# ActivityThread

S

ActivityThread是Android Framework中一个非常重要的类，它代表一个应用进程的主线程（对于应用进程来说，ActivityThread的main函数确实是由该进程的主线程执行），其职责就是调度及执行在该线程中运行的四大组件。

**注意**应用进程指那些运行APK的进程，它们由Zyote 派生（fork）而来，上面运行了dalvik虚拟机。与应用进程相对的就是系统进程（包括Zygote和SystemServer）。

另外，读者须将“应用进程和系统进程”与“应用APK和系统APK”的概念区分开来。APK的判别依赖其文件所在位置（如果apk文件在/data/app目录下，则为应用APK）。

## 承上

AtMS.startProcessLocked函数进一步操作

反射方式启动的

1. **private** **final** **void** startProcessLocked(ProcessRecord app,
2. String hostingType, String hostingNameStr) {
3. **try** {
4. **int** uid = app.info.uid;
5. **int**[] gids = **null**;
6. **try** {
7. gids = mContext.getPackageManager().getPackageGids(
8. app.info.packageName);
9. } **catch** (PackageManager.NameNotFoundException e) {
10. }
11. **int** debugFlags = 0;
12. **int** pid = Process.start("android.app.ActivityThread",
13. mSimpleProcessManagement ? app.processName : **null**, uid, uid,
14. gids, debugFlags, **null**);
15. } **catch** (RuntimeException e) {
16. }
17. }

这里主要是调用**Process.start**接口来创建一个新的进程，新的进程会导入android.app.ActivityThread类，并且执行它的main函数，这就是为什么我们前面说每一个应用程序都有一个ActivityThread实例来对应的原因。

## systemMain

**systemMain函数将为SystemServer进程搭建一个和应用进程一样的Android运行环境。**这句话涉及两个概念。

·  进程：来源于操作系统，是在OS中看到的运行体。我们编写的代码一定要运行在一个进程中。

·  Android运行环境：Android努力构筑了一个自己的运行环境。在这个环境中，进程的概念被模糊化了。组件的运行及它们之间的交互均在该环境中实现。

Android运行环境是构建在进程之上的。有Android开发经验的读者可能会发现，在应用程序中，一般只和Android运行环境交互。基于同样的道理，SystemServer希望它内部的那些Service也通过Android运行环境交互，因此也需为它创建一个运行环境。由于SystemServer的特殊性，此处调用了systemMain函数，而普通的应用进程将在主线程中调用ActivityThread的main函数来创建Android运行环境。

另外，ActivityThread虽然本意是代表进程的主线程，但是作为一个Java类，它的实例到底由什么线程创建，恐怕不是ActivityThread自己能做主的，所以在SystemServer中可以发现，ActivityThread对象由其他线程创建，而在应用进程中，ActivityThread将由主线程来创建。

[-->ActivityThread.java::systemMain]

public static final ActivityThread systemMain() {

   HardwareRenderer.disable(true);//禁止硬件渲染加速

   //创建一个ActivityThread对象，其构造函数非常简单

  ActivityThread thread = new ActivityThread();

  thread.attach(true);//调用它的attach函数，注意传递的参数为true

   returnthread;

 }

在分析ActivityThread的attach函数之前，先提一个问题供读者思考：前面所说的ActivityThread代表应用进程（其上运行了APK）的主线程，而SystemServer并非一个应用进程，那么为什么此处也需要ActivityThread呢？

·  还记得在PackageManagerService分析中提到的framework-res.apk吗？这个APK除了包含资源文件外，还包含一些Activity（如关机对话框），这些Activity实际上运行在SystemServer进程中[②]。从这个角度看，SystemServer是一个特殊的应用进程。

·  另外，通过ActivityThread可以把Android系统提供的组件之间的交互机制和交互接口（如利用Context提供的API）也拓展到SystemServer中使用。

**提示**解答这个问题，对于理解SystemServer中各服务的交互方式是尤其重要的。

### attach函数分析

fa

[-->ActivityThread.java::attach]

private void attach(boolean system) {

    sThreadLocal.set(this);

    mSystemThread= system;//判断是否为系统进程

    if(!system) {

        ......//应用进程的处理流程

     } else {//系统进程的处理流程，该情况只在SystemServer中处理

       //设置DDMS时看到的systemserver进程名为system\_process

       android.ddm.DdmHandleAppName.setAppName("system\_process");

       try {

            //ActivityThread的几员大将出场，见后文的分析

            mInstrumentation = new Instrumentation();

            ContextImpl context = new ContextImpl();

            //初始化context，注意第一个参数值为getSystemContext

            context.init(getSystemContext().mPackageInfo, null, this);

            Application app = //利用Instrumentation创建一个Application对象

                    Instrumentation.newApplication(Application.class,context);

             //一个进程支持多个Application，mAllApplications用于保存该进程中

            //的Application对象

            mAllApplications.add(app);

             mInitialApplication = app;//设置mInitialApplication

            app.onCreate();//调用Application的onCreate函数

           }......//try/catch结束

      }//if(!system)判断结束

     //注册Configuration变化的回调通知

     ViewRootImpl.addConfigCallback(newComponentCallbacks2() {

          publicvoid onConfigurationChanged(Configuration newConfig) {

            ......//当系统配置发生变化（如语言切换等）时，需要调用该回调

          }

           public void onLowMemory() {}

           public void onTrimMemory(int level) {}

        });

 }

attach函数中出现了几个重要成员，其类型分别是Instrumentation类、Application类及Context类，它们的作用如下

Instrumentaion是一个工具类。当它被启用时，系统先创建它，再通过它来创建其他组件。另外，系统和组件之间的交互也将通过Instrumentation来传递，这样，Instrumentation就能监测系统和这些组件的交互情况了。在实际使用中，我们可以创建Instrumentation的派生类来进行相应的处理。读者可查询Android中Junit的使用来了解Intrstrumentation的作用

Application类保存了一个全局的application状态。Application由AndroidManifest.xml中的<application>标签声明。在实际使用时需定义Application的派生类。此处的Application是Android中的一个概念，可理解为一种容器，它内部包含四大组件。另外，一个进程可以运行多个Application。

Context是一个接口，通过它可以获取并操作Application对应的资源、类，甚至包含于Application中的四大组件。

Context是一个抽象类，而由AMS创建的将是它的子类ContextImpl。如前所述，Context提供了Application的上下文信息，这些信息是如何传递给Context的呢？此问题包括两个方面：

·  Context本身是什么？

·  Context背后所包含的上下文信息又是什么？

## getSystemContext

[-->ActivityThread.java::getSystemContext]

public ContextImpl getSystemContext() {

  synchronized(this) {

   if(mSystemContext == null) {//单例模式

       ContextImplcontext =  ContextImpl.createSystemContext(this);

       //LoadedApk是2.3引入的一个新类，代表一个加载到系统中的APK

       LoadedApkinfo = new LoadedApk(this, "android", context, null,

                       CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO);

       //初始化该ContextImpl对象

      context.init(info, null, this);

      //初始化资源信息

      context.getResources().updateConfiguration(

                        getConfiguration(),getDisplayMetricsLocked(

                       CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO, false));

       mSystemContext = context;//保存这个特殊的ContextImpl对象

      }

   }

    returnmSystemContext;

}

以上代码无非是先创建一个ContextImpl，然后再将其初始化（调用init函数）。为什么函数名是getSystemContext呢？因为在初始化ContextImp时使用了一个LoadedApk对象。如注释中所说，LoadedApk是Android 2.3引入的一个类，该类用于保存一些和APK相关的信息（如资源文件位置、JNI库位置等）。在getSystemContext函数中初始化ContextImpl的这个LoadedApk所代表的package名为“android”，其实就是framework-res.apk，由于该APK仅供SystemServer使用，所以此处叫getSystemContext。

上面这些类的关系比较复杂，可通过图6-2展示它们之间的关系。

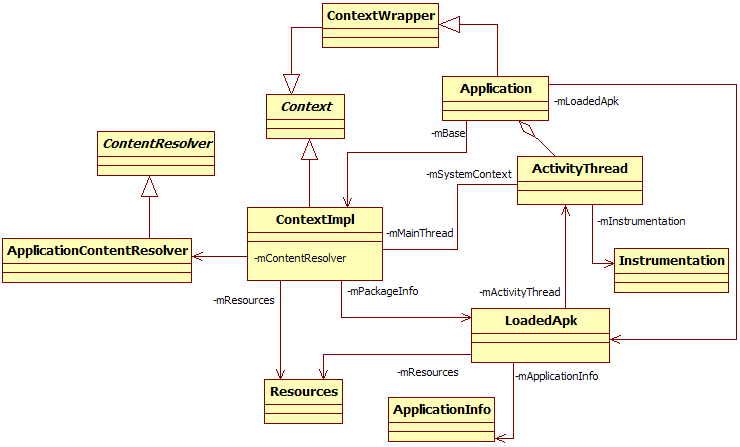


图6-2  ContextImpl和它的“兄弟”们

由图6-2可知：

·  先来看派生关系， ApplicationContentResolver从ConentResolver派生，它主要用于和ContentProvider打交道。ContextImpl和ContextWrapper均从Context继承，而Application则从ContextWrapper派生。

·  从社会关系角度看，ContextImpl交际面最广。它通过mResources指向Resources，mPackageInfo指向LoadedApk，mMainThread指向ActivityThread，mContentResolver指向ApplicationContentResolver。

·  ActivityThread代表主线程，它通过mInstrumentation指向Instrumentation。另外，它还保存多个Application对象。

**注意**在函数中有些成员变量的类型为基类类型，而在图6-2中直接指向了实际类型。

### Context家族图谱

getSystemContext函数分析

该函数在上一节已经见过了。调用该函数后，将得到一个代表系统进程的Context对象。到底什么是Context？先来看如图6-3所示的Context家族图谱。

**注意**该族谱成员并不完全。另外，Activity、Service和Application所实现的接口也未画出。

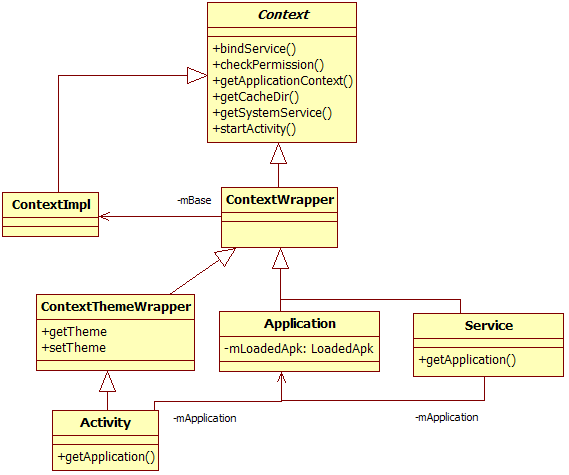


图6-3  Context家族图谱

由图6-3可知：

·  ContextWrapper比较有意思，其在SDK中的说明为“Proxying implementation ofContext that simply delegates all of its calls to another Context. Can besubclassed to modify behavior without changing the original Context.”大概意思是：ContextWrapper是一个代理类，被代理的对象是另外一个Context。在图6-3中，被代理的类其实是ContextImpl，由ContextWrapper通过mBase成员变量指定。读者可查看ContextWrapper.java，其内部函数功能的实现最终都由mBase完成。这样设计的目的是想把ContextImpl隐藏起来。

·  Application从ContextWrapper派生，并实现了ComponentCallbacks2接口。Application中有一个LoadedApk类型的成员变量mLoadedApk。LoadedApk代表一个APK文件。由于一个AndroidManifest.xml文件只能声明一个Application标签，所以一个Application必然会和一个LoadedApk绑定。

·  Service从ContextWrapper派生，其中Service内部成员变量mApplication指向Application（在AndroidManifest.xml中，Service只能作为Application的子标签，所以在代码中Service必然会和一个Application绑定）。

·  ContextThemeWrapper重载了和Theme（主题）相关的两个函数。这些和界面有关，所以Activity作为Android系统中的UI容器，必然也会从ContextThemeWrapper派生。与Service一样，Activity内部也通过mApplication成员变量指向Application。

## ActivityThread的installSystemApplicationInfo函数分析

Sd

installSystemApplicationInfo函数的参数为一个ApplicationInfo对象，该对象由AMS通过Context查询PKMS中一个名为“android”的package得来（根据前面介绍的知识，目前只有framework-res.apk声明其package名为“android”）。

再来看installSystemApplicationInfo的代码，如下所示：

[-->ActivityThread.java::installSystemApplicationInfo]

public voidinstallSystemApplicationInfo(ApplicationInfo info) {

 synchronized (this) {

   //返回的ContextImpl对象即之前在AMS的main函数一节中创建的那个对象

   ContextImpl context = getSystemContext();

    //又调用init初始化该Context，是不是重复调用init了？

   context.init(new LoadedApk(this, "android", context, info,

              CompatibilityInfo.DEFAULT\_COMPATIBILITY\_INFO), null, this);

     //创建一个Profiler对象，用于性能统计

     mProfiler = new Profiler();

     }

 }

在以上代码中看到调用context.init的地方，读者可能会有疑惑，getSystemContext函数将返回mSystemContext，而此mSystemContext在AMS的main函数中已经初始化过了，此处为何再次初始化呢？

仔细查看看代码便会发现：

·  第一次执行init时，在LoadedApk构造函数中第四个表示ApplicationInfo的参数为null。

·  第二次执行init时，LoadedApk构造函数的第四个参数不为空，即该参数将真正指向一个实际的ApplicationInfo，该ApplicationInfo来源于framework-res.apk。

基于上面的信息，某些读者可能马上能想到：Context第一次执行init的目的仅仅是为了创建一个Android运行环境，而该Context并没有和实际的ApplicationInfo绑定。而第二次执行init前，先利用Context和PKMS交互得到一个实际的ApplicationInfo，然后再通过init将此Context和ApplicationInfo绑定。

是否觉得前面的疑惑已豁然而解？且慢，此处又抛出了一个更难的问题：

第一次执行init后得到的Context虽然没有绑定ApplicationInfo，不是也能使用吗？此处为何非要和一个ApplicationInfo绑定？

答案很简单，因为framework-res.apk（包括后面将介绍的SettingsProvider.apk）运行在SystemServer中。和其他所有apk一样，它的运行需要一个正确初始化的Android运行环境。

长嘘一口气，这个大难题终于弄明白了！在此即基础上，AMS下一步的工作就就顺理成章了。

由于framework-res.apk是一个APK文件，和其他APK文件一样，它应该运行在一个进程中。而AMS是专门用于进程管理和调度的，所以运行APK的进程应该在AMS中有对应的管理结构。因此AMS下一步工作就是将这个运行环境和一个进程管理结构对应起来并交由AMS统一管理。

## ActivityThread.main

这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中

1. **public** **final** **class** ActivityThread {
2. **private** **final** **void** attach(**boolean** system) {
3. mSystemThread = system;
4. **if** (!system) {
5. IActivityManager mgr = ActivityManagerNative.getDefault();
6. **try** {
7. mgr.attachApplication(mAppThread);
8. } **catch** (RemoteException ex) {
9. }
10. } **else** {
11. }
12. }
13. **public** **static** **final** **void** main(String[] args) {
14. ActivityThread thread = **new** ActivityThread();
15. thread.attach(**false**);
16. Looper.loop();
17. thread.detach();
18. }
19. }

这个函数在进程中创建一个ActivityThread实例，然后调用它的attach函数，接着就进入消息循环了，直到最后进程退出。

函数attach最终调用了AtMS的远程接口ActivityManagerProxy的attachApplication函数，传入的参数是mAppThread，这是一个ApplicationThread类型的Binder对象，它的作用是用来进行进程间通信的

## ActivityThread（）

# 主题Theme

主要关注themes.xml,themes\_device\_defaults.xml两个文件。

themes.xml定义了android低版本的theme和Holo theme，themes\_device\_defaults.xml定义了DeviceDefault主题（继承自Holo主题），实际上就是在Holo主题上定制主题（For厂商）。

系统如何去选择默认的主题呢？

/\*\*

\* Returns the most appropriate default theme for the specified target SDK version.

\* <ul>

\* <li>Below API 11: Gingerbread

\* <li>APIs 12 thru 14: Holo

\* <li>APIs 15 thru 23: Device default dark

\* <li>APIs 24 and above: Device default light with dark action bar

\* </ul>

\*

\* @param curTheme The current theme, or 0 if not specified.

\* @param targetSdkVersion The target SDK version.

\* @return A theme resource identifier

\* @hide

\*/

public static int selectDefaultTheme(int curTheme, int targetSdkVersion) {

return selectSystemTheme(curTheme, targetSdkVersion,

com.android.internal.R.style.Theme,

com.android.internal.R.style.Theme\_Holo,

com.android.internal.R.style.Theme\_DeviceDefault,

com.android.internal.R.style.Theme\_DeviceDefault\_Light\_DarkActionBar);

}

/\*\* @hide \*/

public static int selectSystemTheme(int curTheme, int targetSdkVersion, int orig, int holo,

int dark, int deviceDefault) {

if (curTheme != 0) {

return curTheme;

}

if (targetSdkVersion < Build.VERSION\_CODES.HONEYCOMB) {

return orig;

}

if (targetSdkVersion < Build.VERSION\_CODES.ICE\_CREAM\_SANDWICH) {

return holo;

}

if (targetSdkVersion < Build.VERSION\_CODES.N) {

return dark;

}

return deviceDefault;

}

# AlertActivity

在开发过程中，如果在一个Recevier里面收到一条广播后需要进行弹出Dialog的处理，那么这时就需要使用到AlertActivity了，就是用Activity来模拟出Dialog，因为在Android中，Recevier里面是不能弹出Dialog的。

<http://007hiccup.blog.163.com/blog/static/444976192014721348761/>

**mAdvanceKey** = KeyEvent.***KEYCODE\_DPAD\_RIGHT***;  
**mRetreatKey** = KeyEvent.***KEYCODE\_DPAD\_LEFT***;

# APP

## DatePickerDialog

**final** Context themeContext = getContext();  
**final** LayoutInflater inflater = LayoutInflater.*from*(themeContext);  
**final** View view = inflater.inflate(R.layout.date\_picker\_dialog, **null**);  
setView(view);  
  
setButton(***BUTTON\_POSITIVE***, themeContext.getString(R.string.ok), **this**);  
setButton(***BUTTON\_NEGATIVE***, themeContext.getString(R.string.cancel), **this**);

android-7.1.1\_r1/frameworks/base/core/res/res/layout/date\_picker\_dialog.xml

<**DatePicker xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/datePicker"  
 android:layout\_gravity="center\_horizontal"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:spinnersShown="true"  
 android:calendarViewShown="false"  
 android:dialogMode="true"** />

## TimePickerDialog

为何最终进入的是android.widget.TimePicker.而不是**com.android.contacts.datepicker.DatePicker呢**

## Notification