Android 开发者指南(4) —— Application Fundamentals

前言

本章内容为开发者指南(Dev Guide)/Framework Topics/Application Fundamentals，版本为 Android2.3 r1，翻译转载并整理自译言："biAji"，原文地址："http://article.yeeyan.org/view/37503/34036"，再次感谢"bjAji"！期待你一起参与翻译 Android 的相关资料，联系我 over140@gmail.com。

声明

本文档转载并整理自译言：Android 开发指南 1——应用程序基础

Android 中文翻译组：http://goo.gl/6vJQl

原文

http://developer.android.com/guide/topics/fundamentals.html

正文

应用程序基础(Application Fundamentals)

Android 应用程序使用 Java 做为开发语言。aapt 工具把编译后的 Java 代码连同其它应用程序需要的数据和资源文件一起打包到一个 Android 包文件中，这个文件使用.apk 做为扩展名，它是分发应用程序并安装到移动设备的媒介，用户只需下载并安装此文件到他们的设备。单一.apk 文件中的所有代码被认为是一个应用程序。

从很多方面来看，每个 Android 应用程序都存在于它自己的世界之中：

* 默认情况下，每个应用程序均运行于它自己的 Linux 进程中。当应用程序中的任意代码开始执行时，Android 启动一个进程，而当不再需要此进程而其它应用程序又需要系统资源时，则关闭这个进程。

✉每个进程都运行于自己的 Java 虚拟机（VM）中。所以应用程序代码实际上与其它应用程序的代码是隔绝的。

✉默认情况下，每个应用程序均被赋予一个唯一的 Linux 用户 ID，并加以权限设置，使得应用程序的文件仅对这个用户、这个应用程序可见。当然，也有其它的方法使得这些文件同样能为别的应用程序所访问。

使两个应用程序共有同一个用户 ID 是可行的，这种情况下他们可以看到彼此的文件。从系统资源维护的角度来看，拥有同一个 ID 的应用程序也将在运行时使用同一个 Linux 进程，以及同一个虚拟机。

应用程序组件(Application Components)

Android 的核心功能之一就是一个应用程序可以使用其它应用程序的元素（如果那个应用程序允许的话）。比如说，如果你的应用程序需要一个图片卷动列 表，而另一个应用程序已经开发了一个合用的而又允许别人使用的话，你可以直接调用那个卷动列表来完成工作，而不用自己再开发一个。你的应用程序并没有吸纳 或链接其它应用程序的代码，它只是在有需求的时候启动了其它应用程序的那个功能部分。

为达到这个目的，系统必须在一个应用程序的一部分被需要时启动这个应用程序，并将那个部分的 Java 对象实例化。与在其它系统上的应用程序不同，Android 应用程序没有为应用准备一个单独的程序入口（比如说，没有 main()方法）， 而是为系统依照需求实例化提供了基本的组件。共有四种组件类型：

Activities

- Activity 是为用户操作而展示的可视化用户界面。比如说，一个 activity 可以展示一个菜单项列表供用户选择，或者显示一些包含说明的照片。一个短消息应用程序可以包括一个用于显示做为发送对象的联系人的列表的 activity，一个给选定的联系人写短信的 activity 以及翻阅以前的短信和改变设置的 activity。尽管它们一起组成了一个内聚的用户界面，但其中每个 activity 都与其它的保持独立。每个都是以 Activity 类为基类的子类实现。
- 一个应用程序可以只有一个 activity，或者，如刚才提到的短信应用程序那样，包含很多个。每个 activity 的作用，以及其数目，自然取决于应用 程序及其设计。一般情况下，总有一个应用程序被标记为用户在应用程序启动的时候第一个看到的。从一个 activity 转向另一个的方式是靠当前的 activity 启动下一个。
- 每个 activity 都被给予一个默认的窗口以进行绘制。一般情况下，这个窗口是满屏的，但它也可以是一个小的位于其它窗口之上的浮动窗口。一个 activity 也可以使用超过一个的窗口——比如，在 activity 运行过程中弹出的一个供用户反应的小对话框，或是当用户选择了屏幕上特定项目后显 示的必要信息。

✉窗口显示的可视内容是由一系列视图构成的，这些视图均继承自 View 基类。每个视图均控制着窗口中一块特定的矩形空 间。父级视图包含并组织它子视图的布局。叶节点视图（位于视图层次最底端）在它们控制的矩形中进行绘制，并对用户对其直接操作做出响应。所以，视图是 activity 与用户进行交互的界面。比如说，视图可以显示一个小图片，并在用户指点它的时候产生动作。Android 有很多既定的视图供用户直接使 用，包括按钮、文本域、卷轴、菜单项、复选框等等。

✉视图层次是由 Activity.setContentView() 方法放入 activity 的窗口之中的。上下文视图是位于视图层次根位置的视图对象。（参见用户界面章节获取关于视图及层次的更多信息。）

服务(Services)

✉服务没有可视化的用户界面，而是在一段时间内在后台运行。比如说，一个服务可以在用户做其它事情的时候在后台播放背景音乐、从网络上获取一些数据或者计算一些东西并提供给需要这个运算结果的 activity 使用。每个服务都继承自 Service 基类。

✉一个媒体播放器播放播放列表中的曲目是一个不错的例子。播放器应用程序可能有一个或多个 activity 来给用户选择歌曲并进行播放。然而，音乐播放这个 任务本身不应该为任何 activity 所处理，因为用户期望在他们离开播放器应用程序而开始做别的事情时，音乐仍在继续播放。为达到这个目的，媒体播放器 activity 应该启用一个运行于后台的服务。而系统将在这个 activity 不再显示于屏幕之后，仍维持音乐播放服务的运行。

✉你可以连接至（绑定）一个正在运行的服务（如果服务没有运行，则启动之）。连接之后，你可以通过那个服务暴露出来的接口与服务进行通讯。对于音乐服务来说，这个接口可以允许用户暂停、回退、停止以及重新开始播放。

✉如同 activity 和其它组件一样，服务运行于应用程序进程的主线程内。所以它不会对其它组件或用户界面有任何干扰，它们一般会派生一个新线程来进行一些耗时任务（比如音乐回放）。参见下述 进程和线程(Processes and Threads)。

广播接收器(Broadcast receivers)

✉广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的——比如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播——比如说，通知其它应用程序一些数据下载完成并处于

可用状态。

✉应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的通知信息予以响应。所有的接收器均继承自 BroadcastReceiver 基类。

✉广播接收器没有用户界面。然而，它们可以启动一个 activity 来响应它们收到的信息，或者用 NotificationManager 来通知用户。通知可以用很多种方式来吸引用户的注意力——闪动背灯、震动、播放声音等等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。

内容提供者(Content providers)

✉内容提供者将一些特定的应用程序数据供给其它应用程序使用。数据可以存储于文件系统、SQLite 数据库或其它方式。内容提供者继承于 ContentProvider 基类，为其它应用程序取用和存储它管理的数据实现了一套标准方法。然而，应用程序并不直接调用这些方法，而是使用一个 ContentResolver 对象，调用它的方法作为替代。ContentResolver 可以与任意内容提供者进行会话，与其合作来对所有相关交互通讯进行管理。

✉参阅独立的内容提供者 Content Providers 章节获得更多关于使用内容提供者的内容。

每当出现一个需要被特定组件处理的请求时，Android 会确保那个组件的应用程序进程处于运行状态，或在必要的时候启动它。并确保那个相应组件的实例的存在，必要时会创建那个实例。

激活组件 Activating components: intents

当接收到 ContentResolver 发出的请求后，内容提供者被激活。而其它三种组件——activity、服务和广播接收器被一种叫做 intent 的异步消息所激活。intent 是一个保存着消息内容的 Intent 对 象。对于 activity 和服务来说，它指明了请求的操作名称以及作为操作对象的数据的 URI 和其它一些信息。比如说，它可以承载对一个 activity 的请求，让它为用户显示一张图片，或者让用户编辑一些文本。而对于广播接收器而言，Intent 对象指明了声明的行为。比如，它可以对所有感兴趣的对象声 明照相按钮被按下。

对于每种组件来说，激活的方法是不同的：

✉通过传递一个 Intent 对象至 Context.startActivity()或 Activity.startActivityForResult()以载入（或指定新工作给）一个 activity。相应的 activity 可以通过调用 getIntent() 方法来查看激活它的 intent。Android 通过调用

activity 的 onNewIntent()方法来传递给它继发的 intent。

一个 activity 经常启动了下一个。如果它期望它所启动的那个 activity 返回一个结果，它会以调用 startActivityForResult()来取代 startActivity()。比如说，如果它启动了另外一个 activity 以使用户挑选一张照片，它也许想知道哪张照片被选中了。结果将会被封装在一个 Intent 对象中，并传递给发出调用的 activity 的 onActivityResult() 方法。

✉通过传递一个 Intent 对象至 Context.startService()将启动一个服务（或给予正在运行的服务以一个新的指令）。Android 调用服务的 onStart()方法并将 Intent 对象传递给它。

与此类似，一个 Intent 可以被调用组件传递给 Context.bindService()以获取一个正在运行的目标服务的连接。这个服务会经由 onBind() 方法的调用获取这个 Intent 对象（如果服务尚未启动，bindService()会先启动它）。比如说，一个 activity 可以连接至前述的音乐回放服务，并提供给用户一个可操作的（用户界面）以对回放进行控制。这个 activity 可以调用 bindService()来建立连接，然后调用服务中定义的对象来影响回放。

后面一节：远程方法调用(Remote procedure calls)将更详细的阐明如何绑定至服务。

✉应用程序可以凭借将 Intent 对象传递给 Context.sendBroadcast() ，Context.sendOrderedBroadcast()， 以及 Context.sendStickyBroadcast()和其它类似方法来产生一个广播。Android 会调用所有对此广播有兴趣的广播接收器的 onReceive()方法，将 intent 传递给它们。

欲了解更多 intent 消息的信息，请参阅独立章节 Intent 和 Intent 滤过器(Intents and Intent Filters)。

关闭组件(Shutting down components)

内容提供者仅在响应 ContentResolver 提出请求的时候激活。而一个广播接收器仅在响应广播信息的时候激活。所以，没有必要去显式的关闭这些组件。

而 activity 则不同，它提供了用户界面，并与用户进行会话。所以只要会话依然持续，哪怕对话过程暂时停顿，它都会一直保持激活状态。与此相似，服务也会在很长一段时间内保持运行。所以 Android 为关闭 activity 和服务提供了一系列的方法。

✉可以通过调用它的 finish()方法来关闭一个 activity。一个 activity 可以通过调用另外一个 activity（它用 startActivityForResult()启动的）的 finishActivity()方法来关闭它。

✉服务可以通过调用它的 stopSelf()方法来停止，或者调用 Context.stopService()。

系统也会在组件不再被使用的时候或者 Android 需要为活动组件声明更多内存的时候关闭它。后面的组件的生命周期一节，将对这种可能及附属情况进行更详细的讨论。

manifest 文件(The manifest file)

当 Android 启动一个应用程序组件之前，它必须知道那个组件是存在的。所以，应用程序会在一个 manifest 文件中声明它的组件，这个文件会被打包到 Android 包中。这个.apk 文件还将涵括应用程序的代码、文件以及其它资源。

这个 manifest 文件以 XML 作为结构格式，而且对于所有应用程序，都叫做 AndroidManifest.xml。为声明一个应用程序组件，它还会 做很多额外工作，比如指明应用程序所需链接到的库的名称（除了默认的 Android 库之外）以及声明应用程序期望获得的各种权限。

但 manifest 文件的主要功能仍然是向 Android 声明应用程序的组件。举例说明，一个 activity 可以如下声明：

```
<?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 属性指定了实现了这个 activity 的 Activity 的子类。icon 和 label 属性指向了包含展示给用户的此 activity 的图标和标签的资源文件。

其它组件也以类似的方法声明——<service> 元素用于声明服务，<receiver> 元素用于

声明广播接收器，而 <provider> 元素用于声明内容提供者。 manifest 文件中未进行声明的 activity、服务以及内容提供者将不为系统所见，从而也就不会被运行。然而，广播接收器既可以在 manifest 文件中声明，也可以在代码中进行动态的创建，并以调用 Context.registerReceiver() 的方式注册至系统。

欲更多了解如何为你的应用程序构建 manifest 文件，请参阅 AndroidManifest.xml 文件一章。

Intent 过滤器(Intent filters)

Intent 对象可以被显式的指定目标组件。如果进行了这种指定，Android 会找到这个组件（依据 manifest 文件中的声明）并激活它。但如果 Intent 没有进行显式的指定，Android 就必须为它找到对于 intent 来说最合适的组件。这个过程是通过比较 Intent 对象和所有可能对象的 intent 过滤器完成的。组件的 intent 过滤器会告知 Android 它所能处理的 intent 类型。如同其它相对于组件很重要的信息一样，这些是在 manifest 文件中进行声明的。这里是上面实例的一个扩展，其中加入了针对 activity 的两个 intent 过滤器声明：

```
<?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>
```

示例中的第一个过滤器——action “android.intent.action.MAIN”和类别“android.intent.category.LAUNCHER”的组合——是通常具有的。它标明了这个 activity 将在应用程

序加载器中显示，就是用户在设备上看到的可供加载的应用程序列表。换句话说，这个 activity 是应用程序的入口，是用户选择运行这个应用程序后所见到的第一个 activity。

第二个过滤器声明了这个 activity 能被赋予一种特定类型的数据。

组件可以拥有任意数量的 intent 过滤器，每个都会声明一系列不同的能力。如果它没有包含任何过滤器，它将只能被显式声明了目标组件名称的 intent 激活。

对于在代码中创建并注册的广播接收器来说，intent 过滤器将被直接以 IntentFilter 对象实例化。其它过滤器则在 manifest 文件中设置。

欲获得更多 intent 过滤器的信息，请参阅独立章节： Intent 和 Intent 过滤器。

Activity 和任务(Activities and Tasks)

如前所述，一个 activity 可以启动另外一个，甚至包括与它不处于同一应用程序之中的。举个例子说，假设你想让用户看到某个地方的街道地图。而已经存 在一个具有此功能的 activity 了，那么你的 activity 所需要做的工作就是把请求信息放到一个 Intent 对象里面，并把它传递给 `startActivity()`。于是地图浏览器就会显示那个地图。而当用户按下 BACK 键的时候，你的 activity 又会再一次的显示在屏幕上。

对于用户来说，这看起来就像是地图浏览器是你 activity 所在的应用程序中的一个组成部分，其实它是在另外一个应用程序中定义，并运行在那个应用程序的进程之中的。Android 将这两个 activity 放在同一个任务中 来维持一个完整的用户体验。简单的说，任务就是用户所体验到的“应用程序”。它是安排在一个堆栈中的一组相关的 activity。堆栈中的根 activity 就是启动了这整个任务的那个——一般情况下，它就是用户在应用程序加载器中所选择的。而堆栈最上方的 activity 则是当前运行的—— 用户直接对其进行操作的。当一个 activity 启动另外一个的时候，新的 activity 就被压入堆栈，并成为当前运行的 activity。而前一个 activity 仍保持在堆栈之中。当用户按下 BACK 键的时候，当前 activity 出栈，而前一个恢复为当前运行的 activity。

堆栈中保存的其实是对象，所以如果发生了诸如需要多个地图浏览器的情况，就会使得一个任务中出现多个同一 Activity 子类的实例同时存在，堆栈会为每个实例单独开辟一个入口。堆栈中的 Activity 永远不会重排，只会压入或弹出。

任务其实就是 activity 的堆栈，而不是 manifest 文件中的一个类或者元素。所以你无法撇开 activity 而为一个任务设置一个值。而事实上 整个任务使用的值是在根 activity 中设置的。比如

说，下一节我们会谈及“任务的 affinity”，从 affinity 中读出的值将会设置到任务的 根 activity 之中。

任务中的所有 activity 是作为一个整体进行移动的。整个的任务（即 activity 堆栈）可以移到前台，或退至后台。举个例子说，比如当前任务在堆 栈中存有四个 activity——三个在当前 activity 之下。当用户按下 HOME 键的时候，回到了应用程序加载器，然后选择了一个新的应用程序（也 就是一个新*任务*）。则当前任务遁入后台，而新任务的根 activity 显示出来。然后，过了一小会儿，用户再次回到了应用程序加载器而又选择了前一个应用程序（上一个任务）。于是那个任务，带着它堆栈中所有的四个 activity，再一次的到了前台。*当用户*按下 BACK 键的时候，屏幕不会显示出用户刚才离开的 activity（上一个任务的根 activity）。取而代之，当前任务的堆栈中最上面的 activity 被弹出，而同一任务中的上一个 activity 显示了出来。

上述的种种即是 activity 和任务的默认行为模式。但是有一些方法可以改变所有这一切。activity 和任务的联系、任务中 activity 的行为 方式都被启动那个 activity 的 Intent 对象中设置的一系列标记和 manifest 文件中那个 activity 中的<activity>元素的系列属性之间的交互所控制。无论是请求发出者和回应者在这里都拥有话语权。

我们刚才所说的这些关键 Intent 标记如下：

FLAG_ACTIVITY_NEW_TASK

FLAG_ACTIVITY_CLEAR_TOP

FLAG_ACTIVITY_RESET_TASK_IF_NEEDED

FLAG_ACTIVITY_SINGLE_TOP

而关键的<activity>属性是：

taskAffinity

launchMode

allowTaskReparenting

clearTaskOnLaunch

alwaysRetainTaskState

finishOnTaskLaunch

接下来的一节会描述这些标记以及属性的作用，它们是如何互相影响的，以及控制它们的使用时必须考虑到的因素。

任务共用性和新任务 Affinities and new tasks

默认情况下，一个应用程序中的 activity 相互之间会有一种 Affinity——也就是说，它们首选都归属于一个任务。然而，可以在<activity>元素中把每个 activity 的 taskAffinity 属 性设置为一个独立的 affinity。于是在不同的应用程序中定义的 activity 可以享有同一个 affinity，或者在同一个应用程序中定义的 activity 有着不同的 affinity。affinity 在两种情况下生效：当加载 activity 的 Intent 对象包含了 FLAG_ACTIVITY_NEW_TASK 标记，或者当 activity 的 allowTaskReparenting 属性设置为“true”。

FLAG_ACTIVITY_NEW_TASK 标记

如前所述，在默认情况下，一个新 activity 被另外一个调用了 startActivity() 方法的 activity 载入了任务之中。并压入了调用者所在的堆栈。然而，如果传递给 startActivity() 的 Intent 对象包含了 FLAG_ACTIVITY_NEW_TASK 标记，系统会为新 activity 安排另外一个任务。一般情况下，如同标记所暗示的那样，这会是一个新任务。然而，这并不是必然的。如果已经存在了一个与新 activity 有着同样 affinity 的任务，则 activity 会载入那个任务之中。如果没有，则启用新任务。

allowTaskReparenting 属性

如果一个 activity 将 allowTaskReparenting 属 性设置为“true”。它就可以从初始的任务中转移到与其拥有同一个 affinity 并转向前台的任务之中。比如说，一个旅行应用程序中包含的预报所选城 市的天气情况的 activity。它与这个应用程序中其它的 activity 拥有同样的 affinity（默认的 affinity）而且允许重定父级。你 的另一个 activity 启动了天气预报，于是它就会与这个 activity 共处与同一任务之中。然而，当那个旅行应用程序再次回到前台的时候，这个天气 预报 activity 就会被再次安排到原先的任务之中并显示出来。

如果在用户的角度看来，一个.apk 文件中包含了多于一个的“应用程序”，你可能会想要为它们所辖的 activity 安排不一样的 affinity。

加载模式(Launch modes)

<activity>元素的 launchMode 属性可以设置四种不同的加载模式：

"standard" (默认模式)

"singleTop"

"singleTask"

"singleInstance"

这些模式之间的差异主要体现在四个方面：

✉**哪个任务会把持对 intent 做出响应的 activity**。对“standard”和“singleTop”模式而言，是产生 intent（并调用 startActivity()）的任务——除非 Intent 对象包含 FLAG_ACTIVITY_NEW_TASK 标记。而在这种情况下，如同上面 Affinitie 和新任务一节所述，会是另外一个任务。

相反，对“singleTask”和“singleInstance”模式而言，activity 总是位于任务的根部。正是它们定义了一个任务，所以它们绝不会被载入到其它任务之中。

✉**activity 是否可以存在多个实例。**一个“standard”或“singleTop”的 activity 可以被多次初始化。它们可以归属于多个任务，而一个任务也可以拥有同一 activity 的多个实例。

相反，对“singleTask”和“singleInstance”的 activity 被限定于只能有一个实例。因为这些 activity 都是任务的起源，这种限制意味着在一个设备中同一时间只允许存在一个任务的实例。

✉**在实例所在的任务中是否会有别的 activity。**一个“singleInstance”模式的 activity 将会是它所在的任务中唯一的 activity。如果它启动了别的 activity，那个 activity 将会依据它自己的加载模式加载到其它的任务中去——如同在 intent 中设置了 FLAG_ACTIVITY_NEW_TASK 标记一样的效果。在其它方面，“singleInstance”模式的效果与“singleTask”是一样的。

剩下的三种模式允许一个任务中出现多个 activity。“singleTask”模式的 activity 将是任务的根 activity，但它可以启动别的 activity 并将它们置入所在的任务中。“standard”和“singleTop”activity 则可以在堆栈的任意位置出现。

✉**是否要载入新的类实例以处理新的 intent。**对默认的"standard"模式来说，对于每个新 intent 都会创建一个新的实例以进行响应，每个实例仅处理一个 intent。“singleTop”模式下，如果 activity 位于目的任务堆栈的最上面，则重用目前现存的 activity 来处理新的 intent。如果它不是在堆栈顶部，则不会发生重用。而是创建一个新实例来处理新的 intent 并将其推入堆栈。

举例来说，假设一个任务的堆栈由根 activityA 和 activity B、C 和位于堆栈顶部的 D 组成，即堆栈 A-B-C-D。一个针对 D 类型的 activity 的 intent 抵达的时候，如果 D 是默认的“standard”加载模式，则创建并加载一个新的类实例，于是堆栈变为 A-B-C-D-D。 然而，如果 D 的载入模式为“singleTop”，则现有的实例会对新 intent 进行处理（因为它位于堆栈顶部）而堆栈保持 A-B-C-D 的形态。

换言之，如果新抵达的 intent 是针对 B 类型的 activity，则无论 B 的模式是“standard”还是“singleTop” ，都会加载一个新的 B 的实例（因为 B 不位于堆栈的顶部），而堆栈的顺序变为 A-B-C-D-B。

如前所述，“singleTask”或“singleInstance”模式的 activity 永远不会存在多于一个实例。所以实例将处理所有新的 intent。一个“singleInstance”模式的 activity 永远保持在堆栈的顶部（因为它是那个堆栈中唯一的一个 activity），所以它一直坚守在处理 intent 的岗位上。然而，对一个“singleTask”模式的 activity 来说，它上面可能有，也可能没有别的 activity 和它处于同一堆栈。在有的情况下，它就不在能够处理 intent 的位置上， 则那个 intent 将被舍弃。（即便在 intent 被舍弃的情况下，它的抵达仍将使这个任务切换至前台，并一直保留）

当一个现存的 activity 被要求处理一个新的 intent 的时候，会调用 onNewIntent() 方法来将 intent 对象传递至 activity。（启动 activity 的原始 intent 对象可以通过调用 getIntent() 方法获得。）

请注意，当一个新的 activity 实例被创建以处理新的 intent 的时候，用户总可以按下 BACK 键来回到前面的状态（回到前一个 activity）。但当使用现存的 activity 来处理新 intent 的时候，用户是不能靠按下 BACK 键回到当这个新 intent 抵达之前的状态 的。

想获得更多关于加载模式的内容，请参阅 <activity> 元素的描述。

清理堆栈(Clearing the stack)

如果用户离开一个任务很长一段时间，系统会清理该任务中除了根 activity 之外的所有 activity。当用户再次回到这个任务的时候，除了只剩下初 始化 activity 尚存之外，其余都跟用户上次离开它的时候一样。这样做的原因是：在一段时间之后，用户再次回到一个任务的时候，他们更期望放弃他们之 前的所作所为，做些新的事情。

这些属于默认行为，另外，也存在一些 activity 的属性用以控制并改变这些行为：

alwaysRetainTaskState 属性

如果一个任务的根 activity 中此属性设置为“true”，则上述默认行为不会发生。任务将在很长的一段时间内保留它堆栈内的所有 activity。

clearTaskOnLaunch 属性

如果一个任务的根 activity 中此属性设置为“true”，则每当用户离开这个任务和返回它的时候，堆栈都会被清空至只留下 rootactivity。换句话说，这是 alwaysRetainTaskState 的另一个极端。哪怕仅是过了一小会儿，用户回到任务时，也是见到它的初始状态。

finishOnTaskLaunch 属性

这个属性与 clearTaskOnLaunch 属性相似，但它仅作用于单个的 activity，而不是整个的 task。而且它可以使任意 activity 都被清理，甚至根 activity 也不例外。当它设置为“true”的时候，此 activity 仅做为任务的一部分存在于当前回话中，一旦用户离开并再次回到这个任务，此 activity 将不复存在。

此外，还有别的方式从堆栈中移除一个 activity。如果一个 intent 对象包含 FLAG_ACTIVITY_CLEAR_TOP 标记，而且目标任务的堆栈中已经存在了一个能够响应此 intent 的 activity 类型的实例。则这个实例之上的所有 activity 都将被清理以使它位于堆栈的顶部来对 intent 做出响应。如果此时指定的 activity 的加载模式为“standard”，则它本身也会从堆栈中移除，并加载一个新的实例来处理到来的 intent。这是因为加载模式为“standard”的 activity 总会创建一个新实例来处理新的 intent。

FLAG_ACTIVITY_CLEAR_TOP 与 FLAG_ACTIVITY_NEW_TASK 经常合并使用。这时，这些标记提供了一种定位其它任务中现存的 activity 并将它们置于可以对 intent 做出响应的位置的方法。

启动任务(Starting tasks)

当一个 activity 被指定一个“android.intent.action.MAIN”做为动作，以及“android.intent.category.LAUNCHER”做为类别的 intent 过滤器之后（在前述 intent 过滤器一节中已经有了这个示例），它就被设置为一个任务的入口点。这样的过滤器设置会在应用程序加载器中为此 activity 显示一个图标和标签，以供用户加载任务或加载之后在任意时间回到这个任务。

第二个能力相当重要：用户必须可以离开一个任务，并在一段时间后返回它。出于这个考虑，

加载模式被设定为“singleTask”和“singleInstance”的 activity 总是会初始化一个新任务，这样的 activity 仅能用于指定了一个 MAIN 和 LAUNCHER 过滤器的情况之下。我们来举例说明如果没指定过滤器的情况下会发生的事情：一个 intent 加载了一个“singleTask”的 activity，初始化了一个新任务，用户在这个任务中花费了一些时间来完成工作。然后用户按下了 HOME 键。于是任务被要求转至后台并被主屏幕所掩盖。因为它并没有在应用程序加载器中显示图标，这将导致用户无法再返回它。

类似的困境也可由 FLAG_ACTIVITY_NEW_TASK 标记引起。如果此标记使一个 activity 启动了一个新任务继而用户按下了 HOME 键离开了它，则用户必须要有一些方法再次回到这个任务。一些实体（诸如通知管理器）总是在另外的任务中启动新 activity，而不是做为它们自己的一部分，所以它们总是将 FLAG_ACTIVITY_NEW_TASK 标记包含在 intent 里面并传递给 startActivity()。如果你写了一个能被外部实体使用这个标记调用的 activity，你必须注意要给用户留一个返回这个被外部实体启动的任务的方法。

当你不想让用户再次返回一个 activity 的情况下，可以将 <activity> 元素的 finishOnTaskLaunch 设置为“true”。参见前述清理堆栈。

进程和线程(Processes and Threads)

当一个应用程序开始运行它的第一个组件时，Android 会为它启动一个 Linux 进程，并在其中执行一个单一的线程。默认情况下，应用程序所有的组件均在这个进程的这个线程中运行。

然而，你也可以安排组件在其他进程中运行，而且可以为任意进程衍生出其它线程。

进程(Processes)

组件运行所在的进程由 manifest 文件所控制。组件元素——<activity>，<service>，<receiver>和<provider>——都有一个 process 属性来指定组件应当运行于哪个进程之内。这些属性可以设置为使每个组件运行于它自己的进程之内，或一些组件共享一个进程而其余的组件不这么做。它们也可以 设置为令不同应用程序的组件在一个进程中运行——使应用程序的组成部分共享同一个 Linux 用户 ID 并赋以同样的权限。<application>元素也有一个 process 属性，以设定所有组件的默认值。

所有的组件实例都位于特定进程的主线程内，而对这些组件的系统调用也将由那个线程进行分发。一般不会为每个实例创建线程。因此，某些方法总是运行在进程的主线程内，这些方法包括诸

如 View.onKeyDown()这样报告用户动作以及后面 组件生命周期一节所要讨论的生命周期通告的。这意味着组件在被系统调用的时候，不应该施行长时间的抑或阻塞的操作（诸如网络相关操作或是循环计算），因为这将阻塞同样位于这个进程的其它组件的运行。你应该如同下面线程一节所叙述的那样，为这些长时间操作衍生出一个单独的线程进行处理。

在可用内存不足而又有一个正在为用户进行服务的进程需要更多内存的时候，Android 有时候可能会关闭一个进程。而在这个进程中运行着的应用程序也因此被销毁。当再次出现需要它们进行处理的工作的时候，会为这些组件重新创建进程。

在决定结束哪个进程的时候，Android 会衡量它们对于用户的相对重要性。比如说，相对于一个仍有用户可见的 activity 的进程，它更有可能去关闭 一个其 activity 已经不为用户所见的进程。也可以说，决定是否关闭一个进程主要依据在那个进程中运行的组件的状态。这些状态将在后续的一节组件生命周期中予以说明。

线程(Threads)

尽管你可以把你的应用程序限制于一个单独的进程中，有时，你仍然需要衍生出一个线程以处理后台任务。因为用户界面必须非常及时的对用户操作做出响应，所 以，控管 activity 的线程不应用于处理一些诸如网络下载之类的耗时操作。所有不能在瞬间完成的任务都应安排到不同的线程中去。

线程在代码中是以标准 Java Thread 对象创建的。Android 提供了很多便于管理线程的类：Looper 用于在一个线程中运行一个消息循环， Handler 用于处理消息，HandlerThread 用于使用一个消息循环启用一个线程。

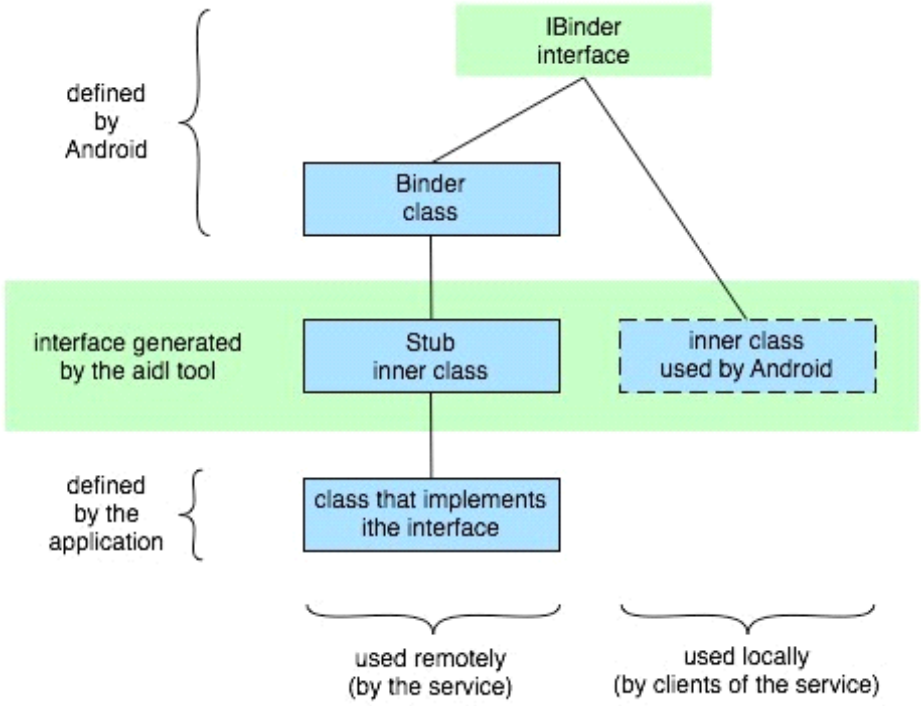
远程方法调用(Remote procedure calls)

Android 有一个轻量级的远程方法调用（RPC）机制：即在本地调用一个方法，但在远程（其它的进程中）进行处理，然后将结果返回调用者。这将方法调用及其附属的数据以系统可以理解的方式进行分离，并将其从本地进程和本地地址空间传送至远程过程和远程地址空间，并在那里重新装配并对调用做出反应。返回 的结果将以相反的方向进行传递。Android 提供了完成这些工作所需的所有的代码，以使你可以集中精力来实现 RPC 接口本身。

RPC 接口可以只包括方法。即便没有返回值，所有方法仍以同步的方式执行（本地方法阻

塞直至远程方法结束）。

简单的说，这套机制是这样工作的：一开始，你用简单的 IDL（界面描绘语言）声明一个你想要实现的 RPC 接口。然后用 aidl 工具为这个声明生成一个 Java 接口定义，这个定义必须对本地和远程进程都可见。它包含两个内部类，如下图所示：



内部类中有管理实现了你用 IDL 声明的接口的远程方法调用所需要的所有代码。两个内部类均实现了 IBinder 接口。一个用于系统在本地内部使用，你些的代码可以忽略它；另外一个，我们称为 Stub，扩展了 Binder 类。除了实现了 IPC 调用的内部代码之外，它还包括了你声明的 RPC 接口中的方法的声明。你应该如上图所示的那样写一个 Stub 的子类来实现这些方法。

一般情况下，远程过程是被一个服务所管理的（因为服务可以通知系统关于进程以及它连接到别的进程的信息）。它包含着 aidl 工具产生的接口文件和实现了 RPC 方法的 Stub 的子类。而客户端只需要包括 aidl 工具产生的接口文件。

下面将说明服务与其客户端之间的连接是如何建立的：

✉服务的客户端（位于本地）应该实现 onServiceConnected() 和 onServiceDisconnected() 方法。这样，当至远程服务的连接成功建立或者断开的时候，它们会收到通知。这样它们就可以调用 bindService() 来设置连接。

✉而服务则应该实现 onBind() 方法以接受或拒绝连接。这取决于它收到的 intent（intent

将传递给 bindService()）。如果接受了连接，它会返回一个 Stub 的子类的实例。

✉如果服务接受了连接，Android 将会调用客户端的 onServiceConnected() 方法，并传递给它一个 IBinder 对象，它是由服务所管理的 Stub 的子类的代理。通过这个代理，客户端可以对远程服务进行调用。

线程安全方法(Thread-safe methods)

在一些情况下，你所实现的方法有可能会被多于一个的线程所调用，所以它们必须被写成线程安全的。

对于我们上一节所讨论的 RPC 机制中的可以被远程调用的方法来说，这是必须首先考虑的。如果针对一个 IBinder 对象中实现的方法的调用源自这个 IBinder 对象所在的进程时，这个方法将会在调用者的线程中执行。然而，如果这个调用源自其它的进程，则这个方法将会在一个线程池中选出的线程中运 行，这个线程池由 Android 加以管理，并与 IBinder 存在于同一进程内；这个方法不会在进程的主线程内执行。反过来说，一个服务的 onBind()方法应为服务进程的主线程所调用，而实现了由 onBind()返回的对象（比如说，一个实现了 RPC 方法的 Stub 的子类）的方法将为池中的线程所调用。因为服务可以拥有多于一个的客户端，而同一时间，也会有多个池中的线程调用同一个 IBinder 方法。因此 IBinder 方法必须实现为线程安全的。

类似的，一个内容提供者能接受源自其它进程的请求数据。尽管 ContentResolver 和 ContentProvider 类隐藏了交互沟通过程的管理细节，ContentProvider 会由 query()，insert()， delete()， update()和 getType()方法来相应这些请求，而这些方法也都是由那个内容提供者的进程中所包涵的线程池提供的，而不是进程的主线程本身。所以这些有可能在同一时间被很多线程调用的方法也必须被实现为线程安全的。

组件生命周期(Component Lifecycles)

应用程序组件有其生命周期——由 Android 初始化它们以相应 intent 直到这个实例被摧毁。在此之间，它们有时是激活的有时则相反。或者，如果它是 一个 activity，则是可为用户所见或者不能。这一节讨论了 activity、服务以及广播接收器的生命周期，包括它们在生命周期中的状态、在状态之 间*转变*时通知你的方法、以及当这些进程被关闭或实例被摧毁时，这些状态产生的效果。

Activity 生命周期(Activity lifecycle)

一个 activity 主要有三个状态：

✉当在屏幕前台时（位于当前任务堆栈的顶部），它是*活跃*或*运行*的状态。它就是相应用户操作的 activity。

✉当它失去焦点但仍然对用户可见时，它处于*暂停*状态。即是：在它之上有另外一个 activity。这个 activity 也许是透明的，或者未能完全遮蔽全屏，所以被暂停的 activity 仍对用户可见。暂停的 activity 仍然是存活状态（它保留着所有的状态和成员信息并连接至窗口管理器），但当系统处于极低内存的情况下，仍然可以杀死这个 activity。

✉如果它完全被另一个 activity 覆盖是，它处于*停止*状态。它仍然保留所有的状态和成员信息。然而它不在为用户可见，所以它的窗口将被隐藏，如果其它地方需要内存，则系统经常会杀死这个 activity。

如果一个 activity 处于暂停或停止状态，系统可以通过要求它结束（调用它的 finish()方法）或直接杀死它的进程来将它驱出内存。当它再次为用户可见的时候，它只能完全重新启动并恢复至以前的状态。

当一个 activity 从这个状态转变到另一个状态时，它被以下列 protected 方法所通知：

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

你可以重载所有这些方法以在状态改变时进行合适的工作。所有的 activity 都必须实现 onCreate() 用以当对象第一次实例化时进行初始化设置。很多 activity 会实现 onPause()以提交数据变化或准备停止与用户的交互。

调用父类(Calling into the superclass)

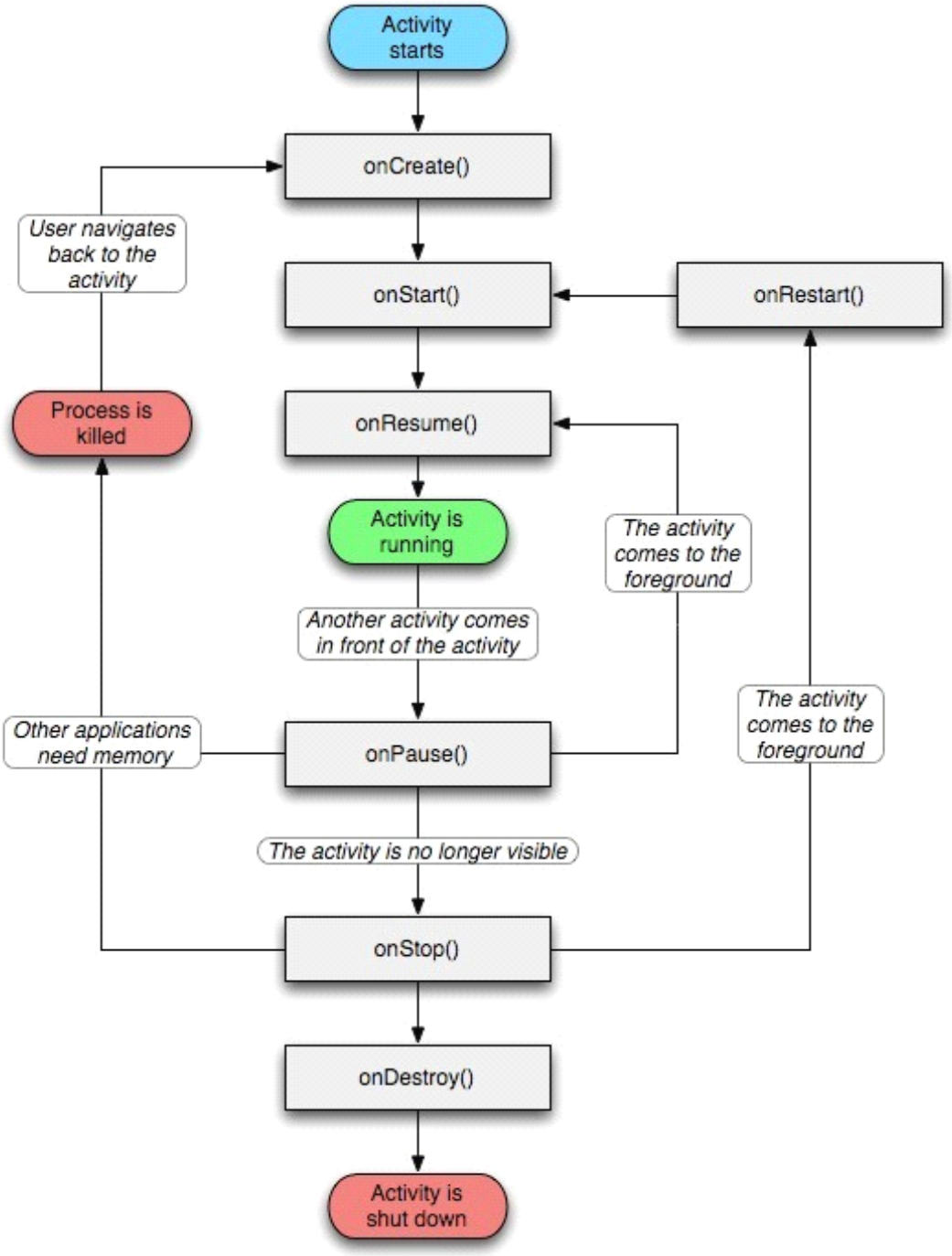
所有 activity 生命周期方法的实现都必须先调用其父类的版本。比如说：

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

总得来说，这七个方法定义了一个 activity 完整的生命周期。实现这些方法可以帮助你监察三个嵌套的生命周期循环：

- 一个 activity **完整的生命周期** 自第一次调用 onCreate()开始，直至调用 onDestroy()为止。activity 在 onCreate()中设置所有“全局”状态以完成初始化，而在 onDestroy()中释放所有系统资源。比如说，如果 activity 有一个线程在后台运行以从网络上下载数据，它会以 onCreate()创建那个线程，而以 onDestroy()销毁那个线程。
- 一个 activity 的 **可视生命周期**自 onStart()调用开始直到相应的 onStop()调用。在此期间，用户可以在屏幕上看到此 activity，尽管它也许并不是位于前台或者正在与用户做交互。在这两个方法中，你可以管控用来向用户显示这个 activity 的资源。比如说，你可以在 onStart()中注册一个 BroadcastReceiver 来监控会影响到你 UI 的改变，而在 onStop()中来取消注册，这时用户是无法看到你的程序显示的内容的。onStart()和 onStop()方法可以随着应用程序是否为用户可见而被多次调用。
- 一个 activity 的 **前台生命周期** 自 onResume()调用起，至相应的 onPause()调用为止。在此期间，activity 位于前台最上面并与用户进行交互。activity 会经常在暂停和恢复之间进行状态转换——比如说当设备转入休眠状态或有新的 activity 启动时，将调用 onPause()方法。当 activity 获得结果或者接收到新的 intent 的时候会调用 onResume()方法。因此，在这两个方法中的代码应当是轻量级的。

下图展示了上述循环过程以及 activity 在这个过程之中历经的状态改变。着色的椭圆是 activity 可以经历的主要状态。矩形框代表了当 activity 在状态间发生改变的时候，你进行操作所要实现的回调方法。



下表详细描述了这些方法，并在 activity 的整个生命周期中定位了它们。

方法	描述	是否可被杀死 (Killable?)	下一个
onCreate()	在 activity 第一次被创建的时候调用。这里是你做所有初始化设置的地方——创建视图、绑	否	onStart()

	定数据至列表等。如果曾经有状态记录（参阅后述 Saving Activity State。），则调用此方法时会传入一个包含着此 activity 以前状态的包对象做为参数。 接下来始终遵循调用 onStart()。		
onRestart()	在 activity 停止后，在再次启动之前被调用。 接下来始终遵循调用 onStart()。	否	onStart()
onStart()	当 activity 正要变得为用户所见时被调用。 当 activity 转向前台时接下来调用 onResume()，在 activity 变为隐藏时接下来调用 onStop()。	否	onResume() 或 onStop()
onResume()	在 activity 开始与用户进行交互之前被调用。此时 activity 位于堆栈顶部，并接受用户输入。 接下来始终遵循调用 onPause()。	否	onPause()
onPause()	当系统将要启动另一个 activity 时调用。此方法主要用来将未保存的变化进行持久化，停止类似动画这样耗费 CPU 的动作等。这一切动作应该在短时间内完成，因为下一个 activity 必须等到此方法返回后才会继续。 当 activity 重新回到前台时接下来调用 onResume()。当 activity 变为用户不可见时接下来调用 onStop()。	是	onResume() 或 onStop()
onStop()	当 activity 不再为用户可见时调用此方法。这可能发生在它被销毁或者另一个 activity（可能是现存的或者是新的）回到运行状态并覆盖了它。 如果 activity 再次回到前台跟用户交互则接下	是	onRestart() or onDestroy()

	来调用 onRestart()，如果关闭 activity 则接下来调用 onDestroy()。		
onDestroy()	在 activity 销毁前调用。这是 activity 接收的最后一个调用。这可能发生在 activity 结束（调用了它的 finish() 方法）或者因为系统需要空间所以临时的销毁了此 acitivity 的实例时。你可以用 isFinishing() 方法来区分这两种情况。	是	无

请注意上表中**可被杀死**一列。它标示了在方法返回后，还没执行 activity 的其余代码的任意时间里，系统是否可以杀死包含此 activity 的进程。三个方法（onPause()、 onStop()和 onDestroy()）被标记为“是”。onPause()是三个中的第一个，它也是唯一一个在进程被杀死之前必然会调用的方法——onStop() 和 onDestroy()有可能不被执行。因此你应该用 onPause()来将所有持久性数据（比如用户的编辑结果）写入存储之中。

在**可被杀死**一列中标记为“否”的方法在它们被调用时将保护 activity 所在的进程不会被杀死。所以只有在 onPause()方法返回后到 onResume()方法被调用时，一个 activity 才处于可被杀死的状态。在 onPause()再次被调用并返回之前，它不会被系统杀死。

如后面一节进程和生命周期所述，即使是在这里技术上没有被定义为“可杀死”的 activity 仍然有可能被系统杀死——但这仅会发生在实在没有其它方法的极端情况之下。

保存 activity 状态(Saving activity state)

当系统而不是用户自己出于回收内存的考虑，关闭了一个 activity 之后。用户会期望当他再次回到那个 activity 的时候，它仍保持着上次离开时的样子。

为了获取 activity 被杀死前的状态，你应该为 activity 实现 onSaveInstanceState() 方法。Android 在 activity 有可能被销毁之前（即 onPause()调用之前）会调用此方法。它会将一个以名称-值对方式记录了 activity 动态状态的 Bundle 对象传递给该方法。当 activity 再次启动时，这个 Bundle 会传递给 onCreate()方法和随着 onStart()方法调用的 onRestoreInstanceState()，所以它们两个都可以恢复捕获的状态。

与 onPause()或先前讨论的其它方法不同，onSaveInstanceState() 和

onRestoreInstanceState()并不是生命周期方法。它们并不是总会被调用。比如说，Android 会在 activity 易于被系统销毁之前调用 onSaveInstanceState()，但用户动作（比如按下了 BACK 键）造成的销毁则不调用。在这种情况下，用户没打算再次回到这个 activity，所以没有保存状态的必要。

因为 onSaveInstanceState()不是总被调用，所以你应该只用它来为 activity 保存一些临时的状态，而不能用来保存持久性数据。而是应该用 onPause()来达到这个目的。

服务生命周期(Coordinating activities)

服务以两种方式使用：

- ✉它可以启动并运行，直至有人停止了它或它自己停止。在这种方式下，它以调用 Context.startService()启动，而以调用 Context.stopService()结束。它可以调用 Service.stopSelf() 或 Service.stopSelfResult()来自己停止。不论调用了多少次 startService()方法，你只需要调用一次 stopService()来停止服务。
- ✉它可以通过自己定义并暴露出来的接口进行程序操作。客户端建立一个到服务对象的连接，并通过那个连接来调用服务。连接以调用 Context.bindService()方法建立，以调用 Context.unbindService()关闭。多个客户端可以绑定至同一个服务。如果服务此时还没有加载，bindService()会先加载它。

这两种模式并不是完全分离的。你可以绑定至一个用 startService()启动的服务。比如说，一个后台音乐播放服务可以调用 startService()并传递给它一个包含欲播放的音乐列表的 Intent 对象来启动。不久，当用户想要对播放器进行控制或者查看当前播放曲目的详情时，会启用一个 activity，调用 bindService()连接到服务来完成操作。在这种情况下，直到绑定连接关闭 stopService()才会真正停止一个服务。

与 activity 一样，服务也有一系列你可以实现以用于监控其状态变化的生命周期方法。但相对于 activity 要少一些，只有三个，而且，它们是 public 属性，并非 protected：

```
void onCreate()

void onStart(Intent intent)

void onDestroy()
```

倚仗实现这些方法，你监控服务的两个嵌套的生命周期循环：

✉服务的**完整生命周期**始于调用 onCreate()而终于 onDestroy()方法返回。如同 activity 一样，服务在 onCreate()里面进行它自己的初始化，而在 onDestroy()里面释放所有资源。比如说，一个音乐回放服务可以在 onCreate()中创建播放音乐的线程， 而在 onDestroy()中停止这个线程。

✉服务的**活跃生命周期**始于调用 onStart()。这个方法用于处理传递给 startService()的 Intent 对象。音乐服务会打开 Intent 来探明将要播放哪首音乐，并开始播放。

服务停止时没有相应的回调方法——不存在 onStop()方法。

onCreate()和 onDestroy()方法在所有服务中都会被调用，无论它们是由 Context.startService()还是由 Context.bindService()所启动的。而 onStart()仅会被 startService()所启用的服务调用。

如果一个服务允许别的进程绑定，则它还会有以下额外的回调方法以供实现：

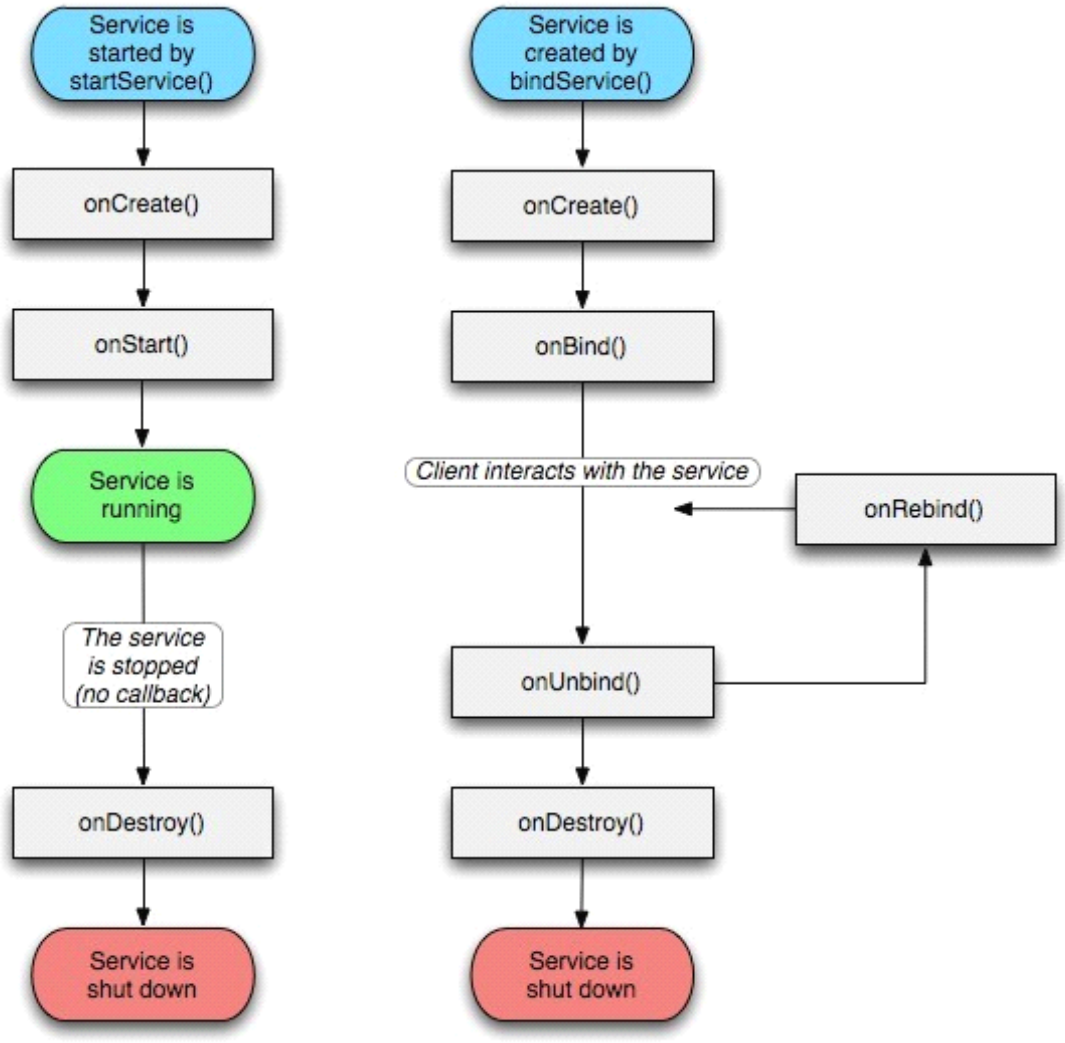
```
IBinder onBind(Intent intent)

boolean onUnbind(Intent intent)

void onRebind(Intent intent)
```

传递给 bindService 的 Intent 的对象也会传递给 onBind()回调方法，而传递给 unbindService()的 Intent 对象同样传递给 onUnbind()。如果服务允许绑定，onBind()将返回一个供客户端与服务进行交互的通讯渠道。如果有新的客户端连接至服务，则 onUnbind()方法可以要求调用 onRebind() 。

下图描绘了服务的回调方法。尽管图中对由 startService 和 startService 方法启动的服务做了区分，但要记住，不论一个服务是怎么启动的，它都可能允许客户端的连接，所以任何服务都可以接受 onBind()和 onUnbind()调用。



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

广播接收器只有一个回调方法：

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

当广播消息抵达接收器时，Android 调用它的 onReceive() 方法并将包含消息的 Intent 对象传递给它。广播接收器仅在它执行这个方法时处于活跃状态。当 onReceive()返回后，它即为失活状态。

拥有一个活跃状态的广播接收器的进程被保护起来而不会被杀死。但仅拥有失活状态组件的进程则会在其它进程需要它所占有的内存的时候随时被杀掉。

这种方式引出了一个问题：如果响应一个广播信息需要很长的一段时间，我们一般会将其纳

入一个衍生的线程中去完成，而不是在主线程内完成它，从而保证用户交互过程的流畅。如果onReceive()衍生了一个线程并且返回，则包涵新线程在内的整个进程都被会判为失活状态（除非进程内的其它应用程序组件仍处于活跃状态），于是它就有可能被杀掉。这个问题的解决方法是令onReceive()启动一个新服务，并用其完成任务，于是系统就会知道进程中仍然在处理着工作。

下一节中，我们会讨论更多进程易误杀的问题。

进程与生命周期(Processes and lifecycles)

Android 系统会尽可能长的延续一个应用程序进程，但在内存过低的时候，仍然会不可避免需要移除旧的进程。为决定保留或移除一个进程，Android 将每个进程都放入一个“重要性层次”中，依据则是它其中运行着的组件及其状态。重要性最低的进程首先被消灭，然后是较低的，依此类推。重要性共分五层，依 据重要性列表如下：

1. **前台进程**是用户操作所必须的。当满足如下任一条件时，进程被认为是处于前台的：

   ✉它运行着正在与用户交互的 activity（Activity 对象的 onResume() 方法已被调用）。

   ✉一个正在与用户交互的 activity 使用着它提供的一个服务。

   ✉它包含着一个正在执行生命周期回调方法（onCreate()、onStart()或 onDestroy()）的 Service 对象。

   ✉它包含着一个正在执行 onReceive() 方法的 BroadcastReceiver 对象。

   任一时间下，仅有少数进程会处于前台，仅当内存实在无法供给它们维持同时运行时才会被杀死。一般来说，在这种情况下，设备已然处于使用虚拟内存的状态，必须要杀死一些前台进程以用户界面保持响应。

2. **可视进程**没有前台组件，但仍可被用户在屏幕上所见。当满足如下任一条件时，进程被认为是可视的：

   ✉它包含着一个不在前台，但仍然为用户可见的 activity（它的 onPause()方法被调用）。这种情况可能出现在以下情况：比如说，前台 activity 是一个对话框，而之前的 activity 位于其下并可以看到。

   ✉它包含了一个绑定至一个可视的 activity 的服务。

   可视进程依然被视为是很重要的，非到不杀死它们便无法维持前台进程运行时，才会被杀

死。

3. **服务进程**是由 startService() 方法启动的服务，它不会变成上述两类。尽管服务进程不会直接为用户所见，但它们一般都在做着用户所关心的事情（比如在后台播放 mp3或者从网上下载东西）。所以系统会尽量维持它们的运行，除非系统内存不足以维持前台进程和可视进程的运行需要。

4. **背景进程**包含目前不为用户所见的 activity（Activity 对象的 onStop() 方法已被调用）。这些进程与用户体验没有直接的联系，可以在任意时间被杀死以回收内存供前台进程、可视进程以及服务进程使用。一般来说，会有很多背景进程 运行，所以它们一般存放于一个 LRU（最后使用）列表中以确保最后被用户使用的 activity 最后被杀死。如果一个 activity 正确的实现了生命周 期方法，并捕获了正确的状态，则杀死它的进程对用户体验不会有任何不良影响。

5. **空进程**不包含任何活动应用程序组件。这种进程存在的唯一原因是做为缓存以改善组件再次于其中运行时的启动时间。系统经常会杀死这种进程以保持进程缓存和系统内核缓存之间的平衡。

Android 会依据进程中当前活跃组件的重要程度来尽可能高的估量一个进程的级别。比如说，如果一个进程中同时有一个服务和一个可视的 activity，则进程会被判定为可视进程，而不是服务进程。

此外，一个进程的级别可能会由于其它进程依赖于它而升高。一个为其它进程提供服务的进程级别永远高于使用它服务的进程。比如说，如果 A 进程中的内容提供者 为进程 B 中的客户端提供服务，或进程 A 中的服务为进程 B 中的组件所绑定，则 A 进程最低也会被视为与进程 B 拥有同样的重要性。

为运行着一个服务的进程重要级别总高于一个背景 activity。所以一个 activity 以启动一个服务的方式启动一个长时间运行过程比简单的衍生一个 线程来进行处理要好。尤其是当处理过程比 activity 本身存在时间要长的情况之下。我们以背景音乐播放和上传一个相机拍摄的图片至网站上为例。使用服 务则不论 activity 发生何事，都至少可以保证操作拥有“服务进程”的权限。如上一节广播接收器生命周期所提到的，这也正是广播接收器使用服务，而不是使用线程来处理耗时任务的原因。