# 四种启动模式的特点

## 1.1 预先知识

Activity是通过任务栈来管理的。

任务栈存放Activity引用。

## 1.2 启动模式

Standare：默认启动模式，每次都会在任务栈顶创建一个实例。

（得清楚这里的栈是什么概念？）

SingleTop：检查该 Activity 的实例是否位于栈顶，位于栈顶时复用，非栈顶时创建新实例。

SingleTask：启动此 Activity 时会检查栈中是否存在它的实例，如果存在直接复用，并把当前Activity之上所有实例全部出栈。

SingleInstance：独享栈。该模式的 Activity 会启动一个新的任务栈来管理 Activity 实例，并且该实例在整个系统中只有一个。无论从那个任务栈中启动该 Activity，都会是该 Activity 所在的任务栈转移到前台，从而使Activity显示。主要作用是为了在不同程序中共享一个 Activity 实例。

（这里说的让不同程序中共享一个Activity 实例是什么意思？

所谓的到单例模式，是在什么范围内的单例）。

## 1.3 Activity不同启动模式的应用场景

Standare：默认，无需设置

SingleTop：APP接收到多条推送消息，点开不同消息，均由同一页面实例展示。

登录页面、WXPayEntryActivity、WXEntryActivity（使用微信第三方登录时用到） 、推送通知栏。

SingleTask：APP的主页，无论哪种业务场景下再次回到此页，都不应保留之上Activity。主页就是做到不能后退其他的activity了，还要退就是退出程序了。所以，一般MainActivity使用SingleTask模式。

还有比如聊天页面也是singleTask。

SingleInstance：如APP经常调用的拨打电话、系统通讯录、地图类APP 等页面，不同APP调用此类Activity 时，首次创建实例，之后其他APP只能复用此实例。

比如系统的Launcher界面。

这种启动模式，在我们使用的还是不多的。

singleInstance 单实例，一个Activity被其他方多次调用(如果不用singleInstance，那会很混乱，好几个人打电话过来怎么办，所以需要singleInstance。

并且这个Activity对性能要求高。

使用这种模式也是合理的，就它一个活动在这个栈，跟别的活动栈没有瓜葛。

## 1.4 singleInstance的坑

此时有三个activity，ActivityA，ActivityB，ActivityC，除了ActivityB的启动模式为singleInstance，其他的启动模式都为默认的。startActivity了一个ActivityA，在ActivityA里startActivity了一个ActivityB，在ActivityB里startActivity了一个ActivityC。此时在当前的任务栈中的顺序是，ActivityA->ActivityB->ActivityC。照理来说在当前ActivityC页面按返回键，finish当前界面后应当回到ActivityB界面。但是事与愿违，奇迹出现了，页面直接回到了ActivityA。这是为什么呢？其实想想就能明白了，上面已经说过，singleInstance模式是存在于另一个任务栈中的。也就是说ActivityA和ActivityC是处于同一个任务栈中的，ActivityB则是存在另个栈中。所以当关闭了ActivityC的时候，它自然就会去找当前任务栈存在的activity。当前的activity都关闭了之后，才会去找另一个任务栈中的activity。也就是说当在ActivityC中finish之后，会回到ActivityA的界面，在ActivityA里finish之后会回到ActivityB界面。如果还想回到ActivityB的页面怎么办呢？

此时有两个activity，ActivityA，ActivityB，ActivityA的启动模式为默认的，ActivityB的启动模式为singleInstance。当在ActivityA里startActivity了ActivityB，当前页面为ActivityB。按下home键。应用退到后台。此时再点击图标进入APP，按照天理来说，此时的界面应该是ActivityB，可是奇迹又出现了，当前显示的界面是ActivityA。这是因为当重新启动的时候，系统会先去找主栈（我是这么叫的）里的activity，也就是APP中LAUNCHER的activity所处在的栈。查看是否有存在的activity。没有的话则会重新启动LAUNCHER。

## 1.5 启动模式的生命周期

其实不同启动模式的生命周期回调是大体按功能描述的，但是细节上是有不同的。

## （1）标准模式log

//启动模式：标准模式standard，ModeStandardActivity上启动ModeStandardActivity，还是会创建对象

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: 实例对象: com.wlp.myapp.activity.ModeStandardActivity@acef048

I/ModeStandardActivity: 任务栈taskId: 72

I/ModeStandardActivity:onStart

I/ModeStandardActivity: onResume:

I/ModeStandardActivity: onClick: //

I/ModeStandardActivity: onPause:

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: 实例对象: com.wlp.myapp.activity.ModeStandardActivity@c3bdacb

I/ModeStandardActivity: 任务栈taskId: 72

I/ModeStandardActivity:onStart

I/ModeStandardActivity: onResume:

I/ModeStandardActivity: onClick:

I/ModeStandardActivity: onPause:

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: 实例对象: com.wlp.myapp.activity.ModeStandardActivity@fd43b25

I/ModeStandardActivity: 任务栈taskId: 72

I/ModeStandardActivity: onResume:

## （2）栈顶复用模式

I/StandardActivity: onCreate:

I/StandardActivity: launchMode 实例对象: com.wlp.myapp.activity.StandardActivity@2374928

I/StandardActivity: launchMode 任务栈taskId: 74

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/StandardActivity: onPause: //点击启动B： ModeStandardActivity

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: launchMode 实例对象: com.wlp.myapp.activity.ModeStandardActivity@acef048

I/ModeStandardActivity: launchMode 任务栈taskId: 74

I/ModeStandardActivity: onStart:

I/ModeStandardActivity: onResume:

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop:

I/ModeStandardActivity: onClick: //在B中点击，再启动B

I/ModeStandardActivity: onPause:

I/ModeStandardActivity: onResume:

I/ModeStandardActivity: onPause: //点击后退键

I/StandardActivity: onRestart:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/ModeStandardActivity: onStop:

I/ModeStandardActivity: onDestroy:

## 栈内复用

1. singleTask模式被启动
2. singleTask模式被覆盖
3. singleTask模式又重新会到

I/StandardActivity: onCreate:

I/StandardActivity: launchMode 实例对象: com.wlp.myapp.activity.StandardActivity@2374928

I/StandardActivity: launchMode 任务栈taskId: 76

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/StandardActivity: onPause: //启动ModeStandardActivity

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: launchMode 实例对象: com.wlp.myapp.activity.ModeStandardActivity@a411919

I/ModeStandardActivity: launchMode 任务栈taskId: 76

I/ModeStandardActivity: onStart:

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@a411919

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop:

I/ModeStandardActivity: onClick:

I/ModeStandardActivity: onPause: //启动BActivity ？？？

I/BActivity: onCreate:

I/BActivity: launchMode 实例对象: com.wlp.myapp.activity.BActivity@157433e

I/BActivity: launchMode 任务栈taskId: 76

I/BActivity: onStart:

I/BActivity: onResume: //点击启动 ModeStandardActivity

I/BActivity: onPause:

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@a411919

I/ModeStandardActivity: onPause:

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@a411919

//看到没有? 这个切换的什么周期是不一样的，SingleTask实例被覆盖只走了onPause()

I/BActivity: onStop:

I/BActivity: onDestroy:

I/ModeStandardActivity: onPause: //后退

I/StandardActivity: onRestart:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/ModeStandardActivity: onStop:

I/ModeStandardActivity: onDestroy:

## 单例模式

1. singleInstance模式被启动
2. singleInstance模式被覆盖
3. singleInstance模式又重新会到

I/StandardActivity: onCreate:

I/StandardActivity: launchMode 实例对象: com.wlp.myapp.activity.StandardActivity@7c33795

I/StandardActivity: launchMode 任务栈taskId: 79

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/StandardActivity: onPause: //打开ModeStandardActivity singleTask

I/ModeStandardActivity: onCreate:

I/ModeStandardActivity: launchMode 实例对象: com.wlp.myapp.activity.ModeStandardActivity@88a8eed

I/ModeStandardActivity: launchMode 任务栈taskId: 79

I/ModeStandardActivity: onStart:

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@88a8eed taskId 79

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop:

I/ModeStandardActivity: onClick: // 打开BActivity singleInstance

I/ModeStandardActivity: onPause:

I/BActivity: onCreate:

I/BActivity: launchMode 实例对象: com.wlp.myapp.activity.BActivity@2be07cd

I/BActivity: launchMode 任务栈taskId: 80 //会新开一个Task

I/BActivity: onStart:

I/BActivity: onResume: // 打开ModeStandardActivity，还是回到原先那个Task === ↓

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@88a8eed taskId 79

I/ModeStandardActivity: onPause:

I/BActivity: onPause:

I/ModeStandardActivity: onResume: com.wlp.myapp.activity.ModeStandardActivity@88a8eed taskId 79

I/BActivity: onSaveInstanceState:

I/BActivity: onStop: // 到这里，才回到ModeStandardActivity。这里调用的有点复杂 === ↑

I/ModeStandardActivity: onClick: // 这里又打开BActivity singleInstance，会回到原先的那个对象

I/ModeStandardActivity: onPause:

I/BActivity: onRestart: //

I/BActivity: onStart:

I/BActivity: onResume:

# 1.4 启动另一个应用的activity

给被启动的activity设置过滤器<activity>标签下

<intent-filter>

<action android:name="devdiv.intent.action.activity4" />

<category android:name="android.intent.category.DEFAULT" />

</intent-filter>

通过action创建Intent，即可被启动意图。

//还有一种是使ComponentName（组件名称）来创建Intent，也可以。但是没有去验证。

Intent intent = new Intent("devdiv.intent.action.activity4");

startActivity(intent);

# 生命周期

## Activity的生命周期

## 生命周期回调方法

生命周期方法 作用 说明

onCreate 表示 Activity 正在被创建 ---- activity 被创建时调用，一般在这个方法中进行活动的初始化工作，如设置布局工作、加载数据、绑定控件、设置监听等。比如使用setContentView加载布局，对一些控件和变量进行初始化等。动画不应该在这里初始化，因为看不到。

onRestart 表示 Activity 正在重新启动 ---- 这个回调代表了 Activity 由完全不可见重新变为可见的过程，当 Activity 经历了 onStop() 回调变为完全不可见后，如果用户返回原 Activity，便会触发该回调，并且紧接着会触发 onStart() 来使活动重新可见。

onStart 表示 Activity 正在被启动 --- 经历该回调后，Activity 由不可见变为可见，但此时处于后台可见，还不能和用户进行交互。 可以做一些动画的初始化。

onResume 表示 Activity 已经可见 ---- 已经可见的 Activity 从后台来到前台，可以和用户进行交互。

onPause 表示 Activity 正在停止 ---- 当用户启动了新的 Activity ，原来的 Activity 不再处于前台，也无法与用户进行交互，并且紧接着就会调用 onStop() 方法，但如果用户这时立刻按返回键回到原 Activity ，就会调用 onResume() 方法让活动重新回到前台。而且在官方文档中给出了说明，不允许在 onPause() 方法中执行耗时操作，因为这会影响到新 Activity 的启动。 onResume 和 onPause主要做一些轻量级的操作。

onStop 表示 Activity 即将停止 ---- 这个回调代表了 Activity 由可见变为完全不可见，在这里可以进行一些稍微重量级的操作。需要注意的是，处于 onPause() 和 onStop() 回调后的 Activity 优先级很低，当有优先级更高的应用需要内存时，该应用就会被杀死，那么当再次返回原 Activity 的时候，会重新调用 Activity 的onCreate()方法。 这个阶段的主要工作也是做一些资源的回收工作。暂时用户不需要的都在这里释放掉。数据的保存等等。

onDestroy 表示 Activity 即将被销毁 ----- 来到了这个回调，说明 Activity 即将被销毁，应该将资源的回收和释放工作在该方法中执行。onStop中没有释放的在这里释放。

## 最简单的

启动——>按退出键,如下：一个完整的流程

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onStart:

\* I/StandardActivity: onResume:

\* I/StandardActivity: onPause:

\* I/StandardActivity: onStop:

\* I/StandardActivity: onDestroy:

## （2）竖屏到横屏

\* ========竖屏到横屏=================

//(要看Android sdk版本的，不同版本略有不同，)

1. 没有设置configChanges属性，Android 6.0 7.0 8.0 系统都是一样的。
2. 设置了configChanges属性为orientation之后，Android6.0 同没有设置configChanges情况相同，完整的走完了两个生命周期，调用了onSaveInstanceState和onRestoreInstanceState方法；Android 7.0则会先回调onConfigurationChanged方法，剩下的流程跟Android 6.0 保持一致；Android 8.0 系统更是简单，只是回调了onConfigurationChanged方法，并没有走Activity的生命周期方法。

\* 操作：启动 ——> 竖屏到横屏：保存状态，系统将当前对象销毁，重新创建一个Activity

\* onSaveInstanceState 在 onPause之后

\* onRestoreInstanceState 在 onResume 之前调用

\*

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onStart:

\* I/StandardActivity: onResume: //这里开始切换

\* I/StandardActivity: onPause:

\* I/StandardActivity: onSaveInstanceState: //保存当前实例的状态：即一些临时的数据，在pause()之后调用

\* I/StandardActivity: onStop:

\* I/StandardActivity: onDestroy: //切换之后，先销毁，再重新创建

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onCreate: wang li ping //Bundle 可以获取值

\* I/StandardActivity: onStart:

\* I/StandardActivity: onRestoreInstanceState: //Bundle 这里也可以获取值

\* I/StandardActivity: onRestoreInstanceState: wang li ping

\* I/StandardActivity: onResume: //到这里才可以进行交互了

\* I/StandardActivity: onPause:

\* I/StandardActivity: onStop:

\* I/StandardActivity: onDestroy: //

\*

## （3）横屏到竖屏

\* ========横屏到竖屏=================：与竖屏切横屏相同：

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onStart:

\* I/StandardActivity: onResume: //

\* I/StandardActivity: onPause:

\* I/StandardActivity: onSaveInstanceState:

\* I/StandardActivity: onStop:

\* I/StandardActivity: onDestroy:

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onCreate: wang li ping

\* I/StandardActivity: onStart:

\* I/StandardActivity: onRestoreInstanceState:

\* I/StandardActivity: onRestoreInstanceState: wang li ping

\* I/StandardActivity: onResume:

\*

## （4）设置configChanges的切换

\* 当设置 android:configChanges="orientation"

\* 当前的Android8.0: 不走那些流程，会回调方法onConfigurationChanged()

\*

\* android:configChanges="orientation|screenSize"

\*对于 android8也是这样的,同设置 android:configChanges="orientation"

## （4-1）切换横竖屏时获取宽高

protected void onResume() {

super.onResume();

Log.i(TAG, "onResume: ");

Log.i(TAG, "onResume: 屏幕尺寸：高 = " + ScreenUtil.getScreenHeight(this)

+ " 宽 = " + ScreenUtil.getScreenWidth(this));

//TODO 当进行横竖屏切换时，获得的屏幕尺寸会自动切换的，得到的宽高就是正确的，我们需要的。

//横屏 onResume: 屏幕尺寸：高 = 1080 宽 = 1776

//竖屏 onResume: 屏幕尺寸：高 = 1776 宽 = 1080

}

## 按Home键

Android 界面下面三个按钮三角形（返回键BACK）／圆形（HOME键）／方形（overview菜单键）

========Home键=================

\* //按Home键时会调用 onSaveInstanceState:

\* I/StandardActivity: onCreate:

\* I/StandardActivity: onStart:

\* I/StandardActivity: onResume:

\* I/StandardActivity: onPause:

\* I/StandardActivity: onSaveInstanceState: //

\* I/StandardActivity: onStop:

\* //又回到该页面

\* I/StandardActivity: onRestart:

\* I/StandardActivity: onStart:

\* I/StandardActivity: onResume: //看到没？这里之前没有调用onRestoreInstanceState

\* //按home键 O 键 或者 方形键，都是这样的

\* //如果在后台直接被杀死，比如手动杀死APP，destroy()方法不会被调用

解锁屏的情况：锁屏是从运行状态running执行onPause()到暂停状态；解锁就是执行OnResume回到运行状态。

## （6）显示Toast、显示Dialog

//显示一个Toast不会影响活动的生命周期

//Toast.makeText(this, "我看看啊",Toast.LENGTH\_LONG).show();

//一个弹窗Dialog也是没有影响的

\* onPause() Called when the system is about to start resuming another activity.

\* 只有再启动另外一个Activity的时候才会进入onPause状态，而不是想象中的被覆盖或者不可见

\* 同时通过AlertDialog源码或者Toast源码我们都可以发现它们实现的原理都是windowmanager.addView();来添加的，

\* 它们都是一个view ,因此不会对activity的生命周期有任何影响。

## （7）启动A再启动B

==========启动A，再启动B，再退出B,返回A

I/StandardActivity: onCreate: //启动A

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/StandardActivity: onPause: //点击启动B

I/BActivity: onCreate:

I/BActivity: onStart:

I/BActivity: onResume:

I/StandardActivity: onSaveInstanceState: //

I/StandardActivity: onStop: //

I/BActivity: onPause: // 点击后退，退出B

I/StandardActivity: onRestart: //

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/BActivity: onStop:

I/BActivity: onDestroy:

## （8）启动A再启动B，B是透明的

==========另一种情况，假如B是透明主题的：启动A，再启动B，再退出B,返回A

I/StandardActivity: onCreate:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/StandardActivity: onPause: //点击启动B ，

//这里是透明主题，前一个活动的回调就简单了，只调用了onPause，onSaveInstanceState都没执行。

I/BActivity: onCreate:

I/BActivity: onStart:

I/BActivity: onResume: //点击后退，返回A

I/BActivity: onPause:

I/StandardActivity: onResume:

I/BActivity: onStop:

I/BActivity: onDestroy:

## （9）onSaveInstanceState和onRestoreInstanceState

\* android中onSaveInstanceState和onRestoreInstanceState什么时候被调用?

\* 当某个activity变得“容易”被系统销毁时，该activity的onSaveInstanceState就会被执行，除非该activity是被用户主动销毁的，例如当用户按BACK键的时候。

\*

\* 注意上面的双引号，何为“容易”？言下之意就是该activity还没有被销毁，而仅仅是一种可能性。这种可能性有哪些？

\* 通过重写一个activity的所有生命周期的onXXX方法，包括onSaveInstanceState和onRestoreInstanceState方法，我们

\* 可以清楚地知道当某个activity（假定为activity A）显示在当前task的最上层时，其onSaveInstanceState方法会在什么时候被执行，有这么几种情况：

//onSaveInstanceState 在stop之前；

//onRestoreInstanceState在start之后。

\* 1、当用户按下HOME键时。

\* 这是显而易见的，系统不知道你按下HOME后要运行多少其他的程序，自然也不知道activity A是否会被销毁，故系统会

\* 调用onSaveInstanceState，让用户有机会保存某些非永久性的数据。以下几种情况的分析都遵循该原则

\* 2、长按HOME键，选择运行其他的程序时。

\* 3、按下电源按键（关闭屏幕显示）时。

\* 4、从activity A中启动一个新的activity时。

\* 5、屏幕方向切换时，例如从竖屏切换到横屏时。

\* 在屏幕切换之前，系统会销毁activity A，在屏幕切换之后系统又会自动地创建activity A，所以onSaveInstanceState一定会被执行。

\*

\* 总而言之，onSaveInstanceState的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的activity，则onSaveInstanceState会被系统调用，这是系统的责任，因为它必须要提供一个机会让你保存你的数据（当然你不保存那就随便你了）。

\* 至于onRestoreInstanceState方法，需要注意的是，onSaveInstanceState方法和onRestoreInstanceState方法“不一定”是成对的被调用的，onRestoreInstanceState被调用的前提是，activity A“确实”被系统销毁了，而如果仅仅是停留在有这种可能性的情况下，则该方法不会被调用，例如，当正在显示activity A的时候，用户按下HOME键回到主界面，然后

\* 用户紧接着又返回到activity A，这种情况下activity A一般不会因为内存的原因被系统销毁，故activityA的onRestoreInstanceState方法不会被执行。

\*

\* 另外，onRestoreInstanceState的bundle参数也会传递到onCreate方法中，你也可以选择在onCreate方法中做数据还原

## Activity与Fragment的生命周期

## Fragment的生命周期:

onAttach()： 完成Fragment和Activity的绑定关联，参数中的Activity即为要绑定的Activity，可以进行赋值等操作；

onCreate() : 完成Fragment的初始化；

onCreateView() : 加载Fragment布局，绑定布局文件，创建视图；

onActivityCreated() : 表名与Fragment绑定的Activity已经执行完成了onCreate，可以与Activity进行交互操作。

onStart() : Fragment变为可见状态

onResume() : Fragment变为可交互状态

onPause()： Fragment变为不可交互状态(不代表不可见)

onStop(): Fragment变为不可见状态

onDestroyView() : 销毁Fragment的有关视图，但并未和Activity解绑，可以通过onCreateView()重新创建视图。

onDestroy() : 销毁Fragment时调用。

onDetach() : 解除和Activity的绑定。Fragmen销毁最后一步

onSaveInstanceState()：保存当前Fragment的状态。记录一些数据，比如EditText键入的文本，即使Fragment被回收又重新创建，一样能恢复EditText之前键入的文本。

## （1）布局中标签<fragment>使用

//在activity中布局添加fragment的生命周期

I/StandardActivity: onPause:

I/FragmentActivity: onCreate: //这里开始activity的创建

I/CFragment: onAttach:

I/CFragment: onCreate:

I/CFragment: onCreateView:

I/CFragment: onActivityCreated:

I/CFragment: onViewStateRestored:

I/CFragment: onStart:

I/FragmentActivity: onStart:

I/FragmentActivity: onResume:

I/CFragment: onResume:

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop:

//这里点击后退页面

I/CFragment: onPause: //

I/FragmentActivity: onPause:

I/StandardActivity: onRestart:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/CFragment: onStop:

I/FragmentActivity: onStop:

I/CFragment: onDestroyView:

I/CFragment: onDestroy:

I/CFragment: onDetach:

I/FragmentActivity: onDestroy: //到这里就销毁完毕

## （2）Activity动态加载Fragment

FragmentManager manager = getSupportFragmentManager();

FragmentTransaction transaction = manager.beginTransaction();

fragment1 = new CFragment();

transaction.add(R.id.layout, fragment1);

transaction.commit();

//在activity中通过代码动态添加fragment的生命周期

I/StandardActivity: onPause:

I/FragmentActivity: onCreate:

I/FragmentActivity: onStart:

I/FragmentActivity: onResume:

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop: //点击按钮，添加Fragment

I/CFragment: onAttach:

I/CFragment: onCreate:

I/CFragment: onCreateView:

I/CFragment: onActivityCreated:

I/CFragment: onViewStateRestored:

I/CFragment: onStart:

I/CFragment: onResume:

I/CFragment: onPause:

I/FragmentActivity: onPause:

I/StandardActivity: onRestart:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/CFragment: onStop:

I/FragmentActivity: onStop:

I/CFragment: onDestroyView:

I/CFragment: onDestroy:

I/CFragment: onDetach:

I/FragmentActivity: onDestroy:

## 两个Fragment切换

FragmentManager manager = getSupportFragmentManager();

FragmentTransaction transaction = manager.beginTransaction();

if (v.getId() == R.id.button1) {

if (fragment1 == null) {

fragment1 = new CFragment();

transaction.add(R.id.layout, fragment1);

}

if(fragment2 != null) transaction.hide(fragment2);

transaction.show(fragment1);

} else if (v.getId() == R.id.button2){

if (fragment2 == null) {

fragment2 = new C2Fragment();

transaction.add(R.id.layout, fragment2);

}

if(fragment1 != null) transaction.hide(fragment1);

transaction.show(fragment2);

}

transaction.commit();

//在activity中通过代码动态添加fragment的生命周期 通过两个之间切换

I/FragmentActivity: onCreate:

I/FragmentActivity: onStart:

I/FragmentActivity: onResume:

I/StandardActivity: onSaveInstanceState:

I/StandardActivity: onStop:

I/CFragment: onAttach: //点击按钮 加载第一个Fragment

I/CFragment: onCreate:

I/CFragment: onCreateView:

I/CFragment: onActivityCreated:

I/CFragment: onViewStateRestored:

I/CFragment: onStart:

I/CFragment: onResume:

I/C2Fragment: onAttach: //点击按钮 加载第二个Fragment

I/C2Fragment: onCreate:

I/C2Fragment: onCreateView:

I/C2Fragment: onActivityCreated:

I/C2Fragment: onViewStateRestored:

I/C2Fragment: onStart:

I/C2Fragment: onResume: //显示出来，后期点击按钮进行切换，都不走生命周期回调了

I/CFragment: onPause: //后退到上一个页面

I/C2Fragment: onPause:

I/FragmentActivity: onPause:

I/StandardActivity: onRestart:

I/StandardActivity: onStart:

I/StandardActivity: onResume:

I/CFragment: onStop:

I/C2Fragment: onStop:

I/FragmentActivity: onStop:

I/CFragment: onDestroyView:

I/CFragment: onDestroy:

I/CFragment: onDetach:

I/C2Fragment: onDestroyView:

I/C2Fragment: onDestroy:

I/C2Fragment: onDetach:

I/FragmentActivity: onDestroy: