App的好坏，卡顿占一半？Android线上卡顿监控

xfhy 鸿洋 2024-02-02 08:35 发表于北京

本文作者：**xfhy**，原文发布于：**xfhy**。

1 卡顿与ANR的关系

卡顿是UI没有及时的按照用户的预期进行反馈，没有及时地渲染出来，从而看起来不连续、不一致。产生卡顿的原因太多了，很难一一列举，但ANR是Google认为规定的概念，产生ANR的原因最多只有4个。分别是：

- **Service Timeout**:比如前台服务在20s内未执行完成，后台服务Timeout时间是前台服务的10倍，200s；
- **BroadcastQueue Timeout**：比如前台广播在10s内未执行完成，后台60s；
- **ContentProvider Timeout**：内容提供者,在publish过超时10s;
- **InputDispatching Timeout**: 输入事件分发超时5s，包括按键和触摸事件。

假如我在一个button的onClick事件中，有一个耗时操作，这个耗时操作的时间是10秒，但这个耗时操作并不会引发ANR，它只是一次卡顿。

一方面，两者息息相关，长时间的UI卡顿是导致ANR的最常见的原因；但另一方面，从原理上来看，两者既不充分也不必要，是两个纬度的概念。

市面上的一些卡顿监控工具，经常被用来监控ANR（卡顿阈值设置为5秒），这其实很不严谨：首先，5秒只是发生ANR的其中一种原因（Touch事件5秒未被及时消费）的阈值，而其他原因发生ANR的阈值并不是5秒；另外，就算是主线程卡顿了5秒，如果用户没有输入任何的Touch事件，同样不会发生ANR，更何况还有后台ANR等情况。真正意义上的ANR监控方案应该是类似matrix里面那样监控signal信号才算。

2 卡顿原理

主线程从ActivityThread的`main`方法开始，准备好主线程的looper，启动loop循环。在loop循环内，无消息则利用epoll机制阻塞，有消息则处理消息。

因为主线程一直在loop循环中，所以要想在主线程执行什么逻辑，则必须发个消息给主线程的looper然后由这个loop循环触发，由它来分发消息，然后交给msg的target（Handler）处理。举个例子：`ActivityThread.H`。

```
public static void loop() {
    ......
    for (;;) {
      Message msg = queue.next(); // might block
      ......
      msg.target.dispatchMessage(msg);
    }
}
```

loop循环中可能导致卡顿的地方有2个：

1. **queue.next()** ：有消息就返回，无消息则使用epoll机制阻塞(`nativePollOnce`里面)，不会使主线程卡顿。

2. **dispatchMessage耗时太久**：也就是Handler处理消息，app卡顿的话大多数情况下可以认为是这里处理消息太耗时了。

3 卡顿监控

- 方案1：WatchDog，往主线程发消息，然后延迟看该消息是否被处理，从而得出主线程是否卡顿的依据。

- 方案2：利用loop循环时的消息分发前后的日志打印（matrix使用了这个）。

3.1 WatchDog

开启一个子线程，死循环往主线程发消息，发完消息后等待5秒，判断该消息是否被执行，没被执行则主线程发生ANR，此时去获取主线程堆栈。

- 优点：简单，稳定，结果论，可以监控到各种类型的卡顿。

- 缺点：轮询不优雅，不环保，有不确定性，随机漏报。

轮询的时间间隔越小，对性能的负面影响就越大，而时间间隔选择的越大，漏报的可能性也就越大。

- UI线程要不断处理我们发送的Message，必然会影响性能和功耗。

- 随机漏报：`ANRWatchDog`默认的轮询时间间隔为5秒，当主线程卡顿了2秒之后，`ANRWatchDog`的那个子线程才开始往主线程发送消息，并且主线程在3秒之后不卡顿了，此时主线程已经卡顿了5秒了，子线程发送的那个消息也随之得到执行，等子线程睡5秒起床的时候发现消息已经被执行了，它没意识到主线程刚刚发生了卡顿。

改进：

- 监控到发生ANR时，除了获取主线程堆栈，再获取一下CPU、内存占用等信息。

- 还可结合`ProcessLifecycleOwner`，app在前台才开启检测，在后台停止检测。

另外有些方案的思路，如果我们不断缩小轮询的时间间隔，用更短的轮询时间，连续几个周期消息都没被处理才视为一次卡顿。则更容易监控到卡顿，但对性能损耗大一些。即使是缩小轮询时间间隔，也不一定能监控到。假设每2秒轮询一次，如果连续三次没被处理，则认为发生了卡顿。在02秒之间主线程开始发生卡顿，在第2秒时开始往主线程发消息，这样在到达次数，也就是8秒时结束，但主线程的卡顿在68秒之间就刚好结束了，此时子线程在第8秒时醒来发现消息已经被执行了，它没意识到主线程刚刚发生了卡顿。

3.2 Looper Printer

替换主线程Looper的Printer，监控`dispatchMessage`的执行时间（大部分主线程的操作最终都会执行到这个`dispatchMessage`中）。这种方案在微信上有较大规模使用，总体来说性能不是很差，matrix目前的`EvilMethodTracer`和`AnrTracer`就是用这个来实现的。

- 优点：不会随机漏报，无需轮询，一劳永逸。
- 缺点：某些类型的卡顿无法被监控到，但有相应解决方案。

`queue.next()`可能会阻塞，这种情况下监控不到。

```
//Looper.java
for (;;) {
 //这里可能会block，Printer无法监控到next里面发生的卡顿
  Message msg = queue.next(); // might block

  // This must be in a local variable, in case a UI event sets the logger
  final Printer logging = me.mLogging;
  if (logging != null) {
    logging.println(">>>>> Dispatching to " + msg.target + " " +
        msg.callback + ": " + msg.what);
  }

  msg.target.dispatchMessage(msg);

  if (logging != null) {
    logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
  }
}

//MessageQueue.java
for (;;) {
  if (nextPollTimeoutMillis != 0) {
    Binder.flushPendingCommands();
  }

  nativePollOnce(ptr, nextPollTimeoutMillis);

  //......

  // Run the idle handlers.
  // We only ever reach this code block during the first iteration.
  for (int i = 0; i < pendingIdleHandlerCount; i++) {
    final IdleHandler idler = mPendingIdleHandlers[i];
    mPendingIdleHandlers[i] = null; // release the reference to the handler

    boolean keep = false;
    try {
   //IdleHandler的queueIdle，如果Looper是主线程，那么这里明显是在主线程执行的，虽然现在主线程空闲，但也不能
      keep = idler.queueIdle();
    } catch (Throwable t) {
      Log.wtf(TAG, "IdleHandler threw exception", t);
    }

    if (!keep) {
      synchronized (this) {
        mIdleHandlers.remove(idler);
      }
```

```
        }
    }
    //......
}
```

1. 主线程空闲时会阻塞`next()`，具体是阻塞在`nativePollOnce()`，这种情况下无需监控。

2. Touch事件大部分是从`nativePollOnce`直接到了`InputEventReceiver`，然后到`ViewRootImpl`进行分发。

3. `IdleHandler`的`queueIdle()`回调方法也无法监控到。

4. 还有一类相对少见的问题是SyncBarrier（同步屏障）的泄漏同样无法被监控到。

第一种情况我们不用管，接下来看一下后面3种情况下如何监控卡顿。

3.2.1 监控TouchEvent卡顿

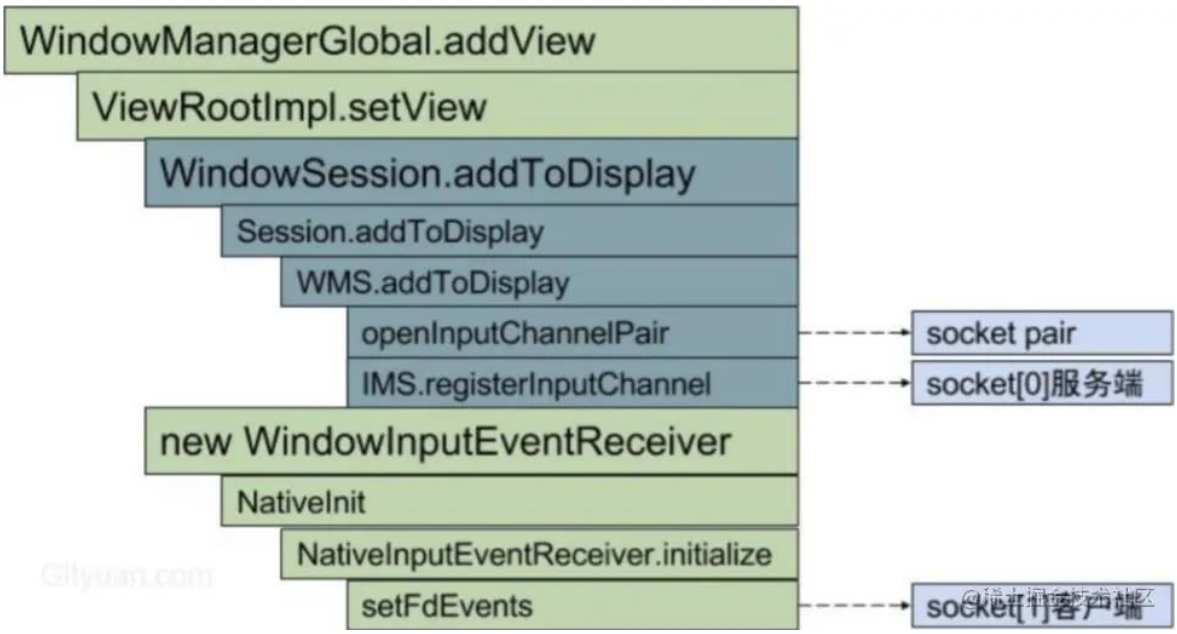
首先，Touch是怎么传递到Activity的？给一个view设置一个`OnTouchListener`，然后看一些Touch的调用栈。

```
com.xfhy.watchsignaldemo.MainActivity.onCreate$lambda-0(MainActivity.kt:31)
com.xfhy.watchsignaldemo.MainActivity.$r8$lambda$f2Bz7skgRCh8TKh1SZX03s91UhA(Unknown Source:0)
com.xfhy.watchsignaldemo.MainActivity$$ExternalSyntheticLambda0.onTouch(Unknown Source:0)
android.view.View.dispatchTouchEvent(View.java:13695)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
android.view.ViewGroup.dispatchTransformedTouchEvent(ViewGroup.java:3249)
android.view.ViewGroup.dispatchTouchEvent(ViewGroup.java:2881)
com.android.internal.policy.DecorView.superDispatchTouchEvent(DecorView.java:741)
com.android.internal.policy.PhoneWindow.superDispatchTouchEvent(PhoneWindow.java:2013)
android.app.Activity.dispatchTouchEvent(Activity.java:4180)
androidx.appcompat.view.WindowCallbackWrapper.dispatchTouchEvent(WindowCallbackWrapper.java:70)
com.android.internal.policy.DecorView.dispatchTouchEvent(DecorView.java:687)
android.view.View.dispatchPointerEvent(View.java:13962)
android.view.ViewRootImpl$ViewPostImeInputStage.processPointerEvent(ViewRootImpl.java:6420)
android.view.ViewRootImpl$ViewPostImeInputStage.onProcess(ViewRootImpl.java:6215)
android.view.ViewRootImpl$InputStage.deliver(ViewRootImpl.java:5604)
android.view.ViewRootImpl$InputStage.onDeliverToNext(ViewRootImpl.java:5657)
android.view.ViewRootImpl$InputStage.forward(ViewRootImpl.java:5623)
android.view.ViewRootImpl$AsyncInputStage.forward(ViewRootImpl.java:5781)
android.view.ViewRootImpl$InputStage.apply(ViewRootImpl.java:5631)
android.view.ViewRootImpl$AsyncInputStage.apply(ViewRootImpl.java:5838)
android.view.ViewRootImpl$InputStage.deliver(ViewRootImpl.java:5604)
android.view.ViewRootImpl$InputStage.onDeliverToNext(ViewRootImpl.java:5657)
android.view.ViewRootImpl$InputStage.forward(ViewRootImpl.java:5623)
android.view.ViewRootImpl$InputStage.apply(ViewRootImpl.java:5631)
android.view.ViewRootImpl$InputStage.deliver(ViewRootImpl.java:5604)
android.view.ViewRootImpl.deliverInputEvent(ViewRootImpl.java:8701)
android.view.ViewRootImpl.doProcessInputEvents(ViewRootImpl.java:8621)
android.view.ViewRootImpl.enqueueInputEvent(ViewRootImpl.java:8574)
android.view.ViewRootImpl$WindowInputEventReceiver.onInputEvent(ViewRootImpl.java:8959)
android.view.InputEventReceiver.dispatchInputEvent(InputEventReceiver.java:239)
android.os.MessageQueue.nativePollOnce(Native Method)
android.os.MessageQueue.next(MessageQueue.java:363)
android.os.Looper.loop(Looper.java:176)
android.app.ActivityThread.main(ActivityThread.java:8668)
java.lang.reflect.Method.invoke(Native Method)
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:513)
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1109)
```

当有触摸事件时，`nativePollOnce()`会收到消息，然后会从native层直接调用`InputEventReceiver.dispatchInputEvent()`。

```
public abstract class InputEventReceiver {
  public InputEventReceiver(InputChannel inputChannel, Looper looper) {
    if (inputChannel == null) {
      throw new IllegalArgumentException("inputChannel must not be null");
    }
    if (looper == null) {
      throw new IllegalArgumentException("looper must not be null");
    }

    mInputChannel = inputChannel;
    mMessageQueue = looper.getQueue();
    //在这里进行的注册，native层会将该实例记录下来，每当有事件到达时就会派发到这个实例上来
    mReceiverPtr = nativeInit(new WeakReference<InputEventReceiver>(this),
        inputChannel, mMessageQueue);

    mCloseGuard.open("dispose");
  }

  // Called from native code.
  @SuppressWarnings("unused")
  @UnsupportedAppUsage
  private void dispatchInputEvent(int seq, InputEvent event) {
    mSeqMap.put(event.getSequenceNumber(), seq);
    onInputEvent(event);
  }
}
```

`InputReader`（读取、拦截、转换输入事件）和`InputDispatcher`（分发事件）都是运行在`system_server`系统进程中，而我们的应用程序运行在自己的应用进程中，这里涉及到跨进程通信，这里的跨进程通信用的非binder方式，而是用的socket。



`InputDispatcher`会与我们的应用进程建立连接，它是socket的服务端；我们应用进程的native层会有一个socket的客户端，客户端收到消息后，会通知我们应用进程里`ViewRootImpl`创建的`WindowInputEventReceiver`（继承自`InputEventReceiver`）来接收这个输入事件。事件传递也就走通了，后面就是上层的View树事件分发了。

这里为啥用socket而不用binder？Socket可以实现异步的通知，且只需要两个线程参与（Pipe两端各一个），假设系统有N个应用程序，跟输入处理相关的线程数目是N+1（1是Input

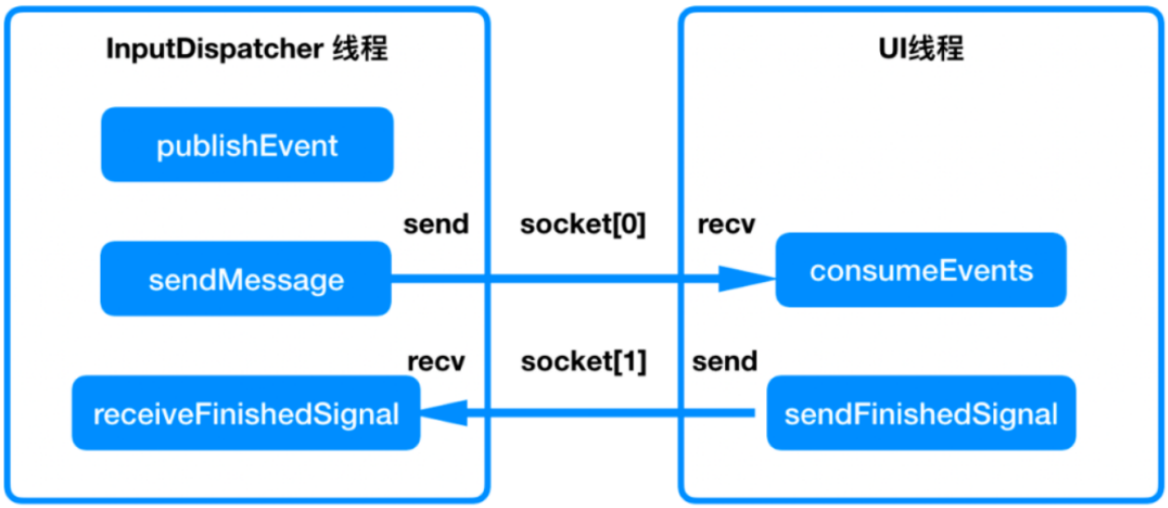
Dispatcher线程）。

然而，如果用Binder实现的话，为了实现异步接收，每个应用程序需要两个线程，一个Binder线程，一个后台处理线程（不能在Binder线程里处理输入，因为这样太耗时，将会阻塞住发射端的调用线程）。在发射端，同样需要两个线程，一个发送线程，一个接收线程来接收应用的完成通知，所以，N个应用程序需要2（N+1）个线程。相比之下，Socket还是高效多了。

```
//frameworks/native/services/inputflinger/InputDispatcher.cpp
void InputDispatcher::startDispatchCycleLocked(nsecs_t currentTime,
    const sp<Connection>& connection) {
  ......
  status = connection->inputPublisher.publishKeyEvent(dispatchEntry->seq,
          keyEntry->deviceId, keyEntry->source,
          dispatchEntry->resolvedAction, dispatchEntry->resolvedFlags,
          keyEntry->keyCode, keyEntry->scanCode,
          keyEntry->metaState, keyEntry->repeatCount, keyEntry->downTime,
          keyEntry->eventTime);
  ......
}

//frameworks/native/libs/input/InputTransport.cpp
status_t InputPublisher::publishKeyEvent(
    uint32_t seq,
    int32_t deviceId,
    int32_t source,
    int32_t action,
    int32_t flags,
    int32_t keyCode,
    int32_t scanCode,
    int32_t metaState,
    int32_t repeatCount,
    nsecs_t downTime,
    nsecs_t eventTime) {
  ......

  InputMessage msg;
  ......
  msg.body.key.keyCode = keyCode;
  ......
  return mChannel->sendMessage(&msg);
}

//frameworks/native/libs/input/InputTransport.cpp
//调用 socket 的 send 接口来发送消息
status_t InputChannel::sendMessage(const InputMessage* msg) {
  size_t msgLength = msg->size();
  ssize_t nWrite;
  do {
    nWrite = ::send(mFd, msg, msgLength, MSG_DONTWAIT | MSG_NOSIGNAL);
  } while (nWrite == -1 && errno == EINTR);
  ......
}
```

有了上面的知识铺垫，现在回到我们的主问题上来，如何监控`TouchEvent`卡顿。既然它们是用socket来进行通信的，那么我们可以通过PLT Hook，去Hook这对socket的发送（send）和接收(recv)方法，从而监控Touch事件。

当调用到了recvfrom时（send和recv最终会调用sendto和recvfrom，这2个函数的具体定义在socket.h源码，说明我们的应用接收到了Touch事件，当调用到了sendto时，说明这个Touch事件已经被成功消费掉了，当两者的时间相差过大时即说明产生了一次Touch事件的卡顿。
https://cs.android.com/android/platform/superproject/+/master:bionic/libc/include/bits/fortify/socket.h;drc=b0193ccac5b8399f9b5ef270d102b5a50f9446ab;l=79



PLT Hook是什么，它是一种native hook，另外还有一种native hook方式是inline hook。PLT hook的优点是稳定性可控，可线上使用，但它只能hook通过PLT表跳转的函数调用，这在一定程度上限制了它的使用场景。

对PLT Hook的具体原理感兴趣的同学可以看一下下面2篇文章：

• Android PLT hook 概述
https://github.com/iqiyi/xHook/blob/master/docs/overview/android_plt_hook_overview.zh-CN.md

• 字节跳动开源 Android PLT hook 方案 bhook
https://zhuanlan.zhihu.com/p/401547387

目前市面上比较流行的PLT Hook开源库主要有2个，一个是爱奇艺开源的xhook（*https://github.com/iqiyi/xHook*），一个是字节跳动开源的bhook（*https://github.com/bytedance/bhook*）。我这里使用xhook来举例，InputDispatcher.cpp最终会被编译成libinput.so（具体Android.mk信息看这里（*https://android.googlesource.com/platform/frameworks/base/+/android-4.2.1_r1/services/input/Android.mk*））。那我们就直接hook这个libinput.so的sendto和recvfrom函数。
理论知识有了，直接开干：

```
ssize_t (*original_sendto)(int sockfd, const void *buf, size_t len, int flags,
                const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t my_sendto(int sockfd, const void *buf, size_t len, int flags,
          const struct sockaddr *dest_addr, socklen_t addrlen) {
  //应用端已消费touch事件
  if (getCurrentTime() - lastTime > 5000) {
    __android_log_print(ANDROID_LOG_DEBUG, "xfhy_touch", "Touch有点卡顿");
    //todo xfhy 在这里调用java去dump主线程堆栈
  }
  long ret = original_sendto(sockfd, buf, len, flags, dest_addr, addrlen);
  return ret;
}

ssize_t (*original_recvfrom)(int sockfd, void *buf, size_t len, int flags,
                struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t my_recvfrom(int sockfd, void *buf, size_t len, int flags,
          struct sockaddr *src_addr, socklen_t *addrlen) {
  //收到touch事件
  lastTime = getCurrentTime();
```

```
    long ret = original_recvfrom(sockfd, buf, len, flags, src_addr, addrlen);
    return ret;
}

void Java_com_xfhy_touch_TouchTest_start(JNIEnv *env, jclass clazz) {
    xhook_register(".*libinput\\.so$", "__sendto_chk",(void *) my_sendto, (void **) (&original_sendto));
    xhook_register(".*libinput\\.so$", "sendto",(void *) my_sendto, (void **) (&original_sendto));
    xhook_register(".*libinput\\.so$", "recvfrom",(void *) my_recvfrom, (void **) (&original_recvfrom));
}
```

上面这个是我写的demo，完整代码看这里，这个demo肯定是不够完善的。但方案是可行的。完善的方案请看matrix的Touch相关源码。

https://github.com/xfhy/xHook/blob/master/libtouch/jni/touch.c

https://github.com/Tencent/matrix/blob/master/matrix/matrix-android/matrix-trace-canary/src/main/cpp/TouchEventTracer.cc

3.2.2 监控IdleHandler卡顿

`IdleHandler`任务最终会被存储到`MessageQueue`的`mIdleHandlers`（一个ArrayList）中，在主线程空闲时，也就是`MessageQueue`的`next`方法暂时没有message可以取出来用时，会从`mIdleHandlers`中取出`IdleHandler`任务进行执行。那我们可以把这个`mIdleHandlers`替换成自己的，重写`add`方法，添加进来的`IdleHandler`给它包装一下，包装的那个类在执行`queueIdle`时进行计时，这样添加进来的每个`IdleHandler`在执行的时候我们都能拿到其`queueIdle`的执行时间 。如果超时我们就进行记录或者上报。

```
fun startDetection() {
    val messageQueue = mHandler.looper.queue
    val messageQueueJavaClass = messageQueue.javaClass
    val mIdleHandlersField = messageQueueJavaClass.getDeclaredField("mIdleHandlers")
    mIdleHandlersField.isAccessible = true

    //虽然mIdleHandlers在Android Q以上被标记为UnsupportedAppUsage,但居然可以成功设置. 只有在反射访问mIdleHandlers时,
    mIdleHandlersField.set(messageQueue, MyArrayList())
}
class MyArrayList : ArrayList<IdleHandler>() {

  private val handlerThread by lazy {
    HandlerThread("").apply {
      start()
    }
  }
  private val threadHandler by lazy {
    Handler(handlerThread.looper)
  }

  override fun add(element: IdleHandler): Boolean {
    return super.add(MyIdleHandler(element, threadHandler))
  }

}
class MyIdleHandler(private val originIdleHandler: IdleHandler, private val threadHandler: Handler) : IdleHandler {

  override fun queueIdle(): Boolean {
    log("开始执行idleHandler")

    //1. 延迟发送Runnable，Runnable收集主线程堆栈信息
    val runnable = {
      log("idleHandler卡顿 \n ${getMainThreadStackTrace()}")
    }
    threadHandler.postDelayed(runnable, 2000)
    val result = originIdleHandler.queueIdle()
    //2. idleHandler如果及时完成，那么就移除Runnable。如果上面的Runnable得到执行，说明主线程的idleHandler已经执行了2秒
    threadHandler.removeCallbacks(runnable)
    return result
  }
}
```

反射完成之后，我们简单添加一个IdleHandler，然后在里面sleep(10000)测试一下，得到结果如下：

```
2022-10-17 07:33:50.282 28825-28825/com.xfhy.allinone D/xfhy_tag: 开始执行idleHandler
2022-10-17 07:33:52.286 28825-29203/com.xfhy.allinone D/xfhy_tag: idleHandler卡顿
    java.lang.Thread.sleep(Native Method)
    java.lang.Thread.sleep(Thread.java:443)
    java.lang.Thread.sleep(Thread.java:359)
    com.xfhy.allinone.actual.idlehandler.WatchIdleHandlerActivity$startTimeConsuming$1.queueIdle(WatchIdleHandler
    com.xfhy.allinone.actual.idlehandler.MyIdleHandler.queueIdle(WatchIdleHandlerActivity.kt:62)
    android.os.MessageQueue.next(MessageQueue.java:465)
    android.os.Looper.loop(Looper.java:176)
    android.app.ActivityThread.main(ActivityThread.java:8668)
    java.lang.reflect.Method.invoke(Native Method)
    com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:513)
    com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1109)
```

从日志堆栈里面很清晰地看到具体是哪里发生了卡顿。

3.2.3 监控SyncBarrier泄漏

什么是SyncBarrier泄漏？在说这个之前，我们得知道什么是SyncBarrier，它翻译过来叫同步屏障，听起来很牛逼，但实际上就是一个Message，只不过这个Message没有target。没有target，那这个Message拿来有什么用？当MessageQueue中存在SyncBarrier的时候，同步消息就得不到执行，而只会去执行异步消息。我们平时用的Message一般是同步的，异步的Message主要是配合SyncBarrier使用。当需要执行一些高优先级的事情的时候，比如View绘制啥的，就需要往主线程MessageQueue插个SyncBarrier，然后ViewRootImpl 将 mTraversalRunnable 交给 Choreographer ，Choreographer 等到下一个VSYNC信号到来时，及时地去执行mTraversalRunnable ，交给Choreographer 之后的部分逻辑优先级是很高的，比如执行mTraversalRunnable 的时候，这种逻辑是放到异步消息里面的。回到ViewRootImpl之后将SyncBarrier移除。

关于同步屏障和Choreographer 的详细逻辑可以看我之前的文章Handler同步屏障 、Choreographer原理及应用 。
https://github.com/xfhy/Android-Notes/blob/master/Blogs/Android/系统源码解析/Handler同步屏障.md
https://github.com/xfhy/Android-Notes/blob/master/Blogs/Android/系统源码解析/Choreographer原理及应用.md

```
@UnsupportedAppUsage
void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        //插入同步屏障，mTraversalRunnable的优先级很高，我需要及时地去执行它
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        //mTraversalRunnable里面会执行doTraversal
        mChoreographer.postCallback(Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}

void unscheduleTraversals() {
    if (mTraversalScheduled) {
        mTraversalScheduled = false;
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
```

```
    mChoreographer.removeCallbacks(Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
  }
}

void doTraversal() {
  if (mTraversalScheduled) {
    mTraversalScheduled = false;
    //移除同步屏障
    mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
    performTraversals();
  }
}
```

再来说说什么是同步屏障泄露：我们看到在一开始的时候`scheduleTraversals`里面插入了一个同步屏障，这时只能执行异步消息了，不能执行同步消息。假设出现了某种状况，让这个同步屏障无法被移除，那么消息队列中就一直执行不到同步消息，可能导致主线程假死，你想想，主线程里面同步消息都执行不了了，那岂不是要完蛋。那什么情况下会导致出现上面的异常情况?

1. `scheduleTraversals`线程不安全，万一不小心post了多个同步屏障，但只移除了最后一个，那有的同步屏障没被移除的话，同步消息无法执行。

2. `scheduleTraversals`中post了同步屏障之后，假设某些操作不小心把异步消息给移除了，导致没有移除该同步屏障，也会造成同样的悲剧。

问题找到了，怎么解决？有什么好办法能监控到这种情况吗（虽然这种情况比较少见）？微信的同学给出了一种方案，我简单描述下：

1. 开个子线程，轮询检查主线程的`MessageQueue`里面的message，检查是否有同步屏障消息的when已经过去了很久了，但还没得到执行。

2. 此时可以合理怀疑该同步屏障消息可能已泄露，但还不能确定。

3. 这个时候，往主线程发一个同步消息和一个异步消息（可以间隔地多发几次，增加可信度），如果同步消息没有得到执行，但异步消息得到执行了，这说明什么？说明主线程的`MessageQueue`中有一个同步屏障一直没得到移除，所以同步消息才没得到执行，而异步消息得到执行了。

4. 此时，可以激进一点，把这个泄露的同步泄露消息给移除掉。

下面是此方案的核心代码，完整源码在这里。
https://github.com/xfhy/AllInOne/blob/331f96b75febc1567c8a98ffc3cdb4df1f26d618/app/src/main/java/com/xfhy/allinone/actual/syncbarrier/WatchSyncBarrierThread.kt

```
override fun run() {
  while (!isInterrupted) {
    val messageHead = mMessagesField.get(mainThreadMessageQueue) as? Message
    messageHead?.let { message ->
      //该消息为同步屏障 && 该消息3秒没得到执行，先怀疑该同步屏障发生了泄露
      if (message.target == null && message.`when` - SystemClock.uptimeMillis() < -3000) {
        //查看MessageQueue#postSyncBarrier(long when)源码得知，同步屏障message的arg1会携带token，
        // 该token类似于同步屏障的序号，每个同步屏障的token是不同的，可以根据该token唯一标识一个同步屏障
        val token = message.arg1
        startCheckLeaking(token)
      }
    }
    sleep(2000)
  }
}

private fun startCheckLeaking(token: Int) {
  var checkCount = 0
  barrierCount = 0
  while (checkCount < 5) {
    checkCount++
    //1. 判断该token对应的同步屏障是否还存在,不存在就退出循环
    if (isSyncBarrierNotExist(token)) {
      break
    }
    //2. 存在的话，发1条异步消息给主线程Handler,再发1条同步消息给主线程Handler，
    // 看一下同步消息是否得到了处理，如果同步消息发了几次都没处理，而异步消息则发了几次都被处理了，说明SyncBarrier泄露了
    if (detectSyncBarrierOnce()) {
      //发生了SyncBarrier泄露
      //3. 如果有泄露，那么就移除该泄露了的同步屏障(反射调用MessageQueue的removeSyncBarrier(int token))
      removeSyncBarrier(token)
      break
    }
    SystemClock.sleep(1000)
  }
}

private fun detectSyncBarrierOnce(): Boolean {
  val handler = object : Handler(Looper.getMainLooper()) {
    override fun handleMessage(msg: Message) {
      super.handleMessage(msg)
      when (msg.arg1) {
        -1 -> {
          //异步消息
          barrierCount++
        }
        0 -> {
          //同步消息 说明主线程的同步消息是能做事的啊，就没有SyncBarrier一说了
          barrierCount = 0
        }
        else -> {}
      }
    }
  }

  val asyncMessage = Message.obtain()
  asyncMessage.isAsynchronous = true
  asyncMessage.arg1 = -1

  val syncMessage = Message.obtain()
  syncMessage.arg1 = 0

  handler.sendMessage(asyncMessage)
  handler.sendMessage(syncMessage)

  //超过3次，主线程的同步消息还没被处理，而异步消息缺得到了处理，说明确实是发生了SyncBarrier泄露
  return barrierCount > 3
}
```

4 小结

文中详细介绍了卡顿与ANR的关系，以及卡顿原理和卡顿监控，详细捋下来可对卡顿有更深的理解。对于Looper Printer方案来说，是比较完善的，而且微信也在使用此方案，该踩的坑也踩完了。

最后推荐一下我做的网站，玩Android: *wanandroid.com* ，包含详尽的知识体系、好用的工具，还有本公众号文章合集，欢迎体验和收藏！

推荐阅读：

你真的了解ViewModel的设计思想吗?

鸿蒙：5 分钟秒懂 ArkTs，不能错过的知识点解析

大公司如何做 APP：背后的开发流程和技术

扫一扫 关注我的公众号

如果你想要跟大家分享你的文章，欢迎投稿~

┏(^ 0 ^)┛明天见！
