[**https://blog.csdn.net/woshimalingyi/article/details/66477437**](https://blog.csdn.net/woshimalingyi/article/details/66477437)

[**https://blog.csdn.net/xiaoqun999/article/details/73335649**](https://blog.csdn.net/xiaoqun999/article/details/73335649)

**一、dumpsys命令介绍**

**1.命令说明**

Dumpsys用户系统诊断，它运行在设备上，并提供系统服务状态信息

命令格式： adb shell dumpsys [system serbices]

**2.系统服务查询**

如果直接运行adb shell dumpsys，将会获得所有的系统服务信息，那是非常多的，为了更容易管理输出，可以指定你想要检查的服务，例如：

adb shell dumpsys –l

adb shell serverce list

#都是查看有哪些系统服务（需要哪些服务就指定哪些服务）

命令行参数：

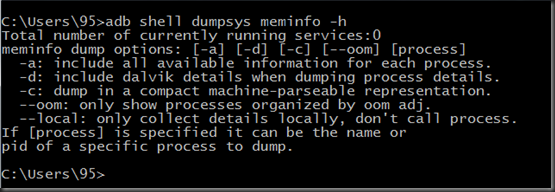
不同的服务有不同的选项，一下两个是常见的参数：

-h：对于大多数的服务，可以添加-h看到文本的帮助

-c：对于一些服务，可以添加-c查看数据会更友好

例如：

adb shell dumpsys meminfo –h

[](http://images2015.cnblogs.com/blog/885707/201604/885707-20160411021111410-503474376.png)

**二、包信息查询**

子命令格式：

adb shell dumpsys package [-h] [-f] [—checkin] [cmd]…

|  |  |
| --- | --- |
| 参数 | 说明 |
| -h | 打印帮助信息 |
| -f | 打印intent filter的信息 |
| --checkin | 打印出已经登记的库、系统功能、安装包 |
| cmd | 子命令（可以在-h帮助文档中查看有哪些子命令） |

|  |  |
| --- | --- |
| cmd子命令 | 说明 |
| prov[iders] | 获取content providers |
| p[ackages] | 获取安装包基本信息 |
| s[hared-user] | 获取共享用户ID的应用 |
| m[essages] | 打印运行时收集的信息 |
| v[erifiers] | 打印包校验信息 |
| version | 打印数据库版本信息 |
| write | 写当前位置 |
| <package.name> | 输出给定包的信息 |
| installs | 安装会话的详细信息 |
| l[ibraries] | 列出已知的共享库 |
| f[ibraries] | 列出手机的功能 |
| k[eysets] | 列出各个包的Signing KeySets |
| r[esolvers] | 获取intent filter |
| perm[issions] | 获取权限 |
| pref[erred] | 打印包首选项 |
| preferred-xml [—full] | 打印包首选项，xml格式打印 |

**三、activity信息查询**

子命令格式：

adb shell dumpsys activity [-a] [-c]…

|  |  |
| --- | --- |
| 参数 | 说明 |
| -a | 包括所有可用的服务器状态 |
| -c | 包括客户端状态 |
| -p | 限制输出为给定的包，例如：  adb shell dumpsys activity -p com.android.browser |
| -h | 打印帮助信息 |
| cmd | 子命令 |

|  |  |
| --- | --- |
| cmd子命令 | 说明 |
| a[ctivities] | activity堆栈状态 |
| r[recents] | 最近activity的状态 |
| b[rodacasts] [package\_name] [histpry [-s]] | 广播状态 |
| i[ntents] [package\_name] | 挂起的intent状态 |
| p[rocesses] [package\_name] | 进程状态 |
| o[om] | oom管理 |
| perm[issions] | url权限授权状态 |
| prov[iders] [comp\_spec…] | content provider状态 |
| provider [comp\_spec] | provider客户端状态 |
| s[ervices] [comp\_spec…] | 服务状态 |
| as[sociations] | 跟踪应用程序的关联 |
| service [comp\_spec] | 服务客户端状态 |
| package [package\_name] | 给的包的所有状态 |
| all | 转储所有的activityes |
| top | 转储栈顶的activity |
| write | 写入所有挂起状态存储 |
| track-associations | 允许会话跟踪 |
| untrack-associations | 禁用和明确会话跟踪，命令参数可能也是一个comp\_spec 转储的activity |

**四、网络信息查询**

|  |  |  |
| --- | --- | --- |
| 子命令 | 说明 | 命令格式 |
| connectivity | 网络连接 | adb shell dumpsys connectivity |
| netpolicy | 网络策略 | adb shell dumpsys netpolicy |
| netstats | 网络状态 | adb shell dumpsys netstats |
| network\_management | 网络管理 | adb shell dumpsys network\_management |

**五、其他常用服务信息查询**

|  |  |  |
| --- | --- | --- |
| 子命令 | 说明 | 命令格式 |
| meminfo | 内存 | adn shell dumpsys meminfo |
| cpuinfo | CPU | adn shell dumpsys cpuinfo |
| gfxinfo | 帧率 | adn shell dumpsys gfxinfo |
| display | 显示 | adn shell dumpsys display |
| power | 电源 | adn shell dumpsys power |
| batterystats | 电池状态 | adn shell dumpsys batterystats |
| battery | 电池 | adn shell dumpsys battery |
| alarm | 闹钟 | adn shell dumpsys alarm |
| location | 位置 | adn shell dumpsys location |

* 本文为博主学习笔记，未经博主允许不得转载
* 本文仅供交流学习，请勿用于非法途径
* 本文仅是个人意见，如有想法，欢迎拍砖

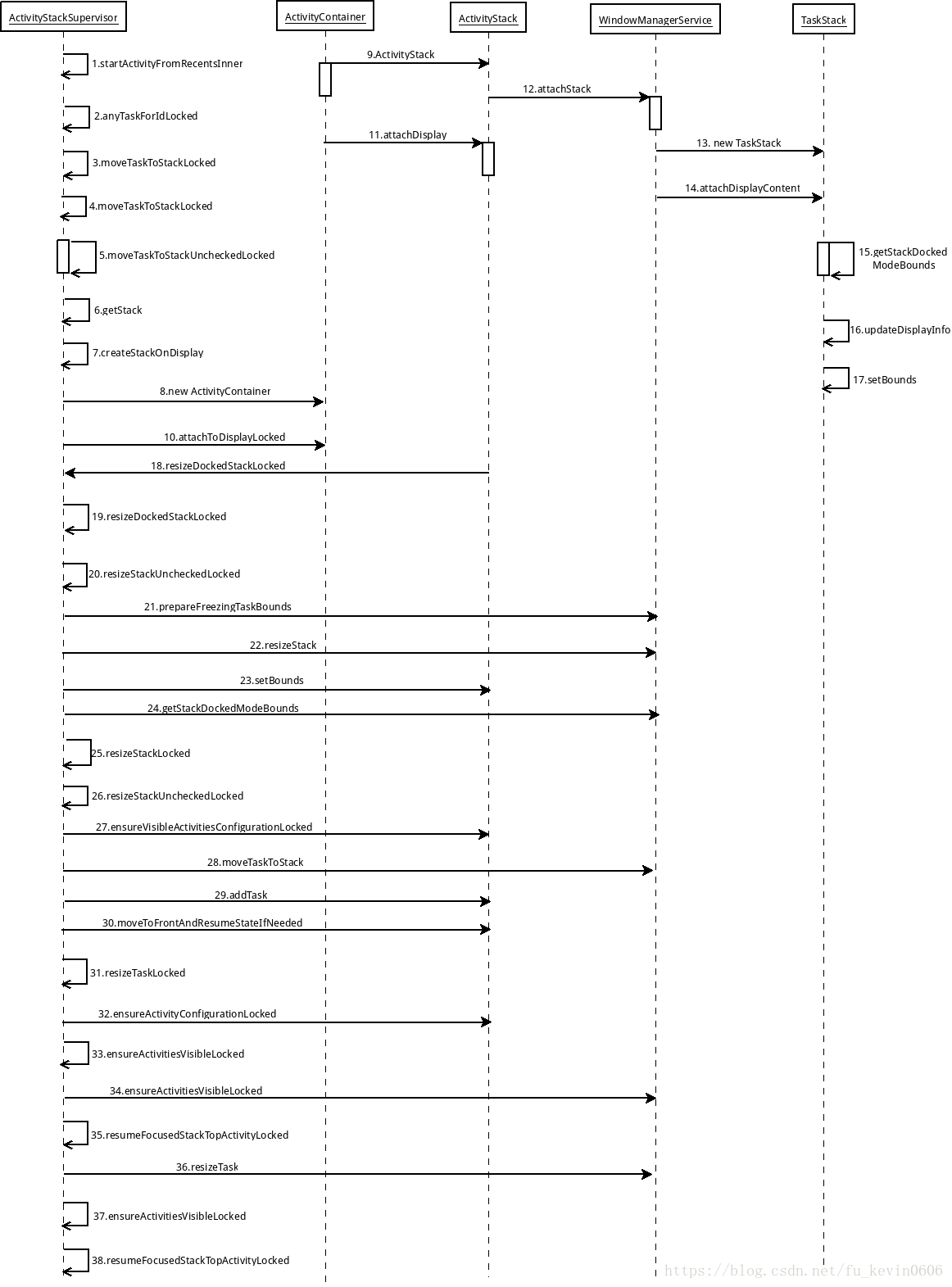
<https://github.com/walfud/TaskDemo>

# Android7.0多窗口实现原理(二)

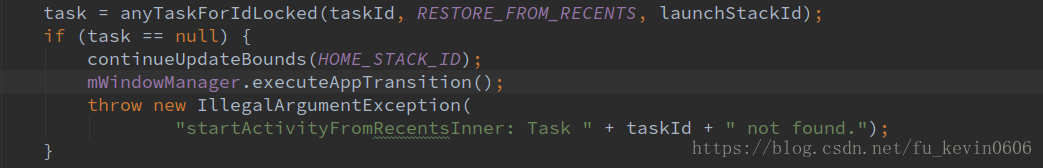
2018年08月30日 19:58:43 [宇落无痕](https://me.csdn.net/fu_kevin0606) 阅读数：315

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/fu\_kevin0606/article/details/82226600

    从[上文](https://blog.csdn.net/fu_kevin0606/article/details/81191078)可以知道当开始分屏时从SystemUI调用到[Activity](https://www.baidu.com/s?wd=Activity&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)StackSupervisor中的startActivityFromRecentsInner函数，当要分屏的Activity已经存在了，属于热启动分屏。

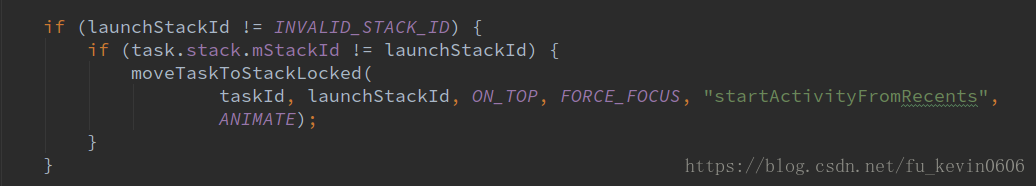


在anyTaskForIdLocked函数中就直接将对应的TaskRecord对象返回。

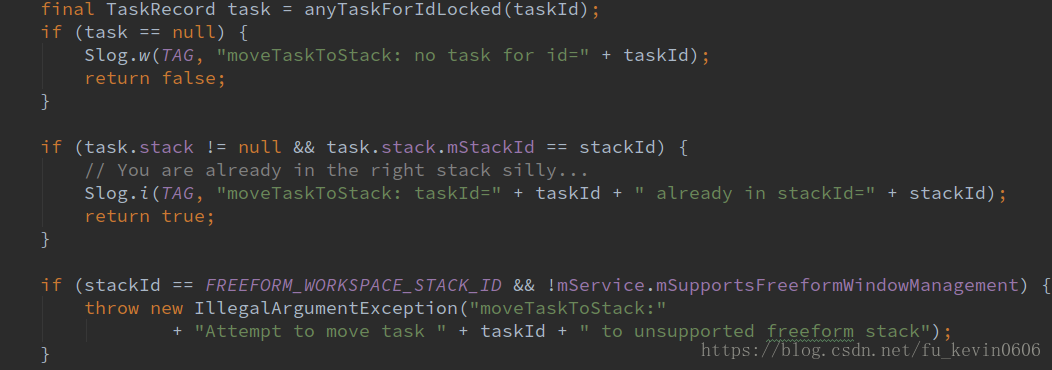


# Move Stack

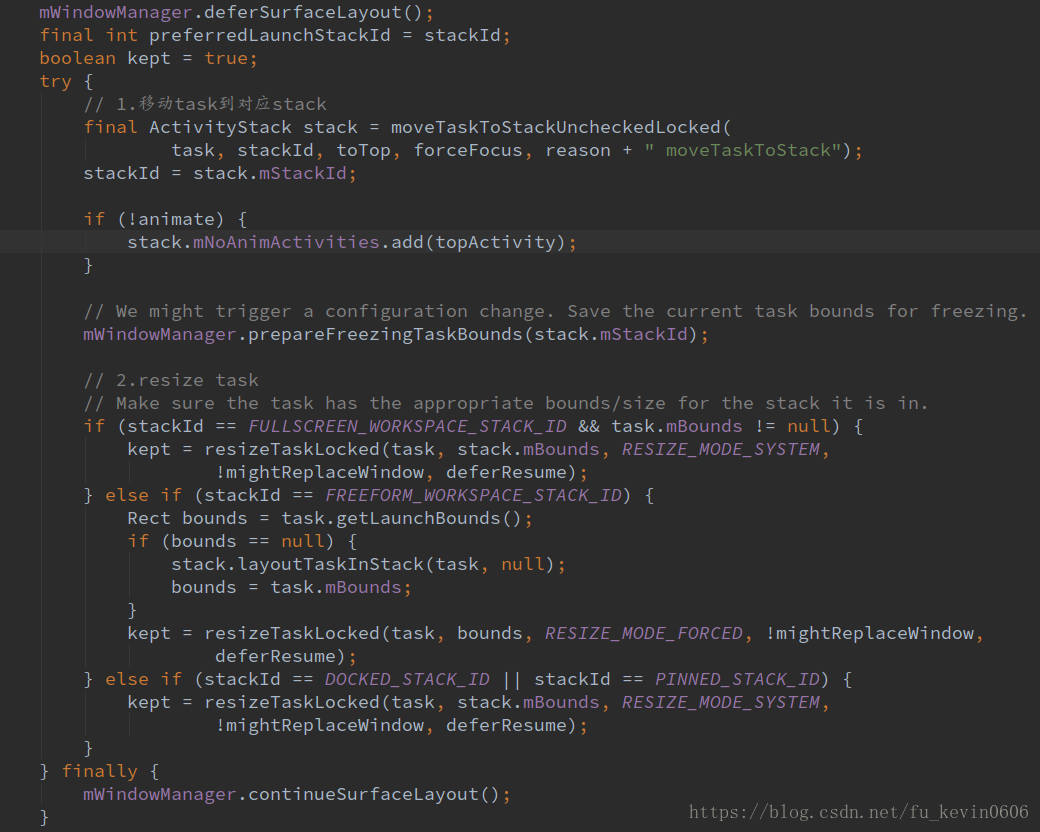
由于获取到的TaskRecord并不是在分屏所属的ActivityStack，launchStackId和获得task的stack id不相同，就调用moveTaskToStackLocked函数将Task移动到分屏的stack中。



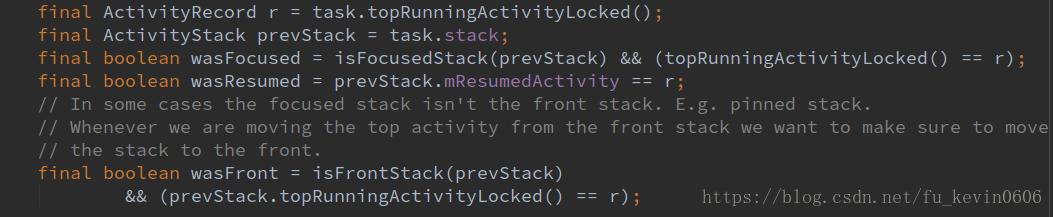
在moveTaskToStackLocked函数中，首先根据taskId获取到对应的TaskRecord，如果task为空，或者task所在的stack就是分屏stack就会直接返回，如果请求stack为freeform是stack，但是系统不支持freeform stack就抛出异常。



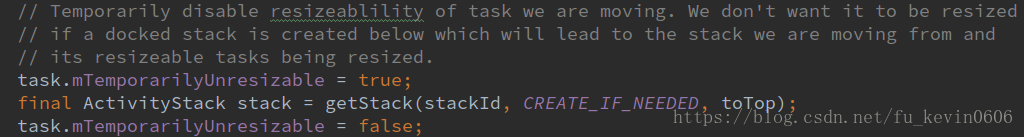
     接下来就需要将tack移动到对应到对应的stack中，之后记录新stack的id，冻结task的边界，对新的stack中的task进行重新计算尺寸。



    首先分析moveTaskToStackUncheckedLocked函数将task移动到分屏的stack中。获取task顶层正在运行的Activity，将task所在的stack记录在preStack中，表示为之前的stack，判断preStack是否为焦点stack并且现在最顶层的Activity是否与之前的相同，preStack中可见的Activity是否与之前最顶端的Activity相同，并且判断之前顶层的Activity是否是最前端的Activity。



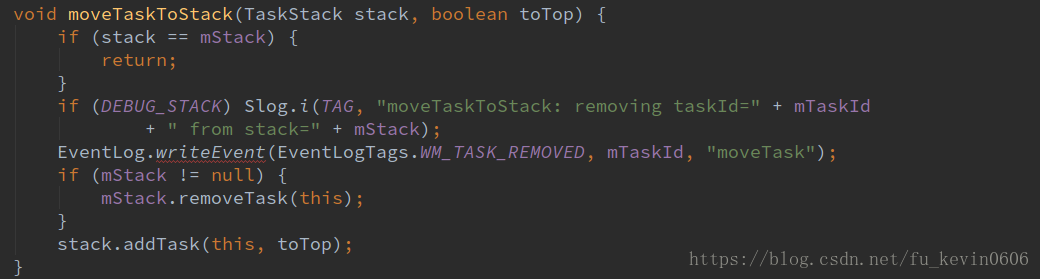
    当在获取分屏stack时，我们要临时将resizeablility设置为false，不能进行resize，当获取完stack才可以resize。getStack的详细调用流程在上文<https://blog.csdn.net/fu_kevin0606/article/details/81191078>做过详细的分析。获取到的stack为分屏stack，stack id为3.



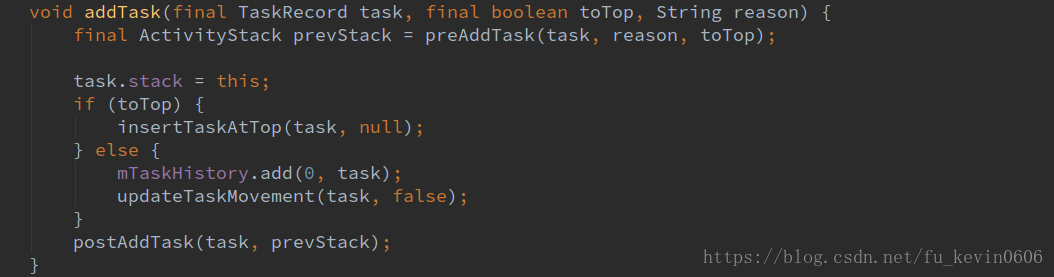
      获取完stack后调用WindowManagerService来移动task，在moveTaskToStack函数中根据taskId从mTaskIdToTask中获取到Task对象，然后根据stackId获取对应的TaskStack对象，之后调用Task.java的moveTaskToStack函数移动task到对应stack。

https://img-blog.csdn.net/20180910200809912?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2Z1X2tldmluMDYwNg==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

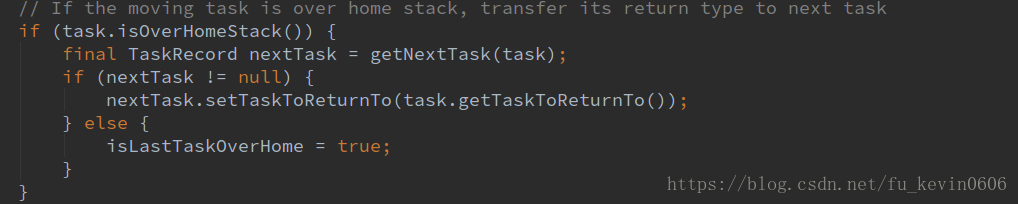
      在Task.java中判断task所属的TaskStack是否为空，如果不为空，就先将该task从TaskStack中移除，再重新添加到顶部。



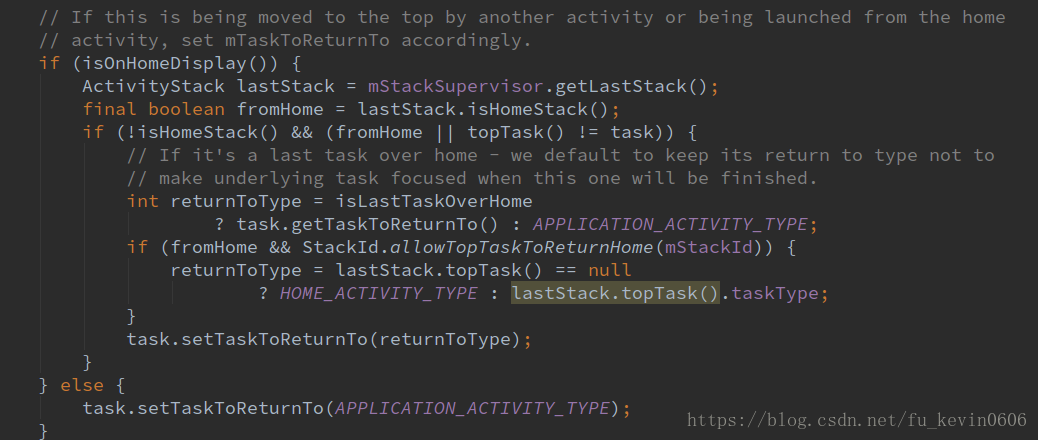
     在WMS中将task移动到对应的TaskStack后，需要将TaskRecord移动到分屏的ActivityStack中。由于需要将task添加的分屏stack的顶端，就调用insertTaskAtTop函数，将task插入顶端。



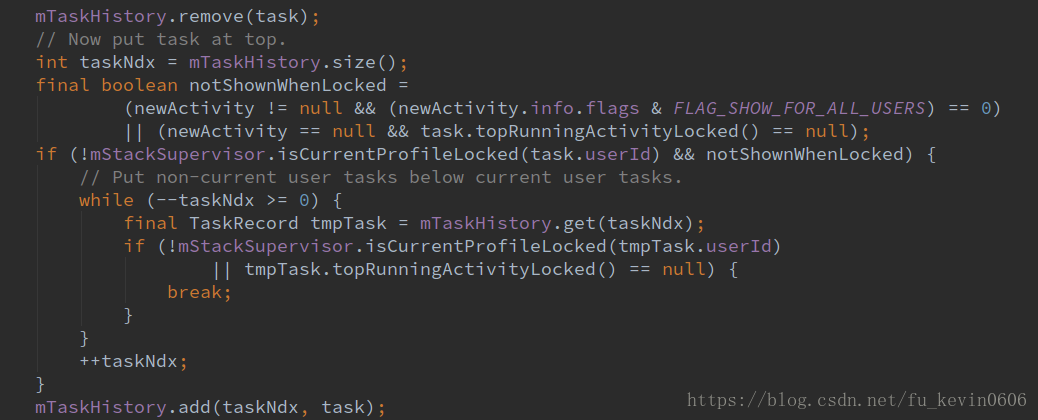
      在将task插入最顶端时，如果task是从Home[界面](https://www.baidu.com/s?wd=%E7%95%8C%E9%9D%A2&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)启动的，就获取这个task的下一个taskRecord，如果下一个task也不为空，就将这个task的返回类型传递给下一个taskRecord。



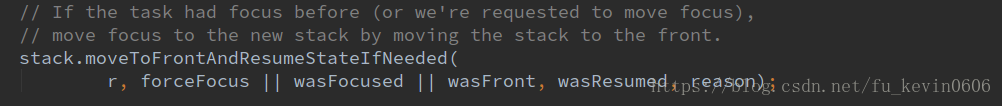
      之后根据Activity的启动类型，从那个界面启动的来设置Activity的返回类型，有事我们会遇到从一个界面返回，应该返回到主界面，却返回到了其他打开过的应用中，就是该返回值设置错误。



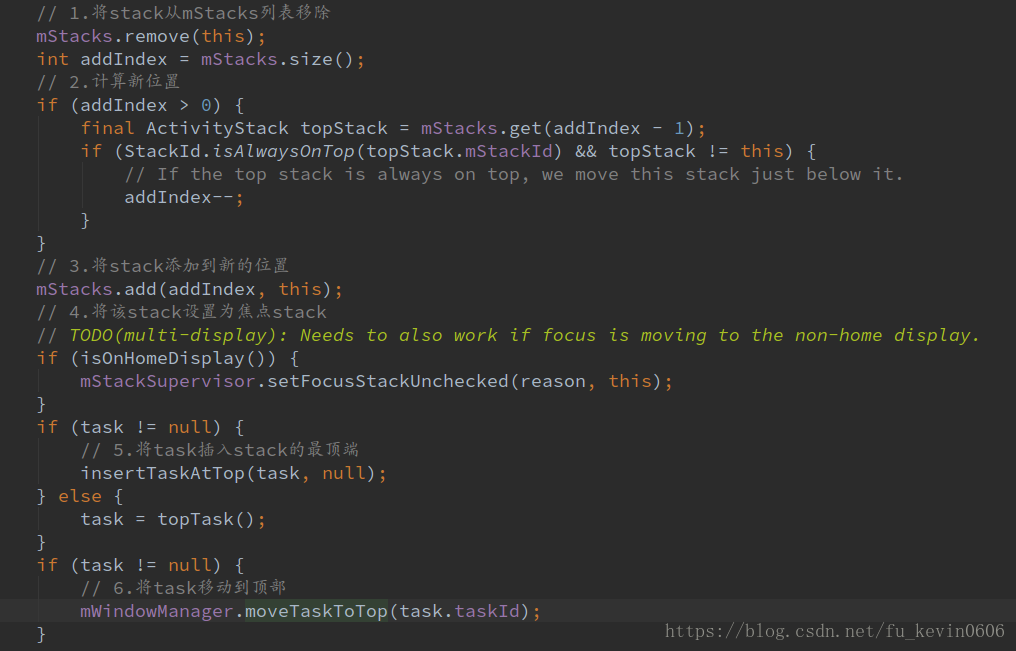
之后将task从历史列表中移除，从新计算位置，插入mTaskHistory列表。



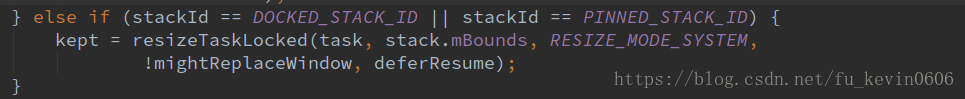
最后，如果我们移动的task之前有焦点，或者我们请求将焦点交给他，我们通过将stack移动到前面来移动焦点。



如果需将r赋值给mResumedActivity，调用moveToFront将stack移动到最前面。



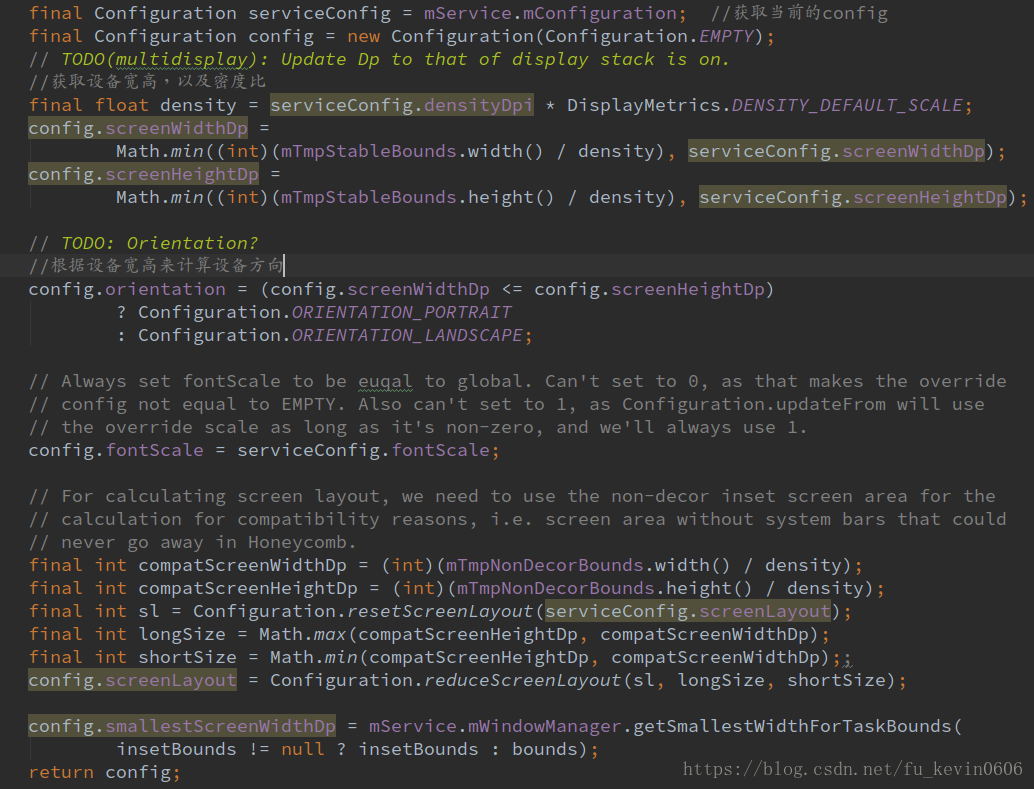
      将task移动到对应的stack中后，调用resizeTaskLocked函数resize task，由于此时为分屏stack，所以就使用分屏stack的边界来resize task。



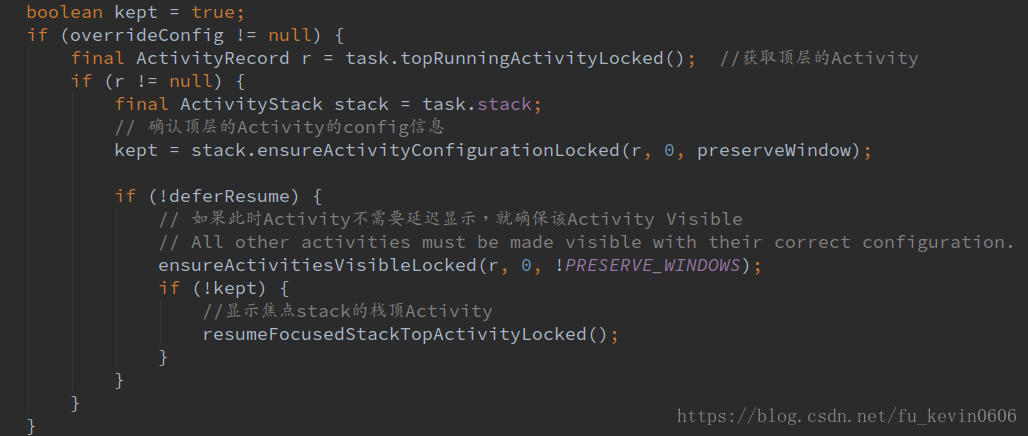
在resize task时会先判断stack边界是否可用，如果可用，之后调用TaskRecord的updateOverrideConfiguration函数来更新Task的配置数据信息。



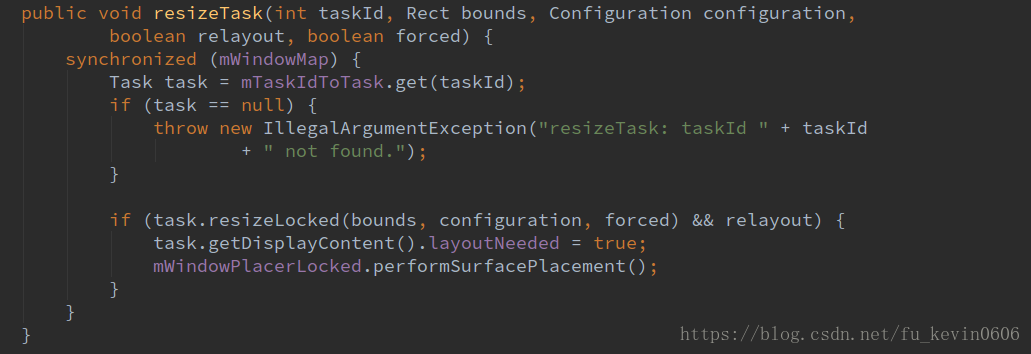
       在计算config信息时调用calculateOverrideConfig函数，在该函数中获得设备的宽高，根据宽高来计算设备的方向，最后将获取的config信息返回。来判断config是否发生改变，如果改变了就将新的config信息返回，如果没有发生改变就返回null。



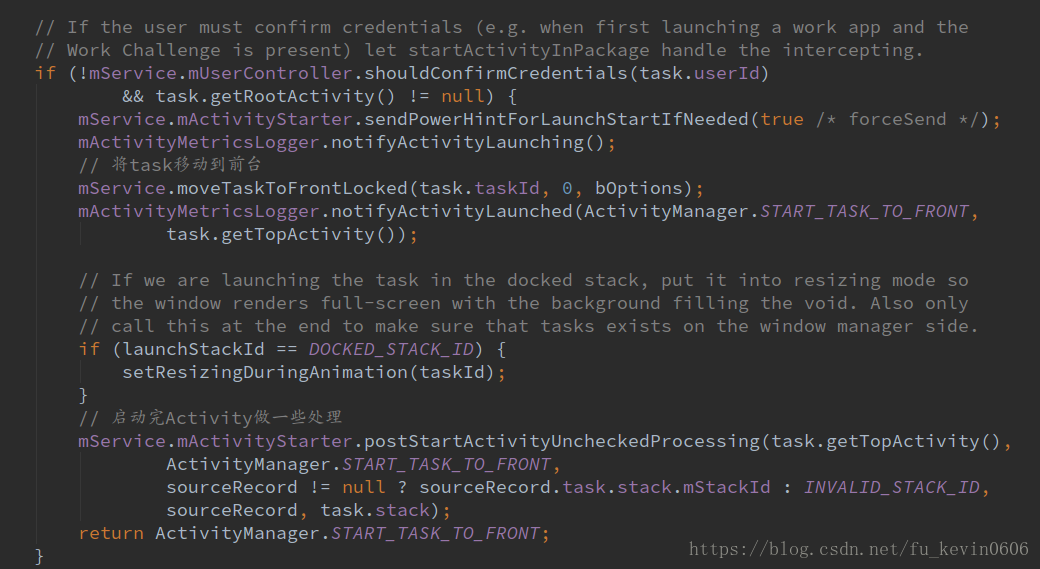
如果此时config发生了改变overrideConfig不为null，就获取顶层的Activity，确认顶层的Activity的config信息与当前系统的config相同，如果config不相同，就将系统的config赋值给顶层Activity，然后通知应用config发生改变调用onConfigurationChanged函数。之后确保Activity在正确的config下使Activity可见。最后显示顶端的Activity。



      之后调用WindowManagerService来resizeTask，根据taskId从mTaskIdToTask中取出对应的Task，调用的task的resizeLocked函数来resize task，当resize成功后，如果需要relayout，就需要调用performSurfacePlacement界面刷新函数来刷新界面。在resizeLocked函数中将bounds设置给task。



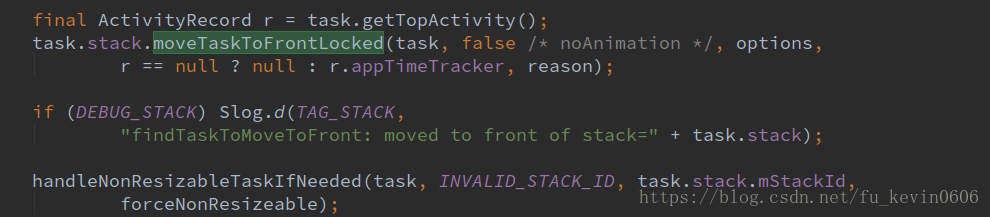
    此时将task从原来的stack[中移动](https://www.baidu.com/s?wd=%E4%B8%AD%E7%A7%BB%E5%8A%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)到了分屏stack中，并且对task重新计算了size，下面重新回到 startActivityFromRecentsInner函数中。



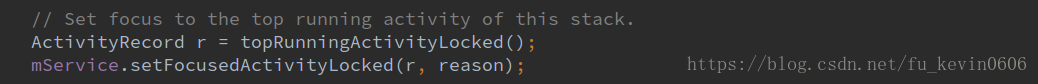
     前面主要是将另一个stack的task移动到分屏stack中，下面主要做的就是将移动到分屏stack的task移动到前台，可以被用户看到。该动作主要在moveTaskToFrontLocked函数中执行。首先根据taskId获取到对应的TaskRecord，获取当前正在运行的Activity记录在prev变量中，表现将要新的Activity要启动，该Activity变为前一个Activity。最后调用findTaskToMoveToFrontLocked函数来移动task。



    在findTaskToMoveToFrontLocked函数中，根据task对象获取到栈顶的Activity，然后调用ActivityStack的moveTaskToFrontLocked函数将分屏task移动到stack的顶端。



     在将task移动到stack顶端时调用insertTaskAtTop函数，将task插入顶端。此时task位于mTaskHistory列表的顶端，此时通过topRunningActivityLocked函数获得mTaskHistory顶端的task中顶端的Activity，也就是要进行分屏的Activity。将焦点设置在该Activity上面。



最后调用resumeFocusedStackTopActivityLocked来显示Activity。

# Android Tasks、ActivityStack和Activity

2017年03月26日 14:21:05 [黄舒颖 咸丫蛋](https://me.csdn.net/vshuang) 阅读数：1118

最近又在开展一些Android的开发工作，出现了在个别场景下发生的[Activity](https://www.baidu.com/s?wd=Activity&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)切换不符合预期的情况，于是来温习一下Android的Tasks、ActivityStack、Activity，以及影响到他们的几个重要参数launchMode、taskAffinity、IntentFlags

**应用程序的进程与线程**

从应用程序的角度来看，应用程序一般定义了应用对应的进程，以及上面运行着的一系列的线程。开发人员进行debug的时候的关注点会是哪个进程的哪个线程的哪个stacktrace。

**Android四大组件**

从Android应用程序的角度来看，除了进程和线程的这一个角度，从程序里面详细类的组织来看，Android应用程序包含有Activity、Service、ContentProvider、Broadcast Receiver4大组件，这四大组件就是通过我们之前说过的[XML插件方式](http://blog.csdn.net/vshuang/article/details/53439839)配置到AndroidManifest当中的。而这其中，Activity组件是Android应用程序[界面](https://www.baidu.com/s?wd=%E7%95%8C%E9%9D%A2&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)开发最重要的一个部分。

**Android系统界面用户体验组织（TASKS）**

从Android系统界面的组织来看，用户体验到的直观界面都是由很多Tasks组成的。比如，我们从最近任务栏，就可以看到很多TASKS。用户可以随意的切换到其中的一个TASK。也可以按[HOME键](https://www.baidu.com/s?wd=HOME%E9%94%AE&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)退出一个TASK。TASK的组织形式是ActivityStack，一个ActivityStack是由好多Activity组成的堆栈。从Android系统设计者的角度来看，一个Task定义了一组行为，而这组行为是由跨多个应用程序的多个Activity组织而构成的。这个打破了以往由应用程序（进程）来定义的资源边界。一个Task就是一个场景的实现。从此构建系统用户体验和行为的边界在于TASK，而非应用程序或者进程。

**Task**

* Task 是activities的集合，通过activity stack来管理，依靠先进后出队列来实现;
* 每个task中都至少有一个activity，新实例出来的activity置于栈顶
* Task可以被切换到后台

**Activity Stack**

* 如上所诉，Activity承担了大量的显示和交互工作，从某种角度上将，我们看见的应用程序就是许多个Activity的组合。为了让这许多 Activity协同工作而不至于产生混乱，Android平台设计了一种堆栈机制用于管理Activity，其遵循先进后出的原则，系统总是显示位于栈 顶的Activity，从逻辑上将，位于栈顶的Activity也就是最后打开的Activity，这也是符合逻辑的。
* 在操作应用程序时，每次启动新的Activity，都会将此压入Activity Stack，当用户执行返回操作时，移除Activity Stack顶上的Activity，这样就实现了返回上一个Activty的功能。直到用户一直返回到Home Screen，这时候可以理解为移除了Activity Stack所有的Activity，这个Activity Stack不再存在，应用程序也结束了运行。
* 可以通过 adb shell dumpsys |grep ActivityRecord 来查看 TASKS的ActivityStacks
* 可以通过 adb shell dumpsys activity activities |grep packageName| grep Run 来查看某个packageName的ActivityStatcks

### 

### task的taskAffinity

* taskAffinity 这个属性主要是决定持有每个activity属于哪个task。
* 默认情况下，同一个包中的activity共享同一个affinity（任务共用性）。

### 

### ****task的launchMode****

* standard(default)：standard，标准的Activity是可以随意插入到TASK中去的一个组织结构，可以去TaskA，也可以去TaskB，也可以去TaskC，直接并无任何的联系
* singleTop，如果在任务的栈顶正好存在该Activity的实例，就重用该实例，否则就创建新的实例并放入栈顶。
* singleTask，如果在栈中已经有该Activity的实例，就重用该实例(会调用实例的onNewIntent())。重用时，会让该实例回到栈顶，因此在它上面的实例将会被移除栈。如果栈中不存在该实例，将会创建新的实例放入栈中。
* singleInstance

### [Intent Flag介绍](http://www.cnblogs.com/xingfuzzhd/archive/2012/12/26/2834078.html" \t "_blank)

* 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\_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\_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\_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。

* 参考一个stackoverflow的问答 <https://stackoverflow.com/questions/7759556/flag-activity-exclude-from-recents-excludes-whole-application-not-only-the-acti>
* I have a Notification which starts an Activity. After a long press on home button and selecting my app, I want to start my main Activity again, and not this Activity started by the Notification. I tried with FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS, but this removed my whole application from the recents, and that's not what I want to achieve. How can I have my app in the recents, but have the main Activity started?
* Okay, I found the solution to my problem. I started an Activity from a Notification with FLAG\_ACTIVITY\_NEW\_TASK. But it seems to me that this Activity only gets started in an own task if affinity is different from the default affinity. So I had to add a different affinity in the manifest.

And it seems that FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS does not (as documented) exlucde the Activity from the recents, rather it excludes the whole task (not the whole application) in which the Activity gets started from the recents. And as I hadn't set a different affinity the Activity which I wanted to exclude was started in the same task (although I had set FLAG\_ACTIVITY\_NEW\_TASK) and so my whole application (as it was running in only one task) was excluded from the recents.

Now I've set a different affinity for the Activity that gets started from the Notification and I start it with FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS. When I leave this Activity and long-press the HOME button I can choose my app and the default task is started or brought to the front.

* 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。

* 参考：
* 1、[来自Android launchMode的温馨提示](http://www.itnose.net/detail/6347138.html)
* 2、[Android启动模式图文详解](http://www.devtf.cn/?p=541)
* 3、[TaskDemo](https://github.com/walfud/TaskDemo)

# 这是一个帮助你理解和记忆 android startActivity 中 launch mode 和 flags 作用的 demo 工程

上图中, 我们可以看到:

* 当前 app 中有 4 个 task
* 当前的 activity(id: 5) 位于 id=297 的 task 中

## xml 中 launchMode 共有四种:

* standard (ActivityInfo.LAUNCH\_MULTIPLE)
* singleTop (ActivityInfo.LAUNCH\_SINGLE\_TOP)
* singleTask (ActivityInfo.LAUNCH\_SINGLE\_TASK)
* singleInstance (ActivityInfo.LAUNCH\_SINGLE\_INSTANCE)

standard 和 singleTop 是一类. 它们都可以被创建多个 instance. 这两者的区别是:

* standard 无论何时接到 Intent, 都会在调用者的 task 中 push 一个新的 activity 放在尾部
* singleTop 当调用者当前 task 的顶部是 Intent 要启动的 activity 时, 则当前顶部的 activity 回收到 onNewIntent 并且不会启动新的 activity. 否则, 也会在调用者的 task 中 push 一个新的 activity 放在尾部

singleTask 和 singleInstance 是一类. 它们在系统中只能存在一个 instance. 这两者的区别是:

* singleTask 根据 affinity 决定在哪个 task 中启动. 如果存在该 affinity 所指定的 task, 则在该 task 中启动, 否则会新建一个该 affinity 的 task, 并在其中启动.
* singleInstance 被启动的 activity 只能单独存在于一个独立的 task 中. 如果从该 task 再次启动其它 activity, 则新启动的 activity 会被放到其它 task 中. 其它规则同 singleTask

以下是我总结的神图:

1. 从 standard activity 中启动其它类型的 activity, 总结如图:

## 代码中常用的有如下 flag:

## FLAG\_ACTIVITY\_NEW\_TASK

* 如果 task 中没有该 Activity, 则新建一个 task 并启动 Activity
* 否则, 拥有该 Activity 的 task 会被带到前台.
* 如果调用者想要从 Activity 获取 result, 请不要使用此标记

通常 launcher 会使用该标记启动新 Activity

## FLAG\_ACTIVITY\_MULTIPLE\_TASK

* 只能和 FLAG\_ACTIVITY\_NEW\_TASK 或者 FLAG\_ACTIVITY\_NEW\_DOCUMENT. 单独使用或和其他标记组合则无效
* 使上述两个标记跳过 task 查找过程, 从而无条件创建 Activity

## FLAG\_ACTIVITY\_CLEAR\_TASK

* 只能和 FLAG\_ACTIVITY\_NEW\_TASK 组合使用
* 如果已经有 task 运行该 Activity, 则清除该 task 以及其中所有 Activity, 然后在一个新的 task 中创建 Activity

## FLAG\_ACTIVITY\_TASK\_ON\_HOME

* 只能和 FLAG\_ACTIVITY\_NEW\_TASK 组合使用
* 该 Activity 会添加到 home 所在的 task 中. 这会导致从该 Activity 返回会直接返回 home

## FLAG\_ACTIVITY\_SINGLE\_TOP

* 如果在 history stack 的 top, 则不再创建新的

## FLAG\_ACTIVITY\_CLEAR\_TOP

* 判断当前栈
* 清空所有上层 Activity
  + 指定 FLAG\_ACTIVITY\_SINGLE\_TOP -> onNewIntent
  + 否则 -> 销毁旧 Activity 并创建新的

## FLAG\_ACTIVITY\_NO\_HISTORY

* 不在 history stack 中显示
* 当切换走的时候 (navigate 或者 home), 则 finish
* 无法接受 onActivityResult

## FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS

* 不在 history stack 中显示

## FLAG\_ACTIVITY\_REORDER\_TO\_FRONT

* 如果 task: A B C D, 那么启动 B 后: A C D B
* 如果同时设置了 FLAG\_ACTIVITY\_CLEAR\_TOP, 则此标记无效

## FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP

* 当前 Activity 不作为 top. 也就是说, top 依然是上一个 Activity

通常用作很快就结束的 Activity 或者是有从属关系的 Activity 之用

## 其它

* xml 中无法指定 FLAG\_ACTIVITY\_CLEAR\_TOP, 而 flag 中无法指定 singleInstance
* 脱离 affinity 讨论 singleTask 和 singleInstance 是不科学的
* [Tasks and Back Stack](http://developer.android.com/guide/components/tasks-and-back-stack.html) 中说:

The system creates a new task and instantiates the activity at the root of the new task...

这是不严谨的. 因为同一个应用中如果不指定 'taskAffinity' 的话, 那么 'singleTask' 会在默认的 task (即 MAIN/LAUNCHER 所在的 task) 中启动.

[official doc](http://developer.android.com/guide/topics/manifest/activity-element.html) 中还说:

Since activities with "singleTask" or "singleInstance" launch modes can only be at the root of a task...

显然, 这也是错误的. 当你从 MAIN/LAUNCHER 的 activity 中启动一个没指定任何 affinity 的 'singleTask' activity 时, 这个新启动的 activity 会在当前 task 中被创建(见上描述), 并且是在 MAIN/LAUNCHER 的上面. 因此, 'singleTask' 的 activity 未必一定是 the root activity of a task.

## Ref

[Tasks and Back Stack](http://developer.android.com/guide/components/tasks-and-back-stack.html)

[official doc](http://developer.android.com/guide/topics/manifest/activity-element.html)

[解开Android应用程序组件Activity的"singleTask"之谜](http://blog.csdn.net/luoshengyang/article/details/6714543)

[深入讲解Android中Activity launchMode](http://droidyue.com/blog/2015/08/16/dive-into-android-activity-launchmode/)

[android深入解析Activity的launchMode启动模式，Intent Flag，taskAffinity](http://blog.csdn.net/self_study/article/details/48055011)

## Contribute

此工程还需要很多努力才能变得更加完善, 比如: 目前 startActivity/back 后需要等待 2s 后才能对 task 进行绘制, 因为要等待 activity 的 onDestroy 完成; 再或者, 目前界面上 task 都是一个 id 标识, 不那么直观, 可以增加一些颜色, 比如 standard 用绿色, singleTop 用黄色等等(参考 [Activities-LaunchMode-demo](https://github.com/gnorsilva/Activities-LaunchMode-demo)). 希望你也能为它添砖加瓦, 帮助更多的人.

Any question, feel free to contact me.

# Android——带你从ADB命令行查看Task任务栈的管理方式

2017年03月26日 17:59:46 [David-Kuper](https://me.csdn.net/woshimalingyi) 阅读数：3818更多

所属专栏： [Android开发](https://blog.csdn.net/column/details/androiddesignstyle.html)

 版权声明：本文为博主原创文章，转载请说明出处。 https://blog.csdn.net/woshimalingyi/article/details/66477437

* [一概念](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E4%B8%80%E6%A6%82%E5%BF%B5)
  + [1Task](https://blog.csdn.net/woshimalingyi/article/details/66477437#1task)
  + [2ActivityStack](https://blog.csdn.net/woshimalingyi/article/details/66477437#2activitystack)
  + [3ActivityRecordProcessRecord](https://blog.csdn.net/woshimalingyi/article/details/66477437#3activityrecordprocessrecord)
* [二任务栈的切换演示](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E4%BA%8C%E4%BB%BB%E5%8A%A1%E6%A0%88%E7%9A%84%E5%88%87%E6%8D%A2%E6%BC%94%E7%A4%BA)
* [情况一单一Task默认的启动模式下App切换时的Activity栈情况](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E6%83%85%E5%86%B5%E4%B8%80%E5%8D%95%E4%B8%80task%E9%BB%98%E8%AE%A4%E7%9A%84%E5%90%AF%E5%8A%A8%E6%A8%A1%E5%BC%8F%E4%B8%8Bapp%E5%88%87%E6%8D%A2%E6%97%B6%E7%9A%84activity%E6%A0%88%E6%83%85%E5%86%B5)
  + [1从App2ActivityB启动App1ActivityA](https://blog.csdn.net/woshimalingyi/article/details/66477437#1%E4%BB%8Eapp2activityb%E5%90%AF%E5%8A%A8app1activitya)
  + [2接着上面我们又从App1ActivityA启动App2ActivityB](https://blog.csdn.net/woshimalingyi/article/details/66477437#2%E6%8E%A5%E7%9D%80%E4%B8%8A%E9%9D%A2%E6%88%91%E4%BB%AC%E5%8F%88%E4%BB%8Eapp1activitya%E5%90%AF%E5%8A%A8app2activityb)
* [情况二单一Task含有SingleTask的模式App切换的Activity栈情况](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E6%83%85%E5%86%B5%E4%BA%8C%E5%8D%95%E4%B8%80task%E5%90%AB%E6%9C%89singletask%E7%9A%84%E6%A8%A1%E5%BC%8Fapp%E5%88%87%E6%8D%A2%E7%9A%84activity%E6%A0%88%E6%83%85%E5%86%B5)
  + [1从ActivityB切换至ActivityA](https://blog.csdn.net/woshimalingyi/article/details/66477437#1%E4%BB%8Eactivityb%E5%88%87%E6%8D%A2%E8%87%B3activitya)
  + [2将MainActivity启动模式变为singleTask这时如果我们从App2ActivityB切换至App1MainActivity会发什么](https://blog.csdn.net/woshimalingyi/article/details/66477437#2%E5%B0%86mainactivity%E5%90%AF%E5%8A%A8%E6%A8%A1%E5%BC%8F%E5%8F%98%E4%B8%BAsingletask%E8%BF%99%E6%97%B6%E5%A6%82%E6%9E%9C%E6%88%91%E4%BB%AC%E4%BB%8Eapp2activityb%E5%88%87%E6%8D%A2%E8%87%B3app1mainactivity%E4%BC%9A%E5%8F%91%E4%BB%80%E4%B9%88)
* [情况三多TaskSingleTask模式App切换时Activity栈的情况](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E6%83%85%E5%86%B5%E4%B8%89%E5%A4%9Atasksingletask%E6%A8%A1%E5%BC%8Fapp%E5%88%87%E6%8D%A2%E6%97%B6activity%E6%A0%88%E7%9A%84%E6%83%85%E5%86%B5)
  + [1从App2ActivatyB启动App1ActivityA](https://blog.csdn.net/woshimalingyi/article/details/66477437#1%E4%BB%8Eapp2activatyb%E5%90%AF%E5%8A%A8app1activitya)
  + [2从App2ActivityB启动App1MainActivity](https://blog.csdn.net/woshimalingyi/article/details/66477437#2%E4%BB%8Eapp2activityb%E5%90%AF%E5%8A%A8app1mainactivity)
* [三总结](https://blog.csdn.net/woshimalingyi/article/details/66477437#%E4%B8%89%E6%80%BB%E7%BB%93)

# ****一、概念****

## ****1、Task****

Task是指将相关的Activity组合到一起，以Activity Stack的方式进行管理。从用户体验上讲，一个“应用程序”就是一个Task(默认情况)，但是从根本上讲，一个Task是可以有一个或多个Android Application组成的。   
Task表示一个任务，每一个Task 会有一个ID作为唯一标识。一个进程（程序）有一个默认的task affinity,默认为包名，也可以多个程序指定同一个TaskAffinity。Activity的启动会进入到对应的Task当中或者重新启动一个新的Task。Android源码里面有一个TaskRecord类，该类的一个实例就对应着一个Task。   
一组Task以链表的方式相互连接，每一个Task会有一个前驱任务点和后继任务点，以此来管理Task的编辑和查找。ActivityStack外部以链表的形式管理Task，Task内部以后进先出的历史顺序方式管理Activity。   
下面的Task的一段源码，我们可以看到它的主要结构：

final class TaskRecord {

private static final String TAG = TAG\_WITH\_CLASS\_NAME ? "TaskRecord" : TAG\_AM;

......

static final String ATTR\_TASKID = "task\_id";

......

*//任务的前驱节点*

TaskRecord mPrevAffiliate; *// previous task in affiliated chain.*

int mPrevAffiliateTaskId = INVALID\_TASK\_ID; *// previous id for persistence.*

*//该任务的后继节点*

TaskRecord mNextAffiliate; *// next task in affiliated chain.*

int mNextAffiliateTaskId = INVALID\_TASK\_ID; *// next id for persistence.*

......

/\*\* List of all activities in the task arranged in history order \*/

*//该Task下的Activity，以后进先出的方式存放*

final ArrayList<ActivityRecord> mActivities;

/\*\* 当前的Activity栈 \*/

ActivityStack stack;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

## ****2、ActivityStack****

Stack表示任务栈，任务栈是以Task为基本单位的。一般我们的程序切换就是Task的切换，一个程序从后台任务栈到前台任务栈，是整个任务栈的移动，而不仅仅是Activity的移动，在任务栈移动之后，再按照LaunchModel的规则对Activity进行操作。与Stack相对应的有Android源码里面的ActivityStack类。一般来说，我们的应用程序都在一个单例的ActivityStack当中。   
ActivityStack里面还维护着一个Activity的LRU运行列表，按照最近访问且正在运行的Activity排列。

/\*\*

\* 单列的Activity栈

\* State and management of a single stack of activities.

\*/

final class ActivityStack {

/\*\*

\* List of running activities, sorted by recent usage.

\* The first entry in the list is the least recently used.

\* It contains HistoryRecord objects.

\*/

final ArrayList<ActivityRecord> mLRUActivities = new ArrayList<>();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

这个是使用adb shell dumpsys activity 查看Activity栈信息的时候打印出来的一个LRU的Activity列表。

*//LRU的方式管理的正在运行的Activity列表*

Running activities (most recent first):

*//#276表示该Task的TaskID; A = com.example.malingyi.testtask1 表示该Task的TaskAffinity值，sz = 2表示该Task内部Activity的数量*

TaskRecord{51e812a *#276 A=com.example.malingyi.testtask1 U=0 sz=2}*

Run *#3: ActivityRecord{c5e365e u0 com.example.malingyi.testtask1/.ActivityA t276}*

TaskRecord{56c2a15 *#277 A=com.example.malingyi.testtask2 U=0 sz=2}*

Run *#2: ActivityRecord{e3bed35 u0 com.example.malingyi.testtask2/.ActivityB t277}*

Run *#1: ActivityRecord{532f30e u0 com.example.malingyi.testtask2/.MainActivity t277}*

TaskRecord{51e812a *#276 A=com.example.malingyi.testtask1 U=0 sz=2}*

Run *#0: ActivityRecord{e5c7c57 u0 com.example.malingyi.testtask1/.MainActivity t276}*

*//前台正在运行的Activity*

mResumedActivity: ActivityRecord{3a73c94 u0 com.example.malingyi.testtask2/.ActivityA t253}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

## ****3、ActivityRecord、ProcessRecord****

ActivityRecord：代表着一个在Task里面正在活动的Acitivty的信息记录。

/\*\*

\* An entry in the history stack, representing an activity.

\*/

final class ActivityRecord {

private static final String TAG = TAG\_WITH\_CLASS\_NAME ? "ActivityRecord" : TAG\_AM;

...

final ActivityManagerService service; *// owner*

final IApplicationToken.Stub appToken; *// window manager token*

final ActivityInfo info; *// all about me*

final ApplicationInfo appInfo; *// information about activity's app*

final int userId; *// Which user is this running for?*

final Intent intent; *// the original intent that generated us*

final ComponentName realActivity; *// the intent component, or target of an alias.*

...

*//包的名称*

final String packageName; *// the package implementing intent's component*

*//进程的名称*

final String processName; *// process where this component wants to run*

*//Task的名称*

final String taskAffinity; *// as per ActivityInfo.taskAffinity*

*//Activity所在的任务栈*

TaskRecord task; *// the task this is in.*

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

ProcessRecord：当前运行的进程信息，描述了该进程的所有信息。

/\*\*

\* Full information about a particular process that

\* is currently running.

\*/

final class ProcessRecord {

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

# ****二、任务栈的切换演示：****

假设：

有两个程序：App1、App2，包名为：App1 = com.example.malingyi.testtask1 App2 = com.example.malingyi.testtask1；   
ActivityA属于App1、ActivityB属于App2；两个App都有MainActivity作为启动入口。

# ****情况一：单一Task&默认的启动模式下，App切换时的Activity栈情况****

**启动模式：**

ActivityA：standard   
ActivityB：standard   
App1、App2：MainActivity：standard

**TaskAffinity：**

都是默认情况下的TaskAffinity（包名）

**初始化启动顺序：**

Step1：启动App1：MainActivity——>ActivityA ;   
Step2：启动App2：MainActivity——>ActivityB;

\*\*栈顶Activity：\*\*ActivityB

**Activity栈：**

App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB

初始任务栈信息如下：

Stack #8:

Task id #282 //App2的Task栈

TaskRecord{cde1503 #282 A=com.example.malingyi.testtask2 U=0 sz=2} //Task信息

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #1: ActivityRecord{218b549 u0 com.example.malingyi.testtask2/.ActivityB t282}//Activity信息

Intent { cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{735919e u0 com.example.malingyi.testtask2/.MainActivity t282}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63} //进程信息

Task id #281。//App1的Task栈

TaskRecord{8480c80 #281 A=com.example.malingyi.testtask1 U=0 sz=2} //Task信息

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #1: ActivityRecord{66d10ee u0 com.example.malingyi.testtask1/.ActivityA t281}//Activity信息

Intent { cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

Hist #0: ActivityRecord{157ed43 u0 com.example.malingyi.testtask1/.MainActivity t281}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60} //进程信息

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

## ****（1）从App2.ActivityB启动App1.ActivityA****

切换后任务栈：

Stack #8:

Task id #282 //栈顶还是App2的Task，APP.ActivityA进入了App2的Task栈中。

TaskRecord{cde1503 #282 A=com.example.malingyi.testtask2 U=0 sz=3}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #2: ActivityRecord{c6170ac u0 com.example.malingyi.testtask1/.ActivityA t282}

Intent { act=com.exaple.malingyi.ActivityA cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

Hist #1: ActivityRecord{218b549 u0 com.example.malingyi.testtask2/.ActivityB t282}

Intent { cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{735919e u0 com.example.malingyi.testtask2/.MainActivity t282}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Task id #281

TaskRecord{8480c80 #281 A=com.example.malingyi.testtask1 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #1: ActivityRecord{66d10ee u0 com.example.malingyi.testtask1/.ActivityA t281}

Intent { cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

Hist #0: ActivityRecord{157ed43 u0 com.example.malingyi.testtask1/.MainActivity t281}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB   
操作后的栈：App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB——>App1.ActivityA

这里我们看到，App1.ActivityA重新创建了一个实例进入了App2的Task栈。

## ****（2）接着上面，我们又从App1.ActivityA启动App2.ActivityB****

任务栈信息：

Task id #282 //栈顶仍然是App2的Task，APP2.ActivityB进入了App2的Task栈中。

TaskRecord{cde1503 #282 A=com.example.malingyi.testtask2 U=0 sz=4}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #3: ActivityRecord{6e68c29 u0 com.example.malingyi.testtask2/.ActivityB t282}

Intent { act=com.exaple.malingyi.ActivityB cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Hist #2: ActivityRecord{c6170ac u0 com.example.malingyi.testtask1/.ActivityA t282}

Intent { act=com.exaple.malingyi.ActivityA cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

Hist #1: ActivityRecord{218b549 u0 com.example.malingyi.testtask2/.ActivityB t282}

Intent { cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{735919e u0 com.example.malingyi.testtask2/.MainActivity t282}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{77bd789 3202:com.example.malingyi.testtask2/u0a63}

Task id #281

TaskRecord{8480c80 #281 A=com.example.malingyi.testtask1 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #1: ActivityRecord{66d10ee u0 com.example.malingyi.testtask1/.ActivityA t281}

Intent { cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

Hist #0: ActivityRecord{157ed43 u0 com.example.malingyi.testtask1/.MainActivity t281}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{57c82b9 3305:com.example.malingyi.testtask1/u0a60}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB   
操作后的栈：App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB——>App1.ActivityA——>App2.ActivityB

同样，我们看到新启动的ActivityB，还是在原有的App2的Task栈中。   
**这里的到了两个结论：**

* standard模式下，Activity会重新创建一个新的实例，不论原有实例是否存在。
* standard模式下，Activity会进入启动它的Activity所在的Task栈中，这个结论对于在启动其他App的Activity也成立。

# ****情况二：单一Task&含有SingleTask的模式，App切换的Activity栈情况****

**启动模式：**

ActivityA：SingleTask   
ActivityB：SingleTask   
App1、App2：MainActivity：standard

**TaskAffinity：**

都是默认情况下的TaskAffinity（包名）

**初始化启动顺序：**

Step1：启动App1：MainActivity——>ActivityA ;   
Step2：启动App2：MainActivity——>ActivityB;

**栈顶Activity**:ActivityB

**Activity栈：**

App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB

初始任务栈：

//所有启动的Activity，按照Task来分组，按照任务的访问顺序从上到下的排列。

Display #0 (activities from top to bottom):

Stack #7:

Task id #277 //App2的Task，里面维护了Activity的入栈顺序

TaskRecord{56c2a15 #277 A=com.example.malingyi.testtask2 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #1: ActivityRecord{e3bed35 u0 com.example.malingyi.testtask2/.ActivityB t277}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{9d3fb1b 2959:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{532f30e u0 com.example.malingyi.testtask2/.MainActivity t277}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{9d3fb1b 2959:com.example.malingyi.testtask2/u0a63} //标明Task所属的进程

Task id #276。 //App1的Task，里面维护了Activity的入栈顺序

TaskRecord{51e812a #276 A=com.example.malingyi.testtask1 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malingyi.testtask1/.MainActivity }

Hist #1: ActivityRecord{c5e365e u0 com.example.malingyi.testtask1/.ActivityA t276}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{7a654b8 2877:com.example.malingyi.testtask1/u0a60}

Hist #0: ActivityRecord{e5c7c57 u0 com.example.malingyi.testtask1/.MainActivity t276}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malingyi.testtask1/.MainActivity }

ProcessRecord{7a654b8 2877:com.example.malingyi.testtask1/u0a60} //标明Task所属的进程

//正在运行的是App2的ActivityB

mResumedActivity: ActivityRecord{bf1f68 u0 com.example.malingyi.testtask1/com.example.malingyi.testtask2.ActivityB t272}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

## ****（1）从ActivityB切换至ActivityA:****

切换后任务栈：

Stack #7:

Task id #276 //App1的整个栈上移了！不止是Activity移到了前面，同时在Task下的MainActivity也一起移动了。

TaskRecord{51e812a #276 A=com.example.malingyi.testtask1 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malingyi.testtask1/.MainActivity }

Hist #1: ActivityRecord{c5e365e u0 com.example.malingyi.testtask1/.ActivityA t276}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{7a654b8 2877:com.example.malingyi.testtask1/u0a60}

Hist #0: ActivityRecord{e5c7c57 u0 com.example.malingyi.testtask1/.MainActivity t276}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10000000 cmp=com.example.malingyi.testtask1/.MainActivity }

ProcessRecord{7a654b8 2877:com.example.malingyi.testtask1/u0a60}

Task id #277 //App2的栈下调了

TaskRecord{56c2a15 #277 A=com.example.malingyi.testtask2 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #1: ActivityRecord{e3bed35 u0 com.example.malingyi.testtask2/.ActivityB t277}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{9d3fb1b 2959:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{532f30e u0 com.example.malingyi.testtask2/.MainActivity t277}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{9d3fb1b 2959:com.example.malingyi.testtask2/u0a63}

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB

操作后的栈：App2.ActivityB——>App1.ActivityA——>App1.MainActivity——>App1.ActivityA

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

原本只是从ActivityA切换到ActivityB，由于ActivityB是SingleTask同时实例已经存在，那么应该是ActivityB移动到ActivityA之前，但是MainActivity也同时移到了ActivityA之前。MainActivity与ActivityA整个Task栈都移动了。按照Task的定义，ActivityStack栈里是按照Task分组来管理Activity，那么有可能APP之间的切换的调度单位会是Task。

## ****（2）将MainActivity启动模式变为：singleTask。这时如果我们从App2.ActivityB切换至App1.MainActivity会发什么？****

切换后任务栈：

Stack #10:

Task id #289 //App1的Task栈移动到了App2的Task栈之前，但是这时情况是MainActivity之前的App1.ActivityA弹出了。Task栈下只剩MainActivity

TaskRecord{fd59417 #289 A=com.example.malingyi.testtask1 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #0: ActivityRecord{24d5faa u0 com.example.malingyi.testtask1/.MainActivity t289}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{219a9ed 3807:com.example.malingyi.testtask1/u0a60}

Task id #290

TaskRecord{1d94004 #290 A=com.example.malingyi.testtask2 U=0 sz=2}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #1: ActivityRecord{dd07174 u0 com.example.malingyi.testtask2/.ActivityB t290}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{101c022 3830:com.example.malingyi.testtask2/u0a63}

Hist #0: ActivityRecord{2c3fdf1 u0 com.example.malingyi.testtask2/.MainActivity t290}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{101c022 3830:com.example.malingyi.testtask2/u0a63}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB   
操作后的栈：App2.ActivityB——>App1.ActivityA——>App1.MainActivity

MainActivity为SingleTask之后，由于它已经存在了一个实例，因此切换的时候会把App1的Task移动到App2的Task之前。然后根据SingleTask的规则，会将Task内在它之前的的Activity弹出，使得它置顶。

**到了这里，我们可以得出一个结论：**

* APP切换时，首先，会将切换的App所在的Task移动到最前面；然后，再使用LaunchModel的规则对Activity进行出栈入栈的操作。

**然后再作出一个假设：**

* 我们在两个程序之间的切换可能就是Task的切换？

# ****情况三：多Task&SingleTask模式，App切换时Activity栈的情况****

**启动模式&TaskAffinity：**

App1、App2的MainActivity的LaunchModel = SingleTask, TaskAffinity = 包名；   
App1中ActivityA: LaunchModel = SingleTask, TaskAffinity = com.TaskA；   
App2中ActivityB: LaunchModel = SingleTask, TaskAffinity = com.TaskB；

**初始化启动顺序：**

Step1：启动App1：MainActivity——>ActivityA ;   
Step2：启动App2：MainActivity——>ActivityB;

**栈顶Activity**:ActivityB

**初始Activity栈：**

App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB

初始任务栈：

//我们可以看到，每一个Activity都在一个Task中，按照他们启动的顺序从上倒下。

Stack #11:

Task id #296 //App2.ActivityB的任务栈com.TaskB

TaskRecord{85070a0 #296 A=com.TaskB U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

Hist #0: ActivityRecord{8c75fd1 u0 com.example.malingyi.testtask2/.ActivityB t296}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{f5bf6cc 4090:com.example.malingyi.testtask2/u0a63}

Task id #295 //App2.MainActivity的任务栈com.example.malingyi.testtask2

TaskRecord{850e459 #295 A=com.example.malingyi.testtask2 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #0: ActivityRecord{52f95ba u0 com.example.malingyi.testtask2/.MainActivity t295}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{f5bf6cc 4090:com.example.malingyi.testtask2/u0a63}

Task id #294 //App1.ActivityA的任务栈

TaskRecord{5da881e #294 A=com.TaskA U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

Hist #0: ActivityRecord{43ed667 u0 com.example.malingyi.testtask1/.ActivityA t294}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{d226d15 4068:com.example.malingyi.testtask1/u0a60}

Task id #293 //App1.MainActivity的任务栈

TaskRecord{8b948ff #293 A=com.example.malingyi.testtask1 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #0: ActivityRecord{7669776 u0 com.example.malingyi.testtask1/.MainActivity t293}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{d226d15 4068:com.example.malingyi.testtask1/u0a60}

//正在最前台的Activity

mResumedActivity: ActivityRecord{8c75fd1 u0 com.example.malingyi.testtask2/.ActivityB t296}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

从上面的结果可以看出，由于给ActivityA、ActivityB指定了TaskAffinity、SingleTask，两个MainActivity跟随App默认的任务栈，因此每一个Activity都有一个Task任务栈。但还是会按照进程将他们分为两个APP。

注意看ProcessRecord字段会发现，App1.ActivityA和App1.MainActivity1是属于同一个Process(d226d15 4068:com.example.malingyi.testtask1/u0a60)。而App2.ActivityB和App2.MainActivity2是属于另一个   
Process(f5bf6cc 4090:com.example.malingyi.testtask2/u0a6)。

## ****（1）从App2.ActivatyB启动App1.ActivityA****

切换任务栈：

Stack #11:

Task id #294 //App1.ActivityA所在的任务栈移到了最前面

TaskRecord{5da881e #294 A=com.TaskA U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

Hist #0: ActivityRecord{43ed667 u0 com.example.malingyi.testtask1/.ActivityA t294}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{d226d15 4068:com.example.malingyi.testtask1/u0a60}

Task id #296 //App2.ActivityB所在的任务栈

TaskRecord{85070a0 #296 A=com.TaskB U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

Hist #0: ActivityRecord{8c75fd1 u0 com.example.malingyi.testtask2/.ActivityB t296}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{f5bf6cc 4090:com.example.malingyi.testtask2/u0a63}

Task id #295 //App2.MainActivity所在的任务栈

TaskRecord{850e459 #295 A=com.example.malingyi.testtask2 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #0: ActivityRecord{52f95ba u0 com.example.malingyi.testtask2/.MainActivity t295}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{f5bf6cc 4090:com.example.malingyi.testtask2/u0a63}

Task id #293 //App1.MainActivity所在的任务栈

TaskRecord{8b948ff #293 A=com.example.malingyi.testtask1 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #0: ActivityRecord{7669776 u0 com.example.malingyi.testtask1/.MainActivity t293}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{d226d15 4068:com.example.malingyi.testtask1/u0a60}

Running activities (most recent first):

TaskRecord{5da881e #294 A=com.TaskA U=0 sz=1}

Run #3: ActivityRecord{43ed667 u0 com.example.malingyi.testtask1/.ActivityA t294}

TaskRecord{85070a0 #296 A=com.TaskB U=0 sz=1}

Run #2: ActivityRecord{8c75fd1 u0 com.example.malingyi.testtask2/.ActivityB t296}

TaskRecord{850e459 #295 A=com.example.malingyi.testtask2 U=0 sz=1}

Run #1: ActivityRecord{52f95ba u0 com.example.malingyi.testtask2/.MainActivity t295}

TaskRecord{8b948ff #293 A=com.example.malingyi.testtask1 U=0 sz=1}

Run #0: ActivityRecord{7669776 u0 com.example.malingyi.testtask1/.MainActivity t293}

mResumedActivity: ActivityRecord{43ed667 u0 com.example.malingyi.testtask1/.ActivityA t294}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

这里我们看到，只有Task#294(App1.ActivityA)移动到了Task#296(App2.ActivityB)之前，App1.MainActivity所在的Task#293，并没有移动。

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB   
操作后的栈：App1.MainActivity——>App2.MainActivity——>App2.ActivityB——>App1.ActivityA

## ****（2）从App2.ActivityB启动App1.MainActivity****

切换任务栈：

Stack #12:

Task id #297 //App1.MainActivity所在的任务栈移到了最前面

TaskRecord{6b63e70 #297 A=com.example.malingyi.testtask1 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

Hist #0: ActivityRecord{f87eb51 u0 com.example.malingyi.testtask1/.MainActivity t297}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask1/.MainActivity bnds=[60,972][180,1092] (has extras) }

ProcessRecord{818d19c 4795:com.example.malingyi.testtask1/u0a60}

Task id #300 //App2.ActivityB所在的任务栈

TaskRecord{d3ee5e9 #300 A=com.TaskB U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

Hist #0: ActivityRecord{3df6276 u0 com.example.malingyi.testtask2/.ActivityB t300}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask2/.ActivityB }

ProcessRecord{d16f7a5 4819:com.example.malingyi.testtask2/u0a63}

Task id #299 //App2.MainActivity所在的任务栈

TaskRecord{90b126e #299 A=com.example.malingyi.testtask2 U=0 sz=1}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

Hist #0: ActivityRecord{9778bab u0 com.example.malingyi.testtask2/.MainActivity t299}

Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.example.malingyi.testtask2/.MainActivity bnds=[236,972][356,1092] (has extras) }

ProcessRecord{d16f7a5 4819:com.example.malingyi.testtask2/u0a63}

Task id #298 //App1.ActivityA所在的任务栈

TaskRecord{8688d0f #298 A=com.TaskA U=0 sz=1}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

Hist #0: ActivityRecord{f78be54 u0 com.example.malingyi.testtask1/.ActivityA t298}

Intent { flg=0x10000000 cmp=com.example.malingyi.testtask1/.ActivityA }

ProcessRecord{818d19c 4795:com.example.malingyi.testtask1/u0a60}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

我重新启动了一次APP，因此Task#ID改变了，但是他们初始的启动顺序没变。

初始栈： App1.MainActivity——>App1.ActivityA——>App2.MainActivity——>App2.ActivityB。   
操作后的栈：App1.ActivityA——>App2.MainActivity——>App2.ActivityB——>App1.MainActivity

我们仍然看到，只有Task#297 (App1.MainActivity)移动到了Task#300(App2.ActivityB)之前，App1.ActivityA所在的Task#298。

**到了这里，假设已经得到了实践的验证：**

* Android的APP切换，实际上是Activity以Task为单位来进行切换，将被选中的Activity所在的Task移动到Stack任务栈的最前端，然后在Task内部使用LaunchModel规则对Task内部的Activity进行操作。

# ****三、总结****

* **standard模式下，Activity会进入启动它的Activity所在的Task栈中，这个结论对于在启动其他App的Activity也成立。**
* **LaunchModel的启动规则，在跨App之间时启动Activity同样成立。**
* **在APP之间切换时。首先，会将切换的App所在的Task移动到最前面；然后，再使用LaunchModel的规则对Activity进行出栈入栈的操作。**
* **Android的APP切换，实际上是以Task为单位来进行切换，将被选中的Activity所在的Task移动到Stack任务栈的最前端，然后在Task内部根据Activity的启动模式，使用LaunchModel规则对Task内部的Activity进行操作。**