**Android 异常解决方法汇总**

（1）异常：Android中引入第三方Jar包的方法(java.lang.NoClassDefFoundError解决办法)

1、在工程下新建lib文件夹，将需要的第三方包拷贝进来。  
2、将引用的第三方包，添加进工作的build path。选中jar包->Build Path  
3、（关键的一步）将lib设为源文件夹。如果不设置，则程序编译可以通过，但运行的时候，会报：选中lib文件夹->source codejava.lang.NoClassDefFoundErrorAndroid中引入第三方Jar包的方法(java.lang.NoClassDefFoundError解决办法) （2）异常：**Caused** **by:** **android.os.**TransactionTooLargeException导致原因是：Binder传输的数据太大如果Binder的参数或返回值太大，不适合的事务缓冲区，然后调用将失败，并将被抛出TransactionTooLargeException。解决方法：不要将大量数据传入Binder （3）异常：android.database.CursorWindowAllocationException:Cursor window allocation of 2048 kb failed导致原因：主要原因是因为使用了SimpleCursorAdapter类，其中的Cursor参数需要我们手动关闭，否则就会出现以上异常。解决方法：cursor.close() （4）异常：Excessive JNI global references错误的解决方案  GREF暴增导致原因：vm对jni层的reference有个数限制，过多很造成VM aborting。因此每次在GREF增加到2000以上的时候就直接aborting了 解决方法：及时回收或关闭引用  （5）异常：android.view.WindowManager$BadTokenException: Unable to add window -- token 导致原因：使用getApplicationContext()获得的Context,而必须使用Activity,因为只有一个Activity才能添加一个窗体。解决方法：采用当前Activity的Context （6）异常：android.database.sqlite.SQLiteCantOpenDatabaseException: unable to open database file导致原因：重复安装软件，导致Sqlite3打开数据库异常解决方法：卸载相应软件，开关机，重新安装该软件，即可以解决相关异常。  
  （7）异常：java.lang.NullPointerException 异常原因：使用fragment时，fragment相关布局，必须给每个view设置id，否则通过activity获得fragment的实例时会出现以上异常解决方法：为fragment布局的每个view设置ID （8）异常：java.util.ConcurrentModificationException（并发操作异常） 异常原因：ArrayList是非线程安全的，当同时在遍历和修改ArrayList时，就会出现该异常解决方法：使用Vector替换ArrayList，Vector是线程安全的。Vector的缺点：大量数据操作时，由于线程安全，性能比ArrayList低 （9）异常：so库文件异常（JNI层异常） 异常原因：在Android应用层开发中，我们经常会使用so库文件。当so库文件发生异常时，我们只能在adb log中发现fatal libc这样的异常信息，仅此而已，并不能发现异常调用的逻辑关系， 这是我们就需要查看so 库异常Log信息。在Android手机的/data/tombstones 的路径下就包含了so库文件发生异常的Log信息，开发者可以查看该路径下载Log文件发现异常调用的逻辑关系。  
(10) 异常：**android.os.**deadObjectException异常deadObjectException异常，说明应用的service已经停止，要么是从操作系统中丧生，要么从应用程序中终止。

**Watchdog**机制：

由于Android的SystemServer内有一票重要Service，所以在进程内有一个软件实现的**Watchdog**机制，用于监视SystemServer中各Service是否正常工作。如果超过一定时间（默认30秒），就dump现场便于分析，再超时（默认60秒）就重启SystemServer保证系统可用性。同时logcat中会打印类似下面信息：W **Watchdog**: \*\*\* **WATCHDOG** KILLING SYSTEM PROCESS: Blocked in monitor com.android.server.am.ActivityManagerService on foreground thread (android.fg), Blocked in handler on ActivityManager (ActivityManager), Blocked in handler on WindowManager thread (WindowManager)主要实现代码位于/frameworks/base/services/core/java/com/android/server/**Watchdog**.java和/frameworks/base/core/jni/android\_server\_**Watchdog**.cpp。大体的框架很简单。**Watchdog**是SystemServer中的独立线程，它隔一定时间间隔会向各监视线程调度一次检查操作。这个检查操作当中会调用已注册的Monitor对象。如果Monitor对象上产生死锁，或是关键线程hang住，那么该检查必定不能按时结束，这样就被**Watchdog**检查到。先来看看总体类图。因为是唯一的，**Watchdog**实现为Singleton。其中维护HandlerChecker数组，对应要检查的线程。HandlerChecker数组中有Monitor数组，对应要检查的Monitor对象。要接受检查的对象需要实现Monitor接口。初始化是从SystemServer的startOtherServices()中开始的，其大体流程如下：首先，在SystemServer.java中，会创建**Watchdog**并启动。472 Slog.i(TAG, "Init **Watchdog**");  
473 final **Watchdog** **watchdog** = **Watchdog**.getInstance();  
474 **watchdog**.init(context, mActivityManagerService);  
…  
1120 **Watchdog**.getInstance().start();在**Watchdog**的构造函数中，会为每个要检查的线程创建HandlerChecker，并加到mHandlerCheckers这个队列中。首先是FgThread。它继承自ServiceThread，是一个单例，负责那些常规的前台操作，它不应该被后台的操作所阻塞。在**Watchdog**.java中：215 mMonitorChecker = new HandlerChecker(FgThread.getHandler(),  
216 "foreground thread", DEFAULT\_TIMEOUT);  
217 mHandlerCheckers.add(mMonitorChecker);接下来，对于System Server的主线程，UI线程，IO线程和Display线程，分别做相同操作，这坨线程和FgThread一样都继承自ServiceThread。在init()函数中，接下来会调用registerReceiver()来注册系统重启的BroadcastReceiver。在收到系统重启广播时会执行RebootRequestReceiver的onReceive()函数，继而调用rebootSystem()重启系统。它允许其它模块（如CTS）通过发广播来让系统重启。然后，各个需要被**Watchdog**监视的Service需要将自己进行注册。它们都实现了**Watchdog**.Monitor接口，其中主要是monitor()函数。例如ActivityManagerService：2150 **Watchdog**.getInstance().addMonitor(this);  
2151 **Watchdog**.getInstance().addThread(mHandler);其中addMonitor()将自身放到foreground thread的HandlerChecker的monitor队列中，addThread()根据当前线程的Handler创建HandlerChecker并放到mHandlerCheckers队列中。monitor()的实现一般很简单，只是尝试去获得锁再释放锁。如果有deadlock，就会卡住无法返回。18767 public void monitor() {  
18768 synchronized (this) { }  
18769 }回到SystemServer中，通过**Watchdog**的start()方法启动**Watchdog**线程，**Watchdog**.run()被执行。**Watchdog**的主体是一个循环。在每一次循环中，所有HandlerChecker的scheduleCheckLocked()函数被调用。其中主要是往被监视线程的Looper中放入HandlerChecker对象，HandlerChecker本身作为Runnable，是线程的可执行体。因此当被监视线程把它从Looper中拿出来后，它的run()函数被调用。然后，**Watchdog**.run()等待最长30秒后，调用evaluateCheckerCompletionLocked()检查各HandlerChecker结果状态。一个HandlerChecker结果状态有四种，COMPLETED(0)，WAITING(1), WAITED\_HALF(2)和OVERDUE(3)。分别代表目标Looper已处理monitor，延时小于30秒，延时大于30秒小于60秒，延时大于60秒。最终的总状态是它们的最大值（也就是意味着最坏情况的那种情况）。如果总状态是COMPLETED，WAITING或WAITED\_HALF，则进入循环下一轮。注意如果是WAITED\_HALF，也就是说等了大于30秒，需要调用AMS.dumpStackTraces()来dump栈。如果状态为WAITED\_HALF，进入下一轮循环后，又会等待最长30秒。假设有线程阻塞，对于阻塞线程的HandlerChecker，它的延迟超过60秒，导致总状态为OVERDUE。这里会调用getBlockedCheckersLocked()和describeCheckersLocked()打印出是哪个Handler阻塞了。在Eventlog中打出信息后，把当前pid和相关进程pid放入待杀列表。然后和上面一样调用AMS.dumpStackTraces()打印栈。之后等待2秒等stacktrace写完。如有需要还会调用dumpKernelStackTraces()将kernel部分的stacktrace打出来。本质上是读取/proc//task下的线程的和相应的/proc//stack文件。然后调用doSyncRq()通知kernel把阻塞线程信息和backtrace打印出来（通过写/proc/**sysrq**-trigger）。之后会创建专门的线程来将信息写入到Dropbox，该线程会执行AMS.addErrorToDropBox()。Dropbox是Android中的一套日志记录系统，作用是将系统出错信息记录下来并且保留一段时间，避免被覆盖。当出现crash, wtf, lowmem或者**Watchdog**被触发都会通过Dropbox来记录。  
425 Thread dropboxThread = new Thread("watchdogWriteToDropbox") {  
426 public void run() {  
427 mActivity.addErrorToDropBox(  
428 "**watchdog**", null, "**system\_server**", null, null,  
429 subject, null, stack, null);  
430 }  
431 };  
432 dropboxThread.start();  
433 try {  
434 dropboxThread.join(2000); // wait up to 2 seconds **for** it to return.  
435 } catch (InterruptedException ignored) {}DropBoxManagerService在SystemServer的startOtherServices()中添加，信息默认存放路径为/data/system/dropbox。DropBoxManagerService实现了IDropBoxManagerService服务接口，Client通过DropBoxManager来访问服务。在**Watchdog**中调用AMS.addErrorToDropBox()后，该函数会起工作线程（因为涉及I/O），将前面得到的stack信息dump到Dropbox，并且获取最近的logcat，最后通过DBMS的addText()接口进行存储。 接下来，如果设置了ActivityController就调用其systemNotResponding()接口（IActivityController是给测试开发时用的接口，用于监视AMS里的行为）。然后判断Debugger是否连着和是否允许restart。如果没有连着debugger且允许restart，就开始大开杀戒了。467 Slog.w(TAG, "\*\*\* **WATCHDOG** KILLING SYSTEM PROCESS: " + subject);  
...  
476 Slog.w(TAG, "\*\*\* GOODBYE!");  
477 Process.killProcess(Process.myPid());  
478 System.exit(10);因为**Watchdog**和SystemServer是同一进程，这里**Watchdog** kill掉了自己，也就是kill了SystemServer。因它是主要进程，杀掉后会被init重启。这就是**Watchdog**的大体流程，回过头看下AMS中dumpStackTraces()的一些细节。参数中的pids包含了本进程，阻塞线程以及phone进程等。NATIVE\_STACKS\_OF\_INTEREST包含了下面三个关键进程。67 public static final String[] NATIVE\_STACKS\_OF\_INTEREST = new String[] {  
68 "/system/bin/mediaserver",  
69 "/system/bin/sdcard",  
70 "/system/bin/surfaceflinger"  
71 };注意虽然它们不在SystemServer中，但因为SystemServer中的Service会用Binder同步调用它们的方法。如果这些进程中阻塞，也可能导致SystemServer中发生阻塞。dumpStackTraces()实现中先从dalvik.vm.stack-trace-file这个system property中取出trace路径，默认为/data/anr/traces.txt。接着它创建该文件（需要的话），设置属性，最后调用同名函数dumpStackTraces()完成真正的dump工作。dump工作首先会用FileObserver（利用inotify机制）监视trace文件什么时候写完。它会创建一个独立的线程ObserverThread并运行。然后对于前面加入到要dump线程列表中的进程，发送SIGQUIT信号。如果是虚拟机进程，ART中的SignalCatcher::HandleSigQuit()（在/art/runtime/signal\_catcher.cc）会被调用来dump信息（DVM也是类似的）。对于前面的核心Service，调用Debug.dumpNativeBacktraceToFile()来输出它们的backtrace。总结下来，dumpStackTraces()大体流程如下：可以看到，其中主要收集三类信息：一是关键进程（也就是上面收集的pid）的stacktrace；二是几个关键native服务的stacktrace；三是CPU的使用率。其中一是通过往目标进程发SIGQUIT来获取，因为Java虚拟机的Runtime会捕获SIGQUIT信号打印栈信息。二的原理是向后台debuggerd这个daemon发起申请，让其用ptrace打印目标进程的stacktrace然后用本地socket传回来。部分实现位于android\_os\_Debug.cpp和/system/core/libcutils/debugger.c。发起申请和接收数据的代码在以下函数：131int dump\_backtrace\_to\_file\_timeout(pid\_t tid, int fd, int timeout\_secs) {  
132 int sock\_fd = make\_dump\_request(DEBUGGER\_ACTION\_DUMP\_BACKTRACE, tid, timeout\_secs);  
...  
137 /\* Write the data read from the socket to the fd. \*/  
...  
141 while ((n = TEMP\_FAILURE\_RETRY(read(sock\_fd, buffer, sizeof(buffer)))) > 0) {  
142 if (TEMP\_FAILURE\_RETRY(write(fd, buffer, n)) != n) {  
143 result = -1;  
144 break;  
145 }  
146 }  
...最后使用ProcessCpuTracker类测量CPU使用率。它主要是通过读系统的/proc//stat文件。里边可以读到进程所占用的时间（user mode和kernel mode）。统计半秒后，排序后输出最占CPU的前几名的stacktrace以便分析谁可能是罪魁祸首。总得来说，**Watchdog**是一个软件实现的检测SystemServer进程内死锁或挂起问题，并能够从中恢复的机制。除了**Watchdog**外，Android中还有一些自检容错及出错信息收集机制，前者有ANR，OOM Killer，init中的重启机制等，后者有Dropbox，Debuggerd，Eventlog，Bugreport等。除此之外，其它的信息查看和调试命令就不计其数了，如dumpsys, dumpstate, showslab, procrank, procmem, latencytop, librank, schedtop, svc, am ,wm, atrace, proc, pm, service, getprop/setprop, logwrapper, input, getevent/sendevent等。充分利用这些工具能够有效提高分析问题的效率。

您查询的关键词是：**[background](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4" \l "baidusnap0)**[**partial**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap1)[**concurrent**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap2)[**mark**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap3)[**sweep**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap4)[**gc**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap5)[**freed**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap6)[**12**](http://cache.baiducontent.com/c?m=9f65cb4a8c8507ed4fece763105392230e54f7307c95974968d4e419ce3b46120720b8ea79750d07d2ce7a6116af3e06acaf6866725e60e19498db1b84eed27568d47123706dc412529542fc955125b071cd06b8e859a7edac7484aea38c8c140add53743bc5a7dc181714bb3aa1033192f2d91f420907ba9c3665fb4e76599c2357b636faea3364108086840c5bc35bc7616080ae45e3604ce842b31e6b6506f31de61f51576d&p=8f66d315d9c041ab0cbcc32d02148c&newp=8b2a970dc5881cf608e2977e09009d231610db2151d6d401298ffe0cc4241a1a1a3aecbf26221b00d0c67d6006a8425aeefa3274330434f1f689df08d2ecce7e679b6c6929&user=baidu&fm=sc&query=Background+partial+concurrent+mark+sweep+GC+freed+42612%281838KB%29+AllocSpace+objects%2C+0%280B%29+LOS+objects%2C+17%25&qid=9f5a34b4000035cb&p1=4#baidusnap8) 以下是该网页在北京时间 2016年09月13日 07:31:34 的快照；

|  |
| --- |
|  |
| 详解 Android 虚拟机 ART 运行时库 分析 2015/3/14  编辑：admin 来源：本站整理 |
| ART 将会取代Dalvik虚拟机，因为 在Dalvik下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。  　　在最新的Google I/O大会上，Google 发布了关于Android上最新的运行时库的情况。这就是Android RunTime (ART). ART 将会取代Dalvik虚拟机，成为Android平台上Java代码的执行工具。虽然自从Android KitKat，就有了一些关于ART的消息，但是基本都是一些新闻性质的，缺乏具体技术细节方面的介绍。本文尝试综合目前已有的各种消息，以及最新放出的 Android L 预览版本的ROM的情况，对ART运行时库做个详细的分析。  　　和IOS，Windows，Tizen之类的移动平台直接将软件编译成能够直接运行在特定硬件平台上的本地代码不同。Android平台上的软件会被编译器首先编译成通用的“byte-code”，然后再在具体的移动设备上被转换成本地指令执行。  　　从Android诞生至今的十几年时间里，Dalvik从开始时非常简单的Java Byte-Code执行虚拟机，逐渐增加各种新的特性，满足应用程序对性能的需求，以及与硬件设备协同演进。这其中包括在Android 2.2版本中引入的即时编译器(JIT-Compiler), 以及随后的多线程支持，以及其他一些优化。  　　不过，在近两年里，Android整个生态系统的进步对Android虚拟机的需求，目前的Dalvik虚拟机的开发已经无法满足了。Dalvik 最初设计时，处理器的性能很弱，移动设备的内存空间非常有限，而且都是32位的系统。于是Google开始构建一个新的虚拟机来更好的面对未来的发展趋势。这种虚拟机的性能能够在目前的多核处理器，甚至未来的8核处理上轻松扩展，能够满足对大容量存储的支持，以及大容量内存的支持。 于是乎，ART出现了。  **1 架构介绍**  　　首先，ART的首要设计需求就是完全兼容能在Dalvik上运行的byte-code，即dex(Dalvik executable)。这样的话，对于程序员来说，就不需要对重新编译已有的程序，直接拿APK就可以在Dalvik和ART虚拟机上运行。ART带来的最大的变化，就是使用预编译技术(Ahead-of-Time compile)取代Dalvik中的即时编译技术(Just-In-Time compile)。之前，在应用程序每次执行的时候，虚拟机需要将bytecode编译成本地码执行，而在ART中这种编译操作只需执行一次，随后对该应用程序的执行都可以通过直接执行保存下来的本地码完成。当然，这种预编译技术，需要占用额外的存储空间来存储本地码。正是因为现在移动设备的存储空间越来越大，这种技术才得以应用。  　　这种预编译技术使得很多原来无法执行的编译优化技术在新的Android平台上成为可能。因为代码只被编译和优化一次，因此值得花费更多的时间在这次编译上，以便进行更多的优化。Google表示，现在可以在应用程序的整体代码技术上进行更高层次的优化，因为编译器现在能够看到应用程序的整体代码，而之前的即时编译，编译器只能看到并优化应用程序中某个函数或者非常小的一部分代码。采用ART后，代码中异常检查带来的开销绝大部分可以避免，对方法和接口的调用也加快了很多。完成这部分功能的是新添加的“dex2oat”组件，用来替代Dalvik中对应的“dexopt”组件。Dalvik中的 Odex文件(优化后的dex)文件，在ART中也用ELF文件代替了。  　　因为ART目前编译生成ELF可执行文件，内核就可以直接对载入内存中的代码进行分页管理，这也会带来更加高效的内存管理，以及更少的内存占用。说到这里，我非常好奇内核中的KSM(Kernel same-page merging)在ART中会有什么样的影响，应该能带来不错的效果吧。我们拭目以待。  　　ART对续航时间的影响也是非常显著的。因为不再需要解释执行，JIT也不用在程序运行时工作，这样会直接节省CPU需要执行的指令数，因而耗电降低。  　　因为预编译时引入了更多分析和优化，编译的时间会变长，这是ART可能会带来的一个副作用。因此相比Dalvik虚拟机，当设备首次启动及应用程序第一次安装时，需要花费的时间更久。Google声称，这种时间上的增加并不那么恐怖。他们希望并预期日后ART上完成上述动作的时间会和目前的 Dalvik差不多，甚至更短些。  　　上面的图显示，ART带来的性能提升是非常明显的。对于同样的代码，性能提升约2倍左右。Google称，将Android L最终发布的时候，可以预计的性能提升将会像Chessbench一样，有3x的加速。  **2 垃圾收集：理论和实践**  　　Android虚拟机依赖自动化的内存管理机制，即自动垃圾收集。这一Java语言编程模式的基石也是Android系统自诞生之日起，非常重要的一部分。这里向不太了解垃圾收集概念的朋友解释一下，所谓自动垃圾收集，就是说程序员在编程过程中，不需要自己负责物理内存的存储的分配和释放。只需要使用固定的模式创建你需要的变量或者对象，然后直接利用该变量或对象即可。程序的运行环境会自动在内存中分配相应的内存空间存储该变量或者对象， 并在该变量或者对象失效后，自动释放所分配的内存。这是和其他需要人工进行存储管理的较低层次语言最大的区别。自动垃圾收集的好处是，程序员不必再在编程时担心内存管理的问题，当然，这也是有代价的，那就是程序员无法控制内存何时分配和释放，因而无法在需要时进行优化(Java语言有一些编程接口可以供程序员手工优化程序，但控制方式和粒度有限).  　　Android曾经被Dalvik的垃圾收集机制折腾了很久。Android平台的内存普遍较小，每次应用程序需要分配内存，当堆空间(分配给应用程序的一块内存空间)不能提供如此大小的空间时，Dalvik的垃圾收集器就会启动。垃圾收集器会遍历整个堆空间，查看每一个应用程序分配的对象，并对所有可到达的对象(即还会被使用的对象)标记，并将那些没有标记的对象空间释放掉。  　　在Dalvik虚拟机中，垃圾收集器执行的过程将导致两次应用程序的停顿：  　　一是在遍历堆地址空间阶段，  　　另一个是标记阶段。  　　所谓停顿，即应用程序所有正在执行的进程将暂停。如果停顿时间过长，将会导致应用程序在渲染时出现丢帧现象，进而导致应用程序的卡顿现象，大大降低用户体验。  　　Google声称，在Nexus 5手机上，这种停顿的平均长度在54ms。这个停顿时间将导致平均每次垃圾收集会导致在应用程序渲染显式时丢掉4帧的。  　　我自己的经验和测试表明，根据应用程序的不同，停顿的时间可能会增大很多。比如，在官方的FIFA应用程序这一典型程序中，垃圾收集的停顿会非常厉害。  　　07-01 15:56:14.275: D/dalvikvm(30615): GCFORALLOC **freed** 4442K, 25% free 20183K/26856K, paused 24ms, total 24ms  　　07-01 15:56:16.785: I/dalvikvm-heap(30615): Grow heap (frag case) to 38.179MB for 8294416-byte allocation  　　07-01 15:56:17.225: I/dalvikvm-heap(30615): Grow heap (frag case) to 48.279MB for 736**12**96-byte allocation  　　07-01 15:56:17.625: I/Choreographer(30615): Skipped 35 frames! The application may be doing too much work on its main thread.  　　07-01 15:56:19.035: D/dalvikvm(30615): GCCONCURRENT **freed** 35838K, 43% free 51351K/89052K, paused 3ms+5ms, total 106ms  　　07-01 15:56:19.035: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 96ms  　　07-01 15:56:19.815: D/dalvikvm(30615): GCCONCURRENT **freed** 7078K, 42% free 52464K/89052K, paused 14ms+4ms, total 96ms  　　07-01 15:56:19.815: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 74ms  　　07-01 15:56:20.035: I/Choreographer(30615): Skipped 141 frames! The application may be doing too much work on its main thread.  　　07-01 15:56:20.275: D/dalvikvm(30615): GCFORALLOC **freed** 4774K, 45% free 49801K/89052K, paused 168ms, total 168ms  　　07-01 15:56:20.295: I/dalvikvm-heap(30615): Grow heap (frag case) to 56.900MB for 4665616-byte allocation  　　07-01 15:56:21.315: D/dalvikvm(30615): GCFORALLOC **freed** 1359K, 42% free 55045K/936**12**K, paused 95ms, total 95ms  　　07-01 15:56:21.965: D/dalvikvm(30615): GCCONCURRENT **freed** 6376K, 40% free 56861K/936**12**K, paused 16ms+8ms, total **12**6ms  　　07-01 15:56:21.965: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 111ms  　　07-01 15:56:21.965: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 97ms  　　07-01 15:56:22.085: I/Choreographer(30615): Skipped 38 frames! The application may be doing too much work on its main thread.  　　07-01 15:56:22.195: D/dalvikvm(30615): GCFORALLOC **freed** 1539K, 40% free 56833K/936**12**K, paused 87ms, total 87ms  　　07-01 15:56:22.195: I/dalvikvm-heap(30615): Grow heap (frag case) to 60.588MB for 1331732-byte allocation  　　07-01 15:56:22.475: D/dalvikvm(30615): GCFORALLOC **freed** 308K, 39% free 59497K/96216K, paused 84ms, total 84ms  　　07-01 15:56:22.815: D/dalvikvm(30615): GCFORALLOC **freed** 287K, 38% free 60878K/97516K, paused 95ms, total 95ms  　　上面的log是从FIFA应用程序运行后的几秒钟时间里截取的。垃圾收集器在短短的8秒内被执行了9次，导致应用程序总共卡顿了603ms，丢帧达214次。绝大多数的卡顿都来自内存分配请求，在log中以”**GC**\_FOR\_ALLOC“标签描述。  　　ART将整个垃圾收集系统做了重新设计和实现。为了能做些对比，下面给出使用ART运行相同的应用程序，在相同的场景下提取的log：  　　07-01 16:00:44.531: I/art(198): Explicit **concurrent** **mark** **sweep** **GC** **freed** 700(30KB) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 186us total **12**.763ms  　　07-01 16:00:44.545: I/art(198): Explicit **concurrent** **mark** **sweep** **GC** **freed** 7(240B) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 198us total 9.465ms  　　07-01 16:00:44.554: I/art(198): Explicit **concurrent** **mark** **sweep** **GC** **freed** 5(160B) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 224us total 9.045ms  　　07-01 16:00:44.690: I/art(801): Explicit **concurrent** **mark** **sweep** **GC** **freed** 65595(3MB) AllocSpace objects, 9(4MB) LOS objects, 810% free, 38MB/58MB, paused 1.195ms total 87.219ms  　　07-01 16:00:46.517: I/art(29197): **Background** **partial** **concurrent** **mark** **sweep** **GC** **freed** 74626(3MB) AllocSpace objects, 39(4MB) LOS objects, 1496% free, 25MB/32MB, paused 4.422ms total 1.371747s  　　07-01 16:00:48.534: I/Choreographer(29197): Skipped 30 frames! The application may be doing too much work on its main thread.  　　07-01 16:00:48.566: I/art(29197): **Background** sticky **concurrent** **mark** **sweep** **GC** **freed** 70319(3MB) AllocSpace objects, 59(5MB) LOS objects, 825% free, 49MB/56MB, paused 6.139ms total 52.868ms  　　07-01 16:00:49.282: I/Choreographer(29197): Skipped 33 frames! The application may be doing too much work on its main thread.  　　07-01 16:00:49.652: I/art(**12**87): Heap transition to ProcessStateJankImperceptible took 45.636146ms saved at least 723KB  　　07-01 16:00:49.660: I/art(**12**56): Heap transition to ProcessStateJankImperceptible took 52.650677ms saved at least 966KB  　　ART和Dalvik的差别非常大，新的运行时内存管理仅仅停顿了**12**.364ms，运行了4次前台垃圾收集，以及2次后台垃圾收集。在应用程序执行的过程中，应用程序的堆空间大小并没有增加，而Dalvik虚拟机中堆空间共增加了4次。丢帧的个数方面，ART虚拟机也降到了63帧。  　　上面这段示例，只不过是一个开发并不完善的应用程序中最坏的一个场景。因为即使在ART虚拟机中，这个应用程序还是丢掉了不少帧渲染图像。不过上面的log对比依然很有参考价值，毕竟牛逼的程序员没几个，大多数的Android程序都没办法开发的很完美。Android需要能hold住这种情况。  　　ART将一些通常需要垃圾收集器做的工作，拆分给应用程序本身完成。这样，Dalvik中因为遍历堆空间引入的第一次停顿，就被完全消除了。而第二次停顿也因为一项预清理技术 (packard pre-cleaning)的应用而大大缩短。使用该技术后，只需要在清理完成后，简单的检查和验证时稍微停顿一下即可。Google声称，他们已经设法将这类停顿的时间缩短到3ms左右，相比Dalvik虚拟机的垃圾收集器来说，基本上是一个多数量级的降低，很不错了。  　　ART还引入了一个特殊的超大对象存储空间(large object space，LOS)，这个空间与堆空间是分开的，不过仍然驻留在应用程序内存空间中。这一特殊的设计是为了让ART可以更好的管理较大的对象，比如位图对象(bitmaps)。在对堆空间分段时，这种较大的对象会带来一些问题。比如，在分配一个此类对象时，相比其他普通对象，会导致垃圾收集器启动的次数增加很多。有了这个超大对象存储空间的支持，垃圾收集器因堆空间分段而引发调用次数将会大大降低，这样垃圾收集器就能做更加合理的内存分配，从而降低运行时开销。  　　一个很好的例子，就是运行Hangouts(环聊)应用程序时，在Dalvik虚拟机中，我们能看到数次因为分配内存，运行**GC**而导致的停顿。  　　07-01 06:37:13.481: D/dalvikvm(7403): GCEXPLICIT **freed** 2315K, 46% free 18483K/34016K, paused 3ms+4ms, total 40ms  　　07-01 06:37:13.901: D/dalvikvm(9871): GCCONCURRENT **freed** 3779K, 22% free 21193K/26856K, paused 3ms+3ms, total 36ms  　　07-01 06:37:14.041: D/dalvikvm(9871): GCFORALLOC **freed** 368K, 21% free 21451K/26856K, paused 25ms, total 25ms  　　07-01 06:37:14.041: I/dalvikvm-heap(9871): Grow heap (frag case) to 24.907MB for 147472-byte allocation  　　07-01 06:37:14.071: D/dalvikvm(9871): GCFORALLOC **freed** 4K, 20% free 22167K/27596K, paused 25ms, total 25ms  　　07-01 06:37:14.111: D/dalvikvm(9871): GCFORALLOC **freed** 9K, 19% free 23892K/29372K, paused 27ms, total 28ms  　　我们从所有的垃圾收集log中截取了上述一段。其中的显式(**GC**\_EXPLICIT)和并发(**GC**\_**CONCURRENT**)是垃圾收集器中比较通用的清理和维护性调用。**GC**\_FOR\_ALLOC则是在内存分配器尝试分配新的内存空间，但堆空间不够用时，调用的。上面的log中，我们能看到堆空间因为分段操作而扩充了堆空间，但仍然无法装下大对象。在整个大对象分配的过程中，停顿时间长达90ms。  　　相比之下，下面这段log是从Android L预览版本的ART运行log中提取的。  　　07-01 06:35:19.718: I/art(10844): Heap transition to ProcessStateJankPerceptible took 17.989063ms saved at least -138KB  　　07-01 06:35:24.171: I/art(**12**56): Heap transition to ProcessStateJankImperceptible took 42.936250ms saved at least 258KB  　　07-01 06:35:24.806: I/art(801): Explicit **concurrent** **mark** **sweep** **GC** **freed** 85790(3MB) AllocSpace objects, 4(10MB) LOS objects, 850% free, 35MB/56MB, paused 961us total 83.110ms  　　我们目前还不知道log中的”Heap Transition”表达的什么意思，不过可以猜测应该是堆空间大小重设机制。在应用程序已经运行之后，唯一的对垃圾收集器的调用仅消耗的961us。我们并没有在这段截取的log之前，发现任何对垃圾收集器的调用操作。这段log中比较有趣的，就是LOS的统计。能够看到，在LOS中有4个较大的对象，共10MB。这块内存并没有分配在堆空间内，否则应该会有类似Dalvik的提示。  　　ART的内存分配系统本身也被重写了。虽然ART相比Dalvik，在内存分配方面，能够带来大约25%的性能提升，不过Google显然对此不满意，因此引入了一个新的内存分配器来取代当前使用的“malloc”分配器。  　　这个新的内存分配器，“rosalloc”(Runs-of-Slots-Allocator)是依据多线程Java应用程序的特点而设计的。此内存分配器有更细粒度的锁机制，可以直接对独立的对象上锁，而非对整个待分配的内存空间上锁。在线程局部区域中的小对象的分配，完全可以无视锁的存在了。没有了锁的请求和释放，线程局部小对象的访问速度也就大幅提升了。  　　这个新的内存分配器大幅提升了内存分配的速度，加速比达到了10x。  　　同时，ART的垃圾回收算法也做了改进，提升了用户使用体验，避免应用程序的卡顿。这些算法在Google内部目前仍然正在开发中。近期，Google仅仅介绍了一个新算法，“Moving Garbage Collector”.核心思想是，当应用程序运行在后台时，将程序的堆空间做段合并操作。  **3 64位支持**  　　ART在设计时充分考虑了将日后可能运行的各种平台进行模块化。因此，ART提供了大量的编译器后端，用于生成目前常见的体系结构的代码，例如ARM，X86和MIPS，其中包括对ARM64， X86-64的支持，以及尚未实现的对MIPS64的支持。  　　对于ARM的64位系统带来的好处，相比很多朋友都了解了。更大的内存地址空间，普适的性能提升，以及加解密的能力和性能提升，此外还有对已有32位应用程序的兼容。  　　除此之外，Google还在ART中引入了引用压缩技术，来避免ART堆空间内部因为64位指针的引入导致的内存占用变大问题。其实，就是在执行时，所有的指针都采用32位表示，而非64位系统应该采用的64位指针。  　　Google公开了一些ARM和X86平台上应用程序在64位和32位模式下的性能对比。这只是一些预览性质数据。X86的性能测试在Intel的 BayTrail系统上进行，对于不同的RenderScript测试程序，性能提升从2x到4.5x不等。ARM平台方面，分别在A57和A53系统上，对crypto的性能做了对比。这些数据因为都是针对非常小的例子，所以代表性不大，因此还无法代表实际应用场景的情况。  　　不过，Google也放出了一些有趣的数据，这些数据是在他们内部使用的系统Panorama上测试的。通过简单的从32位ABI转换为64位 ABI，能够获得13%到19%的性能提升。还有个喜人的结论，那就是ARM的Cortex A53在AArch64模式下能获得性能提升比A57核要多。  　　Google还声称，目前应用商店中85%的应用程序都可以直接在64位模式下运行，也就是说仅有15%的应用程序在某种程度上使用了本地代码，需要重新为64位平台编译该应用程序。这对Google来说将是一个非常大的优势。明年，当大多数芯片厂商都开始推64位片上系统的时候，从32位 Android系统到64位Android系统的的切换将会非常快。  **4 结论**  　　结合上面介绍的诸多方面，ART是Google发布的一款性能提升大杀器，并且ART也解决了多个数年来困扰Android系统的诸多问题。ART 有效地改进了多个解释执行应用程序面临的问题，也提供了一个自动化的高效的存储管理系统。对于开发者来说，许多过去需要手工添加代码解决的性能问题，现在都能被ART轻松hold住了。  　　这也意味着Android系统终于能够在系统平滑度，应用程序性能方面与IOS势均力敌了。对消费者来说，是件喜大普奔的事情。  　　Google目前仍在，而且在未来一段时间内还将大力改进ART。ART目前的状况，与6个月前已经大不相同了，预计等到Android L真正发布的时候，又会有翻天覆地的变化。前途是光明的，让我们拭目以待，翘首期盼吧。 |
|  |