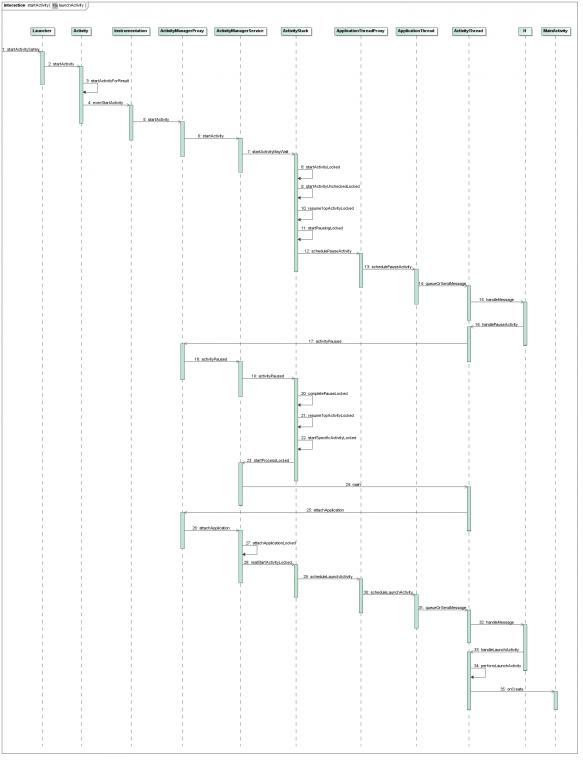
**[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748" \o "Android应用程序启动过程源代码分析)**

分类： [Android](http://blog.csdn.net/Luoshengyang/article/category/838604) 2011-08-19 00:58 5447人阅读 [评论](http://blog.csdn.net/luoshengyang/article/details/6689748#comments)(40) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/luoshengyang/article/details/6689748#report)

        前文简要介绍了Android应用程序的Activity的启动过程。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。

        在上一篇文章[Android应用程序的Activity启动过程简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6685853)中，我们举例子说明了启动Android应用程序中的Activity的两种情景，其中，在手机屏幕中点击应用程序图标的情景就会引发Android应用程序中的默认Activity的启动，从而把应用程序启动起来。这种启动方式的特点是会启动一个新的进程来加载相应的Activity。这里，我们继续以这个例子为例来说明Android应用程序的启动过程，即MainActivity的启动过程。

        MainActivity的启动过程如下图所示：



[点击查看大图](http://hi.csdn.net/attachment/201108/18/0_1313675675dBp4.gif)

        下面详细分析每一步是如何实现的。

        Step 1. Launcher.startActivitySafely

        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。

        Launcher的源代码工程在packages/apps/Launcher2目录下，负责启动其它应用程序的源代码实现在src/com/android/launcher2/Launcher.java文件中：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. /\*\*
2. \* Default launcher application.
3. \*/
4. public final class Launcher extends Activity
5. implements View.OnClickListener, OnLongClickListener, LauncherModel.Callbacks, AllAppsView.Watcher {
7. ......
9. /\*\*
10. \* Launches the intent referred by the clicked shortcut.
11. \*
12. \* @param v The view representing the clicked shortcut.
13. \*/
14. public void onClick(View v) {
15. Object tag = v.getTag();
16. if (tag instanceof ShortcutInfo) {
17. // Open shortcut
18. final Intent intent = ((ShortcutInfo) tag).intent;
19. int[] pos = new int[2];
20. v.getLocationOnScreen(pos);
21. intent.setSourceBounds(new Rect(pos[0], pos[1],
22. pos[0] + v.getWidth(), pos[1] + v.getHeight()));
23. startActivitySafely(intent, tag);
24. } else if (tag instanceof FolderInfo) {
25. ......
26. } else if (v == mHandleView) {
27. ......
28. }
29. }
31. void startActivitySafely(Intent intent, Object tag) {
32. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
33. try {
34. startActivity(intent);
35. } catch (ActivityNotFoundException e) {
36. ......
37. } catch (SecurityException e) {
38. ......
39. }
40. }
42. ......
44. }

回忆一下前面一篇文章[Android应用程序的Activity启动过程简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6685853)说到的应用程序Activity，它的默认Activity是MainActivity，这里是AndroidManifest.xml文件中配置的：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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。事实上，Task是一个非常复杂的概念，有兴趣的读者可以到官网<http://developer.android.com/guide/topics/manifest/activity-element.html>查看相关的资料。这里，我们只要知道，这个MainActivity要在一个新的Task中启动就可以了。

        Step 2. Activity.startActivity

        在Step 1中，我们看到，Launcher继承于Activity类，而Activity类实现了startActivity函数，因此，这里就调用了Activity.startActivity函数，它实现在frameworks/base/core/java/android/app/Activity.java文件中：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class Activity extends ContextThemeWrapper
2. implements LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
6. ......
8. @Override
9. public void startActivity(Intent intent) {
10. startActivityForResult(intent, -1);
11. }
13. ......
15. }

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。

        Step 3. Activity.startActivityForResult

        这个函数也是实现在frameworks/base/core/java/android/app/Activity.java文件中：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class Activity extends ContextThemeWrapper
2. implements LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
6. ......
8. public void startActivityForResult(Intent intent, int requestCode) {
9. if (mParent == null) {
10. Instrumentation.ActivityResult ar =
11. mInstrumentation.execStartActivity(
12. this, mMainThread.getApplicationThread(), mToken, this,
13. intent, requestCode);
14. ......
15. } else {
16. ......
17. }
18. ......
19. }

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，我们在[Android系统在新进程中启动自定义服务过程（startService）的原理分析](http://blog.csdn.net/luoshengyang/article/details/6677029)一文中已经介绍过了。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。（mMainThread怎么来的？）

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class Instrumentation {
3. ......
5. public ActivityResult execStartActivity(
6. Context who, IBinder contextThread, IBinder token, Activity target,
7. Intent intent, int requestCode) {
8. IApplicationThread whoThread = (IApplicationThread) contextThread;
9. if (mActivityMonitors != null) {
10. ......
11. }
12. try {
13. int result = ActivityManagerNative.getDefault()
14. .startActivity(whoThread, intent,
15. intent.resolveTypeIfNeeded(who.getContentResolver()),
16. null, 0, token, target != null ? target.mEmbeddedID : null,
17. requestCode, false, false);
18. ......
19. } catch (RemoteException e) {
20. }
21. return null;
22. }
23. ......
24. }

这里的ActivityManagerNative.getDefault返回ActivityManagerService的远程接口，即ActivityManagerProxy（定义在ActivityManagerNative.java文件里）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](http://blog.csdn.net/luoshengyang/article/details/6677029)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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. class ActivityManagerProxy implements IActivityManager
2. {
4. ......
6. public int startActivity(IApplicationThread caller, Intent intent,
7. String resolvedType, Uri[] grantedUriPermissions, int grantedMode,
8. IBinder resultTo, String resultWho,
9. int requestCode, boolean onlyIfNeeded,
10. boolean debug) throws RemoteException {
11. Parcel data = Parcel.obtain();
12. Parcel reply = Parcel.obtain();
13. data.writeInterfaceToken(IActivityManager.descriptor);
14. data.writeStrongBinder(caller != null ? caller.asBinder() : null);
15. intent.writeToParcel(data, 0);
16. data.writeString(resolvedType);
17. data.writeTypedArray(grantedUriPermissions, 0);
18. data.writeInt(grantedMode);
19. data.writeStrongBinder(resultTo);
20. data.writeString(resultWho);
21. data.writeInt(requestCode);
22. data.writeInt(onlyIfNeeded ? 1 : 0);
23. data.writeInt(debug ? 1 : 0);
24. mRemote.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);
25. reply.readException();
26. int result = reply.readInt();
27. reply.recycle();
28. data.recycle();
29. return result;
30. }
32. ......
34. }

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。

        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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityManagerService extends ActivityManagerNative
2. implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. public final int startActivity(IApplicationThread caller,
7. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
8. int grantedMode, IBinder resultTo,
9. String resultWho, int requestCode, boolean onlyIfNeeded,
10. boolean debug) {
11. return mMainStack.startActivityMayWait(caller, intent, resolvedType,
12. grantedUriPermissions, grantedMode, resultTo, resultWho,
13. requestCode, onlyIfNeeded, debug, null, null);
14. }

17. ......
19. }

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。

        Step 7. ActivityStack.startActivityMayWait

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. final int startActivityMayWait(IApplicationThread caller,
6. Intent intent, String resolvedType, Uri[] grantedUriPermissions,
7. int grantedMode, IBinder resultTo,
8. String resultWho, int requestCode, boolean onlyIfNeeded,
9. boolean debug, WaitResult outResult, Configuration config) {
11. ......
13. boolean componentSpecified = intent.getComponent() != null;
15. // Don't modify the client's object!
16. intent = new Intent(intent);
18. // Collect information about the target of the Intent.
19. ActivityInfo aInfo;
20. try {
21. ResolveInfo rInfo =
22. AppGlobals.getPackageManager().resolveIntent(
23. intent, resolvedType,
24. PackageManager.MATCH\_DEFAULT\_ONLY
25. | ActivityManagerService.STOCK\_PM\_FLAGS);
26. aInfo = rInfo != null ? rInfo.activityInfo : null;  //这里可以打印log，观察aInfo.applicationInfo.packageName和aInfo.name的值，与要启动的Activity的信息是否一致。
27. } catch (RemoteException e) {
28. ......
29. }
31. if (aInfo != null) {
32. // Store the found target back into the intent, because now that
33. // we have it we never want to do this again.  For example, if the
34. // user navigates back to this point in the history, we should
35. // always restart the exact same activity.
36. intent.setComponent(new ComponentName(
37. aInfo.applicationInfo.packageName, aInfo.name));
38. ......
39. }
41. synchronized (mService) {
42. int callingPid;
43. int callingUid;
44. if (caller == null) {
45. ......
46. } else {
47. callingPid = callingUid = -1;
48. }
50. mConfigWillChange = config != null
51. && mService.mConfiguration.diff(config) != 0;
53. ......
55. if (mMainStack && aInfo != null &&
56. (aInfo.applicationInfo.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0) {
58. ......
60. }
62. int res = startActivityLocked(caller, intent, resolvedType,
63. grantedUriPermissions, grantedMode, aInfo,
64. resultTo, resultWho, requestCode, callingPid, callingUid,
65. onlyIfNeeded, componentSpecified);
67. if (mConfigWillChange && mMainStack) {
68. ......
69. }
71. ......
73. if (outResult != null) {
74. ......
75. }
77. return res;
78. }
80. }
82. ......
84. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. ActivityInfo aInfo;
2. try {
3. ResolveInfo rInfo =
4. AppGlobals.getPackageManager().resolveIntent(
5. intent, resolvedType,
6. PackageManager.MATCH\_DEFAULT\_ONLY
7. | ActivityManagerService.STOCK\_PM\_FLAGS);
8. aInfo = rInfo != null ? rInfo.activityInfo : null;
9. } catch (RemoteException e) {
10. ......
11. }

解析之后，得到的aInfo.applicationInfo.packageName的值为"shy.luo.activity"，aInfo.name的值为"shy.luo.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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. final int startActivityLocked(IApplicationThread caller,
6. Intent intent, String resolvedType,
7. Uri[] grantedUriPermissions,
8. int grantedMode, ActivityInfo aInfo, IBinder resultTo,
9. String resultWho, int requestCode,
10. int callingPid, int callingUid, boolean onlyIfNeeded,
11. boolean componentSpecified) {
12. int err = START\_SUCCESS;
14. ProcessRecord callerApp = null;
15. if (caller != null) {
16. callerApp = mService.getRecordForAppLocked(caller); //caller是Launcher所在的线程
17. if (callerApp != null) {
18. callingPid = callerApp.pid;
19. callingUid = callerApp.info.uid;
20. } else {
21. ......
22. }
23. }
25. ......
27. ActivityRecord sourceRecord = null;
28. ActivityRecord resultRecord = null;
29. if (resultTo != null) {
30. int index = indexOfTokenLocked(resultTo);
32. ......
34. if (index >= 0) {
35. sourceRecord = (ActivityRecord)mHistory.get(index);
36. if (requestCode >= 0 && !sourceRecord.finishing) {
37. ......
38. }
39. }
40. }
42. int launchFlags = intent.getFlags();
44. if ((launchFlags&Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0
45. && sourceRecord != null) {
46. ......
47. }
49. if (err == START\_SUCCESS && intent.getComponent() == null) {
50. ......
51. }
53. if (err == START\_SUCCESS && aInfo == null) {
54. ......
55. }
57. if (err != START\_SUCCESS) {
58. ......
59. }
61. ......
63. ActivityRecord r = new ActivityRecord(mService, this, callerApp, callingUid,
64. intent, resolvedType, aInfo, mService.mConfiguration,
65. resultRecord, resultWho, requestCode, componentSpecified);
67. ......
69. return startActivityUncheckedLocked(r, sourceRecord,
70. grantedUriPermissions, grantedMode, onlyIfNeeded, true);
71. }

74. ......
76. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. final int startActivityUncheckedLocked(ActivityRecord r,
6. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
7. int grantedMode, boolean onlyIfNeeded, boolean doResume) {
8. final Intent intent = r.intent;
9. final int callingUid = r.launchedFromUid;
11. int launchFlags = intent.getFlags();
13. // We'll invoke onUserLeaving before onPause only if the launching
14. // activity did not explicitly state that this is an automated launch.
15. mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;
17. ......
19. ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)
20. != 0 ? r : null;
22. // If the onlyIfNeeded flag is set, then we can do this if the activity
23. // being launched is the same as the one making the call...  or, as
24. // a special case, if we do not know the caller then we count the
25. // current top activity as the caller.
26. if (onlyIfNeeded) {
27. ......
28. }
30. if (sourceRecord == null) {
31. ......
32. } else if (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
33. ......
34. } else if (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
35. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {
36. ......
37. }
39. if (r.resultTo != null && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
40. ......
41. }
43. boolean addingToTask = false;
44. if (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
45. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
46. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
47. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
48. // If bring to front is requested, and no result is requested, and
49. // we can find a task that was started with this same
50. // component, then instead of launching bring that one to the front.
51. if (r.resultTo == null) {
52. // See if there is a task to bring to the front.  If this is
53. // a SINGLE\_INSTANCE activity, there can be one and only one
54. // instance of it in the history, and it is always in its own
55. // unique task, so we do a special search.
56. ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
57. ? findTaskLocked(intent, r.info)
58. : findActivityLocked(intent, r.info);
59. if (taskTop != null) {
60. ......
61. }
62. }
63. }
65. ......
67. if (r.packageName != null) {
68. // If the activity being launched is the same as the one currently
69. // at the top, then we need to check if it should only be launched
70. // once.
71. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
72. if (top != null && r.resultTo == null) {
73. if (top.realActivity.equals(r.realActivity)) {
74. ......
75. }
76. }
78. } else {
79. ......
80. }
82. boolean newTask = false;
84. // Should this be considered a new task?
85. if (r.resultTo == null && !addingToTask
86. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
87. // todo: should do better management of integers.
88. mService.mCurTask++;
89. if (mService.mCurTask <= 0) {
90. mService.mCurTask = 1;
91. }
92. r.task = new TaskRecord(mService.mCurTask, r.info, intent,
93. (r.info.flags&ActivityInfo.FLAG\_CLEAR\_TASK\_ON\_LAUNCH) != 0);
94. ......
95. newTask = true;
96. if (mMainStack) {
97. mService.addRecentTaskLocked(r.task);
98. }
100. } else if (sourceRecord != null) {
101. ......
102. } else {
103. ......
104. }
106. ......
108. startActivityLocked(r, newTask, doResume);
109. return START\_SUCCESS;
110. }
112. ......
114. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. private final void startActivityLocked(ActivityRecord r, boolean newTask,
6. boolean doResume) {
7. final int NH = mHistory.size();
9. int addPos = -1;
11. if (!newTask) {
12. ......
13. }
15. // Place a new activity at top of stack, so it is next to interact
16. // with the user.
17. if (addPos < 0) {
18. addPos = NH;
19. }
21. // If we are not placing the new activity frontmost, we do not want
22. // to deliver the onUserLeaving callback to the actual frontmost
23. // activity
24. if (addPos < NH) {
25. ......
26. }
28. // Slot the activity into the history stack and proceed   //这里将要启动的Activity放到了堆栈中
29. mHistory.add(addPos, r);
30. r.inHistory = true;
31. r.frontOfTask = newTask;
32. r.task.numActivities++;
33. if (NH > 0) {
34. // We want to show the starting preview window if we are
35. // switching to a new task, or the next activity's process is
36. // not currently running.
37. ......
38. } else {
39. // If this is the first activity, don't do any fancy animations,
40. // because there is nothing for it to animate on top of.
41. ......
42. }
44. ......
46. if (doResume) {
47. resumeTopActivityLocked(null);
48. }
49. }
51. ......
53. }

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，要做一些任务切换的界面操作，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. Activity.resumeTopActivityLocked

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. /\*\*
6. \* Ensure that the top activity in the stack is resumed.
7. \*
8. \* @param prev The previously resumed activity, for when in the process
9. \* of pausing; can be null to call from elsewhere.
10. \*
11. \* @return Returns true if something is being resumed, or false if
12. \* nothing happened.
13. \*/
14. final boolean resumeTopActivityLocked(ActivityRecord prev) {
15. // Find the first activity that is not finishing.
16. ActivityRecord next = topRunningActivityLocked(null);
18. // Remember how we'll process this pause/resume situation, and ensure
19. // that the state is reset however we wind up proceeding.
20. final boolean userLeaving = mUserLeaving;
21. mUserLeaving = false;
23. if (next == null) {
24. ......
25. }
27. next.delayedResume = false;
29. // If the top activity is the resumed one, nothing to do.
30. if (mResumedActivity == next && next.state == ActivityState.RESUMED) {
31. ......
32. }
34. // If we are sleeping, and there is no resumed activity, and the top
35. // activity is paused, well that is the state we want.
36. if ((mService.mSleeping || mService.mShuttingDown)
37. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
38. ......
39. }
41. ......
43. // If we are currently pausing an activity, then don't do anything
44. // until that is done.
45. if (mPausingActivity != null) {
46. ......
47. }
49. ......
51. // We need to start pausing the current activity so the top one
52. // can be resumed...
53. if (mResumedActivity != null) {
54. ......
55. startPausingLocked(userLeaving, false);
56. return true;
57. }
59. ......
60. }
62. ......
64. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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. }

它首先看要启动的Activity是否就是当前处于Resumed状态的Activity，如果是的话，那就什么都不用做，直接返回就可以了；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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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. ......
11. mResumedActivity = null;
12. mPausingActivity = prev;
13. mLastPausedActivity = prev;
14. prev.state = ActivityState.PAUSING;
15. ......
17. if (prev.app != null && prev.app.thread != null) {
18. ......
19. try {
20. ......
21. prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
22. prev.configChangeFlags);
23. ......
24. } catch (Exception e) {
25. ......
26. }
27. } else {
28. ......
29. }
31. ......
33. }
35. ......
37. }

        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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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。

        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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityThread {
3. ......
5. private final class ApplicationThread extends ApplicationThreadNative {
7. ......
9. public final void schedulePauseActivity(IBinder token, boolean finished,
10. boolean userLeaving, int configChanges) {
11. queueOrSendMessage(
12. finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,
13. token,
14. (userLeaving ? 1 : 0),
15. configChanges);
16. }
18. ......
20. }
22. ......
24. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityThread {
3. ......
5. private final void queueOrSendMessage(int what, Object obj, int arg1) {
6. queueOrSendMessage(what, obj, arg1, 0);
7. }
9. private final void queueOrSendMessage(int what, Object obj, int arg1, int arg2) {
10. synchronized (this) {
11. ......
12. Message msg = Message.obtain();
13. msg.what = what;
14. msg.obj = obj;
15. msg.arg1 = arg1;
16. msg.arg2 = arg2;
17. mH.sendMessage(msg);
18. }
19. }
21. ......
23. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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) {
13. ......
15. case PAUSE\_ACTIVITY:
16. handlePauseActivity((IBinder)msg.obj, false, msg.arg1 != 0, msg.arg2);
17. maybeSnapshot();
18. break;
20. ......
22. }
23. ......
25. }
27. ......
29. }

        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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityThread {
3. ......
5. private final void handlePauseActivity(IBinder token, boolean finished,
6. boolean userLeaving, int configChanges) {
8. ActivityClientRecord r = mActivities.get(token);  //这里获取的是Launcher的R
9. if (r != null) {
10. //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
11. if (userLeaving) {
12. performUserLeavingActivity(r);
13. }
15. r.activity.mConfigChangeFlags |= configChanges;
16. Bundle state = performPauseActivity(token, finished, true);
18. // Make sure any pending writes are now committed.
19. QueuedWork.waitToFinish();
21. // Tell the activity manager we have paused.
22. try {
23. ActivityManagerNative.getDefault().activityPaused(token, state);
24. } catch (RemoteException ex) {
25. }
26. }
27. }
29. ......
31. }

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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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。

        Step 18. ActivityManagerService.activityPaused

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityManagerService extends ActivityManagerNative
2. implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
3. ......
5. public final void activityPaused(IBinder token, Bundle icicle) {
7. ......
9. final long origId = Binder.clearCallingIdentity();
10. mMainStack.activityPaused(token, icicle, false);  //这里的token代表Launcher
12. ......
13. }
15. ......
17. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. final void activityPaused(IBinder token, Bundle icicle, boolean timeout) {
7. ......
9. ActivityRecord r = null;
11. synchronized (mService) {
12. int index = indexOfTokenLocked(token);
13. if (index >= 0) {
14. r = (ActivityRecord)mHistory.get(index);
15. if (!timeout) {
16. r.icicle = icicle;
17. r.haveState = true;
18. }
19. mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);
20. if (mPausingActivity == r) {
21. r.state = ActivityState.PAUSED;
22. completePauseLocked();
23. } else {
24. ......
25. }
26. }
27. }
28. }
30. ......
32. }

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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. private final void completePauseLocked() {
6. ActivityRecord prev = mPausingActivity;
8. ......
10. if (prev != null) {
12. ......
14. mPausingActivity = null;
15. }
17. if (!mService.mSleeping && !mService.mShuttingDown) {
18. resumeTopActivityLocked(prev);
19. } else {
20. ......
21. }
23. ......
24. }
26. ......
28. }

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. final boolean resumeTopActivityLocked(ActivityRecord prev) {
6. ......
8. // Find the first activity that is not finishing.
9. ActivityRecord next = topRunningActivityLocked(null);  //这里的next就是我们要启动的activity了
11. // Remember how we'll process this pause/resume situation, and ensure
12. // that the state is reset however we wind up proceeding.
13. final boolean userLeaving = mUserLeaving;
14. mUserLeaving = false;
16. ......
18. next.delayedResume = false;
20. // If the top activity is the resumed one, nothing to do.
21. if (mResumedActivity == next && next.state == ActivityState.RESUMED) {
22. ......
23. return false;
24. }
26. // If we are sleeping, and there is no resumed activity, and the top
27. // activity is paused, well that is the state we want.
28. if ((mService.mSleeping || mService.mShuttingDown)
29. && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
30. ......
31. return false;
32. }
34. .......

37. // We need to start pausing the current activity so the top one
38. // can be resumed...
39. if (mResumedActivity != null) {
40. ......
41. return true;
42. }
44. ......

47. if (next.app != null && next.app.thread != null) {
48. ......
50. } else {
51. ......
52. startSpecificActivityLocked(next, true, true);
53. }
55. return true;
56. }

59. ......
61. }

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public class ActivityStack {
3. ......
5. private final void startSpecificActivityLocked(ActivityRecord r,  //r即我们创建的Activity
6. boolean andResume, boolean checkConfig) {
7. // Is this activity's application already running?
8. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
9. r.info.applicationInfo.uid);
11. ......
13. if (app != null && app.thread != null) {
14. try {
15. realStartActivityLocked(r, app, andResume, checkConfig);
16. return;
17. } catch (RemoteException e) {
18. ......
19. }
20. }
22. mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
23. "activity", r.intent.getComponent(), false);
24. }

27. ......
29. }

注意，这里由于是第一次启动应用程序的Activity，所以下面语句：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
2. 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"shy.luo.activity"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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityManagerService extends ActivityManagerNative
2. implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. final ProcessRecord startProcessLocked(String processName,
7. ApplicationInfo info, boolean knownToBeDead, int intentFlags,
8. String hostingType, ComponentName hostingName, boolean allowWhileBooting) {
10. ProcessRecord app = getProcessRecordLocked(processName, info.uid);
12. ......
14. String hostingNameStr = hostingName != null
15. ? hostingName.flattenToShortString() : null;
17. ......
19. if (app == null) {
20. app = new ProcessRecordLocked(null, info, processName);
21. mProcessNames.put(processName, info.uid, app);
22. } else {
23. // If this is a new package in the process, add the package to the list
24. app.addPackage(info.packageName);
25. }
27. ......
29. startProcessLocked(app, hostingType, hostingNameStr);
30. return (app.pid != 0) ? app : null;
31. }
33. ......
35. }

     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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityManagerService extends ActivityManagerNative
2. implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. private final void startProcessLocked(ProcessRecord app,
7. String hostingType, String hostingNameStr) {
9. ......
11. try {
12. int uid = app.info.uid;
13. int[] gids = null;
14. try {
15. gids = mContext.getPackageManager().getPackageGids(
16. app.info.packageName);
17. } catch (PackageManager.NameNotFoundException e) {
18. ......
19. }
21. ......
23. int debugFlags = 0;
25. ......
27. int pid = Process.start("android.app.ActivityThread",
28. mSimpleProcessManagement ? app.processName : null, uid, uid,
29. gids, debugFlags, null);  //这里的processName默认是MainActivity所在的包名
31. ......
33. } catch (RuntimeException e) {
35. ......
37. }
38. }
40. ......
42. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityThread {
3. ......
5. private final void attach(boolean system) {
6. ......
8. mSystemThread = system;
9. if (!system) {
11. ......
13. IActivityManager mgr = ActivityManagerNative.getDefault();
14. try {
15. mgr.attachApplication(mAppThread);
16. } catch (RemoteException ex) {
17. }
18. } else {
20. ......
22. }
23. }
25. ......
27. public static final void main(String[] args) {
29. .......
31. ActivityThread thread = new ActivityThread();
32. thread.attach(false);
34. ......
36. Looper.loop();
38. .......
40. thread.detach();
42. ......
43. }
44. }

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. class ActivityManagerProxy implements IActivityManager
2. {
3. ......
5. public void attachApplication(IApplicationThread app) throws RemoteException
6. {
7. Parcel data = Parcel.obtain();
8. Parcel reply = Parcel.obtain();
9. data.writeInterfaceToken(IActivityManager.descriptor);
10. data.writeStrongBinder(app.asBinder());
11. mRemote.transact(ATTACH\_APPLICATION\_TRANSACTION, data, reply, 0);
12. reply.readException();
13. data.recycle();
14. reply.recycle();
15. }
17. ......
19. }

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

       Step 26. ActivityManagerService.attachApplication

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. public final class ActivityManagerService extends ActivityManagerNative
2. implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
4. ......
6. public final void attachApplication(IApplicationThread thread) {
7. synchronized (this) {
8. int callingPid = Binder.getCallingPid();
9. final long origId = Binder.clearCallingIdentity();
10. attachApplicationLocked(thread, callingPid);
11. Binder.restoreCallingIdentity(origId);
12. }
13. }
15. ......
17. }

这里将操作转发给attachApplicationLocked函数。

        Step 27. ActivityManagerService.attachApplicationLocked

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. class ApplicationThreadProxy implements IApplicationThread {
3. ......
5. public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
6. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
7. List<Intent> pendingNewIntents, boolean notResumed, boolean isForward)
8. throws RemoteException {
9. Parcel data = Parcel.obtain();
10. data.writeInterfaceToken(IApplicationThread.descriptor);
11. intent.writeToParcel(data, 0);
12. data.writeStrongBinder(token);
13. data.writeInt(ident);
14. info.writeToParcel(data, 0);
15. data.writeBundle(state);
16. data.writeTypedList(pendingResults);
17. data.writeTypedList(pendingNewIntents);
18. data.writeInt(notResumed ? 1 : 0);
19. data.writeInt(isForward ? 1 : 0);
20. mRemote.transact(SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION, data, null,
21. IBinder.FLAG\_ONEWAY);
22. data.recycle();
23. }
25. ......
27. }

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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.activityInfo.targetActivity);
17. }

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

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

1. mInstrumentation.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

      这个函数定义在packages/experimental/Activity/src/shy/luo/activity/MainActivity.java文件中，这是我们自定义的app工程文件：

[view plaincopy to clipboardprint?](http://blog.csdn.net/luoshengyang/article/details/6689748)

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. ......
14. }

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。