

基于 2.3 *Gingerbread* 版本，
深入剖析Android内核



Android 内核剖析

◎柯元旦 著◎



第 5 章 Binder

Binder，英文的意思是别针、回形针。我们经常用别针把两张纸“别”在一起，而在 Android 中，Binder 用于完成进程间通信（IPC），即把多个进程“别”在一起。比如，普通应用程序可以调用音乐播放服务提供的播放、暂停、停止等功能。

Binder 工作在 Linux 层面，属于一个驱动，只是这个驱动不需要硬件，或者说其操作的硬件是基于一小段内存。从线程的角度来讲，Binder 驱动代码运行在内核态，客户端程序调用 Binder 是通过系统调用完成的。

5.1 Binder 框架

Binder 是一种架构，这种架构提供了服务端接口、Binder 驱动、客户端接口三个模块，如图 5-1 所示。

首先来看服务端。一个 Binder 服务端实际上就是一个 Binder 类的对象，该对象一旦创建，内部就启动一个隐藏线程。该线程接下来会接收 Binder 驱动发送的消息，收到消息后，会执行到 Binder 对象中的 `onTransact()` 函数，并按照该函数的参数执行不同的服务代码。因此，要实现一个 Binder 服务，就必须重载 `onTransact()` 方法。

可以想象，重载 `onTransact()` 函数的主要内容是把 `onTransact()` 函数的参数转换为服务函数的参数，而 `onTransact()` 函数的参数来源是客户端调用 `transact()` 函数时输入的，因此，如果 `transact()` 有固定格式的输入，那么 `onTransact()` 就会有固定格式的输出。

下面再看 Binder 驱动。任意一个服务端 Binder 对象被创建时，同时会在 Binder 驱动中创建一个 `mRemote` 对象，该对象的类型也是 Binder 类。客户端要访问远程服务时，都是通过 `mRemote` 对象。

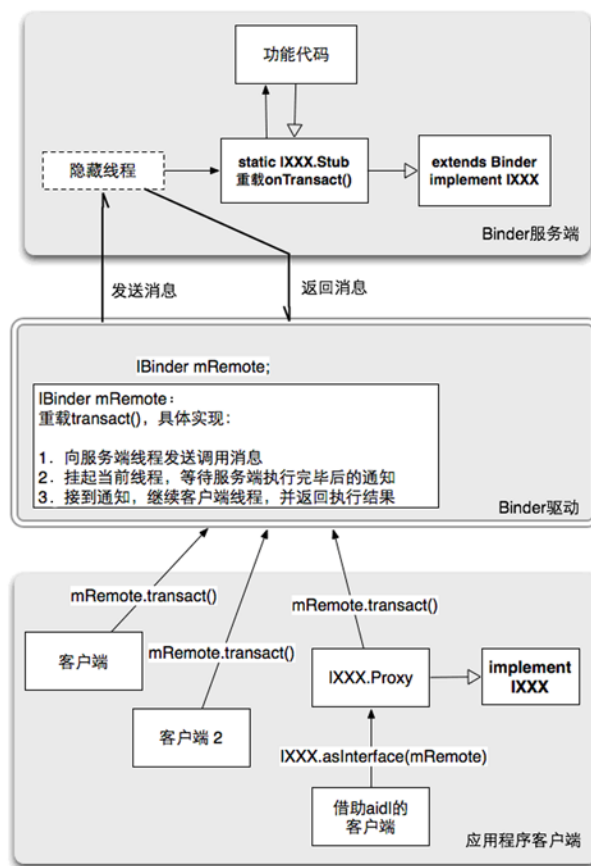


图 5-1 Binder 框架

最后来看应用程序客户端。客户端要想访问远程服务，必须获取远程服务在 Binder 对象中对应的 mRemote 引用，至于如何获取，下面几节将要介绍。获得该 mRemote 对象后，就可以调用其 transact() 方法，而在 Binder 驱动中，mRemote 对象也重载了 transact() 方法，重载的内容主要包括以下几项。

- 以线程间消息通信的模式，向服务端发送客户端传递过来的参数。
- 挂起当前线程，当前线程正是客户端线程，并等待服务端线程执行完指定服务函数后通知 (notify)。
- 接收到服务端线程的通知，然后继续执行客户端线程，并返回到客户端代码区。

从这里可以看出，对应用程序开发人员来讲，客户端似乎是直接调用远程服务对应的 Binder，而事实上则是通过 Binder 驱动进行了中转。即存在两个 Binder 对象，一个是服务端的 Binder 对象，另一个则是 Binder 驱动中的 Binder 对象，所不同的是 Binder 驱动中的对象不会再额外产生一个线程。

5.2 设计 Servier 端

设计 Service 端很简单，从代码的角度来讲，只要基于 Binder 类新建一个 Servier 类即可。以下以设计一个 MusicPlayerService 类为例。

假设该 Service 仅提供两个方法：start(String filePath)和 stop()，那么该类的代码可以如下：

```
public class MusicPlayerService extends Binder{
    @Override
    protected boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
        return super.onTransact(code, data, reply, flags);
    }

    public void start(String filePath){
    }

    public void stop(){
    }
}
```

当要启动该服务时，只需要初始化一个 MusicPlayerService 对象即可。比如可以在主 Activity 里面初始化一个 MusicPlayerService，然后运行，此时可以在 ddms 中发现多了一个线程，如图 5-2 所示。

ID	Tid	Status	utime	stime	Name
1	437	wait	25	30	main
*2	438	vmwait	0	9	HeapWorker
*3	439	vmwait	0	0	Signal Catcher
*4	440	running	5	9	JDWP
5	441	native	0	0	Binder Thread #1
6	442	native	0	0	Binder Thread #2
7	443	native	0	0	Binder Thread #3

图 5-2 使用 ddms 工具查看 Binder 对应的线程

如果不创建 MusicPlayerService，则只有三个 Binder 对象对应的线程。

定义了服务类后，接下来需要重载 onTrasact()方法，并从 data 变量中读出客户端传递的参数，比如 start()方法所需要的 filePath 变量。然而，这里有个问题，服务端如何知道这个参数在 data 变量中的位置？因此，这就需要调用者和服务者双方有个约定。

这里假定客户端在传入的包裹 data 中放入的第一个数据就是 filePath 变量，那么，onTransact()的代码可以如下所示：

```
switch(code){
    case 1000:
        data.enforceInterface("MusicPlayerService");
        String filePath = data.readString();
        start(filePath);
        //replay.writeXXX();
        break;
}
```



code 变量用于标识客户端期望调用服务端的哪个函数，因此，双方需要约定一组 int 值，不同的值代表不同的服务端函数，该值和客户端的 transact() 函数中第一个参数 code 的值是一致的。这里假定 1000 是双方约定要调用 start() 函数的值。

enforceInterface() 是为了某种校验，它与客户端的 writeInterfaceToken() 对应，具体见下一小节。

readString() 用于从包裹中取出一个字符串。取出 filePath 变量后，就可以调用服务端的 start() 函数了。

如果该 IPC 调用的客户端期望返回一些结果，则可以在返回包裹 reply 中调用 Parcel 提供的相关函数写入相应的结果。

5.3 Binder 客户端设计

要想使用服务端，首先要获取服务端在 Binder 驱动中对应的 mRemote 变量的引用，获取的方法后面小节将有介绍。获得该变量的引用后，就可以调用该变量的 transact() 方法。该方法的函数原型如下：

```
public final boolean transact(int code, Parcel data, Parcel reply, int flags)
```

其中 data 表示的是要传递给远程 Binder 服务的包裹(Parcel)，远程服务函数所需要的参数必须放入这个包裹中。包裹中只能放入特定类型的变量，这些类型包括常用的原子类型，比如 String、int、long 等，要查看包裹可以放入的全部数据类型，可以参照 Parcel 类。除了一般的原子变量外，Parcel 还提供了一个 writeParcel() 方法，可以在包裹中包含一个小包裹。因此，要进行 Binder 远程服务调用时，服务函数的参数要么是一个原子类，要么必须继承于 Parcel 类，否则，是不能传递的。

因此，对于 MediaPlayerService 的客户端而言，可以如下调用 transact() 方法。

```
IBinder mRemote = null;
String filePath = "/sdcard/music/heal_the_world.mp3";
int code = 1000;
Parcel data = Parcel.obtain();
Parcel reply = Parcel.obtain();
data.writeInterfaceToken("MediaPlayerService");
data.writeString(filePath);
mRemote.transact(code, data, reply, 0);
IBinder binder = reply.readStrongBinder();
reply.recycle();
data.recycle();
```

现在来分析以上代码。首先，包裹不是客户端自己创建的，而是调用 Parcel.obtain() 申请的，这正如同生活中的邮局一样，用户一般只能用邮局提供的信封（尤其是 EMS）。其中 data 和 reply 变量都由客户端提供，reply 变量用户服务端把返回的结果放入其中。

writeInterfaceToken() 方法标注远程服务名称，理论上讲，这个名称不是必需的，因为客户端既然已经获取指定远程服务的 Binder 引用，那么就不会调用到其他远程服务。该名称将作为 Binder 驱动确保客户端的确想调用指定的服务端。

writeString() 方法用于向包裹中添加一个 String 变量。注意，包裹中添加的内容是有序的，这个顺序必须是客户端和服务端事先约定好的，在服务端的 onTransact() 方法中会按照约定的顺序取出变量。



接着调用 `transact()` 方法。调用该方法后，客户端线程进入 Binder 驱动，Binder 驱动就会挂起当前线程，并向远程服务发送一个消息，消息中包含了客户端传进来的包裹。服务端拿到包裹后，会对包裹进行拆解，然后执行指定的服务函数，执行完毕后，再把执行结果放入客户端提供的 `reply` 包裹中。然后服务端向 Binder 驱动发送一个 `notify` 的消息，从而使得客户端线程从 Binder 驱动代码区返回到客户端代码区。

`transact()` 的最后一个参数的含义是执行 IPC 调用的模式，分为两种：一种是双向，用常量 `0` 表示，其含义是服务端执行完指定服务后会返回一定的数据；另一种是单向，用常量 `1` 表示，其含义是不返回任何数据。

最后，客户端就可以从 `reply` 中解析返回的数据了，同样，返回包裹中包含的数据也必须是有序的，而且这个顺序也必须是服务端和客户端事先约定好的。

5.4 使用 Service 类

以上手工编写 Binder 服务端和客户端的过程存在两个重要问题。

第一，客户端如何获得服务端的 Binder 对象引用。

第二，客户端和服务端必须事先约定好两件事情。

- 服务端函数的参数在包裹中的顺序。
- 服务端不同函数的 `int` 型标识。

关于第一个问题，请思考为什么要用 Binder。答案很简单，即为了提供一个全局服务，所谓的“全局”，是指系统中的任何应用程序都可以访问。很明显，这是一个操作系统应该提供的最基本的功能之一，Android 的工程师自然也是这么认为的，因此，他们提供了一个更傻瓜的解决方法，那就是 `Service`。它是 Android 应用程序四个基本程序片段（Component）之一，四个基本片段包括 `Activity`、`Service`、`Content Provider`、`Receiver`。

无论是否使用 `Service` 类，都必须解决以上两个重要问题。因此，下面先介绍如何解决第一个问题。

5.4.1 获取 Binder 对象

事实上，对于有创造力的程序员来讲，可以完全不使用 `Service` 类，而仅仅基于 `Binder` 类编写服务程序，但只是一部分。具体来讲，可以仅使用 `Binder` 类扩展系统服务，而对于客户端服务则必须基于 `Service` 类来编写。所谓的系统服务是指可以使用 `getSystemService()` 方法获取的服务，所谓的客户端服务是指应用程序提供的自定义服务。

关于 `Service` 的内部机制请参考第 14 章，本章仅指出 `Service` 和 `Binder` 的关系。那么，`Service` 类是如何解决本节开头所提出的两个重要问题的呢？

首先，`AmS` 提供了 `startService()` 函数用于启动客户服务，而对于客户端来讲，可以使用以下两个函数来和一个服务建立连接，其原型在 `android.app.ContextImpl` 类中。



```
public ComponentName startService(Intent intent);
```

该函数用于启动 intent 指定的服务，而启动后，客户端暂时还没有服务端的 Binder 引用，因此，暂时还不能调用任何服务功能。

```
public boolean bindService(Intent service, ServiceConnection conn, int flags);
```

该函数用于绑定一个服务，这就是第一个重要问题的关键所在。其中第二个参数是一个 interface 类，该 interface 的定义如以下代码所示：

```
public interface ServiceConnection {
    public void onServiceConnected(ComponentName name, IBinder service);
    public void onServiceDisconnected(ComponentName name);
}
```

请注意该 interface 中的 onServiceConnected() 方法的第二个变量 Service。当客户端请求 AmS 启动某个 Service 后，该 Service 如果正常启动，那么 AmS 就会远程调用 ActivityThread 类中的 ApplicationThread 对象，调用的参数中会包含 Service 的 Binder 引用，然后在 ApplicationThread 中会回调 bindService 中的 conn 接口。因此，在客户端中，可以在 onServiceConnected() 方法中将其参数 Service 保存为一个全局变量，从而在客户端的任何地方都可以随时调用该远程服务。这就解决了第一个重要问题，即客户端如何获取远程服务的 Binder 引用。

以上流程如图 5-3 所示。

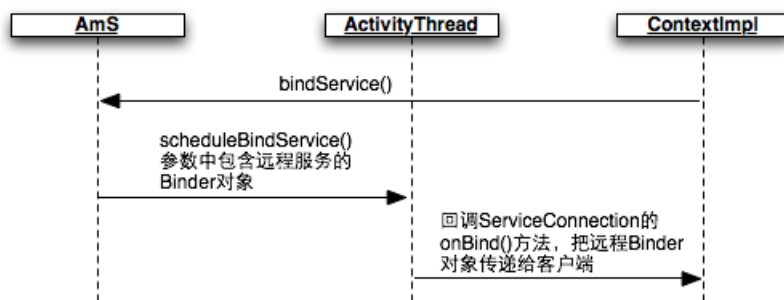


图 5-3 Binder 客户端和服务端的调用过程

5.4.2 保证包裹内参数顺序 aidl 工具的使用

关于第二个问题，Android 的 SDK 中提供了一个 aidl 工具，该工具可以把一个 aidl 文件转换为一个 Java 类文件，在该 Java 类文件，同时重载了 transact 和 onTransact() 方法，统一了存入包裹和读取包裹参数，从而使设计者可以把注意力放到服务代码本身上。

aidl 工具不是必需的，对于有经验的程序员来讲，手工编写一个参数统一的包裹存入和包裹读出代



码并不是一件复杂的事情。

接下来看 `aidl` 工具都做了什么。如本章第一节示例，此处依然假设要编写一个 `MusicPlayerService` 服务，服务中包含两个服务函数，分别是 `start()` 和 `stop()`。那么，可以首先编写一个 `IMusicPlayerService.aidl` 文件。如以下代码所示：

```
package com.haiii.android.client;
interface IMusicPlayerService{
    boolean start(String filePath);
    void stop();
}
```

该文件的名称必须遵循一定的规范，第一个字母“`I`”不是必需的，但是，为了程序风格的统一，“`I`”的含义是 `IInterface` 类，即这是一个可以提供访问远程服务的类。后面的命名——`MusicPlayerService` 对应的是服务的类名，可以是任意的，但是，`aidl` 工具会以该名称命名输出的 `Java` 类。这些规则都只是 `Eclipse` 下 `ADT` 插件的默认规则，`aidl` 本身只是一个命令程序，借助命令行的话，则可以灵活指定输出文件的名称及位置。具体使用方法参照 `aidl` 的执行帮助信息。

`aidl` 文件的语法基本类似于 `Java`，`package` 指定输出后的 `Java` 文件对应的包名。如果该文件需要引用其他 `Java` 类，则可以使用 `import` 关键字，但需要注意的是，包裹内只能写入以下三个类型的内容。

- `Java` 原子类型，如 `int`、`long`、`String` 等变量。
- `Binder` 引用。
- 实现了 `Parcelable` 的对象。

因此，基本上来讲，`import` 所引用的 `Java` 类也只能是以上三个类型。

`interface` 为关键字，有时会在 `interface` 前面加一个 `oneway`，代表该 `service` 提供的方法都是没有返回值的，即都是 `void` 类型。

下面看看该 `aidl` 生成的 `IMusicPlayerService.java` 文件的代码。如下所示：

```
package com.haiii.client;
public interface IMusicPlayerService extends android.os.IInterface
{
    /** Local-side IPC implementation stub class. */
    public static abstract class Stub extends android.os.Binder
        implements com.haiii.client.IMusicPlayerService
    {
        private static final java.lang.String DESCRIPTOR =
            "com.haiii.client.IMusicPlayerService";

        /** Construct the stub at attach it to the interface. */
        public Stub()
        {
            this.attachInterface(this, DESCRIPTOR);
        }

        /**
         * Cast an IBinder object into an com.haiii.client.IMusicPlayerService interface,
```

```

    * generating a proxy if needed.
    */
    public static com.haiii.client.IMusicPlayerService
        asInterface(android.os.IBinder obj)
    {
        if ((obj==null)) {
            return null;
        }

        android.os.IInterface iin =
            (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);

        if (((iin!=null)&&(iin instanceof com.haiii.client.IMusicPlayerService))) {
            return ((com.haiii.client.IMusicPlayerService)iin);
        }
        return new com.haiii.client.IMusicPlayerService.Stub.Proxy(obj);
    }

    public android.os.IBinder asBinder()
    {
        return this;
    }

    @Override
    public boolean onTransact(int code, android.os.Parcel data,
        android.os.Parcel reply, int flags) throws android.os.RemoteException
    {
        switch (code)
        {
            case INTERFACE_TRANSACTION:
            {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_start:
            {
                data.enforceInterface(DESCRIPTOR);
                java.lang.String _arg0;
                _arg0 = data.readString();
                boolean _result = this.start(_arg0);
                reply.writeNoException();
                reply.writeInt((_result)?(1):(0));
                return true;
            }
            case TRANSACTION_stop:
            {
                data.enforceInterface(DESCRIPTOR);
                this.stop();
                reply.writeNoException();
                return true;
            }
        }
    }
}

```



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

    private static class Proxy implements com.haiii.client.IMusicPlayerService
    {
        private android.os.IBinder mRemote;
        Proxy(android.os.IBinder remote)
        {
            mRemote = remote;
        }

        public android.os.IBinder asBinder()
        {
            return mRemote;
        }

        public java.lang.String getInterfaceDescriptor()
        {
            return DESCRIPTOR;
        }

        public boolean start(java.lang.String filePath) throws android.os.RemoteException
        {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            boolean _result;
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                _data.writeString(filePath);
                mRemote.transact(Stub.TRANSACTION_start, _data, _reply, 0);
                _reply.readException();
                _result = (0!=_reply.readInt());
            }
            finally {
                _reply.recycle();
                _data.recycle();
            }
            return _result;
        }

        public void stop() throws android.os.RemoteException
        {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                mRemote.transact(Stub.TRANSACTION_stop, _data, _reply, 0);
                _reply.readException();
            }
            finally {
                _reply.recycle();
                _data.recycle();
            }
        }
    }
}
```

```

    }
    }
}

static final int TRANSACTION_start = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
static final int TRANSACTION_stop = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
}

public boolean start(java.lang.String filePath) throws android.os.RemoteException;
public void stop() throws android.os.RemoteException;
}

```

这些代码主要完成以下三个任务。

- 定义一个 Java interface，内部包含 aidl 文件所声明的服务函数，类名称为 `IMusicPlayerService`，并且该类基于 `IInterface` 接口，即需要提供一个 `asBinder()` 函数。
- 定义一个 `Proxy` 类，该类将作为客户端程序访问服务端的代理。所谓的代理主要就是为了前面所提到的第二个重要问题——统一包裹内写入参数的顺序。
- 定义一个 `Stub` 类，这是一个 `abstract` 类，基于 `Binder` 类，并且实现了 `IMusicPlayerService` 接口，主要由服务端来使用。该类之所以要定义为一个 `abstract` 类，是因为具体的服务函数必须由程序员实现，因此，`IMusicPlayerService` 接口中定义的函数在 `Stub` 类中可以没有具体实现。同时，在 `Stub` 类中重载了 `onTransact()` 方法，由于 `transact()` 方法内部给包裹内写入参数的顺序是由 aidl 工具定义的，因此，在 `onTransact()` 方法中，aidl 工具自然知道应该按照何种顺序从包裹中取出相应参数。

在 `Stub` 类中还定义了一些 `int` 常量，比如 `TRANSACTION_start`，这些常量与服务函数对应，`transact()` 和 `onTransact()` 方法的第一个参数 `code` 的值即来源于此。

刚接触 `IMusicPlayerService` 时，对以上描述的三个任务不容易从代码中看出，原因是这三个任务似乎更应该是分离的三个类，而 aidl 工具却把这些都放入了一个类中。理论上讲，的确可以把这三个任务写成三个类，但那会增加代码维护的烦琐。

在 `Stub` 类中，除了以上所述的任务外，`Stub` 还提供了一个 `asInterface()` 函数。提供这个函数的作用是这样的：首先需要明确的是，aidl 所产生的代码完全可以由应用程序员手工编写，`IMusicPlayerService` 中的函数只是一种编码习惯而已，`asInterface` 即如此，提供这个函数的原因是服务端提供的服务除了其他进程可以使用外，在服务进程内部的其他类也可以使用该服务，对于后者，显然是不需要经过 IPC 调用，而可以直接在进程内部调用的，而 `Binder` 内部有一个 `queryLocalInterface (String description)` 函数，该函数是根据输入的字符串判断该 `Binder` 对象是一个本地的 `Binder` 引用。在如图 5-1 所示中曾经指出，当创建一个 `Binder` 对象时，服务端进程内部创建一个 `Binder` 对象，`Binder` 驱动中也会创建一个 `Binder` 对象。如果从远程获取服务端的 `Binder`，则只会返回 `Binder` 驱动中的 `Binder` 对象，而如果从服务端进程内部获取 `Binder` 对象，则会获取服务端本身的 `Binder` 对象。听起来有点复杂，请看图 5-4 示意。

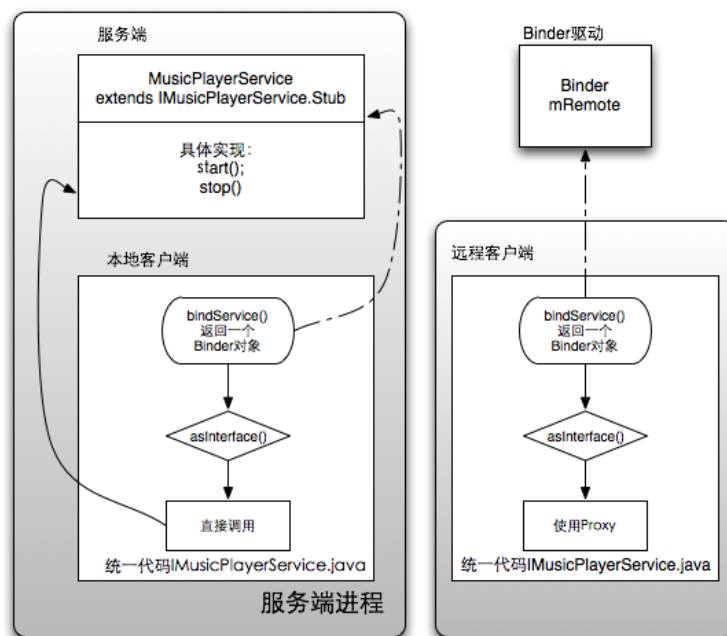


图 5-4 MusicPlayerService 服务中的代码结构

因此，asInterface()函数正是利用了 queryLocalInterface()方法，提供了一个统一的接口。无论是远程客户端还是本地端，当获取 Binder 对象后，可以把获取的 Binder 对象作为 asInterface()的参数，从而返回一个 IMusicPlayerService 接口，该接口要么使用 Proxy 类，要么直接使用 Stub 所实现的相应服务函数。

5.5 系统服务中的 Binder 对象

在应用程序编程时，经常使用 getSystemService (String serviceName) 方法获取一个系统服务，那么，这些系统服务的 Binder 引用是如何传递给客户端的呢？须知系统服务并不是通过 startService()启动的。

getSystemService()函数的实现是在 ContextImpl 类中，该函数所返回的 Service 比较多，具体可参照源码。这些 Service 一般都由 ServiceManager 管理。

5.5.1 ServiceManager 管理的服务

ServiceManager 是一个独立进程，其作用如名称所示，管理各种系统服务，管理的逻辑如图 5-5 所示。

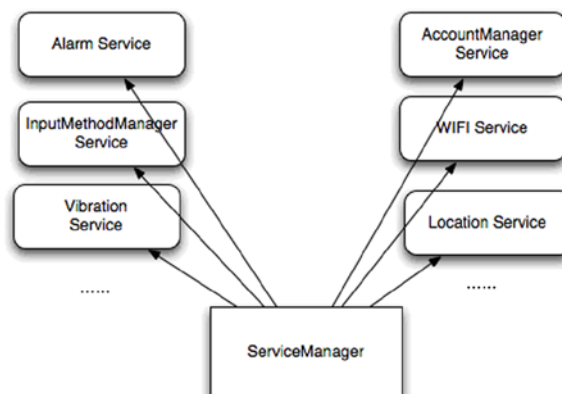


图 5-5 ServiceManager 所管理的服务列表

ServiceManager 本身也是一个 Service，Framework 提供了一个系统函数，可以获取该 Service 对应的 Binder 引用，那就是 `BinderInternal.getContextObject()`。该静态函数返回 `ServiceManager` 后，就可以通过 `ServiceManager` 提供的方法获取其他系统 Service 的 Binder 引用。这种设计模式在日常生活中到处可见，`ServiceManager` 就像是一个公司的总机，这个总机号码是公开的，系统中任何进程都可以使用 `BinderInternal.getContextObject()` 获取该总机的 Binder 对象，而当用户想联系公司中的其他人（服务）时，则要经过总机再获得分机号码。这种设计的好处是系统中仅暴露一个全局 Binder 引用，那就是 `ServiceManager`，而其他系统服务则可以隐藏起来，从而有助于系统服务的扩展，以及调用系统服务的安全检查。其他系统服务在启动时，首先把自己的 Binder 对象传递给 `ServiceManager`，即所谓的注册（`addService`）。

下面从代码实现来看以上逻辑。可以查看 `ContextImpl.getSystemService()` 中各种 Service 的具体获取方式，比如 `INPUT_METHOD_SERVICE`，代码如下：

```

922         } else if (INPUT_METHOD_SERVICE.equals(name)) {
923             return InputMethodManager.getInstance(this);
  
```

而 `InputMethodManager.getInstance (this)` 的关键代码如下：

```

458         synchronized (mInstanceSync) {
459             if (mInstance != null) {
460                 return mInstance;
461             }
462             IBinder b = ServiceManager.getService(Context.INPUT_METHOD_SERVICE);
463             IInputMethodManager service = IInputMethodManager.Stub.asInterface(b);
464             mInstance = new InputMethodManager(service, mainLooper);
465         }
466         return mInstance;
  
```

即通过 `ServiceManager` 获取 `InputMethod Service` 对应的 Binder 对象 `b`，然后再将该 Binder 对象作为 `IInputMethodManager.Stub.asInterface()` 的参数，返回一个 `IInputMethodManager` 的统一接口。

`ServiceManager.getService()` 的代码如下：

```

49 public static IBinder getService(String name) {
50     try {
51         IBinder service = sCache.get(name);
52         if (service != null) {
53             return service;
54         } else {
55             return getIServiceManager().getService(name);
56         }
57     } catch (RemoteException e) {
58         Log.e(TAG, "error in getService", e);
59     }
60     return null;
61 }

```

即首先从 sCache 缓存中查看是否有对应的 Binder 对象，有则返回，没有则调用 getIServiceManager().getService(name)，第一个函数 getIServiceManager()即用于返回系统中唯一的 ServiceManager 对应的 Binder，其代码如下：

```

private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }

    // Find the service manager
    sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
    return sServiceManager;
}

```

以上代码中，BinderInternal.getContextObject()静态函数即用于返回 ServiceManager 对应的全局 Binder 对象，该函数不需要任何参数，因为它的作用是固定的。从这个角度来看，这个函数的命名似乎更应该明确一些，比如，可以命名为 getServiceManager()。

其他所有通过 ServiceManager 获取的系统服务的过程与以上基本类似，所不同的就是传递给 ServiceManager 的服务名称不同，因为 ServiceManager 正是按照服务的名称（String 类型）来保存不同的 Binder 对象的。

关于使用 addService()向 ServiceManager 中添加一个服务一般是在 SystemService 进程启动时完成的，具体见第 9 章中关于 Framework 启动过程的描述。

5.5.2 理解 Manager

ServiceManager 所管理的所有 Service 都是以相应的 Manager 返回给客户端，因此，这里简述一下 Framework 中关于 Manager 的语义。在我们中国的企业里，Manager 一般指经理，比如项目经理、人事经理、部门经理。经理本身的含义比较模糊，其角色有些是给我们分配任务，比如项目经理；有些是给我们提供某种服务，比如人事经理；有些则是监督我们的工作等。而在 Android 中，Manager 的含义更应该翻译为经纪人，Manager 所 manage 的对象是服务本身，因为每个具体的服务一般都会提供多个 API 接口，而 Manager 所 manage 的正是这些 API。客户端一般不能直接通过 Binder 引用去访问具体的服务，





第 6 章 Framework 概述

从本章开始，真正进入 Android Framework 内核之旅。

任何控制类程序都有一个入口，汇编程序的入口由处理器内部的复位（Reset）中断向量表决定；C 程序的入口是 `main()` 函数，一个 C 程序只能有一个 `main()` 函数；Java 程序的入口必须是某个类的静态成员函数 `main()`。

对于依赖于操作系统的程序，客户程序除了包含一个程序入口外，还需要和相关系统服务一起运行，以完成指定的任务。比如，Win32 程序需要和 GUI 系统服务一起实现带有可视窗口的功能；X Window 程序也需要和 X Window Server 一起实现窗口功能。

Android 程序也不例外，那么，Android 程序的入口在哪里？Android Framework 都包含哪些必需的系统服务？这些系统服务是如何与 Android APK 程序配合的？本章将回答这些问题。关于各组件内部具体实现过程将在后续各章中分别阐述。

6.1 Framework 框架

Framework 定义了客户端组件和服务端组件功能及接口。以下阐述中，“应用程序”一般是指“.apk”程序。

框架中包含三个主要部分，分别为服务端、客户端和 Linux 驱动。

6.1.1 服务端

服务端主要包含两个重要类，分别是 `WindowManagerService`（WmS）和 `ActivityManagerService`



(AmS)。WmS 的作用是为所有的应用程序分配窗口，并管理这些窗口。包括分配窗口的大小，调节各窗口的叠放次序，隐藏或者显示窗口。AmS 的作用是管理所有应用程序中的 Activity。

除此之外，在服务端还包括两个消息处理类。

- KeyQ 类：该类为 WmS 的内部类，继承于 KeyInputQueue 类，KeyQ 对象一旦创建，就立即启动一个线程，该线程会不断地读取用户的 UI 操作消息，比如按键、触摸屏、trackball、鼠标等，并把这些消息放到一个消息队列 QueueEvent 类中。
- InputDispatcherThread 类：该类的对象一旦创建，也会立即启动一个线程，该线程会不断地从 QueueEvent 中取出用户消息，并进行一定的过滤，过滤后，再将这些消息发送给当前活动的客户端程序中。

6.1.2 客户端

客户端主要包括以下重要类。

- ActivityThread 类：该类为应用程序的主线程类，所有的 APK 程序都有且仅有一个 ActivityThread 类，程序的入口为该类中的 static main() 函数。
- Activity 类：该类为 APK 程序的一个最小运行单元，一个 APK 程序中可以包含多个 Activity 对象，ActivityThread 主类会根据用户操作选择运行哪个 Activity 对象。
- PhoneWindow 类：该类继承于 Window 类，同时，PhoneWindow 类内部包含了一个 DecorView 对象。简而言之，PhoneWindow 是把一个 FrameLayout 进行了一定的包装，并提供了一组通用的窗口操作接口。
- Window 类：该类提供了一组通用的窗口（Window）操作 API，这里的窗口仅仅是程序层面上的，WmS 所管理的窗口并不是 Window 类，而是一个 View 或者 ViewGroup 类，一般就是指 DecorView 类，即一个 DecorView 就是 WmS 所管理的一个窗口。Window 是一个 abstract 类型。
- DecorView 类：该类是一个 FrameLayout 的子类，并且是 PhoneWindow 中的一个内部类。Decor 的英文是 Decoration，即“修饰”的意思，DecorView 就是对普通的 FrameLayout 进行了一定的修饰，比如添加一个通用的 Title bar，并响应特定的按键消息等。
- ViewRoot 类：WmS 管理客户端窗口时，需要通知客户端进行某种操作，这些都是通过异步消息完成的，实现的方式就是使用 Handler，ViewRoot 就是继承于 Handler，其作用主要是接收 WmS 的通知。
- W 类：该类继承于 Binder，并且是 ViewRoot 的一个内部类。
- WindowManager 类：客户端要申请创建一个窗口，而具体创建窗口的任务是由 WmS 完成的，WindowManager 类就像是一个部门经理，谁有什么需求就告诉它，由它和 WmS 进行交互，客户端不能直接和 WmS 进行交互。

6.1.3 Linux 驱动

Linux 驱动和 Framework 相关的主要包含两部分，分别是 SurfaceFlinger（SF）和 Binder。每一个窗口都对应一个 Surface，SF 驱动的作用是把各个 Surface 显示在同一个屏幕上。

Binder 驱动的作用是提供跨进程的消息传递。

6.2 APK 程序的运行过程

首先，ActivityThread 从 main()函数中开始执行，调用 prepareMainLooper()为 UI 线程创建一个消息队列(MessageQueue)。

然后创建一个 ActivityThread 对象，在 ActivityThread 的初始化代码中会创建一个 H（Handler）对象和一个 ApplicationThread（Binder）对象。其中 Binder 负责接收远程 AmS 的 IPC 调用，接收到调用后，则通过 Handler 把消息发送到消息队列，UI 主线程会异步地从消息队列中取出消息并执行相应操作，比如 start、stop、pause 等。

接着 UI 主线程调用 Looper.loop()方法进入消息循环体，进入后就会不断地从消息队列中读取并处理消息。

当 ActivityThread 接收到 AmS 发送 start 某个 Activity 后，就会创建指定的 Activity 对象。Activity 又会创建 PhoneWindow 类→DecorView 类→创建相应的 View 或者 ViewGroup。创建完成后，Activity 需要把创建好的界面显示到屏幕上，于是调用 WindowManager 类，后者于是创建一个 ViewRoot 对象，该对象实际上创建了 ViewRoot 类和 W 类，创建 ViewRoot 对象后，WindowManager 再调用 WmS 提供的远程接口完成添加一个窗口并显示到屏幕上。

接下来，用户开始在程序界面上操作。KeyQ 线程不断把用户消息存储到 QueueEvent 队列中，InputDispatcherThread 线程逐个取出消息，然后调用 WmS 中的相应函数处理该消息。当 WmS 发现该消息属于客户端某个窗口时，就会调用相应窗口的 W 接口。

W 类是一个 Binder，负责接收 WmS 的 IPC 调用，并把调用消息传递给 ViewRoot，ViewRoot 再把消息传递给 UI 主线程 ActivityThread，ActivityThread 解析该消息并做相应的处理。在客户端程序中，首先处理消息的是 DecorView，如果 DecorView 不想处理某个消息，则可以将该消息传递给其内部包含的子 View 或者 ViewGroup，如果还没有处理，则传递给 PhoneWindow，最后再传递给 Activity。

6.3 客户端中的线程

在多任务操作系统中，任何程序都运行在线程之中。系统首先会为客户端程序分配一个线程，然后该线程从程序的入口处开始执行。那么，请思考以下问题。

- Android APK 程序中都有哪些线程？



- 什么是 UI 线程？
- 程序中自定义 Thread 和 UI 线程的区别是什么？

首先，很明确地讲，包含有 Activity 的客户端程序至少包含三个线程，如图 6-1 所示。每个 Binder 对象都对应一个线程，Activity 启动后会创建一个 ViewRoot.W 对象，同时 ActivityThread 会创建一个 ApplicationThread 对象，这两个对象都继承于 Binder，因此会启动两个线程，负责接收 Linux Binder 驱动发送 IPC 调用。最后一个主要线程也就是程序本身所在的线程，也叫做用户交互（UI）线程，因为所有的处理用户消息，以及绘制界面的工作都在该线程中完成。

为了验证这一点，可以在 Eclipse 中新建一个 Hello Android 的程序，然后以 debug 的方式运行，在 debug 窗口中会看到如图 6-1 所示的界面。

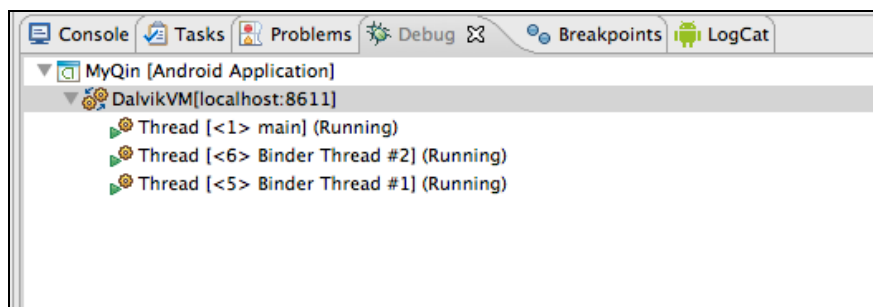


图 6-1 程序中线程

自定义 Thread 和 UI 线程的区别在于，UI 线程是从 ActivityThread 运行的，在该类中的 main() 方法中，已经使用 `Looper.prepareMainLooper()` 为该线程添加了 Looper 对象，即已经为该线程创建了消息队列（MessageQueue），因此，程序员才可以在 Activity 中定义 Handler 对象（因为声明 Handler 对象时，所在的线程必须已经创建了 MessageQueue）。而普通的自定义 Thread 是一个裸线程，因此，不能直接在 Thread 中定义 Handler 对象，从使用场景的角度讲，即不能直接给 Thread 对象发消息，但是却可以给 UI 线程发消息。

6.4 几个常见问题

在过去的实践中，经常有同学问到以下几个问题，故特意将此作为一节。

6.4.1 Activity 之间如何传递消息（数据）

首先，提出这个问题的原因是，程序员需要在不同的 Activity 之间传递数据，然而，这个问题本身就有问题。所谓“传递消息”一般是指多个线程之间，而 Activity 本身并不是线程，ActivityThread 才



是一个线程，即 UI 线程。同一个程序中的多个 Activity 都由 ActivityThread 进行调用，Activity 本身只是一个 Java 类而已，就像 Rect、Trigle 类一样，如果有人问“Rect 类和 Trigle 类之间如何传递消息”，你会不会觉得有点奇怪？

事实上，如果要在两个类中传递数据，方法可以有很多。

方法一：可以先实例化某个类，获得该类的引用，当其他类需要该对象的内部数据时，可以直接通过该引用去访问该类的内部数据。

方法二：对于 A、B 两个类之间，可以先实例化一个第三方类 C，然后两个类都可以把需要传递的数据存入 C 中，或从 C 中取出。

这些方法理论上都可以用在 Activity 类之间传递数据。然而，与普通类传递数据有所不同，普通类的实例化都是程序员显式完成的，而 Activity 类的实例化却是由 Framework 完成的，程序员只能使用 startActivity() 方法来告诉 Framework 去运行哪个 Activity，这就意味着程序员不能得到 Activity 对象的引用，那么就不能直接访问该对象的内部数据。解决的办法是使用 Activity.getApplication() 函数，该函数能够返回一个 Application 对象，该 Application 对象在该程序中是唯一的，同一程序中的不同 Activity 调用该函数所返回的 Application 对象是相同的，该对象的名称可以在 AndroidManifest.xml 中指定。一旦获取了该 Application 对象，就可以借助该对象，在不同的 Activity 之间传递数据。

除此之外，Framework 本身也提供了标准的 Activity 之间传递数据的方法，即 Intent 类。该类作为 startActivity() 的参数，仅用于在启动 Activity 时传递给目标 Activity，同时，如果调用 startActivityForResult()，目标 Activity 在结束后，也会返回一个 Intent 对象给原 Activity。

另外，从设计理念的角度来看，Android 认为，两个 Activity 如果要共享数据，可以通过 Preference Storage 或者文件、数据库进行，同时，在一般情况下，设备上只会有一个 Activity 在运行，因此，多个 Activity 之间传递数据也不是必需的。如果某个 Activity 需要在停止后还能处理某些数据，那么，该 Activity 似乎更应该被设计为一个后台的 Thread 或者一个 Service，无论是 Thread 还是 Service 都很容易获得其引用。

6.4.2 窗口相关的概念

在源码和本书中，会经常提到以下概念，窗口、Window 类、ViewRoot 类以及 W 类，本节简单介绍这些概念的联系和区别。

首先澄清几个概念。

- 窗口 (Window)：这是一个纯语义的说法，即程序员所看到的屏幕上的某个独立的界面，比如一个带有 Title Bar 的 Activity 界面、一个对话框、一个 Menu 菜单等，这些都称之为窗口。本书中所说的窗口管理一般也都泛指所有这些窗口，在 Android 的英文相关文章中则直接使用 Window 这个单词。而从 WmS 的角度来讲，窗口是接收用户消息的最小单元，WmS 内部用特



定的类表示一个窗口，以实现对窗口的管理。WmS 接收到用户消息后，首先要判断这个消息属于哪个窗口，然后通过 IPC 调用把这个消息传递给客户端的 ViewRoot 类。

- Window 类：该类在 android.view 包中，是一个 abstract 类，该类是对包含有可视界面的窗口的一种包装。所谓的可视界面就是指各种 View 或者 ViewGroup，一般可以通过 res/layout 目录下的 xml 文件描述。
- ViewRoot 类：该类在 android.view 包中，客户端申请创建窗口时需要一个客户端代理，用以和 WmS 进行交互，这个就是 ViewRoot 的功能，每个客户端的窗口都会对应一个 ViewRoot 类。
- W 类：该类是 ViewRoot 类的一个内部类，继承于 Binder，用于向 WmS 提供一个 IPC 接口，从而让 WmS 控制窗口客户端的行为。

描述一个窗口之所以使用这么多类的原因在于，窗口的概念存在于客户端和服务端（WmS）之中，客户端所理解的窗口和服务端理解的窗口是不同的，因此，在客户端和服务端会用不同的类来描述窗口。同时，无论是在客户端还是服务端，对窗口都有不同层面的抽象，比如在客户端，用户能看到的窗口一般是 View 或者 ViewGroup 组成的窗口，而与 Activity 对应的窗口却是一个 DecorView 类，而具备常规 Phone 操作接口的窗口却又是一个 PhoneWindow 类。



第 9 章 Framework 的启动过程

本书第 1 章中介绍过 Linux 系统的启动过程，在该过程的最后，内核将读取 init.rc 文件，并启动该文件中定义的各种服务程序。由于 Android 系统相对于 Linux 内核而言仅仅是一个 Linux 应用程序而已，因此，该程序也是在 init.rc 中被声明，从而当 Linux 内核启动后能够接着运行 Android 内核。本章将从 init.rc 文件开始，继续介绍 Android 内核的启动过程，以及该过程中相关的重要模块的交互逻辑。

9.1 Framework 运行环境综述

任何系统启动过程的本质都是要建立一套系统运行所需的环境，因此，本节首先介绍 Framework 的运行环境组成，然后再具体分析环境中所需子模块的启动过程。Framework 的运行环境如图 9-1 所示。

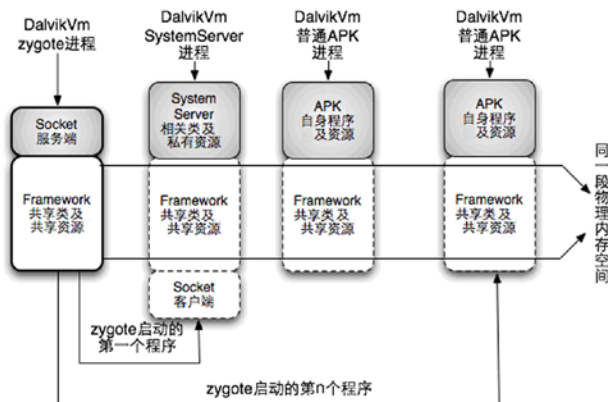


图 9-1 Dalvik 虚拟机进程间的关系

系统中运行的第一个 Dalvik 虚拟机程序叫做 `zygote`，该名称的意义是“一个卵”，因为接下来的所有 Dalvik 虚拟机进程都是通过这个“卵”孵化出来的。

`zygote` 进程中包含两个主要模块，分别如下：

- **Socket 服务端。**该 Socket 服务端用于接收启动新的 Dalvik 进程的命令。
- **Framework 共享类及共享资源。**当 `zygote` 进程启动后，会装载一些共享的类及资源，其中共享类是在 `preload-classes` 文件中被定义，共享资源是在 `preload-resources` 中被定义。因为 `zygote` 进程用于孵化出其他 Dalvik 进程，因此，这些类和资源装载后，新的 Dalvik 进程就不需要再装载这些类和资源了，这也就是所谓的共享。

`zygote` 进程对应的具体程序是 `app_process`，该程序存在于 `system/bin` 目录下，启动该程序的指令是在 `init.rc` 中进行配置的。

`zygote` 孵化出的第一个 Dalvik 进程叫做 `SystemServer`，`SystemServer` 仅仅是该进程的别名，而该进程具体对应的程序依然是 `app_process`，因为 `SystemServer` 是从 `app_process` 中孵化出来的。

`SystemServer` 中创建了一个 Socket 客户端，并有 `AmS` 负责管理该客户端，之后所有的 Dalvik 进程都将通过该 Socket 客户端间接被启动。当需要启动新的 APK 进程时，`AmS` 中会通过该 Socket 客户端向 `zygote` 进程的 Socket 服务端发送一个启动命令，然后 `zygote` 会孵化出新的进程。

从系统架构的角度来讲，就在于此即先创建一个 `zygote`，并加载共享类和资源，然后通过该 `zygote` 去孵化新的 Dalvik 进程，该架构的特点有两个。

- 每一个进程都是一个 Dalvik 虚拟机，而 Dalvik 虚拟机是一种类似于 Java 虚拟机的程序，并且从开发的过程来看，与标准的 Java 程序开发基本一致。因此对于程序员来讲，不需要学习新的语言，并可以使用 Java 程序在过去几十年中已经成熟的各种类库资源。
- `zygote` 进程预先会装载共享类和共享资源，这些类及资源实际上就是 SDK 中定义的大部分类和资源。因此，当通过 `zygote` 孵化出新的进程后，新的 APK 进程只需要去装载 APK 自身包含的类和资源即可，这就有效地解决了多个 APK 共享 Framework 资源的问题。

9.2 Dalvik 虚拟机相关的可执行程序

在 Android 源码中，大家会发现好几处和 Dalvik 这个概念相关的可执行程序，正确区分这些可执行程序的区别将有助于理解 Framework 内部结构。这些可执行程序的名称和源码路径如表 9-1 所示。

表 9-1 和虚拟机相关的源码

名 称	源码路径
<code>dalvikvm</code>	<code>dalvik/dalvikvm</code>
<code>dvz</code>	<code>dalvik/dvz</code>
<code>app_process</code>	<code>frameworks/base/cmds/app_process</code>

下面将分别介绍这些可执行程序的作用。



9.2.1 dalvikvm

当 Java 程序运行时，都是由一个虚拟机来解释 Java 的字节码，它将这些字节码翻译成本地 CPU 的指令码，然后执行。对 Java 程序而言，负责解释并执行的就是一个虚拟机，而对于 Linux 而言，这个进程只是一个普通的进程，它与一个只有一行代码的 Hello World 可执行程序无本质区别。所以启动一个虚拟机的方法就跟启动任何一个可执行程序的方法是相同的，那就是在命令行下输入可执行程序的名称，并在参数中指定要执行的 Java 类。

dalvikvm 的作用就是创建一个虚拟机并执行参数中指定的 Java 类，下面以一个例子来说明该程序的使用方法。

首先新建一个 Foo.java 文件，如以下代码所示：

```
class Foo {
    public static void main(String[] args) {
        System.out.println("Hello dalvik");
    }
}
```

然后编译该文件，并生成 Jar 文件，如以下代码所示：

```
$ javac Foo.java
$ PATH=/Users/keyd/android/out/host/darwin-x86/bin:$PATH
$ dx --dex --output=foo.jar Foo.class
```

dx 工具的作用是将.class 转换为 dex 文件，因为 Dalvik 虚拟机所执行的程序不是标准的 Jar 文件，而是将 Jar 文件经过特别的转换以提高执行效率，而转换后的文件就是 dex 文件。dx 工具是 Android 源码的一部分，其路径是在 out 目录下，因此在执行 dx 之前，需要添加该路径。

dx 执行时，--output 参数用于指定 Jar 文件的输出路径，注意该 Jar 文件内部包含已经不是纯粹的.class 文件，而是 dex 格式文件，Jar 仅仅是 zip 包。

生成了该 Jar 包后，就可以把该 Jar 包 push 到设备中，并执行，如以下代码所示：

```
$ adb push foo.jar /data/app
$ adb shell dalvikvm -cp /data/app/foo.jar Foo
Hello dalvik
```

以上命令首先将该 Jar 包 push 到/data/app 目录下，因为该目录一般用于存放应用程序，接着使用 adb shell 执行 dalvikvm 程序。dalvikvm 的执行语法如下：

```
dalvikvm -cp 类路径 类名
```

从这里也可以感觉到，dalvikvm 的作用就像在 PC 上执行 Java 程序一样。

9.2.2 dvz

dvz 的作用是从 zygote 进程中孵化出一个新的进程，新的进程也是一个 Dalvik 虚拟机。该进程与



dalvikvm 启动的虚拟机相比，区别在于该进程中已经预装了 Framework 的大部分类和资源，下面以一个具体的例子来看 dvz 的使用方法。

首先在 Eclipse 下新建一个 APK 项目，包名称为 com.haiii.android.helloapk，默认的 Activity 名称为 Welcome，其内容如下：

```
public class Welcome extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public static void main(String[] args) {
        System.out.println("Hello dalvik");
    }
}
```

该段代码是大家非常熟悉的 Hello Android 代码，唯一的区别在于增加了一个 static main() 函数，因为该函数将作为 Welcome 类的入口。

接下来，将生成的 APK 文件 push 到 /data/app 目录下，然后运行 Welcome 类，如以下代码所示：

```
# dvz -classpath /data/app/HelloApk.apk com.haiii.android.helloapk.Welcome
Hello dalvik
In mgmain JNI_OnLoad
```

dvz 的语法如下：

```
dvz -classpath 包名称 类名
```

讲到这里，有的读者可能猜想，是否可以在 main() 函数内部构造一个 Welcome 对象，从而可以达到运行该 APK 的目的？答案是否定的，因为 Welcome 类并不是该应用程序的入口类，在后面的章节中，大家将看到，一个 APK 的入口类是 ActivityThread 类，Activity 类仅仅是被回调的类，因此不可以通过 Activity 类来启动一个 APK，dvz 工具仅仅用于 Framework 开发过程的调试。

9.2.3 app_process

以上讲述的 dalvikvm 和 dvz 是通用的两个工具，然而 Framework 在启动时需要加载运行两个特定 Java 类，一个是 ZygoteInit.java，另一个是 SystemServer.java。为了便于使用，系统才提供了一个 app_process 进程，该进程会自动运行这两个类，从这个角度来讲，app_process 的本质就是使用 dalvikvm 启动 ZygoteInit.java，并在启动后加载 Framework 中的大部分类和资源。

下面就来对比一下 app_process 和 dalvikvm 的主要执行过程。首先来看 dalvikvm，其源码文件在 dalvik/dalvikvm/Main.c 中，该源码中的关键代码有两处。

第一处是创建一个 vm 对象，如以下代码所示：

```

209  /*
210   * Start VM. The current thread becomes the main thread of the VM.
211   */
212  if (JNI_CreateJavaVM(&vm, &env, &initArgs) < 0) {
213      fprintf(stderr, "Dalvik VM init failed (check log file)\n");
214      goto bail;
215  }

```

该段代码通过调用 JNI_CreateJavaVM() 同时创建了 JavaVm 对象和 JNIEnv 对象，这两个对象的定义如下：

```

JNIEnvExt* pEnv = NULL;
JavaVMExt* pVM = NULL;

```

该函数的参数是“指针的指针”类型，其原型如下：

```

jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args)

```

创建好了 JavaVm 对象后，就可以使用该对象去加载指定的类了，如以下关键代码所示：

```

249  startClass = (*env)->FindClass(env, slashClass);
250  if (startClass == NULL) {
251      fprintf(stderr, "Dalvik VM unable to locate class '%s'\n", slashClass);
252      goto bail;
253  }
254
255  startMeth = (*env)->GetStaticMethodID(env, startClass,
256      "main", "([Ljava/lang/String;)V");
257  if (startMeth == NULL) {
258      fprintf(stderr, "Dalvik VM unable to find static main(String[]) in '%s'
259      slashClass);
260      goto bail;
261  }
262
263  /*
264   * Make sure the method is public. JNI doesn't prevent us from calling
265   * a private method, so we have to check it explicitly.
266   */
267  if (!methodIsPublic(env, startClass, startMeth))
268      goto bail;
269
270  /*
271   * Invoke main().
272   */
273  (*env)->CallStaticVoidMethod(env, startClass, startMeth, strArray);

```

该段代码首先通过调用 FindClass() 找到指定的 class 文件，然后调用 GetStaticMethodID() 找到 main() 函数，最后调用 CallStaticVoidMethod 执行该 main() 函数。

接下来再来看 app_process 中是如何创建虚拟机并执行指定的 class 文件的。其源文件在



frameworks/base/cmds/app_main.cpp 中，该文件中的关键代码有两处，第一处是先创建一个 AppRuntime 对象，如以下代码所示：

```
130 AppRuntime runtime;
```

第二处是调用 runtime 的 start()方法启动指定的 class，如以下代码所示：

```
153 if (0 == strcmp("--zygote", arg)) {
154     bool startSystemServer = (i < argc) ?
155         strcmp(argv[i], "--start-system-server") == 0 : false;
156     setArgv0(argv0, "zygote");
157     set_process_name("zygote");
158     runtime.start("com.android.internal.os.ZygoteInit",
159         startSystemServer);
```

系统中只有一处使用 app_process，那就是在 init.rc 中，使用时参数包含了 --zygote 及 --start-system-server，因此，这里仅分析包含这两个参数的情况。start()方式是 AppRuntime 类的成员函数，而 AppRuntime 是在该文件中定义的一个应用类，其父类是 AndroidRuntime，该类的实现文件在 frameworks/base/core/jni/AndroidRuntime.cpp 中。start()函数中首先调用 startVm()创建一个 vm 对象，然后就和 dalvikvm 一样先找到 Class()，再执行 class 中 main()函数。

startVm()函数使用以下代码创建 vm 对象：

```
865 if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
866     LOGE("JNI_CreateJavaVM failed\n");
867     goto bail;
868 }
```

由此可以看出，app_process 和 dalvikvm 在本质上是相同的，唯一的区别就是 app_process 可以指定一些特别的参数，这些参数有利于 Framework 启动特定的类，并进行一些特别的系统环境参数设置。有兴趣的读者可以依照以上流程详细分析其内部差别。

9.3 zygote 的启动

前面小节介绍了 Framework 的运行环境，以及 Dalvik 虚拟机的相关启动方法，zygote 进程是所有 APK 应用进程的父进程，接下来就详细介绍 zygote 进程的内部启动过程。

9.3.1 在 init.rc 中配置 zygote 启动参数

init.rc 存在于设备的根目录下，读者可以使用 adb pull /init.rc ~/Desktop 命令取出该文件，文件中和 zygote 相关的配置信息如下：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```



```

socket zygote stream 666
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd

```

首先第一行中使用 `service` 指令告诉操作系统将 `zygote` 程序加入到系统服务中, `service` 的语法如下:

`service service_name` 可执行程序的路径 可执行程序自身所需的参数列表

此处的服务被定义为 `zygote`, 理论上讲该服务的名称可以是任意的。可执行程序的路径正是 `/system/bin/app_process`, 也就是前面所讲的 `app_process`, 参数一共包含四个, 分别如下:

- `-Xzygote`, 该参数将作为虚拟机启动时所需要的参数, 是在 `AndroidRuntime.cpp` 类的 `startVm()` 函数中调用 `JNI_CreateJavaVM()` 时被使用的。
- `/system/bin`, 代表虚拟机程序所在目录, 因为 `app_process` 完全可以不和虚拟机在同一个目录, 而在 `app_process` 内部的 `AndroidRuntime` 类内部需要知道虚拟机所在的目录。
- `--zygote`, 指明以 `ZygoteInit` 类作为虚拟机执行的入口, 如果没有 `--zygote` 参数, 则需要明确指定需要执行的类名。
- `--start-system-server`, 仅在指定 `--zygote` 参数时才有效, 意思是告知 `ZygoteInit` 启动完毕后孵化出第一个进程 `SystemServer`。

接下来的配置命令 `socket` 用于指定该服务所使用到的 `socket`, 后面的参数依次是名称、类型、端口地址。

`onrestart` 命令指定该服务重启的条件, 即当满足这些条件后, `zygote` 服务就需要重启, 这些条件一般是一些系统异常条件。

9.3.2 启动 Socket 服务端口

当 `zygote` 服务从 `app_process` 开始启动后, 会启动一个 Dalvik 虚拟机, 而虚拟机执行的第一个 Java 类就是 `ZygoteInit.java`, 因此接下来的过程就从 `ZygoteInit` 类的 `main()` 函数开始说起。 `main()` 函数中做的第一个重要工作就是启动一个 `Socket` 服务端口, 该 `Socket` 端口用于接收启动新进程的命令。

启动 `Socket` 服务端口是在静态函数 `registerZygoteSocket()` 中完成的, 如以下代码所示:

```

private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
        try {
            String env = System.getenv(ANDROID_ID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
            ... ..
            try {
                sServerSocket = new LocalServerSocket(

```



```

        createFileDescriptor(fileDesc));
    ... ..
}
}

```

在该段代码中，首先调用 `System.getenv()` 获取系统为 `zygote` 进程分配的 `Socket` 文件描述符号，然后再调用 `createFileDescriptor()` 创建一个真正的文件描述符，最后以该描述符为参数，构造了一个 `LocalServerSocket` 对象。

关于 `Socket` 编程的基础知识，读者可以参考《UNIX Networking Programming》一书，作者为 W.Richard Stevens。这里要说明的是，在 Linux 系统中，所有的系统资源都可以看成是文件，甚至包括内存和 CPU，因此，像标准的磁盘文件或者网络 `Socket` 自然也被认为是文件，这就是为什么 `LocalServerSocket` 构造函数的参数是一个文件描述符。

`Socket` 编程中有两种方式去触发 `Socket` 数据读操作。一种是使用 `listen()` 监听某个端口，然后调用 `read()` 去从这个端口上读数据，这种方式被称为阻塞式读操作，因为当端口没有数据时，`read()` 函数将一直等待，直到数据准备好后才返回；另一种是使用 `select()` 函数将需要监测的文件描述符作为 `select()` 函数的参数，然后当该文件描述符上出现新的数据后，自动触发一个中断，然后在中断处理函数中再去读指定文件描述符上的数据，这种方式被称为非阻塞式读操作。`LocalServerSocket` 中使用的正是后者，即非阻塞读操作。

当 `LocalServerSocket` 端口准备好后，`main()` 函数中调用 `runSelectLoopMode()` 进入非阻塞读操作，该函数中首先将 `sServerSocket` 加入到被监测的文件描述符列表中，如以下代码所示：

```

654      FileDescriptor[] fdArray = new FileDescriptor[4];
655
656      fds.add(sServerSocket.getFileDescriptor());

```

然后在 `while(true)` 循环中将该文件描述符添加到 `select` 的列表中，并调用 `ZygoteConnection` 类的 `runOnce()` 函数处理每一个 `Socket` 接收到的命令，如以下代码所示：

```

try {
    fdArray = fds.toArray(fdArray);
    index = selectReadable(fdArray);
} catch (IOException ex) {
    throw new RuntimeException("Error in select()", ex);
}

if (index < 0) {
    throw new RuntimeException("Error in select()");
} else if (index == 0) {
    ZygoteConnection newPeer = acceptCommandPeer();
    peers.add(newPeer);
    fds.add(newPeer.getFileDescriptor());
} else {
    boolean done;
    done = peers.get(index).runOnce();
}

```

```

    if (done) {
        peers.remove(index);
        fds.remove(index);
    }
}

```

selectReadable()函数的返回值有三种。一种是-1，代表着内部错误；第二种是 0，代表着没有可处理的连接，因此会以 Socket 服务端口重新建立一个 ZygoteConnection 对象，并等待客户端的请求；第三种是大于 0，代表着还有没有处理完的连接请求，因此需要先处理该请求，而暂时不需要建立新的连接等待。

runOnce()函数的核心代码是基于 zygote 进程孵化出新的应用进程，如以下代码所示：

```

212     pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
213                                   parsedArgs.gids, parsedArgs.debugFlags, rlimits);

```

关于 folk 的概念将在后面小节中介绍。

以上介绍的是 Socket 的服务端，而在 SystemServer 进程中则会创建一个 Socket 客户端，具体的实现代码是在 Process.java 类中，而调用 Process 类是在 AmS 类中的 startProcessLocked()函数中，如以下代码所示，关于该函数的调用实际参照第 10 章 AmS 原理。

```

1875     int pid = Process.start("android.app.ActivityThread",
1876                           mSimpleProcessManagement ? app.processName : null, uid, uid,
1877                           gids, debugFlags, null);

```

而 start()函数内部又调用了静态函数 startViaZygote()，该函数的实体正是使用一个本地 Socket 向 zygote 中的 Socket 发送进行启动命令，其执行流程如图 9-2 所示。

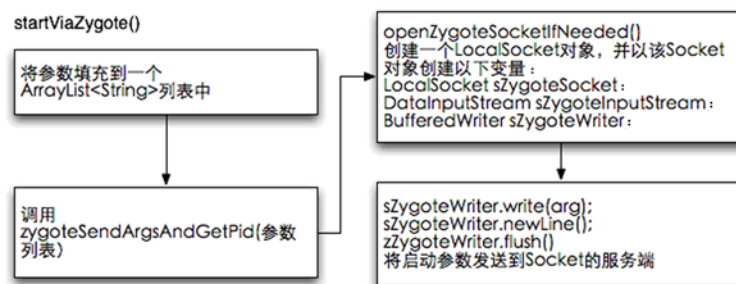


图 9-2 startViaZygote()函数的执行过程

该流程的主要过程就是将 startViaZygote()的函数参数转换为一个 ArrayList<String>列表，然后再构造出一个 LocalSocket 本地 Socket 接口，并通过该 LocalSocket 对象构造出一个 BufferedWriter 对象，最后通过该对象将 ArrayList<String>列表中的参数传递给 zygote 中的 LocalServerSocket 对象，而在 zygote 端，就会调用 Zygote.forkAndSpecialize()函数孵化出一个新的应用进程。

9.3.3 加载 preload-classes

在 `ZygoteInit` 类的 `main()` 函数中，创建完 `Socket` 服务端后还不能立即孵化新的进程，因为这个“卵”中还没有必须“核酸”，这个“核酸”就是指预装的 Framework 大部分类及资源。

预装的类列表是在 `framework.jar` 中的一个文本文件列表，名称为 `preload-classes`，该列表的原始定义在 `frameworks/base/preload-classes` 文本文件中，而该文件又是通过 `frameworks/base/tools/preload/WritePreloadedClassFile.java` 类生成的。产生 `preload-classes` 的方法是在 `Android` 根目录下执行以下命令：

```
$java -Xss512M -cp /path/to/preload.jar WritePreloadedClassFile /path/to/.compiled
1517 classes were loaded by more than one app.
Added 147 more to speed up applications.
1664 total classes will be preloaded.
Writing object model...
Done!
```

在该命令中，`/path/to/preload.jar` 是指 `out/host/darwin-x86/framework/preload.jar`，该 Jar 是由 `frameworks/base/tools/preload` 子项目编译而成的。

`/path/to/.compiled/` 是指 `frameworks/base/tools/preload` 目录下的那几个 `.compiled` 文件。

参数 `-Xss` 用于执行该程序所需要的 Java 虚拟机栈大小，此处使用 512MB，默认的大小不能满足该程序的运行，会抛出 `java.lang.StackOverflowError` 错误信息。

`WritePreloadedClassFile` 是要执行的具体类。

执行完以上命令后，会在 `frameworks/base` 目录下产生 `preload-classes` 文本文件。从该命令的执行情况来看，预装的 Java 类信息包含在 `.compiled` 文件中，而这个文件却是一个二进制文件，尽管我们目前能够确知如何产生 `preload-classes`，但却无法明确这个 `.compiled` 文件是如何产生的，一个可能的假设如下：

在 `Android` 项目组内部可能会存在一个测试项目，该项目一旦运行，就会装载一些 Java 类。当然，这些 Java 类是测试项目中的程序代码主动装载的，而这些程序代码被认为是大多数 `Android` 程序运行时都会执行的代码。一旦该运行环境建立后，`Dalvik` 虚拟机内存中就记录了所有被装载的 Java 类，然后该测试项目会使用一个特别的工具从虚拟机内存中读取所有装载过的类信息，并生成 `compiled` 文件。当然，这只是一种假设。

在 `Android` 源码编译的时候，会最终把 `preload-classes` 文件打包到 `framework.jar` 中，关于其详细过程参见本书 `Android` 源码编译的相关章节。

有了这个列表后，`ZygoteInit` 中通过调用 `preloadClasses()` 完成装载这些类。装载的方法很简单，就是读取 `preload-classes` 列表中的每一行，因为每一行代表了一个具体的类，然后调用 `Class.forName()` 装载目标类，如下代码所示。在装载的过程中，忽略以 `#` 开始的目标类，并忽略换行符及空格。

```

286         while ((line = br.readLine()) != null) {
287             // Skip comments and blank lines.
288             line = line.trim();
289             if (line.startsWith("#") || line.equals("")) {
290                 continue;
291             }
292
293             try {
294                 if (Config.LOGV) {
295                     Log.v(TAG, "Preloading " + line + "...");
296                 }
297                 Class.forName(line);

```

9.3.4 加载 preload-resources

preload-resources 是在 frameworks/base/core/res/res/values/arrays.xml 中被定义的，包含两类资源，一类是 drawable 资源，另一类是 color 资源，如以下代码所示：

```

<array name="preloaded_drawables">
    <item>@drawable/sym_def_app_icon</item>
    ... ..
</array>

<array name="preloaded_color_state_lists">
    <item>@color/hint_foreground_dark</item>
    ... ..
</array>

```

而加载这些资源是在 preloadResources() 函数中完成的，该函数中分别调用 preloadDrawables() 和 preloadColorStateLists() 加载这两类资源。加载的原理很简单，就是把这些资源读出来放到一个全局变量中，只要该类对象不被销毁，这些全局变量就会一直保存。

保存 Drawable 资源的全局变量是 mResources，该变量的类型是 Resources 类，由于该类内部会保存一个 Drawable 资源列表，因此，实际上缓存这些 Drawable 资源是在 Resources 内部；保存 Color 资源的全局变量也是 mResources，同样，Resources 类内部也有一个 Color 资源的列表。

关于 Resources 内部如何保存这些资源，请参照资源访问章节。

9.3.5 使用 fork 启动新的进程

fork 是 Linux 系统的一个系统调用，其作用是复制当前进程，产生一个新的进程。新进程将拥有和原始进程完全相同的进程信息，除了进程 id 不同。进程信息包括该进程所打开的文件描述符列表、所分配的内存等。当新进程被创建后，两个进程将共享已经分配的内存空间，直到其中一个需要向内存中



写入数据时，操作系统才负责复制一份目标地址空间，并将要写的的数据写入到新的地址中，这就是所谓的 copy-on-write 机制，即“仅当写的时候才复制”，这种机制可以最大限度地多个进程中共享物理内存。

第一次接触 folk 的读者可能觉得奇怪，为什么要复制进程呢？在大家熟悉的 Windows 操作系统中，一个应用程序一般对应一个进程，如果说要复制进程，可能的结果就是从计算器程序复制出一个 Office 程序，这听起来似乎很不合理。要立即复制进程就需要首先了解进程的启动过程。

在所有的操作系统中，都存在一个程序装载器，程序装载器一般会作为操作系统的一部分，并由所谓的 Shell 程序调用。当内核启动后，Shell 程序会首先启动。常见的 Shell 程序包含两大类，一类是命令行界面，另一类是窗口界面，Windows 系统中 Shell 程序就是桌面程序，Ubuntu 系统中的 Shell 程序就是 GNOME 桌面程序。Shell 程序启动后，用户可以双击桌面图标启动指定的应用程序，而在操作系统内部，启动新的进程包含三个过程。

第一个过程，内核创建一个进程数据结构，用于表示将要启动的进程。

第二个过程，内核调用程序装载器函数，从指定的程序文件读取程序代码，并将这些程序代码装载到预先设定的内存地址。

第三个过程，装载完毕后，内核将程序指针指向到目标程序地址的入口处开始执行指定的进程。当然，实际的过程会考虑更多的细节，不过大致思路就是这么简单。

在一般情况下，没有必要复制进程，而是按照以上三个过程创建新进程，但当满足以下条件时，则建议使用复制进程：即两个进程中共享了大量的程序。

举个例子，去澳大利亚看袋鼠和去澳大利亚看考拉，这是两个进程，但完成这两个进程的大多数任务都是相同的，即先订机票，然后带照相机，再坐地铁到首都机场，最后再坐 14 个小时的飞机到澳大利亚，到了之后唯一不同就是看考拉和袋鼠。为了更有效地完成这两个任务，可以先雇佣一个精灵进程，让它订机票、带相机、坐地铁、乘飞机，一直到澳大利亚后，从这个精灵进程中复制出两个进程，一个去看考拉，另一个去看袋鼠。如果你愿意，还可以去悉尼歌剧院，这就是进程的复制，其好处是节省了大量共享的内存。

由于 folk() 函数是 Linux 的系统调用，Android 中的 Java 层仅仅是对该调用进行了 JNI 封装而已，因此，接下来以一段 C 代码来介绍 folk() 函数的使用，以便大家对该函数有更具体的认识。

```
/**
 *FileName: MyFolk.c
 */
#include <sys/types.h>
#include <unistd.h>
int main(){
    pid_t pid;
    printf("pid = %d, Take camera, by subway, take air! \n", getpid());
    pid = folk();
    if(pid > 0){
        printf("pid=%d, 我是精灵! \n", getpid());
        pid = folk();
        if(!pid) printf("pid=%d, 去看考拉! \n", getpid());
    }
}
```

```

else if (!pid) printf("pid=%d, 去看袋鼠! \n", getpid());
else if (pid == -1) perror("folk");
getchar();
}

```

以上代码的执行结果如下：

```

$ ./MyFolk.bin
pid = 3927, Take camera, by subway, take air!
pid=3927, 我是精灵!
pid=3929, 去看袋鼠!
pid=3930, 去看考拉!

```

fork()函数的返回值与普通函数调用完全不同。当返回值大于 0 时，代表的是父进程；当等于 0 时，代表的是被复制的进程。换句话说，父进程和子进程的代码都在该 C 文件中，只是不同的进程执行不同的代码，而进程是靠 fork()的返回值进行区分的。

由以上执行结果可以看出，第一次调用 fork()时复制了一个“看袋鼠”进程，然后在父进程中再次调用 fork()复制了“看考拉”的进程，三者都有各自不同的进程 id。

zygote 进程就是本例中的“精灵进程”，那些“拿相机、坐地铁、乘飞机”的操作就是 zygote 进程中加载的 preload-classes 类具备的功能。

ZygoteInit.java 中复制新进程是通过在 runSelectLoopMode()函数中调用 ZygoteConnection 类的 runOnce()函数完成的，而该函数中则调用了以下代码用于复制一个新的进程。

```

212 pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
213     parsedArgs.gids, parsedArgs.debugFlags, rlimits);

```

forkAndSpecialize()函数是一个 native 函数，其内部的执行原理和上面的 C 代码类似。

当新进程被创建好后，还需要做一些“善后”工作。因为当 zygote 复制新进程时，已经创建了一个 Socket 服务端，而这个服务端是不应该被新进程使用的，否则系统中会有多个进程接收 Socket 客户端的命令。因此，新进程被创建好后，首先需要在新进程中关闭该 Socket 服务端，并调用新进程中指定的 Class 文件的 main()函数作为新进程的入口点。而这些正是在调用 forkAndSpecialize()函数后根据返回值 pid 完成的，如以下代码所示：

```

223 if (pid == 0) {
224     // in child
225     handleChildProc(parsedArgs, descriptors, newStderr);
226     // should never happen
227     return true;
228 } else { /* pid != 0 */
229     // in parent...pid of < 0 means failure
230     return handleParentProc(pid, descriptors, parsedArgs);
231 }

```

pid 等于 0 时，代表的是子进程，handleChildProc()函数中的关键代码如下，首先是关于 Socket 服务端。



```

696     } else {
697         closeSocket();
698         ZygoteInit.closeServerSocket();

```

接着从指定 Class 文件的 main()函数处开始执行，如以下代码所示：

```

742     try {
743         ZygoteInit.invokeStaticMain(cloader, className, mainArgs);

```

至此，新进程就完全脱离了 zygote 进程的孵化过程，成为一个真正的应用进程。

9.4 SystemServer 进程的启动

SystemServer 进程是 zygote 孵化出的第一个进程，该进程是从 ZygoteInit.java 的 main()函数中调用 startSystemServer()开始的。与启动普通进程的差别在于，zygote 类为启动 SystemServer 提供了专门的函数 startSystemServer()，而不是使用标准的 forAndSpecilize()函数，同时，SystemServer 进程启动后首先要做的事情和普通进程也有所差别。

startSystemServer()函数的关键代码有三处。

第一处，定义了一个 String[]数组，数组中包含了要启动的进程的相关信息，其中最后一项指定新进程启动后装载的第一个 Java 类，此处即为 com.android.server.SystemServer 类，如以下代码所示：

```

511     String args[] = {
512         "--setuid=1000",
513         "--setgid=1000",
514         "--setgroups=1001,1002,1003,1004,1005,
515         "--capabilities=130104352,130104352",
516         "--runtime-init",
517         "--nice-name=system_server",
518         "com.android.server.SystemServer",
519     };

```

第二处，调用 forkSystemServer()从当前的 zygote 进程孵化出新的进程，如以下代码所示。该函数是一个 native 函数，其作用与 folkAndSpecilize()相似。

```

537     pid = Zygote.forkSystemServer(
538         parsedArgs.uid, parsedArgs.gid,
539         parsedArgs.gids, debugFlags, null,
540         parsedArgs.permittedCapabilities,
541         parsedArgs.effectiveCapabilities);

```

第三处，新进程启动后，首先执行的代码如下：

```

546     /* For child process */
547     if (pid == 0) {
548         handleSystemServerProcess(parsedArgs);
549     }

```

在 handleSystemServerProcess()函数中主要完成两件事情，第一是关闭 Socket 服务端，第二是执行



`com.android.server.SystemServer` 类的 `main()` 函数。除了这两个主要事情外，还做了一些额外的运行环境配置，这些配置主要在 `commonInit()` 和 `zygoteInitNative()` 两个函数中完成。有兴趣的读者可以仔细分析这两个函数的内部实现。

一旦 `SystemServer` 的进程环境配置好后，就从 `SystemServer` 类的 `main()` 函数中开始运行，本节主要介绍该类的内部执行过程。

9.4.1 启动各种系统服务线程

`SystemServer` 进程在 Android 的运行环境中扮演了“神经中枢”的作用，APK 应用中能够直接交互的大部分系统服务都在该进程中运行，常见的比如 `WindowManagerServer`（`Wms`）、`ActivityManagerSystemService`（`AmS`）、`PackageManagerServer`（`PmS`）等，这些系统服务都是以一个线程的方式存在于 `SystemServer` 进程中。下面就来介绍到底都有哪些服务线程，及其启动的顺序。

`SystemServer` 的 `main()` 函数首先调用的是 `init1()` 函数，这是一个 `native` 函数，内部会进行一些与 Dalvik 虚拟机相关的初始化工作。该函数执行完毕后，其内部会调用 Java 端的 `init2()` 函数，这就是为什么 Java 源码中没有引用 `init2()` 的地方，主要的系统服务都是在 `init2()` 函数中完成的。

该函数首先创建了一个 `ServerThread` 对象，该对象是一个线程，然后直接运行该线程，如以下代码所示：

```
626 public static final void init2() {
627     Slog.i(TAG, "Entered the Android system server!");
628     Thread thr = new ServerThread();
629     thr.setName("android.server.ServerThread");
630     thr.start();
631 }
```

于是，从 `ServerThread` 的 `run()` 方法内部开始真正启动各种服务线程。

基本上每个服务都有对应的 Java 类，从编码规范的角度来看，启动这些服务的模式可归类为三种，如图 9-3 所示。

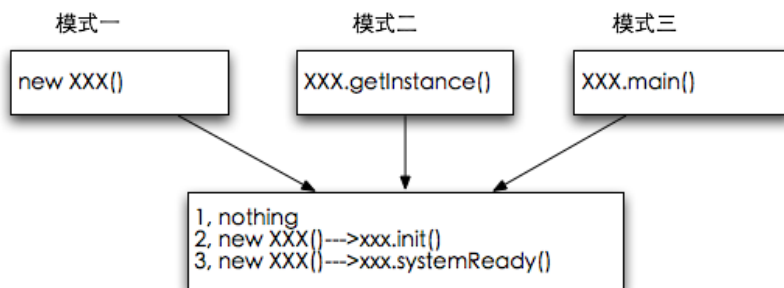


图 9-3 不同服务的启动方式

- 模式一是指直接使用构造函数构造一个服务，由于大多数服务都对应一个线程，因此，在构造函数内部就会创建一个线程并自动运行。
- 模式二是指服务类会提供一个 `getInstance()` 方法，通过该方法获取该服务对象，这样的好处是保证系统中仅包含一个该服务对象。
- 模式三是指从服务类的 `main()` 函数中开始执行。

无论以上何种模式，当创建了服务对象后，有时可能还需要调用该服务类的 `init()` 或者 `systemReady()` 函数以完成该对象的启动，当然这些都是服务类内部自定义的。为了区分以上启动的不同，以下采用一种新的方式描述该启动过程。比如当一个服务对象是通过模式一创建，并调用 `init()` 完成该服务的启动，我们就用模式 1.2 表示；如果构造函数返回后就已经启动，而无须任何其他调用，即什么都不做（nothing），我们就用模式 1.1 表示。

表 9-2 列出了 `SystemService` 中所启动的所有服务，以及这些服务的启动模式。

表 9-2 `SystemService` 中启动服务列表

服务类名称	作用描述	启动模式
<code>EntropyService</code>	提供伪随机数	1.0
<code>PowerManagerService</code>	电源管理服务	1.2/3
<code>ActivityManagerService</code>	最核心的服务之一，管理 Activity	自定义
<code>TelephonyRegistry</code>	通过该服务注册电话模块的事件响应，比如重启、关闭、启动等	1.0
<code>PackageManagerService</code>	程序包管理服务	3.3
<code>AccountManagerService</code>	账户管理服务，是指联系人账户，而不是 Linux 系统的账户	1.0
<code>ContentService</code>	<code>ContentProvider</code> 服务，提供跨进程数据交换	3.0
<code>BatteryService</code>	电池管理服务	1.0
<code>LightsService</code>	自然光强度感应传感器服务	1.0
<code>VibratorService</code>	震动器服务	1.0
<code>AlarmManagerService</code>	定时器管理服务，提供定时提醒服务	1.0
<code>WindowManagerService</code>	Framework 最核心的服务之一，负责窗口管理	3.3
<code>BluetoothService</code>	蓝牙服务	1.0 +
<code>DevicePolicyManagerService</code>	提供一些系统级别的设置及属性	1.3
<code>StatusBarManagerService</code>	状态栏管理服务	1.3
<code>ClipboardService</code>	系统剪切板服务	1.0
<code>InputMethodManagerService</code>	输入法管理服务	1.0
<code>NetStatService</code>	网络状态服务	1.0
<code>NetworkManagementService</code>	网络管理服务	<code>NMS.create()</code>
<code>ConnectivityService</code>	网络连接管理服务	2.3
<code>ThrottleService</code>	暂不清楚其作用	1.3

(续表)

服务类名称	作用描述	启动模式
AccessibilityManagerService	辅助管理程序截获所有的用户输入，并根据这些输入给用户一些额外的反馈，起到辅助的效果	1.0
MountService	挂载服务，可通过该服务调用 Linux 层面的 mount 程序	1.0
NotificationManagerService	通知栏管理服务，Android 中的通知栏和状态栏在一起，只是界面上前者在左边，后者在右边	1.3
DeviceStorageMonitorService	磁盘空间状态检测服务	1.0
LocationManagerService	地理位置服务	1.3
SearchManagerService	搜索管理服务	1.0
DropBoxManagerService	通过该服务访问 Linux 层面的 Dropbox 程序	1.0
WallpaperManagerService	墙纸管理服务，墙纸不等于桌面背景，在 View 系统内部，墙纸可以作为任何窗口的背景	1.3
AudioService	音频管理服务	1.0
BackupManagerService	系统备份服务	1.0
AppWidgetService	Widget 服务	1.3
RecognitionManagerService	身份识别服务	1.3
DiskStatsService	磁盘统计服务	1.0

AmS 的启动模式如下：

- ① 调用 main()函数，返回一个 Context 对象，而不是 AmS 服务本身。
- ② 调用 AmS.setSystemProcess()。
- ③ 调用 AmS.installProviders()。
- ④ 调用 systemReady()，当 AmS 执行完 systemReady()后，会相继启动相关联服务的 systemReady()

函数，完成整体初始化。

关于具体某个服务的内部启动过程，请参照源码，这些过程一般都比较简单。

9.4.2 启动第一个 Activity

当以上服务线程都启动后，其中 ActivityManagerService (AmS) 服务是以 systemReady()调用完成最后启动的，而在 AmS 的 systemReady()函数内部的最后一段代码则发出了启动任务队列中最上面一个 Activity 的消息，如以下代码所示：

```
6294 mMainStack.resumeTopActivityLocked(null);
```

由于系统刚启动时，mMainStack 队列中并没有任何 Activity 对象，因此，ActivityStack 类中将调用 startHomeActivityLocked()函数，如以下代码所示：

```

1053     ActivityRecord next = topRunningActivityLocked(null);
1054
1055     // Remember how we'll process this pause/resume situation,
1056     // that the state is reset however we wind up proceeding.
1057     final boolean userLeaving = mUserLeaving;
1058     mUserLeaving = false;
1059
1060     if (next == null) {
1061         // There are no more activities! Let's just start up
1062         // Launcher...
1063         if (mMainStack) {
1064             return mService.startHomeActivityLocked();
1065         }
1066     }

```

从这里可以看出，开机后系统从哪个 Activity 开始执行完全取决于 mMainStack 中的第一个 Activity 对象。可以想象，如果在 ActivityManagerService 启动时能够构造一个 Activity 对象（并不是说构造出一个 Activity 类的对象），并将其放到 mMainStack 中，那么第一个运行的 Activity 就是这个 Activity，这一点不像其他操作系统中通过设置一个固定程序作为第一个启动程序。

在 AmS 的 startHomeActivityLocked() 中，系统发出了一个 category 字段包含 CATEGORY_HOME 的 intent，如以下代码所示：

```

1964         intent.setComponent(mTopComponent);
1965         if (mFactoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
1966             intent.addCategory(Intent.CATEGORY_HOME);
1967         }

```

无论是哪个应用程序，只要声明自己能够响应该 intent，那么就可以被认为是 Home 程序，这就是为什么在 Android 领域会存在各种所谓的“Home 程序”的原因。系统并没有给任何程序赋予“Home”特权，而把这个权利交给了用户，当系统中有多个程序能够响应该 intent 时，系统会弹出一个对话框，请求用户选择启动哪个程序，并允许用户记住该选择，从而使得以后每次按“Home”键后都启动相同的 Activity。

这就是第一个 Activity 的启动过程。



第 10 章 AmS 内部原理

ActivityManagerService.java 文件，简称 AmS，一共有 14684 行代码，是 Android 内核的三大核心功能之一，另外两个是 WindowManagerService.java 和 View.java。

尽管 AmS 的代码庞大，逻辑纷繁，但从所提供的功能来看，却没有看上去那么复杂。笔者分析 AmS 源码时，首先猜想并验证 AmS 所提供的各种功能，然后根据这些功能再去找相应的代码，从而能够对该文件进行有效的逻辑分隔。

AmS 所提供的主要功能包括以下几项：

- 统一调度各应用程序的 Activity。应用程序要运行 Activity，会首先报告给 AmS，然后由 AmS 决定该 Activity 是否可以启动，如果可以，AmS 再通知应用程序运行指定的 Activity。换句话说，运行 Activity 是各应用进程的“内政”，AmS 并不干预，但是 AmS 必须知道各应用进程都运行了哪些 Activity。
- 内存管理。Android 官方声称，Activity 退出后，其所在的进程并不会被立即杀死，从而下次在启动该 Activity 时能够提高启动速度。这些 Activity 只有当系统内存紧张时，才会被自动杀死，应用程序不用关心这个问题。而这些正是在 AmS 中完成的。
- 进程管理。AmS 向外提供了查询系统正在运行的进程信息的 API。这些都比较简单。

下面就根据这些功能点，来分析 AmS 的代码。

10.1 Activity 调度机制

在 Android 中，Activity 调度的基本思路是这样的：各应用进程要启动新的 Activity 或者停止当前的 Activity，都要首先报告给 AmS，而不能“擅自处理”。AmS 在内部为所有应用进程都做了记录，当

AmS 接到启动或停止的报告时，首先更新内部记录，然后再通知相应客户进程运行或者停止指定的 Activity。由于 AmS 内部有所有 Activity 的记录，也就理所当然地能够调度这些 Activity，并根据 Activity 和系统内存的状态自动杀死后台的 Activity。

具体来讲，启动一个 Activity 有以下几种方式。

- 在应用程序中调用 `startActivity()` 启动指定的 Activity。
- 在 Home 程序中单击一个应用图标，启动新的 Activity。
- 按 “Back” 键，结束当前 Activity，自动启动上一个 Activity。
- 长按 “Home” 键，显示出当前任务列表，从中选择一个启动。

这四种启动方式的主体处理流程都会按照第一种启动方式运行，后面三种方式只是在前端消息处理上各有不同，因此，后面首先介绍第一种启动方式，然后介绍其他启动方式的前端处理差异。

10.1.1 几个重要概念

AmS 中定义了几个重要的数据类，分别用来保存进程（Process）、活动（Activity）和任务（Task）。

1. 进程数据类 ProcessRecord

该类在 `framework/base/services/java/com/android/server/am/` 路径下，该路径最后的 `am` 代表 Activity Manager，和 AmS 有关的重要类都在该目录下。

一个 APK 文件运行时会对应一个进程，当然，多个 APK 文件也可以运行在同一个进程中。ProcessRecord 正是记录一个进程中的相关信息，该类中内部变量可分为三个部分，大家先不用琢磨具体某个变量如何被使用，而只需要先了解它们的作用。这三个部分如表 10-1 所示。

表 10-1 ProcessRecord 类内部信息

变量作用	主要包含
进程文件信息	也就是与该进程对应的 APK 文件的内部信息，比如 ApplicationInfo info; String processName; HashSet<String> pkgList; //保存该进程中的所有 APK 文件包名
该进程的内存状态信息	这些信息将用于 Linux 系统的 Out Of Memory(OOM)情况的处理，当发生系统内部不够用时，Linux 系统会根据进程的内存状态信息，杀掉优先级比较低的进程。 int maxAdj; int curAdj; ... 变量中 adj 的含义是 adjustment，即“调整值”
进程中包含的 Activity、Provider、Service 等	ArrayList activities; ArrayList services; ...



2. HistoryRecord 数据类

AmS 中使用 HistoryRecord 数据类来保存每个 Activity 的信息，有些读者可能奇怪，Activity 本身也是一个类啊，为什么还要用 HistoryRecord 来保存 Activity 的信息，而不直接使用 Activity 呢？因为，Activity 是具体的功能类，这就好比每一个读者都是一个 Activity，而“学校”要为每一个读者建立一个档案，这些档案中并不包含每个读者具体的学习能力，而只是学生的籍贯信息、姓名、出生年月、家庭关系等。HistoryRecord 正是 AmS 为每一个 Activity 建立的档案，该数据类中的变量主要包含两部分，如表 10-2 所示。

表 10-2 HistoryRecord 类内部信息

变量作用	主要包含
环境信息	该 Activity 的工作环境，比如，隶属于哪个 Package，所在的进程名称、文件路径、数据路径、图标、主题，等等，这些信息基本上是固定的
运行状态信息	比如 idle、stop、finishing 等，这些变量一般为 boolean 类型，这些状态值与应用程序中所说的 onCreate、onPause、onStart 等状态有所不同

需要注意，HistoryRecord 类也是一个 Binder，它基于 IApplicationToken.Stub 类，因此，可以被 IPC 调用，一般是在 WmS 中进行该对象的 IPC 调用。

3. TaskRecord 类

AmS 中使用任务的概念确保 Activity 启动和退出的顺序。比如以下启动流程，A、B、C 分别代表三个应用程序，数字 1、2、3 分别代表该应用中的 Activity。

A1→A2→A3→B1→B2→C1→C2，此时应该处于 C2，如果 AmS 中没有任务的概念，此时又要从 C2 启动 B1，那么会存在以下两个问题：

- 虽然程序上是要启动 B1，但是用户可能期望启动 B2，因为 B1 和 B2 是两个关联的 Activity，并且 B2 已经运行于 B1 之后。如何提供给程序员一种选择，虽然指定启动 B1，但如果 B2 已经运行，那么就启动 B2。
- 假设已经成功从 C2 跳转到 B2，此时如果用户按“Back”键，是应该回到 B1 呢，还是应该回到 C2？

任务概念的引入正是为了解决以上两个问题，HistoryRecord 中包含一个 int task 变量，保存该 Activity 所属哪个任务，程序员可以使用 Intent.FLAG_NEW_TASK 标识告诉 AmS 为启动的 Activity 重新创建一个 Task。

有了 Task 的概念后，以上情况将会是这样的：

虽然程序明确指定从 C2 启动到 B1，程序员可以在 intent 的 FLAG 中添加 NEW_TASK 标识，从而使得 AmS 会判断 B1 是否已经在 mHistory 中。如果在，则找到 B1 所在的 Task，并从该 Task 中的最上面的 Activity 处运行，此处也就是 B2。当然，如果程序的确要启动 B1，那么就不要再使用 NEW_TASK 标识，使用的话，mHistory 中会有两个 B1 记录，隶属于不同的 Task。



TaskRecord 类中的变量如表 10-3 所示。

表 10-3 TaskRecord 类内部变量

变量名称	描 述
int taskId;	每一个任务对应一个 Int 型的标识
Intent intent;	创建该任务时对应的 intent
int numActivities;	该任务中的 Activity 数目

需要注意的是, TaskRecord 中并没有该任务中所包含的 Activity 列表, 比如 `ArrayList<HistoryRecord>` 或者 `HistoryRecord[]` 之类的变量, 这意味着不能直接通过任务 id 找到其所包含的 Activity。要达到这个目的, 可以遍历 AmS 中 mHistory 中的全部 HistoryRecord, 然后根据每一个 HistoryRecord 中的 TaskRecord task 变量确定是否属于指定的任务。

10.1.2 AmS 中的一些重要调度相关变量

要了解 AmS 调度、管理系统中的 Activity 的细节, 必须了解 AmS 中定义的重要内部变量。要一下了解这些变量的使用时机并非易事, 因此, 本节仅简要说明一些变量的作用, 至于具体使用的时机, 要结合调度的具体过程了解。

1. 系统常量

- `static final int MAX_ACTIVITIES = 20;`

系统只能有一个 Activity 处于执行状态, 对于非执行状态的 Activity, AmS 会在内部暂时缓存起来, 而不是立即杀死, 但如果后台的 Activity 数目超过该常量, 则会强制杀死一些优先级较低的 Activity, 所谓的“优先级高低”的规则见第 10.2 节。

- `static final int MAX_RECENT_TASKS = 20;`

AmS 会记录最近启动的 Activity, 但只记录 20 个, 超过该常量后, AmS 会舍弃最早记录的 Activity。

- `static final int PAUSE_TIMEOUT = 500;`

当 AmS 通知应用进程暂停指定的 Activity 时, AmS 的忍耐是有限的, 因为只有 500 毫秒, 如果应用进程在该常量时间内还没有暂停, AmS 会强制暂停关闭该 Activity。这就是为什么在应用程序设计时, 不能在 `onPause()` 中做过多事情的原因。

- `static final int LAUNCH_TIMEOUT = 10*1000;`

当 AmS 通知应用进程启动 (Launch) 某个 Activity 时, 如果超过 10s, AmS 就会放弃。

- `static final int PROC_START_TIMEOUT = 10*1000;`

当 AmS 启动某个客户进程后, 客户进程必须在 10 秒之内报告 AmS 自己已经启动, 否则 AmS 会认为指定的客户进程不存在。

2. 等待序列

由于 AmS 采用 Service 机制运作，所有的客户进程要做什么事情，都要先请求 AmS，因此，AmS 内部必须有一些消息序列保存这些请求，并按顺序依次进行相应的操作。

- `final ArrayList mHistory = new ArrayList();`

这是最最重要的内部变量，该变量保存了所有正在运行的 Activity，所谓正在运行是指该 HistoryRecord 的 finishing 状态为 true。比如当前和用户交互的 Activity 属于正在运行，从 A1 启动到 A2，尽管 A1 看不见了，但是依然是正在运行，从 A2 按“Home”键回到桌面，A2 也是正在运行，但如果从 A2 按“Back”键回到 A1，A2 就不是正在运行状态了，它会从 mHistory 中删除掉。

- `private final ArrayList mLRUActivities = new ArrayList();`

LRU 代表 Latest Recent Used，即最近所用的 Activity 列表，它不像 mHistory 仅保存正在运行的 Activity，mLRUActivities 会保存所有过去启动过的 Activity。

- `final ArrayList<PendingActivityLaunch> mPendingActivityLaunches
= new ArrayList<PendingActivityLaunch>();`

当 AmS 内部还没有准备好时，如果客户进程请求启动某个 Activity，那么会被暂时保存到该变量中，这也就是 Pending 的含义。这种情况一般发生在系统启动时，系统进程会查询系统中所有属性为 Persisitant 的客户进程，此时由于 AmS 也正在启动，因此，会暂时保存这些请求。

- `final ArrayList<HistoryRecord> mStoppingActivities
= new ArrayList<HistoryRecord>();`

在 AmS 的设计中，有这样一个理念：优先启动，其次再停止。即当用户请求启动 A2 时，如果 A1 正在运行，AmS 首先会暂停 A1，然后启动 A2，当 A2 启动后再停止 A1。在这个过程中，A1 会被临时保存到 mStoppingActivities 中，知道 A2 启动后并处于空闲时，再回过头来停止 mStoppingActivities 中保存的 HistoryRecord 列表。

- `final ArrayList mFinishingActivities = new ArrayList();`

和 mStoppingActivities 有点类似，当 AmS 认为某个 Activity 已经处于 finish 状态时，不会立即杀死该 Activity，而是会保存到该变量中，直到超过系统设定的警戒线后，才去回收该变量中的 Activity。

3. 当前不同状态的 HistoryRecord

- `HistoryRecord mPausingActivity = null;`

正在暂停的 Activity，该变量只有在暂停某个 Activity 时才有值，代表正在暂停的 Activity。

- `HistoryRecord mResumedActivity = null;`

当前正在运行的 Activity，这里的正在运行并不见得一定是正在与用户交互。比如当用户请求执行 A2 时，当前正在运行 A1，此时 AmS 会首先暂停 A1，而在暂停的过程中，AmS 会通知 WmS 暂停获取用户消息，而此时 mResumedActivity 依然是 A1。

- `HistoryRecord mFocusedActivity = null;`



这里的 Focus 并非是正在和用户交互，而是 AmS 通知 WmS 应该和用户交互的 Activity，而在 WmS 真正处理这个消息之前，用户还是不能和该 Activity 交互。

- HistoryRecord mLastPausedActivity = null;
上一次暂停的 Activity。

10.1.3 startActivity()的流程

程序员可以直接调用 startActivity()启动指定的 Activity。前面说过，尽管从用户的角度来看，启动 Activity 有不同的方式，但是其主体流程是完全相同的，前端各种交互方式最后都会调用到 startActivity() 启动。因此，首先介绍 startActivity() 的流程。

1. 概述

调用流程可大致归结为如图 10-1 和图 10-2 所示，由于版面限制，该图被划分为两部分。

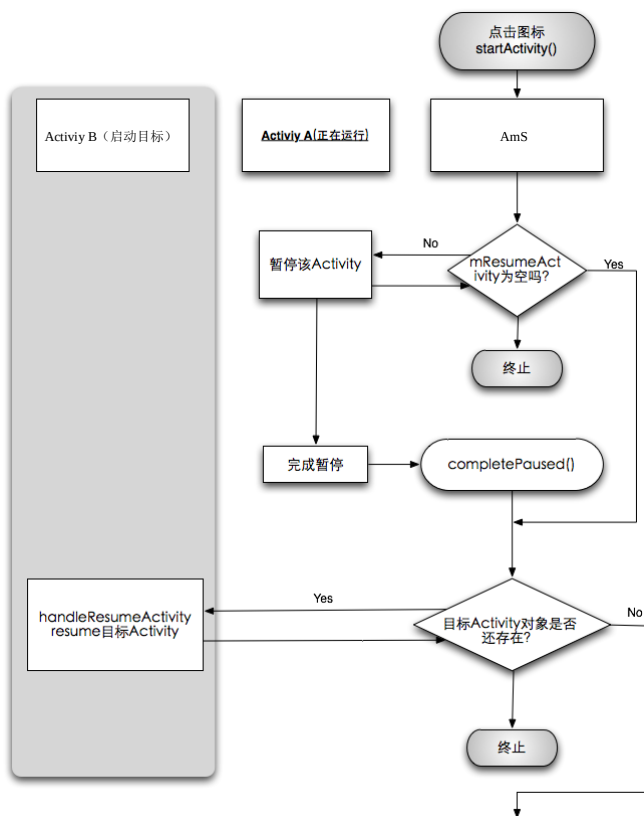


图 10-1 startActivity()的内部调用流程

一般调用 `startActivity()` 方法是从用户单击桌面图标开始的，经过各种调用，最后导致调用 `startActivity()`。本例假设当前正在运行 A，而单击图标后会运行 B。

AmS 收到客户请求 `startActivity()` 后，会首先暂停当前的 Activity，因此要判断 `mResumedActivity` 是否为空。在一般情况下，该值都不为空，如果不为空，AmS 会通知 A 所在客户进程暂停，执行该 Activity，并立即返回，返回后 AmS 就又“睡觉”去了。读者可能觉得奇怪，AmS “睡觉”去了，谁来启动 B 呢？这正是 AmS 的异步机制的好处，异步机制允许执行完一个命令后就去休息，而当远程执行完某个命令后再回调。这种方式广泛应用于 Android 的内核实际中，大家要迅速解除这种疑惑。

当 A 进程完成暂停后，报告 AmS，这时 AmS 开始执行 `completePaused()`。该方法中先要检查目标 Activity 是否在 `mHistory` 列表中，如果在，说明目标进程还在运行，目标 Activity 只是处于 `stop` 状态，还没有 `finish`，所以通知 B 进程直接 `resume` 指定的 Activity 即可。如果不在 `mHistory` 列表中，则继续执行如图 10-2 所示的流程。

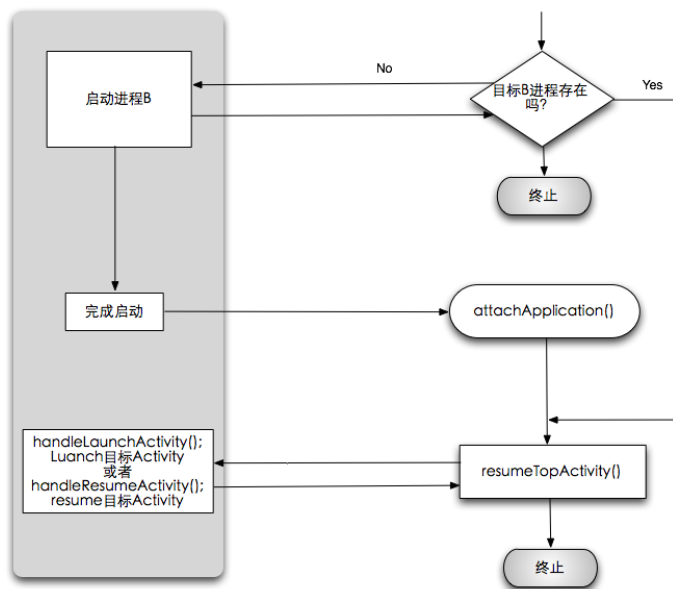


图 10-2 `startActivity()` 的内部调用流程（续）

检查目标 Activity 所在的进程是否存在，如果不存在则必须首先启动对应的进程。当对应的进程启动后，B 进程会报告 AmS 自己已经启动，于是执行 AmS 的 `attachApplication()` 方法，该方法可理解为 B 进程请求 AmS 给自己安排(`attach`)一个具体要执行的 Activity，此时 AmS 继续调用 `resumeTopActivity()`，通知 B 进程执行指定的 Activity。如果指定的 `HistoryRecord` 在 B 进程中不存在，则 B 调用 `handleLaunchActivity()` 创建一个该 Activity 实例；如果已经存在，则调用 `handleResumeActivity()` 恢复已有的 Activity 的运行。这个逻辑的意思就是说，在 `ActivityThread` 中可以存在同一个 Activity 的多个实例，对应了 AmS 中 `mHistory` 的多个 `HistoryRecord` 对象。在一般情况下，当调用 `startActivity(intent)` 的

FLAG 为 NEW_TASK 时，AmS 会首先从 mHistory 中找到指定 Activity 所在的 Task，然后启动 Task 中的最后面一个 Activity；如果 FLAG 不为 NEW_TASK，那么 AmS 会在当前 Task 中重新创建一个 HistoryRecord，这就有可能创建同一个 Activity 的多个 HistoryRecord 对象，在 ActivityThread 端就可能创建同一个 Activity 类的多个实例。

2. 单击图标的过程

关于 startActivity() 的细节过程如附图 3 所示，本图绘制时遵循以下规则：

假设 class A 的内部函数关系如下示意代码。

```
public class Persedu {
    void f1(){
        f2();
    }

    void f2(){
        f3();
    }

    void f3(){
        x1();
        x2();
        x3();
    }

    void x1(){};
    void x2(){};
    void x3(){};
}
```

该类中的函数调用有两个特点，f1 调用 f2，f2 又调用 f3，属于垂直调用，而在 f3 中先后调用 x1、x2、x3，这属于包含式调用。为了区分这两种调用，前者在图中以垂直方式表示，而后者以水平方式表示，如图 10-3 所示。

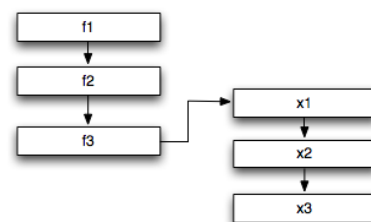


图 10-3 函数调用关系绘图规则示例

startActivity() 的细节过程可分为七步，将分节介绍，首先从用户单击图标开始。

当用户单击某个应用图标后，执行程序会在该图标的 onClick 事件中调用 startActivity() 方法，该方法属于 Activity 类的内部方法，然后该方法会调用 startActivityForResult()，调用时自动把第二个参数设为 -1。所以，startActivity() 和 startActivityForResult() 两者没有本质区别。

在 forResult 方法内部，会调用 Instrumentation 对象的 executeStartActivity() 方法，该对象是在 ActivityThread 创建 Activity 时创建的，一个应用程序中只有一个 Instrumentation 对象，每个 Activity 内部都有一个该对象的引用。Instrumentation 可以理解为应用进程的管家，ActivityThread 要创建或者暂停某个 Activity 时，是通过这个“管家”进行的，设置这个管家的好处是可以统计所有的“开销”，开销

的信息保存在“管家”那里。其实正如其名称所示，Instrumentation 就是为了“测量”、“统计”，因为管家掌握了所有的“开销”，自然也就具有了统计的功能。当然，Instrumentation 类和 ActivityThread 的分工是有本质区别的，后者就像是这个家里的主人，负责创建这个“家庭”，并负责和外界打交道，比如接收 AmS 的通知等。

在 Instrumentation 中，则继续调用 AmS 的 startActivity()方法，从而真正把启动的意图传递给 AmS。Android 中 IPC 调用是同步的，只有当 AmS 执行完 startActivity()方法后才返回，而在此期间，Instrumentation 会一直处于线程等待状态。由于一个应用进程一般只有一个主线程，这就意味着应用主线程在这期间会一直等待，所幸的是 AmS 执行 startActivity()是异步的，基本上会在很短的时间内返回，然后使用异步通知机制，回调 caller 的 onActivityResult()方法并返回执行结果。这就是为什么在 AmS 的 startActivity()参数中需要包含一个 IBinder 类型的 token。该参数来源于 Activity 对象内部的 mToken 变量，而这个变量对应的是 AmS 中的一个 HistoryRecord，AmS 正是通过 HistoryRecord 来识别客户进程中的 Activity 的。当 AmS 执行完指定的 Activity 后，如果 caller 需要 forResult，AmS 就会从 HistoryRecord 中取出 HistoryRecord.app.thread 变量，该变量的类型是 ActivityThread.ApplicationThread 子类，并调用该类的 scheduleSendResult(r, list)方法，该方法中的 r 代表了 HistoryRecord。ActivityThread 收到这个调用后，将根据参数 r 判断属于哪个 Activity，然后调用相应 Activity 的 onActivityResult()方法。这个过程如图 10-4 所示。

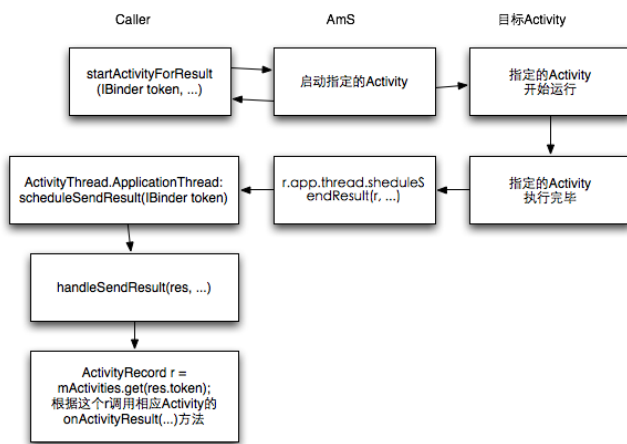


图 10-4 startActivityForResult()的执行过程

3. 运行环境检查

当 AmS 收到启动某个 Activity 的意图 (intent) 后，首先要检查一下环境，比如 Caller 是否有启动的权限、意图是否对应存在的 Activity 等。这就相当于去医院看病时的导医一样，先进行一些简单的检查，如果不满足条件，则立即返回。

这些具体的检查请参照图 10-4 所示。包括如下：



首先，根据 intent 的信息找到匹配的 Component 信息，这就是为什么 intent 中可直接指定 Activity 名称的原因，AmS 会调用系统内部的 PackageManager 查询具体的 Component 名称。当然，如果没有 Component 匹配该 intent，则返回失败。

接着调用 startActivityLocked(caller, intent, ...)方法，AmS 中有两个同名该方法，注意其使用的时机和作用，以下篇幅以不同的参数类型表示不同的方法。该方法的后缀是 Locked，意思是该方法必须是线程安全的，或者叫不可重入的，即该函数体只能由一个线程调用，另一个线程也要调用该函数时，必须等待前一个线程执行完。

在 startActivityLocked(caller, intent, ...)的包含调用中，主要做了以下几件事情。

① 检查当前正在运行的 Activity 是否和目标 Activity 一致，如果一致，则直接返回。这就是说，程序员在 Activity 使用 startActivity 启动自己，不会达到重启当前 Activity 的目的。

② 处理 INTENT_FLAG_FORWARD_RESULT 标志。从代码的角度来看，这个标志有一个特殊的作用，就是能够跨 Activity 传递 Result。比如 A1→A2，此时如果从 A2 中启动 A3，并且设置的启动标志为 FORWARD_RESULT，那么 A3 运行时，可以在 A3 中调用 setResult，然后 finish()，其结果会从 A3 直接返回到 A1，并且 A1 会得到 A3 所 set 的 result。要满足这种调用，必须使用以下方式启动。

A1(startActivityForResult) → A2(StartActivity) → A3。注意 A2 不能使用ForResult 的方式启动 A3，否则会发生冲突 START_FORWARD_AND_REQUEST_CONFLICT。

③ 在此检查 Caller 的 app 是否存在，这种情况发生在发出 startActivity 命令后，Caller 所在的进程被意外杀死，如果是这样，AmS 拒绝继续往下执行。

④ 检查 Caller 是否具备启动指定 Activity 的权限。

⑤ 如果 AmS 中有 IActivityController 对象，则通知该 Controller 进行相应的控制。在一般情况下，mController 始终不存在，可能是 Android 为了某种系统测试而设置的 debug 开关。

⑥ 创建一个临时的 HistoryRecord 对象，该对象只是为了后面调用过程中的各种对比，不一定会最终被加入到 mHistory 列表中。

⑦ 检查当前是否允许切换 Activity，不允许切换的情况一般发生在当前已经暂停的正在执行的 Activity，正在等待下一个 Activity 的启动时，此时不能进行 Activity 切换。如果是这样，则把指定的 Activity 临时添加到 mPendingActivityLaunches 列表中，等待系统恢复后再继续执行。

⑧ 判断 mPendingActivityLaunches 列表中是否有等待的 Activity 要启动，如果有的话，应当先运行这行等待的 Activity。

⑨ 如果等待序列为空，则调用 startActivityUncheckedLocked()方法。此时，要启动 Activity 已经通过重重检验，被确认为是一个“正当”的启动请求。接下来，AmS 就会开始判断以何种方式来启动指定的 Activity。

4. 找到或创建合适的 Task

前面说过，Task 的作用是确保 Activity 按照指定的方式退出，并当用户按“Back”键时按照指定的方式执行下一个 Activity。在真正启动目标 Activity 之前，AmS 先要确定是否需要为目标 Activity 创建

一个新的 Task，过程如下。

① 处理 `FLAG_ACTIVITY_NO_USER_ACTION` 标识。在一般情况下，启动一个 Activity 时都不使用该标识，如果不包含该标识，AmS 会判断一定的时间内是否有用户交互。如果没有用户交互的话，AmS 会通知 Activity 回调 `onUserLeaving()` 方法，然后再回调 `onPause()` 方法，如果使用了该标识，说明目标 Activity 不和用户交互，所以也就不需要回调 `onUserLeaving()` 方法。

② 确定是否现在就启动，在一般情况下都是立即启动。如果立即启动，就把 `r.delay Resume` 为 `true`，意思是不要延迟启动。

③ 处理 `FLAG_ACTIVITY_PREVIOUS_IS_TOP` 标志，这个标识基本无用。接着再处理 `onlyIfNeed` 参数，这个参数会和该标志一起使用，不过都是基本无用。

④ 为目标 Activity 赋予合法的权限。目标 Activity 在安装时会指定所需要的权限，此处正是把这些允许的权限添加到缓存中。

⑤ 根据 `launchMode` 变量设置局部变量 `launchFlags`，因为接下来要根据 `launchFlags` 决定启动 Activity 的方式。注意 `launchMode` 和 `launchFlags` 是两码事，`mode` 是 Activity 自己声明的运行方式，它是在 `AndroidManifest.xml` 中指定的；而 `flag` 是调用者希望以何种方式运行指定的 Activity，是在调用 `startActivity()` 时参数 `intent` 中指定的。

⑥ 对 `NEW_TASK` 和 `SINGLE_TASK` 标识以及 `SINGLE_INSTANCE` 模式进行处理。请注意，这两个标志必须在 `r.resultTo==0` 的条件中，即这两个标识只能用在 `startActivity()` 的方法中，而不能用在 `startActivityForResult()` 方法中。因为从 Task 的角度来看，Android 认为不同 Task 之间的 Activity 是不能传递数据的，所以不能使用 `NEW_TASK` 标识，但是还要调用 `forResult()` 方法。

`NEW_TASK`、`SINGLE_TASK` 以及 `SINGLE_INSTANCE` 三者的作用如图 10-5 所示。

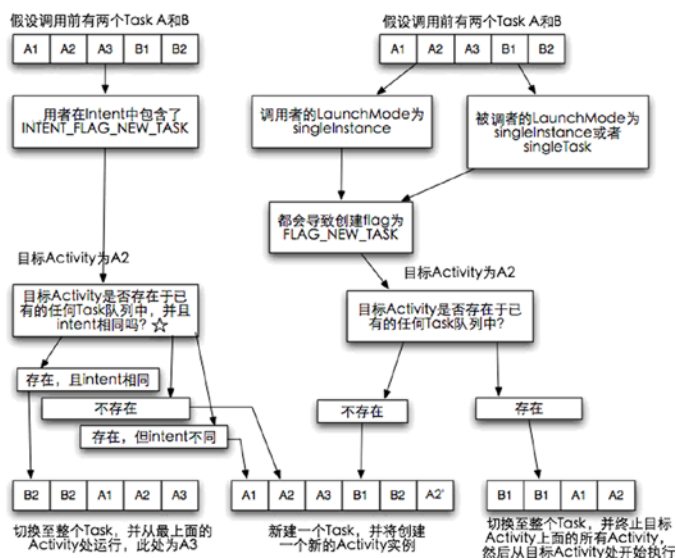


图 10-5 Activity 启动模式

上图中注意“☆”符号处所讲的“intent 相同”的含义，它是指想要启动的 Activity 对应的 intent 和 Task 中第一个 Activity 启动时所使用的 Intent。系统认为，只有 Intent 相同才意味着用户真的是想启动已有的 Task，否则应该创建一个新的 Task。“Intent 相同”具体包含了 intent 中的 Action 字段、Category 字段、data 字段、ComponentName 字段等都要相等。

SINGLE_TASK（简称 ST）和 SINGLE_INSTANCE（简称 SI）的相同点在于，如果目标 Activity 不存在，则两者都会创建一个新的 Task，而如果目标 Activity 存在，则两者都会切换到已有的 Task，并终止目标 Activity 上面的所有 Activity，并从 Activity 处开始执行。

ST 和 SI 的区别有两点：第一，调用者本身如果是 SI，那么肯定会创建一个新的 Task，无论目标 Activity 是否存在，但对于 ST，就没有这么“极端”；第二，SI 所在的 Task 只会有一个 Activity 对象，就是这个 SI，而 ST 中可以包含多个 Activity，但 ST 所对应的 Activity 肯定是根 Activity。

⑦ 如果上一步发现可以从 History 中 resume 一个已经存在的 HistoryRecord 对象，那么就不会执行到这一步。如果找不到，则先判断是否需要添加一个新的 Task，如果需要则创建，并把指定的 Activity 添加到新 Task 中；如果不需要创建一个新的 Task，那么就在当前的 Task 中添加目标 HistoryRecord。

如果要在当前 Task 中添加 HistoryRecord 对象，则要处理以下两个启动标识，这两个标识仅适用于已有的 Task 中包含目标 Activity。

- FLAG_ACTIVITY_CLEAR_TOP：如果指定的 Activity 已经存在，则清空其上面的 Activity。
- FLAG_ACTIVITY_REORDER_TO_FRONT：该标识没有 CLEAR_TOP 那么严厉，它只是把自己放到最上面，而不清空其他已有的。

⑧ 解决了 Task 的问题，并且对 mHistory 列表中的顺序已经作了应有的调整，接下来，就可以按照常规方式去启动相应的 Activity 了。

读者可能会觉得以上八步在代码实现上有些复杂，因此，以下给出另一种思路理解这些代码。

这八步的最终目的是调整 mHistory 中 Activity 的顺序，在调整时，Android 的设计理念遵守了一个简单的原则。即如果是跨 Task 切换，调用者是没有权利更改被调 Task 里面的 Activity 顺序的，除非被调者自己声明了某些属性。这个原则是合理的，因为只有用户有权利去更改自己当前所操作的 Task 中的 Activity 顺序。举例来讲，假设 mHistory 中的 Activity 顺序如下：

A1→A2→A3→B1→B2→B3→C1→C2→C3→C4。其中，A、B、C 代表三个不同的 Task，此时正在运行 C4，如果程序此时需要从 C 切换到 B 中的任何一个 Activity，切换后的顺序只能是：

A1→A2→A3→C1→C2→C3→C4→B1→B2→B3→B4。程序不能对 B 中的 Activity 顺序进行更改，但程序却可以对 C 中的 Activity 的顺序进行调整，因为此时用户正在和 C 交互，系统认为在当前交互的任务中切换 Activity 是合法的。

系统提供了两种方式完成以上的切换，第一种是在 AndroidManifest.xml 文件中声明 Activity 自身的启动属性，另一种是在启动时给 intent 中添加不同的 flag。前者包括：

- android:launchMode=standard/singleTop/singleTask/singleInstance
- android:clearTaskOnLaunch=true/false



- android:finishonTaskLaunch=true/false
- android:allowTaskReparent=true/false

后者包括：

- Intent.FLAG_ACTIVITY_NEW_TASK
- Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED
- Intent.FLAG_ACTIVITY_CLEAR_TOP
- Intent.FLAG_ACTIVITY_REORDER_TO_FRONT
- Intent.FLAG_ACTIVITY_NO_HISTORY
- Intent.FLAG_ACTIVITY_SINGLE_TOP

这些设置表面上看起来挺复杂，实际上按功能可分为三类，从而理解起来很简单。第一类是为了完成在 Task 之间切换，第二类是为了完成在当前 Task 中改变 Activity 的顺序，第三类是为了在 Task 切换时 Task 内部自动重排所属的 Activity。首先来看第一类。

第一类：在 Task 之间切换。如上面例子，系统允许程序从 B 任务切换到 C 任务，具体实现就是在启动的 intent 中设置 FLAG_ACTIVITY_NEW_TASK。如果 intent 所匹配到的 Activity 已经存在于已有的 Task 中，那么就切换 Task；如果没有，则创建一个新的 Task。而这只是一种实现方式，实际的需求有时是调用者希望被调者出现在一个新的 Task 中，有时则是被调者要求自己必须在一个新的 Task 中，对于后者，则使用 launchMode 来完成，如图 10-6 所示。

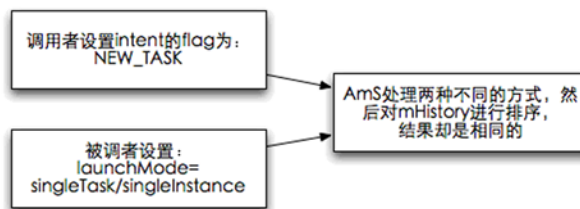


图 10-6 使用 launchMode 启动 Activity

第二类：在同一个 Task 中调整 Activity 的顺序，由于这些 FLAG 是用于在当前 Task 中切换的，因此不能和 NEW_TASK 标识一起使用。这类常见的包括以下几种。

- CLEAR_TOP，作用是清除目标 Activity 上面的 Activity。比如 C1-C2-C3，如果此时要启动 C2，结果将是 C1-C2，C3 会被结束掉。
- REORDER_TO_FRONT，简单重排，结果会是 C1-C3-C2。
- NO_HISTORY，目标 Activity 不出现在 mHistory 中。比如 C1-C2，此时如果要启动 C3，而在 C3 中又启动了 C4，那么 mHistory 中将会是 C1-C2-C4，C4 返回后将从 C2 处继续执行。

第三类：允许被调 Task 在启动前主动重排。首先调用者要发起这个请求，其次是被调者要有这个



功能。发起是通过设置 intent 标识为 RESET_IF_NEEDED 实现的，而功能是在 manifest 文件中使用 android:clearTaskOnLaunch 和 android:finishOnTaskLaunch 属性，其具体的作用如其名称所示，此处不再赘述。

本节内容所对应的 Android 源码正是以上三类功能的代码实现。本节代码执行完毕后，无论是否需要创建新的 Task，也无论是否需要创建新的 HistoryRecord 对象，mHistory 列表中已经包含了这些新的 Task 或者新的 HistoryRecord，下一节中将会从 mHistory 中取出第一个 HistoryRecord 并尝试去执行，尽管该 HistoryRecord 对应的实际 Activity 对象可能还不存在，但这个环节已经结束。

5. 运行 mHistory 中最后一个 HistoryRecord

上一节执行完后，调用 resumeTopActivity(null) 启动真正的 Activity。

- ① 调用 topRunningActivityLocked() 方法取出当前正在运行的 HistoryRecord 对象。
- ② 判断 mHistory 中是否有记录，如果没有意味着还从未运行任何 Activity，需要首先调用 startHomeActivityLocked() 方法启动所谓的“主界面程序”。
- ③ 判断正在执行的 Activity 是否和目标 Activity 一样，如果一样，则万事大吉。
- ④ 判断当前系统是否处于睡眠状态，如果是，也只能作罢，否则继续执行。
- ⑤ 从 mStoppingActivities 和 mWaitingVisibleActivities 列表中删除目标对象，因为目标对象接下来就会被启动。
- ⑥ 检查当前是否正在暂停某个其他的 Activity，如果是则还不能运行。
- ⑦ 检查当前是否有正在运行的 Activity，一般情况下都会有，所以需要首先暂停当前 Activity。这一点也是手机系统和普通桌面系统的主要差别之一，Android 仅允许同时执行一个 Activity，而桌面系统则可以同时打开多个应用窗口。此时，如果需要暂停当前 Activity，则 AmS 的执行会暂时告一段落，剩下的就是等待被暂停的客户进程报告暂停完毕。
- ⑧ 执行到这一步实际上并不是从 startActivity() 调用过来的，而是从另外两种情况过来的。第一种是上一步被暂停的客户进程报告 AmS 已经暂停完毕，于是 AmS 从 activityPaused() 开始执行并间接调用到此；另一种是当用户按“Back”键时，客户进程会请求 AmS 终结（finish）当前 Activity，而在 finish 的过程中也会间接调用到此。

这一步中，设置了上一个 HistoryRecord 的内部变量 waitingVisible，从 false 变为 true，即它也处于等待显示状态，因为接下来 next 要变成显示的状态。

- ⑨ 通知 WmS 添加 Activity 切换的动画，有可能是 Task 切换，也有可能是 Activity 切换。

⑩ 这一步比较重要。此时要执行的 HistoryRecord 对象仅仅存在于 mHistory 列表中，而它仅代表了用户的意图，实际上它对应的客户进程还并不存在。所以首先要判断客户进程是否存在，源码中使用以下代码判断：

```
2728 if (next.app != null && next.app.thread != null) {
```

注意，为什么不是仅仅判断 next.app，而是还要判断 next.app.thread？因为 next.app 对应的是一个 ProcessRecord 对象，有时客户进程被杀死而该对象却会依然保留，thread 才是真正的客户进程引用。所

以，只有满足这两个条件才说明该 Activity 对应的客户进程处于已运行状态，并且相应的 Activity 对象也已经存在，所要做的仅仅是 resume 该 Activity。否则，要么是目标进程不存在，要么是目标进程中还需要创建一个新的 Activity 实例。举个例子，假设当前 mHistory 的内容是 C1→C2→C3，此时如果以默认方式启动 C1 类，姑且称之为 C1'，那么，目标进程 C 已经存在，C1 实例也已经存在，然而由于新启动的 C1' 是新建的 C1 对象，所以其内部的 app 和 app.thread 都为空，所以不能直接 resume 对应的 C1。

⑪ 如果上面不能直接 resume 已有的 Activity 对象，那么接下来就要判断目标 Activity 对应的进程是否存在。这是通过调用 startSpecificActivityLocked() 完成的，如其函数名所指，其内部要启动一个特定的 Activity。

在该方法中，调用 getProcessLocked() 方法首先获取目标 Activity 对应的进程对象 ProcessRecord app。如果 app 不为空，并且 app.thread 也不为空，说明目标进程还处于活动状态，所以只需要让目标进程再创建一个指定 Activity 的对象并执行之即可。相反，如果目标进程还不存在，则需要首先启动目标进程。这个判断语句和上一步中判断目标 Activity 是否存在的代码有一定形式的相似，但其逻辑是完全不同的，请注意区别，如以下代码所示：

```
1895     if (app != null && app.thread != null) {
```

该代码中的 app 变量是目标 Activity 在 AmS 中对应的 ProcessRecord 对象的，而前面代码中的 app 变量却是 mHistory 列表中要被执行的 HistoryRecord 对象的。

6. 应用进程 pause 指定的 Activity

前面讲过，在启动目标 Activity 前，如果当前正在运行任何 Activity，那么必须首先暂停，表面上看起来，这种逻辑严重违反了我们已有的 PC 使用体验，在 PC 中，可以同时运行多个应用程序，最大化一个窗口只是剥夺了其他窗口在屏幕上显示的权利，而该程序本身丝毫没有受到影响，而 Android 中却要强调当启动另一个 Activity 时，当前 Activity 必须首先暂停。事实上，这种规则没有想象的那么不同，因为暂停 Activity 是一种比较轻量级的操作。

暂停 Activity 的流程如图 10-7 所示。

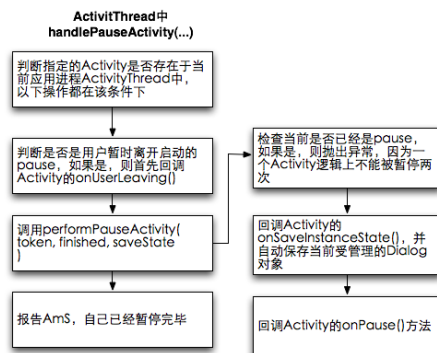


图 10-7 暂停 Activity 的过程

① 判断该 Activity 对象是否存在。

② 判断是否为用户离开产生的暂停消息。Android 后台会自动检测用户多长时间没有和当前 Activity 交互了，如果程序允许该 Activity 接收用户离开消息，那么系统判断是用户离开，就会暂停目标 Activity。此时 `handlePauseActivity` 的参数 `userLeave` 就为 `true`，程序员可以重载 Activity 的 `onUserLeaving()` 方法。

③ 调用 `performPauseActivity()` 完成真正的暂停代码，尽管实际的代码也很简单。这就是为什么说暂停是一种轻量级操作的原因。

(1) 检查当前是否已经 `pause` 了，条件就是当前 `ActivityRecord` 对象的 `pause` 变量值，`true` 代表已经暂停。该步骤对应的代码中，如果已经 `pause`，会抛出一个异常，说 “Performing pause of activity that is not resumed”，但这种情况实际上是不会发生的。

(2) 回调 Activity 的两个暂停操作函数，一个是 `onSaveInstanceState()`，另一个就是应用程序员熟知的 `onPause()`。前者实际上给程序员提供了保存当前状态的机会，该函数的参数是一个 `Bundle outstate` 变量，即程序员可以把需要保存的变量内容保存在系统提供的 `outState` 中，除此之外系统还自动把当前处于被管理状态的 `Dialog` 也保存起来；后者则提供了一个无参数的回调，在笔者看来，完全可以把 `onSaveInstanceState()` 和 `onPause()` 合并到一起，即把 `onSaveInstanceState()` 作为 `onPause()` 的默认处理方式，而只需要给 `onPause()` 添加两个参数，如下：

```
onPause(Bundle outState, boolean saveState)
```

这样的话，程序员不会在这两个回调函数中迷惑了，尤其是当考虑应该在哪个函数中保存当前 Activity 内部数据的时候。

④ 报告 AmS，自己已经暂停完毕，通过 IPC 调用到 AmS 的 `completePausedActivity()` 方法。

从以上流程可看出，暂停的操作的开销最多是在 `saveInstanceState()` 方法中，而这段代码不一定会被执行，只有当参数 `saveState` 为 `true` 时才执行，而 Activity 的内部所有变量都依然会持续保留在内存中，下一个要执行的 Activity 如果不是全屏的话，用户应该还可以看到刚才暂停的 Activity 界面。唯一的区别是被暂停的 Activity 将不能接收用户消息，因为 WmS 在接收到用户消息后，仅仅把该消息 `dispatch` 到了当前的活动窗口。由于被暂停的 Activity 内部的所有数据依然保留，包括内部的 `Handler` 对象，其所在的进程也依然保存，因此可以想象，此时如果从当前的 Activity 向被暂停的后台 Activity 的 `Handler` 对象发送一个消息，那么后台 Activity 中的 `Handler` 对象的 `handleMessage()` 方法将依然会被执行。

再一次强调，被暂停的 Activity 仅仅是被 WmS 剥夺了获取用户消息的权利，如果改变 WmS 内部的消息派发机制，允许根据用户单击的屏幕位置，把消息派发到相应的 Activity 窗口，那么这些 Activity 都将处于活动状态，也就是所谓的多窗口多任务。

7. activityPaused()

上一节中，当 AmS 通知当前 Activity 暂停后，AmS 会立即返回，而在目标进程中则是发送一个暂停的消息，处理完该暂停消息后，目标进程会调用 AmS 的 `activityPaused()` 报告 AmS 自己已经暂停完毕，从而 AmS 将开始启动真正的 Activity。那么，这个 `activityPaused()` 方法内部的流程如何呢？



① 调用 `indexOfTokenLocked()` 找到指定的 token 在 `mHistory` 中对应的 `HistoryRecord`。如果说 token 的确存在，那么实际上可以简单地直接将 token 转换为 `HistoryRecord` 即可。

② 如果 `activityPaused()` 的第三个参数 `timeout` 为 `false`，说明是非超时的、正常时间内暂停的 `Activity`，所以，首先需要从 `mHandler` 中移除还没有被处理的超时消息。这种逻辑整体应用于 `AmS` 和 `WmS` 中的很多函数调用中。

③ 如果参数 `r` 和 `AmS` 内部变量 `mPausingActivity` 相同，说明 `r` 正是刚刚需要暂停的 `Activity`，因此，可以调用 `completePausedLocked()` 执行完成暂停后的行为。

(1) 给 `prev` 赋值 `mPausingActivity`，即此时可以认为上一个被执行的 `Activity` 就是刚刚暂停的 `Activity`。

(2) 如果 `prev` 的 `finishing` 标识为 `true`，说明上一个 `Activity` 已经完成，因此需要调用 `finishCurrentActivityLocked()` 执行相关操作。而对于一般的启动流程来讲，无论是从 A 启动到 B，还是从 B 按“Back”键返回到 A，都不会是 `finishing` 的，而仅仅是 `stop` 状态。这个条件似乎只有在内存回收时才会被执行。

(3) 将 `mPausingActivity` 变量置为空，因为当前已经不存在正在暂停的 `Activity` 了。

④ 调用 `resumeTopActivityLocked()` 方法正式启动目标 `Activity`，而其内部的调用流程是从 `mHistory` 中取出最后一个 `Activity` 对象，并去执行。此处不再赘述。

8. 通知应用进程 resume 指定的 Activity

在上一节中，如果目标 `Activity` 已经在运行的序列中，则直接通知目标进程 resume 暂停的 `Activity` 对象即可。

① 通知 `WmS`，把目标对象的窗口设为可交互状态。这是通过调用 `WmS` 的 `setAppVisible` 方法完成的，其参数是目标 `Activity` 对应的 `HistoryRecord` 对象。

② 改变一些内部变量的状态，比如 `mResumedActivity=next`，即把目标 `Activity` 作为当前正在运行的 `Activity`。

③ 通知目标进程继续运行目标 `Activity`，这是通过调用 `next.app.thread.scheduleResumeActivity(next...)` 完成的。从这里可以再次看出，Android 的架构中始终认为改变目标 `Activity` 状态的具体任务应该由目标进程完成，`AmS` 仅发出指令。

④ 调用 `completeResumeLocked()` 对内部其他相关变量进行设置。

下面对以上第三步调用 `ActivityThread` 中 `scheduleResumeActivity()` 函数的执行流程作更进一步的分析。该函数本身仅仅是发一个异步消息，并立即返回到 `AmS`，而该消息响应函数是 `handleResumeActivity()`，该函数的内部执行过程如图 10-8 所示。



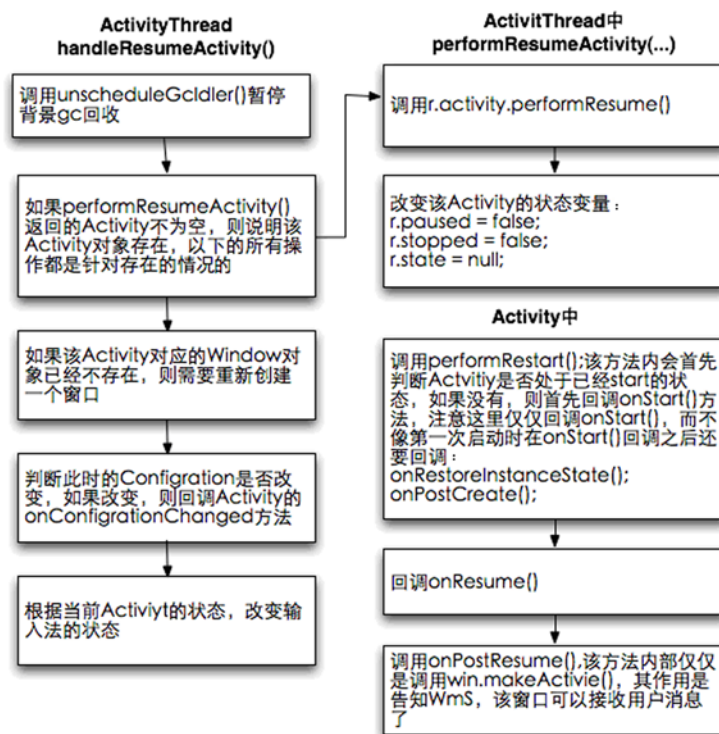


图 10-8 resume Activity 的流程

以上过程描述如下：

- ① 调用 `unscheduleGcIdler()` 暂停停止内部回收。关于内存回收参见本章第 10.2 小节。
- ② 调用 `performResumeActivity()`。该方法内部主要是回调 Activity 对象的 `onResume()` 方法，但是，如果此刻 Activity 处于 stop 状态，即 `r.stopped == true`，那么就需要先 start 该 Activity，具体就是仅仅回调 Activity 的 `onStart()` 函数。
- ③ 判断该 Activity 对应的 Window 是否存在。在一般情况下，该 Window 对象都会存在，只有一种例外，那就是系统内存紧张到把后台窗口所对应的显存也临时释放了。在这种情况下，当被回收的 Activity 对应的窗口重新显示时，系统会重新为 Activity 分配一个窗口显存(Canvas 对象)。
- ④ 查看当前的 Configuration 是否改变，比如用户停止该 Activity 时是竖屏退出，而继续却是横屏进入，则此时应该回调 Activity 的 `onConfiguration()` 函数。
- ⑤ 根据当前 Activity 的状态改变输入法的状态，关于输入的逻辑参见本书第 17 章输入法原理。

9. 创建应用进程

如果目标 Activity 所在的进程还不存在，则首先需要启动应用进程，其大致过程是 AmS 调用 Process 进程类启动一个新的应用进程，新的应用进程会从 ActivityThread 的 `main()` 函数处开始执行，当目标进程启动后，再报告 AmS 自己已经启动，然后 AmS 再通知目标进程启动目标 Activity。



① 首先判断目标 `HistoryRecord` 对象对应的 `app` 数据是否存在, 并且其对应的应用进程是否也存在。正如源码中的注释, 这个判断是由三个具体条件完成的, 如果三个条件都满足, 则说明目标进程的确存在。这个判断中指出了相关三个变量的含义。

- `app`: 这代表的是一个 `ProcessRecord` 对象, 该对象仅仅是一个数据而已, 并不代表目标进程的任何实际状态。
- `app.pid`: 如果 `pid` 大于 0, 说明目标进程一起存在, 但有可能正在初始化。
- `app.thread`: 这是一个 `ActivityThread` 对象, 只有目标进程存在并且已经初始化完毕, 该变量才具备有效值。

以上判断一般会有两个结果。第一个是目标进程已经存在, 但是正在初始化, 这种情况意味着不需要再做什么, 仅仅返回该 `ProcessRecord` 对象, 并等待目标进程初始化完毕并报告给 `AmS` 即可; 另一种情况是 `app` 存在但其不对应任何目标进程, 这说明该 `app` 数据对象是一个死对象, 就像 Linux 中的 zombie 进程数据, 因此, 调用 `handleAppDiedLocked()` 方法清除该 `app`。

② 处理 `FROM_BACKGROUND` 标识。Android 认为, 如果以背景方式启动目标进程, 那么如果该进程已经列为死亡名单 `mBackProcesses` 列表中, 那么就不需要再尝试启动, 系统认为这是一个不可能启动的目标进程; 如果以非背景方式启动, 那么则不判断目标进程是否在死亡名单中, 而是再次尝试启动目标进程, 这就是 `FROM_BACKGROUND` 的代码层面的含义。

③ 判断目标 `HistoryRecord` 的 `app` 变量是否为空。如果为空, 则需要为其创建一个新的 `ProcessRecord` 对象, 并把该 `record` 对象添加到 `mProcessNames` 列表中, 并且同时给 `app.info` 赋值。这点很重要, 因为当目标进程启动后, `AmS` 会命令目标进程运行 `app.info` 中包含的 `Activity` 对象, 而 `app.info` 的内容的最初来源是 `caller` 调用 `AmS` 的 `startActivity()` 方法中根据调用的 `intent` 从系统中查询到的相关 `ApplicationInfo` 信息。

④ 再次判断 `AmS` 是否处于已启动状态, 这只有在系统启动时才会发生, 而在一般情况下, 系统始终都是处于已启动状态。当然, 如果还没有启动好, 则把目标进程请求暂时存储到 `mProcessOnHold` 列表中。

⑤ 调用 `startProcessLocked` (三个参数) 启动真正创建一个进程。注意, 这与本节的入口函数 `startProcessLocked` (七个参数) 是同名的, 注意区分其不同。

⑥ 进入到 `startProcessLocked()` 方法中, 从 `mPidsSelfLocked` 列表中删除目标进程, 当然, 此时该列表中大多数情况下并没有目标进程的记录。该列表是一个 `SparseArray` 对象, 保存了所有正在运行目标进程的 `pid` 到 `ProcessRecord` 的映射。从目标功能的角度来看, 这个变量和 `mProcessNames` 变量都保存了当前正在运行的 `ProcessRecord` 对象, 只是前者提供了 `pid` 映射, 而后者提供了进程名称映射。这也说明了无论是 `pid` 还是进程名称都必须是唯一的, 这也就是为什么在 Android 中进程名称一般使用包名的原因。

⑦ 为变量 `mProcDeath` 赋值。该变量也是一个 `int` 型数组, 内部包含最近 20 个应用进程的意外关闭次数, 如果某应用进程没有关闭, 则其对应的值为 0, 这仅仅是一个统计数据。



⑧ 调用 `Process.start()` 方法，这是一个 native 的方法，内部将会从 `zygote` 进程中 fork 出一个新的应用进程，并且应用进程会从 `ActivityThread` 类的 `main()` 方法中开始执行，关于系统的启动过程参见本书第 9 章。

⑨ 把上一步返回的 `pid` 和变量 `app` 添加到 `mPidsSelfLocked` 容器中，在后面的 `attachApplication()` 方法中还要根据 `pid` 获得该 `app` 对象。

至此，启动进程的事情就算结束了，接下来 AmS 会等待目标进程报告启动完毕，届时，目标进程会 IPC 调用 AmS 的 `attachApplication()` 方法请求 AmS 给目标进程指定目标 Activity 去运行。而此时，`mHistory` 中最上面的 `HistoryRecord` 对象的 `app` 变量依然为空，而在 `mPidsSelfLocked` 和 `mProcessNames` 两个容器中却都保存了新创建的 `ProcessRecord` 对象。而一旦 `HistoryRecord` 对应的 Activity 被启动，那个相应的 `ProcessRecord` 对象才会被赋值给 `HistoryRecord` 的 `app` 变量。

10. attachApplication 的过程

上一节说到，当目标进程启动后，就会报告给 AmS，自己已经启动完毕可以启动 Activity 了，这实际上是通过 IPC 调用 AmS 的 `attachApplication()` 方法完成的，而该方法的内部执行过程如下。

① 根据 `Binder.getCallingPid(thread)` 获得客户进程的 `pid` 值，并调用 `attachApplicationLocked(thread, pid)` 方法，其中参数包含了 `pid` 值。

② 在 `...locked()` 方法中，首先根据 `pid` 找到对应的 `ProcessRecord` 对象，如果找不到对应的 `ProcessRecord` 对象，说明该 `pid` 客户进程是一个没有经过 AmS 允许的“野进程”。为什么呢？因为 AmS 在启动任何客户进程前，都已经在内部为未来的客户进程创建了相应的 `ProcessRecord` 对象，并且在调用 `Process.startProcess()` 返回时，将客户进程的 `pid` 赋值到了 `app.pid` 变量，而此处竟然找不到，那一定是“野进程”。

③ 给 `app` 对象内部的其他相应变量赋值，比如 `app.thread = thread` 等。

④ 确保目标程序（APK）文件已经被转换为了 `odex` 文件。Android 中的安装程序是 APK 文件，该文件实际上是一个 `zip` 文件，该文件在安装时会自动在 `/data/app` 目录产生一个 `dex` 文件。这个 `dex` 文件是从 `zip` 文件中提取的，而为了提高运行效率，该 `dex` 文件在运行前一般都会被系统转换为 `odex` 文件，即所谓的 `Optimized dex` 文件。本步中如果检查还没有生成 `odex` 文件，则先要生成 `odex` 文件。

⑤ 调用 `thread.bindApplication(..., app.info, ...)` 方法通知（命令）客户进程运行指定的 Activity 所在的 APK 文件，这个“指定的 Activity”被包含在了参数 `app.info` 中。`info` 值的最初来源是根据调用者 `intent` 的内容从系统中查询到的 `ApplicationInfo` 信息，参见 10.1.3 节第 7 小节中的第三步。通知完毕后，顺便调用 `updateLruProcess()` 更新“最近调用的进程列表”内容。

⑥ 调用 `realStartActivityLocked()` 通知客户进程运行指定的 Activity。注意，上一步仅仅是告诉目标进程从 APK 文件中加载 `Application` 类并运行，但没有真正启动指定的 Activity，而这一步则会真正启动目标 Activity。

⑦ 查询是否有依赖于目标进程的等待 Service 或等待 Broadcast 服务，如果有的话一并通知目标进程执行之。这两个信息保存在变量 `mPendingService` 和 `mPendingBroadcast` 中。

除了以上这七步外，其中还包含一些其他不重要的细节，有兴趣的读者可以再研究源码。

截至本步结束，尽管客户进程已经存在，然而这个客户进程所包含的程序文件仅仅是 Framework 中的 `ActivityThread` 基类，而没有涉及应用程序本身的任何文件。这点类似于 Linux 程序的启动过程，即首先由操作系统产生一个空进程，然后再通知空进程去加载具体的要执行的程序文件。

11. 应用进程 launch 指定的 Activity

上一节结束后，目标进程（空进程）就需要根据 AmS 所提供的 `ApplicationInfo` 的信息来加载具体的 Activity 了。请注意 `ApplicationInfo` 是实现了 `Parcelable` 接口的类，所以才能在进程间传递数据。

在上一节中，目标进程向 AmS 报告自己已经启动，AmS 会执行 `attachApplication()`，而在该方法中则先后调用了两个 `ActivityThread.ApplicationThread` 的方法，分别是 `scheduleLaunchActivity()` 和 `scheduleResumeActivity()`，如以下代码片段所示。

```
5559 thread.bindApplication(processName, app.instrum
1822 app.thread.scheduleLaunchActivity(new I
```

这两个函数在 `ActivityThread` 中是异步执行的，即向 `ActivityThread` 所在的线程发送两个异步消息，并立即返回，而后 `ActivityThread` 再按顺序处理这两个消息。如图 10-9 所示。

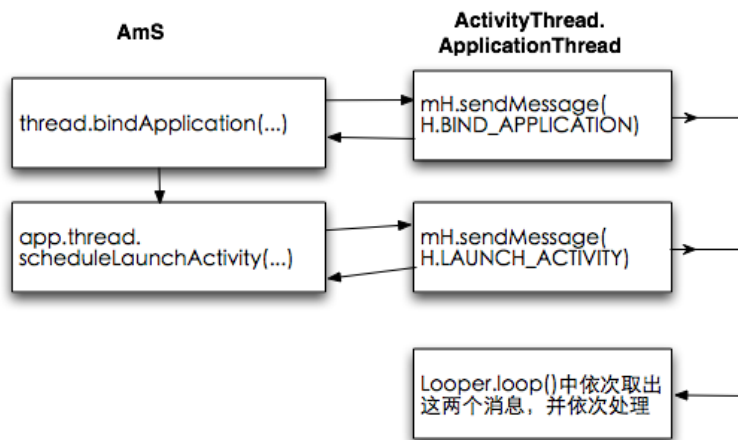


图 10-9 launch Activity 的流程

而这两个消息的处理函数分别是 `handleBindApplication()` 和 `handleLaunchActivity()`，因此本节来分析这两个函数的处理过程。

前者的启动流程如图 10-10 所示。

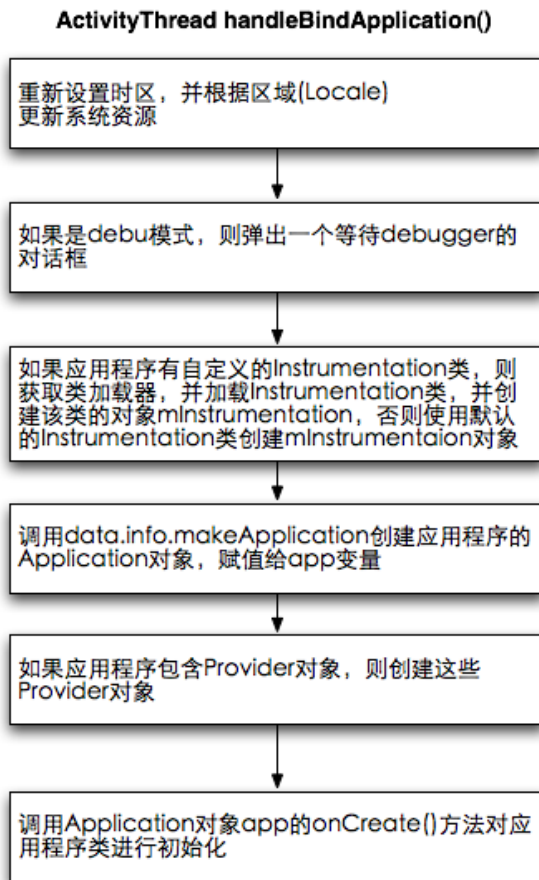


图 10-10 handleBindApplication()的内部过程

① 重新设置时区和区域。为什么呢？因为所有的应用进程都是从 `zygote` 进程复制出来的，`zygote` 本身已经装载了系统资源，不同的区域对应了不同的资源内容，而在 `zygote` 装载系统资源时，系统区域是默认的，而此时用户有可能已经改变了系统区域，因此重新设置区域。当然，该步骤内部会判断当前区域是否和默认区域一样，如果不一样才会重新装载。

② 如果是 `debug` 模式，则弹出一个等待对话框，从这点可以看出，在调试应用程序时，界面上弹出的“Waiting for debugger”对话框其实是应用程序本身的一部分，而不是调试过程特有的。

③ 在一般情况下，应用程序都不会有自定义的 `Instrumentation` 类，只有一个例外，那就是 Android 中的单元测试（`UnitTest`）模块。Android 提供了一个单元测试框架，允许产生一个专门测试应用程序的单元测试 APK，该框架正是利用了自定义 `Instrumentation` 类，使得真正的应用程序在单元测试的“容器”中启动，关于 `UnitTest` 请参照 Android 官方 `sdk` 说明文档。

④ 调用 `makeApplication()` 创建 `Application` 类的对象。该函数主要是调用目标 APK 文件中的 `Application` 类对象，一个 APK 文件中只能有一个 `Application` 对象，程序员可以基于 `Application` 类定义

自己的类，并且在 AndroidManifest 的<Application>标签中使用 android:name 属性给出。如果程序没有自定义的 Application 类，系统会加载默认的 Application 类。

从这里可以看出，Application 类才是一个应用程序被加载后运行的第一个类，而该对象在应用程序中是全局唯一的，因此，对于一些需要全局保存的值可以放在该类中。

⑤ 如果该应用中包含 Provider 对象，则启动这些 Provider 对象。从这里可以看出，应用程序启动时 Provider 会先于 Activity 运行。

⑥ 调用 Application 类的 onCreate()方法，注意，这是 Application 类的，而不是 Activity 类的。

接下来再看 handleLaunchActivity()的执行过程。该函数的主要作用是最终加载指定的 Activity 并运行，其流程如图 10-11 所示。

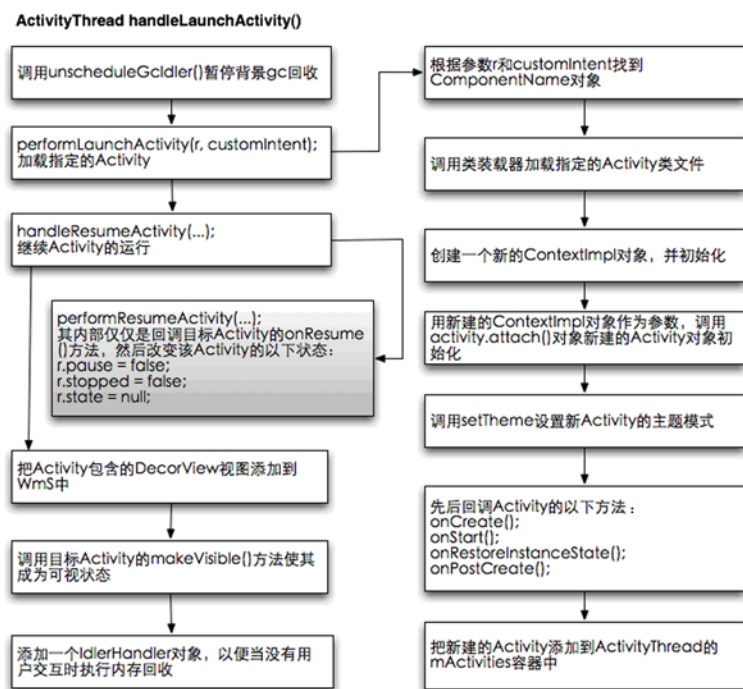


图 10-11 handleLaunchActivity()的内部过程

① 暂停背景运行的内存回收器。该回收器实际上是一个 Handler 对象，该对象会在系统空闲时接收回收消息，并执行相应的回收函数。关于内存回收参见第 10.2 节。

② 调用 performLaunchActivity()继续完成 Activity 的加载。该函数中：

(1) 根据调用参数创建一个 ComponentName 对象。

(2) 加载目标 Activity 类。

(3) 创建一个 ContextImpl 对象。每一个 Activity 都包含一个 ContextImpl 对象，该对象是一个轻量类。创建该对象后，调用 activity.attach()方法，用该 ContextImpl 对象初始化 activity 对象，实际上就

是把 ContextImpl 对象作为 Activity 的 base Context。

(4) 调用 setTheme() 为 Activity 设置主题模式。该主题是从 AndroidManifest.xml 中取得的，如果 xml 文件中没有声明自定义的 theme，则系统使用默认的主题模式。

(5) 先后调用目标 Activity 的 onXXX() 四个函数。

(6) 把新建的 Activity 对象加入到 ActivityThread 的 mActivities 列表中。这里需要注意，mActivities 列表中保存的是 Activity 在 AmS 中对应的 HistoryRecord 对象，而不是直接的 Activity 对象，添加代码如下：

```
2656 mActivities.put(r.token, r);
```

参数 r.token 正是 HistoryRecord 对象的引用，r 是本地的 ActivityRecord 对象，因此，如果是客户程序中同一个 Activity 类，则有可能有多个对象。

③ 这个时候 performLaunchActivity() 方法返回，继续执行 handleResumeActivity()，该方法内部又调用 performResumeActivity() 方法，其内部仅仅是调用了目标 Activity 的 onResume() 方法，并改变了该 Activity 在 ActivityThread 中对应的 ActivityRecord 数据对象的 pause、stopped 变量。

④ 至此，对用户来讲，还没在界面上看到任何内容。尽管以上步骤中 ActivityThread 和 AmS 风风火火地打成一团，并装载了相关的执行类，然而这些都还没有通知给 WmS，而此时，一切就绪。因此本步骤中从 Activity 对象中找到其包含的 DecorView 对象，并把它添加到 WmS 中。

⑤ 调用目标 Activity 的 makeVisible() 函数，该函数其内部又会辗转和 WmS 进行各种调用，并最终导致 Activity 包含的 DecorView 显示到屏幕上，关于这一点请参照第 14 章。

⑥ 添加一个 IdleHandler 对象，因为在一般情况下，该步骤执行完毕后，Activity 就会进入空闲状态，所以可以进行内存回收。关于内存回收请参照第 10.2 节。

执行完这两个函数后，指定的 Activity 就真正启动了，用户此时应该可以看到该 Activity 的界面。

10.1.4 stopActivityLocked() 停止 Activity

在前面几节中，从 A 启动到 B 时，仅说了需要先暂停 A，然后再启动 B，读者是否想过那么 A 到底是什么时候停止 (stop) 或者销毁 (destroy) 的呢？本节就来回答这个问题。

在 Activity 启动的过程中，AmS 一直没有通知相应的应用进程停止 (stop) Activity，而仅仅是暂停 (pause)。这对于 WmS 来讲已经足够了，因为 WmS 只需要知道把用户消息发向哪一个窗口就可以了，所以被暂停的 Activity 没有必要再执行停止操作。

然而，AmS 内部提出了一种主动内存管理机制，即 AmS 会主动根据当前运行的 Activity 的状态，在系统内部不够用的时候杀死优先级较低的应用以释放内存。而如何判断优先级的高低呢，这就需要为不同的 Activity 赋予不同的权值，而这个权值正是 Activity 的状态，包括正在运行、停止、销毁等。这就是为什么 Activity 提供了三个和停止相关的回调 (分别是 onPause()、onStop() 和 onDestroy()) 的原因，不同的回调发生在不同停止时机。比如当 AmS 希望当前的 Activity 暂停和用户交互时，就会执行

onPause()回调;当 AmS 希望把该 Activity 的权值改为 stop 状态时,则执行 onStop()回调;当 AmS 准备在后台杀死该 Activity 时,则执行 onDestroy()回调。

那么,接下来分析 Activity 是什么时候被停止,以及是怎样被停止的。

前面 10.1.3 节中讲述在 resume 具体的 Activity 时,当目标进程 ActivityThread 执行完 handleResumeActivity()时,会添加一个 IdleHandler 对象,如以下代码所示:

```
3222         r.nextIdle = mNewActivities;
3223         mNewActivities = r;
3224         if (localLOGV) Slog.v(
3225             TAG, "Scheduling idle handler for " + r);
3226         Looper.myQueue().addIdleHandler(new Idler());
```

以上代码最后一句 addIdleHandler(new Idler())创建一个 Idler 对象,并添加到线程消息队列中。为什么要此时添加呢?因为 ActivityThread 已经完成了启动 Activity 的全部的工具,接下来就应该是 idle 状态了,因此可以执行一些 idle 行为,而 Idler 类的内部处理函数则是调用 WmS 的 activityIdle()方法,如以下代码所示:

```
2169         try {
2170             am.activityIdle(a.token, a.createdConfig);
```

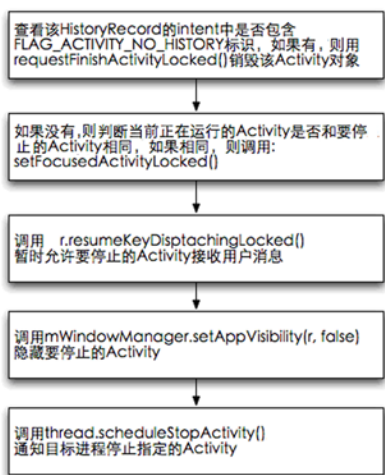


图 10-12 stopActivityLocked()的内部过程

而 AmS 中 activityIdle()的执行流程如图 10-12 所示,在该函数的内部则会辗转调用到 stopActivityLocked()方法。本小节暂不讨论这个辗转的过程,因为这更多的是和 AmS 内部的主动内存管理机制相关,因此把它放到本章第 10.2 节的内存管理中。而本小节则仅关系 stopActivityLocked()的内部执行过程,如图 10-12 所示。

① 处理 FLAG_ACTIVITY_NO_HISTORY 标识。该标识的作用是不要让该 Activity 出现在 mHistory 中,即这是一次性的,比如 A→B→C,如果 B 的启动标识包含该标识,那么当从 C 返回后则会继续执行 A 而不是 B。

② 这一步实际上真的没有必要,因为既然要停止该 Activity,就完全没有必要再让它成为活动的。

③ 允许目标 Activity 接收用户消息,这一步也是多余的,原因同上。大家可以从代码中删除这两步重新生成 sdk,其运行结果是不受影响的,如以下代码所示:

```
4052         } else if (r.app != null && r.app.thread != null)
4053             if (mFocusedActivity == r) {
4054                 setFocusedActivityLocked(topRunningActi
4055             }
4056             r.resumeKeyDispatchingLocked();
```

④ 调用 setAppVisibility()隐藏目标 Activity,注意这里的第二个参数为 false,所以是隐藏。

⑤ 调用 thread.scheduleStopActivity()向目标进程的 ActivityThread 对象发送一个异步消息,通知目

标进程停止指定的 Activity。

执行完以上过程后，ActivityThread 中处理该异步消息的函数 handleStopActivity() 将开始执行，其执行过程如图 10-13 所示。

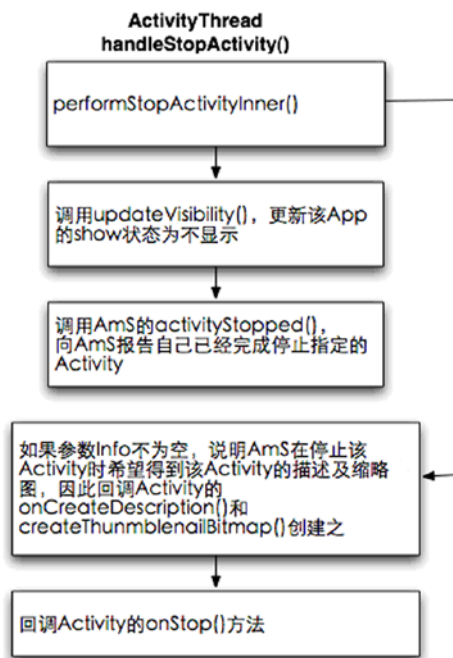


图 10-13 handleStopActivity()的内部流程

① 调用 performStopActivityInner()。

(1) 如果参数 Info 不为空，则回调 Activity 对象的 onCreateDescription() 产生该 Activity 的描述，AmS 可能需要该描述进行一定的统计。info.thumbnail 是一个 Bitmap 型变量，既然该变量也要使用 IPC 进行传递，那么它必须是一个实现了 Parcelable 接口的类，这通过查看 Bitmap 类的代码原型得到了验证。

(2) 回调 onStop() 方法，默认的实现是什么都不做。

② 调用 updateVisibility() 方法。前面在 pause 的处理过程中，仅仅是告诉 WmS 屏蔽了对该 Activity 的消息派发，而该 Activity 对应的窗口却依然是显示的状态，而此时 stop 的含义则包括不显示该窗口，即 View v = r.activity.mDecor 窗口的现实状态被设为 false, v.setVisibility(false)。

③ 调用 AmS 的 activityStopped()，向 AmS 报告，自己已经完成了停止该 Activity。在 AmS 端，该函数内部比较简单，主要是调用 trimApplication() 进行一定的内存回收工作，这点可参照第 10.2 节。

对比以上步骤和 pause 的步骤发现，两者的主要区别有以下几点。

- 发生在不同的时机。pause 发生在目标 Activity 启动之前，而 stop 则发生在目标 Activity 启动之后。
- 回调 Activity 的不同函数。



- 被调方式不同。pause 是由 AmS “有意”调用的，而 stop 则是 AmS “无意”调用的。如图 10-14 所示。

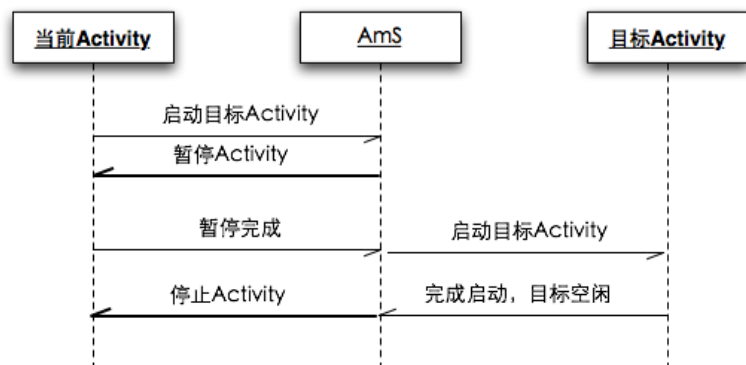


图 10-14 从暂停到启动的全过程

10.1.5 按“Home”键回到桌面的过程

前面曾说过，无论是按“Home”还是“Back”等系统键，最终都会导致执行 startActivit()，区别仅在于前端消息处理的过程。本节就来介绍当用户按下“Home”键后的前端处理过程。

在以往的应用程序开发中，大家可能已经发现，尽管可以在 Activity 的 onKeyDown() 函数中截获“Back”键，但是却不能够截获“Home”键，其原因是 WmS 在进行消息派发时，对“Home”键做了默认处理，而没有把它派发到应用程序里面去。这纯粹是一种产品理念，Android 认为无论应用程序是什么样的，用户都可以通过“Home”键回到桌面程序，这一点和 iPhone 的设计理念相同。

Home 键的具体处理过程参见附图 3。

① 在 WmS 中，调用了 PhoneWindowManager 类中 interceptKeyTi() 方法截获所有消息，而在该函数内部处理了一些系统按键消息，这其中就包括“Home”键。当发现是“Home”键时，则调用 LaunchHomeFromHotKey() 方法。

(1) 判断当前是否处于锁屏状态，如果是，则什么都不做。此时用户应该按“Menu”键进行解除屏幕锁定。

(2) 判断是否处于解锁状态，如果是则弹出解锁对话框。这种情况一般仅发生在开机的时候，此时如果 SIM 卡有启动密码，则系统首先弹出一个对话框请用户输入密码。

(3) 在一般情况下，屏幕没有锁屏，并且也没有密码锁定，此时先通知 AmS 暂停程序切换，然后关闭系统对话框。

(4) 调用 startDockOrHome()，这个方法应该是在 Android2.0 版本以后增加的，系统提供了两个 Home 程序，一个是所谓的插座(Dock)方法，另一个是标准的 Home 程序。前者是当用户把手机插入到了 Android 专用的底座硬件中，而这则是通过底层提供 API 接口进行查询的。



① 调用 `createHomeDockIntent()` 方法构造一个能够启动 Dock 或者 Home 的 Intent 对象。

② 调用 `mContext.startActivity()` 启动 Intent 对应的 Activity。此处需要特别注意，这里是调用 `ContextImpl` 类的 `startActivity()` 方法，而不是调用 `Activity` 类的 `startActivity()` 方法。本章之前所讲的 `startActivity()` 均是指从 `Activity` 中执行，而在 `Activity` 的 `startActivity()` 中会调用 `startActivityForResult()` 方法，而该方法内部则继续调用 `Instrumentation` 类的 `execStartActivity()` 方法，而在 `ContextImpl` 中的 `startActivity()` 方法中也是调用 `Instrumentation` 类的 `execStartActivity()` 方法，但在调用之前检查了一下 Intent 中是否有 `NEW_TASK` 的标识，如以下代码所示：

```

612     public void startActivity(Intent intent) {
613         if ((intent.getFlags() & Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
614             throw new AndroidRuntimeException(
615                 "Calling startActivity() from outside of an Activity
616                 + " context requires the FLAG_ACTIVITY_NEW_TASK fl
617                 + " Is this really what you want?");
618         }
619         mMainThread.getInstrumentation().execStartActivity(
620             getOuterContext(), mMainThread.getApplicationThread(), null
621     }

```

以上代码说明，调用 `startActivity()` 有两种方法，一种是从 `Activity` 中调用，另一种是直接从 `ContextImpl` 中调用，但是后者调用必须包含 `NEW_TASK` 的标识。这种限制的原因是，如果没有 `NEW_TASK` 标识，而且不是从 `Activity` 中调用，意味着此时 AmS 中可能会是一个空 Task，而这是不应该的。因此，在 `createDockHomeIntent()` 方法中，该 Intent 包含了 `NEW_TASK` 标识，如以下代码所示：

```

505     mDeskDockIntent = new Intent(Intent.ACTION_MAIN, null);
506     mDeskDockIntent.addCategory(Intent.CATEGORY_DESK_DOCK);
507     mDeskDockIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK
508         | Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);

```

`Instrumentation.execStartActivity()` 以后的流程就和本章前面讲的 `startActivity()` 完全相同了。

`mContext` 是从系统进程中创建的 `Context` 对象，该对象不与任何 `Activity` 对应。

总结“Home”键与普通 `startActivity()`，其本质区别在于前者构造了一个能够启动 Home 程序的特殊 Intent，并用该 Intent 调用 `ContextImpl` 类中的 `startActivity()`；而后者则是根据运行环境提供的 Intent，并从 `Activity` 类中调用 `startActivity()`。

10.1.6 按“Back”键回到上一个 Activity

“Back”键不是系统按键，但是“Back”键却有默认处理方法，因此，应用程序设计时可以在 `Activity` 的 `onKey()` 方法中截获该消息。

① 在 `Activity` 类的 `onKey()` 方法中，会检测是否是“Back”键，如果是，则调用 `onBackPressed()` 方法。



② 该方法则会调用 `finish()` 方法，该方法也就是应用程序设计时所使用的 `finish()` 方法，也就是说用户按下“Back”键和程序员主动调用 `finish()` 的作用完全相同。

③ 在 `finish()` 方法中，远程调用 AmS 的 `finishActivity()` 方法。

至此，Activity 的任务就完成了，剩下的就是 AmS 执行 `finishActivity()` 方法了。

在 AmS 端的 `finishActivity()` 的执行过程如附图 3 所示。

① 调用 `requestFinishActivityLocked()`。

② 判断该 Activity 是不是最后一个 Activity，判断的方法如以下代码所示：

```
4101 if (!p.finishing && p != r) {
```

这里正是利用了 `mHistory` 列表的 `z-order` 排列顺序，即最后一个总是当前正在运行的 Activity。比如 `A→B→C`，那么 A 是第一个启动的 Activity，B 是第二个，C 是第三个，也是当前正在运行的 Activity，以上代码中的两个条件是指 `p` 没有结束，并且 `p` 不是要停止的 Activity，因为 `mHistory` 中最后一个 Activity 肯定是当前要 finish 的，所以 A 和 B 都不等于 C，这就意味着 C 不是最后一个 Activity。如果不是最后一个 Activity，则可以正常继续执行 `finishActivityLocked()` 结束该 Activity。

③ 如果 C 是最后一个 Activity，并且 C 是 Home 类型的 Activity，那么什么都不做，因为 Home 类型的 Activity 不应该返回到其他应用程序中。但如果不是 Home 类型，并且是最后一个 Activity 会如何呢？答案是依然会继续执行 `finishActivityLocked()` 方法，实际上在 `finishActivityLocked()` 内部的确会结束该 Activity，只是在结束后会判断 `mHistory` 中是否有 Activity，如果没有，则会重新启动 Home 类型的 Activity。

调用 `finishActivityLocked()` 时，参数 `r` 代表要停止的 `HistoryRecord` 对象，`index` 代表该对象在 `mHistoryRecord` 中的位置。

(1) 改变该 `HistoryRecord` 的 `finishing` 状态，当然如果该状态已经为 `true`，意味着已经发出停止的命令了，所以会返回 `false`。

(2) 如果该 Activity 的上面还有其他的 Activity，则把它上面的 `HistoryRecord` 的对象 `next` 的 `frontOfTask` 状态改为 `true`。这种情况一般很少发生，因为 `finishActivityLocked()` 只有在按“Back”键的时候才被调用，而此时要停止的 Activity 正是当前正在运行的 Activity，所以 `index` 始终等于 `mHistory.size() - 1`。只有一种例外，那就是程序员主动调用 `finish()` 方法，而且目标对象是非活动的 Activity。如果这种情况发生，则必须把它上面的 `HistoryRecord` 的 `frontOfTask` 置为 `true`，该变量的含义是该 Activity 是否为 Task 中的第一个 Activity。比如，假设当前 `mHistory` 的列表如下：`A1→A2→B1→B2→B3→C1→C2`，A、B 和 C 代表三个不同的 Task，此时如果要停止 B1，显然 B1 是 B 任务中的第一个 Activity 对象，停止后 B2 将取而代之，因此必须把 B2 的 `frontOfTask` 置为 `true`。

(3) 暂停对该 `HistoryRecord` 的消息派发。

(4) 查看当前是否有正在运行的 Activity，如果有并且就是要停止的 Activity 对象，则调用 `startPausingLocked()` 先暂停该 Activity 对象。

(5) 如果当前正在运行的 Activity 不是要停止的 Activity，则直接调用 `finishCurrentActivityLocked`

(三个参数版本)完成停止工作。该函数内部实际上牵扯到 AmS 的内存回收机制，因此，把它放在第 10.2 节里面详述。

接下来继续分析以上 (4) 中的 `startPausingLocked()` 内部的过程。在前面介绍 `startActivity()` 的启动过程中，如果当前正在执行某个 Activity，也是首先调用 `startPausingLocked()` 暂停当前 Activity，该方法与外部方法的调用关系参见附图 3，此处截取其内部过程如图 10-15 所示。



图 10-15 `startPausingLocked()` 的内部流程

- ① 判断 `mResumeActivity` 是否为空，在一般情况下，该值肯定不为空。
- ② 把 `mResumeActivity` 设为空，并给 `mPausingActivity` 赋值 `prev`，而 `prev` 正是当前需要暂停的 Activity。
- ③ 通知客户进程暂停该 Activity 对象。
- ④ 发送一个超时检测消息。

至此，AmS 就完成了暂停操作，然后就会等待客户进程暂停目标 Activity 后，报告 AmS 暂停完成，而 AmS 会继续从 `activityPaused()` 方法处开始执行。

10.1.7 长按“Home”键

长按“Home”键也是系统消息，应用程序无法对长按“Home”键的消息做自定义处理。

- ① 在 `PhoneWindowManager` 的 `interceptKeyTi()` 方法中检测是否是长按“Home”键的消息，如果是，则发送一个异步消息。处理该异步消息的是一个线程对象，名称为 `mHomeLongPress`，该对象是在构造

函数中创建的，其作用就是用户所看到的弹出一个对话框，里面包含了最近启动的应用列表。而该线程内部的执行过程具体如下：

- (1) 调用 `performHapticFeedback()` 给用户一个震动提醒，Haptic 的意思是“触觉反馈”。
- (2) 关闭所有系统窗口，常见的系统窗口包括状态栏、Toast 窗口、系统警告或者错误对话框等。
- (3) 调用 `showRecentDialog()` 方法弹出最近任务列表对话框。而该方法内部则是首先创建一个

`RecentApplicationsDialog` 类对象，该对象基于 `Dialog` 类，然后再调用 `show()` 方法将该对话框显示出来。

关于 `RecentApplicationsDialog` 类的执行过程如下。

① 在该类的构造函数中，为该 `Dialog` 对象添加视图，使用的是系统内置的 `recent_apps_dialog.xml` 布局文件。该文件中包含了六个 `Button` 视图，如果读者想自定义该视图，则可以修改该文件。

② 在该 `Dialog` 对象的 `onStart()` 方法中，调用 `reloadButtons()` 为这六个 `Button` 视图设置具体的显示内容及 `onClick()` 事件。`reloadButtons()` 方法的内部流程如下。

(1) 调用 `AmS` 的 `getRecentTask()` 方法获取最近的 `Task` 列表，这个列表来源于 `AmS` 中的 `mRecentTask` 列表。但是并不是直接返回 `mRecentTask` 里面的内容，而是把 `mRecentTask` 里面的 `TaskRecord` 数据类转换为 `ActivityManager.RecentTaskInfo` 数据类。

(2) 根据返回的 `RecentTaskInfo` 设置对话框中的六个按钮的显示和 `onClick()` 事件。显示包括任务名称和任务图标，而 `onClick()` 事件主要则是获取任务对应的 `Intent` 对象。

需要注意的是，任务的 `Intent` 对象是指该任务的 `baseIntent`，即创建该任务时的 `Intent`。同时，还给这个 `baseIntent` 添加了一个 `FLAG_NEW_TASK` 标识，添加这个标识的原因有两个，第一是系统希望在一个新的任务中启动 `baseIntent`，第二是当从 `Dialog` 类中调用 `startActivity()` 时只能使用 `Context` 类的 `startActivity()`，而 `Context` 的 `startActivity()` 方法要求 `Intent` 的标识中必须包含 `NEW_TASK`。

此时，长按“Home”键的操作就告一段落，接下来，如果用户单击任务对话框中的某个 `Button` 后，就会调用 `getContext().startActivity()` 启动对应的 `Activity`。注意，这里是使用 `Context` 中的 `startActivity()` 方法，而不是 `Activity` 的 `startActivity()` 的方法。

10.1.8 Activity 生命期的代码含义

在过去的应用程序开发中，读者大多数已经了解了 `Activity` 生命期中的几个主要状态，并知道如何在这些状态中做不同的事情。但多多少少还是存在一些疑惑，比如 `start` 和 `stop` 状态从代码的意义上来讲，差别到底在哪里。尽管你可能会说：“`stop` 代表了 `Activity` 的停止，而 `start` 代表了 `Activity` 的开始”，那么，问题是“开始”和“停止”的差别又在哪里？诸如此类，本节就来揭示这些不同状态背后所隐藏的代码级别的差别。

首先，不同的状态对于应用程序开发者来讲是通过不同的回调函数来区分的，因此，下面先列出所有和状态有关的回调函数，如表 10-4 所示。



表 10-4 Activity 中所有和状态相关的回调函数

回调函数名称	使用场合
onCreate()	Activity 实例被创建后第一次运行时
onNewIntent()	有两种情况会执行该回调： ● Intent 的 Flag 中包含 CLEAR_TOP，并且目标 Activity 已经存在当前任务队列中。 ● Intent 的 Flag 中包括 SINGLE_TOP，并且目标 Activity 已经存在当前任务队列中。 前者和后者的区别在于，举例如下：当前任务队列为 ABCD，此时目标 Activity 为 B，那么前者会把队列改变为 AB，而后者则会改变为 ABCDB；但假设当前为 ABCD，目标为 D，前者则会改变为 ABCD，后者则还是 ABCD，而不会重新创建 D 对象，即 SINGLE_TOP 只对目标在 top 上时才有效
onStart()	Activity 从 stop 状态重新运行时
onRestoreInstanceState(Bundle savedInstanceState)	与 onPostCreate()相同，只是先于 onPostCreate()调用
onPostCreate()	如果 Activity 实例是第一次启动，则不调用，否则，以后的每次重新启动都会调用
onResume()	Activity 继续运行时
onSaveInstanceState(outState)	与 onPause()相同，只是会先于 onPause()调用
onPause()	Activity 暂停时被调用。导致暂停的原因除了 onStop()中描述四个原因外，还包括一个，即当用户长按“Home”键出现最近任务列表时，此时正在运行的 Activity 将被执行 onPause()
onCreateDescription()	仅在要停止 Activity 时调用，先于 onStop()
onStop()	一般会导致变为 stop 状态的原因有以下几个： ● 用户按“Back”键后 ● 用户正在运行 Activity 时，按“Home”键 ● 程序中调用 finish()后 ● 用户从 A 启动 B 后，A 就会变为 stop 状态
onDestroy()	当 Activity 被销毁时，销毁的情况包括： ● 当用户按下“Back”键后 ● 程序中调用 finish()后

下面，按照 AmS 的调度过程重新总结以上回调函数在不同场合下的被调过程，如表 10-5 所示。

表 10-5 不同场合下回调函数被调用的顺序

场合描述	回调过程
创建 Activity 对象并启动	onCreate(); onStart(); onRestoreInstanceState(); onPostCreate(); onResume();



(续表)

场合描述	回调过程
Activity 对象已经存在并继续运行	onStart(); onResume();
Activity 对象已经存在, 并且满足表 10-4 中 onNewIntent() 的条件	onNewIntent(); onStart(); onResume();
AmS 中调用 startPausingLocked()	onSaveInstanceState(); onPause();
AmS 空闲时调用 stopActivityLocked()	如果没有暂停则先调用: onSaveInstanceState(); onPause(); 然后调用: onCreateDescription(); onStop();
AmS 空闲时调用 destroyActivityLocked()	如果没有暂停则先调用: onSaveInstanceState(); onPause(); 如果没有停止则继续调用: onCreateDescription(); onStop(); 然后调用: onDestroy();

以上就是 Activity 不同生命期对应的回调函数的具体执行过程。

10.2 内存管理

Android 中的内存管理分为两个部分。第一部分是当应用程序关闭后, 后台对应的进程并没有真正退出, 以便下次再启动时能够快速启动; 第二部分是, 当系统内存不够用时, AmS 会主动根据一定的优先规则退出优先级较低的进程。下面就分别讲述这两个部分的细节内容。

10.2.1 关闭而不退出

“关闭”仅仅是使其对应的窗口不显示, 而对应的进程却会一直保存。于是有的读者就开始怀疑, 如果后台有很多进程同时存在的话, 运行速度会变慢。事实上, 这种机制除了占用内存外, 基本上不会



降低前台程序的运行速度，下面就来分析为什么。

每个应用程序的主体都对应一个 `ActivityThread` 类，该类初始化后，就进入 `Looper.loop()` 函数中无限循环，如以下代码所示：

```
4622     Looper.prepareMainLooper();
4623
4624     ActivityThread thread = new ActivityThread();
4625     thread.attach(false);
4626
4627     Looper.loop();
```

以后则依靠消息机制运行，即当有消息处理时处理消息，而当没有消息时进程会进入到 `sleep` 状态。`loop()` 方法的内部代码如下所示：

```
106 public static final void loop() {
107     Looper me = myLooper();
108     MessageQueue queue = me.mQueue;
109     while (true) {
110         Message msg = queue.next(); // might block
```

在 `queue.next()` 方法中，会从消息队列中取消息，如果没有消息，该函数内部则会导致当前线程进入 `sleep` 状态，并且只有新消息到达后 `next()` 函数才继续执行并返回新的消息。为了验证这个结论，读者可以在 110 行的下一行代码处设一个断点，然后 debug 当前正在运行的任何应用程序，其结果是显而易见的。如果用户不和当前的 `Activity` 交互，程序是执行不到断点的，而当用户点击屏幕后，则会立即执行到断点。用户没有点击屏幕时，在 `ddms` 中查看应用进程的状态则显示为 `wait`，如图 10-16 所示，该图显示的是线程 id 为 134 的 `com.android.launcher` 应用程序。

 com.android.launcher	134		8603 / 8700		
ID	Tid	Status	utime	stime	Name
1	134	wait	372	94	main
*2	135	vmwait	14	34	HeapWorker
*3	138	vmwait	0	0	Signal Catcher
*4	144	running	11	17	JDWP
5	152	native	1	0	Binder Thread #1
6	153	native	0	0	Binder Thread #2

图 10-16 从 `ddms` 中查看线程 id 为 134 的线程状态

除此之外，也可以使用 `adb shell` 进入设备终端，然后使用 `ps` 命令查看当前进程的状态，同样可以发现当前进程为 `sleep` 状态，如图 10-17 所示。

```
root@android:/ # ps 134
USER      PID   PPID  VSIZE  RSS      WCHAN    PC         NAME
app_14    134    33    108136 23856    ffffffff afd0e828 S com.android.launcher
```

图 10-17 从 `adb shell` 中查看线程 id 为 134 的线程状态

而在 Linux 的内核调度中，如果一个线程的状态为 `sleep`，则除了占用调度本身的时间外，本身则不会占用 CPU 的时间片。因此，在 100 以内的进程数目基本上不会影响当前进程的执行速度，换句话说，系统运行一个进程与 100 个进程的速度是相同的，只要其他 99 个线程都处于 `sleep` 状态。所以，假如有人说他可以提供一个进程查看器，在里面可以杀死不用的进程以提高系统运行速度，读者别相信这个。

有的读者就要问了，当前进程都处于 `sleep` 状态了，那还怎么执行啊？这正是所谓的消息驱动机制，`queue.next()`方法可以被以下三种情况重新唤醒。

定时器中断。如果应用程序中设置了定时器任务，那么当定时器发生时，操作系统会回调该定时器任务，程序员一般会在这个定时器任务中向 `looper` 主线程中发送一个消息，从而 `next()`方法会被重新唤醒，并返回所获得的消息。

用户按键消息。当有用户按键消息时，`WmS` 中的专门处理输入消息的线程会把这个消息发送到 `looper` 主线程，这就是上面例子中所说的，当点击屏幕时会从 `next()`方法中跳出。

Binder 中断。Binder 是 Linux 系统中的一个驱动设备，应用程序中可以包含一个 Binder 对象，该对象会在应用程序中自动创建一个相应的线程，当 Binder 驱动接收到 Binder 消息，并派发到客户端的 Binder 对象后，Binder 线程会开始执行。如果在 Binder 的消息处理中向 `looper` 主线程发送一个消息，而 `next()`会继续执行。

以上三种情况的关系如图 10-18 所示。

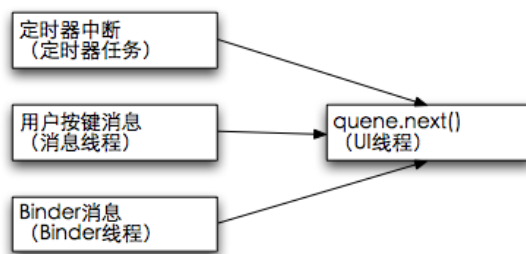


图 10-18 各种能触发 `MessageQueue` 从 `next()`函数返回的情况

基于以上认识，Android 系统设计时才考虑了让应用程序“关闭”而“不退出”，因为这不会降低程序运行的速度。当然，如果所有的应用程序都是某种服务，而不是 UI 程序，那么在这种设计模式下，当应用程序增加时，每个程序的运行速度会立即变慢，CPU 会被这些应用程序“瓜分”。幸运的是，手机不是服务器，大多数应用程序都是消息驱动模式，没有消息时程序就会 `sleep`，直到产生消息。

10.2.2 Android 与 Linux 的配合

关于内存回收的时机，Android 官方文档提出，`Activity` 所占用的内存在一般情况下不会被回收，



只有在系统内存不够用的时候才会回收，并且回收时会会有一个潜规则。大致情况是前台 Activity 最后回收，其次是包含前台的 Service 或者 Provider，再次是后台 Activity，最后是后台空进程。而这种机制就意味着 AmS 需要和 Linux 操作系统有个约定，因为“系统内存是否够用”属于 Linux 内核的内存管理控制的事情，AmS 是无法预知的。

你可能会想，AmS 是运行在 Java 环境中的，当 Java 进行内存分配时，如果发生 Out Of Memory 不就知道系统内存低了吗？事实并非如此，这有两个原因：

- 尽管 AmS 运行在 Java 环境里，但是应用程序和 AmS 运行在两个独立的 Java 虚拟机中，应用程序申请内存并不会通知 AmS，所以 AmS 无法感知应用程序申请内存的状况，除非每次应用程序申请内存并发生 Out Of Memory 时通知 AmS，但实际却没有。
- Java 虚拟机运行时都有各自独立的内存空间，应用程序 A 发生 Out Of Memory 并不意味着应用程序 B 也会发生 Out Of Memory，很有可能仅仅是 A 程序用光了自己内存的上限，而系统内存却还是有的。

以上说明，单纯的 AmS 是无法获知系统内存是否低的。

那么，什么是“系统内存低”或者“系统内存不够用”呢？从 Android 底层的 Linux 来讲，由于其并未采用磁盘虚拟内存机制，所以应用程序能够使用的内存大小完全取决于实际物理内存的大小，所以，“内存低”的含义就是实际物理内存已经被用得所剩无几了。而这又是如何与 AmS 联系到一起的呢？如图 10-19 所示。

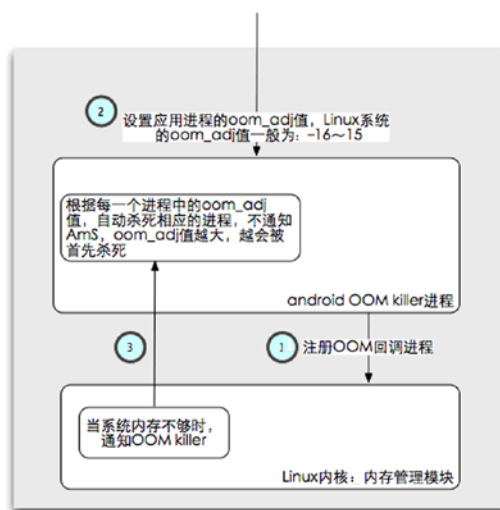


图 10-19 Android 中内存 OOM 机制示意图

在 Android 中运行了一个 OOM 进程，即 Out Of Memory。该进程启动时会首先向 Linux 内核中把自己注册为一个 OOM Killer，即当 Linux 内核的内存管理模块检测到系统内存低的时候就会通知已经注册的

OOM 进程，然后这些 OOM Killer 就可以根据各种规则进行内存释放了，当然也可以什么都不做。

Android 中的 OOM Killer 进程是仅仅适用于 Android 应用程序的，该进程在运行时，AmS 需要把每一个应用程序的 oom_adj 值告知给 Killer。这个值的范围在 -16 到 15，值越低，说明越重要，这个值类似于 Linux 系统中的进程 nice 值，只是在标准的 Linux 中，有其自己的一套 Killer 机制。

当发生内存低的条件时，Linux 内核管理模块通知 OOM Killer，Killer 则根据 AmS 所告知的优先级，强制退出优先级低的应用进程。

以上就是 Android 和 Linux 关于内存管理的配合，分析完相关代码可知，OOM Killer 是一种可选的方案，对于 OEM 厂商来讲，完全可以不支持 OOM，只有在不支持的情况下，Android 才会采用之前所声称的“潜规则”优先退出后台的 Activity。这个结论很重要，因为在现在主流的 Android 平台中都包含了 OOM Killer 模块，这就意味着 AmS 仅仅是根据应用进程前后台的关系给其分配一个 oom_adj 值，剩下的就是 oom_killer 要干的事情了。

10.2.3 各种关闭程序的过程

为了总体了解内存回收的细节，本节先对和内存回收相关的程序关闭过程作一个总体概括，如图 10-20 所示。

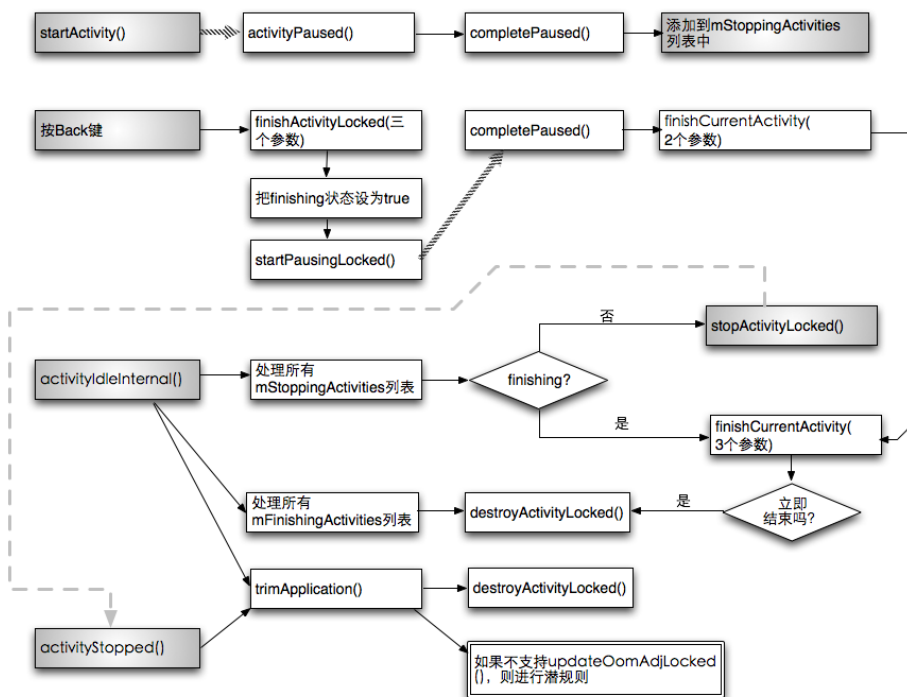


图 10-20 各种关闭当前 Activity 操作引起的内存回收流程

关闭程序的所有过程包含以下三种情况。

第一种，从调用 `startActivity()` 开始，在一般情况下，当前都有正在运行的 Activity，所以需要先暂停当前 Activity，而暂停完毕后，AmS 会收到一个 Binder 消息，并开始从 `completePaused()` 处执行。在该函数中，由于上一个 Activity 并没有 finishing，仅仅是 stop，所以这里会把上一个 Activity 添加到 `mStoppingActivities` 列表中。当目标 Activity 启动后，会向 AmS 发送一个请求进行内存回收的消息，这会导致 AmS 在内部调用 `activityIdleInternal()` 方法，该方法中首先会处理 `mStoppingActivities` 列表中 Activity 对象，这就会调用到 `stopActivityLocked()` 方法。这又会通过 IPC 调用，通知应用进程 stop 指定的 Activity，当 stop 完毕后，再报告给 AmS，于是 AmS 再从 `activityStopped()` 处开始执行，而这会调用 `trimApplications()` 方法，该方法中则会执行和内存相关的操作。

第二种，当按“Back”键后，会调用 `finishActivityLocked()`，然后把该 Activity 的 finishing 标识设为 true，然后再调用 `startPausingLocked()`，当目标 Activity 完成暂停后，就会报告 AmS，此时 AmS 又会从 `completePaused()` 处开始执行。与第一种情况不同，由于此时暂停的 Activity 的 finishing 状态已经设置为 true，所以会执行 `finishingActivityLocked()`，而不是像第一种情况中仅仅把该 Activity 添加到 `mStoppingActivities` 列表。

第三种，当 Activity 启动后，会向 AmS 发送一个 Idle 消息，这会导致 AmS 开始执行 `activityIdleInternal()` 方法。该方法中会首先处理 `mStoppingActivities` 列表中的对象，接着处理 `mFinishingActivities` 列表，最后再调用 `trimApplication()` 方法。

以上即为所有关闭程序的情况，包括 stop 和 destroy，客户进程中与之对应的就是 `onStop()` 和 `onDestroy()` 的调用。

而从内存回收的角度来看，释放内存的地点包含三个。

- 第一个是在 AmS 中进行，即 Android 所声称的当系统内存低时，优先释放没有任何 Activity 的进程，然后释放非前台 Activity 对应的进程。
- 第二个是在 OOM Killer 中，此时 AmS 只要告诉 OOM 各个应用进程的优先级，然后 OOM 就会调用 Linux 内部的进程管理方法杀死优先级较低的进程。这个部分不在本书讲解的范围内。
- 第三个是在应用进程本身之中，当 AmS 认为目标进程需要被杀死时，首先会通知目标进程进行内存释放，这包括调用目标进程的 `scheduleLowMemory()` 方法和 `processInBackground()` 方法。

10.2.4 释放内存详解

上一节介绍了各种关闭、退出程序的过程，本节介绍这些操作过程中所导致的各种具体内存回收的细节，如附图 4 所示。

1. activityIdleInternal()

首先来看 `activityIdleInternal()` 方法的调用，该方法被调用的时机如图 10-21 所示。

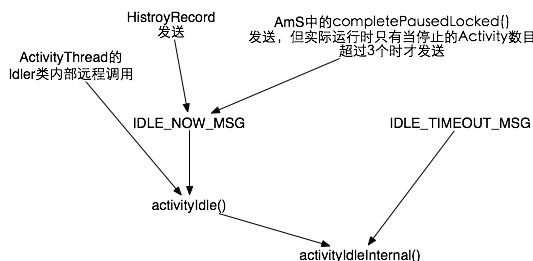


图 10-21 引起 activityIdleInternal()被调用的各种原因

最常见的调用该方法的情况发生在目标 Activity 启动后，这是在 ActivityThread 中调用的，即当指定的 Activity 启动后会报告 AmS，因为此时上一个 Activity 仅仅处于 pause 状态，还没有 stop 或者 destroy。这也是 AmS 的调度特点之一，即优先启动下一个 Activity，只有当启动后，才通知 AmS 去 stop 或者 destroy 上一个 Activity，从而让用户能在最短的时间内看到下一个 Activity 的界面。ActivityThread 中调用该方法的代码如下所示：

```

3226     Looper.myQueue().addIdleHandler(new Idler());
2170     am.activityIdle(a.token,

```

- 首先在 handleResumeActivity()方法中添加一个 Idler 对象，然后在该 Idler 对象的 queueIdle()方法中调用 AmS 的 activityIdle()方法。

除此之外，activityIdle()还会在另外两种情况下被调用，这两种情况都是向 AmS 所在的线程发送一个异步 IDLE_NOW_MSG 消息，而消息的处理函数正是 activityIdle()。

- 一种情况是在 HlstoryRecord 类中的 windowVisible()函数中调用，而这个函数是从 WmS 中调用的，即当目标窗口由“非显示”变为“显示”状态时，会告诉 AmS。因为当窗口显示后如果没有用户操作的话，可认为是空闲状态，这有点类似于在 ActivityThread 中 resume 完指定的 Activity 后也被认为是空闲状态一样。有的读者可能就要问了：对于一般的 Activity 而言，既然 resume 后要通知 AmS 进行空闲回收，而当其窗口显示出来后也要通知 AmS 进行空闲回收，岂不是有点重复？原因是有些窗口仅仅是窗口，而不对应任何 Activity，并且 activityIdle()方法内部并不是仅仅回收刚刚暂停的 Activity，而是整个系统内部的状态检查。
- 另一种情况是 completePausedLocked()方法中，在正常的操作中，似乎不会执行到这段代码，如下所示：

```

2217     mStoppingActivities.add(prev);
2218     if (mStoppingActivities.size() > 3) {
2219         // If we already have a few activities waiting
2220         // then give up on things going idle and start
2221         // them out.
2222         if (DEBUG_PAUSE) Slog.v(TAG, "Too many pending s
2223         Message msg = Message.obtain();
2224         msg.what = ActivityManagerService.IDLE_NOW_MSG;
2225         mHandler.sendMessage(msg);
2226     }

```

这段代码意思是，如果当前要 stop 的 Activity 数量超过 3 时，才发送 IDLE_NOW_MSG 消息。而实际系统运行时，由于每一次 activity 启动后，都会调用到 AmS 的 activityIdle() 方法，而该方法中则会处理 mStoppingActivities 列表中的对象。所以，单次 completePausedLocked() 调用时，一般不会积攒到 3 个，而仅仅是 1 个，所以这段代码似乎永远执行不到。

以上三种情况都是通过 activityIdle() 间接调用到 activityIdleInternal() 方法，另外一种调用到 activityIdleInternal() 方法的情况是发生在 IDL_TIMEOUT_MSG 消息中。

- 第四种情况，该消息是在 completeResumeLocked() 方法中发出的，其作用是检测启动超时。即如果规定的时间(IDLE_TIMEOUT)内不能启动指定的 activity，AmS 中设置的是 10 秒，则会调用 activityIdleInternal() 释放相关资源。completeResumeLocked() 方法是在 resumeTopActivityLocked() 中被调用的。

以上四种情况就是全部调用到 activityIdleInternal() 的情况，下面就来分析该函数内部是如何实现内存回收的。

① 通知所有需要回收内存的客户进程进行内存回收。这些客户进程列表保存在 mProcessesToGc 列表中，关于客户进程中如何进行内存回收参照 10.2.4 节。

② 取出 mStoppingActivities 列表中的内容，并存放临时列表 stops 中，再取出 mFinishingActivities 列表中的内容，并存放临时列表 finishes 中，然后删除原有的记录。这里并不是简单的列表内容复制，而是要判断里面 HistoryRecord 的状态，源码比较简单。

③ 首先对 stops 列表中的内容进行处理，在该处理过程中，有一个判断列表中 Activity 的 finishing 状态是否为 true 的条件。这里需要注意，有的读者可能会觉得奇怪，因为 stops 列表的内容来源于 mStoppingActivities 中，该列表的内容一般是指 finishing 状态为 false，并且 stopped 状态为 true，既然这两个状态都是事先确定的，那么为什么此处还要判断呢？须知，mStoppingActivities 列表中的对象的 finishing 状态不一定全部都是 false，比如在按下“Back”键后，finishing 状态 true，接着在 completePaused() 中调用到了 finishCurrentActivity (3 个参数) 函数，该函数中则会把指定的 Activity 添加到 mStoppingActivities 列表中，显然，此时该 Activity 的 finishing 状态为 true。

在该步中，如果 finish 状态为 false，则调用 stopActivityLocked() 通知客户进程停止该 Activity，这种情况一般发生在调用 startActivity() 后。如果 finish 状态为 true，则执行 f2()、f3() 函数，在 f3() 函数内部则要判断是不是要立即停止，如果要立即停止则调用 destroyActivityLocked() 通知目标进程调用 onDestroy() 方法，否则，先调用 resumeTopActivity() 运行下一个 Activity。

④ 接着对 finishes 列表中的对象进行处理。由于 finishes 列表中的对象其 finishing 状态都为 true，所以，可以直接调用 destroyActivityLocked() 通知客户进程销毁目标 Activity。

以上四步中，除了第一步是通知客户进程进行主动内存回收外，其他三步仅仅是针对不同状态通知客户进程执行不同的回调而已，而没有进行任何内存回收工作。如果 Android 的程序员愿意，他们可以给这三个步骤起任何一个名称，换句话说，stop 和 destroy 仅仅是一个状态的变化，而没有在真正意义上对内存进行实质性的停止或者销毁。

- ⑤ 调用 trimApplications()。该步骤内部真正地进行了内存回收工作，包括杀死不必要的进程等，



详见以下两个小节。

2. trimApplications()

该方法除了在 `activityIdleInternal()` 中被调用外，也在 `stopActivityLocked()` 中被调用，其内部执行流程如附图 4 所示。以下为了叙述方便，截取和该函数相关的大图片段，如图 10-22 所示。

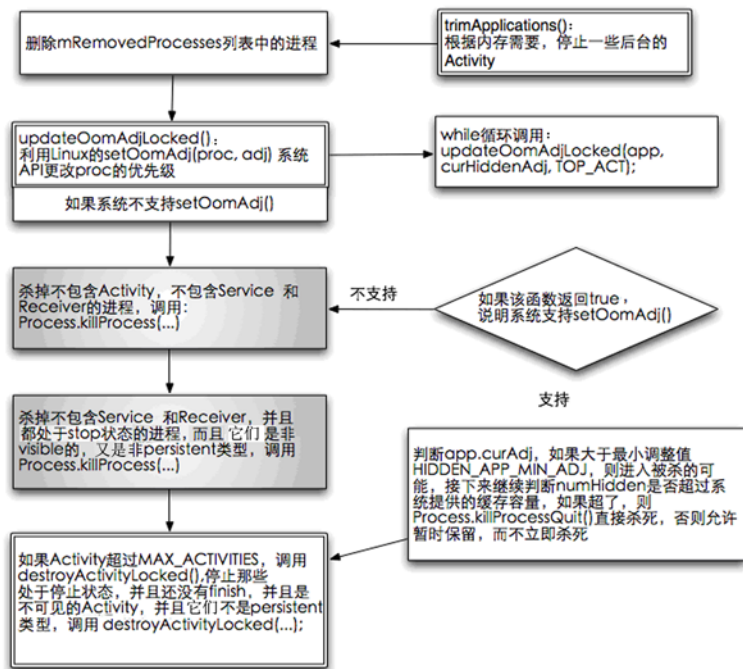


图 10-22 trimApplication()的内部调用过程

- ① 删除 `mRemovedProcesses` 列表中包含的应用进程，该列表的内容来自四种情况。
 - 第一种是当某个进程 crash 后，会被添加到该列表。
 - 第二种是当某个程序的 UI 线程在 5 秒之内没有响应时，系统会弹出一个 ANR 对话框，此时如果用户选择强制关闭，该进程就会被添加到该列表。
 - 第三种是当程序员调用 AmS 提供的 `killBackgroundProcess()` 方法时，调用者必须包含 `KILL_BACKGROUND_PROCESSES` 权限。这说明，程序员调用 `killBackgroundProcess()` 方法后并不会同步杀死指定进程，而仅仅是添加到列表。
 - 第四种是当系统启动时，AmS 的 `systemReady()` 方法中，如果发现启动了非 `persistent` 类型的应用进程，则把这些进程添加到列表中，这种情况基本上不会发生。
- ② 调用 `updateOomAdjLocked()` 方法，该方法的作用是告诉 OOM Killer 指定进程的优先级，即

oom_adj 值，该值的范围为-16~15，Android 中的进程仅仅使用了 0~15。值越小说明优先级越高，越不能被优先杀死。该函数的返回值类型为 boolean，如果底层的 Linux 系统包含 OOM Killer 则返回 true，否则返回 false。关于该方法内部的具体执行过程参照下一小节。

③ 如果第二步中不支持 OOM，则要执行 AmS 内部的“潜规则”，即 Android 官方宣称的优先杀死后台 Activity 的具体过程，详细分析见第 4 小节。

④ 最后，无论是使用 OOM 机制还是“潜规则”，杀死后台进程后，如果此时运行的 Activity 数量依然超过 MAX_ACTIVITES（20），则需要继续销毁满足以下三个条件的 Activity。

- Activity 必须已经 stop，但却没有 finishing。
- 必须是不可见的，即该 Activity 窗口上面有其他全屏的窗口，如果不是全屏，则后面的 Activity 是可见的。
- 不能是 persistent 类型，即常驻进程不能被杀死。

trimApplications()方法执行完毕后，如果依然内存不够用，那就无能为力了，只能说明要么是硬件内存太少，要么是应用程序设计不合理，比如 persistent 的应用太多等。

3. updateOomAdjLocked()

该方法是在 trimApplication()中被调用的，其作用是告诉 OOM 进程指定应用进程的优先级。介绍该函数内部的逻辑之前需要先理解 ProcessRecord 类中和 OOM 模块相关的几个变量的含义，如表 10-6 所示。

表 10-6 ProcessRecord 类内部和 OOM 相关的变量

变量名称	含 义
curAdj	当前的 adj 值
setAdj	上一个设置值
setRawAdj	curAdj 的含义是实际该 app 的 adj 值，这个值经过了两个过程的评估，这在 computeOomAdjLocked()方法中有详细介绍，而 rawAdj 是在第一个过程评估后认为该 app 应该有的优先级，而实际的优先级还要经过第二个过程的调整
curRawAdj	
setShedGroup	上一个设置值
curShedGroup	当前所在的调度组
hiddenAdj	该变量不是反映当前进程的 hidden 值，而是反映了当前系统中的 hidden 值，即当前系统中如果 adj 值大于 hiddenAdj，则都处于隐藏状态

应用进程的 adj 值是可以调整的，以达到动态调整应用进程优先级的目的；schedGroup 也是可以调整的；hiddenAdj 保存着当前系统中的 hidden 边界。

updateOomAdjLocked() 函数中，循环对 mLruPrcesses 列表中的所有进程依次调用 updateOomAdjLocked(..)，注意这是两个同名函数，前者没有参数，后者的参数包含了指定的客户进程。本节主要讨论的是后者，其执行过程主要包含两大步。

- ① 调用 computeOomAdjLocked()方法，估算指定进程的 oom_adj 值，估算的流程如下。



(1) 判断该进程是否是 TOP_APP。TOP_APP 是一个局部变量，保存了当前正在运行的 ProcessRecord 对象，如果是，那么其优先级自然是最高，为常量 FOREGROUND_APP_ADJ，该值是在 Linux 系统启动后在 init.rc 文件中初始化的，其值为 0。以下截取 init.rc 的代码片段：

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
```

(2) 判断该进程中是否包含 instrumentationClass，该值一般在 UnitTest 进程中会存在，而在一般的应用程序中都不包含该对象。

(3) 判断进程中是否包含持久的 Activity 对象，如果有，其优先级也是最高的。

(4) 判断是否包含正在执行的 Receiver 对象，如果有，也是最高优先级。

(5) 判断当前是否有正在执行的 Service，如果有，也是最高优先级。

以上五步中，adj 的赋值都是 FOREGROUND_APP_ADJ，即在这五种情况下，指定的客户进程都不能被杀死，其优先级最高，而且平等。

(6) 判断指定的进程是否正在调用前台进程的 service 对象。如果是的话，则其优先级为 VISIBLE_APP_ADJ，该值为 1，说明这种情况的进程优先级略低于前台进程，但因为它和前台进程正处于互动，所以还是有一定的优先级。

(7) 判断 forcingToForeground 变量是否为空，如果不为空，说明该进程正在被用户强制调到前台。这种情况只是瞬间的，当然如果是这样，其优先级自然比较高，为 VISIBLE_APP_ADJ。

(8) 判断是否为 Home 进程，是的话优先级调整为 HOME_APP_ADJ，其值为 4。有些读者觉得奇怪，为什么 Home 进程的优先级还低于前面的情况呢？原因是，如果当前正在前台执行 Home 程序，其优先级自然是前面几步的情况，而如果不是前台进程的话，其优先级仅仅比普通进程高一点点而已。

(9) 如果当前进程包含的 Activity 数目大于 0，并且包含了 visible 的 Activity，那么其优先级为 VISIBLE_APP_ADJ。如果没有 visible 的 Activity，则意味着该进程可以被隐藏了，所以优先级为 hiddenAdj，即当前系统中的隐藏值。

(10) 对于其他情况，说明仅仅是一个空进程，优先级最低，此时设置 app.empty 变量为 true，并且其 adj 值为 hiddenAdj。

以上十步处理的是进程中包含处于运行状态对象的情况，此时已经获得了一个 adj 值，而这个值正是前面所说的 rawAdj 值的含义，也就是源码注释中所说的 unlimited，即无限制条件下应有的优先级值，如下代码所示：



```

13895     app.adjSeq = mAdjSeq;
13896     app.curRawAdj = adj;

```

```

58     int curRawAdj;           // Current OOM unlimited adjustment for this process
59     int setRawAdj;           // Last set OOM unlimited adjustment for this process

```

除此之外，进程中还包含了一些不处于运行状态的对象，包括 service 和 provider 两种。因此，接下来继续根据这个条件设置 adj 值。

(1) 判断是否为 mBackupTarget 进程。由于“备份进程”要在后台持续收集数据，因此，其优先级应该高于一般的进程，为.BACKUP_APP_ADJ(2)，但却要低于 VISIBLE 进程。

(2) 处理当前进程中包含非前台 service 的情况。所谓的非前台 service 是指该 service 不属于前台进程，也没有和前台进程正在交互，其处理过程如下面子流程。

① 首先以下所有处理均在该 app 包含的 service 对象中，并且其优先级不是前台进程 FOREGROUND_APP_ADJ 或者所在调度组为“背景非交互”类型，如以下代码所示：

```

13908     if (app.services.size() != 0 && (adj > FOREGROUND_APP_ADJ
13909         || schedGroup == Process.THREAD_GROUP_BG_NONINTERACTIVE)) {

```

② 循环针对每一个 service 的状态调整 adj 值。在每一个 service 对象中，判断该 service 是否被请求启动过，即判断 s.startRequested 是否为 true。如果为 true，说明有某个进程请求启动过该 service，否则该 service 还没有被启动过。如果已经启动，并且还没有超出“待机时间 (MAX_SERVICE_INACTIVITY)” 30 分钟，那么其优先级会被设置为 SECONDARY_SERVER_ADJ，仅次于 VISIBLE。

③ 处理和该 service 有连接的客户端(client)，并根据其客户端的优先级调整该 service 的优先级。首先判断客户端是否是该进程本身，即在同一个进程中调用其内部的 service，这种当然就不用再处理了，保持已有的优先级即可，如以下代码所示：

```

13942         if (cr.binding.client == app) {
13943             // Binding to ourself is not interesting.
13944             continue;
13945         }

```

④ 判断该 service 是否以 BIND_AUTO_CREATE 方式启动，如果是则说明该 service 的重要程度取决于客户端本身，否则不做任何处理。处理的逻辑是，首先根据 client 的 hiddenAdj 值调整当前的 hiddenAdj 值，规则是如果当前值大于 client 的值，则把当前值调为 client 值，但不能大于 VISIBLE_APP_ADJ。

获得了新的 myHiddenAdj 值后，以此为新的参数，递归调用 computeOomAdjLocked(..., myHiddenAdj, ...)，并计算该 client 的 adj 值。同样，得到返回的 adj 后，如果当前 adj 大于 client 的 adj，则使用 client 的 adj 作为该 app 的 adj。

以上规则的另一个意思就是，如果当前 app 没有 client 重要，则把其重要性调整成和 client 相同重要，否则保持当前的重要性。

⑤ 判断这个客户连接是否有对应的 Activity，并且该 Activity 是否处于 RESUMED 状态或者正在暂停 (PAUSING) 状态。如果是的话，则说明这个客户连接很重要，同样，该 app 也就很重要了，优先级重新设置为 FOREGROUND_APP_ADJ。

⑥ 估算完 service 后，如果 adj 值大于 hiddenAdj，则使用 hiddenAdj 作为 adj 值。这样仅仅是为了避免“滥杀无辜”，因为如果不调整 adj 为 hiddenAdj，那么该进程的优先级会很低，导致会被优先杀死，然而，既然该 app 包含了 service，所以可以暂时让其存在，以便后面再使用时能够快速启动。

(3) 接下来处理 provider 的情况。其处理逻辑和处理 service 的基本相同，不同处在于，ContentProviderRecord 对象有一个 externals 对象。这是一个 int 值，代表了连接该 provider 的非 framework 的进程 id。这是什么意思呢？即可以使用 C/C++ 写一个 Linux 应用程序，运行于 Android 底层的 Linux 系统之上，并且使用 Android 所提供的底层 provider 库来访问系统中已有的 provider 对象，而这个 externals 就是这个非 framework 进程的 id 号。如果有这种情况存在，则调整 adj 的优先级为最高，因为 framework 并不想破坏 native 进程的访问，毕竟既然能用 native 去访问，足见该 provider 的重要性。

(4) 以上步骤终于计算出了一个合适的 adj 值，最后要做的就是查看该 adj 值是否超过了 app 所指定的最大值，即 maxAdj。如果超过了，则赋值为 maxAdj，该变量是在该 app 进程启动时指定的，其含义是就像在说：“如果系统需要，可以降低我的重要性，但是不能超过 maxAdj，这是我的容忍极限。”

(5) 此时，adj 已经是大家都满意的 adj 值了，于是把它赋值给 app 的 curAdj 值，并把 schedGroup 赋值给 app 的 curSchedGroup 变量。

以上仅仅是 updateOomAdjLocked() 的第一大步而已，接下来看第二步。

② 判断当前 app 是否要从前台切换到后台，判断的方法是对于 curRawAdj 的 setRawAdj，前者代表了当前 rawAdj，后者代表了上一个 rawAdj。这里值得疑惑的是，为什么要使用 rawAdj 而不是 adj，因为 adj 才是该 app 最终的优先级值？尽管在一般情况下，rawAdj 的值和 adj 的相等。如果是的话，则需要通知客户进程进行主动内存回收，通知调用 scheduleAppGcLocked()，该函数将在 5 小节中进行详细介绍。

③ 查看真正的优先级 adj 是否改变，如果改变，则调用 Process.setOomAdj() 通知 OOM Killer 进程当前 app 的新优先级。当然如果系统不支持 OOM 管理，则该方法会返回 false，相应的，updateOomAppLocked() 则也会返回 false。

④ 查看“所在调度组” curSchedGroup 是否变化，如果变化，则调用 Process.setProcGroup 改变当前 app 进程所在的用户组。

至此，指定进程的 updateOomAppLocked(app,...) 调用就结束了，正常结束时，会返回 true，说明系统是支持 OOM 的。

下面再回到 updateOomAppLocked(无参数)方法中。处理完一个 app 进程后，要继续判断该 app 进程的 curAdj 是否大于 HIDDEN_APP_MIN_ADJ (一般为 7)，如果大于则说明该 app 可以隐藏。接着需要查看总的隐藏数量是否超过系统的最大容量值 MAX_HIDDEN_APPS (一般为 15)，如果超过，则立即杀死该进程，并且将其 ProcessRecord 对象的 killedBackground 变量置为 true，代表该进程是被后台杀死的，如下代码所示。



```

14350         if (app.curAdj >= HIDDEN_APP_MIN_ADJ) {
14351             if (!app.killedBackground) {
14352                 numHidden++;
14353                 if (numHidden > MAX_HIDDEN_APPS) {
14354                     Slog.i(TAG, "No longer want " + app
14355                         + " (pid " + app.pid + "):
14356                     EventLog.writeEvent(EventLogTags.AM
14357                         app.processName, app.setAdj
14358                     app.killedBackground = true;
14359                     Process.killProcessQuiet(app.pid);

```

以上就是 updateOomAdjLocked（无参数）的内部执行过程。

4. AmS 内部内存回收的潜规则

在 Android 的官方文档中关于内存回收大致是这样描述的：系统按照以下优先级关闭进程以释放内存，程序员不需要主动去关心退出进程的事情。

- 前台进程(forground process)，是指那些和用户正在做的事情相关的进程，具体包括：
 - 正在和用户交互的 Activity，即该 Activity 的 onResume()已经执行过。
 - 包含一个 service，该 service 正在服务于和用户交互的 Activity。打个比方，该进程可能不是“领导”，但是却正在和领导谈话，则它依然很重要。
 - 包含一个 service，该 service 正在执行 onCreate()，或者 onStart()，或者 onDestroy()。
 - 包含一个 BroadcastReceiver，正在执行 onReceive()函数。
- 可视进程(visible process)，尽管没有和用户交互，但是却可以影响用户所能看得到的内容。
 - 尽管没有包含和用户交互的 Activity，但是用户却可以看得见该 Activity 的窗口，比如一个 Activity 上面弹出一个对话框的情况。
 - 包含一个 service，该 service 服务于可视的 Activity，即上面说的看得见却不能交互的 Activity。
- 服务进程(service process)，凡是使用 startService()所启动的 service 对象，其所在的进程都称之为服务进程。当然，如果该 service 满足上面两个优先级中的条件，则会上升为相应的优先级。
- 后台进程(background process)，不满足以上任何条件的进程，同时该进程中还包含一些不可见的 Activity，这些进程不影响正在和用户交互的 Activity。
- 空进程(empty process)，进程中不包含任何 component，包括 Activity、service、receiver 对象。之所以还保留这些进程的原因是为了减少重新创建该进程的开销，创建一个空进程的开销包括创建进程本身，以及加载该应用中包含的 resources.arsc 资源文件，这些都是比较耗时的。

通过分析 AmS 源码可知，以上规则仅适用于支持 OOM 模块的系统，当然从目前来看，似乎大多数 Android 系统的底层都包含了该模块。关于支持 OOM 时，AmS 按照以上规则给应用进程设置优先级的细节见第 3 小节，这是在 updateOomAdjLocked()方法中完成的。本节主要是结合源码对以上规则中

的部分规则作一个代码层面上的说明。

- 前台进程。
 - “正在执行一个 service 的回调”，对应的代码如下：

```

13839     } else if (app.executingServices.size() > 0) {
13840         // An app that is currently executing a service
13841         // counts as being in the foreground.
13842         adj = FOREGROUND_APP_ADJ;
13843         schedGroup = Process.THREAD_GROUP_DEFAULT;
13844         app.adjType = "exec-service";
13845     } else if (app.foregroundServices) {

```

ProcessRecord 中的 executingServices 是一个 ArrayList 类型变量，当 AmS 中执行 startService 或者 stopService 时，对应的 service 都会被临时添加到该列表中，直到执行完毕。

- “包含一个正在执行的 Receiver 对象”，对应的代码如下：

```

13832     } else if (app.curReceiver != null ||
13833         (mPendingBroadcast != null && mPendingBroadcast.curApp == app)) {
13834         // An app that is currently receiving a broadcast also
13835         // counts as being in the foreground.
13836         adj = FOREGROUND_APP_ADJ;
13837         schedGroup = Process.THREAD_GROUP_DEFAULT;
13838         app.adjType = "broadcast";

```

ProcessRecord 中的 curReceiver 变量是当执行某个 onReceiver() 方法时被置为 true，执行完毕后重新设置为 false。

- 可视进程。
 - “尽管不是前台 Activity，但是却是可视”，如以下代码所示：

```

13850     } else if (app.forcingToForeground != null) {
13851         // The user is aware of this app, so make it visible.
13852         adj = VISIBLE_APP_ADJ;
13853         schedGroup = Process.THREAD_GROUP_DEFAULT;
13854         app.adjType = "force-foreground";
13855         app.adjSource = app.forcingToForeground;

```

ProcessRecord 对象中的 forcingToForeground 变量是一个 IBinder 类型，这是导致该 Activity 可视而不可操作的 Binder 对象。读者们可能会觉得奇怪，为什么一个弹出的对话框还会对应一个 Binder 对象呢？这个所说的弹出对话框并不是 Dialog 类，而是一个具有 Dialog 风格的 Activity 对象，而这个 Binder 对象正是对应了该 Activity 在 AmS 中对应的 HistoryRecord 对象。

- “服务于可视 Activity 的 Service 对象”，如以下代码所示：

```

13845     } else if (app.foregroundServices) {
13846         // The user is aware of this app, so make it visible.
13847         adj = VISIBLE_APP_ADJ;
13848         schedGroup = Process.THREAD_GROUP_DEFAULT;
13849         app.adjType = "foreground-service";

```



- “服务进程”，如以下代码所示：

```

13908         if (app.services.size() != 0 && (adj > FOREGROUND_APP_ADJ
13909             || schedGroup == Process.THREAD_GROUP_BG_NONINTERACTIVE)) {
13910             final long now = SystemClock.uptimeMillis();
13911             // This process is more important if the top activity is
13912             // bound to the service.
13913             Iterator jt = app.services.iterator();
13914             while (jt.hasNext() && adj > FOREGROUND_APP_ADJ) {
13915                 ServiceRecord s = (ServiceRecord)jt.next();
13916                 if (s.startRequested) {
13917                     if (now < (s.lastActivity+MAX_SERVICE_INACTIVITY)) {
13918                         // This service has seen some activity within
13919                         // recent memory, so we will keep its process ahead
13920                         // of the background processes.
13921                         if (adj > SECONDARY_SERVER_ADJ) {
13922                             adj = SECONDARY_SERVER_ADJ;
13923                             app.adjType = "started-services";
13924                             app.hidden = false;

```

如果包含服务进程，其优先级设为 SECONDARY_SERVER_ADJ，其值为 2，仅次于 VISIBLE 进程。

- 后台进程。后台进程并没有在 updateOomAdjLocked(三个参数)函数中处理，而是在 updateOomAdjLocked(无参数)函数中处理。当对比当前 app 的 curAdj 大于 HIDDEN_APP_MIN_ADJ 时，则认为该 app 处于可隐藏状态，此时如果隐藏的 app 数量大于 MAX_HIDDEN_APPS，则立即杀死该 app 进程，如以下代码所示：

```

14350         if (app.curAdj >= HIDDEN_APP_MIN_ADJ) {
14351             if (!app.killedBackground) {
14352                 numHidden++;
14353                 if (numHidden > MAX_HIDDEN_APPS) {
14354                     Slog.i(TAG, "No longer want " + app.processName
14355                         + " (pid " + app.pid + "): hidden #" + numHidden);
14356                     EventLog.writeEvent(EventLogTags.AM_KILL, app.pid,
14357                         app.processName, app.setAdj, "too many background");
14358                     app.killedBackground = true;
14359                     Process.killProcessQuiet(app.pid);

```

- “空进程”在代码中是指 ProcessRecord 中的 thread 变量为空，此时尽管 thread 已经为空，但该客户进程可能还依然存在。在这种情况下，只需要把该进程的 adj 设置为 EMPTY_APP_ADJ 即可，如以下代码所示：

```

13790         if (app.thread == null) {
13791             app.adjSeq = mAdjSeq;
13792             app.curSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
13793             return (app.curAdj=EMPTY_APP_ADJ);
13794         }

```

EMPTY_APP_ADJ 的值为 15，这个优先级是最低的，剩下的杀死该进程的任务则交给 OOM 模块了。当然，如果系统不支持 OOM，那么实际上则没有“空进程”的概念。

以上讨论了 OOM 模式下的“潜规则”，而如果系统不支持 OOM，则 AmS 会采用一种更为简单的方式杀掉进程。

① 杀掉那些非 persistent 类型，不包含任何 Activity，并且当前没有正在执行的 Receiver 对象，内部也不包含 service 对象的进程，如以下代码所示：

```

14446         final ProcessRecord app = mLruProcesses.get(i);
14447         // Quit an application only if it is not currently
14448         // running any activities.
14449         if (!app.persistent && app.activities.size() == 0
14450             && app.curReceiver == null && app.services.size() == 0) {
14451             Slog.i(
14452                 TAG, "Exiting empty application process "
14453                     + app.processName + " ("
14454                     + (app.thread != null ? app.thread.asBinder() : null)
14455                     + ")\n");
14456             if (app.pid > 0 && app.pid != MY_PID) {
14457                 Process.killProcess(app.pid);

```

② 对于那些包含了 Activity 的进程，如果该 Activity 已经处于 stop 状态，并且是不可见的，同时已经在 HistoryRecord 对象中保存了该 Activity 的 state 数据，那么则可以停止该 Activity 所在的进程，相应的 state 数据保存在了 HistoryRecord 的 icle 变量中。这是一个 Bundle 类型，从而使得当下一次重新启动该 Activity 时把该数据传送过去。代码参照 AmS 源码第 14486 行，以下为截取片段。

```

14486         boolean canQuit = !app.persistent && app.curReceiver == null
14487             && app.services.size() == 0
14488             && app.persistentActivities == 0;

```

以上两步仅在系统不支持 OOM 模块的情况下使用。无论系统是否支持 OOM，执行完以上操作后，会进行最后一步内存回收工作。其逻辑是判断 mLRUActivities 列表中保存的已经运行 Activity 的数量是否超过了系统允许的最大值 MAX_ACTIVITIES (20)，如果超过了，则要强行关闭一些 Activity。这里的强行关闭也是有条件的，因此，可能出现死循环，即永远达不到这个条件，如以下代码所示：

```

14542         for ( i=0;
14543             i<mLRUActivities.size()
14544             && mLRUActivities.size() > curMaxActivities;
14545             i++) {
14546             final HistoryRecord r
14547                 = (HistoryRecord)mLRUActivities.get(i);
14548
14549             // We can finish this one if we have its icle saved and
14550             // it is not persistent.
14551             if ((r.haveState || !r.stateNotNeeded) && !r.visible
14552                 && r.stopped && !r.persistent && !r.finishing) {
14553                 final int origSize = mLRUActivities.size();
14554                 destroyActivityLocked(r, true);
14555
14556                 // This will remove it from the LRU list, so keep
14557                 // our index at the same value. Note that this check to
14558                 // see if the size changes is just paranoia -- if
14559                 // something unexpected happens, we don't want to end up
14560                 // in an infinite loop.
14561                 if (origSize > mLRUActivities.size()) {
14562                     i--;
14563                 }
14564             }
14565         }

```

该代码中要求，被停止的 Activity 必须属于 persistent 类型。因此，假设系统中有大于 20 个 persistent 类型的 Activity，而且它们的启动顺序是 1 启动 2，2 启动 3，依次类推，19 启动 20，20 启动后面的 21、22 等，在这种情况下，以上代码将会出现死循环而不能退出。当然如果没有这么多的 persistent 类型的 Activity，这里会 destroy 掉相应的 Activity 对象。

5. 客户进程内存回收

在 AmS 中调用 scheduleAppGcLocked(ProcessRecord app)方法时，就会通知指定的 app 对象进行内存回收，以下对这个过程进行描述。

该函数首先要检测指定 app 上一次进行 gc 的时间，并和当前时间对比。如果时间间隔还没有超过最小间隔，则仅仅把指定的 app 放入 mProcessesToGc 列表中。否则，调用同名函数 scheduleAppGcLocked(无参数)从 mProcessesToGc 列表中取出下一个 app，并发送一个 1 分钟延迟的 GC_BACKGROUND_PROCESSES_MSG 消息。之所以选择 1 分钟延迟，是为了减少客户进程进行 gc 的频率，为什么是 1 分钟而不是 2 分钟或者 30 秒呢？这或许是一个经验值。

处理 GC_BACKGROUND_PROCESSES_MSG 消息的函数是 AmS 中的 performAppGcsIfAppropriate()函数，其内部首先会判断当前是否可以“适合(Appropriate)”进行内存回收，如果可以则调用 performAppGcsLocked(无参数)，否则，再次调用 scheduleAppGcLocked(无参数)发送一个异步消息。

而是否“适合”则是通过调用 canGcNowLocked()方法进行判断的，其代码如下：

```
14103 private final boolean canGcNowLocked() {
14104     return mParallelBroadcasts.size() == 0
14105         && mOrderedBroadcasts.size() == 0
14106         && (mSleeping || (mResumedActivity != null &&
14107             mResumedActivity.idle));
14108 }
```

以上代码包含以下条件。

- 当前没有 Broadcast 在执行，包括 mParallelBradcasts 和 mOrderedBroadcasts。
- 当前处于 mSleeping 状态或者有正在执行的 Activity 但却处于 idle 状态。

以上条件的语义是系统当前必须处于空闲状态，有读者可能问了：“为什么条件中有 Broadcast 但却没有 Service 或者 Receiver 对象呢？”原因是，无论是 Service 还是 Receiver，都是在相应的服务进程中执行代码的，与 AmS 无关，而 Broadcast 则需要 AmS 去派发的，而当派发完毕后，mParallelBroadcasts 或者 mOrderedBroadcasts 列表的内容会被清除。

在 performAppGcLocked(无参数)函数内部，使用 while 循环，从 mProcessesToGc 列表中逐个取出每个需要进行 gc 的 ProcessRecord 对象，并针对该对象调用 performAppGcLocked(app)方法，如以下代码所示：

```

14120         while (mProcessesToGc.size() > 0) {
14121             ProcessRecord proc = mProcessesToGc.remove(0);
14122             if (proc.curRawAdj > VISIBLE_APP_ADJ || proc.reportLowMemory) {
14123                 if ((proc.lastRequestedGc+GC_MIN_INTERVAL)
14124                     <= SystemClock.uptimeMillis()) {
14125                     // To avoid spamming the system, we will GC processes on
14126                     // at a time, waiting a few seconds between each.
14127                     performAppGcLocked(proc);
14128                     scheduleAppGcsLocked();
14129                     return;

```

这里需要注意该 while 循环的内部处理。调用完 performAppGcLocked(proc)方法后，再调用 scheduleAppGcsLocked()，然后使用 return 返回。也就是说，每次仅针对一个 app 进行内存回收，可能 mProcessesToGc 列表中有多个 app 都满足回收内存的条件，但是也只能等到 1 分钟以后才能再次被调度到。

performAppGcLocked(proc)方法的内部流程如下，它会判断当前 app 的 reportLowMemory 标识是否为 true。如果是则调用 app.thread.scheduleLowMemory()函数处理因为内存低而导致的内存释放，否则调用 app.thread.processInBackground()函数处理背景模式内存释放。是在 AmS 通知客户进程启动指定的 Activity 时，如果客户进程报告启动失败，AmS 就会认为客户进程可能出现内存问题，从而将其 reportLowMemory 标识置为 true。

下面分别介绍客户进程中的 scheduleLowMemory()函数和 processInBackground()函数到底都做了哪些与内存回收相关的工作。首先来看前者，客户进程中凡是以 schedule 开始的函数，其内部都是发送一个异步消息并立即返回，即所谓的异步处理，该消息对应的处理函数为 handleLowMemory()，其内部流程如附图 4 所示，此处截取主要片段如图 10-23 所示

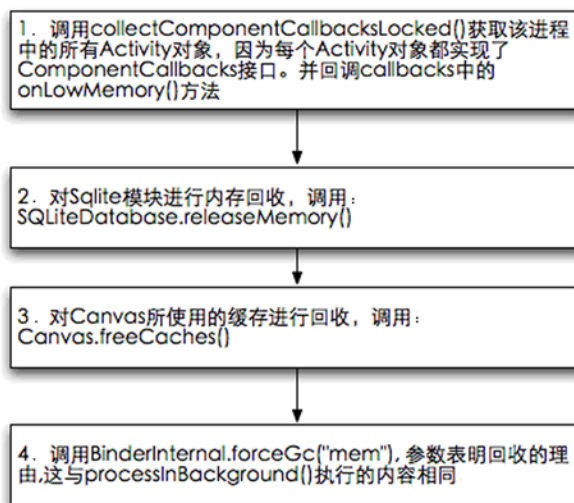


图 10-23 ActivityThread 中 handleLowMemory()内部流程

① 回调该进程中所包含的所有 Component 对象中的 `onLowMemory()` 方法。Component 包括四种，分别为 Activity、Service、Provider、Application，这四个对象分别在 ActivityThread 中的 `mActivities`、`mServices`、`mProviderMap`、`mAllApplications` 容器中保存。

② 调用 `SQLiteDatabase` 的静态方法 `releaseMemory()` 释放该进程中 SQLite 模块所占用的内存。尽管该方法是一个 native 方法，但从这种调用方式上可以猜出，SQLite 的内部肯定为每一个调用者进程分配了一块独立的内存空间缓存查询结果，而 `releaseMemory()` 方法则会判断调用者进程 id，然后根据此值释放相关的内存。

③ 调用 `Canvas` 的静态方法 `freeCaches()` 释放该应用中所包含的 `Canvas` 对象。注意这里 `Caches` 用了复数而不是 `Cache`，再次提示大家，每个应用中会包含多个 `Canvas` 对象。

④ 调用 `BinderInternal.forceGc("mem")` 方法释放该进程中的 `Binder` 对象。

以上四步中三步都和 native 的方法相关，而且都是静态方法，这也再次说明，native 所申请的系统内存无法从 Java 环境中清除，这就是为什么 native 方法常常引起应用程序内存泄露的原因，尤其是按照非对称方式调用 native 方法时。所谓的“非对称”是指，native 一般会提供一对函数，一个包括了申请内存的操作，另一个包括了释放该内存的操作，而 Java 程序仅仅调用了前者而忘记调用后者，那么即使强制关闭该 Java 程序，也依然会出现系统内存泄露的情况。

上面介绍了 `scheduleLowMemory()` 的处理过程，下面继续介绍 `processInBackground()` 方法，该方法的处理过程十分简单。当然，首先依然是发送一个异步消息，而该消息会执行到 `Idler` 对象中的 `doGcIfNeeded()` 方法，而该方法仅仅是调用了 `BinderInternal.forceGc("bg")` 函数而已，即与上面的第四步的执行内容相同。

为什么 `processInBackground()` 仅仅释放 `Binder` 相关的内存呢？原因是此时系统未出现内存低的情况，因此，只需要释放影响 CPU 运行的部分即可。在第 10.1 节中的关于系统效率中曾说过，应用程序占用 CPU 的部分由三个部分组成，其中一项就是 `Binder` 对象。

10.3 对 AmS 中数据对象的理解

AmS 中定义了各种变量，用于表示各种数据对象，这些对象包括“进程”、“任务”、“运行片段(Component)”等，这些变量如表 10-7 所示。

表 10-7 AmS 中定义的数据变量

变量名称	含 义
MAX_ACTIVITY	允许最大的后台 Activity 的数目，为 20
mHistory	所有处于非 <code>finising</code> 状态的 <code>HistoryRecord</code> 列表，按照启动顺序排列
MAX_RECENT_TASK	<code>mRecentTask</code> 列表中允许存储的最大值为 20
mRecentTask	最近启动过的 Task 列表，按照最后活动状态时刻排列，每个 <code>TaskRecord</code> 内部都包含 <code>lastActiveTime</code> ，表示该 Task 最后处于活动状态的时刻

(续表)

变量名称	含 义
MAX_HIDDEN_APP	允许处于 hidden 状态的进程数目的最大值, 超过这个值后, AmS 会杀掉优先级低的进程, 为 15
mLRUActivites	最近启动过的 HistoryRecord 列表。与 mHistory 的区别在于, mLRUActivities 仅仅记录了不同 Activity 的最后启动时刻, 而 mHistory 内部包含了 Task 的信息, 并且可能包含同一个 Activity 的多个实例。LRU 的含义是 Lastest Recent Used
mLruProcess	最近启动过的进程列表, 不包含 persistent 类型的进程
mProcessNames	所有运行的进程列表, 按照名称进行区分
mPidSelfLocked	与 mProcessNames 保存的进程对象相同, 但是按照 pid 进行区分
mServices	系统中已经启动的 Service 列表, 按照 Service 的名称区分
mServicesByIntent	同 mService, 但是按照启动该 Service 的 Intent 区分
mServiceConnections	与当前所有 Service 建立了连接关系的客户端列表
mProviderByName	系统中已经启动的 Provider 列表, 按照名称区分
mProviderByClass	同 mProviderByName, 按照 Provider 的实现类名称区分

10.3.1 常见的对象操作

在表 10-7 的基础上, 可以实现以下几种常见的操作。

- 如何获取系统中运行的类型为 persistent 的进程列表?

可以遍历 mProcessNames 列表, 并且判断每一个 ProcessRecrod 中的 persistent 变量是否为 true。

- 如何获取最近的 Task 列表?

直接遍历 mRecentTask 列表即可, 长按“Home”键时出现的最近六个任务列表正是从该列表中获取的。

- 当前 Task 中都包含哪些 Activity 对象?

首先通过 mCurTask 变量获得当前 Task 的 id 值, 然后遍历 mHistory 中的所有 HistoryRecord 对象, 判断其 taskId 变量是否等于 mCurTask 的 id 值。

- 系统中已经运行了哪些进程?

直接遍历 mProcessNames 或者 mPidSelfLocked 列表即可。

- 一个进程中都包含有哪些正在运行的 Activity 对象?

遍历 mHistory 列表, 查找每一个 HistoryRecord 的 app 变量 (这是一个 ProcessRecord 对象), 然后再判断该 app 对象的 pid 值是否为指定的 pid 值。

- 如何查看与系统中的 Service 对象有连接的客户端?

直接遍历 mServiceConnections 列表即可。如果要判断指定 Service 都包含哪些客户端, 则可以先从 mServices 中按照 Service 的名称找到目标 ServiceRecord 对象, 然后遍历 mServiceConnections 列表, 获



得 ServiceConnection 对象，再获得该对象内部的 AppBindRecord 对象，最后判断其包含的 ServiceRecord 对象是否是指定的 ServiceRecord 对象即可。

- 查看指定进程中都有哪些 Service 或者 Provider 对象在运行？

思路是遍历 mService 和 mProviderByName，然后分别从其对应的数据对象判断是否属于指定的进程 id。

10.3.2 理解 Activity

Activity 本身只是一段程序代码而已，它所执行的内容没有任何的系统调用，客户端程序总是从 ActivityThread 开始执行。这个类已经创建了客户进程，然后从 Activity 所对应的 Class 文件中装载进来程序代码，实际上就只是一个 Activity 类对象，然后继续执行该对象内部的各种代码，这些执行的代码默认不会再创建线程。

有的读者会问一个不是问题的问题：“Android 只允许一个 Activity 在运行，有没有办法让多个 Activity 同时运行，就像 Windows 操作系统中的同时开一个计算器程序，再开一个浏览器程序一样呢？”之所以说这个问题不是问题，原因在于，这个问题成立的条件如下：

- 每个 Activity 都应该对应一个应用程序。
- 每个 Activity 都应该包含一个交互窗口。
- 系统必须为每一个 Activity 的窗口分配显存，并且只有在该应用程序退出时才释放这段显存。

下面对这三个条件逐一进行解析。

- Activity 并不对应一个应用程序，ActivityThread 才对应一个应用程序，所以 Android 允许同时运行多个应用程序，实际上是允许多个 ActivityThread 同时运行。
- 默认的 Activity 实现中的确会添加一个窗口，但实际上可以修改 Activity 的默认实现，使其不添加任何窗口，而且 AmS 对这种修改不会在意，依然会正常运行。换句话说，AmS 会按照定义好的调度顺序来启动、关闭 Activity，至于 Activity 内部是添加窗口或不添加窗口也好，无所谓。
- 在 Android 中，尽管默认的 Activity 会添加一个窗口，但是当系统内存低的时候，后台 Activity 对应的显存会被释放掉，而且这是在用户没有感知的条件下进行的。

因此，以上问题可以这么来问：“Android 是多窗口系统吗？”即在一个屏幕上可以存在不同 Activity 对应的窗口，而且这些窗口可以层叠、移动。这个问题留在 WmS 源码分析一章中解答，不过结论是：Android 窗口系统是一种简单的单窗口系统。

10.3.3 Android 多进程吗？是同时在运行多个应用程序吗

Android 的确是多进程的，因为同时有多个应用程序进程存在。但不幸的是，由于 WmS 子系统的



设计特点,导致非前台应用程序没有机会获得消息,从而没有机会占用 CPU。并且 Android 的底层 Linux 未采用磁盘虚拟内存机制,程序只能使用物理内存作为最大内存,所以 AmS 中采用了自动杀死优先级较低进程的方法以达到释放内存的目的,而这会导致用户在多个(超过 20 个)应用程序中快速切换时就会出现速度变慢的现象。

这并不能说就是 Android 系统的缺点,因为毕竟 Android 是面向移动设备的,对于手机用户来讲,来回在超过 20 个程序间切换似乎是不常发生的事情。

另外,从技术角度来讲,AmS 的 WmS 调度机制也能用来定义“上网本”的概念。在过去的 2 年里,“上网本”被认为是消费电子行业的一个新产品,很多电脑公司都推出了所谓的上网本产品,这里先不说硬件的配置,而来说所使用的操作系统,在 iPad 诞生之前,流行的上网本基本上都是采用 Windows 操作系统。人们对上网本的认识似乎就是“价格低一点,速度慢一点,待机时间长一点”,而从功能的角度来讲,则完全等同于 PC。

如果说 iPad 也是上网本,则操作系统就应该被重新定义了,这里主要就是 AmS 和 WmS 的调度机制的差别。

首先,Windows 系统是完整的多进程、多窗口操作系统,而 iPad 或者 Android 只能算作多进程、单窗口系统。同时,为了配合单窗口设计,AmS 在应用程序调度上也做了一定的调整,使得虽然 Linux 底层是多进程,而 Framework 层却变成了单进程。

单窗口与多窗口系统设计的主要区别在于以下几点。

- 系统必须为每一个应用程序分配窗口,并且这些窗口应该常驻,并且只有在程序退出时才释放。
- 用户的按键等消息要同时被发送到所有的应用窗口,而不仅仅是所谓的“当前交互”窗口。
- 窗口系统应该能处理多个应用窗口之间的层叠、位移等变化,而不能把这个任务交给一个应用窗口自己去处理。
- 窗口系统在将多个窗口绘制到屏幕上时,也就是 Android 中的 SurfaceFlinger 模块,要同时所有的窗口中进行运算,而不仅仅是把当前的窗口绘制到屏幕。

当然,要实现以上四点,对 CPU 来讲也有一定难度,尤其是当所有的窗口都不处于全屏的时候。这就是为什么在 Windows 操作系统中的窗口有一个“最大化”的按钮的原因,因为在“最大化”模式下,用户按键消息的派发及屏幕的绘制都能够简化一些,从而有助于系统的运行效率。当然这只是技术上的考虑,从用户的角度来讲,或许最大化也是他们所想要的功能,而在苹果电脑系统中,则没有最大化的按钮。

以上四点中,Android 仅仅满足第四点,所以说它不是真正的多窗口系统。而 iPad 或者 Android PAD 只是 30%的游戏 + 30%的阅读 + 30%的工具 + 10%的生活乐趣。

10.4 ActivityGroup 的内部机制

分析完 AmS 内部机制后,读者们已经明白,系统一次只允许一个 Activity 处于运行状态,要让另外一个 Activity 处于运行状态必须先暂停当前正在运行的 Activity。而在应用程序设计时,Framework



提供了一个 ActivityGroup 类，它允许把一个 Activity 嵌入到另一个 Activity 中运行，常见的基于该类的实现包括 TabActivity，这到底是怎么回事呢？难道 ActivityGroup 类具备同时使多个 Activity 处于运行状态的能力吗？

本节就以 TabActivity 类的实现为例，说明 ActivityGroup 的内部机制。

10.4.1 TabActivity 使用时的类关系结构

在使用 TabActivity 设计应用程序时，其内部相关的类关系如图 10-24 所示。

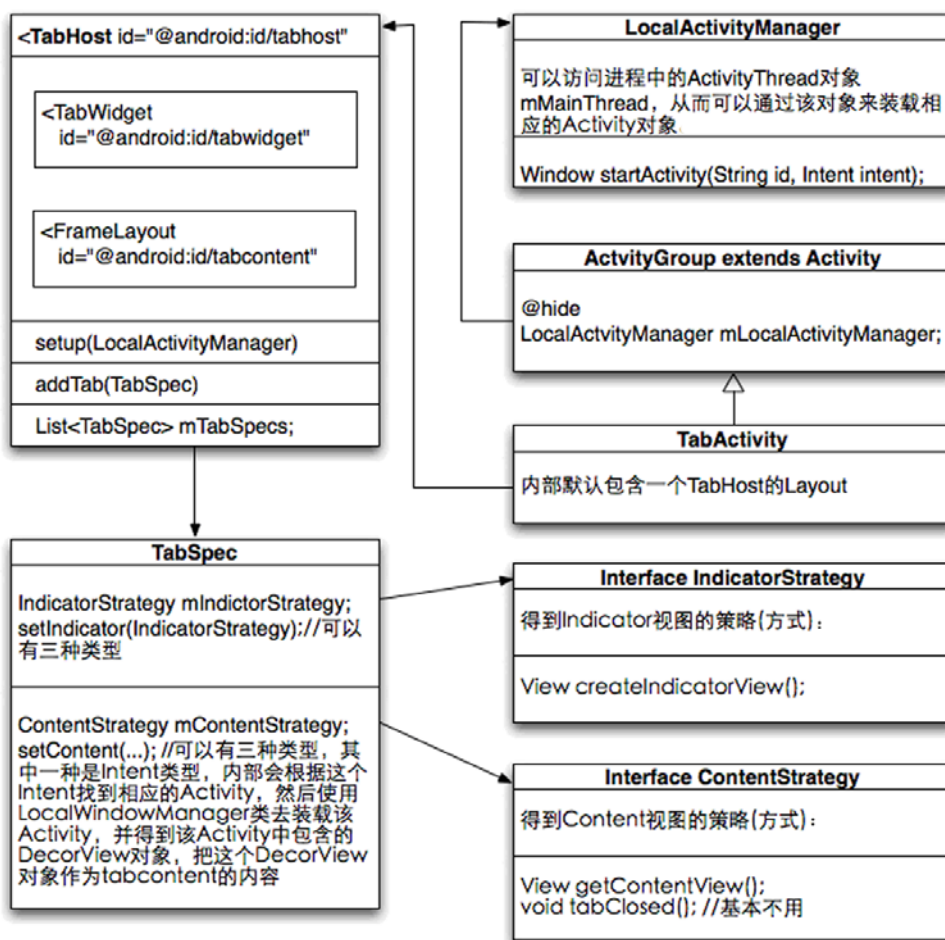


图 10-24 TabActivity 相关类的关系

首先 TabActivity 的祖先类也是 Activity 类，因此，从 AmS 的角度来看，TabActivity 与普通的 Activity 没有什么区别，其生命期包括标准的 start、stop、resume、destroy 等，而且系统中只允许同时运行一个

TabActivity。

所不同的是，TabActivity 基于 ActivityGroup 类，该类的父类是 Activity 类，但 ActivityGroup 内部有一个重要成员变量，其类型为 LocalActivityManager，该类的最大特点在于它可以访问应用进程的主类，即 ActivityThread 类。前面分析过，AmS 要启动某个 Activity 或者暂停某个 Activity 都是通过 ActivityThread 类执行的，而 LocalActivityManager 能够访问 ActivityThread 类就意味着可以通过它来装载不同的 Activity，并且控制 Activity 的不同的状态。

为什么 LocalActivityManager 可以访问主类 ActivityThread，而普通的应用程序无法访问主类呢？因为在编译输出 Android SDK 时，ActivityThread 类是被隐藏的，如以下代码所示：

```
115  * {@hide}
116  */
117  public final class ActivityThread {
```

所以，应用程序无法通过 SDK 来引用 ActivityThread 类。那又有读者问了，如果不使用 SDK 编译，而直接使用源码编译是否可用呢？真的很抱歉，也是不可以，因为客户进程的主类变量的访问权限被设置为包内访问，如以下代码所示，未加权限限定符即指包内访问。

```
157  static final ThreadLocal<ActivityThread> sThreadLocal
```

而 LocalActivityManager 正是和 ActivityThread 在同一个包内，这就是为什么它能够访问主类对象的原因。

TabActivity 类的默认布局文件是 com.android.internal.R.layout.tab_content，该布局的内容是一个 TabHost 视图，其 id 值为系统 id 值 android:id/tabhost。TabHost 中包含了两个子视图，一个是 TabWidget，其 id 值必须是 android:id/tabwidget，这就是大家经常看到的 Tab 页的效果；另一个是 FrameLayout 视图，其 id 值必须是 android:id/tabcontent，TabActivity 所包含内置 Activity 对应的窗口就是被添加到该 FrameLayout 中显示的。

tab_content 仅仅是 TabActivity 的默认布局，应用程序可以使用自定义的布局代替这个布局，但自定义的布局文件中必须包含 tab_content 中所声明的最少三个视图，并且其 id 值必须是系统的 id 值，分别为 TabHost、TabWidget、TabContent。

在 TabHost 的视图类中，包含了三个重要的函数或变量，分别如下。

- setup(LocalActivityManager)：该方法用于给 TabHost 内部设置一个 LocalActivityManager 对象。读者可能会觉得奇怪，TabHost 中为什么还需要这个 Manager 对象呢？这个对象的确不是必需的，假如你所实现的 Tab 页视图不需要嵌入一个 Activity 视图，那么就不需要设置这个对象，否则必须告诉 TabHost 这个 Manager 对象，要不然 TabHost 是无法获得指定 Activity 的内部视图的。
- addTab(TabSpec)：该方法用于向 TabHost 中的 TabWidget 视图添加一个 Tab 页。表面上看，既然是添加一个 Tab 页，那么 TabSpec 类应该是一个视图类才对，而事实上 TabSpec 是一个功能类，其本身并不是一个视图，但是它却可以提供产生视图的方法。



- `List<TabSpec> mTabSpecs`: 这是 `TabHost` 中所包含的全部 `Tab` 页列表，每一项都是一个 `TabSpec` 对象。

下面，再来介绍 `TabSpec` 是如何产生 `Tab` 页视图的。

`TabSpec` 类中有两个重要变量，其类型分别是 `IndicatorStrategy` 和 `ContentStrategy`。这两个子类的名称有点意思，`Indicator` 的意思是“指示”，它指示用户该 `Tab` 页的意义，`Content` 即为单击某个 `Tab` 页后对应的“内容”，而 `Strategy` 是策略的意思，因此这两个子类的作用分别是“产生 `Tab` 页的策略”和“产生内容的策略”。同时，这两个子类仅仅是一个 `interface`，为什么只是一个接口呢？因为既然是策略就意味着可以有不同的策略，在 `TabHost` 中也的确实现了不同的策略，其中 `Tab` 页策略包括以下几项。

- `LableAndIconIndicatorStrategy`: 可以产生一个带图标和标签文字的标签页。
- `LableIndicatorStrategy`: 仅仅有标签文字的标签页。
- `ViewIndicatorStrategy`: 可以使用自定义 `View` 作为标签页视图。

内容策略包括以下。

- `FactoryContentStrategy`: 使用回调方式允许应用程序动态创建内容视图。
- `IntentContentStrategy`: 嵌入 `Activity` 视图作为内容视图，这正是 `TabActivity` 所使用的。
- `ViewIdContentStrategy`: 使用一个静态的布局作为内容视图。

看了这么多的策略后，大家可能会觉得奇怪：“新建一个 `TabSpec` 后，如何指定其标签和内容使用何种策略呢？”事实上，当新建一个 `TabSpec` 对象后，`TabSpec` 提供了三个设置标签策略的函数 `setIndicator(xxx)`，其中不同参数对应不同的标签策略；同时也提供了三个设置内容策略的函数 `setContent(xxx)`，不同的参数对应不同的内容策略。其中 `setContent(Intent intent)` 函数正是使用嵌入 `Activity` 方式，参数 `Intent` 用来匹配相应的 `Activity`。

10.4.2 LocalActivityManager 的内部机制

`LocalActivityManager` 内部机制的核心在于，它使用了主线程对象 `mActivityThread` 来装载指定的 `Activity`。注意，这里是装载，而不是启动，这点很重要。

所谓的启动，一般是指会创建一个进程(如果所在应用进程还不存在)运行该 `Activity`，而装载仅仅是指把该 `Activity` 作为一个普通类进行加载，并创建一个该类的对象而已，而该类的任何函数都没有被运行。

`LocalActivityManager` 提供了一个重要方法 `startActivity()`，该方法正是利用主线程 `mActivityThread` 去装载指定的 `Activity`，其执行过程如图 10-25 所示。



图 10-25 LocalActivityManager 类中 startActivity() 的执行流程

① Manager 对象必须已经被初始化，初始化的工作是在 `dispatchCreate()` 方法中首先被完成的，而这又是在 `ActivityGroup` 类中的 `onCreate()` 中被调用的。也就是说，`LocalActivityManager` 的 `startActivity()` 方法必须在所在的 `Activity` 的 `onCreate()` 方法执行完并被调用。

② 判断目标 `Activity` 是否包含在该 `Manager` 中。`Manager` 中使用两个列表变量保存已经装载过的 `Activity` 对象，分别是 `mActivities` 和 `mActivityArray`。前者是一个 `HashMap` 类型，每一个 `LocalActivityRecord` 按照一个字符串对应；后者是一个 `ArrayList` 列表。

③ 判断装载的 `Activity` 对象是否有正处于 `resume` 状态的，如果有，则要先暂停，事实上可以完全不用暂停，暂停仅仅是 `Manager` 希望完全按照 `Activity` 对象本身的执行顺序调用它，从而使得看上去更像是一个标准的子 `Activity` 启动方式。而暂停则是通过调用 `moveToState()` 完成的。

④ 如果目标 `Activity` 已经被装载到了当前 `Manager` 中，下面就需要判断是直接使用该 `Activity` 的当前窗口呢，还是需要先销毁该 `Activity`，并重新调用其 `onCreate()`？注意，这里所说的销毁仅仅是指把 `Activity` 变成“销毁”的状态而已，并不是说销毁该 `Activity` 对象。而判断的规则有点类似于 `AmS` 中根据 `Activity` 的 `flag` 执行不同的操作，其中包括是否先调用目标 `Activity` 的 `onNewIntent()`，还包括是

否是 CLEAR_TOP 模式。一般作为 TabActivity 的嵌入式 Activity 都不会是 CLEAR_TOP 模式，否则，如果多个 Tab 页使用同一个 Activity 对象将导致所显示的内容完全相同。

- ⑤ 调用 moveToState()改变指定 Activity 到 resume 状态。
- ⑥ 返回 Activity 所对应的 Window 窗口。

从以上步骤可以看出，装载 Activity 对象的过程对 AmS 来讲是完全不可见的，因为这是装载而不是启动，因此看似 TabActivity 同时运行了多个 Activity，而实际上仅仅是运行了 ActivityGroup 一个 Activity。那些嵌入的 Activity 仅仅是贡献了自己所包含的 Window 窗口而已，TabActivity 正是把这些 Window 窗口的 DecorView 作为 tabcontent 的子视图而已。

下面对 moveToState(LocalActivityRecord r, int desireState)函数的过程进行说明，参数 r 代表目标 Activity 对象，desireState 代表期望把目标 Activity 改变成哪种状态。

moveToState()函数内部首先判断 r.curState 是否是 RESTORED 或者 DESTROY 状态，如果是则直接返回。因为 RESTORED 代表刚刚创建了目标 Activity 对象，还没有执行 onCreate()方法，所以不能改变状态；DESTROY 代表已经销毁，也不能改变状态。

接着，判断 r.curState 是否是 INITIALIZE 状态，这种情况只有在第一次调用 startActivity()装载目标 Activity 对象时才会执行到，其内部主要包括调用 startActivityNow()和 performResumeActivity()将目标 Activity 改变到 STARTED 或者 RESUMED 状态。

由于 Activity 当前状态不同，要想达到不同的期望状态自然需要经过不同的步骤。moveToState()函数内部正是使用 switch 语句先判断当前处于什么状态，然后再在 case 里面使用 if...else 语句判断期望的状态，最后再调用不同的函数。其状态和调用关系如表 10-8 所示，该表中调用的函数名称使用了简写，比如 performRestartActivity 简写为 Restart。

表 10-8 Activity 在不同状态中转换时需执行的操作

目标状态 当前状态	CREATED	STARTED	RESUMED
CREATED		Restart ();	Restart(); Resume();
STARTED	Stop();		Resume();
RESUMED	Pause(); Stop();	Pause();	

从以上的步骤可以看出，startActivity()的内部执行逻辑有点像 AmS 中根据当前 Activity 状态调用不同方法。这两者就像《西游记》中的小雷音寺和大雷音寺，两者的本质区别在于 LocalActivityManager 仅仅是为了获取 Activity 对应的 Window 对象，中间的状态切换仅仅是为了保证 Activity 本身的执行过程，从而保证 Window 对象的视图内容有一个正确的呈现。



10.4.3 ActivityGroup 内部的 Activity 生命期控制

前面分析了 LocalActivityManager 内部的执行原理，接下来分析一个有意思的问题：“在 TabActivity 的多个 Tab 页切换时，内嵌的 Activity 对象会在 onPause()和 onResume()之间切换吗？”

从上面的分析可知，Manager 装载 Activity 的目的仅仅是为了获取其所包含的 Window 对象，而一旦获取后，则似乎不需要再纠缠于 Activity 本身的生命期状态变换操作。其实笔者也是这么认为的，可以尝试屏蔽以下代码，该段代码的作用是在装载下一个 Activity 之前先暂停当前的 Activity 对象。

```
287 //         if (old != null && old != r && mCurState == RESUMED) {
288 //             moveToState(old, STARTED);
289 //         }
```

屏蔽后，运行结果丝毫不受影响，原因很简单，这里做暂停的目的仅仅为了保持 Activity 原有的生命期过程，从而可以保持原有 Activity 释放相关资源的行为。比如当 Activity A 以启动的方式运行时，如果另一个 Activity B 要启动，则会先暂停 A。在一般的程序设计中，暂停会回调 onPause()操作，如果该 Activity 使用了大量的内存或者其他资源，在 onPause()函数中程序员可能会尝试释放这些资源以提高系统效率，这就是为什么在 LocalActivityManager 中也保持了这种流程的原因。当然，如果你不释放，也不会发生什么逻辑错误。

而在以上代码的 moveToState()操作中调用了 mActivityThread 的 onPause()或者 onStop()操作，这就是为什么在 Tab 页切换时，对应的 Activity 也会执行 onPause()或者 onStop()的原因。这完全与 AmS 无关，因此不要因为这个而产生 TabActivity 同时运行了两个 Activity (TabActivity 本身和嵌入的 Activity) 的错觉。

另外还有一个有意思的问题，请思考下面的操作过程。

打开一个 TabActivity，比如“联系人”程序，在上面的四个 Tab 页上都点一次，然后再按“Home”键回到桌面，然后再从联系人图标中进入该程序。请思考此时 TabActivity 内嵌的 Activity 会发生生命期状态改变吗？

首先 TabActivity 当然会从 stop 状态转变为 start 状态，并先后调用 onStart()和 onPause()。因为它是一个标准的 Activity，TabActivity 的父类是 ActivityGroup，而在该类中，相应的 onXXX()方法内部都增加了 mLocalActivityManager.dispatchXXX()代码，比如：

```
55 @Override
56 protected void onResume() {
57     super.onResume();
58     mLocalActivityManager.dispatchResume();
59 }
```

而在 LocalActivityManager 中，dispatchXXX()则会把相应的 action 再 dispatch 到所包含的所有嵌入式 Activity 对象中。所以，以上问题的答案是内嵌的 Activity 生命期会从 stop 状态转换到 resume 状态。仔细查看 ActivityGroup 的 onXXX()函数发现，唯独没有 onStart()方法，其原因是在 LocalActivityManager 的 moveToState()方法中可以直接把子 Activity 的状态从 stop 改变到 resume，所以，此处可以省略对 onStart()的重载。

