

# Android 开发指南中文版

---

-应用程序框架

android  
developers

iefreer@hotmail.com

2009/9/10

个人主页:

<http://blog.csdn.net/iefreer>

Pdf 制作 cattong(<http://www.80hou.org.cn>)



本文是对 Android SDK1.5 版的英文开发资料 Android Development Guide 一文应用程序框架部分的翻译，覆盖了 Android 应用开发所有主要的概念。部分内容整理自网络。本文仅用于技

术学习，请勿用于商业用途。

目录

- 应用程序基础 **Application Fundamentals**..... 4
  - 关键类..... 4
  - 应用程序组件..... 5
    - 激活组件: intent.....7
    - 关闭组件..... 7
    - manifest 文件.....8
    - Intent 过滤器.....8
  - Activity 和任务..... 10
    - Affinity（吸引力）和新任务..... 11
    - 加载模式..... 12
    - 清理堆栈..... 13
    - 启动任务..... 14
  - 进程和线程..... 14
    - 进程..... 15
    - 线程..... 15

远程过程调用.....	15
线程安全方法.....	16
组件生命周期.....	17
Activity 生命周期.....	17
调用父类.....	18
服务生命周期.....	21
广播接收器生命周期.....	22
进程与生命周期.....	22
<b>用户界面 User Interface</b> .....	24
视图层次 <b>View Hierarchy</b> .....	24
布局 <b>Layout</b> .....	25
部件 <b>Widgets</b> .....	26
用户界面事件 <b>UI Events</b> .....	26
菜单 <b>Menus</b> .....	26
<b>高级话题 Advanced Topics</b> .....	27
适配器 <b>Adapter</b> .....	27
风格与主题 <b>Styles and Themes</b> .....	27

资源和资产 <b>Resources and Assets</b> .....	28
资源引用 <b>Resource Reference</b> .....	43
国际化和本地化 <b>Internationalization and Localization</b> .....	43
意图和意图过滤器 <b>Intents and Intent Filters</b> .....	43
意图过滤器 <b>Intent filters</b> .....	47
通常情况 <b>Common cases</b> .....	51
使用意图匹配 <b>Using intent matching</b> .....	52
数据存储 <b>Data Storage</b> .....	52
概览 <b>Storage quickview</b> .....	52
◇ 系统偏好：快速，轻量级存储.....	52
◇ 文件：存储到设备内部或可移动闪存.....	52
◇ 数据库：任意的结构化存储.....	52
◇ 支持基于网络的存储.....	52
系统偏好 <b>Preferences</b> .....	53
文件 <b>Files</b> .....	54
数据库 <b>Databases</b> .....	54
网络 <b>Network</b> .....	55

内容提供者 <b>Content Providers</b> .....	55
内容提供器的基础知识 <b>Content Provider Basics</b> .....	55
查询一个内容提供者 <b>Querying a Content Provider</b> .....	57
修改数据 <b>Modifying Data</b> .....	61
创建一个内容提供者 <b>Creating a Content Provider</b> .....	64
<b>Content URI</b> 总结.....	67
清单文件 <b>The AndroidManifest.xml File</b> .....	68
清单文件结构 <b>Structure of the Manifest File</b> .....	68
文件约定 <b>File Conventions</b> .....	70
文件特性 <b>File Features</b> .....	73

## 应用程序基础 **Application Fundamentals**

### 关键类

1. [Activity](#)
2. [Service](#)
3. [BroadcastReceiver](#)
4. [ContentProvider](#)
5. [Intent](#)

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

**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滚动列表，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滚动列表来完成工作，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：

### Activity

*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

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](#)。

## 广播接收器

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

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](#)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：intent

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](#)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 过滤器](#)。

## 关闭组件

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

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` 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:type="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

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。

## **Affinity（吸引力）和新任务**

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。

如果在用户的角度来看，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

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

"[standard](#)"（默认值）

"[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

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对话中，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

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

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

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。一般不会对每个实例创建线程。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

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。

## 远程过程调用

**Android** 有一个轻量级的远程过程调用 (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，它还包括了你所声明的 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

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

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。在此期间，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

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

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();  
  
    ...  
  
}
```

总的来说，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	可被杀死	下一个
<a href="#">onCreate()</a>	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、绑定数据至列表等。如果曾经有状态记录（参阅后述 <a href="#">Saving Activity State</a> 。），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。  总继之以 <a href="#">onStart()</a> 。	否	<a href="#">onStart()</a>
<a href="#">onRestart()</a>	在 activity 停止后，在再次启动之前被调用。  总继之以 <a href="#">onStart()</a> 。	否	<a href="#">onStart()</a>
<a href="#">onStart()</a>	当 activity 正要变得为用户所见时被调用。  当 activity 转向前台时继以 <a href="#">onResume()</a> ，在 activity	否	<a href="#">onResume()</a> or

方法	描述	可 被 杀 死	下一个
	变为隐藏时继以 <a href="#">onStop()</a> 。		<a href="#">onStop()</a>
<a href="#">onResume()</a>	在 <b>activity</b> 开始与用户进行交互之前被调用。此时 <b>activity</b> 位于堆栈顶部，并接受用户输入。  继之以 <a href="#">onPause()</a> 。	否	<a href="#">onPause()</a>
<a href="#">onPause()</a>	当系统将要启动另一个 <b>activity</b> 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 <b>activity</b> 必须等到此方法返回后才会继续。  当 <b>activity</b> 重新回到前台是继以 <a href="#">onResume()</a> 。当 <b>activity</b> 变为用户不可见时继以 <a href="#">onStop()</a> 。	是	<a href="#">onResume()</a> or <a href="#">onStop()</a>
<a href="#">onStop()</a>	当 <b>activity</b> 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 <b>activity</b> （可能是现存的或者是新的）回到运行状态并覆盖了它。  如果 <b>activity</b> 再次回到前台跟用户交互则继以 <a href="#">onRestart()</a> ，如果关闭 <b>activity</b> 则继以 <a href="#">onDestroy()</a> 。	是	<a href="#">onRestart()</a> or <a href="#">onDestroy()</a>
<a href="#">onDestroy()</a>	在 <b>activity</b> 销毁前调用。这是 <b>activity</b> 接收的最后一个调用。这可能发生在 <b>activity</b> 结束（调用了它的 <a href="#">finish()</a> 方法）或者因为系统需要空间所以临时的销毁了此 <b>activity</b> 的实例时。你可以用 <a href="#">isFinishing()</a> 方法来区分这两种情况。	是	<i>nothing</i>

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

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。

## 协调 **activity**

当一个 **activity** 启动了另外一个的时候，它们都会经历生命周期变化。一个会暂停乃至停止，而另一个则启动。这种情况下，你可能需要协调好这些 **activity**：

生命周期回调顺序是已经定义好的，尤其是在两个 **activity** 在同一个进程内的情况下：

1. 调用当前 **activity** 的 [onPause\(\)](#) 方法。

2. 接着，顺序调用新启动 **activity** 的 [onCreate\(\)](#)、[onStart\(\)](#)和 [onResume\(\)](#)方法。
3. 然后，如果启动的 **activity** 不再于屏幕上可见，则调用它的 [onStop\(\)](#)方法。

## 服务生命周期

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，不论它们是由 [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

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，则包含新线程在内的整个进程都会被判为失活状态（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

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进程为 **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。

因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。

## 用户界面 User Interface

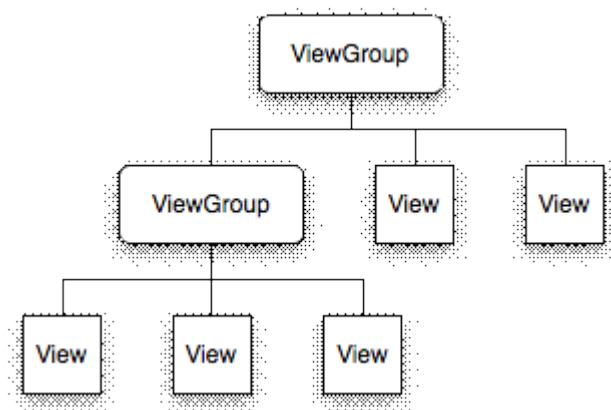
在一个 **Android** 应用中，用户界面是由 [View](#) 和 [ViewGroup](#) 对象构建的。**View** 与 **ViewGroup** 都有很多种类，而它们都是 [View](#) 类的子类。

**View** 对象是 **Android** 平台中用户界面体现的基础单位。**View** 类是它称为“widgets (工具)”的子类的基础，它们提供了诸如文本输入框和按钮之类的 **UI** 对象的完整实现。**ViewGroup** 类同样为其被称为“**Layouts** (布局)”的子类奠定了基础，它们提供了象流式布局、表格布局以及相对布局之类的布局架构。

**View** 对象是一个数据体，它的属性存储了用于屏幕上一块矩形区域的布局参数及内容。并负责这块它所辖的这个矩形区域之中所有测量、布局、焦点转换、滚动以及按键/触摸手势的处理。作为一个用户界面对象，**View** 同时也担任着用户交互关键点以及交互事件接受者的角色。

## 视图层次 View Hierarchy

在 **Android** 平台上，你可以用下图所示的 **View** 和 **ViewGroup** 层次图来定义一个 **Activity** 的 **UI**。这个层次树可随你所愿的简单或者复杂化，你能使用 **Android** 预定义的一套工具和布局来创建它，或者使用你自己定义的 **Views** 来创建。



为了把一个视图层次树展现到屏幕上，你的 **Activity** 必须调用 [setContentView\(\)](#) 方法，并传给它一个根节点对象的引用。**Android** 系统将接受此引用，并用来进行界面的废止、测量并绘制这棵树。层次的根结点会要求它的子节点进行自我绘制——进而，每个视图组节点也负责调用它的子视图进行自我绘制。子节点将向父节点申请绘制的位置以及大小，而其父类享有子节点绘制的位置及大小的最终决定权。**Android** 依次（自层次树顶层开始）解析你布局中的元素，实例化 **View** 并将它们添加到它们的父节点中。因为这个过程是依次进行的，所以如果出现了元素重叠的情况，最后一个绘制的元素将位于所有重叠元素之上显现。

如欲获得更多关于视图层次如何测算以及绘制细节的讨论，请参阅 [Android 如何绘制视图](#)。

## 布局 Layout

定义并展现你的视图层次的最常用的方法是使用 XML 布局文件。如同 HTML 一样，XML 为布局提供了一种可读的结构。XML 中的每个元素都是 **View** 或 **ViewGroup** 对象（抑或它们的子类）。**View** 对象是树的叶节点，而 **ViewGroup** 对象是树的分支（参阅楼上的视图层次图）。

XML 元素的名称与它体现的 Java 类相对应。所以一个 `<TextView>` 元素将在你的 UI 中生成一个 [TextView](#)，而 `<LinearLayout>` 则创建一个 [LinearLayout](#) 视图组。当你载入一个布局资源时，**Android** 系统会根据你布局中的元素初始化这些运行时对象。

举例来说，一个包含文本视图和一个按钮的简单垂直布局如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

请注意：LinearLayout 元素包含了 TextView 和 Button 对象。你可以在其中另外安置一个 LinearLayout（或其它类型的视图组），以延展这个视图层次，构建更复杂的布局。

欲获知更多如何构建 UI 布局的内容，请参阅[声明布局](#)。

**提示：**您也可以使用 Java 代码来绘制 View 和 ViewGroup 对象，并用 [addView\(View\)](#) 方法动态的插入新的 View 和 ViewGroup 对象。

您有相当多的方法来对视图进行布局。使用大量不同种类的视图组，您可以有近乎无穷的方式来构建子视图和视图组。Android 提供了一些预定义的视图组，其中包括 LinearLayout，RelativeLayout，AbsoluteLayout，TableLayout，GridLayout 以及其它的一些。每个都为定义子视图和布局结构提供了一套独特的布局参数。

欲了解其它用于布局的不同种类的视图组，请参阅[普通布局对象](#)。

## 部件 Widgets

部件是为用户交互界面提供服务的视图对象。Android 提供了一套完整的部件实现，包括按钮、复选框、文本输入框等，以助于你快速的构建 UI。Android 还提供了一些更高级的部件，比如日期选择、时钟以及缩放控制。但您并没有被局限于 Android 平台提供的这些部件上。如果您想创建一些您自己的定制动作元素，您可以这么做，只要定义自己的视图对象或者扩展或合并现有的部件就行。

更多信息参阅[构建自定义组件](#)。

您可以在 [android.widget](#) 包中找到 Android 提供的部件列表。

## 用户界面事件 UI Events

当你在用户界面中加入了一些视图和工具之后，你可能想要知道如何让它们与用户交互，进而实现你的动作。如欲获得用户界面事件通知，你需要做以下两件事情之一：

- **定义一个事件侦听器并将其注册至视图。**通常情况下，这是你侦听事件的主要方式。View 类包含了一大堆命名类似 On<什么什么>Listener 的接口，每个都带有一个叫做 On<什么什么>() 的回调方法。比如：[View.OnClickListener](#)（用以处理视图中的点击），[View.OnTouchListener](#)（用以处理视图中的触屏事件），以及 [View.OnKeyListener](#)（用以处理视图中的设备按键事件）。所以，如果你希望你的视图在它被“点击”（比如选择了一个按钮）的时候获得通知，你就要实现 OnClickListener，定义它的 onClick() 回调方法（在其中进行相应处理），并将它用 [setOnClickListener\(\)](#) 方法注册到视图上。

- **为视图覆写一个现有的回调方法。**这种方法主要用于你自己实现了一个 **View** 类，并想侦听其上发生的特定事件。比如说当屏幕被触摸（[onTouchEvent\(\)](#)），当轨迹球发生了移动（[onTrackballEvent\(\)](#)）或者是设备上的按键被按下（[onKeyDown\(\)](#)）。这种方式允许你为自己定制的视图中发生的每个事件定义默认的行为，并决定是否需要将事件传递给其它的子视图。再说一次，这些是 **View** 类相关的回调方法，所以你只能在你[构建自定义组件](#)时定义它们。

如何在视图中处理用户交互请参见[处理用户界面事件](#)文档。

## 菜单 Menus

应用程序菜单是应用程序用户界面中另外一个重要的组成部分。菜单为展现应用程序功能和设置提供了一个可靠的界面。按下设备上的 **MENU** 键会调出最普通的应用程序菜单。然而，你也可以加入当用户长按一个项目时调出的上下文菜单。

菜单也是用视图层次进行构架的，但你不必自己定义这个架构。你只要为你的 **Activity** 定义 [onCreateOptionsMenu\(\)](#) 和 [onCreateContextMenu\(\)](#) 回调方法，并声明你想要包含在菜单中的项目就行了。**Android** 将为你的菜单自动创建视图层次，并在其中绘入你的菜单项。

菜单会自行处理它们的事件，所以你不必为你菜单中的项目注册事件侦听器。当你菜单中的一项被选定，框架将自动调用 [onOptionsItemSelected\(\)](#) 或 [onContextItemSelected\(\)](#) 方法。

如同应用程序布局一样。你也可以在一个 **XML** 文件中定义你菜单中的项目。

更多信息，请参阅[创建菜单](#)。

## 高级话题 Advanced Topics

一旦你对创建用户界面的基础了如指掌，你就可以尝试着用一些高级功能来创建更加复杂的应用程序界面。

### 适配器 Adapter

有时候你会想要用一些无法硬编码的信息来填充视图组。你想将源于外部的数据绑定到你的视图中。为达到这个目的，你可以使用 **AdapterView** 作为你的视图组，并用 **Adapter** 传来的数据初始化每个子视图并填入其中。

**AdapterView** 对象是一个用给定的 **Adapter** 对象为基础构建它的子视图的 **ViewGroup** 实现。而 **Adapter** 在你的数据源（可能是一个外部字符串数组）和显示这些数据的 **AdapterView** 之间扮演着一个信使的角色。针对特定的任务有着很多不同的 **Adapter** 类实现，比如 **CursorAdapter** 依据 **Cursor** 读出一个数据库的数据，而一个 **ArrayAdapter** 则从任一数组进行读取。

想要了解如何运用 **Adapter** 填充你的视图，请参见[用 AdapterView 绑定至数据](#)。

### 风格与主题 Styles and Themes

或许你对标准工具的外表不是那么满意。为了解决这个问题，你可以创建你自己的风格和主题。

- 风格是一套包含一个或多个格式化属性的整体，你可以把它们加诸于你布局中的单个元素之上。比如，你可以定义一个包含特定文本字体大小和颜色的风格，并将它单独施用于特定的视图元素。
- 主题也是一套包含一个或多个格式化属性的整体，但却应用于一个应用程序中的所有 **Activity**，或单独一个 **Activity**。比如说，你可以定义一个包含了特定窗口边框颜色和版面背景、以及一套字体大小和菜单颜色的主题。这个主题可以施用于特定的 **Activity** 抑或整个应用程序。

风格与主题隶属于资源。**Android** 提供了一些默认的风格和主题供你使用，你也可以定制你自己的风格和主题资源。

想了解更多关于使用风格和主题的内容，请参阅[使用风格和主题](#)文档。

## 资源和资产 **Resources and Assets**

资源是 **Android** 应用程序不可或缺的部分。总体而言，资源是你想包含和引入到应用程序里面的一些外部元素，比如图片、音频、视频、文本字符串、布局、主题等。每个 **Android** 应用程序包含一个资源目录（**res/**）和资产目录（**assets/**），资产不经常被使用，因为它们的应用程序很少。你仅在需要读取原始字节流时才需要保存数据为资产。资源和资产目录均驻留在 **Android** 项目树的顶端，和源代码目录（**src/**）处在同一级上。

资源和资产从表面上看没多大区别，不过总体上，在存储外部内容时资源用得更多。真正的区别在于任何放置在资源目录里的内容可以通过您的应用程序的 **R** 类访问，这是被 **Android** 编译过的。而任何存放在资产目录里的内容会保持它的原始文件格式，为了读取它，你必须使用 [AssetManager](#) 来以字节流的方式读取文件。所以保持文件和数据在资源中（**res/**）中会更方便访问。

在这篇文章中，你将获取关于 **Android** 应用程序经常使用的标准资源类型以及如何在代码中引用方面的信息。资源和国际化（[Resources and Internationalization](#)）是第一步，可以知道 **Android** 如何利用项目资源。然后，可用资源类型（[Available Resource Types](#)）汇总描述了各种资源类型及其规格引用。

## 资源和国际化 **Resources and Internationalization**

资源是外部文件（即非源代码文件），它们被你的代码使用，并且在编译时被编译到你的应用程序中。**Android** 支持很多不同类型的资源文件，包括 **XML**、**PNG** 和 **JPEG** 文件。**XML** 文件会由于其所描述的内容不同而形式不同。该文档描述了所有支持的文件类型及每种类型的语法或格式。

资源从源代码中被抽取出来，基于效率考虑，XML 文件被编译成二进制、可以快速加载的形式。字符串，同样被压缩为一种更富效率的存储形式。由于这些原因，在 Android 平台中我们就有了这些不同的资源类型。

这是一篇纯粹的技术性文档，它和可用资源（[Available Resources](#)）一起覆盖了有关资源的众多信息。在使用 Android 时并不需要记住这篇文档，但是当你需要它时你应该知道来这里寻找信息。

## 介绍 Introduction

这个话题包含了与之相应的术语列表，和一系列在代码中使用资源的实例。关于 Android 支持的所有资源类型的完整指南，请查阅可用资源（[Available Resources](#)）。

Android 资源系统记录应用程序中所有非代码资产。你可以使用 [Resources](#) 类来访问应用程序中的资源；一般可以通过 [Context.getResources\(\)](#) 获得这个 Resources 实例。

一个应用程序的资源在生成（build）时被编译器编译到应用程序的二进制文件中。要使用一个资源，你必须把它放置到源代码树中的正确位置，并且生成（build）到你的应用程序中。作为编译过程的一部分，每个资源的标记都会被生成，在你的源代码中可以使用这些标记 - 这允许编译器验证你的应用程序代码是否和你定义的资源相匹配。

本部分的其余内容以一个在应用程序中如何使用资源的指南的形式组织。

## 创建资源 Creating Resources

Android 支持字符串、位图以及其他很多种类型的资源。每一种资源的语法、格式以及存放的位置，都会根据其类型的不同而不同。通常，你创建的资源一般来自于三种文件：XML 文件（除位图和 raw 之外的任何文件）、位图文件（图像）以及 Raw 文件（除前面以外的其他东西，如声音文件，等等）。事实上，XML 文件也有两种不同的类型：被原封不动地编译进包内的文件和被 aapt 用来产生资源的文件。这里有一个每种资源类型的列表，包括文件格式、文件描述以及 XML 文件类型的细节。

你可以在你的项目中的 res/目录的适当的子目录中创建和保存资源文件。Android 有一个资源编译器（aapt），它依照资源所在的子目录及其格式对其进行编译。这里有一个每种资源的文件类型的列表，关于每种类型的描述、语法、格式以及其包含文件的格式或语法见资源参考。

表一

目录 Directory	资源类型 Resource Types
<code>res/anim/</code>	XML 文件，它们被编译进逐帧动画（ <a href="#">frame by frame animation</a> ）或补间动画（ <a href="#">tweened animation</a> ）对象

<p>res/drawable /</p>	<p>.png、.9.png、.jpg 文件，它们被编译进以下的 Drawable 资源子类型中：</p> <p>要获得这种类型的一个资源，可以使用 <code>Resource.getDrawable(id)</code></p> <p><a href="#">位图文件</a> <a href="#">9-patches（可变尺寸的位图）</a></p> <p>为了获取资源类型，使用</p> <pre>mContext.getResources().getDrawable(R.drawable.imageId)</pre> <p><b>注意：</b>放在这里的图像资源可能会被 aapt 工具自动地进行无损压缩优化。比如，一个真彩色但并不需要 256 色的 PNG 可能会被转换为一个带调色板的 8 位 PNG。这使得同等质量的图片占用更少的资源。所以我们得意识到这些放在该目录下的二进制图像在生成时可能会发生变化。如果你想读取一个图像位流并转换成一个位图(bitmap)，请把图像文件放在 <code>res/raw/</code> 目录下，这样可以避免被自动优化。</p>
<p>res/layout/</p>	<p>被编译为屏幕布局（或屏幕的一部分）的 XML 文件。参见布局声明（<a href="#">Declaring Layout</a>）</p>
<p>res/values/</p>	<p>可以被编译成很多种类型的资源的 XML 文件。</p> <p><b>注意：</b>不像其他的 res/文件夹，它可以保存任意数量的文件，这些文件保存了要创建资源的描述，而不是资源本身。XML 元素类型控制这些资源应该放在 R 类的什么地方。</p> <p>尽管这个文件夹里的文件可以任意命名，不过下面使一些比较典型的文件（文件命名的惯例是将元素类型包含在该名称之中）：</p> <p><b>array.xml</b> 定义数据</p> <p><b>colors.xml</b> 定义 <a href="#">color drawable</a> 和 <a href="#">颜色的字符串值（color string values）</a>。使用 <code>Resource.getDrawable()</code>和 <code>Resources.getColor()</code>分别获得这些资源。</p> <p><b>dimens.xml</b> 定义 <a href="#">尺寸值（dimension value）</a>。使用 <code>Resources.getDimension()</code>获得这些资源。</p> <p><b>strings.xml</b> 定义 <a href="#">字符串（string）</a> 值（使用 <code>Resources.getString()</code>或者 <code>Resources.getText()</code>获取这些资源。<code>getText()</code>会保留在 UI 字符串上应用的丰富的文本样式）。</p> <ul style="list-style-type: none"> <li>• <b>styles.xml</b> 定义 <a href="#">样式（style）</a> 对象。</li> </ul>

<code>res/xml/</code>	任意的 XML 文件，在运行时可以通过调用 <a href="#">Resources.getXML()</a> 读取。
<code>res/raw/</code>	直接复制到设备中的任意文件。它们无需编译，添加到你的应用程序编译产生的压缩文件中。要使用这些资源，可以调用 <a href="#">Resources.openRawResource()</a> ，参数是资源的 ID，即 <code>R.raw.somefilename</code> 。

资源被编进最终的 APK 文件中。Android 创建了一个封装类，叫做 **R**，在代码中你可以使用它来引用这些资源。R 包含了根据资源文件的路径和名称命名的子类。

## 全局资源说明 Global Resource Notes

一些资源允许你定义颜色值。Android 接受的颜色值可以使用多种 web 样式的形式--以下几种包含十六进制常数的形式：**#RGB**、**#ARGB**、**#RRGGBB**、**#AARRGGBB**。

所有颜色值支持设置透明度（alpha channel value），前两位的十六进制数指定了透明了。0 在透明度值是全透明。默认值是不透明。

## 使用资源 Using Resources

这一部分描述如何使用你创建的资源。它包含以下主题：

- [代码中使用资源](#) - 如何在你的代码中调用资源进行实例化。
- [从其他资源中引用资源](#)

[源](#)

[../Docs/android\\_dev\\_guide/android\\_dev\\_guide/developer.android.com/guide/topics/resources/resources-i18n.html - ReferencesToResources](https://developer.android.com/guide/topics/resources/resources-i18n.html) - 你可以从其他资源中引用资源。这就使得你可以重用资源中公共资源值。

- [支持针对交替配置的交替资源](#) - 你可以根据主机硬件的语言或显示配置指定加载不同的资源。

在编译时，Android 产生一个名为 **R** 的类，它包含了你的程序中所有资源的资源标识符。这个类包含了一些子类，每一个子类针对一种 Android 支持的资源类型，或者你提供的一个资源文件。每一个类都包含了已编译资源的一个或多个资源标识符，你可以在代码中使用它们来加载资源。下面是一个小的资源文件，包含了字符串、布局（屏幕或屏幕的一部分）和图像资源。

**注意:** R 类是一个自动产生的文件，并没有设计为可以手动编辑。当资源更新时，它会根据需要重新产生。

```
package com.google.android.samples;

public final class R {

    public static final class string {

        public static final int greeting = 0x0204000e;
        public static final int start_button_text = 0x02040001;
        public static final int submit_button_text = 0x02040008;
        public static final int main_screen_title = 0x0204000a;
    };

    public static final class layout {

        public static final int start_screen = 0x02070000;
        public static final int new_user_pane = 0x02070001;
        public static final int select_user_list = 0x02070002;
    };

    public static final class drawable {

        public static final int company_logo = 0x02020005;
        public static final int smiling_cat = 0x02020006;
        public static final int yellow_fade_background = 0x02020007;
        public static final int stretch_button_1 = 0x02020008;
    };
};
```

## 在代码中使用资源 **Using Resources in Code**

在代码中使用资源，只是要知道所有资源 ID 和你的被编译的资源是什么类型。下面是一个引用资源的语法：

```
R.resource_type.resource_name
```

或者

```
android.R.resource_type.resource_name
```

其中 `resource_type` 是 `R` 的子类，保存资源的一个特定类型。`resource_name` 时在 XML 文件定义的资源的 **name** 属性，或者有其他文件类型为资源定义的文件名（不包含扩展名）。每一种资源类型都会根据其类型加为一个特定的 `R` 子类；要了解 `R` 的哪一个子类是关于你的资源

类型的，请参考资源参考（resource reference）文档。被你的应用程序编译的资源可以不加包名引用（就像 `R.resource_type.resource_name` 这样简单）。Android 包含了很多标准资源，如屏幕样式和按钮背景。要在代码中引用这些资源，你必须使用 `android` 进行限定，如 `android.R.drawable.button_background`。

这里有一些在代码中使用已编译资源的正确和错误用法的例子。

```
// Load a background for the current screen from a drawable
resource.
this.getWindow().setBackgroundDrawableResource(R.drawable.my_ba
ckground_image);
// WRONG Sending a string resource reference into a
// method that expects a string.
this.getWindow().setTitle(R.string.main_title);
// RIGHT Need to get the title from the Resources wrapper.
this.getWindow().setTitle(Resources.getText(R.string.main_title)
);
// Load a custom layout for the current screen.
setContentView(R.layout.main_screen);
// Set a slide in animation for a ViewFlipper object.
mFlipper.setInAnimation(AnimationUtils.loadAnimation(this,
R.anim.hyperspace_in));
// Set the text on a TextView object.
TextView msgTextView = (TextView)findViewById(R.id.msg);
msgTextView.setText(R.string.hello_message);
```

## 引用资源 References to Resources

在属性（或资源）中提供的值也可以作为资源的引用。这种情况经常使用在布局文件中，以提供字符串（因此它们可以被本地化<将 UI 上的字符串放在一个单独的文件中，在做国际化时只需要将它们翻译成相应的语言版本，然后应用程序根据 **locale** 信息加载相应的字符串文件——译者注>）和图像（它们存在于另外的文件中），虽然引用可以是任何资源类型，包括颜色和整数。

例如，如果我们有颜色资源（[color resources](#)），我们可以编写一个布局文件，将文本的颜色设为那些资源中包含的值：

```
<?xml version="1.0" encoding="utf-8"?>

<EditText id="text" xmlns:android="http://schemas.android.com/apk/res
```

```

/android"

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@color/opaque_red"
    android:text="Hello, World!" />

```

注意，这里使用“@”前缀引入对一个资源的引用——在@[package:]type/name 形式中后面的文本是资源的名称。在这种情况下，我们不需要指定包名，因为我们引用的是我们自己包中的资源。要引用系统资源，你应该这样写：

```

<?xml version="1.0" encoding="utf-8"?>

<EditText id="text"
xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@android:color/opaque_red"
    android:text="Hello, World!" />

```

另外一个例子，当在布局文件中提供字符串以便于本地化时，你应该一直使用资源引用。

```

<?xml version="1.0" encoding="utf-8"?>

<EditText id="text"
xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@android:color/opaque_red"
    android:text="@string/hello_world" />

```

这种技巧还可以用来创建资源之间的引用。例如，我们可以创建新的 **drawable** 资源作为已存在资源的别名。

```

<?xml version="1.0" encoding="utf-8"?>

<resources>

    <drawable
        id="my_background">@android:drawable/theme2_background</drawabl
    e>

```

```
</resources>
```

## 引用主题属性 References to Theme Attributes

另外一种资源值允许你引用当前主题中的属性的值。这个属性值只能在样式资源和 XML 属性中使用；它允许你通过将它们改变为当前主题提供的标准变化来改变 UI 元素的外观，而不是提供具体的值。

如例中所示，我们在布局资源中使用这个特性将文本颜色设定为标准颜色的一种，这些标准的颜色都是定义在基本系统主题中。

```
<?xml version="1.0" encoding="utf-8"?>

<EditText id="text"
xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="?android:textDisabledColor"
    android:text="@string/hello_world" />
```

注意，这和资源引用非常类似，除了我们使用一个“?”前缀代替了“@”。当你使用这个标记时，你就提供了属性资源的名称，它将会在主题中被查找——因为资源工具知道需要的属性资源，所以你不需显示声明这个类型（如果声明，其形式就是 `?android:attr/android:textDisabledColor`）。

除了使用这个资源的标识符来查询主题中的值代替原始的资源，其命名语法和“@”形式一致：`?[namespace:]type/name`，这里类型可选。

## 使用系统资源 Using System Resources

在系统中的包含了很多应用程序可以使用的资源。所有的这些资源都在“`android.R`”类下定义。例如，使用下面的代码你可以在屏幕上显示标准应用程序的图标：

```
public class MyActivity extends Activity {

    public void onStart() {

        requestScreenFeatures(FEATURE_BADGE_IMAGE);

        super.onStart();

        setBadgeResource(android.R.drawable.sym_def_app_icon);
    }
}
```

```
    }  
}
```

以相似的方式，下面的代码将对你的屏幕应用系统定义的标准“绿色背景”视觉处理。

```
public class MyActivity extends Activity  
  
    public void onStart() {  
  
        super.onStart();  
  
        setTheme(android.R.style.Theme_Black);  
  
    }  
  
}
```

### 替换资源（为了可替换的资源 and 配置）*Alternate Resources*

你可以根据 UI 语言或者设备上的硬件配置，为你的产品提供不同的资源。注意，尽管你可以包含不同的字符串、布局和其他资源，然而 **SDK** 没有方法供你指定加载哪一个替换资源。**Android** 检测关于硬件和未知的适当配置，然后适当加载。用户可以使用设备上的设置面板选择替换语言设置。

为了包含替换资源，需要创建平行的资源文件夹，而文件夹的名字后面要使用限定符表示它要应用的配置（语言、屏幕方向等等）。例如，下面的工程包含了字符串资源，一个用于英语，而另外一个用于法语：

```
MyApp/  
  
    res/  
  
        values-en/  
  
            strings.xml  
  
        values-fr/  
  
            strings.xml
```

**Android** 支持几种类型的限定符，每一个都有不同的值。把它们连接在资源文件夹名称的后面，使用短横线隔开。你可以为每一个文件夹名称添加多个限定符，但是它们必须按照这里列出的顺序排列。例如，一个包含 **drawable** 资源的文件夹，对于一个完整详细的配置，可能看起来像：

```
MyApp/  
  
    res/
```

```
values-en/  
  
drawable-en-rUS-port-160dpi-finger-qwerty-dpad-480x320/
```

更典型的是，你只需指定一些特定的要定义资源的配置选项。你可以放弃完整列表中的任何值，但同时要保证剩下的值仍然保持列表中的顺序。

```
MyApp/  
  
    res/  
  
        drawable-en-rUS-finger/  
  
        drawable-port/  
  
        drawable-port-160dpi/  
  
        drawable-qwerty/
```

表 2 列举了合法的限定符目录名称，按优先级排序。下表中列举在上面的限定符比下面的具有更高的优先级，如同 [Android 如何查找最匹配的目录](#) 中所描述的那样。

表 2

限定符 Qualifier	值 Values
移动国家码 MCC 和移动网络码 MNC	<p>手机设备 SIM 卡上的移动国家码和移动网络码。比如 <code>mcc310-mnc004</code> (美国, Verizon 品牌); <code>mcc208-mnc00</code> (法国, Orange 品牌); <code>mcc234-mnc00</code> (英国, BT 品牌).</p> <p>如果这个设备使用一个无线连接（GSM 电话），则 MCC 来自 SIM 卡，而 MNC 来自该设备将要附着的网络。你有时会仅使用 MCC，例如包含特定国家合法资源在您的应用程序中。如果您的应用程序指定了 MCC/MNC 组合的资源，这些资源仅在 MCC 和 MNC 都匹配的时候才能使用。</p>
语言和区域 Language and region	<p>两个字母的 <a href="#">ISO 639-1</a> 语言码和 <a href="#">ISO 3166-1-alpha-2</a> 区域码 (以"r"为前缀)。比如 <code>en-rUS</code>, <code>fr-rFR</code>, <code>es-rES</code>.</p> <p>这个代码是大小写敏感的：语言码是小写字母，国家码是大写字母。你不能单独指定一个区域，但是你可以单独指定一个语言，比如 <code>en</code>,</p>

	<code>fr, es, zh.</code>
屏幕方向 Screen orientation	纵向, 横向, 正方形 ( <code>port, land, square</code> )
屏幕像素密度 Screen pixel density	<code>92dpi, 108dpi</code> 等. 当 <b>Android</b> 选择使用哪个资源时, 它对屏幕像素密度的处理和其它限定符不同。在文章后面描述的步骤 <a href="#">1Android 如何查找最匹配的目录</a> 中, 屏幕密度总被认为是匹配的。在步骤 <b>4</b> 中, 如果被考虑的限定符是屏幕密度, <b>Android</b> 将选择在那个位置的最佳匹配, 而无需继续步骤 <b>5</b> 。
触摸屏类型 Touchscreen type	非触摸式, 触摸笔, 手指 ( <code>notouch, stylus, finger</code> )
键盘可用方式 Whether the keyboard is available to the user	外在键盘, 隐藏键盘, 软键盘 ( <code>keysexposed, keyshidden, keysoft</code> ) 如果你的应用程序有一个特定的资源只能通过软件盘使用, 则使用 <code>keysoft</code> 值, 如果没有 <code>keysoft</code> 资源可用 (只有 <code>keysexposed</code> 和 <code>keyshidden</code> ) 并且该设备显示了一个软键盘, 那么系统将使用 <code>keysexposed</code> 资源。
首选文本输入方法 Primary text input method	不支持按键, 标准键盘, 12 键 ( <code>nokeys, qwerty, 12key</code> )
首选非触摸式导航方法 Primary non-touchscreen navigation method	不支持导航, 滑板, 跟踪球, 滚轮 ( <code>nonav, dpad, trackball, wheel</code> )
屏幕分辨率 Screen dimensions	<code>320x240, 640x480</code> , 等. 更大的分辨率必须先被指定。
SDK 版本 SDK version	设备支持的 SDK 版本, 比如 <code>v3</code> 。Android1.0 SDK 是 <code>v1</code> , 1.1SDK 是 <code>v2</code> , 1.5SDK 是 <code>v3</code> 。

小版本(Minor version)	你目前还不能指定小版本，它总是被设置为 0。
--------------------	------------------------

这个列表不包含设备特有的参数比如载波，品牌，设备/硬件，或者制造商。所有应用程序需要知道的设备信息均通过上表中的资源限定符编码。

所有资源目录，许可的和未经许可的，都存放在 `res/` 目录下。下面是一些关于许可的资源目录名称的指导原则：

- 你可以指定多个限定符，用破折号分开。比如，`drawable-en-rUS-land` 会被应用在美国英语的横向手机设备中。
- 限定符必须符合表 2 中列举的顺序。比如：
  - 正确的：`values-mcc460-nokeys/`
  - 错误的：`values-nokeys-mcc460/`
- 限定符的值大小写敏感。比如一个纵向特定的 `drawable` 目录必须命名为 `drawable-port`，不可以是 `drawable-PORT` 或 `drawable-Port`。
- 每个限定符类型仅支持一个值。比如，如果你想使用为法国和西班牙使用相同的 `drawable` 文件，你得需要两个资源目录，如 `drawable-rES/` 和 `drawable-rFR/`，包含相同的文件。你不能使用一个名为 `drawable-rES-rFR` 的目录。
- 限定符不能嵌套使用。比如，你不能使用 `res/drawable/drawable-en`。

### 资源怎么在代码中使用 *How resources are referenced in code*

所有的资源均通过它们简单未经修饰的名字在代码或资源引用语法中引用。所以如果一个资源命名如下：

```
MyApp/res/drawable-port-92dpi/myimage.png
```

它会被这样引用：

```
R.drawable.myimage (code)
@drawable/myimage (XML)
```

如果有多个 `drawable` 目录可用，Android 将会选择其一（如下所述）并从中加载 `myimage.png`。

### Android 如何查找最匹配的目录 *How Android finds the best matching directory*

Android 将从各种潜在的资源中挑选出哪个应该在运行时使用，这取决于设备的当前配置。这里的例子假定使用了如下的设备配置：

区域 `Locale = en-GB`

屏幕方向 `Screen orientation = port`

屏幕像素密度 `Screen pixel density = 108dpi`

触摸屏类型 `Touchscreen type = notouch`

首选文本输入方式 `Primary text input method = 12key`

下面说明了 Android 如何作出选择：

1. 排除和设备配置冲突的资源文件。比如，假定如下的 `drawables` 资源目录可用。那么 `drawable-fr-rCA/` 会被排除，因为它和设备的区域配置冲突。

```
MyApp/res/drawable/  
MyApp/res/drawable-en/  
MyApp/res/drawable-fr-rCA/  
MyApp/res/drawable-en-port/  
MyApp/res/drawable-en-notouch-12key/  
MyApp/res/drawable-port-92dpi/
```

**例外：**屏幕像素密度是唯一不用来排除文件的限定符。即使设备屏幕密度是 108dpi，`drawable-port-92dpi/` 也不会被从列表中排除，因为在这里所有的屏幕密度都被视为匹配。

2. 从表 2 中选取最高优先级的限定符（从 MCC 开始，然后自该列表依次往下）。
3. 有没有哪个可用的资源目录包含了这个限定符？
  - ✧ 如果没有，回到步骤 2 然后查看表 2 中所列的下一个限定符。在我们的例子中，答案是“没有”直到我们到达语言这一级。If No, return to step 2 and look at the next qualifier listed in Table 2. In our example, the answer is "no" until we reach Language;
  - ✧ 如果有，则跳转到步骤 4。
4. 排除不包含这个限定符的资源目录，在我们的例子中，我们排除所有不包含语言的目录。

```
MyApp/res/drawable/  
MyApp/res/drawable-en/  
MyApp/res/drawable-en-port/  
MyApp/res/drawable-en-notouch-12key/  
MyApp/res/drawable-port-92dpi/  
MyApp/res/drawable-port-notouch-12key
```

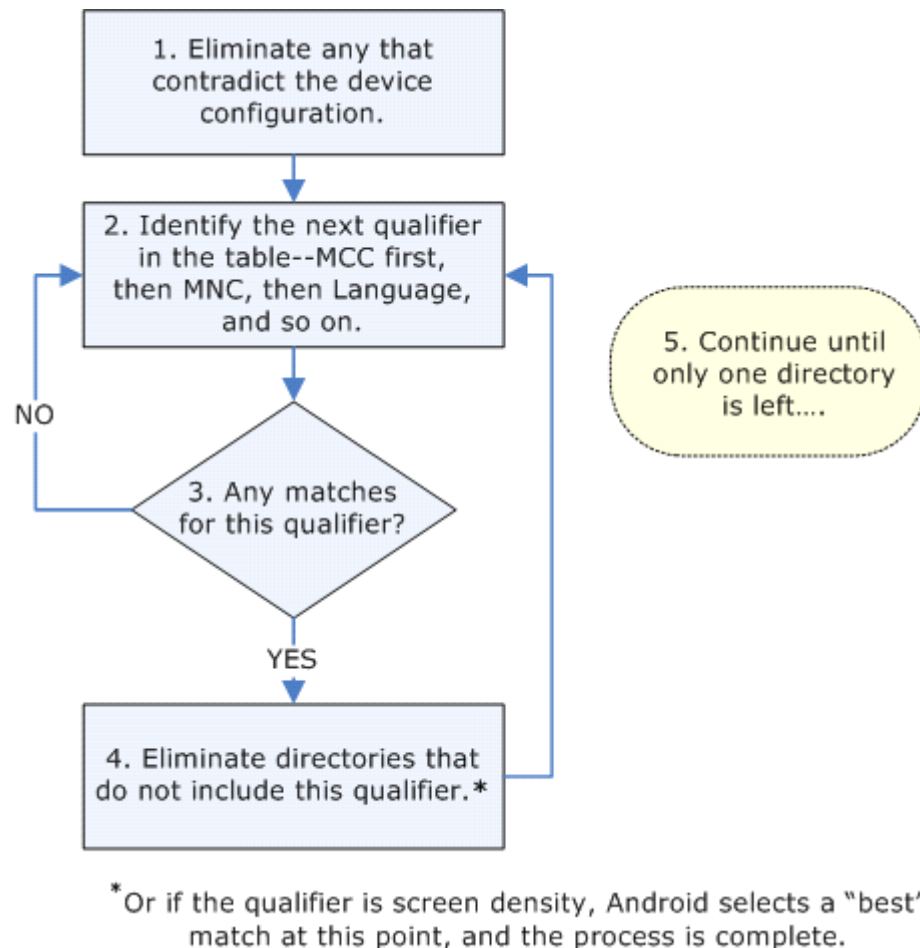
**例外：**如果询问中的限定符是屏幕像素密度，Android 会选择最接近匹配于设备的选项，而且选择过程将会完成。一般而言，Android 会倾向于缩小一个大图片而不是放大一个小图片。

5. 回头重复步骤 2, 3, 4 直到只剩下一个选择。在本例中，屏幕方向是下一个要比较的限定符，我们排除没有指定屏幕方向的资源。现在只剩下一个选择，那就是它了。当 `drawables` 被这个应用程序调用时，Android 系统会从下面这个目录中加载资源：

```
MyApp/res/drawable-en-port/
```

**提示 Tip:** 限定符的优先权比匹配的数目要重要得多。比如，在上面的步骤 4 中，列表中最后的选项包含三个限定符和设备匹配（方向，触摸屏类型，和输入法），而 `drawable-en` 只有一个参数匹配（语言）。但是，语言拥有更高的优先权，所以 `drawable-port-notouch-12key` 被排除出局。

下面的流程图总结了 Android 如何选择资源目录来加载的过程：



## 术语 Terminology

资源系统将一系列分散内容集合在一起形成最终的完整的资源功能，去帮助我们了解整个系统。这里有一些核心概念以及组件的概要说明，你在开发中将可能使用到：

**资产 Asset:** 应用程序的独立的数据块。这包含所有从 java 程序编译成的目标文件，图像（例如 PNG 图片），XML 文件等等。这些文件以一种特定的方式组织在一起，在程序最后打包时，它们被捆绑进一个单独的 ZIP 文件里。

**aapt:** Android 最终文件打包工具。这个工具产生最终程序的 ZIP 文件。除了将资产元数据文件收集在一起，它也把资源定义解析到最终的二进制数据里。

**资源表 Resource Table:** aapt 工具产生的特殊的文件，描述了所有在程序/包里的资源。这个文件可以通过资源类来访问；它不能直接和应用程序接触。

**资源 Resource:** 资源表里一条记录描述的是单一的命名值。大体上，资源分成两种：元资源和包资源。

**资源标识符 Resource Identifier:** 在资源表里所有的资源都被唯一的整数标识着。所有的代码中（资源描述，XML 文件，Java 源代码）你可以直接使用符号名代替真实的整数数值。

**元资源 Primitive Resource:** 所有元资源都可以被写成一个简单的字串，使用一定的格式可以描述资源系统里各种不同的基本类型：整数，颜色，字串，其他资源的引用，等等。像图片以及 XML 描述文件这些复杂资源，被以元字串资源储存，它们的值就是相关最终数据文件的路径。

**包资源 Bag Resource:** 一种特殊类型的资源，不是简单的字符串，而是一个容纳名字/数值对的任意列表。每个名字本身就是资源标识，每个值可以容纳相同类型的字符串格式的数据作为一个普通资源。包资源支持继承：一个包里的数据能从其他包里继承，有选择地替换或者扩展能产生它自己的内容。

**种类 Kind:** 资源种类是对于不同需求的资源标识符而言的。例如，绘制资源类常常实例化绘制类的对象，所以这些包含颜色以及指向图片或 XML 文件的字符串路径数据是原始数据。其它常见资源类型是字符串（本地化字符串），颜色（基本颜色），布局（一个指向 XML 文件的字串路径，它描述的是一个用户界面）以及风格（一个描述用户接口属性的包装资源）。还有一个标准的“attr”资源类型，它定义了命名包装数据以及 XML 属性的资源标识符。

**风格 Style:** 包含包装资源类型的名字常常用来描述一系列用户接口属性。例如，一个 TextView 的类可能会有一个描述界面风格的类来定义文本大小，颜色以及对齐方式。在一个界面布局的 XML 文件中，可以使用“风格”属性来确定整体界面风格，它的值就是风格资源的名字。

**风格类 Style Class:** 这里将详述一些属性资源类。其实数据不会被放在资源表本身，通常在源代码里它以常量的形式出现，这也可以使你在风格类或者 XML 的标签属性里方便找到它的值。例如，Android 平台里定义了一个“视图”的风格类，它包含所有标准视图的属性：画图区域，可视区域，背景等。这个视图被使用时，它就会借助风格类去从 XML 文件取得数据并将其载入到实例中。

**配置 Configuration:** 对许多特殊的资源标识符，根据当前的配置，可以有多种不同的值。配置包括地区（语言和国家），屏幕方向，屏幕分辨率，等等。当前的配置用来选择当资源表载入时哪个资源值生效。

**主题 Theme:** 一个标准类型的资源能为一个特殊的上下文提供全局的属性值。例如，当应用工程师写一个活动时，他能选择一个标准的主题去使用，白色的或者黑色的；这个类型能提供很多信息，如屏幕背景图片/颜色，默认文本颜色，按钮类型，文本编辑框类型，文本大小，等。当布置一个资源布局时，控件（文本颜色，选中后颜色，背景）的大部分设置值取自当前主题；如果需要，布局中的风格以及属性也可以从主题的属性中获得。

覆盖层 **Overlay**: 资源表不能定义新类型的资源，但是你可以在其他表里替换资源值。就像配置值，这可以在装载时候进行；它能加入新的配置值（例如，改变字符串到新的位置），替换现有值（例如，将标准的白色背景替换成"Hello Kitty"的背景图片），修改资源包（例如修改主题的字体系大小。白色主题字体系大小为 18pt）。这实际上允许用户选择设备不同的外表，或者下载新的外表文件。

## 资源引用 Resource Reference

可用资源 [Available Resources](#) 文档提供了一个各种类型资源的详细列表，并描述了如何在 Java 代码中或其他引用中使用它们。

## 国际化和本地化 Internationalization and Localization

**即将完成:** 国际化和本地化是非常关键的，但现在的 SDK 还没有完全准备好。当 SDK 成熟时，这个章节会包含 Android 平台国际化和本地化的相关信息。在此期间，让我们先从把资源外部化以及练习以好的结构创建和使用资源开始做起吧。

## 意图和意图过滤器 Intents and Intent Filters

一个应用程序的三个核心组件-活动，服务和广播接收器是通过消息即意图（Intents）来激活的。Intent 信息传送是相同或不同应用中组件运行时晚绑定的一种机制。意图本身，一个意图对象，是一个包含被执行操作抽象描述的被动的数据结构-或者，对于广播而言，是某件已经发生并被声明的事情的描述。存在不同的机制来传送意图到每种组件中：

- 一个意图对象是传递给 [Context.startActivity\(\)](#) 或者 [Activity.startActivityForResult\(\)](#) 来启动一个活动或者让一个存在的活动去做某些新的事情。
- 一个意图对象是传递给 [Context.startService\(\)](#) 来发起一个服务或者递交新的指令给运行中的服务。类似的，一个意图能被传递给 [Context.bindService\(\)](#) 来在调用组件和一个目标服务之间建立连接。作为一个可选项，它可以发起这个服务如果还没运行的话。
- 传递给任意广播方法(例如

[Context.sendBroadcast\(\)](#), [Context.sendOrderedBroadcast\(\)](#), 或者

[Context.sendStickyBroadcast\(\)](#)) 的意图对象被传递给所有感兴趣的广播接收者。许多种广播产生于系统代码。

在每个例子里，Android 系统找到合适的活动，服务，或者一组广播接收者来回应这个意图，必要时实例化它们。这些消息传送系统没有重叠：广播意图仅被传递给广播接收者，永远不会给活动或者服务。一个

传送给 `startActivity()` 的意图是只会被传递给一个活动，永远不会给一个服务或广播接收者，如此类推。

这篇文档以意图对象的描述开始，然后描述 **Android** 映射意图到组件的规则-如何解决哪个组件应该接收一个意图消息。对于没有显式命名一个目标组件的意图，这个过程包括对照与潜在目标相关联的意图过滤器来测试这个意图对象。

## 意图对象 **Intent Objects**

一个意图 [Intent](#) 对象是一堆信息。它包含接收这个意图的组件感兴趣的信息（例如将要采取的动作和操作的数据）再加上 **Android** 系统感兴趣的信息（例如应该处理这个意图的组件类别和如何启动一个目标活动的指令）：

### 组件名称 **Component name**

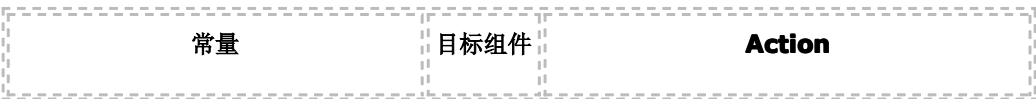
应该处理这个意图的组件名字。这个字段是一个 [ComponentName](#) 对象- 一个组合物：目标组件的完全合格的类名（比如"`com.example.project.app.FreneticActivity`"）以及应用程序描述文件中设置的组件所在包的名称(比如, "`com.example.project`"). 这个组件名称的包部分和描述文件中设置的包名称不一定要匹配。

组件名称是可选的。如果被设置了，这个意图对象将被传递到指定的类。如果没有, **Android** 使用另外的意图对象中的信息去定位一个合适的目标- 请看本文稍后描述的意图解析 [Intent Resolution](#)。

组件名称通过如下方法：[setComponent\(\)](#),[setClass\(\)](#), 或者 [setClassName\(\)](#) 设置并通过 [getComponent\(\)](#) 读取。

### 动作 **Action**

一个将被执行的动作的字符串命名-或者，对于广播意图而言，是发生并被报告的动作。这个意图类定义了一些动作常量，包含下面这些：



ACTION_CALL	活动	开始一个电话呼叫
ACTION_EDIT	活动	显示数据以给用户编辑
ACTION_MAIN	活动	开始任务的初始活动，没有输入数据也没有输出返回
ACTION_SYNC	活动	同步服务器与移动设备之间的数据
ACTION_BATTERY_LOW	广播接收器	电池低电量警告
ACTION_HEADSET_PLUG	广播接收器	耳机插拔
ACTION_SCREEN_ON	广播接收器	屏幕开启
ACTION_TIMEZONE_CHANGED	广播接收器	时区变化

通过查看 [Intent](#) 类描述可获得一个通用动作的预定义常量列表。其他动作被定义在 **Android API** 的其他地方。你也可以自定义动作字符串来激活应用程序中的组件。那些你所创建的动作字符串应该以应用程序包名作为前缀-例如：

"com.example.project.SHOW\_COLOR".

动作很大程度上决定了意图其他部分如何被组织-尤其是数据 [data](#) 和附加字段 [extras](#)-很像一个方法名决定了一些参数和返回值。因此，一个好的想法是使用尽可能具体的动作名并和意图的其他字段紧密联系起来。换句话说，为您的组件能处理的意图对象定义一个整体的协议而不是定义一个孤立的动作。

一个意图对象里的动作可以通过 [setAction\(\)](#) 方法设置和通过 [getAction\(\)](#) 方法读取。

## 数据 Data

想要操作的数据统一资源标识符（URI）和那种数据的多用途互联网邮件扩展（MIME）。不同的动作伴随着不同种类的数据规格。例如，如果动作是 ACTION\_EDIT，数据字段会包含可编辑文档的 URI；如果动作是 ACTION\_CALL，数据字段会是一个电话号码：含呼叫电话号码的 URI；类似的，如果动作是 ACTION\_VIEW 而且数据字段是一个 http:URI，那么接收到的活动将会是下载并显示 URI 所引用数据的请求。当匹配一个意图到一个能处理数据的组件时，除了它的 URI 外，通常要知道数据类型（它的 MIME 类型）。

比如，一个能显示图片的组件不应该被要求去播放一个声音文件。

在很多情况下，这个数据类型可以从 URI 里推断出来-尤其是 content:URIs，这意味着数据被存放在设备上而且由一个内容提供者控制着。（参阅 [separate discussion on content providers](#)）。但类型可以在意图对象里显示的设置。setData()方法指定数据只能为一个 URI，setType()指定它只能是一个 MIME 类型，而 setDataAndType()指定它同时为 URI 和 MIME 类型。URI 通过 getData()读取，类型则通过 getType()。

## 目录 Category

一个包含关于应该处理这个意图的组件的附加信息的字符串。任意数目的类别描述可以被放到一个意图对象里。和动作一样，意图类定义若干类别常量，包含如下这些：

常量	含义
CATEGORY_BROWSABLE	目标活动可以被浏览器安全的唤起来显示被一个链接所引用的数据-比如，一张图片或一条 e-mail 消息。
CATEGORY_GADGET	这个活动可以被嵌入到充当配件宿主的另外的活动里面。
CATEGORY_HOME	这个活动将显示桌面，也就是用户开机后看到的第一个屏幕或者按 HOME 键时看到的屏幕。
CATEGORY_LAUNCHER	这个活动可以是一个任务的初始活动并被列在应用程序启动器的顶层。
CATEGORY_PREFERENCE	目标活动是一个选择面板。

查阅 [Intent](#) 类描述可获取类别的完整列表。

[addCategory\(\)](#) 方法在一个意图对象中添加了一个目录，[removeCategory\(\)](#) 删除之前添加

的目录，而 [getCategories\(\)](#) 可以获取当前对象的所有类别。

## 附加信息 Extras

应该递交给意图处理组件的附加信息键-值对。就像一些动作伴随着特定的数据 URIs 类型，一些动作则伴随着特定的附加信息。比如，一个 ACTION\_TIMEZONE\_CHANGED 意图有一个“时区”附加信息用来区别新的时区，而 ACTION\_HEADSET\_PLUG 有一个“状态”附加字段表明耳机有没有插着，以及一个“名字”附加信息来表示耳机的类型。如果你想要创建一个 SHOW\_COLOR 动作，颜色的值将被设置在一个附加的键-值对中。

意图对象有一系列的 put...() 方法来插入各种不同的附加数据和一个类似的用来读取数据的 get...() 方法系列。这些方法与 [Bundle](#) 对象的方法相似。事实上，附加信息可以被当作一个 Bundle 通过使用 [putExtras\(\)](#) 和 [getExtras\(\)](#) 方法安装和读取。

## 标志 Flags

各种类型的标志。许多标志用来指示 Android 系统如何去加载一个活动（例如，哪个是这个活动应该归属的任务）和启动后如何对待它（比如，它是否属于当前活动列表），所有这些列表都在意图类中定义了。

Android 系统以及这个平台上的应用程序利用意图对象来发送源于系统的广播以及激活系统定义的组件。要查阅如何组织一个意图去激活一个系统组件，请咨询引用中的意图列表 [list of intents](#)。

## 意图解析 Intent Resolution

意图可以被分成两组：

- **显式意图** 通过名字指明目标组件（这个组件名字字段 [component name field](#)，前面提到过，有一个数值集）。既然组件名称通常不为其他应用程序的开发者所了解，显式意图典型的被用作应用程序的内部消息-例如一个活动启动一个附属服务或姊妹活动。
- **隐式意图** 不命名目标组件（组件名称字段为空）。隐式意图经常用来激活其他应用程序的组件。

Android 递交一个显式的意图给一个指定目标类的实例。意图对象中的组件名称唯一的确定哪个组件应该获取这个意图。隐式意图需要一个不同的策略。在没有指定目标的情况下，Android 系统必须找到最合适的组件来处理这个意图-单个活动或者服务来执行这个请求动作或者一系列的广播接收器来应对广播通告。

这是通过比较意图对象的内容和意图过滤器，有可能接收意图的组件相关结构。过滤器公布一个组件具备的能力以及限定它能处理的意图。他们使组件接收该公布类型的隐式意图成为可能。如果一个组件没有任何的意图过滤器，那它只能接收显式意图。一个带过滤器的组件可以同时接收显式和隐式意图。

当一个意图对象被一个意图过滤器测试时，只有三个方面会被参考到：

动作

数据（URI 以及数据类型）

类别

附加信息和标志并不参与解析哪个组件接收一个意图。

## 意图过滤器 Intent filters

为了通知系统它们可以处理哪些意图，活动、服务和广播接收器可以有一个或多个意图过滤器。每个过滤器描述组件的一个能力，一系列组件想要接收的意图。它实际上按照一个期望的类型来进行意图滤入，同时滤出不想要的意图-但是只有不想要的隐式意图会被滤出（那些没有命名目标的对象类）。一个显式意图总能够被递交给它的目标，而无论它包含什么。这种情况下过滤器不起作用。但是一个显式意图仅当它能够通过组件的一个过滤器时才可以被递交到这个组件。

组件为它能做的每项工作，每个呈现给用户的不同方面分有不同的过滤器。比如，范例记事本应用程序中的主要活动有三个过滤器-一个是空白板，另一个是用户可以查看、编辑、或选择的一个指定的记事目录，第三是在没有初始目录说明的情况下查找一个特定的记录。一个意图过滤器是 [IntentFilter](#) 类的一个实例。但是，由于 **Android** 系统在启动一个组件前必须知道这个组件的能力，意图过滤器通常不会用 **Java** 代码来设置，而是在应用程序清单文件（**AndroidManifest.xml**）中设置<intent-filter>元素。（有一个例外，通过调用 [Context.registerReceiver\(\)](#) 来注册的广播接收器的过滤器；它们是作为意图过滤器对象而被直接创建的。

## 过滤器与安全 Filters and security

不能信赖一个意图过滤器的安全性。当它打开一个组件来接收某些特定类型的隐式意图，它并不能阻止以这个组件为目标的显式意图。即使过滤器对组件要处理的意图限制某些动作和数据源，总有人能把一个显式意图和一个不同的动作及数据源组合在一起，然后命名该组件为目标。

一个过滤器和意图对象有同样的动作、数据以及类别字段。一个隐式意图在过滤器的所有三个方面都被测试。为了递交到拥有这个过滤器的组件，它必须通过所有这三项测试。即便只有一个不通过，**Android** 系统都不会把它递交给这个组件-至少以那个过滤器的标准而言。不过，由于一个组件可以包含多个意图过滤器，一个不能通过其中一个组件过滤器的意图可能在另外的过滤器上获得通过。

三个测试详细描述如下：

## 动作测试 **Action test**

清单文件中的意图过滤器元素里列举了动作元素，比如：

```
<intent-filter . . . >

    <action android:name="com.example.project.SHOW_CURRENT" />

    <action android:name="com.example.project.SHOW_RECENT" />

    <action android:name="com.example.project.SHOW_PENDING" />

    . . .

</intent-filter>
```

如同例子所示，一个意图对象只对单个动作命名，而一个过滤器可能列举多个。列表不能为空；一个过滤器必须包含至少一个动作元素，否则它将阻塞所有的意图。

为了通过这个测试，在意图对象中指定的动作必须匹配过滤器中所列举的动作之一。如果意图对象或过滤器不指定一个动作，结果将如下：

- 如果这个过滤器没有列出任何动作，那意图就没有什么可匹配的，因此所有的意图都会测试失败。没有意图能够通过这个过滤器。
- 另一方面，一个未指定动作的意图对象自动通过这个测试-只要过滤器包含至少一个动作。

## 类别测试 **Category test**

一个意图过滤器<intent-filter>元素也列举了类别作为子元素。比如：

```
<intent-filter . . . >

    <category android:name="android.intent.category.DEFAULT" />

    <category android:name="android.intent.category.BROWSABLE" />

    . . .

</intent-filter>
```

注意前面描述的动作和类别常量没有在清单文件中使用。相反使用了完整的字符串。比如，对应于前述 `CATEGORY_BROWSABLE` 常量，上面的例子里使用了

"android.intent.category.BROWSABLE"字符串。类似的，字符串

"android.intent.action.EDIT" 对应于 ACTION\_EDIT 常量。

对一个通过类别测试的意图，每个意图对象中的类别必须匹配一个过滤器中的类别。这个过滤器可以列举另外的类别，但它不能遗漏任何在这个意图中的类别。

因此，原则上一个没有类别的意图对象应该总能够通过测试，而不管过滤器里有什么。绝大部分情况下这个是对的。但有一个例外，Android 把所有传给 [startActivity\(\)](#) 的隐式意图当作他们包

含至少一个类别："android.intent.category.DEFAULT"（CATEGORY\_DEFAULT 常量）。因此，想要接收隐式意图的活动必须在它们的意图过滤器中包含

"android.intent.category.DEFAULT"。（带"android.intent.action.MAIN"和

"android.intent.category.LAUNCHER"设置的过滤器是例外）。它们标记那些启动新任务和呈现在启动屏幕的活动。它们可以在类别列表中包含

"android.intent.category.DEFAULT"，但不是必要的。）可查阅后面的使用意图匹配（[Using intent matching](#)）以获得更多关于过滤器的信息。

## 数据测试 **Data test**

就像动作和类别，一个意图过滤器的数据规格被包含在一个子元素中。而且这个子元素可以出现多次或一次都不出现。例如：

```
<intent-filter . . . >

    <data android:type="video/mpeg" android:scheme="http" . . . />

    <data android:type="audio/mpeg" android:scheme="http" . . . />

    . . .

</intent-filter>
```

每个数据<data>元素可以指定一个 URI 和一个数据类型（MIME 媒体类型）。有一些单独的属性-模式，主机，端口和路径-URI 的每个部分：

scheme://host:port/path

比如，在下面的 URI 里面，

content://com.example.project:200/folder/subfolder/etc

模式是"内容", 主机是"com.example.project", 端口是"200", 路径是

"folder/subfolder/etc"。主机和端口一起组成 URI 鉴权 (*authority*) ; 如果未指定主机, 端口会被忽略。

这些属性都是可选的, 但彼此有依赖关系: 一个授权要有意义, 必须指定一个模式。一个路径要有意义, 必须同时指定模式和鉴权。

当一个意图对象中的 URI 被用来和一个过滤器中的 URI 规格比较时, 它实际上比较的是上面提到的 URI 的各个部分。比如, 如果过滤器仅指定了一个模式, 所有那个模式的 URIs 和这个过滤器相匹配; 如果过滤器指定了一个模式、鉴权但没有路径, 所有相同模式和鉴权的 URIs 可以匹配上, 而不管它们的路径; 如果过滤器指定了一个模式、鉴权和路径, 只有相同模式、鉴权和路径的 URIs 可以匹配上。当然, 一个过滤器中的路径规格可以包含通配符, 这样只需要部分匹配即可。

数据<data>元素的类型属性指定了数据的 MIME 类型。这在过滤器里比在 URI 里更为常见。意

图对象和过滤器都可以使用一个"\*"通配符指定子类型字段-比如, "text/\*"或者"audio/\*"-指示任何匹配的子类型。

数据测试同时比较意图对象和过滤器中指定的 URI 和数据类型。规则如下:

- a. 一个既不包含 URI 也不包含数据类型的意图对象仅在过滤器也同样没有指定任何 URIs 和数据类型的情况下才能通过测试。
- b. 一个包含 URI 但没有数据类型的意图对象仅在它的 URI 和一个同样没有指定数据类型的过滤器里的 URI 匹配时才能通过测试。这通常发生在类似于 mailto: 和 tel: 这样的 URIs 上: 它们并不引用实际数据。
- c. 一个包含数据类型但不包含 URI 的意图对象仅在这个过滤器列举了同样的数据类型而且也没有指定一个 URI 的情况下才能通过测试。
- d. 一个同时包含 URI 和数据类型 (或者可从 URI 推断出数据类型) 的意图对象可以通过测试, 如果它的类型和过滤器中列举的类型相匹配的话。如果它的 URI 和这个过滤器中的一个 URI 相匹配或者它有一个内容 content: 或者文件 file: URI 而且这个过滤器没有指定一个 URI, 那么它也能通过测试。换句话说, 一个组件被假定为支持 content: 和 file: 数据如果它的过滤器仅列举了一个数据类型。

如果一个意图可以通过不止一个活动或服务的过滤器，用户可能会被询问要激活那个组件。如果没有发现目标对象将会出现异常。

## 通常情况 Common cases

上面描述的数据测试的最后一个规则（d），表达了这样一个期望即组件能够从文件或内容提供者中获取本地数据。因此，它们的过滤器可以只列举一个数据类型而不需要显式的命名 **content:**和 **file:**模式。这是一个典型情况。比如，一个如下的数据<data>元素，告诉 **Android** 这个组件能从内容提供者获取图片数据并显示：

```
<data android:type="image/*" />
```

既然大多数可用数据是通过内容提供者来分发，那么过滤器最通常的配置就是指定一个数据类型而不指定 **URI**。另外一个通用的配置是带有一个模式和数据类型的过滤器。比如，一个如下的数据<data>元素告诉 **Android** 可以从网络获取视频数据并显示：

```
<data android:scheme="http" android:type="video/*" />
```

比如，想一下，当用户点击网页上的一个链接时浏览器做了什么。它首先试图去显示这个数据（如果这个链接指向一个 **HTML** 页面）。如果它不能显示这个数据，它会把一个显式意图和一个模式、数据类型组成整体然后尝试启动一个可以处理这个工作的活动。如果没有接受者，它将要求下载管理器来下载数据。这让它处于内容提供者的控制下，以便一个潜在的更大的活动池可以做出反应。

大多数应用程序同样有一个方法去启动刷新，而不包含任何特定数据的引用。能初始化应用程序的活动拥有指定动作为"**android.intent.action.MAIN**"的过滤器。如果它们表述在应用程序启动器中，那它们同样指定了"**android.intent.category.LAUNCHER**"类别：

```
<intent-filter . . . >
```

```
    <action android:name="code android.intent.action.MAIN" />
```

```
    <category android:name="code android.intent.category.LAUNCHER" />
```

```
</intent-filter>
```

## 使用意图匹配 Using intent matching

通过意图过滤器匹配的意图不仅是为了发现要激活的目标组件，而且为了发现这个设备上的一系列组件的某些东西。比如，**Android** 系统通过查找符合条件的所有活动（需要包含指定了动作

"android.intent.action.MAIN"和"android.intent.category.LAUNCHER"类别的意图过滤器，如前面章节所述）来产生应用程序启动器，也就是用户可用程序的前置屏幕。然后它显示在这个启动器里的这些活动的图标和标签。类似的，它通过查找其过滤器配有"android.intent.category.HOME"元素的活动来发现桌面。

你的应用程序可以用类似的方式使用意图匹配。[PackageManager](#)有一系列的查询 `query...()` 方法可以接收一个特定的意图，以及相似的一个解析 `resolve...()` 方法系列可以确定应答意图的最佳组件。比如，[queryIntentActivities\(\)](#) 返回一个所有活动的列表，而 [queryIntentServices\(\)](#) 返回一个类似的服务列表。两个方法都不会激活组件；它们仅仅列举能应答的。对于广播接收者，有一个类似的方法 [queryBroadcastReceivers\(\)](#)。

## 数据存储 Data Storage

### 概览 Storage quickview

- ✧ 系统偏好：快速，轻量级存储
- ✧ 文件：存储到设备内部或可移动闪存
- ✧ 数据库：任意的结构化存储
- ✧ 支持基于网络的存储

一个典型的桌面操作系统提供了一个通用文件系统使得任何应用程序能够使用它来存储文件，这些文件可以被其它应用程序读取（可能有访问权限的设置）。Android 使用一个不同的系统：在 Android 上，所有应用程序数据（包括文件）都是该应用程序私有的。

不过，Android 同样提供了一个应用程序向其它应用程序暴露其私有数据的基本方式-通过内容提供者。内容提供者是应用程序的可选组件，用来暴露该应用程序数据的读写接口，且遵循任何可能引入的约定。内容提供者实现了一个用来请求和修改数据的基本语法，一个读取返回数据的基本机制。Android 为基础数据类型如图像，音频和视频文件以及个人联系人信息提供了许多内容提供者。想要了解更多如何使用内容提供器的信息，请参见一篇单独的文章：内容提供者（[Content Providers](#)）。

无论你是否想把应用程序数据输出给别人，你总需要有一个方法来保存它。Android 提供了下面 4 种机制来保存和获取数据：系统偏好 [Preferences](#)，文件 [Files](#)，数据库 [Databases](#) 和网络 [Network](#)。

### 系统偏好 Preferences

系统偏好是一个用来存放和提取元数据类型键-值对的轻量级机制。它通常用来存放应用程序偏好，例如一个应用程序启动时所使用的默认问候或文本字体。通过调用 [Context.getSharedPreferences\(\)](#) 来读写数值。如果你想分享给应用程序中的其它组件，可以为你的偏好集分配一个名字，或者使用没有名字的 [Activity.getPreferences\(\)](#) 方法来保持对于该调用程序的私有性。你不能跨应用程序共享偏好（除了使用一个内容提供者）。

下面是一个为计算器设置按键静音模式的例子：

```
import android.app.Activity;
import android.content.SharedPreferences;

public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    . . .

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        . . .

        // Restore preferences
        SharedPreferences settings = getSharedPreferences(PREFS_NAME,
0);

        boolean silent = settings.getBoolean("silentMode", false);
        setSilent(silent);
    }

    @Override
    protected void onStop() {
        super.onStop();

        // Save user preferences. We need an Editor object to
        // make changes. All objects are from android.context.Context
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
        SharedPreferences.Editor editor = settings.edit();
        editor.putBoolean("silentMode", mSilentMode);

        // Don't forget to commit your edits!!!
        editor.commit();
    }
}
```

```
}  
  
}
```

## 文件 Files

你可以直接在移动设备或可移动存储媒介里存放文件。缺省情况下，其它应用程序不能访问这些文件。

为了从文件中读取数据，可调用 [Context.openFileInput\(\)](#) 方法并传递本地文件名和文件路径给它。该方法返回一个标准的 Java [FileInputStream](#) 对象。为了写一个文件，可调用 [Context.openFileOutput\(\)](#) 并传递文件名和路径，这个方法也返回 [FileOutputStream](#) 对象。从另外的应用程序中调用这些方法将不起作用，你只能访问本地文件。

如果你有一个静态文件需要在编译时打包进应用程序，你可以保存该文件在你项目中 `res/raw/myDataFile`，然后使用 [Resources.openRawResource \(R.raw.myDataFile\)](#) 打开它。该方法返回一个 [InputStream](#) 对象，你可以使用它读取文件数据。

## 数据库 Databases

Android API 包含对创建和使用 SQLite 数据库的支持。每个数据库都是创建它的应用程序所私有的。

这个 [SQLiteDatabase](#) 对象代表了一个数据库并包含与之交互的方法-生成查询和管理数据。为了创建数据库，调用 [SQLiteDatabase.create\(\)](#) 并同时子类化 [SQLiteOpenHelper](#)。

作为支持 SQLite 数据库的一部分，Android 暴露了数据库管理函数，这让你可以存储复杂的数据集合，这些数据被包装到有用的对象里。比如，Android 为联系人信息定义了一个数据类型；它由很多字段组成，其中包括姓，名（字符串），地址信息和电话号码（也是字符串），照片（位图图像），以及更多其它个人信息。

Android 装载了 sqlite3 数据工具，利用这些工具你可以浏览表内容，运行 SQL 命令，并执行 SQLite 数据库上的其它有用的函数。请查阅[检查数据库 \(Examine databases \(sqlite3\)\)](#) 得知如何运行这个程序。

所有的数据库，SQLite 以及其它，都被保存在设备如下目录里：

```
/data/data/package_name/databases.
```

讨论创建多少表格，包含哪些字段以及它们之间如何连接超出了本文的范围，不过 Android 并没有引入任何在标准 SQLite 概念之外的限制。我们确实推荐包含一个自增长数值的关键域，作为一个唯一 ID 用来快速查找一个记录。这对于私有数据并不必要，但如果你实现了一个内容提供者，你必须包含这样一个唯一 ID 字段。请参见 [Content Providers](#) 文档以获取关于该字段的更多信息，以及 `NotePadProvider` 类（在 `NotePad` 例子代码里）中创建和组装一个新数据库的方法。你创建的任何数据库都将可以通过名字被应用程序中其它的类访问，但不能从应用程序外部访问。

## 网络 Network

你也可以使用网络来存放和获取数据（当它可用时）。要进行网络操作，可使用如下程序包中的类：

- [java.net.\\*](#)
- [android.net.\\*](#)

## 内容提供者 Content Providers

内容提供者用来存放和获取数据并使这些数据可以被所有的应用程序访问。它们是应用程序之间共享数据的唯一方法；不存在所有 Android 软件包都能访问的公共储存区域。

Android 为常见数据类型（音频，视频，图像，个人联系人信息，等等）装载了很多内容提供者。你可以看到在 [android.provider](#) 包里列举了一些。你还能查询这些提供者包含了什么数据（尽管，对某些提供者，你必须获取合适的权限来读取数据）。

如果你想公开你自己的数据，你有两个选择：你可以创建你自己的内容提供者（一个 [ContentProvider](#) 子类）或者你可以给已有的提供者添加数据-如果存在一个控制同样类型数据的内容提供者且你拥有写的权限。

这篇文档是一篇关于如何使用内容提供者的简介。先是一个简短的基础知识讨论，然后探究如何查询一个内容提供者，如何修改内容提供者控制的数据，以及如何创建你自己的内容提供者。

### 内容提供器的基础知识 Content Provider Basics

内容提供者究竟如何在表层下保存它的数据依赖于它的设计者。但是所有的内容提供者实现了一个公共的接口来查询这个提供者和返回结果-增加，替换，和删除数据也是一样。

这是一个客户端直接使用的接口，一般是通过 [ContentResolver](#) 对象。你可以通过 [getContentResolver\(\)](#) 从一个活动或其它应用程序组件的实现里获取一个 ContentResolver:

```
ContentResolver cr = getContentResolver();
```

然后你可以使用这个 ContentResolver 的方法来和你感兴趣的任何内容提供者交互。

当初始化一个查询时，Android 系统识别查询目标的内容提供者并确保它正在运行。系统实例化所有的 ContentProvider 对象；你从来不需要自己做。事实上，你从来不会直接处理 ContentProvider 对象。通常，对于每个类型的 ContentProvider 只有一个简单的实例。但它能够和不同应用程序和进程中的多个 ContentProvider 对象通讯。进程间的交互通过 ContentResolver 和 ContentProvider 类处理。

### 数据模型 *The data model*

内容提供者以数据库模型上的一个简单表格形式暴露它们的数据，这里每一个行是一个记录，每一列是特别类型和含义的数据。比如，关于个人信息以及他们的电话号码可能会以下面的方式展示：

_ID	NUMBER	NUMBER_KEY	LABEL	NAME	TYPE
13	(425) 555 6677	425 555 6677	Kirkland office	Bully Pulpit	TYPE_WORK
44	(212) 555-1234	212 555 1234	NY apartment	Alan Vain	TYPE_HOME

45	(212) 555-6657	212 555 6657	Downtown office	Alan Vain	TYPE_MOBILE
53	201.555.4433	201 555 4433	Love Nest	Rex Cars	TYPE_HOME

每个记录包含一个数字的 `_ID` 字段用来唯一标识这个表格里的记录。`IDs` 可以用来匹配相关表格中的记录-比如，用来在一张表格中查找个人电话号码并在另外一张表格中查找这个人的照片。

一个查询返回一个 [Cursor](#) 对象可在表格和列中移动来读取每个字段的内容。它有特定的方法来读取每个数据类型。所以，为了读取一个字段，你必须了解这个字段包含了什么数据类型。（后面会更多的讨论查询结果和游标 `Cursor` 对象）。

### 唯一资源标识符 *URIs*

每个内容提供者暴露一个公开的 `URI`（以一个 [Uri](#) 对象包装）来唯一的标识它的数据集。一个控制多个数据集（多个表）的内容提供者对每一个数据集暴露一个单独的 `URI`。所有提供器的 `URIs` 以字符串 `"content://"` 开始。这个 `content:` 形式表明了这个数据正被一个内容提供者控制着。

如果你正准备定义一个内容提供者，为了简化客户端代码和使将来的升级更清楚，最好也为它的 `URI` 定义一个常量。`Android` 为这个平台所有的提供者定义了 `CONTENT_URI` 常量。比如，匹配个人电话号码的表的 `URI` 和包含个人照片的表的 `URI` 是：（均由联系人 `Contacts` 内容提供者控制）

```
android.provider.Contacts.Phones.CONTENT_URI
android.provider.Contacts.Photos.CONTENT_URI
```

类似的，最近电话呼叫的表和日程表条目的 `URI` 如下：Similarly, the URIs for the table of recent phone calls and the table of calendar entries are:

```
android.provider.CallLog.Calls.CONTENT_URI
android.provider.Calendar.CONTENT_URI
```

这个 `URI` 常量被使用在和这个内容提供者所有的交互中。每个 [ContentResolver](#) 方法采用这个 `URI` 作为它的第一个参数。正是它标识了 `ContentResolver` 应该和哪个内容提供者对话以及这个内容提供者的哪张表格是其目标。

### 查询一个内容提供者 *Querying a Content Provider*

你需要三方面的信息来查询一个内容提供者：

- 用来标识内容提供者的 `URI`
- 你想获取的数据字段的名称
- 这些字段的数据类型

如果你想查询某一条记录，你同样需要那条记录的 `ID`。

## 生成查询 *Making the query*

你可以使用 [ContentResolver.query\(\)](#) 方法或者 [Activity.managedQuery\(\)](#) 方法来查询一个内容提供者。两种方法使用相同的参数序列，而且都返回一个 `Cursor` 对象。不过，`managedQuery()` 使得活动需要管理这个游标的生命周期。一个被管理的游标处理所有的细节，比如当活动暂停时卸载自身，而活动重新启动时重新查询它自己。你可以让一个活动开始管理一个尚未被管理的游标对象，通过如下调用：

[Activity.startManagingCursor\(\)](#)。

无论 [query\(\)](#) 还是 [managedQuery\(\)](#)，它们的第一个参数都是内容提供器的 `URI-CONTENT_URI` 常量用来标识某个特定的 `ContentProvider` 和数据集（参见前面的 [URIs](#)）。

为了限制只对一个记录进行查询，你可以在 `URI` 后面扩展这个记录的 `_ID` 值-也就是，在 `URI` 路径部分的最后加上匹配这个 `ID` 的字符串。比如，如果 `ID` 是 `23`，那么 `URI` 会是：

```
content://. . . ./23
```

有一些辅助方法，特别是 [ContentUris.withAppendedId\(\)](#) 和 [Uri.withAppendedPath\(\)](#)，使得为 `URI` 扩展一个 `ID` 变得简单。所以，比如，如果你想在联系人数据库中查找记录 `23`，你可能需要构造如下的查询语句：

```
import android.provider.Contacts.People;
import android.content.ContentUris;
import android.net.Uri;
import android.database.Cursor;

// Use the ContentUris method to produce the base URI for the contact
// with _ID == 23.
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);

// Alternatively, use the Uri method to produce the base URI.
// It takes a string rather than an integer.
Uri myPerson = Uri.withAppendedPath(People.CONTENT_URI, "23");

// Then query for this specific record:
Cursor cur = managedQuery(myPerson, null, null, null, null);
```

[query\(\)](#) 和 [managedQuery\(\)](#) 方法的其它参数限定了更多的查询细节。如下：

- 应该返回的数据列的名字。`null` 值返回所有列。否则只有列出名字的列被返回。所有这个平台的内容提供者都为它们的列定义了常量。比如，[android.provider.Contacts.Phones](#) 类对前面说明过的通讯录中各个列的名字定义了常量 `ID`, `NUMBER`, `NUMBER_KEY`, `NAME`, 等等。
- 指明返回行的过滤器，以一个 `SQL WHERE` 语句格式化。`null` 值返回所有行。（除非这个 `URI` 限定只查询一个单独的记录）。
- 选择参数
- 返回行的排列顺序，以一个 `SQL ORDER BY` 语句格式化（不包含 `ORDER BY` 本身）。`null` 值表示以该表格的默认顺序返回，有可能是无序的。

让我们看一个查询的例子吧，这个查询获取一个联系人名字和首选电话号码列表：

```
import android.provider.Contacts.People;
import android.database.Cursor;

// Form an array specifying which columns to return.
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};

// Get the base URI for the People table in the Contacts content
// provider.
Uri contacts = People.CONTENT_URI;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection, // Which columns to return
    null,       // Which rows to return (all rows)
    null,       // Selection arguments (none)
    // Put the results in ascending order by name
    People.NAME + " ASC");
```

这个查询从联系人内容提供者中获取了数据。它得到名字，首选电话号码，以及每个联系人的唯一记录 ID。同时它在每个记录的 `_COUNT` 字段告知返回的记录数目。

列名的常量被定义在不同的接口中- `_ID` 和 `_COUNT` 定义在 [BaseColumns](#) 里， `NAME` 在 [PeopleColumns](#) 里， `NUMBER` 在 [PhoneColumns](#) 里。 [Contacts.People](#) 类已经实现了这些接口，这就是为什么上面的代码实例只需要使用类名就可以引用它们的原因。

### 查询的返回结果 *What a query returns*

一个查询返回零个或更多数据库记录的集合。列名，默认顺序，以及它们的数据类型是特定于每个内容提供器的。但所有提供者都有一个 `_ID` 列，包含了每个记录的唯一 ID。另外所有的提供者都可以通过返回 `_COUNT` 列告知记录数目。它的数值对于所有的行而言都是一样的。

下面是前述查询的返回结果的一个例子：

<code>_ID</code>	<code>_COUNT</code>	<code>NAME</code>	<code>NUMBER</code>
44	3	Alan Vain	212 555 1234
13	3	Bully Pulpit	425 555 6677
53	3	Rex Cars	201 555 4433

获取到的数据通过一个游标 [Cursor](#) 对象暴露出来，通过游标你可以在结果集中前后浏览。你只能用这个对象来读取数据。如果想增加，修改和删除数据，你必须使用一个 `ContentResolver` 对象。

### 读取查询所获数据 *Reading retrieved data*

查询返回的游标对象可以用来访问结果记录集。如果你通过指定的一个 ID 来查询，这个集合将只有一个值。否则，它可以包含多个数值。（如果没有匹配结果，那还可能是空的。）你可以从表格中的特定字段读取数据，但你必须知道这个字段的数据类型，因为这个游标对象对于每种数据类型都有一个单独的读取方法- 比如 [getString\(\)](#)、[getInt\(\)](#)，和 [getFloat\(\)](#)。（不过，对于大多数类型，如果你调用读取字符串的方法，游标对象将返回给你这个数据的字符串表示。）游标可以让你按列索引请求列名，或者按列名请求列索引。

下面的代码片断演示了如何从前述查询结果中读取名字和电话号码：

```
import android.provider.Contacts.People;

private void getColumnData(Cursor cur){
    if (cur.moveToFirst()) {

        String name;

        String phoneNumber;
```

```

        int nameColumn = cur.getColumnIndex(People.NAME);

        int phoneColumn = cur.getColumnIndex(People.NUMBER);

        String imagePath;

        do {

            // Get the field values

            name = cur.getString(nameColumn);

            phoneNumber = cur.getString(phoneColumn);

            // Do something with the values.

            ...

        } while (cur.moveToNext());

    }
}

```

如果一个查询可能返回二进制数据，比如一个图像或声音，这个数据可能直接被输入到表格或表格条目中也可能是一个 `content: URI` 的字符串可用来获取这个数据，一般而言，较小的数据（例如，20 到 50K 或更小）最可能被直接存放到表格中，可以通过调用 [Cursor.getBlob\(\)](#) 来获取。它返回一个字节数组。

如果这个表格条目是一个 `content: URI`，你不该试图直接打开和读取该文件（会因为权限问题而失败）。相反，你应该调用 [ContentResolver.openInputStream\(\)](#) 来得到一个 [InputStream](#) 对象，你可以使用它来读取数据。

## 修改数据 **Modifying Data**

保存在内容提供者中的数据可以通过下面的方法修改：

- 增加新的记录
- 为已有的记录添加新的数据
- 批量更新已有记录
- 删除记录

所有的数据修改操作都通过使用 [ContentResolver](#) 方法来完成。一些内容提供者对写数据需要一个比读数据更强的权限约束。如果你没有一个内容提供器的写权限，这个 `ContentResolver` 方法会失败。

## 增加记录 *Adding records*

想要给一个内容提供者增加一个新的记录，第一步是在 [ContentValues](#) 对象里构建一个键-值对映射，这里每个键和内容提供器的一个列名匹配而值是新记录中那个列期望的值。然后调用 [ContentResolver.insert\(\)](#) 并传递给它提供器的 URI 和这个 ContentValues 映射图。这个方法返回新记录的 URI 全名-也就是，内容提供器的 URI 加上该新记录的扩展 ID。你可以使用这个 URI 来查询并得到这个新记录上的一个游标，然后进一步修改这个记录。下面是一个例子：

```
import android.provider.Contacts.People;
import android.content.ContentResolver;
import android.content.ContentValues;

ContentValues values = new ContentValues();

// Add Abraham Lincoln to contacts and make him a favorite.
values.put(People.NAME, "Abraham Lincoln");
// 1 = the new contact is added to favorites
// 0 = the new contact is not added to favorites
values.put(People.STARRED, 1);

Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

## 增加新值 *Adding new values*

一旦记录已经存在，你就可以添加新的信息或修改已有信息。比如，上例中的下一步就是添加联系人信息-如一个电话号码或一个即时通讯 IM 或电子邮箱地址-到新的条目中。

在联系人数据库中增加一条记录的最佳途径是在该记录 URI 后扩展表名，然后使用这个修正的 URI 来添加新的数据值。为此，每个联系人表暴露一个 `CONTENT_DIRECTORY` 常量的表名。下面的代码继续之前的例子，为上面刚刚创建的记录添加一个电话号码和电子邮件地址：

```
Uri phoneUri = null;
Uri emailUri = null;

// Add a phone number for Abraham Lincoln. Begin with the URI for
// the new record just returned by insert(); it ends with the _ID
// of the new record, so we don't have to add the ID ourselves.
// Then append the designation for the phone table to this URI,
// and use the resulting URI to insert the phone number.
```

```

phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);

values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);

// Now add an email address in the same way.
emailUri = Uri.withAppendedPath(uri,
    People.ContactMethods.CONTENT_DIRECTORY);

values.clear();
// ContactMethods.KIND is used to distinguish different kinds of
// contact methods, such as email, IM, etc.
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE,
    People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);

```

你可以通过调用接收字节流的 [ContentValues.put\(\)](#) 版本来把少量的二进制数据放到一张表格里去。这对于像小图标或短小的音频片断这样的数据是可行的。但是，如果你有大量二进制数据需要添加，比如一张相片或一首完整的歌曲，则需要把该数据的 **content: URI** 放到表里然后以该文件的 **URI** 调用 [ContentResolver.openOutputStream\(\)](#) 方法。（这导致内容提供者把数据保存在一个文件里并且记录文件路径在这个记录的一个隐藏字段中。）

考虑到这一点，[MediaStore](#) 内容提供者，这个用来分发图像，音频和视频数据的主内容提供者，利用了一个特殊的约定：用来获取关于这个二进制数据的元信息的 `query()` 或 `managedQuery()` 方法使用的 **URI**，同样可以被 `openInputStream()` 方法用来数据本身。类似的，用来把元信息放进一个 **MediaStore** 记录里的 `insert()` 方法使用的 **URI**，同样可以被 `openOutputStream()` 方法用来在那里存放二进制数据。下面的代码片断说明了这个约定：

```

import android.provider.MediaStore.Images.Media;
import android.content.ContentValues;
import java.io.OutputStream;

// Save the name and description of an image in a ContentValues map.

```

```

ContentValues values = new ContentValues(3);
values.put(Media.DISPLAY_NAME, "road_trip_1");
values.put(Media.DESCRPTION, "Day 1, trip to Los Angeles");
values.put(Media.MIME_TYPE, "image/jpeg");

// Add a new record without the bitmap, but with the values just set.
// insert() returns the URI of the new record.
Uri uri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI,
values);

// Now get a handle to the file for that record, and save the data
into it.

// Here, sourceBitmap is a Bitmap object representing the file to
save to the database.

try {
    OutputStream outputStream =
getContentResolver().openOutputStream(uri);

    sourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outputStream);

    outputStream.close();
} catch (Exception e) {
    Log.e(TAG, "exception while writing image", e);
}

```

### 批量更新记录 *Batch updating records*

要批量更新一组记录（例如，把所有字段中的"NY"改为"New York"），可以传以需要改变的列和值参数来调用 [ContentResolver.update\(\)](#) 方法。

### 删除一个记录 *Deleting a record*

要删除单个记录，可以传以一个特定行的 **URI** 参数来调用 [ContentResolver.delete\(\)](#) 方法。

要删除多行记录，可以传以需要被删除的记录类型的 **URI** 参数来调用 [ContentResolver.delete\(\)](#) 方法（例如，`android.provider.Contacts.People.CONTENT_URI`）以及一个 **SQL WHERE** 语句来定义哪些行要被删除。（**小心：**如果你想删除一个通用类型，你得确保包含一个合法的 *WHERE* 语句，否则你可能删除比设想的多得多的记录！）

### 创建一个内容提供者 **Creating a Content Provider**

要创建一个内容提供者，你必须：

- 建立一个保存数据的系统。大多数内容提供者使用 **Android** 的文件储存方法或 **SQLite** 数据库来存放它们的数据，但是你可以用任何你想要的方式来存放数据。**Android** 提供 [SQLiteOpenHelper](#) 类来帮助你创建一个数据库以及 [SQLiteDatabase](#) 类来管理它。
- 扩展 [ContentProvider](#) 类来提供数据访问接口。
- 在清单 **manifest** 文件中为你的应用程序声明这个内容提供者 (**AndroidManifest.xml**)。

下面的章节对后来两项任务有一些标注。

## 扩展 *ContentProvider* 类 *Extending the ContentProvider class*

你可以定义一个 [ContentProvider](#) 子类来暴露你的数据给其它使用符合 **ContentResolver** 和游标 **Cursor** 对象约定的应用程序。理论上，这意味需要实现 6 个 **ContentProvider** 类的抽象方法：

```
query()
insert()
update()
delete()
getType()
onCreate()
```

**query()** 方法必须返回一个游标 [Cursor](#) 对象可以用来遍历请求数据，游标本身是一个接口，但 **Android** 提供了一些现成的 **Cursor** 对象给你使用。例如，[SQLiteCursor](#) 可以用来遍历 **SQLite** 数据库。你可以通过

调用任意的 [SQLiteDatabase](#) 类的 **query()** 方法得到它。还有一些其它的游标实现-比如 [MatrixCursor](#)-用来访问没有存放在数据库中的数据。

因为这些内容提供器的方法可以从不同的进程和线程的各个 **ContentResolver** 对象中调用，所以它们必须以线程安全的方式来实现。

周起见，当数据被修改时，你可能还需要调用 [ContentResolver.notifyChange\(\)](#) 方法来通知侦听者。

除了定义子类以外，你应该还需要采取其它一些步骤来简化客户端的工作和让这个类更容易被访问：

- 定义一个 `public static final Uri` 命名为 **CONTENT\_URI**。这是你的内容提供者处理的整个 **content: URI** 的字符串。你必须为它定义一个唯一的字符串。最佳方案是使用这个内容提供器的全称 (**fully qualified**) 类名 (小写)。因此，例如，一个 **TransportationProvider** 类可以定义如下：

```
public static final Uri CONTENT_URI =
Uri.parse("content://com.example.codelab.transportationprovider");
```

如果这个内容提供者有子表，那么为每个子表也都定义 **CONTENT\_URI** 常量。这些 **URIs** 应该全部拥有相同的权限 (既然这用来识别内容提供者)，只能通过它们的路径加以区分。例如：

```
content://com.example.codelab.transportationprovider/train
content://com.example.codelab.transportationprovider/air/domestic
content://com.example.codelab.transportationprovider/air/international
```

请查阅本文最后部分的 [Content URI Summary](#) 以对 **content: URIs** 有一个总体的了解。

- 定义内容提供者返回给客户端的列名。如果你正在使用一个底层数据库，这些列名通常和 SQL 数据库列名一致。同样还需要定义公共的静态字符串常量用来指定查询语句以及其它指令中的列。

确保包含一个名为"`_id`"（常量 `_ID`）的整数列来返回记录的 IDs。你应该有这个字段而不管有没有其它字段（比如 `URL`），这个字段在所有的记录中是唯一的。如果你在使用 SQLite 数据库，这个 `_ID` 字段应该是下面的类型：

```
INTEGER PRIMARY KEY AUTOINCREMENT
```

其中 `AUTOINCREMENT` 描述符是可选的。但是没有它，SQLite 的 ID 数值字段会在列中已存在的最大数值的基础上增加到下一个数字。如果你删除了最后的行，那么下一个新加的行会和这个删除的行有相同的 ID。 `AUTOINCREMENT` 可以避免这种情况，它让 SQLite 总是增加到下一个最大的值而不管有没有删除。

- 在文档中谨慎的描述每个列的数据类型。客户端需要这些信息来读取数据。
- 如果你正在处理一个新的数据类型，你必须定义一个新的 MIME 类型在你的 `ContentProvider.getType()` 实现里返回。这个类型部分依赖于提交给 `getType()` 的 `content: URI` 参数是否对这个请求限制了特定的记录。有一个 MIME 类型是给单个记录用的，另外一个给多记录用。使用 `Uri` 方法来帮助判断哪个是正在被请求的。下面是每个类型的一般格式：

✧ 对于单个记录: `vnd.android.cursor.item/vnd.yourcompanyname.contenttype`

比如，一个火车记录 122 的请求，URI 如下

```
content://com.example.transportationprovider/trains/122
```

可能会返回这个 MIME 类型：

```
vnd.android.cursor.item/vnd.example.rail
```

✧ 对于多个记录: `vnd.android.cursor.dir/vnd.yourcompanyname.contenttype`

比如，一个所有火车记录的请求，URI 如下

```
content://com.example.transportationprovider/trains
```

可能会返回这个 MIME 类型：

```
vnd.android.cursor.dir/vnd.example.rail
```

- 如果你想暴露过于庞大而无法放在表格里的字节数据-比如一个大的位图文件-这个给客户端暴露数据的字段事实上应该包含一个 **content: URI** 字符串。这个字段给了客户端数据访问接口。这个记录应该有另外的一个字段，名为"**\_data**"，列出了这个文件在设备上的准确路径。这个字段不能被客户端读取，而要通过 **ContentResolver**。客户端将在这个包含 **URI** 的用户侧字段上调用 **ContentResolver.openInputStream()** 方法。**ContentResolver** 会请求那个记录的"**\_data**"字段，而且因为它有比客户端更高的许可权，它应该能够直接访问那个文件并返回给客户端一个包装的文件读取接口。

自定义内容提供器的实现的一个例子，参见 SDK 附带的 Notepad 例程中的 **NodePadProvider** 类。

## 声明内容提供器 **Declaring the content provider**

为了让 **Android** 系统知道你开发的内容提供器，可以用在应用程序的 **AndroidManifest.xml** 文件中以 **<provider>** 元素声明它。未经声明的内容提供器对 **Android** 系统不可见。

名字属性是 **ContentProvider** 子类的全称名（fully qualified name）。权限属性是标识提供器的 **content: URI** 的权限认证部分。例如如果 **ContentProvider** 子类是 **AutoInfoProvider**，那么 **<provider>** 元素可能如下：

```
<provider name="com.example.autos.AutoInfoProvider"
          authorities="com.example.autos.autoinfoprovider"
          . . . />
</provider>
```

请注意到这个权限属性忽略了 **content: URI** 的路径部分。例如，如果 **AutoInfoProvider** 为各种不同的汽车或制造商控制着各个子表，Note that the authorities attribute omits the path part of a content: URI. For example, if **AutoInfoProvider** controlled subtables for different types of autos or different manufacturers,

```
content://com.example.autos.autoinfoprovider/honda
content://com.example.autos.autoinfoprovider/gm/compact
content://com.example.autos.autoinfoprovider/gm/suv
```

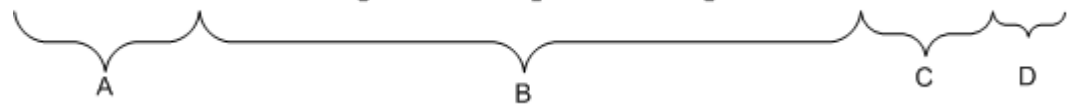
这些路径将不会在 **manifest** 里声明。权限是用来识别提供器的，而不是路径；你的提供器能以任何你选择的方式来解释 **URI** 中的路径部分。

其它 **<provider>** 属性可以设置数据读写许可，提供可以显示给用户的图标和文本，启用或禁用这个提供器，等等。如果数据不需要在多个内容提供器的运行版本中同步则可以把 **multiprocess** 属性设置成 **"true"**。这使得在每个客户进程中都有一个提供器实例被创建，而无需执行 **IPC** 调用。

## **Content URI** 总结

这里回顾一下 **content URI** 的重要内容：

`content://com.example.transportationprovider/trains/122`



A. 标准前缀表明这个数据被一个内容提供者所控制。它不会被修改。

B. URI 的权限部分；它标识这个内容提供者。对于第三方应用程序，这应该是一个全称类名（小写）以确保唯一性。权限在<provider>元素的权限属性中进行声明：

```
<provider name=".TransportationProvider"
          authorities="com.example.transportationprovider"
          . . . >
```

C. 用来判断请求数据类型的路径。这可以是 0 或多个段长。如果内容提供者只暴露了一种数据类型（比如，只有火车），这个分段可以没有。如果提供者暴露若干类型，包括子类型，那它可以是多个分段长-例如，提供"land/bus", "land/train", "sea/ship", 和"sea/submarine"这 4 个可能的值。

D. 被请求的特定记录的 ID，如果有的话。这是被请求记录的\_ID 数值。如果这个请求不局限于单个记录，这个分段和尾部的斜线会被忽略：

`content://com.example.transportationprovider/trains`

## 清单文件 **The AndroidManifest.xml File**

每个应用程序都有一个 **AndroidManifest.xml** 文件（一定是这个名字）在它的根目录里。这个清单文件给 Android 系统提供了关于这个应用程序的基本信息，系统在能运行任何程序代码之前必须知道这些信息。

**AndroidManifest.xml** 主要包含以下功能：

- 命名应用程序的 **Java** 包，这个包名用来唯一标识应用程序；

- 描述应用程序的组件-活动，服务，广播接收者，以及组成应用程序的内容提供者；对实现每个组件和公布其能力（比如，能处理哪些意图消息）的类进行命名。这些声明使得 **Android** 系统了解这些组件以及在什么条件下可以被启动；
- 决定应用程序组件运行在哪个进程里面；
- 声明应用程序所必须具备的权限，用以访问受保护的部分 **API**，以及和其它应用程序交互；
- 声明应用程序其他的必备权限，用以组件之间的交互；
- 列举测试设备 [Instrumentation](#) 类，用来提供应用程序运行时所需的环境配置及其他信息，这些声明只在程序开发和测试阶段存在，发布前将被删除；
- 声明应用程序所要求的 **Android API** 的最低版本级别；
- 列举 **application** 所需要链接的库；

## 清单文件结构 **Structure of the Manifest File**

下面的图表显示了清单文件的基本结构以及它能包含的所有元素。每个元素，和它所有的属性，在一个单独的文件中完整描述。要查看任何元素的细节信息，可在图表下方的以字符序排列的元素列表中点击其元素名称。

```
<?xml version="1.0" encoding="utf-8"?>
```

[<manifest>](#)

[<uses-permission />](#)

[<permission />](#)

[<permission-tree />](#)

[<permission-group />](#)

[<instrumentation />](#)

[<uses-sdk />](#)

[<application>](#)

[<activity>](#)

[<intent-filter>](#)

[<action />](#)

[<category />](#)

[<data />](#)

```

    </intent-filter>

    <meta-data />

</activity>

<activity-alias>

    <intent-filter> ... </intent-filter>

    <meta-data />

</activity-alias>

<service>

    <intent-filter> ... </intent-filter>

    <meta-data/>

</service>

<receiver>

    <intent-filter> ... </intent-filter>

    <meta-data />

</receiver>

<provider>

    <grant-uri-permission />

    <meta-data />

</provider>

<uses-library />

<uses-configuration />

</application>

</manifest>
```

所有清单文件中可能出现的元素按字符序排列如下。只有这些元素是合法的，你不能添加自己的元素或属性：

```
<action>
<activity>
```

[<activity-alias>](#)  
[<application>](#)  
[<category>](#)  
[<data>](#)  
[<grant-uri-permission>](#)  
[<instrumentation>](#)  
[<intent-filter>](#)  
[<manifest>](#)  
[<meta-data>](#)  
[<permission>](#)  
[<permission-group>](#)  
[<permission-tree>](#)  
[<provider>](#)  
[<receiver>](#)  
[<service>](#)  
[<uses-configuration>](#)  
[<uses-library>](#)  
[<uses-permission>](#)  
[<uses-sdk>](#)

## 文件约定 **File Conventions**

下面是一些清单文件中适用于所有元素和属性的约定和规则：

### 元素 **Elements:**

在所有的元素中只有[<manifest>](#)和[<application>](#)是必需的，且只能出现一次。很多其他元素可以出现多次甚或一次都没有-尽管如果清单文件想要完成一些有意义的工作，必须设置至少其中的一些。如果一个元素包含点什么，那就是包含其他元素。所有的值必须通过属性来设置，而不是元素中的字符数据。同一级别的元素一般是没有顺序的。比如，[<activity>](#)、[<provider>](#)、和[<service>](#)元素可以以任意顺序混合使用。（[<activity-alias>](#)元素是个例外：它必须跟在该别名所指的[<activity>](#)后面。）

### 属性 **Attributes:**

正规意义上，所有的属性都是可选的，但实际上有些属性是必须为一个元素指定来完成其目标。把这篇文档当作一个指南。对于那些真正可选的属性，即使不存在一个规格，也会有默认的数值或状态。

除了根元素[<manifest>](#)的一些属性，所有其他元素属性的名字都是以 `android:` 作为前缀的-比如，`android:alwaysRetainTaskState`。因为这个前缀是通用的，这篇文档提及属性名称时一般会忽略它。

### 声明类名 **Declaring class names:**

很多对应于 Java 对象的元素，包括应用程序自己（[<application>](#)元素）以及它的基础组件-活动（[<activity>](#)），服务（[<service>](#)），广播接收器（[<receiver>](#)），以及内容提供者（[<provider>](#)）。

如果你定义一个子类，就像你将经常为组件类([Activity](#)、[Service](#)、[BroadcastReceiver](#)、和 [ContentProvider](#))所做的那样，这个子类通过一个名字属性来声明。这个名字必须包含完整的包名称。比如，一个服务 [Service](#) 子类可能会声明如下：

```
<manifest . . . >
```

```

    <application . . . >
        <service android:name="com.example.project.SecretService" . . .
    >
        . . .
    </service>
    . . .
</application>
</manifest>

```

不过，作为类名的简写，如果这个字符串的第一个字符是一个点号“.”，那么这个字符串将被扩展到应用程序包名的后面（正如[<manifest>](#)元素的 [package](#) 属性所指明的那样）。下面这个赋值和上面效果一样：

```

<manifest package="com.example.project" . . . >
    <application . . . >
        <service android:name=".SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>

```

当启动一个组件时，**Android** 创建了一个命名子类的实例。如果没有指定一个子类，它创建基类的一个实例。

### 多数值项 **Multiple values:**

如果某个元素有超过一个数值，这个元素几乎总是需要被重复声明，而不能将多个数值项列举在一个属性中。比如，一个意图过滤器可以列举多个动作：

```

<intent-filter . . . >
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.INSERT" />
    <action android:name="android.intent.action.DELETE" />
    . . .
</intent-filter>

```

## 资源项 Resource values:

一些属性有能显示给用户的数值-比如，活动（activity）的一个标签和图标。这些属性的值应该被本地化，从一个资源或主题中设置。当需要引用某个资源时，采用如下的表述格式：

```
@[package:] type:name
```

这里 *package* 名称可以被忽略，要是资源和应用程序在同一个包里的话；*type* 是资源的类型-如"string"或"drawable"-而且 *name* 是用来标识特定资源的名字。例如

```
<activity android:icon="@drawable/smallPic" . . . >
```

从主题获取的数据以类似的方式表述，不过以'?'而不是'@'开头。

```
?[package:] type:name
```

## 字符串值 String values:

如果属性值是一个字符串，则必须使用双反斜杠（'\\'）来表示 `escape`（'\'）字符；第一个反斜杠起转义字符的作用）。比如，'\\n'表示换行或'\\uxxxx'表示一个 Unicode 字符。

## 文件特性 File Features

下面的章节描述了一些 Android 特性如何被映射到清单（manifest）文件中。

### 意图过滤器 Intent Filters

应用程序的核心组件（活动，服务和广播接收器）通过意图被激活。意图是描述期望动作的信息包（一个 [Intent](#) 对象）-包括要操作的数据，执行该动作的组件类别，以及其他有关指令。Android 寻找一个合适的组件来响应这个意图，如果需要会启动这个组件一个新的实例，并传递给这个意图对象。

组件通过意图过滤器（*intent filters*）通告它们所具备的能力-能响应的意图类型。由于 Android 系统在启动一个组件前必须知道该组件能够处理哪些意图，那么意图过滤器需要在 manifest 中以[<intent-filter>](#)元素指定。一个组件可以拥有多个过滤器，每一个描述不同的能力。

一个显式命名目标组件的意图将会激活那个组件；过滤器不起作用。但是一个没有指定目标的意图只在它能够通过组件过滤器任一过滤器时才能激活该组件。

请查看关于意图和意图过滤器的文档以获取更多信息：[Intents and Intent Filters](#).

### 图标和标签 Icons and Labels

许多元素有图标（`icon`）和标签（`label`）属性。其中一些还有一个描述（`description`）属性，可以用更长的解释性文字呈现给用户。比如，[<permission>](#)元素有所有这三个属性，因此当用户被询问是

否授予一个应用程序请求的权限许可时，一个代表权限的图标，权限的名称和必定伴有的权限描述会全部被显示给用户。

所有的情况中，设置在一个包含元素里的图标和标签会成为该容器所有子元素的缺省设置。这样，在 [<application>](#) 元素中设置的图标和标签就是该应用程序每个组件的缺省图标和标签。类似的，为一个组件设置的图标和标签-比如，一个 [<activity>](#) 元素-是这个组件 [<intent-filter>](#) 元素的缺省值。如果一个 [<application>](#) 元素设置了一个图标，但活动及其意图过滤器没有，那么程序标签被当作活动和意图过滤器的标签。

当呈现给用户的组件实现一个意图过滤器公告的函数时，为这个过滤器设置的图标和标签将被用来代表这个组件。比如，一个设置了 `"android.intent.action.MAIN"` 和 `"android.intent.category.LAUNCHER"` 的过滤器公告了一个活动来初始化应用程序-也就是，会被显示在应用程序启动器中。因此设置在过滤器中的图标和标签也就是显示在启动器里的那些图标和标签。

## 许可 *Permissions*

一个许可 (*permission*) 是代码对设备上数据的访问限制。这个限制被引入来保护可能会被误用而曲解或破坏用户体验的关键数据和代码。

每个许可被一个唯一的标签所标识。这个标签常常指出了受限的动作。例如，下面是一些 Android 定义的许可：

```
android.permission.CALL_EMERGENCY_NUMBERS
android.permission.READ_OWNER_DATA
android.permission.SET_WALLPAPER
android.permission.DEVICE_POWER
```

一个功能 (feature) 最多只能被一个权限许可保护。

如果一个应用程序需要访问一个需要特定权限的功能，它必须在 manifest 文件中使用 [<uses-permission>](#) 元素来声明这一点。这样，当应用程序安装到设备上时，安装器可以通过检查签署应用程序认证的机构来决定是否授予请求的权限，在某些情况下，会询问用户。如果权限已被授予，那应用程序就能够访问受保护的功能特性。如果没有，访问将失败，但不会给用户任何通知。

应用程序还可以通过权限许可来保护它自己的组件（活动，服务，广播接收器，和内容提供者）。它可以利用 Android 已经定义（列在 [android.Manifest.permission](#) 里）或其他应用程序已声明的权限许可。或者定义自己的许可。一个新的许可通过 [<permission>](#) 元素声明。比如，一个活动可以用下面的方式保护：

```
<manifest . . . >

    <permission android:name="com.example.project.DEBIT_ACCT" . . .
/>

    . . .

    <application . . .>

        <activity android:name="com.example.project.FreneticActivity"
. . . >

            android:permission="com.example.project.DEBIT_ACCT"
```

```

        . . . . >

        . . . .

    </activity>

</application>

. . . .

<uses-permission android:name="com.example.project.DEBIT_ACCT" />

. . . .

</manifest>

```

注意，在这个例子里，这个 `DEBIT_ACCT` 许可并非仅仅在 `<permission>` 元素中声明，它同样声明在 `<uses-permission>` 元素里。为了应用程序的其他组件可以启动这个受保护的活动，必须请求它的使用（`use`），即使这个保护是应用程序自己引入的。

如果，就在这个例子里，这个 `permission` 属性被设置为在其他地方声明的权限许可（例如 `android.permission.CALL_EMERGENCY_NUMBERS`，它将不需要再次声明它，但是，它仍然需要通过 `<uses-permission>` 来请求它的使用。

这个 `<permission-tree>` 元素为一组想在代码中定义的权限许可声明了一个命名空间。而 `<permission-group>` 元素为一系列许可定义了一个标签（用 `<permission>` 元素定义在 `manifest` 中的以及其他地方声明的）。它仅仅影响这些权限许可在显示给用户时如何分组。`<permission-group>` 元素并不指明哪个权限属于这个分组；它只是给这个组命名。一个权限许可通过给 `<permission>` 元素的 `permissionGroup` 属性赋予这个组名来放置到这个权限组中。

## 库 *Libraries*

每个应用程序都链接到缺省的 **Android** 库，这个库包含了基础应用程序开发包（实现了基础类如活动，服务，意图，视图，按钮，应用程序，内容提供者，等等）

然而，一些包处于它们自己的库中。如果你的应用程序使用了其他开发包中的代码，它必须显式的请求链接到它们。这个 `manifest` 必须包含一个单独的 `<uses-library>` 元素来命名每一个库。