**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. BroadcastReceiver广播 4](#_Toc10608)

[1.1. 两种注册方式 4](#_Toc32730)

[1.2. 举例 4](#_Toc23161)

[1.3. 广播类型（有序、无序） 4](#_Toc23862)

[1.4. 发送本地广播 5](#_Toc22693)

[2. Service 5](#_Toc26897)

[2.1. 概述 6](#_Toc18066)

[2.2. 启动方式 6](#_Toc31984)

[2.2.1. Context.startService() 6](#_Toc15269)

[2.2.2. Context.bindService() 6](#_Toc11981)

[2.2.3. 注意事项： 6](#_Toc2414)

[2.3. 生命周期 6](#_Toc22383)

[2.3.1. 被启动的服务的生命周期： 6](#_Toc9516)

[2.3.2. 被绑定的服务的生命周期 7](#_Toc24679)

[2.3.3. 被启动又被绑定的服务的生命周期 7](#_Toc7759)

[2.3.4. 当服务被停止时清除服务 7](#_Toc16127)

[2.3.5. 特别注意 7](#_Toc4651)

[2.4. 代码 8](#_Toc27553)

[2.5. Activity调用Service中的方法 10](#_Toc20034)

[2.6. 前台服务 11](#_Toc24962)

[2.7. Service和Activity通讯 11](#_Toc7623)

[2.7.1. 使用广播BroadcastReceiver 11](#_Toc20629)

[2.7.2. 使用Handler、Messenger 12](#_Toc25673)

[3. IntentService 14](#_Toc15369)

[3.1. 什么是IntentService？ 14](#_Toc30373)

[3.2. 怎样使用IntentService? 14](#_Toc20971)

[3.3. 源码分析 15](#_Toc27832)

[4. 使用Alarm、Service、Receiver定时重复任务 16](#_Toc27809)

[5. Fragment 17](#_Toc14693)

[5.1. 适配问题 17](#_Toc30031)

[5.2. Fragment的基本用法 18](#_Toc992)

[5.2.1. 直接在mainlayout里的name用全类名替换 18](#_Toc22128)

[5.2.2. 用Fragment管理器操作 19](#_Toc21612)

[5.3. FragmentViewPager 20](#_Toc26658)

[5.4. Fragment和Activity生命周期以及通信 22](#_Toc30699)

[5.5. Activity向Fragment传递数据（通过fragment.setArguments(bundle)） 23](#_Toc19456)

[5.6. Fragment向Activity传递数据 24](#_Toc22759)

[5.7. Fragment回退栈 25](#_Toc7280)

[5.8. 在频繁切换Fragment的情况中，List的notify不起作用 25](#_Toc16903)

[5.9. Fragment中嵌套Fragment 26](#_Toc13274)

[6. Cache缓存详解 26](#_Toc23820)

[6.1. LruCache缓存使用 26](#_Toc12835)

[6.1.1. 代码 26](#_Toc6855)

[6.1.2. 使用Volley 27](#_Toc23383)

[6.1.3. 手动存取 27](#_Toc21411)

[6.2. DiskLruCache硬盘缓存 27](#_Toc118)

[6.2.1. 概述 27](#_Toc11717)

[6.2.2. 打开缓存 27](#_Toc27571)

[6.2.3. 写入缓存 28](#_Toc31942)

[6.2.4. 读取缓存 30](#_Toc21420)

[6.2.5. 移除缓存 31](#_Toc12429)

[6.2.6. 其它API 31](#_Toc13667)

[6.2.7. 解读journal 31](#_Toc21477)

[7. 使用线程池处理异步任务 32](#_Toc13057)

[7.1. 概述 32](#_Toc24758)

[7.2. ExecutorService 33](#_Toc17832)

[7.2.1. 使用线程池管理线程的优点 33](#_Toc7949)

[7.2.2. ExecutorService简介 33](#_Toc17121)

[7.3. 线程池：ThreadPoolExecutor 33](#_Toc11602)

[7.3.1. 五种线程池 33](#_Toc31333)

[7.3.2. 通过Executors的工厂方法来获取这五种线程池 34](#_Toc873)

[7.4. 线程池ThreadPoolExecutor的使用 35](#_Toc5976)

[7.4.1. newFixedThreadPool 35](#_Toc3977)

[7.4.2. newSingleThreadExecutor 35](#_Toc2094)

[7.4.3. newCachedThreadPool 36](#_Toc17161)

[7.4.4. newScheduledThreadPool 36](#_Toc2762)

[7.4.5. newSingleThreadScheduledExecutor 37](#_Toc22144)

[7.5. 自定义线程池ThreadPoolExecutor 37](#_Toc9889)

[8. Camera使用相机和获取相册照片 37](#_Toc2018)

[8.1. 一般的拍照，清晰拍照，获取相册，裁剪 37](#_Toc27876)

[8.2. 拍出来的照片“歪了” 39](#_Toc27264)

[8.3. 使用自定义相机 40](#_Toc24598)

[9. 从相册获取相片 43](#_Toc7634)

[9.1. 通过设置intent直接获取 43](#_Toc19284)

[9.2. 查看选择的相册照片的地址 43](#_Toc30944)

[9.3. 选择后直接裁剪 46](#_Toc20342)

[9.4. 查看所有相册，并计算出相册的图片张数，最多者在最靠前 47](#_Toc27058)

[10. InputStream、File、String、bitmap、byte[]之间的转换 48](#_Toc19954)

[10.1. InputStream To byte[] 48](#_Toc8581)

[10.2. InputStream To String（stream--byte[]--String） 49](#_Toc6725)

[10.3. InputStream To File 49](#_Toc14409)

[10.4. byte[] to File 49](#_Toc23902)

[10.5. File To String 50](#_Toc26863)

[10.6. File To byte[](file--stream--byte[]) 50](#_Toc11583)

[10.7. Bitmap转byte[] 51](#_Toc25845)

[10.8. Bitmap转File 51](#_Toc17100)

[10.9. File转Btimap 51](#_Toc8593)

[11. 颜色推荐 51](#_Toc18138)

[12. 所有的权限uses-permission 52](#_Toc21219)

[13. ContentProvider 54](#_Toc14597)

[13.1. ContentProvider机理 54](#_Toc11806)

[13.2. ContentResolver对象与ContentProvider的交互 54](#_Toc17868)

[13.3. ContentProvider的屏蔽性 54](#_Toc4482)

[13.4. ContentProvider提供的数据形式 55](#_Toc20235)

[13.5. 创建ContentProvider的步奏 55](#_Toc27038)

[13.6. 来个例子 59](#_Toc16589)

[13.7. MIME类型与getType() 62](#_Toc9528)

[13.7.1. 概述： 62](#_Toc24064)

[13.7.2. MIME类型 62](#_Toc21954)

[13.7.2.1. 什么是MIME类型 62](#_Toc19358)

[13.7.2.2. MIME类型有什么用 62](#_Toc18515)

[13.7.3. getType() 63](#_Toc5700)

[13.7.3.1. 概述 63](#_Toc884)

[13.7.3.2. getType()与Activity的关系 63](#_Toc3402)

[13.7.4. 新建SecondActivity用来通过URI启动 64](#_Toc14402)

[13.7.5. AndroidManifest.xml 64](#_Toc26590)

[14. ContentResolver 64](#_Toc27059)

[15. 原生定位（不怎么可靠） 65](#_Toc2988)

[16. 百度定位（可靠） 67](#_Toc25349)

[17. 天气预报 72](#_Toc18973)

[18. 获取城市列表和天气 75](#_Toc7838)

[19. AsyncTask 76](#_Toc20529)

[19.1. 代码： 76](#_Toc2274)

[19.2. 请求网络的AsyncTask实例 78](#_Toc9854)

[20. 上传图片 78](#_Toc9934)

[21. 长连接socket 80](#_Toc18912)

[22. WebView显示网页 81](#_Toc9818)

[22.1. 加载本地/Web资源 81](#_Toc27345)

[22.2. WebView基本设置 82](#_Toc20201)

[22.3. 一些功能技巧的使用 86](#_Toc4247)

[22.3.1. 不支持长按 86](#_Toc14939)

[22.3.2. 支持Javascript 86](#_Toc1688)

[22.3.3. 使页面支持缩放 86](#_Toc17268)

[22.3.4. 使字体支持缩放 86](#_Toc28055)

[22.3.5. 需要手动输入用户名等，则必须设置支持获取手势焦点 86](#_Toc15543)

[22.3.6. 打开页面时，自适应屏幕(记载网页居中) 86](#_Toc22448)

[22.4. Java和js交互 87](#_Toc23273)

[22.4.1. js访问Java 87](#_Toc19778)

[22.4.2. Java访问js 88](#_Toc22741)

[22.4.3. 传base64的img字符串 88](#_Toc18995)

[22.4.3.1. 安卓4.4以下 88](#_Toc17930)

[22.4.3.2. 安卓4.4以上 88](#_Toc32292)

[22.4.4. Java和js交互有以下一些特点： 89](#_Toc16551)

[22.4.5. 混淆后交互失效 89](#_Toc12188)

[22.5. WebViewClient---影响View的事件的回调监听 89](#_Toc6772)

[22.5.1. WVC中函数概述 90](#_Toc24881)

[22.5.2. WVC之onPageStarted与onPageFinished 90](#_Toc11227)

[22.5.3. WVC之shouldOverrideUrlLoading 91](#_Toc4396)

[22.5.3.1. 在程序内打开网页 91](#_Toc27511)

[22.5.3.2. shouldOverrideUrlLoading用途 91](#_Toc28493)

[22.5.4. WVC之onReceivedError 92](#_Toc22358)

[22.5.5. WVC之onReceivedSslError 92](#_Toc397)

[22.5.5.1. 默认加载SSL出错的网站——出现空白页面 92](#_Toc25893)

[22.5.5.2. 使用SslErrorHandler.proceed()来继续加载 93](#_Toc14354)

[22.5.5.3. 在SSL发生错误时，onReceivedError会被回调吗？——不会 93](#_Toc6366)

[22.5.5.4. 所以对于onReceivedSslError结论来了 93](#_Toc12726)

[22.5.6. WVC之shouldInterceptRequest 93](#_Toc7593)

[22.5.7. WVC之其余函数 93](#_Toc15821)

[22.6. WebChromeClient----影响浏览器的事件的回调监听 94](#_Toc23318)

[22.6.1. 概述 94](#_Toc19726)

[22.6.1.1. 与WebViewClient的区别 94](#_Toc4315)

[22.6.1.2. 为什么叫WebChromeClient？ 94](#_Toc21193)

[22.6.1.3. WebChromeClient的常用函数 95](#_Toc5552)

[22.6.2. WCC之onJsAlert、onJsConfirm、onJsPrompt 95](#_Toc24788)

[22.6.3. WCC之onConsoleMessage 97](#_Toc31249)

[22.6.4. WCC之onProgressChanged 97](#_Toc23505)

[22.6.5. WCC之其它函数 98](#_Toc5598)

[22.7. LoadData()与loadDataWithBaseURL() 100](#_Toc21182)

[22.7.1. loadData 101](#_Toc55)

[22.7.1.1. 示例1：简单使用 101](#_Toc7152)

[22.7.1.2. 示例2：loadData的正确使用方式 101](#_Toc11017)

[22.7.2. loadDataWithBaseURL 101](#_Toc10976)

[22.7.2.1. 示例1：baseUrl的用法 102](#_Toc28027)

[22.7.2.2. 示例2：三种地址加载情况 102](#_Toc3014)

[22.7.3. 总结 102](#_Toc13520)

[22.8. 缓存的使用 102](#_Toc16673)

[22.8.1. 缓存构成 102](#_Toc3153)

[22.8.2. 缓存模式 102](#_Toc1798)

[22.8.3. 清理WebView缓存 103](#_Toc15368)

[22.9. 支持使用localStorage 104](#_Toc16341)

[22.10. 识别js中是否需要跳到新网页，来确定是否需要新开Activity 104](#_Toc21858)

[22.11. 网页中链接中打开app 105](#_Toc21900)

[22.12. 打开淘宝天猫出现问题解决 105](#_Toc26507)

[22.13. 重定向问题解决 106](#_Toc15171)

[22.14. 地理位置定位问题解决 106](#_Toc8046)

[22.15. 播放土豆优酷的视频 106](#_Toc8101)

[22.16. 返回按键，后退网页 107](#_Toc30924)

[22.16.1. 普通的实现 107](#_Toc40)

[22.16.2. 因重定向无法正常goBack()解决方案 108](#_Toc4367)

[22.16.3. 方法一：移除没有意义的重定向 108](#_Toc2835)

[22.16.4. 方法二：自己维护一个webview的历史栈 108](#_Toc29131)

[22.16.5. 方法三：配合前端类似方法二 108](#_Toc28300)

[22.16.6. 方法二详解 108](#_Toc6138)

[22.16.7. 总结 109](#_Toc25930)

[22.17. 滚动事件监听 109](#_Toc26241)

[22.18. 使用系统浏览器下载 110](#_Toc5530)

[22.19. ERR\_UNKNOWN\_URL\_SCHEME错误 110](#_Toc25986)

[22.20. ScollView 嵌套 WebView 底部空白，高度无法自适应解决 111](#_Toc15839)

[22.20.1. Google 官方建议 111](#_Toc32126)

[22.20.2. 移除重新添加新的 WebView 111](#_Toc16815)

[22.20.3. JS 注入，获取页面内容高度，设置成 WebView 高度 111](#_Toc5214)

# BroadcastReceiver广播

## 两种注册方式

**1、在应用程序的代码中注册**

当BroadcastReceiver更新UI，通常会使用这样的方法注册。启动Activity时候注册BroadcastReceiver，Activity不可见时候，取消注册。

registerReceiver(receiver, intentFilter1);

unregisterReceiver(receiver);

**2、在androidmanifest.xml当中注册**

<receiver android:name=*"com.cgy.sfgb.SmsReceiver"* >

<intent-filter>

<action android:name=*"longpiaopiao"* />

</intent-filter>

</receiver>

1)第一种不是常驻型广播，也就是说广播跟随程序的生命周期。

2)第二种是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

使用这样的方法注册弊端：它会始终处于活动状态，毕竟是手机开发，cpu和电源资源比较少，一直处于活动耗费大，不利。

## 举例

**private** **class** **SmsBroadcastReceiver** **extends** **BroadcastReceiver**{

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

**if** (intent.getAction().equals("gagafa")) {

**Toast**.*makeText*(**MainActivity**.**this**, "老子收到了嘎嘎发", 1000).show();

}**else** **if** (intent.getAction().equals("fengfeifei")) {

**Toast**.*makeText*(**MainActivity**.**this**, "老子收到了凤飞飞", 1000).show();

}

}

}

receiver = **new** SmsBroadcastReceiver();

intentFilter1 = **new** IntentFilter("gagafa")

registerReceiver(receiver, intentFilter1);**//非常驻型，需要动态注册**

## 广播类型（有序、无序）

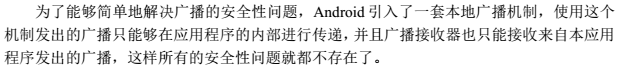
**有序广播**



**无序广播**

就是普通的广播，任何程序都能收到，同时收到

## 发送本地广播





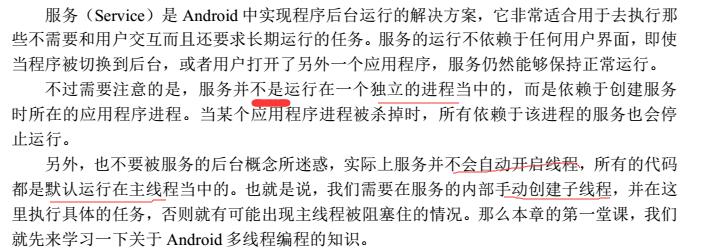






# Service

## 概述



## 启动方式

注意：Service是不能自己启动的，只有通过 Context 对象 调用startService和bindService方法来启动。也就是说Service不能自己调用的，所以我们看见很多Service都是用Activity调用的。

### Context.startService()

Service会经历onCreate -> onStart（如果Service还没有运行，则android先调用onCreate()然后调用onStart()；如果Service已经运行，则只调用onStart()，所以一个Service的onStart方法可能会重复调用多次）；stopService的时候直接onDestroy，如果是调用者自己直接退出而没有调用stopService的话，Service会一直在后台运行。该Service的调用者再启动起来后可以通过stopService关闭Service。 注意，多次调用Context.startservice()不会嵌套（即使会有相应的onStart()方法被调用），所以无论同一个服务被启动了多少次，一旦调用Context.stopService()或者stopSelf()，他都会被停止。

补充说明：传递给startService()的Intent对象会传递给onStart()方法。调用顺序为：onCreate --> onStart(可多次调用) --> onDestroy。

### Context.bindService()

Service会经历onCreate() -> onBind()，onBind将返回给客户端一个IBind接口实例，IBind允许客户端回调服务的方法，比如得到Service运行的状态或其他操作。这个时候把调用者（Context，例如Activity）会和Service绑定在一起，Context退出了，Srevice就会调用onUnbind -> onDestroyed相应退出，所谓绑定在一起就共存亡了。

补充说明：传递给bindService()的Intent对象会传递给onBind()，传递给unbindService()的Intent对象会传递给onUnbind()方法。调用顺序为：onCreate --> onBind(只一次，不可多次绑定) --> onUnbind --> onDestory。

### 注意事项：

在Service每一次的开启关闭过程中，只有onStart可被多次调用(通过多次startService调用)，其他onCreate，onBind，onUnbind，onDestory在一个生命周期中只能被调用一次。还有一点，目前我没有遇到过需要startService和bindService交互使用的情况（我认为不会有这种需求），所以不必去考虑交互的问题，待遇到时再考虑不迟。BroadcastReceiver只能通过startService启动Service，因为广播本身生命周期很短，bind的话没有意义

## 生命周期

**onCreate　　onStart　　onDestroy　　onBind**

### 被启动的服务的生命周期：

如果一个Service被某个Activity 调用 Context.startService 方法启动，那么不管是否有Activity使用bindService绑定或unbindService解除绑定到该Service，该Service都在后台运行。如果一个Service被startService 方法多次启动，那么onCreate方法只会调用一次，onStart将会被调用多次（对应调用startService的次数），并且系统只会创建Service的一个实例（因此你应该知道只需要一次stopService调用）。该Service将会一直在后台运行，而不管对应程序的Activity是否在运行，直到被调用stopService，或自身的stopSelf方法。当然如果系统资源不足，android系统也可能结束服务。

### 被绑定的服务的生命周期

如果一个Service被某个Activity 调用 Context.bindService 方法绑定启动，不管调用 bindService 调用几次，onCreate方法都只会调用一次，同时onStart方法始终不会被调用。当连接建立之后，Service将会一直运行，除非调用Context.unbindService 断开连接或者之前调用bindService 的 Context 不存在了（如Activity被finish的时候），系统将会自动停止Service，对应onDestroy将被调用。

### 被启动又被绑定的服务的生命周期

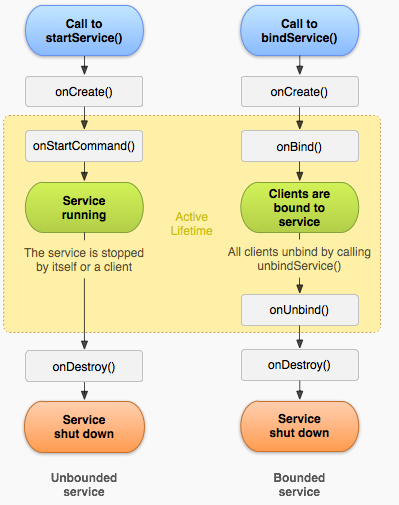
如果一个Service又被启动又被绑定，则该Service将会一直在后台运行。并且不管如何调用，onCreate始终只会调用一次，对应startService调用多少次，Service的onStart便会调用多少次。调用unbindService将不会停止Service，需要unbindService与stopService同时调用，才能终止 Service。

### 当服务被停止时清除服务

当一个Service被终止（1、调用stopService；2、调用stopSelf；3、不再有绑定的连接（没有被启动））时，onDestroy方法将会被调用，在这里你应当做一些清除工作，如停止在Service中创建并运行的线程。

### 特别注意

* 你应当知道在调用 bindService 绑定到Service的时候，你就应当保证在某处调用 unbindService 解除绑定（尽管 Activity 被 finish 的时候绑定会自动解除，并且Service会自动停止）；
* 你应当注意 使用 startService 启动服务之后，一定要使用 stopService停止服务，不管你是否使用bindService；
* 同时使用 startService 与 bindService 要注意到，Service 的终止，需要unbindService与stopService同时调用，才能终止 Service，不管 startService 与 bindService 的调用顺序，如果先调用 unbindService 此时服务不会自动终止，再调用 stopService 之后服务才会停止，如果先调用 stopService 此时服务也不会终止，而再调用 unbindService 或者 之前调用 bindService 的 Context 不存在了（如Activity 被 finish 的时候）之后服务才会自动停止；
* 当在旋转手机屏幕的时候，当手机屏幕在“横”“竖”变换时，此时如果你的 Activity 如果会自动旋转的话，旋转其实是 Activity 的重新创建，因此旋转之前的使用 bindService 建立的连接便会断开（Context 不存在了），对应服务的生命周期与上述相同。
* 在 sdk 2.0 及其以后的版本中，对应的 onStart 已经被否决变为了 onStartCommand，不过之前的 onStart 任然有效。这意味着，如果你开发的应用程序用的 sdk 为 2.0 及其以后的版本，那么你应当使用 onStartCommand 而不是 onStart。



## 代码

**MainActivity**

@Override

**public** **void** **onClick**(**View** v) {

**switch** (v.getId()) {

**case** **R**.**id**.**button\_on**:

**Log**.*v*(TAG, "ActivitystartService");

**Intent** intent = **new** Intent(**this**,**PlayService**.**class**);

**this**.startService(intent);

**break**;

**case** **R**.**id**.**button\_bind**:

**Log**.*v*(TAG, "ActivityBindService");

**Intent** intent2 = **new** Intent(**this**,**PlayService**.**class**);

intent2.putExtra("bind", "bindgaga");

//([Intent](eclipse-javadoc:%E2%98%82=%E9%9F%B3%E4%B9%90Service/D:\\/adt-bundle-windows-x86_64-20140702\\/sdk\\/platforms\\/android-22\\/android.jar<android.content(ContextWrapper.class%E2%98%83ContextWrapper~bindService~Landroid.content.Intent;~Landroid.content.ServiceConnection;~I%E2%98%82Intent) service, [ServiceConnection](eclipse-javadoc:%E2%98%82=%E9%9F%B3%E4%B9%90Service/D:\\/adt-bundle-windows-x86_64-20140702\\/sdk\\/platforms\\/android-22\\/android.jar<android.content(ContextWrapper.class%E2%98%83ContextWrapper~bindService~Landroid.content.Intent;~Landroid.content.ServiceConnection;~I%E2%98%82ServiceConnection) conn, int flags)

**this**.bindService(intent2, conn, **BIND\_AUTO\_CREATE**);

**break**;

**case** **R**.**id**.**button\_off**:

**Log**.*v*(TAG, "ActivitystopService");

**this**.stopService(**new** Intent(**this**,**PlayService**.**class**));

**break**;

}

}

**private** ServiceConnection conn = **new** ServiceConnection() {

**public** **void** **onServiceConnected**(**ComponentName** name, IBinder service) {

**Log**.*v*(TAG, "onServiceConnected");

}

**public** **void** **onServiceDisconnected**(**ComponentName** name) {

**Log**.*v*(TAG, "onServiceDisconnected");

}

};

@Override

**protected** **void** **onStop**() {

**super**.onStop();

**Log**.*v*(TAG, "ActivityonStop");

}

@Override

**protected** **void** **onDestroy**() {

**Log**.*v*(TAG, "ActivityonDestroy");

unbindService(conn);

**super**.onDestroy();

}

**PlayService**

**public** **class** **PlayService** **extends** **Service** {

**String** TAG = "TGA";

**MediaPlayer** mediaPlayer;

@Override

**public** IBinder **onBind**(**Intent** intent) {

**String** string = intent.getExtras().getString("bind");

**Toast**.*makeText*(**this**, string, **Toast**.**LENGTH\_SHORT**).show();

mediaPlayer.start();

**return** **null**;

}

@Override

**public** **boolean** **onUnbind**(**Intent** intent) {

mediaPlayer.stop();

**return** **super**.onUnbind(intent);

}

@Override

**public** **void** **onCreate**() {

**super**.onCreate();

**Toast**.*makeText*(**this**, "Play Service Created", **Toast**.**LENGTH\_SHORT**).show();

**Log**.*v*(TAG, "ServiceonCreate");

mediaPlayer = **MediaPlayer**.*create*(**this**, **R**.**raw**.**test**);

}

@Override

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

**Toast**.*makeText*(**this**, "Play Service onStart", **Toast**.**LENGTH\_SHORT**).show();

**Log**.*v*(TAG, "ServiceonStart");

mediaPlayer.start();

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

}

@Override

**public** **void** **onDestroy**() {

**super**.onDestroy();

**Toast**.*makeText*(**this**, "Play Service Stopped", **Toast**.**LENGTH\_SHORT**).show();

**Log**.*v*(TAG, "ServiconDestroy");

mediaPlayer.stop();

}

}

**Manifest**

<service android:name=*"com.cgy.servicedemo.PlayService"* >

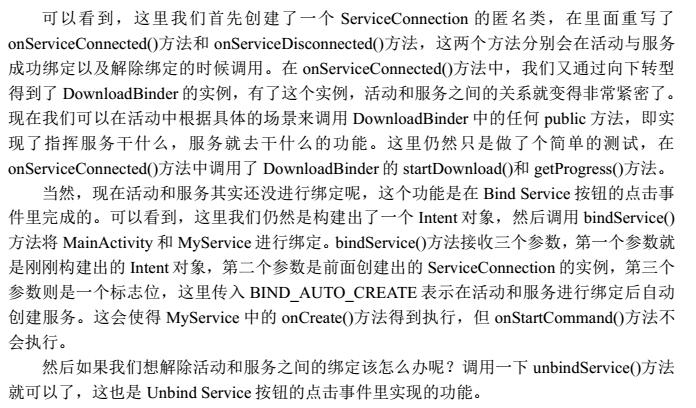
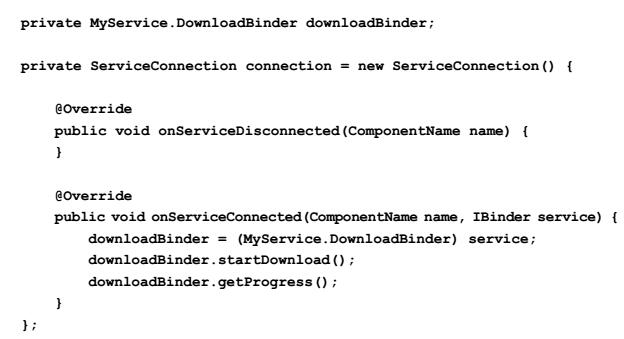
<intent-filter>

<action android:name=*"com.cgy.servicedemo.PlayService"* />

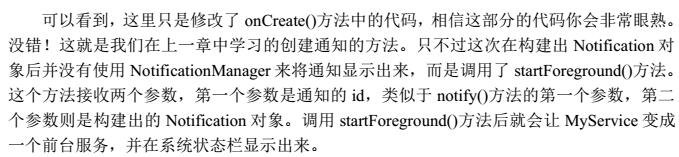
</intent-filter>

</service>

## Activity调用Service中的方法



## 前台服务

****

## Service和Activity通讯

### 使用广播BroadcastReceiver

Activity里：

**BroadcastReceiver** mReceiver = **new** BroadcastReceiver(){

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

**if** (**DownloadService**.**ACTION\_START**.equals(intent.getAction())) {

**FileInfo** fileInfo = (**FileInfo**) intent.getSerializableExtra("fileInfo");

mNotificationUtil.showNotification(fileInfo);

}

}

};

Service里：

**Intent** intent = **new** Intent(**DownloadService**.**ACTION\_START**);

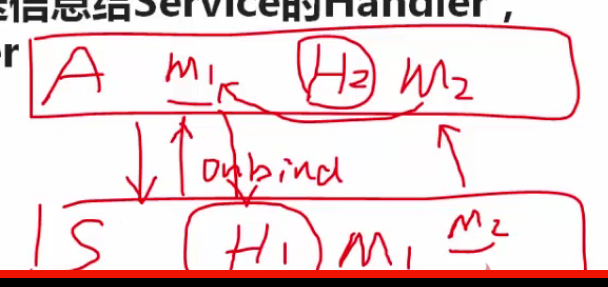
intent.putExtra("fileInfo", fileInfo);

sendBroadcast(intent);

### 使用Handler、Messenger

**使用Messenger实现跨进程或跨组件通信**

1. **在Service中创建Messenger包含Handler引用**
2. **在onBind方法中返回Messenger**
3. **Activity绑定Service，获得Service的Messenger**
4. **Activity中创建Messenger包含Handler引用**
5. **使用Service的Messenger发送消息给Service的Handler，信息包括Activity的Messenger**



Service里：

**public** **static** **final** **int** **MSG\_BIND** = 1;

**public** **static** **final** **int** **MSG\_START** = 2;

**public** **static** **final** **int** **MSG\_STOP** = 3;

**public** **static** **final** **int** **MSG\_FINISH** = 4;

**public** **static** **final** **int** **MSG\_UPDATED** = 5;

**private** **Messenger** mActivityMessenger = **null**;**//来自Activity的Messenger**

@Override

**public** IBinder **onBind**(**Intent** intent) {

**//创建一个Messenger对象,包含了Handler的引用**

**Messenger** messenger = **new** Messenger(mHandler);

**//返回Messenger的Binder；**

**return** messenger.getBinder();

}

**Handler** mHandler = **new** Handler(){

**public** **void** **handleMessage**(android.os.**Message** msg) {

**switch** (msg.what) {

**case** **MSG\_BIND**:

**//处理绑定的Messenger**

mActivityMessenger = msg.replyTo;

**break**;

}

};

};

**Message** message = **new** Message();

message.what = **MSG\_START**;

message.obj = fileInfo;

**try** {

mActivityMessenger.send(message);

} **catch** (**RemoteException** e) {

e.printStackTrace();

}

Activity里：

**private** **Messenger** mServiceMessenger = **null**;**//Service中的Messenger**

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

**//绑定Service**

**Intent** intent = **new** Intent(**this**, **DownloadService**.**class**);

bindService(intent, conn, **Service**.**BIND\_AUTO\_CREATE**);

}

ServiceConnection conn = **new** ServiceConnection() {

**public** **void** **onServiceConnected**(**ComponentName** name, IBinder service) {

**//获得Service中的Messenger**

mServiceMessenger = **new** Messenger(service);

**//创建Activity中的Messenger**

**Messenger** messenger = **new** Messenger(mHandler);

**//创建消息**

**Message** msg = **new** Message();

msg.what = **DownloadService**.**MSG\_BIND**;

**//msg的回复是Activity中的Messenger**

msg.replyTo = messenger;

**//使用Service的Messenger发送Activity中的Messenger**

**try** {

mServiceMessenger.send(msg);

} **catch** (**RemoteException** e) {

e.printStackTrace();

}

}

**public** **void** **onServiceDisconnected**(**ComponentName** name) {

**Log**.*v*("TAG", "onServiceDisconnected");

}

};

**Handler** mHandler = **new** Handler(){

**public** **void** **handleMessage**(**Message** msg) {

**switch** (msg.what) {

**case** **DownloadService**.**MSG\_UPDATED**:

mNotificationUtil.updateNotification(id, finished);

**break**;

**case** **DownloadService**.**MSG\_STOP**:

**FileInfo** fileInfo = (**FileInfo**) msg.obj;

mNotificationUtil.cancelNotification(fileInfo.getId());

**break**;

**case** **DownloadService**.**MSG\_START**:

mNotificationUtil.showNotification((**FileInfo**) msg.obj);

**break**;

**default**:

**break**;

}

}

};

# IntentService

## 什么是IntentService？

Service简称为后台服务，具有较高的优先级别。我们平时在Activity中直接开启Service，是运行在主线程的。如果想要执行耗时任务，我们必须自己开启线程。而IntentService是把任务放在子线程中执行的。

我们先来看一下官方的解析是怎样的？

This “work queue processor” pattern is commonly used to offload tasks from an application’s main thread. The IntentService class exists to simplify this pattern and take care of the mechanics. To use it, extend IntentService and implement onHandleIntent(Intent). IntentService will receive the Intents, launch a worker thread, and stop the service as appropriate.

All requests are handled on a single worker thread – they may take as long as necessary (and will not block the application’s main loop), but only one request will be processed at a time.

简单来说主要有一下几点

* IntentService是运行在子线程的；
* 多个请求会按启动的顺序执行,但是一次只会处理一个任务；
* 任务执行完毕以后会自动退出IntentService，不需要我们自己处理

## 怎样使用IntentService?

其实跟普通的Service差不多，自定义一个MyIntentService集成IntentService，重写构造方法和onHandleIntent方法，在HandleIntent里面执行我们的耗时任务等操作

**public** **class** **MyIntentService** **extends** **IntentService** {

**private** **static** **final** **String** **ACTION\_DOWNLOAD** = "com.szl.intentservicedemo.action.DOWNLOAD";

**private** **static** **final** **String** **ACTION\_UPLOAD** = "com.szl.intentservicedemo.action.UPLOAD ";

**private** **static** **final** **String** **EXTRA\_PARAM1** = "com.szl.intentservicedemo.extra.PARAM1";

**private** **static** **final** **String** **EXTRA\_PARAM2** = "com.szl.intentservicedemo.extra.PARAM2";

**public** **static** **final** **String** **TAG**="tag";

**public** **MyIntentService**() {

**super**("MyIntentService");

}

//这里省略了若干个方法

**private** **void** **handleDOwnload**(**String** param1, **String** param2){

}

**private** **void** **handleUpload**(**String** param1, **String** param2){

}

//处理我们启动的Service

@Override

**protected** **void** **onHandleIntent**(**Intent** intent) {

**if** (intent != **null**) {

**final** **String** action = intent.getAction();

**if** (**ACTION\_DOWNLOAD**.equals(action)) {

**final** **String** param1 = intent.getStringExtra(**EXTRA\_PARAM1**);

**final** **String** param2 = intent.getStringExtra(**EXTRA\_PARAM2**);

handleDOwnload(param1, param2);

} **else** **if** (**ACTION\_UPLOAD**.equals(action)) {

**final** **String** param1 = intent.getStringExtra(**EXTRA\_PARAM1**);

**final** **String** param2 = intent.getStringExtra(**EXTRA\_PARAM2**);

handleUpload(param1, param2);

}

}

}

}

别忘了在清单文件里面注册我们的Service

<service

android:name=*".MyIntentService"*

android:exported=*"false"* />

启动我们的Service,这里使用context.startService(intent)启动，当然你也可以用bindService启动

**public** **void** **startActionDownLoad**(**Context** context, **String** param1, **String** param2) {

**Intent** intent = **new** Intent(context, **MyIntentService**.**class**);

intent.setAction(**MyIntentService**.**ACTION\_DOWNLOAD**);

intent.putExtra(**MyIntentService**.**EXTRA\_PARAM1**, param1);

intent.putExtra(**MyIntentService**.**EXTRA\_PARAM2**, param2);

context.startService(intent);

}

**public** **void** **startActionUpload**(**Context** context, **String** param1, **String** param2) {

**Intent** intent = **new** Intent(context, **MyIntentService**.**class**);

intent.setAction(**MyIntentService**.**ACTION\_UPLOAD**);

intent.putExtra(**MyIntentService**.**EXTRA\_PARAM1**, param1);

intent.putExtra(**MyIntentService**.**EXTRA\_PARAM2**, param2);

context.startService(intent);

}

## 源码分析

从我们启动一个IntentService，调用的过程、生命周期和Service是一样的

1)首先我们先来看一下构造方法里面做了什么事情

public IntentService(String name) {

super();

mName = name;

}

其实很简单，只是调用父类的构造方法，并保存我们的字段

2）接着我们来看我们的onCreate方法做了什么？

@Override

public void onCreate() {

super.onCreate();

HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");

thread.start();

mServiceLooper = thread.getLooper();

mServiceHandler = new ServiceHandler(mServiceLooper);

}

简单来说，就是为我们初始化一个线程thread并启动它，并将线程的looper与我们的mServiceHandler绑定在一起。

3)接着我们来看onStartCommand（）方法做了什么？

@Override

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

onStart(intent, startId);

return mRedelivery ? START\_REDELIVER\_INTENT : START\_NOT\_STICKY;

}

我们可以看到在onStartCommand转调了onStart（）方法

4）onStart（）方法

@Override

public void onStart(Intent intent, int startId) {

Message msg = mServiceHandler.obtainMessage();

msg.arg1 = startId;

msg.obj = intent;

mServiceHandler.sendMessage(msg);

}

在onStart（）方法里面其实就是用我们的mServiceHandler发送信息（mServiceHandler.sendMessage(msg);），这样在我们handleMessage（）里面讲可以收到我们的信息，在handleMessage（）里面有调用onHandleIntent（）去处理我们的Intent，这就是为什么我们需要重写onHandleIntent的原因。

private final class ServiceHandler extends Handler {

public ServiceHandler(Looper looper) {

super(looper);

}

@Override

public void handleMessage(Message msg) {

onHandleIntent((Intent)msg.obj);

stopSelf(msg.arg1);

}

}

5）至于上面提到的我们的handleIntent是运行在子线程的，其实也很容易理解，因为我们知道handle运行在主线程还是子线程，是取决于我们与那个线程的looper绑定在一个的，而IntentService在onCreate方法中将我们的mServiceHandler与子线程的looper绑定在一起。核心代码体现如下

mServiceLooper = thread.getLooper();

mServiceHandler = new ServiceHandler(mServiceLooper);

6)为什么多个请求会按启动的顺序执行,但是一次只会处理一个任务？

我们知道多次调用context.startService方法，不会多次调用我们的onCreate（）方法，但会调用我们的onStart（）方法，而在我们的onStart（）方法里面我们调用mServiceHandler.sendMessage(msg);相当于是向消息队列里面插入一条信息，Looper会不断地从里面取出消息，交给相应 的hanlder处理，直到没有消息为止。如果对Handler消息机制不了解的话，建议先去了解

7）为什么任务执行完毕以后会自动退出Service，不需要我们自己处理？

这个就很简单了,因为在处理完信息以后，会调用stopSelf去停止相应的服务。

public void handleMessage(Message msg) {

onHandleIntent((Intent)msg.obj);

stopSelf(msg.arg1);

}

# 使用Alarm、Service、Receiver定时重复任务

**MyService**

**public** **class** **MyService** **extends** **Service**{

@Override

**public** IBinder **onBind**(**Intent** intent) {

**return** **null**;

}

@Override

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

**new** Thread(){

**public** **void** **run**() {

**//doSomeThing**

**Log**.*e*("tga", "hahah" + "----" + **new** Date().toString());

}

}.start();

**AlarmManager** manager = (**AlarmManager**) getSystemService(**ALARM\_SERVICE**);

**int** oneTime = 10 \* 1000;**//每十秒来一次**

**long** doAtTime = **SystemClock**.*elapsedRealtime*() + oneTime;

**Intent** in = **new** Intent(**this**, **MyReceiver**.**class**);

**PendingIntent** pendingIntent = **PendingIntent**.*getBroadcast*(**this**, 0, in, 0);

manager.set(**AlarmManager**.**ELAPSED\_REALTIME\_WAKEUP**, doAtTime, pendingIntent);

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

}

}

**MainActivity**

**public** **class** **MainActivity** **extends** **Activity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

startService(**new** Intent(**this**, **MyService**.**class**));

}

}

**MyReceiver**

**public** **class** **MyReceiver** **extends** **BroadcastReceiver**{

@Override

**public** **void** **onReceive**(**Context** context, **Intent** intent) {

context.startService(**new** Intent(context, **MyService**.**class**));

}

}

# Fragment

## 适配问题

1. 尽量不要用app包中的fragment，因为这个是在3.0之后才有的，支持的版本太高，在低版本中是是用不了的，尽量用android.support.v4.app.Fragment：可以兼容到1.6的版本
2. 关于这两个fragment使用<fragment>标签的问题
3. app.fragment和v4.fragment都是可以使用<fragment>标签的,只是在在使用的时候如果是app.fragment则没有什么特殊的地方继承Activity即可。
4. 当v4.fragment使用<fragment>标签的时候就要特别注意了,当这个Activity的布局中有<fragment>标签的时候，这个Activity必须继承 FragmentActivity，否则就会报错
5. FragmentManager用getSupportFragmentManager()获取，得到v4的FragmentManager；用 getFragmentManager()获取，得到app包的FragmentManager ；
6. [FragmentPagerAdapter与FragmentStatePagerAdapter差异](http://blog.csdn.net/caesardadi/article/details/11470397)：FragmentPagerAdapter最多保存3个，其他会被销毁掉。FragmentStatePagerAdapter用于fragment较多的情况

## Fragment的基本用法

### 直接在mainlayout里的name用全类名替换

**public** **class** **MainActivity** **extends** **FragmentActivity** {

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

requestWindowFeature(**Window**.**FEATURE\_NO\_TITLE**);

setContentView(**R**.**layout**.**activity\_main**);

}

}

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

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

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"* >

<fragment

android:id=*"@+id/fragment\_title"*

android:name=*"com.example.fragment922.TitleFragment"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"45dp"* />

<fragment

android:id=*"@+id/fragment\_content"*

android:name=*"com.example.fragment922.ContentFragment"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:layout\_below=*"@id/fragment\_title"* />

</RelativeLayout>

**public** **class** **TitleFragment** **extends** **Fragment**{

**private** **ImageButton** imageButton;

/\*

\* 表示fragment第一次创建绘制时系统回调的方法

\* 返回值view表示当前加载fragment视图，如果fragment不提供视图可以放回null

\* container，表示当前fragment插入actiivty的布局视图对象

\* savedInstanceState表示村春上一个fragment 的信息

\*/

@Override

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

**View** view = inflater.inflate(**R**.**layout**.**fragment\_title**,container,**false**);

imageButton = (**ImageButton**) view.findViewById(**R**.**id**.**title\_left\_btn**);

imageButton.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**Toast**.*makeText*(getActivity(),"i am an ImageButton in TitleFragment ! ",**Toast**.**LENGTH\_SHORT**).show();

}

});

**return** view;

}

}

**public** **class** **ContentFragment** **extends** **Fragment**{

@Override

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

**return** inflater.inflate(**R**.**layout**.**fragment\_content**, container,**false**);

}

}

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

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

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<TextView

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:gravity=*"center"*

android:text=*"使用Fragment做主面板"*

android:textSize=*"20sp"*

android:textStyle=*"bold"* />

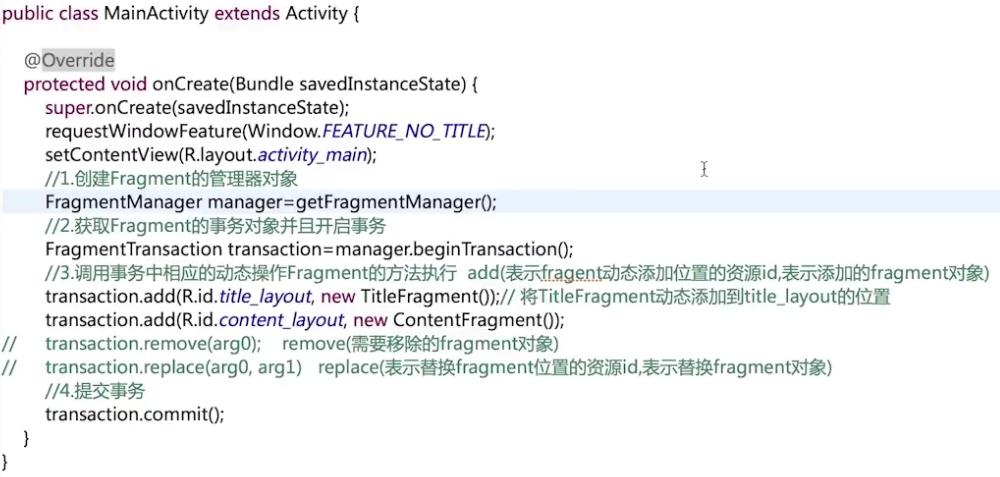
</LinearLayout>

### 用Fragment管理器操作

**FragmentManager** 可以初始化一次，重复使用：**FragmentManager** fm = getSupportFragmentManager();

**FragmentTransaction** commit一次后再次使用必须重新通过fm获取一次：**FragmentTransaction** transaction = fm.beginTransaction();

最低版本大于3.0的fragment



最低版本小于3.0，为了兼容

* 使用fragment的Activity要用FragmentActivity
* 获取事务管理器的方法为getSupportFragmentManager()
* 编写的Fragment继承于v4包下

**public** **class** **MainActivity** **extends** **FragmentActivity** **implements** OnClickListener{

**private** **LinearLayout** weixin;

**private** **LinearLayout** pengyou;

**private** **WeixinFragment** mweixin;

**private** **FriendFragment** mfriend;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

requestWindowFeature(**Window**.**FEATURE\_NO\_TITLE**);

setContentView(**R**.**layout**.**activity\_main**);

weixin = (**LinearLayout**) findViewById(**R**.**id**.**ll1**);

pengyou = (**LinearLayout**) findViewById(**R**.**id**.**ll2**);

weixin.setOnClickListener(**this**);

pengyou.setOnClickListener(**this**);

setDefaultFragment();**//设置默认的Fragment**

}

**private** **void** **setDefaultFragment**(){

**FragmentManager** fm = getSupportFragmentManager();

**FragmentTransaction** transaction = fm.beginTransaction();

mweixin = **new** WeixinFragment();

transaction.replace(**R**.**id**.**id\_content**, mweixin);

transaction.commit();

}

@Override

**public** **void** **onClick**(**View** v) {

**FragmentManager** fm= getSupportFragmentManager();

**FragmentTransaction** transaction = fm.beginTransaction();

**switch** (v.getId()) {

**case** **R**.**id**.**ll1**:

**if** (mweixin==**null**) {

mweixin = **new** WeixinFragment();

}

transaction.replace(**R**.**id**.**id\_content**, mweixin);**// 使用当前Fragment的布局替代id\_content的控件**

**break**;

**case** **R**.**id**.**ll2**:

**if** (mfriend==**null**) {

mfriend = **new** FriendFragment();

}

transaction.replace(**R**.**id**.**id\_content**, mfriend);

**break**;

}

transaction.commit();

}

}

## FragmentViewPager

### 普通使用

<android.support.v4.view.ViewPager

android:id=*"@+id/viewpager"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:layout\_below=*"@id/ll"* />

**public** **class** **MainActivity** **extends** **FragmentActivity** **implements** OnClickListener{

**ViewPager** viewPager;

**private** List<**Fragment**> list;

**private** **Button** but1;

**private** **Button** but2;

**private** **Button** but4;

**private** **Button** but3;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

init();

}

**private** **void** **init**() {

**Fragmentone** fragmentone = **new** Fragmentone();

**Fragmenttwo** fragmenttwo = **new** Fragmenttwo();

**Fragmentthree** fragmentthree = **new** Fragmentthree();

**Fragmentfour** fragmentfour = **new** Fragmentfour();

list = **new** ArrayList<**Fragment**>();

list.add(fragmentone);

list.add(fragmenttwo);

list.add(fragmentthree);

list.add(fragmentfour);

but1 = (**Button**) findViewById(**R**.**id**.**btn1**);

but2 = (**Button**) findViewById(**R**.**id**.**btn2**);

but3 = (**Button**) findViewById(**R**.**id**.**btn3**);

but4 = (**Button**) findViewById(**R**.**id**.**btn4**);

but1.setOnClickListener(**this**);

but2.setOnClickListener(**this**);

but3.setOnClickListener(**this**);

but4.setOnClickListener(**this**);

viewPager = (**ViewPager**) findViewById(**R**.**id**.**viewpager**);

viewPager.setAdapter(**new** MyAdapter(getSupportFragmentManager()));

}

@Override

**public** **void** **onClick**(**View** v) {

**switch** (v.getId()) {

**case** **R**.**id**.**btn1**:

viewPager.setCurrentItem(0, **true**);

**break**;

**case** **R**.**id**.**btn2**:

viewPager.setCurrentItem(1, **true**);

**break**;

**case** **R**.**id**.**btn3**:

viewPager.setCurrentItem(2, **true**);

**break**;

**case** **R**.**id**.**btn4**:

viewPager.setCurrentItem(3, **true**);

**break**;

**default**:

**break**;

}

}

**class** **MyAdapter** **extends** **FragmentStatePagerAdapter**{

**public** **MyAdapter**(**FragmentManager** fm) {

**super**(fm);

}

@Override

**public** **Fragment** **getItem**(**int** arg0) {

**return** list.get(arg0);

}

@Override

**public** **int** **getCount**() {

**return** list.size();

}

}

}

### 保持数据实时刷新，包括从子界面返回后

//是否已经在setUserVisibleHint中完成刷新

private var isUserVisibleHintFresh = false

//在不切换fragment情况下（从子界面回来），不会触发setUserVisibleHint，所以需要在onResume中刷新

override fun setUserVisibleHint(isVisibleToUser: Boolean) {

super.setUserVisibleHint(isVisibleToUser)

if (!index.isNullOrBlank()) {

if (isVisibleToUser) {

refreshList()

//如果setUserVisibleHint中刷了，onResume中不要再刷

isUserVisibleHintFresh = true

//setUserVisibleHint会优先于onResume回调，找不到下级方法了，只能手动重置isUserVisibleHintFresh

Handler().postDelayed({ isUserVisibleHintFresh = false }, 300)

}

}

}

override fun onResume() {

super.onResume()

//如果setUserVisibleHint中没有刷，onResume中刷，即子页面回来的情况

if (!isUserVisibleHintFresh) refreshList()

}

## Fragment切换的生命周期

### 通过 add hide show 方式来切换 Fragment

* Fragment1 的生命周期变化为：onCreate（）、onCreateView、onStart（）、onResume（） 回调 onHiddenChanged（） 方法
* Fragment2 的生命周期变化为： onCreate（）、onCreateView、onStart（）、onResume（）
* Fragment 2 再次返回到 Fragment 1：不走任何生命周期方法但是回调 onHiddenChanged（）方法
* 总结：当以这种方式进行 Fragment 1 与 Fragment 2 的切换时，Fragment 隐藏的时候并不走 onDestroyView，所有的显示也不会走 onCreateView 方法，所有的 view 都会保存在内存

### 使用 replace 的方法进行切换时

* 载入Fragment 1时：Fragment 1的生命周期：onCreate（）、onCreateView（）、onStart（）、onResume（）
* 切换到Fragment2时：
  + Fragment 1的生命周期：onPause（）、onStop()、onDestroyView（）、onDestroy（）；
  + Fragment 2的生命周期：onCreate（）、onCreateV（）、onStart（）、onResume（）
* Fragment 2切换回Fragment 1时：
  + Fragment2的生命周期：onPause（）、onStop()、onDestroyView（）、onDestroy（）；
  + Fragment 1的生命周期：onCreate（）、onCreateV（）、onStart（）、onResume（）
* 总结：通过 replace 方法进行替换的时，Fragment 都是进行了销毁，重建的过程，相当于走了一整套的生命周期

### 锁屏

* 进行锁屏操作：Fragment 的生命周期方法：onPause（）、onSaveInstanceState（）、onStop（）。
* 进行解锁操作：onStart（）、 onResume（）

### 使用 ViewPager 进行切换时

* 当使用 ViewPager 与 Fragment 进行切换时，Fragment 会进行预加载操作，所有的 Fragment 都会提前初始--->预加载；
* 初始化时 Fragment 们的生命周期：
  + Fragment 1 的生命周期：onCreate（）、onCreateView（）
  + Fragment 2 的生命周期：onCreate（）、 onCreateView（）
* Fragment 1 切换到 Fragment 2 的生命周期：
  + Fragment 1 ：不走任何生命周期；
  + Fragment 2 ：走 setUserVisVleHint（）方法
* 切回去也是一样的
* 注意： setUserVisVleHint（）方法在 Fragment 1 第一次加载的时候不走，只有在切换的时候 走该方法

/\*\*

\* 第一个 Fragment 需要在onActivityCreated中处理 setUserVisVleHint（）方法，设置为

\* setUserVisibleHint(true); 否则会产空指针异常。因为主动调用 setUserVisibleHint（）方法来控制第一次不会

\* 调用setUserVisibleHint方法的问题，因此 setUserVisVleHint（）方法的优先级高于了 onCreate（）方法。

\* 所以要对是否调用了onCreateView（）方法进行标记判断。

\*/

@Override

public void onActivityCreated(Bundle savedInstanceState) {

setUserVisibleHint(true);

super.onActivityCreated(savedInstanceState);

}

/\*\*

\* 标志位，标志已经初始化完成

\*/

private boolean isPrepared;

@Override

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

View view = LayoutInflater.from(getActivity()).inflate(R.layout.fragment\_a, container, false);

//已经初始化

isPrepared = true;

return view;

}

@Override

public void setUserVisibleHint(boolean isVisibleToUser) {

super.setUserVisibleHint(isVisibleToUser);

//可见的并且是初始化之后才加载

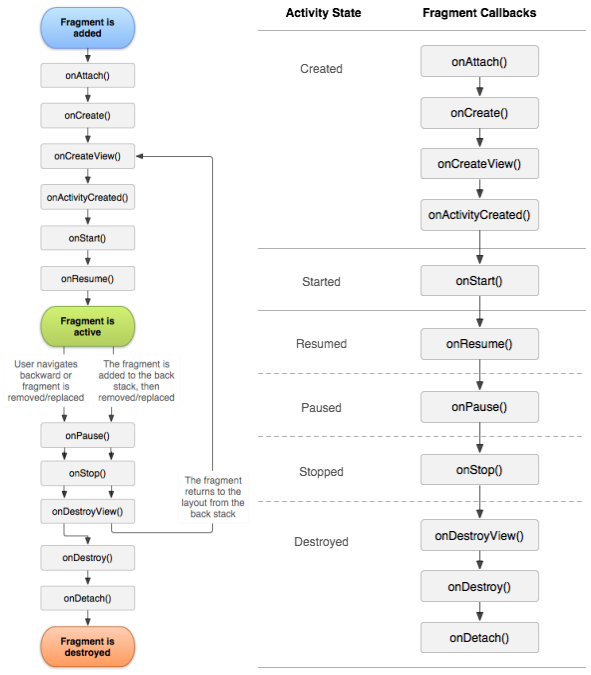
if (isPrepared && isVisibleToUser) {

getList（）；

}

}

## Fragment和Activity生命周期以及通信



### 调用顺序，粗略的看了一下源码

setUserVisibleHint -> performPendingDeferredStart -> moveToState -> performCreateView -> onCreateView -> onViewCreated -> performActivityCreated -> onActivityCreated

public void setUserVisibleHint(boolean isVisibleToUser) {

if (!mUserVisibleHint && isVisibleToUser && mState<STARTED && mFragmentManager != null&&isAdded()) {

mFragmentManager.performPendingDeferredStart(this);

}

...

}

public void performPendingDeferredStart(Fragment f) {

if (f.mDeferStart) {

if (mExecutingActions) {

...

}

f.mDeferStart = false;

moveToState(f, mCurState, 0, 0, false);

}

}

void moveToState(Fragment f, int newState, int transit, int transitionStyle, boolean keepActive) {

...

f.mView = f.performCreateView(f.getLayoutInflater(f.mSavedFragmentState), null, f.mSavedFragmentState);

f.onViewCreated(f.mView, f.mSavedFragmentState);

f.performActivityCreated(f.mSavedFragmentState);

}

View performCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

if (mChildFragmentManager != null) {

mChildFragmentManager.noteStateNotSaved();

}

return onCreateView(inflater, container, savedInstanceState);

}

void performActivityCreated(Bundle savedInstanceState) {

...

onActivityCreated(savedInstanceState);

}

### 具体操作

切换到该Fragment

11-29 14:26:35.095: D/AppListFragment(7649): onAttach  
11-29 14:26:35.095: D/AppListFragment(7649): onCreate  
11-29 14:26:35.095: D/AppListFragment(7649): onCreateView  
11-29 14:26:35.100: D/AppListFragment(7649): onActivityCreated  
11-29 14:26:35.120: D/AppListFragment(7649): onStart  
11-29 14:26:35.120: D/AppListFragment(7649): onResume

屏幕灭掉：

11-29 14:27:35.185: D/AppListFragment(7649): onPause  
11-29 14:27:35.205: D/AppListFragment(7649): onSaveInstanceState  
11-29 14:27:35.205: D/AppListFragment(7649): onStop

屏幕解锁

11-29 14:33:13.240: D/AppListFragment(7649): onStart  
11-29 14:33:13.275: D/AppListFragment(7649): onResume

切换到其他Fragment:  
11-29 14:33:33.655: D/AppListFragment(7649): onPause  
11-29 14:33:33.655: D/AppListFragment(7649): onStop  
11-29 14:33:33.660: D/AppListFragment(7649): onDestroyView

切换回本身的Fragment:

11-29 14:33:55.820: D/AppListFragment(7649): onCreateView  
11-29 14:33:55.825: D/AppListFragment(7649): onActivityCreated  
11-29 14:33:55.825: D/AppListFragment(7649): onStart  
11-29 14:33:55.825: D/AppListFragment(7649): onResume

回到桌面

11-29 14:34:26.590: D/AppListFragment(7649): onPause  
11-29 14:34:26.880: D/AppListFragment(7649): onSaveInstanceState  
11-29 14:34:26.880: D/AppListFragment(7649): onStop

回到应用

11-29 14:36:51.940: D/AppListFragment(7649): onStart  
11-29 14:36:51.940: D/AppListFragment(7649): onResume

退出应用

11-29 14:37:03.020: D/AppListFragment(7649): onPause  
11-29 14:37:03.155: D/AppListFragment(7649): onStop  
11-29 14:37:03.155: D/AppListFragment(7649): onDestroyView  
11-29 14:37:03.165: D/AppListFragment(7649): onDestroy  
11-29 14:37:03.165: D/AppListFragment(7649): onDetach

## Activity向Fragment传递数据（通过fragment.setArguments(bundle)）

实际上是new一个新的有值的Fragment替换掉了之前的那个

**String** text = edt.getText().toString();

**ContentFragment** fragment = **new** ContentFragment();

**Bundle** bundle = **new** Bundle();

bundle.putString("name", text);

fragment.setArguments(bundle);

beginTransaction = getFragmentManager().beginTransaction();//每次都要初始化，因为commit只能一次，想再提交，只能新建

beginTransaction.replace(**R**.**id**.**ll**, fragment, "fragment");

beginTransaction.commit();

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

**View** view = inflater.inflate(**R**.**layout**.**fragment\_content**, container,**false**);

**TextView** tv = (**TextView**) view.findViewById(**R**.**id**.**fra\_tv**);

**Bundle** bundle = getArguments();

**String** text = bundle.getString("name");

tv.setText(text);

**Toast**.*makeText*(getActivity(), text, **Toast**.**LENGTH\_LONG**).show();

**return** view;

}

## Fragment向Activity传递数据

**1.可以用标准的回调**

**public** **interface** MyListener{

**public** **void** **thank**(**String** code);

}

**private** MyListener mListener;

**public** **void** **setListener**(MyListener listener){

**this**.mListener = listener;

}

**2.可以用onAttach()，在里边把绑定了的实现了接口的Activity赋值给Fragmet中的的listener。**

**public** **interface** MyListener{

**public** **void** **thank**(**String** code);

}

**private** MyListener mListener;

@Override

**public** **void** **onAttach**(**Activity** activity) {

mListener = (MyListener) activity;

**super**.onAttach(activity);

}

**public** **class** **MainActivity** **extends** **FragmentActivity** **implements** **MyListener**{

@Override

**public** **void** **thank**(**String** code) {

**Toast**.*makeText*(**MainActivity**.**this**, code, **Toast**.**LENGTH\_LONG**).show();

}

}

**3.可以静态传值**

**private** **String** aaa;

**public** **String** **getAaa**() {

**return** aaa;

}

**public** **void** **setAaa**(**String** aaa) {

**this**.aaa = aaa;

}

@Override

**public** **void** **onClick**(**View** v) {

**String** text = getAaa();

tv.setText(text);

}

**TitleFragment** titleFragment = (**TitleFragment**) fragmentManager.findFragmentById(**R**.**id**.**fragment\_title**);

titleFragment.setAaa("aaaa");

## Fragment回退栈



## 在频繁切换Fragment的情况中，List的notify不起作用

当一个Fragment在来回切换时，分别调用的方法如下：

第一次显示：onCreate() -> onCreateView() -> onActivityCreated()

第二次显示：onCreateView() -> onActivityCreated()

这里不难理解，因为Fragment一般使用的时候会被缓存，所以，当第二次显示的时候，不会调用onCreate()。只会调用onCreateView()和onActivityCreated()，这也就是RecyclerView地址不一样的原因所在，因为RecyclerView的获取操作在onCreateView()中，而Fragment的每次显示都会调用onCreateView()，所以RecyclerView控件会被再次获取，即重新创建一个对象（此时hashCode就变化了）。

结论：所以，在Fragment中使用RecyclerView或AbsListView控件的notifyDataSetChanged()方法时，除了保证数据(mData对象)不能变以外，控件本身一样也不能变。

解决方案：

方案一：因为Fragment的onCreateView()和onActivityCreated()方法在每次Fragment显示的时候会被调用，控件会被重新创建一次，所以，解决方法只能是在这两个方法中重新对RecyclerView设置适配器，而不要使用notifyDataSetChanged()，故代码改为如下：

public void initData() {

if (mData == null) {

mData = new ArrayList<>();

}

mData.clear();

...

数据填充

...

if (mAdapter == null) {

mAdapter = new LQRAdapterForRecyclerView<String>(getActivity(), mData, R.layout.item\_senior) {

@Override

public void convert(LQRViewHolderForRecyclerView helper, String item, int position) {

...

视图填充

...

}

};

}

mRvList.setAdapter(mAdapter);

}

注：只是建议不要在上述两个生命周期方法中使用notifyDataSetChanged()而已，只要在保证RecyclerView等列表控件设置完适配器后，可以在任意地方继续使用notifyDataSetChanged()。

方案二：让Fragment的rootView作为全局变量，在回调onCreateView()时不再重新创建。

public abstract class BaseFragment extends Fragment {

View rootView;

@Nullable

@Override

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

//子类不再需要设置布局ID

if(rootView == null){

rootView = inflater.inflate(provideContentViewId(), container, false);

...

}

return rootView;

}

}

## Fragment中嵌套Fragment

不能使用mActivity.supportFragmentManager()获取出来的FragmentManager。Fragment中嵌套Fragment时，要使用getChildFragmentManager来控制内嵌的Fragment

class FindCarFragment : BaseFragment<FragmentFindCarBinding, FindCarViewModel>() {

...  
 override fun initView() {  
 ...  
 /\*不能使用mActivity.supportFragmentManager()获取出来的FragmentManager  
 Fragment中嵌套Fragment时，要使用getChildFragmentManager来控制内嵌的Fragment\*/  
 val adapter: FragmentPagerAdapter = FindCarFragmentAdapter(*childFragmentManager*)  
 mBinding.vp.*adapter* = adapter  
 ...  
 }

}

## Kotlinx在fragment中直接引用控件报空指针

Fragment引用控件id 有2种方法

1. 在onCreateView（）通过传统findViewById方法

if (view == null){

view = inflater.inflate(initView(), container, false);

}

tv\_show = view?.findViewById(R.id.tv\_show) as TextView.

2.在onViewCreated或之后的方法（比如onActivityCreated、onViewCreated）中使用kotlinx

这是因为如果在onCreateView使用，由于xml没有载入，会导致使用id的时候会报空指针

# Cache缓存详解

## LruCache缓存使用

### 代码

**public** **class** **BitmapCache** **implements** ImageCache {

**public** **LruCache**<**String**, **Bitmap**> cache;

**public** **BitmapCache**() {

**int** maxMemory = (**int**) **Runtime**.*getRuntime*().maxMemory();**//获取最大可用内存**

**int** max = maxMemory / 4;

cache = **new** LruCache<**String**, **Bitmap**>(max){

@Override

**protected** **int** **sizeOf**(**String** key, **Bitmap** value) {**//每次存入缓存时计算存入对象的大小**

**//return value.getByteCount();等同value.getRowBytes() \* value.getHeight();**

**return** value.getByteCount();

}

};

}

@Override

**public** **Bitmap** **getBitmap**(**String** url) {

**return** cache.get(url);**//通过url获取bitmap**

}

@Override

**public** **void** **putBitmap**(**String** url, **Bitmap** bitmap) {

cache.put(url, bitmap);**//通过url获存储bitmap**

}

}

### 使用Volley

**private** **ImageLoader** loader;

**private** **BitmapCache** mCache;**//保证只有一个cache，才能实现缓存**

loader = **new** ImageLoader(**MyApplication**.*getHttpQueues*(), mCache);

### 手动存取

**private** **void** **getImg**(**ImageView** img, **String** url) {

**Bitmap** bitmap = mCache.getBitmap(url);

**if** (bitmap != **null**) {

img.setImageBitmap(bitmap);

}**else** {

getImgUrl(img, url);

}

}

@Override

**public** **void** **onResponse**(**Bitmap** response) {

img.setImageBitmap(response);

mCache.putBitmap(url, response);

}

## DiskLruCache硬盘缓存

### 概述

DiskLruCache并没有限制数据的缓存位置，可以自由地进行设定，但是通常情况下多数应用程序都会将缓存的位置选择为 /sdcard/Android/data/<application package>/cache 这个路径。选择在这个位置有两点好处：第一，这是存储在SD卡上的，因此即使缓存再多的数据也不会对手机的内置存储空间有任何影响，只要SD卡空间足够就行。第二，这个路径被Android系统认定为应用程序的缓存路径，当程序被卸载的时候，这里的数据也会一起被清除掉，这样就不会出现删除程序之后手机上还有很多残留数据的问题。

然后你将会看到一堆文件名很长的文件，这些文件命名没有任何规则，完全看不懂是什么意思，但如果你一直向下滚动，将会看到一个名为journal的文件那些文件名很长的文件就是一张张缓存的图片，每个文件都对应着一张图片，而journal文件是DiskLruCache的一个日志文件，程序对每张图片的操作记录都存放在这个文件中，基本上看到journal这个文件就标志着该程序使用DiskLruCache技术了。

下载DiskLruCache的源码。下载好了源码之后，只需要在项目中新建一个libcore.io包，然后将DiskLruCache.java文件复制到这个包中即可。

### 打开缓存

首先你要知道，DiskLruCache是不能new出实例的，如果我们要创建一个DiskLruCache的实例，则需要调用它的open()方法，接口如下所示：public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)。 open()方法接收四个参数，第一个参数指定的是数据的缓存地址，第二个参数指定当前应用程序的版本号，第三个参数指定同一个key可以对应多少个缓存文件，基本都是传1，第四个参数指定最多可以缓存多少字节的数据。

其中缓存地址前面已经说过了，通常都会存放在 /sdcard/Android/data/<application package>/cache 这个路径下面，但同时我们又需要考虑如果这个手机没有SD卡，或者SD正好被移除了的情况，因此比较优秀的程序都会专门写一个方法来获取缓存地址，如下所示：

//根据传入的uniqueName获取硬盘缓存的路径地址。

**public** **File** **getDiskCacheDir**(**Context** context, **String** uniqueName) {

**String** cachePath;

**if** (**Environment**.**MEDIA\_MOUNTED**.equals(**Environment**.*getExternalStorageState*()) || !**Environment**.*isExternalStorageRemovable*()) {

cachePath = context.getExternalCacheDir().getPath();

} **else** {

cachePath = context.getCacheDir().getPath();

}

**return** **new** File(cachePath + **File**.**separator** + uniqueName);

}

当SD卡存在或者SD卡不可被移除的时候，就调用getExternalCacheDir()方法来获取缓存路径，否则就调用getCacheDir()方法来获取缓存路径。前者获取到的就是 /sdcard/Android/data/<application package>/cache 这个路径，而后者获取到的是 /data/data/<application package>/cache 这个路径。接着又将获取到的路径和一个uniqueName进行拼接，作为最终的缓存路径返回。那么这个uniqueName又是什么呢？其实这就是为了对不同类型的数据进行区分而设定的一个唯一值，比如说在网易新闻缓存路径下看到的bitmap、object等文件夹。接着是应用程序版本号，我们可以使用如下代码简单地获取到当前应用程序的版本号：

//获取版本号

**public** **int** **getAppVersion**(**Context** context) {

**try** {

**PackageInfo** info = context.getPackageManager().getPackageInfo(context.getPackageName(), 0);

**return** info.versionCode;

} **catch** (**NameNotFoundException** e) {

e.printStackTrace();

}

**return** 1;

}

需要注意的是，每当版本号改变，缓存路径下存储的所有数据都会被清除掉，因为DiskLruCache认为当应用程序有版本更新的时候，所有的数据都应该从网上重新获取。后面两个参数就没什么需要解释的了，第三个参数传1，第四个参数通常传入10M的大小就够了，这个可以根据自身的情况进行调节。因此，一个非常标准的open()方法就可以这样写：

**DiskLruCache** mDiskLruCache = null;

**try** {

**File** cacheDir = getDiskCacheDir(context, "thumb");

**if** (!cacheDir.exists()) {

cacheDir.mkdirs();

}

mDiskLruCache = **DiskLruCache**.*open*(cacheDir, getAppVersion(context), 1, 10 \* 1024 \* 1024);

} **catch** (**IOException** e) {

e.printStackTrace();

}

### 写入缓存

先来看写入，比如说现在有一张图片，地址是http://img.my.csdn.net/uploads/201309/01/1378037235\_7476.jpg，那么为了将这张图片下载下来，就可以这样写：

**private** **boolean** **downloadUrlToStream**(**String** urlString, **OutputStream** outputStream) {

**HttpURLConnection** urlConnection = **null**;

**BufferedOutputStream** out = **null**;

**BufferedInputStream** in = **null**;

**try** {

**final** **URL** url = **new** URL(urlString);

urlConnection = (**HttpURLConnection**) url.openConnection();

in = **new** BufferedInputStream(urlConnection.getInputStream(), 8 \* 1024);

out = **new** BufferedOutputStream(outputStream, 8 \* 1024);

**int** b;

**while** ((b = in.read()) != -1) {

out.write(b);

}

**return** **true**;

} **catch** (**final** **IOException** e) {

e.printStackTrace();

} **finally** {

**if** (urlConnection != **null**) {

urlConnection.disconnect();

}

**try** {

**if** (out != **null**) {

out.close();

}

**if** (in != **null**) {

in.close();

}

} **catch** (**final** **IOException** e) {

e.printStackTrace();

}

}

**return** **false**;

}

有了这个方法之后，下面我们就可以使用DiskLruCache来进行写入了，写入的操作是借助DiskLruCache.Editor这个类完成的。类似地，这个类也是不能new的，需要调用DiskLruCache的edit()方法来获取实例，接口如下所示public Editor edit(String key) throws IOException

可以看到，edit()方法接收一个参数key，这个key将会成为缓存文件的文件名，并且必须要和图片的URL是一一对应的。那么怎样才能让key和图片的URL能够一一对应呢？直接使用URL来作为key？不太合适，因为图片URL中可能包含一些特殊字符，这些字符有可能在命名文件时是不合法的。其实最简单的做法就是将图片的URL进行MD5编码，编码后的字符串肯定是唯一的，并且只会包含0-F这样的字符，完全符合文件的命名规则。那么我们就写一个方法用来将字符串进行MD5编码，代码如下所示：

**public** **String** **hashKeyForDisk**(**String** key) {

**String** cacheKey;

**try** {

**final** **MessageDigest** mDigest = **MessageDigest**.*getInstance*("MD5");

mDigest.update(key.getBytes());

cacheKey = bytesToHexString(mDigest.digest());

} **catch** (**NoSuchAlgorithmException** e) {

cacheKey = **String**.*valueOf*(key.hashCode());

}

**return** cacheKey;

}

**private** **String** **bytesToHexString**(**byte**[] bytes) {

**StringBuilder** sb = **new** StringBuilder();

**for** (**int** i = 0; i < bytes.length; i++) {

**String** hex = **Integer**.*toHexString*(0xFF & bytes[i]);

**if** (hex.length() == 1) {

sb.append('0');

}

sb.append(hex);

}

**return** sb.toString();

}

因此，现在就可以这样写来得到一个DiskLruCache.Editor的实例：

**String** imageUrl = "http://img.my.csdn.net/uploads/201309/01/1378037235\_7476.jpg";

**String** key = hashKeyForDisk(imageUrl);

**DiskLruCache**.**Editor** editor = mDiskLruCache.edit(key);

有了DiskLruCache.Editor的实例之后，我们可以调用它的newOutputStream()方法来创建一个输出流，然后把它传入downloadUrlToStream()中就能实现下载并写入缓存的功能了。注意newOutputStream()方法接收一个index参数，由于前面在设valueCount的时候指定的是1，所以这里index传0就可以了。在写入操作执行完之后，我们还需要调用一下commit()方法进行提交才能使写入生效，调用abort()方法的话则表示放弃此次写入。因此，一次完整写入操作的代码如下所示：

**new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**try** {

**String** imageUrl = "http://img.my.csdn.net/uploads/201309/01/1378037235\_7476.jpg";

**String** key = hashKeyForDisk(imageUrl);

**DiskLruCache**.**Editor** editor = mDiskLruCache.edit(key);

**if** (editor != **null**) {

**OutputStream** outputStream = editor.newOutputStream(0); **//创建输出流，传给下载图片的输出流**

**if** (downloadUrlToStream(imageUrl, outputStream)) {

editor.commit();

} **else** {

editor.abort();

}

}

mDiskLruCache.flush();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

}).start();

由于这里调用了downloadUrlToStream()方法来从网络上下载图片，所以一定要确保这段代码是在子线程当中执行的。注意在代码的最后我还调用了一下flush()方法，这个方法并不是每次写入都必须要调用的，但在这里却不可缺少，我会在后面说明它的作用。现在的话缓存应该是已经成功写入了

### 读取缓存

缓存已经写入成功之后，接下来我们就该学习一下如何读取了。读取的方法要比写入简单一些，主要是借助DiskLruCache的get()方法实现的，接口如下所示：public synchronized Snapshot get(String key) throws IOException。很明显，get()方法要求传入一个key来获取到相应的缓存数据，而这个key毫无疑问就是将图片URL进行MD5编码后的值了，获取到的是一个DiskLruCache.Snapshot对象，只需要调用它的getInputStream()方法就可以得到缓存文件的输入流了。同样地，getInputStream()方法也需要传一个index参数，这里传入0就好。有了文件的输入流之后，想要把缓存图片显示到界面上就轻而易举了。所以，一段完整的读取缓存，并将图片加载到界面上的代码如下所示：

**try** {

**String** imageUrl = "http://img.my.csdn.net/uploads/201309/01/1378037235\_7476.jpg";

**String** key = hashKeyForDisk(imageUrl);

**DiskLruCache**.**Snapshot** snapShot = mDiskLruCache.get(key);

**if** (snapShot != **null**) {

**InputStream** is = snapShot.getInputStream(0);

**Bitmap** bitmap = **BitmapFactory**.*decodeStream*(is);

mImage.setImageBitmap(bitmap);

}

} **catch** (**IOException** e) {

e.printStackTrace();

}

### 移除缓存

学习完了写入缓存和读取缓存的方法之后，最难的两个操作你就都已经掌握了，那么接下来要学习的移除缓存对你来说也一定非常轻松了。移除缓存主要是借助DiskLruCache的remove()方法实现的，接口如下所示：public synchronized boolean remove(String key) throws IOException

**try** {

**String** imageUrl = "http://img.my.csdn.net/uploads/201309/01/1378037235\_7476.jpg";

**String** key = hashKeyForDisk(imageUrl);

mDiskLruCache.remove(key);

} **catch** (**IOException** e) {

e.printStackTrace();

}

### 其它API

1. size()

这个方法会返回当前缓存路径下所有缓存数据的总字节数，以byte为单位，如果应用程序中需要在界面上显示当前缓存数据的总大小，就可以通过调用这个方法计算出来

2.flush()

这个方法用于将内存中的操作记录同步到日志文件（也就是journal文件）当中。这个方法非常重要，因为DiskLruCache能够正常工作的前提就是要依赖于journal文件中的内容。前面在讲解写入缓存操作的时候我有调用过一次这个方法，但其实并不是每次写入缓存都要调用一次flush()方法的，频繁地调用并不会带来任何好处，只会额外增加同步journal文件的时间。比较标准的做法就是在Activity的onPause()方法中去调用一次flush()方法就可以了。

3.close()

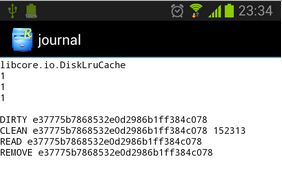
这个方法用于将DiskLruCache关闭掉，是和open()方法对应的一个方法。关闭掉了之后就不能再调用DiskLruCache中任何操作缓存数据的方法，通常只应该在Activity的onDestroy()方法中去调用close()方法。

4.delete()

这个方法用于将所有的缓存数据全部删除，比如说网易新闻中的那个手动清理缓存功能，其实只需要调用一下DiskLruCache的delete()方法就可以实现了。

### 解读journal

前面已经提到过，DiskLruCache能够正常工作的前提就是要依赖于journal文件中的内容，因此，能够读懂journal文件对于我们理解DiskLruCache的工作原理有着非常重要的作用。那么journal文件中的内容到底是什么样的呢？



由于现在只缓存了一张图片，所以journal中并没有几行日志，我们一行行进行分析。第一行是个固定的字符串“libcore.io.DiskLruCache”，标志着我们使用的是DiskLruCache技术。第二行是DiskLruCache的版本号，这个值是恒为1的。第三行是应用程序的版本号，我们在open()方法里传入的版本号是什么这里就会显示什么。第四行是valueCount，这个值也是在open()方法中传入的，通常情况下都为1。第五行是一个空行。前五行也被称为journal文件的头，这部分内容还是比较好理解的，但是接下来的部分就要稍微动点脑筋了。

第六行是以一个DIRTY前缀开始的，后面紧跟着缓存图片的key。通常我们看到DIRTY这个字样都不代表着什么好事情，意味着这是一条脏数据。没错，每当我们调用一次DiskLruCache的edit()方法时，都会向journal文件中写入一条DIRTY记录，表示我们正准备写入一条缓存数据，但不知结果如何。然后调用commit()方法表示写入缓存成功，这时会向journal中写入一条CLEAN记录，意味着这条“脏”数据被“洗干净了”，调用abort()方法表示写入缓存失败，这时会向journal中写入一条REMOVE记录。也就是说，每一行DIRTY的key，后面都应该有一行对应的CLEAN或者REMOVE的记录，否则这条数据就是“脏”的，会被自动删除掉。

如果你足够细心的话应该还会注意到，第七行的那条记录，除了CLEAN前缀和key之外，后面还有一个152313，这是什么意思呢？其实，DiskLruCache会在每一行CLEAN记录的最后加上该条缓存数据的大小，以字节为单位。152313也就是我们缓存的那张图片的字节数了

前面我们所学的size()方法可以获取到当前缓存路径下所有缓存数据的总字节数，其实它的工作原理就是把journal文件中所有CLEAN记录的字节数相加，求出的总合再把它返回而已。

除了DIRTY、CLEAN、REMOVE之外，还有一种前缀是READ的记录，这个就非常简单了，每当我们调用get()方法去读取一条缓存数据时，就会向journal文件中写入一条READ记录。因此，像网易新闻这种图片和数据量都非常大的程序，journal文件中就可能会有大量的READ记录。

那么你可能会担心了，如果我不停频繁操作的话，就会不断地向journal文件中写入数据，那这样journal文件岂不是会越来越大？这倒不必担心，DiskLruCache中使用了一个redundantOpCount变量来记录用户操作的次数，每执行一次写入、读取或移除缓存的操作，这个变量值都会加1，当变量值达到2000的时候就会触发重构journal的事件，这时会自动把journal中一些多余的、不必要的记录全部清除掉，保证journal文件的大小始终保持在一个合理的范围内。

# 使用线程池处理异步任务

## 概述

说到线程，大家都不陌生，因为在开发时候或多或少都会用到线程，而通常创建线程有两种方式：

* 继承Thread类
* 实现Runnable接口

虽说这两种方式都可以创建出一个线程，不过它们之间还是有一点区别的，主要区别在于在多线程访问同一资源的情况下，用Runnable接口创建的线程可以处理同一资源，而用Thread类创建的线程则各自独立处理，各自拥有自己的资源。所以，在Java中大多数多线程程序都是通过实现Runnable来完成的，而对于Android来说也不例外，当涉及到需要开启线程去完成某件事时，我们都会这样写：

new Thread(new Runnable() {

@Override

public void run() {

//do sth .

}

}).start();

这段代码创建了一个线程并执行，它在任务结束后GC会自动回收该线程，一切看起来如此美妙，是的，它在线程并发不多的程序中确实不错，而假如这个程序有很多地方需要开启大量线程来处理任务，那么如果还是用上述的方式去创建线程处理的话，那么将导致系统的性能表现的非常糟糕，更别说在内存有限的移动设备上，主要的影响如下：

* 线程的创建和销毁都需要时间，当有大量的线程创建和销毁时，那么这些时间的消耗则比较明显，将导致性能上的缺失
* 大量的线程创建、执行和销毁是非常耗cpu和内存的，这样将直接影响系统的吞吐量，导致性能急剧下降，如果内存资源占用的比较多，还很可能造成OOM
* 大量的线程的创建和销毁很容易导致GC频繁的执行，从而发生内存抖动现象，而发生了内存抖动，对于移动端来说，最大的影响就是造成界面卡顿

而针对上述所描述的问题，解决的办法归根到底就是：重用已有的线程，从而减少线程的创建。 所以这就涉及到线程池（ExecutorService）的概念了，线程池的基本作用就是进行线程的复用，下面将具体介绍线程池的使用

## ExecutorService

通过上述分析，我们知道了通过new Thread().start()方式创建线程去处理任务的弊端，而为了解决这些问题，Java为我们提供了ExecutorService线程池来优化和管理线程的使用

### 使用线程池管理线程的优点

* 线程的创建和销毁由线程池维护，一个线程在完成任务后并不会立即销毁，而是由后续的任务复用这个线程，从而减少线程的创建和销毁，节约系统的开销
* 线程池旨在线程的复用，这就可以节约我们用以往的方式创建线程和销毁所消耗的时间，减少线程频繁调度的开销，从而节约系统资源，提高系统吞吐量
* 在执行大量异步任务时提高了性能
* Java内置的一套ExecutorService线程池相关的api，可以更方便的控制线程的最大并发数、线程的定时任务、单线程的顺序执行等

### ExecutorService简介

通常来说我们说到线程池第一时间想到的就是它：ExecutorService，它是一个接口，其实如果要从真正意义上来说，它可以叫做线程池的服务，因为它提供了众多接口api来控制线程池中的线程，而真正意义上的线程池就是：ThreadPoolExecutor，它实现了ExecutorService接口，并封装了一系列的api使得它具有线程池的特性，其中包括工作队列、核心线程数、最大线程数等。

## 线程池：ThreadPoolExecutor

### 五种线程池

既然线程池就是ThreadPoolExecutor，所以我们要创建一个线程池只需要new ThreadPoolExecutor(…);就可以创建一个线程池，而如果这样创建线程池的话，我们需要配置一堆东西，非常麻烦，我们可以看一下它的构造方法就知道了：

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, Time Unitunit,

BlockingQueue workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler){…}

所以，官方也不推荐使用这种方法来创建线程池，而是推荐使用Executors的工厂方法来创建线程池，Executors类是官方提供的一个工厂类，它里面封装好了众多功能不一样的线程池，从而使得我们创建线程池非常的简便，主要提供了如下五种功能不一样的线程池：

**1、newFixedThreadPool() ：**

作用：该方法返回一个固定线程数量的线程池，该线程池中的线程数量始终不变，即不会再创建新的线程，也不会销毁已经创建好的线程，自始自终都是那几个固定的线程在工作，所以该线程池可以控制线程的最大并发数。

栗子：假如有一个新任务提交时，线程池中如果有空闲的线程则立即使用空闲线程来处理任务，如果没有，则会把这个新任务存在一个任务队列中，一旦有线程空闲了，则按FIFO方式处理任务队列中的任务。

**2、newCachedThreadPool() ：**

作用：该方法返回一个可以根据实际情况调整线程池中线程的数量的线程池。即该线程池中的线程数量不确定，是根据实际情况动态调整的。

栗子：假如该线程池中的所有线程都正在工作，而此时有新任务提交，那么将会创建新的线程去处理该任务，而此时假如之前有一些线程完成了任务，现在又有新任务提交，那么将不会创建新线程去处理，而是复用空闲的线程去处理新任务。那么此时有人有疑问了，那这样来说该线程池的线程岂不是会越集越多？其实并不会，因为线程池中的线程都有一个“保持活动时间”的参数，通过配置它，如果线程池中的空闲线程的空闲时间超过该“保存活动时间”则立刻停止该线程，而该线程池默认的“保持活动时间”为60s。

**3、newSingleThreadExecutor() ：**

作用：该方法返回一个只有一个线程的线程池，即每次只能执行一个线程任务，多余的任务会保存到一个任务队列中，等待这一个线程空闲，当这个线程空闲了再按FIFO方式顺序执行任务队列中的任务。

**4、newScheduledThreadPool() ：**

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。

**5、newSingleThreadScheduledExecutor() ：**

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。只不过和上面的区别是该线程池大小为1，而上面的可以指定线程池的大小。

### 通过Executors的工厂方法来获取这五种线程池

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

ScheduledExecutorService singleThreadScheduledPool = Executors.newSingleThreadScheduledExecutor();

我们可以看到通过Executors的工厂方法来创建线程池极其简便，其实它的内部还是通过new ThreadPoolExecutor(…)的方式创建线程池的，我们看一下这些工厂方法的内部实现：

Public static ExecutorService newFixedThreadPool(int nThreads){

Return newThreadPoolExecutor(nThreads,nThreads,

0L,TimeUnit.MILLISECONDS,

newLinkedBlockingQueue());

}

Public static ExecutorService newSingleThreadExecutor(){

Return newFinalizableDelegatedExecutorService

(newThreadPoolExecutor(1,1,

0L,TimeUnit.MILLISECONDS,

newLinkedBlockingQueue()));

}

Public static ExecutorService newCachedThreadPool(){

Return newThreadPoolExecutor(0,Integer.MAX\_VALUE,

60L,TimeUnit.SECONDS,

newSynchronousQueue());

}

我们可以清楚的看到这些方法的内部实现都是通过创建一个ThreadPoolExecutor对象来创建的，正所谓万变不离其宗，所以我们要了解线程池还是得了解ThreadPoolExecutor这个线程池类，其中由于和定时任务相关的线程池比较特殊（newScheduledThreadPool()、newSingleThreadScheduledExecutor()），它们创建的线程池内部实现是由ScheduledThreadPoolExecutor这个类实现的，而ScheduledThreadPoolExecutor是继承于ThreadPoolExecutor扩展而成的，所以本质还是一样的，只不过多封装了一些定时任务相关的api，所以我们主要就是要了解ThreadPoolExecutor，从构造方法开始：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {//...}

我们可以看到它构造方法的参数比较多，有七个，下面一一来说明这些参数的作用：

* corePoolSize：线程池中的核心线程数量
* maximumPoolSize：线程池中的最大线程数量
* keepAliveTime：这个就是上面说到的“保持活动时间“，上面只是大概说明了一下它的作用，不过它起作用必须在一个前提下，就是当线程池中的线程数量超过了corePoolSize时，它表示多余的空闲线程的存活时间，即：多余的空闲线程在超过keepAliveTime时间内没有任务的话则被销毁。而这个主要应用在缓存线程池中
* unit：它是一个枚举类型，表示keepAliveTime的单位，常用的如：TimeUnit.SECONDS（秒）、TimeUnit.MILLISECONDS（毫秒）
* workQueue：任务队列，主要用来存储已经提交但未被执行的任务，不同的线程池采用的排队策略不一样，稍后再讲
* threadFactory：线程工厂，用来创建线程池中的线程，通常用默认的即可
* handler：通常叫做拒绝策略，1、在线程池已经关闭的情况下 2、任务太多导致最大线程数和任务队列已经饱和，无法再接收新的任务 。在上面两种情况下，只要满足其中一种时，在使用execute()来提交新的任务时将会拒绝，而默认的拒绝策略是抛一个RejectedExecutionException异常

上面的参数理解起来都比较简单，不过workQueue这个任务队列却要再次说明一下，它是一个BlockingQueue对象，而泛型则限定它是用来存放Runnable对象的，刚刚上面讲了，不同的线程池它的任务队列实现肯定是不一样的，所以，保证不同线程池有着不同的功能的核心就是这个workQueue的实现了，细心的会发现在刚刚的用来创建线程池的工厂方法中，针对不同的线程池传入的workQueue也不一样，下面我总结一下这五种线程池分别用的是什么BlockingQueue：

1、newFixedThreadPool()—>LinkedBlockingQueue 无界的队列

2、newSingleThreadExecutor()—>LinkedBlockingQueue

3、newCachedThreadPool()—>SynchronousQueue 直接提交的队列

4、newScheduledThreadPool()—>DelayedWorkQueue

5、newSingleThreadScheduledExecutor()—>DelayedWorkQueue 等待队列

当然实现了BlockingQueue接口的队列还有：ArrayBlockingQueue（有界的队列）、PriorityBlockingQueue（优先级队列）。这些队列的详细作用就不多介绍了。

## 线程池ThreadPoolExecutor的使用

使用线程池，其中涉及到一个极其重要的方法，即：

execute(Runnable command)

该方法意为执行给定的任务，该任务处理可能在新的线程、已入池的线程或者正调用的线程，这由ThreadPoolExecutor的实现决定。

### newFixedThreadPool

创建一个固定线程数量的线程池，示例为：

ExecutorService fixedThreadPool = **Executors**.*newFixedThreadPool*(3);

**for** (**int** i = 1; i < 10; i++) {

**final** **int** index = i;

fixedThreadPool.execute(**new** Runnable() {

@Override

**public** **void** **run**() {

**String** threadName = **Thread**.*currentThread*().getName();

**Log**.*v*("zxy", "线程："+ threadName +",正在执行第" + index + "个任务");

**try** {

**Thread**.*sleep*(2000);

} **catch** (**InterruptedException** e) {

e.printStackTrace();

}

}

});

}

上述代码，创建了一个线程数为3的固定线程数量的线程池，同理该线程池支持的线程最大并发数也是3，而我模拟了10个任务让它处理，执行的情况则是首先执行前三个任务，后面7个则依次进入任务队列进行等待，执行完任意一个任务后，再通过FIFO的方式从任务队列中取任务执行，直到最后任务都执行完毕。

### newSingleThreadExecutor

创建一个只有一个线程的线程池，每次只能执行一个线程任务，多余的任务会保存到一个任务队列中，等待线程处理完再依次处理任务队列中的任务，示例为：

Executors.newSingleThreadExecutor()

代码还是差不多，只不过改了线程池的实现方式，效果即依次一个一个的处理任务，而且都是复用一个线程，其实我们通过newSingleThreadExecutor()和newFixedThreadPool()的方法发现，创建一个singleThreadExecutorPool实际上就是创建一个核心线程数和最大线程数都为1的fixedThreadPool。

### newCachedThreadPool

创建一个可以根据实际情况调整线程池中线程的数量的线程池，示例为：

ExecutorService cachedThreadPool = **Executors**.*newCachedThreadPool*();

**for** (**int** i = 1; i < 10; i++) {

**final** **int** index = i;

**try** {

**Thread**.*sleep*(1000);

} **catch** (**InterruptedException** e) {

e.printStackTrace();

}

cachedThreadPool.execute(**new** Runnable() {

@Override

**public** **void** **run**() {

**String** threadName = **Thread**.*currentThread*().getName();

**Log**.*v*("zxy", "线程：" + threadName + ",正在执行第" + index + "个任务");

**try** {

**long** time = index \* 500;

**Thread**.*sleep*(time);

} **catch** (**InterruptedException** e) {

e.printStackTrace();

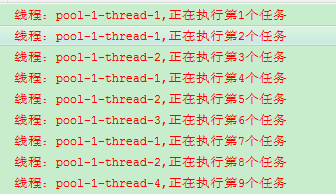
}

}

});

}

为了体现该线程池可以自动根据实现情况进行线程的重用，而不是一味的创建新的线程去处理任务，设置了每隔1s去提交一个新任务，这个新任务执行的时间也是动态变化的，所以，效果为：



### newScheduledThreadPool

创建一个可以定时或者周期性执行任务的线程池，示例为：

ScheduledExecutorService scheduledThreadPool = **Executors**.*newScheduledThreadPool*(3);

//延迟2秒后执行该任务

scheduledThreadPool.schedule(**new** Runnable() {

@Override

**public** **void** **run**() {

}

}, 2, TimeUnit.**SECONDS**);

//延迟1秒后，每隔2秒执行一次该任务

scheduledThreadPool.scheduleAtFixedRate(**new** Runnable() {

@Override

**public** **void** **run**() {

}

}, 1, 2, TimeUnit.**SECONDS**);

### newSingleThreadScheduledExecutor

创建一个可以定时或者周期性执行任务的线程池，该线程池的线程数为1，示例为：

ScheduledExecutorService singleThreadScheduledPool = **Executors**.*newSingleThreadScheduledExecutor*();

//延迟1秒后，每隔2秒执行一次该任务

singleThreadScheduledPool.scheduleAtFixedRate(**new** Runnable() {

@Override

**public** **void** **run**() {

**String** threadName = **Thread**.*currentThread*().getName();

**Log**.*e*("zxy", "线程：" + threadName + ",正在执行");

}

},1,2,TimeUnit.**SECONDS**);

## 自定义线程池ThreadPoolExecutor

Java内置只为我们提供了五种常用的线程池，一般来说这足够用了，不过有时候我们也可以根据需求来自定义我们自己的线程池，而要自定义不同功能的线程池，上面我们也说了线程池功能的不同归根到底还是内部的BlockingQueue实现不同，所以，我们要实现我们自己相要的线程池，就必须从BlockingQueue的实现上做手脚。详情略。。。

# Camera使用相机和获取相册照片

## 一般的拍照，清晰拍照，获取相册，裁剪

**public** **class** **MainActivity** **extends** **Activity** {

**private** **final** **static** **int** **TAKE\_PHOTO** = 1;

**private** **final** **static** **int** **TAKE\_PHOTO2** = 2;

**private** **final** **static** **int** **SHOW\_CROP\_PHOTO** = 3;

**private** **final** **static** **int** **GET\_PHOTO** = 4;

**private** **ImageView** imageView;

**private** **String** mFilePath;

**private** **Uri** phptouUri;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

imageView = (**ImageView**) findViewById(**R**.**id**.**img**);

mFilePath = **Environment**.*getExternalStorageDirectory*().getPath();

mFilePath += "/temp.png";

}

**//初始化自定义照片文件**

**private** **void** **CreatNewOutFile**() {

**File** outputImage = **new** File(mFilePath);

**if** (outputImage.exists()) {

outputImage.delete();

}

**try** {

outputImage.createNewFile();

phptouUri = **Uri**.*fromFile*(outputImage);

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

**//使用相机拍照**

**public** **void** **startCamera**(**View** view) {

**Intent** intent = **new** Intent(**MediaStore**.**ACTION\_IMAGE\_CAPTURE**);

startActivityForResult(intent, **TAKE\_PHOTO**);

}

**//照片放入自己的文件里**

**public** **void** **startCameraclear**(**View** view) {

**CreatNewOutFile**()**//初始化自定义照片文件**

**Intent** intent = **new** Intent(**MediaStore**.**ACTION\_IMAGE\_CAPTURE**);

intent.putExtra(**MediaStore**.**EXTRA\_OUTPUT**, phptouUri);**//将拍照存储路径自己更改，这样任何程序可以访问该路径**

startActivityForResult(intent, **TAKE\_PHOTO2**);

}

**//获取相册**

**public** **void** **getPhoto**(**View** view) {

**CreatNewOutFile**()**//初始化自定义照片文件**

**Intent** intent = **new** Intent(**Intent**.**ACTION\_GET\_CONTENT**, **null**);

intent.setType("image/\*");

startActivityForResult(intent, **GET\_PHOTO**);

}

@Override

**protected** **void** **onActivityResult**(**int** requestCode, **int** resultCode, **Intent** data) {

**super**.onActivityResult(requestCode, resultCode, data);

**Bitmap** bitmap = **null**;

**switch** (requestCode) {

**case** **TAKE\_PHOTO**:**//使用相机拍照**

**if** (data.getExtras() != **null**) {**//先判断，有些手机会空指针**

bitmap = (**Bitmap**) data.getExtras().get("data");**//不是很清晰**

**if** (bitmap != **null**) {

imageView.setImageBitmap(bitmap);

}

}

**break**;

**case** **TAKE\_PHOTO2**:**//照片放入自己的文件里**

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

intent.setDataAndType(phptouUri, "image/\*");**//从自己文件里取照片，开始裁剪**

intent.putExtra("scale", **true**);**//定义裁剪**

**// aspectX aspectY 是宽高的比例**

intent.putExtra("aspectX", 2);

intent.putExtra("aspectY", 1);

**// outputX,outputY 是剪裁图片的宽高**

intent.putExtra("outputX", 320 \* 2);

intent.putExtra("outputY", 320);

intent.putExtra(**MediaStore**.**EXTRA\_OUTPUT**, phptouUri);**//裁剪后放入文件里**

startActivityForResult(intent, **SHOW\_CROP\_PHOTO**);

**break**;

**case** **GET\_PHOTO**:**//获取相册**

**//bitmap = MediaStore.Images.Media.getBitmap(getContentResolver(), data.getData());**

**Intent** intent2 = **new** Intent("com.android.camera.action.CROP");

intent2.setDataAndType(data.getData(), "image/\*");**//从选择的照片里获取照片**

intent2.putExtra("scale", **true**);

intent2.putExtra(**MediaStore**.**EXTRA\_OUTPUT**, phptouUri);

startActivityForResult(intent2, **SHOW\_CROP\_PHOTO**);

**break**;

**case** **SHOW\_CROP\_PHOTO**:**//显示裁剪后的照片**

**try** {

**//bitmap = BitmapFactory.decodeStream(getContentResolver().openInputStream(phptouUri));**

bitmap = **BitmapFactory**.*decodeFile*(mFilePath);**//内存卡上的文件可以读出来**

**if** (bitmap != **null**) {

imageView.setImageBitmap(bitmap);

}

} **catch** (**FileNotFoundException** e) {

e.printStackTrace();

}

**break**;

**default**:

**break**;

}

}

}

## 拍出来的照片“歪了”

拍照时看到照片是正的，但是当我们的 app 获取到这张照片时，却发现旋转了 90 度（也有可能是180、270，不过90度比较多见，貌似都是由于手机传感器导致的），那么通过 ExifInterface 类拿到 TAG\_ORIENTATION 属性对应的值，即为我们想要得到旋转角度。再根据利用 Matrix 进行旋转纠正即可

bitmap = **BitmapFactory**.*decodeStream*(getContentResolver().openInputStream(phptouUri));

**if** (bitmap != **null**) {

**int** degree = getbmdegree();

**if** (degree == 0) {//0度好像旋转后会出问题

imageView.setImageBitmap(bitmap);

}**else** {

**Bitmap** bitmap2 = rotateBmDegree(bitmap, getbmdegree());

imageView.setImageBitmap(bitmap2);

}

}

/\*\*

\* 获取照片旋转角度

\*/

**private** **int** **getbmdegree**(){

**int** degree = 0;

**try** {

**ExifInterface** exifInterface = **new** ExifInterface(mFilePath);

**int** orientation = exifInterface.getAttributeInt(**ExifInterface**.**TAG\_ORIENTATION**, **ExifInterface**.**ORIENTATION\_NORMAL**);

**switch** (orientation) {

**case** **ExifInterface**.**ORIENTATION\_ROTATE\_90**:

degree = 90;

**break**;

**case** **ExifInterface**.**ORIENTATION\_ROTATE\_180**:

degree = 180;

**break**;

**case** **ExifInterface**.**ORIENTATION\_ROTATE\_270**:

degree = 270;

**break**;

**default**:

**break**;

}

} **catch** (**IOException** e) {

e.printStackTrace();

}

**return** degree;

}

/\*\*

\* 根据旋转角度，旋转回来

\*/

**private** **Bitmap** **rotateBmDegree**(**Bitmap** bitmap, **int** degree){

**Matrix** matrix = **new** Matrix();

matrix.postRotate(90);

**Bitmap** newBitmap = **Bitmap**.*createBitmap*(bitmap, 0, 0, bitmap.getWidth(), bitmap.getHeight(), matrix, **true**);

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

bitmap.recycle();

}

**return** newBitmap;

}

## 使用自定义相机

**Manifest中Activity中的<intent-filter>设置**

<action android:name="android.media.action.IMAGE\_CAPTURE"/>

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

**public** **class** **CustomActivity** **extends** **Activity** **implements** SurfaceHolder.Callback{

**private** **Camera** mCamera;

**private** **SurfaceView** mPreView;

**private** SurfaceHolder mHolder;

**private** **Camera**.**PictureCallback** mPictureCallback = **new** **Camera**.**PictureCallback**() {

@Override

**public** **void** **onPictureTaken**(**byte**[] data, **Camera** camera) {

**File** tempFile = **new** File(**Environment**.*getExternalStorageDirectory*().getPath() + "/temp.png");

**try** {

**FileOutputStream** fos = **new** FileOutputStream(tempFile);

fos.write(data);

fos.close();

**Intent** intent = **new** Intent(**CustomActivity**.**this**, **ResultActivity**.**class**);

intent.putExtra("picPath", tempFile.getAbsolutePath());

startActivity(intent);

**CustomActivity**.**this**.finish();

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

};

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_custom**);

mPreView = (**SurfaceView**) findViewById(**R**.**id**.**preview**);

mHolder = mPreView.getHolder();

mHolder.addCallback(**this**);

mPreView.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

mCamera.autoFocus(**null**);**//只是让他对焦，不对结果操作**

}

});

}

**public** **void** **capture**(**View** view){

**Camera**.**Parameters** parameters = mCamera.getParameters();

parameters.setPictureFormat(**ImageFormat**.**JPEG**);**//拍照的格式**

parameters.setPreviewSize(400, 800);**//显示大小**

parameters.setFocusMode(**Camera**.**Parameters**.**FOCUS\_MODE\_AUTO**);**//对焦方式**

mCamera.autoFocus(**new** **AutoFocusCallback**() {

@Override

**public** **void** **onAutoFocus**(**boolean** success, **Camera** camera) {

**if** (success) {**//如果对焦完成**

mCamera.takePicture(**null**, **null**, mPictureCallback);**//拍照，照片传给mPictureCallback**

}

}

});

}

@Override

**protected** **void** **onResume**() {

**super**.onResume();

**if** (mCamera == **null**) {

mCamera = getCamera();

**if** (mHolder != **null**) {

setStartPreview(mCamera, mHolder);

}

}

}

@Override

**protected** **void** **onPause**() {

**super**.onPause();

releaseCamera();

}

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

releaseCamera();

}

/\*\*

\* 获取camera对象

\* **@return**

\*/

**private** **Camera** **getCamera**(){

**Camera** camera;

**try** {

camera = **Camera**.*open*();;**//hardware这个包而不是graphics这个包**

} **catch** (**Exception** e) {

camera = **null**;

e.printStackTrace();

}

**return** camera;

}

/\*\*

\* 开始预览相机内容

\*/

**private** **void** **setStartPreview**(**Camera** camera, SurfaceHolder holder){

**try** {

camera.setPreviewDisplay(holder);**//绑定进holder**

camera.setDisplayOrientation(90);**//默认是横屏，将它竖过来**

camera.startPreview();**//开始预览**

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

/\*\*

\* 释放相机资源

\*/

**private** **void** **releaseCamera**(){

**if** (mCamera != **null**) {

mCamera.setPreviewCallback(**null**);**//设置回调为空，取消camera和surview绑定**

mCamera.stopPreview();

mCamera.release();

mCamera = **null**;

}

}

@Override

**public** **void** **surfaceCreated**(SurfaceHolder holder) {

setStartPreview(mCamera, mHolder);

}

@Override

**public** **void** **surfaceChanged**(SurfaceHolder holder, **int** format, **int** width,

**int** height) {

mCamera.stopPreview();

setStartPreview(mCamera, mHolder);

}

@Override

**public** **void** **surfaceDestroyed**(SurfaceHolder holder) {

releaseCamera();

}

}

# 从相册获取相片

## 通过设置intent直接获取

**Intent** intent = **new** Intent(**Intent**.**ACTION\_GET\_CONTENT**, **null**);

intent.setType("image/\*");

startActivityForResult(intent, **GET\_PHOTO**);

@Override

**protected** **void** **onActivityResult**(**int** requestCode, **int** resultCode, **Intent** data) {

**super**.onActivityResult(requestCode, resultCode, data);

**Bitmap** bitmap = **null**;

**switch** (requestCode) {

**case** **GET\_PHOTO**://获取相册

bitmap = MediaStore.Images.Media.getBitmap(getContentResolver(), data.getData());

imageView.setImageBitmap(bitmap);

**break**;

**default**:

**break**;

}

}

## 查看选择的相册照片的地址

**public** **class** **GetPathUntil** {

**public** **static** **String** **getPath**(**final** **Context** context, **final** **Uri** uri) {

**final** **boolean** isKitKat = **Build**.**VERSION**.**SDK\_INT** >= **Build**.**VERSION\_CODES**.**KITKAT**;

// DocumentProvider

**if** (isKitKat && **DocumentsContract**.*isDocumentUri*(context, uri)) {

// ExternalStorageProvider

**if** (*isExternalStorageDocument*(uri)) {

**final** **String** docId = **DocumentsContract**.*getDocumentId*(uri);

**final** **String**[] split = docId.split(":");

**final** **String** type = split[0];

**if** ("primary".equalsIgnoreCase(type)) {

**return** **Environment**.*getExternalStorageDirectory*() + "/" + split[1];

}

// TODO handle non-primary volumes

}

// DownloadsProvider

**else** **if** (*isDownloadsDocument*(uri)) {

**final** **String** id = **DocumentsContract**.*getDocumentId*(uri);

**final** **Uri** contentUri = **ContentUris**.*withAppendedId*(

**Uri**.*parse*("content://downloads/public\_downloads"), **Long**.*valueOf*(id));

**return** *getDataColumn*(context, contentUri, **null**, **null**);

}

// MediaProvider

**else** **if** (*isMediaDocument*(uri)) {

**final** **String** docId = **DocumentsContract**.*getDocumentId*(uri);

**final** **String**[] split = docId.split(":");

**final** **String** type = split[0];

**Uri** contentUri = **null**;

**if** ("image".equals(type)) {

contentUri = **MediaStore**.**Images**.**Media**.**EXTERNAL\_CONTENT\_URI**;

} **else** **if** ("video".equals(type)) {

contentUri = **MediaStore**.**Video**.**Media**.**EXTERNAL\_CONTENT\_URI**;

} **else** **if** ("audio".equals(type)) {

contentUri = **MediaStore**.**Audio**.**Media**.**EXTERNAL\_CONTENT\_URI**;

}

**final** **String** selection = "\_id=?";

**final** **String**[] selectionArgs = **new** **String**[] {

split[1]

};

**return** *getDataColumn*(context, contentUri, selection, selectionArgs);

}

}

// MediaStore (and general)

**else** **if** ("content".equalsIgnoreCase(uri.getScheme())) {

// Return the remote address

**if** (*isGooglePhotosUri*(uri))

**return** uri.getLastPathSegment();

**return** *getDataColumn*(context, uri, **null**, **null**);

}

// File

**else** **if** ("file".equalsIgnoreCase(uri.getScheme())) {

**return** uri.getPath();

}

**return** **null**;

}

/\*\*

\* Get the value of the data column for this Uri. This is useful for

\* MediaStore Uris, and other file-based ContentProviders.

\*

\* **@param** context The context.

\* **@param** uri The Uri to query.

\* **@param** selection (Optional) Filter used in the query.

\* **@param** selectionArgs (Optional) Selection arguments used in the query.

\* **@return** The value of the \_data column, which is typically a file path.

\*/

**public** **static** **String** **getDataColumn**(**Context** context, **Uri** uri, **String** selection,

**String**[] selectionArgs) {

Cursor cursor = **null**;

**final** **String** column = "\_data";

**final** **String**[] projection = {

column

};

**try** {

cursor = context.getContentResolver().query(uri, projection, selection, selectionArgs,

**null**);

**if** (cursor != **null** && cursor.moveToFirst()) {

**final** **int** index = cursor.getColumnIndexOrThrow(column);

**return** cursor.getString(index);

}

} **finally** {

**if** (cursor != **null**)

cursor.close();

}

**return** **null**;

}

/\*\*

\* **@param** uri The Uri to check.

\* **@return** Whether the Uri authority is ExternalStorageProvider.

\*/

**public** **static** **boolean** **isExternalStorageDocument**(**Uri** uri) {

**return** "com.android.externalstorage.documents".equals(uri.getAuthority());

}

/\*\*

\* **@param** uri The Uri to check.

\* **@return** Whether the Uri authority is DownloadsProvider.

\*/

**public** **static** **boolean** **isDownloadsDocument**(**Uri** uri) {

**return** "com.android.providers.downloads.documents".equals(uri.getAuthority());

}

/\*\*

\* **@param** uri The Uri to check.

\* **@return** Whether the Uri authority is MediaProvider.

\*/

**public** **static** **boolean** **isMediaDocument**(**Uri** uri) {

**return** "com.android.providers.media.documents".equals(uri.getAuthority());

}

/\*\*

\* **@param** uri The Uri to check.

\* **@return** Whether the Uri authority is Google Photos.

\*/

**public** **static** **boolean** **isGooglePhotosUri**(**Uri** uri) {

**return** "com.google.android.apps.photos.content".equals(uri.getAuthority());

}

}

**String** path = **GetPathUntil**.*getPath*(**MainActivity**.**this**, data.getData());

//storage/emulated/0/Tencent/QQ\_Images/1024072563582E5B7D1A079ABA87AD4F2F80868328.jpg

## 选择后直接裁剪

//初始化自定义照片文件的uri

**private** **void** **CreatNewOutFile**() {

mFilePath = **Environment**.*getExternalStorageDirectory*().getPath();

mFilePath += "/temp.png";

**File** outputImage = **new** File(mFilePath);

**if** (outputImage.exists()) {

outputImage.delete();

}

**try** {

outputImage.createNewFile();

phptouUri = **Uri**.*fromFile*(outputImage);

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

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

intent.setDataAndType(data.getData(), "image/\*");//从自己文件里取照片，开始裁剪

///！！！！！！！！！！！------------固定比例 "scale", **true**

intent.putExtra("scale", **true**);

// aspectX aspectY 是宽高的比例

intent.putExtra("aspectX", 2);

intent.putExtra("aspectY", 1);

// outputX,outputY 是剪裁图片的宽高

intent.putExtra("outputX", 320 \* 2);

intent.putExtra("outputY", 320);

///！！！！！！！！！！！------------不固定比例 "scale", **false**

intent.putExtra("scale", **false**);

intent.putExtra(**MediaStore**.**EXTRA\_OUTPUT**, phptouUri);//裁剪后放入文件里

startActivityForResult(intent, **SHOW\_CROP\_PHOTO**);

**case** **SHOW\_CROP\_PHOTO**://显示裁剪后的照片

**try** {

bitmap = BitmapFactory.decodeStream(getContentResolver().openInputStream(phptouUri));

或者bitmap = **BitmapFactory**.*decodeFile*(mFilePath);//内存卡上的文件可以读出来

或者bitmap = MediaStore.Images.Media.getBitmap(getContentResolver(), phptouUri);

**if** (bitmap != **null**) {

imageView.setImageBitmap(bitmap);

}

} **catch** (**Exception** e) {

e.printStackTrace();

}

**break**;

## 查看所有相册，并计算出相册的图片张数，最多者在最靠前

/\*\*

\* 利用ContentProvider扫描手机中的所有图片,最终获得jpg最多的那个文件夹

\* \*/

**private** **void** **initDatas**() {

**if** (!**Environment**.*getExternalStorageState*().equals(

**Environment**.**MEDIA\_MOUNTED**)) {

**Toast**.*makeText*(**this**, "当前储存卡不可用！", **Toast**.**LENGTH\_SHORT**).show();

**return**;

}

mpProgressDialog = **ProgressDialog**.*show*(**this**, **null**, "正在加载…");

**new** Thread() {

@Override

**public** **void** **run**() {

**Uri** mImageUri = **MediaStore**.**Images**.**Media**.**EXTERNAL\_CONTENT\_URI**;

**ContentResolver** mContentResolver = **MainActivity**.**this**.getContentResolver();

// 只查询jpeg和png的图片

Cursor mCursor = mContentResolver.query(mImageUri, **null**,

**MediaStore**.**Images**.**Media**.**MIME\_TYPE** + "=? or "

+ **MediaStore**.**Images**.**Media**.**MIME\_TYPE** + "=?",

**new** **String**[] { "image/jpeg", "image/png" },

**MediaStore**.**Images**.**Media**.**DATE\_MODIFIED**);

**HashSet**<**String**> mDirPaths = **new** HashSet<**String**>();// 存储是否包含已经扫描的文件夹

**while** (mCursor.moveToNext()) {

**String** path = mCursor.getString(mCursor.getColumnIndex(**MediaStore**.**Images**.**Media**.**DATA**));

**File** parentFile = **new** File(path).getParentFile();// 获取图片的父路径

**if** (parentFile == **null**) // 有时候有图片但是无法查出路径

**continue**;

**String** dirPath = parentFile.getAbsolutePath();

**FolderBean** folderBean = **null**;

**if** (mDirPaths.contains(dirPath)) {// 如果当前文件夹已经扫描过

**continue**;//结束单次循环

} **else** {

mDirPaths.add(dirPath);// 加入set

folderBean = **new** FolderBean();

folderBean.setDir(dirPath);

folderBean.setFirstImgPath(path);// 扫描的图片的父文件夹没加入set，那么这张path就是第一张

}

**if** (parentFile.list() == **null**){ // 有时候图片也有这种情况，没有长度

**continue**;

}

**int** picSize = parentFile.list(**new** FilenameFilter() {

@Override

**public** **boolean** **accept**(**File** dir, **String** filename) {

**if** (filename.endsWith(".jpg")

|| filename.endsWith(".png")

|| filename.endsWith(".jpeg"))

**return** **true**;

**return** **false**;

}

}).length;

folderBean.setCount(picSize);// folderBean设置数量

mFolderBeans.add(folderBean);

**if** (picSize > mMaxCount) {

mMaxCount = picSize;

mCurrentDir = parentFile;

}

}

mCursor.close();// 关闭查询；

// 通知handler扫描完成

mHandler.sendEmptyMessage(0x111);

// mDirPaths = null;//扫描完成，释放临时存是否是第一次扫描的文件夹

// 由于mDirPaths是在线程内部初始化的，每次自动新建，不用释放

}

}.start();

}

# InputStream、File、String、bitmap、byte[]之间的转换

## InputStream To byte[]

**private** **byte**[] **streamTobyteArray**(**InputStream** is){

**ByteArrayOutputStream** baos = **null**;

**byte**[] byteArry = **null**;

**try** {

baos = **new** ByteArrayOutputStream();

**byte**[] buffer = **new** **byte**[1024];

**int** len = 0;

**while** ((len = is.read(buffer)) != -1) {

baos.write(buffer, 0 ,len);

}

byteArry = baos.toByteArray();

} **catch** (**Exception** e) {

e.printStackTrace();

} **finally** {

**...**

}

**return** byteArry;

}

## InputStream To String（stream--byte[]--String）

**private** **String** **streamToString**(**InputStream** is){

**String** result = **null**;

**try** {

**byte**[] byteArry = streamTobyteArray(is);//输入流转字节数组

result = **new** String(byteArry ,**HTTP**.**UTF\_8**);

} **catch** (**UnsupportedEncodingException** e) {

e.printStackTrace();

}

**return** result;

}

## InputStream To File

**private** **void** **streamToFile**(**String** filePath, **String** fileName, **InputStream** is){

**FileOutputStream** outputStream = **null**;

**File** file = **null**;

**try** {

**File** dir = **new** File(filePath);

**if** (!dir.exists() && dir.isDirectory()) {

dir.mkdirs();

}

file = **new** File(filePath + **File**.**separator** + fileName);

outputStream = **new** FileOutputStream(file);

**byte**[] buffer = **new** **byte**[1024];

**int** len = 0;

**while** ((is.read(buffer)) != 0) {

outputStream.write(buffer, 0 ,len);

}

} **catch** (**Exception** e) {

e.printStackTrace();

} **finally** {

**...**

}

}

## byte[] to File

**private** **void** **byteArrayToFile**(**String** filePath, **String** fileName, **byte**[] byteArray){

**BufferedOutputStream** outputStream = **null**;

**File** file = **null**;

**try** {

**File** dir = **new** File(filePath);

**if** (!dir.exists() && dir.isDirectory()) {

dir.mkdirs();

}

file = **new** File(filePath + **File**.**separator** + fileName);

outputStream = **new** BufferedOutputStream(**new** FileOutputStream(file));

outputStream.write(byteArray);

} **catch** (**Exception** e) {

e.printStackTrace();

} **finally** {

**...**

}

}

## File To String

**private** **String** **fileToString**(**String** filePath){

**String** result = **null**;

**BufferedReader** reader = **null**;

**try** {

**StringBuilder** builder = **new** StringBuilder();

reader = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(filePath), "UTF-8"));

**String** line = **null**;

**while** ((line = reader.readLine()) != **null**) {

builder.append(line);

builder.append("\r\n");

}

result = builder.toString();

} **catch** (**Exception** e) {

e.printStackTrace();

} **finally** {

**...**

}

**return** result;

}

## File To byte[](file--stream--byte[])

**private** **byte**[] **fileToByteArray**(**String** filePath){

**FileInputStream** in = **null**;

**File** file = **null**;

**byte**[] byteArry = **null**;

**try** {

file = **new** File(filePath);

in = **new** FileInputStream(file);

byteArry = streamTobyteArray(in);//输入流转字节数组

} **catch** (**Exception** e) {

e.printStackTrace();

} **finally** {

**...**

}

**return** byteArry;

}

## Bitmap转byte[]

**ByteArrayOutputStream** bStream = **new** ByteArrayOutputStream();

bitmap.compress(**Bitmap**.CompressFormat.**PNG**, 100, bStream);

**byte**[] temp = bStream.toByteArray();

## Bitmap转File

//实例化字节数组输出流

**ByteArrayOutputStream** bStream = **new** ByteArrayOutputStream();

**FileOutputStream** fos = null;

//将Bitmap写入bStream

bitmap.compress(**Bitmap**.CompressFormat.**PNG**, 100, bStream);

**byte**[] temp = bStream.toByteArray();

**try** {

fos = **new** FileOutputStream(mFilePath);

fos.write(temp);

} **catch** (**FileNotFoundException** e) {

e.printStackTrace();

}**catch** (**IOException** e) {

e.printStackTrace();

}**finally** {

**try** {

fos.close();

} **catch** (**IOException** e) {

e.printStackTrace();

}

}

## File转Btimap

**Bitmap** bitmap = **BitmapFactory**.*decodeFile*(mFilePath);

# 颜色推荐



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

<resources>

<color name=*"white"*>#ffffff</color><!--白色 -->

<color name=*"ivory"*>#fffff0</color><!--象牙色 -->

<color name=*"lightyellow"*>#ffffe0</color><!--亮黄色 -->

<color name=*"yellow"*>#ffff00</color><!--黄色 -->

<color name=*"snow"*>#fffafa</color><!--雪白色 -->

<color name=*"floralwhite"*>#fffaf0</color><!--花白色 -->

<color name=*"lemonchiffon"*>#fffacd</color><!--柠檬绸色 -->

<color name=*"cornsilk"*>#fff8dc</color><!--米绸色 -->

<color name=*"seaShell"*>#fff5ee</color><!--海贝色 -->

<color name=*"lavenderblush"*>#fff0f5</color><!--淡紫红 -->

<color name=*"papayawhip"*>#ffefd5</color><!--番木色 -->

<color name=*"blanchedalmond"*>#ffebcd</color><!--白杏色 -->

<color name=*"mistyrose"*>#ffe4e1</color><!--浅玫瑰色 -->

<color name=*"bisque"*>#ffe4c4</color><!--桔黄色 -->

<color name=*"moccasin"*>#ffe4b5</color><!--鹿皮色 -->

<color name=*"navajowhite"*>#ffdead</color><!--纳瓦白 -->

<color name=*"peachpuff"*>#ffdab9</color><!--桃色 -->

<color name=*"gold"*>#ffd700</color><!--金色 -->

<color name=*"pink"*>#ffc0cb</color><!--粉红色 -->

<color name=*"lightpink"*>#ffb6c1</color><!--亮粉红色 -->

<color name=*"orange"*>#ffa500</color><!--橙色 -->

<color name=*"lightsalmon"*>#ffa07a</color><!--亮肉色 -->

<color name=*"darkorange"*>#ff8c00</color><!--暗桔黄色 -->

<color name=*"coral"*>#ff7f50</color><!--珊瑚色 -->

<color name=*"hotpink"*>#ff69b4</color><!--热粉红色 -->

<color name=*"tomato"*>#ff6347</color><!--西红柿色 -->

<color name=*"orangered"*>#ff4500</color><!--红橙色 -->

<color name=*"deeppink"*>#ff1493</color><!--深粉红色 -->

<color name=*"fuchsia"*>#ff00ff</color><!--紫红色 -->

<color name=*"magenta"*>#ff00ff</color><!--红紫色 -->

<color name=*"red"*>#ff0000</color><!--红色 -->

<color name=*"oldlace"*>#fdf5e6</color><!--老花色 -->

<color name=*"lightgoldenrodyellow"*>#fafad2</color><!--亮金黄色 -->

<color name=*"linen"*>#faf0e6</color><!--亚麻色 -->

<color name=*"antiquewhite"*>#faebd7</color><!--古董白 -->

<color name=*"salmon"*>#fa8072</color><!--鲜肉色 -->

<color name=*"ghostwhite"*>#f8f8ff</color><!--幽灵白 -->

<color name=*"mintcream"*>#f5fffa</color><!--薄荷色 -->

<color name=*"whitesmoke"*>#f5f5f5</color><!--烟白色 -->

<color name=*"beige"*>#f5f5dc</color><!--米色 -->

<color name=*"wheat"*>#f5deb3</color><!--浅黄色 -->

<color name=*"sandybrown"*>#f4a460</color><!--沙褐色 -->

<color name=*"azure"*>#f0ffff</color><!--天蓝色 -->

<color name=*"honeydew"*>#f0fff0</color><!--蜜色 -->

<color name=*"aliceblue"*>#f0f8ff</color><!--艾利斯兰 -->

<color name=*"khaki"*>#f0e68c</color><!--黄褐色 -->

<color name=*"lightcoral"*>#f08080</color><!--亮珊瑚色 -->

<color name=*"palegoldenrod"*>#eee8aa</color><!--苍麒麟色 -->

<color name=*"violet"*>#ee82ee</color><!--紫罗兰色 -->

<color name=*"darksalmon"*>#e9967a</color><!--暗肉色 -->

<color name=*"lavender"*>#e6e6fa</color><!--淡紫色 -->

<color name=*"lightcyan"*>#e0ffff</color><!--亮青色 -->

<color name=*"burlywood"*>#deb887</color><!--实木色 -->

<color name=*"plum"*>#dda0dd</color><!--洋李色 -->

<color name=*"gainsboro"*>#dcdcdc</color><!--淡灰色 -->

<color name=*"crimson"*>#dc143c</color><!--暗深红色 -->

<color name=*"palevioletred"*>#db7093</color><!--苍紫罗兰色 -->

<color name=*"goldenrod"*>#daa520</color><!--金麒麟色 -->

<color name=*"orchid"*>#da70d6</color><!--淡紫色 -->

<color name=*"thistle"*>#d8bfd8</color><!--蓟色 -->

<color name=*"lightgray"*>#d3d3d3</color><!--亮灰色 -->

<color name=*"lightgrey"*>#d3d3d3</color><!--亮灰色 -->

<color name=*"tan"*>#d2b48c</color><!--茶色 -->

<color name=*"chocolate"*>#d2691e</color><!--巧可力色 -->

<color name=*"peru"*>#cd853f</color><!--秘鲁色 -->

<color name=*"indianred"*>#cd5c5c</color><!--印第安红 -->

<color name=*"mediumvioletred"*>#c71585</color><!--中紫罗兰色 -->

<color name=*"silver"*>#c0c0c0</color><!--银色 -->

<color name=*"darkkhaki"*>#bdb76b</color><!--暗黄褐色 -->

<color name=*"rosybrown"*>#bc8f8f</color><!--褐玫瑰红 -->

<color name=*"mediumorchid"*>#ba55d3</color><!--中粉紫色 -->

<color name=*"darkgoldenrod"*>#b8860b</color><!--暗金黄色 -->

<color name=*"firebrick"*>#b22222</color><!--火砖色 -->

<color name=*"powderblue"*>#b0e0e6</color><!--粉蓝色 -->

<color name=*"lightsteelblue"*>#b0c4de</color><!--亮钢兰色 -->

<color name=*"paleturquoise"*>#afeeee</color><!--苍宝石绿 -->

<color name=*"greenyellow"*>#adff2f</color><!--黄绿色 -->

<color name=*"lightblue"*>#add8e6</color><!--亮蓝色 -->

<color name=*"darkgray"*>#a9a9a9</color><!--暗灰色 -->

<color name=*"darkgrey"*>#a9a9a9</color><!--暗灰色 -->

<color name=*"brown"*>#a52a2a</color><!--褐色 -->

<color name=*"sienna"*>#a0522d</color><!--赭色 -->

<color name=*"darkorchid"*>#9932cc</color><!--暗紫色 -->

<color name=*"palegreen"*>#98fb98</color><!--苍绿色 -->

<color name=*"darkviolet"*>#9400d3</color><!--暗紫罗兰色 -->

<color name=*"mediumpurple"*>#9370db</color><!--中紫色 -->

<color name=*"lightgreen"*>#90ee90</color><!--亮绿色 -->

<color name=*"darkseagreen"*>#8fbc8f</color><!--暗海兰色 -->

<color name=*"saddlebrown"*>#8b4513</color><!--重褐色 -->

<color name=*"darkmagenta"*>#8b008b</color><!--暗洋红 -->

<color name=*"darkred"*>#8b0000</color><!--暗红色 -->

<color name=*"blueviolet"*>#8a2be2</color><!--紫罗兰蓝色 -->

<color name=*"lightskyblue"*>#87cefa</color><!--亮天蓝色 -->

<color name=*"skyblue"*>#87ceeb</color><!--天蓝色 -->

<color name=*"gray"*>#808080</color><!--灰色 -->

<color name=*"grey"*>#808080</color><!--灰色 -->

<color name=*"olive"*>#808000</color><!--橄榄色 -->

<color name=*"purple"*>#800080</color><!--紫色 -->

<color name=*"maroon"*>#800000</color><!--粟色 -->

<color name=*"aquamarine"*>#7fffd4</color><!--碧绿色 -->

<color name=*"chartreuse"*>#7fff00</color><!--黄绿色 -->

<color name=*"lawngreen"*>#7cfc00</color><!--草绿色 -->

<color name=*"mediumslateblue"*>#7b68ee</color><!--中暗蓝色 -->

<color name=*"lightslategray"*>#778899</color><!--亮蓝灰 -->

<color name=*"lightslategrey"*>#778899</color><!--亮蓝灰 -->

<color name=*"slategray"*>#708090</color><!--灰石色 -->

<color name=*"slategrey"*>#708090</color><!--灰石色 -->

<color name=*"olivedrab"*>#6b8e23</color><!--深绿褐色 -->

<color name=*"slateblue"*>#6a5acd</color><!--石蓝色 -->

<color name=*"dimgray"*>#696969</color><!--暗灰色 -->

<color name=*"dimgrey"*>#696969</color><!--暗灰色 -->

<color name=*"mediumaquamarine"*>#66cdaa</color><!--中绿色 -->

<color name=*"cornflowerblue"*>#6495ed</color><!--菊兰色 -->

<color name=*"cadetblue"*>#5f9ea0</color><!--军兰色 -->

<color name=*"darkolivegreen"*>#556b2f</color><!--暗橄榄绿 -->

<color name=*"indigo"*>#4b0082</color><!--靛青色 -->

<color name=*"mediumturquoise"*>#48d1cc</color><!--中绿宝石 -->

<color name=*"darkslateblue"*>#483d8b</color><!--暗灰蓝色 -->

<color name=*"steelblue"*>#4682b4</color><!--钢兰色 -->

<color name=*"royalblue"*>#4169e1</color><!--皇家蓝 -->

<color name=*"turquoise"*>#40e0d0</color><!--青绿色 -->

<color name=*"mediumseagreen"*>#3cb371</color><!--中海蓝 -->

<color name=*"limegreen"*>#32cd32</color><!--橙绿色 -->

<color name=*"darkslategray"*>#2f4f4f</color><!--暗瓦灰色 -->

<color name=*"darkslategrey"*>#2f4f4f</color><!--暗瓦灰色 -->

<color name=*"seagreen"*>#2e8b57</color><!--海绿色 -->

<color name=*"forestgreen"*>#228b22</color><!--森林绿 -->

<color name=*"lightseagreen"*>#20b2aa</color><!--亮海蓝色 -->

<color name=*"dodgerblue"*>#1e90ff</color><!--闪兰色 -->

<color name=*"midnightblue"*>#191970</color><!--中灰兰色 -->

<color name=*"aqua"*>#00ffff</color><!--浅绿色 -->

<color name=*"cyan"*>#00ffff</color><!--青色 -->

<color name=*"springgreen"*>#00ff7f</color><!--春绿色 -->

<color name=*"lime"*>#00ff00</color><!--酸橙色 -->

<color name=*"mediumspringgreen"*>#00fa9a</color><!--中春绿色 -->

<color name=*"darkturquoise"*>#00ced1</color><!--暗宝石绿 -->

<color name=*"deepskyblue"*>#00bfff</color><!--深天蓝色 -->

<color name=*"darkcyan"*>#008b8b</color><!--暗青色 -->

<color name=*"teal"*>#008080</color><!--水鸭色 -->

<color name=*"green"*>#008000</color><!--绿色 -->

<color name=*"darkgreen"*>#006400</color><!--暗绿色 -->

<color name=*"blue"*>#0000ff</color><!--蓝色 -->

<color name=*"mediumblue"*>#0000cd</color><!--中兰色 -->

<color name=*"darkblue"*>#00008b</color><!--暗蓝色 -->

<color name=*"navy"*>#000080</color><!--海军色 -->

<color name=*"black"*>#000000</color><!--黑色 -->

</resources>

# 所有的权限uses-permission

访问登记属性 android.permission.ACCESS\_CHECKIN\_PROPERTIES ，读取或写入登记check-in数据库属性表的权限

获取错略位置 android.permission.ACCESS\_COARSE\_LOCATION，通过WiFi或移动基站的方式获取用户错略的经纬度信息，定位精度大概误差在30~1500米

获取精确位置 android.permission.ACCESS\_FINE\_LOCATION，通过GPS芯片接收卫星的定位信息，定位精度达10米以内

访问定位额外命令 android.permission.ACCESS\_LOCATION\_EXTRA\_COMMANDS，允许程序访问额外的定位提供者指令

获取模拟定位信息 android.permission.ACCESS\_MOCK\_LOCATION，获取模拟定位信息，一般用于帮助开发者调试应用

获取网络状态 android.permission.ACCESS\_NETWORK\_STATE，获取网络信息状态，如当前的网络连接是否有效

访问Surface Flinger android.permission.ACCESS\_SURFACE\_FLINGER，Android平台上底层的图形显示支持，一般用于游戏或照相机预览界面和底层模式的屏幕截图

获取WiFi状态 android.permission.ACCESS\_WIFI\_STATE，获取当前WiFi接入的状态以及WLAN热点的信息

账户管理 android.permission.ACCOUNT\_MANAGER，获取账户验证信息，主要为GMail账户信息，只有系统级进程才能访问的权限

验证账户 android.permission.AUTHENTICATE\_ACCOUNTS，允许一个程序通过账户验证方式访问账户管理ACCOUNT\_MANAGER相关信息

电量统计 android.permission.BATTERY\_STATS，获取电池电量统计信息

绑定小插件 android.permission.BIND\_APPWIDGET，允许一个程序告诉appWidget服务需要访问小插件的数据库，只有非常少的应用才用到此权限

绑定设备管理 android.permission.BIND\_DEVICE\_ADMIN，请求系统管理员接收者receiver，只有系统才能使用

绑定输入法 android.permission.BIND\_INPUT\_METHOD ，请求InputMethodService服务，只有系统才能使用

绑定RemoteView android.permission.BIND\_REMOTEVIEWS，必须通过RemoteViewsService服务来请求，只有系统才能用

绑定壁纸 android.permission.BIND\_WALLPAPER，必须通过WallpaperService服务来请求，只有系统才能用

使用蓝牙 android.permission.BLUETOOTH，允许程序连接配对过的蓝牙设备

蓝牙管理 android.permission.BLUETOOTH\_ADMIN，允许程序进行发现和配对新的蓝牙设备

变成砖头 android.permission.BRICK，能够禁用手机，非常危险，顾名思义就是让手机变成砖头

应用删除时广播 android.permission.BROADCAST\_PACKAGE\_REMOVED，当一个应用在删除时触发一个广播

收到短信时广播 android.permission.BROADCAST\_SMS，当收到短信时触发一个广播

连续广播 android.permission.BROADCAST\_STICKY，允许一个程序收到广播后快速收到下一个广播

WAP PUSH广播 android.permission.BROADCAST\_WAP\_PUSH，WAP PUSH服务收到后触发一个广播

拨打电话 android.permission.CALL\_PHONE，允许程序从非系统拨号器里输入电话号码

通话权限 android.permission.CALL\_PRIVILEGED，允许程序拨打电话，替换系统的拨号器界面

拍照权限 android.permission.CAMERA，允许访问摄像头进行拍照

改变组件状态 android.permission.CHANGE\_COMPONENT\_ENABLED\_STATE，改变组件是否启用状态

改变配置 android.permission.CHANGE\_CONFIGURATION，允许当前应用改变配置，如定位

改变网络状态 android.permission.CHANGE\_NETWORK\_STATE，改变网络状态如是否能联网

改变WiFi多播状态 android.permission.CHANGE\_WIFI\_MULTICAST\_STATE，改变WiFi多播状态

改变WiFi状态 android.permission.CHANGE\_WIFI\_STATE，改变WiFi状态

清除应用缓存 android.permission.CLEAR\_APP\_CACHE，清除应用缓存

清除用户数据 android.permission.CLEAR\_APP\_USER\_DATA，清除应用的用户数据

底层访问权限 android.permission.CWJ\_GROUP，允许CWJ账户组访问底层信息

手机优化大师扩展权限 android.permission.CELL\_PHONE\_MASTER\_EX，手机优化大师扩展权限

控制定位更新 android.permission.CONTROL\_LOCATION\_UPDATES，允许获得移动网络定位信息改变

删除缓存文件 android.permission.DELETE\_CACHE\_FILES，允许应用删除缓存文件

删除应用 android.permission.DELETE\_PACKAGES，允许程序删除应用

电源管理 android.permission.DEVICE\_POWER，允许访问底层电源管理

应用诊断 android.permission.DIAGNOSTIC，允许程序到RW到诊断资源

禁用键盘锁 android.permission.DISABLE\_KEYGUARD，允许程序禁用键盘锁

转存系统信息 android.permission.DUMP，允许程序获取系统dump信息从系统服务

状态栏控制 android.permission.EXPAND\_STATUS\_BAR，允许程序扩展或收缩状态栏

工厂测试模式 android.permission.FACTORY\_TEST，允许程序运行工厂测试模式

使用闪光灯 android.permission.FLASHLIGHT，允许访问闪光灯

强制后退 android.permission.FORCE\_BACK，允许程序强制使用back后退按键，无论Activity是否在顶层

访问账户Gmail列表 android.permission.GET\_ACCOUNTS，访问GMail账户列表

获取应用大小 android.permission.GET\_PACKAGE\_SIZE，获取应用的文件大小

获取任务信息 android.permission.GET\_TASKS，允许程序获取当前或最近运行的应用

允许全局搜索 android.permission.GLOBAL\_SEARCH，允许程序使用全局搜索功能

硬件测试 android.permission.HARDWARE\_TEST，访问硬件辅助设备，用于硬件测试

注射事件 android.permission.INJECT\_EVENTS，允许访问本程序的底层事件，获取按键、轨迹球的事件流

安装定位提供 android.permission.INSTALL\_LOCATION\_PROVIDER，安装定位提供

安装应用程序 android.permission.INSTALL\_PACKAGES，允许程序安装应用

内部系统窗口 android.permission.INTERNAL\_SYSTEM\_WINDOW，允许程序打开内部窗口，不对第三方应用程序开放此权限

访问网络 android.permission.INTERNET，访问网络连接，可能产生GPRS流量

结束后台进程 android.permission.KILL\_BACKGROUND\_PROCESSES，允许程序调用killBackgroundProcesses(String).方法结束后台进程

管理账户 android.permission.MANAGE\_ACCOUNTS，允许程序管理AccountManager中的账户列表

管理程序引用 android.permission.MANAGE\_APP\_TOKENS，管理创建、摧毁、Z轴顺序，仅用于系统

高级权限 android.permission.MTWEAK\_USER，允许mTweak用户访问高级系统权限

社区权限 android.permission.MTWEAK\_FORUM，允许使用mTweak社区权限

软格式化 android.permission.MASTER\_CLEAR，允许程序执行软格式化，删除系统配置信息

修改声音设置 android.permission.MODIFY\_AUDIO\_SETTINGS，修改声音设置信息

修改电话状态 android.permission.MODIFY\_PHONE\_STATE，修改电话状态，如飞行模式，但不包含替换系统拨号器界面

格式化文件系统 android.permission.MOUNT\_FORMAT\_FILESYSTEMS，格式化可移动文件系统，比如格式化清空SD卡

挂载文件系统 android.permission.MOUNT\_UNMOUNT\_FILESYSTEMS，挂载、反挂载外部文件系统

允许NFC通讯 android.permission.NFC，允许程序执行NFC近距离通讯操作，用于移动支持

永久Activity android.permission.PERSISTENT\_ACTIVITY，创建一个永久的Activity，该功能标记为将来将被移除

处理拨出电话 android.permission.PROCESS\_OUTGOING\_CALLS，允许程序监视，修改或放弃播出电话

读取日程提醒 android.permission.READ\_CALENDAR，允许程序读取用户的日程信息

读取联系人 android.permission.READ\_CONTACTS，允许应用访问联系人通讯录信息

屏幕截图 android.permission.READ\_FRAME\_BUFFER，读取帧缓存用于屏幕截图

读取收藏夹和历史记录 com.android.browser.permission.READ\_HISTORY\_BOOKMARKS，读取浏览器收藏夹和历史记录

读取输入状态 android.permission.READ\_INPUT\_STATE，读取当前键的输入状态，仅用于系统

读取系统日志 android.permission.READ\_LOGS，读取系统底层日志

读取电话状态 android.permission.READ\_PHONE\_STATE，访问电话状态

读取短信内容 android.permission.READ\_SMS，读取短信内容

读取同步设置 android.permission.READ\_SYNC\_SETTINGS，读取同步设置，读取Google在线同步设置

读取同步状态 android.permission.READ\_SYNC\_STATS，读取同步状态，获得Google在线同步状态

重启设备 android.permission.REBOOT，允许程序重新启动设备

开机自动允许 android.permission.RECEIVE\_BOOT\_COMPLETED，允许程序开机自动运行

接收彩信 android.permission.RECEIVE\_MMS，接收彩信

接收短信 android.permission.RECEIVE\_SMS，接收短信

接收Wap Push android.permission.RECEIVE\_WAP\_PUSH，接收WAP PUSH信息

录音 android.permission.RECORD\_AUDIO，录制声音通过手机或耳机的麦克

排序系统任务 android.permission.REORDER\_TASKS，重新排序系统Z轴运行中的任务

结束系统任务 android.permission.RESTART\_PACKAGES，结束任务通过restartPackage(String)方法，该方式将在外来放弃

发送短信 android.permission.SEND\_SMS，发送短信

设置Activity观察其 android.permission.SET\_ACTIVITY\_WATCHER，设置Activity观察器一般用于monkey测试

设置闹铃提醒 com.android.alarm.permission.SET\_ALARM，设置闹铃提醒

设置总是退出 android.permission.SET\_ALWAYS\_FINISH，设置程序在后台是否总是退出

设置动画缩放 android.permission.SET\_ANIMATION\_SCALE，设置全局动画缩放

设置调试程序 android.permission.SET\_DEBUG\_APP，设置调试程序，一般用于开发

设置屏幕方向 android.permission.SET\_ORIENTATION，设置屏幕方向为横屏或标准方式显示，不用于普通应用

设置应用参数 android.permission.SET\_PREFERRED\_APPLICATIONS，设置应用的参数，已不再工作具体查看addPackageToPreferred(String) 介绍

设置进程限制 android.permission.SET\_PROCESS\_LIMIT，允许程序设置最大的进程数量的限制

设置系统时间 android.permission.SET\_TIME，设置系统时间

设置系统时区 android.permission.SET\_TIME\_ZONE，设置系统时区

设置桌面壁纸 android.permission.SET\_WALLPAPER，设置桌面壁纸

设置壁纸建议 android.permission.SET\_WALLPAPER\_HINTS，设置壁纸建议

发送永久进程信号 android.permission.SIGNAL\_PERSISTENT\_PROCESSES，发送一个永久的进程信号

状态栏控制 android.permission.STATUS\_BAR，允许程序打开、关闭、禁用状态栏

访问订阅内容 android.permission.SUBSCRIBED\_FEEDS\_READ，访问订阅信息的数据库

写入订阅内容 android.permission.SUBSCRIBED\_FEEDS\_WRITE，写入或修改订阅内容的数据库

显示系统窗口 android.permission.SYSTEM\_ALERT\_WINDOW，显示系统窗口

更新设备状态 android.permission.UPDATE\_DEVICE\_STATS，更新设备状态

使用证书 android.permission.USE\_CREDENTIALS，允许程序请求验证从AccountManager

使用SIP视频 android.permission.USE\_SIP，允许程序使用SIP视频服务

使用振动 android.permission.VIBRATE，允许振动

唤醒锁定 android.permission.WAKE\_LOCK，允许程序在手机屏幕关闭后后台进程仍然运行

写入GPRS接入点设置 android.permission.WRITE\_APN\_SETTINGS，写入网络GPRS接入点设置

写入日程提醒 android.permission.WRITE\_CALENDAR，写入日程，但不可读取

写入联系人 android.permission.WRITE\_CONTACTS，写入联系人，但不可读取

写入外部存储 android.permission.WRITE\_EXTERNAL\_STORAGE，允许程序写入外部存储，如SD卡上写文件

写入Google地图数据 android.permission.WRITE\_GSERVICES，允许程序写入Google Map服务数据

写入收藏夹和历史记录 com.android.browser.permission.WRITE\_HISTORY\_BOOKMARKS，写入浏览器历史记录或收藏夹，但不可读取

读写系统敏感设置 android.permission.WRITE\_SECURE\_SETTINGS，允许程序读写系统安全敏感的设置项

读写系统设置 android.permission.WRITE\_SETTINGS，允许读写系统设置项

编写短信 android.permission.WRITE\_SMS，允许编写短信

写入在线同步设置 android.permission.WRITE\_SYNC\_SETTINGS，写入Google在线同步设置

# 权限机制

## Android权限机制

Android6.0（api 23）以前，Android的权限机制比较简单，开发者在AndroidManifest文件中声明需要的权限，APP安装时，系统提示用户APP将获取的权限，需要用户同意授权才能继续安装，从此APP便永久的获得了授权。然而，同期的iOS对于权限的处理会更加灵活，权限的授予并不是在安装时，而是在APP运行时，用户可以根据自身的需要，决定是否授予APP某一权限，同时，用户也可以很方便回收授予的权限。显然，动态权限管理的机制，对于用户的隐私保护是更加适用的，Android过于简单的权限机制也受到了不少人的吐槽。终于，Android6.0也发布了动态权限的机制。

### 开始适配和如何兼容

APP要适配Android6.0非常简单，只需要将targetSdkVersion和compileSdkVersion都升级到23及以上，同时加入权限检查申请等代码逻辑即可。这里很多人会有一些疑惑，如果针对旧版本的APP在Android6.0机型上运行或者针对Android6.0适配了的APP在Android6.0以下机型上运行，会有什么表现呢？是如何兼容的呢？

* 首先，旧版本APP（targetSdkVersion低于23），因为没有适配权限的申请相关逻辑，在Android6.0以上机型运行的时候，仍然采用安装时授权的方案。
* 适配了Android6.0的APP，在低版本Android系统上运行的时候，仍然采用安装时授权的方案，但是开发者需要注意的是，权限申请的代码逻辑只应该在Android6.0及以上的机型被执行。

### 危险权限与普通权限

一开始，听到要加入权限判断和申请代码逻辑的程序员内心可能是崩溃的：正常的一个有一定规模的APP，很容易就七七八八的声明了很多权限，如果每个权限都申请岂不是非常麻烦？

好歹，Google还算比较明智，并不是所有的权限都需要运行时申请才能使用。Google对每个权限的隐私危害性进行了评估。将权限分为了两大类：普通权限和危险权限。举个例子，控制手机震动的权限对于用户并没有什么危害，只要开发者声明了这个权限，安装后就可以一直被授权，也不能被回收，但是，像读取sd卡数据这类权限，很显然就是危险权限了，APP必须向用户申请这个权限。

Google还是很体贴我们开发者的，为了进一步减少开发的工作量和申请权限对用户的骚扰，对危险权限根据各自的属性进行了分组。举个例子，读sd卡和写sd卡，这两个权限通常都是成对声明和使用的，因此，它们被分为一组，而且，只要我们获取了这个权限组里面的任意一个权限，就可以获取整个权限组的权限。Google对于危险权限的定义和分组见下图。

## 权限相关API说明

首先，在动态权限申请的流程中，开发者主要关注流程和API如下：（仍然需在manifest里设置权限请求）

### 检查权限是否授予

public int checkSelfPermission(permission)

### 申请权限

public final void requestPermissions(new String[permission1,permission2,...], requestCode)

这个时候，会弹出系统授权弹窗（授权弹窗是不支持自定义的，原因理所当然）。

### 权限回调

public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults){

//继续执行逻辑或者提示权限获取失败

}

用户在系统弹窗里面选择后，结果会通过Activity的onRequestPermissionsResult方法回调APP。

### 权限说明

用户如果选择了拒绝，下一次在需要声明该权限的时候，Google建议APP开发者给予用户更多的说明，因此提供了下面这个API，这个方法返回值在使用过程中会发现有点纠结。

public boolean shouldShowRequestPermissionRationale(permission){

1、APP没有申请这个权限的话，返回false

2、用户拒绝时，勾选了不再提示的话，返回false

3、用户拒绝，但是没有勾选不再提示的话，返回true

}

因此如果想在第一次就给用户提示，需要记录权限是否申请过，没有申请过的话，强制弹窗提示，而不能根据这个方法的返回值来。

### 获取照相权限举例说明

private static final int REQUEST\_CODE\_PERMISSION = 1;//请求权限 请求码

private static final String REQUEST\_CAMERA\_PERMISSION = Manifest.permission.CAMERA;//请求的照相机权限

/\*\*请求权限\*/

private void initPermission() {

int flag = ActivityCompat.checkSelfPermission(this, REQUEST\_CAMERA\_PERMISSION);//检查有没有照相机权限

if(PackageManager.PERMISSION\_GRANTED != flag){//如果没有权限

ActivityCompat.requestPermissions(this,new String[]{REQUEST\_CAMERA\_PERMISSION}, REQUEST\_CODE\_PERMISSION);//请求权限

}else{//如果有权限

doYourWant();

}

}

/\*\*请求权限的回调\*/

@Override

public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {

super.onRequestPermissionsResult(requestCode, permissions, grantResults);

if (REQUEST\_CODE\_PERMISSION == requestCode){

switch (grantResults[0]){//权限列表中0角标权限的授权结果

case PackageManager.PERMISSION\_DENIED://用户选择不给权限

//判断用户有没有勾选“不再询问”

boolean isSecondRequest = ActivityCompat.shouldShowRequestPermissionRationale(this,REQUEST\_CAMERA\_PERMISSION);

if(isSecondRequest){//没有勾选“不再询问”，那就再次请求，这次会出现“不再询问”

ActivityCompat.requestPermissions(this,new String[]{REQUEST\_CAMERA\_PERMISSION}, REQUEST\_CODE\_PERMISSION);

}else{//一旦用户并勾选不再询问，就提示用户怎么挽回，如下

Toast.makeText(this, "拍照权限被禁用，请在权限管理修改", Toast.LENGTH\_SHORT).show();

}

break;

case PackageManager.PERMISSION\_GRANTED://用户第一次选择给权限

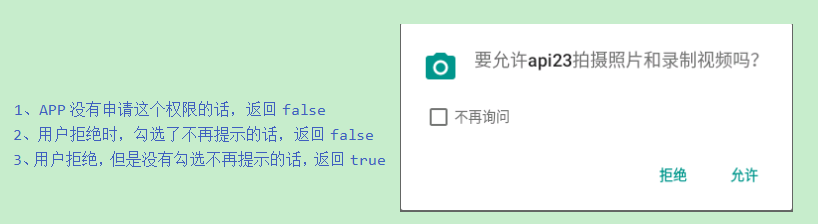
doYourWant();

break;

}

}

}



## QQ音乐的权限适配经验

### 不同权限，申请的时机不同

QQ音乐作为一个比较复杂的流媒体应用，也需要不少权限，但是究竟在什么时候来申请这些权限就成了适配6.0时首当其冲问题。针对这个问题，我们也对需要的权限进行了思考，大致认为申请权限需要分为两个时机。

用户触发：这个很好理解，有些和特性相关的权限，比如说听歌识曲的录音权限、自建歌单封面拍照权限等，这类权限平时APP运行时并不需要，那么我们选择在用户触发或者进入该功能的时候，进行授权受阻逻辑。

应用启动时：我们在梳理的时候发现，有些权限（读取设备信息，读写sd卡等）并不是由用户或者特性触发的，而是网络免流，登录安全，日志系统这些底层逻辑无时不刻触发的。对于这些权限，就比较纠结了。不过回过头来看，这些权限通常是开发者或者APP不能妥协的权限，因为如果用户不授权的话，将会影响整个APP的功能和数据。所以，我们选择比较暴力的方式，在应用启动的时候，就受阻。这也是Google建议的一种方式。

但是需要注意的是，一开始就申请授权也不要冷冰冰地直接拉起系统弹窗授权，建议先用APP自己的弹窗向用户礼貌地说明为什么需要这几个权限，比如，读取不到设备信息无法联通免流，无法保证登录安全，读取不到SD卡无法播放歌曲等，避免太生硬引起用户的反感。特别是，因为本地化翻译的原因，Google对于权限的弹窗说明很不local，例如我们申请读取设备信息的权限时，系统的弹窗是“电话权限”，这里很容易引起用户的误解，所以，合理的引导和解释是必不可少的。

### 应用启动授权，需要一个壳

刚刚已经说到了，很多隐形的权限和特性无关。那么，如果我们直接启动APP，用户又还没有授权的情况下，很多初始化逻辑很容易就因为没有权限crash了，即使没有crash，后面也可能会有或多或少其他的问题。因此，我们需要在这些权限完全授予前，禁止这些逻辑的执行。

做过启动相关的同学都知道，拦截一个APP正常的启动后面再恢复，是很复杂的一件事情，往往我们需要一个外壳来把业务逻辑的内壳隔绝开。就QQ音乐而言，我们很容易的就想到了dex加载的壳，需求也很类似，dex加载也需要优先于业务来做。顺着这个思路，很自然地，我们就选择了在dex的壳里面做权限的受阻逻辑，而且也很快很好的达到了预期的效果。相信现在大部分APP都是分dex的了，因此建议按照这个方式来做，可以节省很多的工作量。

## Android权限机制“乱象”

这里要说的乱象，其实是和Android严重的碎片化有一定的关系。随着国产ROM越来越个性，很多ROM在尝试建立自己的权限机制，有些甚至基于Android5.x就开放了原生的或者开发了自己的权限机制。而面对这些情况，我们往往能做的非常有限，举几个例子。

### 读取运动数据权限

开发QQ音乐跑步电台的过程中发现，在某国产ROM的一些机型上会提示“应用读取运动数据权限”的系统弹窗。可是，反复查阅相关API发现，我们使用的计步相关的Sensor并不需要申请什么权限。可如果用户选择了拒绝，即使APP注册了Sensor，也收不到系统的回调。后来联系该厂商的相关人员后，给出的答复是，第三方APP无法检查和申请这个权限，这个权限本身也属于该厂商ROM自己的权限机制。

类似的案例还有一个，就是在某厂商的手机管家，会一直提示QQ音乐尝试读取应用程序列表。其实，我们并没有读取应用程序列表，只是调用了PackageManager相关的一些API，就是触发这个告警。

对于这类问题，我们怀疑，第三方ROM是在运行时检测到了APP调用了相关的API后，进行权限阻断。这里开发同学需要注意的是，被阻断的API不一定会导致crash，但是可能导致我们获取不到正确的返回值或者收不到系统的一些消息回调。

### 无法添加快捷方式

本来<uses-permission android:name="com.android.launcher.permission.INSTALL\_SHORTCUT"/>声明后，我们就可以在桌面上创建快捷方式了，而且这个权限也不是危险权限。可是某些国产ROM，对于APP添加快捷方式限制的比较严，必须要用户在设置里面手动允许添加快捷方式后，APP才能最终成功的添加。这种情况，APP也不能知道是否能添加快捷方式，只能默默的添加失败了。不过好在这里受影响并不是主快捷方式，而且某些功能的快捷方式入口。

### 消失的桌面歌词，悬浮窗权限

QQ音乐桌面歌词采用了向WindowManager里面添加View的方式实现。可是很多国产ROM很早就具备了悬浮窗权限。一开始，我们将type改为LayoutParams.TYPE\_TOAST同时声明<uses-permission android:name="android.permission.SYSTEM\_ALERT\_WINDOW"/>这个普通权限，躲避了大多数系统的问题。可是，2016年底，随着某ROM系统的升级，这一招也没用了，大批用户反馈爆发。

我们继续尝试检测悬浮窗权限，发现checkPermission（"android.permission.SYSTEM\_ALERT\_WINDOW"）返回的结果永远是true，因此这条路也走不通。

最终，经过各种查阅，发现这个悬浮窗权限并不在Android6.0标准的权限机制内，而是AppOpsManager里面已经被隐藏了的一个开关位，对应于第24个开关。需要注意的是，AppOpsManager这个类很早就有了，但是很多ROM隐藏了checkOp的方法，好在最后发现通过反射仍旧可以调用这个方法检测权限是否打开。

AppOpsManager manager = (AppOpsManager) context.getSystemService("appops");

try {

Object object = invokeMethod(manager, "checkOp", op, Binder.getCallingUid(), getPackageName(context));

return AppOpsManager.MODE\_ALLOWED == (Integer) object;

} catch (Exception e) {

MLog.e(TAG, "CheckPermission " + e.toString());

}

不过，要打开悬浮窗权限，不同ROM的路径还不一样，有的是在设置里面，有的是在系统自带的管家里面，最后我们只能根据不同的ROM，给予用户不同的引导，终于将反馈量降了下去。

# 获取权限工具类

工具类

public class MyPermissionUtil {  
 private static final String *SPLIT\_REG* = "、";  
 private static final int *REQUEST\_CODE* = 1123;  
 protected Activity activity;  
 protected Fragment fragment;  
 private RequestPermissionCallBack callback;  
  
 public MyPermissionUtil(Activity activity, RequestPermissionCallBack callback) {  
 this.activity = activity;  
 this.callback = callback;  
 }  
  
 //fragment中获取权限需要用fragment.xxx,否则会回调到activity中  
 public MyPermissionUtil(Activity activity, Fragment fragment, RequestPermissionCallBack callback) {  
 this(activity, callback);  
 this.fragment = fragment;  
 }  
  
 public void requestPermissions(final String... permissions) {  
 boolean isAllGranted = true;  
 if (Build.VERSION.*SDK\_INT* < Build.VERSION\_CODES.*M*) {  
 Log.*w*("Permissions", "hasPermissions: API version < M, returning true by default");  
 } else {  
 //如果所有权限都已授权，则直接返回授权成功,只要有一项未授权，则发起权限请求  
 for (String permission : permissions) {  
 if (ContextCompat.*checkSelfPermission*(activity, permission) == PackageManager.*PERMISSION\_DENIED*) {  
 isAllGranted = false;  
 if (sysShouldShowRequestPermissionRationale(permission)) {  
 //点击了拒绝，没有点击不再提醒  
 new AlertDialog.Builder(activity).setTitle("提示")  
 .setMessage("您好，需要如下权限：" + getTransPermissionStrs(permissions) + "请允许，否则将影响部分功能的正常使用。")  
 .setPositiveButton("确定", (dialog, which) -> sysRequestPermissions(permissions)).show();  
 } else {  
 sysRequestPermissions(permissions);  
 }  
 //只要有任意一项权限没获取，break跳出for循环  
 break;  
 }  
 }  
 }  
 if (isAllGranted) callback.granted();  
 }  
  
 public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {  
 boolean isAllGranted = true;  
 if (requestCode == *REQUEST\_CODE*) {  
 for (int i = 0; i < grantResults.length; ++i) {  
 if (grantResults[i] == PackageManager.*PERMISSION\_DENIED*) {  
 isAllGranted = false;  
 // 用户选择了“不在提示”选项，在这种情况下需要引导用户至设置页手动授权  
 if (!sysShouldShowRequestPermissionRationale(permissions[i])) {  
 goSystemSettingAuth(permissions);  
 } else {  
 //用户拒绝权限请求，但未选中“不再提示”选项  
 callback.denied();  
 }  
 //只要有任意一项权限没获取，break跳出for循环  
 break;  
 }  
 }  
 if (isAllGranted) callback.granted();  
 }  
 }  
  
 */\*\*  
 \* 去系统设置界面授权  
 \*/* private void goSystemSettingAuth(String[] permissions) {  
 new AlertDialog.Builder(activity).setTitle("提示")  
 .setMessage("获取相关权限失败:" + getTransPermissionStrs(permissions) + "将导致部分功能无法正常使用，需要到设置页面手动授权")  
 .setPositiveButton("去授权", (dialog, which) -> {  
 Intent intent = new Intent(Settings.*ACTION\_APPLICATION\_DETAILS\_SETTINGS*);  
 Uri uri = Uri.*fromParts*("package", activity.getApplicationContext().getPackageName(), null);  
 intent.setData(uri);  
 activity.startActivity(intent);  
 dialog.dismiss();  
 }).setNegativeButton("取消", (dialog, which) -> dialog.dismiss())  
 .setOnCancelListener(dialog -> callback.denied()).show();  
 }  
  
 */\*\*  
 \* 系统-判断是否否点击了不再提示  
 \*/* private boolean sysShouldShowRequestPermissionRationale(String permission) {  
 if (fragment != null) {  
 return fragment.shouldShowRequestPermissionRationale(permission);  
 } else {  
 return ActivityCompat.*shouldShowRequestPermissionRationale*(activity, permission);  
 }  
 }  
  
  
 */\*\*  
 \* 系统-开始授权请求  
 \*/* private void sysRequestPermissions(String[] permissions) {  
 if (fragment != null) {  
 fragment.requestPermissions(permissions, *REQUEST\_CODE*);  
 } else {  
 ActivityCompat.*requestPermissions*(activity, permissions, *REQUEST\_CODE*);  
 }  
 }  
  
 */\*\*  
 \* 转换权限字符串的通俗叫法  
 \*/* private String getTransPermissionStrs(String[] permissions) {  
 StringBuilder resultSb = new StringBuilder();  
 StringBuilder permissionSb = new StringBuilder();  
 for (String s : permissions) permissionSb.append(s).append(*SPLIT\_REG*);  
 String[] strs = permissionSb.toString().split(*SPLIT\_REG*);  
 if (strs.length > 0) {  
 for (String s : strs) {  
 if (s.contains("CAMERA")) {  
 resultSb.append("相机").append(*SPLIT\_REG*);  
 } else if (s.contains("EXTERNAL\_STORAGE")) {  
 resultSb.append("存储").append(*SPLIT\_REG*);  
 } else if (s.contains("CONTACT")) {  
 resultSb.append("通讯录").append(*SPLIT\_REG*);  
 } else if (s.contains("LOCATION")) {  
 resultSb.append("您的位置").append(*SPLIT\_REG*);  
 } else if (s.contains("RECORD")) {  
 resultSb.append("麦克风").append(*SPLIT\_REG*);  
 } else if (s.contains("PHONE")) {  
 resultSb.append("电话").append(*SPLIT\_REG*);  
 } else {  
 resultSb.append("其他").append(*SPLIT\_REG*);  
 }  
 }  
 resultSb.replace(resultSb.lastIndexOf(*SPLIT\_REG*), resultSb.lastIndexOf(*SPLIT\_REG*) + 1, "，");  
 }  
 return resultSb.toString();  
 }  
  
 public boolean judgeExternalStorageState() {  
 return Environment.*getExternalStorageState*().equals(Environment.*MEDIA\_MOUNTED*);  
 }  
}

使用

//region RequestPermission  
  
private void doRequestPermission() {

/\* 在Fragment中：permissionUtil = new MyPermissionUtil(

getActivity(), HomeFragment.this, new RequestPermissionCallBack(getActivity()) { ... } \*/  
 permissionUtil = new MyPermissionUtil(this, new RequestPermissionCallBack(context) {  
 @Override  
 public void granted() {  
 if (permissionUtil.judgeExternalStorageState()) {  
 doYouWant();  
 } else {  
 Toast.*makeText*(context, "设备没有SD卡！", Toast.*LENGTH\_SHORT*).show();  
 }  
 }  
 });  
 permissionUtil.requestPermissions(Manifest.permission.*CAMERA*, Manifest.permission.*READ\_EXTERNAL\_STORAGE*);  
}

@Override  
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {  
 super.onRequestPermissionsResult(requestCode, permissions, grantResults);  
 permissionUtil.onRequestPermissionsResult(requestCode, permissions, grantResults);  
}  
  
private MyPermissionUtil permissionUtil;  
  
//endregion

# easypermissions

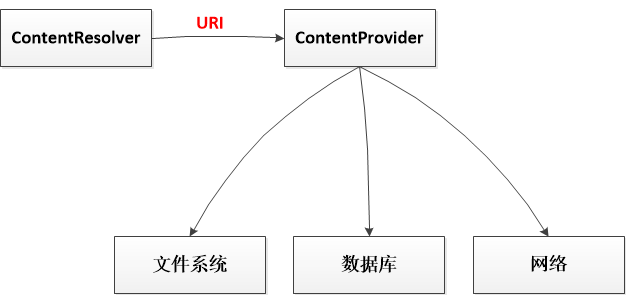
implementation 'pub.devrel:easypermissions:2.0.1'

public class EasyPermissionActivity extends BaseActivity implements EasyPermissions.PermissionCallbacks {  
 @Override  
 protected void onCreate(@Nullable Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_easy\_permission*);  
 findViewById(R.id.*btn*).setOnClickListener(v -> {  
 EasyPermissions.*requestPermissions*(this, "需要权限哟", 1123,  
 Manifest.permission.*CAMERA*, Manifest.permission.*READ\_EXTERNAL\_STORAGE*);  
 });  
 }  
 @Override  
 public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {  
 super.onRequestPermissionsResult(requestCode, permissions, grantResults);  
 EasyPermissions.*onRequestPermissionsResult*(requestCode, permissions, grantResults, this);  
 }  
 @Override  
 public void onPermissionsGranted(int requestCode, @NonNull List<String> perms) {  
 Toast.*makeText*(bActivity, "权限请求成功", Toast.*LENGTH\_SHORT*).show();  
 }  
 @Override  
 public void onPermissionsDenied(int requestCode, @NonNull List<String> perms) {  
 if (EasyPermissions.*somePermissionPermanentlyDenied*(this, perms)) {  
 new AppSettingsDialog.Builder(this).build().show();  
 }  
 }  
}

# ContentProvider

ContentProvider(数据提供者)是应用程序之间共享数据的一种接口机制，是一种更为高级的数据共享方法。ContentProvider可以指定需要共享的数据，而其他应用程序则可以在不知道数据来源、路径的情况下，对共享数据进行增删改查等操作。在Android系统中，许多Android系统内置的数据也是通过ContentProvider提供给用户使用，例如通讯录、音视频文件和图像文件等。

## ContentProvider机理



在创建ContentProvider前，首先要实现底层的数据源，数据源包括数据库、文件系统或网络等，然后继承ContentProvider类，实现基本数据操作的接口函数

## ContentResolver对象与ContentProvider的交互

在ContentResolver对象与ContentProvider进行交互时，通过URI确定要访问的ContentProvider数据集。在发起的一个请求的过程中，Android系统根据URI确定处理这个查询的ContentProvider，然后初始化ContentProvider所有需要的资源，这个初始化的工作是Android系统完成的，无需我们参与。一般情况下只有一个ContentProvider对象，但却可以同时与多个ContentResolver进行交互。

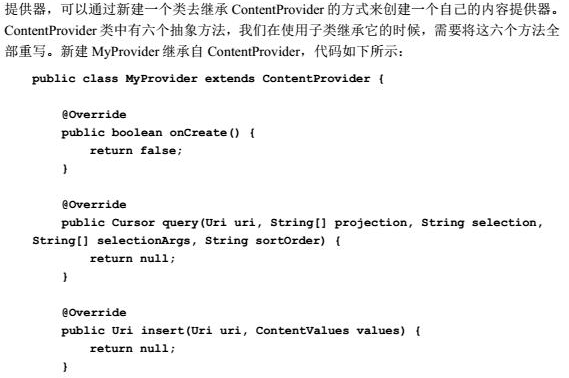
## ContentProvider的屏蔽性

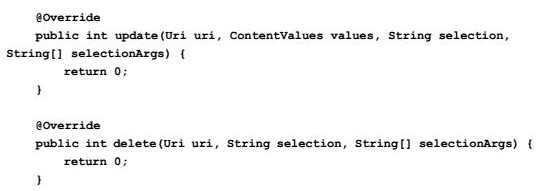
ContentProvider完全屏蔽了底层数据源的数据存储方法。数据提供者通过ContentProvider提供了一组标准的数据操作接口，但却无须知道数据提供者的内部数据的存储方法。数据提供者可以使用SQLite数据库存储数据，也可以通过文件系统或SharedPreferences存储数据，甚至是使用网络存储的方法，这些数据的存储方法和存储设备对数据使用者都是不可见的。同时，也正是这种屏蔽模式，很大程度上简化了ContentProvider的使用方法，使用者只要调用ContentProvider提供的接口函数，即可完成所有的数据操作，而数据存储方法则是ContentProvider设计者需要考虑的问题。

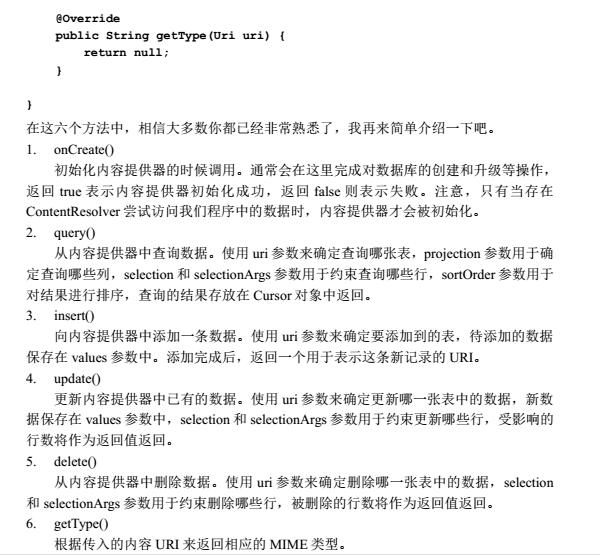
## ContentProvider提供的数据形式

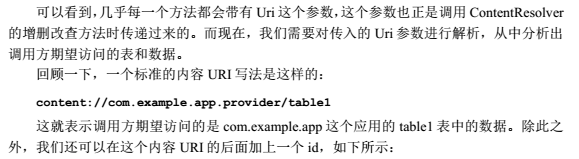
ContentProvider的数据集类似于数据库的数据表，每行是一条记录，每列具有相同的数据类型。每条记录都包含一个长整型的字段 \_ID，用来唯一标识每条记录。ContentProvider可以提供多个数据集，调用者使用URI对不同数据集的数据进行操作。

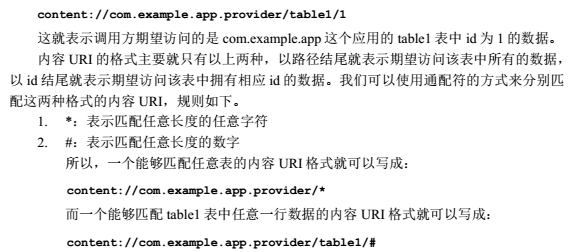
## 创建ContentProvider的步奏

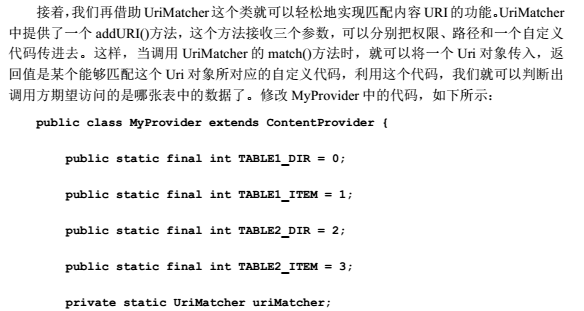




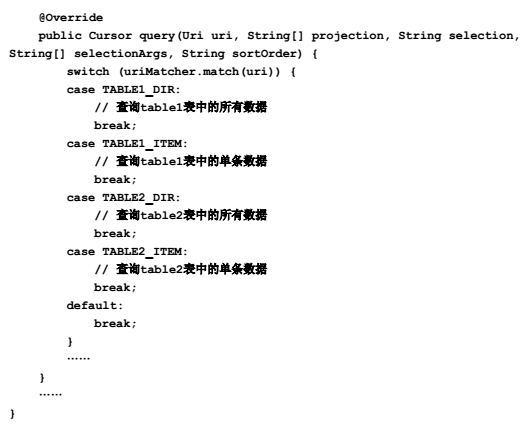


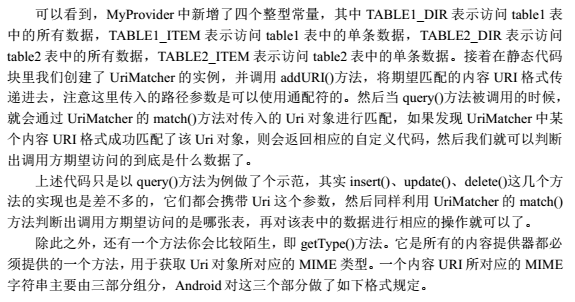
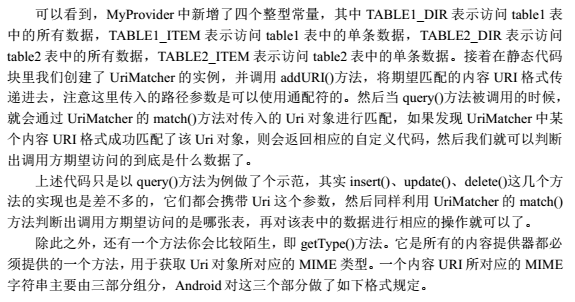


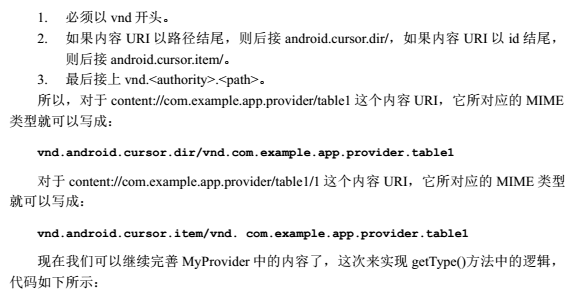


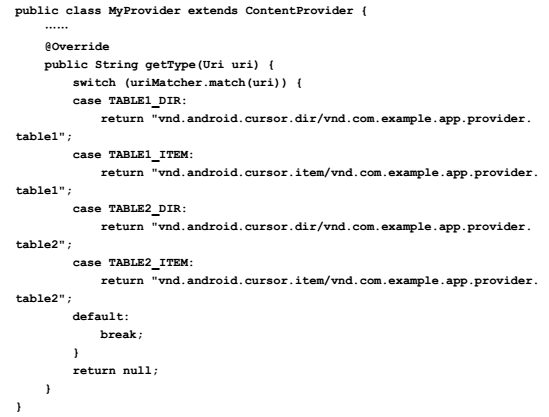




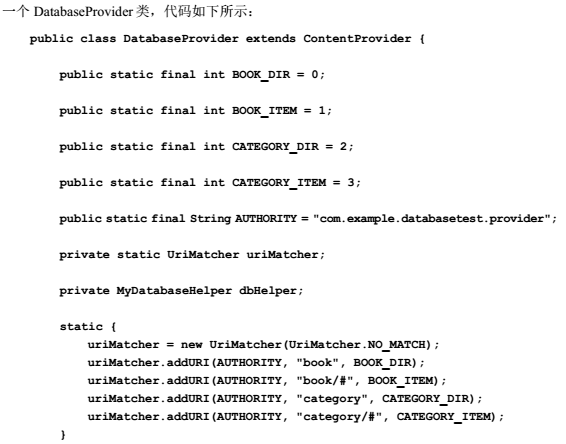


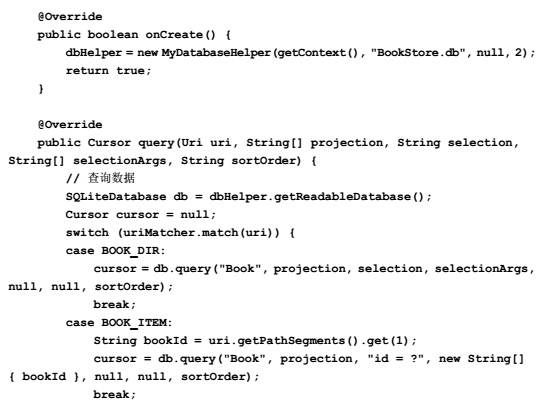


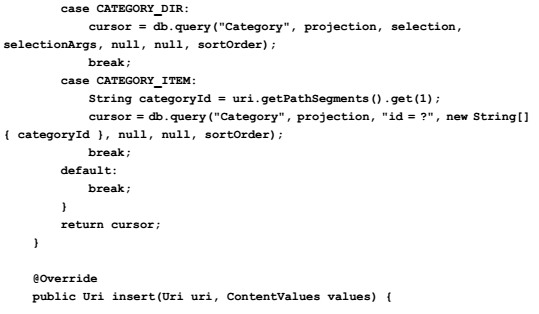


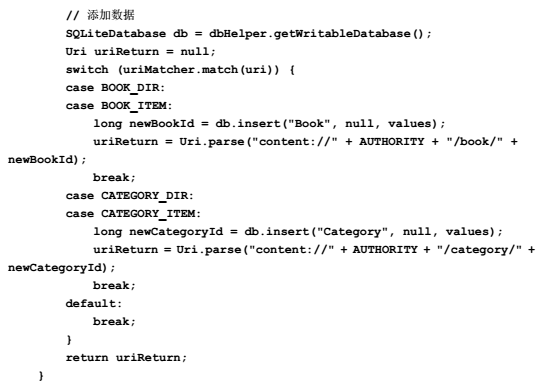


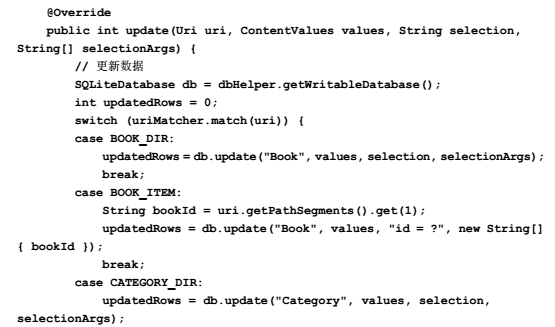
## 来个例子

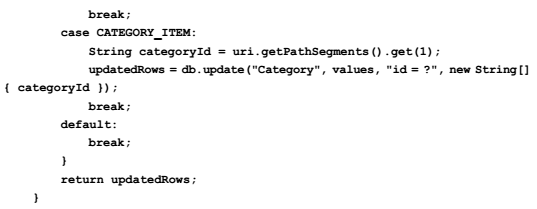


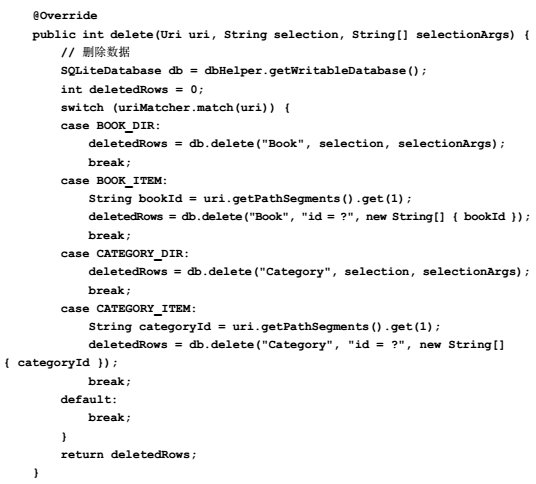


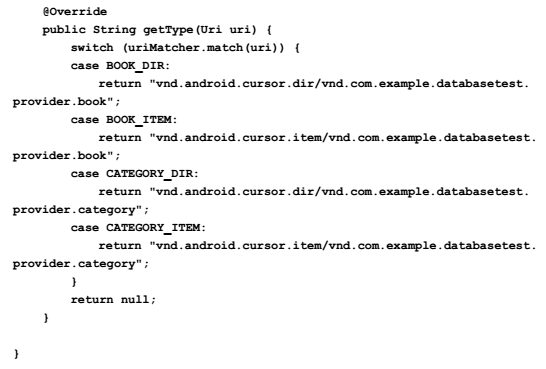


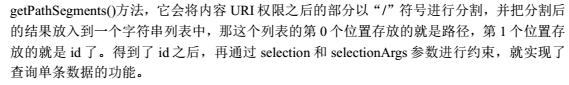














## MIME类型与getType()

### 概述：

ContentProvider中需要重写getType()函数，用于获得Uri对象对应的MIME类型，这个getType()函数有什么用呢？先看下getType()的官方说明，总体来说，就是传进去一个URI，返回一个表示MIME类型的字符串；里面还说，如果是单条记录应该返回以vnd.android.cursor.item/ 为首的字符串，如果是多条记录，应该返回vnd.android.cursor.dir/ 为首的字符串，下面先来看看MIME类型是什么

### MIME类型

#### 什么是MIME类型

根据百度百科的解释：MIME：全称Multipurpose Internet Mail Extensions，多功能Internet邮件扩充服务。它是一种多用途网际邮件扩充协议，在1992年最早应用于电子邮件系统，但后来也应用到浏览器。MIME类型就是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。看完之后，只有一个感觉……依然不懂！简单来讲，MIME类型就是用来标识当前的Activity所能打开的文件类型！

下面简单列出来系统中自带的几种文件类型和对应的MIME类型：（前面是文件名，后面是对应的MIME类型字符串）

{".bmp", "image/bmp"}

{".c", "text/plain"}

{".class", "application/octet-stream"}

{".conf", "text/plain"}

{".cpp", "text/plain"}

{".doc", "application/msword"}

#### MIME类型有什么用

那现在看看在Android中，MIME类型是用来干什么的呢？首先，MIME类型主要是Activity的Intent-filter的data域;比如下面这个Activity:

<activity

android:name=".SecondActivity"

android:label="@string/title\_activity\_second" >

<intent-filter>

<action android:name="harvic.test.qijian"/>

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

<data android:mimeType="image/bmp"/>

</intent-filter>

</activity>

这里指定了data域的MimeType值是"image/bmp"，即在利用隐式Intent匹配时，只有指定MimeType是"image/bmp"时，才能启用这个Activity，也就是说，这个Activity只能打开image/bmp类型的文件！！！！这才是MIME类型匹配的重点；所以MIME类型在Activity中是用来指定，当前的Activity所支持打开的文件类型！！

### getType()

#### 概述

现在再回过来看看ContentProvider中的getType()函数，这个函数会根据传进来的URI，生成一个代表MimeType的字符串；而此字符串的生成也有规则：

* 如果是单条记录应该返回以vnd.android.cursor.item/ 为首的字符串
* 如果是多条记录，应该返回vnd.android.cursor.dir/ 为首的字符串

至于自符串/后的字符就随便定义了。这里考虑一个问题，为什么我们返回的MimeType，要以vnd.android.cursor.item/ 或vnd.android.cursor.dir/ 开头？我们知道，MIME类型其实就是一个字符串，中间有一个 “/” 来隔开，“/”前面的部分是系统识别的部分，就相当于我们定义一个变量时的变量数据类型，通过这个“数据类型”，系统能够知道我们所要表示的是个什么东西。至于 “/” 后面的部分就是我们自已来随便定义的“变量名”了。

#### getType()与Activity的关系

上面我们讲了MIME存在于Activity的intent-filter中，那我们的getType() 跟Activity的intent-filter之间又有什么关系呢？其实，getType()返回的MIME类型，主要就是用来隐式匹配Intent的MIMETYPE域来启动Activity的。下面来看看通过URI来启用Activity的方式：

public static final String AUTHORITY = "com.harvic.provider.PeopleContentProvider";

public static final Uri CONTENT\_URI\_FIRST = Uri.parse("content://" + AUTHORITY + "/first");

public static Uri mCurrentURI = CONTENT\_URI\_FIRST;

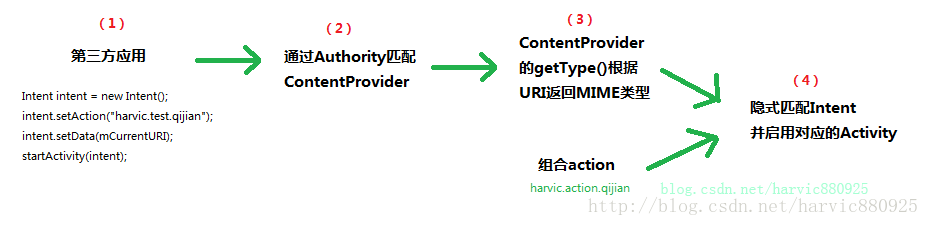
Intent intent = new Intent();

intent.setAction("harvic.test.qijian");

intent.setData(mCurrentURI);

startActivity(intent);

在上面的代码中，我们设置了action和content uri;这里利用Content URI来启用隐式启用Activity又是怎样一个流程呢？



1. 首先，第三方应用通过content Uri和action来隐式匹配Intent来启用Activity.
2. 系统通过URI中的Authority来匹配ContentProvider，从而找到我们的PeopleContentProvider。
3. 在找到PeopleContentProvider，由于我们是来匹配Intent的，所以这时候会调用getType(uri)来返回URI类型：

static {

sUriMatcher = new UriMatcher(UriMatcher.NO\_MATCH);

sUriMatcher.addURI(AUTHORITY, "first", MATCH\_FIRST);

sUriMatcher.addURI(AUTHORITY, "second", MATCH\_SECOND);

}

上面是UriMather的构造方法，由上面的代码可知，当匹配"/first"时返回MATCH\_FIRST即数值1，匹配“/second”时返回MATCH\_SECOND，即数值2，所以：

* 当匹配"/fist"时，我们返回自定义的MIME类型：vnd.android.cursor.dir/harvic.first
* 当匹配“/second”时，返回MIME类型：vnd.android.cursor.item/harvic.second

代码如下：

public String getType(Uri uri) {

switch (sUriMatcher.match(uri)){

case MATCH\_FIRST:{

return "vnd.android.cursor.dir/harvic.first";

}

case MATCH\_SECOND:{

return "vnd.android.cursor.item/harvic.second";

}

}

return null;

}

1. 下面就是根据Action和MIME类型来匹配Intent了

我们到现在在ContentProviderBlog项目中还没有一个Activity能匹配这个Action和MIME类型的，所以我们新建一个SecondActivity;

### 新建SecondActivity用来通过URI启动

在这个Activity中，我们只利用TextView来标识当前这个Activity来自ContentProviderBlog的secondActivity；XML代码:所以，我们在TextView中，将其Text属性改为"ContentProviderBlog的secondActivity"以便识别这个Activity；

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

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

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:text="ContentProviderBlog的secondActivity"

android:layout\_width="wrap\_content"

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

</RelativeLayout>

### AndroidManifest.xml

在AndroidManifest.xml中，为SecondActivity添加上隐式匹配所需要的Intent-filter;注意我们在getType（）里根据不同的URI返回了两种MIME类型，而这里的SecondActivity的data域只添加一个mimeType:vnd.android.cursor.dir/harvic.first;即当我们使用content://com.harvic.provider.PeopleContentProvider/second来隐式匹配Intent时，是没办法启用SecondActivity的，因为MIME类型不匹配！

<activity

android:name=".SecondActivity"

android:label="@string/title\_activity\_second">

<intent-filter>

<action android:name="harvic.test.qijian" />

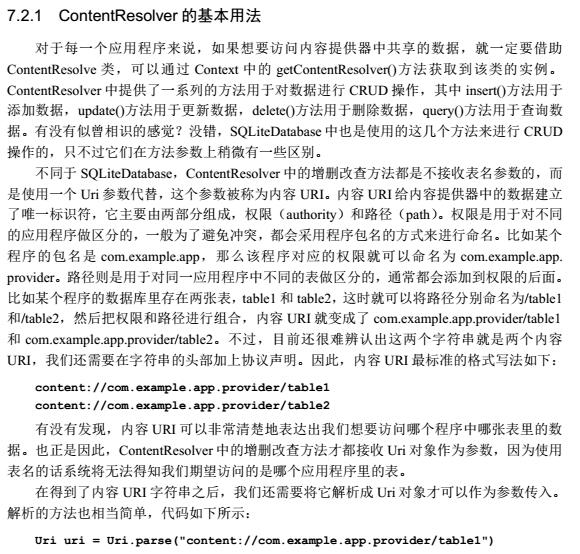
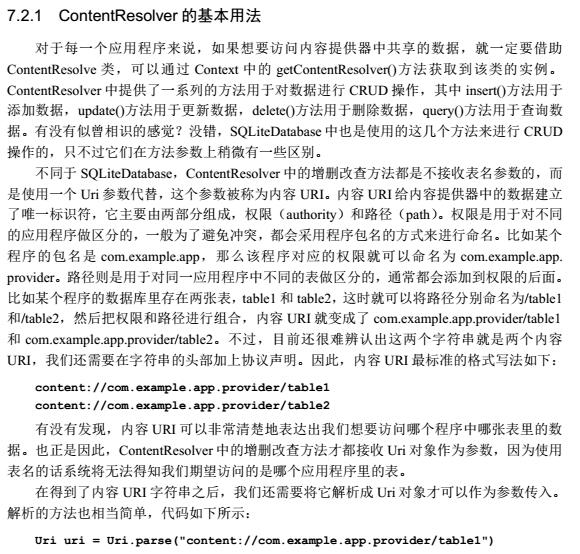
<category android:name="android.intent.category.DEFAULT" />

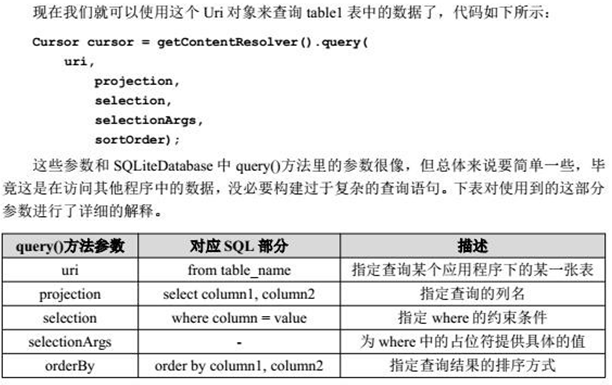
<data android:mimeType="vnd.android.cursor.dir/harvic.first" />

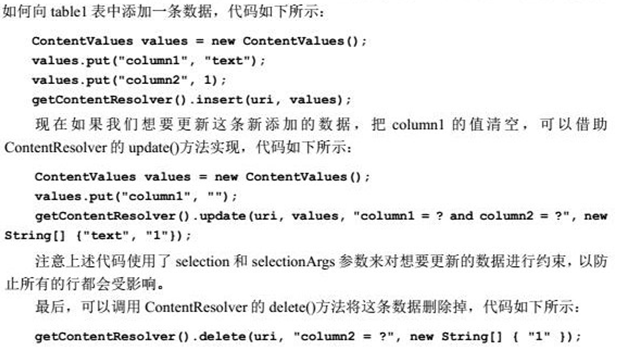
</intent-filter>

</activity>

# ContentResolver







# 原生定位（不怎么可靠）

**public** **class** **MainActivity** **extends** **Activity** {

**private** **LocationManager** manager;

**private** **Location** mLocation;

**private** **String** provider;

**private** **TextView** tv;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

tv = (**TextView**) findViewById(**R**.**id**.**tv**);

//获取位置管理器

manager = (**LocationManager**) getSystemService(**Service**.**LOCATION\_SERVICE**);

//获取可用的定位服务,network即通过基站或者wifi的方式获取位置信息，GPS不用解释。Passive，即被动方式，是位置更新监测器，当其他App获取地理位置时，向你的App进行位置更新

List<**String**> providers = manager.getProviders(**true**);

**if** (providers.contains(**LocationManager**.**GPS\_PROVIDER**)) {

provider = **LocationManager**.**GPS\_PROVIDER**;

}**else** **if** (providers.contains(**LocationManager**.**NETWORK\_PROVIDER**)) {

provider = **LocationManager**.**NETWORK\_PROVIDER**;

}**else** {

**Toast**.*makeText*(**MainActivity**.**this**, "定位服务未开启", **Toast**.**LENGTH\_SHORT**).show();

}

**if** (provider != **null**) {

mLocation = manager.getLastKnownLocation(provider);

}

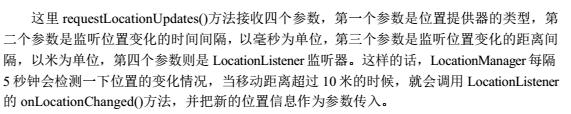
**if** (mLocation != **null**) {

show();

//设置位置变化监听

manager.requestLocationUpdates(provider, 2000, 3, listener);

}else{  
 provider = LocationManager.NETWORK\_PROVIDER;  
 manager.requestLocationUpdates(provider, 2000, 3, listener);  
}



}

//位置变化监听器

**private** LocationListener listener = **new** LocationListener() {

@Override

**public** **void** **onStatusChanged**(**String** provider, **int** status, **Bundle** extras) {

}

@Override

**public** **void** **onProviderEnabled**(**String** provider) {

}

@Override

**public** **void** **onProviderDisabled**(**String** provider) {

}

@Override

**public** **void** **onLocationChanged**(**Location** location) {

mLocation = location;

show();

}

};

//显示地理信息

**private** **void** **show**() {

**String** loationString = "latitude is:" + mLocation.getLatitude() + "\n" +

"longitude is:" + mLocation.getLongitude();

tv.setText(loationString);

}

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

manager.removeUpdates(listener);//注销监听

}

}

# 百度定位（可靠）

**public** **class** **MainActivity** **extends** **Activity** {

**private** **MLocation** mLocation;

**private** **TextView** tv, tv2;

**private** **LocationUtil** locationUtil;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

tv = (**TextView**) findViewById(**R**.**id**.**tv**);

tv2 = (**TextView**) findViewById(**R**.**id**.**tv2**);

locationUtil = **LocationUtil**.*getInstance*(**MainActivity**.**this**);

locationUtil.startMonitor();

locationUtil.setTextListener(**new** TextListener() {

@Override

**public** **void** **onBaseText**(**MLocation** mBaseLocation) {

mLocation = mBaseLocation;

**String** loationString = "latitude is:" + mLocation.latitude + "\n" +

"longitude is:" + mLocation.longitude;

tv.setText(loationString);

}

@Override

**public** **void** **onText**(**String** locString) {

tv2.setText(locString);

}

});

}

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

locationUtil.stopMonitor();

}

}

**public** **class** **LocationUtil** {

**private** **final** **static** **boolean** **DEBUG** = **true**;

**private** **final** **static** **String** **TAG** = "LocationUtil";

**private** **static** **LocationUtil** ***mInstance***;

**private** **BDLocation** mLocation = **null**;

**private** **MLocation** mBaseLocation = **new** MLocation();

**private** **Context** mContext;

**private** **String** mProvider;

**private** **LocationClient** mLocationClient;

**public** BDLocationListener myListener = **new** MyLocationListener();

**public** **interface** TextListener{

**void** **onBaseText**(**MLocation** mBaseLocation);

**void** **onText**(**String** locString);

**void** **onGetWeather**(**Weather** weather);

}

**private** TextListener textListener;

**public** **void** **setTextListener**(TextListener textListener){

**this**.textListener = textListener;

}

**private** **Handler** mHandler = **new** Handler(){

**public** **void** **handleMessage**(android.os.**Message** msg) {

textListener.onBaseText(mBaseLocation);

textListener.onGetWeather(weather);

textListener.onText(sb.toString());

}

};

**public** **static** **LocationUtil** **getInstance**(**Context** context) {

**if** (***mInstance*** == **null**) {

**synchronized** (**LocationClient**.**class**) {

**if** (***mInstance*** == **null**) {

***mInstance*** = **new** LocationUtil(context);

}

}

}

**return** ***mInstance***;

}

**private** **LocationUtil**(**Context** context) {

mLocationClient = **new** LocationClient(context.getApplicationContext());

initParams();

}

/\*\*

\* start：启动定位SDK。 stop：关闭定位SDK。调用start之后只需要等待定位结果自动回调即可。

\* 开发者定位场景如果是单次定位的场景，在收到定位结果之后直接调用stop函数即可。

\* 如果stop之后仍然想进行定位，可以再次start等待定位结果回调即可。

\* 如果开发者想按照自己逻辑请求定位，可以在start之后按照自己的逻辑请求locationclient.requestLocation()函数，会主动触发定位SDK内部定位逻辑，等待定位回调即可。

\*/

**public** **void** **startMonitor**() {

**if** (**DEBUG**) **Log**.*d*(**TAG**, "start monitor location");

**if** (!mLocationClient.isStarted()) {

mLocationClient.registerLocationListener(myListener);

mLocationClient.start();

}

**if** (mLocationClient != **null** && mLocationClient.isStarted()) {

mLocationClient.requestLocation();

} **else** {

**Log**.*d*("LocSDK3", "locClient is null or not started");

}

}

**public** **void** **stopMonitor**() {

**if** (**DEBUG**) **Log**.*d*(**TAG**, "stop monitor location");

**if** (mLocationClient != **null** && mLocationClient.isStarted()) {

mLocationClient.unRegisterLocationListener(myListener);

mLocationClient.stop();

}

}

**public** **BDLocation** **getLocation**() {

**if** (**DEBUG**) **Log**.*d*(**TAG**, "get location");

**return** mLocation;

}

**public** **MLocation** **getBaseLocation**() {

**if** (**DEBUG**) **Log**.*d*(**TAG**, "get location");

**return** mBaseLocation;

}

**private** **void** **initParams**() {

**LocationClientOption** option = **new** LocationClientOption();

//option.setLocationMode(LocationMode.Hight\_Accuracy);//可选，默认高精度，设置定位模式，高精度，低功耗，仅设备

option.setCoorType("bd09ll");//返回的定位结果是百度经纬度,默认值gcj02

option.setScanSpan(5000);//可选，默认0，即仅定位一次，设置发起定位请求的间隔需要大于等于1000ms才是有效的

option.setIsNeedAddress(**true**);//可选，设置是否需要地址信息，默认不需要

option.setOpenGps(**true**);//可选，默认false,设置是否使用gps

option.setLocationNotify(**true**);//可选，默认false，设置是否当gps有效时按照1S1次频率输出GPS结果

option.setIsNeedLocationDescribe(**true**);//可选，默认false，设置是否需要位置语义化结果，可以在BDLocation.getLocationDescribe里得到，结果类似于“在北京天安门附近”

option.setIsNeedLocationPoiList(**true**);//可选，默认false，设置是否需要POI结果，可以在BDLocation.getPoiList里得到

option.setIgnoreKillProcess(**false**);//可选，默认true，定位SDK内部是一个SERVICE，并放到了独立进程，设置是否在stop的时候杀死这个进程，默认不杀死

option.SetIgnoreCacheException(**false**);//可选，默认false，设置是否收集CRASH信息，默认收集

option.setEnableSimulateGps(**false**);//可选，默认false，设置是否需要过滤gps仿真结果，默认需要

mLocationClient.setLocOption(option);

// option.setAddrType("all");//返回的定位结果包含地址信息

// option.disableCache(true);//禁止启用缓存定位

}

**public** **class** **MyLocationListener** **implements** BDLocationListener {

@Override

**public** **void** **onReceiveLocation**(**BDLocation** location) {

**if** (location == **null**) {

**return** ;

}

mLocation = location;

mBaseLocation.latitude = mLocation.getLatitude();

mBaseLocation.longitude = mLocation.getLongitude();

**if** (textListener != **null**) {

textListener.onBaseText(mBaseLocation);

}

//Receive Location

**StringBuffer** sb = **new** StringBuffer(256);

sb.append("time : ");

sb.append(location.getTime());

sb.append("\nerror code : ");

sb.append(location.getLocType());

sb.append("\nlatitude : ");

sb.append(location.getLatitude());

sb.append("\nlontitude : ");

sb.append(location.getLongitude());

sb.append("\nradius : ");

sb.append(location.getRadius());

**if** (location.getLocType() == **BDLocation**.**TypeGpsLocation**){// GPS定位结果

sb.append("\nspeed : ");

sb.append(location.getSpeed());// 单位：公里每小时

sb.append("\nsatellite : ");

sb.append(location.getSatelliteNumber());

sb.append("\nheight : ");

sb.append(location.getAltitude());// 单位：米

sb.append("\ndirection : ");

sb.append(location.getDirection());// 单位度

sb.append("\naddr : ");

sb.append(location.getAddrStr());

sb.append("\ndescribe : ");

sb.append("gps定位成功");

} **else** **if** (location.getLocType() == **BDLocation**.**TypeNetWorkLocation**){// 网络定位结果

sb.append("\naddr : ");

sb.append(location.getAddrStr());

//运营商信息

sb.append("\noperationers : ");

sb.append(location.getOperators());

sb.append("\ndescribe : ");

sb.append("网络定位成功");

} **else** **if** (location.getLocType() == **BDLocation**.**TypeOffLineLocation**) {// 离线定位结果

sb.append("\ndescribe : ");

sb.append("离线定位成功，离线定位结果也是有效的");

} **else** **if** (location.getLocType() == **BDLocation**.**TypeServerError**) {

sb.append("\ndescribe : ");

sb.append("服务端网络定位失败，可以反馈IMEI号和大体定位时间到loc-bugs@baidu.com，会有人追查原因");

} **else** **if** (location.getLocType() == **BDLocation**.**TypeNetWorkException**) {

sb.append("\ndescribe : ");

sb.append("网络不同导致定位失败，请检查网络是否通畅");

} **else** **if** (location.getLocType() == **BDLocation**.**TypeCriteriaException**) {

sb.append("\ndescribe : ");

sb.append("无法获取有效定位依据导致定位失败，一般是由于手机的原因，处于飞行模式下一般会造成这种结果，可以试着重启手机");

}

sb.append("\nlocationdescribe : ");

sb.append(location.getLocationDescribe());// 位置语义化信息

List<**Poi**> list = location.getPoiList();// POI数据

**if** (list != **null**) {

sb.append("\npoilist size = : ");

sb.append(list.size());

**for** (**Poi** p : list) {

sb.append("\npoi= : ");

sb.append(p.getId() + " " + p.getName() + " " + p.getRank());

}

}

**Log**.*i*("BaiduLocationApiDem", sb.toString());

**if** (**DEBUG**) **Log**.*d*(**TAG**, "" + sb);

**if** (textListener != **null**) {

textListener.onText(sb.toString());

}

}

}

**public** **class** **MLocation** {

**public** **double** latitude;

**public** **double** longitude;

}

}

</application>

...

<!-- 这个权限用于进行网络定位-->

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"></uses-permission>

<!-- 这个权限用于访问GPS定位-->

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION"></uses-permission>

<!-- 用于访问wifi网络信息，wifi信息会用于进行网络定位-->

<uses-permission android:name="android.permission.ACCESS\_WIFI\_STATE"></uses-permission>

<!-- 获取运营商信息，用于支持提供运营商信息相关的接口-->

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"></uses-permission>

<!-- 这个权限用于获取wifi的获取权限，wifi信息会用来进行网络定位-->

<uses-permission android:name="android.permission.CHANGE\_WIFI\_STATE"></uses-permission>

<!-- 用于读取手机当前的状态-->

<uses-permission android:name="android.permission.READ\_PHONE\_STATE"></uses-permission>

<!-- 写入扩展存储，向扩展卡写入数据，用于写入离线定位数据-->

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"></uses-permission>

<!-- 访问网络，网络定位需要上网-->

<uses-permission android:name="android.permission.INTERNET" />

<!-- SD卡读取权限，用户写入离线定位数据-->

<uses-permission android:name="android.permission.MOUNT\_UNMOUNT\_FILESYSTEMS"></uses-permission>

<application

android:allowBackup=*"true"*

...

<service

android:name=*"com.baidu.location.f"*

android:enabled=*"true"*

android:process=*":remote"* >

</service>

<meta-data

android:name=*"com.baidu.lbsapi.API\_KEY"*

android:value=*"nGj5RqcMhLu6rB0hUS7XgVTLl4bIhUMP"* />

<activity

...

</application>

# 天气预报

**在获取地理位置的基础上，调用httpget请求，获取天气**

**private** **static** **final** **String** **akString** = "ak=APgpDrijEeEv6BCjKhxiLXUOWzCeGnsS";

**private** **static** **final** **String** **mcodeString** = "mcode=3A:35:6B:B8:F5:7C:BA:7F:19:E2:08:24:A1:9E:EC:2A:13:97:D8:23;com.example.baidumapandlbs";

**private** **static** **final** **String** **weatherurl** = "http://api.map.baidu.com/telematics/v3/weather?location=%s&output=json&" + **akString** + "&coord\_type=bd09ll&" + **mcodeString**;

**private** **Handler** mHandler = **new** Handler(){

**public** **void** **handleMessage**(android.os.**Message** msg) {

textListener.onBaseText(mBaseLocation);

textListener.onGetWeather(weather);

textListener.onText(sb.toString());

}

};

**public** **class** **MyLocationListener** **implements** BDLocationListener {

@Override

**public** **void** **onReceiveLocation**(**BDLocation** location) {

......

**final** **String** locationStr = mLocation.getLongitude() + "," + mLocation.getLatitude();

**new** Thread(**new** Runnable() {

**public** **void** **run**() {

**try** {

**if** (textListener != **null**) {

weather = getWeather(locationStr);

mHandler.sendEmptyMessage(**SUCCESSED**);

}

} **catch** (**JSONException** e) {

e.printStackTrace();

}

}

}).start();

....

**private** **Weather** **getWeather**(**String** param) **throws** JSONException {

**String** result = "";

**Weather** weatherResult = **null**;

**BufferedReader** in = **null**;

**try** {

**String** urlNameString = **String**.*format*(**weatherurl**, param);

**URL** realUrl = **new** URL(urlNameString);

**URLConnection** connection = realUrl.openConnection();

// 设置通用的请求属性

connection.setRequestProperty("accept", "\*/\*");

connection.setRequestProperty("connection", "Keep-Alive");

connection.setRequestProperty("user-agent", "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;SV1)");

connection.connect();

// 获取所有响应头字段

Map<**String**, List<**String**>> map = connection.getHeaderFields();

// 遍历所有的响应头字段

**for** (**String** key : map.keySet()) {

**System**.**out**.println(key + "--->" + map.get(key));

}

in = **new** BufferedReader(**new** InputStreamReader(connection.getInputStream()));

**String** line;

**while** ((line = in.readLine()) != **null**) {

result += line;

}

**Log**.*e*("tag", result);

} **catch** (**Exception** e) {

e.printStackTrace();

}

**finally** {

**if** (in != **null**) {

**try** {

in.close();

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

}

**JSONObject** resultJson = **new** JSONObject(result);

**if** (resultJson.has("status") && resultJson.getString("status").equals("success")) {

**JSONArray** results = resultJson.getJSONArray("results");

**if** (results.length() > 0) {

weatherResult = **new** Weather();

**JSONObject** result0 = results.getJSONObject(0);

**String** cityStr = result0.getString("currentCity");

**String** pmStr = result0.getString("pm25");

weatherResult.city = cityStr;

weatherResult.pm = pmStr;

**JSONArray** weather\_data = result0.getJSONArray("weather\_data");

**if** (weather\_data.length() > 0) {

**JSONObject** weatherjson = weather\_data.getJSONObject(0);

**String** weatherStr = weatherjson.getString("weather");

**String** windStr = weatherjson.getString("wind");

**String** temperatureStr = weatherjson.getString("temperature");

weatherResult.weather = weatherStr;

weatherResult.wind = windStr;

weatherResult.temperature = temperatureStr;

}

}

}

**return** weatherResult;

}

**public** **static** **class** **Weather** {

**public** **String** city;

**public** **String** pm;

**public** **String** wind;

**public** **String** weather;

**public** **String** temperature;

@Override

**public** **String** **toString**() {

**return** **new** StringBuilder().append("城市:").append(city).append("; pm:").append(pm).append("; 风向:").append(wind).append("; 天气:").append(weather).append("; 温度:").append(temperature).toString();

}

**public** **String** **getweatherString**() {

**return** weather;

}

**public** **String** **gettemperatureString**() {

**return** temperature;

}

**public** **String** **getcityString**() {

**return** city;

}

}

**API地址：http://lbsyun.baidu.com/index.php?title=car/api/weather#.E6.8E.A5.E5.8F.A3.E8.AF.B4.E6.98.8E**

# 获取城市列表和天气

/\*\*

\* 根据传入的代号和类型从服务器上查询省市县数据。

\*/

**private** **void** **queryFromServer**(**final** **String** code) {

**String** address;

**if** (!**TextUtils**.*isEmpty*(code)) {

address = "http://www.weather.com.cn/data/list3/city" + code + ".xml";

} **else** {

address = "http://www.weather.com.cn/data/list3/city.xml";

}

**HttpUtil**.*sendHttpRequest*(address, **new** HttpCallbackListener() {

@Override

**public** **void** **onFinish**(**final** **String** response) {

runOnUiThread(**new** Runnable() {

@Override

**public** **void** **run**() {

dataList.clear();

codeList.clear();

**if** (!**TextUtils**.*isEmpty*(response)) {

**String**[] all = response.split(",");

**if** (all != **null** && all.length > 0) {

**for** (**String** p : all) {

**String**[] array = p.split("\\|");

codeList.add(array[0]);

dataList.add(array[1]);

}

}

}

});

}

});

}

**String** address = "http://www.weather.com.cn/data/list3/city" + codeList.get(index) + ".xml";

queryFromServer2(address, "countyCode");

/\*\*

\* 根据传入的地址和类型去向服务器查询天气代号或者天气信息。

\*/

**private** **void** **queryFromServer2**(**final** **String** address, **final** **String** type) {

**HttpUtil**.*sendHttpRequest*(address, **new** HttpCallbackListener() {

@Override

**public** **void** **onFinish**(**final** **String** response) {

**if** ("countyCode".equals(type)) {

**if** (!**TextUtils**.*isEmpty*(response)) {

// 从服务器返回的数据中解析出天气代号

**String**[] array = response.split("\\|");

**if** (array != **null** && array.length == 2) {

**String** weatherCode = array[1];

**String** address = "http://www.weather.com.cn/data/cityinfo/" + weatherCode + ".html";

queryFromServer2(address, "weatherCode");

}

}

} **else** **if** ("weatherCode".equals(type)) {

runOnUiThread(**new** Runnable() {

@Override

**public** **void** **run**() {

**JSONObject** jsonObject;

**try** {

jsonObject = **new** JSONObject(response);

**JSONObject** weatherInfo = jsonObject.getJSONObject("weatherinfo");

**StringBuilder** builder = **new** StringBuilder();

builder.append("city----: " + weatherInfo.getString("city") + "\n");

builder.append("cityid----: " + weatherInfo.getString("cityid") + "\n");

builder.append("temp2----: " + weatherInfo.getString("temp2") + "\n");

builder.append("temp1----: " + weatherInfo.getString("temp1") + "\n");

builder.append("weather----: " + weatherInfo.getString("weather") + "\n");

builder.append("ptime----: " + weatherInfo.getString("ptime"));

tv1.setText(builder.toString());

} **catch** (**JSONException** e) {

e.printStackTrace();

}

}

});

}

}

});

}

}

# AsyncTask

AsyncTask定义了三种泛型类型 Params，Progress和Result。

Params 启动任务执行的输入参数，比如HTTP请求的URL。

Progress 后台任务执行的百分比。

Result 后台执行任务最终返回的结果，比如String。

doInBackground(Params…) 后台执行，比较耗时的操作都可以放在这里。注意这里不能直接操作UI。此方法在后台线程执行，完成任务的主要工作，通常需要较长的时间。在执行过程中可以调用publicProgress(Progress…)来更新任务的进度。

onPostExecute(Result)  相当于Handler 处理UI的方式，在这里面可以使用在doInBackground 得到的结果处理操作UI。 此方法在主线程执行，任务执行的结果作为此方法的参数返回

任务实例必须创建在 UI线程

[execute(Params...)](http://developer.android.com/reference/android/os/AsyncTask.html" \l "execute(Params...))必须在UI线程上调用

## 代码：

/\*\*

\* 生成该类的对象，并调用execute方法之后

\* 首先执行的是onProExecute方法

\* 其次执行doInBackgroup方法

\*/

**public** **class** **ProgressBarAsyncTask** **extends** **AsyncTask**<**Integer**, **Integer**, **String**> {

**private** **TextView** textView;

**private** **ProgressBar** progressBar;

**public** **ProgressBarAsyncTask**(**TextView** textView, **ProgressBar** progressBar) {

**super**();

**this**.textView = textView;

**this**.progressBar = progressBar;

}

/\*\*

\* 这里的Integer参数对应AsyncTask中的第一个参数

\* 这里的String返回值对应AsyncTask的第三个参数

\* 该方法并不运行在UI线程当中，主要用于异步操作，所有在该方法中不能对UI当中的空间进行设置和修改

\* 但是可以调用publishProgress方法触发onProgressUpdate对UI进行操作

\*/

@Override

**protected** **String** **doInBackground**(**Integer**... params) {

**NetOperator** netOperator = **new** NetOperator();

**int** i = 0;

**for** (i = 10; i <= 100; i+=10) {

netOperator.operator();

publishProgress(params[0].intValue());

}

**return** params[0].intValue() + "";

}

/\*\*

\* 这里的String参数对应AsyncTask中的第三个参数（也就是接收doInBackground的返回值）

\* 在doInBackground方法执行结束之后在运行，并且运行在UI线程当中 可以对UI空间进行设置

\*/

@Override

**protected** **void** **onPostExecute**(**String** result) {

textView.setText("异步操作执行结束" + result);

}

//该方法运行在UI线程当中,并且运行在UI线程当中 可以对UI空间进行设置

@Override

**protected** **void** **onPreExecute**() {

textView.setText("开始执行异步线程");

}

/\*\*

\* 这里的Intege参数对应AsyncTask中的第二个参数

\* 在doInBackground方法当中，，每次调用publishProgress方法都会触发onProgressUpdate执行

\* onProgressUpdate是在UI线程中执行，所有可以对UI空间进行操作

\*/

@Override

**protected** **void** **onProgressUpdate**(**Integer**... values) {

// int vlaue = values[0];

// progressBar.setProgress(vlaue);

textView.setText(values[0] + "");

}

## 请求网络的AsyncTask实例

**1.public** **interface** CgyBacks<T> {

**public** **void** **onResult**(T t);

}

**2.public** **class** **CgyAsyncTask**<T> **extends** **AsyncTask**<**Void**, **Void**, T>{

**private** **String** username;

**private** **String** pwd;

**private** CgyBacks<T> cgyBacks;

**public** **CgyAsyncTask**(**String** username , **String** pwd, CgyBacks<T> cgyBacks) {

**this**.cgyBacks = cgyBacks;

**this**.username = username;

**this**.pwd = pwd;

}

**public** **void** **doAsyncTack**() {

**this**.execute();

}

@Override

**protected** T **doInBackground**(**Void**... params) {

T result = (T) **HttpUtil**.requestByGet(username , pwd);

**return** result;

}

@Override

**protected** **void** **onPostExecute**(T result) {

cgyBacks.onResult(result);

**super**.onPostExecute(result);

}

}

**3.case** **R**.**id**.**btn1**:

**CgyAsyncTask** cgyTask = **new** CgyAsyncTask<**Object**>("helloworld", "android", **new** CgyBacks<**Object**>() {

@Override

**public** **void** **onResult**(**Object** t) {

**if** (t != **null**) {

tx = t.toString();

tv.setText(tx);

}

}

});

cgyTask.doAsyncTack();

**break**;

}

# 上传图片

**public** **class** **UpImgUtil** {

**private** **String** end = "\r\n";

**private** **String** twoHyphens = "--";

**private** **String** boundary = "\*\*\*\*\*";

**private** **String** newName = "image.jpg";

/\*\*

\* 上传图片方法

\* \*/

**public** **String** **UpRun**(**ArrayList**<**BasicNameValuePair**> dataArr, **Bitmap** bitmap){

**String** result = "ds";

**String** path = "http://www.baidu.com/";

**try** {

**URL** url = **new** URL(path);//新建一个URL对象

//打开一个HttpClientConnection连接

**HttpURLConnection** urlConn = (**HttpURLConnection**) url.openConnection();

urlConn.setConnectTimeout(5 \* 1000);//设置连接超时时间

urlConn.setRequestMethod("POST");//设置请求方式

urlConn.setUseCaches(**false**); // Post请求不能使用缓存 ，因为易出错

//设置http头 setRequestProperty

urlConn.setRequestProperty("Connection", "Keep-Alive");

urlConn.setRequestProperty("Charset", "UTF-8");

urlConn.setRequestProperty("contentType", "UTF-8");

urlConn.setRequestProperty("Accept-Charset", "UTF-8");

urlConn.setRequestProperty("Content-Type", "multipart/form-data;boundary=" + boundary);

urlConn.setDoOutput(**true**); //Post请求 必须 设置允许输出

**OutputStream** os = urlConn.getOutputStream(); // 获取输出流

**DataOutputStream** ds = **new** DataOutputStream(os);// 获取数据输出流

//开始拼凑头头

ds.writeBytes(twoHyphens + boundary + end);

ds.writeBytes("Content-Disposition: form-data; " + "name=\"image\";filename=\"" + newName + "\"" + ";boundary=" + boundary);

ds.writeBytes(end + end);

// 也可以上传文件

// FileInputStream fStream =new FileInputStream(uploadFile);

//开始压缩图片

**ByteArrayOutputStream** baos = **new** ByteArrayOutputStream();

bitmap.compress(**Bitmap**.CompressFormat.**PNG**, 100, baos);//压缩图片到输出流

//图片的输出流写到输入流,为了缓冲

**InputStream** fStream = **new** ByteArrayInputStream(baos.toByteArray());

//缓冲流上传

**byte**[] buffer = **new** **byte**[1024];

**int** length = 0;

**while** ((length = fStream.read(buffer)) != -1) {//从文件读取数据至缓冲区

ds.write(buffer, 0, length);//继续拼凑，将资料写入DataOutputStream中

}

ds.writeBytes(end);

fStream.close();

//图片上传完毕

//开始上传表单字段数据

addFormField(dataArr, ds); //添加表单字段内容

ds.writeBytes(twoHyphens + boundary + twoHyphens + end + end);

ds.flush();//全部上传完毕

**if** (urlConn.getResponseCode() == 200) {//判断请求是否成功

**InputStream** is = urlConn.getInputStream();//获取输入流

result = streamToString(is);//流转换成字符串

}**else** {

}

} **catch** (**Exception** e) {

}

**return** result;

}

/\*\*

\* 添加表单字段内容

\* \*/

**private** **void** **addFormField**(**ArrayList**<**BasicNameValuePair**> params, **DataOutputStream** output) {

**String** lineEnd = **System**.getProperty("line.separator");//在这个平台下行与行之间的分隔符,相当于“\n”

**StringBuilder** sb = **new** StringBuilder();

**for** (**int** i = 0; i < params.size(); i++) {

sb.append(twoHyphens + boundary + end + "Content-Disposition: form-data; name=\"" + params.get(i).getName() + "\"");

sb.append(lineEnd + end);

sb.append(params.get(i).getValue() + lineEnd);

}

**try** {

output.writeBytes(sb.toString());

} **catch** (**Exception** e) {

}

}

# 长连接socket

**public** **class** **MainActivity** **extends** **Activity** **implements** Runnable{

**private** **TextView** tv\_msg = **null**;

**private** **EditText** ed\_msg = **null**;

**private** **Button** btn\_send = **null**;

**private** **static** **final** **String** **HOST** = "10.0.2.2";

**private** **static** **final** **int** **PORT** = 9999;

**private** **Socket** socket = **null**;

**private** **BufferedReader** in = **null**;

**private** **PrintWriter** out = **null**;

**private** **String** content = "";

**private** **Context** context;

**Handler** handler = **new** Handler(){

**public** **void** **handleMessage**(android.os.**Message** msg) {

tv\_msg.setText(content);

}

};

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

context = **MainActivity**.**this**;

tv\_msg = (**TextView**) findViewById(**R**.**id**.**TextView**);

ed\_msg = (**EditText**) findViewById(**R**.**id**.**EditText01**);

btn\_send = (**Button**) findViewById(**R**.**id**.**Button02**);

**try** {

socket = **new** Socket(**HOST**, **PORT**);

in = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));

out = **new** PrintWriter(**new** BufferedWriter(**new** OutputStreamWriter(socket.getOutputStream())), **true**);

} **catch** (**Exception** e) {

**Toast**.makeText(context, "出错了", 1000).show();

}

btn\_send.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**String** msg = ed\_msg.getText().toString();

**if** (socket.isConnected()) {

**if** (!socket.isOutputShutdown()) {

out.println(msg);

}

}

}

});

**//启动线程，接收服务器发送过来的数据**

**new** Thread(**MainActivity**.**this**).start();

}

@Override

**public** **void** **run**() {

**try** {

**while** (**true**) {**//**

**if** (!socket.isClosed()) {

**if** (socket.isConnected()) {

**if** (!socket.isInputShutdown()) {

**if** ((content = in.readLine()) != **null**) {

content += "\n";

handler.sendMessage(handler.obtainMessage());

**//handler.sendMessage(new Message());**

} **else** {

}

}

}

}

}

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

**/\* 解析：除了isClose方法，Socket类还有一个isConnected方法来判断Socket对象是否连接成功。 看到这个名字，也许读者会产生误解。**

**\* 其实isConnected方法所判断的并不是Socket对象的当前连接状态， 而是Socket对象是否曾经连接成功过，如果成功连接过，即使现**

**\* 在isClose返回true， isConnected仍然返回true。因此，要判断当前的Socket对象是否处于连接状态， 必须同时使用isClose和**

**\* isConnected方法， 即只有当isClose返回false，isConnected返回true的时候Socket对象才处于连接状态。 虽然在大多数的时候**

**\* 可以直接使用Socket类或输入输出流的close方法关闭网络连接，但有时我们只希望关闭OutputStream或InputStream，而在关闭输**

**\* 入输出流的同时，并不关闭网络连接。这就需要用到Socket类的另外两个方法：shutdownInput和shutdownOutput，**

**\* 这两个方法只关闭相应的输入、输出流，而它们并没有同时关闭网络连接的功能。和isClosed、isConnected**

**\* 方法一样，Socket类也提供了两个方法来判断Socket对象的输入、输出流是否被关闭，这两个方法是**

**\* isInputShutdown()和isOutputShutdown()。shutdownInput和shutdownOutput并不影响Socket对象的状态。\*/**

}

# WebView显示网页

## 加载本地/Web资源

**加载本地资源**

webView.loadUrl("file:///android\_asset/example.html");//example.html 存放在assets文件夹内

**加载web资源：**

webView.loadUrl("http://baidu.com");

**加载URL的总结就是：**

1、如果是在线网址记得添加网络访问权限

2、在线网址中，如果要使用webview打开，记得设置WebViewClient

3、打开本地html文件时，是不需要设置WebViewClient，对应的asstes目录的url为：[file:///android\_asset/xxxxx](file://android_asset\\xxxxx)

## WebView基本设置

如果我们需要设置WebView的属性，是通过WebView.getSettings()获取设置WebView的WebSettings对象,然后调用WebSettings中的方法来实现的。 WebSettings的方法及说明如下：

/\*\*

\* 是否支持缩放，配合方法setBuiltInZoomControls使用，默认true

\*/

setSupportZoom(**boolean** support)

/\*\*

\* 是否使用WebView内置的缩放组件，由浮动在窗口上的缩放控制和手势缩放控制组成，默认false

\*/

setBuiltInZoomControls(**boolean** enabled)

/\*\*

\* 基于WebView导航的类型使用缓存：正常页面加载会加载缓存并按需判断内容是否需要重新验证。

\* 如果是页面返回，页面内容不会重新加载，直接从缓存中恢复。setCacheMode允许客户端根据指定的模式来

\* 使用缓存。

\* LOAD\_DEFAULT 默认加载方式

\* LOAD\_CACHE\_ELSE\_NETWORK 按网络情况使用缓存

\* LOAD\_NO\_CACHE 不使用缓存

\* LOAD\_CACHE\_ONLY 只使用缓存

\*/

setCacheMode(**int** mode)

/\*\*

\* 是否启动概述模式浏览界面，当页面宽度超过WebView显示宽度时，缩小页面适应WebView。默认false

\*/

setLoadWithOverviewMode(**boolean** overview)

/\*\*

\* 是否支持ViewPort的meta tag属性，如果页面有ViewPort meta tag 指定的宽度，则使用meta tag指定的值，否则默认使用宽屏的视图窗口

\*/

setUseWideViewPort(**boolean** use)

/\*\*

\* 设置是否允许执行JS。

\*/

setJavaScriptEnabled(**boolean** flag)

/\*\*

\* 设置存储定位数据库的位置，考虑到位置权限和持久化Cache缓存，Application需要拥有指定路径的write权限

\*/

setGeolocationDatabasePath(String databasePath)

/\*\*

\* 是否允许Cache，默认false。考虑需要存储缓存，应该为缓存指定存储路径setAppCachePath

\*/

setAppCacheEnabled(**boolean** flag)

/\*\*

\* 设置Cache API缓存路径。为了保证可以访问Cache，Application需要拥有指定路径的write权限。

\* 该方法应该只调用一次，多次调用自动忽略。

\*/

setAppCachePath(String appCachePath)

/\*\*

\* 是否允许数据库存储。默认false。查看setDatabasePath API 如何正确设置数据库存储。

\* 该设置拥有全局特性，同一进程所有WebView实例共用同一配置。注意：保证在同一进程的任一WebView

\* 加载页面之前修改该属性，因为在这之后设置WebView可能会忽略该配置

\*/

setDatabaseEnabled(**boolean** flag)

/\*\*

\* 是否存储页面DOM结构，默认false。

\*/

setDomStorageEnabled(**boolean** flag)

/\*\*

\* 是否允许定位，默认true。注意：为了保证定位可以使用，要保证以下几点：

\* Application 需要有android.Manifest.permission#ACCESS\_COARSE\_LOCATION的权限

\* Application 需要实现WebChromeClient#onGeolocationPermissionsShowPrompt的回调，

\* 接收Js定位请求访问地理位置的通知

\*/

setGeolocationEnabled(**boolean** flag)

/\*\*

\* 指定WebView的页面布局显示形式，调用该方法会引起页面重绘。默认LayoutAlgorithm. NARROW\_COLUMNS

\*/

setLayoutAlgorithm(LayoutAlgorithm l)

/\*\*

\* 是否需要用户手势来播放Media，默认true

\*/

setMediaPlaybackRequiresUserGesture(**boolean** require)

/\*\*

\* 是否显示窗口悬浮的缩放控制，默认true

\*/

setDisplayZoomControls(**boolean** enabled)

/\*\*

\* 是否允许访问WebView内部文件，默认true

\*/

setAllowFileAccess(**boolean** allow)

/\*\*

\* 是否允许获取WebView的内容URL ，可以让WebView访问ContentPrivider存储的内容。 默认true

\*/

setAllowContentAccess(**boolean** allow)

/\*\*

\* 是否保存表单数据，默认false

\*/

setSaveFormData(**boolean** save)

/\*\*

\* 设置页面文字缩放百分比，默认100%

\*/

setTextZoom(**int** textZoom)

/\*\*

\* 是否支持多窗口，如果设置为true ，WebChromeClient#onCreateWindow方法必须被主程序实现，默认false

\*/

setSupportMultipleWindows(**boolean** support)

/\*\*

\* 设置标准的字体族，默认”sans-serif”。font-family 规定元素的字体系列。

\* font-family 可以把多个字体名称作为一个“回退”系统来保存。如果浏览器不支持第一个字体，

\* 则会尝试下一个。也就是说，font-family 属性的值是用于某个元素的字体族名称或/及类族名称的一个

\* 优先表。浏览器会使用它可识别的第一个值。

\*/

setStandardFontFamily(String font)

/\*\*

\* 设置混合字体族。默认”monospace”

\*/

setFixedFontFamily(String font)

/\*\*

\* 设置SansSerif字体族。默认”sans-serif”

\*/

setSansSerifFontFamily(String font)

/\*\*

\* 设置SerifFont字体族，默认”sans-serif”

\*/

setSerifFontFamily(String font)

/\*\*

\* 设置CursiveFont字体族，默认”cursive”

\*/

setCursiveFontFamily(String font)

/\*\*

\* 设置FantasyFont字体族，默认”fantasy”

\*/

setFantasyFontFamily(String font)

/\*\*

\* 设置最小字体，默认8. 取值区间[1-72]，超过范围，使用其上限值。

\*/

setMinimumFontSize(**int** size)

/\*\*

\* 设置最小逻辑字体，默认8. 取值区间[1-72]，超过范围，使用其上限值。

\*/

setMinimumLogicalFontSize(**int** size)

/\*\*

\* 设置默认字体大小，默认16，取值区间[1-72]，超过范围，使用其上限值。

\*/

setDefaultFontSize(**int** size)

/\*\*

\* 设置默认填充字体大小，默认16，取值区间[1-72]，超过范围，使用其上限值。

\*/

setDefaultFixedFontSize(**int** size)

/\*\*

\* 设置是否加载图片资源，注意：方法控制所有的资源图片显示，包括嵌入的本地图片资源。

\* 使用方法setBlockNetworkImage则只限制网络资源图片的显示。值设置为true后，

\* webview会自动加载网络图片。默认true

\*/

setLoadsImagesAutomatically(**boolean** flag)

/\*\*

\* 是否加载网络图片资源。注意如果getLoadsImagesAutomatically返回false，则该方法没有效果。

\* 如果使用setBlockNetworkLoads设置为false，该方法设置为false，也不会显示网络图片。

\* 当值从true改为false时。WebView会自动加载网络图片。

\*/

setBlockNetworkImage(**boolean** flag)

/\*\*

\* 设置是否加载网络资源。注意如果值从true切换为false后，WebView不会自动加载，

\* 除非调用WebView#reload().如果没有android.Manifest.permission#INTERNET权限，

\* 值设为false，则会抛出java.lang.SecurityException异常。

\* 默认值：有android.Manifest.permission#INTERNET权限时为false，其他为true。

\*/

setBlockNetworkLoads(**boolean** flag)

/\*\*

\* 是否允许Js访问任何来源的内容。包括访问file scheme的URLs。考虑到安全性，

\* 限制Js访问范围默认禁用。注意：该方法只影响file scheme类型的资源，其他类型资源如图片类型的，

\* 不会受到影响。ICE\_CREAM\_SANDWICH\_MR1版本以及以下默认为true，JELLY\_BEAN版本

\* 以上默认为false

\*/

setAllowUniversalAccessFromFileURLs(**boolean** flag)

/\*\*

\* 是否允许Js访问其他file scheme的URLs。包括访问file scheme的资源。考虑到安全性，

\* 限制Js访问范围默认禁用。注意：该方法只影响file scheme类型的资源，其他类型资源如图片类型的，

\* 不会受到影响。如果getAllowUniversalAccessFromFileURLs为true，则该方法被忽略。

\* ICE\_CREAM\_SANDWICH\_MR1版本以及以下默认为true，JELLY\_BEAN版本以上默认为false

\*/

setAllowFileAccessFromFileURLs(**boolean** flag)

/\*\*

\* 是否允许JS自动打开窗口。默认false

\*/

setJavaScriptCanOpenWindowsAutomatically(**boolean** flag)

/\*\*

\* 设置页面的编码格式，默认UTF-8

\*/

setDefaultTextEncodingName(String encoding)

/\*\*

\* 设置WebView代理，默认使用默认值

\*/

setUserAgentString(String ua)

/\*\*

\* 通知WebView是否需要设置一个节点获取焦点当

\* WebView#requestFocus(int,android.graphics.Rect)被调用的时候，默认true

\*/

setNeedInitialFocus(**boolean** flag)

/\*\*

\* 设置加载不安全资源的WebView加载行为。KITKAT版本以及以下默认为MIXED\_CONTENT\_ALWAYS\_ALLOW方

\* 式，LOLLIPOP默认MIXED\_CONTENT\_NEVER\_ALLOW。强烈建议：使用MIXED\_CONTENT\_NEVER\_ALLOW

\*/

setMixedContentMode(**int** mode)

## 一些功能技巧的使用

### 不支持长按

mWebView.setLongClickable(false);// js中不能长按

### 支持Javascript

settings.setJavaScriptEnabled(true);

### 使页面支持缩放

webSettings.setJavaScriptEnabled(true); //开启javascript支持

webSettings.setSupportZoom(true); // 设置可以支持缩放

webSettings.setBuiltInZoomControls(true); // 设置出现缩放工具

### 使字体支持缩放

webSettings.setTextZoom(100); //设置字体在页面的百分比，默认100

### 需要手动输入用户名等，则必须设置支持获取手势焦点

webview.requestFocusFromTouch();

### 打开页面时，自适应屏幕(加载网页居中)

**方法一：**

webSettings.setUseWideViewPort(true);//设置此属性，可任意比例缩放

webSettings.setLoadWithOverviewMode(true);

注意：对于我们自己写的网页代码，不必利用这处函数来做页面缩放适配，因为对于有些手机存在适配问题，只需要在HTML中做宽度100%自适应屏幕就行了

**方法二：**

webSettings.setLayoutAlgorithm(LayoutAlgorithm.SINGLE\_COLUMN);

把所有内容放在webview等宽的一列中。（可能会出现页面中链接失效）

**第三种：**

DisplayMetrics metrics = new DisplayMetrics();

getWindowManager().getDefaultDisplay().getMetrics(metrics);

int mDensity = metrics.densityDpi;

if (mDensity == 120) {

settings.setDefaultZoom(ZoomDensity.CLOSE);

}else if (mDensity == 160) {

settings.setDefaultZoom(ZoomDensity.MEDIUM);

}else if (mDensity == 240) {

settings.setDefaultZoom(ZoomDensity.FAR);

}

这方法可以让你的页面适应手机屏幕的分辨率，完整的显示在屏幕上，可以放大缩小。例如适合800px宽度的页面，如果通过WebView在1024px的屏幕宽度下显示时，内容(图片)会被放大，整体页面会超出屏幕。试了将WebView的settings中的缩放都关闭了也不行。后来发现了WebSettings.ZoomDensity这个设置，并在文档中找到了以下说明：

Enum for specifying the WebView's desired density. FAR makes 100% looking like in 240dpi MEDIUM makes 100% looking like in 160dpi CLOSE makes 100% looking like in 120dpi

这样就能很好的说明以上问题的原因了，又是密度惹的祸。默认WebView的ZoomDensity是MEDIUM，对应160dpi。而我之前800px宽度的屏幕对应的是60dpi，1024px宽度的屏幕对应的是240dpi。所以，页面在高分辨率的屏幕上被放大了1.5倍。这个和图片自缩放的机制是一样的。于是，只要通过当前屏幕的密度，动态设置该属性就能适应不同屏幕（当然，默认你的页面是针对160dpi的密度大小设计的）,即可解决：

int screenDensity = getResources().getDisplayMetrics().densityDpi ;

WebSettings.ZoomDensity zoomDensity = WebSettings.ZoomDensity.MEDIUM ;

switch (screenDensity){

case DisplayMetrics.DENSITY\_LOW :

zoomDensity = WebSettings.ZoomDensity.CLOSE;

break;

case DisplayMetrics.DENSITY\_MEDIUM:

zoomDensity = WebSettings.ZoomDensity.MEDIUM;

break;

case DisplayMetrics.DENSITY\_HIGH:

zoomDensity = WebSettings.ZoomDensity.FAR;

break ;

}

webSettings.setDefaultZoom(zoomDensity);

## Java和js交互

其实webview加载资源的速度并不慢，但是如果资源多了，当然就很慢。图片、 css 、 js 、 html 这些资源每个大概需要 10-200ms ，一般都是 30ms 就 ok 了。不过webview是必须等到全部资源都完成加载，才会进行渲染的，所以加载的速度很重要！从Google上我们了解到，webview 加载页面的顺序是：先加载 html ，然后从里面解析出 css 、 js 文件和页面上的图片资源进行加载。如果 webkit 的缓存里面有，就不加载。加载完这些资源之后，就进行 css 的渲染和 js 的执行。 Css 的渲染一般不需要很长时间，几十毫秒就 ok 。关键是 js 的执行，如果用了 jQuery ，则执行起来需要 5-6 秒。而在这段时间，如果不在 webview 里设置背景，网页部分是白色的，很难看。这是一个很糟糕的用户体验。所以如果用网页布局程序，最好别用那些庞大的 js 框架。最好使用原生的js写业务脚本，以提升加载速度，改善用户体验。

Android 的 webview 是基于 webkit 内核的，webview 中集成了js与java互调的接口函数，通过addJavascriptInterface方法，可以将Java的类注册进webkit，给网页上的js进行调用，而且还可以通过loadUrl方法是给webkit传递一个uri，供浏览器来进行解析，实现Java和js交互。

### js访问Java

**普通用法**

**public** **class** **IdReaderBase** {

...

@JavascriptInterface

**public** **void** **takeIdCard**() {

//该线程不是主线程，如果调用View的方法会报错，所以如果如果操作了ui，需要回到主线程

runOnUiThread(new Runnable() {

@Override

public void run() {

tvTitle.setText("短信验证");

tv\_appId.setVisibility(View.INVISIBLE);

}

});

}

@JavascriptInterface

**public** **void** **faceRecognition**() {

}

...

}

通过addJavascriptInterface注入IdReaderBase对象，以便JS访问其中的Public函数

mWebView.addJavascriptInterface(new IdReaderBase(context), "idReader");

Js访问java函数

window.IdReaderBase.takeIdCard() 或者使用别名 idReader.takeIdCard()

**Js调Java有参方法**

Js调Java有参方法时，传入的参数必须为String格式，特别是JSON和拼凑的出来的数据，最好用var data = JSON.stringfy(tempData)转换一下

**JAVA中如何得到JS中的返回值(Android4.4以前)**

现在我们再考虑一下，如果我们要在JAVA中需要得到JS的结果返回值要怎么办？比如在上面的例子中，我们需要在JAVA中得到在计算后的结果值 ，Android在4.4之前并没有提供直接调用js函数并获取值的方法，也就是说，我们只能调用JS中的函数，并不能得到该函数的返回值，想得到返回值我们就得想其它办法，所以在此之前，常用的思路是 java调用js方法，js方法执行完毕，再次调用java代码将值返回。

**JAVA中如何得到JS中的返回值(Android4.4之后)**

Android 4.4之后使用evaluateJavascript即可

先写一个具有返回值的js方法

function getGreetings() {

return 1;

}

java代码时用evaluateJavascript方法调用:

private void testEvaluateJavascript(WebView webView) {

webView.evaluateJavascript("getGreetings()", new ValueCallback() {

@Override

public void onReceiveValue(String value) {

Log.i(LOGTAG, "onReceiveValue value=" + value);

}

});

}

从上面的用法中很明显看到，通过evaluateJavascript调用JS中的方法，可以向其中添加结果回调，来接收JS的return值。 注意：上面限定了结果返回结果为String，对于简单的类型会尝试转换成字符串返回，对于复杂的数据类型，建议以字符串形式的json返回。evaluateJavascript方法必须在UI线程（主线程）调用，因此onReceiveValue也执行在主线程。

### Java访问js

mWebView.loadUrl("javascript: showFromHtml()");//函数需在UI线程运行，因为mWebView为UI控件

在需要传递参数时，String需要有单引号，否则是未定义的；eg：

mWebView.loadUrl("javascript:yshd.fn('" + idcardno + "')");

### 传base64的img字符串

* img含有“/n”，js识别会出错；
* 传json也会加json的编码，js识别会出错；
* 所以要先给他加url编码，并且不能传json，才能正确传递

#### 安卓4.4以下

Base64的编码可以是default

mWebView.loadUrl("javascript:yshd.fn(decodeURI('"

+ **URLEncoder**.**encode**(img) + "')" + ", '" + name

+ "', '" + idcardno + "')");

#### 安卓4.4以上

将字节数组转换成base64的字符串时，编码必须为NO\_WRAP，如果是DEFAULT 无法传递，并且不用使用URLEncoder编码了

**String** strEnrollBuf = **Base64**.*encodeToString*(EnrollBuf, **Base64**.**NO\_WRAP**);

* CRLF 这个参数看起来比较眼熟，它就是Win风格的换行符，意思就是使用CR LF这一对作为一行的结尾而不是Unix风格的LF
* DEFAULT 这个参数是默认，使用默认的方法来加密
* NO\_PADDING 这个参数是略去加密字符串最后的”=”
* NO\_WRAP 这个参数意思是略去所有的换行符（设置后CRLF就没用了）
* URL\_SAFE 这个参数意思是加密时不使用对URL和文件名有特殊意义的字符来作为加密字符，具体就是以-和\_取代+和 /

Base64字符串传给后台的时候，服务器端接收的字符串会把“+”替换成空格。因为加号(+)是BASE64编码的一部分，而加号在后台接收时被当成是空格。 所以在接收后台返回的base64处理办法是：直接把空格替换成“+”，然后解码.如果后台不用显示，可以原封传回给java，java来替换，再用base64解回byte[]

**String** veinStr = intentStart.getStringExtra("veinStr");//获取指静脉信息

veinStr = veinStr.replace(" ", "+");

veinByte = **Base64**.*decode*(veinStr, **Base64**.**NO\_WRAP**);

### Java和js交互有以下一些特点：

1．Java 调用 js 里面的函数，速度并不令人满意，大概一次一两百毫秒吧，如果要做交互性很强的事情，这种速度会让人疯掉的。而反过来就不一样了， js 去调 java 的方法，速度很快，基本上 40-50 毫秒一次。所以尽量用 js 调用 java 方法，而不是 java 去调用 js 函数。

 2．Java 调用 js 的函数，没有返回值，而 Js 调用 java 方法，可以有返回值。返回值可以是基本类型、字符串，也可以是对象。如果是字符串，有个很讨厌的问题.如果是对象，这个对象会被转换为 js 的对象，直接可以访问里面的方法。但是我不推荐 java 返回给 js 的是对象，除非是必须。因为 js 收到 java 返回的对象，会产生一些交换对象，而如果这些对象的数量增加到了 500 或 600 以上，程序就会出问题。所以尽量返回基本数据类型或者字符串。

 3．Js 调用 Java 的方法，返回值如果是字符串，你会发现这个字符串是 native 的，不能对它进行一些修改操作，比如想对它 substr ，取不到。怎么解决呢？转成 locale 的。使用 toLocaleString() 函数就可以了。不过这个函数的速度并不快，转化的字符串如果很多，将会很耗费时间。

### 混淆后交互失效

* 混淆规则加入 js公用的规则
* 并且找出与js交互有关的bean类，进行保护
* 混淆规则所匹配的类，必须加入该类的 全部路径
* 对于内部类，应该使用 $ 符号，而不是用 .
* 对于混淆出现的问题，可以通过log和mapping文件进行追踪

示例

#保留使用js交互的类中所有注入接口（由于是内部类，所以干脆保留所有内部类即可）

-keepclassmembers class com.zhualaizhuaqu.gift.ui.web.activity.RankActivity$\*{\*;}

-keep class com.zhualaizhuaqu.gift.ui.web.activity.RankActivity$\*{\*;}

-keepclassmembers class com.zhualaizhuaqu.gift.ui.web.activity.FeedBackActivity$\*{\*;}

-keep class com.zhualaizhuaqu.gift.ui.web.activity.FeedBackActivity$\*{\*;}

-keepclassmembers class com.zhualaizhuaqu.gift.ui.web.activity.MallActivity$\*{\*;}

-keep class com.zhualaizhuaqu.gift.ui.web.activity.MallActivity$\*{\*;}

#如果交互中有传某个bean的json格式，bean也不能混淆

-keepclassmembers class com.zhualaizhuaqu.gift.model.file.SharePreferenceHelper$UserInfoForWeb{\*;}

#保留JavascriptInterface中的方法

-keepclassmembers class \* {@android.webkit.JavascriptInterface <methods>;}

#保留webViewClient中的配置

-keepclassmembers class \* extends android.webkit.webViewClient {\*;}

#保留annotation， 例如 @JavascriptInterface 等 annotation

-keepattributes \*Annotation\*

#保留跟 javascript相关的属性

-keepattributes \*JavascriptInterface\*

## WebViewClient---影响View的事件的回调监听

有关各种事件的回调都被封装在WebViewClient类中了，在WebViewClient中有各种的回调方法，就是在某个事件发生时供我们监听

mWebView.setWebViewClient(new WebViewClient(){

@Override

public void onPageStarted(WebView view, String url, Bitmap favicon) {

......

}

......

});

### WVC中函数概述

/\*\*

\* 在开始加载网页时会回调

\*/

**public** **void** onPageStarted(WebView view, String url, Bitmap favicon)

/\*\*

\* 在结束加载网页时会回调

\*/

**public** **void** onPageFinished(WebView view, String url)

/\*\*

\* 拦截 url 跳转,在里边添加点击链接跳转或者操作

\*/

**public** **boolean** shouldOverrideUrlLoading(WebView view, String url)

/\*\*

\* 加载错误的时候会回调，在其中可做错误处理，比如再请求加载一次，或者提示404的错误页面

\*/

**public** **void** onReceivedError(WebView view, **int** errorCode,String description, String failingUrl)

/\*\*

\* 当接收到https错误时，会回调此函数，在其中可以做错误处理

\*/

**public** **void** onReceivedSslError(WebView view, SslErrorHandler handler,SslError error)

/\*\*

\* 在每一次请求资源时，都会通过这个函数来回调

\*/

**public** WebResourceResponse shouldInterceptRequest(WebView view, String url) {

**return** **null**;

}

### WVC之onPageStarted与onPageFinished

onPageStarted：通知主程序页面当前开始加载。该方法只有在加载main frame时加载一次，如果一个页面有多个frame，onPageStarted只在加载main frame时调用一次。也意味着若内置frame发生变化，onPageStarted不会被调用，如：在iframe中打开url链接。

onPageFinished：通知主程序页面加载结束。方法只被main frame调用一次。

我们就利用上面的想法来举个例子：开始加载时显示加载圆圈，结束加载时隐藏加载圆圈

@Override

public void onPageStarted(WebView view, String url, Bitmap favicon) {

super.onPageStarted(view, url, favicon);

mProgressDialog.show();

}

@Override

public void onPageFinished(WebView view, String url) {

super.onPageFinished(view, url);

mProgressDialog.hide();

}

### WVC之shouldOverrideUrlLoading

这个函数会在加载超链接时回调过来；所以通过重写shouldOverrideUrlLoading，可以实现对网页中超链接的拦截；

返回值是boolean类型，表示是否屏蔽WebView继续加载URL的默认行为，因为这个函数是WebView加载URL前回调的，所以如果我们return true，则WebView接下来就不会再加载这个URL了，所有处理都需要在WebView中操作，包含加载。如果我们return false，则系统就认为上层没有做处理，接下来还是会继续加载这个URL的。WebViewClient默认就是return false的

#### 在程序内打开网页

必须使用webview客户端，即：mWebView.setWebViewClient(new WebViewClient());

方法一：

返回false，让webview自己处理，或者因为默认返回false，直接设置mWebView.setWebViewClient(new WebViewClient())即可

mWebView.setWebViewClient(**new** WebViewClient(){

@Override

**public** **boolean** **shouldOverrideUrlLoading**(**WebView** view, **String** url) {

**return** **false**;

}

});

方法二：

Webview先加载，然后返回true，webview不再加载

**public** **boolean** **shouldOverrideUrlLoading**(**WebView** view, **String** url) {

view.loadUrl(url);

**return** **true**;

}

#### shouldOverrideUrlLoading用途

由于每次超链接在加载前都会先走shouldOverrideUrlLoading回调，所以我们如果想拦截某个URL，将其转换成其它URL可以在这里做。 比如，我们拦截所有包含“blog.csdn.net”的地址，将其替换成”www.baidu.com”：

public boolean shouldOverrideUrlLoading(WebView view, String url) {

if (url.contains("blog.csdn.net")){

view.loadUrl("http://www.baidu.com");

}else {

view.loadUrl(url);

}

return true;

}

这里需要非常注意的是：如果我们在shouldOverrideUrlLoading中return true,就表示告诉系统我们已经拦截了URL并做处理，不需要再触发系统默认的行为（）在WebView中加载URL；所以对于其它URL我们需要在else里重新调用view.loadUrl(url)来加载；不然WebView将会白屏，因为这个URL根本就没有加载进WebView，在shouldOverrideUrlLoading这就被我们拦截掉了。

那么问题来了，在我们return true了以后，WebView还会请求网络吗？我们来抓下请求，从请求中可以看到，我们虽然拦截了”http://blog.csdn.net/harvic880925“但是仍然还是会请求网络的。只是请求以后结果并没有通过WebView加载。

那问题来了，如果我们return false呢，如果我们return false的话，是不需要else语句的，因为系统默认会加载这个URL，所以上面的语句与下面的意义相等：

public boolean shouldOverrideUrlLoading(WebView view, String url) {

if (url.contains("blog.csdn.net")){

view.loadUrl("http://www.baidu.com");

}

return false;

}

所以相对而言，我们使用return false好像更方便，只需要对需要拦截的URL进行拦截，拦截以后，让WebView处理默认操作即可。 所以结论来了：在利用shouldOverrideUrlLoading来拦截URL时，如果return true，则会屏蔽系统默认的显示URL结果的行为，不需要处理的URL也需要调用loadUrl()来加载进WebVIew，不然就会出现白屏；如果return false，则系统默认的加载URL行为是不会被屏蔽的，所以一般建议大家return false，我们只关心我们关心的拦截内容，对于不拦截的内容，让系统自己来处理即可。

### WVC之onReceivedError

加载错误的时候会产生这个回调，在其中可做错误处理，比如我们可以加载一个错误提示页面

public void onReceivedError(WebView view, int errorCode,String description, String failingUrl)

* WebView view:当前的WebView实例
* int errorCode：错误码
* String description：错误描述
* String failingUrl：当前出错的URL

我们可以先写一个错误提示的本地页面：（error.html），出现错误就加载它

mWebView.setWebViewClient(new WebViewClient(){

@Override

public void onReceivedError(WebView view, int errorCode, String description, String failingUrl) {

super.onReceivedError(view, errorCode, description, failingUrl);

mWebView.loadUrl("file:///android\_asset/error.html");

}

});

### WVC之onReceivedSslError

我们知道HTTPS协议是通过SSL来通信的，所以当使用HTTPS通信的网址（以https://开头的网站）出现错误时，就会通过onReceivedSslError回调通知过来，它的函数声明为：

public void onReceivedSslError(WebView view, SslErrorHandler handler,SslError error)

* WebView view:当前的WebView实例
* SslErrorHandler handler：当前处理错误的Handler，它只有两个函数SslErrorHandler.proceed()和SslErrorHandler.cancel()，SslErrorHandler.proceed()表示忽略错误继续加载，SslErrorHandler.cancel()表示取消加载。在onReceivedSslError的默认实现中是使用的SslErrorHandler.cancel()来取消加载，所以一旦出来SSL错误，HTTPS网站就会被取消加载了，如果想忽略错误继续加载就只有重写onReceivedSslError，并在其中调用SslErrorHandler.proceed()
* SslError error：当前的的错误对象，SslError包含了当前SSL错误的基本所有信息

#### 默认加载SSL出错的网站——出现空白页面

我们先举个例子来看下默认情况下加载SSL有错的网站，WebView的表现是怎样的：(12306是通过Https协议来传输的，但是它的SSL证书是有问题的，所以我们就以12306网站为例)

mWebView.setWebViewClient(new WebViewClient(){

@Override

public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {

super.onReceivedSslError(view, handler, error);

Log.e(TAG,"sslError:"+error.toString());

}

});

错误日志如下：

08-11 18:08:41.079: E/tag(10037): sslError:primary error: 3 certificate: Issued to: CN=kyfw.12306.cn,OU=Áï[¢b7g

R¡N-\_Ã,O=Sinorail Certification Authority,C=CN;

08-11 18:08:41.079: E/tag(10037): Issued by: CN=SRCA,O=Sinorail Certification Authority,C=CN;

08-11 18:08:41.079: E/tag(10037): on URL: https://www.12306.cn/

#### 使用SslErrorHandler.proceed()来继续加载

mWebView.setWebViewClient(new WebViewClient(){

@Override

public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {

//super.onReceivedSslError(view, handler, error); // 一定要注释掉！

handler.proceed();

Log.e(TAG,"sslError:"+error.toString());

}

});

这里做了两个改变：

第一：注释掉super.onReceivedSslError(view, handler, error);，取消系统的默认行为，我们看源码，可以发现在WebViewClient中onReceivedSslError的默认实现是这样的：

public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {

handler.cancel();

}

所以默认是取消继续加载的，所以我们必须注释掉super.onReceivedSslError(view, handler, error)来取消这个默认行为！

第二：调用handler.proceed();来忽略错误继续加载页面。

#### 在SSL发生错误时，onReceivedError会被回调吗？——不会

大家可能还有一个疑问：当SSL发生错误时，我们说会回调onReceivedSslError，我们前面还说了一个出错时会回调的函数：onReceivedError，那么问题来了，当出现SSL错误时onReceivedError会被回调吗？ 答案是不会的

#### 所以对于onReceivedSslError结论来了

当出现SSL错误时，WebView默认是取消加载当前页面，只有去掉onReceivedSslError的默认操作，然后添加SslErrorHandler.proceed()才能继续加载出错页面，当HTTPS传输出现SSL错误时，错误会只通过onReceivedSslError回调传过来，并不会触发onReceivedError回调。

### WVC之shouldInterceptRequest

在每一次请求资源时，都会通过这个函数来回调，比如超链接、JS文件、CSS文件、图片等，也就是说浏览器中每一次请求资源时，都会回调回来，无论任何资源！但是必须注意的是shouldInterceptRequest函数是在非UI线程中执行的，在其中不能直接做UI操作，如果需要做UI操作，则需要利用Handler来实现，该函数声明如下：

public WebResourceResponse shouldInterceptRequest(WebView view, String url) {

return null;

}

该函数会在请求资源前调用，我们可以通过返回WebResourceResponse的处理结果来让WebView直接使用我们的处理结果。如果我们不想处理，则直接返回null，系统会继续加载该资源。利用这个特性，我们可以解决一个需求：假如网页中需要加载本地的图片，我们就可以通过拦截shouldInterceptRequest，并返回结果即可

### WVC之其余函数

/\*\*

\* 在加载页面资源时会调用，每一个资源（比如图片）的加载都会调用一次

\*/

**public** **void** onLoadResource(WebView view, String url)

/\*\*

\* (WebView发生改变时调用)

\* 可以参考http://www.it1352.com/191180.html的用法

\*/

**public** **void** onScaleChanged(WebView view, **float** oldScale, **float** newScale)

/\*\*

\* 重写此方法才能够处理在浏览器中的按键事件。

\* 是否让主程序同步处理Key Event事件，如过滤菜单快捷键的Key Event事件。

\* 如果返回true，WebView不会处理Key Event，

\* 如果返回false，Key Event总是由WebView处理。默认：false

\*/

**public** **boolean** shouldOverrideKeyEvent(WebView view, KeyEvent event)

/\*\*

\* 是否重发POST请求数据，默认不重发。

\*/

onFormResubmission(WebView view, Message dontResend, Message resend)

/\*\*

\* 更新访问历史

\*/

doUpdateVisitedHistory(WebView view, String url, **boolean** isReload)

/\*\*

\* 通知主程序输入事件不是由WebView调用。是否让主程序处理WebView未处理的Input Event。

\* 除了系统按键，WebView总是消耗掉输入事件或shouldOverrideKeyEvent返回true。

\* 该方法由event 分发异步调用。注意：如果事件为MotionEvent，则事件的生命周期只存在方法调用过程中，

\* 如果WebViewClient想要使用这个Event，则需要复制Event对象。

\*/

onUnhandledInputEvent(WebView view, InputEvent event)

/\*\*

\* 通知主程序执行了自动登录请求。

\*/

onReceivedLoginRequest(WebView view, String realm, String account, String args)

/\*\*

\* 通知主程序：WebView接收HTTP认证请求，主程序可以使用HttpAuthHandler为请求设置WebView响应。默认取消请求。

\*/

onReceivedHttpAuthRequest(WebView view, HttpAuthHandler handler, String host, String realm)

/\*\*

\* 通知主程序处理SSL客户端认证请求。如果需要提供密钥，主程序负责显示UI界面。

\* 有三个响应方法：proceed(), cancel() 和 ignore()。

\* 如果调用proceed()和cancel()，webview将会记住response，

\* 对相同的host和port地址不再调用onReceivedClientCertRequest方法。

\* 如果调用ignore()方法，webview则不会记住response。该方法在UI线程中执行，

\* 在回调期间，连接被挂起。默认cancel()，即无客户端认证

\*/

onReceivedClientCertRequest(WebView view, ClientCertRequest request)

## WebChromeClient----影响浏览器的事件的回调监听

### 概述

#### 与WebViewClient的区别

很多同学一看到这里有Chrome，立马就会想到google 的Chrome浏览器；这里并不是指Chrome浏览器的意思，而是泛指浏览器，WebView的内部实现并不是完全使用Chrome的内核，而是部分使用Chome内核，其它都是与Chrome不相同的； 我们再来对比下WebViewClient:

* WebViewClient:在影响View的事件到来时，会通过WebViewClient中的方法回调通知用户
* WebChromeClient：当影响浏览器的事件到来时，就会通过WebChromeClient中的方法回调通知用法。

通过上面的对比，我们发现WebViewClient和WebChromeClient都是针对不同事件的回调，而google将这些回调进行分类集合，就产生了WebViewClient、WebChromeClient这两个大类，其中管理着针对不同类型的回调而已。

#### 为什么叫WebChromeClient？

因为WebChromeClient中集合了影响浏览器的事件到来时的回调方法，所以这里更需要突出浏览器的概念，而Chrome则是google自家的浏览器名称，所以使用WebChromeClient来做为名称吧，纯属臆想……

#### WebChromeClient的常用函数

我们先来看一下，在WebChromeClient中我们将要讲解的函数，其实WebChromeClient里的函数是非常多的，可以监控到浏览器的很多方面，这里我们就不再一个个来讲了，只讲解常用的几个函数，而且随着ReactNative和Hybird的普及，WebView的使用场景会越来越少，指不定哪一天就被废弃了

// 当网页调用alert()来弹出alert弹出框前回调，用以拦截alert()函数

**public** **boolean** onJsAlert(WebView view, String url, String message,JsResult result)

// 当网页调用confirm()来弹出confirm弹出框前回调，用以拦截confirm()函数

**public** **boolean** onJsConfirm(WebView view, String url, String message,JsResult result)

//当网页调用prompt()来弹出prompt弹出框前回调，用以拦截prompt()函数

**public** **boolean** onJsPrompt(WebView view, String url, String message,String defaultValue, JsPromptResult result)

// 打印 console 信息

**public** **boolean** onConsoleMessage(ConsoleMessage consoleMessage)

// 通知程序当前页面加载进度

**public** **void** onProgressChanged(WebView view, **int** newProgress)

### WCC之onJsAlert、onJsConfirm、onJsPrompt

即使启用了JavaScript但是在网页中的confrim()、alert()、prompt()却是没有效果！ 这是因为我们需要设置WebChromClient! 在程序中，我们只需要加上mWebView.setWebChromeClient(new WebChromeClient());就可以实现confrim()、alert()、prompt()的弹出效果了，但是显示的很丑，title里显示为“：来自file:////”。。。。。。，需要重写这些方法

**public** **class** **MyWebChromeClient** **extends** **WebChromeClient** {

**private** **Context** mContext;

**public** **MyWebChromeClient**(**Context** context) {

**this**.mContext = context;

}

/\*\*\* 覆盖默认的window.alert展示界面\*/

@Override

**public** **boolean** **onJsAlert**(**WebView** view, **String** url, **String** message, **JsResult** result) {

**final** **AlertDialog**.**Builder** builder = **new** **AlertDialog**.Builder(view.getContext());

builder.setTitle("对话框")

.setMessage(message)

.setPositiveButton("确定", **null**);

// 不需要绑定按键事件

// 屏蔽abcd之类的按键

builder.setOnKeyListener(**new** OnKeyListener() {

**public** **boolean** **onKey**(DialogInterface dialog, **int** keyCode,**KeyEvent** event) {

**Log**.*v*("onJsAlert", "keyCode==" + keyCode + "event="+ event);

**return** **true**;

}

});

builder.setCancelable(**false**);// 禁止响应按back键的事件

**AlertDialog** dialog = builder.create();

dialog.show();

result.confirm();// 因为没有绑定事件，需要强行confirm,否则页面会变黑显示不了内容。

/\*\*\* reture ture表示告诉WebView我们已经拦截了alert()函数，不需要再弹出网页中的alert弹出框了，

如果我们return false,那么WebView就会认为我们没有拦截alert()函数，会继续弹出alert对话框。\*/

**return** **true**;

//return super.onJsAlert(view, url, message, result);//默认返回false

}

/\*\*\* 覆盖默认的window.confirm展示界面 \*/

@Override

**public** **boolean** **onJsConfirm**(**WebView** view, **String** url, **String** message, **final** **JsResult** result) {

**final** **AlertDialog**.**Builder** builder = **new** **AlertDialog**.Builder(view.getContext());

builder.setTitle("警告")

.setMessage(message)

.setPositiveButton("确定",**new** OnClickListener() {

**public** **void** **onClick**(DialogInterface dialog,**int** which) {

result.confirm();

}

})

.setNeutralButton("取消", **new** OnClickListener() {

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

result.cancel();

}

});

builder.setOnCancelListener(**new** OnCancelListener() {

@Override

**public** **void** **onCancel**(DialogInterface dialog) {

result.cancel();

}

});

builder.setOnKeyListener(**new** OnKeyListener() {

@Override

**public** **boolean** **onKey**(DialogInterface dialog, **int** keyCode,**KeyEvent** event) {

**Log**.*v*("onJsConfirm", "keyCode==" + keyCode + "event="+ event);

**return** **true**;

}

});

// 不 禁止响应按back键的事件

**AlertDialog** dialog = builder.create();

dialog.show();

**return** **true**;

}

/\*\*\* 覆盖默认的window.prompt展示界面\*/

@Override

**public** **boolean** **onJsPrompt**(**WebView** view, **String** url, **String** message, **String** defaultValue, **final** **JsPromptResult** result){

**final** **AlertDialog**.**Builder** builder = **new** **AlertDialog**.Builder(view.getContext());

builder.setTitle("对话框").setMessage(message);

**final** **EditText** et = **new** EditText(view.getContext());

et.setSingleLine();

et.setText(defaultValue);

builder.setView(et)

.setPositiveButton("确定", **new** OnClickListener() {

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

result.confirm(et.getText().toString());// 有参，提交数据

}

})

.setNeutralButton("取消", **new** OnClickListener() {

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

result.cancel();

}

});

// 不 屏蔽keycode等于84之类的按键，避免按键后导致对话框消息而页面无法再弹出对话框的问题

// 不 禁止响应按back键的事件

**AlertDialog** dialog = builder.create();

dialog.show();

**return** **true**;

}

}

总结一下：

* 如果需要使网页中的confrim()、alert()、prompt()函数生效，需要设置WebChromeClient！
* 在使用他们来拦截alert对话框时，如果不需要再弹出alert对话框，一定要return true;在return false以后，会依然调用系统的默认处理来弹出对话框的
* 如果我们return true,则需要在处理完成以后调用JsResult.confirm()或者JsResult.cancel()来告诉WebView我们点中哪个按钮来取消程序对话框。否则再次点击按钮将会失败

### WCC之onConsoleMessage

当html中调用console相关输出的时候，就会通过onConsoleMessage进行通知

public boolean onConsoleMessage(ConsoleMessage consoleMessage)

* ConsoleMessage consoleMessage:保存着当前消息的类型和消息内容
* 返回值：如果返回true时，就表示拦截了console输出，系统就不再通过console输出出来了，如果返回false则表示没有拦截console输出，调用系统默认处理。

我们来看下正常情况下，console输出的内容及用法 ：不用特意去设置，默认就是输出了，当然也可以重写自定义。

html内容：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<h1 id="h">欢迎光临启舰的blog</h1>

<button onclick="log()">log</button>

</body>

<script type="text/javascript">

function log(){

console.log("console.log");

console.warn("warnning");

console.error("error");

}

</script>

</html>

结果：

08-11 19:29:59.119: I/chromium(12799): [INFO:CONSOLE(14)] "warnning", source: file:///android\_asset/cons.html (14)

08-11 19:29:59.119: I/chromium(12799): [INFO:CONSOLE(15)] "error", source: file:///android\_asset/cons.html (15)

08-11 19:29:59.319: I/chromium(12799): [INFO:CONSOLE(13)] "console.log", source: file:///android\_asset/cons.html (13)

### WCC之onProgressChanged

表示当前页面的加载速度，函数声明如下：public void onProgressChanged(WebView view, int newProgress)

WebView view:当前WebView实例

int newProgress：当前的加载进度，值从0到100

@Override

**public** **void** **onProgressChanged**(**WebView** view, **int** newProgress) {

((**Activity**) mContext).setTitle("Loading...");

//((Activity) mContext).setProgress(newProgress);//不知道这是干什么用的

**String** titleString = mContext.getClass().getSimpleName();

**if** (newProgress >= 80) {

((**Activity**) mContext).setTitle(titleString);

}

}

大家一定要注意，底层实现时，是利用handler来定时轮循当前进度的，每隔一定时间查询一次，所以每次拿到的进度数据是不一样的，也就是说如果页面较简单，可能会直接返回100，而跳过中间的各个数据。也就是说，除了100，其它任何一个数值不是一定返回的，所以大家如果要用到进度除了数值100可以用等号来判断，其它一定要用大于号或小于号，如果用了等号，可能永远也不会执行到！

### WCC之其它函数

/\*

\* 通知页面标题变化

\*/

nReceivedTitle(WebView view, String title)

/\*

\* 通知当前页面网站新图标

\*/

onReceivedIcon(WebView view, Bitmap icon)

/\*

\* 通知主程序图标按钮URL

\*/

onReceivedTouchIconUrl(WebView view, String url, **boolean** precomposed)

/\*

\* 通知主程序当前页面将要显示指定方向的View，该方法用来全屏播放视频。

\*/

**public** **interface** CustomViewCallback {

// 通知当前页面自定义的View被关闭

**public** **void** **onCustomViewHidden**();

}

onShowCustomView(View view, CustomViewCallback callback)

/\*

\* 与onShowCustomView对应，通知主程序当前页面将要关闭Custom View

\*/

onHideCustomView()

/\*\*

\* 请求主程序创建一个新的Window，如果主程序接收请求，返回true并创建一个新的WebView来装载Window，然后添加到View中，发送带有创建的WebView作为参数的resultMsg的给Target。如果主程序拒绝接收请求，则方法返回false。默认不做任何处理，返回false

\*/

onCreateWindow(WebView view, **boolean** isDialog, **boolean** isUserGesture, Message resultMsg)

/\*

\* 显示当前WebView，为当前WebView获取焦点。

\*/

onRequestFocus(WebView view)

/\*

\* 通知主程序关闭WebView，并从View中移除，WebCore停止任何的进行中的加载和JS功能。

\*/

onCloseWindow(WebView window)

/\*\*

\* 告诉客户端显示离开当前页面的导航提示框。如果返回true，由客户端处理确认提示框，调用合适的JsResult方法。如果返回false，则返回默认值true给javascript接受离开当前页面的导航。默认：false。JsResult设置false，当前页面取消导航提示，否则离开当前页面。

\*/

onJsBeforeUnload(WebView view, String url, String message, JsResult result)

/\*\*

\*通知主程序web内容尝试使用定位API，但是没有相关的权限。主程序需要调用调用指定的定位权限申请的回调。更多说明查看GeolocationPermissions相关API。

\*/

onGeolocationPermissionsShowPrompt(String origin,GeolocationPermissions.Callback callback)

/\*

\* 通知程序有定位权限请求。如果onGeolocationPermissionsShowPrompt权限申请操作被取消，则隐藏相关的UI界面。

\*/

onGeolocationPermissionsHidePrompt()

/\*\*

\*通知主程序web内容尝试申请指定资源的权限（权限没有授权或已拒绝），主程序必须调用PermissionRequest#grant(String[])或PermissionRequest#deny()。如果没有覆写该方法，默认拒绝。

\*/

onPermissionRequest(PermissionRequest request)

/\*\*

\* 通知主程序相关权限被取消。任何相关UI都应该隐藏掉。

\*/

onPermissionRequestCanceled(PermissionRequest request)

/\*\*

\* 通知主程序 执行的Js操作超时。客户端决定是否中断JavaScript继续执行。如果客户端返回true，JavaScript中断执行。如果客户端返回false，则执行继续。注意：如果继续执行，重置JavaScript超时计时器。如果Js下一次检查点仍没有结束，则再次提示。

\*/

onJsTimeout()

/\*\*

\*当停止播放，Video显示为一张图片。默认图片可以通过HTML的Video的poster属性标签来指定。如果poster属性不存在，则使用默认的poster。该方法允许ChromeClient提供默认图片。

\*/

getDefaultVideoPoster()

/\*\*

\* 当用户重放视频，在渲染第一帧前需要花费时间去缓冲足够的数据。在缓冲期间，ChromeClient可以提供一个显示的View。如：可以显示一个加载动画。

\*/

getVideoLoadingProgressView()

/\*\*

\* 获取访问历史Item，用于链接颜色。

\*/

getVisitedHistory(ValueCallback callback)

/\*\*

\* 通知客户端显示文件选择器。用来处理file类型的HTML标签，响应用户点击选择文件的按钮操作。调用filePathCallback.onReceiveValue(null)并返回true取消请求操作。

\* FileChooserParams参数的枚举列表：

MODE\_OPEN 打开

MODE\_OPEN\_MULTIPLE 选中多个文件打开

MODE\_OPEN\_FOLDER 打开文件夹（暂不支持）

MODE\_SAVE 保存

\*/

onShowFileChooser(WebView webView, ValueCallback filePathCallback,FileChooserParams fileChooserParams)

/\*\*

\* 解析文件选择Activity返回的结果。需要和createIntent一起使用。

\*/

parseResult(**int** resultCode, Intent data)

/\*\*

\* 创建Intent对象来启动文件选择器。Intent支持可访问的简单类型文件资源。不支持高级文件资源如live media capture媒体快照。如果需要访问这些资源或其他高级文件类型资源可以自己创建Intent对象。

\*/

createIntent()

/\*\*

\* 返回文件选择模式

\*/

getMode()

/\*\*

\* 返回可访问MIME类型数组，如audio/\*，如果没有指定可访问类型，数组返回为null

\*/

getAcceptTypes()

/\*\*

\* 返回优先的媒体快照类型值如Camera、Microphone。true：允许快照。false，禁止快照。使用getAcceptTypes方法确定合适的capture设备。

\*/

isCaptureEnabled()

/\*\*

\* 返回文件选择器的标题。如果为null，使用默认名称。

\*/

getTitle()

/\*\*

\*指定默认选中的文件名或为null

\*/

getFilenameHint()

## LoadData()与loadDataWithBaseURL()

通过loadUrl()来加载本地页面和在线地址的方式，这里的两个方法LoadData()与loadDataWithBaseURL()，它们不是用来加载整个页面文件的，而是用来加载一段代码片的。这两个函数平时用的比较少，这里只讲基本用法，就不再深入了。

声明方式为：

public void loadData(String data, String mimeType, String encoding)

public void loadDataWithBaseURL(String baseUrl, String data,String mimeType, String encoding, String historyUrl)

这里每个参数的具体意义后面再详细讲解，这里涉及到MIME类型，简单来讲，MIME的类型格式是跟文件的后缀相关联的，比如下面的几个关联方式，这里仅涉及到网页和图片的类型，还有其它的就不再一一列举

文件后缀 MIME类型 文件后缀 MIME类型

html text/html bmp image/bmp

jpg image/jpeg png image/png

### loadData

#### 示例1：简单使用

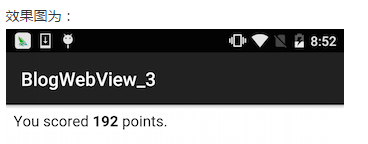
我们先来看看loadData是如何使用的，它的声明如下：public void loadData(String data, String mimeType, String encoding)

* String data：代码片段内容
* String mimeType：代码片段所对应的MIME类型，如果传null，则默认为text/html
* String encoding：代码片段的编码方式

我们先举个例子来看下：

String summary = "<html><body>You scored <b>192</b> points.</body></html>";

mWebView.loadData(summary, "text/html", "utf-8");



#### 示例2：loadData的正确使用方式

在使用loadData时，在数据里面不能出现英文字符：’#’, ‘%’, ‘\’ , ‘?’ 这四个字符，如果有的话可以用 %23, %25, %27, %3f，这些字符来替换，在平时测试时，你的数据时，你的数据里含有这些字符，但不会出问题，当出问题时，你可以替换下。 直接使用这四个字符会造成的问题如下：

* %：会报找不到页面错误，页面全是乱码。
* #，会让你的goBack失效，但canGoBAck是可以使用的。于是就会产生返回按钮生效，但不能返回的情况。
* \ 和? 我在转换时，会报错，因为它会把\当作转义符来使用，如果用两级转义，也不生效，我是对它无语了。

我们虽然可以把代码中的字符逐个读取，然后给转换掉，但运行速度上会大打折扣，因为webview.loadData()是运行在主线程的，所以如果你的代码段很长，那就会卡死主线程！所以Android给我们提供了一个专门用来转码的函数：URLEncoder.encode(String s, String charsetName) ,它能将冲突的字符进行转义，然后再传给webview，这样webview在加载时就不会有冲突了,encode函数的声明如下：public static String encode(String s, String charsetName);

* String s:代码段
* String charsetName：编码类型

或者URLEncoder.encode(String s)，默认编码utf-8;

下面我们来看使用这个encode函数的例子：

try {

String summary = "<html><body>You scored <b>192</b> points.</body></html>";

mWebView.loadData(URLEncoder.encode(summary, "utf-8"), "text/html", "utf-8");

} catch (Exception e) {

Log.e("qijian", e.getMessage());

}

data中，有人会遇到中文乱码问题，解决办法：参数传”utf-8”，页面的编码格式也必须是utf-8，这样编码统一就不会乱了。别的编码我也没有试过。注意：

* loadData()应该是不能加载图片的，反正我是加载不出来，加载图片的内容我们后面会使用loadDataWithBaseURL来实现。
* 为了防止字符冲突，在传递loadData的数据时，必须使用URLEncoder.encode()函数来转义
* 页面的编码格式必须与代码中传参的编码格式一致，不然会导致乱码

### loadDataWithBaseURL

相比loadData，这个函数更常用，因为loadData能实现的功能，它都能实现，而且也不会出现字符冲突。其函数声明如下：public void loadDataWithBaseURL(String baseUrl, String data,String mimeType, String encoding, String historyUrl)

* String baseUrl：基准URL，不需要可以传null，它的意思是，如果data中的url是相对地址，则就会加上基准url来拼接出完整的地址，比如baseUrl是http://img6.ph.126.net，data中有个Img标签，它的内容是：<img src='hBiG96B8egigBULxUWcOpA==/109212290980771276.jpg'>，很明显src的地址不是本地地址也不是在线地址，那它就是一个相对地址，所以加上baseUrl以后才是它的完整地址：http://img6.ph.126.net/hBiG96B8egigBULxUWcOpA==/109212290980771276.jpg
* String mimeType:MIME类型
* String encoding：编码方式
* String historyUrl：当前的历史记录所要存储的值。如果不需要可以传Null，loadDataWithBaseURL它本身并不会向历史记录中存储数据，要想实现历史记录，需要我们自己来实现；有关历史记录的实现方式是比较复杂的，历史记录是以Key/value的方式存储在一个historyList里的，当前进后退时，会用Key来取出对应的value值来加载进webview中。而Key就是这里的baseUrl,Value就是这里的historyUrl；history所指向的必须是一个页面，并且页面存在于SD卡中或程序中（assets）；

下面我们举个例子来看下baseUrl的概念：

#### 示例1：baseUrl的用法

String baseURL = "http://img6.ph.126.net";

String data = "嘎嘎发 <img src='hBiG96B8egigBULxUWcOpA==/109212290980771276.jpg'>";

mWebView.loadDataWithBaseURL(baseURL, data, "text/html", "utf-8", null);

#### 示例2：三种地址加载情况

我们再进阶一下，看如果网页地址是绝对地址，本地文件地址也是绝对地址，而另外一个是相对地址的情况下，看baseUrl是如何操作的

String baseURL = "http://img6.ph.126.net";

String data = "绝对网页地址:" +

"<img src='http://img3.3lian.com/2013/v8/72/d/61.jpg'>" +

"本地地址:" +

"<img src='file:///android\_asset/s07.jpg'>" +

"相对地址:" +

"<img src='hBiG96B8egigBULxUWcOpA==/109212290980771276.jpg'>";

mWebView.loadDataWithBaseURL(baseURL, data, "text/html", "utf-8", null);

在代码中有三种地址：第一个是绝对的http地址，第二个是通过file指定的本地地址，对于这两类的绝对地址，baseUrl是不起作用的，而对于第三个相对地址，是会启用baseUrl的来拼接完整地址的。

### 总结

* 面这两种方法，我建议使用后者，虽然loadData的历史记录不需要我们自己来实现，但在使用时，这就两个加载上后者比前者快一到两倍。
* 另外loadData不能加载图片，而loadDataWithBaseURL是可以加载图片的

## 缓存的使用

当我们加载Html时候，会在我们data/应用package下生成database与cache两个文件夹:

WebView中存在着两种缓存：网页数据缓存（存储打开过的页面及资源）、H5缓存（即AppCache）。

### 缓存构成

/data/data/package\_name/cache/

/data/data/package\_name/database/webview.db

/data/data/package\_name/database/webviewCache.db

我们请求的Url记录是保存在webviewCache.db里，而url的内容是保存在webviewCache文件夹下.

我们浏览过的网页url以及网页文件(css、图片、js等)保存到数据库表中

### 缓存模式

**LOAD\_CACHE\_ONLY:**

不使用网络，只读取本地缓存数据

**LOAD\_NO\_CACHE**

不使用缓存，只从网络获取数据.  
**LOAD\_DEFAULT**

根据cache-control决定是否从网络上取数据。m.taobao.com的cache-control为no-cache，在模式LOAD\_DEFAULT下，无论如何都会从网络上取数据，如果没有网络，就会出现错误页面  
**LOAD\_CACHE\_ELSE\_NETWORK**

只要本地有，无论是否过期，或者no-cache，都使用缓存中的数据。无论是否有网络，只要本地有缓存，都使用缓存。本地没有缓存时才从网络上获取。

**总结：**

根据以上两种模式，建议缓存策略为，判断是否有网络，有的话，使用LOAD\_DEFAULT，无网络时，使用LOAD\_CACHE\_ELSE\_NETWORK。

//android 4.0-, API 14- 写法 /data/data/packagename/files/webcache

//String cacheDirPath = context.getFilesDir().getAbsolutePath()+"/webcache";

//android 4.0+, API 14+ 写法

**String** cacheDirPath = context.getApplicationContext().getDir("cache", **Context**.**MODE\_PRIVATE**).getPath();

**String** geolocationDatabasePath = context.getApplicationContext().getDir("database", **Context**.**MODE\_PRIVATE**).getPath();

webSettings.setGeolocationEnabled(**true**);

webSettings.setAllowFileAccess(**true**);

webSettings.setGeolocationDatabasePath(geolocationDatabasePath);

**WebSettings** webSettings = mWebView.getSettings();

webSettings.**setRenderPriority**(RenderPriority.**HIGH**);

webSettings.setDomStorageEnabled(**true**);//// 开启 DOM storage API 功能

webSettings.setDatabaseEnabled(**true**);//开启 database storage API 功能

webSettings.setAppCacheEnabled(**true**);//开启 Application Caches 功能

webSettings.**setAppCacheMaxSize**(1024 \* 1024 \* 8);

**int** nettag = getNetInfo();

**if** (nettag == -1) {

webSettings.setCacheMode(**WebSettings**.**LOAD\_CACHE\_ELSE\_NETWORK**);

}**else** {

webSettings.setCacheMode(**WebSettings**.**LOAD\_DEFAULT**);

}

webSettings.setAppCachePath(cacheDirPath);

webSettings.**setDatabasePath**(cacheDirPath);

### 清理WebView缓存

**public** **void** **clearWebViewCache**(){

**try** { //清理Webview缓存数据库

deleteDatabase("webview.db");

deleteDatabase("webviewCache.db");

} **catch** (**Exception** e) {

e.printStackTrace();

}

**File** appCacheDir = **new** File(cacheDirPath );

**File** webviewCacheDir = **new** File(cacheDirPath );

**if**(webviewCacheDir.exists()){

deleteFile(webviewCacheDir); //删除webview 缓存目录

}

**if**(appCacheDir.exists()){

deleteFile(appCacheDir); //删除app缓存目录

}

}

/\*\*

\* 递归删除 文件/文件夹

\*/

**public** **void** **deleteFile**(**File** file) {

**if** (file.exists()) {

**if** (file.isFile()) {

file.delete();

} **else** **if** (file.isDirectory()) {

**File** files[] = file.listFiles();

**for** (**int** i = 0; i < files.length; i++) {

deleteFile(files[i]);

}

}

file.delete();

} **else** {

}

}

## 支持使用localStorage

webSettings.setDomStorageEnabled(**true**);//最重要的方法，一定要设置，这就是出不来的主要原因

webView.getSettings().setDatabaseEnabled(**true**);

webView.getSettings().**setDatabasePath**(cacheDirPath);

## 识别js中是否需要跳到新网页，来确定是否需要新开Activity

**public** **class** **MyWebViewClient** **extends** **WebViewClient** {

**private** **Context** mContext;

**private** **Class**<?> mClass;

**public** **MyWebViewClient**(**Context** context, **Class**<?> mClass) {

**this**.mContext = context;

**this**.mClass = mClass;

}

@Override

**public** **void** **onPageFinished**(**WebView** view, **String** url) {

**// 页面加载完成后加载下面的javascript，修改页面中所有用target="\_blank"标记的url（在url前加标记为“newtab”）**

view.loadUrl("javascript: var allLinks = document.getElementsByTagName('a'); if (allLinks) {var i;for (i=0; i<allLinks.length; i++) {var link = allLinks[i];var target = link.getAttribute('target'); if (target && target == '\_blank') {link.setAttribute('target','\_self');link.href = 'newtab:'+link.href;}}}");

}

@Override

**public** **boolean** **shouldOverrideUrlLoading**(**WebView** view, **String** url) {

**if** (url.startsWith("newtab:")) {

**String** realUrl = url.substring(7,url.length());

**Intent** intent = **new** Intent(mContext, mClass);

intent.putExtra("url", realUrl);

mContext.startActivity(intent);

} **else** {

view.loadUrl(url);

}

**return** **true**;

}

}

## 网页中链接中打开app

注意：一定不能覆盖shouldOverrideUrlLoading方法，否则打不开！！！！！

一定不能有这个设置，否则打不开mWebView.setWebViewClient(new WebViewClient());

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Insert title here</title>

</head>

<body>

<a href="m://my.com?name=cgy">打开app</a><br/>

</body>

</html>

<activity

android:name=".MainActivity"

android:label="@string/app\_name" >

<intent-filter>

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

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

</intent-filter>

<intent-filter>

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

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

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

<data android:host="my.com" android:scheme="m" />

</intent-filter>

</activity>

**Uri** uri = getIntent().getData();

**if** (uri != **null**) {

**Log**.e("tag", uri.getQueryParameter("name"));

}

getIntent().getScheme(); //获得Scheme名称

getIntent().getDataString(); //获得Uri全部路径

getIntent().getHost(); //获得host

## 清除所有cookie

*/\*\*  
 \* 清除webView cookie  
 \*/*public static void clearCookie(Context context) {  
 CookieSyncManager.*createInstance*(context);  
 CookieManager.*getInstance*().removeAllCookie();  
}

## 打开淘宝天猫出现问题解决



重写

@Override    
 public boolean shouldOverrideUrlLoading(WebView view, String url) {     
 return super.shouldOverrideUrlLoading(view, url);     
     }     
 }

## 重定向问题解决

WebView有一个方法getHitTestResult();它的返回值是WebView.HitTestResult。HitTestResult是WebView的一个内部类，那么它有什么作用呢？我们在点击网页的链接时，都会有一个TYPE(类型)，根据HitTestResult的TYPE(类型)返回一个”extra”。如果打开的是UNKNOWN\_TYPE，那么就返回null，所以可以通过返回值是否为null，来判断是否是重定向。

@Override

public boolean shouldOverrideUrlLoading(WebView view, String url) {

WebView.HitTestResult hitTestResult = view.getHitTestResult();

if (!TextUtils.isEmpty(url) && hitTestResult == null) {//hitTestResult==null时，是重定向

view.loadUrl(url);

return true;

}

return super.shouldOverrideUrlLoading(view, url);

}

## 地理位置定位问题解决

//启用数据库

webSettings.setDatabaseEnabled(true);

String dir = this.getApplicationContext().getDir("database", Context.MODE\_PRIVATE).getPath();

//启用地理定位

webSettings.setGeolocationEnabled(true);

//设置定位的数据库路径

webSettings.setGeolocationDatabasePath(dir);

//最重要的方法，一定要设置，这就是出不来的主要原因

webSettings.setDomStorageEnabled（true）

//配置权限（在WebChromeClient中实现）

public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {

callback.invoke(origin, true, false);

super.onGeolocationPermissionsShowPrompt(origin, callback);

}

//配置权限：

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION" />

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION" />

## 播放土豆优酷的视频

在android 4.x之后都不好使去除了flash的支持。所以无法支持播放土豆优酷等视频网站的视频，下面是解决办法。

**1. 修改AndroidManifest.xml**

在Application节点上增加

android:hardwareAccelerated="true"

当然你也可以在WebView所在的Activity节点上增加此属性

**2. 在WebView的布局XML上增加 android:layerType="hardware"**

<WebView

android:id="@+id/newsContent"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layerType="hardware" />

**3. 在Java代码中设置相应属性**

mWebView.requestFocus();

**WebSettings** ws = mWebView.getSettings();

ws.setJavaScriptEnabled(**true**);

ws.setAllowFileAccess(**true**);

ws.setDatabaseEnabled(**true**);

ws.setDomStorageEnabled(**true**);

ws.setSaveFormData(**false**);

ws.setAppCacheEnabled(**true**);

ws.setCacheMode(**WebSettings**.**LOAD\_DEFAULT**);

ws.setLoadWithOverviewMode(**false**);//一定要设置为false，不然有声音没图像

ws.setUseWideViewPort(**true**);

mWebView.setLongClickable(**false**);// js中不能长按

mWebView.setWebChromeClient(**new** MyWebChromeClient(context));//设置自定义WebChromeClient（对话框覆盖）

mWebView.setWebViewClient(**new** MyWebViewClient(context));

mWebView.loadUrl(tempUrl);

**4.Activity关闭后，视频无法停止的问题**

@Override

**protected** **void** **onResume**() {

**super**.onResume();

//恢复播放

mWebView.resumeTimers();

}

@Override

**protected** **void** **onPause**() {

**super**.onPause();

//暂停播放

mWebView.pauseTimers();

}

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

//一定要销毁，否则无法停止播放

mWebView.destroy();

}

**5.分享链接只能是通用链接**

需要分享的链接是通用的链接即，<iframe src="http://play.youku.com/xjlkdfjaklfjdsa" /> ，若直接使用embed的不会播放，还是那个原因，android不支持flash插件播放

## 返回按键，后退网页

### 普通的实现

如果用webview点链接看了很多页以后，如果不做任何处理，点击系统“Back”键，整个浏览器会调用finish()而结束自身，如果希望浏览的网页回退而不是退出浏览器，需要在当前Activity中处理并消费掉该Back事件。 覆盖Activity类的onKeyDown(int keyCoder,KeyEvent event)方法。

@Override

**public** **boolean** **onKeyDown**(**int** keyCode, **KeyEvent** event) {

**if** ((keyCode == **KeyEvent**.**KEYCODE\_BACK**) && mWebView.canGoBack()) {

mWebView.goBack();

**return** **true**;

}

**return** **super**.onKeyDown(keyCode, event);

}

### 因重定向无法正常goBack()解决方案

首先说下问题，初始页面为A，点击某个链接跳转到B（http://xxx.com.cn/）,B页面重定向到C页面(http://xxx.com.cn/website/index.html)，当调用webview.goBack()时，页面回退到B，然后接着会重定向回C页面。这样会导致两个问题:

* 无法回退到webview的初始页面A
* 无法正常退出Activity或者Fragment(只有还未加载完C时进行回退才能退出页面)

关于如何解决这个问题，我总结了如下三种方法，可以根据具体情况进行使用:

### 方法一：移除没有意义的重定向

首先需要和前端开发人员沟通，看重定向是否必要，如果跳转链接只是域名，然后默认重定向到域名/index.html,并没有特殊处理的话，那么这种重定向并没有意义.只要将网页中的连接，比如

<a href="http://xxx.com.cn"/> 直接替换为<a href="http:///xxx.com.cn/index.html"/> 即可解决该问题.

### 方法二：自己维护一个webview的历史栈

页面中的重定向是必须的，那么我们就需要自己维护一个webview的历史栈,根据自己的需求进行过滤跳转或者重新加载页面:

判断到当前为重定向后的链接，那么那么当回退的时候就需要忽略上一级的链接，不使用webview.goback(),移除重定向和重定向后的url,获取到初始页面链接后自己进行loadUrl()操作.

### 方法三：配合前端类似方法二

还有一种方法，和方法2类似,需要自己维护webview的历史栈，但是需要前端的配合，提供js函数获取网页是否进行重定向在webviewClient回调shouldoverloading()中过滤url时，若属于重定向的地址，则不加入栈中，回退时根据历史栈加载即可

### 方法二详解

首先定义一个历史栈

private ArrayList<String> loadHistoryUrls = new ArrayList<String>();

把初始页面Url加入

loadHistoryUrls.add(INITAL\_WEB\_URL);

然后加入加载的url:

public boolean shouldOverrideUrlLoading(WebView view,String url){

//将过滤到的url加入历史栈中

loadHistoryUrls.add(url);

return true;

}

最后在webview.goback()处理：

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

//判断是否可以返回操作

if (webView.canGoBack() && event.getKeyCode() == KeyEvent.KEYCODE\_BACK) {

//过滤是否为重定向后的链接

if(loadHistoryUrls.size()>0 && loadUrls.get(loadHistoryUrls.size()-1).contains("index.html"))

//移除加载栈中的最后两个链接

loadHistoryUrls.remove(loadHistoryUrls.get(loadHistoryUrls.size()-1));

loadHistoryUrls.remove(loadHistoryUrls.get(loadHistoryUrls.size()-1));

//加载重定向之前的页

webview.load(loadUrls.get(loadHistoryUrls.size()-1));

return true;

}

}

}

关于加载栈，后来发现webview本身也有对应的API:

//获取历史列表

WebBackForwardList mWebBackForwardList = webView.copyBackForwardList();

不过这个api可能受系统版本的影响或者不同手机系统进行了修改，所以解决该问题时，大家可以自己根据需求，自己维护加载的历史栈或者直接调用系统api.

### 总结

这里总结一下，若重定向非必要，采取方案一，最简单，修改量也非常小. 重定向必要，则使用方案二或者方案三.因为需要和前端人员交互，方案三所需要的沟通，开发，维护的成本要比方案二高出不少，但对于是否重定向的判断非常准确,若有多个重定向的情况,一次开发完成后不需要对代码再次改动. 方案二则需要写死需要过滤的url,若出现多个重定向，则会显得代码比较臃肿,每次都需要重新增加代码. 具体使用依据项目中的开发情况而定.

最后再补充一种通用的办法，但是需要后台的强大支持: 在webview进行加载时，将请求发送至服务器，然后由服务器进行分析处理，将处理后的结果返回给客户端进行显示. 并且可以由服务器对网页内容进行编码或者取出冗余，并结合cdn提升响应速度,这也是目前浏览器开发常用的一种策略.但是需要大量的数据收集，分析和处理，对于服务器的依赖比较严重,若开发进度较紧或者公司资源有限，可先参照以上办法进行解决.

最重还要讲的一点， 本篇文章主要是对于加载己方开发的H5中遇到问题的解决，至于第三方网站加载，这个是没有办法解决的. 包括微信上也一样，对于各种公众平台和第三方链接，是没有通用解决方案的, 所以他们在交互上进行了处理 ，在H5进行一次跳转就会在标题栏左上角出现关闭按钮. 毕竟用户是不知道快速连续点击两次返回才能正常返回首页的.

## 滚动事件监听

我们都知道监听滚动事件一般都是设置setOnScrollChangedListener，可惜的是 WebView并没有给我们提供这样的方法，但是我们可以重写WebView，覆盖里面的一个方法： protected void onScrollChanged(final int l, final int t, final int oldl,final int oldt){} 然后再对外提供一个回调接口，示例代码如下：

public class MyWebView extends WebView {

private OnScrollChangedCallback mOnScrollChangedCallback;

public MyWebView(Context context) {

super(context);

}

public MyWebView(Context context, AttributeSet attrs) {

super(context, attrs);

}

public MyWebView(Context context, AttributeSet attrs, int defStyleAttr) {

super(context, attrs, defStyleAttr);

}

@Override

protected void onScrollChanged(int l, int t, int oldl, int oldt) {

super.onScrollChanged(l, t, oldl, oldt);

if (mOnScrollChangedCallback != null) {

mOnScrollChangedCallback.onScroll(l,t,oldl,oldt);

}

}

public OnScrollChangedCallback getOnScrollChangedCallback() {

return mOnScrollChangedCallback;

}

public void setOnScrollChangedCallback(

final OnScrollChangedCallback onScrollChangedCallback) {

mOnScrollChangedCallback = onScrollChangedCallback;

}

public static interface OnScrollChangedCallback {

public void onScroll(int left,int top ,int oldLeft,int oldTop);

}

}

## 使用系统浏览器下载

mWebView.setDownloadListener(**new** DownloadListener() {

@Override

**public** **void** **onDownloadStart**(**String** url, **String** userAgent, **String** contentDisposition, **String** mimetype, **long** contentLength) {

**Uri** uri = **Uri**.parse(url);

**Intent** intent = **new** Intent(**Intent**.**ACTION\_VIEW**, uri);

startActivity(intent);

}

});

## ERR\_UNKNOWN\_URL\_SCHEME错误

未知协议名错误，原因



仔细观察图中url可以发现这是一个自定协议的url，究其原因，就是拦截webview中的url, 如果url是自定义协议(如: tel, weixin, alipays 等等)开头的，因为webview只能识别http、https这样的协议，遇到http、https开头的url时会向host发起一个请求, 而遇到自定义的协议时就不知道该如何处理了, 因此就会出现ERR\_UNKNOWN\_URL\_SCHEME这样的错误。

解决办法

给WebView设置WebViewClient，并重写WebViewClient中的shouldOverrideUrlLoading方法

webView.setWebViewClient(new WebViewClient() {

@Override

public boolean shouldOverrideUrlLoading(WebView view, String url) {

if (url == null) return false;

try {

if (url.startsWith("http")) {

//处理http和https开头的url

view.loadUrl(url);

return true;

} else {

//其他自定义的scheme

if (url.startsWith("weixin") //微信

|| url.startsWith("alipays") //支付宝

|| url.startsWith("mailto") //邮件

|| url.startsWith("tel")//电话

|| url.startsWith("dianping")//大众点评

|| url.startsWith("baidu")//百度

) {

//我不跳

//Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse(url));

//startActivity(intent);

return true;

} else {

return true;

}

}

} catch (Exception e) { //防止crash (如果手机上没有安装处理某个scheme开头的url的APP, 会导致crash)

return true;//没有安装该app时，返回true，表示拦截自定义链接，但不跳转，避免弹出上面的错误页面

}

}

});

## ScollView 嵌套 WebView 底部空白，高度无法自适应解决

最近要做一个页面，需要 ScrollView 嵌套 WebView，怎么嵌套，怎么解决焦点和 touch 事件冲突，网上一大堆，这里就不赘述了，但是发现 WebView 从一个高度很高的网页加载一个高度很低的网页的时候，高度无法自适应了，造成底部会有一大片的空白

### Google 官方建议

Google 建议不要在 ScrollView 中使用 WebView，但是毕竟无法避免，产品的需求；

### 移除重新添加新的 WebView

每次加载新的 url 的时候 remove 掉旧的 WebView，重新加入一个新的 WebView 去加载这个新的 url，这种方案是可行的，但是毕竟不太好，效率不高，而且会造成体验上不太好；

### JS 注入，获取页面内容高度，设置成 WebView 高度

JS 注入，通过前端 JS 调用 native 方法，把高度值传递过来，然后通过 layoutparams 去设置 webview 的高度：

mWebView.setWebViewClient(new WebViewClient() {

@Override

public void onPageFinished(WebView view, String url) {

mWebView.loadUrl("javascript:App.resize(document.body.getBoundingClientRect().height)");

super.onPageFinished(view, url);

}

});

mWebView.addJavascriptInterface(this, "App");

@JavascriptInterface

public void resize(final float height) {

getActivity().runOnUiThread(new Runnable() {

@Override

public void run() {

//Toast.makeText(getActivity(), height + "", Toast.LENGTH\_LONG).show();

//此处的 layoutParmas 需要根据父控件类型进行区分，这里为了简单就不这么做了

mWebView.setLayoutParams(new LinearLayout.LayoutParams(getResources().getDisplayMetrics().widthPixels, (int) (height \* getResources().getDisplayMetrics().density)));

}

});

}

需要注意的是上面的 resize 方法一定要上面那么写，写这样是无法成功的，看来宽度也是有影响的

mWebView.getLayoutParams().height = (int) (height \* getResources().getDisplayMetrics().density);

这段代码有亮点可以完善的地方：

* mWebVew.setLayoutParam() 时，应该根据 WebView 父控件的类型，创建 LayoutParam，或者直接用 mWebView.getLayoutParams() 获取 LayoutParams；
* 调用 JAVA 方法的 JS ，可以在网页加载完成时自己加载下面这段 JS 代码（区别在于一个是终端处理，一个是前段处理），比如：

(function(w){

window.App.resize(document.body.getBoundingClientRect().height);

})(window);

加载这段 JS 就可以了，“APP” 和上面一样是自己注册是自己注册的接口名字，思路是一样的，处理方式有差异。