深入理解ActivityManagerService

本章主要内容：

·  详细分析ActivityManagerService

本章所涉及的源代码文件名及位置：

·  SystemServer.java

frameworks/base/services/java/com/android/server/SystemServer.java

·  ActivityManagerService.java

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

·  ContextImpl.java

frameworks/base/core/java/android/app/ContextImpl.java

·  ActivityThread.java

frameworks/base/core/java/android/app/ActivityThread.java

·  ActivityStack.java

frameworks/base/services/java/com/android/server/am/ActivityStack.java

·  Am.java

frameworks/base/cmds/am/src/com/android/commands/am/Am.java

·  ProcessRecord.java

frameworks/base/services/java/com/android/server/am/ProcessRecord.java

·  ProcessList.java

frameworks/base/services/java/com/android/server/am/ProcessList.java

·  RuntimeInit.java

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

6.1  概述

相信绝大部分读者对本书提到的ActivityManagerService（以后简称AMS）都有所耳闻。AMS是Android中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块相类似，因此它在Android中非常重要。

AMS是本书碰到的第一块难啃的骨头[①]，涉及的知识点较多。为了帮助读者更好地理解AMS，本章将带领读者按五条不同的线来分析它。

·  第一条线：同其他服务一样，将分析SystemServer中AMS的调用轨迹。

·  第二条线：以am命令启动一个Activity为例，分析应用进程的创建、Activity的启动，以及它们和AMS之间的交互等知识。

·  第三条线和第四条线：分别以Broadcast和Service为例，分析AMS中Broadcast和Service的相关处理流程。

·  第五条线：以一个Crash的应用进程为出发点，分析AMS如何打理该应用进程的身后事。

除了这五条线外，还将统一分析在这五条线中频繁出现的与AMS中应用进程的调度、内存管理等相关的知识。

**提示**ContentProvider将放到下一章分析，不过本章将涉及和ContentProvider有关的知识点。

先来看AMS的家族图谱，如图6-1所示。



图6-1  AMS家族图谱

由图6-1可知：

·  AMS由ActivityManagerNative（以后简称AMN）类派生，并实现Watchdog.Monitor和BatteryStatsImpl.BatteryCallback接口。而AMN由Binder派生，实现了IActivityManager接口。

·  客户端使用ActivityManager类。由于AMS是系统核心服务，很多API不能开放供客户端使用，所以设计者没有让ActivityManager直接加入AMS家族。在ActivityManager类内部通过调用AMN的getDefault函数得到一个ActivityManagerProxy对象，通过它可与AMS通信。

AMS的简单介绍就到此为止，下面分析AMS。相信不少读者已经磨拳擦掌，跃跃欲试了。

**提示**读者们最好在桌上放一杯清茶，以保持AMS分析旅途中头脑清醒。

6.2  初识ActivityManagerService

AMS由SystemServer的ServerThread线程创建，提取它的调用轨迹，代码如下：

[-->SystemServer.java::ServerThread的run函数]

//①调用main函数，得到一个Context对象

context =ActivityManagerService.main(factoryTest);

//②setSystemProcess：这样SystemServer进程可加到AMS中，并被它管理

ActivityManagerService.setSystemProcess();

//③installSystemProviders：将SettingsProvider放到SystemServer进程中来运行

ActivityManagerService.installSystemProviders();

//④在内部保存WindowManagerService（以后简称WMS）

ActivityManagerService.self().setWindowManager(wm);

//⑤和WMS交互，弹出“启动进度“对话框

ActivityManagerNative.getDefault().showBootMessage(

             context.getResources().getText(

               //该字符串中文为：“正在启动应用程序”

               com.android.internal.R.string.android\_upgrading\_starting\_apps),

              false);

//⑥AMS是系统的核心，只有它准备好后，才能调用其他服务的systemReady

//注意，有少量服务在AMS systemReady之前就绪，它们不影响此处的分析

ActivityManagerService.self().systemReady(newRunnable() {

    publicvoid run() {

   startSystemUi(contextF);//启动systemUi。如此，状态栏就准备好了

    if(batteryF != null) batteryF.systemReady();

    if(networkManagementF != null) networkManagementF.systemReady();

    ......

    Watchdog.getInstance().start();//启动Watchdog

    ......//调用其他服务的systemReady函数

}

在以上代码中，一共列出了6个重要调用及这些调用的简单说明，本节将分析除与WindowManagerService（以后简称WMS）交互的4、5外的其余四项调用。

先来分析1处调用。

6.2.1  ActivityManagerService main分析

AMS的main函数将返回一个Context类型的对象，该对象在SystemServer中被其他服务大量使用。Context，顾名思义，代表了一种上下文环境（笔者觉得其意义和JNIEnv类似），有了这个环境，我们就可以做很多事情（例如获取该环境中的资源、Java类信息等）。那么AMS的main将返回一个怎样的上下文环境呢？来看以下代码：

[-->ActivityManagerService.java::main]

 publicstatic final Context main(int factoryTest) {

    AThreadthr = new AThread();//①创建一个AThread线程对象

    thr.start();

    ......//等待thr创建成功

    ActivityManagerServicem = thr.mService;

    mSelf =m;

    //②调用ActivityThread的systemMain函数

    ActivityThreadat = ActivityThread.systemMain();

    mSystemThread= at;

    //③得到一个Context对象，注意调用的函数名为getSystemContext，何为System Context

    Contextcontext = at.getSystemContext();

    context.setTheme(android.R.style.Theme\_Holo);

    m.mContext= context;

    m.mFactoryTest= factoryTest;

    //ActivtyStack是AMS中用来管理Activity的启动和调度的核心类，以后再分析它

    m.mMainStack = new ActivityStack(m, context,true);

    //调用BSS的publish函数，我们在第5章的BSS知识中介绍过了

    m.mBatteryStatsService.publish(context);

    //另外一个service：UsageStatsService。后续再分析该服务

    m.mUsageStatsService.publish(context);

    synchronized (thr) {

           thr.mReady = true;

           thr.notifyAll();//通知thr线程，本线程工作完成

     }

    //④调用AMS的startRunning函数

    m.startRunning(null, null, null, null);

   returncontext;

}

在main函数中，我们又列出了4个关键函数，分别是：

·  创建AThread线程。虽然AMS的main函数由ServerThread线程调用，但是AMS自己的工作并没有放在ServerThread中去做，而是新创建了一个线程，即AThread线程。

·  ActivityThread.systemMain函数。初始化ActivityThread对象。

·  ActivityThread.getSystemContext函数。用于获取一个Context对象，从函数名上看，该Context代表了System的上下文环境。

·  AMS的startRunning函数。

注意，main函数中有一处等待（wait）及一处通知（notifyAll），原因是：

·  main函数首先需要等待AThread所在线程启动并完成一部分工作。

·  AThread完成那一部分工作后，将等待main函数完成后续的工作。

这种双线程互相等待的情况，在Android代码中比较少见，读者只需了解它们的目的即可。下边来分析以上代码中的第一个关键点。

1.  AThread分析

（1） AThread分析

AThread的代码如下：

[-->ActivityManagerService.java::AThread]

static class AThread extends Thread {//AThread从Thread类派生

   ActivityManagerServicemService;

   booleanmReady = false;

   publicAThread() {

     super("ActivityManager");//线程名就叫“ActivityManager”

   }

   publicvoid run() {

     Looper.prepare();//看来，AThread线程将支持消息循环及处理功能

     android.os.Process.setThreadPriority(//设置线程优先级

                   android.os.Process.THREAD\_PRIORITY\_FOREGROUND);

     android.os.Process.setCanSelfBackground(false);

      //创建AMS对象

     ActivityManagerService m = new ActivityManagerService();

     synchronized (this) {

           mService= m;//赋值AThread内部成员变量mService，指向AMS

          notifyAll();  //通知main函数所在线程

      }

     synchronized (this) {

        while (!mReady) {

           try{

                 wait();//等待main函数所在线程的notifyAll

               }......

           }

       }......

    Looper.loop();//进入消息循环

 }

 }

从本质上说，AThread是一个支持消息循环及处理的线程，其主要工作就是创建AMS对象，然后通知AMS的main函数。这样看来，main函数等待的就是这个AMS对象。

（2） AMS的构造函数分析

AMS的构造函数的代码如下：

[-->ActivityManagerService.java::ActivityManagerService构造]

private ActivityManagerService() {

    FiledataDir = Environment.getDataDirectory();//指向/data/目录

    FilesystemDir = new File(dataDir, "system");//指向/data/system/目录

   systemDir.mkdirs();//创建/data/system/目录

    //创建BatteryStatsService（以后简称BSS）和UsageStatsService（以后简称USS）

   //我们在分析PowerManageService时已经见过BSS了

   mBatteryStatsService = new BatteryStatsService(new File(

               systemDir, "batterystats.bin").toString());

   mBatteryStatsService.getActiveStatistics().readLocked();

    mBatteryStatsService.getActiveStatistics().writeAsyncLocked();

   mOnBattery = DEBUG\_POWER ? true

               : mBatteryStatsService.getActiveStatistics().getIsOnBattery();

   mBatteryStatsService.getActiveStatistics().setCallback(this);

    //创建USS

    mUsageStatsService= new UsageStatsService(new File(

               systemDir, "usagestats").toString());

    //获取OpenGl版本

   GL\_ES\_VERSION = SystemProperties.getInt("ro.opengles.version",

           ConfigurationInfo.GL\_ES\_VERSION\_UNDEFINED);

     //mConfiguration类型为Configuration，用于描述资源文件的配置属性，例如

     //字体、语言等。后文再讨论这方面的内容

    mConfiguration.setToDefaults();

    mConfiguration.locale = Locale.getDefault();

     //mProcessStats为ProcessStats类型，用于统计CPU、内存等信息。其内部工作原理就是

    //读取并解析/proc/stat文件的内容。该文件由内核生成，用于记录kernel及system

    //一些运行时的统计信息。读者可在Linux系统上通过man proc命令查询详细信息

    mProcessStats.init();

     //解析/data/system/packages-compat.xml文件，该文件用于存储那些需要考虑屏幕尺寸

    //的APK的一些信息。读者可参考AndroidManifest.xml中compatible-screens相关说明。

    //当APK所运行的设备不满足要求时，AMS会根据设置的参数以采用屏幕兼容的方式去运行它

    mCompatModePackages = new CompatModePackages(this, systemDir);

     Watchdog.getInstance().addMonitor(this);

     //创建一个新线程，用于定时更新系统信息（和mProcessStats交互）

    mProcessStatsThread = new Thread("ProcessStats") {...//先略去该段代码}

    mProcessStatsThread.start();

 }

AMS的构造函数比想象得要简单些，下面回顾一下它的工作：

·  创建BSS、USS、mProcessStats （ProcessStats类型）、mProcessStatsThread线程，这些都与系统运行状况统计相关。

·  创建/data/system目录，为mCompatModePackages（CompatModePackages类型）和mConfiguration（Configuration类型）等成员变量赋值。

AMS main函数的第一个关键点就分析到此，再来分析它的第二个关键点。

2.  ActivityThread.systemMain函数分析

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）。

该函数代码如下：

[-->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。

下面来看ActivityThread的attach函数。

（1） attach函数分析

[-->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（为了保证准确，这里先引用Android的官方说明）。

·  Instrumentation：Base class for implementingapplication instrumentation code. When running with instrumentation turned on,this class will be instantiated for you before any of the application code,allowing you to monitor all of the interaction the system has with the application.An Instrumentation implementation is described to the system through anAndroidManifest.xml's <instrumentation> tag.大意是：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。本书不讨论Intrstrumentation方面的内容。

·  Application：Base class for those who need tomaintain global application state. You can provide your own implementation byspecifying its name in your AndroidManifest.xml's <application> tag,which will cause that class to be instantiated for you when the process foryour application/package is created.大意是：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。

·  Context：Interface to global informationabout an application environment. This is an abstract class whoseimplementation is provided by the Android system. It allows access toapplication-specific resources and classes, as well as up-calls forapplication-level operations such as launching activities, broadcasting andreceiving intents, etc.大意是：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。

**提示**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？

下面来关注上边代码中调用的getSystemContext函数。

（2） getSystemContext函数分析

[-->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。

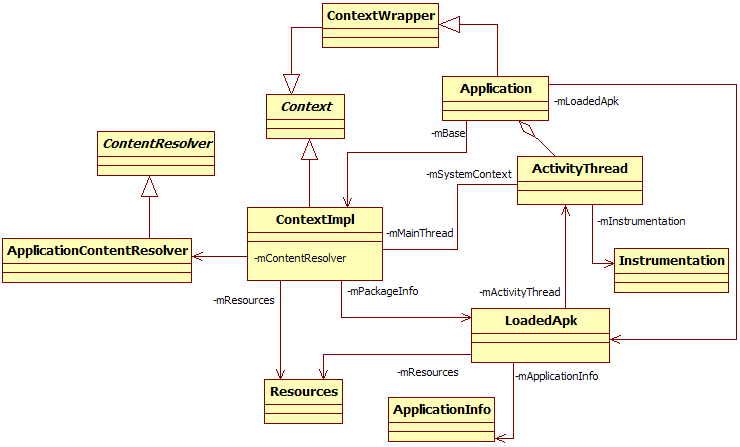


图6-2  ContextImpl和它的“兄弟”们

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。

（3） systemMain函数总结

调用systemMain函数结束后，我们得到了什么？

·  得到一个ActivityThread对象，它代表应用进程的主线程。

·  得到一个Context对象，它背后所指向的Application环境与framework-res.apk有关。

费了如此大的功夫，systemMain函数的目的到底是什么？一针见血地说：

**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。

3. ActivityThread.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。

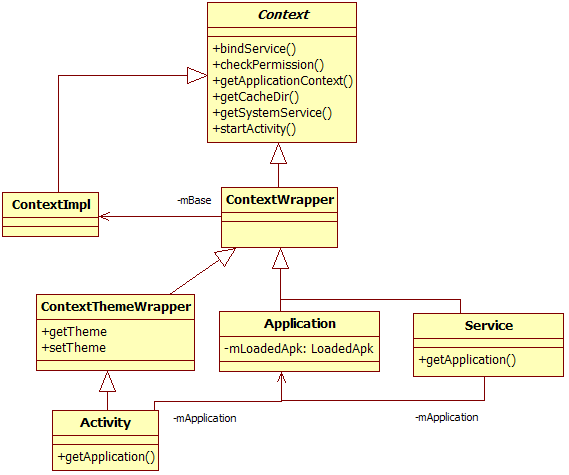


图6-3  Context家族图谱

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。

对Context的分析先到这里，再来分析第三个关键函数startRunning。

4.  AMS的startRunning函数分析

[-->ActivityManagerService.java::startRunning]

//注意调用该函数时所传递的4个参数全为null

public final void startRunning(String pkg, Stringcls, String action,

                                    String data) {

  synchronized(this) {

     if (mStartRunning) return;  //如果已经调用过该函数，则直接返回

    mStartRunning = true;

     //mTopComponent最终赋值为null

    mTopComponent = pkg != null && cls != null

                   ? new ComponentName(pkg, cls) : null;

    mTopAction = action != null ? action : Intent.ACTION\_MAIN;

    mTopData = data; //mTopData最终为null

     if(!mSystemReady) return; //此时mSystemReady为false，所以直接返回

    }

   systemReady(null);//这个函数很重要，可惜不在本次startRunning中调用

}

startRunning函数很简单，此处不赘述。

至此，ASM 的main函数所涉及的4个知识点已全部分析完。下面回顾一下AMS 的main函数的工作。

5.  ActivityManagerService的main函数总结

AMS的main函数的目的有两个：

·  首先也是最容易想到的目的是创建AMS对象。

·  另外一个目的比较隐晦，但是非常重要，那就是创建一个供SystemServer进程使用的Android运行环境。

根据目前所分析的代码，Android运行环境将包括两个成员：ActivityThread和ContextImpl（一般用它的基类Context）。

图6-4展示了在这两个类中定义的一些成员变量，通过它们可看出ActivityThread及ContextImpl的作用。

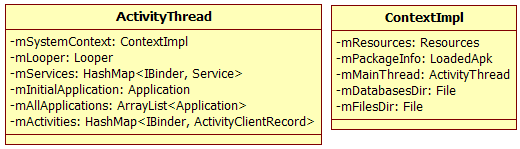


图6-4  ActivityThread和ContextImpl的部分成员变量

由图6-4可知：

·  ActivityThread中有一个mLooper成员，它代表一个消息循环。这恐怕是ActivityThread被称做“Thread”的一个直接证据。另外，mServices用于保存Service，Activities用于保存ActivityClientRecord，mAllApplications用于保存Application。关于这些变量的具体作用，以后遇到时再说。

·  对于ContextImpl，其成员变量表明它和资源、APK文件有关。

AMS的main函数先分析到此，至于其创建的Android运行环境将在下节分析中派上用场。

接下来分析AMS的第三个调用函数setSystemProcess。

6.2.2  AMS的setSystemProcess分析

AMS的setSystemProcess的代码如下：

[-->ActivityManagerService.java::setSystemProcess]

public static void setSystemProcess() {

  try {

      ActivityManagerService m = mSelf;

       //向ServiceManager注册几个服务

       ServiceManager.addService("activity",m);

       //用于打印内存信息

      ServiceManager.addService("meminfo", new MemBinder(m));

       /\*

        Android4.0新增的，用于打印应用进程使用硬件显示加速方面的信息（Applications

         Graphics Acceleration Info）。读者通过adb shell dumpsys gfxinfo看看具体的

         输出

       \*/

      ServiceManager.addService("gfxinfo", new GraphicsBinder(m));

        if(MONITOR\_CPU\_USAGE)//该值默认为true，添加cpuinfo服务

            ServiceManager.addService("cpuinfo", new CpuBinder(m));

        //向SM注册权限管理服务PermissionController

       ServiceManager.addService("permission", newPermissionController(m));

      /\*    **重要说明：**

        向PackageManagerService查询package名为"android"的ApplicationInfo。

        注意这句调用：虽然PKMS和AMS同属一个进程，但是二者交互仍然借助Context。

        其实，此处完全可以直接调用PKMS的函数。为什么要费如此周折呢

      \*/

      ApplicationInfo info = //使用AMS的mContext对象

               mSelf.mContext.getPackageManager().getApplicationInfo(

                        "android",STOCK\_PM\_FLAGS);

          //①调用ActivityThread的installSystemApplicationInfo函数

            mSystemThread.installSystemApplicationInfo(info);

           synchronized (mSelf) {

               //②此处涉及AMS对进程的管理，见下文分析

               ProcessRecord app = mSelf.newProcessRecordLocked(

                       mSystemThread.getApplicationThread(), info,

                        info.processName);//注意，最后一个参数为字符串，值为“system”

               app.persistent = true;

               app.pid = MY\_PID;

               app.maxAdj = ProcessList.SYSTEM\_ADJ;

               //③保存该ProcessRecord对象

               mSelf.mProcessNames.put(app.processName, app.info.uid, app);

               synchronized (mSelf.mPidsSelfLocked) {

                   mSelf.mPidsSelfLocked.put(app.pid, app);

               }

               //根据系统当前状态，调整进程的调度优先级和OOM\_Adj，后续将详细分析该函数

                mSelf.updateLruProcessLocked(app, true,true);

           }

        } ......//抛异常

    }

在以上代码中列出了一个重要说明和两个关键点。

·  重要说明：AMS向PKMS查询名为“android”的ApplicationInfo。此处AMS和PKMS的交互是通过Context来完成的，查看这一系列函数调用的代码，最终发现AMS将通过Binder发送请求给PKMS来完成查询功能。AMS和PKMS同属一个进程，它们完全可以不通过Context来交互。此处为何要如此大费周章呢？原因很简单，Android希望SystemServer中的服务也通过Android运行环境来交互。这更多是从设计上来考虑的，比如组件之间交互接口的统一及未来系统的可扩展性。

·  关键点一：ActivityThread的installSystemApplicationInfo函数。

·  关键点二：ProcessRecord类，它和AMS对进程的管理有关。

通过重要说明，相信读者能真正理解AMS的 main函数中第二个隐含目的的作用，故此处不再展开叙述。

现在来看第一个关键点，即ActivityThread的installSystemApplicationInfo函数。

1.  ActivityThread的installSystemApplicationInfo函数分析

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。

AMS中的进程管理结构是ProcessRecord。

2.  关于ProcessRecord和IApplicationThread的介绍

分析ProcessRecord之前，先来思考一个问题：

AMS如何与应用进程交互？例如AMS启动一个位于其他进程的Activity，由于该Activity运行在另外一进程中，因此AMS势必要和该进程进行跨进程通信。

答案自然是通过Binder进行通信。为此，Android提供了一个IApplicationThread接口，该接口定义了AMS和应用进程之间的交互函数，如图6-5所示为该接口的家族图谱。

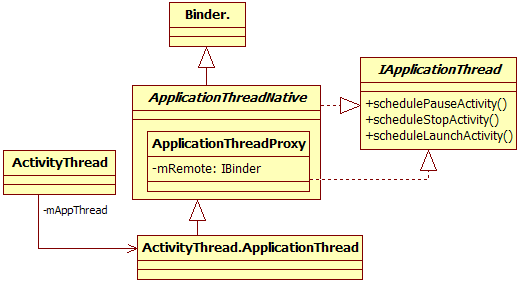


图6-5  ApplicationThread类

由图6-5可知：

·  ApplicationThreadNative实现了IApplicationThread接口。从该接口定义的函数可知，AMS通过它可以和应用进程进行交互，例如，AMS启动一个Activity的时候会调用该接口的scheduleLaunchActivity函数。

·  ActivityThread通过成员变量mAppThread指向它的内部类ApplicationThread，而ApplicationThread从ApplicationThreadNative派生。

基于以上知识，读者能快速得出下面问题的答案吗？

IApplicationThread的Binder服务端在应用进程中还是在AMS中？

**提示**如果读者知道Binder系统支持客户端监听服务端的死亡消息，那么这个问题的答案就简单了：服务端自然在应用进程中，因为AMS需要监听应用进程的死亡通知。

有了IApplicationThread接口，AMS就可以和应用进程交互了。例如，对于下面一个简单的函数：

[-->ActivityThread.java::scheduleStopActivity]

public final void scheduleStopActivity(IBindertoken, boolean showWindow,

                                     intconfigChanges) {

  queueOrSendMessage(//该函数内部将给一个Handler发送对应的消息

               showWindow ? H.STOP\_ACTIVITY\_SHOW : H.STOP\_ACTIVITY\_HIDE,

               token, 0, configChanges);

 }

当AMS想要停止（stop）一个Activity时，会调用对应进程IApplicationThread Binder客户端的scheduleStopActivity函数。该函数服务端实现的就是向ActivityThread所在线程发送一个消息。在应用进程中，ActivityThread运行在主线程中，所以这个消息最终在主线程被处理。

**提示**Activity的onStop函数也将在主线程中被调用。

IApplicationThread仅仅是AMS和另外一个进程交互的接口，除此之外，AMS还需要更多的有关该进程的信息。在AMS中，进程的信息都保存在ProcessRecord数据结构中。那么，ProcessRecord是什么呢？先来看setSystemProcess的第二个关键点，即newProcessRecordLocked函数，其代码如下：

[-->ActivityManagerService.java::newProcessRecordLocked]

final ProcessRecordnewProcessRecordLocked(IApplicationThread thread,

                ApplicationInfo info, String customProcess) {

   Stringproc = customProcess != null ? customProcess : info.processName;

  BatteryStatsImpl.Uid.Proc ps = null;

  BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();

  synchronized (stats) {

        //BSImpl将为该进程创建一个耗电量统计项

        ps =stats.getProcessStatsLocked(info.uid, proc);

   }

   //创建一个ProcessRecord对象，用于和其他进程通信的thread作为第一个参数

   returnnew ProcessRecord(ps, thread, info, proc);

 }

ProcessRecord的成员变量较多，先来看看再其构造函数中都初始化了哪些成员变量。

[-->ProcessRecord.java::ProcessRecord]

ProcessRecord(BatteryStatsImpl.Uid.Proc\_batteryStats,

        IApplicationThread\_thread,ApplicationInfo \_info, String \_processName) {

    batteryStats = \_batteryStats; //用于电量统计

     info =\_info; //保存ApplicationInfo

    processName = \_processName; //保存进程名

      //一个进程能运行多个Package，pkgList用于保存package名

    pkgList.add(\_info.packageName);

     thread= \_thread;//保存IApplicationThread,通过它可以和应用进程交互

     //下面这些xxxAdj成员变量和进程调度优先级及OOM\_adj有关。以后再分析它的作用

     maxAdj= ProcessList.EMPTY\_APP\_ADJ;

    hiddenAdj = ProcessList.HIDDEN\_APP\_MIN\_ADJ;

    curRawAdj = setRawAdj = -100;

     curAdj= setAdj = -100;

   //用于控制该进程是否常驻内存（即使被杀掉，系统也会重启它），只有重要的进程才会有此待遇

    persistent = false;

     removed= false;

}

ProcessRecord除保存和应用进程通信的IApplicationThread对象外，还保存了进程名、不同状态对应的Oom\_adj值及一个ApplicationInfo。一个进程虽然可运行多个Application，但是ProcessRecord一般保存该进程中先运行的那个Application的ApplicationInfo。

至此，已经创建了一个ProcessRecord对象，和其他应用进程不同的是，该对象对应的进程为SystemServer。为了彰显其特殊性，AMS为其中的一些成员变量设置了特定的值：

   app.persistent = true;//设置该值为true

   app.pid =MY\_PID;//设置pid为SystemServer的进程号

   app.maxAdj= ProcessList.SYSTEM\_ADJ;//设置最大OOM\_Adj，系统进程默认值为-16

   //另外，app的processName被设置成“system”

这时，一个针对SystemServer的ProcessRecord对象就创建完成了。此后AMS将把它并入自己的势力范围内。

AMS中有两个成员变量用于保存ProcessRecord，一个是mProcessNames，另一个是mPidsSelfLocked，如图6-6所示为这两个成员变量的数据结构示意图。

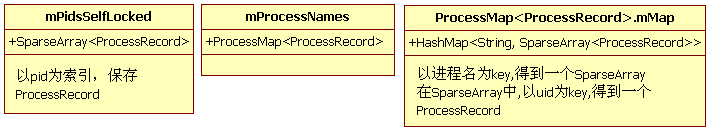


图6-6  mPidsSelfLocked和mProcessNames数据结构示意图

3.  AMS的setSystemProcess总结

现在来总结回顾setSystemProcess的工作：

·  注册AMS、meminfo、gfxinfo等服务到ServiceManager中。

·  根据PKMS返回的ApplicationInfo初始化Android运行环境，并创建一个代表SystemServer进程的ProcessRecord，从此，SystemServer进程也并入AMS的管理范围内。

6.2.3  AMS的installSystemProviders函数分析

还记得Settings数据库吗？SystemServer中很多Service都需要向它查询配置信息。为此，Android提供了一个SettingsProvider来帮助开发者。该Provider在SettingsProvider.apk中，installSystemProviders就会加载该APK并把SettingsProvider放到SystemServer进程中来运行。

此时的SystemServer已经加载了framework-res.apk，现在又要加载另外一个APK文件，这就是多个APK运行在同一进程的典型案例。另外，通过installSystemProviders函数还能见识ContentProvider的安装过程，下面就来分析它。

**提示**读者在定制自己的Android系统时，万不可去掉/system/app/SettingsProvider.apk，否则系统将无法正常启动。

[-->ActivityManagerService.java::installSystemProviders]

public static final void installSystemProviders(){

 List<ProviderInfo> providers;

 synchronized (mSelf) {

    /\*

    从mProcessNames找到进程名为“system”且uid为SYSTEM\_UID的ProcessRecord，

    返回值就是前面在installSystemApplication中创建的那个ProcessRecord，它代表

    SystemServer进程

    \*/

   ProcessRecord app = mSelf.mProcessNames.get("system",Process.SYSTEM\_UID);

    //①关键调用，见下文分析

    providers= mSelf.generateApplicationProvidersLocked(app);

    if(providers != null) {

       ......//将非系统APK（即未设ApplicationInfo.FLAG\_SYSTEM标志）提供的Provider

      //从providers列表中去掉

        }

     if(providers != null) {//②为SystemServer进程安装Provider

         mSystemThread.installSystemProviders(providers);

     }

    //监视Settings数据库中Secure表的变化，目前只关注long\_press\_timeout配置的变化

    mSelf.mCoreSettingsObserver = new CoreSettingsObserver(mSelf);

     //UsageStatsService的工作，以后再讨论

    mSelf.mUsageStatsService.monitorPackages();

 }

在代码中列出了两个关键调用，分别是：

·  调用generateApplicationProvidersLocked函数，该函数返回一个ProviderInfo List。

·  调用ActivityThread的installSystemProviders函数。ActivityThread可以看做是进程的Android运行环境，那么installSystemProviders表示为该进程安装ContentProvider。

**注意**此处不再区分系统进程还是应用进程。由于只和ActivityThread交互，因此它运行在什么进程无关紧要。

下面来看第一个关键点generateApplicationProvidersLocked函数。

1.  AMS的 generateApplicationProvidersLocked函数分析

[-->ActivityManagerService.java::generateApplicationProvidersLocked]

private final List<ProviderInfo> generateApplicationProvidersLocked(

                              ProcessRecordapp) {

  List<ProviderInfo> providers = null;

  try {

         //①向PKMS查询满足要求的ProviderInfo，最重要的查询条件包括：进程名和进程uid

         providers = AppGlobals.getPackageManager().

            queryContentProviders(app.processName, app.info.uid,

             STOCK\_PM\_FLAGS | PackageManager.GET\_URI\_PERMISSION\_PATTERNS);

        } ......

   if(providers != null) {

      finalint N = providers.size();

      for(int i=0; i<N; i++) {

         //②AMS对ContentProvider的管理，见下文解释

        ProviderInfo cpi =   (ProviderInfo)providers.get(i);

        ComponentName comp = new ComponentName(cpi.packageName, cpi.name);

        ContentProviderRecord cpr = mProvidersByClass.get(comp);

         if(cpr == null) {

             cpr = new ContentProviderRecord(cpi, app.info, comp);

              //ContentProvider在AMS中用ContentProviderRecord来表示

              mProvidersByClass.put(comp, cpr);//保存到AMS的mProvidersByClass中

          }

          //将信息也保存到ProcessRecord中

         app.pubProviders.put(cpi.name, cpr);

          //保存PackageName到ProcessRecord中

         app.addPackage(cpi.applicationInfo.packageName);

         //对该APK进行dex优化

         ensurePackageDexOpt(cpi.applicationInfo.packageName);

         }

     }

    returnproviders;

 }

由以上代码可知：generateApplicationProvidersLocked先从PKMS那里查询满足条件的ProviderInfo信息，而后将它们分别保存到AMS和ProcessRecord中对应的数据结构中。

先看查询函数queryContentProviders。

（1） PMS中 queryContentProviders函数分析

[-->PackageManagerService.java::queryContentProviders]

public List<ProviderInfo>queryContentProviders(String processName,

    int uid,int flags) {

  ArrayList<ProviderInfo> finalList = null;

  synchronized (mPackages) {

     //还记得mProvidersByComponent的作用吗？它以ComponentName为key，保存了

     //PKMS扫描APK得到的PackageParser.Provider信息。读者可参考图4-8

     finalIterator<PackageParser.Provider> i =

                      mProvidersByComponent.values().iterator();

       while(i.hasNext()) {

          final PackageParser.Provider p = i.next();

           //下面的if语句将从这些Provider中搜索本例设置的processName为“system”，

          //uid为SYSTEM\_UID，flags为FLAG\_SYSTEM的Provider

          if (p.info.authority != null

               && (processName == null

                   ||(p.info.processName.equals(processName)

                        &&p.info.applicationInfo.uid == uid))

               && mSettings.isEnabledLPr(p.info, flags)

                && (!mSafeMode || ( p.info.applicationInfo.flags

                        &ApplicationInfo.FLAG\_SYSTEM) != 0)) {

                    if (finalList == null) {

                        finalList = newArrayList<ProviderInfo>(3);

                   }

                 //由PackageParser.Provider得到ProviderInfo，并添加到finalList中

                //关于Provider类及ProviderInfo类，可参考图4-5

                  finalList.add(PackageParser.generateProviderInfo(p,flags));

               }

           }

        }

   if(finalList != null)

      //最终结果按provider的initOrder排序，该值用于表示初始化ContentProvider的顺序

      Collections.sort(finalList, mProviderInitOrderSorter);

   returnfinalList;//返回最终结果

 }

queryContentProviders函数很简单，就是从PKMS那里查找满足条件的Provider，然后生成AMS使用的ProviderInfo信息。为何偏偏能找到SettingsProvider呢？来看它的AndroidManifest.xml文件，如图6-7所示。



图6-7  SettingsProvider的AndroidManifest.xml文件示意

由图6-7可知，SettingsProvider设置了其uid为“android.uid.system”，同时在application中设置了process名为“system”。而在framework-res.apk中也做了相同的设置。所以，现在可以确认SettingsProvider将和framework-res.apk运行在同一个进程，即SystemServer中。

**提示**从运行效率角度来说，这样做也是合情合理的。因为SystemServer的很多Service都依赖Settings数据库，把它们放在同一个进程中，可以降低由于进程间通信带来的效率损失。

（2）  关于ContentProvider的介绍

前面介绍的从PKMS那里查询到的ProviderInfo还属于公有财产，现在我们要将它与AMS及ProcessRecord联系起来。

·  AMS保存ProviderInfo的原因是它要管理ContentProvider。

·  ProcessRecord保存ProviderInfo的原因是ContentProvider最终要落实到一个进程中。其实也是为了方便AMS管理，例如该进程一旦退出，AMS需要把其中的ContentProvider信息从系统中去除。

AMS及ProcessRecord均使用了一个新的数据结构ContentProviderRecord来管理ContentProvider信息。图6-8展示了ContentProviderRecord相应的数据结构。



图6-8  ContentProvicerRecord及相应的“管理团队”

由图6-8可知：

·  ContentProviderRecord从ContentProviderHolder派生，内部保存了ProviderInfo、该Provider所驻留的进程ProcessRecord，以及使用该ContentProvider的客户端进程ProcessRecord（即clients成员变量）。

·  AMS的mProviderByClass成员变量及ProcessRecord的pubProviders成员变量均以ComponentName为Key来保存对应的ContentProviderRecord对象。

至此，Provider信息已经保存到AMS及ProcessRecord中了。那么，下一步的工作是什么呢？

2.  ActivityThread 的installSystemProviders函数分析

在AMS和ProcessRecord中都保存了Provider信息，但这些仅仅都是一些信息，并不是ContentProvider，因此下面要创建一个ContentProvider实例（即SettingsProvider对象）。该工作由ActivityThread的installSystemProviders来完成，代码如下：

[-->ActivityThread.java::installSystemProviders]

public final void installSystemProviders(List<ProviderInfo>providers) {

  if(providers != null)

    //调用installContentProviders，第一个参数真实类型是Application

   installContentProviders(mInitialApplication, providers);

}

installContentProviders这个函数是所有ContentProvider产生的必经之路，其代码如下：

[-->ActivityThread.java::installContentProviders]

private void installContentProviders(

           Context context, List<ProviderInfo> providers) {

   finalArrayList<IActivityManager.ContentProviderHolder> results =

                   new ArrayList<IActivityManager.ContentProviderHolder>();

  Iterator<ProviderInfo> i = providers.iterator();

   while(i.hasNext()) {

    ProviderInfo cpi = i.next();

     //①调用installProvider函数，得到一个IContentProvider对象

    IContentProvider cp = installProvider(context, null, cpi, false);

     if (cp!= null) {

       IActivityManager.ContentProviderHolder cph =

                    newIActivityManager.ContentProviderHolder(cpi);

       cph.provider = cp;

       //将返回的cp保存到results数组中

       results.add(cph);

        synchronized(mProviderMap){

         //mProviderRefCountMap，；类型为HashMap<IBinder,ProviderRefCount>，

         //主要通过ProviderRefCount对ContentProvider进行引用计数控制，一旦引用计数

         //降为零，表示系统中没有地方使用该ContentProvider，要考虑从系统中注销它

         mProviderRefCountMap.put(cp.asBinder(), new ProviderRefCount(10000));

         }

     }

   }

    try {

       //②调用AMS的publishContentProviders注册这些ContentProvider，第一个参数

       //为ApplicationThread

       ActivityManagerNative.getDefault().publishContentProviders(

                      getApplicationThread(), results);

        } ......

    }

installContentProviders实际上是标准的ContentProvider安装时调用的程序。安装ContentProvider包括两方面的工作：

·  先在ActivityThread中通过installProvider得到一个ContentProvider实例。

·  向AMS发布这个ContentProvider实例。如此这般，一个APK中声明的ContentProvider才能登上历史舞台，发挥其该有的作用。

提示 上述工作其实和Binder Service类似，一个Binder Service也需要先创建，然后注册到ServiceManager中。

马上来看ActivityThread的installProvider函数。

（1） ActivityThread的installProvider函数分析

[-->ActivityThread.java::installProvider]

private IContentProvider installProvider(Contextcontext,

           IContentProvider provider, ProviderInfoinfo, boolean noisy) {

//注意本例所传的参数：context为mInitialApplication，provider为null,info不为null，

// noisy为false

  ContentProvider localProvider = null;

   if(provider == null) {

      Context c = null;

      ApplicationInfo ai = info.applicationInfo;

       /\*

        下面这个if判断的作用就是为该ContentProvider找到对应的Application。

        在AndroidManifest.xml中，ContentProvider是Application的子标签，所以

       ContentProvider和Application有一种对应关系。在本例中，传入的context（

        其实是mInitialApplication）代表的是framework-res.apk，而Provider代表的

        是SettingsProvider。而SettingsProvider.apk所对应的Application还未创建，

        所以下面的判断语句最终会进入最后的else分支

      \*/

       if(context.getPackageName().equals(ai.packageName)) {

          c= context;

       }else if (mInitialApplication != null &&

                 mInitialApplication.getPackageName().equals(ai.packageName)){

          c = mInitialApplication;

       } else {

         try{

             //ai.packageName应该是SettingsProvider.apk的Package，

             //名为“com.android.providers.settings”

             //下面将创建一个Context，指向该APK

              c = context.createPackageContext(ai.packageName,

                           Context.CONTEXT\_INCLUDE\_CODE);

           }

       }//if(context.getPackageName().equals(ai.packageName))判断结束

       if (c == null)  return null;

       try {

         /\*

         为什么一定要找到对应的Context呢？除了ContentProvider和Application的

         对应关系外，还有一个决定性原因：即只有对应的Context才能加载对应APK的Java字节码，

         从而可通过反射机制生成ContentProvider实例

        \*/

         finaljava.lang.ClassLoader cl = c.getClassLoader();

         //通过Java反射机制得到真正的ContentProvider，

         //此处将得到一个SettingsProvider对象

         localProvider=(ContentProvider)cl.loadClass(info.name).newInstance();

         //从ContentProvider中取出其mTransport成员（见下文分析）

         provider =localProvider.getIContentProvider();

         if (provider == null) return null;

         //初始化该ContentProvider，内部会调用其onCreate函数

        localProvider.attachInfo(c, info);

      }......

   }//if(provider == null)判断结束

  synchronized (mProviderMap) {

      /\*

        ContentProvider必须指明一个和多个authority，在第4章曾经提到过，

         在URL中host:port的组合表示一个authority。这个单词不太好理解，可简单

         认为它用于指定ContentProvider的位置（类似网站的域名）

     \*/

      String names[] =PATTERN\_SEMICOLON.split(info.authority);

      for (int i=0; i<names.length; i++) {

             ProviderClientRecord pr = newProviderClientRecord(names[i],

                                           provider, localProvider);

            try {

                //下面这句对linkToDeath的调用颇让人费解，见下文分析

                  provider.asBinder().linkToDeath(pr, 0);

                  mProviderMap.put(names[i], pr);

           }......

      }//for循环结束

     if(localProvider != null) {

         // mLocalProviders用于存储由本进程创建的ContentProvider信息

          mLocalProviders.put(provider.asBinder(),

                   new ProviderClientRecord(null, provider, localProvider));

              }

     }//synchronized 结束

  return provider;

}

以上代码不算复杂，但是涉及一些数据结构和一句令人费解的对inkToDeath函数的调用。先来说说那句令人费解的调用。

在本例中，provider变量并非通过函数参数传入，而是在本进程内部创建的。provider在本例中是Bn端（后面分析ContentProvider的getIContentProvider时即可知道），Bn端进程为Bn端设置死亡通知本身就比较奇怪。如果Bn端进程死亡，它设置的死亡通知也无法发送给自己。幸好源代码中有句注释：“Cache the pointer for the remote provider”。意思是如果provider参数是通过installProvider传递过来的（即该Provider代表远端进程的ContentProvider，此时它应为Bp端），那么这种处理是合适的。不管怎样，这仅仅是为了保存pointer，所以也无关宏旨。

至于代码中涉及的数据结构，我们整理为如图6-9所示。



图6-9  ActivityThread中ContentProvider涉及的数据结构

由图6-9可知：

·  ContentProvider类本身只是一个容器，而跨进程调用的支持是通过内部类Transport实现的。Transport从ContentProviderNative派生，而ContentProvider的成员变量mTransport指向该Transport对象。ContentProvider的getIContentProvider函数即返回mTransport成员变量。

·  ContentProviderNative从Binder派生，并实现了IContentProvider接口。其内部类ContentProviderProxy是供客户端使用的。

·  ProviderClientRecord是ActivityThread提供的用于保存ContentProvider信息的一个数据结构。它的mLocalProvider用于保存ContentProvider对象，mProvider用于保存IContentProvider对象。另外一个成员mName用于保存该ContentProvider的一个authority。注意，ContentProvider可以定义多个authority，就好像一个网站有多个域名一样。

至此，本例中的SettingProvider已经创建完毕，接下来的工作就是把它推向历史舞台——即发布该Provider。

（2） ASM的 publishContentProviders分析

publicContentProviders函数用于向AMS注册ContentProviders，其代码如下：

[-->ActivityManagerService.java::publishContentProviders]

 publicfinal void publishContentProviders(IApplicationThread caller,

                           List<ContentProviderHolder>providers) {

  ......

  synchronized(this){

    //找到调用者所在的ProcessRecord对象

   finalProcessRecord r = getRecordForAppLocked(caller);

   ......

   finallong origId = Binder.clearCallingIdentity();

   final intN = providers.size();

   for (inti=0; i<N; i++) {

      ContentProviderHolder src = providers.get(i);

       ......

       //①注意：先从该ProcessRecord中找对应的ContentProviderRecord

     ContentProviderRecord dst = r.pubProviders.get(src.info.name);

      if(dst != null) {

          ComponentName comp = newComponentName(dst.info.packageName,

                                                         dst.info.name);

          //以ComponentName为key，保存到mProvidersByClass中

         mProvidersByClass.put(comp, dst);

         String names[] = dst.info.authority.split(";");

         for (int j = 0; j < names.length; j++)

               mProvidersByName.put(names[j], dst);//以authority为key，保存

              //mLaunchingProviders用于保存处于启动状态的Provider

               int NL = mLaunchingProviders.size();

               int j;

               for (j=0; j<NL; j++) {

                 if (mLaunchingProviders.get(j) ==dst) {

                      mLaunchingProviders.remove(j);

                      j--;

                      NL--;

                  }//

              }//for (j=0; j<NL; j++)结束

              synchronized (dst) {

                  dst.provider = src.provider;

                  dst.proc = r;

                  dst.notifyAll();

              }//synchronized结束

             updateOomAdjLocked(r);//每发布一个Provider，需要调整对应进程的oom\_adj

            }//for(int j = 0; j < names.length; j++)结束

        }//for(int i=0; i<N; i++)结束

        Binder.restoreCallingIdentity(origId);

    }// synchronized(this)结束

 }

这里应解释一下publishContentProviders的工作流程：

·  先根据调用者的pid找到对应的ProcessRecord对象。

·  该ProcessRecord的pubProviders中保存了ContentProviderRecord信息。该信息由前面介绍的AMS的generateApplicationProvidersLocked函数根据Package本身的信息生成。此处将判断要发布的ContentProvider是否由该Package声明。

·  如果判断返回成功，则将该ContentProvider及其对应的authority加到mProvidersByName中。注意，AMS中还有一个mProvidersByClass变量，该变量以ContentProvider的ComponentName为key，即系统提供多种方式找到某一个ContentProvider，一种是通过 authority，另一种方式就是指明ComponentName。

·  mLaunchingProviders和最后的notifyAll函数用于通知那些等待ContentProvider所在进程启动的客户端进程。例如，进程A要查询一个数据库，需要通过进程B中的某个ContentProvider 来实施。如果B还未启动，那么AMS就需要先启动B。在这段时间内，A需要等待B启动并注册对应的ContentProvider。B一旦完成注册，就需要告知A退出等待以继续后续的查询工作。

现在，一个SettingsProvider就算正式在系统中挂牌并注册了，此后，和Settings数据库相关的操作均由它来打理。

3.  ASM的installSystemProviders总结

AMS的installSystemProviders函数其实就是用于启动SettingsProvider，其中比较复杂的是ContentProvider相关的数据结构，读者可参考图6-9。

6.2.4  ASM的systemReady分析

作为核心服务，AMS的systemReady会做什么呢？由于该函数内容较多，我们将它分为三段。首先看第一段的工作。

1.  systemReady第一阶段的工作

[-->ActivityManagerService.java::systemReady]

public void systemReady(final RunnablegoingCallback) {

     synchronized(this){

       ......

       if(!mDidUpdate) {//判断是否为升级

           if(mWaitingUpdate)   return; //升级未完成，直接返回

           //准备PRE\_BOOT\_COMPLETED广播

            Intent intent = newIntent(Intent.ACTION\_PRE\_BOOT\_COMPLETED);

           List<ResolveInfo> ris = null;

           //向PKMS查询该广播的接收者

            ris= AppGlobals.getPackageManager().queryIntentReceivers(

                                intent, null,0);

            ......//从返回的结果中删除那些非系统APK的广播接收者

            intent.addFlags(Intent.FLAG\_RECEIVER\_BOOT\_UPGRADE);

            //读取/data/system/called\_pre\_boots.dat文件，这里存储了上次启动时候已经

             //接收并处理PRE\_BOOT\_COMPLETED广播的组件。鉴于该广播的特殊性，系统希望

           //该广播仅被这些接收者处理一次

             ArrayList<ComponentName>lastDoneReceivers =

                                          readLastDonePreBootReceivers();

              final ArrayList<ComponentName> doneReceivers=

                                         newArrayList<ComponentName>();

             ......//从PKMS返回的接收者中删除那些已经处理过该广播的对象

            for (int i=0; i<ris.size(); i++) {

                ActivityInfo ai = ris.get(i).activityInfo;

                 ComponentName comp = newComponentName(ai.packageName, ai.name);

                doneReceivers.add(comp);

                intent.setComponent(comp);

                IIntentReceiver finisher = null;

                if (i == ris.size()-1) {

                    //为最后一个广播接收者注册一个回调通知，当该接收者处理完广播后，将调用该

                   //回调

                   finisher = new IIntentReceiver.Stub() {

                       public voidperformReceive(Intent intent, int resultCode,

                                        Stringdata, Bundle extras, boolean ordered,

                                        booleansticky) {

                                 mHandler.post(newRunnable() {

                                   public void run(){

                                   synchronized(ActivityManagerService.this) {

                                       mDidUpdate = true;

                                    }

                                 //保存那些处理过该广播的接收者信息

                                 writeLastDonePreBootReceivers(doneReceivers);

                                  showBootMessage(mContext.getText(

                                        R.string.android\_upgrading\_complete),

                                        false);

                                   systemReady(goingCallback);

                                }//run结束

                            });// new Runnable结束

                        }//performReceive结束

                  };//finisher创建结束

              }// if (i == ris.size()-1)判断结束

           //发送广播给指定的接收者

             broadcastIntentLocked(null, null, intent,null, finisher,

              0, null, null, null, true, false, MY\_PID, Process.SYSTEM\_UID);

            if (finisher != null)   mWaitingUpdate = true;

         }

       if(mWaitingUpdate)    return;

       mDidUpdate= true;

  }

   mSystemReady = true;

    if(!mStartRunning)  return;

   }//synchronized(this)结束

由以上代码可知，systemReady第一阶段的工作并不轻松，其主要职责是发送并处理与PRE\_BOOT\_COMPLETED广播相关的事情。目前代码中还没有接收该广播的地方，不过从代码中的注释中可猜测到，该广播接收者的工作似乎和系统升级有关。

**建议**如有哪位读者了解与此相关的知识，不妨和大家分享。

下面来介绍systemReady第二阶段的工作。

2.  systemReady第二阶段的工作

[-->ActivityManagerService.java::systemReady]

   ArrayList<ProcessRecord>procsToKill = null;

  synchronized(mPidsSelfLocked) {

     for(int i=mPidsSelfLocked.size()-1; i>=0; i--) {

         ProcessRecord proc = mPidsSelfLocked.valueAt(i);

          //从mPidsSelfLocked中找到那些先于AMS启动的进程，哪些进程有如此能耐，

         //在AMS还未启动完毕就启动完了呢？对，那些声明了persistent为true的进程有可能

          if(!isAllowedWhileBooting(proc.info)){

            if (procsToKill == null)

                 procsToKill = new ArrayList<ProcessRecord>();

             procsToKill.add(proc);

           }

       }//for结束

   }// synchronized结束

   synchronized(this){

       if(procsToKill != null) {

          for (int i=procsToKill.size()-1; i>=0; i--) {

             ProcessRecord proc = procsToKill.get(i);

             //把这些进程关闭，removeProcessLocked函数比较复杂，以后再分析

              removeProcessLocked(proc, true, false);

           }

        }

      //至此，系统已经准备完毕

      mProcessesReady = true;

   }

  synchronized(this) {

     if(mFactoryTest == SystemServer.FACTORY\_TEST\_LOW\_LEVEL) {

     }//和工厂测试有关，不对此进行讨论

   }

   //查询Settings数据，获取一些配置参数

   retrieveSettings();

systemReady第二阶段的工作包括：

·  杀死那些竟然在AMS还未启动完毕就先启动的应用进程。注意，这些应用进程一定是APK所在的Java进程，因为只有应用进程才会向AMS注册，而一般Native（例如mediaserver）进程是不会向AMS注册的。

·  从Settings数据库中获取配置信息，目前只取4个配置参数，分别是："debug\_app"（设置需要debug的app的名称）、"wait\_for\_debugger"（如果为1，则等待调试器，否则正常启动debug\_app）、"always\_finish\_activities"（当一个activity不再有地方使用时，是否立即对它执行destroy）、"font\_scale"（用于控制字体放大倍数，这是Android 4.0新增的功能）。以上配置项由Settings数据库的System表提供。

3.  systemReady第三阶段的工作

[-->ActivityManagerService.java::systemReady]

//调用systemReady传入的参数，它是一个Runnable对象，下节将分析此函数

if (goingCallback != null) **goingCallback.run**();

 synchronized (this) {

     if(mFactoryTest != SystemServer.FACTORY\_TEST\_LOW\_LEVEL) {

        try{

            //从PKMS中查询那些persistent为1的ApplicationInfo

            List apps = AppGlobals.getPackageManager().

                           getPersistentApplications(STOCK\_PM\_FLAGS);

            if (apps != null) {

                int N = apps.size();

                 int i;

                 for (i=0; i<N; i++) {

                   ApplicationInfo info = (ApplicationInfo)apps.get(i);

                   //由于framework-res.apk已经由系统启动，所以这里需要把它去除

                   //framework-res.apk的packageName为"android"

                   if (info != null && !info.packageName.equals("android"))

                        addAppLocked(info);//启动该Application所在的进程

                      }

                   }

               }......

      }

   mBooting= true; //设置mBooting变量为true，其作用后面会介绍

   try {

        if(AppGlobals.getPackageManager().hasSystemUidErrors()) {

            ......//处理那些Uid有错误的Application

      }......

       //启动全系统第一个Activity，即Home

      mMainStack.resumeTopActivityLocked(null);

    }// synchronized结束

}

systemReady第三阶段的工作有3项：

·  调用systemReady设置的回调对象goingCallback的run函数。

·  启动那些声明了persistent的APK。

·  启动桌面。

先看回调对象goingCallback的run函数的工作。

（1） goingCallback的run函数分析

[-->SystemServer.java::ServerThread.run]

ActivityManagerService.self().systemReady(newRunnable() {

    publicvoid run() {

   startSystemUi(contextF);//启动SystemUi

   //调用其他服务的systemReady函数

    if(batteryF != null) batteryF.systemReady();

    if(networkManagementF != null) networkManagementF.systemReady();

    ......

    Watchdog.getInstance().start();//启动Watchdog

    ......//调用其他服务的systemReady函数

 }

run函数比较简单，执行的工作如下：

·  执行startSystemUi，在该函数内部启动SystemUIService，该Service和状态栏有关。

·  调用一些服务的systemReady函数。

·  启动Watchdog。

startSystemUi的代码如下：

[-->SystemServer.java::startSystemUi]

static final void startSystemUi(Context context) {

     Intentintent = new Intent();

     intent.setComponent(new ComponentName("com.android.systemui",

                 "com.android.systemui.SystemUIService"));

       context.startService(intent);

 }

SystemUIService由SystemUi.apk提供，它实现了系统的状态栏。

**注意**在精简ROM时，也不能删除SystemUi.apk。

（2）  启动Home界面

如前所述，resumeTopActivityLocked将启动Home界面，此函数非常重要也比较复杂，故以后再详细分析。我们提取了resumeTopActivityLocked启动Home界面时的相关代码，如下所示：

[-->ActivityStack.java::resumeTopActivityLocked]

final booleanresumeTopActivityLocked(ActivityRecord prev) {

   //找到下一个要启动的Activity

  ActivityRecord next = topRunningActivityLocked(null);

   finalboolean userLeaving = mUserLeaving;

  mUserLeaving = false;

   if (next== null) {

       //如果下一个要启动的ActivityRecord为空，则启动Home

       if(mMainStack) {//全系统就一个ActivityStack，所以mMainStack永远为true

          //mService指向AMS

          return mService.startHomeActivityLocked();//mService指向AMS

        }

    }

   ......//以后再详细分析

}

下面来看AMS的startHomeActivityLocked函数，代码如下：

[-->ActivityManagerService.java::startHomeActivityLocked]

boolean startHomeActivityLocked() {

   Intentintent = new Intent( mTopAction,

               mTopData != null ? Uri.parse(mTopData) :null);

   intent.setComponent(mTopComponent);

   if(mFactoryTest != SystemServer.FACTORY\_TEST\_LOW\_LEVEL)

       intent.addCategory(Intent.CATEGORY\_HOME);//添加Category为HOME类别

    //向PKMS查询满足条件的ActivityInfo

   ActivityInfo aInfo =

           intent.resolveActivityInfo(mContext.getPackageManager(),

                                  STOCK\_PM\_FLAGS);

    if(aInfo != null) {

       intent.setComponent(new ComponentName(

                          aInfo.applicationInfo.packageName,aInfo.name));

       ProcessRecordapp = getProcessRecordLocked(aInfo.processName,

                                             aInfo.applicationInfo.uid);

       //在正常情况下，app应该为null，因为刚开机，Home进程肯定还没启动

       if(app == null || app.instrumentationClass == null) {

            intent.setFlags(intent.getFlags()|

                                 Intent.FLAG\_ACTIVITY\_NEW\_TASK);

            //启动Home

            mMainStack.startActivityLocked(null, intent, null, null, 0, aInfo,

                                  null, null, 0, 0, 0, false, false,null);

          }

      }//if(aInfo != null)判断结束

  return true;

}

至此，AMS携诸位Service都启动完毕，Home也靓丽登场，整个系统就准备完毕，只等待用户的检验了。不过在分析逻辑上还有一点没涉及，那会是什么呢？

（3） 发送ACTION\_BOOT\_COMPLETED广播

由前面的代码可知，AMS发送了ACTION\_PRE\_BOOT\_COMPLETED广播，可系统中没有地方处理它。在前面的章节中，还碰到一个ACTION\_BOOT\_COMPLETED广播，该广播广受欢迎，却不知道它是在哪里发送的。

当Home Activity启动后，ActivityStack的activityIdleInternal函数将被调用，其中有一句代码颇值得注意：

[-->ActivityStack.java::activityIdleInternal]

final ActivityRecord activityIdleInternal(IBindertoken, boolean fromTimeout,

           Configuration config) {

   booleanbooting = false;

   ......

   if(mMainStack) {

      booting = mService.mBooting; //在systemReady的第三阶段工作中设置该值为true

      mService.mBooting = false;

   }

   ......

   if(booting)   mService.finishBooting();//调用AMS的finishBooting函数

}

[-->ActivityManagerService.java::finishBooting]

final void finishBooting() {

   IntentFilter pkgFilter = new IntentFilter();

   pkgFilter.addAction(Intent.ACTION\_QUERY\_PACKAGE\_RESTART);

   pkgFilter.addDataScheme("package");

   mContext.registerReceiver(new BroadcastReceiver() {

      public void onReceive(Context context, Intent intent) {

       ......//处理Package重启的广播

      }

    }, pkgFilter);

  synchronized (this) {

      finalint NP = mProcessesOnHold.size();

      if (NP> 0) {

          ArrayList<ProcessRecord> procs =

                   new ArrayList<ProcessRecord>(mProcessesOnHold);

           for(int ip=0; ip<NP; ip++)//启动那些等待启动的进程

             startProcessLocked(procs.get(ip), "on-hold", null);

           }

       if(mFactoryTest != SystemServer.FACTORY\_TEST\_LOW\_LEVEL) {

          //每15钟检查系统各应用进程使用电量的情况，如果某进程使用WakeLock时间

          //过长，AMS将关闭该进程

          Message nmsg =

                      mHandler.obtainMessage(CHECK\_EXCESSIVE\_WAKE\_LOCKS\_MSG);

          mHandler.sendMessageDelayed(nmsg, POWER\_CHECK\_DELAY);

           //设置系统属性sys.boot\_completed的值为1

           SystemProperties.set("sys.boot\_completed","1");

          //发送ACTION\_BOOT\_COMPLETED广播

          broadcastIntentLocked(null, null,

                        newIntent(Intent.ACTION\_BOOT\_COMPLETED, null),

                        null, null, 0, null,null,

                       android.Manifest.permission.RECEIVE\_BOOT\_COMPLETED,

                        false, false, MY\_PID,Process.SYSTEM\_UID);

           }

        }

 }

原来，在Home启动成功后，AMS才发送ACTION\_BOOT\_COMPLETED广播。

4.  ASM的 systemReady总结

systemReady函数完成了系统就绪的必要工作，然后它将启动Home Activity。至此，Android系统就全部启动了。

6.2.5  初识ActivityManagerService总结

本节所分析的4个关键函数均较复杂，与之相关的知识点总结如下：

·  AMS的main函数：创建AMS实例，其中最重要的工作是创建Android运行环境，得到一个ActivityThread和一个Context对象。

·  AMS的setSystemProcess函数：该函数注册AMS和meminfo等服务到ServiceManager中。另外，它为SystemServer创建了一个ProcessRecord对象。由于AMS是Java世界的进程管理及调度中心，要做到对Java进程一视同仁，尽管SystemServer贵为系统进程，此时也不得不将其并入AMS的管理范围内。

·  AMS的installSystemProviders：为SystemServer加载SettingsProvider。

·  AMS的systemReady：做系统启动完毕前最后一些扫尾工作。该函数调用完毕后，HomeActivity将呈现在用户面前。

对AMS 调用轨迹分析是我们破解AMS的第一条线，希望读者反复阅读，以真正理解其中涉及的知识点，尤其是和Android运行环境及Context相关的知识。

6.3  startActivity分析

本节将重点分析Activity的启动过程，它是五条线中最难分析的一条，只要用心相信读者能啃动这块“硬骨头”。

6.3.1  从am说起

am和pm（见4.4.2节）一样，也是一个脚本，它用来和AMS交互，如启动Activity、启动Service、发送广播等。其核心文件在Am.java中，代码如下：

[-->Am.java::main]

public static void main(String[] args) {

   try {

        (newAm()).run(args);//构造一个Am对象，并调用run函数

   }......

 }

am的用法很多，读者可通过adb shell登录手机，然后执行am，这样就能获取它的用法。

利用am启动一个activity的方法如下：

am start [-D] [-W] [-P <FILE>][--start-profiler <FILE>] [-S] <INTENT>

其中：

·  方括号中的参数为可选项，尖括号中的参数为必选项。

·  <INTENT>参数有很多，主要是用于设置Intent的各项参数。

假设已知某个Activity的ComponentName（package名和Activity的Class名），启动这个Activity的相应命令如下：

am start -W -n com.dfp.test/.TestActivity

其中，-W选项表示am将会等目标Activity启动后才返回，-n表示后面的参数用于设置Intent的Component。就本例而言，com.dfp.test为Package名，.TestActivity为该Package下对应的Activity类名，所以将要启动的Activity的全路径名为com.dfp.test.TestActivity。

现在就以上面的命令为例来分析Am的run函数，代码如下：

[-->Am.java::run]

 privatevoid run(String[] args) throws Exception {

  mAm =ActivityManagerNative.getDefault();

  mArgs =args;

  String op= args[0];

  mNextArg =1;

  if (op.equals("start"))  runStart();//用于启动Activity

  else if ......//处理其他参数

}

runStart函数用于处理Activity启动请求，其代码如下：

[-->Am.java::runStart]

 privatevoid runStart() throws Exception {

    Intentintent = makeIntent();

    StringmimeType = intent.getType();

    //获取mimeType，

    if(mimeType == null && intent.getData() != null

         && "content".equals(intent.getData().getScheme())) {

        mimeType = mAm.getProviderMimeType(intent.getData());

    }

    if(mStopOption) {

     ......//处理-S选项，即先停止对应的Activity，再启动它

    }

    //FLAG\_ACTIVITY\_NEW\_TASK这个标签很重要

   intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);

   ParcelFileDescriptor fd = null;

    if(mProfileFile != null) {

         try{......//处理-P选项，用于性能统计

            fd = ParcelFileDescriptor.open(......)

          }......

    }

   IActivityManager.WaitResult result = null;

    int res;

    if(mWaitOption) {//mWaitOption控制是否等待启动结果，如果有-W选项，则该值为true

               //调用AMS的startActivityAndWait函数

        result = mAm.startActivityAndWait(null,intent, mimeType,

                        null, 0, null, null, 0,false, mDebugOption,

                        mProfileFile, fd,mProfileAutoStop);

         res= result.result;

     } ......

     ......//打印结果

 }

am最终将调用AMS的startActivityAndWait函数来处理这次启动请求。下面将深入到AMS内部去继续这次旅程。

**提示**为什么选择从am来分析Activity的启动呢？如果选择从一个Activity来分析如何启动另一个Activity，则将给人一种鸡生蛋、蛋孵鸡的感觉，故此处选择从am入手。除此之外，从am来分析Activity的启动也是Activity启动分析中相对简单的一条路线。

6.3.2  AMS的startActivityAndWait函数分析

startActivityAndWait函数有很多参数，先来认识一下它们。

[-->ActiivtyManagerService.java::startActivityAndWait原型]

 publicfinal WaitResult startActivityAndWait(

   /\*

    在绝大多数情况下，一个Acitivity的启动是由一个应用进程发起的，IApplicationThread是

    应用进程和AMS交互的通道，也可算是调用进程的标示，在本例中，AM并非一个应用进程，所以

    传递的caller为null

   \*/

  IApplicationThread caller,

   //Intent及resolvedType，在本例中，resolvedType为null

   Intentintent, String resolvedType,

   //下面两个参数和授权有关，读者可参考第3章对CopyboardService分析中介绍的授权知识

   Uri[] grantedUriPermissions,//在本例中为null

   intgrantedMode,//在本例中为0

   IBinder resultTo,//在本例中为null，用于接收startActivityForResult的结果

   StringresultWho,//在本例中为null

   //在本例中为0，该值的具体意义由调用者解释。如果该值大于等于0，则AMS内部保存该值，

   //并通过onActivityResult返回给调用者

   int requestCode,

   boolean onlyIfNeeded,//本例为false

   boolean debug,//是否调试目标进程

   //下面3个参数和性能统计有关

   StringprofileFile,

   ParcelFileDescriptor profileFd, booleanautoStopProfiler)

关于以上代码中一些参数的具体作用，以后碰到时会再作分析。建议读者先阅读SDK文档中关于Activity类定义的几个函数，如startActivity、startActivityForResult及onActivityResult等。

startActivityAndWait的代码如下：

[-->ActivityManagerService.java::startActivityAndWait]

 publicfinal WaitResult startActivityAndWait(IApplicationThread caller,

      Intent intent, String resolvedType, Uri[]grantedUriPermissions,

      int grantedMode, IBinder resultTo,StringresultWho, int requestCode,

      booleanonlyIfNeeded, boolean debug,String profileFile,

      ParcelFileDescriptor profileFd, booleanautoStopProfiler) {

     //创建WaitResult对象用于存储处理结果

    WaitResult res = new WaitResult();

    //mMainStack为ActivityStack类型，调用它的startActivityMayWait函数

   mMainStack.startActivityMayWait(caller, -1, intent, resolvedType,

               grantedUriPermissions, grantedMode, resultTo, resultWho,

               requestCode, onlyIfNeeded, debug, profileFile, profileFd,

              autoStopProfiler, res, null);//最后一个参数为Configuration，

    //在本例中为null

    returnres;

 }

mMainStack为AMS的成员变量，类型为ActivityStack，该类是Activity调度的核心角色。正式分析它之前，有必要先介绍一下相关的基础知识。

1.  Task、Back Stack、ActivityStack及Launch mode

（1） 关于Task及Back Stack的介绍

先来看图6-10。

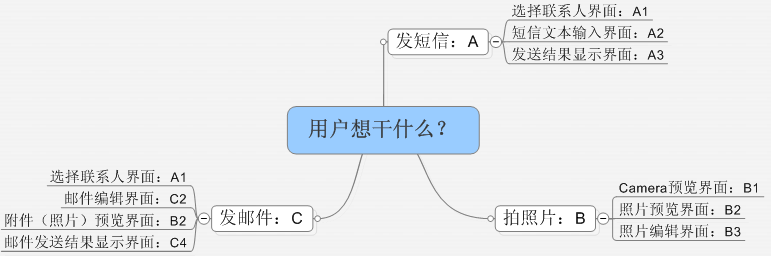


图6-10  用户想干什么

图6-10列出了用户在Android系统上想干的三件事情，分别用A、B、C表示，将每一件事情称为一个Task。一个Task还可细分为多个子步骤，即Activity。

**提示**为什么叫Activity？读者可参考Merrian-Webster词典对Activity的解释[③]：“an organizational unit forperforming a specific function”，也就是说，它是一个有组织的单元，用于完成某项指定功能。

由图6-10可知，A、B两个Task使用了不同的Activity来完成相应的任务。注意，A、B这两个Task的Activity之间没有复用。

再来看C这个Task，它可细分为4个Activity，其中有两个Activity分别使用了A Task的A1、B Task的B2。C Task为什么不新建自己的Activity，反而要用其他Task的呢？这是因为用户想做的事情（即Task）可以完全不同，但是当细分Task为Activity时，就可能出现Activity功能类似的情况。既然A1、B2已能满足要求，为何还要重复“发明轮子”呢？另外，通过重用Activity，也可为用户提供一致的界面和体验。

了解Android设计理念后，我们来看看Android是如何组织Task及它所包含的Activity的。此处有一个简单的例子，如图6-11所示。

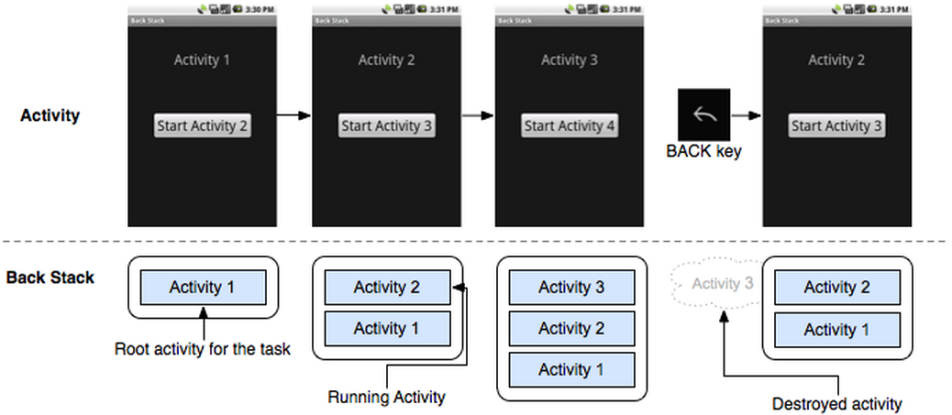


图6-11  Task及Back Stack示例

由图6-11可知：

·  本例中的Task包含4个Activity。用户可单击按钮跳转到下一个Activity。同时，通过返回键可回到上一个Activity。

·  虚线下方是这些Activity的组织方式。Android采用了Stack的方法管理这3个Activity。例如在A1启动A2后，A2入栈成为栈顶成员，A1成为栈底成员，而界面显示的是栈顶成员的内容。当按返回键时，A3出栈，这时候A2成为栈顶，界面显示也相应变成了A2。

以上是一个Task的情况。那么，多个Task又会是何种情况呢？如图6-12所示。

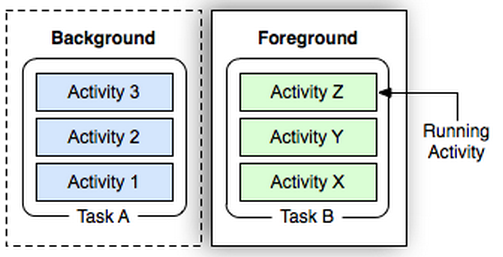
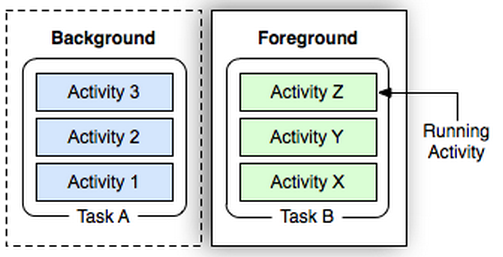


图6-12  多个Task的情况

由图6-12可知：对多Task的情况来说，系统只支持一个处于前台的Task，即用户当前看到的Activity所属的Task，其余的Task均处于后台，这些后台Task内部的Activity保持顺序不变。用户可以一次将整个Task挪到后台或者置为前台。

**提示**用过Android手机的读者应该知道，长按Home键，系统会弹出近期Task列表，使用户能快速在多个Task间切换。

以上内容从抽象角度介绍了什么是Task，以及Android如何分解Task和管理Activity，那么在实际代码中，是如何考虑并设计的呢？

（2） 关于ActivityStack的介绍

通过上述分析，我们对Android的设计有了一定了解，那么如何用代码来实现这一设计呢？此处有两点需要考虑：

·  Task内部Activity的组织方式。由图6-11可知，Android通过先入后出的方式来组织Activity。数据结构中的Stack即以这种方式工作。

·  多个Task的组织及管理方式。

Android设计了一个ActivityStack类来负责上述工作，它的组成如图6-13所示。

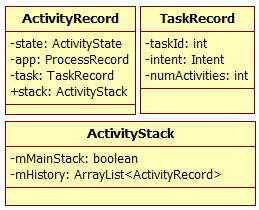


图6-13  ActivityStack及相关成员

由图6-13可知：

·  Activity由ActivityRecord表示，Task由TaskRecord表示。ActivityRecord的task成员指向该Activity所在的Task。state变量用于表示该Activity所处的状态（包括INITIALIZING、RESUMED、PAUSED等状态）。

·  ActivityStack用mHistory这个ArrayList保存ActivityRecord。令人大跌眼镜的是，该mHistory保存了系统中所有Task的ActivityRecord，而不是针对某个Task进行保存。

·  ActivityStack的mMainStack成员比较有意思，它代表此ActivityStack是否为主ActivityStack。有主必然有从，但是目前系统中只有一个ActivityStack，并且它的mMainStack为true。从ActivityStack的命名可推测，Android在开发之初也想用ActivityStack来管理单个Task中的ActivityRecord（在ActivityStack.java的注释中说过，该类为“State and management of asingle stack of activities”），但不知何故，在现在的代码实现将所有Task的ActivityRecord都放到mHistory中了，并且依然保留mMainStack。

·  ActivityStack中没有成员用于保存TaskRecord。

由上述内容可知，ActivityStack采用数组的方式保存所有Task的ActivityRecord，并且没有成员保存TaskRecord。这种实现方式有优点亦有缺点：

·  优点是少了TaskRecord一级的管理，直接以ActivityRecord为管理单元。这种做法能降低管理方面的开销。

·  缺点是弱化了Task的概念，结构不够清晰。

下面来看ActivityStack中几个常用的搜索ActivityRecord的函数，代码如下：

[-->ActivityStack.java::topRunningActivityLocked]

/\* topRunningActivityLocked:

找到栈中第一个与notTop不同的，并且不处于finishing状态的ActivityRecord。当notTop为

null时，该函数即返回栈中第一个需要显示的ActivityRecord。提醒读者，栈的出入口只能是栈顶。

虽然mHistory是一个数组，但是查找均从数组末端开始，所以其行为也粗略符合Stack的定义

\*/

final ActivityRecordtopRunningActivityLocked(ActivityRecord notTop) {

    int i =mHistory.size()-1;

    while (i>= 0) {

        ActivityRecord r = mHistory.get(i);

         if (!r.finishing && r != notTop)  return r;

         i--;

     }

     returnnull;

  }

类似的函数还有：

[-->ActivityStack.java::topRunningNonDelayedActivityLocked]

/\* topRunningNonDelayedActivityLocked

与topRunningActivityLocked类似，但ActivityRecord要求增加一项，即delayeResume为

false

\*/

final ActivityRecordtopRunningNonDelayedActivityLocked(ActivityRecord notTop) {

     int i =mHistory.size()-1;

     while(i >= 0) {

        ActivityRecord r = mHistory.get(i);

        //delayedResume变量控制是否暂缓resume Activity

         if (!r.finishing && !r.delayedResume&& r != notTop)   return r;

        i--;

     }

    returnnull;

  }

ActivityStack还提供findActivityLocked函数以根据Intent及ActivityInfo来查找匹配的ActivityRecord，同样，查找也是从mHistory尾端开始，相关代码如下：

[-->ActivityStack.java::findActivityLocked]

private ActivityRecord findActivityLocked(Intentintent, ActivityInfo info) {

  ComponentName cls = intent.getComponent();

   if(info.targetActivity != null)

        cls= new ComponentName(info.packageName, info.targetActivity);

   final intN = mHistory.size();

   for (inti=(N-1); i>=0; i--) {

      ActivityRecord r = mHistory.get(i);

     if (!r.finishing)

         if (r.intent.getComponent().equals(cls))return r;

   }

  return null;

 }

另一个findTaskLocked函数的返回值是ActivityRecord，其代码如下：

[ActivityStack.java::findTaskLocked]

private ActivityRecord findTaskLocked(Intentintent, ActivityInfo info) {

  ComponentName cls = intent.getComponent();

   if(info.targetActivity != null)

        cls= new ComponentName(info.packageName, info.targetActivity);

  TaskRecord cp = null;

   final intN = mHistory.size();

   for (inti=(N-1); i>=0; i--) {

       ActivityRecord r = mHistory.get(i);

       //r.task!=cp，表示不搜索属于同一个Task的ActivityRecord

        if(!r.finishing && r.task != cp

                   && r.launchMode != ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {

              cp = r.task;

              //如果Task的affinity相同，则返回这条ActivityRecord

              if(r.task.affinity != null) {

               if(r.task.affinity.equals(info.taskAffinity))  return r;

               }else if (r.task.intent != null

                        &&r.task.intent.getComponent().equals(cls)) {

                 //如果Task Intent的ComponentName相同

                     return r;

                }else if (r.task.affinityIntent != null

                        &&r.task.affinityIntent.getComponent().equals(cls)) {

                   //Task affinityIntent考虑

                       return r;

               }//if (r.task.affinity != null)判断结束

         }//if(!r.finishing && r.task != cp......)判断结束

     }//for循环结束

    returnnull;

 }

其实，findTaskLocked是根据mHistory中ActivityRecord所属的Task的情况来进行相应的查找工作。

以上这4个函数均是ActivityStack中常用的函数，如果不需要逐项（case by case）地研究AMS，那么读者仅需了解这几个函数的作用即可。

（3） 关于Launch Mode的介绍

Launch Mode用于描述Activity的启动模式，目前一共有4种模式，分别是standard、singleTop、singleTask和singleInstance。初看它们，较难理解，实际上不过是Android玩的一个“小把戏“而已。启动模式就是用于控制Activity和Task关系的。

·  standard：一个Task中可以有多个相同类型的Activity。注意，此处是相同类型的Activity，而不是同一个Activity对象。例如在Task中有A、B、C、D4个Activity，如果再启动A类Activity， Task就会变成A、B、C、D、A。最后一个A和第一个A是同一类型，却并非同一对象。另外，多个Task中也可以有同类型的Activity。

·  singleTop：当某Task中有A、B、C、D4个Activity时，如果D想再启动一个D类型的Activity，那么Task将是什么样子呢？在singleTop模式下，Task中仍然是A、B、C、D，只不过D的onNewIntent函数将被调用以处理这个新Intent，而在standard模式下，则Task将变成A、B、C、D、D，最后的D为新创建的D类型Activity对象。在singleTop这种模式下，只有目标Acitivity当前正好在栈顶时才有效，例如只有处于栈顶的D启动时才有用，如果D启动不处于栈顶的A、B、C等，则无效。

·  singleTask：在这种启动模式下，该Activity只存在一个实例，并且将和一个Task绑定。当需要此Activity时，系统会以onNewIntent方式启动它，而不会新建Task和Activity。注意，该Activity虽只有一个实例，但是在Task中除了它之外，还可以有其他的Activity。

·  singleInstance：它是singleTask的加强版，即一个Task只能有这么一个设置了singleInstance的Activity，不能再有别的Activity。而在singleTask模式中，Task还可以有其他的Activity。

注意，Android建议一般的应用开发者不要轻易使用最后两种启动模式。因为这些模式虽然名意上为Launch Mode，但是它们也会影响Activity出栈的顺序，导致用户按返回键返回时导致不一致的用户体验。

除了启动模式外，Android还有其他一些标志用于控制Activity及Task之间的关系。这里只列举一二，详细信息请参阅SDK文档中Intent的相关说明。

·  FLAG\_ACTIVITY\_NEW\_TASK：将目标Activity放到一个新的Task中。

·  FLAG\_ACTIVITY\_CLEAR\_TASK：当启动一个Activity时，先把和目标Activity有关联的Task“干掉“，然后启动一个新的Task，并把目标Activity放到新的Task中。该标志必须和FLAG\_ACTIVITY\_NEW\_TASK标志一起使用。

·  FLAG\_ACTIVITY\_CLEAR\_TOP：当启动一个不处于栈顶的Activity时候，先把排在它前面的Activity“干掉”。例如Task有A、B、C、D4个Activity，要要启动B，直接把C、D“干掉”，而不是新建一个B。

**提示**这些启动模式和标志，在笔者看来很像洗扑克牌时的手法，因此可以称之为小把戏。虽是小把戏，但是相关代码的逻辑及分支却异常繁杂，我们应从更高的角度来看待它们。

介绍完上面的知识后，下面来分析ActivityStack的startActivityMayWait函数。

2.  ActivityStack的startActivityMayWait函数分析

startActivityMayWait函数的目标是启动com.dfp.test.TestActivity，假设系统之前没有启动过该Activity，本例最终的结果将是：

·  由于在am中设置了FLAG\_ACTIVITY\_NEW\_TASK标志，因此除了会创建一个新的ActivityRecord外，还会新创建一个TaskRecord。

·  还需要启动一个新的应用进程以加载并运行com.dfp.test.TestActivity的一个实例。

·  如果TestActivity不是Home，还需要停止当前正在显示的Activity。

好了，将这个函数分三部分进行介绍，先来分析第一部分。

（1） startActivityMayWait分析之一

[-->ActivityStack.java::startActivityMayWait]

final int startActivityMayWait(IApplicationThreadcaller, int callingUid,

        Intentintent, String resolvedType, Uri[] grantedUriPermissions,

        intgrantedMode, IBinder resultTo,

        StringresultWho, int requestCode, boolean onlyIfNeeded,

        booleandebug, String profileFile, ParcelFileDescriptor profileFd,

        booleanautoStopProfiler, WaitResult outResult, Configuration config) {

    ......

    //在本例中，已经指明了Component，这样可省去为Intent匹配搜索之苦

    booleancomponentSpecified = intent.getComponent() != null;

    //创建一个新的Intent，防止客户传入的Intent被修改

    intent =new Intent(intent);

    //查询满足条件的ActivityInfo，在resolveActivity内部和PKMS交互，读者不妨自己

   //尝试分析该函数

    ActivityInfoaInfo = resolveActivity(intent, resolvedType, debug,

                     profileFile, profileFd,autoStopProfiler);

    synchronized(mService) {

      int callingPid;

      if (callingUid >= 0) {

          callingPid= -1;

      } else if (caller == null) {//本例中，caller为null

           callingPid= Binder.getCallingPid();//取出调用进程的Pid

           //取出调用进程的Uid。在本例中，调用进程是am，它由shell启动

           callingUid= Binder.getCallingUid();

       } else {

           callingPid= callingUid = -1;

       }// if (callingUid >= 0)判断结束

     //在本例中config为null

     mConfigWillChange= config != null

                  && mService.mConfiguration.diff(config) != 0;

    finallong origId = Binder.clearCallingIdentity();

    if (mMainStack && aInfo != null&&  (aInfo.applicationInfo.flags&

                  ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0){

        /\*

        AndroidManifest.xml中的Application标签可以声明一个cantSaveState

        属性，设置了该属性的Application将不享受系统提供的状态保存/恢复功能。

        当一个Application退到后台时，系统会为它保存状态，当调度其到前台运行时，        将恢复它之前的状态，以保证用户体验的连续性。声明了该属性的Application被称为

        “heavy weight process”。可惜系统目前不支持该属性，因为PackageParser

        将不解析该属性。详情请见PackageParser.java parseApplication函数

       \*/

    }

   ......//待续

startActivityMayWait第一阶段的工作内容相对较简单：

·  首先需要通过PKMS查找匹配该Intent的ActivityInfo。

·  处理FLAG\_CANT\_SAVE\_STATE的情况，但系统目前不支持此情况。

·  另外，获取调用者的pid和uid。由于本例的caller为null，故所得到的pid和uid均为am所在进程的uid和pid。

下面介绍startActivityMayWait第二阶段的工作。

（2） startActivityMayWait分析之二

[-->ActivityStack.java::startActivityMayWait]

    //调用此函数启动Activity，将返回值保存到res

   int res = startActivityLocked(caller, intent,resolvedType,

            grantedUriPermissions, grantedMode, aInfo,

            resultTo, resultWho, requestCode, callingPid, callingUid,

            onlyIfNeeded, componentSpecified, null);

     //如果configuration发生变化，则调用AMS的updateConfigurationLocked

    //进行处理。关于这部分内容，读者学完本章后可自行分析

    if(mConfigWillChange && mMainStack) {

            mService.enforceCallingPermission(

                  android.Manifest.permission.CHANGE\_CONFIGURATION,

                  "updateConfiguration()");

            mConfigWillChange= false;

            mService.updateConfigurationLocked(config,null, false);

   }

此处，启动Activity的核心函数是startActivityLocked，该函数异常复杂，将用一节专门分析。下面先继续分析startActivityMayWait第三阶段的工作。

（3） startActivityMayWait分析之三

[-->ActivityStack.java::startActivityMayWait]

    if(outResult != null) {

      outResult.result = res;//设置启动结果

       if(res == IActivityManager.START\_SUCCESS) {

          //将该结果加到mWaitingActivityLaunched中保存

           mWaitingActivityLaunched.add(outResult);

           do {

                try {

                         mService.wait();//等待启动结果

                       }

                } while (!outResult.timeout && outResult.who == null);

          }else if (res == IActivityManager.START\_TASK\_TO\_FRONT) {

              ......//处理START\_TASK\_TO\_FRONT结果，读者可自行分析

          }

       }//if(outResult!= null)结束

           return res;

     }

 }

第三阶段的工作就是根据返回值做一些处理，那么res返回成功（即res== IActivityManager.START\_SUCCESS的时候）后为何还需要等待呢？

这是因为目标Activity要运行在一个新的应用进程中，就必须等待那个应用进程正常启动并处理相关请求。注意，只有am设置了-W选项，才会进入wait这一状态。

6.3.3  startActivityLocked分析

startActivityLocked是startActivityMayWait第二阶段的工作重点，该函数有点长，请读者耐心看代码。

[-->ActivityStack.java::startActivityLocked]

final int startActivityLocked(IApplicationThreadcaller,

           Intent intent, String resolvedType,

           Uri[] grantedUriPermissions,

           int grantedMode, ActivityInfo aInfo, IBinder resultTo,

           String resultWho, int requestCode,

           int callingPid, int callingUid, boolean onlyIfNeeded,

             boolean componentSpecified, ActivityRecord[]outActivity) {

   int err = START\_SUCCESS;

  ProcessRecord callerApp = null;

  //如果caller不为空，则需要从AMS中找到它的ProcessRecord。本例的caller为null

   if(caller != null) {

      callerApp = mService.getRecordForAppLocked(caller);

       //其实就是想得到调用进程的pid和uid

       if(callerApp != null) {

           callingPid = callerApp.pid;//一定要保证调用进程的pid和uid正确

           callingUid = callerApp.info.uid;

          }else {//如调用进程没有在AMS中注册，则认为其是非法的

               err = START\_PERMISSION\_DENIED;

         }

     }// if (caller != null)判断结束

   //下面两个变量意义很重要。sourceRecord用于描述启动目标Activity的那个Activity，

 //resultRecord用于描述接收启动结果的Activity，即该Activity的onActivityResult

  //将被调用以通知启动结果，读者可先阅读SDK中startActivityForResult函数的说明

   ActivityRecordsourceRecord = null;

  ActivityRecord resultRecord = null;

   if(resultTo != null) {

      //本例resultTo为null，

   }

   //获取Intent设置的启动标志，它们是和Launch Mode相类似的“小把戏”，

  //所以，读者务必理解“关于Launch Mode的介绍”一节的知识点

   intlaunchFlags = intent.getFlags();

   if((launchFlags&Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0

       && sourceRecord != null) {

       ......

       /\*

         前面介绍的Launch Mode和Activity的启动有关，实际上还有一部分标志用于控制

         Activity启动结果的通知。有关FLAG\_ACTIVITY\_FORWARD\_RESULT的作用，读者可

         参考SDK中的说明。使用这个标签有个前提，即A必须先存在，正如if中sourceRecord

         不为null的判断所示。另外，读者自己可尝试编写例子，以测试FLAG\_ACTIVITY\_FORWARD\_

         RESULT标志的作用

      \*/

    }

   //检查err值及Intent的情况

    if (err== START\_SUCCESS && intent.getComponent() == null)

           err = START\_INTENT\_NOT\_RESOLVED;

    ......

    //如果err不为0，则调用sendActivityResultLocked返回错误

    if (err!= START\_SUCCESS) {

        if(resultRecord != null) {// resultRecord接收启动结果

          sendActivityResultLocked(-1,esultRecord, resultWho, requestCode,

            Activity.RESULT\_CANCELED, null);

        }

       .......

        returnerr;

    }

    //检查权限

    finalint perm = mService.checkComponentPermission(aInfo.permission,

            callingPid,callingUid,aInfo.applicationInfo.uid, aInfo.exported);

    ......//权限检查失败的处理，不必理会

   if (mMainStack) {

       //可为AMS设置一个IActivityController类型的监听者，AMS有任何动静都会回调该

       //监听者。不过谁又有如此本领去监听AMS呢？在进行Monkey测试的时候，Monkey会

       //设置该回调对象。这样，Monkey就能根据AMS放映的情况进行相应处理了

        if(mService.mController != null) {

             boolean abort = false;

            try {

                   Intent watchIntent = intent.cloneFilter();

                   //交给回调对象处理，由它判断是否能继续后面的行程

                   abort = !mService.mController.activityStarting(watchIntent,

                           aInfo.applicationInfo.packageName);

               }......

            //回调对象决定不启动该Activity。在进行Monkey测试时，可设置黑名单，位于

            //黑名单中的Activity将不能启动

             if (abort) {

                .......//通知resultRecord

                return START\_SUCCESS;

             }

           }

      }// if(mMainStack)判断结束

  //创建一个ActivityRecord对象

   ActivityRecordr = new ActivityRecord(mService, this, callerApp, callingUid,

               intent, resolvedType, aInfo, mService.mConfiguration,

               resultRecord, resultWho, requestCode, componentSpecified);

   if(outActivity != null)

        outActivity[0] = r;//保存到输入参数outActivity数组中

   if(mMainStack) {

       //mResumedActivity代表当前界面显示的Activity

       if(mResumedActivity == null

         || mResumedActivity.info.applicationInfo.uid!= callingUid) {

           //检查调用进程是否有权限切换Application，相关知识见下文的解释

           if(!mService.checkAppSwitchAllowedLocked(callingPid, callingUid,

               "Activity start")) {

              PendingActivityLaunch pal = new PendingActivityLaunch();

             //如果调用进程没有权限切换Activity，则只能把这次Activity启动请求保存起来，

             //后续有机会时再启动它

              pal.r = r;

              pal.sourceRecord = sourceRecord;

              ......

              //所有Pending的请求均保存到AMS mPendingActivityLaunches变量中

              mService.mPendingActivityLaunches.add(pal);

              mDismissKeyguardOnNextActivity = false;

              return START\_SWITCHES\_CANCELED;

           }

      }//if(mResumedActivity == null...)判断结束

     if (mService.mDidAppSwitch) {//用于控制app switch，见下文解释

      mService.mAppSwitchesAllowedTime = 0;

     } else{

        mService.mDidAppSwitch = true;

     }

    //启动处于Pending状态的Activity

    mService.doPendingActivityLaunchesLocked(false);

   }// if(mMainStack)判断结束

   //调用startActivityUncheckedLocked函数

   err =startActivityUncheckedLocked(r, sourceRecord,

            grantedUriPermissions, grantedMode, onlyIfNeeded, true);

   ......

   return err;

}

startActivityLocked函数的主要工作包括：

·  处理sourceRecord及resultRecord。其中，sourceRecord表示发起本次请求的Activity，resultRecord表示接收处理结果的Activity（启动一个Activity肯定需要它完成某项事情，当目标Activity将事情成后，就需要告知请求者该事情的处理结果）。在一般情况下，sourceRecord和resultRecord应指向同一个Activity。

·  处理app Switch。如果AMS当前禁止app switch，则只能把本次启动请求保存起来，以待允许app switch时再处理。从代码中可知，AMS在处理本次请求前，会先调用doPendingActivityLaunchesLocked函数，在该函数内部将启动之前因系统禁止app switch而保存的Pending请求。

·  调用startActivityUncheckedLocked处理本次Activity启动请求。

先来看app Switch，它虽然是一个小变量，但是意义重大。

1.  关于resume/stopAppSwitches的介绍

AMS提供了两个函数，用于暂时（注意，是暂时）禁止App切换。为什么会有这种需求呢？因为当某些重要（例如设置账号等）Activity处于前台（即用户当前所见的Activity）时，不希望系统因用户操作之外的原因而切换Activity（例如恰好此时收到来电信号而弹出来电界面）。

先来看stopAppSwitches，代码如下：

[-->ActivityManagerService.java::stopAppSwitches]

public void stopAppSwitches() {

    ......//检查调用进程是否有STOP\_APP\_SWITCHES权限

  synchronized(this) {

      //设置一个超时时间，过了该时间，AMS可以重新切换App（switch app）了

     mAppSwitchesAllowedTime = SystemClock.uptimeMillis()

                   + APP\_SWITCH\_DELAY\_TIME;

     mDidAppSwitch = false;//设置mDidAppSwitch为false

     mHandler.removeMessages(DO\_PENDING\_ACTIVITY\_LAUNCHES\_MSG);

     Message msg =//防止应用进程调用了stop却没调用resume，5秒后处理该消息

           mHandler.obtainMessage(DO\_PENDING\_ACTIVITY\_LAUNCHES\_MSG);

     mHandler.sendMessageDelayed(msg, APP\_SWITCH\_DELAY\_TIME);

   }

}

在以上代码中有两点需要注意：

·  此处控制机制名叫app switch，而不是activity switch。为什么呢？因为如果从受保护的activity中启动另一个activity，那么这个新activity的目的应该是针对同一任务，这次启动就不应该受app switch的制约，反而应该对其大开绿灯。目前，在执行Settings中设置设备策略（DevicePolicy）时就会stopAppSwitch。

·  执行stopAppSwitch后，应用程序应该调resumeAppSwitches以允许app switch，但是为了防止应用程序有意或无意忘记resume app switch，系统设置了一个超时（5秒）消息，过了此超时时间，系统将处理相应的消息，其内部会resume app switch。

再来看resumeAppSwitches函数，代码如下：

[-->ActivityManagerService::resumeAppSwitches]

 public voidresumeAppSwitches() {

    ......//检查调用进程是否有STOP\_APP\_SWITCHES权限

    synchronized(this) {

       mAppSwitchesAllowedTime = 0;

     }

    //注意，系统并不在此函数内启动那些被阻止的Activity

}

在resumeAppSwitches中只设置mAppSwitchesAllowedTime的值为0，它并不处理在stop和resume这段时间内积攒起的Pending请求，那么这些请求是在何时被处理的呢？

·  从前面代码可知，如果在执行resume app switch后，又有新的请求需要处理，则先处理那些pending的请求（调用doPendingActivityLaunchesLocked）。

·  在resumeAppSwitches中并未撤销stopAppSwitches函数中设置的超时消息，所以在处理那条超时消息的过程中，也会处理pending的请求。

在本例中，由于不考虑app switch的情况，那么接下来的工作就是调用startActivityUncheckedLocked函数来处理本次activity的启动请求。此时，我们已经创建了一个ActivityRecord用于保存目标Activity的相关信息。

2.  startActivityUncheckedLocked函数分析

startActivityUncheckedLocked函数很长，但是目的比较简单，即为新创建的ActivityRecord找到一个合适的Task。虽然本例最终的结果是创建一个新的Task，但是该函数的处理逻辑却比较复杂。先看第一段分析。

（1） startActivityUncheckedLocked分析之一

[-->ActivityStack.java::startActivityUncheckedLocked]

final intstartActivityUncheckedLocked(ActivityRecord r,

   ActivityRecord sourceRecord, Uri[] grantedUriPermissions,

    intgrantedMode, boolean onlyIfNeeded, boolean doResume) {

   //在本例中，sourceRecord为null，onlyIfNeeded为false，doResume为true

   finalIntent intent = r.intent;

   final intcallingUid = r.launchedFromUid;

   intlaunchFlags = intent.getFlags();

   //判断是否需要调用因本次Activity启动而被系统移到后台的当前Activity的

  //onUserLeaveHint函数。可阅读SDK文档中关于Activity onUserLeaveHint函数的说明

  mUserLeaving = (launchFlags&Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION) ==0;

   //设置ActivityRecord的delayedResume为true，本例中的doResume为true

   if (!doResume)   r.delayedResume = true;

    //在本例中，notTop为null

    ActivityRecord notTop = (launchFlags&Intent.FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP)

               != 0 ? r : null;

    if(onlyIfNeeded) {....//在本例中，该变量为false，故略去相关代码

    }

   //根据sourceRecord的情况进行对应处理，能理解下面这段if/else的判断语句吗

   if(sourceRecord == null) {

      //如果请求的发起者为空，则当然需要新建一个Task

      if((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) == 0)

         launchFlags |= Intent.FLAG\_ACTIVITY\_NEW\_TASK;

   }else if (sourceRecord.launchMode ==ActivityInfo.LAUNCH\_SINGLE\_INSTANCE){

      //如果sourceRecord单独占一个Instance，则新的Activity必然处于另一个Task中

     launchFlags |= Intent.FLAG\_ACTIVITY\_NEW\_TASK;

   } else if(r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE

               || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK) {

      //如果启动模式设置了singleTask或singleInstance，则也要创建新Task

      launchFlags |= Intent.FLAG\_ACTIVITY\_NEW\_TASK;

   }//if(sourceRecord== null)判断结束

   //如果新Activity和接收结果的Activity不在一个Task中，则不能启动新的Activity

   if(r.resultTo!= null && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

      sendActivityResultLocked(-1,r.resultTo, r.resultWho, r.requestCode,

                                     Activity.RESULT\_CANCELED, null);

      r.resultTo = null;

   }

startActivityUncheckedLocked第一阶段的工作还算简单，主要确定是否需要为新的Activity创建一个Task，即是否设置FLAG\_ACTIVITY\_NEW\_TASK标志。

接下来看下一阶段的工作。

（2） startActivityUncheckedLocked分析之二

[-->ActivityStack.java::startActivityUncheckedLocked]

   booleanaddingToTask = false;

  TaskRecord reuseTask = null;

   if(((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&

         (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK)== 0)

               || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK

               || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {

          if(r.resultTo == null) {

              //搜索mHistory，得到一个ActivityRecord

                ActivityRecord taskTop = r.launchMode !=

                                   ActivityInfo.LAUNCH\_SINGLE\_INSTANCE

                        ?findTaskLocked(intent, r.info)

                        : findActivityLocked(intent,r.info);

               if (taskTop != null ){

                   ......//一堆复杂的逻辑处理，无非就是找到一个合适的Task，然后对应做一些

                  //处理。此处不讨论这段代码，读者可根据工作中的具体情况进行研究

               }

         }//if(r.resultTo == null)判断结束

 }

在本例中，目标Activity首次登场，所以前面的逻辑处理都没有起作用，建议读者根据具体情况分析该段代码。

下面来看startActivityUncheckLocked第三阶段的工作。

（3） startActivityUncheckLocked分析之三

[-->ActivityStack.java::startActivityUncheckLocked]

   if(r.packageName != null) {

        //判断目标Activity是否已经在栈顶，如果是，需要判断是创建一个新的Activity

        //还是调用onNewIntent（singleTop模式的处理）

       ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);

        if (top != null && r.resultTo == null){

            ......//不讨论此段代码

        }//if(top != null...)结束

    } else {

         ......//通知错误

         returnSTART\_CLASS\_NOT\_FOUND;

   }

  //在本例中，肯定需要创建一个Task

   booleannewTask = false;

   booleankeepCurTransition = false;

   if(r.resultTo == null && !addingToTask

               && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

       if(reuseTask == null) {

          mService.mCurTask++;//AMS中保存了当前Task的数量

          if (mService.mCurTask <= 0) mService.mCurTask = 1;

          //为该AactivityRecord设置一个新的TaskRecord

          r.setTask(new TaskRecord(mService.mCurTask, r.info, intent),

                                          null,true);

       }else   r.setTask(reuseTask, reuseTask,true);

      newTask = true;

      //下面这个函数为Android 4.0新增的，用于处理FLAG\_ACTIVITY\_TASK\_ON\_HOME的情况，

      //请阅读SDK文档对Intent的相关说明

      moveHomeToFrontFromLaunchLocked(launchFlags);

   }elseif......//其他处理情况

  //授权控制。在SDK中启动Activity的函数没有授权设置方面的参数。在实际工作中，笔者曾碰

  //到过一个有趣的情况：在开发的一款定制系统中，用浏览器下载了受DRM保护的图片，

  //此时要启动Gallery3D来查看该图片，但是由于为DRM目录设置了读写权限，而Gallery3D

  //并未声明相关权限，结果抛出异常，导致不能浏览该图片。由于startActivity等函数不能设置

  //授权，最终只能修改Gallery3D并为其添加use-permissions项了

  if(grantedUriPermissions != null && callingUid > 0) {

         for(int i=0; i<grantedUriPermissions.length; i++) {

            mService.grantUriPermissionLocked(callingUid, r.packageName,

              grantedUriPermissions[i],grantedMode,

             r.getUriPermissionsLocked());

   }

  mService.grantUriPermissionFromIntentLocked(callingUid, r.packageName,

               intent, r.getUriPermissionsLocked());

   //调用startActivityLocked，此时ActivityRecord和TaskRecord均创建完毕

  startActivityLocked(r, newTask, doResume, keepCurTransition);

   return START\_SUCCESS;

}//startActivityUncheckLocked函数结束

startActivityUncheckLocked的第三阶段工作也比较复杂，不过针对本例，它将创建一个新的TaskRecord，并调用startActivityLocked函数进行处理。

下面我们转战startActivityLocked函数。

（4） startActivityLocked函数分析

[-->ActivityStack.java::startActivityLocked]

private final voidstartActivityLocked(ActivityRecord r, boolean newTask,

           boolean doResume, boolean keepCurTransition) {

   final intNH = mHistory.size();

   intaddPos = -1;

   if(!newTask){//如果不是新Task，则从mHistory中找到对应的ActivityRecord的位置

   ......

   }

   if(addPos < 0)  addPos = NH;

   //否则加到mHistory数组的最后

   mHistory.add(addPos,r);

   //设置ActivityRecord的inHistory变量为true，表示已经加到mHistory数组中了

  r.putInHistory();

  r.frontOfTask = newTask;

   if (NH> 0) {

    //判断是否显示Activity切换动画之类的事情，需要与WindowManagerService交互

  }

   //最终调用resumeTopActivityLocked

   if (doResume) resumeTopActivityLocked(null);//重点分析这个函数

 }

在以上列出的startActivityLocked函数中，略去了一部分逻辑处理，这部分内容和Activity之间的切换动画有关（通过这些动画，使切换过程看起来更加平滑和美观，需和WMS交互）。

**提示**笔者认为，此处将Activity切换和动画处理这两个逻辑揉到一起并不合适，但是似乎也没有更合适的地方来进行该工作了。读者不妨自行研读一下该段代码以加深体会。

（5） startActivityUncheckedLocked总结

说实话，startActivityUncheckedLocked函数的复杂度超乎笔者的想象，光这些函数名就够让人头疼的。但是针对本例而言，相关逻辑的难度还算适中，毕竟这是Activity启动流程中最简单的情况。可用一句话总结本例中startActivityUncheckedLocked函数的功能：创建ActivityRecord和TaskRecord并将ActivityRecord添加到mHistory末尾，然后调用resumeTopActivityLocked启动它。

下面用一节来分析resumeTopActivityLocked函数。

3.  resumeTopActivityLocked函数分析

[-->ActivityStack.java::resumeTopActivityLocked]

 finalboolean resumeTopActivityLocked(ActivityRecord prev) {

   //从mHistory中找到第一个需要启动的ActivityRecord

  ActivityRecord next = topRunningActivityLocked(null);

   finalboolean userLeaving = mUserLeaving;

  mUserLeaving = false;

   if (next== null) {

      //如果mHistory中没有要启动的Activity，则启动Home

      if(mMainStack)    returnmService.startHomeActivityLocked();

   }

   //在本例中，next将是目标Activity

   next.delayedResume= false;

   ......//和WMS交互，略去

   //将该ActivityRecord从下面几个队列中移除

   mStoppingActivities.remove(next);

  mGoingToSleepActivities.remove(next);

  next.sleeping = false;

  mWaitingVisibleActivities.remove(next);

   //如果当前正在中断一个Activity，需先等待那个Activity pause完毕，然后系统会重新

   //调用resumeTopActivityLocked函数以找到下一个要启动的Activity

   if(mPausingActivity != null)  return false;

   /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***请读者注意**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

   //①mResumedActivity指向上一次启动的Activity，也就是当前界面显示的这个Activity

  //在本例中，当前Activity就是Home界面

   if(mResumedActivity != null) {

      //先中断 Home。这种情况放到最后进行分析

      startPausingLocked(userLeaving,false);

      return true;

   }

   //②如果mResumedActivity为空，则一定是系统第一个启动的Activity，读者应能猜测到它就

   //是Home

   ......//如果prev不为空，则需要通知WMS进行与Activity切换相关的工作

   try {

           //通知PKMS修改该Package stop状态，详细信息参考第4章“readLPw的‘佐料’”

           //一节的说明

           AppGlobals.getPackageManager().setPackageStoppedState(

                   next.packageName, false);

    }......

   if(prev!= null){

   ......//还是和WMS有关，通知它停止绘画

   }

  if(next.app != null && next.app.thread != null) {

   //如果该ActivityRecord已有对应的进程存在，则只需要重启Activity。就本例而言，

   //此进程还不存在，所以要先创建一个应用进程

  }  else {

           //第一次启动

           if (!next.hasBeenLaunched) {

               next.hasBeenLaunched = true;

           } else {

               ......//通知WMS显示启动界面

           }

       //调用另外一个startSpecificActivityLocked函数

      startSpecificActivityLocked(next, true, true);

    }

    returntrue;

}

resumeTopActivityLocked函数中有两个非常重要的关键点：

·  如果mResumedActivity不为空，则需要先暂停（pause）这个Activity。由代码中的注释可知，mResumedActivity代表上一次启动的（即当前正显示的）Activity。现在要启动一个新的Activity，须先停止当前Activity，这部分工作由startPausingLocked函数完成。

·  那么，mResumedActivity什么时候为空呢？当然是在启动全系统第一个Activity时，即启动Home界面的时候。除此之外，该值都不会为空。

先分析第二个关键点，即mResumedActivity为null的情况选择分析此种情况的原因是：如果先分析startPausingLocked，则后续分析会牵扯三个进程，即当前Activity所在进程、AMS所在进程及目标进程，分析的难度相当大。

好了，继续我们的分析。resumeTopActivityLocked最后将调用另外一个startSpecificActivityLocked，该函数将真正创建一个应用进程。

（1） startSpecificActivityLocked分析

[-->ActivityStack.java::startSpecificActivityLocked]

private final voidstartSpecificActivityLocked(ActivityRecord r,

           boolean andResume, boolean checkConfig) {

   //从AMS中查询是否已经存在满足要求的进程（根据processName和uid来查找）

  //在本例中，查询结果应该为null

  ProcessRecord app = mService.getProcessRecordLocked(r.processName,

               r.info.applicationInfo.uid);

   //设置启动时间等信息

   if(r.launchTime == 0) {

        r.launchTime = SystemClock.uptimeMillis();

       if(mInitialStartTime == 0)  mInitialStartTime = r.launchTime;

   } else if(mInitialStartTime == 0) {

           mInitialStartTime = SystemClock.uptimeMillis();

   }

   //如果该进程存在并已经向AMS注册（例如之前在该进程中启动了其他Activity）

   if (app!= null && app.thread != null) {

       try {

           app.addPackage(r.info.packageName);

           //通知该进程中的启动目标Activity

           realStartActivityLocked(r, app, andResume, checkConfig);

           return;

         }......

    }

    //如果该进程不存在，则需要调用AMS的startProcessLocked创建一个应用进程

    mService.startProcessLocked(r.processName, r.info.applicationInfo,

              true, 0,"activity",r.intent.getComponent(), false);

}

来看AMS的startProcessLocked函数，它将创建一个新的应用进程。

（2） startProcessLocked分析

[-->ActivityManagerService.java::startProcessLocked]

final ProcessRecord startProcessLocked(StringprocessName,

           ApplicationInfo info, boolean knownToBeDead, int intentFlags,

           String hostingType, ComponentName hostingName,

            boolean allowWhileBooting) {

   //根据processName和uid寻找是否已经存在ProcessRecord

   ProcessRecordapp = getProcessRecordLocked(processName, info.uid);

   if (app!= null && app.pid > 0) {

       ......//处理相关情况

   }

    StringhostingNameStr = hostingName != null

               ? hostingName.flattenToShortString() : null;

   //①处理FLAG\_FROM\_BACKGROUND标志，见下文解释

   if((intentFlags&Intent.FLAG\_FROM\_BACKGROUND) != 0) {

       if(mBadProcesses.get(info.processName, info.uid) != null)

            return null;

     } else {

        mProcessCrashTimes.remove(info.processName,info.uid);

        if(mBadProcesses.get(info.processName, info.uid) != null) {

            mBadProcesses.remove(info.processName, info.uid);

            if (app != null)    app.bad =false;

         }

   }

    if (app== null) {

        //创建一个ProcessRecord，并保存到mProcessNames中。注意，此时还没有创建实际进程

        app= newProcessRecordLocked(null, info, processName);

       mProcessNames.put(processName, info.uid, app);

    }else   app.addPackage(info.packageName);

       ......

    //②调用另外一个startProcessLocked函数

   startProcessLocked(app, hostingType, hostingNameStr);

     return(app.pid != 0) ? app : null;

 }

在以上代码中列出两个关键点，其中第一点和FLAG\_FROM\_BACKGROUND有关，相关知识点如下：

·  FLAG\_FROM\_BACKGROUND标识发起这次启动的Task属于后台任务。很显然，手机中没有界面供用户操作位于后台Task中的Activity。如果没有设置该标志，那么这次启动请求就是由前台Task因某种原因而触发的（例如用户单击某个按钮）。

·  如果一个应用进程在1分钟内连续崩溃超过2次，则AMS会将其ProcessRecord加入所谓的mBadProcesses中。一个应用崩溃后，系统会弹出一个警告框以提醒用户。但是，如果一个后台Task启动了一个“BadProcess”，然后该Process崩溃，结果弹出一个警告框，那么用户就会觉得很奇怪：“为什么突然弹出一个框？”因此，此处将禁止后台Task启动“Bad Process”。如果用户主动选择启动（例如单击一个按钮），则不能禁止该操作，并且要把应用进程从mBadProcesses中移除，以给它们“重新做人”的机会。当然，要是该应用每次启动时都会崩溃，而且用户不停地去启动，那该用户可能是位测试工作者。

**提示**这其实是一种安全机制，防止不健全的程序不断启动可能会崩溃的组件，但是这种机制并不限制用户的行为。

下面来看第二个关键点，即另一个startProcessLocked函数，其代码如下：

[-->ActivityManagerService.java::startProcessLocked]

private final voidstartProcessLocked(ProcessRecord app,

                            String hostingType, StringhostingNameStr) {

   if(app.pid > 0 && app.pid != MY\_PID) {

       synchronized (mPidsSelfLocked) {

        mPidsSelfLocked.remove(app.pid);

        mHandler.removeMessages(PROC\_START\_TIMEOUT\_MSG, app);

       }

      app.pid = 0;

  }

   //mProcessesOnHold用于保存那些在系统还没有准备好就提前请求启动的ProcessRecord

   mProcessesOnHold.remove(app);

   updateCpuStats();

  System.arraycopy(mProcDeaths, 0, mProcDeaths, 1, mProcDeaths.length-1);

   mProcDeaths[0] = 0;

    try {

         intuid = app.info.uid;

        int[] gids = null;

           try {//从PKMS中查询该进程所属的gid

               gids = mContext.getPackageManager().getPackageGids(

                        app.info.packageName);

           }......

       ......//工厂测试

      intdebugFlags = 0;

      if((app.info.flags & ApplicationInfo.FLAG\_DEBUGGABLE) != 0) {

                debugFlags |=Zygote.DEBUG\_ENABLE\_DEBUGGER;

              debugFlags |= Zygote.DEBUG\_ENABLE\_CHECKJNI;

         }......//设置其他一些debugFlags

     //发送消息给Zygote，它将派生一个子进程，该子进程执行ActivityThread的main函数

    //注意，我们传递给Zygote的参数并没有包含任何与Activity相关的信息。现在仅仅启动

    //一个应用进程

    Process.ProcessStartResult startResult =

                   Process.start("android.app.ActivityThread",

                   app.processName, uid, uid, gids, debugFlags,

                   app.info.targetSdkVersion, null);

     //电量统计项

    BatteryStatsImpl bs = app.batteryStats.getBatteryStats();

    synchronized (bs) {

          if(bs.isOnBattery()) app.batteryStats.incStartsLocked();

     }

      //如果该进程为persisitent，则需要通知Watchdog，实际上processStarted内部只

      //关心刚才创建的进程是不是com.android.phone

     if(app.persistent) {

         Watchdog.getInstance().processStarted(app.processName,

                                    startResult.pid);

     }

    app.pid= startResult.pid;

   app.usingWrapper = startResult.usingWrapper;

   app.removed = false;

   synchronized (mPidsSelfLocked) {

          //以pid为key，将代表该进程的ProcessRecord对象加入到mPidsSelfLocked中保管

         this.mPidsSelfLocked.put(startResult.pid, app);

          //发送一个超时消息，如果这个新创建的应用进程10秒内没有和AMS交互，则可断定

         //该应用进程启动失败

         Message msg = mHandler.obtainMessage(PROC\_START\_TIMEOUT\_MSG);

         msg.obj = app;

          //正常的超时时间为10秒。不过如果该应用进程通过valgrind加载，则延长到300秒

        //valgrind是Linux平台上一款检查内存泄露的程序，被加载的应用将在它的环境中工作，

         //这项工作需耗费较长时间。读者可查询valgrind的用法

          mHandler.sendMessageDelayed(msg,startResult.usingWrapper

                        ?PROC\_START\_TIMEOUT\_WITH\_WRAPPER : PROC\_START\_TIMEOUT);

    }......

 }

startProcessLocked通过发送消息给Zygote以派生一个应用进程[④]，读者仔细研究所发消息的内容，大概会发现此处并未设置和Activity相关的信息，也就是说，该进程启动后，将完全不知道自己要干什么，怎么办？下面就此进行分析。

4.  startActivity分析之半程总结

很抱歉，我们现在还处于startActivity分析之旅的中间点，即使越过了很多险滩恶途，一路走来还是发觉有点艰难。此处用图6-14来记录半程中的各个关键点。

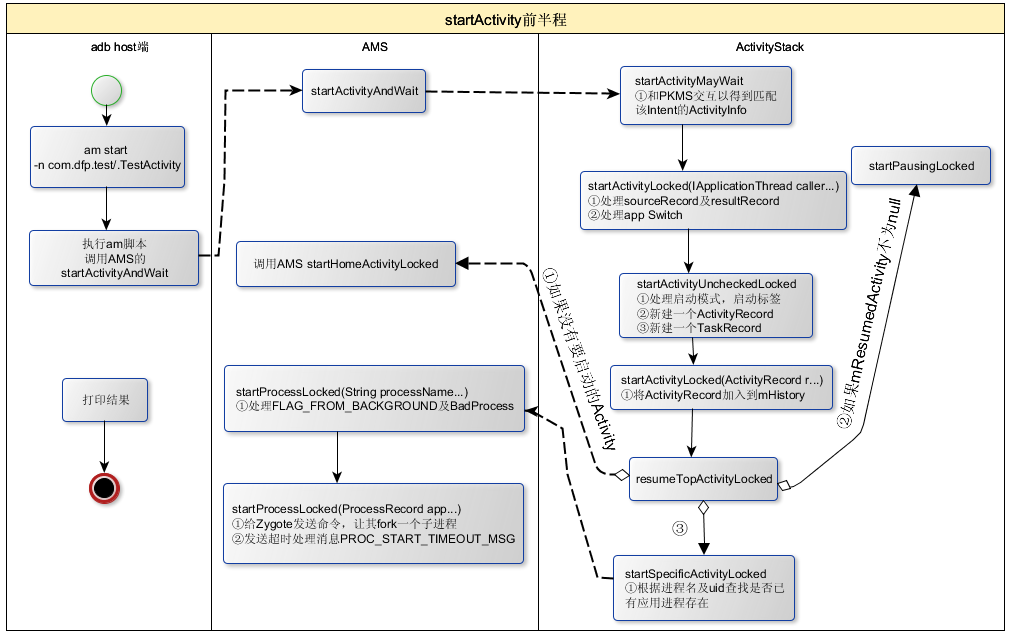


图6-14  startActivity半程总结

图6-14列出了针对本例的调用顺序，其中对每个函数的大体功能也做了简单描述。

**注意**图6-14中的调用顺序及功能说明只是针对本例而言的。读者以后可结合具体情况再深入研究其中的内容。

5.  应用进程的创建及初始化

如前所述，应用进程的入口是ActivityThread的main函数，它是在主线程中执行的，其代码如下：

[-->ActivityThread.java::main]

public static void main(String[] args) {

   SamplingProfilerIntegration.start();

   //和调试及strictMode有关

  CloseGuard.setEnabled(false);

   //设置进程名为"<pre-initialized>"

  Process.setArgV0("<pre-initialized>");

   //准备主线程消息循环

  Looper.prepareMainLooper();

   if(sMainThreadHandler == null)

      sMainThreadHandler = new Handler();

   //创建一个ActivityThread对象

  ActivityThread thread = new ActivityThread();

   //①调用attach函数，注意其参数值为false

  thread.attach(false);

  Looper.loop(); //进入主线程消息循环

   throw newRuntimeException("Main thread loop unexpectedly exited");

}

在main函数内部将创建一个消息循环Loop，接着调用ActivityThread的attach函数，最终将主线程加入消息循环。

我们在分析AMS的setSystemProcess时曾分析过ActivityThread的attach函数，那时传入的参数值为true。现在来看设置其为false的情况：

[-->ActivityThread.java::attach]

private void attach(boolean system) {

  sThreadLocal.set(this);

  mSystemThread = system;

   if(!system) {

      ViewRootImpl.addFirstDrawHandler(new Runnable() {

          public void run() {

             ensureJitEnabled();

           }

         });

       //设置在DDMS中看到的本进程的名字为"<pre-initialized>"

      android.ddm.DdmHandleAppName.setAppName("<pre-initialized>");

       //设置RuntimeInit的mApplicationObject参数，后续会介绍RuntimeInit类

      RuntimeInit.setApplicationObject(mAppThread.asBinder());

       //获取和AMS交互的Binder客户端

      IActivityManager mgr = ActivityManagerNative.getDefault();

       try {

            //①调用AMS的attachApplication，mAppThread为ApplicationThread类型，

            //它是应用进程和AMS交互的接口

             mgr.attachApplication(mAppThread);

         }......

   } else......// system process的处理

   ViewRootImpl.addConfigCallback(newComponentCallbacks2()

   {.......//添加回调函数});

}

我们知道，AMS创建一个应用进程后，会设置一个超时时间（一般是10秒）。如果超过这个时间，应用进程还没有和AMS交互，则断定该进程创建失败。所以，应用进程启动后，需要尽快和AMS交互，即调用AMS的attachApplication函数。在该函数内部将调用attachApplicationLocked，所以此处直接分析attachApplicationLocked，先看其第一阶段的工作。

（1） attachApplicationLocked分析之一

[-->ActivityManagerService.java::attachApplicationLocked]

private final booleanattachApplicationLocked(IApplicationThread thread,

           int pid) {//此pid代表调用进程的pid

   ProcessRecord app;

    if (pid != MY\_PID && pid >= 0) {

        synchronized (mPidsSelfLocked) {

           app = mPidsSelfLocked.get(pid);//根据pid查找对应的ProcessRecord对象

        }

    }else    app = null;

    /\*

    如果该应用进程由AMS启动，则它一定在AMS中有对应的ProcessRecord，读者可回顾前面创建

    应用进程的代码：AMS先创建了一个ProcessRecord对象，然后才发命令给Zygote。

    如果此处app为null，表示AMS没有该进程的记录，故需要“杀死”它

   \*/

    if (app== null) {

       if(pid > 0 && pid != MY\_PID) //如果pid大于零，且不是SystemServer进程，则

           //Quietly（即不打印任何输出）”杀死”process

           Process.killProcessQuiet(pid);

       else{

           //调用ApplicationThread的scheduleExit函数。应用进程完成处理工作后

           //将退出运行

           //为何不像上面一样直接杀死它呢？可查阅linux pid相关的知识并自行解答

          thread.scheduleExit();

        }

      returnfalse;

   }

   /\*

     判断app的thread是否为空，如果不为空，则表示该ProcessRecord对象还未和一个

     应用进程绑定。注意，app是根据pid查找到的，如果旧进程没有被杀死，系统则不会重用

     该pid。为什么此处会出现ProcessRecord thread不为空的情况呢？见下面代码的注释说明

   \*/

   if(app.thread != null)  handleAppDiedLocked(app, true, true);

   StringprocessName = app.processName;

   try {

          /\*

          创建一个应用进程讣告接收对象。当应用进程退出时，该对象的binderDied将被调

          用。这样，AMS就能做相应处理。binderDied函数将在另外一个线程中执行，其内部也会

          调用handleAppDiedLocked。假如用户在binderDied被调用之前又启动一个进程，

          那么就会出现以上代码中app.thread不为null的情况。这是多线程环境中常出现的

          情况，不熟悉多线程编程的读者要仔细体会。

          \*/

         AppDeathRecipient adr = new AppDeathRecipient(pp, pid, thread);

         thread.asBinder().linkToDeath(adr, 0);

         app.deathRecipient = adr;

    }......

   //设置该进程的调度优先级和oom\_adj等成员

   app.thread= thread;

  app.curAdj = app.setAdj = -100;

  app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;

  app.setSchedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

  app.forcingToForeground = null;

  app.foregroundServices = false;

  app.hasShownUi = false;

  app.debugging = false;

   //启动成功，从消息队列中撤销PROC\_START\_TIMEOUT\_MSG消息

  mHandler.removeMessages(PROC\_START\_TIMEOUT\_MSG, app);

attachApplicationLocked第一阶段的工作比较简单：

·  设置代表该应用进程的ProcessRecrod对象的一些成员变量，例如用于和应用进程交互的thread对象、进程调度优先级及oom\_adj的值等。

·  从消息队列中撤销PROC\_START\_TIMEOUT\_MSG。

至此，该进程启动成功，但是这一阶段的工作仅针对进程本身（如设置调度优先级，oom\_adj等），还没有涉及和Activity启动相关的内容，这部分工作将在第二阶段完成。

（2） attachApplicationLocked分析之二

[-->ActivityManagerService.java::attachApplicationLocked]

   ......

   //SystemServer早就启动完毕，所以normalMode为true

   booleannormalMode = mProcessesReady || isAllowedWhileBooting(app.info);

   /\*

    我们在6.2.3的标题1中分析过generateApplicationProvidersLocked函数，

    在该函数内部将查询（根据进程名，uid确定）PKMS以获取需运行在该进程中的ContentProvider

   \*/

   Listproviders = normalMode ? generateApplicationProvidersLocked(app) : null;

   try {

         int testMode = IApplicationThread.DEBUG\_OFF;

          if(mDebugApp != null && mDebugApp.equals(processName)) {

               ......//处理debug选项

          }

          ......//处理Profile

          boolean isRestrictedBackupMode = false;

          ......//

          //dex化对应的apk包

          ensurePackageDexOpt(app.instrumentationInfo!= null ?

                app.instrumentationInfo.packageName : app.info.packageName);

         //如果设置了Instrumentation类，该类所在的Package也需要dex化

         if(app.instrumentationClass != null)

              ensurePackageDexOpt(app.instrumentationClass.getPackageName());

          ApplicationInfo appInfo =app.instrumentationInfo != null

                             ? app.instrumentationInfo :app.info;

         //查询该Application使用的CompatibiliyInfo

         app.compat =compatibilityInfoForPackageLocked(appInfo);

         if (profileFd != null) //用于记录性能文件

               profileFd = profileFd.dup();

           //①通过ApplicationThread和应用进程交互，调用其bindApplication函数

            thread.bindApplication(processName,appInfo, providers,

                   app.instrumentationClass, profileFile, profileFd,

                    profileAutoStop,app.instrumentationArguments,

                    app.instrumentationWatcher,testMode,

                   isRestrictedBackupMode || !normalMode, app.persistent,

                   mConfiguration, app.compat, getCommonServicesLocked(),

                   mCoreSettingsObserver.getCoreSettingsLocked());

           //updateLruProcessLocked函数以后再作分析

           updateLruProcessLocked(app,false, true);

           //记录两个时间

           app.lastRequestedGc= app.lastLowMemory = SystemClock.uptimeMillis();

   }......//try结束

..//从mProcessesOnHold和mPersistentStartingProcesses中删除相关信息

   mPersistentStartingProcesses.remove(app);

  mProcessesOnHold.remove(app);

由以上代码可知，第二阶段的工作主要是为调用ApplicationThread的bindApplication做准备，将在后面的章节中分析该函数的具体内容。此处先来看它的原型。

/\*

   正如我们在前面分析时提到的，刚创建的这个进程并不知道自己的历史使命是什么，甚至连自己的

   进程名都不知道，只能设为"<pre-initialized>"。其实，Android应用进程的历史使命是

   AMS在其启动后才赋予它的，这一点和我们理解的一般意义上的进程不太一样。根据之前的介绍，   Android的组件应该运行在Android运行环境中。从OS角度来说，该运行环境需要和一个进程绑定。

   所以，创建应用进程这一步只是创建了一个能运行Android运行环境的容器，而我们的工作实际上

   还远未结束。

   bindApplication的功能就是创建并初始化位于该进程中的Android运行环境

\*/

public final void bindApplication(

       StringprocessName,//进程名，一般是package名

      ApplicationInfo appInfo,//该进程对应的ApplicationInfo

       List<ProviderInfo> providers,//在该APackage中声明的Provider信息

       ComponentName instrumentationName,//和instrumentation有关

       //下面3个参数和性能统计有关

       StringprofileFile,

      ParcelFileDescriptor profileFd, boolean autoStopProfiler,

      //这两个和Instrumentation有关，在本例中，这几个参数暂时都没有作用

      Bundle instrumentationArgs,

       IInstrumentationWatcherinstrumentationWatcher,

       intdebugMode,//调试模式

       boolean isRestrictedBackupMode,

       boolean persistent,//该进程是否是persist

       Configuration config,//当前的配置信息，如屏幕大小和语言等

       CompatibilityInfocompatInfo,//兼容信息

       //AMS将常用的Service信息传递给应用进程，目前传递的Service信息只有PKMS、

       //WMS及AlarmManagerService。读者可参看AMS getCommonServicesLocked函数

       Map<String,IBinder> services,

       BundlecoreSettings)//核心配置参数，目前仅有“long\_press”值

对bindApplication的原型分析就到此为止，再来看attachApplicationLocked最后一阶段的工作。

（3） attachApplicationLocked分析之三

[-->ActivityManagerService.java::attachApplicationLocked]

   booleanbadApp = false;

   booleandidSomething = false;

   /\*

   至此，应用进程已经准备好了Android运行环境，下面这句调用代码将返回ActivityStack中

   第一个需要运行的ActivityRecord。由于多线程的原因，难道能保证得到的hr就是我们的目标

   Activity吗？

   \*/

  ActivityRecord hr = mMainStack.topRunningActivityLocked(null);

  if (hr !=null && normalMode) {

       //需要根据processName和uid等确定该Activity是否运行与目标进程有关

       if(hr.app == null && app.info.uid == hr.info.applicationInfo.uid

            && processName.equals(hr.processName)) {

            try {

               //调用AS的realStartActivityLocked启动该Activity，最后两个参数为true

                if (mMainStack.realStartActivityLocked(hr, app, true, true)) {

                      didSomething = true;

                 }

            } catch (Exception e) {

                  badApp = true; //设置badApp为true

            }

      } else{

         //如果hr和目标进程无关，则调用ensureActivitiesVisibleLocked函数处理它

        mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);

       }

 }// if (hr!= null && normalMode)判断结束

   //mPendingServices存储那些因目标进程还未启动而处于等待状态的ServiceRecord

   if(!badApp && mPendingServices.size() > 0) {

       ServiceRecord sr = null;

        try{

           for (int i=0; i<mPendingServices.size(); i++) {

                sr = mPendingServices.get(i);

                //和Activity不一样的是，如果Service不属于目标进程，则暂不处理

                if (app.info.uid != sr.appInfo.uid

                     ||!processName.equals(sr.processName)) continue;//继续循环

                   //该Service将运行在目标进程中，所以从mPendingService中移除它

                   mPendingServices.remove(i);

                   i--;

                   //处理此service的启动，以后再作分析

                   realStartServiceLocked(sr, app);

                   didSomething = true;//设置该值为true

               }

           }

      }......

   ......//启动等待的BroadcastReceiver

   ......//启动等待的BackupAgent，相关代码类似Service的启动

  if(badApp) {

   //如果以上几个组件启动有错误，则设置badApp为true。此处将调用handleAppDiedLocked

   //进行处理。该函数我们以后再作分析

   handleAppDiedLocked(app, false, true);

    returnfalse;

  }

   /\*

   调整进程的oom\_adj值。didSomething表示在以上流程中是否启动了Acivity或其他组件。

   如果启动了任一组件，则didSomething为true。读者以后会知道，这里的启动只是向

   应用进程发出对应的指令，客户端进程是否成功处理还是未知数。基于这种考虑，所以此处不宜

   马上调节进程的oom\_adj。

   读者可简单地把oom\_adj看做一种优先级。如果一个应用进程没有运行任何组件，那么当内存

   出现不足时，该进程是最先被系统杀死的。反之，如果一个进程运行的组件越多，那么它就越不易被

   系统杀死以回收内存。updateOomAdjLocked就是根据该进程中组件的情况对应调节进程的

   oom\_adj值的。

  \*/

   if(!didSomething)  updateOomAdjLocked();

   returntrue;

 }

attachApplicationLocked第三阶段的工作就是通知应用进程启动Activity和Service等组件，其中用于启动Activity的函数是ActivityStack realStartActivityLocked。

此处先来分析应用进程的bindApplication，该函数将为应用进程绑定一个Application。

**提示**还记得AMS中System Context执行的两次init吗？第二次init的功能就是将Context和对应的Application绑定在一起。

（4） ApplicationThread的bindApplication分析

bindApplication在ApplicationThread中的实现，其代码如下：

[-->ActivityThread.java::bindApplication]

public final void bindApplication(......) {

   if(services != null)//保存AMS传递过来的系统Service信息

       ServiceManager.initServiceCache(services);

    //向主线程消息队列添加SET\_CORE\_SETTINGS消息

   setCoreSettings(coreSettings);

     //创建一个AppBindData对象，其实就是用来存储一些参数

    AppBindData data = new AppBindData();

    data.processName = processName;

    data.appInfo = appInfo;

    data.providers = providers;

    data.instrumentationName = instrumentationName;

    ......//将AMS传过来的参数保存到AppBindData中

     //向主线程发送H.BIND\_APPLICATION消息

    queueOrSendMessage(H.BIND\_APPLICATION, data);

 }

由以上代码可知，ApplicationThread接收到来自AMS的指令后，均会将指令中的参数封装到一个数据结构中，然后通过发送消息的方式转交给主线程去处理。BIND\_APPLICATION最终将由handleBindApplication函数处理。该函数并不复杂，但是其中有些点是值得关注的，这些点主要是初始化应用进程的一些参数。handleBindApplication函数的代码如下：

[-->ActivityThread.java::handleBindApplication]

private void handleBindApplication(AppBindDatadata) {

   mBoundApplication = data;

   mConfiguration = new Configuration(data.config);

   mCompatConfiguration = new Configuration(data.config);

    //初始化性能统计对象

   mProfiler = new Profiler();

   mProfiler.profileFile = data.initProfileFile;

   mProfiler.profileFd = data.initProfileFd;

   mProfiler.autoStopProfiler = data.initAutoStopProfiler;

    //设置进程名。从此，之前那个默默无名的进程终于有了自己的名字

   Process.setArgV0(data.processName);

   android.ddm.DdmHandleAppName.setAppName(data.processName);

    if(data.persistent) {

       //对于persistent的进程，在低内存设备上，不允许其使用硬件加速显示

       Display display =

                 WindowManagerImpl.getDefault().getDefaultDisplay();

       //当内存大于512MB，或者屏幕尺寸大于1024\*600，可以使用硬件加速

       if(!ActivityManager.isHighEndGfx(display))

           HardwareRenderer.disable(false);

     }

    //启动性能统计

     if(mProfiler.profileFd != null)   mProfiler.startProfiling();

   //如果目标SDK版本小于12，则设置AsyncTask使用pool executor，否则使用

   //serializedexecutor。这些executor涉及Java Concurrent类，对此不熟悉的读者

   //请自行学习和研究。

   if(data.appInfo.targetSdkVersion <= 12)

           AsyncTask.setDefaultExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR);

    //设置timezone

   TimeZone.setDefault(null);

    //设置语言

    Locale.setDefault(data.config.locale);

     //设置资源及兼容模式

    applyConfigurationToResourcesLocked(data.config, data.compatInfo);

    applyCompatConfiguration();

      //根据传递过来的ApplicationInfo创建一个对应的LoadApk对象

     data.info = getPackageInfoNoCheck(data.appInfo, data.compatInfo);

     //对于系统APK，如果当前系统为userdebug/eng版，则需要记录log信息到dropbox的日志记录

     if((data.appInfo.flags &

            (ApplicationInfo.FLAG\_SYSTEM |

             ApplicationInfo.FLAG\_UPDATED\_SYSTEM\_APP)) != 0) {

           StrictMode.conditionallyEnableDebugLogging();

    }

    /\*

    如目标SDK版本大于9，则不允许在主线程使用网络操作（如Socketconnect等），否则抛出

    NetworkOnMainThreadException，这么做的目的是防止应用程序在主线程中因网络操作执行

    时间过长而造成用户体验下降。说实话，没有必要进行这种限制，在主线程中是否网络操作

    是应用的事情。再说，Socket也可作为进程间通信的手段，在这种情况下，网络操作耗时很短。

    作为系统，不应该设置这种限制。另外，Goolge可以提供一些开发指南或规范来指导开发者，

    而不应如此蛮横地强加限制。

    \*/

    if (data.appInfo.targetSdkVersion> 9)

        StrictMode.enableDeathOnNetwork();

    //如果没有设置屏幕密度，则为Bitmap设置默认的屏幕密度

   if((data.appInfo.flags

               &ApplicationInfo.FLAG\_SUPPORTS\_SCREEN\_DENSITIES) == 0)

    Bitmap.setDefaultDensity(DisplayMetrics.DENSITY\_DEFAULT);

   if(data.debugMode != IApplicationThread.DEBUG\_OFF){

    ......//调试模式相关处理

   }

   IBinder b= ServiceManager.getService(Context.CONNECTIVITY\_SERVICE);

  IConnectivityManager service =

                        IConnectivityManager.Stub.asInterface(b);

  try {

       //设置Http代理信息

       ProxyPropertiesproxyProperties = service.getProxy();

      Proxy.setHttpProxySystemProperty(proxyProperties);

   } catch(RemoteException e) {}

   if(data.instrumentationName != null){

        //在正常情况下，此条件不满足

   } else {

      //创建Instrumentation对象，在正常情况都再这个条件下执行

     mInstrumentation = new Instrumentation();

   }

   //如果Package中声明了FLAG\_LARGE\_HEAP，则可跳过虚拟机的内存限制，放心使用内存

   if((data.appInfo.flags&ApplicationInfo.FLAG\_LARGE\_HEAP) != 0)

        dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();

   //创建一个Application，data.info为LoadedApk类型，在其内部会通过Java反射机制

   //创建一个在该APK AndroidManifest.xml中声明的Application对象

   Applicationapp = data.info.makeApplication(

                                     data.restrictedBackupMode, null);

 //mInitialApplication保存该进程中第一个创建的Application

  mInitialApplication = app;

   //安装本Package中携带的ContentProvider

   if(!data.restrictedBackupMode){

      List<ProviderInfo> providers = data.providers;

        if(providers != null) {

               //installContentProviders我们已经分析过了

               installContentProviders(app, providers);

               mH.sendEmptyMessageDelayed(H.ENABLE\_JIT, 10\*1000);

           }

  }

   //调用Application的onCreate函数，做一些初始工作

  mInstrumentation.callApplicationOnCreate(app);

}

由以上代码可知，bindApplication函数将设置一些初始化参数，其中最重要的有：

·  创建一个Application对象，该对象是本进程中运行的第一个Application。

·  如果该Application有ContentProvider，则应安装它们。

**提示**从以上代码可知，ContentProvider的创建就在bindApplication函数中，其时机早于其他组件的创建。

（5） 应用进程的创建及初始化总结

本节从应用进程的入口函数main开始，分析了应用进程和AMS之间的两次重要交互，它们分别是：

·  在应用进程启动后，需要尽快调用AMS的attachApplication函数，该函数是这个刚呱呱坠地的应用进程第一次和AMS交互。此时的它还默默“无名”，连一个确定的进程名都没有。不过没关系，attachApplication函数将根据创建该应用进程之前所保存的ProcessRecord为其准备一切“手续”。

·  attachApplication准备好一切后，将调用应用进程的bindApplication函数，在该函数内部将发消息给主线程，最终该消息由handleBindApplication处理。handleBindApplication将为该进程设置进程名，初始化一些策略和参数信息等。另外，它还创建一个Application对象。同时，如果该Application声明了ContentProvider，还需要为该进程安装ContentProvider。

**提示**这个流程有点类似生孩子，一般生之前需要到医院去登记，生完后又需去注册户口，如此这般，这个孩子才会在社会有合法的身份。

6.  ActivityStack realStartActivityLocked分析

如前所述，AMS调用完bindApplication后，将通过realStartActivityLocked启动Activity。在此之前，要创建完应用进程并初始化Android运行环境（除此之外，连ContentProvider都安装好了）。

[-->ActivityStack.java::realStartActivityLocked]

//注意，在本例中该函数的最后两个参数的值都为true

final booleanrealStartActivityLocked(ActivityRecord r, ProcessRecord app,

       boolean andResume, boolean checkConfig)   throws RemoteException {

   r.startFreezingScreenLocked(app, 0);

    mService.mWindowManager.setAppVisibility(r,true);

    if(checkConfig) {

       ......//处理Config发生变化的情况

      mService.updateConfigurationLocked(config, r, false);

    }

    r.app =app;

   app.waitingToKill = null;

    //将ActivityRecord加到ProcessRecord的activities中保存

    int idx= app.activities.indexOf(r);

    if (idx< 0)   app.activities.add(r);

    //更新进程的调度优先级等，以后再分析该函数

    mService.updateLruProcessLocked(app, true, true);

     try {

        List<ResultInfo> results = null;

        List<Intent> newIntents = null;

         if(andResume) {

               results = r.results;

               newIntents = r.newIntents;

           }

          if(r.isHomeActivity)  mService.mHomeProcess = app;

        //看看是否有dex对应Package的需要

         mService.ensurePackageDexOpt(

                   r.intent.getComponent().getPackageName());

        r.sleeping = false;

        r.forceNewConfig = false;

       ......

          //①通知应用进程启动Activity

           app.thread. scheduleLaunchActivity (new Intent(r.intent), r,

                   System.identityHashCode(r), r.info, mService.mConfiguration,

                   r.compat, r.icicle, results, newIntents, !andResume,

                   mService.isNextTransitionForward(), profileFile, profileFd,

                    profileAutoStop);

           if ((app.info.flags&ApplicationInfo.FLAG\_CANT\_SAVE\_STATE) != 0) {

               ......//处理heavy-weight的情况

               }

           }

     } ......//try结束

    r.launchFailed = false;

     ......

     if(andResume) {

         r.state = ActivityState.RESUMED;

         r.stopped = false;

        mResumedActivity = r;//设置mResumedActivity为目标Activity

        r.task.touchActiveTime();

         //添加该任务到近期任务列表中

         if(mMainStack)  mService.addRecentTaskLocked(r.task);

         //②关键函数，见下文分析

        completeResumeLocked(r);

         //如果在这些过程中，用户按了Power键，怎么办？

        checkReadyForSleepLocked();

        r.icicle = null;

        r.haveState = false;

    }......

    //启动系统设置向导Activity，当系统更新或初次使用时需要进行配置

    if(mMainStack)  mService.startSetupActivityLocked();

    returntrue;

 }

在以上代码中有两个关键函数，分别是：scheduleLaunchActivity和completeResumeLocked。其中，scheduleLaunchActivity用于和应用进程交互，通知它启动目标Activity。而completeResumeLocked将继续AMS的处理流程。先来看第一个关键函数。

（1） scheduleLaunchActivity函数分析

[-->ActivityThread.java::scheduleLaunchActivity]

public final void scheduleLaunchActivity(Intentintent, IBinder token, int ident,

     ActivityInfo info, Configuration curConfig,CompatibilityInfo compatInfo,

     Bundlestate, List<ResultInfo> pendingResults,

    List<Intent> pendingNewIntents, boolean notResumed, booleanisForward,

     StringprofileName, ParcelFileDescriptor profileFd,

     booleanautoStopProfiler) {

  ActivityClientRecord r = new ActivityClientRecord();

   ......//保存AMS发送过来的参数信息

   //向主线程发送消息，该消息的处理在handleLaunchActivity中进行

  queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);

 }

[-->ActivityThread.java::handleMessage]

public void handleMessage(Message msg) {

  switch(msg.what) {

       case LAUNCH\_ACTIVITY: {

        ActivityClientRecord r = (ActivityClientRecord)msg.obj;

        //根据ApplicationInfo得到对应的PackageInfo

        r.packageInfo = getPackageInfoNoCheck(

                           r.activityInfo.applicationInfo, r.compatInfo);

         //调用handleLaunchActivity处理

        handleLaunchActivity(r, null);

       }break;

......

}

[-->ActivityThread.java::handleLaunchActivity]

private voidhandleLaunchActivity(ActivityClientRecord r,

                             Intent customIntent){

   unscheduleGcIdler();

   if (r.profileFd != null) {......//略去}

  handleConfigurationChanged(null, null);

   /\*

   ①创建Activity，通过Java反射机制创建目标Activity，将在内部完成Activity生命周期

   的前两步，即调用其onCreate和onStart函数。至此，我们的目标com.dfp.test.TestActivity

   创建完毕

   \*/

   Activitya = performLaunchActivity(r, customIntent);

   if (a !=null) {

     r.createdConfig = new Configuration(mConfiguration);

      BundleoldState = r.state;

      //②调用handleResumeActivity，其内部有个关键点，见下文分析

     handleResumeActivity(r.token, false, r.isForward);

      if(!r.activity.mFinished && r.startsNotResumed) {

         .......//

.       r.paused = true;

       }else {

            //如果启动错误，通知AMS

           ActivityManagerNative.getDefault()

                    .finishActivity(r.token,Activity.RESULT\_CANCELED, null);

      }

  }

handleLaunchActivity的工作包括：

·  首先调用performLaunchActivity，该在函数内部通过Java反射机制创建目标Activity，然后调用它的onCreate及onStart函数。

·  调用handleResumeActivity，会在其内部调用目标Activity的onResume函数。除此之外，handleResumeActivity还完成了一件很重要的事情，见下面的代码：

[-->ActivityThread.java::handleResumeActivity]

final void handleResumeActivity(IBinder token,boolean clearHide,

                                     booleanisForward) {

 unscheduleGcIdler();

  //内部调用目标Activity的onResume函数

 ActivityClientRecord r = performResumeActivity(token, clearHide);

  if (r !=null) {

   finalActivity a = r.activity;

   final int forwardBit = isForward ?

          WindowManager.LayoutParams.SOFT\_INPUT\_IS\_FORWARD\_NAVIGATION : 0;

   ......

   if(!r.onlyLocalRequest) {

       //将上面完成onResume的Activity保存到mNewActivities中

      r.nextIdle = mNewActivities;

      mNewActivities = r;

      //①向消息队列中添加一个Idler对象

     Looper.myQueue().addIdleHandler(new Idler());

   }

   r.onlyLocalRequest = false;

   ......

 }

根据第2章对MessageQueue的分析，当消息队列中没有其他要处理的消息时，将处理以上代码中通过addIdleHandler添加的Idler对象，也就是说，Idler对象的优先级最低，这是不是说它的工作不重要呢？非也。至少在handleResumeActivity函数中添加的这个Idler并不不简单，其代码如下：

[-->ActivityThread.java::Idler]

private class Idler implements MessageQueue.IdleHandler{

    publicfinal boolean queueIdle() {

   ActivityClientRecord a = mNewActivities;

    booleanstopProfiling = false;

    ......

   if (a !=null) {

    mNewActivities = null;

    IActivityManager am = ActivityManagerNative.getDefault();

    ActivityClientRecord prev;

    do {

       if(a.activity != null && !a.activity.mFinished) {

          //调用AMS的activityIdle

          am.activityIdle(a.token, a.createdConfig, stopProfiling);

          a.createdConfig = null;

      }

     prev =a;

     a =a.nextIdle;

    prev.nextIdle = null;

    } while(a != null); //do循环结束

  }//if(a!=null)判断结束

   ......

   ensureJitEnabled();

    returnfalse;

   }// queueIdle函数结束

 }

由以上代码可知，Idler将为那些已经完成onResume的Activity调用AMS的activityIdle函数。该函数是Activity成功创建并启动的流程中与AMS交互的最后一步。虽然对应用进程来说，Idler处理的优先级最低，但AMS似乎不这么认为，因为它还设置了超时等待，以处理应用进程没有及时调用activityIdle的情况。这个超时等待即由realStartActivityLocked中最后一个关键点completeResumeLocked函数设置。

（2） completeResumeLocked函数分析

[-->ActivityStack.java::completeResumeLocked]

private final voidcompleteResumeLocked(ActivityRecord next) {

   next.idle = false;

   next.results = null;

   next.newIntents = null;

   //发送一个超时处理消息，默认为10秒。IDLE\_TIMEOUT\_MSG就是针对acitivityIdle函数的

    Messagemsg = mHandler.obtainMessage(IDLE\_TIMEOUT\_MSG);

    msg.obj= next;

   mHandler.sendMessageDelayed(msg, IDLE\_TIMEOUT);

     //通知AMS

     if(mMainStack)   mService.reportResumedActivityLocked(next);

     ......//略去其他逻辑的代码

}

由以上代码可知，AMS给了应用进程10秒的时间，希望它在10秒内调用activityIdle函数。这个时间不算长，和前面AMS等待应用进程启动的超时时间一样。所以，笔者有些困惑，为什么要把这么重要的操作放到idler中去做。

下面来看activityIdle函数，在其内部将调用ActivityStack activityIdleInternal。

（3） activityIdleInternal函数分析

[-->ActivityStack.java::activityIdleInternal]

final ActivityRecord activityIdleInternal(IBindertoken, boolean fromTimeout,

                                      Configuration config) {

   /\*

   如果应用进程在超时时间内调用了activityIdleInternal函数，则fromTimeout为false

   否则，一旦超时，在IDLE\_TIMEOUT\_MSG的消息处理中也会调用该函数，并设置fromTimeout

   为true

   \*/

   ActivityRecord res = null;

  ArrayList<ActivityRecord> stops = null;

  ArrayList<ActivityRecord> finishes = null;

  ArrayList<ActivityRecord> thumbnails = null;

   int NS =0;

   int NF =0;

   int NT =0;

  IApplicationThread sendThumbnail = null;

   booleanbooting = false;

   booleanenableScreen = false;

  synchronized (mService) {

     //从消息队列中撤销IDLE\_TIMEOUT\_MSG

     if(token != null)   mHandler.removeMessages(IDLE\_TIMEOUT\_MSG, token);

      int index= indexOfTokenLocked(token);

      if(index >= 0) {

        ActivityRecord r = mHistory.get(index);

         res =r;

        //注意，只有fromTimeout为true，才会走执行下面的条件语句

        if(fromTimeout) reportActivityLaunchedLocked(fromTimeout, r, -1, -1);

        if(config != null)  r.configuration =config;

        /\*

        mLaunchingActivity是一个WakeLock，它能防止在操作Activity过程中掉电，同时

        这个WakeLock又不能长时间使用，否则有可能耗费过多电量。所以，系统设置了一个超时

        处理消息LAUNCH\_TIMEOUT\_MSG，超时时间为10秒。一旦目标Activity启动成功，

        就需要需要释放 WakeLock

        \*/

        if(mResumedActivity == r && mLaunchingActivity.isHeld()) {

            mHandler.removeMessages(LAUNCH\_TIMEOUT\_MSG);

            mLaunchingActivity.release();

         }

       r.idle = true;

       mService.scheduleAppGcsLocked();

       ......

       ensureActivitiesVisibleLocked(null, 0);

     if(mMainStack) {

         if(!mService.mBooted) {

            mService.mBooted = true;

             enableScreen = true;

          }

       }//if (mMainStack)判断结束

   } else if(fromTimeout) {//注意，只有fromTimeout为true，才会走下面的case

       reportActivityLaunchedLocked(fromTimeout, null, -1, -1);

  }

  /\*

   ①processStoppingActivitiesLocked函数返回那些因本次Activity启动而

   被暂停（paused）的Activity

  \*/

  stops =processStoppingActivitiesLocked(true);

 ......

   for (i=0;i<NS; i++) {

      ActivityRecord r = (ActivityRecord)stops.get(i);

       synchronized (mService) {

      //如果这些Acitivity 处于finishing状态，则通知它们执行Destroy操作，最终它们

     //的onDestroy函数会被调用

       if(r.finishing) finishCurrentActivityLocked(r, FINISH\_IMMEDIATELY);

       else  //否则将通知它们执行stop操作，最终Activity的onStop被调用

         stopActivityLocked(r);

       }//synchronized结束

  }//for循环结束

   ......//处理等待结束的Activities

   //发送ACTION\_BOOT\_COMPLETED广播

   if(booting)  mService.finishBooting();

   ......

   returnres;

 }

在activityIdleInternal中有一个非常重要的关键点，即处理那些因为本次Activity启动而被暂停的Activity。有两种情况需考虑：

·  如果被暂停的Activity处于finishing状态（例如Activity在其onStop中调用了finish函数），则调用finishCurrentActivityLocked。

·  否则，要调用stopActivityLocked处理暂停的Activity。

此处涉及除AMS和目标进程外的第三个进程，即被切换到后台的那个进程。不过至此，我们的目标Activity终于正式登上了历史舞台。

**提示**本例的分析结束了吗？没有。因为am设置了-W选项，所以其实我们还在startActivityAndWait函数中等待结果。ActivityStack中有两个函数能够触发AMS notifyAll，一个是reportActivityLaunchedLocked，另一个是reportActivityVisibleLocked。前面介绍的activityInternal函数只在fromTimeout为true时才会调用reportActivityLaunchedLocked，而本例中fromTimeout为false，如何是好？该问题的解答非常复杂，姑且先一语带过：当Activity显示出来时，其在AMS中对应ActivityRecord对象的windowVisible函数将被调用，其内部会触发reportActivityLaunchedLocked函数，这样我们的startActivityAndWait才能被唤醒。

7.  startActivity分析之后半程总结

总结startActivity后半部分的流程，主要涉及目标进程和AMS的交互，如图6-15所示。

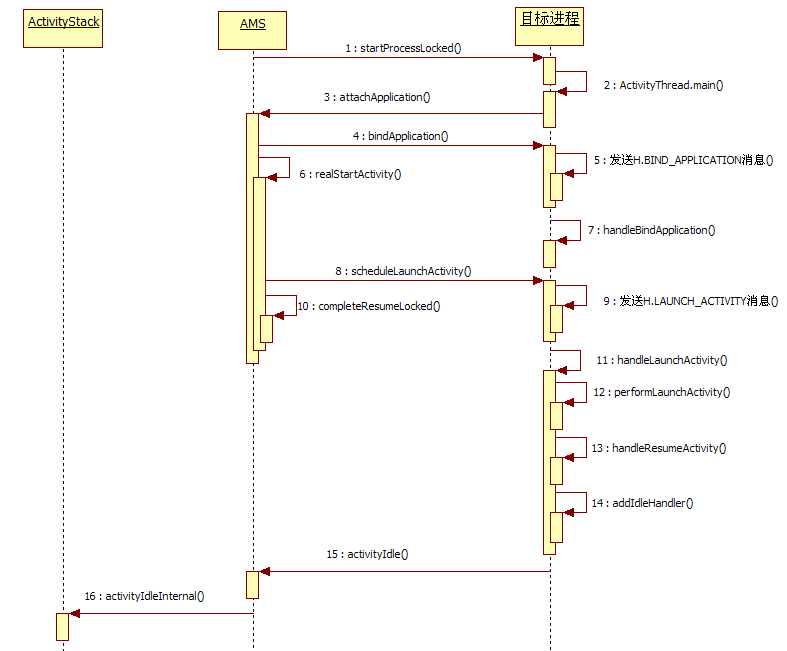


图6-15  startActivity后半程总结

图6-15中涉及16个重要函数调用，而且这仅是startActivity后半部分的调用流程，可见整个流程有多么复杂！

8. startPausingLocked函数分析

现在我们分析图6-14中的startPausingLocked分支。根据前面的介绍，当启动一个新Activity时，系统将先行处理当前的Activity，即调用startPausingLocked函数来暂停当前Activity。

（1） startPausingLocked分析

[-->ActivityStack.java::startPausingLocked]

private final void startPausingLocked(booleanuserLeaving, boolean uiSleeping) {

   //mResumedActivity保存当前正显示的Activity，

  ActivityRecord prev = mResumedActivity;

  mResumedActivity = null;

   //设置mPausingActivity为当前Activity

  mPausingActivity = prev;

  mLastPausedActivity = prev;

  prev.state = ActivityState.PAUSING;//设置状态为PAUSING

  prev.task.touchActiveTime();

   ......

   if(prev.app != null && prev.app.thread != null) {

     try {

        //①调用当前Activity所在进程的schedulePauseActivity函数

        prev.app.thread.schedulePauseActivity(prev,prev.finishing,

                                userLeaving,prev.configChangeFlags);

       if(mMainStack) mService.updateUsageStats(prev, false);

     } ......//catch分支

    }......//else分支

   if(!mService.mSleeping && !mService.mShuttingDown) {

        //获取WakeLock，以防止在Activity切换过程中掉电

       mLaunchingActivity.acquire();

        if(!mHandler.hasMessages(LAUNCH\_TIMEOUT\_MSG)) {

            Message msg = mHandler.obtainMessage(LAUNCH\_TIMEOUT\_MSG);

            mHandler.sendMessageDelayed(msg, LAUNCH\_TIMEOUT);

        }

   }

    if(mPausingActivity != null) {

      //暂停输入事件派发

      if(!uiSleeping) prev.pauseKeyDispatchingLocked();

     //设置PAUSE超时，时间为500毫秒，这个时间相对较短

     Message msg = mHandler.obtainMessage(PAUSE\_TIMEOUT\_MSG);

     msg.obj = prev;

     mHandler.sendMessageDelayed(msg, PAUSE\_TIMEOUT);

    }......//else分支

 }

startPausingLocked将调用应用进程的schedulePauseActivity函数，并设置500毫秒的超时时间，所以应用进程需尽快完成相关处理。和scheduleLaunchActivity一样，schedulePauseActivity将向ActivityThread主线程发送PAUSE\_ACTIVITY消息，最终该消息由handlePauseActivity来处理。

（2） handlePauseActivity分析

[-->ActivityThread.java::handlePauseActivity]

private void handlePauseActivity(IBinder token,boolean finished,

                               boolean userLeaving, int configChanges){

  //当Activity处于finishing状态时，finished参数为true，不过在本例中该值为false

  ActivityClientRecord r = mActivities.get(token);

   if (r !=null) {

      //调用Activity的onUserLeaving函数，

      if(userLeaving) performUserLeavingActivity(r);

      r.activity.mConfigChangeFlags |=configChanges;

      //调用Activity的onPause函数

     performPauseActivity(token, finished, r.isPreHoneycomb());

      ......

      try {

             //调用AMS的activityPaused函数

             ActivityManagerNative.getDefault().activityPaused(token);

         }......

    }

 }

[-->ActivityManagerService.java::activityPaused]

public final void activityPaused(IBinder token) {

   ......

   mMainStack.activityPaused(token, false);

}

[-->ActivityStack.java::activityPaused]

final void activityPaused(IBinder token, booleantimeout) {

 ActivityRecord r = null;

  synchronized (mService) {

   int index= indexOfTokenLocked(token);

   if (index>= 0) {

        r =mHistory.get(index);

         //从消息队列中撤销PAUSE\_TIMEOUT\_MSG消息

        mHandler.removeMessages(PAUSE\_TIMEOUT\_MSG, r);

         if(mPausingActivity == r) {

            r.state = ActivityState.PAUSED;//设置ActivityRecord的状态

            completePauseLocked();//完成本次Pause操作

        }......

   }

 }

（3） completePauseLocked分析

[-->ActivityStack.java::completePauseLocked]

private final void completePauseLocked() {

  ActivityRecord prev = mPausingActivity;

   if (prev!= null) {

     if(prev.finishing) {

          prev = finishCurrentActivityLocked(prev, FINISH\_AFTER\_VISIBLE);

      } elseif (prev.app != null) {

        if(prev.configDestroy) {

             destroyActivityLocked(prev, true, false);

         } else {

          //①将刚才被暂停的Activity保存到mStoppingActivities中

          mStoppingActivities.add(prev);

          if(mStoppingActivities.size() > 3) {

            //如果被暂停的Activity超过3个，则发送IDLE\_NOW\_MSG消息，该消息最终

           //由我们前面介绍的activeIdleInternal处理

             scheduleIdleLocked();

            }

         }

      //设置mPausingActivity为null，这是图6-14②、③分支的分割点

       mPausingActivity = null;

    }

     //②resumeTopActivityLocked将启动目标Activity

    if(!mService.isSleeping()) resumeTopActivityLocked(prev);

   ......

}

就本例而言，以上代码还算简单，最后还是通过resumeTopActivityLocked来启动目标Activity。当然，由于之前已经设置了mPausingActivity为null，所以最终会走到图6-14中③的分支。

（4） stopActivityLocked分析

根据前面的介绍，此次目标Activity将走完onCreate、onStart和onResume流程，但是被暂停的Activity才刚走完onPause流程，那么它的onStop什么时候调用呢？

答案就在activityIdelInternal中，它将为mStoppingActivities中的成员调用stopActivityLocked函数。

[-->ActivityStack.java::stopActivityLocked]

 privatefinal void stopActivityLocked(ActivityRecord r) {

   if((r.intent.getFlags()&Intent.FLAG\_ACTIVITY\_NO\_HISTORY) != 0

           || (r.info.flags&ActivityInfo.FLAG\_NO\_HISTORY) != 0) {

       if(!r.finishing) {

           requestFinishActivityLocked(r, Activity.RESULT\_CANCELED, null,

                       "no-history");

         }

     } elseif (r.app != null && r.app.thread != null) {

         try {

         r.stopped = false;

          //设置STOPPING状态，并调用对应的scheduleStopActivity函数

          r.state = ActivityState.STOPPING;

          r.app.thread.scheduleStopActivity(r, r.visible,

                     r.configChangeFlags);

    }......

  }

对应进程的scheduleStopActivity函数将根据visible的情况，向主线程消息循环发送H. STOP\_ACTIVITY\_HIDE或H. STOP\_ACTIVITY\_SHOW消息。不论哪种情况，最终都由handleStopActivity来处理。

[-->ActivityThread.java::handleStopActivity]

private void handleStopActivity(IBinder token,boolean show, int configChanges) {

 ActivityClientRecord r = mActivities.get(token);

 r.activity.mConfigChangeFlags |= configChanges;

  StopInfoinfo = new StopInfo();

  //调用Activity的onStop函数

 performStopActivityInner(r, info, show, true);

 ......

  try { //调用AMS的activityStopped函数

        ActivityManagerNative.getDefault().activityStopped(

               r.token, r.state, info.thumbnail, info.description);

   }

}

AMS没有为stop设置超时消息处理。严格来说，还是有超时限制的，只是这个超时处理与activityIdleInternal结合起来了。

（5） startPausingLocked总结

总结startPausingLocked流程，如图6-16所示。

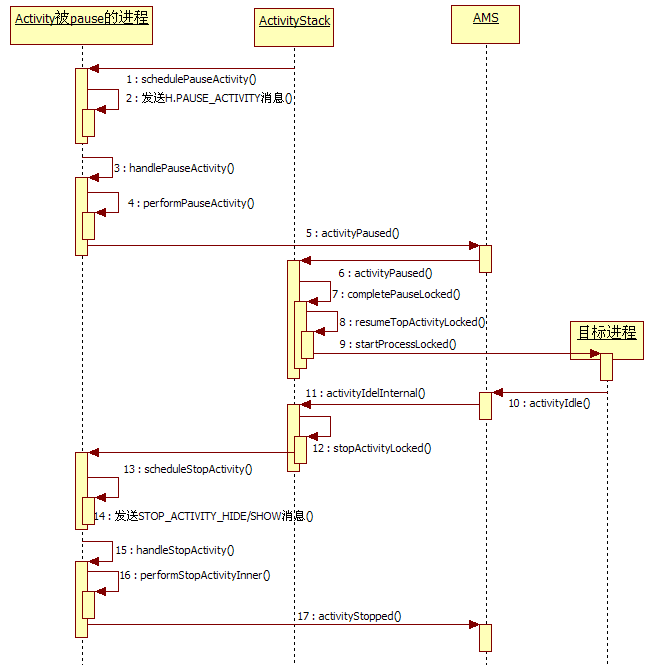


图6-16  startPausingActivity流程总结

图6-16比较简单，读者最好结合代码再把流程走一遍，以加深理解。

9.  startActivity总结

Activity的启动就介绍到这里。这一路分析下来，相信读者也和笔者一样觉得此行绝不轻松。先回顾一下此次旅程：

·   行程的起点是am。am是Android中很重要的程序，读者务必要掌握它的用法。我们利用am start命令，发起本次目标Activity的启动请求。

·  接下来进入ActivityManagerService和ActivityStack这两个核心类。对于启动Activity来说，这段行程又可分细分为两个阶段：第一阶段的主要工作就是根据启动模式和启动标志找到或创建ActivityRecord及对应的TaskRecord；第二阶段工作就是处理Activity启动或切换相关的工作。

·  首先讨论了AMS直接创建目标进程并运行Activity的流程，其中涉及目标进程的创建，在目标进程中Android运行环境的初始化，目标Activity的创建以及触发onCreate、onStart及onResume等其生命周期中重要函数调用等相关知识点。

·  接着又讨论了AMS先pause当前Activity，然后再创建目标进程并运行Activity的流程。其中牵扯到两个应用进程和AMS的交互，其难度之大可见一斑。

读者在阅读本节时，务必要区分此旅程中两个阶段工作的重点：其一是找到合适的ActivityRecord和TaskRecord；其二是调度相关进程进行Activity切换。在SDK文档中，介绍最为详细的是第一阶段中系统的处理策略，例如启动模式、启动标志的作用等。第二阶段工作其实是与Android组件调度相关的工作。SDK文档只是针对单个Activity进行生命周期方面的介绍。

坦诚地说，这次旅程略过不少逻辑情况。原因有二，一方面受限于精力和篇幅，另方面是作为调度核心类，和AMS相关的代码及处理逻辑非常复杂，而且其间还夹杂了与WMS的交互逻辑，使复杂度更甚。再者，笔者个人感觉这部分代码绝谈不上高效、严谨和美观，甚至有些丑陋（在分析它们的过程中，远没有研究Audio、Surface时那种畅快淋漓的感觉）。

此处列出几个供读者深入研究的点：

·  各种启动模式、启动标志的处理流程。

·  Configuration发生变化时Activity的处理，以及在Activity中对状态保存及恢复的处理流程。

·  Activity生命周期各个阶段的转换及相关处理。Android 2.3以后新增的与Fragment的生命周期相关的转换及处理。

**建议**在研究代码前，先仔细阅读SDK文档相关内容，以获取必要的感性认识，否则直接看代码很容易迷失方向。

6.4  Broadcast和BroadcastReceiver分析

Broadcast，汉语意思为“广播”。它是Android平台中的一种通知机制。从广义来说，它是一种进程间通信的手段。有广播，就对应有广播接收者。Android中四大组件之一的BroadcastReceiver即代表广播接收者。目前，系统提供两种方式来声明一个广播接收者。

·  在AndroidManifest.xml中声明<receiver>标签。在应用程序运行时，系统会利用Java反射机制构造一个广播接收者实例。本书将这种广播接收者称为静态注册者或静态接收者。

·  在应用程序运行过程中，可调用Context提供的registerReceiver函数注册一个广播接收者实例。本书将这种广播接收者称为动态注册者或动态接收者。与之相对应，当应用程序不再需要监听广播时（例如当应用程序退到后台时），则要调用unregisterReceiver函数撤销之前注册的BroadcastReceiver实例。

当系统将广播派发给对应的广播接收者时，广播接收者的onReceive函数会被调用。在此函数中，可对该广播进行相应处理。

另外，Android定义了三种不同类型的广播发送方式，它们分别是：

·  普通广播发送方式，由sendBroadcast及相关函数发送。以工作中的场景为例，当程序员们正埋头工作之时，如果有人大喊一声“吃午饭去”，前刻还在专心编码的人即作鸟兽散。这种方式即为普通广播发送方式，所有对“吃午饭”感兴趣的接收者都会响应。

·  串行广播发送方式，即ordered广播，由sendOrdedBroadcast及相关函数发送。在该类型方式下，按接收者的优先级将广播一个个地派发给接收者。只有等这一个接收者处理完毕，系统才将该广播派发给下一个接收者。其中，任意一个接收者都可以中止后续的派发流程。还是以工作中的场景为例：经常有项目经理（PM）深夜组织一帮人跟踪bug的状态。PM看见一个bug，问某程序员，“这个bug你能改吗?”如果得到的答案是“暂时不会”或“暂时没时间”，他会将目光转向下一个神情轻松者，直到找到一个担当者为止。这种方式即为ordered广播发送方式，很明显，它的特点是“一个一个来”。

·  Sticky广播发送方式，由sendStickyBroadcast及相关函数发送。Sticky的意思是“粘”，其背后有一个很重要的考虑。我们举个例子：假设某广播发送者每5秒发送一次携带自己状态信息的广播，此时某个应用进程注册了一个动态接收者来监听该广播，那么该接收者的OnReceive函数何时被调用呢？在正常情况下需要等这一轮的5秒周期结束后才调用（因为发送者在本周期结束后会主动再发一个广播）。而在Sticky模式下，系统将马上派发该广播给刚注册的接收者。注意，这个广播是系统发送的，其中存储的是上一次广播发送者的状态信息。也就是说，在Sticky模式下，广播接收者能立即得到广播发送者的信息，而不用等到这一轮周期结束。其实就是系统会保存Sticky的广播[⑤]，当有新广播接收者来注册时，系统就把Sticky广播发给它。

以上我们对广播及广播接收者做了一些简单介绍，读者也可参考SDK文档中的相关说明来加强理解。

下面将以动态广播接收者为例，分析Android对广播的处理流程。

6.4.1  registerReceiver流程分析

1.  ContextImpl registerReceiver分析

registerReceiver函数用于注册一个动态广播接收者，该函数在Context.java中声明。根据本章前面对Context家族的介绍（参考图6-3），其功能最终将通过ContextImpl类的registerReceiver函数来完成，可直接去看ContextImpl是如何实现此函数的。在SDK中一共定义了两个同名的registerReceiver函数，其代码如下：

[-->ContextImpl.java::registerReceiver]

/\*

  在SDK中输出该函数，这也是最常用的函数。当广播到来时，BroadcastReceiver对象的onReceive

  函数将在主线程中被调用

\*/

public Intent registerReceiver(BroadcastReceiverreceiver, IntentFilter filter) {

    returnregisterReceiver(receiver, filter, null, null);

 }

/\*

   功能和前面类似，但增加了两个参数，分别是broadcastPermission和scheduler，作用有

   两个：

   其一：对广播者的权限增加了控制，只有拥有相应权限的广播者发出的广播才能被此接收者接收

   其二：BroadcastReceiver对象的onReceiver函数可调度到scheduler所在的线程中执行

\*/

 publicIntent registerReceiver(BroadcastReceiver receiver,

       IntentFilterfilter, String broadcastPermission, Handler scheduler) {

/\*

 注意，下面所调用函数的最后一个参数为getOuterContext的返回值。前面曾说过，ContextImpl为Context家族中真正干活的对象，而它对外的代理人可以是Application和Activity等，

getOuterContext就返回这个对外代理人。一般在Activity中调用registerReceiver函数，故此处getOuterContext返回的对外代理人的类型就是Activity。

\*/

     returnregisterReceiverInternal(receiver, filter, broadcastPermission,

               scheduler, getOuterContext());

}

殊途同归，最终的功能由registerReceiverInternal来完成，其代码如下：

[-->ContextImpl.java::registerReceiverInternal]

 privateIntent registerReceiverInternal(BroadcastReceiver receiver,

      IntentFilter filter, String broadcastPermission, Handler scheduler,

      Context context) {

   IIntentReceiver rd = null;

   if(receiver != null) {

        //①准备一个IIntentReceiver对象

        if(mPackageInfo != null && context != null) {

          //如果没有设置scheduler，则默认使用主线程的Handler

          if (scheduler == null)   scheduler= mMainThread.getHandler();

          //通过getReceiverDispatcher函数得到一个IIntentReceiver类型的对象

          rd = mPackageInfo.getReceiverDispatcher(

             receiver, context, scheduler, mMainThread.getInstrumentation(),

             true);

        } else {

          if (scheduler == null)  scheduler= mMainThread.getHandler();

          //直接创建LoadedApk.ReceiverDispatcher对象

          rd = new LoadedApk.ReceiverDispatcher(receiver, context, scheduler,

                   null, true).getIIntentReceiver();

         }//if (mPackageInfo != null && context != null)结束

     }// if(receiver != null)结束

  try {

        //②调用AMS的registerReceiver函数

       return ActivityManagerNative.getDefault().registerReceiver(

                     mMainThread.getApplicationThread(),mBasePackageName,

                     rd, filter,broadcastPermission);

        } ......

 }

以上代码列出了两个关键点：其一是准备一个IIntentReceiver对象；其二是调用AMS的registerReceiver函数。

先来看IIntentReceiver，它是一个Interface，图6-17列出了和它相关的成员图谱。

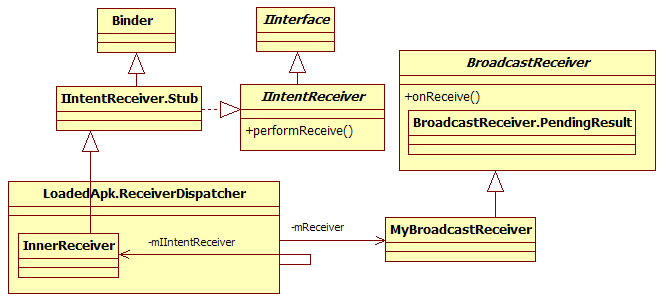


图6-17  IIntentReceiver相关成员示意图

由图6-17可知：

·  BroadcastReceiver内部有一个PendingResult类。该类是Android 2.3以后新增的，用于异步处理广播消息。例如，当BroadcastReceiver收到一个广播时，其onReceive函数将被调用。一般都是在该函数中直接处理该广播。不过，当该广播处理比较耗时，还可采用异步的方式进行处理，即先调用BroadcastReceiver的goAsync函数得到一个PendingResult对象，然后将该对象放到工作线程中去处理，这样onReceive就可以立即返回而不至于耽误太长时间（这一点对于onReceive函数被主线程调用的情况尤为有用）。在工作线程处理完这条广播后，需调用PendingResult的finish函数来完成整个广播的处理流程。

·  广播由AMS发出，而接收及处理工作却在另外一个进程中进行，整个过程一定涉及进程间通信。在图6-17中，虽然在BroadcastReceiver中定义了一个广播接收者，但是它与Binder没有有任何关系，故其并不直接参与进程间通信。与之相反，IIntentReceiver接口则和Binder有密切关系，故可推测广播的接收是由IIntentReceiver接口来完成的。确实，在整个流程中，首先接收到来自AMS的广播的将是该接口的Bn端，即LoadedApk.ReceiverDispather的内部类InnerReceiver。

接收广播的处理将放到本节最后再来分析，下面先来看AMS 的registerReceiver函数。

2.  AMS的registerReceiver分析

AMS的registerReceiver函数比较简单，但是由于其中将出现一些新的变量类型和成员，因此接下来按分两部分进行分析。

（1） registerReceiver分析之一

registerReceiver的返回值是一个Intent，它指向一个匹配过滤条件（由filter参数指明）的Sticky Intent。即使有多个符合条件的Intent，也只返回一个。

[-->ActivityManagerService.java::registerReceiver]

public Intent registerReceiver(IApplicationThreadcaller, String callerPackage,

           IIntentReceiver receiver, IntentFilter filter, String permission) {

  synchronized(this) {

      ProcessRecord callerApp = null;

      if(caller != null) {

         callerApp = getRecordForAppLocked(caller);

         ....... //如果callerApp为null，则抛出异常，即系统不允许未登记照册的进程注册

         //动态广播接收者

          //检查调用进程是否有callerPackage的信息，如果没有，也抛异常

          if(callerApp.info.uid != Process.SYSTEM\_UID &&

             !callerApp.pkgList.contains(callerPackage)){

                throw new SecurityException(......);

             }

       }......//if(caller != null)判断结束

     List allSticky = null;

      //下面这段代码的功能是从系统中所有Sticky Intent中查询匹配IntentFilter的Intent，

     //匹配的Intent保存在allSticky中

     Iterator actions = filter.actionsIterator();

      if(actions != null) {

        while (actions.hasNext()) {

              String action = (String)actions.next();

              allSticky = getStickiesLocked(action, filter, allSticky);

           }

       } ......

     //如果存在sticky的Intent，则选取第一个Intent作为本函数的返回值

     Intentsticky = allSticky != null ? (Intent)allSticky.get(0) : null;

     //如果没有设置接收者，则直接返回sticky 的intent

     if(receiver == null)  return sticky;

    //新的数据类型ReceiverList及mRegisteredReceivers成员变量，见下文的解释

  //receiver.asBinder将返回IIntentReceiver的Bp端

    ReceiverList rl

               = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());

     //如果是首次调用，则此处rl的值将为null

     if (rl== null) {

       rl =new ReceiverList(this, callerApp, Binder.getCallingPid(),

                     Binder.getCallingUid(),receiver);

       if (rl.app != null) {

           rl.app.receivers.add(rl);

       }else {

         try {

                 //监听广播接收者所在进程的死亡消息

                 receiver.asBinder().linkToDeath(rl, 0);

             }......

          rl.linkedToDeath = true;

        }// if(rl.app != null)判断结束

           //将rl保存到mRegisterReceivers中

          mRegisteredReceivers.put(receiver.asBinder(), rl);

     }

    //新建一个BroadcastFilter对象

    BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage,

                                                        permission);

    rl.add(bf);//将其保存到rl中

     // mReceiverResolver成员变量，见下文解释

    mReceiverResolver.addFilter(bf);

以上代码的流程倒是很简单，不过其中出现的几个成员变量和数据类型却严重阻碍了我们的思维活动。先解决它们，BroadcastFilter及相关成员变量如图6-18所示。

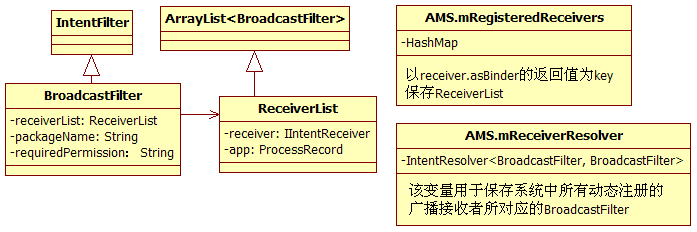


图6-18  BroadcastFilter及相关成员变量

结合代码，对图6-18中各数据类型和成员变量的作用及关系的解释如下：

·  在AMS中，BroadcastReceiver的过滤条件由BroadcastFilter表示，该类从IntentFilter派生。由于一个BroadcastReceiver可设置多个过滤条件（即多次为同一个BroadcastReceiver对象调用registerReceiver函数以设置不同的过滤条件），故AMS使用ReceiverList（从ArrayList<BroadcastFilter>派生）这种数据类型来表达这种一对多的关系。

·  ReceiverList除了能存储多个BroadcastFilter外，还应该有成员指向某一个具体BroadcastReceiver，否则如何知道到底是哪个BroadcastReceiver设置的过滤条件呢？前面说过，BroadcastReceiver接收广播是通过IIntentReceiver接口进行的，故ReceiverList中有receiver成员变量指向IIntentReceiver。

·  AMS提供mRegisterReceivers用于保存IIntentReceiver和对应ReceiverList的关系。此外，AMS还提供mReceiverResolver变量用于存储所有动态注册的BroadcastReceiver所设置的过滤条件。

清楚这些成员变量和数据类型之间的关系后，接着来分析registerReceiver第二阶段的工作。

（2） registerReceiver分析之二

[-->ActivityManagerService.java::registerReceiver]

    //如果allSticky不为空，则表示有Sticky的Intent，需要立即调度广播发送

     if(allSticky != null) {

        ArrayList receivers = new ArrayList();

         receivers.add(bf);

         intN = allSticky.size();

         for(int i=0; i<N; i++) {

            Intent intent = (Intent)allSticky.get(i);

            //为每一个需要发送的广播创建一个BroadcastRecord（暂称之为广播记录）对象

            BroadcastRecord r = new BroadcastRecord(intent, null,

                            null, -1, -1, null,receivers, null, 0, null, null,

                            false, true, true);

            //如果mParallelBroadcasts当前没有成员，则需要触发AMS发送广播

            if (mParallelBroadcasts.size() == 0)

                 scheduleBroadcastsLocked();//向AMS发送BROADCAST\_INTENT\_MSG消息

             //所有非ordered广播记录都保存在mParallelBroadcasts中

             mParallelBroadcasts.add(r);

          }//for循环结束

    }//if (allSticky != null)判断结束

    returnsticky;

   }//synchronized结束

}

这一阶段的工作用一句话就能说清楚：为每一个满足IntentFilter的Sticky的intent创建一个BroadcastRecord对象，并将其保存到mParllelBroadcasts数组中，最后，根据情况调度AMS发送广播。

从上边的描述中可以看出，一旦存在满足条件的Sticky的Intent，系统需要尽快调度广播发送。说到这里，想和读者分享一种在实际工作中碰到的情况。

我们注册了一个BroadcastReceiver，用于接收USB的连接状态。在注册完后，它的onReceiver函数很快就会被调用。当时笔者的一些同事认为是注册操作触发USB模块又发送了一次广播，却又感到有些困惑，USB模块应该根据USB的状态变化去触发广播发送，而不应理会广播接收者的注册操作，这到底是怎么一回事呢？

相信读者现在应该轻松解决这种困惑了吧？对于Sticky的广播，一旦有接收者注册，系统会马上将该广播传递给它们。

和ProcessRecord及ActivityRecord类似，AMS定义了一个BroadcastRecord数据结构，用于存储和广播相关的信息，同时还有两个成员变量，它们作用和关系如图6-19所示。

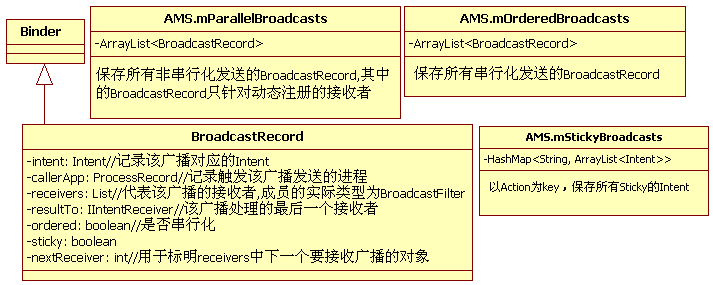


图6-19  BroadcastReceiver及相关变量

图6-19比较简单，读者可自行研究。

在代码中，registerReceiver将调用scheduleBroadcastsLocked函数，通知AMS立即着手开展广播发送工作，在其内部就发送BROADCAST\_INTENT\_MSG消息给AMS。相关的处理工作将放到本节最后再来讨论。

6.4.2  sendBroadcast流程分析

在SDK中同样定义了好几个函数用于发送广播。不过，根据之前的经验，最终和AMS交互的函数可能通过一个接口就能完成。来看最简单的广播发送函数sendBroadcast，其代码如下：

[-->ContextImpl.java::sendBroadcast]

public void sendBroadcast(Intent intent) {

   StringresolvedType = intent.resolveTypeIfNeeded(getContentResolver());

   try {

       intent.setAllowFds(false);

         //调用AMS的brodcastIntent，在SDK中定义的广播发送函数最终都会调用它

        ActivityManagerNative.getDefault().broadcastIntent(

               mMainThread.getApplicationThread(), intent, resolvedType, null,

               Activity.RESULT\_OK, null, null, null, false, false);

        }......

 }

AMS的broadcastIntent函数的主要工作将交由AMS的broadcastIntentLocked来完成，故此处直接分析broadcastIntentLocked。

1.  broadcastIntentLocked分析

我们分阶段来分析broadcastIntentLocked的工作，先来看第一阶段工作。

（1） broadcastIntentLocked分析之一

[-->ActivityManagerService.java::broadcastIntentLocked]

private final int broadcastIntentLocked(ProcessRecordcallerApp,

      StringcallerPackage, Intent intent, String resolvedType,

     IIntentReceiver resultTo, int resultCode, String resultData,

      Bundlemap, String requiredPermission,

     boolean ordered, boolean sticky, int callingPid, int callingUid) {

   intent =new Intent(intent);

   //为Intent增加FLAG\_EXCLUDE\_STOPPED\_PACKAGES标志，表示该广播不会传递给被STOPPED

   //的Package

  intent.addFlags(Intent.FLAG\_EXCLUDE\_STOPPED\_PACKAGES);

   //处理一些特殊的广播，包括UID\_REMOVED,PACKAGE\_REMOVED和PACKAGE\_ADDED等

   finalboolean uidRemoved = Intent.ACTION\_UID\_REMOVED.equals(

                                                        intent.getAction());

   ......//处理特殊的广播，主要和PACKAGE相关，例如接收到PACKAGE\_REMOVED广播后,AMS

   //需将该Package的组件从相关成员中删除，相关代码可自行阅读

   //处理TIME\_ZONE变化广播

   if(intent.ACTION\_TIMEZONE\_CHANGED.equals(intent.getAction()))

        mHandler.sendEmptyMessage(UPDATE\_TIME\_ZONE);

   //处理CLEAR\_DNS\_CACHE广播

   if(intent.ACTION\_CLEAR\_DNS\_CACHE.equals(intent.getAction()))

        mHandler.sendEmptyMessage(CLEAR\_DNS\_CACHE);

   //处理PROXY\_CHANGE广播

   if(Proxy.PROXY\_CHANGE\_ACTION.equals(intent.getAction())) {

        ProxyProperties proxy = intent.getParcelableExtra("proxy");

         mHandler.sendMessage(mHandler.obtainMessage(

                                  UPDATE\_HTTP\_PROXY,proxy));

   }

从以上代码可知，broadcastIntentLocked第一阶段的工作主要是处理一些特殊的广播消息。

下面来看broadcastIntentLocked第二阶段的工作。

（2） broadcastIntentLocked分析之二

[-->ActivityManagerService.java::broadcastIntentLocked]

   //处理发送sticky广播的情况

  if(sticky) {

   ......//检查发送进程是否有BROADCAST\_STICKY权限

   if(requiredPermission != null) {

       //在发送Sticky广播的时候，不能携带权限信息

        returnBROADCAST\_STICKY\_CANT\_HAVE\_PERMISSION;

     }

   //在发送Stikcy广播的时候，也不能指定特定的接收对象

   if(intent.getComponent() != null) ......//抛异常

   //将这个Sticky的Intent保存到mStickyBroadcasts中

  ArrayList<Intent> list =mStickyBroadcasts.get(intent.getAction());

   if (list== null) {

        list= new ArrayList<Intent>();

        mStickyBroadcasts.put(intent.getAction(), list);

    }

    ......//如果list中已经有该intent，则直接用新的intent替换旧的intent

    //否则将该intent加入到list中保存

    if (i >= N) list.add(new Intent(intent));

  }

    //定义两个变量，其中receivers用于保存匹配该Intent的所有广播接收者，包括静态或动态

    //注册的广播接收者，registeredReceivers用于保存符合该Intent的所有动态注册者

   Listreceivers = null;

   List<BroadcastFilter>registeredReceivers = null;

   try {

       if(intent.getComponent() != null) {

        ......//如果指定了接收者，则从PKMS中查询它的信息

       }else {

          //FLAG\_RECEIVER\_REGISTERED\_ONLY标签表明该广播只能发给动态注册者

          if ((intent.getFlags()&Intent.FLAG\_RECEIVER\_REGISTERED\_ONLY)== 0) {

            //如果没有设置前面的标签，则需要查询PKMS，获得那些在AndroidManifest.xml

           //中声明的广播接收者，即静态广播接收者的信息，并保存到receivers中

             receivers =

                   AppGlobals.getPackageManager().queryIntentReceivers(

                          intent, resolvedType,STOCK\_PM\_FLAGS);

            }

          //再从AMS的mReceiverResolver中查询符合条件的动态广播接收者

            registeredReceivers =mReceiverResolver.queryIntent(intent,

                                                resolvedType, false);

         }

        }......

  /\*

   判断该广播是否设置了REPLACE\_PENDING标签。如果设置了该标签，并且之前的那个Intent还

   没有被处理，则可以用新的Intent替换旧的Intent。这么做的目的是为了减少不必要的广播发送，

   但笔者感觉，这个标签的作用并不靠谱，因为只有旧的Intent没被处理的时候，它才能被替换。

   因为旧Intent被处理的时间不能确定，所以不能保证广播发送者的一番好意能够实现。因此，

   在发送广播时，千万不要以为设置了该标志就一定能节约不必要的广播发送。

  \*/

   final boolean replacePending =

        (intent.getFlags()&Intent.FLAG\_RECEIVER\_REPLACE\_PENDING) != 0;

  //先处理动态注册的接收者

   int NR =registeredReceivers != null ? registeredReceivers.size() : 0;

   //如果此次广播为非串行化发送，并且符合条件的动态注册接收者个数不为零

   if(!ordered && NR > 0) {

      //创建一个BroadcastRecord对象即可，注意，一个BroadcastRecord对象可包括所有的

     //接收者（可参考图6-19）

     BroadcastRecord r = new BroadcastRecord(intent, callerApp,

                   callerPackage, callingPid, callingUid, requiredPermission,

                   registeredReceivers, resultTo, resultCode, resultData, map,

                   ordered, sticky, false);

       boolean replaced = false;

       if (replacePending) {

       ......//从mParalledBroadcasts中查找是否有旧的Intent，如果有就替代它，并设置

      //replaced为true

     }

       if(!replaced) {//如果没有被替换，则保存到mParallelBroadcasts中

          mParallelBroadcasts.add(r);

          scheduleBroadcastsLocked();//调度一次广播发送

       }

       //至此，动态注册的广播接收者已处理完毕，设置registeredReceivers为null

      registeredReceivers = null;//

       NR =0;

  }

broadcastIntentLocked第二阶段的工作有两项：

·  查询满足条件的动态广播接收者及静态广播接收者。

·  当本次广播不为ordered时，需要尽快发送该广播。另外，非ordered的广播都被AMS保存在mParallelBroadcasts中。

（3） broadcastIntentLocked分析之三

下面来看broadcastIntentLocked最后一阶段的工作，其代码如下：

[-->ActivityManagerService.java::broadcastIntentLocked]

   int ir = 0;

   if(receivers != null) {

      StringskipPackages[] = null;

     ......//处理PACKAGE\_ADDED的Intent，系统不希望有些应用程序一安装就启动。

      //这类程序的工作原理是什么呢？即在该程序内部监听PACKAGE\_ADDED广播。如果系统

     //没有这一招防备，则PKMS安装完程序后所发送的PAKCAGE\_ADDED消息将触发该应用的启动

    ......//处理ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE广播

   ......//将动态注册的接收者registeredReceivers的元素合并到receivers中去

   //处理完毕后，所有的接收者（无论动态还是静态注册的）都存储到receivers变量中了

   if((receivers != null && receivers.size() > 0) || resultTo != null) {

         //创建一个BroadcastRecord对象，注意它的receivers中包括所有的接收者

        BroadcastRecord r = new BroadcastRecord(intent, callerApp,

              callerPackage, callingPid, callingUid, requiredPermission,

              receivers, resultTo, resultCode, resultData, map, ordered,

              sticky, false);

       booleanreplaced = false;

       if (replacePending) {

        ......//替换mOrderedBroadcasts中旧的Intent

        }//

     if(!replaced) {//如果没有替换，则添加该条广播记录到mOrderedBroadcasts中

        mOrderedBroadcasts.add(r);

        scheduleBroadcastsLocked();//调度AMS进行广播发送工作

     }

  }

  returnBROADCAST\_SUCCESS;

}

由以上代码可知，AMS将动态注册者和静态注册者都合并到receivers中去了。注意，如果本次广播不是ordered，那么表明动态注册者就已经在第二阶段工作中被处理了。因此在合并时，将不会有动态注册者被加到receivers中。最终所创建的广播记录存储在mOrderedBroadcasts中，也就是说，不管是否串行化发送，静态接收者对应的广播记录都将保存在mOrderedBroadcasts中。为什么不将它们保存在mParallelBroadcasts中呢？

结合代码会发现，保存在mParallelBroadcasts中的BroadcastRecord所包含的都是动态注册的广播接收者信息，这是因为动态接收者所在的进程是已经存在的（如果该进程不存在，如何去注册动态接收者呢？），而静态广播接收者就不能保证它已经和一个进程绑定在一起了（静态广播接收者此时可能还仅仅是在AndroidManifest.xml中声明的一个信息，相应的进程并未创建和启动）。根据前一节所分析的Activity启动流程，AMS还需要进行创建应用进程，初始化Android运行环境等一系列复杂的操作。读到后面内容时会发现，AMS将在一个循环中逐条处理mParallelBroadcasts中的成员（即派发给接收者）。如果将静态广播接收者也保存到mParallelBroadcasts中，会有什么后果呢？假设这些静态接收者所对应的进程全部未创建和启动，那么AMS将在那个循环中创建多个进程！结果让人震惊，系统压力一下就上去了。所以，对于静态注册者，它们对应的广播记录都被放到mOrderedBroadcasts中保存。AMS在处理这类广播信息时，一个进程一个进程地处理，只有处理完一个接收者，才继续下一个接收者。这种做法的好处是，避免了惊群效应的出现，坏处则是延时较长。

下面进入AMS的BROADCAST\_INTENT\_MSG消息处理函数，看看情况是否如上所说。

6.4.3  BROADCAST\_INTENT\_MSG消息处理函数

BROADCAST\_INTENT\_MSG消息将触发processNextBroadcast函数，下面分阶段来分析它。

1.  processNextBroadcast分析之一

[-->ActivityManagerService.java::processNextBroadcast]

private final void processNextBroadcast(booleanfromMsg) {

   //如果是BROADCAST\_INTENT\_MSG消息触发该函数，则fromMsg为true

  synchronized(this) {

    BroadcastRecord r;

   updateCpuStats();//更新CPU使用情况

    if(fromMsg)  mBroadcastsScheduled = false;

    //先处理mParallelBroadcasts中的成员。如前所述，AMS在一个循环中处理它们

    while(mParallelBroadcasts.size() > 0) {

         r =mParallelBroadcasts.remove(0);

         r.dispatchTime =SystemClock.uptimeMillis();

         r.dispatchClockTime =System.currentTimeMillis();

        final int N = r.receivers.size();

         for(int i=0; i<N; i++) {

           Object target = r.receivers.get(i);

            //①mParallelBroadcasts中的成员全为BroadcastFilter类型，所以下面的函数

           //将target直接转换成BroadcastFilter类型。注意，最后一个参数为false

           deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target,

                                                         false);

        }

        //将这条处理过的记录保存到mHistoryBroadcast中，供调试使用

        addBroadcastToHistoryLocked(r);

  }

deliverToRegisteredReceiverLocked函数的功能就是派发广播给接收者，其代码如下：

[-->ActivityManagerService.java::deliverToRegisteredReceiverLocked]

private final voiddeliverToRegisteredReceiverLocked(BroadcastRecord r,

                         BroadcastFilter filter, booleanordered) {

   booleanskip = false;

   //检查发送进程是否有filter要求的权限

   if(filter.requiredPermission != null) {

       intperm = checkComponentPermission(filter.requiredPermission,

                   r.callingPid, r.callingUid, -1, true);

       if(perm != PackageManager.PERMISSION\_GRANTED) skip = true;

  }

   //检查接收者是否有发送者要求的权限

   if(r.requiredPermission != null) {

       intperm = checkComponentPermission(r.requiredPermission,

               filter.receiverList.pid, filter.receiverList.uid, -1, true);

       if(perm != PackageManager.PERMISSION\_GRANTED) skip = true;

    }

   if(!skip) {

     if(ordered) {

     ......//设置一些状态，成员变量等信息，不涉及广播发送

      }

     try {

         //发送广播

        performReceiveLocked(filter.receiverList.app,

              filter.receiverList.receiver,new Intent(r.intent), r.resultCode,

              r.resultData, r.resultExtras, r.ordered, r.initialSticky);

         if(ordered) r.state = BroadcastRecord.CALL\_DONE\_RECEIVE;

       }......

     }

   }

 }

来看performReceiveLocked函数，其代码如下：

[-->ActivityManagerService.java::performReceiveLocked]

static void performReceiveLocked(ProcessRecordapp, IIntentReceiver receiver,

      Intentintent, int resultCode, String data, Bundle extras,

     boolean ordered, boolean sticky) throws RemoteException {

      if(app != null && app.thread != null) {

         //如果app及app.thread不为null，则调度scheduleRegisteredReceiver，

        //注意这个函数名为scheduleRegisteredReceiver，它只针对动态注册的广播接收者

        app.thread.scheduleRegisteredReceiver(receiver,intent, resultCode,

                   data, extras, ordered, sticky);

      } else{

       //否则调用IIntentReceiver的performReceive函数

      receiver.performReceive(intent, resultCode, data, extras,

                                     ordered, sticky);

   }

 }

对于动态注册者而言，在大部分情况下会执行if分支，所以应用进程ApplicationThread的scheduleRegisteredReceiver函数将被调用。稍后再分析应用进程的广播处理流程。

2.  processNextBroadcast分析之二

至此，processNextBroadcast已经在一个while循环中处理完mParallelBroadcasts的所有成员了，实际上，这种处理方式也会造成惊群效应，但影响相对较少。这是因为对于动态注册者来说，它们所在的应用进程已经创建并初始化成功。此处的广播发送只是调用应用进程的一个函数而已。相比于创建进程，再初始化Android运行环境所需的工作量，调用scheduleRegisteredReceiver的工作就比较轻松了。

来看processNextBroadcast第二阶段的工作，代码如下：

[-->ActivityManagerService.java::processNextBroadcast]

   /\*

    现在要处理mOrderedBroadcasts中的成员。如前所述，它要处理一个接一个的接受者，如果

    接收者所在进程还未启动，则需要等待。mPendingBroadcast变量用于标识因为应用进程还未

    启动而处于等待状态的BroadcastRecord。

   \*/

  if(mPendingBroadcast != null) {

     boolean isDead;

     synchronized (mPidsSelfLocked) {

        isDead= (mPidsSelfLocked.get(mPendingBroadcast.curApp.pid) == null);

      }

      /\***重要说明**

       判断要等待的进程是否为dead进程，如果没有dead进程，则继续等待。仔细思考，此处直接

       返回会有什么问题。

       问题不小！假设有两个ordered广播A和B，有两个接收者，AR和BR，并且BR所

       在进程已经启动并完成初始化Android运行环境。如果processNextBroadcast先处理A，

       再处理B，那么此处B的处理将因为mPendingBroadcast不为空而被延后。虽然B和A

       之间没有任何关系（例如完全是两个不同的广播消息），

       但是事实上，大多数开发人员理解的order是针对单个广播的，例如A有5个接收者，那么对这

       5个接收者的广播的处理是串行的。通过此处的代码发现，系统竟然串行处理A和B广播，即

       B广播要待到A的5个接收者都处理完了才能处理。

      \*/

      if(!isDead)   return;

      else {

       mPendingBroadcast.state = BroadcastRecord.IDLE;

       mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;

       mPendingBroadcast = null;

     }

   }

  boolean looped = false;

  do {

        // mOrderedBroadcasts处理完毕

        if (mOrderedBroadcasts.size() == 0) {

            scheduleAppGcsLocked();

            if (looped)  updateOomAdjLocked();

             return;

        }

        r =mOrderedBroadcasts.get(0);

       boolean forceReceive = false;

        //下面这段代码用于判断此条广播是否处理时间过长

        //先得到该条广播的所有接收者

        intnumReceivers = (r.receivers != null) ? r.receivers.size() : 0;

        if(mProcessesReady && r.dispatchTime > 0) {

            long now = SystemClock.uptimeMillis();

           //如果总耗时超过2倍的接收者个数\*每个接收者最长处理时间（10秒），则

           //强制结束这条广播的处理

            if ((numReceivers > 0) &&

                   (now > r.dispatchTime +

                              (2\*BROADCAST\_TIMEOUT\*numReceivers))){

               broadcastTimeoutLocked(false);//读者阅读完本节后可自行研究该函数

              forceReceive = true;

              r.state = BroadcastRecord.IDLE;

            }

      }//if(mProcessesReady...)判断结束

     if (r.state != BroadcastRecord.IDLE)   return;

     //如果下面这个if条件满足，则表示该条广播要么已经全部被处理，要么被中途取消

     if(r.receivers == null || r.nextReceiver >= numReceivers

            || r.resultAbort || forceReceive) {

         if(r.resultTo != null) {

          try {

               //将该广播的处理结果传给设置了resultTo的接收者

               performReceiveLocked(r.callerApp, r.resultTo,

                  new Intent(r.intent), r.resultCode,

                 r.resultData, r.resultExtras, false, false);

               r.resultTo = null;

            }......

       }

      cancelBroadcastTimeoutLocked();

       addBroadcastToHistoryLocked(r);

      mOrderedBroadcasts.remove(0);

       r = null;

      looped = true;

       continue;

     }

  } while (r== null);

processNextBroadcast第二阶段的工作比较简单：

·  首先根据是否处于pending状态（mPendingBroadcast不为null）进行相关操作。读者要认真体会代码中的重要说明。

·  处理超时的广播记录。这个超时时间是2\*BROADCAST\_TIMEOUT\*numReceivers。BROADCAST\_TIMEOUT默认为10秒。由于涉及创建进程，初始化Android运行环境等重体力活，故此处超时时间还乘以一个固定倍数2。

3.  processNextBroadcast分析之三

来看processNextBroadcast第三阶段的工作，代码如下：

[-->ActivityManagerService.java::processNextBroadcast]

  int recIdx= r.nextReceiver++;

 r.receiverTime = SystemClock.uptimeMillis();

  if (recIdx== 0) {

     r.dispatchTime = r.receiverTime;//记录本广播第一次处理开始的时间

      r.dispatchClockTime= System.currentTimeMillis();

  }

  //设置广播处理超时时间，BROADCAST\_TIMEOUT为10秒

  if (!mPendingBroadcastTimeoutMessage) {

        longtimeoutTime = r.receiverTime + BROADCAST\_TIMEOUT;

       setBroadcastTimeoutLocked(timeoutTime);

  }

  //取该条广播下一个接收者

  ObjectnextReceiver = r.receivers.get(recIdx);

  if(nextReceiver instanceof BroadcastFilter) {

     //如果是动态接收者，则直接调用deliverToRegisteredReceiverLocked处理

    BroadcastFilter filter = (BroadcastFilter)nextReceiver;

    deliverToRegisteredReceiverLocked(r, filter, r.ordered);

     if(r.receiver == null || !r.ordered) {

        r.state = BroadcastRecord.IDLE;

        scheduleBroadcastsLocked();

       }

     return;//已经通知一个接收者去处理该广播，需要等它的处理结果，所以此处直接返回

  }

 //如果是静态接收者，则它的真实类型是ResolveInfo

 ResolveInfo info =  (ResolveInfo)nextReceiver;

  booleanskip = false;

  //检查权限

  int perm =checkComponentPermission(info.activityInfo.permission,

        r.callingPid, r.callingUid, info.activityInfo.applicationInfo.uid,

        info.activityInfo.exported);

  if (perm!= PackageManager.PERMISSION\_GRANTED) skip = true;

  if(info.activityInfo.applicationInfo.uid != Process.SYSTEM\_UID &&

        r.requiredPermission != null) {

        ......

       }

   //设置skip为true

   if(r.curApp != null && r.curApp.crashing) skip = true;

   if (skip){

     r.receiver = null;

     r.curFilter = null;

     r.state = BroadcastRecord.IDLE;

     scheduleBroadcastsLocked();//再调度一次广播处理

     return;

   }

    r.state= BroadcastRecord.APP\_RECEIVE;

    StringtargetProcess = info.activityInfo.processName;

   r.curComponent = new ComponentName(

             info.activityInfo.applicationInfo.packageName,

             info.activityInfo.name);

  r.curReceiver = info.activityInfo;

   try {

          //设置Package stopped的状态为false

          AppGlobals.getPackageManager().setPackageStoppedState(

                       r.curComponent.getPackageName(),false);

         } ......

  ProcessRecord app = getProcessRecordLocked(targetProcess,

                         info.activityInfo.applicationInfo.uid);

   //如果该接收者对应的进程已经存在

   if (app!= null && app.thread != null) {

    try {

           app.addPackage(info.activityInfo.packageName);

           //该函数内部调用应用进程的scheduleReceiver函数，读者可自行分析

           processCurBroadcastLocked(r, app);

           return;//已经触发该接收者处理本广播，需要等待处理结果

        }......

     }

   //最糟的情况就是该进程还没有启动

   if((r.curApp=startProcessLocked(targetProcess,

          info.activityInfo.applicationInfo, true,......)!= 0)) == null) {

          ......//进程启动失败的处理

         return;

   }

  mPendingBroadcast = r; //设置mPendingBroadcast

  mPendingBroadcastRecvIndex = recIdx;

 }

对processNextBroadcast第三阶段的工作总结如下：

·  如果广播接收者为动态注册对象，则直接调用deliverToRegisteredReceiverLocked处理它。

·  如果广播接收者为静态注册对象，并且该对象对应的进程已经存在，则调用processCurBroadcastLocked处理它。

·  如果广播接收者为静态注册对象，并且该对象对应的进程还不存在，则需要创建该进程。这是最糟糕的情况。

此处，不再讨论新进程创建及Android运行环境初始化相关的逻辑，读者可返回阅读“attachApplicationLocked分析之三”，其中有处理mPendingBroadcast的内容。

6.4.4   应用进程处理广播分析

下面来分析当应用进程收到广播后的处理流程，以动态接收者为例。

1.  ApplicationThreadscheduleRegisteredReceiver函数分析

如前所述，AMS将通过scheduleRegisteredReceiver函数将广播交给应用进程，该函数代码如下：

[-->ActivityThread.java::scheduleRegisteredReceiver]

public voidscheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,

       intresultCode, String dataStr, Bundle extras, boolean ordered,

      boolean sticky) throws RemoteException {

  //又把receiver对象传了回来。还记得注册时传递的一个IIntentReceiver类型

  //的对象吗？

   receiver.performReceive(intent,resultCode, dataStr, extras, ordered,

                               sticky);

 }

就本例而言，receiver对象的真实类型为LoadedApk.ReceiverDispatcher，来看它的performReceive函数，代码如下：

[-->LoadedApk.java::performReceive]

public void performReceive(Intent intent, intresultCode,

   Stringdata, Bundle extras, boolean ordered, boolean sticky) {

   //Args是一个runnable对象

   Args args= new Args(intent, resultCode, data, extras, ordered, sticky);

      // mActivityThread是一个Handler，还记得SDK提供的两个同名的registerReceiver

     //函数吗？如果没有传递Handler，则使用主线程的Handler。

  if(!mActivityThread.post(args)) {

      if(mRegistered && ordered) {

        IActivityManager mgr = ActivityManagerNative.getDefault();

        args.sendFinished(mgr);

      }

   }

}

scheduleRegisteredReceiver最终向主线程的Handler投递了一个Args对象，这个对象的run函数将在主线程中被调用。

2.  Args.run分析

[-->LoadedApk.java::Args.run]

   publicvoid run() {

     finalBroadcastReceiver receiver = mReceiver;

     final boolean ordered = mOrdered;

     finalIActivityManager mgr = ActivityManagerNative.getDefault();

     finalIntent intent = mCurIntent;

    mCurIntent = null;

     ......

    try {

          //获取ClassLoader对象，千万注意，此处并没有通过反射机制创建一个广播接收者，

          //对于动态接收者来说，在注册前就已经创建完毕

           ClassLoadercl = mReceiver.getClass().getClassLoader();

          intent.setExtrasClassLoader(cl);

          setExtrasClassLoader(cl);

          receiver.setPendingResult(this);//设置pendingResult

          //调用该动态接收者的onReceive函数

          receiver.onReceive(mContext, intent);

    }......

     //调用finish完成工作

     if(receiver.getPendingResult() != null) finish();

    }

Finish的代码很简单，此处不在赘述，在其内部会通过sendFinished函数调用AMS的finishReceiver函数，以通知AMS。

3.  AMS的finishReceiver函数分析

不论ordered还是非orded广播，AMS的finishReceiver函数都会被调用，它的代码如下：

[-->ActivityManagerService.java::finishReceiver]

public void finishReceiver(IBinder who, intresultCode, String resultData,

           Bundle resultExtras, boolean resultAbort) {

      ......

    booleandoNext;

    final long origId =Binder.clearCallingIdentity();

    synchronized(this){

        //判断是否还需要继续调度后续的广播发送

       doNext = finishReceiverLocked(

             who, resultCode, resultData, resultExtras, resultAbort, true);

        }

        if(doNext) { //发起下一次广播发送

           processNextBroadcast(false);

        }

       trimApplications();

   Binder.restoreCallingIdentity(origId);

}

由以上代码可知，finishReceiver将根据情况调度下一次广播发送。

6.4.5  广播处理总结

广播处理的流程及相关知识点还算比较简单，可以用图6-20来表示本例的流程。

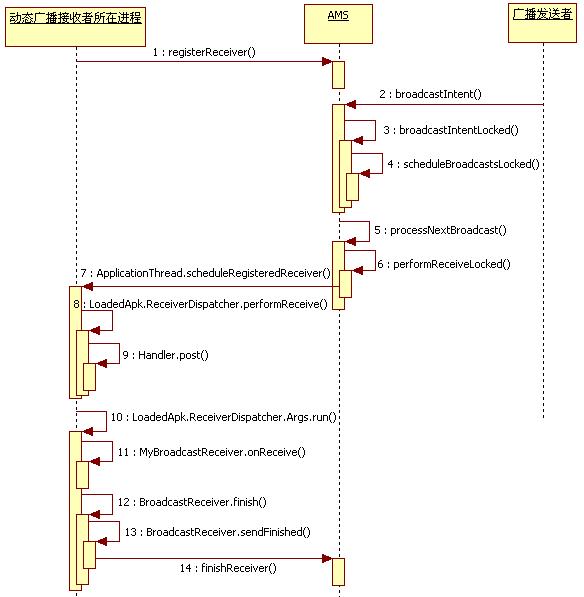


图6-20  Broadcast处理流程

在图6-20中，将调用函数所属的实际对象类型标注了出来，其中第11步的MyBroadcastReceiver为本例中所注册的广播接收者。

**注意**任何广播对于静态注册者来说，都是ordered，而且该order是全局性的，并非只针对该广播的接收者，故从广播发出到静态注册者的onReceive函数被调用中间经历的这段时间相对较长。

6.5  startService之按图索骥

Service是Android的四大组件之一。和Activity，BroadcastReceiver相比，Service定位于业务层逻辑处理，而Activity定位于前端UI层逻辑处理，BroadcastReceiver定位于通知逻辑的处理。

做为业务服务提供者，Service自有一套规则，先来看有关Service的介绍。

6.5.1  Service知识介绍

四大组件之一的Service，其定义非常符合C/S架构中Service的概念，即为Client服务，处理Client的请求。在Android中，目前接触最多的是Binder中的C/S架构。在这种架构中，Client通过调用预先定义好的业务函数向对应的Service发送请求。作为四大组件之一Service，其响应Client的请求方式有两种：

·  Client通过调用startService向Service端发送一个Intent，该Intent携带请求信息。而Service的onStartCommand会接受该Intent，并处理之。该方式是Android平台特有的，借助Intent来传递请求。

·  Client调用bindService函数和一个指定的Service建立Binder关系，即绑定成功后，Client端将得到处理业务逻辑的Binder Bp端。此后Client直接调用Bp端提供的业务函数向Service端发出请求。注意，在这种方式中，Service的onBind函数被调用，如果该Service支持Binder，需返回一个IBinder对象给客户端。

以上介绍的是Service响应客户端请求的两种方式，务必将两者分清楚。此外，这两种方式还影响Service对象的生命周期，简单总结如下：

·  对于以startService方式启动的Service对象，其生命周期一直延续到stopSelf或stopService被调用为止。

·  对于以bindService方式启动的Service对象，其生命周期延续到最后一个客户端调用完unbindService为止。

**注意**生命周期控制一般都涉及引用计数的使用。如果某Service对象同时支持这两种请求方式，那么当总引用计数减为零时，其生命就走向终点。

和Service相关的知识还有，当系统内存不足时，系统如何处理Service。如果Service和UI某个部分绑定（例如类似通知栏中Music播放的信息），那么此Service优先级较高（可通过调用startForeground把自己变成一个前台Service），系统不会轻易杀死这些Service来回收内存。

以上这些内容都较简单，阅读SDK文档中Service的相关说明即可了解，具体路径为SDK路径/docs/guide/topics/fundamentals/services.html。

6.5.2  startService流程图

本章不过多介绍和Service相关的知识，原因有二：

·  Service的处理流程和本章重点介绍的Activity的处理流程差不多，并且Service的处理逻辑更简单。能阅读到此处的读者，想必对拿下Service信心满满。

·  “授人以鱼，不如授人以渔”。希望读者在经历过如此大量而又复杂的代码分析考验后，能学会和掌握分析方法，因此本节将以startService为分析对象，把相关的流程图描绘出来，旨在帮读者根据该流程图自行研读与Service相关的处理逻辑。

startService调用轨迹如图6-21和图6-22所示。

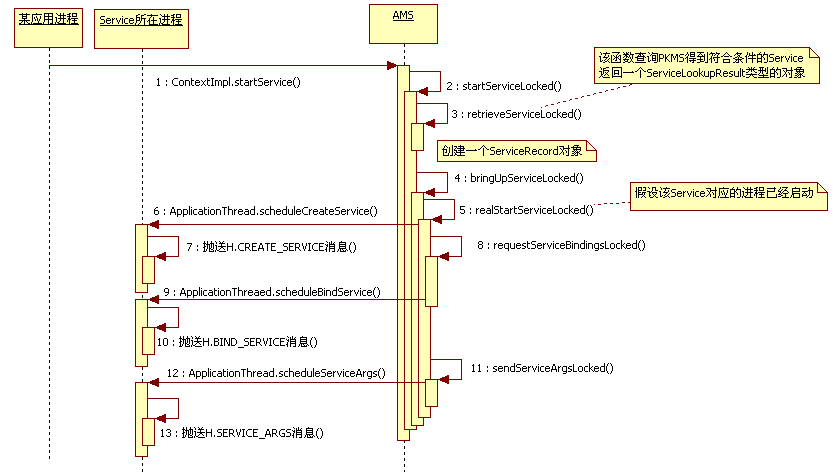


图6-21  startService流程图之一

图6-21列出了和startService相关的调用流程。在这个流程中，可假设Service所对应的进程已经存在。

单独提取图6-21中Service所在进程对H.CREATE\_SERVICE等消息的处理流程如图6-22所示。

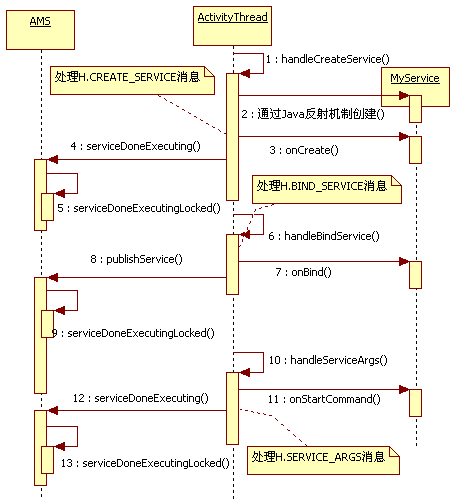


图6-22  startService中相关Message的处理流程

**注意**图6-21和图6-22中也包含了bindService的处理流程。在实际分析时，读者可分开研究bindService和startService的处理流程。

6.6  AMS中的进程管理

前面曾反复提到，Android平台中很少能接触到进程的概念，取而代之的是有明确定义的四大组件。但是作为运行在Linux用户空间内的一个系统或框架，Android不仅不能脱离进程，反而要大力利用Linux OS提供的进程管理机制和手段，更好地为自己服务。作为Android平台中组件运行管理的核心服务，ActivityManagerService当仁不让地接手了这方面的工作。目前，AMS对进程的管理仅涉及两个方面：

·  调节进程的调度优先级和调度策略。

·  调节进程的OOM值。

先来看在Linux OS中这两方面的进程管理和控制手段。

6.6.1  Linux进程管理介绍[⑥]

1.  Linux进程调度优先级和调度策略

调度优先级和调度策略是操作系统中一个很重要的概念。简而言之，它是系统中CPU资源的管理和控制手段。如何理解？此处进行简单介绍。读者可自行阅读操作系统方面的书籍以加深理解。

·  相对于在OS上运行的应用进程个数来说，CPU的资源非常有限。

·  调度优先级是OS分配CPU资源给应用进程时（即调度应用进程运行）需要参考的一个指标。一般而言，优先级高的进程将更有机会得到CPU资源。

调度策略用于描述OS调度模块分配CPU给应用进程所遵循的规则，即当将CPU控制权交给调度模块时，系统如何选择下一个要运行的进程。此处不能仅考虑各进程的调度优先级，因为存在多个进程具有相同调度优先级的情况。在这种情况下，一个需要考虑的重要因素是，每个进程所分配的时间片及它们的使用情况.

**提示**可简单认为，调度优先级及调度策略二者一起影响了系统分配CPU资源给应用进程。注意，此处的措词为“影响”，而非“控制”。因为对于用户空间可以调用的API来说，如果二者能控制CPU资源分配，那么该系统的安全性会大打折扣。

Linux提供了两个API用于设置调度优先级及调度策略。先来看设置调度优先级的函数setpriority，其原型如下：

int setpriority(int which, int who, int prio);

其中：

·  which和who参数联合使用。当which为PRIO\_PROGRESS时，who代表一个进程；当which为PRIO\_PGROUP时，who代表一个进程组；当which为PRIO\_USER时，who代表一个uid。

·  第三个参数prio用于设置应用进程的nicer值，可取范围从-20到19。Linux kernel用nicer值来描述进程的调度优先级，该值越大，表明该进程越友（nice），其被调度运行的几率越低。

下面来看设置调度策略的函数sched\_setscheduler，其原型如下：

int sched\_setscheduler(pid\_t pid, int policy,conststruct sched\_param \*param);

其中：

·  第一个参数为进程id。

·  第二个参数为调度策略。目前Android支持三种调度策略：SCHED\_OTHER，标准round-robin分时共享策略(也就是默认的策略)；SCHED\_BATCH，针对具有batch风格（批处理）进程的调度策略；SCHED\_IDLE，针对优先级非常低的适合在后台运行的进程。除此之外，Linux还支持实时（Real-time）调度策略，包括SCHED\_FIFO，先入先出调度策略；SCHED\_RR，：round-robin调度策略，也就是循环调度。

·  param参数中最重要的是该结构体中的sched\_priority变量。针对Android中的三种非实时调度策略，该值必须为NULL。

以上介绍了调度优先级和调度策略的概念。建议读者做个小实验来测试调动优先级及调动策略的作用，步骤如下：

·  挂载SD卡到PC机并往向其中复制一些媒体文件，然后重新挂载SD卡到手机。该操作就能触发MediaScanner扫描新增的这些媒体文件。

·  利用top命令查看CPU使用率，会发现进程com.process.media（即MediaScanner所在的进程）占用CPU较高（可达70%以上），原因是该进程会扫描媒体文件，该工作将利用较多的CPU资源。

·  此时，如果启动另一个进程，然后做一些操作，会发现MediaScanner所在进程的CPU利用率会降下来（例如下降到30%~40%），这表明系统将CPU资源更多地分给了用户正在操作的这个进程。

出现这种现象的原因是，MediaScannerSerivce的扫描线程将调度优先级设置为11，而默认的调度优先级为0。 相比而言，MediaScannerService优先级真的很高。

2.  关于Linux进程oom\_adj的介绍

从Linux kernel 2.6.11开始，内核提供了进程的OOM控制机制，目的是当系统出现内存不足（out of memory，简称OOM）的情况时，Kernel可根据进程的oom\_adj来选择并杀死一些进程，以回收内存。简而言之，oom\_adj可标示Linux进程内存资源的优先级，其可取范围从-16到15，另外有一个特殊值-17用于禁止系统在OOM情况下杀死该进程。和nicer值一样，oom\_adj的值越高，那么在OOM情况下，该进程越有可能被杀掉。每个进程的oom\_adj初值为0。

Linux没有提供单独的API用于设置进程的oom\_adj。目前的做法就是向/proc/进程id/oom\_adj文件中写入对应的oom\_adj值，通过这种方式就能调节进程的oom\_adj了。

另外，有必要简单介绍一下Android为Linux Kernel新增的lowmemorykiller（以后简称LMK）模块的工作方式：LMK的职责是根据当前内存大小去杀死对应oom\_adj及以上的进程以回收内存。这里有两个关键参数：为LMK设置不同的内存阈值及oom\_adj，它们分别由/sys/module/lowmemorykiller/parameters/minfree和/sys/module/lowmemorykiller/parameters/adj控制。

**注意**这两个参数的典型设置为：

minfree，2048,3072,4096,6144,7168,8192 用于描述不同级别的内存阈值，单位为KB。

adj，0,1,2,4,7,15 用于描述对应内存阈值的oom\_adj值。

表示当剩余内存为2048KB时，LMK将杀死oom\_adj大于等于0的进程。

网络上有一些关于Android手机内存优化的方法，其中一种就利用了LMK的工作原理。

**提示**lowmemorykiller的代码在kernel/drivers/staging/android/lowmemorykiller.c中，感兴趣的读者可尝试自行阅读。

6.6.2  关于Android中的进程管理的介绍

前面介绍了Linux OS中进程管理（包括调度和OOM控制）方面的API，但AMS是如何利用它们的呢？这就涉及AMS中的进程管理规则了。这里简单介绍相关规则。

Android将应用进程分为五大类，分别为Forground类、Visible类、Service类、Background类及Empty类。这五大类的划分各有规则。

1.  进程分类

（1） Forground类

该类中的进程重要性最高，属于该类的进程包括下面几种情况：

·  含一个前端Activity（即onResume函数被调用过了，或者说当前正在显示的那个Activity）。

·  含一个Service，并且该Service和一个前端Activity绑定（例如Music应用包括一个前端界面和一个播放Service，当我们一边听歌一边操作Music界面时，该Service即和一个前端Activity绑定）。

·  含一个调用了startForground的Service，或者该进程的Service正在调用其生命周期的函数（onCreate、onStart或onDestroy）。

·  最后一种情况是，该进程中有BroadcastReceiver实例正在执行onReceive函数。

（2） Visible类

属于Visible类的进程中没有处于前端的组件，但是用户仍然能看到它们，例如位于一个对话框后的Activity界面。目前该类进程包括两种：

·  该进程包含一个仅onPause被调用的Activity（即它还在前台，只不过部分界面被遮住）。

·  或者包含一个Service，并且该Service和一个Visible（或Forground）的Activity绑定（从字面意义上看，这种情况不太好和Forground进程中第二种情况区分）。

（3） Service类、Background类及Empty类

这三类进程都没有可见的部分，具体情况如下。

·  Service进程：该类进程包含一个Service。此Service通过startService启动，并且不属于前面两类进程。这种进程一般在后台默默地干活，例如前面介绍的MediaScannerService。

·  Background进程：该类进程包含当前不可见的Activity（即它们的onStop被调用过）。系统保存这些进程到一个LRU（最近最少使用）列表。当系统需要回收内存时，该列表中那些最近最少使用的进程将被杀死。

·  Empty进程：这类进程中不包含任何组件。为什么会出现这种不包括任何组件的进程呢？其实很简单，假设该进程仅创建了一个Activity，它完成工作后主动调用finish函数销毁（destroy）自己，之后该进程就会成为Empty进程。系统保留Empty进程的原因是当又重新需要它们时（例如用户在别的进程中通过startActivity启动了它们），可以省去fork进程、创建Android运行环境等一系列漫长而艰苦的工作。

通过以上介绍可发现，当某个进程和前端显示有关系时，其重要性相对要高，这或许是体现Google重视用户体验的一个很直接的证据吧。

**建议**读者可阅读SDK/docs/guide/topics/fundamentals/processes-and-threads.html以获取更为详细的信息。

2.  Process类API介绍

我们先来介绍Android平台中进程调度和OOM控制的API，它们统一被封装在Process.java中，其相关代码如下：

[-->Process.java]

//设置线程的调度优先级，Linux kernel并不区分线程和进程，二者对应同一个数据结构Task

public static final native void setThreadPriority(inttid, int priority)

           throws IllegalArgumentException, SecurityException;

/\*

   设置线程的Group，实际上就是设置线程的调度策略，目前Android定义了三种Group。

   由于缺乏相关资料，加之笔者感觉对应的注释也只可意会不可言传，故此处直接照搬了英文

   注释，敬请读者谅解。

**THREAD\_GROUP\_DEFAULT**：Default thread group - gets a 'normal'share of the CPU

**THREAD\_GROUP\_BG\_NONINTERACTIVE**：Background non-interactive thread group.

   Allthreads in this group are scheduled with a reduced share of the CPU

**THREAD\_GROUP\_FG\_BOOST**：Foreground 'boost' thread group - Allthreads in

   this group are scheduled with an increasedshare of the CPU.

   目前代码中还没有地方使用THREAD\_GROUP\_FG\_BOOST这种Group

\*/

public static final native void setThreadGroup(inttid, int group)

           throws IllegalArgumentException, SecurityException;

//设置进程的调度策略，包括该进程的所有线程

public static final native voidsetProcessGroup(int pid, int group)

           throws IllegalArgumentException, SecurityException;

//设置线程的调度优先级

public static final native voidsetThreadPriority(int priority)

           throws IllegalArgumentException, SecurityException;

//调整进程的oom\_adj值

public static final native boolean setOomAdj(intpid, int amt);

Process类还为不同调度优先级定义一些非常直观的名字以避免在代码中直接使用整型，例如为最低的调度优先级19定义了整型变量THREAD\_PRIORITY\_LOWEST。除此之外，Process还提供了fork子进程等相关的函数。

**注意**Process.java中的大多数函数是由JNI层实现的，其中Android在调度策略设置这一功能上还有一些特殊的地方，感兴趣的读者不妨阅读system/core/libcutils/sched\_policy.c文件。

3.  关于ProcessList类和ProcessRecord类的介绍

（1） ProcessList类的介绍

ProcessList类有两个主要功能：

·  定义一些成员变量，这些成员变量描述了不同状态下进程的oom\_adj值。

·  在Android 4.0之后，LMK的配置参数由ProcessList综合考虑手机总内存大小和屏幕尺寸后再行设置（在Android 2.3中，LMK的配置参数在init.rc中由init进程设置，并且没有考虑屏幕尺寸的影响）。读者可自行阅读其updateOomLevels函数，此处不再赘述。

本节主要关注ProcessList对oom\_adj的定义。虽然前面介绍时将Android进程分为五大类，但是在实际代码中的划分更为细致，考虑得更为周全。

[-->ProcessList.java]

class ProcessList {

    //当一个进程连续发生Crash的间隔小于60秒时，系统认为它是为Bad进程

    staticfinal int MIN\_CRASH\_INTERVAL = 60\*1000;

   //下面定义各种状态下进程的oom\_adj

   //不可见进程的oom\_adj，最大15，最小为7。杀死这些进程一般不会带来用户能够察觉的影响

    staticfinal int HIDDEN\_APP\_MAX\_ADJ = 15;

    staticint HIDDEN\_APP\_MIN\_ADJ = 7;

   /\*

   位于B List中Service所在进程的oom\_adj。什么样的Service会在BList中呢？其中的

   解释只能用原文来表达” these are the old anddecrepit services that aren't as

   shiny andinteresting as the ones in the A list“

   \*/

    staticfinal int SERVICE\_B\_ADJ = 8;

   /\*

   前一个进程的oom\_adj，例如从A进程的Activity跳转到位于B进程的Activity，

   B进程是当前进程，而A进程就是previous进程了。因为从当前进程A退回B进程是一个很简单

   却很频繁的操作（例如按back键退回上一个Activity），所以previous进程的oom\_adj

   需要单独控制。这里不能简单按照五大类来划分previous进程，还需要综合考虑

   \*/

   staticfinal int PREVIOUS\_APP\_ADJ = 7;

   //Home进程的oom\_adj为6，用户经常和Home进程交互，故它的oom\_adj也需要单独控制

    staticfinal int HOME\_APP\_ADJ = 6;

   //Service类中进程的oom\_adj为5

    staticfinal int SERVICE\_ADJ = 5;

   //正在执行backup操作的进程，其oom\_adj为4

    staticfinal int BACKUP\_APP\_ADJ = 4;

   //heavyweight的概念前面曽提到过，该类进程的oom\_adj为3

    staticfinal int HEAVY\_WEIGHT\_APP\_ADJ = 3;

   //Perceptible进程指那些当前并不在前端显示而用户能感觉到它在运行的进程，例如正在

   //后台播放音乐的Music

    staticfinal int PERCEPTIBLE\_APP\_ADJ = 2;

   //Visible类进程，oom\_adj为1

    staticfinal int VISIBLE\_APP\_ADJ = 1;

   //Forground类进程，oom\_adj为0

    staticfinal int FOREGROUND\_APP\_ADJ = 0;

   //persistent类进程（即退出后，系统需要重新启动的重要进程），其oom\_adj为-12

    staticfinal int PERSISTENT\_PROC\_ADJ = -12;

    //核心服务所在进程，oom\_adj为-12

    staticfinal int CORE\_SERVER\_ADJ = -12;

    //系统进程，其oom\_adj为-16

    staticfinal int SYSTEM\_ADJ = -16;

    //内存页大小定义为4KB

    staticfinal int PAGE\_SIZE = 4\*1024;

    //系统中能容纳的不可见进程数最少为2，最多为15。在Android 4.0系统中，可通过设置程序来

    //调整

    staticfinal int MIN\_HIDDEN\_APPS = 2;

    staticfinal int MAX\_HIDDEN\_APPS = 15;

  //下面两个参数用于调整进程的LRU权重。注意，它们的注释和具体代码中的用法不能统一起来

  staticfinal long CONTENT\_APP\_IDLE\_OFFSET = 15\*1000;

  staticfinal long EMPTY\_APP\_IDLE\_OFFSET = 120\*1000;

  //LMK设置了6个oom\_adj阈值

  privatefinal int[] mOomAdj = new int[] {

           FOREGROUND\_APP\_ADJ, VISIBLE\_APP\_ADJ, PERCEPTIBLE\_APP\_ADJ,

           BACKUP\_APP\_ADJ, HIDDEN\_APP\_MIN\_ADJ, EMPTY\_APP\_ADJ

  };

   //用于内存较低机器（例如小于512MB）中LMK的内存阈值配置

    privatefinal long[] mOomMinFreeLow = new long[] {

           8192, 12288, 16384,

           24576, 28672, 32768

    };

   //用于较高内存机器（例如大于1GB）的LMK内存阈值配置

   privatefinal long[] mOomMinFreeHigh = new long[] {

           32768, 40960, 49152,

           57344, 65536, 81920

    };

从以上代码中定义的各种ADJ值可知，AMS中的进程管理规则远比想象得要复杂（读者以后见识到具体的代码，更会有这样的体会）。

**说明**在ProcessList中定义的大部分变量在Android 2.3代码中定义于ActivityManagerService.java中，但这段代码的开发者仅把代码复制了过来，其中的注释并未随着系统升级而更新。

（2） ProcessRecord中相关成员变量的介绍

 ProcessRecord定义了较多成员变量用于进程管理。笔者不打算深究其中的细节。这里仅把其中的主要变量及一些注释列举出来。下文会分析到它们的作用。

[-->ProcessRecord.java]

//用于LRU列表控制

long lastActivityTime;   // For managing the LRU list

long lruWeight;      // Weight for ordering in LRU list

//和oom\_adj有关

int maxAdj;           // Maximum OOM adjustment for thisprocess

int hiddenAdj;       // If hidden, this is the adjustment touse

int curRawAdj;       // Current OOM unlimited adjustment forthis process

int setRawAdj;       // Last set OOM unlimited adjustment forthis process

int curAdj;           // Current OOM adjustment for thisprocess

int setAdj;           // Last set OOM adjustment for thisprocess

//和调度优先级有关

int curSchedGroup;   // Currently desired scheduling class

int setSchedGroup;   // Last set to background scheduling class

//回收内存级别，见后文解释

int trimMemoryLevel; // Last selected memorytrimming level

//判断该进程的状态，主要和其中运行的Activity，Service有关

boolean keeping;     // Actively running code sodon't kill due to that?

boolean setIsForeground;    // Running foreground UI when last set?

boolean foregroundServices; // Running anyservices that are foreground?

boolean foregroundActivities; // Running anyactivities that are foreground?

boolean systemNoUi;   // This is a system process, but notcurrently showing UI.

boolean hasShownUi;  // Has UI been shown in this process since itwas started?

boolean pendingUiClean;     // Want to clean up resources from showingUI?

boolean hasAboveClient;     // Bound using BIND\_ABOVE\_CLIENT, so wantto be lower

//是否处于系统BadProcess列表

boolean bad;                // True if disabled in the badprocess list

//描述该进程因为是否有太多后台组件而被杀死

boolean killedBackground;   // True when proc has been killed due to toomany bg

String waitingToKill;       // Process is waiting to be killed whenin the bg; reason

//序号，每次调节进程优先级或者LRU列表位置时，这些序号都会递增

int adjSeq;                 // Sequence id for identifyingoom\_adj assignment cycles

int lruSeq;                 // Sequence id for identifyingLRU update cycles

上面注释中提到了LRU（最近最少使用）一词，它和AMS另外一个用于管理应用进程ProcessRecord的数据结构有关。

**提示**进程管理和调度一向比较复杂，从ProcessRecord定义的这些变量中可见一斑。需要提醒读者的是，对这部分功能的相关说明非常少，代码读起来会感觉比较晦涩。

6.6.3  AMS进程管理函数分析

在AMS中，和进程管理有关的函数只要有两个，分别是updateLruProcessLocked和updateOomAdjLocked。这两个函数的调用点有多处，本节以attachApplication为切入点，尝试对它们进行分析。

**注意**AMS一共定义了3个updateOomAdjLocked函数，此处将其归为一类。

先回顾一下attachApplication函数被调用的情况：AMS新创建一个应用进程，该进程启动后最重要的就是调用AMS的attachApplication。

**提示**不熟悉的读者可阅读6.3.3节的第5小节。

其相关代码如下：

[-->ActivityManagerService.java::attachApplicationLocked]

//attachApplication主要工作由attachApplicationLocked完成，故直接分析它

private final booleanattachApplicationLocked(IApplicationThread thread,

                                                      int pid) {

 ProcessRecord app;

  //根据之前的介绍的内容，AMS在创建应用进程前已经将对应的ProcessRecord保存到

  //mPidsSelfLocked中了

  ...... //其他一些处理

  //初始化ProcessRecord中的一些成员变量

  app.thread= thread;

  app.curAdj= app.setAdj = -100;

 app.curSchedGroup = Process.THREAD\_GROUP\_DEFAULT;

  app.setSchedGroup= Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

 app.forcingToForeground = null;

 app.foregroundServices = false;

 app.hasShownUi = false;

  ......

  //调用应用进程的bindApplication，以初始化其内部的Android运行环境

  thread.bindApplication(......);

  //①调用updateLruProcessLocked函数

 updateLruProcessLocked(app, false, true);

 app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();

  ......//启动Activity等操作

  //②didSomething为false，则调用updateOomAdjLocked函数

  if(!didSomething) {

      updateOomAdjLocked();

 }

在以上这段代码中有两个重要函数调用，分别是updateLruProcessLocked和updateOomAdjLocked。

1.  updateLruProcessLocked函数分析

根据前文所述，我们知道了系统中所有应用进程（同时包括SystemServer）的ProcessRecord信息都保存在mPidsSelfLocked成员中。除此之外，AMS还有一个成员变量mLruProcesses也用于保存ProcessRecord。mLruProcesses的类型虽然是ArrayList，但其内部成员却是按照ProcessRecord的lruWeight大小排序的。在运行过程中，AMS会根据lruWeight的变化调整mLruProcesses成员的位置。

就本例而言，刚连接（attach）上的这个应用进程的ProcessRecord需要通过updateLruProcessLocked函数加入mLruProcesses数组中。来看它的代码，如下所示：

[-->ActivityManagerService.java::updateLruProcessLocked]

final void updateLruProcessLocked(ProcessRecordapp,

           boolean oomAdj, boolean updateActivityTime) {

  mLruSeq++;//每一次调整LRU列表，系统都会分配一个唯一的编号

  updateLruProcessInternalLocked(app, oomAdj, updateActivityTime, 0);

 }

[-->ActivityManagerService.java::updateLruProcessInternalLocked]

private final voidupdateLruProcessInternalLocked(ProcessRecord app,

           boolean oomAdj, boolean updateActivityTime, int bestPos) {

  //获取app在mLruProcesses中的索引位置，对于本例而言，返回值lrui为-1

  int lrui =mLruProcesses.indexOf(app);

  //如果之前有记录，则先从数组中删掉，因为此处需要重新调整位置

  if (lrui>= 0) mLruProcesses.remove(lrui);

  //获取mLruProcesses中数组索引的最大值，从0开始

  int i =mLruProcesses.size()-1;

  intskipTop = 0;

  app.lruSeq= mLruSeq;  //将系统全局的lru调整编号赋给ProcessRecord的lruSeq

  //更新lastActivityTime值，其实就是获取一个时间

  if(updateActivityTime) {

        app.lastActivityTime =SystemClock.uptimeMillis();

  }

  if(app.activities.size() > 0) {

      //如果该app含Activity，则lruWeight为当前时间

      app.lruWeight = app.lastActivityTime;

   } else if(app.pubProviders.size() > 0) {

       /\*

        如果有发布的ContentProvider，则lruWeight要减去一个OFFSET。

        对此的理解需结合CONTENT\_APP\_IDLE\_OFFSET的定义。读者暂时把它

        看做一个常数

        \*/

       app.lruWeight = app.lastActivityTime -

                            ProcessList.CONTENT\_APP\_IDLE\_OFFSET;

        //设置skipTop。这个变量实际上没有用，放在此处让人很头疼

        skipTop = ProcessList.MIN\_HIDDEN\_APPS;

   } else {

         app.lruWeight = app.lastActivityTime -

                              ProcessList.EMPTY\_APP\_IDLE\_OFFSET;

         skipTop = ProcessList.MIN\_HIDDEN\_APPS;

   }

  //从数组最后一个元素开始循环

  while (i>= 0) {

     ProcessRecord p = mLruProcesses.get(i);

     //下面这个if语句没有任何意义，因为skipTop除了做自减操作外，不影响其他任何内容

      if(skipTop > 0 && p.setAdj >= ProcessList.HIDDEN\_APP\_MIN\_ADJ) {

          skipTop--;

       }

       //将app调整到合适的位置

       if(p.lruWeight <= app.lruWeight || i < bestPos) {

           mLruProcesses.add(i+1, app);

           break;

       }

       i--;

  }

  //如果没有找到合适的位置，则把app加到队列头

  if (i <0)   mLruProcesses.add(0, app);

   //如果该将app 绑定到其他service，则要对应调整Service所在进程的LRU

   if (app.connections.size() > 0) {

       for(ConnectionRecord cr : app.connections) {

          if(cr.binding != null && cr.binding.service != null

                        && cr.binding.service.app!= null

                        &&cr.binding.service.app.lruSeq != mLruSeq) {

                        updateLruProcessInternalLocked(cr.binding.service.app,

                               oomAdj,updateActivityTime,i+1);

            }

        }

    }

  //conProviders也是一种Provider，相关信息下一章再介绍

    if(app.conProviders.size() > 0) {

         for(ContentProviderRecord cpr : app.conProviders.keySet()) {

            ......//对ContentProvider所在进程做类似的调整

           }

        }

  //在本例中，oomAdj为false，故updateOomAdjLocked不会被调用

  if (oomAdj) updateOomAdjLocked(); //以后分析

}

从以上代码可知，updateLruProcessLocked的主要工作是根据app的lruWeight值调整它在数组中的位置。lruWeight值越大，其在数组中的位置就越靠后。如果该app和某些Service（仅考虑通过bindService建立关系的那些Service）或ContentProvider有交互关系，那么这些Service或ContentProvider所在的进程也需要调节lruWeight值。

下面介绍第二个重要函数updateOomAdjLocked。

**提示**在以上代码中， skipTop变量完全没有实际作用，却给为阅读代码带来了很大干扰。

2.  updateOomAdjLocked函数分析

（1） updateOomAdjLocked分析之一

分段来看updateOomAdjLocked函数。

[-->ActivityManagerService.java::updateOomAdjLocked()]

final void updateOomAdjLocked() {

  //在一般情况下，resumedAppLocked返回 mResumedActivity，即当前正处于前台的Activity

  finalActivityRecord TOP\_ACT = resumedAppLocked();

  //得到前台Activity所属进程的ProcessRecord信息

  finalProcessRecord TOP\_APP = TOP\_ACT != null ? TOP\_ACT.app : null;

  mAdjSeq++;//oom\_adj在进行调节时也会有唯一的序号

  mNewNumServiceProcs= 0;

  /\*

    下面这几句代码的作用如下：

    1 根据hidden adj划分级别，一共有9个级别（即numSlots值）

    2 根据mLruProcesses的成员个数计算平均落在各个级别的进程数（即factor值）。但是这里

      的魔数（magic number）4却令人头疼不已。如有清楚该内容的读者，不妨让分享一下

      研究结果

  \*/

  intnumSlots = ProcessList.HIDDEN\_APP\_MAX\_ADJ -

                                          ProcessList.HIDDEN\_APP\_MIN\_ADJ + 1;

  int factor= (mLruProcesses.size()-4)/numSlots;

  if (factor< 1) factor = 1;

  int step =0;

  intnumHidden = 0;

  int i =mLruProcesses.size();

  intcurHiddenAdj = ProcessList.HIDDEN\_APP\_MIN\_ADJ;

 //从mLruProcesses数组末端开始循环

  while (i> 0) {

     i--;

     ProcessRecordapp = mLruProcesses.get(i);

     //①调用另外一个updateOomAdjLocked函数

     updateOomAdjLocked(app,curHiddenAdj, TOP\_APP, true);

     // updateOomAdjLocked函数会更新app的curAdj

    if(curHiddenAdj < ProcessList.HIDDEN\_APP\_MAX\_ADJ

                              &&app.curAdj == curHiddenAdj) {

       /\*

       这段代码的目的其实很简单。即当某个adj级别的ProcessRecord处理个数超过均值后，

       就跳到下一级别进行处理。注意，这段代码的结果会影响updateOomAdjLocked的第二个参数

      \*/

           step++;

           if(step >= factor) {

               step = 0;

               curHiddenAdj++;

           }

       }// if(curHiddenAdj < ProcessList.HIDDEN\_APP\_MAX\_ADJ...)判断结束

     //app.killedBackground初值为false

     if(!app.killedBackground) {

          if (app.curAdj >= ProcessList.HIDDEN\_APP\_MIN\_ADJ) {

                numHidden++;

              //mProcessLimit初始值为ProcessList.MAX（值为15），

             //可通过setProcessLimit函数对其进行修改

                if (numHidden > mProcessLimit) {

                      app.killedBackground =true;

                      //如果后台进程个数超过限制，则会杀死对应的后台进程

                     Process.killProcessQuiet(app.pid);

                }

            }

      }//if(!app.killedBackground)判断结束

  }//while循环结束

updateOomAdjLocked第一阶段的工作看起来很简单，但是其中也包含一些较难理解的内容。

·  处理hidden adj，划分9个级别。

·  根据mLruProcesses中进程个数计算每个级别平均会存在多少进程。在这个计算过程中出现了一个魔数4令人极度费解。

·  然后利用一个循环从mLruProcesses末端开始对每个进程执行另一个updateOomAdjLocked函数。关于这个函数的内容，我们放到下一节再讨论。

·  判断处于Hidden状态的进程数是否超过限制，如果超过限制，则会杀死一些进程。

接着来看updateOomAdjLocked下一阶段的工作。

（2） updateOomAdjLocked分析之二

[-->ActivityManagerService.java::updateOomAdjLocked]

 mNumServiceProcs = mNewNumServiceProcs;

 //numHidden表示处于hidden状态的进程个数

  //当Hidden进程个数小于7时候（15/2的整型值）,执行if分支

  if(numHidden <= (ProcessList.MAX\_HIDDEN\_APPS/2)) {

       ......

     /\*

      我们不讨论这段缺乏文档及使用魔数的代码，但这里有个知识点要注意：

      该知识点和Android 4.0新增接口ComponentCallbacks2有关，主要是通知应用进程进行

      内存清理，ComponentCallbacks2接口定了一个函数onTrimMemory(int level)，

      而四大组件除BroadcastReceiver外，均实现了该接口。系统定义了4个level以通知进程

      做对应处理：

**TRIM\_MEMORY\_UI\_HIDDEN**，提示进程当前不处于前台，故可释放一些UI资源

**TRIM\_MEMORY\_BACKGROUND**，表明该进程已加入LRU列表，此时进程可以对一些简单的资源

      进行清理

**TRIM\_MEMORY\_MODERATE**，提示进程可以释放一些资源，这样其他进程的日子会好过些。

      即所谓的“我为人人，人人为我”

**TRIM\_MEMORY\_COMPLETE**，该进程需尽可能释放一些资源，否则当内存不足时，它可能被杀死

      \*/

   } else {//假设hidden进程数超过7，

      finalint N = mLruProcesses.size();

      for(i=0; i<N; i++) {

       ProcessRecord app = mLruProcesses.get(i);

        if ((app.curAdj >ProcessList.VISIBLE\_APP\_ADJ || app.systemNoUi)

              && app.pendingUiClean) {

            if (app.trimMemoryLevel <

                        ComponentCallbacks2.TRIM\_MEMORY\_UI\_HIDDEN

                        && app.thread!= null) {

                   try {//调用应用进程ApplicationThread的scheduleTrimMemory函数

                        app.thread.scheduleTrimMemory(

                                    ComponentCallbacks2.TRIM\_MEMORY\_UI\_HIDDEN);

                      }......

                }// if (app.trimMemoryLevel...)判断结束

              app.trimMemoryLevel =

                             ComponentCallbacks2.TRIM\_MEMORY\_UI\_HIDDEN;

                   app.pendingUiClean = false;

         }else {

              app.trimMemoryLevel = 0;

          }

       }//for循环结束

        }//else结束

   //Android4.0中设置有一个开发人员选项，其中有一项用于控制是否销毁后台的Activity。

   //读者可自行研究destroyActivitiesLocked函数

   if (mAlwaysFinishActivities)

       mMainStack.destroyActivitiesLocked(null, false,"always-finish");

 }

通过上述代码，可获得两个信息：

·  Android 4.0增加了新的接口类ComponentCallbacks2，其中只定义了一个函数onTrimMemory。从以上描述中可知，它主要通知应用进程进行一定的内存释放。

·  Android 4.0 Settings新增了一个开放人员选项，通过它可控制AMS对后台Activity的操作。

这里和读者探讨一下ComponentCallbacks2接口的意义。此接口的目的是通知应用程序根据情况做一些内存释放，但笔者觉得，这种设计方案的优劣尚有待考证，主要是出于以下下几种考虑：

第一，不是所有应用程序都会实现该函数。原因有很多，主要原因是，该接口只是SDK 14才有的，之前的版本没有这个接口。另外，应用程序都会尽可能抢占资源（在不超过允许范围内）以保证运行速度，不应该考虑其他程序的事情。

第二个重要原因是无法区分在不同的level下到底要释放什么样的内存。代码中的注释也是含糊其辞。到底什么样的资源可以在TRIM\_MEMORY\_BACKGROUND级别下释放，什么样的资源不可以在TRIM\_MEMORY\_BACKGROUND级别下释放？

既然系统加了这些接口，读者不妨参考源码中的使用案例来开发自己的程序。

**建议**真诚希望Google能给出一个明确的文档，说明这几个函数该怎么使用。

接下来分析在以上代码中出现的针对每个ProcessRecord都调用的updateOomAdjLocked函数。

3.  第二个updateOomAdjLocked分析

[-->ActivityManagerService.java::updateOomAdjLocked]

private final boolean updateOomAdjLocked( ProcessRecordapp, int hiddenAdj,

           ProcessRecord TOP\_APP, boolean doingAll) {

  //设置该app的hiddenAdj

 app.hiddenAdj = hiddenAdj;

  if(app.thread == null)  return false;

  finalboolean wasKeeping = app.keeping;

  booleansuccess = true;

   //下面这个函数的调用极其关键。从名字上看，它会计算该进程的oom\_adj及调度策略

  computeOomAdjLocked(app, hiddenAdj, TOP\_APP, false, doingAll);

  if(app.curRawAdj != app.setRawAdj) {

      if (wasKeeping && !app.keeping) {

           .....//统计电量

          app.lastCpuTime = app.curCpuTime;

        }

     app.setRawAdj = app.curRawAdj;

  }

  //如果新旧oom\_adj不同，则重新设置该进程的oom\_adj

  if(app.curAdj != app.setAdj) {

       if(Process.setOomAdj(app.pid, app.curAdj)) //设置该进程的oom\_adj

            app.setAdj = app.curAdj;

       .....

   }

  //如果新旧调度策略不同，则需重新设置该进程的调度策略

  if(app.setSchedGroup != app.curSchedGroup) {

      app.setSchedGroup = app.curSchedGroup;

        //waitingToKill是一个字符串，用于描述杀掉该进程的原因

       if (app.waitingToKill != null &&

            app.setSchedGroup == Process.THREAD\_GROUP\_BG\_NONINTERACTIVE) {

             Process.killProcessQuiet(app.pid);//

            success = false;

        } else{

          if (true) {//强制执行if分支

                long oldId = Binder.clearCallingIdentity();

                   try {//设置进程调度策略

                       Process.setProcessGroup(app.pid, app.curSchedGroup);

                   }......

               } ......

           }

        }

   returnsuccess;

 }

上面的代码还算简单，主要完成两项工作：

·  调用computeOomAdjLocked计算获得某个进程的oom\_adj和调度策略。

·  调整进程的调度策略和oom\_adj。

**建议**思考一个问题：为何AMS只设置进程的调度策略，而不设置进程的调度优先级？

看来AMS调度算法的核心就在computeOomAdjLocked中。

4.  computeOomAdjLocked分析

这段代码较长，其核心思想是综合考虑各种情况以计算进程的oom\_adj和调度策略。建议读者阅读代码时聚焦到AMS关注的几个因素上。computeOomAdjLocked的代码如下：

[-->ActivityManagerService.java::computeOomAdjLocked]

private final intcomputeOomAdjLocked(ProcessRecord app, int hiddenAdj,

           ProcessRecord TOP\_APP, boolean recursed, boolean doingAll) {

  ......

  app.adjTypeCode= ActivityManager.RunningAppProcessInfo.REASON\_UNKNOWN;

 app.adjSource = null;

 app.adjTarget = null;

  app.empty= false;

  app.hidden= false;

  //该应用进程包含Activity的个数

  final intactivitiesSize = app.activities.size();

  //如果maxAdj小于FOREGROUND\_APP\_ADJ，基本上没什么工作可以做了。这类进程优先级相当高

  if(app.maxAdj <= ProcessList.FOREGROUND\_APP\_ADJ) {

       ......//读者可自行阅读这块代码

      return(app.curAdj=app.maxAdj);

   }

  finalboolean hadForegroundActivities = app.foregroundActivities;

 app.foregroundActivities = false;

 app.keeping = false;

 app.systemNoUi = false;

  int adj;

  intschedGroup;

  //如果app为前台Activity所在的那个应用进程

  if (app ==TOP\_APP) {

       adj =ProcessList.FOREGROUND\_APP\_ADJ;

      schedGroup = Process.THREAD\_GROUP\_DEFAULT;

       app.adjType= "top-activity";

      app.foregroundActivities = true;

    } else if (app.instrumentationClass != null){

             ......//略过instrumentationClass不为null的情况

    } else if (app.curReceiver != null ||

         (mPendingBroadcast != null && mPendingBroadcast.curApp == app)){

         //此情况对应正在执行onReceive函数的广播接收者所在进程，它的优先级也很高

         adj = ProcessList.FOREGROUND\_APP\_ADJ;

         schedGroup = Process.THREAD\_GROUP\_DEFAULT;

         app.adjType = "broadcast";

   } else if(app.executingServices.size() > 0) {

        //正在执行Service生命周期函数的进程

        adj= ProcessList.FOREGROUND\_APP\_ADJ;

       schedGroup = Process.THREAD\_GROUP\_DEFAULT;

       app.adjType = "exec-service";

    } elseif (activitiesSize > 0) {

         adj = hiddenAdj;

        schedGroup =Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

        app.hidden = true;

        app.adjType = "bg-activities";

    } else {//不含任何组件的进程，即所谓的Empty进程

         adj= hiddenAdj;

        schedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

         app.hidden= true;

         app.empty = true;

        app.adjType = "bg-empty";

   }

   //下面几段代码将根据情况重新调整前面计算处理的adj和schedGroup，我们以后面的

   //mHomeProcess判断为例

   if(!app.foregroundActivities && activitiesSize > 0) {

         //对无前台Activity所在进程的处理

   }

   if (adj> ProcessList.PERCEPTIBLE\_APP\_ADJ) {

        .......

    }

   //如果前面计算出来的adj大于HOME\_APP\_ADJ，并且该进程又是Home进程，则需要重新调整

   if (adj> ProcessList.HOME\_APP\_ADJ && app == mHomeProcess) {

        //重新调整adj和schedGroupde的值

        adj = ProcessList.HOME\_APP\_ADJ;

       schedGroup = Process.THREAD\_GROUP\_BG\_NONINTERACTIVE;

       app.hidden = false;

       app.adjType = "home";//描述调节adj的原因

   }

   if (adj> ProcessList.PREVIOUS\_APP\_ADJ && app == mPreviousProcess

           && app.activities.size() > 0) {

          ......

   }

  app.adjSeq = mAdjSeq;

  app.curRawAdj = adj;

    ......

   //下面这几段代码处理那些进程中含有Service，ContentProvider组件情况下的adj调节

   if(app.services.size() != 0 && (adj > ProcessList.FOREGROUND\_APP\_ADJ

          || schedGroup == Process.THREAD\_GROUP\_BG\_NONINTERACTIVE)) {

   }

   if(s.connections.size() > 0 && (adj >ProcessList.FOREGROUND\_APP\_ADJ

             || schedGroup == Process.THREAD\_GROUP\_BG\_NONINTERACTIVE)) {

   }

   if(app.pubProviders.size() != 0 && (adj >ProcessList.FOREGROUND\_APP\_ADJ

              || schedGroup == Process.THREAD\_GROUP\_BG\_NONINTERACTIVE)) {

           ......

   }

   //终于计算完毕

  app.curRawAdj = adj;

   if (adj> app.maxAdj) {

        adj= app.maxAdj;

        if(app.maxAdj <= ProcessList.PERCEPTIBLE\_APP\_ADJ)

           schedGroup = Process.THREAD\_GROUP\_DEFAULT;

    }

    if (adj< ProcessList.HIDDEN\_APP\_MIN\_ADJ)

        app.keeping = true;

     ......

   app.curAdj = adj;

   app.curSchedGroup = schedGroup;

   ......

   returnapp.curRawAdj;

}

computeOomAdjLocked的工作比较琐碎，实际上也谈不上什么算法，仅仅是简单地根据各种情况来设置几个值。随着系统的改进和完善，这部分代码变动的可能性比较大。

5. updateOomAdjLocked调用点统计

updateOomAdjLocked调用点很多，这里给出其中一个updateOomAdjLocked函数的调用点统计，如图6-23所示。

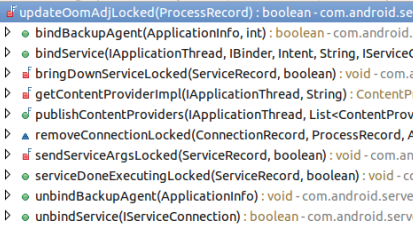


图6-23  updateOomAdjLocked函数的调用点统计图

注意，图6-23统计的是updateOomAdjLocked(ProcessRecord)函数的调用点。从该图可知，此函数被调用的地方较多，这也说明AMS非常关注应用进程的状况。

**提示**笔者觉得，AMS中这部分代码不是特别高效，不知各位读者是否有同感,？

6.6.4  AMS进程管理总结

本节首先向读者介绍了Linux平台中和进程调度、OOM管理方面的API，然后介绍了AMS如何利用这些API完成Android平台中进程管理方面的工作，从中可以发现，AMS设置的检查点比较密集，也就是经常会进行进程调度方面的操作。

6.7  App的 Crash处理

在Android平台中，应用进程fork出来后会为虚拟机设置一个未截获异常处理器，即在程序运行时，如果有任何一个线程抛出了未被截获的异常，那么该异常最终会抛给未截获异常处理器去处理。设置未截获异常处理器的代码如下：

[-->RuntimeInit.java::commonInit]

private static final void commonInit() {

   //调用完毕后，该应用中所有线程抛出的未处理异常都会由UncaughtHandler来处理

   Thread.setDefaultUncaughtExceptionHandler(newUncaughtHandler());

  ......

}

应用程序有问题是再平常不过的事情了，不过，当抛出的异常没有被截获时，系统又会做什么处理呢？来看UncaughtHandler的代码。

6.7.1  应用进程的Crash处理

[-->RuntimeInit.java::UncaughtHandler]

 privatestatic class UncaughtHandler implements

                                           Thread.UncaughtExceptionHandler{

   publicvoid uncaughtException(Thread t, Throwable e) {

    try {

          if (mCrashing) return;

           mCrashing = true;

         //调用AMS的handleApplicationCrash函数。第一个参数mApplicationObject其实

        //就是前面经常见到的ApplicationThread对象

          ActivityManagerNative.getDefault().handleApplicationCrash(

               mApplicationObject, new  ApplicationErrorReport.CrashInfo(e));

     } ......

    finally{

           //这里有一句注释很有意思：Try everything to make surethis process goes

          //away.从下面这两句调用来看，该应用进程确实想方设法要离开Java世界

          Process.killProcess(Process.myPid());//把自己杀死

          System.exit(10);//如果上面那句话不成功，再尝试exit方法

     }

   }

}

6.7.2  AMS的handleApplicationCrash分析

AMS handleApplicationCrash函数的代码如下：

[-->ActivityManagerService.java::handleApplicationCrash]

public void handleApplicationCrash(IBinder app,

                             ApplicationErrorReport.CrashInfocrashInfo) {

  //找到对应的ProcessRecord信息，后面那个字符串“Crash”用于打印输出

 ProcessRecord r = findAppProcess(app, "Crash");

  finalString processName = app == null ? "system\_server"

                        : (r == null ? "unknown": r.processName);

   //添加crash信息到dropbox中

 ddErrorToDropBox("crash", r, processName, null, null, null,null, null,

                         crashInfo);

  //调用crashApplication函数

 crashApplication(r, crashInfo);

}

[-->ActivityManagerService.java::crashApplication]

private void crashApplication(ProcessRecord r,

                   ApplicationErrorReport.CrashInfocrashInfo) {

   longtimeMillis = System.currentTimeMillis();

   //从应用进程传递过来的crashInfo中获取相关信息

   StringshortMsg = crashInfo.exceptionClassName;

   StringlongMsg = crashInfo.exceptionMessage;

   StringstackTrace = crashInfo.stackTrace;

   if(shortMsg != null && longMsg != null) {

      longMsg = shortMsg + ": " + longMsg;

     } elseif (shortMsg != null) {

      longMsg = shortMsg;

   }

  AppErrorResult result = new AppErrorResult();

  synchronized (this) {

      if(mController != null) {

         //通知监视者。目前仅MonkeyTest中会为AMS设置监听者。测试过程中可设定是否一检测

         //到App Crash即停止测试。另外，Monkey测试也会将App Crash信息保存起来

        //供开发人员分析

       }

      final long origId =Binder.clearCallingIdentity();

      if (r!= null && r.instrumentationClass != null) {

       ......// instrumentationClass不为空的情况

      }

      //①调用makeAppCrashingLocked处理，如果返回false，则整个处理流程完毕

     if (r == null || !makeAppCrashingLocked(r,shortMsg, longMsg,

                        stackTrace)) {

         ......

       return;

      }

     Message msg = Message.obtain();

     msg.what = SHOW\_ERROR\_MSG;

     HashMap data = new HashMap();

     data.put("result", result);

     data.put("app", r);

     msg.obj = data;

    //发送SHOW\_ERROR\_MSG消息给mHandler，默认处理是弹出一个对话框，提示用户某进程   //崩溃（Crash），用户可以选择“退出”或 “退出并报告”

     mHandler.sendMessage(msg);

      ......

   }//synchronized(this)结束

   //下面这个get函数是阻塞的，直到用户处理了对话框为止。注意，此处涉及两个线程：

 //handleApplicationCrash函数是在Binder调用线程中处理的，而对话框则是在mHandler所

  //在线程中处理的

   int res =result.get();

   IntentappErrorIntent = null;

  synchronized (this) {

     if (r!= null)

        mProcessCrashTimes.put(r.info.processName, r.info.uid,

                       SystemClock.uptimeMillis());

   if (res== AppErrorDialog.FORCE\_QUIT\_AND\_REPORT)

      //createAppErrorIntentLocked返回一个Intent，该Intent的Action是

      //"android.intent.action.APP\_ERROR"，指定接收者是app. errorReportReceiver

      //成员，该成员变量在关键点makeAppCrashingLocked中被设置

       appErrorIntent =createAppErrorIntentLocked(r, timeMillis,

                       crashInfo);

   }//synchronized(this)结束

   if(appErrorIntent != null) {

       try {//启动能处理APP\_ERROR的应用进程，目前的源码中还没有地方处理它

             mContext.startActivity(appErrorIntent);

       } ......

    }

 }

以上代码中还有一个关键函数makeAppCrashingLocked，其代码如下：

[-->ActivityManagerService.java::makeAppCrashingLocked]

private booleanmakeAppCrashingLocked(ProcessRecord app,

           String shortMsg, String longMsg, String stackTrace) {

  app.crashing = true;

   //生成一个错误报告，存放在crashingReport变量中

  app.crashingReport = generateProcessError(app,

               ActivityManager.ProcessErrorStateInfo.CRASHED, null, shortMsg,

               longMsg, stackTrace);

   /\*

    在上边代码中，我们知道系统会通过APP\_ERROR Intent启动一个Activity去处理错误报告，

    实际上在此之前，系统需要为它找到指定的接收者（如果有）。代码在startAppProblemLocked

    中，此处简单描述该函数的处理过程如下：

    1 先查询Settings Secure表中“send\_action\_app\_error”是否使能，如果没有使能，则

      不能设置指定接收者

    2 通过PKMS查询安装此Crash应用的应用是否能接收APP\_ERROR Intent。必须注意

     安装此应用的应用（例如通过“安卓市场”安装了该Crash应用）。如果没有，则转下一步处理

    3 查询系统属性“ro.error.receiver.system.apps”是否指定了APP\_ERROR处理者，如果

      没有，则转下一步处理

    4 查询系统属性“ro.error.receiver.default”是否指定了默认的处理者

    5 处理者的信息保存在app的errorReportReceiver变量中

    另外，如果该Crash应用正好是串行广播发送处理中的一员，则需要结束它的处理流程以

    保证后续广播处理能正常执行。读者可参考skipCurrentReceiverLocked函数

  \*/

  startAppProblemLocked(app);

  app.stopFreezingAllLocked();

   //调用handleAppCrashLocked做进一步处理。读者可自行阅读

   returnhandleAppCrashLocked(app);

 }

当App的Crash处理完后，事情并未就此结束，因为该应用进程退出后，之前AMS为它设置的讣告接收对象将被唤醒。接下来介绍AppDeathRecipientbinderDied的处理流程。

6.7.3  AppDeathRecipient binderDied分析

1.  binderDied函数分析

[-->ActvityManagerService.java::AppDeathRecipientbinderDied]

public void binderDied() {

  //注意，该函数也是通过Binder线程调用的，所以此处要加锁

 synchronized(ActivityManagerService.this) {

       appDiedLocked(mApp, mPid, mAppThread);

   }

 }

最终的处理函数是appDiedLocked，其中所传递的3个参数保存了对应死亡进程的信息。来看appDiedLocked的代码：：

[-->ActvityManagerService.java::appDiedLocked]

final void appDiedLocked(ProcessRecord app, intpid,

                                                IApplicationThread thread) {

   ......

   if(app.pid == pid && app.thread != null &&

          app.thread.asBinder() == thread.asBinder()) {

   //当内存低到水位线时，LMK模块也会杀死进程。对AMS来说，需要区分进程死亡是LMK导致的

   //还是其他原因导致的。App instrumentationClass一般都为空，故此处doLowMem为true

    booleandoLowMem = app.instrumentationClass == null;

    //①下面这个函数非常重要

   handleAppDiedLocked(app, false, true);

    if(doLowMem) {

      boolean haveBg = false;

      //如果系统中还存在oom\_adj大于HIDDEN\_APP\_MIN\_ADJ的进程，就表明不是LMK模块因

      //内存不够而导致进程死亡的

       for(int i=mLruProcesses.size()-1; i>=0; i--) {

          ProcessRecord rec = mLruProcesses.get(i);

          if (rec.thread != null && rec.setAdj >=

                                           ProcessList.HIDDEN\_APP\_MIN\_ADJ){

                haveBg = true;//还有后台进程，故可确定系统内存尚未吃紧

                break;

            }

        }//for循环结束

      //如果没有后台进程，表明系统内存已吃紧

      if(!haveBg) {

         long now = SystemClock.uptimeMillis();

          for(int i=mLruProcesses.size()-1; i>=0; i--) {

               .....//将这些进程按一定规则加到mProcessesToGc中，尽量保证

              //heavy/important/visible/foreground的进程位于mProcessesToGc数组

              //的前端

          }//for循环结束

          /\*

           发送GC\_BACKGROUND\_PROCESSES\_MSG消息给mHandler，该消息的处理过程就是：

           调用应用进程的scheduleLowMemory或processInBackground函数。其中，

           scheduleLowMemory将触发onLowMemory回调被调用，而processInBackground将

           触发应用进程进行一次垃圾回收

           读者可自行阅读该消息的处理函数performAppGcsIfAppropriateLocked

          \*/

         scheduleAppGcsLocked();

       }// if(!haveBg)判断结束

    }//if(doLowMem)判断结束

 }

以上代码中有一个关键函数handleAppDiedLocked，下面来看它的处理过程。

2.  handleAppDiedLocked函数分析

[-->ActivityManagerService.java::handleAppDiedLocked]

private final voidhandleAppDiedLocked(ProcessRecord app,

           boolean restarting, boolean allowRestart) {

   //①在本例中，传入的参数为restarting=false, allowRestart=true

  cleanUpApplicationRecordLocked(app, restarting, allowRestart, -1);

   if(!restarting) {

       mLruProcesses.remove(app);

   }

   ......//下面还有一部分代码处理和Activity相关的收尾工作，读者可自行阅读

}

重点看上边代码中的cleanUpApplicationRecordLocked函数，该函数的主要功能就是处理Service、ContentProvider及BroadcastReceiver相关的收尾工作。先来看Service方面的工作。

（1） cleanUpApplicationRecordLocked之处理Service

[-->ActivityManagerService.java::cleanUpApplicationRecordLocked]

 privatefinal void cleanUpApplicationRecordLocked(ProcessRecord app,

       boolean restarting, boolean allowRestart, int index) {

   if (index>= 0) mLruProcesses.remove(index);

   mProcessesToGc.remove(app);

   //如果该Crash进程有对应打开的对话框，则关闭它们，这些对话框包括crash、anr和wait等

   if(app.crashDialog != null) {

       app.crashDialog.dismiss();

       app.crashDialog = null;

   }

   ......//处理anrDialog、waitDialog

   //清理app的一些参数

  app.crashing = false;

  app.notResponding = false;

   ......

   //处理该进程中所驻留的Service或它和别的进程中的Service建立的Connection关系

   //该函数是AMS Service处理流程中很重要的一环，读者要仔细阅读

   killServicesLocked(app,allowRestart);

cleanUpApplicationRecordLocked函数首先处理几个对话框（dialog），然后调用killServicesLocked函数做相关处理。作为Service流程的一部分，读者需要深入研究。

（2） cleanUpApplicationRecordLocked之处理ContentProvider

再来看cleanUpApplicationRecordLocked下一阶段的工作，主要和ContentProvider有关。

[-->ActivityManagerService.java::cleanUpApplicationRecordLocked]

   booleanrestart = false;

   int NL = mLaunchingProviders.size();

   if(!app.pubProviders.isEmpty()) {

       //得到该进程中发布的ContentProvider信息

       Iterator<ContentProviderRecord> it =

                          app.pubProviders.values().iterator();

       while(it.hasNext()) {

         ContentProviderRecord cpr = it.next();

          cpr.provider = null;

         cpr.proc = null;

         int i = 0;

          if(!app.bad && allowRestart) {

              for (; i<NL; i++) {

                 /\*

                   如果有客户端进程在等待这个已经死亡的ContentProvider,则系统会

                   尝试重新启动它，即设置restart变量为true

                \*/

                 if (mLaunchingProviders.get(i) == cpr) {

                       restart = true;

                        break;

                  }

              }//for循环结束

          } else  i = NL;

          if(i >= NL) {

             /\*

              如果没有客户端进程等待这个ContentProvider，则调用下面这个函数处理它，我们

              在卷I的第10章曾提过一个问题，即ContentProvider进程被杀死

              后，系统该如何处理那些使用了该ContentProvider的客户端进程。例如，Music和

              MediaProvider之间有交互，如果杀死了MediaProvider，Music会怎样呢？

              答案是系统会杀死Music，证据就在removeDyingProviderLocked函数

              中，读者可自行阅读其内部处理流程

             \*/

             removeDyingProviderLocked(app, cpr);

             NL = mLaunchingProviders.size();

           }

      }// while(it.hasNext())循环结束

    app.pubProviders.clear();

   }

   //下面这个函数的功能是检查本进程中的ContentProvider是否存在于

  // mLaunchingProviders中，如果存在，则表明有客户端在等待，故需考虑是否重启本进程或者

  //杀死客户端（当死亡进程变成bad process的时，需要杀死客户端）

   if(checkAppInLaunchingProvidersLocked(app, false)) restart = true;

   ......

从以上的描述中可知，ContentProvider所在进程和其客户端进程实际上有着非常紧密而隐晦（之所以说其隐晦，是因为SDK中没有任何说明）的关系。在目前软件开发追求模块间尽量保持松耦合关系的大趋势下，Android中的ContentProvider和其客户端这种紧耦合的设计思路似乎不够明智。不过，这种设计是否是不得已而为之呢？读者不妨探讨一下，如果有更合适的解决方案，期待能一起分享。

（3） cleanUpApplicationRecordLocked之处理BroadcastReceiver

[-->ActivityManagerService.java::cleanUpApplicationRecordLocked]

  skipCurrentReceiverLocked(app);

    //从AMS中去除接收者

   if(app.receivers.size() > 0) {

      Iterator<ReceiverList> it = app.receivers.iterator();

       while(it.hasNext()) {

          removeReceiverLocked(it.next());

        }

       app.receivers.clear();

   }

   if(mBackupTarget != null && app.pid == mBackupTarget.app.pid) {

         //处理Backup信息

    }

    mHandler.obtainMessage(DISPATCH\_PROCESS\_DIED, app.pid,

                                   app.info.uid,null).sendToTarget();

    //注意该变量名为restarting，前面设置为restart.

    if(restarting) return;

    if(!app.persistent) {

        mProcessNames.remove(app.processName, app.info.uid);

         if(mHeavyWeightProcess == app) {

             ......//处理HeavyWeightProcess

           }

      } elseif (!app.removed) {

         if(mPersistentStartingProcesses.indexOf(app) < 0) {

              mPersistentStartingProcesses.add(app);

               restart = true;

          }

      }

    mProcessesOnHold.remove(app);

     if (app== mHomeProcess)  mHomeProcess = null;

     if(restart) {//如果需要重启，则调用startProcessLocked处理它

          mProcessNames.put(app.processName, app.info.uid, app);

         startProcessLocked(app, "restart", app.processName);

     } elseif (app.pid > 0 && app.pid != MY\_PID) {

        synchronized (mPidsSelfLocked) {

             mPidsSelfLocked.remove(app.pid);

             mHandler.removeMessages(PROC\_START\_TIMEOUT\_MSG, app);

          }

         app.setPid(0);

     }

 }

在这段代码中，除了处理BrodcastReceiver方面的工作外，还包括其他方面的收尾工作。最后，如果要重启该应用，则需调用startProcessLocked函数进行处理。这部分代码不再详述，读者可自行阅读。

6.7.4  App的Crash处理总结

分析完整个处理流程，有些读者或许会咂舌。应用进程的诞生是一件很麻烦的事情，没想到应用进程的善后工作居然也很费事，希望各个应用进程能活得更稳健点儿。

图6-24展示了应用进程进行Crash处理的流程。

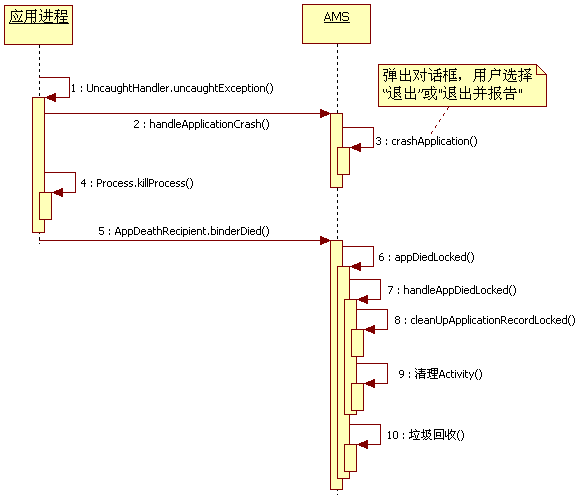


图6-24  应用进程的Crash处理流程

6.7  本章学习指导

本章内容较为复杂，即使用了这么长的篇幅来讲解AMS，依然只能覆盖其中一部分内容。读者在阅读本章时，一定要注意文中的分析脉络，以搞清楚流程为主旨。以下是本章的思路总结：

·  首先搞清楚AMS初创时期的一系列流程，这对理解Android运行环境和系统启动流程等很有帮助。

·  搞清楚一个最简单的情形下Activity启动所历经的磨难。这部分流程最复杂，建议读者在搞清书中所阐述内容的前提下结合具体问题进行分析。

·  BroadcastReceiver的处理流程相对简单。读者务必理解AMS发送广播的处理流程，这对实际工作非常有帮助。例如最近在处理一个广播时，发现静态注册的广播接收者收到广播的时间较长，研究了AMS广播发送的流程后，将其改成了动态注册，结果响应速度就快了很多。

·  关于Service的处理流程希望读者根据流程图自行分析和研究。

·  AMS中的进程管理这一部分内容最难看懂。此处有多方面的原因，笔者觉得和缺乏相关说明有重要关系。建议读者只了解AMS进程管理的大概面貌即可。另外，建议读者不要试图通过修改这部分代码来优化Android的运行效率。进程调度规则向来比较复杂，只有在大量实验的基础上才能得到一个合适的模型。

·  AMS在处理应用进程的Crash及死亡的工作上也是不遗余力的。这部分工作相对比较简单，相信读者能轻松掌握。

由于精力和篇幅的原因，AMS中还有很多精彩的内容未能涉及，建议读者在本章学习基础上，根据具体情况继续深入研究。

6.8  本章小结

本章对AMS进行有针对性的分析：

·  先分析了AMS的创建及初始化过程。

·  以启动一个Activity为例，分析了Activity的启动，应用进程的创建等一系列比较复杂的处理流程。

·  介绍了广播接收者及广播发送的处理流程。Service的处理流程读者可根据流程图并结合代码自行开展研究。

·  还介绍了AMS中的进程管理及相关的知识。重点是在AMS中对LRU、oom\_adj及scheduleGroup的调整。

·  最后介绍了应用进程Crash及死亡后，AMS的处理流程。