**Context都没弄明白，还怎么做Android开发？**

Context的两个子类分工明确，其中ContextImpl是Context的具体实现类，ContextWrapper是Context的包装类。Activity，Application，Service虽都继承自ContextWrapper（Activity继承自ContextWrapper的子类ContextThemeWrapper），但它们初始化的过程中都会创建ContextImpl对象，由ContextImpl实现Context中的方法。

**一个应用程序有几个Context**

在应用程序中Context的具体实现子类就是：Activity，Service，Application。那么Context数量=Activity数量+Service数量+1。当然如果你足够细心，可能会有疑问：我们常说四大组件，这里怎么只有Activity，Service持有Context，那Broadcast Receiver，Content Provider呢？Broadcast Receiver，Content Provider并不是Context的子类，他们所持有的Context都是其他地方传过去的，所以并不计入Context总数。

**Context能干什么**

Context到底可以实现哪些功能呢？这个就实在是太多了，弹出Toast、启动Activity、启动Service、发送广播、操作数据库等等都需要用到Context。

TextView tv = new TextView(getContext());

ListAdapter adapter = new SimpleCursorAdapter(getApplicationContext(), ...);

AudioManager am = (AudioManager) getContext().getSystemService(Context.AUDIO\_SERVICE);

getApplicationContext().getSharedPreferences(name, mode);

getApplicationContext().getContentResolver().query(uri, ...);

getContext().getResources().getDisplayMetrics().widthPixels \* 5 / 8;

getContext().startActivity(intent);

getContext().startService(intent);

getContext().sendBroadcast(intent);

**Context作用域**

虽然Context神通广大，但并不是随便拿到一个Context实例就可以为所欲为，它的使用还是有一些规则限制的。由于Context的具体实例是由ContextImpl类去实现的，因此在绝大多数场景下，Activity、Service和Application这三种类型的Context都是可以通用的。不过有几种场景比较特殊，比如启动Activity，还有弹出Dialog。出于安全原因的考虑，Android是不允许Activity或Dialog凭空出现的，一个Activity的启动必须要建立在另一个Activity的基础之上，也就是以此形成的返回栈。而Dialog则必须在一个Activity上面弹出（除非是System Alert类型的Dialog），因此在这种场景下，我们只能使用Activity类型的Context，否则将会出错。



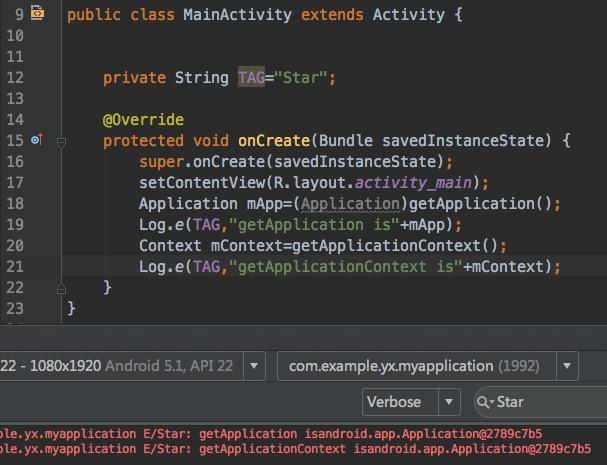
从上图我们可以发现Activity所持有的Context的作用域最广，无所不能。因为Activity继承自ContextThemeWrapper，而Application和Service继承自ContextWrapper，很显然ContextThemeWrapper在ContextWrapper的基础上又做了一些操作使得Activity变得更强大。上图中的YES和NO我也不再做过多的解释了，这里我说一下上图中Application和Service所不推荐的两种使用情况。  
1：如果我们用ApplicationContext去启动一个LaunchMode为standard的Activity的时候会报错android.util.AndroidRuntimeException: Calling startActivity from outside of an Activity context requires the FLAG\_ACTIVITY\_NEW\_TASK flag. Is this really what you want?这是因为非Activity类型的Context并没有所谓的任务栈，所以待启动的Activity就找不到栈了。解决这个问题的方法就是为待启动的Activity指定FLAG\_ACTIVITY\_NEW\_TASK标记位，这样启动的时候就为它创建一个新的任务栈，而此时Activity是以singleTask模式启动的。所有这种用Application启动Activity的方式不推荐使用，Service同Application。  
2：在Application和Service中去layout inflate也是合法的，但是会使用系统默认的主题样式，如果你自定义了某些样式可能不会被使用。所以这种方式也不推荐使用。  
一句话总结：凡是跟UI相关的，都应该使用Activity做为Context来处理；其他的一些操作，Service,Activity,Application等实例都可以，当然了，注意Context引用的持有，防止内存泄漏。

**如何获取Context**

通常我们想要获取Context对象，主要有以下四种方法  
1：View.getContext,返回当前View对象的Context对象，通常是当前正在展示的Activity对象。  
2：Activity.getApplicationContext,获取当前Activity所在的(应用)进程的Context对象，通常我们使用Context对象时，要优先考虑这个全局的进程Context。  
3：ContextWrapper.getBaseContext():用来获取一个ContextWrapper进行装饰之前的Context，可以使用这个方法，这个方法在实际开发中使用并不多，也不建议使用。  
4：Activity.this 返回当前的Activity实例，如果是UI控件需要使用Activity作为Context对象，但是默认的Toast实际上使用ApplicationContext也可以。

getApplication()和getApplicationContext()

上面说到获取当前Application对象用getApplicationContext，不知道你有没有联想到getApplication()，这两个方法有什么区别？相信这个问题会难倒不少开发者。



程序是不会骗人的，我们通过上面的代码，打印得出两者的内存地址都是相同的，看来它们是同一个对象。其实这个结果也很好理解，因为前面已经说过了，Application本身就是一个Context，所以这里获取getApplicationContext()得到的结果就是Application本身的实例。那么问题来了，既然这两个方法得到的结果都是相同的，那么Android为什么要提供两个功能重复的方法呢？实际上这两个方法在作用域上有比较大的区别。getApplication()方法的语义性非常强，一看就知道是用来获取Application实例的，但是这个方法只有在Activity和Service中才能调用的到。那么也许在绝大多数情况下我们都是在Activity或者Service中使用Application的，但是如果在一些其它的场景，比如BroadcastReceiver中也想获得Application的实例，这时就可以借助getApplicationContext()方法了。

public class MyReceiver extends BroadcastReceiver{

@Override

Public void onReceive(Context context,Intent intent){

Application myApp=(Application)context.getApplicationContext();

}

}

**Context引起的内存泄露**

但Context并不能随便乱用，用的不好有可能会引起内存泄露的问题，下面就示例两种错误的引用方式。

错误的单例模式

public class Singleton {

private static Singleton instance;

private Context mContext;

private Singleton(Context context) {

this.mContext = context;

}

public static Singleton getInstance(Context context) {

if (instance == null) {

instance = new Singleton(context);

}

return instance;

}

}

这是一个非线程安全的单例模式，instance作为静态对象，其生命周期要长于普通的对象，其中也包含Activity，假如Activity A去getInstance获得instance对象，传入this，常驻内存的Singleton保存了你传入的Activity A对象，并一直持有，即使Activity被销毁掉，但因为它的引用还存在于一个Singleton中，就不可能被GC掉，这样就导致了内存泄漏。

View持有Activity引用

public class MainActivity extends Activity {

private static Drawable mDrawable;

@Override

protected void onCreate(Bundle saveInstanceState) {

super.onCreate(saveInstanceState);

setContentView(R.layout.activity\_main);

ImageView iv = new ImageView(this);

mDrawable = getResources().getDrawable(R.drawable.ic\_launcher);

iv.setImageDrawable(mDrawable);

}

}

有一个静态的Drawable对象当ImageView设置这个Drawable时，ImageView保存了mDrawable的引用，而ImageView传入的this是MainActivity的mContext，因为被static修饰的mDrawable是常驻内存的，MainActivity是它的间接引用，MainActivity被销毁时，也不能被GC掉，所以造成内存泄漏。

**正确使用Context**

一般Context造成的内存泄漏，几乎都是当Context销毁的时候，却因为被引用导致销毁失败，而Application的Context对象可以理解为随着进程存在的，所以我们总结出使用Context的正确姿势：  
1：当Application的Context能搞定的情况下，并且生命周期长的对象，优先使用Application的Context。  
2：不要让生命周期长于Activity的对象持有到Activity的引用。  
3：尽量不要在Activity中使用非静态内部类，因为非静态内部类会隐式持有外部类实例的引用，如果使用静态内部类，将外部实例引用作为弱引用持有。