***Android知识点梳理***

***（入门级别）***

***枯叶生，绿叶落，红豆微微张；***

***冰初融，水始流，绿随将入海；***

***风微起，日渐红，思潮似波涛；***

***歌声起，拙笔题，浮生亦浮云。***

***<https://github.com/EbookFoundation/free-programming-books/blob/master/free-programming-books-zh.md>***

***免费书籍地址***

**写在前面：**

本文章只是基于别人总结出来的知识和结构上进行增删的，本文并不适合进阶高阶和中级解惑，如果没有你想要的，自己去找资料，看源码，哔哔的时间都可以看好多代码，找好多资料了。没人有义务去教你该怎么，怎么解决。所以少年们，学会独立解决问题总是好的。

好吧，本文仅仅对基础知识做了概述（看清楚了是概述不是详述），能分享出来的东西不一定都是好的，所以有帮助最好，没帮助把你去看（个）锤子。喷子自行绕道（就算你喷了，我也不会改的。）

文章目录

[0、Application+主题 1](#_Toc2237)

[0.1、关于白屏/黑屏的解决方案 1](#_Toc22013)

[0.2、App的主题样式 4](#_Toc6544)

[0.3、App的标题栏 4](#_Toc657)

[1、Activity 5](#_Toc19277)

[1.1 Activity是什么 5](#_Toc18700)

[1.2 Activity四种状态 5](#_Toc24169)

[1.3、Activity生命周期 7](#_Toc28475)

[1.4、进程的优先级 10](#_Toc5112)

[1.5、Activity任务栈 17](#_Toc599)

[1.6、Activity启动模式 18](#_Toc15076)

[1.7、scheme跳转协议 20](#_Toc31217)

[2、Fragment 24](#_Toc26525)

[1、Fragment为什么被称为第五大组件 24](#_Toc12112)

[2、Activity创建Fragment的方式 24](#_Toc12928)

[3、FragmentPageAdapter和FragmentPageStateAdapter的区别 27](#_Toc13514)

[4、Fragment生命周期 28](#_Toc31306)

[5、Fragment的通信 28](#_Toc13535)

[6、Fragment的replace、add、remove方法 28](#_Toc21330)

[7、FragmentTransaction的show、hide方法 28](#_Toc32535)

[3、Service 31](#_Toc20829)

[1、Service是什么 31](#_Toc22778)

[2、Service和Thread的区别 31](#_Toc2378)

[3、Service启动方式 31](#_Toc7063)

[4、Service生命周期 31](#_Toc20713)

[4、Broadcast Receiver 32](#_Toc10870)

[1、Broadcast Receiver是什么 32](#_Toc14007)

[2、Broadcast Receiver的使用场景 32](#_Toc30442)

[3、Broadcast Receiver的种类 32](#_Toc14427)

[4、Broadcast Receiver的实现 32](#_Toc6940)

[5、Broadcast Receiver实现机制 32](#_Toc13837)

[6、LocalBroadcastManager特点 32](#_Toc12872)

[5、WebView 33](#_Toc21085)

[1、WebView安全漏洞 33](#_Toc0)

[2、WebView销毁步骤 33](#_Toc10470)

[3、WebView的jsbridge 33](#_Toc18035)

[4、WebViewClient的onPageFinished 33](#_Toc23546)

[5、WebView后台耗电 33](#_Toc27662)

[6、WebView硬件加速 33](#_Toc14408)

[7、WebView内存泄漏 33](#_Toc21226)

[6、Binder 35](#_Toc26371)

[1、Linux内核的基本知识 35](#_Toc6374)

[2、为什么使用Binder 35](#_Toc10790)

[3、Binder通信模型 35](#_Toc28000)

[4、AIDL 36](#_Toc15282)

[7、Handler 37](#_Toc20970)

[1、Handler是什么 37](#_Toc28713)

[2、Handler使用方法 37](#_Toc1388)

[3、Handler工作原理 37](#_Toc32142)

[4、Handler引起的内存泄漏 37](#_Toc5524)

[8、AsyncTask 38](#_Toc5040)

[1、AsyncTask是什么 38](#_Toc12418)

[2、AsyncTask使用方法 38](#_Toc6030)

[3、AsyncTask工作原理 38](#_Toc24846)

[4、AsyncTask引起的内存泄漏 38](#_Toc8981)

[5、AsyncTask生命周期 39](#_Toc29218)

[6、AsyncTask结果丢失 39](#_Toc13185)

[7、AsyncTask并行or串行 39](#_Toc23036)

[9、HandlerThread 40](#_Toc1074)

[1、HandlerThread产生背景 40](#_Toc21583)

[2、HanlderThread的特点、 40](#_Toc16557)

[10、IntentService 41](#_Toc19100)

[1、IntentService是什么 41](#_Toc20515)

[2、IntentService使用方法 41](#_Toc25667)

[11、View绘画机制+常见的View 42](#_Toc19895)

[11.1自定义View 42](#_Toc9234)

[11.2、TabLayout 43](#_Toc7246)

[11.3、EditText 57](#_Toc26465)

[11.4、CardView 60](#_Toc14787)

[11.5底部导航栏的实现 61](#_Toc31485)

[12、ListView和RecyclerView 63](#_Toc21665)

[1、ListView是什么 63](#_Toc23659)

[2、ListView的RecycleBin机制 63](#_Toc219)

[3、ListView的优化 63](#_Toc12878)

[13、Android项目构建 64](#_Toc2843)

[1、android构建流程 64](#_Toc7413)

[2、jenkins持续集成构建 65](#_Toc10133)

[3、Git常用命令 65](#_Toc18522)

[4、git工作流 65](#_Toc24718)

[5、proguard是什么 65](#_Toc13082)

[6、proguard技术功能 65](#_Toc2544)

[7、proguard工作原理 66](#_Toc31278)

[8、为什么要混淆 66](#_Toc27299)

[14、ANR 67](#_Toc14189)

[1、什么是ANR 67](#_Toc4480)

[2、发生ANR的条件 67](#_Toc12578)

[3、造成ANR的主要原因 67](#_Toc15134)

[4、如何解决ANR 67](#_Toc19250)

[15、OOM 68](#_Toc5835)

[1、什么是OOM 68](#_Toc28337)

[2、OOM相关概念 68](#_Toc27747)

[3、解决OOM 68](#_Toc719)

[16、Bitmap 69](#_Toc29170)

[1、recycle 70](#_Toc10887)

[2、LruCache原理 70](#_Toc22329)

[3、计算inSampleSize 70](#_Toc21629)

[4、缩略图 71](#_Toc32404)

[5、保存Bitmap 71](#_Toc22236)

[6、保存到SD卡 72](#_Toc21967)

[7、三级缓存 72](#_Toc24387)

[8、压缩 72](#_Toc5871)

[17、UI卡顿 83](#_Toc32513)

[1、UI卡顿原理 83](#_Toc14333)

[2、UI卡顿的原因分析 83](#_Toc7717)

[3、UI卡顿的优化 83](#_Toc19016)

[18、内存泄漏 84](#_Toc13368)

[1、Java内存泄漏引起的主要原因 84](#_Toc19659)

[2、Java内存分配策略 84](#_Toc23449)

[3、Android解决内存泄漏的例子 84](#_Toc21133)

[19、内存管理 85](#_Toc15561)

[1、Android内存管理机制 85](#_Toc9527)

[2、内存管理机制的特点 85](#_Toc31034)

[3、内存优化方法 85](#_Toc3386)

[20、冷启动和热启动 86](#_Toc14674)

[1、什么是冷启动和热启动 86](#_Toc29747)

[2、冷启动和热启动的区别 86](#_Toc24512)

[3、冷启动时间的计算 86](#_Toc31609)

[4、冷启动流程 86](#_Toc13407)

[5、冷启动优化 86](#_Toc12270)

[21、优化 88](#_Toc11056)

[1.布局优化 88](#_Toc3469)

[2.内存优化 89](#_Toc20538)

[3.电量优化 89](#_Toc22612)

[4.网络优化 90](#_Toc26448)

[5.启动优化 91](#_Toc15728)

[6.体积优化 91](#_Toc1797)

[7.性能优化 92](#_Toc1752)

[8.业务优化 92](#_Toc28766)

[9.其他优化 93](#_Toc15547)

[22、架构模式 95](#_Toc12549)

[22.1、框架模式MVC在安卓中的实践 97](#_Toc11121)

[22.2、框架模式MVP在安卓中的实践 98](#_Toc19457)

[22.3、框架模式MVVM之DataBinding的实践 99](#_Toc14791)

[谷歌的DataBinding的使用文档 106](#_Toc17729)

[23、插件化 138](#_Toc6882)

[1、插件化解决的问题 138](#_Toc3712)

[2、类加载器（Java中字节码添加到虚拟机中） 138](#_Toc18)

[24、热更新 139](#_Toc2124)

[1、热更新主要流程 139](#_Toc21055)

[2、热更新主流框架 139](#_Toc8242)

[3、热更新的原理 139](#_Toc30692)

[25、进程保活 140](#_Toc7519)

[1、进程的优先级 140](#_Toc5037)

[2、Android进程回收策略 140](#_Toc30518)

[3、Android保活方案 140](#_Toc18738)

[26、Lint 141](#_Toc11690)

[1、什么是Android Lint 141](#_Toc16489)

[2、Lint工作流程 141](#_Toc6499)

[3、配置Lint 141](#_Toc9666)

[27、Kotlin 142](#_Toc6636)

[1、什么是Kotlin 142](#_Toc5132)

[2、Kotlin环境搭建 142](#_Toc31362)

[面试总结： 143](#_Toc32400)

[蓝厂面试问题： 143](#_Toc27202)

[度娘电话面试： 144](#_Toc9485)

[直播公司 145](#_Toc2747)

[由鹅厂与其他公司合资创立的公司 146](#_Toc10353)

[某ding： 147](#_Toc28732)

[某条： 148](#_Toc20103)

[UI基础 149](#_Toc2382)

[应用核心组件 149](#_Toc13302)

[菜单与对话框 150](#_Toc20278)

[数据存储与IO 150](#_Toc26428)

[移动开发主流界面实现 151](#_Toc28548)

[应用核心组件进阶 151](#_Toc31891)

[后台服务 151](#_Toc24294)

[View深入探究 152](#_Toc27099)

[Android新技术 152](#_Toc2020)

[Android网络应用 153](#_Toc13056)

[流行技术 153](#_Toc7217)

[Android高级 153](#_Toc29310)

# 0、Application+主题

为什么放在第一部分讲？

1、Application是程序的主入口；

2、主题决定整个app的风格；

3、在此处需要做的优化很多。

## 0.1、关于白屏/黑屏的解决方案

百度一下你就知道（原理自己参考百度，一般会告诉你有三种方法，然后对比，最后告诉你什么时候用哪种，绝对不是坑，用完之后你就知道。）

以下这篇文章的原理讲的还比较清楚（不过不是参考的这篇，想要学习的孩子，自己看。）：

<http://blog.csdn.net/yanzhenjie1003/article/details/52201896、>

下面是解决方案：

1、在Style里面设置一个主题样式BaseAppThemeLight（并不是Application的样式）

<!-- App Theme Light -->  
<style name="BaseAppThemeLight" parent="Theme.AppCompat.Light.NoActionBar">  
 <item name="android:windowNoTitle">true</item>  
 <item name="android:windowFullscreen">false</item>  
 <item name="android:windowActionBar">false</item>  
 <item name="windowActionModeOverlay">true</item>  
</style>

2、创建一个过度的主题样式，父类是BaseAppThemeLight

<!--在替换CompatAppTheme的parent的时候，只需要在这里做修改-->  
<style name="AppBaseTheme" parent="BaseAppThemeLight">  
 <item name="themeColor">@color/colorPrimary</item>  
</style>

3、创建Splash和AppTheme（这个是创建项目的时候AS帮我们写好的）的父类主题样式；

<style name="CompatAppTheme" parent="AppBaseTheme">  
 <item name="android:windowNoTitle">true</item>  
 <item name="android:windowFullscreen">false</item>  
 <item name="android:windowActionBar">false</item>  
 <item name="windowActionModeOverlay">true</item>  
</style>

4、在res目录下创建values-v21文件夹，并创建styles.xml文件（为了兼容性）

<?xml version="1.0" encoding="utf-8"?>  
<resources>  
  
 <style name="CompatAppTheme" parent="AppBaseTheme">  
 <item name="android:windowNoTitle">true</item>  
 <item name="android:windowFullscreen">false</item>  
 <item name="android:windowActionBar">false</item>  
 <item name="windowActionModeOverlay">true</item>  
 <item name="android:windowDrawsSystemBarBackgrounds">true</item>  
 <item name="android:statusBarColor">@android:color/transparent</item>  
 </style>  
  
</resources>

5、修改AppTheme的父类为CompatAppTheme

<!-- Base application theme. -->  
<style name="AppTheme" parent="CompatAppTheme">  
 <!-- Customize your theme here. -->  
 <item name="android:windowActionBar">true</item>  
 <item name="colorPrimary">@color/colorPrimary</item>  
 <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
 <item name="colorAccent">@color/colorAccent</item>  
</style>

6、创建Splash需要的背景图片splash.xml，图片自己找资源。

<?xml version="1.0" encoding="utf-8"?>  
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
  
 <item>  
 <!-- 图片 -->  
 <bitmap  
 android:id="@+id/bitmap\_splash"  
 android:src="@drawable/ic\_splash" />  
 </item>  
</layer-list>

7、创建Splash的样式SplashTheme；

<style name="SplashTheme" parent="CompatAppTheme">  
 <item name="android:windowBackground">@drawable/splash</item>  
</style>

8、修改AndroidManifest.xml文件（关键部分已经标注为红色）

<application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:supportsRtl="true"  
 android:theme="@style/AppTheme">  
 <activity android:name=".SplashActivity"  
 android:theme="@style/SplashTheme">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN"/>  
  
 <category android:name="android.intent.category.LAUNCHER"/>  
 </intent-filter>  
 </activity>  
  
 <activity android:name=".MainActivity"/>  
  
</application>

9、一点小建议：

在SplashActivity中不推荐设置布局的（setContentView(R.layout.xxx);就是这句话），在开启后马上关闭。代码如下：

public class SplashActivity extends AppCompatActivity {  
  
 @Override  
 protected void onCreate(@Nullable Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 new Handler().postDelayed(new Runnable() {  
 @Override  
 public void run() {  
 startActivity(new Intent(SplashActivity.this, MainActivity.class));  
 finish();  
 }  
 }, 1000);//为了使效果更加明显  
 }  
}

完成之后就会有一种，即开即用的效果。第9步中的建议的原因是：Splash.xml是在布局显示之前展示的界面，即onCreate（）方法走完就会消失或者activity自身的布局已经加载完成。说简单就是：如果设置setContentView(R.layout.xxx)，那么界面的展示效果就是由Splash到xxx布局。而在这个变化过程中会出现界面的收缩而不是界面的切换，这样就会给用户一种界面卡顿的感觉。所以不推荐。

## 0.2、App的主题样式

我现在不想写，不代表以后就会写，说不定就不写了。

## 0.3、App的标题栏

1、在mvc和MVP模式下，可以同过重写setContentView的三个方法来实现通用的标题栏。不多说，参考项目代码即可。（已经完成）

2、在MVVM模式下就是一个坑，我也不知道怎么写比较好。

如果用到DataBindingUtil的setContentView方法，你完全不能通过重写来实现，而且activity只能使用一次DataBindingUtil.setContentView(this, R.layout.activity\_main);你无从下手，之前项目给的经验是抽出来，但是每次需要的标题的时候就必须将布局添加进去。代码还是多了，如何优化代码就成了问题（日后再处理。）

# 1、Activity

## 1.1 Activity是什么

Activity是四大组件之一，它提供一个界面让用户点击和各种滑动操作。Activity是一个应用程序组件，提供一个屏幕，用户可以用来交互为了完成某项任务。Activity中所有操作都与用户密切相关，是一个负责与用户交互的组件，可以通过setContentView(View)来显示指定控件。在一个android应用中，一个Activity通常就是一个单独的屏幕，它上面可以显示一些控件也可以监听并处理用户的事件做出响应。

Activity之间通过Intent进行通信。

## 1.2 Activity四种状态

在android 中，Activity 拥有四种基本状态：

### 1.Running

一个新 Activity 启动入栈后，它显示在屏幕最前端，处理是处于栈的最顶端（Activity栈顶），此时它处于可见并可和用户交互的激活状态,叫做活动状态或者运行状态（active or running）。

### 2.Paused

当Activity失去焦点， 被一个新的非全屏的Activity 或者一个透明的Activity 被放置在栈顶，此时的状态叫做暂停状态（Paused）。此时它依然与窗口管理器保持连接，Activity依然保持活力（保持所有的状态，成员信息，和窗口管理器保持连接），但是在系统内存极端低下的时候将被强行终止掉。所以它仍然可见，但已经失去了焦点故不可与用户进行交互。

### 3.Stopped

如果一个Activity被另外的Activity完全覆盖掉，叫做停止状态（Stopped）。它依然保持所有状态和成员信息，但是它不再可见，所以它的窗口被隐藏，当系统内存需要被用在其他地方的时候，Stopped的Activity将被强行终止掉。

### 4.Killed

如果一个Activity是Paused或者Stopped状态，系统可以将该Activity从内存中删除，Android系统采用两种方式进行删除，要么要求该Activity结束，要么直接终止它的进程。当该Activity再次显示给用户时，它必须重新开始和重置前面的状态。

### 状态转换

当一个 Activity 实例被创建、销毁或者启动另外一个 Activity 时，它在这四种状态之间进行转换，这种转换的发生依赖于用户程序的动作。下图说明了 Activity 在不同状态间转换的时机和条件：

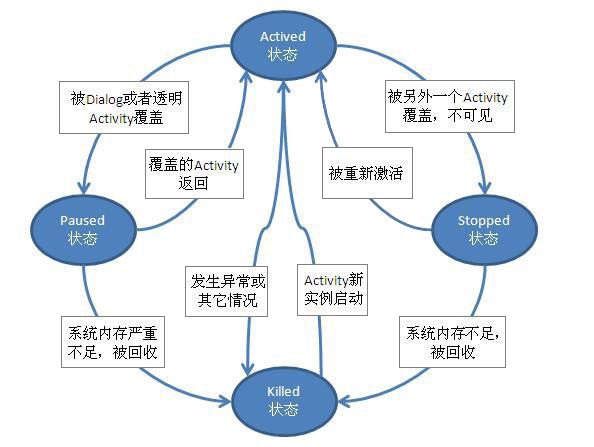


图1.2.1 Activity 的状态转换

如上所示，Android程序员可以决定一个Activity的“生”，但不能决定它的“死”，也就是说程序员可以启动一个Activity，但是却不能手动的“结束”一个 Activity。当你调用Activity.finish()方法时，结果和用户按下BACK键一样：告诉Activity Manager该Activity实例完成了相应的工作，可以被“回收”。随后Activity Manager激活处于栈第二层的Activity并重新入栈，同时原 Activity 被压入到栈的第二层，从Active状态转到Paused状态。例如：从Activity1中启动了Activity2，则当前处于栈顶端的是Activity2，第二层是Activity1，当我们调用Activity2.finish()方法时，Activity Manager重新激活Activity1并入栈，Activity2从Active状态转换Stoped状态，Activity1.onActivityResult(int requestCode, int resultCode, Intent data)方法被执行，Activity2 返回的数据通过data参数返回给 Activity1。

## 1.3、Activity生命周期

下面的图显示了Activity的重要状态转换（生命周期），矩形框表明Activity在状态转换之间的回调接口，开发人员可以重载实现以便执行相关代码，带有颜色的椭圆形表明Activity所处的状态。



图1.3.1 Activity的状态转换的方法和实现

user navigates to the activity 用户导航到该活动

Apps with higher priority need memory具有更高优先级的应用程序需要内存

Aonther activity comes into the foreground 另一个活动进入前台

the activity is no longer visible 活动不再可见

the activity is finishing or being destroyed by the system 活动正在完成或被系统破坏

user retruns to the activity 用户对该活动进行重新处理

在上图中，Activity有三个关键的循环：

1.整个的生命周期，从[onCreate](https://baike.baidu.com/item/onCreate" \t "https://baike.baidu.com/item/activity/_blank)(Bundle)开始到onDestroy()结束。Activity在onCreate()设置所有的“全局”状态，在onDestory()释放所有的资源。例如：某个Activity有一个在后台运行的线程，用于从网络下载数据，则该Activity可以在onCreate()中创建线程,在onDestory()中停止线程。

2.可见的生命周期，从onStart()开始到onStop()结束。在这段时间，可以看到Activity在屏幕上，尽管有可能不在前台，不能和用户交互。在这两个接口之间，需要保持显示给用户的[UI](https://baike.baidu.com/item/UI" \t "https://baike.baidu.com/item/activity/_blank)数据和资源等，例如：可以在onStart中注册一个IntentReceiver来监听数据变化导致UI的变动，当不再需要显示时候，可以在onStop()中注销它。onStart()，onStop()都可以被多次调用，因为Activity随时可以在可见和隐藏之间转换。

3.前台的[生命周期](https://baike.baidu.com/item/%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F" \t "https://baike.baidu.com/item/activity/_blank)，从onResume()开始到onPause()结束。在这段时间里，该Activity处于所有 Activity的最前面，和用户进行交互。Activity可以经常性地在resumed和paused状态之间切换，例如：当设备准备休眠时，当一个 Activity处理结果被分发时，当一个新的Intent被分发时。所以在这些接口方法中的代码应该属于非常轻量级的。

### 详解：

onCreate()

onStart()

onResume()

onPause()

onStop()

onDestroy()

onRestart()

PS：在某里上面试问到的生命周期是9种，另外两种分别是onPostCreate onPostResume 其实纯属扯蛋装逼，用的好自然确实对app优化有帮助，问题是优化的太有限了，官方都不推荐调用。在这里需要解释一下：

**onPostCreate（）**

官方解释：

当Activity启动时调用(在onStart()和onRestoreInstanceState(Bundle)调用之后之后)。应用程序通常不会实现这个方法;它用于在应用程序代码运行后进行最终初始化的系统类。继承类必须调用该方法父类的实现。不这样做，就会抛出一个异常。

用途（待添加）：

想要在Activity布局完成，彻底跑起来之后，再获取当前Activity的窗口中，某个View的宽高。

一般实现思路弄个Handler，发个Message出来，使用sendMessageDelayed或者sendEmptyMessageDelayed。说白了就是延迟若干时间之后，等Activity彻底跑起来之后，再取获取View的宽高。缺点：delay的时间太长了，使用这个获取的宽高的值的时候，自己获取宽高的函数还没被调用；delay的时间太短了，在某些配置比较低的机型上，在delay的时间内Activity没能彻底的跑起来，获取到的值可能会不正确。

**onPostResume()**

*/\*\*  
 \* Called when activity resume is complete (after {****@link*** *#onResume} has  
 \* been called). Applications will generally not implement this method;  
 \* it is intended for system classes to do final setup after application  
 \* resume code has run.  
 \*  
 \* <p><em>Derived classes must call through to the super class's  
 \* implementation of this method. If they do not, an exception will be  
 \* thrown.</em></p>  
 \*  
 \** ***@see*** *#onResume  
 \*/*@CallSuper  
protected void onPostResume() {  
 final Window win = getWindow();  
 if (win != null) win.makeActive();  
 if (mActionBar != null) mActionBar.setShowHideAnimationEnabled(true);  
 mCalled = true;  
}

在activity恢复完成后调用它(在调用onResume后)。应用程序通常不会实现这个方法;它的目的是在应用程序恢复代码运行后为系统类做最后的设置。继承类必须调用该方法父类的实现。不这样做，就会抛出一个异常。

这个类的作用官方注释：在应用程序恢复代码运行后为系统类做最后的设置

## 1.4、进程的优先级

空进程

后台进程

服务进程

可见进程

前台进程

进程的生命周期

Android系统会尽可能长的延续一个应用程序进程，但在内存过低的时候，仍然会不可避免需要移除旧的进程。为了决定哪些进程留下，哪些进程被杀死，系统根据在进程中在运行的组件及组件的状态，为每一个进程分配了一个优先级等级。优先级最低的进程首先被杀死。这个进程重要性的层次结构主要有五个等级。

Android会尽可能高的估量一个进程的级别。比如，一个进程拥有一个可见状态的activity和一个service，这个进程会被认为是可见进程，而不是服务进程。

一个进程的级别可能会由于其它进程依赖于它而升高。一个为其它进程提供服务的进程级别永远高于使用它服务的进程。比如说，如果A进程中的内容提供者为进程B中的客户端提供服务，或进程A中的服务为进程B中的组件所绑定，则A进程高于或者等于进程B的等级。

**进程的五个常用等级:**

### 一．前台进程（Foreground process）

前台进程是用户当前做的事所必须的进程，如果满足下面各种情况中的一种，一个进程被认为是在前台：

1、进程持有一个正在与用户交互的Activity。

2、进程持有一个Service，这个Service处于这几种状态:①Service与用户正在交互的Activity绑定。②Service是在前台运行的，即它调用了 startForeground()。③Service正在执行它的生命周期回调函数（onCreate(), onStart(), or onDestroy()）。

3、进程持有一个BroadcastReceiver，这个BroadcastReceiver正在执行它的 onReceive() 方法。

杀死前台进程需要用户交互，因为前台进程的优先级是最高的。

### 二．可见进程（Visible process）

如果一个进程不含有任何前台的组件，但仍可被用户在屏幕上所见。当满足如下任一条件时，进程被认为是可见的：

1、进程持有一个Activity，这个Activity不在前台，但是仍然被用户可见（处于onPause()调用后又没有调用onStop()的状态，比如，前台的activity打开了一个对话框，这样activity就会在其后可见）。

2、进程持有一个Service，这个Service和一个可见的（或者前台的）Activity绑定。

可见的进程也被认为是很重要的，一般不会被销毁，除非是为了保证所有前台进程的运行而不得不杀死可见进程的时候。

### 三．服务进程 （Service process）

如果一个进程中运行着一个service，这个service是通过 startService() 开启的，并且不属于上面两种较高优先级的情况，这个进程就是一个服务进程。

尽管服务进程没有和用户可以看到的东西绑定，但是它们一般在做的事情是用户关心的，比如后台播放音乐，后台下载数据等。所以系统会尽量维持它们的运行，除非系统内存不足以维持前台进程和可见进程的运行需要。

### 四．后台进程 （Background process）

如果进程不属于上面三种情况，但是进程持有一个用户不可见的activity（activity的onStop()被调用，但是onDestroy()没有调用的状态），就认为进程是一个后台进程。

后台进程不直接影响用户体验，系统会为了前台进程、可见进程、服务进程而任意杀死后台进程。

通常会有很多个后台进程存在，它们会被保存在一个LRU (least recently used)列表中，这样就可以确保用户最近使用的activity最后被销毁，即最先销毁时间最远的activity。

### 五．空进程

如果一个进程不包含任何活跃的应用组件，则认为是空进程。

例如：一个进程当中已经没有数据在运行了，但是内存当中还为这个应用驻留了一个进程空间。

保存这种进程的唯一理由是为了缓存的需要，为了加快下次要启动这个进程中的组件时的启动时间。系统为了平衡进程缓存和底层内核缓存的资源，经常会杀死空进程。

### Service中新开线程和直接新开线程的区别

（1）若直接在Activity中新开一条线程来做耗时操作，当该Activity退出到桌面或其他情况时将成为一个后台进程。

（2）若在Service中新启动线程，则此时Android会依据进程中当前活跃组件重要程度，将其判断为服务进程，优先级比（1）高。

因为服务进程的优先级比后台进程的优先级高，所以对于一个需要启动一个长时间操作的activity来说，开启一个service比创建一个工作线程的方法更好，尤其是对于操作将很可能超出activity的持续时间时。

比如要上传一个图片文件，应该开启一个service来进行上传工作，这样在用户离开activity时工作仍在进行。使用service将会保证操作至少有服务进程的优先级。

### 尽可能保证Service不死常用技巧总结

#### 1、通过priority设置最高优先级

在AndroidManifest.xml文件中对于intent-filter可以通过android:priority = “1000”这个属性设置最高优先级，1000是最高值，如果数字越小则优先级越低，同时实用于广播。

<service

android:name="com.dbjtech.acbxt.waiqin.UploadService"

android:enabled="true" >

<intent-filter android:priority="1000" >

<action android:name="com.dbjtech.myservice" />

</intent-filter> </service>

#### 2、调用startForeground()

在onStartCommand里面调用startForeground()方法把Service提升为前台进程级别，然后再onDestroy里面要记得调用stopForeground ()方法。

#### 3、onStartCommand方法，返回START\_STICKY。

手动返回START\_STICKY，亲测当service因内存不足被kill，当内存又有的时候，service又被重新创建，比较不错，但是不能保证任何情况下都被重建，比如进程被干掉了…

public int onStartCommand(Intent intent, int flags, int startId) {

flags = START\_STICKY;

return super.onStartCommand(intent, flags, startId);

}

补充说明：onStartCommand()方法，返回的是一个int整形。  
这个整形可以有以下四个取值：  
1):START\_STICKY：如果service进程被kill掉，保留service的状态为开始状态，但不保留递送的intent对象。随后系统会尝试重新创建service，由于服务状态为开始状态，所以创建服务后一定会调用onStartCommand(Intent,int,int)方法。如果在此期间没有任何启动命令被传递到service，那么参数Intent将为null。   
2):START\_NOT\_STICKY：“非粘性的”。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统不会自动重启该服务   
3):START\_REDELIVER\_INTENT：重传Intent。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统会自动重启该服务，并将Intent的值传入。   
4):START\_STICKY\_COMPATIBILITY：START\_STICKY的兼容版本，但不保证服务被kill后一定能重启。

#### 4、在onDestroy方法里发广播重启service

service +broadcast 方式，就是当service走ondestory的时候，发送一个自定义的广播，当收到广播的时候，重新启动service。（第三方应用或是在setting里-应用-强制停止时，APP进程就直接被干掉了，onDestroy方法都进不来，所以无法保证会执行）

<receiver android:name="com.dbjtech.acbxt.waiqin.BootReceiver" >

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED" />

<action android:name="android.intent.action.USER\_PRESENT" />

<action android:name="com.dbjtech.waiqin.destroy" />//这个就是自定义的action

</intent-filter>

</receiver>

在onDestroy时：

@Override

public void onDestroy() {

stopForeground(true);

Intent intent = new Intent("com.dbjtech.waiqin.destroy");

sendBroadcast(intent);

super.onDestroy();

}

在BootReceiver里：

public class BootReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

if (intent.getAction().equals("com.dbjtech.waiqin.destroy")) {

//TODO

//在这里写重新启动service的相关操作

startUploadService(context);

}

}

}

#### 5、监听系统广播判断Service状态。

通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的Service是否还存活。

<receiver android:name="com.dbjtech.acbxt.waiqin.BootReceiver" >

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED" />

<action android:name="android.intent.action.USER\_PRESENT" />

<action android:name="android.intent.action.PACKAGE\_RESTARTED" />

<action android:name="com.dbjtech.waiqin.destroy" />

</intent-filter>

</receiver>

BroadcastReceiver中：

@Override

public void onReceive(Context context, Intent intent) {

if (Intent.ACTION\_BOOT\_COMPLETED.equals(intent.getAction())) {

System.out.println("手机开机了....");

startUploadService(context);

}

if (Intent.ACTION\_USER\_PRESENT.equals(intent.getAction())) {

startUploadService(context);

}

}

#### 6、Application加上Persistent属性。

看Android的文档知道，当进程长期不活动，或系统需要资源时，会自动清理门户，杀死一些Service，和不可见的Activity等所在的进程。但是如果某个进程不想被杀死（如数据缓存进程，或状态监控进程，或远程服务进程），应该怎么做，才能使进程不被杀死。

add android:persistent=”true” into the section in your AndroidManifest.xml

加上以上属性相当于将该进程设置为常驻内存进程。   
切记，这个不可滥用，一般只适用于放在/system/app下的app，系统中用这个的service，app一多，整个系统可能就会崩溃。   
比如系统phone中配置了android:persistent=”true”属性，并且Phone.apk是安装在/system/app/目录下的，所以在开机时会自动启动PhoneApp类。

参考地址：

<http://blog.csdn.net/mad1989/article/details/22492519#>

<http://blog.csdn.net/wuseyukui/article/details/48004687>

## 1.5、Activity任务栈

**特点：先进后出**

Android 是通过一种 Activity 栈的方式来管理 Activity 的，一个 Activity 的实例的状态决定它在栈中的位置。处于前台的 Activity 总是在栈的顶端，当前台的 Activity 因为异常或其它原因被销毁时，处于栈第二层的 Activity 将被激活，上浮到栈顶。当新的 Activity 启动入栈时，原 Activity 会被压入到栈的第二层。一个 Activity 在栈中的位置变化反映了它在不同状态间的转换。Activity 的状态与它在栈中的位置关系如下图所示：

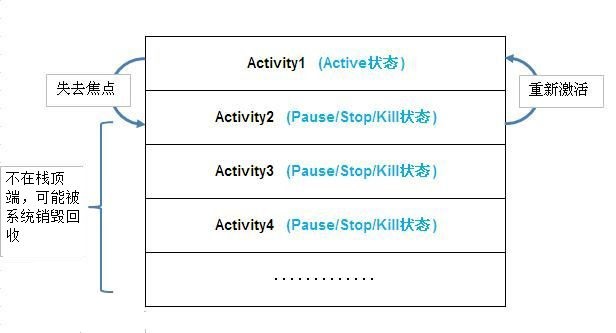


图1.2.2 Activity 的状态与它在栈中的位置关系

如上所示，除了最顶层即处在 Active 状态的 Activity 外，其它的 Activity 都有可能在系统内存不足时被回收，一个 Activity 的实例越是处在栈的底层，它被系统回收的可能性越大。系统负责管理栈中 Activity 的实例，它根据 Activity 所处的状态来改变其在栈中的位置。

## 1.6、Activity启动模式

standard 标准

singletop 置顶

singletask 单一任务栈

singleinstance

**standard**

默认模式，可以不用写配置。在这个模式下，都会默认创建一个新的实例。因此，在这种模式下，可以有多个相同的实例，也允许多个相同Activity叠加。

例如：

若有一个Activity名为A1, 上面有一个按钮可跳转到A1。那么如果点击按钮，便会新启一个Activity A1叠在刚才的A1之上，再点击，又会再新启一个在它之上……

点back键会依照栈顺序依次退出。

**singleTop**

可以有多个实例，但是不允许多个相同Activity叠加。即，如果Activity在栈顶的时候，启动相同的Activity，不会创建新的实例，而会调用其onNewIntent方法。

例如：

若我有两个Activity名为B1,B2,两个Activity内容功能完全相同，都有两个按钮可以跳到B1或者B2，唯一不同的是B1为standard，B2为singleTop。

若我意图打开的顺序为B1->B2->B2，则实际打开的顺序为B1->B2（后一次意图打开B2，实际只调用了前一个的onNewIntent方法）

若我意图打开的顺序为B1->B2->B1->B2，则实际打开的顺序与意图的一致，为B1->B2->B1->B2。

**singleTask**

只有一个实例。在同一个应用程序中启动他的时候，若Activity不存在，则会在当前task创建一个新的实例，若存在，则会把task中在其之上的其它Activity destory掉并调用它的onNewIntent方法。

如果是在别的应用程序中启动它，则会新建一个task，并在该task中启动这个Activity，singleTask允许别的Activity与其在一个task中共存，也就是说，如果我在这个singleTask的实例中再打开新的Activity，这个新的Activity还是会在singleTask的实例的task中。

例如：

若我的应用程序中有三个Activity,C1,C2,C3，三个Activity可互相启动，其中C2为singleTask模式，那么，无论我在这个程序中如何点击启动，如：C1->C2->C3->C2->C3->C1-C2，C1,C3可能存在多个实例，但是C2只会存在一个，并且这三个Activity都在同一个task里面。

但是C1->C2->C3->C2->C3->C1-C2，这样的操作过程实际应该是如下这样的，因为singleTask会把task中在其之上的其它Activity destory掉。

操作：C1->C2 C1->C2->C3 C1->C2->C3->C2 C1->C2->C3->C2->C3->C1 C1->C2->C3->C2->C3->C1-C2

实际：C1->C2 C1->C2->C3 C1->C2 C1->C2->C3->C1 C1->C2

若是别的应用程序打开C2，则会新启一个task。

如别的应用Other中有一个activity，taskId为200，从它打开C2，则C2的taskIdI不会为200，例如C2的taskId为201，那么再从C2打开C1、C3，则C2、C3的taskId仍为201。

注意：如果此时你点击home，然后再打开Other，发现这时显示的肯定会是Other应用中的内容，而不会是我们应用中的C1 C2 C3中的其中一个。

**singleInstance**

只有一个实例，并且这个实例独立运行在一个task中，这个task只有这个实例，不允许有别的Activity存在。

例如：

程序有三个Activity：D1,D2,D3，三个Activity可互相启动，其中D2为singleInstance模式。那么程序从D1开始运行，假设D1的taskId为200，那么从D1启动D2时，D2会新启动一个task，即D2与D1不在一个task中运行。假设D2的taskId为201，再从D2启动D3时，D3的taskId为200，也就是说它被压到了D1启动的任务栈中。

若是在别的应用程序打开D2，假设Other的taskId为200，打开D2，D2会新建一个task运行，假设它的taskId为201，那么如果这时再从D2启动D1或者D3，则又会再创建一个task，因此，若操作步骤为other->D2->D1，这过程就涉及到了3个task了。

## 1.7、scheme跳转协议

Android中的scheme是一种页面内跳转协议，通过定义自己的scheme协议，可以跳转到app中的各个页面

1. 服务器可以定制化告诉app跳转哪个页面
2. App可以通过跳转到另一个App页面
3. 可以通过H5页面跳转页面

**1.7.1协议格式**

Uri.parse("qh://test:8080/goods?goodsId=8897&name=fuck")

qh: 代表Scheme协议名称

test: 代表Scheme作用的地址域

8080: 代表改路径的端口号

/goods: 代表的是指定页面(路径)

goodsId和name: 代表传递的两个参数

**1.7.2Scheme使用**

定义一个Scheme

<activity android:name=".SchemeActivity">

<!-- 给页面添加指定的过滤器-->

<intent-filter>

<!--该页面的路径配置-->

<data

android:host="test"

android:path="/goods"

android:port="8080"

android:scheme="qh"/>

<!--下面这几行也必须得设置-->

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

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

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

</intent-filter>

</activity>

获取Scheme跳转的参数

Uri uri = getIntent().getData();

if (uri != null) {

// 完整的url信息

String s = uri.toString();

sb.append(s + "\n");

// scheme部分

String scheme = uri.getScheme();

sb.append("scheme=" + scheme + "\n");

// host部分

String host = uri.getHost();

sb.append("host=" + host + "\n");

// 访问路劲

String path = uri.getPath();

sb.append("path=" + path + "\n");

//port部分

int port = uri.getPort();

sb.append("port=" + port + "\n");

// Query部分

String query = uri.getQuery();

sb.append("query=" + query + "\n");

//获取指定参数值

String goodsId = uri.getQueryParameter("goodsId");

sb.append("goodsId=" + goodsId + "\n");

//列举所以参数名

Set<String> queryParameterNames = uri.getQueryParameterNames();

tv\_scheme.setText(sb.toString());

}

调用方式

**1. 原生调用**

Intent intent1 = new Intent(Intent.ACTION\_VIEW, Uri.parse("qh://test:8080/goods?goodsId=8897&name=fuck"));

startActivity(intent1);

**2. html调用**

<a href="qh://test:8080/goods?goodsId=8897&name=fuck">打开商品详情</a>

**判断某个Scheme是否有效**

Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse("qh://test:8080/goods?goodsId=8897&name=fuck"));

List<ResolveInfo> activities = getPackageManager().queryIntentActivities(intent, 0);

boolean isValid = !activities.isEmpty();

if (isValid) {

startActivity(intent);

}

# 2、Fragment

## 1、Fragment为什么被称为第五大组件

Fragment比Activity更节省内存，其切换模式也更加舒适，使用频率不低于四大组件，且有自己的生命周期，而且必须依附于Activity

## 2、Activity创建Fragment的方式

### 静态创建：

**1、在activity的xml文件申明一个Fragment的xml文件：**

<fragment

android:layout\_width="0dp"

android:layout\_height="match\_parent"

android:layout\_weight="1"

android:name="com.example.administrator.fragmenttest.LeftFragment"

tools:layout="@layout/my\_layout"/>

**2、创建Fragment的xml文件my\_layout**

**3、创建Fragment类**

public class MyFragment extends Fragment {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

}

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

//通过参数中的布局填充获取对应布局

View view =inflater.inflate(R.layout.my\_layout,container,false);

return view;

}

@Override

public void onPause() {

super.onPause();

}

}

### 动态创建

**1、在activity的xml文件中创建一个Fragment的容器：**

<FrameLayout

android:id="@+id/container"

android:layout\_width="0dp"

android:layout\_height="match\_parent"

android:layout\_weight="match\_parent"/>

**2、创建一个Fragment类：**

（代码同上）

**3、在activity里面代码添加Fragment：**

public class MainActivity extends AppCompatActivity {

//声明本次使用到的java类

FragmentManager fragmentManager;

FragmentTransaction fragmentTransaction;

RightFragment rightFragment;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

/\*在activity对应java类中通过getFragmentManager()获得FragmentManager，用于管理ViewGrop中的fragment \* \*/

fragmentManager=getFragmentManager();

/\*FragmentManager要管理fragment（添加，替换以及其他的执行动作）

\*的一系列的事务变化，需要通过fragmentTransaction来操作执行

\*/

fragmentTransaction = fragmentManager.beginTransaction();

//实例化要管理的fragment

myFragment = new MyFragment();

//通过添加（事务处理的方式）将fragment加到对应的布局中

fragmentTransaction.add(R.id.container,myFragment );

//事务处理完需要提交

fragmentTransaction.commit();

}

}

小结：当前流行的用法是动态添加，但是需要注意fragment重叠的问题

### 重叠解决方案：（需要校验，并没有验证）

1、在实现Fragment的切换并保存Fragment之前的状态的功能功能之后，有个很大的问题：当我们把app退到后台，我们去玩其他的app，过一段时间回来，这个时候我们的app已经被销毁，我们按多任务键切换回来，发现界面上多个Fragment出现了重叠的情况，这是因为多个Fragment同时显示了，出现了重叠的情况，解决的办法如下：重写Activity的onRestoreInstanceState方法就能很好的解决了Fragment重叠的情况

@Override

protected void onRestoreInstanceState(Bundle savedInstanceState) {

super.onRestoreInstanceState(savedInstanceState);

for (int i = 0; i < radioGroup.getChildCount(); i++) {

RadioButton mTab = (RadioButton) radioGroup.getChildAt(i);

FragmentManager fm = getSupportFragmentManager();

Fragment fragment = fm.findFragmentByTag((String) mTab.getTag());

FragmentTransaction ft = fm.beginTransaction();

if (fragment != null) {

if (!mTab.isChecked()) {

ft.hide(fragment);

}

}

ft.commit();

}

}

2、如果在Fragment有操作toolbar的菜单的情况，除了要在Fragment中设置setHasOptionsMenu(true);之外，还需要Fragment中重写onHiddenChanged方法：

@Override

public void onHiddenChanged(boolean hidden) {

super.onHiddenChanged(hidden);

if (!hidden) {

((AppCompatActivity) getActivity()).setSupportActionBar(mToolbar);

((AppCompatActivity)getActivity()).getSupportActionBar().setDisplayShowTitleEnabled(false);

setHasOptionsMenu(true);

}

}

## 3、FragmentPageAdapter和FragmentPageStateAdapter的区别

FragmentPageAdapter在每次切换页面的的时候，是将Fragment进行分离，适合页面较少的Fragment使用以保存一些内存，对系统内存不会多大影响

FragmentPageStateAdapter在每次切换页面的时候，是将Fragment进行回收，适合页面较多的Fragment使用，这样就不会消耗更多的内存

## 4、Fragment生命周期

onAttach()

onCreate()

onCreateView()

onActivityCreated()

onStart()

onResume()

onPause()

onStop()

onDestroyView()

onDestroy()

onDetach()

## 5、Fragment的通信

Fragment调用Activity中的方法：getActivity

Activity调用Fragment中的方法：接口回调

Fragment调用Fragment中的方法：FragmentManager.findFragmentById

## 6、Fragment的replace、add、remove方法

replace：替代Fragment的栈顶页面

add：添加Fragment到栈顶页面

remove：移除Fragment栈顶页面

## 7、FragmentTransaction的show、hide方法

**show：**

显示先前隐藏的fragment。这只与视图已经添加到容器中的fragments相关，因为这将导致视图显示。

*/\*\*  
 \* Shows a previously hidden fragment. This is only relevant for fragments whose  
 \* views have been added to a container, as this will cause the view to  
 \* be shown.  
 \*   
 \** ***@param*** *fragment The fragment to be shown.  
 \*   
 \** ***@return*** *Returns the same FragmentTransaction instance.  
 \*/*public abstract FragmentTransaction show(Fragment fragment);

**hide：**

隐藏了一个现有的fragment。这只与视图已经添加到容器中的fragments相关，因为这将导致视图被隐藏。

*/\*\*  
 \* Hides an existing fragment. This is only relevant for fragments whose  
 \* views have been added to a container, as this will cause the view to  
 \* be hidden.  
 \*   
 \** ***@param*** *fragment The fragment to be hidden.  
 \*   
 \** ***@return*** *Returns the same FragmentTransaction instance.  
 \*/*public abstract FragmentTransaction hide(Fragment fragment);

add两个Fragment 其生命周期如下：

1. I/Fragment\_A(27457):onAttach
2. I/Fragment\_A(27457):onCreate
3. I/Fragment\_A(27457):onCreateView
4. I/Fragment\_A(27457):onActivityCreated
5. I/Fragment\_A(27457):onAttach
6. I/Fragment\_A(27457):onCreate
7. I/Fragment\_A(27457):onCreateView
8. I/Fragment\_A(27457):onActivityCreated
9. I/Fragment\_A(27457):onStart
10. I/Fragment\_A(27457):onStart

不会调用 onHiddenChanged

如果添加完成然后再两个Fragment交替show 和 hide ,那么只会调用 onHiddenChanged, 其中返回其是否显示的状态

**销毁过程：**

1. I/Fragment\_A(27457):onPause
2. I/Fragment\_A(27457):onPause
3. I/Fragment\_A(27457):onStop
4. I/Fragment\_A(27457):onStop
5. I/Fragment\_A(27457):onDestroyView
6. I/Fragment\_A(27457):onDestroy
7. I/Fragment\_A(27457):onDetach
8. I/Fragment\_A(27457):onDestroyView
9. I/Fragment\_A(27457):onDestroy
10. I/Fragment\_A(27457):onDetach

在销毁过程中, 交替指向 onPasue 和 onStop , 然后 一个一个执行 onDestroyView 到 onDetach 过程

小结：如果不调用hide方法，show方法是不会执行的

# 3、Service

## 1、Service是什么

Service是四大组件之一，它可以在后台执行长时间运行操作而没有用户界面的应用组件

## 2、Service和Thread的区别

Service是安卓中系统的组件，它运行在独立进程的主线程中，不可以执行耗时操作。Thread是程序执行的最小单元，分配CPU的基本单位，可以开启子线程执行耗时操作

Service在不同Activity中可以获取自身实例，可以方便的对Service进行操作。Thread在不同的Activity中难以获取自身实例，如果Activity被销毁，Thread实例就很难再获取得到

## 3、Service启动方式

startService

bindService

## 4、Service生命周期

startService

onCreate()

onStartCommand()

onDestroy()

bindService

onCreate()

onBind()

onUnbind()

onDestroy()

# 4、Broadcast Receiver

## 1、Broadcast Receiver是什么

Broadcast是四大组件之一，是一种广泛运用在应用程序之间传输信息的机制，通过发送Intent来传送我们的数据

## 2、Broadcast Receiver的使用场景

同一App具有多个进程的不同组件之间的消息通信

不同App之间的组件之间的消息通信

## 3、Broadcast Receiver的种类

普通广播

有序广播

本地广播

Sticky广播

## 4、Broadcast Receiver的实现

静态注册：注册后一直运行，尽管Activity、进程、App被杀死还是可以接收到广播

动态注册：跟随Activity的生命周期

## 5、Broadcast Receiver实现机制

自定义广播类继承BroadcastReceiver，复写onReceiver()

通过Binder机制向AMS进行注册广播

广播发送者通过Binder机制向AMS发送广播

AMS查找符合相应条件的广播发送到BroadcastReceiver相应的循环队列中

消息队列执行拿到广播，回调BroadcastReceiver的onReceiver()

## 6、LocalBroadcastManager特点

本地广播只能在自身App内传播，不必担心泄漏隐私数据

本地广播不允许其他App对你的App发送该广播，不必担心安全漏洞被利用

本地广播比全局广播更高效

以上三点都是源于其内部是用Handler实现的

# 5、WebView

## 1、WebView安全漏洞

API16之前存在远程代码执行安全漏洞，该漏洞源于程序没有正确限制使用WebView.addJavascriptInterface方法，远程攻击者可通过使用Java反射机制利用该漏洞执行任意Java对象的方法

## 2、WebView销毁步骤

WebView在其他容器上时（如：LinearLayout），当销毁Activity时，需要在onDestroy()中先移除容器上的WebView，然后再将WebView.destroy()，这样就不会导致内存泄漏

## 3、WebView的jsbridge

客户端和服务端之间可以通过Javascript来互相调用各自的方法

## 4、WebViewClient的onPageFinished

WebViewClient的onPageFinished在每次完成页面的时候调用，但是遇到未加载完成的页面跳转其他页面时，就会一直调用，使用WebChromeClient.onProgressChanged可以替代

## 5、WebView后台耗电

在WebView加载页面的时候，会自动开启线程去加载，如果不很好的关闭这些线程，就会导致电量消耗加大，可以采用暴力的方法，直接在onDestroy方法中System.exit(0)结束当前正在运行中的java虚拟机

## 6、WebView硬件加速

Android3.0引入硬件加速，默认会开启，WebView在硬件加速的情况下滑动更加平滑，性能更加好，但是会出现白块或者页面闪烁的副作用，建议WebView暂时关闭硬件加速

## 7、WebView内存泄漏

由于WebView是依附于Activity的，Activity的生命周期和WebView启动的线程的生命周期是不一致的，这会导致WebView一直持有对这个Activity的引用而无法释放，解决方案如下

独立进程，简单暴力，不过可能涉及到进程间通信（推荐）

动态添加WebView，对传入WebView中使用的Context使用弱引用

# 6、Binder

## 1、Linux内核的基本知识

进程隔离/虚拟地址空间：进程间是不可以共享数据的，相当于被隔离，每个进程被分配到不同的虚拟地址中

系统调用：Linux内核对应用有访问权限，用户只能在应用层通过系统调用，调用内核的某些程序

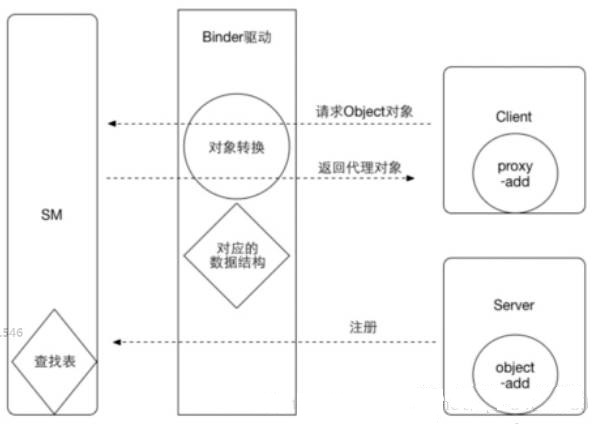
binder驱动：它负责各个用户的进程，通过binder通信内核来进行交互的模块

## 2、为什么使用Binder

性能上，相比传统的Socket更加高效

安全性高，支持协议双方互相校验

## 3、Binder通信模型



Service服务端通过Binder驱动在ServiceManager的查找表中注册Object对象的add方法

Client客户端通过Binder驱动在ServiceManager的查找表中找到Object对象的add方法，并返回proxy的add方法，add方法是个空实现，proxy也不是真正的Object对象，是通过Binder驱动封装好的代理类的add方法

当Client客户端调用add方法时，Client客户端通过Binder驱动将proxy的add方法，请求ServiceManager来找到Service服务端真正对象的add方法，进行调用

## 4、AIDL

客户端通过aidl文件的Stub.asInterface()方法，拿到Proxy代理类

通过调用Proxy代理类的方法，将参数进行封包后，调用底层的transact()方法

transact()方法会回调onTransact()方法，进行参数的解封

在onTransact()方法中调用服务端对应的方法，并将结果返回

# 7、Handler

## 1、Handler是什么

Handler通过发送和处理Message和Runnable对象来关联相对应线程的MessageQueue

## 2、Handler使用方法

post(runnable)

sendMessage(message)

## 3、Handler工作原理

Android进阶——Android消息机制之Looper、Handler、MessageQueen

<http://blog.csdn.net/qq_30379689/article/details/53394061>

## 4、Handler引起的内存泄漏

原因：非静态内部类持有外部类的匿名引用，导致Activity无法释放

解决：

Handler内部持有外部Activity的弱引用

Handler改为静态内部类

Handler.removeCallback()

# 8、AsyncTask

## 1、AsyncTask是什么

它本质上就是一个封装了线程池和Handler的异步框架

## 2、AsyncTask使用方法

三个参数

Params：表示后台任务执行时的参数类型，该参数会传给AysncTask的doInBackground()方法

Progress：表示后台任务的执行进度的参数类型，该参数会作为onProgressUpdate()方法的参数

Result：表示后台任务的返回结果的参数类型，该参数会作为onPostExecute()方法的参数

五个方法

onPreExecute()：异步任务开启之前回调，在主线程中执行

doInBackground()：执行异步任务，在线程池中执行

onProgressUpdate()：当doInBackground中调用publishProgress时回调，在主线程中执行

onPostExecute()：在异步任务执行之后回调，在主线程中执行

onCancelled()：在异步任务被取消时回调

## 3、AsyncTask工作原理

Android进阶——多线程系列之异步任务AsyncTask的使用与源码分析

<http://blog.csdn.net/qq_30379689/article/details/53203556>

## 4、AsyncTask引起的内存泄漏

原因：非静态内部类持有外部类的匿名引用，导致Activity无法释放

解决：

AsyncTask内部持有外部Activity的弱引用

AsyncTask改为静态内部类

AsyncTask.cancel()

## 5、AsyncTask生命周期

在Activity销毁之前，取消AsyncTask的运行，以此来保证程序的稳定

## 6、AsyncTask结果丢失

由于屏幕旋转、Activity在内存紧张时被回收等情况下，Activity会被重新创建，此时，旧的AsyncTask持有旧的Activity引用，这个时候会导致AsyncTask的onPostExecute()对UI更新无效

## 7、AsyncTask并行or串行

AsyncTask在Android 2.3之前默认采用并行执行任务，AsyncTask在Android 2.3之后默认采用串行执行任务

如果需要在Android 2.3之后采用并行执行任务，可以调用AsyncTask的executeOnExecutor()

# 9、HandlerThread

## 1、HandlerThread产生背景

当系统有多个耗时任务需要执行时，每个任务都会开启一个新线程去执行耗时任务，这样会导致系统多次创建和销毁线程，从而影响性能。为了解决这一问题，Google提供了HandlerThread，HandlerThread是在线程中创建一个Looper循环器，让Looper轮询消息队列，当有耗时任务进入队列时，则不需要开启新线程，在原有的线程中执行耗时任务即可，否则线程阻塞

## 2、HanlderThread的特点、

HandlerThread本质上是一个线程，继承自Thread

HandlerThread有自己的Looper对象，可以进行Looper循环，可以创建Handler

HandlerThread可以在Handler的handlerMessage中执行异步方法

HandlerThread优点是异步不会堵塞，减少对性能的消耗

HandlerThread缺点是不能同时继续进行多任务处理，需要等待进行处理，处理效率较低

HandlerThread与线程池不同，HandlerThread是一个串行队列，背后只有一个线程。

# 10、IntentService

## 1、IntentService是什么

IntentService是继承自Service并处理异步请求的一个类，其内部采用HandlerThread和Handler实现的，在IntentService内有一个工作线程来处理耗时操作，其优先级比普通Service高。当任务完成后，IntentService会自动停止，而不需要手动调用stopSelf()。另外，可以多次启动IntentService，每个耗时操作都会以工作队列的方式在IntentService中onHandlerIntent()回调方法中执行，并且每次只会执行一个工作线程

## 2、IntentService使用方法

创建Service继承自IntentService

覆写构造方法和onHandlerIntent()方法

在onHandlerIntent()中执行耗时操作

# 11、View绘画机制+常见的View

## 11.1自定义View

### 1、三大重要方法

### 2、事件分发机制

### 3、onScroll

### 4、View中的动画（ValueAnimation）

### 5、项目实践总结

inflate(int resource, ViewGroup root)

inflate(int resource, ViewGroup root, boolean attachToRoot)

两个参数：调用三个参数---> root不为空attachToRoot为true； root为空attachToRoot为false

参数详解：

resource：需要加载布局文件的id（xml布局的名称）

root：需要附加到resource资源文件的根控件

inflate()返回一个View对象，

如果第三个参数attachToRoot为true，就将这个root作为根对象返回，为false，将这个root对象的LayoutParams属性附加到resource对象的根布局对象上，也就是布局文件resource的最外层的View上，比如是一个LinearLayout或者其它的Layout对象。

attachToRoot：是否将root附加到布局文件的根视图上

我们在ListView的Adapter的getView方法里面经常会调用两个参数的inflate方法，

Java代码 收藏代码

mInflater.inflate(R.layout.adv\_viewpager, null)

我们可能会发现layout外层的layout\_width layout\_height属性都没起作用，全都变成wrap\_content的值了。

这个问题就是因为我们使用了错误的参数照成的， 系统在inflate layout的时候 如果传入的root为空的话 就会忽略LayoutParams.

所以在getView里面应该调用

Java代码 收藏代码

mInflater.inflate(R.layout.adv\_viewpager, root, false)

这样在R.layout.adv\_viewpager layout xml文件里面所有关于Layout的属性就都能生效。

那么attachToRoot是什么作用呢？ 他会自动把layout加到View hierarchy中，不需要手动调用root.addView，当然在Adapter.getView里面不用手动调用root.addView是因为Adapter已经帮我们做了，所以如果我们在Adapter.getView里面传入attachToRoot为true的话，就会报错，因为一个view不能attach两次。

Android进阶——Android视图工作机制之measure、layout、draw

<http://blog.csdn.net/qq_30379689/article/details/54588736>

Android事件分发机制之dispatchTouchEvent、onInterceptTouchEvent、onTouchEvent

http://blog.csdn.net/qq\_30379689/article/details/53967177

## 11.2、TabLayout

### 11.2.1、简介

TabLayout提供了一个水平的布局用来展示Tabs

样式：



使用：

TabLayout的基本使用方式

1.在xml中加入控件：

<android.support.design.widget.TabLayout

android:id="@+id/tabLayout"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

2.在代码中

tabLayout= (TabLayout) findViewById(R.id.tabLayout);

tabLayout.addTab(tabLayout.newTab().setText("Tab 1"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 2"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 3"));

3.效果



下图是简书Android端的主页Tab,在其他的App中Tab也是很常见的，它的实现方式也有很多：TabHost，自定义控件（第三方库），RadioGroup等等。这里主要介绍Android Design库中的TabLayout的使用。

#### 1.什么是TabLayout

在源码中给出了TabLayout的定义：

TabLayout provides a horizontal layout to display tabs.

意思很明显：TabLayout提供了一个水平的布局用来展示Tabs。

Design库：AS有直接的引用，如果是Eclipse这里提供一个通道-[Design库](http://download.csdn.net/detail/android_yh/9525604" \t "_blank)

#### 特别说明：

Caused by: java.lang.IllegalArgumentException: You need to use a Theme.AppCompat theme (or descendant) with the design library.

在清单文件中设置如下代码即可：

android:theme="@style/Theme.AppCompat"

#### 2.TabLayout的基本使用方式

方式一：

1.在布局中加入该控件：

<android.support.design.widget.TabLayout

android:id="@+id/tabLayout"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

2.在代码中

tabLayout= (TabLayout) findViewById(R.id.tabLayout);

tabLayout.addTab(tabLayout.newTab().setText("Tab 1"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 2"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 3"));

3.显示效果

[](11.2.1.1.gif)

感觉还不错吧，挺简单就实现了这个UI效果。

方式二：

<android.support.design.widget.TabLayout

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content">

<android.support.design.widget.TabItem

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tab1"/>

.......

</android.support.design.widget.TabLayout>

这样同样也可以实现方式一的效果，可是单单这样并不能满足我们。于是接下来看看有什么属性可以使用。

#### 3.改变下TabLayout的颜色

上面的Tab颜色感觉不好看，打算换换：

1.改变选中字体的颜色

app:tabSelectedTextColor="@android:color/holo\_orange\_light"

2.改变未选中字体的颜色

app:tabTextColor="@color/colorPrimary"

3.改变指示器下标的颜色

app:tabIndicatorColor="@android:color/holo\_orange\_light"

4.改变整个TabLayout的颜色

app:tabBackground="color"

于是现在我的Tab变成了这个样子：



#### 4.改变TabLayout内部字体大小

总觉得这个字体有点小了，于是想找方法把这个字变得大一点，

好像没有直接变大的方法，可是找到了这个：

app:tabTextAppearance="@android:style/TextAppearance.Holo.Large"//设置文字的外貌

效果：



#### 5.改变指示器下标的高度

既然字体变大了，指示器太小就显得不太好看了，

设置指示器下标的高度：

app:tabIndicatorHeight="4dp"

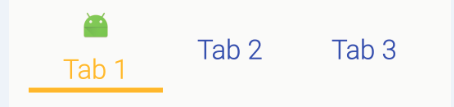
效果：



#### 6.添加图标

有时候Tab只有文字感觉有点单调了：

tabLayout.addTab(tabLayout.newTab().setText("Tab1").setIcon(R.mipmap.ic\_launcher));



#### 7.Tab的模式

数据很多的时候我们应该怎么办呢，简书中的第二个Tab就是可以滑动的：

我们先多加几个tab：

tabLayout.addTab(tabLayout.newTab().setText("Tab 4"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 5"));

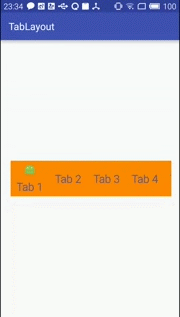
tabLayout.addTab(tabLayout.newTab().setText("Tab 6"));

tabLayout.addTab(tabLayout.newTab().setText("Tab 7"));

然后设置属性为：app:tabMode="scrollable"

默认是fixed：固定的，标签很多时候会被挤压，不能滑动。

效果如下：



#### 8.加入Padding

设置Tab内部的子控件的Padding：

app:tabPadding="xxdp"

app:tabPaddingTop="xxdp"

app:tabPaddingStart="xxdp"

app:tabPaddingEnd="xxdp"

app:tabPaddingBottom="xxdp"

设置整个TabLayout的Padding：

app:paddingEnd="xxdp"

app:paddingStart="xxdp"

#### 9.内容的显示模式

app:tabGravity="center"//居中，如果是fill，则是充满

#### 10.Tab的宽度限制

设置最大的tab宽度：

app:tabMaxWidth="xxdp"

设置最小的tab宽度：

app:tabMinWidth="xxdp"

#### 11.Tab的“Margin”

TabLayout开始位置的偏移量：

app:tabContentStart="100dp"

#### 12.TabLayout的监听事件

选中了某个tab的监听事件OnTabSelectedListener()：

tabLayout.setOnTabSelectedListener(newTabLayout.OnTabSelectedListener() {

@Override

public voidonTabSelected(TabLayout.Tab tab) {

//选中了tab的逻辑

}

@Override

public voidonTabUnselected(TabLayout.Tab tab) {

//未选中tab的逻辑

}

@Override

public voidonTabReselected(TabLayout.Tab tab) {

//再次选中tab的逻辑

}

});

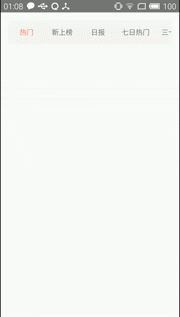
#### 13.和ViewPager的联动

最后也是最重要的：

tabLayout.setupWithViewPager(Viewpager);

一行代码和ViewPager联动起来，简单粗暴。

#### 14.简单模仿简书Android端的Tab



主要就是设置下标的高度为0，相当于没有下标。

app:tabIndicatorHeight="0dp"

然后设置背景颜色以及选中文字颜色

app:tabSelectedTextColor="#ff7a61"

app:tabBackground="#f6f4f2"//这里不能直接写RGB，需要@color/xx

最后设置Tab的模式：

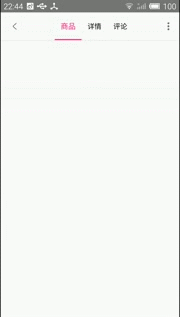
app:tabMode="scrollable"

当然Tablayout还可以做很多的样式，如果有关于Tablayout的注意事项或者相关知识的文章，希望大家可以分享给我，多谢！

#### 15.仿京东商品详情Android端的Tab

主要原理是setCustomView()加载自定义视图，来实现字体大小的改变，默认的TabLayout不能改变。

[Design库-TabLayout仿京东商品详情Tab](http://www.jianshu.com/p/116f87abdaea)



#### 补充：默认选中某项

tablayout.getTabAt(position).select();

标签1-15来自于：<http://www.jianshu.com/p/2b2bb6be83a8>

#### **16.项目中使用（自定义tabIndicator的样式）**

（想偷懒，直接复制代码）

**XML文件：**

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

<LinearLayout

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

xmlns:tools="http://schemas.android.com/tools"

android:id="@+id/activity\_main"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

tools:context="com.zht.tablayoutdemo.MainActivity">

<android.support.design.widget.TabLayout

xmlns:app="http://schemas.android.com/apk/res-auto"

android:id="@+id/main\_tablayout"

android:layout\_width="match\_parent"

android:layout\_height="80dp"

android:background="@android:color/white"

android:paddingBottom="5dp"

android:paddingTop="10dp"

app:tabIndicatorHeight="1dp"

app:tabIndicatorColor="@android:color/holo\_orange\_dark"

app:tabBackground="@android:color/white"/>

<android.support.v4.view.ViewPager

android:id="@+id/main\_viewpager"

android:layout\_width="fill\_parent"

android:layout\_height="0dp"

android:layout\_weight="1"/>

</LinearLayout>

**Activity代码：**

public class MainActivity extends AppCompatActivity {

private final static String TAG = "MainActivity";

private TabLayout mTabLayout;

private ViewPager mViewpager;

private TabFragmentPagerAdapter adapter;

private List<Fragment> fragments;

private String[] mTabItemNameArray;

private int[] mTabItemIconArray;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

initView();//初始化试图

addFragment();//初始化Fragmrnt，并添加到ViewPager中。

initViewPager();

}

/\*\*

\* 初始化试图和获取资源文件

\*/

private void initView() {

mTabLayout = (TabLayout) findViewById(R.id.main\_tablayout);

mViewpager = (ViewPager) findViewById(R.id.main\_viewpager);

mTabItemNameArray = getResources().getStringArray(R.array.main\_tab\_item);

mTabItemIconArray = new int[]{

R.drawable.ic\_tab\_selector,

R.drawable.ic\_tab1\_selector,

R.drawable.ic\_tab2\_selector,

R.drawable.ic\_tab3\_selector

};

}

/\*\*

\* 在这里添加Fragment

\*/

private void addFragment() {

if (mTabLayout == null || mViewpager == null) {

return;

}

fragments = new ArrayList<>();

fragments.add(new HomeFragment());

fragments.add(new BusinessFragment());

fragments.add(new DepartmentFragment());

fragments.add(new PersonFragment());

}

/\*\*

\* 在这里初始化ViewPager

\*/

private void initViewPager() {

if (fragments == null) {

return;

}

adapter = new TabFragmentPagerAdapter(getSupportFragmentManager(), fragments, mTabItemNameArray);

mViewpager.setAdapter(adapter);

mTabLayout.setupWithViewPager(mViewpager);//将tablayout和viewpager绑定

//绘制自定义的Tab布局

setupTabIcons();

}

/\*\*

\* 设置条目的布局

\*/

private void setupTabIcons() {

if (mTabLayout == null || mTabLayout.getTabCount() <= 0) {

return;

}

for (int i = 0; i < mTabLayout.getTabCount(); i++) {

mTabLayout.getTabAt(i).setCustomView(getTabView(i));

}

}

/\*\*

\* 创建tab对应的视图

\* @param position

\* @return

\*/

public View getTabView(int position) {

View view = LayoutInflater.from(this).inflate(R.layout.item\_tab, null);

TextView txt\_title = (TextView) view.findViewById(R.id.tab\_text);

//设置文字（文字的颜色只需要在xml里面设置一次）

txt\_title.setText(mTabItemNameArray[position]);

//设置图片背景（只需要设置一次）

ImageView img\_title = (ImageView) view.findViewById(R.id.tab\_icon);

img\_title.setImageResource(mTabItemIconArray[position]);

return view;

}

}

**ViewPager的Adapter：**

public class TabFragmentPagerAdapter extends FragmentPagerAdapter {

/\*\*

\* fragment集合

\*/

private List<Fragment> mFragmentList = null;

/\*\*

\* 标题集合

\*/

private String[] titles;

public TabFragmentPagerAdapter(FragmentManager mFragmentManager,

ArrayList<Fragment> fragmentList) {

super(mFragmentManager);

mFragmentList = fragmentList;

}

/\*\*

\* titles是给TabLayout设置title用的

\*

\* @param mFragmentManager

\* @param fragmentList

\* @param titles

\*/

public TabFragmentPagerAdapter(FragmentManager mFragmentManager,

List<Fragment> fragmentList, String[] titles) {

super(mFragmentManager);

mFragmentList = fragmentList;

this.titles = titles;

}

/\*\*

\* 获取数量

\*

\* @return the count

\* @see android.support.v4.view.PagerAdapter#getCount()

\*/

@Override

public int getCount() {

return mFragmentList.size();

}

/\*\*

\* 获取索引位置的Fragment.

\*

\* @param position the position

\* @return the item

\* @see android.support.v4.app.FragmentPagerAdapter#getItem(int)

\*/

@Override

public Fragment getItem(int position) {

Fragment fragment = null;

if (position < mFragmentList.size()) {

fragment = mFragmentList.get(position);

} else {

fragment = mFragmentList.get(0);

}

return fragment;

}

/\*\*

\* 将标题设置到相对应的位置

\* @param position

\* @return

\*/

@Override

public CharSequence getPageTitle(int position) {

if (titles != null && titles.length > 0)

return titles[position];

return null;

}

}

**自定义的tab(item\_tab.xml)：**

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

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

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:gravity="center"

android:orientation="vertical">

<ImageView

android:id="@+id/tab\_icon"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"/>

<TextView

android:id="@+id/tab\_text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:textColor="@drawable/ic\_tab\_text\_selector"

android:text="首页"/>

</LinearLayout>

**图片资源+选择器（自己定义）模板：**

ic\_tab\_selector.xml代码：

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

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

<item android:drawable="@drawable/ic\_tab\_normal" android:state\_selected="false"/>

<item android:drawable="@drawable/ic\_tab\_selected" android:state\_selected="true"/>

</selector>

**ic\_tab\_text\_selector.xml代码：**

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

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

<item android:color="@color/BottomTabTextSelect" android:state\_focused="true" />

<item android:color="@color/BottomTabTextSelect" android:state\_pressed="true" />

<item android:color="@color/BottomTabTextSelect" android:state\_selected="true" />

<item android:color="@color/BottomTabTextSelect" android:state\_checked="true" />

<item android:color="@color/BottomTabTextNormal" />

</selector>

**底部导航栏文字的颜色：**

<color name="BottomTabTextSelect">#F5af66</color>

<color name="BottomTabTextNormal">#808080</color>

## 11.3、EditText

### 11.3.1、屏幕自适应

可以解决：

a、一进入界面就出现输入框

b、输入框遮挡EditText

代码：

<activity android:name=".XXXActivity"

android:windowSoftInputMode="adjustPan"/>

### 11.3.2、修改下划线（附带光标）的颜色

1、activity继承android.support.v7.app.AppCompatActivity

2、 在styles.xml文件中声明自定义样式

<style name="EditTextStyle" parent="Theme.AppCompat.Light">

<!--失去焦点的时候-->

<item name="colorControlNormal">@color/transparent</item>

<!--获取焦点的时候-->

<item name="colorControlActivated">@color/dialog\_button\_ensure</item>

</style>

3、通过android：theme属性将此样式应用于您的EditText

<EditText

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="XXXXXXXX"

android:theme="@style/EditTextStyle"/>

### 11.3.3、修改光标的颜色

在xml布局文件中

android:textCursorDrawable=”@null” 表示光标的颜色和字体的颜色一样，也可以不设置。

自定义光标的颜色：

1、drawable文件夹下写个edit\_cursor\_color.xml文件

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

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

android:shape="rectangle" >

<size android:width="2dp" />

<solid android:color="#ff7200" />

</shape>

2、在EditText中设置

android:textCursorDrawable=”@drawable/edit\_cursor\_color”

### 11.3.4、去掉下划线

在EditText中设置属性android:background为null。

android:background="@null"

在设置过EditText的背景，下滑线就会消失。

### 11.3.5、监听

1、setOnKeyListener(),监听硬键盘事件

代码：

edittext.setOnKeyListener(new View.OnKeyListener() {

@Override

public boolean onKey(View v, int keyCode, KeyEvent event) {

textview.setText(edittext.getText());

return false;

}

});

2.使用TextWatcher类，可以监听软键盘和硬键盘的

代码：

edittext.addTextChangedListener(new TextWatcher() {

/\*\*

\*

\* @param s 输入框中改变前的字符串信息

\* @param start 输入框中改变前的字符串的起始位置

\* @param count 输入框中改变前后的字符串改变数量一般为0

\* @param after 输入框中改变后的字符串与起始位置的偏移量

\*/

@Override

public void beforeTextChanged(CharSequence s, int start, int count, int after) {

}

/\*\*

\*

\* @param s 输入框中改变后的字符串信息

\* @param start 输入框中改变后的字符串的起始位置

\* @param before 输入框中改变前的字符串的位置 默认为0

\* @param count 输入框中改变后的一共输入字符串的数量

\*/

@Override

public void onTextChanged(CharSequence s, int start, int before, int count) {

}

/\*\*

\*

\* @param s 输入结束呈现在输入框中的信息

\*/

@Override

public void afterTextChanged(Editable s) {

} });

3、实现当输入到最大值时则不允许再输入了，另外，由于EditText并没有提供给我们EditText的禁止输入功能，以下方法也实现了此功能。

代码：

private void setEditable(EditText mEdit, int maxLength, boolean value) {

if (value) {

mEdit.setFilters(new InputFilter[] { new MyEditFilter(maxLength) });

mEdit.setCursorVisible(true);

mEdit.setFocusableInTouchMode(true);

mEdit.requestFocus();

} else {

mEdit.setFilters(new InputFilter[] { new InputFilter() {

@Override

public CharSequence filter(CharSequence source, int start,

int end, Spanned dest, int dstart, int dend) {

return source.length() < 1 ? dest.subSequence(dstart, dend)

: "";

}

} });

mEdit.setCursorVisible(false);

mEdit.setFocusableInTouchMode(false);

mEdit.clearFocus();

}

}

4、EditText焦点的监听

edit.setOnFocusChangeListener(new OnFocusChangeListener() {

@Override

public void onFocusChange(View v, boolean hasFocus) {

// 当hasFocus = true 是获取焦点

if (!hasFocus) {

}

}

});

## 11.4、CardView

在gradle文件下添加依赖库：

compile 'com.android.support:cardview-v7:25.3.1'

**基本属性：**

|  |  |
| --- | --- |
| 属性 | 说明 |
| cardElevation | 阴影的大小 |
| cardMaxElevation | 阴影最大高度 |
| cardBackgroundColor | 卡片的背景色 |
| cardCornerRadius | 卡片的圆角大小 |
| contentPadding | 卡片内容于边距的间隔 |
| contentPaddingBottom | 卡片内容与底部的边距 |
| contentPaddingTop | 卡片内容与顶部的边距 |
| contentPaddingLeft | 卡片内容与左边的边距 |
| contentPaddingRight | 卡片内容与右边的边距 |
| contentPaddingStart | 卡片内容于边距的间隔起始 |
| contentPaddingEnd | 卡片内容于边距的间隔终止 |
| cardUseCompatPadding | 设置内边距，V21+的版本和之前的版本具有一样的计算方式 |
| cardPreventCornerOverlap | 在V20和之前的版本中添加内边距，为了防止内容和边角重叠 |

**CardView是在布局中使用的：**

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

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

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_marginBottom="10dp"

android:background="@color/gray">

<android.support.v7.widget.CardView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layout\_marginLeft="10dp"

android:layout\_marginRight="10dp"

app:cardBackgroundColor="@color/blue"

app:cardCornerRadius="16dp"

app:cardElevation="16dp">

<TextView

android:id="@+id/id\_num"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center"

android:layout\_margin="10dp"

android:gravity="center"

android:textColor="@color/white"

android:textSize="20sp" />

</android.support.v7.widget.CardView>

</FrameLayout>

## 11.5底部导航栏的实现

1、使用LinearLayout + TextView实现了底部导航栏的效果

2、使用RadioGroup + RadioButton实现了底部导航栏的效果

3、利用BottomNavigationBar实现底部导航栏

符合MD设计的,希望自己的APP炫酷一点的.毫无疑问都应该用BottomNavigation规范的控件。

文章推荐：

<http://blog.csdn.net/u010046908/article/details/50962081>

4、复杂的底部导航栏

<http://blog.csdn.net/jxq1994/article/details/52573506>

5、ViewPager + List<View> + PagerAdapter

创建就会一次性加载完四个页面.适合简单的页面,比如app一开始的导航页.

6、ViewPager + List<Fragment> + FragmentPagerAdapter或FragmentStatePagerAdapter

给力的方法.适合能左右滑动的页面.可以自己定制缓存策略.配合神方法,也能一开始就加载全部页面.

7、用fragmentTransaction的show和hide方法隐藏和显示Fragment

用的才加载. 一旦加载就不删除. 切换只用hide和show,速度飞快. 当然你也可以自己定制适合自己的缓存策略.

8、FragmentTabHost

简单快捷.代码少.但是切换速度理论不够三快.

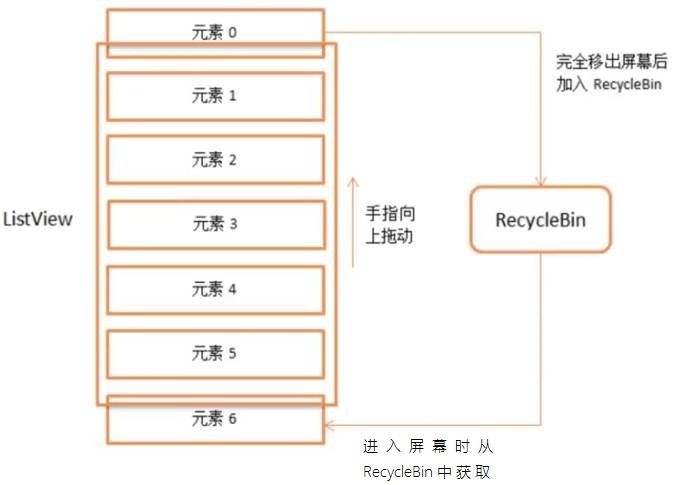
<http://blog.csdn.net/alcoholdi/article/details/51594061>

# 12、ListView和RecyclerView

## 1、ListView是什么

ListView是能将一个数据集合以动态滚动的方式展示到用户界面上的View

## 2、ListView的RecycleBin机制



## 3、ListView的优化

重用convertView

使用ViewHolder

图片三级缓存

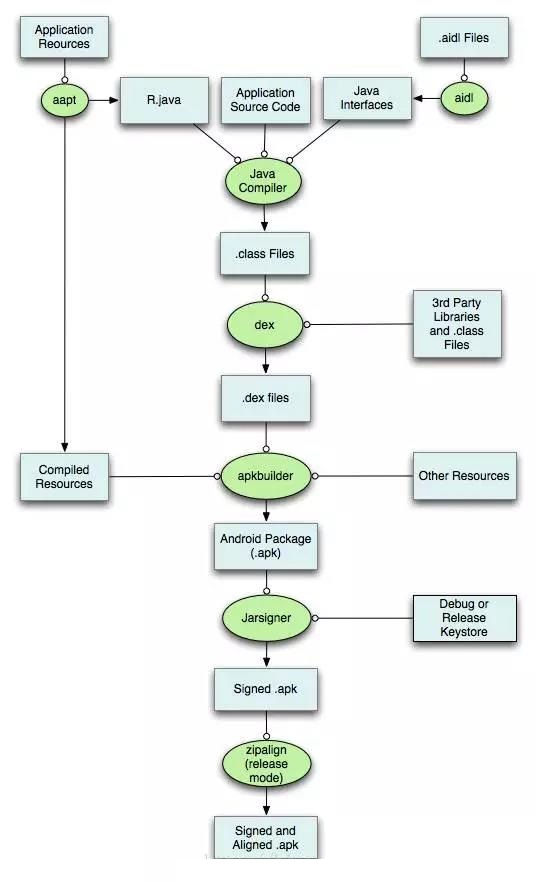
监听滑动事件

少用透明View

开启硬件加速

# 13、Android项目构建

## 1、android构建流程



## 2、jenkins持续集成构建

这里可参考蒲公英文档

http://www.pgyer.com/doc/view/jenkins

## 3、Git常用命令

git init：仓库的初始化

git status：查看当前仓库的状态

git diff：查看仓库与上次修改的内容

git add：将文件放进暂存区

git commit：提交代码

git clone：克隆代码

git bransh：查看当前分支

git checkout：切换当前分支

## 4、git工作流

fork/clone（主流）

fork：将别人的仓库代码fork到自己的仓库上

clone：克隆下自己仓库的代码

update、commit：修改代码并提交到自己的仓库

push：提交到自己的仓库

pull request：请求添加到别人的仓库

clone

## 5、proguard是什么

ProGuard工具是用于压缩、优化和混淆我们的代码，其主作用是移除或混淆代码中无用类、字段、方法和属性

## 6、proguard技术功能

压缩

优化

混淆

预检测

## 7、proguard工作原理

将无用的字段或方法存入到EntryPoint中，将非EntryPoint的字段和方法进行替换

## 8、为什么要混淆

由于Java是一门跨平台的解释性语言，其源代码被编译成class字节码来适应其他平台，而class文件包含了Java源代码信息，很容易被反编译

# 14、ANR

## 1、什么是ANR

Application Not Responding，页面无响应的对话框

## 2、发生ANR的条件

应用程序的响应性是由ActivityManager和WindowManager系统服务监视的，当ANR发生条件满足时，就会弹出ANR的对话框

Activity超过5秒无响应

BroadcastReceiver超过10秒无响应

Service超过20秒无响应

## 3、造成ANR的主要原因

主线程被IO操作阻塞

Activity的所有生命周期回调都是执行在主线程的

Service默认执行在主线程中

BoardcastReceiver的回调onReceive()执行在主线程中

AsyncTask的回调除了doInBackground，其他都是在主线程中

没有使用子线程Looper的Handler的handlerMessage，post(Runnable)都是执行在主线程中

## 4、如何解决ANR

使用AsyncTask处理耗时IO操作

使用Thread或HandlerThread提供优先级

使用Handler处理工作线程的耗时操作

Activity的onCreate和onResume回调尽量避免耗时操作

# 15、OOM

## 1、什么是OOM

OOM指Out of memory（内存溢出），当前占用内存加上我们申请的内存资源超过了Dalvik虚拟机的最大内存限制就会抛出Out of memory异常

## 2、OOM相关概念

内存溢出：指程序在申请内存时，没有足够的空间供其使用

内存泄漏：指程序分配出去的内存不再使用，无法进行回收

内存抖动：指程序短时间内大量创建对象，然后回收的现象

## 3、解决OOM

Bitmap相关

图片压缩（见下一章节Bimap中的压缩）

加载缩略图

在滚动时不加载图片

回收Bitmap

使用inBitmap属性

捕获异常

其他相关

listview重用convertView、使用lru

避免onDraw方法执行对象的创建

谨慎使用多进程

# 16、Bitmap

位图文件（Bitmap），扩展名可以是.bmp或者.dib。位图是Windows标准格式图形文件，它将图像定义为由点（像素）组成，每个点可以由多种色彩表示，包括2、4、8、16、24和32位色彩。

例如，一幅1024×768分辨率的32位真彩图片，其所占存储字节数为：1024×768×32/(8\*1024)=3072KB

位图文件图像效果好，但是非压缩格式的，需要占用较大存储空间，不利于在网络上传送。jpg/png格式则恰好弥补了位图文件的缺点。

在Android中计算bitmap的大小：

bitmap.getByteCount()（返回byte）

扫盲：1M=1024KB=1024\*1024byte

一般1920X1080尺寸的图片在内存中的大小，1920x1080x4=2025kb=1.977539M

乘以4位的原因是在安卓系统中bitmap图片一般是以ARGB\_8888（ARGB分别代表的是透明度,红色,绿色,蓝色,每个值分别用8bit来记录,也就是一个像素会占用4byte,共32bit。）来进行存储的。

#### **Android中图片有四种颜色格式**

|  |  |  |
| --- | --- | --- |
| 颜色格式 | 每个像素占用内存（单位byte） | 每个像素占用内存（单位bit） |
| ALPHA\_8 | 1 | 8 |
| ARGB\_8888(默认) | 4 | 32 |
| ARGB\_4444 | 2 | 16 |
| RGB\_565 | 2 | 16 |

算法： 8+8+8+8 =32

1 bit 0/1；最小单位

1 byte = 8bit 10101010 （0-255）00000000 --- 11111111

ARGB\_8888：ARGB分别代表的是透明度,红色,绿色,蓝色,每个值分别用8bit来记录,也就是一个像素会占用4byte,共32bit.  
ARGB\_4444：ARGB的是每个值分别用4bit来记录,一个像素会占用2byte,共16bit.  
RGB\_565：R=5bit,G=6bit,B=5bit，不存在透明度,每个像素会占用2byte,共16bit.  
ALPHA\_8:该像素只保存透明度,会占用1byte,共8bit.  
在实际应用中而言,建议使用ARGB\_8888以及RGB\_565。

如果你不需要透明度,那么就选择RGB\_565,可以减少一半的内存占用.

## 1、recycle

在安卓3.0以后Bitmap是存放在堆中的，我们只要回收堆内存即可

在安卓3.0以前Bitmap是存放在内存中的，我们需要回收native层和Java层的内存

官方建议我们3.0以后使用recycle方法进行回收，该方法也可以不主动调用，因为垃圾回收器会自动收集不可用的Bitmap对象进行回收

recycle方法会判断Bitmap在不可用的情况下，将发送指令到垃圾回收器，让其回收native层和Java层的内存，则Bitmap进入dead状态

recycle方法是不可逆的，如果再次调用getPixels()等方法，则获取不到想要的结果

释放与此位图关联的本地对象，并清除对像素数据的引用。这不会同步释放像素数据; 它只是允许它被垃圾收集，如果没有其他的参考。位图被标记为“死”，意味着如果getPixels（）或setPixels（）被调用，它将抛出异常，并且不会画任何东西。这个操作是不能逆转的，所以只有在你确定没有进一步的位图使用时才能调用它。这是一个高级调用，通常不需要调用，因为当没有更多的引用到这个位图时，正常的GC进程将释放这个内存。

在Android3.0之前，Bitmap的内存分配分为两部分，一部分是分配在Dalvik的VM堆中，而像素数据的内存是分配在Native堆中，而到了Android3.0之后，Bitmap的内存则已经全部分配在VM堆上，这两种分配方式的区别在于，Native堆的内存不受Dalvik虚拟机的管理，我们想要释放Bitmap的内存，必须手动调用Recycle方法，而到了Android 3.0之后的平台，我们就可以将Bitmap的内存完全放心的交给虚拟机管理了，我们只需要保证Bitmap对象遵守虚拟机的GC Root Tracing的回收规则即可。

## 2、LruCache原理

LruCache是个泛型类，内部采用LinkedHashMap来实现缓存机制，它提供get方法和put方法来获取缓存和添加缓存，其最重要的方法trimToSize是用来移除最少使用的缓存和使用最久的缓存，并添加最新的缓存到队列中

## 3、计算inSampleSize

public static int calculateInSampleSize(BitmapFactory.Options options, int reqWidth, int reqHeight) {

final int width = options.outWidth;

final int height = options.outHeight;

int inSampleSize = 1;

if (width > reqWidth || height > reqHeight) {

if (width > height) {

inSampleSize = Math.round((float) height / (float) reqHeight);

} else {

inSampleSize = Math.round((float) width / (float) reqWidth);

}

}

return inSampleSize;

}

## 4、缩略图

public static Bitmap thumbnail(String path, int maxWidth, int maxHeight,boolean autoRotate) {

BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true;

Bitmap bitmap = BitmapFactory.decodeFile(path);

options.inJustDecodeBounds = false;

int sampleSize = calculateInSampleSize(options,maxWidth,maxHeight);

options.inSampleSize =sampleSize;

options.inPreferredConfig = Bitmap.Config.RGB\_565;

options.inPurgeable =true;

options.inInputShareable = true;

if(bitmap !=null&&!bitmap.isRecycled()){

bitmap.recycle();

}

bitmap = BitmapFactory.decodeFile(path,options);

return bitmap;

}

## 5、保存Bitmap

public static String save(Bitmap bitmap, Bitmap.CompressFormat format, int quality,File desFile) {

try{

FileOutputStream out = new FileOutputStream(desFile);

if(bitmap.compress(format,quality,out)){

out.flush();

out.close();

}

if(bitmap!=null&&!bitmap.isRecycled()){

bitmap.recycle();

}

return desFile.getAbsolutePath();

}catch (FileNotFoundException e){

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

## 6、保存到SD卡

public static String save(Bitmap bitmap, Bitmap.CompressFormat format, int quality,Context context) {

if(!Environment.getExternalStorageState()

.equals(Environment.MEDIA\_MOUNTED)){

return null;

}

File dir = new File(Environment.getExternalStorageDirectory()+

"/"+context.getPackageName());

if(!dir.exists()){

dir.mkdir();

}

File desFile = new File(dir, UUID.randomUUID().toString());

return save(bitmap,format,quality,desFile);

}

## 7、三级缓存

网络缓存

本地缓存

内存缓存

## 8、压缩

### 一、质量压缩

质量压缩方法：在保持像素的前提下改变图片的位深及透明度等，来达到压缩图片的目的

1、bitmap图片的大小不会改变

2、bytes.length是随着quality变小而变小的。

这样适合去传递二进制的图片数据，比如分享图片，要传入二进制数据过去，限制500kb之内。

#### 代码

public static Bitmap compressImage(Bitmap image) {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

// 把ByteArrayInputStream数据生成图片

Bitmap bitmap = null;

image.compress(Bitmap.CompressFormat.JPEG, 5, baos);

byte[] bytes = baos.toByteArray();

// 把压缩后的数据baos存放到bytes中

bitmap = BitmapFactory.decodeByteArray(bytes, 0, bytes.length);

if (bitmap != null) {

loga(bitmap, baos.toByteArray());

}

return bitmap;

}

// 质量压缩方法，options的值是0-100，这里100表示原来图片的质量，不压缩，把压缩后的数据存放到baos中

image.compress(Bitmap.CompressFormat.JPEG, 100, baos);

int options = 90;

// 循环判断如果压缩后图片是否大于500kb,大于继续压缩

while (baos.toByteArray().length / 1024 > 100) {

// 重置baos即清空baos

baos.reset();

// 这里压缩options%，把压缩后的数据存放到baos中

image.compress(Bitmap.CompressFormat.JPEG, options, baos);

// 每次都减少10

if(options == 1){

break;

}else if (options <= 10) {

options -= 1;

} else {

options -= 10;

}

}

### 二、采样率压缩

设置inSampleSize的值(int类型)后，假如设为n，则宽和高都为原来的1/n，宽高都减少，内存降低。上面的代码没用过options.inJustDecodeBounds = true; 因为我是固定来取样的数据，为什么这个压缩方法叫采样率压缩？ 是因为配合inJustDecodeBounds，先获取图片的宽、高[这个过程就是取样]， 然后通过获取的宽高，动态的设置inSampleSize的值。 当inJustDecodeBounds设置为true的时候， BitmapFactory通过decodeResource或者decodeFile解码图片时， 将会返回空(null)的Bitmap对象，这样可以避免Bitmap的内存分配， 但是它可以返回Bitmap的宽度、高度以及MimeType。

#### 用法

int inSampleSize = getScaling(bitmap); bitmap = samplingRateCompression(path,inSampleSize);

#### 代码

private Bitmap samplingRateCompression(String path, int scaling) {

BitmapFactory.Options options = new BitmapFactory.Options();

options.inSampleSize = scaling;

Bitmap bitmap = BitmapFactory.decodeFile(path, options);

int size = (bitmap.getByteCount() / 1024 / 1024);

Log.i("wechat", "压缩后图片的大小" + (bitmap.getByteCount() / 1024 / 1024)

+ "M宽度为" + bitmap.getWidth() + "高度为" + bitmap.getHeight());

return bitmap;

}

/\*\*

\* 获取缩放比例

\* @param bitmap

\* @return

\*/

private int getScaling(Bitmap bitmap) {

//设置目标尺寸（以像素的宽度为标准）

int Targetsize = 1500;

int width = bitmap.getWidth();

int height = bitmap.getHeight();

//选择最大值作为比较值（保证图片的压缩大小）

int handleValue = width > height ? width : height;

//循环计算压缩比

int i = 1;

while (handleValue / i > Targetsize) {

i++;

}

}

### 三、缩放法压缩, 效果和方法2一样

Android中使用Matrix对图像进行缩放、旋转、平移、斜切等变换的。 Matrix是一个3\*3的矩阵，其值对应如下：

|scaleX, skewX, translateX|

|skewY, scaleY, translateY|

|0 ， 0 ， scale |

Matrix提供了一些方法来控制图片变换：

setTranslate(float dx,float dy)：控制Matrix进行位移。

setSkew(float kx,float ky)：控制Matrix进行倾斜，kx、ky为X、Y方向上的比例。

setSkew(float kx,float ky,float px,float py)：控制Matrix以px、py为轴心进行倾斜，kx、ky为X、Y方向上的倾斜比例。

setRotate(float degrees)：控制Matrix进行depress角度的旋转，轴心为（0,0）。

setRotate(float degrees,float px,float py)：控制Matrix进行depress角度的旋转，轴心为(px,py)。

setScale(float sx,float sy)：设置Matrix进行缩放，sx、sy为X、Y方向上的缩放比例。

setScale(float sx,float sy,float px,float py)：设置Matrix以(px,py)为轴心进行缩放，sx、sy为X、Y方向上的缩放比例。

注意：以上的set方法，均有对应的post和pre方法， Matrix调用一系列set,pre,post方法时,可视为将这些方法插入到一个队列. 当然,按照队列中从头至尾的顺序调用执行. 其中pre表示在队头插入一个方法,post表示在队尾插入一个方法. 而set表示把当前队列清空,并且总是位于队列的最中间位置. 当执行了一次set后: pre方法总是插入到set前部的队列的最前面,post方法总是插入到set后部的队列的最后面

#### 代码

private Bitmap ScalingCompression(Bitmap bitmap) {

Matrix matrix = new Matrix();

matrix.setScale(0.25f, 0.25f);//缩放效果类似于方法2

Bitmap bm = Bitmap.createBitmap(bitmap, 0, 0, bitmap.getWidth(),

bitmap.getHeight(), matrix, true);

Log.i("wechat", "压缩后图片的大小" + (bm.getByteCount() / 1024 / 1024)

+ "M宽度为" + bm.getWidth() + "高度为" + bm.getHeight());

return bm;

}

### 四、Bitmap.Config

#### 对比

原图尺寸：4M----转化为File---Bitmap大小

* ALPHA\_8----------6.77M -------45M
* ARGB\_4444-------9.37M -------22M
* ARGB\_8888-------6.77M -------45M
* RGB\_565-----------8.13M -------22M

Bitmap.Config ARGB\_4444：每个像素占四位，即A=4，R=4，G=4，B=4，那么一个像素点占4+4+4+4=16位

Bitmap.Config ARGB\_8888：每个像素占八位，即A=8，R=8，G=8，B=8，那么一个像素点占8+8+8+8=32位

Bitmap.Config RGB\_565：每个像素占位R=5，G=6，B=5，没有透明度，那么一个像素点占5+6+5=16位

Bitmap.Config ALPHA\_8：每个像素占八位，只有透明度，没有颜色。

一般情况下默认使用的是ARGB8888，由此可知它是最占内存的，因为一个像素占32位，8位=1字节，所以一个像素占4字节的内存。假设有一张480x800的图片，如果格式为ARGB8888，那么将会占用1500KB的内存。

#### 代码

private Bitmap bitmapConfig(String path) {

BitmapFactory.Options options = new BitmapFactory.Options();

options.inPreferredConfig = Bitmap.Config.RGB\_565;

Bitmap bm = BitmapFactory.decodeFile(path, options);

Log.i("wechat", "压缩后图片的大小" + (bm.getByteCount() / 1024f / 1024f)

+ "M宽度为" + bm.getWidth() + "高度为" + bm.getHeight());

return bm;

}

### 五、Bitmap提供的：createScaledBitmap

#### 方法

Bitmap.createScaledBitmap(src, dstWidth, dstHeight, filter);

**参数说明：**

src 用来构建子集的源位图

dstWidth 新位图期望的宽度

dstHeight 新位图期望的高度

filter 为true则选择抗锯齿

#### 补充抗锯齿的知识点

在Android中，目前，我知道有两种出现锯齿的情况。

* 1、当我们用Canvas绘制位图的时候，如果对位图进行了选择，则位图会出现锯齿。
* 2、在用View的RotateAnimation做动画时候， 如果View当中包含有大量的图形，也会出现锯齿。

我们分别以这两种情况加以考虑。 用Canvas绘制位图的的情况。 在用Canvas绘制位图时，一般地，我们使用drawBitmap函数家族， 在这些函数中，都有一个Paint参数， 要做到防止锯齿，我们就要使用到这个参数。如下：

首先在你的构造函数中，需要创建一个Paint。 Paint mPaint = new Paint（）；

然后，您需要设置两个参数:

* 1)mPaint.setAntiAlias(Boolean aa);
* 2)mPaint.setBitmapFilter(true)。

第一个函数是用来防止边缘的锯齿， (true时图像边缘相对清晰一点，锯齿痕迹不那么明显， false时，写上去的字不饱满，不美观，看地不太清楚)。

第二个函数是用来对位图进行滤波处理。

最后，在画图的时候，调用drawBitmap函数，只需要将整个Paint传入即可。

有时候，当你做RotateAnimation时， 你会发现，讨厌的锯齿又出现了。 这个时候，由于你不能控制位图的绘制， 只能用其他方法来实现防止锯齿。 另外，如果你画的位图很多。 不想每个位图的绘制都传入一个Paint。 还有的时候，你不可能控制每个窗口的绘制的时候， 您就需要用下面的方法来处理——对整个Canvas进行处理。

* 1）在您的构造函数中，创建一个Paint滤波器。 PaintFlagsDrawFilter mSetfil = new PaintFlagsDrawFilter(0, Paint.FILTERBITMAPFLAG); 第一个参数是你要清除的标志位， 第二个参数是你要设置的标志位。此处设置为对位图进行滤波。
* 2）当你在画图的时候， 如果是View则在onDraw当中，如果是ViewGroup则在dispatchDraw中调用如下函数。 canvas.setDrawFilter( mSetfil );

另外，在Drawable类及其子类中， 也有函数setFilterBitmap可以用来对Bitmap进行滤波处理， 这样，当你选择Drawable时，会有抗锯齿的效果。

#### 代码

private Bitmap createScaledBitmap(Bitmap bitmap) {

Bitmap bm = Bitmap.createScaledBitmap(bitmap, 200, 200, true);

Log.i("wechat", "压缩后图片的大小" + (bm.getByteCount() / 1024) + "KB宽度为"

+ bm.getWidth() + "高度为" + bm.getHeight());

return bm;

}

### 六、辅助方法（上述方法的）：

#### 通过路径获取bitmap的方法

* 1、利用BitmapFactory解析文件，转换为Bitmap

bitmap = BitmapFactory.decodeFile(path);

* 2、自己写解码，转换为Bitmap过程， 同样需使用BitmapFactory.decodeByteArray(buf, 0, len)；代码如下：

private Bitmap getBitmapByPath(String path) {

if (!new File(path).exists()) {

System.err.println("getBitmapByPath: 文件不存在");

return null;

}

byte[] buf = new byte[1024 \* 1024];// 1M

Bitmap bitmap = null;

try {

FileInputStream fis = new FileInputStream(path);

int len = fis.read(buf, 0, buf.length);

bitmap = BitmapFactory.decodeByteArray(buf, 0, len);

//当bitmap为空的时候，说明解析失败

if (bitmap == null) {

System.out.println("文件长度:" + len);

System.err.println("path: " + path + " 无法解析!!!");

}

} catch (Exception e) {

e.printStackTrace();

}

return bitmap;

}

#### 保存图片

代码：

private void savaPictrue(Bitmap bitmap) {

File file = new File("storage/emulated/0/DCIM/Camera/test.jpg");

FileOutputStream stream = null;

try {

stream = new FileOutputStream(file);

bitmap.compress(Bitmap.CompressFormat.JPEG, 100, stream);

Log.e("图片大小：", file.length() / 1024 / 1024 + "M");

} catch (Exception e) {

e.printStackTrace();

}

}

#### 剪切图片（这里只是裁剪图片，但是对图片的大小并不影响）

代码：

private void crop(Uri uri) {

// 裁剪图片意图

Intent intent = new Intent("com.android.camera.action.CROP");

intent.setDataAndType(uri, "image/\*");

intent.putExtra("crop", "true");

// 裁剪框的比例，1：1

intent.putExtra("aspectX", 1);

intent.putExtra("aspectY", 1);

// 裁剪后输出图片的尺寸大小

intent.putExtra("outputX", 300);

intent.putExtra("outputY", 300);

intent.putExtra("outputFormat", "JPEG");// 图片格式

intent.putExtra("noFaceDetection", true);// 取消人脸识别

intent.putExtra("return-data", true);

// 开启一个带有返回值的Activity，请求码为PHOTO\_REQUEST\_CUT

startActivityForResult(intent, 200);

}

private void logp(Bitmap bitmap) {

Log.i("wechat", "压缩前图片的大小" + (bitmap.getByteCount() / 1024 / 1024)

+ "M宽度为" + bitmap.getWidth() + "高度为" + bitmap.getHeight());

}

private static void loga(Bitmap bitmap, byte[] bytes) {

Log.i("wechat", "压缩后图片的大小" + (bitmap.getByteCount() / 1024 / 1024)

+ "M宽度为" + bitmap.getWidth() + "高度为" + bitmap.getHeight()

+ "bytes.length= " + (bytes.length / 1024) + "KB"

);

}

# 17、UI卡顿

## 1、UI卡顿原理

View的绘制帧数保持60fps是最佳，这要求每帧的绘制时间不超过16ms（1000/60），如果安卓不能在16ms内完成界面的渲染，那么就会出现卡顿现象

## 2、UI卡顿的原因分析

在UI线程中做轻微的耗时操作，导致UI线程卡顿

布局Layout过于复杂，无法在16ms内完成渲染

同一时间动画执行的次数过多，导致CPU和GPU负载过重

overDraw，导致像素在同一帧的时间内被绘制多次，使CPU和GPU负载过重

View频繁的触发measure、layout，导致measure、layout累计耗时过多和整个View频繁的重新渲染

频繁的触发GC操作导致线程暂停，会使得安卓系统在16ms内无法完成绘制

冗余资源及逻辑等导致加载和执行缓慢

ANR

## 3、UI卡顿的优化

布局优化

使用include、ViewStub、merge

不要出现过于嵌套和冗余的布局

使用自定义View取代复杂的View

ListView优化

复用convertView

滑动不加载

背景和图片优化

缩略图

图片压缩

避免ANR

不要在UI线程中做耗时操作

# 18、内存泄漏

## 1、Java内存泄漏引起的主要原因

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏

## 2、Java内存分配策略

静态存储区：又称方法区，主要存储全局变量和静态变量，在整个程序运行期间都存在

栈区：方法体的局部变量会在栈区创建空间，并在方法执行结束后会自动释放变量的空间和内存

堆区：保存动态产生的数据，如：new出来的对象和数组，在不使用的时候由Java回收器自动回收

## 3、Android解决内存泄漏的例子

单例造成的内存泄漏：在单例中，使用context.getApplicationContext()作为单例的context

匿名内部类造成的内存泄漏：由于非静态内部类持有匿名外部类的引用，必须将内部类设置为static

Handler造成的内存泄漏：使用static的Handler内部类，同时在实现内部类中持有Context的弱引用

避免使用static变量：由于static变量会跟Activity生命周期一致，当Activity退出后台被后台回收时，static变量是不安全，所以也要管理好static变量的生命周期

资源未关闭造成的内存泄漏：比如Socket、Broadcast、Cursor、Bitmap、ListView等，使用完后要关闭

AsyncTask造成的内存泄漏：由于非静态内部类持有匿名内部类的引用而造成内存泄漏，可以通过AsyncTask内部持有外部Activity的弱引用同时改为静态内部类或在onDestroy()中执行AsyncTask.cancel()进行修复

# 19、内存管理

## 1、Android内存管理机制

分配机制

管理机制

## 2、内存管理机制的特点

更少的占用内存

在合适的时候，合理的释放系统资源

在系统内存紧张的时候，能释放掉大部分不重要的资源

能合理的在特殊生命周期中，保存或还原重要数据

## 3、内存优化方法

Service完成任务后应停止它，或用IntentService（因为可以自动停止服务）代替Service

在UI不可见的时候，释放其UI资源

在系统内存紧张的时候，尽可能多的释放非重要资源

避免滥用Bitmap导致内存浪费

避免使用依赖注入框架

使用针对内存优化过的数据容器

使用ZIP对齐的APK

使用多进程

# 20、冷启动和热启动

## 1、什么是冷启动和热启动

冷启动：在启动应用前，系统中没有该应用的任何进程信息

热启动：在启动应用时，在已有的进程上启动应用（用户使用返回键退出应用，然后马上又重新启动应用）

## 2、冷启动和热启动的区别

冷启动：创建Application后再创建和初始化MainActivity

热启动：创建和初始化MainActivity即可

## 3、冷启动时间的计算

这个时间值从应用启动（创建进程）开始计算，到完成视图的第一次绘制为止

## 4、冷启动流程

Zygote进程中fork创建出一个新的进程

创建和初始化Application类、创建MainActivity

inflate布局、当onCreate/onStart/onResume方法都走完

contentView的measure/layout/draw显示在界面上

总结：Application构造方法->attachBaseContext()->onCreate()->Activity构造方法->onCreate()->配置主题中背景等属性->onStart()->onResume()->测量布局绘制显示在界面上

## 5、冷启动优化

减少第一个界面onCreate()方法的工作量

不要让Application参与业务的操作

不要在Application进行耗时操作

不要以静态变量的方式在Application中保存数据

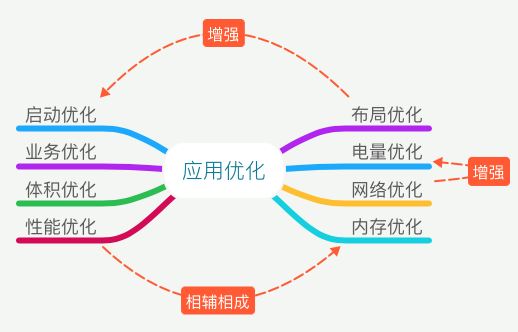
减少布局的复杂性和深度

不要在mainThread中加载资源

通过懒加载方式初始化第三方SDK

# 21、优化

对我们技术从业者而言,很多时候时候不是我们不知道怎么做,而是不知道做什么?今天系统的总结自己关于如何对Android应用进行优化的一些经验,共计八个维度.



## 1.布局优化

Android系统每个16ms发出VSYNC信号,触发对UI的渲染,要想达到界面流畅,必须实现60fps,也就意味着大多数的操作必须在16ms完成.

除了上面界面过于复杂导致渲染不能及时完成之外,还存在过度绘制问题.所谓过度绘制就是某个像素在同一帧的时间内被绘制多次.在多层次的UI界面中,如果不可见的UI也在进行绘制,那么这些重合区域的像素就会被绘制多次,从而浪费大量的CPU和GPU资源.过度绘制也发生在背景重叠的情况下,比如Layout中有自己的背景,同时子View中又有自己的背景.

### 检测

* 使用HierarchyViewer来查找Activity中的布局是否过于复杂
* 在开发者选项中打开Show GPU Overdraw选项进行观察是否存在过度绘制
* 在开发者选项中选择Profile GPU Rendering,选中On screen as bar
* 使用TraceView来观察CPU执行情况

### 优化

* 减少布局的层级,合理的使用include,merge,ViewStub
* 自定义组件的onDraw()中避免大量创建临时对象,比如String,以免频繁触发GC
* 自定义组件的onDraw()中,考虑使用canvas.clipRect()绘制需要被绘制的区域
* 对像ListView这样的组件容器,考虑使用convertView,使用ViewHolder,
* 考虑使用性能更高的组件,比如推荐使用RecycleView来代替ListView,使用staticlayout来实现自动换行

## 2.内存优化

资源总是有限的,内存同样也是一种资源.在Android当中,过度的/不恰当占用内存资源,会导致应用频繁被杀死,最终也会影响用户的整体体验.任何一名开发者,都应该将节省内存牢记心中.

### 检测

* 使用LeakCanary
* 使用MAT分析Java堆
* 使用Android Device Monitor中的Application Tracker追踪内存分配信息
* Android Studio中的Android Monitor,选择其中的Memory

### 优化

* 主动的释放内存,在onLowMemory()和onTrimMemory()中适当的释放内存
* 避免内存泄漏和内存溢出
* 在使用Bitmap的时候,考虑对其进行压缩,使用缓存或者改变颜色模式,比如android默认的颜色格式是ARGB\_8888,在要求不高的情况下可以采用RGB\_\_565,这样每个像素[1](http://blog.csdn.net/dd864140130/article/details/62431927" \l "fn:foot" \t "http://blog.csdn.net/dd864140130/article/details/_blank" \o "See footnote)占用的内存就可懂4byte到2byte.
* 减少帧动画的使用,如果需要,通过SurfaceView实现
* 使用更轻量级的数据结构,比如ArrayMap/SparseArray
* 合理的使用相关组件,比如Service和Webview,在不需要的时候主动结束其生命周期
* 合理的使用多进程,比如像音乐播放器类,可以分为主进程和播放进程
* 使用异步队列时考虑有界队列
* 如果你能明确知道HashMap的大小,那就再初始化时为其制定容量
* 数据需要在内存中进行缓存的时候，引入软引用/弱引用。

## 3.电量优化

电量是移动设备非常宝贵的资源,作为一名开发者,有必要为用户着想,减少电量的消耗.调查显示通常只有30%左右的电量是被程序核心的功能所消耗,比如界面渲染,剩下的70%则是被上报数据,位置更新,后台通知所消耗.

### 检测

* 手机选项中通过查看APP的电量消耗的统计数据
* 使用[Battery Historian Tool](https://github.com/google/battery-historian)来查看详细的电量消耗

### 优化

* 减少唤醒屏幕的次数与持续的时间,正确的使用WakeLock.
* 延迟非必须的操作到充电状态时,比如日志上报完全可以在夜间充电时完成,这点可以结合JobScheduler使用
* 使用传感器采集数据时,一旦不需要记得取消注册.
* 减少网络通信,合并通信.
* 合理使用定位功能,减少位置更新频率以及根据实际情况使用不同精度的定位需求

## 4.网络优化

现在App几乎都需要联网操作,做好网络优化一方面可以提高体验,另一方面可以减少流量和电量的损耗.另外,无论是对用户还是网络服务提供者,网络同样是一种资源,任何开发者都不应该假设网络资源是无限制的.

### 检测

* 使用Android Studio里的Network Traffic Tools来查看网络请求
* 使用Android Studio中的Monitor
* 使用Fidder或者Charles等抓包工具分析网络数据包

### 优化

* 有必要的时候务必做好缓存,无论是图片还是普通的数据,使用LruCache和DiskLruCache构建自己的缓存系统,并根据实际场景设计缓存策略
* 避免过度的网络同步,合并相关的网络请求
* 根据实际场景确定请求策略,避免使用固定的间隔频率来进行网络操作.比如连接WiFi并充电的情况下请求频率可以高,第一次网络请求失败后可以使用双倍的时间间隔来进行下一次
* 减少数据传输量,对传输的数据做压缩.如果传输的是图片,需要选择合适的图片格式以及根据显示大小请求合适规格的图片.对于普通数据,可以考虑使用ProtocalBuffers来减小传输数据的大小.
* 某些情况下可以采用IP直连,一方面可以减少DNS解析时间,另一方面可以防止域名劫持

## 5.启动优化

启动优化看起来并不是那么必要,但从心理学角度而言,越快的启动速度往往给用户以性能好,高效可靠的心理暗示,这就很容易让用户对其产生好感,为你后面打动用户留下了余地.

### 检测

* 使用Method Tracing
* 使用Systrace,比如在onCreate中添加trace.beginSection()和trace.endSection()
* 使用adb shell am start -W [packageName]/[packageName.MainActivity]测量冷启动时间

### 优化

* Activity的onCreate()中减少复杂和耗时的操作
* Application的onCreate(),attachBaseContext()中同样减少复杂和耗时的操作,但是对于很多App在此处会执行大量组件和服务的初始化操作,如果可能考虑并行初始化
* 提供自定义启动窗口,比如将一张图片通过设置主题的方式显示为启动窗口.
* 优化布局

## 6.体积优化

对用户而言,无论是用户空间还是网络,亦或是时间,都是资源.体积优化就是为用户节省资源的重要一环.如果你现在做的是SDK类产品,那么体积优化同样重要.

### 检测

* 使用Android Lint检查没有使用的资源

### 优化

* 减少不必要的依赖库/Jar,在满足需求的前提下优先选择体积小的.
* 使用Proguard工具进行代码瘦身,优化,混淆
* 减少so文件的数量,根据实际情况提供so文件
* 使用Gradle中的shrinkResource来将无用的代码和资源排除在APK安装包之外
* 减少图片资源的大小,考虑图片压缩或者使用Vertor Drawable替代png/jpeg
* 有选择的提供对应分辨率的图片资源
* 复用已经存在的图片,多用通过代码对已有图片进行变换的方式实现复用
* 使用插件化技术(如果项目简单就不要使用)

## 7.性能优化

能发挥出100%的能力就不要只发挥其中的50%,这对应用而言并非坏事.同样的价格卖给用户两辆车,我想大多数人会选择性能更好的.

### 如何检测?

* 使用Lint执行静态分析,在Android Studio的Analysis->Inspect Code
* 在开发者选项中开启StrictMode或者在代码中开启
* 代码Review

### 如何优化?

* 任务并行化,对可能的任务进行并行操作,多借助线程池而非直接使用线程
* 如何需要序列化数据,优先考虑Android自身提供的而非Java提供的Serializable
* 选择合适的数据结构,明确List/Set/Map/Stack操作的复杂度
* 使用Android提供更高效的容器,比如使用ArrayMap来代替HashMap,此外还是有SparseBoolMap,SparseIntMap,SparseLongMap
* 使用静态常量代替Enum类型,可以减少至少两倍的内存消耗
* 使用对象池技术,比如提供想String一样的对象池
* 使用缓存技术
* 字符串拼接操作有限使用StringBuilder
* 对相关的算法和逻辑进行优化,减少不必要的流程
* 采用JNI,对计算量较大的逻辑将其协程so文件,如图片处理

## 8.业务优化

除了上述比较通用的优化方案之外,也应该花点时间放在业务优化上.很多时候,迫于时间压迫,当前实现业务的方案并非最优.比如为了支持多张图片上传,很多人直接使用串行操作,尽管这样做容易实现,但是却并非最佳.

由于每个产品的业务并不相同,也就很难有通用的优化方案,这里又两个目标值得思考:

* 如果有可能,串行业务并行化
* 如果有可能,简化业务流程.将一大象关进冰箱的方法就是打开冰箱,将大象放进去,最后关闭冰箱.

之所以把业务优化放在最后的根本原因是业务优化的风险较高,需要团队的整体配合来完成.

## 9.其他优化

### 1、Android不用静态变量存储数据

静态变量等数据由于进程已经被杀死而被初始化

使用其他数据传输方式：文件/sp/contentProvider

### 2、SharePreference安全问题

不能跨进程同步

文件不宜过大

### 3、内存对象序列化

Serializeble：是java的序列化方式，Serializeble在序列化的时候会产生大量的临时对象，从而引起频繁的GC

Parcelable：是Android的序列化方式，且性能比Serializeble高，Parcelable不能使用在要将数据存储在硬盘上的情况

### 4、避免在UI线程中做繁重的操作

### 5、数据存储

1、持久化存储--->保存到SD卡上

2、跟随APP安装到卸载（不包括更新）--->SP、数据库、APP磁盘缓存空间

3、跟随APP启动到结束--->强应用、软引用、弱引用。

# 22、架构模式

**模式名词解释：**

**在MVC中：**

Model：业务逻辑和实体模型

View：对应于布局文件

Controllor：对应于Activity

**在MVP中：**

Model业务逻辑和实体模型

View 对应于Activity，负责View的绘制以及与用户交互

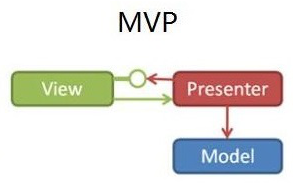
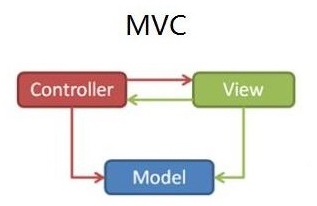
Presenter 负责完成View于Model间的交互

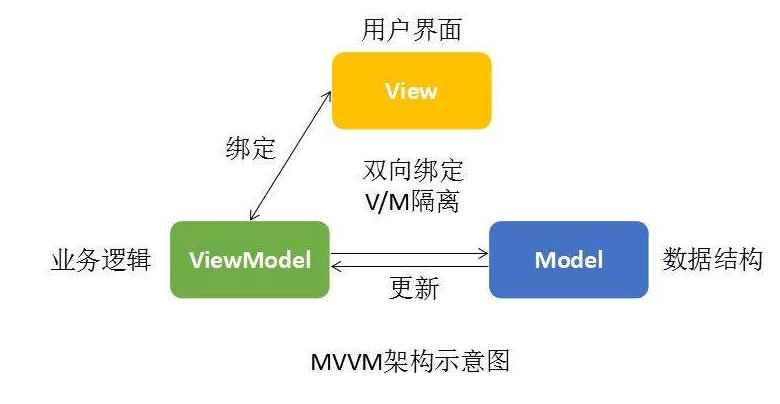
**在MVVM中：**

Model：数据模型层，包含业务逻辑和校验逻辑。

View：屏幕上显示的UI界面（layout、views）。

ViewModel：View和Model之间的链接桥梁，处理视图逻辑。





MVC的不足在此不做讨论。

MVP的不足:

1、接口过多，一定程度影响了编码效率。其实这也不算是不足，为了更好的分层解耦，这也是必须的。

2、Presenter的代码量过大。

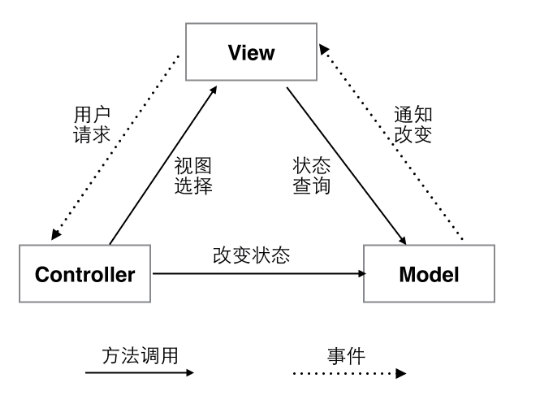
MVVM的优点：

1、可重用性。你可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑。 在Android中，布局里可以进行一个视图逻辑，并且Model发生变化，View也随着发生变化。

2、低耦合。以前Activity、Fragment中需要把数据填充到View，还要进行一些视图逻辑。现在这些都可在布局中完成。甚至都不需要再Activity、Fragment去findViewById。这时候Activity、Fragment只需要做好的逻辑处理就可以了。

## 22.1、框架模式MVC在安卓中的实践

模型图：



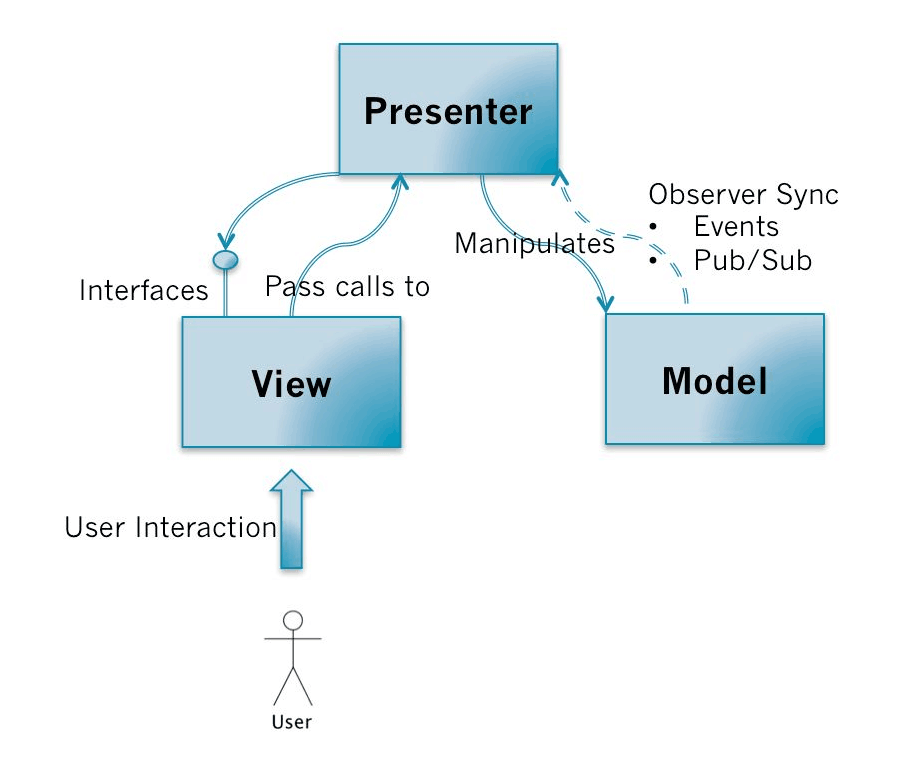
参考地址：（反正我没看）

<http://blog.csdn.net/qq_30379689/article/details/52909656>

下面是的我写的MVC的项目地址：

（项目不是MVP和MVVM，那就是MVC，你好意思要项目例子？）

## 22.2、框架模式MVP在安卓中的实践



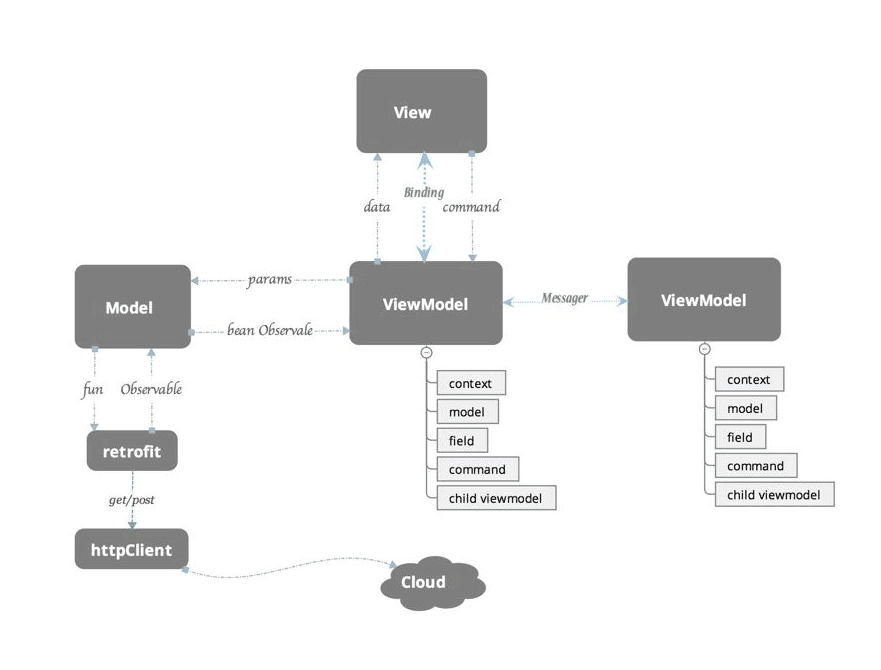
参考地址：（反正我没看）

<http://blog.csdn.net/qq_30379689/article/details/52910567>

下面是的我写的MVP的项目地址：

(还没有相关项目，正在努力搭建中)

## 22.3、框架模式MVVM之DataBinding的实践



参考地址：（反正我没看）

http://blog.csdn.net/qq\_30379689/article/details/53037430

下面是的我写的MVP的项目地址：

(还没有相关项目，正在努力搭建中)

### DataBinding上的双向数据绑定！

https://halfthought.wordpress.com/2016/03/23/2-way-data-binding-on-android/comment-page-1

在Android Studio 2.1预览版3发布后，Android DataBinding就具有双向数据绑定功能。

DataBinding快速回顾

有关DataBinding的简短概述，现在可以将表达式添加到布局文件以引用数据模型中的变量。例如：

<layout ...>

<data>

<variable type="com.example.myapp.User" name="user"/>

</data>

<RelativeLayout ...>

<TextView android:text="@{user.firstName}" .../>

</RelativeLayout>

</layout>

以上将用户的名字绑定到TextView的文本字段。每当用户的数据发生变化时，用户界面都会更新。

**双向数据绑定：**

Android对于典型的数据输入是不可避免的，将用户输入的更改反映回模型通常很重要。例如，如果上述数据是联系人表单，那么将编辑的文本推回到模型中而不必从EditText中提取数据将会很好。以下是你如何做到这一点：

<layout ...>

<data>

<variable type="com.example.myapp.User" name="user"/>

</data>

<RelativeLayout ...>

<EditText android:text="@={user.firstName}" .../>

</RelativeLayout>

</layout>

唯一的区别是表达式用“@ = {}”而不是“@ {}”标记。预计大部分数据绑定将继续是单向的，我们不希望让所有这些listeners 创建去观察永远不会发生的变化。

隐式属性监听器

您也可以在其他视图上引用属性：

<layout ...>

<data>

<import type="android.view.View"/>

</data>

<RelativeLayout ...>

<CheckBox android:id="@+id/seeAds" .../>

<ImageView android:visibility="@{seeAds.checked ? View.VISIBLE : View.GONE}" .../>

</RelativeLayout>

</layout>

在上面的情况中，只要CheckBox的选中状态发生变化，ImageView的visibility就会改变。不需要自己附加一个监听器！这种表达式只适用于支持双向数据绑定的属性和具有绑定表达式的属性。

启用双向数据绑定

那么，你还需要什么？首先，您必须确保在项目的build.gradle中使用2.1-alpha3或更高版本的android gradle插件：

classpath'com.android.tools.build: gradle:2.1.0-alpha3 '

当然，您需要在build.gradle的模块的android部分启用数据绑定：

android {

...

dataBinding.enabled = true

}

最主要的是只有少数属性被支持。这是因为没有监听器允许数据绑定框架知道什么时候发生了变化。好消息是这些可能是你最关心的属性：

AbsListView android:selectedItemPosition

CalendarView android:date

CompoundButton android:checked

DatePicker android:year, android:month, android:day (是的，这些都是合成的，但我们有一个监听器，所以我们认为你会想要使用它们)

NumberPicker android:value

RadioGroup android:checkedButton

RatingBar android:rating

SeekBar android:progress

TabHost android:currentTab (you probably don’t care, but we had the listener)

TextView android:text

TimePicker android:hour, android:minute (again, synthetic, but we had the listener)

如果您的变量名称与您的布局中的视图ID相冲突，您现在也将开始收到警告。我们如何知道您的意思是表达式中的视图还是变量？

自定义双向绑定

我们假设你关心的属性不是上面列出的属性之一。你如何去做自己的？让我们假设你有一个颜色选择器查看你想要双向绑定的颜色选择。

public class ColorPicker extends View {

public void setColor(int color) { /\* ... \*/ }

public int getColor() { /\* ... \*/ }

public void setOnColorChangeListener(OnColorChangeListener listener) {

/\*...\*/

}

public interface OnColorChangeListener {

void onColorChange(ColorPicker view, int color);

}

}

上述View的一个重要方面是它有一个数据绑定框架可以监听的监听器。现在我们需要告诉数据绑定关于属性。最简单的方法是在任何类上使用InverseBindingMethod属性：

@InverseBindingMethods（{

@InverseBindingMethod（type = ColorPicker.class，attribute =“color”），

}）

在这种情况下，getter的名称与“app：color”的属性“getColor”的名称相匹配。如果名称不同，则可以提供方法属性来纠正该错误。这将创建一个综合属性，在使用带有后缀“AttrChanged”的属性名称绑定事件时使用。让我们来看看如何使用它来设置侦听器：

@BindingAdapter("colorAttrChanged")

public static void setColorListener(ColorPicker view,

final InverseBindingListener colorChange) {

if (colorChange == null) {

view.setOnColorChangeListener(null);

} else {

view.setOnColorChangeListener(new OnColorChangeListener() {

@Override

public void onColorChange(ColorView view, int color) {

colorChange.onChange();

}

});

}

}

这是最简单的一种BindingAdapter - 它在颜色更改侦听器类型和数据绑定框架InverseBindingListener使用的类型之间进行转换。然而，你也经常要支持绑定到colorChange事件，对吗？你将不得不结合听众：

@BindingAdapter(value = {"onColorChange", "colorAttrChanged"},

requireAll = false)

public static void setColorListener(ColorPicker view,

final OnColorChangeListener listener,

final InverseBindingListener colorChange) {

if (colorChange == null) {

view.setOnColorChangeListener(listener);

} else {

view.setOnColorChangeListener(new OnColorChangeListener() {

@Override

public void onColorChange(ColorView view, int color) {

if (listener != null) {

listener.onColorChange(view, color);

}

colorChange.onChange();

}

});

}

}

现在，您可以使用双向数据绑定并绑定到onColorChange事件，即使在相同的视图中。

InverseBindingAdapter

与BindingAdapter和setter类似，有些时候除了调用getter来获取值之外，还必须做其他的事情。例如，如果我们的ColorPicker仅枚举了几种颜色，我们可能需要将其转换为int：

@InverseBindingAdapter(attribute = "color")

public static int getColorInt(ColorPicker view) {

return ConvertColorEnumToInt(view.getColor());

}

防止循环

我们在双向数据绑定中遇到的一个问题是一种无限循环。当对某个值进行更改引发事件时，监听器将设置目标对象的值。这可能会引发另一个事件，并且可能会再次触发对视图的更改，而我们将停止循环。这种类型的循环通常在View的setter或者数据值的setter中被捕获，但是如果你不能控制View或者数据值的话，这是不能保证的。我们已经决定阻止通过BindingAdapters循环。对于我们的ColorPicker示例：

@BindingAdapter("color")

public static void setColor(ColorPicker view, int color) {

if (color != view.getColor()) {

view.setColor(color);

}

}

## 谷歌的DataBinding的使用文档

本文档解释了如何使用数据绑定库来编写声明式布局，并尽量减少绑定应用程序逻辑和布局所需的胶合代码。

数据绑定库提供了灵活性和广泛的兼容性 - 这是一个支持库，所以你可以在Android 2.1（API级别7+）的所有Android平台上使用它。

要使用数据绑定，Android Plugin for Gradle 1.5.0-alpha1 或更高版本是必需的。了解如何[更新Android插件for Gradle](https://developer.android.google.cn/studio/releases/gradle-plugin.html" \l "updating-plugin)。

### 建设环境

要开始使用数据绑定，请从Android SDK管理器的支持库中下载库。

要配置应用程序以使用数据绑定，请将该dataBinding 元素添加到build.gradle应用程序模块中的文件中。

使用下面的代码片段来配置数据绑定：

android { ....

dataBinding {

enabled = true }

}

如果您的应用程序模块依赖于使用数据绑定的库，则您的应用程序模块也必须在其build.gradle 文件中配置数据绑定。

另外，请确保您使用的是Android Studio的兼容版本。 Android Studio 1.3及更高版本支持数据绑定，如[Android Studio支持数据绑定中所述](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "studio_support)。

数据绑定编译器V2

Android Gradle插件3.1.0 Canary 6附带一个可选的新编译器。要开始使用它，请更新您的gradle.properties文件以包含以下行：

android.databinding.enableV2=true

### 在编译器v2中：

ViewBinding类是在java编译器之前由Android Gradle Plugin生成的。这可以避免由于不相关的原因导致java编译失败而导致太多的错误肯定错误。

在V1中，编译应用程序时会重新生成库的绑定类（以共享生成的代码并访问最终的“BR”和“R”文件）。在V2中，库保持其生成的绑定类以及映射器信息，这为多模块项目显着提高了数据绑定性能。

请注意，这个新的编译器是向后不兼容的，所以用v1编译的库不能被v2使用，反之亦然。

V2还会删除一些很少使用的功能来允许这些更改：

在V1中，一个应用程序能够提供绑定适配器，可以覆盖依赖项中的适配器。在V2中，它只对自己的模块/应用程序及其依赖项中的代码生效。

以前，如果一个布局文件在两个或多个不同的资源配置中包含一个 View具有相同id但不同类的数据，则数据绑定将查找最常见的父类。在V2中，[View](https://developer.android.google.cn/reference/android/view/View.html)当配置之间的类型不匹配时，它将始终默认为 。

在V2中，不同的模块不能在清单中使用相同的包名称，因为数据绑定将使用该包名来生成绑定映射器类。

### 数据绑定布局文件

#### 编写你的第一个数据绑定表达式

数据绑定布局文件稍有不同，从布局的根标签开始， 后跟数据元素和 视图根元素。这个视图元素是你的根在非绑定布局文件中的东西。示例文件如下所示：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
 <data>  
 <variable name="user" type="com.example.User"/>  
 </data>  
 <LinearLayout  
 android:orientation="vertical"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"/>  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.lastName}"/>  
 </LinearLayout>  
</layout>

用户变量中的数据描述了可以此布局中使用的性质。

< variable name = “user” type = “com.example.User” />

布局中的表达式使用“ @{}”语法写入属性属性中。在这里，TextView的文本被设置为用户的firstName属性：

<TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"/>

#### 数据对象

现在让我们假设你有一个普通的Java对象（POJO）用户：

public class User {  
 public final String firstName;  
 public final String lastName;  
 public User(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
}

这种类型的对象的数据永远不会改变。在应用程序中通常会读取一次数据，之后再也不会更改。也可以使用JavaBeans对象：

public class User {  
 private final String firstName;  
 private final String lastName;  
 public User(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 }  
 public String getFirstName() {  
 return this.firstName;  
 }  
 public String getLastName() {  
 return this.lastName;  
 }  
}

从数据绑定的角度来看，这两个类是等价的。@{user.firstName}用于TextView android:text属性的表达式将访问firstName前一类中的字段和getFirstName()后一类中的方法。或者，firstName()如果该方法存在，也将被解析。

#### 绑定数据

默认情况下，将根据布局文件的名称生成一个Binding类，将其转换为Pascal格式并将“Binding”后缀添加到该文件中。上面的布局文件是main\_activity.xml这样的生成类是 MainActivityBinding。这个类将布局属性（例如user变量）的所有绑定保存到布局的视图中，并知道如何为绑定表达式赋值。创建绑定的最简单方法是在膨胀时进行：

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 MainActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.main\_activity);  
 User user = new User("Test", "User");  
 binding.setUser(user);  
}

你完成了！运行应用程序，你会看到用户界面中的测试用户。或者，您可以通过以下方式获取视图：

MainActivityBinding binding = MainActivityBinding.inflate(getLayoutInflater());

如果您在ListView或RecyclerView适配器内使用数据绑定项目，则可能更愿意使用：

ListItemBinding binding = ListItemBinding.inflate(layoutInflater, viewGroup, false);  
//or  
ListItemBinding binding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false);

#### 事件处理

数据绑定允许您编写表达式来处理从视图中分派的事件（例如onClick）。除少数例外，事件属性名称由侦听器方法的名称来管理。例如，[View.OnLongClickListener](https://developer.android.google.cn/reference/android/view/View.OnLongClickListener.html)有一个方法[onLongClick()](https://developer.android.google.cn/reference/android/view/View.OnLongClickListener.html" \l "onLongClick(android.view.View))，所以这个事件的属性是android:onLongClick。处理事件有两种方法。

[方法引用](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "method_references)：在你的表达式中，你可以引用符合侦听器方法签名的方法。当表达式评估为方法引用时，数据绑定将方法引用和所有者对象包装在侦听器中，并将该侦听器设置在目标视图上。如果表达式求值为null，则数据绑定不会创建侦听器，而是设置空侦听器。

[监听器绑定](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "listener_bindings)：这些是在事件发生时被评估的lambda表达式。数据绑定总是创建一个监听器，它在视图上设置。事件发送时，监听器评估lambda表达式。

##### 方法参考

事件可以直接绑定到处理方法，类似于 android:onClick可以分配给Activity中的方法。与View#onClick属性相比，一个主要的优点是表达式在编译时被处理，所以如果方法不存在或者它的签名不正确，你会收到一个编译时错误。

方法引用和监听器绑定的主要区别在于实际的监听器实现是在绑定数据时创建的，而不是在事件触发时创建的。如果您喜欢在事件发生时评估表达式，则应该使用[侦听器绑定](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "listener_bindings)。

要将事件分配给其处理程序，请使用常规绑定表达式，其值是要调用的方法名称。例如，如果你的数据对象有两个方法：

public class MyHandlers {  
 public void onClickFriend(View view) { ... }  
}

绑定表达式可以为View分配一个点击监听器：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
 <data>  
 <variable name="handlers" type="com.example.MyHandlers"/>  
 <variable name="user" type="com.example.User"/>  
 </data>  
 <LinearLayout  
 android:orientation="vertical"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"  
 android:onClick="@{handlers::onClickFriend}"/>  
 </LinearLayout>  
</layout>

请注意，表达式中方法的签名必须与Listener对象中方法的签名完全匹配。

##### 监听器绑定

监听器绑定是事件发生时运行的绑定表达式。它们类似于方法引用，但是它们允许您运行任意的数据绑定表达式。此功能适用于Gradle 2.0版及更高版本的Android Gradle插件。

在方法引用中，方法的参数必须与事件侦听器的参数匹配。在监听器绑定中，只有你的返回值必须与监听器的期望返回值相匹配（除非它预期为void）。例如，您可以有一个具有以下方法的演示者类：

public class Presenter {  
 public void onSaveClick(Task task){}  
}

然后，您可以将click事件绑定到您的类，如下所示：

<?xml version="1.0" encoding="utf-8"?>  
 <layout xmlns:android="http://schemas.android.com/apk/res/android">  
 <data>  
 <variable name="task" type="com.android.example.Task" />  
 <variable name="presenter" type="com.android.example.Presenter" />  
 </data>  
 <LinearLayout android:layout\_width="match\_parent" android:layout\_height="match\_parent">  
 <Button android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
 android:onClick="@{() -> presenter.onSaveClick(task)}" />  
 </LinearLayout>  
 </layout>

监听器由仅允许作为表达式的根元素的lambda表达式表示。在表达式中使用回调函数时，数据绑定会自动为事件创建必要的侦听器和注册表。当视图触发事件时，数据绑定将评估给定的表达式。就像在常规的绑定表达式中一样，当这些监听器表达式被评估的时候，你仍然可以获得数据绑定的null和线程安全性。

请注意，在上面的例子中，我们没有定义view传入的参数[onClick(android.view.View)](https://developer.android.google.cn/reference/android/view/View.OnClickListener.html" \l "onClick(android.view.View))。监听器绑定为监听器参数提供了两个选择：您可以忽略该方法的所有参数或将其全部命名。如果您想要命名参数，则可以在表达式中使用它们。例如，上面的表达式可以写成：

android:onClick="@{(view) -> presenter.onSaveClick(task)}"

或者如果你想使用表达式中的参数，它可以如下工作：

public class Presenter {  
 public void onSaveClick(View view, Task task){}  
}

android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"

您可以使用多于一个参数的lambda表达式：

public class Presenter {  
 public void onCompletedChanged(Task task, boolean completed){}  
}

<CheckBox android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
 android:onCheckedChanged="@{(cb, isChecked) -> presenter.completeChanged(task, isChecked)}" />

如果正在侦听的事件返回一个其类型不是的值，则void表达式必须返回相同类型的值。例如，如果要监听长按事件，则应该返回表达式boolean。

public class Presenter {  
 public boolean onLongClick(View view, Task task){}  
}

android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"

如果由于null对象而无法评估表达式，Data Binding将返回该类型的默认Java值。例如，null对于引用类型，0for int， falsefor boolean等

如果您需要使用谓词（例如三元）表达式，则可以将其 void用作符号。

android:onClick="@{(v) -> v.isVisible() ? doSomething() : void}"

避免复杂的听众

Listener表达式非常强大，可以让你的代码非常容易阅读。另一方面，包含复杂表达式的听众会使您的布局难以阅读和维护。这些表达式应该像从UI中传递可用数据到回调方法一样简单。您应该在您从侦听器表达式调用的回调方法内实现任何业务逻辑。

存在一些专门的单击事件处理程序，它们需要一个属性， android:onClick以避免冲突。已经创建了以下属性以避免这种冲突：

|  |  |  |
| --- | --- | --- |
| 类 | 听众二传手 | 属性 |
| [SearchView](https://developer.android.google.cn/reference/android/widget/SearchView.html) | [setOnSearchClickListener(View.OnClickListener)](https://developer.android.google.cn/reference/android/widget/SearchView.html" \l "setOnSearchClickListener(android.view.View.OnClickListener)) | android:onSearchClick |
| [ZoomControls](https://developer.android.google.cn/reference/android/widget/ZoomControls.html) | [setOnZoomInClickListener(View.OnClickListener)](https://developer.android.google.cn/reference/android/widget/ZoomControls.html" \l "setOnZoomInClickListener(android.view.View.OnClickListener)) | android:onZoomIn |
| [ZoomControls](https://developer.android.google.cn/reference/android/widget/ZoomControls.html) | [setOnZoomOutClickListener(View.OnClickListener)](https://developer.android.google.cn/reference/android/widget/ZoomControls.html" \l "setOnZoomOutClickListener(android.view.View.OnClickListener)) | android:onZoomOut |

#### 布局细节

##### 进口

import元素中可以使用 零个或多个元素data。这些就像在Java中一样可以轻松地引用布局文件中的类。

<data>  
 <import type="android.view.View"/>  
</data>

现在，可以在你的绑定表达式中使用View：

<TextView  
 android:text="@{user.lastName}"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>

当有类名冲突时，其中一个类可能会被重命名为“别名：”

<import type="android.view.View"/>  
<import type="com.example.real.estate.View"  
 alias="Vista"/>

现在，Vista可能被用来引用 com.example.real.estate.View和 View可能被用来android.view.View在布局文件内引用 。导入的类型可以用作变量和表达式中的类型引用：

<data>  
 <import type="com.example.User"/>  
 <import type="java.util.List"/>  
 <variable name="user" type="User"/>  
 <variable name="userList" type="List&lt;User&gt;"/>  
</data>

注意：Android Studio尚未处理导入，因此导入变量的自动填充可能无法在您的IDE中工作。您的应用程序仍然可以正常编译，您可以通过在变量定义中使用完全限定的名称来解决IDE问题。

<TextView  
 android:text="@{((User)(user.connection)).lastName}"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

当在表达式中引用静态字段和方法时，也可以使用导入的类型：

<data>  
 <import type="com.example.MyStringUtils"/>  
 <variable name="user" type="com.example.User"/>  
</data>  
…  
<TextView  
 android:text="@{MyStringUtils.capitalize(user.lastName)}"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

就像在Java中一样，java.lang.\*自动导入。

##### 变量

variable元素内可以使用 任意数量的元素data。每个 variable元素描述可以在布局上设置的属性，以用于布局文件中的绑定表达式。

<data>  
 <import type="android.graphics.drawable.Drawable"/>  
 <variable name="user" type="com.example.User"/>  
 <variable name="image" type="Drawable"/>  
 <variable name="note" type="String"/>  
</data>

变量类型在编译时被检查，所以如果一个变量实现 [Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html)或者是一个[可观察的集合](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "observable_collections)，那么这个类型应该被反映出来。如果变量是不实现Observable \*接口的基类或接口，将不会观察变量！

当不同的配置文件（例如横向或纵向）有不同的布局文件时，变量将被合并。这些布局文件之间不得存在冲突的变量定义。

生成的绑定类将为每个描述的变量设置一个setter和getter。变量将采用默认的Java值，直到调用者被调用 - null用于引用类型， 0for int，falseforboolean等

根据context需要生成一个名为特殊的变量用于绑定表达式。该值context是 Context从根查看的[getContext()](https://developer.android.google.cn/reference/android/view/View.html" \l "getContext())。该context变量将被具有该名称的显式变量声明覆盖。

##### 自定义绑定类名称

默认情况下，根据布局文件的名称生成一个Binding类，以大写字母开头，删除下划线（\_）并大写下面的字母，然后后缀“Binding”。这个类将被放置在模块包下的数据绑定包中。例如，布局文件contact\_item.xml将生成 ContactItemBinding。如果模块包是 com.example.my.app，那么它将被放入com.example.my.app.databinding。

通过调整元素的class属性， 可以将绑定类重命名或放置在不同的包中data。例如：

<data class="ContactItem">  
 ...  
</data>

这将ContactItem在模块包中的数据绑定包中生成绑定类。如果该类应该在模块包中的其他包中生成，则可以用“。”作为前缀：

<data class=".ContactItem">  
 ...  
</data>

在这种情况下，ContactItem直接在模块包中生成。如果提供完整的软件包，则可以使用任何软件包：

<data class="com.example.ContactItem">  
 ...  
</data>

##### 包括

通过在属性中使用应用程序名称空间和变量名称，变量可以从包含的布局传递到包含的布局的绑定中：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:bind="http://schemas.android.com/apk/res-auto">  
 <data>  
 <variable name="user" type="com.example.User"/>  
 </data>  
 <LinearLayout  
 android:orientation="vertical"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
 <include layout="@layout/name"  
 bind:user="@{user}"/>  
 <include layout="@layout/contact"  
 bind:user="@{user}"/>  
 </LinearLayout>  
</layout>

在这里，和布局文件user中都必须有一个变量 。 name.xmlcontact.xml

数据绑定不支持include作为合并元素的直接子元素。例如， 不支持以下布局：

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:bind="http://schemas.android.com/apk/res-auto">  
 <data>  
 <variable name="user" type="com.example.User"/>  
 </data>  
 <merge>  
 <include layout="@layout/name"  
 bind:user="@{user}"/>  
 <include layout="@layout/contact"  
 bind:user="@{user}"/>  
 </merge>  
</layout>

##### 表达语言

共同特征

表达式语言看起来很像Java表达式。这些是一样的：

数学运算 + - / \* %

字符串连接 +

与或逻辑 && ||

二进制 & | ^

一元 + - ! ~

转移 >> >>> <<

判断 == > < >= <=

**instanceof**

Grouping ()

Literals - character, String, numeric, null

Cast

Method calls

Field access

Array access []

Ternary operator ?:

对照：

**instanceof**

分组 ()

文字 - 字符，字符串，数字， null

投

方法调用

现场访问

数组访问 []

三元操作符 ?:

例子：

android:text="@{String.valueOf(index + 1)}"  
android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"  
android:transitionName='@{"image\_" + id}'

**减少操作**

您可以在Java中使用的表达式语法中减少一些操作：

this

super

new

明确的泛型调用

空合并运算符

null合并运算符（??） 选择左操作数（如果不是null）或右操作数（如果为空）。

android:text="@{user.displayName ?? user.lastName}"

这在功能上等同于：

android:text="@{user.displayName != null ? user.displayName : user.lastName}"

**属性参考**

第一个已经在[编写](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "writing_your_first_data_binding_expressions)上面的[第一个数据绑定表达式中](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "writing_your_first_data_binding_expressions)讨论过了：简短形式的JavaBean引用。当一个表达式引用一个类的属性时，它对字段，获取器和ObservableFields使用相同的格式。

android:text="@{user.lastName}"

避免NullPointerException

生成的数据绑定代码自动检查空值并避免空指针异常。例如，在表达式中 @{user.name}，如果user为null， user.name将被赋予其默认值（null）。如果你是参考user.age，年龄是一个int，那么它将默认为0。

集合

公共集合：数组，列表，稀疏列表和地图，可以使用[]操作符为了方便而访问。

<data>  
 <import type="android.util.SparseArray"/>  
 <import type="java.util.Map"/>  
 <import type="java.util.List"/>  
 <variable name="list" type="List&lt;String&gt;"/>  
 <variable name="sparse" type="SparseArray&lt;String&gt;"/>  
 <variable name="map" type="Map&lt;String, String&gt;"/>  
 <variable name="index" type="int"/>  
 <variable name="key" type="String"/>  
</data>  
…  
android:text="@{list[index]}"  
…  
android:text="@{sparse[index]}"  
…  
android:text="@{map[key]}"

字符串文字

在属性值周围使用单引号时，在表达式中使用双引号很容易：

android:text='@{map["firstName"]}'

也可以使用双引号来包围属性值。这样做时，字符串文字应该使用'或者反引号（`）。

android:text="@{map[`firstName`}"  
android:text="@{map['firstName']}"

资源

使用正常语法可以将资源作为表达式的一部分进行访问：

android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"

格式字符串和复数可以通过提供参数来评估：

android:text="@{@string/nameFormat(firstName, lastName)}"  
android:text="@{@plurals/banana(bananaCount)}"

当一个复数有多个参数时，所有参数都应该传递：

Have an orange  
Have %d oranges  
  
android:text="@{@plurals/orange(orangeCount, orangeCount)}"

有些资源需要明确的类型评估。

|  |  |  |
| --- | --- | --- |
| 类型 | 正常参考 | 表达参考 |
| String[] | @array | @stringArray |
| INT [] | @array | @intArray |
| TypedArray | @array | @typedArray |
| Animator | @animator | @animator |
| StateListAnimator | @animator | @stateListAnimator |
| color int | @color | @color |
| ColorStateList | @color | @colorStateList |

#### 数据对象

任何普通的旧Java对象（PO​​JO）都可以用于数据绑定，但修改POJO不会导致UI更新。数据绑定的真正威力可以通过给你的数据对象在数据改变时提供通知。有三种不同的数据更改通知机制： [可观察对象](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "observable_objects)， [可观察域](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "observablefields)和 [可观察集合](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "observable_collections)。

当这些可观察的数据对象之一被绑定到UI并且数据对象的属性改变时，UI将被自动更新。

##### 可观察的对象

实现[Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html)接口的类将允许绑定将单个侦听器附加到绑定对象，以侦听该对象上所有属性的更改。

该[Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html)接口具有添加和删除侦听器的机制，但通知由开发人员决定。为了简化开发，[BaseObservable](https://developer.android.google.cn/reference/android/databinding/BaseObservable.html)创建了一个基类来实现监听器注册机制。数据类实现者仍然负责通知属性何时改变。这是通过给[Bindable](https://developer.android.google.cn/reference/android/databinding/Bindable.html)getter 分配一个注释并通知setter来完成的。

private static class User extends BaseObservable {  
 private String firstName;  
 private String lastName;  
 @Bindable  
 public String getFirstName() {  
 return this.firstName;  
 }  
 @Bindable  
 public String getLastName() {  
 return this.lastName;  
 }  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 notifyPropertyChanged(BR.firstName);  
 }  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 notifyPropertyChanged(BR.lastName);  
 }  
}

}

该[Bindable](https://developer.android.google.cn/reference/android/databinding/Bindable.html)注解生成编译期间在BR类文件中的条目。BR类文件将在模块包中生成。如果数据类的基类不能改变，那么 [Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html)可以使用方便[PropertyChangeRegistry](https://developer.android.google.cn/reference/android/databinding/PropertyChangeRegistry.html)的存储和有效地通知侦听器来实现接口 。

##### ObservableFields

A little work is involved in creating [Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html) classes, so developers who want to save time or have few properties may use [ObservableField](https://developer.android.google.cn/reference/android/databinding/ObservableField.html) and its siblings [ObservableBoolean](https://developer.android.google.cn/reference/android/databinding/ObservableBoolean.html), [ObservableByte](https://developer.android.google.cn/reference/android/databinding/ObservableByte.html), [ObservableChar](https://developer.android.google.cn/reference/android/databinding/ObservableChar.html), [ObservableShort](https://developer.android.google.cn/reference/android/databinding/ObservableShort.html), [ObservableInt](https://developer.android.google.cn/reference/android/databinding/ObservableInt.html), [ObservableLong](https://developer.android.google.cn/reference/android/databinding/ObservableLong.html), [ObservableFloat](https://developer.android.google.cn/reference/android/databinding/ObservableFloat.html),[ObservableDouble](https://developer.android.google.cn/reference/android/databinding/ObservableDouble.html), and [ObservableParcelable](https://developer.android.google.cn/reference/android/databinding/ObservableParcelable.html). ObservableFields are self-contained observable objects that have a single field. The primitive versions avoid boxing and unboxing during access operations. To use, create a public final field in the data class:

private static class User {  
 public final ObservableField<String> firstName =  
 new ObservableField<>();  
 public final ObservableField<String> lastName =  
 new ObservableField<>();  
 public final ObservableInt age = new ObservableInt();  
}

而已！要访问该值，请使用set和get访问器方法：

user.firstName.set("Google");  
int age = user.age.get();

Observable Collections

##### 可观察的集合

一些应用程序使用更多的动态结构来保存数据 可观察集合允许对这些数据对象进行键控访问。 [ObservableArrayMap](https://developer.android.google.cn/reference/android/databinding/ObservableArrayMap.html)当键是引用类型（如String）时非常有用。

ObservableArrayMap<String, Object> user = new ObservableArrayMap<>();  
user.put("firstName", "Google");  
user.put("lastName", "Inc.");  
user.put("age", 17);

在布局中，可以通过String键访问地图：

<data>  
 <import type="android.databinding.ObservableMap"/>  
 <variable name="user" type="ObservableMap&lt;String, Object&gt;"/>  
</data>  
…  
<TextView  
 android:text='@{user["lastName"]}'  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>  
<TextView  
 android:text='@{String.valueOf(1 + (Integer)user["age"])}'  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

[ObservableArrayList](https://developer.android.google.cn/reference/android/databinding/ObservableArrayList.html) 当键是一个整数时是有用的：

ObservableArrayList<Object> user = new ObservableArrayList<>();  
user.add("Google");  
user.add("Inc.");  
user.add(17);

在布局中，列表可以通过索引来访问：

<data>  
 <import type="android.databinding.ObservableList"/>  
 <import type="com.example.my.app.Fields"/>  
 <variable name="user" type="ObservableList&lt;Object&gt;"/>  
</data>  
…  
<TextView  
 android:text='@{user[Fields.LAST\_NAME]}'  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>  
<TextView  
 android:text='@{String.valueOf(1 + (Integer)user[Fields.AGE])}'  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

#### 生成的绑定

生成的绑定类将布局变量与布局中的视图链接起来。如前所述，绑定的名称和包可能是 [自定义的](https://developer.android.google.cn/topic/libraries/data-binding/index.html" \l "custom_binding_class_names)。生成的绑定类全部扩展[ViewDataBinding](https://developer.android.google.cn/reference/android/databinding/ViewDataBinding.html)。

##### 创建

应该在通货膨胀之后立即创建绑定，以确保在绑定到布局中带有表达式的视图之前，View层次结构不受干扰。有几种方法可以绑定到布局。最常见的是在Binding类中使用静态方法。膨胀方法膨胀了View层次结构，并将其绑定到一步。有一个简单的版本，只需要一个[LayoutInflater](https://developer.android.google.cn/reference/android/view/LayoutInflater.html)，也需要一个[ViewGroup](https://developer.android.google.cn/reference/android/view/ViewGroup.html)：

MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater);  
MyLayoutBinding binding = MyLayoutBinding.inflate(layoutInflater, viewGroup, false);

如果布局使用不同的机制膨胀，它可能会被分开绑定：

MyLayoutBinding binding = MyLayoutBinding.bind(viewRoot);

有时绑定不能预先知道。在这种情况下，绑定可以使用[DataBindingUtil](https://developer.android.google.cn/reference/android/databinding/DataBindingUtil.html)类创建：

ViewDataBinding binding = DataBindingUtil.inflate(LayoutInflater, layoutId,  
 parent, attachToParent);  
ViewDataBinding binding = DataBindingUtil.bindTo(viewRoot, layoutId);

Views With IDs

##### 带有ID的视图

将在布局中为每个视图生成一个公开的最终字段。该绑定在View层次结构上执行单个传递，提取带有ID的视图。这个机制可以比调用多个视图的findViewById更快。例如：

<layout xmlns:android="http://schemas.android.com/apk/res/android">  
 <data>  
 <variable name="user" type="com.example.User"/>  
 </data>  
 <LinearLayout  
 android:orientation="vertical"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName}"  
 android:id="@+id/firstName"/>  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.lastName}"  
 android:id="@+id/lastName"/>  
 </LinearLayout>  
</layout>

将会生成一个绑定类：

public final TextView firstName;  
public final TextView lastName;

ID不像没有数据绑定那样必要，但是仍然有一些情况下代码仍然需要访问视图。

##### 变量

每个变量将被赋予访问器方法。

<data>  
 <import type="android.graphics.drawable.Drawable"/>  
 <variable name="user" type="com.example.User"/>  
 <variable name="image" type="Drawable"/>  
 <variable name="note" type="String"/>  
</data>

将在绑定中生成setter和getters：

public abstract com.example.User getUser();  
public abstract void setUser(com.example.User user);  
public abstract Drawable getImage();  
public abstract void setImage(Drawable image);  
public abstract String getNote();  
public abstract void setNote(String note);

##### ViewStubs

[ViewStub](https://developer.android.google.cn/reference/android/view/ViewStub.html)与正常观点有些不同。他们从不可见的时候开始，当他们要么变得可见时，要么被明确告知膨胀时，他们通过膨胀另一种布局来取代布局。

由于ViewStub本质上从View层次消失，绑定对象中的View也必须消失以允许收集。因为视图是最终的，所以一个[ViewStubProxy](https://developer.android.google.cn/reference/android/databinding/ViewStubProxy.html)对象代替了这个ViewStub视图，当开发者存在的时候，它允许开发人员访问ViewStub，并且在充气时也可以访问充气的View层次结构ViewStub。

当膨胀另一个布局时，必须为新的布局建立绑定。因此，ViewStubProxy一定要听ViewStub的 [ViewStub.OnInflateListener](https://developer.android.google.cn/reference/android/view/ViewStub.OnInflateListener.html)，建立的绑定在那个时候。由于只有一个可以存在，所以ViewStubProxy允许开发者OnInflateListener在建立绑定之后设置 它将会调用。

##### 高级绑定

动态变量

有时，特定的绑定类将不被知道。例如，[RecyclerView.Adapter](https://developer.android.google.cn/reference/android/support/v7/widget/RecyclerView.Adapter.html)针对任意布局的 操作将不知道具体的绑定类。它仍然必须分配绑定值[onBindViewHolder(VH, int)](https://developer.android.google.cn/reference/android/support/v7/widget/RecyclerView.Adapter.html" \l "onBindViewHolder(VH, int))。

在这个例子中，RecyclerView绑定的所有布局都有一个“item”变量。所述BindingHolder具有getBinding返回方法 [ViewDataBinding](https://developer.android.google.cn/reference/android/databinding/ViewDataBinding.html)基础。

public void onBindViewHolder(BindingHolder holder, int position) {  
 final T item = mItems.get(position);  
 holder.getBinding().setVariable(BR.item, item);  
 holder.getBinding().executePendingBindings();  
}

立即绑定

当变量或可观察的变化时，绑定将被安排在下一帧之前改变。但有时候，绑定必须立即执行。要强制执行，请使用该 [executePendingBindings()](https://developer.android.google.cn/reference/android/databinding/ViewDataBinding.html" \l "executePendingBindings())方法。

背景线程

只要不是集合，就可以在后台线程中更改数据模型。数据绑定将在评估时本地化每个变量/字段，以避免任何并发问题。

#### 属性设置者

每当绑定值发生变化时，生成的绑定类必须使用绑定表达式在视图上调用setter方法。数据绑定框架可以自定义调用哪个方法来设置值。

##### 自动安装程序

对于一个属性，数据绑定试图找到方法setAttribute。属性的命名空间并不重要，只有属性名称本身。

例如，与TextView属性关联的表达式 android:text将查找setText（String）。如果表达式返回一个int，那么数据绑定将搜索一个setText（int）方法。请注意让表达式返回正确的类型，如果需要的话就进行转换。请注意，即使给定名称不存在任何属性，数据绑定也可以工作。然后，您可以使用数据绑定轻松地为任何setter创建属性。例如，支持DrawerLayout没有任何属性，但是有很多setter。您可以使用自动设置器来使用其中的一个。

<android.support.v4.widget.DrawerLayout  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 app:scrimColor="@{@color/scrim}"  
 app:drawerListener="@{fragment.drawerListener}"/>

Renamed Setters

##### 重命名Setters

一些属性的设置者不符合名称。对于这些方法，一个属性可能通过[BindingMethods](https://developer.android.google.cn/reference/android/databinding/BindingMethods.html)注释与setter相关联 。这必须与一个类相关联，并包含[BindingMethod](https://developer.android.google.cn/reference/android/databinding/BindingMethod.html)注释，每个重命名的方法一个。例如，android:tint属性确实与之关联[setImageTintList(ColorStateList)](https://developer.android.google.cn/reference/android/widget/ImageView.html" \l "setImageTintList(android.content.res.ColorStateList))，而不是 setTint。

@BindingMethods({  
 @BindingMethod(type = "android.widget.ImageView",  
 attribute = "android:tint",  
 method = "setImageTintList"),  
})

开发人员不太可能需要重命名setter; android框架的属性已经实现了。

##### 自定义安装程序

一些属性需要自定义绑定逻辑。例如，该android:paddingLeft 属性没有关联的setter 。相反，setPadding(left, top, right, bottom)存在。带[BindingAdapter](https://developer.android.google.cn/reference/android/databinding/BindingAdapter.html) 注释的静态绑定适配器方法允许开发人员定制如何调用属性的setter。

Android属性已经BindingAdapter创建。例如，这里是一个用于paddingLeft：

@BindingAdapter("android:paddingLeft")  
public static void setPaddingLeft(View view, int padding) {  
 view.setPadding(padding,  
 view.getPaddingTop(),  
 view.getPaddingRight(),  
 view.getPaddingBottom());  
}

绑定适配器对其他类型的自定义非常有用。例如，一个自定义的加载器可以被脱机调用来加载一个图像。

当发生冲突时，开发人员创建的绑定适配器将覆盖数据绑定默认适配器。

您也可以让适配器接收多个参数。

@BindingAdapter({"bind:imageUrl", "bind:error"})  
public static void loadImage(ImageView view, String url, Drawable error) {  
 Picasso.with(view.getContext()).load(url).error(error).into(view);  
}

<ImageView app:imageUrl="@{venue.imageUrl}"  
app:error="@{@drawable/venueError}"/>

如果imageUrl和 error都用于ImageView且imageUrl是字符串，并且错误是可绘制的，则将调用此适配器。

自定义名称空间在匹配过程中被忽略。

您也可以为android命名空间编写适配器。

绑定适配器方法可以选择在其处理程序中使用旧值。采用新旧价值观的方法应该具有所有旧属性的价值观，其次是新的价值观：

@BindingAdapter("android:paddingLeft")  
public static void setPaddingLeft(View view, int oldPadding, int newPadding) {  
 if (oldPadding != newPadding) {  
 view.setPadding(newPadding,  
 view.getPaddingTop(),  
 view.getPaddingRight(),  
 view.getPaddingBottom());  
 }  
}

事件处理程序只能用于一个抽象方法的接口或抽象类。例如：

@BindingAdapter("android:onLayoutChange")  
public static void setOnLayoutChangeListener(View view, View.OnLayoutChangeListener oldValue,  
 View.OnLayoutChangeListener newValue) {  
 if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB) {  
 if (oldValue != null) {  
 view.removeOnLayoutChangeListener(oldValue);  
 }  
 if (newValue != null) {  
 view.addOnLayoutChangeListener(newValue);  
 }  
 }  
}

当一个监听器有多个方法时，它必须被分成多个监听器。比如 [View.OnAttachStateChangeListener](https://developer.android.google.cn/reference/android/view/View.OnAttachStateChangeListener.html)有两种方法： [onViewAttachedToWindow()](https://developer.android.google.cn/reference/android/view/View.OnAttachStateChangeListener.html" \l "onViewAttachedToWindow(android.view.View))和[onViewDetachedFromWindow()](https://developer.android.google.cn/reference/android/view/View.OnAttachStateChangeListener.html" \l "onViewDetachedFromWindow(android.view.View))。然后我们必须创建两个接口来区分它们的属性和处理程序。

@TargetApi(VERSION\_CODES.HONEYCOMB\_MR1)  
public interface OnViewDetachedFromWindow {  
 void onViewDetachedFromWindow(View v);  
}  
  
@TargetApi(VERSION\_CODES.HONEYCOMB\_MR1)  
public interface OnViewAttachedToWindow {  
 void onViewAttachedToWindow(View v);  
}

因为更改一个侦听器也会影响另一个侦听器，所以我们必须有三个不同的绑定适配器，一个用于每个属性，另一个用于两个，如果它们都被设置。

@BindingAdapter("android:onViewAttachedToWindow")  
public static void setListener(View view, OnViewAttachedToWindow attached) {  
 setListener(view, null, attached);  
}  
  
@BindingAdapter("android:onViewDetachedFromWindow")  
public static void setListener(View view, OnViewDetachedFromWindow detached) {  
 setListener(view, detached, null);  
}  
  
@BindingAdapter({"android:onViewDetachedFromWindow", "android:onViewAttachedToWindow"})  
public static void setListener(View view, final OnViewDetachedFromWindow detach,  
 final OnViewAttachedToWindow attach) {  
 if (VERSION.SDK\_INT >= VERSION\_CODES.HONEYCOMB\_MR1) {  
 final OnAttachStateChangeListener newListener;  
 if (detach == null && attach == null) {  
 newListener = null;  
 } else {  
 newListener = new OnAttachStateChangeListener() {  
 @Override  
 public void onViewAttachedToWindow(View v) {  
 if (attach != null) {  
 attach.onViewAttachedToWindow(v);  
 }  
 }  
  
 @Override  
 public void onViewDetachedFromWindow(View v) {  
 if (detach != null) {  
 detach.onViewDetachedFromWindow(v);  
 }  
 }  
 };  
 }  
 final OnAttachStateChangeListener oldListener = ListenerUtil.trackListener(view,  
 newListener, R.id.onAttachStateChangeListener);  
 if (oldListener != null) {  
 view.removeOnAttachStateChangeListener(oldListener);  
 }  
 if (newListener != null) {  
 view.addOnAttachStateChangeListener(newListener);  
 }  
 }  
}

上面的例子比正常情况稍微复杂，因为View使用add和remove来代替set的方法[View.OnAttachStateChangeListener](https://developer.android.google.cn/reference/android/view/View.OnAttachStateChangeListener.html)。该android.databinding.adapters.ListenerUtil班可以帮助跟踪以前的听众，让他们可以在绑定Adaper被删除的。

通过注释的接口OnViewDetachedFromWindow和 OnViewAttachedToWindow用 @TargetApi(VERSION\_CODES.HONEYCOMB\_MR1)，数据绑定代码生成器知道在Honeycomb MR1和新设备，所支持的相同版本运行时，听众应该只产生 [addOnAttachStateChangeListener(View.OnAttachStateChangeListener)](https://developer.android.google.cn/reference/android/view/View.html" \l "addOnAttachStateChangeListener(android.view.View.OnAttachStateChangeListener))。

##### 转换器

##### 对象转换

从绑定表达式返回一个对象时，将从自动，重命名和自定义设置器中选择一个设置器。该对象将被转换为所选setter的参数类型。

这对于那些使用ObservableMaps来保存数据的人来说是很方便的。例如：

<TextView  
 android:text='@{userMap["lastName"]}'  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

将userMap返回一个对象，并且将对象自动转换为在二传手发现参数类型setText(CharSequence)。当参数类型可能混淆时，开发人员需要在表达式中输入。

##### 自定义转换

有时转换应该在特定类型之间自动进行。例如，设置背景时：

<View  
 android:background="@{isError ? @color/red : @color/white}"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

在这里，背景需要一个Drawable，但颜色是一个整数。每当a Drawable被预期和返回一个整数，int应该被转换成一个 ColorDrawable。这个转换是通过一个带有BindingConversion注解的静态方法完成的：

@BindingConversion  
public static ColorDrawable convertColorToDrawable(int color) {  
 return new ColorDrawable(color);  
}

请注意，转换只发生在setter级别，所以不允许混合类型如下所示：

<View  
 android:background="@{isError ? @drawable/error : @color/white}"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"/>

#### Android Studio支持数据绑定

Android Studio支持数据绑定代码的许多代码编辑功能。例如，它支持数据绑定表达式的以下功能：

语法突出显示

表达式语言语法错误的标记

XML代码完成

包括 [导航](https://www.jetbrains.com/help/idea/2016.1/navigation-in-source-code.html?origin=old_help)（如导航到声明）和 [快速文档的参考](https://www.jetbrains.com/help/idea/2016.1/viewing-inline-documentation.html?origin=old_help)

注意：如果没有错误，数组和 [泛型类型](https://docs.oracle.com/javase/tutorial/java/generics/types.html)（如[Observable](https://developer.android.google.cn/reference/android/databinding/Observable.html)类）可能会显示错误。

预览窗格显示数据绑定表达式的默认值（如果提供）。在下面的示例摘录了布局XML文件中的元素后，“预览”窗格将显示PLACEHOLDER默认的文本值TextView。

<TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{user.firstName, default=PLACEHOLDER}"/>

如果需要在项目的设计阶段显示默认值，则还可以使用工具属性而不是默认表达式值，如 [Design Time Layout Attributes中所述](http://tools.android.com/tips/layout-designtime-attributes)

# 23、插件化

## 1、插件化解决的问题

动态加载APK（反射、类加载器）

资源加载（反射、AssetManager、独立资源、分段资源）

代码加载（反射获取生命周期）

## 2、类加载器（Java中字节码添加到虚拟机中）

DexClassLoader：能够加载未安装的jar/apk/dex，主要用于动态加载和代码热更新

PathClassLoader：只能加载系统中已经安装过的apk

# 24、热更新

## 1、热更新主要流程

线上检查到Crash

拉出Bugfix分支修复Crash问题

jenkins构建和补丁生成

app通过推送或主动拉取补丁文件

将Bugfix代码合到master上

## 2、热更新主流框架

Dexposed

AndFix

Nuwa

Tinker

## 3、热更新的原理

在ClassLoader创建一个dexElements数组

将修复好的dex文件存放在dexElements数组的最前面

ClassLoader会遍历dexElements数组，找到最前面的dex文件优先加载

# 25、进程保活

## 1、进程的优先级

空进程

后台进程

服务进程

可见进程

前台进程

## 2、Android进程回收策略

Low memory Killer（定时执行）：通过一些比较复杂的评分机制，对进程进行打分，然后将分数高的进程判定为bad进程，杀死并释放内存

OOM\_ODJ：判别进程的优先级

## 3、Android保活方案

利用系统广播拉活

利用系统Service机制拉活

利用Native进程拉活

利用JobScheduler机制拉活

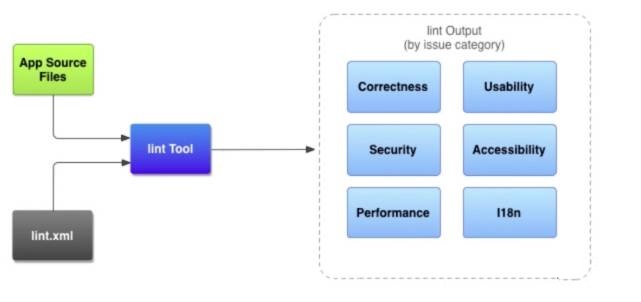
利用账号同步机制拉活

# 26、Lint

## 1、什么是Android Lint

Android Lint是一个静态代码分析工具，它能够对你的Android项目中潜在的Bug、可优化的代码、安全性、性能、可用性、可访问性、国际化等进行检查

## 2、Lint工作流程



## 3、配置Lint

创建Lint.xml到根目录下，自定义Lint安全等级等

在Java文件中可以使用@suppressLint(“NewApi”)来忽视Lint的报错

在xml文件中可以使用tool:ignore(“UnusedResources”)来忽视Lint的报错

自定义Lint检查，可以创建类，继承Detector和实现JavaPsiScanner

# 27、Kotlin

## 1、什么是Kotlin

Kotlin是一种基于JVM的编程语言

对Java的一种拓展，比Java更简洁

Kotlin支持函数式编程

Kotlin类和Java类可以相互调用

## 2、Kotlin环境搭建

Android Studio3.0之后不需要添加依赖，Android Studio3.0已经集成了Kotlin

直接在Plugin中下载Kotlin插件即可

系统会自动配置到Kotlin环境

# 面试总结：

http://blog.csdn.net/moyinghui/article/details/78114604

关于技术准备，主要是包含下面几项：

项目经历

Java基础

设计模式

算法

网络相关知识

Android基础

Android源码

Android性能优化

开源库源码

整个技术准备过程与面试过程是交替进行的，一边面试一边查缺补漏，如此往复。我当时是每天早上7点起床坐班车去公司，8点半吃完早餐就在工位上学习，一直学到9点半左右。然后10点开始办公，18时吃完晚餐，19时到20时，如果工作忙的话就办公，不忙的话就接着学习，后期公司任务变少了，有时候全天都在学习。

在这个过程中我真的是学的越多越觉得自己真是菜啊，一是觉得不知道的东西太多了，即广度太窄，二是知道的东西掌握的又不深，即深度不够。

面试知识点：

## 蓝厂面试问题：

事件分发流程

View的渲染机制

动画的原理，底层如何给上层信号

编译打包的过程

Android有多个资源文件夹，应用在不同分辨率下是如何查找对应文件夹下的资源的，描述整个过程

ANR的原理（回答主线程5秒阻塞是不行的，要读源码）

面试官是做framework的，所以问的东西偏framework，最后他说“虽然你是做应用的，但是不能浮于表面，要深入研究”，我觉得他说的很有道理。

## 度娘电话面试：

Bitmap 使用时候注意什么？

Oom 是否可以try catch ？

内存泄露如何产生？

适配器模式，装饰者模式，外观模式的异同？

ANR 如何产生？

String buffer 与string builder 的区别？

如何保证线程安全？

java四中引用

Jni 用过么？

多进程场景遇见过么？

关于handler，在任何地方new handler 都是什么线程下

sqlite升级，增加字段的语句

bitmap recycler 相关

强引用置为null，会不会被回收？

glide 使用什么缓存？

Glide 内存缓存如何控制大小？

如何保证多线程读写文件的安全？

## 直播公司

一面:

线程和进程的区别？

为什么要有线程，而不是仅仅用进程？

算法判断单链表成环与否？

如何实现线程同步？

hashmap数据结构？

arraylist 与 linkedlist 异同？

object类的equal 和hashcode 方法重写，为什么？

hashmap如何put数据（从hashmap源码角度讲解）？

简述IPC？

fragment之间传递数据的方式？

简述tcp四次挥手?

threadlocal原理

内存泄漏的可能原因？

用IDE如何分析内存泄漏？

OOM的可能原因？

线程死锁的4个条件？

差值器&估值器

二面：

简述消息机制相关

进程间通信方式？

Binder相关？

触摸事件的分发？

简述Activity启动全部过程？

okhttp源码？

RxJava简介及其源码解读？

性能优化如何分析systrace？

广播的分类？

点击事件被拦截，但是相传到下面的view，如何操作？

Glide源码？

ActicityThread相关？

volatile的原理

synchronize的原理

lock原理

三面：

三道算法题，要求在一个小时之内做完。

翻转一个单项链表 1->2->3->4->5->null =====> 5->4->3->2->1->null

string to integer

合并多个单有序链表（假设都是递增的）

四面：总监面，问了一些java 同步相关的。

## 由鹅厂与其他公司合资创立的公司

一场笔试加一场面试后挂了，面试官T4级别……。

笔试：

Activity生命周期简述

.常见内存泄漏情景及避免内存泄漏的措施

Actvity启动模式简述

简绘观察者设计模式UML图

算法，求公共子序列（或者是子串，记不清了）

Java四种引用

自定义view重写哪几个方法？

http 的session&cookie的区别

简述工作线程更新UI的方法

面试：

应用最多占多少内存

滑动卡顿如何解决（不同原因及对应处理方式）

自定义view实战

多线程，多进程 相关

Java四种引用的使用

## 某ding：

XX项目你负责什么

Sqlite 怎么增加一个字段

XX项目中是怎么创建数据库的

Sqlite 怎么删除一个字段

有什么你觉得自己做得好的地方

为什么用Retrofit（一个开源库）

Retrofit与之前的网络库有什么优势

XX项目中你们自己定义的线程池来管理任务，不使用框架，那么，后来新的项目怎么设计的

你认为Rxjava的线程池与你们自己实现任务管理框架有什么区别？

内存泄漏的常见场景

怎么发现&分析内存泄漏

## 某条：

处理有序数组为什么比无序数组更快 参考StackOverflow

热修复与插件化相关

Integer类是不是线程安全的，为什么

不使用同步锁如何实现线程安全

面试头条的时候在线编程：从上到下从左到右输出二叉树

针对concurrent包下面的一些类的问题

Android面试再也不是“能答出来Activity生命周期就OK”的时代了，你们看前面的面试题，有些问题牵扯到底层原理。另外光是concurrent包下面的东西就能把你问懵逼了。

有些规模小的公司很注重你是不是全能的，恨不得你什么都会，又给不起钱，福利待遇很不好，各种避税措施。

马爸爸的公司很注重项目经验是否匹配，往死了问项目，项目简单的话是很吃亏；

鹅厂希望你是聪明的机灵的；

度娘希望你基础好，热爱技术；

有些猎头朋友是非常专业的，不仅给你安排面试流程，还给你提供很多有价值的信息，另外猎头一般推荐的都是高级职位；、

基础很重要，大厂还是很青睐名校计算机专业出身的工程师的（我不是），根正苗红啊，他们基础一般都很好。所以半路出家的朋友们，注意补补基础，基础好，才能走得远。

要深入研究，不能浮于表面，多看看源码，被嘲笑为 API Player 是很难受的。

要广泛接触，不要只看着自己的一亩三分地，多看看流行的库，技术等。

最好是有blog和gayhub，并写一些有价值的内容，可以加分。

这篇面试纪要给我的感觉非常接地气，也恭喜作者实现了自己的求职目标，希望对大家有帮助。此外如果你希望更深入的了解面试相关，也可以访问【wanandroid.com】为大家准备的面试专题：

<http://www.wanandroid.com/article/list/0?cid=73>

# UI基础

- Android发展史、移动设备操作系统介绍

- Android环境搭建及工程目录介绍

- 常用UI布局:LinearLayout,RelativeLayout,FrameLayout,GridLayout

- 常用控件介绍：EditText、Button、ImageView、TextView等

- Spinner常用属性和方法

- 适配器介绍与使用

- Android应用的资源介绍

- AutoCompleteTextView常用属性和方法

- ScrollView和android.support.v4.widget.NestedScrollView的使用

- UI美化及Android资源

# 应用核心组件

- Activity的声明和使用

- Activity生命周期

- Activity的4种启动模式

- Activity现场保护

- Activity页面跳转及传值

- Task和Back Stack的基本概念

- Activity启动模式

- Intent常用属性和方法

- 使用Intent和IntentFilter进行通信

- AsyncTask异步任务

- ListView常用属性和方法

- ListView优化及分页

- AdapterView与Adaptr介绍

# 菜单与对话框

- Android菜单系统简介

- 使用XML定义菜单层次结构

- 使用AlertDialog类

- 通过Dialog Fragment管理和显示对话框

- 通过Activity事件处理程序管理和显示对话框

- 在工作线程中使用Toast

# 数据存储与IO

- 使用SharedPreference

- File存储

- SQLite数据库介绍

- SQLiteOpenHelper的用法

- CursorAdapter适配器使用

- ContentResolver实现系统数据库的操作

- 自定义ContentProvider

- CursorLoader的用法

- AsyncTaskLoader加载数据

# 移动开发主流界面实现

- 帧动画、补间动画、属性动画

- Fragment基本用法、传值和性能优化

- ViewPager的基本使用

- ViewPager的适配器

- ViewPager加载网络图片及优化

- 书签导航多种实现效果展示

- Handler、Looper消息传递机制的原理

- Handler、Looper源码分析

# 应用核心组件进阶

- ActionBar （Action View、SearchView）使用

- ToolBar使用

- 各种常用通知使用

- 广播接收器使用

- 跨进程服务AIDL和Messenger使用

- LruCache与Bitmap二次采样

# 后台服务

- Service介绍及期生命周期

- 使用AsyncTask运行异步任务

- IntentService介绍

- 手动创建线程和GUI线程同步

- 使用Alarm

# View深入探究

- 百度地图的使用

- 继承已有控件方式的自定义视图

- 继承于ViewGroup的自定义视图

- 自绘制的自定义视图

- PullToRefresh框架使用

- SlidingPaneLayout使用

- 机型适配的解决方案

- Git的使用

# Android新技术

- 音频和视频的播放

- 使用MediaRecorder录制音频

- RecyclerView使用

- TextInputLayout使用

- CoordinatorLayout使用

- Snackbar使用

- material design

- 利用Intent实现摄像头拍照

- 加速度传感器实现《摇一摇》

- ZXing实现生成二维码

- JPush极光推送的原理及用法

- 第三方登录原理及使用

- 友盟统计分析与使用

# Android网络应用

- 基于TCP协议的网络通信

- 使用URL访问网络资源

- 使用HTTP访问网络

- 使用WebService进行网络编程

- 使用Download Manager

- 使用蓝牙实现通信

- Connectivity Manager使用

- 管理和使用WiFi

- 近场通信介绍

# 流行技术

- 常用加密解密介绍

- 常用网络支付使用

- 事件分发原理

- ORM框架介绍

- Android WebView 应用

- 项目管理工具SVN使用

# Android高级

- Linux内核安全介绍

- Cloud to Device Messaging简介

- 使用License Verification Library实现版权保护

- 应用程序内收费实现

- 使用Wake Lock

- 处理不同硬件和软件的可用性

- 签名和发布应用程