# Android O 适配

# 一.前言

前一段时间，做了android Oro的适配，记录一下。

使用黑科技和第三方库，在适配的时候，都会引起很多麻烦，以前很多安全问题，都是直接忽略掉了，现在会出现crash。

## 1.1 ContentProvider

监听ContentProvider改变，如果ContentProvider没有在xml中注册，就抛出异常。

如果使用黑科技，用反射的方式，在代码动态注册ContentProvider，在Android oro以下，是可以使用的，在Android Oro以上会crash。

## 1.2 service

后台使用startService(),会导致进程crash，适配的时候要检查自己service不是一件麻烦的事情，但是第三方库的service，不一定可以保证。

如果是google的库，一般更新到新的版本，就可以了。

如果是比较冷门的第三方库，作者又放弃维护了，那这个库就只能放弃使用了。

如果使用了乱七八糟的黑科技，比如：1.反翻译google的包，做了修改；2.使用反射调用了混淆过的方法。那更新新版本库的时候，就会碰到各种各样的坑。

## 1.3 尽量在规则下干活

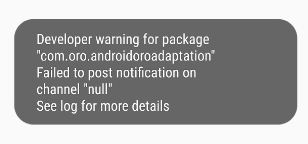
综上，开发的时候尽量遵从google的开发规则，一般情况下，google提供的api，是可以满足需求的，不到万不得已，不要使用黑科技来完成需求。如果你觉得需求本身就不合理的，那么上述提到的点，都要加到需求评审里面，看看需求是否可行。

# 二.通知栏

## 2.1缺少channel的反应

Android O的通知栏需要加上channel，如果没有channel，不会显示通知栏。

如果按照之前的方式弹出通知栏，在android oro的机器上，会出有如下的toast提示。



## 2.2老的通知栏方法

*/\*\*  
 \* 获取没有渠道的通知，android oro 以下，才可以使用  
 \** ***@param*** *c  
 \** ***@return*** *\*/*public static Notification getNotifactionWithoutChannel(Context c){  
 Notification.Builder b = new Notification.Builder(c)  
 .setContentTitle("New Message")  
 .setSmallIcon(R.drawable.*ic\_launcher\_background*)  
 .setContentText("You've received new messages,without channel");  
 Notification notification = b.build();  
 return notification;  
}

## 2.3 Android oro通知栏弹窗

1初始化一个channel对象

2用NotifactionManger，创建一个channel

3使用build，build一个notifation，记得setChannelId，来设置channel

4注意一下适配之前的版本

*/\*\*  
 \* 获取有渠道的通知，android oro 以上以下，都可以使用  
 \** ***@param*** *c  
 \** ***@return*** *\*/*public static Notification getNotifactionWithChannel(Context c){  
 // 通知渠道的id String id = "my\_channel\_01"; // 用户可以看到的通知渠道的名字.  
 Notification.Builder b = new Notification.Builder(c).setSmallIcon(R.drawable.*ic\_launcher\_background*);  
  
 if (android.os.Build.VERSION.*SDK\_INT* >= android.os.Build.VERSION\_CODES.*O*) {  
 //初始化channel的基本参数  
 String name = "notifation\_name";  
 String id = "notifation\_id";  
 int importance = NotificationManager.*IMPORTANCE\_HIGH*;  
 NotificationChannel mChannel = new NotificationChannel(id, name, importance);  
  
 //创建channel  
 NotificationManager mNotificationManager = (NotificationManager) c.getSystemService(Context.*NOTIFICATION\_SERVICE*);  
 mNotificationManager.createNotificationChannel(mChannel);  
  
 //同之前的方式，build  
 b.setContentText("You've received new messages.channel name=" + name )  
 .setContentTitle("AndroidOro New Message")  
 .setChannelId(id);  
 } else{  
 //如果不是android o的机器，修改一下他的文案，这个算是适配的工作  
 b.setContentText("You've received new messages,but Version number below AndroidOro")  
 .setContentTitle("New Message");  
 }  
 Notification notification = b.build();  
 return notification;  
}

## 2.4为啥要有channel

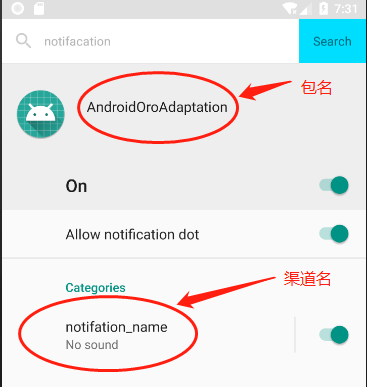
Channel可以方便管理notifacation。

老版本只是根据包名来打开和屏蔽通知。

Andorid Oro以上，可以根据包名和渠道名来管理通知。

如图：

在setting的notifacation中，可以按照包名和渠道名，来管理通知栏。



# 三 Service

核心思想是，不能在后台启动service，如果启动service，需要弹出通知栏，告知用户，哪个应用启动了后台服务。

## 3.1 关于startService

有很多资料说，startService不能使用了，这种说法是错误的。

1.如果程序在前台跑，可以用startService的方式，来启动服务，没有任何问题，只是在程序进入后台1分钟之后，Service会被销毁，走到onDestory。

2.如果程序刚刚从前台，切换到后台（1分钟之内），依然可以用startService来启动服务，并不会crash。

这个场景和场景1相同，也是到1分钟，这个服务就会被销毁，走到onDestory。

3.如果程序在后台很久了（大于1分钟），这时候用startService的方式来启动服务，会发生crash。

4.综上，以Activitiy为例，一个Activity,从启动，到切换到后台1分钟，这个时间段内，都是可以使用startService的。在切换到后台1分钟的这个时间点，会把通过startService启动的服务，都关闭。从这个时间点往后，调用startService，会Crash。

5.图解，看看下图就非常清楚了：



## 3.2 前台启动startService，退出之后在1分钟被销毁

用startService之后，退出Acitivity，可以看到很快就走了onDestory。



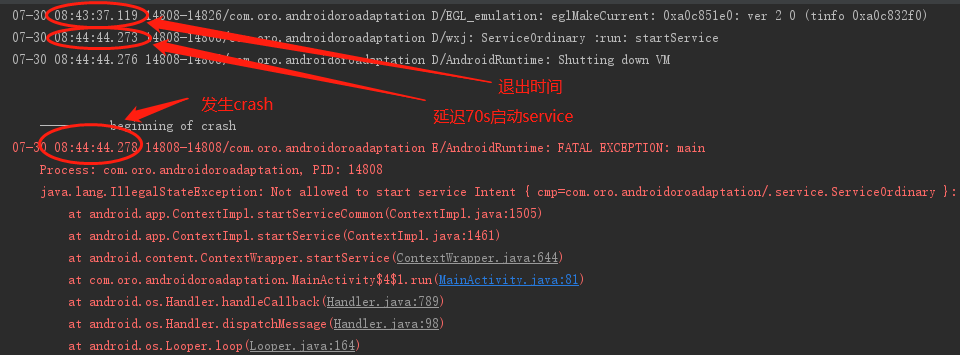
## 3.3 后台启动Service引起Crash

做了一个延迟启动的方法，点击按钮以后，马上退出activity，看看效果。

### 1.代码：

//用startService方式，延迟70秒，启动service  
Button btnStartServiceBackgound = findViewById(R.id.*btn\_start\_service\_backgound*);  
btnStartServiceBackgound.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 new Handler().postDelayed(new Runnable() {  
 @Override  
 public void run() {  
 Log.*d*("wxj", "ServiceOrdinary :run: startService ");  
 Intent i = new Intent();  
 i.setClass(MainActivity.this, ServiceOrdinary.class);  
 startService(i);  
 }  
 },70 \* 1000);  
 }  
});

### 2.log



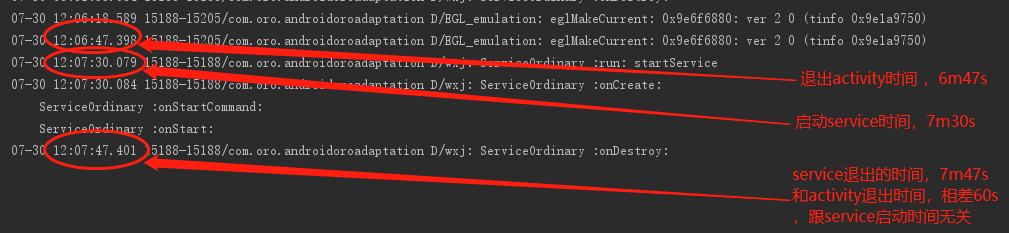
## 3.4后台启动Service不引起Crash

上面是延迟70s，这里只延迟45s

### 1.代码：

//用startService方式，延迟45秒，启动service  
Button btnStartServiceBackgound = findViewById(R.id.*btn\_start\_service\_backgound*);  
btnStartServiceBackgound.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 new Handler().postDelayed(new Runnable() {  
 @Override  
 public void run() {  
 Log.*d*("wxj", "ServiceOrdinary :run: startService ");  
 Intent i = new Intent();  
 i.setClass(MainActivity.this, ServiceOrdinary.class);  
 startService(i);  
 }  
 },45 \* 1000);  
 }  
});

### 2.log



## 3.5启动带通知栏的service

1.使用startForegroundService，来启动service

2.在service的onStartCommend里面，绑定notifacation

//启动service,Oro  
Button btnStartServiceforeGroud = findViewById(R.id.*btn\_start\_service\_fore\_gound*);  
btnStartServiceforeGroud.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 Intent i = new Intent();  
 i.setClass(MainActivity.this, ServiceForAndroidOro.class);  
 if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*O*) {  
 startForegroundService(i);  
 }  
 }  
});

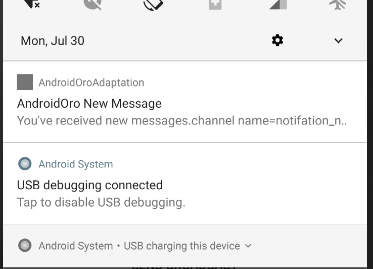
在ServiceForAndroidOro里：

@Override  
public int onStartCommand(Intent intent, int flags, int startId) {  
 Log.*d*("wxj", "ServiceForAndroidOro :onStartCommand: ");  
 //这里调用startForeground，在前台生成一个通知栏，就可以了  
 Notification notification = NotifactionHelper.*getNotifactionWithChannel*(this);  
 startForeground(1,notification);  
 return super.onStartCommand(intent, flags, startId);  
}

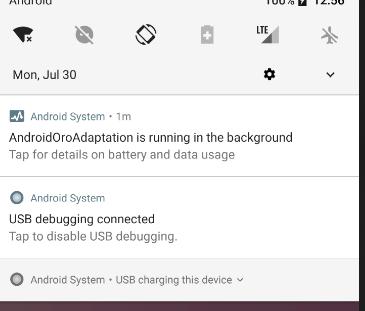
## 3.6特别注意

这样使用以后，Activity就变成前台应用了，会有个通知栏一直保持。

如果再通过startService启动service，不会被自动销毁，也不会引起crash。



删掉这个通知以后，会变成下面的通知。



## 3.7 意义

Service的这项改动，是为了不让程序员，在后台启动Service，避免在用户不知情的情况下，后台一直跑，做了很多工作，导致机器卡顿。

那如果我在Activity里，启动一个线程，那么在Activity销毁之后，线程依然在跑，依然可以做下载，收集数据上报等等工作。这并不能完全避免很多后台一直跑的问题。

那么Google这么做的意义在哪里呢？

### 3.7.1可行性以及合理性

1. Service是android的控件，那么对某一个控件的修改，是完全合理，并且成本很低。
2. 如果把线程的生命周期和某个控件的生命周期绑定，比如activity销毁以后，挂起或者终止它内部的所有线程，这个成本就大了
3. 也很难判断哪些线程属于哪个控件，比如Activity发了个广播，reciver启动了一个线程，这个启动的位置是可以转移的。
4. 如果真的要用线程做正常的功能呢？

### 3.7.2效果

1. 关闭一个渠道，确实可以让很多人，把service放到前台来工作，虽然不能完全避免，但是可以约束大部分的开发者。

2. service作为4大组件，可以静态注册，比如响应开机广播，比线程要重一些，吃的资源，对系统的影响更大一些。

# 四.广播

## 4.1 取消大部分静态注册

reciver在AndroidMainfest里面静态注册，不生效。

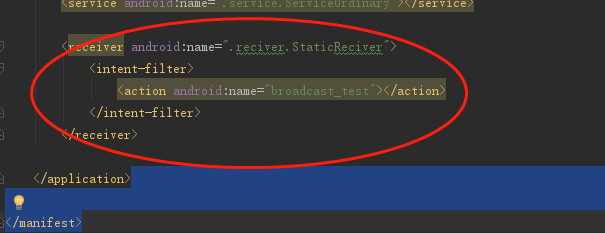
看看例子，同时写两个reciver，一个动态注册，一个静态注册，发送同一个广播，看谁能收到。

### 4.1.1 静态的接收器

这个类很简单：

public class StaticReciver extends BroadcastReceiver{  
 @Override  
 public void onReceive(Context context, Intent intent) {  
 Log.*d*("wxj", "StaticReciver :onReceive: intent.action = "+intent.getAction());  
 }  
}

AndroidManifest中注册：

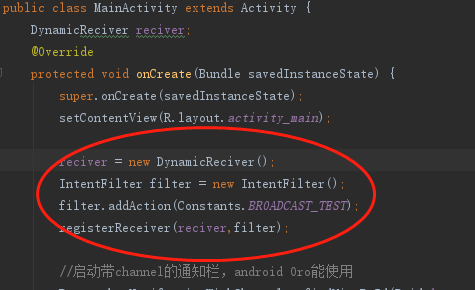


### 4.1.2 动态的接收器

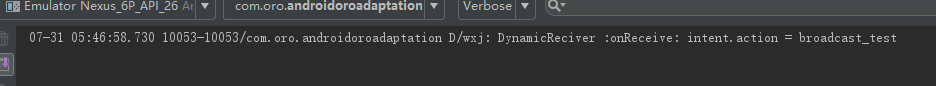
跟静态类一样的，只有logcat不同

public class DynamicReciver extends BroadcastReceiver{  
 @Override  
 public void onReceive(Context context, Intent intent) {  
 Log.*d*("wxj", "DynamicReciver :onReceive: intent.action = "+intent.getAction());  
 }  
}

Activity中注册：



### 4.1.3 只有动态能收到

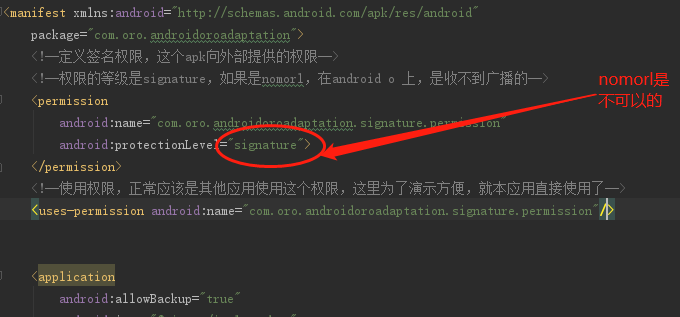


## 4.2 签名权限的应用，可以注册接受广播

这里为了方便测试，提供签名和使用签名，我们用在一个apk里了，正常应该是分开使用两个apk。

注意：要签名权限才可以，普通权限是不可以的。

### 4.2.1 AndroidManifest定义

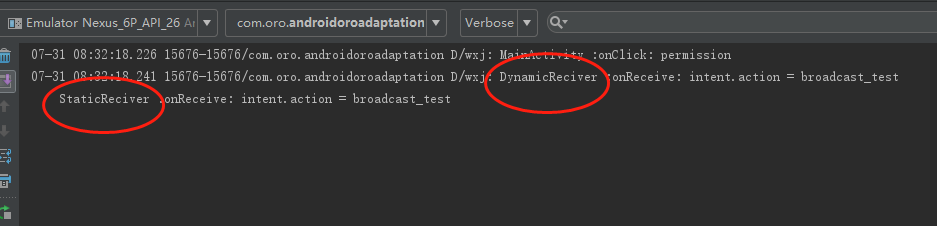


### 4.2.2 发送广播

添加权限就可以了

Button btnStaticBroadcastWithSignature = findViewById(R.id.*btn\_broadcast\_with\_signature*);  
btnStaticBroadcastWithSignature.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 Intent i = new Intent();  
 i.setAction(Constants.*BROADCAST\_TEST*);  
 sendBroadcast(i,"com.oro.androidoroadaptation.signature.permission");  
 }  
});

### 4.2.3 静态和动态都可以收到广播



## 4.3白名单

动态注册的广播，都是在应用被拉起来以后，才有机会注册，而静态广播，应用不用被拉起，也可以响应，所以以下白名单，基本都会和这个特点相关。

豁免广播列表如下:

### 4.3.1 开机

ACTION\_LOCKED\_BOOT\_COMPLETED、ACTION\_BOOT\_COMPLETED  
豁免原因：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。

### 4.3.2 特权保护

ACTION\_USER\_INITIALIZE、"android.intent.action.USER\_ADDED"、"android.intent.action.USER\_REMOVED"  
这些广播受特权保护，因此大多数正常应用无论如何都无法接收它们。

λ "android.intent.action.TIME\_SET"、ACTION\_TIMEZONE\_CHANGED  
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。

### 4.3.3切换语言

ACTION\_LOCALE\_CHANGED  
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。

### 4.3.4 USB

ACTION\_USB\_ACCESSORY\_ATTACHED、ACTION\_USB\_ACCESSORY\_DETACHED、ACTION\_USB\_DEVICE\_ATTACHED、ACTION\_USB\_DEVICE\_DETACHED  
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。

### 4.3.5 物理插拔

ACTION\_HEADSET\_PLUG  
由于此广播只在用户进行插头的物理连接或拔出时发送，因此不太可能会在应用响应此广播时影响用户体验。

### 4.3.6 蓝牙

ACTION\_CONNECTION\_STATE\_CHANGED  
与 ACTION\_HEADSET\_PLUG 类似,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。

### 4.3.7 OEM相关

ACTION\_CARRIER\_CONFIGCHANGED、TelephonyIntents.ACTION\*\_SUBSCRIPTION\_CHANGED、"TelephonyIntents.SECRET\_CODE\_ACTION"  
原始设备制造商(OEM)电话应用可能需要接收这些广播。

### 4.3.8存储相关

ACTION\_DEVICE\_STORAGE\_LOW、ACTION\_DEVICE\_STORAGE\_OK  
一些应用可能需要通过删除数据来响应存储空间不足广播。例如，照片应用可能会删除已备份至云存储空间的旧照片。

### 4.3.9账号变化

LOGIN\_ACCOUNTS\_CHANGED\_ACTION  
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。

### 4.3.10 Setting清除数据

ACTION\_PACKAGE\_DATA\_CLEARED  
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。

### 4.3.11卸载包

ACTION\_PACKAGE\_FULLY\_REMOVED  
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。  
注：其他软件包相关广播（例如：ACTION\_PACKAGE\_REPLACED）不能免受这些新增限制。 这些广播相当常见，豁免它们可能会影响性能。

### 4.3.12电话

ACTION\_NEW\_OUTGOING\_CALL  
执行操作来响应用户打电话行为的应用需要接收此广播。

### 4.3.13安装状态变化

ACTION\_DEVICE\_OWNER\_CHANGED  
此广播发送得不是很频繁；一些应用需要接收它，以便知晓设备的安装状态发生了变化。

### 4.3.14日历提醒

ACTION\_EVENT\_REMINDER

由日历提供程序发送，用于向日历应用发布事件提醒。因为日历提供程序不清楚日历应用是什么，所以此广播必须是隐式广播。

## 五．ShortcutView

这里说的是，普通App往launcher上，添加一个自己的快捷方式，普通App里的做法。

至于launcher有什么改动，这里先不说。

### 5.1 老方法广播

往Intent里，添加图标，标题，跳转

然后发送广播com.android.launcher.action.INSTALL\_SHORTCUT

*/\*\*  
 \* oro以下，添加快捷方式的方法  
 \** ***@param*** *name  
 \*/*private void addShortcut(String name) {  
 Intent addShortcutIntent = new Intent("com.android.launcher.action.INSTALL\_SHORTCUT");  
  
 // 不允许重复创建  
 // 经测试不是根据快捷方式的名字判断重复的  
 // 应该是根据快链的Intent来判断是否重复的,即Intent.EXTRA\_SHORTCUT\_INTENT字段的value  
 // 但是名称不同时，虽然有的手机系统会显示Toast提示重复，仍然会建立快链  
 // 屏幕上没有空间时会提示  
 // 注意：重复创建的行为MIUI和三星手机上不太一样，小米上似乎不能重复创建快捷方式  
 addShortcutIntent.putExtra("duplicate", false);  
  
 // 名字  
 addShortcutIntent.putExtra(Intent.*EXTRA\_SHORTCUT\_NAME*, name);  
  
 // 图标  
 addShortcutIntent.putExtra(Intent.*EXTRA\_SHORTCUT\_ICON\_RESOURCE*,  
 Intent.ShortcutIconResource.*fromContext*(MainActivity.this,  
 R.drawable.*ic\_launcher\_background*));  
  
 // 设置关联程序  
 Intent launcherIntent = new Intent(Intent.*ACTION\_MAIN*);  
 launcherIntent.setClass(MainActivity.this, MainActivity.class);  
 launcherIntent.addCategory(Intent.*CATEGORY\_LAUNCHER*);  
 addShortcutIntent.putExtra(Intent.*EXTRA\_SHORTCUT\_INTENT*, launcherIntent);  
  
 // 发送广播  
 sendBroadcast(addShortcutIntent);  
}

### 5.2 新方法ShortcutManager

往Intent里，添加图标，标题，跳转

然后通过ShortcutManager添加。

*/\*\*  
 \* oro 以上，添加快捷方式的方法  
 \** ***@param*** *name  
 \*/*private void addShortcutOro(String name) {  
 if (android.os.Build.VERSION.*SDK\_INT* >= android.os.Build.VERSION\_CODES.*O*) {  
 ShortcutManager scm = (ShortcutManager) getSystemService(*SHORTCUT\_SERVICE*);  
 Intent launcherIntent = new Intent(Intent.*ACTION\_MAIN*);  
 launcherIntent.setClass(MainActivity.this, MainActivity.class);  
 launcherIntent.addCategory(Intent.*CATEGORY\_LAUNCHER*);  
  
 ShortcutInfo si = new ShortcutInfo.Builder(this, "dataroam")  
 .setIcon(Icon.*createWithResource*(this, R.drawable.*ic\_launcher\_background*))  
 .setShortLabel(name)  
 .setIntent(launcherIntent)  
 .build();  
 scm.requestPinShortcut(si, null);  
 }  
}

调用方法以后，会有如下提示，选择ADD，luancher上会生成shortcut。

