# TODO

HW Watchdog

http://gityuan.com/2016/06/21/watchdog/

# 概述

## 为什么需要看门狗?

　　Watchdog,初次见到这个词语是在大学的单片机书上, 谈到了看门狗定时器. 在很早以前那个单片机刚发展的时候, 单片机容易受到外界工作影响, 导致自己的程序跑飞, 因此有了看门狗的保护机制, 即:需要每多少时间内都去喂狗, 如果不喂狗, 看门狗将触发重启. 大体原理是, 在系统运行以后启动了看门狗的计数器，看门狗就开始自动计数，如果到了一定的时间还不去清看门狗，那么看门狗计数器就会溢出从而引起看门狗中断，造成系统复位。

而手机, 其实是一个超强超强的单片机, 其运行速度比单片机快N倍, 存储空间比单片机大N倍, 里面运行了若干个线程, 各种软硬件协同工作, 不怕一万,就怕万一, 万一我们的系统死锁了, 万一我们的手机也受到很大的干扰程序跑飞了. 都可能发生jj思密达的事情, 因此, 我们也需要看门狗机制.

门狗有硬件看门狗和软件看门狗之分, 硬件就是单片机那种的定时器电路, 软件, 则是我们自己实现一个类似机制的看门狗

## HW看门口

现在的CPU基本上都带有WatchDog功能，这种硬件的WatchDog可以在系统死掉(死锁或者程序跑飞)后重启系统，让系统回到可以工作的状态。WatchDog不能防止系统死掉，但是它能够起死回生，从而提高系统的可用性。

硬件级的WatchDog也有它的局限性，它只能在系统范围内生效，不能针对单个进程，某个进程死掉了，WatchDog未必知道。对于像Linux这类久经考验的操作系统来说，整个系统死掉概率非常低，所以此时硬件级的WatchDog意义反而不大。

## Android系统层看门狗

　　对手机系统而言，因为肩负着接听电话和接收短信的“重任”，所以被寄予7x24小 时正常工作的希望。但是作为一个在嵌入式设备上运行的操作系统，Android运行中必须面对各种软硬件干扰，从最简单的代码出现死锁或者被阻塞，到内存越界导致的内存破坏，或者由于硬件问题导致的内存反转，甚至是极端工作环境下出现的CPU电子迁移和存储器消磁。这一切问题都可能导致系统服务发生难以预料的崩溃和死机。

    想解决这一问题，可以从正反两个方向出发，其一是提高软硬件在极端状态下的可靠性，如进行程序终止性验证，或选用抗辐射加固器件。但是基于成本考虑，普通的手机系统很难做到完全不出故障；另一个方法是及时发现系统崩溃并重启系统。手机系统的大部分的故障都会在重启后消失，不会影响继续使用。所以简单的办法是，如果检测到系统不正常了，将设备重新启动，这样用户就能继续使用了。那么如何才能判断系统是否正常呢。在早期的手机平台上通常的做法是在设备中增加一个硬件看门狗，软件系统必须定 时的向看门狗硬件中写值来表示自己没出故障（俗称“喂狗”），否则超过了规定的时间看门狗就会重新启动设备。

    硬件看门狗的问题是它的功能比较单一，只能监控整个系统。早期的手机操作系统大多是单任务的，硬件看门狗勉强能胜任。Android的SystemServer是一个非常复杂的进程，里面运行的服务超过五十种，是最可能出问题的进程，因此有必要对SystemServer中运行的各种线程实施监控。但是如果使用硬件看门狗的工作方式，每个线程隔一段时间去喂狗，不但非常浪费CPU，而且会导致程序设计更加复杂。因此Android开发了WatchDog类作为软件看门狗来监控SystemServer中的线程。一旦发现问题，WatchDog会杀死SystemServer进程。

SystemServer的父进程Zygote接收到SystemServer的死亡信号后，会杀死自己。Zygote进程死亡的信号传递到Init进程后，Init进程会杀死Zygote进程所有的子进程并重启Zygote。这样整个手机相当于重启一遍。通常SystemServer出现问题和kernel并没有关系，所以这种“软重启”大部分时候都能够解决问题。而且这种“软重启”的速度更快，对用户的影响也更小。

---------------------

作者：宇落无痕

来源：CSDN

原文：https://blog.csdn.net/fu\_kevin0606/article/details/64479489

版权声明：本文为博主原创文章，转载请附上博文链接！

在android 系统中，看门狗可以分成以下两种类型。

### HW Watchdog

用于监测CPU 执行是否异常, 启用Kernel RT

thread tick HW watchdog 来达成, 如果异常, 则重启整个系统。

对单片机的运行状态进行实时监测，针对运行故障做一些保护处理，譬如让系统重启。这种Watchdog属于硬件层面，必须有硬件电路的支持。

Linux也引入了Watchdog，在Linux内核下，当Watchdog启动后，便设定了一个定时器，如果在超时时间内没有对/dev/Watchdog进行写操作，则会导致系统重启。通过定时器实现的Watchdog属于软件层面。

### 2）System Server Watchdog

Android系统为了保证系统的稳定性，也设计了这么一个看门狗，其为了保证各种系统服务能够正常工作，要监控很多的服务，并且在核心服务异常时要进行重启，还要保存现场。

用于监测Android System Server 关键线程和资源使用是否正常, 如果异常则重启android 上层。

Android 平台实现了一个软件的WatchDog来监护SystemServer。SystemServer无疑是Android平台中最重要的进程了，里面运行了整个平台中绝大多数的服务。在这个进程中运行着近50个线程，任何一个线程死掉都可能导致整个系统死掉。SystemServer退出反而问题不大，因为 init进程会重新启动它，但是它死锁就麻烦了，因为整个系统就没法动了。

主要作用：

1)接收系统内部reboot请求,重启系统。

2)监护SystemServer进程,防止系统死锁，转载请附上博文链接！

# 启动

Watchdog的初始化位于SystemServer.这是一个软件看门狗的实现，其主要目的为检测系统锁的持有情况，若发生超时持有锁情况，则视为系统锁死，将作出终止或者继续等待等处理。

    而最近我还是遇到系统在开机启动时一直停留在开机动画界面，从trace上看，系统不幸发生死锁了。为何系统有看门狗，狗狗却没有发现死锁呢？带着问题，来watchdog（WTD）走一遭吧。

从systemserver.java文件上可以看到WTD线程的启动是在很多service注册之后才启动的，那么如果service注册过程死锁，WTD就没法启动检测了。所以上面trace死锁问题的原因就找到了，接下来想办法如何解决这个问题。我大致觉得办法有三，如下：

## SystemServer. Otherservice()

traceBeginAndSlog(**"InitWatchdog"**);  
**final** Watchdog watchdog = Watchdog.getInstance();  
watchdog.init(context, mActivityManagerService);  
traceEnd()

## Watchdog

Gfh

// 构造函数中，默认添加了四个主线程到WTD检测服务中

// 但原生的Android并没有在每个主线程上都添加了监听器

// 只有mMonitorChecker上添加了监听器Monitor，监听器里实现了扫描检测锁情况的

在Watchdog构造函数中将main thread，UIthread，Iothread，DisplayThread加入mHandlerCheckers列表中。最后初始化monitor放入mMonitorCheckers列表中。

Android的Watchdog是一个单例线程，在System Server时就会初始化Watchdog。Watchdog在初始化时，会构建很多**HandlerChecker**，大致可以分为两类：

* **Monitor Checker**，用于检查是Monitor对象可能发生的死锁, AMS, PKMS, WMS等核心的系统服务都是Monitor对象。
* **Looper Checker**，用于检查线程的消息队列是否长时间处于工作状态。Watchdog自身的消息队列，Ui, Io, Display这些全局的消息队列都是被检查的对象。此外，一些重要的线程的消息队列，也会加入到**Looper Checker**中，譬如AMS, PKMS，这些是在对应的对象初始化时加入的。

两类**HandlerChecker**的侧重点不同，**Monitor Checker**预警我们不能长时间持有核心系统服务的对象锁，否则会阻塞很多函数的运行; **Looper Checker**预警我们不能长时间的霸占消息队列，否则其他消息将得不到处理。这两类都会导致系统卡住(System Not Responding)。

**private** Watchdog() {  
 **super**(**"watchdog"**);  
 *// Initialize handler checkers for each common thread we want to check. Note  
 // that we are not currently checking the background thread, since it can  
 // potentially hold longer running operations with no guarantees about the timeliness  
 // of operations there.  
  
 // The shared foreground thread is the main checker. It is where we  
 // will also dispatch monitor checks and do other work.* mMonitorChecker = **new** HandlerChecker(FgThread.getHandler(),  
 **"foreground thread"**, DEFAULT\_TIMEOUT);  
 mHandlerCheckers.add(mMonitorChecker);  
 *// Add checker for main thread. We only do a quick check since there  
 // can be UI running on the thread.* mHandlerCheckers.add(**new** HandlerChecker(**new** Handler(Looper.getMainLooper()),  
 **"main thread"**, DEFAULT\_TIMEOUT));  
 *// Add checker for shared UI thread.* mHandlerCheckers.add(**new** HandlerChecker(UiThread.getHandler(),  
 **"ui thread"**, DEFAULT\_TIMEOUT));  
 *// And also check IO thread.* mHandlerCheckers.add(**new** HandlerChecker(IoThread.getHandler(),  
 **"i/o thread"**, DEFAULT\_TIMEOUT));  
 *// And the display thread.* mHandlerCheckers.add(**new** HandlerChecker(DisplayThread.getHandler(),  
 **"display thread"**, DEFAULT\_TIMEOUT));  
  
 *// Initialize monitor for Binder threads.* addMonitor(**new** BinderThreadMonitor());  
}

## start

Watchdog这个类继承于Thread，所以还会需要一个启动的地方，就是下面这行代码，这是在ActivityManagerService的SystemReady接口中干的。

mActivityManagerService.systemReady(() -> {

traceBeginAndSlog(**"StartWatchdog"**);  
Watchdog.getInstance().start();  
traceEnd();

}

注意，systemReady()方法主要是告知各个服务系统已经准备就绪，可以开始实现各自的职

### Watchdog.run

是一个while死循环，保证持续监测状态。如示意图所示，WTD run函数实体中主要有以下三个函数实现，简要描述各自的作用

scheduleCheckLocked：从对应线程上启动监视器Monitor, 调用子线程HandlerChecker 检查

evaluateCheckerCompletionLocked：计算监视器完成状态，如果有监视器存在锁等待状态，将发生超时结果

getBlockedCheckersLocked：获取超时状态监视器，用于之后的打印堆栈信息，方便分析处理

！

各个Service 服务是否正常工作，此时的看门狗会不断的去检查并且等待它反馈

回来的结果，如果出现死锁的情况，立马杀掉SystemServer 进程。而且这其中如

果超时，会利用debuggerd 打印backtrace 到/proc/sysrq-trigger 当中。

检测是否发生死锁，这是在Watchdog线程中运行的。如果发生死锁而且没有被调试，则退出SystemServer，init进程就会重启SystemServer进程。

主要的运行逻辑如下：

1. Watchdog运行后，便开始无限循环，依次调用每一个HandlerChecker的scheduleCheckLocked()方法
2. 调度完HandlerChecker之后，便开始定期检查是否超时，每一次检查的间隔时间由**CHECK\_INTERVAL**常量设定，为30秒
3. 每一次检查都会调用evaluateCheckerCompletionLocked()方法来评估一下HandlerChecker的完成状态：
   * COMPLETED表示已经完成
   * WAITING和WAITED\_HALF表示还在等待，但未超时
   * OVERDUE表示已经超时。默认情况下，timeout是1分钟，但监测对象可以通过传参自行设定，譬如PKMS的**Handler Checker**的超时是10分钟
4. 如果超时时间到了，还有HandlerChecker处于未完成的状态(OVERDUE)，则通过getBlockedCheckersLocked()方法，获取阻塞的HandlerChecker，生成一些描述信息
5. 保存日志，包括一些运行时的堆栈信息，这些日志是我们解决Watchdog问题的重要依据。如果判断需要杀掉system\_server进程，则给当前进程(system\_server)发送signal 9

只要Watchdog没有发现超时的任务，HandlerChecker就会被不停的调度

---------------------

链接！

@Override  
**public void** run() {  
 **boolean** waitedHalf = **false**;  
 **while** (**true**) {  
 **final** ArrayList<HandlerChecker> blockedCheckers;  
 **final** String subject;  
 **final boolean** allowRestart;  
 **int** debuggerWasConnected = 0;  
 **synchronized** (**this**) {  
 **long** timeout = CHECK\_INTERVAL;  
 *// Make sure we (re)spin the checkers that have become idle within  
 // this wait-and-check interval* **for** (**int** i=0; i<mHandlerCheckers.size(); i++) { //调用HandlerChecker 线程  
 HandlerChecker hc = mHandlerCheckers.get(i);  
 hc.scheduleCheckLocked();  
 }  
  
 **if** (debuggerWasConnected > 0) {  
 debuggerWasConnected--;  
 }  
  
 *// NOTE: We use uptimeMillis() here because we do not want to increment the time we  
 // wait while asleep. If the device is asleep then the thing that we are waiting  
 // to timeout on is asleep as well and won't have a chance to run, causing a false  
 // positive on when to kill things.* **long** start = SystemClock.uptimeMillis();  
 **while** (timeout > 0) {、、等待超时时间，判断是否退出循环  
 **if** (Debug.isDebuggerConnected()) {//出现死锁，杀死SystemServer 进程  
 debuggerWasConnected = 2;  
 }  
 **try** {  
 wait(timeout);  
 } **catch** (InterruptedException e) {  
 Log.wtf(TAG, e);  
 }  
 **if** (Debug.isDebuggerConnected()) {  
 debuggerWasConnected = 2;  
 }  
 timeout = CHECK\_INTERVAL - (SystemClock.uptimeMillis() - start);  
 }  
  
 **final int** waitState = evaluateCheckerCompletionLocked();  
 **if** (waitState == COMPLETED) {  
 *// The monitors have returned; reset* waitedHalf = **false**;  
 **continue**;  
 } **else if** (waitState == WAITING) {  
 *// still waiting but within their configured intervals; back off and recheck* **continue**;  
 } **else if** (waitState == WAITED\_HALF) {  
 **if** (!waitedHalf) {  
 *// We've waited half the deadlock-detection interval. Pull a stack  
 // trace and wait another half.* ArrayList<Integer> pids = **new** ArrayList<Integer>();  
 pids.add(Process.myPid());  
 ActivityManagerService.dumpStackTraces(**true**, pids, **null**, **null**,  
 getInterestingNativePids());  
 waitedHalf = **true**;  
 }  
 **continue**;  
 }  
  
 *// something is overdue!* blockedCheckers = getBlockedCheckersLocked();  
 subject = describeCheckersLocked(blockedCheckers);  
 allowRestart = mAllowRestart;  
 }  
  
 *// If we got here, that means that the system is most likely hung.  
 // First collect stack traces from all threads of the system process.  
 // Then kill this process so that the system will restart.* EventLog.writeEvent(EventLogTags.WATCHDOG, subject);  
  
 ArrayList<Integer> pids = **new** ArrayList<>();  
 pids.add(Process.myPid());  
 **if** (mPhonePid > 0) pids.add(mPhonePid);  
 *// Pass !waitedHalf so that just in case we somehow wind up here without having  
 // dumped the halfway stacks, we properly re-initialize the trace file.* **final** File stack = ActivityManagerService.dumpStackTraces(  
 !waitedHalf, pids, **null**, **null**, getInterestingNativePids());  
  
 *// Give some extra time to make sure the stack traces get written.  
 // The system's been hanging for a minute, another second or two won't hurt much.* SystemClock.sleep(2000);  
  
 *// Pull our own kernel thread stacks as well if we're configured for that* **if** (RECORD\_KERNEL\_THREADS) {  
 dumpKernelStackTraces();  
 }  
  
 *// Trigger the kernel to dump all blocked threads, and backtraces on all CPUs to the kernel log* doSysRq(**'w'**);  
 doSysRq(**'l'**);  
  
 *// Try to add the error to the dropbox, but assuming that the ActivityManager  
 // itself may be deadlocked. (which has happened, causing this statement to  
 // deadlock and the watchdog as a whole to be ineffective)* Thread dropboxThread = **new** Thread(**"watchdogWriteToDropbox"**) {  
 **public void** run() {  
 mActivity.addErrorToDropBox(  
 **"watchdog"**, **null**, **"system\_server"**, **null**, **null**,  
 subject, **null**, stack, **null**);  
 }  
 };  
 dropboxThread.start();  
 **try** {  
 dropboxThread.join(2000); *// wait up to 2 seconds for it to return.* } **catch** (InterruptedException ignored) {}  
  
 IActivityController controller;  
 **synchronized** (**this**) {  
 controller = mController;  
 }  
 **if** (controller != **null**) {  
 Slog.i(TAG, **"Reporting stuck state to activity controller"**);  
 **try** {  
 Binder.setDumpDisabled(**"Service dumps disabled due to hung system process."**);  
 *// 1 = keep waiting, -1 = kill system* **int** res = controller.systemNotResponding(subject);  
 **if** (res >= 0) {  
 Slog.i(TAG, **"Activity controller requested to coninue to wait"**);  
 waitedHalf = **false**;  
 **continue**;  
 }  
 } **catch** (RemoteException e) {  
 }  
 }  
  
 *// Only kill the process if the debugger is not attached.* **if** (Debug.isDebuggerConnected()) {  
 debuggerWasConnected = 2;  
 }  
 **if** (debuggerWasConnected >= 2) {  
 Slog.w(TAG, **"Debugger connected: Watchdog is \*not\* killing the system process"**);  
 } **else if** (debuggerWasConnected > 0) {  
 Slog.w(TAG, **"Debugger was connected: Watchdog is \*not\* killing the system process"**);  
 } **else if** (!allowRestart) {  
 Slog.w(TAG, **"Restart not allowed: Watchdog is \*not\* killing the system process"**);  
 } **else** {  
 Slog.w(TAG, **"\*\*\* WATCHDOG KILLING SYSTEM PROCESS: "** + subject);  
 **for** (**int** i=0; i<blockedCheckers.size(); i++) {  
 Slog.w(TAG, blockedCheckers.get(i).getName() + **" stack trace:"**);  
 StackTraceElement[] stackTrace  
 = blockedCheckers.get(i).getThread().getStackTrace();  
 **for** (StackTraceElement element: stackTrace) {  
 Slog.w(TAG, **" at "** + element);  
 }  
 }  
 Slog.w(TAG, **"\*\*\* GOODBYE!"**);  
 Process.killProcess(Process.myPid());  
 System.exit(10);  
 }  
  
 waitedHalf = **false**;  
 }  
}

**EventLog.writeEvent(EventLogTags.WATCHDOG, subject);**

上述可以看到， 如果走到412行处。便是重启系统前的准备了。  
会进行以下事情：

1. 写Eventlog
2. 以追加的方式，输出system\_server和3个native进程的栈信息
3. 输出kernel栈信息
4. dump所有阻塞线程
5. 输出dropbox信息
6. 判断有没有debuger，没有的话，重启系统了，并输出log： \*\*\* WATCHDOG KILLING SYSTEM PROCESS:

### HandlerChecker

重要的类，HandlerChecker,这是Watchdog用来检测主线程，io线程，显示线程，UI线程的机制，代码也不长，直接贴出来吧。其原理就是通过各个Handler的looper的MessageQueue来判断该线程是否卡住了。当然，该线程是运行在SystemServer进程中的线程

HandlerChecker继承了Runnable，每个HandlerChecker在各自服务的主线程中运行并完成相应的检查，不会互相干扰, 每个通过addThread向watchdog注册自身的服务都对应一个HandlerChecker类实例

**public final class** HandlerChecker **implements** Runnable {  
 **private final** Handler mHandler;、、*// 监视器所在线程*  
 **private final** String mName; *// 线程名*  
 **private final long** mWaitMax; *// 最大等待阈值，超出则视为锁死*

*// 运行在线程上的监视器*  
 **private final** ArrayList<Monitor> mMonitors = **new** ArrayList<Monitor>();  
 **private boolean** mCompleted; *// 检测状态：完成、进行中*  
 **private** Monitor mCurrentMonitor; *// 当前检测的监视器*  
 **private long** mStartTime; *// 本轮检测开始时间，用于判断超时的相对起始时间点*  
  
 HandlerChecker(Handler handler, String name, **long** waitMaxMillis) {  
 mHandler = handler;  
 mName = name;  
 mWaitMax = waitMaxMillis;  
 mCompleted = **true**;  
 }  
  
 **public void** addMonitor(Monitor monitor) {  
 mMonitors.add(monitor);  
 }

#### scheduleCheckLocked

记录当前的开始时间, 一个核心的方法是mHandler.getLooper().getQueue().isPolling()

这个方法的实现在MessageQueue中，我将代码贴出来，我们可以看到上面的注释写到：返回当前的looper线程是否在polling工作来做，这个是个很好的用于检测loop是否存活的方法。我们从HandlerChecker源码可以看到

如果looper这个返回true，将会直接返回

若没有返回true，表明looper当前正在工作，会post一下自己，同时将mComplete置为false，标明已经发出一个消息正在等待处理。如果当前的looper没有阻塞，那很快，将会调用到自己的run方法

对于**Looper Checker**而言，会判断线程的消息队列是否处于空闲状态。 如果被监测的消息队列一直闲不下来，则说明可能已经阻塞等待了很长时间

**public void** scheduleCheckLocked() {  
 **if** (mMonitors.size() == 0 && mHandler.getLooper().getQueue().isPolling()) {  
 *// If the target looper has recently been polling, then  
 // there is no reason to enqueue our checker on it since that  
 // is as good as it not being deadlocked. This avoid having  
 // to do a context switch to check the thread. Note that we  
 // only do this if mCheckReboot is false and we have no  
 // monitors, since those would need to be executed at this point.* mCompleted = **true**;  
 **return**;  
 }  
  
 **if** (!mCompleted) {  
 *// we already have a check in flight, so no need* **return**;  
 }  
  
 mCompleted = **false**;  
 mCurrentMonitor = **null**;  
 mStartTime = SystemClock.uptimeMillis();

*// 将Monitor Checker的对象置于消息队列之前，优先运行*  
 mHandler.postAtFrontOfQueue(**this**);

*// 这里是WTD检测锁重要的实现方法，依靠向对应线程发送启动Monitor函数检测锁的超时情况，示意图中示意了几个线程中的Monitor执行过程*  
 }  
  
 **public boolean** isOverdueLocked() {  
 **return** (!mCompleted) && (SystemClock.uptimeMillis() > mStartTime + mWaitMax);  
 }

#### run()

对自己的Monitors遍历并进行monitor。（注：此处的monitor下面会讲到），若有monitor发生了阻塞，那么mComplete会一直是false。

定期调用被监护对象的monitor函数，这是在主线程中完成的。如果被监护对象死锁，则会阻塞在这里

@Override  
 **public void** run() {  
 **final int** size = mMonitors.size();///获取注册Watchdog 服务的  
Services 数量  
 **for** (**int** i = 0 ; i < size ; i++) {  
 **synchronized** (Watchdog.**this**) {  
 mCurrentMonitor = mMonitors.get(i);  
 }  
 mCurrentMonitor.monitor();  
 }  
  
 **synchronized** (Watchdog.**this**) {//**如果没有死锁一般都会走到这里**  
 mCompleted = **true**;// //**表示正常**  
 mCurrentMonitor = **null**;// //**没有死锁**  
 }  
 }  
}

如果被看门狗监护的服务对象发生了死锁，则线程会一直阻塞在这里。前

面提到了许多”死锁”，死锁？无非就是系统中各个进程互相抢占资源的过程中导

致的一种现象。对于死锁的产生原因非常多，比如说java 层死锁可能发生在调

用native 函数，而native 函数可能与硬件交互导致时间过长而没有返回，从而导

致长时间占用导致问题。具体问题具体分析

**public** String describeBlockedStateLocked() {  
 **if** (mCurrentMonitor == **null**) {  
 **return "Blocked in handler on "** + mName + **" ("** + getThread().getName() + **")"**;  
 } **else** {  
 **return "Blocked in monitor "** + mCurrentMonitor.getClass().getName()  
 + **" on "** + mName + **" ("** + getThread().getName() + **")"**;  
 }  
 }  
作者：Anderson大码渣  
链接：https://www.jianshu.com/p/5c18c4e8c826  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

#### getCompletionStateLocked

获取完成时间标识，那么在系统检测调用这个获取完成状态时，就会进入else里面，进行了时间的计算，并返回相应的时间状态码  
 **public int** getCompletionStateLocked() {  
 **if** (mCompleted) {  
 **return** COMPLETED;  
 } **else** {  
 **long** latency = SystemClock.uptimeMillis() - mStartTime;  
 **if** (latency < mWaitMax/2) {  
 **return** WAITING;  
 } **else if** (latency < mWaitMax) {  
 **return** WAITED\_HALF;  
 }  
 }  
 **return** OVERDUE;  
 }

#### 流程分析

其原理就是通过各个Handler的looper的MessageQueue（**isPolling**）来判断该线程是否卡住了。当然，该线程是运行在SystemServer进程中的线程。

如果**isPolling**这个返回true，将会直接返回。若没有返回true，表明looper当前正在工作，会post一下自己，同时将mComplete置为false，标明已经发出一个消息正在等待处理。如果当前的looper没有阻塞，那很快，将会调用到自己的run方法。

自己的run方法干了什么呢。干的是TAG: HandlerChecker源码里面的166行，里面对自己的Monitors遍历并进行monitor。（注：此处的monitor下面会讲到），若有monitor发生了阻塞，那么mComplete会一直是false。

那么在系统检测调用这个获取完成状态时，就会进入else里面，进行了时间的计算，并返回相应的时间状态码

好了，到这我们已经知道是怎么判断线程是否卡住的了

1. MessageQueue.isPolling
2. Monitor.monitor

### BinderThreadMonitor

## watchdog.init

**public void** init(Context context, ActivityManagerService activity) {  
 mResolver = context.getContentResolver();  
 mActivity = activity;  
  
 context.registerReceiver(**new** RebootRequestReceiver(),  
 **new** IntentFilter(Intent.ACTION\_REBOOT),  
 android.Manifest.permission.REBOOT, **null**);  
}

它会通过registerReceive（r ）方法注册一个广播接收reboot重启的请求

## 如何监控服务

Watchdog提供两种监视方式，一种是通过monitor()回调监视服务关键区是否出现死锁或阻塞，一种是通过发送消息监视服务主线程是否阻塞。

注意这里有两个检测项，一个是addMonitor，在每一个检测周期中watchdog会使用foreground thread的HandlerChecker回调服务注册的monitor()方法给服务的关键区上锁并马上释放，以检测关键区是否存在死锁或阻塞；另一个是addThread，watchdog会定时通过HandlerChecker向系统服务发送消息，以检测服务主线程是否被阻塞。这就是为什么在watchdog重启时有有两种提示语：“Block in Handler in ......”和“Block in monitor”，它们分别对应不同的阻塞类型。

1. Watchdog.getInstance().addMonitor(this);
2. Watchdog.getInstance().addThread(mHandler);

从上面分析可以知道，在watchdog的构造函数中将foreground thread、mian thread传入了一个HandlerChecker类。这个类就是watchdog检测超时的执行者。HandlerChecker类有多个实例，每个通过addThread向watchdog注册自身的服务都对应一个HandlerChecker类实例

---------------------

作者：宇落无痕

来源：CSDN

原文：https://blog.csdn.net/fu\_kevin0606/article/details/64479489

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：宇落无痕

来源：CSDN

原文：https://blog.csdn.net/fu\_kevin0606/article/details/64479489

版权声明：本文为博主原创文章，转载请附上博文链接！

### addMonitor

**public void** addMonitor(Monitor monitor) {  
 **synchronized** (**this**) {  
 **if** (isAlive()) {  
 **throw new** RuntimeException(**"Monitors can't be added once the Watchdog is running"**);  
 }  
 mMonitorChecker.addMonitor(monitor);  
 }  
}

### addThread

**public void** addThread(Handler thread) {  
 addThread(thread, DEFAULT\_TIMEOUT);  
}  
  
**public void** addThread(Handler thread, **long** timeoutMillis) {  
 **synchronized** (**this**) {  
 **if** (isAlive()) {  
 **throw new** RuntimeException(**"Threads can't be added once the Watchdog is running"**);  
 }  
 **final** String name = thread.getLooper().getThread().getName();  
 mHandlerCheckers.add(**new** HandlerChecker(thread, name, timeoutMillis));  
 }  
}

SystemServer 中被看门狗监控的三大服务：

1）、ActivityManagerService.java

2）、PowerManagerService.java

3）、WindowManagerService.java

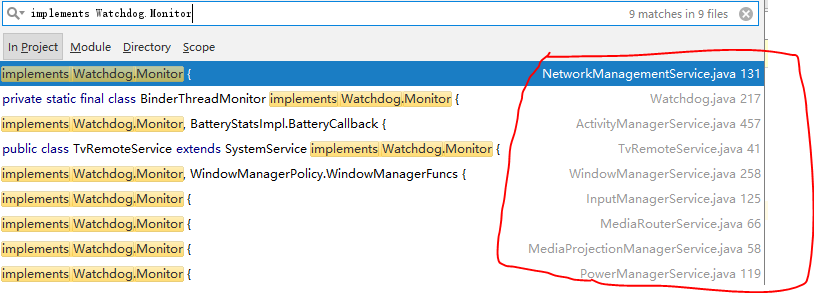
WatchDog 主要就是确保上述的服务发生死锁之后，退出  
SystemServer 进程，让init 进程重启它，让系统回到可用状态。  
由于上述服务实现看门狗的过程都类似，这里我们只以AMS 为例  
进行分析。

### 如何实现监控

addMonitor：添加监视器到mMonitorChecker上，它运行在FgThread线程上

addThread：创建对应线程的MonitorChecker，并添加到mMonitorCheckers中

利用Monitor，Monitor是一个接口，实现这个接口的类有很多



### 比如ActivityMS.java

AMS 实现接口Watchdog.Monitor

每个被监护的Service 服务必须实现看门狗的”Watchdog.Monitor”

接口。如`public final class ActivityManagerService extends

ActivityManagerNative implements Watchdog.Monitor,

BatteryStatsImpl.BatteryCallback` 。这个接口很简单， 就只有一个

monitor()方法。其次，在AMS 中实现的这个方法也只是锁一下对象，

什么都没有做，直接返回。如：

/\*\* In this method we try to acquire our lock to make sure that we have

not deadlocked \*/

public void monitor() {

synchronized (this) { }

}

从它的注释中也可以看出来“在这个方法中，我们试图获得锁，以确

保我们没有死锁。”

Note:实现完接口之后，其实，它在AMS 的构造函数中有这么两行代

码：

“

Watchdog.getInstance().addMonitor(this);

Watchdog.getInstance().addThread(mHandler);

“ ，其作用是把AMS 注册到Watchdog 服务中。

注冊方法:Watchdog.getInstance().addMonitor(this),并且实现了接口

*/\*\* In this method we try to acquire our lock to make sure that we have not deadlocked \*/***public void** monitor() {  
 **synchronized** (**this**) { }  
}

将自己注册进Watchdog. 同时其monitor方法只是同步一下自己，确保自己没有死锁。 干的事情虽然不多，但这足够了。足够让外部通过这个方法得到AMS是否死了

所以，有可能会出现，应用连续ANR或者崩溃，已经启动服务等操作交替出现的时候，超过了TIMEOUT定义的60s，Watchdog monitor在监视到服务持有的锁超过60s没有释放的情况下，认为SystemServer进程异常，强行退出SystemServer进程，由init进程重新启动SystemServer。

系统表现为，从开机动画处重新启动。

---------------------

作者：Danny\_姜

来源：CSDN

原文：https://blog.csdn.net/zxm317122667/article/details/8511647

版权声明：本文为博主原创文章，转载请附上博文链接！

## 总结

以上便是Android系统层Watchdog的原理了。设计的比较好。若由我来设计，我还真想不到使用Monitor那个锁机制来判断。

Watchdog的实现利用了锁和消息队列机制。当system\_server发生死锁或消息队列一直处于忙碌状态时，则认为系统已经没有响应了(System Not Responding)。

在分析Watchdog问题的时候，首先要有详尽的日志，其次要能定位出导致Watchdog超时的直接原因，最重要的是能还原出问题发生的场景

每个通过addThread向watchdog注册自身的服务都对应一个HandlerChecker类实例，那么通过addMonitor()注册的服务由谁来检查呢？答案就是前面出现的mMonitorChecker，也就是foreground thread的HandlerChecker。它除了需要检测主线程是否堵塞外，还需要回调系统服务注册的monitor()方法，以检测这些服务的关键区是否存在死锁或阻塞。

之所以不能在watchdog的主线程中回调monitor()方法，是由于如果被监控服务的关键区被占用，其monitor()方法可能需要一段时间才能返回。这样就无法保证watchdog每次个检测周期都是30s，所以必须交由foreground thread代为检查。

addMonitor()中会把每个monitor添加到mMonitorChecker也就是foreground thread的HandlerChecker中。除了它以外，所有HandlerChecker的mMonitors都是空的。

当watchdog的主循环开始运行后，每隔30秒，都会依次调用所有HandlerChecker的scheduleCheckLocked()方法。对于foreground thread的HandlerChecker，由于它的mMonitors不为空，需要它去锁各服务的monitor()来检查是否出现死锁，因此每个检测周期都要执行它。

对于其他的HandlerChecker，需要判断线程的Looper是否处于Idling，若为空就说明前一个消息已经执行完毕正在等下一个，消息循环肯定没阻塞，不用继续检测直接跳过本轮。

如果线程的消息循环不是Idling状态，说明服务的主线程正在处理某个消息，有阻塞的可能，就需要使用PostAtFrontOfQueue发出消息到消息队列，并记录下当前系统时间，同时将mComplete置为false，标明已经发出一个消息正在等待处理。

如果线程的消息队列没有阻塞，PostAtFrontOfQueue很快就会触发HandlerChecker的run方法。对于foreground thread的HandlerChecker，它会回调被监控服务的monitor方法，对其关键区上锁并马上释放，以检查是否存在死锁或阻塞。对于其他线程，仅需要将mComplete标记为true，表明消息已经处理完成即可。

如果服务的消息循环发生了堵塞，那么mComplete就会一直处于false状态。watchdog在每一个检测周期中都会一次调用每个HandlerChecker的getCompletionStateLocked方法检测超时时间，如果任何一个服务的主线程30s无响应就会提前输出其堆栈为重启做准备，如果60s无响应则进入重启流程

对于每个检测周期，首先需要将timeout计时器复位，而后依次检查在watchdog的init方法中注册的foreground thread，main thread，UI thread，i/o thread，以及其他通过addThread方法注册的服务的主线程是否阻塞。

检查主线程是否阻塞的方法是，如果线程Looper状态不是Idling，就通过HandlerChecker的postAtFrontOfQueue方法发送一个消息。稍后检测这个消息是否超时未返回。

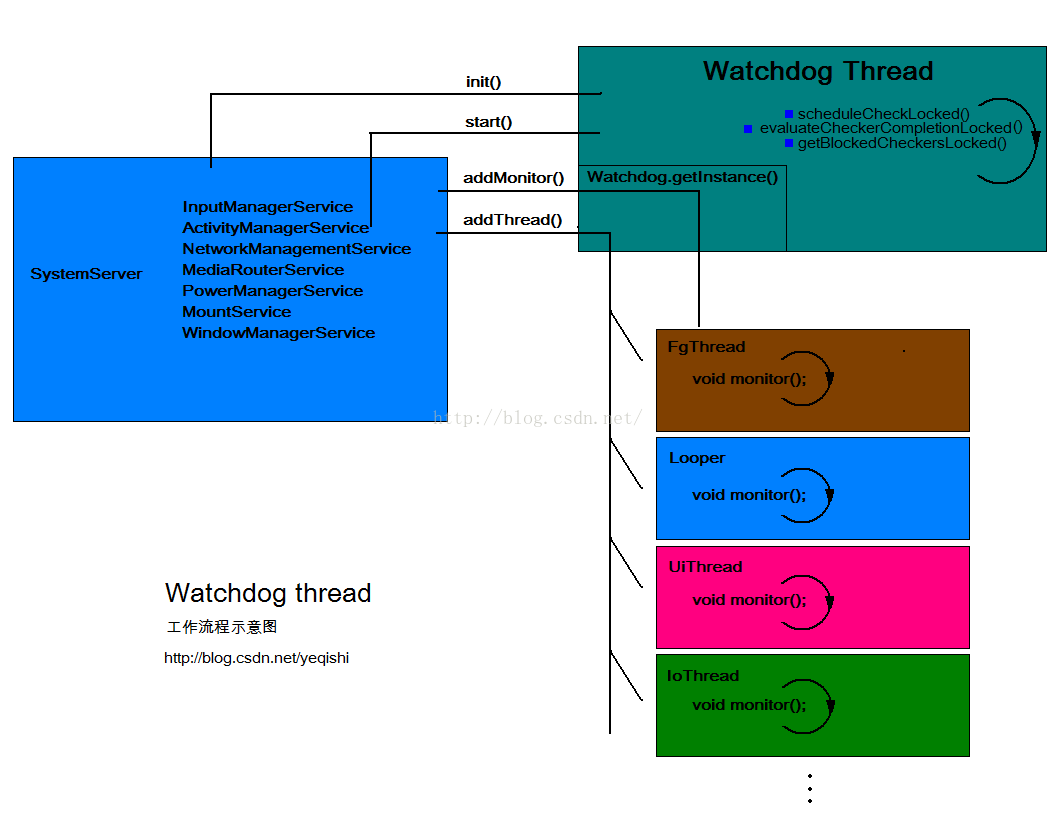
通过postAtFrontOfQueue送出消息后睡眠30s。注意这里使用uptimeMillis()计算时间，不计手机在睡眠中度过的时间。这是由于手机睡眠时系统服务同样也在睡眠，无法响应watchdog送出的消息，如果把睡眠时间计算在内当手机被再次唤醒时会导致watchdog认为时间已经过去了很久，从而发生误杀。

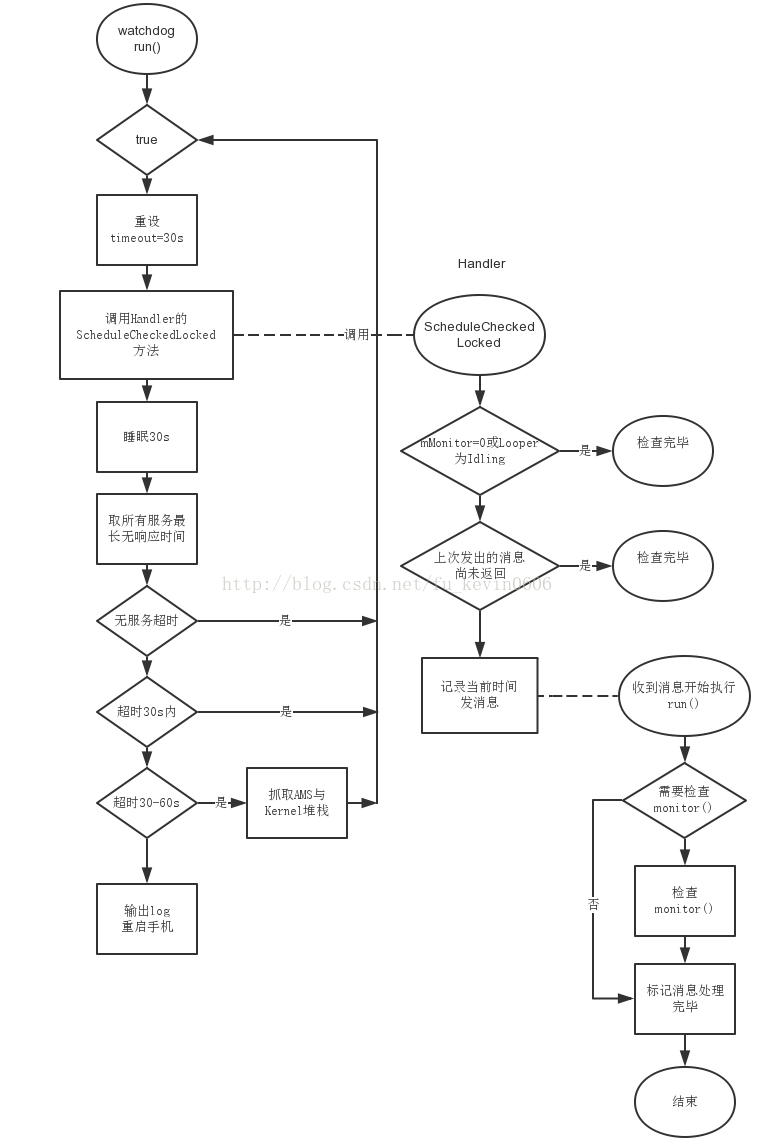
30秒等待完成后，就要检测之前送出的消息是否已经执行完毕。通过evaluateCheckerCompletionLocked遍历所有的HandlerChecker，返回最大的waitState值。waitState共有四种情况：COMPLETED对应消息已处理完毕线程无阻塞；WAITING对应消息处理花费0～29秒，需要继续运行；WAITED\_HALF对应消息处理花费30～59秒，线程可能已经被阻塞，需要保存当前AMS堆栈状态，用以在超时发生时输出堆栈；OVERDUE对应消息处理已经花费超过60s，此时便进入下一流程，输出堆栈信息并重启手机。

Watchdog超时已经发生，但之前evaluateCheckerCompletionLocked并不关心是哪个服务发生阻塞，仅仅返回所有服务最大的waitState值。此时需要调用getBlockedCheckersLocked判断具体是哪些应用发生了阻塞，阻塞的原因是什么。这就是我们在dropbox中看到的阻塞原因描述。而后依次输出AMS与Kernel调用堆栈。

输出dropbox，并检查activity controller连接的调试器是否可以处理这次watchdog无响应，如果activity controller不要求重启，那么就忽视这次超时，从头继续运行watchdog循环。杀死SystemServer并重启手机

### WTD工作流程





### 总时序图

SystemServer Watchdog Thread(this) HandlerChecker AMS

1 : addmonitor()

2 : new Watchdog()

3 : start()

4 : run()

5 : scheduleCheckLocked()

6 : timeout() 7 : mCurrentMonitor.monitor()

8 : mCompleted=true()

9 : Service is normal()

10 : Please Killing SystemService()

---------------------

接下来总结以下：

1. Watchdog是一个线程，用来监听系统各项服务是否正常运行，没有发生死锁
2. HandlerChecker用来检查Handler以及monitor
3. monitor通过锁来判断是否死锁
4. 超时30秒会输出log，超时60秒会重启（debug情况除外）

### **提供的问题定位方法**

1.ActivityManagerService.dumpStackTraces

2.dumpKernelStackTraces

# 分析方法

## 日志获取

Andriod的日志门类繁多，而且，为了调试的需要，设备厂商和应用开发者都会在AOSP的基础上增加很多日志。 面对如此庞大复杂的日志系统，通常只有对应领域的专家才能看懂其透露的细节信息，就像去医院就诊，医生一看检查报告就知道患者身体出了什么问题，而外行对这些诊断信息往往是束手无策的。

解决Watchdog相关的问题，对日志的要求比较高，有些问题与当时的系统环境相关，仅仅凭借单一的日志并不能定位问题。 以下罗列出获取Android日志的一些重要手段，部分场景下，Watchdog相关的问题甚至需要以下所有的日志：

* **logcat** 通过adb logcat命令输出Android的一些当前运行日志，可以通过logcat的 **-b** 参数指定要输出的日志缓冲区，缓冲区对应着logcat的一种日志类型。 高版本的logcat可以使用 **-b all** 获取到所有缓冲区的日志
  + event 通过android.util.EventLog工具类打印的日志，一些重要的系统事件会使用此类日志
  + main 通过android.util.Log工具类打印的日志，应用程序，尤其是基于SDK的应用程序，会使用此类日志
  + system 通过android.util.Slog工具类打印的日志，系统相关的日志一般都是使用此类日志，譬如SystemServer
  + radio 通过android.util.Rlog工具类打印的日志，通信模块相关的日志一般都是使用此类日志，譬如RIL
* **dumpsys** 通过adb dumpsys命令输出一些重要的系统服务信息，譬如内存、电源、磁盘等， 工作原理可以查阅[dumpsys介绍](http://duanqz.github.io/2015-07-19-Intro-to-dumpsys)一文
* **traces** 该文件记录了一个时间段的函数调用栈信息，通常在应用发生ANR(Application Not Responding)时，会触发打印各进程的函数调用栈。 站在Linux的角度，其实就是向进程发送SIGNAL\_QUIT(3)请求，譬如，我们可以通过adb shell kill -3 <pid>命令，打印指定进程的的trace。 SIGNAL\_QUIT(3)表面意思有一点误导，它其实并不会导致进程退出。输出一般在 \*/data/anr/traces.txt\* 文件中，当然，这是可以灵活配置的， Android提供的系统属性dalvik.vm.stack-trace-file可以用来配置生成traces文件的位置。
* **binder** 通过Binder跨进程调用的日志，可以通过adb shell cat命令从 /proc/binder 下取出对应的日志
  + failed\_transaction\_log
  + transaction\_log
  + transactions
  + stats
* **dropbox** 为了记录历史的logcat日志，Android引入了Dropbox，将历史日志持久化到磁盘中(**/data/system/dropbox**)。 logcat的缓冲区大小毕竟是有限的，所以需要循环利用，这样历史的日志信息就会被冲掉。在一些自动化测试的场景下，譬如Monkey需要长时间的运行， 就需要把历史的日志全都保存下来。
* **tombstone** tombstone错误一般由Dalvik错误、native层的代码问题导致的。当系统发生tombstone时，内核会上报一个严重的警告信号， 上层收到后，把当前的调用栈信息持久化到磁盘中(**/data/tombstone**)
* **bugreport** 通过adb bugreport命令输出，日志内容多到爆，logcat, traces, dmesg, dumpsys, binder的日志都包含在其中。 由于输出bugreport的时间很长，当系统发生错误时，我们再执行bugreport往往就来不及了(此时，系统可能都已经重启了)，所以，要动用bugreport就需要结合一些其他机制， 譬如在杀掉system\_server进程之前，先让bugreport运行完

## 问题定位

发生Watchdog检测超时这么重要的系统事件，Android会打印一个EventLog：

1. watchdog: Blocked in handler XXX *# 表示HandlerChecker超时了*
2. watchdog: Blocked in monitor XXX *# 表示MonitorChecker超时了*

Watchdog是运行在system\_server进程中，会打印一些System类型的日志。在手机处于非调试状态时，伴随Watchdog出现的往往是system\_server进程被杀，从而系统重启。 当Watchdog要主动杀掉system\_server进程时，以下关键字就会出现在SystemLog中：

1. Watchdog: \*\*\* WATCHDOG KILLING SYSTEM PROCESS: XXX
2. Watchdog: XXX
3. Watchdog: "\*\*\* GOODBYE!

当我们在日志中检索到上述两类关键信息时，说明“Watchdog显灵”了，从另一个角度来理解，就是“System Not Responding”了。 接下来，我们需要进一步定位在watchdog出现之前

system\_server进程在干什么，处于一个什么状态。 这与排除”Application Not Responding“问题差不多，我们需要进程的traces信息、当前系统的CPU运行信息、IO信息。

找到Watchddog出现之前的traces.txt文件，这个时间差最好不要太大，因为Watchdog默认的超时时间是1分钟，太久以前的traces并不能说明问题。 诱导Watchdong出现的直接原因其实就是system\_server中某个线程被阻塞了，这个信息在event和system的log中清晰可见。 我们以一个systemLog为例：

W Watchdog: \*\*\* WATCHDOG KILLING SYSTEM PROCESS: Blocked in monitor com.android.server.wm.WindowManagerService on foreground thread (android.fg)

Watchdog告诉我们**Monitor Checker**超时了，具体在哪呢？ 名为**android.fg**的线程在WindowManagerService的monitor()方法被阻塞了。这里隐含了两层意思：

1. WindowManagerService实现了Watchdog.Monitor这个接口，并将自己作为**Monitor Checker**的对象加入到了Watchdog的监测集中
2. monitor()方法是运行在**android.fg**线程中的。Android将**android.fg**设计为一个全局共享的线程，意味着它的消息队列可以被其他线程共享， Watchdog的**Monitor Checker**就是使用的**android.fg**线程的消息队列。因此，出现**Monitor Checker**的超时，肯定是**android.fg**线程阻塞在monitor()方法上。

我们打开system\_server进程的traces，检索 **android.fg** 可以快速定位到该线程的函数调用栈：

1. "android.fg" prio=5 tid=25 Blocked
2. | group="main" sCount=1 dsCount=0 obj=0x12eef900 self=0x7f7a8b1000
3. | sysTid=973 nice=0 cgrp=default sched=0/0 handle=0x7f644e9000
4. | state=S schedstat=( 3181688530 2206454929 8991 ) utm=251 stm=67 core=1 HZ=100
5. | stack=0x7f643e7000-0x7f643e9000 stackSize=1036KB
6. | held mutexes=
7. at com.android.server.wm.WindowManagerService.monitor(WindowManagerService.java:13125)
8. - waiting to lock <0x126dccb8> (a java.util.HashMap) held by thread 91
9. at com.android.server.Watchdog$HandlerChecker.run(Watchdog.java:204)
10. at android.os.Handler.handleCallback(Handler.java:815)
11. at android.os.Handler.dispatchMessage(Handler.java:104)
12. at android.os.Looper.loop(Looper.java:194)
13. at android.os.HandlerThread.run(HandlerThread.java:61)
14. at com.android.server.ServiceThread.run(ServiceThread.java:46)

**android.fg**线程调用栈告诉我们几个关键的信息：

* 这个线程当前的状态是**Blocked**，阻塞
* 由Watchdog发起调用monitor()，这是一个Watchdog检查，阻塞已经超时
* **waiting to lock <0x126dccb8>**： 阻塞的原因是monitor()方法中在等锁<0x126dccb8>
* **held by thread 91**： 这个锁被编号为91的线程持有，需要进一步观察91号线程的状态。

题外话：每一个进程都会对自己所辖的线程编号，从1开始。1号线程通常就是我们所说的主线程。 线程在Linux系统中还有一个全局的编号，由sysTid表示。我们在logcat等日志中看到的一般是线程的全局编号。 譬如，本例中android.fg线程在system\_server进程中的编号是25，系统全局编号是973。

可以在traces.txt文件中检索 tid=91 来快速找到91号线程的函数调用栈信息：

1. "Binder\_C" prio=5 tid=91 Native
2. | group="main" sCount=1 dsCount=0 obj=0x12e540a0 self=0x7f63289000
3. | sysTid=1736 nice=0 cgrp=default sched=0/0 handle=0x7f6127c000
4. | state=S schedstat=( 96931835222 49673449591 260122 ) utm=7046 stm=2647 core=2 HZ=100
5. | stack=0x7f5ffbc000-0x7f5ffbe000 stackSize=1008KB
6. | held mutexes=
7. at libcore.io.Posix.writeBytes(Native method)
8. at libcore.io.Posix.write(Posix.java:258)
9. at libcore.io.BlockGuardOs.write(BlockGuardOs.java:313)
10. at libcore.io.IoBridge.write(IoBridge.java:537)
11. at java.io.FileOutputStream.write(FileOutputStream.java:186)
12. at com.android.internal.util.FastPrintWriter.flushBytesLocked(FastPrintWriter.java:334)
13. at com.android.internal.util.FastPrintWriter.flushLocked(FastPrintWriter.java:355)
14. at com.android.internal.util.FastPrintWriter.appendLocked(FastPrintWriter.java:303)
15. at com.android.internal.util.FastPrintWriter.print(FastPrintWriter.java:466)
16. - locked <@addr=0x134c4910> (a com.android.internal.util.FastPrintWriter$DummyWriter)
17. at com.android.server.wm.WindowState.dump(WindowState.java:1510)
18. at com.android.server.wm.WindowManagerService.dumpWindowsNoHeaderLocked(WindowManagerService.java:12279)
19. at com.android.server.wm.WindowManagerService.dumpWindowsLocked(WindowManagerService.java:12266)
20. at com.android.server.wm.WindowManagerService.dump(WindowManagerService.java:12654)
21. - locked <0x126dccb8> (a java.util.HashMap)
22. at android.os.Binder.dump(Binder.java:324)
23. at android.os.Binder.onTransact(Binder.java:290)

91号线程的名字是**Binder\_C**，它的函数调用栈告诉我们几个关键信息：

* Native，表示线程处于运行状态(RUNNING)，并且正在执行JNI方法
* 在WindowManagerService.dump()方法申请了锁<0x126dccb8>，这个锁正是**android.fg**线程所等待的
* FileOutputStream.write()表示**Binder\_C**线程在执行IO写操作，正式因为这个写操作一直在阻塞，导致线程持有的锁不能释放

题外话：关于Binder线程。当Android进程启动时，就会创建一个线程池，专门处理Binder事务。线程池中会根据当前的binder线程计数器的值来构造新创建的binder线程, 线程名”Binder\_%X”，X是十六进制。当然，线程池的线程数也有上限，默认情况下为16，所以，可以看到 Binder\_1 ~ Binder\_F 这样的线程命名。

聪明的你看到这或许已经能够想到解决办法了，在这个IO写操作上加一个超时机制，并且这个超时小于Watchdog的超时，不就可以让线程释放它所占有的锁了吗？ 是的，这确实可以作为一个临时解决方案(Workaround)，或者说一个保护机制。但我们可以再往深处想一想，这个IO写操作为什么会阻塞：

* 是不是IO缓冲区满了，导致写阻塞呢？
* 是不是写操作有什么锁，导致这个write方法在等锁呢？
* 是不是当前系统的IO负载过于高，导致写操作效率很低呢？

这都需要我们再进一步从日志中去找原因。如果已有的日志不全，找不到论据，我们还需要设计场景来验证假设，解决问题的难度陡然上升。

## 3.3 场景还原

我们经历了两个关键步骤：

1. 通过event或system类型的日志，发现了Watchdog杀掉system\_server导致系统重启
2. 通过traces日志，发了导致Watchdog出现的具体线程操作

这两个过程基本就涵盖了Watchdog的运行机制了，但这并没有解决问题啊。我们需要找到线程阻塞的原因是什么，然而，线程阻塞的原因就千奇百怪了。 如果有问题出现的现场，并且问题可以重现，那么我们可以通过调试的手段来分析问题产生的原因。 如果问题只是偶然出现，甚至只有一堆日志，我们就需要从日志中来还原问题出现的场景，这一步才是真正考验大家Android/Linux功底的地方。

继续以上述问题为例，我们来进一步还原问题出现的场景，从Java层的函数调用栈来看：

* 首先，跨进程发起了Binder.dump()方法的调用：at android.os.Binder.dump(Binder.java:324)
* 然后，进入了WMS的dump()：at com.android.server.wm.WindowManagerService.dump(WindowManagerService.java:12654)
* 接着，发生了写文件操作：at java.io.FileOutputStream.write(FileOutputStream.java:186)
* 最后，调用了JNI方法：at libcore.io.Posix.writeBytes(Native method)

**Binder\_C**线程要出现这种函数调用栈，我们可以初步确定是Android接受了如下命令 (dumpsys原理请查阅[dumpsys介绍](http://duanqz.github.io/2015-07-19-Intro-to-dumpsys)一文)：

$ **adb** shell dumpsys window

当通过命令行运行以上命令时，客户端(PC)的adb server会向服务端(手机)的adbd发送指令， adbd进程会fork出一个叫做dumpsys的子进程，dumpsys进程再利用Binder机制和system\_server通信 (adb的实现原理可以查阅[adb介绍](http://duanqz.github.io/2015-05-21-Intro-adb)一文)。

仅凭这个还是分析不出问题所在，我们需要启用内核的日志了。当调用JNI方法libcore.io.Posix.writeBytes()时，会触发系统调用， Linux会从用户态切换到内核态，内核的函数调用栈也可以从traces中找到：

1. kernel: \_\_switch\_to+0x74/0x8c
2. kernel: pipe\_wait+0x60/0x9c
3. kernel: pipe\_write+0x278/0x5cc
4. kernel: do\_sync\_write+0x90/0xcc
5. kernel: vfs\_write+0xa4/0x194
6. kernel: SyS\_write+0x40/0x8c
7. kernel: cpu\_switch\_to+0x48/0x4c

在Java层，明确指明要写文件(FileOutputStream)，正常情况下，系统调用write()就完事了，但Kernel却打开了一个管道，最终阻塞在了pipe\_wait()方法。 什么场景下会打开一个管道，而且管道会阻塞呢？一系列的猜想和验证过程接踵而至。

这里有必要先补充一些基础知识了：

* [**Linux进程间通信之管道(pipe)**](http://www.cnblogs.com/biyeymyhjob/archive/2012/11/03/2751593.html)

Linux的管道实现借助了文件系统的file结构和VFS(Virtual File System)，通过将两个file结构指向同一个临时的VFS索引节点，而这个VFS索引节点又指向一个物理页面时， 实际上就建立了一个管道。

这就解释了为什么发起系统调用write的时候，打开了一个管道。因为dumpsys和system\_server进程，将自己的file结构指向了同一个VFS索引节点。

* [**管道挂起的案例**](http://blog.csdn.net/sj13051180/article/details/47865803)

管道是一个生产者-消费者模型，当缓冲区满时，则生产者不能往管道中再写数据了，需等到消费者读数据。如果消费者来不及处理缓冲区的数据，或者锁定缓冲区，则生产者就挂起了。

结合到例子中的场景，system\_server进程无法往管道中写数据，很可能是dumpsys进程一直忙碌来不及处理新的数据。

接下来，需要再从日志中寻找dumpsys进程的运行状态了：

* 是不是dumpsys进程的负载太高？
* 是不是dumpsys进程死掉了，导致一直没有处理缓冲区数据？
* 是不是dumpsys进程有死锁？

接下来的分析过程已经偏离Watchdog机制越来越远了，我们点到为止。

小伙伴们可以看到，场景还原涉及到的知识点非常之宽泛，而且有一定的深度。在没有现场的情况下，伴随一系列的假设和验证过程，充满了不确定性和发现问题的喜悦。 正所谓，同问题做斗争，其乐无穷！

**至此，我们分析Watchdog问题的惯用方法，回答前面提出来的第二个问题：**

**通过event或system类型的logcat日志，检索Watchdog出现的关键信息；通过traces，分析出导致Watchdog检查超时的直接原因；通过其他日志，还原出问题出现的场景。**

# 实例分析

在上面介绍Watchdog问题分析方法的时候，我们其实已经举了一个例子。通常，比较容易定位导致Watchdog出现的直接原因(Direct Cause)，但很难找到更深层次的原因(Root Cause)。 这个小节，我们再介绍一个实例，来分析Watchdog出现的另一种场景。诚然，仅凭几个例子，远不够涵盖Watchdog的所有问题，我们的章法还是按照一定的方法论来深究问题。

## Dhfg

回顾一下解决问题三部曲：

1. 日志获取。日志种类繁多，分析Watchdog问题，宁滥毋缺
2. 问题定位。从logcat中锁定watchdog的出现，从traces锁定直接原因
3. 场景还原。结合各类日志，不断假设验证

**以CPU占用过高的场景为例：**[**下载该问题的全部日志**](http://duanqz.github.io/2015-10-12-Watchdog-Analysis)

**从sys\_log中，检索到了Watchdog的出现关键信息**

TIPS: 在sys\_log中搜索关键字”WATCHDOG KILLING SYSTEM PROCESS”

10-14 17:10:51.548 892 1403 W Watchdog: \*\*\* WATCHDOG KILLING SYSTEM PROCESS: Blocked in handler on ActivityManager (ActivityManager)

这是一个Watchdog的**Looper Checker**超时，由于ActivityManager这个线程一直处于忙碌状态，导致Watchdog检查超时。 Watchdog出现的时间是**10-14 17:10:51.548**左右，需要从traces.txt中找到这个时间段的system\_server进程的函数调用栈信息， system\_server的进程号是892。

**从traces.txt中找到对应的函数调用栈**

traces.txt包含很多进程在不同时间段的函数调用栈信息，为了检索的方便，首先可以将traces.txt分块。 笔者写了一个[工具](https://github.com/duanqz)，可以从traces.txt文件中分割出指定进程号的函数调用栈信息。

TIPS: 在system\_server的traces中(通过工具分割出的system\_server\_892\_2015-10-14-17:09:06文件)搜索关键字”ActivityManager”

1. "ActivityManager" prio=5 tid=17 TimedWaiting
2. | group="main" sCount=1 dsCount=0 obj=0x12c0e6d0 self=0x7f84caf000
3. | sysTid=938 nice=-2 cgrp=default sched=0/0 handle=0x7f7d887000
4. | state=S schedstat=( 107864628645 628257779012 60356 ) utm=7799 stm=2987 core=2 HZ=100
5. | stack=0x7f6e68f000-0x7f6e691000 stackSize=1036KB
6. | held mutexes=
7. at java.lang.Object.wait!(Native method)
8. - waiting on <0x264ff09d> (a com.android.server.am.ActivityManagerService$5)
9. at java.lang.Object.wait(Object.java:422)
10. at com.android.server.am.ActivityManagerService.dumpStackTraces(ActivityManagerService.java:5395)
11. at com.android.server.am.ActivityManagerService.dumpStackTraces(ActivityManagerService.java:5282)
12. at com.android.server.am.ActivityManagerService$AnrActivityManagerService.dumpStackTraces(ActivityManagerService.java:22676)
13. at com.mediatek.anrmanager.ANRManager$AnrDumpMgr.dumpAnrDebugInfoLocked(SourceFile:1023)
14. at com.mediatek.anrmanager.ANRManager$AnrDumpMgr.dumpAnrDebugInfo(SourceFile:881)
15. at com.android.server.am.ActivityManagerService.appNotResponding(ActivityManagerService.java:6122)
16. - locked <0x21c77912> (a com.mediatek.anrmanager.ANRManager$AnrDumpRecord)
17. at com.android.server.am.BroadcastQueue$AppNotResponding.run(BroadcastQueue.java:228)
18. at android.os.Handler.handleCallback(Handler.java:815)
19. at android.os.Handler.dispatchMessage(Handler.java:104)
20. at android.os.Looper.loop(Looper.java:192)
21. at android.os.HandlerThread.run(HandlerThread.java:61)
22. at com.android.server.ServiceThread.run(ServiceThread.java:46)

ActivityManager线程实际上运行着AMS的消息队列，这个函数调用栈的关键信息：

* 线程状态为TimedWaiting, 这表示当前线程阻塞在一个超时的wait()方法
* 正在处理广播消息超时发生的ANR(Application Not Responding)，需要将当前的函数调用栈打印出来
* 最终在<0x264ff09d>等待，可以从[AMS的源码](https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/am/ActivityManagerService.java#4830) 中找到这一处锁的源码，因为dumpStackTraces()会写文件，所以AMS设计了一个200毫秒的超时锁。

observer**.**wait**(**200**);** *// Wait for write-close, give up after 200msec*

**还原问题的场景**

从ActivityManager这个线程的调用栈，我们就会有一些疑惑：

* 是哪个应用发生了ANR？为什么会发生ANR？
* 超时锁只用200毫秒就释放了，为什么会导致Watchdog检查超时？(AMS的Looper默认超时是1分钟)

带着这些疑惑，我们再回到日志中：

从sys\_log中，可以检索到Watchdog出现的时间点(**17:10:51.548**)之前，com.android.systemui发生了ANR，从而引发AMS打印函数调用栈:

TIPS: 在sys\_log中检索”ANR in”关键字或在event\_log中检索”anr”关键字

1. 10-14 17:10:04.215 892 938 E ANRManager: ANR in com.android.systemui, time=27097912
2. 10-14 17:10:04.215 892 938 E ANRManager: Reason: Broadcast of Intent { act=android.intent.action.TIME\_TICK flg=0x50000114 (has extras) }
3. 10-14 17:10:04.215 892 938 E ANRManager: Load: 89.22 / 288.15 / 201.91
4. 10-14 17:10:04.215 892 938 E ANRManager: Android time :[2015-10-14 17:10:04.14] [27280.396]
5. 10-14 17:10:04.215 892 938 E ANRManager: CPU usage from 17016ms to 0ms ago:
6. 10-14 17:10:04.215 892 938 E ANRManager: 358% 23682/float\_bessel: 358% user + 0% kernel
7. 10-14 17:10:04.215 892 938 E ANRManager: 57% 23604/debuggerd64: 3.8% user + 53% kernel / faults: 11369 minor
8. 10-14 17:10:04.215 892 938 E ANRManager: 2% 892/system\_server: 0.9% user + 1% kernel / faults: 136 minor

从这个日志信息中，我们两个疑惑就释然了：

发生ANR之前的CPU负载远高于正常情况好几倍(Load： 89.22 / 288.15 / 201.91)，在这种CPU负载下，com.android.systemui进程发生处理广播消息超时(Reason: Broadcast of Intent)再正常不过了。 在这之前CPU都被**float\_bessel**这个进程给占了，这货仅凭一己之力就耗了358%的CPU资源。

observer.wait(200)在调用后，便进入排队等待唤醒状态(Waiting)，在等待200毫秒后，便重新开始申请CPU资源，而此时，CPU资源一直被**float\_bessel**占着没有释放，所以该线程一直在等CPU资源。 等了1分钟后，Watchdog跳出来说“不行，你已经等了1分钟了，handler处理其他消息了”。

在多核情况下，CPU的使用率统计会累加多个核的使用率，所以会出现超过100%的情况。那么**float\_bessel**究竟是什么呢？它是一个Linux的测试样本，贝塞尔函数的计算，耗的就是CPU。

这样，该问题的场景我们就还原出来了：在压力测试的环境下，CPU被**float\_bessel**运算占用，导致com.android.systemui进程发生ANR，从而引发AMS打印trace; 但由于AMS一直等不到CPU资源，Watchdog检测超时，杀掉system\_server进程，系统重启。

对于压力测试而言，我们一般会设定一个通过标准，在某些压力情况下，出现一些错误是允许的。对于Android实际用户的使用场景而言，本例中的压力通常是不存在的，所以在实际项目中，这种类型的Watchdog问题，我们一般不解决。

## Android看门狗设计

<https://blog.csdn.net/yunmenggyy/article/details/50320469>

## Android Watchdog提前启动

https://blog.csdn.net/yeqishi/article/details/50262699

注：抓取MTK3710 的开关机log，查看ksernerl log 对比下面的信息

看门狗在检查各个Service 的过程中，分下面三种情形做出一次举动，

第一种情形已经过验证，下面两种情形还未实验：

1）、正常情况下，tick 300s, 对应count=10.

[ 66.841723]: (0)[1147:watchdog]AEEIOCTL\_RT\_MON\_Kick ( 300)

[ 66.841753]: (0)[1147:watchdog][Hang\_Detect] hang\_detect enabled 10

2）、在dump backtrace 时，tick 600s, 对应count=20.

[ 258.218145] (0)[1322:watchdog]AEEIOCTL\_RT\_MON\_Kick ( 600)

[ 258.218171] (0)[1322:watchdog][Hang\_Detect] hang\_detect enabled 20

3）、在SWT 发生的情况下，tick 720s, 对应count=24.

[ 299.046542] (0)[1322:watchdog]AEEIOCTL\_RT\_MON\_Kick ( 720)

[ 299.046572] (0)[1322:watchdog][Hang\_Detect] hang\_detect enabled 24

其次，一条可以快速定位是否是进程卡住的问题，正常情况下Kernel threrad

打印的信息：

[ 60.561702]: (0)[118:hang\_detect][Hang\_Detect] hang\_detect thread counts down

10:10.

原文：https://blog.csdn.net/Toc\_SunWinner/article/details/79314732

版权声明：本文为博主原创文章，转载请附上博文链接！

# 参考

Watchdog机制以及问题分析

https://blog.csdn.net/lezhang123/article/details/72537647

Android7.0 Watchdog机制

<https://blog.csdn.net/fu_kevin0606/article/details/64479489>

Android系统层Watchdog机制源码分析

<https://www.jianshu.com/p/5c18c4e8c826>