Activity

Activity正常生命周期

- onCreate
- onRestart
- onStart
- onResume
- onPause
- onStop
- onDestroy
- View的onAttachedToWindow和onDetachedFromWindow的调用时机?
- 两个问题：
- Activity启动时序图(android-26)
- Activity启动大致流程(android-28)

Activity异常情况生命周期

- 情况1：资源相关的系统配置发生改变导致Activity被杀死并重新创建
  - 旋转屏幕若不想销毁Activity
- 情况2：资源内存不足导致低优先级的Activity被杀死

Activity的启动模式

- Standard标准模式
- SingleTop栈顶复用模式
- SingTask栈内复用模式
- SingleInstance单实例模式
- Activity的Flags

IntentFilter的匹配规则

- action的匹配规则
- category的匹配规则
- data匹配规则

ActivityRecord、TaskRecord、ActivityStack、ActivityDisplay、ActivityStackSupervisor。它们是什么？可以用...

- 前台任务栈与后台任务栈

ActivityRecord：Activity信息记录者
- ActivityRecord的创建
- Token的利用

TaskRecord：任务栈记录者
- 示例分析
- TaskRecord的创建

ActivityStack

ActivityStackSupervisor

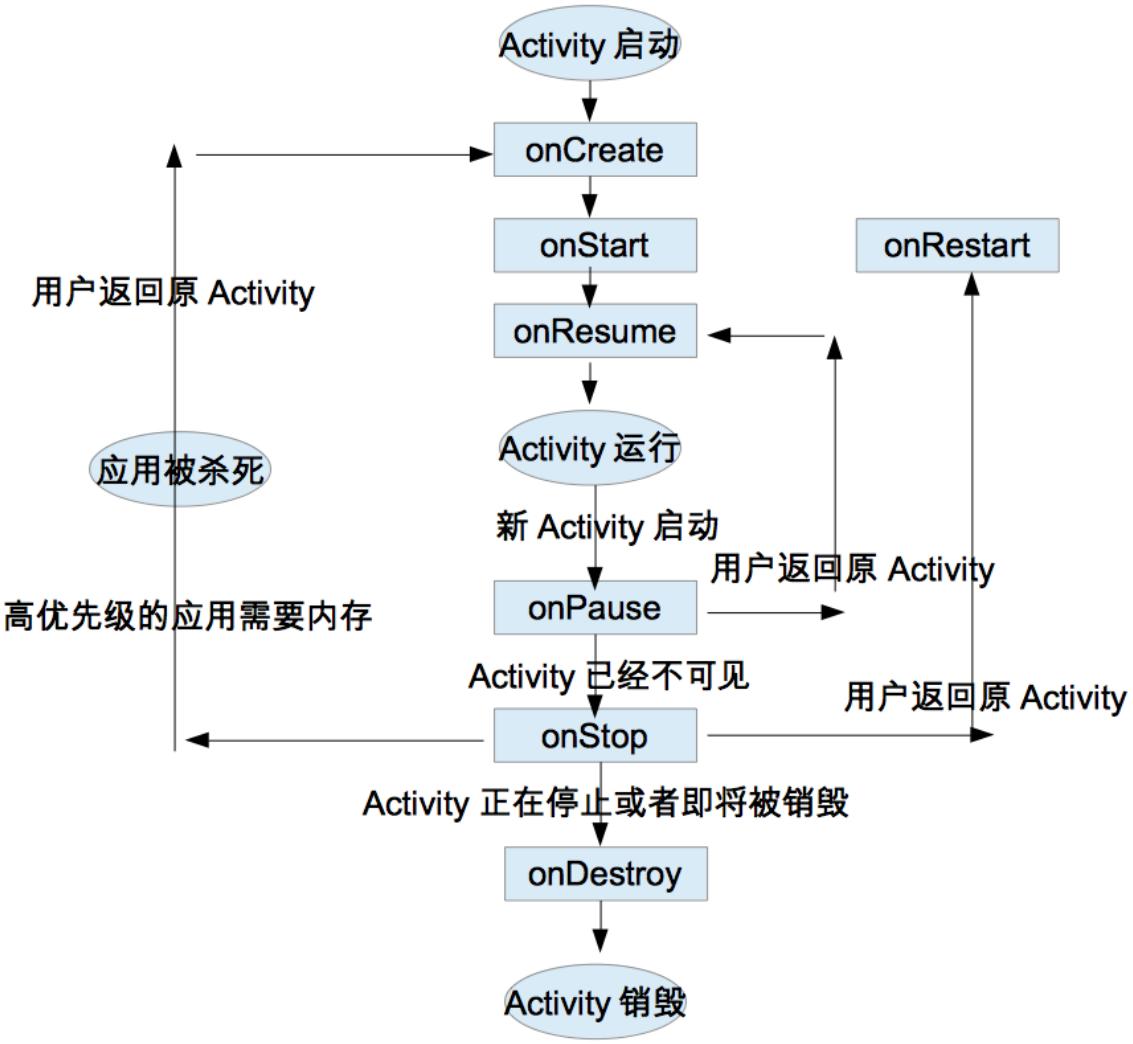
Activity栈结构体的组成关系

场景任务栈分析
- 桌面
- 从桌面启动一个Activity
- 默认模式从A启动B
- 从A启动B创建新栈

启动流程任务栈分析
- 启动流程

启动模式任务栈分析
- standerd
- singleTop
- singleTask
- singleInstance

Activity正常生命周期



onCreate

表示Activity正在被创建，这是生命周期的第一个方法，在这个方法中，我们可以做一些初始化的工作，比如调用onContentView去加载界面布局资源，初始化Activity所需数据等。
若此时调用onContentView方法，则会初始化DecorView根布局，并通过LayoutInflater解析XML布局文件中的节点View添加addView至DecorView的子布局**mContentParent**中。

onRestart

表示Activity正在重新启动，一般情况下，当当前Activity从不可见重新变为可见时，onRestart就会被调用，这种情况一般是用户行为所导致的，比如用户按home键切换到桌面或者用户打开了一个新的Activity，这时当前的Activity就会被暂停，也就是onPause和onStop方法被执行了，接着用户又回到了这个Activity，就会出现这种情况。

onStart

表示Activity正在被启动，即将开始，这个时候Activity已经可见了，但是还没有出现在前台，还无法和用户交互，这个时候我们可以理解为Activity已经启动了，但是我们还没有看见。

onResume

表示Activity已经可见了，并且出现在前台，并开始活动了，要注意这个和onStart的对比，这两个都表示Activity已经可见了，但是onStart的时候Activity还处于后台，onResume的时候Activity才显示到前台。Activity的每个生命周期都是ActivityThread中mH的消息处理所执行，也是在而此生命周期则是来源于ActivityThread中的handleResumeActivity方法。而WindowManager和DecorView的关联也是在handleResumeActivity方法中处理的--wm.addView(decor, 1)，显示DecorView r.activity.makeVisible()和View绘制// 执行ViewRootImpl的setView--View绘制的起点;root.setView(view, wparams, panelParentView)均在此阶段执行，root.setView中通过调用requestLayout()方法触发View的mesure、layout和draw等方法。

onPause

表示Activity正在停止，正常情况下，紧接着onStop就会被调用，在特殊情况下，如果这个时候再快速的回到当前Activity，那么onResume就会被调用。这个情况比较极端，用户操作很难重现这个场景，此时可以做一些数据存储，停止动画等工作，但是注意不要太耗时了，因为这样会影响到新的Activity的显示，onPause必须先执行完，新Activity的onResume才会执行。

onStop

表示Activity即将停止，同样可以做一些轻量级的资源回收，但是不要太耗时了。

onDestroy

表示Activity即将被销毁，这是Activity生命周期的最后一个回调，在这里我们可以做一些最后的回收工作和资源释放。

View的onAttachedToWindow和onDetachedFromWindow的调用时机?

1. onAttachedToWindow的调用：`ActivityThread.handleResumeActivity`的过程中，会将Activity的DecorView添加到`WindowManager`中，而`WindowManager`实际上只是个继承了`ViewManager`的接口。当在`ActivityThread.handleResumeActivity()`方法中调用`WindowManager.addView()`方法时，最终是调去了WindowManagerGlobal.addView()，其中`root.setView(view, wparams, panelParentView);`，正是这行代码将调用流程转移到了`ViewRootImpl.setView()`里面，继而触发了`ViewRootImpl.performTraversals()`方法，开启了View从无到有要经历的3个阶段(measure, layout, draw)。而正是`performTraversals()`方法中host.dispatchAttachedToWindow(mAttachInfo, 0);开启了调用onAttachedToWindow的时机。host实

际上是DecorView，dispatch方法将这个调用沿着View tree分发了下去，ViewGroup先是调用自己的`onAttachedToWindow()`方法，再调用其每个child的`onAttachedToWindow()`方法，这样此方法就在整个view树中遍布开了，visibility并不会对这个方法产生影响。

2. onDetachedFromWindow的调用：和attched对应的，detached的发生是从Activity的销毁开始的。

```
ActivityThread.handleDestroyActivity() --> WindowManager.removeViewImmediate()
--> WindowManagerGlobal.removeViewLocked()方法
--> ViewRootImpl.die() --> doDie() --> ViewRootImpl.dispatchDetachedFromWindow()
```

最终会调用到View层次结构的dispatchDetachedFromWindow方法，和attached类似会沿着View Tree遍历调用所有子View的onDetachedFromWindow方法。

两个问题：

- onStart和onResume，onPause和onStop从描述上都差不多，对我们来说有什么实质性的不同呢?
- 假设当前Activity为A，如果用户打开了一个新的Activity为B，那么B的onResume和A的onPause谁先执行?

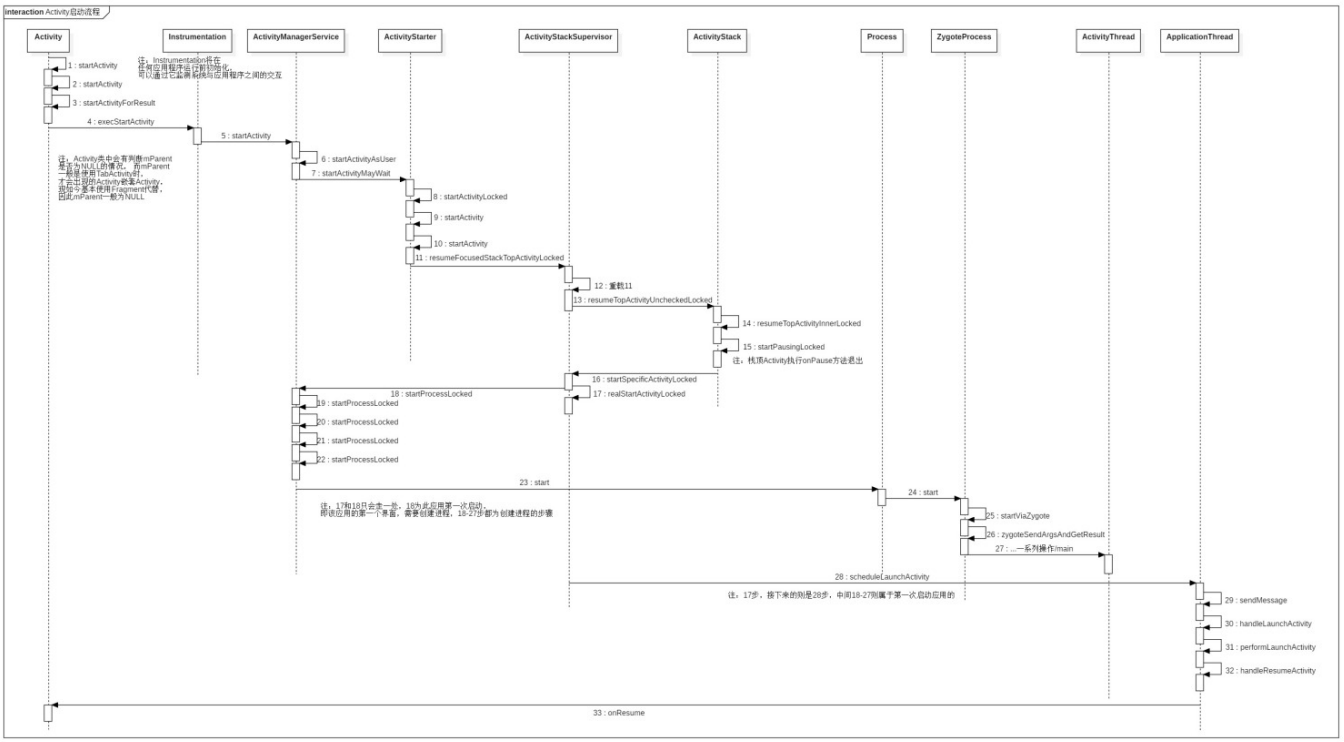
```
问题1：从实际使用过程来说， onStart和onResume，onPause和onStop看起来的确差不多，
甚至我们可以只保留其中的一对，比如只保留onStart和onStop，既然如此，那为什么Android系统还会提供看起来重复的接口呢?
根据上面的分析，我们知道，这两个配对的回调分别代表不同的意义，onStart和onStop是从Activity是否可见这个角度来回调的，除了这种区别，在实际的使用中，没有其他明显的区别

问题2：从源码的角度来分析以及得到解释了，关于Activity的工作原理会在本书后续章节进行讲解，
这里我们大致的了解即可，从Activity的启动过程来看，我们来看一下系统的源码，
Activity启动过程的源码相当复杂，设计到了Instrumentation,Activit和ActivityManagerService(AMS)，
这里不详细分析这一过程，简单理解，启动Activity的请求会由Instrumentation 来处理，然后它通过Binder向AMS发请求，
AMS内部维护着一个ActivityStack，并负责栈内的Activity的状态同步，AMS通过ActivityThread去同步Activity的状态
从而完成生命周期方法的调用，在ActivityStack中的resumeTopActivityLnnerLocked方法中可以了解到，
在新Activity启动之前，栈顶的Activity需要先onPause后，新的Activity才能启动，最终，
```

在ActvityStackSupervisor中的realStartActivityLocked方法中，最后会调用到ActivityThread的scheduleLaunchActivity方法，

而scheduleLaunchActivity方法最终会完成生命周期的调用过程，因此可以得出结论，是旧Activity县onPause，然后新的Activityy再启动。

Activity启动时序图(android-26)



Activity启动大致流程(android-28)

```
1. startActivity--> Activity(startActivity-> startActivityForResult)
    --> Instrumentation(execStartActivity)--> ActivityManager(getService.startActivity)
  --> ActivityManagerService(startActivity)--> ActivityStartController(obtainStarter工厂方法模式)
  --> ActivityStarter(execute--> startActivityMayWait--> startActivity--> startActivityUnchecked)
2. --> ActivityStackSupervisor(resumeTopActivityUncheckedLocked)
    --> ActivityStack(resumeTopActivityUncheckedLocked--> resumeTopActivityInnerLocked)
  --> ActivityStartSupervisor(startSpecificActivityLocked--> realStartActivityLocked)
  --> ClientLifecycleManager(scheduleTransation)--> ClientTransation(schedule)
3. --> ActivityThread(ApplicationThread(scheduleTransation)--> sched
```

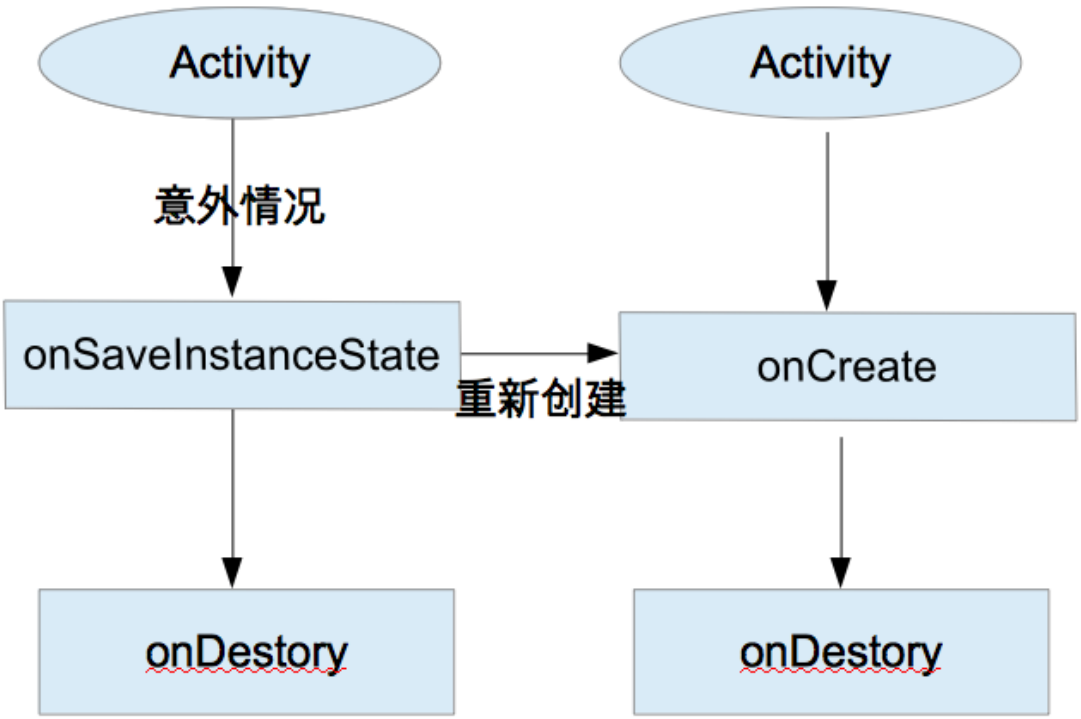
```
uleTransation)
    --> ClientTransationHandler(scheduleTransation--> sendMessage(ActivityThread.H.EXECUTE_TRANSATION))
    --> ActivityThread(H(handleMessage))--> TransationExceutor(execute)
  --> LaunchActivityItem(excute)--> ClientTransationHandler(handleLaunchActivity)
4(最后使用反射创建Activity). --> ActivityThread(handleLaunchActivity--> performLaunchActivity)
    --> Instrumentation(newActivity--> getFactory(pkg))--> ActivityThread(peekPackageInfo)
  --> LoadedApk(getAppFactory)--> AppComponentFactory(instantiateActivity(cl, className, intent)
  --> (Activity) cl.loadClass(className).newInstance())--> Activity(performCreate--> onCreate)
```

Activity异常情况生命周期

Activity除了受用户操作导致的正常生命周期的调度，同时还会存在一些异常的情况，比如当资源相关的系统配置发生改变以及系统内存不足的时候，Activity就有可能被杀死。

情况1：资源相关的系统配置发生改变导致Activity被杀死并重新创建

比如：旋转屏幕



当系统配置发生改变的时候，Activity会被销毁，其onPause，onStop,onDestroy均会被调用，同时由于Activity是异常情况下终止的，系统会调用onSaveInstanceState来保存当前Activity的状态，这个方法调用的时机是在onStop之前，他和onPause没有既定的时序关系，他即可能在onPause之前调用，也有可能在之后调用，需要强调的是，这个方法只出现在Activity被异常终止的情况下，正常情况下是不会走这个方法的吗，当我们onSaveInstanceState保存到Bundler对象作为参数传递给onRestoreInstanceState和onCreate方法，因此我们可以通过onRestoreInstanceState和onCreate方法来判断Activity是否被重建。如果被重建了，我们就取出之前的数据恢复，从时序上来说，onRestoreInstanceState的调用时机应该在onStart之后。

同时我们要知道，在onSaveInstanceState和onRestoreInstanceState方法中，系统自动为我们做了一些恢复工作，当Activity在异常情况下需要重新创建时，系统会默认我们保存当前的Activity视图架构，并且为我们恢复这些数据，比如文本框中用户输入的数据，ListView滚动的位置，这些View相关的状态系统都会默认恢复，具体针对某一个特定的View系统能为们恢复那些数据？我们可以查看View的源码，和Activity一样，每一个View都有onSaveInstanceState和onRestoreInstanceState这两个方法，看一下他们的实现，就能知道系统能够为每一个View恢复数据。

关于保存和恢复View的层次结构，系统的工作流程是这样的：首先Activity被意外终止时，Activity会调用onSaveInstanceState去保存数据，然后Activity会委托Window去保存数据，接着Window再委托上面的顶级容器去保存数据，顶级容器是一个ViewGroup，一般来说他可能是一个DecorView,最后顶层容器再去一一通知他的子元素来保存数据，这样整个数据保存过程就完成了，可以发现，这是一种典型的委托思想，上层委托下层，父容器委托子容器，去处理一件事件，这种思想在Android 中有很多的应用。

旋转屏幕若不想销毁Activity

```
// AndroidManifest.xml中配置相应Activity
android:configChanges="keyboardHidden|orientation|screenSize"

// 配置发生改变：包括屏幕旋转
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    // 可以监听旋转，此回调代码配置改变，可在此回调中，重新刷新部分数据
}

// 监听屏幕旋转角度
mOrientationListener = new OrientationEventListener(this, SensorManager.SENSOR_DELAY_NORMAL) {
    @Override
    public void onOrientationChanged(int orientation) {
```

```
15          Log.e(DEBUG_TAG, "Orientation changed to " + orientation);
16          Log.e(DEBUG_TAG, "Orientation changed  width to " + ScreenUt
   il.getScreenWidth());
17          Log.e(DEBUG_TAG, "Orientation changed height to " + ScreenUt
   il.getScreenHeight());
18      }
19  };
20  if (mOrientationListener.canDetectOrientation()) {
21      Log.i(DEBUG_TAG, "Can detect orientation");
22      mOrientationListener.enable();
23  } else {
24      Log.i(DEBUG_TAG, "Cannot detect orientation");
25      mOrientationListener.disable();
26  }
```

情况2：资源内存不足导致低优先级的Activity被杀死

Activity优先级：

- 1.前台Activity:正在和用户交互的Activity，优先级最高
- 2.可见但非前台Activity:比如对话框，导致Activity可见但是位于后台无法和用户直接交互
- 3.后台Activity:已经被暂停的Activity，比如执行了onStop，优先级最低

当系统内存不足的时候，系统就会按照上述优先级去杀死目标Activity所在的进程，并且在后续通过onSaveInstanceState和onRestoreInstanceState来存储和恢复数据，如果一个进程中没有四大组件在执行，那么这个进程将很快被系统杀死，因此，一些后台工作不适合脱离四大组建而独立运行在后台中，这样进程很容易就被杀死了，比较好的方法就是将后台工作放在Service中从而保证了进程有一定的优先级，这样就不会轻易的被杀死。

表 1-1　configChanges 的项目和含义

项　目	含　义
mcc	SIM 卡唯一标识 IMSI（国际移动用户识别码）中的国家代码，由三位数字组成，中国为 460。此项标识 mcc 代码发生了改变
mnc	SIM 卡唯一标识 IMSI（国际移动用户识别码）中的运营商代码，由两位数字组成，中国移动 TD 系统为 00，中国联通为 01，中国电信为 03。此项标识 mnc 发生改变
locale	设备的本地位置发生了改变，一般指切换了系统语言
touchscreen	触摸屏发生了改变，这个很费解，正常情况下无法发生，可以忽略它
keyboard	键盘类型发生了改变，比如用户使用了外插键盘
keyboardHidden	键盘的可访问性发生了改变，比如用户调出了键盘
navigation	系统导航方式发生了改变，比如采用了轨迹球导航，这个有点费解，很难发生，可以忽略它
screenLayout	屏幕布局发生了改变，很可能是用户激活了另外一个显示设备
fontScale	系统字体缩放比例发生了改变，比如用户选择了一个新字号
uiMode	用户界面模式发生了改变，比如是否开启了夜间模式（API 8 新添加）
orientation	屏幕方向发生了改变，这个是最常用的，比如旋转了手机屏幕
screenSize	当屏幕的尺寸信息发生了改变，当旋转设备屏幕时，屏幕尺寸会发生变化，这个选项比较特殊，它和编译选项有关，当编译选项中的 minSdkVersion 和 targetSdkVersion 均低于 13 时，此选项不会导致 Activity 重启，否则会导致 Activity 重启（API 13 新添加）
smallestScreenSize	设备的物理屏幕尺寸发生改变，这个项目和屏幕的方向没关系，仅仅表示在实际的物理屏幕的尺寸改变的时候发生，比如用户切换到了外部的显示设备，这个选项和 screenSize 一样，当编译选项中的 minSdkVersion 和 targetSdkVersion 均低于 13 时，此选项不会导致 Activity 重启，否则会导致 Activity 重启（API 13 新添加）
layoutDirection	当布局方向发生变化，这个属性用的比较少，正常情况下无须修改布局的 layoutDirection 属性（API 17 新添加）

从上面的属性中我们可以知道，如果我们没有在Activity的configChanges中设备属性的话，当系统发生改变后就会导致Activity重新被创建，上面表格中的项目很多，但是我们常用的只有locale,orientation,keyboardHidden这三个选项。

Activity的启动模式

Standard标准模式

这也是系统的默认模式，每次启动一个Activity都会重新创建一个实例。

ApplicationContext去启动standard模式的Activity的时候就会报错：这是因为我们的standard模式的Activity默认会进入启动它的Activity所属的任务栈中，但是由于非Activity类型的Context（如ApplicationContext）并没有所谓的任务栈，所以这就有问题了，解决这个问题，就是待启动Activity指定FLAG_ACTIVITY_TASK标记位，这样启动的时候就会为他创建一个新的任务栈，这个时候待启动Activity实际上是以singleTask模式启动的。

SingleTop栈顶复用模式

这个模式下，如果新的Activity已经位于任务栈的栈顶，那么此Activity不会被重新创建，同时他的onNewIntent方法会被调用，通过此方法的参数我们可以取出当前请求的信息，需要注意的是，这个

Activity的onCreate,onStart不会被系统调用，因为他并没有发生改变，如果新Activity已存在但不是在栈顶，那么新Activity则会重新创建。

SingTask栈内复用模式

这是一种单实例模式，在这种模式下，只要Activity在一个栈内存在，那么多次启动此Activity都不会创建实例，和singTop一样，系统也会回调其onNewIntent方法，具体一点，当一个具有singleTask模式的Activity请求启动后，比如Activity A，系统首先会去寻找是否存在A想要的任务栈，如果不存在，就重新创建一个任务栈，然后创建A的实例把A放进栈中，如果存在A所需要的栈，这个时候就要看A是否在栈中有实例存在，如果实例存在，那么系统就会把A调到栈顶并调用它的onNewIntent方法，如果实例不存在，就创建A的实例并且把A压入栈中。若Activity A存在任务栈中，而Activity A之上还有其它Activity，则其它Activity先出栈，直到Activity A出现在栈顶。

SingleInstance单实例模式

这是一种加强的singleTask的模式，他除了具有singleTask的所有属性之外，还加强了一点，那就是具有此模式下的Activity只能单独的处于一个任务栈中，换句话说，比如Activity A是singleInstance模式，当A启动的时候，系统会为创建创建一个新的任务栈，然后A独立在这个任务栈中，由于栈内复用的特性，后续的请求均不会创建新的Activity，除非这个独特的任务栈被系统销毁了。

问题1：我们假设目前有两个任务栈，前台任务栈的情况为AB，而后台任务栈的情况是CD，这里假设CD的启动模式都是singleTask，现在请求启动D，那么整个后台任务站都会被切换到前台，这个时候整个后退列表变成了ABCD，当用户按back键的时候，列表中的Activity会一一出栈。

问题2：

在singkleTask启动模式中，多次提到了某个Activity所需的任务栈，什么是Activity所需的任务栈尼？这要从一个参数说起：TaskAffinity,可以翻译成任务相关性，这个参数标示了一个Activity所需要的任务栈的名字默认情况下，所有的Activity所需要的任务栈的名字为应用的包名，当然，我们可以为每个Activity都单独指定TaskAffinity，这个属性值必须必须不能和包名相同，否则就相当于没有指定，TaskAffinity属性主要和singleTask启动模式或者allowTaskReparenting属性配合使用，在其他状况下没有意义，另外，任务栈分为前台任务栈和后台任务栈，后台任务栈中的Activity位于暂停状态，用户可以通过切换将后台任务栈再次调为前台。当TaskAffinity和singleTask启动模式配对使用的时候，他是具有该模式Activity目前任务栈的名字，待启动的Activity会运行在名字和TaskAffinity相同的任务栈中。

当TaskAffinity和allowTaskReparentiing结合的时候，这种情况比较复杂，会产生特殊的效果，当一个应用A启动了应用B的某一个Activity后，如果这个Activity会直接从应用A的任务栈转移到应用B的任务栈中，这还是很抽象的，再具体点，比如现在有2个应用A和B，A启动了B的一个Activity C ，然后按Home键回到桌面，然后再单击B的桌面图标，这个时候并不是启动； B的主Activity，而是重新显示了已经被应用A启动的Activity C,或者说，C从A的任务栈转移到了B的任务栈中，可以这么理解，由于A启动了C，这个时候C只能运行在A的任务栈中，但是C属于B应用，正常情况下，他的TaskAffinity值肯定不可能和A的任务栈相同（因为包名不同），所以，当B启动后，B会创建自己的任务栈，这个时候系统发现C原本所想要的任务栈已经被创建出来了，所以就把C从A的任务栈中转移过来，这种情况读者可以写一个例子测试一下，这里就不做演示了

如何给Activity指定启动模式?

第一种：通过清单文件为Activity指定：android:launchMode="singleTask"

指定taskAffinity：android:taskAffinity="com.xxx.xxx"

第二种：通过intent的标志位为Activity指定启动模式：

```
Intent intent = new Intent();
intent.setClass(this,SecondActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);
```

这两种方式都可以为Activity指定启动模式，但是二者还是有一些区别的，首先，优先级上，第二种比第一种高，当两种同时存在的时候，以第二种为准，其次，上述两种方式在限定范围内有所不同，比如，第一种方式无法直接为Activity设置FLAG_ACTIVITY_CLEAR_TOP标识，而第二种方式无法指定singleInstance模式。

Activity的Flags

如果`startActivity()`时往`Intent`中加入相应的标志来指定启动模式，这种方式的优先级会比在`AndroidManifest`中定义的优先级高;但是`AndroidManifest`中只能定义四种启动方式：`standard`、`singleTop`、`singleTask`、`singleInstance`，而`Intent`的`flag`则有很多种。具体的可以看看文档，我们这里看看部分`flag`：

- FLAG_ACTIVITY_NEW_TASK ： 跟launchMode中的singleTask一样。
- FLAG_ACTIVITY_SINGLE_TOP ： 跟launchMode中的singleTop一样。
- FLAG_ACTIVITY_CLEAR_TOP ： launchMode中没有对应的值，如果要启动的Activity已经存在于栈中，则将所有位于它上面的Activity出栈。singleTask默认具有此标记位的效果。
- FLAG_ ACTIVITY_ EXCLUDE_ FROM _ RECENTS：具有此标记位的Activity，不会出现在历史Activity的列表当中，当某种情况下我们不希望用户通过历史列表回到我们的Activity的时候就使用这个标记位了，他等同于在XML中指定Activity的属性：android:excludeFromRecents="true"。

IntentFilter的匹配规则

启动Activity分为两种，显示调用和隐式调用，二者的区别这里就不多讲了，显示调用需要明确的指定被启动对象的组件信息，包括包名和类名，而隐式意图则不需要明确指定调用信息，原则上一个intent不应该即是显式调用又是隐式调用，如果二者共存的话以显式调用为主，显式调用很简单，这里主要介绍隐式调用，隐式调用需要intent能够匹配目标组件的IntentFilter中所设置的过滤信息，如果不匹配将无法启动目标Activity，IntentFilter中的过滤信息有action、category、data。

为了匹配过滤列表，需要同时匹配过滤列表中的action,category,data信息，否则匹配失败，一个过滤列表中的action,category,data可以有多个，所有的action,category,data分别构成不同类别，同一类型的信息共同约束当前类别的匹配过程，只有一个intent同时匹配action类别,category类别,data类别才算是匹配完成，只有完全匹配才能成功启动目标Activity，另外一点，一个Activity钟可以有多个intent-filter,一个intent只要能匹配一组intent-filter即可成功启动Activity。

action的匹配规则

action是一个字符串，系统预定了一些action,同时我们也可以在应用中定义自己的action,action的匹配规则是intent中的action必须能够和过滤规则中的action匹配，这里说的匹配是指action的字符串值完全一样，一个过滤规则中的可以有多个action,那么只要intent中的action能够和过滤规则匹配成功，针对上面的过滤规则，需要注意的是，intent如果没有指定action，那么匹配失败，总结一下，action的匹配需求就是intent中的action存在且必和过滤规则一样的action，这里需要注意的是他和category匹配规则的不同，另外，action区分大小写，大小写不同的字符串匹配也会失败。

category的匹配规则

category是一个字符串，系统预定义了一些category，同时我们也可以在应用中定义自己的category。category的匹配规则和action不同，它要求Intent中如果含有category，那么所有的category都必须和过滤规则中的其中一个category相同。换句话说，Intent如果出现了category，不管有几个category，对于每个category来说，它必须是过滤规则中已经定义的category。当然，Intent中可以没有category，如果没有category的话，按照上面的描述，这个Intent仍然可以匹配成功。这里要注意下它和action匹配过程的不同，action
是要求Intent中必须有一个action且必须能够和过滤规则中的某个action相同，而category要求Intent可以没有category，但是如果你一旦有category，不管有几个，每个都要能和过滤规则中的任何一个category相同。为了匹配前面的过滤规则中的category，我们可以写出下面的Intent，intent.addcategory ("com.ryg.category.c")或者Intent.addcategory ("com.ryg.rcategory.d")亦或者不设category。为什么不设置category也可以匹配呢？原因是系统在调用startActivity或者startActivityForResult的时候会默认为Intent加上“android.intent.category.DEFAULT”这个category，所以这个category就可以匹配前面的过滤规则中的第三个category。同时，为了我们的activity能够接收隐式调用，就必须在intent-filter中指定“android intent categor.DEFAULT”这个category，原因刚才已经说明了。

data匹配规则

data由两部分组成，mimeType和URI，前者是媒体类型，比如image/jpeg等，可以表示图片等，而URI包含的数据可就多了，下面的URI的结构：

<**scheme**>://<**host**>"<**port**>/[<**path**>|<**pathPrefix**>|<**pathPattern**>]

URI举例说明：

http://www.baidu.com:80/search/info

content://com.liuguilin.project:200/folder/subfolder/etc

ActivityRecord、TaskRecord、ActivityStack、ActivityDisplay、ActivityStackSupervisor。它们是什么？可以用来干什么？以及怎么干的？

- 常规回答：*ActivityStackSupervisor管理ActivityDisplay，ActivityDisplay管理ActivityStack、ActivityStack管理TaskRecord、TaskRecord管理ActivityRecord。*

- **ActivityRecord**：ActivityRecord是Activity在system_server进程中的镜像，Activity实例与ActivityRecord实例一一对应。ActivityRecord用来存储Activity的信息，如所在的进程名称，应用的包名，所在的任务栈的taskAffinity等。
- **TaskRecord**：TaskRecord表示任务栈，用于记录activity开启的先后顺序。其所存放的Activity是不支持重新排序的，只能根据压栈和出栈操作更改Activity的顺序。有了TaskRecord，Android系统才能知道当一个Activity退出时，接下来该显示哪一个Activity。
- **ActivityStack**：ActivityStack这个类在名字上给人很大的误解，Stack是栈的意思（Heap是堆的意思），那ActivityStack就表示“Activity的栈”？其实不是。从下面的代码中可以看出ActivityStack维护的是TaskRecord的列表。而且该列表也不是栈结构，列表中的TaskRecord可以重排顺序。
- **ActivityDisplay**：ActivityDisplay表示一个屏幕，Android支持三种屏幕：主屏幕，外接屏幕（HDMI等），虚拟屏幕（投屏）。一般情况下，即只有主屏幕时，ActivityStackSupervisor与ActivityDisplay都是系统唯一。ActivityDisplay是ActivityStackSupervisor的内部类，它相当于一个工具类，封装了移除和添加ActivityStack的方法。
- **ActivityStackSupervisor**：ActivityStackSupervisor是ActivityStack的管理者，内部管理了mHomeStack、mFocusedStack和mLastFocusedStack三个ActivityStack。其中，mHomeStack管理的是Launcher相关的Activity栈， stackId为0；mFocusedStack管理的是当前显示在前台Activity的Activity栈；mLastFocusedStack管理的是上一次显示在前台Activity的Activity栈。ActivityDisplay的添加和移除ActivityStack的方法被封装进了ActivityContainer类中，ActivityStackSupervisor调用ActivityContainer的attachToDisplayLocked与detachLocked对ActivityStack列表进行重排序，将任务栈从后台切换至前台。
- ***注：任务栈TaskRecord和虚拟机栈不一致，虚拟机栈以方法为单位，任务栈以ActivityRecord(即Activity)为单位***

前台任务栈与后台任务栈

ActivityRecord：Activity信息记录者

Activity的信息记录在ActivityRecord对象，并通过通过成员变量task指向TaskRecord：

- ProcessRecord app // 跑在哪个进程
- TaskRecord task // 跑在哪个task
- ActivityInfo info // Activity信息
- int mActivityType // Activity类型
- ActivityState state // Activity状态
- ApplicationInfo appInfo // 跑在哪个app

- ComponentName realActivity // 组件名
- String packageName // 包名
- String processName // 进程名
- int launchMode // 启动模式
- int userId // 该Activity运行在哪个用户id
- **final** IApplicationToken.Stub **appToken**; *// window manager token*

ActivityRecord的创建

`startActivity()`时会创建一个`ActivityRecord`：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java
Activity启动过程中：startActivity--> AMS--> ActivityStarter#startActivity
class ActivityStarter {
    private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,
                              String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
                              IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
                              IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,
                              String callingPackage, int realCallingPid, int realCallingUid, int startFlags,
                              ActivityOptions options, boolean ignoreTargetSecurity, boolean componentSpecified,
                              com.android.server.am.ActivityRecord[] outActivity, TaskRecord inTask) {

        //其他代码略

        ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid, callingUid,
                callingPackage, intent, resolvedType, aInfo, mService.getGlobalConfiguration(),
                resultRecord, resultWho, requestCode, componentSpecified, voiceSession != null,
                mSupervisor, options, sourceRecord);

```

```
19          //其他代码略
20      }
21  }
```

ActivityStarter类：ActivityStarter是Android7.0新加入的类，它是加载Activity的控制类，会收集所有的逻辑来决定如何将Intent和Flags转换为Activity，并将Activity和Task(TasRecord)以及Stack(ActivityStack)相关联。

ActivityRecord是system_server进程中的对象，ActivityClientRecord和Activity都是App进程中的对象，那么三者之间是如何建立起一一对应的关系呢?

从上面成员中可以看出ActivityRecord有一个成员变量appToken，类型为Token，继承了IApplicationToken.Stub。很明显，appToken可以跨进程传输（因为IApplicationToken.Stub继承了Binder，appToken本身就是个Binder对象嘛）。而且Token中还持有ActivityRecord的弱引用，也就是说可以通过appToken找到ActivityRecord。
因此只要把appToken传到App进程中，并赋值给ActivityClientRecord和Activity，那么三者的一一对应关系就建立来了。具体步骤：

1. 在system_server进程中的调用ActivityStackSupervisor.realStartActivityLocked()后，Android系统会在App进程中调用ApplicationThread.scheduleLaunchActivity()方法。然后ApplicationThread.scheduleLaunchActivity()方法中创建了ActivityClientRecord，并将token赋值给了该ActivityClientRecord。
2. 然后又将ActivityClientRecord发送给主线程ActivityThread处理。
3. ActivityThread.performLauncherActivity()方法中创建了Activity，并调用了Activity.attach()。在Activity.attach()方法中传入了ActivityClientRecord.token。至此，ActivityRecord、ActivityClientRecord、Activity三者的一一对应关系就建立完毕了。

Token的利用

上文已经分析完ActivityRecord、ActivityClientRecord、Activity三者如何建立对应关系的了，那么Android系统又是如何利用这层对应关系的呢。熟悉Activity启动流程的同学都知道Activity的生命周期实际上是由AMS控制的，比如启动Activity时需要令上一个Activity执行onPause()方法。这时会经历以下方法调用链：

```
1 ActivityStack.startPausingLocked()
2 IApplicationThread.schudulePauseActivity()
3 ActivityThread.sendMessage()
4 ActivityThread.H.sendMessage();
5 ActivityThread.H.handleMessage()
6 ActivityThread.handlePauseActivity()
7 ActivityThread.performPauseActivity()
8 Activity.performPause()
9 Activity.onPause()
```

```
ActivityManagerNative.getDefault().activityPaused(token)
ActivityManagerService.activityPaused()
ActivityStack.activityPausedLocked()
ActivityStack.completePauseLocked()
```

1. 将当前获得焦点的ActivityRecord的appToken发给App进程。
2. 之前启动Activity时会调用performLaunchActivity(ActivityClientRecord r, Intent customIntent)，并在最后以token为Key，ActivityClientRecord为value保存ActivityClientRecord。然后到了performPauseActivity()中又会根据token取出对应ActivityClientRecord。再调用ActivityClientRecord中保存的activity的onPause()方法。
3. 执行完performPauseActivity()方法后，还要告知AMS，onPause()方法调用完毕。将token传回AMS。
4. 接着调用到ActivityStack.activityPausedLocked() 。
5. AMS收到token后，根据token中的弱引用找到了对应的ActivityRecord。最后判断找到的ActivityRecord是否与保存的mPausingActivity是同一个对象。若是，就执行completePauseLocked(true)，说明AMS已经收到Activity已调用onPause()的消息。

TaskRecord：任务栈记录者

Task的信息记录在TaskRecord对象中：

- ActivityStack stack; // 当前所属的stack
- ArrayList <activityrecord>mActivities; // 当前task的所有Activity列表</activityrecord>
- int taskId
- String affinity； // 是指root activity的affinity，即该Task中第一个Activity;
- int mCallingUid;
- String mCallingPackage； // 调用者的包名

开发过程中，为满足各种业务需求，开发者需要灵活运用Activity的四种启动模式，通常我们通过指定`launchMode`就能解决大部分的问题，而对活动任务记录栈的理解只是停留在抽象的概念当中。
首先，通过activity的`taskAffinity`属性，可以指定activity的活动任务记录栈，在不指定`taskAffinity`的条件下，**启动时入口Activity的默认TaskRecord为软件包名applicationId，随后被启动的Activity的TaskRecord默认与启动它的Activity一致。**

示例分析

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.student0.ada01">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
```

```
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:configChanges="orientation|screenSize"
            android:launchMode="standard"
            >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".SecondActivity"
            android:configChanges="screenLayout"
            android:label="@string/app_name"
            android:taskAffinity="com.example.student.task1"
            android:launchMode="singleTask"
            />
        <activity
            android:name=".ThirdActivity"
            android:configChanges="screenLayout"
            android:taskAffinity="com.example.student.task1"
            android:launchMode="singleTask"
            />
    </application>

</manifest>
```

查看Activity任务栈：adb shell dumpsys activity

- 指定了`SecondActivity`和`ThirdActivity`的活动任务堆栈为`com.example.student.task1`
- 软件的唯一标识applicationId，改为了"com.example.student0.ada101"
- 然后分别在每个Activity中添加`Button`实现Activity路由，路由的规则如下：MainActivity-->SecondActivity-->ThirdActivity->MainActivity

任务栈流程：系统首先启动MainActivity,由于没有设置taskAffinity属性，此时系统将建立第一个管理活动的Task，且Task的TaskRecord名字与applicationId一致,当通过MainActivity启动SecondActivity时，由于此时已经指定了SecondActivity的TaskAffinity属性为`com.example.student0.task1`，系统将寻找TaskRecord为`com.example.student0.task1`的Task，如果不存在则建立一个满足要求的Task,如果Activity指定的TaskRecord已存在，该Activity则交由此TaskRecord维护，这一点可以从SecondActivity启动ThirdActivity中看出。最后，由于MainActivity并未指定TaskAffinity属性，且LaunchMode也是默认的Standard，所以当使用ThirdActivity启动MainActivity时，在ThirdActivity所在的Task中的TaskRecord将再次创建并维护一个MainActivity。

TaskRecord的创建

```
frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

class ActivityStarter {

    private int setTaskFromReuseOrCreateNewTask(TaskRecord taskToAffiliate, int preferredLaunchStackId, ActivityStack topStack) {
        mTargetStack = computeStackFocus(mStartActivity, true, mLaunchBounds, mLaunchFlags, mOptions);

        if (mReuseTask == null) {
            //创建一个createTaskRecord，实际上是调用ActivityStack里面的createTaskRecord（）方法，ActivityStack下面会讲到
            final TaskRecord task = mTargetStack.createTaskRecord(
                    mSupervisor.getNextTaskIdForUserLocked(mStartActivity.userId),
                    mNewTaskInfo != null ? mNewTaskInfo : mStartActivity.info,
                    mNewTaskIntent != null ? mNewTaskIntent : mIntent, mVoiceSession,
                    mVoiceInteractor, !mLaunchTaskBehind /* toTop */, mStartActivity.mActivityType);

            //其他代码略
        }
    }
}
```

ActivityStack

- final int mStackId;
- int mDisplayId;
- ActivityRecord mPausingActivity = null;//正在暂停的
- Activity ActivityRecord mLastPausedActivity = null;//上一个已经暂停的
- Activity ActivityRecord mLastNoHistoryActivity = null;//最近一次没有历史记录的
- Activity ActivityRecord mResumedActivity = null;//已经Resume的
- Activity ActivityRecord mLastStartedActivity = null;//最近一次启动的
- Activity ActivityRecord mTranslucentActivityWaiting = null;//传递给convertToTranslucent方法的最上层的Activity
- mTaskHistory TaskRecord 所有没有被销毁的Task
- mLRUActivities ActivityRecord 正在运行的Activity，列表中的第一个条目是最近最少使用的元素
- mNoAnimActivities ActivityRecord 不考虑转换动画的Activity mValidateAppTokens TaskGroup 用于与窗口管理器验证应用令牌

所有前台stack的mResumedActivity的state == RESUMED, 则表示allResumedActivitiesComplete, 此时mLastFocusedStack = mFocusedStack;

再来说一说Activity类型和Activity状态的常量：
mActivityType：

- APPLICATION_ACTIVITY_TYPE：普通应用类型
- HOME_ACTIVITY_TYPE：桌面类型
- RECENTS_ACTIVITY_TYPE：最近任务类型

ActivityState(定义在ActivityStack中)：

- INITIALIZING
- RESUMED：已恢复
- PAUSING
- PAUSED：已暂停
- STOPPING
- STOPPED：已停止
- FINISHING
- DESTROYING
- DESTROYED：已销毁

```
 frameworks/base/services/core/java/com/android/server/am/ActivityStack.java

class ActivityStack<T extends StackWindowController> extends ConfigurationContainer implements StackWindowListener {

    private final ArrayList<TaskRecord> mTaskHistory = new ArrayList<>();//使用一个ArrayList来保存TaskRecord
```

```
6
7     final int mStackId;
8
9     protected final ActivityStackSupervisor mStackSupervisor;//持有一个ActivityStackSupervisor，所有的运行中的ActivityStacks都通过它来进行管理
10
11    //构造方法
12    ActivityStack(ActivityStackSupervisor.ActivityDisplay display, int stackId,
13                  ActivityStackSupervisor supervisor, RecentTasks recentTasks, boolean onTop) {
14
15    }
16
17    TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,
18                                IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
19                                boolean toTop, int type) {
20
21        //创建一个task
22        final TaskRecord task = TaskRecord.create(
23                mService, taskId, info, intent, voiceSession, voiceInteractor);
24
25        //将task添加到ActivityStack中去
26        addTask(task, toTop, "createTaskRecord");
27
28        //其他代码略
29        return task;
30    }
31
32    //添加Task
33    void addTask(final TaskRecord task, final boolean toTop, String reason) {
34
35        addTask(task, toTop ? MAX_VALUE : 0, true /* schedulePictureInPictureModeChange */, reason);
36
37        //其他代码略
```

```
    }

    //添加Task到指定位置
    void addTask(final TaskRecord task, int position, boolean schedu
lePictureInPictureModeChange,
                 String reason) {
        mTaskHistory.remove(task);//若存在，先移除

        //...

        mTaskHistory.add(position, task);//添加task到mTaskHistory
        task.setStack(this);//为TaskRecord设置ActivityStack

        //...
    }

    //其他代码略
}
```

`ActivityStack`,内部维护了一个`ArrayList<TaskRecord>`，用来管理`TaskRecord`。

- 可以看到`ActivityStack`使用了一个`ArrayList`来保存`TaskRecord`。
- 另外，`ActivityStack`中还持有`ActivityStackSupervisor`对象，这个是用来管理`ActivityStacks`的。

`ActivityStack`是由`ActivityStackSupervisor`来创建的，实际`ActivityStackSupervisor`就是用来管理`ActivityStack`的，继续看下面的`ActivityStackSupervisor`分析。

ActivityStackSupervisor

- ActivityStack mHomeStack //桌面的stack
- ActivityStack mFocusedStack //当前聚焦stack
- ActivityStack mLastFocusedStack //正在切换
- SparseArray <activitydisplay>mActivityDisplays //displayId为key</activitydisplay>
- SparseArray <activitycontainer>mActivityContainers // mStackId为key</activitycontainer>

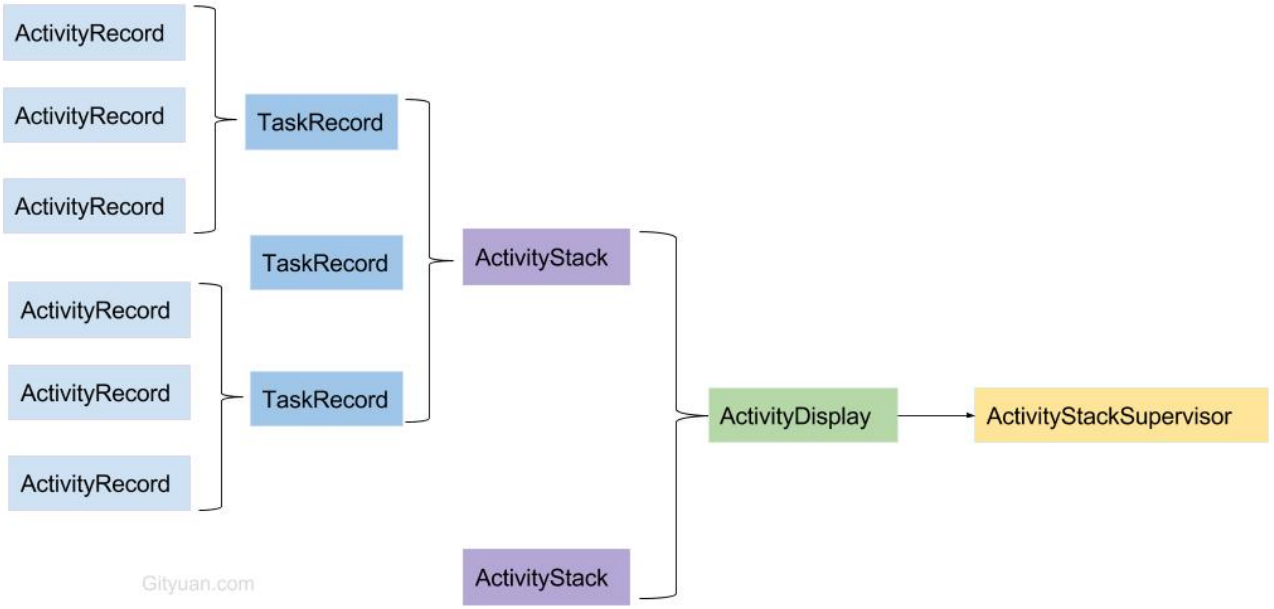
home的栈ID等于0,即HOME_STACK_ID = 0;

`ActivityStackSupervisor`，顾名思义，就是用来管理`ActivityStack`的。

- `ActivityStackSupervisor`内部有两个不同的`ActivityStack`对象：`mHomeStack`、`mFocusedStack`，用来管理不同的任务。
- `ActivityStackSupervisor`内部包含了创建`ActivityStack`对象的方法。

AMS初始化时会创建一个`ActivityStackSupervisor`对象。

Activity栈结构体的组成关系



- 一般地，对于没有分屏功能以及虚拟屏的情况下，ActivityStackSupervisor与ActivityDisplay都是系统唯一；
- ActivityDisplay主要有Home Stack和App Stack这两个栈；
- 每个ActivityStack中可以有若干个TaskRecord对象；
- 每个TaskRecord包含如果个ActivityRecord对象；
- 每个ActivityRecord记录一个Activity信息。

场景任务栈分析

下面通过启动Activity的代码来分析一下：

桌面

首先，我们看下处于桌面时的状态，运行命令：

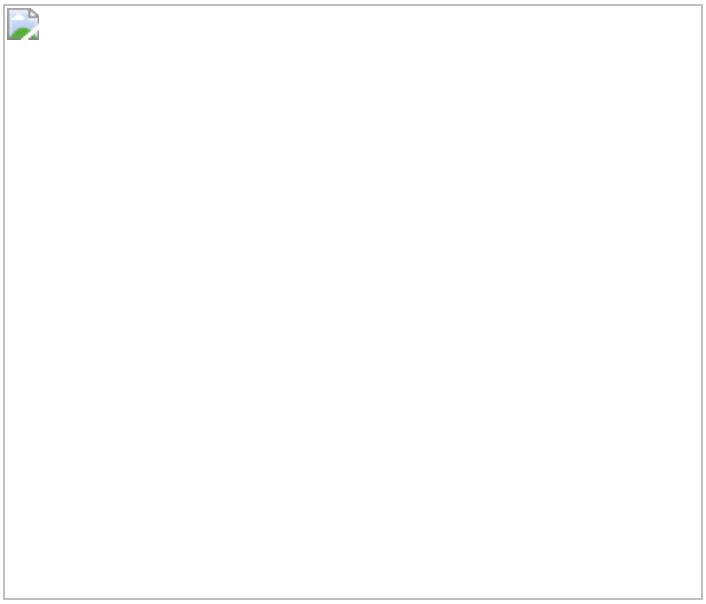
```
1 adb shell dumpsys activity
```

结果如下：

```
1 ACTIVITY MANAGER ACTIVITIES (dumpsys activity activities)
2 Display #0 (activities from top to bottom):
3   Stack #0:
4
5   //中间省略其他...
```

```
6
7     Task id #102
8
9   //中间省略其他...
10
11      TaskRecord{446ae9e #102 I=com.google.android.apps.nexuslaunche
    r/.NexusLauncherActivity U=0 StackId=0 sz=1}
12      Intent { act=android.intent.action.MAIN cat=[android.intent.ca
    tegory.HOME] flg=0x10000100 cmp=com.google.android.apps.nexuslaunche
    r/.NexusLauncherActivity }
13        Hist #0: ActivityRecord{54fa22 u0 com.google.android.apps.ne
    xuslauncher/.NexusLauncherActivity t102}
14          Intent { act=android.intent.action.MAIN cat=[android.inten
    t.category.HOME] flg=0x10000100 cmp=com.google.android.apps.nexuslau
    ncher/.NexusLauncherActivity }
15          ProcessRecord{19c7c43 2203:com.google.android.apps.nexusla
    uncher/u0a22}
16    Running activities (most recent first):
17      TaskRecord{446ae9e #102 I=com.google.android.apps.nexuslaunche
    r/.NexusLauncherActivity U=0 StackId=0 sz=1}
18        Run #0: ActivityRecord{54fa22 u0 com.google.android.apps.nex
    uslauncher/.NexusLauncherActivity t102}
19    mResumedActivity: ActivityRecord{54fa22 u0 com.google.android.ap
    ps.nexuslauncher/.NexusLauncherActivity t102}
20
21  //省略其他
```

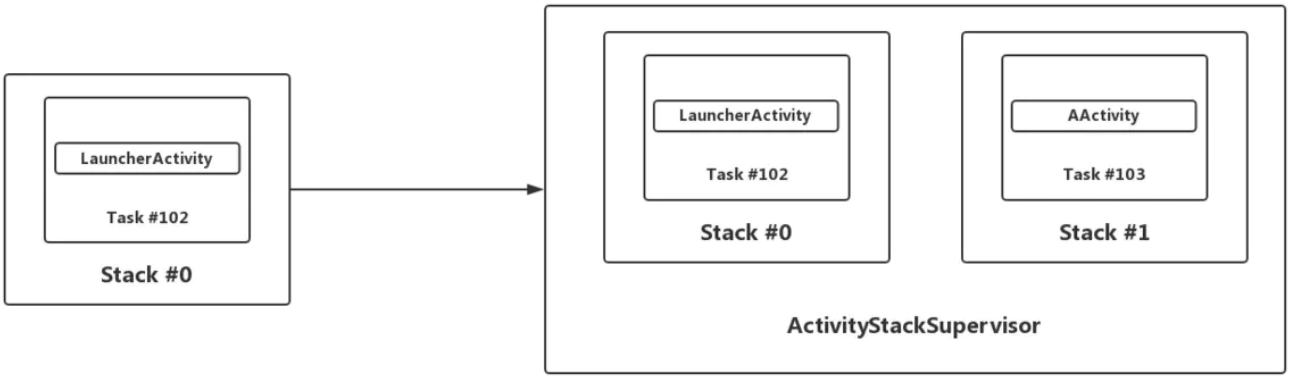
实际上就是如下图所示的结构，这里的`Stack #0`就是`ActivityStackSupervisor`中的`mHomeStack`，`mHomeStack`管理的是Launcher相关的任务。



从桌面启动一个Activity

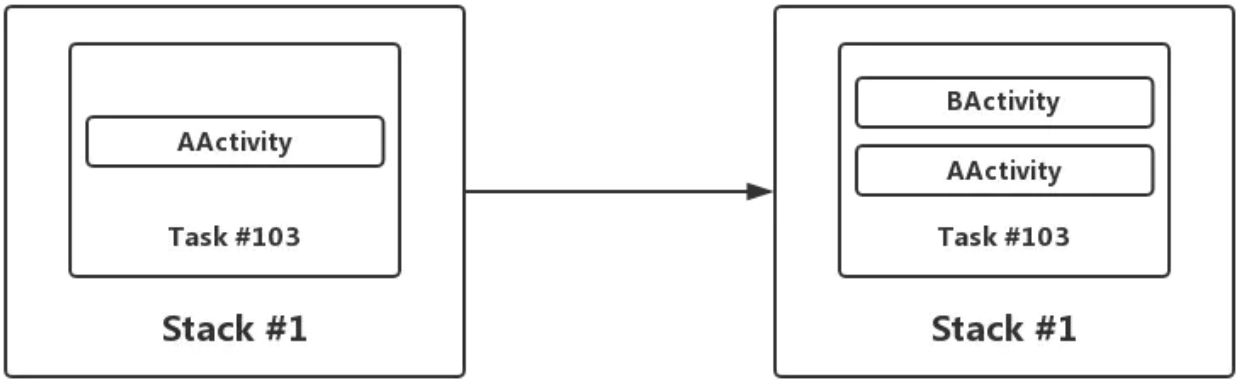
> 从桌面启动一个APP，然后运行上面的命令，为了节省篇幅，这里和后面就不贴结果了，直接放图了。

从桌面点击图标启动一个`AActivity`，可以看到，会多了一个`Stack #1`，这个`Stack #1`就是`ActivityStackSupervisor`中的`mFocusedStack`，`mFocusedStack`负责管理的是非Launcher相关的任务。同时也会创建一个新的`ActivityRecord`和`TaskRecord`，`ActivityRecord`放到`TaskRecord`中，`TaskRecord`则放进`mFocusedStack`中。



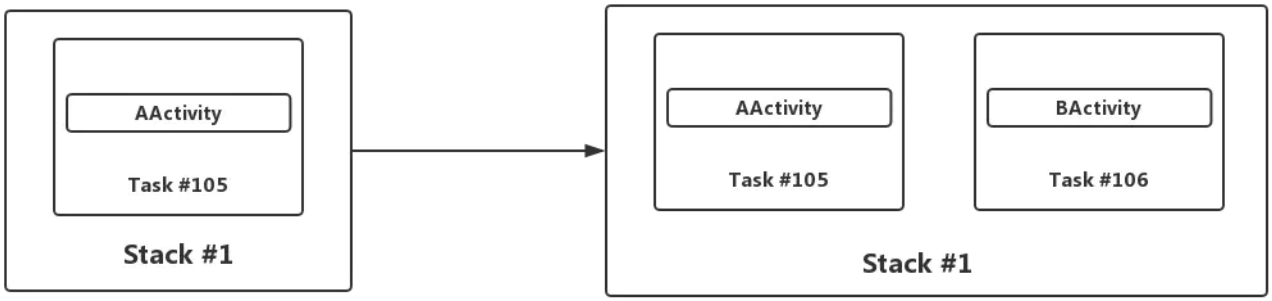
默认模式从A启动B

然后，我们从`AActivity`中启动一个`BActivity`,可以看到会创建一个新的`ActivityRecord`然后放到已有的`TaskRecord`栈顶。



从A启动B创建新栈

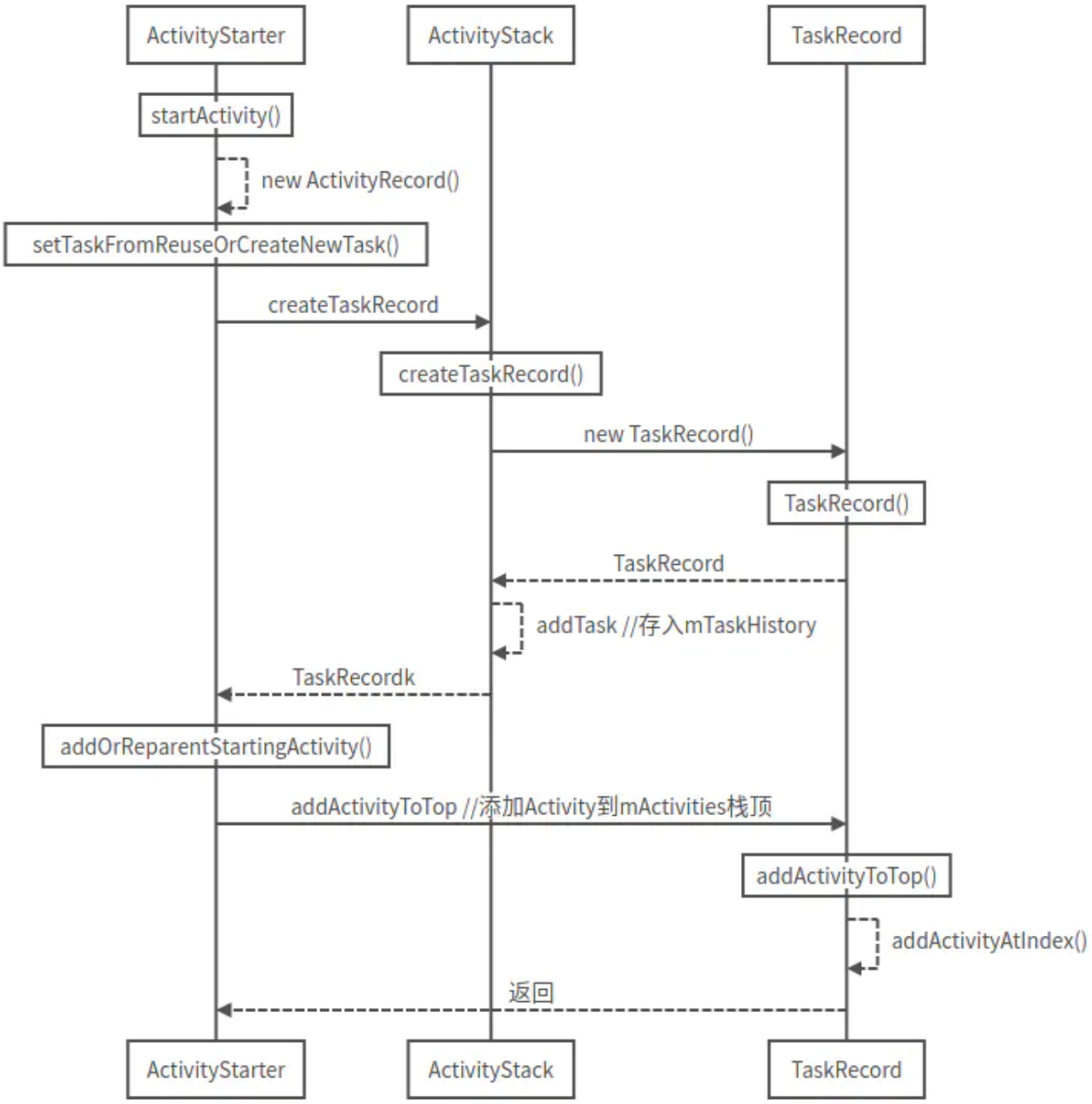
如果我们想启动的`BActivity`在一个新的栈中呢，我们可以用`singleInstance`的方式来启动`BActivity`。`singleInstance`后面也会讲到。这种方式会创建一个新的`ActivityRecord`和`TaskRecord`，把`ActivityRecord`放到新的`TaskRecord`中去。



启动流程任务栈分析

启动流程

这里对启动Activity过程中涉及到的`ActivityStack`、`TaskRecord`、`ActivityRecord`、`ActivityStackSupervisor`进行简单的分析，实际上一张时序图就可以看明白了。相关的代码可以看上面的内容。



简单总结：

1.`startActivity`时首先会创建一个`ActivityRecord`。

2.如果有需要，会创建一个`TaskRecord`，并把这个`TaskRecord`加入到`ActivityStack`中。

3.将`ActivityRecord`添加到`TaskRecord`的栈顶。

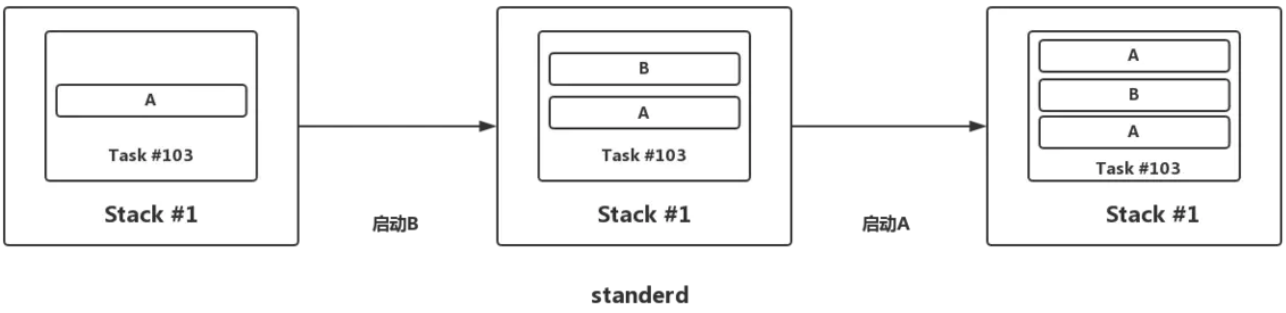
启动模式任务栈分析

相信看完上面的介绍，现在再来看启动模式那是so easy了。

standerd

> 默认模式，每次启动Activity都会创建一个新的Activity实例。

比如：现在有个A Activity,我们在A上面启动B，再然后在B上面启动A，其过程如图所示：

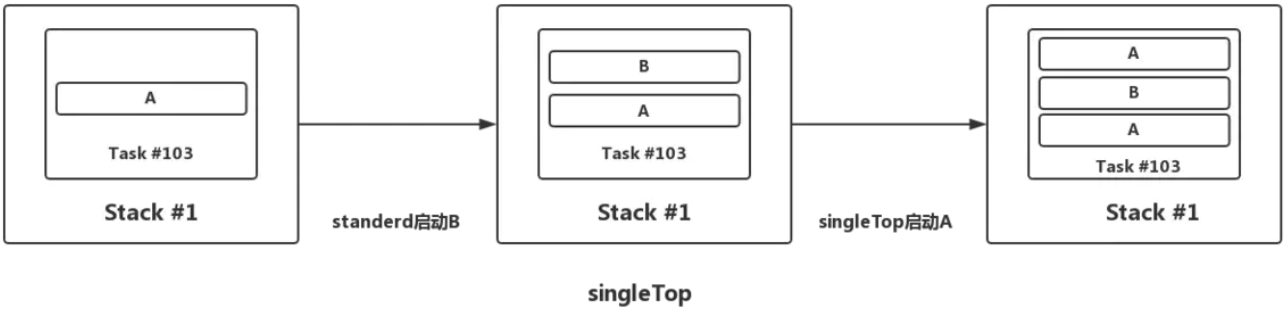


standerd

singleTop

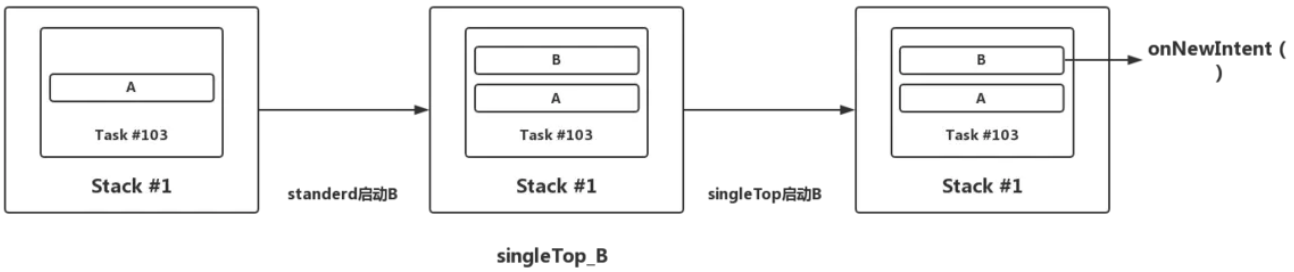
> 如果要启动的Activity已经在栈顶，则不会重新创建Activity，只会调用该该Activity的`onNewIntent()`方法。
> 如果要启动的Activity不在栈顶，则会重新创建该Activity的实例。

比如：现在有个A Activity,我们在A以`standerd`模式上面启动B，然后在B上面以`singleTop`模式启动A，其过程如图所示，这里会新创建一个A实例：



singleTop

如果在B上面以`singleTop`模式启动B的话，则不会重新创建B，只会调用`onNewIntent()`方法，其过程如图所示：

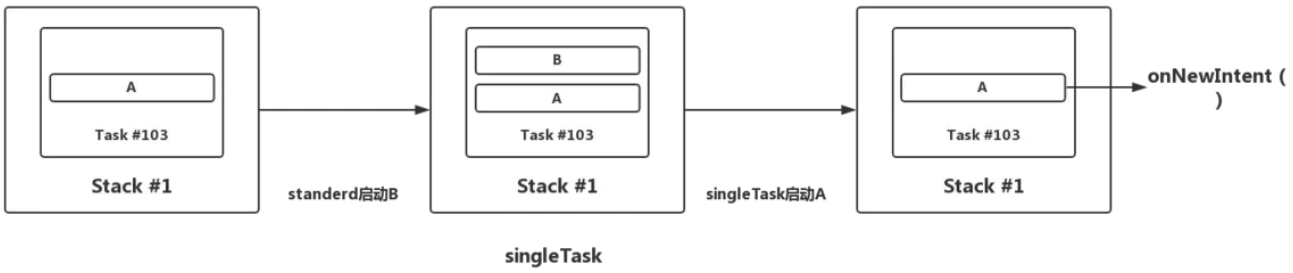


singleTop_B

singleTask

> 如果要启动的Activity已经存在于它想要归属的栈中，那么不会创建该Activity实例，将栈中位于该Activity上的所有的Activity出栈，同时该Activity的`onNewIntent()`方法会被调用。
> 如果要启动的Activity不存在于它想要归属的栈中，并且该栈存在，则会创建该Activity的实例。
> 如果要启动的Activity想要归属的栈不存在，则首先要创建一个新栈，然后创建该Activity实例并压入到新栈中。

比如：现在有个A Activity，我们在A以`standerd`模式上面启动B，然后在B上面以`singleTask`模式启动A，其过程如图所示：

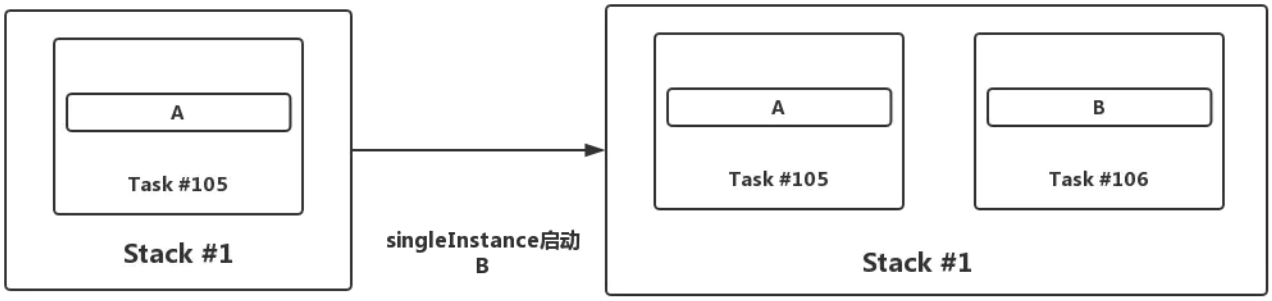


singleTask

singleInstance

> 基本和`singleTask`一样，不同的是启动Activity时，首先要创建在一个新栈，然后创建该Activity实例并压入新栈中，新栈中只会存在这一个Activity实例。

比如：现在有个A Activity,我们在A以`singleInstance`模式上面启动B，其过程如图所示：



singleInstance