采集，以找出重复率高的堆栈这样一种方式进行优化，在学习的过程中，我们不仅需要学会怎样去使用工具，更要去理解它们的实现原理以及各自的使用场景。

同时，我们对于卡顿优化工具的建设也做了细化，对于单点问题，比如说IPC监控，我们通过AOP或者是Hook的手段来做到尽早的发现问题。对于耗时盲区的监控，我们在线上采用的是替换Handler的方式来监控所有子线程message执行的耗时以及调用堆栈。

最后，我们来看一下卡顿监控的指标。我们会计算应用整体的卡顿率，ANR率、界面秒开率以及交换时间、生命周期时间等等。在上报ANR信息的同时，我们也需要上报环境和场景信息，这样不仅方便我们在不同版本之家进行横向对比，同时，也可以结合我们的报警平台在第一时间感知到异常。

六、常见卡顿问题解决方案总结

1、CPU资源争抢引发的卡顿问题如何解决?

此时，我们的应用不仅应该控制好核心功能的CPU消耗，也需要尽量减少非核心需求的CPU消耗。

2、要注意Android Java中提供的哪些低效的API?

比如List.removeall方法，它内部会遍历一次需要过滤的消息列表，在已经存在循环列表的情况下会造成CPU资源的冗余使用，此时应该去优化相关的算法，避免使用List.removeall这个方法。

3、如何减少图形处理的CPU消耗?

这个时候我们需要使用神器renderscript来图形处理的相关运算，将CPU转换到GPU。关于renderscript的背景知识可以看看笔者之前写的深入探索Android布局优化（下）。

4、硬件加速长中文字体渲染时造成的卡顿如何解决?

此时只能关闭文本TextView的硬件加速，如下所示：

```
1 textView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

当开启了硬件加速进行长中文字体的渲染时，首先会调用ViewRootImpl.draw()方法，最后会调用GLES20Canvas.nDrawDisplayList()方法开始通过JNI调整到Native层。在这个方法里，会继续调用OpenGLRenderer.drawDisplayList()方法，它通过调用DisplayList的replay方法，以回放前面录制的DisplayList执行绘制操作。

DisplayList的replay方法会遍历DisplayList中保存的每一个操作。其中渲染字体的操作名是DrawText，当遍历到一个DrawText操作时，会调用OpenGLRender::drawText方法区渲染字体。最终，会在OpenGLRender::drawText方法里去调用Font::render()方法渲染字体，而在这个方法中有一个很关键的操作，即获取字体缓存。我们都知道每一个中文的编码都是不同的，因此中文的缓存效果非常不理想，但是对于英文而言，只需要缓存26个字母就可以了。在Android 4.1.2版本之前对文本的Buffer设置过小，所以情况比较严重，如果你的应用在其它版本的渲染性能尚可，就可以仅仅把Android 4.0.x的硬件加速关闭，代码如下所示：

```
1 // AndroidManifest中
2 <Applicaiton
3         ...
4         android:hardwareAccelerated="@bool/hardware_acceleration">
5 // value-v14、value-v15中设置相应的Bool
6 值即可
7 <bool name="hardware_acceleration">false</bool>
```

此外，硬件渲染还有一些其它的问题在使用时需要注意，具体为如下所示：

- 1、在软件渲染的情况下，如果需要重绘某个父View的所有子View，只需要调用这个Parent View的invalidate()方法即可，但如果开启了硬件加速，这么做是行不通的，需要遍历整个子View并调用invalidate()。
- 2、在软件渲染的情况下，会常常使用Bitmap重用的方式来节省内存，但是如果开启了硬件加速，这将会无效。
- 3、当开启硬件加速的UI在前台运行时，需要耗费额外的内存。当硬件加速的UI切换到后台时，上述额外内存有可能不会释放，这大多存在于Android 4.1.2版本中。
- 4、长或宽大于2048像素的Bitmap无法绘制，显示为一片透明。原因是OpenGL的材质大小上限为2048*2048，因此对于超过2048像素的Bitmap，需要将其切割成2048*2048以内的图片块，最后在显示的时候拼起来。
- 5、当UI中存在过渡绘制时，可能会发生花屏，一般来说绘制少于5层不会出现花屏现象，如果有大块红色区域就要十分小心了。
- 6、需要注意，关于LAYER_TYPE_SOFTWARE，虽然无论在App打开硬件加速或没有打开硬件加速的时候，都会通过软件绘制Bitmap作为离屏缓存，但区别在于打开硬件加速的时候，Bitmap最终还会通过硬件加速方式drawDisplayList去渲染这个Bitmap。

七、卡顿优化的常见问题

1、你是怎么做卡顿优化的?

从项目的初期到壮大期，最后再到成熟期，每一个阶段都针对卡顿优化做了不同的处理。各个阶段所做的事情如下所示：

- 1、系统工具定位、解决
- 2、自动化卡顿方案及优化
- 3、线上监控及线下监测工具的建设

我做卡顿优化也是经历了一些阶段，最初我们的项目当中的一些模块出现了卡顿之后，我是通过系统工具进行了定位，我使用了Systrace，然后看了卡顿周期内的CPU状况，同时结合代码，对这个模块进行了重构，将部分代码进行了异步和延迟，在项目初期就是这样解决了问题。但是呢，随着我们项目的扩大，线下卡顿的问题也越来越多，同时，在线上，也有卡顿的反馈，但是线上的反馈卡顿，我们在线下难以复现，于是我们开始寻找自动化的卡顿监测方案，其思路是来自于Android的消息处理机制，主线程执行任何代码都会回到Looper.loop方法当中，而这个方法中有一个mLogging对象，它会在每个message的执行前后都会被调用，我们就是利用这个前后处理的时机来做到的自动化监测方案的。同时，在这个阶段，我们也完善了线上ANR的上报，我们采取的方式就是监控ANR的信息，同时结合了ANR-WatchDog，作为高版本没有文件权限的一个补充方案。在做完这个卡顿检测方案之后呢，我们还做了线上监控及线下检测工具的建设，最终实现了一整套完善，多维度的解决方案。

2、你是怎么样自动化的获取卡顿信息?

我们的思路是来自于Android的消息处理机制，主线程执行任何代码它都会走到Looper.loop方法当中，而这个函数当中有一个**mLogging**对象，它会在每个message处理前后都会被调用，而主线程发生了卡顿，那就一定会在dispatchMessage方法中执行了耗时的代码，那我们在这个message执行之前呢，我们可以

在子线程当中去postDelayed一个任务，这个Delayed的时间就是我们设定的阈值，如果主线程的messaege在这个阈值之内完成了，那就取消掉这个子线程当中的任务，如果主线程的message在阈值之内没有被完成，那子线程当中的任务就会被执行，它会获取到当前主线程执行的一个堆栈，那我们就可以知道哪里发生了卡顿。

经过实践，我们发现这种方案获取的堆栈信息它不一定是准确的，因为获取到的堆栈信息它很可能是主线程最终执行的一个位置，而真正耗时的地方其实已经执行完成了，于是呢，我们就对这个方案做了一些优化，我们采取了**高频采集**的方案，也就是在一个周期内我们会多次采集主线程的堆栈信息，如果发生了卡顿，那我们就将这些卡顿信息压缩之后上报给APM后台，然后找出重复的堆栈信息，这些重复发生的堆栈大概率就是卡顿发生的一个位置，这样就提高了获取卡顿信息的一个准确性。

3、卡顿的一整套解决方案是怎么做的?

首先，针对卡顿，我们采用了**线上、线下工具相结合**的方式，线下工具我们册中医药尽可能早地去暴露问题，而针对于线上工具呢，我们侧重于监控的全面性、自动化以及异常感知的灵敏度。

同时呢，卡顿问题还有很多的难题。比如说**有的代码呢，它不到你卡顿的一个阈值，但是执行过多，或者它错误地执行了很多次，它也会导致用户感官上的一个卡顿**，所以我们在线下通过AOP的方式对常见的耗时代码进行了Hook，然后对一段时间内获取到的数据进行分析，我们就可以知道这些耗时的代码发生的时机和次数以及耗时情况。然后，看它是不是满足我们的一个预期，不满足预期的话，我们就可以直接到线下进行修改。同时，卡顿监控它还有很多容易被忽略的一个**盲区**，比如说生命周期的一个间隔，那对于这种特定的问题呢，我们就采用了编译时注解的方式修改了项目当中所有Handler的父类，对于其中的两个方法进行了监控，我们就可以知道主线程message的执行时间以及它们的调用堆栈。

对于**线上卡顿**，我们除了计算App的卡顿率、ANR率等常规指标之外呢，我们还计算了页面的秒开率、生命周期的执行时间等等。而且，在卡顿发生的时刻，我们也尽可能多地保存下来了当前的一个场景信息，这为我们之后解决或者复现这个卡顿留下了依据。

八、总结

恭喜你，如果你看到了这里，你会发现要做好应用的卡顿优化的确不是一件简单的时，它需要你有成体系的知识构建基底。最后，我们再来回顾一下面对卡顿优化，我们已经探索的以下九大主题：

- 1、卡顿优化分析方法与工具：背景介绍、卡顿分析方法之使用shell命令分析CPU耗时、卡顿优化工具。
- 2、自动化卡顿检测方案及优化：卡顿检测方案原理、AndroidPerformanceMonitor实战及其优化。
- 3、ANR分析与实战：ANR执行流程、线上ANR监控方式、ANR-WatchDog原理。
- 4、卡顿单点问题检测方案：IPC单点问题检测方案、卡顿问题检测方案。
- 5、如何实现界面秒开？：界面秒开实现、Lancet、界面秒开监控纬度。
- 6、优雅监控耗时盲区：耗时盲区监控难点以及线上与线下的监控方案。
- 7、卡顿优化技巧总结：卡顿优化实践经验、卡顿优化工具建设。
- 8、常见卡顿问题解决方案总结
- 9、卡顿优化的常见问题

相信看到这里，你一定收获满满，但是要记住，方案再好，也只有自己动手去实践，才能真正地掌握它。**只有重视实践，充分运用感性认知潜能，在项目中磨炼自己，才是正确的学习之道。在实践中，在某些关键动作上刻意练习，也会取得事半功倍的效果。**

参考链接：

1、国内Top团队大牛带你玩转Android性能分析与优化 第6章 卡顿优化
2、极客时间之Android开发高手课 卡顿优化
3、《Android移动性能实战》第四章 CPU
4、《Android移动性能实战》第七章 流畅度
5、Android dumpsys cpuinfo 信息解读
6、如何清楚易懂的解释“UV和PV " 的定义?
7、nanoscope-An extremely accurate Android method tracing tool
8、DroidAssist-A lightweight Android Studio gradle plugin based on Javassist for editing bytecode in Android.
9、lancet-A lightweight and fast AOP framework for Android App and SDK developers
10、MethodTraceMan-用于快速找到高耗时方法，定位解决Android App卡顿问题
11、Linux环境下进程的CPU占用率
12、使用 ftrace
13、profilo-A library for performance traces from production
14、ftrace 简介
15、atrace源码
16、AndroidAdvanceWithGeektime / Chapter06
17、AndroidAdvanceWithGeektime / Chapter06-plus