**Android Intent FLAG介绍**

2018年02月27日 11:46:41 [秦川小将](https://me.csdn.net/mjb00000) 阅读数：1521

 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/mjb00000/article/details/79386308

1.FLAG\_GRANT\_READ\_URI\_PERMISSION   
临时访问读权限 intent的接受者将被授予 INTENT 数据uri 或者 在ClipData 上的读权限。

2.FLAG\_GRANT\_WRITE\_URI\_PERMISSION   
临时访问写权限 intent的接受者将被授予 INTENT 数据uri 或者 在ClipData 上的写权限。

3.FLAG\_GRANT\_PERSISTABLE\_URI\_PERMISSION   
区别于 FLAG\_GRANT\_READ\_URI\_PERMISSION 跟 FLAG\_GRANT\_WRITE\_URI\_PERMISSION， URI权限会持久存在即使重启，直到明确的用 revokeUriPermission(Uri, int) 撤销。 这个flag只提供可能持久授权。但是接收的应用必须调用ContentResolver的takePersistableUriPermission(Uri, int)方法实现 。

4.FLAG\_GRANT\_PREFIX\_URI\_PERMISSION   
权限授予任何原始授权URI前缀匹配的URI。

5.FLAG\_DEBUG\_LOG\_RESOLUTION   
解析intent时打印log messages，展示创建最终的resolved list 找到的信息 。比如有如下代码 ：

Intent intent = new Intent(“android.provider.Telephony.SMS\_RECEIVED”);

intent.addFlags(Intent.FLAG\_DEBUG\_LOG\_RESOLUTION);

sendBroadcast(intent);

将会按照优先级打印出系统所有注册”android.provider.Telephony.SMS\_RECEIVED”的广播接收者。

6.FLAG\_FROM\_BACKGROUND   
指明Intent来自后台操作 ，不是来自用户直接互动。

7.FLAG\_[ACTIVITY](https://www.baidu.com/s?wd=ACTIVITY&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)\_BROUGHT\_TO\_FRONT   
通常不是通过应用程序代码设置，而是通过系统如launchMode singleTask模式。

8.FLAG\_ACTIVITY\_CLEAR\_TASK   
如果在通过Context.startActivity()启动activity时为Intent设置了此标识，这个标识将导致：任何与此activity相关联的task都会被清除。也就是说， 此activity将变成一个空栈中新的最底端的activity，所有的旧activity都会被finish掉，这个标识仅仅和FLAG\_ACTIVITY\_NEW\_TASK联合起来才能使用。

9.FLAG\_ACTIVITY\_CLEAR\_TOP   
当设置此标致，并且acitivity已经启动，那么不是启动一个新的activity，所有其他顶部的activity都会关闭，这个intent将被交付到（现在顶部）老的activity 做为新的intent。如果一个task由A,B,C,D组成，如果D调用startActivity（），跳到B, 然后C,D被finish掉，B接收新的intent ，结束栈中：A,B.现在运行的B的实例或者在onNewIntent方法中接收你start的新intent，或者自己finish掉然后重启一个新的intent。如果声明启动了启动模式是“multiple”(默认)，并且你没有在这个intent中设置FLAG\_ACTIVITY\_SINGLE\_TOP，就会finish掉然后重新创建。其他的启动模式。或者FLAG\_ACTIVITY\_SINGLE\_TOP被设置了，intent将会传送到当前实例的onNewIntent方法中。这个启动模式也可以跟FLAG\_ACTIVITY\_NEW\_TASK结合使用：如果用来start根activity，它将会在此task任务当前正在执行的实例bring to foreground，然后清除到跟状态。比如，当从notification manager启动一个activity。

10.FLAG\_ACTIVITY\_CLEAR\_WHEN\_TASK\_RESET   
API21过期，被FLAG\_ACTIVITY\_NEW\_DOCUMENT代替。

11.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS   
如果设置，新的Activity不会在最近启动的Activity的占中保存。

12.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。

13.FLAG\_ACTIVITY\_LAUNCHED\_FROM\_HISTORY   
这个标记通常不由应用程序代码来设置，如果是从历史中启动这个Activity，系统就会设置这个标记(长按[home键](https://www.baidu.com/s?wd=home%E9%94%AE&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)) 。

14.FLAG\_ACTIVITY\_MULTIPLE\_TASK   
可以跟FLAG\_ACTIVITY\_MULTIPLE\_TASK结合使用，当只用自己的时候相当于Manifast中android.R.attr.documentLaunchMode=”intoExisting”，当跟FLAG\_ACTIVITY\_MULTIPLE\_TASK结合使用相当于 Manifast中android.R.attr.documentLaunchMode=”always”.

15.FLAG\_ACTIVITY\_NEW\_DOCUMENT   
默认情况FLAG\_ACTIVITY\_NEW\_DOCUMENT创建的document当用户关闭时之前tasks的entry会被remove掉，如果想保持在历史中一遍重新launch，就要用到这个flag.当使task的activity finish掉以后，历史entry将保持在界面以便用户重新打开类似顶级应用程序的历史。

16.FLAG\_ACTIVITY\_NEW\_TASK   
如果设置了，这个Activity将会成为新任务历史栈的开始，如果已经有一个task运行着邀请新的activity，将不会启动新的activity；当前任务栈最后状态将会被展示在屏幕上查看FLAG\_ACTIVITY\_MULTIPLE\_TASK ，关闭这一特性。

17.FLAG\_ACTIVITY\_NO\_ANIMATION   
如果设置，将阻止系统get next activity的过渡动画。并不意味着一直不会有动画，如果另一个activity 的变化发生没有在start activity 显示之前指定，会有过渡动画。

18.FLAG\_ACTIVITY\_NO\_HISTORY   
如果设置，新的activity将不会保存在历史栈中。一旦用户离开这个activity，它就会被finish掉。也可以在manifest.xml中设置activity android:hoHistory属性设置。如果设置， OnActivityResult()方法将不会再被调用 。

19.FLAG\_ACTIVITY\_NO\_USER\_ACTION   
onUserLeaveHint()作为activity周期的一部分，它在activity因为用户要跳转到别的activity而要退到background时使用。比如,在用户按下Home键，它将被调用。比如有电话进来（不属于用户的选择），它就不会被调用。如果设置，作为新启动的Activity进入前台时，这个标志将在Activity暂停之前阻止从最前方的Activity回调的onUserLeaveHint()。典型的，一个Activity可以依赖这个回调指明显式的用户动作引起的Activity移出后台。这个回调在Activity的生命周期中标记一个合适的点，并关闭一些Notification。 如果一个Activity通过非用户驱动的事件，如来电或闹钟，启动的，这个标志也应该传递给Context.startActivity，保证暂停的Activity不认为用户已经知晓其Notification。

20.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP   
如果给Intent对象设置了这个标记，并且这个Intent对象被用于从一个既存的Activity中启动一个新的Activity，这个Activity不被看作决定是否传送新的intent到top而不是start新的，通常认为使用这个flag启动的Activity会被自己立即终止。

21.FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED   
FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED:如果设置该属性，并且这个activity在一个新的task中正在被启动或者被带到一个已经存在的task的顶部，这时这个activity将会被作为这个task的首个页面加载。这将会导致拥有这个应用的affinities的task处于一个合适的状态(移动activity到这个task或者activity从中移出)，或者简单的重置这个task到它的初始状态

22.FLAG\_ACTIVITY\_REORDER\_TO\_FRONT   
如果在intent里设置交给 startActivity（）,这个flag会把已经运行过的acivity带到task历史栈的顶端。例如，一个task由A,B,C,D四个activity组成，如果D携带这个flag的intent调用startActivity()打开B，那么B就会被带到历史栈的前部，结果是:A,C,D,B.如果LAG\_ACTIVITY\_CLEAR\_TOP 被设置，那么FLAG\_ACTIVITY\_REORDER\_TO\_FRONT将被忽略。

23.FLAG\_ACTIVITY\_SINGLE\_TOP   
如果设置了，如过Activity在栈顶将不会启动。

24.FLAG\_ACTIVITY\_TASK\_ON\_HOME   
把当前新启动的任务置于Home任务之上，也就是按back键从这个任务返回的时候会回到home，即使这个不是他们最后看见的activity，注意这个标记必须和FLAG\_ACTIVITY\_NEW\_TASK一起使用

25.FLAG\_RECEIVER\_REGISTERED\_ONLY   
设置这个flag，发送广播只有动态注册才能调用，组件(xml 中定义action)不会被被launch

[**Android总结篇系列：Activity启动模式（lauchMode）**](https://www.cnblogs.com/lwbqqyumidi/p/3771542.html)

本来想针对Activity中的启动模式写篇文章的，后来网上发现有人已经总结的相当好了，在此直接引用过来，并加上自己的一些理解，在此感谢原作者。

文章地址：

<http://blog.csdn.net/liuhe688/article/details/6754323>

-------------------------------------------------------------------------------------------------

launchMode在多个Activity跳转的过程中扮演着重要的角色，它可以决定是否生成新的Activity实例，是否重用已存在的Activity实例，是否和其他Activity实例公用一个task里。这里简单介绍一下task的概念，task是一个具有栈结构的对象，一个task可以管理多个Activity，启动一个应用，也就创建一个与之对应的task。

Activity一共有以下四种launchMode：

1.standard

2.singleTop

3.singleTask

4.singleInstance

我们可以在AndroidManifest.xml配置<activity>的android:launchMode属性为以上四种之一即可。

下面我们结合实例一一介绍这四种lanchMode：

**1.standard**

standard模式是默认的启动模式，不用为<activity>配置android:launchMode属性即可，当然也可以指定值为standard。

我们将会一个Activity，命名为FirstActivity，来演示一下标准的启动模式。FirstActivity代码如下：

[复制代码](javascript:void(0);)

1 package com.scott.launchmode;

2

3 import android.app.Activity;

4 import android.content.Intent;

5 import android.os.Bundle;

6 import android.view.View;

7 import android.widget.Button;

8 import android.widget.TextView;

9

10 public class FirstActivity extends Activity {

11 @Override

12 public void onCreate(Bundle savedInstanceState) {

13 super.onCreate(savedInstanceState);

14 setContentView(R.layout.first);

15 TextView textView = (TextView) findViewById(R.id.textView);

16 textView.setText(this.toString());

17 Button button = (Button) findViewById(R.id.button);

18 button.setOnClickListener(new View.OnClickListener() {

19 @Override

20 public void onClick(View v) {

21 Intent intent = new Intent(FirstActivity.this, FirstActivity.class);

22 startActivity(intent);

23 }

24 });

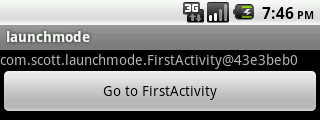
25 }

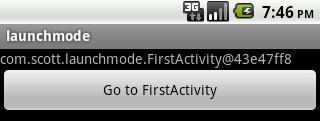
26 }

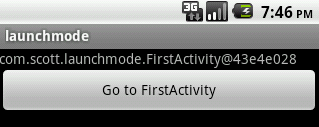
[复制代码](javascript:void(0);)

我们FirstActivity界面中的TextView用于显示当前Activity实例的序列号，Button用于跳转到下一个FirstActivity界面。

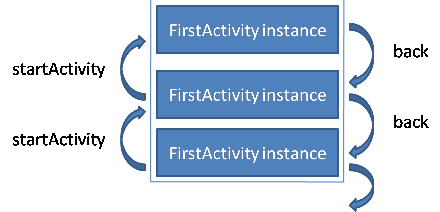
然后我们连续点击几次按钮，将会出现下面的现象：







我们注意到都是FirstActivity的实例，但序列号不同，并且我们需要连续按后退键两次，才能回到第一个FristActivity。standard模式的原理如下图所示：



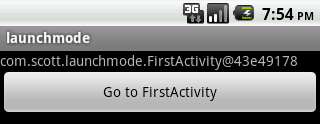
如图所示，每次跳转系统都会在task中生成一个新的FirstActivity实例，并且放于栈结构的顶部，当我们按下后退键时，才能看到原来的FirstActivity实例。

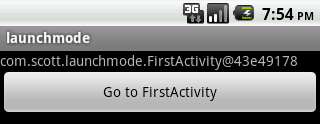
这就是standard启动模式，不管有没有已存在的实例，都生成新的实例。

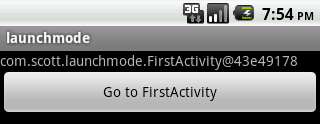
**简单点理解：standard启动模式Activity栈从栈底到栈顶顺序为A1 -> B -> C -> A2...。（其中A、B、C等都表示不同的Activity实例，A1、A2则表示属于具有同一Activity类的不同实例）**

**2.singleTop**

我们在上面的基础上为<activity>指定属性android:launchMode="singleTop"，系统就会按照singleTop启动模式处理跳转行为。我们重复上面几个动作，将会出现下面的现象：







我们看到这个结果跟standard有所不同，三个序列号是相同的，也就是说使用的都是同一个FirstActivity实例；如果按一下后退键，程序立即退出，说明当前栈结构中只有一个Activity实例。singleTop模式的原理如下图所示：



正如上图所示，跳转时系统会先在栈结构中寻找是否有一个FirstActivity实例正位于栈顶，如果有则不再生成新的，而是直接使用。也许朋友们会有疑问，我只看到栈内只有一个Activity，如果是多个Activity怎么办，如果不是在栈顶会如何？我们接下来再通过一个示例来证实一下大家的疑问。

我们再新建一个Activity命名为SecondActivity，如下：

[复制代码](javascript:void(0);)

1 package com.scott.launchmode;

2

3 import android.app.Activity;

4 import android.content.Intent;

5 import android.os.Bundle;

6 import android.view.View;

7 import android.widget.Button;

8 import android.widget.TextView;

9

10 public class SecondActivity extends Activity {

11 @Override

12 protected void onCreate(Bundle savedInstanceState) {

13 super.onCreate(savedInstanceState);

14 setContentView(R.layout.second);

15 TextView textView = (TextView) findViewById(R.id.textView);

16 textView.setText(this.toString());

17 Button button = (Button) findViewById(R.id.button);

18 button.setOnClickListener(new View.OnClickListener() {

19 @Override

20 public void onClick(View v) {

21 Intent intent = new Intent(SecondActivity.this, FirstActivity.class);

22 startActivity(intent);

23 }

24 });

25 }

26 }

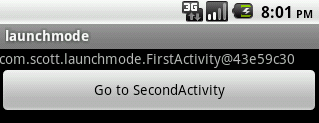
[复制代码](javascript:void(0);)

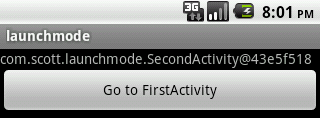
然后将之前的FirstActivity跳转代码改为：

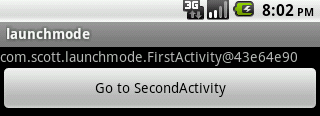
1 Intent intent = new Intent(FirstActivity.this, SecondActivity.class);

2 startActivity(intent);

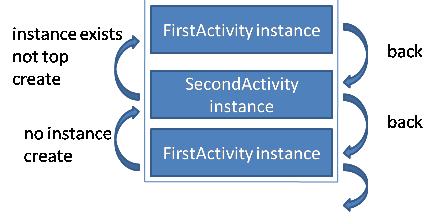
是的，FirstActivity会跳转到SecondActivity，SecondActivity又会跳转到FirstActivity。演示结果如下：







我们看到，两个FirstActivity的序列号是不同的，证明从SecondActivity跳转到FirstActivity时生成了新的FirstActivity实例。原理图如下：



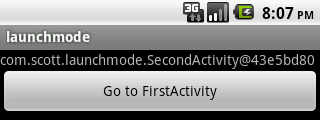
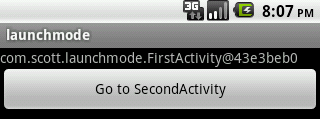
我们看到，当从SecondActivity跳转到FirstActivity时，系统发现存在有FirstActivity实例,但不是位于栈顶，于是重新生成一个实例。

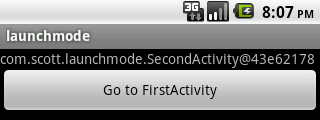
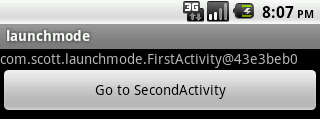
**这就是singleTop启动模式，如果发现有对应的Activity实例正位于栈顶，则重复利用，不再生成新的实例。**

**简单点理解，singleTop即表示当前Activity栈中“栈顶唯一”，Activity跳转顺序或standard模式下栈结构如果为：A -> B -> C -> D1 -> D2，则singleTop启动模式为：A -> B -> C -> D1(此时回调D1的onNewIntent()..)。**

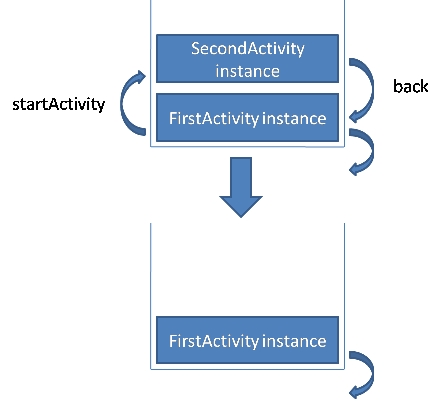
**3.singleTask**

在上面的基础上我们修改FirstActivity的属性android:launchMode="singleTask"。演示的结果如下：





我们注意到，在上面的过程中，FirstActivity的序列号是不变的，SecondActivity的序列号却不是唯一的，说明从SecondActivity跳转到FirstActivity时，没有生成新的实例，但是从FirstActivity跳转到SecondActivity时生成了新的实例。singleTask模式的原理图如下图所示：



在图中的下半部分是SecondActivity跳转到FirstActivity后的栈结构变化的结果，我们注意到，SecondActivity消失了，没错，在这个跳转过程中系统发现有存在的FirstActivity实例，于是不再生成新的实例，而是将FirstActivity之上的Activity实例统统出栈，将FirstActivity变为栈顶对象，显示到幕前。也许朋友们有疑问，如果将SecondActivity也设置为singleTask模式，那么SecondActivity实例是不是可以唯一呢？在我们这个示例中是不可能的，因为每次从SecondActivity跳转到FirstActivity时，SecondActivity实例都被迫出栈，下次等FirstActivity跳转到SecondActivity时，找不到存在的SecondActivity实例，于是必须生成新的实例。但是如果我们有ThirdActivity，让SecondActivity和ThirdActivity互相跳转，那么SecondActivity实例就可以保证唯一。

**这就是singleTask模式，如果发现所在Activity栈中有对应的Activity实例，则使此Activity实例之上的其他Activity实例统统出栈，使此Activity实例成为栈顶对象，显示到幕前。**

**简单点理解，singleTask表示当前Activity栈中“实例唯一”，Activity跳转顺序或standard模式下栈结构如果为：A -> B1 -> C -> D -> B2，则singleTask启动模式为：A -> B1(此时回调onNewIntent()..)**

**4.singleInstance**

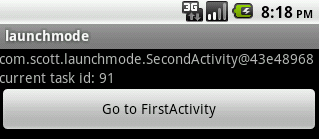
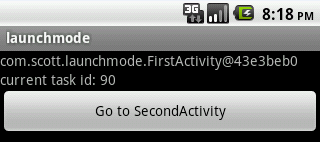
这种启动模式比较特殊，因为它会启用一个新的栈结构，将Acitvity放置于这个新的栈结构中，并保证不再有其他Activity实例进入。

我们修改FirstActivity的launchMode="standard"，SecondActivity的launchMode="singleInstance"，由于涉及到了多个栈结构，我们需要在每个Activity中显示当前栈结构的id，所以我们为每个Activity添加如下代码：

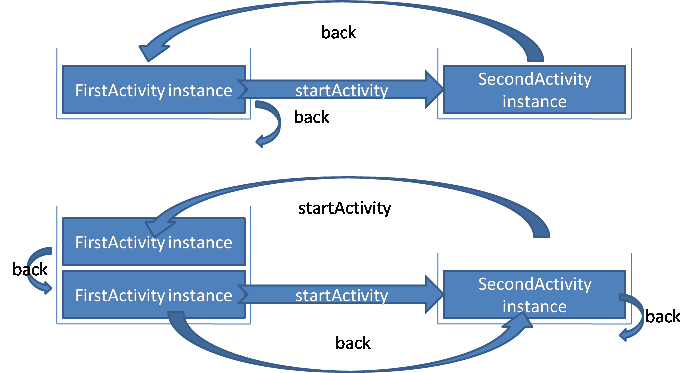
1 TextView taskIdView = (TextView) findViewById(R.id.taskIdView);

2 taskIdView.setText("current task id: " + this.getTaskId());

然后我们再演示一下这个流程：



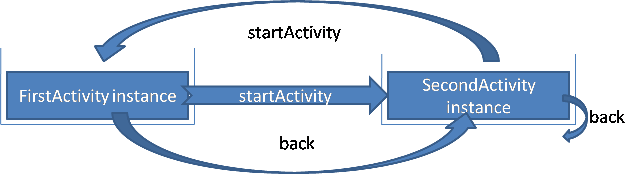
我们发现这两个Activity实例分别被放置在不同的栈结构中，关于singleInstance的原理图如下：



我们看到从FirstActivity跳转到SecondActivity时，重新启用了一个新的栈结构，来放置SecondActivity实例，然后按下后退键，再次回到原始栈结构；图中下半部分显示的在SecondActivity中再次跳转到FirstActivity，这个时候系统会在原始栈结构中生成一个FirstActivity实例，然后回退两次，注意，并没有退出，而是回到了SecondActivity，为什么呢？是因为从SecondActivity跳转到FirstActivity的时候，我们的起点变成了SecondActivity实例所在的栈结构，这样一来，我们需要“回归”到这个栈结构。

此处的解释不是很赞同，第一次按Back键首先是在当前Activity栈中将栈顶元素出栈，然后显示当前Activity栈中下一个Activity栈，这个没什么解释的，然后按下Back键，不是回到手机桌面，而是回到另一个Activity栈中的SecondActivityInstance，我认为原因在于在于“最近栈”，只要此栈位于上次Home操作之后，就会先显示它。

如果我们修改FirstActivity的launchMode值为singleTop、singleTask、singleInstance中的任意一个，流程将会如图所示：



singleInstance启动模式可能是最复杂的一种模式，为了帮助大家理解，我举一个例子，假如我们有一个share应用，其中的ShareActivity是入口Activity，也是可供其他应用调用的Activity，我们把这个Activity的启动模式设置为singleInstance，然后在其他应用中调用。我们编辑ShareActivity的配置：

[复制代码](javascript:void(0);)

<activity android:name=".ShareActivity" android:launchMode="singleInstance">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

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

</intent-filter>

<intent-filter>

<action android:name="android.intent.action.SINGLE\_INSTANCE\_SHARE" />

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

</intent-filter>

</activity>

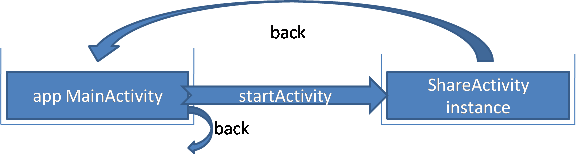
[复制代码](javascript:void(0);)

然后我们在其他应用中这样启动该Activity：

1 Intent intent = new Intent("android.intent.action.SINGLE\_INSTANCE\_SHARE");

2 startActivity(intent);

当我们打开ShareActivity后再按后退键回到原来界面时，ShareActivity做为一个独立的个体存在，如果这时我们打开share应用，无需创建新的ShareActivity实例即可看到结果，因为系统会自动查找，存在则直接利用。大家可以在ShareActivity中打印一下taskId，看看效果。关于这个过程，原理图如下：



原作者此处的解释可能有点让人误解。当我们打开ShareActivity后再按后退键回到原来界面时，ShareActivity做为一个独立的个体存在，此处不应该是按Back键，而是Home键，因为一旦按下了Back键，ShrareActivityInstance自然就销毁了，也就不存在所谓的“无须重新创建了”。按下Home键后，接下来打开app MainActivity，在另一个Activity栈中app MainActivity入栈，此时startActivity到ShareActivity，无需创建新的ShareActivity实例即可看到结果，因为系统会自动查找，存在则直接利用。此时第一次按下Back，ShareActivity Instance出栈，此时这个栈中没有其他Activity了，自然是回到了app MainActivity所在的栈并显示app MainActivity，接下来按Back键，此时app MainActivity所在的栈也没有其他Activity了，同时又不包含任何其他的“最近栈”，自然是回到了手机桌面。注：此处理解的“最近栈”是以Home键或桌面状态为间隔区分。

**简单点理解，singleInstance所标识的Activity，当被启动时，系统会首先判断系统其他栈中是否已经存在此Activity实例，有则直接使用，并且其所在的Activity栈理论上只有它一个Activity元素。所以启动它的Activity与它并不在一个task中，所以才需要特别注意Back的问题。一般表示为：task1 A -> task2 B。**

**singleInstance表示该Activity在系统范围内“实例唯一”。由此我们发现，singInstance和singleTask主要区别在与系统范围内的“实例唯一”还是当前Activity栈“实例唯一”。**

---------------------------------------------------------------------------------   
笔者水平有限，若有错漏，欢迎指正，如果转载以及CV操作，请务必注明出处，谢谢！

#### [android – 如何读取“adb shell dumpsys alarm”输出](https://codeday.me/bug/20171005/79983.html)

时间：2017-10-05 标签：[adb](https://codeday.me/tag/adb)[alarmmanager](https://codeday.me/tag/alarmmanager)[android](https://codeday.me/tag/android)

我正在努力正确设置闹钟，并了解取消和重新安排闹钟的机制。

我发现，有一个adb命令来检索设备上安排的所有警报，但我还没有找到一个文档，解释输出的格式。

我明白，我在这里要求很多解释，所以如果有人会抛出一个链接与详细的解释关于“adb shell dumpsys报警”，我会真的很感激。

所以，这里有问题：

>

Pending alarm batches: 23

一个。 ’23’是当前活动的多个预定报警吗？  
>

Batch{4293d3a8 num=1 start=1369361 end=1407261}:  
  
  RTC #0: Alarm{4293d358 type 1 com.android.chrome}  
  
    type=1 whenElapsed=1369361 when=+19s304ms window=-1 repeatInterval=0 count=0  
  
    operation=PendingIntent{429e4500: PendingIntentRecord{429dbbc8 com.android.chrome broadcastIntent}}

一个。什么是“num = 1”，“start = 1369361”和“end = 1407261”？  
b。 ‘RTC’代表RTC报警，我假设。  
C。什么’#0’代表？  
d。什么意思’type = 1’？  
e。是’when = 19s304ms’意味着警报将在19秒内触发？  
F。什么意思是’window = -1’？  
G。是’repeatInterval = 0’这意味着这是不重复的报警？  
H。是’count = 0’意味着这个警报没有推迟，由于手机睡眠状态？  
一世。 ‘operation = PendingIntent {…}’表示待处理的意图，将由警报触发，我假设。  
>

Broadcast ref count: 0

一个。这是什么？  
>

Top Alarms:

一个。这是什么？  
>

+47s271ms running, 0 wakeups, 2 alarms: com.username.weatherinfo  
  
  act=com.username.receivers.CyclicWeatherUpdater.WEATHER\_UPDATE\_ACTION  
  
    cmp={com.username.weatherinfo/com.username.receivers.CyclicWeatherUpdater}

一个。是’47s271ms’意思是这个警报将在47秒内触发？  
b。什么是’0 wakeups’ – 警报从未被触发？  
C。什么是“2报警”？  
d。是“com.username.weatherinfo”代表程序包的名称，在上下文字段中被赋予pending intent吗？  
e。 “行为”是指为了意图而发出的行动？  
F。什么是“cmp”？我看到，它是从包名称和类名称组成 – 但从他们被带走？从意图构造函数？  
G。为什么部分报警只有“行为”或只有“cmp”？我假设，没有’cmp’字段的报警是隐式广播意图。然而，为什么没有“行动”字段的报警？  
>

Alarm Stats:

一个。这是什么？

我意识到这个线程是老的，但答案不容易找到，并可以使用。我花了很多时间来研究这些消息的含义。

Q1：批量

Pending alarm batches: 23

警报按批次组织。 [As described in the documentation](http://developer.android.com/reference/android/app/AlarmManager.html#set(int,%20long,%20android.app.PendingIntent))：

Beginning in API 19, the trigger time passed to this method is treated as inexact: the alarm will not be delivered before this time, but may be deferred and delivered some time later. The OS will use this policy in order to **“batch”** alarms together across the entire system, minimizing the number of times the device needs to “wake up” and minimizing battery use. In general, alarms scheduled in the near future will not be deferred as long as alarms scheduled far in the future.

每个批次可能有多个警报。在这种情况下，有23批次的报警，这意味着可能有超过23个报警计划。在dumpsys报警输出中，描述每个批次的行如下所示：

Batch{4293d3a8 num=1 start=1369361 end=1407261}:

其中：

> 4293d3a8是与批处理相关联的内部标识。  
> num = 1是此批处理中的报警数。在这种情况下，批处理中只有一个警报。  
>开始和结束数字表示自从系统最后重新启动以来已过去的毫秒数为[described in this post](https://stackoverflow.com/a/15400014/296725)，并且还粗略地表示应当触发批量中的警报的时间窗口。

Q2：报警

每个报警由三条线描述，如下所示：

RTC #0: Alarm{4293d358 type 1 com.android.chrome}

type=1 whenElapsed=1369361 when=+19s304ms window=-1 repeatInterval=0 count=0

operation=PendingIntent{429e4500: PendingIntentRecord{429dbbc8 com.android.chrome broadcastIntent}}

其中：

>第一部分，即RTC\_WAKEUP，RTC，ELAPSED\_WAKEUP或ELAPSED中的一个，表示报警的类型，分别对应于整数值0-3  
>#0是批次中的报警编号，其中数字从0到n-1，其中n是批次中的报警数。如果您的报警与其他人分开，未来最远的“when =”定义将触发批次中的所有报警的时间。  
> 4293d358是与警报关联的内部标识号  
> com.android.chrome是设置警报的类的包名称  
> type = 1，报警类型，见上面第一个项目符号  
> whenElapsed = 1369361指自系统启动以来将触发此警报的毫秒数(大约)  
> when = 19s304ms表示报警将在调用dumpsys报警时的19秒，即304毫秒内触发。同样，像2d13h29m03s882ms这样的值指的是未来的相对时间2天，13小时，29分钟…  
> window =指与报警批处理的方法有关的两个内部常量之一。 AlarmManager.WINDOW\_EXACT = 0，并在使用setExact()或setAlarmClock()计划报警时设置。 AlarmManager.WINDOW\_HEURISTIC = -1，在使用setInexactRepeating()计划报警时设置。否则，该值由API版本确定。对于API< 19(KitKat)，使用WINDOW\_EXACT，对于API> = 19，使用WINDOW\_HEURISTIC。 (我不得不[dig into the AlarmManager.java source code](https://github.com/android/platform_frameworks_base/blob/master/core/java/android/app/AlarmManager.java)来解决这个问题。)  
> repeatInterval = 900000是警报重复的频率，例如。每900000ms或15分钟。值为0表示报警不重复。  
> count =指警报应该被触发，但不是由于某种原因的次数。 0是一个很好的数字。 > 0表示由于某种原因跳过了闹钟。  
> operation = PendingIntent {…}是对由警报触发的[PendingIntent](http://developer.android.com/reference/android/app/PendingIntent.html)的引用。根据是否使用getService，getBroadcast，getActivity或getActivities实例化PendingIntent，警报将启动服务，发送广播或启动一个或多个活动。

Q3：广播参考计数

要找出这个和其他输出项目后，我不得不[dig into the AlarmManagerService.java source code](https://github.com/android/platform_frameworks_base/blob/master/services/core/java/com/android/server/AlarmManagerService.java)。

为了使一些警报工作，设备必须被唤醒，并且在所有必要的广播被发送之前不应该回到睡眠。内部变量mBroadcastRefCount初始化为0，并且随着要发送的广播排队而递增。当每个广播被发送时，它被递减，并且当它返回到0时，wakeLock被释放，并且设备可以回到睡眠状态。

广播参考计数：0仅仅意味着在dumpsys警报运行时，它不是在发送任何广播的中间。

Q4：顶部警报

这是按报警代码已运行的总聚合时间降序排列的前十个报警。这可以用于找到消耗最大量的系统资源的警报，例如。找到可能出现故障以排除电池寿命的过程。

Q5：报警统计

此部分显示自上次重新启动系统以来运行的所有警报的统计信息。在这里，您可以查看您在过去设置的警报是否已触发，如果他们唤醒了手机等。这些条目的格式是覆盖下面。

Q6：报警统计输入

报警统计信息条目如下所示：

com.example.someapp +1s857ms running, 0 wakeups:

+1s817ms 0 wakes 83 alarms: cmp={com.example.someapp/com.example.someapp.someservice}

+40ms 0 wakes 1 alarms: cmp={com.example.someapp/com.example.someapp.someotherservice}

其中在第一行：

> com.example.someapp是触发警报的进程的包名称  
> 1s857ms running是进程消耗的总系统时间  
> 0 wakeups是设备被这些警报之一唤醒的次数

然后每行之后引用设置的警报之一，其中：

> 1s817ms是系统消耗的总时间  
> 0 wakes是设备必须被唤醒的次数  
> 83报警是报警触发的次数;对于重复报警，这将仅为> 1  
> cmp = {…}触发警报时启动的服务

或者，如果警报触发广播，则条目可能如下：

android +4m51s566ms running, 281 wakeups:

+2m46s583ms 0 wakes 1224 alarms: act=android.intent.action.TIME\_TICK

+1m25s624ms 89 wakes 89 alarms: act=android.content.syncmanager.SYNC\_ALARM

+52s898ms 0 wakes 41 alarms: act=com.android.server.action.NETWORK\_STATS\_POLL

...

具有：

> act = …是被广播的意图的名称

警报可能具有cmp = {…}和act = …条目，这意味着警报既广播意图并启动服务。

概要

调试android报警使用adb shell dumpsys报警的输出可能是棘手的，没有中心位置，dumpsys消息完全解释。并不总是明显的警报如何批处理在一起，有时很难得到一个服务或活动被触发完全在需要的时候。希望这将是人们试图调试他们的报警的有用的参考。

# 咦，Oreo怎么收不到广播了？

忙啊～最近好忙呀。

忙的我连**SystemUI系列**的文章推进向蜗牛一样慢～

这篇文章偷个闲，记录下Android8.0上的广播限制。

最近在基于Android 8.1的系统项目中有用到静态注册广播去监听广播。可是不论我是普通的将Apk install进去抑或是高贵的push到对应的system/priv-app/目录下，都收不到这个广播。心态，DUANG，炸了。

后来灵光一闪，扒出记忆角落的Android7.0的广播限制，赶紧Google一下。原来如此，恍然大悟：\*\*Android8.0后，当App targetSDK >= 26，几乎禁止了所有的隐式广播的静态注册监听。\*\*特在此记录，防止我以后又提莫的忘记了。

本篇文章主要讲述以下内容，还请拿起小板凳，带好零食，前来观赏：

* Android广播科普
* Android8.0的后台限制
* 具体广播限制和对应赦免清单
* 适配/解决方法

#### 科普科普广播知识

来来来，先科普下，广播两种监听/接收注册方式和两种类型，拿小本本记下来，记住了！

**注册方式**：

* 静态注册：也称为清单注册，就是在AndroidManifest.xml中注册的广播。此类广播接收器在**应用尚未启动**的时候就可以接收到相应广播。
* 动态注册：也称为运行时注册，也就是在Service或者Activity组件中，通过Context.registerReceiver()注册广播接收器。此类广播接收器是在应用已启动后，通过代码进行注册。

**两种类型**：

* 显式广播（Explicit Broadcast）：发送的Intent是显示Intent的广播。通过指定Intent组件名称来实现的，它一般用在知道目标组件名称的前提下，去调用以下方法。意图明确，指定了要激活的组件是哪个组件，一般是在相同的应用程序内部实现的。

Intent.setComponent()

Intent.setClassName()

Intent.setClass()

**new** Intent(A.**this**,B.class)

复制代码

* 隐式广播（Implicit Broadcast）：通过Intent Filter来实现的，它一般用在没有明确指出目标组件名称的前提下。Android系统会根据隐式意图中设置的动作(action)、类别(category)、数据（URI和数据类型）找到最合适的组件来处理这个意图。一般是用于在不同应用程序之间。

#### Android8.0的后台执行限制

注意是针对targetSDK >= 26的应用，也就是说，targetSDK小于26的话，暂不受影响

在Oreo中，为了进一步提升用户体验，进一步节省功耗，对应用在后台运行时可以执行的操作又进一步施加了限制。

* **后台服务限制**：处于空闲状态时，限制应用的后台服务。例如：通过静态注册接收开机广播（假设你的设备没做定制，能收到~），并在onReceive方法中启动一个Service，在API 26上，是不允许且会报错的。当然，对于前台服务，这种限制是不存在的。官方说法是：前台服务更容易引起用户注意。
* **广播限制**：除了有限的例外之外，应用无法使用清单注册（**静态注册**）的方式来接收**隐式广播**。
  + 但对于这些隐式广播，可以通过运行时注册（**动态注册**）的方式注册。
  + 对于**显式广播**，则依然可以通过清单注册（静态注册）的方式监听

这里多说一句，Android手机的卡顿，很大程度是由于应用滥用且自私的使用各种手段（权限滥用，广播注册，后台服务常驻等）保活或做一些PY事情。Google显然很早就意识到这一点，并从Android 6.0 开始就逐步引入各种限制，比如运行时权限和Doze。

#### 具体广播限制和对应赦免清单

如果应用注册了广播接收器，那么每次发送广播后，应用的广播接收器就会消耗资源，如RAM，CPU等。如果有很多应用对系统事件广播注册广播接收器，这....，就会很卡的嘛！

所以从Android 7.0 （API 级别 24）开始，就对广播做了一些限制：

* API24及以上应用，静态注册的广播接收器无法监听网络变化：android.net.conn.CONNECTIVITY\_CHANGE
* 在Android7.0设备上，App无法发送或者接收ACTION\_NEW\_PICTURE和ACTION\_NEW\_VIDEO广播。

只不过，在Android8.0上，又进一步的增强了限制，**除了以下隐式广播外**，其他所有隐式广播均无法通过在AndroidManifest.xml中注册监听。参考[官网](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.android.com%2Fguide%2Fcomponents%2Fbroadcast-exceptions)。

// Android 8.0 上不限制的隐式广播

/\*\*

开机广播

Intent.ACTION\_LOCKED\_BOOT\_COMPLETED

Intent.ACTION\_BOOT\_COMPLETED

\*/

"保留原因：这些广播只在首次启动时发送一次，并且许多应用都需要接收此广播以便进行作业、闹铃等事项的安排。"

/\*\*

增删用户

Intent.ACTION\_USER\_INITIALIZE

"android.intent.action.USER\_ADDED"

"android.intent.action.USER\_REMOVED"

\*/

"保留原因：这些广播只有拥有特定系统权限的app才能监听，因此大多数正常应用都无法接收它们。"

/\*\*

时区、ALARM变化

"android.intent.action.TIME\_SET"

Intent.ACTION\_TIMEZONE\_CHANGED

AlarmManager.ACTION\_NEXT\_ALARM\_CLOCK\_CHANGED

\*/

"保留原因：时钟应用可能需要接收这些广播，以便在时间或时区变化时更新闹铃"

/\*\*

语言区域变化

Intent.ACTION\_LOCALE\_CHANGED

\*/

"保留原因：只在语言区域发生变化时发送，并不频繁。 应用可能需要在语言区域发生变化时更新其数据。"

/\*\*

Usb相关

UsbManager.ACTION\_USB\_ACCESSORY\_ATTACHED

UsbManager.ACTION\_USB\_ACCESSORY\_DETACHED

UsbManager.ACTION\_USB\_DEVICE\_ATTACHED

UsbManager.ACTION\_USB\_DEVICE\_DETACHED

\*/

"保留原因：如果应用需要了解这些 USB 相关事件的信息，目前尚未找到能够替代注册广播的可行方案"

/\*\*

蓝牙状态相关

BluetoothHeadset.ACTION\_CONNECTION\_STATE\_CHANGED

BluetoothA2dp.ACTION\_CONNECTION\_STATE\_CHANGED

BluetoothDevice.ACTION\_ACL\_CONNECTED

BluetoothDevice.ACTION\_ACL\_DISCONNECTED

\*/

"保留原因：应用接收这些蓝牙事件的广播时不太可能会影响用户体验"

/\*\*

Telephony相关

CarrierConfigManager.ACTION\_CARRIER\_CONFIG\_CHANGED

TelephonyIntents.ACTION\_\*\_SUBSCRIPTION\_CHANGED

TelephonyIntents.SECRET\_CODE\_ACTION

TelephonyManager.ACTION\_PHONE\_STATE\_CHANGED

TelecomManager.ACTION\_PHONE\_ACCOUNT\_REGISTERED

TelecomManager.ACTION\_PHONE\_ACCOUNT\_UNREGISTERED

\*/

"保留原因：设备制造商 (OEM) 电话应用可能需要接收这些广播"

/\*\*

账号相关

AccountManager.LOGIN\_ACCOUNTS\_CHANGED\_ACTION

\*/

"保留原因：一些应用需要了解登录帐号的变化，以便为新帐号和变化的帐号设置计划操作"

/\*\*

应用数据清除

Intent.ACTION\_PACKAGE\_DATA\_CLEARED

\*/

"保留原因：只在用户显式地从 Settings 清除其数据时发送，因此广播接收器不太可能严重影响用户体验"

/\*\*

软件包被移除

Intent.ACTION\_PACKAGE\_FULLY\_REMOVED

\*/

"保留原因：一些应用可能需要在另一软件包被移除时更新其存储的数据；对于这些应用，尚未找到能够替代注册此广播的可行方案"

/\*\*

外拨电话

Intent.ACTION\_NEW\_OUTGOING\_CALL

\*/

"保留原因：执行操作来响应用户打电话行为的应用需要接收此广播"

/\*\*

当设备所有者被设置、改变或清除时发出

DevicePolicyManager.ACTION\_DEVICE\_OWNER\_CHANGED

\*/

"保留原因：此广播发送得不是很频繁；一些应用需要接收它，以便知晓设备的安全状态发生了变化"

/\*\*

日历相关

CalendarContract.ACTION\_EVENT\_REMINDER

\*/

"保留原因：由日历provider发送，用于向日历应用发布事件提醒。因为日历provider不清楚日历应用是什么，所以此广播必须是隐式广播。"

/\*\*

安装或移除存储相关广播

Intent.ACTION\_MEDIA\_MOUNTED

Intent.ACTION\_MEDIA\_CHECKING

Intent.ACTION\_MEDIA\_EJECT

Intent.ACTION\_MEDIA\_UNMOUNTED

Intent.ACTION\_MEDIA\_UNMOUNTABLE

Intent.ACTION\_MEDIA\_REMOVED

Intent.ACTION\_MEDIA\_BAD\_REMOVAL

\*/

"保留原因：这些广播是作为用户与设备进行物理交互的结果：安装或移除存储卷或当启动初始化时（当可用卷被装载）的一部分发送的，因此它们不是很常见，并且通常是在用户的掌控下"

/\*\*

短信、WAP PUSH相关

Telephony.Sms.Intents.SMS\_RECEIVED\_ACTION

Telephony.Sms.Intents.WAP\_PUSH\_RECEIVED\_ACTION

注意：需要申请以下权限才可以接收

"android.permission.RECEIVE\_SMS"

"android.permission.RECEIVE\_WAP\_PUSH"

\*/

"保留原因：SMS短信应用需要接收这些广播"

复制代码

呼，终于列完了，以上。可以说写的比官网还全~

建议**收藏一波**防止以后用的到哈。

#### 解决方法

按照官方推荐，对于隐式广播，通过以下方法进行替换。

* 动态通过调用 Context.registerReceiver()注册广播接收器而不是在清单中声明接收器。
* 使用[JobScheduler](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.android.com%2Freference%2Fandroid%2Fapp%2Fjob%2FJobScheduler.html)。

我选择动态注册的方式来处理这个问题。

好了，关于Oreo的广播限制的唠嗑就先唠到这里。

最近受到一位小伙伴的启发，得到一句话：努力的人，运气和机遇往往都不会差！

共勉！

最后，欢迎关注微信公众号：猿湿Xoong  
开心的进行Android高质量干货分享

#### 参考链接

[1] Android Oreo 后台执行限制 https://developer.android.com/about/versions/oreo/background#broadcasts

[2] Android Oreo Implicit Broadcast Exceptions https://developer.android.com/guide/components/broadcast-exceptions

[3] Android中显式和隐式intent的特点和区别 https://blog.csdn.net/u014177843/article/details/50596863

[4] Android O行为变更--隐式广播限制 https://blog.csdn.net/hqocshheqing/article/details/76850164