PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

# Activity、Task、应用和进程

很想弄清楚启动一个 Activity 和 Task(任务)的关系，网上也有很多相关资料，由从源码来具体分析的，也有针对 launchmode 来分析，但都不是自己的，理解起来总不是那么容易，为此，尝试着自己去理解其中的逻辑。不过事先需要弄清楚两个问题：

## 谁负责管理 Activity？

Android 的 framework 框架中采用 C/S 的方式实现由后台服务 ActivityManagerService（很多书上将其简称为 Ams）来管理具体的 Acitivity 实例，虽然 Ams 命名为 ActivityManagerService，但是它其实不仅仅管理 Activity，他还管理除 Acitvity 外的其它三大组件，还包括 Intent、pendingintent、apk 进程和 task 等等，具体可以查看源码 frameworks\base\services\java\com\android\server\am 下面的 Ams 源码和相关的*Record 类。

网上资料显示在 2.3 以后的 SDK 中，Ams 将原来的 HistoryRecord 类重新命名为 ActivityRecord 类，并将其原来 Ams 中的一些处理抽出来成为一个 ActivityStack 类，每一个 Activity 实例对应一个 ActivityRecord 对象，并存放在 ActivityStack 中，手机一旦启动，就会启动一个 Ams 服务，在 Ams 服务中有一个 ActivityStack 实例专门管理手机上的 ActivityRecord 实例。这样一来，不具体分析源码，仅从架构角度来说谁负责管理 Activity 的问题就清晰多了。

## Task 到底是什么？

根据对上面问题的解答，既然 Activity 是由 Ams 通过 ActivityStack 来管理的，那么这个 Task 又是干什么的呢？

以往基于应用（application）的程序开发中，程序具有明确的边界，一个程序就是一个应用，一个应用为了实现功能可以采用开辟新线程甚至新进程来辅助，但是应用与应用之间不能复用资源和功能。而 Android 引入了基于组件开发的软件架构，虽然我们开发 android 程序，仍然使用一个 apk 工程一个 Application 的开发形式，但是对于 Aplication 的开发就用到了 Activity、service 等四大组件，其中的每一个组件，都是可以被跨应用复用的哦，这个就是 android 的神奇之处。

另外值得一提的是，虽然组件可以跨应用被调用，但是一个组件所在的进程必须是在组件所在的 Aplication 进程中。由于 android 强化了组件概念，弱化了 Aplication 的概念，所以在 android 程序开发中，A 应用的 A 组件想要使用拍照或录像的功能就可以不用去针对 Camera 类进行开发，直接调用系统自带的摄像头应用（称其 B 应用）中的组件（称其 B 组件）就可以了，但是这就引发了一个新问题，A 组件跑在 A 应用中，B 组件跑在 B 应用中，自然都不在同一个进程中，那么从 B 组件中返回的时候，如何实现正确返回到 A 组件呢？Task 就是来负责实现这个功能的，它是从用户角度来理解应用而建立的一个抽象概念。因为用户所能看到的组件就是 Activity，所以 Task 可以理解为实现一个功能而负责管理所有用到的 Activity 实例的栈。

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

其实查看源码，在 Ams 内部，并不是真的有这么一个存放 Activity 的 Task 栈，Activity 还是通过 ActivityStack 来管理，在 ActivityRecord 中有一个 TaskRecord 对象记录了真实的 Activity 实例是属于哪个 Task 的。Task 通过一个 int 类型的 TaskId 来唯一标识，该值在手机重启时将会被置零。

说了这么多，还是找一个 Task 任务最直观的体现吧。先重启手机，长按 home 键，发现弹出的最近任务中一个任务也没有，然后开启 A 应用，长按 home 键，会发现有一个 A 应用的任务，查看手机进程，应该还没有 B 进程的；在 A 应用的 A 组件中调 B 应用的 B 组件，此时看手机的进程，除了 A 进程外，还有个 B 的进程，但是长按 home 键，能看到的还是只有一个 A 应用的任务。其实这个时候，B 应用已经跑起来了，但是对用户来说，他其实没有开启过 B 应用，所以 Task 任务自始至终都是从用户的角度出发而设计的概念，保证用户的调用逻辑。

## Activity、Application 与进程的关系

理清楚了上述两个概念问题，进入 Activity 的启动与 Task 的关系之前，让我们先来了解下启动一个 Activity 与 apk Application 和进程的关系。对 Activity 启动过程具体的分析，工程耗时很庞大，网上有个老罗整理的三篇博客，我看了半天还是云里雾里的，有兴趣的可以查看如下链接：http://blog.csdn.net/luoshengyang/article/details/6685853。我通过我现在的认识和目前通过 Demo 的测试来看，启动一个 Activity 时，Ams 首先会去查询该 Activity 所在的应用是否开启，如果没有开启则会启动一个进程去 Run 这个 Application，因此无论通过 Launcher 还是通过常规的程序内部调用 startActivity 来启动一个 Activity，所要启动的 Activity 都是跑在其注册的 apk 所在的 Application 进程中（或者该组件 android:process 指定的进程中），而其 TaskID 一般是和启动它的组件所属的 TaskId 一直，但是也不尽然，这就要看下面的具体分析了。

另外通过对 service 的应用，可以得出结论，一个 apk，即一个应用（Application）可以跑在多个进程中，一个进程在一个虚拟机中运行，也即一个 apk 可以启动多个虚拟机。

通过 shareuserID 可以将多个 apk，跑在同一个进程中。

从而得出结论：一个虚拟机只能跑一个进程，一个进程里可以跑多个应用，一个应用也可以跑在多个进程中，这就是他们的关系。

## Activity 和 Task 的关系

启动一个 Activity 有两种方式，一种就是通过 Launcher，另外一种是通过程序代码调用 startActivity 函数实现（验证过 AppWidget 其实与这种方式是一致的）其实影响 Activity 启动关键点大致有三个因素：Activity 注册信息中的 launchMode、启动 Activity 时 Intent 中的 launchFlags 和 Activity 注册信息中的 taskAffinity、allowTaskReparenting、clearTaskOnLaunch、alwaysRetainTaskState、finishOnTaskLaunch 等信息。通常情况下，我们只需要针对第一个因素进行合理设置就能满足我们应用开发的需求了。

launchMode 的四种方式

launchMode 方法是在 apk 的 manifest 文件中针对每一个 Activity 的 android:launchMode 属性进行设置的方式，共有四种模式可以设置，分别是 standard、singleTop、singleTask、singleInstance，下面分别阐述之（由于其中几种因素有设置时，会影响 lauchMode 的四种模式，所以下面情况下，其它因素都是缺省不设置的情况）。

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

standard

standard 是默认模式，即假设用户在 manifest 中对 Activity 不指定 android:launchMode 的情况下，缺省启动模式即为 standard。在启动一个以 standard 为 launchMode 的 Activity 时，Ams 只是简单的创建一个新的 Activity 实例，将其放到 ActivityStack（为了行文方便，后面将 ActivityStack 简称 AS），其 TaskId 则与启动这个 Activity 的调用者 Activity 相同（即就算创建一个新的进程，但是其 TaskId 还是跟调用者一致的，以确保用户回退操作时保持一致）。这种是最常见的使用方式。

singleTop

启动一个以 singleTop 为 lauchMode 的 Activity 时，Ams 会查询 AS：假如在 AS 顶端正是要启动的 Activity 实例，那么 Ams 就不会重新启动一个 Activity 实例，而是调用 AS 栈顶的该 Activity 实例的 OnNewIntent 函数（自然不会修改原来的 TaskId 值）；假如在 AS 栈顶不是该 Activity 的实例，那么就会创建一个新的实例，将其压入 AS，其 TaskId 与调用者 Activity 相同。这种方式主要用于避免自调自过程中，产生多个实例的情况。

singleTask

启动一个以 singleTask 为 lauchMode 的 Activity 时，Ams 会查询 AS：如果 AS 内有一个该类 Activity 的实例，那么就会将该实例置于 TS 的顶端（原来位于该实例上面的其它同 TaskId 的 activity 实例，将被销毁），并调用该实例的 onNewIntent 函数；如果 AS 内没有该类的实例，就会启动一个新的实例，将其压入 AS，其 TaskID 与启动它的调用者没有必然关系，而是取决于该 Activity 所在 apk 进程是否有 TaskId，假如没有就会创建一个新的 TaskId。在实测中发现，如果 singleTask 模式启动的 Activity 是 AS 中同 TaskId 的最底部一个（或被称谓 Task 栈的根实例），那么在通过桌面长按，在近期任务中跳转到 Activity 所在的任务时，即使该 Activity 实例不是在栈顶，也会被置到栈顶（还会调用其 onNewIntent 函数），并将 AS 上同 TaskId 的其它 Activity 实例销毁，具体可以通过附带的 demo 来验证，其中 TaskOne 中的 Activity1 置成 singleTask 启动模式，其它均为默认的 standard，其 log 输出如下：

```
//在 TaskTwo 中启动 Activity1,此前 TaskOne apk 没有在运行，所以启动 Activity1 时会申请一个新的 TaskId，2afcfbd8 这个 Activity1 成为了 TaskId 为 53 的 Task 栈栈底实例
04-05 16:21:27.144: E/ActivityB @2b003230(17933): onPause pid 17933 taskid 52
04-05 16:21:27.404: E/Activity1 @2afcfbd8(18489): onCreate pid 18489 taskid 53
04-05 16:21:29.214: E/ActivityB @2afecaf0(17933): onCreate pid 17933 taskid 53
04-05 16:21:30.254: E/Activity2 @2afdeef0(18489): onCreate pid 18489 taskid 53
04-05 16:21:30.894: E/ActivityB @2aff9328(17933): onCreate pid 17933 taskid 53
04-05 16:21:31.454: E/Activity2 @2afe9330(18489): onCreate pid 18489 taskid 53
04-05 16:21:31.864: E/ActivityB @2b009910(17933): onCreate pid 17933 taskid 53
04-05 16:21:32.424: E/Activity2 @2aff2570(18489): onCreate pid 18489 taskid 53

//通过长按跳转到 TaskOne 任务，此时在 AS 中 id 为 53 的在 Activity1 上面有 6 个
04-05 16:22:35.144: E/ActivityB @2afecaf0(17933): onDestroy pid 17933 taskid 53
04-05 16:22:35.144: E/Activity2 @2afdeef0(18489): onDestroy pid 18489 taskid 53
04-05 16:22:35.234: E/ActivityB @2aff9328(17933): onDestroy pid 17933 taskid 53
04-05 16:22:35.254: E/Activity2 @2afe9330(18489): onDestroy pid 18489 taskid 53
04-05 16:22:35.324: E/ActivityB @2b009910(17933): onDestroy pid 17933 taskid 53
04-05 16:22:35.344: E/Activity2 @2aff2570(18489): onPause pid 18489 taskid 53
04-05 16:22:35.394: E/Activity1 @2afcfbd8(18489): onNewIntent pid 18489 taskid 53
04-05 16:22:35.394: E/Activity1 @2afcfbd8(18489): onStart pid 18489 taskid 53
```

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

```
04-05 16:22:35.394: E/Activity1 @2afcfbd8(18489): onResume pid 18489 taskid 53
04-05 16:22:35.524: E/Activity2 @2aff2570(18489): onStop pid 18489 taskid 53
04-05 16:22:35.524: E/Activity2 @2aff2570(18489): onDestroy pid 18489 taskid 53
```

singleInstance

启动一个以 singleInstance 为 launchmode 的 Activity 时，假如 AS 中已经有一个该类实例，那么调用其 onNewIntent 函数；否则就会创建一个新的 TaskId，与该 Activity 所在的 apk 进程完全不同的 TaskId，而且这个 TaskId 值以后也不会被用于其他任何 Activity 实例中。

简单小结

一般我们开发普通的应用程序时，我们只需要使用缺省的 standard 和 singTop 方式就够用了，不需要使用 singleTask 和 singleInstance 来声明注册的 Activity，因为它将破坏用户感觉上的回退操作，给用户使用上带来迷惑，所以一般将这两者用于很耗资源的 Activity，通过查看源码发现在源码 packages\apps 中的程序，有如下一些应用使用了这两者高级设置

```
AndroidManifest.xml (packages\apps\browser): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\calendar): <activity android:name="AlertActivity" android:launchMode="singleInstance"
AndroidManifest.xml (packages\apps\contacts): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\deskclock): android:launchMode="singleInstance"
AndroidManifest.xml (packages\apps\deskclock): android:launchMode="singleInstance"
AndroidManifest.xml (packages\apps\deskclock): android:launchMode="singleInstance"
AndroidManifest.xml (packages\apps\email): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\launcher2): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\music): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\phone): android:launchMode="singleInstance"
AndroidManifest.xml (packages\apps\phone): android:launchMode="singleInstance">
AndroidManifest.xml (packages\apps\quicksearchbox): android:launchMode="singleTask"
AndroidManifest.xml (packages\apps\settings): android:launchMode="singleTask"
AndroidManifest.xml (packages\providers\downloadprovider): android:launchMode="singleTask"
```

在上述分析中没有将其它两个因素引入，主要是自己在这方面接触的也比较少，而且这三方面因素共同组合会产生很多种不同效果，所以就没做具体展开。转载一些网友收集的资料如下：

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

# 跟 Task 有关的 manifest 文件中 Activity 的特性值介绍

android:allowTaskReparenting

用来标记 Activity 能否从启动的 Task 移动到有着 affinity 的 Task（当这个 Task 进入到前台时）

“true”，表示能移动，“false”，表示它必须呆在启动时呆在的那个 Task 里。

如果这个特性没有被设定，设定到<application>元素上的 allowTaskReparenting 特性的值会应用到 Activity 上。默认值为“false”。

一般来说，当 Activity 启动后，它就与启动它的 Task 关联，并且在那里耗尽它的整个生命周期。当当前的 Task 不再显示时，你可以使用这个特性来强制 Activity 移动到有着 affinity 的 Task 中。典型用法是：把一个应用程序的 Activity 移到另一个应用程序的主 Task 中。

例如，如果 email 中包含一个 web 页的链接，点击它就会启动一个 Activity 来显示这个页面。这个 Activity 是由 Browser 应用程序定义的，但是，现在它作为 email Task 的一部分。如果它重新宿主到 Browser Task 里，当 Browser 下一次进入到前台时，它就能被看见，并且，当 email Task 再次进入前台时，就看不到它了。

Actvity 的 affinity 是由 taskAffinity 特性定义的。Task 的 affinity 是通过读取根 Activity 的 affinity 决定。因此，根 Activity 总是位于相同 affinity 的 Task 里。由于启动模式为“singleTask”和“singleInstance”的 Activity 只能位于 Task 的底部，因此，重新宿主只能限于“standard”和“singleTop”模式。

android:alwaysRetainTaskState

用来标记 Activity 所在的 Task 的状态是否总是由系统来保持。

“true”，表示总是；“false”，表示在某种情形下允许系统恢复 Task 到它的初始化状态。默认值是“false”。

这个特性只针对 Task 的根 Activity 有意义；对其它 Activity 来说，忽略之。

一般来说，特定的情形如当用户从主画面重新选择这个 Task 时，系统会对这个 Task 进行清理（从 stack 中删除位于根 Activity 之上的所有 Activivity）。典型的情况，当用户有一段时间没有访问这个 Task 时也会这么做，例如 30 分钟。

然而，当这个特性设为“true”时，用户总是能回到这个 Task 的最新状态，无论他们是如何启动的。这非常有用，例如，像 Browser 应用程序，这里有很多的状态（例如多个打开的 Tab），用户不想丢失这些状态。

android:clearTaskOnLaunch

用来标记是否从 Task 中清除所有的 Activity，除了根 Activity 外（每当从主画面重新启动时）

“true”，表示总是清除至它的根 Activity，“false”表示不。默认值是“false”。

这个特性只对启动一个新的 Task 的 Activity（根 Activity）有意义；对 Task 中其它的 Activity 忽略。

当这个值为“true”，每次用户重新启动这个 Task 时，都会进入到它的根 Activity 中，不管这个 Task 最后在做些什么，也不管用户是使用 BACK 还是 HOME 离开的。当这个值为“false”时，可能会在一些情形下（参考 alwaysRetainTaskState 特性）清除 Task 的 Activity，但不总是。

假设，某人从主画面启动了 Activity P，并从那里迁移至 Activity Q。接下来用户按下 HOME，然后返回 Activity P。一般，用户可能见到的是 Activity Q，因为它是 P 的 Task

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

中最后工作的内容。然而，如果 P 设定这个特性为“ true”，当用户按下 HOME 并使这个 Task 再次进入前台时，其上的所有的 Activity(在这里是 Q)都将被清除。因此，当返回到这个 Task 时，用户只能看到 P。

如果这个特性和 allowTaskReparenting 都设定为“ true”，那些能重新宿主的 Activity 会移动到共享 affinity 的 Task 中；剩下的 Activity 都将被抛弃，如上所述。

android:finishOnTaskLaunch

用来标记当用户再次启动它的 Task（在主画面选择这个 Task）时已经存在的 Activity 实例是否要关闭（结束）

“ true”，表示应该关闭，“ false”表示不关闭。默认值是“ false”。

如果这个特性和 allowTaskReparenting 都设定为“ true”，这个特性胜出。Activity 的 affinity 忽略。这个 Activity 不会重新宿主，但是会销毁。

android:launchMode

用于指示 Activity 如何启动。这里有四种模式，与 Intent 对象中的 Activity Flags（FLAG_ACTIVITY_*变量）共同作用，来决定 Activity 如何启动来处理 Intent。它们是：

"standard"

"singleTop"

"singleTask"

"singleInstance"

默认模式是“ standard”。

前面文章：“ Android 四种 Activity 的加载模式”已经详细描述,这里就不做描述了.

android:noHistory

用于标记当用户从 Activity 上离开并且它在屏幕上不再可见时 Activity 是否从 Activity stack 中清除并结束（调用 finish()方法）——“ true”，表示它应该关闭，“ false”，表示不需要。默认值是“ false”。

“ true”值意味着 Activity 不会留下历史痕迹。因为它不会在 Activity stack 的 Task 中保留，因此，用户不能返回它。

比如启用界面的就可以借用这个。

android:taskAffinity

这就是本文所描述的任务共用性。

Activity 为 Task 拥有的一个 affinity。拥有相同的 affinity 的 Activity 理论上属于相同的 Task（在用户的角度是相同的“应用程序”）。Task 的 affinity 是由它的根 Activity 决定的。

affinity 决定两件事情——Activity 重新宿主的 Task（参考 allowTaskReparenting 特性）和使用 FLAG_ACTIVITY_NEW_TASK 标志启动的 Activity 宿主的 Task。

默认情况，一个应用程序中的所有 Activity 都拥有相同的 affinity。捏可以设定这个特性来重组它们，甚至可以把不同应用程序中定义的 Activity 放置到相同的 Task 中。为了明确 Activity 不宿主特定的 Task，设定该特性为空的字符串。

如果这个特性没有设置，Activity 将从应用程序的设定那里继承下来（参考<application>元素的 taskAffinity 特性）。应用程序默认的 affinity 的名字是<manifest>元素中设定的 package 名。

## 跟 Task 有关的 Intent 对象中设置的 Flag

FLAG_ACTIVITY_BROUGHT_TO_FRONT

这个标志一般不是由程序代码设置的，如在 launchMode 中设置 singleTask 模式时

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

系统帮你设定。

FLAG_ACTIVITY_CLEAR_TOP

如果设置，并且这个 Activity 已经在当前的 Task 中运行，因此，不再是重新启动一个这个 Activity 的实例，而是在这个 Activity 上方的所有 Activity 都将关闭，然后这个 Intent 会作为一个新的 Intent 投递到老的 Activity（现在位于顶端）中。

例如，假设一个 Task 中包含这些 Activity：A，B，C，D。如果 D 调用了 startActivity()，并且包含一个指向 Activity B 的 Intent，那么，C 和 D 都将结束，然后 B 接收到这个 Intent，因此，目前 stack 的状况是：A，B。

上例中正在运行的 Activity B 既可以在 onNewIntent()中接收到这个新的 Intent，也可以把自己关闭然后重新启动来接收这个 Intent。如果它的启动模式声明为“multiple”(默认值)，并且你没有在这个 Intent 中设置 FLAG_ACTIVITY_SINGLE_TOP 标志，那么它将关闭然后重新创建；对于其它的启动模式，或者在这个 Intent 中设置 FLAG_ACTIVITY_SINGLE_TOP 标志，都将把这个 Intent 投递到当前这个实例的 onNewIntent()中。

这个启动模式还可以与 FLAG_ACTIVITY_NEW_TASK 结合起来使用：用于启动一个 Task 中的根 Activity，它会把那个 Task 中任何运行的实例带入前台，然后清除它直到根 Activity。这非常有用，例如，当从 Notification Manager 处启动一个 Activity。

FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET

如果设置，这将在 Task 的 Activity stack 中设置一个还原点，当 Task 恢复时，需要清理 Activity。也就是说，下一次 Task 带着 FLAG_ACTIVITY_RESET_TASK_IF_NEEDED 标记进入前台时（典型的操作是用户在主画面重启它），这个 Activity 和它之上的都将关闭，以至于用户不能再返回到它们，但是可以回到之前的 Activity。

这在你的程序有分割点的时候很有用。例如，一个 e-mail 应用程序可能有一个操作是查看一个附件，需要启动图片浏览 Activity 来显示。这个 Activity 应该作为 e-mail 应用程序 Task 的一部分，因为这是用户在这个 Task 中触发的操作。然而，当用户离开这个 Task，然后从主画面选择 e-mail app，我们可能希望回到查看的会话中，但不是查看图片附件，因为这让人困惑。通过在启动图片浏览时设定这个标志，浏览及其它启动的 Activity 在下次用户返回到 mail 程序时都将全部清除。

FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS

如果设置，新的 Activity 不会在最近启动的 Activity 的列表中保存。

FLAG_ACTIVITY_FORWARD_RESULT

如果设置，并且这个 Intent 用于从一个存在的 Activity 启动一个新的 Activity，那么，这个作为答复目标的 Activity 将会传到这个新的 Activity 中。这种方式下，新的 Activity 可以调用 setResult(int)，并且这个结果值将发送给那个作为答复目标的 Activity。

FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY

这个标志一般不由应用程序代码设置，如果这个 Activity 是从历史记录里启动的（常按 HOME 键），那么，系统会帮你设定。

FLAG_ACTIVITY_MULTIPLE_TASK

不要使用这个标志，除非你自己实现了应用程序启动器。与 FLAG_ACTIVITY_NEW_TASK 结合起来使用，可以禁用把已存的 Task 送入前台的行为。当设置时，新的 Task 总是会启动来处理 Intent，而不管这是是否已经有一个 Task 可以处理相同的事情。

由于默认的系统不包含图形 Task 管理功能，因此，你不应该使用这个标志，除非你提供给用户一种方式可以返回到已经启动的 Task。

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

如果 FLAG_ACTIVITY_NEW_TASK 标志没有设置，这个标志被忽略。

FLAG_ACTIVITY_NEW_TASK

如果设置，这个 Activity 会成为历史 stack 中一个新 Task 的开始。一个 Task（从启动它的 Activity 到下一个 Task 中的 Activity）定义了用户可以迁移的 Activity 原子组。Task 可以移动到前台和后台；在某个特定 Task 中的所有 Activity 总是保持相同的次序。

这个标志一般用于呈现“启动”类型的行为：它们提供用户一系列可以单独完成的事情，与启动它们的 Activity 完全无关。

使用这个标志，如果正在启动的 Activity 的 Task 已经在运行的话，那么，新的 Activity 将不会启动；代替的，当前 Task 会简单的移入前台。参考 FLAG_ACTIVITY_MULTIPLE_TASK 标志，可以禁用这一行为。

这个标志不能用于调用方对已经启动的 Activity 请求结果。

FLAG_ACTIVITY_NO_ANIMATION

如果在 Intent 中设置，并传递给 Context.startActivity()的话，这个标志将阻止系统进入下一个 Activity 时应用 Acitivity 迁移动画。这并不意味着动画将永不运行——如果另一个 Activity 在启动显示之前，没有指定这个标志，那么，动画将被应用。这个标志可以很好的用于执行一连串的操作，而动画被看作是更高一级的事件的驱动。

FLAG_ACTIVITY_NO_HISTORY

如果设置，新的 Activity 将不再历史 stack 中保留。用户一离开它，这个 Activity 就关闭了。这也可以通过设置 noHistory 特性。

FLAG_ACTIVITY_NO_USER_ACTION

如果设置，作为新启动的 Activity 进入前台时，这个标志将在 Activity 暂停之前阻止从最前方的 Activity 回调的 onUserLeaveHint()。

典型的，一个 Activity 可以依赖这个回调指明显式的用户动作引起的 Activity 移出后台。这个回调在 Activity 的生命周期中标记一个合适的点，并关闭一些 Notification。

如果一个 Activity 通过非用户驱动的事件，如来电或闹钟，启动的，这个标志也应该传递给 Context.startActivity，保证暂停的 Activity 不认为用户已经知晓其 Notification。

FLAG_ACTIVITY_PREVIOUS_IS_TOP

If set and this intent is being used to launch a new activity from an existing one, the current activity will not be counted as the top activity for deciding whether the new intent should be delivered to the top instead of starting a new one. The previous activity will be used as the top, with the assumption being that the current activity will finish itself immediately.

FLAG_ACTIVITY_REORDER_TO_FRONT

如果在 Intent 中设置，并传递给 Context.startActivity()，这个标志将引发已经运行的 Activity 移动到历史 stack 的顶端。

例如，假设一个 Task 由四个 Activity 组成：A,B,C,D。如果 D 调用 startActivity()来启动 Activity B，那么，B 会移动到历史 stack 的顶端，现在的次序变成 A,C,D,B。如果 FLAG_ACTIVITY_CLEAR_TOP 标志也设置的话，那么这个标志将被忽略。

FLAG_ACTIVITY_RESET_TASK_IF_NEEDED

If set, and this activity is either being started in a new task or bringing to the top an existing task, then it will be launched as the front door of the task. This will result in the application of any affinities needed to have that task in the proper state (either moving activities to or from it), or simply resetting that task to its initial state if needed.

FLAG_ACTIVITY_SINGLE_TOP

如果设置，当这个 Activity 位于历史 stack 的顶端运行时，不再启动一个新的。