Activity的工作过程

显示启动一个Activity非常简单

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

Activity的startActivity方法有好几种重载方式，但最后都调用startActivityForResult方法:

```
public void startActivityForResult(Intent intent, int requestCode,
@Nullable Bundle options) {
    if (mParent == null) {
        Instrumentation.ActivityResult ar =
mInstrumentation.execStartActivity(this,
mMainThread.getApplicationThread(), mToken, this, intent,
requestCode, options);
        if (ar != null) {
            mMainThread.sendActivityResult(mToken, mEmbeddedID,
requestCode, ar.getResultCode(), ar.getResultData());
        }
        if (requestCode >= 0) {
            mStartedActivity = true;
        }

        final View decor = mWindow != null ?
mWindow.peekDecorView() : null;
        if (decor != null) {
            decor.cancelPendingInputEvents();
        }
    }
}
```

mMainThread.getApplicationThread()返回的是ApplicationThread，是ActivityThread的一个内部类，也是一个Binder对象，用于跟ActivityManagerService(后面简称AMS)通信。接着看Instrumentation的execStartActivity方法:

```
public ActivityResult execStartActivity(Context who,IBinder

contextThread, IBinder token, Activity target, Intent intent, int
requestCode, Bundle options) {
    ...
    IApplicationThread whoThread = (IApplicationThread)
contextThread;
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess();
        int result =
ActivityManagerNative.getDefault().startActivity(whoThread,
who.getBasePackageName(), intent,
intent.resolveTypeIfNeed(who.getContentResolver()), token, target
!= null ? target.mEmbeddedID : null, requestCode, 0, null,
options);

        //Activity没有在AndroidManifest中注册时，这里会抛出异常
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {

    }
}
```

可以看出，启动Activity的真正实现由ActivityManagerNative.getDefault()的startActivity方法来完成。而AMS继承自ActivityManagerNative，ActivityManagerNative继承Binder并实现了IActivityManager这个Binder接口，因此AMS也是一个Binder。 初始化AMS这个Binder对象：

```
static public IActivityManager getDefault() {
    return gDefault.get();
}

private static final Singleton<IActivityManager> gDefault = new
Singleton<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");
        IActivityManager am = asInterface(b);
        return am;
    }
}
```

AMS的startActivity经过一系列调用，最终会转移到ActivityStackSupervisor的realStartActivityLocked方法中：

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

System.identityHashCode(r), r.info, new
Configuration(mService.mConfiguration), r.compat,
r.task.voiceInteractor, app.repProcState, r.icicle,
r.persistentState, results, newIntents, !andResume,
mService.isNextTransitionForward(), profilerInfo);
```

中间调用关系如下图：

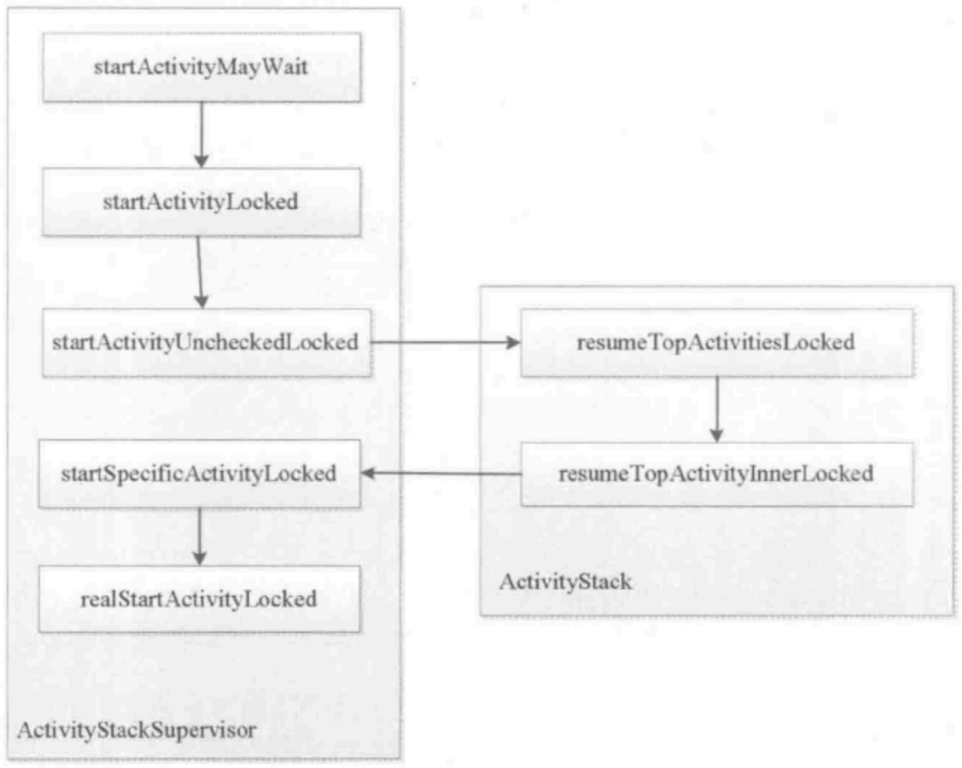
startActivityMayWait
startActivityLocked
startActivityUncheckedLocked
resumeTopActivitiesLocked
startSpecificActivityLocked
resumeTopActivityInnerLocked
realStartActivityLocked
ActivityStack
ActivityStackSupervisor

图 9-1 Activity 的启动过程在 ActivityStackSupervisor 和 ActivityStack 之间的传递顺序

app.thread的类型为IApplicationThread，IApplicationThread继承了IInterface接口，是一个Binder类型的接口。而IApplicationThread的实现者就是ActivityThread的内部类ApplicationThread。

```
private class ApplicationThread extends ApplicationThreadNative
public abstract class ApplicationThreadNative extends Binder
implements IApplicationThread
```

饶了一大圈，又回到了ApplicationThread的scheduleLaunchActivity方法中,方法内的实现很简单，就是发送一个启动Activity的消息给ActivityThread的内部Handler——H。

```
...
sendMessage(H.LAUNCH_ACTIVITY, r);
```

H收到LAUNCH_ACTIVITY消息后，最终会调用ActivityThread的handleLaunchActivity

```
private void handleLaunchActivity(ActivityClientRecord r, Intent
customIntent) {
    ...
    Activity a = performLaunchActivity(r, customIntent);

    if (a != null) {
        r.createdConfig = new Configuration(mConfiguration);
        Bundle oldState = r.state;
        handleResumeActivity(r.token, false, r.isForward,
!r.activity.mFinished && !r.startsNotResumed);
    }
    ...
}
```

可以看出performLaunchActivity完成了Activity的创建和启动过程，并且通过handleResumeActivity调用被启动Activity的onResume生命周期方法。
performLaunchActivity方法主要完成以下几件事：

1. 从ActivityClientRecord中获取待启动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);
}

ComponentName component = r.intent.getComponent();
if (component == null) {
    component =
r.intent.resolveActivity(mInitialApplication.getPackageManager());
    r.intent.setComponent(component);
}

if (r.activityInfo.targetActivity != null) {
    component = new ComponentName(r.activityInfo.packageName,
r.activityInfo.targetActivity);
}
```

2. 通过Instrumentations的newActivity方法使用类加载器创建Activity对象

```
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = appContext.getClassLoader();
        activity = mInstrumentation.newActivity(cl,
component.getClassName(), r.intent);

StrictMode.incrementExpectedActivityCount(activity.getClass());
        r.intent.setExtrasClassLoader(cl);
        r.intent.prepareToEnterProcess();
        if (r.state != null) {
            r.state.setClassLoader(cl);
        }
    } catch (Exception e) {
        if (!mInstrumentation.onException(activity, e)) {
            throw new RuntimeException("Unable to instantiate
activity " + component + ": " + e.toString(), e);
        }
    }
```

Instrumentation的newActivity比较简单，就是通过类加载器创建Activity对象

```
public Activity newActivity(ClassLoader cl, String className,

Intent intent) throws InstantiationException,
IllegarAccessException, ClassNotFoundException {
    return (Activity) cl.loadClass(className).newInstance();
}
```

3. 通过LoadedApk的makeApplication方法来创建Application对象

```
public Applicaiton makeApplication(boolean forceDefaultAppClass,
Instrumentation instrumentation) {
    if (mApplication != null) {
        return mApplication;
    }

    Application app = null;
    String appClass = mApplication.className;
    if (forceDefaultAppClass || (appClass == null)) {
        appClass = "android.app.Application";
    }

    try {
        java.lang.ClassLoader cl = getClassLoader();
        if (!mPackageName.equals("android")) {
            initializeJavaContextClassLoader();
        }
        ContextImpl appContext =
ContextImpl.createAppContext(mActivityThread, this);
        app = mActivityThread.mInstrumentation.newApplication(cl,
appClass, appContext);
        appContext.setOuterContext(app);
    } catch (Exception e) {
        ...
    }
    mActivityThread.mAllApplications.add(app);
    mApplication = app;

    if (instrumentation != null) {
        try {
            instrumentation.callApplicationOnCreate(app);
        } catch (Exception e) {

        }
    }
    ...
    return app;
}
```

4. 创建ContextImpl对象并通过Acitivity的attach方法来完成一些重要数据的初始化

```
Context appContext = createBaseContextForActivity(r, activity);

CharSequence title =
r.activityInfo.loadLabel(appContext.getPackageManager());
Configuration config = new Configuration(mCompatConfiguration);
activity.attach(appContext, this, getInstrumentation(), r.token,
r.ident, app, r.intent, r.activityInfo, title, r.parent,
r.embeddedID, r.lastNonConfigurationInstance, config,
r.voiceInteractor);
```

ContextImpl是一个很重要的数据结构，是Context的具体实现，Context中的大部分逻辑都是由ContextImpl来完成的。ContextImpl是通过Activity的attach方法来和Activity建立关联的。另外，attach 方法中Activity还会完成Window 的创建并建立自己和Window的关联，这样当Window接收到外部输入事件后，就可以将事件传递给Activity。

5. 调用Activity的onCreate方法 mInstrumentation.callActivityOnCreate(activity, r.state)，由于Activity的onCreate已经被调用，这也意外着Activity已经完成了整个启动过程。

Service的工作过程

Service分为两种工作状态，一种是启动状态，主要用于执行后台计算；另一种是绑定状态，主要用于其他组件和Service交互。两种状态可以共存。通过Context的startService可以启动一个Service。

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

Service的启动过程

Service的启动从ContextWrapper的startActivity开始，如下所示：

```
public ComponentName startService(Intent service) {
    return mBase.startService(service);
}
```

上面代码的mBase的类型是ContextImpl，Activity被创建时会通过attach方法将一个ContextImpl对象关联起来，这个ContextImpl对象就是上述代码中的mBase。下面看ContextImpl的startActivity的实现：

```
public ComponentName startService(Intent service) {
    warnIfCallingFromSystemProcess();
    return startServiceCommon(service, mUser);
}

private ComponentName startServiceCommon(Intent service, UserHandle
user) {
    try {
        validateServiceIntent(service);
        service.prepareToLeaveProcess();
        ComponentName cn =
ActivityManagerNative.getDefault().startService(mMainThread.getAppl
icationThread(), service,
service.resolveTypeIfNeeded(getContentResolver()),
user.getIdentifier());
        ...
        return cn;
    } catch (RemoteException e) {
        return null;
    }
}
```

在ContextImpl的startService方法会通过ActivityManagerNative.getDefault()对象来启动服务，而ActivityManagerNative就是AMS。

```
public ComponentName startService(IApplicationThread caller, Intent
service, String resolvedType, int userId) {
    ...
    Synchronized(this) {
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        ComponentName res = mServices.startServiceLocked(caller,
service, resolvedType, callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }

}
```

ASM会通过mServices对象来完成Service后续的启动过程，mServices对象的类型是ActiveServices,ActiveServices是一个辅助AMS进行Service管理的类，包括Service的启动、绑定和停止等。在ActiveServices的startServiceLocked方法尾部会调用startServiceInnerLocked。

```
ComponentName startServiceInnerLocked(ServiceMap smap, Intent
service, ServiceRecord r, boolean callerFg, boolean addToStarting)
{
    ...
    r.callStart = false;
    Synchronized (r.stats.getBatteryStats()) {
        r.stats.startRunningLocked();
    }
    String error = bringUpServiceLocked(r, service.getFlags(),
callerFg, false);
    if (error != null) {
        return new ComonentName("!!", error);
    }
    ...
}
```

ServiceRecord是一个Service记录，贯穿着整个Service的启动过程。bringUpServiceLocked方法会调用realStartServiceLocked。

}

```
private final void realStartServiceLocked(ServiceRecord r,

ProcessRecord app, boolean execInFg) throws RemoteException {
    ...
    boolean created = false;
    try {
        String nameTerm;
        int lastPeriod = r.shotName.lastIndexOf('.');
        nameTerm = lastPeriod >= 0 ?
r.shortName.subString(lastPeriod) : r.shortName;
        ...
        synchronized (r.stats.getBatteryStats()) {
            r.stats.startLaunchedLocked();
        }
        mAm.ensurePackageDexOpt(r.serviceInfo.packageName);

app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
        app.thread.scheduleCreateService(r, r.serviceInfo,
mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo
), app.repProcState);
        r.postNotification();
        created = true;
    } catch (DeadObjectException e) {
        mAm.appDiedLocked(app);
    } finally {
        if (!created) {
            app.services.remove(r);
            r.app = null;
            scheduleServiceRestartLocked(r, false);
            return;
        }
    }

    requestServiceBindingsLocked(r, execInFg);

    updateServiceClientActivitiesLocked(app, null, true);

    if (r.startRequested && r.callStart && r.pendingStarts.size()
== 0) {
        r.pendingStarts.add(new ServiceRecord.StartItem(r, false,
r.makeNextStartId(), null, null));
    }
    sendServiceArgsLocked(r, execInFg, true);
    ...
}
```

在realStartServiceLocked方法中，首先通过app.thread的scheduleCreateService方法来创建Service对象并调用其onCreate,接着再通过sendServiceArgsLocked方法来调用Servcie的其他方法，比如onStartCommand,这两个过程都是IPC。app.thread对象是IApplicationThread类型，它实际上是一个Binder，具体实现是ApplicationThread和ApplicationThreadNative。由于ApplicationThread继承了ApplicationThreadNative,因此只需要看ApplicationThread对Service启动过程的处理即可。

```
public final void scheduleCreateService(IBinder token, ServiceInfo
info, CompatibilityInfo compatInfo, int processState) {
    updateProcessState(processState, false);
    CreateServiceData s = new CreateServiceData();
    s.token = token;
    s.info = info;
    s.compatInfo = compatInfo;

    sendMessage(H.CREATE_SERVICE, s);
}
```

很显然，这个过程和Activity的启动过程类似，都是发送消息给Handler H来处理。H接收CREATE_SERVICE消息并通过AcitivtyThread的handleCreateService方法来完成Service的最终启动。

```
private void handleCreateService(CreateServiceData data) {
    unscheduleGcIdler();

    LoadedApk packageInfo =
getPackageInfoNoCheck(data.info.applicationInfo, data.compatInfo);
    Service service = null;
    try {
        java.lang.ClassLoader cl = packageInfo.getClassLoader();
        service = (Service)
cl.loadClass(data.info.name).newInstance();
    } catch (Exception e) {
        if (!mInstrumentation.onException(service, e)) {
            throw new RuntimeException("Unable to instantiate
servcie " + data.info.name + " : " + e.toString(), e);
        }
    }

    try {
        ContextImpl context = ContextImpl.createAppContext(this,
packageInfo);
        context.setOuterContext(service);

        Application app = packageInfo.makeApplication(false,
mInstrumentation);
        service.attach(context, this, data.info.name, data.token,
app, ActivityManagerNative.getDefault());
        service.onCreate();
        mServices.put(data.token, service);
        try {

ActivityManagerNative.getDefault().serviceDoneExecuting(data.token,
0, 0, 0);
        } catch (RemoteException e) {

        }
    } catch (Exception e) {

    }
}
```

handleCreateService主要完成了如下几件事:
首先通过类加载器创建Service的实例。
然后创建ContextImpl对象并通过Service的attach方法建立二者之间的关系，这个过程和Activity实际上是类似的，毕竟Service和Activity都是一个Context。
最后调用Service的onCreate方法并将Service对象存储到ActivityThread中的一个列表中。列表定义如下 `final ArrayMap<IBinder, Service> mServices = new ArrayMap<IBinder, Service>();` 。
由于Service的onCreate方法被执行了，这也意味着Service已经启动了，除此之外，ActivityThread中还会通过handleServiceArgs方法调用Service的onStartCommand方法。

```java
private void handleServiceArgs(ServiceArgsData data) {
    Service s = mServices.get(data.token);
    if (s != null) {
        try {
            if (data.args != null) {
                data.args.setExtrasClassLoader(s.getClassLoader());
                data.args.prepareToEnterProcess();
            }
            int res;
            if (!data.taskRemoved) {
                res = s.onStartCommand(data.args, data.flags, 
data.startId);
            } else {
                s.onTaskRemoved(data.args);
                res = Service.START_TASK_REMOVED_COMPLETE;
            }

            QueuedWork.waitToFinish();

            try {
                
ActivityManagerNative.getDefault().serviceDoneExecuting(data.token, 
1, data.startId, res);
            } catch (RemoteException e) {

            }
        }
    }
}
```

到这里，Service的启动过程已经分析完了，下面分析Service的绑定过程。

Service的绑定过程

和Service的启动过程一样，Service的绑定过程也是从ContextWrapper开始

```java
public boolean bindService(Intent service, ServiceConnection conn,
int flags) {
    return mBase.bindService(service, conn, flags);
}
```

mBase同样是ContextImpl类型的对象，ContextImpl的bindService方法最终会调用自己的bindServiceCommon方法。

```
private boolean bindServiceCommon(Intent service, ServiceConnection

conn, int flags, UserHandle user) {
    IServiceConnection sd;
    if (conn == null) {
        throw new IllegalArgumentException("connection is null");
    }
    if (mPackageInfo != null) {
        sd = mPackageInfo.getServiceDispatcher(conn,
getOuterContext(), mMainThread.getHandler(), flags);
    } else {
        throw new RuntimeException("Not supported in system
context");
    }
    validateServiceIntent(service);
    try {
        IBinder token = getActivityToken();
        if (token == null && (flags&BIND_AUTO_CREATE) == 0 &&
mPackageInfo != null &&
mPackageInfo.getApplicationInfo().targetSdkVersion <
android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
            flags |= BIND_WAIVE_PRIORITY;
        }
        service.prepareToLeaveProcess();
        int res =
ActivityManagerNative.getDefault().bindService(mMainThread.getAppli
cationThread(), getActivityToken(), service,
service.resolveTypeIfNeeded(getContentResolver()), sd, flags,
user.getIdentifier());
        if (res < 0) {
            throw new SecurityException("Not allowed to bind to
service " + service);
        }
        return res != 0;
    } catch (RemoteException e) {
        return false;
    }
}
```

bindServiceCommon主要完成两件事： 首先将客户端的ServiceConnection对象转化为ServiceDispatcher.InnerConnection对象。之所以不能直接使用ServiceConnection对象，因为服务的绑定可能是跨进程的，因此ServiceConnection对象必须借助Binder才能让远程服务端回调自己的方法，而ServiceDispatcher的内部类InnerConnection刚好充当了Binder角色。其实ServiceDispatcher起着连接ServiceConnection和InnerConnection的作用。这个过程由LoadedApk的getServiceDispatcher方法来完成。

```
public final IServiceConnection

getServiceDispatcher(ServiceConnection c, Context context, Handler
handler, int flags) {
    synchronized (mServices) {
        LoadedApk.ServiceDispatcher sd = null;
        ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>
map = mServices.get(context);
        if (map != null) {
            sd = map.get(c);
        }
        if (sd == null) {
            sd = new ArrayMap<ServiceConnection,
LoadedApk.ServiceDispatcher>();
            mServices.put(context, map);
        } else {
            sd.validate(context, handler);
        }
        return sd.getIServiceConnection();
    }
}
```

在上面的代码中，mServices是一个ArrayMap，它存储了一个应用当前活动的ServiceConnection和ServiceDispatcher的映射关系。定义如下

```
private final ArrayMap<Context, ArrayMap<ServiceConnection,
LoadedApk.ServiceDispatcher>> mServices = new ArrayMap<>();
```

系统首先会查找是否存在相同的ServiceConnection，如果不存在就重新创建一个ServiceDispatcher对象并将其存储在mServices中，其中映射关系的key是ServiceConnection，value是ServiceDispatcher，在ServiceDispatcher内部又保存了ServiceConnection和InnerConnection对象。当Service和客户端建立连接后，系统会通过InnerConnection来调用ServiceConnection中的onServiceConnected方法，这个过程有可能是跨进程的。当ServiceDispatcher创建好了以后，getServiceDispatcher会返回其保存的InnerConnection对象。 接着bindServiceCommon方法会通过AMS来完成Service的具体的绑定过程，这对应于AMS的bindService方法。

```
public int bindService(IApplicationThread caller, IBinder token,

Intent Service, String resolvedType, IServiceConnection connection,
int flags, int userId) {
    enforceNotIsolatedCaller("bindService");
    if (service != null && service.hasFileDescriptions() == true) {
        throw new IllegalArgumentException("File descriptors passed
in Intent");
    }

    synchronized(this) {
        return  mServices.bindServiceLocked(caller, token, service,
resolvedType, connection, flags, userId);
    }
}
```

接下来，AMS会调用ActiveServices的bindServiceLocked方法，bindServiceLocked再调用bringUpServiceLocked，bringUpServiceLocked又会调用realStartServiceLocked方法。和Service的启动过程类似，最终都是通过ApplicationThread来完成Service实例的创建并执行其onCreate方法。不同的是，Service的绑定过程会调用app.thread的scheduleBindService方法。这个过程的实现在ActiveServices的requestServiceBindingLocked方法中。

```
private final boolean requestServiceBindingLocked(ServiceRecord r,

IntentBindRecord i, boolean execInFg, boolean rebind) {
    if (r.app == null || r.app.thread == null) {
        return false;
    }
    if ((!i.requested || rebind) && i.apps.size() > 0) {
        try {
            bumpServiceExecutingLocked(r, execInFg, "bind");

r.app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
            r.app.thread.scheduleBindService(r,
i.intent.getIntent(), rebind, r.app.repProcState);
            if (!rebind) {
                i.requested = true;
            }
            i.hasBound = true;
            i.doRebind = false;
        } catch (RemoteException e) {
            return false;
        }
    }
    return true;
}
```

app.thread对象多次出现，它实际上就是ApplicationThread。ApplicationThread的一系列以schedule开头的方法，内部都是通过Handler H来中转的，对于scheduleBindService也是如此。

```
public final void scheduleBindService(IBinder token, Intent intent,
boolean rebind, int processState) {
    updateProcessState(processState, false);
    BindServiceData s = new BindServiceData();
    s.token = token;
    s.intent = intent;
    s.rebind = rebind;

    sendMessage(H.BIND_SERVICE, s);
}
```

Handler H接收到BIND_SERVICE消息，会交给ActivityThread的handleBindService来处理。在HandleBindService中，首先根据Service的token取出Service对象，然后调用Service的onBind方法，Service的onBind 方法会返回一个Binder对象给客户端使用，这个过程我们在Service的开发过程中都比较熟悉了。原则上来说，Service的onBind方法被调用后，Service就处于绑定状态了，但是onBind是Service的方法，这个时候客户端并不知道Service已经成功连接了，所以还必须调用客户端的ServiceConnection的onServiceConnected，这个过程是由ActivityManagerNative.getDefault()的publishService方法来完成的。
handleBindService实现如下：

```
private void handleBindService(BindServiceData data) {
    Service s = mServices.get(data.token);
    if (s != null) {
        try {
            data.intent.setExtrasClassLoader(s.getClassLoader());
            data.intent.prepareToEnterProcess();
            try {
                if (!data.rebind) {
                    IBinder binder = s.onBind(data.intent);

ActivityManagerNative.getDefault().publishService(data.token,
data.intent, binder);
                } else {
                    s.onRebind(data.intent);

ActivityManagerNative.getDefault().serviceDoneExecuting(data.token,
0, 0, 0);
                }
                ensureJitEnabled();
            } catch(RemoteException e) {

            }
        } catch (Exceptio e) {

        }
    }
}
```

Service有一个特性，当多次绑定同一个Service是，Service的onBind方法只会执行一次，除非Service终止了。当Service的onBind执行以后，系统还要告知客户端已经成功连接Service了。这个过程由AMS的publishService方法来完成。

```
public void publishService(IBinder token, Intent intent, IBinder
service) {
    ...
    synchronized(this) {
        if (!(token instanceof ServiceRecord) {
            throw new IllegalArgumentException("Invalid service
token");
        }
        mServices.publishServiceLocked((ServiceRecord)token,
intent, service);
    }
}
```

从上面代码可以看出，AMS的publishService方法将具体的工作交给了ActiveServices类型的mServices对象来处理。ActiveServices的publishServiceLocked方法看起来很复杂，但核心代码就只有一句 `c.conn.connected(r.name, service)` ，其中c是ConnectionRecord，c.conn是ServiceDispatcher.InnerConnection,service就是Service的onBind方法返回的Binder对象。下面是ServiceDispatcher的InnerConnection定义：

```
private static class InnerConnection extends
IServiceConnection.Stub {
    final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;

    InnerConnection(LoadedApk.ServiceDispatcher sd) {
        mDispatcher = new
WeakReference<LoadedApk.ServiceDispatcher>(sd);
    }

    public void connected(ComponentName, IBinder service) throws
RemoteException {
        LoadedApk.ServiceDispatcher sd = mDispatcher.get();
        if (sd != null) {
            sd.connected(name, service);
        }
    }
}
```

从InnerConnection的定义可以看出，它的connected方法又调用了ServiceDispatcher的connected方法，ServiceDispatcher的connected方法的实现如下

```
public void connected(ComponentName name, IBinder service) {
    if (mActivityThread != null) {
        mActivityThread.post(new RunConnection(name, service, 0));
    } else {
        doConnected(name, service);
    }
}
```

对于Service的绑定过程来说，ServiceDispatcher的mActivityThread是一个Handler，其实它就是AcitivityThread中的H，从前面ServiceDispatcher的创建过程来说，mActivityThread不会为null，这样，RunConnection就可由H的post方法从而运行在主线程，因此客户端ServiceConnection的方法是在主线程被回调的。RunConnection定义如下：

```
private final class RunConnection implements Runnable {
    RunConnection(ComponentName name, IBinder service, int command)
{
        mName = name;
        mService = service;
        mCommand = command;
    }

    public void run() {
        if (mCommand == 0) {
            doConnected(mName, mService);
        } else if (mCommand == 1) {
            doDeath(mName, mService);
        }
    }

    final ComponentName mName;
    final IBinder mService;
    final int mCommand;
}
```

很显然，RunConnection的run方法也是简单调用ServiceDispatcher的doConnected方法，由于ServiceDispatcher内部保存了客户端的ServiceConnection对象，因此它可以很方便地调用ServiceConnection对象的onServiceConnected方法：

```
if (service != null) {
    mConnection.onServiceConnected(name, service);
}
```

客户端的onServiceConnected方法执行后，Service的绑定过程就完成了。

BroadcastReceiver的工作过程

广播的注册过程

广播的注册分为静态注册和动态注册，静态注册在应用安装时由系统自动完成注册，具体来说是由PMS(PackageManagerService)来完成整个注册过程的，其他三大组件也是在应用安装时由PMS解析并注册的。这里只分析动态注册过程，也是从ContextWrapper的registerReceiver方法开始，ContextWrapper没有做实际工作，直接交给ContextImpl来完成。

```
public Intent registerReceiver(BroadcastReceiver receiver,
IntentFilter filter) {
    return mBase.registerReceiver(receiver, filter);
}
```

ContextImpl的registerReceiver方法调用了自己的registerInternal方法

```
private Intent registerReceiverInternal(BroadcastReceiver receiver,
int userId,
            IntentFilter filter, String broadcastPermission,
            Handler scheduler, Context context, int flags) {
        IIntentReceiver rd = null;
        if (receiver != null) {
            if (mPackageInfo != null && context != null) {
                if (scheduler == null) {
                    scheduler = mMainThread.getHandler();
                }
                rd = mPackageInfo.getReceiverDispatcher(
                    receiver, context, scheduler,
                    mMainThread.getInstrumentation(), true);
            } else {
                if (scheduler == null) {
                    scheduler = mMainThread.getHandler();
                }
                rd = new LoadedApk.ReceiverDispatcher(
                        receiver, context, scheduler, null,
true).getIIntentReceiver();
            }
        }
        try {
            final Intent intent =
ActivityManager.getService().registerReceiver(
                    mMainThread.getApplicationThread(),
mBasePackageName, rd, filter,
                    broadcastPermission, userId, flags);
            if (intent != null) {
                intent.setExtrasClassLoader(getClassLoader());
                intent.prepareToEnterProcess();
            }
            return intent;
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
```

在上面的代码中，首先从PackageInfo获取IIntentReceiver对象，然后再采用跨进程的方式向AMS发送广播注册的请求。之所以采用IIntentReceiver，而不是直接采用BroadcastReceiver，这是因为上述注册过程是一个进程间通信的过程，而BroadcastReceiver是不能直接跨进程传递的。IIntentReceiver是一个Binder接口，它的具体实现是LoadedApk.ReceiverDispatcher.InnerReceiver，ReceiverDispatcher内部同时保存了BroadcastReceiver和InnerReceiver，这样当接收到广播时，ReceiverDispatcher可以方便地调用BroadcastReceiver的onReceive方法。可以发现BroadcastReceiver的这个过程和Service实现原理类似，Service也有一个叫ServiceDispatcher的类，并且内部的InnerConnection也是一个Binder接口。
下面看一下ReceiverDispatcher的getIIntentReceiver的实现

```
public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
Context context, Handler handler, Instrumentation instrumentation,
boolean registered) {
    synchronized(mReceivers) {
        LoadedApk.ReceiverDispatcher rd = null;
        ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>
map = null;
        if (registered) {
            map = mReceivers.get(context);
            if (map != null) {
                rd = map.get(r);
            }
        }
        if (rd == null) {
            rd = new ReceiverDispatcher(r, context, handler,
instrumentation, registered);
            if (registered) {
                if (map == null) {
                    map = new ArrayMap<BroadcastReceiver,
LoadedApk.ReceiverDispatcher>();
                    mReceivers.put(context, map);
                }
                map.put(r, rd);
            }
        } else {
            rd.validate(context, handler);
        }
        rd.mForgotten = false;
        return rd.getIIntentReceiver();
    }
}
```

很显然，getReceiverDispatcher方法重新创建了一个ReceiverDispatcher对象并将其保存的InnerReceiver对象作为返回值返回，其中InnerReceiver对象和BroadcastReceiver都是在ReceiverDispatcher的构造方法中被保存的。 由于注册广播的真正实现是在AMS中，接下来看AMS的registerReceiver方法。会把远程的InnerReceiver对象以及IntentFilter对象存储起来。

```
public Intent registerReceiver(IApplicationThread caller, String
callerPackage, IIntentReceiver receiver, IntentFilter filter,
String permission, int userId) {
    ...
    mRegisteredReceivers.put(receiver.asBinder(), rl);
    BroadcastFilter bf = new BroadcastFilter(filter, rl,
callerPackage, permission, callingUid, userId);
    rl.add(bf);
    mReceiverResolver.addFilter(bf);
    ...
}
```

广播的发送和接收过程

当通过send方法来发送广播时，AMS会查找出匹配的广播接收者并将广播发送给它们处理。广播的发送有几种类型：普通广播、有序广播和粘性广播，有序广播和粘性广播与普通广播有不同的特性，但发送和接收过程类似，这里这分析普通广播的实现。 广播的发送和接收，其本质是一个过程的两个阶段。先从广播的发送说起，ContextWrapper的sendBroadcast也是交由ContextImpl处理。

```
public void sendBroadcast(Intent intent) {
    warnIfCallingFromSystemProcess();
    String resolvedType =
intent.resolveTypeIfNeeded(getContentResolver());
    try {
        intent.prepareToLeaveProcess();

ActivityManagerNative.getDefault().broadcastIntent(mMainThread.getA
pplicationThread(), intent, resolvedType, null,
        Activity.RESULT_OK, null, null, null,
AppOpsManager.OP_NONE, false, fasle, getUserId());
    } catch (RemoteException e) {

    }
}
```

从上面的代码来看，ContextImpl直接向AMS请求发送广播。

```
public final int broadcastIntent(IApplicationThread caller, Intent
intent, String resolvedType, IIntentReceiver resultTo, int
resultCode, String resultData, Bundle map, String
requiredPermission, int appOp, boolean serialized, boolean sticky,
int userId) {
    enforceNotIsolatedCaller("broadcastIntent");
    synchronized(this) {
        intent = verifyBroadcastLocked(intent);

        final ProcessRecord callerApp =
getRecordForAppLocked(caller);
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        int res = broadcastIntentLocked(callerApp, callerApp !=
null ? callerApp.info.packageName : null, intent, resolvedType,
resultTo, resultCode, resultData, map, requiredPermission, appOp,
serialized, sticky, callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }
}
```

broadcastIntent调用了broadcastIntentLocked方法，该方法比较复杂，最开始有如下一行：

```
intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);
```

在Android 5.0中，默认广播不会发给已经停止的应用。在Android 3.1中为Intent添加了两个标记位，分别是FLAG_INCLUDE_STOPPED_PACKAGES和FLAG_EXCLUDE_STOPPED_PACKAGES，用来控制广播是否要对处于停止状态的应用起作用。Android 3.1开始，系统为所有广播默认添加了FLAG_EXCLUDE_STOPPED_PACKAGES标志，为了防止广播无意间启动已经停止运行的应用。如果的确需要调起，这需要为广播的Intent添加FLAG_INCLUDE_STOPPED_PACKAGES标记即可。Android 3.1开始，处于停止状态的应用同样无法接收到开机广播，而在3.1之前，是可以收到开机广播的。

在broadcastIntentLocked内部，会根据intent-filter查找出匹配的广播接收者并经过一系列条件过滤，最终会将满足条件的广播接收者添加到BroadcastQueue中，接着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, callerPackage, callingPid, callingUid, resolvedType,
requiredPermission, appOp, receivers, resultTo, resultCode,
resultData, map, ordered, sticky, false, userId);
}
boolean replaced = replacePending &&
queue.replaceOrderedBroadcastLocked(r);
if (!replaced) {
    queue.enqueueOrderedBroadcastLocked(r);
    queue.scheduleBroadcastsLocked();
}
```

下面看BroadcastQueue中广播的发送过程。

```
public void scheduleBroadcastsLocked() {
    if (mBroadcastsScheduled) {
        return;
    }

mHandler.sendMessage(mHandler.obtainMessage(BROADCAST_INTENT_MSG,
this));
    mBroadcastsScheduled = true;
}
```

BroadcastQueue的scheduleBroadcastsLocked方法并没有立即发送广播，而是发送了一个BROADCAST_INTENT_MSG类型的消息，BroadcastQueue收到消息后会调用processNextBroadcast方法，BroadcastQueue的processNextBroadcast方法对普通广播的处理如下：

```
while (mParallelBroadcasts.size() > 0) {
    r = mParallelBroadcasts.remove(0);
    r.dispatchTime = SystemClock.uptimeMillis();
    r.dispatchClockTime = System.currentTimeMillis();
    final int N = r.receivers.size();
    for (int i = 0; i < N; i++) {
        Object target = r.receivers.get(i);
        deliverToRegisteredReceiverLocked(r,
(BroadcastFilter)target, false);
    }
    addBroadcastToHistoryLocked(r);
}
```

可以看到，无序广播存储在mParallelBroadcasts中，系统会遍历mParallelBroadcasts并将其中的广播发送给它们所有的接收者，具体发送过程是通过deliverToRegisteredReceiverLocked方法来实现的。deliverToRegisteredReceiverLocked方法负责将一个广播发送给一个特定的接收者，它内部调用了performReceiveLocked方法来完成具体的发送过程。

```
performReceiveLocked(filter.receiverList.app,
filter.receiverList.receiver, new Intent(r.intent), r.resultCode,
r.resultData, r.resultExtras, r.ordered, r.initialSticky,
r.userId);
```

performReceiveLocked方法如下。由于接收广播会调起应用程序，因此app.thread不为null，根据前面的分析我们知道这里的app.thread仍然指ApplicationThread。

```
private static void performReceivedLocked(ProcessRecord app,

IIntentReceicer receiver, Intent intent, int resultCode, String
data, Bundle extras, boolean ordered, boolean sticky, int
sendingUser) throws RemoteException {
    if (app != null) {
        if (app.thread != null) {
            app.thread.scheduleRegisteredReceiver(receiver, intent,
resultCode, data, extras, ordered, sticky, sendingUser,
app.repProcState);
        } else {
            throw new RemoteException("app.thread must not be
null");
        }
    } else {
        receiver.performReceive(intent, resultCode, data, extras,
ordered, sticky, sendingUser);
    }
}
```

ApplicationThread的scheduleRegisteredReceiver的实现比较简单，通过InnerReceiver来实现广播的接收。

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver,
Intent intent, int resultCode, String dataStr, Bundle extras,
boolean ordered, boolean sticky, int sendingUser, int processState)
throws RemoteException {
    updateProcessState(processState, false);
    receiver.performReceive(intent, resultCode, dataStr, extras,
ordered, sticky, sendingUser);
}
```

InnerReceiver的performReceive方法会调用LoadedApk.ReceiverDispatcher的performReceive方法。

```
public void performReceive(Intent intent, int resultCode, String

data, Bundle extras, boolean ordered, boolean sticky, int
sendingUser) {
    Args args = new Args(intent, resultCode, data, extras, ordered,
sticky, sendingUser);
    if (!mActivityThread.post(args)) {
        if (mRegistered && ordered) {
            IActivityManager mgr =
ActivityManagerNative.getDefault();
            args.sendFinished(mgr);
        }
    }
}
```

LoadedApk.ReceiveDispatcher的performReceive方法会创建一个Args对象，并通过mActivityThread的post方法来执行Args中的逻辑，而Args实现了Runnable接口。mActivityThread是一个Handler，其实就是ActivityThread中的mH。在Args的run方法有如下几行代码：

```
final BroadcastReceiver receiver = mReceiver;
receiver.setPendingResult(this);
receiver.onReceive(mContext, intent);
```

很显然，这个时候BroadcastReceiver的onReceive方法被执行了，也就是应用已经接收到广播了，同时onReceive方法是在广播接收者的主线程中被调用的。

ContentProvider的工作过程

ContentProvider是一个内容共享型组件，它通过Binder向其他组件乃至其他应用提供数据。当ContentProvider所在的进程启动时，ContentProvider会同时启动并被发布到AMS中。

> ContentProvider的onCreate要先于Application的onCreate而执行。

当一个应用启动是，入口方法为ActivityThread的main方法，main方法是一个静态方法，在main方法中会创建ActivityThread的实例并创建主线程的消息队列，然后在ActivityThread的attach方法中远程调用AMS的attachApplication方法，并将ApplicationThread提供给AMS。ApplicationThread是一个Binder对象，它的Binder接口是IApplicationThread，主要用于ActivityThread和AMS之间的通信。在AMS的attachApplication方法中，会调用ApplicationThread的bindApplication方法，这个同样是跨进程完成的。bindApplication会经过ActivityThread的mH Handler切换到ActivityThread中去执行，具体的方法是handleBindApplication。在handleBindApplication方法中，ActivityThread会创建Application对象并加载ContentProvider。需要注意的是，ActivityThread会先加载ContentProvider，然后再调用Application的onCreate方法。

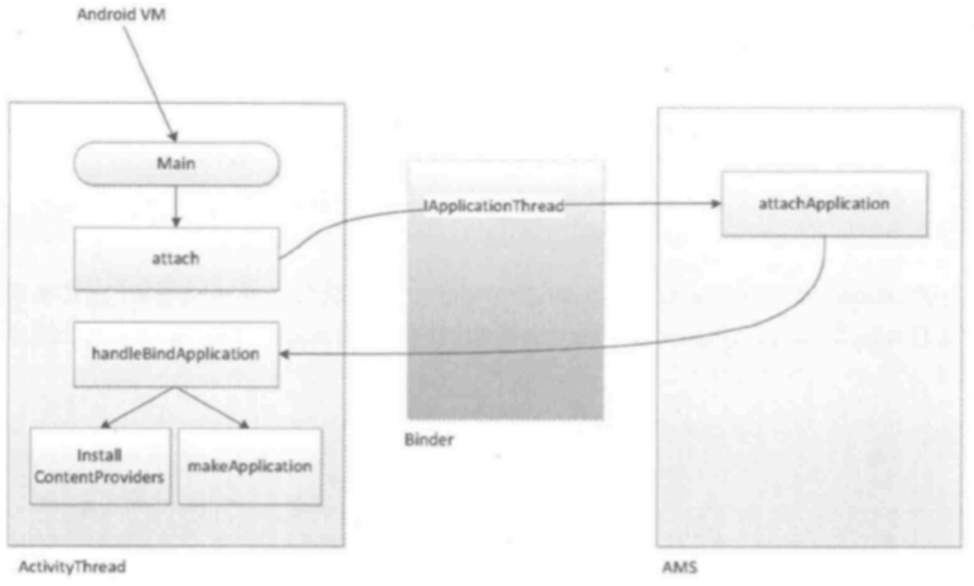

图 9-2 ContentProvider 的启动过程

ContentProvider启动后，外界就可以通过提供的增删改查四个接口来操作ContentProvider的数据源。这四个方法都是通过Binder来调用的。外界通过AMS根据Uri来获取对应的ContentProvider的Binder接口IContentProvider，然后再通过IContentProvider来访问ContentProvider数据源。

ContentProvider一般是单实例的，通过android:multiprocess属性决定，当android:multiprocess为false时，ContentProvider是单实例，也是默认值。当android:multiprocess为true是，ContentProvider为多实例，这个时候在每个调用者的进程中都存在一个ContentProvider对象。多实例的ContentProvider暂无使用价值。

访问ContentProvider需要通过ContentResolver，ContentResolver是一个抽象类，通过Context的getContentResolver方法获取的实际上是ApplicationContentResolver对象，ApplicationContentResolver类继承了ContentResolver并实现了ContentResolver中的抽象方法。当ContentProvider所在的进程未启动时，第一次访问它时就会触发ContentProvider的创建，当然这也伴随着ContentProvider所在进程的启动。通过ContentProvider的四个方法的任何一个都可以触发ContentProvider的启动过程。这里选择query方法。ContentProvider的query方法中，首先会获取IContentProvider对象，不管是acquireUnstableProvider方法还是直接通过acquireProvider方法，它们的本质都是一样，最终通过acquireProvider方法来获取ContentProvider。下面是ApplicationContentResolver的acquireProvider方法的具体实现：

```
protected IContentProvider acquireProvider(Context context, String
auth) {
    return mMainThread.acquireProvider(context,
ContentProvider.getAuthorityWithoutUserId(auth),
    resolveUserIdFromAuthority(auth), true);
}
```

ApplicationContentResolver的acquireProvider方法并没有处理任何逻辑，它直接调用了ActivityThread的acquireProvider方法，ActivityThread的acquireProvider方法的源码如下：

```
public final IContentProvider acquireProvider(Context c, String 
auth, int userId, boolean stable) {
    final IContentProvider provider = acquireExistingProvider(c,
auth, userId, stable);
    if (provider != null) {
        return provider;
    }

    IActivityManager.ContentProviderHolder holder = null;
    try {
        holder = 
ActivityManagerNative.getDefault().getContentProvider(getApplicatio
nThread(), auth, userId, stable);
    } catch (RemoteException e) {

    }

    if (holder == null) {
        return null;
    }

    holder = installProvider(c, holder, holder.info, true,
holder.noReleaseNeeded, stable);
    return holder.provider;
}
```

上面的代码首先会从ActivityThread中查找是否已经存在目标ContentProvider了，如果存在就直接返回。ActivityThread中通过mProviderMap来存储已经启动的ContentProvider对象，mProviderMap的声明如下：

```
final ArrayMap<ProviderKey, ProviderClientRecord> mProviderMap =
new ArrayMap<ProviderKey, ProviderClientRecord>();
```

如果目前ContentProvider没有启动，就发生一个进程间请求给AMS让其启动目标ContentProvider，最好再通过installProvider方法来修改引用计数。那么AMS是如何启动ContentProvider的呢？我们知道，ContentProvider被启动时会伴随着进程的启动。在AMS中，首先会启动ContentProvider所在的进程，然后再启动ContentProvider。启动进程是由AMS的startProcessLocked方法来完成的，其内部主要是通过Process的start方法来完成一个新进程的启动，新进程启动后其入口方法为ActivityThread的main方法，如下方式：

```
public static void main(String[] args) {
    SamplingProfileIntegration.start();

    CloseGuard.setEnabled(false);

    Environment.initForCurrentUser();

    EventLogger.setReporter(new EventLoggingReporter());

    Security.addProvider(new AndroidKeyStoreProvider());

    final File configDir =
Environment.getUserConfigDirectory(UserHandle.myUserId());
    TrustedCertificateStore.setDefaultUserDirectory(configDir);

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

    ActivityThread thread = new ActivityThread();
    thread.attach(false);

    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }

    AsyncTask.init();

    if (false) {
        Looper.myLooper().setMessageLogging(new
LogPrinter(Log.DEBUG, "ActivityThread"));
    }

    Looper.loop();

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

可以看到，ActivityThread的main方法是一个静态方法，在它内部首先会创建ActivityThread的实例并调用attach方法来进行一系列初始化，接着就开始进行消息循环了。ActivityThread的attach方法会将ApplicationThread对象通过AMS的attachApplication方法跨进程传递给AMS，最终AMS会完成ContentProvider的创建过程，源码如下所示：

```
try {
    mgr.attachApplication(mAppThread);
} catch (RemoteException e) {

}
```

AMS的attachApplication方法调用了attachApplicationLocked方法，attachApplicationLocked中又调用了ApplicationThread的bindApplication，注意这个过程也是进程间调用。如下所示：

```
thread.bindApplication(processName, appInfo, providers,
app.instrumentationClass, profilerInfo,
app.instrumentationArguments, app.instrumentationWatcher,
app.instrumentationUiAutomationConnection, testMode,
enableOpenGlTrace,
isRestrictedBackupMode || !normalMode, app.persistent,
new Configuration(mConfiguration), app.compat,
getCommonServicesLocked(),
mCoreSettingsObserver.getCoreSettingLocked());
```

ActivityThread的bindApplication会发送一个BIND_APPLICATION类型的消息给mH，mH是一个Handler，它收到消息后会调用ActivityThread的handleBindApplication方法，bindApplication发送消息的过程如下所示：

```
AppBindData data = new AppBindData();
data.processName = processName;
data.appInfo = appInfo;
data.providers = providers;
data.instrumentationName = instrumentationName;
data.instrumentationArgs = instrumentationArgs;
data.instrumentationWatcher = instrumentationWatcher;
data.instrumentationUiAutomationConnection =
instrumentationUiConnection;
data.debugMode = debugMode;
data.enableOpenGlTrace = enableOpenGlTrace;
data.restrictedBackupMode = isRestrictedBackupMode;
data.persistent = persistent;
data.config = config;
data.compatInfo = compatInfo;
data.initProfilerInfo = profilerInfo;
sendMessage(H.BIND_APPLICATION, data);
```

ActivityThread的handleBindApplication则完成了Application的创建以及ContentProvider的创建，可以分为如下四个步骤：

1. 创建ContextImpl和Intrumentation

```
ContextImpl = instrContext = ContextImpl.createAppContext(this,
pi);

try {
    java.lang.ClassLoader cl = instrContext.getClassLoader();
    mInstrumentation = (Instrumentation)
cl.loadClass(data.instrumentationName.getClassName()).newInstance()
;
} catch(Exception e) {

}

mInstrumentation.init(this, instrContext, appContext,
    new ComponentName(ii.packageName, ii.name),
data.instrumentationWatcher,
data.instrumentationUiAutomationConnection);
```

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

```
List<ProviderInfo> providers = data.providers;
if (provider != null) {
    installContentProvider(app, providers);
    mH.sendEmptyMessageDelayed(H.ENABLE_JIT, 10 * 1000);
}
```

installContentProviders完成了ContentProvider的启动工作，它的实现方式如下所示。首先会遍历当前进程的ProviderInfo的列表并一一调用installProvider方法来启动它们，接着将已启动的ContentProvider发布到AMS中，AMS会把它们存储在ProviderMap中，这样一来外部调用者就可以直接从AMS中获取ContentProvider了。

```java
private void installContentProviders(Context context,
List<ProviderInfo> providers) {
    final ArrayList<IActivityManager.ContentProviderHolder> results
= new ArrayList<IActivityManager.ContentProviderHolder>();

    for (ProviderInfo cpi : providers) {
        IActivityManager.ContentProviderHolder cph =
installProvider(context, null, cpi, false, true, true);

        if (cph != null) {
            cph.noReleaseNeeded = true;
            result.add(cph);
        }
    }

    try {

ActivityManagerNative.getDefault().publishContentProviders(getAppli
cationThread(), results);
    } catch(RemoteException e) {

    }
}
```

下面看一下ContentProvider对象的创建过程，在installProder方法中有如下一段代码，其通过类加载器完成了ContentProvider对象的创建。

```java
final java.lang.ClassLoader cl = c.getClassLoader();
localProvider = (ContentProvider)
cl.loadClass(info.name).newInstance();
provider = localProvider.getIContentProvider();

localProvider.attachInfo(c, info);
```

在上述代码中，除了完成ContentProvider对象的创建，还会通过ContentProvider的attachInfo方法来调用它的onCreate方法，如下所示：

```
private void attachInfo(Context context, ProviderInfo info, boolean
testing) {
    if (mContext == null) {
        mContext = context;
        if (context != null) {
            mTransport.mAppOpsManager = (AppOpsManager)
context.getSystemService(Context.APP_OPS_SERVICE);
        }
        mMyUid = Process.myUid();
        ...
        ContentProvider.this.onCreate();
    }
}
```

到此为止，ContentProvider已经被创建并且其onCreate方法也已经被调用，这意味着ContentProvider已经启动完成了。

4. 调用Application的onCreate方法

```
try {
    mInstrumentation.callApplicationOnCreate(app);
} catch (Exception e) {

}
```

经过上面四个步骤，ContentProvider已经启动，并且其所在进程的Application也已经启动，这意味着ContentProvider所在的进程已经完成了整个的启动过程，然后其他应用就可以通过AMS来访问这个ContentProvider了。拿到ContentProvider以后，就可以通过它所提供的接口来访问它了。需要注意的是，这里的ContentProvider并不是原始的ContentProvider，而是ContentProvider的Binder类型的对象IContentProvider，IContentProvider的具体实现是ContentProviderNative和ContentProvider.Transport，其中ContentProvider.Transport继承了ContentProviderNative。这里仍然选择query方法，首先其他应用会通过AMS获取到ContentProvider的Binder对象即IContentProvider，而IContentProvider的实现者实际上是ContentProvider.Transport。因此其他应用调用IContentProvider的query方法时最终会以进程间的通信方法调用到ContentProvider.Transport的query方法，它的实现如下所示：

```
public Cursor query(String callingPkg, Uri uri, String[]

projection, String selection, String[] selectionArgs, String
sortOrder, ICancellationSignal cancellationSignal) {
    validateIncomingUri(uri);
    uri = getUriWithoutUserId(uri);
    if (enforceReadPermission(callingPkg, uri) !=
AppOpsManager.MODE_ALLOWED) {
        return rejectQuery(uri, projection, selection,
selectionArgs, sortOrder,
CancellationSignal.fromTransport(cancellationSignal));
    }
    final String original = setCallingPackage(callingPkg);
    try {
        return ContentProvider.this.query(uri, projection,
selection, selectionArgs, sortOrder,
CancellationSignal.fromTransport(cancellationSignal));
    } final {
        setCallingPackage(original);
    }
}
```

很显然，ContentProvider.Transport的query方法调用了ContentProvider的query方法，query方法的执行结果再通过Binder返回给调用者，这样一来整个调用过程就完成了。除了query方法，insert、delete和update也是类似的。