[无涯教程网](https://www.learnfk.com/)

* [[](http://share.learnfk.com/f/18605620-495896203-fd1f6c)《Android》PDF下载](http://share.learnfk.com/f/18605620-495896203-fd1f6c)
* [[](https://www.learnfk.com/pdf/download.html)PDF电子书集合](https://www.learnfk.com/pdf/download.html" \o "PDF电子书集合)

Android AMS核心分析教程

[首页](https://www.learnfk.com/)/ [Android](https://www.learnfk.com/android/android-tutorial.html)/ [Android AMS核心分析教程](https://www.learnfk.com/android/android-tutorial-ams.html)

AMS是系统的引导服务，应用进程的启动、切换、调度以及四大组件的启动和管理都需要AMS的支持。

一、AMS概述

AMS的启动

1. SystemServer的main方法中实例化SystemServer，并且调用它的run方法。
2. run方法中会加载动态库以及启动系统服务，这里就包括AMS的启动。
   * 通过SystemServiceManager的start方法启动服务，这个方法接受一个SystemService类型的class，通过反射调用SystemService的构造方法来初始化。
   * Lifecycle是一个位于ActivityManagerService中的静态内部类，继承了SystemService，它内部构造方法中实例化了ActivityManagerService，并将其启动。

AMS 与 Activity 的启动

1. startActivity会调用Instrumentation的execStartActivity方法
2. execStartActivity方法中通过ActivityManagerNative的getDefault方法获取AMS的代理类ActivityManagerProxy，然后通过代理的startActivity来向AMS发起一次启动Activity的IPC。
   * ActivityManagerNative是AMS的父类，它的getDefault方法中会通过ServiceManager获得AMS的Binder，并将这个Binder封装成ActivityManagerProxy作为代理。
   * ActivityManagerProxy的startActivity其实是通过被代理的Binder向AMS发起请求，以START\_ACTIVITY\_TRANSACTION作为请求code。
3. AMS接收到客户端的请求后在onTransact方法中判断请求code是START\_ACTIVITY\_TRANSACTION则调用自身的startActivity。
4. AMS的startActivity调用自身的startActivityAsUser将启动交给StackSupervisor来处理。
5. StackSupervisor中经过多个方法的调用最终会在realStartActivityLocked方法中通过ApplicationThreadProxy代理向客户端发起一次请求code为SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION的IPC，即启动Activity的过程交给了客户端。
6. 客户端会在ApplicationThreadNative中接收请求，并在onTransact方法中判断code如果是SCHEDULE\_LAUNCH\_ACTIVITY\_TRANSACTION，则调用ApplicationThread的scheduleLaunchActivity来启动Activity。

后续内容详见[ActivityThread](https://www.learnfk.com/android/android-tutorial-activitythread.html)

二、AMS的启动

AMS的是在SystemServer进程中启动的,SystemServer的main方法中实例化了一个SystemServer，并调用了他的run方法。

// SystemServer

public static void main(String[] args) {

new SystemServer().run();

}

// SystemServer

private void run() {

// ...

// 加载动态库

System.loadLibrary("android\_servers");

// ...

// Initialize the system context.

createSystemContext();

// 创建SystemServiceManager

mSystemServiceManager = new SystemServiceManager(mSystemContext);

LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);

// Start services.

try {

// 启动引导服务,使用SystemServiceManager启动AMS、PMS、PKMS等

startBootstrapServices();

// 启动核心服务，包括BatteryService、UsageStatsService、WebViewUpdateService等

startCoreServices();

// 启动其他服务，包括CameraService、AlarmManagerService、VrManagerService等

startOtherServices();

}

// ...

// Loop forever.

Looper.loop();

throw new RuntimeException("Main thread loop unexpectedly exited");

}

重点看startBootstrapServices中启动AMS的过程

// SystemServer

private void startBootstrapServices() {

Installer installer = mSystemServiceManager.startService(Installer.class);

// Activity manager runs the show.

mActivityManagerService = mSystemServiceManager.startService(

ActivityManagerService.Lifecycle.class).getService();

mActivityManagerService.setSystemServiceManager(mSystemServiceManager);

mActivityManagerService.setInstaller(installer);

// ...

}

通过SystemServiceManager的startService方法启动获取AMS，这个方法传进去一个ActivityManagerService.Lifecycle类型的class，Lifecycle是AMS中的静态内部类，继承了SystemService。其源码如下：

public static final class Lifecycle extends SystemService {

private final ActivityManagerService mService;

public Lifecycle(Context context) {

super(context);

mService = new ActivityManagerService(context);

}

@Override

public void onStart() {

mService.start();

}

public ActivityManagerService getService() {

return mService;

}

}

可以看到，在Lifecycle的构造方法中实例化了ActivityManagerService，getService方法会返回AMS的实例，并且在onStart方法中启动AMS服务。

接着看startService接收了Lifecycle之后，做了哪些处理：

// SystemServiceManager

private final ArrayList<SystemService> mServices = new ArrayList<SystemService>();

public <T extends SystemService> T startService(Class<T> serviceClass) {

final String name = serviceClass.getName();

// ...

final T service;

try {

// 反射获取Lifecycle的构造方法

Constructor<T> constructor = serviceClass.getConstructor(Context.class);

// 实例化Lifecycle，此时AMS会在Lifecycle构造方法中被实例化

service = constructor.newInstance(mContext);

}

// ...

// 将Lifecycle添加到List集合中进行注册

mServices.add(service);

// Start it.

try {

// 调用Lifecycle的onStart方法，启动AMS服务

service.onStart();

}

return service;

}

最后启动服务后，startService方法会将这个Lifecycle实例作为返回值返回。再来看SystemServer的startBootServices方法中获取AMS的代码就一目了然了：

mActivityManagerService = mSystemServiceManager.startService( ActivityManagerService.Lifecycle.class).getService(); SystemServiceManager的startService用来启动多种服务，这里仅仅是拿AMS的启动来分析。

三、AMS与Activity启动

ActivityManager是一个和AMS相关连的类，它主要对运行中的Activity进行管理，这些管理工作并不是真正由ActivityManager来处理的，而是交由AMS来处理。ActivityManager中的方法会通过ActivityManagerNative的getDefault方法来得到ActivityManagerProxy，通过ActivityManagerProxy可以和ActivityManagerNative通信。而ActivityManagerNative是一个抽象类，AMS继承了ActivityManagerNative，这些功能是在AMS中实现的。因此，ActivityManagerProxy是AMS的代理类。

以Activity的启动流程为例来分析。Activity的启动过程中会调用Instrumentation的execStartActivity方法，代码如下：

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

public ActivityResult execStartActivity(

Context who, IBinder contextThread, IBinder token, Activity target,

Intent intent, int requestCode, Bundle options) {

// ...

try {

intent.migrateExtraStreamToClipData();

intent.prepareToLeaveProcess();

int result = ActivityManagerNative.getDefault()

.startActivity(whoThread, who.getBasePackageName(), intent,

intent.resolveTypeIfNeeded(who.getContentResolver()),

token, target != null ? target.mEmbeddedID : null,

requestCode, 0, null, options);

checkStartActivityResult(result, intent);

} catch (RemoteException e) {

throw new RuntimeException("Failure from system", e);

}

return null;

}

在execStartActivity方法中会调用ActivityManagerNative的getDefault方法来获取AMS的代理类ActivityManagerProxy实例，并调用它的startActivity方法。

getDefault的源码如下：

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

static public IActivityManager getDefault() {

return gDefault.get();

}

private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {

protected IActivityManager create() {

// 从ServiceManager中通过"activity"获取IBinder类型的AMS引用。

IBinder b = ServiceManager.getService("activity");

if (false) {

Log.v("ActivityManager", "default service binder = " + b);

}

// 调用asInterface方法来得到ActivityManagerProxy，并将其赋值给am

IActivityManager am = asInterface(b);

if (false) {

Log.v("ActivityManager", "default service = " + am);

}

return am;

}

};

getDefault方法中调用了gDefault的get方法，gDefault是一个Singleton类型的实例，在它内部首先通过ServiceManager拿到AMS的IBinder，然后通过asInterface实例化ActivityManagerProxy并让其代理IBinder。asInterface源码如下：

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

static public IActivityManager asInterface(IBinder obj) {

if (obj == null) {

return null;

}

IActivityManager in =

(IActivityManager)obj.queryLocalInterface(descriptor);

if (in != null) {

return in;

}

return new ActivityManagerProxy(obj);

}

IActivityManager是一个接口，ActivityManagerNative和ActivityManagerProxy都实现了这个接口，用于实现代理模式和IBinder通信。了解aidl的同学应该很容易理解这个过程。

class ActivityManagerProxy implements IActivityManager{

// AMS 的Binder

private IBinder mRemote;

public ActivityManagerProxy(IBinder remote){

mRemote = remote;

}

public IBinder asBinder(){

return mRemote;

}

// ...

}

ActivityManagerProxy中将IBinder赋值给了mRemote成员变量，意味着ActivityManagerProxy作为代理类来代理IBinder。当有向调用AMS的方法时会首先调用ActivityManagerProxy的方法，然后这个方法中通过IBinder和AMS通信。

了解了代理IBinder的这个过程之后，继续看Instrumentation的execStartActivity，这个方法中ActivityManagerNative.getDefault()拿到的就是ActivityManagerProxy，紧接着调用了ActivityManagerProxy的startActivity方法。

frameworks/base/core/java/android/app/ActivityManagerNative.java#ActivityManagerProxy

class ActivityManagerProxy implements IActivityManager

{

public ActivityManagerProxy(IBinder remote)

{

mRemote = remote;

}

public IBinder asBinder()

{

return mRemote;

}

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);

}

// 第一个参数用来表示是调用什么服务，START\_ACTIVITY\_TRANSACTION表示启动Activity。

mRemote.transact(START\_ACTIVITY\_TRANSACTION, data, reply, 0);

reply.readException();

int result = reply.readInt();

reply.recycle();

data.recycle();

return result;

}

// ...

}

这段代码其实就是一个客户端调用服务端的过程，将数据存入data中，通过mRemote.transact将data传递给服务端AMS，AMS接收到数据后做相应的处理。要注意的是mRemote.transact的第一个参数，通过这个参数来标明调用什么服务，START\_ACTIVITY\_TRANSACTION表示是一个启动Activity的操作，AMS通过判断START\_ACTIVITY\_TRANSACTION进而执行Activity的启动流程。

看下服务端AMS的实现

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

@Override

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);

reply.writeNoException();

reply.writeInt(result);

return true;

}

}

// ...

return super.onTransact(code, data, reply, flags);

}

可以看到在ActivityManagerNative的onTransact方法中判断code如果为START\_ACTIVITY\_TRANSACTION，那么久通过data取出客户端传来的数据，并调用startActivity来启动Activity。startActivity的具体实现是在**ActivityManagerService**中，其源码如下：

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

@Override

public final int startActivity(IApplicationThread caller, String callingPackage,

Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,

int startFlags, ProfilerInfo profilerInfo, Bundle options) {

return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,

resultWho, requestCode, startFlags, profilerInfo, options,

UserHandle.getCallingUserId());

}

这个方法中仅仅是调用了startActivityAsUser方法，其源码如下：

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

@Override

public final int startActivityAsUser(IApplicationThread caller, String callingPackage,

Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,

int startFlags, ProfilerInfo profilerInfo, Bundle options, int userId) {

enforceNotIsolatedCaller("startActivity");

userId = handleIncomingUser(Binder.getCallingPid(), Binder.getCallingUid(), userId,

false, ALLOW\_FULL\_ONLY, "startActivity", null);

// TODO: Switch to user app stacks here.

return mStackSupervisor.startActivityMayWait(caller, -1, callingPackage, intent,

resolvedType, null, null, resultTo, resultWho, requestCode, startFlags,

profilerInfo, null, null, options, false, userId, null, null);

}

startActivityAsUser方法中又会通过mStackSupervisor.startActivityMayWait方法来启动Activity，具体过程这里不再赘述。

详细参考 [Activity的启动流程](https://www.learnfk.com/android/android-tutorial-android-system-startup.html)