# AMS

在android系统中，应用程序是由Activity组成的，因此，应用程序的启动过程实际上就是应用程序中的默认Activity的启动过程，本文将详细分析应用程序框架层的源代码，了解Android应用程序的启动过程。在手机屏幕中点击应用程序图标开始——到应用MainActivity展示出来结束。

下面详细分析每一步是如何实现的。

## Luancher进程

### Step 1. Launcher.startActivitySafely

1. **void** startActivitySafely(Intent intent, Object tag) {
2. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
3. try {
4. startActivity(intent);
5. } catch (ActivityNotFoundException e) {
6. ......
7. } catch (SecurityException e) {
8. ......
9. }
10. }

MainActivity，这里是AndroidManifest.xml文件中配置的：

1. **<activity** android:name=".MainActivity"
2. android:label="@string/app\_name"**>**
3. **<intent-filter>**
4. **<action** android:name="android.intent.action.MAIN" **/>**
5. **<category** android:name="android.intent.category.LAUNCHER" **/>**
6. **</intent-filter>**
7. **</activity>**

因此，这里的intent包含的信息为：action = "android.intent.action.Main"，category="android.intent.category.LAUNCHER", cmp="shy.luo.activity/.MainActivity"，表示它要启动的Activity为shy.luo.activity.MainActivity。Intent.FLAG\_ACTIVITY\_NEW\_TASK表示要在一个新的Task中启动这个Activity，注意，Task是Android系统中的概念，它不同于进程Process的概念。简单地说，一个Task是一系列Activity的集合，这个集合是以堆栈的形式来组织的，遵循后进先出的原则。一个task可以包含多个process。





在Android系统中，应用程序是由Launcher启动起来的，其实，Launcher本身也是一个应用程序，其它的应用程序安装后，就会Launcher的界面上出现一个相应的图标，点击这个图标时，Launcher就会对应的应用程序启动起来。

### Step 2. Activity.startActivity

Launcher的父类Activity实现了startActivity函数，因此，这里就调用了Activity.startActivity函数。 这个函数实现很简单，它调用startActivityForResult来进一步处理，第二个参数传入-1表示不需要这个Actvity结束后的返回结果。

1. @Override
2. **public** **void** startActivity(Intent intent) {
3. startActivityForResult(intent, -1);
4. }

### Step 3. Activity.startActivityForResult

1. **public** **void** startActivityForResult(Intent **intent**, **int** requestCode) {
2. **if** (mParent == **null**) {
3. Instrumentation.ActivityResult ar =
4. mInstrumentation.execStartActivity(
5. **this**, mMainThread.getApplicationThread(), mToken, **this**,
6. **intent**, requestCode);
7. ......
8. } **else** {
9. ......
10. }

这里的mInstrumentation是Activity类的成员变量，它的类型是Intrumentation，定义在frameworks/base/core/java/android/app/Instrumentation.java文件中，它用来监控应用程序和系统的交互。

这里的mMainThread也是Activity类的成员变量，它的类型是ActivityThread，它代表的是应用程序的主线程，这里通过mMainThread.getApplicationThread获得它里面的ApplicationThread成员变量，它是一个Binder对象，后面我们会看到，ActivityManagerService会使用它来和ActivityThread来进行进程间通信。这里我们需注意的是，**这里的mMainThread代表的是Launcher应用程序运行的进程。**

这里的mToken也是Activity类的成员变量，它是一个Binder对象的远程接口。

### Step 4. Instrumentation.execStartActivity

在frameworks/base/core/java/android/app/Instrumentation.java文件中：

1. **public** **class** Instrumentation {
2. **public** ActivityResult execStartActivity(
3. Context who, IBinder contextThread, IBinder token, Activity target,
4. Intent intent, **int** requestCode) {
5. IApplicationThread whoThread = (IApplicationThread) contextThread;
6. **if** (mActivityMonitors != **null**) {
7. ......
8. }
9. **try** {
10. **int** result = ActivityManagerNative.getDefault()
11. .**startActivity**(whoThread, intent,
12. intent.**resolveTypeIfNeeded**(who.getContentResolver()),
13. **null**, 0, token, target != **null** ? target.mEmbeddedID : **null**,
14. requestCode, **false**, **false**);
15. ......
16. } **catch** (RemoteException e) {
17. }
18. **return** **null**;
19. }
21. }

这里的ActivityManagerNative.getDefault返回**ActivityManagerService**的远程接口，即**ActivityManagerProxy**接口，具体可以参考Android系统在新进程中启动自定义服务过程（startService）的原理分析一文。

这里的intent.resolveTypeIfNeeded返回这个intent的MIME类型，在这个例子中，没有AndroidManifest.xml设置MainActivity的MIME类型，因此，这里返回null。

这里的target不为null，但是target.mEmbddedID为null，我们不用关注。

### Step 5. ActivityManagerProxy.startActivity

定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **int** startActivity(IApplicationThread caller, Intent intent,
4. String resolvedType, Uri[] grantedUriPermissions, **int** grantedMode,
5. IBinder resultTo, String resultWho,
6. **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) **throws** RemoteException {
8. Parcel data = Parcel.obtain();
9. Parcel reply = Parcel.obtain();
10. data.writeInterfaceToken(IActivityManager.descriptor);
11. data.writeStrongBinder(caller != **null** ? caller.asBinder() : **null**);
12. intent.writeToParcel(data, 0);
13. data.writeString(resolvedType);  **//null**
14. data.writeTypedArray(grantedUriPermissions, 0);   **//null**
15. data.writeInt(grantedMode);
16. data.writeStrongBinder(resultTo);
17. data.writeString(resultWho);
18. data.writeInt(requestCode);  //-1
19. data.writeInt(onlyIfNeeded ? 1 : 0);  //false
20. data.writeInt(debug ? 1 : 0);  //false
21. **mRemote**.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);
22. reply.readException();
23. **int** result = reply.readInt();
24. reply.recycle();
25. data.recycle();
26. **return** result;
27. }
28. }

这里的参数比较多，我们先整理一下。从上面的调用可以知道，这里的参数resolvedType、grantedUriPermissions和resultWho均为null；参数caller为ApplicationThread类型的Binder实体；参数resultTo为一个Binder实体的远程接口，我们先不关注它；参数grantedMode为0，我们也先不关注它；参数requestCode为-1；参数onlyIfNeeded和debug均空false。**mRemote开始向AMS进程间通信，发送**START\_ACTIVITY\_TRANSACTION消息。

## AMS进程

上述都在launer中处理的，接下是AMS响应START\_ACTIVITY\_TRANSACTIO的进程间通信

### Step 6. ActivityManagerService.startActivity

上一步Step 5通过Binder驱动程序就进入到ActivityManagerService的startActivity函数来了，它定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **int** startActivity(IApplicationThread caller,
4. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
5. **int** grantedMode, IBinder resultTo,
6. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
7. **boolean** debug) {
8. **return** **mMainStack**.startActivityMayWait(caller, intent, resolvedType,
9. grantedUriPermissions, grantedMode, resultTo, resultWho,
10. requestCode, onlyIfNeeded, debug, **null**, **null**);
11. }
12. }

这里只是简单地将操作转发给成员变量mMainStack的startActivityMayWait函数，这里的mMainStack的类型为ActivityStack。

### ActivityStack.startActivityMayWait

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityMayWait(IApplicationThread caller,
3. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
4. **int** grantedMode, IBinder resultTo,
5. String resultWho, **int** requestCode, **boolean** onlyIfNeeded,
6. **boolean** debug, WaitResult outResult, Configuration config) {
7. **boolean** componentSpecified = intent.getComponent() != **null**;
8. // Don't modify the client's object!
9. intent = **new** Intent(intent);
10. // Collect information about the target of the Intent.
11. ActivityInfo aInfo;
12. **try** {
13. ResolveInfo rInfo =
14. AppGlobals.getPackageManager().resolveIntent(
15. intent, resolvedType,
16. PackageManager.MATCH\_DEFAULT\_ONLY
17. | ActivityManagerService.STOCK\_PM\_FLAGS);
18. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
19. } **catch** (RemoteException e) {
20. ......
21. }
23. **if** (aInfo != **null**) {
24. // Store the found target back into the intent, because now that
25. // we have it we never want to do this again.  For example, if the
26. // user navigates back to this point in the history, we should
27. // always restart the exact same activity.
28. intent.setComponent(**new** ComponentName(
29. aInfo.applicationInfo.packageName, aInfo.name));
30. ......
31. }
33. **synchronized** (mService) {
34. **int** callingPid;
35. **int** callingUid;
36. **if** (caller == **null**) {
37. ......
38. } **else** {
39. callingPid = callingUid = -1;
40. }
42. mConfigWillChange = config != **null**
43. && mService.mConfiguration.diff(config) != 0;
44. **int** res = **startActivityLocked**(caller, intent, resolvedType,
45. grantedUriPermissions, grantedMode, aInfo,
46. resultTo, resultWho, requestCode, callingPid, callingUid,
47. onlyIfNeeded, componentSpecified);
48. **return** res;
49. }

注意，从Step 6传下来的参数outResult和config均为null，此外，表达式(aInfo.applicationInfo.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0为false，因此，这里忽略了无关代码。 下面语句对参数intent的内容进行解析，得到MainActivity的相关信息，保存在aInfo变量中：

1. ActivityInfo aInfo;
2. **try** {
3. ResolveInfo rInfo =   AppGlobals.getPackageManager().resolveIntent(  intent, resolvedType,
4. PackageManager.MATCH\_DEFAULT\_ONLY
5. | ActivityManagerService.STOCK\_PM\_FLAGS);
6. aInfo = rInfo != **null** ? rInfo.activityInfo : **null**;
7. } **catch** (RemoteException e) {
8. ......
9. }

解析之后，得到的aInfo.applicationInfo.packageName的值为"k.demo.asys"，aInfo.name的值为"k.demo.asys.activity.MainActivity"，这是在这个实例的配置文件AndroidManifest.xml里面配置的。此外，函数开始的地方调用intent.getComponent()函数的返回值不为null，因此，这里的componentSpecified变量为true。

接下去就调用startActivityLocked进一步处理了。

### Step 8. ActivityStack.startActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityLocked(IApplicationThread caller,
3. Intent intent, String resolvedType,
4. Uri[] grantedUriPermissions,
5. **int** grantedMode, ActivityInfo aInfo, IBinder resultTo,
6. String resultWho, **int** requestCode,
7. **int** callingPid, **int** callingUid, **boolean** onlyIfNeeded,
8. **boolean** componentSpecified) {
9. **int** err = START\_SUCCESS;
11. ProcessRecord callerApp = **null**;
12. **if** (caller != **null**) {
13. **callerApp** = mService.getRecordForAppLocked(caller);
14. **if** (callerApp != **null**) {
15. callingPid = callerApp.pid;
16. callingUid = callerApp.info.uid;
17. }
18. }
19. ActivityRecord sourceRecord = **null**;
20. ActivityRecord resultRecord = **null**;
21. **if** (resultTo != **null**) {
22. **int** index = indexOfTokenLocked(resultTo);
23. **if** (index >= 0) {
24. sourceRecord = (ActivityRecord)mHistory.get(index);
25. **if** (requestCode >= 0 && !sourceRecord.finishing) {
26. ......
27. }
28. }
29. }
31. **int** launchFlags = intent.getFlags();
33. **if** ((launchFlags&Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0
34. && sourceRecord != **null**) {
35. ......
36. }
38. **if** (err == START\_SUCCESS && intent.getComponent() == **null**) {
39. ......
40. }
42. **if** (err == START\_SUCCESS && aInfo == **null**) {
43. ......
44. }
46. **if** (err != START\_SUCCESS) {
47. ......
48. }
49. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
50. intent, resolvedType, aInfo, mService.mConfiguration,
51. resultRecord, resultWho, requestCode, componentSpecified);
52. **return** **startActivityUncheckedLocked**(r, sourceRecord,
53. grantedUriPermissions, grantedMode, onlyIfNeeded, **true**);
54. }
55. }

从传进来的参数caller得到调用者的进程信息，并保存在callerApp变量中，这里就是Launcher应用程序的进程信息了。前面说过，参数resultTo是Launcher这个Activity里面的一个Binder对象，通过它可以获得Launcher这个Activity的相关信息，保存在sourceRecord变量中。再接下来，创建即将要启动的Activity的相关信息，并保存在r变量中：

1. ActivityRecord r = **new** ActivityRecord(mService, **this**, callerApp, callingUid,
2. intent, resolvedType, aInfo, mService.mConfiguration,
3. resultRecord, resultWho, requestCode, componentSpecified);

接着调用startActivityUncheckedLocked函数进行下一步操作。

### Step 9. ActivityStack.startActivityUncheckedLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **int** startActivityUncheckedLocked(ActivityRecord r,
3. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
4. **int** grantedMode, **boolean** onlyIfNeeded, **boolean** doResume) {
5. **final** Intent intent = r.intent;
6. **final** **int** callingUid = r.launchedFromUid;
7. **int** launchFlags = intent.getFlags();
8. // We'll invoke onUserLeaving before onPause only if the launching
9. // activity did not explicitly state that this is an automated launch.
10. mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;
11. ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)
12. != 0 ? r : **null**;
13. // If the onlyIfNeeded flag is set, then we can do this if the activity
14. // being launched is the same as the one making the call...  or, as
15. // a special case, if we do not know the caller then we count the
16. // current top activity as the caller.
17. **if** (onlyIfNeeded) {
18. ......
19. }
21. **if** (sourceRecord == **null**) {
22. ......
23. } **else** **if** (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
24. ......
25. } **else** **if** (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
26. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {
27. ......
28. }
30. **if** (r.resultTo != **null** && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
31. ......
32. }
34. **boolean** addingToTask = **false**;
35. **if** (((**launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK)** != 0 &&
36. (launchFlags&Intent.**FLAG\_ACTIVITY\_MULTIPLE\_TASK**) == 0)
37. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
38. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
39. // If bring to front is requested, and no result is requested, and
40. // we can find a task that was started with this same
41. // component, then instead of launching bring that one to the front.
42. **if** (r.resultTo == **null**) {
43. // See if there is a task to bring to the front.  If this is
44. // a SINGLE\_INSTANCE activity, there can be one and only one
45. // instance of it in the history, and it is always in its own
46. // unique task, so we do a special search.
47. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
48. ? findTaskLocked(intent, r.info)
49. : findActivityLocked(intent, r.info);
50. **if** (taskTop != **null**) {
51. ......
52. }
53. }
54. }
55. **if** (r.packageName != **null**) {
56. // If the activity being launched is the same as the one currently
57. // at the top, then we need to check if it should only be launched
58. // once.
59. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
60. **if** (top != **null** && r.resultTo == **null**) {
61. **if** (top.realActivity.equals(r.realActivity)) {
62. ......
63. }
64. }
66. }
68. **boolean** newTask = **false**;
70. // Should this be considered a new task?
71. **if** (r.resultTo == **null** && !addingToTask
72. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
73. // todo: should do better management of integers.
74. mService.mCurTask++;
75. **if** (mService.mCurTask <= 0) {
76. mService.mCurTask = 1;
77. }
78. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
79. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
80. ......
81. newTask = **true**;
82. **if** (mMainStack) {
83. mService.addRecentTaskLocked(r.task);
84. }
85. }
86. **startActivityLocked**(r, newTask, doResume);
87. **return** START\_SUCCESS;
88. }
89. }

函数首先获得intent的标志值，保存在launchFlags变量中。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION没有置位，因此 ，成员变量mUserLeaving的值为true。

这个intent的标志值的位Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP也没有置位，因此，变量notTop的值为null。

由于在这个例子的AndroidManifest.xml文件中，MainActivity没有配置launchMode属值，因此，这里的r.launchMode为默认值0，表示以标准（Standard，或者称为ActivityInfo.LAUNCH\_MULTIPLE）的方式来启动这个Activity。Activity的启动方式有四种，其余三种分别是ActivityInfo.LAUNCH\_SINGLE\_INSTANCE、ActivityInfo.LAUNCH\_SINGLE\_TASK和ActivityInfo.LAUNCH\_SINGLE\_TOP，具体可以参考官方网站<http://developer.android.com/reference/android/content/pm/ActivityInfo.html>。

传进来的参数r.resultTo为null，表示Launcher不需要等这个即将要启动的MainActivity的执行结果。

由于这个intent的标志值的位Intent.FLAG\_ACTIVITY\_NEW\_TASK被置位，而且Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK没有置位，因此，下面的if语句会被执行：

1. **if** (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
2. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
3. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
4. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
5. // If bring to front is requested, and no result is requested, and
6. // we can find a task that was started with this same
7. // component, then instead of launching bring that one to the front.
8. **if** (r.resultTo == **null**) {
9. // See if there is a task to bring to the front.  If this is
10. // a SINGLE\_INSTANCE activity, there can be one and only one
11. // instance of it in the history, and it is always in its own
12. // unique task, so we do a special search.
13. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
14. ? findTaskLocked(intent, r.info)
15. : findActivityLocked(intent, r.info);
16. **if** (taskTop != **null**) {
17. ......
18. }
19. }
20. }

这段代码的逻辑是查看一下，当前有没有Task可以用来执行这个Activity。由于r.launchMode的值不为ActivityInfo.LAUNCH\_SINGLE\_INSTANCE，因此，它通过findTaskLocked函数来查找存不存这样的Task，这里返回的结果是null，即taskTop为null，因此，需要创建一个新的Task来启动这个Activity。

接着往下看：

1. **if** (r.packageName != **null**) {
2. // If the activity being launched is the same as the one currently
3. // at the top, then we need to check if it should only be launched
4. // once.
5. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
6. **if** (top != **null** && r.resultTo == **null**) {
7. **if** (top.realActivity.equals(r.realActivity)) {
8. ......
9. }
10. }
12. }

这段代码的逻辑是看一下，当前在堆栈顶端的Activity是否就是即将要启动的Activity，有些情况下，如果即将要启动的Activity就在堆栈的顶端，那么，就不会重新启动这个Activity的别一个实例了，具体可以参考官方网站http://developer.android.com/reference/android/content/pm/ActivityInfo.html。现在处理堆栈顶端的Activity是Launcher，与我们即将要启动的MainActivity不是同一个Activity，因此，这里不用进一步处理上述介绍的情况。

执行到这里，我们知道，要在一个新的Task里面来启动这个Activity了，于是新创建一个Task：

1. **if** (r.resultTo == **null** && !addingToTask
2. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
3. // todo: should do better management of integers.
4. mService.mCurTask++;
5. **if** (mService.mCurTask <= 0) {
6. mService.mCurTask = 1;
7. }
8. r.task = **new** TaskRecord(mService.mCurTask, r.info, intent,
9. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
10. ......
11. newTask = **true**;
12. **if** (mMainStack) {
13. mService.addRecentTaskLocked(r.task);
14. }
16. }

新建的Task保存在r.task域中，同时，添加到mService中去，这里的mService就是ActivityManagerService了。

最后就进入startActivityLocked(r, newTask, doResume)进一步处理了。这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **private** **final** **void** **startActivityLocked**(ActivityRecord r, **boolean** newTask,
3. **boolean** doResume) {
4. **final** **int** NH = mHistory.size();
5. **int** addPos = -1;
6. **if** (!newTask) {
7. ......
8. }
9. // Place a new activity at top of stack, so it is next to interact
10. // with the user.
11. **if** (addPos < 0) {
12. addPos = NH;
13. }
14. // If we are not placing the new activity frontmost, we do not want
15. // to deliver the onUserLeaving callback to the actual frontmost
16. // activity
17. **if** (addPos < NH) {
18. ......
19. }
21. // Slot the activity into the history stack and proceed
22. mHistory.add(addPos, r);
23. r.inHistory = **true**;
24. r.frontOfTask = newTask;
25. r.task.numActivities++;
26. **if** (NH > 0) {
27. // We want to show the starting preview window if we are
28. // switching to a new task, or the next activity's process is
29. // not currently running.
30. ......
31. } **else** {
32. // If this is the first activity, don't do any fancy animations,
33. // because there is nothing for it to animate on top of.
34. ......
35. }
36. **if** (doResume) {
37. resumeTopActivityLocked(**null**);
38. }
39. }
40. }

这里的NH表示当前系统中历史任务的个数，这里肯定是大于0，因为Launcher已经跑起来了。当NH>0时，并且现在要切换新任务时，要做一些任务切的界面操作，这段代码我们就不看了，这里不会影响到下面启Activity的过程，有兴趣的读取可以自己研究一下。

这里传进来的参数doResume为true，于是调用resumeTopActivityLocked进一步操作。

### Step 10. ActivityStack.resumeTopActivityLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. /\*\*
3. \* Ensure that the top activity in the stack is resumed.
4. \*
5. \* @param prev The previously resumed activity, for when in the process
6. \* of pausing; can be null to call from elsewhere.
7. \*
8. \* @return Returns true if something is being resumed, or false if
9. \* nothing happened.
10. \*/
11. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
12. // Find the first activity that is not finishing.
13. ActivityRecord next = topRunningActivityLocked(**null**);
15. // Remember how we'll process this pause/resume situation, and ensure
16. // that the state is reset however we wind up proceeding.
17. **final** **boolean** userLeaving = mUserLeaving;
18. mUserLeaving = **false**;
20. **if** (next == **null**) {
21. ......
22. }
24. next.delayedResume = **false**;
26. // If the top activity is the resumed one, nothing to do.
27. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
28. ......
29. }
31. // If we are sleeping, and there is no resumed activity, and the top
32. // activity is paused, well that is the state we want.
33. **if** ((mService.mSleeping || mService.mShuttingDown)
34. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
35. ......
36. }
38. ......
40. // If we are currently pausing an activity, then don't do anything
41. // until that is done.
42. **if** (mPausingActivity != **null**) {
43. ......
44. }
46. ......
48. // We need to start pausing the current activity so the top one
49. // can be resumed...
50. **if** (mResumedActivity != **null**) {
51. ......
52. startPausingLocked(userLeaving, **false**);
53. **return** **true**;
54. }
56. ......
57. }
59. ......
61. }

函数先通过调用topRunningActivityLocked函数获得堆栈顶端的Activity，这里就是**MainActivity**了，这是在上面的Step 9设置好的，保存在next变量中。

接下来把mUserLeaving的保存在本地变量userLeaving中，然后重新设置为false，在上面的Step 9中，mUserLeaving的值为true，因此，这里的userLeaving为true。

这里的mResumedActivity为Launcher，因为Launcher是当前正被执行的Activity。

当我们处理休眠状态时，mLastPausedActivity保存堆栈顶端的Activity，因为当前不是休眠状态，所以mLastPausedActivity为null。

有了这些信息之后，下面的语句就容易理解了：

1. // If the top activity is the resumed one, nothing to do.
2. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
3. ......
4. }
6. // If we are sleeping, and there is no resumed activity, and the top
7. // activity is paused, well that is the state we want.
8. **if** ((mService.mSleeping || mService.mShuttingDown)
9. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
10. ......
11. }

就可以了；否则再看一下系统当前是否休眠状态，如果是的话，再看看要启动的Activity是否就是当前处于堆栈顶端的Activity，如果是的话，也是什么都不用做。

上面两个条件都不满足，因此，在继续往下执行之前，首先要把当处于Resumed状态的Activity推入Paused状态，然后才可以启动新的Activity。但是在将当前这个Resumed状态的Activity推入Paused状态之前，首先要看一下当前是否有Activity正在进入Pausing状态，如果有的话，当前这个Resumed状态的Activity就要稍后才能进入Paused状态了，这样就保证了所有需要进入Paused状态的Activity串行处理。

这里没有处于Pausing状态的Activity，即mPausingActivity为null，而且mResumedActivity也不为null，于是就调用startPausingLocked函数把Launcher推入Paused状态去了。

### Step 11. ActivityStack.startPausingLocked

1. **public** **class** ActivityStack {
3. ......
5. **private** **final** **void** startPausingLocked(**boolean** userLeaving, **boolean** uiSleeping) {
6. **if** (mPausingActivity != **null**) {
7. ......
8. }
9. ActivityRecord prev = mResumedActivity;
10. **if** (prev == **null**) {
11. ......
12. }
13. ......
14. mResumedActivity = **null**;
15. mPausingActivity = prev;
16. mLastPausedActivity = prev;
17. prev.state = ActivityState.PAUSING;
18. ......
20. **if** (prev.app != **null** && prev.app.thread != **null**) {
21. ......
22. **try** {
23. ......
24. prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
25. prev.configChangeFlags);
26. ......
27. } **catch** (Exception e) {
28. ......
29. }
30. } **else** {
31. ......
32. }
34. ......
36. }
38. ......
40. }

函数首先把mResumedActivity保存在本地变量prev中。在上一步Step 10中，说到mResumedActivity就是Launcher，因此，这里把Launcher进程中的ApplicationThread对象取出来，通过它来通知Launcher这个Activity它要进入Paused状态了。当然，这里的prev.app.thread是一个ApplicationThread对象的远程接口，通过调用这个远程接口的schedulePauseActivity来通知Launcher进入Paused状态。

参数prev.finishing表示prev所代表的Activity是否正在等待结束的Activity列表中，由于Laucher这个Activity还没结束，所以这里为false；参数prev.configChangeFlags表示哪些config发生了变化，这里我们不关心它的值。

### Step 12. ApplicationThreadProxy.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
3. ......
5. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
6. **boolean** userLeaving, **int** configChanges) **throws** RemoteException {
7. Parcel data = Parcel.obtain();
8. data.writeInterfaceToken(IApplicationThread.descriptor);
9. data.writeStrongBinder(token);
10. data.writeInt(finished ? 1 : 0);
11. data.writeInt(userLeaving ? 1 :0);
12. data.writeInt(configChanges);
13. mRemote.transact(SCHEDULE\_PAUSE\_ACTIVITY\_TRANSACTION, data, **null**,
14. IBinder.FLAG\_ONEWAY);
15. data.recycle();
16. }
18. ......
20. }

这个函数通过Binder进程间通信机制进入到ApplicationThread.schedulePauseActivity函数中。

## L进程

### Step 13. ApplicationThread.schedulePauseActivity

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中，它是ActivityThread的内部类：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
3. **public** **final** **void** schedulePauseActivity(IBinder token, **boolean** finished,
4. **boolean** userLeaving, **int** configChanges) {
5. queueOrSendMessage(
6. finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,
7. token,
8. (userLeaving ? 1 : 0),
9. configChanges);
10. }
11. }
12. }

这里调用的函数queueOrSendMessage是ActivityThread类的成员函数。

上面说到，这里的finished值为false，因此，queueOrSendMessage的第一个参数值为H.PAUSE\_ACTIVITY，表示要暂停token所代表的Activity，即Launcher。

### Step 14. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1) {
3. queueOrSendMessage(what, obj, arg1, 0);
4. }
5. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
6. **synchronized** (**this**) {
7. ......
8. Message msg = Message.obtain();
9. msg.what = what;
10. msg.obj = obj;
11. msg.arg1 = arg1;
12. msg.arg2 = arg2;
13. mH.sendMessage(msg);
14. }
15. }

这里首先将相关信息组装成一个msg，然后通过mH成员变量发送出去，mH的类型是H，继承于Handler类，是ActivityThread的内部类，因此，这个消息最后由H.handleMessage来处理。

### Step 15. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
2. **private** **final** **class** H **extends** Handler {
3. **public** **void** handleMessage(Message msg) {
4. **switch** (msg.what) {
5. **case** PAUSE\_ACTIVITY:
6. handlePauseActivity((IBinder)msg.obj, **false**, msg.arg1 != 0, msg.arg2);
7. maybeSnapshot();
8. **break**;
9. }

这里调用ActivityThread.handlePauseActivity进一步操作，msg.obj是一个ActivityRecord对象的引用，它代表的是Launcher这个Activity。

### Step 16. ActivityThread.handlePauseActivity

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** handlePauseActivity(IBinder token, **boolean** finished,
3. **boolean** userLeaving, **int** configChanges) {
4. ActivityClientRecord r = mActivities.get(token);
5. **if** (r != **null**) {
6. //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
7. **if** (userLeaving) {
8. performUserLeavingActivity(r);
9. }
10. r.activity.mConfigChangeFlags |= configChanges;
11. Bundle state = **performPauseActivity**(token, finished, **true**);
12. // Make sure any pending writes are now committed.
13. QueuedWork.waitToFinish();
14. // Tell the activity manager we have paused.
15. **try** {
16. ActivityManagerNative.getDefault().**activityPaused**(token, state);
17. } **catch** (RemoteException ex) {
18. }
19. }
20. }
21. }

函数首先将Binder引用token转换成ActivityRecord的远程接口ActivityClientRecord，然后做了三个事情：1. 如果userLeaving为true，则通过调用performUserLeavingActivity函数来调用Activity.onUserLeaveHint通知Activity，用户要离开它了；2. 调用**performPauseActivity函数来调用Activity.onPause函数**，我们知道，在Activity的生命周期中，当它要让位于其它的Activity时，系统就会调用它的onPause函数；3. 它通知ActivityManagerService，这个Activity已经进入Paused状态了，ActivityManagerService现在可以完成未竟的事情，即启动MainActivity了。

### Step 17. ActivityManagerProxy.activityPaused

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. ......
5. **public** **void** activityPaused(IBinder token, Bundle state) **throws** RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(token);
11. data.writeBundle(state);
12. mRemote.transact(ACTIVITY\_PAUSED\_TRANSACTION, data, reply, 0);
13. reply.readException();
14. data.recycle();
15. reply.recycle();
16. }
18. ......
20. }

这里通过Binder进程间通信机制就进入到ActivityManagerService.activityPaused函数中去了。

## AMS

### Step 18. ActivityManagerService.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **void** activityPaused(IBinder token, Bundle icicle) {
4. **final** **long** origId = Binder.clearCallingIdentity();
5. mMainStack.activityPaused(token, icicle, **false**);
6. }
7. }

这里，又再次进入到ActivityStack类中，执行activityPaused函数。

### Step 19. ActivityStack.activityPaused

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

1. **public** **class** ActivityStack {
2. **final** **void** activityPaused(IBinder token, Bundle icicle, **boolean** timeout) {
3. ActivityRecord r = **null**;
4. **synchronized** (mService) {
5. **int** index = indexOfTokenLocked(token);
6. **if** (index >= 0) {
7. r = (ActivityRecord)mHistory.get(index);
8. **if** (!timeout) {
9. r.icicle = icicle;
10. r.haveState = **true**;
11. }
12. mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);
13. **if** (mPausingActivity == r) {
14. r.state = ActivityState.PAUSED;
15. completePauseLocked();
16. } **else** {
17. ......
18. }
19. }
20. }
21. }
22. }

这里通过参数token在mHistory列表中得到ActivityRecord，从上面我们知道，这个ActivityRecord代表的是Launcher这个Activity，而我们在Step 11中，把Launcher这个Activity的信息保存在mPausingActivity中，因此，这里mPausingActivity等于r，于是，执行completePauseLocked操作。

### Step 20. ActivityStack.completePauseLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** completePauseLocked() {
3. ActivityRecord prev = mPausingActivity;
4. **if** (prev != **null**) {
5. mPausingActivity = **null**;
6. }
7. **if** (!mService.mSleeping && !mService.mShuttingDown) {
8. resumeTopActivityLocked(prev);
9. } **else** {
10. }
11. }
12. }

函数首先把mPausingActivity变量清空，因为现在不需要它了，然后调用resumeTopActivityLokced进一步操作，它传入的参数即为代表Launcher这个Activity的ActivityRecord。

### Step 21. ActivityStack.resumeTopActivityLokced

1. **public** **class** ActivityStack {
2. **final** **boolean** resumeTopActivityLocked(ActivityRecord prev) {
3. // Find the first activity that is not finishing.
4. ActivityRecord next = topRunningActivityLocked(**null**);
6. // Remember how we'll process this pause/resume situation, and ensure
7. // that the state is reset however we wind up proceeding.
8. **final** **boolean** userLeaving = mUserLeaving;
9. mUserLeaving = **false**;
10. next.delayedResume = **false**;
12. // If the top activity is the resumed one, nothing to do.
13. **if** (mResumedActivity == next && next.state == ActivityState.RESUMED) {
14. ......
15. **return** **false**;
16. }
18. // If we are sleeping, and there is no resumed activity, and the top
19. // activity is paused, well that is the state we want.
20. **if** ((mService.mSleeping || mService.mShuttingDown)
21. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
22. ......
23. **return** **false**;
24. }
26. .......

29. // We need to start pausing the current activity so the top one
30. // can be resumed...
31. **if** (mResumedActivity != **null**) {
32. ......
33. **return** **true**;
34. }
35. **if** (next.app != **null** && next.app.thread != **null**) {
36. } **else** {
37. startSpecificActivityLocked(next, **true**, **true**);
38. }
40. **return** **true**;
41. }
42. }

通过上面的Step 9，我们知道，**当前在堆栈顶端的Activity为我们即将要启动的MainActivity**，这里通过调用topRunningActivityLocked将它取回来，保存在next变量中。之前最后一个Resumed状态的Activity，即Launcher，到了这里已经处于Paused状态了，因此，mResumedActivity为null。最后一个处于Paused状态的Activity为Launcher，因此，这里的mLastPausedActivity就为Launcher。前面我们为MainActivity创建了ActivityRecord后，它的app域一直保持为null。有了这些信息后，上面这段代码就容易理解了，它最终调用startSpecificActivityLocked进行下一步操作。

### Step 22. ActivityStack.startSpecificActivityLocked

1. **public** **class** ActivityStack {
2. **private** **final** **void** startSpecificActivityLocked(ActivityRecord r,
3. **boolean** andResume, **boolean** checkConfig) {
4. // Is this activity's application already running?
5. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
6. r.info.applicationInfo.uid);
7. **if** (app != **null** && app.thread != **null**) {
8. **try** {
9. realStartActivityLocked(r, app, andResume, checkConfig);
10. **return**;
11. } **catch** (RemoteException e) {
12. ......
13. }
14. }
16. mService.startProcessLocked(r.processName, r.info.applicationInfo, **true**, 0,
17. "activity", r.intent.getComponent(), **false**);
18. }
19. }

注意，这里由于是第一次启动应用程序的Activity，所以下面语句：

ProcessRecord app = mService.getProcessRecordLocked(r.processName, r.info.applicationInfo.uid); 取回来的app为null。

在Activity应用程序中的AndroidManifest.xml配置文件中，我们没有指定Application标签的process属性，系统就会默认使用package的名称，这里就是"k.demo.system"了。每一个应用程序都有自己的uid，因此，这里uid + process的组合就可以为每一个应用程序创建一个ProcessRecord。当然，我们可以配置两个应用程序具有相同的uid和package，或者在AndroidManifest.xml配置文件的application标签或者activity标签中显式指定相同的process属性值，这样，不同的应用程序也可以在同一个进程中启动。

函数最终执行ActivityManagerService.startProcessLocked函数进行下一步操作。

### Step 23. ActivityManagerService.startProcessLocked

这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityManagerService.java文件中：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **final** ProcessRecord startProcessLocked(String processName,
4. ApplicationInfo info, **boolean** knownToBeDead, **int** intentFlags,
5. String hostingType, ComponentName hostingName, **boolean** allowWhileBooting) {
6. ProcessRecord app = getProcessRecordLocked(processName, info.uid);
7. String hostingNameStr = hostingName != **null**
8. ? hostingName.flattenToShortString() : **null**;
9. **if** (app == **null**) {
10. app = **new** ProcessRecordLocked(**null**, info, processName);
11. mProcessNames.put(processName, info.uid, app);
12. } **else** {
13. // If this is a new package in the process, add the package to the list
14. app.addPackage(info.packageName);
15. }
16. startProcessLocked(app, hostingType, hostingNameStr);
17. **return** (app.pid != 0) ? app : **null**;
18. }
19. }

这里再次检查是否已经有以process + uid命名的进程存在，在我们这个情景中，返回值app为null，因此，后面会创建一个ProcessRecord，并存保存在成员变量mProcessNames中，最后，调用另一个startProcessLocked函数进一步操作：

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **private** **final** **void** startProcessLocked(ProcessRecord app,
4. String hostingType, String hostingNameStr) {
5. **try** {
6. **int** uid = app.info.uid;
7. **int**[] gids = **null**;
8. **try** {
9. gids = mContext.getPackageManager().getPackageGids(
10. app.info.packageName);
11. } **catch** (PackageManager.NameNotFoundException e) {
12. }
13. **int** debugFlags = 0;
14. **int** pid = Process.start("android.app.ActivityThread",
15. mSimpleProcessManagement ? app.processName : **null**, uid, uid,
16. gids, debugFlags, **null**);
17. } **catch** (RuntimeException e) {
18. }
19. }
20. }

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

### Step 24. ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. mgr.attachApplication(mAppThread);
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了ActivityManagerService的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的。

### Step 25. ActivityManagerProxy.attachApplication

这个函数定义在frameworks/base/core/java/android/app/ActivityManagerNative.java文件中：

1. **class** ActivityManagerProxy **implements** IActivityManager
2. {
3. **public** **void** attachApplication(IApplicationThread app) **throws** RemoteException
4. {
5. Parcel data = Parcel.obtain();
6. Parcel reply = Parcel.obtain();
7. data.writeInterfaceToken(IActivityManager.descriptor);
8. data.writeStrongBinder(app.asBinder());
9. mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);
10. reply.readException();
11. data.recycle();
12. reply.recycle();
13. }
14. }

这里通过Binder驱动程序，最后进入ActivityManagerService的attachApplication函数中。

## AMS

### Step 26. ActivityManagerService.attachApplication

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. **public** **final** **void** attachApplication(IApplicationThread thread) {
4. **synchronized** (**this**) {
5. **int** callingPid = Binder.getCallingPid();
6. **final** **long** origId = Binder.clearCallingIdentity();
7. attachApplicationLocked(thread, callingPid);
8. Binder.restoreCallingIdentity(origId);
9. }
10. }
11. }

这里将操作转发给attachApplicationLocked函数。

### Step 27. ActivityManagerService.attachApplicationLocked

1. **public** **final** **class** ActivityManagerService **extends** ActivityManagerNative
2. **implements** Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. **private** **final** **boolean** attachApplicationLocked(IApplicationThread thread,
7. **int** pid) {
8. // Find the application record that is being attached...  either via
9. // the pid if we are running in multiple processes, or just pull the
10. // next app record if we are emulating process with anonymous threads.
11. ProcessRecord app;
12. **if** (pid != MY\_PID && pid >= 0) {
13. **synchronized** (mPidsSelfLocked) {
14. app = mPidsSelfLocked.get(pid);
15. }
16. } **else** **if** (mStartingProcesses.size() > 0) {
17. ......
18. } **else** {
19. ......
20. }
22. **if** (app == **null**) {
23. ......
24. **return** **false**;
25. }
27. ......
29. String processName = app.processName;
30. **try** {
31. thread.asBinder().linkToDeath(**new** AppDeathRecipient(
32. app, pid, thread), 0);
33. } **catch** (RemoteException e) {
34. ......
35. **return** **false**;
36. }
38. ......
40. app.thread = thread;
41. app.curAdj = app.setAdj = -100;
42. app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;
43. app.setSchedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;
44. app.forcingToForeground = **null**;
45. app.foregroundServices = **false**;
46. app.debugging = **false**;
48. ......
50. **boolean** normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
52. ......
54. **boolean** badApp = **false**;
55. **boolean** didSomething = **false**;
57. // See if the top visible activity is waiting to run in this process...
58. ActivityRecord hr = mMainStack.topRunningActivityLocked(**null**);
59. **if** (hr != **null** && normalMode) {
60. **if** (hr.app == **null** && app.info.uid == hr.info.applicationInfo.uid
61. && processName.equals(hr.processName)) {
62. **try** {
63. **if** (mMainStack.realStartActivityLocked(hr, app, **true**, **true**)) {
64. didSomething = **true**;
65. }
66. } **catch** (Exception e) {
67. ......
68. }
69. } **else** {
70. ......
71. }
72. }
74. ......
76. **return** **true**;
77. }
79. ......
81. }

在前面的Step 23中，已经创建了一个ProcessRecord，这里首先通过pid将它取回来，放在app变量中，然后对app的其它成员进行初始化，最后调用mMainStack.realStartActivityLocked执行真正的Activity启动操作。这里要启动的Activity通过调用mMainStack.topRunningActivityLocked(null)从堆栈顶端取回来，这时候在堆栈顶端的Activity就是MainActivity了。

### Step 28. ActivityStack.realStartActivityLocked

1. **public** **class** ActivityStack {
3. ......
5. **final** **boolean** realStartActivityLocked(ActivityRecord r,
6. ProcessRecord app, **boolean** andResume, **boolean** checkConfig)
7. **throws** RemoteException {
9. ......
11. r.app = app;
13. ......
15. **int** idx = app.activities.indexOf(r);
16. **if** (idx < 0) {
17. app.activities.add(r);
18. }
20. ......
22. **try** {
23. ......
25. List<ResultInfo> results = **null**;
26. List<Intent> newIntents = **null**;
27. **if** (andResume) {
28. results = r.results;
29. newIntents = r.newIntents;
30. }
32. ......
34. app.thread.scheduleLaunchActivity(**new** Intent(r.intent), r,
35. System.identityHashCode(r),
36. r.info, r.icicle, results, newIntents, !andResume,
37. mService.isNextTransitionForward());
39. ......
41. } **catch** (RemoteException e) {
42. ......
43. }
45. ......
47. **return** **true**;
48. }
50. ......
52. }

这里最终通过app.thread进入到ApplicationThreadProxy的scheduleLaunchActivity函数中，注意，这里的第二个参数r，是一个ActivityRecord类型的Binder对象，用来作来这个Activity的token值。

### Step 29. ApplicationThreadProxy.scheduleLaunchActivity

这个函数定义在frameworks/base/core/java/android/app/ApplicationThreadNative.java文件中：

1. **class** ApplicationThreadProxy **implements** IApplicationThread {
2. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
3. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
4. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward)
5. **throws** RemoteException {
6. Parcel data = Parcel.obtain();
7. data.writeInterfaceToken(IApplicationThread.descriptor);
8. intent.writeToParcel(data, 0);
9. data.writeStrongBinder(token);
10. data.writeInt(ident);
11. info.writeToParcel(data, 0);
12. data.writeBundle(state);
13. data.writeTypedList(pendingResults);
14. data.writeTypedList(pendingNewIntents);
15. data.writeInt(notResumed ? 1 : 0);
16. data.writeInt(isForward ? 1 : 0);
17. mRemote.transact(SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION, data, **null**,
18. IBinder.FLAG\_ONEWAY);
19. data.recycle();
20. }
21. }

这个函数最终通过Binder驱动程序进入到ApplicationThread的scheduleLaunchActivity函数中。

### Step 30. ApplicationThread.scheduleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // we use token to identify this activity without having to send the
10. // activity itself back to the activity manager. (matters more with ipc)
11. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
12. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
13. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward) {
14. ActivityClientRecord r = **new** ActivityClientRecord();
16. r.token = token;
17. r.ident = ident;
18. r.intent = intent;
19. r.activityInfo = info;
20. r.state = state;
22. r.pendingResults = pendingResults;
23. r.pendingIntents = pendingNewIntents;
25. r.startsNotResumed = notResumed;
26. r.isForward = isForward;
28. queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);
29. }
31. ......
33. }
35. ......
36. }

函数首先创建一个ActivityClientRecord实例，并且初始化它的成员变量，然后调用ActivityThread类的queueOrSendMessage函数进一步处理。

### Step 31. ActivityThread.queueOrSendMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // if the thread hasn't started yet, we don't have the handler, so just
10. // save the messages until we're ready.
11. **private** **final** **void** queueOrSendMessage(**int** what, Object obj) {
12. queueOrSendMessage(what, obj, 0, 0);
13. }
15. ......
17. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
18. **synchronized** (**this**) {
19. ......
20. Message msg = Message.obtain();
21. msg.what = what;
22. msg.obj = obj;
23. msg.arg1 = arg1;
24. msg.arg2 = arg2;
25. mH.sendMessage(msg);
26. }
27. }
29. ......
31. }
33. ......
34. }

函数把消息内容放在msg中，然后通过mH把消息分发出去，这里的成员变量mH我们在前面已经见过，消息分发出去后，最后会调用H类的handleMessage函数。

### Step 32. H.handleMessage

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. **case** LAUNCH\_ACTIVITY: {
13. ActivityClientRecord r = (ActivityClientRecord)msg.obj;
15. r.packageInfo = getPackageInfoNoCheck(
16. r.activityInfo.applicationInfo);
17. handleLaunchActivity(r, **null**);
18. } **break**;
19. ......
20. }
22. ......
24. }
26. ......
27. }

这里最后调用ActivityThread类的handleLaunchActivity函数进一步处理。

### Step 33. ActivityThread.handleLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **void** handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
6. ......
8. Activity a = performLaunchActivity(r, customIntent);
10. **if** (a != **null**) {
11. r.createdConfig = **new** Configuration(mConfiguration);
12. Bundle oldState = r.state;
13. handleResumeActivity(r.token, **false**, r.isForward);
15. ......
16. } **else** {
17. ......
18. }
19. }
21. ......
22. }

这里首先调用performLaunchActivity函数来加载这个Activity类，即shy.luo.activity.MainActivity，然后调用它的onCreate函数，最后回到handleLaunchActivity函数时，再调用handleResumeActivity函数来使这个Activity进入Resumed状态，即会调用这个Activity的onResume函数，这是遵循Activity的生命周期的。

### Step 34. ActivityThread.performLaunchActivity

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
7. ActivityInfo aInfo = r.activityInfo;
8. **if** (r.packageInfo == **null**) {
9. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
10. Context.CONTEXT\_INCLUDE\_CODE);
11. }
13. ComponentName component = r.intent.getComponent();
14. **if** (component == **null**) {
15. component = r.intent.resolveActivity(
16. mInitialApplication.getPackageManager());
17. r.intent.setComponent(component);
18. }
20. **if** (r.activityInfo.targetActivity != **null**) {
21. component = **new** ComponentName(r.activityInfo.packageName,
22. r.activityInfo.targetActivity);
23. }
25. Activity activity = **null**;
26. **try** {
27. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
28. activity = mInstrumentation.newActivity(
29. cl, component.getClassName(), r.intent);
30. r.intent.setExtrasClassLoader(cl);
31. **if** (r.state != **null**) {
32. r.state.setClassLoader(cl);
33. }
34. } **catch** (Exception e) {
35. ......
36. }
38. **try** {
39. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);
41. ......
43. **if** (activity != **null**) {
44. ContextImpl appContext = **new** ContextImpl();
45. appContext.init(r.packageInfo, r.token, **this**);
46. appContext.setOuterContext(activity);
47. CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
48. Configuration config = **new** Configuration(mConfiguration);
49. ......
50. activity.attach(appContext, **this**, getInstrumentation(), r.token,
51. r.ident, app, r.intent, r.activityInfo, title, r.parent,
52. r.embeddedID, r.lastNonConfigurationInstance,
53. r.lastNonConfigurationChildInstances, config);
55. **if** (customIntent != **null**) {
56. activity.mIntent = customIntent;
57. }
58. r.lastNonConfigurationInstance = **null**;
59. r.lastNonConfigurationChildInstances = **null**;
60. activity.mStartedActivity = **false**;
61. **int** theme = r.activityInfo.getThemeResource();
62. **if** (theme != 0) {
63. activity.setTheme(theme);
64. }
66. activity.mCalled = **false**;
67. mInstrumentation.callActivityOnCreate(activity, r.state);
68. ......
69. r.activity = activity;
70. r.stopped = **true**;
71. **if** (!r.activity.mFinished) {
72. activity.performStart();
73. r.stopped = **false**;
74. }
75. **if** (!r.activity.mFinished) {
76. **if** (r.state != **null**) {
77. mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
78. }
79. }
80. **if** (!r.activity.mFinished) {
81. activity.mCalled = **false**;
82. mInstrumentation.callActivityOnPostCreate(activity, r.state);
83. **if** (!activity.mCalled) {
84. **throw** **new** SuperNotCalledException(
85. "Activity " + r.intent.getComponent().toShortString() +
86. " did not call through to super.onPostCreate()");
87. }
88. }
89. }
90. r.paused = **true**;
92. mActivities.put(r.token, r);
94. } **catch** (SuperNotCalledException e) {
95. ......
97. } **catch** (Exception e) {
98. ......
99. }
101. **return** activity;
102. }
104. ......
105. }

函数前面是收集要启动的Activity的相关信息，主要package和component信息：

1. ActivityInfo aInfo = r.activityInfo;
2. **if** (r.packageInfo == **null**) {
3. r.packageInfo = getPackageInfo(aInfo.applicationInfo,
4. Context.CONTEXT\_INCLUDE\_CODE);
5. }
7. ComponentName component = r.intent.getComponent();
8. **if** (component == **null**) {
9. component = r.intent.resolveActivity(
10. mInitialApplication.getPackageManager());
11. r.intent.setComponent(component);
12. }
14. **if** (r.activityInfo.targetActivity != **null**) {
15. component = **new** ComponentName(r.activityInfo.packageName,
16. r.

然后通过ClassLoader将shy.luo.activity.MainActivity类加载进来：

1. Activity activity = **null**;
2. **try** {
3. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
4. activity = mInstrumentation.newActivity(
5. cl, component.getClassName(), r.intent);
6. r.intent.setExtrasClassLoader(cl);
7. **if** (r.state != **null**) {
8. r.state.setClassLoader(cl);
9. }
10. } **catch** (Exception e) {
11. ......
12. }

接下来是创建Application对象，这是根据AndroidManifest.xml配置文件中的Application标签的信息来创建的：

1. Application app = r.packageInfo.makeApplication(**false**, mInstrumentation);

后面的代码主要创建Activity的上下文信息，并通过attach方法将这些上下文信息设置到MainActivity中去：

1. activity.attach(appContext, **this**, getInstrumentation(), r.token,
2. r.ident, app, r.intent, r.activityInfo, title, r.parent,
3. r.embeddedID, r.lastNonConfigurationInstance,
4. r.lastNonConfigurationChildInstances, config);

最后还要调用MainActivity的onCreate函数：

Instrumentation.callActivityOnCreate(activity, r.state);

这里不是直接调用MainActivity的onCreate函数，而是通过mInstrumentation的callActivityOnCreate函数来间接调用，前面我们说过，mInstrumentation在这里的作用是监控Activity与系统的交互操作，相当于是系统运行日志。

### Step 35. MainActivity.onCreate

1. **public** **class** MainActivity **extends** Activity  **implements** OnClickListener {
3. ......
5. @Override
6. **public** **void** onCreate(Bundle savedInstanceState) {
7. ......
9. Log.i(LOG\_TAG, "Main Activity Created.");
10. }
12. ......

## 小结

这样，MainActivity就启动起来了，整个应用程序也启动起来了。

       整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知ActivityManagerService，它要启动一个Activity；

       二. Step 12 - Step 16：ActivityManagerService通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给ActivityManagerService，以便以后ActivityManagerService能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：ActivityManagerService通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

       这里不少地方涉及到了Binder进程间通信机制，相关资料请参考[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)一文。

       这样，应用程序的启动过程就介绍完了，它实质上是启动应用程序的默认Activity，在下一篇文章中，我们将介绍在应用程序内部启动另一个Activity的过程，即新的Activity与启动它的Activity将会在同一个进程（Process）和任务（Task）运行，敬请关注。

## 参考

[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)

[**Android深入四大组件（一）应用程序启动过程**](http://blog.csdn.net/itachi85/article/details/69388942)

http://blog.csdn.net/itachi85/article/details/69388942

# startService的原理分析

http://blog.csdn.net/luoshengyang/article/details/6677029

# apk安装过程PMS

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)：

/data/system/packages.xml：通过它可以看到系统安装的所有软件包，以及软件包的信息

### 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

## 源码分析

### 配置文件package.xml

/data/system/packages.xml

通过它可以看到系统安装的所有软件包，以及软件包的信息

系统自带的软件能升级吗（即安装在系统分区system中的包，如电话，短信），可以升级，如果升级/system/app目录中的包，PackageManagerServer.java对此情况进行处理，被升级的包出现package.xml的updated-package字段中，新的包信息会写在package字段中，卸载新包后，原包会恢复到package字段中。启动时新的包会优先地被启动

|  |
| --- |
| **<updated-package** name="com.android.providers.settings" codePath="/system/priv-app/SettingsProvider" ft="15e3b5e10b0" it="15e2e387ad8" ut="15e3b5e10b0" version="22" nativeLibraryPath="/system/priv-app/SettingsProvider/lib" primaryCpuAbi="armeabi-v7a" sharedUserId="1000">  <perms />  **</updated-package>** |

## 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)

# 属性系统build.prop

## 概述

### build.prop是什么？

在Android中.prop类型的文件是一个属性文件，记录一些系统设置。.prop文件类似/etc中的文件。更简单的来说，就相当于Windows中的系统注册表。所以，build.prop中记录了一些系统的设置属性。这些属性包括系统初始或固定的一些参数属性、功能的开放等。通过调整/增加这些参数可以达到较调系统性能偏重点和附加功能开启的作用。

### build.prop是怎样生成的？

首先需要确定build.prop在编译过程中是由哪个Makefile、.mk或编译脚本控制的。熟悉Android编译过程的可能很容易定位到./build/core/Makefile这个文件。如果不熟悉Android编译过程可以通过查看Android整编的log。在./build/core/Makefile中，可以看到：

# build.prop

INSTALLED\_BUILD\_PROP\_TARGET := $(TARGET\_OUT)/build.prop

ALL\_DEFAULT\_INSTALLED\_MODULES += $(INSTALLED\_BUILD\_PROP\_TARGET)

ADDITIONAL\_BUILD\_PROPERTIES := \

$(call collapse-pairs, $(ADDITIONAL\_BUILD\_PROPERTIES))

ADDITIONAL\_BUILD\_PROPERTIES := $(call uniq-pairs-by-first-component, \

$(ADDITIONAL\_BUILD\_PROPERTIES),=)

ifdef TARGET\_SYSTEM\_PROP

system\_prop\_file := $(TARGET\_SYSTEM\_PROP)

else

system\_prop\_file := $(wildcard $(TARGET\_DEVICE\_DIR)/system.prop)

endif

通过Makefile文件，可以得到，build.prop的来源有：

* ./build/tools/buildinfo.sh：将buildinfo.sh的内容，直接echo到build.prop中
* system.prop：把默认的system.prop内容追加到build.prop中。如，./device/jrdcom/Alto5TF/system.prop
* ADDITIONAL\_BUILD\_PROPERTIES：把ADDITIONAL\_BUILD\_PROPERTIES中的属性追加到build.prop
* ODM/OED厂商的定制：一些厂商一般都会有自己的一些额外定制

### build.prop读/写

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Code-java | 主要通过SystemProperties中提供的get相关的方法；者直接用类**Build**.java。 | get(String key, String def)  **Build.XXX** | frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java调用jni  \frameworks\base\core\jni\android\_os\_SystemProperties.cpp |  |
| C/C++层 |  | property\_get("xxxx", value, "0"); |  |  |
| Adb命令 |  | adb shell getprop xxxx | adb shell setprop xxxx  临时修改，不重启系统 |  |
| 文件修改 | /system/build.prop | Pull/push要修改build.prop权限。可以改为644，重启生效。 |  |  |
|  |  |  |  |  |

### 特殊生命周期

如果属性名称以“ro.”开头，那么这个属性被视为只读属性。一旦设置，属性值不能改变。

如果属性名称以“persist.”开头，当设置这个属性时，其值也将写入/data/property。

**特别属性**

## 参数详解

build.prop中参数包括

1. Dalvik虚拟机相关参数
2. 系统版本、定义相关参数、
3. 基本性能相关参数、
4. 基本耗电相关参数、
5. 扩展性能较调及附加功能开启

### Dalvik虚拟机相关参数

**dalvik.vm.** **heapstartsize**

本参数控制Dalvik虚拟机在启动一个应用程序之后为其分配的初始堆栈大小，可填写的值为**2m~48m**。例如：dalvik.vm.heapstartsize=8m，就表示应用程序启动后为其分配的初始堆栈大小为8兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程度，和第一次使用应用程序时的流畅程序。这个值越大，系统消耗RAM则越快，但是应用程序打开后的反应也越快。值越小，系统的RAM剩余则越多，但是程序在启动后会很卡。建议值是8m，既可以保持140M左右的RAM，程序的反应速度也会大幅度提高。

**dalvik.vm.heapsize**

本参数控制Dalvik虚拟机给一个应用程序分配的最大堆栈量，可填写的值为12m~48m。例如：dalvik.vm.heapsize=48m，就表示应用程序在任意时刻内可以使用的最大堆栈大小为48兆字节。这里分配的内存容量会影响到整个系统对RAM的使用程序，和程序在运行一段时间后的反应速度。**这个值越大，系统消耗RAM则越快，但是程序会运行的非常稳定**，尤其是游戏和视频程序的内容加载速度可以大幅度提升。值越小，系统的RAM剩余则越多，但是程序会很卡，尤其是游戏在切换场景Loading的时候会花费很多的时间。若应用程序需要使用超过这个值的内存时，将会触发系统的垃圾收集器，系统和程序就会卡顿。建议值是40~40m。

**dalvik.vm.lockprof.threshold**

本参数控制Dalvik虚拟机调试记录程序内部锁资源争夺的阈值，默认值是500。多用于程序的数据统计，对性能较调意义不大。

**dalvik.vm.stack-trace-file**

本参数控制Dalvik虚拟机的堆栈记录调试文件。用于系统调试，一般用户对其调整无意义,默认为/data/anr/traces.txt，这是开发者分析anr的重要文件。

**dalvik.vm.execution-mode**

本参数控制Dalvik虚拟机的程序执行机制。可填写的值有"int:portable"、"int:fast"和"int:jit"。int:portable表示以兼容模式运行(脚本翻译模式)，此模式下程序的兼容性最高，但其执行效率最低(程序优化度依赖于dalvik虚拟机版本)。官方默认此模式。int:fast表示以快速自优化模式运行(脚本翻译和预优化混合)，此模式下程序的兼容性很高，执行效率也比较高。因为此时dalvik虚拟机允许程序使用自己的预定义优化模式和代码(包括C/C  /汇编代码)。推荐使用。**int:jit表示以Just-In-Time**模式运行(JIT模式)，此模式下程序的兼容性最差，**但程序一旦加载后其运行效率最高(与C/C  直接编写的程序效率无异)**，因为在此模式下dalvik虚拟机会预先将Java程序翻译成针对机器平台的本地语言(Native)，同时完全允许代码中的所有预优化和代码，**允许所有不安全的非托管代码，同时不严谨的程序如果运行在JIT模式可能会造成内存泄露**。但要注意，**很多Dalvik虚拟机并不支持此模式**(如官方2.2)。

**dalvik.vm.dexopt-flags**

本参数控制Dalvik虚拟机的程序代码校验和优化。可填写的值有m、v和o。m为标准选项，可以是m=y或m=n，若m=y则启用不安全代码的校验和托管代码的优化，兼容性和安全性最高，推荐使用；

v为校验选项，可与o并存，可以是v=a或v=n。若v=a则表示校验所有代码，v=n则关闭代码的校验

o为优化选项，可与v并存。可以是o=v或o=a。若o=v则表示优化以校验过的代码，o=a则表示优化所有代码。例如：dalvik.vm.dexopt-flags=m=ydalvik.vm.dexopt-flags=v=n,o=v注意，这个参数只会影响到安装APK之后或初次使用APK时生成dex文件时有效。若整个系统(包括应用程序)为odex化，则无意义。

**dalvik.vm.verify-bytecode**

本参数控制Dalvik虚拟机是否验证应用程序的可执行代码。可以与上一个参数配合使用。可填写的值为true和false。其具体意义与dalvik.vm.dexopt-flags的v=n一模一样。但可以与dalvik.vm.dexopt-flags配合使用以取得更好的效果。例如：dalvik.vm.dexopt-flags=v=n,o=vdalvik.vm.verify-bytecode=false这样可以令后来安装的apk文件可以被优化而不被检验。dalvik.vm.checkjni，本参数控制Dalvik虚拟机在调用外部jni链接库的时候是否对其做安全性检验。可填写的值为true和false。此参数会覆盖ro.kernel.android.checkjni。若值为true，会增加程序的兼容性和稳定性，但也会增加其加载和执行的时间。推荐为false。

**dalvik.vm.deadlock-predict**

本参数控制Dalvik虚拟机对程序死锁预测处理。可填写的值有off、warn和err。off表示关闭死锁预测功能(默认设置)。warn表示在继续程序运行的同时只记录该死锁预测(如果为真死锁就会出现程序假死现象，然后等N久出现关闭)。err表示预测到死锁时马上弹出FC。注意：有些Dalvik虚拟机版本并不支持此参数。

超级急速流畅型

dalvik.vm.startheapsize=16m

dalvik.vm.heapsize=48m

dalvik.vm.execution-mode=int:jit

dalvik.vm.dexopt-flags=v=n,o=v

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

常用稳定加流畅型：

dalvik.vm.startheapsize=8m

dalvik.vm.heapsize=40m

dalvik.vm.execution-mode=int:fast

dalvik.vm.dexopt-flags=m=y

dalvik.vm.verify-bytecode=false

dalvik.vm.checkjni=false

超级稳定大内存型：

dalvik.vm.startheapsize=4m

dalvik.vm.heapsize=30m

dalvik.vm.execution-mode=int:portable

dalvik.vm.dexopt-flags=v=a,o=v

dalvik.vm.verify-bytecode=true

dalvik.vm.checkjni=true

**(以荣耀一代为例，RAM 512M，以下是像我一样追求速度的童鞋的推荐参数)**  
**dalvik.vm.startheapsize=16m  
dalvik.vm.heapsize=64m（RAM大的童鞋这个参数可以再大一些）  
dalvik.vm.execution-mode=int:jit  
dalvik.vm.dexopt-flags=v=n,o=v  
dalvik.vm.verify-bytecode=false  
dalvik.vm.checkjni=false**

### 系统版本、定义等参数

**ro.build.id**

本参数定义了系统的版本ID。为系统内部使用，OTA时作为粗略版本比较。更改后可避免OTA提示，但可能会引起预装程序(如Blur)的稳定性。

**ro.build.display.id**

本参数定义了设置中显示的系统版本号。主要用于设置中显式出现可读版本，一般用于个性化定制和第三方应用程序对系统版本的判断(如魔趣设置)。更改后可自定义版本显示，但某些第三方应用程序会出现错误(如魔趣设置无法实现机器保修查询)。

**ro.build.version.incremental**

本参数定义了系统的升级字。主要用于系统OTA精确版本比对，同时与ro.build.description和ro.build.fingerprint相匹配。更改后可以免OTA提示(如避免Miui的升级提示和Blur的升级提示)。

**ro.product.model**

本参数定义了机器的型号字符串。主要用于机器型号**显式定义(如系统设置中的手机型号和Blur、Google设置向导中的机型等)**。更改后可自定义手机型号名称，供用户界面显示。

**ro.product.locale.language**

本参数定义了系统的初始(默认)语言。此处注意是语言，如中文是zh，英文是en。更改后改变系统**初次启动**时的语言设置。

**ro.product.locale.region**

本参数定义了系统的初始(默认)区域。此处注意是区域，如中国大陆为CN，台湾为TW，美国为US。更改后改变系统初次启动时的区域设置。

**ro.build.description和ro.build.fingerprint**

ROM的编译综合说明。其中包含了平台硬件、Android版本、源代码分支和标签、OTA详细版本等。其中的OTA部分，例如：umts\_jordan\_china-user 2.3.6 4.5.3-109\_DPP-141323416413release-keys将此数字与ro.build.version.incremental一同更改可避免OTA升级提醒(如Miui和Blur等)。

### 基本性能相关参数

**windowsmgr.max\_events\_per\_sec**

本参数定义了Android系统的窗体事件管理器在单位时间内可以处理的最大事件数量。通过更改本参数可以获得非常明显的丝滑流畅体验。可填写的值范围为“大于0的正整数”，官方默认为60。建议150、200、260、300这几个值。当此值变大时，系统触控平滑度明显提高，但对应的CPU使用率也会升高，最终的结果就是电池续航能力下降。以我个人的经验来说，此值取到240左右时在系统设置中滑动可以得到接近WP7的流畅和平滑度。

**ro.min\_pointer\_dur**

本参数定义了两次触摸之间的最短时间间隔，单位是毫秒。默认值为25，推荐值是10。通过调整此参数可以提高系统触控的灵敏度或稳定度。当此值越大时，触控越稳定。此值越小，触控越灵敏。

**mot.proximity.delay**

本参数定义了手机光纤感应器的抖动消除时间，单位是毫秒。默认值是500，推荐值是250。通过调整此参数可以提高在通话结束后屏幕点亮的速度。当此值越大时，通话结束后屏幕点亮所需要的时间越长，但在通话过程中如果手机意外瞬间离开脸部也不会点亮屏幕，可防止通话过程中的误操作(比方说通话时不小心手机移动了一下，屏幕就会点亮，此时如果脸部触碰到了屏幕就会对通话造成影响)。此值越小，则当手机离开脸部或装入口袋后会立即点亮或关闭屏幕。

**mot.proximity.distance**

本参数定义了手机屏幕上的两个触摸点之间的最短距离，若距离小于此值则认为是一个触摸点，单位是像素。默认值是60，推荐值是100。为什么推荐100呢？因为Defy的屏幕分辨率为480x854，也就是说横向有480个像素点，对应上去也就相当于是横向并排允许4个触摸点，平均一个手指一个点，这样在类似于杀西瓜等游戏中可以提升游戏操作。

**ro.kernel.android.checkjni**

本参数定义了Dalvik虚拟机在执行程序的时候是否要做Jni链接库的检查工作。详细见Dalvik参数属性期。若考虑稳定性可使用true，若需要性能可使用false。注意：此参数会被Dalvik参数覆盖。

**ro.media.enc.jpeg.quality**

本参数定义了JPEG图像编码器所使用的质量因子，可填写的值为1~100，默认为80，推荐为100。想照出更好的照片吗？想让照片的大小轻松上M吗？那就使用100吧。

**debug.sf.hw**

本参数定义了系统是否启用GPU来渲染程序的UI，默认为0，推荐为1。但要注意，如果此值为1，在某些应用程序中可能会出现显示错乱的现象(极少见)。

**persist.sys.use\_dithering**

本参数定义了系统渲染器对图像的缩放是否启用抖动技术。可填写的值为0或1。当开启抖动后，图像的显示(指背景、解锁等的图像，并非图库、相机那些的)会很柔和，但会增加CPU负载，最终导致ROM卡顿。

**persist.sys.purgeable\_assets**

本参数定义了系统是否可以清除暂时不用的数据以释放更多的RAM。可填写的值为0或1。当值为1时，系统会定期清理不用的数据以释放更多的RAM，同时作为代价就是下次启动程序或游戏加载数据会变慢。

**video.accelerate.hw**

本参数定义了系统是否对视频启用**硬件加速**功能。这里的视频指代屏幕上显示的东西，不仅仅是“电影视频”。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

**debug.performance.tuning**

本参数定义了系统是否针对性能做较调。可填写的值为0或1。需要注意的是：摩托官方的2.2和2.3系统对此功能支持的不是很好，开启后有时反而会降低系统流畅度。但CM系统绝对建议开启。

ro.HOME\_APP\_ADJ

ro.FOREGROUND\_APP\_ADJ

ro.VISIBLE\_APP\_ADJ

ro.PERCEPTIBLE\_APP\_ADJ

ro.HEAVY\_WEIGHT\_APP\_ADJ

ro.SECONDARY\_SERVER\_ADJ

ro.BACKUP\_APP\_ADJ

ro.HIDDEN\_APP\_MIN\_ADJ

ro.EMPTY\_APP\_ADJ

以上参数可填写的值为整数。这里只给出值的规律

0代表降低进程的优先级且驻留内存，1代表驻留内存，4代表缓存较多的内存，15代表尽量缓存内存。也就是说内存缓存器是按照ADJ从大到小来进行缓存的。大家可根据自系统中自己对各种应用程序的要求进行更改。

以下给出一个经典用例：

ro.FOREGROUND\_APP\_ADJ=0 前台程序驻留内存(不缓存)

ro.VISIBLE\_APP\_ADJ=1         可见的程序驻留内存(不缓存)

ro.PERCEPTIBLE\_APP\_ADJ=2    缓存的RAM多一些

ro.HOME\_APP\_ADJ=3         桌面程序，缓存的RAM稍多一些

ro.HEAVY\_WEIGHT\_APP\_ADJ=4        缓存的RAM再多一些

ro.SECONDARY\_SERVER\_ADJ=5        缓存的RAM再再多一些

ro.BACKUP\_APP\_ADJ=6       缓存的RAM再再再多一些

ro.HIDDEN\_APP\_MIN\_ADJ=7         隐藏的程序，根据程序的类型进行内存管理，最低为缓存的RAM再再再再多一些，最高就是直接缓存内存。

ro.EMPTY\_APP\_ADJ=15         已经退出的程序，直接缓存内存

ro.FOREGROUND\_APP\_MEM

ro.VISIBLE\_APP\_MEM

ro.PERCEPTIBLE\_APP\_MEM

ro.HEAVY\_WEIGHT\_APP\_MEM

ro.SECONDARY\_SERVER\_MEM

ro.BACKUP\_APP\_MEM

ro.HOME\_APP\_MEM

ro.HIDDEN\_APP\_MEM

ro.CONTENT\_PROVIDER\_MEM

ro.EMPTY\_APP\_MEM

以上参数定义了各种类型的应用程序在内存缓冲的大小，单位是页下面给出一个经典用例：

ro.FOREGROUND\_APP\_MEM=1280

ro.VISIBLE\_APP\_MEM=2560

ro.PERCEPTIBLE\_APP\_MEM=3840

ro.HEAVY\_WEIGHT\_APP\_MEM=6400

ro.SECONDARY\_SERVER\_MEM=7680

ro.BACKUP\_APP\_MEM=8960

ro.HOME\_APP\_MEM=5120

ro.HIDDEN\_APP\_MEM=12800

ro.CONTENT\_PROVIDER\_MEM=15360

ro.EMPTY\_APP\_MEM=20480

### 基本耗电相关参数

**wifi.supplicant\_scan\_interval**

本参数定义了Wifi扫描已保存节电的时间间隔。当点亮屏幕或打开Wifi时，系统会不停的扫描环境中是否存在已经保存的Wifi节点，当发现后则进行连接，而这个参数控制了每次扫描的时间间隔。单位是秒。取值范围是正整数。**官方默认为45，推荐180**。

**ro.mot.battmanager.wifictrl**

本参数定义了电源管理模块对Wifi的控制。默认为0。当此值为1时可以明显节电，但有时Wifi会出现不稳定的情况(不是所有ROM都如此)。

**ro.mot.deep.sleep.supported**

本参数定义了是否开启摩托的“休眠”模式。取值为true或false。当值为true时，在电源菜单中会出现“休眠”模式。此模式类似于电脑的睡眠，即将CPU等部件的电源全部关闭，只为**RAM供电**以保存休眠前的系统状态。耗电量比完全关机多一些，但可以做到**瞬间开机**。仅在官方ROM有效。

**pm.sleep\_mode**

本参数定义了系统待机时的睡眠深度，在所有Android系统上有效。取值范围是0~4，对应解释如下。0：强制关闭除RAM之外的所有部件，此状态下最省电。Defy几乎可以纯待机3~4个礼拜。但是此模式与“休眠”类似，一旦进入之后射频也会关闭，手机的2G/3G信号也就断了(语音和数据)。

1：让ARM进入中断触发的待机(超低功耗)模式。与模式0相比，本模式下射频不会关闭，而ARM可以通过软件(闹铃)和硬件(来电)中断来唤醒，因此耗电方面远大于模式0，Defy可以纯待机7天(不安装任何软件)。非常建议使用。  
2：将所有应用程序挂起到后台。与模式1相比，本模式下硬件几乎不参与多少节电，耗电自然比模式1多很多。当应用程序被挂起后，CPU的负载会大幅度降低，从而节电。此模式下Defy纯待机5天。  
3：将CPU的频率和电压降至最低，低到主频只有几十MHz的水平，而此时CPU接受外部中断(通过中断来恢复频率和电压)。与模式2相比，本模式下CPU通过降频和降压参与了节电，因此本模式的耗电比模式2多了一点。Defy纯待机约4~5天。本模式也是官方ROM和官方CM系统的默认值。  
4：CPU接受外部中断。与上述4个模式相比，此模式下几乎不做任何节电，只是关闭了屏幕和按键背光而已。Defy纯待机约2天。将上述5个模式的节电按照星级来分就是，模式0和1为5颗星，模式2和3为3颗星，模式4为1颗星。综上所属就是，模式0和模式1基本一样，是靠完全关闭几乎所有硬件部件来进行节电，省电效果最佳。模式2和模式3是靠调节CPU频率来进行节电。个人强烈推荐采用pm.sleep\_mode=1，即省电又稳定。如果想用模式0但又担心基带射频的同学可以继续往下看，解决办法在下面。

**ro.ril.disable.power.collapse**

本参数定义了是否禁止射频参与电源休眠。取值是0或1。这个参数的使用需要与上一个参数相匹配(我看到很多ROM中的这两个参数都是不匹配的，**最终造成的效果就是点亮屏幕后信号存在问题**)。当本参数为1的时候即射频永远打开，为0的时候根据上一个参数pm.sleep\_mode来判断是否关闭射频。永远打开射频必然费电，但是如果射频关闭，那手机就没信号了。那么当pm.sleep\_mode=0的时候，上面说过，此时待机会关闭几乎所有硬件部件，包括射频。而此时如果ro.ril.disable.power.collapse=1，就会保持射频的开启(即使进入休眠模式也一样)。这样即使待机，手机也有信号。但是又存在这样一个现象，在有些ROM中pm.sleep\_mode=0会带来更多的问题， **如睡死、亮屏后Wifi打不开、蓝牙打不开等**。因此建议同学们可以先尝试一下pm.sleep\_mode=0和ro.ril.disable.power.collapse=1组合使用，看看是否有bug，如果没有那自然使用此种模式，毕竟最省电了(极端省电)。对于稳定与省电兼得，可使用如下组合：pm.sleep\_mode=1ro.ril.disable.power.collapse=0这样射频在pm.sleep\_mode=1下不会被关闭，而进入休眠模式后射频会关闭。

## 源码分析

**JAVA**：\frameworks\base\core\[**Java**](http://lib.csdn.net/base/java)\[**android**](http://lib.csdn.net/base/android)\os\ SystemProperties.java：get()—》native\_get

**JIN**:\frameworks\base\core\jni\android\_os\_SystemProperties.cpp: SystemProperties\_getSS-> property\_get

**操作在\bionic\libc\bionic\system\_properties.c中：**

|  |
| --- |
| int \_\_system\_property\_get(const char \*name, char \*value)  {    //数据已经存储在内存中\_\_system\_property\_area\_\_ 等待读取完返回  const prop\_info \*pi = \_\_system\_property\_find(name);  return \_\_system\_property\_read(pi, 0, value);  } |

进程启动后数据已经将系统属性数据读取到相应的共享内存中，保存在全局变量\_\_system\_property\_area\_\_

### **设置属性**异步socket通信

|  |
| --- |
| int \_\_system\_property\_set(const char \*key, const char \*value)  {       msg.cmd = PROP\_MSG\_SETPROP;     strlcpy(msg.name, key, sizeof msg.name);     strlcpy(msg.value, value, sizeof msg.value);     err = send\_prop\_msg(&msg);  }  static int send\_prop\_msg(prop\_msg \*msg)  {    //sokcet 通信 /dev/socket/property\_service     s = socket(AF\_LOCAL, SOCK\_STREAM, 0);     connect(s, (struct sockaddr \*) &addr, alen)     send(s, msg, sizeof(prop\_msg), 0)     close(s);  } |

通过socket向property\_service发送消息，property\_service运行在哪里呢？

### Property Service创建服务端socket

**init进程启动监听过程中：**，**Property Service 是运行在init守护进程中。**

|  |
| --- |
| \system\core\init\Init.c  int main(int argc, char \*\*argv)  {     //加入到action queue队列  queue\_builtin\_action(property\_service\_init\_action, "property\_service\_init");   for(;;)  //执行action queue队列  //接收通过socket向property service 发送的数据;         nr = poll(ufds, fd\_count, timeout);         ……         handle\_property\_set\_fd();  }  static int property\_service\_init\_action(int nargs, char \*\*args)  {     start\_property\_service();  }  \system\core\init\property\_service.c：  void start\_property\_service(void)  {    //加载属性配置文件      load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_BUILD);     load\_properties\_from\_file(PROP\_PATH\_SYSTEM\_DEFAULT);     load\_properties\_from\_file(PROP\_PATH\_LOCAL\_OVERRIDE);     load\_persistent\_properties();     //创建socket资源 并绑定     fd = create\_socket(PROP\_SERVICE\_NAME, SOCK\_STREAM, 0666, 0, 0);    //监听     listen(fd, 8);  } |

### Property Service 监听socket处理

|  |
| --- |
| //\system\core\init\property\_service.c：  void handle\_property\_set\_fd()  {    //等待建立通信     s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)    //获取套接字相关信息 uid gid     getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size);    //接收属性设置请求消息     recv(s, &msg, sizeof(msg), 0);    //处理消息     switch(msg.cmd) {    case PROP\_MSG\_SETPROP:        //通过设置系统属性  处理ctl.开头消息         if(memcmp(msg.name,"ctl.",4) == 0)         {            //权限检测             if (check\_control\_perms(msg.value, cr.uid, cr.gid))             {                 handle\_control\_message((char\*) msg.name + 4, (char\*) msg.value);             }         } else         {            //更改系统属性值             if (check\_perms(msg.name, cr.uid, cr.gid))             {                 property\_set((char\*) msg.name, (char\*) msg.value);             }         }    break;     }     close(s);  } |

### 设置属性

|  |
| --- |
| **int** property\_set(**const char** \*name, **const char** \*value)  {  prop\_info \*pi;  **int** ret;   size\_t namelen = strlen(name);  size\_t valuelen = strlen(value);   **if** (!is\_legal\_property\_name(name, namelen)) **return** -1;  **if** (valuelen >= PROP\_VALUE\_MAX) **return** -1;   pi = (prop\_info\*) \_\_system\_property\_find(name);   **if**(pi != 0) {  */\* ro.\* properties may NEVER be modified once set \*/* **if**(!strncmp(name, **"ro."**, 3)) **return** -1;   \_\_system\_property\_update(pi, value, valuelen);  } **else** {  ret = \_\_system\_property\_add(name, namelen, value, valuelen);  **if** (ret < 0) {  ERROR(**"Failed to set '%s'='%s'\n"**, name, value);  **return** ret;  }  }  */\* If name starts with "net." treat as a DNS property. \*/* **if** (strncmp(**"net."**, name, strlen(**"net."**)) == 0) {  **if** (strcmp(**"net.change"**, name) == 0) {  **return** 0;  }  */\*  \* The 'net.change' property is a special property used track when any  \* 'net.\*' property name is updated. It is \_ONLY\_ updated here. Its value  \* contains the last updated 'net.\*' property.  \*/* property\_set(**"net.change"**, name);  } **else if** (persistent\_properties\_loaded &&  strncmp(**"persist."**, name, strlen(**"persist."**)) == 0) {  */\*  \* Don't write properties to disk until after we have read all default properties  \* to prevent them from being overwritten by default values.  \*/* write\_persistent\_property(name, value);  } **else if** (strcmp(**"selinux.reload\_policy"**, name) == 0 &&  strcmp(**"1"**, value) == 0) {  selinux\_reload\_policy();  }  property\_changed(name, value);  **return** 0;  }   **void** handle\_property\_set\_fd()  {  prop\_msg msg;  **int** s;  **int** r;  **int** res;  struct ucred cr;  struct sockaddr\_un addr;  socklen\_t addr\_size = sizeof(addr);  socklen\_t cr\_size = sizeof(cr);  **char** \* source\_ctx = NULL;  struct pollfd ufds[1];  **const int** timeout\_ms = 2 \* 1000; */\* Default 2 sec timeout for caller to send property. \*/* **int** nr;   **if** ((s = accept(property\_set\_fd, (struct sockaddr \*) &addr, &addr\_size)) < 0) {  **return**;  }   */\* Check socket options here \*/* **if** (getsockopt(s, SOL\_SOCKET, SO\_PEERCRED, &cr, &cr\_size) < 0) {  close(s);  ERROR(**"Unable to receive socket options\n"**);  **return**;  }   ufds[0].fd = s;  ufds[0].events = POLLIN;  ufds[0].revents = 0;  nr = TEMP\_FAILURE\_RETRY(poll(ufds, 1, timeout\_ms));  **if** (nr == 0) {  ERROR(**"sys\_prop: timeout waiting for uid=%d to send property message.\n"**, cr.uid);  close(s);  **return**;  } **else if** (nr < 0) {  ERROR(**"sys\_prop: error waiting for uid=%d to send property message. err=%d %s\n"**, cr.uid, errno, strerror(errno));  close(s);  **return**;  }   r = TEMP\_FAILURE\_RETRY(recv(s, &msg, sizeof(msg), MSG\_DONTWAIT));  **if**(r != sizeof(prop\_msg)) {  ERROR(**"sys\_prop: mis-match msg size received: %d expected: %zu errno: %d\n"**,  r, sizeof(prop\_msg), errno);  close(s);  **return**;  }   **switch**(msg.cmd) {  **case** PROP\_MSG\_SETPROP:  msg.name[PROP\_NAME\_MAX-1] = 0;  msg.value[PROP\_VALUE\_MAX-1] = 0;   **if** (!is\_legal\_property\_name(msg.name, strlen(msg.name))) {  ERROR(**"sys\_prop: illegal property name. Got: \"%s\"\n"**, msg.name);  close(s);  **return**;  }   getpeercon(s, &source\_ctx);   **if**(memcmp(msg.name,**"ctl."**,4) == 0) {  *// Keep the old close-socket-early behavior when handling  // ctl.\* properties.* close(s);  **if** (check\_control\_mac\_perms(msg.value, source\_ctx)) {  handle\_control\_message((**char**\*) msg.name + 4, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n"**,  msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);  }  } **else** {  **if** (check\_perms(msg.name, source\_ctx)) {  property\_set((**char**\*) msg.name, (**char**\*) msg.value);  } **else** {  ERROR(**"sys\_prop: permission denied uid:%d name:%s\n"**,  cr.uid, msg.name);  }   *// Note: bionic's property client code assumes that the  // property server will not close the socket until \*AFTER\*  // the property is written to memory.* close(s);  }  freecon(source\_ctx);  **break**;  **default**:  close(s);  **break**;  }  } |

**修改系统属性权限表：**

property\_perms[] = {

   { "net.dns",          AID\_RADIO,    0 },

   { "net.",             AID\_SYSTEM,   0 },

   { "dev.",             AID\_SYSTEM,   0 },

   { "runtime.",         AID\_SYSTEM,   0 },

   { "sys.",             AID\_SYSTEM,   0 },

   { "service.",         AID\_SYSTEM,   0 },

   { "persist.sys.",     AID\_SYSTEM,   0 },

   { "persist.service.", AID\_SYSTEM,   0 },

   ……

    { NULL, 0, 0 }

};

一般property启动应该加在init.<your hardware>.rc而不是直接init.rc里。下面是一个init.rc里的例子：

### 开关服务

可以通过设置系统属性 改变服务的执行状态 start/stop：如果想要应用有权限启动/关闭某Native Service：需要具有system/root权限，找到对应应用uid gid，将应用名称加入到control\_perms列表中。

连着前面就是ctr.start和ctr.stop系统属性：用来启动和停止服务的。

|  |
| --- |
| void handle\_control\_message(const char \*msg, const char \*arg)  {    if (!strcmp(msg,"start")) {         msg\_start(arg);     } else if (!strcmp(msg,"stop")) {         msg\_stop(arg);     } else if (!strcmp(msg,"restart")) {         msg\_stop(arg);         msg\_start(arg);     }  }static void msg\_start(const char \*name)  {     service\_start(svc, args);  }void service\_start(struct service \*svc, const char \*dynamic\_args){    //创建进程启动服务     pid = fork();     execve(svc->args[0], (char\*\*) svc->args, (char\*\*) ENV);     //修改服务的系统属性 执行状态     notify\_service\_state(svc->name, "running");  } |

例如：　// start boot animation

　　　　property\_set("ctl.start", "bootanim");

定义开机启动的服务:**启动服务的时候会判断：**

static void service\_start\_if\_not\_disabled(struct service \*svc)

{        //判断是否启动

       if (!(svc->flags & SVC\_DISABLED)) {

           service\_start(svc, NULL);

       }

}

在init.rc中表明服务是否在开机时启动：

service adbd /sbin/adbd

　　　　class core

　　　　disabled //不自动启动

## 属性系统设计

　Property Service运行在init进程中，开机从属性文件中加载到共享内存中；设置系统属性通过socket与Property Service通信。

　　Property Consumer进程将存储系统属性值的共享内存，加载到当前进程虚拟空间中，实现对系统属性值的读取。

Property Setter进程修改系统属性，通过socket向Property Service发送消息，更改系统属性值。

属性系统设计的关键就是：跨进程共享内存的实现。

[详情看](http://blog.csdn.net/hecong_kit/article/details/46008895)

## 小结

属性初始化的入口点是property\_init ，在system/core/init/property\_service.c中定义。它的主要工作是申请32k共享内存，其中前1k是属性区的头，后面31k可以存247个属性（受前1k头的限制）。property\_init初始化完property以后，加载/default.prop的属性定义。   
  
其它的系统属性（build.prop, local.prop,…）在start\_property\_service中加载。加载完属性服务创建一个socket和其他进程通信（设置或读取属性）。   
  
Init进程poll属性的socket，等待和处理属性请求。如果有请求到来，则调用handle\_property\_set\_fd来处理这个请求。在这个函数里，首先检查请求者的uid/gid看看是否有权限，如果有权限则调property\_service.c中的property\_set函数。   
  
在property\_set函数中，它先查找就没有这个属性，如果找到，更改属性。如果找不到，则添加新属性。更改时还会判断是不是“ro”属性，如果是，则不能更改。如果是persist的话还会写到/data/property/<name>中。   
  
最后它会调property\_changed，把事件挂到队列里，如果有人注册这个属性的话（比如init.rc中on property:ro.kernel.qemu=1），最终会调它的会调函数。   
  
property名字长度限制是32字节，值的限制是92字节。不知道是google怎么想的 — 一般都是名字比值长得多！比如[dalvik.vm.heapgrowthlimit]: [48m]

## Ref

# 系统升级

升级方式：SD卡、usb、OTA升级

（以OTA升级方式梳理整个升级流程）

在setting中触发升级按钮，检测prop中的服务器路径上的版本是否比本地版本高，如果低于或者等于本地版本，则提示已是最新版本，如果高于本地版本，则提示有新版本是否进行升级，点击立即升级。

bootCommand（）此函数在main system重启之前做准备工作。首先创建/cache/recovery/目录，删除该目录下的command和log（可能不存在）在sqlite数据库中的备份，然后将上一步的arg命令写入到/cache/recovery/command文件中，下一步就真正重启了。

源码路径：frameworks/base/core/java/android/os/RecoverySystem.java

BCB：bootloader control block

arg就是进入recovery模式后，recovery服务需要进行的操作

RecoverySystem类中的**installPackage(Context context,FilepackageFile)**函数根据传进来的包文件获取包的绝对路径filename，然后将其拼成**arg=“--update\_package=”+filename**，它最终会被传到BCB中，然后被传到**bootCommand(context,arg)。**

升级包在此处也被传进来了

mService是SystemUpdateService实例，跟踪发现SystemUpdateService.Java中的rebootAndUpdate函数中新建了一个线程，此此线程调用了**RecoverySystem.installPackage(mContext,mFile)**

按钮监听事件，调用mService.rebootAndUpdate（new  File(mFile)）。此处的file即为升级包

跟踪file来源，mService.getInstallFile() 获取到OTA在线下载的文件路径

注：从Bootloader开始如果没有组合键按下，就从MISC分区读取BCB块的command域，在主系统时已经将“boot-recovery”写入。然后进入recovery模式开始启动，与正常的启动系统类似，也有启动内核，然后启动文件系统，在进入文件系统后会执行/init，init的配置文件就是/init.rc。（bootable/recovery/etc/init.rc）

主要工作：

1. 设置环境变量
2. 建立etc连接
3. 新建目录，备用
4. 挂载/tmp为内存文件系统tmpfs
5. 启动recovery（/sbin/recovery）服务
6. 启动adbd服务（用于调试）

（\*其中最重要的就是启动recovery服务）

源码路径：bionic/libc/unistd/reboot.c

此函数将”recovery”传过去，之后将”boot-recovery”写入到BCB数据块的command域中，重启之后就知道进入recover

进入reboot：

调用pm.reboot(“recovery”)，跟踪发现其实是一个系统调用\_\_reboot(LINUX\_REBOOT\_MAGIC1,LINUX\_REBOOT\_MAGIC2,mode,NULL)，前两个参数代表组合键，mode代表传过来的”recovery”

## Recovery流程

Recovery的内容有：恢复出厂设置、OTA升级、使能/关闭加密文件系统

（下图是recovery整个流程，很详细



1、ui\_init():在recovery的过程中主要用于显示一个背景图片（正在安装或者安装失败）和一个进度条（用于显示进度）,另外还启动了两个线程，一个用于处理进度条的显示（progress\_thread），另一个用于响应用户的按键（input\_thread）。

2、 get\_arg()：这个函数主要做了上图中get\_arg()往右往下直到parse arg/v的工作。对照着流程看。

      ①get\_bootloader\_message()：主要工作是根据分区的文件格式类型（mtd或emmc）从MISC分区中读取BCB数据块到一个临时的变量中。

      ②然后开始判断Recovery服务是否有带命令行的参数（/sbin/recovery，根据现有的逻辑是没有的），若没有就从BCB中读取recovery域。如果读取失败则从/cache/recovery/command中读取然后。这样这个BCB的临时变量中的recovery域就被更新了。在将这个BCB的临时变量写回真实的BCB之前，又更新的这个BCB临时变量的command域为“boot-recovery”。这样做的目的是如果在升级失败（比如升级还未结束就断电了）时，系统在重启之后还会进入Recovery模式，直到升级完成。

      ③在这个BCB临时变量的各个域都更新完成后使用set\_bootloader\_message()写回到真正的BCB块中。

                这个过程可以用一个简单的图来概括，这样更清晰：



  3、parserargc/argv：解析我们获得参数。注册所解析的命令（register\_update\_command），在下面的操作中会根据这一步解析的值进行一步步的判断，然后进行相应的操作。

   4、 if(update\_package)：判断update\_package是否有值，若有就表示需要升级更新包，此时就会调用install\_package()（即图中红色的第二个阶段）。在这一步中将要完成安装实际的升级包。

     5、 if(wipe\_data/wipe\_cache)：这一步判断实际是两步，在源码中是先判断是否擦除data分区（用户数据部分）的，然后再判断是否擦除cache分区。值得注意的是在擦除data分区的时候必须连带擦除cache分区。在只擦除cache分区的情形下可以不擦除data分区。

     6、maybe\_install\_firmware\_update()：如果升级包中包含/radio/hboot firmware的更新，则会调用这个函数。查看源码发现，在注释中（OTA INSTALL）有这一个流程。但是main函数中并没有显示调用这个函数。目前尚未发现到底是在什么地方处理。但是其流程还是向上面的图示一样。即，① 先向BCB中写入“boot-recovery”和“—wipe\_cache”之后将cache分区格式化，然后将firmware image 写入原始的cache分区中。②将命令“update-radio/hboot”和“—wipe\_cache”写入BCB中，然后开始重新安装firmware并刷新firmware。③之后又会进入图示中的末尾，即finish\_recovery()。

     7、 prompt\_and\_wait()：这个函数是在一个判断中被调用的。其意义是如果安装失败（update.zip包错误或验证签名失败），则等待用户的输入处理（如通过组合键reboot等）。

     8、 finish\_recovery()：这是Recovery关闭并进入Main System的必经之路。其大体流程如下：



 ① 将intent（字符串）的内容作为参数传进finish\_recovery中。如果有intent需要告知Main System，则将其写入/cache/recovery/intent中。这个intent的作用尚不知有何用。

       ② 将内存文件系统中的Recovery服务的日志（/tmp/recovery.log）拷贝到cache（/cache/recovery/log）分区中，以便告知重启后的Main System发生过什么。

       ③ 擦除MISC分区中的BCB数据块的内容，以便系统重启后不在进入Recovery模式而是进入更新后的主系统。

        ④ 删除/cache/recovery/command文件。这一步也是很重要的，因为重启后Bootloader会自动检索这个文件，如果未删除的话又会进入Recovery模式。原理在上面已经讲的很清楚了。

9、 reboot()：这是一个系统调用。在这一步Recovery完成其服务重启并进入Main System。这次重启和在主系统中重启进入Recovery模式调用的函数是一样的，但是其方向是不一样的。所以参数也就不一样。查看源码发现，其重启模式是RB\_AUTOBOOT。这是一个系统的宏。

## Install\_package详解

源码路径：/bootable/recovery/install.c



①ensure\_path\_mount()：先判断所传的update.zip包路径所在的分区是否已经挂载。如果没有则先挂载。

②load\_keys()：加载公钥源文件，路径位于/res/keys。这个文件在Recovery镜像的根文件系统中。

 ③verify\_file()：对升级包update.zip包进行签名验证。

 ④mzOpenZipArchive()：打开升级包，并将相关的信息拷贝到一个临时的ZipArchinve变量中。这一步并未对我们的update.zip包解压。

  ⑤try\_update\_binary()：在这个函数中才是对我们的update.zip升级的地方。这个函数一开始先根据我们上一步获得的zip包信息，以及升级包的绝对路径将update\_binary文件拷贝到内存文件系统的/tmp/update\_binary中。以便后面使用。

   ⑥pipe()：创建管道，用于下面的子进程和父进程之间的通信。

   ⑦fork()：创建子进程。其中的子进程主要负责执行binary（execv(binary,args)，即执行我们的安装命令脚本），父进程负责接受子进程发送的命令去更新ui显示（显示当前的进度）。子父进程间通信依靠管道。

   ⑧其中，在创建子进程后，父进程有两个作用。一是通过管道接受子进程发送的命令来更新UI显示。二是等待子进程退出并返回INSTALL SUCCESS。其中子进程在解析执行安装脚本的同时所发送的命令有以下几种：

     progress  <frac> <secs>：根据第二个参数secs（秒）来设置进度条。

     set\_progress  <frac>：直接设置进度条，frac取值在0.0到0.1之间。

     firmware <”hboot”|”radio”><filename>：升级firmware时使用，在API  V3中不再使用。

      ui\_print <string>：在屏幕上显示字符串，即打印更新过程。

      execv(binary,args)的作用就是去执行binary程序，这个程序的实质就是去解析update.zip包中的updater-script脚本中的命令并执行。由此，Recovery服务就进入了实际安装update.zip包的过程。

|  |
| --- |
|  |

# 出厂设置

<http://blog.csdn.net/mirkerson/article/details/24322207>

<http://blog.csdn.net/woshing123456/article/details/44524051>

# PowerManagerService

第5章 深入理解 PowerManagerService

<http://wiki.jikexueyuan.com/project/deep-android-v2/powermanagerservice.html>

# [Android5.1--PowerManagerService电源管理](http://blog.csdn.net/kitty_landon/article/details/47107045)

<http://blog.csdn.net/kitty_landon/article/details/47107045>

# [Android5.1 PowerManagerService深入分析](http://blog.csdn.net/kc58236582/article/details/45537749)

<http://blog.csdn.net/kc58236582/article/details/45537749>

、、、、、、、、、、、、、、、、、、、、、、、、、

[Android5.1AlarmManagerService深入分析（Android4.4补充）](http://blog.csdn.net/kc58236582/article/details/45666709)

<http://blog.csdn.net/kc58236582/article/details/45666709>

# Android之AlarmManagerService源码分析

<http://www.procedurego.com/article/128389.html>

<http://blog.csdn.net/zhangyongfeiyong/article/details/52224300>

<https://github.com/clarkehe/Android/wiki/Android%E4%BC%91%E7%9C%A0%E6%9C%BA%E5%88%B6>

休眠机制

# [AlarmManager 实现闹钟的基本功能](http://www.cnblogs.com/rainly/p/3437627.html)

## Ref

# Path

编译生成的apk路径：out/target/product/ maguro/system/app

签名文件目录：/build/target/product/security

Android系统应用的源码路径：/packages/apps目录或其子目录中

源码目录/package/apps/PackageInstaller

# Task

Settings.Global

Settings源码解读

Wifi热点的ip的查询

Root和非root区别是什么？是否存在root也无法修改的build.prop

# QA

1. adb install –r A.apk 执行原理？tmp目录作用，INSTALL\_FAILED\_ILLEGITIMATE\_APK各种安装报错原因，cnt的作用，为啥可以。
2. 系统目录结构是啥
3. App和系统签名的原理是什么？
4. 在sws下正常运行as，执行一次系统签名呢

# 参考

[Android系统应用的开发和测试](http://www.epubit.com.cn/book/onlinechapter/30644)

# [Android 源代码目录结构1 - bionic](http://blog.csdn.net/kickxxx/article/details/6927272)

# [Android init.rc文件解析过程详解(一)](http://blog.csdn.net/mk1111/article/details/16357327)