Android 部分 Activity 篇

1.Activity 是什么？

Activity 实际上只是一个与用户交互的接口而已。

2.Activity 生命周期

2.1 Activity 的 4 种状态

Active/Paused/Stopped/Killed

Activie:当前 Activity 正处于运行状态，指的是当前 Activity 获取了焦点。

Paused：当前 Activity 正处于暂停状态，指的是当前 Activity 失去焦点，此时的 Activity 并没有被销毁，内存里面的成员变量，状态信息等仍然存在，当然这个 Activity 也仍然可见，但是焦点却不在它身上，比如被一个对话框形式的 Activity 获取了焦点，或者被一个透明的 Activity 获取了焦点，这都能导致当前的 Activity 处于 paused 状态。

Stopped:与 paused 状态相似，stopped 状态的 Activity 是完全不可见的，但是内存里面的成员变量，状态信息等仍然存在，但是也没有被销毁。

Killed:已经被销毁的 Activity 才处于 killed 状态，它的内存里面的成员变量，状态信息等都会被一并回收。

2.2 Activity 的生命周期分析

正常情况下的生命周期：

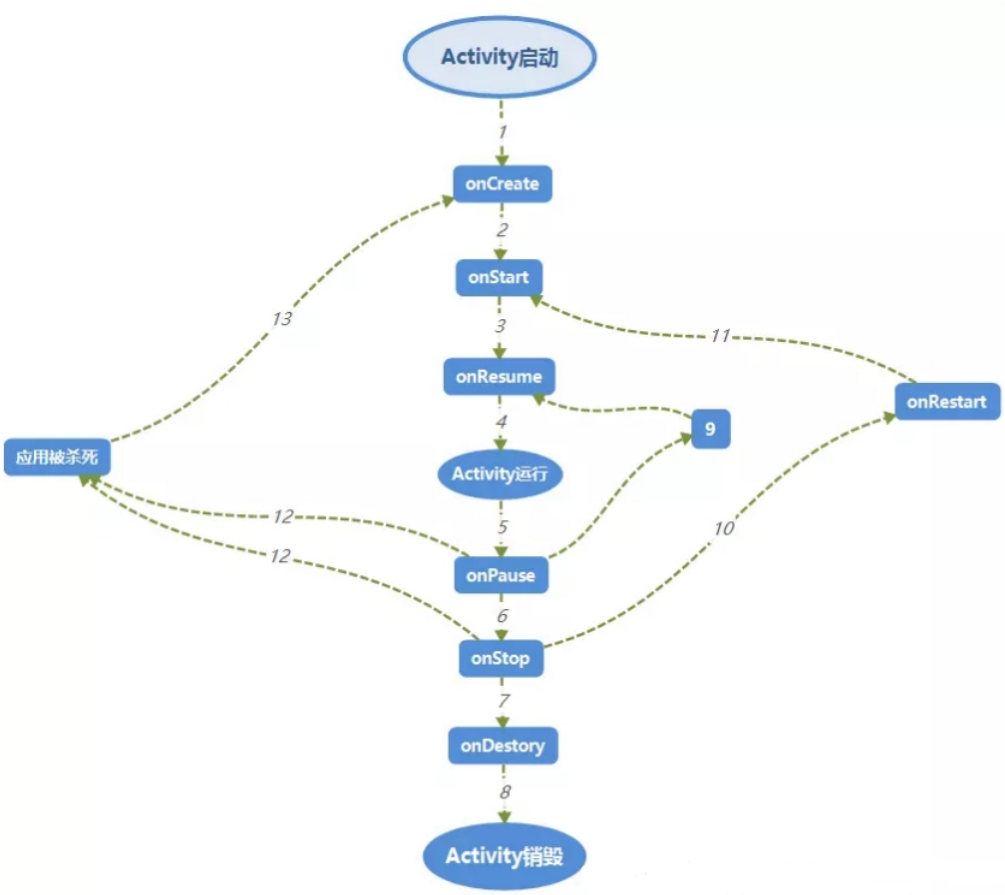
Activity 启动–>onCreate()–>onStart()–>onResume()

点击 home 键回到桌面–>onPause()–>onStop()

再次回到原 Activity–>onRestart()–>onStart()–>onResume()

退出当前 Activity 时–>onPause()–>onStop()–>onDestroy()

详细生命周期如下：



1.启动了一个 Activity,通常是 Intent 来完成。启动一个 Activity 首先要执行的回调函数是 onCreate(),通常在代码中你需要在此函数中绑定布局，绑定控件，初始化数据等做一些初始化的工作。

2.即将执行 Activity 的 onStart()函数，执行之后 Activity 已经可见，但是还没有出现在前台，无法与用户进行交互。这个时候通常 Activity 已经在后台准备好了，但是就差执行 onResume()函数出现在前台。

3.即将执行 Activity 的 onResume()函数，执行之后 Activity 不止可见而且还会出现在前台，可以与用户进行交互啦。

4.由于 Activity 执行了 onResume()函数，所以 Activity 出现在了前台。也就是 Activity 处于运行状态。

5.处于运行状态的 Activity 即将执行 onPause()函数，什么情况下促使 Activity 执行 onPause()方法呢？

[1]启动了一个新的 Activity

[2]返回上一个 Activity

可以理解为当需要其他 Activity，当前的 Activity 必须先把手头的工作暂停下来，再来把当前的界面空间交给下一个需要界面的 Activity，而 onPause()方法可以看作是一个转接工作的过程，因为屏幕空间只有那么一个，每次只允许一个 Activity 出现在前台进行工作。通常情况下 onPause()函数不会被单独执行，执行完 onPause()方法后会继续执行 onStop()方法，执行完 onStop()方法才真正意味着当前的 Activity 已经退出前台，存在于后台。

6.Activity 即将执行 onStop()函数，在“5”中已经说得很清楚了，当 Activity 要从前台切换至后台的时候会执行，比如：用户点击了返回键，或者用户切换至其他 Activity 等。

7.当前的 Activity 即将执行 onDestory()函数，代表着这个 Activity 即将进入生命的终结点，这是 Activity 生命周期中的最后一次回调生命周期，我们可以在 onDestory()函数中，进行一些回收工作和资源的释放工作，比如：广播接收器的注销工作等。

8.执行完 onDestory()方法的 Activity 接下来面对的是被 GC 回收，宣告生命终结。

9.很少情况下 Activity 才走“9”，网上一些关于对话框弹出后 Activity 会走“9”的说法，经过笔者验证，在某个 Activity 内弹出对话框并没有走“9”，所以网上大部分这样说法的文章要么是没验证，要么直接转载的，这个例子说明，实验出真知，好了，不废话了，那么什么情况下，Activity 会走“9”呢？

10.当用户在其他的 Activity 或者桌面回切到这个 Activity 时，这个 Activity 就会先去执行 onRestart()函数，Restart 有“重新开始”的意思，然后接下来执行 onStart()函数，接着执行 onResume()函数进入到运行状态。

11.在“10”中讲的很清楚了。

12.高优先级的应用急需要内存，此时处于低优先级的此应用就会被 kill 掉。

13.用户返回原 Activity。

下面来着重说明一下 Activity 每个生命周期函数：

onCreate():

表示 Activity 正在被创建，这是 Activity 生命周期的第一个方法。通常我们程序员要在此函数中做初始化的工作，比如：绑定布局，控件，初始化数据等。

onStart():

表示 Activity 正在被启动，这时候的 Activity 已经被创建好了，完全过了准备阶段，但是没有出现在前台，需要执行 onResume()函数才可以进入到前台与用户进行交互。

onResume():

表示 Activitiy 已经可见了，并且 Activity 处于运行状态，也就是 Activity 不止出现在了前台，而且还可以让用户点击，滑动等等操作与它进行交互。

onPause():

表示 Activity 正在暂停，大多数情况下，Activity 执行完 onPause()函数后会继续执行 onStop()函数，造成这种函数调用的原因是当前的 Activity 启动了另外一个 Activity 或者回切到上一个 Activity。还有一种情况就是 onPause()函数被单独执行了，并没有附带执行 onStop()方法，造成这种函数调用的原因很简单，就是当前 Activity 里启动了类似于对话框的东东。

onStop():

表示 Activity 即将停止，我们程序员应该在此函数中做一些不那么耗时的轻量级回收操作。

onRestart():

表示 Activity 正在重新启动。一般情况下，一个存在于后台不可见的 Activity 变为可见状态，都会去执行 onRestart()函数，然后会继续执行 onStart()函数，onResume()函数出现在前台并且处于运行状态。

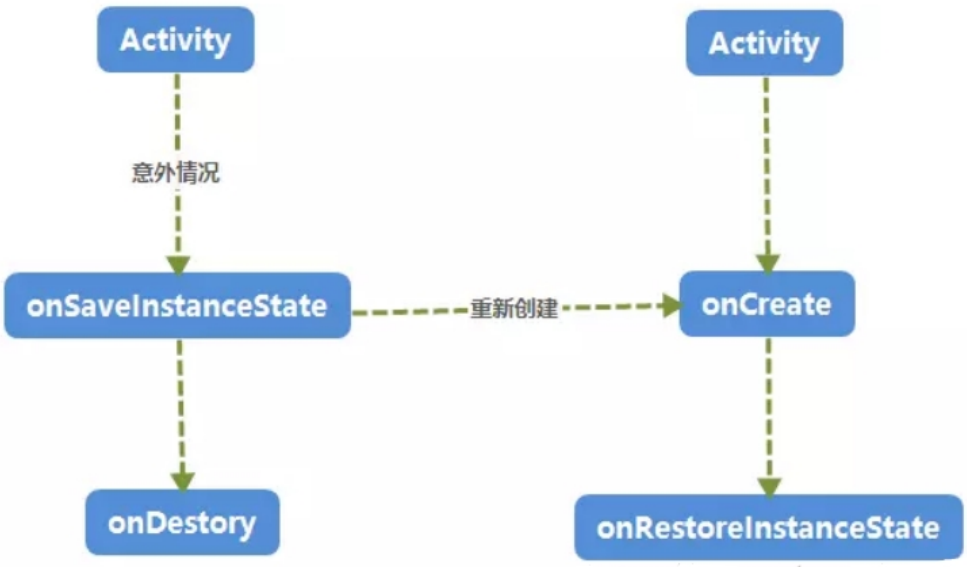
onDestory():

表示 Activity 要被销毁了。这是 Activity 生命中的最后一个阶段，我们可以在 onDestory()函数中做一些回收工作和资源释放等，比如：广播接收器的注销等。

异常情况下的生命周期：

什么是异常情况呢？

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。



可以从图中看出当 Activity 发生意外的情况的时候，这里的意外指的就是系统配置发生改变，Activity 会被销毁，其 onPause,OnStop,onDestory 函数均会被调用，同时由于 Actiivty 是在异常情况下终止的，系统会调用 onSaveInstanceState 来保存当前 Activity 状态。调用 onSaveInstanceState 的时机总会发生在 onStop 之前，至于会不会调用时机发生在 onPause 方法之前，那就说不定了，这个没有固定的顺序可言，正常情况下一般 onSaveInstanceState 不会被调用。当 Activity 被重新创建后，系统会调用 onRestoreInstanceState,并且把 Actiivty 销毁时 onSaveInstanceState 方法所保存的 Bundle 对象作为参数传递给 onRestoreInstanceState 和 onCreate 方法。所以我们可以通过 onRestoreInstanceState 和 onCreate 方法来判断 Actiivty 是否被重建了，如果被重建了，那么我们就可以取出之前保存的数据并恢复，从时序上来看，onRestoreInstanceState 的调用时机发生在 onStart 之后。

同时，在 onSaveInstanceState 和 onRestoreInstanceState 方法中，系统自动为我们做了一定的恢复工作。当 Activity 在异常情况下需要重新创建时，系统会默认为我们保存当前 Activity 的视图结构。当 Activity 在异常情况下需要重新创建时，系统会默认为我们保存当前 Activity 的视图结构，并且在 Activity 重启后为我们恢复这些数据，比如：文本框中用户输入的数据,ListView 滚动的位置等，这些 View 相关的状态系统都能够默认为我们恢复。具体针对某一个特定的 View 系统 能为我们恢复哪些数据，我们可以查看 View 的源码。和 Activity 一样，每个 View 都有 onSaveInstanceState 和

onRestoreInstanceState 这两个方法，看一下它们的具体实现，就能知道系统能够自动为每个 View 恢复哪些数据。

关于保存和恢复 View 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 在委托它上面的顶级容器去保存数据。顶级容器是一个 ViewGroup，一般来说它很可能是 DecorView。最后顶层容器再去一一通知它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，这是一个典型的委托思想，上层委托下层，父容器去委托子元素去处理一件事情，这种思想在 Android 中有很多应用，比如：View 的绘制过程，事件分发等都是采用类似的思想。至于数据恢复过程也是类似的，这样就不再重复介绍了。

情况 2：资源内存不足导致低优先级的 Activity 被杀死。

首先，Activity 有优先级？你肯定怀疑，代码中都没设置过啊！优先级从何而来，其实这里的 Activity 的优先级是指一个 Activity 对于用户的重要程度，比如：正在与用户进行交互的 Activity 那肯定是最重要的。我们可以按照重要程度将 Activity 分为以下等级：

优先级最高： 与用户正在进行交互的 Activity，即前台 Activity。

优先级中等：可见但非前台的 Activity,比如：一个弹出对话框的 Activity,可见但是非前台运行。

优先级最低：完全存在与后台的 Activity,比如：执行了 onStop。

当内存严重不足时，系统就会按照上述优先级去 kill 掉目前 Activity 所在的进程，并在后续通过 onSaveInstanceState 和 onRestoreInstanceState 来存储和恢复数据。如果一个进程中没有四大组件的执行，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件独立运行在后台中，这样进程更容易被杀死。比较好的方法就是将后台工作放入 Service 中从而保证进程有一定的优先级，这样就不会轻易地被系统杀死。

总结：

上面分析了系统的数据存储和恢复机制，我们知道，当系统配置发生改变之后，Activity 会被重新创建，那么有没有办法不重新创建呢？答案是有的，接下来我们就来分析这个问题。系统配置中有很多内容，如果某项内容发生了该变后，我们不想系统重新创建 Activity 可以给 Activity 指定 configChanges 属性。比如我们不想让 Actiivty 在屏幕旋转的时候重新创建，就可以给 configChanges 属性添加一些值，请继续往下看。

2.3 一些特殊情况下的生命周期分析

2.3.1 Activity 的横竖屏切换

与横竖屏生命周期函数有关调用的属性是"android:configChanges",关于它的属性值设置影响如下：

- orientation：消除横竖屏的影响
- keyboardHidden：消除键盘的影响
- screenSize：消除屏幕大小的影响

当我们设置 Activity 的 android:configChanges 属性为 orientation 或者 orientation|keyboardHidden 或者不设置这个属性的时候，它的生命周期会走如下流程：

```
1.    刚刚启动 Activity 的时候：
2.    onCreate
3.    onStart
4.    onResume
5.    由竖屏切换到横屏：
6.    onPause
7.    onSaveInstanceState //这里可以用来横竖屏切换的保存数据
8.    onStop
9.    onDestroy
10.   onCreate
11.   onStart
12.   onRestoreInstanceState//这里可以用来横竖屏切换的恢复数据
13.   onResume
14.   横屏切换到竖屏：
15.   onPause
16.   onSaveInstanceState
17.   onStop
18.   onDestroy
19.   onCreate
20.   onStart
21.   onRestoreInstanceState
22.   onResume
```

当我们设置 Activity 的 android:configChanges 属性为 orientation|screenSize 或者 orientation|screenSize|keyboardHidden

```
1.  刚刚启动 Activity 的时候：
2.  onCreate
3.  onStart
4.  onResume
5.  由竖屏切换到横屏：
6.
7.  什么也没有调用
8.  横屏切换到竖屏：
9.  什么也没有调用
```

而且需要注意一点的是设置了 orientation|screenSize 属性之后，在进行横竖屏切换的时候调用的方法是 onConfigurationChanged()，而不会回调 Activity 的各个生命周期函数；

当然在显示中我们可以屏蔽掉横竖屏的切换操作，这样就不会出现切换的过程中 Activity 生命周期重新加载的情况了，具体做法是，在 Activity 中加入如下语句：

```
1.  android:screenOrientation="portrait" 始终以竖屏显示
2.  android:screenOrientation="landscape" 始终以横屏显示
```

如果不想设置整个软件屏蔽横竖屏切换，只想设置屏蔽某个 Activity 的横竖屏切换功能的话，只需要下面操作：

```
1.  Activity.this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRA
    IT);以竖屏显示
2.  Activity.this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSC
    APE);以横屏显示
```

最后提一点,当你横竖屏切换的时候，如果走了销毁 Activity 的流程，那么需要保存当前和恢复当前 Activity 的状态的话，我们可以灵活运用 onSaveInstanceState()方法和 onRestoreInstanceState()方法。

2.3.2 什么时候 Activity 单独走 onPause()不走 onStop()？

关于这个特殊情况，笔者在上面的生命周期图解析的时候，贴了一个链接，这里主要是检验你是否会了这个问题的答案，这里笔者就不贴答案了，答案全在那个链接里，你会了吗？

2.3.3 什么时候导致 Activity 的 onDestory()不执行？

当用户后台强杀应用程序时，当前返回栈仅有一个 activity 实例时，这时候，强杀，是会执行 onDestroy 方法的；当返回栈里面存在多个 Activity 实例时，栈里面的第一个没有销毁的 activity 执行会 ondestroy 方法，其他的不会执行；比如说：从 mainactivity 跳转到 activity-A（或者继续从 activity-A 再跳转到 activity-B），这时候，从后台强杀，只会执行 mainactivity 的 onDestroy 方法，activity-A（以及 activity-B）的 onDestroy 方法都不会执行；

2.4 进程的优先级

前台>可见>服务>后台>空

前台：与当前用户正在交互的 Activity 所在的进程。

可见：Activity 可见但是没有在前台所在的进程。

服务：Activity 在后台开启了 Service 服务所在的进程。

后台：Activity 完全处于后台所在的进程。

空：没有任何 Activity 存在的进程，优先级也是最低的。

3.Android 任务栈

任务栈与 Activity 的启动模式密不可分，它是用来存储 Activity 实例的一种数据结构，Activity 的跳转以及回跳都与这个任务栈有关。详情请看下面的 Activity 的启动模式。

4.Activity 的启动模式

Activity 的启动模式，你在初学期间一定很熟悉了吧！不管你是否熟悉还是不熟悉，跟随笔者的思路把 Activity 的启动模式整理一遍：

问题 1：Activity 为什么需要启动模式？

问题 2：Activity 的启动模式有哪些？特性如何

问题 3：如何给 Activity 选择合适的启动模式

问题 1：Activity 为什么需要启动模式？

我们都知道启动一个 Activity 后，这个 Activity 实例就会被放入任务栈中，当点击返回键的时候，位于任务栈顶层的 Activity 就会被清理出去，当任务栈中不存在任何 Activity 实例后，系统就回去回收这个任务栈，也就是程序退出了。这只是对任务栈的基本认识，深入学习，笔者会在之后文章中提到。那么问题来了，既然每次启动一个 Activity 就会把对应的要启动的 Activity 的实例放入任务栈中，假如这个 Activity 会被频繁启动，那岂不是会生成很多这个 Activity 的实例吗？对内存而言这可不是什么好事，明明可以一个 Activity 实例就可以应付所有的启动需求，为什么要频繁生成新的 Activity 实例呢？杜绝这种内存的浪费行为，所以 Activity 的启动模式就被创造出来去解决上面所描述的问题。

问题 2：Activity 的启动模式有哪些？特性如何

Activity 的启动模式有 4 种，分别是：standard,singleTop,singleTask 和 singleInstance。下面一一作介绍：

1.系统默认的启动模式:Standard

标准模式，这也是系统的默认模式。每次启动一个 Activity 都会重新创建一个新的实例，不管这个实例是否存在。被创建的实例的生命周期符合典型情况下的 Activity 的生命周期。在这种模式下，谁启动了这个 Activity,那么这个 Activity 就运行在启动它的那个 Activity 的任务栈中。比如 Activity A 启动了 Activity B(B 是标准模式)，那么 B 就会进入到 A 所在的任务栈中。有个注意的地方就是当我们用 ApplicationContext 去启动 standard 模式的 Activity 就会报错，这是因为 standard 模式的 Actiivty 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，所以这就会出现错误。解决这个问题的方法就是为待启动的 Activity 指定 FLAG_ACTIVITY_NEW_TASK 标记位，这样启动的时候就会为它创建一个新的任务栈，这个时候启动 Activity 实际上以 singleTask 模式启动的，读者可以自己仔细体会。

2.栈顶复用模式：SingleTop

在这种模式下，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，同时它的 onNewIntent 方法被回调，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，因为它并没有发生改变。如果新的 Activity 已经存在但不是位于栈顶，那么新的 Activity 仍然会重新重建。举个例子，假设目前栈内的情况为 ABCD,其中 ABCD 为四个 Activity,A 位于栈低，D 位于栈顶，这个时候假设要再次启动 D,如果 D 的启动模式为 singleTop,那么栈内的情况依然为 ABCD;如果 D 的启动模式为 standard,那么由于 D 被重新创建，导致栈内的情况为 ABCDD。

3.栈内复用模式：SingTask

这是一种单例实例模式，在这种模式下，只要 Activity 在一个栈中存在，那么多次启动此 Activity 都不会重新创建实例，和 singleTop 一样，系统也会回调其 onNewIntent。具体一点，当一个具有 singleTask 模式的 Activity 请求启动后，比如 Activity A，系统首先寻找任务栈中是否已存在 Activity A 的实例，如果已经存在，那么系统就会把 A 调到栈顶并调用它的 onNewIntent 方法，如果 Activity A 实例不存在，就创建 A 的实例并把 A 压入栈中。举几个栗子：

- 比如目前任务栈 S1 的情况为 ABC,这个时候 Activity D 以 singleTask 模式请求启动，其所需的任务栈为 S2，由于 S2 和 D 的实例均不存在，所以系统会先创建任务栈 S2,然后再创建 D 的实例并将其投入到 S2 任务栈中。
- 另外一种情况是，假设 D 所需的任务栈为 S1,其他情况如同上面的例子所示，那么由于 S1 已经存在，所以系统会直接创建 D 的实例并将其投入到 S1。
- 如果 D 所需的任务栈为 S1,并且当前任务栈 S1 的情况为 ADBC,根据栈内复用的原则，此时 D 不会重新创建，系统会把 D 切换到栈顶并调用其 onNewIntent 方法，同时由于 singleTask 默认具有 clearTop 的效果，会导致栈内所有在 D 上面的 Activity 全部出栈，于是最终 S1 中的情况为 AD。

通过以上 3 个例子，你应该能比较清晰地理解 singleTask 的含义了。

4.单实例模式：SingleInstance

这是一种加强的 singleTask 模式，它除了具有 singleTask 模式所有的特性外，还加强了一点，那就是具有此种模式的 Activity 只能单独位于一个任务栈中，换句话说，比如 Activity A 是 singleInstance 模式，当 A 启动后，系统会为它创建一个新的任务栈，然后 A 独自在这个新的任务栈中，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。

对于 SingleInstance,面试时你有说明它的以下几个特点：

（1）以 singleInstance 模式启动的 Activity 具有全局唯一性，即整个系统中只会存在一个这样的实例。

（2）以 singleInstance 模式启动的 Activity 在整个系统中是单例的，如果在启动这样的 Activiyt 时，已经存在了一个实例，那么会把它所在的任务调度到前台，重用这个实例。

（3）以 singleInstance 模式启动的 Activity 具有独占性，即它会独自占用一个任务，被他开启的任何 activity 都会运行在其他任务中。

（4）被 singleInstance 模式的 Activity 开启的其他 activity，能够在新的任务中启动，但不一定开启新的任务，也可能在已有的一个任务中开启。

换句话说，其实 SingleInstance 就是我们刚才分析的 SingleTask 中，分享 Activity 为栈底元素的情况。

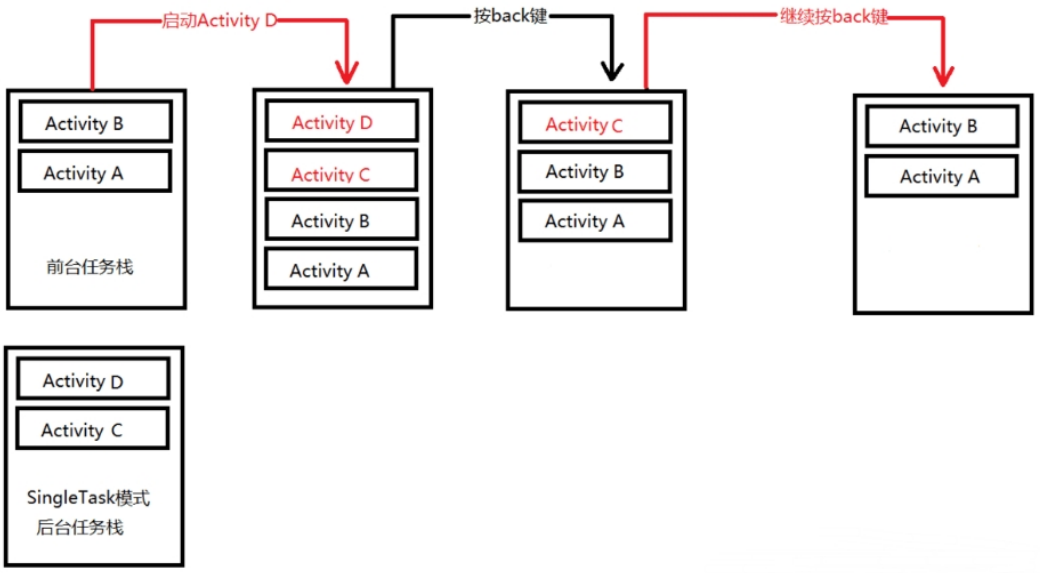
总结

上面介绍了 4 种启动模式，这里需要指出一种情况，我们假设目前有 2 个任务栈，前台任务栈的情况为 AB,而后台任务栈的情况为 CD，这里假设 CD 的启动模式均为 singleTask。现在请求启动 D,那么整个后台任务栈都会被切换到前台，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，如下图 1 所示：

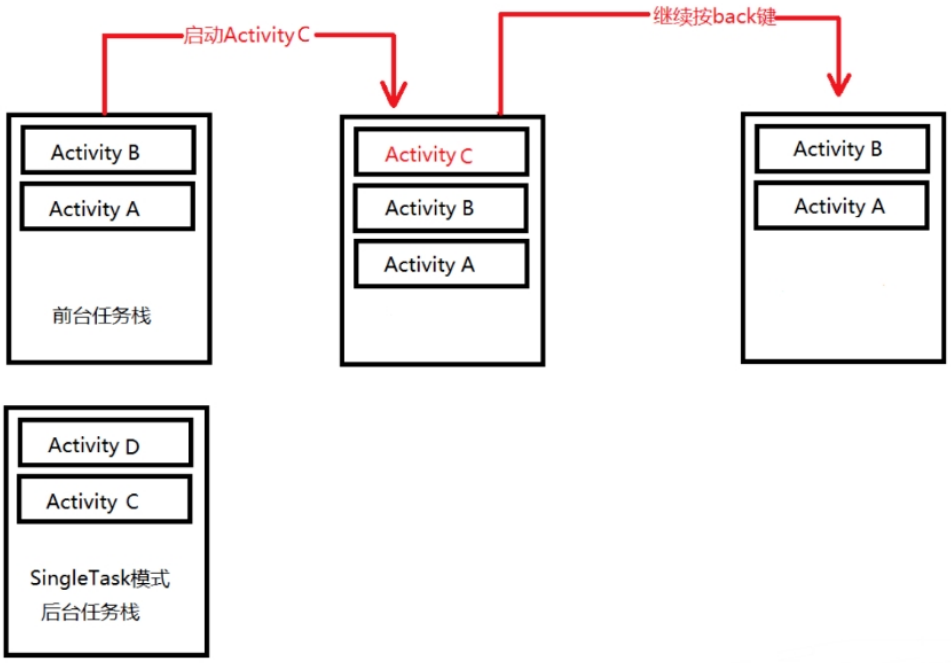
注意：

前台任务栈：就是指和用户正在交互的应用程序所在的任务栈。

后台任务栈：就是指处于后台的应用程序所在的任务栈。



如果不是请求的 D 而是请求的 C,那么情况就不一样了，如下图 2 所示：



如何指定活动的启动模式呢？在 AndroidManifest.xml 文件当注册活动的代码中去指定

比如：我要把 MainActivity 活动的启动模式指定为 singleInstance 模式

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name"
    android:launchMode="singleInstance">

    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

</activity>
```

也可以在代码中指定：

```
1.  Intent pack = new Inten(MCPersonalCenterActivity.this,MCGiftsCenterActivity.
    class);
```

```
2.  pack.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
3.  startActivity(pack);
```

5.Activity 组件之间的通信

1.Activity->Activity

[1]Intent/Bundle

这种方式多用于 Activity 之间传递数据。示例代码如下：

```
1.  //首先创建一个 Bundle 对象
2.      Bundle bundle = new Bundle();
3.      bundle.putString("data_string","数据");
4.      bundle.putInt("data_int",10);
5.      bundle.putChar("da_char",'a');
6.
7.      //然后创建一个 Intent 对象
8.      Intent intent = new Intent(FirstActivity.this,SecondActivity.class);
9.      intent.putExtras(bundle);
10.     startActivity(intent);
```

[2]类静态变量

在 Activity 内部定义静态的变量，这种方式见于少量的数据通信，如果数据过多，还是使用第一种方式。示例代码如下：

```
1.  public class FirstActivity extends AppCompatActivity {
2.
3.      //声明为静态
4.      static boolean isFlag = false;
5.
6.      @Override
7.      protected void onCreate(Bundle savedInstanceState) {
8.          super.onCreate(savedInstanceState);
9.          setContentView(R.layout.activity_first);
10.
11.         //首先创建一个 Bundle 对象
12.         Bundle bundle = new Bundle();
```

```
        bundle.putString("data_string","数据");
        bundle.putInt("data_int",10);
        bundle.putChar("da_char",'a');

        //然后创建一个 Intent 对象
        Intent intent = new Intent(FirstActivity.this,SecondActivity.class);

        intent.putExtras(bundle);
        startActivity(intent);


    }

}
```

[3]全局变量

创建一个类，里面定义一批静态变量，Activity 之间通信都可以访问这个类里面的静态变量，这就是全局变量。这种方式笔者就不给代码了。

2.Activity->Service

[1]绑定服务的方式，利用 ServiceConnection 这个接口。

首先我们需要在要绑定的服务中声明一个 Binder 类

```
public class MyService1 extends Service {

    public String data = "";

    public MyService1() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        return new Binder();
```

```
    }


    public class Binder extends android.os.Binder{

        public void sendData(String data){

            MyService1.this.data = data;

        }

    }

}
```

然后我们让 Activity 实现 ServiceConnection 这个接口，并且在 onServiceConnected 方法中获取到 Service 提供给 Activity 的 Binder 实例对象，通过这个对象我们就可以与 Service 进行通信可以通过上述代码的 Binder 类中的 sendData()方法进行通信。

```
public class ServiceBindActivity extends AppCompatActivity implements Servic
eConnection,View.OnClickListener {

    private Button bt0,bt1,bt2;

    public MyService1.Binder binder = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_bind);

        bt0 = findViewById(R.id.bt0);
        bt1 = findViewById(R.id.bt1);
        bt2 = findViewById(R.id.bt2);

        bt0.setOnClickListener(this);
        bt1.setOnClickListener(this);
        bt2.setOnClickListener(this);


    }
```

```
22.
23.     @Override
24.     protected void onDestroy() {
25.        super.onDestroy();
26.        unbindService(this);
27.     }
28.
29.     //这个是服务绑定的时候调用
30.     @Override
31.     public void onServiceConnected(ComponentName componentName, IBinder iBin
    der) {
32.         binder = (MyService1.Binder) iBinder;
33.     }
34.
35.     //这个是服务解绑的时候调用
36.     @Override
37.     public void onServiceDisconnected(ComponentName componentName) {
38.
39.     }
40.
41.     @Override
42.     public void onClick(View view) {
43.
44.         switch (view.getId()){
45.
46.             case R.id.bt0:
47.
48.                 //绑定服务
49.                 Intent intent = new Intent(ServiceBindActivity.this,MyServic
    e1.class);
50.                 bindService(intent,this, Context.BIND_AUTO_CREATE);
51.
52.                break;
53.
54.             case R.id.bt1:
55.
56.                 //通过 binder 对象来和 Service 进行通信
57.                 if(binder != null)
58.                 binder.sendData("bt1");
59.
60.                 break;
61.             case R.id.bt2:
62.
63.                 //通过 binder 对象来和 Service 进行通信
```

```
                if(binder != null)
                binder.sendData("bt2");

                break;

        }

    }
}
```

也不一定非要笔者这种写法，你也可以有自己的写法，但核心部分都一样。

[2]Intent

这种方式很简单，我们在启动和停止 Service 时所调用的方法都需要传入一个 Intent 实例对象，通过这个传入的 Intent 对象，我们就可以与 Service 进行通信。示例代码如下：

Activity 代码是这样的：

```
public class ServiceStartActivity extends AppCompatActivity implements View.OnClickListener {

    private Button bt0,bt1;

    private Intent intent ;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_start);

        intent = new Intent(this, MyService2.class);

        bt0 = findViewById(R.id.bt0);
        bt1 = findViewById(R.id.bt1);

        bt0.setOnClickListener(this);
        bt1.setOnClickListener(this);

```

```
    }

    @Override
    public void onClick(View view) {

        switch (view.getId()){

            case R.id.bt0:

                //开启服务并且传递数据
                intent.putExtra("data_stirng","string 数据");
                startActivity(intent);

                break;

            case R.id.bt1:

                //结束服务
                stopService(intent);

                break;

        }

    }
}
```

Service 中的代码是这样的：

```
public class MyService2 extends Service {

    public String data = "";

    public MyService2() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        // TODO: Return the communication channel to the service.
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
```

```
        //得到 Activity 传递过来的数据
        data = intent.getStringExtra("data_string");
        return super.onStartCommand(intent, flags, startId);
    }
}
```

这种通信方式的缺点显而易见，那就是只能传递少量的数据。

[3]CallBack + Handler,监听服务的进程变化

Service 中的代码：

```
public class MyService3 extends Service {

    //在 Service 中如果要进行耗时任务，可以通过 CallBack 接口提供的方法与 Activity 进行通信
    public Callback callback;

    public MyService3() {
    }

    @Override
    public IBinder onBind(Intent intent) {
       // TODO: Return the communication channel to the service.
       return new Binder();
    }

    public void setCallBack(CallBack callBack){
        this.callback = callback;
    }

    public Callback getCallback() {
        return callback;
    }

    public interface CallBack{
        void onDataChange(String data);
    }

    public class Binder extends android.os.Binder{

        public MyService3 getMyService3(){
```

```
            return MyService3.this;
        }

    }
}
```

Activity 中的代码：

```
public class ServiceBind2Activity extends AppCompatActivity implements ServiceConnection{



    public MyService3.Binder binder = null;

    private Handler handler = new Handler(){

        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);

            Bundle bundle = msg.getData();
            String data_string = bundle.getString("data_string");

            //接下来就是更新 ui

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_service_bind2);
    }

    @Override
    public void onServiceConnected(ComponentName componentName, IBinder iBinder) {

        binder = (MyService3.Binder) iBinder;
        binder.getMyService3().setCallBack(new MyService3.CallBack() {

            //此方法提供给 MyService3 在子线程中调用
```

```
            @Override
            public void onDataChange(String data) {
                Message message = new Message();
                Bundle bundle = new Bundle();
                bundle.putString("data_string","String 数据");
                message.setData(bundle);
                //通过 Handler 进行异步通信，不过耗时操作放在 MyService3 中
                handler.sendMessage(message);
            }
        });

    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {

    }


}
```

可能第一次看到这段代码的你很懵逼吧，其实很简单，当 ServiceBind2Activity 去绑定服务 MyService3 的时候,那么在 Activity 中的 onServiceConnected()方法被调用，此时位于 MySerivce3 的 CallBack 接口引用被实例化，并且 onDataChange()方法被实现，可以看到里面是一段 Handler 通信的代码，不错，这个方法是为 MyService3 做耗时操作调用的，笔者没有在 MyService3 中写耗时操作的代码，不过说到这里你应该明白了这种通信方式的好处了吧，也印证了标题：监听服务的进程变化。

3.Activity->Fragment

[1]Bundle

在创建 Fragment 实例的时候，调用方法 setArguments 将一个 Bundle 对象传递给 Fragment，然后在 Fragment 中先去判断是否和当前 Activity 绑定上了，如果绑定上了，就可以拿出这个 Bundle 中的数据啦。示例代码如下：

在 Activity 中代码是这样的：

```
1.  //首先创建一个 Bundle 对象
2.    Bundle bundle = new Bundle();
3.    bundle.putString("data_string","数据");
4.    bundle.putInt("data_int",10);
5.    bundle.putChar("da_char",'a');
6.
7.    Fragment fragment = new MyFragment1();
8.    fragment.setArguments(bundle);
```

在 MyFragment1 中代码是这样的：

```
1.  if(isAdded()){//这里判断是否 Fragment 和 Activity 进行了绑定
2.
3.     Bundle bundle = getArguments();
4.     String data_string = bundle.getString("data_string");
5.     String data_int = bundle.getInt("data_int");
6.     String data_char = bundle.getChar("data_char");
7.
8.
9.  }
```

对于这个 isAdded()方法笔者还需要提出一点，为什么要这么写呢？因为如果这个 Fragment 没有和 Activity 绑定的话，那么那个 Bundle 对象是无法从 Activity 传递给 Fragment 的，因此这种写法是必须的。

[2]直接进行方法调用

在 Activity 里通过 Fragment 的引用，可以直接调用 Framgent 中的定义的任何方法。示例代码如下：

```
1.  MyFragment1 myFragment1 = new MyFragment1();
2.  myFragment.toString("传送的 string 数据");
```

6. scheme 跳转协议

Android 中的 scheme 是一种页面内跳转协议，通过自定义 scheme 协议，可以非常方便的跳转到 app 中的各个页面，通过 scheme 协议，服务器可以定制化告诉 app 跳转到哪个页面，可以通过通知栏消息定制化跳转页面，可以通过 H5 页面跳转到相应页面等等。