# 2016-7-21

## 解决Eclipse中文乱码的四种方式

使用Eclipse编辑文件经常出现中文乱码或者文件中有中文不能保存的问题，Eclipse提供了灵活的设置文件编码格式的选项，我们可以通过设置编码 格式解决乱码问题。在Eclipse可以从几个层面设置编码格式：Workspace、Project、Content Type、File

本文以Eclipse 3.3（英文）为例加以说明：

1. 设置Workspace的编码格式：

A. Windows->Preferences... 打开"首选项"窗口，点击左侧导航树到General->Workspace，在右侧视图中找到“Text file encoding”选项设置，一种是默认（Default），另一种是从下拉列表中选择（Other）。Eclipse Workspace默认的编码方式是操作系统的编码格式，这跟操作系统的设置有关系；另外我们可以选择Other单选按钮，然后从按钮右侧的下拉列表中选 择需要的编码格式（GBK、ISO-8859-1、UTF-16、UFT-16 etc.）。

B. 第二种方式修改，这是修改eclipse的默认编码格式的：

选择eclipse菜单中【file】---->【properties】,然后修改里面的编码

2. 设置Project的编码格式：

在 Workspace中新建的项目默认继承Workspace的编码设置，我们也可以单独更改某个项目的编码格式。右键点击工程，选择 Properties，打开项目属性设置窗口，左侧导航树选择Resource，在右侧视图中找到“Text file encoding”，两种设置项目的编码格式，默认选中的是“Inherited from container (XXX)”（注：XXX为Workspace设置的编码），我们也可以选择其他的编码格式，设置方式同Workspace。

3. 设置Content Type的编码格式：

有 时我们想使整个Workspace某种类型的文件保持同一种编码格式，这就需要用到Content Type设置来达到目的，具体方式如下：Windows->Preferences...打开"首选项"窗口，左侧导航树选择 General->Content Types，在右侧视图中选择Text->Java Source File，在最下侧有Default encoding输入框，手动输入编码格式，点击Update按钮使设置生效（切记啊！）。有两点需要注意一下：

a. 这个设置使Workspace所有项目下的相同类型的文件有相同的编码格式，改变项目的编码设置不影响项目中文件类型的编码设置；

b. Default encoding是手动输入的，输入的编码名称要准确，否则在打开此类型文件时会显示Unsupported Character Encoding。

4. 设置File的编码格式：

我 们还可以单独设置某个文件的编码格式，一种是通过在文件中设置编码格（如：'charset=UTF-8'）,还可以通过文件属性设置。右键点击某一文 件，选择Properties，打开文件属性设置对话框，右侧导航树选择Resource，在右侧视图中通过“Text file encoding”选择设置文件的编码格式。

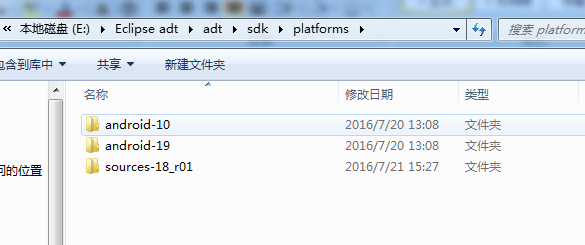
(http://blog.csdn.net/yangqicong11/article/details/10491389)

## 开发中关于编码的问题

在IDE中，整个IDE有个编码格式，IDE中的各个工程也可以有自己的编码格式，工程中不同的文件又可以有各自的编码格式，有时我们会发现，同一工程下的不同文件里，有的文件中没有乱码，而有的文件却乱码了，大多是因为他们各自的编码方式不同。

为了避免这种问题，最好同一个工程都使用一种编码，也可以统一一下整个IDE编码方式，具体的编码方式，最好采用的“utf-8”。

## 查看源码时遇到的问题

今天在看android源码时候，发现没有关联上，于是一番操作后关联上相关的文件（文件夹），但是居然不行，试了很多次，无论是用文件还是用文件夹，都没找到原因。后来发现是关联的文件夹里面没有对应的东西（.java文件），于是重新下了一个sources-18-r01，如下  


终于可以了。回想一下，如果开始直接在上面两个包中，搜索后缀名为.java的文件，发现没有相应的文件，就可以直接新下载一个，不用浪费时间在根本没有的东西上了（本来就没有，还以为是操作出现了瑕疵）。

2016-7-22

## 静态语句块

static{}(即static块)，会在类被加载的时候执行且仅会被执行一次，一般用来初始化静态变量和调用静态方法, 因为在虚拟机的生命周期中一个类只被加载一次；又因为static{}是伴随类加载执行的，所以，不管你new多少次对象实例，static{}都只执行一次。

## Android 开发习惯

（来自网络：http://www.zhihu.com/question/27227425）

作者：漫步  
链接：http://www.zhihu.com/question/27227425/answer/35973793  
来源：知乎  
著作权归作者所有，转载请联系作者获得授权。

受一个朋友邀请来回答这问题，以上各位都回答得很不错，不过对于一个Android开发新手来说，能有更详细的答案那可能会更有帮助。以下内容是来自eoe的一个网友的回答，但现在已经找不回那个帖子，文字已经珍藏起来。感谢原著作者的。

一、Android编码规范  
1.java代码中不出现中文，最多注释中可以出现中文；  
2.局部变量命名、静态成员变量命名：只能包含字母，单词首字母出第一个都为大写，其他字母都为小写；  
3.常量命名：只能包含字母和\_，字母全部大写，单词之间用\_隔开；  
4.layout中的id命名：命名模式为：view缩写\_模块名称\_view的逻辑名称  
view的缩写详情如下  
LinearLayout：ll  
RelativeLayout:rl  
TextView:tv  
ImageView:iv  
ImageButton:ib  
Button:btn  
5.activity中的view变量命名  
命名模式为：view缩写+逻辑名称  
建议：如果layout文件很复杂，建议将layout分成多个模块，每个模块定义一个moduleViewHolder，其成员变量包含所属view  
6.strings.xml中的id命名：

命名模式：activity名称\_功能模块名称\_逻辑名称 activity名称\_逻辑名称 common\_逻辑名称  
strings.xml中，使用activity名称注释，将文件内容区分开来  
7.drawable中的图片命名  
命名模式：activity名称\_逻辑名称/common\_逻辑名称  
7.styles.xml：将layout中不断重现的style提炼出通用的style通用组件，放到styles.xml中；  
8.使用layer-list和selector  
9.图片尽量分拆成多个可重用的图片  
10.服务端可以实现的，就不要放在客户端  
11.引用第三方库要慎重，避免应用大容量的第三方库，导致客户端包非常大  
12.处理应用全局异常和错误，将错误以邮件的形式发送给服务端  
13.图片的.9处理  
14.使用静态变量方式实现界面间共享要慎重  
15.Log(系统名称 模块名称 接口名称，详细描述)  
16.单元测试（逻辑测试、界面测试）  
17.不要重用父类的handler，对应一个类的handler也不应该让其子类用到，否则会导致message.what冲突  
18.activity中在一个View.OnClickListener中处理所有的逻辑  
19.strings.xml中使用%1$s实现字符串的通配  
20.如果多个Activity中包含共同的UI处理，那么可以提炼一个CommonActivity，把通用部分叫由它来处理，其他activity只要继承它即可  
21.使用button+activitgroup实现tab效果时，使用Button.setSelected(true)，确保按钮处于选择状态，并使activitygroup的当前activity与该button对应  
22.如果所开发的为通用组件，为避免冲突，将drawable/layout/menu/values目录下的文件名增加前缀  
23.数据一定要效验，例如  
字符型转数字型，如果转换失败一定要有缺省值；  
服务端响应数据是否有效判断；

二、Android性能优化  
1.http用gzip压缩，设置连接超时时间和响应超时时间  
http请求按照业务需求，分为是否可以缓存和不可缓存，那么在无网络的环境中，仍然通过缓存的httpresponse浏览部分数据，实现离线阅读。  
2.listview 性能优化  
1).复用convertView  
在getItemView中，判断convertView是否为空，如果不为空，可复用。如果couvertview中的view需要添加listerner，代码一定要在if(convertView==null){}之外。  
2).异步加载图片  
item中如果包含有webimage，那么最好异步加载  
3).快速滑动时不显示图片  
当快速滑动列表时（SCROLL\_STATE\_FLING），item中的图片或获取需要消耗资源的view，可以不显示出来；而处于其他两种状态（SCROLL\_STATE\_IDLE 和SCROLL\_STATE\_TOUCH\_SCROLL），则将那些view显示出来  
3.使用线程池，分为核心线程池和普通线程池，下载图片等耗时任务放置在普通线程池，避免耗时任务阻塞线程池后，导致所有异步任务都必须等待  
4.异步任务，分为核心任务和普通任务，只有核心任务中出现的系统级错误才会报错，异步任务的ui操作需要判断原activity是否处于激活状态  
5.尽量避免static成员变量引用资源耗费过多的实例,比如Context  
6.使用WeakReference代替强引用，弱引用可以让您保持对对象的引用，同时允许GC在必要时释放对象，回收内存。对于那些创建便宜但耗费大量内存的对象，即希望保持该对象，又要在应用程序需要时使用，同时希望GC必要时回收时，可以考虑使用弱引用。  
7.超级大胖子Bitmap  
及时的销毁(Activity的onDestroy时，将bitmap回收)  
设置一定的采样率  
巧妙的运用软引用  
drawable对应resid的资源，bitmap对应其他资源8.保证Cursor 占用的内存被及时的释放掉，而不是等待GC来处理。并且 Android明显是倾向于编程者手动的将Cursor close掉  
9.线程也是造成内存泄露的一个重要的源头。线程产生内存泄露的主要原因在于线程生命周期的不可控  
10.如果ImageView的图片是来自网络，进行异步加载  
11.应用开发中自定义View的时候，交互部分，千万不要写成线程不断刷新界面显示，而是根据TouchListener事件主动触发界面的更新

三、Android UI优化  
1.layout组件化，尽量使用merge及include复用  
2.使用styles，复用样式定义  
3.软键盘的弹出控制，不要让其覆盖输入框  
4.数字、字母和汉字混排占位问题：将数字和字母全角化。由于现在大多数情况下我们的输入都是半角，所以 字母和数字的占位无法确定，但是一旦全角化之后，数字、字母的占位就和一个汉字的占位相同了，这样就可以避免由于占位导致的排版问题。  
5.英文文档排版：textview自动换行时要保持单词的完整性，解决方案是计算字符串长度，然后手动设定每一行显示多少个字母并加上‘n‘  
6.复杂布局使用RelativeLayout  
7.自适应屏幕，使用dp替代pix  
8.使用android:layout\_weight或者TableLayout制作等分布局  
9.使用animation-list制作动画效果

四、其他的一些Android开发建议

1.跟上时代的步伐，把Eclipse换成Android Studio，把SVN换成Git，这当然要适合项目开发属性的需要，Git学习中文网站：[https://git-scm.com/book/zh/v2](//link.zhihu.com/?target=https://git-scm.com/book/zh/v2" \t "_blank)

2.勤做总结，推荐使用印象笔记，把一些懂的经验总结起来，把还不懂的文章挂里面，有时间就慢慢消化；

3.定期code review，不断迭代，你总会发现一些不合理的代码，或者需要优化的地方。

4.关注一些知名的技术大V或网站，里面许多东西值得你去消化，推荐：[Android 开源项目集合](//link.zhihu.com/?target=http://www.codekk.com/" \t "_blank)，[http://tech.meituan.com/](//link.zhihu.com/?target=http://tech.meituan.com/" \t "_blank)，[stormzhang](https://www.zhihu.com/people/stormzhang)，但总归来说，去Android官网或者参考一份Java API文档虽说枯燥，但熟悉之后，你会有更大的进步。

5.如想更深入了解可阅读珍藏许久的文章：[Android应用程序开发以及背后的设计思想深度剖析](//link.zhihu.com/?target=http://www.uml.org.cn/mobiledev/201211063.asp%231" \t "_blank)

6.如果你公司没有强大的测试团队，发布应用前最好把应用放到测试平台去测测，比如云测之类的；

7.取应用包名的时候切忌取太容易重复的，如果同款手机已经有该包名，那么会因为签名不同而导致安装不上，这也怪中国安卓市场太多，无法像Google Play那样进行包名审核。

## 如何看懂android源码

1.首先看AndroidManifest.xml告诉你主activity service receiver contentprovider,

2.根据生命周期去看主activity的操作，对其它类的初始化 调用等 主要是oncreate onstart 不要太关注细节

3.再去看看其它类的实现，看看是否有继承原生的类 实现原生的接口，这个时候就是百度 官方文档 或者源码 查查这个类干什么的 有什么方法

4.看个大概理一下代码的结构，分析一下 比如说输入法 主要有键盘容器 按键区 候选区 键盘 按键

5.细节上加Log打印信息 看看自己的理解 修改bug

6.剩下的事情就是你可以考虑一下自己能怎么实现，哪个部分可以优化

如果单论一个简单工具型app来看的话，剥除layout xml等布局文件，整个项目代码量应该在5w-10w档次，更何况Android每个界面独自为Activity，首先看AndroidManifest.xml文件就可以对整个项目有大概了解有哪些Activity View Service，然后由点带面·~~

按照楼主描述就是新手刚入门，如果不是公司项目的话，建议仿照你要看的Android项目来自己实现一个，这样时间成本虽然更大，但是对项目掌握 学习更好。

如果是公司项目，那就加班去加断点看代码啊，还用问吗

——————————————————————————————————————

比如简单地实现一些material design，了解一些简单地开源项目，简单了解一些常用库的功能和用法。然后github上找别人比较简单地项目，比如简单的天气，新闻，笔记类项目，大神的项目就先别想，找些同样初学的人做的项目，我看的是第一个仿知乎日报的项目，叫做Kuaihu，想看的github上搜索去吧，这之前还看了下吴小龙的androidmvpsample，也在github有，画一个小时弄懂的才看kuaihu。kuaihu看了两天，结构还是比较清楚，注意各个类的继承和接口关系，然后看manifest找到主活动，然后从主活动看起，看它怎么加载界面，怎么加载数据。怎么把各种界面连接起来，其实无非就是一些点击事件，顺着这些点击事件，找下一个界面，难的应该是加载数据这一块，就是网络，图片，adapter，数据库的各种裙带关系。看着理解吧，不过话说回来，最终还是多花点时间多有点耐心，硬啃下来吧

# 2016-7-25

## Android System.out.println在控制台无输出

java中一般打印方法为System.out.println("");如果程序没问题，要打印的内容会在控制台显示。

但在Android平台，调用System.out.println("")函数，即使程序正确，也不会在控制台显示。不过可以在LogCat中查看。Android自身提供的打印方法是Log.v,引入android.util.Log 即可使用。查看打印的字符也不在console面板，而在Log面板上，可以选择ide右上角debug模式，查看Log面板。

2016-7-28

## 在debug时出现source not found的解决办法

在eclipse adt 中debug调试一个android项目时，总是提示source not found（进入了类ActivityThread.java这个类中，但是这个类找不到），网上找了很多原因，都没用，给出的解决方法大多如下：

方法a：

Debug 视图下

-》在调试的线程上 右键单击

-》选择Edit Source Lookup Path

-》选择Add

-》选择Project

-》选择正在调试的project

选择相应的Project 进行OK确定即可

方法b：

1.将工程目录下的.metadata\.plugins\下的 org.eclipse.debug.core和org.eclipse.debug.ui目 录删除

2.重启eclipse即可

方法c：

<http://stackoverflow.com/questions/1960158/eclipse-debugging-source-not-found>

方法d（最后解决问题的方法）：

<http://zhidao.baidu.com/question/1689479740656221868.html?fr=iks&word=activitythread.performlaunchactivity%28source+not+found%29&ie=gbk>

“提示source not found的时候，它会提示你是否查找的，你把放源代码的目录再告诉它一遍，它就能找到了，就是这么变态”

经过摸索，在方法a中，选择add之后，不要选择project

Debug 视图下

-》在调试的线程上 右键单击

-》选择Edit Source Lookup Path

-》选择Add

-》选择“File System Dicrionary”的选项（具体记不清了），然后选择相应的文件夹，把有源码的文件夹的地址放上就OK了，笔者的文件夹为E:\Eclipse adt\adt\sdk\platforms\android-19-，就是我们要看源码时配置的地址。这时候，我们就很困惑了，配源码时不是已经配置过一次了吗，为什么还要在配一次，原因正如一位网友所说，因为“它变态”！（Annoying for hours）

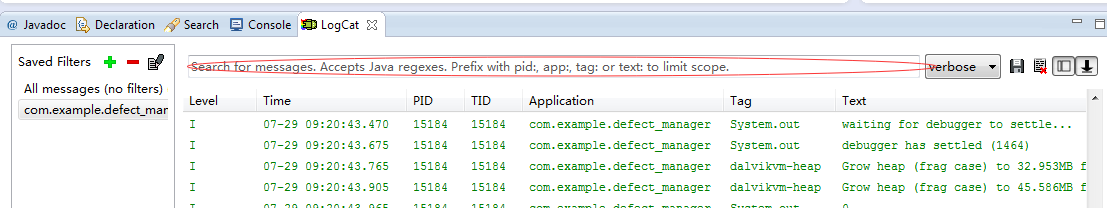
2016-7-29

## Eclipse连上真机logcat中出现很多无关信息

大部分都是手机中其他应用或者手机状态的信息，但很多时候无关信息太多会妨碍我们查看调试程序的信息，解决方法如下：

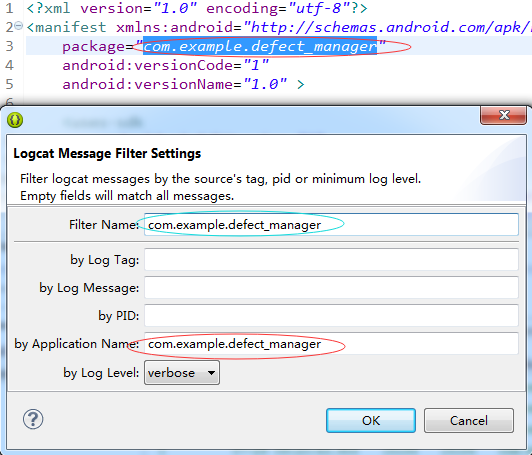
方法A.

logcat窗口上方有个搜索栏，可以用关键字搜索想要查看的相关信息，但是当信息太多时候，也是不好用（不推荐）；



方法B.

点击左侧绿色“+”按钮，出现：



其中，Filter Name（绿框）随便填，by Application Name（红框）中的信息填写manifest.xml文件中的package中的信息，这样，在logcat中就只会显示和你本项目有关的信息了。

## 连上真机后LogCat不显示日志的问题

有时候在Eclipse打开后，将真实的Android设备通过USB线连接到电脑，并不会在LogCat中看到有日志输入。

这可能是因为电脑中还安装了其它手机工具程序，占用了运行ADB（Android 调试桥）服务所需的端口，导致不能通过ADB连接到Android设备，进而导致没有日志输出。

可以用下面的步骤解决该问题：

1.关闭Eclipse；

2.打开操作系统命令行工具，使用cd命令切换到Android SDK的bin目录，输入adb nodaemon server，查看adb服务所使用的接口是否被占用，如果占用，给出的提示会显示占用的端口，一般是 5037；

3.继续在命令行工具中输入命令 netstat -ano | findstr 5037， 查看占用了该端口的进程信息，假设得到占用该端口的进程号为 xxx；

4.继续在命令行工具中输入命令 taskkill /F /PID xxx，杀掉这个进程；

5.重新启动Eclipse。

***Tips：*** Android调试桥（ADB）是一个客户端-服务器应用程序，可以通过它来连接到任何Android设备（真实设备或者虚拟机）。

## 对GIt中的私钥和公钥的解释

GIT服务器上存储的是公钥，你本地存储的是私钥，当你push本地代码库到远程代码库，服务器会要求你出示私钥，并且用你出示的私钥和它的公钥配对来完成认证。

由于使用的是**不对称加密**，所以公钥可以公开，只要保管好私钥就可以。  
路人甲只要没有私钥，所以无法push。这样才能保证开源项目的完整性，否则阿猫阿狗都去push代码，那项目不就乱套了？

如果路人甲想对你的项目做出贡献，那么以GITHUB为例，他会**发送一个pull request**给你，然后**由你**来审核他作出的改变，如果审核通过，那么你就可以**将他的pull request合并**到你工程的某一分支中。这里只以GITHUB为例，私有的GIT服务器可能没有类似功能，比如GITLAB就没有pull request

# 2016-8-1

## ListView的item中的TextView控件点击不灵敏

在item的布局文件里面的*TextView控件中*加上属性

android:textIsSelectable=*"true"*

同时，在item的*根部局文件中*加上属性

android:descendantFocusability=*"blocksDescendants"*

调试过程中，发现虽然控件点击灵敏了，但是发现

### 第一次点击TextView控件无效，之后的点击才有效

网上给的原因是：当前（第一次点击之前）TextView控件没有获取焦点，给其加上属性android:focusable=*"true"*

即可

## 点击按钮返回上个Activity

开发中遇到需要点击按钮返回上上个Activity的需求，这里记录下解决的方法：

A，B，C都是activity，顺序是A--->B--->C

先说一下怎么返回上个Activity：

从C返回B，C--->B：在C中（按钮的监听事件中）使用this.finish(),结束当前activity，就会直接到B中；

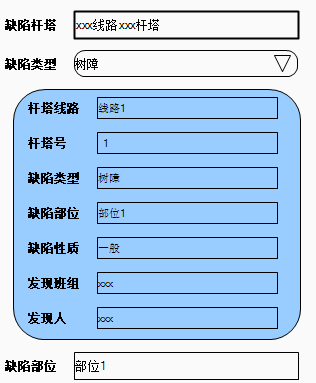
从C--->A，直接返回，不经过B:

当从B--->C时，在跳转代码写完的时候，在B中调用this.finish()，接着，在C中使用this.finish(),就不会经过B了，因为B已经finish（）了，从而达到C直接返回A的效果；

**Tips：**最好不要使用Intent跳转来达到返回上一个Activity的效果，因为它只是界面上过去了，但并不是返回，而是界面跳转【可以通过activity中的界面转换的效果（界面是左移还是右移）来观察他们的区别】。

## Spinner中不同的下拉选项有不同的布局

如图所示：



问题描述：

缺陷类型是可选择的（选项在xml文件中配置），当选择的是“树障”时，出现蓝色部分，如果是其他选项，则不出现；

解决办法：

将蓝色部分在整个布局文件中，用LinearLayout布局属性包裹起来，在activity中定义此LinearLayout，

private LinearLayout defect\_SZSJ; // 树障数据

defect\_SZSJ = (LinearLayout) findViewById(R.id.defect\_SZSJ);// 树障数据

接着定义一个Spinner，

private Spinner spinner;

spinner = (Spinner) findViewById(R.id.sp\_Defectclass);

然后调用Spinner控件的setOnItemSelectedListener（）方法**进行监听**，在此方法中

String value=spinner.getSelectedItem().toString();

将得到的值（value）进行判断，

if (value.equals("树障")) {

defect\_SZSJ.setVisibility(View.VISIBLE);

} else {

defect\_SZSJ.setVisibility(View.GONE);

}

总结一下：

1、LinearLayout类的setVisibility（\*\*）方法的使用；

2、不止控件类，布局类也是可以定义的，也有其相应的方法和属性。

## 对LayoutInflater的理解并与findViewById()和setContentView()比较

LayoutInflater()：

——是用来找res/layout/下的xml布局文件，并且实例化（实例化为View类对象）；

——对于一个没有被载入或者想要动态载入的界面，都需要使用LayoutInflater.inflate()来载入；

findViewById()：

——找xml布局文件下的具体widget控件(如Button、TextView等)；

——对于一个已经载入的界面，就可以使用Activiyt.findViewById()方法来获得其中的界面元素；

setContentView()：

—— 一旦调用, layout就会立刻显示UI；

setContentView()一旦调用, layout就会立刻显示UI；而inflate只会把Layout形成一个以view类实现成的对象，有需要时再用setContentView(view)显示出来。一般在activity中通过setContentView()将界面显示出来，但是如果在非activity中如何对控件布局设置操作了，这就需要LayoutInflater动态加载。

*LayoutInflater与setContentView()的字面理解（个人理解）：*

前者是“在平板中布局”，后者是“设置上下文视图”，因此：

就可以理解activity中一般使用setContentView（）了，因为activity是content的子类；

而非activity中一般使用LayoutInflater，因为LayoutInflater是在“平板上”布局，不是“activity”

（http://www.cnblogs.com/top5/archive/2012/05/04/2482328.html

<http://blog.csdn.net/lmj623565791/article/details/38171465）>

# 2016-8-8

## Android中EditText控件自动换行

在EditText控件中添加android:inputType=*"textMultiLine"*属性（没想到inputType中还可以设置自动换行）

有时还会遇到EditText中文英文长度控制，英文字母最多N个，而中文最多N/2个。这样的话直接用LengthFilter就无法实现了，但是我们可以通过自定义InputFilter实现：

（http://blog.csdn.net/songzhiyong1121/article/details/8758902）

## in的嵌套查询

in关键字用于where子句中用来判断查询的表达式是否在多个值的列表中。返回满足in列表中的满足条件的记录。

（http://www.cnblogs.com/kissdodog/archive/2013/06/03/3116284.html）

## 一个低级错误（StringBuilder）

今天在使用StringBuffer的时候，直接

StringBuffer chosedPeople;

没有new StringBuffer(),然后

chosedPeople.append(\*\*);

很显然的报了

NullPointerException，

以后要注意。

*（StringBuilder是非线程安全的，StringBuffer是线程安全的）*

## Android中什么时候需要为控件注册监听

Android中不需要注册监听就能获取控件中的（输入）内容，如

~~edt\_mission\_time.setOnClickListener(this);~~

也能通过edt\_mission\_time.getText().toString();

获得控件edt\_mission\_time中的内容。（因为监听是对行为的响应，并产生另外的行为），所以，当只需要获取信息而不需要作出相应的动作时，是不需要注册监听的

## 使用SQLite遇到的坑

——

想在SQLite中将一个字段设为主键并自增，遇到一点问题：

字段为ID，数据类型为INT(11),当设置ID为“主键”时没有问题出现，但是当添加“自增”时，就提示需要将INT改为INTEGER。之后还是提示不行，于是抱着尝试的态度将INTERGER(11)改成INTEGER就可以了，不能指定长度。（此处是使用SQLiteStudio，不知道使用命令行是不是一样的）

——

SQLite中的数据类型：

DATETIME：使用此类型，存入日期数据时，使用的是12小时制；

DATE：使用此类型，存入日期数据时，使用的是24小时制；

## Android中数据库的关闭问题

通常在一个Android Activity中有很多方法，可能在不同的方法中都会对数据库进行操作，那么db的关闭问题就不得不考虑，下面提供两个笔者的解决办法：

*问题由来：*  
在方法A中db.close(),在方法B中又调用了db的方法，会提示:

错误( java.lang.IllegalStateException: attempt to re-open an already-closed object)，

因为在用db之前需要将其打开;

（这里A方法在B方法之前调用，所以执行到B的时候db已经关闭了）

*方法1：*

在每次用到db的时候，都将其打开，用完后关闭；

（不推荐，消耗资源，性能差）

*方法2：*

了解活动的生命周期，以及活动中各方法间的调用（先后执行）关系，将db.close(),放到最后执行的方法中，就不需要每次没打开和关闭了；

（使用此方法，要掌握好activity的生命周期，以及活动中方法执行的先后顺序）

*两者比较：*

当activity中多个方法经常和数据库有交互的时候，方法2是更好的选择；

当整个activity跟数据库只有很少的交互的手，方法1似乎比方法2更好。

## SQLite中模糊查询的条件是变量

今天在用SQLite写模糊查询的时候，条件是个变量，费了些时间，如下：

*原来的语句：*

"select FZR,RWFL,GZFX,RWNR,ZT,RWSJ from task where CYRY like '%''+username+''%' and RWSJ = ?"

这里的username是个变量，但是尝试了几次，也换了不同的写法，在(java)代码中运行的时候都解析不出它的值（值应该是test1），还是把它当成了字符串，最后终于测试到了正确的写法：

*正确的写法：*

"select FZR,RWFL,GZFX,RWNR,ZT,RWSJ from task where CYRY like '%"+username+"%' and RWSJ = ?"

浪费了些时间，在此记下来，希望以后再遇到能很快解决。

*Tips（小技巧）：*

代码是在Eclipse中写的，其中的代码会有颜色的区分，可以注意到，原来错误的写法中，username的颜色是蓝色，和整个SQL语句中的颜色一致，可见是被当成了 字符串；在正确的写法中的username是黑色的，可见是被当成了 变量。（以前没注意到此技巧，如果之前注意到的话，这次能节约很多的调试时间）

## 在一个Activity中调用另一个xml布局文件

有时候，我们需要在一个Activity中调用另一个xml布局文件，即非本Activity所绑定的xml布局文件中的控件，这时候就不能直接findViewById，不然会报错指向空对象，这时就需要像下面这样做。

LayoutInflater factory = LayoutInflater.from(当前类.this);

View layout = factory.inflate(R.layout.你要获取的另一个XML, null);

TextView textview = (TextView) layout.findViewById(R.id.控件ID);

一行写完就是：

TextView textview = (TextView) LayoutInflater.from(当前类.this).inflate(R.layout.你要获取的另一个XML, null).findViewById(R.id.控件ID);

## Invalid layout of java.lang.String at value问题的解决

问题描述：

在Eclipse中建立android工程后在里面写一个java类，在类中写上main方法，然后运行这个类，提示上述错误

解决办法：

选中此类——>右击run as——>run Configuration——>在classpath选显卡中将android4.4.2 Remove 掉即可；

不足：

如果有其他类中仍然用了main方法运行，还需在那个类中重复此操作。

## Eclipse中的内存分析

<http://my.oschina.net/jielucky/blog/169793>

<http://www.cnblogs.com/java-class/archive/2013/04/18/3027884.html>

<http://chiyx.iteye.com/blog/1528782>

<http://my.oschina.net/biezhi/blog/286223>

<http://blog.csdn.net/rogerjava/article/details/23255277>

## 日期正则表达式demo

**public** **class** RegexUtil {

**static** **boolean** *flag* = **false**;

**public** **static** **final** String *date\_regex* = "^[0-9]{4}-(0?[0-9]|1[0-2])-(0?[1-9]|[12]?[0-9]|3[01])$";

**public** **static** **boolean** isRegexValidateTrue(String str) {

**try** {

Pattern pattern = Pattern.*compile*(*date\_regex*);

Matcher matcher = pattern.matcher(str);

*flag* = matcher.matches();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** *flag*;

}

}

# 2016-8-15

## Java中的回调函数

先放上一个知乎链接:http://www.zhihu.com/question/19801131,

这个链接解释的很好。

简单的说下我看完上面链接后对回调函数的理解（这里旨在用简明的语句和代码说明有关回调函数的问题，能让读者快速的理解它，如果你时间充足又想更深一步，不妨去看一下上面给出的链接）：

在我们用 回调函数 的时候，一般会涉及到3个函数（方法），暂时不说这三个函数之间的关系，先看完下面的例子，然后再回来说明：

从简单的开始：

在A类中，（暂时）有两个方法——a1()、a2():

class A{

a1(){

...

}

a2(){

...

}

}

B类中，有一个方法b1():

class B{

b1(){

...

}

}

在开发中经常遇到在A类中方法a1()调用B类的方法b1()【例如应用层调用系统的库函数】，但有时，b1()方法需要传入参数，当需要的参数是a2()时，这个a2()就被称为——*回调函数*。为了区分，a1()称为*起始函数*，b1()称为*中间函数*（这就是上面提到的三个函数）。

注意，这里往b1()中传入的参数是A类中的方法a2()；有时我们也会直接传入一个A类的引用a，然后通过a.a2()来达到传入方法a2()的效果，但这种不推荐。想想看，本来只要传入个方法为参数，现在你给了A的引用，那么在B中就能得到A类中的任何信息，显然这是不安全的。那么，怎么解决呢？

复杂点的：

在说这个之前，还是说一个容易被忽略的问题：

网上的很多博客中，都把起始函数和回调函数看做是一体的，认为回调是一种双方互动（if you call me , I will call you back）。其实不是这样的，回调应该是一种三方联动，理解这个概念很重要。

好了，说下怎么解决上面提到的安全问题：

其实解决上面提到的安全问题很简单，就是不用A的引用当参数，直接中a2()这个方法为参数；

但是，这样的方法也有一个弊端，

当用a2()作为参数时，在B中最终还是通过A中的a2()来解决问题的，那么当还有其他类，例如C类，

class C{

c1(){

...

}

a2(){

...

}

}

其中a2()方法也是作为回调函数，b1()还是作为中间函数，同样需要a2()方法作为参数，那么这个时候在B中最终就是通过C中的a2()方法来解决问题的。这样写显然会让你觉得不舒服，那么有没有什么好的解决办法？答案是肯定的。我们可以写一个接口D，D中有a2()方法，然后然A和C都实现D接口，这样的话，在B中的b1()方法里，就不用传入各自的a2()方法了，可以直接传入一个D接口的引用，例如D a、或者是D c，利用向下转型来解决上面提到的安全问题（此时只能调用a.a2()、c.c2()方法，不能调用a.a1()、c.c1()方法，上面写的是D a、D c，而不是A a、C c）。

到这里，回调函数大概说的差不多了，但还差一点：

回调函数中，通常使用匿名内部类的方式，即将D a、D c改成new D(){};的形式，至于为什么叫匿名内部类，大家通过代码得差异也能看出来，后面的写法并不像前者赋予a、c之类的名字，而是直接通过new的形式。

到这里，怎么用回调函数都了解差不多了，在回到我们为什么要用回调函数这个问题上（或者说回调函数适合什么场景），

回想一下上面的例子，a1()中用到b1(),b1()又需要a2()作为参数，b1()执行完之后再回到a1()继续执行下面的代码，那如果b1()是个很耗时的操作，那代码就会在a1()那阻塞了，用户体验将很不好，所以，回调函数一般用其做异步回调，这也是它为什么被广泛运用的原因。

至于怎么异步，...后面我们再说。

## 经纬度换算

E 108.90593度 N 34.21630度

108.90593度换算成度分秒(DMS)东经E 108度54分22.2秒?转换方法是将108.90593整数位不变取108(度),用0.90593\*60=54.3558,取整数位54(分),0.3558\*60=21.348再取整数位21(秒),故转化为108度54分21秒。

同样将度分秒(DMS):东经E 108度54分22.2秒 换算成度(DDD)的方法如下:108度54分22.2秒=108+(54/60)+(22.2/3600)=108.90616度。

因为计算时小数位保留的原因，导致正反计算存在一定误差，但误差影响不是很

大。（这里没有涉及和米的换算）

## Android（Eclipse ADT）中互换控件位置造成的控件上点击事件也互换的bug

Android中，当对页面布局中的控件做了调整（比如说控件间位置互换）之后，最好将这个项目“clean”一下，不然有时候Eclipse会“抽风”，就是控件间的点击事件也互换了，虽然代码中逻辑上都对的。

分析原因可能是：

R文件中的id可能和控件位置有关，当控件位置互换了之后，虽然字面上的id名字没变，在id上的点击事件也没变，但此时R文件中的id真正所指向控件可能也互换了。所以会产生，当我们点击这个控件时，相应的点击事件变成和他互换的控件上的了（really confused me）。

解决办法有二：

——“clean”一下此项目；

——把自动生成的“gen”目录删了，让它等会在重新自动生成；

## 获取ListView中的item值（不用重写BaseAdapter）

今天发现一个能够获取ListView中item值的简单方法：

一般有上面这个需求都伴随着在item上加监听事件，即调用

listView.setOnItemClickListener(\*\*\*)这个方法，在这个方法中，含有四个参数，如下：

listView.setOnItemClickListener(**new** OnItemClickListener() {

@Override

**public** **void** onItemClick(AdapterView<?> parent,

View view, **int** position, **long** id) {

......

}

};

其中的view，代表的就是item布局文件的view，可以通过

findViewById(R.id.\*\*)来取得，其中的\*\*代表item布局文件中的控件id，假设要获取item中的TextView控件的值，写法如下：

TextView text = (TextView) view.findViewById(R.id.*normal\_left*);

String msg = text.getText().toString();

上面代码中的normal\_left就是item布局文件中的TextView控件，msg就是得到的TextView控件中的文字信息。这样，需求就完成了。

Tips：之前一直存在的疑问是，这个item的布局文件是个单独的布局文件，也没有在主布局文件使用标签的方式中引入，为什么使用

findViewById(R.id.\*\*)的方式没有报错呢？

原因是，我们在使用adapter的时候，已经使用R.layout.item的方式将这个item的布局文件在代码中引入了，所以即使没有在主布局文件中引入，使用上面的findViewById()这个方法也不会出错。

## ListView中的item监听点击反应不灵敏

在项目中，需要给ListView中的item有其他控件，但需要item设置监听（不是给其中的控件设置），代码如下：

listView.setOnItemClickListener(**new** OnItemClickListener() {

@Override

**public** **void** onItemClick(AdapterView<?> parent,

View view, **int** position, **long** id) {

......

}

});

但是在测试的时候会发现，在手机中点击item时，反应很不灵敏，可能需要点击很多下，才会触发点击事件，原因如下：

其实是item中的其他控件，抢走了item的焦点，导致我们虽然点击了item，但是item并没有识别出来，因为他们没有焦点；而有焦点的item中的控件并没有被我们加上监听事件，所以，点击不会出现反应（反应很不灵敏）；

解决方法如下：

在item的布局文件的根标签中，加上如下属性：

android:descendantFocusability=*"blocksDescendants"*

这个属性的意思是，屏蔽子控件，自身获得焦点，

为了保险起见，我们在给每个子控件中加上如下属性：

android:focusable=*"false"*

android:clickable=*"false"*

让子控件获取不到焦点，并且不可点击（双重保障）。

这样，item的监听事件就会很灵敏了。

## Android中使用shape自定义形状

*用处：*

比如说用到一个自定义的形状作为背景；

*用法：*

在drawable目录下新建shape1.xml文件，在布局文件（控件）中通过

android:background=*"@drawable/shape1"*的方式引入；

*写法：*

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<shape** xmlns:android="http://schemas.android.com/apk/res/android" **>**
4. <!-- 填充 -->
5. **<solid** android:color="#ff9d77" **/>** <!-- 定义填充的颜色值 -->
7. <!-- 描边 -->
8. **<stroke**
9. android:width="2dp"
10. android:color="#fad3cf" **/>** <!-- 定义描边的宽度和描边的颜色值 -->
12. <!-- 圆角 -->
13. **<corners**
14. android:bottomLeftRadius="5dp"
15. android:bottomRightRadius="5dp"
16. android:topLeftRadius="5dp"
17. android:topRightRadius="5dp" **/>** <!-- 设置四个角的半径 -->
19. <!-- 间隔 -->
20. **<padding**
21. android:bottom="10dp"
22. android:left="10dp"
23. android:right="10dp"
24. android:top="10dp" **/>** <!-- 设置各个方向的间隔 -->
26. **</shape>**

当四个角的半径都相同时，可以这样写：

1. <!-- 圆角 -->
2. **<corners** android:radius="5dp" **/>**

属性基本上就包括上面的几个，有时候还可以加上渐变属性：

1. <!-- 渐变 -->
2. **<gradient**
3. android:endColor="#FFFFFF"
4. android:gradientRadius="50"
5. android:startColor="#ff8c00"
6. android:type="radial" **/>**

具体可以参考：

<http://blog.csdn.net/bear_huangzhen/article/details/24488337>

## Cannot refer to a non-final variable \* inside an inner class defined in a different method"错误解析

在Java中，使用局部内部类或者匿名内部类时（之所以称为“内部”，是因为该类在方法内部），若该类调用了所在方法的局部变量，则该局部变量必须使用final关键字来修饰，否则将会出现编译错误。

*原因：*

在方法中定义的变量是局部变量，当方法返回时，局部变量对应的栈就被回收了（局部变量在所处的函数执行完之后就释放了），当方法内部类去访问局部变量时就会发生错误。当在局部变量前加上final时，变量就成了“常量”，这样在编译器进行编译时(即编译阶段)就会用变量的值来代替变量。

（这种矛盾是由局部内部类可以访问局部变量但是局部内部类对象和局部变量的生命周期不同而引起的。）

*例如：*

final Panel p = new Panel();

定义一个引用p,指向heap的对象new Panel(),前面添加final是为了不让你修改p的指向地址，但是你可以修改这个地址指向对象的内部数据。

*Tips：*

当你在类中定义变量时，在其前面加上final关键字，那便是说，这个变量一旦被初始化便不可改变，这里不可改变的意思*对基本类型来说是其值不可变*，而对于*对象变量来说其引用不可再变*，但*引用变量所指向的对象中的内容还是可以改变的*。

## Bundle传值时的坑

今天在从A活动传值给B活动时，采用了Bundle传值，由于记不清用法，翻了一下身边的书，书上是这样说的：

在A中传值：

Intent intent = **new** Intent(MissionManage.**this**,

MissionShow.**class**);

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

bundle.putString("mId", mId);

intent.putExtras(bundle);

startActivity(intent);

其中，MissionManage就是A，MissionShow是B，

在B中接收值：

Intent intent = **new** Intent();

String result = intent.getStringExtra("mId");

结果测试的时候显示result中的值为null，试了几次之后，对书上的内容产生了怀疑，想想也是，在B中每次还要new Intent()，那还怎么取到放进取得bundle中的值啊，后来发现，正确的写法应该这样写：

Intent intent = **this**.getIntent();

String result = intent.getStringExtra("mId");

调用当前的上下文实例的getIntent()方法即可，看看getIntent()的源码也确实如此：

/\*\* Return the intent that started this activity. \*/

**public** Intent getIntent() {

**return** mIntent;

}

上面的注释已经解释的很清楚了，获取传值的intent实例，才能取到我们传进去值。

## 在ListView的setOnItemClickListener()方法中，不能使用cursor.getString()等方法

有时需要从数据库中取一些数据，然后将这些数据用ListView展示出来，这样ListView中就有很多子项（item），同时我们可能还需要给这些item加上监听事件，比如说加上setOnItemClickListener()的监听事件。

在今天工作的过程中，我想在监听方法的内部用cursor.getString()这样的方法去取得某一列的值，比如cursor.getString(1),即代表从cursor中取出第二列的值，我们都知道光有列还不行，还得有行，这样才能定位到具体的值，当时想当然的认为点击哪一个item，就是选中了行。事实证明，这种想法错了。每次都提示：”CursorIndexOutOfBoundException: Index 1 requested, with a size of 1”这样的错误，而网上关于这个的解释很少，只能自己调试。后来终于有了一点眉目：

因为在item的监听事件中，使用cursor.getString(1)这样的方法，只能根据传的参数，确定是那一列，并不会跟我们点击的item而确定是哪一行，所以，每次这样写，都不能定位到确定的一个值，才报上述错误，到这里，总算对错误有个大概的了解了。

那如果我们就想在setOnItemClickListener()方法中去取得数据库中某一确定位置的值，应该怎么办呢？

我是这样解决的,在：

**while** (cursor.moveToNext()) {

......

}

的代码段中，使用cursor.getString(1)方法，应为moveToNext()是逐行的，在加上给定的参数作为列数，可以很好的定位我们想取得值，并取出来。然后将取得的值赋给item中的某一控件（假设为TextView），接着在setOnItemClickListener()方法中使用：

TextView midTv = (TextView) view.findViewById(R.id.*item\_mid*);

注意这里的view，这里特地用不同的颜色标出来，这个view代表的是item的布局View，只有这样，才能调用item中的控件item\_mid，不然是会报空指针异常的（因为找不到此控件，只有通过此控件所在的布局View才能得到此控件的实例）。看来这个view挺重要的，那么我们从哪获得这个view呢，不担心！在我们使用setOnItemClickListener()的时候，我们需要去重写这个借口Listener中的方法onItemClick()方法，而这个方法提供了四个参数，其中view就是这四个参数之一。所以，我们想要的其实都已经准备好了，并不需要我们再去麻烦了。这也是我们为什么要在setOnItemClickListener()方法中（上文灰色文字）去使用view.findViewById(\*\*\*)的原因，因为它里面为我们提供了我们想要的view（指向item的view）。

到这里，我们就在setOnItemClickListener()方法中取得了这个存着数据库某一位置信息的控件实例——midTv，接下来就很简单了，我们可以通过

midTv.getText()等方法去取得这个信息，也就变相的实现了在

setOnItemClickListener()方法中取得了数据库中某一位置的信息（不直接使用cursor）。

# 2016-8-22

## 删除表中全部数据的两种方式（SQL）

A.delete from student

该方式下数据库空间不一定能及时释放出来，而且如果有identity字段（被标识字段，也就是该字段自增长），之后再插入数据也不会从定义的初始值开始自增；

B.truncate table student

该方式可以及时释放表空间，同时也能让是identity的字段的新增数据的值从定义的初始值开始。

## 第一行代码

——android的系统架构 P2

Linux内核层（提供诸如蓝牙等硬件的底层驱动）、系统运行库层（SQLite、 OpenGL|ES）、应用框 架层（各种API）、应用层（安装的app）；

——android系统提供了哪些（东西）P4

—四大组件

活动（Activity）：所有应用中看到的东西，都放在活动中；

服务（Service）：无法看见，后台运行，即使退出了应用仍然可以继 续运行；

广播接收器（Broadcast Receiver）：允许你的应用*接收*系统中（其 他应用）*发出*的广播，你的应用也可以发出广播；

内容提供器（Content Provider）：方便应用程序之间共享数据；

（这四大组件都要在Manifest.xml中注册才能使用）

—丰富的系统控件

—SQLite数据库

—地理位置定位

—强大的多媒体

—传感器

——AndroidManifest.xml P18

—四大组件的注册

—应用程序的权限声明

—指定程序最低兼容版本和目标版本

——res目录

—drawable开头的文件夹用来放图片（这是在EC中的，AS中的是mipmap）；

—values开头的文件夹用来放字符串；

—layout文件夹用来放布局文件；

—menu文件夹用来放菜单文件；

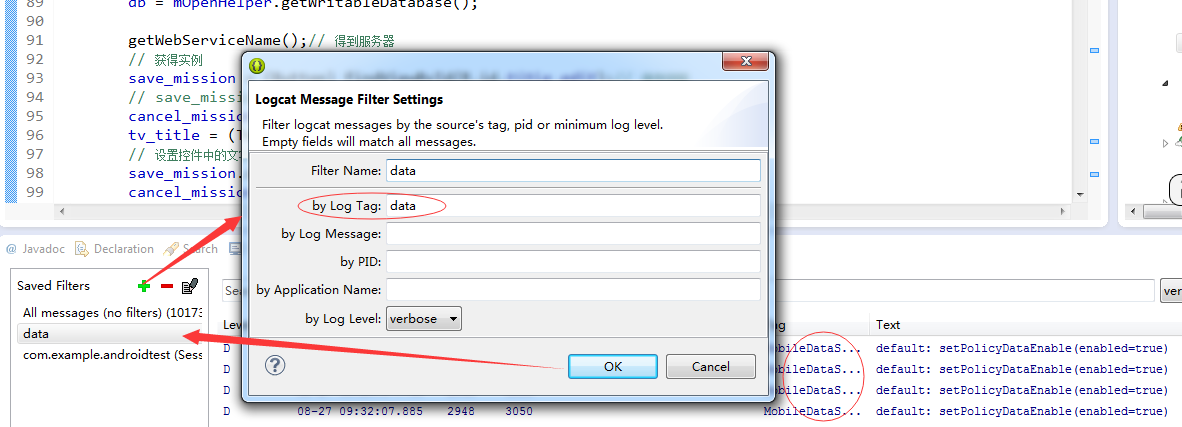
——android中的日志工具Log与Logcat

—Log是android提供的日志工具类，可以通过其中的不同方法，来打印出 不同日志级别的信息供查看；

—Logcat是Eclipse中的一个视图，其中显示的是应用程序的（各种各样） 日志信息，可以选择不同的日志级别来过滤不想查看的信息；



—Log类中的不同方法都需要一个TAG，可以在Logcat定义一个TAG的过 滤器，这样就只查看有该TAG的日志信息，这个在日志很多的时候很好 用；



如图，点击绿色“+”自定义filter，这个filter的名字我们就设为和tag 一样的data，点 击“OK”，在左侧出现名为“data”的filter，这个 filter中的tag信息是data，所以在 logcat中只显示包含“data”字符 串的日志信息；

—如果Logcat中没有打印出任何信息，可能是因为当前的设备失去了焦点。 解决方法是进入DDMS视图，在Devices窗口中点击你的设备，日志信息就 会出来了（如果还不行的话，重启IDE就好了）；

（为什么用Log的方法打印日志信息，而不用system.out.println()方 法打印，因为后者：

日志打印不可控制、

打印时间不可控制、

*不能添加过滤器、*

*日志没有级别区分*

*后两个原因其实就是（导致）不能和logcat配合使用）* ）

——关于活动

— 一个app可以包含零个或多个活动，但不包含任何活动的app很少见， 因为没有活动就意味着没有界面；

—onCreate方法时一个活动被创建时必定要执行的方法（经测试，活动会 先执行它的构造方法，在执行onCreate方法【为什么构造方法先？】）；

—如果要一个活动在打开应用时就执行，需要在manifest.xml文件中注册 的<activity>标签中，加上<intent-filter>子标签，然后在此标签 中再加上相关子标签；

（如果应用程序中没有声明任何一个活动作为主活动，这个程序仍然是 可以正常安装的，只是无法再启动器中看到或者打开这个程序，这种程 序一般都是作为第三方服务供其他的应用在内部进行调用的，如支付宝 的快捷支付服务）

—不推荐在活动中直接编写界面，因为android程序的设计讲究逻辑和视图 分离，通常的做法是，在布局文件中编写界面，然后在活动中引入；

——Intent的使用 P43

—Intent是android中*组件之间*交互的一种重要方式，它*不仅*可以指明当 前组件想要执行的动作（启动活动、启动服务、发送广播等），*还*可以 在不同组件间传递数据；

—显式Intent与隐式Intent（本质上就是对Intent中不同构造函数的使 用）

*显式Intent：*

Intent intent = new Intent(First.this,Second.class);

startActivity(intent);

这个构造函数接收两个参数，第一个参数Context提供启动活动的上下 文，第二个参数clas指定想要启动的目标活动，通过这个构造函数就 可以构建出Intent的“意图”，然后通过startActivity()来执行这 个intent。

*隐式Intent：*

它并不明确指出我们想要启动哪一个活动，而是指定了一系列更为抽象 的action和category，然后交由系统去分析这个intent，并帮我们 找出合适的活动去启动；

在First中：

Intent intent = new Intent(“com.activity.ACTION\_TEST”);

startActivity(intent);

表示启动能够响应“com.activity.ACTION\_TEST”的这个action（没 有加category是因为用的是默认的category），如果我们想要Second 能响应这个intent，我们就需要在Second的<activity>标签中加上：

<activity android:name=”.Second”>

**<intent-filter>**

**<action android:name=”com.activity.ACTION\_TEST”>**

**</action>**

**<category android:name=”android.intent.category.DEFAULT”**

**/>**

**</intent-filter>**

</activity>

这时候，Second才能响应First的这个intent。可以看到，所谓“隐 式”，就是不明确指出目标activity，而是通过action交由系统去分 析，然后去启动合适的活动

（每个Intent中只能指定一个action，但却可以指定多个category）

—隐式Intent相比显式Intent的的*优点*

由于隐式Intent并不需要指定目标class，只需要指定action即可， 而android系统中会有很多*内置动作（action）*，所以当我们想启动 这些内置action时，显式Intent就不能完成我们想要的了。

——活动的生命周期

—返回栈

android中是使用任务（Task）来管理活动的，*一个*任务就是*一组*存放 在栈里的活动的*集合*，这个栈被称作返回栈（Back Stack）。

默认情况下，每当我们启动一个新的活动，它会在返回栈中入栈，并处 于栈顶的位置，每当我们按下Back键或者调用finish()方法去销 毁 一个活动时，处于栈顶的活动就会出栈，这是前一个入栈的活动就会重 新处于栈顶位置。系统总会显示处于栈顶的活动给用户。

只要是同一栈中的Activity，他们的TaskId就一样，因为同一栈中就 是同一任务中，所以，如果两个Activity的TaskId不一样，那么他 们肯定不在一个栈中。

—活动的四种状态、生存期、完整生存期的概念 P56

—活动的启动模式

Standard：活动默认的启动模式，对于使用standard模式启动的活动， 系统不会在乎这个活动是否已经在返回栈中存在，每次启动 都会创建该活动的一个新实例；

singleTop：在启动活动时，如果发现返回栈的*栈顶*已经是该活动，则 认为可以直接使用它，不会再创建新的活动实例；

注意到，只有栈顶是该活动才不会新建，若不在栈顶，还是 会创建新实例；

（因为不是默认的，所以需要在manifest.xml中声明该活 动的启动方式）

singleTask：每次启动活动时，系统首先会在返回栈中检查是否存在该 活动的实例，如果发现已经存在则直接使用该实例，并 把其他在这活动之上的所有活动统统出栈（注意执行的 顺序，线执行该活动的onRestart方法，再执行需要出 栈活动的onDestroy方法），如果没有发现就会创建一 个新的活动实例；

singleInstance：指定为此模式的活动会启用一个新的返回栈来管理 这个活动（如果singleTask模式指定了不同的 taskAffinity，也会启用一个新的返回栈）。

那么这种启动模式有什么意义呢？

它提供了同一个返回栈，解决了不同应用程序间共 享活动实例的问题。

（如果依次打开A、B、C三个活动，B活动是以singleInstance 模式启动的，A、C的启动模式其他三种，那么在此时（C处 在栈顶时），点击Back键，会直接返回到A活动。因为AC 在一个任务栈中，B在另一个栈中。当再次按下Back键时， 会调到B活动界面，因为AC的任务栈空了，就会显示另一个 任务栈栈顶的活动，也就是B。）

——零碎知识点

[—在XML文件中引用资源：@String/\*\*,](mailto:—在XML文件中引用资源，@String/**,表示引用String.xml文件中的**)

[表示引用String.xml文件中的\*\*](mailto:—在XML文件中引用资源，@String/**,表示引用String.xml文件中的**)代表的（字符串）值；

在引用id时也是一样，@id/\*\*,

表示引用id.xml中\*\*代表的（字符串）值；

—当在XML文件中定义id时：@+id/\*\*,

中间多了个“+”号，\*\*标识这个id对应的名字；

（任何资源都会在R文件中生成一个相应的资源id）

—活动被回收了怎么办

了解其原因，带来的影响，以及怎么解决 P66

—在代码中动态更改ImageView中的图片

ImageView.setImageResource()方法

—为控件注册监听的两种方式

使用匿名类；

实现监听接口（然后重写里面的监听方法）。

—布局

它是一种容器，在其内部不仅可以放置控件还可以放置布局

—LinearLayout中android:layout\_weight的使用

—android:gravity用于指定文字在控件中的对齐方式

android:layout\_gravity用于指定控件在布局中的对齐方式

—当一个控件去引用另一个控件的id时，该控件一定要定义在引用控件的 后面，不然会出现找不到id的情况。

—TableRow中的控件是不能指定宽度的

可以通过android:layout\_span=””和android:stretchColumns来 解决显示上的问题

——奇巧淫技（开发当中的一些小技巧）

—知晓当前是在哪一个活动

自定义BaseActivity继承自Activity，重写onCreate()方法，在此 方法中加上：

Log.d(“BaseActivity”,getClass().getSimpleName);

之后当我们在写activity时，让他们继承BaseActivity，这样就能 在日志中看到当前处于哪个活动中了。  
 （这个BaseActivity是不需要再AndroidManifest.xml中注册的， 为什么呢？

我的想法是，它虽然是个activity，但是它是不需要被启动的，就像 Activity类一样，只是用于被继承，不需要被启动。）

—随时随地退出程序 P78

新建一个ActivityCollector类作为活动管理器，定义一个List：

public static List<Activity> activities

= new ArrayList<Activity>();

定义三个方法：addActivity、removeActivity和finishAll，

其实本质上，是将Android中Activity的生存期中的onCreate()与 List中的add()方法对应，将onDestroy()方法与List中的remove() 方法对应。

当我们新建一个活动时，就将它添加（add）到这个List中，销毁时将 将它从 这个List中remove掉（这两个功能可以再BaseActivity中 添加，这样就不用一一添加了，只要后面的活动继承了这个 BaseActivity就行了），需要退出程序时，直接将这个List 实例循 环finish掉即可。

—启动活动的最佳写法 P80

如果要在FirstActivity中启动SecondActivity，而启动后者需要两 个参数，抛开传统的方法不谈，我们来说说最佳写法：

在SecondActivity中添加actionStart()方法，

public class SecondActivity extends BaseActivity(){

Public static void actionStart(Context context,String data1,String data2){

Intent intent = new Intent(context,SecondActivity.class)

intent.putExtras(“param1”,data1);

intent.putExtras(“param2”,data2);

context.startActivity(intent);

}

}

可以看到，SecondActivity所需要的数据全部都在方法参数中体现了 出来，

当我们需要在FirstActivity中启动SecondActivity时，直接这样 写：

SecondActivity.actionStart(FirstActivity.this,”data1” , ”data2”);

可以看到，只要一行代码就能启动了。

（需要注意的是：此时虽然成功启动了SecondActivity了，但如果这 两个数据在接下来的SecondActivity的生存期中还有用的话，我们还 要在SecondActivity的onCreate()方法 中写上如下代码：

Intent intent = getIntent();

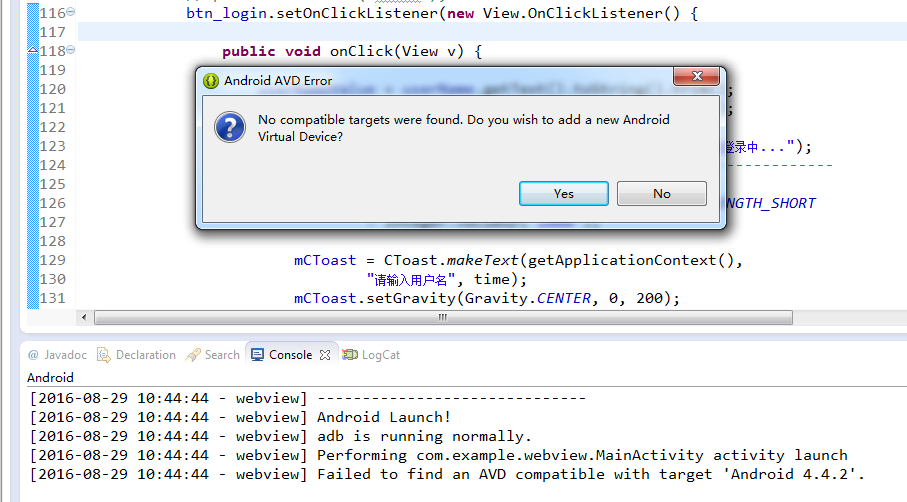
String s = intent.getExtrasString(“param1”);

这样写的目的，是取得从FirstActivity中传过来的值，之前虽然将值传到了SecondActivity的actionStart()方法中，但这个方法并不是Activity类的生存期方法，只要通过Intent将值传到onCreate方法中，才能在此活动之后的生存期中使用这些数据。由此也可知道，intent不仅可以用于不同活动间传值，还有一用于同一活动，不同方法间的传值。）

我们要习惯为每个活动都编写类似的启动方法，这样不仅可以让启动活 动变得简单，还可以节省同事来询问你的时间。

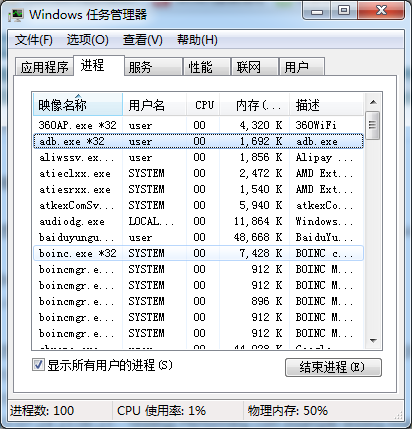
# 2016-8-29

## Run as -->android application 时出现问题



解决办法：

关掉adb.exe



使用豌豆荚，就可以进行安装调试了

## 修改adb端口

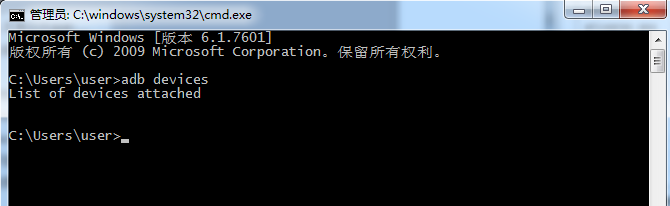
Android开发中，有时会出现端口被占用的情况，这里我们可以修改它的端口来解决这样的问题，方法也很简单：

新建一个环境变量，名字为ANDROID\_ADB\_SERVER\_PORT，然后它的值设为一个没被其他程序占用的新的端口

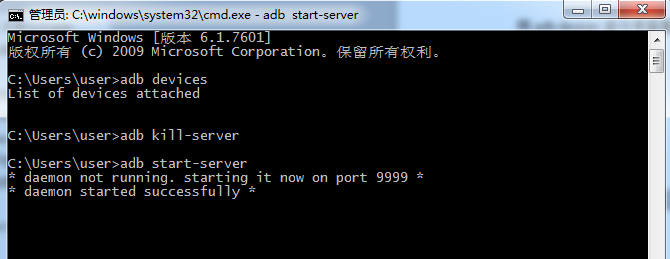


这里我们设置的是9999（原来端口是5037），接下来我们在DOS窗口（win系统）下来查看：

用adb devices命令来查看，

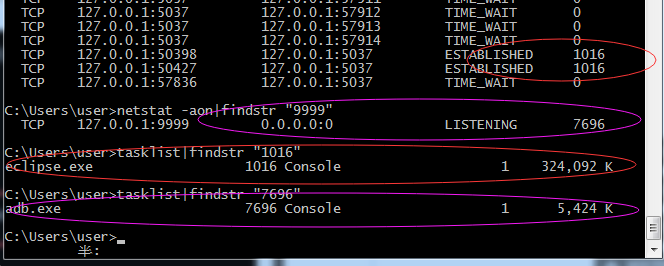


没有什么显示，接下来我们可以：



先kill在start，就可以发现这个adb在端口9999上运行了。

但是问题还没有彻底解决，当我们用netstat -aon|findstr “5037”等命令查询的时候，你会发下如下图所示情形：



上面的红框是关于端口5037的，1016表示占用这个端口的进程号，然后我们用

Tasklist|findstr “1016”来查看跟这个进程号对应的进程时，会发现其实是eclipse占用了，而9999这个端口号查询出来其实是被adb这个程序给占用了，所以当我们使用eclipse来

Run app时还是使用的5037这个端口号，这个时候我们可以尝试着使用“重启大法”，将电脑重启后，在重复上面的步骤查询，会发现

使用9999的端口是 adb.exe

使用5037的端口是 豌豆荚

但是由于项目中使用的测试手机的机型问题，这个手机必须要通过手机助手之类的软件才能连上电脑调试，例如使用豌豆荚。虽然我们成功改了adb的端口，但是这款手机自身并不能通过adb和电脑形成连接（如果可以就不会出现这么多问题），而是需要通过豌豆荚之类的手机助手在通过adb然后再和电脑连接（由通过手机助手能连上可知，手机助手中的adb与电脑中的adb的端口号一致，只有一致才能连上。但是一致也会导致一个问题：就是当我们debug项目的时候，界面会卡在waiting for debugger to attach这不动），但是由于adb与豌豆荚的端口号现在不一致了，所以会发现，现在还是连接不上电脑。

总结：机型问题。

## Android网络通信框架值Volley

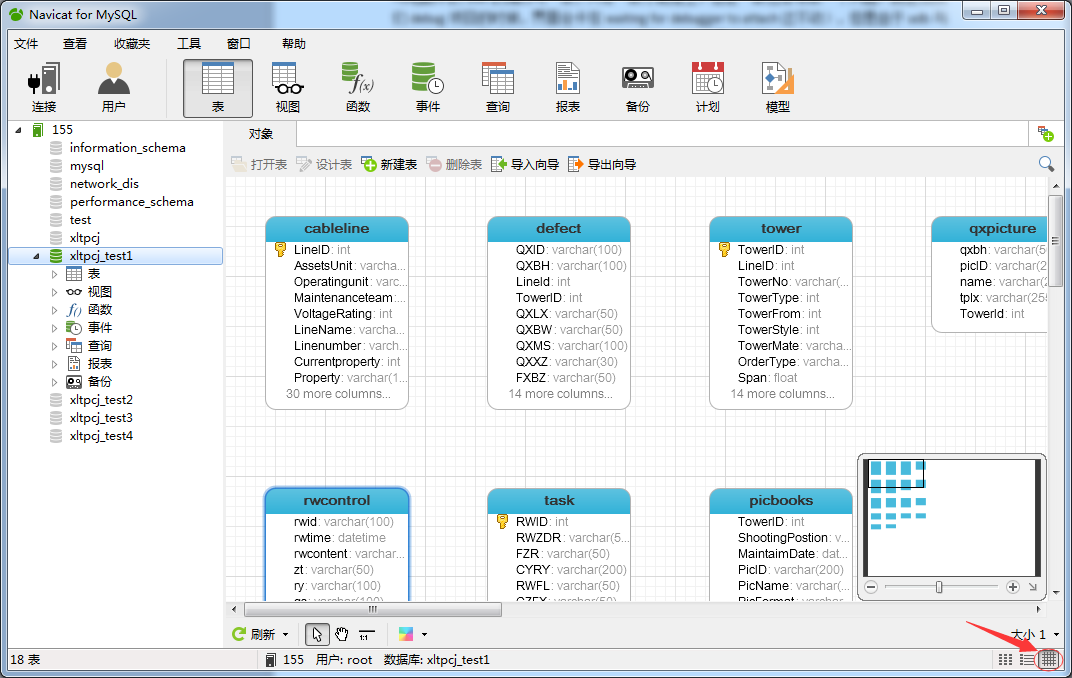
除了简单易用之外，Volley在性能方面也进行了大幅度的调整，它的设计目标就是非常适合去进行数据量不大，但通信频繁的网络操作，而对于大数据量的网络操作，比如说下载文件等，Volley的表现就会非常糟糕。

<http://www.apihome.cn/view-detail-70211.html>

## Navicat for MySQL的一些使用技巧

查看数据库中表的E-R图：

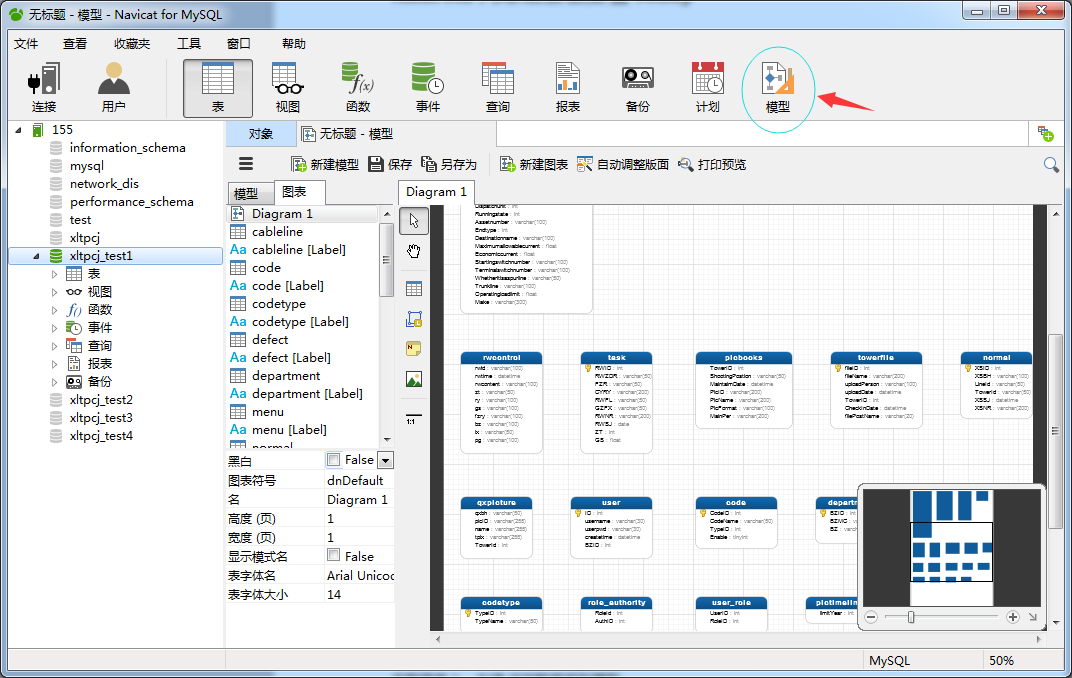
打开数据库时，在Navicat for MySQL的界面右下角有个图标，点击即可：



查看数据库中表的表间关系：

利用Navicat for MySQL的“模型”功能（旧版的Navicat for MySQL中没有这一功能）：

在数据库上，右键-逆向数据库到模型：



要带有绿圈圈中的选项才能使用此功能（由于此数据库中的表没有设定主外键，所以表间没有连线）。

那么我们可以利用这个“模型”功能做些什么呢？如下：

<https://www.navicat.com.cn/datamodeling>

<http://blog.sina.com.cn/s/blog_134b5dd300102v4gf.html>

## Android中默认隐藏软键盘

打开一个app时，一般需要登录，有两个输入框需要编辑，但是通常一打开就输入框就会默认获取焦点从而弹出软键盘，使得用户体验效果不好，现在想让它默认不弹出软键盘：

做法是：

在这个androidmanifest.xml中给这个activity添加相应属性：

android:windowSoftInputMode=*"stateAlwaysHidden"*

其中，*stateAlwaysHidden*的意思是：

当该Activity主窗口获取焦点时，软键盘总是被隐藏的。

关于android:windowSoftInputMode这个属性，其实是Android1.5后的一个新特性， 是关于“activity主窗口与软键盘的交互模式”的属性：

它的另外几个属性值如下：

——stateUnspecified：软键盘的状态并没有指定，系统将选择一个合适的状态或依赖于主题的设置

——stateUnchanged：当这个activity出现时，软键盘将一直保持在上一个activity里的状态，无论是隐藏还是显示

——stateHidden：用户选择activity时，软键盘总是被隐藏

——stateAlwaysHidden：当该Activity主窗口获取焦点时，软键盘也总是被隐藏的

——stateVisible：软键盘通常是可见的

——stateAlwaysVisible：用户选择activity时，软键盘总是显示的状态

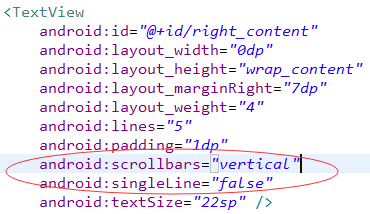
——adjustUnspecified：默认设置，通常由系统自行决定是隐藏还是显示

——adjustResize：该Activity总是调整屏幕的大小以便留出软键盘的空间

——adjustPan：当前窗口的内容将自动移动以便当前焦点从不被键盘覆盖和用户能总是看到输入内容的部分

## Android中TextView控件内容过多时，实现可滑动查看

在layout.xml中为TextView加上如下属性：



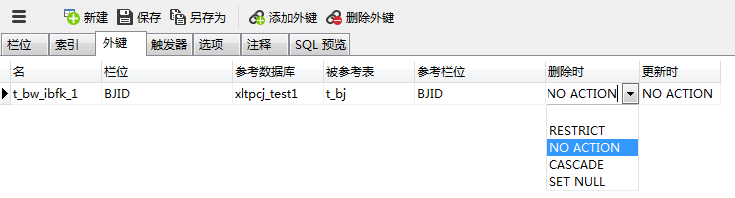
这只是工作的一半，接下来还要在java代码中添加一行：



在找到这个控件的实例后，为其添加setMovementMethod()方法，这样，当此TextView中的内容过多时，便可以实现垂直方向的滑动查看。

## MySQL添加外键约束时的级联操作

今天在用Navicat for Mysql给Mysql中的表添加外键时，发现要进行如下选择：



这个其实就是mysql中的级联操作（此时是添加外键时进行的级联操作），在解释这几个级联操作选项之前，我们先来说一点

必要的“前戏”知识：

——MySQL目前的常用引擎类型有两种：MyISAM和InnoDB；

——MyISAM不支持外键约束，InnoDB支持外键约束；

那么，什么情况下可以使用外键呢？

——两张表必须都是InnoDB表，并且它们没有临时表；

——建立外键关系的对应列必须具有相似的InnoDB内部数据类型；

——建立外键关系的对应列必须建立了索引；

——假如显式的给出了CONSTRAINT symbol，那symbol在数据库中必须是唯一的。假如没 有显式的给出，InnoDB会自动的创建；

那么，在什么情况下需要使用外键呢？

——父表更新时子表也更新，父表删除时如果子表有匹配的项，删除失败；

——父表更新时子表也更新，父表删除时子表匹配的项也删除。

前一种情况，在外键定义中，我们使用ON UPDATE CASCADE ON DELETE RESTRICT；

后一种情况，可以使用ON UPDATE CASCADE ON DELETE CASCADE。

上图中既然可以设置外键，就说明它使用的引擎类型是InnoDB，而InnoDB支持的级联操作主要有四个：

RESTRICT | CASCADE | SET NULL | NO ACTION

下面我们来分别解释这四个选项的含义：

——cascade方式

在父表上update/delete记录时，同步update/delete掉子表的匹配记录

——set null方式

在父表上update/delete记录时，将子表上匹配记录的列设为null

要注意子表的外键列不能为not null

——No action方式

如果子表中有匹配的记录,则不允许对父表对应候选键进行update/delete操作

——Restrict方式

同no action, 都是立即检查外键约束

（——Set default方式

父表有变更时,子表将外键列设置成一个默认的值 但Innodb不能识别）

# 2016-9-5

## SQL之where、group by、having、order by

今天在写sql语句的时候，发现where条件必须要写在order by之前，不然就提示错误，现在来将这几个sql关键字来进行一个总结：

当一个查询语句同时出现了where,group by,having,order by的时候，执行顺序和编写顺序是：

——执行where xx对全表数据做筛选，返回第1个结果集；

——针对第1个结果集使用group by分组，返回第2个结果集；

——针对第2个结果集中的每1组数据执行select xx，有几组就执行几次，返回第3个结果集；

——针对第3个结果集执行having xx进行筛选，返回第4个结果集；

——针对第4个结果集排序；

（http://blog.csdn.net/superhosts/article/details/39298529）

上面提到的“执行顺序和编写顺序”，我认为应该仅仅只是编写顺序，因为SQL语句真正的执行顺序是和编写顺序无关的

where,group by,having,order by间的区别：

——order by是按字段进行排序，字段后面可跟desc降序或asc升序，默认为升序；

——group by是进行分组查询；

——having和where都属于条件过滤，区别在于：

一般having是和 group by 连用，目的是分组后进行的条件查询，而如果在group by前面有where则是表示先条件过滤再分组，这个在实际中特殊的查询会影响到查询结果

需要注意having和where的用法区别：

——having只能用在group by之后，对分组后的结果进行筛选(即使用having的前提条件是分组)；

——where肯定在group by 之前，即也在having之前；

——where后的条件表达式里不允许使用聚合函数（count(),sum(),avg(),max(),min()），而having可以；

## 动态语言与静态语言的一点区别

(http://www.jianshu.com/p/cd1426ec875b)

今天在看Gradle的文章时，里面提到了groovy，顺带也说了一下：

Groovy是动态语言，而比较常用的java则是静态语言。

那么，这两者之间的区别是什么呢？

Groovy中支持动态类型，*这是动态语言和静态语言比较重要的区别。*

即定义变量的时候可以不指定其类型，同一个变量可以反复赋值，而且可以是不同类型的变量。例如：

def test=2

test='tt'

println "${test}"

可以看到，定义test时没有指定类型，先给其赋值2（整型），然后又给其赋值tt（字符串型），最后打印出来的结果是：tt

这种语法早在别的动态语言，如Python、Ruby中出现过了。

既然说到了Groovy，那我们就多说一点，在说一下其中比较重要的一点——闭包（Closure）

闭包（Closure）是Groovy中*函数式编程*的最好体现。这里提到的函数式编程其实就是如何编写程序的方法论，和一些我们已经熟识的方法论一样，比如：

——面向过程编程（C、）

——面向对象编程（C++、java）

——函数式编程（Groovy、）

## Android中的性能优化

关于android中的性能优化，我们大概可以从以下*5个方面*去了解和进行：

—— 布局复杂度：布局复杂会导致布局需要更长的时间，从而导致进入应用慢、页面切换 慢；

—— 耗电量：耗电量大会导致机器发热、缩短机器的有效使用时长；

—— 内存：内存消耗大会导致频繁GC，GC时会暂停其它工作，导致页面卡顿；内存泄露 会导致剩余可用内存越来越小；内存不足会导致应用异常；

—— 网络：频繁的网络访问会导致耗电和影响应用的性能；网络交互数据大小会影响网络 传输的效率；

—— 程序执行效率：糟糕的代码会严重影响程序的运行效率，UI线程过多的任务会阻塞应 用的正常运行，长时间持有某个对象会导致潜在的内存泄露，频繁的 IO操作、网络操作而不用缓存会严重影响程序的运行效率。

详细点的，可以去看这篇文章：http://www.jianshu.com/p/be05874965d4

## litepal的优点

——建表时

LitePal根据类中的字段和相关的简单配置自动帮我们建表；

——升级表时

升级表不需要编写（大量的）SQL，只需简单的配置、改动即可

——表关联时

使用LitePal来自动建立表关联又是一个非常不错的选择，我们不需要关心什么外键、中间表等实现的细节，只需要在对象中声明好它们相互之间的引用关系，LitePal就会自动在数据库表之间建立好相应的关联关系了；

# 2016-9-12

## Android中的动画

（http://blog.csdn.net/guolin\_blog/article/details/43536355）

*（上）*

逐帧动画（frame-by-frame animation）：将一个完整的动画拆分成一张张单独的图片， 然后再将它们连贯起来进行播放，类似于动画片的工作原理；

补间动画（tweened animation）：对View进行一系列的动画操作，包括淡入淡出、缩 放、平移、旋转四种；

属性动画（property animation）：功能非常强大，弥补了之前补间动画的一些缺陷，几 乎是可以完全替代掉补间动画；

补间动画的局限：

——只能对“View”进行操作；

——只能通过它实现移动、缩放、旋转和淡入淡出这四种动画效果，基本没有任何扩展性可 言；

——它只改变了View的显示效果，并没有真正的改变View的属性，比如屏幕右上方有一个 按钮包含了点击事件，当我们用补间动画将它移到其他位置，点击事件是不会触发的， 因为实际上这个按钮还是停留在屏幕的左上角，只不过补间动画将这个按钮绘制到了屏 幕的右下角而已；

属性动画：

——属性动画机制已经不再是针对于View来设计的了，也不限定于只能实现移动、缩放、 旋转和淡入淡出这几种动画操作，同时也不再只是一种视觉上的动画效果了；

——它实际上是一种不断地对值进行操作的机制，并将值赋值到指定对象的指定属性上，可 以是任意对象的任意属性；

——所以我们仍然可以将一个View进行移动或者缩放，但同时也可以对自定义View中的 Point对象进行动画操作了。我们只需要告诉系统动画的运行时长，需要执行哪种类型 的动画，以及动画的初始值和结束值，剩下的工作就可以全部交给系统去完成了；

——既然属性动画的实现机制是通过对目标对象进行赋值并修改其属性来实现的，那么之前 所说的按钮显示的问题也就不复存在了，如果我们通过属性动画来移动一个按钮，那么 这个按钮就是真正的移动了，而不再是仅仅在另外一个位置绘制了而已；

属性动画的详细介绍：

**ValueAnimation**

它是整个属性动画机制中最核心的一个类，它负责：

——管理动画的播放次数、播放模式；

——对动画设置监听器；

——计算初始值和结束值之间的动画过渡

（属性动画的运行机制是通过不断地对值进行操作来实现的，而初始值和结束值之间的 动画过渡就是由ValueAnimator这个类来负责计算的。它的内部使用一种时间循环的机 制来计算值与值之间的动画过渡，我们只需要将初始值和结束值提供给ValueAnimator， 并且告诉它动画所需运行的时长，那么ValueAnimator就会自动帮我们完成从初始值平 滑地过渡到结束值这样的效果）；

**ObjectAnimator**

——相比于ValueAnimator，ObjectAnimator可能才是我们最常接触到的类，因为 ValueAnimator只不过是对值进行了一个平滑的动画过渡，但我们实际使用到这种功能 的场景好像并不多。而ObjectAnimator则就不同了，它是可以直接对任意对象的任意属 性进行动画操作的，比如说View的alpha属性；

——不过虽说ObjectAnimator会更加常用一些，但是它其实是继承自ValueAnimator的，底 层的动画实现机制也是基于ValueAnimator来完成的，因此ValueAnimator仍然是整个 属性动画当中最核心的一个类。那么既然是继承关系，说明ValueAnimator中可以使用 的方法在ObjectAnimator中也是可以正常使用的，它们的用法也非常类似；

**Animator**

——独立的动画能够实现的视觉效果毕竟是相当有限的，因此将多个动画组合到一起播放就 显得尤为重要；

——实现组合动画功能主要需要借助AnimatorSet这个类；

**AnimatorListener**

——Animator监听器，在很多时候，我们希望可以监听到动画的各种事件，比如动画何时开 始，何时结束，然后在开始或者结束的时候去执行一些逻辑处理；

——这个功能是完全可以实现的，Animator类当中提供了一个addListener()方法，这个方法 接收一个AnimatorListener，我们只需要去实现这个AnimatorListener就可以监听动画的 各种事件了；

——ObjectAnimator是继承自ValueAnimator的，而ValueAnimator又是继承自Animator的， 因此不管是ValueAnimator还是ObjectAnimator都是可以使用addListener()这个方法的。 另外AnimatorSet也是继承自Animator的，因此addListener()这个方法算是个通用的方 法；

*（中）*

**ValueAnimator的高级用法**

——通过对对象进行值操作来实现动画效果的功能，这就是ValueAnimator的高级用法；

**ObjectAnimator的高级用法**

## 几款常用的开发工具

——Velocity：Windows下类似Mac上Dash的文档工具；

（http://velocity.silverlakesoftware.com/）

——Sublime Text：查看源码时的阅读工具，也可以用来编码；

——Genymotion：Android开发最快模拟器，比原生的模拟器快很多；

——LICEcap：GIF屏幕录制工具，灵活好用；

## Android中的adapter

*为什么要用adapter*

为了解决复杂数据在界面上的显示问题——我们知道，一般数据都是直接在layout中绑定，但是当数据复杂或者数据是动态生成的时候直接绑定就会显得很无力，此时就需要adapter作为一个载体。

## Android图片压缩质量参数

（http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2014/1023/1825.html）

ARGB\_8888、ALPHA\_8、ARGB\_4444、RGB\_565

A（Alpha）：透明度、 R（red）：红色 、G（green）：绿、 B（blue）：蓝

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

——Bitmap.Config ARGB\_8888：每个像素占四位，即A=8，R=8，G=8，B=8，那么一个像素 点占8+8+8+8=32位；

——Bitmap.Config RGB\_565：每个像素占四位，即R=5，G=6，B=5，没有透明度，那么一个 像素点占5+6+5=16位；

——Bitmap.Config ALPHA\_8：每个像素占四位，只有透明度，没有颜色；

一般情况下我们都是使用的ARGB\_8888，而它也是最占内存的，因为一个像素占32位，8位=1字节，所以一个像素占4字节的内存。假设有一张480x800的图片，如果格式为ARGB\_8888，那么将会占用1500KB的内存。

# 2016-9-18

## Android中的反编译技术

（http://blog.csdn.net/guolin\_blog/article/details/49738023）

***反编译代码***

要想将APK文件中的代码反编译出来，我们需要用到以下两款工具：

——dex2jar 这个工具用于将dex文件转换成jar文件

下载地址：http://sourceforge.net/projects/dex2jar/files/

——jd-gui 这个工具用于将jar文件转换成java代码

下载地址：<http://jd.benow.ca/>

***反编译资源***

——apktool.jar & apktool.bat

## Android中的沉浸式状态栏

***定义***

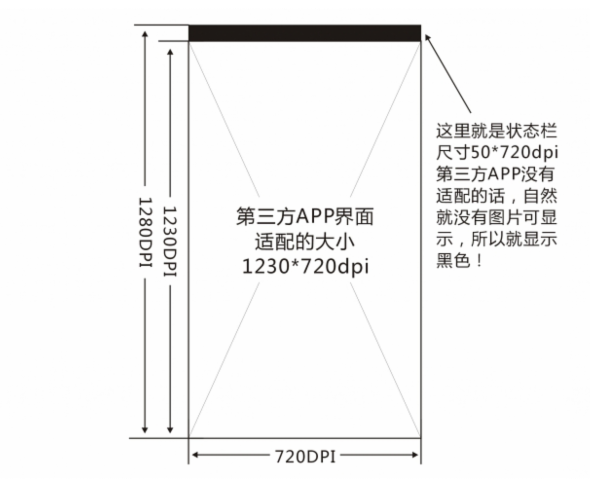
沉浸式状态栏：准确来说应该是“透明栏”，英文名“Translucent Bars”，是Android 4.4 新定义的设计规范。简单来说就是在软件打开的时候通知栏和软件顶部颜色融为一体，这样可以使软件和系统本身更加融为一体，同时通知栏的颜色不再是白色、黑色简单的两种了。

***说明***

其实，Android官方从来没有给出过沉浸式状态栏这样的命名，只有沉浸式模式（Immersive Mode）这种说法。只是有些人在没有完全了解清楚沉浸模式到底是什么东西的情况下，就张冠李戴地认为一些系统提供的状态栏操作就是沉浸式的，并且还起了一个沉浸式状态栏的名字。

***解释***

那么，沉浸式状态栏是该由系统适配软件呢，还是由软件适配系统呢？



引用上图来说明，在软件开发中，如果软件没有对状态栏做了沉浸式处理，状态栏自然是黑色。如果由系统来强制抓取软件顶栏颜色从而实现“假沉浸式”的效果，在一些颜色比较复杂的软件中就会出现无法判断颜色、颜色撕裂等视觉问题，因此，沉浸式状态栏由第三方软件来主动适配，不管是在视觉上会更完美，在内容突出和体验上也有很大提升。

***真正的沉浸式模式***  
虽说沉浸式导航栏这个东西是被很多人误叫的一种称呼，但沉浸式模式的确是存在的。

——首先你应该确定自己是否真的需要这个功能，因为除了像游戏或者视频软件这类特殊的 应用，大多数的应用程序都是用不到沉浸式模式的。

——当你确定要使用沉浸式模式，那么只需要重写Activity的onWindowFocusChanged()方法， 然后加入如下逻辑即可：

public class MainActivity extends AppCompatActivity {

...

@Override

public void onWindowFocusChanged(boolean hasFocus) {

super.onWindowFocusChanged(hasFocus);

if (hasFocus && Build.VERSION.SDK\_INT >= 19) {

View decorView = getWindow().getDecorView();

decorView.setSystemUiVisibility(

View.SYSTEM\_UI\_FLAG\_LAYOUT\_STABLE

| View.SYSTEM\_UI\_FLAG\_LAYOUT\_HIDE\_NAVIGATION

| View.SYSTEM\_UI\_FLAG\_LAYOUT\_FULLSCREEN

| View.SYSTEM\_UI\_FLAG\_HIDE\_NAVIGATION

| View.SYSTEM\_UI\_FLAG\_FULLSCREEN

| View.SYSTEM\_UI\_FLAG\_IMMERSIVE\_STICKY);

}

}

}

沉浸式模式的UI Flag就这些，也没什么好解释的，如果你需要实现沉浸式模式，直接将上面的代码复制过去就行了。需要注意的是，只有在Android 4.4及以上系统才支持沉浸式模式，因此这里也是加入了if判断。

另外，为了让我们的界面看上去更像是游戏，这里我将MainActivity设置成了横屏模式：

<activity android:name=".MainActivity"

android:screenOrientation="landscape">

...

activity>

***总结***

Android沉浸式模式的本质就是全屏化。

（沉浸式状态栏实现：<http://chuansong.me/n/548108251837）>

## Android中Toast使用的小技巧

我们平时使用Toast，最常用基本写法可能就是如下的写法：

Toast.*makeText*(con text, "message", Toast.*LENGTH\_SHORT*).show();

可以看到，Toast的基本用法很简单，但是也不是说没有一点深度。例如当我们按照上面的写法来运行时，就会出现：

<http://chuansong.me/n/461738551755>

这样的情况，即我连续点击5次，Toast就会触发5次，这样的话用户体验的效果会很不好。

因此，最佳的做法是将Toast的调用封装在一个工具类中，在工具类中进行逻辑判断，如下：

public class Util {

private static Toast toast;

public static void showToast(Context context,

String content) {

if (toast == null) {

toast = Toast.makeText(context,

content,

Toast.LENGTH\_SHORT);

} else {

toast.setText(content);

}

toast.show();

}

}

这里和我们平时使用Toast的方式并不一样，这里会先判断Toast对象是否为空，如果是空的情况下才会调用makeText()方法来去生成一个Toast对象，否则就直接调用setText()方法来设置显示的内容，最后再调用show()方法将Toast显示出来。

由于不会每次调用的时候都生成新的Toast对象，因此刚才我们遇到的问题在这里就不会出现了。调用的时候也很简单，只需要把Context对象和Toast要显示的内容传进来就可以了。

Util.showToast(context, "things happened");

现在不管我们触发多少次Toast调用，都只会持续一次Toast显示的时长了。

## Android提示信息之Snackbar的使用

***说明***

*(http://chuansong.me/n/461738551755)*

如果说Dialog和Toast是两个极端的话，那么Snackbar就是处于中间的位置了。Snackbar和Toast比较相似，但是用途更加广泛，并且它是可以和用户进行交互的。Snackbar使用一个动画效果从屏幕的底部弹出来，过一段时间后也会自动消失。

***用法***

在使用Snackbar之前，首先需要在app/build.gradle中添加相应的依赖：

dependencies {

compile 'com.android.support:design:23.4.0'

}

然后就可以使用Snackbar了，它的用法和Toast是比较相似的：

Snackbar.make(view, "data deleted",

Snackbar.LENGTH\_LONG)

.setAction("Undo",

new View.OnClickListener(){

@Override

public void onClick(View v) {

// snackbar.dismiss();

}

})

.show();

这里调用Snackbar的make()方法来创建一个Snackbar对象，make()方法的第一个参数需要传入一个view，只要是当前界面布局的任意一个view都可以，Snackbar会使用这个view来自动查找最外层的布局，用于展示Snackbar。第二个参数就是Snackbar中显示的内容，第三个参数是Snackbar显示的时长。这些和Toast都是类似的。

同时，要想支持Swipe（轻扫）手势的话,这个view需要是一个CoordinatorLayout(也是在Android Support Library里面的控件)，如下：

*（http://www.jcodecraeer.com/plus/view.php?aid=3187）*

<android.support.design.widget.CoordinatorLayout

        android:id="@+id/container"

        android:layout\_width="match\_parent"

        android:layout\_height="wrap\_content"

        android:layout\_alignParentBottom="true"/>

public class SnackbarTestActivity extends Activity {

CoordinatorLayout container;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_snackbar\_test);

container=(CoordinatorLayout)

findViewById(R.id.container);

}

public void createSnackbar(View v) {

Snackbar.make(container,"SnackbarTest", Snackbar.LENGTH\_LONG).show();

}

}

这样，这个Snackbar就支持右滑退出的功能了

接着这里又调用了一个setAction()方法来设置一个动作，从而让Snackbar不仅仅是一个提示，而是可以和用户进行交互的。最后调用show()方法让Snackbar显示出来。

***总结***

现在我们有三种方式可以给用户提示信息：

Dialog：当提示信息是至关重要的，并且必须要由用户做出决定才能继续的时 候，使用Dialog。

Toast：当提示信息只是告知用户某个事情发生了，用户不需要对这个事情做出 响应的时候，使用Toast。

Snackbar：以上两者之外的任何其他场景，Snackbar可能会是你最好的选择。

***Tips****(http://www.wtoutiao.com/p/h6dlCN.html)*

——使用Android Support Design Library可能会与之前使用 NavigationDrawerLibrary开源库产生冲突；

— —make()方法的第一个参数的view,不能是有一个ScrollView。

因为SnackBar的实现逻辑是往这个View去addView.而ScrollView我们 知道,是只能有一个Child的.否则会Exception;

— —如果大家在想把Toast替换成SnackBar.需要注意的是,Toast和 SnackBar的区别是,前者是悬浮在所有布局之上的包括键盘,而SnackBar 是在View上直接addView的;

所以SnackBar.show()的时候,要注意先把Keyboard.hide()了.不然,键 盘就会遮住SnackBar;

— —在Android2.3,SnackBar即使用了CoordinatorLayout也是不支持 Swipe功能;

— —在Android2.3,SnackBar的深色背景颜色和字体颜色相近.可以用 SpannableString换一下String的颜色在传给make()方法;

## Android中的各种bar——StatusBar、TitleBar、ActionBar、ToolBar

StatusBar:状态栏，它处于屏幕的最顶部，正常情况下它是显示的，它和 TitleBar和ActionBar之间没有直接的关系；

TitleBar:标题栏,它紧挨状态栏的下面，正常情况下它的布局和主题样式都是 使用系统定义好的，且默认情况下只显示图标和文本;

ActionBar：操作栏，它是Android3.0以后引入SDK的，通过它去**取代**之前标 题栏，它和标题栏一样紧挨状态栏的下面；

ToolBar：Toolbar是在 Android 5.0 开始推出的一个 Material Design 风 格的导航控件 ，来**取代**之前的 Actionbar ；

NotificationBar：通知栏；

SystemBar：上面提到的所有Bar都属于SystemBar（应该是这样的）。

*ActionBar与TitleBar的区别与联系*

ActionBar和TitleBar是同一个位置的，都是在状态栏下，ActionBar 上面可以放入按钮,或下拉式的按钮，可以有文字,logo等信息，还可以设置返回按钮等信息；

TitleBar只有个app图片和文字。

*ToolBar与ActionBar的区别与联系*

与 Actionbar 相比， Toolbar 明显要灵活的多。它不像 Actionbar 一样，一定要固定在Activity的顶部，而是可以放到界面的任意位置。除此之外，在设计 Toolbar 的时候，Google也留给了开发者很多可定制修改的余地，这些可定制修改的属性在API文档中都有详细介绍

## Android Toolbar初识

*(http://www.codeceo.com/article/android-toolbar-develop.html)*

*(http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2014/1118/2006.html)*

*(http://blog.csdn.net/jdsjlzx/article/details/41441083/)*

*(http://www.imooc.com/article/2844)*

Toolbar是在 Android 5.0 开始推出的一个 Material Design 风格的导航控件 ，Google 非常推荐大家使用 Toolbar 来作为Android客户端的导航栏，以此来取代之前的 Actionbar 。

与 Actionbar 相比， Toolbar 明显要灵活的多。它不像 Actionbar 一样，一定要固定在Activity的顶部，而是可以放到界面的任意位置。除此之外，在设计 Toolbar 的时候，Google也留给了开发者很多可定制修改的余地，这些可定制修改的属性在API文档中都有详细介绍，如：

——设置导航栏图标；

——设置App的logo；

——支持设置标题和子标题；

——支持添加一个或多个的自定义控件；

——支持Action Menu；

## 介绍Toolbar之前先简单了解下Actionbar

*<http://blog.csdn.net/guolin_blog/article/details/18234477>*

*<http://www.tuicool.com/articles/ERvAfyn>*

*<http://www.2cto.com/kf/201604/497523.html>*

*<http://blog.csdn.net/bigconvience/article/details/28886241>*

## 为什么能在子线程中更新UI线程的ProgressBar的进度条

***说明***

android中更新UI只能用UI线程，要在子线程中更新主线程的UI，一般是通过消息机制——message和handler结合的方式。但是有时我们会发现ProgressBar更新进度条时，却可以在子线程刷新，如下：

new Thread() {

@Override

public void run() {

try {

while (count <= 100) {

pg.**setProgress**(count++);

Thread.sleep(100);

} }catch (InterruptedException e) {

e.printStackTrace();

}

}

}.start();

上面的代码，就展示了在子线程中通过setProgress()方法来更新UI线程中的ProgressBar的进度条

原因

在查看setProgress的源码后，我们发现它回去调用refreshProgress()这个方法，这个方法的源码如下：

private synchronized void refreshProgress(int id, int progress, boolean fromTouch) {

if (mUiThreadId == Thread.currentThread().getId()) {

doRefreshProgress(id, progress, fromTouch);

} else {

RefreshProgressRunnable r;

if (mRefreshProgressRunnable != null) {

r = mRefreshProgressRunnable;

mRefreshProgressRunnable = null;

r.setup(id, progress, fromTouch);

} else {

r = new RefreshProgressRunnable(id, progress, fromTouch);

}

post(r);

}

}

在这个方法中，

首先会判断当前线程是否为UI线程，

——如果是，则直接刷新进度；

——如果不是，

——判断RefreshProgressRunnable是否为空，

——非空，将引用r指向已有的mRefreshProgressRunnable 进行下一 步操作；

——空，创建一个RefreshProgressRunnable对象r；

——调用View的post(Runnable action)方法，通过post(r)方法将

RefreshProgressRunnabled的实例r 放到主UI线程的消息队列等 待处理。

所以更新进度的时候，在不在主线程调用setProgress 方法是没有影响的。

***Tips：***

其实最根本的原因是因为在setProgress()方法中调用的refreshProgress这个方法是线程安全的（有synchronized 修饰），所以即使是在子线程中调用setProgress方法，它最后还是调用refreshProgress这个线程安全的方法，所以就不会因为线程不安全而产生冲突。

## 关于Android中invalidate()方法的思考

***说明***

invalidate()是View中的一个公共方法，用来刷新View的，且必须在UI线程中工作。当我们对某个View的显示进行修改时，调用invalidate()才能看到重新绘制的界面，invalidate()的调用就是把旧得View从UI线程队列中pop掉。

*(http://kalogen.iteye.com/blog/1566111)*

一般情况下（不使用多线程），当我们想在View的显示发生变化时对UI进行重绘。此时，我们只需在Activity中显示的调用View对象中的invalidate()方法即可，系统会自动调用View的onDraw()方法。

***使用注意***

— —Invalidate()不能在非UI线程中调用

Android提供了invalidate()方法实现界面刷新，但是其不能再非UI线程中调用，因为这样违背了单线程模型（关于单线程模型的概念可以看下一个总结）。

## 关于Android是单线程模型以及UI线程是不安全的等说法的讨论

***说明***

一个 Android 程序开始运行时，就有一个主线程Main Thread被创建。该线程主要负责UI界面的显示、更新和控件交互，所以又叫UI Thread。由于只有UI线程更新界面所以说— —Android是单线程模型。

***网上的一些观点***

*（http://bbs.csdn.net/topics/360244795）*

— —为什么UI操作必须在UI线程中调用

因为Android中UI操作是线程不安全的，如果可以再非UI线程中对其进行操作的话，那么我们可以想象这种情况：

*我在一个非UI线程中刷新了界面，同时在UI线程（或者其他非UI线程） 中也刷新界面，这样就会导致同时对一个界面操作而出现问题。*

所以，只能在UI线程中操作UI，是因为UI线程是非线程安全的；

— —为什么UI线程是非线程安全的

可能我们又要问了：为什么UI线程是非线程安全的？

因为Android中的View提供invalidate()方法来更新界面（UI），而 invalidate()这个方法本身是非线程安全的。所以，由此我们就可以说— — Android中的UI线程是非线程安全的。（这个解释感觉有点怪怪的）

***网上的另外一些观点***

— —并不是说UI线程是不安全的，线程没有安全不安全这样的说法。而是更新 UI的方法不是线程安全的，比如我们上面的invalidate()方法就是非线程 安全的，即只能在单线程中完成UI更新，不能使用多线程。

因此，只能在android的UI线程中进行UI更新。

***我的看法***

我们经常见到这样的说法，“Android中UI线程是线程不安全的”，其实这样的说法从Java的角度来说并不正确，因为线程是没有安不安全这样的说法的，只有方法有线程安全或线程不安全的说法。

但是这句话想表达的意思，我想我们多少都有点了解：

— —更新UI只能在UI线程中，子线程是不能更新UI的。

那为什么子线程不能更新UI呢？

因为子线程可能有多个，而如果这些线程同时操作一个控件就有可能会发生冲突，比如：在多个子线程中同时刷新某个控件（View），也就会同时调用invalidate()这个方法，而这个方法在View中的源码如下：

/\*\*

\* Invalidate the whole view. If the view is visible,

\* {@link #onDraw(android.graphics.Canvas)} will be called at some point in

\* the future. This must be called from a UI thread. To call from a non-UI thread,

\* call {@link #postInvalidate()}.

\*/

**public** **void** invalidate() {

invalidate(**true**);

}

可以看到，这是一个非线程安全的方法（没有用synchronized 修饰），因此多个子线程同时调用这个方法，就会冲突。（这个冲突是调用这个方法产生的冲突，还是说在调用这个方法中对控件操作而使这个控件的状态产生冲突？*应该是后者，同时调用一个方法并不会有冲突。*）

而如果在UI线程中更新这个View，就不会产生这样的情况，因为一个运行的app中，只有一个UI线程，也就是同时只有一个线程调用invalidate()方法，所以不管这个方法是否是线程安全的都不会产生冲突。

如果子线程想更新UI，就告诉主线程（UI线程），由UI线程来进行相应的操作。

## 为什么#00FF00是绿色

***说明***

首先说明一下，“#00FF00”这种表达方式是颜色的十六进制表达法，这6为数字分为3组，每组2个，依次代表红、绿、蓝这三原色。

每组的两个数字的大小就代表颜色的深度，所以对于#00FF00：

00：代表红色，但是深度没有，所以可以看为白色；

FF：代表绿色，深度最大，随意是绿色；

00：代表蓝色，深度没有，也堪称白色；

所以，#00FF00就是白色、绿色、白色混合，也就是绿色了。

***解惑***

可能有人有疑问，为什么00是没有深度，而FF是深度最大？

原因是，我们之前说了，“#00FF00”这种方式十六进制的颜色表达法，所以这里的F其实是代表了十进制里面的“15”，所以数字的大小是深度不同的原因，也就可以理解为什么00是没有深度，而FF是深度最大了。（如果没看懂的话，可以先去看一下什么是十六进制）

***补充***

其实颜色处理这种“00FF00”这种十六进制表达法外，我们还经常看到“0~255”这种表达方式，比如绿色的这种表达方式为（0,255,0），不信的话，可以打开你电脑中的“画图”软件，在红、绿、蓝后面的方框中分别输入0、255、0，就可以看到显示的是绿色，如下：



本质上来说，这种颜色的表达方式跟第一种方式是一致的，“0~255”代表颜色的深度，这里的“0~255”的任一数字就代表一个颜色，而十六进制表达法中两个数字才代表一个颜色。

这里的“0~255”其实就是“00~FF”，因为“FF”从十六进制转换为十进制是256，又因为是从0开始计数的，所以就代表“0~255”（从0~255共256个数）。

***Tips***

三基色：红、黄、蓝，一般指的是印刷的三基色；

三原色：红、绿、蓝，一般指的是光的三原色；

光线越加越亮，印刷越糊越暗。

光的三原色是叠加型，色料则是削减型。

***见闻***

我们有时除了这种6位的颜色表达方式：#ffffff,还会见到一种8位的颜色表达方式：#ffffffff,这种方式的表达方式中：

前两位代表：透明度（Alpha），后面依次是：红、绿、蓝

其中，Alpha值中，00是完全透明，ff是完全不透明，比较适中的是1e,Alpha值也可以忽略，如果忽略，那么该颜色默认是不透明的。

Android中的颜色值是通过红（Red）、绿（Green）、蓝（Blue）三原色，以及一个透明度（Alpha）值来表示，颜色值总是以井（＃）开头，接着是Alpha-Red-Green-Blue的形式。

## Android中获取设备的Mac地址

***方法一***

通常我们使用WifiManager来获取，如下：

/\*\*

\* 获取本机Mac地址

\*/

**private** String getMacAddress() {

String mac = "";

// 获取wifi管理器

WifiManager wifiMng = (WifiManager) getSystemService(Context.*WIFI\_SERVICE*);

WifiInfo wifiInfor = wifiMng.getConnectionInfo();

mac = "本机的mac地址是：" + wifiInfor.getMacAddress();

Log.*d*("data", mac);

**return** mac;

}

还需要在Manifest.xml中加上如下权限：

<uses-permission

android:name=*"android.permission.ACCESS\_WIFI\_STATE"*>

</uses-permission>

但是这种方法可能会因为Wifi的关系而出现问题，所以就有了第二种方法

***方法二***

（http://www.cnblogs.com/ziyouchutuwenwu/archive/2013/07/19/3201568.html）

因为android的底层是Linux，所以我们用Linux的方式来获取：

**private** String getMac() {

String macSerial = **null**;

String str = "";

**try** {

Process pp = Runtime.*getRuntime*().exec(

"cat /sys/class/net/wlan0/address ");

InputStreamReader ir = **new**

InputStreamReader(pp.getInputStream());

LineNumberReader input = **new** LineNumberReader(ir);

**for** (; **null** != str;) {

str = input.readLine();

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

macSerial = str.trim();// 去空格

**break**;

}

}

} **catch** (IOException ex) {

// 赋予默认值

ex.printStackTrace();

}

Log.*d*("data", macSerial);

**return** macSerial;

}

***方法三***

调用Linux的busybox，直接用busybox读取系统文件中的MAC地址

public String getMacAddress() {

String result = "";

String Mac = "";

result = callCmd("busybox ifconfig", "HWaddr");

if (result == null) {

return "网络出错，请检查网络";

}

if (result.length() > 0 && result.contains("HWaddr")) {

Mac = result.substring(result.indexOf("HWaddr") + 6, result.length() - 1);

if (Mac.length() > 1) {

result = Mac.toLowerCase();

}

}

return result.trim();

}

## Android获取网络文件大小的坑

URL url = **new** URL

("http://139.129.27.87:8090/GetQuestions.aspx?major=");

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

size = connection.getContentLength();

本来这样写，但是获得的size大小总是-1；

在中间加上一行代码即可：

(http://www.cnblogs.com/helloandroid/archive/2013/01/21/2869776.html)

URL url = **new** URL

("http://139.129.27.87:8090/GetQuestions.aspx?major=");

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

connection.setRequestProperty("Accept-Encoding", "identity");

size = connection.getContentLength();

## Android中对话框Dialog 的dismiss和hide方法区别

***hide()：***

查看dialog.hide()的注释：

Hide the dialog, but do not dismiss it。

说明只是隐藏dialog视图，并没有把视图移走。

在hide()的源码中你会发现：

/\*\*

\* Hide the dialog, but do not dismiss it.

\*/

public void hide() {

if (mDecor != null) {

mDecor.setVisibility(View.GONE);

}

}

它最终是调用了View的GONE属性，把视图设为了GONE，也就是看不见了。

***dismiss()：***

再看dismiss()方法的源码：

void dismissDialog() {

if (mDecor == null || !mShowing) {

return;

}

if (mWindow.isDestroyed()) {

Log.e(TAG, "Tried to dismissDialog() but the Dialog's window was already destroyed!");

return;

}

try {

mWindowManager.removeView(mDecor);

} finally {

if (mActionMode != null) {

mActionMode.finish();

}

mDecor = null;

mWindow.closeAllPanels();

onStop();

mShowing = false;

sendDismissMessage();

}

}

是整个结束掉这个视图，会调用onStop()。

# 2016-9-26

## Android中的技术更新

***ActionBar——>ToolBar****（http://blog.csdn.net/codingismylife/article/details/50814633）*

——自从Toolbar出现以后，Actionbar的命运貌似就只能是慢慢被淘汰了。相比Actionbar， Toolbar更加灵活多变，首先Toolbar并不是Window的一部分，它可以被当作控件来放 在任何地方。与此同时，Toolbar也是在Android设备上实现Material Design特效的一个 重要的工具。

***Eclipse\_ADT——>Android Studio***

——自从去年年中Google官方说明了不再维护Eclipse，Eclipse恐怕真的要慢慢淡出开发者 的视野了。取而代之的Android Studio集成了IntelliJ、Gradle等一些较为新的工具，因 此具有更好的开发体验，在2.0的版本中更是对Emulator做了优化以及增加了动态更改 代码调试的功能。尽管国内很多公司还在使用Eclipse，但Google的这些所作所为使我 们相信Android Studio全面取代Eclipse的那一天并不会非常遥远。

***ListView/GridView——>RecyclerView***

——在Listview和Gridview出生的那一天，有关使用Viewholder模式对其进行优化也就成为 了使用它们的必备知识。Recyclerview对两者进行了整合，将两者的一些不同的特性用 不同的LayoutManager去实现，与此同时加入了许多不同的动画效果。对于初学者而言， Recyclerview的耦合度低下，不是很容易上手。不过这并不影响Recyclerview取代 Listview/Gridview的趋势。

***AsyncTask——>RxJava***

——曾经的Asynctask被普遍运用在网络请求等一些轻量级的异步消息处理中，但是容易造 成内存泄漏也是Asynctask的一个很明显的缺点。Rxjava可以使得异步消息的代码更加 简洁，与此同时程序运行更加安全。尽管Rxjava的学习曲线更加陡峭，于此同时基于 Rxjava的框架仍然没有成熟，但是这也不妨碍其作为一种新的趋势取代Asynctask。

***Activity——>AppCompatActivity***

没错，就是Activity这个类，AppCompatActivity对Activity、FragmentActivity、ActionbarActivity、PreferenceActivity等做了整合。以后Activity以及其一系列的扩展可能都要被AppCompatActivity所取代了。

***HttpURLConnection/HttpClient——>OKHttp、Volley等框架***

——Apache的Http库有着很多的Bug以及版本的兼容问题。与此同时，三方的开源库可以 对其进行很好的取代和缺点的弥补，其中最具代表性的就是Square公司的OkHttp。不 甘落后的Google也推出了自己的Volley库。最终，Apache库在Android6.0中不再作为 默认的SDK。这代表着Android应用开发正在被越来越多的三方库所改进和完善。同时 也在告诉我们，Android应用的开发应当借助开源库去提高开发效率、程序的稳定性、 以及用户体验。

## Android线程间通信方式

*(http://www.tuicool.com/articles/7NZnayB)*

——使用管道流Pipes

——使用共享内存

——使用Handler和Message

——使用AsyncTask

## 使用事务来提高数据的批量写入（数据库）速度

***说明***

在android项目过程中，使用SQLite作为数据库，遇到了需要将大量的数据插入库中，导致很耗时，考虑到可以从数据库的角度来优化，就是采用——事务！

***例子***

SQLite的数据库中默认的插入操作，如果没有用事务的话，它的每次写入操作都会触发一次事务，当有几万条数据时，就会触发几万个事务操作，就造成了时间和资源上的浪费。如果将这几万个操作使用事务的话，就只会进行一次事务操作，节约了大量的时间和资源，代码如下：

db.beginTransaction();

try{

//批量处理操作（这里可能是一个进行几万次的循环操作）

db.execSQL("SQL语句", new Object[]{});

//设置事务标志为成功，当结束事务时就会提交事务

db.setTransactionSuccessful();

}

catch(Exception e){

}

finally{

//结束事务

db.endTransaction();

}

***补充***

上面事务的使用，是用在SQLite的写入操作中，那么当我们用在update或者其他的操作中，是否还会节约时间呢？（主要看update或者其他操作是否默认每一次都会使用事务）

其他数据库（如MySql...）的写入操作是否也可以用事务来节省大批量的操作时间呢？

## RxJava

*（http://gank.io/post/560e15be2dca930e00da1083#toc\_1）*

***定义***

RxJava是ReactiveX中使用*Java语言*实现的版本（还有RxJS、RxRuby...）。

RxJava 在 GitHub 主页上的自我介绍是 "a library for composing asynchronous and event-based programs using observable sequences for the Java VM"（一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库）。

***ReactiveX***

简单来说，ReactiveX是一个专注于异步编程与控制可观察数据（或者事件）流式API，它是”观察者模式+迭代器模式+函数式编程”，它扩展了观察者模式，通过使用可观察的对象序列流来表述一系列事件，订阅者进行占点观察并对序列流做出反应（或持久化或输出显示等等）；借鉴迭代器模式，对多个对象序列进行迭代输出，订阅者可以依次处理不同的对象序列；使用函数式编程思想(functional programming)，极大简化问题解决的步骤。

***RxJava 到底是什么***

一个词：异步。说到根上，它就是一个实现异步操作的库，而别的定语都是基于这之上的。

## 引入RxJava依赖时遇到的问题

***问题描述***

今天准备学习使用RxJava，参照网上博客，但是在AS中引入依赖就出了问题，现在终于解决，记录下：

***错误一***

直接在project级的build.gradle中加上依赖：

buildscript {

repositories {

jcenter()

}

dependencies {

classpath 'com.android.tools.build:gradle:1.0.0'

compile 'io.reactivex:rxjava:1.1.6'

compile 'io.reactivex:rxandroid:1.2.1'

}

}

但是sync过程中报错，显示：Could not find method compile()...

经过一番查找，发现问题是，不能在buildscript中的dependencies 中加，应该另写一个dependencies ，在其中加上依赖。（buildscript中的dependencies中只能加形容classpath ‘...’这样的）

***错误二***

经过错误一调试后，另写一个dependencies，如下：

Dependencies{

compile 'io.reactivex:rxjava:1.1.6'

compile 'io.reactivex:rxandroid:1.2.1'

}

这样虽然可以sync成功，但是在代码中使用Observer等RxJava提供的类（或接口）时，总是自动引入java.util包下的Observer，而且没有rx包下的Observer这个选项，于是经过一番查找，发现正确的引入应该是这样的：

***正确的***

在**project级**下的build.gradle中，代码如下：

buildscript {

repositories {

jcenter()

}

dependencies {

classpath 'com.android.tools.build:gradle:1.0.0'

classpath 'me.tatarka:gradle-retrolambda:3.0.1'

}

}

在**module级**下的build.gradle中添加如下dependencies：

Dependencies{

compile 'io.reactivex:rxjava:1.1.6'

compile 'io.reactivex:rxandroid:1.2.1'

}

这样，才算是正确的引入了依赖，并且在代码中使用Observer时，自动导入的是rx包下的Observer

***总结***

由于是采用Androidstudio，对于gradle还不是很熟，导致出了这些问题。

## Builder——构造者模式

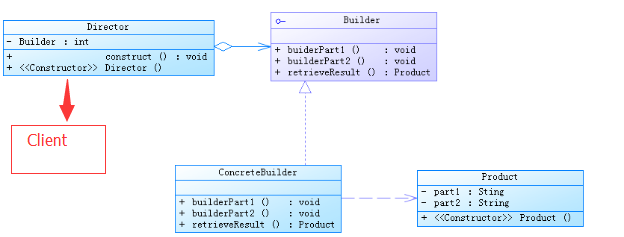
***定义***

（http://www.2cto.com/kf/201503/386548.html）

建造模式利用一个导演者对象和具体建造者对象一个个地建造出所有的零件，从而建造出完整的产品对象。

建造者模式将产品的结构和产品的零件的建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

***构造者模式的结构***



在Client类中需要得到产品Product。

***对上图的文字描述***

抽象建造者（Builder）角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体建造者 (ConcreteBuilder)角色。具体建造者类必须实现这个接口所要求的两种方法：一种是建造方法(buildPart1和 buildPart2)，另一种是返还结构方法(retrieveResult)。*一般来说，产品所包含的零件数目与建造方法的数目相符*。换言之，有多少零件，就有多少相应的建造方法。

具体建造者（ConcreteBuilder）角色：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：1.实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。2.在建造过程完成后，提供产品的实例。

导演者（Director）角色：担任这个角色的类调用具体建造者角色以创建产品对象。应当指出的是，导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色。

产品（Product）角色：产品便是建造中的复杂对象。一般来说，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关联的。

导演者角色是与客户端打交道的角色。导演者将客户端创建产品的请求划分为对各个零件的建造请求，再将这些请求委派给具体建造者角色。具体建造者角色是做具体建造工作的，但是却不为客户端所知。

一般来说，每有一个产品类，就有一个相应的具体建造者类。这些产品应当有一样数目的零件，而每有一个零件就相应地在所有的建造者角色里有一个建造方法。

***现实中的构造者模式（应用场景）***

就好象我要一座房子住，可是我不知道怎么盖（简单的砌墙，层次较低），也不知道怎么样设计（建几个房间，几个门好看，层次较高），于是我需要找一帮民工，他们会砌墙，还得找个设计师，他知道怎么设计，我还要确保民工听设计师的领导（说明设计师类持有民工类或其父类的引用），而设计师本身也不干活，光是下命令，这里砌一堵墙，这里砌一扇门，这样民工开始建设，最后，我可以向民工要房子了。在这个过程中，设计师是什么也没有，除了他在脑子里的设计和命令，所以要房子也是跟民工要，记住了！

***代码说明***

*（http://blog.csdn.net/linshutao/article/details/5925071）*

***— —***

定义工人接口，就是能够完成建造房子任务的人的通用要求：

// 工人接口，定义了各个工人所要进行的工所作。他们负责进行具体部件如窗户，地板的建造。

// 同时因为房子是民工建的，因此建设完成后由他把房子递交回房主

public interface Builder { Builder

public void makeWindow();

public void makeFloor();

public Room getRoom();

}

***— —***

定义设计师，他的职责是指挥房主指派给他的工人按照自己的设计意图建造房子：

// 设计师。他知道房子应该怎么设计，但他不会自己去建造，而是指挥民工去建造。

public class Designer { 设计师就是Director

// 指挥民工进行工作

public void order(Builder builder) {

builder.makeWindow();

builder.makeFloor();

}

}

***— —***

民工，他负责具体事物的实施：

// 民工。负责进行具体部件如窗户，地板的建造。

//同时因为房子是民工建的，因此建设完成后由他把房子递交回房主

public class Mingong implements Builder { ConcreteBuilder

private String window="";

private String floor="";

public void makeWindow() {

window=new String("window");

}

public void makeFloor(){

floor=new String("floor");

}

// 回交房子给房主

public Room getRoom() { 房子就是产品

if((!window.equals(""))&&(!floor.equals(""))) {

System.out.println("room ready!");

return new Room();

}

else return null;

}

}

***— —***

房主，就是雇人，收房：

// 房主。房主的任务就是聘请一个民工，一个设计师，同时把民工给设计师指挥，督促设计师开展工作。最后从民工手上收房。

public class Client {

public static void main(String[] args) {

Builder mingong = new Mingong();

Designer designer = new Designer();

designer.order(mingong);

mingong.getRoom();

}

}

***总结***

将这段代码结合上面的图片和文字描述，应该就能很好的理解***构造者模式***了!

## 结构经济学

*（https://www.huxiu.com/article/165497.html）*

现代社会中，如果一个国家经济以农业为主，百分之八九十的人口都是农民，靠耕作为生，这个国家必定是个穷国。想要变富，要想繁荣，只是继续发展农业是没有出路的，即使年年大丰收也无济于事。需要的是让国家的生产结构发生变化，从农业为主转为工业为主，也就是工业化。现代化的过程，不是原来的农业产量增加的过程，而必定是个生产结构发生深刻变迁的过程。

这就是从结构变迁的角度理解经济发展。这样研究经济发展的就是结构经济学。二战以后，很多新独立的国家在设计本国的经济发展之路时，主要的理论依据就是结构经济学。

***“新”“旧”结构经济学***

共同点是：都强调结构变迁对经济发展的重要意义，都认为国家在经济发展中有很大作用，主张国家通过产业政策的方式引导经济发展。但除此之外，就都是区别了。

区别是什么呢？

旧结构经济学主张，按照本国和先进国家生产结构上的差异，政府直接进行投资和建设，缺什么补什么，一项一项把工业体系搭建起来。新结构经济学则认为，这样做只能打造出一批不符合本国比较优势的企业，比如，在资金匮乏的落后国家兴建需要大量资金的重化工业。由于不符合比较优势，这些企业不可能形成自生能力，离不开政府的财政补贴。表面上看，国家建立起了前所未有的工业体系，但这个体系不但不能创造财富，反而每天都在消耗财富。

为了补贴这些没有自生能力的国营企业，国家要么用“剪刀差”的方式从农民那里弄来资金，要么从其他国家借来资金。苏东国家多采用前一种办法，拉美国家多用后一种办法。结果都很糟糕。苏东国家陷入经济停滞，拉美国家则被债务危机摧垮。

新结构经济学主张，国家的产业政策必须符合比较优势，因势利导，务求企业能获得自生能力，不能长期依靠财政补贴生存。产业政策的作用在于引导和帮助企业按照比较优势发展，而不是直接投资打造相关结构。

旧结构经济学认为，工业化的经济结构，对落后国家来说就是“有”“无”之分。没有就要去建，把“无”变成“有”。没有那么多钢铁厂，就直接建高炉。钢铁产量赶上发达国家，自然也就成了发达国家。

新结构经济学则认为，在落后国家和发达国家之间，有一系列台阶需要攀登。落后国家不可能一步赶上发达国家，而要依照比较优势，一步一步地改变生产结构，逐渐完成结构变迁，从最初的农业为主，转向发展轻工业、简单服务业。积累了资金和基础设施以后，才可以发展重工业。资金和技术进一步积累以后，才可以发展高科技产业和复杂服务业，等等。

一蹴而就的结构变迁之所以是不可能的，就是因为不同发展阶段的资源禀赋不同，比较优势也就不同。发展初期，落后国家往往劳动力资源丰富，资金匮乏，这时只能发展劳动密集型产业。如果勉强发展资金密集型产业，就赚不到钱，企业无法形成自生能力。新结构经济学主张，产业政策在发展的每一步都遵从比较优势，这就能让企业赚到钱，然后资源禀赋也跟着转变，比如资金渐渐就不那么匮乏了，这时，才可以着手发展资金密集型产业。

旧结构经济学已被抛弃。学术理论上，新结构经济学的主要对手是华盛顿共识。华盛顿共识把旧结构经济学的失败归因于政府的产业政策，所以主张尽可能排除政府产业政策，让企业家去自由发展经济。他们质问新结构经济学的首要问题就是：为什么是政府？你们为什么主张由政府引导，而不是让企业家自主摸索？

我认为，到底应该由政府通过产业政策引导，还是应该由企业家去自由摸索，这个问题的答案应该来自现实，而不应根据某些理论预先就做出明确判断。换句话说，这是一个经验问题，而不是先验问题。

......

当然不能根据现象就简单地得出理论，理论要靠深入的分析和对规律的探索。现象的作用在于提出问题，引发思考。

......

可见，“为什么要由政府来做？”这个问题很大程度上在现实中从来就不存在。这个问题反倒是学者后来在书房中空想出来的。

......

政府如何制定产业政策，这其实是新、旧结构经济学的核心区别。如前所述，旧结构经济学要求政府仿照发达国家，有意识地打造本国的经济结构，不必去考虑是否符合本国比较优势。

......

新结构经济学针对性地提出，政府的产业政策应该符合本国的比较优势，这样，或许初期政府需要做出一些补贴、基建或税收优惠，但企业很快就能赚到钱，有了自生能力。政府也就不必继续补贴了，反而可以去收税了。政府产业政策的引导目的就达到了。

......

相比之下，计划经济对国企、官员的考核则是“数量型”的，只看产量不看效益。赚钱不重要，产量赶上资本主义国家才重要。1958年8月17日，中共中央通过了《全党全民为生产1070万吨钢而奋斗》的决议，由此发动了大炼钢铁运动。看到了吧，目标是1070万吨钢，而不是通过炼钢赚钱。

因此，新结构经济学强烈地要求国家整体环境的“市场化”，只有有了“市场化”，才能有正确的价格信号，政府才能评估产业政策的效果。不仅如此，甚至对官员的考核也要“市场化”。那些好大喜功，不会赚钱只会花钱的官员，在现有的中国体制内是很难升迁的。

......

那些指责新结构经济学反市场化的批评是不得要领的。很多人对市场化的态度是叶公好龙式的。他们严重缺乏对市场化真实状况的理解能力。在书面上，他们赞颂市场化和市场经济，但当市场经济真出现在面前的时候，他们却惊骇于其中的种种具体景象而大加鞭挞。凡是他们不理解的市场化现象，就是邪恶的，就是反市场的。

# 2016-10-8

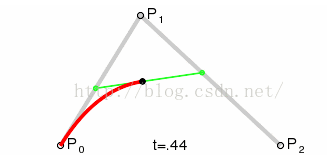
## 贝塞尔曲线

***概念***

三点确定一条曲线：起点，终点，辅助点。

***图示***

三个点的基本关系如下：



（http://blog.csdn.net/wingichoy/article/details/50492828）

P0、P2为起点和终点，P1为辅助点，红色的是贝塞尔曲线（二阶）。

***使用***

Android 的Path类提供了绘制二阶贝塞尔曲线的方法，使用方法如下：

//设置起点

path.moveTo(200,200);

//设置辅助点坐标 300，200、终点坐标400，200

path.quadTo(300, 200, 400, 200);

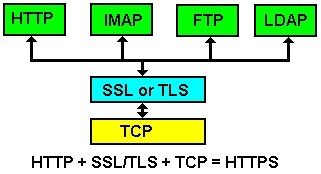
***总结***

贝塞尔曲线可以实现很多绚丽的效果，难的不是贝塞尔，而是good idea。

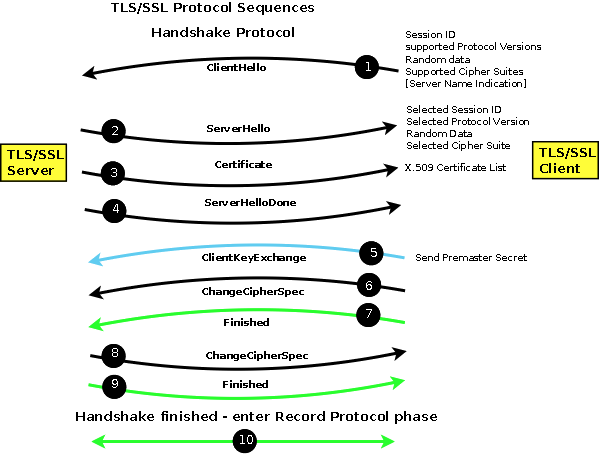
## HTTPS通信原理

*（https://jaq.alibaba.com/community/art/show?articleid=545）*

HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL/TLS：



SSL/TLS层负责客户端和服务器之间的加解密算法协商、密钥交换、通信连接的建立，安全连接的建立过程如下所示：



HPPTS握手协议有很丰富的内容，建议读者使用wireshark抓包进行分析，由于篇幅所限，这里不再进一步深入。

## Java8中的新特性——Lambda

*（http://blog.720ui.com/2016/java\_se8\_01\_lambda/）*

函数式编程是Java8中最值得学习的新特性。

***从函数式接口说起***

理解Functional Interface（函数式接口）是学习Java8 lambda表达式的关键所在。

***函数式接口的定义***

任何接口，如果只包含唯一一个抽象方法，那么它就是一个函数式接口。

对于函数式接口，我们可以通过lambda表达式来创建该接口的对象。

为了让编译器帮助我们确保一个接口满足函数式接口的要求，也就是说有且仅有一个抽象方法。Java8提供了@FunctionalInterface注解。举个简单的例子，Runnable接口就是一个FI，下面是它的源代码：

@FunctionalInterface

public interface Runnable {

public abstract void run();

}

使用@FunctionalInterface注解并不强制要求。但是使用注解会让代码看上去更清楚。

***lambda表达式的语法糖***

Java中lambda表达式的格式：参数、箭头->、一个表达式。

***Tips***

Java8中除了Lambda表达式外，还有一个新特性Stream API。

## PNG、JPG与SVG的区别

***定义***

——PNG，Portable Network Graphic Format，即可移植网络图形格式；

是一种位图文件存储格式，可以进行无损压缩。

——JPG，JointPhotographicExpertsGroup，联合图像专家组的缩写，也称为 JPEG，文件后辍名为"．jpg"或"．jpeg"；

是我们最常见的图片格式，图片占用存储较少，但是也牺牲了图片质量。

——SVG，Scalable Vector Graphics，可伸缩矢量图形；

它可以随意改变大小，而不影响图标质量。

所以，png格式的图片一般比jgp格式的图片占用内存大。

***区别***

——PNG格式可编辑，如图片中有字体等，可利用PS再做更改，而JPG格式不 可编辑；

——对于两种格式的解析度,PNG高于JPG；

——PNG不适用任何场合，因为它的体积很大，不利用显示，完全用于编辑者作 为以后更改图片时做的备份之用，而JPG格式因为体积小等原因适用于很多 场合；

***Android上的使用***

小尺寸，色彩数少，或者需要用到透明的时候，使用PNG；

大尺寸，色彩渐变色多的用JPG（这种条件下的PNG图片体积比JPG的大的多）；

# 2016-10-17

## RelativeLayout中的控件显示层次

RelativeLayout中的控件，如果没有设置相对位置，那么后添加的控件会被覆盖在前一个控件的上面。

## Android布局中weight的理解

***Google给的定义***

LinearLayout also supports assigning a weight to individual children with the android:layout\_weight attribute. This attribute assigns an "importance" value to a view in terms of how much space it should occupy on the screen. A larger weight value allows it to expand to fill any remaining space in the parent view. Child views can specify a weight value, and then any remaining space in the view group is assigned to children in the proportion of their declared weight. Default weight is zero.

这段描述中最重要的词已经用蓝色标注了，就是“remaining”，代表“剩余”的意思。

***举个例子***

水平方向布局的父类有三个子类，父类总的宽度是100，子类的宽度分别是10，20，30。 那么 android:layout\_weight 这个属性的目的，就是瓜分剩余的 100 - 10 - 20 - 30，也就是剩余的40的使用权。也就是将这40按各自的weight分配，而非将100按weight值分配。

## Android自定义View中的命名空间

***解释***

布局文件中使用自定义属性时，必须在最外层根布局添加

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

命名空间名称可以随便取，比如这里的hc，命名空间后面的值是固定的：

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

在AndroidStudio中统一使用这种方式，如果在Eclipse中，是不能用res-auto，

必须得替换成你自定义view所属的包名，如果你在恰好使用的自定义属性被做成了lib，那就只能使用res-auto了，而在android-studio里，无论你是自己写自定义view，还是引用的lib里的自定义的view 都只能使用res-auto这个写法。

***例子***

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

<LinearLayout

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

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

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

android:layout\_width="match\_parent"

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

<com.hc.studyview.MyView

android:layout\_width="match\_parent"

android:layout\_height="100dp"

hc:default\_size="100dp" />

</LinearLayout>

注意，当我们使用了hc作为我们自定义View的命名空间时，我们在定义我们的自定义View中的属性值时，就要用hc:而不能使用android:了。

***补充***

我们注意到，在xmlns:hc之外还有两个与其相似的xmlns:android与xmlns:tools,那么这两个是干什么用的呢？

据网上的说法：

xmlns:andoid——是为了引用android系统自带的属性（类比上面的xmlns:hc）；

xmlns:tools——主要是为adt插件使用的；

## 我理解的自定义View

***步骤***

——自定义View的属性

在res/values/ 下建立一个attrs.xml ， 在里面定义我们的属性和声明 我们的整个样式：

<?xml version="1.0" encoding="utf-8"?>  
<resources>  
 <declare-styleable name="CircleView">  
 <attr name="CircleColor" format="color"></attr>  
 <attr name="textSize" format="dimension"></attr>  
 </declare-styleable>  
</resources>

——在自定义View的构造方法中获取我们自定义的属性

public CircleView(Context context, AttributeSet attrs, int defStyleAttr) {  
 super(context, attrs, defStyleAttr);  
  
 TypedArray a = context.getTheme().obtainStyledAttributes(attrs,R.styleable.*CircleView*,defStyleAttr,0);  
 //获取自定义的CircleColor属性，并将默认颜色设为 黑色  
 circleColor = a.getColor(R.styleable.*CircleView\_CircleColor*, Color.*BLACK*);  
 //获取自定义的textSize属性  
 textSize = a.getDimensionPixelSize(*CircleView\_textSize*,defStyleAttr);  
 a.recycle();  
 p = new Paint();  
}

我们重写了3个构造方法，默认的布局文件调用的是两个参数的构造方法， 所以记得让所有的构造调用我们的三个参数的构造，我们在三个参数的构造 中获得自定义属性。

——重写onMeasure()方法

—这里主要要理清 MeasureSpec.EXACTLY和MeasureSpec.AT\_MOST这两种 模式与wrap\_content和match\_parent之间的关系。

—必须要在onMeasure()中调用setMeasuredDimension()方法:

*/\*\*  
 \* <p>This method must be called by {****@link*** *#onMeasure(int, int)} to store the  
 \* measured width and measured height. Failing to do so will trigger an  
 \* exception at measurement time.</p>  
 \*/*

——重写onLayout()方法

这里主要牵涉到自定义View/ViewGroup在界面中的布局问题。

——重写onDraw()方法

—自定义View的大小测量好了，位置也布局好了，接着就要在界面上绘制 了。于是调用onDraw()方法对此View进行绘制。

—onDraw()方法（绘制中），需要画笔——Paint类，和画布——Canvas类。

*还可以重写View中的很多方法，比如监听事件等，不过主要是以上几个方法的重写。*

——把自定义的View放到我们想要展示的界面中，当然也可以在代码中通过 addview函数加载到布局中：

<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:cc="http://schemas.android.com/apk/res-auto"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
  
 <com.example.gx.circleview.CircleView  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 cc:CircleColor="#ff0000"  
 cc:textSize="40sp"  
 />  
</RelativeLayout>

关于这部分代码的解释，可以参考上一个知识点。

***说明***

上面的步骤基本是一个自定义View的全部过程了，下面我们来简单介绍一下自定义View的分类

——自定义View按继承对象的不同可以分为自定义View和自定义ViewGroup;

——按类型来分，可以分为：自绘控件、组合控件以及继承控件；

## 对自定义View中onMeasure()方法的理解

***说明***

之前一直对于onMeasure()方法中的两个参数不太理解，在尝试了几个自定义View之后，有了点眉目，顺手几下。如下代码：

@Override  
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
 super.onMeasure(widthMeasureSpec, heightMeasureSpec);  
 widthSize = MeasureSpec.*getSize*(widthMeasureSpec);  
 int widthMode = MeasureSpec.*getMode*(widthMeasureSpec);  
 heightSize = MeasureSpec.*getSize*(heightMeasureSpec);  
 int heightMode = MeasureSpec.*getMode*(heightMeasureSpec);  
 setMeasuredDimension(width,height);  
}

这里得到的widthSize与heightSize其实是和layout.xml中的

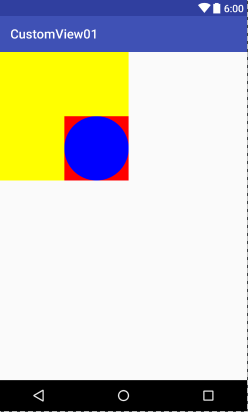
<com.example.gx.customview01.CustomView01  
 android:layout\_width="200dp"  
 android:layout\_height="200dp"  
 cc:radius="50dp"  
 />

layout\_width与layout\_height对应的（这里是确切的值）。

当在onDraw()中写如下代码（onMeasure()、onLayout()代码省略）时：

@Override  
protected void onDraw(Canvas canvas) {  
 super.onDraw(canvas);  
 p.setColor(Color.*YELLOW*);  
 canvas.drawRect(0,0,widthSize,heightSize,p);  
  
 p.setColor(Color.*RED*);  
 canvas.drawRect(getWidth()/2,getHeight()/2,width,height,p);  
  
 p.setColor(Color.*BLUE*);  
 p.setAntiAlias(true);  
 canvas.drawCircle(circleX,circleY,radius,p);  
}

得到的界面如下：



***总结***

由此可见，我们的自定义View的占用的空间大小实际上是黄色部分的大小，但我们通常可能只是在这个自定义View中取一部分出来绘制，例如红色或蓝色部分，而黄色部分通常是白色的，导致我们意识不到其存在，认为我们自定义View的大小就是红色或蓝色部分（笔者一开始也是这样认为的，因此理解不了onMeasure()方法中的两个参数究竟代表哪一个，因为实践的结果总是跟我预想的不一致，现在终于弄懂了）。

***补充***

——什么时候调用onLayout()方法：

一般调用onLayout()方法时，都牵涉到比较复杂的坐标计算，比如我们需要在黄色的 View内部画一个圆，然后在圆的边上画一个小矩形绕着圆边运动，这时的坐标就比较 多，也相对比较复杂了，我们可以再onLayout()中进行计算，然后再在onDraw()中绘制。

——关于MeasureSpec中的三种模式：

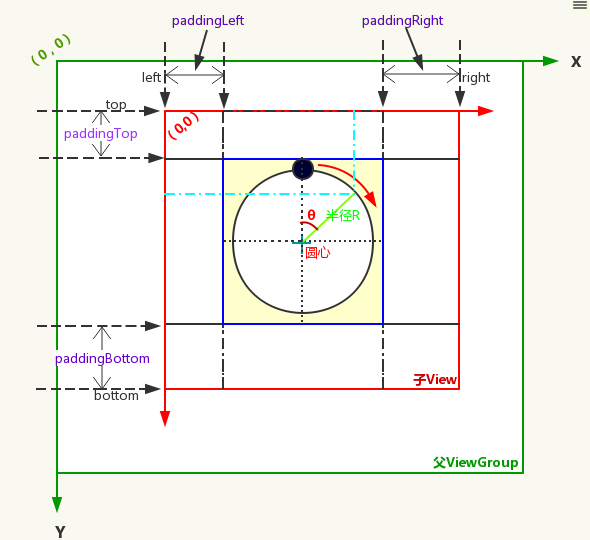
MeasureSpec.EXACTLY一般与match\_parent和确切值 所对应；

MeasureSpec.At\_MOST一般与wrap\_content对应（可将其设置为某个size的1/4这样）；

MeasureSpec.UPSPECIFIED意为没有限制，一般很少使用；

——关于View的空间问题：

可以参考一下[这篇文章](http://leoray.leanote.com/post/view-custom)中的这个配图

通过这篇文章和这个配图，再结合上面的描述，应该可以有更好的理解。

## Android子线程中更新UI的小想法

***描述***

当我们尝试在子线程中去更新UI时，android一般会提示以下错误：

android.view.ViewRootImpl$CalledFromWrongThreadException:

Only the original thread that created a view hierarchy can touch its views.

at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:6581)

at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:924)

***发现***

如果我们跟进ViewRootImpl类中，去查看checkThread(）和requestLayout()这两个方法，我们会发现在checkThread()的方法如下：

void checkThread(){

if(mThread!=Thread.currentThrend()){

throw new CalledFromWrongThreadException(

“Only the original thread that created a view hierarchy can

Touch its view.”

);

}

}

其中，mThread是主线程，在应用程序启动的时候，就已经被初始化了。由此我们可以得出结论：

在访问UI的时候，ViewRoot（ViewRootImpl的父类）会去检查当前是哪个线程访问的UI，如果不是主线程，那就会抛出上面的异常。

***想法***

那么，如果我们在子线程中更新UI的时候，ViewRootImpl还没创建，是否就不会发生崩溃而可以更新UI呢？

答案是可以的（http://blog.csdn.net/xyh269/article/details/52728861）。

# 2016-10-24

## Android中的动画

***简介***

Android系统提供了很多丰富的API去实现UI的2D与3D动画，最主要的划分可以分为如下几类：

View Animation： 视图动画，又叫补间（Tween）动画在古老的Android版本系 统中就已经提供了，*只能被用来设置View的动画*。

Drawable Animation： 这种动画（也叫Frame动画、帧动画）其实可以划分到 视图动画的类别，专门用来一个一个的显示Drawable 的resources，就像放幻灯片一样。

Property Animation： 属性动画只对Android 3.0（API 11）以上版本的Android 系统才有效，这种动画可以设置给任何Object，包括那 些还没有渲染到屏幕上的对象。这种动画是可扩展的， 可以让你自定义任何类型和属性的动画。

## Android中的各种支持包——Android Support Library

***前言***

在android开发中经常需要导入一些v4、v7、v13等官方支持包，但一直都只是使用，并没有思考诸如：

——Android官方为什么要提供支持包，

——都提供哪些支持包，

——这些支持包又提供了什么特性，

——开发者又应该如何选择使用这些支持包，

这类的问题。

***为什么提供支持包***

——向后兼容

比如，我们开的App需要支持的 minSdkVersion=9，targetSdkVersion=11，在程序里使用了android 3.0 (API level 11)提供的 ActionBar类，使用 compileSdkVersion=11 成功编译出apk。在 android 3.0 的设备上完美运行，但是app在 android 2.3 的设备就会 crash，报找不到 ActionBar 的错误。

这很好理解，因为旧版本上没有新版本里新增的类。为了避免使用了最新功能开发的app只能在最新系统的设备上运行的尴尬，官方把新版系统framework中新增加的接口提出来放到了 Android Support Library（支持包）中，开发者在遇到上面的情况时，就可以使用支持包中具有同样功能的 ActionBar类，这个支持包会打包进App里，这样使用了新版本系统上功能的App也可以向后兼容以前的老系统版本设备了。

使用支持包中的类除了让我们免于判断App运行的系统版本外，还可以使App在各个版本保持同样的用户体验。如在5.0以下系统使用 material design。

*App编译时用的 android sdk（android.jar）不会打包进我们的App中。因为App编码是使用 android.jar 中的接口就是 android设备里系统框架层（framework）对外提供的接口。*

——提供不合适打包进framework的功能

Android官方 对App开发提供了推荐设计，希望 Android应用 都有相对一致的交互设计来减少用户的使用成本，希望三方App类似系统应用从而完美融入到Android生态系统中。但是这都仅仅是推荐，不要求开发者一定要这样，如果有这种需求就可以使用官方支持包提供的这些功能，避免重复造轮子。如支持包中的 DrawerLayout、Snackbar 等类都是这种情况。

——为了支持不同形态的设备

通过使用支持包来在不同形态设备上提供功能，如手机、电视、可穿戴设备等。

***有哪些支持包和特性***

现在Android官方发布了下面 13类支持库，不同的支持库包含不同特征，适用的Android版本也不相同。通常情况下我们都使用到 v4 和 v7 这两个集合库，因为这两个库支持的android系统版本范围比较广，官方推荐的UI设计样式相关类也都在这两集合库中，同时v4和v7都有很多子库。

*gradle中jar依赖语句格式如 ：*

*compile 'jar文件组（group/命名空间）:jar文件名（name）:jar文件版本（version）'*

*例如Gradle编译脚本中整个v4库的依赖语句如下：*

*compile 'com.android.support:support-v4:24.2.1'*

***选择使用包***

其实在了解了支持包特性之后（具体可以上网查，这里不再赘述），这个问题也就迎刃而解了，这里再做下总结：

在使用Android Support Library 之前我们需要通过 sdk manager 安装 Android Support Repository，然后再在 gradle编译脚本 中添加如下依赖语句就可以了：

//以v4为例  
 compile 'com.android.support:support-v4:24.2.1'

前面文章说过gradle中jar依赖语句格式，对于Android Support Library库的依赖语句jar文件名和jar文件版本两部分需要选择确定。

——jar文件名

在选择之前要明确两件事，需要使用支持包的哪种特性、需要兼容的最低 Android版本，然后就可以确定具体依赖哪个支持库。

——jar文件版本

支持库的版本需要跟compileSdkVersion保持一致。

***补充***

由于依赖的支持库会打包进apk，所以官方推荐开发者在编译时使用ProGuard工具预处理release版本的apk。ProGuard工具除了*混淆源代码*外，还会移除那些依赖的支持库中没有使用到的类，达到*apk瘦身*的效果。

*（官方文档：https://developer.android.com/topic/libraries/support-library/index.html*

*参考博客：*<http://www.jianshu.com/p/f5f9a4fd22e8）>

## Android中实现再按一次退出程序

@Override

public boolean onKeyDown(int keyCode, KeyEvent event) {

// TODO Auto-generated method stub

if (keyCode == KeyEvent.KEYCODE\_BACK) {

if ((System.currentTimeMillis() - mExitTime) > 2000) {// 如果两次按键时间间隔大于2000毫秒，则不退出

Toast.makeText(this, "再按一次退出程序", Toast.LENGTH\_SHORT).show();

mExitTime = System.currentTimeMillis();// 更新mExitTime

} else {

System.exit(0);// 否则退出程序

}

return true;

}

return super.onKeyDown(keyCode, event);

}

注意其中的mExitTime 需要生命为*全局变量*并初始化为0，如下：

**long** mExitTime =0;

如果在onKeyDown方法中声明，则会一直弹出提示，并且退不出程序，因为两次的时间间隔将一直大于2000ms。

## Android数据存储中SharedPreferences文件的保存位置

***问题描述***

在模拟器上运行时，可以通过File Explorer查看到，文件保存在data\data\<项目资源包名>\shared\_prefs\ 下，但在真机上运行时却找不到这个路径。

***原因***

在真机上是存在的，不过如果没root的话是查看不到的。如果没root，只能看到mnt/sdcard；root之后能看到rootexplorer/data/data,保存的文件就在这里。

## Dagger2与ButterKnife的比较

***说明***

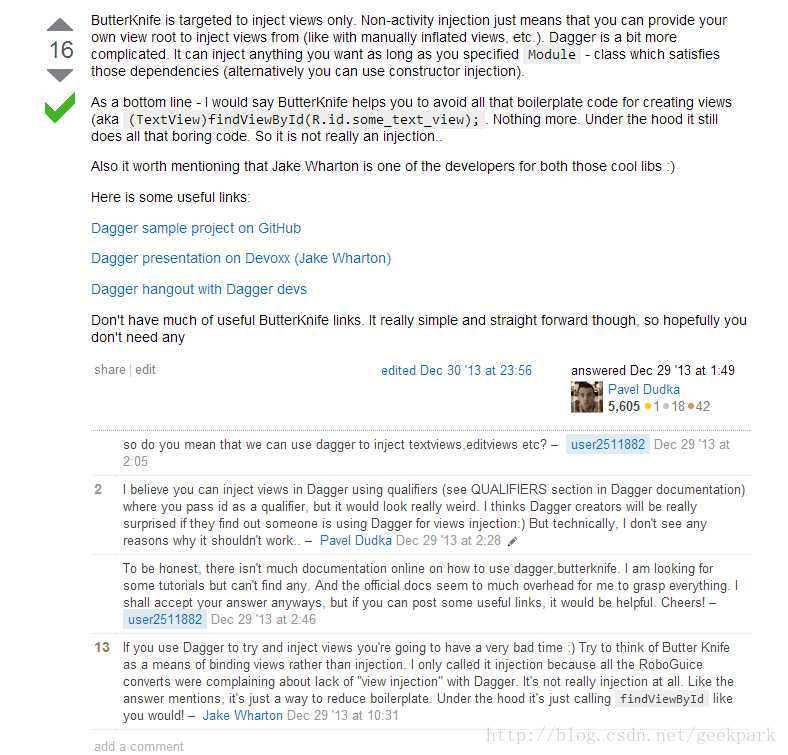
两者都是依赖注入的框架，但是Dagger2在View的注入上比较复杂，所以ButterKnife应运而生（甚至可以直接叫它findViewById），所以在有关View的注入上，选择ButterKnife更为方便，其他的就选择Dagger2。

（参考网址 http://www.itnose.net/detail/6101347.html）

***作者意见***

附上ButterKnife的作者JakeWharton在stackoverflow上的回答

Difference between Dagger and ButterKnife Android这个问题的截图：



# 2016-10-31

## Android中成员变量的命名为什么以“m”开始

***摘自官方文档：***

（http://source.android.com/source/code-style.html#follow-field-naming-conventions）

**Follow Field Naming Conventions**

——Non-public, non-static field names start with m.

——Static field names start with s.

——Other fields start with a lower case letter.

——Public static final fields (constants) are

ALL\_CAPS\_WITH\_UNDERSCORES.

**For example:**

public class MyClass {

public static final int SOME\_CONSTANT = 42;

public int publicField;

private static MyClass sSingleton;

int mPackagePrivate;

private int mPrivate;

protected int mProtected;

}

*注意上面说的都是成员变量的命名。*

## Android中关于sqlite的问题研究

***基本使用***

**关于sqlite的使用主要关注两个类**

——SQLiteOpenHelper

是数据库的帮助类，方便数据库的创建和更新，而且是个抽象类，所以我们需要写一个自己的类（作为帮助类）去继承它；其中还包括两个非常重要的实例方法 —getReadableDatabase()

—getWritableDatabase()

通过这两个方法得到数据库实例；

——SQLiteDatabase

是数据库类，里面包含了数据库中的CRUD等操作方法，在数据库创建之后（得到数据库的实例之后） ，就可以使用这个类中的方法来进行CRUD等操作；

**自己的帮助类（MyDatabaseHelper）**

MyDatabaseHelper extends SQLiteOpenHelper并重写里面的方法

使用代码：

SQLiteOpenHelper myDatabaseHelper = new MyDatabaseHelper();

// 得到自己帮助类的引用

SQLiteDatabase db = myDatabaseHelper.getWritableDatabase();

// 得到数据库实例(通过db进行CRUD等操作)

***使用事务（四步，见注释）***

try {

db .beginTransaction();//开启事务

db...;// 执行数据库操作

db .setTransactionSuccessful();//操作执行完成将事务设置成功 //，这一句必须要有。否则的话，在关闭

//事务的时候会回滚结果不提交。

}catch (Exception e){

e.printStackTrace();

} finally {

db .endTransaction();//数据库关闭前关闭事务

if (db != null){

db .close();

}

}

***一些细节***

——

数据的批量插入和数据的修改与删除建议加上事务

对于批量插入数据来说，如果没有开启事务，那么每执行一次插入之后都会提交一次，从而造成性能的极大浪费，耗时严重，如果开启事务，只会在事务成功之后才提交一次从而提升速度，且数据量越多，节约的时间越明显；（对于很多数据库来说，比如sqlite、mysql...如果没有使用事务，那么对于每一条语句来说，他们都是一个单独的事务，有多少条数据就有多少条磁盘操作。使用了事务，减少了创建事务的消耗，同时也减少了提交的次数，减少了磁盘操作）

对于数据的修改和删除来说，如果我们要求多个修改和删除操作要么同时成功要么同时不成功，那我们就需要使用事务了，因为事务具有原子性（遇到错误会回滚，要么都成功要么都不成功）。

（*SQLite的数据库本质上来讲就是一个磁盘上的文件，所以一切的数据库操作其实都会转化为对文件的操作，而频繁的文件操作将会是一个很好时的过程，会极大地影响数据库存取的速度。例如：向数据库中插入100万条数据，如果不使用事务，将会重复的打开关闭数据库文件100万次，所以速度当然会很慢；如果使用了事务，SQLite将把全部要执行的SQL语句先缓存在内存当中，然后等到COMMIT的时候一次性的写入数据库，这样数据库文件只被打开关闭了一次，效率自然大大的提高。*

*可见使用了事务之后却是极大的提高了数据库的效率。但是我们也要注意，使用事务也是有一定的开销的，所以对于数据量很小的操作可以不必使用，以免造成而外的消耗。*）

——

多线程中使用SQLiteDatabase

当我们在一个线程中db.close()时，如果别的线程还在使用这个db实例，那么正在使用db的这个线程就会出现异常。

解决办法：

a.将SQLiteDatabase实例放在Application中，使其生命周期和App一致;

b.采用计数器的方式，在Application中添加一个线程安全的计数器，每次数据库操作完成后检查一下计数是否为0。为0就关闭SQLiteDatabase，不为0就不关闭;

——

ContentValues容量调整

ContentValues内部采用了HashMap来存储Key-Value数据，ContentValues的初始容量是8，如果当添加的数据超过8之前，则会进行双倍扩容操作，因此建议对ContentValues填入的内容进行估量，设置合理的初始化容量，减少不必要的内部扩容操作。

——

数据库是什么时候创建的

SQLiteOpenHelper的子类被实例化的时候，并不会马上创建数据库，只有当用户调用getReadableDatabase()、getWritableDatabase()两个方法之一的时候，系统才会去创建数据库。

首次执行上述两个函数的时候，会回调SQLiteOpenHelper的onCreate(), onUpgrade(), onOpen()三个函数。

——

使用SqliteDatabase的execSQL(String sql)有可能会导致注入攻击

比如

delete from user where name='a';

有可能就变成

delete from user where name='a' or 1=1;

所以，最好使用

executeSql(String sql, Object[] bindArgs);

或者使用SqliteDatabase提供的insert,update,delete等方法，可防止注入攻击。

——

Sqlite的读操作是线程安全的，但是写操作是非线程安全的

## 关于RxJava的理解

***概念***

RxJava 在 GitHub 主页上的自我介绍是：

"a library for composing asynchronous and event-based programs using observable sequences for the Java VM"

翻译一下就是：  
“一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库”

***关键词***

观察者模式、异步

***特点***

Android 创造的 AsyncTask 和Handler ，其实都是为了让异步代码更加简洁。而RxJava 的*优势也是简洁*，但它的简洁的与众不同之处在于，随着程序逻辑变得越来越复杂，它依然能够保持简洁（通过流式API让逻辑变得很简洁）。还有一点很重要的是，对于异步来说，RxJava提供了很方便的线程切换。

***一些细节***

——

RxJava最核心的两个东西是Observables（被观察者，事件源）和Observers/Subscribers（观察者）。Observables发出一系列事件，Subscribers处理这些事件。这里的*事件可以是****任何****你感兴趣的东西*（触摸事件，web接口调用返回的数据。。。）

——

一个Observable可以*发出零个或者多个*事件，直到结束或者出错。每发出一个事件，都会调用它的Subscriber的onNext方法，最后调用Subscriber.onCompleted()/Subscriber.onNext()或者

Subscriber.onError()结束。

——

Rxjava的看起来很想设计模式中的观察者模式，但是*有一点明显不同*，那就是如果一个Observerble没有任何的的Subscriber，那么这个Observable是不会发出任何事件的。

***让简洁更简洁***

上面说了RxJava会让程序在逻辑上更简洁，那么能不能在代码上更加简洁呢？RxJava自然想到了这点。

一个简单例子：

让Observable对象一个发出“Hello World”字符串，然后Subscriber对象来处理这个事件——将“Hello World”打印出来

*（这个例子的功能真的很简单，没有涉及到异步，没有复杂的逻辑，仅仅只是打印一个对象发出的字符串。使用这个简单的例子一是为了更好的说明RxJava怎么使用，二是为了证明我们上面提到的“RxJava中的事件可以使任何你感兴趣的东西”）*

首先我们需要创建一个Observable（被观察者/事件源）对象:

1. Observable<String> myObservable = Observable.create(
2. **new** Observable.OnSubscribe<String>() {
3. @Override
4. **public** **void** call(Subscriber<? **super** String> sub) {
5. sub.onNext("Hello, world!");
6. sub.onCompleted();
7. }
8. }
9. );

接着创建一个Subscriber/Observer对象来处理Observable对象发出的字符串：

1. Subscriber<String> mySubscriber = **new** Subscriber<String>() {
2. @Override
3. **public** **void** onNext(String s) { System.out.println(s); }
5. @Override
6. **public** **void** onCompleted() { }
8. @Override
9. **public** **void** onError(Throwable e) { }
10. };

*Tips: Observer是观察者的接口， Subscriber是实现这个接口的抽象类,因此两个类都可以被当做观察者，由于Subscriber在Observe的基础上做了一些拓展，加入了新的方法，一般会更加倾向于使用Subscriber。*

最后，通过subscribe()函数就可以将我们定义的myObservable对象和mySubscriber对象关联起来，这样就完成了subscriber对observable的订阅：

1. myObservable.subscribe(mySubscriber);

看到这里，可能我们会奇怪：应该是“subscriber对observable的订阅”，但是代码中体现出来的却是“observable对subscriber的订阅”逻辑。

客官别急！这就为您上菜！

这样写的目的，完全是为了体现“流式API”的思想！

其实在代码内部，依然是“subscriber订阅了observable”，但是为了流式API，才让它们以“observable订阅了subscriber”的“假象”呈现出来。

好了，说到这里其实还没有涉及到“让简洁更简洁”（都是废话）。我们看到，为了打印一个“Hello World”就写了这么多代码，简洁个毛啊！

“Well,I have to take a step back and explain a litte bit!”

“让简洁更简洁”中的第一个“简洁”是RxJava体现出的逻辑上的简洁，它能让复杂的逻辑以*链式调用（流式API）*的形式来实现，避免多层嵌套循环带来的“迷之缩进”，并且当复杂度不断增加的时候，依然能够保持简洁，大大的方便了后期维护或者是增加需求等。而第二个“简洁”就是RxJava提供的代码上的简洁，也是我们接下来要说的。

**简化代码：**

对于Observable对象来说：

RxJava内置了很多简化创建Observable对象的函数，比如Observable.just就是用来创建只发出一个事件就结束的Observable对象，上面创建Observable对象的代码可以简化为一行：

1. Observable<String> myObservable = Observable.just("Hello, world!");

对于Subscriber/Obserable对象来说：

上面的例子中，我们其实并不关心OnComplete和OnError，我们只需要在onNext的时候做一些处理，这时候就可以使用Action1类（当参数不同的时候可以使用Action2、Action3...）：

1. Action1<String> onNextAction = **new** Action1<String>() {
2. @Override
3. **public** **void** call(String s) {
4. System.out.println(s);
5. }
6. };

subscribe方法有一个重载版本，接受三个Action1类型的参数，分别对应OnNext，OnComplete， OnError函数:

1. myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);

这里我们并不关心onError和onComplete，所以只需要第一个参数就可以:

1. myObservable.subscribe(onNextAction);
2. // Outputs "Hello, world!"

上面例子的代码最终可以简化成这样：

1. Observable.just("Hello, world!")
2. .subscribe(**new** Action1<String>() {
3. @Override
4. **public** **void** call(String s) {
5. System.out.println(s);
6. }
7. });

*Tips:其实RxJava中代码的简洁不仅是这里说的，还有上面提到的线程变换时的简洁*

到此，我们终于实现了第二个“简洁”了。

不过，使用java8的lambda可以使代码更简洁：

1. Observable.just("Hello, world!")
2. .subscribe(s -> System.out.println(s));

Android开发中，可以使用retrolambda这个gradle插件，这样你就可以在你的代码中使用lambda了。

***停止订阅***

当我们想要停止订阅的时候该怎么操作呢？

事实上，在执行Observable.subscribe()时，它会返回一个Subscrition,它代表了Observable和Subscriber之间的关系。你可以通过Subscrition解除Observable和Subscriber之间的订阅关系，并立即停止执行整个订阅链。示例代码如下：

Subscription subscription =

Observable.just("Unsubscribe me later").subscribe(s -> log(s));

subscription.unsubscribe();

***错误处理***

到目前为止，我们都没怎么介绍onComplete()和onError()函数。这两个函数是用来通知订阅者，被观察的对象将停止发送数据以及为什么停止（成功完成或者出错了）。

下面我们通过代码来展示怎么使用这两个函数：

Observable.just("Hello, world!")

.subscribe(new Subscriber<String>() {

@Override

public void onNext(String s) { ... }

@Override

public void onCompleted() { ... }

@Override

public void onError(Throwable e) { ... }

});

可以看到，这两个方法都是在Subscriber中调用的。使用这种模式，有以下**几个优点：**

——只要有异常发生onError()一定会被调用

这极大的简化了错误处理，只需要在一个地方处理错误即可以。而且 onError()和onCompleted()是互斥的，即同时只会有一个执行；

——操作符不需要处理异常

将异常处理交给订阅者来做，Observerable的操作符调用链中一旦有一个 抛出了异常，就会直接执行onError()方法（后面的操作符都不会执行）；

——能够知道什么时候订阅者已经接收了全部的数据

知道什么时候任务结束能够帮助简化代码的流程（虽然有可能Observable 对象永远不会结束）；

**与传统的错误处理相比，**这种错误处理方式更简单。传统的错误处理中，通常是在每个回调中处理错误。这不仅导致了重复的代码，并且意味着每个回调都必须知道如何处理错误，你的回调代码将和调用者紧耦合在一起。

使用RxJava，Observable对象根本不需要知道如何处理错误！操作符也不需要处理错误状态，一旦发生错误，就会跳过当前和后续的操作符。所有的错误处理都交给订阅者来做。

***Tips***

*Observable和Subscriber可以做任何事情*

——Observable可以是一个数据库查询，Subscriber用来显示查询结果；

——Observable可以是屏幕上的点击事件，Subscriber用来响应点击事件；——Observable可以是一个网络请求，Subscriber用来显示请求结果；

## RxJava中的操作符（Operators）

***前言***

上一篇我们说了关于RxJava的一些理解，但是没有提到RxJava中非常重要的一点——操作符。因为其比较重要，所以打算单独提出来说一下。

***概念***

操作符就是为了解决对Observable对象的变换的问题，操作符用于在Observable和最终的Subscriber之间*修改Observable发出的事件（意味着时对Observable使用而非Subscriber）*。

***Map操作符***

其是用来将一个事件转化为另一个事件的，比如我们将上篇中的例子的事件进行转化：

1. Observable.just("Hello, world!")
2. .map(**new** Func1<String, String>() {
3. @Override
4. **public** String call(String s) {
5. **return** s + " -Dan";
6. }
7. })
8. .subscribe(s -> System.out.println(s));

可以看到，map操作符返回一个Observable对象，这样我们就能实现链式调用。同时map操作符不必返回Observable对象原来返回的类型，比如上篇的例子中，是void无返回类型，现在我们使用了map操作符和

new Func1<String,String>，将它原来的void改变为返回String类型的值。

map操作符的作用也因此体现：

*返回一个Observable对象的同时转换Observable对象的返回类型*。

为了更好的说明，让上面例子中的subscriber不关心返回的字符串，而是拿到字符串的hash值：

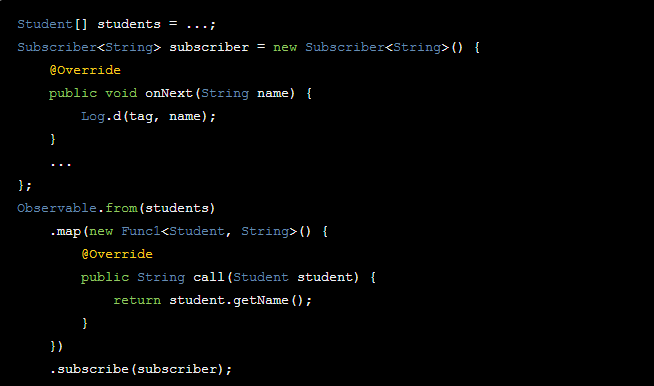
1. Observable.just("Hello, world!")
2. .map(**new** Func1<String, Integer>() {
3. @Override
4. **public** Integer call(String s) {
5. **return** s.hashCode();
6. }
7. })
8. .subscribe(i -> System.out.println(Integer.toString(i)));

我们初始的Observable返回的是字符串，最终的Subscriber收到的却是Integer。

***flatmap操作符***

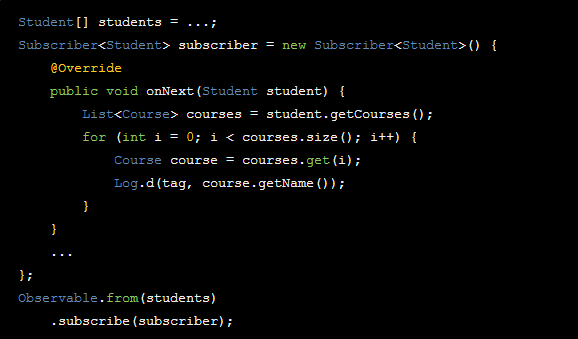
这是一个有用但是比较难理解的变换，所以我们需要多花点篇幅来介绍它。现在有这样一个需求：

假设有一个数据结构『学生』，现在需要打印出一组学生的名字。实现方式很简单：



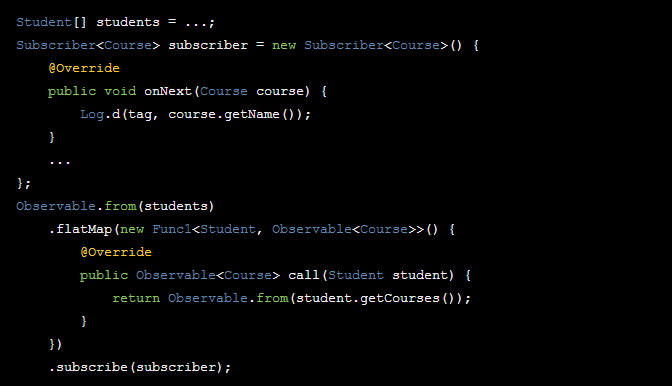
但是，当我们需要打印出每个学生所需要修的所有课程的名称呢？（区别在于，每个学生只有一个名字，但却有多个课程。）

我们可以这样来实现：



看上去似乎也很简单，但是仔细看代码我们发现，在Subscriber中我们使用了for循环，这好像有点违背了RxJava的初心——逻辑上的简洁。如果我们想直接在Subscriber中传入Course对象呢（这对于代码复用很重要）。继续用map()很显然是不行的，因为map()是一对一的转化，而我们现在要求的是一对多的转化。那怎么才能把一个Student转化成多个Course呢？

这时候，就需要flatMap()了：



从上面的代码可以看出， flatMap() 和 map() 有一个**相同点**：

它也是把传入的参数转化之后返回另一个对象。

但需要注意，和 map() **不同的是**：

flatMap() 中返回的是个 Observable 对象，并且这个 Observable 对象并不是被直接发送到了 Subscriber 的回调方法中。

flatMap() 的**原理**是这样的：

1. 使用传入的事件对象创建一个 Observable 对象；

2. 并不发送这个 Observable, 而是将它激活，于是它开始发送事件；

3. 每一个创建出来的 Observable 发送的事件，都被汇入同一个 Observable ，而这个 Observable 负责将这些事件统一交给 Subscriber 的回调方法。

这三个步骤，把事件拆成了两级，通过一组新创建的 Observable 将初始的对象『铺平』之后通过统一路径分发了下去。而这个『铺平』就是 flatMap() 所谓的 flat。

*（从代码看来，我们是通过两个Observable实现需求的：*

*第一个Observable.from(students)，将student逐个取出，*

*第二个Observable.from(student.getCourse()),将每个student中的course一个个取出，并发给Subscriber，让Subscriber直接来处理course，也就是将一个student对象变成多个course对象，即flatMap()的功能。*

*现在我们来想一想为什么flatMap()的第二个参数需要是Observable<T>类型的？*

*按照上面代码：Observable.from(student.getCourse())发出course类型的同时，返回类型是Observable<Course>,所以call(\*\*)方法的返回类型的Observable<T>的，从而flatMap()的*

*new Func1<Student,Obserable<Course>>中的第二个类型需要是Observable<T>的 。）*

Map()与flatMap()是Rxjava中俩个非常重要的**变换**操作符，我们要好好掌握他们。

***更多的操作符***

除了map()、flatMap()之外，RxJava还提供了一些操作符：

——filter()

输入和输出相同的元素，并且会过滤掉不满足检查条件的元素；

——take()

输出最多指定数量的结果，比如我们只想要最多5个结果，那我们就可以这样写： .take(5). ...;

——doOnNext()

允许我们在每次输出一个元素之前做一些额外的事情；

RxJava包含了大量的操作符。操作符的数量是有点吓人，但是很值得你去挨个看一下，这样你可以知道有哪些操作符可以使用。弄懂这些操作符可能会花一些时间，但是一旦弄懂了，你就完全掌握了RxJava的威力。

***Tips***

——Subscriber做的事情越少越好，对此我们可以通过增加操作符来实现；

——整个系统是高度可组合的，操作数据是一个很简单的过程；

——Observable和Subscriber是独立于中间的变换过程的，中间可以增减任 何数量的map；

## RxJava中的线程控制（Scheduler）

***前言***

把操作符和线程控制单独拿出来说，是因为我觉得这两者是RxJava中最重要的两个方面。RxJava中的Scheduler会大大的方便我们进行线程切换。

***说明***

在不指定线程的情况下，RxJava遵循的是线程不变的原则，即：在哪个线程调用subscribe()，就在哪个线程生产事件，在哪个线程生产事件，就会在哪个线程消费事件。

如果我们需要切换线程，就需要用到Scheduler（调度器）。RxJava中它就相当于线程控制器，RxJava通过它来指定每一段代码应该运行在什么样的线程。

RxJava已经内置了几个Scheduler，他们已经适合大多数的使用场景。

***内置的Scheduler API***

——Schedulers.immediate()

直接在当前线程运行，相当于不指定线程。这是默认的Scheduler；

——Schedulers.newThread()

总是启用新线程，并在新线程执行操作；

——Schedulers.io()

I/O操作（读写文件、读写数据库、网络信息交互等）所使用的Scheduler。 行为模式和newThread()差不多，区别在于io()的内部实现是使用一个无 数量上限的线程池，可以重用空闲的线程，因此多数情况下io()比 newThread()更有效率。不要把计算工作放在io()中，可以避免创建不必 要的线程；

——Schedulers.computation()

用来计算的Scheduler。这个计算指的是CPU密集型计算，即不会被I/O 等操作限制性能的操作，例如图形的计算。这个Scheduler使用的是固定 的线程池，大小为CPU核数。不要把I/O操作放在computation()中，否 则I/O操作的等待时间会浪费CPU。

——AndroidSchedulers.mainThread()

这是Android专用的Scheduler，它制定的操作将在Android主线程运行；

***切换Scheduler（一）***

有了这些Scheduler，接下来我们要关心的就是如何进行线程切换了。

RxJava提供了subcribeOn()和observeOn()这两个方法来对线程进行控制。

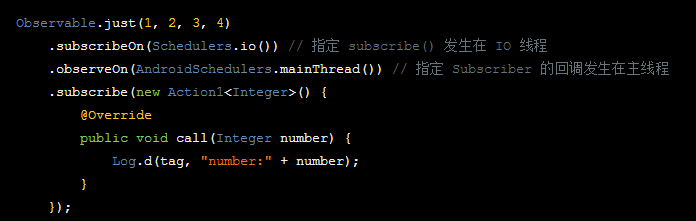
——subscribeOn()

指定subscribe()所发生的线程，即事件产生的线程；

——observeOn()

指定Subscriber所运行在的线程，或者叫做事件消费的线程；

文字描述看起来总是比较晦涩，我们通过代码来看一看：



上面这段代码中，由于 subscribeOn(Schedulers.io()) 的指定，被创建的事件的内容 1、2、3、4 将会在 IO 线程发出；而由于 observeOn(AndroidScheculers.mainThread()) 的指定，因此 subscriber 数字的打印将发生在主线程 。

事实上，这种在subscribe()之前写上两句

subscribeOn(Scheduler.io())和

observeOn(AndroidSchedulers.mainThread())

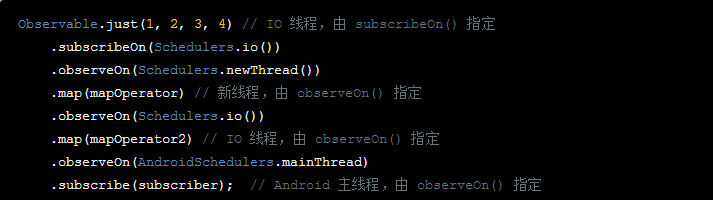
的使用方式非常常见，它适用于多数的『后台线程取数据，主线程显示』的程序策略。

***切换Scheduler（二）***

前面讲到了，可以利用 subscribeOn() 结合 observeOn() 来实现线程控制，让事件的产生和消费发生在不同的线程。可是在了解了 map() 、flatMap() 等变换方法后，有些人就问了：能不能多切换几次线程？

答案是：能。

因为 observeOn() 指定的是 Subscriber 的线程，而这个 Subscriber 并不是（严格说应该为『不一定是』，但这里不妨理解为『不是』）subscribe() 参数中的 Subscriber ，而是 observeOn() 执行时的当前 Observable 所对应的 Subscriber ，即它的直接下级 Subscriber 。换句话说，observeOn() 指定的是它之后的操作所在的线程。因此如果有多次切换线程的需求，只要在每个想要切换线程的位置调用一次 observeOn() 即可。上代码：



*（ subscribeOn() 的线程切换时事件还没有开始发送，因此 subscribeOn() 的线程控制可以从事件发出的开端就造成影响；*

*observeOn() 的线程切换则发生在它即将给下一级 Subscriber 发送事件时，因此 observeOn() 控制的是它后面的操作所在的线程。）*

如上，通过 observeOn() 的多次调用，程序实现了线程的多次切换。

不过，不同于 observeOn() ， subscribeOn() 的位置放在哪里都可以，而且它是只能调用一次的。

## RxBus——用RxJava来实现EventBus

***说明***

——

这里的RxBus不是第三方库，这是自己定义的一个用RxJava实现的类；

——

这个EventBus是指事件总线，而非第三方库EventBus。实现事件总线有两个比较有名的库：EventBus和otto，但是如果我们项目中使用了RxJava的话，就不妨用RxJava来写一个自己的事件总线（省去更多库的依赖）——RxBus。

***使用***

1.新建RxBus类（使用单例模式，保证对象一致）：

public class RxBus {  
 private static RxBus *rxBus*;  
 private final Subject<Object,Object> bus = new SerializedSubject<>(PublishSubject.*create*());  
 // PublishSubject只会把在订阅发生的时间点之后来自原始

// Observable的数据发射给观察者

private RxBus(){  
 }  
  
 public static synchronized RxBus getInstance(){  
 if(*rxBus* == null){  
 *rxBus* = new RxBus();  
 }  
 return *rxBus*;  
 }  
  
 public void send(Object o){  
 bus.onNext(o);  
 }  
  
 public Observable<Object> toObservable(){  
 return bus;  
 }  
}

1. 发送事件

\*\*(){

//方法中

RxBus.*getInstance*().send(\*\*);//\*\*可以是对象、字符串等

}

1. 订阅事件

\*\*(){

//方法中

RxBus.*getInstance*().toObservable().subscribe(new Subscriber<Object>() {  
 @Override  
 public void onCompleted() {  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 }  
  
 @Override  
 public void onNext(Object o) {  
 Log.*e*("tag","-------------eeeeee---------------");  
 if (o instanceof String){  
 tv.setText("aaa");  
 } //因为RxJava中的send()方法实际是调用Observable的onNext()方法，  
 } //所以需要在Subscriber的onNext()中对接收到的事件进行处理  
});

}

***遇到的问题***

**问题描述**

在刚开始写RxBus的demo测试的时候，我的想法是这样的：

在activity1中使用RxBus发送事件，然后通过activity1启动activity2，在activity2中订阅事件（处理事件）。

但是测试的结果是：

虽然订阅了，但是不会调用activity2中Subscriber的任何方法。

**问题原因**

经过Google和思考之后，发现原因如下：

我们发送事件在订阅事件之前，而普通的eventbus（事件总线）是必须先订阅再发送事件的，否则RxBus就接收不到被观察者发送的事件。

**解决问题**

现在我们在发送之前进行订阅，然后测试能不能通过RxBus实现eventbus（事件总线）的功能，发现结果是可以的。

但是，当我们想要实现“问题描述”中的功能时，该怎么办呢？

别但心，RxBus中也有相应的解决方案——Sticky事件！

***Sticky事件（粘性事件）***

**说明**

在Android开发中，Sticky事件只指事件消费者在事件发布之后才注册的也能接收到该事件的特殊类型。

Android中就有这样的实例，也就是Sticky Broadcast，即粘性广播。正常情况下如果发送者发送了某个广播，而接收者在这个广播发送后才注册自己的Receiver，这时接收者便无法接收到刚才的广播，为此Android引入了StickyBroadcast，在广播发送结束后会保存刚刚发送的广播（Intent），这样当接收者注册完Receiver后就可以接收到刚才已经发布的广播。

这就使得我们*可以预先处理一些事件*，让有消费者时再把这些事件投递给消费者。

# 2016-11-7

## Java中一些问题的解惑

**——**

**Q:**一个接口被多个类实现，调用这个接口中的抽象方法，哪个类的实现方法会被实际调用?

**A:**接口是不能实例化的，在调用接口中的方法时，你的这个接口引用必须指向某一个实现类，所以，当你此时调用接口中的某个方法时，实际调用的就是这个实现类中的方法。

——

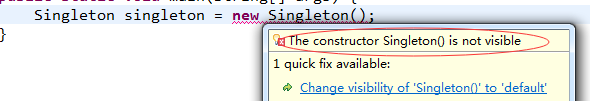
**Q:**如果 new 一个单例模式的类（构造方法是私有的），会怎么样?

**A:猜想：**

因为是单例模式，所以要保证不能让其他类通过new来得到单例类的实例，因此其构造方法需要私有；

**实测：**

当new一个单例类的时候IDE会出现以下提示：



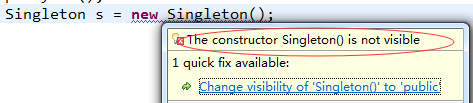
所以，如果一个类的构造方法是私有的，则这个类的实例不能通过new来得到。

那我们怎么才能得到这个类的实例呢？

答案是，只有这个类自己向外提供自己的实例（单例模式也利用这一特点）。

**Tips**

如果类的构造方法是protected，则同一包下的类可以通过new来获取它的实例，而对于其他package，其效果相当于private，如果new，则出现以下提示：



可以看到，提示的内容也是和private的提示是一样的。

## RecyclerView介绍

***官方说明***

“RecyclerView is a more advanced and flexible version of ListView. This widget is a container for large sets of views that can be recycled and scrolled very efficiently. Use the RecyclerView widget when you have lists with elements that change dynamically.”

***简介***

RecyclerView在职责分离上做的非常好，这意味着其内部有明确的分工，不同的类管理不同的任务。

所以，我们先来了解一下RecyclerView的结构层次和其各个部分的职责。

*RecyclerView中主要的五个类：*

——RecyclerView.LayoutManager

布局管理器，负责Item视图的布局的显示管理；

——RecyclerView.Adapter

适配器，为每一项Item创建视图（提供数据）；

——RecyclerView.ViewHolder

承载Item视图的子布局，可以理解为用来装子view的容器；

——RecyclerView.ItmeAnimator

负责处理数据添加或删除时的动画效果；

——RecyclerView.ItemDecoration

用来装饰子View，例如画分割线、偏移之类；

*Tip:当我们想要控制点击、长按事件时，我们只有自己实现了（RecyclerView没有为我们实现一些默认的点击事件）。*

***与ListView的异同***

——Adapter中的VIewHolder模式

对于ListView来说，通过创建ViewHolder来提升性能并不是必须的。因为ListView并没有严格的ViewHolder设计模式。但是在使用RecyclerView的时候，Adapter*必须实现至少一个*ViewHolder，必须遵循ViewHolder设计模式。

——定制Item条目

ListView只能实现垂直线性排列的列表视图，与之不同的是，RecyclerView可以通过设置RecyclerView.LayoutManager来定制不同风格的视图，比如水平滚动列表或者不规则的瀑布流列表。

——Item动画

在ListView中没有提供任何方法或者接口，方便开发者实现Item的增删动画。相反地，可以通过设置RecyclerView的RecyclerView.ItemAnimator来为条目增加动画效果。

——设置数据源（Adapter）

在LisView中针对不同数据封装了各种类型的Adapter，比如用来处理数组的ArrayAdapter和用来展示Database结果的CursorAdapter。相反地，在RecyclerView中*必须自定义实现*RecyclerView.Adapter并为其提供数据集合。

——设置条目分割线

在ListView中可以通过设置android:divider属性来为两个Item间设置分割线。如果想为RecyclerView添加此效果，则必须使用RecyclerView.ItemDecoration，这种实现方式不仅更灵活，而且样式也更加丰富。

——设置点击事件

在ListView中存在AdapterView.OnItemClickListener接口，用来绑定条目的*点击事件*。但是，很遗憾的是在RecyclerView中，并没有提供这样的接口，不过，提供了另外一个接口RcyclerView.OnItemTouchListener，用来响应条目的*触摸事件*。

***补充***

——

对于RecyclerView.Adapter来说，Adapter扮演着*两个角色*。

一是，根据不同的ViewType创建与之对应的ItemLayout;

二是，访问数据集合并将数据绑定到正确的View上；

因此就需要我们重写以下两个函数:

——

public VH onCreateViewHolder(ViewGroup parent, int viewType)

创建Item视图，并返回相应的ViewHolder；

——

public void onBindViewHolder(VH holder, int position)

绑定数据到正确的Item视图上；

另外我们还需要重写另一个方法，像ListView-Adapter那样，同样地告诉RecyclerView-Adapter列表Items的总数：

public int getItemCount()

 返回该Adapter所持有的Itme数量。

——

对于RecyclerView.LayoutManager来说，它是负责摆放Item的位置，并且负责决定何时回收和重用Item。

必须为RecyclerView指定LayoutManager,否则会出现异常。

## Java中的单例模式

***特点***

——私有的构造方法（不能让外界通过new得到类的实例）；

——只能存在一个实例（保证‘单’例）；

——要能为外界提供这个实例；

***优点***

——只存在一个实例对象，减少了内存开支；

——避免对文件等资源的多重占用；

——可以利用单例模式设置全局变量，用来进行共享资源的访问；

***缺点***

——一般没有接口，扩展困难；

——在Android中，如果单例模式持有Context的引用，很容易引发内存泄漏。所以在单例中推荐使用ApplicationContext；

***几种实现方法***

饿汉式

public class Singleton{

private static final Singleton singleton = new Singleton();

public Singleton(){

}

public static Singleton getInstance(){

return singleton;

}

}

小结：

不管什么时候用这个实例，只要加载这个单例类，它就会为你提供这个类的实例；

过早的提供了这个实例，造成内存不必要的浪费；

懒汉式

public class Singleton{

private static Singleto singleton = null;

private Singleton(){

}

public static synchronized Singleton getInstance(){

If(singleton = null){

singleton = new Singleton();

}

return singleton;

}

}

小结：

懒汉式是先不建立对象，当你需要时，调用单例类的getInstance()方法，这时会判断是否有这个实例了，如果没有，new一个，如果有，直接返回当前有的这个实例；

需要注意的是，我们给这个获取实例的getInstance()方法加上了synchronized修饰符，为的是防止异步请求或者多线程时发生new多个实例的情况；

有时候当实例singleton不为null的时候，我们还是要执行这个同步方法，造成不必要的同步开销；

DCL模式（Double CheckLock）

public class Singleton{

private static Singleton singleton = null;

private Singleton(){

}

public static Singleton getInstance(){

If(singleton == null){

synchronized(Singleton.class){

If(singleton == null){

singleton = new Singleton();

}

}

}

return singleton;

}

}

小结：

可以看到，DCL其实是对懒汉式单例的优化，将方法上的synchronized同步移到方法内部，避免了存在singleton实例还要执行同步的内存开销；

可能有人会对进行两次非空判断存在疑惑：

第一次判断，当singleton非空，不进行同步，避免了不必要的同步开销；

第二次判断，则是要确定非空，才能实例化，不然可能多线程环境下，new出 两个实 例。

# 2016-11-14

## Android中的几种测试介绍

*（https://www.zhihu.com/question/19716849）*

——Monkey

Monkey是Android SDK自带的测试工具，在测试过程中会向系统发送伪随机的用户事件流，如按键输入、触摸屏输入、手势输入等)，实现对正在开发的应用程序进行压力测试，也有日志输出。实际上该工具只能做程序做一些压力测试，由于测试事件和数据都是随机的，不能自定义，所以有很大的局限性；

——MonkeyRunner

MonkeyRunner是Android SDK提供的测试工具。严格意义上来说MonkeyRunner其实是一个Api工具包，比Monkey强大，可以编写测试脚本来自定义数据、事件。缺点是脚本用Python来写，对测试人员来说要求较高，有比较大的学习成本；

——Instrumentation

Instrumentation是早期Google提供的Android自动化测试工具类，虽然在那时候JUnit也可以对Android进行测试，但是Instrumentation允许你对应用程序做更为复杂的测试，甚至是框架层面的。通过Instrumentation你可以模拟按键按下、抬起、屏幕点击、滚动等事件。Instrumentation是通过将主程序和测试程序运行在同一个进程来实现这些功能，你可以把Instrumentation看成一个类似Activity或者Service并且不带界面的组件，在程序运行期间监控你的主程序。缺点是对测试人员来说编写代码能力要求较高，需要对Android相关知识有一定了解，还需要配置AndroidManifest.xml文件，不能跨多个App；

——UiAutomator

UiAutomator也是Android提供的自动化测试框架，基本上支持所有的Android事件操作，对比Instrumentation它不需要测试人员了解代码实现细节（可以用UiAutomatorviewer抓去App页面上的控件属性而不看源码）。基于Java，测试代码结构简单、编写容易、学习成本，一次编译，所有设备或模拟器都能运行测试，能跨App（比如：很多App有选择相册、打开相机拍照，这就是跨App测试）。缺点是只支持SDK 16（Android 4.1）及以上，不支持Hybird App、WebApp；

——Espresso

Espresso是Google的开源自动化测试框架。相对于Robotium和UIAutomator，它的特点是规模更小、更简洁，API更加精确，编写测试代码简单，容易快速上手。因为是基于Instrumentation的，所以不能跨App；

——Robotium

Robotium也是基于Instrumentation的测试框架，目前国内外用的比较多，资料比较多，社区也比较活跃。缺点是对测试人员来说要有一定的Java基础，了解Android基本组件，不能跨App；

——Appium

Appium是最近比较热门的框架，社区也很活跃。这个框架应该是是功能最强大的，

**优点：**

开源；

支持Native App、Hybird App、Web App；

支持Android、iOS、Firefox OS；

Server也是跨平台的，你可以使用Mac OS X、Windows或者Linux；

**原理：**

用Appium自动化测试不需要重新编译App；

支持很多语言来编写测试脚本，Java、Javascript、PHP、Python、C#、Ruby等主流语言；

不需要为了自动化测试来重造轮子，因为扩展了WebDriver。（WebDriver是测试WebApps 的一种简单、快速的自动化测试框架，所以有Web自动化测试经验的测试人员可以直 接上手）；

移动端自动化测试应该是开源的；

**设计理念：**

Client/Server架构，运行的时候Server端会监听Client端发过来的命令，翻译这些命令发送给移动设备或模拟器，然后移动设备或模拟器做出响应的反应。正是因为这种架构，所以Client可以使用Appium client libraries多种语言的测试脚本，而且Server端完全可以部署在服务器上，甚至云服务器；

Session，每个Client连接到Server以后都会有一个Session ID，而且Client发送命令到Server端都需要这个Session ID，因为这个seesion id代表了你所打开的浏览器或者是移动设备的模拟器。所以你甚至可以打开N个Session，同时测试不同的设备或模拟器；

Desired Capabilities，其实就是一个键值对，设置一些测试的相关信息来告诉Server端，我们需要测试iOS、还是Android，或者换是WebApp等信息；

Appium Server是Node.js写的，所以可以直接用NPM来进行安装；

Appium Clients，Mac OS和Win下提供GUI，不需要装Node.js，方便测试人员操作；

**相关限制：**

如果你在Windows使用Appium，你没法使用预编译专用于OS X的.app文件，因为Appium依赖OS X专用的库来支持iOS测试，所以在Windows平台你不能测试iOS Apps。这意味着你只能通过在Mac上来运行iOS测试；

**总结：**

在iOS部分是封装了UIAutomation；Android 4.2以上是用UiAutomator，Android 2.3 ~ 4.1用的是 Instrumentation，也就说Appium同时封装了UiAutomator和Instrumentation。所以Appium拥有了以上几大框架的所有优点：跨App，支持Native App、Hybird App、Web App，还支持N种语言来编写你的测试脚本。

## Android中的事件分发机制

*（http://blog.csdn.net/sinyu890807/article/details/9097463#reply）*

***说明***

android中触摸/点击任一控件，即会调用此控件的dispatchTouchEvent()方法，此方法在View类和ViewGroup类中（它们的子类没有重写）。

***View的事件分发***

触摸控件（View），首先执行dispatchTouchEvent()方法；

在dispatchTouchEvent()方法中，

1. **public** **boolean** dispatchTouchEvent(MotionEvent event) {
2. **if** (mOnTouchListener != **null** && (mViewFlags & ENABLED\_MASK) == ENABLED  &&  mOnTouchListener.onTouch(**this**, event)) {
3. **return** **true**;
4. }
5. **return** onTouchEvent(event);
6. }

在进行if条件判断时，执行了onTouch()方法。

当三个条件判断都为真的时候，返回true，不再继续执行onTouchEvent()方法。

现在，我们来看看这三个条件判断代表什么：

mOnTouchListener：代表控件有无设置setOnTouchListener()方法；

mViewFlags & ENABLED\_MASK：代笔控件是否是ENABLED的状态，默认是；

mOnTouchListener.onTouch()：是此方法的返回值，默认为false；

可以看到，默认情况下，dispatchTouchEvent()方法中的if判断结果为false，即在执行了判断中的onTouch()方法后，会跳出判断，执行onTouchEvent()方法。

而onClick()方法在onTouchEvent()方法内部，所以在View的事件分发中，先是执行onTouch()方法（如果注册监听了的话），再执行onClick()方法（如果注册了监听的话）。而如果我们修改了一些代码（例如修改onTouch()默认的返回值false为true），使得这三个条件都为true的话，就不会执行onTouchEvent()方法了，也就是不会执行onClick()方法了；

C.

到这里VIew的整个事件分发流程我们大概清楚了，但还有一个重要的知识点需要说明，就是touch事件的层级传递：

我们知道，如果给一个控件注册了touch事件，每次点击它的时候都会触发一系列的ACTION\_DOWN,ACTION\_MOVE,ACTION\_UP等事件。这里需要注意的是，如果在执行前一个action的时候返回了false，那么其后面的一系列action都不会得到执行了。简单来说，就是单dispatchTouchEvent()在进行事件分发的时候，只有前一个action返回true，才会触发后一个action。

联系前面的内容，我们可能会产生疑问：

我们在onTouch()中返回了false，但是ACTION\_DOWN和ACTION\_UP等方法还是得到了执行啊。

且慢慢看来。

onTouch()方法返回了false，使得onTouchEvent()方法得到了执行，而ACTION\_DOWN和ACTION\_UP等都是在onTouchEvent()方法中的。所以，虽然onTouch()方法返回了false，但是在onTouchEvent()方法中，执行action等代码之前，返回值又被更改为了true，因此action等代码才得以继续执行。

那么，是因为什么返回值成为了true了呢？

我们来看一下onTouchEvent()方法中的部分代码（代码较多，只贴部分）：

**public** **boolean** onTouchEvent(MotionEvent event) {

......

**if** (((viewFlags & CLICKABLE) == CLICKABLE ||

            (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {

**switch** (event.getAction()) {

**case** MotionEvent.ACTION\_UP:

......

**break**;

**case** MotionEvent.ACTION\_DOWN:

......

**break**;

......

**return** **true**;

     }

**return** **false**;

}

可以看到，在onTouchEvent()方法中，会先对此控件是否可： 点击/长按 进行判断，如果其中一个满足，则执行下面的各种action中的其中一种，并且返回true，这样，当我一个action动作完成，执行下一个action之前，返回值就是true，因此会继续执行下一个action。

那如果不满足if中的可：点击/长按 判断，就直接返回了false，也就不会执行一系列的action了，举个例子：

我们使用ImageView控件，然后给它注册一个touch事件，并返回false，如下：

1. imageView.setOnTouchListener(**new** OnTouchListener() {
2. @Override
3. **public** **boolean** onTouch(View v, MotionEvent event) {
4. Log.d("TAG", "onTouch execute, action " + event.getAction());
5. **return** **false**;
6. }
7. });

运行程序，点击ImageView，结果如下：



可以看到，在ACTION\_DOWN执行完之后，后面的一系列action都不会得到执行了。

这是为什么呢？

因为ImageVIew和Button控件不同，它是默认不可点击的，因此通过onTouchEvent()方法内部的if条件判断，直接返回了false，也就导致了其后面的其他action都无法执行（按说进不到if判断内部，也就不会执行ACTION\_DOWN啊，但为什么可以执行此action呢？）。

那么，有没有什么办法能让ImageView中的一系列action都能得到执行呢?

答案是肯定的。有两种：

一是:在ImageView的onTouch()方法中返回true来替换其默认的false；

二是：在布局文件中，给ImageView增加个android:clickable=”true”的属性；

***ViewGroup的事件分发***

A.

Android中的事件分发，先传递到ViewGroup，再由ViewGroup传递到View；

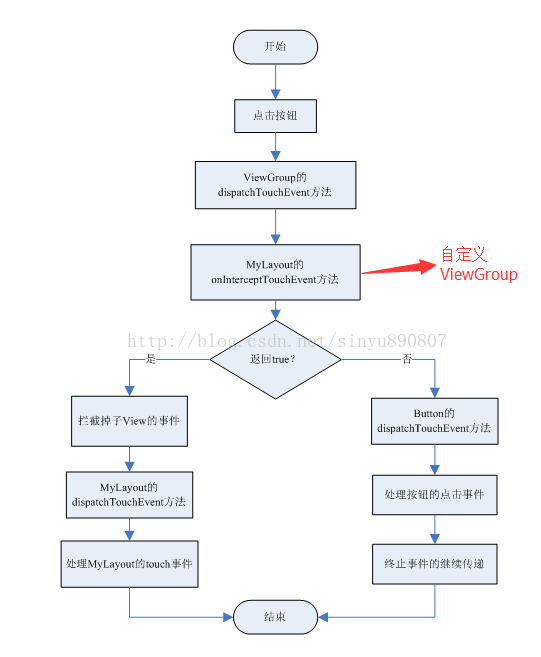
B.

在ViewGroup中，可以通过onInterceptTouchEvent方法对事件传递进行拦截，onInterceptTouchEvent方法返回true代表不允许事件继续向子View传递，返回false代表不对事件进行拦截，默认返回false;

C.

子View中如果将传递的事件消费掉，ViewGroup中将无法接收到任何事件；

ViewGroup事件分发过程流程图：



前面我们说了，只要你触摸了任何控件，就一定会调用该控件的dispatchTouchEvent方法。这个说法没错，只不过还不完整而已。

实际情况是，当你点击了某个控件，**首先**会去调用该控件所在布局的dispatchTouchEvent方法，**然后**在布局的dispatchTouchEvent方法中找到被点击的相应控件，再去调用该控件的dispatchTouchEvent方法。

如果我们点击了MyLayout中的按钮，会先去调用MyLayout的dispatchTouchEvent方法，可是你会发现MyLayout中并没有这个方法。那就再到它的父类LinearLayout中找一找，发现也没有这个方法。那只好继续再找LinearLayout的父类ViewGroup，你终于在ViewGroup中看到了这个方法，**按钮的**dispatchTouchEvent方法就是在这里调用的。

## Android中的权限机制

*（http://gold.xitu.io/post/57d5de3e2e958a00546a7465）* ***前言***

Android的权限机制可以从Android 6.0（Android M,API level 23）开始分为两个阶段——之前、之后。

在旧的权限管理系统中，权限仅仅在App安装时询问用户一次，用户同意了这些权限App才能被安装（某些深度定制系统另说），App一旦安装后就可以偷偷的做一些不为人知的事情了。

在Android 6.0开始，App可以直接安装，App在运行时一个一个询问用户授予权限，系统会弹出一个对话框让用户选择是否授权某个权限给App（这个Dialog不能由开发者定制），当App需要用户授予不恰当的权限的时候，用户可以拒绝，用户也可以在设置页面对每个App的权限进行管理。

*Tips:这个对话框不是开发者调用某个权限的功能时由系统自动弹出，而是需要开发者手动调用，如果你直接调用而没有去申请权限的话，将会导致App奔溃。*

这对于用户来说是好事，但是对于开发者来说我们不能直接调用方法了，我们不得不在每一个需要权限的地方先检查并请求用户授权，所以就引出了下面的问题：

***哪些权限需要动态申请***

新的权限策略将权限分为两类：

**第一类（Normal Permissions）** 是不涉及用户隐私的，只需要在Manifest中声明即可，比如网络、蓝牙、NFC等；

此类权限都是正常保护的权限，只需要在AndroidManifest.xml中简单声明这些权限即可，安装即授权，不需要每次使用时都检查权限，而且用户不能取消以上授权，除非用户卸载App；

**第二类（Dangerous Permissions）** 是涉及到用户隐私信息的，需要用户授权后才可使用，比如SD卡读写、联系人、短信读写等；

所有危险的Android系统权限属于权限组，如果APP运行在Android 6.0 (API level 23)或者更高级别的设备中，而且targetSdkVersion>=23时，系统将会自动采用动态权限管理策略，如果你在涉及到特殊权限操作时没有做动态权限的申请将会导致App崩溃，因此你需要注意：

——此类权限也必须在Manifest中申明，否则申请时不提示使用用户，直接 回调开发者权限被拒绝；

——同一个权限组的任何一个权限被授权了，这个权限组的其他权限也自动 被授权。例如，一旦WRITE\_CONTACTS被授权了，App也有READ\_CONTACTS 和GET\_ACCOUNTS了；

——申请某一个权限的时候系统弹出的Dialog是对整个权限组的说明，而不 是单个权限。例如我申请READ\_EXTERNAL\_STORAGE，系统会提示"允许 xxx访问设备上的照片、媒体内容和文件吗？"；

如果App运行在Android 5.1 (API level 22)或者更迭级别的设备中，或者targetSdkVersion<=22时（此时设备可以是Android 6.0 (API level 23)或者更高），在所有系统中仍将采用旧的权限管理策略，系统会要求用户在安装的时候授予权限。其次，系统就告诉用户App需要什么权限组，而不是个别的某个权限。

*Tips:当我们使用Android 6.0或之后的系统时，对于危险权限，我们首先需要在manifest.xml中声明（可能是为了兼容6.0以前的系统，因为他们没有动态申请），然后在代码中动态进行动态申请权限（这一步，可能要先判断当前系统的API level，如果>=23，就执行动态申请代码，否则不执行，因为6.0以前的系统不支持动态申请权限）。*

***关于运行时权限的一些建议***

——只请求你需要的权限，减少请求的次数，或用Intent来代替，让其他的应用来处理：

—如果你使用Intent，你不需要设计界面，由第三方的应用来完成所有操作。 比如打电话、选择图片等；

—如果你请求权限，你可以完全控制用户体验，自己定义UI。但是用户也可 以拒绝权限，就意味着你的应用不能执行这个特殊操作；

——防止一次请求太多的权限或请求次数太多，用户可能对你的应用感到厌烦，在应用启动的时候，最好先请求应用必须的一些权限，非必须权限在使用的时候才请求，建议整理并按照上述分类管理自己的权限：

—普通权限（Normal PNermissions）：只需要在Androidmanifest.xml中声明 相应的权限，安装即许可；

—需要运行时申请的权限（Dangerous Permissions）：

1. 必要权限：最好在应用启动的时候，进行请求许可的一些权限（主要 是应用中主要功能需要的权限，比如对应在线应用来说的 网络权限）;
2. 附带权限：不是应用主要功能需要的权限(如：选择图片时，需要读取 SD卡权限)；

——解释你的应用为什么需要这些权限：在你调用requestPermissions()之前，你为什么需要这个权限：

—例如，一个摄影的App可能需要使用定位服务，因为它需要用位置标记照 片。一般的用户可能会不理解，他们会困惑为什么他们的App想要知道他 的位置。所以在这种情况下，所以你需要在requestpermissions()之前告诉 用户你为什么需要这个权限；

——使用兼容库support-v4中的方法：

—ContextCompat.checkSelfPermission();

ActivityCompat.requestPermissions();

ActivityCompat.shouldShowRequestPermissionRationale();

***几个重要的方法与常量解释***

——PackageManager中的两个常量：

—PackageManager.PERMISSION\_DENIED：该权限是被拒绝的；

—PackageManager.PERMISSION\_GRANTED：该权限是被授权的；

——Activity中或者Fragment都会有以下几个方法：

—int checkSelfPermission(String);

—void requestPermissions(int, String...);

—boolean shouldShowRequestPermissionRationale(String);

—void onRequestPermissionsResult();

*上述四个方法中，前三个方法在support-v4的ActivityCompat中都有，建议使用兼容库中的方法。最后一个方法是用户授权或者拒绝某个权限组时系统会回调Activity或者Fragment中的方法。*

***checkSelfPermission()检查权限***

——检查某一个权限的当前状态，你应该在请求某个权限时检查这个权限是否已 经被用户授权，已经授权的权限重复申请可能会让用户产生厌烦；

——该方法有一个参数是权限名称，有一个int的返回值，用这个值与上面提到 的两个常量做比较可判断检查的权限当前的状态：

if (ContextCompat.checkSelfPermission(context, Manifest.permission.READ\_CONTACTS)

!= PackageManager.PERMISSION\_GRANTED) {

// 没有权限，申请权限。

}else{

// 有权限了，去放肆吧。

}

***requestPermissions()申请权限***

——请求用户授权几个权限，调用后系统会显示一个请求用户授权的提示对话框，App不能配置和修改这个对话框，如果需要提示用户这个权限相关的信息或说明，需要在调用 requestPermissions() 之前处理，该方法有两个参数：

—int requestCode，会在回调onRequestPermissionsResult()时返回，用来判断 是哪个授权申请的回调；

—String[] permissions，权限数组，你需要申请的权限的数组；

——由于该方法是异步的，所以无返回值，当用户处理完授权操作时，会回调Activity或者Fragment的onRequestPermissionsResult()方法；

***onRequestPermissionsResult处理权限结果回调***

——该方法在Activity/Fragment中应该被重写，当用户处理完授权操作时，系统 会自动回调该方法，该方法有三个参数：

—int requestCode，在调用requestPermissions()时的第一个参数；

—String[] permissions，权限数组，在调用requestPermissions()时的第二个参 数；

—int[] grantResults，授权结果数组，对应permissions，具体值和上方提到的 PackageManager中的两个常量做比较，例如：

@Override

public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {

switch (requestCode) {

case MMM: {

if (grantResults.length > 0

&& grantResults[0] ==

PackageManager.PERMISSION\_GRANTED) {

// 权限被用户同意，可以去放肆了。

} else {

// 权限被用户拒绝了，洗洗睡吧。

}

return;

}

}

}

***shouldShowRequestPermissionRationale()***

——是否应该显示请求权限的说明，即提示用户申请此权限的原因；

——第一次请求权限时，用户拒绝了，调用

shouldShowRequestPermissionRationale()后返回true，应该显示一些为什么 需要这个权限的说明；

——用户在第一次拒绝某个权限后，下次再次申请时，授权的dialog中将会出现 “不再提醒”选项，一旦选中勾选了，那么下次申请将不会提示用户；

——第二次请求权限时，用户拒绝了，并选择了“不在提醒”的选项，调用 shouldShowRequestPermissionRationale()后返回false；

——设备的策略禁止当前应用获取这个权限的授权： shouldShowRequestPermissionRationale()返回false；

——加这个提醒的好处在于，用户拒绝过一次权限后我们再次申请时可以提醒该 权限的重要性，面得再次申请时用户勾选“不再提醒”并决绝，导致下次申 请权限直接失败；

综上所述，整合之后的代码如下：



***运行时权限最佳实践的套路***

总体下来我们对运行时权限有了一个系统的认识，同时也注意到一些细节：

——需要区分各种Normal Permissioin和Dangerous Permissions；

——判断多个权限授权回调时需要判断每一个权限是否全都是被授权了，否则操 作不能继续；

——需要请求多个权限时需要挨个检查是否已经被授权过，没授权的才去请求， 还要检查这些权限是否需要提示用户，如果多个权限都需要提示，该如何处 理；

——上述1 2 3如果在需要在多个页面 实现，代码重复；

——......

其实问题远远不止这些，虽然实现代码比较简单，但是代码重复加上需要我们考虑和注意的细节太多，都使得我们要面临的问题很多。

那么下面就为大家介绍一个开源内裤来解决这一系列问题。

***AndPermission***

这个开源库名叫AndPermission：<https://github.com/yanzhenjie/AndPermission，经过我的实践是完全解决了上述问题，推荐大家使用。具体使用方法这里不作叙述，如果想了解，可以搜索前文给出的网址或者可以进入这个开源项目的github网址查看。>

## Looper、Handler、Message三者的关系

***说明***

Looper、Handler、Message这三者都是与Android*异步消息处理线程*相关的概念。

***异步消息处理线程***

异步消息线程启动后，会出现一个无限的循环体，每循环一次，都会从循环体内部的消息队列（MessageQueue，MQ）中取出一个消息，然后回调相应的消息处理函数。执行完成一个消息后则继续循环。如果消息队列（MQ）为空，则此异步线程会阻塞等待。

***三者的关系***

——Handler发送消息（Message）；

——Looper创建MessageQueue（MQ）；

——MQ维护Handler创建的Message，使不同的消息成为一个消息队列；

异步消息处理线程每次循环都会从MQ中取出一个Message来处理，处理完后继续循环，然后在从MQ中取Message处理，依次这样进行。直到MQ为空，使得异步线程阻塞等待。

***注意***

——一个线程可以有多个Handler，但只能有一个Looper（也只能有一个MQ， ，MQ被封装在Looper中）；

——Looper是MQ的管家，调用Looper的loop()方法后，才进入循环体，Looper 会不断从MQ中取消息，发送到Handler的handlerMessage()方法中；

——Message一般在子线程中创建，然后被Handler发出，再进入Looper创建 的MQ中成为消息队列的一部分，之后不断循环直到被Looper取出，发送 给主线程中的Handler的handlerMessage()方法。然后handlerMessage()方法 根据消息的不同，进行不同的处理，如UI更新等。

——暂时从概念角度说明了一下，如果想更详细的了解，可以查看这三个类的源 码；

## 异步和多线程的区别

*（http://www.cnblogs.com/DebugLZQ/archive/2012/09/05/2670986.html）*

***说明***

有些时候我们就认为异步和多线程是等同的概念。但是，异步和多线程还是有一些区别的，而这些区别造成了使用异步和多线程的时机的区别。

***异步操作的本质***

所有的程序最终都会由计算机硬件来执行!

所以为了更好的理解异步操作的本质，我们有必要了解一下它的硬件基础。 熟悉电脑硬件的朋友肯定对DMA这个词不陌生，硬盘、光驱的技术规格中都有明确DMA的模式指标，其实网卡、声卡、显卡也是有DMA功能的。

DMA就是直 接内存访问的意思，也就是说，拥有DMA功能的硬件在和内存进行数据交换的时候可以不消耗CPU资源。只要CPU在发起数据传输时发送一个指令，硬件就开 始自己和内存交换数据，在传输完成之后硬件会触发一个中断来通知操作完成。这些无须消耗CPU时间的I/O操作正是异步操作的硬件基础。 所以即使在DOS 这样的单进程（而且无线程概念）系统中也同样可以发起异步的DMA操作。

***线程的本质***

线程不是一个计算机硬件的功能，而是操作系统提供的一种逻辑（功能）。 线程本质上是进程中一段并发运行的代码，所以线程需要操作系统投入CPU资源来运行和调度。

***异步操作的优缺点***

因为异步操作无须额外的线程负担，并且使用回调的方式进行处理，在设计良好的情况下，处理函数可以不必使用共享变量（即使无法完全不用，最起码可以减少 共享变量的数量），减少了死锁的可能。

当然异步操作也并非完美无暇，编写异步操作的复杂程度较高，程序主要使用回调方式进行处理，与普通人的思维方式有些出入，而且难以调试。

***多线程的优缺点***

多线程的优点很明显，线程中的处理程序依然是顺序执行，符合普通人的思维习惯，所以编程简单。

但是多线程的缺点也同样明显，线程的使用（滥用）会给系统带来上下文切换的额外负担，并且线程间的共享变量可能造成死锁的出现。

***Tips***

——我们可以把多线程可以看成是实现异步的一种方式，异步是个概念，而多线 程是它的一种具体执行；

——实现异步可以不消耗CPU资源，但是如果要实现多线程则必须需要消耗CPU 资源。

## Android布局优化之ViewStub、include、merge的使用

***（****http://blog.csdn.net/bboyfeiyu/article/details/45869393）*

***说明***

ViewStub是View的子类；

include、merge是布局中的标签（可以去看一下他们是哪个的子类）；

***Include***

*场景*

Include是为了解决重复定义相同布局的问题。例如我们有五个界面都要用相同的标题栏（包括标题TextView和返回ImageButton），这时如果我们使用include就能避免在这五个界面中写相同的标题栏的布局代码。

*使用*

我们把这个标题TextView和返回ImageButton封装成一个单独的xml文件，在需要此标题栏的布局中通过include标签引入即可。代码：

标题栏xml——my\_title\_layout.xml



需要使用此标题栏布局的xml

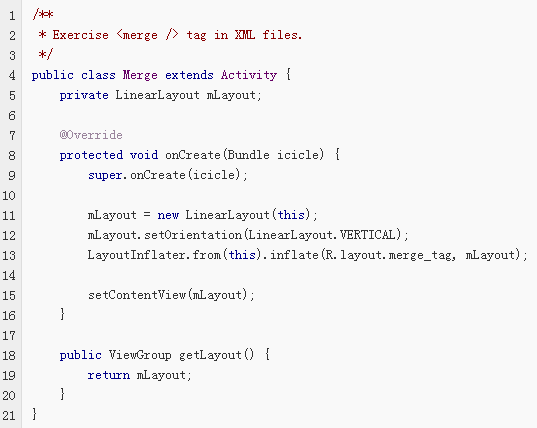


可以看到，在此布局文件中，我们使用include标签引入了我们封装的xml标题栏文件。

***Merge***

Merge标签是为了减少在使用include引入布局文件时的层级。

对于这个标签，我们可以看一下它的源码：



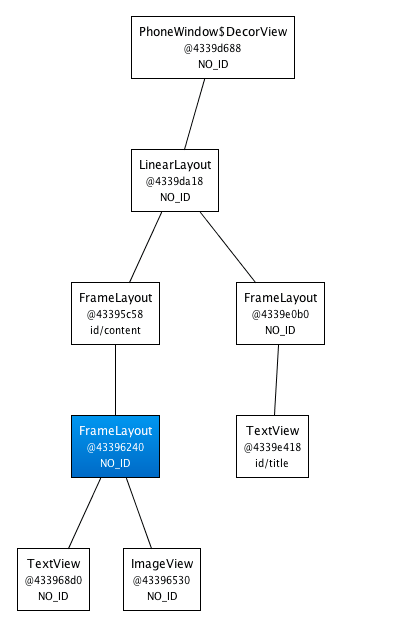
我们发现，这个标签竟然是Activity的子类，而且里面有一个LinearLayout对象。

下面我们来说一下，它到底减少了层级中的哪一层。

例如我们在复用一个含有多个子控件的布局时，肯定需要一个ViewGroup来管理，例如这样 :



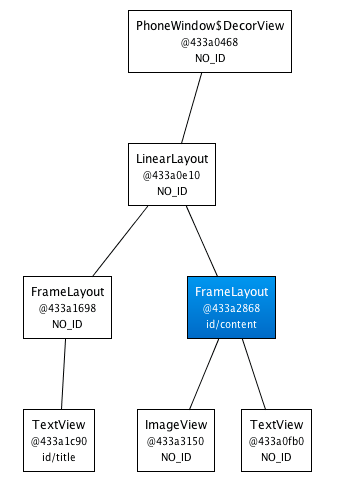
此时，将该布局通过include引入另一个布局文件时就会多引入了一个FrameLayout层级，此时结构如下 :



而如果使用merge标签就会消除上图中蓝色的FrameLayout层级。示例如下 :



此时的层级结构图如下：



可以看到，原来的FramLayout层消失了。

*Tips:具体是如何实现的，我们可以去查看源码，也可以看一下开头给出的地址链接中的博客。*

***ViewStub***

*说明*

“ViewStub is a lightweight view with no dimension and doesn’t draw anything or participate in the layout. As such, it’s cheap to inflate and cheap to leave in a view hierarchy. Each ViewStub simply needs to include the android:layout attribute to specify the layout to inflate.”

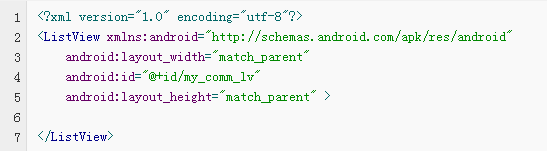
ViewStub是一个宽高都为0的一个View，它默认是不可见的，只有通过调用setVisibility函数或者Inflate函数才会将其要装载的目标布局给加载出来，从而达到延迟加载的效果，这个要被加载的布局通过android:layout属性来设置。

*使用*

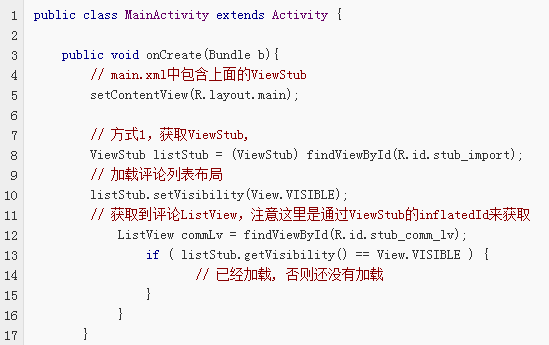
例如我们通过一个ViewStub来惰性加载一个消息流的评论列表，因为一个帖子可能并没有评论，此时我可以不加载这个评论的ListView，只有当有评论时我才把它加载出来，这样就去除了加载ListView带来的资源消耗以及延时，示例如下：



my\_comment\_layout.xml如下:



在运行时，我们只需要控制id为stub\_import的ViewStub的可见性或者调用inflate()函数来控制是否加载这个评论列表即可。示例如下 :



*Tips：通过setVisibility(View.VISIBILITY)来加载评论列表，此时你要获取到评论ListView对象的话，则需要通过findViewById来查找，而这个id并不是ViewStub的id（是inflatedId）。*

*注意*

——判断是否已经加载过， 如果通过setVisibility来加载，那么通过判断可见性即可，如果通过inflate()来加载是不可以通过判断可见性来处理的，而需要使用方式2来进行判断；

——findViewById的问题，注意ViewStub中是否设置了inflatedId，如果设置了则需要通过inflatedId来查找目标布局的根元素；

# 2016-11-21

## 为什么有时候要使用support包里面的控件

***使用情境***

当我们使用NavigationView的时候，我们需要使用

<android.support.design.widget.NavigationView

而不能只是

<NavigationView

这样来引入。

***原因解析***

NavigationView是Android 5.0新提供的控件，也就是说在5.0+以上的手机系统里，自带了这个控件有关的库，此时我们可以只使用：

<NavigationView

来引入，在5.0+系统的手机上也可以正常运行。

但是，一旦我们将同样的app安装在API 5.0系统以下的系统上时，app就不能正常运行了，因为5.0以下的系统没有提供有关这个控件的库。

而如果我们使用的是：

<android.support.design.widget.NavigationView

这个来引入的，此时的app就能兼容5.0系统以下的手机。因为此时虽然手机本身没有提供这个支持库，但是我们为它引入了，从而可以正常运行。

***后记***

Android最新版现在已经是7.0了，而每个版本几乎都会提供几个新的控件，如果我们用到了此类控件，又希望它们能在低版本的系统中兼容，此时，我们就应该使用support支持包里面提供的这些控件（为了向下兼容，Google都会相应的发布一些support包），而非直接引用它们。

***Tips***

目前android手机上经常使用的support包一般是：

compile 'com.android.support:design:24.1.1'  
compile 'com.android.support:support-v4:24.1.1'

如果使用的是AndroidStudio，里面一般会自动引入常用的v7包：

compile 'com.android.support:appcompat-v7:24.1.1'

## Android 5.0、6.0、7.0的新特性及开发者需要注意的地方

***新特性***

**Android 5.0——Lollipop——Android L:**

Material Design设计风格——视觉、交互、动画三个方面来评价Material Design， 顶栏的动画好似一张纸从原本的平面抽离，而其它 的卡片则从四面八方汇聚过来形成一体，Google在 虚拟世界里构建了几个图层，让卡片、按钮等视觉 元素铺在上面，而且还运用阴影以及空间来突出内 容；

Ok Google语音指令——在新系统中，Ok Google语音搜索功能得到了更好的应 用，即便当手机处于待机状态时，对你的手机轻轻说声 “Ok Google”，手机即刻被唤醒；

Face unlock面部解锁——Android系统的面部解锁功能，用起来总是让人感觉不 爽，不过Android 5.0终于有所改善了，当用户拿起手 机处理锁屏界面上的消息通知时，面部解锁功能便自动 被激活；

全新的通知中心设计——谷歌在Android Lollipop中加入了全新风格的通知系统。 改进后的通知系统会优先显示对用户来说比较重要的 信息，而将不太紧急的内容隐藏起来。用户只需要向下 滑动就可以查看全部的通知内容；

新的通知系统另外一个很酷的新功能是在锁屏界面也 可以直接查看通知消息了。不仅如此，用户还可以直接 在锁屏的情况下就行回复或进入应用。另外，如果在操 作手机的过程中有电话进入，也不会进行全画面切换， 而是同样以弹出通知的方式告知用户；

Project Volta电池续航改进计划——Android 5.0 更好的优化了系统的续航，系统 能够根据电池电量来减少处理器功耗、屏幕 亮度等等，能为为用户带来额外90分钟的续 航时间；

**Android 6.0——Marshmallow——Android M：**

Now on Tap——Now on Tap能够让谷歌自动从任何应用中进行搜索。例如当你正 在Whatsapp中聊天的时候提及餐馆，那么可以在不跳转的情况 下进行谷歌搜索。Now on Tap很酷，它与Google的搜索引擎结 合得更加紧密了；

NPR One——这是一个语音交互应用，说出“Okay Google, listen to NPR”，然后 手机就会问你想要什么了；

Doze 和 App Standby——这两是个提升续航的重要（全自动）模式，

Doze 的原理是通过对各种传感器的监控，来判断机器 是否处于使用状态。当 Doze 认为机器长时间处于待 机状态的时候，它会自动暂停绝大部分后台应用，同 时关闭网络连接，仅在需要时偶尔连接，如此来达到 在待机时节省电量的效果；

App Standby，指的是系统会自动停用那些许久不用的 应用，从而达到节省电量和减少后台占用的作用；

权限管理——相比起以前将应用权限一股脑都列出来，让用户决定装还是不装， Android 6.0 的权限管理会在应用需要用到权限的时候弹窗提醒， 让用户进行授权，用户授权后，应用才能得到相应的权限；

另外，在你向一个应用程序授予权限之后，你也可以选择在应用 详情里取消某些权限的授权；

（那么，怎么解决这种动态权限的问题呢？方法有二：

一：动态判断和申请；

二：将targetSDKVersion<23；判断是否是需要运行时权限的标记 就是targetSDKVersion，当targetSDKVersion<23的时候，仅 在安装时赋予权限，使用时将不被提醒，当targetSDKVersion ≥23的时候才会使用新的运行时权限规则。所有在最早遇 见因权限未适配的导致的崩溃的时候，我们可以人为的降 到小于23，这样就变成了还是默认使用权限，但是Google 所推荐使用的是第一种方案）

语音搜索——现在锁屏状态下也可以进行语音搜索，而且你能够从角落滑动进行 激活语音搜索的命令；

内存管理——Android 6.0 将原先存在于「应用」中的内存管理方面的功能单独 提出了，现在可以更加直观地看到平均内存的使用量，还能查看 过去 3 小时到一天的应用的内存使用量，从而判断哪些应用是 「吃内存大户」；

指纹识别——在 6.0 中加入了原生支持，并且对开发者提供了API；

其他——还有些其他的改进，比如说充电指示可告诉你是否处于快充状态，以及 预估的剩余充电时间。“抬头式通知”功能就更简单了，从顶部自动跳 下，再拉下来即可，“应用抽屉”可便于用户快速搜索应用；

**Android 7.0——Nougat——Android N：**

分屏多任务——其实这个功能，之前不少厂商已经做了，而谷歌终于让7.0原生 支持了它，从实际体验来看，户点击多任务按键后，长按其中一 个应用，将其拖拽到屏幕顶部或底部，然后再点击另外一个应用 就可以实现分屏多任务功能；

省流量模式（Data Saver）——Android7.0中引入“省流量”模式，在接近用 户计费周期末，或是流量包本身较小的情况下， 减少应用消耗的数据流量。在启用这一模式时， 系统将拦截后台的数据使用，并在可能的情况下 减少前台运行应用使用的数据量，例如限制你看 视频，下调画质，以及减少缓存等。当然你也可 以对个别应用不进行限制，痛快使用；

新增VR支持——Android7.0将会是谷歌充分执行其VR计划的操作系统，它内置 谷歌的全新VR平台Daydream。Daydream是一个虚拟现实平台， 由Daydream头盔、手柄和智能手机构成，支持Daydream的智 能手机需满足一定的硬件要求。目前，我们所知的就是智能手 机的传感器、屏幕和系统芯片需要达到一定标准，满足了标准 使用将会很畅快；

可以彻底删除原生应用——将不用root也可以实现这个功能；

***开发者需注意的地方***

**Android 5.0——Lollipop——Android L:**

*（http://blog.csdn.net/IJUST/article/details/41252927）*

——

在调用最新的Android 5.0 API的时候要注意在调用前判断系统版本号以兼容之前的API版本，并且不能使用低于minSdkVersion的API，同时，如果使用5.0 API提供的新控件的话，最后从support里面引入，这也是为了兼容之前的API版本。

*（这条在Android 6.0、Android 7.0......中都适用）*

——

设置targetSdkVersion=21；

——

新增Material Design样式的支持，包括以下方面：

—系统自带Material design主题；

—组件阴影；

—新的控件RecyclerView；

—Drawable动画和样式效果。（这里应该是指Ripple Drawable之类）（Drawable animation and styling effects）；

—基于组件状态的Animator。（Animators for view properties based on the state of the view）；

—可定制的UI组件和工具栏（这里指的应该是ToolBar）；

—基于XML的矢量动画和图形（Animated and non-animated drawables）；

**Android 6.0——Marshmallow——Android M：**

*（http://blog.csdn.net/ccpat/article/details/51151863）*

——

设置targetSDKVersion=23（也就是只有当app的targetSDKVersion>=23时才会调用6.0的API）；

——

动态权限分配；（当我们不想做动态权限分配时，我们可以设置targetSDKVersion<23，这样就不会调用6.0的API，所以即使没有作动态权限分配也不会出现异常。但是Google Play现在强制要求所有需要申请Google推荐的app的targetSDKVersion必须为23，所以如果要将应用发布到Google Play且要申请Google推荐的， 就不能这样设置了）

——

新增NavigationView；

——

移除HTTPclient，早在API 22的时候Google已经开始不推荐使用HttpClient，取而代之的是HttpURLConnection，在这次在API 23中直接移除了HttpClient，从Deprecated到Removal可以看出谷歌从Android中废除HttpClient的决心。如果你的项目目前还无法切换到HttpURLConnection，那么下面有两种方式可以让你继续使用HttpClient：

一：target SDK版本选择23以下的版本；

二：手动引入HttpClient支持包，

Android studio项目可以在build文件中添加：

android{

useLibrary 'org.apache.http.legacy'

}

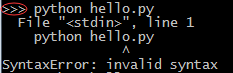
或直接导入org.apache.http.legacy.jar，Eclipse项目直接导入org.apache.http.legacy.jar即可。

**Android 7.0——Nougat——Android N：**

## Python运行出错

***问题描述***

写了一个hello.py，仅有一句，print ('hello world')，在命令行中运行，却提示出错：



***解决过程***

使用的是python3.5,代码中print()函数也是加了括号的，但是一直出现这个问题。

***原因***

一直是在python的命令行中运行hello.py文件的，而不是在系统（Windows）的命令行中运行，所以一直提示错误。（python命令行直接运行python代码，而python文件需要在系统的命令行中运行）



注意两张图的运行环境的区别。

***Tips***

——In the shell you can run shell commands, in the Python command line you can run Python code.

——"python hello.py" is a shell command, not Python code, so you should run it in the shell, but not on the Python command line.

——"print "hello"" is Python code, so you can run it on the Python command line, but not in the shell.

# 2016-11-28

## Python简单爬虫架构

*（http://www.guolanzhe.com/?p=163）*

***说明***

一个简单的爬虫架构，都有哪些方面需要考虑。

***爬虫调度端***

首先，需要有一个爬虫调度段，来启动爬虫，停止爬虫，或者监视爬虫的运行情况。然后，在爬虫中需要实现三个模块。

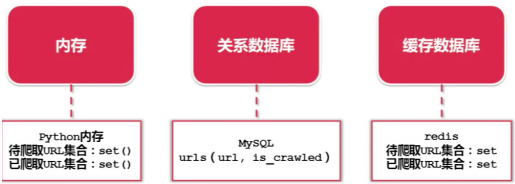
***三个模块***

URL管理器——实现对已经爬取过的URL和待爬取的URL的管理（为了防止重复 抓取和循环抓取）；

**管理过程：**



**实现方式：**



网页下载器——从URL管理器中我们可以取出一个待爬去的URL，将其传送给网 页下载器，下载器会将URL指定的网页下载下来，存储成一个字 符串，这个字符串会传送 给网页解析器进行解析；

**概念：**

只有将网页下载到本地，才能进行后续对数据的分析和处理，因 此，网页下载器是爬虫的不可缺少的核心组件；

网页下载器类似于网络浏览器，会将URL对应的网页，以html 的形式下载到本地，然后存储成一个本地文件或者说本地字符串 然后我们才能进行后续的分析和处理；

**种类：**

urllib2 这是一个python 官方的基础模块，它支持直接的URL下 载，或者说向网页提供一些需要用户输入的东西，甚至支持需要 登录网页的cookie处理，需要代理访问的代理处理等这些高级 功能；

requests 它是一个python第三方的插件，它提供更为强大的功 能；

网页解析器——利用正则表达式等工具对（下载下来的）网页的字符串进行解析， 一方面可以解析出我们所需要的有价值的数据，另一方面可以解 析出该网页相关联的其它url，并将这些URL补充进URL管理器；

**种类：**

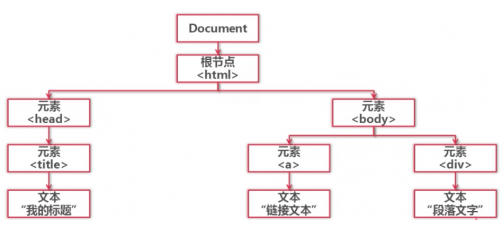
正则表达式——正则表达式会将整个网页当作一个字符串，然后 采用模糊匹配的方式来提取出有价值的信息。虽 然很直观，但是如果文档比较复杂的话， 这种 方式将会非常的麻烦；

html.parser——使用python自带的html.parser 这个模块来解析 网页；

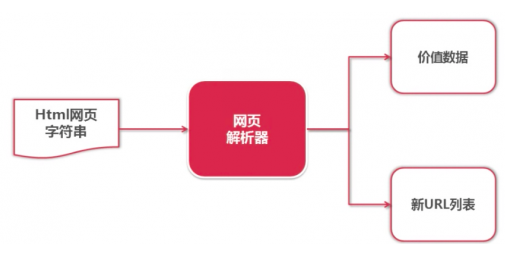
lxml——使用lxml这个第三方插件来解析html网页或者xml网 页；

beautifulsoup——beautifulsoup这个第三方插件既可以使用 python自带 的html.parser来作为它的解析 器，也可以使用lxml来作为它的解析器，相对 来说比较强大；

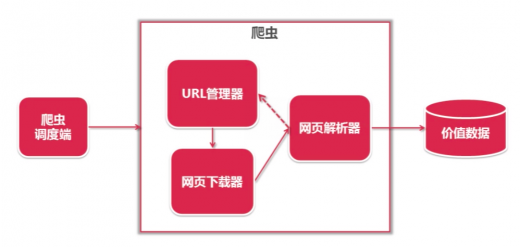
上面提到的四种方法中，正则表达式是基于字符串的模糊匹配， 而另外三种则是基于结构化的解析。那么，什么叫结构化的解析 呢。是将整个html文档下载成一棵树，也就是结构化解析 － DOM(Document Object Model)树。以树的方式来进行元素的遍历 和信息的获取。利用树的上下级关系可以很方便的定位到某个元 素，使我们获取信息更加便利。树的大致结构如下图所示：



**图示：**



***整体图示***



URL管理器，网页下载器，网页解析器这三个模块就形成了一个循环，只要有相关联的URL，爬虫就会一直运行下去。

## 正向代理与反向代理

***说明***

正向代理：代理(v)客户端，为客户端收发请求，使真实的客户端对服务器不可见；

反向代理：代理(v)服务器，为服务器收发请求，使真实的服务器对客户端不可见；

由于服务对象和自身角色的不同，刚好是相反的。

（v代表动词）

***图示***



正向代理中，proxy和client同属一个LAN，对server透明；

反向代理中，proxy和server同属一个LAN，对client透明。

实际上proxy在两种代理中做的事都是代为收发请求和响应，不过从结构上来看正好左右互换了下，所以把后出现的那种代理方式叫成了反向代理。

## 关于Windows的批处理文件.bat

***让.bat文件开机自动执行***

用BAT自己写自己开机运行不可以的、如果要开机自动运行的话把要运行的那个复制到C:\Documents and Settings\Administrator\「开始」菜单\程序\启动。

***让.bat在后台执行***

**IEXPRESS**

IEXPRESS是WINDOWS自带的打包程序,在“运行”中输入IEXPRESS,这是WINDOWS自带的打包程序,点击下一步->下一步->随便填个名字(下一步)->下一步->下一步->ADD(选好你的BAT文件 继续下一步）->Install里面选你刚加进去的文件（下一步）->hidden(下一步)->下一步->保存打包好的文件(勾一下hidden下一步)->下一步->dont save下一步->下一步->完成。这样，当需要的时候，运行打好包的EXE文件即可。

**VBS脚本**

我们知道运行批处理文件过程中会出现DOS窗口，如果我们不想看到这样的窗口的话，可以通过VBS隐藏运行批处理程序，让它在后台运行。.VBS文件中的代码如下：

Set ws = CreateObject("Wscript.Shell")

ws.run "cmd /c run",vbhide

其中的run就是.bat批处理文件的文件名——run.bat。

***让.vbs文件开机自动执行***

参考上面第一条的方法测试一下。

## Python中的Function(函数)与Method(方法)

***说明***

Function——A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body.

Method——A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called self).

***解析***

从定义的角度看：

函数(function)相当于一个数学公式，理论上不与其它东西关系，只需要相关的参数就可以，所以在module中定义的称为函数是很有道理的；

而方法的意思也很明确，它是与某个对象相互关联的，也就是说它的实现与某个对象有关联关系，这就是方法，虽然它的定义方式和函数是一样的。也就是说，在Class中定义的函数就是方法。

***总结***

python程序由包(package)、模块(module)和函数组成。包是由一系列模块组成的集合。模块是处理某一类问题的函数和类的集合；

python中的函数并不是只能依附于类才存在，当一个函数是在模块中而不是依附于类时，这时称为函数，但当他定义在类中，也就是依附于类时，称为方法，但本质上，方法是一个函数。

***简而言之***

属于某个类的函数就是方法；

不属于任何类的函数就是函数；

***Tips***

——方法的第一个参数不一定必须是self；

——直接定义的是函数，可以在py文件的任何地方随意调用；

——类里面定义的是方法，第一个参数是self，需要实例化类以后调用；

## Python中的小细节

——无论是行注释还是段注释，均以#加一个空格来注释；

——如果需要在代码中使用中文注释，必须在python文件的最前面加上如下注 释说明：

# -\* - coding: UTF-8 -\* -

——python中的变量不需要声明，变量的赋值操作即是变量声明和定义的过程；

——python中一次新的赋值，将创建一个新的变量。即使变量的名称相同，变 量的标识并不相同。用id()函数可以获取变量标识；

——python的数字类型分为整型、长整型、浮点型、布尔型、复数类型；

——python没有字符类型；

——python内部没有普通类型，任何类型都是对象；

——如果需要查看变量的类型，可以使用type类，该类可以返回变量的类型或 创建一个新的类型；

——python不支持自增运算符和自减运算符。例如i++/i-是错误的，但i+=1 是可以的；

——1/2在python2.5之前会等于0.5，在python2.5之后会等于0；

——不等于为!=或<>，等于用==表示；

——python程序由包(package)、模块(module)和函数组成。包是由一系列模块 组成的集合，模块是处理某一类问题的函数和类的集合；

——包就是一个完成特定任务的工具箱，包必须含有一个\_\_init\_\_.py文件，它 用于标识当前文件夹是一个包，包的作用是实现程序的重用；

——python的程序是由一个个模块组成的一个文件即是一个模块，模块由代码、 函数和类组成，导入模块使用import语句；

——python使用==和!=来进行字符串比较。如果比较的两个变量的类型不相同， 那么结果必然为不同；

——python用class保留字来定义一个类，类名的首字符要大写。当程序员需 要创建的类型不能用简单类型来表示时，就需要定义类，然后利用定义的类 创建对象；

——类的方法也分为公有方法和私有方法；

——私有函数不能被该类之外的函数调用，私有的方法也不能被外部的类或函数 调用。

——python使用函数”staticmethod()“或”@ staticmethod“指令的方法把 普通的函数转换为静态方法，静态方法相当于全局函数；

——python的构造函数名为\_\_init\_\_，析构函数名为\_\_del\_\_；

——python 默认编码方式的方法是ascii编码，而我们从网上爬取的格式以 UTF-8为主，更改python默认编码方式的方法如下：

import sys

reload(sys)

sys.setdefaultencoding('utf8')

## Python 2.X与3.X中的各种库

***Python 2.X***

urllib:提供一些比较原始基础的函数，这些函数在urllib2中没有提供；

urllib2:可以接受request对象为URL设置头信息、修改用户代理、设置cookie 等，而urllib只能接受一个普通的URL；

所以，一般情况下，两者都是同时导入使用的。

***Python 3.X***

urllib.request:3.X中使用urllib.request取代了2.X中的urllib2；

urllib：在3.X中成了一个包，此包下分成以下几个某块：

——urllib.request:用于打开和读取URL；

——urllib.error:用于处理前面request引起的异常；

——urllib.parse:用于解析URL；

——urllib.robotparser:用于解析robots.txt文件；

***扩展库requests、beautifulsoup4***

**说明**

这两个库不是python内置的，所以想要使用的话，需要另外安装。

**介绍**

requests：是python的一个HTTP客户端库，更urllib、urllib2类似，可以看 成是urllib的升级版本，打包了全部功能并简化了使用方法，所以 python3.X中一般用它来代替urllib；

（文档：<http://docs.python-requests.org/zh_CN/latest/user/quickstart.html>

http://www.zhidaow.com/post/python-requests-install-and-brief-introduction）

BeautifulSoup:是一个可以从HTML或者XML文件提取数据的python库，我们从 网页上爬下来的数据一般是HTML文件，使用它能很方便的获得 文档中你想要的节点、标签或标签属性中的数据，能极大的减少 我们的工作量；

（文档：<http://beautifulsoup.readthedocs.io/zh_CN/v4.4.0/#>

http://beautifulsoup.readthedocs.io/zh\_CN/latest/）

*（Tips:一般使用beautifulsoup来代替python的内置正则表达式库re, 因为当html文件中的标签格式比较复杂时，通过正则获取数据一般比较 麻烦）*

LXML：LXML也是需要我们在安装python后手动安装的，它是一个HTML解析包， 用于辅助beautifulsoup解析网页；

***Python的urllib模块***

这里说的urllib模块，是urllib\*模块，包括了urllib、urllib2和urllib3等几个。urllib和urllib2都是属于python的基础库，而urllib3是一个拓展库，在python2.\*到python3.\*过程中，urllib模块自身也演化了许多，比如命名上已经不再是1,2这样，而是细分成了urllib.request、urllib.error、urllib.parse和urllib.robotparser等。

## Python库BeautifulSoup的使用

***题外话***

Python3.X中，对于一般的从网页爬取数据，使用python的requests、BeautifulSoup外置包来代替urllib、re这样的内置包能减少很多的代码量。

requests——用于获取网页信息；

BeautifulSoup——用于解析信息并获取节点的内容；

eg:

*# -\*- coding: utf-8 -\*-***import** requests  
**from** bs4 **import** BeautifulSoup  
  
headers = {**'User-Agent'**:**'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 '  
 '(KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36'**}  
url = **'http://gold.xitu.io/welcome/android'**html = requests.get(url,headers=headers) *# 获取网页信息*soup = BeautifulSoup(html.text,**'lxml'**) *# 创建beautifulsoup对象soup，参数中：html.text是要解析的文本，lxml是指定用lxml解析器来解析  
#print(soup.prettify())*all\_div = soup.find\_all(**'div'**,class\_=**'entry clearfix'**) *# 找到class为entry clearfix的div，当div较多时需要用class来确定找哪个div***for** div **in** all\_div: *# 对找到的所有div进行遍历* **try**:  
 sub\_div=div.find(**'div'**,class\_=**'entry-title ellipsis'**) *# 有很多子div，所以要用class确定找到哪一种子类的div* sub\_div2=div.find(**'div'**,class\_=**'entry-meta'**) *# 找到class为entry-meta的子div* a=sub\_div2.find(**'a'**) *# 因为sub\_div2底下只有一个a标签，所以不用写class来定位* span=sub\_div2.find(**'span'**) *# sub\_div2底下只有一个span标签，直接写span即可（省去class='...'）* href=a.get(**'href'**) *# 取a标签中的href属性值（a.get\_text()是取a标签的值）,这个表达式等价于 href=a['href']* print(**'收藏数：'**+span.get\_text()+sub\_div.get\_text(),a.get\_text())  
 **except** KeyError:  
 **pass**

***BeautifulSoup***

BeautifulSoup将浮躁的HTML文档转换成一个复杂的树形结构，每个节点都是python对象，所有的对象可归纳为四种：

——Tag

——NavigableString

——BeautifulSoup

——Comment

(http://cuiqingcai.com/1319.html)

# 2016-12-5

## JDBC中为什么使用PreparedStatement代替Statement

***前言***

我们先来看下通常使用jdbc的套路是什么，这里以MySQL为例：

Class.forName(“com.mysql.jdbc.Driver”);//得到驱动

conn = DriverManager.getConnection(url,username,password);//得到连接

//使用statement执行查询语句

stmt = conn.createStatement();

resultSet = stmt.executeQuery(sql);

while(resultSet.next()){......}

//使用preparedStatement执行查询语句

sql=”select \* from table t where t.\*1,t.\*2 values(?,?) ”;//?是占位符

pstmt = conn.preparedStatement(sql);

pstmt.setString(1,args);//第一个占位符的值为args

pstmt.setString(2,args2);//第二个占位符的值为args2

rs=pstmt.executeQuery();//注意不需要使用参数

while(rs.next()){......}

可以看到，使用PreparedStatement的代码比使用Statement要多了几行，那为什么我们还提倡使用PreparedStatement呢？

***原因***

我们虽然多写了几行代码，但是带来的好处却更多，主要有下：

**继承角度**

PreparedStatement接口继承自Statement，后者有的功能前者都有，而且之多不少；

**可读性和可维护性**

我们用代码来更好的说明一下：

//stmt是Statement对象实例

stmt.executeUpdate("insert into tb\_name (col1,col2,col2,col4) values ('"+var1+"','"+var2+"',"+var3+",'"+var4+"')");

//pstmt是 PreparedStatement 对象实例

pstmt =

con.prepareStatement("insert into tb\_name (col1,col2,col2,col4) values (?,?,?,?)");

pstmt.setString(1,var1);

pstmt.setString(2,var2);

pstmt.setString(3,var3);

pstmt.setString(4,var4);

pstmt.executeUpdate();

多了几行代码，但是结构更加清晰，方便理解和维护。

**性能角度**

从前言中提到的如何使用两者我们就能看出来，PreparedStatement会把sql语句进行预编译（sql语句主体都知道了，只是几个参数缺失），每次使用只需传入不同的参数即可；而Statement则是每次执行sql语句时进行编译，当要重复执行某个类型的查询语句多次时，前者会极大的提升性能并节省时间。

**安全性（最重要）**

有时候在做一些web端的开发时，为了防止sql注入，提升安全性PreparedStatement会比Statement方便多得多。因为PreparedStatement对sql语句做了预编译，无论用户怎么输入，都不会威胁安全性；而如果使用Statement，则需要做很多sql语句的判断等等，麻烦不说，还不能保证安全（因为Statement对sql语句是运行时编译）。

例如：

String sql = "select \* from tb\_name where name= '"+varname+"' and passwd='"+varpasswd+"'";

如果我们把[' or '1' = '1]作为varpasswd传入进来.用户名随意,看看会成为什么?

select \* from tb\_name = '随意' and passwd = '' or '1' = '1';

因为'1'='1'肯定成立,所以可以任何通过验证.

更有甚者，把

[';drop table tb\_name;]

作为varpasswd传入进来,则:

select \* from tb\_name = '随意' and passwd = '';drop table tb\_name;有些数据库是不会让你成功的,但也有很多数据库就可以使这些语句得到执行。

而如果你使用预编译语句.你传入的任何内容就不会和原来的语句发生任何匹配的关系.只要全使用预编译语句,你就用不着对传入的数据做任何过虑.而如果使用普通的statement,有可能要对drop,;等做费尽心机的判断和过虑。

***Tips***

——

PreparedStatement是预编译的,对于批量处理可以大大提高效率. 也叫JDBC存储过程；

——

在对数据库只执行一次性存取的时侯，用 Statement 对象进行处理，使用 PreparedStatement 对象的开销比Statement大，对于一次性操作并不会带来额外的好处；

## 将Java程序变成windows服务

***前言***

本文中，我们分三步将一个Java程序变成windows服务。

***Java程序导出为Jar包***

将java程序导出为jar包这一步并不麻烦，可以借助Eclipse等集成工具来实现，具体就不赘述。

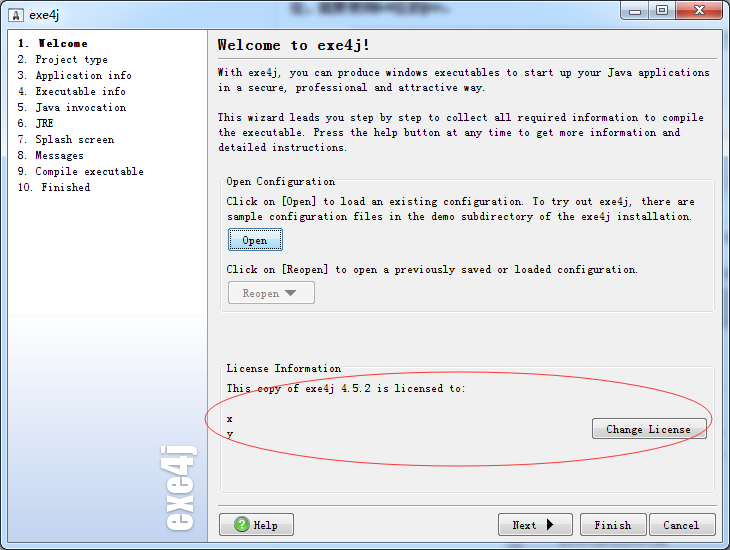
***将Jar包转换成.exe文件***

*（http://www.jb51.net/article/44392.htm）*

其实第一步导出的jar包就可以直接运行，只要电脑中有Java环境即可，但是为了实现最终的变成windows服务，我们需要先将jar包转换成.exe文件。

步骤如下：

1. 下载安装exe4j，打开后界面如下：



这是欢迎页，通常不用做什么，直接next就可以了，但是有时候需要我们去注册一下，就是红圈部分，其中的x,y是随便输入的，还需要注册码，可以从下面选一个填上即可：

A-XVK258563F-1p4lv7mg7sav

A-XVK209982F-1y0i3h4ywx2h1

A-XVK267351F-dpurrhnyarva

A-XVK204432F-1kkoilo1jy2h3r

A-XVK246130F-1l7msieqiwqnq

A-XVK249554F-pllh351kcke50

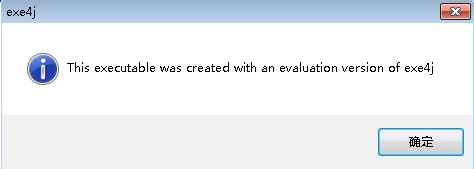
A-XVK238729F-25yn13iea25i

A-XVK222711F-134h5ta8yxbm0

A-XVK275016F-15wjjcbn4tpj

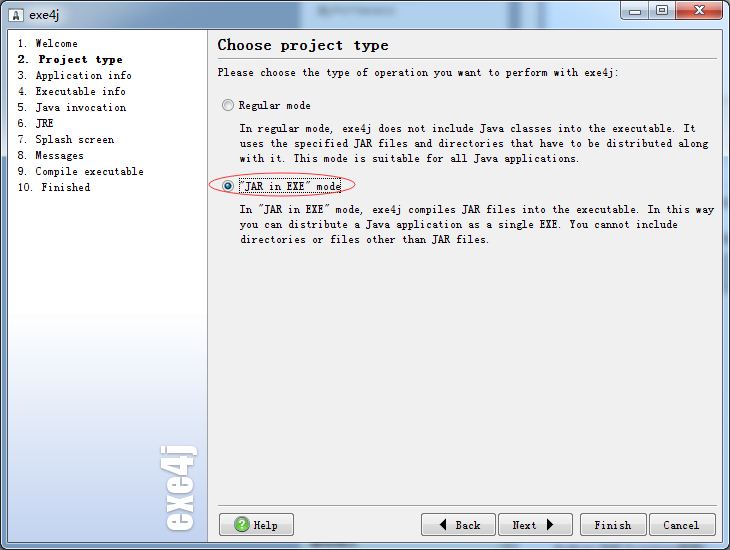
A-XVK275016F-15wjjcbn4tpj

如果我们没有注册就使用的话，通常在点击使用最后生成的.exe文件时会出现如下提示框：

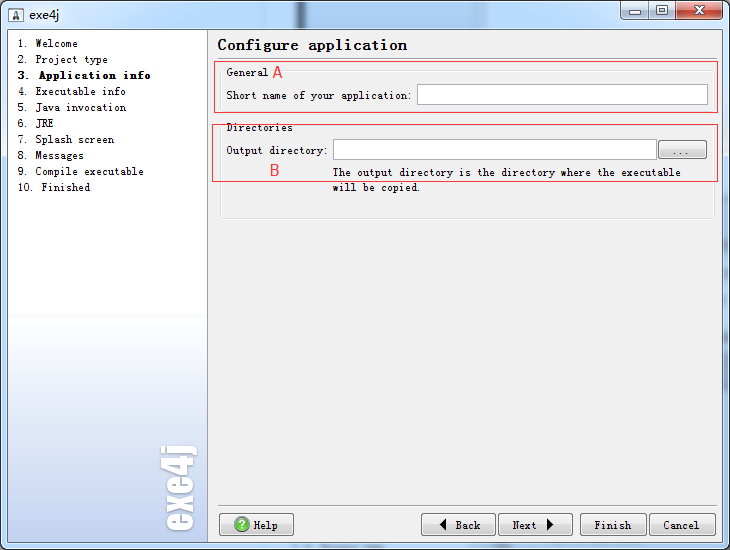


意思是exe4j没有注册，虽然程序最后也可以正常运行，但是这样的提示框的用户体验非常差，所以最好还是注册一下再使用。

1. 点击next之后，进入第二个界面：



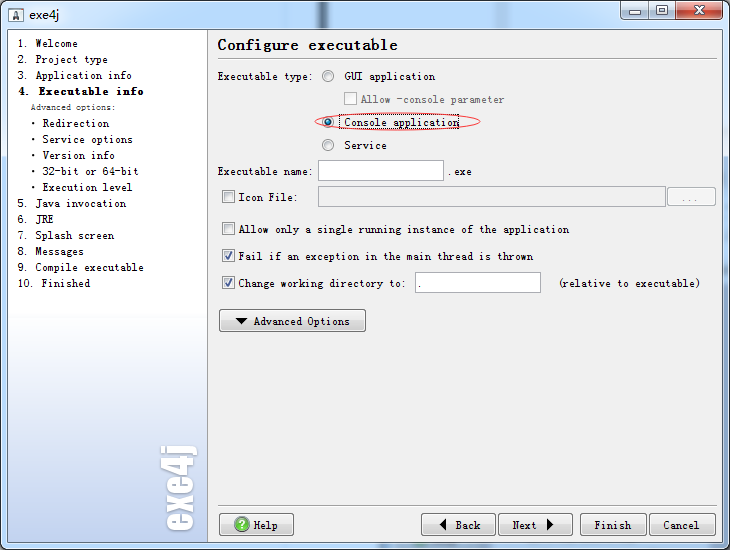
选择第二项”’JAR in EXE’mode”,点击”next”；

3.

这里的A框是输入的应用名字，就是最后转换成的.exe文件的名字；B框是.exe文件的导出位置。

一般的，我们会单独的新建一个文件夹，里面存放原始的jar文件、jre文件夹和导出后的.exe文件。（为什么要放jre文件夹呢，请接着看。）

1. 上面都填写好之后，点击“next”进入界面4：



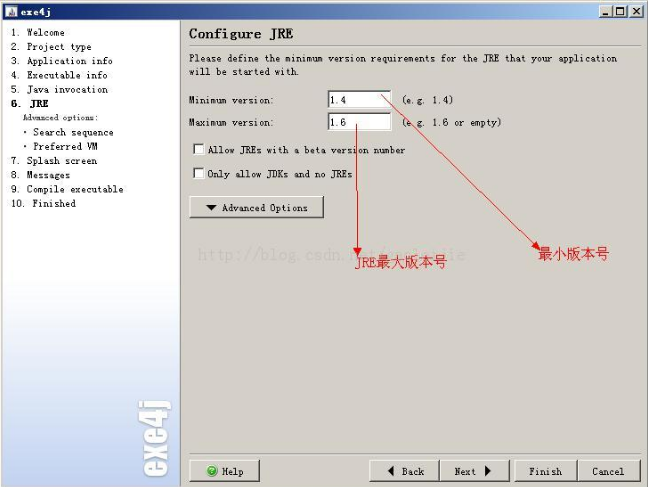
根据你的程序选择不同的type，我们这里选择“Console application”，同时需要在下面的“Executable name中输入一个名字”，“Advanced Options”选项是选择.exe文件运行，然后点击“next”；

1. 进入界面5：



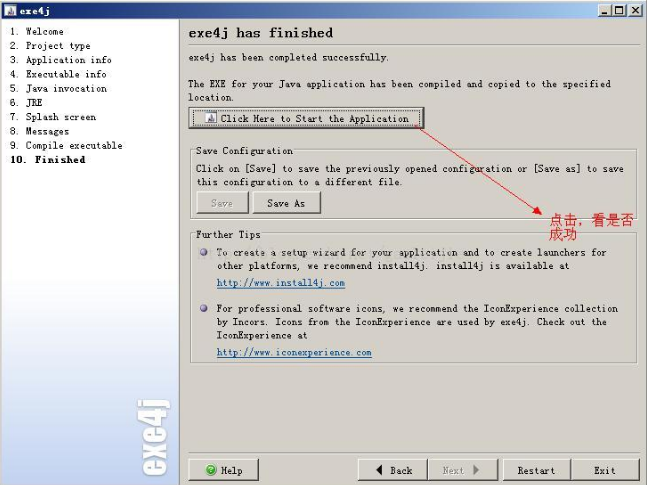
先点击绿色“+”按钮选择Class Path，一般是需要被转换的jar文件，有时可能也需要其他外部的jar包作支持，比如要用到第三方库；接着选择主class，就是包含main方法的类；

1. 点击“next”继续：

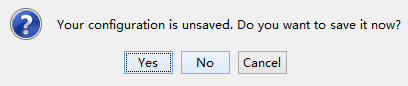


设置大小版本号以后，点击右边节点的”Search sequence”，接着点击 “ 绿色加号” 选择”Directory“添加JRE，JRE是第一步创建文件夹里的JRE。

7.然后“7、8、9”步骤默认就好，第十步时，按图片操作：



然后直接点击“Exit”退出即可，这里会弹出对话框询问是否保存生成过程：



可以根据需要进行选择。

*Tips*

*我们说一下为什么要将步骤三提到的三个文件放一个单独的文件夹，首先为了更好的管理，第二放入jre文件是因为不一定每个电脑都有java环境，我们的java 程序是需要java环境才能执行的，放入jre文件夹之后，即使目标电脑上没有配置java环境也能运行.exe文件。*

到这里，第二大步将jar文件转换成.exe文件就完成了。

***将.exe文件转换成windows服务***

**方法一：**

这里我们使用windows的命令行sc命令来实现这个功能。

关于sc命令的详解，可以在命令行中使用”sc /?”来查看，在此只简单介绍如何使用其加入和删除系统服务：

加入服务：

sc create ServiceName binPath= 路径 start= auto

这里的ServiceName就是转换后的服务名字，可以按照你想要的写一个，后面的路径需要指定到具体盘符底下的具体.exe文件，例如：

sc create pms binPath= F:\pms\_service\pms.exe start= auto

需要注意一点，在上面的这两个“=”后面必须几个空格才能正确设置。（.exe文件也可以换成.bat文件）。

删除服务：

sc delete pms

**方法二：**

在第一种方法失败的情况下，我们可以使用instsrv.exe和srvany.exe这两个windows小工具（需要下载）来实现。

## 数据库中如何清空表并将自增归零

***前言***

在开发过程中，有时会需要将一个表数据清空重新开始记录，例如表中的id列为自增，在重新添加数据的时候，id并不是从1开始，而是接着原本的id开始记录。

***如何解决***

**MySQL等数据库中**

方法1：

此类数据库中，提供了如何实现此功能的关键字——truncate，用法如下：

Mysql—>truncate tablename;

方法2：

Mysql—>alter table table\_name auto\_increment=1;

**Sqlite数据库中**

方法1：

Sqlite中不支持上面的方法1，因为没有truncate这个关键字，因此可以用上面的方法2尝试一下；

方法2：

当SQLite数据库中包含自增列时，会自动建立一个名为sqlite\_sequence的表，可以先使用delete语句清空数据（此时id并不会重新从1开始），接着通过：

delete from sqlite\_sequence;

将表的自增列归零，这样id就会从1开始重新计算。

*Tip：还有一个终极解决办法，适用各种数据库，就是先drop表，再重新建表。*

# 2016-12-12

## 使用Builder代替构造方法

*(https://hjdzone.github.io/blog/2016/11/04/how-to-improve-code-readability/)*

***目的***

提高代码可读性，减少代码量，降低维护成本。

***使用背景***

当某个类的构造方法有很多个参数或者有很多个重载版本时，我们应该考虑为这个类写一个 Builder，通过这个 Builder 创建配置并创建该类的实例。

***如何使用***

假设我们有个类用来代表一个矩形，它的名字叫做 Rectangle，它的代码如下所示：

public class Rectangle {

private int mId;

private int mWidth;

private int mHeight;

private int mStroke;// 边框宽度

public Rectangle(int id) { ... }

public Rectangle(int id, int width, int height) { ... }

public Rectangle(int id, int width, int height, int stroke) { ... }

}

于是我们就可能看见这样的代码 ：

new Rectangle(1, 1, 1, 1)，

这样的代码可读性是很差的，因为我们无法一眼就看出这个矩形设置了哪些信息，还需要去查阅下相关的 API 文档。此外，当我们需要创建一个只需指定 id 和 stroke 的矩形的时候，我们就必须再写一个新的构造方法，当一个对象的属性较多的时候，构造方法的重载版本就可能变得非常的多，维护成本也随之提高。如果我们为 Rectangle 创建一个 Builder，通过 Builder 创建矩形实例的过程就会变得灵活而清晰很多：

public class Rectangle {

private int mId;

private int mWidth;

private int mHeight;

private int mStroke;// 边框宽度

private Rectangle(Builder builder) {// 构造方法私有，因此只能通过builder来

mId = builder.id; // 创建实例

mWidth = builder.width;

mHeight = builder.height;

mStroke = builder.stroke;

}

public static class Builder { //静态内部类（静态类只能是内部类）

private int id;

private int width;

private int height;

private int stroke;// 边框宽度

public Builder(int id) { // Builder类的构造方法

this.id = id;

}

public Rectangle build() {

return new Rectangle(this);// 因为Rectangle类的构造方法私有，因此只能

// 在其内部类中调用（外部类无法调用）它的私有 // 构造方法,来new出一个rectangle的实例

}

public Builder width(int width) {

this.width = width;

return this;

}

public Builder height(int height) {

this.height = height;

return this;

}

public Builder stroke(int stroke) {

this.stroke = stroke;

return this;

}

}

}

现在，我们创建 Rectangle 实例的过程就会变成下面的样子，它不仅让我们一眼就看出矩形设置了哪些属性，而且我们还可以自由组合这些属性，达到重载构造方法想要的结果：

Rectangle.Builder builder1 = new Rectangle.Builder(1);// new一个Buildr实例buildr1， //它的id为1；

Rectangle rect1 = builder1.width(1).height(1).stroke(1).build();//调用Builder类 //的各种方法来设置Rectangle //的各个属性，最后调用.build() //方法返回一个Rectangle的实例

Rectangle.Builder builder2 = new Rectangle.Builder(2);// new一个Buildr实例buildr2， //它的id为2；

Rectangle rect2 = builder2.width(1).height(1).build();//调用Builder类 //的各种方法来设置Rectangle //的各个属性，最后调用.build() //方法返回一个Rectangle的实例

Rectangle.Builder builder3 = new Rectangle.Builder(3);// new一个Buildr实例buildr3， //它的id为3；

Rectangle rect3 = builder3.stroke(1).build();//调用Builder类 //的各种方法来设置Rectangle //的各个属性，最后调用.build() //方法返回一个Rectangle的实例

# 2016-12-19

## Mysql中执行delete和update出现的问题

***问题描述***

使用软件：MySQL Workbench

提示内容：“You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column To disable safe mode, toggle the option in Preferences -> SQL Editor -> Query Editor and reconnect.”

***原因分析***

因为MySQL运行在safe-updates模式下，该模式会导致非主键条件下无法执行update或delete命令。

通俗的说，就是在此模式下的update和delete语句中，如果表中的主键不是作为where的条件，那么久不能执行。

***解决方法***

执行命令

SET SQL\_SAFE\_UPDATES = 0;

修改下数据库模式

***Tips***

设置safe-updates模式是为了提高数据库安全等级，上面解决方法最然解决了问题，但是也降低了安全等级，我们可以通过如下命令来恢复：

SET SQL\_SAFE\_UPDATES = 1；

执行此命令成功后，以delete SQL语句为例，非主键情况下又报错了，说明安全等级修改成功。

# 2016-12-26

## Android中使用cursor的注意点

***问题描述***

代码片段如下：

while(cursor.moveToNext()){

cursor.getCount();

}

这样写，总是报cursor.getCount()这一行出错，将代码改为：

cursor.getCount();

while(cursor.moveToNext()){

...

}

则不报错。

***原因分析***

原因是在while循环中，针对的是每一条查询出来的记录，所以当你用cursor.getCount()来取总的查询记录条数时，其实是取不到的，只有在while循环外部，才能取得。

***Tips***

SQLite数据库中对结果集的用法如此，在其他数据库中的用法也要注意。

## android.content.res.Resources$NotFoundException

***错误描述***

Android.content.res.Resources$NotFoundException: String resource ID #0x1

***原因分析***

一般是用来错误的重载方法，android中的方法一般都有很多重载，例如：

textView.setText(CharSequence text);

textView.setText(int resId);

如果不小心将一个int值传给了它，那它不会显示该int值，而是跑到工程下去找一个对应的resource的id,那肯定是找不到的，于是就报错了。

比如这样：

count.setText(incall.getCount());

incall.getCount()返回的是一个int值,直接setText就出出现上面的错误类型；

***解决方法***

要怎么样才能显示这个int值呢？解决方法有两种：

count.setText(String.valueOf(incall.getCount()));

或者

count.setText(incall.getCount() + "");

***Tips***

安卓对于数字很多时候当做资源的id而不仅仅是数字，当用到数字作为参数时要留意，比如这里的setText(\*\*)方法。

## Android中多进程（单个app中）

***使用场景***

——常驻后台任务应用

例如音乐类、计步类、手机管家类等需要长时间在后台运行的应用。

——多模块应用

比如应用大而全，里面肯定会有很多模块，假如有地图模块、大图 浏览、 自定义WebView等等（这些都是吃内存大户），还会有一些诸如下载服务， 监控服务等等，一个成熟的应用一定是多模块化的。

***优点***

——解决应用的OOM问题

Android对内存的限制是针对于进程的，这个阈值可以是48M、24M、16M 等，视机型而定。所以，当我们需要进行加载大图之类的耗内存操作时， 可以在新的进程中去执行，避免主进程OOM；

——有效、合理的利用内存

我们可以在适当的时候生成新的进程，在不需要的时候及时杀掉，合理分 配，提升用户体验，减少app被杀掉的风险；

——单一进程崩溃不影响整体应用的使用

例如我在图片浏览进程打开了一个过大的图片，java heap 申请内存失败， 但是不影响我主进程的使用，而且，还能通过监控进程，将这个错误上报 给系统，告知他在什么机型、环境下、产生了什么样的Bug，提升用户体验。

——方便团队之间的协作开发

当我们的应用开发越来越大，模块越来越多，团队规模也越来越大，协作 开发也是个很麻烦的事情。项目解耦，模块化，是这阶段的目标。通过模 块解耦，开辟新的进程，独立的JVM，来达到数据解耦目的。模块之间互 不干预，团队并行开发，责任分工也明确。至于模块化开发与多进程的结 合，后续会 写一篇专门的文章来研究这个问题。

# 2017-1-3

## 数据库中外键使用与否的讨论

***前言***

近年来，关于数据库中是否使用外键有很大的讨论，两种意见的观点主要如下。

***使用外键***

——数据一致性

由数据库自身保证数据一致性，完整性，更可靠，因为程序很难100％保证 数据 的完整性，而用外键即使在数据库服务器当机或者出现其他问题的时 候，也能够最大限度的保证数据的一致性和完整性。

eg：数据库和应用是一对多的关系，Ａ应用会维护他那部分数据的完整性， 系统一变大时，增加了Ｂ应用，Ａ和Ｂ两个应用也许是不同的开发团队来做 的。他们如何协调保证数据的完整性，而且一年以后如果又增加了C应用呢；

——ER图可靠性

有主外键的数据库设计可以增加ER图的可读性，这点在数据库设计时非常 重要；

——设计全面性

外键在一定程度上说明的业务逻辑，会使设计周到具体全面；

——级联性能未必最低：

除非能证明触发器的性能和可维护性比外键更优，否则凭什么否定外 键；

——安全性

使用应用程序来维护数据完整性，则是严重降低数据库安全性的一种做法。 连接字符串里如果有uid和pwd或者类似的这两个东西，可以使用工具获取 你数据库的帐号。因为每次建立数据库联接的时候都会在网线上以标准格式 传递这些信息。从此编码内的任何保证都是空的

（依靠代码提供的安全性不如数据库中外键提供的安全性稳定严谨）；

——冗余性

不使用外键，会导致数据冗余，在级联最底层的表可能会重复好几层的数据 必然导致最底层的表数据量翻倍，IO瓶颈是数据库性能瓶颈之一；

***不用外键***

——程序逻辑性

某些程序逻辑中，程序的逻辑已经足够保证完整性，我会在存储过程或包等 地方做严谨的判断；

——性能

这是很多人不喜欢用的关键原因，比如一个业务流水表，频繁插入数据，如 果这个表身上有3外键，那么每次插入一条，就必须对这3个外键对应的3 个表做相应的查找判断有无对应数据，如果这3个表也很大，那就这3个表 的判断时间就很常，虽然外键指向的关联表的字段肯定是索引，但是我觉得 很多时候，这样的判断本来就在程序里控制好了，通过外键再判断一次，就 是降低性能；而且其实有的地方判不判断也无所谓的，但是用了外键，就必 须化时间去判断，无论oracle内部多么优化外键对于数据的检索速度，它总 是一个不小的消耗；

——可维护性

很多公司的软件都是定制的，这种定制的东西，随意性相对较大，项目开发 实施过程中，需要经常对表修修补补；还有就是业务逻辑有bug或者其他情 况，需要经常手工维护数据，有错综复杂的外键关联着，很是麻烦；

——外键定死了先后生成关系：

外键定死了两个表之间数据的先后生成关系，最常见的是单据主从表，有的 时候，在生成单据的时候，是先生成明细，再生成主表；如果钉死了外键， 这个就没法实现；

***总结***

——

在大型系统中（性能要求不高，安全要求高），使用外键；

在大型系统中（性能要求高，安全自己控制），不用外键；

小系统随便，最好用外键；

——

用外键要适当，不能过分追求；

——

不用外键而用程序控制数据一致性和完整性时，应该写一层来保证，然后个个应用通过这个层来访问数据库。

## 用flag替换boolean来优化代码（按位与“&”的巧妙使用）

***前言***

一般情况下，一款APP都会有config接口，用于获取一些常量文案，通用配置等信息，会有很多类似开关的字段，如："isNew"，"isVip"，"isShowBalance"等等。如下：

{

"isNew":"1",// 是否是新用户

"isVip":"1",// 是否是VIP用户

"isShowBalance":"1",//是否显示侧边栏余额模块

}

***如何优化***

通过二进制第1位表示"isNew"，二进制第2位表示"isVip"，二进制第3位表示"isShowBalance"。如果有其他新增状态，不需要新增字段，只需要改变返回的数据即可。如下：

{

"flag":"7"// 二进制：111，表示3个状态都为true

"flag":"5"// 二进制：101，表示isNew，isShowBalance为true，isVip为 false

}

客户端在解析的时候，通过按位与“&”来判断对应"isNew"，"isVip"，"isShowBalance"的状态，代码如下：

long flag = 5;

System.out.println("bit=" + Long.toBinaryString(flag));

System.out.println("isNew=" + ((flag & 1) == 1)); //1的2进制表示为1

System.out.println("isVip=" + ((flag & 2) == 2)); //2的2进制表示为10

System.out.println("isShowBalance=" + ((flag & 4) == 4));//4的2进制表示为100

bit=101

isNew=true

isVip=false

isShowBalance=true

***补充***

按位与：两位同时为“1”，结果才为“1”，否则为0；

按位或：参加运算的两个对象只要有一个为1，其值为1；

异 或：参加运算的两个对象，如果两个相应位为“异”（值不同），则该位结 果为1，否则为0；

## SQL语句中的where 1 = 1与where 1 = 2

***前言***

我们经常能看到代码里面的SQL语句中包含这样的条件where 1=1或where 1=2，那么这样写的原因是什么呢。

***解惑***

**where 1 = 1的使用**

使用where 1 = 1是为了方便能够在sql语句中动态增加条件，例如：

string MySqlStr=”select \* from table where”；

if(Age.Text.Lenght>0) // 条件一

{

MySqlStr=MySqlStr+“Age=“+“’Age.Text’“；

}

if(Address.Text.Lenght>0) // 条件二

{

MySqlStr=MySqlStr+“and Address=“+“’Address.Text’“；

}

其中的Age、Address为获取的用户输入的字段，要将它们作为条件加入sql中，现在考虑以下几种情况：

情况一：

用户输入Age为18，Address为HF，那么条件一、二都成立，拼接和的sql语句为：

select \* from table where Age=’18’ and Address=’HF’

可以看出，这是一条完整的SQL查询语句，能够正确的被执行，并根据数据库是否存在记录，返回数据。

情况二：

用户没有输入Age值，输入Address值为HF，拼接后的sql语句为：

select \* from table **where and** Address=’HF’

可以看出，此sql语句存在语法错误，在执行代码的过程中会报错。

情况三：Age值和Address值用户都没有输入，拼接后的sql语句为：

select \* from table **where**

显然，此sql语句也存在语法错误，在执行过程中肯定也会报错。

为了解决这些情况下的报错，我们使用了where 1 = 1 的这种“江湖手段”，现在我们来看看使用where 1 = 1之后，上述三种情况下还会不会报错：

string MySqlStr=”select \* from table **where 1 = 1**”；

if(Age.Text.Lenght>0) // 条件一

{

MySqlStr=MySqlStr+“**and** Age=“+“’Age.Text’“；

}

if(Address.Text.Lenght>0) // 条件二

{

MySqlStr=MySqlStr+“and Address=“+“’Address.Text’“；

}

使用之后，三种情况下拼接后的sql语句分别为：

select \* from table **where** **1 = 1** andAge=’18’ and Address=’HF’

select \* from table **where 1 = 1** and Address=’HF’

select \* from table **where 1 = 1**

可以看到，无论哪种情况下的sql语句都是正确的语法，都会正确的执行。

说到这里，相信你应该明白了，where 1=1的应用，不是什么高级的应用，也不是所谓的智能化的构造，仅仅只是为了**满足多条件查询**页面中不确定的各种因素而采用的一种构造一条正确能运行的**动态（拼接）SQL语句**的一种方法。

**where 1 = 2的使用**

使用这个条件，在SQL中是一定不会出现结果的，通常是在需要获得一个空的表结构时候使用。

where 1=2 只得到表的结构，不获取表的数据，这样节省了内存，因为可以不用保存结果集，而且速度较快，因为它不用和表内任何字段比对。

通常where 1 = 2写成where 1 < >2或者是where 1 = 0,达到的效果是一样的。

## Android市场是如何区分不同的app的

Android系统上是**以包名来区分不同的App**的，同一个App如果在不同渠道使用不同的包名，那么在同一台设备上是可以同时安装多个市场渠道的包的，尽管这些App都是同一个App)。所以app市场上有很多名字相同的app，但其实是不同的app，因为他们的包名不同。

如果包名相同，会提示：

“对不起，您上传的APK包名(Package Name)已存在!”，

从而不让你发布。

还有一种可能出现这种提示的原因：

你的APP在其他市场提交过，但没在当前发布的市场（比如应用宝）提交的话，应用宝会从别的市场抓你们的包。这样你再在应用宝提交的话 就会提示已存在。

现在大部分安卓应用市场都存在这种抓包问题，搞的人很烦躁。更烦躁的是，比如你在A市场提交了APP的独家首发，B市场很可能就抓取过来了，等A市场

的独家首发时间过了，你准备在B市场更新的时候，B市场会提醒你，你提交的产品版本号低于B市场的线上版本，更新失败。。。

淘宝手机助手也抓包，但处理的还不算特别流氓，淘宝手机助手会抓取你产品在市面上最新的版本，当你在淘宝手机助手准备提交的时候，只需要把线上他们抓取的最新版本删除掉，再上传就ok了。

说到这很多人就会有疑问，如果有人恶意做了一个和微信内容一模一样的App，名字和包名都和微信一样，那怎么分辨呢？Android中为了防止这样，还做了一个叫签名的东西，微信的签名一般我们是没有的，所以就可以**通过签名来标识出这款App到底是谁的**了，有兴趣了可以了解下签名，这里就不再赘述了。

# 2017-1-9

## 经纬度的一些知识

***前言***

赤道半径长度： 6,378.2千米

极半径长度： 6,356.8千米

所以赤道周长（即0度纬线周长）为2πR≈40075km,

子午线周长为2πR≈40009km。

***概念***

纬线周长的 长度 随着 纬度 的变化而变化，

经度1度 的长度 随着 纬度 的变化而变化，

（因为经度1度的长度其实是在纬线周长除以360）

（纬度越高，经度1度的长度越短，所以赤道上的经度1度距离最长）

***计算***

一个经度单位长度=（一个纬度单位的长度）乘以（该地区纬度的余弦值）

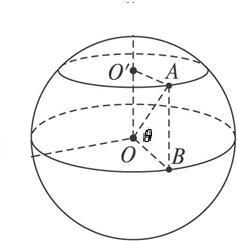
例如：

1°的纬线长=111km；（40009/360=111.136km，单位纬线的长度基本不变，与 经纬度无关）

1°的经线长=111xcosa；（a指对应纬线的纬度，单位经线的长度与纬度有关）

北/南纬 30°的纬线周长=360x111xcos30=30x111x0.866=34605.36km

北/南纬 60°的纬线周长=360x111xcos60=360x111x0.5=20000km



以安徽省（30o,117o）为例，安徽地区的：

1秒经线长：34605.36/360/60/60=0.0267km=26.7m，

（1秒经线的长度差很多，但是不同经线的周长差不多）

1秒纬线长：111.136/60/60=0.0308km=30.8m

（1秒纬线的长度都差不多，但是不同纬线的周长差很多）

***Tips***

经纬度十进制表示法**转换**为度数表示法：

经纬度十进制的的整数部分就是度数(°)，

小数部分乘以60得到的数取整数部分就是分数(′)，

再用该数的小数部分乘以60就是秒数(″)。

例如：

一个经度的十进制为:10.121806，转换成度分秒为:10°7′18.501611351966858″

转换过程：

第一步：度数(°)---10°；

第二步：分数(′)---0.121806×60=7.308360189199448，取整数部分为7，做为分数；

第三步:秒数(″)---0.30836018919944763×60=18.501611351966858，做为秒数；

即度分秒为10°7′18.501611351966858″。

同样的，如果需要将度数转为十进制，将上述算法过程倒过来即可。

## 为什么使用Unix时间戳

***定义***

从1970年1月1日（UTC/GMT的午夜）开始所经过的秒数，不考虑闰秒。

UTC（Coordinated Universal Time）：协调世界时；

GMT（Greenwich Mean Time）：格林尼治标准时间；

Unix时间戳0用UTC/GMT表示为1970/1/1 0:0:0，

用北京时间表示为1970/1/1 8:0:0，因为北京位于东八区。

（全球分为24个时区 每个时区之间相差一小时）

***原因***

——

无时区干扰，表示绝对时间，不受服务器所在时区的影响；

——

存储空间小，用数字储存节省空间，一个数字比一个字符串占用空间小得多；

——

方便计算，比如计算两个时间相差多少秒，很多程序都要用到时间差值来定时执行任务；

——

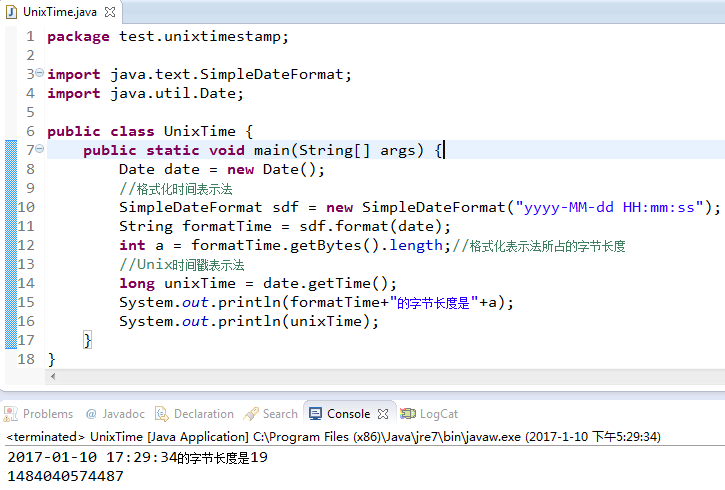
可以更好的跨平台，在现在的系统中经常遇到跨数据库的应用开发，在数据库系统中不同的数据库对与时间类型却有不同解释，因此在时间方面，我们就可以使用记录UNIX时间戳的方法做到垮平台性。

——

检索效率高；

***Tips***

解释一下为什么Unix时间戳的存储空间小，代码说明：



注意到，代码中用格式化表示时间得到的是字符串类型，占用19个字节的内存，而Unix时间戳的表示法得到的是基本类型long，占用8个字节的内存，可见大大节省了存储空间。（是字节Byte不是位Bit）

## 使用for(int i= 100;i>0;i--;)代替for(int i = 0;i <100;i++)

***前言***

在网上的看到一种说法是：

“用for(int i= 100;i>0;i--;)去代替for(int i = 0;i <100;i++) 这样可以提高程序的运行效率”

***原因***

有关解释主要分为以下几点：

——

跟0比较有直接的指令来处理，跟非零的数比较需要先做个减法，再跟0比较，当循环次数多了，意味着比较的次数就多，所以要节省时间；

——

举个代码的例子

List list = new ArrayList(); //假定现在Size为5000

for(int i = 0 ; i < list.Size() ; i++){

System.out.println("List size i=="+i);

}

for(int i = list.Size() ; i>0 ; i-- ){

System.out.println("List size i=="+i);

}

执行过程中，会发现下面的for会比上面的快，

因为第一个for每次循环都需要执行一次Size()方法，而第二个循环只执行了一次Size()方法，然后跟java基本类型作的对比，所以更快；

——

从字节码的角度来说

public void test1(){

for(int i = 0;i<100;i++){

}

}

public void test2(){

for(int i =100;i>0;i--){

}

}

其字节码代码如下

public void test1();

Code:

0: iconst\_0

1: istore\_1

2: iload\_1

3: bipush 100

5: if\_icmpge 14

8: iinc 1, 1

11: goto 2

14: return

public void test2();

Code:

0: bipush 100

2: istore\_1

3: iload\_1

4: ifle 13

7: iinc 1, -1

10: goto 3

13: return

}

前者判断循环条件：

iload\_1

bipush 100

if\_icmpge 14

而后者只需要

iload\_1

ifle 13

因此，后者执行起来更有效率。

***不同意见***

当然也有人认为没有替换的必要，他们认为实际见过的代码中绝大部分是

for(int i = 0;i <100;i++)

这种，大部分情况下，可读性比这一条指令的优化更重要。

***Tips***

读到这里，不知道细心的你有没有发现一个问题：

上面的两种循环其实是不对等的，正确的应该是这样

“for(int i= 99;i>=0;i--;) 代替 for(int i = 0;i <100;i++)”

## Java编码中的一些优化技巧

——

if (result.size() > 0) return true;

return false;

优化为

return return result.size()>0

——

variable.eqauls("const")

优化为

"const".eqauls(variable) 来避免null point exception

——

能用常量的东西都要用常量来完成，避免使用硬编码,增加可维护性质。

比如少用 String str = "123"

使用private static final CONST = "123" ; String str = CONST;

代码中尽量少出现"123".equals(str)

这样的危险字符，要用public static final String STR = "123"; STR.equals(str)去代替；

——

在进行多异常捕获的时候，最后建议加上Exception异常做没有考虑到的异常捕获，比如

try{}

catch(OtherException e){

}

catch(Exception e){

}

finally{

if(conn!=null){

conn.colse();

conn==null;

}

}

——

在使用字串的拼接的时候,建议使用StringBuffer代替String（使用String会多出很多String对象）。

避免使用String = "str"+"str2";

而使用StringBuffer str = new StringBuffer("str");str.append("str2")代替；

——

使用Integer.valueOf()代替new Integer()，因为前者用到了缓存机制，例如：

Integer a=new Integer(1)

Integer a=Integer.valueOf(1);

两个都是得到一个Integer对象a，那究竟有什么不同呢，我们来看一下Integer类的源码（JDK 1.8）：

构造方法Integer(\*\*)：

**public** Integer(**int** value) {

**this**.value = value;

}

静态方法valueOf(\*\*)：

**public** **static** Integer valueOf(**int** i) {

**if** (i >= IntegerCache.*low* && i <= IntegerCache.*high*)

**return** IntegerCache.*cache*[i + (-IntegerCache.*low*)];

**return** **new** Integer(i);

}

这个方法内部用到了Integer类的内部类IntegerCache，我们在看下IntegerCache的源码：

**private** **static** **class** IntegerCache {

**static** **final** **int** *low* = -128;

**static** **final** **int** *high*;

**static** **final** Integer *cache*[];

**static** {

// high value may be configured by property

**int** h = 127;

String integerCacheHighPropValue = sun.misc.VM.*getSavedProperty*("java.lang.Integer.IntegerCache.high");

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

**try** {

**int** i = *parseInt*(integerCacheHighPropValue);

i = Math.*max*(i, 127);

// Maximum array size is Integer.MAX\_VALUE

h = Math.*min*(i, Integer.*MAX\_VALUE* - (-*low*) -1);

} **catch**( NumberFormatException nfe) {

// If the property cannot be parsed into an int, ignore it.

}

}

*high* = h;

*cache* = **new** Integer[(*high* - *low*) + 1];

**int** j = *low*;

**for**(**int** k = 0; k < *cache*.length; k++)

*cache*[k] = **new** Integer(j++);

// range [-128, 127] must be interned (JLS7 5.1.7)

**assert** IntegerCache.*high* >= 127;

}

**private** IntegerCache() {}

}

可以看到在IntegerCache中已经初始化了cache数组。所以，要注意的是，new Integer返回的永远是不同的对象，但是当整数范围在-128<i<=127时，Integer.valueOf返回的是同一个对象。

举例说明：

Integer a=new Integer(1);

Integer b=new Integer(1);

//整数范围在-128到127之间，返回的是同一个对象

Integer c=Integer.valueOf(1);

Integer d=Integer.valueOf(1);

//返回的不是同一个对象

Integer e=Integer.valueOf(200);

Integer f=Integer.valueOf(200);

System.out.println(a==b);

System.out.println(c==d);

System.out.println(e==f);

程序运行的结果为：

false

true

false

由此，使用Integer.valueOf()的效率要高于直接new Integer()。

# 2017-1-16

## Java中的访问修饰符

Java访问修饰符包括private，default，protected和public，权限级别**从高到低**为：

private > default > protected > public

**分别表示:**

private ——私有的访问权限，也是最严格的访问权限，仅只能在设置了该权限的类中访问，利用这个访问权限，表现出封装思想；

default ——默认的访问权限，也是可以省略的访问权限，它不仅能在设置了该权限的类中访问，也可以在同一包中的类或子类中访问；

protected——受保护的访问权限，它除了具有default的访问权限外，还可以在不同包中所继承的子类访问；

public ——公有的访问权限，也是最宽松的访问权限，不仅可以是同一个类或子类，还是同一个包中的类或子类，又还是不同包中的类或子类，都可以访问；

要想更深入地理解这些访问权限，一方面需要明白各自的作用域，另一方面更是要弄清楚各自所体现出来的编程思想，比方说，prvite表现的封装思想，public表现的面向接口编程的思想等。

## ARGB\_8888、ALPHA\_8、ARGB\_4444、RGB\_565的理解

***前言***

A：透明度

R：红色

G：绿

B：蓝

***解释***

——Bitmap.Config ARGB\_4444

每个像素占四位，即A=4，R=4，G=4，B=4，那么一个像素点占4+4+4+4=16 位；

——Bitmap.Config ARGB\_8888

每个像素占四位，即A=8，R=8，G=8，B=8，那么一个像素点占8+8+8+8=32 位；

——Bitmap.Config RGB\_565

每个像素占四位，即R=5，G=6，B=5，没有透明度，那么一个像素点占 5+6+5=16位；

——Bitmap.Config ALPHA\_8

每个像素占四位，只有透明度，没有颜色；

一般情况下我们都是使用的ARGB\_8888，由此可知它是最占内存的，因为一个像素占32位，8位=1字节，所以一个像素占4字节的内存。

（Picasso at ARGB8888 and Glide at RGB565）

***举例***

假设有一张480x800的图片，如果格式为ARGB\_8888，那么将会占用1500KB的内存：

480\*800=38400（像素）

38400\*4=1536000（字节Byte）

1536000/1024=1500（KB）

# 2017-2-6

## AS集成科大讯飞MSC（Mobile Speech Client，移动语音终端）的语音识别功能

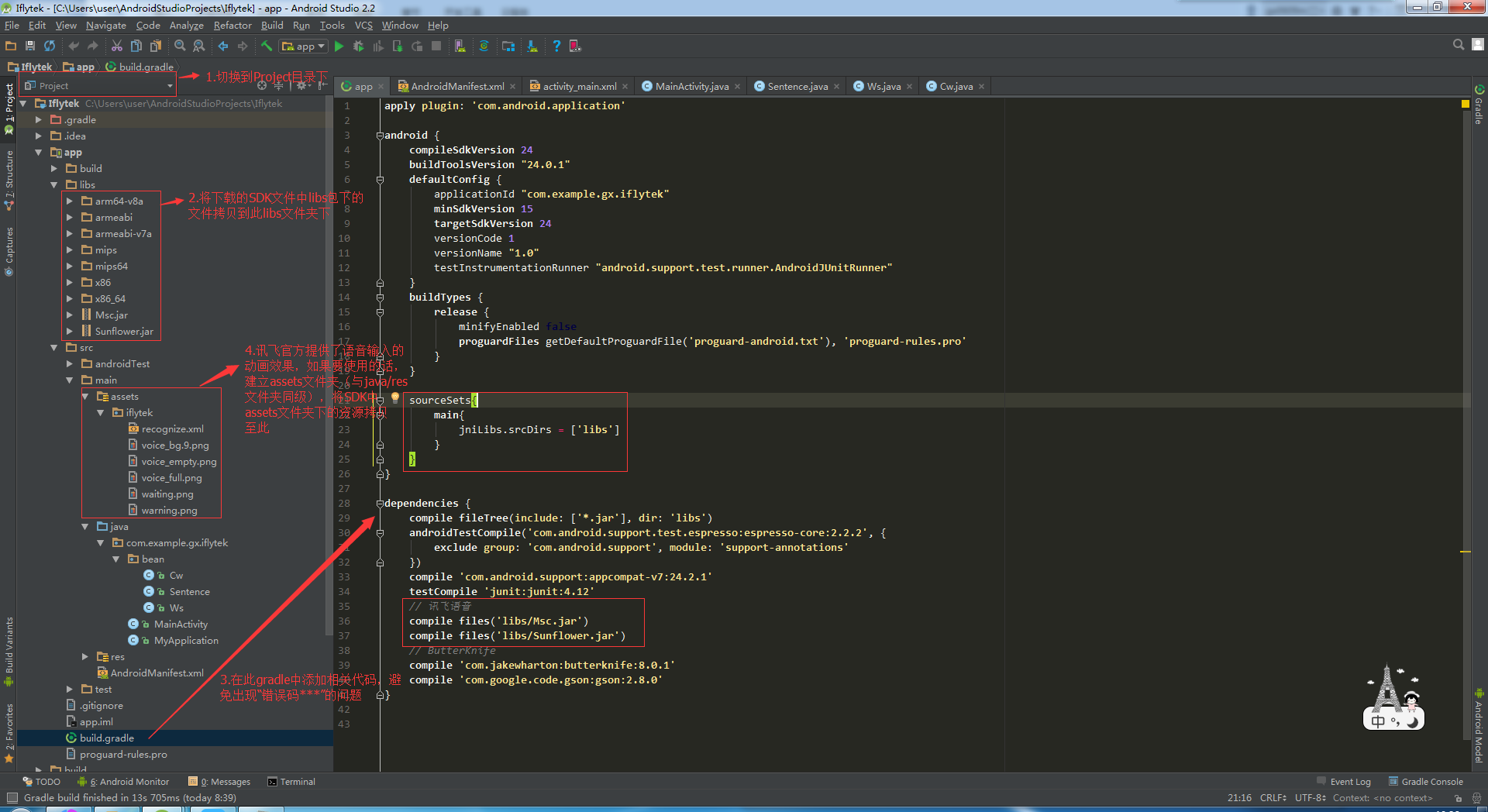
***前期工作***

注册账号——>创建应用——>申请功能——>下载相关SDK：



***开发准备***

引用和配置相关语音开发的资源文件：



***代码编写***

——

**申请相关权限**

在工程AndroidManifest.xml文件中添加如下权限：

<!--连接网络权限，用于执行云端语音能力 -->

<uses-permission android:name="android.permission.INTERNET"/>

<!--获取手机录音机使用权限，听写、识别、语义理解需要用到此权限 -->

<uses-permission android:name="android.permission.RECORD\_AUDIO"/>

<!--读取网络信息状态 -->

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"/>

<!--获取当前wifi状态 -->

<uses-permission android:name="android.permission.ACCESS\_WIFI\_STATE"/>

<!--允许程序改变网络连接状态 -->

<uses-permission android:name="android.permission.CHANGE\_NETWORK\_STATE"/>

<!--读取手机信息权限 -->

<uses-permission android:name="android.permission.READ\_PHONE\_STATE"/>

<!--读取联系人权限，上传联系人需要用到此权限 -->

<uses-permission android:name="android.permission.READ\_CONTACTS"/>

<!--摄相头权限，拍照需要用到 -->

<uses-permission android:name="android.permission.CAMERA" />

——

**混淆代码**

如需在打包或者生成APK的时候进行混淆，请在proguard.cfg中添加如下代码

-keep class com.iflytek.\*\*{\*;}

——

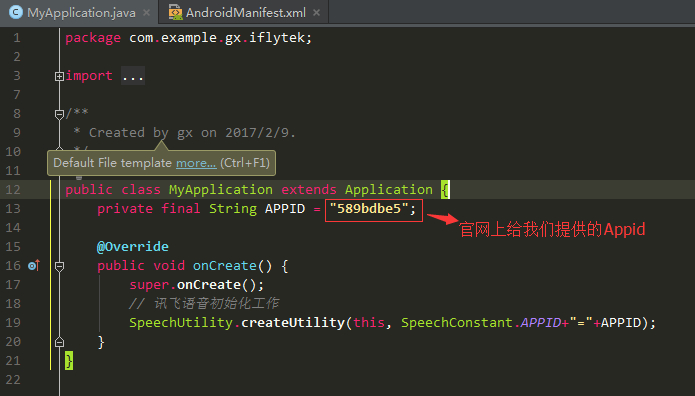
**初始化**

初始化即创建语音配置对象，只有初始化后才可以使用MSC的各项服务。建议将初始化放在程序入口处（如Application、Activity的onCreate方法),初始化代码如下：

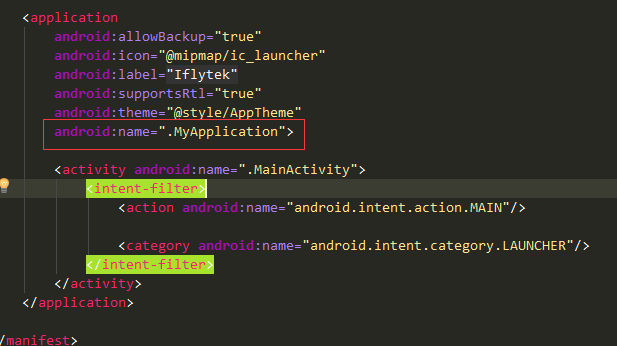
// 将“12345678”替换成您申请的APPID，申请地址：http://open.voicecloud.cn

SpeechUtility.createUtility(context, SpeechConstant.APPID +"=12345678");

为了便于初始化，我们编写一个Application 子类（Application类不能修改），在其中进行初始化的工作：



下一步要在androidmanifest.xml中修改app默认的Application类我们编写的MyApplication类：



到这里，语音的初始化工作算是完成了。

——

**编写实体类**

我们录入的语音，讯飞识别之后是按Json格式返回给我们的，因此我们还有将Json格式的数据进行解析：

讯飞返回的Json数据如下：



方法一：

我们编写三个实体类

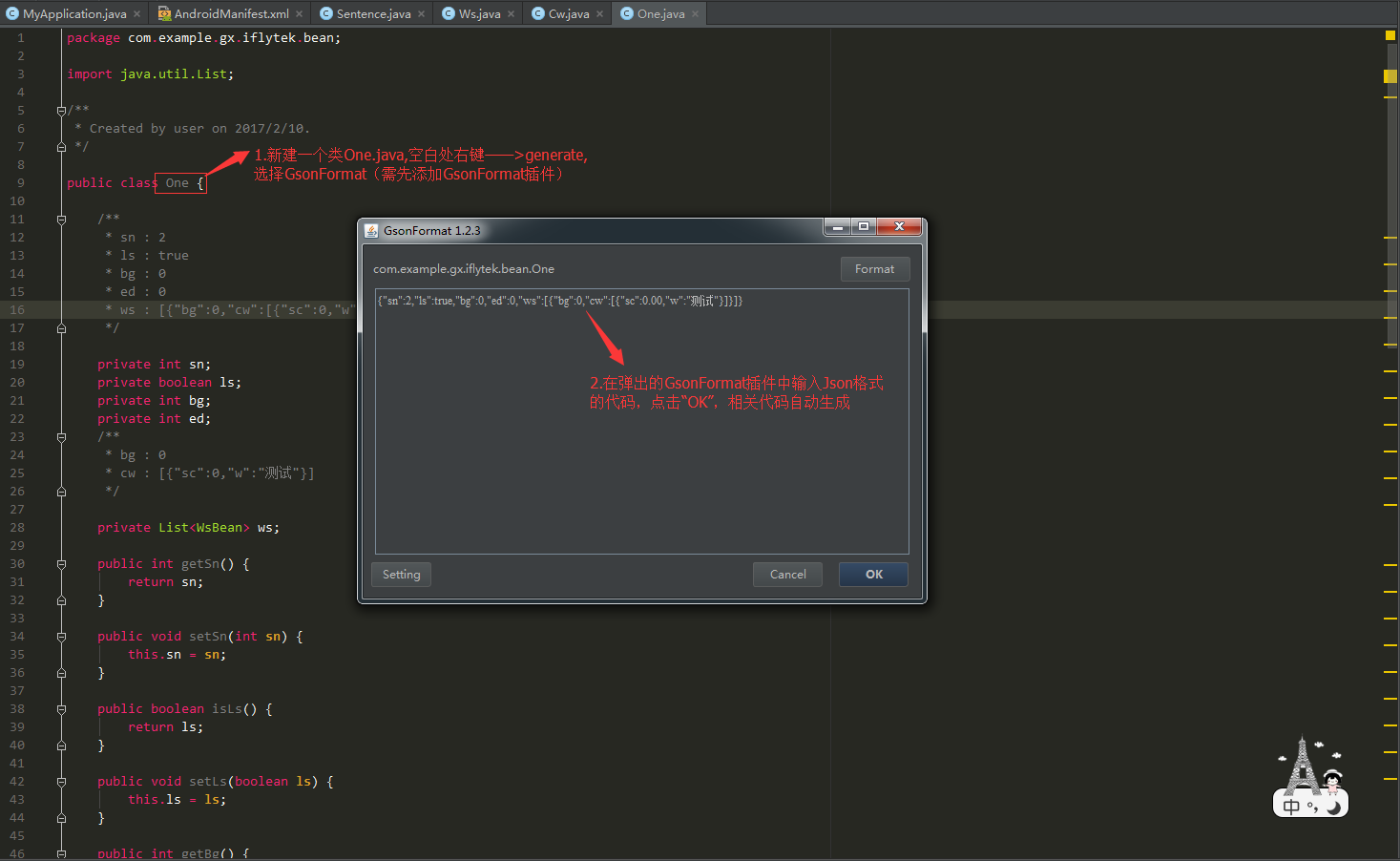
Sentence.java——>private int sn,boolean ls,int bg,int ed,List<Ws> ws , getters and setters;

Ws.java——> private int bg,List<Cw> cw , getters and setters;

Cw.java——>private String w,int sc , getters and setters;

方法二：

上述代码需要手动编写，在AndroidStudio中可以使用GsonFormat插件大大简化我们开发，



使用此方法，类的数量变少了，还方便了我们开发，还有什么理由不站在巨人的肩膀上呢。

——

**语音识别代码**

先来看看官方文档的说明：

听写主要指将连续语音快速识别为文字的过程，科大讯飞语音听写能识别通用常见的语句、词汇，而且不限制说法。语音听写的调用方法如下：

//1.创建SpeechRecognizer对象，第二个参数：本地听写时传InitListener

SpeechRecognizer mIat= SpeechRecognizer.createRecognizer(context, null);

//2.设置听写参数，详见《科大讯飞MSC API手册(Android)》SpeechConstant类

mIat.setParameter(SpeechConstant.DOMAIN, "iat");

mIat.setParameter(SpeechConstant.LANGUAGE, "zh\_cn");

mIat.setParameter(SpeechConstant.ACCENT, "mandarin ");

//3.开始听写

mIat.startListening(mRecoListener);

//听写监听器

private RecognizerListener mRecoListener = new RecognizerListener(){

//听写结果回调接口(返回Json格式结果，用户可参见附录12.1)；

//一般情况下会通过onResults接口多次返回结果，完整的识别内容是多次结果的 //累加；

//关于解析Json的代码可参见MscDemo中JsonParser类；

//isLast等于true时会话结束。

public void onResult(RecognizerResult results, boolean isLast) {

Log.d("Result:",results.getResultString ());}

//会话发生错误回调接口

public void onError(SpeechError error) {

error.getPlainDescription(true) //获取错误码描述}

//开始录音

public void onBeginOfSpeech() {}

//音量值0~30

public void onVolumeChanged(int volume){}

//结束录音

public void onEndOfSpeech() {}

//扩展用接口

public void onEvent(int eventType, int arg1, int arg2, Bundle obj) {}

};

为了便于快速开发，SDK还提供了一套默认的语音交互动画以及调用接口，如需使用请将SDK资源包assets路径下的资源文件拷贝至Android工程asstes目录下，然后通过以下代码使用交互动画：

//1.创建SpeechRecognizer对象，第二个参数：本地听写时传InitListener

RecognizerDialog iatDialog = new RecognizerDialog(this,mInitListener);

//2.设置听写参数，同上节

//3.设置回调接口

iatDialog.setListener(recognizerDialogListener);

//4.开始听写

iatDialog.show();

官方文档看完了，在给出代码的实例：

package com.example.gx.iflytek;  
  
import ...  
  
public class MainActivity extends AppCompatActivity {  
  
 private String TAG = **"SpeechIflytek"**;  
 private SpeechRecognizer mlat;  
 private TextView tvContent;  
 private StringBuilder sb = new StringBuilder();  
  
 **@Override** protected void onCreate(Bundle **savedInstanceState**) {  
 super.**onCreate**(**savedInstanceState**);  
 **setContentView**(R.layout.*activity\_main*);  
 tvContent = (TextView) **findViewById**(R.id.*speechContent*);  
 }  
  
 *// 自定义语音界面* private void initSpeechRecognizer(){  
 sb.**delete**(**0**,sb.**length**());  
 *//1.创建SpeechRecognizer对象* mlat = SpeechRecognizer.**createRecognizer**(this, new InitListener() {  
 **@Override** public void onInit(int **i**) {  
 Log.**d**(TAG,**"SpeechRecognizer init() code = "**+**i**);  
 if(**i** != ErrorCode.*SUCCESS*){  
 System.*out*.**println**(**"初始化失败，错误码："**+**i**);  
 }  
 }  
 });  
 *//2.设置听写参数* mlat.**setParameter**(SpeechConstant.*DOMAIN*,**"iat"**);  
 mlat.**setParameter**(SpeechConstant.*LANGUAGE*,**"zh\_ch"**);  
 mlat.**setParameter**(SpeechConstant.*ACCENT*,**"mandarin"**);  
 *//3.开始听写,编写听写监听器* mlat.**startListening**(new RecognizerListener() {  
 **@Override** public void onVolumeChanged(int **i**, byte[] **bytes**) {  
 }  
 **@Override** public void onBeginOfSpeech() {  
 }  
 **@Override** public void onEndOfSpeech() {  
 }  
 **@Override** public void onResult(RecognizerResult **result**, boolean **b**) {  
 String content = **result**.**getResultString**();  
 System.*out*.**println**(content);  
 Gson g = new Gson();  
 *// content为json格式的数据，官网上没有提供json的格式，我们就是在此将其打印出来才获得其  
 // 内容：{"sn":2,"ls":true,"bg":0,"ed":0,"ws":[{"bg":0,"cw":[{"sc":0.00,"w":"测试"}]}]}* tvContent.**setText**(**"语音识别结果（style1）:"**+content);  
 }  
 **@Override** public void onError(SpeechError **error**) {  
 System.*out*.**println**(**error**.**getErrorCode**());  
 }  
 **@Override** public void onEvent(int **i**, int **i1**, int **i2**, Bundle **bundle**) {  
 }  
 });  
 }  
 *// 讯飞官方语音界面* private void initDialog(){  
 *// 每次语音录入清空之前的内容* sb.**delete**(**0**,sb.**length**());  
 *//1.创建SpeechRecognizer对象* RecognizerDialog mDialog = new RecognizerDialog(this, new InitListener() {  
 **@Override** public void onInit(int **i**) {  
 Log.**d**(TAG,**"SpeechRecognizer init() code = "**+**i** );  
 if (**i** != ErrorCode.*SUCCESS*){  
 System.*out*.**println**(**"初始化失败，错误码："**+**i**);  
 }  
 }  
 });  
 *//2.设置听写参数，同上节* mDialog.**setParameter**(SpeechConstant.*LANGUAGE*,**"zh\_cn"**);  
 mDialog.**setParameter**(SpeechConstant.*ACCENT*,**"mandarin"**);  
 *//3.设置回调接口* mDialog.**setListener**(new RecognizerDialogListener() {  
 **@Override** public void onResult(RecognizerResult **result**, boolean **b**) {  
 String content = **result**.**getResultString**();  
 System.*out*.**println**(content);  
 *// 使用Gson解析，需要在gradle中引入 compile 'com.google.code.gson:gson:2.8.0'* Gson gson = new Gson();  
 Sentence sentence = gson.**fromJson**(content,new Sentence().**getClass**());  
 List<Ws> wsList = sentence.**getWs**();  
 for (Ws w:wsList){  
 List<Cw> cwList = w.**getCw**();  
 for (Cw c:cwList){  
 String chineseWord = c.**getW**();  
 sb.**append**(chineseWord);  
 }  
 }  
 tvContent.**setText**(**"语音识别结果（style0）:"**+sb.**toString**());  
 }  
 **@Override** public void onError(SpeechError **error**) {  
 }  
 });  
 *//4.开始听写* mDialog.**show**();  
 }  
 public void showDialogStyle0(View **view**){  
 **initDialog**();  
 }  
 public void showDialogStyle1(View **view**){  
 tvContent.**setText**(**"请说话..."**);  
 **initSpeechRecognizer**();  
 }  
}

到此，引入讯飞语音识别的功能就完成了，如果还想添加讯飞的其他功能，可参照此，查看其官方网站上的其他开发文档。

# 2017-2-13

## Retrofit使用说明

*（http://blog.csdn.net/tencent\_bugly/article/details/51580627）*

***前言***  
Android 开发中，从原生的 HttpUrlConnection 到经典的 Apache 的 HttpClient，再到对前面这些网络基础框架的封装，比如 Volley、Async Http Client，Http 相关开源框架的选择还是很多的，其中由著名的 Square 公司开源的 Retrofit 更是以其简易的接口配置、强大的扩展支持、优雅的代码结构受到大家的追捧。也正是由于 Square 家的框架一如既往的简洁优雅，所以我一直在想，Square 公司是不是只招处女座的程序员？

***初识Retrofit***

Retrofit：意为“改进，式样更新”，单从字面上似乎看不出来什么，看看维基百科上对它的定义：

Retrofitting refers to the addition of new technology or features to older systems.

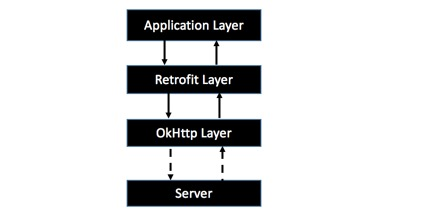
–From Wikipedia

于是我们就明白了，冠以 Retrofit 这个名字的这个家伙，应该是某某某的 『Plus』 版本了。

***Retrofit 概览***

Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。注意这里并没有说它是网络请求框架，主要原因在于网络请求的工作并不是 Retrofit 来完成的。

Retrofit 2.0 开始内置 OkHttp，前者专注于接口的封装，后者专注于网络请求的高效，二者分工协作，宛如古人的『你耕地来我织布』，小日子别提多幸福了。（由此看来，Retrofit负责头尾的封装处理工作，而OkHttp则专注于处理中间的过程）



我们的应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作，在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，后者根据用户的需求对结果进行解析的过程。

讲到这里，你就会发现所谓 Retrofit，其实就是 **Retrofitting OkHttp** 了。

***Hello Retrofit***

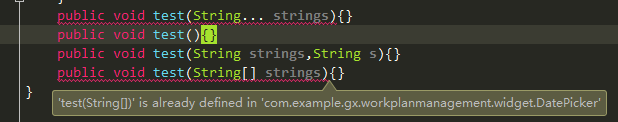
## 方法参数中String...与String[]的区别

***前言***

String...这种写法是从JDK1.5开始的，是Java对方法参数提供的一种新写法支持，称为可变长度参数列表，其语法就是类型后面跟“...”，表示此处接受的参数为0到多个Object类型的对象，或者是一个Object[]。

***说明***

例如我们有一个方法 test(String... strings){}，那么我们还可以再定义一个无参方法test(){}，或者是确定参数个数的方法test(String strings){}，但如果我们定义数组作为参数的test(String[] strings){}，则会出现编译错误，系统会提示出现重复的方法，如下：



***使用***

在使用的时候，对于test(String...strings)，你可以直接用test()去调用，标示没有参数，也可以用去test("aaa")，也可以用test(new String[]{"aaa","bbb"})；

如果既有test(String...strings)函数，又有test()函数，我们在调用test()时，会优先使用test()函数，只有当没有test()函数时，我们调用test()，程序才会走test(String...strings)方法。

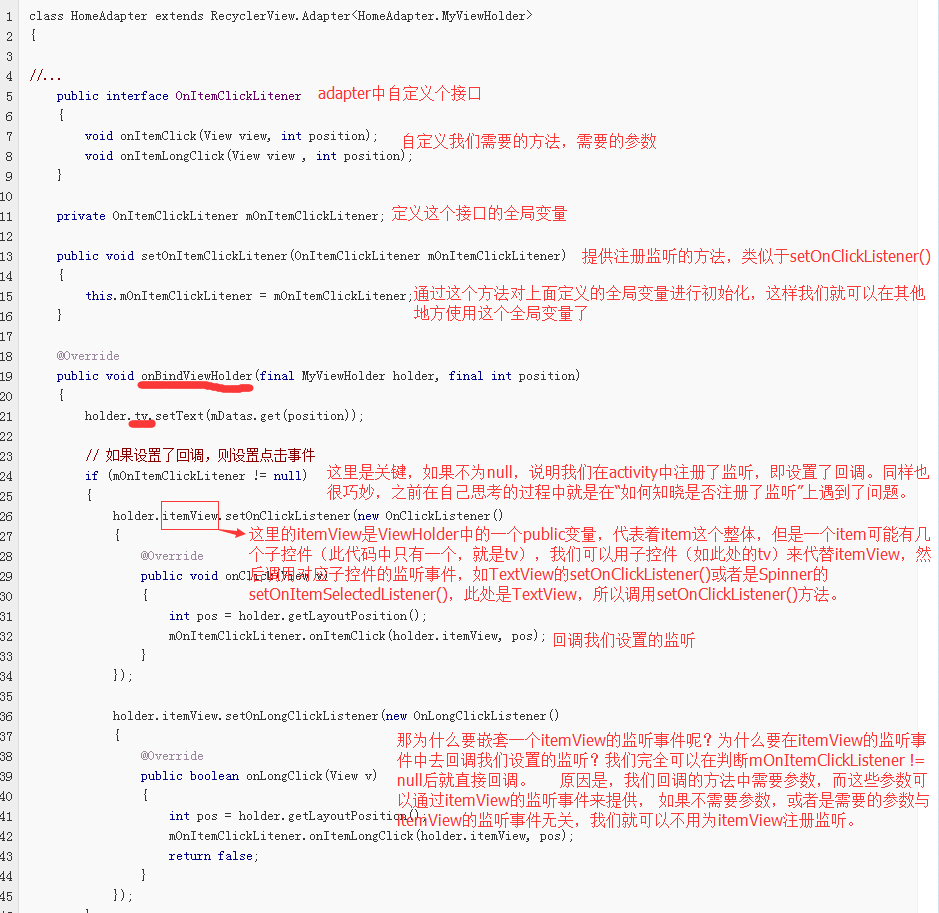
# 2017-2-27

## 由给RecyclerView编写item的监听事件所想到的

***前言***

看到鸿洋大神有关RecyclerView的博客中写的item的监听事件时深受启发，因为之前自己想的比较麻烦，而大神的代码如此简洁好用又便于理解。

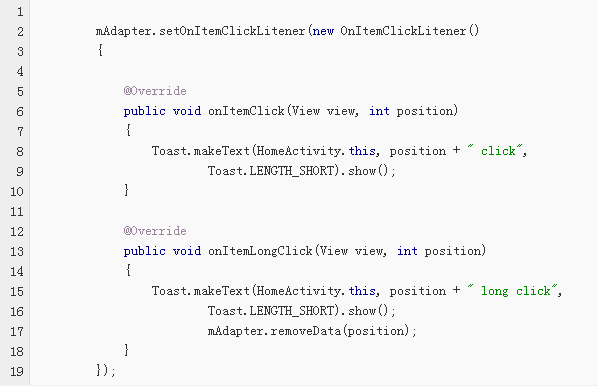
***代码***



注释的比较详细，以后在遇到有关（控件）的回调监听可参考上面的思想

（其实就是：adapter中自己定义了个接口，然后在onBindViewHolder中去为holder.itemView去设置相应的监听（这个监听是itemView的监听，根据itemView类型的不同，监听方法也不同），最后回调我们设置的监听）。

当我们需要在activity中注册监听时：



说到这里，我们在顺便提一下RecyclerView的Adapter中主要的方法负责的内容：



# 2017-3-6

## 更改AndroidStudio中的gradle版本

***前言***

在升级AndroidStudio或者是从github上pull项目到本地，在编译时都有可能出现问题，而这些问题大部分都是由于gradle的版本不同导致的。

例如升级AS时，高版本的AS可能要求相应的gradle版本也比较高，所以就需要重新下载（不能用本地的低版本），而gradle下载需要科学上网，因此你就出现了问题；

再如同步项目到本地时，由于远程项目的gradle版本不一定和本地的一致，因而出现了问题；

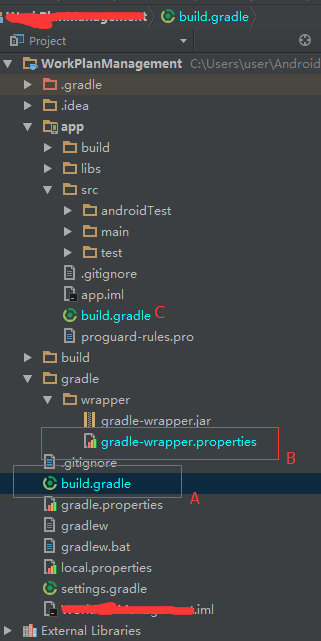
***解决方法***

**方法一：使用本地的gradle**

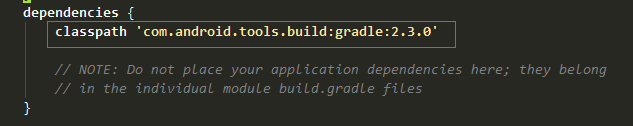
对于AS升级，我们可以不升级gradle（如果没有强制要求）；

对于同步项目到本地，可以将远程项目的gradle版本配置改为本地的版本；

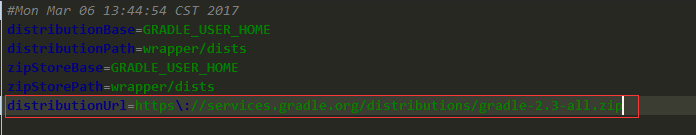
见下图：



我们只需打开A、B文件，修改A文件中的

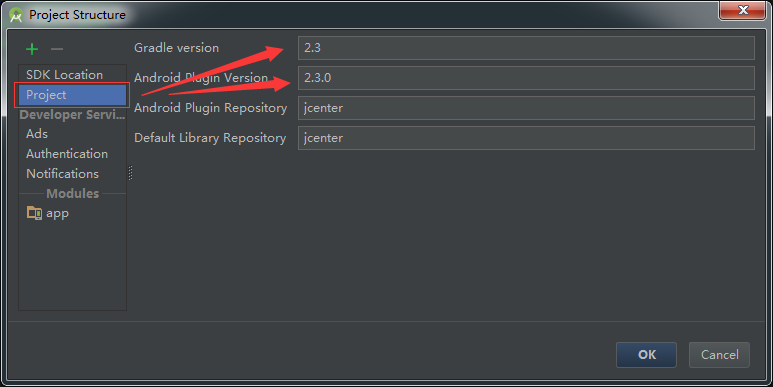


为本地的gradle版本，修改B文件中的



为本地gradle版本即可，注意A与C的区别。

选择File——>Project Structure，点击project选项，



查看gradle是否为本地的版本，即可解决问题。

**方法二：使用下载的Gradle**

下载需要的gradle.zip，例如gradle-3.3-all.zip，解压和放到正确的文件夹，配置一下即可：

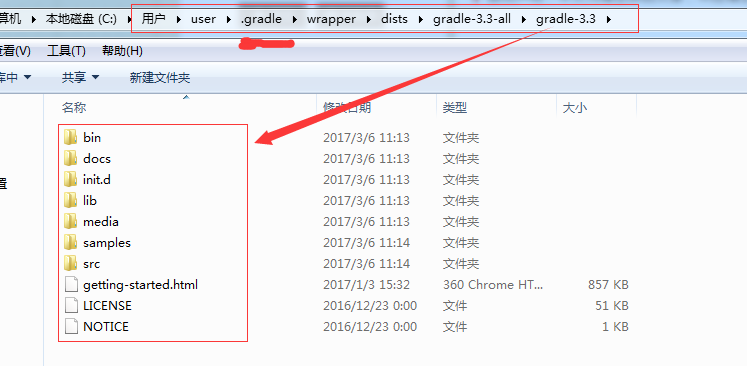
如何下载：

<http://services.gradle.org/distributions（不须科学上网）；>

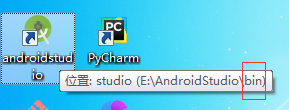
放到哪个文件夹：

关于这个网上有两个说法，说法A如下：

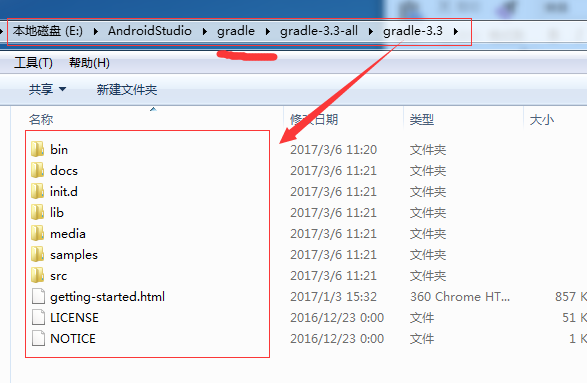
AndroidStudio的安装目录下，会有一个.gradle文件夹，放到这个文件下的子文件夹中



说法B:

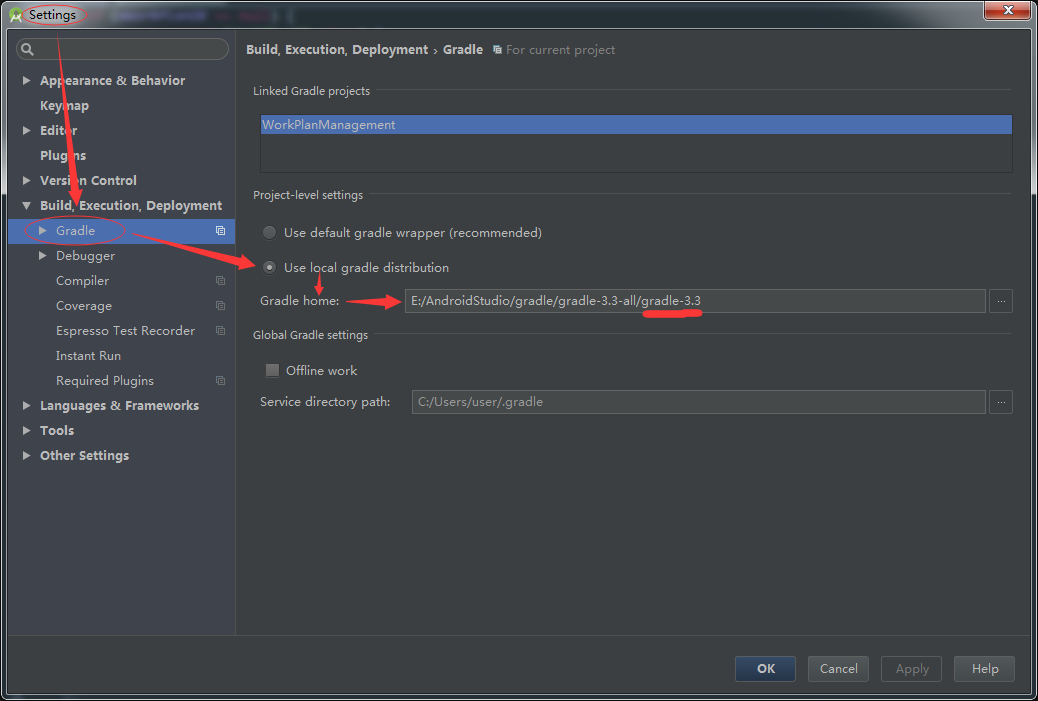


在与bin目录同级有一个gradle文件夹（不是.gradle），将解压后的文件放到如图位置



解压后的文件可能是gradle-3.3.all也可能是gradle-3.3，但是在AS中指定目录时一定要指定到gradle-3.3，即要指定到包含大红框中文件的文件夹。

如何配置：



在Settings中找到Gradle，选择Use local...，配置Gradle Home，这里我是按照说法B配置的（说法A配置后出了问题）。这样，即是在AS中使用我们下载的需要的Gradle版本。

## Android项目版本控制提交忽略表

***前言***

Android项目中有很多是工程信息或者是配置相关的文件时不需要同步的，那么具体是那些呢？又如何知道哪些不用同步呢？

***不用同步的文件***

**A**

(1) .idea 文件夹，此文件夹是用来保存开发工具的设置信息；

(2) .gradle文件夹，此文件夹是用来保存gradle的依赖信息；

(3) build 文件夹，build文件夹是用来保存编译后的文件目录；

(4) .iml 文件，是用来保存开发工具信息；

(5) local.properties 文件，是用来保存项目依赖信息；

**B**

Android Studio在Project和app下各有一个 .gitignore 文件,Project下的.gitignore忽略文件如下:

\*.iml  
.gradle  
/local.properties  
/.idea/workspace.xml  
/.idea/libraries  
.DS\_Store  
/build  
/captures  
.externalNativeBuild

app文件夹下的.gitignore文件内容如下：

/build

可以看出:

所有的 .iml 文件都不需要提交；

build 文件夹下的文件不需要提交,这是编译文件；

本地设置相关文件不需要提交；

性能分析时的文件会在/captures文件夹下,也不需要提交；

**C**

可以直接去github找个有名的安卓项目，看别人的.gitignore里有啥，在根据公司的具体情况实际处理（跟B差不多，因为github上面的项目也基本是别人从AS上提交上去的，也就是AS中.gitignore文件中的内容）；

## 将本地代码提交到gitOSChina时出现的问题

这里只是简述一下大概的步骤和遇到问题的解决方法（注意是从本地提交到远程）：

——建立远程仓库\*\*.git，

——在本地的项目根目录下git bash here，

——然后git init，这样此项目就加入了git管理了，

——接着git add -m “注释”,将此项目加入本地暂存区，

——git commit 将暂存区的内容放置到本地工作区，

——git remote add origin [https://......./\*\*.git，本地仓库与远程关联](https://......./**.git，本地仓库与远程关联)

——git push origin master，同步到远程，

这里出现了如下问题：

”Updates were rejected, because the tip of your current branch is behind its remote counterpart”

这是因为远程作了更新，需要先git pull，从远程拉取合并到本地，再同步，

——接着我们git pull origin master，

不过也出现了问题：

“fatal: refusing to merge unrelated histories”

这是因为远程把这看成是两个不同的项目了，上面的git命令需要加点参数，如下：

git pull origin master --allow-unrelated-histories

——上面修改后的git命令执行成功后，再执行git push origin master就可以了

（注意上面的各种命令可能有不同的实现细节，如是全部add还是部分add等，那就需要不同的命令参数了）

## Android应用性能指标分析工具

***Lint——AS自带的静态代码分析工具***

**（http://blog.csdn.net/u010687392/article/details/47835743）**

**功能**

能够检测到：

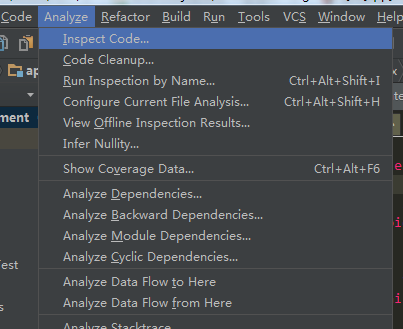
——xml文件中是否存在hardcode硬编码；

——是否存在unused resources没有使用到的资源；

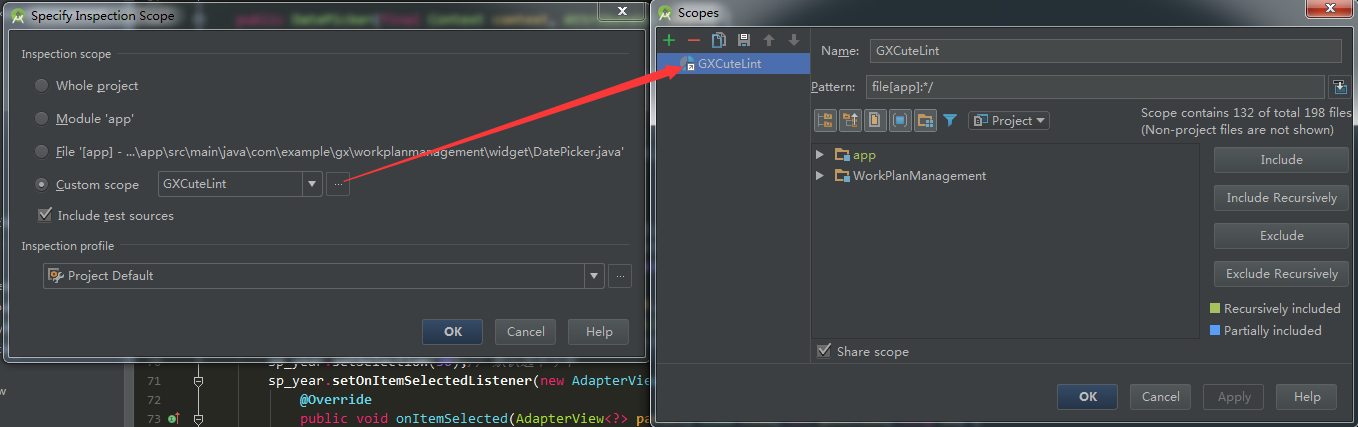
——probable bug可能的bug等等；

**使用**

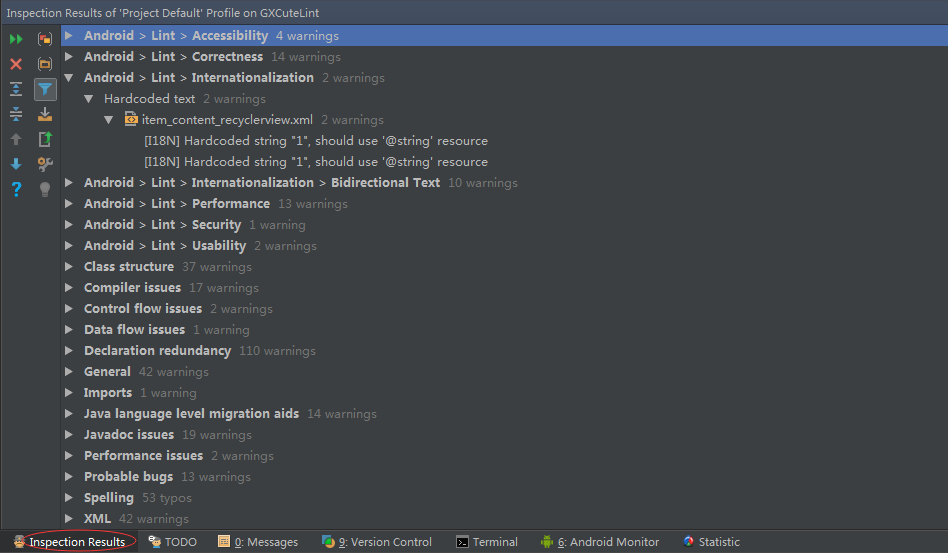
——点击Analyze目录下的Inspect Code选项



——在弹出的检测范围选项框中进行选择，可以选择Whole project，也可以选择Custom project来进行自定义范围的选择，



确定好范围之后点击“OK”,



我们会在Inspection Results选项中看到结果，并可以根据给出的提示和建议进行修改（点击对应的问题后后边会给出问题的具体位置），虽然这些问题并不会影响App的正常运行，不过这对于项目的规范性和维护性来说是非常重要的。

***LeakCanary——square公司推出的一款检测内存泄漏的工具***

**（https://github.com/square/leakcanary**

**https://www.liaohuqiu.net/cn/posts/leak-canary-read-me/）**

**原理和优点**

——LeakCanary会检测应用的内存回收情况，如果发现有垃圾对象没有被回收，就会去分析 当前的内存快照，也就是使用MAT会用到的.hprof文件，找到对象的引用链，并显示 在页面上；

——这款插件的好处就是,可以在手机端直接查看内存泄露的地方,可以辅助我们检测内存泄 露；

**使用**

A.

在build.gradle中添加：

dependencies {

debugCompile 'com.squareup.leakcanary:leakcanary-android:1.5'

releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5'

testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5'

}

B.

在你的Application类的onCreate()方法中添加：

public class ExampleApplication extends Application {

@Override public void onCreate() {

super.onCreate();

if (LeakCanary.isInAnalyzerProcess(this)) {

// This process is dedicated to LeakCanary for heap analysis.

// You should not init your app in this process.

return;

}

LeakCanary.install(this);

// Normal app init code...

}

}

C.

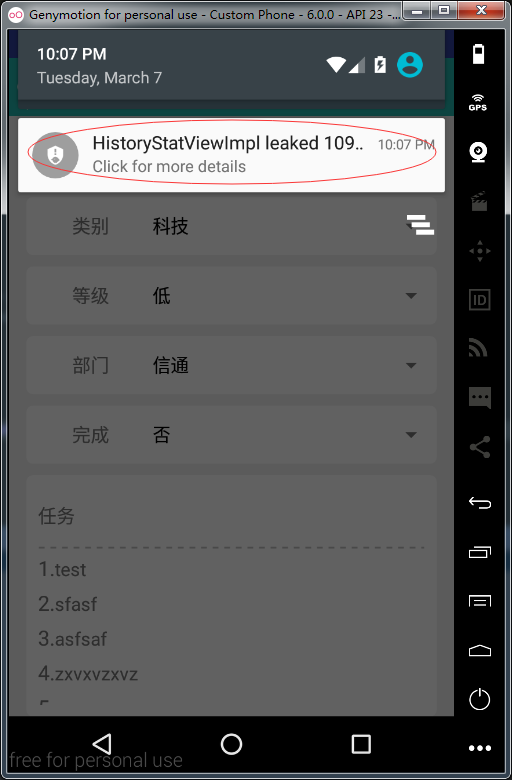
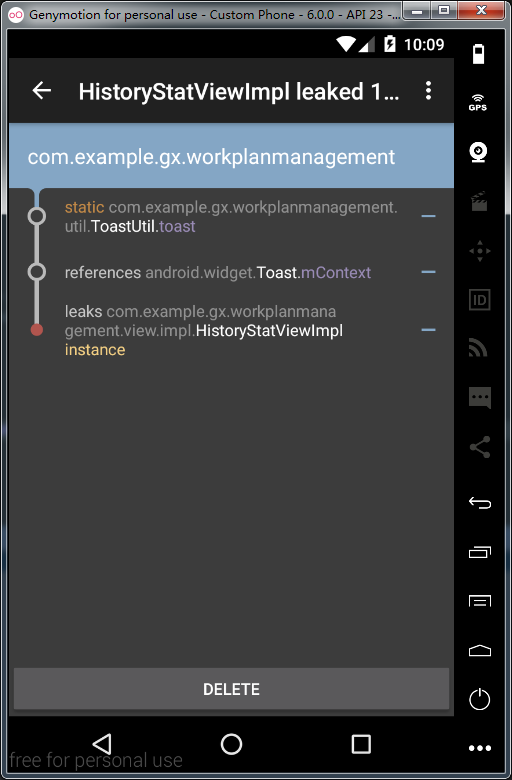
应用运行起来后，LeakCanary会自动去分析当前的内存状态，如果检测到泄漏会发送到通知栏（

“LeakCanary will automatically show a notification when an activity memory leak is detected in your debug build.”

“在 debug build 中，如果检测到某个 activity 有内存泄露，LeakCanary 就是自动地显示一个通知”），

（上面提到的debug build是什么暂时不很清楚，不过直接运行就可以了），

点击通知栏就可以跳转到具体的泄漏分析页面，如下：

***Memory控制台——AS自带的内存监视器***

**（http://blog.csdn.net/zxc123e/article/details/52924264**

**http://www.2cto.com/kf/201512/455421.html）**

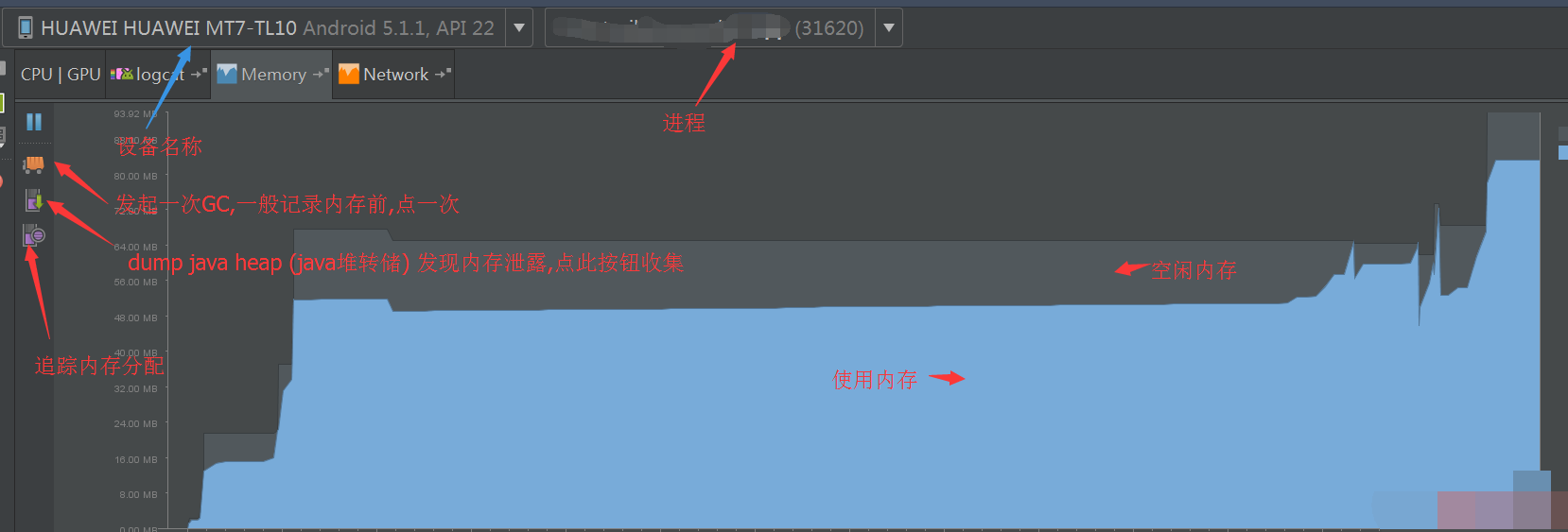
AndroidStudio 中Memory控件台（显示器）提供了一个内存监视器。我们可以通过它方便地查看应用程序的性能和内存使用情况。

**使用**

打开Android Monitor选项卡



找到Memory一栏



不同版本的AS，界面展示可能有一点区别，但大致相同。

运行要监控的程序（APP）后，打开Android Monitor控制台窗口，可以看到Memory控制台。 点击Memory控制台上Enable按钮，Memory控制台开始显示正在运行时程序的Memory使用情况，如上图所示。

**功能介绍**

AndroidStudio Memory的功能：

——启动与关闭Memory监测按钮；

——手动触发GC按钮；

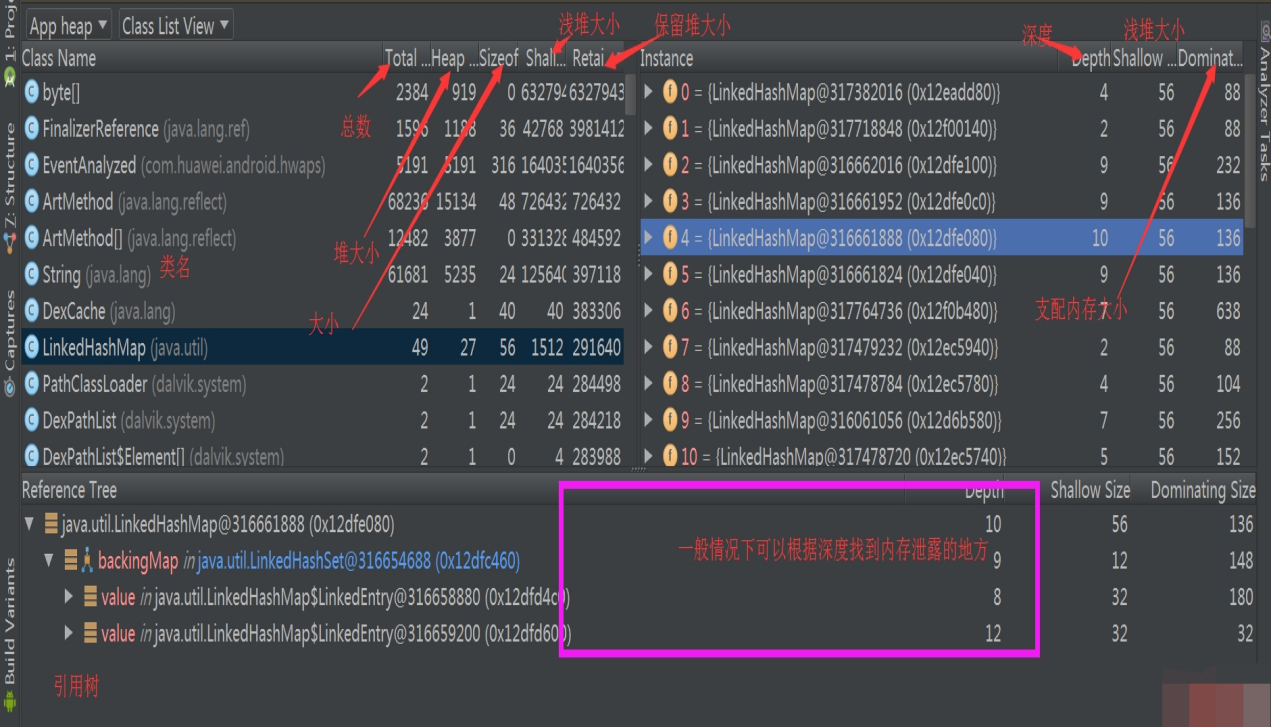
——dump java heap 按钮,点击Android Studio就开始干活了，成功后会自动 打开 hprof文件；

——start(stop) allocation tracking按钮先点击一次，然后会看到Memory

Recorder开始转动，然后自己开始在APP上面做相应的操作。在合适的时 间再点一次，结束记录；

**检测内存泄露**

我们点击dump Java heap 这个按钮，APP会Freeze住。大概几十秒后，dump成功后会自动打开.hprof文件：



如果我们想了解内存分配更详细的情况,可以使用Allocation Traker来查看内存到底被什么占用了。 点击Start Allocation Tracking按钮,开始分配追踪，过一些时间后，点击Stop Allocation Tracking结束追踪的位置。停止追踪后 .alloc文件会自动打开:



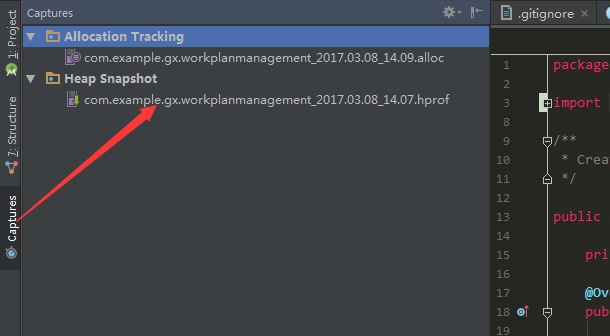
当你想查看某个方法的源码时,右键选择的方法,点击Jump to source就可以了。

**Tips**

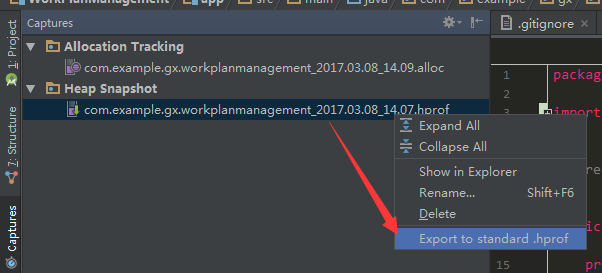
——上面提到的“自动打开.hprof文件”，打开后是Android Studio自带的界 面,查看内存泄露还不是很智能,我们可以借助第三方工具,常见的工具就 是MAT；

——我们需要下载独立版的MAT。这里需要提醒大家的是，MAT并不会准确地告 诉我们哪里发生了内存泄漏，而是会提供一大堆的数据和线索，我们需要 自己去分析这些数据来去判断到底是不是真的发生了内存泄漏；

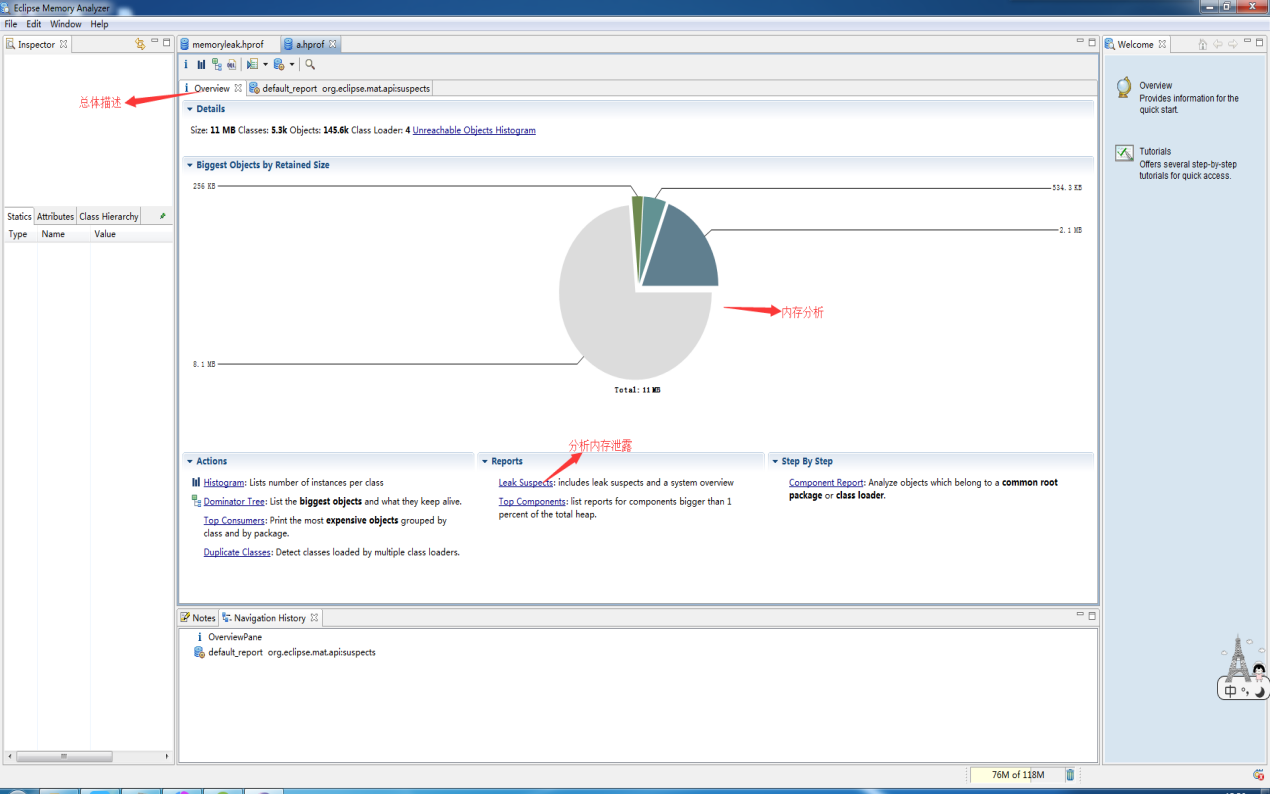
——接下来我们需要用MAT打开内存分析的文件, 上文给大家介绍了使用 Android Studio生成了 hprof文件, 这个文件在呢, 在Android Studio 中的Captrues这个目录中,可以找到：



——注意,上面得到的.hprof文件不能直接交给MAT, MAT是不识别的, 我们需 要右键点击这个文件,转换成MAT识别的:



——然后用MAT打开导出的hprof(File->Open heap dump) MAT会帮我们分析内 存泄露的原因：



***TraceView——Android 平台特有的数据采集和分析工具***

...

...

...

***最后***

上面给大家介绍了若干使用Android Studio检查程序性能的工具,工具永远是辅助,不要因为工具耽误太长时间。

## 如何正确的使用try...catch

***（https://www.zhihu.com/question/29459586）***

***前言***

对于try...catch的使用,不同的人有不同的观点，有的人建议尽量少用。经过一番了解之后，我感觉try...catch还是很不错的，只不过关键在于**如何正确的使用**。

***说明***

很多人觉得try...catch不行的原因可能是经常遇到这样的代码：

try

{

....;

}

catch( Exception e )

{

....//代码进入这里说明发生错误

}

当出错后，只知道它出错了，但并不知道是什么原因导致错误；

而正确的使用方式应该是这样的：

try{

....;

}catch( AccessException e )

{

....//代码进入这里说明发生【没有权限错误】

}catch( FileExistedException e )

{

....//代码进入这里说明发生【文件已经存在错误】

}catch( TimeoutException e )

{

....//代码进入这里说明发生【超时错误】

}

要catch具体的Exception，并且给出具体的错误原因以及需要改进的地方和建议等（在catch完记后，打好日志，记录详细的输入输出，异常原因等，然后通过异常报警通知给开发）。

有时也可以再catch住后还可以做一些补救措施，如连接某个服务器, 连接失败的话, 捕获住异常可以隔几秒再尝试连接。

***另外***

——不要把异常处理当成条件判断，该用if else的别用try catch；

——不要捕捉处理不了的异常，把异常抛到外面，能处理或者需要处理的人会 try catch这个异常；

——打印异常堆栈，输出错误信息不叫异常处理；

***Tips***

需要注意的是，try catch的异常处理机制，灵活强大的同时也会造成它的代价比较大，相对于判断返回值，跑出异常到捕获，需要更多的cpu指令和代码。同时，为了不让系统的“稳定”是建立在try...catch之上的，在“何时使用”和“如何使用”上，我们还是应该多加斟酌——即**如何正确的使用**。

# 2017-3-13

## UML几种图例之类图中的聚合与组合关系的区别

***前言***

对于聚合和组合，如果换成英文，可能会比较好理解：

——聚合是“has-a”；

——组合是“contains-a”；

类图中的表示方式的区别：

——聚合是空心的菱形；

——组合是实心的菱形（理解为实心的要比空心的扎实稳定）；

***说明***

聚合和组合的**区别**在于：

——聚合关系是“has-a”关系，组合关系是“contains-a”关系；

——聚合关系表示**整体与部分的关系**比较弱，而组合比较强；

——聚合关系中代表部分事物的对象与代表聚合事物的**对象的生存期无关**，一旦 删除了聚合对象不一定就删除了代表部分事物的对象，组合中一旦删除了组 合对象，同时也就删除了代表部分事物的对象；

***举例***

我们用几个例子来说明聚合和组合的区别：

——汽车和轮胎，就是聚合关系，因为轮胎的生存期和汽车的生存期无关；

——人和手臂，就是组合关系，因为他们的生存期是关联的；

——“国破家亡”，国灭了，家自然也没有了，“国”和“家”显然也是组合关 系；

——计算机和它的外设之间就是聚合关系，因为它们之间的关系相对松散，计算 机没了，外设还可以独立存在，还可以接在别的计算机上；

——公司和部门，也是组合关系，没有公司就不存在部门；

## 使用Stetho在Chrome上调试Android数据库/网络

***（http://facebook.github.io/stetho/）***

***前言***

“Stetho is a sophisticated debug bridge for Android applications. When enabled, developers have access to the Chrome Developer Tools feature natively part of the Chrome desktop browser. Developers can also choose to enable the optional dumpapp tool which offers a powerful command-line interface to application internals.”

简言之，就是：

“A debug bridge for Android applications”

***使用***

——调试android数据库

1.引入依赖包

compile 'com.facebook.stetho:stetho:1.3.1'

2.初始化

public class MyApplication extends Application {

public void onCreate() {

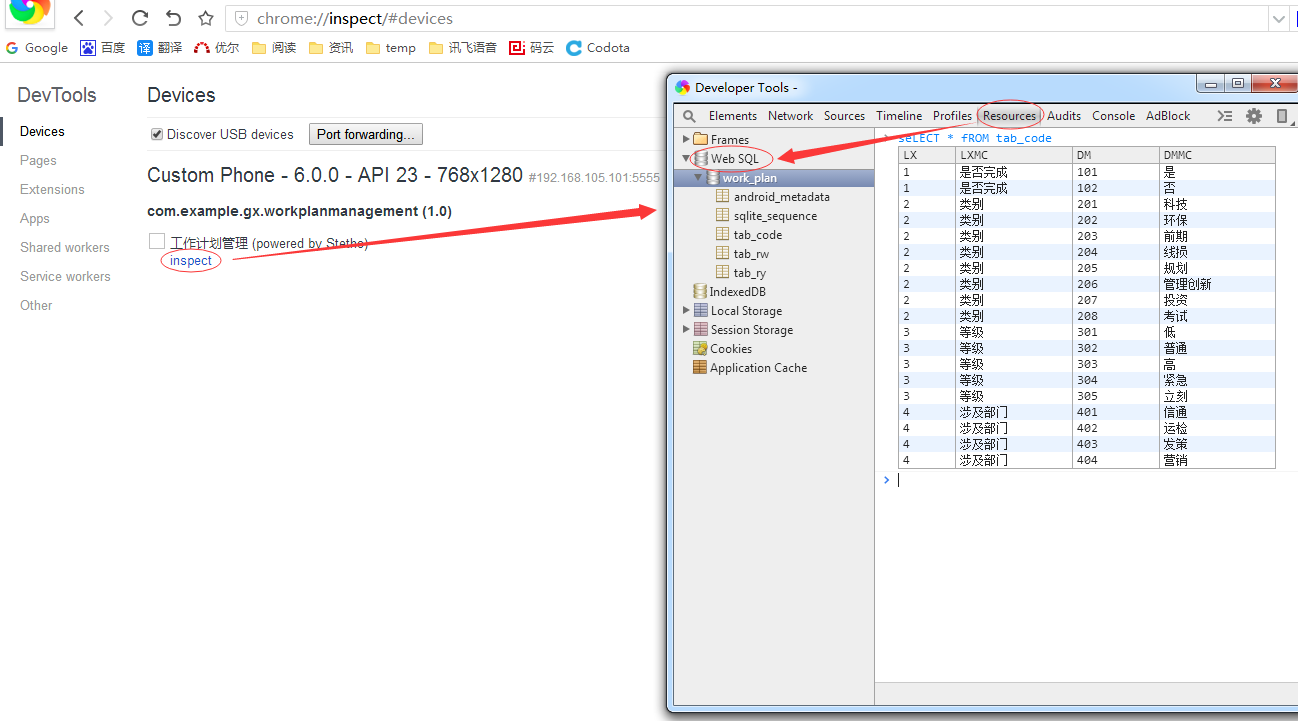
super.onCreate();

Stetho.initializeWithDefaults(this);

}

}

1. 运行App, 打开Chrome输入chrome://inspect/#devices（别忘了用数据线把手机和电脑连起来）



这里我使用的是360极速浏览器（内核是Chrome），模拟器使用的是Genymotion，点击inspect，在弹出框中点击Resources选项卡，选择Web SQL,即可查询表信息，也能通过写sql语句进行增查改删（CRUD）操作。

（**Tips**

——需要科学上网才能出现右侧的弹出框；

——没有VPN的可以搜索老D hosts；）

**PS**

网上是用真机做的示例，我用真机试验时，在左侧的Devices区没有出现设备，可能是真机需要root权限才行（但是Stetho明明声称的是免root啊**！后面又尝试了一下，不用root也是可以的**）。

——调试Android网络

1.引入依赖包

compile 'com.facebook.stetho:stetho:1.3.1'

compile 'com.facebook.stetho:stetho-okhttp3:1.3.1'

compile 'com.squareup.retrofit2:retrofit:2.0.0-beta4'

compile 'com.squareup.okhttp3:okhttp:3.2.0'

compile 'com.squareup.retrofit2:converter-gson:2.0.0-beta4'

2.初始化

public class MyApplication extends Application {

public void onCreate() {

super.onCreate();

Stetho.initializeWithDefaults(this);

}

}

3.添加拦截器

OkHttpClient client = new OkHttpClient.Builder()

.addNetworkInterceptor(new StethoInterceptor())

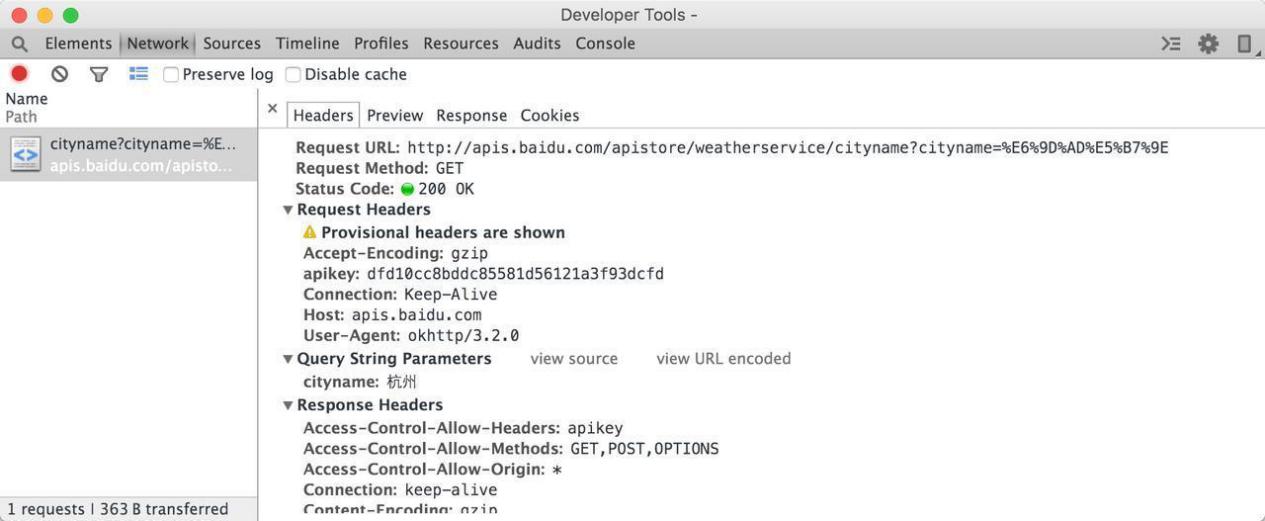
.build();

（**Tips**

——是addNetworkInterceptor不是addInterceptor；

——此例子是采用okhttp来发请求的，如果是HttpURLConnection或其 他可以到Stetho官网查看相应配置；）

1. 运行App, 打开Chrome输入chrome://inspect/#devices（跟查看数据库内容的步骤一样）



*（图片来自网络：<http://www.jianshu.com/p/03da9f91f41f）>*

（Tips

——用Chrome开发者工具抓包的时候，会发现如果页面跳转了，那么上 一个页面的请求信息就没有了。这个只要勾选上Preserve log就不 会了；）

（http://www.jianshu.com/p/cb71df5d3c45）

***Tips***

——自行钻研更多功能；

## Android ORM框架GreenDAO 3.0的使用

***（http://greenrobot.org/greendao/）***

***前言***

“greenDAO is an open source Android ORM making development for SQLite databases fun again. It relieves developers from dealing with low-level database requirements while saving development time. SQLite is an awesome embedded relational database. Still, writing SQL and parsing query results are quite tedious and time-consuming tasks. greenDAO frees you from these by mapping Java objects to database tables (called ORM, “object/relational mapping”). This way you can store, update, delete, and query for Java objects using a simple object oriented API.”

简言之，greenDAO就是一个可以将Java对象映射到SQLite数据库,让开发人员方便、快速的进行数据库操作的Android ORM框架。

***与2.x相比***

greenDAO 2.x使用相对繁琐，需要建立独立的java工程来存放相应的文件，具体这里就不细述了。而3.x开始使用注解的方式来定义实体类，并通过安装gradle插件的方式来生成相关代码，相比之下，3.x简单很多。

***优势***

——ORM的优势

我们为什么不使用原生的SQLite而要使用ORM之类的框架呢？

——正如前言中所说，ORM把我们从编写底层的SQL、解析查询结果这样沉 闷、耗时的工作中解放了出来，从而让我们可以更加专注于业务逻辑代 码；

——更好的防注入，不用ORM其实也可以做到防止SQL注入（如拼SQL）， 但比较来说麻烦了些；

——greenDAO的优势

——Maximum performance,性能最好，可能是最快的Android ORM；

——Easy to use,易用，强大的API；

——Minimal memory consumption，内存消耗最少；

——Small，体积小，减少build时间，避免65K；

——Database encryption,数据库加密，支持SQLCipher来保障用户数据安 全；

——Strong community,强大的社区，在GitHub上7000+的star；

***使用***

废话不多说了，下面进入正题：

**——引入**

A.在project的build.gradle中添加如下代码：



试着将此段代码放入module的build.gradle中，结果报如下错误：

“Error:Execution failed for task ':app:greendao'.

>org.eclipse.jdt.internal.compiler.impl.CompilerOptions.versionToJdkLevel(Ljava/lang/Object;)...”

放弃之。

B.在module的build.gradle中添加如下代码：



这里我们对第二个方框中的相关配置进行说明：

schemaVersion：当前数据库结构的版本，每次数据库升级的时候我们修改这里 的版本号即可（修改这里的版本号，greenDAO会自动修改生成 到DAOMaster中的版本号）；

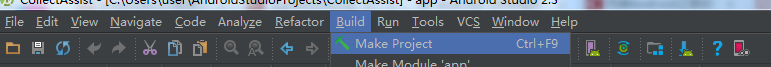
daoPackage：生成的DAO，DaoMaster和DaoSession的包名，默认是实体的包名；

targetGenDir：生成源文件的路径，默认源文件目录是在build目录中的 (build/generated/source/greendao)。

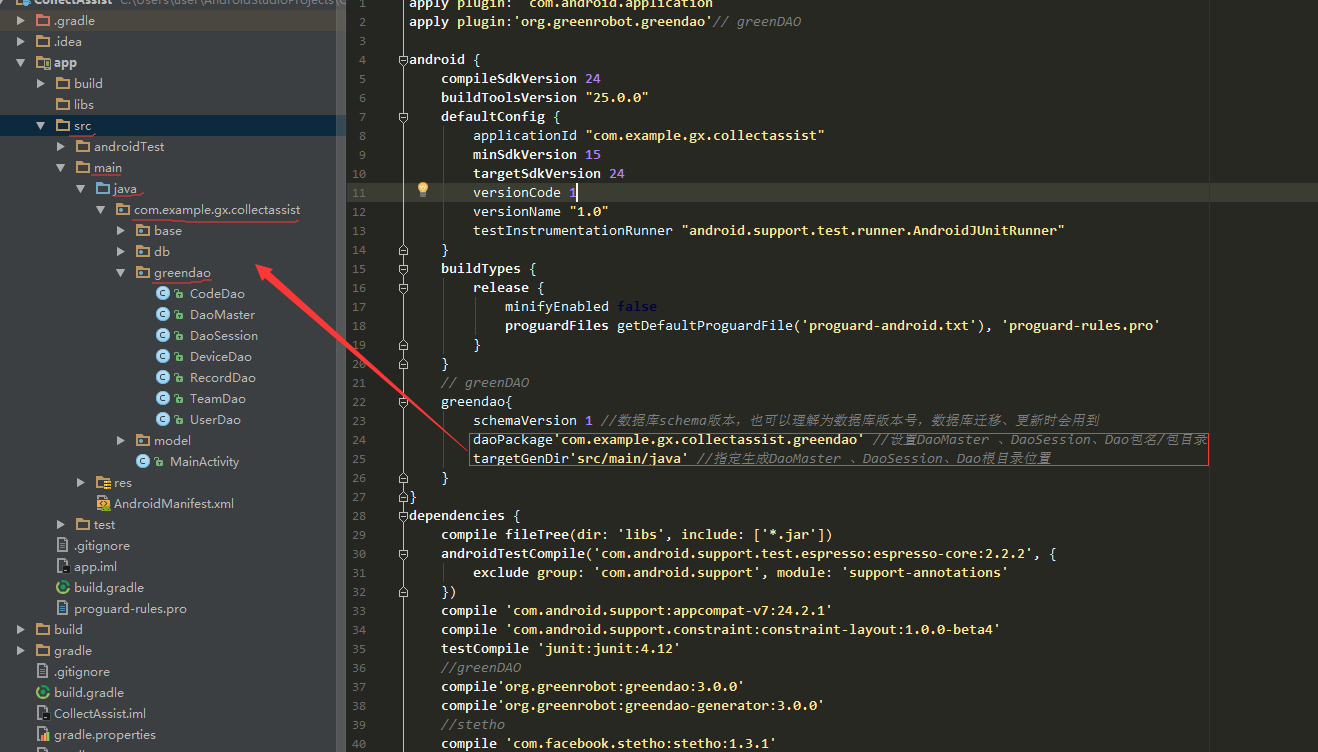
generateTests：设置是否自动生成单元测试；

targetGenDirTest：生成的单元测试的根目录；

C.点击“make project”：



接着会发现，在我们配置的目录下生成了相应的文件DaoMaster.java、DaoSession.java以及 \*\*\*Dao.java等文件：



DaoMaster：一个DaoMaster就代表着一个数据库的连接，它保存了 sqlitedatebase对象以及操作DAO classes（注意：不是对象）， 其提供了一些创建和删除table的静态方法，其内部类OpenHelper 和DevOpenHelper实现了SQLiteOpenHelper并创建数据库的框架；  
DaoSession：会话层，操作具体的DAO对象（注意：是对象），比如各种getter 方法等，可以让我们使用一些Entity的基本操作和获取Dao操作 类，DaoSession可以创建多个，每一个都是属于同一个数据库连接 的；

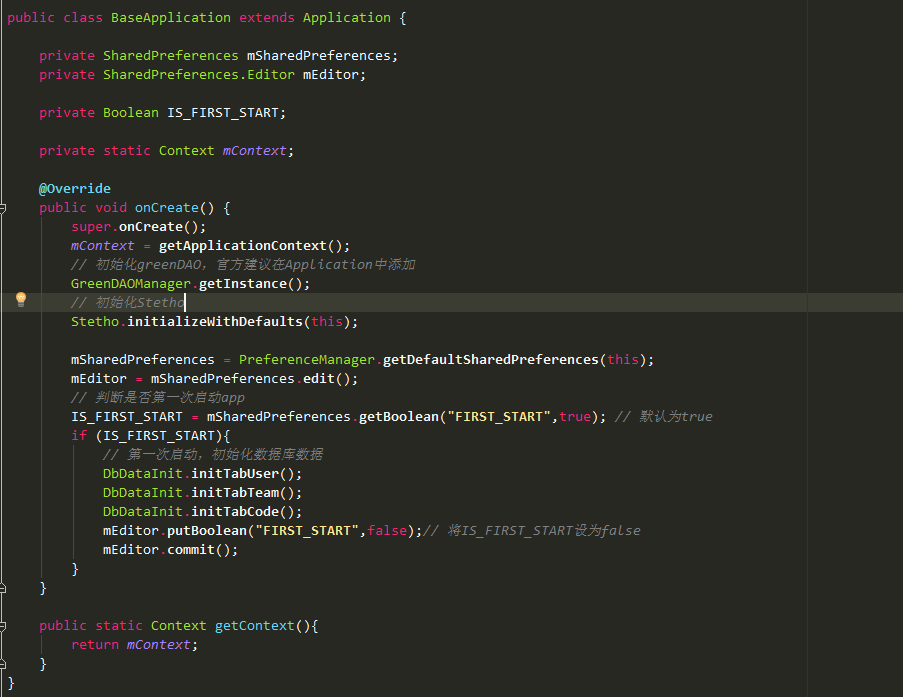
\*\*\*Dao：实体类，与数据库中的\*\*\*表一一对应，拿到了\*\*\*Dao,我们就可以操作 \*\*\*这张表了；

（Tips:

上图中我们生成了好几个\*\*\*Dao文件，这些都是greenDao初始化时同时生成的，如果我们先建好了User.java文件，“make project”后，生成了UserDao，此时在建立Code.java，然后点击“make project”虽然会生成CodeDao,但是数据库中只会有User表而不会有Code表，所以这一步需要同时操作）

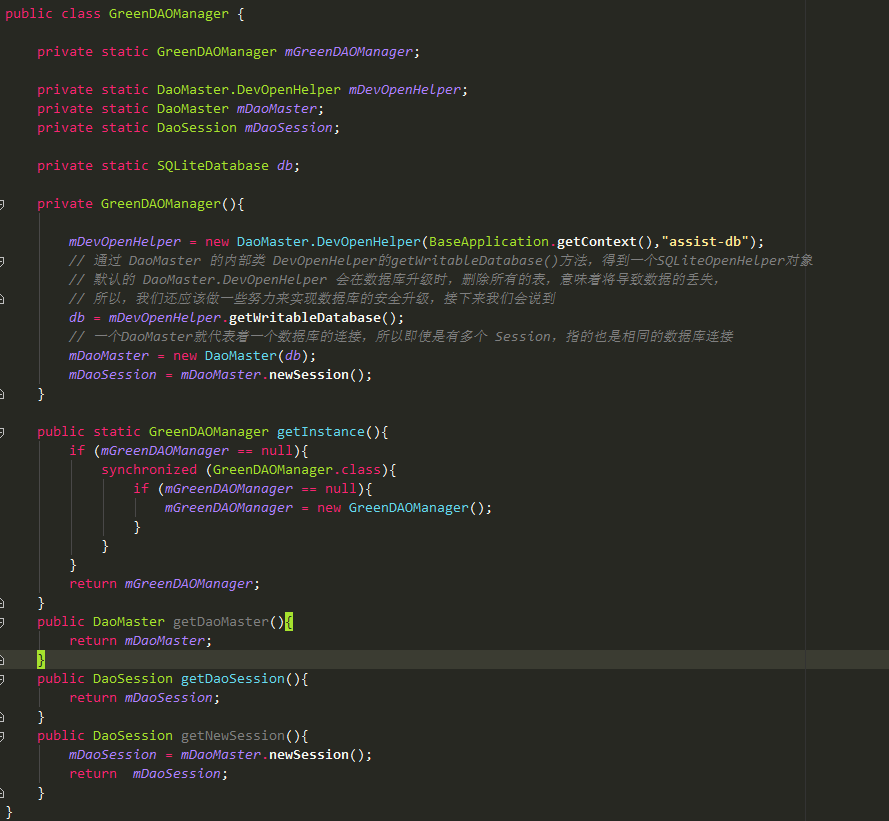
**——相关代码**

上面说到greenDao的初始化（初始化greenDAO时会建立数据库和相关的表），官方建议在Application中添加：



这里使用SharedPreferences是添加一个标记，为的是判断app是否是第一次启动，因为第一次启动要在表中初始化一些数据，这些数据只在app第一次启动时被添加一次。

这里初始化greenDAO是使用我们自定义的GreenDAOManager单例类来实现的：



实体类的代码：

@Entity(

nameInDb = “tab\_user”// 实体注解属性

)

public class User {

@Id(autoincrement = true) // 基础注解

private Long id;// 数据库中是主键时，这里必须要用Long型

...

@Property(nameInDb = "USERNAME") //@Property则表示该属性将作为表 //的一个字段，其中nameInDb看名字

private String username; //就知道这个属性在数据库中对应的数据名称

}

**——实体类中的相关注解解析**

实体注解属性：

schema：告知GreenDao当前实体属于哪个schema；

active：标记一个实体处于活动状态，活动实体有更新、删除和刷新方法；

nameInDb：在数据中使用的别名，默认使用的是实体的类名；

indexes：定义索引，可以跨越多个列；

createInDb：标记创建数据库表

基础注解：

@Id :主键 Long型，可以通过@Id(autoincrement = true)设置自增长；

@Property：设置一个非默认关系映射所对应的列名，默认是的使用字段名 举例：@Property (nameInDb="name")；

@NotNul：设置数据库表当前列不能为空；

@Transient ：添加次标记之后不会生成数据库表的列；

索引注解：

@Index：使用@Index作为一个属性来创建一个索引，通过name设置索引别名， 也可以通过unique给索引添加约束；

@Unique：向数据库列添加了一个唯一的约束；

关系注解：

@ToOne：定义与另一个实体（一个实体对象）的关系；

@ToMany：定义与多个实体对象的关系；

在上面的一系列配置和实体类代码编写完成后，点击“make project”，就会出现图3的所示的项目结构了，关于greenDAO的基本工作我们已经完成了，下面要做的就是通过greenDAO进行数据库的一系列操作了。

（Tips：

在笔者点击“make project”后，出现了

“\*\*\*Dao不是抽象的, 并且未覆盖AbstractDao中的抽象方法 hasKey(User)”，

我最后的解决办法是，将greenDAO版本换成3.0.0，删除之前自动生成的代 码，重新“make project”。）

**——增查改删**

首先获得\*\*\*Dao的对象，根据我们上面的代码：

UserDao userDao = GreenDAOManager.**getInstance**().**getDaoSession**().**getUserDao**();

增：

User admin = new User(null,**"admin"**,**"1"**,**1**);  
userDao.**insert**(admin);

查：

——查询全部数据：

**List**<User> users = userDao.**loadAll**();

——查询一个数据：

**User** user = userDao.**queryBuilder**().**where**(UserDao.Properties.*User\_name*.

**eq**(**"admin"**)).**unique**();

改：

根据ID更改：

**admin** = new User((long)**2**,**"anye0803"**,**"2"**,**2**);  
userDao.**update**(**admin**);

删：

删除数据和修改数据的思路一样，都是要先查找到数据：

**List**<User> userList = (**List**<User>)userDao.**queryBuilder**().

**where**(UserDao.Properties.*User\_id*.**le**(**10**)).**build**().**list**();  
 for (User user : userList) {  
 userDao.**delete**(user);  
 }

where表示查询条件，这里我是查询id小于等于10的数据，where中的参数可以有多个，就是说可以添加多个查询条件。最后的list表示查询结果是一个List集合，如果你只想查询一条数据，最后unique即可。当然，我们也可以根据id来删除数据：

**User** user = userDao.**queryBuilder**().

**where**(UserDao.Properties.*User\_id*.**eq**(**16**)).**build**().**unique**();

根据主键删除：

**userDao**.**deleteByKey**(*id*);

删除所有数据：

**userDao**.**deleteAll**();

**——使用原生SQL**

有的时候需要用到group by或者left join等复杂的语句，可以调用android原生的sqlite去进行查询：

**Cursor** cursor = userDao.**getDatabase**().**rawQuery**(**""**,null);  
while (cursor.**moveToNext**()){  
 ...  
}

**——数据库升级**

**一般升级（开发使用）**

如果某张表修改了字段，或者新增了一张表，必须要修改build.gradle中的schemaVersion，否则当你升级app的时候，如果进行了数据库操作，会发现列不匹配或者表不存在等问题，直接会导致app闪退。步骤：

1. 在module的build.gradle文件中修改版本号：

*//数据库schema版本，也可以理解为数据库版本号，数据库迁移、更新时会用到*

schemaVersion **2**

B.修改需要更改的实体类；

C.重新编译运行项目即可；

在这种情况升级下，其实是使用DevOpenHelper打开数据库，在它的onUpgrade方法中调用以下两个方法：

**@Override**public void onUpgrade(Database **db**, int **oldVersion**, int **newVersion**) {  
 Log.**i**(**"greenDAO"**, **"Upgrading schema from version "** + **oldVersion** + **" to "** + **newVersion** + **" by dropping all tables"**);  
 **dropAllTables**(**db**, true); // 调用此方法  
 **onCreate**(**db**); //在onCreate中调用**createAllTables**(**db**, false)方法  
}

现在我们只要看dropAllTables()与createAllTables()方法：

/\*\* Creates underlying database table using DAOs. \*/  
public static void createAllTables(Database **db**, boolean **ifNotExists**) {  
 UserDao.**createTable**(**db**, **ifNotExists**);  
 DeviceDao.**createTable**(**db**, **ifNotExists**);  
 RecordDao.**createTable**(**db**, **ifNotExists**);  
 CodeDao.**createTable**(**db**, **ifNotExists**);  
 TeamDao.**createTable**(**db**, **ifNotExists**);  
}  
  
/\*\* Drops underlying database table using DAOs. \*/  
public static void dropAllTables(Database **db**, boolean **ifExists**) {  
 UserDao.**dropTable**(**db**, **ifExists**);  
 DeviceDao.**dropTable**(**db**, **ifExists**);  
 RecordDao.**dropTable**(**db**, **ifExists**);  
 CodeDao.**dropTable**(**db**, **ifExists**);  
 TeamDao.**dropTable**(**db**, **ifExists**);  
}

很显然，这种方法每次升级时都会删除原来的表，然后创建新表，所以不会出现闪退等情况，不过只适用于开发阶段，当app上线后，我们就不能这样做了。

**安全升级（上线使用）**

但是如果仅仅是将schemaVersion加1，虽然程序不会崩溃，并且数据表的结构也会更新成功，但是之前表中的数据会全部清空。我们需要进行手动操作来进行数据库里面的数据迁移，大致的思路是：创建临时表（结构与上一版本的表结构相同），将旧数据移到临时表中，删除旧版本的表，创建新版本的表，将临时表中的数据转移到新表中，最后再删除临时表：

1. 封装一个自己类去继承DaoMaster.OpenHelper

public class MyOpenHelper extends DaoMaster.OpenHelper{  
 public MyOpenHelper(Context **context**, String **name**, SQLiteDatabase.CursorFactory **factory**)

{  
 super(**context**, **name**, **factory**);  
 }  
   
 **@Override** public void onUpgrade(Database **db**, int **oldVersion**, int **newVersion**) {  
 *//操作数据库的更新* MigrationHelper.**migrate**(**db**,

UserDao.class,CodeDao.class,

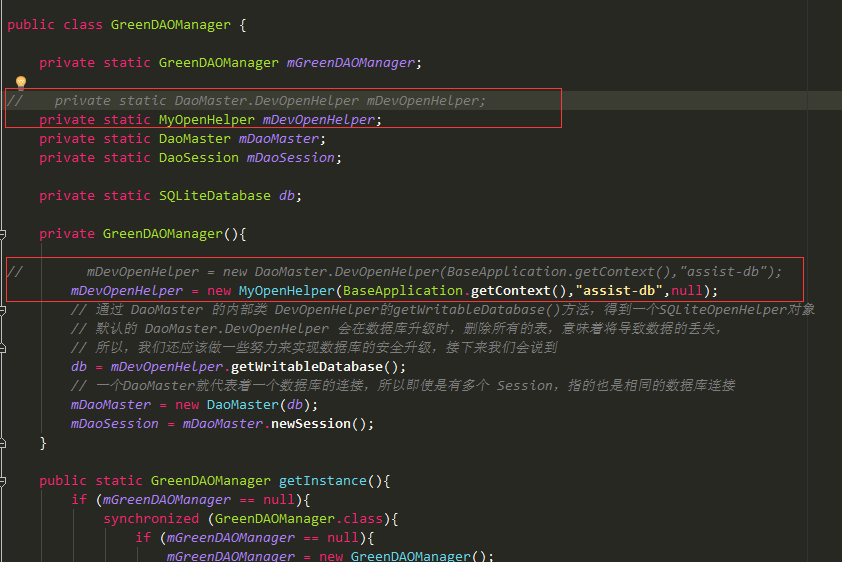
DeviceDao.class,RecordDao.class,

TeamDao.class);  
 }  
}

B.在onUpgrade()升级方法中去调用另外的自定义类MigrationHelper,在这个类中我们实现*“将旧数据移到临时表中，删除旧版本的表，创建新版本的表，将临时表中的数据转移到新表中，最后再删除临时表”*这个逻辑*：*

public class MigrationHelper {  
 /\*\*  
 \* 调用升级方法  
 \* **@param *db*** \* **@param *daoClasses*** 一系列dao.class  
 \*/  
 public static void migrate(Database **db**, Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 *//1 新建临时表* **generateTempTables**(**db**, **daoClasses**);  
 *//2 创建新表* **createAllTables**(**db**, false, **daoClasses**);  
 *//3 临时表数据写入新表，删除临时表* **restoreData**(**db**, **daoClasses**);  
 }  
 /\*\*  
 \* 生成临时表，存储旧的表数据  
 \* **@param *db*** \* **@param *daoClasses*** \*/  
 private static void generateTempTables(Database **db**, Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 for (int i=**0**;i<**daoClasses**.length;i++){  
 DaoConfig daoConfig = new DaoConfig(**db**,**daoClasses**[i]);  
 String tableName = daoConfig.tablename;  
 if (!**checkTable**(**db**,tableName))  
 continue;  
 String tempTableName = daoConfig.tablename.**concat**(**"\_TEMP"**);  
 StringBuilder insertTableStringBuilder = new StringBuilder();  
 insertTableStringBuilder.**append**(**"alter table "**)  
 .**append**(tableName)  
 .**append**(**" rename to "**)  
 .**append**(tempTableName)  
 .**append**(**";"**);  
 **db**.**execSQL**(insertTableStringBuilder.**toString**());  
 }  
 }  
 /\*\*  
 \* 检测table是否存在  
 \* **@param *db*** \* **@param *tableName*** \*/  
 private static Boolean checkTable(Database **db**,String **tableName**){  
 StringBuilder query = new StringBuilder();  
 query.**append**(**"SELECT count(\*) FROM sqlite\_master WHERE type='table' AND name='"**).**append**(**tableName**).**append**(**"'"**);  
 Cursor c = **db**.**rawQuery**(query.**toString**(), null);  
 if (c.**moveToNext**()){  
 int count = c.**getInt**(**0**);  
 if(count>**0**){  
 return true;  
 }  
 return false;  
 }  
 return false;  
 }  
 /\*\*  
 \* 删除所有旧表  
 \* **@param *db*** \* **@param *ifExists*** \* **@param *daoClasses*** \*/  
 private static void dropAllTables(Database **db**, boolean **ifExists**, **@NonNull** Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 **reflectMethod**(**db**, **"dropTable"**, **ifExists**, **daoClasses**);  
 }  
 /\*\*  
 \* 创建新的表结构  
 \* **@param *db*** \* **@param *ifNotExists*** \* **@param *daoClasses*** \*/  
 private static void createAllTables(Database **db**, boolean **ifNotExists**, **@NonNull** Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 **reflectMethod**(**db**, **"createTable"**, **ifNotExists**, **daoClasses**);  
 }  
  
 /\*\*  
 \* 创建根删除都在NoteDao声明了，可以直接拿过来用  
 \* dao class already define the sql exec method, so just invoke it  
 \*/  
 private static void reflectMethod(Database **db**, String **methodName**, boolean **isExists**, **@NonNull** Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 if (**daoClasses**.length < **1**) {  
 return;  
 }  
 try {  
 for (Class cls : **daoClasses**) {  
 *//根据方法名，找到声明的方法* Method method = cls.**getDeclaredMethod**(**methodName**, Database.class, boolean.class);  
 method.**invoke**(null, **db**, **isExists**);  
 }  
 } catch (NoSuchMethodException **e**) {  
 **e**.**printStackTrace**();  
 } catch (InvocationTargetException **e**) {  
 **e**.**printStackTrace**();  
 } catch (IllegalAccessException **e**) {  
 **e**.**printStackTrace**();  
 }  
 }  
 /\*\*  
 \* 临时表的数据写入新表  
 \* **@param *db*** \* **@param *daoClasses*** \*/  
 private static void restoreData(Database **db**, Class<? extends AbstractDao<?, ?>>... **daoClasses**) {  
 for (int i = **0**; i < **daoClasses**.length; i++) {  
 DaoConfig daoConfig = new DaoConfig(**db**, **daoClasses**[i]);  
 String tableName = daoConfig.tablename;  
 String tempTableName = daoConfig.tablename.**concat**(**"\_TEMP"**);  
 if (!**checkTable**(**db**,tempTableName))  
 continue;  
 *// get all columns from tempTable, take careful to use the columns list* List<String> columns = **getColumns**(**db**, tempTableName);  
 *//新表，临时表都包含的字段* ArrayList<String> properties = new ArrayList<>(columns.**size**());  
 for (int j = **0**; j < daoConfig.properties.length; j++) {  
 String columnName = daoConfig.properties[j].columnName;  
 if (columns.**contains**(columnName)) {  
 properties.**add**(columnName);  
 }  
 }  
 if (properties.**size**() > **0**) {  
 final String columnSQL = TextUtils.**join**(**","**, properties);  
  
 StringBuilder insertTableStringBuilder = new StringBuilder();  
 insertTableStringBuilder.**append**(**"INSERT INTO "**).**append**(tableName).**append**(**" ("**);  
 insertTableStringBuilder.**append**(columnSQL);  
 insertTableStringBuilder.**append**(**") SELECT "**);  
 insertTableStringBuilder.**append**(columnSQL);  
 insertTableStringBuilder.**append**(**" FROM "**).**append**(tempTableName).**append**(**";"**);  
 **db**.**execSQL**(insertTableStringBuilder.**toString**());  
 }  
 StringBuilder dropTableStringBuilder = new StringBuilder();  
 dropTableStringBuilder.**append**(**"DROP TABLE "**).**append**(tempTableName);  
 **db**.**execSQL**(dropTableStringBuilder.**toString**());  
 }  
 }  
 private static List<String> getColumns(Database **db**, String **tableName**) {  
 List<String> columns = null;  
 Cursor cursor = null;  
 try {  
 cursor = **db**.**rawQuery**(**"SELECT \* FROM "** + **tableName** + **" limit 0"**, null);  
 if (null != cursor && cursor.**getColumnCount**() > **0**) {  
 columns = Arrays.**asList**(cursor.**getColumnNames**());  
 }  
 } catch (Exception **e**) {  
 **e**.**printStackTrace**();  
 } finally {  
 if (cursor != null)  
 cursor.**close**();  
 if (null == columns)  
 columns = new ArrayList<>();  
 }  
 return columns;  
 }  
}

C.将我们在初始化greenDAO操作中的DaoMaster.DevOpenHelper替换成我们自定义的MyOpenHelper实例,这样在升级数据库时，就会按照我们自定义类中*“将旧数据移到临时表中，删除旧版本的表，创建新版本的表，将临时表中的数据转移到新表中，最后再删除临时表”*逻辑运行，



经测试，数据库升级后（无论表结构是否变化），原本的数据都仍然存在。

***——参考文章***

*数据库升级部分*

*<http://hammercui.github.io/post/android%E6%9C%80%E4%BD%B3%E5%AE%9E%E8%B7%B59-2%EF%BC%9AgreenDao%E7%9A%84%E6%95%B0%E6%8D%AE%E5%BA%93%E5%8D%87%E7%BA%A7/>*

*<http://stackoverflow.com/questions/13373170/greendao-schema-update-and-data-migration/30334668#30334668>*

*非升级部分*

*https://www.daidingkang.cc/2016/12/08/GreenDao/*

*<http://blog.csdn.net/njweiyukun/article/details/51893092>*

## 依赖注入（Dependency Injection）

*（https://zhuanlan.zhihu.com/p/24454466）*

***概念***

什么是依赖注入，我们通过代码来解释一下：

public class Car{

private Engine engine;

public Car(){

engine = new Engine();

}

}

这段Java代码中Car类持有了对Engine实例的引用，我们称之为Car类对Engine类有了依赖，但是仅仅是*依赖*，这段代码并没有体现出*注入*。因为我们需要在Car的构造函数中去自己创建Engine对象实例，而不是将Engine实例作为参数传入给Car类。

***三种依赖注入***

依赖注入是指通过注入的方式实现类与类之间的依赖，下面是常见的三种依赖注入方式：

——构造注入

通过构造函数传参给依赖的成员变量赋值，从而实现注入：

public class Car{

private Engine engine;

public Car(Engine engine){

this.engine = engine;

}

}

——接口注入

实现接口方法，同样以传参的方式实现注入：

public interface Injection<T>{

void inject(T t);

}

public class Car implements Injection<Engine>{

private Engine engine;

public Car(){}

public void inject(Engine engine){

this.engine = engine;

}

}

——注解注入

使用Java注解在编译阶段生成代码实现注入或者是在运行阶段通过反射实现注入：

public class Car{

@Inject

Engine engine;

public Car(){}

}

前两种注入方式需要我们编写大量的模板代码，而注解的方式则省时省力。

***为什么要用依赖注入***

我们之所是要依赖注入，最重要的就是为了解耦，达到高内聚低耦合的目的，保证代码的健壮性、灵活性和可维护性。

下面我们看看同一个业务的两种实现方案：

A.

public class Car{

private Engine engine;

private List<Wheel> wheels;

public Car(){

engine = new Engine();

wheels = new ArrayList<>();

for(int i = 0; i < 4; i++){

wheels.add(new Wheel());

}

}

public void start{

System.out.println("启动汽车");

}

}

public class CarTest{

public static void main(String[] args){

Car car = new Car();

car.start();

}

}

B.

public class Car{

private Engine engine;

private List<Wheel> wheels;

public Car(Engine engine, List<Wheel> wheels){

this.engine = engine;

this.wheels = wheels;

}

public void start{

System.out.println("启动汽车");

}

}

public class CarTest{

public static void main(String[] args){

Engine engine = new Engine();

List<Wheel> wheels = new ArrayList<>();

for(int i = 0; i < 4; i++){

wheels.add(new Wheel());

}

Car car = new Car(engine, wheels);

car.start();

}

}

A：由于没有依赖注入，因此需要我们自己是在Car的构造函数中创建Engine和 Wheel对象；

B：我们手动以构造函数的方式注入依赖，将engine和wheels作为参数传入而 不是在Car的构造函数中去显示的创建。

方案A明显丧失了灵活性，一切依赖都是在Car类的内部创建，Car与Engine和Wheel严重耦合。一旦Engine或者Wheel的创建方式发生了改变，我们就必须要去修改Car类的构造函数（比如说现在创建Wheel实例的构造函数改变了，需要传入Rubber（橡胶）了）；另外我们也没办法替换动态的替换依赖实例（比如我们想把Car的Wheel（轮胎）从邓禄普（轮胎品牌）换成米其林（轮胎品牌）的）。

这类问题在大型的商业项目中则更加严重，往往A依赖B、B依赖C、C依赖D、D依赖E；一旦稍有改动便牵一发而动全身，想想都可怕！而依赖注入则很好的帮我们解决了这一问题。

## getDefaultSharedPreferences()与getSharedPreferences()的区别

***getSharedPreferences()***

——Context中的getSharedPreferences()方法

/\*\*  
 \* Retrieve and hold the contents of the preferences file 'name', returning  
 \* a SharedPreferences through which you can retrieve and modify its  
 \* values. Only one instance of the SharedPreferences object is returned  
 \* to any callers for the same name, meaning they will see each other's  
 \* edits as soon as they are made.  
 \*  
 \* **@param *name*** Desired preferences file. If a preferences file by this name  
 \* does not exist, it will be created when you retrieve an  
 \* editor (SharedPreferences.edit()) and then commit changes (Editor.commit()).  
 \* **@param *mode*** Operating mode. Use 0 or {**@link** #MODE\_PRIVATE} for the  
 \* default operation.  
 \*  
 \* **@return** The single {**@link** SharedPreferences} instance that can be used  
 \* to retrieve and modify the preference values.  
 \*  
 \* **@see** #MODE\_PRIVATE  
 \*/  
public abstract SharedPreferences getSharedPreferences(String **name**, int **mode**);

这是一个抽象方法，返回一个SharedPreferences对象，根据注释可以看出：

“通过这个方法，可以得到名为‘name’的preferences文件。同时可以更改和得到他里面的值。任何调用者在调用同样名字的偏好文件时只有一个实例返回，这就意味着这些调用者都可以看到其他调用者做出的更改。”

参数name：

期望的preferences文件，如果此name的文件不存在，则会在我们得到editor时（(SharedPreferences.edit()) and then commit changes (Editor.commit())）自动创建。

——PreferenceManager中的getSharedPreferences()方法

/\*\*  
 \* Gets a SharedPreferences instance that preferences managed by this will  
 \* use.  
 \*   
 \* **@return** A SharedPreferences instance pointing to the file that contains  
 \* the values of preferences that are managed by this.  
 \*/  
public SharedPreferences getSharedPreferences() {  
 if (mSharedPreferences == null) {  
 final Context storageContext;  
 switch (mStorage) {  
 case *STORAGE\_DEVICE\_PROTECTED*:  
 storageContext = mContext.**createDeviceProtectedStorageContext**();  
 break;  
 case *STORAGE\_CREDENTIAL\_PROTECTED*:  
 storageContext = mContext.createCredentialProtectedStorageContext();  
 break;  
 default:  
 storageContext = mContext;  
 break;  
 }  
  
 mSharedPreferences = storageContext.**getSharedPreferences**(mSharedPreferencesName,  
 mSharedPreferencesMode);  
 }  
   
 return mSharedPreferences;  
}

这是一个公共非静态方法，需要PreferenceManager的实例才能调用，接下来我们去看PreferenceManager的构造方法：

/\*\*  
 \* **@hide** \*/  
public PreferenceManager(Activity **activity**, int **firstRequestCode**) {  
 mActivity = **activity**;  
 mNextRequestCode = **firstRequestCode**;  
 **init**(**activity**);  
}

在构造方法中又调用了init()方法：

private void init(Context **context**) {  
 mContext = **context**;  
 **setSharedPreferencesName**(**getDefaultSharedPreferencesName**(**context**));  
}

接着调用了setSharedPreferencesName()方法：

/\*\*  
 \* Sets the name of the SharedPreferences file that preferences managed by this  
 \* will use.  
 \*   
 \* **@param *sharedPreferencesName*** The name of the SharedPreferences file.  
 \* **@see** Context#**getSharedPreferences**(String, int)  
 \*/  
public void setSharedPreferencesName(String **sharedPreferencesName**) {  
 mSharedPreferencesName = **sharedPreferencesName**;  
 mSharedPreferences = null;  
}

和getDefaultSharedPreferencesName()方法：

/\*\*  
 \* Returns the name used for storing default shared preferences.  
 \*  
 \* **@see** #**getDefaultSharedPreferences**(Context)  
 \* **@see** Context#getSharedPreferencesPath(String)  
 \*/  
public static String getDefaultSharedPreferencesName(Context **context**) {  
 return **context**.**getPackageName**() + **"\_preferences"**;  
}

在这里，我们终于看到，PreferenceManager的getSharedPreferences()方法最终是得到一个名为“packageName\_preferences”的文件。

***getDefaultSharedPreferences()***

他是PreferenceManager中的公共静态方法：

/\*\*  
 \* Gets a SharedPreferences instance that points to the default file that is  
 \* used by the preference framework in the given context.  
 \*   
 \* **@param *context*** The context of the preferences whose values are wanted.  
 \* **@return** A SharedPreferences instance that can be used to retrieve and  
 \* listen to values of the preferences.  
 \*/  
public static SharedPreferences getDefaultSharedPreferences(Context **context**) {  
 return **context**.**getSharedPreferences**(**getDefaultSharedPreferencesName**(**context**),  
 **getDefaultSharedPreferencesMode**());  
}

从代码中可以看出，最后调用的是Context类的getSharedPreferences()方法，但是方法中的两个参数已经确定了，分别是：

getDefaultSharedPreferencesName():

/\*\*  
 \* Returns the name used for storing default shared preferences.  
 \*  
 \* **@see** #**getDefaultSharedPreferences**(Context)  
 \* **@see** Context#getSharedPreferencesPath(String)  
 \*/  
public static String getDefaultSharedPreferencesName(Context **context**) {  
 return **context**.**getPackageName**() + **"\_preferences"**;  
}

getDefaultSharedPreferencesMode():

private static int getDefaultSharedPreferencesMode() {  
 return Context.*MODE\_PRIVATE*;  
}

所以getDefaultSharedPreferences()方法最后得到的还是一个名为

“packageName\_preferences”的文件，并且属性私有。

***总结***

得到SharedPreferences实例的方法有三个：

PreferenceManager中提供了两个，分别是

getSharedPreferences()与getDefaultSharedPreferences(Context context)方法，不过都是操作的同一个名为“packageName\_preferences”且属性私有的preferences文件；

Context中提供了一个，

getSharedPreferences(String name,int mode)，我们可以通过传入不同的参数，来创建不同的preferences文件或是得到其中的值（values）；

这三个方法都返回一个SharedPreferences对象，那这个SharedPreferences究竟是什么呢？

它其实是一个轻量级的存储类，特别适合用于保存软件配置参数。使用SharedPreferences保存数据，其背后是用xml文件存放数据，文件存放在/data/data/<package name>/shared\_prefs目录下

***Tips***

——注意上述三个方法的参数上的区别；

——Activity也提供一个返回SharedPreferences的方法getPreferences(int mode)， 这个方法默认使用当前的类名（不带包名）作为文件的名称；

# 2017-3-20

## AndroidStudio获取开发版和发布版的SHA1

*(http://blog.csdn.net/jdsjlzx/article/details/52144870)*

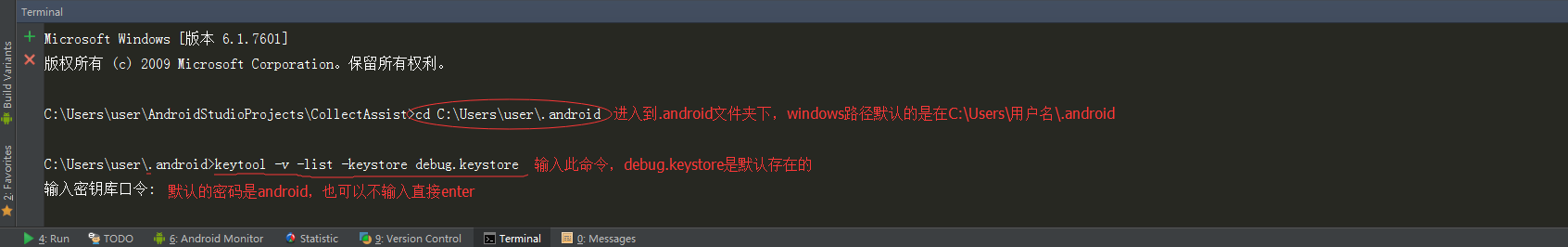
***前言***

Android应用开发的时候会经常使用到第三方SDK，例如地图、推送、支付以及分享等。而这些第三方服务SDK为了应用不被滥用都会有签名验证机制，我们通过上传Android应用的签名来在服务端配置以验证应用的合法性。

在集成百度地图SDK时，需要用到应用的开发版和发布版的SHA1，下面我们来说明一下怎么在androidstudio中获取这两个值。

***获取开发版SHA1***

在Terminal中输入相关信息：

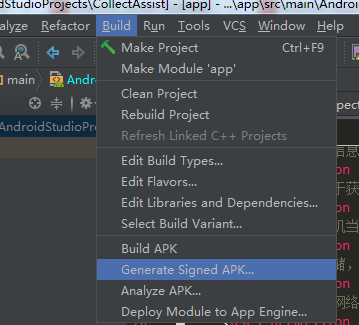


接着就能看到相应的信息：

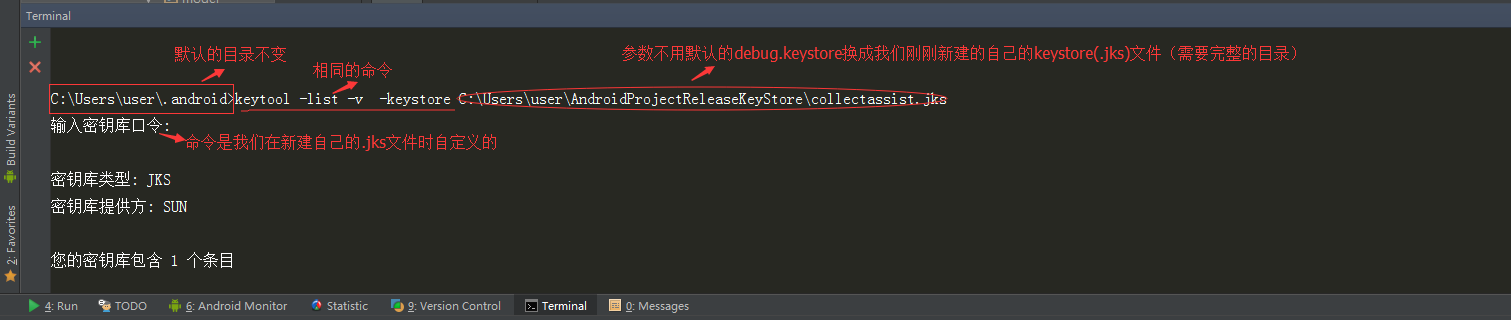


***获取发布版SHA1***

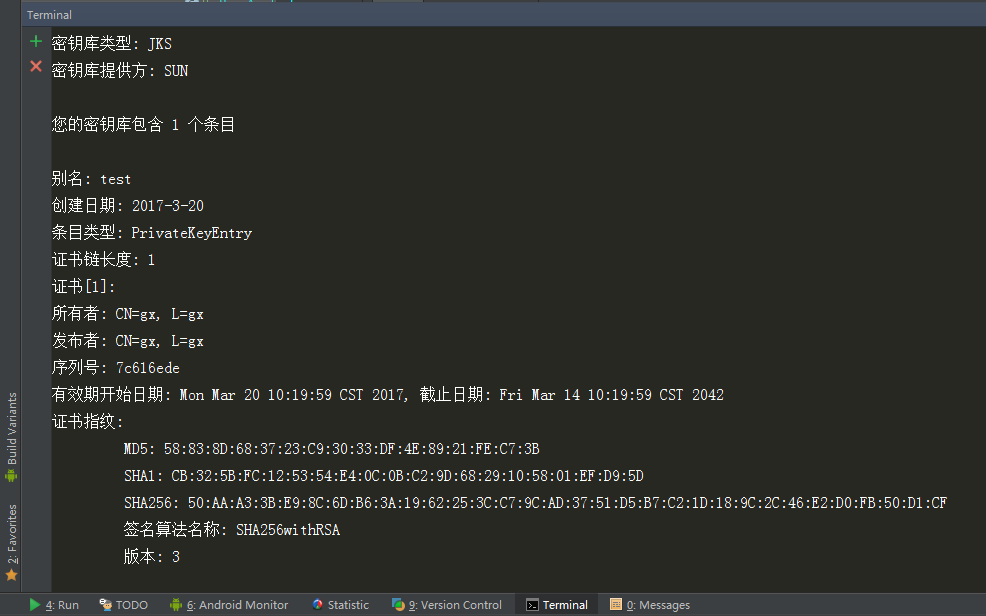
发布版的SHA1获取需要我们新建自己的keystore（.jks）文件，通过Build——>Generate Signed APK...获取：



具体步骤就不详述了，接下来同样在Terminal窗口中输入命令：



弹出相应的发布版签名信息：



有了开发版和发布版的各种签名信息（MD5、SHA1、SHA256）之后，我们就可以做很多事情了。

## RxBus的简单使用

***前言***

随着RxJava的流行，现在越来越多的使用RxBus来取代EventBus与Otto了，需要注意的是后两者都是第三方的事件总线库，而RxBus是我们自己编写的java文件,它的核心是RxJava。

***实现***

**——添加依赖**

在module的build.gradle文件中：

**compile 'io.reactivex:rxandroid:1.1.0'  
compile 'io.reactivex:rxjava:1.1.0'**

**——编写RxBus类**

public class RxBus {  
 private static RxBus *mInstance*;  
 private Subject \_bus;  
 private RxBus(){  
 *// ReplaySubject在订阅者订阅时，会发送所有的数据给订阅者，无论它们是何时订阅的  
 // PublishSubject只会给在订阅者订阅的时间点之后的数据发送给观察者  
 // BehaviorSubject在订阅者订阅时，会发送其最近发送的数据（如果此时还没有收到任何数据，它会发送一个默认值）  
 // AsyncSubject只在原Observable事件序列完成后，发送最后一个数据，后续如果还有订阅者继续订阅该Subject, 则可以直接接收到最后一个值* \_bus = new SerializedSubject<>(BehaviorSubject.**create**());

// 这里由于我们是先发送再订阅的，如果使用PublishSubject就会接收不到  
 }  
 public static RxBus getInstance(){  
 if (*mInstance* == null){  
 synchronized (RxBus.class){  
 if (*mInstance* == null){  
 *mInstance* = new RxBus();  
 }  
 }  
 }  
 return *mInstance*;  
 }  
 public void send(Object **o**){  
 \_bus.**onNext**(**o**);  
 }  
 public<**T**> Observable<**T**> toObservable(Class<**T**> **eventType**){  
 // 根据传递的 eventType 类型返回特定类型(eventType)的 被观察者

// return bus.ofType(eventType);

// ofType = filter + cast

// return bus.filter(new Func1<Object, Boolean>() {

// @Override

// public Boolean call(Object o) {

// return eventType.isInstance(o);

// }

// }) .cast(eventType);

return \_bus;  
 }

// 也可以用无参的toObservable()方法

// public Observable<Object> toObservable() {

// return \_bus;

// }

public boolean hasObservers(){  
 return \_bus.**hasObservers**();  
 }  
}

**——发送事件**

RxBus.**getInstance**().**send**(new User(userId,username,password,teamId));

这里是发送一个User对象。

**——接收事件**

private Subscription mSubscription;

**mSubscription** = RxBus.**getInstance**().**toObservable**(User.class).**subscribe**(new Action1<User>() {  
 **@Override** public void call(User **user**) { // 接收到User对象  
 userId = **user**.**getUser\_id**();  
 }  
});

// 如果我们使用的是无参的toObservable()方法，在接收时代码这样实现

// RxBus.getDefault().toObservable().subscribe(new Action1<Object>() {

// @Override

// public void call(Object event) {

// if (event instanceof TapEvent) { // 加个判断

// //do something

// }else if(event instanceof otherEvent){

// //do otherthing

// }

// }

// });

**——取消订阅**

通常在OnDestroy()方法中：

if (!mSubscription.**isUnsubscribed**()){  
 mSubscription.**unsubscribe**();  
}

***总结***

这里只是一个简单的使用，关于RxBus的知识还有很多，例如Sticky事件等，最重要的还是要先掌握好RxJava。关于Sticky事件，可以参考http://www.jianshu.com/p/71ab00a2677b。