Android 常见问题及解决方法（一）

1、Java 工程中调用 Android 库出现“Stub!”错误

描述：控制台显示错误：Exception in thread “main” java.lang.RuntimeException: Stub!

原因：在 Java 工程中尝试使用 Android 库中的 org.json.JSONObject 类，在执行时出现“Stub！”错误，Android 工程中无法执行 java 的 main 函数相似，Android 工程和 Java 工程还有一定的差异，不能混用他们的库，和函数入口方法。

解决：将执行的代码，移植到在 Android 工程可以正确执行！

2、使用 shape 的同时能通过代码修改 shape 的颜色属性

描述：有时会遇到这种需求：不同状态的背景标识不同，并且背景有特定的 shape 样式。

原因：一般的 shape 文件都是在 xml 中固定好颜色，从而需要在代码中修改 shape 文件中的颜色值。

解决：直接通过控件获取该控件的背景，通过更改背景颜色从而更改 shape 文件中的颜色，代码如下所示：

```
GradientDrawable gradientDrawable = (GradientDrawable)view.getBackgrou
nd();gradientDrawable.setColor(color);12
```

3、Failure [INSTALL_FAILED_OLDER_SDK]

描述：编译的时候，报 Failure [INSTALL_FAILED_OLDER_SDK]错误。

原因：一般是系统自动帮你设置了 compileSdkVersion，且版本过高导致的错误。

解决：修改 build.gradle 下的 compileSdkVersion xxx 为 compileSdkVersion 19（或者你本机已有的 SDK 即可）.

4、Popupwindow 使用异常：unable to add window - token null is not valid

描述：Popupwindow 必须依赖一个 view 进行弹窗，`void android.widget.PopupWindow.showAtLocation(View parent, int gravity, int x, int y)`

调用这个方法就能显示 Popupwindow 了，但是有时会碰到这样一个异常：unable to add window – token null is not valid;is your activity running?

原因： 导致这个的原因一般是 Activity 的 onCreate()函数里面调用了 showAtLocation，由于你的 popupwindow 要依附于一个 activity，而 activity 的 onCreate()还没执行完就需要弹窗肯定会出问题的。

解决： 在 Handler 中进行弹窗，在 onCreate 中通过延时调用就 OK 了，具体代码如下：

```
private Handler popupHandler = new Handler() {     @Override
  public void handleMessage(Message msg) {       switch (msg.what) {
case 0:
        popupWindow.showAtLocation(findViewById(R.id.rlShowImage), Gravity.CENTER|Gravity.CENTER, 0, 0);
        popupWindow.update();          break;
    }
  }
};1234567891011
popupHandler.sendEmptyMessageDelayed(0, 1000);1
```

5、使用 shape 绘制虚线时，在 4.0 以上机型上显示实线

描述： 在利用 shape 绘制虚线时，在 Graphical Layout 中能正常显示，但在 Android4.0 上的机型显示成了实线。

原因： 4.0 以上默认把 Activity 的硬件加速打开了，所以我们在 Manifest.xml 中关掉即可。

解决： 在需要显示的 activity 中增加如下属性：android:hardwareAccelerated=”false”，也可以通过从 View 层级上把硬件加速关掉 view.setLayerType(View.LAYER_TYPE_SOFTWARE, null)。

6、Application does not specify an API level requirement

描述： 编译时报警告：Application does not specify an API level requirement!

原因：在 AndroidManifest.xml 或者 build.gradle 文件中没有添加 API 版本号，不影响运行。

解决：在对应的地方添加上 minSdkVersion 和 targetSdkVersion 的版本号就行。

7、Installation error: INSTALL_FAILED_INSUFFICIENT_STORAGE

描述：运行时报错：Installation error: INSTALL_FAILED_INSUFFICIENT_STORAGE。

原因：一般应用默认安装都是手机存储空间，而该设备没有足够的存储空间来安装应用程序。

解决：一般手机都有 SD 卡，可以在 AndroidManifest.xml 文件中设置属性 android:installLocation=”auto”就行了。

8、Android 图片加载 Bitmap OOM 错误解决办法

描述： Android 加载资源图片时，很容易出现 OOM 的错误，因为 Android 系统对内存有一个限制，如果超出该限制，就会出现 OOM，为了避免这个问题，需要在加载资源时尽量考虑如何节约内存，尽快释放资源等等。

原因： Android 系统版本对图片加载回收的不同：

（1）在 Android 2.3 以及之后，采用的是并发回收机制，避免在回收内存时的卡顿现象；

（2）在 Android 2.3.3(API Level 10)以及之前，Bitmap 的 backing pixel 数据存储在 native memory，与 Bitmap 本身是分开的,Bitmap 本身存储在 dalvik heap 中，导致其 pixel 数据不能判断是否还需要使用，不能及时释放，容易引起 OOM 错误，从 Android 3.0(API 11)开始，pixel 数据与 Bitmap 一起存储在 Dalvik heap 中。

解决：在加载图片资源时，可采用以下一些方法来避免 OOM 的问题：

（1）在 Android 2.3.3 以及之前，建议使用 Bitmap.recycle()方法，及时释放资源；

（2）在 Android 3.0 开始，可设置 BitmapFactory.options.inBitmap 值，(从缓存中获取)达到重用 Bitmap 的目的，如果设置，则 inPreferredConfig 属性值会被重用的 Bitmap 该属性值覆盖；

（3）通过设置 Options.inPreferredConfig 值来降低内存消耗：

默认为 ARGB_8888：每个像素 4 字节．共 32 位。

Alpha_8：只保存透明度，共 8 位，1 字节。

ARGB_4444：共 16 位，2 字节。

RGB_565:共 16 位，2 字节。

如果不需要透明度，可把默认值 ARGB_8888 改为 RGB_565,节约一半内存。

（4）通过设置 Options.inSampleSize 对大图片进行压缩，可先设置 Options.inJustDecodeBounds，获取 Bitmap 的外围数据，宽和高等。然后计算压缩比例，进行压缩；

（5）设置 Options.inPurgeable 和 inInputShareable：让系统能及时回收内存。

inPurgeable:设置为 True,则使用 BitmapFactory 创建的 Bitmap 用于存储 Pixel 的内存空间，在系统内存不足时可以被回收，当应用需要再次访问该 Bitmap 的 Pixel 时，系统会再次调用 BitmapFactory 的 decode 方法重新生成 Bitmap 的 Pixel 数组；设置为 False 时，表示不能被回收。

inInputShareable：设置是否深拷贝，与 inPurgeable 结合使用，inPurgeable 为 false 时，该参数无意义；True： share a reference to the input data(inputStream, array,etc) 。 False ：a deep copy。

（6）使用 decodeStream 代替其他 decodeResource,setImageResource,setImageBitmap 等方法来加载图片。

区别：

decodeStream 直接读取图片字节码，调用 nativeDecodeAsset/nativeDecodeStream 来完成 decode,无需使用 Java 空间的一些额外处理过程，节省 dalvik 内存。但是由于直接读取字节码，没有处理过程，因此不会根据机器的各种分辨率来自动适应，需要在 hdpi,mdpi 和 ldpi 中分别配置相应的图片资源，否则在不同分辨率机器上都是同样的大小(像素点数量)，显示的实际大小不对；

decodeResource 会在读取完图片数据后，根据机器的分辨率，进行图片的适配处理，导致增大了很多 dalvik 内存消耗；

decodeStream 调用过程： decodeStream(InputStream,Rect,Options) -> nativeDecodeAsset/nativeDecodeStream;

decodeResource 调用过程：即 finishDecode 之后，调用额外的 Java 层的 createBitmap 方法，消耗更多 dalvik 内存;

decodeResource(Resource,resId,Options) -> decodeResourceStream（设置 Options 的 inDensity 和 inTargetDensity 参数） -> decodeStream()（在完成 Decode 后，进行 finishDecode 操作）finishDecode() -> Bitmap.createScaleBitmap()(根据 inDensity 和 inTargetDensity 计算 scale) -> Bitmap.createBitmap()。

以上方法的组合使用，合理避免 OOM 错误。