第二章 Android 基础知识

2.1 Android 是什么?

（Simon 翻译自 http://developer。android。com/guide/basics/what-is-android.html）

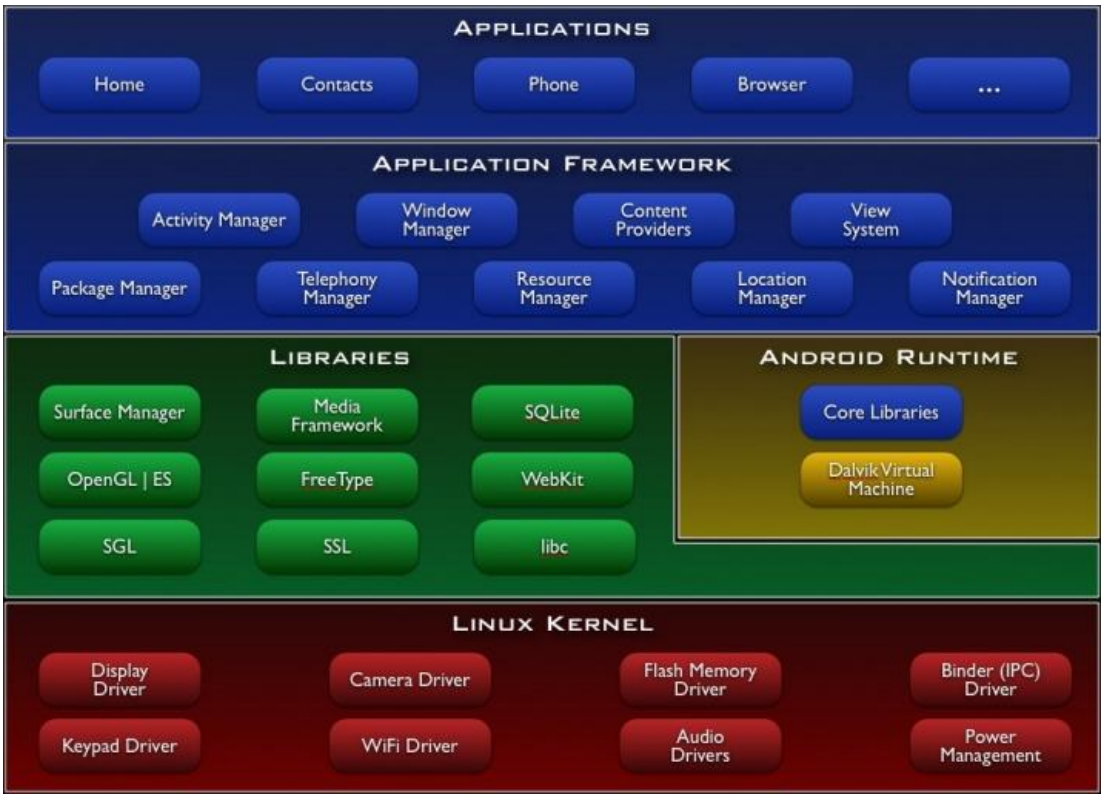
Android 是一个面向移动设备的软件堆层（software stack），包含了一个操作系统，中间件和关键的应用程序。Android SDK 提供了必要的工具和 API，你可以在这些的基础上使用 java 编程语言开发 Android 平台上的应用程序。

特性

- 应用程序框架　支持组件的复用和更换
- Dalvik 虚拟机　专门为移动设备进行过优化
- 集成的浏览器　基于开源的 WebKit 引擎
- 优化的图形机制　自定义的 2D 图形库，基于 OpenGL ES 1。0 规范的 3D图形实现（本项硬件加速器可选
- SQLite　轻量级的数据库，支持结构化数据的存储
- 媒体支持　面向常见的音频、视频以及静态图形档案格式(MPEG4， H。264，MP3，AAC，AMR，JPG，PNG，GIF)
- GSM 技术　(依赖硬件支持)
- Bluetooth，EDGE，3G，和 WiFi　(依赖硬件支持)
- Camera，GPS，compass，和 accelerometer　(依赖硬件支持)
- 丰富的开发环境　包含一套硬件仿真器，一些用于程序调试、内存和性能剖析的工具，以及支持 Eclipse 集成开发环境的插件（ADT）。

Android 框架

下图表显示了 Android 操作系统的主要组件。下面会对每个部分进行更详细的描述。



应用

Android 将预装一组核心应用程序，包括 email 客户端、短信服务、日历日程、地图服务、浏览器、联系人和其他应用程序。所有应用程序都是 Java 编程语言编写的。

应用框架

通过提供一个开放的开发平台，开发者使用 Android 可以开发出极为丰富且新颖的应用程序。开发者可以自由地利用设备硬件优势、访问位置信息、运行后台服务、设置闹钟、向状态栏添加通知等等。

开发者拥有对核心应用程序所使用的相同框架 API 的完全访问权力。应用程序框架的设计旨在简化组件的复用；所有应用程序都可以发布其能力。任何应用程序都可以发布自己的功能，然后其他任何应用程序都可以使用这些功能(需要符合框架强制要求的安全约束)。这一相同的机制允许用户替换组件。

所有应用都是一组系统和服务，一般包含：

- 一组丰富和可扩展的，可以用来构建应用程序的视图组件，含有 lists， grids ， text boxes，buttons，甚至内嵌网络浏览器
- Content Providers(内容提供器)使一个应用程序可以访问另外一个应用程序的数据(如联系人），或者应用程序内部共享自有数据
- Resource Manager（资源管理器），提供对本地化字符串、图形和布局文件等非代码资源的访问通道

- Notification Manager（通知管理器），使所有的应用程序在状态栏显示自定义的警告通知
- Activity Manager（活动管理器）负责管理应用程序的生命周期，提供通用导航回退

更多应用程序细节，请参考"记事本教程"。

库

Android 包含一组各种各样的 Android 系统组件都在使用的 C/C++库。这些功能通过 Android 应用程序框架提供给开发者。下面列举一些核心库：

- 系统 C 库——基于 BSD 的标准 C 系统库（libc）实现，移植到了 Linux 嵌入式设备上
- 媒体库——基于 PacketVideo 的 OpenCORE；媒体库支持很多流行音频和视频格式、静态图形文件（包括 MPEG4，H。264，MP3，AAC，AMR，JPG 和 PNG）的播放和录制
- 表面管理器——管理对显示子系统访问，无缝组合多个应用程序的二维和三维图形层
- LibWebCore——是流行的浏览器引擎，可以支持 Android 浏览器和嵌入式的网页视图
- SGL——底层的 2D 图形引擎
- 3D 库——基于 OpenGL ES 1。0 API 的实现；该类库使用硬件 3D 加速器（有相应硬件时）或者内置的、高度优化的 3D 软件加速机制。
- FreeType——支持位图和矢量字体渲染
- SQLite——面向所有应用的，强大且轻量级的关系型数据库引擎

Android 运行时刻(Android Runtime)

Android 的核心类库提供 Java 类库所提供的绝大部分功能。

每个 Android 应用程序都通过 Dalvik 虚拟机在自己的进程中运行。Dalvik 被设计来使一台设备有效地运行多个虚拟机。Dalvik 虚拟机执行的是 Dalvik 格式的可执行文件（。dex）——该格式经过优化，以降低内存耗用到最低。虚拟机是基于寄存器，运行 Java 编译器编译的类，这些类通过 Android 内置的“dx”工具编译成了。dex 格式。

在一些底层功能，比如线程和低内存管理方面，Dalvik 虚拟机是依赖 Linux 内核的。

Linux 内核

Android 在安全、内存管理、进程管理、网络组、驱动模型等核心系统服务上依赖 Linux 2.6 版。内核部分还相当于一个介于硬件层和系统中其他软件组之间的一个抽象层。

2.2 Android 应用程序基础

（Simon 翻译自 http://developer.android.com/guide/topics/fundamentals.html）

Android 应用程序是用 java 语言写的，通过 aapt 工具把编译好的 java 代码和应用程序所需要的所有数据、资源文件打包成 Android 包，即后缀为.apk 的压缩文件，这个文件是发布应用程序和在移动设备上安装应用程序的媒介，是用户下载到他们设备上的文件。一个.apk 文件中的所有代码同属于一个应用程序。

从很多方面来说，每个 android 应用程序都运行在自己的空间里：

- 默认每个应用程序在自己的 Linux 进程中运行，当应用程序中的任何代码需要执行时 android 就启动一个的进程，当不再需要或系统资源被其他应用程序请求时 android 就关闭这个进程。
- 每个进程都有其专属的 Java 虚拟机（VM），所以应用程序代码运行时与其他的应用程序是彼此隔离的。
- 默认的，每个应用被赋予一个唯一的 Linux 用户 ID，由于权限设置的原因，一个应用程序的文件只有本用户（应用程序本身）可见——当然，也有把他们导出给其他应用程序的机制。

可以为两个应用程序安排使用同一个用户 ID，这种情况下他们彼此之间是可以看见对方的文件。为了节约系统资源，拥有相同 ID 的应用也能被安排运行在一个相同的 Linux 进程中，共享同一个虚拟机。

应用程序组件(Application Components)

Android 一个核心特点就是一个应用程序能使用另一个应用程序的元素（在提供元素的应用程序允许的情况下）。例如，如果你的应用程序想要显示一个滑动图片列表，另一个应用程序正巧开发了合适的滑动模块，并且同意共享，你就可以调用那个滑屏模块处理这些图片并显示出来，而不是自己再去开发一个。你的应用程序并没有包含或链接了提供元素的应用程序的代码，只是在需要的时候启动使用其他程序的部分功能。

为了实现这样的过程，系统必须在应用程序的任何部分被请求时启动这个程序的进程，实例化那部分 Java 对象。因此，和其他大多数系统不同的是，android 应用程序没有一个单独的程序入口（例如：没有 main 函数）。而是包含运行所需的必要组件，使得系统可以实例化对象。android 中有四种组件：

活动(Activity)

一个 Activity 表示用户可视化界面，用户可以在上面进行一些操作。例如，活动会显示一个用户可选的菜单项的列表，或是显示带有标题的图片。一个文本信息应用程序可能有一个活动来显示将要发送信息的联系人对象，一个活动用于显示撰写信息文本给

选定的联系人，其它的活动用于查看旧的消息或者显示设置的界面。虽然它们作为一个整体的用户界面进行协同工作，但是每一个活动都是相对独立的。每一个活动都是活动基类（类 Activity）的一个子类实现。

一个 Android 应用可能由一个活动组成，或者像上面提到的文本信息应用程序一样包含了多个活动。活动是什么样的以及需要多少的活动，这些取决于你的应用程序是如何设计的。最典型的是将一个活动被标记为第一个，当应用被加载时显示给用户。从一个活动转到另一个活动是通过在当前活动来运行下一个活动实现的。

每个活动提供了一个用于绘制的默认窗口。通常窗口将占满整个屏幕，但是也有可能比屏幕小并且浮在另一个窗口的上面。一个活动可以使用多个窗口——例如，在活动中央显示一个需要用户回应的弹出对话框，或者当用户选择屏幕上一个特定项目时为用户显示一些重要信息的窗口。

窗口中的可见的内容是由一组继承自 View 基类的 view 组成的层次体系。每个 view 控制窗口中一块特定的矩形区域，父 view 包含并组织子 view 的布局。叶 view（层次底端的 view）绘制它们管理的矩形，并且负责响应用户在此区域的操作，因此 view 就是活动和用户交互的地方。例如，一个 view 显示一个小图片，当用户点击这个图片后开始一个操作。Android 有很多已经做好的 view 你可以选择使用，包括按钮，文本输入框，滚动条，菜单项，多选列表等等。

通过使用 Activity.setContentView()方法将一组 view 层放置到一个活动窗口中，content view 是 view 层中最顶端的那个 view。（参见 User Interface 的文档获取更多有关 view 和层次的信息。）

服务(Services)

服务没有可见的用户界面，但是可以在后台运行任意长的时间。例如，一个服务可以在用户转向其他工作后仍然在后台播放音乐，或者从网上下载数据，或者计算一些东西然后在需要的时候提供给活动。每个服务都继承自 Service 基类。

一个主要的例子就是从列表中播放音乐的媒体播放器。播放器程序可能会有一个或几个活动，这些活动可以让用户选择希望播放的音乐然后显示播放。但是音乐播放过程本身不会使用一个活动，因为用户希望在切出播放器界面做别的事情时音乐也能一直放下去。为了保持播放继续，播放器的活动可以启动一个在后台运行的服务。然后即使启动这个服务的活动退出，音乐播放服务也能继续运行。

你可以连接（connect）或者绑定（bind）到一个正在运行的服务（如果这个服务还没运行的话就启动它）。当连接到服务后，你可以通过服务暴露出来的接口和这个服务进行通信，对音乐播放服务来说，这个接口可能允许用户暂停，后退，停止，重新播放等操作。

像活动和其他组件一样，服务运行在这个应用程序进程的主线程中。因此为了不阻塞其他的组件或者用户界面，服务经常为那些耗时长的任务单独开一个线程(比如音乐播放)。

广播接收器(Broadcast receivers)

广播接收器只是接收广播并对广播信息做出作出反应，多数的广播是由系统代码发出的——-比如反应时区变化的通知，电量低的通知，照了一张照片的通知，或者用户修改了系统语言的通知。应用程序也可以自己定义广播，比如定义这样一个广播，让其他的应用程序知道某些数据已经下载完毕了可以使用了。

应用程序可以有任意数量的广播接收器来对他所关心的广播进行监听并作出反应。所有的广播接收器都继承自 BroadcastReceiver 基类。

广播接收器不显示在用户界面上，但是可以启动一个活动来对接收到的信息进行响应，或者可以使用 BroadcastReceiver 来警告用户。Notifications(通知)可以通过不同的方式引起用户的注意，比如使背景灯闪烁，使设备振动，播放声音等等。通常是在状态栏上显示一个不会消失的图标，用户可以打开这个图标查看通知。

内容提供者(Content providers)

内容提供者使程序中特定的数据可以被其他程序使用。这些数据可以存储在文件系统中，SQLite 数据库中，或者任何其他可以存数据的地方。内容提供者继承自 ContentProvider 基类，实现了一系列的使其他程序获取和存储其支持的数据格式的方法，但是应用程序不直接调用这些方法。而是使用一个 ContentResolver 对象，然后调用这个对象的方法。ContentResolver 可以和任何的内容提供者交流，它和提供者协作来管理所有涉及到的进程间的通信。

获取更多使用内容提供者的详细信息请参见另一篇文档——Content Providers。

无论何时请求都应该由一个特定的组件来处理，Android 将确认组件的应用程序进程是否处于运行状态，并在需要的时候启动它，以及组件的一个特定实例是否可得，并在需要的时候生成该实例。

激活组件：intent

当有一个来自于 content resolver 的请求指向内容提供者时，内容提供者被激活。其他的三个组件——活动，服务，广播接收器——是通过一个叫做 intent 的异步的消息来激活的，intent 持有异步消息的内容。对于活动和服务，它主要是为被请求的动作命名，然后指定需要操作的数据的 URI。例如，它可能携带让一个活动为用户展现一张图片或者让用户编辑文本的请求。对于广播接收器，intent 对象为将要广播的内容命名。例如，它可能会通知感兴趣的一方相机的按钮被按下了。

激活不同的组件需要使用不同的方法：

- 活动（或者新的任务）是通过传递一个 Intent 对象到 Context.startActivity()或者 Activity.startActivityForResult()来。被激活的活动可以通过 getIntent()方法来查看使它启动的原始 intent。Android 调用 Activity.onNewIntent()方法来传递之后的 intent。
  一个活动常常会启动下一个活动。如果前一个活动希望下一个启动的活动返回一个结果，那么它将调用 startActivityForResult()而不是 startActivity()。例如，如果一个老活动开启了一个让用户选择照片的新活动，它可能期待返回选中的照片。结果通过传递给

被调用的活动的 onActivityResult()方法的 Intent 对象返回。

- 服务是通过传递 Intent 对象到 Context.startService()来启动的（或者将新的命令指派给正在运行的 service），android 调用 service 的 onStart()方法，并且把 Intent 对象传递给他。

  类似的，一个 Intent 可以被传递到 Context.bindService()方法里来建立呼叫组件与被叫目标服务的实时连接，调用 service 的 onBind()方法来接收这个 Intent 对象。（如果这个 service 还没有运行，可以选择使用 bindservice()启动它）。例如，一个 activity 可以建立一个与前面提及的音乐播放服务的连接，这样它就能够提供控制播放的方式给用户(一个用户界面)。activity 将调用 bindService()来建立该连接，然后调用该 service 定义的方法来进行播放。

  后面讲到 Remote procedure calls，有更详细的绑定服务的信息。

- 初始化 broadcast 可以通过传递一个 Intent 对象给诸如 Context.sendBroadcast()、Context.sendOrderedBroadcast()和 Context.sendStickyBroadcast()方法。Android 通过调用它们的 onReceive()方法，将 intent 传递给所有感兴趣的 broadcast receiver。

想要了解更多的 intent 信息，参照另一篇文档，Intents and Intent Filters。

关闭组件（Shutting down components）

一个 content provider 仅仅在它响应从 ContentResolver 来的请求时处于活跃状态。一个 broadcast receiver 仅仅在它响应 broadcast 信息时处于活跃状态。所以没必要显示地关闭这些组件。

另一方面，Activity 提供用户界面。它们会与用户长时间地对话，而且可能在整个对话过程中，即使处于空闲状态，都保持活跃状态。同样的，service 也可能长时间保持运行状态，所以 Android 提供了下列关闭 activity 和 service 的方法：

- Activity 可以通过调用它自己的 finish()方法来关闭。一个 Activity 也可以通过调用 finishActivity()来关闭另一个 Activity（只能是由它通过调用 startActivityForResult()启动的）。
- 一个 service 可以通过调用自己的 stopSelf()方法，或者 Context.stopService()方法来结束。

当组件不再使用时或者 android 为了别的更活跃的组件能运行而回收内存时，android 系统会关闭这些组件，在后面介绍组件生命周期的部分会有更多详细的介绍。

manifest 文件(The manifest file)

在 Android 启动一个应用程序组件之前，它必须知道该组件确实存在。因此，应用程序在 manifest 文件中声明了他的全部组件。manifest 文件随同应用程序的代码、文件、资源一同

打包在了 Android 包中，即.apk 文件中。

manifest 是一个结构化的 XML 格式文件，对所有应用程序它都是命名为 AndroidManifest.xml。除声明应用程序组件之外，它还做一些别的事情，例如为所有应用程序需要进行连接的库命名（除默认 Android 库）以及识别所有应用程序期望获得的权限。

但是 manifest 的主要任务还是为 Android 提供应用程序组件的的信息。例如，一个 activity 可以声明如下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
   <application>
       <activity android:name="com.example.project.FreneticActivity"
                 android:icon="@drawable/small_pic.png"
                 android:label="@string/freneticLabel"
        ... >
        </activity>
   ......
   </application>
</manifest>
```

<activity>元素的 name 属性指定实现了 activity 的 Activity 子类。icon 和 label 属性指向包括一个图标和标签的资源文件，这些资源文件可以在 activity 被显示时显示给用户。

其他的组件也以同样的方式声明——声明 service 使用<service>元素，broadcast receiver 使用<receiver>元素，content provider 则使用<provider>元素。系统将无法看到没有声明在 manifest 里的 activity，service 以及 content provider，当然也就永远无法运行它们。但是 broadcast receiver 既可以声明在 manifest 里，也可以在代码里动态创建（如创建 Broadcast Receiver 对象），然后通过调用 Context.registerReceiver()注册到系统里。

更多关于构建应用程序的 manifest 文件的信息，请参照 The AndroidManifest.xml File

Intent 过滤器(Intent filters)

Intent 对象能够明确的指定一个目标组件。如果它指定了，Android 将找到那个组件(根据 manifest 文件里的声明)并激活它。但是如果目标没有被明确的被指定，Android 就必须找到最佳的组件来响应 intent。Android 将 Intent 对象与 intent filter 中可能的目标组件相比较来找出最佳的组件。组件的 intent filter 通知 Android 该组件所能够使用的 intent 的种类。和其他关于组件的必要的信息一样，它们也在 manifest 文件里声明。这里扩展了前面的例子，将两个 intent filter 加到 activity 里去：

```
<?xml version="1。0" encoding="utf-8"?>
<manifest... >
   <application ...>
      <activity android:name="com.example.project.FreneticActivity"
                android:icon="@drawable/small_pic.png"
                android:label="@string/freneticLabel"
                ... >
         <intent-filter... >
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
         <intent-filter..... >
```

```
            <action android:name="com.example.project.BOUNCE" />
            <data android:mimeType="image/jpeg" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
    </activity>
    ......
  </application>
</manifest>
```

例子中的第一个 filter 的动作 action.MAIN 和目录 category.LAUNCHER 组合是很常见一种。它标识这个 activity 要在应用程序启动器屏幕中被列出，就是可以列出设备上用户可用的的应用程序的那个屏幕。换句话说，该 activity 是应用程序的入口点，是用户在应用程序启动器上选择该应用程序时所看到的初始的 activity。

第二个 filter 声明一个 action，使得该 activity 能够处理一种特定类型的数据。

一个组件可以有任意数量的 intent filter，每一个声明这个组件的一种能力。如果一个组件不含任何 filter，它只能够被明确指定作为目标组件的 intent 激活。

对于在代码里生成和注册的 broadcast receiver，intent filter 将作为 IntentFilter 对象直接实例化。所有其他 filter 都是在 manifest 中创建。

更多 intent filter 相关内容，请参照另外的文档 Intents and Intent Filters。

活动和任务(Activities and Tasks)

上文提到，一个 Activity 可以启动另一个 Activity，即使这个 Activity 是定义在另一个应用程序里的，比如说，你想展示给用户一条街的地图，现在已经有一个 Activity 可以做这件事，那么现在你的 Activity 需要做的就是将请求信息放进一个 Intent 对象里，并且将这个 Intent 对象传递给 startActivity()，地图就可以显示出来了，但用户按下 BACK 键之后，你的 Activity 又重新出现在屏幕上。

对用户来说，显示地图的 Activity 和你的 Activity 好像在一个应用程序中的，虽然是他们是定义在其他的应用程序中并且运行在那个应用进程中。android 将你的 Activity 和借用的那个 Activity 放进一个 task 里，以维持用户体验。简单来讲，task 就是用户觉得好像是一个"应用程序"的东西。task 是以栈的形式组织起来的一组相互关联的 Activity，，栈中底部的 Activity 就是开辟这个 task 的，通常是用户在应用程序启动器中选择的 Activity。，栈顶部的 Activity 是当前正在运行的 Activity——用户正在交互操作的 Activity。当一个 Activity 启动另一个 Activity 时，新启动的 Activity 被压进栈中，成为正在运行的 Activity。旧的 Activity 仍然在栈中。当用户按下 BACK 键后，正在运行的 Activity 弹出栈，旧的 Activity 恢复成为运行的 Activity。

栈中包含对象，因此如果一个任务中开启了同一个 Activity 子类的多个对象——例如，多个地图浏览器——则栈对每一个实例都有一个单独的入口。栈中的 Activity 不会被重新排序，只会被压入、弹出。

task 是一组 Activity 实例组成的栈，不是在 manifest 文件里的某个类或者元素，所以无法设定一个 task 的属性而不管它的 Activity，一个 task 的所有属性值是在底部的 Activity 里设置的。例如， 下一节会讲到"任务的 affinity"，affinity 信息就是从底部 Activity 中获取的。

一个 task 里的所有 Activity 作为一个整体运转。整个 task（整个 Activity 堆栈）可以被送到前台或者被推到后台。假设一个正在运行的 task 中有四个 Activity——正在运行的 Activity 下面有三个 Activity，这时用户按下 HOME 键，回到应用程序启动器然后运行新的应用程序（实际上是一个新的 task），那么当前的 task 就退到后台，新开启的应用程序的 root Activity 此时就显示出来了；一段时间后，用户又回到应用程序启动器，又重新选择了之前的那个应用程序（先前的那个 task），那么先前的那个 task 此时又回到了前台了，当用户按下 BACK 键时，屏幕不是显示刚刚离开的新开启的那个应用程序的 Activity，而是移除回到前台的这个 task 的栈顶 Activity，将这个 task 的下一个 Activity 显示出来。

以上描述的情况是 Activity 和 task 默认的行为，但是那个行为的几乎所有方面都是可以修改的。Activity 和 task 的关系，以及 task 中 Activity 的行为，是受启动该 Activity 的 Intent 对象的标识和在 manifest 文件中的 Activity 的<activity>元素的属性共同影响的。

在这种情况下，主要的 Intent 控制标识有：

```
FLAG_ACTIVITY_NEW_TASK
FLAG_ACTIVITY_CLEAR_TOP
FLAG_ACTIVITY_RESET_TASK_IF_NEEDED
FLAG_ACTIVITY_SINGLE_TOP
```

重要的<activity>属性有：

```
taskAffinity
launchMode
allowTaskReparenting
clearTaskOnLaunch
alwaysRetainTaskState
finishOnTaskLaunch
```

稍后会描述这些标识和属性做什么，它们之间如何相互作用，使用它们时应该注意什么。

affinity 和新的 tasks(Affinities and new tasks)

默认的，一个应用程序中的所有 Activity 都有血缘关系——就是都属于同一个 task 的。但是，可以通过<activity>元素下的 taskAffinity 属性来为某个 Activity 设置单独的 affinity。定义在不同应用程序中的 Activity 可以共享一种 affinity，或者一个应用中的不同的 Activity 可以定义不同的 affinity。affinity 满足以下两种情况时起作用：一是当启动 Activity 的 Intent 对象包含有 FLAG_ACTIVITY_NEW_TASK 标志时，二是当 Activity 的 allowTaskReparenting 属性设置为“true”。

FLAG_ACTIVITY_NEW_TASK 标志位

如前面提到的，默认情况下，Activity 调用 startActivity()启动一个新的 Activity 时，新的 Activity 会压入到相同的 task 中，但是如果传递给 startactivity()的 Intent 对象含有

FLAG_ACTIVITY_NEW_TASK 标志，系统就会寻找一个新的 task 来装这个新的 Activity。通常就如控制标识的字面的意思一样，它是一个新的 task。然而并不是一定要那样，如果已经有一个 task 和这个新的的 Activity 有相同的 affinity，那么就把这个新的 Activity 放进那个 task 里，如果没有，就启动一个新的 task。

allowTaskReparenting 属性

如果一个 Activity 的 allowTaskReparenting 属性设置为 true，这个 Activity 就可以从启动时的那个 task 移动到一个和它有相同 affinity 的前台的一个 task 里去，比如，假设现在有一个用于天气预报的 Activity 被定义在一个旅行的应用程序里，他和这个应用里的其他 Activity 有相同的 affinity（默认的 affinity），并且允许重定父级。现在你自己的应用程序中有一个 Activity 启动这个天气预报的 Activity，那么天气预报 Activity 就会移动到你的 Activity 所在的 task 里，当旅行的应用程序回到前台时，天气预报 Activity 重新回到以前的那个 task 并显示。（译者注：如果说没有设置这个属性，或者这个属性设置为 false，那么一个应用里的 Activity 调用另一个应用里的 Activity 时，系统是为另一个应用里的 Activity 创建一个实例，然后放到同一个 task 里，但是如果设置了 allowTaskReparenting 为 true，那么另一个应用里的 Activity 是可以在不同的 task 间来回移动的，那个 task 在前台就移动到那个 task 里）

如果从用户的角度来看，一个.apk 文件里包含不止一个"应用程序"的话，你将可能会给每个 activity 指定不同的 affinity 来关联它们。

启动模式(Launch modes)

<activity>下的 launchMode 属性可以设置四种启动方式：

```
standard （默认模式）
singleTop
singleTask
singleInstance
```

这些模式有以下四点不同：

- **响应 Intent 时 Activity 将被装入哪个 task。**对于 standard 和 singleTop 模式，由产生该 Intent(调用 startActivity()) 的 task 持有该 Activity——除非 Intent 对象里含有 FLAG_ACTIVITY_NEW_TASK 标志，那么就像前面章节讲的那样的寻找一个新的 task。

  相反的，singTask 和 singleInstance 模式，总是标志 Activity 为 task 的 root Activity，开启这样的活动会新建一个 task，而不是装入某个正在运行的任务。

- **一个 Activity 是否可以有多个实例。**一个 standard 或者 singleTop 属性的 Activity 可以实例化多次，他们可以属于多个不同的 task，而且一个 task 也可以含有相同 Activity 的多个实例。

  相反的，singleTask 或者 singleInstance 属性的 Activity 只能有一个实例（单例），因为

这些 Activity 是位于 task 的底部，这种限制意味着同一设备的同一时刻该 task 只能有一个实例。

- **实例是否能允许在它的 task 里有其他的 Activity。**一个 singleInstance 属性的 Activity 是它所在的 task 里仅有的一个 Activity，如果他启动了另一个 Activity，那个 Activity 会被加载进一个不同的 task 而无视它的启动模式——就如 Intent 里有 FLAG_ACTIVITY_NEW_TASK 标识一样。在其他的方面，singleInstance 和 singleTask 一样的。

  其他三个模式允许有多个 Activity 在一个 task 里，一个 singleTask 属性的 Activity 总是一个 task 里的 root Activity，但是他可以启动另外的 Activity 并且将这个新的 Activity 装进同一个 task 里，standard 和 singleTop 属性的 Activity 可以出现在 task 的任何位置。

- **是否创建一个新的 Activity 实例来处理一个新的 Intent。**对于默认的 standard 方式，将会生成新的实例来处理每一个新的 Intent。每个实例处理一个新的 Intent。对 singleTop 模式，如果一个已经存在的实例在目标 task 的栈顶，那么就重用这个实例来处理这个新的 Intent，如果这个实例存在但是不在栈顶，那就不重用他，而是重新创建一个实例来处理这个新的 Intent 并且将这个实例压入栈。

  例如现在有一个 task 堆栈 ABCD，A 是 root Activity，D 是栈顶 Activity，现在有一个启动 D 的 Intent 来了，如果 D 是默认的 standard 方法，那么就会创建一个新的实例来处理这个 Intent，所以这个堆栈就变为 ABCDD，然而如果 D 是 singleTop 方式，这个已经存在的栈顶的 D 就会来处理这个 Intent，所以堆栈还是 ABCD。

  如果另外一种情况，到来的 Intent 是给 B 的，不管 B 是 standard 还是 singleTop（因为现在 B 不在栈顶），都会创建一个新的实例，所以堆栈变为 ABCDB。

  如上所述，一个"singleTask"或"singleInstance"模式的 activity 只会有一个实例， 这样它们的实例就会处理所有的新 intent。一个"singleInstance" activity 总是在栈里的最上面(因为它是 task 里的唯一的 activity)， 这样它总是可以处理一个 intent。而一个"singleTask" activity 在栈里可以有或没有其他 activity 在它上面。如果有的话，它就不能对新到的 intent 进行处理，intent 将被丢弃。(即使 intent 被丢弃，它的到来将使 task 来到前台，并维持在那里。)

当一个已有的 Activity 被请求去处理一个新的 Intent 时，Intent 对象会通过 onNewIntent()的调用传递给这个活动。(传递进来的原始的 Intent 对象可以通过调用 getIntent()获取)。

注意，当创建一个新的 Activity 的实例来处理一个新收到的 Intent 时，用户可以按 BACK 键回到上一个状态（上一个 Activity）。但是使用一个已有的 Activity 实例操作新收到的 Intent 时，用户不能通过按下 BACK 键回到这个实例在接受到新 Intent 之前的状态。

启动模式的更多信息，参见清单文件<activity>元素的描述。

清理堆栈(Clearing the stack)

当用户长时间没有使用一个运行着的 task，系统就会清理掉 task 里除了 root Activity 以外的所有的 Activity，当用户再次使用这个 task 时，显示的是 root Activity。之所以这样做是认为，用户长时间不使用这个 task，就很可能是希望放弃他们之前的操作，再次回到这个 task 就要重新开始。

上面说的是默认的情况，有一些 Activity 的属性可以用来控制和修改这些行为：

alwaysRetainTaskState 属性

如果一个 task 里的 root Activity 的 alwaysRetainTaskState 属性设置为 true，那么前面描述的默认情况就不会出现了，无论用户多长时间没有使用 task，task 也会一直保留栈中所有的 Activity。

clearTaskOnLaunch 属性

如果一个 task 里的 root Activity 的 clearTaskOnLaunch 属性设置为 true，和 alwaysRetainTaskState 相反，即使是一瞬间的离开，系统马上就会清理掉 task 里除 root Activity 以外的所有 Activity，task 变回初始的状态。

finishOnTaskLaunch 属性

这个属性和 clearTaskOnLaunch 一样，但是他是对一个 Activity 起作用，不是整个 task，他能引起所有的 Activity 离开，包括 root Activity，当这个属性设置为 true，只是当用户使用这个应用程序时 Activity 才在 task 里，一旦用户离开该 task 后重新回来，该 Activity 不再存在。

还有一种方法来从 task 里强制移除 Activity，如果一个 Intent 对象里包含 FLAG_ACTIVITY_CLEAR_TOP 标志，并且目标 task 里已经有一个可以处理此 Intent 的 Activity 实例，那么在栈中这个 Activity 之上的所有 Activity 将被清除，这时这个 Activity 就位于栈顶，可以响应该 Intent。如果此 Activity 的启动模式是"standard"，那么这个 Activity 本身也会被移除出栈，建立新的 Activity 实例来处理这个 Intent。这是因为如果启动模式是"standard"，那么每一个 Intent 都会用一个新的实例进行处理。

FLAG_ACTIVITY_CLEAR_TOP 通常会和 FLAG_ACTIVITY_NEW_TASK 一起使用。同时使用时，这个组合是找到另一个 task 中的已有 Activity 然后将它转入新的 task 中以响应一个 Intent 的一种方法。

启动任务(Starting tasks)

可以通过将 activity 的 intent filter 的 action 指定为“android.intent.action.MAIN”，以及类别(category)指定为“android.intent.category.LAUNCHER”，可以将它设置为 task 的入口 activity。(关于这种类型的 filter，在上面的章节的 Intent Filters 里有一个例子)。一个该类型的 filter 使该 activity 的图标和标签显示在应用程序启动器(launcher)上，这样就提供给用户一个方法，既可以加载该 task，又可以在它被加载后随时回到它。

第二个功能很重要：用户必须能够在离开一个 task 后回到这个 task。“singleTask”和“singleInstance”这两种启动模式的 activity 总是启动一个 task，所以只有当 activiay 有 MAIN 和 LAUNCHER filter 的时候才使用。设想，如果在没有这两个 filter 的 activity 中使用将会发生什么：一个 Intent 激活了一个"singleTask"的 activity，创建了一个新的 task，用户在这个 task 中做了一些操作，然后用户按下 HOME 键。这个 activity 就退到后台，但是由于这个 activity 不在应用程序启动器(launcher)中显示，用户无法再回到那个 activity 中了。

相似的难题在使用 FLAG_ACTIVITY_NEW_TASK 控制标识时也会出现。如果该标识使一个 activity 开始了一个新的 task，然后用户按了 HOME 键离开这个 activity，也没有办法再回来了。一些东西(例如通知管理器)总是在一个新的 task 里打开 activity，而从来不在自己的 task 中打开，所以它们总是将包含 FLAG_ACTIVITY_NEW_TASK 的 Intent 传递给 startActivity()。所以如果你有一个可以被其他的东西以这个控制标志调用的 activity，请注意用户有独立的回到这个 activity 的方法。

如果你希望用户离开 activity 之后就不能回到这个 activity，可以将<activity>元素的 finishOnTaskLaunch 的值设为"true"。参照之前的 Clearing the stack。

进程和线程（Processes and Threads）

当应用程序第一个组件需要运行时，android 系统就为这个组件分配一个 Linux 进程，这个进程只有一个运行线程。默认这个应用程序所有组件都运行这个进程中的这个线程中。

但是你可以将一些组件运行在其他进程中，并且你可以为任意的进程添加线程。

进程

组件运行在哪个进程中是在 manifest 文件里设置的，四大组件——<activity>， <service>，<receiver>和<provider>——都有一个 process 属性来指定组件运行在哪个进程中。你可以通过设置这个属性，使得每个组件运行在它们自己的进程中，或者几个组件共享一个进程，或者不共享。你甚至可以设定位于不同的应用程序中的组件运行在同一个进程中——这两个不同的应用程序需要由同一作者签名，并且分享同一个 linux 用户 ID，<application>元素也有一个 process 属性，用来指定所有组件的默认属性。

所有的组件都在指定的进程中的主线程中实例化的，对组件的系统调用也是由主线程发出的。每个实例不会建立新的线程。对系统调用进行响应的方法——比如负责报告用户动作的 View.onKeyDown()和后面讨论的组件生命周期通知函数——都是运行在这个主线程中的。这意味着当系统调用这个组件时，这个组件不能长时间的阻塞线程(比如说网络操作，循环计算)，因为这样会阻塞这个进程中其他组件，你可以将很耗时的任务分到其他的线程中。

当内存不足并且有其他更紧急的进程请求时，Android 系统可能结束一个进程，运行在这个进程中的组件会被销毁，当用户重新打开这个应用程序时时，系统会重新启动这个进程。

Android 权衡进程的重要性来决定结束哪个进程。例如，一个在后台的进程比正在显示的进程更容易被结束。是否要结束某个进程是由里面运行的组件的状态决定的。组件的状态在下一章节，组件生命周期中讨论。

线程

即使你限制了你的应用程序运行在一个进程中，但是有的时候你可能需要新开一个线程在后台运行。因为用户界面需要随时对用户的动作做出反应，所以一些很耗时的工作应该重新启动一个线程来做，以免阻塞主进程。

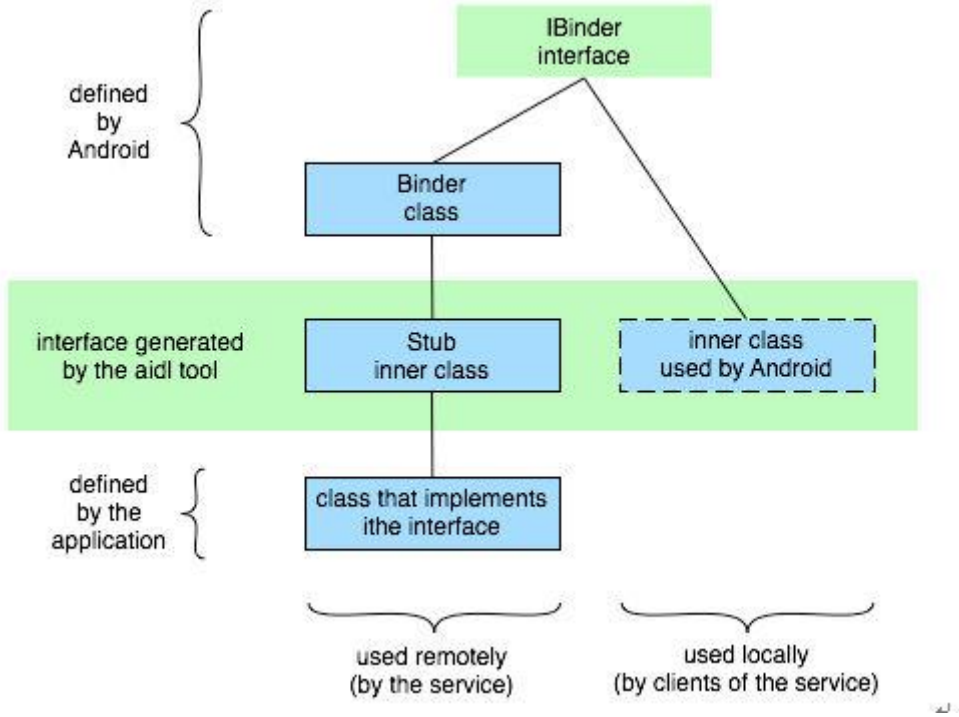
线程在代码中使用标准的 java Thread 对象来建立，Android 系统提供了一系列方便的类来管理线程——Looper 用来在一个线程执行消息循环，Handler 处理消息，HandlerThread 创建带有消息循环的线程。

远程调用（Remote procedure calls）

Android 系统有一个轻量级的远程调用机制（RPCs）——使方法在本地调用，在远程执行（在另外一个进程里），并将所有结果返回本地。这需要将方法的调用和随之的数据解释成操作系统可以识别的级别，将数据从本地进程和地址空间传递到远程的进程和地址空间，并在远端重新装配和组织。返回数据的时候传输方向相反，android 系统会去做这些传输的工作，让你能够集中精力来定义和实现你的 RPC 接口。

一个 RPC 接口可以只包含包含方法，所有的方法都是同步执行的(本地方法会等待远程方法)，在没有返回值的情况下也是这样。

简单的说，RPC 机制是这样工作的：首先你需要用 IDL（接口定义语言）声明你想要实现的 RPC 接口，然后使用 aidl 工具来生成该接口的 java 接口定义，该接口在本地进程和远端进程都是可用的，这个 java 接口中包含了两个内部类，如下图所示：



这两个内部类管理你用 IDL 声明的接口的远程调用的所有代码，两个内部类都实现 IBinder 接口，一个是在系统在本地内部使用，你自己写的代码可以忽略它。另外一个叫做 Stub，继承自 Binder 类。除了包含执行 IPC 调用的代码，还包含你在声明的 RPC 接口中的方法的声明，你应该继续继承 Stub 类来实现这些方法，就像图中所示。

一般的，远端进程应该由 service 来管理（因为 service 能够将进程以及它与其他进程间的连接相关情况通知给系统)。这样的 service 既有 aidl 工具生成的接口文件，又包含了 RPC 方法的实现的 Stub 的子类。service 的客户端将只有 aidl 工具生成的接口文件。

以下是 service 如何和它的客户端建立连接：

- Service 的客户端(在本地)将实现 onServiceConnected()和 onServiceDisconnected()方法，这样当一个到远程 service 连接建立或断开的时候，客户端可以得到这一消息。然后客户端将调用 bindService()方法来设置这个连接。
- 根据接收到的 Intent(传递给 bindService()的 intent)，service 的 onBind()方法将被用于实现接受或者拒绝连接。如果连接被接受，该方法返回一个 Stub 子类的实例。
- 如果该 service 接受了该连接，Android 调用客户端的 onServiceConnected()方法并传递给它一个 IBinder 对象，IBinder 对象是由 service 管理的 Stub 子类的代理。通过该代理，客户端能够调用远程 service。

这个简要的 RPC 机制介绍省略了一些细节。更多的信息请参照 Designing a Remote Interface Using AIDL 和 IBinder 类的介绍。

线程安全方法(Thread-safe methods)

在某些时候，你实现的方法可被不止一个线程调用，因此你的实现必须是线程安全的。

这种情况主要出现在被远程调用的方法中——比如在前面讨论的 RPC 机制。当一个实现了 IBinder 对象方法的调用发生在该 IBinder 相同的进程里，方法在调用方的线程里执行。但是当调用来自另外的进程时，方法将运行在 Android 为 IBinder 进程维护的线程池里选择的一个线程中，而不会运行在另外那个进程的主线程中。例如，尽管一个 service 的 onBind()方法的调用来自 service 进程的主线程，onBind()返回的对象的实现的方法(比如一个实现了 RPC 方法 Stub 子类)会被在线程池中的线程调用。因为 service 可以有很多客户端，因此在同一时间可能有多个线程池中的线程调用了 IBinder 方法。因此 IBinder 方法必须被实现成线程安全的。

类似的，一个 content provider 能够接收来自其他进程的数据请求。尽管 ContentResolver 和 ContentProvider 类隐藏了管理内部进程间通信的细节，可是响应这些请求的 ContentProvider 方法—query()，insert()，delete()，update()和 getType()方法—是从 content provider 进程的线程池中调用的，而不是从该进程的主线程。因为这些方法在同一时刻可能被任意数量的线程调用，所以它们也必须被实装为线程安全的。

组件生命周期(Component Lifecycles)

程序组件有生命周期——从开始 Android 实例化它们以响应 intent 到实例被销毁时结束。在这段时间，组件有时候是激活状态，有时候是不活跃状态，对于 activity 组件而言，就是用户用户时而可见时而不可见。本节讨论 activity，service 以及 broadcast receiver 的生命周期——包括它们存在时可能处于的状态，状态转换时通知的方式，和那些状态对掌管它们的进程的影响。

活动的生命周期(Activity lifecycle)

Activity 基本上有三种状态：

- 当它在屏幕前端时(处于当前 task 的 activity 栈的顶端)状态为激活（active）或正在运行（running）。该 activity 是用户动作的焦点。
- 如果它失去用户焦点，但是仍然对用户可见，它的状态为暂停(paused)。也就是说，另一个 activity 在其上方，并且那个 activity 是透明的或者未覆盖整个屏幕，因此这个暂停的 activity 仍然有一部分显示出来。一个暂停的 activity 依然是活动的(它保存了所有的状态和成员信息并且和窗口管理器连接)，但是可以在内存不足时被系统关闭。
- 当它被另一个 activity 完全掩盖时状态为停止(stopped)。这时它仍然保存了所有的状态和成员信息。然而，它对用户来说不再可见，它的窗口被隐藏，并且在其他地方需要内存的时候常常会被系统 kill。

如果一个 activity 处于暂停或停止状态，系统可以通过调用 finsish()方法或者简单地直接 kill 进程来将其从内存中清理掉。当它再次显示给用户时，就必须完全重启并恢复到原来的状态。

当一个 activity 转换状态时，系统会通过调用下面受保护(protected)的方法对其进行通知：

```
void onCreate(Bundle savedInstanceState)
void onStart()
void onRestart()
void onResume()
void onPause()
void onStop()
void onDestroy()
```

所有这些方法都是可以被重载来做状态改变时做适当工作的钩子(hooks)。在对象第一次被实例化的时候，所有的 activity 必须执行 onCreate()方法来做初始化工作。许多 activity 也实现 onPause()来确认数据改变并准备好停止和用户交互。

加在一起，这七个方法定义了一个 activity 的完整的生命周期。这儿有三个内嵌循环，你可以通过实现它们来进行监听：

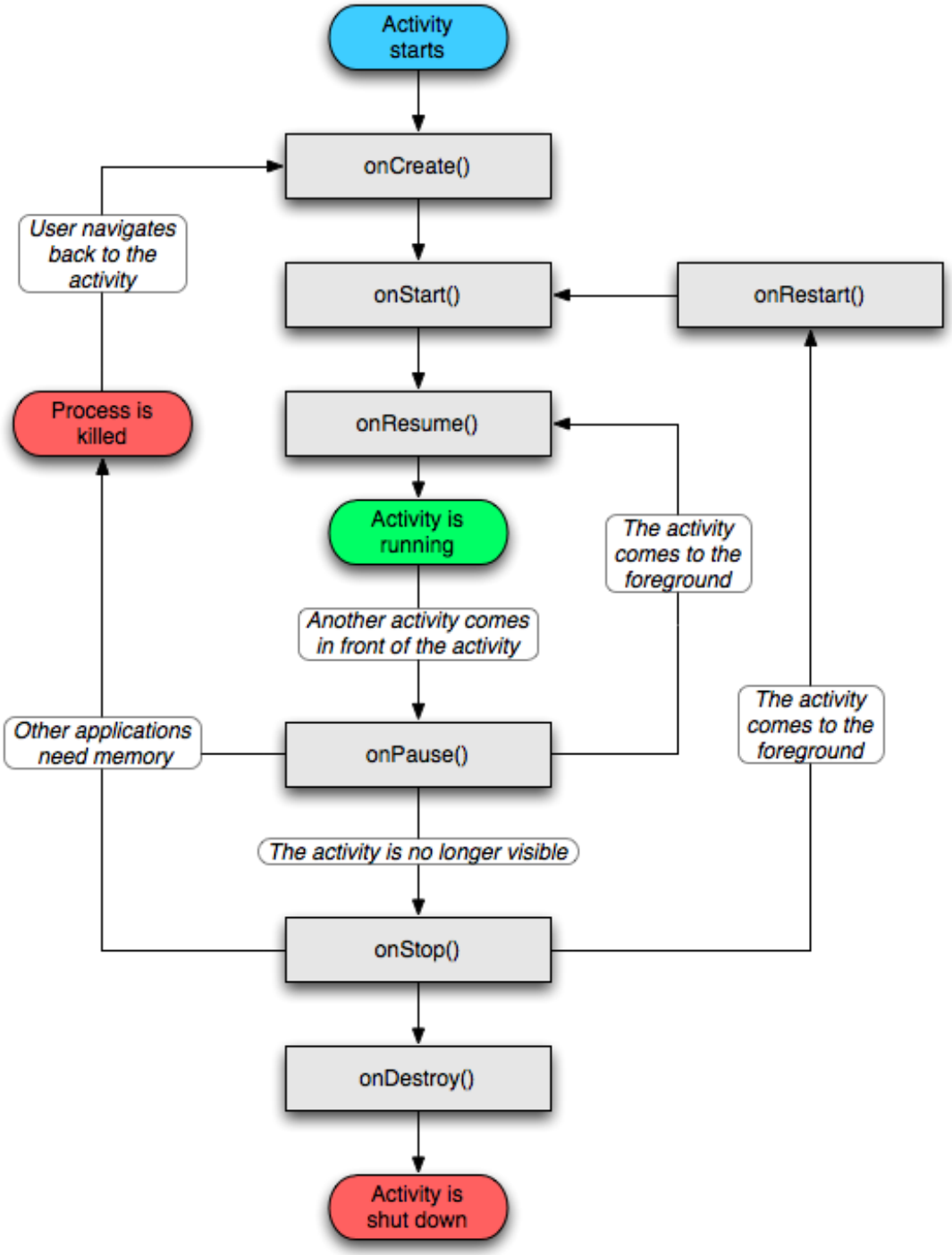
调用超类(Calling into the superclass)

实现 activity 的任何生命周期方法的时候都必须首先调用父类的方法，例如:

```
protected void onPause() {
      super。onPause();
    ......
}
```

- activity 的**完整生命周期**起始于 onCreate()的初次调用，结束于单一的 onDestroy()调用。activity 通过 onCreate()进行它"global"状态的初步建立，通过 onDestroy()释放所有剩余资源。例如，如果它有一个从网络下载数据的线程运行在后台，它可以通过 onCreate()生成那个线程以及通过 onDestroy()停止该线程。
- activity 的**可视生命周期**从一个 onStart()调用开始，直到一个相对的 onStop()调用。在此期间，用户可以在屏幕上看到该 activity，尽管它可能不在前台并与用户交互。在这两个方法之间，你可以维护需要向用户显示该 activity 的所需资源。例如，你可以使用 onStart()注册一个 BroadcastReceiver 来监听对你的 UI 有影响的变化，以及当用户不再能够看到你所显示的时候通过 onStop()取消注册。当 activity 在用户可视和不可视间交替的时候，onStart()和 onStop()方法能够被多次调用。
- activity 的**前台生命周期**开始于一个 onResume()调用，终止于一个相对的 onPause()调用。在此期间，该 activity 位于屏幕上所有其他 activity 的前面并与用户交互。一个 activity 可以频繁地在重新恢复和暂停状态之间转换—例如，当该设备变成睡眠状态或当开启一个新的 activity 时，onPause()被调用，当一个 activity 返回结果或者收到一个新的 intent 时，onResume()被调用。因此，这两个方法里的代码应该相当轻量。

下面的图示说明了这些循环和一个 activity 在各个状态间转换的步骤。彩色的椭圆是 activity 所处的主要状态。长方形表示当 activity 在状态间转换的时候，你可以实现的用来执行操作的回调方法。



下面的表格描述了这些方法的详情，和在整个活动生命周期中的位置：

方法	描述	Killable?	下一个
onCreate()	activity 最初建立时被调用。在这里你应该做通常的所有静态的常规的设定—建立 view，绑定数据到列表等等。这个方法调用时如果有先前状态可用，会接受到一个包含这个活动之前的状态的 Bundle 对象。(参照稍后的 Saving Activity	No	onStart()

	State)。下一步总是 onStart()。		
onRestart()	当 activity 被停止后，被再启动之前调用。下一步总是 onStart()	No	onStart()
onStart()	在 activity 变成对用户可视之前调用。如果活动切到前台，下一步是调用 onResume()；如果活动被隐藏，下一步是调用 onStop()。	No	onResume() 或 onStop()
onResume()	在 activity 开始与用户交互之前调用。此时，该 activity 位于 activity 栈的最上端，用户在其上进行输入。下一步总是 onPause()。	No	onPause()
onPause()	当系统要开始恢复另外一个 activity 时被调用。该方法通常用于将未保存的数据保存为永久数据，停止动画和其他可能消耗 CPU 的处理，等等。它必须快速进行任何该做的处理，因为在它返回后下一个 activity 才开始。 如果 activity 回到前台，下一步是 onResume()； 如果它变成用户不可视，下一步是 onStop()。	Yes	onResume() 或 onStop()
onStop()	当 activity 不再对用户可视时被调用。它可能发生在它就要被销毁时，或者另外一个 activity(无论一个既存的或是一个新的)被恢复并且覆盖了它。 如果 activity 又回来和用户交互，下一步是 onRestart()； 如果该 activity 将被销毁，下一步是 onDestroy()。	Yes	onRestart() 或 onDestroy()
onDestroy()	在 activity 被销毁之前被调用。这是该 activity 所能接到的最后调用。当活动完成之后（有些人调用 finish()）或系统为了节约空间将这个实例暂时销毁 时会被调用。您应该可以看出和调用 isFinishing()方法之间的区别。	Yes	Nothing

注意上表中的 Killable 栏。它标明系统是否可以在该方法返回的任何时候杀死该 activity 所在进程，而无需执行该 activity 的其他行代码。三个方法(onPause()，onStop()和 onDestroy())标记为"Yes"。因为 onPause()是三者的第一个，它是唯一一个允许在进程被杀死前调用的方法，onStop()和 onDestroy()可能不被调用。因此，你应该使用 onPause()将数据(比如用户编辑)进行永久存储。

Killable 栏被标明"No"的方法在被调用的时候能够保护该 activity 所处进程不会被杀死。例如，从 onPause()返回的时刻到 onResume()被调用的时刻，一个 activity 处于可杀死状态。它将不会处于可被杀死状态直到 onPause()再次返回。

就如稍后章节 Processes and lifecycle 将要提到的，技术上来讲，一个 activity 在这个定义的时候不"可杀"，但仍然可能被系统杀死—但也仅仅当已经没有其他资源的极端严重的情况下才会发生。

保存活动状态(Saving activity state)

当系统而不是用户关闭一个 activity 以节省内存，该用户可能希望返回到该 activity 的时候，它仍处于之前的状态。

你可以实现一个 activity 的 onSaveInstanceState()方法，从而在该 activity 被杀死前采集它的状态。Android 在 activity 将被销毁前调用该方法，也就是，在 onPause()被调用之前。它传递给该方法一个 Bundle 对象，那里你可以以键值对(name-value pairs)方式记录 activity 的动态状态。当该 activity 重启时，该 Bundle 会传递给 onCreate()和在 onStart()之后调用的方法，onRestoreInstanceState()，这样它们任何一方都可以重新生成采集的状态。

不像在前面讨论的 onPause()和其他方法，onSaveInstanceState()和 onRestoreInstanceState()不是生命周期方法。它们不是总被调用。例如，Android 在该 activity 将被系统销毁之前调用 onSaveInstanceState()，但是当该实例由于用户操作(比如按了 BACK 键)而被销毁前，不会特意调用它。在这种情况下，用户不会希望返回该 activity，所以也没有理由保留它的状态。

因为 onSaveInstanceState()不是总被调用，你应该使用它仅仅记录该 activity 的短暂状态，而不是储存为永久性数据。那种情况应该改用 onPause()。

协调活动(Coordinating activities)

当一个 activity 开始了另外一个，它们都经历了生命周期的转变。一个 activity 暂停或者停止，而另外一个 activity 启动了。在这种场合，你可以需要协调这些 activity。

生命周期回调方法的顺序已经定义好，尤其当两个 activity 在同一个进程里的时候：

1. 当前的 activity 的 onPause()方法被调用。
2. 接着，被启动的 activity 的 onCreate()，onStart()和 onResume()方法相继被调用。
3. 然后，如果旧的的 activity 在屏幕上不再可视时，它的 onStop()方法被调用。

服务的生命周期(Service lifecycle)

一个 service 能够以两种方式使用：

- 它能够被启动和允许运行，直到有人停止了它或它自己停止自己为止。在这种模式下，通过调用 Context.startService()启动它，通过调用 Context.stopService()停止它。它能够通过调用 Service.stopSelf()或 Service.stopSelfResult()停止自己。不管调用过多少次 startService()，仅仅调用一次 stopService()就可以停止 service。
- 通过它定义和导出的接口，可以对 service 进行程式化操作。客户端建立到 Service 对象的连接，然后使用这个连接对 service 进行请求。通过调用 Context.bindService()建立连接，通过调用 Context.unbindService()关闭连接。多个客户端可以绑定到相同的 service。如果该 service 还没有被载入，bindService()能够选择启动它。

两种模式不是完全分离的。你能够绑定到一个由 startService()启动的 service 上。例如，一个背景音乐的 service 可以通过调用 startService()被启动，它带有一个确认音乐播放的 Intent 对象。片刻之后，可能是用户想通过播放器进行某些控制或者取得当前歌曲的信息时，通过调用 bindService()，将一个 activity 建立到这个 service 的连接。像这样的情况，stopService()将不会真正的停止该 service，直到最后的绑定被解开。

就像 activity，一个 service 也有生命周期方法，你可以实现它来监听它的状态的变化。但是它们比 activity 方法只有三个，并且它们是 public 的，不是 protected 的：

```
void onCreate()
void onStart(Intent intent)
void onDestroy()
```

通过实现这些方法，你可以监听 service 生命周期的两个内嵌的循环：

- 一个 service 的完整生命周期起始于 onCreate()的调用，终止于 onDestroy()§的返回。像一个 activity，一个 service 通过 onCreate()进行它的所有的初始化，通过 onDestroy()释放所有它保留的资源。例如，一个音乐播放器 service 可以通过 onCreate()生成将要播放的音乐的线程，通过 onDestroy()停止该线程。
- 一个 service 的活动生命周期起始于对 onStart()的调用。该方法持有传递给 startService()的 Intent 对象。该音乐 service 将打开 Intent 找到播放哪个音乐，然后开始播放。

service 没有像 activity 的停止回调方法—没有 onStop()方法。

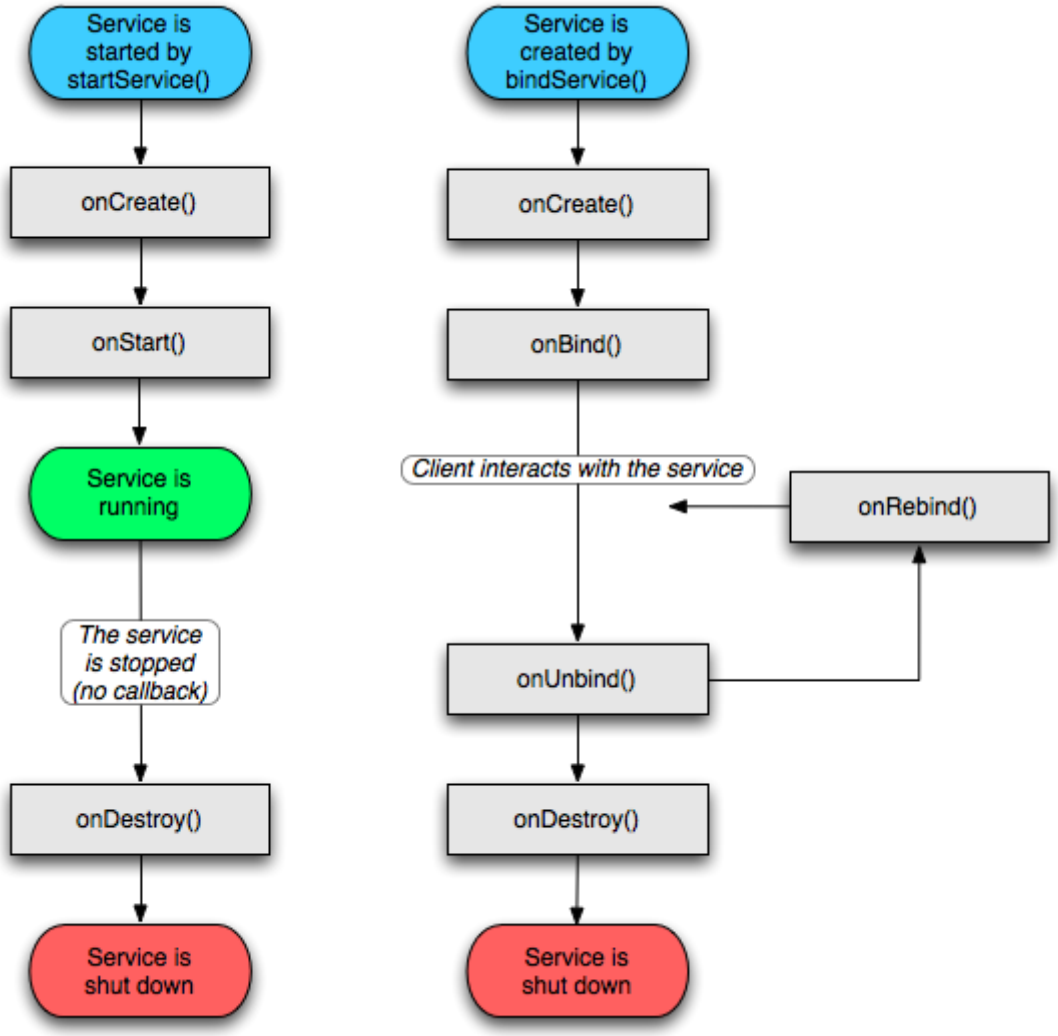
无论 service 是由 Context.startService()还是 Context.bindService()启动的，onCreate()和 onDestroy()方法都会被调用。但是，onStart()只在 service 是由 startService()启动的情况下被调用。

如果一个 service 允许别人绑定到它，还需要实现附加的回调方法：

```
IBinder onBind(Intent intent)
boolean onUnbind(Intent intent)
void onRebind(Intent intent)
```

onBind()回调获得传递给 bindService 的 Intent 对象，onUnbind()回调获得传递给 unbindService()的 intent 对象。如果该 service 允许绑定，onBind()返回客户端用来与该 service 交互的信道。如果一个新的客户端连接到该 service，onUnbind()方法能够要求调用 onRebind()。

下面的图示说明了一个 service 的回调方法。虽然，它区分 service 是由 startService 生成的还是那些由 bindService()生成的，但是请记住，所有的 sevice，不管它是如何启动的，一般都会允许客户端绑定到它，所以所有的 service 都可能接收到 onBind()和 onUnbind()调用。



广播接收器生命周期(Broadcast receiver lifecycle)

broadcast receiver 仅有一个回调方法：

```
void onReceive(Context curContext,  Intent broadcastMsg)
```

当一个 broadcast 信息到达该 receiver， Android 调用它的 onReceive()方法并将含有该广播信息的 intent 对象传递它。broadcast receiver 仅仅在执行该方法时才被认为是活跃的。当 onReceive()返回后，它又处于非活跃状态。

一个包含活跃的 broadcast receiver 的进程会被保护起来不被杀死。但是一个仅仅含有非活跃组件的进程，在它消耗的内存被其它进程需要时可能随时被系统杀死。

当该 broadcast 信息的响应很耗时时会存在问题，这时应该单独给他一个线程运行，而不是在其他组件所在的与用户交互的线程中。如果 onReceive()生成该线程后返回，整个进程，

包括那个新的线程，都被判断为非活跃的（除非该进程里的其他组件是活跃的），归入了可以被杀死的一类。解决该问题的答案是使用 onReceive()开始一个 service，让该 service 进行该处理，那样一来，系统就会知道该进程里仍有活跃的处理在进行。

接下来的章节将进一步讨论容易被杀掉的进程。

进程的生命周期（Processes and lifecycles）

Android 系统总是尽最大的努力来保留一个应用程序的进程，但系统的内存不足时就可能需要关闭一些旧的进程了，但是决定关闭哪个进程呢，android 系统把所以的进程放进一个重要度等级里，重要度最低的进程将会被停止，然后是次不重要的，依此类推。系统有 5 种重要性等级，重要性从高到低如下：

1. 前台进程。一个前台进程是当前执行用户请求的进程，如果有如下的一种情形的那么他就是前台进程：

   - 这个进程里运行着一个正在和用户交互的 Activity（这个 Activity 的 onResume（）方法被调用）。
   - 这个进程里有绑定到当前正在和用户交互的 Activity 的一个 service
   - 这个进程里有一个 service 对象，这个 service 对象正在执行一个它的生命周期的函数回调（onCreate()， onStart()， or onDestroy()）。
   - 这个进程里有一个正在执行 onReceive()方法的 broadcastreceiver 对象

   只有少数的前台进程在任何时间都存在，他们只有在最后的时刻被停止--系统的内存太少以至于不能运行这些仅有的前台进程了）。通常，在那个时刻，设备就到了内存分页状态，所以停止一些前台进程是为了保持对用户界面的快速响应。

2. 可视进程。可见进程不含任何前台显示的组件，但是仍然可以影响到用户当前屏幕所看见的东西，如果有如下的一种情形那么他就是可见进程。

   - 这个进程含有一个不位于前台的 Activity，但是仍然对用户是可见的（这个 Activity 的 onPause()方法被调用），这时很可能发生的，例如，如果前台 activity 是一个对话框的话，就会允许在它后面看到前一个 activity。
   - 这个进程里有一个绑定到一个可见 Activity 的 service

   可见进程非常重要的，除非为了保持所有前台进程的运行，才会杀死可见进程。

3. 服务进程。服务进程是一个通过 startService（）启动的但是没有在前两个分类中的进程，虽然服务进程不是用户直接能看见的，但是他也总是做一些用户很关心的事（如在后台播放 mp3，从网络上下载东西），所以系统会一直保持服务进程运行，除非内存不足以运行前台进程和可见进程。

4. 后台进程。后台进程是运行一个当前对用户是不可见的 Activity（这个 Activity 的 onStop()被调用），这些进程对用户体验没有什么直接的影响，当内存不足以运行前台

进程，可见进程，服务进程时，可以随时停止后台进程，通常系统会有很多的后台进程在运行，系统会把这些后台进程放进一个 LRU 中（最近使用列表），最近使用的就最后停止。如果一个 activity 正确的实现了它的生命周期方法，并且收集了它的当前状态，杀死它所在的进程将对用户体验没有任何有害影响。

5. 空进程。空进程就是进程里没有任何活动的应用组件，维持这种进程的唯一原因就是作为一种缓存，当一个组件需要启动时加快启动的速度，系统为了平衡进程缓存和核心缓存会停止这些空的进程。

Android 根据该进程里的当前活跃的组件的重要性，将进程设置在它能够属于的最高的等级里。例如，如果一个进程含有一个 service 和一个可视 activity，进程将被归入一个可视进程而不是 service 进程。

另外，如果其他进程依赖于它的话，一个进程的等级可以提高。一个服务于其他进程的进程将不会归入比它所服务的进程低的等级里。例如，如果一个 A 进程的 content provider 服务于 B 进程里的一个客户端，或如果一个 A 进程里的 service 被绑定到 B 进程里的组件上，进程 A 将总被认为至少和 B 进程一样重要。

因为一个运行着 service 的进程比一个运行着后台 activity 的等级高，一个 activity，要初始化一个长期运行的操作的话，可以为那个操作开始一个 service， 而不是单单生成一个线程，尤其在该操作比该 activity 更长久的情况下。相关的例子是在后台播放音乐和上传相机所拍摄的照片到网络上。使用一个 service 保证该操作将至少有“service 进程”权限，而与 activity 发生了什么无关。如之前在 Broadcast receiver lifecycle 章节里所提到的，由于同样的原因 broadcast receiver 应该使用 service 而不是简单的在一个线程里放置耗时操作。