对于目前的开发有什么作用

1.2. 域名防火墙的设计和实现

PackageInstaller，apk的安装过程

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

android应用性能优化最佳实践.note-》ANR分类和原理(kaka)，死锁分析

系统签名（安全问题？），系统应用Laucher遇到的问题，binder通信，setup的webview，互斥逻辑，setup吊起失败问题

## ANR剖析

Application Not Responding, 和发生Crash一样，ANR也是Android应用程序无法继续运行的一种异常，和Crash的区别是，它不一定是由于程序的异常错误导致的，一般是应用处理长时间没有结果响应导致主进程不能处理下一件事情。本节介绍ANR的定义和发生ANR的原因，掌握如何快速定位导致ANR的原因以及如何在开发过程中尽量避免发生ANR。

### ANR介绍

ANR（Application Not Responding）即应用无响应，在Android系统中，应用发生的ANR有以下几种类型：

1.KeyDispatchTimeout

最常见的ANR类型是对输入事件5s内无响应，比如**按键或触摸事件**在此时间内无响应。

2.BroadcastTimeout

BroadcastReceiver在指定时间（原生系统默认是10s）内无法处理完成，并且没有结束执行onReceive。

3.ServiceTimeout

这种类型在Android应用中出现的概率很小，是指Service在特定的时间（原生系统是20s）内无法处理完成。

引起ANR的根本原因总体来说有以下两种：

1）应用程序自身逻辑有缺陷，或者在某些异常场景触发了此缺陷，如主线程堵塞、死循环等导致。

2）由于Android设备**其他进程的CPU**占用高、事件没有得到及时的响应、死锁，导致当前应用进程无法抢占到CPU时间片。

### KeyDispatchTimeout具体的原因

我们需要关注的是第一种情况导致的ANR。**KeyDispatchTimeout具体的原因有**：

(1). 当前的事件没有机会得到处理（即UI线程正在处理前一个事件，没有及时的完成或者looper被某种原因阻塞住了）

(2). 当前的事件正在处理，但没有及时完成

首先要知道事件发生的线程，一般来说大多数可能是ui线程操作超时，那么ui线程都有哪些呢：

(1). Activity 生命周期

(2). View post 的runnable方法 、 handler（MainLooper） 的 handleMessage()

(3). Asycktask 的 onPreExecute(), onPostExecute() , onProgressUpdate()方法

(4). Broadcast 的onReceive()

(5). Service

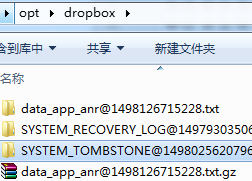
其他原因造成的anr：

(1). 线程死锁

(2). cpu 饥饿

### ANR分析

在Android系统上，如果发生ANR，Logcat会产生对应的日志和一个**/data/anr/traces.txt**文件，只保留最后一次发生ANR时的信息，“/data/system/dropbox则是保存最近3d的anr。”是DB指定的文件存放位置.分析ANR原因主要是分析这两个信息。但是Logcat不一定保证在发生ANR时可以拿到，特别是在非开发人员手上发生ANR情况时，再去拿logcat已经晚了，并且ANR的原因非常复杂。



#### Traces系统源码

SystemServer在启动时， 会创建并添加DROPBOX\_SERVICE.

|  |
| --- |
| //SystemServer.java  private void startOtherServices() {  try {  Slog.i(TAG, "DropBox Service");  ServiceManager.addService(Context.DROPBOX\_SERVICE,  new DropBoxManagerService(context, new File("/data/system/dropbox")));  } catch (Throwable e) {  reportWtf("starting DropBoxManagerService", e);  }  } |

framework是如何输出ANR信息到traces.txt文件中去的？

ActivityManagerService.java中的appNotResponding()方法

|  |
| --- |
| final void appNotResponding(ProcessRecord app, ActivityRecord activity,  ActivityRecord parent, boolean aboveSystem, final String annotation) {  //firstPids和lastPids两个集合存放那些将会在traces中输出信息的进程的进程号  ArrayList<Integer> firstPids = new ArrayList<Integer>(5);  SparseArray<Boolean> lastPids = new SparseArray<Boolean>(20);  //mController是IActivityController接口的实例，是为Monkey测试程序预留的，默认为null  if (mController != null) {  try {  // 0 == continue, -1 = kill process immediately  int res = mController.appEarlyNotResponding(app.processName, app.pid, annotation);  if (res < 0 && app.pid != MY\_PID) {  app.kill("anr", true);  }  } catch (RemoteException e) {  mController = null;  Watchdog.getInstance().setActivityController(null);  }  }  long anrTime = SystemClock.uptimeMillis();  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow(); //更新CPU使用率  }  synchronized (this) {  //一些特定条件下会忽略ANR  // PowerManager.reboot() can block for a long time, so ignore ANRs while shutting down.  if (mShuttingDown) {  Slog.i(TAG, "During shutdown skipping ANR: " + app + " " + annotation);  return;  } else if (app.notResponding) {  Slog.i(TAG, "Skipping duplicate ANR: " + app + " " + annotation);  return;  } else if (app.crashing) {  Slog.i(TAG, "Crashing app skipping ANR: " + app + " " + annotation);  return;  }  // In case we come through here for the same app before completing  // this one, mark as anring now so we will bail out.  app.notResponding = true;  // Log the ANR to the event log.  EventLog.writeEvent(EventLogTags.AM\_ANR, app.userId, app.pid,  app.processName, app.info.flags, annotation);  //当前发生ANR的应用进程被第一个添加进firstPids集合中.  //**所以会第一个向traces文件中写入信息**。反过来说，traces文件中出现的第一个进程正常情况下就是发生ANR的那个进程。  // Dump thread traces as quickly as we can, starting with "interesting" processes.  firstPids.add(app.pid);  int parentPid = app.pid;  if (parent != null && parent.app != null && parent.app.pid > 0) parentPid = parent.app.pid;  if (parentPid != app.pid) firstPids.add(parentPid);  if (MY\_PID != app.pid && MY\_PID != parentPid) firstPids.add(MY\_PID);  for (int i = mLruProcesses.size() - 1; i >= 0; i--) {  ProcessRecord r = mLruProcesses.get(i);  if (r != null && r.thread != null) {  int pid = r.pid;  if (pid > 0 && pid != app.pid && pid != parentPid && pid != MY\_PID) {  if (r.persistent) {  firstPids.add(pid);  } else {  lastPids.put(pid, Boolean.TRUE);  }  }  }  }  }  // Log the ANR to the main log.  StringBuilder info = new StringBuilder();  info.setLength(0);  info.append("ANR in ").append(app.processName);  if (activity != null && activity.shortComponentName != null) {  info.append(" (").append(activity.shortComponentName).append(")");  }  info.append("\n");  info.append("PID: ").append(app.pid).append("\n");  if (annotation != null) {  info.append("Reason: ").append(annotation).append("\n");  }  if (parent != null && parent != activity) {  info.append("Parent: ").append(parent.shortComponentName).append("\n");  }  final ProcessCpuTracker processCpuTracker = new ProcessCpuTracker(true);  //dumpStackTraces()是输出ANR信息到traces文件的函数  File tracesFile = dumpStackTraces(true, firstPids, processCpuTracker, lastPids,  NATIVE\_STACKS\_OF\_INTEREST);  String cpuInfo = null;  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow();  synchronized (mProcessCpuTracker) {  cpuInfo = mProcessCpuTracker.printCurrentState(anrTime);  }  //输出ANR发生前一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentLoad());  info.append(cpuInfo);  }  //输出ANR发生后一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentState(anrTime));  //此处， 把ANR的信息输出到main log中.  Slog.e(TAG, info.toString());  if (tracesFile == null) {  // There is no trace file, so dump (only) the alleged culprit's threads to the log  Process.sendSignal(app.pid, Process.SIGNAL\_QUIT);  }  //将ANR信息同时输出到DropBox中  addErrorToDropBox("anr", app, app.processName, activity, parent, annotation,  cpuInfo, tracesFile, null);  //在>=Android4.0中可以设置是否不显示ANR提示对话框，如果设置的话就不会显示对话框，并且会杀掉ANR进程  // Unless configured otherwise, swallow ANRs in background processes & kill the process.  boolean showBackground = **Settings.Secure.getInt(mContext.getContentResolver(),**  **Settings.Secure.ANR\_SHOW\_BACKGROUND, 0) != 0;**  synchronized (this) {  mBatteryStatsService.noteProcessAnr(app.processName, app.uid);  if (!showBackground && !app.isInterestingToUserLocked() && app.pid != MY\_PID) {  app.kill("bg anr", true);  return;  }  // Set the app's notResponding state, and look up the errorReportReceiver  makeAppNotRespondingLocked(app,  activity != null ? activity.shortComponentName : null,  annotation != null ? "ANR " + annotation : "ANR",  info.toString());  // 显示ANR提示对话框  // Bring up the infamous App Not Responding dialog  Message msg = Message.obtain();  HashMap<String, Object> map = new HashMap<String, Object>();  msg.what = SHOW\_NOT\_RESPONDING\_MSG;  msg.obj = map;  msg.arg1 = aboveSystem ? 1 : 0;  map.put("app", app);  if (activity != null) {  map.put("activity", activity);  }  mUiHandler.sendMessage(msg);  }  }  public static File dumpStackTraces(boolean clearTraces, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  //系统属性“dalvik.vm.stack-trace-file”用来配置trace信息输出文件  //之所以trace信息会输出到“/data/anr/traces.txt”文件中，就是系统属性“dalvik.vm.stack-trace-file”设置的  //adb shell下， 可以使用setprop和getprop对系统属性进行设置和读取  //getprop dalvik.vm.stack-trace-file  //setprop dalvik.vm.stack-trace-file /tmp/stack-traces.txt  String tracesPath = SystemProperties.get("dalvik.vm.stack-trace-file", null);  if (tracesPath == null || tracesPath.length() == 0) {  return null;  }  File tracesFile = new File(tracesPath);  try {  File tracesDir = tracesFile.getParentFile();  if (!tracesDir.exists()) {  tracesDir.mkdirs();  if (!SELinux.restorecon(tracesDir)) {  return null;  }  }  //FileUtils.setPermissions是个很有用的函数，设置文件属性时经常会用到  FileUtils.setPermissions(tracesDir.getPath(), 0775, -1, -1); // drwxrwxr-x  //clearTraces为true，会删除旧文件，创建新文件  if (clearTraces && tracesFile.exists()) tracesFile.delete();  tracesFile.createNewFile();  FileUtils.setPermissions(tracesFile.getPath(), 0666, -1, -1); // -rw-rw-rw-  } catch (IOException e) {  Slog.w(TAG, "Unable to prepare ANR traces file: " + tracesPath, e);  return null;  }  dumpStackTraces(tracesPath, firstPids, processCpuTracker, lastPids, nativeProcs);  return tracesFile;  }  private static void dumpStackTraces(String tracesPath, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  // Use a FileObserver to detect when traces finish writing.  // The order of traces is considered important to maintain for legibility.  //使用FileObserver监听SignalCatcher线程是否已经完成写入traces文件的操作  FileObserver observer = new FileObserver(tracesPath, FileObserver.CLOSE\_WRITE) {  @Override  public synchronized void onEvent(int event, String path) {  notify(); //让其他阻塞在observer.wait()这行代码的线程继续往下执行.  }  };  try {  observer.startWatching();  //首先输出firstPids集合中指定的进程，这些也是对ANR问题来说最重要的进程  // First collect all of the stacks of the most important pids.  if (firstPids != null) {  try {  int num = firstPids.size();  for (int i = 0; i < num; i++) {  synchronized (observer) {  //前面提到的SIGNAL\_QUIT, 由SignalCatcher线程处理Process.SIGNAL\_QUIT信号， 并把进程中的各个线程输出到traces.txt文件中.  //具体代码可以查看/dalvik/vm/目录下的SignalCatcher.cpp::logThreadStacks函数和Thread.cpp:: dvmDumpAllThreadsEx函数。  Process.sendSignal(firstPids.get(i), Process.SIGNAL\_QUIT);  observer.wait(200); // Wait for write-close, give up after 200msec  //block here, 直到另一个线程调用了observer.notify() or observer.notifyAll()方法， 再或是等待了200毫秒超时后， 才会继续往下执行代码.  }  }  } catch (InterruptedException e) {  Slog.wtf(TAG, e);  }  }  } finally {  observer.stopWatching();  }  }  Object.java  final void notify()  //Causes a thread which is waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void notifyAll()  //Causes all threads which are waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void wait()  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method of this object.  final void wait(long millis)  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method  //of this object or until the specified timeout expires. |

getprop dalvik.vm.stack-trace-file

#### 分析工具Analyze Stacktrace

在Android Studio上提供了一个分析trace文件的工具：**Analyze Stacktrace**。Analyze Stacktrace可以更直观地分析导致ANR的原因。

Analyze Stacktrace使用方法如下：

1）在Android Studio的工具栏中，选择Analyze→Analyze Stacktrace，打开Analyze Stacktrace工具窗口。

2）将traces.txt中的内容复制到窗口，单击Normalize按钮，生成Thread Dump列表，如图5-2所示，左边为所有线程列表，右边为选中线程的具体信息。注意有可能不显示没有上锁的线程。值得注意的是， 当前运行的其他进程也会把当前的函数堆栈信息输出到traces.txt文件中, 但发生ANR的进程正常情况下会第一个输出，所以一般情况下我们只看traces.txt的开头就行了。

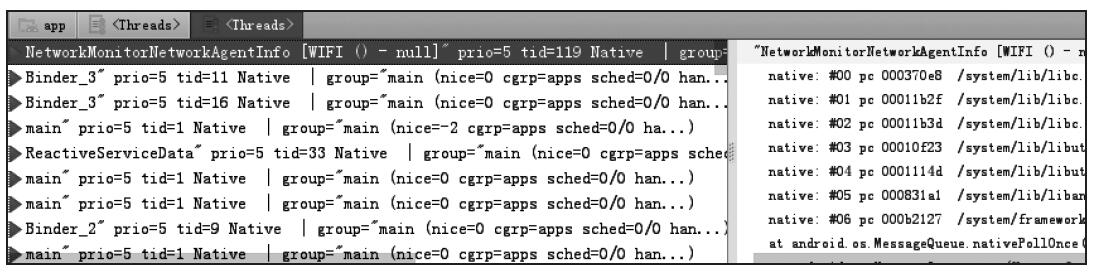


图5-2　Thread Dump列表

3）如果某个线程被标红，说明此线程被堵塞了，然后在右边的详细信息中查看堵塞的具体原因。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　部分ANR是由于多个线程之间的同步锁导致，在使用Analyze Stacktrace分析时，可以在详情页查看是在等待哪个线程，再定位到对应的线程。

从以上的分析能够知道， 发生ANR时Android为我们提供了两种“利器”：traces文件和CPU使用率。**线程有很多状态，了解这些状态的意义对分析ANR的原因是有帮助的.**



开发中定位ANR问题日志有个很简单的规律，就是直接找到我们自己开发App所使用的包名（包括第三方Library库）信息开始定位找就可以了

[浅析android 线程状态](http://www.jianshu.com/p/3fb50ddc36d0)，java的6种线程状态定义在/java/lang/Thread.java中:在VMThread.java中, 可以看到下面的代码， native thread有10种状态, 对应着java thread的6种状态.

//

static final Thread.State[] STATE\_MAP = new Thread.State[] {

Thread.State.TERMINATED, // ZOMBIE

Thread.State.RUNNABLE, // RUNNING

Thread.State.TIMED\_WAITING, // TIMED\_WAIT

Thread.State.BLOCKED, // MONITOR

Thread.State.WAITING, // WAIT

Thread.State.NEW, // INITIALIZING

Thread.State.NEW, // STARTING

**Thread.State.RUNNABLE, // NATIVE**

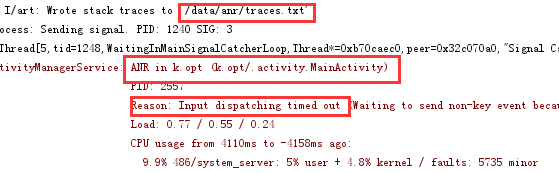
Thread.State.WAITING, // VMWAIT

**Thread.State.RUNNABLE // SUSPENDED**

};

#### KeyDispatchTimeout分析

这里简单构造一个ANR，在HomePageActivity中的单击处理函数中等待100s（Thread，sleep），然后在屏幕上滑动，投递触屏事件，结果出现ANR。发生ANR后，首先看Logcat的信息，**Logcat**信息如下：

从以上Logcat信息中，关注标粗的几个值的信息，它们的意义如下：

·ANR IN：发生ANR的具体类。

·PID：发生ANR的进程，系统在此时会生成trace文件，当前的时间点也是发生ANR的具体时间，以及生成trace文件的时间。

·Reason：当前ARN的类型以及导致ANR的原因。

·CPU usage：CPU的使用情况，在日志中CPU usage有两个时间点，第一个是发生ANR前的CPU使用情况，第二个是发生ANR后的CPU使用情况。

此时用户如果在屏幕上滑动，会产生InputDispatcher事件得不到处理

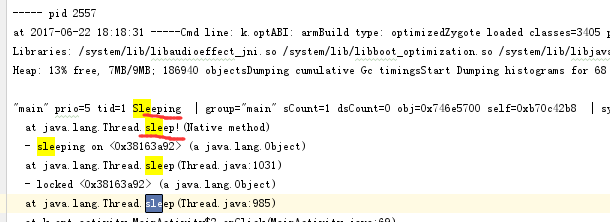
从Logcat中除了能看出在哪个类发生ANR以及ANR的类型，具体的原因主要还是看CPU的使用情况，如果**CPU使用量很少，说明主线程可能阻塞**，如果IOwait很高，说明ANR有可能是由于主线程进行耗时的I/O操作造成的。

接下来分析这个文件**/data/anr/traces.txt**。

可以直接使用adb工具获取traces.txt文件：adb pull/data/anr/traces.txt.

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　trace文件在非root手机上也可以读取，虽然在非root手机上没有权限查看data目录下的内容，**但anr目录可以读取**。

用工具使用，读取出来的traces.txt内容如下：



因此当前这个例子故意写了一个ANR，所以很容易定位到就是在onClick事件中超时导致，在这个日志中需要关注的是线程的状态。

#### [死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)分析

参考：[通过Android trace文件分析死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)

|  |
| --- |
| 对应的部分trace文件内容如下：  "PowerManagerService" prio=5 tid=24 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41dd0eb0 self=0x5241b218   | sysTid=567 nice=0 sched=0/0 cgrp=apps handle=1380038664   | state=S schedstat=( 6682116007 11324451214 33313 ) utm=450 stm=219 core=1   at com.[android](http://lib.csdn.net/base/android).server.am.ActivityManagerService.broadcastIntent(ActivityManagerService.[Java](http://lib.csdn.net/base/java):~13045)   - waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)   at android.app.ContextImpl.sendBroadcast(ContextImpl.java:1144)   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.unblankAllDisplays(PowerManagerService.java:3442)   at com.android.server.power.DisplayPowerState$PhotonicModulator$1.run(DisplayPowerState.java:456)   at android.os.Handler.handleCallback(Handler.java:800)   at android.os.Handler.dispatchMessage(Handler.java:100)   at android.os.Looper.loop(Looper.java:194)   at android.os.HandlerThread.run(HandlerThread.java:60)      "Binder\_B" prio=5 tid=85 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x42744770 self=0x58329e88   | sysTid=3700 nice=-20 sched=0/0 cgrp=apps handle=1471424616   | state=S schedstat=( 1663727513 2044643318 6806 ) utm=132 stm=34 core=1   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.toString(PowerManagerService.java:~3449)   - waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)   at java.lang.StringBuilder.append(StringBuilder.java:202)   at com.android.server.power.PowerManagerService.dump(PowerManagerService.java:3052)   at android.os.Binder.dump(Binder.java:264)   at android.os.Binder.onTransact(Binder.java:236)   at android.os.IPowerManager$Stub.onTransact(IPowerManager.java:373)   at android.os.Binder.execTransact(Binder.java:351)   at dalvik.system.NativeStart.run(Native Method)    "android.server.ServerThread" prio=5 tid=12 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41a76178 self=0x507837a8   | sysTid=545 nice=-2 sched=0/0 cgrp=apps handle=1349936616   | state=S schedstat=( 15368096286 21707846934 69485 ) utm=1226 stm=310 core=0   at com.android.server.power.PowerManagerService.isScreenOnInternal(PowerManagerService.java:~2529)   - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)   at com.android.server.power.PowerManagerService.isScreenOn(PowerManagerService.java:2522)   at com.android.server.wm.WindowManagerService.sendScreenStatusToClientsLocked(WindowManagerService.java:7749)   at com.android.server.wm.WindowManagerService.setEventDispatching(WindowManagerService.java:7628)   at com.android.server.am.ActivityManagerService.updateEventDispatchingLocked(ActivityManagerService.java:8083)   at com.android.server.am.ActivityManagerService.wakingUp(ActivityManagerService.java:8077)   at com.android.server.power.Notifier.sendWakeUpBroadcast(Notifier.java:474)   at com.android.server.power.Notifier.sendNextBroadcast(Notifier.java:455)   at com.android.server.power.Notifier.access$700(Notifier.java:62)   at com.android.server.power.Notifier$NotifierHandler.handleMessage(Notifier.java:600)   at android.os.Handler.dispatchMessage(Handler.java:107)   at android.os.Looper.loop(Looper.java:194)   at com.android.server.ServerThread.run(SystemServer.java:1328) |

从trace文件看，是因为TID为24的线程等待一个TID为12的线程持有的锁，TID为12的线程等待一个TID为85的线程持有的锁，而TID为85的线程确等待一个TID为24的线程持有的锁，导致了循环等待的现象，对应的trace文件的语句如下：

TID 24：- waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)

TID 12： - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)

TID 85：- waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)

    既然是死锁，那么先看各线程都有那些锁。

先看TID=24的线程的栈顶，ActivityManagerService的broadcastIntent函数代码如下:

public final int broadcastIntent(IApplicationThread caller,

Intent intent, String resolvedType, IIntentReceiver resultTo,

int resultCode, String resultData, Bundle map,

String requiredPermission, boolean serialized, boolean sticky, int userId) {

enforceNotIsolatedCaller("broadcastIntent");

synchronized(this) {

intent = verifyBroadcastLocked(intent);

final ProcessRecord callerApp = getRecordForAppLocked(caller);

final int callingPid = Binder.getCallingPid();

final int callingUid = Binder.getCallingUid();

final long origId = Binder.clearCallingIdentity();

int res = broadcastIntentLocked(callerApp,

callerApp != null ? callerApp.info.packageName : null,

intent, resolvedType, resultTo,

resultCode, resultData, map, requiredPermission, serialized, sticky,

callingPid, callingUid, userId);

Binder.restoreCallingIdentity(origId);

return res;

}

可以看到TID=24需要ActivityManagerService这个锁。再看TID=12线程的栈顶，PowerManagerService的isScreenOnInternal函数代码如下：

private boolean isScreenOnInternal() {

synchronized (mLock) {

return !mSystemReady

|| mDisplayPowerRequest.screenState != DisplayPowerRequest.SCREEN\_STATE\_OFF;

}

}

可以看到需要PowerManagerService的mlock这个锁。最后看TID=85线程的栈顶，同样在PowerManagerService里面，内部类DisplayBlankerImpl的toString函数：

public String toString() {

synchronized (this) {

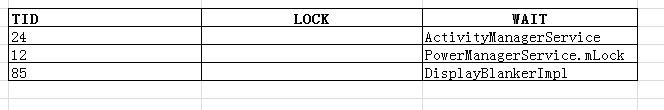
return "blanked=" + mBlanked;

}

}

这是在内部类DisplayBlankerImpl里面实现的，所以需要DisplayBlankerImpl这个锁。

对应的表格如下：



各线程等待的锁情况

死锁的原因是持有锁，又等待锁，下面我们要填充LOCK列，死锁原因就一目了然了。

跟踪TID=24线程的堆栈，在PowerManagerService内部类**DisplayBlankerImp**l的unblankAllDisplays函数中持有锁，持有了this就是持有了**DisplayBlankerImpl**

public void unblankAllDisplays() {

**synchronized (this)** {

mDisplayManagerService.unblankAllDisplaysFromPowerManager()}

}

跟踪TID=12线程的堆栈，在ActivityManagerService的wake\_up函数中持有锁，持有了this就是持有了**ActivityManagerService**

public void wakingUp() {

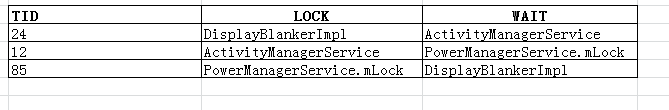
synchronized(this) { } }

跟踪TID=85线程的堆栈，在PowerManagerService的dump有持有锁**mLock**的操作

protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {

synchronized (mLock) {}

因此得到以下的变量lock和wait图，可以清晰的看出死锁了，资源循环等待了。



表二 各线程锁的情况

分析方法：先填充wait列的同步变量，然后在每个线程执行的堆栈的方法中，找出这些同步变量，添加到lock里面就好！

[非常经典的一个案例：Android ANR问题原因分析](http://blog.csdn.net/sinat_22657459/article/details/52780759)

[ANR问题的分析流程](http://blog.csdn.net/jiangguohu1/article/details/52636470)

#### App多线程超时案例

|  |
| --- |
| //多线程同步，造成超时  private void testAnr() {  new Thread(new Runnable() {  @Override  public void run() {  initInChildThread();//2.  }  }).start();//1.  KLogUtil.*D*(TAG,"1");  SystemClock.*sleep*(10);//确保按照1.2.3执行  initView();//3.  }  private synchronized void initView() {  KLogUtil.*D*(TAG,"3");   }  private synchronized void initInChildThread() {  KLogUtil.*D*(TAG,"2");  SystemClock.*sleep*(30\*1000);  } |

### 如何避免

* 避免在主线程进行复杂耗时的操作，特别是文件读取或者数据库操作；
* 避免频繁实时更新UI；

long currTime = SystemClock.uptimeMillis();

if (currTime - lastUpdateTime >= MIN\_RATE&&!progressModel.isDone()) {

lastUpdateTime = currTime;

* BroadCastReceiver 要进行复杂操作的的时候，可以在onReceive()方法中启动一个Service来处理；
* 避免在IntentReceiver里启动一个Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。如果你的应用程序在响应Intent广 播时需要向用户展示什么，你应该使用Notification Manager来实现。
* 在设计及代码编写阶段避免出现出现同步/死锁或者错误处理不恰当等情况。

### ANR监控

发生ANR的体验非常糟糕，因此有必要掌握外网发生ANR的情况。ANR的监控原理和卡顿监控的原理类似，我们知道，卡顿监控可以通过一个子线程向主线程发消息，通过时间差来判断是否发生卡顿，如图5-3所示。卡顿监控详见2.8.2节。

从图5-3中可以看到，是否卡顿取决于T2-T1的时间，发生ANR后，只有T1的时间，没有T2的时间，发生消息后，由于主线程卡住导致消息一直没有执行，这时就不会再回到打印T2的时间戳了，因此，可以使用这个原理监控和上报ANR。

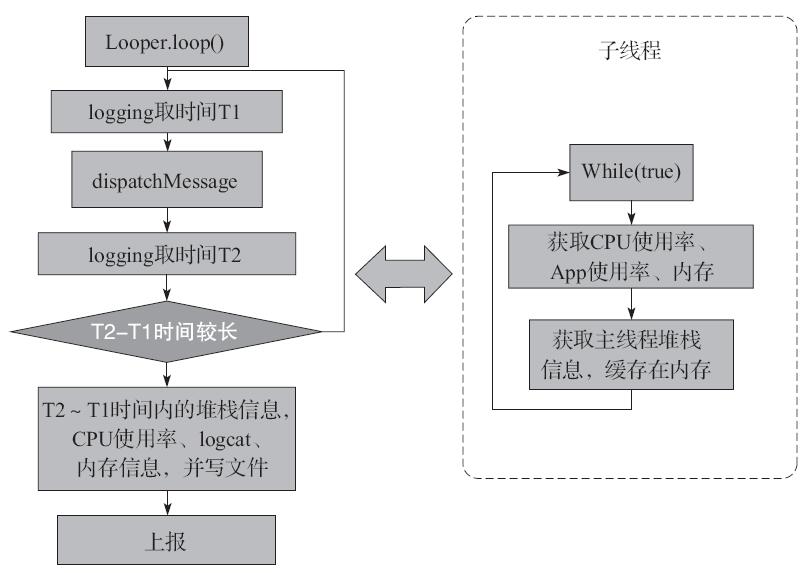
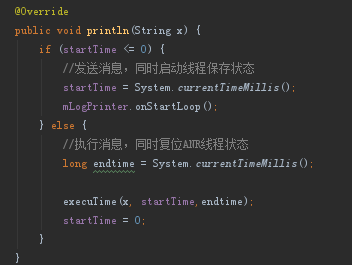


图5-3　卡顿监控原理

在代码清单2-10的基础上，为卡顿监控代码增加ANR的线程监控，在发送消息时，在ANR线程中保存一个状态，主线程消息执行完后再Reset标志位。如果在ANR线程中收到发送消息后，超过一定时间（也就是超时时间），没有复位，就可以认为发生了ANR，这时可以上报相关trace数据给开发人员分析，代码如下：



这个方案有个缺点就是它是通过在主线程监控执行时间的，但不同手机系统的ANR时间不一致，阈值的设定不好掌握，在这个阈值内，主线程还没有处理完，则认为发生了ANR；或者主线程已经发生ANR了，但还要等线程执行完后才能记录该ANR。会有一定概率的误报。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　和卡顿监控一样，数据的上报和Dump信息最好不要在全量版本中打开，否则会影响性能。

# Android系统应用

## 什么是Android系统应用

Android系统应用具有如下几个特点，这些特点都是普通的Android应用不具备的。

* 嵌入到Android ROM中，通过普通的方法无法卸载这些程序。
* 拥有更高的权限。例如，可以实现静默安装，不会出现权限提示对话框，由于Google Play是Android系统应用，所以即使在没有root权限的情况下，Google Play也可以读写/data/app目录1，所以只需要将APK文件复制到这个目录即可安装程序，根本就不需要通过PackageInstaller2进行安装。
* 可以调用Android SDK的私有API，这些API在ADT开发环境下无法调用。

注：安装apk的本质就是拷贝文件到**/data/app或者其子目录**，1）在root的条件下C:\Users\key.guan>adb push A.apk /data/app 然后reboot就完成了A.apk的安装了；2）adb install –r A.apk

## 为什么要研究Android系统应用

有很多直接与Android应用交互的接口都是在这些系统应用中实现的。了解系统应用，意味着对Android系统最上层的API有一个非常透彻的了解。所有的Android系统应用的源代码都在packages目录。其中packages/app是最核心的目录，所有内嵌的APK程序都在该目录中。app目录告诉我们一切Android系统的窗口（Activity）。研究Android系统应用的源代码好处多多：

* 了解Android系统中有哪些窗口、Content Provider、Service和Broadcast可以与之交互。
* 充分掌握很多高级应用的使用方法，例如，OTA升级是如何实现的。
* 对实现原理比较感兴趣的读者可以通过这些源代码了解像Launcher2、短信管理等应用的内容构造。
* 对于想编写可以完成系统级操作的应用的读者，可以学会如何将Android应用嵌入到ROM，升级为Android系统应用，进而可以做任何自己想做的事。
* 对于那些有极客情结的读者，完全定制自己的ROM是最令人振奋的梦想，而Android系统应用将是实现这一梦想最重要的基石。

## 如何编写Android系统应用

Android系统应用和普通的Android应用基本相同，都是主要用Java语言编写的APK程序。不过前者与后者有如下两点不同。

* 签名不同。Android系统应用使用的是系统签名，或称为平台（Platform）签名，而普通的Android应用使用的是一般的签名。
* 可以访问的API不同。在Android SDK中有很多API（类、接口、方法等）都声明为hide1。这类API不允许在普通的Android应用中访问，而只有经过Platform签名的Android系统应用才能使用这些API。

Android源代码中包含了4个签名，前面使用的platform就是其中之一。这些签名文件中在如下的目录中。

<Android源代码根目录>/build/target/product/security，含义。

* testkey：普通的Key，默认情况下使用。如果Android.mk文件不设置LOCAL\_CERTIFICATE 变量，就使用该签名。
* **platform** ：使用该签名的系统应用可以**执行平台的核心功能**。
* shared ：使用该签名的系统应用可以和home/contacts进程共享数据。
* media ：使用该签名的系统应用将成为media/download 系统的一部分。

LOCAL\_CERTIFICATE 变量可以指定上述4个值，例如，如果值为platform，会使用security目录的platform.pk8文件对APK文件进行签名。如果值为media，会使用media.pk8文件对APK文件进行签名。

我们开发的apk需要用到系统权限，需要在AndroidManifest.xml中添加共享系统进程属性：

android:sharedUserId="android.uid.system"

android:sharedUserId="android.uid.shared"

android:sharedUserId="android.media

apk的签名就需要是系统签名(platform、shared或media)才能正常使用。否则报INSTALL\_FAILED\_SHARED\_USER\_INCOMPATIBLE错，如何编译和运行Android系统应用呢？

### 源码命令行签名（Android.mk）

这种方式比较麻烦，你需要有编译过的源码环境，并按如下步骤：

1、拷贝App源码到Android源码的packages/apps/目录或者子目录下，且App源码是普通(Eclipse)格式的

2、配置Android.mk，在其中添加LOCAL\_CERTIFICATE := platform 或 shared 或 media

3、使用mm编译App，生成的apk即系统签名

Android.mk是Android源代码专用的编译文件，相当于GCC的Makefile文件，内容只需要在apps目录中找一个Android系统应用，如PackageInstaller，将该程序中Android.mk文件复制一份，然后进行适当修改即可。

下面先看一下FirstSystemApp程序的Android.mk文件的内容。

src/ch06/FirstSystemApp/Android.mk

LOCAL\_PATH:= $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES :=$(call all-java-files-under, src)

LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v4

#　将编译生成FirstSystemApp.apk文件

**LOCAL\_PACKAGE\_NAME := FirstSystemApp**

**LOCAL\_CERTIFICATE := platform**

include $(BUILD\_PACKAGE)

Android.mk文件的大多数内容都是标准的写法，只有下面两行代码需要了解一下。

LOCAL\_PACKAGE\_NAME := FirstSystemApp

LOCAL\_CERTIFICATE := platform

其中第一行需要根据不同的Android系统应用进行修改。系统会根据LOCAL\_PACKAGE\_NAME变量的值生成APK文件，例如，本例该变量的值是FirstSystemApp，所以会在out/target/product/ maguro/system/app目录生成FirstSystemApp.apk文件， adb push命令上传到/system/app目录即可完成安装。

LOCAL\_CERTIFICATE变量表示签名类型，系统应用通常设为platform，表示Platform签名。

### Java命令行签名

这种方式签名简单，App可以在Eclipse或Android Studio下编译，然后给apk重新签名即可。但调试比较痛苦，即使写成脚本，也需要重复一样的操作，相关文件：

../build/target/product/security/platform.x509.pem

../build/target/product/security/platform.pk8

../out/host/linux-x86/framework/signapk.jar （源码路径: ../build/tools/signapk）

步骤

1、将相关文件及源apk文件置于同一路径下  
2、检查源apk包，去掉META-INF/CERT.SF 和 META-INF/CERT.RSA 文件（可以忽略这一步）  
3、执行签名命令java -jar signapk.jar platform.x509.pem platform.pk8 old.apk new.apk即可

### keytool-importkeypair集成系统签名

让Android Studio集成系统签名，需要用到一个工具keytool-importkeypair，这个工具的作用是将系统签名的相关信息导入到已有的签名文件里。可从[这里](https://github.com/getfatday/keytool-importkeypair)下载。工具的使用方法参考[使用keytool工具](http://czj4451.iteye.com/blog/1487684)这篇文章。最好使用mac 或Linux系统下的AS生成，因为keytool-importkeypair工具是执行在类**Unix**内核的系统上的，不要整成**win**格式的文本了，在window上会有些问题（在git的sh执行，需要修改命令tmpdir=`mkdir -p "/tmp/${scriptname}.XXXX"`）,步骤

1. 生成demo.jks签名文件：菜单栏->build->Generate Signed APK..->

证书查看：keytool -list -v -keystore SystemSignDemo2.jks -storepass 123456

2、执行命令 ./keytool-importkeypair -k demo.jks -p 123456 -pk8 platform.pk8 -cert platform.x509.pem -alias demo

# demo.jks : 签名文件

# 123456 : 签名文件密码

# platform.pk8、platform.x509.pem : 系统签名文件

# demo : 签名文件别名

## 小结

完整的系统签名的app过程

1. 应该是使用平台签名，例如此处的Settings.apk的Android.mk文件中指定的签名平台是platform即系统平台签名，所以在签名的时候会使用系统的签名文件进行签名。platform对应的系统签名文件的位置为：

android源码根目录/build/target/product/security/platform.pk8和android源码根目录/build/target/product/security/platform.x509.pem 两个文件

2. 在应用工程的清单配置文件AndroidManifest.xml文件中指定共享用户ID，并将coreApp属性设置为true

3. ndroid源码中使用mm/mmm命令进行编译，其权限就会与系统设置一致；

4. 将apk文件复制到Android设备的/system/app目录中

# apk安装过程

1.系统应用安装

2.网络下载应用安装

3. ADB工具安装

4.第三方应用安装



## 概述

众所周知，Android应用最终是打包成.apk格式（其实就是一个压缩包），然后安装至手机并运行的。APK即Android Package的缩写。

Android系统在启动的过程中，会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，完成应用程序的安装过程。

**应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了**。

### 安装有如下四种方式

一般而言，Android应用安装有如下四种方式：

* 系统应用安装：开机时加载系统的APK和应用，没有安装界面；
* 网络下载应用安装：通过各种market应用完成，没有安装界面；
* ADB工具安装：即通过Android的SDK开发tools里面的adb.exe程序安装，没有安装界面；
* 第三方应用安装：通过SD卡里的APK文件安装(比如双击APK文件触发)，有安装界面，系统默认已经安装了一个安装卸载应用的程序，即由packageinstaller.apk应用处理安装及卸载过程的界面。

安装过程：复制APK安装包到data/app目录下，解压并扫描安装包，把dex文件(Dalvik字节码)保存到**dalvik-cache**目录，并data/data目录下创建对应的应用数据目录。

卸载过程：删除安装过程中在上述三个目录下创建的文件及目录。

### 应用安装涉及到的目录

/system/app ：系统自带的应用程序，获得adb root权限才能删除

/data/app ：用户程序安装的目录。安装时把apk文件复制到此目录

/data/data ：存放应用程序的数据

/data/dalvik-cache：将apk中的dex文件安装到dalvik-cache目录下(dex文件是dalvik虚拟机的可执行文件,当然，ART–Android Runtime的可执行文件格式为oat，启用ART时，系统会执行dex文件转换至oat文件)

/data/system ：该目录下的packages.xml文件，类似于Windows**的注册表**，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。/data/system/packages.xml中内容详解(这里列举的标签内容不一定完整，只是列举核心内容，packages.xml的完整定义详见官方文档)：

### 安装概述

#### 拷贝apk文件到指定目录

在Android系统中，apk安装文件是会被保存起来的，默认情况下，用户安装的apk首先会被拷贝到 /data/app 目录下。/data/app目录是用户有权限访问的目录，在安装apk的时候会自动选择该目录存放用户安装的文件，而系统出厂的apk文件则被放到了 /system 分区下,包括 /system/app，/system/vendor/app，以及 /system/priv-app 等等，该分区只有Root权限的用户才能访问，这也就是为什么在**没有Root手机之前，我们无法删除系统出厂的app的原因了。**

#### 解压apk，拷贝文件，创建应用的数据目录

为了加快app的启动速度，apk在安装的时候，会首先将app的可执行文件（dex）拷贝到 /data/dalvik-cache 目录，缓存起来。

然后，在/data/data/目录下创建应用程序的数据目录（以应用的包名命名），存放应用的相关数据，如数据库、xml文件、cache、二进制的so动态库等等。

#### 解析apk的AndroidManifinest.xml文件

Android系统中，也有一个类似注册表的东西，用来记录当前所有安装的应用的基本信息，每次系统安装或者卸载了任何apk文件，都会更新这个文件。这个文件位于如下目录：

/data/system/packages.xml。系统在安装apk的过程中，会解析apk的AndroidManifinest.xml文件，提取出这个apk的重要信息写入到packages.xml文件中，这些信息包括：权限、应用包名、APK的安装位置、版本、userID等等。

由此，我们就知道了为啥一些**应用市场**和软件管理类的app能够很清楚地知道当前手机所安装的所有的app，以及这些**app的详细信息**了。

另外一件事就是Linux的用户Id和用户组Id，以便他可以获得合适的运行权限。以上这些都是由PackageServiceManager完成的，下面我们会重点介绍PackageServiceManager。

#### 显示快捷方式

这些应用程序只是相当于在PackageManagerService服务注册好了，如果我们想要在Android桌面上看到这些应用程序，还需要有一个Home应用程序，负责从PackageManagerService服务中把这些安装好的应用程序取出来，并以友好的方式在桌面上展现出来，例如以快捷图标的形式。在Android系统中，负责把系统中已经安装的应用程序在桌面中展现出来的Home应用程序就是Launcher了

## PackageManagerService启动过程

SystemServer组件是由Zygote进程负责启动的，启动的时候就会调用它的main函数，Android系统在启动的过程中，会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息。应用程序管理服务PackageManagerService安装应用程序的过程，其实就是解析析应用程序配置文件AndroidManifest.xml的过程，并从里面得到得到应用程序的相关信息，例如得到应用程序的组件Activity、Service、Broadcast Receiver和Content Provider等信息，有了这些信息后，通过ActivityManagerService这个服务，我们就可以在系统中正常地使用这些应用程序了。应用程序管理服务PackageManagerService是系统启动的时候由SystemServer组件启动的，启后它就会执行应用程序安装的过程，因此，本文将从SystemServer启动PackageManagerService服务的过程开始分析系统中的应用程序安装的过程。



## PackageManagerService入口



## 安装详解

### 系统应用安装方式



**1.PackageManagerService.main()初始化注册**

将PackageManagerService服务初始化并注册到ServiceManager里面进行管理。

**2.建立java层的installer与c层的installd的socket联接**

建立Java层的installer与c层的installd的socket联接，使得在上层的install,remove,dexopt等功能最终由installd在底层实现；

**3.建立PackageHandler消息循环**

建立PackageHandler消息循环，用于处理外部的apk安装请求消息，如adb install,packageinstaller安装apk时会发送消息；

典型的比如INIT\_COPY和MCS\_BOUND等，在通过网络下载时候会调用。

**4. 成员变量readLp（）恢复上一次的安装信息**

由于Android每次启动的时候都需要安装一次信息，但是有些信息是保持不变的，例如Linux用户组Id，PackageManagerService 每次安装程序之后，都会把这些程序的信息保存下来，以便下次使用， 恢复上一次程序的安装信息是通过PackageManagerService 的成员变量mSetting的readLP()来实现的，恢复信息之后就开始扫描和安装app了。

检查/data/system/packages.xml是否存在，这个文件是在解析apk时由writeLP()创建的，里面记录了系统的permissions，以及每个apk的name,codePath,flags,ts,version,uesrid等信息，这些信息主要通过apk的AndroidManifest.xml解析获取，解析完apk后将更新信息写入这个文件并保存到flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有apk升级，安装或删除时会更新这个文件。

**5.jar的detopt优化**

检查**BootClassPath，mSharedLibraries**及/system/framework下的jar是否需要dexopt，需要的则通过dexopt进行优化；

**6.scanDirLI函数扫描特定目录的apk文件解析**

启动AppDirObserver线程监测**/system/framework,/system/app,/data/app,/data/app-privat**e目录的事件,主要监听add和remove事件。对于目录监听底层通过inotify机制实现，inotify 是一种文件系统的变化通知机制，如文件增加、删除等事件可以立刻让用户态得知,它为用户态监视文件系统的变化提供了强大的支持。当有add event时调用scanPackageLI(File , int , int)处理；当有remove event时调用removePackageLI()处理;

调用installer.install()进行安装工作,检查apk里的dex文件是否需要再优化,如果需要优化则通过辅助工具dexopt进行优化处理；将解析出的componet添加到pkg的对应列表里；对apk进行签名和证书校验,进行完整性验证。



**7.updatePermmisonLp函数分配权限**

这个函数为申请了特定资源访问权限的app，分配相应的用户组ID.

**8.writeLP()函数保存安装信息**

mSetting的writeLP（）将所获得应用程序的安装信息，保存在一个本地的配置文件中。以便下次安装的时候，将应用的信息回复过来。

### 网络下载应用安装



### ADB工具安装

其入口函数源文件为pm.java

(源文件路径：android\frameworks\base\cmds\pm\src\com\android\commands\pm\pm.java)

其中\system\framework\pm.jar 包管理库

包管理脚本 \system\bin\pm 解析

showUsage就是使用方法

|  |
| --- |
| private static void showUsage() {  System.err.println("usage: pm [list|path|install|uninstall]");  System.err.println(" pm list packages [-f]");  System.err.println(" pm list permission-groups");  System.err.println(" pm list permissions [-g] [-f] [-d] [-u] [GROUP]");  System.err.println(" pm list instrumentation [-f] [TARGET-PACKAGE]");  System.err.println(" pm list features");  System.err.println(" pm path PACKAGE");  System.err.println(" pm install [-l] [-r] [-t] [-i INSTALLER\_PACKAGE\_NAME] [-s] [-f] PATH");  System.err.println(" pm uninstall [-k] PACKAGE");  System.err.println(" pm enable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm disable PACKAGE\_OR\_COMPONENT");  System.err.println(" pm setInstallLocation [0/auto] [1/internal] [2/external]");  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*省略\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  } |

安装时候会调用 runInstall（）方法

private void runInstall() {

int installFlags = 0;

String installerPackageName = null;

String opt;

while ((opt=nextOption()) != null) {

if (opt.equals("-l")) {

installFlags |= PackageManager.INSTALL\_FORWARD\_LOCK;

} else **if (opt.equals("-r")) {**

**installFlags |= PackageManager.INSTALL\_REPLACE\_EXISTING;**

} else if (opt.equals("-i")) {

installerPackageName = nextOptionData();

if (installerPackageName == null) {

System.err.println("Error: no value specified for -i");

showUsage();

return;

}

} else if (opt.equals("-t")) {

installFlags |= PackageManager.INSTALL\_ALLOW\_TEST;

} else if (opt.equals("-s")) {

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_EXTERNAL;

} else if (opt.equals("-f")) {

// Override if -s option is specified.

installFlags |= PackageManager.INSTALL\_INTERNAL;

} else {

System.err.println("Error: Unknown option: " + opt);

showUsage();

return;

}

}

String apkFilePath = nextArg();

System.err.println("\tpkg: " + apkFilePath);

if (apkFilePath == null) {

System.err.println("Error: no package specified");

showUsage();

return;

}

PackageInstallObserver obs = new PackageInstallObserver();

try {

mPm.installPackage(Uri.fromFile(new File(apkFilePath)), obs, installFlags,

installerPackageName);

synchronized (obs) {

while (!obs.finished) {

try {

obs.wait();

} catch (InterruptedException e) {

}

}

if (obs.result == PackageManager.INSTALL\_SUCCEEDED) {

System.out.println("Success");

} else {

System.err.println("Failure ["

+ installFailureToString(obs.result)

+ "]");

}

}

} catch (RemoteException e) {

System.err.println(e.toString());

System.err.println(PM\_NOT\_RUNNING\_ERR);

}

}

其中的 因为有

mPm = IpackageManager.Stub.asInterface(ServiceManager.getService("package"));

Stub是接口IPackageManage的静态抽象类，asInterface是返回IPackageManager代理的静态方法。因为class PackageManagerService extends IPackageManager.Stub所以mPm.installPackage 调用

/\* Called when a downloaded package installation has been confirmed by the user \*/

public void installPackage(

final Uri packageURI, final IPackageInstallObserver observer, final int flags,final String installerPackageName)

这样就是从网络下载安装的入口了。



### 第三方应用安装



## 总结

1.安装和卸载都是通过PackageManager，实质上是实现了PackageManager的远程服务PackageManagerService来完成具体的操作，所有细节和逻辑均可以在PackageManagerService中跟踪查看；

2.所有安装方式殊途同归，最终就回到PackageManagerService中，然后调用底层本地代码的installd来完成。

3.再看apk 的安装过程。回个我们再看apk的安装过程，主要分为如下几部

* 拷贝apk文件到指定目录
* 解压apk，拷贝文件，创建应用的数据目录
* 解析apk的AndroidManifinest.xml文件
* 向Launcher应用申请添加创建快捷方式

## Ref

[深度探究apk安装过程](http://www.sohu.com/a/132599459_465908)

[Android应用程序安装过程解析(源码角度)](http://www.jianshu.com/p/21412a697eb0)