# TODO

HW Watchdog

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

<https://blog.csdn.net/zxm317122667/article/details/8511647>

<https://blog.csdn.net/yeqishi/article/details/50262087>

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

# 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.

## Android系统层看门狗

　　看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.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。

接下来我们就看看Android系统的Watchdog是怎么设计的。

在android 系统中，看门狗可以分成以下两种类型。

1）、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。

2）、System Server Watchdog：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。

# 启动

Watchdog的初始化位于SystemServer.  
[/frameworks/base/services/java/com/android/server/SystemServer.java](https://link.jianshu.com/?t=http%3A%2F%2Fandroidxref.com%2F6.0.0_r5%2Fxref%2Fframeworks%2Fbase%2Fservices%2Fjava%2Fcom%2Fandroid%2Fserver%2FSystemServer.java)

## SystemServer. Otherservice()

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

## Watchdog

**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

会去调用子线程HandlerChecker 检查

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。

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

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

**public final class** HandlerChecker **implements** Runnable {  
 **private final** Handler mHandler;  
 **private final** String mName;  
 **private final long** mWaitMax;  
 **private final** ArrayList<Monitor> mMonitors = **new** ArrayList<Monitor>();  
 **private boolean** mCompleted;  
 **private** Monitor mCurrentMonitor;  
 **private long** mStartTime;  
  
 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

**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();  
 mHandler.postAtFrontOfQueue(**this**);  
 }  
  
 **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。  
 @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

SystemServer 中被看门狗监控的三大服务：

1）、ActivityManagerService.java

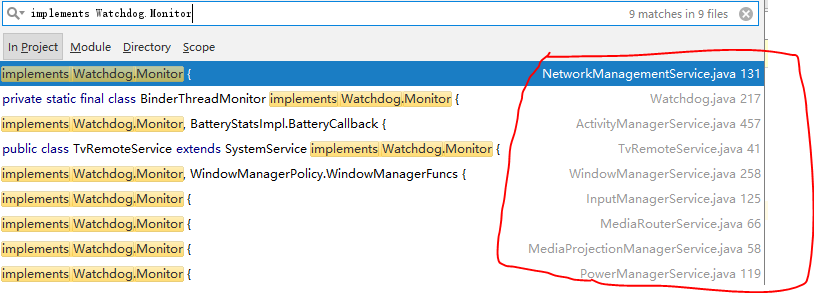
2）、PowerManagerService.java

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

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

## 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。

### 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）

# 常见问题分析

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

Android系统层Watchdog机制源码分析

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