前言：本文是我读《Android内核剖析》第7章 后形成的读书笔记 ，在此向欲了解Android框架的书籍推荐此书。

        大家好，  今天给大家介绍下我们在应用开发中最熟悉而陌生的朋友-----Context类 ，说它熟悉，是应为我们在开发中

   时刻的在与它打交道，例如：Service、BroadcastReceiver、Activity等都会利用到Context的相关方法 ； 说它陌生，完全是因为我们真正的不懂Context的原理、类结构关系。一个简单的问题是，一个应用程序App中存在多少个Context实例对象呢？

   一个、两个？ 在此先卖个关子吧。读了本文，相信您会豁然开朗的 。

      Context，中文直译为“上下文”，SDK中对其说明如下：

         Interface to global information about an application environment. This is an abstract class whose implementation

  is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls

  for application-level operations such as launching activities, broadcasting and receiving intents, etc

    从上可知一下三点,即：

        1、它描述的是一个应用程序环境的信息，即上下文。

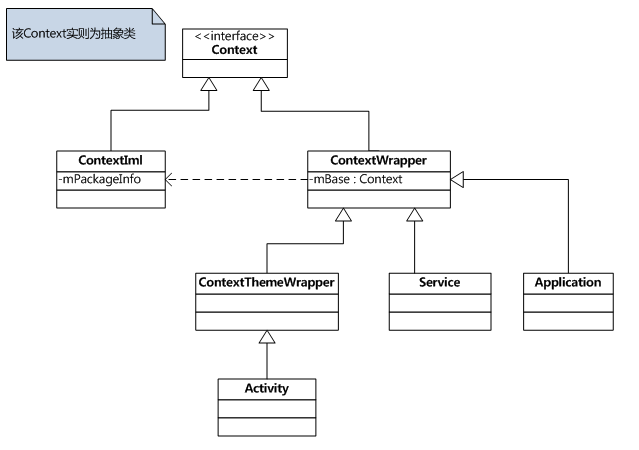
        2、该类是一个抽象(abstract class)类，Android提供了该抽象类的具体实现类(后面我们会讲到是ContextIml类)。

        3、通过它我们可以获取应用程序的资源和类，也包括一些应用级别操作，例如：启动一个Activity，发送广播，接受Intent

      信息 等。。

   于是，我们可以利用该Context对象去构建应用级别操作(application-level operations) 。

**一、Context相关类的继承关系**



**相关类介绍：**

**Context类**    路径： /frameworks/base/core/java/android/content/Context.java

            说明：  抽象类，提供了一组通用的API。

      源代码(部分)如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. **public** **abstract** **class** Context {
2. ...
3. **public** **abstract** Object getSystemService(String name);  //获得系统级服务
4. **public** **abstract** **void** startActivity(Intent intent);     //通过一个Intent启动Activity
5. **public** **abstract** ComponentName startService(Intent service);  //启动Service
6. //根据文件名得到SharedPreferences对象
7. **public** **abstract** SharedPreferences getSharedPreferences(String name,**int** mode);
8. ...
9. }

**ContextIml.java类**  路径 ：/frameworks/base/core/java/android/app/ContextImpl.java

          说明：该Context类的实现类为ContextIml，该类实现了Context类的功能。请注意，该函数的大部分功能都是直接调用

      其属性mPackageInfo去完成，这点我们后面会讲到。

         源代码(部分)如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. /\*\*
2. \* Common implementation of Context API, which provides the base
3. \* context object for Activity and other application components.
4. \*/
5. **class** ContextImpl **extends** Context{
6. //所有Application程序公用一个mPackageInfo对象
7. /\*package\*/ ActivityThread.PackageInfo mPackageInfo;
9. @Override
10. **public** Object getSystemService(String name){
11. ...
12. **else** **if** (ACTIVITY\_SERVICE.equals(name)) {
13. **return** getActivityManager();
14. }
15. **else** **if** (INPUT\_METHOD\_SERVICE.equals(name)) {
16. **return** InputMethodManager.getInstance(**this**);
17. }
18. }
19. @Override
20. **public** **void** startActivity(Intent intent) {
21. ...
22. //开始启动一个Activity
23. mMainThread.getInstrumentation().execStartActivity(
24. getOuterContext(), mMainThread.getApplicationThread(), **null**, **null**, intent, -1);
25. }
26. }

**ContextWrapper类** 路径 ：\frameworks\base\core\java\android\content\ContextWrapper.java

        说明： 正如其名称一样，该类只是对Context类的一种包装，该类的构造函数包含了一个真正的Context引用，即ContextIml

       对象。    源代码(部分)如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. **public** **class** ContextWrapper **extends** Context {
2. Context mBase;  //该属性指向一个ContextIml实例，一般在创建Application、Service、Activity时赋值
4. //创建Application、Service、Activity，会调用该方法给mBase属性赋值
5. **protected** **void** attachBaseContext(Context base) {
6. **if** (mBase != **null**) {
7. **throw** **new** IllegalStateException("Base context already set");
8. }
9. mBase = base;
10. }
11. @Override
12. **public** **void** startActivity(Intent intent) {
13. mBase.startActivity(intent);  //调用mBase实例方法
14. }
15. }

**ContextThemeWrapper类** 路径：/frameworks/base/core/java/android/view/ContextThemeWrapper.java

      说明：该类内部包含了主题(Theme)相关的接口，即android:theme属性指定的。只有Activity需要主题，Service不需要主题，

   所以Service直接继承于ContextWrapper类。

      源代码(部分)如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. **public** **class** ContextThemeWrapper **extends** ContextWrapper {
2. //该属性指向一个ContextIml实例，一般在创建Application、Service、Activity时赋值
4. **private** Context mBase;
5. //mBase赋值方式同样有一下两种
6. **public** ContextThemeWrapper(Context base, **int** themeres) {
7. **super**(base);
8. mBase = base;
9. mThemeResource = themeres;
10. }
12. @Override
13. **protected** **void** attachBaseContext(Context newBase) {
14. **super**.attachBaseContext(newBase);
15. mBase = newBase;
16. }
17. }

     Activity类 、Service类 、Application类本质上都是Context子类， 更多信息大家可以自行参考源代码进行理解。

**二、 什么时候创建Context实例**

      熟悉了Context的继承关系后，我们接下来分析应用程序在什么情况需要创建Context对象的？应用程序创建Context实例的

 情况有如下几种情况：

**1、创建Application 对象时， 而且整个App共一个Application对象**

**2、创建Service对象时**

**3、创建Activity对象时**

    因此应用程序App共有的Context数目公式为：

**总Context实例个数 = Service个数 + Activity个数 + 1（Application对应的Context实例）**

**具体创建Context的时机**

     1、创建Application对象的时机

       每个应用程序在第一次启动时，都会首先创建Application对象。如果对应用程序启动一个Activity(startActivity)流程比较

清楚的话，创建Application的时机在创建handleBindApplication()方法中，该函数位于 ActivityThread.java类中 ，如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. //创建Application时同时创建的ContextIml实例
2. **private** **final** **void** handleBindApplication(AppBindData data){
3. ...
4. ///创建Application对象
5. Application app = data.info.makeApplication(data.restrictedBackupMode, **null**);
6. ...
7. }
9. **public** Application makeApplication(**boolean** forceDefaultAppClass, Instrumentation instrumentation) {
10. ...
11. **try** {
12. java.lang.ClassLoader cl = getClassLoader();
13. ContextImpl appContext = **new** ContextImpl();    //创建一个ContextImpl对象实例
14. appContext.init(**this**, **null**, mActivityThread);  //初始化该ContextIml实例的相关属性
15. ///新建一个Application对象
16. app = mActivityThread.mInstrumentation.newApplication(
17. cl, appClass, appContext);
18. appContext.setOuterContext(app);  //将该Application实例传递给该ContextImpl实例
19. }
20. ...
21. }

    2、创建Activity对象的时机

       通过startActivity()或startActivityForResult()请求启动一个Activity时，如果系统检测需要新建一个Activity对象时，就会

  回调handleLaunchActivity()方法，该方法继而调用performLaunchActivity()方法，去创建一个Activity实例，并且回调

 onCreate()，onStart()方法等， 函数都位于 ActivityThread.java类 ，如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. //创建一个Activity实例时同时创建ContextIml实例
2. **private** **final** **void** handleLaunchActivity(ActivityRecord r, Intent customIntent) {
3. ...
4. Activity a = performLaunchActivity(r, customIntent);  //启动一个Activity
5. }
6. **private** **final** Activity performLaunchActivity(ActivityRecord r, Intent customIntent) {
7. ...
8. Activity activity = **null**;
9. **try** {
10. //创建一个Activity对象实例
11. java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
12. activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);
13. }
14. **if** (activity != **null**) {
15. ContextImpl appContext = **new** ContextImpl();      //创建一个Activity实例
16. appContext.init(r.packageInfo, r.token, **this**);   //初始化该ContextIml实例的相关属性
17. appContext.setOuterContext(activity);            //将该Activity信息传递给该ContextImpl实例
18. ...
19. }
20. ...
21. }

   3、创建Service对象的时机

       通过startService或者bindService时，如果系统检测到需要新创建一个Service实例，就会回调handleCreateService()方法，

 完成相关数据操作。handleCreateService()函数位于 ActivityThread.java类，如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. //创建一个Service实例时同时创建ContextIml实例
2. **private** **final** **void** handleCreateService(CreateServiceData data){
3. ...
4. //创建一个Service实例
5. Service service = **null**;
6. **try** {
7. java.lang.ClassLoader cl = packageInfo.getClassLoader();
8. service = (Service) cl.loadClass(data.info.name).newInstance();
9. } **catch** (Exception e) {
10. }
11. ...
12. ContextImpl context = **new** ContextImpl(); //创建一个ContextImpl对象实例
13. context.init(packageInfo, **null**, **this**);   //初始化该ContextIml实例的相关属性
14. //获得我们之前创建的Application对象信息
15. Application app = packageInfo.makeApplication(**false**, mInstrumentation);
16. //将该Service信息传递给该ContextImpl实例
17. context.setOuterContext(service);
18. ...
19. }

    另外，需要强调一点的是，通过对ContextImp的分析可知，其方法的大多数操作都是直接调用其属性mPackageInfo(该属性类

型为PackageInfo)的相关方法而来。这说明**ContextImp是一种轻量级类，而PackageInfo才是真正重量级的类**。而一个App里的

所有ContextIml实例，都对应同一个packageInfo对象。

     最后给大家分析利用Context获取SharedPreferences类的使用方法，SharedPreferences类想必大家都使用过，其一般获取方

法就是通过调用getSharedPreferences()方法去根据相关信息获取SharedPreferences对象。具体流程如下：

    1 、调用  getSharedPreferences()获取对应的的文件，该函数实现功能如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. //Context类静态数据集合，以键值对保存了所有读取该xml文件后所形成的数据集合
2. **private** **static** **final** HashMap<File, SharedPreferencesImpl> sSharedPrefs =
3. **new** HashMap<File, SharedPreferencesImpl>();
5. @Override
6. **public** SharedPreferences getSharedPreferences(String name, **int** mode){
7. //其所对应的SharedPreferencesImpl对象 ，该对象已一个HashMap集合保存了我们对该文件序列化结果
8. SharedPreferencesImpl sp;
9. File f = getSharedPrefsFile(name);  //该包下是否存在对应的文件，不存在就新建一个
10. **synchronized** (sSharedPrefs) {       //是否已经读取过该文件，是就直接返回该SharedPreferences对象
11. sp = sSharedPrefs.get(f);
12. **if** (sp != **null** && !sp.hasFileChanged()) {
13. //Log.i(TAG, "Returning existing prefs " + name + ": " + sp);
14. **return** sp;
15. }
16. }
17. //以下为序列化该xml文件，同时将数据写到map集合中
18. Map map = **null**;
19. **if** (f.exists() && f.canRead()) {
20. **try** {
21. str = **new** FileInputStream(f);
22. map = XmlUtils.readMapXml(str);
23. str.close();
24. }
25. ...
26. }
28. **synchronized** (sSharedPrefs) {
29. **if** (sp != **null**) {
30. //Log.i(TAG, "Updating existing prefs " + name + " " + sp + ": " + map);
31. sp.replace(map);   //更新数据集合
32. } **else** {
33. sp = sSharedPrefs.get(f);
34. **if** (sp == **null**) {
35. //新建一个SharedPreferencesImpl对象，并且设置其相关属性
36. sp = **new** SharedPreferencesImpl(f, mode, map);
37. sSharedPrefs.put(f, sp);
38. }
39. }
40. **return** sp;
41. }
42. }

   2、 SharedPreferences 不过是个接口，它定义了一些操作xml文件的方法，其真正实现类为SharedPreferencesImpl ，该类是

    ContextIml的内部类，该类如下：

**[java]** [view plain](http://blog.csdn.net/qinjuning/article/details/7310620" \o "view plain)[copy](http://blog.csdn.net/qinjuning/article/details/7310620" \o "copy)[print](http://blog.csdn.net/qinjuning/article/details/7310620" \o "print)[?](http://blog.csdn.net/qinjuning/article/details/7310620" \o "?)

1. //soga，这种形式我们在分析Context ContextIml时接触过
2. //SharedPreferences只是一种接口，其真正实现类是SharedPreferencesImpl类
3. **private** **static** **final** **class** SharedPreferencesImpl **implements** SharedPreferences{
4. **private** Map mMap;  //保存了该文件序列化结果后的操作， 键值对形式
6. //通过key值获取对应的value值
7. **public** String getString(String key, String defValue) {
8. **synchronized** (**this**) {
9. String v = (String)mMap.get(key);
10. **return** v != **null** ? v : defValue;
11. }
12. }
13. ...
14. //获得该SharedPreferencesImpl对象对应的Edito类，对数据进行操作
15. **public** **final** **class** EditorImpl **implements** Editor {
16. **private** **final** Map<String, Object> mModified = Maps.newHashMap(); //保存了对键值变化的集合
17. }
18. }

       基本上获取SharedPreferences 对象就是这么来的，关于Context里的更多方法请大家参照源代码认真学习吧。

## Android Context的几点看法

  (2012-07-26 18:08:44)



[转载▼](javascript:;)



|  |  |
| --- | --- |
| 标签： [it](http://search.sina.com.cn/?c=blog&q=it&by=tag" \t "_blank) | 分类： [Android学习笔记](http://blog.sina.com.cn/s/articlelist_2160042087_2_1.html" \t "_blank) |

1.首先什么是context?

文档语焉不详，说是resource什么的，反正是没看懂，实际上可以认为它是一个指向parent对象的指针，受到那个parent对象的控制。

2.为什么需要context?

试想一下这个语句：

Button myButton =new Button(this);

这个this就是context，前面说了它指向了parent对象，那指向了哪里？为什么需要这么做？原因是Button为了能完成自己的使命（响应各种操作）就得让Android系统知道自己是属于哪个Activity的，这个信息是必须的，因为只有这样Android系统才会对其进行管理，比如响应onClick()事件，否则系统连Button是属于哪个Activity的都不知道，怎么响应呢？如果Button的context和Activity的context是一样的，那么它们都是可见的，比如上面那个button跟Activity一样是可见的，它们的context指向另一个不可见的对象，也就是它们受那个不可见的对象控制，我理解为系统。

3.Application context和Activity context。

这是两种不同的context，也是最常见的两种。第一种中context的生命周期与Application的生命周期相关的，context随着Application的销毁而销毁，第二种中的context跟Activity的生命周期是相关的，但是对一个Application来说，Activity可以销毁几次，那么属于Activity的context就会销毁多次。至于用哪种context，得看应用场景，个人感觉用Activity的context好一点，不过也有的时候必须使用Application的context。

举例来说：SQLite建立时是需要context的，那么我们怎么传呢？

一、建立SQLiteOpenHelper 的子类

public class mySQLiteHelper extends SQLiteOpenHelper {

..........

public DBOpenHelper(Context context, String name, CursorFactory factory,

int version) {

super(context, name, factory, version);

// TODO Auto-generated constructor stub

}

.......

}

二、我们用另一个类封装一下这个类，完善数据查询、插入、删除、更新等操作，具体方法不表。

public class DBManager {

//定义上面那个类对象；

private mySQLiteHelper dbHelper;

//注意这里得定义一个context，你想直接得到是得不到的，需要调用这个类的对象自己传一个context过来，这个context再将得到的context传到mySQLiteHelper 那个类去。

private Context context;

//构造方法

public DBManager (Context context) {

this.context = context;

dbHelper = new DBOpenHelper(context,DB\_NAME,null,VERSION);

db = dbHelper.getWritableDatabase();

}

}

 三、最后，我们在一个Activity中进行调用

public class myActivity extends Activity {

private DBManager db;

 public void onCreate(Bundle savedInstanceState) {

........

     //dbmgr = new DataManager(getApplicationContext()); 第一种

       dbmgr = new DataManager(this);  //第二种

//两种方法都可以，第一种是Application的context，第二种是Activity的context

.........

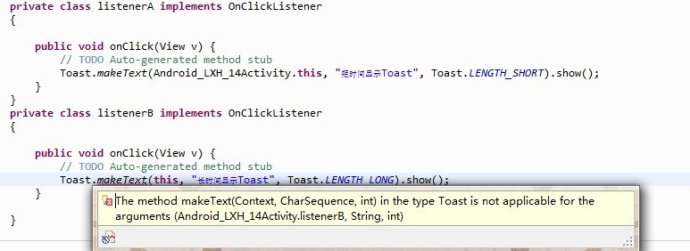
}

}

在android中context可以作很多操作，但是最主要的功能是加载和访问资源。在android中有两种context，一种是 application context，一种是activity context，通常我们在各种类和方法间传递的是activity context。  
比如一个activity的onCreate：  
protected void onCreate(Bundle state) {  
        super.onCreate(state);  
  
        TextView label = new TextView(this); //传递context给view control  
        label.setText("Leaks are bad");  
  
        setContentView(label);  
}  
把activity context传递给view，意味着view拥有一个指向activity的引用，进而引用activity占有的资源：view hierachy, resource等。  
这样如果context发生内存泄露的话，就会泄露很多内存。  
这里泄露的意思是gc没有办法回收activity的内存。  
  
Leaking an entire activity是很容易的一件事。  
  
当屏幕旋转的时候，系统会销毁当前的activity，保存状态信息，再创建一个新的。  
  
比如我们写了一个应用程序，它需要加载一个很大的图片，我们不希望每次旋转屏 幕的时候都销毁这个图片，重新加载。实现这个要求的简单想法就是定义一个静态的Drawable，这样Activity 类创建销毁它始终保存在内存中。  
实现类似：  
public class myactivity extends Activity {  
        private static Drawable sBackground;  
        protected void onCreate(Bundle state) {  
                super.onCreate(state);  
  
                TextView label = new TextView(this);  
                label.setText("Leaks are bad");  
  
                if (sBackground == null) {  
                        sBackground = getDrawable(R.drawable.large\_bitmap);  
                }  
        label.setBackgroundDrawable(sBackground);//drawable attached to a view  
  
        setContentView(label);  
        }  
}  
这段程序看起来很简单，但是却问题很大。当屏幕旋转的时候会有leak（即gc没法销毁activity）。  
我们刚才说过，屏幕旋转的时候系统会销毁当前的activity。但是当drawable和view关联后，drawable保存了view的 reference，即sBackground保存了label的引用，而label保存了activity的引用。既然drawable不能销毁，它所 引用和间接引用的都不能销毁，这样系统就没有办法销毁当前的activity，于是造成了内存泄露。gc对这种类型的内存泄露是无能为力的。  
  
避免这种内存泄露的方法是避免activity中的任何对象的生命周期长过activity，避免由于对象对 activity的引用导致activity不能正常被销毁。我们可以使用application context。application context伴随application的一生，与activity的生命周期无关。application context可以通过Context.getApplicationContext或者Activity.getApplication方法获取。  
  
避免context相关的内存泄露，记住以下几点：  
1. 不要让生命周期长的对象引用activity context，即保证引用activity的对象要与activity本身生命周期是一样的  
2. 对于生命周期长的对象，可以使用application context  
3. 避免非静态的内部类，尽量使用静态类，避免生命周期问题，注意内部类对外部对象引用导致的生命周期变化

Context字面意思是上下文，位于framework package的android.content.Context中，其实该类为LONG型，类似Win32中的Handle句柄。很多方法需要通过 Context才能识别调用者的实例：比如说Toast的第一个参数就是Context，一般在Activity中我们直接用this代替，代表调用者的实例为Activity，而到了一个button的onClick(View view)等方法时，我们用this时就会报错，所以我们可能使用ActivityName.this来解决，主要原因是因为实现Context的类主要有Android特有的几个模型，Activity以及Service。

如下图所示：



Context提供了关于应用环境全局信息的接口。它是一个抽象类，它的执行被Android系统所提供。它允许获取以应用为特征的资源和类型。同时启动应用级的操作，如启动Activity，broadcasting和接收intents。

[Android中Context简介](http://blog.csdn.net/zhangqijie001/article/details/5891682)

Context字面意思是上下文，位于framework package的android.content.Context中，其实该类为LONG型，类似Win32中的Handle句柄。很多方法需要通过 Context才能识别调用者的实例：比如说Toast的第一个参数就是Context，一般在Activity中我们直接用this代替，代表调用者的实例为Activity，而到了一个button的onClick(View view)等方法时，我们用this时就会报错，所以我们可能使用ActivityName.this来解决，主要原因是因为实现Context的类主要有Android特有的几个模型，Activity以及Service。

Context提供了关于应用环境全局信息的接口。它是一个抽象类，它的执行被Android系统所提供。它允许获取以应用为特征的资源和类型。同时启动应用级的操作，如启动Activity，broadcasting和接收intents。

下面介绍Context的一些get方法，通过这些get方法可以获取应用环境全局信息：

#### 1.public abstract Context getApplicationContext ()

Return the context of the single, global Application object of the current process.

#### 2.public abstract ApplicationInfo getApplicationInfo ()

Return the full application info for this context's package.

#### 3.public abstract ContentResolver getContentResolver ()

Return a ContentResolver instance for your application's package.

#### 4.public abstract PackageManager getPackageManager ()

Return PackageManager instance to find global package information.

#### 5.public abstract String getPackageName ()

Return the name of this application's package.

#### 6.public abstract Resources getResources ()

Return a Resources instance for your application's package.

#### 7.public abstract SharedPreferences getSharedPreferences (String name, int mode)

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.

#### 8.public final String getString (int resId)

Return a localized string from the application's package's default string table.

#### 9.public abstract Object getSystemService (String name)

Return the handle to a system-level service by name. The class of the returned object varies by the requested name. Currently available names are:

还有很多有用的方法，具体不一一列举。详情请参考文档，反正Context很有用。