第一章 启动流程 2

1 应用进程 2

1.1 startActivity 2

1.2 startActivity 2

1.3 startActivityForResult 3

1.3.1 cancelInputsAndStartExitTransition 4

1.4 execStartActivity 5

1.5 startActivity 7

2. 进入system\_server进程 9

2.1 startActivity 9

2.2 startActivityAsUser 10

2.3 startActivityMayWait 11

2.3.1 resolveIntent 15

2.3.2 resolveActivity 16

2.4 startActivityLocked 17

2.4.1 getRecordForAppLocked 25

2.4.2 ActivityRecord 26

2.5 startActivityUnchecked 31

2.5.1 setInitialState 40

2.5.1.1 adjustLaunchFlagsToDocumentMode 43

2.5.1.2 sendNewTaskResultRequestIfNeeded 44

2.5.2 getReusableIntentActivity 45

2.5.3 deliverNewIntentLocked 46

2.5.4 setTaskFromReuseOrCreateNewTask 47

2.5.4.1 computeStackFocus Activity栈的查找和创建 48

2.5.4.2 55

2.5.5 56

2.5.5.1 检查是否需要生成UriPermission 57

2.5.5.2 生成UriPermission 60

2.6 startActivityLocked 配置activity 62

2.6.1 addActivityToTop 添加目标activity到目标task 的top位置 67

2.6.2 创建目标activity对应的AppWindowToken以及相应的WM端Task对象 69

2.6.3 合成startingView 71

2.7 resumeFocusedStackTopActivityLocked 正式启动activity 72

2.8 startSpecificActivityLocked 启动activity或者启动进程 85

2.9 91

2.9.1 startProcessLocked 95

2.9.2 102

3 ActivityThread 应用主进程的启动 103

3.1 类的初始化 103

3.2 main方法的执行 104

3.2.1 105

3.2.1.1 创建Looper和MessageQueue 106

3.3 进程attach 106

3.3.1 RuntimeInit 109

3.3.1.1 RuntimeInit 关于异常捕获部分的初始化 109

3.3.1.2 RuntimeInit 异常捕获初始化 109

3.3.1.3 未捕获异常处理handler 110

3.1.3.4 wtf异常处理 112

3.4 AMS进程attach 114

附录 120

一 名词解释 120

二 EventLogs 121

# 启动流程

## 1 应用进程

### 1.1 startActivity

[android/app/Activity.java]

public void startActivity(Intent intent) {  
 this.startActivity(intent, null);  
}

描述：启动新的Activity，

参数：

intent代表要启动的新的activity

### 1.2 startActivity

public void startActivity(Intent intent, @Nullable Bundle options) {  
 if (options != null) {  
 startActivityForResult(intent, -1, options);  
 } else {  
 // Note we want to go through this call for compatibility with  
 // applications that may have overridden the method.  
 startActivityForResult(intent, -1);  
 }  
}

描述：最终会调用到startActivityForResult

参数：

intent代表要启动的新的activity

options 启动Activity的一些可选参数，即转场动画，可以配置一些启动效果，构建方法var options = ActivityOptions.makeThumbnailScaleUpAnimation(findViewById(R.id.button), BitmapFactory.decodeResource(resources, R.drawable.test\_bg), 20, 20).toBundle()，细节可以参考ActivityOptions

### 1.3 startActivityForResult

public void startActivityForResult(@RequiresPermission Intent intent, int requestCode,  
 @Nullable Bundle options) {  
 if (mParent == null) {  
 Instrumentation.ActivityResult ar =  
 mInstrumentation.execStartActivity(  
 this, mMainThread.getApplicationThread(), mToken, this,  
 intent, requestCode, options); //参考1.4  
 if (ar != null) {  
 mMainThread.sendActivityResult(  
 mToken, mEmbeddedID, requestCode, ar.getResultCode(),  
 ar.getResultData());   
 }  
 if (requestCode >= 0) {  
 // If this start is requesting a result, we can avoid making  
 // the activity visible until the result is received. Setting  
 // this code during onCreate(Bundle savedInstanceState) or onResume() will keep the  
 // activity hidden during this time, to avoid flickering.  
 // This can only be done when a result is requested because  
 // that guarantees we will get information back when the  
 // activity is finished, no matter what happens to it.  
 mStartedActivity = true;  
 }  
  
 cancelInputsAndStartExitTransition(options); //参考1.3.1  
 // *TODO Consider clearing/flushing other event sources and events for child windows.* } else {  
 if (options != null) {  
 mParent.startActivityFromChild(this, intent, requestCode, options);  
 } else {  
 // Note we want to go through this method for compatibility with  
 // existing applications that may have overridden it.  
 mParent.startActivityFromChild(this, intent, requestCode);  
 }  
 }  
}

描述：

mParent 表示当前Activity是否为child activity，如果是，则会调用startActivityFromChild，后记>

mInstrumentation 为Instrumentation对象，在activity启动的时候在ActivityThread调用performLaunchActivity的时候调用activity.attach(…)赋值的，具体参考activity启动之后的介绍。

mMainThread 同mInstrumentation赋值方式，代表ActivityThread，即所谓的主线程

mToken 来自ActivityRecord，后续分析，是IBinder类型，代表当前activity

mEmbeddedID 用于ActivityGroup来管理Activity嵌入的，一般用于穿戴式设备的，与activity定义的属性android:allowEmbedded

sendActivityResult 后记>

参数：

requestCode 如果大于0，表示启动的Activity被结束后，会回调onActivityResult()

其它参数同上一步startActivity

#### 1.3.1 cancelInputsAndStartExitTransition

private void cancelInputsAndStartExitTransition(Bundle options) {  
 final View decor = mWindow != null ? mWindow.peekDecorView() : null;  
 if (decor != null) {  
 decor.cancelPendingInputEvents();  
 }  
 if (options != null && !isTopOfTask()) {  
 mActivityTransitionState.startExitOutTransition(this, options);  
 }  
}

描述：

cancelPendingInputEvents取消Input事件，针对decorView进行，该方法会遍历清除所有子View的InputEvent，并且会清除点击事件的回调，包括LongClick。

startExitOutTransition执行activity退出的transition动作，该动作就是之前提到的ActivityOptions定义的，如果没有，则忽略。主要是针对decor进行的。

### 1.4 execStartActivity

[android/app/Instrumentation.java]

public ActivityResult execStartActivity(  
 Context who, IBinder contextThread, IBinder token, Activity target,  
 Intent intent, int requestCode, Bundle options) {  
 IApplicationThread whoThread = (IApplicationThread) contextThread;  
 Uri referrer = target != null ? target.onProvideReferrer() : null;  
 if (referrer != null) {  
 intent.putExtra(Intent.*EXTRA\_REFERRER*, referrer);  
 }  
 if (mActivityMonitors != null) {  
 synchronized (mSync) {  
 final int N = mActivityMonitors.size();  
 for (int i=0; i<N; i++) {  
 final ActivityMonitor am = mActivityMonitors.get(i);  
 if (am.match(who, null, intent)) {  
 am.mHits++;  
 if (am.isBlocking()) {  
 return requestCode >= 0 ? am.getResult() : null;  
 }  
 break;  
 }  
 }  
 }  
 }  
 try {  
 intent.migrateExtraStreamToClipData();  
 intent.prepareToLeaveProcess(who);  
 int result = ActivityManagerNative.*getDefault*()  
 .startActivity(whoThread, who.getBasePackageName(), intent,  
 intent.resolveTypeIfNeeded(who.getContentResolver()),  
 token, target != null ? target.mEmbeddedID : null,  
 requestCode, 0, null, options); //参考1.5  
 *checkStartActivityResult*(result, intent);  
 } catch (RemoteException e) {  
 throw new RuntimeException("Failure from system", e);  
 }  
 return null;  
}

描述：

referrer 代表发起者的一些信息，可以传递给被调起的activity，一般为空，如果要用，需要重写Activity的方法onProvideReferrer，如果有数据，会直接添加到目标activity的intent中

mActivityMonitors 维护ActivityMonitor对象列表 用来管理一些特殊的activity，当要启动的activity在该列表存在，那么就在判断该activity是否被阻止启动，如果是阻止的，则直接返回，否则按正常启动流程走。关于ActivityMonitor应该是需要Activity自己定义是否需要放到该列表里面，以及自己定义是否阻止启动。

migrateExtraStreamToClipData 针对Intent的以下几个action的EXTRA\_STREAM数据迁移到ClipData中，Intent会持有ClipData对象来保存当前Inent包含的相关extra信息，被迁移到ClipData中的数据将不再进行一些权限限制，主要用来FLAG\_GRANT\_READ\_URI\_PERMISSION和FLAG\_GRANT\_WRITE\_URI\_PERMISSION。

ACTION\_CHOOSER，ACTION\_SEND，ACTION\_SEND\_MULTIPLE等

prepareToLeaveProcess 可能离开当前进程的一些准备工作

ActivityManagerNative.*getDefault*()获取AMS服务在当前UI线程的Binder带来AMP。

*checkStartActivityResult检查Activity启动结果，如果异常则抛出相关异常，参考ActivityManager.* *START\_INTENT\_NOT\_RESOLVED等的描述*

参数：

who 发起startActivity者，即当前activity对应的context

contextThread 发起者主线程里面的ApplicationThread线程对象，该对象在ActivityThread创建的时候初始化，继承自ApplicationThreadNative，是Binder线程的服务端，对应于运行在system\_server进程里面的ApplicationThreadProxy线程，该对象将用于system\_server到UI线程的通信。

token 当前activity 的token

target 当前activity

intent 要启动的Activity的intent

requestCode 同前

options 同前

### 1.5 startActivity

[ActivityManagerProxy 位于ActivityManagerNative.java]

public int startActivity(IApplicationThread caller, String callingPackage, Intent intent,  
 String resolvedType, IBinder resultTo, String resultWho, int requestCode,  
 int startFlags, ProfilerInfo profilerInfo, Bundle options) throws RemoteException {  
 Parcel data = Parcel.*obtain*();  
 Parcel reply = Parcel.*obtain*();  
 data.writeInterfaceToken(IActivityManager.*descriptor*);  
 data.writeStrongBinder(caller != null ? caller.asBinder() : null);  
 data.writeString(callingPackage);  
 intent.writeToParcel(data, 0);  
 data.writeString(resolvedType);  
 data.writeStrongBinder(resultTo);  
 data.writeString(resultWho);  
 data.writeInt(requestCode);  
 data.writeInt(startFlags);  
 if (profilerInfo != null) {  
 data.writeInt(1);  
 profilerInfo.writeToParcel(data, Parcelable.*PARCELABLE\_WRITE\_RETURN\_VALUE*);  
 } else {  
 data.writeInt(0);  
 }  
 if (options != null) {  
 data.writeInt(1);  
 options.writeToParcel(data, 0);  
 } else {  
 data.writeInt(0);  
 }  
 mRemote.transact(*START\_ACTIVITY\_TRANSACTION*, data, reply, 0);  
 reply.readException();  
 int result = reply.readInt();  
 reply.recycle();  
 data.recycle();  
 return result;  
}

参数：

caller 发起方的ApplicationThread对象，属于Binder线程

callingPackage 发起方报名

intent 要启动的activity

resolvedType 一般供framework使用，普通应用不会用到

resultTo 发起方activity token

resultWho 发起方activity对应的mEmbeddedID

其它的参考前面解释，其中startFlags和profilerInfo没有用到, startFlags是AMS用来检查一些启动配置项的，比如debug配置，在后续解析activity信息的时候再分析，profilerInfo也是类似的，它主要是配置堆信息相关的调试。但是检查Instrumentation类中对于startActivity的调用，这两个值一直为空。

描述：

ActivityManagerProxy是AMS在UI进程的Binder客户端类，该调用会触发ActivityManagerNative的onTransact方法，见如下代码。具体需要阅读Binder相关知识。

public boolean onTransact(int code, Parcel data, Parcel reply, int flags)  
 throws RemoteException {  
 switch (code) {  
 case START\_ACTIVITY\_TRANSACTION:  
 {  
 data.enforceInterface(IActivityManager.*descriptor*);  
 IBinder b = data.readStrongBinder();  
 IApplicationThread app = ApplicationThreadNative.*asInterface*(b);  
 String callingPackage = data.readString();  
 Intent intent = Intent.*CREATOR*.createFromParcel(data);  
 String resolvedType = data.readString();  
 IBinder resultTo = data.readStrongBinder();  
 String resultWho = data.readString();  
 int requestCode = data.readInt();  
 int startFlags = data.readInt();  
 ProfilerInfo profilerInfo = data.readInt() != 0  
 ? ProfilerInfo.*CREATOR*.createFromParcel(data) : null;  
 Bundle options = data.readInt() != 0  
 ? Bundle.*CREATOR*.createFromParcel(data) : null;  
 int result = startActivity(app, callingPackage, intent, resolvedType,  
 resultTo, resultWho, requestCode, startFlags, profilerInfo, options); //参考2.1  
 reply.writeNoException();  
 reply.writeInt(result);  
 return true;  
 }

ActivityManagerProxy和ActivityManagerNative两个都继承自IActivityManager，是一对Binder C/S体。

## 2. 进入system\_server进程

### 2.1 startActivity

[ActivityManagerService extends ActivityManagerNative]

@Override  
public final int startActivity(IApplicationThread caller, String callingPackage,  
 Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,  
 int startFlags, ProfilerInfo profilerInfo, Bundle bOptions) {  
 return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,  
 resultWho, requestCode, startFlags, profilerInfo, bOptions,  
 UserHandle.getCallingUserId()); //参考2.2  
}

参数：

caller 发起方的ApplicationThread对象，属于Binder线程

callingPackage 发起方包名

intent 要启动的activity

resolvedType 一般供framework使用，普通应用不会用到

resultTo 发起方activity token

resultWho 发起方activity

其实各个参数与AMP里面是一一对应的。

### 2.2 startActivityAsUser

[AMS]

public final int startActivityAsUser(IApplicationThread caller, String callingPackage,  
 Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,  
 int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int userId) {  
 enforceNotIsolatedCaller("startActivity");

//做一次检查，查看调用者userId和参数传递进来的userId是否相同，相同直接赋值，不同的话会产生一个新的userId。关于这个问题，需要研究多用户的管理相关知识。  
 userId = mUserController.handleIncomingUser(Binder.getCallingPid(), Binder.getCallingUid(),  
 userId, false, ALLOW\_FULL\_ONLY, "startActivity", null);  
 // TODO: Switch to user app stacks here.  
 return mActivityStarter.startActivityMayWait(caller, -1, callingPackage, intent,  
 resolvedType, null, null, resultTo, resultWho, requestCode, startFlags,  
 profilerInfo, null, null, bOptions, false, userId, null, null);// 参考2.3  
}

参数：

userId 为调用者的userId，通过UserHandle.getCallingUserId()获取，然后会先调用到Binder.getCallingUid()，去native获取当前进程的uid，然后再调用getUserId(int uid)返回该uid对应的userId，如果当前不支持多用户，则直接返回UserHandle.USER\_SYSTEM，其值为0，如果支持多用户则返回uid / PER\_USER\_RANGE，结果可能为1，2，3 等等。PER\_USER\_RANGE的值为10000，代表多用户的时候每个用户的空间里面允许存储至多10000个uid，每个应用都有一个uid。

其它与前一步完全相同

描述：

mActivityStarter 是ActivityStarter对象，在AMS初始化的时候定义，ActivityStarter初始化会持有ActivityStackSupervisor对象，ActivityStackSupervisor对象也是在AMS初始化的时候定义

### 2.3 startActivityMayWait

[ActivityStarter.java]

final int startActivityMayWait(IApplicationThread caller, int callingUid,  
 String callingPackage, Intent intent, String resolvedType,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 IBinder resultTo, String resultWho, int requestCode, int startFlags,  
 ProfilerInfo profilerInfo, IActivityManager.WaitResult outResult, Configuration config,  
 Bundle bOptions, boolean ignoreTargetSecurity, int userId,  
 IActivityContainer iContainer, TaskRecord inTask) {  
 // Refuse possible leaked file descriptors  
 if (intent != null && intent.hasFileDescriptors()) {  
 throw new IllegalArgumentException("File descriptors passed in Intent");  
 }

// mSupervisor 为初始化时携带的ActivityStackSupervisor对象，此处通知ActivityMetricsLogger来记录activity开始launch的时间，属于最早的记录了，ActivityMetricsLogger最后会通过EventLogTags来记录log。EventLogTags是由logcat解析文件EventLogTags.logtags生成，关于logtags类型文件请参考/system/core/logcat/event.logcats的相关描述  
 mSupervisor.mActivityMetricsLogger.notifyActivityLaunching();

//intent的component是否为空，component是ComponentName类型，由包名和类名组成，只有android的四大组件可用。  
 boolean componentSpecified = intent.getComponent() != null;  
  
 // Save a copy in case ephemeral needs it //用于肯能临时需要  
 final Intent ephemeralIntent = new Intent(intent);  
 // Don't modify the client's object!  
 intent = new Intent(intent);  
  
 ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType, userId); //参考2.3.1，根据intent解析ResolveInfo  
 if (rInfo == null) { //特殊情况  
 …  
 }  
 // Collect information about the target of the Intent.  
 ActivityInfo aInfo = mSupervisor.resolveActivity(intent, rInfo, startFlags, profilerInfo);//参考2.3.2根据ResolveInfo获取ActivityInfo  
 //把ActivityOptions从之前传进来的bOptions 从bundle中解析出来  
 ActivityOptions options = ActivityOptions.fromBundle(bOptions);

//这个为空  
 ActivityStackSupervisor.ActivityContainer container =  
 (ActivityStackSupervisor.ActivityContainer)iContainer;  
 synchronized (mService) { //线程安全

//container为空  
 if (container != null && container.mParentActivity != null &&  
 container.mParentActivity.state != RESUMED) {  
 // Cannot start a child activity if the parent is not resumed.  
 return ActivityManager.START\_CANCELED;  
 }  
 final int realCallingPid = Binder.getCallingPid();  
 final int realCallingUid = Binder.getCallingUid();  
 int callingPid;  
 if (callingUid >= 0) {  
 callingPid = -1;  
 } else if (caller == null) {  
 callingPid = realCallingPid;  
 callingUid = realCallingUid;  
 } else {

//因为传进来的callingUid为-1，所以会走到这儿  
 callingPid = callingUid = -1;  
 }  
  
 final ActivityStack stack;  
 if (container == null || container.mStack.isOnHomeDisplay()) {

// mFocusedStack 代表当前获取到焦点的Activity或者正在启动下一个Activity的Activity所在的ActivityStack  
 stack = mSupervisor.mFocusedStack;  
 } else {  
 stack = container.mStack;  
 }

//config 为空  
 stack.mConfigWillChange = config != null && mService.mConfiguration.diff(config) != 0;  
 if (DEBUG\_CONFIGURATION) Slog.v(TAG\_CONFIGURATION,  
 “Starting activity when config will change = “ + stack.mConfigWillChange);  
  
 final long origId = Binder.clearCallingIdentity();  
// PRIVATE\_FLAG\_CANT\_SAVE\_STATE 一种特殊的应用进程，即heavy\_weight process重量级进程，系统中没有查到使用的。如果被设定为重量级进程，其优先级就比较高。  
 if (aInfo != null &&  
 (aInfo.applicationInfo.privateFlags  
 & ApplicationInfo.PRIVATE\_FLAG\_CANT\_SAVE\_STATE) != 0) {  
 // This may be a heavy-weight process! Check to see if we already  
 // have another, different heavy-weight process running.  
 。。。  
 }  
  
 final ActivityRecord[] outRecord = new ActivityRecord[1];  
 int res = startActivityLocked(caller, intent, ephemeralIntent, resolvedType,  
 aInfo, rInfo, voiceSession, voiceInteractor,  
 resultTo, resultWho, requestCode, callingPid,  
 callingUid, callingPackage, realCallingPid, realCallingUid, startFlags,  
 options, ignoreTargetSecurity, componentSpecified, outRecord, container,  
 inTask); //参考2.4 Locked代表线程安全，因为前面加了mService锁  
  
 Binder.restoreCallingIdentity(origId);  
  
 if (stack.mConfigWillChange) {  
 // If the caller also wants to switch to a new configuration,  
 // do so now. This allows a clean switch, as we are waiting  
 // for the current activity to pause (so we will not destroy  
 // it), and have not yet started the next activity.  
 mService.enforceCallingPermission(android.Manifest.permission.CHANGE\_CONFIGURATION,  
 “updateConfiguration()”);  
 stack.mConfigWillChange = false;  
 if (DEBUG\_CONFIGURATION) Slog.v(TAG\_CONFIGURATION,  
 “Updating to new configuration after starting activity.”);  
 mService.updateConfigurationLocked(config, null, false);  
 }  
  
 if (outResult != null) {  
 outResult.result = res;  
 if (res == ActivityManager.START\_SUCCESS) {  
 mSupervisor.mWaitingActivityLaunched.add(outResult);  
 do {  
 try {  
 mService.wait();  
 } catch (InterruptedException e) {  
 }  
 } while (outResult.result != START\_TASK\_TO\_FRONT  
 && !outResult.timeout && outResult.who == null);  
 if (outResult.result == START\_TASK\_TO\_FRONT) {  
 res = START\_TASK\_TO\_FRONT;  
 }  
 }  
 if (res == START\_TASK\_TO\_FRONT) {  
 ActivityRecord r = stack.topRunningActivityLocked();  
 if (r.nowVisible && r.state == RESUMED) {  
 outResult.timeout = false;  
 outResult.who = new ComponentName(r.info.packageName, r.info.name);  
 outResult.totalTime = 0;  
 outResult.thisTime = 0;  
 } else {  
 outResult.thisTime = SystemClock.uptimeMillis();  
 mSupervisor.mWaitingActivityVisible.add(outResult);  
 do {  
 try {  
 mService.wait();  
 } catch (InterruptedException e) {  
 }  
 } while (!outResult.timeout && outResult.who == null);  
 }  
 }  
 }  
  
 final ActivityRecord launchedActivity = mReusedActivity != null  
 ? mReusedActivity : outRecord[0];  
 mSupervisor.mActivityMetricsLogger.notifyActivityLaunched(res, launchedActivity);  
 return res;  
 }  
}

参数：

ignoreTargetSecurity 为false

callingUid 为-1

caller，callingPackage，intent，resolvedType，resultTo，resultWho，requestCode，startFlags，profilerInfo, bOptions, userId均与之前的保持一致，其它参数皆为空。

描述：

根据Intent解析ResolveInfo

#### 2.3.1 resolveIntent

[ActivityStackSupervisor.java]

ResolveInfo resolveIntent(Intent intent, String resolvedType, int userId) {  
 return resolveIntent(intent, resolvedType, userId, 0);  
}  
  
ResolveInfo resolveIntent(Intent intent, String resolvedType, int userId, int flags) {  
 try {

//这个会最终调用到PMS的resolveIntent  
 return AppGlobals.getPackageManager().resolveIntent(intent, resolvedType,  
 PackageManager.MATCH\_DEFAULT\_ONLY | flags  
 | ActivityManagerService.STOCK\_PM\_FLAGS, userId);  
 } catch (RemoteException e) {  
 }  
 return null;  
}

参数：

intent 目标intent

resolveType 为空

userId 发起方用户ID

描述：

ResolveInfo 通过IntentFilter解析Intent产生，通常代表从AndroidManifest.xml中收集到的intent信息

PackageManager.MATCH\_DEFAULT\_ONLY代表只解析与android.content.Intent#CATEGORY\_DEFAULT相匹配的intent

ActivityManagerService.STOCK\_PM\_FLAGS指向PackageManager.GET\_SHARED\_LIBRARY\_FILES，对于所有指向PackageManager的调用都会携带这个Flag

resolveIntent 的主要工作有：

判断userId是否在sUserManager中存在，如果不存在，直接返回null，与多用户有关

检查跨用户访问权限，包括检查uid是否为root，system，shell或者普通用户，然后针对不同的用户做不同的检查

调用queryIntentActivitiesInternal来获取所有满足该intent的activity信息，存放到List<ResolveInfo> query中

然后从List<ResolveInfo> query中选择合适的一项并返回

关于intent的解析和ResolveInfo相关信息会在PMS里面再做详细介绍，这里先简单介绍一下这些信息包含者和相关继承关系：

PackageItemInfo： manifest最基本的信息，包含有包名，以及<application>标签下面的一些信息，如logo，name，lableres，metadata等，这里包含的是最最基本的信息，不一定属于哪个标签

---PermissionInfo permission相关，来自于<permission>标签

---PermissionGroupInfo permission相关，来自于<permission-group>标签

---InstrumentationInfo 来自于<instrumentation>标签

---ComponentInfo 功能类似PackageItenInfo，包含了三个子类信息的部分共用信息

------ActivityInfo

------ServiceInfo

------ProviderInfo

---ApplicationInfo来自于<instrumentation>标签，如taskAffinity，processName等

ResolveInfo持有ActivityInfo，ServiceInfo和ProviderInfo对象，以及IntentFilter等。

#### 2.3.2 resolveActivity

[ActivityStackSupervisor.java]

ActivityInfo resolveActivity(Intent intent, ResolveInfo rInfo, int startFlags,  
 ProfilerInfo profilerInfo) {  
 final ActivityInfo aInfo = rInfo != null ? rInfo.activityInfo : null;  
 if (aInfo != null) {  
 // Store the found target back into the intent, because now that  
 // we have it we never want to do this again. For example, if the  
 // user navigates back to this point in the history, we should  
 // always restart the exact same activity.  
 intent.setComponent(new ComponentName(  
 aInfo.applicationInfo.packageName, aInfo.name));  
// 根据startFlags打开一些debug信息，之前我们传进来的startFlags为0  
 // Don't debug things in the system process  
 if (!aInfo.processName.equals("system")) {

//根据startFlags值来记录当前进程的debug状态，这三个配置项可以参考AM的定义。  
 if ((startFlags & ActivityManager.START\_FLAG\_DEBUG) != 0) {  
 mService.setDebugApp(aInfo.processName, true, false);  
 }  
  
 if ((startFlags & ActivityManager.START\_FLAG\_NATIVE\_DEBUGGING) != 0) {  
 mService.setNativeDebuggingAppLocked(aInfo.applicationInfo, aInfo.processName);  
 }  
  
 if ((startFlags & ActivityManager.START\_FLAG\_TRACK\_ALLOCATION) != 0) {  
 mService.setTrackAllocationApp(aInfo.applicationInfo, aInfo.processName);  
 }  
 //设置profiler信息调试信息  
 if (profilerInfo != null) {  
 mService.setProfileApp(aInfo.applicationInfo, aInfo.processName, profilerInfo);  
 }  
 }  
 }  
 return aInfo;  
}

描述：

从ResolvedInfo中获取ActivityInfo

然后就是把找到的ActivityInfo的component信息回写到目标intent里面，为方便后续使用该intent的时候不再从ActivityInfo去获取

### 2.4 startActivityLocked

[ActivityStarter.java]

final int startActivityLocked(IApplicationThread caller, Intent intent, Intent ephemeralIntent,  
 String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,  
 String callingPackage, int realCallingPid, int realCallingUid, int startFlags,  
 ActivityOptions options, boolean ignoreTargetSecurity, boolean componentSpecified,  
 ActivityRecord[] outActivity, ActivityStackSupervisor.ActivityContainer container,  
 TaskRecord inTask) {  
 int err = ActivityManager.START\_SUCCESS; //标识Activity启动成功  
  
 ProcessRecord callerApp = null; //发起方所在进程ProcessRecord  
 if (caller != null) {  
 callerApp = mService.getRecordForAppLocked(caller); //参考2.4.1 查找调用方进程，并把该进程的uid pid赋值给callingPid和callingUid  
 if (callerApp != null) {  
 callingPid = callerApp.pid;  
 callingUid = callerApp.info.uid;  
 } else {  
 Slog.w(TAG, "Unable to find app for caller " + caller  
 + " (pid=" + callingPid + ") when starting: "  
 + intent.toString());  
 err = ActivityManager.START\_PERMISSION\_DENIED;  
 }  
 }  
 //目标activity对应的userId  
 final int userId = aInfo != null ? UserHandle.getUserId(aInfo.applicationInfo.uid) : 0;  
  
 if (err == ActivityManager.START\_SUCCESS) {  
 Slog.i(TAG, "START u" + userId + " {" + intent.toShortString(true, true, true, false)  
 + "} from uid " + callingUid  
 + " on display " + (container == null ? (mSupervisor.mFocusedStack == null ?  
 Display.DEFAULT\_DISPLAY : mSupervisor.mFocusedStack.mDisplayId) :  
 (container.mActivityDisplay == null ? Display.DEFAULT\_DISPLAY :  
 container.mActivityDisplay.mDisplayId)));  
 }  
  
 ActivityRecord sourceRecord = null;  
 ActivityRecord resultRecord = null;  
 if (resultTo != null) {

//查找源sourceRecord，通过遍历目前系统中支持的所有ActivityDisplay来查找，每一个ActivityDisplay都对应一个Display对象，并且持有ArrayList<ActivityStack> mStacks，该对象保存了该Display的所有ActivityStack，遍历这些栈以查询源ActivityRecord是否存在，存在则赋值给sourceRecord和resultRecord  
 sourceRecord = mSupervisor.isInAnyStackLocked(resultTo);  
 if (DEBUG\_RESULTS) Slog.v(TAG\_RESULTS,  
 "Will send result to " + resultTo + " " + sourceRecord);  
 if (sourceRecord != null) {  
 if (requestCode >= 0 && !sourceRecord.finishing) {  
 resultRecord = sourceRecord;  
 }  
 }  
 }  
//获取Activity启动类型  
 final int launchFlags = intent.getFlags();  
  
 if ((launchFlags & Intent.FLAG\_ACTIVITY\_FORWARD\_RESULT) != 0 && sourceRecord != null) {  
 // Transfer the result target from the source activity to the new  
 // one being started, including any failures.  
 if (requestCode >= 0) {  
 ActivityOptions.abort(options);  
 return ActivityManager.START\_FORWARD\_AND\_REQUEST\_CONFLICT;  
 }  
 resultRecord = sourceRecord.resultTo;  
 if (resultRecord != null && !resultRecord.isInStackLocked()) {  
 resultRecord = null;  
 }  
 resultWho = sourceRecord.resultWho;  
 requestCode = sourceRecord.requestCode;  
 sourceRecord.resultTo = null;  
 if (resultRecord != null) {  
 resultRecord.removeResultsLocked(sourceRecord, resultWho, requestCode);  
 }  
 if (sourceRecord.launchedFromUid == callingUid) {  
 // The new activity is being launched from the same uid as the previous  
 // activity in the flow, and asking to forward its result back to the  
 // previous. In this case the activity is serving as a trampoline between  
 // the two, so we also want to update its launchedFromPackage to be the  
 // same as the previous activity. Note that this is safe, since we know  
 // these two packages come from the same uid; the caller could just as  
 // well have supplied that same package name itself. This specifially  
 // deals with the case of an intent picker/chooser being launched in the app  
 // flow to redirect to an activity picked by the user, where we want the final  
 // activity to consider it to have been launched by the previous app activity.  
 callingPackage = sourceRecord.launchedFromPackage;  
 }  
 }  
//异常情况，没有component或者ActivityInfo为空  
 if (err == ActivityManager.START\_SUCCESS && intent.getComponent() == null) {  
 // We couldn't find a class that can handle the given Intent.  
 // That's the end of that!  
 err = ActivityManager.START\_INTENT\_NOT\_RESOLVED;  
 }  
  
 if (err == ActivityManager.START\_SUCCESS && aInfo == null) {  
 // We couldn't find the specific class specified in the Intent.  
 // Also the end of the line.  
 err = ActivityManager.START\_CLASS\_NOT\_FOUND;  
 }  
//语音交互，如果是通过语音方式启动目标acitity ，需要做相应的安全检查

if (err == ActivityManager.START\_SUCCESS && sourceRecord != null  
 && sourceRecord.task.voiceSession != null) {  
 // If this activity is being launched as part of a voice session, we need  
 // to ensure that it is safe to do so. If the upcoming activity will also  
 // be part of the voice session, we can only launch it if it has explicitly  
 // said it supports the VOICE category, or it is a part of the calling app.  
 if ((launchFlags & FLAG\_ACTIVITY\_NEW\_TASK) == 0  
 && sourceRecord.info.applicationInfo.uid != aInfo.applicationInfo.uid) {  
 try {  
 intent.addCategory(Intent.CATEGORY\_VOICE);  
 if (!AppGlobals.getPackageManager().activitySupportsIntent(  
 intent.getComponent(), intent, resolvedType)) {  
 Slog.w(TAG,  
 "Activity being started in current voice task does not support voice: "  
 + intent);  
 err = ActivityManager.START\_NOT\_VOICE\_COMPATIBLE;  
 }  
 } catch (RemoteException e) {  
 Slog.w(TAG, "Failure checking voice capabilities", e);  
 err = ActivityManager.START\_NOT\_VOICE\_COMPATIBLE;  
 }  
 }  
 }  
//语音启动，同上，没有源activity的情况  
 if (err == ActivityManager.START\_SUCCESS && voiceSession != null) {  
 // If the caller is starting a new voice session, just make sure the target  
 // is actually allowing it to run this way.  
 try {  
 if (!AppGlobals.getPackageManager().activitySupportsIntent(intent.getComponent(),  
 intent, resolvedType)) {  
 Slog.w(TAG,  
 "Activity being started in new voice task does not support: "  
 + intent);  
 err = ActivityManager.START\_NOT\_VOICE\_COMPATIBLE;  
 }  
 } catch (RemoteException e) {  
 Slog.w(TAG, "Failure checking voice capabilities", e);  
 err = ActivityManager.START\_NOT\_VOICE\_COMPATIBLE;  
 }  
 }  
  
 final ActivityStack resultStack = resultRecord == null ? null : resultRecord.task.stack;  
  
 if (err != START\_SUCCESS) {  
 if (resultRecord != null) {  
 resultStack.sendActivityResultLocked(  
 -1, resultRecord, resultWho, requestCode, RESULT\_CANCELED, null);  
 }  
 ActivityOptions.abort(options);  
 return err;  
 }  
//权限检查  
 boolean abort = !mSupervisor.checkStartAnyActivityPermission(intent, aInfo, resultWho,  
 requestCode, callingPid, callingUid, callingPackage, ignoreTargetSecurity, callerApp,  
 resultRecord, resultStack, options);  
 abort |= !mService.mIntentFirewall.checkStartActivity(intent, callingUid,  
 callingPid, resolvedType, aInfo.applicationInfo);  
  
 if (mService.mController != null) {  
 try {  
 // The Intent we give to the watcher has the extra data  
 // stripped off, since it can contain private information.  
 Intent watchIntent = intent.cloneFilter();  
 abort |= !mService.mController.activityStarting(watchIntent,  
 aInfo.applicationInfo.packageName);  
 } catch (RemoteException e) {  
 mService.mController = null;  
 }  
 }  
  
 // ActivityStartInterceptor是一个用来管理activity调用startActivityLocked启动过程的一些阻断(intercepting)逻辑的类

mInterceptor.setStates(userId, realCallingPid, realCallingUid, startFlags, callingPackage);  
 mInterceptor.intercept(intent, rInfo, aInfo, resolvedType, inTask, callingPid, callingUid,  
 options);  
 intent = mInterceptor.mIntent;  
 rInfo = mInterceptor.mRInfo;  
 aInfo = mInterceptor.mAInfo;  
 resolvedType = mInterceptor.mResolvedType;  
 inTask = mInterceptor.mInTask;  
 callingPid = mInterceptor.mCallingPid;  
 callingUid = mInterceptor.mCallingUid;  
 options = mInterceptor.mActivityOptions;  
 if (abort) {  
 if (resultRecord != null) {  
 resultStack.sendActivityResultLocked(-1, resultRecord, resultWho, requestCode,  
 RESULT\_CANCELED, null);  
 }  
 // We pretend to the caller that it was really started, but  
 // they will just get a cancel result.  
 ActivityOptions.abort(options);  
 return START\_SUCCESS;  
 }  
  
 //构建一个ActivityRecord，参考2.4.2  
 ActivityRecord r = new ActivityRecord(mService, callerApp, callingUid, callingPackage,  
 intent, resolvedType, aInfo, mService.mConfiguration, resultRecord, resultWho,  
 requestCode, componentSpecified, voiceSession != null, mSupervisor, container,  
 options, sourceRecord);  
 if (outActivity != null) {  
 outActivity[0] = r;  
 }  
  
 if (r.appTimeTracker == null && sourceRecord != null) {  
 // If the caller didn't specify an explicit time tracker, we want to continue  
 // tracking under any it has.  
 r.appTimeTracker = sourceRecord.appTimeTracker;  
 }  
 //拥有焦点的activity所在栈，即处于前台的ActivityStack  
 final ActivityStack stack = mSupervisor.mFocusedStack;

//如果该栈没有处于resume状态的activity或者处于rusume的activity与调用发起方的uid不同，那么认为app发生切换了，即启动发起者目前处于后台。当然会排除语音启动的应用。

//对于后台发起启动activity的动作，需要检查其是否申请了权限android.Manifest.permission.STOP\_APP\_SWITCHES，这个权限的意思是停止app切换策略（即延迟5s），直接切换，该权限一般只有系统应用才有，比如来电，就是从后台直接启动IncomingCall界面。google提倡开发者一般不要开发从后台直接启动Activity的功能，比如从service或者broadcast去启动activity，因为这样对用户影响很大。

//针对这样的启动，系统如果检查到没有申请STOP\_APP\_SWITCHES权限，则会把该请求添加到mPendingActivityLaunches里面，间隔5秒后再启动。  
 if (voiceSession == null && (stack.mResumedActivity == null  
 || stack.mResumedActivity.info.applicationInfo.uid != callingUid)) {  
 if (!mService.checkAppSwitchAllowedLocked(callingPid, callingUid,  
 realCallingPid, realCallingUid, "Activity start")) {  
 PendingActivityLaunch pal = new PendingActivityLaunch(r,  
 sourceRecord, startFlags, stack, callerApp);  
 mPendingActivityLaunches.add(pal);  
 ActivityOptions.abort(options);  
 return ActivityManager.START\_SWITCHES\_CANCELED;  
 }  
 }  
 //是否允许app switch的标记  
 if (mService.mDidAppSwitch) {  
 // This is the second allowed switch since we stopped switches,  
 // so now just generally allow switches. Use case: user presses  
 // home (switches disabled, switch to home, mDidAppSwitch now true);  
 // user taps a home icon (coming from home so allowed, we hit here  
 // and now allow anyone to switch again).  
 mService.mAppSwitchesAllowedTime = 0;  
 } else {  
 mService.mDidAppSwitch = true;  
 }  
//启动pending activity，一般在按home键的时候PWM会调用ActivityManagerNative.getDefault().stopAppSwitches();来设置延迟启动activity的延迟时间，即5s，它最终也会调用到startActivityUnchecked，但是参数doResume为false，doResume的作用后续分析。  
 doPendingActivityLaunchesLocked(false);  
  
 try {

//在启动activity的时候可以先暂停surface更新，此时需要更新数据，具体实现在WMS中  
 mService.mWindowManager.deferSurfaceLayout();  
 err = startActivityUnchecked(r, sourceRecord, voiceSession, voiceInteractor, startFlags,  
 true, options, inTask); //参考2.5  
 } finally {  
 mService.mWindowManager.continueSurfaceLayout();  
 }  
 postStartActivityUncheckedProcessing(r, err, stack.mStackId, mSourceRecord, mTargetStack);  
 return err;  
}

参数：

caller 调用者ApplicationThread

intent 目标intent

ephemeralIntent 之前保存的临时目标intent

resolvedType 一般供framework使用，普通应用不会用到，此处为空

aInfo 目标activity相关信息

rInfo 目标Intent解析出来的信息

voiceSession 为空

voiceInteractor 为空

resultTo 发起方activity token

resultWho 发起方activity对应的mEmbeddedID

requestCode 同前

callingPid -1

callingUid -1

callingPackage 发起方包名

realCallingPid 通过Binder.getCallingPid()获取到的调用方pid

realCallingUid通过Binder.getCallingUid()获取到的调用方uid

startFlags 空

options 同前

ignoreTargetSecurity false

componentSpecified 标识最初传进来的intent是否指定component，即ephemeralIntent是否携带component

outActivity ActivityRecord[]类型的空数组，大小为1

container 为空

inTask 为空

#### 2.4.1 getRecordForAppLocked

final ProcessRecord getRecordForAppLocked(  
 IApplicationThread thread) {  
 if (thread == null) {  
 return null;  
 }  
  
 int appIndex = getLRURecordIndexForAppLocked(thread);  
 return appIndex >= 0 ? mLruProcesses.get(appIndex) : null;  
}

private final int getLRURecordIndexForAppLocked(IApplicationThread thread) {  
 IBinder threadBinder = thread.asBinder();  
 // Find the application record.  
 for (int i=mLruProcesses.size()-1; i>=0; i--) {  
 ProcessRecord rec = mLruProcesses.get(i);  
 if (rec.thread != null && rec.thread.asBinder() == threadBinder) {  
 return I;  
 }  
 }  
 return -1;  
}

*/\*\*  
 \* List of running applications, sorted by recent usage.  
 \* The first entry in the list is the least recently used.  
 \*/*final ArrayList<ProcessRecord> mLruProcesses = new ArrayList<ProcessRecord>();

描述:

倒序遍历mLruProcesses，找出与thread对应的binder相同的ProcessRecord，然后返回。

mLruProcesses记录所有运行的应用进程，按最近使用情况进行排序，第一条是最近最少使用的，这也是为啥要倒序遍历的原因。

ProcessRecord 运行中进程的超详细信息

#### 2.4.2 ActivityRecord

ActivityRecord(ActivityManagerService \_service, ProcessRecord \_caller,  
 int \_launchedFromUid, String \_launchedFromPackage, Intent \_intent, String \_resolvedType,  
 ActivityInfo aInfo, Configuration \_configuration,  
 ActivityRecord \_resultTo, String \_resultWho, int \_reqCode,  
 boolean \_componentSpecified, boolean \_rootVoiceInteraction,  
 ActivityStackSupervisor supervisor,  
 ActivityContainer container, ActivityOptions options, ActivityRecord sourceRecord) {  
 service = \_service; //AMS  
 appToken = new Token(this, service);// IApplicationToken

对象，每一个activity都会持有一个appToken，用于管理window显示隐藏等相关的一些东西  
 info = aInfo; //activityInfo, 从manifest解析得到的信息，属于PM的范畴  
 launchedFromUid = \_launchedFromUid; //调用发起者uid  
 launchedFromPackage = \_launchedFromPackage; //调用发起者包名  
 userId = UserHandle.getUserId(aInfo.applicationInfo.uid); //关于applicationInfo来源请参考ApplicationInfo.java  
 intent = \_intent;  
 shortComponentName = \_intent.getComponent().flattenToShortString();  
 resolvedType = \_resolvedType;  
 componentSpecified = \_componentSpecified;  
 rootVoiceInteraction = \_rootVoiceInteraction;  
 configuration = \_configuration;  
 taskConfigOverride = Configuration.EMPTY;  
 resultTo = \_resultTo; //启动发起者，用于接收result  
 resultWho = \_resultWho;  
 requestCode = \_reqCode;  
 state = ActivityState.INITIALIZING;  
 frontOfTask = false;  
 launchFailed = false;  
 stopped = false;  
 delayedResume = false;  
 finishing = false;  
 deferRelaunchUntilPaused = false;  
 keysPaused = false;  
 inHistory = false;  
 visible = false;  
 nowVisible = false;  
 idle = false;  
 hasBeenLaunched = false;  
 mStackSupervisor = supervisor;  
 mInitialActivityContainer = container;  
 if (options != null) {  
 pendingOptions = options;

//检查activity的ActivityOptions有没有设置ANIM\_LAUNCH\_TASK\_BEHIND属性，该属性如果被设定，那么意味着这个activity一旦被launch完成就会被清除。  
 mLaunchTaskBehind = pendingOptions.getLaunchTaskBehind();

//用于定义设备旋转时的window的退出和进入的动画，替换默认的window动画，详情可参看WM.LayoutParams#rotationAnimation以及ActivityOptions.setRotationAnimationHint中相关定义。  
 mRotationAnimationHint = pendingOptions.getRotationAnimationHint();  
 // appTimeTracker用于跟踪管理activity执行时间的

//PendingIntent是一个对于普通Intent的封装，使用它可以实现Intent的延迟执行，即在满足某些条件的时候才执行该intent，通常用于发送Notification或者短消息发送结果处理等。而此处是用于时间跟踪结束后执行相关intent。因为PendingIntent会持有Application context，所以即使Application结束了，在需要发送intent的时候依然可以使用该context进行发送。AppTimeTracker.java中deliverResult方法通过调用mReceiver.send(context, 0, fillinIntent);来发送时间跟踪结果给context对应的Application。

PendingIntent usageReport = pendingOptions.getUsageTimeReport();  
 if (usageReport != null) {  
 appTimeTracker = new AppTimeTracker(usageReport);  
 }  
 }  
  
 // This starts out true, since the initial state of an activity  
 // is that we have everything, and we shouldn't never consider it  
 // lacking in state to be removed if it dies.  
 haveState = true;  
  
 if (aInfo != null) {  
 // If the class name in the intent doesn't match that of the target, this is  
 // probably an alias. We have to create a new ComponentName object to keep track  
 // of the real activity name, so that FLAG\_ACTIVITY\_CLEAR\_TOP is handled properly.

//处理activity别名问题，参考manifest中<activity-alias>标签，如果是这类activity，那么需要跟踪指定realActivity.  
 if (aInfo.targetActivity == null  
 || (aInfo.targetActivity.equals(\_intent.getComponent().getClassName())  
 && (aInfo.launchMode == ActivityInfo.LAUNCH\_MULTIPLE  
 || aInfo.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TOP))) {  
 realActivity = \_intent.getComponent();  
 } else {  
 realActivity = new ComponentName(aInfo.packageName, aInfo.targetActivity);  
 }

//taskAffinity，affinity是密切关系、类同的意思，每个Activity都有taskAffinity属性，这个属性指出了它希望进入的Task。如果一个Activity没有显式的指明该 Activity的taskAffinity，那么它的这个属性就等于Application指明的taskAffinity，如果 Application也没有指明，那么该taskAffinity的值就等于包名，参考ApplicationInfo.java。而Task也有自己的affinity属性，它的值等于它的根 Activity的taskAffinity的值。  
 taskAffinity = aInfo.taskAffinity;

//参考ActivityInfo# FLAG\_STATE\_NOT\_NEEDED  
 stateNotNeeded = (aInfo.flags&  
 ActivityInfo.FLAG\_STATE\_NOT\_NEEDED) != 0;

//配置activity的一些信息  
 appInfo = aInfo.applicationInfo;  
 nonLocalizedLabel = aInfo.nonLocalizedLabel;  
 labelRes = aInfo.labelRes;  
 if (nonLocalizedLabel == null && labelRes == 0) {  
 ApplicationInfo app = aInfo.applicationInfo;  
 nonLocalizedLabel = app.nonLocalizedLabel;  
 labelRes = app.labelRes;  
 }  
 icon = aInfo.getIconResource();  
 logo = aInfo.getLogoResource();

//配置activity的theme，如果未指定，则使用默认theme  
 theme = aInfo.getThemeResource();  
 realTheme = theme;  
 if (realTheme == 0) {  
 realTheme = aInfo.applicationInfo.targetSdkVersion  
 < Build.VERSION\_CODES.HONEYCOMB  
 ? android.R.style.Theme  
 : android.R.style.Theme\_Holo;  
 }  
 if ((aInfo.flags&ActivityInfo.FLAG\_HARDWARE\_ACCELERATED) != 0) {  
 windowFlags |= WindowManager.LayoutParams.FLAG\_HARDWARE\_ACCELERATED;  
 }  
 if ((aInfo.flags&ActivityInfo.FLAG\_MULTIPROCESS) != 0  
 && \_caller != null  
 && (aInfo.applicationInfo.uid == Process.SYSTEM\_UID  
 || aInfo.applicationInfo.uid == \_caller.info.uid)) {  
 processName = \_caller.processName;  
 } else {  
 processName = aInfo.processName;  
 }  
  
 if (intent != null && (aInfo.flags & ActivityInfo.FLAG\_EXCLUDE\_FROM\_RECENTS) != 0) {  
 intent.addFlags(Intent.FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS);  
 }  
  
 packageName = aInfo.applicationInfo.packageName;  
 launchMode = aInfo.launchMode;  
  
 AttributeCache.Entry ent = AttributeCache.instance().get(packageName,  
 realTheme, com.android.internal.R.styleable.Window, userId);

//判断是否为透明activity，满足如下任何条件则表明为透明，包括半透明

// 1. 如果指定了Window\_windowIsTranslucent为true

// 2. 如果指定了Window\_windowSwipeToDismiss为ture并且未明确指定Window\_windowIsTranslucent为false  
 final boolean translucent = ent != null && (ent.array.getBoolean(  
 com.android.internal.R.styleable.Window\_windowIsTranslucent, false)  
 || (!ent.array.hasValue(  
 com.android.internal.R.styleable.Window\_windowIsTranslucent)  
 && ent.array.getBoolean(  
 com.android.internal.R.styleable.Window\_windowSwipeToDismiss,  
 false)));

//判断是否为全屏，条件为既不是悬浮窗也不是透明。  
 fullscreen = ent != null && !ent.array.getBoolean(  
 com.android.internal.R.styleable.Window\_windowIsFloating, false)  
 && !translucent;  
 noDisplay = ent != null && ent.array.getBoolean(  
 com.android.internal.R.styleable.Window\_windowNoDisplay, false);  
 //设置activity类型

//APPLICATION\_ACTIVITY\_TYPE = 0; 普通应用内部activity

//HOME\_ACTIVITY\_TYPE = 1; 系统桌面对应activity，设置了Intent.CATEGORY\_HOME，也就是系统启动后的第一个activity

//RECENTS\_ACTIVITY\_TYPE = 2;recent  
 setActivityType(\_componentSpecified, \_launchedFromUid, \_intent, sourceRecord);  
 //android:immersive  
 immersive = (aInfo.flags & ActivityInfo.FLAG\_IMMERSIVE) != 0;  
  
 requestedVrComponent = (aInfo.requestedVrComponent == null) ?  
 null : ComponentName.unflattenFromString(aInfo.requestedVrComponent);  
 } else {  
 realActivity = null;  
 taskAffinity = null;  
 stateNotNeeded = false;  
 appInfo = null;  
 processName = null;  
 packageName = null;  
 fullscreen = true;  
 noDisplay = false;  
 mActivityType = APPLICATION\_ACTIVITY\_TYPE;  
 immersive = false;  
 requestedVrComponent = null;  
 }  
}

描述:包含了要启动的activity的所有信息，可以参考ActivityRecord.java中的相关注释去了解每一项的信息。

activity启动模式关系:

1. standard它是默认模式。在该模式下，Activity可以拥有多个实例，并且这些实例既可以位于同一个task，也可以位于不同的task。

2.singleTop该模式下，在同一个task中，如果存在该Activity的实例，并且该Activity实例位于栈顶(即，该Activity位于前端)，则调用startActivity()时，不再创建该Activity的示例；而仅仅只是调用Activity的onNewIntent()。否则的话，则新建该Activity的实例，并将其置于栈顶。

3. singleTask顾名思义，只容许有一个包含该Activity实例的task存在！

总的来说：singleTask的结论与android:taskAffinity相关。以A启动B来说

(01) 当A和B的taskAffinity相同时：第一次创建B的实例时，并不会启动新的task，而是直接将B添加到A所在的task；否则，将B所在task中位于B之上的全部Activity都删除，然后跳转到B中。

(02) 当A和B的taskAffinity不同时：第一次创建B的实例时，会启动新的task，然后将B添加到新建的task中；否则，将B所在task中位于B之上的全部Activity都删除，然后跳转到B中。

4. singleInstance顾名思义，是单一实例的意思，即任意时刻只允许存在唯一的Activity实例，而且该Activity所在的task不能容纳除该Activity之外的其他Activity实例！

它与singleTask有相同之处，也有不同之处。

相同之处：任意时刻，最多只允许存在一个实例。

不同之处：(01) singleTask受android:taskAffinity属性的影响，而singleInstance不受android:taskAffinity的影响。 (02) singleTask所在的task中能有其它的Activity，而singleInstance的task中不能有其他Activity。 (03) 当跳转到singleTask类型的Activity，并且该Activity实例已经存在时，会删除该Activity所在task中位于该Activity之上的全部Activity实例；而跳转到singleInstance类型的Activity，并且该Activity已经存在时，不需要删除其他Activity，因为它所在的task只有该Activity唯一一个Activity实例。

### 2.5 startActivityUnchecked

[ActivityStarter.java]

private int startActivityUnchecked(final ActivityRecord r, ActivityRecord sourceRecord,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 int startFlags, boolean doResume, ActivityOptions options, TaskRecord inTask) {  
  
 setInitialState(r, options, inTask, doResume, startFlags, sourceRecord, voiceSession,  
 voiceInteractor); 参考2.5.1  
//检查launch task flag：

//如果明确指定了inTask不为空，那么检查是否合适，如果合适，则加入该task中。其中一个条件就是源ActivityRecord必须为空，也就是发起方不能是另一个activity。

//如果未指定inTask，则针对以下3种情况，给mLaunchFlags 添加FLAG\_ACTIVITY\_NEW\_TASK

// 1. 源ActivityRecord为空，即发起方的task已经消失

// 2. 源ActivityRecord不为空，但是它设置了LAUNCH\_SINGLE\_INSTANCE，即它是singleInstance的，单独占一个task

// 3.目标activity设置了mLaunchSingleInstance || mLaunchSingleTask  
 computeLaunchingTaskFlags();  
//计算SourceRecord相关信息，如果存在则把他当stack赋值给mSourceStack，如果不存在则把mSourceStack置位空，如果存在但是正在finishing，即那个activity正在被finish，那么给新activity的mLaunchFlag添加FLAG\_ACTIVITY\_NEW\_TASK属性，并记录保存源task的mNewTaskInfo = mSourceRecord.info;和mNewTaskIntent = mSourceRecord.task.intent;属性  
 computeSourceStack();  
  
 mIntent.setFlags(mLaunchFlags);  
 //参考2.5.2，检查是否有可重用的activity  
 mReusedActivity = getReusableIntentActivity();  
  
 final int preferredLaunchStackId =  
 (mOptions != null) ? mOptions.getLaunchStackId() : INVALID\_STACK\_ID;  
 //有可重用activity的情况  
 if (mReusedActivity != null) {  
 // When the flags NEW\_TASK and CLEAR\_TASK are set, then the task gets reused but  
 // still needs to be a lock task mode violation since the task gets cleared out and  
 // the device would otherwise leave the locked task.  
 if (mSupervisor.isLockTaskModeViolation(mReusedActivity.task,  
 (mLaunchFlags & (FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_CLEAR\_TASK))  
 == (FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_CLEAR\_TASK))) {  
 mSupervisor.showLockTaskToast();  
 Slog.e(TAG, "startActivityUnchecked: Attempt to violate Lock Task Mode");  
 return START\_RETURN\_LOCK\_TASK\_MODE\_VIOLATION;  
 }  
 // 统一两个activity的task  
 if (mStartActivity.task == null) {  
 mStartActivity.task = mReusedActivity.task;  
 }

//虽然activity可以复用，但是intent携带的数据未必相同，所以需要把新的数据设置给复用的activity。  
 if (mReusedActivity.task.intent == null) {  
 // This task was started because of movement of the activity based on affinity...  
 // Now that we are actually launching it, we can assign the base intent.  
 mReusedActivity.task.setIntent(mStartActivity);  
 }  
  
 // This code path leads to delivering a new intent, we want to make sure we schedule it  
 // as the first operation, in case the activity will be resumed as a result of later  
 // operations.

//针对这几种类型，都需要清除位于目标activity之上的其它activity，然后直接通过onNewIntent唤醒要启动的activity  
 if ((mLaunchFlags & FLAG\_ACTIVITY\_CLEAR\_TOP) != 0  
 || mLaunchSingleInstance || mLaunchSingleTask) {  
 // In this situation we want to remove all activities from the task up to the one  
 // being started. In most cases this means we are resetting the task to its initial  
 // state.  
 //清除activity

final ActivityRecord top = mReusedActivity.task.performClearTaskForReuseLocked(  
 mStartActivity, mLaunchFlags);  
 if (top != null) {  
 if (top.frontOfTask) {  
 // Activity aliases may mean we use different intents for the top activity,  
 // so make sure the task now has the identity of the new intent.  
 top.task.setIntent(mStartActivity);  
 }

//打印EventLog,关于EventLog可以通过adb命令“adb logcat –v events”来查看，EventLog详细记录了activity的生命周期。  
 ActivityStack.logStartActivity(AM\_NEW\_INTENT, mStartActivity, top.task);  
 top.deliverNewIntentLocked(mCallingUid, mStartActivity.intent,  
 mStartActivity.launchedFromPackage);//参考2.5.3  
 }  
 }  
 //电源相关，最终会调用到PMS的powerHintInternal，具体有native实现  
 sendPowerHintForLaunchStartIfNeeded(false /\* forceSend \*/);  
 //把目标stack移到front，涉及到WMS对task的操作。

//首先一定会把目标stack移到front

//其次如果需要的话会把目标task移动到目标stack 的top，但是如果设置了mTaskOverLay, 则不会移到task到stack top，如果同时设置了FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_CLEAR\_TASK，则没有必要移动task，因为会首先清空task，后续会调用setTaskFromIntentActivity 进行相关设置。  
 mReusedActivity = setTargetStackAndMoveToFrontIfNeeded(mReusedActivity);  
 //一些特殊情况，需要调用resume  
 if ((mStartFlags & START\_FLAG\_ONLY\_IF\_NEEDED) != 0) {  
 // We don't need to start a new activity, and the client said not to do anything  
 // if that is the case, so this is it! And for paranoia, make sure we have  
 // correctly resumed the top activity.  
 resumeTargetStackIfNeeded();  
 return START\_RETURN\_INTENT\_TO\_CALLER;  
 }

//设置Task：

// 针对设置FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_CLEAR\_TASK的，需要清除整个栈，然后再把mReusedActivity加入栈，并且置位 mReuseTask，后续在判断task的时候会用到，参考2.5.4.1.1 getLaunchStack

// 针对FLAG\_ACTIVITY\_CLEAR\_TOP 或者mLaunchSingleInstance 或者mLaunchSingleTask，需要清除位于mReusedActivity之上的所有Activity

//针对已经在top的情况，需要调用onNewIntent

//以及其它情况  
 setTaskFromIntentActivity(mReusedActivity);  
 //一些特殊情况，需要调用resume  
 if (!mAddingToTask && mReuseTask == null) {  
 // We didn't do anything... but it was needed (a.k.a., client don't use that  
 // intent!) And for paranoia, make sure we have correctly resumed the top activity.  
 resumeTargetStackIfNeeded();  
 return START\_TASK\_TO\_FRONT;  
 }  
 }  
 //未找到activity的情况，返回START\_CLASS\_NOT\_FOUND（见ActivityManager.java）错误  
 if (mStartActivity.packageName == null) {

//在未找到要启动的activity的情况下，返回给需要结果的activity RESULT\_CANCELED  
 if (mStartActivity.resultTo != null && mStartActivity.resultTo.task.stack != null) {  
 mStartActivity.resultTo.task.stack.sendActivityResultLocked(  
 -1, mStartActivity.resultTo, mStartActivity.resultWho,  
 mStartActivity.requestCode, RESULT\_CANCELED, null);  
 }  
 ActivityOptions.abort(mOptions);  
 return START\_CLASS\_NOT\_FOUND;  
 }  
  
 // If the activity being launched is the same as the one currently at the top, then  
 // we need to check if it should only be launched once.

//不需要真的启动activity的情况，可能需要调用onNewIntent，返回activity未真的启动的消息START\_RETURN\_INTENT\_TO\_CALLER或者START\_DELIVERED\_TO\_TOP，具体介绍请参考AM源码。  
 final ActivityStack topStack = mSupervisor.mFocusedStack;  
 final ActivityRecord top = topStack.topRunningNonDelayedActivityLocked(mNotTop);

//当前focus的栈顶acitivity如果与要启动的activity相同，并且启动方式为singleTop或者singleTask，并且满足以下其它条件，则dontStart为true。表明不需要重新启动activity，并返回START\_DELIVERED\_TO\_TOP（见ActivityManager.java）.  
 final boolean dontStart = top != null && mStartActivity.resultTo == null  
 && top.realActivity.equals(mStartActivity.realActivity)  
 && top.userId == mStartActivity.userId  
 && top.app != null && top.app.thread != null  
 && ((mLaunchFlags & FLAG\_ACTIVITY\_SINGLE\_TOP) != 0  
 || mLaunchSingleTop || mLaunchSingleTask);  
 if (dontStart) {

//记录EventsLog，参考附录二，其中am\_new\_intent代表一个activity被赋予了新的intent事件  
 ActivityStack.logStartActivity(AM\_NEW\_INTENT, top, top.task);  
 // For paranoia, make sure we have correctly resumed the top activity.

// mLastPausedActivity 大意是说这个对象是用来标记是否在睡眠的时候需要做activity切换，请参考ActivityStack里面的注释。  
 topStack.mLastPausedActivity = null;

//如果不是延迟启动的activity，该变量都是true。  
 if (mDoResume) {

//详见activity的启动  
 mSupervisor.resumeFocusedStackTopActivityLocked();  
 }

//消除转场动画  
 ActivityOptions.abort(mOptions);  
 if ((mStartFlags & START\_FLAG\_ONLY\_IF\_NEEDED) != 0) {  
 // We don't need to start a new activity, and the client said not to do  
 // anything if that is the case, so this is it!  
 return START\_RETURN\_INTENT\_TO\_CALLER;  
 }  
 //参考2.5.3，调用onNewIntent。

top.deliverNewIntentLocked(  
 mCallingUid, mStartActivity.intent, mStartActivity.launchedFromPackage);  
  
 // Don't use mStartActivity.task to show the toast. We're not starting a new activity  
 // but reusing 'top'. Fields in mStartActivity may not be fully initialized.  
 mSupervisor.handleNonResizableTaskIfNeeded(  
 top.task, preferredLaunchStackId, topStack.mStackId);  
  
 return START\_DELIVERED\_TO\_TOP;  
 }  
   
 boolean newTask = false;

// mLaunchTaskBehind 表明activity启动完成后立即清理掉，一般不会用到，可以通过ActivityOptions.setLaunchTaskBehind进行设置  
 final TaskRecord taskToAffiliate = (mLaunchTaskBehind && mSourceRecord != null)  
 ? mSourceRecord.task : null;  
  
 // Should this be considered a new task?  
 if (mStartActivity.resultTo == null && mInTask == null && !mAddingToTask  
 && (mLaunchFlags & FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

//需要放到新的task里面  
 newTask = true;

//参考2.5.4，针对设置了FLAG\_ACTIVITY\_NEW\_TASK属性的情况设置目标task  
 setTaskFromReuseOrCreateNewTask(taskToAffiliate);  
  
 if (mSupervisor.isLockTaskModeViolation(mStartActivity.task)) {  
 Slog.e(TAG, "Attempted Lock Task Mode violation mStartActivity=" + mStartActivity);  
 return START\_RETURN\_LOCK\_TASK\_MODE\_VIOLATION;  
 }

//设置task exit的时候应该返回到哪个task  
 if (!mMovedOtherTask) {  
 // If stack id is specified in activity options, usually it means that activity is  
 // launched not from currently focused stack (e.g. from SysUI or from shell) - in  
 // that case we check the target stack.  
 updateTaskReturnToType(mStartActivity.task, mLaunchFlags,  
 preferredLaunchStackId != INVALID\_STACK\_ID ? mTargetStack : topStack);  
 }  
 } else if (mSourceRecord != null) {  
 if (mSupervisor.isLockTaskModeViolation(mSourceRecord.task)) {  
 Slog.e(TAG, "Attempted Lock Task Mode violation mStartActivity=" + mStartActivity);  
 return START\_RETURN\_LOCK\_TASK\_MODE\_VIOLATION;  
 }  
 //对于发起方不为空的情况，设置task  
 final int result = setTaskFromSourceRecord();  
 if (result != START\_SUCCESS) {  
 return result;  
 }  
 } else if (mInTask != null) {  
 // The caller is asking that the new activity be started in an explicit  
 // task it has provided to us.  
 if (mSupervisor.isLockTaskModeViolation(mInTask)) {  
 Slog.e(TAG, "Attempted Lock Task Mode violation mStartActivity=" + mStartActivity);  
 return START\_RETURN\_LOCK\_TASK\_MODE\_VIOLATION;  
 }  
  
 //对于指定inTask不为空的情况，设置task  
 final int result = setTaskFromInTask();  
 if (result != START\_SUCCESS) {  
 return result;  
 }  
 } else {  
 // This not being started from an existing activity, and not part of a new task...  
 // just put it in the top task, though these days this case should never happen.  
 setTaskToCurrentTopOrCreateNewTask();  
 }  
 //到这里，目标activity所属的task以及stack都已经确定并拿到最上层

//检查目标activity是否需要UriPermission，如果需要则给添加UriPermission。参考2.5.5，UriPermission主要是用来不同应用间的私有数据传递共享使用，比如应用A把自己的ContentProvider对应的URI传递给应用B的某个activity，并分配相关的访问权限。  
 mService.grantUriPermissionFromIntentLocked(mCallingUid, mStartActivity.packageName,  
 mIntent, mStartActivity.getUriPermissionsLocked(), mStartActivity.userId);  
 //当调用方是RecentsActivity的时候，设置目标task退出的时候应该退到哪个task，这里是直接退到HOME\_ACTIVITY\_TYPE，参考TaskRecord对于setTaskToReturnTo的实现。这也符合实际情况，从最近列表启动activity之后，退出该activity会直接回答桌面。  
 if (mSourceRecord != null && mSourceRecord.isRecentsActivity()) {  
 mStartActivity.task.setTaskToReturnTo(RECENTS\_ACTIVITY\_TYPE);  
 }  
 if (newTask) {

//记录新创建task的event log，也就是说在events log里面可以看到task的创建记录。  
 EventLog.writeEvent(  
 EventLogTags.AM\_CREATE\_TASK, mStartActivity.userId, mStartActivity.task.taskId);  
 }

//记录Activity创建的events Log，对应EventsLog am\_create\_activity  
 ActivityStack.logStartActivity(  
 EventLogTags.AM\_CREATE\_ACTIVITY, mStartActivity, mStartActivity.task);  
 mTargetStack.mLastPausedActivity = null;  
 //这里针对目标task是新建的情况会发送PowerHint，目的是省电。  
 sendPowerHintForLaunchStartIfNeeded(false /\* forceSend \*/);  
 //参考2.6，配置activity resume前的一些task、动画等属性，创建AppWindowToken，创建启动窗口View  
 mTargetStack.startActivityLocked(mStartActivity, newTask, mKeepCurTransition, mOptions);

//接下来就是着手显示activity  
 if (mDoResume) {   
 if (!mLaunchTaskBehind) {

//设置获取焦点，主要功能点包括:

//1. 启动一个AppTimeTracker

//2. 把对应的stack移动到顶端并通知WM也把对应的stack移动到front

//3. 调用WMS的setFocusedApp设置焦点  
 mService.setFocusedActivityLocked(mStartActivity, "startedActivity");  
 }

final ActivityRecord topTaskActivity = mStartActivity.task.topRunningActivityLocked();  
 if (!mTargetStack.isFocusable()  
 || (topTaskActivity != null && topTaskActivity.mTaskOverlay  
 && mStartActivity != topTaskActivity)) {  
 // If the activity is not focusable, we can't resume it, but still would like to  
 // make sure it becomes visible as it starts (this will also trigger entry  
 // animation). An example of this are PIP activities.  
 // Also, we don't want to resume activities in a task that currently has an overlay  
 // as the starting activity just needs to be in the visible paused state until the  
 // over is removed.  
 mTargetStack.ensureActivitiesVisibleLocked(null, 0, !PRESERVE\_WINDOWS);  
 // Go ahead and tell window manager to execute app transition for this activity  
 // since the app transition will not be triggered through the resume channel.  
 mWindowManager.executeAppTransition();  
 } else {

//正式启动Activity,进入所谓的生命周期 参考2.7  
 mSupervisor.resumeFocusedStackTopActivityLocked(mTargetStack, mStartActivity,  
 mOptions);  
 }  
 } else {

//不需要resume的Activity，需要加入到recent list中  
 mTargetStack.addRecentActivityLocked(mStartActivity);  
 }

//给多用户使用，多用户时其它用户可能需要知道当前用户处于focus状态的activity。  
 mSupervisor.updateUserStackLocked(mStartActivity.userId, mTargetStack);  
  
 mSupervisor.handleNonResizableTaskIfNeeded(  
 mStartActivity.task, preferredLaunchStackId, mTargetStack.mStackId);  
  
 return START\_SUCCESS;  
}

参数：

r 目标ActivityRecord

sourceRecord 源ActivityRecord

voiceSession和voiceInteractor都是和语音启动方式相关的

startFlags 同前

doResume true

options和inTask 同前

描述:

1. 解析singleTask,singleTop以及FLAG\_ACTIVITY\_NEW\_DOCUMENT等可能会重用已有activity的属性，然后检查是否存在可重用的activity，如果存在，则走重用流程。
2. task，ActivityStack，ActivityRecord关系：
   1. stack可以包含多个task，stack按顺序维护一个task列表
   2. 每个task可以包含多个ActivityRecord，但一个task只能属于一个stack
   3. 每个ActivityRecord都可以属于不同的进程，比如微信调起camera，两个activity属于不同的进程，但是却属于同一个task同一个stack，所以推出camera的时候能返回到微信界面

#### 2.5.1 setInitialState

[ActivityStarter.java]

private void setInitialState(ActivityRecord r, ActivityOptions options, TaskRecord inTask,  
 boolean doResume, int startFlags, ActivityRecord sourceRecord,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor) {  
 reset(); //清空当前类变量  
  
 mStartActivity = r;//目标ActivityRecord  
 mIntent = r.intent;  
 mOptions = options;  
 mCallingUid = r.launchedFromUid;  
 mSourceRecord = sourceRecord;  
 mVoiceSession = voiceSession;  
 mVoiceInteractor = voiceInteractor;  
//activity边界，这个涉及到AM中定义的StackId，包含了stack中的一些属性，用于指定activity的一些属性，比如画中画，常驻最上层，输入法那样的半框，dock，多窗口等窗口类型  
 mLaunchBounds = getOverrideBounds(r, options, inTask);  
//从manifest中解析到的activity启动模式  
 mLaunchSingleTop = r.launchMode == LAUNCH\_SINGLE\_TOP;  
 mLaunchSingleInstance = r.launchMode == LAUNCH\_SINGLE\_INSTANCE;  
 mLaunchSingleTask = r.launchMode == LAUNCH\_SINGLE\_TASK;

//参考2.5.1.1  
 mLaunchFlags = adjustLaunchFlagsToDocumentMode(  
 r, mLaunchSingleInstance, mLaunchSingleTask, mIntent.getFlags());  
 mLaunchTaskBehind = r.mLaunchTaskBehind  
 && !mLaunchSingleTask && !mLaunchSingleInstance  
 && (mLaunchFlags & FLAG\_ACTIVITY\_NEW\_DOCUMENT) != 0;  
 //如果在新的task中启动activity，而且发起启动的是通过startActivityForResult启动的，那么通知对方，启动activity被取消，因为跨task是无法返回activity结果的。在这里会把r.resultTo设置为null，注意mStartActivity 和 r指向同一个对象。也就是说startActivityForResult与FLAG\_ACTIVITY\_NEW\_TASK是冲突的

//参考2.5.1.2  
 sendNewTaskResultRequestIfNeeded();  
 //此处需要注意r.resultTo == null，可以看到只要resultTo不为空，则mLaunchFlags就不会添加FLAG\_ACTIVITY\_NEW\_TASK属性，也就是说通过startActivityForResult启动的activity不会创建新的task(除非明确通过intent添加FLAG\_ACTIVITY\_NEW\_TASK属性)，因为不同的task无法通知调用结果，除非明确配置了FLAG\_ACTIVITY\_NEW\_TASK，否则即使配置了launchMode中的singleTask或者singleInstance也没用，配置taskAffinity也没用，都不会创建新的task。  
 if ((mLaunchFlags & FLAG\_ACTIVITY\_NEW\_DOCUMENT) != 0 && r.resultTo == null) {  
 mLaunchFlags |= FLAG\_ACTIVITY\_NEW\_TASK;  
 }  
  
 // If we are actually going to launch in to a new task, there are some cases where  
 // we further want to do multiple task.  
 if ((mLaunchFlags & FLAG\_ACTIVITY\_NEW\_TASK) != 0) {  
 if (mLaunchTaskBehind  
 || r.info.documentLaunchMode == DOCUMENT\_LAUNCH\_ALWAYS) {  
 mLaunchFlags |= FLAG\_ACTIVITY\_MULTIPLE\_TASK;  
 }  
 }  
 // mUserLeaving为true表示在调用onPause之前会调用onUserLeaveHint，除非明确指定FLAG\_ACTIVITY\_NO\_USER\_ACTION属性。如果指定了会把mUserLeaving置位false。

//onUserLeaving属于activity生命周期的一部分，调用时间在onPause之前，调用条件是用户主动离开该activity，比如按home键或者通过recentList切换到其它应用或者点击按钮跳转到其它activity，但是按back键或者通过来电界面等非用户操作离开则不会调用。  
 // We'll invoke onUserLeaving before onPause only if the launching  
 // activity did not explicitly state that this is an automated launch.  
 mSupervisor.mUserLeaving = (mLaunchFlags & FLAG\_ACTIVITY\_NO\_USER\_ACTION) == 0;  
 if (DEBUG\_USER\_LEAVING) Slog.v(TAG\_USER\_LEAVING,  
 "startActivity() => mUserLeaving=" + mSupervisor.mUserLeaving);  
  
 // If the caller has asked not to resume at this point, we make note  
 // of this in the record so that we can skip it when trying to find  
 // the top running activity.  
 mDoResume = doResume;  
 if (!doResume || !mSupervisor.okToShowLocked(r)) {  
 r.delayedResume = true;  
 mDoResume = false;  
 }  
// getTaskOverlay()是通过在ActivityOptions里面设置的。ActivityRecord的参数mTaskOverlay只有在这个地方可能置为true，也就是说这有在设置了options的launchTaskId和TaskOverly的情况下才会生效。

if (mOptions != null && mOptions.getLaunchTaskId() != -1 && mOptions.getTaskOverlay()) {

//如果launchTask存在，并且在目标task的Top activity不为空且不可见的情况下，这个activity不需要resume。

// mTaskOverlay表示这个activity可以被其他activity覆盖，所以在启动该activity的时候，如果该activity已经在task里面，但是不在task最前端的时候，不需要把activity置为resume状态，但是它可能在visible状态（也就是pause状态），也可能不可见。  
 r.mTaskOverlay = true;  
 final TaskRecord task = mSupervisor.anyTaskForIdLocked(mOptions.getLaunchTaskId());  
 final ActivityRecord top = task != null ? task.getTopActivity() : null;  
 if (top != null && !top.visible) {  
  
 // The caller specifies that we'd like to be avoided to be moved to the front, so be  
 // it!  
 mDoResume = false;  
 mAvoidMoveToFront = true;  
 }  
 }  
  
 mNotTop = (mLaunchFlags & FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP) != 0 ? r : null;  
  
 mInTask = inTask;  
 // In some flows in to this function, we retrieve the task record and hold on to it  
 // without a lock before calling back in to here... so the task at this point may  
 // not actually be in recents. Check for that, and if it isn't in recents just  
 // consider it invalid.  
 if (inTask != null && !inTask.inRecents) {  
 Slog.w(TAG, "Starting activity in task not in recents: " + inTask);  
 mInTask = null;  
 }  
  
 mStartFlags = startFlags;  
 // If the onlyIfNeeded flag is set, then we can do this if the activity being launched  
 // is the same as the one making the call... or, as a special case, if we do not know  
 // the caller then we count the current top activity as the caller.  
 if ((startFlags & START\_FLAG\_ONLY\_IF\_NEEDED) != 0) {  
 ActivityRecord checkedCaller = sourceRecord;  
 if (checkedCaller == null) {  
 checkedCaller = mSupervisor.mFocusedStack.topRunningNonDelayedActivityLocked(  
 mNotTop);  
 }  
 if (!checkedCaller.realActivity.equals(r.realActivity)) {  
 // Caller is not the same as launcher, so always needed.  
 mStartFlags &= ~START\_FLAG\_ONLY\_IF\_NEEDED;  
 }  
 }  
  
 mNoAnimation = (mLaunchFlags & FLAG\_ACTIVITY\_NO\_ANIMATION) != 0;  
}

参数同2.5完全一样

描述：该方法的主要功能就是通过从之前解析生成的ActivityRecord初始化ActivityStarter的类成员。

##### 2.5.1.1 adjustLaunchFlagsToDocumentMode

[ActivityStarter]

private int adjustLaunchFlagsToDocumentMode(ActivityRecord r, boolean launchSingleInstance,  
 boolean launchSingleTask, int launchFlags) {

//表示同时给inten设置了FLAG\_ACTIVITY\_NEW\_DOCUMENT flag和在manifest配置了SingleInstance或者SingleTask，这种情况，忽略FLAG\_ACTIVITY\_NEW\_DOCUMENT  
 if ((launchFlags & Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT) != 0 &&  
 (launchSingleInstance || launchSingleTask)) {  
 // We have a conflict between the Intent and the Activity manifest, manifest wins.  
 Slog.i(TAG, "Ignoring FLAG\_ACTIVITY\_NEW\_DOCUMENT, launchMode is " +  
 "\"singleInstance\" or \"singleTask\"");

//移除这两个flag  
 launchFlags &=  
 ~(Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT | FLAG\_ACTIVITY\_MULTIPLE\_TASK);  
 } else {

// r.info.documentLaunchMode 代表从manifest中配置的属性  
 switch (r.info.documentLaunchMode) {  
 case ActivityInfo.DOCUMENT\_LAUNCH\_NONE:  
 break;  
 case ActivityInfo.DOCUMENT\_LAUNCH\_INTO\_EXISTING:  
 launchFlags |= Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT;  
 break;  
 case ActivityInfo.DOCUMENT\_LAUNCH\_ALWAYS:  
 launchFlags |= Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT;  
 break;  
 case ActivityInfo.DOCUMENT\_LAUNCH\_NEVER:  
 launchFlags &= ~FLAG\_ACTIVITY\_MULTIPLE\_TASK;  
 break;  
 }  
 }  
 return launchFlags;  
}

描述:

注：以下介绍请注意区分manifest配置和Intent setFlag配置的不同。

该方法的主要功能就是检查并处理activity启动时intent携带的flag中的 Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT。该属性主要功能是对于每一个activity task都会在最近使用列表中展示一项，可以使用该功能使你的应用中的精彩页面都呈现在最近使用列表中。

在manifest中配置android:documentLaunchMode可以达到呈现document的效果，该属性有四种模式，分别为：

**intoExisting 启动该activity的时候会重新创建一个task，并把activity入栈，每一个task会维护一个stack，stack的属性是先进后出。如果已经有包含该activity的task的时候，则直接把该task变为top task，即可见task，但是栈里面的activity顺序并不变，即要启动的activity并不一定会在最上层。这种启动模式相当于给intent设置了flag** Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT。该属性相当于singleTask，新创建的task里面允许入栈新的activity。

**always 启动该activity的时候总是会重新创建一个stack，相当于同时给intent设置了flag** Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT 和Intent. FLAG\_ACTIVITY\_MULTIPLE\_TASK。

**none 默认情况，跟没设置相同。**

**never 设置never其结果和none一样，而且会直接移除**FLAG\_ACTIVITY\_MULTIPLE\_TASK 属性

launchFlags &= ~FLAG\_ACTIVITY\_MULTIPLE\_TASK;

当manifest中配置了android:launchMode="singleTask"/"singleInstance"，那么会使得Intent.FLAG\_ACTIVITY\_NEW\_DOCUMENT flag无效。

##### 2.5.1.2 sendNewTaskResultRequestIfNeeded

private void sendNewTaskResultRequestIfNeeded() {  
 if (mStartActivity.resultTo != null && (mLaunchFlags & FLAG\_ACTIVITY\_NEW\_TASK) != 0  
 && mStartActivity.resultTo.task.stack != null) {  
 mStartActivity.resultTo.task.stack.sendActivityResultLocked(-1, mStartActivity.resultTo,  
 mStartActivity.resultWho, mStartActivity.requestCode, RESULT\_CANCELED, null);  
 mStartActivity.resultTo = null;  
 }  
}

描述:从该方法可以看到只要同时设置了resultTo和FLAG\_ACTIVITY\_NEW\_TASK，就会把resultTo置为空，也就是说FLAG\_ACTIVITY\_NEW\_TASK的优先级更高。一旦置位FLAG\_ACTIVITY\_NEW\_TASK，则表示无法再通知result，所以会先通知RESULT\_CANCELED。

这里需要注意，即使在配置了FLAG\_ACTIVITY\_NEW\_TASK而没有配置新的taskAffinity，使得两个activity实际上在同一个栈里面，也同样会直接返回RESULT\_CANCELED。

#### 2.5.2 getReusableIntentActivity

private ActivityRecord getReusableIntentActivity() {  
 // We may want to try to place the new activity in to an existing task. We always  
 // do this if the target activity is singleTask or singleInstance; we will also do  
 // this if NEW\_TASK has been requested, and there is not an additional qualifier telling  
 // us to still place it in a new task: multi task, always doc mode, or being asked to  
 // launch this as a new task behind the current one.  
 boolean putIntoExistingTask = ((mLaunchFlags & FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&  
 (mLaunchFlags & FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)  
 || mLaunchSingleInstance || mLaunchSingleTask;  
 // If bring to front is requested, and no result is requested and we have not been given  
 // an explicit task to launch in to, and we can find a task that was started with this  
 // same component, then instead of launching bring that one to the front.  
 //对于未指定目标task并且源ActivityRecord为空的情况，则置为false，因为肯定不存在，就相当于一定会重新创建task

putIntoExistingTask &= mInTask == null && mStartActivity.resultTo == null;  
 ActivityRecord intentActivity = null;  
 if (mOptions != null && mOptions.getLaunchTaskId() != -1) {  
 final TaskRecord task = mSupervisor.anyTaskForIdLocked(mOptions.getLaunchTaskId());  
 intentActivity = task != null ? task.getTopActivity() : null;  
 } else if (putIntoExistingTask) {  
 if (mLaunchSingleInstance) {  
 // There can be one and only one instance of single instance activity in the  
 // history, and it is always in its own unique task, so we do a special search.  
 intentActivity = mSupervisor.findActivityLocked(mIntent, mStartActivity.info, false);  
 } else if ((mLaunchFlags & FLAG\_ACTIVITY\_LAUNCH\_ADJACENT) != 0) {  
 // For the launch adjacent case we only want to put the activity in an existing  
 // task if the activity already exists in the history.  
 intentActivity = mSupervisor.findActivityLocked(mIntent, mStartActivity.info,  
 !mLaunchSingleTask);  
 } else {  
 // Otherwise find the best task to put the activity in.  
 intentActivity = mSupervisor.findTaskLocked(mStartActivity);  
 }  
 }  
 return intentActivity;  
}

描述:该方法的作用是寻找要启动的activity是否有可重用的，查询方法是判断要启动的activity是否已经存储在某个task中，如果是，则返回该task中的相应ActivityRecord，如果不存在，则返回null；

判断方法: 如果满足下面几个条件之一，则可能已经存在某个task中，需要去查询

1. 通过manifest指定为mLaunchSingleInstance
2. 通过manifest指定为mLaunchSingleTask
3. 通过Intent flag 指定为FLAG\_ACTIVITY\_NEW\_TASK但是未指定FLAG\_ACTIVITY\_MULTIPLE\_TASK，因为如果指定了FLAG\_ACTIVITY\_MULTIPLE\_TASK那么无论如何都会重新创建task，肯定不可能重用已经存在的activity。

#### 2.5.3 deliverNewIntentLocked

[ActivityRecord.java]

final void deliverNewIntentLocked(int callingUid, Intent intent, String referrer) {  
 // The activity now gets access to the data associated with this Intent.  
 service.grantUriPermissionFromIntentLocked(callingUid, packageName,  
 intent, getUriPermissionsLocked(), userId);  
 final ReferrerIntent rintent = new ReferrerIntent(intent, referrer);  
 boolean unsent = true;  
 final ActivityStack stack = task.stack;  
 final boolean isTopActivityInStack =  
 stack != null && stack.topRunningActivityLocked() == this;  
 final boolean isTopActivityWhileSleeping =  
 service.isSleepingLocked() && isTopActivityInStack;  
  
 // We want to immediately deliver the intent to the activity if:  
 // - It is currently resumed or paused. i.e. it is currently visible to the user and we want  
 // the user to see the visual effects caused by the intent delivery now.  
 // - The device is sleeping and it is the top activity behind the lock screen (b/6700897).  
 if ((state == ActivityState.RESUMED || state == ActivityState.PAUSED  
 || isTopActivityWhileSleeping) && app != null && app.thread != null) {  
 try {  
 ArrayList<ReferrerIntent> ar = new ArrayList<>(1);  
 ar.add(rintent);  
 app.thread.scheduleNewIntent(  
 ar, appToken, state == ActivityState.PAUSED /\* andPause \*/);  
 unsent = false;  
 } catch (RemoteException e) {  
 Slog.w(TAG, "Exception thrown sending new intent to " + this, e);  
 } catch (NullPointerException e) {  
 Slog.w(TAG, "Exception thrown sending new intent to " + this, e);  
 }  
 }  
 if (unsent) {  
 addNewIntentLocked(rintent);  
 }  
}

描述：该方法的主要功能是通过app.thread.scheduleNewIntent(…)调用执行目标activity的onNewIntent回调方法。其中app为ProcessRecord类型，每一个ActivityRecord都会持有一个ProcessRecord对象，代表该activity所属的进程，app.thread为IApplicationThread类型，它会通过Binder的方式最终调用到ActivityThread中的 handleNewIntent，最终通过Instrumentation的callActivityOnNewIntent回调到用户重写的onNewIntent。具体Binder调用流程会在activity启动后续流程中详细介绍，二者流程一样。至于ProcessRecord中的IApplicationThread对象是哪里来的，也会在后续介绍。

#### 2.5.4 setTaskFromReuseOrCreateNewTask 重用或者重建task的情况

[ActivityStarter.java]

从前面分析可知只有设置了FLAG\_ACTIVITY\_NEW\_TASK属性才会进入该方法

private void setTaskFromReuseOrCreateNewTask(TaskRecord taskToAffiliate) {  
 //计算获得或者通过创建获得目标activity所在task对应的stack，以及获取焦点，参考2.5.4.1

mTargetStack = computeStackFocus(mStartActivity, true, mLaunchBounds, mLaunchFlags,  
 mOptions);  
 // mReuseTask为true表示目标activity有可重用的task，这个变量在前面检查是否有可重用的activity的时候赋值，主要有两个地方，一是通过指定inTask的方式，二是设置了FLAG\_ACTIVITY\_NEW\_TASK | FLAG\_ACTIVITY\_CLEAR\_TASK属性。除此之外，该值为空  
 if (mReuseTask == null) {

//在设置了FLAG\_ACTIVITY\_NEW\_TASK但是没有可以复用的task的情况下会走到这里去创建新的TaskRecord，参考2.5.4.2  
 final TaskRecord task = mTargetStack.createTaskRecord(  
 mSupervisor.getNextTaskIdForUserLocked(mStartActivity.userId),  
 mNewTaskInfo != null ? mNewTaskInfo : mStartActivity.info,  
 mNewTaskIntent != null ? mNewTaskIntent : mIntent,  
 mVoiceSession, mVoiceInteractor, !mLaunchTaskBehind /\* toTop \*/);  
 mStartActivity.setTask(task, taskToAffiliate);  
 if (mLaunchBounds != null) {  
 final int stackId = mTargetStack.mStackId;  
 if (StackId.resizeStackWithLaunchBounds(stackId)) {  
 mService.resizeStack(  
 stackId, mLaunchBounds, true, !PRESERVE\_WINDOWS, ANIMATE, -1);  
 } else {  
 mStartActivity.task.updateOverrideConfiguration(mLaunchBounds);  
 }  
 }  
 if (DEBUG\_TASKS) Slog.v(TAG\_TASKS,  
 "Starting new activity " +  
 mStartActivity + " in new task " + mStartActivity.task);  
 } else {  
 mStartActivity.setTask(mReuseTask, taskToAffiliate);  
 }  
}

##### 2.5.4.1 computeStackFocus Activity栈的查找和创建

private ActivityStack computeStackFocus(ActivityRecord r, boolean newTask, Rect bounds,  
 int launchFlags, ActivityOptions aOptions) {  
 final TaskRecord task = r.task; //该activity所依附的task

//判断activity类型是不是APPLICATION\_ACTIVITY\_TYPE，如果不是再判断他所依赖的task是不是APPLICATION\_ACTIVITY\_TYPE类型（注意activity和task的类型是一样的，参考TaskRecord#taskType的初始化），如果都不是，则直接返回mHomeStack。这表明要启动的activity是系统launcher，比如按home键。  
 if (!(r.isApplicationActivity() || (task != null && task.isApplicationTask()))) {  
 return mSupervisor.mHomeStack;  
 }  
 //按顺序根据设置的mReuseTask或者aOptions.getLaunchStackId()获取或者创建stack，如果没有指定这两项(此处忽略掉多窗口相关逻辑，如果有指定多窗口属性FLAG\_ACTIVITY\_LAUNCH\_ADJACENT，则还有其他可能性)，则结果为null，参考2.5.4.1.1  
 ActivityStack stack = getLaunchStack(r, launchFlags, task, aOptions);  
 if (stack != null) {  
 return stack;  
 }  
 //以下类同，都是查找目标stack是否存在，如果存在则返回  
 if (task != null && task.stack != null) {

//目标task携带了stack且不为空，则直接使用  
 stack = task.stack;  
 if (stack.isOnHomeDisplay()) {  
 if (mSupervisor.mFocusedStack != stack) {  
 if (DEBUG\_FOCUS || DEBUG\_STACK) Slog.d(TAG\_FOCUS,  
 "computeStackFocus: Setting " + "focused stack to r=" + r  
 + " task=" + task);  
 } else {  
 if (DEBUG\_FOCUS || DEBUG\_STACK) Slog.d(TAG\_FOCUS,  
 "computeStackFocus: Focused stack already="  
 + mSupervisor.mFocusedStack);  
 }  
 }  
 return stack;  
 }  
  
 final ActivityStackSupervisor.ActivityContainer container = r.mInitialActivityContainer;  
 if (container != null) {  
 // The first time put it on the desired stack, after this put on task stack.  
 r.mInitialActivityContainer = null;  
 return container.mStack;  
 }  
  
 // The fullscreen stack can contain any task regardless of if the task is resizeable  
 // or not. So, we let the task go in the fullscreen task if it is the focus stack.  
 // If the freeform or docked stack has focus, and the activity to be launched is resizeable,  
 // we can also put it in the focused stack.  
 final int focusedStackId = mSupervisor.mFocusedStack.mStackId;  
 final boolean canUseFocusedStack = focusedStackId == FULLSCREEN\_WORKSPACE\_STACK\_ID  
 || (focusedStackId == DOCKED\_STACK\_ID && r.canGoInDockedStack())  
 || (focusedStackId == FREEFORM\_WORKSPACE\_STACK\_ID && r.isResizeableOrForced());  
 if (canUseFocusedStack && (!newTask  
 || mSupervisor.mFocusedStack.mActivityContainer.isEligibleForNewTasks())) {  
 if (DEBUG\_FOCUS || DEBUG\_STACK) Slog.d(TAG\_FOCUS,  
 "computeStackFocus: Have a focused stack=" + mSupervisor.mFocusedStack);  
 return mSupervisor.mFocusedStack;  
 }  
  
 // We first try to put the task in the first dynamic stack.  
 final ArrayList<ActivityStack> homeDisplayStacks = mSupervisor.mHomeStack.mStacks;  
 for (int stackNdx = homeDisplayStacks.size() - 1; stackNdx >= 0; --stackNdx) {  
 stack = homeDisplayStacks.get(stackNdx);  
 if (!ActivityManager.StackId.isStaticStack(stack.mStackId)) {  
 if (DEBUG\_FOCUS || DEBUG\_STACK) Slog.d(TAG\_FOCUS,  
 "computeStackFocus: Setting focused stack=" + stack);  
 return stack;  
 }  
 }  
  
 // If there is no suitable dynamic stack then we figure out which static stack to use.  
 final int stackId = task != null ? task.getLaunchStackId() :  
 bounds != null ? FREEFORM\_WORKSPACE\_STACK\_ID :  
 FULLSCREEN\_WORKSPACE\_STACK\_ID;

//根据设置的默认stackId去创建新的stack并返回，参考2.5.4.1.2，后两个参数表明如果无法获取则重新创建，完成之后置为top stack  
 stack = mSupervisor.getStack(stackId, CREATE\_IF\_NEEDED, ON\_TOP);  
 if (DEBUG\_FOCUS || DEBUG\_STACK) Slog.d(TAG\_FOCUS, "computeStackFocus: New stack r="  
 + r + " stackId=" + stack.mStackId);  
 return stack;  
}

描述：

StackId: 定义在AM中，如下：

/\*\* Invalid stack ID. \*/

public static final int INVALID\_STACK\_ID = -1;

/\*\* First static stack ID. 除了动态StackID之外，其它都属于静态StackID\*/

public static final int FIRST\_STATIC\_STACK\_ID = 0;

/\*\* Home activity stack ID. 桌面activity\*/

public static final int HOME\_STACK\_ID = FIRST\_STATIC\_STACK\_ID;

/\*\* ID of stack where fullscreen activities are normally launched into. 全屏activity，普通应用都是这样的\*/

public static final int FULLSCREEN\_WORKSPACE\_STACK\_ID = 1;

/\*\* ID of stack where freeform/resized activities are normally launched into. 可调整大小的窗口\*/

public static final int FREEFORM\_WORKSPACE\_STACK\_ID = FULLSCREEN\_WORKSPACE\_STACK\_ID + 1;

/\*\* ID of stack that occupies a dedicated region of the screen. \*/

public static final int DOCKED\_STACK\_ID = FREEFORM\_WORKSPACE\_STACK\_ID + 1;

/\*\* ID of stack that always on top (always visible) when it exist. \*/

public static final int PINNED\_STACK\_ID = DOCKED\_STACK\_ID + 1;

/\*\* Last static stack stack ID. \*/

public static final int LAST\_STATIC\_STACK\_ID = PINNED\_STACK\_ID;

/\*\* Last static stack stack ID. \*/

public static final int LAST\_STATIC\_STACK\_ID = PINNED\_STACK\_ID;

/\*\* Start of ID range used by stacks that are created dynamically. \*/

public static final int FIRST\_DYNAMIC\_STACK\_ID = LAST\_STATIC\_STACK\_ID + 1;

也就是说系统总共就只支持这几种stack，一般的普通全屏activity都会放到同一个stack中，即FULLSCREEN\_WORKSPACE\_STACK\_ID

###### 2.5.4.1.1 getLaunchStack

private ActivityStack getLaunchStack(ActivityRecord r, int launchFlags, TaskRecord task,  
 ActivityOptions aOptions) {  
  
 // We are reusing a task, keep the stack!

//如果是重用已有Task，则直接返回该Task对应的stack。mReuseTask主要通过明确指定inTask或者通过前面查到的mReusedActivity（会涉及到设置了singleTask，singleInstance 或者 clearTop等）赋值，所以此处一般为空。  
 if (mReuseTask != null) {  
 return mReuseTask.stack;  
 }  
  
 //检查aOptions里面是否携带了StackId,如果有，则检查是否合法（关于StackId的合法性和种类的定义在AM中，参考2.5.4.1描述），合法则通过该id去查找，如果找不到则创建一个stack，关于stack的创建参考2.5.4.1.2

final int launchStackId =  
 (aOptions != null) ? aOptions.getLaunchStackId() : INVALID\_STACK\_ID;  
  
 if (isValidLaunchStackId(launchStackId, r)) {  
 return mSupervisor.getStack(launchStackId, CREATE\_IF\_NEEDED, ON\_TOP);  
 } else if (launchStackId == DOCKED\_STACK\_ID) {  
 // The preferred launch stack is the docked stack, but it isn't a valid launch stack  
 // for this activity, so we put the activity in the fullscreen stack.  
 return mSupervisor.getStack(FULLSCREEN\_WORKSPACE\_STACK\_ID, CREATE\_IF\_NEEDED, ON\_TOP);  
 }  
 //多窗口相关，如果没有设置该属性则直接返回null  
 if ((launchFlags & FLAG\_ACTIVITY\_LAUNCH\_ADJACENT) == 0) {  
 return null;  
 }  
 // Otherwise handle adjacent launch.  
 … //省略关于多窗口这一块  
}

###### 2.5.4.1.2 getStack 获取或者创建Stack

[ASS]

ActivityStack getStack(int stackId, boolean createStaticStackIfNeeded, boolean createOnTop) {

//ActivityContainer是ASS的内部类，ASS会维护一个ActivityContainer类型的数组mActivityContainers，该内部类会维护一个ActivityStack以及他的StackId，每当创建stack的时候都会把stack与一个ActivityContainer变量对应起来，并加入mActivityContainers数组中，这样就可以通过mActivityContainers来查询获取具体stakId对应的ActivityStack。普通全屏应用都会放到同一个stack中，id为1。参考AM对于StackId的定义。  
 ActivityContainer activityContainer = mActivityContainers.get(stackId);  
 if (activityContainer != null) {  
 return activityContainer.mStack;  
 }

//如果没有找到该stack，则判断是否需要创建，不需要则返回，关于静态StackID参考2.5.4.1描述  
 if (!createStaticStackIfNeeded || !StackId.isStaticStack(stackId)) {  
 return null;  
 }

//根据stackID创建新的Stack，使用默认Display  
 return createStackOnDisplay(stackId, Display.DEFAULT\_DISPLAY, createOnTop);  
}

//创建Stack

ActivityStack createStackOnDisplay(int stackId, int displayId, boolean onTop) {

// ActivityDisplay是ASS的内部类，ASS会维护一个ActivityDisplay类型的数组mActivityDisplays，ActivityDisplay类会维护一个Display以及他的mDisplayId，并且维护一个依附于该Display的ActivityStack数组mStacks.这里需要注意，每一个ActivityStack都会依附于一个Display对象，代表这个栈里面的所有activity都会显示到这个Display上面。 关于mActivityDisplays的创建和初始化参考2.5.4.1.2.1 ，这里先从mActivityDisplays中获取到ID为Display.DEFAULT\_DISPLAY的ActivityDisplay对象，然后再在这个Display上面创建一个ActivityContainer  
 ActivityDisplay activityDisplay = mActivityDisplays.get(displayId);  
 if (activityDisplay == null) {  
 return null;  
 }  
 //创建ActivityContainer，参考2.5.4.1.2.2  
 ActivityContainer activityContainer = new ActivityContainer(stackId);  
 mActivityContainers.put(stackId, activityContainer);

//参考2.5.4.1.2.3  
 activityContainer.attachToDisplayLocked(activityDisplay, onTop);  
 return activityContainer.mStack;  
}

2.5.4.1.2.1

system\_server进程启动之后，当完成WMS的初始化之后会调用

mActivityManagerService.setWindowManager(wm);

把wm的引用传递给AMS。

【AMS】

public void setWindowManager(WindowManagerService wm) {

//保存wm对象引用  
 mWindowManager = wm;

//传递wm对象引用给ASS，见如下分析  
 mStackSupervisor.setWindowManager(wm);

//把wm对象引用传递给AS，AS也会保存wm对象  
 mActivityStarter.setWindowManager(wm);  
}

【ASS】

void setWindowManager(WindowManagerService wm) {  
 synchronized (mService) {

//保存wm引用  
 mWindowManager = wm;  
  
 mDisplayManager =  
 (DisplayManager)mService.mContext.getSystemService(Context.DISPLAY\_SERVICE);  
 mDisplayManager.registerDisplayListener(this, null);  
 //通过DisplayManager获取当前系统支持的Display,然后根据每个Display的ID创建对应的ActivityDisplay对象并保存到mActivityDisplays数组中  
 Display[] displays = mDisplayManager.getDisplays();  
 for (int displayNdx = displays.length - 1; displayNdx >= 0; --displayNdx) {  
 final int displayId = displays[displayNdx].getDisplayId();  
 ActivityDisplay activityDisplay = new ActivityDisplay(displayId);  
 if (activityDisplay.mDisplay == null) {  
 throw new IllegalStateException("Default Display does not exist");  
 }  
 mActivityDisplays.put(displayId, activityDisplay);

//获取可调整大小的task对应的display默认最小尺寸，默认为220dp  
 calculateDefaultMinimalSizeOfResizeableTasks(activityDisplay);  
 }  
 //因为是初始化的时候调用，此处初始化创建mHomeStack，并且把mFocusedStack和mLastFocusedStack都指向mHomeStack  
 mHomeStack = mFocusedStack = mLastFocusedStack =  
 getStack(HOME\_STACK\_ID, CREATE\_IF\_NEEDED, ON\_TOP);  
 //初始化InputManager  
 mInputManagerInternal = LocalServices.getService(InputManagerInternal.class);  
 }  
}

2.5.4.1.2.2 创建并初始化ActivityStack

【ASS#ActivityContainer】

ActivityContainer(int stackId) {  
 synchronized (mService) {  
 mStackId = stackId;  
 mStack = new ActivityStack(this, mRecentTasks);  
 mIdString = "ActivtyContainer{" + mStackId + "}";  
 if (DEBUG\_STACK) Slog.d(TAG\_STACK, "Creating " + this);  
 }  
}

【ActivityStack】

ActivityStack(ActivityStackSupervisor.ActivityContainer activityContainer,  
 RecentTasks recentTasks) {  
 mActivityContainer = activityContainer;  
 mStackSupervisor = activityContainer.getOuter();  
 mService = mStackSupervisor.mService;

//用来管理ActivityStack中的一些异步消息  
 mHandler = new ActivityStackHandler(mService.mHandler.getLooper());  
 mWindowManager = mService.mWindowManager;  
 mStackId = activityContainer.mStackId;  
 mCurrentUser = mService.mUserController.getCurrentUserIdLocked();  
 mRecentTasks = recentTasks;

//可调节窗口位置的task的位置  
 mTaskPositioner = mStackId == FREEFORM\_WORKSPACE\_STACK\_ID  
 ? new LaunchingTaskPositioner() : null;  
}

2.5.4.1.2.3 附着stack到Display

【ASS#ActivityContainer】

void attachToDisplayLocked(ActivityDisplay activityDisplay, boolean onTop) {  
 if (DEBUG\_STACK) Slog.d(TAG\_STACK, "attachToDisplayLocked: " + this  
 + " to display=" + activityDisplay + " onTop=" + onTop);  
 mActivityDisplay = activityDisplay;  
 mStack.attachDisplay(activityDisplay, onTop);  
 activityDisplay.attachActivities(mStack, onTop);  
}

【ActivityStack】

void attachDisplay(ActivityStackSupervisor.ActivityDisplay activityDisplay, boolean onTop) {  
 mDisplayId = activityDisplay.mDisplayId;  
 mStacks = activityDisplay.mStacks;

//把当前stack 附着到Display上面，并返回当前stack的边界，如果是全屏stack则返回的边界值为null，mBounds是Rect类型。附着比较复杂，后续分析WM和Display的时候再做分析。这里需要知道，在attach的时候WM会创建一个新的TaskStack对象，TaskStack对象也有一个stackId变量，这个变量与ActivityStack的stackID是相同的，也就是说WM也会维护一个包含TaskStack的列表，而这个列表里面的TaskStack与AMS维护的所有ActivityStack是一一对应的。同样每一个TaskStack里面都会包含一个Task对象，这个Task对象由于ActivityStack里面的TaskRecord又是一一对应的。WM用这个来处理task对应的window之间的切换显示。关于Task的创建，后续分析。  
 mBounds = mWindowManager.attachStack(mStackId, activityDisplay.mDisplayId, onTop);  
 mFullscreen = mBounds == null;  
 if (mTaskPositioner != null) {  
 mTaskPositioner.setDisplay(activityDisplay.mDisplay);  
 mTaskPositioner.configure(mBounds);  
 }  
  
 if (mStackId == DOCKED\_STACK\_ID) {  
 // If we created a docked stack we want to resize it so it resizes all other stacks  
 // in the system.  
 mStackSupervisor.resizeDockedStackLocked(  
 mBounds, null, null, null, null, PRESERVE\_WINDOWS);  
 }  
}

##### 2.5.4.2 创建新的task

【ActivityStack】

TaskRecord createTaskRecord(int taskId, ActivityInfo info, Intent intent,  
 IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,  
 boolean toTop) {  
 TaskRecord task = new TaskRecord(mService, taskId, info, intent, voiceSession,  
 voiceInteractor);  
 // add the task to stack first, mTaskPositioner might need the stack association  
 addTask(task, toTop, "createTaskRecord");  
 final boolean isLockscreenShown = mService.mLockScreenShown == LOCK\_SCREEN\_SHOWN;

//因为这里是新创建task，所以这个info就是该task里面的第一个activity，所以根据这个activity配置的布局属性设置其layout  
 if (!layoutTaskInStack(task, info.windowLayout) && mBounds != null && task.isResizeable()  
 && !isLockscreenShown) {  
 task.updateOverrideConfiguration(mBounds);  
 }  
 return task;  
}

#### 2.5.5 生成Uri Permission

【AMS】

void grantUriPermissionFromIntentLocked(int callingUid,  
 String targetPkg, Intent intent, UriPermissionOwner owner, int targetUserId) {

//检查目标activity是否需要添加URIPermission，参考2.5.5.1  
 NeededUriGrants needed = checkGrantUriPermissionFromIntentLocked(callingUid, targetPkg,  
 intent, intent != null ? intent.getFlags() : 0, null, targetUserId);  
 if (needed == null) {  
 return;  
 }  
 //生成UriPermission，参考2.5.5.2  
 grantUriPermissionUncheckedFromIntentLocked(needed, owner);  
}

参数:

callingUid 调用方UID

targetPkg 目标activity对应的包名

intent 调用intent

owner 代表目标activity，每个ActivityRecord都有一个UriPermissionOwner uriPermissions对象，如果为空，会在通过方法getUriPermissionsLocked初始化一个，UriPermissionOwner是用来标识当前ActivityRecord是否有URIPermission的

##### 2.5.5.1 检查是否需要生成UriPermission

NeededUriGrants checkGrantUriPermissionFromIntentLocked(int callingUid,  
 String targetPkg, Intent intent, int mode, NeededUriGrants needed, int targetUserId) {  
 if (targetPkg == null) {  
 throw new NullPointerException("targetPkg");  
 }  
  
 if (intent == null) {  
 return null;  
 }

//Intent在创建的时候可以通过putExtra或者setClipData来设置uri。该功能主要用来分享发起方的ContentProvider或者私有文件给目标activity，比如应用A内启动图片编辑应用B去编辑A的私有图片pic并返回，那么就可以把pic对应的uri放到intent的Extra中传递给应用B，如果pic是多张图片对应多个uri，那么就可以借用ClipData的多Item功能实现uri的传输。  
 Uri data = intent.getData();  
 ClipData clip = intent.getClipData();

//如果没有uri，则表示不需要生成UriPermission权限  
 if (data == null && clip == null) {  
 return null;  
 }  
 // Default userId for uris in the intent (if they don't specify it themselves)  
 int contentUserHint = intent.getContentUserHint();  
 if (contentUserHint == UserHandle.USER\_CURRENT) {  
 contentUserHint = UserHandle.getUserId(callingUid);  
 }  
 final IPackageManager pm = AppGlobals.getPackageManager();  
 int targetUid;  
 if (needed != null) {  
 targetUid = needed.targetUid;  
 } else {  
 try {  
 targetUid = pm.getPackageUid(targetPkg, MATCH\_DEBUG\_TRIAGED\_MISSING,  
 targetUserId);  
 } catch (RemoteException ex) {  
 return null;  
 }  
 if (targetUid < 0) {  
 if (DEBUG\_URI\_PERMISSION) Slog.v(TAG\_URI\_PERMISSION,  
 "Can't grant URI permission no uid for: " + targetPkg  
 + " on user " + targetUserId);  
 return null;  
 }  
 }  
 if (data != null) {

//解析URI，并检查URI是否合法，是否需要权限  
 GrantUri grantUri = GrantUri.resolve(contentUserHint, data);

//参考2.5.5.1.1  
 targetUid = checkGrantUriPermissionLocked(callingUid, targetPkg, grantUri, mode,  
 targetUid);  
 if (targetUid > 0) {  
 if (needed == null) {

// NeededUriGrants继承自ArrayList保存了每条uri针对具体uid需要生成的具体permission FLAG  
 needed = new NeededUriGrants(targetPkg, targetUid, mode);  
 }

//设置结果，需要生成permission  
 needed.add(grantUri);  
 }  
 }

//clipdata包含多条uri，每一条都和前面的做同样处理。针对每一条uri生成一个NeededUriGrants。  
 if (clip != null) {  
 for (int i=0; i<clip.getItemCount(); i++) {  
 Uri uri = clip.getItemAt(i).getUri();  
 if (uri != null) {  
 GrantUri grantUri = GrantUri.resolve(contentUserHint, uri);  
 targetUid = checkGrantUriPermissionLocked(callingUid, targetPkg, grantUri, mode,  
 targetUid);  
 if (targetUid > 0) {  
 if (needed == null) {  
 needed = new NeededUriGrants(targetPkg, targetUid, mode);  
 }  
 needed.add(grantUri);  
 }  
 } else {  
 Intent clipIntent = clip.getItemAt(i).getIntent();  
 if (clipIntent != null) {  
 NeededUriGrants newNeeded = checkGrantUriPermissionFromIntentLocked(  
 callingUid, targetPkg, clipIntent, mode, needed, targetUserId);  
 if (newNeeded != null) {  
 needed = newNeeded;  
 }  
 }  
 }  
 }  
 }  
 //返回所有需要产生的permission记录  
 return needed;  
}

###### 2.5.5.1.1

int checkGrantUriPermissionLocked(int callingUid, String targetPkg, GrantUri grantUri,  
 final int modeFlags, int lastTargetUid) {

//检查modeFlags是否有URI permission，如果没有，则直接返回

//AccessUriMode包括FLAG\_GRANT\_READ\_URI\_PERMISSION和FLAG\_GRANT\_WRITE\_URI\_PERMISSION，分别由Intent定义的时候指定，如果未指定这两个FLAG，那么即使配置了URI，目标activity也将没有权限访问  
 if (!Intent.isAccessUriMode(modeFlags)) {  
 return -1;  
 }  
  
 final IPackageManager pm = AppGlobals.getPackageManager();  
 //此处检查URI的scheme是否是”content://”类型的，如果不是，则直接返回，所以如果要目标activity可以访问你的私有数据，必须使用”content:”，不能使用”file://”或者其他的  
 // If this is not a content: uri, we can't do anything with it.  
 if (!ContentResolver.SCHEME\_CONTENT.equals(grantUri.uri.getScheme())) {  
 if (DEBUG\_URI\_PERMISSION) Slog.v(TAG\_URI\_PERMISSION,  
 "Can't grant URI permission for non-content URI: " + grantUri);  
 return -1;  
 }  
 //检查uri的authority是否有对应的provider存在，如果没有，则返回。  
 final String authority = grantUri.uri.getAuthority();  
 final ProviderInfo pi = getProviderInfoLocked(authority, grantUri.sourceUserId,  
 MATCH\_DEBUG\_TRIAGED\_MISSING);  
 if (pi == null) {  
 Slog.w(TAG, "No content provider found for permission check: " +  
 grantUri.uri.toSafeString());  
 return -1;  
 }  
  
 int targetUid = lastTargetUid;  
 。。。//检查uid合法性,因为这里还会涉及的跨用户访问provider的问题

//首先检查目标package是否已经拥有该权限，如果有了，则直接返回

//接着检查要访问的provider是否允许生成该权限

//最后检查发起方是否拥有访问该provider的权限

//如果上述条件都满足，则返回需要生成permission，否则返回不需要或者抛出异常。  
 。。。//代码略

return targetUid;  
}

##### 2.5.5.2 生成UriPermission

void grantUriPermissionUncheckedFromIntentLocked(NeededUriGrants needed,  
 UriPermissionOwner owner) {  
 if (needed != null) {  
 for (int i=0; i<needed.size(); i++) {  
 GrantUri grantUri = needed.get(i);

//针对每一条权限信息生成相关的权限  
 grantUriPermissionUncheckedLocked(needed.targetUid, needed.targetPkg,  
 grantUri, needed.flags, owner);  
 }  
 }  
}

void grantUriPermissionUncheckedLocked(int targetUid, String targetPkg, GrantUri grantUri,  
 final int modeFlags, UriPermissionOwner owner) {  
 if (!Intent.isAccessUriMode(modeFlags)) {  
 return;  
 }  
  
 // So here we are: the caller has the assumed permission  
 // to the uri, and the target doesn't. Let's now give this to  
 // the target.  
  
 if (DEBUG\_URI\_PERMISSION) Slog.v(TAG\_URI\_PERMISSION,  
 "Granting " + targetPkg + "/" + targetUid + " permission to " + grantUri);  
  
 final String authority = grantUri.uri.getAuthority();  
 final ProviderInfo pi = getProviderInfoLocked(authority, grantUri.sourceUserId,  
 MATCH\_DEBUG\_TRIAGED\_MISSING);  
 if (pi == null) {  
 Slog.w(TAG, "No content provider found for grant: " + grantUri.toSafeString());  
 return;  
 }  
  
 if ((modeFlags & Intent.FLAG\_GRANT\_PREFIX\_URI\_PERMISSION) != 0) {  
 grantUri.prefix = true;  
 }

//检查目标package和目标应用ID针对当前uri是否已经有权限条目存在，如果没有那就生成一个，见稍后对mGrantedUriPermissions的介绍  
 final UriPermission perm = findOrCreateUriPermissionLocked(  
 pi.packageName, targetPkg, targetUid, grantUri);

//然后针对这个perm对象生成相应的读写权限，主要是把owner加入到mReadOwners和(或) mWriteOwners中，参考稍后对UriPermission的介绍。到此为止，UriPermission生成完成。  
 perm.grantModes(modeFlags, owner);  
}

//这里有两个对象需要注意

//1. GrantUri 他是AMS的内部类，维护了一个具体的URI以及一个prefix标量

//2. UriPermission 该类维护了一个GrantUri对象，还维护了两个列表ArraySet<UriPermissionOwner> mReadOwners和ArraySet<UriPermissionOwner> mWriteOwners用来保存对当前GrantUri代表的URI具有读或者写权限的UriPermissionOwner，前面介绍过ActivityRecord会维护一个UriPermissionOwner对象，也就是说UriPermissionOwner其实就是可以代表一个ActivityRecord

private UriPermission findOrCreateUriPermissionLocked(String sourcePkg,  
 String targetPkg, int targetUid, GrantUri grantUri) {

// SparseArray<ArrayMap<GrantUri, UriPermission>> mGrantedUriPermissions维护一个ArrayMap的数组,每个ArrayMap保存了某一个目标uid targetUid

所拥有的所有URI的GrantUri以及该GrantUri对应的UriPermission。也就是说mGrantedUriPermissions保存了AMS经手的所有uri permission信息  
 ArrayMap<GrantUri, UriPermission> targetUris = mGrantedUriPermissions.get(targetUid);  
 if (targetUris == null) {  
 targetUris = Maps.newArrayMap();

//针对不存在的情况创建一个targetUris并添加到mGrantedUriPermissions里面  
 mGrantedUriPermissions.put(targetUid, targetUris);  
 }  
 //检查grantUri对应的UriPermission是否为空，为空表示是刚创建的，需要初始化UriPermission。  
 UriPermission perm = targetUris.get(grantUri);  
 if (perm == null) {  
 perm = new UriPermission(sourcePkg, targetPkg, targetUid, grantUri);  
 targetUris.put(grantUri, perm);  
 }  
 //返回找到(创建)的结果  
 return perm;  
}

描述：这里提到的targetUid等UID一般都指的是目标应用的应用ID，而不是多用户里面的userID。

### 2.6 startActivityLocked 配置activity

功能简介：

添加目标activity到合适的task中

设置目标activity对应window的启动动画类型

创建AppWindowToken和WM端的task

创建启动窗口

【ActivityStack.java】

final void startActivityLocked(ActivityRecord r, boolean newTask, boolean keepCurTransition,  
 ActivityOptions options) {  
 TaskRecord rTask = r.task;  
 final int taskId = rTask.taskId;  
 // mLaunchTaskBehind tasks get placed at the back of the task stack.  
 if (!r.mLaunchTaskBehind && (taskForIdLocked(taskId) == null || newTask)) {  
 // Last activity in task had been removed or ActivityManagerService is reusing task.  
 // Insert or replace.  
 // Might not even be in.  
 insertTaskAtTop(rTask, r);  
 mWindowManager.moveTaskToTop(taskId);  
 }  
 TaskRecord task = null;  
 if (!newTask) {

//启动一个已经存在的task，在mTaskHistory里面查找符合目标ActivityRecord对应的task的task  
 // If starting in an existing task, find where that is...  
 boolean startIt = true;  
 for (int taskNdx = mTaskHistory.size() - 1; taskNdx >= 0; --taskNdx) {  
 task = mTaskHistory.get(taskNdx);  
 if (task.getTopActivity() == null) {  
 // All activities in task are finishing.  
 continue;  
 }  
 if (task == r.task) {  
 // Here it is! Now, if this is not yet visible to the  
 // user, then just add it without starting; it will  
 // get started when the user navigates back to it.  
 if (!startIt) {  
 //目标ActivityRecord不需要启动到前台

//添加目标acitivity到目标task中，参考2.6.1  
 task.addActivityToTop(r);

//标识inHistory  
 r.putInHistory();

//覆盖目标task的configuration，因为新的activity携带的configuration与目标task中本来的configuration可能不同  
 addConfigOverride(r, task);

// VALIDATE\_TOKENS为false  
 if (VALIDATE\_TOKENS) {  
 validateAppTokensLocked();  
 }

//因为被加入到了后台task中，所以阻塞所有options  
 ActivityOptions.abort(options);  
 return;  
 }

//到这里表示startIt为true，也就是代表找到了目标task  
 break;  
 } else if (task.numFullscreen > 0) {

//到这里表示目标task不在最上层，并且位于目标task之上的task中至少有一个ActivityRecord是全屏的，startIt置位false表示目标activity将不会去显示给用户，只会加入的目标task里面  
 startIt = false;  
 }  
 }  
 }  
  
 // Place a new activity at top of stack, so it is next to interact  
 // with the user.

//到这里表示要么在stack中找到了目标task并且需要显示，要么就是没有找到，接着就是尝试把目标activity加入到目标task中了  
  
 // If we are not placing the new activity frontmost, we do not want  
 // to deliver the onUserLeaving callback to the actual frontmost  
 // activity  
 if (task == r.task && mTaskHistory.indexOf(task) != (mTaskHistory.size() - 1)) {

//这里表示虽然找到目标task，而且目标activity需要显示，但是目标task并不是stack中的最上层的task，在目标task之上还有其他task，只不过那些task所包含的activity都不是全屏的。针对这种情况，把mUserLeaving置位false，也就是不会触发activity生命周期中的onUserLeaveHint回调。  
 mStackSupervisor.mUserLeaving = false;  
 if (DEBUG\_USER\_LEAVING) Slog.v(TAG\_USER\_LEAVING,  
 "startActivity() behind front, mUserLeaving=false");  
 }  
 //无论是否找到，都确保task指向目标activity携带过来的应该所属的task  
 task = r.task;  
  
 // Slot the activity into the history stack and proceed  
 if (DEBUG\_ADD\_REMOVE) Slog.i(TAG, "Adding activity " + r + " to stack to task " + task,  
 new RuntimeException("here").fillInStackTrace());

//把目标activity加入到目标task顶部，参考2.6.1  
 task.addActivityToTop(r);

//这个调用的主要作用是确保task中的根activity正确。当task中的activity发生移动或者finish的时候，根activity可能发生改变，所以需要确保正确。这里主要是找到并把ActivityRecord中的变量frontOfTask置为true，查找的方式是从mActivitys的根部开始，排除掉所有正在结束的activity，然后把找到的第一个合适的ActivityRecord的frontOfTask置为true，并把其它的都置为false。  
 task.setFrontOfTask();  
 //置位变量inHistory为ture  
 r.putInHistory();  
 if (!isHomeStack() || numActivities() > 0) {  
 // We want to show the starting preview window if we are  
 // switching to a new task, or the next activity's process is  
 // not currently running.  
 boolean showStartingIcon = newTask;  
 ProcessRecord proc = r.app;  
 if (proc == null) {  
 proc = mService.mProcessNames.get(r.processName, r.info.applicationInfo.uid);  
 }  
 if (proc == null || proc.thread == null) {  
 showStartingIcon = true;  
 }  
 if (DEBUG\_TRANSITION) Slog.v(TAG\_TRANSITION,  
 "Prepare open transition: starting " + r);

//准备activity启动的动画类型，由WM去执行，\*#\*#1000#  
 if ((r.intent.getFlags() & Intent.FLAG\_ACTIVITY\_NO\_ANIMATION) != 0) {

//没有动画的情况  
 mWindowManager.prepareAppTransition(TRANSIT\_NONE, keepCurTransition);

// mNoAnimActivities用来记录当前栈里面所有无动画的activity  
 mNoAnimActivities.add(r);  
 } else {  
 mWindowManager.prepareAppTransition(newTask  
 ? r.mLaunchTaskBehind  
 ? TRANSIT\_TASK\_OPEN\_BEHIND  
 : TRANSIT\_TASK\_OPEN  
 : TRANSIT\_ACTIVITY\_OPEN, keepCurTransition);  
 mNoAnimActivities.remove(r);  
 }

//addAppToken调用WM 的addAppToken方法，参考2.6.2  
 addConfigOverride(r, task);

//接下来创建activity启动窗口（View）  
 boolean doShow = true;  
 if (newTask) {  
 // Even though this activity is starting fresh, we still need  
 // to reset it to make sure we apply affinities to move any  
 // existing activities from other tasks in to it.  
 // If the caller has requested that the target task be  
 // reset, then do so.  
 if ((r.intent.getFlags() & Intent.FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED) != 0) {  
 resetTaskIfNeededLocked(r, r);  
 doShow = topRunningNonDelayedActivityLocked(null) == r;  
 }  
 } else if (options != null && options.getAnimationType()  
 == ActivityOptions.ANIM\_SCENE\_TRANSITION) {  
 doShow = false;  
 }  
 if (r.mLaunchTaskBehind) {  
 // Don't do a starting window for mLaunchTaskBehind. More importantly make sure we  
 // tell WindowManager that r is visible even though it is at the back of the stack.  
 mWindowManager.setAppVisibility(r.appToken, true);  
 ensureActivitiesVisibleLocked(null, 0, !PRESERVE\_WINDOWS);  
 } else if (SHOW\_APP\_STARTING\_PREVIEW && doShow) {  
 // Figure out if we are transitioning from another activity that is  
 // "has the same starting icon" as the next one. This allows the  
 // window manager to keep the previous window it had previously  
 // created, if it still had one.

//一般都会走到这里，首先检查是否存在running状态的activity（即正在启动，但是启动动画还没有完成），在满足条件的情况下，会复用这个找到的activity的启动动画。  
 ActivityRecord prev = r.task.topRunningActivityWithStartingWindowLocked();  
 if (prev != null) {

//如果找到的activity的task和目标activity task不同，或者找到的activity已经处于visible状态，则不能复用。  
 // We don't want to reuse the previous starting preview if:  
 // (1) The current activity is in a different task.  
 if (prev.task != r.task) {  
 prev = null;  
 }  
 // (2) The current activity is already displayed.  
 else if (prev.nowVisible) {

//ActivityRecord的nowVisible是在AppWindowToken在窗口绘制完成之后通过发送handler消息REPORT\_APPLICATION\_TOKEN\_DRAWN给WMS的mH（即WMS的内部类H），mH收到该消息后通过binder调用调到ActivityRecord的windowsVisible方法，然后调用windowsVisibleLocked把nowVisible置为true。  
 prev = null;  
 }  
 }

//调用WM构建启动窗口，参考2.6.3  
 r.showStartingWindow(prev, showStartingIcon);  
 }  
 } else {  
 // If this is the first activity, don't do any fancy animations,  
 // because there is nothing for it to animate on top of.

//这里代表的是launcher中的第一个activity，不需要启动动画。  
 addConfigOverride(r, task);  
 ActivityOptions.abort(options);  
 options = null;  
 }

// VALIDATE\_TOKENS为false  
 if (VALIDATE\_TOKENS) {  
 validateAppTokensLocked();  
 }  
}

参数：

keepCurTransition 代表目标activity的启动动画不做改变，保持之前的，这个一般是设置了FLAG\_ACTIVITY\_CLEAR\_TOP属性之后，目标activity所要加入的task和reused activity存在，所以只需要保持原来的动画设置即可，此处的动画指的是window的动画，参考AppTransition.java中TRANSIT\_\*的定义和注释。

#### 2.6.1 addActivityToTop 添加目标activity到目标task 的top位置

【TaskRecord.java】

void addActivityToTop(ActivityRecord r) {

//index 为mActivities.size()，即以mActivities的末尾代表task的顶部  
 addActivityAtIndex(mActivities.size(), r);  
}  
  
void addActivityAtIndex(int index, ActivityRecord r) {  
 // Remove r first, and if it wasn't already in the list and it's fullscreen, count it.

//如果mActivities中包含目标activity，则remove函数返回true，一般肯定不会包含（注意：一个task中可能包含某个activity的多个实例，但是这些实例并不是同一个对象）。  
 if (!mActivities.remove(r) && r.fullscreen) {  
 // Was not previously in list.

//目标activity是全屏的，添加全屏计数  
 numFullscreen++;  
 }  
 // Only set this based on the first activity  
 if (mActivities.isEmpty()) {

//目标Activity是这个task中的第一个activity，所以会把目标activity的一些属性设置为task的属性。

//之前提过，task和activity使用的是同样的类型定义，就提现在这里了，直接把activity的类型赋值给task。  
 taskType = r.mActivityType;

//记录这个task是否是persistable，关于persist见后面描述

//在activity里面有几个生命周期函数  
 isPersistable = r.isPersistable();  
 mCallingUid = r.launchedFromUid;  
 mCallingPackage = r.launchedFromPackage;  
 // Clamp to [1, max].

//记录最大Recents数，r.info.maxRecents在未指定的情况下默认为15，ActivityManager.getMaxAppRecentsLimitStatic()的默认值是18或者24，低内存手机是18.不同厂商可以配置这个值，在AM中。maxRecents代表的意义是在recents列表中最多会保存的以这个activity为根activity的task个数，还记得前面提到过得Intent# FLAG\_ACTIVITY\_NEW\_DOCUMENT属性吗？没错，这个属性设置为always的时候可以让启动的每个activity都在单独的task中并呈现在recents列表中。  
 maxRecents = Math.min(Math.max(r.info.maxRecents, 1),  
 ActivityManager.getMaxAppRecentsLimitStatic());  
 } else {  
 // Otherwise make all added activities match this one.

//如果不是根Activity，则忽略activity自己指定的type，即同一个task中的activity都将指定为根activity的type  
 r.mActivityType = taskType;  
 }  
  
 final int size = mActivities.size();  
   
 if (index == size && size > 0) {  
 final ActivityRecord top = mActivities.get(size - 1);  
 if (top.mTaskOverlay) {  
 // Place below the task overlay activity since the overlay activity should always  
 // be on top.  
 index--;  
 }  
 }  
 //添加目标activity到mActivities中，位置为index，一般index为mActivities的大小，也就是top  
 mActivities.add(index, r);  
 updateEffectiveIntent();  
 if (r.isPersistable()) {  
 mService.notifyTaskPersisterLocked(this, false);  
 }  
}

描述:

1. persist

<attr name="persistableMode">

<!-- The default. If this activity forms the root of a task then that task will be

persisted across reboots but only the launching intent will be used. If the task

relinquishes its identity then the intent used is that of the topmost inherited

identity. All activities above this activity in the task will not be persisted.

In addition this activity will not be passed a PersistableBundle into which it

could have stored its state. -->

<enum name="persistRootOnly" value="0" />

<!-- If this activity forms the root of a task then that task will not be persisted

across reboots -->

<enum name="persistNever" value="1" />

<!-- If this activity forms the root of a task then the task and this activity will

be persisted across reboots. If the activity above this activity is also

tagged with the attribute <code>"persist"</code> then it will be persisted as well.

And so on up the task stack until either an activity without the

<code>persistableMode="persistAcrossReboots"</code> attribute or one that was launched

with the flag Intent.FLAG\_CLEAR\_TASK\_WHEN\_RESET is encountered.

<p>Activities that are declared with the persistAcrossReboots attribute will be

provided with a PersistableBundle in onSavedInstanceState(), These activities may

use this PeristableBundle to save their state. Then, following a reboot, that

PersistableBundle will be provided back to the activity in its onCreate() method. -->

<enum name="persistAcrossReboots" value="2" />

</attr>

在activity中有如下几个生命周期函数：

onSaveInstanceState(Bundle outState, PersistableBundle outPersistentState)

onCreate(Bundle savedInstanceState, PersistableBundle persistentState)

onRestoreInstanceState(Bundle savedInstanceState,

PersistableBundle persistentState)

其中参数PersistableBundle就是与这个persist属性相关。

#### 2.6.2 创建目标activity对应的AppWindowToken以及相应的WM端Task对象

功能：

创建目标activity的AppWindowToken

创建目标activity所在task对应的WM端的Task对象

【ActivityStack.java】

void addConfigOverride(ActivityRecord r, TaskRecord task) {  
 final Rect bounds = task.updateOverrideConfigurationFromLaunchBounds();  
 // 调用WM方法，其中r.appToken是ActivityRecord对应的WM token，在ActivityRecord初始化的时候通过appToken = new Token(this, service)创建的唯一标识当前activity的window manager Token。是IApplicationToken类型的binder对象。  
 mWindowManager.addAppToken(task.mActivities.indexOf(r), r.appToken,  
 r.task.taskId, mStackId, r.info.screenOrientation, r.fullscreen,  
 (r.info.flags & FLAG\_SHOW\_FOR\_ALL\_USERS) != 0, r.userId, r.info.configChanges,  
 task.voiceSession != null, r.mLaunchTaskBehind, bounds, task.mOverrideConfig,  
 task.mResizeMode, r.isAlwaysFocusable(), task.isHomeTask(),  
 r.appInfo.targetSdkVersion, r.mRotationAnimationHint);  
 r.taskConfigOverride = task.mOverrideConfig;  
}

【WindowManagerService.java】 这里先简单介绍一下WM对于token和task的创建，会省略部分代码

public void addAppToken(int addPos, IApplicationToken token, int taskId, int stackId,  
 int requestedOrientation, boolean fullscreen, boolean showForAllUsers, int userId,  
 int configChanges, boolean voiceInteraction, boolean launchTaskBehind,  
 Rect taskBounds, Configuration config, int taskResizeMode, boolean alwaysFocusable,  
 boolean homeTask, int targetSdkVersion, int rotationAnimationHint) {  
 synchronized(mWindowMap) {

//根据目标Activity对应的token binder对象查找AppWindowToken，如果存在，则直接返回，表示不需要添加。  
 AppWindowToken atoken = findAppWindowToken(token.asBinder());  
 if (atoken != null) {  
 Slog.w(TAG\_WM, "Attempted to add existing app token: " + token);  
 return;  
 }

//如果不存在，则根据传进来的各个参数数据创建一个新的AppWindowToken对象  
 atoken = new AppWindowToken(this, token, voiceInteraction);  
 atoken.inputDispatchingTimeoutNanos = inputDispatchingTimeoutNanos;  
 atoken.appFullscreen = fullscreen;  
 atoken.showForAllUsers = showForAllUsers;  
 atoken.targetSdk = targetSdkVersion;  
 atoken.requestedOrientation = requestedOrientation;  
 atoken.layoutConfigChanges = (configChanges &  
 (ActivityInfo.CONFIG\_SCREEN\_SIZE | ActivityInfo.CONFIG\_ORIENTATION)) != 0;  
 atoken.mLaunchTaskBehind = launchTaskBehind;  
 atoken.mAlwaysFocusable = alwaysFocusable;   
 atoken.mRotationAnimationHint = rotationAnimationHint;  
 //查找获取目标activity对应的task在WM这边的Task对象  
 Task task = mTaskIdToTask.get(taskId);  
 if (task == null) {

//如果不存在，则创建一个Task对象  
 task = createTaskLocked(taskId, stackId, userId, atoken, taskBounds, config);  
 }

//Task会维护一个AppTokenList mAppTokens对象，是ArrayList类型，它保存了属于这个Task的所有AppWindowToken。这里负责把新建的token加入到task里面。  
 task.addAppToken(addPos, atoken, taskResizeMode, homeTask);  
 // mTokenMap映射一个Token Ibinder到WindowToken，这个map保存了WMS管理的所有WindowToken。请注意AppWindowToken继承自WindowToken，主要代表应用对应的token（实际上是activity对应的Token）。  
 mTokenMap.put(token.asBinder(), atoken);  
  
 // Application tokens start out hidden.  
 atoken.hidden = true;  
 atoken.hiddenRequested = true;  
 }  
}

#### 2.6.3 合成startingView

【ActivityRecord.java】

void showStartingWindow(ActivityRecord prev, boolean createIfNeeded) {  
 final CompatibilityInfo compatInfo =  
 service.compatibilityInfoForPackageLocked(info.applicationInfo);  
 final boolean shown = service.mWindowManager.setAppStartingWindow(  
 appToken, packageName, theme, compatInfo, nonLocalizedLabel, labelRes, icon,  
 logo, windowFlags, prev != null ? prev.appToken : null, createIfNeeded);  
 if (shown) {  
 mStartingWindowState = STARTING\_WINDOW\_SHOWN;  
 }  
}

【WMS】 简介

public boolean setAppStartingWindow(IBinder token, String pkg,  
 int theme, CompatibilityInfo compatInfo,  
 CharSequence nonLocalizedLabel, int labelRes, int icon, int logo,  
 int windowFlags, IBinder transferFrom, boolean createIfNeeded) {  
 synchronized(mWindowMap) {  
 AppWindowToken wtoken = findAppWindowToken(token);//目标Activity token  
 //startingData代表启动窗口数据，如果已经存在则直接返回  
 if (wtoken.startingData != null) {  
 return false;  
 }  
  
 。。。//处理透明窗口等几个特殊的窗口style，在window 主题中配置的，对于这些特殊窗口不需要设置启动窗口。

//如果存在可以复用的启动窗口，则直接复用。  
 if (transferStartingWindow(transferFrom, wtoken)) {  
 return true;  
 }  
  
 // There is no existing starting window, and the caller doesn't  
 // want us to create one, so that's it!  
 if (!createIfNeeded) {  
 return false;  
 }  
  
 //创建启动窗口的相关数据，这些数据都是ActivityRecord调用的时候传递进来的  
 wtoken.startingData = new StartingData(pkg, theme, compatInfo, nonLocalizedLabel,  
 labelRes, icon, logo, windowFlags);  
 Message m = mH.obtainMessage(H.ADD\_STARTING, wtoken);  
 //立即执行  
 mH.sendMessageAtFrontOfQueue(m);  
 }  
 return true;  
}

【H extends Handler】【WMS】

case ADD\_STARTING: {  
 。。。  
 View view = null;  
 。。。

//此处mPolicy对应的是PhoneWindowManager,调用addStartingWindow会结合startingData和系统DecorView合成一个View，然后调用WindowManager的addView添加到窗口，WindowManager是一个接口，其实现为WindowManagerImpl.java，并返回合成的view。关于addView方法会在activity绘制的时候再做介绍  
 view = mPolicy.addStartingWindow(wtoken.token, sd.pkg, sd.theme,  
 sd.compatInfo, sd.nonLocalizedLabel, sd.labelRes, sd.icon, sd.logo,  
 sd.windowFlags, overrideConfig);  
 。。。  
 //最后把合成的view赋值给wtoken.startingView以备后用  
 wtoken.startingView = view;  
 。。。  
} break;

### 2.7 resumeFocusedStackTopActivityLocked 正式启动activity

【ASS】resume当前focused stack中的top activity

boolean resumeFocusedStackTopActivityLocked(  
 ActivityStack targetStack, ActivityRecord target, ActivityOptions targetOptions) {

//确保目标栈获得焦点  
 if (targetStack != null && isFocusedStack(targetStack)) {

//最后会调用到ActivityStack的result = resumeTopActivityInnerLocked(prev, options);  
 return targetStack.resumeTopActivityUncheckedLocked(target, targetOptions);  
 }  
 final ActivityRecord r = mFocusedStack.topRunningActivityLocked();  
 if (r == null || r.state != RESUMED) {  
 mFocusedStack.resumeTopActivityUncheckedLocked(null, null);  
 }  
 return false;  
}

【ActivityStack.java】

private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions options) {  
 if (!mService.mBooting && !mService.mBooted) {  
 // Not ready yet!  
 return false;  
 }  
 //与ActivityView有关，普通activity都没有parent一说  
 ActivityRecord parent = mActivityContainer.mParentActivity;  
 if ((parent != null && parent.state != ActivityState.RESUMED) ||  
 !mActivityContainer.isAttachedLocked()) {  
 return false;  
 }  
 //取消其它stack中正在启动的activity的启动窗口，必须在至少一个全屏activity之后才会被取消，而在顶部的或者焦点状态的activity的启动窗口(如果有)是不会被取消的。需要注意，在顶部的activity并不一定就是获取到焦点的activity，因为它可能不是全屏的，比如多窗口等等。而第一个全屏activity也是需要保留启动窗口的（如果有），因为是可见的。

//关于启动窗口的创建参考2.6.3  
 mStackSupervisor.cancelInitializingActivities();  
  
 // Find the first activity that is not finishing.  
 final ActivityRecord next = topRunningActivityLocked();  
  
 // Remember how we'll process this pause/resume situation, and ensure  
 // that the state is reset however we wind up proceeding.  
 final boolean userLeaving = mStackSupervisor.mUserLeaving;  
 mStackSupervisor.mUserLeaving = false;  
  
 final TaskRecord prevTask = prev != null ? prev.task : null;  
 if (next == null) {  
 // There are no more activities!  
 final String reason = "noMoreActivities";  
 final int returnTaskType = prevTask == null || !prevTask.isOverHomeStack()  
 ? HOME\_ACTIVITY\_TYPE : prevTask.getTaskToReturnTo();  
 if (!mFullscreen && adjustFocusToNextFocusableStackLocked(returnTaskType, reason)) {  
 // Try to move focus to the next visible stack with a running activity if this  
 // stack is not covering the entire screen.  
 return mStackSupervisor.resumeFocusedStackTopActivityLocked(  
 mStackSupervisor.getFocusedStack(), prev, null);  
 }  
  
 // Let's just start up the Launcher...  
 ActivityOptions.abort(options);  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: No more activities go home");  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 // Only resume home if on home display  
 return isOnHomeDisplay() &&  
 mStackSupervisor.resumeHomeStackTask(returnTaskType, prev, reason);  
 }  
  
 next.delayedResume = false;  
  
 // If the top activity is the resumed one, nothing to do.  
 if (mResumedActivity == next && next.state == ActivityState.RESUMED &&  
 mStackSupervisor.allResumedActivitiesComplete()) {  
 // Make sure we have executed any pending transitions, since there  
 // should be nothing left to do at this point.  
 mWindowManager.executeAppTransition();  
 mNoAnimActivities.clear();  
 ActivityOptions.abort(options);  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: Top activity resumed " + next);  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return false;  
 }  
  
 final TaskRecord nextTask = next.task;  
 if (prevTask != null && prevTask.stack == this &&  
 prevTask.isOverHomeStack() && prev.finishing && prev.frontOfTask) {  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 if (prevTask == nextTask) {  
 prevTask.setFrontOfTask();  
 } else if (prevTask != topTask()) {  
 // This task is going away but it was supposed to return to the home stack.  
 // Now the task above it has to return to the home task instead.  
 final int taskNdx = mTaskHistory.indexOf(prevTask) + 1;  
 mTaskHistory.get(taskNdx).setTaskToReturnTo(HOME\_ACTIVITY\_TYPE);  
 } else if (!isOnHomeDisplay()) {  
 return false;  
 } else if (!isHomeStack()){  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: Launching home next");  
 final int returnTaskType = prevTask == null || !prevTask.isOverHomeStack() ?  
 HOME\_ACTIVITY\_TYPE : prevTask.getTaskToReturnTo();  
 return isOnHomeDisplay() &&  
 mStackSupervisor.resumeHomeStackTask(returnTaskType, prev, "prevFinished");  
 }  
 }  
  
 // If we are sleeping, and there is no resumed activity, and the top  
 // activity is paused, well that is the state we want.  
 if (mService.isSleepingOrShuttingDownLocked()  
 && mLastPausedActivity == next  
 && mStackSupervisor.allPausedActivitiesComplete()) {  
 // Make sure we have executed any pending transitions, since there  
 // should be nothing left to do at this point.  
 mWindowManager.executeAppTransition();  
 mNoAnimActivities.clear();  
 ActivityOptions.abort(options);  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: Going to sleep and all paused");  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return false;  
 }  
  
 // Make sure that the user who owns this activity is started. If not,  
 // we will just leave it as is because someone should be bringing  
 // another user's activities to the top of the stack.  
 if (!mService.mUserController.hasStartedUserState(next.userId)) {  
 Slog.w(TAG, "Skipping resume of top activity " + next  
 + ": user " + next.userId + " is stopped");  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return false;  
 }  
  
 // The activity may be waiting for stop, but that is no longer  
 // appropriate for it.  
 mStackSupervisor.mStoppingActivities.remove(next);  
 mStackSupervisor.mGoingToSleepActivities.remove(next);  
 next.sleeping = false;  
 mStackSupervisor.mWaitingVisibleActivities.remove(next);  
  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH, "Resuming " + next);  
  
 // If we are currently pausing an activity, then don't do anything until that is done.  
 if (!mStackSupervisor.allPausedActivitiesComplete()) {  
 if (DEBUG\_SWITCH || DEBUG\_PAUSE || DEBUG\_STATES) Slog.v(TAG\_PAUSE,  
 "resumeTopActivityLocked: Skip resume: some activity pausing.");  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return false;  
 }  
  
 mStackSupervisor.setLaunchSource(next.info.applicationInfo.uid);  
  
 // We need to start pausing the current activity so the top one can be resumed...  
 final boolean dontWaitForPause = (next.info.flags & FLAG\_RESUME\_WHILE\_PAUSING) != 0;  
 boolean pausing = mStackSupervisor.pauseBackStacks(userLeaving, next, dontWaitForPause);  
 if (mResumedActivity != null) {  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: Pausing " + mResumedActivity);  
 pausing |= startPausingLocked(userLeaving, false, next, dontWaitForPause);  
 }  
 if (pausing) {  
 if (DEBUG\_SWITCH || DEBUG\_STATES) Slog.v(TAG\_STATES,  
 "resumeTopActivityLocked: Skip resume: need to start pausing");  
 // At this point we want to put the upcoming activity's process  
 // at the top of the LRU list, since we know we will be needing it  
 // very soon and it would be a waste to let it get killed if it  
 // happens to be sitting towards the end.  
 if (next.app != null && next.app.thread != null) {  
 mService.updateLruProcessLocked(next.app, true, null);  
 }  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
 } else if (mResumedActivity == next && next.state == ActivityState.RESUMED &&  
 mStackSupervisor.allResumedActivitiesComplete()) {  
 // It is possible for the activity to be resumed when we paused back stacks above if the  
 // next activity doesn't have to wait for pause to complete.  
 // So, nothing else to-do except:  
 // Make sure we have executed any pending transitions, since there  
 // should be nothing left to do at this point.  
 mWindowManager.executeAppTransition();  
 mNoAnimActivities.clear();  
 ActivityOptions.abort(options);  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "resumeTopActivityLocked: Top activity resumed (dontWaitForPause) " + next);  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
 }  
  
 // If the most recent activity was noHistory but was only stopped rather  
 // than stopped+finished because the device went to sleep, we need to make  
 // sure to finish it as we're making a new activity topmost.  
 if (mService.isSleepingLocked() && mLastNoHistoryActivity != null &&  
 !mLastNoHistoryActivity.finishing) {  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES,  
 "no-history finish of " + mLastNoHistoryActivity + " on new resume");  
 requestFinishActivityLocked(mLastNoHistoryActivity.appToken, Activity.RESULT\_CANCELED,  
 null, "resume-no-history", false);  
 mLastNoHistoryActivity = null;  
 }  
  
 if (prev != null && prev != next) {  
 if (!mStackSupervisor.mWaitingVisibleActivities.contains(prev)  
 && next != null && !next.nowVisible) {  
 mStackSupervisor.mWaitingVisibleActivities.add(prev);  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH,  
 "Resuming top, waiting visible to hide: " + prev);  
 } else {  
 // The next activity is already visible, so hide the previous  
 // activity's windows right now so we can show the new one ASAP.  
 // We only do this if the previous is finishing, which should mean  
 // it is on top of the one being resumed so hiding it quickly  
 // is good. Otherwise, we want to do the normal route of allowing  
 // the resumed activity to be shown so we can decide if the  
 // previous should actually be hidden depending on whether the  
 // new one is found to be full-screen or not.  
 if (prev.finishing) {  
 mWindowManager.setAppVisibility(prev.appToken, false);  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH,  
 "Not waiting for visible to hide: " + prev + ", waitingVisible="  
 + mStackSupervisor.mWaitingVisibleActivities.contains(prev)  
 + ", nowVisible=" + next.nowVisible);  
 } else {  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH,  
 "Previous already visible but still waiting to hide: " + prev  
 + ", waitingVisible="  
 + mStackSupervisor.mWaitingVisibleActivities.contains(prev)  
 + ", nowVisible=" + next.nowVisible);  
 }  
 }  
 }  
  
 // Launching this app's activity, make sure the app is no longer  
 // considered stopped.  
 try {  
 AppGlobals.getPackageManager().setPackageStoppedState(  
 next.packageName, false, next.userId); /\* TODO: Verify if correct userid \*/  
 } catch (RemoteException e1) {  
 } catch (IllegalArgumentException e) {  
 Slog.w(TAG, "Failed trying to unstop package "  
 + next.packageName + ": " + e);  
 }  
  
 // We are starting up the next activity, so tell the window manager  
 // that the previous one will be hidden soon. This way it can know  
 // to ignore it when computing the desired screen orientation.  
 boolean anim = true;  
 if (prev != null) {  
 if (prev.finishing) {  
 if (DEBUG\_TRANSITION) Slog.v(TAG\_TRANSITION,  
 "Prepare close transition: prev=" + prev);  
 if (mNoAnimActivities.contains(prev)) {  
 anim = false;  
 mWindowManager.prepareAppTransition(TRANSIT\_NONE, false);  
 } else {  
 mWindowManager.prepareAppTransition(prev.task == next.task  
 ? TRANSIT\_ACTIVITY\_CLOSE  
 : TRANSIT\_TASK\_CLOSE, false);  
 }  
 mWindowManager.setAppVisibility(prev.appToken, false);  
 } else {  
 if (DEBUG\_TRANSITION) Slog.v(TAG\_TRANSITION,  
 "Prepare open transition: prev=" + prev);  
 if (mNoAnimActivities.contains(next)) {  
 anim = false;  
 mWindowManager.prepareAppTransition(TRANSIT\_NONE, false);  
 } else {  
 mWindowManager.prepareAppTransition(prev.task == next.task  
 ? TRANSIT\_ACTIVITY\_OPEN  
 : next.mLaunchTaskBehind  
 ? TRANSIT\_TASK\_OPEN\_BEHIND  
 : TRANSIT\_TASK\_OPEN, false);  
 }  
 }  
 } else {  
 if (DEBUG\_TRANSITION) Slog.v(TAG\_TRANSITION, "Prepare open transition: no previous");  
 if (mNoAnimActivities.contains(next)) {  
 anim = false;  
 mWindowManager.prepareAppTransition(TRANSIT\_NONE, false);  
 } else {  
 mWindowManager.prepareAppTransition(TRANSIT\_ACTIVITY\_OPEN, false);  
 }  
 }  
  
 Bundle resumeAnimOptions = null;  
 if (anim) {  
 ActivityOptions opts = next.getOptionsForTargetActivityLocked();  
 if (opts != null) {  
 resumeAnimOptions = opts.toBundle();  
 }  
 next.applyOptionsLocked();  
 } else {  
 next.clearOptionsLocked();  
 }  
  
 ActivityStack lastStack = mStackSupervisor.getLastStack();

//判断要执行的activity对应的app和thread是否为空，如果为空，则表示需要重新启动activity，如果不为空，则尝试执行resume Activity

//app为目标Activity所在的应用进程，这个进程是在启动应用的时候（一般是第一个activity）通过Zygote孵化出来的，启动进程的时候会给进程创建一个名字，这个名字通过manifest的Application标签的process属性指定，如果未指定则默认为包名。

//在启动activity并创建ActivityRecord的时候会检查activity的标签属性是否指定了android:process属性，如果指定了则该activity将运行于新的process，如果未指定，则默认使用Application标签的process name。

//在每个activity第一次启动的时候app都为空，在调用到startSpecificActivityLocked的时候会通过activity的process name去查找系统进程里面是否有这个进程，如果有，则会初始化该activity的app对象，于是当再次启动这个activity的时候app就不为空，这也就代表该activity已经启动了，只需要resume就行。第一次启动activity参考2.8

//thread 表示目标activity对应ApplicationThread 对象，这个在后续activity启动的时候再分析，这里只需要知道当目标activity已经启动并保存在栈里面的时候thread就一定不为空，也就是说符合resume条件，直接resume就可以。一般app不为空的话thread也不为空。  
 if (next.app != null && next.app.thread != null) {

//进入resume流程  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH, "Resume running: " + next  
 + " stopped=" + next.stopped + " visible=" + next.visible);  
  
 // If the previous activity is translucent, force a visibility update of  
 // the next activity, so that it's added to WM's opening app list, and  
 // transition animation can be set up properly.  
 // For example, pressing Home button with a translucent activity in focus.  
 // Launcher is already visible in this case. If we don't add it to opening  
 // apps, maybeUpdateTransitToWallpaper() will fail to identify this as a  
 // TRANSIT\_WALLPAPER\_OPEN animation, and run some funny animation.  
 final boolean lastActivityTranslucent = lastStack != null  
 && (!lastStack.mFullscreen  
 || (lastStack.mLastPausedActivity != null  
 && !lastStack.mLastPausedActivity.fullscreen));  
  
 // This activity is now becoming visible.  
 if (!next.visible || next.stopped || lastActivityTranslucent) {  
 mWindowManager.setAppVisibility(next.appToken, true);  
 }  
  
 // schedule launch ticks to collect information about slow apps.  
 next.startLaunchTickingLocked();  
  
 ActivityRecord lastResumedActivity =  
 lastStack == null ? null :lastStack.mResumedActivity;  
 ActivityState lastState = next.state;  
  
 mService.updateCpuStats();  
  
 if (DEBUG\_STATES) Slog.v(TAG\_STATES, "Moving to RESUMED: " + next + " (in existing)");  
 next.state = ActivityState.RESUMED;  
 mResumedActivity = next;  
 next.task.touchActiveTime();  
 mRecentTasks.addLocked(next.task);  
 mService.updateLruProcessLocked(next.app, true, null);  
 updateLRUListLocked(next);  
 mService.updateOomAdjLocked();  
  
 // Have the window manager re-evaluate the orientation of  
 // the screen based on the new activity order.  
 boolean notUpdated = true;  
 if (mStackSupervisor.isFocusedStack(this)) {  
 Configuration config = mWindowManager.updateOrientationFromAppTokens(  
 mService.mConfiguration,  
 next.mayFreezeScreenLocked(next.app) ? next.appToken : null);  
 if (config != null) {  
 next.frozenBeforeDestroy = true;  
 }  
 notUpdated = !mService.updateConfigurationLocked(config, next, false);  
 }  
  
 if (notUpdated) {  
 // The configuration update wasn't able to keep the existing  
 // instance of the activity, and instead started a new one.  
 // We should be all done, but let's just make sure our activity  
 // is still at the top and schedule another run if something  
 // weird happened.  
 ActivityRecord nextNext = topRunningActivityLocked();  
 if (DEBUG\_SWITCH || DEBUG\_STATES) Slog.i(TAG\_STATES,  
 "Activity config changed during resume: " + next  
 + ", new next: " + nextNext);  
 if (nextNext != next) {  
 // Do over!  
 mStackSupervisor.scheduleResumeTopActivities();  
 }  
 if (mStackSupervisor.reportResumedActivityLocked(next)) {  
 mNoAnimActivities.clear();  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
 }  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return false;  
 }  
  
 try {  
 // Deliver all pending results.  
 ArrayList<ResultInfo> a = next.results;  
 if (a != null) {  
 final int N = a.size();  
 if (!next.finishing && N > 0) {  
 if (DEBUG\_RESULTS) Slog.v(TAG\_RESULTS,  
 "Delivering results to " + next + ": " + a);  
 next.app.thread.scheduleSendResult(next.appToken, a);  
 }  
 }  
  
 boolean allowSavedSurface = true;  
 if (next.newIntents != null) {  
 // Restrict saved surface to launcher start, or there is no intent at all  
 // (eg. task being brought to front). If the intent is something else,  
 // likely the app is going to show some specific page or view, instead of  
 // what's left last time.  
 for (int i = next.newIntents.size() - 1; i >= 0; i--) {  
 final Intent intent = next.newIntents.get(i);  
 if (intent != null && !ActivityRecord.isMainIntent(intent)) {  
 allowSavedSurface = false;  
 break;  
 }  
 }  
 next.app.thread.scheduleNewIntent(  
 next.newIntents, next.appToken, false /\* andPause \*/);  
 }  
  
 // Well the app will no longer be stopped.  
 // Clear app token stopped state in window manager if needed.  
 mWindowManager.notifyAppResumed(next.appToken, next.stopped, allowSavedSurface);  
  
 EventLog.writeEvent(EventLogTags.AM\_RESUME\_ACTIVITY, next.userId,  
 System.identityHashCode(next), next.task.taskId, next.shortComponentName);  
  
 next.sleeping = false;  
 mService.showUnsupportedZoomDialogIfNeededLocked(next);  
 mService.showAskCompatModeDialogLocked(next);  
 next.app.pendingUiClean = true;  
 next.app.forceProcessStateUpTo(mService.mTopProcessState);  
 next.clearOptionsLocked();  
 next.app.thread.scheduleResumeActivity(next.appToken, next.app.repProcState,  
 mService.isNextTransitionForward(), resumeAnimOptions);  
  
 mStackSupervisor.checkReadyForSleepLocked();  
  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES, "resumeTopActivityLocked: Resumed " + next);  
 } catch (Exception e) {  
 // Whoops, need to restart this activity!  
 if (DEBUG\_STATES) Slog.v(TAG\_STATES, "Resume failed; resetting state to "  
 + lastState + ": " + next);  
 next.state = lastState;  
 if (lastStack != null) {  
 lastStack.mResumedActivity = lastResumedActivity;  
 }  
 Slog.i(TAG, "Restarting because process died: " + next);  
 if (!next.hasBeenLaunched) {  
 next.hasBeenLaunched = true;  
 } else if (SHOW\_APP\_STARTING\_PREVIEW && lastStack != null &&  
 mStackSupervisor.isFrontStack(lastStack)) {  
 next.showStartingWindow(null, true);  
 }  
 mStackSupervisor.startSpecificActivityLocked(next, true, false);  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
 }  
  
 // From this point on, if something goes wrong there is no way  
 // to recover the activity.  
 try {  
 completeResumeLocked(next);  
 } catch (Exception e) {  
 // If any exception gets thrown, toss away this  
 // activity and try the next one.  
 Slog.w(TAG, "Exception thrown during resume of " + next, e);  
 requestFinishActivityLocked(next.appToken, Activity.RESULT\_CANCELED, null,  
 "resume-exception", true);  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
 }  
 } else {

//进入启动activity的流程  
 if (!next.hasBeenLaunched) {  
 next.hasBeenLaunched = true;  
 } else {  
 if (SHOW\_APP\_STARTING\_PREVIEW) {  
 next.showStartingWindow(null, true);  
 }  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH, "Restarting: " + next);  
 }  
 if (DEBUG\_STATES) Slog.d(TAG\_STATES, "resumeTopActivityLocked: Restarting " + next);

//正式启动activity，参考2.8  
 mStackSupervisor.startSpecificActivityLocked(next, true, true);  
 }  
  
 if (DEBUG\_STACK) mStackSupervisor.validateTopActivitiesLocked();  
 return true;  
}

### 2.8 startSpecificActivityLocked 启动activity或者启动进程

【ASS】

void startSpecificActivityLocked(ActivityRecord r,  
 boolean andResume, boolean checkConfig) {

// andResume与checkConfig均为true  
 // Is this activity's application already running?  
 ProcessRecord app = mService.getProcessRecordLocked(r.processName,  
 r.info.applicationInfo.uid, true);  
 //记录activity开始启动的时间，这个时间主要是用来记录activity启动时长的，当启动完成之后会计算出时长并记录到EventLogs里面，参考附录2，对应的log tag 为am\_activity\_launch\_time  
 r.task.stack.setLaunchTime(r);  
  
 if (app != null && app.thread != null) {  
 try {  
 if ((r.info.flags&ActivityInfo.FLAG\_MULTIPROCESS) == 0  
 || !"android".equals(r.info.packageName)) {  
 // Don't add this if it is a platform component that is marked  
 // to run in multiple processes, because this is actually  
 // part of the framework so doesn't make sense to track as a  
 // separate apk in the process.  
 app.addPackage(r.info.packageName, r.info.applicationInfo.versionCode,  
 mService.mProcessStats);  
 }

//启动activity  
 realStartActivityLocked(r, app, andResume, checkConfig);  
 return;  
 } catch (RemoteException e) {  
 Slog.w(TAG, "Exception when starting activity "  
 + r.intent.getComponent().flattenToShortString(), e);  
 }  
  
 // If a dead object exception was thrown -- fall through to  
 // restart the application.  
 }  
  
 //在进程为空的情况下，创建进程，参考2.9

mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,  
 "activity", r.intent.getComponent(), false, false, true);  
}

【】

final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app,  
 boolean andResume, boolean checkConfig) throws RemoteException {  
  
 if (!allPausedActivitiesComplete()) {  
 // While there are activities pausing we skipping starting any new activities until  
 // pauses are complete. NOTE: that we also do this for activities that are starting in  
 // the paused state because they will first be resumed then paused on the client side.  
 if (DEBUG\_SWITCH || DEBUG\_PAUSE || DEBUG\_STATES) Slog.v(TAG\_PAUSE,  
 "realStartActivityLocked: Skipping start of r=" + r  
 + " some activities pausing...");  
 return false;  
 }  
  
 if (andResume) {  
 r.startFreezingScreenLocked(app, 0);  
 mWindowManager.setAppVisibility(r.appToken, true);  
  
 // schedule launch ticks to collect information about slow apps.  
 r.startLaunchTickingLocked();  
 }  
  
 // Have the window manager re-evaluate the orientation of  
 // the screen based on the new activity order. Note that  
 // as a result of this, it can call back into the activity  
 // manager with a new orientation. We don't care about that,  
 // because the activity is not currently running so we are  
 // just restarting it anyway.  
 if (checkConfig) {  
 Configuration config = mWindowManager.updateOrientationFromAppTokens(  
 mService.mConfiguration,  
 r.mayFreezeScreenLocked(app) ? r.appToken : null);  
 // Deferring resume here because we're going to launch new activity shortly.  
 // We don't want to perform a redundant launch of the same record while ensuring  
 // configurations and trying to resume top activity of focused stack.  
 mService.updateConfigurationLocked(config, r, false, true /\* deferResume \*/);  
 }  
  
 r.app = app;  
 app.waitingToKill = null;  
 r.launchCount++;  
 r.lastLaunchTime = SystemClock.uptimeMillis();  
  
 if (DEBUG\_ALL) Slog.v(TAG, "Launching: " + r);  
  
 int idx = app.activities.indexOf(r);  
 if (idx < 0) {  
 app.activities.add(r);  
 }  
 mService.updateLruProcessLocked(app, true, null);  
 mService.updateOomAdjLocked();  
  
 final TaskRecord task = r.task;  
 if (task.mLockTaskAuth == LOCK\_TASK\_AUTH\_LAUNCHABLE ||  
 task.mLockTaskAuth == LOCK\_TASK\_AUTH\_LAUNCHABLE\_PRIV) {  
 setLockTaskModeLocked(task, LOCK\_TASK\_MODE\_LOCKED, "mLockTaskAuth==LAUNCHABLE", false);  
 }  
  
 final ActivityStack stack = task.stack;  
 try {  
 if (app.thread == null) {  
 throw new RemoteException();  
 }  
 List<ResultInfo> results = null;  
 List<ReferrerIntent> newIntents = null;  
 if (andResume) {  
 results = r.results;  
 newIntents = r.newIntents;  
 }  
 if (DEBUG\_SWITCH) Slog.v(TAG\_SWITCH,  
 "Launching: " + r + " icicle=" + r.icicle + " with results=" + results  
 + " newIntents=" + newIntents + " andResume=" + andResume);  
 if (andResume) {  
 EventLog.writeEvent(EventLogTags.AM\_RESTART\_ACTIVITY,  
 r.userId, System.identityHashCode(r),  
 task.taskId, r.shortComponentName);  
 }  
 if (r.isHomeActivity()) {  
 // Home process is the root process of the task.  
 mService.mHomeProcess = task.mActivities.get(0).app;  
 }  
 mService.notifyPackageUse(r.intent.getComponent().getPackageName(),  
 PackageManager.NOTIFY\_PACKAGE\_USE\_ACTIVITY);  
 r.sleeping = false;  
 r.forceNewConfig = false;  
 mService.showUnsupportedZoomDialogIfNeededLocked(r);  
 mService.showAskCompatModeDialogLocked(r);  
 r.compat = mService.compatibilityInfoForPackageLocked(r.info.applicationInfo);  
 ProfilerInfo profilerInfo = null;  
 if (mService.mProfileApp != null && mService.mProfileApp.equals(app.processName)) {  
 if (mService.mProfileProc == null || mService.mProfileProc == app) {  
 mService.mProfileProc = app;  
 final String profileFile = mService.mProfileFile;  
 if (profileFile != null) {  
 ParcelFileDescriptor profileFd = mService.mProfileFd;  
 if (profileFd != null) {  
 try {  
 profileFd = profileFd.dup();  
 } catch (IOException e) {  
 if (profileFd != null) {  
 try {  
 profileFd.close();  
 } catch (IOException o) {  
 }  
 profileFd = null;  
 }  
 }  
 }  
  
 profilerInfo = new ProfilerInfo(profileFile, profileFd,  
 mService.mSamplingInterval, mService.mAutoStopProfiler);  
 }  
 }  
 }  
  
 if (andResume) {  
 app.hasShownUi = true;  
 app.pendingUiClean = true;  
 }  
 app.forceProcessStateUpTo(mService.mTopProcessState);  
 app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,  
 System.identityHashCode(r), r.info, new Configuration(mService.mConfiguration),  
 new Configuration(task.mOverrideConfig), r.compat, r.launchedFromPackage,  
 task.voiceInteractor, app.repProcState, r.icicle, r.persistentState, results,  
 newIntents, !andResume, mService.isNextTransitionForward(), profilerInfo);  
  
 if ((app.info.privateFlags&ApplicationInfo.PRIVATE\_FLAG\_CANT\_SAVE\_STATE) != 0) {  
 // This may be a heavy-weight process! Note that the package  
 // manager will ensure that only activity can run in the main  
 // process of the .apk, which is the only thing that will be  
 // considered heavy-weight.  
 if (app.processName.equals(app.info.packageName)) {  
 if (mService.mHeavyWeightProcess != null  
 && mService.mHeavyWeightProcess != app) {  
 Slog.w(TAG, "Starting new heavy weight process " + app  
 + " when already running "  
 + mService.mHeavyWeightProcess);  
 }  
 mService.mHeavyWeightProcess = app;  
 Message msg = mService.mHandler.obtainMessage(  
 ActivityManagerService.POST\_HEAVY\_NOTIFICATION\_MSG);  
 msg.obj = r;  
 mService.mHandler.sendMessage(msg);  
 }  
 }  
  
 } catch (RemoteException e) {  
 if (r.launchFailed) {  
 // This is the second time we failed -- finish activity  
 // and give up.  
 Slog.e(TAG, "Second failure launching "  
 + r.intent.getComponent().flattenToShortString()  
 + ", giving up", e);  
 mService.appDiedLocked(app);  
 stack.requestFinishActivityLocked(r.appToken, Activity.RESULT\_CANCELED, null,  
 "2nd-crash", false);  
 return false;  
 }  
  
 // This is the first time we failed -- restart process and  
 // retry.  
 app.activities.remove(r);  
 throw e;  
 }  
  
 r.launchFailed = false;  
 if (stack.updateLRUListLocked(r)) {  
 Slog.w(TAG, "Activity " + r + " being launched, but already in LRU list");  
 }  
  
 if (andResume) {  
 // As part of the process of launching, ActivityThread also performs  
 // a resume.  
 stack.minimalResumeActivityLocked(r);  
 } else {  
 // This activity is not starting in the resumed state... which should look like we asked  
 // it to pause+stop (but remain visible), and it has done so and reported back the  
 // current icicle and other state.  
 if (DEBUG\_STATES) Slog.v(TAG\_STATES,  
 "Moving to PAUSED: " + r + " (starting in paused state)");  
 r.state = PAUSED;  
 }  
  
 // Launch the new version setup screen if needed. We do this -after-  
 // launching the initial activity (that is, home), so that it can have  
 // a chance to initialize itself while in the background, making the  
 // switch back to it faster and look better.  
 if (isFocusedStack(stack)) {  
 mService.startSetupActivityLocked();  
 }  
  
 // Update any services we are bound to that might care about whether  
 // their client may have activities.  
 if (r.app != null) {  
 mService.mServices.updateServiceConnectionActivitiesLocked(r.app);  
 }  
  
 return true;  
}

### 2.9

【AMS】

final ProcessRecord startProcessLocked(String processName,  
 ApplicationInfo info, boolean knownToBeDead, int intentFlags,  
 String hostingType, ComponentName hostingName, boolean allowWhileBooting,  
 boolean isolated, boolean keepIfLarge) {  
 return startProcessLocked(processName, info, knownToBeDead, intentFlags, hostingType,  
 hostingName, allowWhileBooting, isolated, 0 /\* isolatedUid \*/, keepIfLarge,  
 null /\* ABI override \*/, null /\* entryPoint \*/, null /\* entryPointArgs \*/,  
 null /\* crashHandler \*/);  
}  
参数解释：

processName 进程名，从activity的manifest配置而来，默认为应用配置的processname,如果都没有配置则是包名

info activity对应的ApplicationInfo，从manifest而来

knownToBeDead true

intentFlags 0

hostingType “activity”

hostingName 目标activity对应的intent设置的component，即activity的component

allowWhileBooting false

isolated false

keepIfLarge true  
final ProcessRecord startProcessLocked(String processName, ApplicationInfo info,  
 boolean knownToBeDead, int intentFlags, String hostingType, ComponentName hostingName,  
 boolean allowWhileBooting, boolean isolated, int isolatedUid, boolean keepIfLarge,  
 String abiOverride, String entryPoint, String[] entryPointArgs, Runnable crashHandler) {

//记录当前时间，这个是自系统boot之后开始计时的时间，包括睡眠状态。单位毫秒  
 long startTime = SystemClock.elapsedRealtime();  
 ProcessRecord app;

//参考2.9.1类后面描述isolated  
 if (!isolated) {

// ProcessRecord根据processName和uid标识一个process

//如果查询结果不为空，标识当前应用对应的process已经启动，需要注意的是一个进程里面可以包含多个应用，这里就是这个意思。  
 app = getProcessRecordLocked(processName, info.uid, keepIfLarge);  
 checkTime(startTime, "startProcess: after getProcessRecord");  
 // FLAG\_FROM\_BACKGROUND表示进程是由后台自动启动的，不是通过用户手动启动。  
 if ((intentFlags & Intent.FLAG\_FROM\_BACKGROUND) != 0) {  
 // If we are in the background, then check to see if this process  
 // is bad. If so, we will just silently fail.

//AppErrors类会维护一个mBadProcesses列表，每当app发生crash并达到一定次数的时候，就会把这个process加入列表。

//当启动进程发起方是后台进程并且目标process在mBadProcesses列表，则直接拒绝启动   
 if (mAppErrors.isBadProcessLocked(info)) {  
 if (DEBUG\_PROCESSES) Slog.v(TAG, "Bad process: " + info.uid  
 + "/" + info.processName);  
 return null;  
 }  
 } else {  
 // When the user is explicitly starting a process, then clear its  
 // crash count so that we won't make it bad until they see at  
 // least one crash dialog again, and make the process good again  
 // if it had been bad.

//这里表示是用户明确要启动，那么清除mBadProcesses中该process的记录  
 if (DEBUG\_PROCESSES) Slog.v(TAG, "Clearing bad process: " + info.uid  
 + "/" + info.processName);  
 mAppErrors.resetProcessCrashTimeLocked(info);  
 if (mAppErrors.isBadProcessLocked(info)) {  
 EventLog.writeEvent(EventLogTags.AM\_PROC\_GOOD,  
 UserHandle.getUserId(info.uid), info.uid,  
 info.processName);  
 mAppErrors.clearBadProcessLocked(info);  
 if (app != null) {  
 app.bad = false;  
 }  
 }  
 }  
 } else {  
 // If this is an isolated process, it can't re-use an existing process.  
 app = null;  
 }  
  
 // app launch boost for big.little configurations  
 // use cpusets to migrate freshly launched tasks to big cores

//不大懂，大概意思是把当前进程强制移动到某个空闲cpu执行？

//反正作用就是使用cpuset使得加快执行速度  
 nativeMigrateToBoost();  
 mIsBoosted = true;  
 mBoostStartTime = SystemClock.uptimeMillis();  
 Message msg = mHandler.obtainMessage(APP\_BOOST\_DEACTIVATE\_MSG);  
 mHandler.sendMessageDelayed(msg, APP\_BOOST\_MESSAGE\_DELAY);  
  
 // We don't have to do anything more if:  
 // (1) There is an existing application record; and  
 // (2) The caller doesn't think it is dead, OR there is no thread  
 // object attached to it so we know it couldn't have crashed; and  
 // (3) There is a pid assigned to it, so it is either starting or  
 // already running.  
 if (app != null && app.pid > 0) {  
 if ((!knownToBeDead && !app.killed) || app.thread == null) {  
 // We already have the app running, or are waiting for it to  
 // come up (we have a pid but not yet its thread), so keep it.  
 if (DEBUG\_PROCESSES) Slog.v(TAG\_PROCESSES, "App already running: " + app);  
 // If this is a new package in the process, add the package to the list  
 app.addPackage(info.packageName, info.versionCode, mProcessStats);  
 checkTime(startTime, "startProcess: done, added package to proc");  
 return app;  
 }  
  
 // An application record is attached to a previous process,  
 // clean it up now.  
 if (DEBUG\_PROCESSES || DEBUG\_CLEANUP) Slog.v(TAG\_PROCESSES, "App died: " + app);  
 checkTime(startTime, "startProcess: bad proc running, killing");  
 killProcessGroup(app.uid, app.pid);  
 handleAppDiedLocked(app, true, true);  
 checkTime(startTime, "startProcess: done killing old proc");  
 }  
  
 String hostingNameStr = hostingName != null  
 ? hostingName.flattenToShortString() : null;  
  
 if (app == null) {  
 checkTime(startTime, "startProcess: creating new process record");

//创建一个ProcessRecord  
 app = newProcessRecordLocked(info, processName, isolated, isolatedUid);  
 if (app == null) {  
 Slog.w(TAG, "Failed making new process record for "  
 + processName + "/" + info.uid + " isolated=" + isolated);  
 return null;  
 }  
 app.crashHandler = crashHandler;  
 checkTime(startTime, "startProcess: done creating new process record");  
 } else {  
 // If this is a new package in the process, add the package to the list  
 app.addPackage(info.packageName, info.versionCode, mProcessStats);  
 checkTime(startTime, "startProcess: added package to existing proc");  
 }  
  
 // 处理系统还没有ready的情况，添加到mProcessesOnHold中后退出，参照后面描述  
 if (!mProcessesReady  
 && !isAllowedWhileBooting(info)  
 && !allowWhileBooting) {  
 if (!mProcessesOnHold.contains(app)) {  
 mProcessesOnHold.add(app);  
 }  
 if (DEBUG\_PROCESSES) Slog.v(TAG\_PROCESSES,  
 "System not ready, putting on hold: " + app);  
 checkTime(startTime, "startProcess: returning with proc on hold");  
 return app;  
 }  
  
 checkTime(startTime, "startProcess: stepping in to startProcess");

//参考2.9.1  
 startProcessLocked(  
 app, hostingType, hostingNameStr, abiOverride, entryPoint, entryPointArgs);  
 checkTime(startTime, "startProcess: done starting proc!");  
 return (app.pid != 0) ? app : null;  
}

描述：

1. 可以留意一下这个方法里面有大量的checkTime方法，它的作用是检查上一次调用到这一次调用之间的时间跨度，如果超过50ms就会打印一句log，调试性能的时候可能会用上。
2. 系统启动ready之后会调用finishBooting，在这个方法里面会检查mProcessesOnHold，如果里面有待启动的进程任务，则会依次启动。

#### 2.9.1 startProcessLocked

【AMS】

private final void startProcessLocked(ProcessRecord app, String hostingType,  
 String hostingNameStr, String abiOverride, String entryPoint, String[] entryPointArgs) {  
 long startTime = SystemClock.elapsedRealtime();  
 if (app.pid > 0 && app.pid != MY\_PID) {  
 checkTime(startTime, "startProcess: removing from pids map");  
 synchronized (mPidsSelfLocked) {  
 mPidsSelfLocked.remove(app.pid);  
 mHandler.removeMessages(PROC\_START\_TIMEOUT\_MSG, app);  
 }  
 checkTime(startTime, "startProcess: done removing from pids map");  
 app.setPid(0);  
 }  
  
 //mProcessesOnHold记录了系统启动未完成时候就开始启动的进程，用于当系统boot完成之后再依次启动。能走到这里表示系统已经ready，所以移除mProcessesOnHold中的记录。  
 mProcessesOnHold.remove(app);  
  
 checkTime(startTime, "startProcess: starting to update cpu stats");  
 updateCpuStats();  
 checkTime(startTime, "startProcess: done updating cpu stats");  
  
 try {  
 try {

//获取userId，这里需要区分一下userId和app.uid，见后面介绍。  
 final int userId = UserHandle.getUserId(app.uid);  
 //检查应用是否可以被启动，如果不可启动则直接抛出异常，退出启动。

//满足以下条件的应用将不能被启动：

//1. 要启动的应用在已安装列表里面没有找到

//2. 对于目标用户，该应用未安装

//3. 安全模式下，若非系统应用（即设置了ApplicationInfo.FLAG\_SYSTEM），则不允许启动

//4. 已冻结的应用不允许启动

//5. 对于非Unlock的用户，如果应用不是系统启动是自动启动的，那么不允许启动

AppGlobals.getPackageManager().checkPackageStartable(app.info.packageName, userId);  
 } catch (RemoteException e) {  
 throw e.rethrowAsRuntimeException();  
 }  
 //进程uid，一般与info.uid相同，对应isolated的进程，则不想同  
 int uid = app.uid;  
 int[] gids = null;

//不需要mount外部存储  
 int mountExternal = Zygote.MOUNT\_EXTERNAL\_NONE;

//isolated 一般为false，参考类后面的描述  
 if (!app.isolated) {  
 int[] permGids = null;  
 try {  
 checkTime(startTime, "startProcess: getting gids from package manager");  
 final IPackageManager pm = AppGlobals.getPackageManager();

//计算当前用户所属的组ids  
 permGids = pm.getPackageGids(app.info.packageName,  
 MATCH\_DEBUG\_TRIAGED\_MISSING, app.userId);  
 MountServiceInternal mountServiceInternal = LocalServices.getService(  
 MountServiceInternal.class);

//计算当前用户访问外部存储的mount类型  
 mountExternal = mountServiceInternal.getExternalStorageMountMode(uid,  
 app.info.packageName);  
 } catch (RemoteException e) {  
 throw e.rethrowAsRuntimeException();  
 }  
  
 /\*  
 \* Add shared application and profile GIDs so applications can share some  
 \* resources like shared libraries and access user-wide resources  
 \*/

//至少需要添加两个gid，即shared application and profile GIDs  
 if (ArrayUtils.isEmpty(permGids)) {  
 gids = new int[2];  
 } else {  
 gids = new int[permGids.length + 2];  
 System.arraycopy(permGids, 0, gids, 2, permGids.length);  
 }  
 gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppId(uid));  
 gids[1] = UserHandle.getUserGid(UserHandle.getUserId(uid));  
 }  
 checkTime(startTime, "startProcess: building args");

//根据一些配置来打开Zygote的debugFlags的 JDWP和JNI check等等  
 int debugFlags = 0;  
 if ((app.info.flags & ApplicationInfo.FLAG\_DEBUGGABLE) != 0) {  
 debugFlags |= Zygote.DEBUG\_ENABLE\_DEBUGGER;  
 // Also turn on CheckJNI for debuggable apps. It's quite  
 // awkward to turn on otherwise.  
 debugFlags |= Zygote.DEBUG\_ENABLE\_CHECKJNI;  
 }  
 // Run the app in safe mode if its manifest requests so or the  
 // system is booted in safe mode.  
 if ((app.info.flags & ApplicationInfo.FLAG\_VM\_SAFE\_MODE) != 0 ||  
 mSafeMode == true) {  
 debugFlags |= Zygote.DEBUG\_ENABLE\_SAFEMODE;  
 }  
 if ("1".equals(SystemProperties.get("debug.checkjni"))) {  
 debugFlags |= Zygote.DEBUG\_ENABLE\_CHECKJNI;  
 }  
 String genDebugInfoProperty = SystemProperties.get("debug.generate-debug-info");  
 if ("true".equals(genDebugInfoProperty)) {  
 debugFlags |= Zygote.DEBUG\_GENERATE\_DEBUG\_INFO;  
 }  
 if ("1".equals(SystemProperties.get("debug.jni.logging"))) {  
 debugFlags |= Zygote.DEBUG\_ENABLE\_JNI\_LOGGING;  
 }  
 if ("1".equals(SystemProperties.get("debug.assert"))) {  
 debugFlags |= Zygote.DEBUG\_ENABLE\_ASSERT;  
 }  
 if (mNativeDebuggingApp != null && mNativeDebuggingApp.equals(app.processName)) {  
 // Enable all debug flags required by the native debugger.  
 debugFlags |= Zygote.DEBUG\_ALWAYS\_JIT; // Don't interpret anything  
 debugFlags |= Zygote.DEBUG\_GENERATE\_DEBUG\_INFO; // Generate debug info  
 debugFlags |= Zygote.DEBUG\_NATIVE\_DEBUGGABLE; // Disbale optimizations  
 mNativeDebuggingApp = null;  
 }  
 //配置运行的abi，primaryCpuAbi跟应用配置的native部分的abi有关，abiOverride则是应用manifest明确指定的abi模式，如果都为空，则使用系统支持的主abi，即Build.SUPPORTED\_ABIS[0]  
 String requiredAbi = (abiOverride != null) ? abiOverride : app.info.primaryCpuAbi;  
 if (requiredAbi == null) {  
 requiredAbi = Build.SUPPORTED\_ABIS[0];  
 }  
 //根据abi获取相应的指令集，这个参考文后描述介绍  
 String instructionSet = null;  
 if (app.info.primaryCpuAbi != null) {  
 instructionSet = VMRuntime.getInstructionSet(app.info.primaryCpuAbi);  
 }  
  
 app.gids = gids;  
 app.requiredAbi = requiredAbi;  
 app.instructionSet = instructionSet;  
  
 // Start the process. It will either succeed and return a result containing  
 // the PID of the new process, or else throw a RuntimeException.

//这个entryPoint比较重要，后续再分析，这里需要记住他的存在。  
 boolean isActivityProcess = (entryPoint == null);  
 if (entryPoint == null) entryPoint = "android.app.ActivityThread";  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "Start proc: " +  
 app.processName);  
 checkTime(startTime, "startProcess: asking zygote to start proc");

//调用Process.start, 参考2.9.2  
 Process.ProcessStartResult startResult = Process.start(entryPoint,  
 app.processName, uid, uid, gids, debugFlags, mountExternal,  
 app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,  
 app.info.dataDir, entryPointArgs);  
 checkTime(startTime, "startProcess: returned from zygote!");  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
 //通知进程启动  
 mBatteryStatsService.noteProcessStart(app.processName, app.info.uid);  
 checkTime(startTime, "startProcess: done updating battery stats");  
  
 EventLog.writeEvent(EventLogTags.AM\_PROC\_START,  
 UserHandle.getUserId(uid), startResult.pid, uid,  
 app.processName, hostingType,  
 hostingNameStr != null ? hostingNameStr : "");  
  
 try {  
 AppGlobals.getPackageManager().logAppProcessStartIfNeeded(app.processName, app.uid,  
 app.info.seinfo, app.info.sourceDir, startResult.pid);  
 } catch (RemoteException ex) {  
 // Ignore  
 }  
  
 if (app.persistent) {  
 Watchdog.getInstance().processStarted(app.processName, startResult.pid);  
 }  
  
 checkTime(startTime, "startProcess: building log message");  
 StringBuilder buf = mStringBuilder;  
 buf.setLength(0);  
 buf.append("Start proc ");  
 buf.append(startResult.pid);  
 buf.append(':');  
 buf.append(app.processName);  
 buf.append('/');  
 UserHandle.formatUid(buf, uid);  
 if (!isActivityProcess) {  
 buf.append(" [");  
 buf.append(entryPoint);  
 buf.append("]");  
 }  
 buf.append(" for ");  
 buf.append(hostingType);  
 if (hostingNameStr != null) {  
 buf.append(" ");  
 buf.append(hostingNameStr);  
 }  
 Slog.i(TAG, buf.toString());  
 app.setPid(startResult.pid);  
 app.usingWrapper = startResult.usingWrapper;  
 app.removed = false;  
 app.killed = false;  
 app.killedByAm = false;  
 checkTime(startTime, "startProcess: starting to update pids map");  
 ProcessRecord oldApp;  
 synchronized (mPidsSelfLocked) {  
 oldApp = mPidsSelfLocked.get(startResult.pid);  
 }  
 // If there is already an app occupying that pid that hasn't been cleaned up  
 if (oldApp != null && !app.isolated) {  
 // Clean up anything relating to this pid first  
 Slog.w(TAG, "Reusing pid " + startResult.pid  
 + " while app is still mapped to it");  
 cleanUpApplicationRecordLocked(oldApp, false, false, -1,  
 true /\*replacingPid\*/);  
 }  
 synchronized (mPidsSelfLocked) {  
 this.mPidsSelfLocked.put(startResult.pid, app);  
 if (isActivityProcess) {  
 Message msg = mHandler.obtainMessage(PROC\_START\_TIMEOUT\_MSG);  
 msg.obj = app;  
 mHandler.sendMessageDelayed(msg, startResult.usingWrapper  
 ? PROC\_START\_TIMEOUT\_WITH\_WRAPPER : PROC\_START\_TIMEOUT);  
 }  
 }  
 checkTime(startTime, "startProcess: done updating pids map");  
 } catch (RuntimeException e) {  
 Slog.e(TAG, "Failure starting process " + app.processName, e);  
  
 // Something went very wrong while trying to start this process; one  
 // common case is when the package is frozen due to an active  
 // upgrade. To recover, clean up any active bookkeeping related to  
 // starting this process. (We already invoked this method once when  
 // the package was initially frozen through KILL\_APPLICATION\_MSG, so  
 // it doesn't hurt to use it again.)  
 forceStopPackageLocked(app.info.packageName, UserHandle.getAppId(app.uid), false,  
 false, true, false, false, UserHandle.getUserId(app.userId), "start failure");  
 }  
}

描述：

1. userId和app.uid
   1. app.uid对应的是ApplicationInfo中的变量uid，这个变量代表了当前应用所属进程的uid，这个uid是在应用进程被启动的时候，PM解析package的时候，通过pm中的类Settings.java中的函数newUserIdLPw生成的。
   2. Process定义了应用程序所属的uid的范围是10000到19999，函数newUserIdLPw会在这个范围内选取最小的未被使用的数字分配给新启动的应用。
   3. 不同的应用可能通过配置运行在同一个进程里面，这样的情况下多个应用可能拥有相同的uid。
   4. userId对应的概念是多用户，如果系统支持多用户，那么UserHandle的方法会通过uid / PER\_USER\_RANGE返回代表当前uid所属的userId，其中PER\_USER\_RANGE为100000，根据多用户的定义，每个用户的uid的范围是100000个（不止包括app.uid,还有一些其他的）。当系统不支持多用户的时候userId默认为UserHandle.USER\_SYSTEM，即为0.
2. app.isolated
   1. 搜索整个源代码，发现isolated的赋值只有在ProcessRecord初始化的时候会判断ApplicationInfo的uid和ProcessRecord的uid是否相同，不同则isolated置为true。而ProcessRecord初始化只有在startProcessLocked的调用中通过方法newProcessRecordLocked是的isolated可能为真
   2. 而对于startProcessLocked的参数isolated的值只有在四大组件之一的service启动的时候通过判断r.serviceInfo.flags&ServiceInfo.FLAG\_ISOLATED\_PROCESS来决定isolated是否为true，而这个Flag正是对应的service在manifest中的标签android:isolatedProces
   3. 也就是说只有在service配置了android:isolatedProces的时候，isolated才会为true。而这个FLAG代表的意思是这个service启动之后将在一个新的独立进程里面运行，
   4. 官方解释: If set to true, this service will run under a special process that is isolated from the rest of the system. The only communication with it is through the Service API (binding and starting).
3. abi与指令集
   1. 在文件/libcore/libart/src/main/java/dalvik/system/VMRuntime.java中定义了abi与指令集的对应关系如下，可以使用方法VMRuntime.getInstructionSet通过abi获取指令集

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("armeabi", "arm");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("armeabi-v7a", "arm");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("mips", "mips");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("mips64", "mips64");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("x86", "x86");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("x86\_64", "x86\_64");

ABI\_TO\_INSTRUCTION\_SET\_MAP.put("arm64-v8a", "arm64");

* 1. 在文件./core/java/android/os/Build.java里面的变量SUPPORTED\_ABIS定义了系统支持的abi类型，这个类型通过访问ro.product.cpu.abilist属性获得。调试的时候可以通过adb getprop|grep abilist查看
  2. 文件build/target/board/xxx/BoardConfig.mk里面定义了TARGET\_CPU\_ABI := <abi-type>和TARGET\_CPU\_ABI2 := <abi-type>，然后在文件build/core/config.mk中把他们合并赋值给TARGET\_CPU\_ABI\_LIST\_32\_BIT或者TARGET\_CPU\_ABI\_LIST\_64\_BIT然后再把他们赋值给TARGET\_CPU\_ABI\_LIST，然后文件build/tools/buildinfo.sh中有ro.product.cpu.abilist\ =$TARGET\_CPU\_ABI\_LIST，于是系统编译之后这个东东就产生了。
  3. 当然，以上配置文件对于不同的厂商可能配置地方不尽相同。

#### 2.9.2

【Process.java】

Process.start会直接调用内部函数startViaZygote

*/\*\*  
 \* Starts a new process via the zygote mechanism.  
 \*  
 \** ***@param*** *processClass Class name whose static main() to run  
 \** ***@param*** *niceName 'nice' process name to appear in ps  
 \** ***@param*** *uid a POSIX uid that the new process should setuid() to  
 \** ***@param*** *gid a POSIX gid that the new process shuold setgid() to  
 \** ***@param*** *gids null-ok; a list of supplementary group IDs that the  
 \* new process should setgroup() to.  
 \** ***@param*** *debugFlags Additional flags.  
 \** ***@param*** *targetSdkVersion The target SDK version for the app.  
 \** ***@param*** *seInfo null-ok SELinux information for the new process.  
 \** ***@param*** *abi the ABI the process should use.  
 \** ***@param*** *instructionSet null-ok the instruction set to use.  
 \** ***@param*** *appDataDir null-ok the data directory of the app.  
 \** ***@param*** *extraArgs Additional arguments to supply to the zygote process.  
 \** ***@return*** *An object that describes the result of the attempt to start the process.  
 \** ***@throws*** *ZygoteStartFailedEx if process start failed for any reason  
 \*/*

private static ProcessStartResult startViaZygote(final String processClass,  
 final String niceName,  
 final int uid, final int gid,  
 final int[] gids,  
 int debugFlags, int mountExternal,  
 int targetSdkVersion,  
 String seInfo,  
 String abi,  
 String instructionSet,  
 String appDataDir,  
 String[] extraArgs)  
 throws ZygoteStartFailedEx {  
 synchronized(Process.class) {  
 ArrayList<String> argsForZygote = new ArrayList<String>();  
 //配置参数argsForZygote  
 // --runtime-args, --setuid=, --setgid=,  
 // and --setgroups= must go first  
 argsForZygote.add("--runtime-args");  
 argsForZygote.add("--setuid=" + uid);  
 argsForZygote.add("--setgid=" + gid);  
 。。。//这里有大量的配置  
  
 return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi), argsForZygote);  
 }  
}

## 3 ActivityThread 应用主进程的启动

### 3.1 类的初始化

【ActivityThread.java】

//ApplicationThread是ActivityThread的内部类，继承自ApplicationThreadNative，ApplicationThreadNative继承自Binder实现了IApplicationThread，ApplicationThread是该binder C/S中的服务端，另外位于ApplicationThreadNative.java中的类ApplicationThreadProxy是该binder的client端。ApplicationThread一般用来接收从ApplicationThreadProxy发来的调用，ApplicationThreadProxy运行在system\_server进程中。

final ApplicationThread mAppThread = new ApplicationThread();

//主线程looper，在main方法中初始化，其顺序问题参考后面对于static和final的知识点介绍

final Looper mLooper = Looper.myLooper();

//主方法Handler，用于处理主线程looper获取到的消息回调，handler的介绍参考文后知识点介绍  
final H mH = new H();

描述：严格来讲，main方法的执行应该是在ActivityThread对象初始化之前，之所以把对象初始化放到前面，是想先了解一下ActivityThread中包含的一些重要信息。

知识点:

1. static和final
   1. final和普通变量一样，在类对象初始化的时候进行初始化，也就是new关键字的时候进行，所以，final修饰的变量其实也是属于类对象，参考虚拟机相关知识中的<init>方法的描述。
   2. static属于类变量，意思是在类加载的时候就会进行初始化，而不必等等new关键字的执行，参考虚拟机相关知识中的<cinit>方法的描述
   3. 所以本例中main方法执行的时候，类对象变量(包括final修饰的变量)其实还没有初始化，只有main方法中执行到ActivityThread thread = new ActivityThread();的时候才开始初始化类对象。具体参考main方法的分析。
2. Handler用来实现线程间通信，其通信的根本原理是线程间共享数据，所以handler不能用于进程间通信。
   1. Handler, Message，MessageQueue，Looper
   2. Looper运行在主线程（或者创建Looper的线程）中，Looper会持有一个MessageQueue对象，Looper的loop方法会持续从MessageQueue中取出Message对象。
   3. Looper需要通过prepare的方式创建，这样做是为了保持一个线程只能持有一个Looper对象，prepare方法中会把新创建的Looper对象与当前线程组成一个关系体ThreadLocal并加入列表sThreadLocal中。需要的时候再从其中取出。
   4. Message对象包含了一些信息和一个Handler对象
   5. Handler对象在创建的时候会与当前thread的Looper对象绑定并持有Looper对象，如果当前thread没有Looper对象，则会抛出异常，所以在创建Handler对象之前必须通过Looper.prepare创建Looper对象
   6. Handler对象会通过Looper对象持有的MessageQueue对象间接持有MessageQueue对象，并通过这个对象把新生成的Message加入的MessageQueue中
   7. 通常Handler需要实现一个回调函数handleMessage来处理回调信息
   8. Message被取出来之后，MessageQueue会根据Message持有的Handler对象回调handleMessage。

### 3.2 main方法的执行

【ActivityThread.java】

public static void main(String[] args) {  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "ActivityThreadMain");

//这个的主要作用是生成一个进程相关信息的堆快照，并保存到目录/data/snapshots中，相关信息可以参考文件 ./libcore/dalvik/src/main/java/dalvik/system/profiler/SamplingProfiler.java  
 SamplingProfilerIntegration.start();  
  
 // CloseGuard defaults to true and can be quite spammy. We  
 // disable it here, but selectively enable it later (via  
 // StrictMode) on debug builds, but using DropBox, not logs.

//关闭CloseGuard，参考./libcore//dalvik/src/main/java/dalvik/system/CloseGuard.java  
 CloseGuard.setEnabled(false);  
 //配置用户信息，Environment类是用来访问环境变量的，可以访问文件系统中的各个文件目录以及外部存储目录等，有很多目录是跟用户id有关的，如果只有一个用户，那么userid对应的是0.  
 Environment.initForCurrentUser();  
  
 // Set the reporter for event logging in libcore  
 EventLogger.setReporter(new EventLoggingReporter());  
  
 // Make sure TrustedCertificateStore looks in the right place for CA certificates

//即文件目录/data/misc/user/<userId>/,只有一个用户的时候userId为0  
 final File configDir = Environment.getUserConfigDirectory(UserHandle.myUserId());  
 TrustedCertificateStore.setDefaultUserDirectory(configDir);  
 //设置进程参数，这个主要是用来显示进程的一些信息  
 Process.setArgV0("<pre-initialized>");  
 //构建主线程Looper，参考3.2.1  
 Looper.prepareMainLooper();  
 //创建对象，参考3.1  
 ActivityThread thread = new ActivityThread();

//主要步骤，参考3.3  
 thread.attach(false);  
  
 if (sMainThreadHandler == null) {

//即mH，参考3.1  
 sMainThreadHandler = thread.getHandler();  
 }  
  
 // End of event ActivityThreadMain.  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

//启动Looper的循环，不断的从MessageQueue中取出消息并处理  
 Looper.loop();  
  
 throw new RuntimeException("Main thread loop unexpectedly exited");  
}

描述： Zygote在fork出新的子进程之后，会调用子进程的第一个方法，也就是这里的main方法，使其运行到子进程中。这就意味着这个main方法是子进程中执行的第一个方法，是子进程的第一个线程，它通过looper的方法执行所有UI相关的动作，这也是它被称为主线程的原因。

#### 3.2.1

public static void prepareMainLooper() {  
 *prepare*(false);//见后面  
 synchronized (Looper.class) {  
 if (*sMainLooper* != null) {  
 throw new IllegalStateException("The main Looper has already been prepared.");  
 }

//通过*sThreadLocal获取当前thread对应的Looper对象*  
 *sMainLooper* = *myLooper*();  
 }  
}

*sThreadLocal是一个ThreadLocal<Looper>类型对象，ThreadLocal用来根据Thread信息维护一个Entry对象，而此处维护的Entry即为Looper对象，也就是说sThreadLocal里面维护者Thread和它的Looper对象并且是一一对应的，可以看出一个Thread只有一个Looper。*

private static void prepare(boolean quitAllowed) {  
 if (*sThreadLocal*.get() != null) {  
 throw new RuntimeException("Only one Looper may be created per thread");  
 }

// *sThreadLocal中没有Looper对象的时候，新建一个Looper对象并加入sThreadLocal中，注意主线程的*quitAllowed为false,表示不允许退出。参考3.2.1.1  
 *sThreadLocal*.set(new Looper(quitAllowed));  
}

##### 3.2.1.1 创建Looper和MessageQueue

private Looper(boolean quitAllowed) {

//Looper会持有一个MessageQueue对象  
 mQueue = new MessageQueue(quitAllowed);  
 mThread = Thread.*currentThread*();  
}

MessageQueue(boolean quitAllowed) {  
 mQuitAllowed = quitAllowed;

//通过native的方式创建MessageQueue，返回的是一个long型变量mPtr，其实是一个指针，但是只是供native使用，比如销毁队列、取出下一个消息等。  
 mPtr = *nativeInit*();  
}

### 3.3 进程attach

【ActivityThread.java】

private void attach(boolean system) {  
 sCurrentActivityThread = this;  
 mSystemThread = system;

//系统进程与普通进程的处理方式不同，这里只分析普通进程启动时的流程  
 if (!system) {

// ViewRootImpl 添加一个回调给ViewRootImpl，这个回调的目的是在ViewRootImpl第一次调用draw的时候执行Jit(Just In Time)编译, Jit 是JVM运行时编译器，负责把Java字节码编译成CPU平台相关的二进制码。关于Jit的作用请查看相关文档。  
 ViewRootImpl.addFirstDrawHandler(new Runnable() {  
 @Override  
 public void run() {  
 ensureJitEnabled();  
 }  
 });  
 android.ddm.DdmHandleAppName.setAppName("<pre-initialized>",  
 UserHandle.myUserId());

//RuntimeInit参考3.3.1  
 RuntimeInit.setApplicationObject(mAppThread.asBinder());  
 final IActivityManager mgr = ActivityManagerNative.getDefault();  
 try {

//调用AMS的attachApplication，参考3.4  
 mgr.attachApplication(mAppThread);  
 } catch (RemoteException ex) {  
 throw ex.rethrowFromSystemServer();  
 }  
 // Watch for getting close to heap limit.  
 BinderInternal.addGcWatcher(new Runnable() {  
 @Override public void run() {  
 if (!mSomeActivitiesChanged) {  
 return;  
 }  
 Runtime runtime = Runtime.getRuntime();  
 long dalvikMax = runtime.maxMemory();  
 long dalvikUsed = runtime.totalMemory() - runtime.freeMemory();  
 if (dalvikUsed > ((3\*dalvikMax)/4)) {  
 if (DEBUG\_MEMORY\_TRIM) Slog.d(TAG, "Dalvik max=" + (dalvikMax/1024)  
 + " total=" + (runtime.totalMemory()/1024)  
 + " used=" + (dalvikUsed/1024));  
 mSomeActivitiesChanged = false;  
 try {

//发生GC的时候，如果应用堆栈占用超过最大堆栈的3/4，则调用AMS接口释放部分activity。  
 mgr.releaseSomeActivities(mAppThread);  
 } catch (RemoteException e) {  
 throw e.rethrowFromSystemServer();  
 }  
 }  
 }  
 });  
 } else {

//系统进程  
 // Don't set application object here -- if the system crashes,  
 // we can't display an alert, we just want to die die die.  
 android.ddm.DdmHandleAppName.setAppName("system\_process",  
 UserHandle.myUserId());  
 try {  
 mInstrumentation = new Instrumentation();  
 ContextImpl context = ContextImpl.createAppContext(  
 this, getSystemContext().mPackageInfo);  
 mInitialApplication = context.mPackageInfo.makeApplication(true, null);  
 mInitialApplication.onCreate();  
 } catch (Exception e) {  
 throw new RuntimeException(  
 "Unable to instantiate Application():" + e.toString(), e);  
 }  
 }  
  
 // add dropbox logging to libcore  
 DropBox.setReporter(new DropBoxReporter());  
  
 ViewRootImpl.addConfigCallback(new ComponentCallbacks2() {  
 @Override  
 public void onConfigurationChanged(Configuration newConfig) {  
 synchronized (mResourcesManager) {  
 // We need to apply this change to the resources  
 // immediately, because upon returning the view  
 // hierarchy will be informed about it.  
 if (mResourcesManager.applyConfigurationToResourcesLocked(newConfig, null)) {  
 updateLocaleListFromAppContext(mInitialApplication.getApplicationContext(),  
 mResourcesManager.getConfiguration().getLocales());  
  
 // This actually changed the resources! Tell  
 // everyone about it.  
 if (mPendingConfiguration == null ||  
 mPendingConfiguration.isOtherSeqNewer(newConfig)) {  
 mPendingConfiguration = newConfig;  
  
 sendMessage(H.CONFIGURATION\_CHANGED, newConfig);  
 }  
 }  
 }  
 }  
 @Override  
 public void onLowMemory() {  
 }  
 @Override  
 public void onTrimMemory(int level) {  
 }  
 });  
}

#### 3.3.1 RuntimeInit

【RuntimeInit.java】

public static final void setApplicationObject(IBinder app) {  
 mApplicationObject = app;  
}

描述：mApplicationObject为IBinder类型，作用是当AMS处理进程crash的时候用来查找对应的进程，参考3.3.1.3.1。另外应用通过调用Log.wtf的时候也会调用到RuntimeInit的方法wtf，进一步调用到AMS的handleApplicationWtf时候也会用到该变量，作用与处理crash的时候相同。

##### 3.3.1.1 RuntimeInit 关于异常捕获部分的初始化

【RuntimeInit.java】

//Zygote在启动新的进程之后，在子进程里面会调用该方法，参考ZygoteConnection.java中的handleChildProc方法

public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)  
 throws ZygoteInit.MethodAndArgsCaller {  
 if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");  
  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "RuntimeInit");

//设置异常信息输出的优先级和Tag： Log.INFO对应"System.out"；Log.WARN对应"System.err"  
 redirectLogStreams();  
 //参考3.3.1.2  
 commonInit();

//以下部分在Zygote部分分析  
 nativeZygoteInit();  
 applicationInit(targetSdkVersion, argv, classLoader);  
}

##### 3.3.1.2 RuntimeInit 异常捕获初始化

private static final void commonInit() {  
 /\* set default handler; this applies to all threads in the VM \*/

//设置异常捕获Handler，参考3.3.1.3，当线程设置了UncaughtExceptionHandler，则线程发生未捕获的异常导致退出的时候会回调这个UncaughtExceptionHandler。注意，此处的Thread即为Zygote创建的子进程的第一个线程也就是应用主线程，所以这个异常handler就是用来捕获主线程异常信息的  
 Thread.setDefaultUncaughtExceptionHandler(new UncaughtHandler());  
  
 。。。//其它的信息在进程创建过程分析  
  
 initialized = true;  
}

##### 3.3.1.3 未捕获异常处理handler

private static class UncaughtHandler implements Thread.UncaughtExceptionHandler {  
 public void uncaughtException(Thread t, Throwable e) {  
 try {  
 // Don't re-enter -- avoid infinite loops if crash-reporting crashes.  
 if (mCrashing) return;  
 mCrashing = true;  
 // mApplicationObject对应的是前面传递进来的ApplicationThread对象  
 if (mApplicationObject == null) {  
 Clog\_e(TAG, "\*\*\* FATAL EXCEPTION IN SYSTEM PROCESS: " + t.getName(), e);  
 } else {  
 StringBuilder message = new StringBuilder();

//这个便是经常在log中看到的FATAL EXCEPTION  
 message.append("FATAL EXCEPTION: ").append(t.getName()).append("\n");  
 final String processName = ActivityThread.currentProcessName();  
 if (processName != null) {  
 message.append("Process: ").append(processName).append(", ");  
 }  
 message.append("PID: ").append(Process.myPid());

//打印log，adb logcat可以看到的log  
 Clog\_e(TAG, message.toString(), e);  
 }  
  
 // Try to end profiling. If a profiler is running at this point, and we kill the  
 // process (below), the in-memory buffer will be lost. So try to stop, which will  
 // flush the buffer. (This makes method trace profiling useful to debug crashes.)  
 if (ActivityThread.currentActivityThread() != null) {  
 ActivityThread.currentActivityThread().stopProfiling();  
 }  
  
 // Bring up crash dialog, wait for it to be dismissed

//应用crash对话框，这个会调用到AMS的handleApplicationCrash，参考3.3.1.3.1  
 ActivityManagerNative.getDefault().handleApplicationCrash(  
 mApplicationObject, new ApplicationErrorReport.CrashInfo(e));  
 } catch (Throwable t2) {  
 if (t2 instanceof DeadObjectException) {  
 // System process is dead; ignore  
 } else {  
 try {  
 Clog\_e(TAG, "Error reporting crash", t2);  
 } catch (Throwable t3) {  
 // Even Clog\_e() fails! Oh well.  
 }  
 }  
 } finally {  
 // Try everything to make sure this process goes away.

//异常信息处理完成之后，确保退出进程，如果有重启的动作，会在AMS里面直接执行  
 Process.killProcess(Process.myPid());  
 System.exit(10);  
 }  
 }  
}

###### 3.3.1.3.1 AMS处理应用crash

【AMS】

public void handleApplicationCrash(IBinder app, ApplicationErrorReport.CrashInfo crashInfo) {

// app的作用就是用来查找对应的进程信息  
 ProcessRecord r = findAppProcess(app, "Crash");  
 final String processName = app == null ? "system\_server"  
 : (r == null ? "unknown" : r.processName);  
  
 handleApplicationCrashInner("crash", r, processName, crashInfo);  
}  
  
/\* Native crash reporting uses this inner version because it needs to be somewhat  
 \* decoupled from the AM-managed cleanup lifecycle  
 \*/  
void handleApplicationCrashInner(String eventType, ProcessRecord r, String processName,  
 ApplicationErrorReport.CrashInfo crashInfo) {

//设置EventLog，参考附录二，对应的log信息为:am\_crash (User|1|5),(PID|1|5),(Process Name|3),(Flags|1|5),(Exception|3),(Message|3),(File|3),(Line|1|5)  
 EventLog.writeEvent(EventLogTags.AM\_CRASH, Binder.getCallingPid(),  
 UserHandle.getUserId(Binder.getCallingUid()), processName,  
 r == null ? -1 : r.info.flags,  
 crashInfo.exceptionClassName,  
 crashInfo.exceptionMessage,  
 crashInfo.throwFileName,  
 crashInfo.throwLineNumber);  
 //添加crash信息到DropBox中，dropbox中的信息可以通过命令dumpsys dropbox (-f/-p)获得，这里只是crash，其实WTF和ANR也是通过这个方法添加到dropbox中的，后续遇到了再分析  
 addErrorToDropBox(eventType, r, processName, null, null, null, null, null, crashInfo);  
 //处理crash信息，显示crash对话框，处理重启等策略  
 mAppErrors.crashApplication(r, crashInfo);  
}

##### 3.1.3.4 wtf异常处理

[RuntimeInit.java]

//应用程序通过主动调用Log.wtf来记录wtf类型的信息

public static void wtf(String tag, Throwable t, boolean system) {  
 try {

//调用AMS方法处理WTF异常，见3.1.3.4.1  
 if (ActivityManagerNative.getDefault().handleApplicationWtf(  
 mApplicationObject, tag, system, new ApplicationErrorReport.CrashInfo(t))) {  
 // The Activity Manager has already written us off -- now exit.  
 Process.killProcess(Process.myPid());  
 System.exit(10);  
 }  
 } catch (Throwable t2) {  
 if (t2 instanceof DeadObjectException) {  
 // System process is dead; ignore  
 } else {  
 Slog.e(TAG, "Error reporting WTF", t2);  
 Slog.e(TAG, "Original WTF:", t);  
 }  
 }  
}

###### 3.1.3.4.1

【AMS】

public boolean handleApplicationWtf(final IBinder app, final String tag, boolean system,  
 final ApplicationErrorReport.CrashInfo crashInfo) {  
 final int callingUid = Binder.getCallingUid();  
 final int callingPid = Binder.getCallingPid();  
  
 if (system) {  
 mHandler.post(new Runnable() {  
 @Override public void run() {

//如果是system，异步处理wtf异常，完事了直接返回，并通知进程无需kill掉自己  
 handleApplicationWtfInner(callingUid, callingPid, app, tag, crashInfo);  
 }  
 });  
 return false;  
 }  
 //普通进程，直接处理wtf异常  
 final ProcessRecord r = handleApplicationWtfInner(callingUid, callingPid, app, tag,  
 crashInfo);  
  
 if (r != null && r.pid != Process.myPid() &&

//检查系统设置“wtf\_is\_fatal”，如果不为0，则构建crash 提示框，同app crash的处理。  
 Settings.Global.getInt(mContext.getContentResolver(),  
 Settings.Global.WTF\_IS\_FATAL, 0) != 0) {  
 mAppErrors.crashApplication(r, crashInfo);  
 return true;  
 } else {  
 return false;  
 }  
}  
//处理wtf，主要有两件工作，记录EventsLog和写DropBox  
ProcessRecord handleApplicationWtfInner(int callingUid, int callingPid, IBinder app, String tag,  
 final ApplicationErrorReport.CrashInfo crashInfo) {  
 final ProcessRecord r = findAppProcess(app, "WTF");  
 final String processName = app == null ? "system\_server"  
 : (r == null ? "unknown" : r.processName);  
 //记录EventsLog  
 EventLog.writeEvent(EventLogTags.AM\_WTF, UserHandle.getUserId(callingUid), callingPid,  
 processName, r == null ? -1 : r.info.flags, tag, crashInfo.exceptionMessage);  
 //把wtf信息添加到dropbox里面  
 addErrorToDropBox("wtf", r, processName, null, null, tag, null, null, crashInfo);  
  
 return r;  
}

### 3.4 AMS进程attach

【AMS】

public final void attachApplication(IApplicationThread thread) {  
 synchronized (this) {  
 int callingPid = Binder.getCallingPid();  
 final long origId = Binder.clearCallingIdentity();  
 attachApplicationLocked(thread, callingPid);  
 Binder.restoreCallingIdentity(origId);  
 }  
}

private final boolean attachApplicationLocked(IApplicationThread thread,  
 int pid) {  
  
 // Find the application record that is being attached... either via  
 // the pid if we are running in multiple processes, or just pull the  
 // next app record if we are emulating process with anonymous threads.  
 ProcessRecord app;  
 if (pid != MY\_PID && pid >= 0) {  
 synchronized (mPidsSelfLocked) {  
 app = mPidsSelfLocked.get(pid);  
 }  
 } else {  
 app = null;  
 }  
  
 if (app == null) {  
 Slog.w(TAG, "No pending application record for pid " + pid  
 + " (IApplicationThread " + thread + "); dropping process");  
 EventLog.writeEvent(EventLogTags.AM\_DROP\_PROCESS, pid);  
 if (pid > 0 && pid != MY\_PID) {

//进程对应的ProcessRecord不存在，则以静默的方式杀死相应进程，不会记录任何信息包括log。  
 Process.killProcessQuiet(pid);  
 //TODO: killProcessGroup(app.info.uid, pid);  
 } else {  
 try {

//这个会通过binder调用到ActivityThread，执行Looper.myLooper().quit();结束主线程looper  
 thread.scheduleExit();  
 } catch (Exception e) {  
 // Ignore exceptions.  
 }  
 }  
 return false;  
 }  
  
 // thread变量表示当前进程绑定的IApplicationThread对象，如果不为空，表示当前进程已经有绑定IApplicationThread。当重新绑定的时候需要解绑以前的。  
 if (app.thread != null) {

//处理进程死亡，参考3.4.1  
 handleAppDiedLocked(app, true, true);  
 }  
  
 final String processName = app.processName;  
 try {

//进程死亡监听，AppDeathRecipient实现了接口DeathRecipient，当目标进程对应的IBinder死亡的时候，Binder会回调DeathRecipient的回调函数binderDied。进而会调用AMS的appDiedLocked方法。这个一般是binder的server端监听client状态用。参考3.4.2  
 AppDeathRecipient adr = new AppDeathRecipient(  
 app, pid, thread);  
 thread.asBinder().linkToDeath(adr, 0);  
 app.deathRecipient = adr;  
 } catch (RemoteException e) {  
 app.resetPackageList(mProcessStats);  
 startProcessLocked(app, "link fail", processName);  
 return false;  
 }  
  
 EventLog.writeEvent(EventLogTags.AM\_PROC\_BOUND, app.userId, app.pid, app.processName);  
 //把当前进程置位active状态，对应ProcessState中的变量mActive  
 app.makeActive(thread, mProcessStats);

//初始化相关的adj值和schedule值，相关定义都在文件ProcessList中  
 app.curAdj = app.setAdj = app.verifiedAdj = ProcessList.INVALID\_ADJ;  
 app.curSchedGroup = app.setSchedGroup = ProcessList.SCHED\_GROUP\_DEFAULT;  
 app.forcingToForeground = null;

//AMS维护一个名为mForegroundPackages的ProcessMap<ArrayList<ProcessRecord>>,这个map以uid和ArrayList<ProcessRecord>为元组来记录处于前台的uid对应的进程列表，注意一个uid可以对应多个进程，比如除了主进程还有后台服务或者前台服务等。而这个变量记录进程列表主要是用来通知电源服务来记录耗电情况的。这里只是把该进程信息从变量mForegroundPackages列表中移除，因为进程刚创建，所以旧的记录应当失效。第二个参数表示新进程是否在前台，目前为false，第三个参数表示是否需要进行进程策略adj调整，关于adj的知识参考附录三  
 updateProcessForegroundLocked(app, false, false);  
 app.hasShownUi = false;  
 app.debugging = false;  
 app.cached = false;  
 app.killedByAm = false;  
  
 // We carefully use the same state that PackageManager uses for  
 // filtering, since we use this flag to decide if we need to install  
 // providers when user is unlocked later  
 app.unlocked = StorageManager.isUserKeyUnlocked(app.userId);  
 //异常进程启动timeout消息，表示启动成功未超时，在前面分析进程启动的时候，在方法startProcessLocked中配置了启动超时的消息PROC\_START\_TIMEOUT\_MSG，当时对于普通进程配置的超时时长为10s，也就是说在启动一个应用（进程不存在的情况）的时候发生ANR的时间为10s  
 mHandler.removeMessages(PROC\_START\_TIMEOUT\_MSG, app);  
  
 boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);  
 List<ProviderInfo> providers = normalMode ? generateApplicationProvidersLocked(app) : null;  
  
 if (providers != null && checkAppInLaunchingProvidersLocked(app)) {  
 Message msg = mHandler.obtainMessage(CONTENT\_PROVIDER\_PUBLISH\_TIMEOUT\_MSG);  
 msg.obj = app;  
 mHandler.sendMessageDelayed(msg, CONTENT\_PROVIDER\_PUBLISH\_TIMEOUT);  
 }  
  
 if (!normalMode) {  
 Slog.i(TAG, "Launching preboot mode app: " + app);  
 }  
  
 if (DEBUG\_ALL) Slog.v(  
 TAG, "New app record " + app  
 + " thread=" + thread.asBinder() + " pid=" + pid);  
 try {  
 int testMode = IApplicationThread.DEBUG\_OFF;  
 if (mDebugApp != null && mDebugApp.equals(processName)) {  
 testMode = mWaitForDebugger  
 ? IApplicationThread.DEBUG\_WAIT  
 : IApplicationThread.DEBUG\_ON;  
 app.debugging = true;  
 if (mDebugTransient) {  
 mDebugApp = mOrigDebugApp;  
 mWaitForDebugger = mOrigWaitForDebugger;  
 }  
 }  
 String profileFile = app.instrumentationProfileFile;  
 ParcelFileDescriptor profileFd = null;  
 int samplingInterval = 0;  
 boolean profileAutoStop = false;  
 if (mProfileApp != null && mProfileApp.equals(processName)) {  
 mProfileProc = app;  
 profileFile = mProfileFile;  
 profileFd = mProfileFd;  
 samplingInterval = mSamplingInterval;  
 profileAutoStop = mAutoStopProfiler;  
 }  
 boolean enableTrackAllocation = false;  
 if (mTrackAllocationApp != null && mTrackAllocationApp.equals(processName)) {  
 enableTrackAllocation = true;  
 mTrackAllocationApp = null;  
 }  
  
 // If the app is being launched for restore or full backup, set it up specially  
 boolean isRestrictedBackupMode = false;  
 if (mBackupTarget != null && mBackupAppName.equals(processName)) {  
 isRestrictedBackupMode = mBackupTarget.appInfo.uid >= Process.FIRST\_APPLICATION\_UID  
 && ((mBackupTarget.backupMode == BackupRecord.RESTORE)  
 || (mBackupTarget.backupMode == BackupRecord.RESTORE\_FULL)  
 || (mBackupTarget.backupMode == BackupRecord.BACKUP\_FULL));  
 }  
  
 if (app.instrumentationClass != null) {  
 notifyPackageUse(app.instrumentationClass.getPackageName(),  
 PackageManager.NOTIFY\_PACKAGE\_USE\_INSTRUMENTATION);  
 }  
 if (DEBUG\_CONFIGURATION) Slog.v(TAG\_CONFIGURATION, "Binding proc "  
 + processName + " with config " + mConfiguration);  
 ApplicationInfo appInfo = app.instrumentationInfo != null  
 ? app.instrumentationInfo : app.info;  
 app.compat = compatibilityInfoForPackageLocked(appInfo);  
 if (profileFd != null) {  
 profileFd = profileFd.dup();  
 }  
 ProfilerInfo profilerInfo = profileFile == null ? null  
 : new ProfilerInfo(profileFile, profileFd, samplingInterval, profileAutoStop);

//参考3.5  
 thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,  
 profilerInfo, app.instrumentationArguments, app.instrumentationWatcher,  
 app.instrumentationUiAutomationConnection, testMode,  
 mBinderTransactionTrackingEnabled, enableTrackAllocation,  
 isRestrictedBackupMode || !normalMode, app.persistent,  
 new Configuration(mConfiguration), app.compat,  
 getCommonServicesLocked(app.isolated),  
 mCoreSettingsObserver.getCoreSettingsLocked());  
 updateLruProcessLocked(app, false, null);  
 app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();  
 } catch (Exception e) {  
 // todo: Yikes! What should we do? For now we will try to  
 // start another process, but that could easily get us in  
 // an infinite loop of restarting processes...  
 Slog.wtf(TAG, "Exception thrown during bind of " + app, e);  
  
 app.resetPackageList(mProcessStats);  
 app.unlinkDeathRecipient();  
 startProcessLocked(app, "bind fail", processName);  
 return false;  
 }  
  
 // Remove this record from the list of starting applications.  
 mPersistentStartingProcesses.remove(app);  
 if (DEBUG\_PROCESSES && mProcessesOnHold.contains(app)) Slog.v(TAG\_PROCESSES,  
 "Attach application locked removing on hold: " + app);  
 mProcessesOnHold.remove(app);  
  
 boolean badApp = false;  
 boolean didSomething = false;  
  
 // See if the top visible activity is waiting to run in this process...  
 if (normalMode) {  
 try {  
 if (mStackSupervisor.attachApplicationLocked(app)) {  
 didSomething = true;  
 }  
 } catch (Exception e) {  
 Slog.wtf(TAG, "Exception thrown launching activities in " + app, e);  
 badApp = true;  
 }  
 }  
  
 // Find any services that should be running in this process...  
 if (!badApp) {  
 try {  
 didSomething |= mServices.attachApplicationLocked(app, processName);  
 } catch (Exception e) {  
 Slog.wtf(TAG, "Exception thrown starting services in " + app, e);  
 badApp = true;  
 }  
 }  
  
 // Check if a next-broadcast receiver is in this process...  
 if (!badApp && isPendingBroadcastProcessLocked(pid)) {  
 try {  
 didSomething |= sendPendingBroadcastsLocked(app);  
 } catch (Exception e) {  
 // If the app died trying to launch the receiver we declare it 'bad'  
 Slog.wtf(TAG, "Exception thrown dispatching broadcasts in " + app, e);  
 badApp = true;  
 }  
 }  
  
 // Check whether the next backup agent is in this process...  
 if (!badApp && mBackupTarget != null && mBackupTarget.appInfo.uid == app.uid) {  
 if (DEBUG\_BACKUP) Slog.v(TAG\_BACKUP,  
 "New app is backup target, launching agent for " + app);  
 notifyPackageUse(mBackupTarget.appInfo.packageName,  
 PackageManager.NOTIFY\_PACKAGE\_USE\_BACKUP);  
 try {  
 thread.scheduleCreateBackupAgent(mBackupTarget.appInfo,  
 compatibilityInfoForPackageLocked(mBackupTarget.appInfo),  
 mBackupTarget.backupMode);  
 } catch (Exception e) {  
 Slog.wtf(TAG, "Exception thrown creating backup agent in " + app, e);  
 badApp = true;  
 }  
 }  
  
 if (badApp) {  
 app.kill("error during init", true);  
 handleAppDiedLocked(app, false, true);  
 return false;  
 }  
  
 if (!didSomething) {  
 updateOomAdjLocked();  
 }  
  
 return true;  
}

3.4.2 进程死亡处理流程

【AMS】

final void appDiedLocked(ProcessRecord app, int pid, IApplicationThread thread,  
 boolean fromBinderDied) {  
 // First check if this ProcessRecord is actually active for the pid.  
 synchronized (mPidsSelfLocked) {  
 ProcessRecord curProc = mPidsSelfLocked.get(pid);

//传入的进程id对应的进程与传进来的进程不相符，则认为是假死，直接返回，不做处理  
 if (curProc != app) {  
 Slog.w(TAG, "Spurious death for " + app + ", curProc for " + pid + ": " + curProc);  
 return;  
 }  
 }

//通知移除电源状态对该进程的监控  
 BatteryStatsImpl stats = mBatteryStatsService.getActiveStatistics();  
 synchronized (stats) {  
 stats.noteProcessDiedLocked(app.info.uid, pid);  
 }  
  
 if (!app.killed) {

//如果不是因为binder的死亡导致的进程死亡，则以静默的方式杀掉进程。  
 if (!fromBinderDied) {  
 Process.killProcessQuiet(pid);  
 }

//通过native方式杀掉由当前pid对应进程启动的进程组中所有进程，然后标注该进程已被杀。  
 killProcessGroup(app.uid, pid);  
 app.killed = true;  
 }  
  
 // Clean up already done if the process has been re-started.  
 if (app.pid == pid && app.thread != null &&  
 app.thread.asBinder() == thread.asBinder()) {  
 boolean doLowMem = app.instrumentationClass == null;  
 boolean doOomAdj = doLowMem;  
 if (!app.killedByAm) {  
 Slog.i(TAG, "Process " + app.processName + " (pid " + pid  
 + ") has died");  
 mAllowLowerMemLevel = true;  
 } else {  
 // Note that we always want to do oom adj to update our state with the  
 // new number of procs.  
 mAllowLowerMemLevel = false;  
 doLowMem = false;  
 }  
 EventLog.writeEvent(EventLogTags.AM\_PROC\_DIED, app.userId, app.pid, app.processName);  
 if (DEBUG\_CLEANUP) Slog.v(TAG\_CLEANUP,  
 "Dying app: " + app + ", pid: " + pid + ", thread: " + thread.asBinder());

//处理进程被杀之后的收尾工作  
 handleAppDiedLocked(app, false, true);  
  
 if (doOomAdj) {

//更新进程OomAdj值，参考附录三  
 updateOomAdjLocked();  
 }  
 if (doLowMem) {

//报告Low Mem  
 doLowMemReportIfNeededLocked(app);  
 }  
 } else if (app.pid != pid) {  
 // A new process has already been started.  
 Slog.i(TAG, "Process " + app.processName + " (pid " + pid  
 + ") has died and restarted (pid " + app.pid + ").");  
 EventLog.writeEvent(EventLogTags.AM\_PROC\_DIED, app.userId, app.pid, app.processName);  
 } else if (DEBUG\_PROCESSES) {  
 Slog.d(TAG\_PROCESSES, "Received spurious death notification for thread "  
 + thread.asBinder());  
 }  
}

### 3.5 handleBindApplication 处理app 绑定流程

【】

private void handleBindApplication(AppBindData data) {  
 // Register the UI Thread as a sensitive thread to the runtime.

//敏感线程，JIT 即时编译会用到  
 VMRuntime.registerSensitiveThread();

// trackAllocation这个变量在启动进程的第一个activity的时候，ASS中通过调用resolveActivity解析startFlags来进行配置，如果配置了，那么此处调用DdmVmInternal使能内存分配跟踪，用于跟踪内存分配情况。AndroidStudio的monitor工具的Allocation tracking类似。DdmVmInternal位于libcore目录。  
 if (data.trackAllocation) {  
 DdmVmInternal.enableRecentAllocations(true);  
 }  
  
 // Note when this process has started.  
 Process.setStartTimes(SystemClock.elapsedRealtime(), SystemClock.uptimeMillis());  
  
 mBoundApplication = data;  
 mConfiguration = new Configuration(data.config);  
 mCompatConfiguration = new Configuration(data.config);  
 //进程的profile相关信息，profile的配置与trackAllocation类似，也是在ASS中进行。  
 mProfiler = new Profiler();  
 if (data.initProfilerInfo != null) {  
 mProfiler.profileFile = data.initProfilerInfo.profileFile;  
 mProfiler.profileFd = data.initProfilerInfo.profileFd;  
 mProfiler.samplingInterval = data.initProfilerInfo.samplingInterval;  
 mProfiler.autoStopProfiler = data.initProfilerInfo.autoStopProfiler;  
 }  
  
 // send up app name; do this \*before\* waiting for debugger  
 Process.setArgV0(data.processName);  
 android.ddm.DdmHandleAppName.setAppName(data.processName,  
 UserHandle.myUserId());

//对于persistent类型的进程，在低内存设备是关闭硬件加速，因为persistent的进程不属于lowmemorykiller自动kill进程的范畴，而硬件加速往往会占用较多内存。  
 if (data.persistent) {  
 // Persistent processes on low-memory devices do not get to  
 // use hardware accelerated drawing, since this can add too much  
 // overhead to the process.

//如果配置了系统属性ro.config.low\_ram为true或者在res/res/values/config.xml文件里面配置了config\_avoidGfxAccel为true，则表明当前设备是低内存的。  
 if (!ActivityManager.isHighEndGfx()) {  
 ThreadedRenderer.disable(false);  
 }  
 }  
  
 if (mProfiler.profileFd != null) {

//如果支持profile收集，该方法会调用VMDebug，去启动profile收集工作，并把收集到的信息写入profileFile指定的文件中。  
 mProfiler.startProfiling();  
 }  
  
 // If the app is Honeycomb MR1 or earlier, switch its AsyncTask  
 // implementation to use the pool executor. Normally, we use the  
 // serialized executor as the default. This has to happen in the  
 // main thread so the main looper is set right.  
 if (data.appInfo.targetSdkVersion <= android.os.Build.VERSION\_CODES.HONEYCOMB\_MR1) {  
 AsyncTask.setDefaultExecutor(AsyncTask.THREAD\_POOL\_EXECUTOR);  
 }  
  
 Message.updateCheckRecycle(data.appInfo.targetSdkVersion);  
  
 /\*  
 \* Before spawning a new process, reset the time zone to be the system time zone.  
 \* This needs to be done because the system time zone could have changed after the  
 \* the spawning of this process. Without doing this this process would have the incorrect  
 \* system time zone.  
 \*/  
 TimeZone.setDefault(null);  
  
 /\*  
 \* Set the LocaleList. This may change once we create the App Context.  
 \*/  
 LocaleList.setDefault(data.config.getLocales());  
  
 synchronized (mResourcesManager) {  
 /\*  
 \* Update the system configuration since its preloaded and might not  
 \* reflect configuration changes. The configuration object passed  
 \* in AppBindData can be safely assumed to be up to date  
 \*/  
 mResourcesManager.applyConfigurationToResourcesLocked(data.config, data.compatInfo);  
 mCurDefaultDisplayDpi = data.config.densityDpi;  
  
 // This calls mResourcesManager so keep it within the synchronized block.  
 applyCompatConfiguration(mCurDefaultDisplayDpi);  
 }  
  
 data.info = getPackageInfoNoCheck(data.appInfo, data.compatInfo);  
  
 */\*\*  
 \* Switch this process to density compatibility mode if needed.  
 \*/* if ((data.appInfo.flags&ApplicationInfo.FLAG\_SUPPORTS\_SCREEN\_DENSITIES)  
 == 0) {  
 mDensityCompatMode = true;  
 Bitmap.setDefaultDensity(DisplayMetrics.DENSITY\_DEFAULT);  
 }  
 updateDefaultDensity();  
  
 final boolean is24Hr = "24".equals(mCoreSettings.getString(Settings.System.TIME\_12\_24));  
 DateFormat.set24HourTimePref(is24Hr);  
  
 View.mDebugViewAttributes =  
 mCoreSettings.getInt(Settings.Global.DEBUG\_VIEW\_ATTRIBUTES, 0) != 0;  
  
 */\*\*  
 \* For system applications on userdebug/eng builds, log stack  
 \* traces of disk and network access to dropbox for analysis.  
 \*/* if ((data.appInfo.flags &  
 (ApplicationInfo.FLAG\_SYSTEM |  
 ApplicationInfo.FLAG\_UPDATED\_SYSTEM\_APP)) != 0) {  
 StrictMode.conditionallyEnableDebugLogging();  
 }  
  
 */\*\*  
 \* For apps targetting Honeycomb or later, we don't allow network usage  
 \* on the main event loop / UI thread. This is what ultimately throws  
 \* {****@link*** *NetworkOnMainThreadException}.  
 \*/* if (data.appInfo.targetSdkVersion >= Build.VERSION\_CODES.HONEYCOMB) {  
 StrictMode.enableDeathOnNetwork();  
 }  
  
 */\*\*  
 \* For apps targetting N or later, we don't allow file:// Uri exposure.  
 \* This is what ultimately throws {****@link*** *FileUriExposedException}.  
 \*/* if (data.appInfo.targetSdkVersion >= Build.VERSION\_CODES.N) {  
 StrictMode.enableDeathOnFileUriExposure();  
 }  
  
 NetworkSecurityPolicy.getInstance().setCleartextTrafficPermitted(  
 (data.appInfo.flags & ApplicationInfo.FLAG\_USES\_CLEARTEXT\_TRAFFIC) != 0);  
  
 if (data.debugMode != IApplicationThread.DEBUG\_OFF) {  
 // XXX should have option to change the port.  
 Debug.changeDebugPort(8100);  
 if (data.debugMode == IApplicationThread.DEBUG\_WAIT) {  
 Slog.w(TAG, "Application " + data.info.getPackageName()  
 + " is waiting for the debugger on port 8100...");  
  
 IActivityManager mgr = ActivityManagerNative.getDefault();  
 try {  
 mgr.showWaitingForDebugger(mAppThread, true);  
 } catch (RemoteException ex) {  
 throw ex.rethrowFromSystemServer();  
 }  
  
 Debug.waitForDebugger();  
  
 try {  
 mgr.showWaitingForDebugger(mAppThread, false);  
 } catch (RemoteException ex) {  
 throw ex.rethrowFromSystemServer();  
 }  
  
 } else {  
 Slog.w(TAG, "Application " + data.info.getPackageName()  
 + " can be debugged on port 8100...");  
 }  
 }  
  
 // Allow application-generated systrace messages if we're debuggable.  
 boolean isAppDebuggable = (data.appInfo.flags & ApplicationInfo.FLAG\_DEBUGGABLE) != 0;  
 Trace.setAppTracingAllowed(isAppDebuggable);  
 if (isAppDebuggable && data.enableBinderTracking) {  
 Binder.enableTracing();  
 }  
  
 */\*\*  
 \* Initialize the default http proxy in this process for the reasons we set the time zone.  
 \*/* Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "Setup proxies");  
 final IBinder b = ServiceManager.getService(Context.CONNECTIVITY\_SERVICE);  
 if (b != null) {  
 // In pre-boot mode (doing initial launch to collect password), not  
 // all system is up. This includes the connectivity service, so don't  
 // crash if we can't get it.  
 final IConnectivityManager service = IConnectivityManager.Stub.asInterface(b);  
 try {  
 final ProxyInfo proxyInfo = service.getProxyForNetwork(null);  
 Proxy.setHttpProxySystemProperty(proxyInfo);  
 } catch (RemoteException e) {  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
 throw e.rethrowFromSystemServer();  
 }  
 }  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
  
 // Instrumentation info affects the class loader, so load it before  
 // setting up the app context.  
 final InstrumentationInfo ii;  
 if (data.instrumentationName != null) {  
 try {  
 ii = new ApplicationPackageManager(null, getPackageManager())  
 .getInstrumentationInfo(data.instrumentationName, 0);  
 } catch (PackageManager.NameNotFoundException e) {  
 throw new RuntimeException(  
 "Unable to find instrumentation info for: " + data.instrumentationName);  
 }  
  
 mInstrumentationPackageName = ii.packageName;  
 mInstrumentationAppDir = ii.sourceDir;  
 mInstrumentationSplitAppDirs = ii.splitSourceDirs;  
 mInstrumentationLibDir = getInstrumentationLibrary(data.appInfo, ii);  
 mInstrumentedAppDir = data.info.getAppDir();  
 mInstrumentedSplitAppDirs = data.info.getSplitAppDirs();  
 mInstrumentedLibDir = data.info.getLibDir();  
 } else {  
 ii = null;  
 }  
  
 final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);  
 updateLocaleListFromAppContext(appContext,  
 mResourcesManager.getConfiguration().getLocales());  
  
 if (!Process.isIsolated() && !"android".equals(appContext.getPackageName())) {  
 // This cache location probably points at credential-encrypted  
 // storage which may not be accessible yet; assign it anyway instead  
 // of pointing at device-encrypted storage.  
 final File cacheDir = appContext.getCacheDir();  
 if (cacheDir != null) {  
 // Provide a usable directory for temporary files  
 System.setProperty("java.io.tmpdir", cacheDir.getAbsolutePath());  
 } else {  
 Log.v(TAG, "Unable to initialize \"java.io.tmpdir\" property "  
 + "due to missing cache directory");  
 }  
  
 // Setup a location to store generated/compiled graphics code.  
 final Context deviceContext = appContext.createDeviceProtectedStorageContext();  
 final File codeCacheDir = deviceContext.getCodeCacheDir();  
 if (codeCacheDir != null) {  
 setupGraphicsSupport(data.info, codeCacheDir);  
 } else {  
 Log.e(TAG, "Unable to setupGraphicsSupport due to missing code-cache directory");  
 }  
 }  
  
 // Install the Network Security Config Provider. This must happen before the application  
 // code is loaded to prevent issues with instances of TLS objects being created before  
 // the provider is installed.  
 Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "NetworkSecurityConfigProvider.install");  
 NetworkSecurityConfigProvider.install(appContext);  
 Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);  
  
 // Continue loading instrumentation.  
 if (ii != null) {  
 final ApplicationInfo instrApp = new ApplicationInfo();  
 ii.copyTo(instrApp);  
 instrApp.initForUser(UserHandle.myUserId());  
 final LoadedApk pi = getPackageInfo(instrApp, data.compatInfo,  
 appContext.getClassLoader(), false, true, false);  
 final ContextImpl instrContext = ContextImpl.createAppContext(this, pi);  
  
 try {  
 final ClassLoader cl = instrContext.getClassLoader();  
 mInstrumentation = (Instrumentation)  
 cl.loadClass(data.instrumentationName.getClassName()).newInstance();  
 } catch (Exception e) {  
 throw new RuntimeException(  
 "Unable to instantiate instrumentation "  
 + data.instrumentationName + ": " + e.toString(), e);  
 }  
  
 final ComponentName component = new ComponentName(ii.packageName, ii.name);  
 mInstrumentation.init(this, instrContext, appContext, component,  
 data.instrumentationWatcher, data.instrumentationUiAutomationConnection);  
  
 if (mProfiler.profileFile != null && !ii.handleProfiling  
 && mProfiler.profileFd == null) {  
 mProfiler.handlingProfiling = true;  
 final File file = new File(mProfiler.profileFile);  
 file.getParentFile().mkdirs();  
 Debug.startMethodTracing(file.toString(), 8 \* 1024 \* 1024);  
 }  
 } else {  
 mInstrumentation = new Instrumentation();  
 }  
  
 if ((data.appInfo.flags&ApplicationInfo.FLAG\_LARGE\_HEAP) != 0) {  
 dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();  
 } else {  
 // Small heap, clamp to the current growth limit and let the heap release  
 // pages after the growth limit to the non growth limit capacity. b/18387825  
 dalvik.system.VMRuntime.getRuntime().clampGrowthLimit();  
 }  
  
 // Allow disk access during application and provider setup. This could  
 // block processing ordered broadcasts, but later processing would  
 // probably end up doing the same disk access.  
 final StrictMode.ThreadPolicy savedPolicy = StrictMode.allowThreadDiskWrites();  
 try {  
 // If the app is being launched for full backup or restore, bring it up in  
 // a restricted environment with the base application class.  
 Application app = data.info.makeApplication(data.restrictedBackupMode, null);  
 mInitialApplication = app;  
  
 // don't bring up providers in restricted mode; they may depend on the  
 // app's custom Application class  
 if (!data.restrictedBackupMode) {  
 if (!ArrayUtils.isEmpty(data.providers)) {  
 installContentProviders(app, data.providers);  
 // For process that contains content providers, we want to  
 // ensure that the JIT is enabled "at some point".  
 mH.sendEmptyMessageDelayed(H.ENABLE\_JIT, 10\*1000);  
 }  
 }  
  
 // Do this after providers, since instrumentation tests generally start their  
 // test thread at this point, and we don't want that racing.  
 try {  
 mInstrumentation.onCreate(data.instrumentationArgs);  
 }  
 catch (Exception e) {  
 throw new RuntimeException(  
 "Exception thrown in onCreate() of "  
 + data.instrumentationName + ": " + e.toString(), e);  
 }  
  
 try {  
 mInstrumentation.callApplicationOnCreate(app);  
 } catch (Exception e) {  
 if (!mInstrumentation.onException(app, e)) {  
 throw new RuntimeException(  
 "Unable to create application " + app.getClass().getName()  
 + ": " + e.toString(), e);  
 }  
 }  
 } finally {  
 StrictMode.setThreadPolicy(savedPolicy);  
 }  
}

# 附录

## 一 名词解释

ADJ adjustment 进程调整策略，用于OOM的时候对于进程的调整

AS ActivityStack 存储activity的栈

ASS ActivityStackSupervisor 管理ActivityStack

ActivityStarter 管理activity启动

AMS ActivityManagerService 整体activity调度

LMK Low Memory Killer

lmkd Low Memory Killer Daemon，常驻进程，lmkd.c

OOM out of memory

PIP Picture In Picture Activity

WTF What a Terrible Failure， Log类用来记录重要异常的，调用Log.wtf可能导致应用退出也可能不会退出，依赖于系统设置。

\*#\*#1000# WM相关

## 二 EventLogs

1. EventLogs是记录系统事件的一个log策略，相关事件都定义在\*.logcats类型的文件中，存放在不同的目录，比如activity生命周期相关的events都定义在文件

./services/core/java/com/android/server/am/EventLogTags.logtags

之中，比较重要的几个logtag文件列举如下：

./packages/SystemUI/src/com/android/systemui/EventLogTags.logtags 这个与SystemUI相关，比如sysui\_statusbar\_touch，sysui\_lockscreen\_gesture等

./services/core/java/com/android/server/am/EventLogTags.logtags am相关

./services/core/java/com/android/server/EventLogTags.logtags 其它系统相关，比如power，notification，battery，wm等

2. 使用命令logcat –b events可以查看系统events log，如下

I/am\_destroy\_activity( 878): [0,966143252,4993,com.UCMobile/.main.UCMobile,finish-idle]

am\_destroy\_activity对应于文件

./services/core/java/com/android/server/am/EventLogTags.logtags

之中的如下定义：

# An activity is being destroyed:

30018 am\_destroy\_activity (User|1|5),(Token|1|5),(Task ID|1|5),(Component Name|3),(Reason|3)

查询该文件可以具体了解log中的每一项代表的意义。

3. 关于\*.logcats类型的文件的描述和定义，参考文件

./system//core/logcat/event.logtags

中的定义，该类型文件由logcat进程进行解析的。

三: AMS进程管理

1. ProcessList.java定义了OOM 调整（adjustment）进程的类别，如下表：

|  |  |  |
| --- | --- | --- |
| 变量名称 | adj值 | 描述 |
| UNKNOWN\_ADJ | 1001 | 未知调整策略值 |
| CACHED\_APP\_MAX\_ADJ\* | 906 | 只拥有activity并且不可见的进程最大adj，对于普通应用切换到后台之后，都会经过PREVIOUS\_APP\_ADJ然后进入CACHED状态，并且越早值越大，也就是优先级越低，越容易被kill，当后台Cached进程大于7个的时候，会有个调整策略，即3（测试所得）个一组置位相同的adj，顺序还是按切换先后来的。 |
| CACHED\_APP\_MIN\_ADJ\* | 900 | 只拥有activity并且不可见的进程最小adj |
| SERVICE\_B\_ADJ | 800 | B类service，属于用户不太感兴趣的服务 |
| PREVIOUS\_APP\_ADJ | 700 | 当前应用的前一个应用所在进程，即通过back键退出当前应用后会直接切换到的进程 |
| HOME\_APP\_ADJ | 600 | HOME 进程，一般要避免被kill掉，虽然属于后台不可见的应用，但是用户会经常与之交互，所以尽可能避免被kill |
| SERVICE\_ADJ | 500 | 后台服务进程，就用户关注度来说，这类进程就显得没有那么重要了，但总的来说还是比home更重要一些，毕竟home影响的只是重新加载的速度问题。 |
| HEAVY\_WEIGHT\_APP\_ADJ | 400 | 高权重进程 |
| BACKUP\_APP\_ADJ\* | 300 | 后台备份进程，算是后台进程中的最重要的进程，作为adj阈值的一个级别，它可以表示当需要kill它的时候其它后台进程都已经被kill。作为后台进程，被杀了用户一般感知不到，但是kill备份进程往往不是一个好的选择，所以它的级别紧邻着可感知应用进程 |
| PERCEPTIBLE\_APP\_ADJ\* | 200 | 用户可感知的应用进程，比如后台音乐播放器，后台下载进程等 |
| VISIBLE\_APP\_ADJ\* | 100 | 当前可见（没有获得焦点）的应用所在进程 |
| FOREGROUND\_APP\_ADJ\* | 0 | 当前前台应用对应的进程 |
| PERSISTENT\_SERVICE\_ADJ | -700 | 被PERSISTENT\_PROC\_ADJ和SYSTEM\_ADJ绑定的服务进程，属于重要进程 |
| PERSISTENT\_PROC\_ADJ | -800 | 系统永久进程，比如telephony，明确指定不想被kill |
| SYSTEM\_ADJ | -900 | 系统进程 |
| NATIVE\_ADJ | -1000 | native进程，不由system进程管理，所以不属于system进程调整管理范畴 |

根据表中的定义，进程级别从上到下重要性越来越高。进程的kill工作主要是由lmkd执行的，而lmkd只支持6个级别的进程定义，分别对应上表中带\*部分。

1. AM关于进程状态定义了如下18中状态：

|  |  |  |
| --- | --- | --- |
| PROCESS\_STATE\_NONEXISTENT | -1 | 进程不存在 |
| PROCESS\_STATE\_PERSISTENT | 0 | 永久进程 |
| PROCESS\_STATE\_PERSISTENT\_UI | 1 | 永久进程并且正在执行UI动作 |
| PROCESS\_STATE\_TOP | 2 | 拥有可视activity的进程，包括拥有焦点和没有焦点的 |
| PROCESS\_STATE\_BOUND\_FOREGROUND\_SERVICE | 3 | 拥有绑定到系统进程的service |
| PROCESS\_STATE\_FOREGROUND\_SERVICE | 4 | 拥有绑定到前台进程的service |
| PROCESS\_STATE\_TOP\_SLEEPING | 5 | 同PROCESS\_STATE\_TOP，只是目前处于锁屏状态 |
| PROCESS\_STATE\_IMPORTANT\_FOREGROUND | 6 | 对用户重要，并且用户可以感知到 |
| PROCESS\_STATE\_IMPORTANT\_BACKGROUND | 7 | 对用户重要，但是用户感知不到 |
| PROCESS\_STATE\_BACKUP | 8 | 正在后台执行备份或恢复 |
| PROCESS\_STATE\_HEAVY\_WEIGHT | 9 | 处于后台，但是进程状态无法恢复，所以一般会避免被kill |
| PROCESS\_STATE\_SERVICE | 10 | 后台服务 |
| PROCESS\_STATE\_RECEIVER | 11 | 后台receiver |
| PROCESS\_STATE\_HOME | 12 | Home |
| PROCESS\_STATE\_LAST\_ACTIVITY | 13 | Last shown activity所在进程 |
| PROCESS\_STATE\_CACHED\_ACTIVITY | 14 | 缓存的以备后用的包含activity的进程 |
| PROCESS\_STATE\_CACHED\_ACTIVITY\_CLIENT | 15 | 缓存以备后用的进程，属于某个缓存Activity进程的client |
| PROCESS\_STATE\_CACHED\_EMPTY | 16 | 缓存以备后用的空进程 |

1. AMS通过如下三个方法管理进程的adj策略
   * updateOomAdjLocked：更新adj，当目标进程为空，或者被杀则返回false；否则返回true;
   * computeOomAdjLocked：计算adj，返回计算后RawAdj值;
   * applyOomAdjLocked：应用adj，当需要杀掉目标进程则返回false；否则返回true。
   * 如下各种情况下会触发updateOomAdjLocked来更新进程adj状态
     1. Activity
        1. ASS.realStartActivityLocked: 启动Activity
        2. AS.resumeTopActivityInnerLocked: 恢复栈顶Activity
        3. AS.finishCurrentActivityLocked: 结束当前Activity
        4. AS.destroyActivityLocked: 销毁当前Activity
     2. Service位于ActiveServices.java
        1. realStartServiceLocked: 启动服务
        2. bindServiceLocked: 绑定服务(只更新当前app)
        3. unbindServiceLocked: 解绑服务 (只更新当前app)
        4. bringDownServiceLocked: 结束服务 (只更新当前app)
        5. sendServiceArgsLocked: 在bringup或则cleanup服务过程调用 (只更新当前app)
     3. broadcast
        1. BQ.processNextBroadcast: 处理下一个广播
        2. BQ.processCurBroadcastLocked: 处理当前广播
        3. BQ.deliverToRegisteredReceiverLocked: 分发已注册的广播 (只更新当前app)
     4. ContentProvider
        1. AMS.removeContentProvider: 移除provider
        2. AMS.publishContentProviders: 发布provider (只更新当前app)
        3. AMS.getContentProviderImpl: 获取provider (只更新当前app)
     5. Process位于ActivityManagerService.java
        1. setSystemProcess: 创建并设置系统进程
        2. addAppLocked: 创建persistent进程
        3. attachApplicationLocked: 进程创建后attach到system\_server的过程;
        4. trimApplications: 清除没有使用app
        5. appDiedLocked: 进程死亡
        6. killAllBackgroundProcesses: 杀死所有后台进程.即(ADJ>9或removed=true的普通进程)
        7. killPackageProcessesLocked: 以包名的形式 杀掉相关进程;
   * 在方法updateOomAdjLocked里面会分别调用computeOomAdjLocked和applyOomAdjLocked来计算和设置adj
2. ProcessList定义了三种与lmkd通信的命令，这些命令与lmkd的定义相匹配，如下:

|  |  |  |
| --- | --- | --- |
| **功能** | **命令** | **对应方法** |
| LMK\_PROCPRIO | 设置进程adj | PL.setOomAdj() |
| LMK\_TARGET | 更新oom\_adj | PL.updateOomLevels() |
| LMK\_PROCREMOVE | 移除进程 | PL.remove() |

* 当AMS.applyOomAdjLocked()过程,则会设置某个进程的adj;
* 当AMS.updateConfiguration()过程中便会更新整个各个级别的oom\_adj信息.
* 当AMS.cleanUpApplicationRecordLocked()或者handleAppDiedLocked()过程,则会将某个进程从lmkd策略中移除.

1. drivers/staging/android/lowmemorykiller.c
   1. lowmemorykiller.c维护两个数组lowmem\_adj和lowmem\_minfree, 这两个数组的内容可以到系统文件/sys/module/lowmemorykiller/parameters/adj和/sys/module/lowmemorykiller/parameters/minfree中查看，内容格式如下

adj：0,100,200,300,900,906

minfree： 18432,23040,27648,32256,36864,46080

分别代表进程adj阈值和当前内存剩余量阈值。

* 1. lowmemorykiller.c在加载到内核的时候会注册一个shrinker为lowmem\_shrinker，当内存不足时就会回调该shrinker定义的函数lowmem\_shrink, 该函数检查当前系统内存剩余量，查看空闲内存在minfree几个级别中的哪一档，比如剩余内存大于等于32256小于36864。那么，接着检查所有进程的adj值，在所有adj大于等于300的进程中找出adj值最大的进程，如果最大的有多个进程，则计算找出占用内存最大的进程，把这个进程设置为selected状态。然后通知kernel发送SIGKILL信号给该进程，从而kill掉该进程。注意一次执行只会选出一个进程去kill。如果没有adj大于等于300的进程则不执行kill动作，直到内存减少到下一个minfree档位，即大于等于27648小于32256
  2. 关于adj和minfree则是在文件ProcessList.java中定义的，通过方法updateOomLevels把数据写入到lmkd的socket中，对应的socket名字就是“lmkd”。
  3. lmkd定义在目录system/core/lmkd/中，lmkd.c通过epoll的方式监听socket lmkd，获取到消息之后通过ctrl\_command\_handler分发给前面第4节提到的三种命令，而设置adj的命令是LMK\_TARGET, 分发给函数cmd\_target，该函数把adj和minfree写入到相应文件中。
  4. 设置进程adj的时候，ProcessList调用setOomAdj发送命令LMK\_PROCPRIO给lmkd，lmkd收到之后会分发给方法cmd\_procprio,该方法会把传递进来的adj写入进程文件/proc/<pid>/oom\_score\_adj中，这样lowmemorykiller在检查到内存不足的时候就可以查看到每个进程的adj值，并决定是否符合kill条件。

1. 综述：ProcessList定义adj值，以及adj和minfree阈值，AM定义Process state值，AMS根据各个进程的动态来调整设置进程的state和adj值，lmkd接收ProcessList设置adj的命令设置各个进程的adj到相应的文件，kernel定期检查系统内存状况，通知lowmemorykiller，lowmemorykiller比较minfree阈值与当前内存状况，需要kill进程的时候检查各个进程的adj值，选择合适的进程执行kill动作