**目 录**

[**Activity启动模式 1**](#_Toc403051483)

[**文档修订记录 2**](#_Toc403051484)

[**android:launchMode启动模式 4**](#_Toc403051485)

[**任务栈 4**](#_Toc403051486)

[**Standard 4**](#_Toc403051487)

[**singleTop 5**](#_Toc403051488)

[**singleTask 5**](#_Toc403051489)

[**singleInstance 6**](#_Toc403051490)

[**Android开发\_后台任务task管理 6**](#_Toc403051491)

[**android:allowTaskReparenting 6**](#_Toc403051492)

[**android:alwaysRetainTaskState 7**](#_Toc403051493)

[**android:clearTaskOnLaunch 7**](#_Toc403051494)

[**android:finishOnTaskLaunch 7**](#_Toc403051495)

[**android:taskAffinity 7**](#_Toc403051496)

[**intent.setFlags()方法中的参数值含义： 8**](#_Toc403051497)

[**FLAG\_ACTIVITY\_CLEAR\_TOP 8**](#_Toc403051498)

[**FLAG\_ACTIVITY\_NEW\_TASK 9**](#_Toc403051499)

[**FLAG\_ACTIVITY\_NO\_HISTORY 9**](#_Toc403051500)

[**FLAG\_ACTIVITY\_SINGLE\_TOP 10**](#_Toc403051501)

## android:launchMode启动模式

#### 任务栈

每个应用都有一个任务栈，是用来存放Activity的，功能类似于函数调用的栈，先后顺序代表了Activity的出现顺序。如果启动Activity的顺序为Activity C -> Activity B -> Activity A，则任务栈的存在可以抽象的以图1表示。

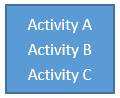


图1 任务栈

#### Standard

默认模式，不管将要启动的Activity X是否存在，都会在当前栈顶创建X的实例。

如果启动Activity的循序为Activity A -> Activity A -> Activity A，并且A的launchMode为standard,则整个流程及任务栈如图1所示：

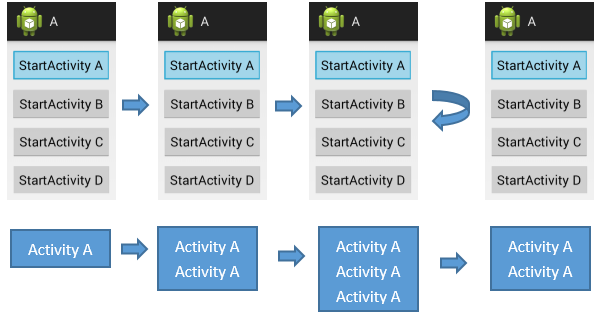


图2 Standard模式

#### singleTop

只有当将要启动的Activity X处于当前任务栈的栈顶，才不会新建X的实例。否则，都会在当前栈顶创建X的实例。图3列出了两种不同情况下启动一个launchMode为singleTop的Activity A，流程及任务栈分别如图1所示。

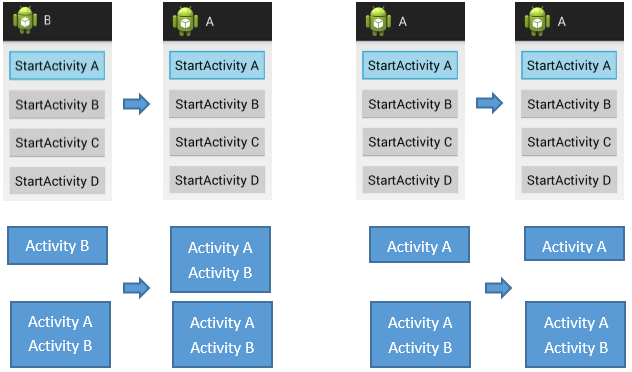


图3 singleTop模式

#### singleTask

在所有的任务栈中只能存在一份Activity X的实例，如果所有的任务栈中存在X的实例，就跳转到该任务栈的X，并且将X以上的Activity都出栈。

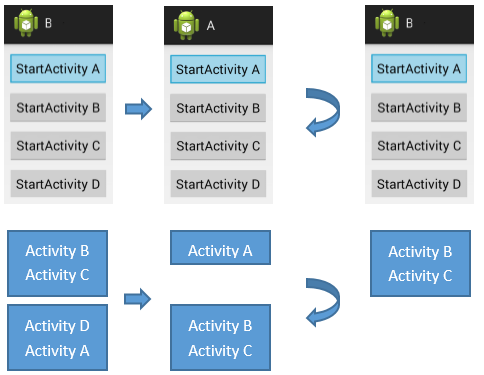


图4 singleTask模式

#### singleInstance

在所有的任务栈中只能存在一份Activity X的实例，并且X独占一个任务栈。在这种情况下能保证不管你是在哪一个Activity X启动了一个singleInstance模式的Activity A，都能从A返回到调用它的X。

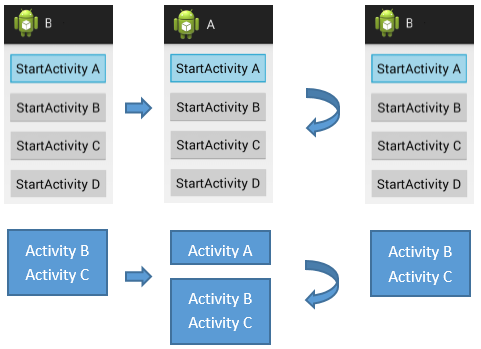


图5 singleInstance模式

## Android开发\_后台任务task管理

#### android:allowTaskReparenting

这个属性用来标记一个Activity实例在当前应用退居后台后，是否能从启动它的那个task移动到有共同affinity的task，“true”表示可以移动，“false”表示它必须呆在当前应用的task中，默认值为false。如果一个这个Activity的元素没有设定此属性，设定在上的此属性会对此Activity起作用。例如在一个应用中要查看一个web页面，在启动系统浏览器Activity后，这个Activity实例和当前应用处于同一个task，当我们的应用退居后台之后用户再次从主选单中启动应用，此时这个Activity实例将会重新宿主到Browser应用的task内，在我们的应用中将不会再看到这个Activity实例，而如果此时启动Browser应用，就会发现，第一个界面就是我们刚才打开的web页面，证明了这个Activity实例确实是宿主到了Browser应用的task内。

#### android:alwaysRetainTaskState

这个属性用来标记应用的task是否保持原来的状态，“true”表示总是保持，“false”表示不能够保证，默认为“false”。此属性只对task的根Activity起作用，其他的Activity都会被忽略。 默认情况下，如果一个应用在后台呆的太久例如30分钟，用户从主选单再次选择该应用时，系统就会对该应用的task进行清理，除了根Activity，其他Activity都会被清除出栈，但是如果在根Activity中设置了此属性之后，用户再次启动应用时，仍然可以看到上一次操作的界面。 这个属性对于一些应用非常有用，例如Browser应用程序，有很多状态，比如打开很多的tab，用户不想丢失这些状态，使用这个属性就极为恰当。

#### android:clearTaskOnLaunch

这个属性用来标记是否从task清除除根Activity之外的所有的Activity，“true”表示清除，“false”表示不清除，默认为“false”。同样，这个属性也只对根Activity起作用，其他的Activity都会被忽略。 如果设置了这个属性为“true”，每次用户重新启动这个应用时，都只会看到根Activity，task中的其他Activity都会被清除出栈。如果我们的应用中引用到了其他应用的Activity，这些Activity设置了allowTaskReparenting属性为“true”，则它们会被重新宿主到有共同affinity的task中。

#### android:finishOnTaskLaunch

这个属性和android:allowReparenting属性相似，不同之处在于allowReparenting属性是重新宿主到有共同affinity的task中，而finishOnTaskLaunch属性是销毁实例。如果这个属性和android:allowReparenting都设定为“true”，则这个属性好些。

## android:taskAffinity

Activity的归属，也就是Activity应该在哪个Task中，Activity与Task的吸附关系。我们知道，一般情况下在同一个应用中，启动的Activity都在同一个Task中，它们在该Task中度过自己的生命周期，这些Activity是从一而终的好榜样。

那么为什么我们创建的Activity会进入这个Task中？它们会转到其它的Task中吗？如果转到其它的Task中，它们会到什么样的Task中去？

解决这些问题的关键，在于每个Activity的taskAffinity属性。

每个Activity都有taskAffinity属性，这个属性指出了它希望进入的Task。如果一个Activity没有显式的指明该 Activity的taskAffinity，那么它的这个属性就等于Application指明的taskAffinity，如果 Application也没有指明，那么该taskAffinity的值就等于包名。而Task也有自己的affinity属性，它的值等于它的根 Activity的taskAffinity的值。

下面就通过一个例子来验证taskAffinity的作用。

第一种情况我们创建两个工程application1和application2。application1含有Activity1A和Activity1B两个Activity，默认为Activity1A，并且Activity1A的allowTaskReparenting为true。application2含有Activity2A和Activity2B两个Activity，默认为Activity2B。Activity 1A、1B、2A、2B的taskAffinity全都相同。  
    首先，我们启动application1,加载Activity1A，然后按Home键，使该任务栈进入后台。然后启动application2，默认加载Activity2B。   
    我们看到了什么现象？没错，本来应该是显示Activity2B，但是我们却看到了Activity1A。实际上Activity2B也被加载了，只是Activity1A重新宿主，所以看到了Activity1A。

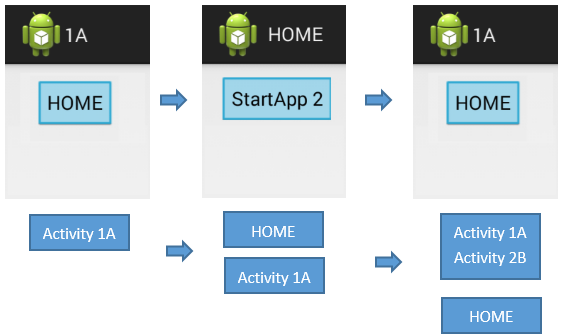


图6 taskAffinity实例1

下面我们又将结合上文提到的launchMode来验证taskAffinity的作用。

我们以singleTask为例。启动launchMode为singleTask的Activity A。先会检查是否存在与A的taskAffinity相同的任务栈。 如果不存在，那么就重新创建任务栈，并将A入栈。如果存在，那么又分以下两种情况。A已经在该任务栈中实例化，A不存在于该任务栈中。



图7 taskAffinity实例2

## **intent.setFlags()方法中的参数值含义：**

通过以上知识点的积累，我们就很好理解intent.setFlags()中参数的具体含义。

#### FLAG\_ACTIVITY\_CLEAR\_TOP

如现在的栈情况为：A B C 。C此时通过intent跳转到B，如果这个intent添加FLAG\_ACTIVITY\_CLEAR\_TOP标记，则栈情况变为：A B。如果没有添加这个标记，则栈情况将会变成：A B C B。也就是说，如果添加了FLAG\_ACTIVITY\_CLEAR\_TOP标记，并且目标Activity在栈中已经存在，则将会把位于该目标 activity之上的activity从栈中弹出销毁。这跟上面把B的Launch mode设置成singleTask类似。简而言之，跳转到的activity若已在栈中存在，则将其上的activity都销掉。



图8 FLAG\_ACTIVITY\_CLEAR\_TOP

#### FLAG\_ACTIVITY\_NEW\_TASK

如现在任务栈1的情况是：A B C。C通过intent跳转到D，并且这个intent添加了FLAG\_ACTIVITY\_NEW\_TASK标记，如果D这个Activity在 Manifest.xml中的声明中添加了Task affinity，系统首先会查找有没有和D的Task affinity相同的任务栈存在，如果有存在，将D压入那个任务栈，如果不存在则会新建一个D的affinity的任务栈将其压入。如果D的Task affinity默认没有设置，则会把其压入任务栈1，变成：A B C D，这样就和不加FLAG\_ACTIVITY\_NEW\_TASK标记效果是一样的了。（可以参考图7的流程）

注意如果试图从非activity的非正常途径启动一个 activity，比如从一个service中启动一个activity，则intent比如要添加FLAG\_ACTIVITY\_NEW\_TASK标记。简而言之，跳转到的activity根据情况，可能压在一个新建的栈中。

#### FLAG\_ACTIVITY\_NO\_HISTORY

如现在栈情况为：A B。B通过intent跳转到C，这个intent添加FLAG\_ACTIVITY\_NO\_HISTORY标志，则此时界面显示C的内容，但是它并不会压 入栈中。如果按返回键，返回到B，栈的情况还是：A B。如果此时C中又跳转到D，栈的情况变为：A B D，此时按返回键会回到B，因为C根本就没有被压入栈中。简而言之，跳转到的activity不压在栈中。

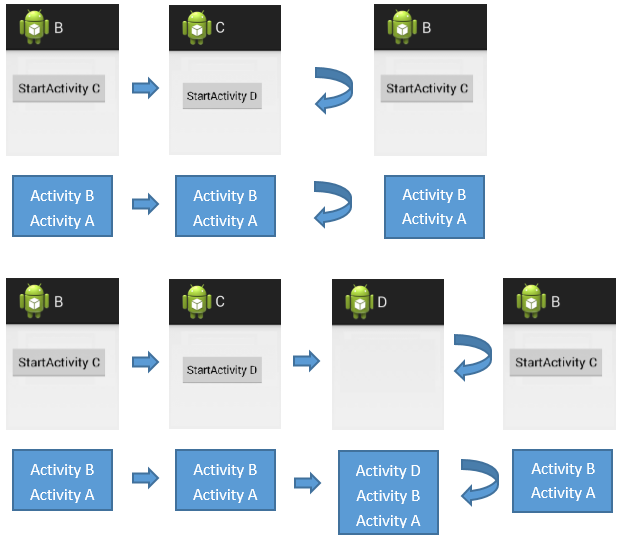


图9 FLAG\_ACTIVITY\_NO\_HISTORY

#### FLAG\_ACTIVITY\_SINGLE\_TOP

和Activity的Launch mode的singleTop类似。如果某个intent添加了这个标志，并且这个intent的目标activity就是栈顶的activity，那么将不会新建一个实例压入栈中。简而言之，目标activity已在栈顶则跳转过去，不在栈顶则在栈顶新建activity。