android:的笔记

关于build.gradle(app)的笔记

**build.gradle(Module:app)**

// 声明是Android程序，

//com.android.application 表示这是一个应用程序模块

//com.android.library 标识这是一个库模块

//而这区别：前者可以直接运行，后着是依附别的应用程序运行

apply plugin: 'com.android.application'

android {

// 编译SDK的版本

compileSdkVersion 21

// build tools的版本

buildToolsVersion "21.1.1"

defaultConfig {

// 应用的包名

applicationId "me.storm.ninegag"

//Android支持的最低版本

minSdkVersion 14

targetSdkVersion 21

versionCode 1

versionName "1.0.0"

}

// java版本

compileOptions {

sourceCompatibility JavaVersion.VERSION\_1\_7

targetCompatibility JavaVersion.VERSION\_1\_7

}

//指定生成安装文件的相关配置

buildTypes {

release {

// 是否进行混淆

minifyEnabled false

// 混淆文件的位置

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.txt'

//proguard-android.txt 在Android SDK目录下，里面是所有项目的混淆规则

//proguard-rules.pro 是当前项目的根目录下的，里面可以编写特有的混淆规则

}

debug{ //测试版安装文件的配置（可以忽略）

}

}

// 移除lint检查的error

lintOptions {

abortOnError false

}

}

//依赖关系

//本地依赖：可以对本地Jar包或目录添加依赖关系

//库依赖：可以对项目中的库模块添加依赖关系

//远程以来：可以对 jcenter库上的开源项目添加依赖

//

dependencies {

// 编译libs目录下的所有jar包

compile fileTree(dir: 'libs', include: ['\*.jar'])

//依赖声明，标识将libs目录下多有.jar后缀的文件都添加到项目的构建路径当中

compile 'com.android.support:support-v4:21.0.2'

//标准的远程以来格式 依次是 域名：组织名：版本号

compile 'com.etsy.android.grid:library:1.0.5'

compile 'com.alexvasilkov:foldable-layout:1.0.1'

// 编译extras目录下的ShimmerAndroid模块

compile project(':extras:ShimmerAndroid')

}

关于一些配置参数的解释：

<?xml version="1.0" encoding="utf-8"?>

<set xmlns:android="http://schemas.android.com/apk/res/android">

<translate

android:duration="800"

android:fromYDelta="115%"

android:interpolator="@android:interpolator/accelerate\_cubic"

android:toYDelta="100%"></translate>

<alpha

android:duration="@android:integer/config\_mediumAnimTime"

android:fromAlpha="0.0"

android:toAlpha="1.0" />

</set>

[复制代码](javascript:void(0);)

其中：

    translate 位置转移动画效果    
            整型值:    
                fromXDelta 属性为动画起始时 X坐标上的位置    
                toXDelta   属性为动画结束时 X坐标上的位置    
                fromYDelta 属性为动画起始时 Y坐标上的位置    
                toYDelta   属性为动画结束时 Y坐标上的位置    
                注意:    
                         没有指定，    
                         默认是以自己为相对参照物    
            长整型值：    
                duration  属性为动画持续时间

                说明:   时间以毫秒为单位

在这些属性里面还可以加上%和p,例如:

android:toXDelta="100%",表示自身的100%,也就是从View自己的位置开始。

android:toXDelta="80%p",表示父层View的80%,是以它父层View为参照的。

romAlpha 属性为动画起始时透明度;

toAlpha 属性为动画结束时透明度

其余部分参数参考这个网址：http://www.360doc.com/content/11/0505/12/2113138\_114523286.shtml

关于注解的使用：

**准备工作**

        默认情况下,android中的注解包并没有包括在framework中,它独立成一个单独的包,通常我们需要引入这个包.

[Java] 查看源文件 复制代码

[?](http://www.apkbus.com/)

|  |  |
| --- | --- |
| 1  2  3 | dependencies {    compile 'com.android.support:support-annotations:22.2.0'  } |

  但是如果我们已经引入了appcompat则没有必要再次引用support-annotations,因为appcompat默认包含了对其引用.

使用bindview替代繁琐的findviewbyid和类型转换

使用onclick注解方法来替换显式声明的匿名内部类

使用bindstring,bindbool,binddrawable等注解实现资源获取

ButterKnife是一个专注于Android系统的View注入框架,以前总是要写很多findViewById来找到View对象，有了ButterKnife可以很轻松的省去这些步骤。最重要的一点，使用ButterKnife对性能基本没有损失，因为ButterKnife用到的注解并不是在运行时反射的，而是在编译的时候生成新的class。项目集成起来也是特别方便，使用起来也是特别简单。

**ButterKnife的优势：**

1、强大的View绑定和Click事件处理功能，简化代码，提升开发效率

2、方便的处理Adapter里的ViewHolder绑定问题

3、运行时不会影响APP效率，使用配置方便

4、代码清晰，可读性强

**ButterKnife的注册与绑定**

**▲ ButterKnife使用心得与注意事项：**

1、在Activity 类中绑定 ：ButterKnife.bind(this);必须在setContentView();之后绑定；且父类bind绑定后，子类不需要再bind。

2、在非Activity 类（eg：Fragment、ViewHold）中绑定： ButterKnife.bind(this，view);这里的this不能替换成getActivity（）。

3、在Activity中不需要做解绑操作，在Fragment 中必须在onDestroyView()中做解绑操作。

4、使用ButterKnife修饰的方法和控件，不能用private or static 修饰，否则会报错。错误: @BindView fields must not be private or static. (com.zyj.wifi.ButterknifeActivity.button1)

5、setContentView()不能通过注解实现。（其他的有些注解框架可以）

6、使用Activity为根视图绑定任意对象时，如果你使用类似MVC的设计模式你可以在Activity 调用ButterKnife.bind(this, activity)，来绑定Controller。

7、使用ButterKnife.bind(this，view)绑定一个view的子节点字段。如果你在子View的布局里或者自定义view的构造方法里 使用了inflate,你可以立刻调用此方法。或者，从XML inflate来的自定义view类型可以在onFinishInflate回调方法中使用它。

BufferNife 中的注解使用可以参考这个网站：

https://blog.csdn.net/donkor\_/article/details/77879630

**事件绑定：**

* 绑定点击事件：
  + 绑定控件点击事件：@OnClick( )
  + 绑定控件长按事件：@OnLongClick( )



部分方法的执行顺序：

onResume是在启动activity启动之后才能执行的，也就是恢复执行。程序正常启动：onCreate()->onStart()->onResume();  
正常退出：onPause()->onStop()->onDestory()  
一个Activity启动另一个Activity: onPause()->onStop(), 再返回：onRestart()->onStart()->onResume()  
程序按back 退出： onPause()->onStop()->onDestory(),再进入：onCreate()->onStart()->onResume();  
程序按home 退出： onPause()->onStop(),再进入：onRestart()->onStart()->onResume();

**android studio 的快捷键:**



**补充：Android Activity中的六个主要函数**

Android中一个Activity一般都需要实现六个函数:

onCreate(), onStart(), onResume(),onPause(),onStop(),onDestroy().

1. onCreate函数:注册你要用到的变量，比如说service，receiver，这些变量是无论你的Activity是在前台还是在后台都能够被响应到的，然后调用上面那个用来初始化的函数初始化布局信息。

2. onStart函数:注册一些变量。这些变量必须在Android Activity类在前台的时候才能够被响应。

3. onResume函数:调用一些刷新UI的函数，每当Activity调用到这里时就要刷新一下UI各控件的状态。

4. onPause函数:一般是做一些变量的设置，因为这个时候Activity马上就要切到后台处理，可能有些变量就要被释放掉或者状态要做些相应的调整。

5. onStop函数:反注册在onStart函数中注册的变量。

6. onDestory函数:反注册在onCreate函数中注册的变量。

**说明：所谓魔法数值:**

**是指在代码中直接出现的数值，而只有在这个数值记述的那部分代码中才能明确了解其含义。**

**mAdapter**.notifyDataSetChanged();刷新更改的数据

**关于android的Service的介绍：**

Android中有两种主要方式使用Service，通过调用Context的startService方法或调用Context的bindService方法，本文只探讨纯startService的使用，不涉及任何bindService方法调用的情况。如果想了解bindService的相关使用，请参见[**《Android中bindService基本使用方法概述》。**](http://www.jb51.net/article/76474.htm)

当我们通过调用了Context的startService方法后，我们便启动了Service，通过startService方法启动的Service会一直无限期地运行下去，只有在外部调用Context的stopService或Service内部调用Service的stopSelf方法时，该Service才会停止运行并销毁。

要想使用Service，首先我们要继承自Service，然后重写如下方法:   
**onCreate, onStartCommand, onBind 和 onDestroy。**

这几个方法都是回调方法，都是由Android操作系统在合适的时机调用的，并且需要注意的是这几个回调方法都是在主线程中被调用的。

**onCreate:** 执行startService方法时，如果Service没有运行的时候会创建该Service并执行Service的onCreate回调方法；如果Service已经处于运行中，那么执行startService方法不会执行Service的onCreate方法。也就是说如果多次执行了Context的startService方法启动Service，Service方法的onCreate方法只会在第一次创建Service的时候调用一次，以后均不会再次调用。我们可以在onCreate方法中完成一些Service初始化相关的操作。

**onStartCommand**: 在执行了startService方法之后，有可能会调用Service的onCreate方法，在这之后一定会执行Service的onStartCommand回调方法。也就是说，如果多次执行了Context的startService方法，那么Service的onStartCommand方法也会相应的多次调用。**onStartCommand方法很重要，我们在该方法中根据传入的Intent参数进行实际的操作，比如会在此处创建一个线程用于下载数据或播放音乐等。**

**onBind:** Service中的onBind方法是抽象方法，所以Service类本身就是抽象类，也就是onBind方法是必须重写的，即使我们用不到。在通过startService使用Service时，我们在重写onBind方法时，只需要将其返回null即可。onBind方法主要是用于给bindService方法调用Service时才会使用到。

**onDestroy:** 通过startService方法启动的Service会无限期运行，只有当调用了Context的stopService或在Service内部调用stopSelf方法时，Service才会停止运行并销毁，在销毁的时候会执行Service回调函数。

我们为了探究通过startService方法启动的Service的生命周期以验证上面对各个回调函数方法的描述，写了如下的一个测试案例。   
首先创建一个服务类TestService，该类继承自Service，代码如下:

[?](http://www.jb51.net/article/76470.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package com.ispring.startservicedemo;    import android.app.Service;  import android.content.Intent;  import android.os.IBinder;  import android.util.Log;    public class TestService extends Service {     @Override   public void onCreate() {    Log.i("DemoLog","TestService -> onCreate, Thread ID: " + Thread.currentThread().getId());    super.onCreate();   }     @Override   public int onStartCommand(Intent intent, int flags, int startId) {    Log.i("DemoLog", "TestService -> onStartCommand, startId: " + startId + ", Thread ID: " + Thread.currentThread().getId());    return START\_STICKY;   }     @Override   public IBinder onBind(Intent intent) {    Log.i("DemoLog", "TestService -> onBind, Thread ID: " + Thread.currentThread().getId());    return null;   }     @Override   public void onDestroy() {    Log.i("DemoLog", "TestService -> onDestroy, Thread ID: " + Thread.currentThread().getId());    super.onDestroy();   }  } |

我们在TestService的各个回调方法中只是简单打印出了相应的信息，并没有做很多复杂的处理操作。

然后我们在Activity中调用该Serivce，Activity中相应的代码如下:

[?](http://www.jb51.net/article/76470.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package com.ispring.startservicedemo;    import android.app.Activity;  import android.content.Intent;  import android.os.Bundle;  import android.util.Log;      public class MainActivity extends Activity {     @Override   protected void onCreate(Bundle savedInstanceState) {    super.onCreate(savedInstanceState);    setContentView(R.layout.activity\_main);      Log.i("DemoLog", "Thread ID: " + Thread.currentThread().getId());      Log.i("DemoLog", "before test startService");      //连续启动Service    Intent intent1 = new Intent(this, TestService.class);    startService(intent1);    Intent intent2 = new Intent(this, TestService.class);    startService(intent2);    Intent intent3 = new Intent(this, TestService.class);    startService(intent3);      //停止Service    Intent intent4 = new Intent(this, TestService.class);    stopService(intent4);      //再次启动Service    Intent intent5 = new Intent(this, TestService.class);    startService(intent5);      Log.i("DemoLog", "after test startService");   }  } |

我们在Activity中，首先连续三次调用了Activity的startService方法以启动Service，然后调用Activity的stopService方法停止Service，然后又通过调用Activity的startService方法启动Service。

运行程序的输出结果如下:

我们分析一下上面的输出结果，首先打印出了主线程的ID是1， 然后我们发现后面所有在回调函数中打印出的执行线程的ID也就是1，这就说明了Service中的各个回调方法是运行在主线程中的。其次我们可以发现在我们连续调用了三次startService方法之后，只触发了一次onCreate回调方法，触发了三次onStartCommand方法，在onStartCommand中我们可以读取到通过startService方法传入的Intent对象，并且这三次的startId都不同，分别是1,2,3，每次调用startService都会自动分配一个startId，startId可以用来区分不同的startService的调用，一般情况下startId都是从1开始计数，以后每次调用startService之后startId自动加一递增。

之后我们又调用了Activity的stopService(intent4)方法用于停止Service，通过输出结果我们发现Service执行了onDestroy方法，一般情况下我们可以在onDestroy方法中执行一些资源释放的操作。执行完onDestroy之后该Service的实例就销毁了。虽然我们之前调用了三次startService方法，但是只要调用一次stopService就可以让运行中的Service停止运行并销毁。

最后我们再次通过startService(intent5)启动Service时，通过输出结果我们发现再次执行了Service的onCreate方法，这说明Service在通过stopService销毁之后重新创建了，并随之再次调用onStartCommand回调方法，并且startId再次从1开始计数。

最后需要注意的是我们在Activity中操作Service的开始和结尾处分别写了两句输出代码，分别是

复制代码 代码如下:

Log.i("DemoLog", "before test startService");

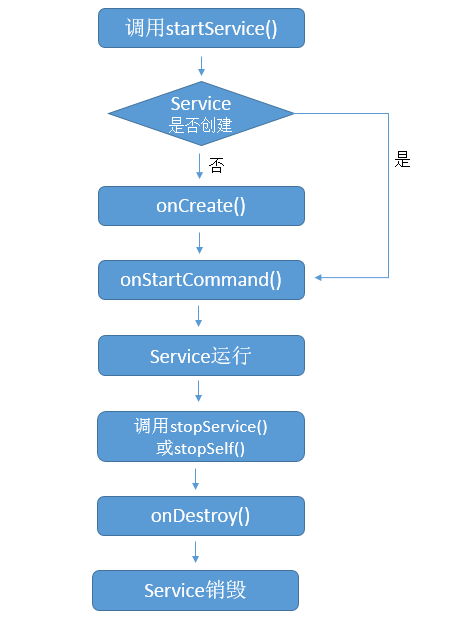
和

复制代码 代码如下:

Log.i("DemoLog", "after test startService");

但是我们再看一下输出结果会发现，程序直接上来在输出了before test startService之后，却立即输出了after test startService，在这之后才是TestService内部各个回调方法的输出，这说明startService()方法和stopService()方法在执行完后立即返回了，也就是这两个方法都不是阻塞式的，启动service和停止service都是异步操作，startService()、stopService()都是将intent对象发送给Android Framework，然后Framework层异步地启动、停止Service。

我们用一张图来概括一下通过startService启动的Service的生命周期：



当Android面临内存匮乏的时候，可能会销毁掉你当前运行的Service，然后待内存充足的时候可以重新创建Service，Service被Android系统强制销毁并再次重建的行为依赖于Service中onStartCommand方法的返回值。我们常用的返回值有三种值，**START\_NOT\_STICKY、START\_STICKY和START\_REDELIVER\_INTENT**，这三个值都是Service中的静态常量。

**START\_NOT\_STICKY：** 如果返回START\_NOT\_STICKY，表示当Service运行的进程被Android系统强制杀掉之后，不会重新创建该Service，当然如果在其被杀掉之后一段时间又调用了startService，那么该Service又将被实例化。那什么情境下返回该值比较恰当呢？如果我们某个Service执行的工作被中断几次无关紧要或者对Android内存紧张的情况下需要被杀掉且不会立即重新创建这种行为也可接受，那么我们便可将 onStartCommand的返回值设置为START\_NOT\_STICKY。举个例子，某个Service需要定时从服务器获取最新数据：通过一个定时器每隔指定的N分钟让定时器启动Service去获取服务端的最新数据。当执行到Service的onStartCommand时，在该方法内再规划一个N分钟后的定时器用于再次启动该Service并开辟一个新的线程去执行网络操作。假设Service在从服务器获取最新数据的过程中被Android系统强制杀掉，Service不会再重新创建，这也没关系，因为再过N分钟定时器就会再次启动该Service并重新获取数据。

**START\_STICKY：** 如果返回START\_STICKY，表示Service运行的进程被Android系统强制杀掉之后，Android系统会将该Service依然设置为started状态（即运行状态），但是不再保存onStartCommand方法传入的intent对象，然后Android系统会尝试再次重新创建该Service，并执行onStartCommand回调方法，但是onStartCommand回调方法的Intent参数为null，也就是onStartCommand方法虽然会执行但是获取不到intent信息。如果你的Service可以在任意时刻运行或结束都没什么问题，而且不需要intent信息，那么就可以在onStartCommand方法中返回START\_STICKY，比如一个用来播放背景音乐功能的Service就适合返回该值。

**START\_REDELIVER\_INTENT：** 如果返回START\_REDELIVER\_INTENT，表示Service运行的进程被Android系统强制杀掉之后，与返回START\_STICKY的情况类似，Android系统会将再次重新创建该Service，并执行onStartCommand回调方法，但是不同的是，Android系统会再次将Service在被杀掉之前最后一次传入onStartCommand方法中的Intent再次保留下来并再次传入到重新创建后的Service的onStartCommand方法中，这样我们就能读取到intent参数。只要返回START\_REDELIVER\_INTENT，那么onStartCommand重的intent一定不是null。如果我们的Service需要依赖具体的Intent才能运行（需要从Intent中读取相关数据信息等），并且在强制销毁后有必要重新创建运行，那么这样的Service就适合返回START\_REDELIVER\_INTENT。