转载请注明链接：http://blog.csdn.net/feather_wch/article/details/79219316

> 1. 直接从源码角度分析四大组件的机制
> 2. 所有知识点以面试题形式汇总，便于学习和复习背诵

Android四大组件机制详解

版本：2018.2.5-1

参考和学习资料

1. 剖析Activity、Window、ViewRootImpl和View之间的关系

1、四大组件的注册和调用方式

> 1. Activity、Service、ContenProvider必须在 `AndroidManifest` 中注册
> 2. `BroadcastReceiver` 可以在 `AndroidManifest` 中注册，也可以 代码 中注册
> 3. Activity、Service、ContenProvider的调用需要借助 `Intent`
> 4. `BroadcastReceiver` 不需要借助 `Intent`

2、Activity是什么?

> 1. 一种 展示型组件 ， 用于展示界面，并且与用户进行交互。
> 2. Activity的启动由 `Intent` 触发， `Intent` 可分为隐式 `Intent` 和显式 `Intent`
> 3. 显式 `Intent` 需要明确指向一个 `Activity`
> 4. 隐式 `Intent` 可以指向一个或者多个 `Activity` 组件，也可能没有任何 `Activity` 处理该隐式 `Intent`
> 5. `Activity` 具有特定的 启动模式 ， 也可以通过 `finish` 方法结束运行。

3、Service是什么?

> 1. 一种 计算型组件 ， 用于在后台执行一系列计算任务。
> 2. `Service` 具有两种状态： 启动状态 和 绑定状态
> 3. 启动状态: 进行后台任务， `Service` 本身运行在 主线程 ， 因此耗时操作需要在 新线程 中处理
> 4. 绑定状态: 内部同样可以进行后台运算，但是此时 外界 可以很方便与 `Service` 通信
> 5. `Service` 的停止需要灵活采用 `stopService`和`unBindService` 才能完全停止

4、BroadcastReceiver是什么?

> 1. 一种 消息型组件 ， 用于在不同组件甚至不同应用间传递消息
> 2. 静态注册：在AndroidManifest中注册广播，会在 应用安装时被系统解析 ， 不需要启动应用就可以接收到相应广播
> 3. 动态注册： `Context.registerReceiver()` 进行注册， `Context.unRegisterReceiver()` 解除注册. 需要APP启动才能注册并且接收广播。
> 4. 广播发送通过 `Context` 的一系列 `send` 方法完成
> 5. 发送和接收 过程的匹配通过广播接收者的 `intent-filter` 来描述

5、ContentProvider是什么?

> 1. 一种 数据共享型组件
> 2. 内部需要实现 增删改查 四种操作
> 3. 内部的 `insert\delete\update\query` 方法需要处理好线程同步，因为这些方法都在 `Binder`线程池 中调用

Activity

6、Activity的启动方法

```
Intent intent = new Intent(MainActivity.this, Main2Activity.class);
startActivity(intent);
```

7、Activity的startActivity机制分析

```
//Activity.java
//1. 所有`startActivity()`方法最终会调用`startActivityForResult()`方法:
public void startActivityForResult(Intent intent, int requestCode, Bundle options) {
    //2. 父亲不为Null
    if (mParent == null) {
        options = transferSpringboardActivityOptions(options);
        //3. Instrumentation的execStartActivity去启动Activity
        Instrumentation.ActivityResult ar =
                mInstrumentation.execStartActivity(
                        this, mMainThread.getApplicationThread(),//获取ApplicationThread
                        mToken, this,
                        intent, requestCode, options);
        ...
    } else {
        ...
    }
}

//Instrumentation.java
public ActivityResult execStartActivity(Context who, IBinder contextThread, IBinder token, Activity target, Intent intent, int requestCode, Bu
    ...省略...
    try {
        /**=============================================
         * 1.开启Activity
         *  1-获取到IActivityManager的Binder对象
         *  2-通过IPC让ActivityManagerService执行startActivity方法
         *=============================================*/
        int result = ActivityManager.getService() //Binder对象
                .startActivity(whoThread, ... ,options);
        /**=============================================
         *2. 检查启动Activity的结果
         * 没有成功启动就会抛出异常,例如Activity没有注册:
         *   Unable to find explicit activity class...have you declared this activity in your AndroidManifest.xml?"
         *=============================================*/
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}

/**
 * ActivityManagerService处理startActivity流程:
 * startActivity() -> startActivityAsUser -> ...
 * -> ActivityStack的resumeTopActivityUncheckedLocked() -> ...
 * -> ActivityStackSupervisor的`realStartActivityLocked方法`
 */
//ActivityManagerService.java
public final int startActivity(IApplicationThread caller, ...,Bundle bOptions) {
    return startActivityAsUser(caller, ...,UserHandle.getCallingUserId());
}

//ActivityStackSupervisor.java
final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app, boolean andResume, boolean checkConfig) throws RemoteException {
    ......
    //app.thread的类型为IApplicationThread
    /**=============================================================
     * app.thread的类型为IApplicationThread(继承IInterface接口-Binder类型接口)
     * --内部包含大量Activity和Service启动/停止相关功能
     * --具体实现：ActivityThread(继承了ApplicationThreadNative)
     *    * ApplicationThreadNative继承Binder并且实现了IApplicationThread接口
     *      (ApplicationThreadNative和系统为AIDL文件生成的类的作用是一样的)
     *=============================================================*/
    app.thread.scheduleLaunchActivity(new Intent(r.intent), ... ,profilerInfo);
    ......
}

//ActivityThread.java的内部类：ApplicationThread
public final void scheduleLaunchActivity(Intent intent, IBinder token, ...,ProfilerInfo profilerInfo) {
    //1. 保存ActivityClientRecord需要的所有数据
    ActivityClientRecord r = new ActivityClientRecord();
    r.token = token;
    r.ident = ident;
    r.intent = intent;
    ...
    r.overrideConfig = overrideConfig;
    //2. 发送消息给Handler H处理
    sendMessage(H.LAUNCH_ACTIVITY, r);
}

//ActivityThread.java
```

```
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    ......

    public void handleMessage(Message msg) {
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                //1. 交给`ActivityThread`的`handleLaunchActivity`处理
                handleLaunchActivity(r, null, "LAUNCH_ACTIVITY");
            }
            break;
        }
    }
}

//ActivityThread.java
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent, String reason) {
    ......
    //0. 创建Activity前初始化WindowManagerGlobal
    WindowManagerGlobal.initialize();
    //1. 完成Activity对象的创建和启动过程
    Activity a = performLaunchActivity(r, customIntent);
    if (a != null) {
        //2. 调用Activity的onResume这一生命周期
        handleResumeActivity(r.token, ... ,reason);
    } else {
        //3. 如果出错，会finishActivity
        ActivityManager.getService().finishActivity(r.token, Activity.RESULT_CANCELED, null,
                Activity.DONT_FINISH_TASK_WITH_ACTIVITY);
    }
}

//ActivityThread.java
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    //1. 从ActivityClientRecord中获取待启动的Activity的组件信息
    ActivityInfo aInfo = r.activityInfo;
    if (r.packageInfo == null) {
        r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo, Context.CONTEXT_INCLUDE_CODE);
    }
    ...
    //2. 通过Instrumentation的newActivity方法使用类加载器创建Activity对象
    Activity activity = null;
    java.lang.ClassLoader cl = appContext.getClassLoader();
    // 实现简单，就是通过类加载器来创建Activity对象
    activity = mInstrumentation.newActivity(cl, component.getClassName(), r.intent);
    ...
    /**============================================================
     * 3. 通过LoadedApk的makeApplication方法创建Application对象
     *  如果Application已经被创建，则不会重复创建-Application对象唯一
     *  1-内部是通过Instruction来完成，也是通过类加载器来实现
     *  2-Application创建好后，系统会通过Instruction的
     *    callApplicationOnCreate()来调用Application的onCreate()方法
     *============================================================*/
    Application app = r.packageInfo.makeApplication(false, mInstrumentation);
    if (localLOGV) Slog.v(
            TAG, r + ": app=" + app
                    + ", appName=" + app.getPackageName()
                    + ", pkg=" + r.packageInfo.getPackageName()
                    + ", comp=" + r.intent.getComponent().toShortString()
                    + ", dir=" + r.packageInfo.getAppDir());
    /**============================================================
     *4. 创建ContextImpl对象并通过Activity的attach方法来完成一些重要的数据初始化
     * -ContextImpl是Context的具体实现，ContextImpl通过Activity的attach方法和Activity建立关联
     * -attach方法中Activity会完成Window的创建并且建立自己和Window的关联
     *============================================================*/
    ContextImpl appContext = createBaseContextForActivity(r);
    if (activity != null) {
        CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
            ...
        activity.attach(appContext, this, getInstrumentation(), r.token,
                r.ident, app, r.intent, r.activityInfo, title, r.parent,
                r.embeddedID, r.lastNonConfigurationInstances, config,
                r.referrer, r.voiceInteractor, window, r.configCallback);
            ...
        //5. 调用Activity的onCreate方法——Activity完成了整个启动过程
        if (r.isPersistable()) {
            mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
        } else {
            mInstrumentation.callActivityOnCreate(activity, r.state);
        }
    }
    ...
```

```
    return activity;
}

//Activity.java
final void attach(Context context, ActivityThread aThread, ...) {
      //1. 建立Context和Activity的关联
      attachBaseContext(context);
      //2. 数据初始化：UI线程为当前线程，application等等
      mUiThread = Thread.currentThread();
      mApplication = application;
      ...
      //3. 创建Window(PhoneWindow)
      mWindow = new PhoneWindow(this);
      mWindow.setCallback(this);
      mWindow.setOnWindowDismissedCallback(this);
      mWindow.getLayoutInflater().setPrivateFactory(this);
      ...
      //4. 给Window设置WindowManager(从WindowManagerService获取)
      mWindow.setWindowManager(
                (WindowManager)context.getSystemService(Context.WINDOW_SERVICE), ...);
      //5. 当前Window的容器是父Activity的Window
      if (mParent != null) {
            mWindow.setContainer(mParent.getWindow());
      }
      //6. 将当前Window的WindowManager保存到Activity内部
      mWindowManager = mWindow.getWindowManager();
      mCurrentConfig = config;
}
```

Service

7、Service的启动方法

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

> 1. Service有 启动状态 和 绑定状态
> 2. 两个状态可以共存 ， Service可以既处于启动状态又处于绑定状态

8、Service的启动过程详解

```
/**
 * ====================================
 * 1. Activty层层继承自ContextWrapper
 * 2. Activty的startService()方法来自于ContextWrapper
 * 3. ContextWrapper最终由mBase(ContextImpl)完成-典型桥接
 * ====================================
 */
//ContextWrapper.java
public ComponentName startService(Intent service) {
    //1. mBase就是Context的实现ContextImpl对象(也就是Activity创建时关联的对象)
    return mBase.startService(service);
}

//ContextImpl.java：直接调用startServiceCommon
public ComponentName startService(Intent service) {
    warnIfCallingFromSystemProcess();
    return startServiceCommon(service, false, mUser);
}

//ContextImpl.java
private ComponentName startServiceCommon(Intent service, boolean requireForeground, UserHandle user) {
    ......
    //1. 让`ActivityManagerService`启动一个Service服务
    ComponentName cn = ActivityManager.getService().startService(
            mMainThread.getApplicationThread(), service, ...省略...);
    ......
}

//ActivityManagerService.java
public ComponentName startService(IApplicationThread caller, Intent service, ...) {
    /**==================================
     * 1. 通过mService(ActiveServices)完成后续过程
     * 2. ActiveServices是辅助AMS进行Service管理的类
     *     -包括：启动、绑定、停止
     * 3. `startServiceLocked`方法尾部会调用`startServiceInnerLocked`
     *==================================*/
    res = mServices.startServiceLocked(caller, service, ...,userId);
}

//ActiveServices.java
ComponentName startServiceInnerLocked(...,ServiceRecord r) {
    ......
    /**========================================
     * ServiceRecord描述的是一个Service记录(贯穿整个启动过程)
     * 1. startServiceInnerLocked并没有完成具体启动工作，而是把后续任务交给了bringUpServiceLocked
     * 2. bringUpServiceLocked内部调用`realStartServiceLocked`
     * 3. realStartServiceLocked真正启动了Service
     *========================================*/
    String error = bringUpServiceLocked(r, service.getFlags(), callerFg, false, false);
    ......
    return r.name;
}

//ActiveServices.java
private final void realStartServiceLocked(ServiceRecord r, ProcessRecord app, boolean execInFg) {
    ......
    /**==========================================================
     * 创建了Service对象，并且调用了onCreate()方法-IPC通信
     * 1. app.thread对象是IApplicationThread类型(Binder)
     * 2. 具体实现是ActivityThread(继承了ApplicationThreadNative)
     *==========================================================*/
    app.thread.scheduleCreateService(r, r.serviceInfo, ......);
    ......
    //2. 用于调用Service的其他方法(如onStartCommand)-IPC通信
    sendServiceArgsLocked(r, execInFg, true);
    ......
}

//ActivityThread.java的内部类：ApplicationThread
public final void scheduleCreateService(IBinder token, ...,int processState) {
    updateProcessState(processState, false);
    CreateServiceData s = new CreateServiceData();
    s.token = token;
    s.info = info;
    s.compatInfo = compatInfo;
    /**========================================
     *  1. 发送消息给Handler H处理
     *  2. H会接受消息，并且调用ActivityThread的handleCreateService
     *========================================*/
    sendMessage(H.CREATE_SERVICE, s);
}
```

```
/**====================================
 * 完成Service最终启动工作
 * //ActivityThread.java
 *====================================*/
private void handleCreateService(CreateServiceData data) {
    //1. 通过类加载器创建Service实例
    Service service = null;
    java.lang.ClassLoader cl = packageInfo.getClassLoader();
    service = (Service) cl.loadClass(data.info.name).newInstance();
    //2. 创建Application对象并调用其onCreate方法(Application是唯一的不会重复创建)
    Application app = packageInfo.makeApplication(false, mInstrumentation);
    //3. 创建ContextImpl对象并通过Service的attach方法建立两者关系(类似Activity的过程)
    ContextImpl context = ContextImpl.createAppContext(this, packageInfo);
    context.setOuterContext(service);
    service.attach(context, this, data.info.name, data.token, app, ActivityManager.getService());
    //4. 调用service的onCreate方法，并且将Service对象存储到ActivityThread中的一个列表中
    service.onCreate();
    mServices.put(data.token, service);
    ......
}

/**============================================================
 * ActivityThread中还会通过handleServiceArgs方法调用Service的onStartCommand
 *============================================================*/
private void handleServiceArgs(ServiceArgsData data) {
    Service s = mServices.get(data.token);
    ......
    //1. Service的onStartCommand方法
    res = s.onStartCommand(data.args, data.flags, data.startId);
    ......
}
```

9、Service的绑定过程

```
/**
 * ============================================
 * 1. bindService最终也是调用的ContextWrapper的方法
 * 2. 与启动过程类似，mBase是ContextImpl最终会调用自身的bindServiceCommon方法
 * //ContextWrapper.java
 * ============================================
 */
public boolean bindService(Intent service, ServiceConnection conn, int flags) {
    return mBase.bindService(service, conn, flags);
}

//ContextImpl.java
private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags, Handler
        handler, UserHandle user) {
    /**=========================================================================
     * 1. 将客户端的ServiceConnection对象转化为`ServiceDispatcher.InnerConnection`对象
     *  -ServiceConnection必须借助于Binder才能让远程服务端回调自己的方法
     *  -ServiceDispatcher的内部类InnerConnection就起到了Binder的作用
     *  -ServiceDispatcher起到连接ServiceConnection和InnerConnection的作用
     *=========================================================================*/
    IServiceConnection sd;
    sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(), handler, flags);
    //2. 通过ActivityManagerService完成Service的绑定过程
    int res = ActivityManager.getService().bindService(... , service,...);
    ......
}

//LoadedApk.java
public final IServiceConnection getServiceDispatcher(ServiceConnection c, Context context, Handler handler, int flags) {
    /**================================
     * 1.mServices是ArrayMap:存储应用当前活动的ServiceConnection
     *    和ServiceDispatcher的映射关系
     *================================*/
    synchronized (mServices) {
        LoadedApk.ServiceDispatcher sd = null;
        //2. 获取`映射关系`的map
        ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher> map = mServices.get(context);
        if (map != null) {
            //3. 通过ServiceConnection去查询是否有ServiceDispatcher
            sd = map.get(c);
        }
        //4. 不存在ServiceDispatcher,新建ServiceDispatcher对象,
        if (sd == null) {
            sd = new ServiceDispatcher(c, context, handler, flags);
            if (map == null) {
                map = new ArrayMap<>();
                //6. 将该`映射关系`与Context放置到ArrayMap中
                mServices.put(context, map);
            }
            //5. key=ServiceConnection,value=ServiceDispatcher,建立映射关系
            map.put(c, sd);
        }
        //7. 返回ServiceDispatcher内部保存的InnerConnection
        return sd.getIServiceConnection();
    }
}

//ActivityManagerService.java
public int bindService(IApplicationThread caller, IBinder token, Intent service,...) {
    ......
    /**=========================
     * ActiveServices的方法:
     *  1. bindServiceLocked
     *  2. bringUpServiceLocked
     *  3. realStartServiceLocked
     *  4. 最后都是通过ActivityThread来完成Service实例的创建
     *      并且执行Services的onCreate方法
     *  * Service绑定与启动的不同在于会调用app.thread的scheduleBindService方法
     *      (在ActiveServices的requestServiceBindingLocked中调用)
     *==========================*/
    return mServices.bindServiceLocked(caller, token, service,...);
}

//ActiveServices.java
private final boolean requestServiceBindingLocked(ServiceRecord r, IntentBindRecord i,...) {
    ......
    //ActivityThread内部类：`ApplicationThread`一中一系列`schedule`方法之一,最终通过Handler H进行中转，最终交给handleBindServices
    r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind, r.app.repProcState);
    ......
}
```

```
//ActivityThread
private void handleBindService(BindServiceData data) {
    //1. 根据token取出Service
    Service s = mServices.get(data.token);
    if (s != null) {
        if (!data.rebind) {
            /**================================
             * 2. 调用Service的onBind方法
             *  -此时Service就已经处于绑定状态，但此时客户端并不知道连接成功
             *  -因此必须调用客户端ServiceConnection中的onServiceConnected
             *================================*/
            IBinder binder = s.onBind(data.intent);
            /**========================================
             * 3. ActivityManagerService的publishService
             *  -1.会执行客户端ServiceConnection中的onServiceConnected
             *  -2.保证Service的onBind方法之调用一次(多次绑定同一个Service)
             *  -3.最终将具体任务交给ActiveServices的publishServiceLocked方法
             *========================================*/
            ActivityManager.getService().publishService(data.token, data.intent, binder);
        }
    }
    ......
}

//ActiveServices.java
void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {
    ......
    /**==================================
     * 1. c是ConnectionRecord
     * 2. c.conn是ServiceDispatcher.InnerConnection
     * 3. service就是Service的onBind方法返回的Binder对象
     *==================================*/
    c.conn.connected(r.name, service, false);
    ......
}

//LoadedApk.java的内部类ServiceDispatcher的内部类InnerConnection
private static class InnerConnection extends IServiceConnection.Stub {
    ......

    public void connected(ComponentName name, IBinder service, boolean dead) {
        LoadedApk.ServiceDispatcher sd = mDispatcher.get();
        if (sd != null) {
            //1. 调用ServiceDispatcher的方法
            sd.connected(name, service, dead);
        }
    }
}

//LoadedApk.java的内部类: ServiceDispatcher
public void connected(ComponentName name, IBinder service, boolean dead) {
    /**=========================================================
     *1. mActivityThread是一个Handler，其实就是ActivityThread中的H
     *2. 最终RunConnection通过H的post方法从而运行在主线程中
     *3. 因此客户端ServiceConnection就是在主线程被回调
     *=========================================================*/
    mActivityThread.post(new RunConnection(name, service, 0, dead));
}

//LoadedApk.java内部类ServiceDispatcher的内部类: RunConnection
private final class RunConnection implements Runnable {
    ......
    /**
     * ==========================================================
     * 1. 本质调用ServiceDispatcher的doConnected
     * 2. ServiceDispatcher内部拥有客户端的ServiceConnection
     * ==========================================================
     */
    public void run() {
        if (mCommand == 0) {
            doConnected(mName, mService, mDead);
        } else if (mCommand == 1) {
            doDeath(mName, mService);
        }
    }
}

//LoadedApk.java内部类: ServiceDispatcher
public void doConnected(ComponentName name, IBinder service, boolean dead) {
    ....
    if (service != null) {
        //1. 可以通过客户端的ServiceConnection调用onServiceConnected
```

```
            mConnection.onServiceConnected(name, service);
        }
    }
```

10、广播的静态注册过程:

> 1. 安装应用时由系统自动完成注册
> 2. 具体是由 PMS(Package Manager Service) 来完成注册过程
> 3. 本质其他 三大组件 的注册都是在安装时由 PMS 解析并注册

11、广播的动态注册过程:

```
/**
 * ================================
 * 1. 动态注册是从ContextWrapper的registerReceiver方法开始
 * 2. 之后直接交给ContextImpl完成
 * //ContextWrapper.java
 * ================================
 */
public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {
    //1. mBase = ContextImpl
    return mBase.registerReceiver(receiver, filter);
}

//ContextImpl.java
public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,......) {
    return registerReceiverInternal(receiver, ......);
}

//ContextImpl.java
private Intent registerReceiverInternal(BroadcastReceiver receiver, ......) {
    IIntentReceiver rd = null;
    if (receiver != null) {
        if (mPackageInfo != null && context != null) {
            //1. 从mPackageInfo获取IIntentReceiver对象
            rd = mPackageInfo.getReceiverDispatcher(receiver, context, scheduler, ......);
        } else {
            /**================================================================================
             *2.从mPackageInfo获取IIntentReceiver对象
             * 1-采用IIntentReceiver而不是BroadcastReceiver是因为这是IPC过程
             * 2-BroadcastReceiver作为组件不能直接进行IPC，需要进行中转
             * 3-IIntentReceiver是Binder接口，具体实现是LoadedApk.ReceiverDispatcher.InnerReceiver
             * 4-ReceiverDispatcher中同时保存了 BroadcastReceiver和InnerReceiver，接收广播时ReceiverDispatcher
             *     可以很方便调用BroadcastReceiver的onReceive()方法
             * 5-可以发现Service也有ServiceDispatcher和内部类InnerConnection(Binder接口)，原理相同
             *================================================================================*/
            rd = new LoadedApk.ReceiverDispatcher(receiver, context, scheduler, null, true)
                    .getIIntentReceiver();
        }
    }
    //3. 通过ActivityManagerService，远程进行注册
    final Intent intent = ActivityManager.getService().registerReceiver(
            mMainThread.getApplicationThread(), mBasePackageName, rd, filter,
            broadcastPermission, userId, flags);
        ......
}

//ActivityManagerService.java：广播完成注册
public Intent registerReceiver(IApplicationThread caller, ...,IIntentReceiver receiver, ...) {
    ......
    //1. 存储远程的InnerReceiver对象(本地的BroadcastReceiver对应的对象)
    mRegisteredReceivers.put(receiver.asBinder(), rl);
    //2. 存储IntentFiler对象
    BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage,
            permission, callingUid, userId, instantApp, visibleToInstantApps);
    rl.add(bf);

    mReceiverResolver.addFilter(bf);
    ......
}
```

12、广播的发送和发送过程(普通广播为例):

> 1. 通过 sendBroadcast 发送广播时，AMS会查找出匹配的广播接收者并将广播发送给它们处理
> 2. 广播分为：普通广播、有序广播和粘性广播

```
/**
 * ================================================
 * 1. 广播的发送开始于ContextWrapper的sendBroadcast方法
 * 2. 最终会交给ContextImpl的sendBroadcast方法去处理
 * // ContextImpl.java
 * ================================================
 */
public void sendBroadcast(Intent intent) {
    //1. 直接向AMS发起一个异步请求用于发送广播
    ActivityManager.getService().broadcastIntent(......);
    ......
}
//ActivityManagerService.java
public final int broadcastIntent(IApplicationThread caller, Intent intent, ......) {
    ....
    int res = broadcastIntentLocked(callerApp, ......);
}
//ActivityManagerService.java
final int broadcastIntentLocked(ProcessRecord callerApp, ......) {
    intent = new Intent(intent);
    /**============================================================
     *1. 默认情况下广播不会发送给已经停止的应用(从Android 3.1开始)
     * Intent中新增两个标记:
     *  FLAG_EXCLUDE_STOPPED_PACKAGES-不包含已经停止应用
     *  FLAG_INCLUDE_STOPPED_PACKAGES-包含已经停止应用
     * -如果两个标记共存，则以FLAG_INCLUDE_STOPPED_PACKAGES为准
     * -停止状态为：1-应用安装后未运行 2-应用被手动或者其他应用强制停止
     *============================================================*/
    intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);
    ......
    /**================================================
     *2. 根据intent-filter查找出匹配的广播接收者
     *3. 进过一系列过滤后，将满足条件的广播接收者添加到`BroadcastQueue`
     *4. BroadcastQueue就会将广播发送给相应的广播接收者
     *================================================*/
    if ((receivers != null && receivers.size() > 0)
            || resultTo != null) {
        BroadcastQueue queue = broadcastQueueForIntent(intent);
        BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp, ......);

       ......
        queue.enqueueOrderedBroadcastLocked(r);
        //4. BroadcastQueue就会将广播发送给相应的广播接收者
        queue.scheduleBroadcastsLocked();
    }
    ......
    return ActivityManager.BROADCAST_SUCCESS;
}
//BroadcastQueue.java
public void scheduleBroadcastsLocked() {
    //1. 发送消息，BroadcastQueue收到消息后会调用processNextBroadcast方法
    mHandler.sendMessage(mHandler.obtainMessage(BROADCAST_INTENT_MSG, this));
}
//BroadcastQueue.java
final void processNextBroadcast(boolean fromMsg) {
    synchronized (mService) {
        BroadcastRecord r;
        ......
        //1. 普通广播处理
        while (mParallelBroadcasts.size() > 0) {
            //2. 无序广播存储在mParallelBroadcasts中
            r = mParallelBroadcasts.remove(0);
            final int N = r.receivers.size();
            //3. 取出广播并发送给他们所有的接受者
            for (int i = 0; i < N; i++) {
                Object target = r.receivers.get(i);
                //4. 发送广播
                deliverToRegisteredReceiverLocked(r, (BroadcastFilter) target, false, i);
            }
            addBroadcastToHistoryLocked(r);
        }
        ......
    }
}
//BroadcastQueue.java
private void deliverToRegisteredReceiverLocked(BroadcastRecord r, BroadcastFilter filter, boolean ordered, int index) {
    performReceiveLocked(filter.receiverList.app, filter.receiverList.receiver, ......);
}
//BroadcastQueue.java
void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver, ...) {
    ......
```

```
    //1. app.thread为ActivityThread,会调用其中方法
    app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode, data, extras, ordered, sticky, sendingUser, app.repProcState);
}
//ActivityThread.java
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent, ......) {
    updateProcessState(processState, false);
    //1. 通过`InnerReceiver`实现广播的接收，内部会调用ReceiverDispatcher的performReceive方法
    receiver.performReceive(intent, resultCode, dataStr, extras, ordered, sticky, sendingUser);
}
//LoadedApk.java内部类ReceiverDispatcher
public void performReceive(Intent intent, int resultCode, String data, ......) {
    //1. 创建Args对象
    final Args args = new Args(intent, resultCode, data, extras, ordered, sticky, sendingUser);
    ......
    /**=========================================================
     * 2. 通过mActivityThread的post方法来执行args中的逻辑
     *   -mActivityThread是Handler(也就是ActivityThread中的Handler H)
     *   -Args中实现了Runnable接口-在广播接受线程中执行了onReceive方法
     *=========================================================*/
    if (intent == null || !mActivityThread.post(args.getRunnable())) {
        ......
    }
}
//LoadedApk.java内部类ReceiverDispatcher.Args
final class Args extends BroadcastReceiver.PendingResult {
    .......

    public final Runnable getRunnable() {
        return () -> {
            //1. 执行了BroadcastReceiver的onReceive方法
            final BroadcastReceiver receiver = mReceiver;
            ......
            receiver.onReceive(mContext, intent);
            ......
        };
    }
}
```

13、ContentProvider要点

> 1. ContentProvider所在进程启动时，就会同时启动并且发布到AMS中
> 2. ContentProvider的onCreate要先于Application的onCreate执行

14、ContentProvider的启动流程

> 1. `ActivityThread` 的 `main` 方法为应用启动时的入口， `main` 是静态方法——会创建 `ActivityThread` 的实例，并且创建 主线程 的 消息队列
> 2. 然后会在 `ActivityThread` 的 `attach()` 方法中远程调用 `AMS` 的 `attachApplication` 方法并将 `ApplicationThread` 对象提供给 `AMS`
> 3. `ApplicationThread` 是Binder对象，Binder接口是 `IApplicationThread` ，主要用于 `ActivityThread` 和 `AMS` 之间的通信
> 4. `AMS` 的 `attachApplication` 中会调用 `ApplicationThread` 的 `bindApplication` 方法(IPC过程), `bindApplication` 的逻辑会通过 `ActivityThread` 中的 `Handler H` 切换到 `ActivityThread` 中的 `handleBindApplication` 去处理
> 5. `handleBindApplication` 中会创建 `Application` 对象并且加载 `ContentProvider`
> 6. 加载 `ContentProvider` 后，才会调用 `Application` 的 `onCreate` 方法

15、ContentProvider的数据访问

> 1. ContentProvider启动后，外界就可以通过提供的接口进行增删改查
> 2. 外界无法直接访问 `ContentProvider` ，需要通过 `AMS` 根据 `Uri` 来获取对应的 `ContentProvider` 的Binder接口 `IContentProvider`
> 3. 然后通过 `IContentProvider` 来访问其数据源

```
/**
 * ============================================================================
 * 1. 其他应用通过AMS来访问指定的ContentProvider
 * 2. 通过AMS获得ContentProvider的Binder对象：IContentProvider
 * 3. IContentProvider的具体实现ContentProvider.Transport(继承自ContentProviderNative)
 * 以query为例：最终会通过IPC调用到ContentProvider.Transport的query方法
 * //ContentProvider.java内部类：Transport
 * ============================================================================
 */
public Cursor query(String callingPkg, Uri uri, String[] projection, ......) {
    ......
    //1. 调用了ContentProvider的query方法
    Cursor cursor = ContentProvider.this.query(uri, projection, queryArgs, CancellationSignal.fromTransport(cancellationSignal));
    ......
}
```

16、ContentProvider的数据访问解析

> 1. 访问 `ContentProvider` 需要通过 `ContentResolver` ， 这是一个抽象类
> 2. `Context`的`getContentResolver()` 本质获取的是 `ApplicationContentResolver` 对象(ContextImpl的内部类)
> 3. 当 `ContentProvider` 所在进程未启动时，第一次访问会触发 `ContentProvider` 的创建和所在进程的启动。
> 4. 例如 `query` 方法，首先会获取 `IContentProvider` 对象，最终通过 `acquireProvider` 来获取 `ContentProvider`

17、ContentProvider源码解析

```
//ContextImpl.java的内部类：ApplicationContentResolver
protected IContentProvider acquireProvider(Context context, String auth) {
    //1. 直接调用`ActivityThread`的方法
    return mMainThread.acquireProvider(context, ContentProvider.getAuthorityWithoutUserId(auth), resolveUserIdFromAuthority(auth), true);
}

//ActivityThread.java
public final IContentProvider acquireProvider(Context c, String auth, int userId, boolean stable) {
    //1. 查找是否已经存在需要的ContenProvider
    final IContentProvider provider = acquireExistingProvider(c, auth, userId, stable);
    if (provider != null) {
        //2. 存在就直接返回——ActivityThread通过mProviderMap来存储已经启动的ContentProvider
        return provider;
    }
    ContentProviderHolder holder = null;
    //3. 不存在就发送请求让`AMS`启动需要的`ContentProvider`
    holder = ActivityManager.getService().getContentProvider(getApplicationThread(), auth, userId, stable);
    ......
    //4. 最后修改引用计数
    holder = installProvider(c, holder, holder.info, true, holder.noReleaseNeeded, stable);
    return holder.provider;
}

//ActivityManagerService.java
public final ContentProviderHolder getContentProvider(IApplicationThread caller, String name, int userId, boolean stable) {
    ...
    return getContentProviderImpl(caller, name, null, stable, userId);
}

//ActivityManagerService.java
private ContentProviderHolder getContentProviderImpl(IApplicationThread caller, ...) {
    ContentProviderRecord cpr;
    ContentProviderConnection conn = null;
    ProviderInfo cpi = null;
    ......
    //1. 会先启动ContentProvider所在的进程，然后才会启动ContentProvider
    /**=================================================================
     * 1. 会先启动ContentProvider所在的进程，然后才会启动ContentProvider
     * 2. startProcessLocked中主要是通过Process的start方法来完成新进程的启动
     * 3. 新进程启动后入口方法在ActivityThread的main方法(个人认为这是ContentProvider的进程不是我们自己应用的)
     *=================================================================*/
    proc = startProcessLocked(cpi.processName,
            cpr.appInfo, false, 0, "content provider",
            new ComponentName(cpi.applicationInfo.packageName,
                    cpi.name), false, false, false);
    ......
    return cpr != null ? cpr.newHolder(conn) : null;
}

//ActivityThread.java
public static void main(String[] args) {
    ...
    //1. 首先会创建ActivityThread实例
    ActivityThread thread = new ActivityThread();
    //2. 然后调用attach-进行一系列初始化
    thread.attach(false);
    //3. 然后开始消息循环
    Looper.prepareMainLooper();
    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }
    if (false) {
        Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG, "ActivityThread"));
    }
    Looper.loop();
    ...
}

//ActivityThread.java
private void attach(boolean system) {
    ......
    //1. 将ApplicationThread对象传输给AMS(IPC)
    final IActivityManager mgr = ActivityManager.getService();
    mgr.attachApplication(mAppThread);
    ......
}

//ActivityManagerService.java
public void attachApplication(IApplicationThread thread) {
    attachApplicationLocked(thread, callingPid);
}
```

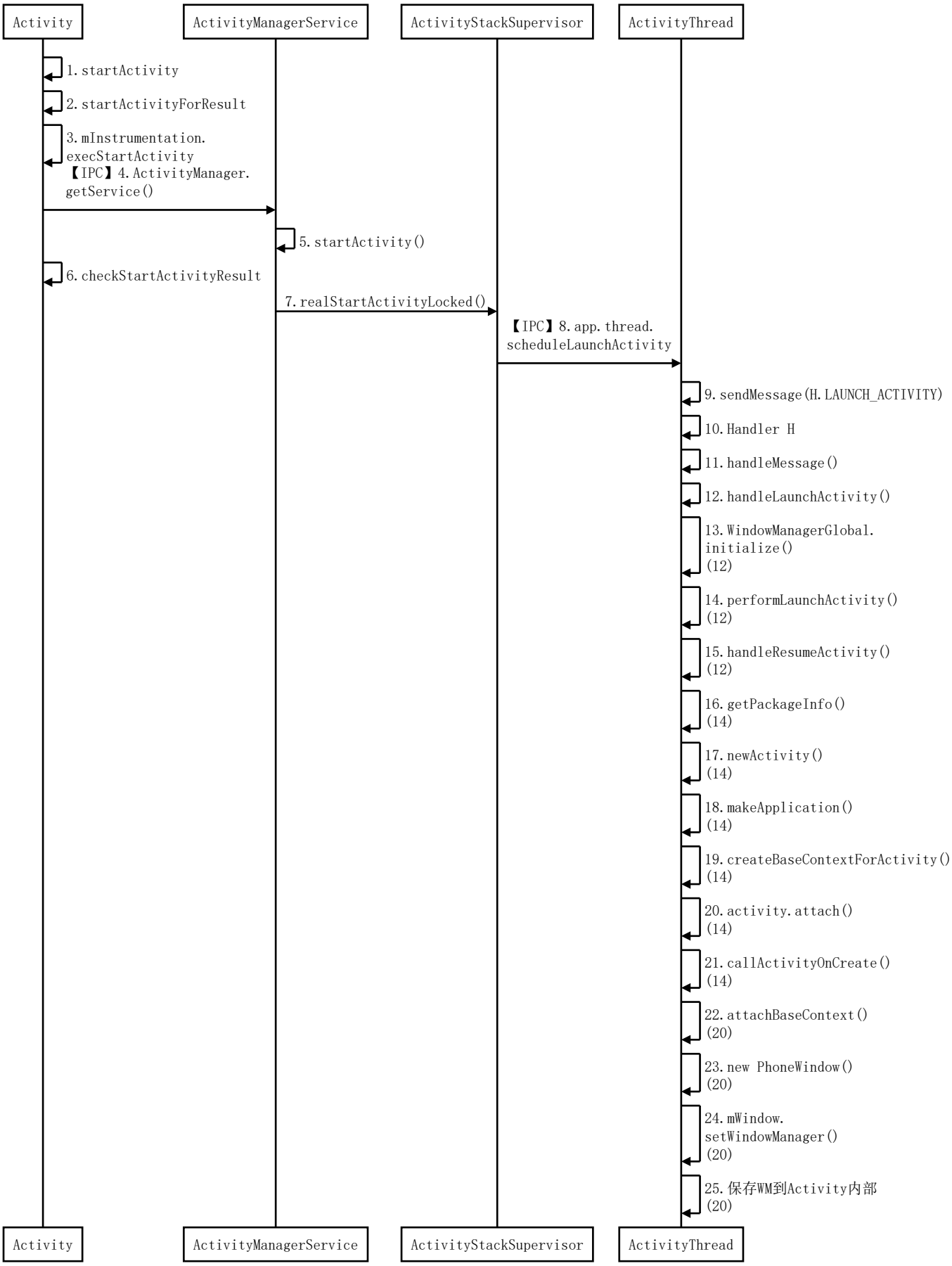
```
//ActivityManagerService.java
private boolean attachApplicationLocked(IApplicationThread thread, int pid) {
    ......
    thread.bindApplication(processName, appInfo, providers, ......);
    ......
}

//ActivityThread.java内部类：ApplicationThread
public final void bindApplication(String processName, ApplicationInfo appInfo,......) {
    ......
    //1. 发送消息给Handler H(ActivityThread)
    sendMessage(H.BIND_APPLICATION, data);
}
//ActivityThread.java

/**
 * ================================================
 * -完成了Application的创建
 * -以及ContentProvider的创建
 * //ActivityThread.java
 * ================================================
 */
private void handleBindApplication(AppBindData data) {
    ...
    //1. 创建ContextImpl对象和Instrumentation
    final ContextImpl instrContext = ContextImpl.createAppContext(this, pi);
    final ClassLoader cl = instrContext.getClassLoader();
    //Instrumentation
    mInstrumentation = (Instrumentation) cl.loadClass(data.instrumentationName.getClassName()).newInstance();
    final ComponentName component = new ComponentName(ii.packageName, ii.name);
    mInstrumentation.init(this, instrContext, appContext, component, data.instrumentationWatcher, data.instrumentationUiAutomationConnectio
    //2. 创建Application对象
    Application app = data.info.makeApplication(data.restrictedBackupMode, null);
    mInitialApplication = app;
    //3. 启动当前进程的ContentProvider并调用其onCreate方法
    if (!data.restrictedBackupMode) {
        if (!ArrayUtils.isEmpty(data.providers)) {
            installContentProviders(app, data.providers); //启动并且调用onCreate
            mH.sendEmptyMessageDelayed(H.ENABLE_JIT, 10 * 1000);
        }
    }
    //4. 调用Application的onCreate方法
    mInstrumentation.callApplicationOnCreate(app);
}
//ActivityThread.java
private void installContentProviders(Context context, List<ProviderInfo> providers) {
    final ArrayList<ContentProviderHolder> results = new ArrayList<>();
    //1. 遍历当前进程的Provider列表
    for (ProviderInfo cpi : providers) {
        //2. 调用installProvider进行启动
        ContentProviderHolder cph = installProvider(context, null, cpi, ......);
        if (cph != null) {
            cph.noReleaseNeeded = true;
            results.add(cph);
        }
    }
    //2. 将已经启动的ContentProvider保存在AMS的ProviderMap中， 外部调用者就可以直接从AMS中获取ContentProvider
    ActivityManager.getService().publishContentProviders(getApplicationThread(), results);
    ......
}
//ActivityThread.java
private ContentProviderHolder installProvider(Context context, ......) {
    ContentProvider localProvider = null;
    IContentProvider provider;
    ...
    //1. 通过类加载器完成了ContentProvider对象的创建
    final java.lang.ClassLoader cl = c.getClassLoader();
    localProvider = (ContentProvider) cl.loadClass(info.name).newInstance();
    provider = localProvider.getIContentProvider();
    if (provider == null) {
        return null;
    }
    //2. 通过ContextProvider方法调用了onCreate方法
    localProvider.attachInfo(c, info);
    ...
}
```

序列图解析四大组件流程

18、Activity的启动

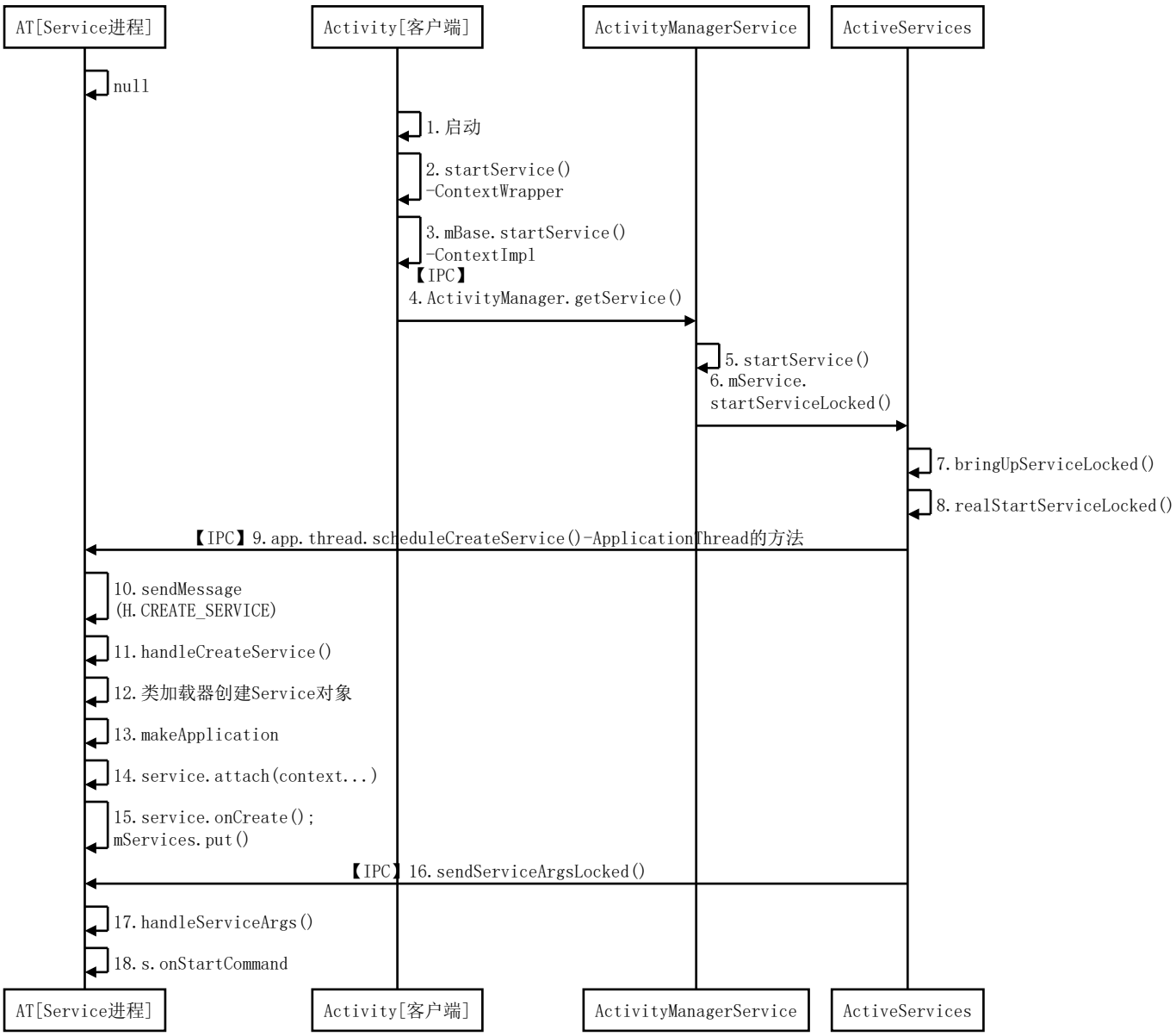


13.创建Activity前初始化WindowManagerGlobal
14.完成Activity对象的创建和启动过程
15.调用Activity的onResume这一生命周期
17.通过类加载器来创建Activity对象
18.通过LoadedApk的makeApplication方法创建Application对象(唯一)，并会调用 `onCreate()`

> 19.创建ContextImpl，并调用attach
> 20.关联了Context和Activity，并且创建Window加载WM等初始化工作
> 21.调用Activity的onCreate方法

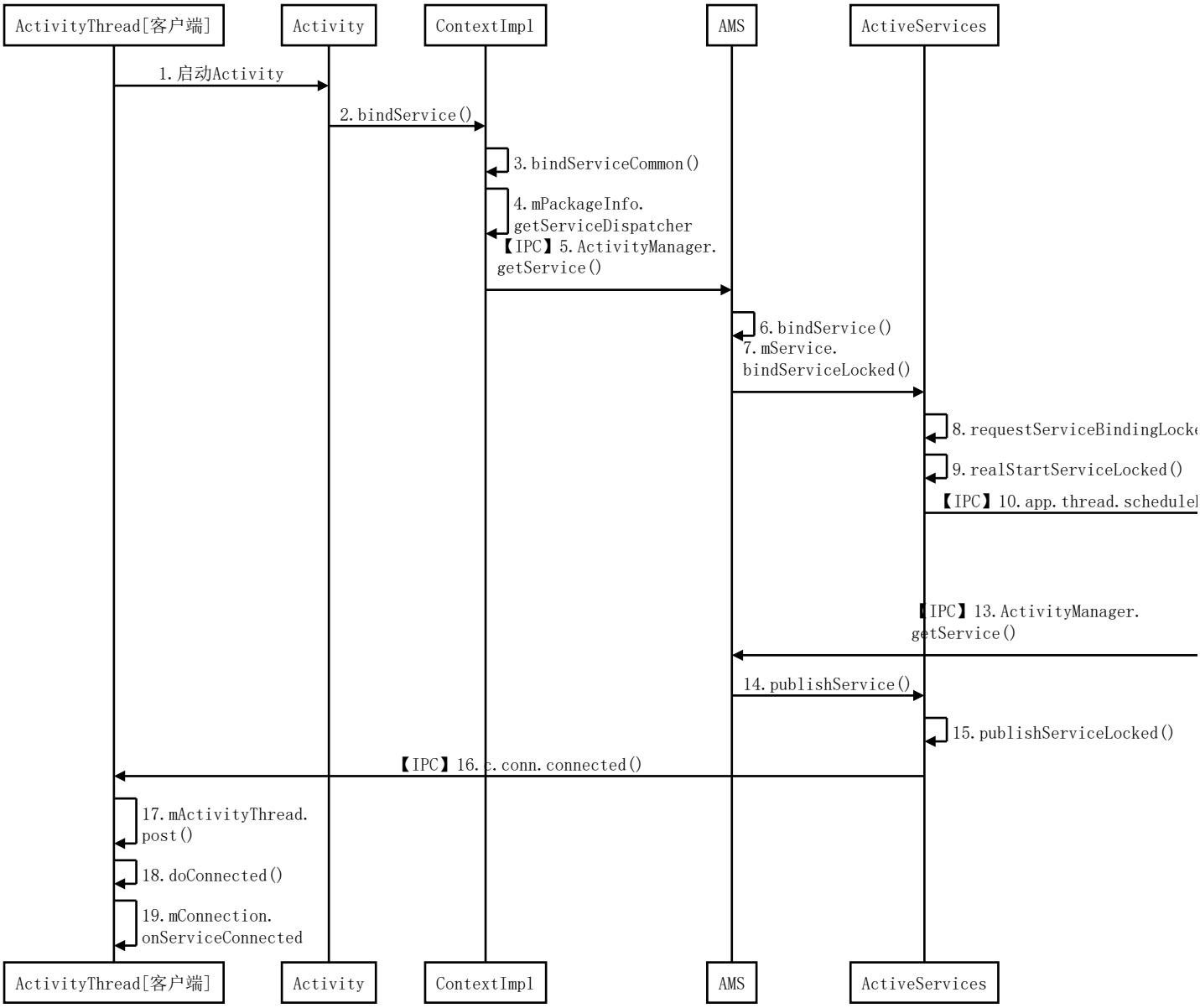
19、Service的启动

```
AT: ActivityThread
```



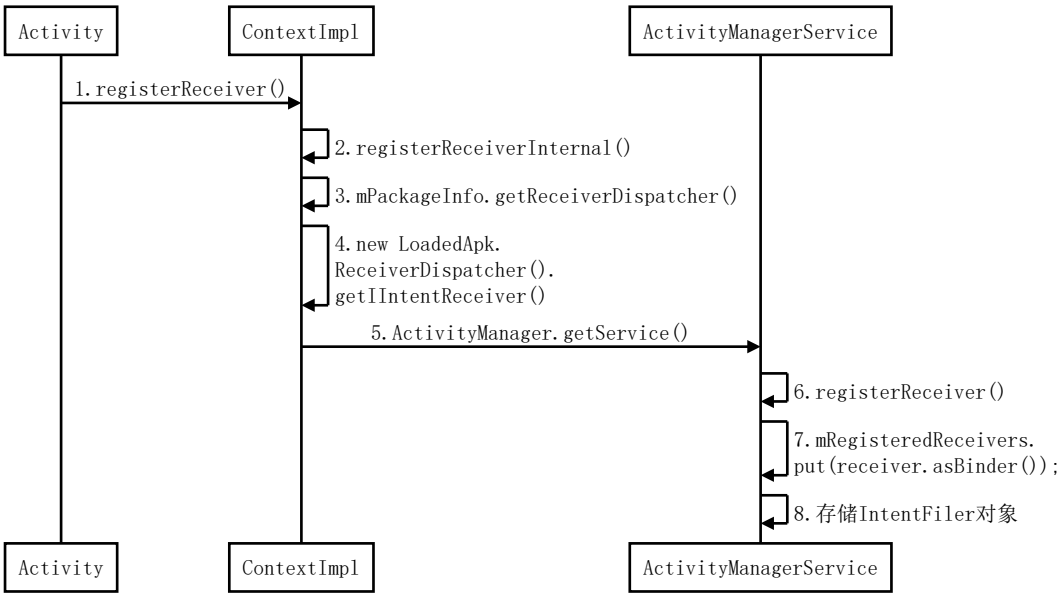
> 11.Handler H接受并且处理消息，最终调用handleCreateService()
> 13.makeApplication(创建Application对象并调用onCreate()-若已经存在则不创建)
> 14.创建ContextImpl并调用attach方法-建立ContextImpl和Service的联系
> 15.service.onCreate()，并将Service添加到ActivityThread内部的Service列表中
> 16.sendServiceArgsLocked()-内部最终调用Service的其他方法(onStartCommand等)

20、Service的绑定



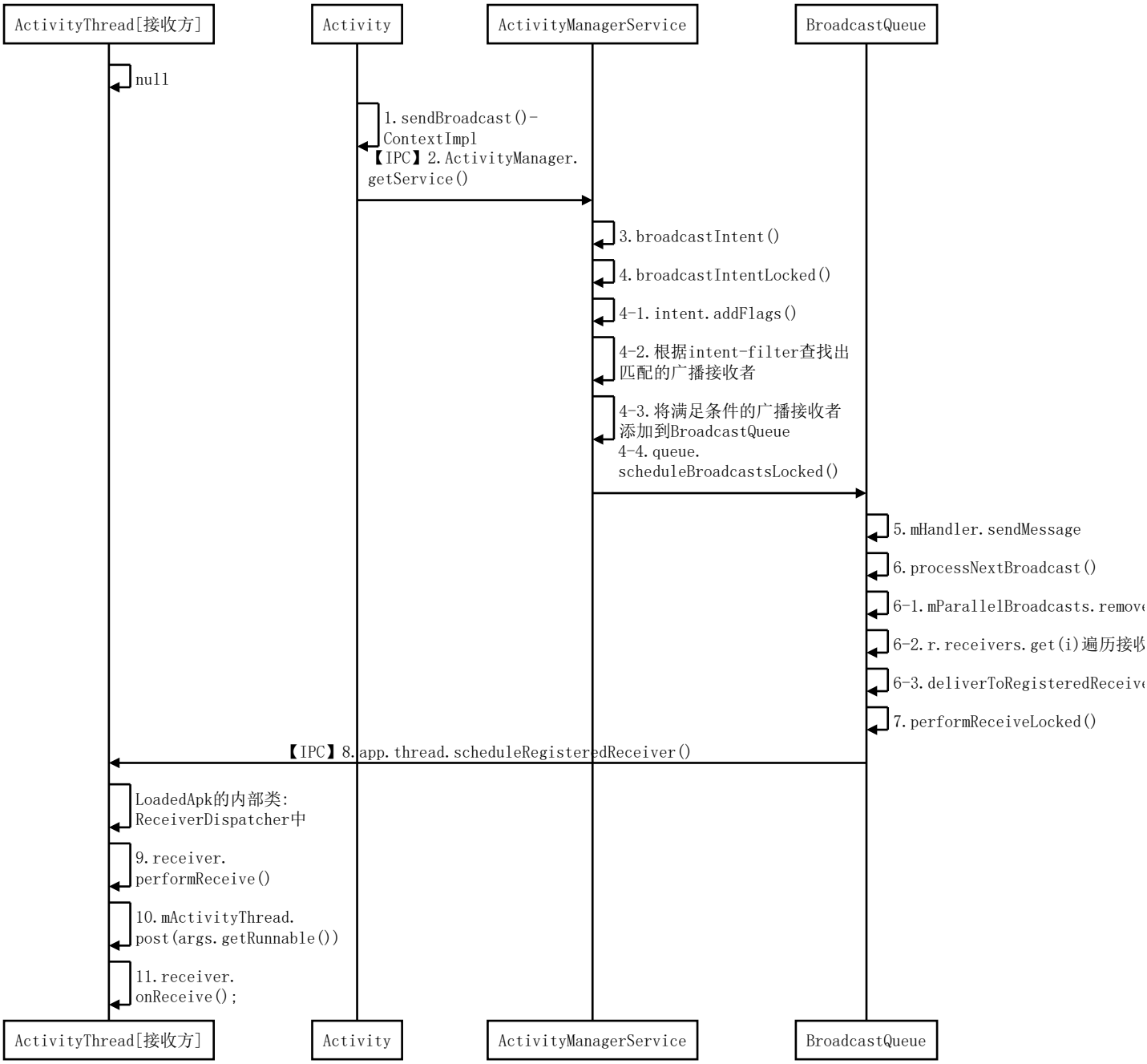
> 2 最终是会调用ContextImpl的bindServiceCommon方法
> 4.ServiceConnection需要借助binder才能让远程服务回调自己的方法(借助于ServiceDispatcher.InnerConnection)
> 10.scheduleBindService会发送消息，最终由handleBindService处理
> 12.调用Service的onBind方法-绑定成功
> 13.绑定成功后需要通知客户端：最终调用客户端ServiceConnection中的onServiceConnected
> 16.c.conn是ServiceDispatcher.InnerConnection(ServiceConnection的Binder中转对象)，最终调用ServiceDispatcher的connected
> 17.mActivityThread就是ActivityThread的Hanlder H
> 18.通过post最终运行在主线程
> 19.调用客户端的onServiceConnected方法

21、广播的动态注册



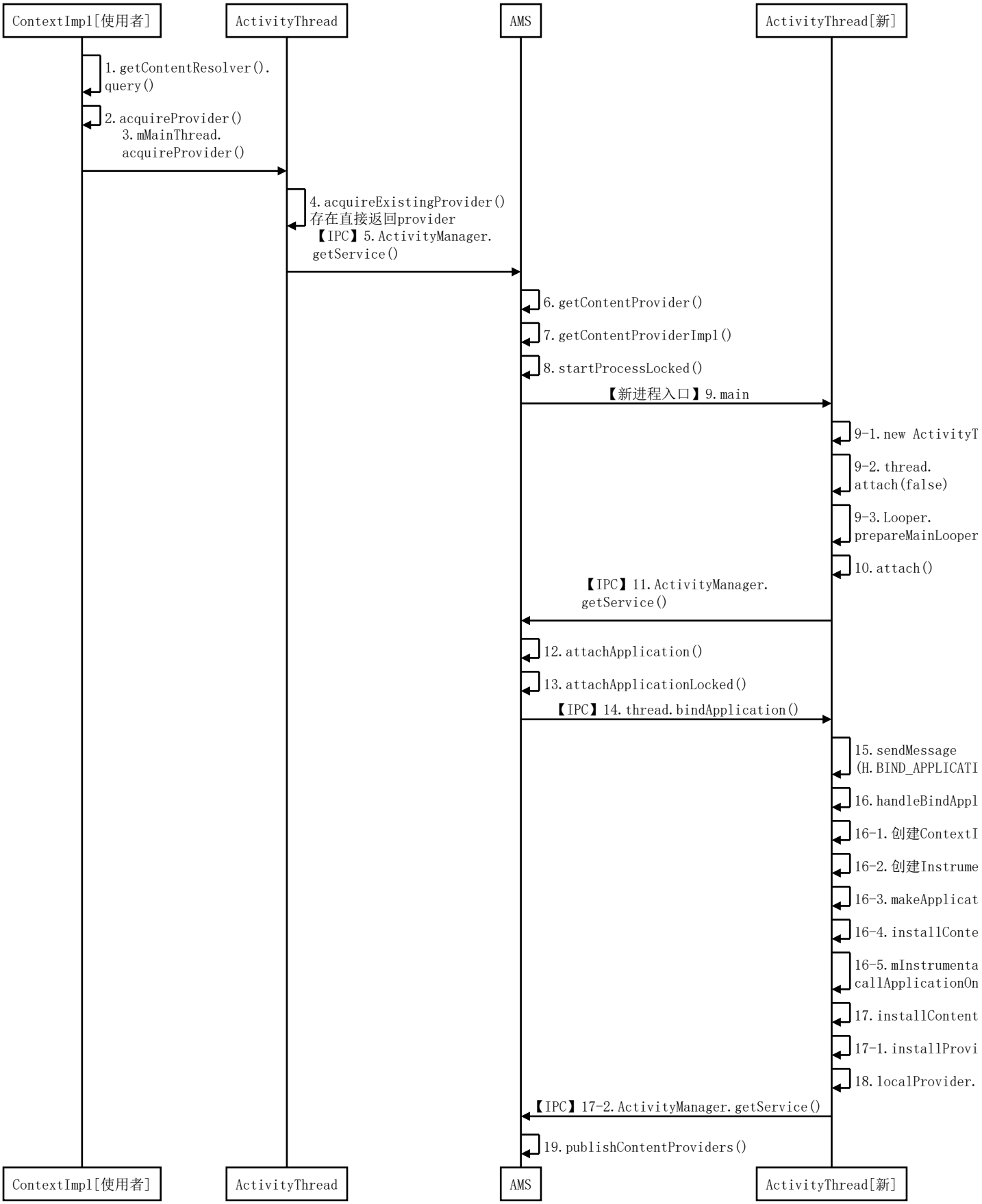
> 1.动态注册从ContextWrapper开始，之后直接交给ContextImpl完成
> 3.已有，从mPackageInfo获取IIntentReceiver对象
> 4.没有则新建IIntentReceiver对象，本质是为了IPC通信需要进行中转，ReceiverDispatcher中同时保存了 BroadcastReceiver和InnerReceiver
> 7.存储远程的InnerReceiver对象(本地的BroadcastReceiver对应的对象)

21、广播的发送和接收



> 5.默认FLAG_EXCLUDE_STOPPED_PACKAGES-广播不会发送给已经停止的应用
> 4-4.BroadcastQueue就会将广播发送给相应的广播接收者
> 6.接收消息并且处理
> 6-1.取出无序广播列表中的广播
> 8.通过 `InnerReceiver` 实现广播的接收,内部会调用ReceiverDispatcher的performReceive方法
> 10.通过Hanlder H的post方法来执行args中的逻辑
> 11.LoadedApk.java内部类ReceiverDispatcher的内部类Args，主要是执行BroadcastReceiver的接收方法

23、ContentProvider的机制



1.获得ContextImpl的内部类：ApplicationContentResolver
5.不存在ContentProvider让MAS启动需要的ContentProvider
8.通过Process的start方法来完成新进程的启动
9-1.首先会创建ActivityThread实例
9-2.然后调用attach-进行一系列初始化
9-3.然后开始消息循环
12.将 `ApplicationThread` 传输给AMS
15. 发送消息给Handler H

16. 完成了Application的创建以及ContentProvider的创建
16-3. makeApplication()创建Application对象
16-4. 启动ContentProvider并调用onCreate方法
16-5. 调用Application的onCreate方法
17-1. 遍历当前进程的Provider列表并调用installProvider()
18.创建ContentProvider对象,并调用onCreate方法
19.将已经启动的ContentProvider保存在AMS的ProviderMap中，外部调用者就可以直接从AMS中获取ContentProvider