



Android Training Course in Chinese

Android Training交流组：363415744，欢迎所有学习Android开发的同学加入交流，更欢迎有意向参与到这个课程汉化项目中的同学。



项目主页

<https://github.com/kesenhoo/android-training-course-in-chinese>

麻烦点击Star进行支持！

项目初衷

Google Android团队在2012年起开设了Training板块，从开始的十几篇文章，不断的增加，截止到现在2014年中，已经有近百个课程。无疑，这是学习Android应用开发的绝佳一手资料。

这么好的资料一直没有一份完整的中文版，本人从2012年发现Training课程开始，断断续续的在学习Android官方的Training课程，并的输出了不少翻译[学习笔记](#)，因个人实力与精力有限，很期待能够通过Github发起这个协作项目，借助大家的力量，一起尽快完成所有课程的中文版。这不仅仅只是在翻译，更是一个学习积累并输出帮助别人的好事情。期待大家的加入！

编写:

校对:

开始

编写:[yuanfentiank789](#)

校对:

建立你的第一个App

欢迎开始Android应用开发！本章节教你如何建立你的第一个Android应用程序。您将学到如何创建一个Android项目和运行可调试版本的应用程序。你还将学习一些Android应用程序设计的基础知识，包括如何创建一个简单的用户界面，处理用户输入。

开始本章节学习之前，确保你已经安装了开发环境。你需要：

1 下载Android SDK.

2 为Eclipse安装ADT插件 (假设你使用Eclipse作为开发工具).

3 使用SDK Manager下载最新的SDK tools和platforms。

注意：开始本节学习之前，确保你安装了最新版本的ADT插件和Android SDK。本章节描述的学习过程有可能不适用早期版本。

如果你尚未完成这些任务，开始[Android SDK](#)的下载和安装步骤。安装完成后，你就可以准备开始本章节的学习了。

本章节通过教程的方式逐步建立一个小型的Android应用，来交给你一下Android开发的基本概念，因此对你来说每一步都很重要。

[开始第一节课](#)

编写:[yuanfentiank789](#)

校对:


创建一个Android项目

一个Android项目由许多源代码文件构成，使用Android SDK Tools可以很容易地创建一个新的Android项目，同时创建好默认的项目相关的目录和文件。

本小节介绍如何使用Eclipse（已安装ADT插件）或SDK Tools命令行来创建一个新的项目。

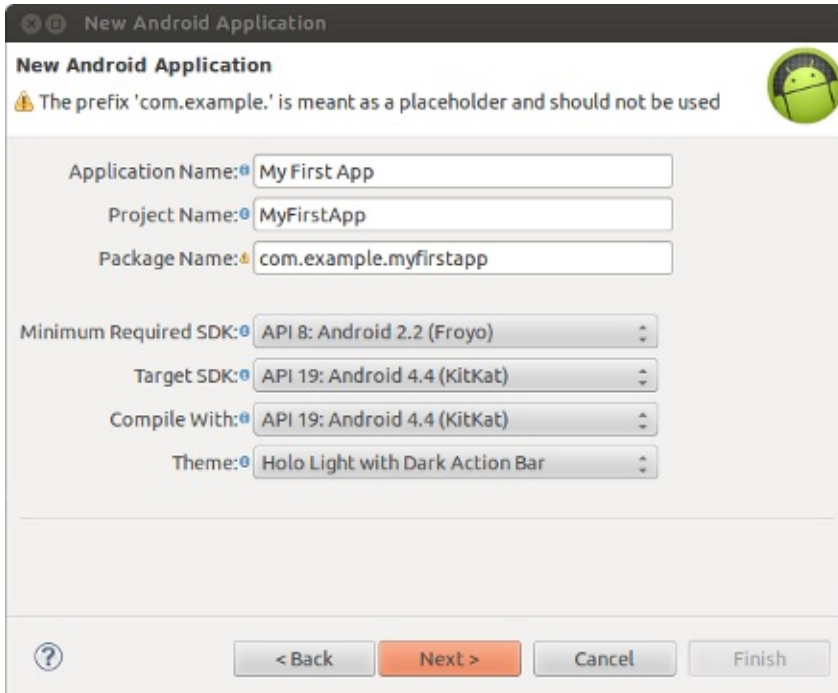
注意：如果你使用Eclipse开发，应该确保已经安装了Android SDK，并且为Eclipse安装了ADT（version 22.6.2或更高版本）插件。否则，请先阅读 [Installing the Android SDK](#)按照向导完成安装。

使用Eclipse创建项目

1 点击Eclipse工具栏的New  按钮。

2 在弹出的窗口打开Android文件夹，选择**Android Application Project**，点击**Next**。

3 填写如下图弹出的表格：



Application Name 此处填写想呈现给用户的应用名称，此处我们使用“My First App”。

Project Name 是项目的文件夹名称和在Eclipse中显示的名称。

Package Name 是应用的包命名空间（同Java的包的概念），该包名在同一Android系统上已安装所有应用包名中具有唯一性，因此，通常使用你所在公司组织或发布实体的反向域名作为包名的开始是一个很好的选择。此处可以使用“com.example.myfirstapp”，但是你不能在 Google Play上发布使用“com.example”作为包名的应用。

Minimum Required SDK 用API level表示你的应用支持的最低Android版本，为了支持尽可能多的设备，你应该设置为能支持你应用核心功能的最低API版本。如果某些非核心功能尽在较高版本的API支持，你可以只在支持这些功能的版本上开启它们（参考 [Supporting Different Platform Versions](#)），此处采用默认值即可。

Target SDK 表示你测试过你的应用支持的最高Android版本（同样用API level表示）。当Android发布最新版本后，你应该在最新版本的Android测试你的应用同时更新target sdk到Android最新版本，以便充分利用Android新版本的特性。

Compile With 是你的应用将要编译的目标Android版本，此处默认为你的SDK已安装的最新Android版本（目前应该是4.1或更高版本，如果你没有安装一个可用Android版本，就要先用[SDK Manager](#)来完成安装），你仍然可以使用较老的版本编译项目，但把该值设为最新版本，你可以使用Android的最新特性同时可以优化应用来提高用户体验，运行在最新的设备上。

Theme 为你的应用指定界面风格，此处采用默认值即可。

点击**Next**

4 接下来的窗口配置项目，保持默认值即可，点击**Next**。

5 这一步帮助你给你的应用创建一个启动图标，你也可以自定义应用启动图标，通过用工具为各种屏幕密度的屏幕各创建一个对应图标。但在发布应用之前，应确保你设计的图标符合[Iconography](#)中规定的设计规范。

点击**Next**。

6 这一步为默认的入口Activity选择一个模板，此处选择**BlankActivity** 然后点击**Next**。

7 这一步保持Activity的默认配置即可。

到此为止，你的Android项目已经是一个基本的“Hello World”程序，包含了一些默认的文件。要运行它，继续[下个小节](#)的学习。

使用命令行创建项目

如果你没有使用Eclipse+ADT开发Android项目，也可以在命令行使用SDK提供的tools来创建一个Android项目。

1 打开命令行切换到SDK根目录/tools目录下；

2 执行

```
android list targets
```

会在屏幕上打印出所有你使用Android SDK下载好的可用platforms，找到你想要创建项目的目标platform，记录该platform对应的Id，推荐你使用最新的platform，可以使你的应用支持较老版本的platform，同时允许你为最新的Android设备优化你的应用。如果你没有看到任何可用的platform，你需要使用SDK Manager完成下载安装，参见 [Adding Platforms and Packages](#)。

3 执行

```
android create project --target <target-id> --name MyFirstApp \  
--path <path-to-workspace>/MyFirstApp --activity MainActivity \  
--package com.example.myfirstapp
```

替换为上一步记录好的Id，替换为你想要保存项目的路径，到此为止，你的Android项目已经是一个基本的“Hello World”程序，包含了一些默认的文件。要运行它，继续[下个小节](#)的学习。

小提示：把 platform-tools/和 tools/添加到环境变量PATH，开发更方便。

编写:[yuanfentiank789](#)

校对:

执行你的程序

通过 [上一节课](#) 创建了一个Android项目，项目默认包含一系列源文件，它让您可以立即运行的应用程序。

如何运行Android应用取决于你是否有一个Android设备和你是否正在使用Eclipse开发。本节课将会教使用Eclipse和命令行两种方式在真实地android设备或者android模拟器上安装并且运行你的应用。

在运行应用之前，你得认识项目里的几个文件和目录：

AndroidManifest.xml

[manifest file](#) 描述了应用程序的基本特性并且定义了每一个组件。当你学了更多课程，你将会理解这里的各种声明。最重要的一点：你的manifest应该包括 `android:minSdkVersion` 和 `android:targetSdkVersion` 两个属性来声明你应用程序对于不同的android版本的兼容性。在你的第一个应用里，它看起来应该是这样：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ... >
<uses-sdk android:minSdkVersion="8" android:targetSdkVersion="19" />
...
</manifest>
```

你应该总是把 `android:targetSdkVersion` 设置的尽可能的高并且在对应版本的Android系统上测试你的应用。详见 [Supporting Different Platform Versions](#).

src/

这是存放应用的主要源文件的文件夹，默认情况下，里面会包括一个 [Activity](#) 的类，这个类会在点击应用程序图标启动的时候运行。

res/

包含一些存放资源文件的目录，例如：

drawable-hdpi/

存放适用于HDPI屏幕的图片素材。同理其他类似文件夹存放适用于其他屏幕的图片素材。

layout/

存放定义用户界面的的文件。

values/

存放其他各种XML文件，也是所有资源的集合，例如字符串和颜色的定义。

当完成该项目的编译和运行工作后，默认的 [Activity](#) 类启动并加载一个布局文件，界面显示 "Hello World." 这本身没有什么值得兴奋的，重要的是你学会了如何运行一个Android应用在你开始进行开发之前。

在真实设备上运行

如果你有一个真实地Android设备，以下的步骤可以使你在你的设备上安装和运行你的应用程序：

1 把你的设备用USB线连接到计算机上。如果你是在windows系统上进行开发的，你可能还需要安装你设备对应的USB驱动，详见[OEM USB Drivers](#) 文档。


2 开启设备上的USB调试选项。

2.1 在大部分运行Android3.2或更老版本系统的设备上，这个选项位于“设置——应用程序——开发选项”里。

2.2 在Android 4.0或更新版本中，这个选项在“设置——开发人员选项”里。

注意: 在Android4.2或更新版本中，开发人员选项在默认情况下是隐藏的，想让它可见，可以去“设置——关于手机（或者关于设备）”点击“版本号”七次。再返回就能找到开发人员选项了。

用Eclipse在设备里运行程序：

1 打开项目文件，点击工具栏里的 **Run**  按钮。

2 在 Run as 弹出窗口中，选择 Android Application 然后点击 OK。

Eclipse 会把应用程序安装到你的设备中并启动应用程序。

也可以利用命令行安装运行你的应用程序。

1 命令行切换当前目录到Android项目的根目录，执行：

```
ant debug
```

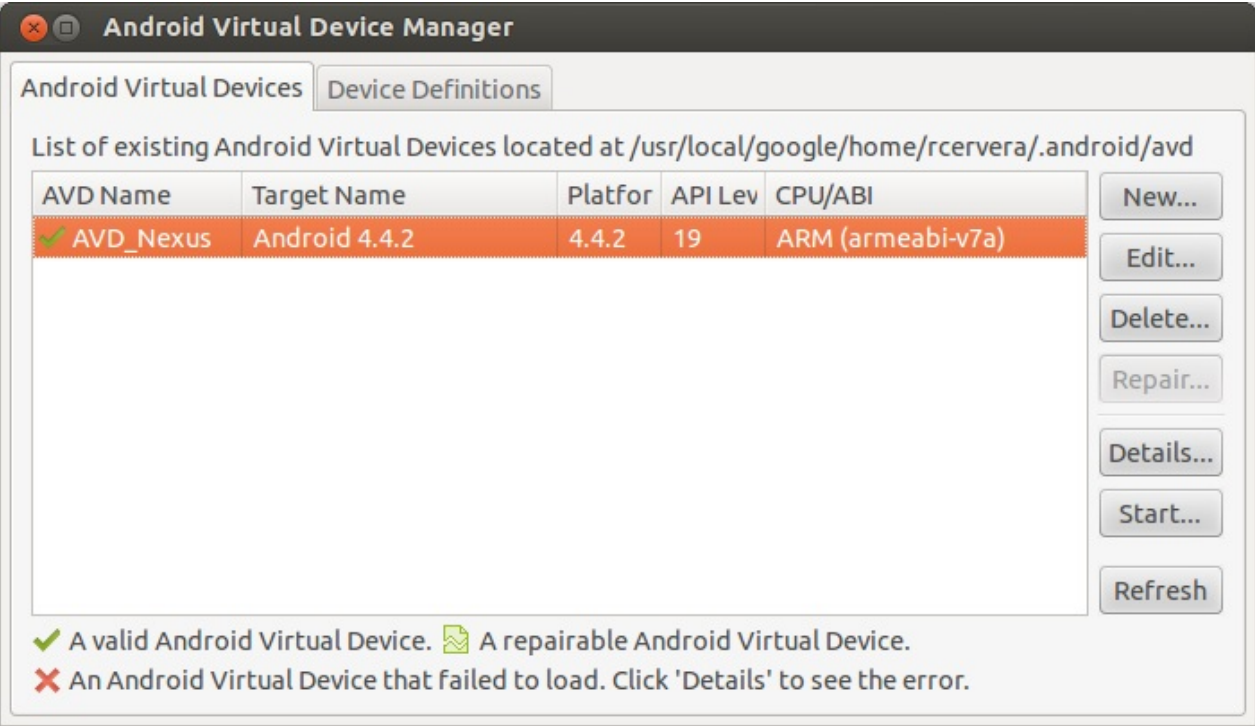
2 确保 Android SDK里的 platform-tools/ 路径已经添加到环境变量的Path中，执行：adb install bin/MyFirstApp-debug.apk

3 在你的Android设备中找到 MyFirstActivity，点击打开。

以上就是创建并在设备上运行一个应用的全部过程！想要开始开发，点击[next lesson](#)。

在模拟器上运行

无论你是用Eclipse还是命令行，在模拟其中运行程序首先要创建一个模拟器，即 Android Virtual Device (AVD)，配置AVD 可以让你模拟在不同版本和尺寸的Android设备运行应用程序。



创建一个 AVD: 1 启动 Android Virtual Device Manager（AVD Manager）的两种方式：a 用Eclipse, 点击工具栏里面 Android Virtual Device Manager Android 图标。

b 在命令行窗口中，把当前目录切换到/tools/ 后执行：

```
android avd
```

- 2 在 Android Virtual Device Manager 面板中， 点击 New.
 - 3 填写AVD的详细信息，包括名字，平台版本，SD卡大小以及屏幕大小（默认是HVGA）。
 - 4 点击 Create AVD.
 - 5 在Android Virtual Device Manager 选中创建的新AVD， 点击 Start.
 - 6 在模拟器启动完毕后， 解锁模拟器的屏幕。
- 接下来就可以像前边讲过的一样用Eclipse或命令行来往模拟器发布运行你的应用程序了。

编写:

校对:

建立一个简单的用户界面

编写:

校对:

启动另外的**Activity**

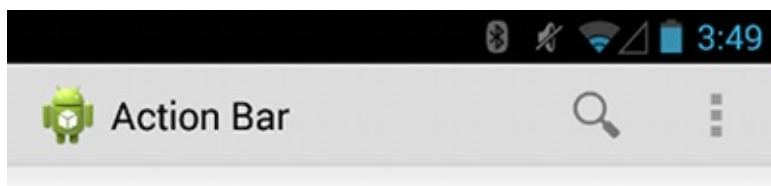
编写: [Vincent 4J](#)

校对:

添加 Action Bar

Action bar 是最重要的设计元素之一，你可以在 activity 中实现它。它提供了多种 UI 特性，可以让你的 app 相比于其他 Android app 比较一致，为用户所熟悉。核心的功能包括：

- 一个专门空间，给你的 app 一个个性，并且指示出用户的位置
- 以一种可预见的方式接入重要的操作（比如检索）
- 支持导航和视图切换（通过页签和下拉列表）



本章培训课程对 action bar 的基本知识提供了一个快速指南。关于 action bar 的更多特性，请查看 [Action Bar](#) 指南。

课程

[建立 Action Bar](#)

学习如何为你的 activity 添加一个基本的 action bar，不仅支持 Android 3.0 和更高，同时也支持不低于 Android 2.1（通过使用 Android Support 库）。

[添加 Action 按钮](#)

学习如何在 action bar 中添加和响应用户操作。

[Action Bar 风格化](#)

学习如何自定义你的 action bar 的外观

[Action Bar 覆盖叠加](#)

学习如何在布局上面叠加 action bar，允许 action bar 隐藏时无缝过渡。

编写: [Vincent 4J](#)

校对:

建立 Action Bar

Action bar 最基本的形式，就是为 activity 显示标题，并且在标题左边显示一个 app icon。即使在这样简单的形式下，对于所有的 activity 来说，action bar 对告知用户他们当前所处的位置十分有用，并为你的 app 保持了一致性。

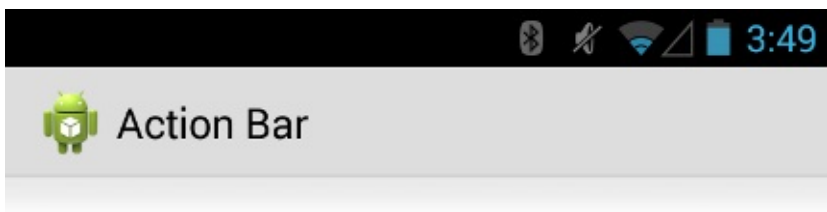


图 1. 一个有 app icon 和 activity 标题的 action bar

设置一个基本的 action bar，需要你的 app 使用一个 action bar 可用的 activity 主题。如何声明这样的主题取决于你的 app 支持的 Android 最低版本。所以本课程根据你的 app 支持的 Android 最低版本分为两部分。

仅支持 **Android 3.0** 及以上版本

从 Android 3.0(API level 11) 开始，所有使用 Theme.Holo 主题（或者它的子类）的所有 activity 都包含 action bar，当 `targetSdkVersion` 或 `minSdkVersion` 属性被设置成 “11” 或更大时，它是默认主题。

所以，为你的 activity 添加 action bar，只需简单地设置属性为 11 或者更大。例如：

```
<manifest ... >
    <uses-sdk android:minSdkVersion="11" ... />
    ...
</manifest>
```

注释：如果创建一个自定义主题，需确保它使用一个 Theme.Holo 主题作为父辈。详情请查看 [Action bar 风格化](#)

到此，你的 app 使用了 Theme.Holo 主题，并且所有的 activity 都显示 action bar。

支持 Android 2.1 及以上版本

当 app 运行在 Android 3.0 以下版本（不低于 Android 2.1）时，如果要添加 action bar，需要加载 Android Support 库。

通过阅读 [安装 Support 库](#) 文档和安装 v7 appcompat 库 来开始（下载完库包之后，按照 [添加资源库](#) 的说明来添加）。

一旦 Support 库集成到你的 app 工程之中：

1、更新 activity，以便于它继承于 ActionBarActivity。例如：

```
public class MainActivity extends ActionBarActivity { ... }
```

2、在 manifest 文件中，更新 <application> 元素或者单一的 <activity> 元素来使用一个 Theme.AppCompat 主题。例如：

```
<activity android:theme="@style/Theme.AppCompat.Light" ... >
```

注释：如果创建一个自定义主题，需确保它使用一个 Theme.AppCompat 主题作为父辈。详情请查看 [Action bar 风格化](#)

当 app 运行在 Android 2.1(API level 7) 或者以上时，activity 将包含 action bar。

切记，在 manifest 中正确地设置 app 支持的 API level：

```
<manifest ... >
    <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="18" />
    ...
</manifest>
```

编写: [Vincent 4J](#)

校对:

添加 **Action** 按钮

Action bar 允许你为当前上下文中最重要的操作添加按钮。那些直接出现在 action bar 中的 icon 和/或文本被称作操作按钮。不匹配的或不足够重要的操作被隐藏在 action overflow 中。

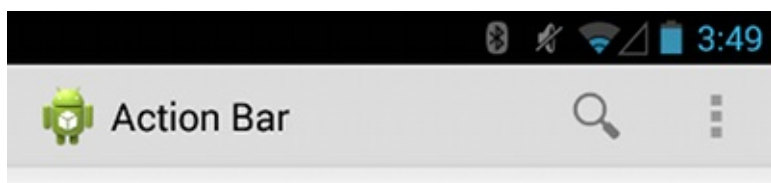


图 1. 一个有检索操作按钮和 action overflow 的 action bar，在 action overflow 里能展现额外的操作

在 XML 中指定操作

所有的操作按钮和 action overflow 中其他可用的条目都被定义在 [菜单资源](#) 的 XML 文件中。通过在项目的 res/menu 目录中新增一个 XML 文件来为 action bar 添加操作。

为你想添加到 action bar 中的每个条目添加一个 <item> 元素。例如：

res/menu/main_activity_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- Search, should appear as action button -->
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search"
        android:title="@string/action_search"
        android:showAsAction="ifRoom" />
    <!-- Settings, should always be in the overflow -->
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:showAsAction="never" />
</menu>
```

上述声明是这样的，当 action bar 有可用空间时，检索操作将作为一个操作按钮来显示，但设置操作将一直只在 action overflow 中显示。（默认情况下，所有的操作都显示在 action overflow 中，但为每一个操作指明设计意图是很好的做法。）

icon 属性要求每张图片提供一个 resource ID。在 @drawable/ 之后的名字必须是存储在项目目录 res/drawable/ 下图片的名字。例如：ic_action_search.png 对应 "@drawable/ic_action_search"。同样地，title 属性使用通过 XML 文件定义在项目目录 res/values/ 中的一个 string resource，详情请参见 [创建一个简单的 UI](#)。

注释：当在创建 icon 和其他 bitmap 图片时，你得为优化不同屏幕密度下的显示效果提供多个版本，这一点很重要。在 [支持不同屏幕](#) 课程中将会更详细地讨论。

如果你为了兼容 Android 2.1 以下版本使用了 Support 库，在 android 命名空间下 showAsAction 属性是不可用的。Support 库会提供替代它的属性，你必须声明自己的 XML 命名空间，并且使用该命名空间作为属性前缀。（一个自定义 XML 命名空间需要以你的 app 名称为基础，但是可以取任何你想要的名称，它的作用域仅仅在你声明的文件之内。）例如：

res/menu/main_activity_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <!-- Search, should appear as action button -->
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search"
        android:title="@string/action_search"
        yourapp:showAsAction="ifRoom" />
    . . .
</menu>
```

为 Action Bar 添加操作

为 action bar 布局菜单条目，是通过在 activity 中实现 [onCreateOptionsMenu\(\)](#) 回调方法来 inflate 菜单资源从而获取 [Menu](#) 对象。例如：

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu items for use in the action bar
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

为操作按钮添加响应事件

当用户按下某一个操作按钮或者 action overflow 中的其他条目，系统将调用 activity 中 [onOptionsItemSelected\(\)](#) 回调方法。在该方法的实现里面调用 [getItemId\(\)](#) 获取 [MenuItem](#) 来判断哪个条目被按下——返回的 ID 会匹配你声明对应的 `<item>` 元素中 `<android:id>` 属性的值。

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle presses on the action bar items
    switch (item.getItemId()) {
        case R.id.action_search:
            openSearch();
            return true;
        case R.id.action_settings:
            openSettings();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```


为下级 Activity 添加向上按钮

在不是主要入口的其他所有屏中（activity 不位于主屏时），需要在 action bar 中为用户提供一个导航到逻辑父屏的向上按钮。



图 2. Gmail 中向上按钮

当运行在 Android 4.1(API level 16) 或更高版本，或者使用 Support 库中的 [ActionBarActivity](#) 时，实现向上导航需要你在 manifest 文件中声明父 activity，同时在 action bar 中设置向上按钮可用。

如何在 manifest 中声明一个 activity 的父辈，例如：

```
<application ... >
    ...
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- A child of the main activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="com.example.myfirstapp.MainActivity" >
        <!-- Parent activity meta-data to support 4.0 and lower -->
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.myfirstapp.MainActivity" />
        </activity>
</application>
```

然后，通过调用 [setDisplayHomeAsUpEnabled\(\)](#) 来把 app icon 设置成可用的向上按钮：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_displaymessage);

    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    // If your minSdkVersion is 11 or higher, instead use:
    // getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

由于系统已经知道 MainActivity 是 DisplayMessageActivity 的父 activity，当用户按下向上按钮时，系统会导航到恰当的父 activity —— 你不需要去处理向上按钮的事件。

更多关于向上导航的信息，请见 [提供向上导航](#)。

编写: [Vincent 4J](#)

校对:

Action Bar 风格化

Action bar 为用户提供一种熟悉可预测的方式来展示操作和导航，但是这并不意味着你的 app 要看起来和其他 app 一样。如果你想将 action bar 的风格设计的合乎你产品的定位，你只需简单地使用 Android 的 [式样和主题](#) 资源。

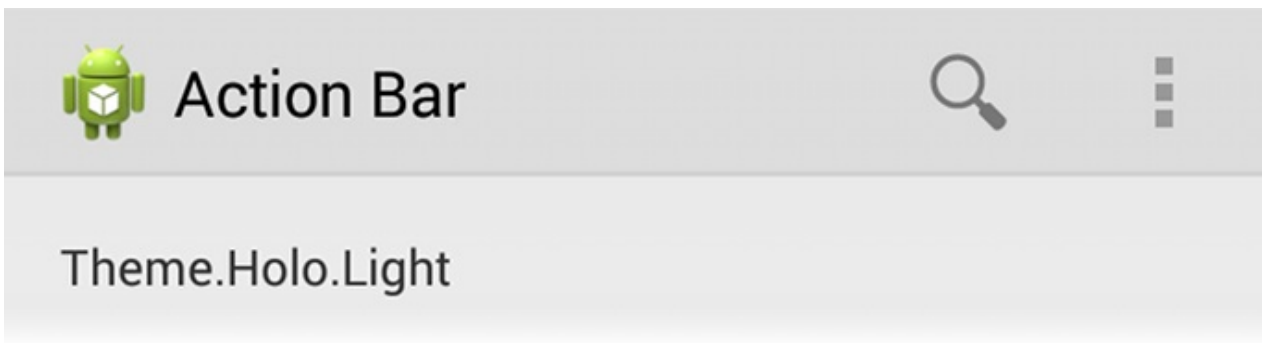
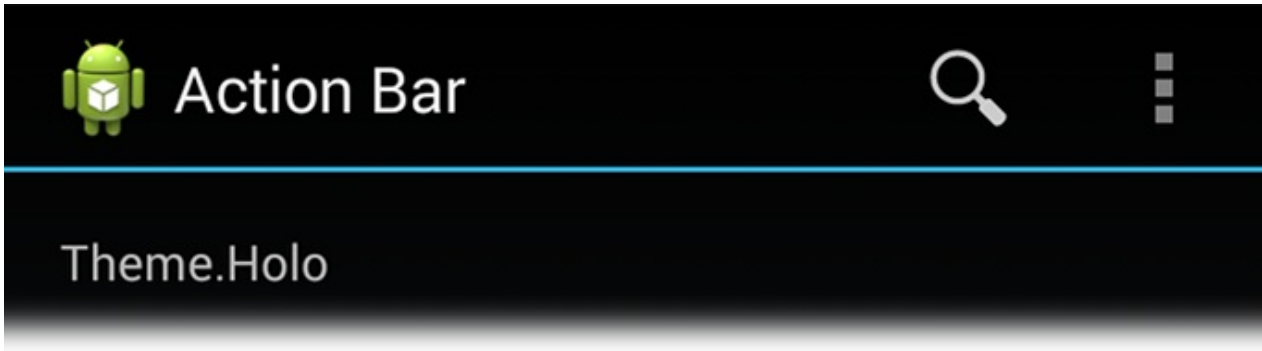
Android 包括一少部分内置的 activity 主题，这些主题中包含“暗”或“淡”的 action bar 式样。你也可以扩展这些主题，以便于更好的为你的 action bar 自定义外观。

注释：如果你为 action bar 使用了 Support 库的 API，那你必须使用（或重写）[Theme.AppCompat](#) 家族式样（甚至 [Theme.Holo](#) 家族，在 API level 11 或更高版本中可用）。如此一来，你声明的每一个式样属性都必须被声明两次：一次使用平台的式样属性（`android:` 属性），另一次使用 Support 库中的式样属性（`appcompat.R.attr` 属性——这些属性的上下文其实就是你的 app）。更多细节请查看下面的示例。

使用一个 **Android** 主题

Android 包含两个基本的 activity 主题，这两个主题决定了 action bar 的颜色：

- [Theme.Holo](#)，一个“暗”的主题
- [Theme.Holo.Light](#)，一个“淡”的主题

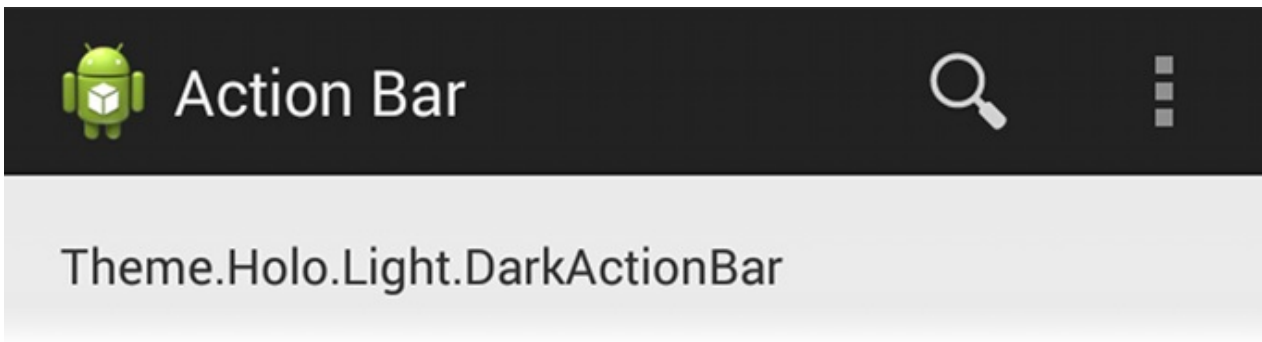


这些主题即可以被应用到 app 全局，又可以为单一的 activity 通过在 manifest 文件中设置 [application](#) 元素 或 [activity](#) 元素的 `android:theme` 属性。

例如：

```
<application android:theme="@android:style/Theme.Holo.Light" ... />
```

你可以通过声明 activity 的主题为 [Theme.Holo.Light.DarkActionBar](#) 来达到如下效果：action bar 为暗色，其他部分为淡色。



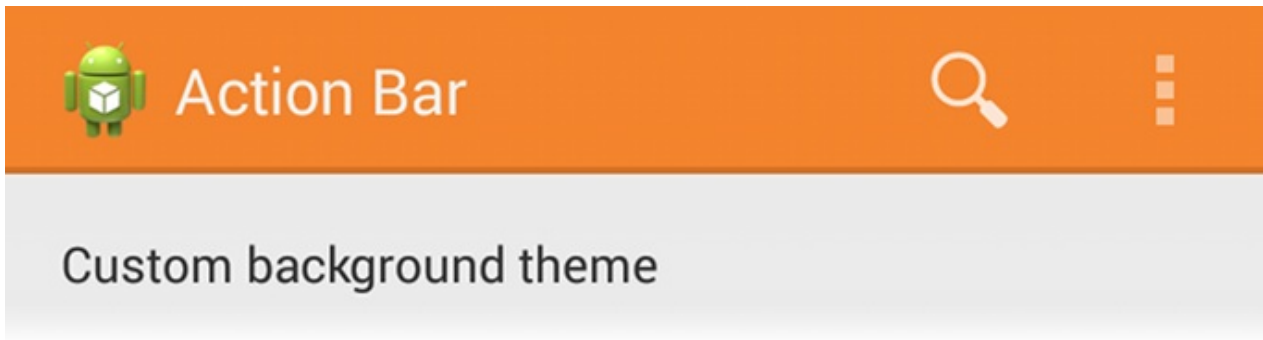
当使用 Support 库时，必须使用 [Theme.AppCompat](#) 主题替代：

- [Theme.AppCompat](#)，一个“暗”的主题
- [Theme.AppCompat.Light](#)，一个“淡”的主题
- [Theme.AppCompat.Light.DarkActionBar](#)，一个带有“暗”action bar 的“淡”主题

一定要确保你使用的 action bar icon 的颜色与 action bar 本身的颜色有差异。为了能帮助你，[Action Bar Icon Pack](#) 为 Holo “暗”和“淡”的 action bar 提供了标准的 action icon。

自定义背景

为 activity 创建一个自定义主题，通过重写 [actionBarStyle](#) 属性来改变 action bar 的背景。[actionBarStyle](#) 属性指向另一个式样；在该式样里，通过指定一个 drawable 资源来重写 [background](#) 属性。



如果你的 app 使用了 [navigation tabs](#) 或 [split action bar](#)，你也可以通过分别设置 [backgroundStacked](#) 和 [backgroundSplit](#) 属性来为这些条指定背景。

注意：声明一个合适的父主题，进而你的自定义主题和式样可以继承父主题的风格，这点很重要。如果没有父式样，你的 action bar 将会失去很多式样属性，除非你自己显式的对他们进行声明。

仅支持 **Android 3.0** 和更高

当仅支持 Android 3.0 和更高版本时，你可以像这样定义 action bar 的背景：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@android:style/Theme.Holo.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
        parent="@android:style/Widget.Holo.Light.ActionBar.Solid.Inverse">
        <item name="android:background">@drawable/actionbar_background</item>
    </style>
</resources>
```

然后，将你的主题应该到你的 app 全局或单个的 activity 之中：

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

支持 **Android 2.1** 和更高

当使用 Support 库时，上面同样的主题必须被替代成如下：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@style/Theme.AppCompat.Light.DarkActionBar">
        <item name="android:actionBarStyle">@style/MyActionBar</item>

        <!-- Support library compatibility -->
        <item name="actionBarStyle">@style/MyActionBar</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
        parent="@style/Widget.AppCompat.Light.ActionBar.Solid.Inverse">
        <item name="android:background">@drawable/actionbar_background</item>
```

```
<!-- Support library compatibility -->
<item name="background">@drawable/actionbar_background</item>
</style>
</resources>
```

然后，将你的主题应该到你的 app 全局或单个的 activity 之中：

```
<application android:theme="@style/CustomActionBarTheme" ... />
```

自定义文本颜色

修改 action bar 中的文本颜色，你需要分别重写每个元素的属性：

- Action bar 的标题：创建一种自定义式样，并指定 `textColor` 属性；同时，在你的自定义 [actionBarStyle](#) 中为 [titleTextStyle](#) 属性指定为刚才的自定义式样。

注释：被应用到 [titleTextStyle](#) 的自定义式样应该使用 [TextAppearance.Holo.Widget.ActionBar.Title](#) 作为父式样。

- Action bar 的页签：在你的 activity 主题中重写 [actionBarTabTextStyle](#)
- Action 按钮：在你的 activity 主题中重写 [actionMenuTextColor](#)

仅支持 **Android 3.0** 和更高

当仅支持 Android 3.0 和更高时，你的式样 XML 文件应该是这样的：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@style/Theme.Holo">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="android:actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
        parent="@style/Widget.Holo.ActionBar">
        <item name="android:titleTextStyle">@style/MyActionBarTitleText</item>
    </style>

    <!-- ActionBar title text -->
    <style name="MyActionBarTitleText"
        parent="@style/TextAppearance.Holo.Widget.ActionBar.Title">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar tabs text styles -->
    <style name="MyActionBarTabText"
        parent="@style/Widget.Holo.ActionBar.TabText">
        <item name="android:textColor">@color/actionbar_text</item>
    </style>
</resources>
```

支持 **Android 2.1** 和更高

当使用 Support 库时，你的式样 XML 文件应该是这样的：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@style/Theme.AppCompat">
        <item name="android:actionBarStyle">@style/MyActionBar</item>
        <item name="android:actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="android:actionMenuTextColor">@color/actionbar_text</item>

        <!-- Support library compatibility -->
        <item name="actionBarStyle">@style/MyActionBar</item>
        <item name="actionBarTabTextStyle">@style/MyActionBarTabText</item>
        <item name="actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- ActionBar styles -->
    <style name="MyActionBar"
        parent="@style/Widget.AppCompat.ActionBar">
        <item name="android:titleTextStyle">@style/MyActionBarTitleText</item>
```

```
<!-- Support library compatibility -->
<item name="titleTextStyle">@style/MyActionBarTitleText</item>
</style>

<!-- ActionBar title text -->
<style name="MyActionBarTitleText"
    parent="@style/TextAppearance.AppCompat.Widget.ActionBar.Title">
    <item name="android:textColor">@color/actionbar_text</item>
    <!-- The textColor property is backward compatible with the Support Library -->
</style>

<!-- ActionBar tabs text -->
<style name="MyActionBarTabText"
    parent="@style/Widget.AppCompat.ActionBar.TabText">
    <item name="android:textColor">@color/actionbar_text</item>
    <!-- The textColor property is backward compatible with the Support Library -->
</style>
</resources>
```


自定义 Tab Indicator

为 activity 创建一个自定义主题，通过重写 [actionBarTabStyle](#) 属性来改变 [navigation tabs](#) 使用的指示器。[actionBarTabStyle](#) 属性指向另一个式样资源；在该式样资源里，通过指定一个状态列表 drawable 来重写 [background](#) 属性。

注释：一个状态列表 drawable 是重要的，以便通过不同的背景来指出当前选择的 tab 与其他 tab 的区别。更多关于如何创建一个 drawable 资源来处理多个按钮状态，请阅读 [State List](#) 文档。

例如，这是一个状态列表 drawable，为一个 action bar tab 的多种不同状态分别指定背景图片：

res/drawable/actionbar_tab_indicator.xml

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

  <!-- STATES WHEN BUTTON IS NOT PRESSED -->

  <!-- Non focused states -->
  <item android:state_focused="false" android:state_selected="false"
        android:state_pressed="false"
        android:drawable="@drawable/tab_unselected" />
  <item android:state_focused="false" android:state_selected="true"
        android:state_pressed="false"
        android:drawable="@drawable/tab_selected" />

  <!-- Focused states (such as when focused with a d-pad or mouse hover) -->
  <item android:state_focused="true" android:state_selected="false"
        android:state_pressed="false"
        android:drawable="@drawable/tab_unselected_focused" />
  <item android:state_focused="true" android:state_selected="true"
        android:state_pressed="false"
        android:drawable="@drawable/tab_selected_focused" />

  <!-- STATES WHEN BUTTON IS PRESSED -->

  <!-- Non focused states -->
  <item android:state_focused="false" android:state_selected="false"
        android:state_pressed="true"
        android:drawable="@drawable/tab_unselected_pressed" />
  <item android:state_focused="false" android:state_selected="true"
        android:state_pressed="true"
        android:drawable="@drawable/tab_selected_pressed" />

  <!-- Focused states (such as when focused with a d-pad or mouse hover) -->
  <item android:state_focused="true" android:state_selected="false"
        android:state_pressed="true"
        android:drawable="@drawable/tab_unselected_pressed" />
  <item android:state_focused="true" android:state_selected="true"
        android:state_pressed="true"
        android:drawable="@drawable/tab_selected_pressed" />
</selector>
```

仅支持 **Android 3.0** 和更高

当仅支持 Android 3.0 和更高时，你的式样 XML 文件应该是这样的：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <!-- the theme applied to the application or activity -->
  <style name="CustomActionBarTheme"
        parent="@style/Theme.Holo">
    <item name="android:actionBarTabStyle">@style/MyActionBarTabs</item>
  </style>

  <!-- ActionBar tabs styles -->
  <style name="MyActionBarTabs"
        parent="@style/Widget.Holo.ActionBar.TabView">
    <!-- tab indicator -->
    <item name="android:background">@drawable/actionbar_tab_indicator</item>
```

```
</style>
</resources>
```

支持 **Android 2.1** 和更高

当使用 Support 库时，你的式样 XML 文件应该是这样的：

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@style/Theme.AppCompat">
        <item name="android:actionBarTabStyle">@style/MyActionBarTabs</item>

        <!-- Support library compatibility -->
        <item name="actionBarTabStyle">@style/MyActionBarTabs</item>
    </style>

    <!-- ActionBar tabs styles -->
    <style name="MyActionBarTabs"
        parent="@style/Widget.AppCompat.ActionBar.TabView">
        <!-- tab indicator -->
        <item name="android:background">@drawable/actionbar_tab_indicator</item>

        <!-- Support library compatibility -->
        <item name="background">@drawable/actionbar_tab_indicator</item>
    </style>
</resources>
```

更多资源

- 关于 action bar 的更多式样属性，请查看 [Action Bar](#) 指南
- 学习更多式样的工作机制，请查看 [式样和主题](#) 指南
- 全面的 action bar 式样，请尝试 [Android Action Bar 式样生成器](#)

编写: [Vincent 4J](#)

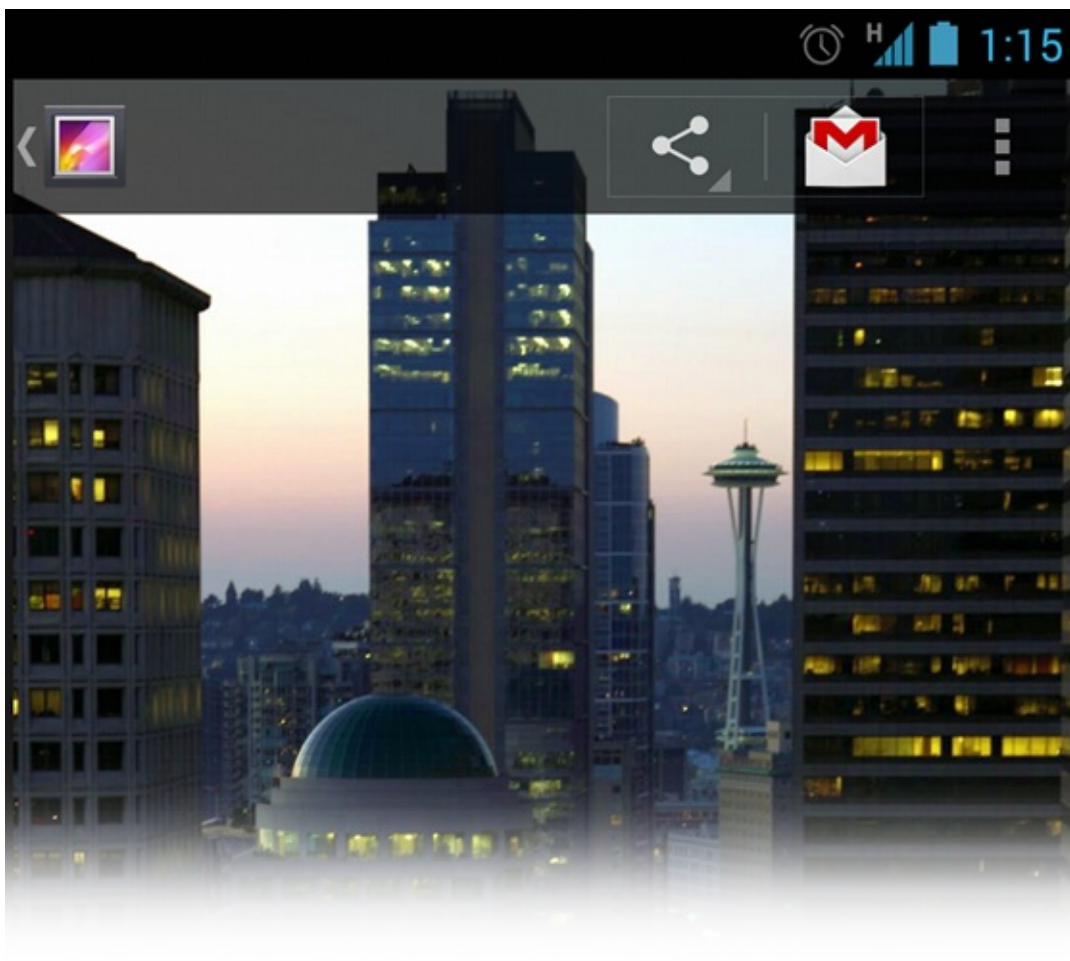
校对:

Action Bar 覆盖叠加

默认情况下，action bar 显示在 activity 窗口的顶部，会稍微地减少其他布局的有效空间。如果在用户交互过程中你要隐藏和显示 action bar，可以通过调用 [ActionBar](#) 中的 [hide\(\)](#) 和 [show\(\)](#) 来实现。但是，这将会导致 activity 基于新尺寸重现计算和重新绘制布局。

为了避免在 action bar 隐藏和显示过程中调整布局，可以为 action bar 启用叠加模式。在叠加模式下，所有可用的空间都会被用来布局，并且 action bar 会叠加在布局之上。这样布局顶部就会被遮挡，但当 action bar 隐藏或显示时，系统不再需要调整布局而是无缝过渡。

提示：如果你希望 action bar 下面的布局部分可见，可以创建一个背景部分透明的自定义式样的 action bar，如图 1 所示。如何定义 action bar 的背景，请查看 [ActionBar 风格化](#)。



的 gallery action bar

图 1. 叠加模式下

启用叠加模式

要为 action bar 启用叠加模式，需要自定义一个主题，该主题继承于已经存在的 action bar 主题，并设置 `android:windowActionBarOverlay` 属性的值为 `true`。

仅支持 **Android 3.0** 和以上

如果 [minSdkVersion](#) 为 11 或更高，自定义主题必须继承 [Theme.Holo](#) 主题（或者它的子主题）。例如：

```
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@android:style/Theme.Holo">
        <item name="android:windowActionBarOverlay">true</item>
    </style>
</resources>
```

支持 **Android 2.1** 和更高

如果为了兼容运行在 Android 3.0 以下版本的设备而使用了 Support 库，自定义主题必须继承 [Theme.AppCompat](#) 主题（或者它的子主题）。例如：

```
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme"
        parent="@android:style/Theme.AppCompat">
        <item name="android:windowActionBarOverlay">true</item>

        <!-- Support library compatibility -->
        <item name="windowActionBarOverlay">true</item>
    </style>
</resources>
```

请注意，这主题包含两种不同的 `windowActionBarOverlay` 样式定义：一个带 `android:` 前缀，另一个不带。带前缀的适用于包含该式样的 Android 版本，不带前缀的适用于通过从 Support 库中读取式样的旧版本。

指定布局的顶部边距

当 action bar 启用叠加模式时，它可能会遮挡住本应保持可见状态的布局。为了确保这些布局始终位于 action bar 下部，可以使用 [actionBarSize](#) 属性来指定顶部外边距或顶部内边距的高度来达到。例如：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?android:attr/actionBarSize">
    ...
</RelativeLayout>
```

如果在 action bar 中使用 Support 库，需要移除 android: 前缀。例如：

```
<!-- Support library compatibility -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="?attr/actionBarSize">
    ...
</RelativeLayout>
```

在这种情况下，不带前缀的 ?attr/actionBarSize 适用于 Android 3.0 和更高的所有版本。



编写:[Lin-H](#)

校对:

兼容不同的设备

全世界的Android设备有着各种各样的大小和尺寸。而通过各种各样的设备类型，能让你通过你的app接触到广大的用户群体。为了能在各种Android平台上使用，你的app需要兼容各种不同的设备类型。某些重要的变动你需要考虑，比如语言，屏幕尺寸，和Android的版本。

本课程会教你如何使用基础的平台功能，利用替代资源和其他功能，使你的app仅用一个app程序包(APK)，就能向用Android兼容设备的用户提供最优的用户体验。

课程

[适配不同的语言](#)

学习如何使用字符串替代资源实现支持多国语言。

[适配不同的屏幕](#)

学习如何根据不同尺寸分辨率的屏幕来优化用户体验。

[适配不同的系统版本](#)

学习如何在使用新的用户编程接口(API)时向下兼容旧版本Android。

编写:[Lin-H](#)

校对:

适配不同的语言

把UI中的字符串存储在外部文件，通过代码提取，总是一种很好的做法。Android可以通过工程中的资源目录轻松实现这一功能。

如果你使用Android SDK Tools(详见[Creating an Android Project](#))来创建工程，则在工程的根目录会创建一个res/的目录，目录中包含所有资源类型的子目录。其中包含工程的默认文件比如res/values/strings.xml，用来保存你的字符串值。

创建区域设置目录和字符串文件

为了支持多国语言，在res/中创建一个额外的values目录以连字符和ISO国家代码结尾命名，比如values-es/ 是包含简单的区域资源，语言代码为"es"的区域设置目录。Android会在运行时根据设备的区域设置，加载相应的资源。

若你决定支持某种语言，则需要创建资源子目录和字符串资源文件，例如：

```
MyProject/  
  res/  
    values/  
      strings.xml  
    values-es/  
      strings.xml  
    values-fr/  
      strings.xml
```

添加不同区域语言的字符串值到相应的文件。

在运行时，Android系统会根据用户设备当前的区域设置，使用相应的字符串资源。

例如 下面列举了几个不同语言对应不同的字符串资源文件。

英语(默认区域语言)，/values/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
  <string name="title">My Application</string>  
  <string name="hello_world">Hello World!</string>  
</resources>
```

西班牙语，/values-es/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
  <string name="title">Mi Aplicación</string>  
  <string name="hello_world">Hola Mundo!</string>  
</resources>
```

法语，/values-fr/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
  <string name="title">Mon Application</string>  
  <string name="hello_world">Bonjour le monde !</string>  
</resources>
```

Note : 你可以在任何资源类型中使用区域修饰词(或者任何配置修饰符)，比如给bitmap提供本地化的版本，更多信息见[Localization](#)。

使用字符资源

你可以在你的源代码和其他XML文件中，通过<string>元素的name属性来引用你的字符串资源。

在你的源代码中你可以通过R.string.<string_name>语法来引用一个字符串资源，很多方法都可以通过这种方式来接受字符串。

例如:

```
// 从你的 app's 资源中获取一个字符串资源
String hello = getResources().getString(R.string.hello_world);

// 或者提供给一个需要字符串作为参数的方法
TextView textView = new TextView(this);
textView.setText(R.string.hello_world);
```

在其他XML文件中，每当XML属性要接受一个字符串值时，你都可以通过@string/<string_name>语法来引用字符串资源。

例如:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

编写: [Lin-H](#)

校对:

适配不同的屏幕

Android将设备屏幕归类为两种常规属性：尺寸和分辨率。你应该想到你的app会被安装在各种屏幕尺寸和分辨率的设备中。这样，你的app就应该包含一些可选资源，针对不同的屏幕尺寸和分辨率，来优化你的app外观。

- 有4种普遍尺寸：小(small)，普通(normal)，大(large)，超大(xlarge)
- 4种普遍分辨率：低精度(ldpi), 中精度(mdpi), 高精度(hdpi), 超高精度(xhdpi)

声明针对不同屏幕所用的layout和bitmap，你必须把这些可选资源放置在独立的目录中，与你适配不同语言时的做法类似。

同样要注意屏幕的方向(横向或纵向)也是一种需要考虑的屏幕尺寸变化，所以许多app会修改layout，来针对不同的屏幕方向优化用户体验。

创建不同的layout

为了针对不同的屏幕去优化用户体验，你需要对每一种将要支持的屏幕尺寸，创建唯一的XML文件。每一种layout需要保存在相应的资源目录中，目录以-<screen_size>为后缀命名。例如，对大尺寸屏幕(large screens)，一个唯一的layout文件应该保存在res/layout-large/中。

Note:为了匹配合适的屏幕尺寸Android会自动地测量你的layout文件。所以你不需因不同的屏幕尺寸去担心UI元素的大小，而应该专注于layout结构对用户体验的影响。(比如关键视图相对于同级视图的尺寸或位置)

例如，这个工程包含一个默认layout和一个适配大屏幕的layout：

```
MyProject/  
  res/  
    layout/  
      main.xml  
    layout-large/  
      main.xml
```

layout文件的名字必须完全一样，为了对相应的屏幕尺寸提供最优的UI，文件的内容不同。

按照惯例在你的app中简单引用：

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
}
```

系统会根据你的app所运行的设备屏幕尺寸，在与之对应的layout目录中加载layout。更多关于Android如何选择恰当资源的信息，详见[Providing Resources](#)。

另一个例子，这一个工程中有为适配横向屏幕的layout:

```
MyProject/  
  res/  
    layout/  
      main.xml  
    layout-land/  
      main.xml
```

默认的，layout/main.xml文件用作纵向屏幕layout。

如果你想给横向屏幕提供一个特殊的layout，也适配于大屏幕，那么你需要使用large和land修饰符。

```
MyProject/  
  res/  
    layout/          # default (portrait)  
      main.xml  
    layout-land/     # landscape  
      main.xml  
    layout-large/    # large (portrait)  
      main.xml  
    layout-large-land/ # large landscape  
      main.xml
```

Note:Android 3.2和以上版本支持定义屏幕尺寸的高级方法，它允许你根据屏幕最小长度和宽度，为各种屏幕尺寸指定与密度无关的layout资源。这节课不会涉及这一新技术，更多信息详见[Designing for Multiple Screens](#)。

创建不同的bitmap

你应该为4种普遍分辨率:低, 中, 高, 超高精度, 都提供相适配的bitmap资源。这能帮助你在所有屏幕分辨率中都能有良好的画质和效果。

要生成这些图像, 你应该从原始的矢量图像资源着手, 然后根据下列尺寸比例, 生成各种密度下的图像。

- xhdp: 2.0
- hdpi: 1.5
- mdpi: 1.0 (基准)
- ldpi: 0.75

这意味着, 如果你针对超高密度的设备生成了一张200x200的图像, 同样的你应该对150x150 高密度,100x100中密度, 和75x75 低密度的设备生成同样的资源。

然后, 将这些文件放入相应的drawable资源目录中:

```
MyProject/  
  res/  
    drawable-xhdp/  
      awesomeimage.png  
    drawable-hdpi/  
      awesomeimage.png  
    drawable-mdpi/  
      awesomeimage.png  
    drawable-ldpi/  
      awesomeimage.png
```

任何时候, 当你引用@drawable/awesomeimage时系统会根据屏幕的分辨率选择恰当的bitmap。

Note:低密度(ldpi)资源是非必要的, 当你提供了高精度assets, 系统会把高密度图像按比例缩小一半, 去适配低密度屏幕。

更多关于为app创建图标assets的贴士和指导, 详见[Iconography design](#)。

编写: [Lin-H](#)

校对:

适配不同的系统版本

新的Android版本会为你的app提供更棒的APIs，但你的app仍应该支持旧版本的Android，直到更多的设备升级到新版本为止。这节课向你展示如何在利用新的APIs的同时仍支持旧版本Android。

[Platform Versions](#)的控制面板会定时更新，通过统计访问Google Play Store的设备数量，来显示运行每个版本的安卓设备的分布。一般情况下，在更新你的app至最新Android版本时，最好先保证你的新版app可以支持90%的设备使用。

Tip:为了能在几个Android版本中都能提供最好的特性和功能，你应该在你的app中使用[Android Support Library](#)，它能使你的app能在旧平台上使用最近的几个平台的APIs。

指定最小和目标API级别

[AndroidManifest.xml](#)文件中描述了你的app的细节，并且标明app支持哪些Android版本。具体来说，[<uses-sdk>](#)元素中的`minSdkVersion`和`targetSdkVersion`属性，标明在设计 and 测试app时，最低兼容API的级别和最高适用的API级别。

例如：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ... >
    <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15" />
    ...
</manifest>
```

随着新版本Android的发布，一些风格和行为可能会改变，为了能使你的app能利用这些变化，而且能适配不同风格的用户设备，你应该设置`targetSdkVersion`的值去匹配最新的可用Android版本。

在运行时检查系统版本

Android在[Build](#)常量类中提供了对每一个版本的唯一代号，在你的app中使用这些代号可以建立条件，保证依赖于高级别的API的代码，只会这些API在当前系统中可用时，才会执行。

```
private void setUpActionBar() {  
    // 保证我们是运行在Honeycomb或者更高版本时，才使用ActionBar APIs  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {  
        ActionBar actionBar = getActionBar();  
        actionBar.setDisplayHomeAsUpEnabled(true);  
    }  
}
```

Note:当解析XML资源时，Android会忽略当前设备不支持的XML属性。所以你可以安全地使用较新版本的XML属性，而不需要担心旧版本Android遇到这些代码时会崩溃。例如如果你设置`targetSdkVersion="11"`，你的app会在Android 3.0或更高时默认包含[ActionBar](#)。然后添加menu items到action bar时，你需要在你的menu XML资源中设置`android:showAsAction="ifRoom"`。在跨版本的XML文件中这么做是安全的，因为旧版本的Android会简单地忽略`showAsAction`属性(就是这样，你并不需要用到`res/menu-v11/`中单独版本的文件)。

使用平台风格和主题

Android提供了用户体验主题，为app提供基础操作系统的外观和体验。这些主题可以在manifest文件中被应用于你的app中.通过使用内置的风格和主题，你的app自然地随着Android新版本的发布，自动适配最新的外观和体验。

使你的activity看起来像对话框:

```
<activity android:theme="@android:style/Theme.Dialog">
```

使你的activity有一个透明背景:

```
<activity android:theme="@android:style/Theme.Translucent">
```

应用在/res/values/styles.xml中定义的自定义主题:

```
<activity android:theme="@style/CustomTheme">
```

使整个app应用一个主题(全部activities)在元素中添加android:theme属性:

```
<application android:theme="@style/CustomTheme">
```

更多关于创建和使用主题，详见[Styles and Themes](#)。

编写:[kesenhoo](#)

校对:

管理Activity的生命周期(Managing the Activity Lifecycle)

- 当用户进入，退出，回到你的App，在程序中的[Activity](#) 实例都经历了生命周期中的不同状态。例如，当你的activity第一次启动的时候，它来到系统的前台，开始接受用户的焦点。在此期间，Android系统调用了一系列的生命周期中的方法。如果用户执行了启动另一个activity或者切换到另一个app的操作，系统又会调用一些生命周期中的方法。
- 在生命周期的回调方法里面，你可以声明当用户离开或者重新进入这个Activity所需要执行的操作。例如，如果你建立了一个streaming video player，在用户切换到另外一个app的时候，你应该暂停video 并终止网络连接。当用户返回时，你可以重新建立网络连接并允许用户从同样的位置恢复播放。
- 这一章会介绍一些[Activity](#)中重要的生命周期回调方法，如何使用那些方法使得程序符合用户的期望且在activity不需要的时候不会导致系统资源的浪费。
- 完整的Demo示例：[ActivityLifecycle.zip](#)

Lessons

这一章我们将学习下面4个课程:

- [启动与销毁Activity](#)
- [暂停与恢复Activity](#)
- [停止与重启Activity](#)
- [重新创建Activity](#)

编写:[kesenhoo](#)

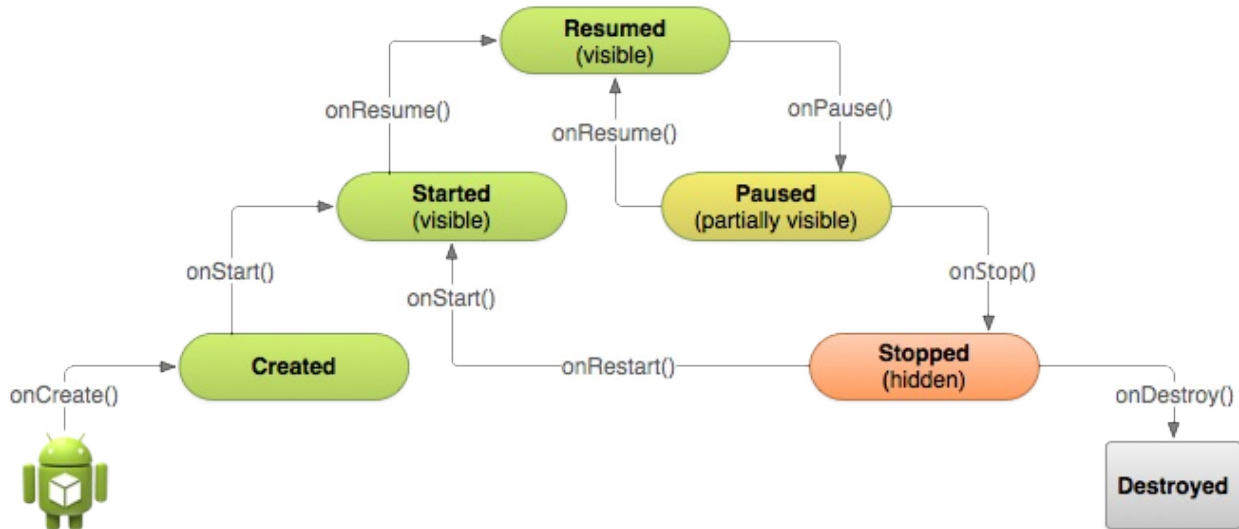
校对:

启动与销毁Activity

- 不像其他编程范式一样：程序从main()方法开始启动。Android系统根据生命周期的不同阶段唤起对应的回调函数来执行代码。系统存在启动与销毁一个activity的一套有序的回调函数。
- 这一个课会介绍那些生命周期中最重要的回调函数，并演示如何处理启动一个activity所涉及到的回调函数。

Understand the Lifecycle Callbacks[理解生命周期的回调]

- 在一个activity的生命周期中，系统会像金字塔模型一样去调用一系列的生命周期回调方法。Activity生命周期的每一个阶段就像金字塔中的台阶。当系统创建了一个新的activity实例，每一个回调函数会向上一阶的移动activity状态。金字塔顶端意味着activity是跑在最前端的并且用户可以与它进行交互。
- 当用户开始离开这个activity,为了卸载这个activity，系统会调用其它方法来向下一阶移动activity状态。在某些情况下，activity会隐藏在金字塔下等待(例如当用户切换到其他app),这个时候activity可以重新回到顶端(如果用户回到这个activity)并且恢复用户离开时的状态。
- Figure 1. 下面这张图讲解了activity的生命周期：(显然，这个金字塔模型要比之前Dev Guide里面的生命周期图更加容易理解，更加形象)



- 根据你的activity的复杂度，你也许不需要实现所有的生命周期方法。然而，你需要知道每一个方法的功能并确保你的app能够像用户期望的那样执行。如何实现一个符合用户期待的app，你需要注意下面几点：
 - 当使用你的app的时候，不会因为来电或者切换到其他app而导致程序crash。
 - 当用户没有激活某个组件的时候不要消耗宝贵的系统资源。
 - 当离开你的app并且一段时间后返回，不要丢失用户的使用进度。
 - 当设备发送屏幕旋转的时候，不会crash或者丢失用户的使用进度。
- 在下面的课程中会介绍上图所示的几个生命状态。然而，其中只有三个状态是静态的，这三个状态下activity可以存在一段比较长的时间。(其它几个状态会很快就切换掉，停留的时间比较短暂)
 - **Resumed**：在这个状态，activity是在最前端的，用户可以与它进行交互。(通常也被理解为"running"状态)
 - **Paused**：在这个状态，activity被另外一个activity所遮盖：另外的activity来到最前面，但是半透明的，不会覆盖整个屏幕。被暂停的activity不会再接受用户的输入且不会执行任何代码。
 - **Stopped**：在这个状态，activity完全被隐藏，不被用户可见。可以认为是在后台。当stopped, activity实例与它的所有状态信息都会被保留，但是activity不能执行任何代码。
- 其它状态 (Created and Started)都是短暂的，系统快速的执行那些回调函数并通过执行下一阶段的回调函数移动到下一个状态。也就是说，在系统调用onCreate(), 之后会迅速调用onStart(), 之后再迅速执行onResume()。上面就是基本的activity生命周期。

Specify Your App's Launcher Activity[指定你的程序首次启动的Activity]

- 当用户从主界面点击你的程序图标时，系统会调用你的app里面的被声明为"launcher" (or "main") activity中的onCreate()方法。这个Activity被用来当作你的程序的主要进入点。
- 你可以在AndroidManifest.xml中定义哪个activity作为你的主activity。
- 这个main activity必须在manifest使用包括 MAIN action and LAUNCHER category 的 标签来声明。例如：

```
<activity android:name=".MainActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

- 如果你的程序中没有有一个activity声明了MAIN action 或者LAUNCHER category，那么在设备的主界面列表里面不会呈现你的app图标。

Create a New Instance[创建一个新的实例]

- 大多数app都包括许多不同的activities，这样使得用户可以执行不同的动作。不论这个activity是创建的主activity还是为了响应用户行为而新创建的，系统都会调用新的activity实例中的onCreate()方法。
- 你必须实现onCreate()方法来执行程序启动所需要的基本逻辑。
- 例如：下面的onCreate()方法演示了为了建立一个activity所需要的一些基础操作。如，声明UI元素，定义成员变量，配置UI等。(onCreate里面尽量少做事情，避免程序启动太久都看不到界面)

```
TextView mTextView; // Member variable for text view in the layout

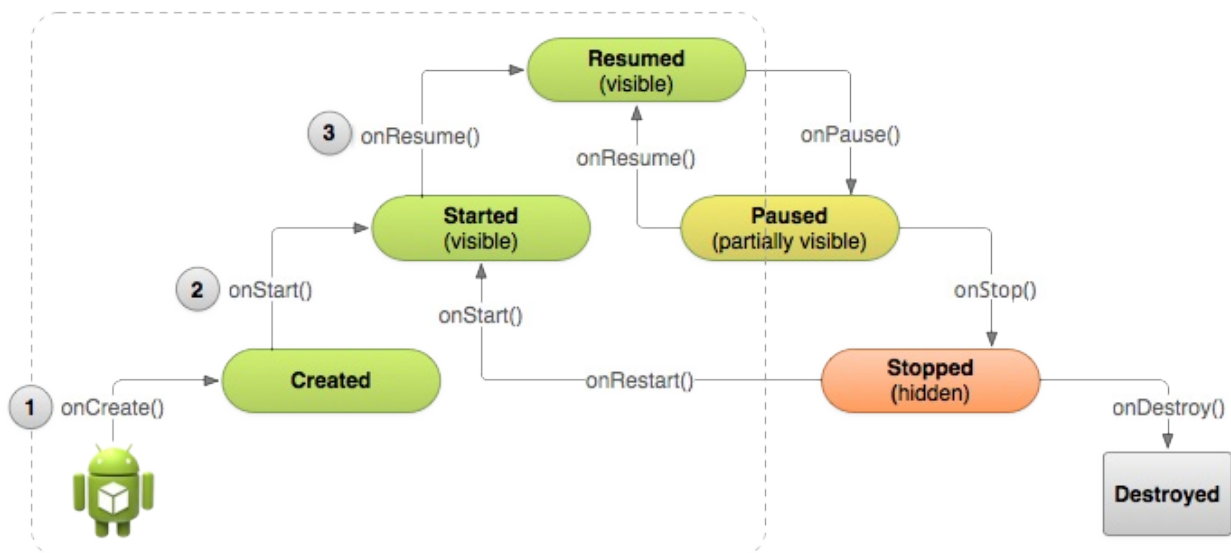
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);

    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

- 一旦结束onCreate 操作，系统会迅速调用onStart() 与onResume()方法。你的activity不会在Created或者Started状态停留。技术上来说，activity在onStart()被调用后会开始被用户可见，但是 onResume()会迅速被执行使得activity停留在Resumed状态，直到一些因素发生变化才会改变这个状态。例如接受到一个来电，用户切换到另外一个activity，或者是设备屏幕关闭。
- 在后面的课程中，你将看到其他方法是如何使用的，onStart() 与 onResume()在用户从Paused or Stopped状态中恢复的时候非常有用。
- **Note:** onCreate() 方法包含了一个参数叫做savedInstanceState，这将会在后面的课程：重新创建一个activity的时候涉及到。



- Figure 2. 上图显示了onCreate(), onStart(), and onResume()是如何执行的。当这三个顺序执行的回调函数完成后，activity会到达Resumed状态。

Destroy the Activity[销毁Activity]

- activity的第一个生命周期回调函数是 onCreate(),它最后一个回调是 onDestroy().系统会执行这个方法作为你的activity要从系统中完全移除的信号。
- 大多数 apps并不需要实现这个方法，因为局部类的references会被destroyed并且你的activity应该在 onPause() 与 onStop() 中执行清除的操作。然而，如果你的activity包含了你在onCreate时创建的后台线程，或者是其他有可能导致内存泄漏的资源，你应该在OnDestroy()时杀死他们。

```
@Override
public void onDestroy() {
    super.onDestroy(); // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

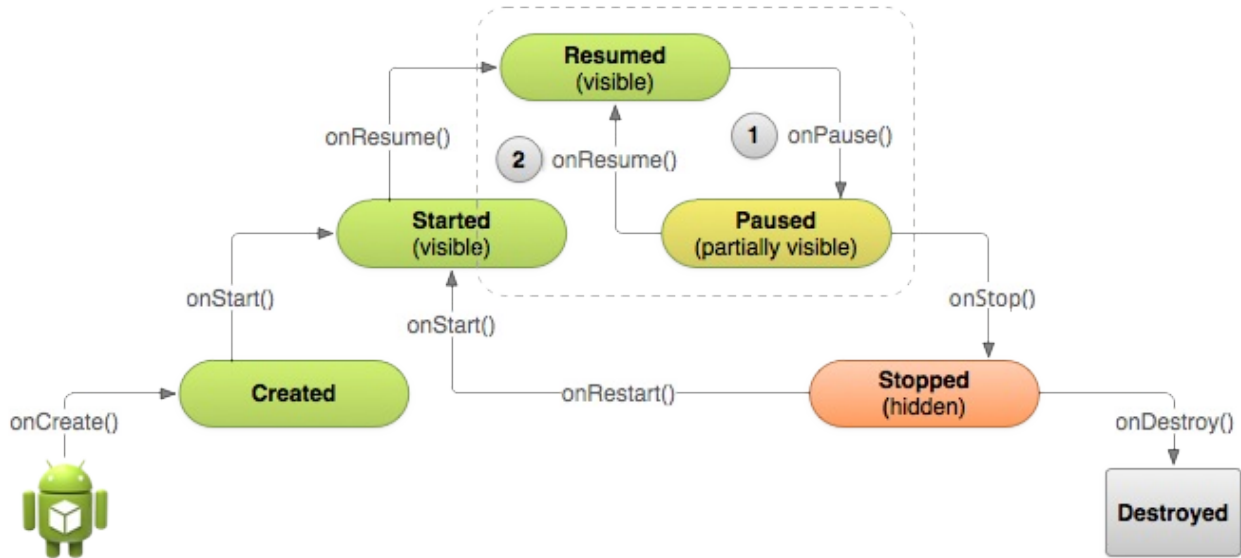
- **Note:** 系统通常是在执行了 onPause() and onStop() 之后再调用onDestroy()，除非你的程序里面再其他地方调用了 finish()方法，这样系统会直接就调用onDestory方法，其它生命周期的方法则不会被执行。

编写:[kesenhoo](#)

校对:

暂停与恢复Activity

- 在使用通常的app时，前端的activity有时候会被其他可见的组件而阻塞(obstructed)，这样会导致当前的activity进入Pause状态。例如，当打开一个半透明的activity时(such as one in the style of a dialog)，之前的activity会被暂停。只要这个activity仍然被部分可见，之前的activity则一直处于Paused状态。
- 然而，一旦之前的activity被完全阻塞并不可见，它则会进入Stop状态(which is discussed in the next lesson)。
- 当你的activity进入paused状态，系统会调用你的activity中的onPause()方法，这个方法会停止目前正在运行的操作，比如暂停视频播放。它会保存那些有可能需要长期保存的信息。(请注意是在onPause里面去保存那些信息)。如果用户从暂停状态回到你的activity，系统会恢复那些数据并执行onResume()方法。
- **Note:** 当你的activity接受到调用onPause()信息，那可能意味着activity将被暂停一段时间，并且用户很可能回到你的activity。然而，那也是用户要离开你的activity的第一个信号。
- **Figure 1.** 下图显示了，当一个半透明的activity阻塞你的activity时，系统会调用onPause()方法并且这个activity会停留在Paused state (1)。如果用户在这个activity还是在Paused State时回到这个activity，系统则会调用它的onResume() (2)。



Pause Your Activity[暂停你的Activity]

- 当系统调用你的activity中的onPause(),从技术上讲,那意味着你的activity仍然处于部分可见的状态,当时大多数时候,那意味着用户正在离开这个activity并马上会进入Stopped state. 你通常应该在onPause()回调方法里面做下面的事情:
 - 停止动画或者是其他正在运行的操作,那些都会导致CPU的浪费.
 - 提交没有保存的改变,但是仅仅是在用户离开时期待保存的内容(such as a draft email).
 - 释放系统资源,例如broadcast receivers, sensors (like GPS), 或者是其他任何会影响到电量的资源。
 - 例如,如果你的程序使用Camera,onPause()会是一个比较好的地方去做那些释放资源的操作。

```
@Override
public void onPause() {
    super.onPause(); // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release()
        mCamera = null;
    }
}
```

- 通常,你**not**应该使用onPause()来保存用户改变的数据(例如填入表格中的个人信息)到永久磁盘上。仅仅当你确认用户期待那些改变能够被自动保存的时候such as when drafting an email), 你可以把那些数据存到permanent storage。然而,你应该避免在onPause()时执行CPU-intensive的工作,例如写数据到DB,因为它会导致切换到下一个activity变得缓慢(你应该把那些heavy-load的工作放到onStop()去做)。
- 如果你的activity实际上是要被Stop,那么你应该为了切换的顺畅而减少在OnPause()方法里面的工作量。
- **Note:** 当你的activity处于暂停状态, [Activity](#)实例是驻留在内存中的,并且在activity恢复的时候重新调用。你不需要在恢复到Resumed状态的一系列回调方法中重新初始化组件。

Resume Your Activity[恢复你的activity]

- 当用户从Paused状态恢复你的activity时，系统会调用onResume()方法。
- 请注意，系统每次调用这个方法时，activity都处于最前台，包括第一次创建的时候。所以，你应该实现onResume()来初始化那些你在onPause方法里面释放掉的组件，并执行那些activity每次进入Resumed state都需要的初始化动作 (例如开始动画与初始化那些只有在获取用户焦点时才需要的组件)
- 下面的onResume()的例子是与上面的onPause()例子相对应的。

```
@Override
public void onResume() {
    super.onResume(); // Always call the superclass method first

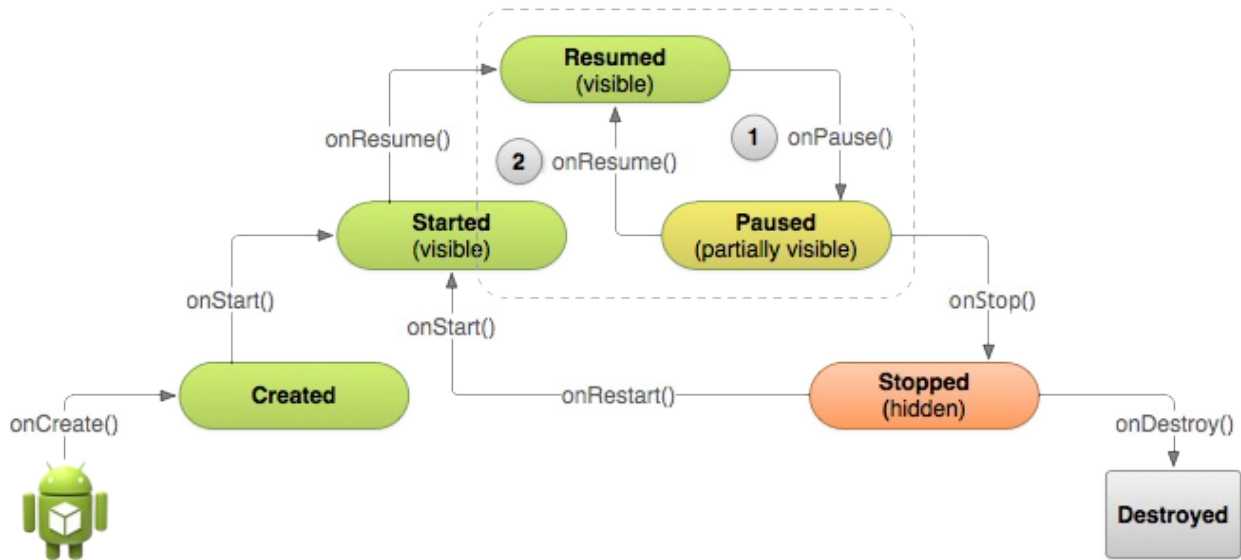
    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

编写:[kesenhoo](#)

校对:

停止与重启Activity

- 恰当的停止与重启你的activity是很重要的，在activity生命周期中，他们能确保用户感知到程序的存在并不会丢失他们的进度。在下面一些关键的场景中会涉及到停止与重启：
 - 用户打开最近使用app的菜单并切换你的app到另外一个app，这个时候你的app是被停止的。如果用户通过统一的办法回到你的app，那么你的activity会重启。
 - 用户在你的app里面执行启动一个新的activity的操作，当前activity会在第二个activity被创建后停止。如果用户点击back按钮，第一个activity会被重启。
 - 用户在使用你的app时接受到一个来电电话。
- [Activity](#)类提供了onStop()与onRestart(), 方法来允许你在activity停止与重启时进行调用。不像暂停状态是部分阻塞UI，停止状态是UI不在可见并且用户的焦点转移到另一个activity中。
- Note:**因为系统在activity停止时会在内存中保存了Activity实例。有些时候你不需要实现onStop(),onRestart()甚至是onStart()方法, 因为大多数的activity相对比较简单，activity会自己停止与重启，你只需要使用onPause()来停止正在运行的动作并断开系统资源链接。



- Figure 1.**上图显示：当用户离开你的activity，系统会调用onStop()来停止activity (1). 这个时候如果用户返回，系统会调用onRestart()(2), 之后会迅速调用onStart()与onResume()(4). 请注意：无论什么原因导致activity停止，系统总是会在onStop()之前调用onPause()方法。

Stop Your Activity[停止你的activity]

- 当你的activity调用onStop()方法, activity不再可见, 并且应该释放那些不再需要的所有资源。一旦你的activity停止了, 系统会在不再需要这个activity时摧毁它的实例。在极端情况下, 系统会直接杀死你的app进程, 并且不执行activity的onDestroy()回调方法, 因此你需要使用onStop()来释放资源, 从而避免内存泄漏。(这点需要注意)
- 尽管onPause()方法是在onStop()之前调用, 你应该使用onStop()来执行那些CPU intensive的shut-down操作, 例如writing information to a database.
- 例如, 下面是一个在onStop()的方法里面保存笔记草稿到persistent storage的示例:

```
@Override
protected void onStop() {
    super.onStop(); // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
        mUri,      // The URI for the note to update.
        values,    // The map of column names and new values to apply to them.
        null,      // No SELECT criteria are used.
        null       // No WHERE columns are used.
    );
}
```

- 当你的activity已经停止, [Activity](#) 对象会保存在内存中, 并且在activity resume的时候重新被调用到。你不需要在恢复到 Resumed state 状态前重新初始化那些被保存在内存中的组件。系统同样保存了每一个在布局中的视图的当前状态, 如果用户在EditText组件中输入了text, 它会被保存, 因此不需要保存与恢复它。
- **Note:** 即时系统会在activity stop的时候销毁这个activity, 它仍然会保存View objects (such as text in an [EditText](#)) 到一个 [Bundle](#) 中, 并且在用户返回这个activity时恢复他们(下一个会介绍在activity销毁与重新建立时如何使用 [Bundle](#) 来保存其他数据的状态)。

Start/Restart Your Activity[启动与重启你的activity]

- 当你的activity从Stopped状态回到前台时，它会调用onRestart().系统再调用onStart()方法，onStart()方法会在每次你的activity可见时都会被调用。onRestart()方法则是只在activity从stopped状态恢复时才会被调用，因此你可以使用它来执行一些特殊的恢复(restoration)工作，请注意之前是被stopped而不是destroy。
- 使用onRestart()来恢复activity状态是不太常见的，因此对于这个方法如何使用没有任何的guidelines。然而，因此你的onStop()方法应该做清除所有activity资源的操作，你将会在重新启动activity时re-instantiate那些被清除的资源，同样，你也需要在activity第一次创建时instantiate那些资源。介于上面的原因，你应该使用onStart()作为onStop()所对应方法。因为系统会在创建activity与从停止状态重启activity时都会调用onStart()。(这个地方的意思应该是说你在onStop里面做了哪些清除的操作就应该在onStart里面重新把那些清除掉的资源重新创建出来)
- 例如：因为用户很可能在回到这个activity之前需要过一段时间，所以onStart()方法是一个比较好的地方用来验证某些必须的功能是否已经Ready。

```
@Override
protected void onStart() {
    super.onStart(); // Always call the superclass method first

    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
}

@Override
protected void onRestart() {
    super.onRestart(); // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

- 当系统Destory你的activity，它会为你的activity调用onDestroy()方法。因为我们会在onStop方法里面做释放资源的操作，那么onDestory方法则是你最后去清除那些可能导致内存泄漏的地方。因此你需要确保那些线程都被destroyed并且所有的操作都被停止。

编写:[kesenhoo](#)

校对:

重新创建Activity

- 有几个场景中,Activity是由于正常的程序行为而被Destory的,例如当用户点击返回按钮或者是你的Activity通过调用finish()来发出停止信号。系统也有可能会在你的Activity处于stop状态且长时间不被使用,或者是在前台activity需要更多系统资源的时候把关闭后台进程,这样来获取更多的内存。
- 当你的Activity是因为用户点击Back按钮或者是activity通过调用finish()结束自己时,系统就丢失了Activity实例这个概念,因为前面的行为意味着不再需要这个activity了。然而,如果因为系统资源紧张而导致Activity的Destory,系统会在用户回到这个Activity时有这个Activity存在过的记录,系统会使用那些保存的记录数据(描述了当Activity被Destory时的状态)来重新创建一个新的Account实例。那些被系统用来恢复之前状态而保存的数据被叫做 "instance state", 它是一些存放在 Bundle 对象中的key-value pairs。
- **Caution:**你的Activity会在每次旋转屏幕时被destroyed与recreated。当屏幕改变方向时,系统会Destory与Recreate前台的activity,因为屏幕配置被改变,你的Activity可能需要加载一些alternative的资源(例如layout)。
- 默认情况下,系统使用 Bundle 实例来保存每一个视图对象中的信息(例如输入EditText 中的文本内容)。因此,如果你的Activity被destroyed与recreated,那么layout的状态信息会自动恢复到之前的状态。然而,你的activity也许存在更多你想要恢复的状态信息,例如记录用户Progress的成员变量(member variables)。
- 请注意为了让你可以保存额外的数据到saved instance state。在Activity的声明周期里面存在一个添加的回调函数。这个回调函数并没有在前面课程的图片示例中显示。这个方法是 onSaveInstanceState(), 当用户离开你的Activity时,系统会调用它。当系统调用这个函数时,系统会在你的Activity被异常Destory时传递 Bundle 对象,这样你可以增加额外的信息到Bundle中并保存与系统中。然后如果系统在Activity被Destory之后想重新创建这个Activity实例时,之前的那个Bundle对象会(系统)被传递到你的activity的 onRestoreInstanceState() 方法与 onCreate() 方法中。



上图：当系统开始停止你的Activity时，会调用到[onSaveInstanceState\(\)](#) (1)，因此你可以在Activity实例需要重新创建的情况下，指定特定的附加状态数据到Bunde中。如果这个Activity被destroyed而且同样的实例被重新创建，系统会传递在 (1) 中的状态数据到 onCreate() (2) 与 [onRestoreInstanceState\(\)](#)(3)。

Save Your Activity State[保存Activity状态]

- 当你的activity开始Stop，系统会调用 `onSaveInstanceState()`，因此你的Activity可以用键值对的集合来保存状态信息。这个方法会默认保存Activity视图的状态信息，例如在 `EditText` 组件中的文本或者是 `ListView` 的滑动位置。
- 为了给Activity保存额外的状态信息，你必须实现`onSaveInstanceState()` 并增加key-value pairs到 `Bundle` 对象中，例如：

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

- Caution: 总是需要调用 `onSaveInstanceState()` 方法的父类实现，这样默认的父亲实现才能保存视图状态的信息。

Restore Your Activity State [恢复Activity状态]

- 当你的Activity从Destory中重建。你可以从系统传递给你的Activity的Bundle中恢复保存的状态。 onCreate() 与 onRestoreInstanceState() 回调方法都接收到了同样的Bundle，里面包含了同样的实例状态信息。
- 因为 onCreate() 方法会在第一次创建新的Activity实例与重新创建之前被Destory的实例时都被调用，你必须在你尝试读取 Bundle 对象前Check它是否为null。如果它为Null，系统则是创建一个新的Activity instance，而不是恢复之前被Destory的Activity。
- 下面是一个示例：演示在onCreate方法里面恢复一些数据：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

- 你也可以选择实现 onRestoreInstanceState()，而不是在onCreate方法里面恢复数据。 onRestoreInstanceState()方法会在 onStart() 方法之后执行. 系统仅仅会在存在需要恢复的状态信息时才会调用 onRestoreInstanceState()，因此你不需要检查 Bundle 是否为Null:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

- Caution: 与上面保存一样，总是需要调用onRestoreInstanceState()方法的父类实现，这样默认的父亲实现才能保存视图状态的信息。如果想了解更多关于运行时状态改变引起的recreate你的activity。请参考[Handling Runtime Changes](#).

编写:[fastcome1985](#)

校对:

使用Fragment建立动态UI

- 为了在Android上创建动态的、多窗口的用户交互体验，你需要将UI组件封装成模块化进行使用,在activity中你可以对这些模块进行切入切出操作。你可以用[Fragment](#)来创建这些模块，Fragment就像一个嵌套的activity,拥有自己的布局（layout）以及管理自己的生命周期。
- 如果一个fragment定义了自己的布局，那么在activity中它可以与其他的fragments生成不同的组合，从而为不同的屏幕尺寸生成不同的布局（一个小的屏幕一次只放一个fragment，大的屏幕则可以两个或以上的fragment）。
- 这一章将向你展示如何用fragment来创建动态的用户体检，以及在不同屏幕尺寸的设备上优化你的APP的用户体验。像运行着android1.6这样老版本的设备，也都将继续得到支持。
- 完整的Demo示例：[FragmentBasics.zip](#)

Lessons

- [创建一个fragment](#) 学习如何创建一个fragment，以及实现它生命周期内的基本功能。
- [构建复杂的UI](#) 学习在APP内，对不同的屏幕尺寸用fragments构建不同的布局。
- [与其他fragments交互](#) 学习fragment与activity以及其他fragments之间交互。

编写:[fastcome1985](#)

校对:

创建一个Fragment

- 你可以把fragment想象成activity中一个模块化的部分，它拥有自己的生命周期，接收自己的输入事件，可以在activity运行过程中添加或者移除（有点像"子activity"，你可以在不同的activities里面重复使用）。这一课教你继承[Support Library](#) 中的[Fragment](#)，以使你的应用在Android1.6这样的低版本上仍能保持兼容。注意：如果你的APP的最低API版本是11或以上，你不必使用Support Library，你可以直接使用API框架里面的[Fragment](#)，这节课主要是讲基于Support Library的API，Support Library有一个特殊的包名，有时候与平台版本的API名字有些轻微的不一样。
- 在开始这节课前，你必须先让你的项目引用Support Library。如果你没有使用过Support Library，你可以根据文档[Support Library Setup](#) 来设置你的项目使用Support Library。当然，你也可以使用包含[action bar](#)的 v7 appcompat library。v7 appcompat library 兼容Android2.1(API level 7),也包含[Fragment](#) APIs。

创建一个Fragment类

- 创建一个fragment,首先需要继承[Fragment](#)类，然后在关键的生命周期方法中插入你APP的逻辑，就像[activity](#)一样。
- 其中一个区别是当你创建[Fragment](#)的时候，你必须重写[onCreateView\(\)](#)回调方法来定义你的布局。事实上，这是使Fragment运行起来，唯一一个需要你重写的回调方法。比如，下面是一个自定义布局的示例fragment。

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;

public class ArticleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.article_view, container, false);
    }
}
```

- 就像activity一样，当fragment从activity添加或者移除、当activity生命周期发生变化时，fragment应该是实现生命周期回调来管理它的状态。例如，当activity的[onPause\(\)](#)被调用时，它里面的所有fragment的[onPause\(\)](#)方法也会被触发。

用XML将fragment添加到activity

- fragments是可以重用的，模块化的UI组件，每一个Fragment的实例都必须与一个[FragmentActivity](#)关联。你可以在activity的XML布局文件中定义每一个fragment来实现这种关联。

注意：[FragmentActivity](#)是Support Library提供的一个特殊activity，用来在API11版本以下的系统上处理fragment。如果你APP中的最低版本大于等于11，你可以使用普通的[Activity](#)。

- 下面是一个XML布局的例子，当屏幕被认为是large(用目录名称中的large字符来区分)时，它在布局中增加了两个fragment。

res/layout-large/news_articles.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <fragment android:name="com.example.android.fragments.HeadlinesFragment"
        android:id="@+id/headlines_fragment"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />

    <fragment android:name="com.example.android.fragments.ArticleFragment"
        android:id="@+id/article_fragment"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />

</LinearLayout>
```

小贴士：更多关于不同屏幕尺寸创建不同布局的信息，请阅读[Supporting Different Screen Sizes](#)

- 然后将这个布局文件用到你的activity中。

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);
    }
}
```

- 如果你用的是 [v7 appcompat library](#)，你的activity应该改为继承[ActionBarActivity](#)，ActionBarActivity是FragmentActivity的一个子类（更多关于这方面的内容，请阅读[Adding the Action Bar](#)）。

注意：当你用XML布局文件的方式将Fragment添加进activity时，你的Fragment是不能被动态移除的。如果你想要在用户交互的时候把fragment切入与切出，你必须在activity启动后，将fragment添加进activity.这将在下节课讲到。

编写:[fastcome1985](#)

校对:

建立灵活动态的UI

- 如果你的APP设计成要支持范围广泛的屏幕尺寸时，在可利用的屏幕空间内，你可以通过在不同的布局配置中重用你的fragment来优化你的用户体验。
- 比如，一个手机设备可能适合一次只有一个fragment的单面板用户交互。相反，在更大屏幕尺寸的平板电脑上，你可能更想要两个fragment并排在一起，用来向用户展示更多信息。



图1：两个fragments，在同一个activity不同屏幕尺寸中用不同的配置来展示。在大屏幕上，两个fragment被并排放置，但是在手机上，一次只放置一个fragment，所以在用户导航中，两个fragment必须进行替换。

- [FragmentManager](#)类提供了方法，让你在activity运行时能够对fragment进行添加，移除，替换，以创建动态的用户体验。

在activity运行时添加fragment

- 比起在activity的布局文件中定义fragments,就像[上节课](#)说的用标签,你也可以在activity运行时动态添加fragment,如果你在打算在activity的生命周期内替换fragment, 这是必须的。
- 为了执行fragment的增加或者移除操作, 你必须用 [FragmentManager](#) 创建一个 [FragmentTransaction](#) 对象, FragmentTransaction提供了用来增加、移除、替换以及其它一些操作的APIs。
- 如果你的activity允许fragments移除或者替换, 你应该在activity的[onCreate\(\)](#)方法中添加初始化的fragment(s).
- 运用fragment (除了那些你在运行时添加的) 的一个很重要的规则就是在布局中你必须有一个容器[view](#), fragment将会放在这个view里面。
- 下面的这个布局是[上节课](#)的一次只显示一个fragment的布局的替代布局。为了从一个布局替换为另外一个布局, activity的布局包含了一个空的 [FrameLayout](#)作为fragment的容器。
- 注意文件名与上节课的布局一样, 但是文件目录没有large标识, 所以你的布局将会在比large小的屏幕上被使用, 因为这个屏幕无法满足同时放置两个fragments

res/layout/news_articles.xml:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

- 在你的activity里面, 用Support Library APIs调用 [getSupportFragmentManager\(\)](#)方法获取[FragmentManager](#) 对象, 然后调用 [beginTransaction\(\)](#) 方法创建一个 [FragmentTransaction](#) 对象, 然后调用[add\(\)](#)方法添加一个fragment.
- 你可以使用同一个 [FragmentTransaction](#)进行多次fragment事物。当你完成这些变化操作的时候, 必须调用[commit\(\)](#)方法。

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.news_articles);

        // Check that the activity is using the layout version with
        // the fragment_container FrameLayout
        if (findViewById(R.id.fragment_container) != null) {

            // However, if we're being restored from a previous state,
            // then we don't need to do anything and should return or else
            // we could end up with overlapping fragments.
            if (savedInstanceState != null) {
                return;
            }

            // Create a new Fragment to be placed in the activity layout
            HeadlinesFragment firstFragment = new HeadlinesFragment();

            // In case this activity was started with special instructions from an
            // Intent, pass the Intent's extras to the fragment as arguments
            firstFragment.setArguments(getIntent().getExtras());

            // Add the fragment to the 'fragment_container' FrameLayout
            getSupportFragmentManager().beginTransaction()
                .add(R.id.fragment_container, firstFragment).commit();
        }
    }
}
```

- 当fragment在activity运行时被添加进来时 (不是在XML布局中用定义的), activity 可以移除这个fragment或者用另外一个来替换它。

Fragment替换

- 替换fragment的过程与添加过程类似，只需要将[add\(\)](#)方法替换为 [replace\(\)](#)方法。
- 记住当你执行fragment事物时，例如移除或者替换，你经常要适当地让用户可以向后导航与"撤销"这次改变。为了让用户向后导航fragment事物，你必须在[FragmentTransaction](#)提交前调用[addToBackStack\(\)](#)方法。

注意：当你移除或者替换一个fragment并把它放入返回栈中时，被移除的fragment的生命周期是stopped(不是destroyed).当用户返回重新恢复这个fragment,它的生命周期是restarts。如果你没把fragment放入返回栈中，那么当他被移除或者替换时，它的生命周期是 destroyed。

- 下面是一个fragment替换的例子

```
// Create fragment and give it an argument specifying the article it should show
ArticleFragment newFragment = new ArticleFragment();
Bundle args = new Bundle();
args.putInt(ArticleFragment.ARG_POSITION, position);
newFragment.setArguments(args);

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this fragment,
// and add the transaction to the back stack so the user can navigate back
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

- [addToBackStack\(\)](#)方法提供了一个可选的String参数为事物指定了一个唯一的名字。这个名字不是必须的，除非你打算用[FragmentManager.BackStackEntry](#) APIs来进行一些高级的fragments操作。

编写:[fastcome1985](#)

校对:

Fragments之间的交互

- 为了重用Fragment UI组件，你应该把每一个fragment都构建成为完全的自包含的、模块化的组件，定义他们自己的布局与行为。当你定义好这些模块化的Fragments的时，你就可以让他们关联activity，使他们与Application的逻辑结合起来，实现全局的复合的UI。
- 经常地，你想fragment之间能相互交互，比如基于用户事件改变fragment的内容。所有fragment之间的交互需要通过他们关联的activity，两个fragment之间不应该直接交互。

定义一个接口

- 为了让fragment与activity交互，你可以在Fragment 类中定义一个接口，并且在activity中实现这个接口。Fragment在他们生命周期的onAttach()方法中捕获接口的实现，然后调用接口的方法来与Activity交互。

下面是一个fragment与activity交互的例子：

```
public class HeadlinesFragment extends ListFragment {
    OnHeadlineSelectedListener mCallback;

    // Container Activity must implement this interface
    public interface OnHeadlineSelectedListener {
        public void onArticleSelected(int position);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);

        // This makes sure that the container activity has implemented
        // the callback interface. If not, it throws an exception
        try {
            mCallback = (OnHeadlineSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString()
                + " must implement OnHeadlineSelectedListener");
        }
    }

    ...
}
```

- 现在Fragment就可以通过调用OnHeadlineSelectedListener接口实例的mCallback中的onArticleSelected()（也可以是其它方法）方法与activity传递消息。
- 举个例子，在fragment中的下面的方法在用户点击列表条目时被调用，fragment用回调接口来传递事件给父Activity。

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    // Send the event to the host activity
    mCallback.onArticleSelected(position);
}
```

实现接口

- 为了接收回调事件，宿主activity必须实现在Fragment中定义的接口。
- 举个例子，下面的activity实现了上面例子中的接口。

```
public static class MainActivity extends Activity
    implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // The user selected the headline of an article from the HeadlinesFragment
        // Do something here to display that article
    }
}
```

传消息给Fragment

- 宿主activity通过[findFragmentById\(\)](#)方法来获取[fragment](#)的实例，然后直接调用Fragment的public方法来向fragment传递消息。
- 例如，想象一下，上面所示的activity可能包含另外一个fragment,这个fragment用来展示从上面的回调方法中返回的指定的数据。在这种情况下，activity可以把从回调方法中接收到的信息传递给这个展示数据的Fragment。

```
public static class MainActivity extends Activity
    implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // The user selected the headline of an article from the HeadlinesFragment
        // Do something here to display that article

        ArticleFragment articleFrag = (ArticleFragment)
            getSupportFragmentManager().findFragmentById(R.id.article_fragment);

        if (articleFrag != null) {
            // If article frag is available, we're in two-pane layout...

            // Call a method in the ArticleFragment to update its content
            articleFrag.updateArticleView(position);
        } else {
            // Otherwise, we're in the one-pane layout and must swap frags...

            // Create fragment and give it an argument for the selected article
            ArticleFragment newFragment = new ArticleFragment();
            Bundle args = new Bundle();
            args.putInt(ArticleFragment.ARG_POSITION, position);
            newFragment.setArguments(args);

            FragmentTransaction transaction = getSupportFragmentManager().beginTransaction()

            // Replace whatever is in the fragment_container view with this fragment,
            // and add the transaction to the back stack so the user can navigate back
            transaction.replace(R.id.fragment_container, newFragment);
            transaction.addToBackStack(null);

            // Commit the transaction
            transaction.commit();
        }
    }
}
```

编写:[kesenhoo](#)

校对:

数据保存

虽然可以在onPause()的时候保存一些信息以免用户的使用进度被丢失，但是大多数Android app仍然是需要做保存数据的动作。大多数比较好的apps都需要保存用户的设置信息，而且有一些apps必须维护大量的文件信息与DB信息。这一章节会介绍给你在Android中一些重要的数据存储方法，例如：

- [以key-value的方式保存一些简单的数据到shared preferences文件中](#)
- [在Android文件系统中保存任意格式的文件](#)
- [通过SQLite来使用DB](#)

编写:[kesenhoo](#)

校对:

Saving Key-Value Sets保存到Preference

如果你有一个相对较小的key-value集合需要保存，你应该使用[SharedPreferences](#) APIs。SharedPreferences 对象指向了一个保存 key-value pairs的文件，并且它提供了简单的方法来读写这个文件。每一个 SharedPreferences 文件都是由framework管理的并且可以是私有或者可分享的。这节课会演示如何使用 SharedPreferences APIs 来存储与检索简单的数据。**Note:** SharedPreferences APIs 仅仅提供了读写key-value对的功能，请不要与 Preference APIs相混淆。后者可以帮助你建立一个设置用户配置的页面（尽管它实际上是使用SharedPreferences 来实现保存用户配置的）。如果想了解更多关于Preference APIs的信息，请参考Settings 指南。

Get a Handle to a SharedPreferences [获取SharedPreferences的Handle]

你可以通过下面两个方法之一来创建或者访问shared preference 文件:

- **getSharedPreferences()** — 如果你需要多个通过名称参数来区分的shared preference文件, 名称可以通过第一个参数来指定。你可以在你的app里面通过任何一个Context 来执行这个方法。
- **getPreferences()** — 当你的activity 仅仅需要一个shared preference文件时。因为这个方法会检索activity下的默认的shared preference文件, 并不需要提供文件名称。

例如: 下面的示例是在 Fragment 中被执行的, 它会访问名为 R.string.preference_file_key 的shared preference文件, 并使用private 模式来打开它, 这样的话, 此时文件就仅仅可以被你的app访问了。

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
    getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

当命名你的shared preference文件时, 你应该像 "com.example.myapp.PREFERENCE_FILE_KEY" 这样来命名。

当然, 如果你的activity 仅仅需要一个shared preference文件时, 你可以使用[getPreferences\(\)](#)方法:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

Caution: 如果你创建了一个[MODE_WORLD_READABLE](#)或者[MODE_WORLD_WRITEABLE](#) 模式的shared preference文件, 那么任何其他的app只要知道文件名, 则可以访问这个文件。

Write to Shared Preferences[写Shared Preference]

为了写shared preferences文件，需要通过执行 `edit()` 来创建一个 `SharedPreferences.Editor`。

通过类似 `putInt()` 与 `putString()`方法来传递keys与values。然后执行 `commit()` 来提交改变。(后来有建议除非是出于线程同步的需要，否则请使用`apply()`方法来替代`commit()`，因为后者有可能会卡到`UI Thread`.)

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

Read from Shared Preferences[读Shared Preference]

为了从shared preference中检索读取数据，可以通过类似 getInt() 与 getString()等方法来读取。在那些方法里面传递你想要获取value对应的key，并且提供一个默认的值。如下：

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);  
long default = getResources().getInteger(R.string.saved_high_score_default);  
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), default);
```

编写:[kesenhoo](#)

校对:

Saving to Files保存到文件

Android使用与其他平台类似的基于磁盘文件系统(disk-based file systems)。这节课会描述如何在Android文件系统中使用 [File](#) 的读写APIs。File 对象非常适合用来读写那种流式顺序的数据。例如，很适合用来读写图片文件或者是网络中交换的数据。这节课会演示在app中如何执行基本的文件操作任务。假定你已经对linux的文件系统与java.io中标准的I/O APIs有一定认识。

Choose Internal or External Storage[选择Internal还是External存储]

所有的Android设备都有两个文件存储区域："internal" 与 "external" 存储。那两个名称来自与早先的Android系统中，那个时候大多数的设备都内置了不可变的内存（internal storage），然后再加上一个类似SD card（external storage）这样可以卸载的存储部件。后来有一些设备把"internal" 与 "external" 的部分都做成不可卸载的内置存储了，虽然如此，但是这一整块还是从逻辑上有被划分为"internal"与"external"的。只是现在不再以是否可以卸载来区分了。下面列出了两者的区别：

- **Internal storage:**

- 总是可用的
- 这里的文件默认是只能被你的app所访问的。
- 当用户卸载你的app的时候，系统会把internal里面的相关文件都清除干净。
- Internal是在你想确保不被用户与其他app所访问的最佳存储区域。

- **External storage:**

- 并不总是可用的，因为用户可以选择把这部分作为USB存储模式，这样就不可以访问了。
- 是大家都可以访问的，因此保存到这里的文件是失去访问控制权限的。
- 当用户卸载你的app时，系统仅仅会删除external根目录（getExternalFilesDir()）下的相关文件。
- External是在你不需要严格的访问权限并且你希望这些文件能够被其他app所共享或者是允许用户通过电脑访问时的最佳存储区域。

Tip: 尽管app是默认被安装到internal storage的，你还是可以通过在程序的manifest文件中声明android:installLocation 属性来指定程序也可以被安装到external storage。当某个程序的安装文件很大，用户会倾向这个程序能够提供安装到external storage的选项。更多安装信息，请参考[App Install Location](#)。

obtain Permissions for External Storage [获取External存储的权限]

为了写数据到external storage, 你必须在你的manifest文件中请求[WRITE_EXTERNAL_STORAGE](#)权限：

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

Caution:目前，所有的apps都可以在不指定写的权限下做读external storage的动作。但是，这会在以后的版本中被修正。如果你的app需要读的权限(不是写), 那么你需要声明 [READ_EXTERNAL_STORAGE](#) 权限。为了确保你的app能够在正常工作，你需要现在就声明读权限。但是，如果你的程序有声明写的权限，那么就默认有了读的权限。

```
<manifest ...>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    ...
</manifest>
```

对于internal storage，你不需要声明任何权限，因为你的程序默认就有读写程序目录下的文件的权限。

Save a File on Internal Storage[保存文件到Internal Storage]

当保存文件到internal storage时，你可以通过执行下面两个方法之一来获取合适的目录作为File的对象：

- **getFilesDir()** : 返回一个 File，代表了你的app的internal目录。
- **getCacheDir()** : 返回一个 File，代表了你的app的internal缓存目录。请确保这个目录下的文件在一旦不再需要的时候能够马上被删除，还有请给予一个合理的大小，例如1MB。如果系统的内存不够，会自行选择删除缓存文件。为了在那些目录下创建一个新的文件，你可以使用 File() 构造器，如下：

```
File file = new File(context.getFilesDir(), filename);
```

同样，你也可以执行[openFileOutput\(\)](#) 来获取一个 FileOutputStream 用来写文件到internal目录。如下：

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(string.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

如果，你需要缓存一些文件，你可以使用[createTempFile\(\)](#)。例如：下面的方法从URL中抽取了一个文件名，然后再创建了一个以这个文件名命名的文件。

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

Note: 你的app的internal storage 目录是以你的app的包名作为标识存放在Android文件系统的特定目录下 [data/data/com.example.xx]。从技术上讲，如果你设置文件为可读的，那么其他app就可以读取你的internal文件。然而，其他app需要知道你的包名与文件名。若是你没有设置为可读或者可写，其他app是没有办法读写的。因此只要你使用[MODE_PRIVATE](#)，那么这些文件就不可能被其他app所访问。

Save a File on External Storage [保存文件到External Storage]

因为external storage可能是不可用的，那么你应该在访问之前去检查是否可用。你可以通过执行 `getExternalStorageState()` 来查询external storage的状态。如果返回的状态是MEDIA_MOUNTED, 那么你可以读写。示例如下：

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

尽管external storage对与用户与其他app是可修改的，那么你可能会保存下面两种类型的文件。

- **Public files** :这些文件对与用户与其他app来说是public的，当用户卸载你的app时，这些文件应该保留。例如，那些被你的app拍摄的图片或者下载的文件。
- **Private files** :这些文件应该被你的app所拥有的，它们应该在你的app被卸载时删除掉。尽管那些文件从技术上可以被用户与其他app所访问，实际上那些文件对于其他app是没有意义的。所以，当用户卸载你的app时，系统会删除你的app的private目录。例如，那些被你的app下载的缓存文件。

如果你想要保存文件为public形式的，请使用[getExternalStoragePublicDirectory\(\)](#)方法来获取一个 File 对象来表示存储在external storage的目录。这个方法会需要你带有一个特定的参数来指定这些public的文件类型，以便于与其他public文件进行分类。参数类型包括[DIRECTORY_MUSIC](#) 或者 [DIRECTORY_PICTURES](#)。如下：

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

如果你想要保存文件为私有的方式，你可以通过执行[getExternalFilesDir\(\)](#) 来获取相应的目录，并且传递一个指示文件类型的参数。每一个以这种方式创建的目录都会被添加到external storage封装你的app目录下的参数文件夹下（如下则是albumName）。这下面的文件会在用户卸载你的app时被系统删除。如下示例：

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

如果刚开始的时候，没有预定义的子目录存放你的文件，你可以在 `getExternalFilesDir()`方法中传递 `null`。它会返回你的app在external storage下的private的根目录。

请记住， `getExternalFilesDir()` 方法会创建的目录会在app被卸载时被系统删除。如果你的文件想在app被删除时仍然保留，请使用[getExternalStoragePublicDirectory\(\)](#)。

不管你是使用 `getExternalStoragePublicDirectory()` 来存储可以共享的文件，还是使用 `getExternalFilesDir()` 来储存那些对与你的app来说是私有的文件，有一点很重要，那就是你要使用那些类似[DIRECTORY_PICTURES](#) 的API的常量。那些目录类型参数可以确保那些文件被系统正确的对待。例如，那些以[DIRECTORY_RINGTONES](#) 类型保存的文件就会被系统的media scanner认为是ringtone而不是音乐。

Query Free Space[查询剩余空间]

如果你事先知道你想要保存的文件大小，你可以通过执行[getFreeSpace\(\)](#) or [getTotalSpace\(\)](#) 来判断是否有足够的空间来保存文件，从而避免发生IOException。那些方法提供了当前可用的空间还有存储系统的总容量。

然而，系统并不会授权你写入通过getFreeSpace().查询到的容量文件，如果查询的剩余容易比你的文件大小多几MB，或者说文件系统使用率还不足90%，这样则可以继续进行写的操作，否则你最好不要写进去。

Note:你并没有强制要求在写文件之前一定不要去检查剩余容量。你可以尝试先做写的动作，然后通过捕获 IOException。这种做法仅适合于你并不知道你想要写的文件的确切大小。

Delete a File [删除文件]

你应该在不需要使用某些文件的时候，删除它。删除文件最直接的方法是直接执行文件的 `delete()` 方法。

```
myFile.delete();
```

如果文件是保存在internal storage，你可以通过 Context 来访问并通过执行`deleteFile()`进行删除

```
myContext.deleteFile(fileName);
```

Note: 当用户卸载你的app时，android系统会删除下面的文件：

- 所有保存到internal storage的文件。
- 所有使用`getExternalFilesDir()`方式保存在external storage的文件 然而，你应该手动删除所有通过 `getCacheDir()` 方式创建的缓存文件，还有那些通常来说不会再用的文件。

编写:[kesenhoo](#)

校对:

Saving to database保存到数据库

对于重复或者结构化的数据（如联系人信息）等保存到DB是个不错的主意。这节课假定你已经熟悉SQL数据库的操作。在Android上可能会使用到的APIs，可以从[android.database.sqlite](#)包中找到。

Define a Schema and Contract[定义Schema与Contract]

SQL中一个中重要的概念是schema：一种DB结构的正式声明。schema是从你创建DB的SQL语句中生成的。你可能会发现创建一个创建一个伴随类（companion class）是很有益的，这个类成为合约类（contract class），它用一种系统化并且自动生成文档的方式，显示指定了你的schema样式。

Contract Class是一些常量的容器。它定义了例如URIs, 表名, 列名等。这个contract类允许你在同一个包下与其他类使用同样的常量。它让你只需要在一个地方修改列名，然后这个列名就可以自动传递给你整个code。

一个组织你的contract类的好方法是在你的类的根层级定义一些全局变量，然后为每一个table来创建内部类。

Note:通过实现 [BaseColumns](#) 的接口，你的内部类可以继承到一个名为_ID的主键，这个对于Android里面的一些类似cursor adaptor类是很有必要的。这样能够使得你的DB与Android的framework能够很好的相容。

例如，下面的例子定义了表名与这个表的列名：

```
public static abstract class FeedEntry implements BaseColumns {
    public static final String TABLE_NAME = "entry";
    public static final String COLUMN_NAME_ENTRY_ID = "entryid";
    public static final String COLUMN_NAME_TITLE = "title";
    public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    ...
}
```

为了防止一些人不小心实例化contract类，像下面一样给一个空的构造器。

```
// Prevents the FeedReaderContract class from being instantiated.
private FeedReaderContract() {}
```

Create a Database Using a SQL Helper[使用SQL Helper创建DB]

当你定义好了你的DB应该是什么样之后，你应该实现那些创建与维护db与table的方法。下面是一些典型的创建与删除table的语句。

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedReaderContract.FeedEntry.TABLE_NAME + " (" +
    FeedReaderContract.FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    ")";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + TABLE_NAME_ENTRIES;
```

就像保存文件到设备的 internal storage 一样，Android会保存db到你的程序的private的空间上。你的数据是受保护的，因为那些区域默认是私有的，不可被其他程序所访问。

在`SQLiteOpenHelper`类中有一些很有用的APIs。当你使用这个类来做一些与你的db有关的操作时，系统会对那些有可能比较耗时的操作（例如创建与更新等）在真正需要的时候才去执行，而不是在app刚启动的时候就去做那些动作。你所需要做的仅是执行 `getWritableDatabase()` 或者 `getReadableDatabase()`。

Note:因为那些操作可能是很耗时的，请确保你在background thread（`AsyncTask` or `IntentService`）里面去执行 `getWritableDatabase()` 或者 `getReadableDatabase()`。

为了使用 `SQLiteOpenHelper`，你需要创建一个子类并重写 `onCreate()`, `onUpgrade()` 与 `onOpen()`等callback方法。你也许还需要实现 `onDowngrade()`，但是这并不是必需的。

例如，下面是一个实现了`SQLiteOpenHelper` 类的例子：

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

为了访问你的db，需要实例化你的 `SQLiteOpenHelper`的子类：

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Put Information into a Database [添加信息到DB]

通过传递一个 ContentValues 对象到 insert() 方法：

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
    FeedReaderContract.FeedEntry.TABLE_NAME,
    FeedReaderContract.FeedEntry.COLUMN_NAME_NULLABLE,
    values);
```

insert() 方法的第一个参数是table名，第二个参数会使得系统自动对那些 ContentValues 没有提供数据的列填充数据为null，如果第二个参数传递的是null，那么系统则不会对那些没有提供数据的列进行填充。

Read Information from a Database [从DB中读取信息]

为了从DB中读取数据，需要使用 query() 方法， 传递你需要查询的条件。查询后会返回一个 Cursor 对象。

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedReaderContract.FeedEntry._ID,
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE,
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED,
    ...
};

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedReaderContract.FeedEntry.TABLE_NAME, // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
);
```

下面是演示如何从course对象中读取数据信息：

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedReaderContract.FeedEntry._ID)
);
```


Delete Information from a Database [删除DB中的信息]

和查询信息一样，删除数据，同样需要提供一些删除标准。DB的API提供了一个防止SQL注入的机制来创建查询与删除标准。**SQL Injection**(随着B/S模式应用开发的发展，使用这种模式编写应用程序的程序员也越来越多。但是由于程序员的水平及经验也参差不齐，相当大一部分程序员在编写代码的时候，没有对用户输入数据的合法性进行判断，使应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果，获得某些他想得知的数据，这就是所谓的SQL Injection，即SQL注入)

这个机制把查询语句划分为选项条款与选项参数两部分。条款部分定义了查询的列是怎怎样的，参数部分用来测试是否符合前面的条款。(这里翻译的怪怪的，附上原文，*The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause.*) 因为处理的结果与通常的SQL语句不同，这样可以避免SQL注入问题。

```
// Define 'where' part of query.
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, mySelection, selectionArgs);
```

Update a Database [更新数据]

当你需要修改DB中的某些数据时，使用 update() 方法。

更新操作结合了插入与删除的语法。

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

编写:[kesenhoo](#)

校对:

与其他应用的交互

Outlines

- 一个Android app通常都会有好几个activities. 每一个activity的界面都可能允许用户执行一些特殊任务（例如查看地图或者是开始拍照等）。为了让用户从一个activity跳到另外一个activity，你的app必须使用Intent来定义你的app想做的事情。当你使用startActivity()的方法，而且参数是intent时，系统会使用这个 Intent 来定义并启动合适的app组件。使用 intents还可以让你的app来启动另外一个app里面的activity。
- 一个 Intent 可以显式的指明需要启动的模块，也可以隐式的指明自己可以处理哪种类型的动作。
- 这一章节会演示如何使用Intent 来做一些与其他app之间的简单交互。类似，启动另外一个app,从其他app接受数据，并且使得你的app能够响应从其他发出的intent。

Lessons

- [Sending the User to Another App:Intent的发送](#)

演示如何创建隐式的Intent来唤起能够接收这个动作的App。

- [Getting a Result from an Activity:接收Activity返回的结果](#)

演示如何启动另外一个Activity并接收返回值。

- [Allowing Other Apps to Start Your Activity:Intent过滤](#)

演示如何通过定义隐式的Intent的过滤器来使得能够被其他应用唤起。

编写:[kesenhoo](#)

校对:

Intent的发送

Android中最重要的功能之一就是可以利用一个带有**action**的**intent**使得当前app能够跳转到其他的app。例如：如果你的app拥有一个地址想要显示在地图上，你并不需要在你的app里面创建一个activity用来显示地图。你只需要使用Intent来发出查看地址的请求。Android系统则会启动能够显示地图的程序来呈现那个地址。

正如在2.1章节:[Building Your First App:建立你的第一个App](#)中所说的，你必须使用intent来在同一个app的两个activity之间进行切换。在那种情况下通常是定义一个显示（explicit）的intent，它指定了需要叫起组件。然而，当你想要叫起不同的app来执行那个动作，则必须使用隐式（implicit）的intent。

这节课会介绍如何为特殊的动作创建一个implicit intent，并使用它来启动另外一个app去执行intent中的action。

Build an Implicit Intent[建立一个隐式的Intent]

Implicit intents并不会声明需要启动的组件的类名，它使用的是声明一个需要执行的动作。这个action指定了你想做的事情，例如查看，编辑，发送或者是获取什么。Intents通常会在发送action的同时附带一些数据，例如你想要查看的地址或者是你想要发送的邮件信息。依赖于你想要创建的Intent，这些数据需要是Uri，或者是其他规定的数据类型。如果你的数据是一个Uri，会有一个简单的 Intent() constructor 用来定义 action与data。

例如，下面是一个带有指定电话号码的intent。

```
Uri number = Uri.parse("tel:5551234");
Intent callIntent = new Intent(Intent.ACTION_DIAL, number);
```

当你的app通过执行startActivity()来启动这个intent时，Phone app会使用之前的电话号码来拨出这个电话。

下面是一些其他intent的例子：

- View a map:

```
// Map point based on address
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
// Or map point based on latitude/longitude
// Uri location = Uri.parse("geo:37.422219,-122.08364?z=14"); // z param is zoom level
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
```

- View a web page:

```
Uri webpage = Uri.parse("http://www.android.com");
Intent webIntent = new Intent(Intent.ACTION_VIEW, webpage);
```

另外一些需要"extra"数据的implicit intent。你可以使用 putExtra() 方法来添加那些数据。默认的，系统会根据Uri数据类型来决定需要哪些合适的MIME type。如果你没有在intent中包含一个Uri，则通常需要使用 setType() 方法来指定intent附带的数据类型。设置MIME type是为了指定哪些activity可以应该接受这个intent。例如：

- Send an email with an attachment:

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
// The intent does not have a URI, so declare the "text/plain" MIME type
emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"jon@example.com"}); // recipients
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
emailIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("content://path/to/email/attachment"));
// You can also attach multiple items by passing an ArrayList of Uris
```

- Create a calendar event:

```
Intent calendarIntent = new Intent(Intent.ACTION_INSERT, Events.CONTENT_URI);
Calendar beginTime = Calendar.getInstance().set(2012, 0, 19, 7, 30);
Calendar endTime = Calendar.getInstance().set(2012, 0, 19, 10, 30);
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME, beginTime.getTimeInMillis());
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_END_TIME, endTime.getTimeInMillis());
calendarIntent.putExtra(Events.TITLE, "Ninja class");
calendarIntent.putExtra(Events.EVENT_LOCATION, "Secret dojo");
```

Note: 这个intent for Calendar的例子只使用于>=API Level 14。

Note: 请尽可能的定义你的intent更加确切。例如，如果你想要使用ACTION_VIEW的intent来显示一张图片，你还应该指定MIME type 为 image/*。这样能够阻止其他能够 "查看" 其他数据类型的app (like a map app) 被这个intent叫起。

Verify There is an App to Receive the Intent[验证是否有App去接收这个Intent]

尽管Android系统会确保每一个确定的intent会被系统内置的app(such as the Phone, Email, or Calendar app)之一接收，但是你还是应该在触发一个intent之前做验证是否有App接受这个intent的步骤。

Caution: 如果你触发了一个intent，而且没有任何一个app会去接收这个intent，那么你的app会crash。

为了验证是否有合适的activity会响应这个intent,需要执行 queryIntentActivities() 来获取到能够接收这个intent的所有activity的list。如果返回的List非空，那么你可以安全的使用这个intent。例如：

```
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(intent, 0);
boolean isIntentSafe = activities.size() > 0;
```

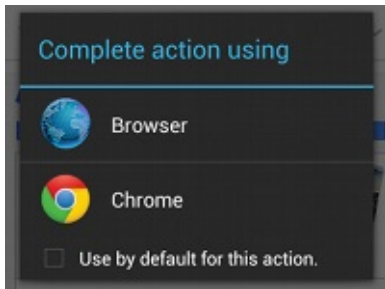
如果 isIntentSafe 是 true, 那么至少有一个app可以响应这个intent。如果是 false则说明没有app可以handle这个intent。

Note:你必须在第一次使用之前做这个检查，若是不可行，则应该关闭这个功能。如果你知道某个确切的app能够handle这个intent，你也应该提供给用户去下载这个app的链接。（[see how to link to your product on Google Play](#)）.

Start an Activity with the Intent [使用Intent来启动Activity]

当你创建好了intent并且设置好了extra数据，通过执行startActivity() 来发送到系统。如果系统确定有多个activity可以handle这个intent,它会显示出一个dialog，让用户选择启动哪个app。如果系统发现只有一个app可以handle这个intent，那么就会直接启动这个app。

```
startActivity(intent);
```



下面是一个完整的例子，演示了如何创建一个intent来查看地图，验证有app可以handle这个intent,然后启动它。

```
// Build the intent
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);

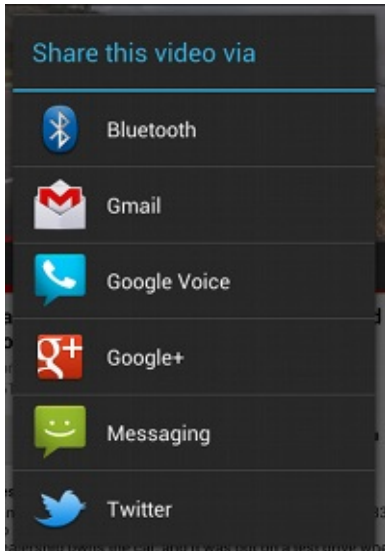
// Verify it resolves
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(mapIntent, 0);
boolean isIntentSafe = activities.size() > 0;

// Start an activity if it's safe
if (isIntentSafe) {
    startActivity(mapIntent);
}
```

Show an App Chooser[显示一个App选择界面]

请注意，当你发送一个intent，有多个app可以handle的情况，用户可以在弹出dialog的时候，选择默认启动的app（通过勾选dialog下面的选择框，如上图所示）。这个功能对于用户有特殊偏好的时候非常有用（例如用户总是喜欢启动某个app来查看网页，总是喜欢启动某个camera来拍照）。

然而，如果用户希望每次都弹出选择界面，而且每次都不确定会选择哪个app启动，例如分享功能，用户选择分享到哪个app都是不确定的，这个时候，需要强制弹出选择的对话框。（这种情况下用户不能选择默认启动的app）。



为了显示chooser, 需要使用createChooser()来创建Intent

```
Intent intent = new Intent(Intent.ACTION_SEND);
...

// Always use string resources for UI text. This says something like "Share this photo with"
String title = getResources().getText(R.string.chooser_title);
// Create and start the chooser
Intent chooser = Intent.createChooser(intent, title);
startActivity(chooser);
```

编写:[kesenhoo](#)

校对:

接收Activity返回的结果

启动另外一个activity并不一定是单向的。你也可以启动另外一个activity然后接受一个result回来。为了接受这个result,你需要使用[startActivityForResult\(\)](#) (而不是[startActivity\(\)](#))。

例如, 你的app可以启动一个camera程序并接受拍的照片作为result。或者你可以启动People程序并获取其中联系的人的详情作为result。

当然, 被启动的activity需要指定返回的result。它需要把这个result作为另外一个intent对象返回, 你的activity需要在[onActivityResult\(\)](#)的回调方法里面去接收result。

Note:在执行 `startActivityForResult()`时, 你可以使用explicit 或者 implicit 的intent。当你启动另外一个位于你的程序中的activity时, 你应该使用explicit intent来确保你可以接收到期待的结果。

Start the Activity(启动Activity)

对于startActivityResult() 方法中的intent与之前介绍的并没有什么差异，只不过是需要在这个方法里面多添加一个int类型的参数。这个integer的参数叫做"request code"，它标识了你的请求。当你接收到result Intent时，可以从回调方法里面的参数去判断这个result是否是你想要的。

例如，下面是一个启动activity来选择联系人的例子：

```
static final int PICK_CONTACT_REQUEST = 1; // The request code
...
private void pickContact() {
    Intent pickContactIntent = new Intent(Intent.ACTION_PICK, new Uri("content://contacts"))
    pickContactIntent.setType(Phone.CONTENT_TYPE); // Show user only contacts w/ phone number
    startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST);
}
```

Receive the Result(接收Result)

当用户完成了启动之后activity操作之后，系统会调用你的activity的onActivityResult() 回调方法。这个方法有三个参数：

- 你通过startActivityForResult()传递的request code。
- 第二个activity指定的result code。如果操作成功则是RESULT_OK，如果用户没有操作成功，而是直接点击回退或者其他什么原因，那么则是RESULT_CANCELED
- 第三个参数则是intent,它包含了返回的result数据部分。

例如，下面是如何处理pick a contact的result的例子：对应上面的例子

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // Check which request we're responding to
    if (requestCode == PICK_CONTACT_REQUEST) {
        // Make sure the request was successful
        if (resultCode == RESULT_OK) {
            // The user picked a contact.
            // The Intent's data Uri identifies which contact was selected.

            // Do something with the contact here (bigger example below)
        }
    }
}
```

为了正确的handle这些result，你必须了解那些result intent的格式。对于你自己程序里面的返回result是比较简单的。Apps都会有一些自己的api来指定特定的数据。例如，People app (Contacts app on some older versions) 总是返回一个URI来指定选择的contact，Camera app 则是在data数据区返回一个 Bitmap (see the class about [Capturing Photos](#)).

编写:[kesenhoo](#)

校对:

Intent过滤

前两节课主要讲了从你的app启动另外一个app。但是如果你的app可以响应前面发出的action，那么你的app应该做好响应的准备。例如，如果你创建了一个social app，它可以分享messages 或者 photos 给好友，那么最好你的app能够接收ACTION_SEND的intent,这样当用户在其他app触发分享功能的时候，你的app能够出现在待选对话框。

为了使得其他的app能够启动你的activity，你需要在你的manifest文件的 标签下添加 的属性。

当你的app被安装到设备上时，系统可以识别你的intent filter并把这些信息记录下来。当其他app通过执行 startActivity() 或者 startActivityForResult()方法，并使用implicit intent时，系统可以自动查找出那些可以响应这个intent的activity。

Add an Intent Filter(添加Intent Filter)

为了尽可能确切的定义你的activity能够handle哪些intent, 每一个intent filter都应该尽可能详尽的定义好action与data。

主要有下面三个方面需要定义：

- **Action:**一个想要执行的动作的名称。通常是系统已经定义好的值, 例如 ACTION_SEND 或者 ACTION_VIEW。
- **Data:**Intent附带数据的描述。可以使用一个或者多个属性, 你可以只定义MIME type或者是只指定URI prefix, 也可以只定义一个URI scheme, 或者是他们综合使用。**Note:** 如果你不想handle Uri 类型的数据, 那么你应该指定 android:mimeType 属性。例如 text/plain or image/jpeg.
- **Category:**提供一个附加的方法来标识这个activity能够handle的intent。通常与用户的手势或者是启动位置有关。系统有支持几种不同的categories,但是大多数都不怎么用的到。而且, 所有的implicit intents都默认是 CATEGORY_DEFAULT 类型的。

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
    <data android:mimeType="image/*"/>
  </intent-filter>
</activity>
```

每一个发送出来的intent只会包含一个action与type, 但是handle这个intent的activity的 <intent-filter>是可以声明多个<action>, <category>与<data> 的。

如果任何的两对action与data是互相矛盾的, 你应该创建不同的intent filter来指定特定的action与type。

例如, 假设你的activity可以handle 文本与图片, 无论是ACTION_SEND 还是 ACTION_SENDTO 的intent。在这种情况下, 你必须为两个action定义两个不同的intent filter。因为ACTION_SENDTO intent 必须使用 Uri 类型来指定接收者使用 send 或 sendto 的地址。例如：

```
<activity android:name="ShareActivity">
  <!-- filter for sending text; accepts SENDTO action with sms URI schemes -->
  <intent-filter>
    <action android:name="android.intent.action.SENDTO"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="sms" />
    <data android:scheme="smsto" />
  </intent-filter>
  <!-- filter for sending text or images; accepts SEND action and text or image data -->
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="image/*"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Note:为了接受implicit intents, 你必须在你的intent filter中包含 CATEGORY_DEFAULT 的category。关于更多sending 与 receiving ACTION_SEND intents来执行social sharing行为的, 请查看上一课：[Getting a Result from an Activity:接收Activity返回的结果](#)

Handle the Intent in Your Activity [在你的Activity中Handle发送过来的Intent]

为了决定采用哪个action，你可以读取Intent的内容。

你可以执行 `getIntent()` 来获取启动你的activity的那个intent。你可以在activity生命周期的任何时候去执行这个方法，当是你最好是在 `onCreate()` 或者 `onStart()` 里面去执行。

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

    setContentView(R.layout.main);

    // Get the intent that started this activity
    Intent intent = getIntent();
    Uri data = intent.getData();

    // Figure out what to do based on the intent type
    if (intent.getType().indexOf("image/") != -1) {
        // Handle intents with image data ...
    } else if (intent.getType().equals("text/plain")) {
        // Handle intents with text ...
    }
}
```

Return a Result(返回Result)

如果你想返回一个result给启动你的那个activity，仅仅需要执行 setResult()，通过指定一个result code与result intent。当你的操作成功之后，用户需要返回到原来的activity，通过执行finish()来关闭被叫起的activity。

```
// Create intent to deliver some kind of result data
Intent result = new Intent("com.example.RESULT_ACTION", Uri.parse("content://result_uri"));
setResult(Activity.RESULT_OK, result);
finish();
```

你必须总是指定一个result code。通常不是 RESULT_OK 就是 RESULT_CANCELED。你可以通过Intent来添加需要返回的数据。

Note:默认的result code是RESULT_CANCELED.因此，如果用户在没有完成操作之前点击了back key，那么之前的activity接受到的result code就是"canceled"。

如果你只是纯粹想要返回一个int来表示某些返回的result数据之一，你可以设置result code为任何大于0的数值。如果你返回的result只是一个int，那么连intent都可以不需要返回了，如下：

```
setResult(RESULT_COLOR_RED);
finish();
```

Note:我们没有必要在意你的activity是被用startActivity() 还是 startActivityForResult()方法所叫起的。系统会自动去判断该如何传递result。在不需要的result的case下，result会被自动忽略。

编写:

校对:

分享

编写:[kesenhoo](#)

校对:

分享简单的数据

Android程序中很炫的一个功能是程序之间可以互相通信。为什么要重新发明一个已经存在于另外一个程序中的功能呢，而且这个功能并非自己程序的核心部分。

这一章节会讲述一些通常使用的方法来在不同程序之间通过使用[Intent](#) APIs与[ActionProvider](#)对象来发送与接受content。

当你构建一个intent，你必须指定这个intent需要触发的actions。Android定义了一些actions，包括ACTION_SEND，这个action表明着这个intent是用来从一个activity发送数据到另外一个activity的，甚至是跨进程之间的。

为了发送数据到另外一个activity，你需要做的是指定数据与数据的类型，系统会识别出能够兼容接受的这些数据的activity并且把这些activity显示给用户进行选择(如果有多个选择)，或者是立即启动Activity(只有一个兼容的选择)。同样的，你可以在manifest文件的Activity描述中添加接受哪些数据类型。

在不同的程序之间使用intent来发送与接受数据是在社交分享内容的时候最常用的方法。Intent使得用户用最常用的程序进行快速简单的分享信息。

注意:为ActionBar添加分享功能的最好方法是使用[ShareActionProvider](#)，它能够在API level 14以上进行使用。ShareActionProvider会在第3课中进行详细介绍。

编写:[kesenhoo](#)

校对:

给其他App发送简单的数据

Send Text Content(分享文本内容)

ACTION_SEND的最直接与最常用的是从一个Activity发送文本内容到另外一个Activity。例如，Android内置的浏览器可以把当前显示页面的URL作为文本内容分享到其他程序。这是非常有用的，通过邮件或者社交网络来分享文章或者网址给好友。下面是一段Sample Code:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

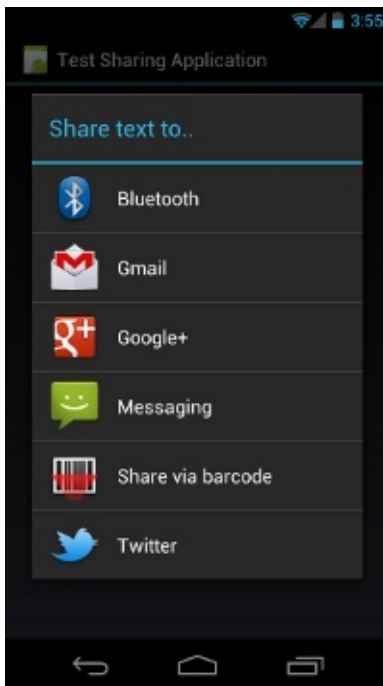
如果设备上有安装某个能够匹配ACTION_SEND与MIME类型为text/plain程序，那么Android系统会自动把他们都给筛选出来，并呈现Dialog给用户进行选择。如果你为intent调用了Intent.createChooser()，那么Android总是会显示可供选择。这样有一些好处：

- 即使用户之前为这个intent设置了默认的动作，选择界面还是会被显示。
- 如果没有匹配的程序，Android会显示系统信息。
- 你可以指定选择界面的标题。

下面是更新后的代码：

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent, getResources().getText(R.string.send_to)));
```

效果图如下：



Optionally,你可以为intent设置一些标准的附加值，例如：EXTRA_EMAIL, EXTRA_CC, EXTRA_BCC, EXTRA_SUBJECT.然而，如果接收程序没有针对那些做特殊的处理，则不会有对应的反应。你也可以使用自定义的附加值，但是除非接收的程序能够识别出来，不然没有任何效果。典型的做法是，你使用被接受程序定义的附加值。

注意:一些e-mail程序，例如Gmail,对应接收的是EXTRA_EMAIL与EXTRA_CC，他们都是String类型的，可以使用putExtra(string,string[])方法来添加到intent里面。

Send Binary Content(分享二进制内容)

分享二进制的数据需要结合设置特定的MIME Type，需要在EXTRA_STREAM里面放置数据的URI,下面有个分享图片的例子，这个例子也可以修改用来分享任何类型的二进制数据：

```
Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND);
shareIntent.putExtra(Intent.EXTRA_STREAM, uriToImage);
shareIntent.setType("image/jpeg");
startActivity(Intent.createChooser(shareIntent, getResources().getText(R.string.send_to)));
```

请注意下面的内容：

- 你可以使用`/*`这样的方式来制定MIME类型，但是这仅仅会match到那些能够处理一般数据类型的Activity(即一般的Activity无法详尽所有的MIME类型)
- 接收的程序需要有访问URI资源的权限。下面有一些方法来处理这个问题：
 - 把文件写到外部存储设备上，类似SDCard，这样所有的app都可以进行读取。使用`Uri.fromFile()`方法来创建可以用在分享时传递到intent里面的Uri。然而，请记住，不是所有的程序都遵循`file://`这样格式的Uri。
 - 在调用`getFileStreamPath()`返回一个File之后，使用带有`MODE_WORLD_READABLE`模式的`openFileOutput()`方法把数据写入到你自己的程序目录下。像上面一样，使用`Uri.fromFile()`创建一个`file://`格式的Uri用来添加到intent里面进行分享。
 - 媒体文件，例如图片，视频与音频，可以使用`scanFile()`方法进行扫描并存储到MediaStore里面。`onScanCompleted()`回调函数会返回一个`content://`格式的Uri，这样便于你进行分享的时候把这个uri放到intent里面。
 - 图片可以使用`insertImage()`方法直接插入到MediaStore系统里面。那个方法会返回一个`content://`格式的Uri。
 - 存储数据到你自己的ContentProvider里面，确保其他app可以有访问你的provider的权限。(或者使用 per-URI permissions)

Send Multiple Pieces of Content(发送多块内容)

为了同时分享多种不同类型的内容，需要使用ACTION_SEND_MULTIPLE与指定到那些数据的URIs列表。MIME类型会根据你分享的混合内容而不同。例如，如果你分享3张JPEG的图片，那么MIME类型仍然是image/jpeg。如果是不同图片格式的话，应该用image/*来匹配那些可以接收任何图片类型的activity。如果你需要分享多种不同类型的数据，可以使用*/*来表示MIME。像前面描述的那样，这取决于那些接收的程序解析并处理你的数据。下面是一个例子：

```
ArrayList<Uri> imageUris = new ArrayList<Uri>();
imageUris.add(imageUri1); // Add your image URIs here
imageUris.add(imageUri2);

Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUris);
shareIntent.setType("image/*");
startActivity(Intent.createChooser(shareIntent, "Share images to.."));
```

当然，请确保指定到数据的URIs能够被接收程序所访问(添加访问权限)。

编写:[kesenhoo](#)

校对:

接收从其他App返回的数据

就像你的程序能够发送数据到其他程序一样，其他程序也能够方便的接收发送过来的数据。需要考虑的是用户与你的程序如何进行交互，你想要从其他程序接收哪些数据类型。例如，一个社交网络程序会希望能够从其他程序接受文本数据，像一个有趣的网址链接。Google+的Android客户端会接受文本数据与单张或者多张图片。用这个app，用户可以简单的从Gallery程序选择一张图片来启动Google+进行发布。

Update Your Manifest[更新你的manifest文件]

Intent filters通知了Android系统说，一个程序会接受哪些数据。像上一课一样，你可以创建intent filters来表明程序能够接收哪些action。下面是个例子，对三个activity分别指定接受单张图片，文本与多张图片。(这里有不清楚Intent filter的，请参考[Intents and Intent Filters](#))

```
<activity android:name=".ui.MyActivity" >
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.SEND_MULTIPLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
  </intent-filter>
</activity>
```

当另外一个程序尝试分享一些东西的时候，你的程序会被呈现在一个列表里面让用户进行选择。如果用户选择了你的程序，相应的activity就应该被调用开启，这个时候就是你如何处理获取到的数据的问题了。

Handle the Incoming Content[处理接受到的数据]

为了处理从Intent带过来的数据，可以通过调用getIntent()方法来获取到Intent对象。一旦你拿到这个对象，你可以对里面的数据进行判断，从而决定下一步应该做什么。请记住，如果一个activity可以被其他的程序启动，你需要在检查intent的时候考虑这种情况(是被其他程序而调用启动的)。

```
void onCreate (Bundle savedInstanceState) {
    ...
    // Get intent, action and MIME type
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
        if ("text/plain".equals(type)) {
            handleSendText(intent); // Handle text being sent
        } else if (type.startsWith("image/")) {
            handleSendImage(intent); // Handle single image being sent
        }
    } else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type != null) {
        if (type.startsWith("image/")) {
            handleSendMultipleImages(intent); // Handle multiple images being sent
        }
    } else {
        // Handle other intents, such as being started from the home screen
    }
    ...
}

void handleSendText(Intent intent) {
    String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
    if (sharedText != null) {
        // Update UI to reflect text being shared
    }
}

void handleSendImage(Intent intent) {
    Uri imageUri = (Uri) intent.getParcelableExtra(Intent.EXTRA_STREAM);
    if (imageUri != null) {
        // Update UI to reflect image being shared
    }
}

void handleSendMultipleImages(Intent intent) {
    ArrayList<Uri> imageUris = intent.getParcelableArrayListExtra(Intent.EXTRA_STREAM);
    if (imageUris != null) {
        // Update UI to reflect multiple images being shared
    }
}
```

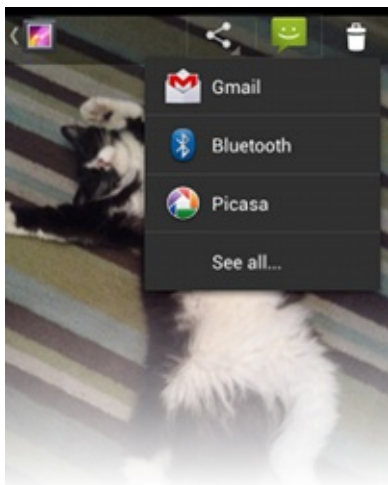
请注意，因为你无法知道其他程序发送过来的数据内容是文本还是其他的数据，因此你需要避免在UI线程里面去处理那些获取到的数据。更新UI可以像更新EditText一样简单，也可以是更加复杂一点的操作，例如过滤出感兴趣的图片。It's really specific to your application what happens next.

编写:[kesenhoo](#)

校对:

添加一个简便的分享动作

这一课会介绍在ActionBar 中添加一个高效率且比较友好的Share功能，会使用到ActionProvider(在Android 4.0上才被引进)。它会handle出现share功能的appearance与behavior。在ShareActionProvider的例子中，你只需要提供一个share intent，剩下的就交给[ShareActionProvider](#)来做。



Update Menu Declarations(更新菜单声明)

使用ShareActionProvider的第一步，在你的menu resources对应item中定义android:actionProviderClass属性。

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item_share"
        android:showAsAction="ifRoom"
        android:title="Share"
        android:actionProviderClass="android.widget.ShareActionProvider" />
  ...
</menu>
```

这表明了这个item的appearance与function需要与ShareActionProvider匹配。然而，你还是需要告诉provider你想分享的内容。

Set the Share Intent(设置分享的intent)

为了能够实现ShareActionProvider的功能，你必须提供给它一个intent。这个share intent应该像第一课讲的那样，带有ACTION_SEND和附加数据(例如EXTRA_TEXT与 EXTRA_STREAM)的。如何使用ShareActionProvider，请看下面的例子：

```
private ShareActionProvider mShareActionProvider;
...

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate menu resource file.
    getMenuInflater().inflate(R.menu.share_menu, menu);

    // Locate MenuItem with ShareActionProvider
    MenuItem item = menu.findItem(R.id.menu_item_share);

    // Fetch and store ShareActionProvider
    mShareActionProvider = (ShareActionProvider) item.getActionProvider();

    // Return true to display menu
    return true;
}

// Call to update the share intent
private void setShareIntent(Intent shareIntent) {
    if (mShareActionProvider != null) {
        mShareActionProvider.setShareIntent(shareIntent);
    }
}
```

你也许在创建菜单的时候仅仅需要设置一次share intent就满足需求了，或者说你可能想先设置share intent，然后根据UI的变化来对intent进行更新。例如，当你在Gallery里面全图查看照片的时候，share intent会在你切换图片的时候进行改变。想要查看更多关于ShareActionProvider的内容，请查看[Action Bar](#)。

编写:[jdneo](#)

校对:

分享文件

一个应用经常需要向其他应用发送一个甚至多个文件。例如，一个图库应用可能需要向图片编辑器提供多个文件，或者一个文件管理器可能希望允许用户在外存储的不同区域之间复制粘贴文件。一种让应用可以分享文件的方法，接收文件应用所发出的请求进行响应。

在所有情况下，唯一的一个将一个文件从你的应用发送至另一个应用的安全方法是向接收文件应用发送这个文件的URI，然后对这个URI授予临时的可访问权限。具有URI临时访问权限的URI是安全的，因为访问权限只授权于接收这个URI的应用，并且它们会自动过期。Android的[FileProvider](#)组件提供了[getUriForFile\(\)](#)方法来创建一个文件的URI。

如果你希望在应用之间共享少量的文本或者数字的数据，你应该发送一个包含该数据的Intent。要学习如何通过Intent发送简单数据，可以阅读：[Sharing Simple Data](#)。

这系列课程将会介绍如何使用Android的[FileProvider](#)组件创建的URI，以及你向接收URI的应用授予的临时访问权限，来安全地在应用之间共享文件。

编写:[jdneo](#)

校对:

建立文件分享

为了从你的应用安全地将一个文件发送给另一个应用，你需要配置你的应用来提供安全的文件句柄（URI的形式），Android 的[FileProvider](#)的默认的实现，以及如何指定你要共享的文件。

Note: [FileProvider](#)是[v4 Support Library](#)中的。关于如何在你的应用中包含此库，可以阅读：[Support Library Setup](#)。

指定FileProvider

为你的应用定义一个[FileProvider](#)，需要在你的清单文件中定义一个字段，这个字段指明了使用创建URI的权限。以及一个XML文件，它指定了你的应用可以共享的目录路径。

下面的例子展示的是，如何在清单文件中添加[<provider>](#)标签，来指定[FileProvider](#)类，权限和XML文件名：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.myapp">
<application
...>
  <provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.example.myapp.fileprovider"
    android:grantUriPermissions="true"
    android:exported="false">
    <meta-data
      android:name="android.support.FILE_PROVIDER_PATHS"
      android:resource="@xml/filepaths" />
    </provider>
  ...
</application>
</manifest>
```

这里，[android:authorities](#)属性字段指定了你希望使用由[FileProvider](#)生成的URI的URI authority。在这个例子中，这个authority是“com.example.myapp.fileprovider”。对于你自己的应用，定义authority时，是在你的应用包名（[android:package](#)的值）之后追加“fileprovider”。为了学习更多关于authority的知识，可以阅读：[Content URIs](#)，以及[android:authorities](#)。

[<provider>](#)下的子标签[<meta-data>](#)指定了一个XML文件，它指定了你希望共享的目录路径。“android:resource”属性字段是这个文件的路径和名字（无“.xml”后缀）。该文件的内容将在下一节讨论。

指定可共享目录路径

一旦你在你的清单文件中为你的应用添加了[FileProvider](#)，你需要指定你希望共享文件的目录路径。为了指定这个路径，我们首先在“res/xml/”下创建文件“filepaths.xml”。在这个文件中，为每一个目录添加一个XML标签。下面的例子展示的是一个“res/xml/filepaths.xml”的例子。这个例子也说明了如何在你的内部存储区域共享一个“files/”目录的子目录：

```
<paths>
  <files-path path="images/" name="myimages" />
</paths>
```

在这个例子中，<files-path>标签共享的是在你的应用的内部存储中“files/”目录下的目录。“path”属性字段指出了该子目录为“files/”目录下的子目录“images/”。“name”属性字段告知[FileProvider](#)向在“files/images/”子目录中的文件URI添加一个路径分段标记“myimages”。

<paths>标签可以有多个子标签，每一个子标签都指定一个不同的要共享的目录。除了<files-path>标签，你可以使用<external-path>来分享位于外部存储的文件，而<cache-path>标签用来共享在你的内部缓存目录下的目录。学习更多关于指定共享目录的子标签的知识，可以阅读：[FileProvider](#)。

Note：XML文件是你定义共享目录的唯一方式，你不可以以代码的形式添加目录。

现在你有一个完整的[FileProvider](#)说明，它为在你应用的内部存储中“files/”目录下创建文件的URI，或者是在“files/”中的子目录内的文件创建URI。当你的应用为一个文件创建了URI，它就包含了在<provider>标签中指定的Authority（“com.example.myapp.fileprovider”），路径“myimages/”，和文件的名字。

例如，如果你根据这节课的例子定义了一个[FileProvider](#)，然后你需要一个文件“default_image.jpg”的URI，[FileProvider](#)会返回如下URI：

```
content://com.example.myapp.fileprovider/myimages/default_image.jpg
```

编写:[jdneo](#)

校对:

分享文件

一旦你配置了你的应用来使用URI共享文件，你可以响应其他应用关于这些文件的需求。一种响应的方法是在服务应用端提供一个文件选择接口，它可以由其他应用激活。这种方法可以允许客户应用端让用户从服务应用端选择一个文件，然后接收这个文件的URI。

这节课将会向你展示如何在你的应用中创建一个用来选择文件的[Activity](#)，来响应这些索取文件的需求。

接收文件请求

为了从客户应用端接收一个文件索取请求，然后以URI形式进行响应，你的应用应该提供一个选择文件的[Activity](#)。客户应用端通过调用[startActivityForResult\(\)](#)来启动这个[Activity](#)。该方法包含了一个[Intent](#)，它具有[ACTION_PICK](#)的Action。当客户应用端调用了[startActivityForResult\(\)](#)，你的应用可以向客户应用端返回一个结果，该结果即用户所选文件对应的URI。

学习如何在客户应用端实现文件索取请求，可以阅读：[Requesting a Shared File](#)。

创建一个文件选择Activity

为了配置文件选择Activity，我们从在清单文件中定义你的Activity开始，在其intent过滤器中，匹配ACTION_PICK的Action，以及CATEGORY_DEFAULT和CATEGORY_OPENABLE的Category。另外，为你的应用向其他应用所提供的文件设置MIME类型过滤器。下面的这段代码展示了如何在清单文件中定义新的Activity和intent过滤器：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <application>
        ...
        <activity
            android:name=".FileSelectActivity"
            android:label="@string/File_Selector" >
            <intent-filter>
                <action
                    android:name="android.intent.action.PICK"/>
                <category
                    android:name="android.intent.category.DEFAULT"/>
                <category
                    android:name="android.intent.category.OPENABLE"/>
                <data android:mimeType="text/plain"/>
                <data android:mimeType="image/*"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

在代码中定义文件选择Activity

下面，定义一个Activity子类，它用来显示在你内部存储的“files/images/”目录下可以获得的文件，然后允许用户选择期望的文件。下面的代码显示了如何定义这个Activity。并且响应用户的选择：

```
public class MainActivity extends Activity {
    // The path to the root of this app's internal storage
    private File mPrivateRootDir;
    // The path to the "images" subdirectory
    private File mImagesDir;
    // Array of files in the images subdirectory
    File[] mImageFiles;
    // Array of filenames corresponding to mImageFiles
    String[] mImageFilenames;
    // Initialize the Activity
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Set up an Intent to send back to apps that request a file
        mResultIntent =
            new Intent("com.example.myapplication.ACTION_RETURN_FILE");
        // Get the files/ subdirectory of internal storage
        mPrivateRootDir = getFilesDir();
        // Get the files/images subdirectory;
        mImagesDir = new File(mPrivateRootDir, "images");
        // Get the files in the images subdirectory
        mImageFiles = mImagesDir.listFiles();
        // Set the Activity's result to null to begin with
        setResult(Activity.RESULT_CANCELED, null);
        /*
         * Display the file names in the ListView mFileListView.
         * Back the ListView with the array mImageFilenames, which
         * you can create by iterating through mImageFiles and
         * calling File.getAbsolutePath() for each File
         */
        ...
    }
    ...
}
```

响应一个文件选择

一旦一个用户选择了一个共享的文件，你的应用必须明确哪个文件被选择了，然后为这个文件生成一个对应的URI。若[Activity](#)在[ListView](#)中显示了可获得文件的清单，当用户点击了一个文件名时，系统调用了方法[onItemClick\(\)](#)，在该方法中你可以获取被选择的文件。

在[onItemClick\(\)](#)中，为选择的文件文件名获取一个[File](#)对象，然后将它作为参数传递给[getUriForFile\(\)](#)，另外还需传入的参数是你为[FileProvider](#)所指定的`<provider>`标签值。这个结果URI包含了相应的被访问权限，一个对应于文件目录的路径标记（如在XML meta-data中定义的），以及包含扩展名的文件名。有关[FileProvider](#)如何匹配基于XML meta-data的目录路径的信息，可以阅读：[Specify Sharable Directories](#)。

下面的例子展示了你如何检测选中的文件并且获得一个URI：

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Define a listener that responds to clicks on a file in the ListView
    mFileListView.setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override
            /*
             * When a filename in the ListView is clicked, get its
             * content URI and send it to the requesting app
             */
            public void onItemClick(AdapterView<?> adapterView,
                                    View view,
                                    int position,
                                    long rowId) {
                /*
                 * Get a File for the selected file name.
                 * Assume that the file names are in the
                 * mImageFilename array.
                 */
                File requestFile = new File(mImageFilename[position]);
                /*
                 * Most file-related method calls need to be in
                 * try-catch blocks.
                 */
                // Use the FileProvider to get a content URI
                try {
                    fileUri = FileProvider.getUriForFile(
                        MainActivity.this,
                        "com.example.myapp.fileprovider",
                        requestFile);
                } catch (IllegalArgumentException e) {
                    Log.e("File Selector",
                        "The selected file can't be shared: " +
                        clickedFilename);
                }
                ...
            }
        });
    ...
}
```

记住，你能生成的那些URI所对应的文件，是那些在meta-data文件中包含标签的（即你定义的）目录内的文件，这方面知识在[Specify Sharable Directories](#)中已经讨论过。如果你为一个在你没有指定的目录内的文件调用了[getUriForFile\(\)](#)方法，你会收到一个[IllegalArgumentException](#)。

为文件授权

现在你有了你想要共享给其他应用的文件URI，你需要允许客户端访问这个文件。为了允许访问，可以通过将URI添加至一个[Intent](#)，然后为该[Intent](#)设置权限标记。你所授予的权限是临时的，并且当接收应用的任务栈被完成后，会自动过期。

下面的例子展示了如何为文件设置读权限：

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Define a listener that responds to clicks in the ListView
    mFileListView.setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                                    View view,
                                    int position,
                                    long rowId) {
                ...
                if (fileUri != null) {
                    // Grant temporary read permission to the content URI
                    mResultIntent.addFlags(
                        Intent.FLAG_GRANT_READ_URI_PERMISSION);
                }
                ...
            }
        });
    ...
}
```

Caution：调用[setFlags\(\)](#)是唯一安全的方法，为你的文件授予临时的被访问权限。避免对文件URI调用[Context.grantUriPermission\(\)](#)，因为通过该方法授予的权限，你只能通过调用[Context.revokeUriPermission\(\)](#)来撤销。

与请求应用共享文件

为了与需求应用共享其需要的文件，将包含了URI和响应权限的[Intent](#)传递给[setResult\(\)](#)。当你定义的[Activity](#)被结束后，系统会把这个包含了URI的[Intent](#)传递给客户端应用。下面的例子展示了你应该如何做：

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Define a listener that responds to clicks on a file in the ListView
    mFileListView.setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                                    View view,
                                    int position,
                                    long rowId) {
                ...
                if (fileUri != null) {
                    ...
                    // Put the Uri and MIME type in the result Intent
                    mResultIntent.setDataAndType(
                        fileUri,
                        getContentResolver().getType(fileUri));
                    // Set the result
                    MainActivity.this.setResult(Activity.RESULT_OK,
                                                mResultIntent);
                } else {
                    mResultIntent.setDataAndType(null, "");
                    MainActivity.this.setResult(RESULT_CANCELED,
                                                mResultIntent);
                }
            }
        });
}
```

向用户提供一个一旦他们选择了文件就能立即回到客户应用的方法。一种实现的方法是提供一个勾选框或者一个完成按钮。使用按钮的[android:onClick](#)属性字段为它关联一个方法。在该方法中，调用[finish\(\)](#)。例如：

```
public void onDoneClick(View v) {
    // Associate a method with the Done button
    finish();
}
```

编写:[jdneo](#)

校对:

请求分享一个文件

当一个应用希望访问由其它应用所共享的文件时，请求应用（即客户端）经常会向其它应用（服务端）发送一个文件请求。在大多数情况下，这个请求会在服务端应用启动一个Activity显示可以共享的文件。当服务应用向客户应用返回了URI后，用户即选择了文件。

这节课向你展示一个客户应用如何向服务应用请求一个文件，接受服务应用发来的URI，然后使用这个URI打开这个文件。

发送一个文件请求

为了向服务应用发送文件请求，在客户应用，需要调用[startActivityForResult\(\)](#)，同时传递给这个方法一个[Intent](#)，它包含了客户应用能处理的某行为，比如[ACTION_PICK](#)；以及一个MIME类型。

例如，下面的代码展示了如何向服务端应用发送一个Intent，来启动在[Sharing a File](#)中提到的[Activity](#)：

```
public class MainActivity extends Activity {
    private Intent mRequestFileIntent;
    private ParcelFileDescriptor mInputPFD;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mRequestFileIntent = new Intent(Intent.ACTION_PICK);
        mRequestFileIntent.setType("image/jpeg");
        ...
    }
    ...
    protected void requestFile() {
        /**
         * When the user requests a file, send an Intent to the
         * server app.
         * files.
         */
        startActivityForResult(mRequestFileIntent, 0);
        ...
    }
    ...
}
```

访问请求的文件

当服务应用向客户应用发回包含URI的[Intent](#)时，这个[Intent](#)会传递给客户应用的覆写的[onActivityResult\(\)](#)方法当中。一旦客户应用有了文件的URI，它就可以通过获取其[FileDescriptor](#)来访问文件。

文件的安全问题在这一过程中不用过多担心，因为这个客户应用所受到的所有数据只有URI而已。由于URI不包含目录路径，客户应用无法查询出或者打开任何服务应用的其他文件。客户应用仅仅获取了这个文件的访问渠道和访问的权限。同时访问权限是临时的，一旦这个客户应用的任务栈被完成了，这个文件将只能被服务应用访问。

下面的例子展示了客户应用如何处理发自服务应用的[Intent](#)，以及如何客户应用使用URI获取[FileDescriptor](#)：

```
/*
 * When the Activity of the app that hosts files sets a result and calls
 * finish(), this method is invoked. The returned Intent contains the
 * content URI of a selected file. The result code indicates if the
 * selection worked or not.
 */
@Override
public void onActivityResult(int requestCode, int resultCode,
    Intent returnIntent) {
    // If the selection didn't work
    if (resultCode != RESULT_OK) {
        // Exit without doing anything else
        return;
    } else {
        // Get the file's content URI from the incoming Intent
        Uri returnUrl = returnIntent.getData();
        /*
         * Try to open the file for "read" access using the
         * returned URI. If the file isn't found, write to the
         * error log and return.
         */
        try {
            /*
             * Get the content resolver instance for this context, and use it
             * to get a ParcelFileDescriptor for the file.
             */
            mInputPFD = getContentResolver().openFileDescriptor(returnUrl, "r");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            Log.e("MainActivity", "File not found.");
            return;
        }
        // Get a regular file descriptor for the file
        FileDescriptor fd = mInputPFD.getFileDescriptor();
        ...
    }
}
```

方法[openFileDescriptor\(\)](#)返回一个文件的[ParcelFileDescriptor](#)。从这个对象中，客户应用可以获取[FileDescriptor](#)对象，然后用户就可以利用这个对象读取这个文件。

编写:[jdneo](#)

校对:

获取文件信息

当一个客户端应用尝试对一个有URI的文件进行操作时，应用可以向服务应用索取关于文件的信息，包括文件的数据类型和文件大小。数据类型可以帮助客户应用确定该文件自己能否处理，文件大小能帮助客户应用为文件设置合理的缓冲区。

这节课将展示如何通过查询服务应用的[FileProvider](#)来获取文件的MIME类型和尺寸。

获取文件的MIME类型

一个文件的数据类型能够告知客户应用应该如何处理这个文件的内容。为了得到URI所对应文件的数据类型，客户应用调用[ContentResolver.getType\(\)](#)。这个方法返回了文件的MIME类型。默认的，一个[FileProvider](#)通过文件的后缀名来确定其MIME类型。

```
...
    /*
     * Get the file's content URI from the incoming Intent, then
     * get the file's MIME type
     */
    Uri returnUri = returnIntent.getData();
    String mimeType = getContentResolver().getType(returnUri);
...

```


获取文件名和文件大小

[FileProvider](#)类有一个默认的[query\(\)](#)方法的实现，它返回一个[Cursor](#)，它包含了URI所关联的文件的名字和尺寸。默认的实现返回两列：

[DISPLAY_NAME](#)

是文件的文件名，一个[String](#)。这个值和[File.getName\(\)](#)所返回的值是一样的。

[SIZE](#)

文件的大小，字节为单位，一个“long”型。这个值和[File.length\(\)](#)所返回的值是一样的。

客户端应用可以通过将[query\(\)](#)的参数都设置为“null”，只保留URI这一参数，来同时获取文件的[名字](#)和[大小](#)。例如，下面的代码获取一个文件的[名字](#)和[大小](#)，然后在两个[TextView](#)中进行显示：

```
...
/*
 * Get the file's content URI from the incoming Intent,
 * then query the server app to get the file's display name
 * and size.
 */
Uri returnUrl = returnIntent.getData();
Cursor returnCursor =
    getContentResolver().query(returnUri, null, null, null, null);
/*
 * Get the column indexes of the data in the Cursor,
 * move to the first row in the Cursor, get the data,
 * and display it.
 */
int nameIndex = returnCursor.getColumnIndex(OpenableColumns.DISPLAY_NAME);
int sizeIndex = returnCursor.getColumnIndex(OpenableColumns.SIZE);
returnCursor.moveToFirst();
TextView nameView = (TextView) findViewById(R.id.filename_text);
TextView sizeView = (TextView) findViewById(R.id.filesize_text);
nameView.setText(returnCursor.getString(nameIndex));
sizeView.setText(Long.toString(returnCursor.getLong(sizeIndex)));
...
```

编写:[jdneo](#)

校对:

使用NFC分享文件

Android允许你通过Android Beam文件传输功能在设备之间传送大文件。这个功能键具有简单的API，同时，它允许用户仅需要点击设备就能启动文件传输的过程。Android Beam会自动地将文件从一台设备拷贝至另外一台，并且在完成时告知用户。

Android Beam文件传输API可以用来处理大量的数据，而在Android4.0（API Level 14）引入的Android BeamNDEF传输API则用来处理少量的数据，比如：URI等一些体积较小的数据。另外，Android Beam是在Android NFC框架中唯一允许你从NFC标签中读取NDEF消息的方法。想要学习更多有关Android Beam的知识，可以阅读：[Beaming NDEF Messages to Other Devices](#)。想要学习更多有关NFC框架的知识，可以阅读：[Near Field Communication](#)。

编写:[jdneo](#)

校对:

发送文件给其他设备

这节课将向你展示如何通过Android Beam文件传输向另一台设备发送大文件。要发送文件，首先需要申明使用NFC和外部存储的权限，你需要测试一下你的设备是否支持NFC，这样，你才能够向Android Beam文件传输提供文件的URI。

使用Android Beam文件传输功能有下列要求：

1. Android Beam文件传输功能只能在Android 4.1（API Level 16）及以上使用。
2. 你希望传送的文件必须放置于外部存储。学习更多关于外部存储的知识，可以阅读：[Using the External Storage](#)。
3. 每个你希望传送的文件必须是全局可读的。你可以通过[File.setReadable\(true,false\)](#)来为文件设置相应的读权限。
4. 你必须提供你要传输文件的URI。Android Beam文件传输无法处理由[FileProvider.getUriForFile](#)生成的URI。

在清单文件中声明权限和功能

首先，编辑你的清单文件来声明你的应用所需要声明的权限和功能。

声明权限

为了允许你的应用使用Android Beam文件传输使用NFC从外部存储发送文件，你必须在你的应用清单申明下面的权限：

[NFC](#)

允许你的应用通过NFC发送数据。为了声明该权限，添加下面的标签作为一个[manifest](#)标签的子标签：

```
<uses-permission android:name="android.permission.NFC" />
```

[READ_EXTERNAL_STORAGE](#)

允许你的应用读取外部存储。为了声明该权限，添加下面的标签作为一个[manifest](#)标签的子标签：

```
<uses-permission  
    android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Note：对于Android 4.2.2（API Level 17）及之前的系统版本，这个权限不是必需的。在后续的系统版本中，若应用需要读取外部存储，可能会需要申明该权限。为了保证将来程序稳定性，建议在该权限申明变成必需的之前，就在清单文件中申明好。

指定NFC功能

指定你的应用使用使用NFC，添加[uses-feature](#)标签作为一个[manifest](#)标签的子标签。设置android:required属性字段为true，这样可以使得你的应用只有在NFC可以使用时，才能运行。

下面的代码展示了如何指定[uses-feature](#)标签：

```
<uses-feature  
    android:name="android.hardware.nfc"  
    android:required="true" />
```

注意，如果你的应用将NFC作为可选的一个功能，但期望在NFC不可使用时程序还能继续执行，你就应该设置android:required属性字段为false，然后在代码中测试NFC的可用性。

指定Android Beam文件传输

由于Android Beam文件传输只能在Android 4.1（API Level 16）及以上的平台使用，如果你的应用将Android Beam文件传输作为一个不可缺少的核心模块，那么你必须指定[uses-sdk](#)标签为：[android:minSdkVersion="16"](#)。或者，你可以将[android:minSdkVersion](#)设置为其它值，然后在代码中测试平台版本，这将在下一节中展开。

测试设备是否支持Android Beam文件传输

为了在你的应用清单文件中，定义NFC是可选的，你应该使用下面的标签：

```
<uses-feature android:name="android.hardware.nfc" android:required="false" />
```

如果你设置了[android:required="false"](#)，你必须要在代码中测试NFC和Android Beam文件传输是否被支持。

为了再代码中测试Android Beam文件传输，我们先通过[PackageManager.hasSystemFeature\(\)](#)和参数[FEATURE_NFC](#)，来测试设备是否支持NFC。下一步，通过[SDK_INT](#)的值测试系统版本是否支持Android Beam文件传输。如果Android Beam文件传输是支持的，那么获得一个NFC控制器的实例，它能允许你与NFC硬件进行通信，例如：

```
public class MainActivity extends Activity {
    ...
    NfcAdapter mNfcAdapter;
    // Flag to indicate that Android Beam is available
    boolean mAndroidBeamAvailable = false;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // NFC isn't available on the device
        if (!PackageManager.hasSystemFeature(PackageManager.FEATURE_NFC)) {
            /*
             * Disable NFC features here.
             * For example, disable menu items or buttons that activate
             * NFC-related features
             */
            ...
            // Android Beam file transfer isn't supported
        } else if (Build.VERSION.SDK_INT <
            Build.VERSION_CODES.JELLY_BEAN_MR1) {
            // If Android Beam isn't available, don't continue.
            mAndroidBeamAvailable = false;
            /*
             * Disable Android Beam file transfer features here.
             */
            ...
            // Android Beam file transfer is available, continue
        } else {
            mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
            ...
        }
    }
    ...
}
```

创建一个提供文件的回调函数

一旦你确认了设备支持Android Beam文件传输，那么可以添加一个回调函数，当Android Beam文件传输监测到用户希望与另一个支持NFC的设备发送文件时，系统会调用它。在该回调函数中，返回一个[Uri](#)对象数组，Android Beam文件传输将URI对应的文件拷贝发送给要接收的设备。

要添加这个回调函数，我们需要实现[NfcAdapter.CreateBeamUriCallback](#)接口，和它的方法：[createBeamUri\(\)](#)，下面是一个例子：

```
public class MainActivity extends Activity {
    ...
    // List of URIs to provide to Android Beam
    private Uri[] mFileUri = new Uri[10];
    ...
    /**
     * Callback that Android Beam file transfer calls to get
     * files to share
     */
    private class FileUriCallback implements
        NfcAdapter.CreateBeamUriCallback {
        public FileUriCallback() {
        }
        /**
         * Create content URIs as needed to share with another device
         */
        @Override
        public Uri[] createBeamUri(NfcEvent event) {
            return mFileUri;
        }
    }
    ...
}
```

一旦你实现了这个接口，通过调用[setBeamPushUriCallback\(\)](#)将回调函数提供给Android Beam文件传输。下面是一个例子：

```
public class MainActivity extends Activity {
    ...
    // Instance that returns available files from this app
    private FileUriCallback mFileUriCallback;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Android Beam file transfer is available, continue
        ...
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        /*
         * Instantiate a new FileUriCallback to handle requests for
         * URIs
         */
        mFileUriCallback = new FileUriCallback();
        // Set the dynamic callback for URI requests.
        mNfcAdapter.setBeamPushUriCallback(mFileUriCallback, this);
        ...
    }
    ...
}
```

Note：你也可以将[Uri](#)对象数组通过你应用的[NfcAdapter](#)实例，直接提供给NFC框架。如果你能在NFC触碰事件发生之前，定义这些URI，那么你可以选择这个方法。要学习关于这个方法的知识，可以阅读：[NfcAdapter.setBeamPushUri\(\)](#)。

定义要发送的文件

为了将一个或多个文件发送给支持NFC的设备，需要为每一个文件获取一个文件URI（一个具有文件格式（file scheme）的URI），然后将它们添加至一个Uri对象数组中。要传输一个文件，你必须也有读文件的权限。例如，下面的例子展示的是你如何根据文件名获取它的文件URI，然后将URI添加至数组当中：

```
/*
 * Create a list of URIs, get a File,
 * and set its permissions
 */
private Uri[] mFileUris = new Uri[10];
String transferFile = "transferimage.jpg";
File extDir = getExternalFilesDir(null);
File requestFile = new File(extDir, transferFile);
requestFile.setReadable(true, false);
// Get a URI for the File and add it to the list of URIs
fileUri = Uri.fromFile(requestFile);
if (fileUri != null) {
    mFileUris[0] = fileUri;
} else {
    Log.e("My Activity", "No File URI available for file.");
}
```

编写:[jdneo](#)

校对:

接收其他设备的文件

Android Beam文件传输将文件拷贝至接收设备上的一个特殊目录。同时使用Android媒体扫描器（Android Media Scanner）扫描拷贝的文件，并在媒体库（[MediaStore](#) provider）中为媒体文件添加对应的条目录录。这节课将向你展示当文件拷贝完成时要如何响应，并且在接收设备上应该如何放置拷贝的文件。

响应请求并显示数据

当Android Beam文件传输将文件拷贝至接收设备后，它会发布一个通知，包含了一个[Intent](#)，它有一个[ACTION_VIEW](#)的Action，第一个传输文件的MIME类型，和一个指向第一个文件的URI。当用户点击了这个通知后，intent会被发送至系统。为了让你的应用能够响应这个intent，我们需要为响应的[Activity](#)所对应的[<activity>](#)标签添加[<intent-filter>](#)标签，在[<intent-filter>](#)标签中，添加下面的子标签：

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

用来匹配通知中的intent。

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

匹配隐式的[Intent](#)。

```
<data android:mimeType="mime-type" />
```

匹配一个MIME类型。要指定那些你的应用能够处理的类型。

例如，下面的例子展示了如何添加一个intent filter来激活你的activity：

```
<activity
    android:name="com.example.android.nfctransfer.ViewActivity"
    android:label="Android Beam Viewer" >
    ...
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        ...
    </intent-filter>
</activity>
```

Note：不仅仅只有Android Beam文件传输会发送含有[ACTION_VIEW](#)的intent。在接收设备上的其它应用也有可能发送含有该行为的intent。我们马上会进一步讨论这一问题。

请求文件读权限

如果要读取Android Beam文件传输所拷贝到设备上的文件，需要[READ_EXTERNAL_STORAGE](#)权限。例如：

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

如果你希望将文件拷贝至你自己应用的存储区，那么需要的权限改为[WRITE_EXTERNAL_STORAGE](#)，另外[WRITE_EXTERNAL_STORAGE](#)权限包含了[READ_EXTERNAL_STORAGE](#)权限。

Note：对于Android 4.2.2（API Level 17），[READ_EXTERNAL_STORAGE](#)权限仅在用户选择要读文件时才是强制需要的。而在今后的版本中会在所有情况下都需要该权限。为了保证应用在未来的兼容性，建议在清单文件中声明该权限。

由于你的应用对于其内部存储区域具有控制权，所以若要将文件拷贝至你应用的内部存储区域，写权限是不需要声明的。

获取拷贝文件的目录

Android Beam文件传输一次性将所有文件拷贝到目标设备的一个目录内，Android Beam文件传输通知所发出的[Intent](#)中包含有URI，他指向了第一个传输的文件。然而，你的应用也有可能接收到除了Android Beam文件传输之外的某个来源所发出的含有[ACTION_VIEW](#)行为的Intent。为了明确你应该如何处理接收的Intent，你需要检查它的scheme和authority。

为了获得URI的scheme，调用[Uri.getScheme\(\)](#)，下面的代码展示了如何明确架构并处理URI：

```
public class MainActivity extends Activity {
    ...
    // A File object containing the path to the transferred files
    private File mParentPath;
    // Incoming Intent
    private Intent mIntent;
    ...
    /*
     * Called from onNewIntent() for a SINGLE_TOP Activity
     * or onCreate() for a new Activity. For onNewIntent(),
     * remember to call setIntent() to store the most
     * current Intent
     */
    private void handleViewIntent() {
        ...
        // Get the Intent action
        mIntent = getIntent();
        String action = mIntent.getAction();
        /*
         * For ACTION_VIEW, the Activity is being asked to display data.
         * Get the URI.
         */
        if (TextUtils.equals(action, Intent.ACTION_VIEW)) {
            // Get the URI from the Intent
            Uri beamUri = mIntent.getData();
            /*
             * Test for the type of URI, by getting its scheme value
             */
            if (TextUtils.equals(beamUri.getScheme(), "file")) {
                mParentPath = handleFileUri(beamUri);
            } else if (TextUtils.equals(
                beamUri.getScheme(), "content")) {
                mParentPath = handleContentUri(beamUri);
            }
        }
        ...
    }
    ...
}
```

从文件URI中获取目录

如果接收的[Intent](#)包含一个文件URI，则该URI包含了一个文件的绝对文件名，包括了完整的路径和文件名。对于Android Beam文件传输来说，目录路径指向了其它传输文件的位置（如果有其它传输文件的话），要获得这个目录路径，取得URI的路径部分（URI中除去“file:”前缀的部分），根据路径创建一个[File](#)对象，然后获取这个[File](#)的父目录：

```
...
public String handleFileUri(Uri beamUri) {
    // Get the path part of the URI
    String fileName = beamUri.getPath();
    // Create a File object for this filename
    File copiedFile = new File(fileName);
    // Get a string containing the file's parent directory
    return copiedFile.getParent();
}
...
```

从内容URI获取目录

如果接收的[Intent](#)包含一个内容URI，这个URI可能指向的是一个存储于[MediaStore](#)Content Provider的目录和文件名。你可以通过检测URI的authority值来判断是否是[MediaStore](#)的内容URI。一个[MediaStore](#)的内容URI可能来自Android Beam文件传输也可能来自其它应用，但不管怎么样，你都能根据该内容URI获得一个目录和文件名。

你也可以接收一个[ACTION_VIEW](#)的Intent，它包含有一个content provider的URI，而不是[MediaStore](#)，在这种情况下，这个内容URI不包含[MediaStore](#)的authority，且这个URI一般不指向一个目录。

Note：对于Android Beam文件传输，如果第一个接收的文件，其MIME类型为“audio/”，“image/”或者“video/*”，那么你会接收这个在[ACTION_VIEW](#)的Intent中的内容URI。Android Beam文件传输会在它存储传输文件的目录内运行Media Scanner，以此为媒体文件添加索引。同时Media Scanner将结果写入[MediaStore](#)的content provider，之后它将第一个文件的内容URI回递给Android Beam文件传输。这个内容URI就是你在通知[Intent](#)中所接收到的。要获得第一个文件的目录，你需要使用该内容URI从[MediaStore](#)中获取它。

指明Content Provider

为了明确你能从内容URI中获取文件目录，你可以通过调用[Uri.getAuthority\(\)](#)获取URI的Authority，以此确定与该URI相关联的Content Provider。其结果有两个可能的值：

[MediaStore.AUTHORITY](#)

表明这个URI关联了被[MediaStore](#)追踪的一个文件或者多个文件。可以从[MediaStore](#)中获取文件的全名，目录名就自然可以从文件全名中获取。

其他值

来自其他Content Provider的内容URI。可以显示与该内容URI相关联的数据，但是不要尝试去获取文件目录。

为了[MediaStore](#)的内容URI中获取目录，执行一个查询操作，它将[Uri](#)参数指定为收到的内容URI，列名为[MediaColumns.DATA](#)。返回的[Cursor](#)包含了完整路径和URI所代表的文件名。该目录路径下还包含了由Android Beam文件传输传送到该设备上的其它文件。

下面的代码展示了你要如何测试内容URI的Authority，并获取传输文件的路径和文件名：

```
...
    public String handleContentUri(Uri beamUri) {
        // Position of the filename in the query Cursor
        int filenameIndex;
        // File object for the filename
        File copiedFile;
        // The filename stored in MediaStore
        String fileName;
        // Test the authority of the URI
        if (!TextUtils.equals(beamUri.getAuthority(), MediaStore.AUTHORITY)) {
            /*
             * Handle content URIs for other content providers
             */
            // For a MediaStore content URI
        } else {
            // Get the column that contains the file name
            String[] projection = { MediaStore.MediaColumns.DATA };
            Cursor pathCursor =
                getContentResolver().query(beamUri, projection,
                    null, null, null);
            // Check for a valid cursor
            if (pathCursor != null &&
                pathCursor.moveToFirst()) {
                // Get the column index in the Cursor
                filenameIndex = pathCursor.getColumnIndex(
                    MediaStore.MediaColumns.DATA);
                // Get the full file name including path
                fileName = pathCursor.getString(filenameIndex);
                // Create a File object for the filename
                copiedFile = new File(fileName);
                // Return the parent directory of the file
                return new File(copiedFile.getParent());
            } else {
                // The query didn't work; return null
                return null;
            }
        }
    }
}
...
```

要学习更多关于从Content Provider获取数据的知识，可以阅读：[Retrieving Data from the Provider](#)。

编写:

校对:

多媒体

编写:

校对:

管理音频播放

编写:[kesenhoo](#)

校对:

控制你的应用的音量与播放

如果你的App在播放音频，显然用户能够以预期的方式来控制音频是很重要的。为了保证好的用户体验，同样App能够获取音频焦点是很重要的，这样才能确保不会在同一时刻出现多个App的声音。在学习这个课程后，你将能够创建对硬件音量按钮进行响应的App，当按下音量按钮的时候需要获取到当前音频的焦点，然后以适当的方式改变音量从而进行响应用户的行为。

一个好的用户体验是可预期可控的。如果App是在播放音频，那么显然我们需要做到能够通过硬件按钮，软件按钮，蓝牙耳机等来控制音量。同样的，我们需要能够进行play, stop, pause, skip, and previous等动作，并且进行对应的响应。

鉴别使用的是哪个音频流(Identify Which Audio Stream to Use)

首先需要知道的是我们的App会使用到哪些音频流。Android为播放音乐，闹铃，通知铃，来电声音，系统声音，打电话声音与DTMF频道分别维护了一个隔离的音频流。这是我们能够控制不同音频的前提。其中大多数都是被系统限制的，不能胡乱使用。除了你的App是需要做替换闹钟的铃声的操作，那么几乎其他的播放音频操作都是使用"STREAM_MUSIC"音频流。

Use Hardware Volume Keys to Control Your App's Audio Volume(使用硬件音量键来控制App的音量)

默认情况下，按下音量控制键会调节当前被激活的音频流，如果你的App没有在播放任何声音，则会调节响铃的声音。如果是一个游戏或者音乐程序，需要在不管是否目前正在播放歌曲或者游戏目前是否发出声音的时候，按硬件的音量键都会有对应的音量调节。我们需要监听音量键是否被按下，Android提供了setVolumeControlStream()的方法来直接控制指定的音频流。在鉴别出App会使用哪个音频流之后，需要在Activity或者Fragment创建的时候就设置音量控制，这样能确保不管App是否可见，音频控制功能都正常工作。

```
setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

使用硬件的播放控制按键来控制App的音频播放(Use Hardware Playback Control Keys to Control Your App's Audio Playback)

媒体播放按钮，例如play, pause, stop, skip, and previous的功能同样可以在一些线控，耳麦或者其他无线控制设备上实现。无论用户按下上面任何设备上的控制按钮，系统都会广播一个带有ACTION_MEDIA_BUTTON的Intent。为了响应那些操作，需要像下面一样注册一个BroadcastReceiver在Manifest文件中。

```
<receiver android:name=".RemoteControlReceiver">
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_BUTTON" />
    </intent-filter>
</receiver>
```

Receiver需要判断这个广播是来自哪个按钮的操作，Intent在EXTRA_KEY_EVENT中包含了KEY的信息，同样KeyEvent类包含了一列KEYCODE_MEDIA_的静态变量来表示不同的媒体按钮，例如KEYCODE_MEDIA_PLAY_PAUSE 与 KEYCODE_MEDIA_NEXT。

```
public class RemoteControlReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_MEDIA_BUTTON.equals(intent.getAction())) {
            KeyEvent event = (KeyEvent)intent.getParcelableExtra(Intent.EXTRA_KEY_EVENT);
            if (KeyEvent.KEYCODE_MEDIA_PLAY == event.getKeyCode()) {
                // Handle key press.
            }
        }
    }
}
```

因为可能有多个程序都同样监听了哪些控制按钮，那么必须在代码中特意控制当前哪个Receiver会进行响应。下面的例子显示了如何使用AudioManager来注册监听与取消监听，当Receiver被注册上时，它将是唯一响应Broadcast的Receiver。

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Start listening for button presses
am.registerMediaButtonEventReceiver(RemoteControlReceiver);
...

// Stop listening for button presses
am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
```

通常，App需要在Receiver没有激活或者不可见的时候（比如在onStop的方法里面）取消注册监听。但是在媒体播放的时候并没有那么简单，实际上，我们需要在后台播放歌曲的时候同样需要进行响应。一个比较好的注册与取消监听的方法是当程序获取与失去音频焦点的时候进行操作。这个内容会在后面的课程中详细讲解。

编写:[kesenhoo](#)

校对:

管理音频焦点

很多App都可以播放音频，因此在播放前如何获取到音频焦点就显得很重要了，这样可以避免同时出现多个声音，Android使用audio focus来节制音频的播放，仅仅是获取到audio focus的App才能够播放音频。

在App开始播放音频之前，它需要经过发出请求[request]->接受请求[receive]->音频焦点锁定[Audio Focus]的过程。同样，它需要知道如何监听失去音频焦点[lose of audio focus]的事件并进行合适的响应。

Request the Audio Focus(请求获取音频焦点)

通过call [requestAudioFocus\(\)](#) 方法来获取你想要获取到的音频流焦点。如果请求成功这个方法会返回AUDIOFOCUS_REQUEST_GRANTED。

我们必须指定正在使用哪个音频流，而且需要确定请求的是短暂的还是永久的audio focus。

- 短暂的焦点锁定：当期待播放一个短暂的音频的时候（比如播放导航指示）
- 永久的焦点锁定：当计划播放可预期到的较长的音频的时候（比如播放音乐）

下面是一个在播放音乐的时候请求永久的音频焦点的例子，我们必须在开始播放之前立即请求音频焦点，比如在用户点击播放或者游戏程序中下一关开始的片头音乐。

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);
...

// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                                // Use the music stream.
                                AudioManager.STREAM_MUSIC,
                                // Request permanent focus.
                                AudioManager.AUDIOFOCUS_GAIN);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
    // Start playback.
}
```

一旦结束了播放，需要确保call [abandonAudioFocus\(\)](#)方法。这样会通知系统说你不再需要获取焦点并且取消注册[AudioManager.OnAudioFocusChangeListener](#)的监听。在这样释放短暂音频焦点的case下，可以允许任何打断的App继续播放。

```
// Abandon audio focus when playback complete
am.abandonAudioFocus(afChangeListener);
```

当请求短暂音频焦点的时候，我们可以选择是否开启“ducking”。Ducking是一个特殊的机制使得允许音频间歇性的短暂播放。通常情况下，一个好的App在失去音频焦点的时候它会立即保持安静。如果我们选择在请求短暂音频焦点的时候开启了ducking，那意味着其它App可以继续播放，仅仅是在这一刻降低自己的音量，在短暂重新获取到音频焦点后恢复正常音量(也就是说：不用理会这个请求短暂焦点的请求，这并不会导致目前在播放的音频受到牵制，比如在播放音乐的时候突然出现一个短暂的短信提示声音，这个时候仅仅是把播放歌曲的音量暂时调低，好让短信声能够让用户听到，之后立马恢复正常播放)。

```
// Request audio focus for playback
int result = am.requestAudioFocus(afChangeListener,
                                // Use the music stream.
                                AudioManager.STREAM_MUSIC,
                                // Request permanent focus.
                                AudioManager.AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK);

if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    // Start playback.
}
```

处理失去音频焦点Handle the Loss of Audio Focus

如果A程序可以请求获取音频焦点，那么在B程序请求获取的时候，A获取到的焦点就会失去。显然我们需要处理失去焦点的事件。

在音频焦点的监听器里面，当接受到描述焦点改变的事件时会触发[onAudioFocusChange\(\)](#)回调方法。对应于获取焦点的三种类型，我们同样会有三种失去焦点的类型。

失去短暂焦点：通常在失去这种焦点的情况下，我们会暂停当前音频的播放或者降低音量，同时需要准备恢复播放重新获取到焦点之后。

失去永久焦点：假设另外一个程序开始播放音乐等，那么我们的程序就应该有效的结束自己。实用的做法是停止播放，移除button监听，允许新的音频播放器独占监听那些按钮事件，并且放弃自己的音频焦点。

在重新播放器自己的音频之前，我们需要确保用户重新点击自己App的播放按钮等。

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AudioManager.AUDIOFOCUS_LOSS_TRANSIENT
            // Pause playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Resume playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_LOSS) {
            am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
            am.abandonAudioFocus(afChangeListener);
            // Stop playback
        }
    }
};
```

在上面失去短暂焦点的例子中，如果允许ducking，那么我们可以选择“duck”的行为而不是暂停当前的播放。

Duck! [闪避]

Ducking是一个特殊的机制使得允许音频间歇性的短暂播放。在Ducking的情况下，正常播放的歌曲会降低音量来凸显这个短暂的音频声音，这样既让这个短暂的声音比较突出，又不至于打断正常的声音。

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {  
    public void onAudioFocusChange(int focusChange) {  
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK  
            // Lower the volume  
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {  
            // Raise it back to normal  
        }  
    }  
};
```

监听失去音频焦点是最重要的广播之一，但不是唯一的方法。系统广播了一系列的intent来警示你去改变用户的音频使用体验。下节课会演示如何监视那些广播来提升用户的整体体验。

编写:[kesenhoo](#)

校对:

兼容音频输出设备

用户在播放音乐的时候有多个选择，可以使用内置的扬声器，有线耳机或者是支持A2DP的蓝牙耳机。【补充：A2DP全名是Advanced Audio Distribution Profile 蓝牙音频传输模型协定! A2DP是能够采用耳机内的芯片来堆栈数据，达到声音的高清晰度。有A2DP的耳机就是蓝牙立体声耳机。声音能达到44.1kHz，一般的耳机只能达到8kHz。如果手机支持蓝牙，只要装载A2DP协议，就能使用A2DP耳机了。还有消费者看到技术参数提到蓝牙V1.0 V1.1 V1.2 V2.0——这些是指蓝牙的技术版本，是指通过蓝牙传输的速度，他们是否支持A2DP具体要看蓝牙产品制造商是否使用这个技术。来自[百度百科](#)】

Check What Hardware is Being Used(检测目前正在使用的硬件设备)

选择的播放设备会影响App的行为。可以使用AudioManager来查询某个音频输出到扬声器，有线耳机还是蓝牙上。

```
if (isBluetoothA2dpOn()) {  
    // Adjust output for Bluetooth.  
} else if (isSpeakerphoneOn()) {  
    // Adjust output for Speakerphone.  
} else if (isWiredHeadsetOn()) {  
    // Adjust output for headsets  
} else {  
    // If audio plays and noone can hear it, is it still playing?  
}
```


Handle Changes in the Audio Output Hardware(处理音频输出设备的改变)

当有线耳机被拔出或者蓝牙设备断开连接的时候，音频流会自动输出到内置的扬声器上。假设之前播放声音很大，这个时候突然转到扬声器播放会显得非常嘈杂。

幸运的是，系统会在那种事件发生时广播带有ACTION_AUDIO_BECOMING_NOISY的intent。无论何时播放音频去注册一个BroadcastReceiver来监听这个intent会是比较好的做法。

在音乐播放器下，用户通常希望发生那样事情的时候能够暂停当前歌曲的播放。在游戏里，通常会选择减低音量。

```
private class NoisyAudioStreamReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (AudioManager.ACTION_AUDIO_BECOMING_NOISY.equals(intent.getAction())) {
            // Pause the playback
        }
    }
}

private IntentFilter intentFilter = new IntentFilter(AudioManager.ACTION_AUDIO_BECOMING_NOISY);

private void startPlayback() {
    registerReceiver(myNoisyAudioStreamReceiver(), intentFilter);
}

private void stopPlayback() {
    unregisterReceiver(myNoisyAudioStreamReceiver());
}
```

编写:[kesenhoo](#)

校对:

拍照

在多媒体流行之前，世界是沉闷(dismal)并且特色稀少(featureless)的。还记得Gopher? (*Gopher*是计算机上的一个工具软件，是Internet提供的一种由菜单式驱动的信息查询工具，采用客户机/服务器模式)。因为你的app将要成为你的生活的一部分，请赋予你的app能够把用户生活装进去的功能。使用内置的Camera，你的程序可以使得用户扩展（augment）他们所看的事物，生成唯一的头像，查找角落的人偶（zombies），或者仅仅是分享他们的经验。

这一章节，会教你如何简单的使用已经存在的Camera程序。在后面的课程中，你会更加深入的（dive deeper）学习如何直接控制Camera的硬件。

试试下面的例子程序 [PhotoIntentActivity.zip](#)

编写:[kesenhoo](#)

校对:

简单的拍照

假设你想通过你的客户端程序实现一个聚合全球天气的地图，上面会有各地的当前天气图片。那么集合图片只是你程序的一部分。你想要最简单的动作来获取图片，而不是重新发明（reinvent）一个camera。幸运的是，大多数Android设备都已经至少安装了一款相机程序。在这节课中，你会学习，如何拍照

Request Camera Permission(请求使用相机权限)

在写程序之前，需要在你的程序的manifest文件中添加下面的权限：

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera" />
    ...
</manifest ... >
```

如果你的程序并不需要一定有Camera，可以添加`android:required="false"`的tag属性。这样的话，Google Play 也会允许没有camera的设备下载这个程序。当然你有必要在使用Camera之前通过`hasSystemFeature(PackageManager.FEATURE_CAMERA)`方法来检查设备上是否有Camera。如果没有，你应该关闭你的Camera相关的功能！

Take a Photo with the Camera App(使用相机应用程序进行拍照)

Android中的方法是：启动一个Intent来完成你想要的动作。这个步骤包含三部分：Intent 本身，启动的外部 Activity, 与一些处理返回照片的代码。如：

```
private void dispatchTakePictureIntent(int actionCode) {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(takePictureIntent, actionCode);
}
```

当然在发出Intent之前，你需要检查是否有app会来handle这个intent，否则会引起启动失败：

```
public static boolean isIntentAvailable(Context context, String action) {
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
        packageManager.queryIntentActivities(intent, PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}
```

View the Photo(查看照片)

Android的Camera程序会把拍好的照片编码为bitmap，使用extra value的方式添加到返回的 Intent 当中， 对应的key为data。

```
private void handleSmallCameraPhoto(Intent intent) {  
    Bundle extras = intent.getExtras();  
    mImageBitmap = (Bitmap) extras.get("data");  
    mImageView.setImageBitmap(mImageBitmap);  
}
```

Note: 这仅仅是处理一张很少的缩略图而已，如果是大的全图，需要做更多的事情来避免ANR。

Save the Photo(保存照片)

如果你提供一个file对象给Android的Camera程序，它会保存这张全图到给定的路径下。你必须提供存储的卷名，文件夹名与文件名。对于2.2以上的系统，如下操作即可：

```
storageDir = new File(  
    Environment.getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES  
    ),  
    getAlbumName()  
);
```

Set the file name(设置文件名)

正如上面描述的那样，文件的路径会有设备的系统环境决定。你自己需要做的只是定义个不会引起文件名冲突的命名 scheme。下面会演示一种解决方案：

```
private File createImageFile() throws IOException {
    // Create an image file name
    String timeStamp =
        new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    String imageFileName = JPEG_FILE_PREFIX + timeStamp + "_";
    File image = File.createTempFile(
        imageFileName,
        JPEG_FILE_SUFFIX,
        getAlbumDir()
    );
    mCurrentPhotoPath = image.getAbsolutePath();
    return image;
}
```

Append the file name onto the Intent(把文件名添加到网络上)

Once you have a place to save your image, pass that location to the camera application via the Intent.

```
File f = createImageFile();  
takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(f));
```

Add the Photo to a Gallery(添加照片到相册)

对于大多数人来说，最简单查看你的照片的方式是通过系统的Media Provider。下面会演示如何触发系统的Media Scanner来添加你的照片到Media Provider的DB中，这样使得相册程序与其他程序能够读取到那些图片。

```
private void galleryAddPic() {  
    Intent mediaScanIntent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);  
    File f = new File(mCurrentPhotoPath);  
    Uri contentUri = Uri.fromFile(f);  
    mediaScanIntent.setData(contentUri);  
    this.sendBroadcast(mediaScanIntent);  
}
```

Decode a Scaled Image(解码缩放图片)

在有限的内存下，管理全尺寸的图片会很麻烦。下面会介绍如何缩放图片来适应程序的显示：

```
private void setPic() {
    // Get the dimensions of the View
    int targetW = mImageView.getWidth();
    int targetH = mImageView.getHeight();

    // Get the dimensions of the bitmap
    BitmapFactory.Options bmOptions = new BitmapFactory.Options();
    bmOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    int photoW = bmOptions.outWidth;
    int photoH = bmOptions.outHeight;

    // Determine how much to scale down the image
    int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

    // Decode the image file into a Bitmap sized to fill the View
    bmOptions.inJustDecodeBounds = false;
    bmOptions.inSampleSize = scaleFactor;
    bmOptions.inPurgeable = true;

    Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    mImageView.setImageBitmap(bitmap);
}
```

编写:[kesenhoo](#)

校对:

简单的录像

这节课会介绍如何使用现有的Camera程序来录制一个视频。和拍照一样，我们没有必要去重新发明录像程序。大多数的Android程序都有自带Camera来进行录像。(这一课的内容大多数与前面一课类似，简要带过，一些细节不赘述了)

Request Camera Permission [请求权限]

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera" />
    ...
</manifest ... >
```

与上一课的拍照一样，你可以在启动Camera之前，使用`hasSystemFeature(PackageManager.FEATURE_CAMERA)`来检查是否存在Camera。

Record a Video with a Camera App(使用相机程序来录制视频)

```
private void dispatchTakeVideoIntent() {
    Intent takeVideoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    startActivityForResult(takeVideoIntent, ACTION_TAKE_VIDEO);
}
public static boolean isIntentAvailable(Context context, String action) {
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
        packageManager.queryIntentActivities(intent,
            PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}
```

View the Video(查看视频)

Android的Camera程序会把拍好的视频地址返回。下面的代码演示了，如何查询到这个视频并显示到VideoView。

```
private void handleCameraVideo(Intent intent) {  
    mVideoUri = intent.getData();  
    mVideoView.setVideoURI(mVideoUri);  
}
```

编写:[kesenhoo](#)

校对:

控制相机硬件

在这一节课，我们会讨论如何通过使用framework的APIs来直接控制相机的硬件。直接控制设备的相机，相比起拍照与录像来说，要复杂一些。然而，如果你想要创建一个专业的特殊的相机程序，这节课会演示这部分内容。

Open the Camera Object(打开相机对象)

获取到 Camera 对象是直接控制Camera的第一步。正如Android自带的相机程序一样，推荐访问Camera的方式是在onCreate方法里面另起一个Thread来打开Camera。这个方法可以避免因为打开工作比较费时而引起ANR。在一个更加基础的实现方法里面，打开Camera的动作被延迟到onResume()方法里面去执行，这样使得代码能够更好的重用，并且保持控制流程不会复杂化。(原文是：In a more basic implementation, opening the camera can be deferred to the onResume() method to facilitate code reuse and keep the flow of control simple.)

在camera正在被另外一个程序使用的时候去执行 Camera.open() 会抛出一个exception，所以需要捕获起来。

```
private boolean safeCameraOpen(int id) {
    boolean qOpened = false;

    try {
        releaseCameraAndPreview();
        mCamera = Camera.open(id);
        qOpened = (mCamera != null);
    } catch (Exception e) {
        Log.e(getString(R.string.app_name), "failed to open Camera");
        e.printStackTrace();
    }

    return qOpened;
}

private void releaseCameraAndPreview() {
    mPreview.setCamera(null);
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

自从API level 9开始，camera的framework可以支持多个cameras。如果你使用 open()，你会获取到最后的一个camera。

Create the Camera Preview(创建相机预览界面)

拍照通常需要提供的一个预览界面来显示待拍的事物。和拍照类似，你需要使用一个 SurfaceView 来展现录制的画面。

Preview Class

为了显示一个预览界面，你需要一个Preview类。这个类需要实现android.view.SurfaceHolder.Callback 接口，这个接口用来传递从camera硬件获取的数据到程序。

```
class Preview extends ViewGroup implements SurfaceHolder.Callback {

    SurfaceView mSurfaceView;
    SurfaceHolder mHolder;

    Preview(Context context) {
        super(context);

        mSurfaceView = new SurfaceView(context);
        addView(mSurfaceView);

        // Install a SurfaceHolder.Callback so we get notified when the
        // underlying surface is created and destroyed.
        mHolder = mSurfaceView.getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
    ...
}
```

这个Preview类必须在查看图片之前传递给 Camera 对象。正如下面描述的：

Set and Start the Preview

一个Camera实例与它相关的preview必须以一种指定的顺序来创建，首先是创建Camera对象。在下面的示例中，初始化camera的动作被封装起来，这样，无论用户想对Camera做任何的改变，都通过执行setCamera()来呼叫[Camera.startPreview\(\)](#)。Preview对象必须在 surfaceChanged() 的回调方法里面去做重新创建的动作。

```
public void setCamera(Camera camera) {
    if (mCamera == camera) { return; }

    stopPreviewAndFreeCamera();

    mCamera = camera;

    if (mCamera != null) {
        List<Size> localSizes = mCamera.getParameters().getSupportedPreviewSizes();
        mSupportedPreviewSizes = localSizes;
        requestLayout();

        try {
            mCamera.setPreviewDisplay(mHolder);
        } catch (IOException e) {
            e.printStackTrace();
        }

        /*
         * Important: Call startPreview() to start updating the preview surface. Preview must
         * be started before you can take a picture.
         */
        mCamera.startPreview();
    }
}
```

Modify Camera Settings(修改相机设置)

相机设置可以改变拍照的方式，从缩放级别到曝光补偿(exposure compensation)。下面的例子仅仅演示了改变预览大小的设置，更多设置请参考Camera的源代码。

```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    // Now that the size is known, set up the camera parameters and begin
    // the preview.
    Camera.Parameters parameters = mCamera.getParameters();
    parameters.setPreviewSize(mPreviewSize.width, mPreviewSize.height);
    requestLayout();
    mCamera.setParameters(parameters);

    /*
     * Important: Call startPreview() to start updating the preview surface. Preview must be
     * started before you can take a picture.
     */
    mCamera.startPreview();
}
```

Set the Preview Orientation(设置预览方向)

大多数相机程序会锁定预览为横屏的，因为那是人拍照的自然方式。设置里面并没有阻止你去拍竖屏的照片，这些信息会被记录在EXIF里面。[setCameraDisplayOrientation\(\)](#) 方法可以使得你改变预览的方向，并且不会影响到图片被记录的效果。然而，在Android API level 14之前，你必须在改变方向之前，先停止你的预览，然后才能去重启它。

Take a Picture(拍一张图片)

只要预览开始之后，可以使用[Camera.takePicture\(\)](#) 方法来拍下一张图片。你可以创建Camera.PictureCallback 与 Camera.ShutterCallback 对象并传递他们到Camera.takePicture()中。

如果你想要做连拍的动作，你可以创建一个Camera.PreviewCallback 并实现onPreviewFrame().你还可以选择几个预览帧来进行拍照，或是建立一个延迟拍照的动作。

Restart the Preview(重启预览)

在图片被获取后，你必须在用户拍下一张图片之前重启预览。在下面的示例中，通过重载shutter button来实现重启。

```
@Override
public void onClick(View v) {
    switch(mPreviewState) {
        case K_STATE_FROZEN:
            mCamera.startPreview();
            mPreviewState = K_STATE_PREVIEW;
            break;

        default:
            mCamera.takePicture( null, rawCallback, null);
            mPreviewState = K_STATE_BUSY;
    } // switch
    shutterBtnConfig();
}
```

Stop the Preview and Release the Camera(停止预览并释放相机)

当你的程序在使用Camera之后，有必要做清理的动作。特别是，你必须释放 Camera 对象，不然会引起其他app crash。

那么何时应该停止预览并释放相机呢? 在预览的surface被摧毁之后，可以做停止预览与释放相机的动作。如下所示：

```
public void surfaceDestroyed(SurfaceHolder holder) {
    // Surface will be destroyed when we return, so stop the preview.
    if (mCamera != null) {
        /*
         * Call stopPreview() to stop updating the preview surface.
         */
        mCamera.stopPreview();
    }
}

/**
 * When this function returns, mCamera will be null.
 */
private void stopPreviewAndFreeCamera() {
    if (mCamera != null) {
        /*
         * Call stopPreview() to stop updating the preview surface.
         */
        mCamera.stopPreview();

        /*
         * Important: Call release() to release the camera for use by other applications.
         * Applications should release the camera immediately in onPause() (and re-open() it
         * onResume()).
         */
        mCamera.release();

        mCamera = null;
    }
}
```

在这节课的前面，这一些系列的动作也是setCamera() 方法的一部分，因此初始化一个camera的动作，总是从停止预览开始的。

编写:[jdneo](#)

校对:

打印

Android用户经常需要在设备上单独地浏览信息，但也有时候需要分享信息而不得不给其他人看自己的设备屏幕，这显然不是分享信息的好方法。若能够从你的Android应用打印信息，这将给用户提供一种从你应用获取更多信息的良好方法，这么做还能将信息分享给不使用你的应用的其他人。打印同时还能允许他们创建信息的快照，而这不需要具有充足电量的设备或是无线网络连接。

在Android 4.4（API Level 19）及更高的系统版本中，框架提供了直接从Android应用打印图片和文字的服务。这系列课程将展示如何在你的应用中打印，包括打印图片，HTML页面以及创建自定义的打印文档。

编写:[jdneo](#)

校对:

打印照片

拍摄并分享照片是移动设备最流行的用法之一。如果你的应用拍摄了照片，展示他们，或者允许用户共享照片，你就应该考虑在你的应用中可以打印他们。[Android Support Library](#)提供了一个方便的函数，它可以仅仅使用很少量的代码和一些简单的打印布局配置集，就能打印出照片来。

这堂课将向你展示如何使用v4 support library中的[PrintHelper](#)类来打印一幅图片。

打印一幅图片

Android Support Library中的[PrintHelper](#)类提供了一个打印图片的简单方法。这个类有一个简单的布局选项：[setScaleMode\(\)](#)，它能允许你使用下面的两个选项之一：

- [SCALE_MODE_FIT](#)：这个选项会调整图像大小，这样整个图像就会在打印有效区域内全部显示出来（缩放至长和宽都包含在纸张页面内）。
- [SCALE_MODE_FILL](#)：这个选项同样会调整图像大小使图像充满整个打印有效区域，即让图像充满这个纸张页面。这就意味着如果选择这个选项，那么图片的一部分（顶部和底部，或者左侧和右侧）将无法打印出来。如果你不设置图像拉伸的选项，该模式将是默认的图像拉伸方式。

这两个[setScaleMode\(\)](#)的图像缩放选项都会保持图像原有的长宽比。下面的代码展示了如何创建一个[PrintHelper](#)类的实例，设置缩放选项，并开始打印进程：

```
private void doPhotoPrint() {
    PrintHelper photoPrinter = new PrintHelper(getActivity());
    photoPrinter.setScaleMode(PrintHelper.SCALE_MODE_FIT);
    Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
        R.drawable.droids);
    photoPrinter.printBitmap("droids.jpg - test print", bitmap);
}
```

这个方法可以作为一个菜单项的行为来被调用。注意对于那些不一定都能支持的菜单项（比如打印），应该放置在“更多菜单（overflow menu）”中。要获取更多知识，可以阅读：[Action Bar](#)。

在[printBitmap\(\)](#)被调用之后，你的应用不再需要其他的操作了。之后Android打印界面就会出现，允许用户选择一个打印机和它的打印选项。之后用户就可以打印图像或者取消这一次操作。如果用户选择了打印图像，那么一个打印的任务就被创建了，并且一个打印的提醒通知会显示在系统的任务栏中。

如果你希望在你的打印输出中包含更多的内容，而不仅仅是一张图片，你就必须构造一个打印文档。这方面知识将会在后面的两节课程中展开。

编写:[jdneo](#)

校对:

打印HTML文档

在Android上要打印比一副照片更丰富的内容，换句话说，若需要将文本和图片组合在一个打印的文档中。Android框架提供了一种使用HTML语言来组织一个文档并打印的方法，它使用的代码数量是很小的。

在Android 4.4（API Level 19），[WebView](#)类更新了，使得它可以打印HTML内容。这个类允许你加载一个本地的HTML资源或者从一个网页下载一个页面，创建一个打印任务，并把它交给Android打印服务。

这节课将向您展示如何快速地构建一个HTML文档，它包含文本和图片，并使用[WebView](#)来打印它。

加载一个HTML文档

用[WebView](#)打印一个HTML文档包含加载一个HTML资源或者以String的形式构建一个HTML文档。这一节将描述如果构建一个HTML的字符串并将它加载到[WebView](#)中，以备打印。

这个View对象一般被用来作为一个activity布局的一部分。然而，如果你的应用不使用[WebView](#)，你可以创建一个该类的实例，以进行打印。创建该自定义打印界面的主要步骤是：

1. 在HTML资源加载完毕后，创建一个[WebViewClient](#)用来启动一个打印任务。
2. 加载HTML资源至[WebView](#)对象。

下面的代码展示了如何创建一个简单的[WebViewClient](#)并且加载一个动态创建的HTML文档：

```
private WebView mWebView;

private void doWebViewPrint() {
    // Create a WebView object specifically for printing
    WebView webView = new WebView(getActivity());
    webView.setWebViewClient(new WebViewClient() {

        public boolean shouldOverrideUrlLoading(WebView view, String url) {
            return false;
        }

        @Override
        public void onPageFinished(WebView view, String url) {
            Log.i(TAG, "page finished loading " + url);
            createWebPrintJob(view);
            mWebView = null;
        }
    });

    // Generate an HTML document on the fly:
    String htmlDocument = "<html><body><h1>Test Content</h1><p>Testing, " +
        "testing, testing...</p></body></html>";
    webView.loadDataWithBaseURL(null, htmlDocument, "text/HTML", "UTF-8", null);

    // Keep a reference to WebView object until you pass the PrintDocumentAdapter
    // to the PrintManager
    mWebView = webView;
}
```

Note：确保你所调用的生成打印的任务发生在在之前那一节所创建的[WebViewClient](#)中的[onPageFinished\(\)](#)方法内。如果你不等待页面加载完毕后再打印，打印的输出可能会不完整或空白，甚至可能会失败。

Note：上面的样例代码维护了一个[WebView](#)对象实例，这样就保证了它不会在打印任务创建之前就被垃圾回收器所回收。请确保你在你的实现中也同样这么做，否则打印的进程可能会无法继续执行。

如果你希望页面中包含图像，将这个图像文件放置在你的工程的“assets/”目录，并指定一个基URL，作为[loadDataWithBaseURL\(\)](#)方法的第一个参数，就像下面所显示的一样：

```
webView.loadDataWithBaseURL("file:///android_asset/images/", htmlBody,
    "text/HTML", "UTF-8", null);
```

你也可以加载一个网页来打印，方法是[将loadDataWithBaseURL\(\)方法替换为loadUrl\(\)](#)，如下所示：

```
// Print an existing web page (remember to request INTERNET permission!):
webView.loadUrl("http://developer.android.com/about/index.html");
```

当使用[WebView](#)来创建一个打印文档时，你要注意下面的一些限制：

- 你不能为文档添加页眉和页脚，包括页号。
- HTML文档的打印选项不包含选择打印的页数范围，例如：对于一个10页的HTML文档，只打印2到4页是不可以的。
- 一个[WebView](#)的实例只能在同一时间处理一个打印任务。
- 若一个HTML文档包含CSS打印属性，比如一个landscape属性，是不支持的。
- 你不能使用一个HTML文档中的JavaScript来激活打印。

Note：一旦在布局中包含的[WebView](#)对象加载好了文档，就可以打印[WebView](#)对象的内容。

如果你希望创建一个更加自定义化的打印输出并希望可以完全控制打印页面上绘制的内容，可以学习下一节课程：[Printing a Custom Document](#)。

创建一个打印任务

在创建了[WebView](#)并加载了你的HTML内容之后，你的应用就基本完成了打印进程的归属于它的部分的任务。下一步是访问[PrintManager](#)，创建一个打印适配器，并在最后，创建一个打印任务。下面的代码展示了如何执行这些步骤：

```
private void createWebPrintJob(WebView webView) {  
  
    // Get a PrintManager instance  
    PrintManager printManager = (PrintManager) getActivity()  
        .getSystemService(Context.PRINT_SERVICE);  
  
    // Get a print adapter instance  
    PrintDocumentAdapter printAdapter = webView.createPrintDocumentAdapter();  
  
    // Create a print job with name and adapter instance  
    String jobName = getString(R.string.app_name) + " Document";  
    PrintJob printJob = printManager.print(jobName, printAdapter,  
        new PrintAttributes.Builder().build());  
  
    // Save the job object for later status checking  
    mPrintJobs.add(printJob);  
}
```

这个例子保存了应用使用的[PrintJob](#)对象的实例，这是不必须的。你的应用可以使用这个对象来跟踪打印任务执行时的进度。当你希望监控你应用中的打印任务是否完成，是否失败或者是否被用户取消，这个方法非常有用。另外，不需要创建一个应用内置的通知，因为打印框架会自动的创建一个该打印任务的系统通知。

编写:[jdneo](#)

校对:

打印自定义文档

对一些应用，比如绘图应用，页面布局应用和其它一些聚焦于图像输出的应用，创建美丽的打印页面是它的核心功能。在这种情况下，仅仅打印一副图片或一个HTML文档就不够了。这种类型应用的打印输出需要精确地控制每个进入页面的东西，包括字体，文本流，分页符，页眉，页脚和一些图像元素。

创建完全由你自定义的打印输出需要投入比之前讨论的方法更多的编程精力。你必须构建可以和打印架构相互通信的组件，调整打印选项，绘制页面元素并管理多个页面的打印。

这节课将向你展示如何连接打印管理器，创建一个打印适配器并构建要打印的内容。

连接打印管理器

当你的应用直接管理打印进程，在收到来自用户的打印请求后，第一步要做的是连接Android打印框架并获取一个[PrintManager](#)类的实例。这个类允许你初始化一个打印任务并开始打印生命周期。下面的代码展示了如何获得打印管理器并开始打印进程。

```
private void doPrint() {
    // Get a PrintManager instance
    PrintManager printManager = (PrintManager) getActivity()
        .getSystemService(Context.PRINT_SERVICE);

    // Set job name, which will be displayed in the print queue
    String jobName = getActivity().getString(R.string.app_name) + " Document";

    // Start a print job, passing in a PrintDocumentAdapter implementation
    // to handle the generation of a print document
    printManager.print(jobName, new MyPrintDocumentAdapter(getActivity()),
        null); //
}
```

上面的代码展示了如何命名一个打印任务并且设置一个[PrintDocumentAdapter](#)类的实例，它处理打印生命周期的每一步。打印适配器的实现会在下一节中进行讨论。

Note : [print\(\)](#)方法的最后一个参数接收一个[PrintAttributes](#)对象。你可以使用这个参数来提供对于打印框架的提示，以及基于前一个打印周期的预设，从而改善用户体验。你也可以使用这个参数对被打印对象进行设置一些更符合实际情况的设定，比如当打印一副照片时，设置打印的方向与照片方向一致。

创建一个打印适配器

打印适配器和Android打印框架交互并处理打印过程的每一步。这个过程需要用户在创建打印文档前选择打印机和打印选项。这些选项可以影响最终的输出，因为用户选择的打印机可能会有不同的打印的能力，不同的页面尺寸或不同的页面方向。当这些选项配置好之后，这个打印框架会询问你的适配器进行布局和生成一个打印文档，作为最终打印的前期准备。一旦用户点击了打印按钮，框架会接收最终的打印文档，并将它传递给一个打印提供程序来打印输出。在打印过程中，用户可以选择取消打印，所以你的打印适配器必须监听并响应一个取消请求。

[PrintDocumentAdapter](#)抽象类被设计用来处理打印的生命周期，它有四个主要的回调函数。你必须在你的打印适配器中实现这些方法，以此来恰当地和打印框架交互：

- [onStart\(\)](#)：一旦打印的进程开始了就被调用。如果你的应用有任何一次性的准备任务要执行，比如获取一个要打印数据的快照，那么将它们在此处执行。在你的适配器中，这个回调函数不是必须实现的。
- [onLayout\(\)](#)：每次一个用户改变了一个打印设置并影响了打印的输出时调用，比如改变了页面的尺寸，或者页面的方向，给你的应用一个机会去重新计算要打印页面的布局。这个方法必须返回打印文档包含多少页面。
- [onWrite\(\)](#)：调用它以此将打印页面交付给一个要打印的文件。这个方法可以在被[onLayout\(\)](#)调用后调用一次或多次。
- [onFinish\(\)](#)：一旦打印进程结束后被调用。如果你的应用有任何一次性销毁任务要执行，在这里执行。这个回调函数不是必须实现的。

下面的部分将介绍如何实现布局和写方法，这两个方法是一个打印适配器的核心功能。

Note：这些适配器的回调函数会在你的主线程上被调用。如果你的这些方法的实现需要花费大量的时间，那么应该在一个另外的线程里执行。例如：你可以将布局或者写入打印文档的操作封装在一个[AsyncTask](#)对象中。

计算打印文档信息

在一个[PrintDocumentAdapter](#)类的实现中，你的应用必须指定所创建文档的类型并计算所有打印任务所需要的页数，提供被打页面尺寸信息。在适配器中[onLayout\(\)](#)方法的实现中会执行这些计算，并提供打印任务输出的信息，这些信息在一个[PrintDocumentInfo](#)类中，包括页数和内容类型。下面的例子展示了[PrintDocumentAdapter](#)中[onLayout\(\)](#)方法的基本实现：

```
@Override
public void onLayout(PrintAttributes oldAttributes,
                    PrintAttributes newAttributes,
                    CancellationSignal cancellationSignal,
                    LayoutResultCallback callback,
                    Bundle metadata) {
    // Create a new PdfDocument with the requested page attributes
    mPdfDocument = new PrintedPdfDocument(getActivity(), newAttributes);

    // Respond to cancellation request
    if (cancellationSignal.isCancelled() ) {
        callback.onLayoutCancelled();
        return;
    }

    // Compute the expected number of printed pages
    int pages = computePageCount(newAttributes);

    if (pages > 0) {
        // Return print information to print framework
        PrintDocumentInfo info = new PrintDocumentInfo
            .Builder("print_output.pdf")
            .setContentType(PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
            .setPageCount(pages);
        .build();
        // Content layout reflow is complete
        callback.onLayoutFinished(info, true);
    } else {
        // Otherwise report an error to the print framework
        callback.onLayoutFailed("Page count calculation failed.");
    }
}
```

[onLayout\(\)](#)方法的执行结果有三种：完成，取消或失败（计算布局无法顺利完成时会失败）。你必须通过调用[PrintDocumentAdapter.LayoutResultCallback](#)对象中的适当方法来指明这些结果中的一个。

Note：[onLayoutFinished\(\)](#)方法的布尔参数明确了这个布局内容是否和上一次请求相比确实改变了。恰当地设定了这个参数将避免打印框架不必要的调用[onWrite\(\)](#)方法，缓存之前的打印文档，并提升性能。

[onLayout\(\)](#)的主要工作是计算打印文档的页数，作为交给打印机的参数。如何计算页数则高度依赖于你的应用时如何布局打印页面的。下面的代码展示了页数是如何根据打印方向确定的：

```
private int computePageCount(PrintAttributes printAttributes) {
    int itemsPerPage = 4; // default item count for portrait mode

    MediaSize pageSize = printAttributes.getMediaSize();
    if (!pageSize.isPortrait()) {
        // Six items per page in landscape orientation
        itemsPerPage = 6;
    }

    // Determine number of print items
    int printItemCount = getPrintItemCount();

    return (int) Math.ceil(printItemCount / itemsPerPage);
}
```

将打印文档写入文件

当需要将打印输出写入一个文件时，Android打印框架会调用你的应用[PrintDocumentAdapter](#)类的[onWrite\(\)](#)方法。这个方法的参数指定了哪一页要被打印以及要使用的输出文件。你的这个方法的实现必须将每一个请求页的内容交付给一个多页PDF文档文件。当这个过程结束以后，你需要调用回调对象的[onWriteFinished\(\)](#)方法。

Note : Android打印框架可能会在每次调用[onLayout\(\)](#)后，调用[onWrite\(\)](#)方法一次甚至更多次。在这节课当中，有一件非常重要的事情是当打印内容的布局没有变化时，需要将[onLayoutFinished\(\)](#)方法的布尔参数设置为“false”，以避免不必要的重写打印文档的操作。

Note : [onLayoutFinished\(\)](#)方法的布尔参数明确了这个布局内容是否和上一次请求相比确实改变了。恰当地设定了这个参数将避免打印框架不必要的调用[onLayout\(\)](#)方法，缓存之前的打印文档，并提升性能。

下面的代码展示了使用[PrintedPdfDocument](#)类的打印过程基本原理，并创建了一个PDF文件：

```
@Override
public void onWrite(final PageRange[] pageRanges,
                    final ParcelFileDescriptor destination,
                    final CancellationSignal cancellationSignal,
                    final WriteResultCallback callback) {
    // Iterate over each page of the document,
    // check if it's in the output range.
    for (int i = 0; i < totalPages; i++) {
        // Check to see if this page is in the output range.
        if (containsPage(pageRanges, i)) {
            // If so, add it to writtenPagesArray. writtenPagesArray.size()
            // is used to compute the next output page index.
            writtenPagesArray.append(writtenPagesArray.size(), i);
            PdfDocument.Page page = mPdfDocument.startPage(i);

            // check for cancellation
            if (cancellationSignal.isCancelled()) {
                callback.onWriteCancelled();
                mPdfDocument.close();
                mPdfDocument = null;
                return;
            }

            // Draw page content for printing
            drawPage(page);

            // Rendering is complete, so page can be finalized.
            mPdfDocument.finishPage(page);
        }
    }

    // Write PDF document to file
    try {
        mPdfDocument.writeTo(new FileOutputStream(
            destination.getFileDescriptor()));
    } catch (IOException e) {
        callback.onWriteFailed(e.toString());
        return;
    } finally {
    }
```

```
        mPdfDocument.close();
        mPdfDocument = null;
    }
    PageRange[] writtenPages = computeWrittenPages();
    // Signal the print framework the document is complete
    callback.onWriteFinished(writtenPages);

    ...
}
```

这个代码中将PDF页面递交给了drawPage()方法，这个方法会在下一部分介绍。

就布局而言，[onWrite\(\)](#)方法的执行可以有三种结果：完成，取消或者失败（内容无法被写入）。你必须通过调用[PrintDocumentAdapter.WriteResultCallback](#)对象中的适当方法来指明这些结果中的一个。

Note：递交一个打印的文档可以是一个和大量资源相关的操作。为了避免阻塞应用的主UI线程，你应该考虑将页面的递交和写操作在另一个线程中执行，比如在AsyncTask[\[http://developer.android.com/reference/android/os/AsyncTask.html\]](http://developer.android.com/reference/android/os/AsyncTask.html)中。关于更多异步任务线程的知识，可以阅读：[\[Processes and Threads\]](#) (<http://developer.android.com/guide/components/processes-and-threads.html>)。

绘制PDF页面内容

当你的应用打印时，你的应用必须生成一个PDF文档并将它传递给Android打印框架来打印。你可以使用任何PDF生成库来协助完成这个操作。本节将展示如何使用[PrintedPdfDocument](#)类从你的内容生成PDF页面。

[PrintedPdfDocument](#)类使用一个[Canvas](#)对象来在PDF页面上绘制元素，和在activity布局上进行绘制很类似。你可以再打印页面上使用[Canvas](#)的绘图方法绘制元素。下面的代码展示了如何使用相关的函数在PDF文档页面上绘制简单元素：

```
private void drawPage(PdfDocument.Page page) {
    Canvas canvas = page.getCanvas();

    // units are in points (1/72 of an inch)
    int titleBaseLine = 72;
    int leftMargin = 54;

    Paint paint = new Paint();
    paint.setColor(Color.BLACK);
    paint.setTextSize(36);
    canvas.drawText("Test Title", leftMargin, titleBaseLine, paint);

    paint.setTextSize(11);
    canvas.drawText("Test paragraph", leftMargin, titleBaseLine + 25, paint);

    paint.setColor(Color.BLUE);
    canvas.drawRect(100, 100, 172, 172, paint);
}
```

当使用[Canvas](#)在一个PDF页面上绘图时，元素通过单位“点（point）”来指定大小，它是七十二分之一英寸大小。确保你使用这个测量单位来指定页面上的元素大小。在定位绘制的元素时，坐标系的原点（即（0,0））在页面的最左上角。

Tip：虽然[Canvas](#)对象允许你将打印元素放置在一个PDF文档的边缘，但许多打印机并不能再纸张边缘打印。所以当使用这个类构建一个打印文档时，确保你考虑了那些无法打印的边缘区域。

编写:

校对:

图像

编写:[kesenhoo](#)

校对:

高效显示Bitmap

这一章节会介绍一些通用的用来处理与加载Bitmap对象的方法，这些技术能够使得不会卡到程序的UI并且避免程序消耗过度内存。如果你不注意这些，Bitmaps会迅速的消耗你可用的内存而导致程序crash,出现下面的异常：`java.lang.OutOfMemoryError: bitmap size exceeds VM budget`.

有许多原因说明在你的Android程序中加载Bitmaps是非常棘手的，需要你特别注意：

- 移动设备的系统资源有限。Android设备对于单个程序至少需要16MB的内存。[Android Compatibility Definition Document \(CDD\)](#), Section 3.7. Virtual Machine Compatibility 给出了对于不同大小与密度的屏幕的最低内存需求。程序应该在这个最低内存限制下最优化程序的效率。当然，大多数设备的都有更高的限制需求。
- Bitmap会消耗很多内存，特别是对于类似照片等更加丰富的图片。例如，Galaxy Nexus的照相机能够拍摄2592x1936 pixels (5 MB)的图片。如果bitmap的配置是使用ARGB_8888 (the default from the Android 2.3 onward)，那么加载这张照片到内存会大概需要19MB(2592*1936*4 bytes) 的内存，这样的话会迅速消耗掉设备的整个内存。
- Android app的UI通常会在一次操作中立即加载许多张bitmaps。例如在ListView, GridView 与 ViewPager 等组件中通常会需要一次加载许多张bitmaps，而且需要多加载一些内容为了用户可能的滑动操作。

Lessons

- [Loading Large Bitmaps Efficiently:高效的加载大图](#)

这节课会带领你学习如何解析很大的Bitmaps并且避免超出程序的内存限制。

- [Processing Bitmaps Off the UI Thread:非UI线程处理Bitmaps](#)

处理Bitmap(裁剪,下载等操作)不能执行在主线程。这节课会带领你学习如何使用AsyncTask在后台线程对Bitmap进行处理,并解释如何处理并发带来的问题。

- [Caching Bitmaps:缓存Bitmap](#)

这节课会带领你学习如何使用内存与磁盘缓存来提升加载多张Bitmaps时的响应速度与流畅度。

- [Managing Bitmap Memory:管理Bitmap占用的内存](#)

这节课会介绍为了最大化程序的性能如何管理Bitmap的内存占用。

- [Displaying Bitmaps in Your UI](#)

这节课会把前面介绍的内容综合起来,演示如何在类似ViewPager与GridView的控件中使用后台线程与缓存进行加载多张Bitmaps。

编写:[kesenhoo](#)

校对:

Loading Large Bitmaps Efficiently(有效地加载大尺寸位图)

图片有不同的形状与大小。在大多数情况下它们的实际大小都比需要呈现出来的要大很多。例如，系统的Gallery程序会显示那些你使用设备camera拍摄的图片，但是那些图片的分辨率通常都比你的设备屏幕分辨率要高很多。

考虑到程序是在有限的内存下工作，理想情况是你只需要在内存中加载一个低分辨率的版本即可。这个低分辨率的版本应该是与你的UI大小所匹配的，这样才便于显示。一个高分辨率的图片不会提供任何可见的好处，却会占用宝贵的(precious)的内存资源，并且会在快速滑动图片时导致(incurs)附加的效率问题。

这一课会介绍如何通过加载一个低版本的图片到内存中去decoding大的bitmaps，从而避免超出程序的内存限制。

Read Bitmap Dimensions and Type(读取位图的尺寸与类型)

BitmapFactory 类提供了一些decode的方法 ([decodeByteArray\(\)](#), [decodeFile\(\)](#), [decodeResource\(\)](#), etc.) 用来从不同的资源中创建一个Bitmap. 根据你的图片数据源来选择合适的decode方法. 那些方法在构造位图的时候会尝试分配内存, 因此会容易导致 OutOfMemory 的异常。每一种decode方法都提供了通过[BitmapFactory.Options](#) 来设置一些附加的标记来指定decode的选项。设置 [inJustDecodeBounds](#) 属性为true可以在decoding的时候避免内存的分配, 它会返回一个null的bitmap, 但是 outWidth, outHeight 与 outMimeType 还是可以获取。这个技术可以允许你在构造bitmap之前优先读图片的尺寸与类型。

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

为了避免java.lang.OutOfMemory 的异常, 我们在真正decode图片之前检查它的尺寸, 除非你确定这个数据源提供了准确无误的图片且不会导致占用过多的内存。

Load a Scaled Down Version into Memory(加载一个按比例缩小的版本到内存中)

通过上面的步骤我们已经知道了图片的尺寸，那些数据可以用来决定是应该加载整个图片到内存中还是加一个缩小的版本。下面有一些因素需要考虑：

- 评估加载完整图片所需要耗费的内存。
- 程序在加载这张图片时会涉及到其他内存需求。
- 呈现这张图片的组件的尺寸大小。
- 屏幕大小与当前设备的屏幕密度。

例如，如果把一个原图是1024*768 pixel的图片显示到ImageView为128*96 pixel的缩略图就没有必要把整张图片都加载到内存中。

为了告诉decoder去加载一个低版本的图片到内存，需要在你的BitmapFactory.Options 中设置 inSampleSize 为 true。For example, 一个分辨率为2048x1536 的图片，如果设置 inSampleSize 为4，那么会产出一个大概为512x384的bitmap。加载这张小的图片仅使用大概0.75MB，如果是加载全图那么大概要花费12MB(前提都是bitmap的配置是 ARGB_8888)。下面有一段根据目标图片大小来计算Sample图片大小的Sample Code:

```
public static int calculateInSampleSize(
    BitmapFactory.Options options, int reqWidth, int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {
        if (width > height) {
            inSampleSize = Math.round((float)height / (float)reqHeight);
        } else {
            inSampleSize = Math.round((float)width / (float)reqWidth);
        }
    }
    return inSampleSize;
}
```

Note: 设置 inSampleSize 为2的幂对于decoder会更加的有效率，然而，如果你打算把调整过大小的图片Cache到磁盘上，设置为更加接近的合适大小则能够更加有效的节省缓存的空间。

为了使用这个方法，首先需要设置 inJustDecodeBounds 为 true, 把options的值传递过来，然后使用 inSampleSize 的值并设置 inJustDecodeBounds 为 false 来重新Decode一遍。

```
public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
    int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

使用上面这个方法可以简单的加载一个任意大小的图片并显示为100*100 pixel的缩略图形式。像下面演示的一样：

```
mImageView.setImageBitmap(
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));
```

你可以通过替换合适的BitmapFactory.decode* 方法来写一个类似的方法从其他的数据源进行decode bitmap。

编写:[kesenhoo](#)

校对:

非UI线程处理Bitmap

在上一课中有介绍一系列的BitmapFactory.decode* 方法，当数据源是网络或者是磁盘时(或者是任何实际源不在内存的)，这些方法都不应该在main UI 线程中执行。那些情况下加载数据是不可以预知的，它依赖于许多因素(从网络或者硬盘读取数据的速度, 图片的大小, CPU的速度, etc.)。如果其中任何一个任务卡住了UI thread, 系统会出现ANR的错误。

这一节课会介绍如何使用 AsyncTask 在后台线程中处理bitmap并且演示了如何处理并发(concurrency)的问题。

Use an AsyncTask(使用AsyncTask)

AsyncTask 类提供了一个简单的方法在后台线程执行一些操作，并且可以把后台的结果呈现到UI线程。下面是一个加载大图的示例：

```
class BitmapWorkerTask extends AsyncTask {
    private final WeakReference<ImageView> imageViewReference;
    private int data = 0;

    public BitmapWorkerTask(ImageView imageView) {
        // Use a WeakReference to ensure the ImageView can be garbage collected
        imageViewReference = new WeakReference<ImageView>(imageView);
    }

    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        data = params[0];
        return decodeSampledBitmapFromResource(getResources(), data, 100, 100);
    }

    // Once complete, see if ImageView is still around and set bitmap.
    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

为ImageView使用WeakReference 确保了 AsyncTask 所引用的资源可以被GC(garbage collected)。因为当任务结束时不能确保ImageView 仍然存在，因此你必须在 onPostExecute() 里面去检查引用。这个ImageView 也许已经不存在了，例如，在任务结束时用户已经不在那个Activity或者是设备已经发生配置改变(旋转屏幕等)。

开始异步加载位图，只需要创建一个新的任务并执行它即可：

```
public void loadBitmap(int resId, ImageView imageView) {
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);
    task.execute(resId);
}
```


Handle Concurrency(处理并发问题)

通常类似 ListView 与 GridView 等视图组件在使用上面演示的 AsyncTask 方法时会同时带来另外一个问题。为了更有效的处理内存，那些视图的子组件会在用户滑动屏幕时被循环使用。如果每一个子视图都触发一个 AsyncTask，那么就无法确保当前视图在结束 task 时，分配的视图已经进入循环队列中给另外一个子视图进行重用。而且，无法确保所有的异步任务能够按顺序执行完毕。

[Multithreading for Performance](#) 这篇博文更进一步的讨论了如何处理并发并且提供了一种解决方法，当任务结束时 ImageView 保存一个最近常使用的 AsyncTask 引用。使用类似的方法，AsyncTask 可以扩展出一个类似的模型。创建一个专用的 Drawable 子类来保存一个可以回到当前工作任务的引用。在这种情况下，BitmapDrawable 被用来作为占位图片，它可以在任务结束时显示到 ImageView 中。

```
static class AsyncDrawable extends BitmapDrawable {
    private final WeakReference bitmapWorkerTaskReference;

    public AsyncDrawable(Resources res, Bitmap bitmap,
        BitmapWorkerTask bitmapWorkerTask) {
        super(res, bitmap);
        bitmapWorkerTaskReference =
            new WeakReference(bitmapWorkerTask);
    }

    public BitmapWorkerTask getBitmapWorkerTask() {
        return bitmapWorkerTaskReference.get();
    }
}
```

在执行 BitmapWorkerTask 之前，你需要创建一个 AsyncDrawable 并且绑定它到目标组件 ImageView 中：

```
public void loadBitmap(int resId, ImageView imageView) {
    if (cancelPotentialWork(resId, imageView)) {
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
            new AsyncDrawable(getResources(), mPlaceholderBitmap, task);
        imageView.setImageDrawable(asyncDrawable);
        task.execute(resId);
    }
}
```

在上面的代码示例中，cancelPotentialWork 方法检查确保了另外一个在 ImageView 中运行的任务得以取消。如果是这样，它通过执行 cancel() 方法来取消之前的一个任务。在小部分情况下，New 出来的任务有可能已经存在，这样就不需要执行这个任务了。下面演示了如何实现一个 cancelPotentialWork。

```
public static boolean cancelPotentialWork(int data, ImageView imageView) {
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

    if (bitmapWorkerTask != null) {
        final int bitmapData = bitmapWorkerTask.data;
        if (bitmapData != data) {
            // Cancel previous task
            bitmapWorkerTask.cancel(true);
        } else {
            // The same work is already in progress
            return false;
        }
    }
    // No task associated with the ImageView, or an existing task was cancelled
    return true;
}
```

在上面有一个帮助方法，getBitmapWorkerTask(), 被用作检索任务是否已经被分配到指定的 ImageView:

```
private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
    if (imageView != null) {
        final Drawable drawable = imageView.getDrawable();
        if (drawable instanceof AsyncDrawable) {
            final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
            return asyncDrawable.getBitmapWorkerTask();
        }
    }
    return null;
}
```

最后一步是在BitmapWorkerTask 的onPostExecute() 方法里面做更新操作:

```
class BitmapWorkerTask extends AsyncTask {
    ...

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            final BitmapWorkerTask bitmapWorkerTask =
                getBitmapWorkerTask(imageView);
            if (this == bitmapWorkerTask && imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

这个方法不仅仅适用于 ListView 与 GridView 组件，在那些需要循环利用子视图的组件中同样适用。只需要在设置图片到 ImageView 的地方调用 loadBitmap 方法。例如，在 GridView 中实现这个方法会是在 getView() 方法里面调用。

编写:[kesenhoo](#)

校对:

Cached Bitmap

加载单个Bitmap到UI是简单直接的，但是如果你需要一次加载大量的图片，事情则会变得复杂起来。在大多数情况下(例如在ListView,GridView or ViewPager), 显示图片的数量通常是没有限制的。

通过循环利用子视图可以抑制内存的使用，GC(garbage collector)也会释放那些不再需要使用的bitmap。这些机制都非常好，但是为了保持一个流畅的用户体验，你想要在屏幕滑回来时避免每次重复处理那些图片。内存与磁盘缓存通常可以起到帮助的作用，允许组件快速的重新加载那些处理过的图片。

这一课会介绍在加载多张位图时使用内存Cache与磁盘Cache来提高反应速度与UI的流畅度。

Use a Memory Cache(使用内存缓存)

内存缓存以花费宝贵的程序内存为前提来快速访问位图。[LruCache](#) 类(在Support Library 中也可以找到) 特别合适用来caching bitmaps, 用一个strong referenced的 LinkedHashMap 来保存最近引用的对象, 并且在Cache超出设置大小的时候踢出(evict)最近最少使用到的对象。

Note: 在过去, 一个比较流行的内存缓存实现方法是使用 SoftReference or WeakReference, 然而这是不推荐的。从Android 2.3 (API Level 9) 开始, GC变得更加频繁的去释放soft/weak references, 这使得他们就显得效率低下. 而且在Android 3.0 (API Level 11)之前, 备份的bitmap是存放在native memory 中, 它不是以可预知的方式被释放, 这样可能导致程序超出它的内存限制而崩溃。

为了给LruCache选择一个合适的大小, 有下面一些因素需要考虑到:

- 你的程序剩下多少可用的内存?
- 多少图片会被一次呈现到屏幕上? 有多少图片需要准备好以便马上显示到屏幕?
- 设备的屏幕大小与密度是多少? 一个具有特别高密度屏幕(xhdpi)的设备, 像 Galaxy Nexus 会比 Nexus S (hdpi)需要一个更大的Cache来hold住同样数量的图片.
- 位图的尺寸与配置是多少, 会花费多少内存?
- 图片被访问的频率如何? 是其中一些比另外的访问更加频繁吗? 如果是, 也许你想要保存那些最常访问的到内存中, 或者为不同组别的位图(按访问频率分组)设置多个LruCache 对象。
- 你可以平衡质量与数量吗? 某些时候保存大量低质量的位图会非常有用, 在加载更高质量图片的任务则交给另外一个后台线程。

没有指定的大小与公式能够适用与所有的程序, 那取决于分析你的使用情况后提出一个合适的解决方案。一个太小的Cache会导致额外的开销却没有明显的好处, 一个太大的Cache同样会导致java.lang.OutOfMemory的异常(Cache占用太多内存, 其他活动则会因为内存不够而异常), 并且使得你的程序只留下小部分的内存用来工作。

下面是一个为bitmap建立LruCache 的示例:

```
private LruCache mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Get memory class of this device, exceeding this amount will throw an
    // OutOfMemory exception.
    final int memClass = ((ActivityManager) context.getSystemService(
        Context.ACTIVITY_SERVICE)).getMemoryClass();

    // Use 1/8th of the available memory for this memory cache.
    final int cacheSize = 1024 * 1024 * memClass / 8;

    mMemoryCache = new LruCache(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // The cache size will be measured in bytes rather than number of items.
            return bitmap.getByteCount();
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

Note: 在上面的例子中, 有1/8的程序内存被作为Cache. 在一个常见的设备上(hdpi), 最小大概有4MB (32/8). 如果一个填满图片的GridView组件放置在800x480像素的手机屏幕上, 大概会花费1.5MB (800x480x4 bytes), 因此缓存的容量大概可以缓存2.5页的图片内容。

当加载位图到 ImageView 时, LruCache 会先被检查是否存在这张图片。如果找到有, 它会被用来立即更新 ImageView 组件, 否则一个后台线程则被触发去处理这张图片。

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);

    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        mImageView.setImageBitmap(bitmap);
    }
}
```

```
    } else {  
        mImageView.setImageResource(R.drawable.image_placeholder);  
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);  
        task.execute(resId);  
    }  
}
```

上面的程序中 BitmapWorkerTask 也需要做添加到内存Cache中的动作：

```
class BitmapWorkerTask extends AsyncTask {  
    ...  
    // Decode image in background.  
    @Override  
    protected Bitmap doInBackground(Integer... params) {  
        final Bitmap bitmap = decodeSampledBitmapFromResource(  
            getResources(), params[0], 100, 100);  
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);  
        return bitmap;  
    }  
    ...  
}
```

Use a Disk Cache(使用磁盘缓存)

内存缓存能够提高访问最近查看过的位图，但是你不能保证这个图片会在Cache中。像类似 GridView 等带有大量数据的组件很容易就填满内存Cache。你的程序可能会被类似Phone call等任务而中断，这样后台程序可能会被杀死，那么内存缓存就会被销毁。一旦用户恢复前面的状态，你的程序就又要为每个图片重新处理。

磁盘缓存磁盘缓存可以用来保存那些已经处理好的位图，并且在那些图片在内存缓存中不可用时减少加载的次数。当然从磁盘读取图片会比从内存要慢，而且读取操作需要在后台线程中处理，因为磁盘读取操作是不可预期的。

Note:如果图片被更频繁的访问到，也许使用 ContentProvider 会更加的合适，比如在Gallery程序中。

在下面的sample code中实现了一个基本的 DiskLruCache。然而，Android 4.0 的源代码提供了一个更加robust并且推荐使用的 DiskLruCache 方案。(libcore/luni/src/main/java/libcore/io/DiskLruCache.java). 因为向后兼容，所以在前面发布的Android版本中也可以使用。(quick search 提供了一个实现这个解决方案的示例)。

```
private DiskLruCache mDiskCache;
private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB
private static final String DISK_CACHE_SUBDIR = "thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Initialize memory cache
    ...
    File cacheDir = getCacheDir(this, DISK_CACHE_SUBDIR);
    mDiskCache = DiskLruCache.openCache(this, cacheDir, DISK_CACHE_SIZE);
    ...
}

class BitmapWorkerTask extends AsyncTask {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);

        // Check disk cache in background thread
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);

        if (bitmap == null) { // Not found in disk cache
            // Process as normal
            final Bitmap bitmap = decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100);
        }

        // Add final bitmap to caches
        addBitmapToCache(String.valueOf(imageKey), bitmap);

        return bitmap;
    }
    ...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // Add to memory cache as before
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    // Also add to disk cache
    if (!mDiskCache.containsKey(key)) {
        mDiskCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    return mDiskCache.get(key);
}

// Creates a unique subdirectory of the designated app cache directory. Tries to use external
// but if not mounted, falls back on internal storage.
public static File getCacheDir(Context context, String uniqueName) {
    // Check if media is mounted or storage is built-in, if so, try and use external cache d
    // otherwise use internal cache dir
    final String cachePath = Environment.getExternalStorageState() == Environment.MEDIA_MOUN
        || !Environment.isExternalStorageRemovable() ?
```

```
context.getExternalCacheDir().getPath() : context.getCacheDir().getPath()  
  
    return new File(cachePath + File.separator + uniqueName);  
}
```

内存缓存的检查是可以在UI线程中进行的，磁盘缓存的检查需要在后台线程中处理。磁盘操作永远都不应该在UI线程中发生。当图片处理完成后，最后的位图需要添加到内存缓存与磁盘缓存中，方便之后的使用。

Handle Configuration Changes(处理配置改变)

运行时配置改变，例如屏幕方向的变化会导致Android去destory并restart当前运行的Activity。(关于这一行为的更多信息，请参考[Handling Runtime Changes](#)). 你想要在配置改变时避免重新处理所有的图片，这样才能提供给用户一个良好的平滑过度的体验。

幸运的是，在前面介绍Use a Memory Cache的部分，你已经知道如何建立一个内存缓存。这个缓存可以通过使用一个Fragment去调用 [setRetainInstance\(true\)](#) 传递到新的Activity中。在这个activity被recreate之后，这个保留的 Fragment 会被重新附着上。这样你就可以访问Cache对象，从中获取到图片信息并快速的重新添加到ImageView对象中。

下面配置改变时使用Fragment来重新获取LruCache 的示例：

```
private LruCache mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    RetainFragment mRetainFragment =
        RetainFragment.findOrCreateRetainFragment(getFragmentManager());
    mMemoryCache = RetainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache(cacheSize) {
            ... // Initialize cache here as usual
        }
        mRetainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
        }
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
    }
}
```

为了测试上面的效果，尝试对比retaining 这个 Fragment.与没有这样做的时候去旋转屏幕。你会发现从内存缓存中重新绘制几乎没有卡的现象，而从磁盘缓存则显得稍慢，如果两个缓存中都没有，则处理速度像平时一样。

编写:[kesenhoo](#)

校对:

管理Bitmap的内存占用

作为缓存Bitmaps的进一步延伸, 为了促进GC与bitmap的重用, 你还有一些特定的事情可以做. 推荐的策略会根据Android的版本不同而有所差异. [BitmapFun](#)的示例程序会演示如何设计你的程序使得能够在不同的Android平台上高效的运行.

我们首先要知道Android管理bitmap memory的演变进程:

- 在Android 2.2 (API level 8)以及之前, 当GC发生时, 你的应用的线程是会stopped的. 这导致了一个滞后, 它会降低效率. 在**Android 2.3**上, 添加了并发GC的机制, 这意味着在一个**bitmap**不再被引用到之后, 内存会被立即**reclaimed**.
- 在Android 2.3.3 (API level 10)已经之后, 一个bitmap的像素级数据是存放在native内存中的. 这些数据与bitmap本身是隔离的, bitmap本身是被存放在Dalvik heap中. 在native内存中的pixel数据不是以可以预测的方式去释放的, 这意味着有可能导致一个程序容易超过它的内存限制并Crash. 在**Android 3.0 (API Level 11)**, pixel数据则是与**bitmap**本身一起存放在**dalvik heap**中.

下面会介绍如何在不同的Android版本上优化bitmap内存使用.

Manage Memory on Android 2.3.3 and Lower

在Android 2.3.3 (API level 10) 以及更低版本上, 推荐使用[recycle\(\)](#). 如果在你的程序中显示了大量的bitmap数据, 你很可能遇到OutOfMemoryError错误. recycle()方法可以使得程序尽快的reclaim memory. **Caution:** 只有你确保这个bitmap不再需要用的时候才应该使用recycle(). 如果你执行recycle(), 然后尝试绘画这个bitmap, 你将得到错误:"Canvas: trying to use a recycled bitmap".

下面的例子演示了使用recycle()的例子. 它使用了引用计数的方法(mDisplayRefCount 与 mCacheRefCount)来追踪一个bitmap目前是否有被显示或者是在缓存中. 当下面条件满足时回收bitmap:

- mDisplayRefCount 与 mCacheRefCount 的引用计数均为 0.
- bitmap不为null, 并且它还没有被回收.

```
private int mCacheRefCount = 0;
private int mDisplayRefCount = 0;
...
// Notify the drawable that the displayed state has changed.
// Keep a count to determine when the drawable is no longer displayed.
public void setIsDisplayed(boolean isDisplayed) {
    synchronized (this) {
        if (isDisplayed) {
            mDisplayRefCount++;
            mHasBeenDisplayed = true;
        } else {
            mDisplayRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

// Notify the drawable that the cache state has changed.
// Keep a count to determine when the drawable is no longer being cached.
public void setIsCached(boolean isCached) {
    synchronized (this) {
        if (isCached) {
            mCacheRefCount++;
        } else {
            mCacheRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

private synchronized void checkState() {
    // If the drawable cache and display ref counts = 0, and this drawable
    // has been displayed, then recycle.
    if (mCacheRefCount <= 0 && mDisplayRefCount <= 0 && mHasBeenDisplayed
        && hasValidBitmap()) {
        getBitmap().recycle();
    }
}

private synchronized boolean hasValidBitmap() {
    Bitmap bitmap = getBitmap();
    return bitmap != null && !bitmap.isRecycled();
}
```

Manage Memory on Android 3.0 and Higher

在Android 3.0 (API Level 11) 介绍了 [BitmapFactory.Options.inBitmap](#). 如果这个值被设置了, decode方法会在加载内容的时候去reuse已经存在的bitmap. 这意味着bitmap的内存是被reused的, 这样可以提升性能, 并且减少了内存的allocation与de-allocation. 在使用inBitmap时有几个注意点(caveats):

- reused的bitmap必须和原数据内容大小一致, 并且是JPEG 或者 PNG 的格式 (或者是某个resource 与 stream).
- reused的bitmap的[configuration](#)值如果有设置, 则会覆盖掉[inPreferredConfig](#)值.
- 你应该总是使用decode方法返回的bitmap, 因为你不可以假设reusing的bitmap是可用的(例如, 大小不对).

Save a bitmap for later use

下面演示了一个已经存在的bitmap是如何被存放起来以便后续使用的. 当一个应用运行在Android 3.0或者更高的平台上并且bitmap被从LruCache中移除时, bitmap的一个soft reference会被存放在HashSet中, 这样便于之后有可能被inBitmap进行reuse:

```
HashSet<SoftReference<Bitmap>> mReusableBitmaps;
private LruCache<String, BitmapDrawable> mMemoryCache;

// If you're running on Honeycomb or newer, create
// a HashSet of references to reusable bitmaps.
if (Utils.hasHoneycomb()) {
    mReusableBitmaps = new HashSet<SoftReference<Bitmap>>();
}

mMemoryCache = new LruCache<String, BitmapDrawable>(mCacheParams.memCacheSize) {

    // Notify the removed entry that is no longer being cached.
    @Override
    protected void entryRemoved(boolean evicted, String key,
        BitmapDrawable oldValue, BitmapDrawable newValue) {
        if (RecyclingBitmapDrawable.class.isInstance(oldValue)) {
            // The removed entry is a recycling drawable, so notify it
            // that it has been removed from the memory cache.
            ((RecyclingBitmapDrawable) oldValue).setIsCached(false);
        } else {
            // The removed entry is a standard BitmapDrawable.
            if (Utils.hasHoneycomb()) {
                // We're running on Honeycomb or later, so add the bitmap
                // to a SoftReference set for possible use with inBitmap later.
                mReusableBitmaps.add
                    (new SoftReference<Bitmap>(oldValue.getBitmap()));
            }
        }
    }
};
....
}
```

Use an existing bitmap

在运行的程序中, decoder方法会去做检查是否有可用的bitmap. 例如:

```
public static Bitmap decodeSampledBitmapFromFile(String filename,
    int reqWidth, int reqHeight, ImageCache cache) {

    final BitmapFactory.Options options = new BitmapFactory.Options();
    ...
    BitmapFactory.decodeFile(filename, options);
    ...

    // If we're running on Honeycomb or newer, try to use inBitmap.
    if (Utils.hasHoneycomb()) {
        addInBitmapOptions(options, cache);
    }
    ...
    return BitmapFactory.decodeFile(filename, options);
}
```

下面的代码演示了上面被执行的addInBitmapOptions()方法. 它会为inBitmap查找一个已经存在的bitmap设置为value. 注意这个方法只是去为inBitmap尝试寻找合适的值, 但是并不一定能够找到:

```
private static void addInBitmapOptions(BitmapFactory.Options options,
    ImageCache cache) {
    // inBitmap only works with mutable bitmaps, so force the decoder to
```

```

// return mutable bitmaps.
options.inMutable = true;

if (cache != null) {
    // Try to find a bitmap to use for inBitmap.
    Bitmap inBitmap = cache.getBitmapFromReusableSet(options);

    if (inBitmap != null) {
        // If a suitable bitmap has been found, set it as the value of
        // inBitmap.
        options.inBitmap = inBitmap;
    }
}

// This method iterates through the reusable bitmaps, looking for one
// to use for inBitmap:
protected Bitmap getBitmapFromReusableSet(BitmapFactory.Options options) {
    Bitmap bitmap = null;

    if (mReusableBitmaps != null && !mReusableBitmaps.isEmpty()) {
        final Iterator<SoftReference<Bitmap>> iterator
            = mReusableBitmaps.iterator();
        Bitmap item;

        while (iterator.hasNext()) {
            item = iterator.next().get();

            if (null != item && item.isMutable()) {
                // Check to see if the item can be used for inBitmap.
                if (canUseForInBitmap(item, options)) {
                    bitmap = item;

                    // Remove from reusable set so it can't be used again.
                    iterator.remove();
                    break;
                }
            } else {
                // Remove from the set if the reference has been cleared.
                iterator.remove();
            }
        }
    }
    return bitmap;
}

```

最后，下面这个方法去判断候选bitmap是否满足inBitmap的大小条件:

```

private static boolean canUseForInBitmap(
    Bitmap candidate, BitmapFactory.Options targetOptions) {
    int width = targetOptions.outWidth / targetOptions.inSampleSize;
    int height = targetOptions.outHeight / targetOptions.inSampleSize;

    // Returns true if "candidate" can be used for inBitmap re-use with
    // "targetOptions".
    return candidate.getWidth() == width && candidate.getHeight() == height;
}

```

编写:[kesenhoo](#)

校对:

在UI上显示Bitmap

这一课会演示如何运用前面几节课的内容，使用后台线程与Cache机制来加载图片到 ViewPager 与 GridView 组件，并且学习处理并发与配置改变问题。

Load Bitmaps into a ViewPager Implementation(实现加载图片到ViewPager)

[swipe view pattern](#)是一个用来切换显示不同详情界面的很好的方法。(关于这种效果请先参看[Android Design: Swipe Views](#)).

你可以通过 [PagerAdapter](#) 与 [ViewPager](#) 组件来实现这个效果. 然而, 一个更加合适的Adapter是PagerAdapter 的子类 [FragmentStatePagerAdapter](#):它可以在某个ViewPager中的子视图切换出屏幕时自动销毁与保存 Fragments 的状态。这样能够保持消耗更少的内存。

Note: 如果你只有为数不多的图片并且确保不会超出程序内存限制, 那么使用 PagerAdapter 或 FragmentPagerAdapter 会更加合适。

下面是一个使用ViewPager与ImageView作为子视图的示例。

```
public class ImageDetailActivity extends FragmentActivity {
    public static final String EXTRA_IMAGE = "extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new Integer[] {
        R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image_3,
        R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image_6,
        R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image_9
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_detail_pager); // Contains just a ViewPager

        mAdapter = new ImagePagerAdapter(getSupportFragmentManager(), imageResIds.length);
        mPager = (ViewPager) findViewById(R.id.pager);
        mPager.setAdapter(mAdapter);
    }

    public static class ImagePagerAdapter extends FragmentStatePagerAdapter {
        private final int mSize;

        public ImagePagerAdapter(FragmentManager fm, int size) {
            super(fm);
            mSize = size;
        }

        @Override
        public int getCount() {
            return mSize;
        }

        @Override
        public Fragment getItem(int position) {
            return ImageDetailFragment.newInstance(position);
        }
    }
}
```

Fragment 里面包含了ImageView 的子组件:

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA = "resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
    public ImageDetailFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```

        super.onCreate(savedInstanceState);
        mImageNum = getArguments() != null ? getArguments().getInt(IMAGE_DATA_EXTRA) : -1;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // image_detail_fragment.xml contains just an ImageView
        final View v = inflater.inflate(R.layout.image_detail_fragment, container, false);
        mImageView = (ImageView) v.findViewById(R.id.imageView);
        return v;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final int resId = ImageDetailActivity.imageResIds[mImageNum];
        mImageView.setImageResource(resId); // Load image into ImageView
    }
}

```

希望你发现上面示例存在的问题：在UI Thread中读取图片可能会导致程序ANR。使用在Lesson 2中学习的 AsyncTask 会比较好。

```

public class ImageDetailActivity extends FragmentActivity {
    ...

    public void loadBitmap(int resId, ImageView imageView) {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (ImageDetailActivity.class.isInstance(getActivity())) {
            final int resId = ImageDetailActivity.imageResIds[mImageNum];
            // Call out to ImageDetailActivity to load the bitmap in a background thread
            ((ImageDetailActivity) getActivity()).loadBitmap(resId, mImageView);
        }
    }
}

```

在 BitmapWorkerTask 中做一些例如resizing or fetching images from the network, 不会卡到UI Thread。如果后台线程不仅仅是做个简单的直接加载动作, 增加一个内存Cache或者磁盘Cache会比较好[参考Lesson 3], 下面是一些为了内存Cache而附加的内容:

```

public class ImageDetailActivity extends FragmentActivity {
    ...
    private LruCache mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        // initialize LruCache as per Use a Memory Cache section
    }

    public void loadBitmap(int resId, ImageView imageView) {
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            mImageView.setImageBitmap(bitmap);
        } else {
            mImageView.setImageResource(R.drawable.image_placeholder);
            BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }
}

```

```
}  
... // include updated BitmapWorkerTask from Use a Memory Cache section  
}
```

Load Bitmaps into a GridView Implementation(实现加载图片到GridView)

[Grid list building block](#) 是一种有效显示大量图片的方式。这样能够一次显示许多图片，而且那些即将被显示的图片也处于准备显示状态。如果你想要实现这种效果，你必须确保UI是流畅的，能够控制内存使用，并且正确的处理并发问题（因为GridView 会循环使用子视图）。

下面是一个在Fragment里面内置了ImageView作为GridView子视图的示例：

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListener {
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new Integer[] {
        R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image_3,
        R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image_6,
        R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image_9}

    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
        LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        final View v = inflater.inflate(R.layout.image_grid_fragment, container, false);
        final GridView mGridView = (GridView) v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView parent, View v, int position, long id) {
        final Intent i = new Intent(getActivity(), ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE, position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;

        public ImageAdapter(Context context) {
            super();
            mContext = context;
        }

        @Override
        public int getCount() {
            return imageResIds.length;
        }

        @Override
        public Object getItem(int position) {
            return imageResIds[position];
        }

        @Override
        public long getItemId(int position) {
            return position;
        }

        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ImageView imageView;
            if (convertView == null) { // if it's not recycled, initialize some attributes
                imageView = new ImageView(mContext);
                imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
                imageView.setLayoutParams(new GridView.LayoutParams(
                    LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT));
            } else {
```

```

        imageView = (ImageView) convertView;
    }
    //请注意下面的代码
    imageView.setImageResource(imageResIds[position]); // Load image into ImageView
    return imageView;
}
}

```

与前面加载到图片到ViewPager一样，如果setImageResource的操作会比较耗时，有可能会卡到UI Thread。可以使用类似前面异步处理图片与增加缓存的方法来解决那个问题。然而，我们还需要考虑GridView的循环机制所带来的并发问题。为了处理这个问题，请参考前面的课程。下面是一个更新的解决方案：

```

public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListener {
    ...

    private class ImageAdapter extends BaseAdapter {
        ...

        @Override
        public View getView(int position, View convertView, ViewGroup container) {
            ...
            loadBitmap(imageResIds[position], imageView)
            return imageView;
        }
    }

    public void loadBitmap(int resId, ImageView imageView) {
        if (cancelPotentialWork(resId, imageView)) {
            final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
            final AsyncDrawable asyncDrawable =
                new AsyncDrawable(getResources(), mPlaceholderBitmap, task);
            imageView.setImageDrawable(asyncDrawable);
            task.execute(resId);
        }
    }

    static class AsyncDrawable extends BitmapDrawable {
        private final WeakReference bitmapWorkerTaskReference;

        public AsyncDrawable(Resources res, Bitmap bitmap,
            BitmapWorkerTask bitmapWorkerTask) {
            super(res, bitmap);
            bitmapWorkerTaskReference =
                new WeakReference(bitmapWorkerTask);
        }

        public BitmapWorkerTask getBitmapWorkerTask() {
            return bitmapWorkerTaskReference.get();
        }
    }

    public static boolean cancelPotentialWork(int data, ImageView imageView) {
        final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

        if (bitmapWorkerTask != null) {
            final int bitmapData = bitmapWorkerTask.data;
            if (bitmapData != data) {
                // Cancel previous task
                bitmapWorkerTask.cancel(true);
            } else {
                // The same work is already in progress
                return false;
            }
        }
        // No task associated with the ImageView, or an existing task was cancelled
        return true;
    }

    private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
        if (imageView != null) {
            final Drawable drawable = imageView.getDrawable();
            if (drawable instanceof AsyncDrawable) {
                final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
                return asyncDrawable.getBitmapWorkerTask();
            }
        }
    }
}

```

```
    }  
    return null;  
}  
  
... // include updated BitmapWorkerTask class
```

Note:对于 ListView 同样可以套用上面的方法。

上面的方法提供了足够的弹性，使得你可以做从网络加载与Resize大的数码照片等操作而不至于卡到UI Thread。

编写:[jdneo](#)

校对:

使用OpenGL ES显示图像

Android框架提供了大量的标准工具，用来创建吸引人的，功能化的用户接口。然而，如果你希望对你的应用在屏幕上的绘图行为进行更多的控制，或者你在尝试建立三维图像，那么你就需要一个不同的工具了。由Android框架提供的OpenGL ES接口提供了显示高级动画图形的工具，它的功能仅仅受限于你自身的想象力，并且在许多Android设备上搭载的图形处理单元（GPU）都能为其提供GPU加速等性能优化。

这系列课程将教会你使用OpenGL搭建基本的应用，包括配置，绘制对象，移动图形单元及响应点击事件。

这系列课程所使用的样例代码使用的是OpenGL ES 2.0接口，这是当前Android设备所推荐的接口版本。关于跟多OpenGL ES的版本信息，可以阅读：[OpenGL](#)开发手册。

Note：注意不要把OpenGL ES 1.x版本的接口和OpenGL ES 2.0的接口混合调用。这两种版本的接口不是通用的。如果尝试混用它们，其输出结果可能会让你感到无奈和沮丧。

样例代码

[OpenGLS.zip](#)

编写:[jdneo](#)

校对:

建立OpenGL ES的环境

要在你的应用中使用OpenGL ES绘制图像，你必须为它们创建一个View容器。一个比较直接的方法是同时实现一个[GLSurfaceView](#)和一个[GLSurfaceView.Renderer](#)。[GLSurfaceView](#)是那些用OpenGL所绘制的图形的View容器，而[GLSurfaceView.Renderer](#)则用来控制在该View中绘制的内容。关于这两个类的更多信息，你可以阅读：[OpenGL ES开发手册](#)。

使用[GLSurfaceView](#)只是一种将你的应用与OpenGL ES合并起来的方法。对于一个全屏的或者接近全屏的图形View，使用它是一个理想的选择。开发者如果希望把OpenGL ES的图形融合在布局的一小部分里面，那么可以考虑使用[TextureView](#)。对于自己动手开发的开发者来说（DIY），还可以通过使用[SurfaceView](#)来搭建一个OpenGL ES View，但这将需要编写更多的代码。

在这节课中，我们将解释如何在一个简单地应用activity中完成[GLSurfaceView](#)和[GLSurfaceView.Renderer](#)的最小实现。

在配置文件中声明使用OpenGL ES

为了让你的应用能够使用OpenGL ES 2.0接口，你必须将下列声明添加到配置文件当中：

```
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

如果你的应用使用纹理压缩（texture compression），那么你必须对你支持的压缩格式也进行声明，这样的话那些不支持这些格式的设备就不会尝试运行你的应用：

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />  
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

更多关于纹理压缩的内容，可以阅读：[OpenGL开发手册](#)。

为OpenGL ES图形创建一个activity

使用OpenGL ES的安卓应用就像其它类型的应用有自己的用户接口一样，也拥有多个activity。主要的区别就在于activity布局上的不同。在许多应用中你可能会使用[TextView](#)，[Button](#)和[ListView](#)，在使用OpenGL ES的应用中，你也需要添加一个[GLSurfaceView](#)。

下面的代码展示了使用一个[GLSurfaceView](#)的最小化实现。它作为主View：

```
public class OpenGL20Activity extends Activity {  
  
    private GLSurfaceView mGLView;  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Create a GLSurfaceView instance and set it  
        // as the ContentView for this Activity.  
        mGLView = new MyGLSurfaceView(this);  
        setContentView(mGLView);  
    }  
}
```

Note : OpenGL ES 2.0需要Android 2.2（API Level 8）或更高版本的系统，所以确保你的Android项目的API版本满足该要求。

构建一个GLSurfaceView对象

一个[GLSurfaceView](#)是一个特定的View，在View中你可以绘制OpenGL ES图形。不过它自己所做的事情并不多。对于绘制对象的控制实际上是由你在该View中配置的[GLSurfaceView.Renderer](#)<http://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer.html>所负责的。事实上，这个对象的代码非常简短，你可能会希望跳过继承它，并且只创建一个未经修改的GLSurfaceView实例，不过请不要这么做。你需要继承该类来捕捉触控事件，这方面知识在[Responding to Touch Events](#)（该系列课程的最后一节课）中会做进一步的介绍。

[GLSurfaceView](#)的核心代码是很小的，所以对于一个快速地实现，通常可以在activity中创建一个内部类并使用它：

```
class MyGLSurfaceView extends GLSurfaceView {  
  
    public MyGLSurfaceView(Context context){  
        super(context);  
  
        // Set the Renderer for drawing on the GLSurfaceView  
        setRenderer(new MyRenderer());  
    }  
}
```

当使用OpenGL ES 2.0时，你必须对你的[GLSurfaceView](#)构造函数添加另一个调用，以此来指明你希望使用的是2.0版本的接口：

```
// Create an OpenGL ES 2.0 context  
setEGLContextClientVersion(2);
```

Note：如果你在使用OpenGL ES 2.0版本的接口，确保在你的应用配置文件中也进行了相关声明。这在之前的章节中已经讨论过了。

另一个对于[GLSurfaceView](#)实现的可选选项，是将渲染模式设置为：[GLSurfaceView.RENDERMODE_WHEN_DIRTY](#)，其含义是：仅在你的绘画数据发生变化时才在视图中进行绘画操作：

```
// Render the view only when there is a change in the drawing data  
setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
```

这一配置将防止[GLSurfaceView](#)框架被重新绘制，直到你调用了[requestRender\(\)](#)，这将让应用的性能及效率得到提高。

构建一个渲染类

[GLSurfaceView.Renderer](#)类的实现，或者说在一个应用中使用OpenGL ES来进行渲染，正是事情变得有趣的地方。该类会控制和其相关联的[GLSurfaceView](#)，决定在上面画什么。一共有三个渲染器的方法被Android系统调用，以此来明确要在[GLSurfaceView](#)上画什么以及如何画：

- [onSurfaceCreated\(\)](#)：调用一次，用来配置视图的OpenGL ES环境。
- [onDrawFrame\(\)](#)：每次重画视图时被调用。
- [onSurfaceChanged\(\)](#)：如果视图的几何形态发生变化时会被调用，例如当设备的屏幕方向发生改变时。

下面是一个非常基本的OpenGL ES渲染器的实现，作用仅仅是在[GLSurfaceView](#)中画一个灰色的背景：

```
public class MyGLRenderer implements GLSurfaceView.Renderer {  
  
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
        // Set the background frame color  
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
    }  
  
    public void onDrawFrame(GL10 unused) {  
        // Redraw background color  
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);  
    }  
  
    public void onSurfaceChanged(GL10 unused, int width, int height) {  
        GLES20.glViewport(0, 0, width, height);  
    }  
}
```

就仅仅是这样！上面的代码创建了一个简单地应用程序，它在使用OpenGL显示一个灰色的屏幕。虽然它的代码做的事情并不怎么有趣，但是通过创建这些类，你已经为使用OpenGL绘制图形有了基本的认识和铺垫。

Note：你可能想知道为什么这些方法有一个[GL10](#)的参数，在你明明使用的是OpenGL ES 2.0的接口的时候。这是因为这些方法在2.0接口中被简单地重用了，以此来保持Android框架尽量简单。

如果你对OpenGL ES接口很熟悉，那么你现在就可以在你的应用中部署一个OpenGL ES的环境并绘制图形。然而，如果你希望获取更多的帮助来学会使用OpenGL，那么请继续学习下一节课程获取更多的知识。

编写:[jdneo](#)

校对:

定义 Shapes

在一个OpenGL ES视图的上下文中定义形状，是创建你的杰作所需要的第一步。在不知道关于OpenGL ES如何期望你来定义图形对象的基本知识的时候，通过OpenGL ES 绘图可能会有些困难。

这节课将解释OpenGL ES相对于Android设备屏幕的坐标系，定义形状和形状表面的基本知识，如定义一个三角形和一个矩形。

定义一个三角形

OpenGL ES允许你使用三维空间的坐标来定义绘画对象。所以在你能画三角形之前，你必须先定义它的坐标。在OpenGL中，典型的办法是以浮点数的形式为坐标定义一个顶点数组。为了让效率最大化，你可以将坐标写入一个[ByteBuffer](#)，它将会传入OpenGL ES的图形处理流程中：

```
public class Triangle {

    private FloatBuffer vertexBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float triangleCoords[] = { // in counterclockwise order:
        0.0f,  0.622008459f, 0.0f, // top
        -0.5f, -0.311004243f, 0.0f, // bottom left
        0.5f, -0.311004243f, 0.0f  // bottom right
    };

    // Set color with red, green, blue and alpha (opacity) values
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };

    public Triangle() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
            // (number of coordinate values * 4 bytes per float)
            triangleCoords.length * 4);
        // use the device hardware's native byte order
        bb.order(ByteOrder.nativeOrder());

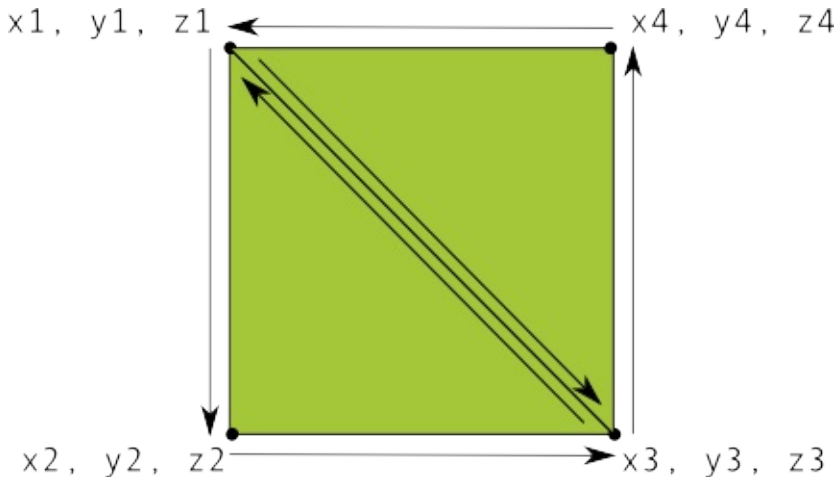
        // create a floating point buffer from the ByteBuffer
        vertexBuffer = bb.asFloatBuffer();
        // add the coordinates to the FloatBuffer
        vertexBuffer.put(triangleCoords);
        // set the buffer to read the first coordinate
        vertexBuffer.position(0);
    }
}
```

默认情况下，OpenGL ES会假定一个坐标系，在这个坐标系中，[0, 0, 0] (X, Y, Z) 对应的是GLSurfaceView框架的中心。[1, 1, 0]对应的是框架的右上角，[-1, -1, 0]对应的则是左下角。如果想要看此坐标系的插图说明，可以阅读[OpenGL ES](#)开发手册。

注意到这个形状的坐标是以逆时针顺序定义的。绘制的顺序非常关键，因为它定义了哪一面是形状的正面（你希望绘制的一面），以及背面（你可以使用OpenGL ES的cull face特性来让它不要绘制）。更多关于该方面的信息，可以阅读[OpenGL ES](#)开发手册。

定义一个矩形

在OpenGL中定义三角形非常简单，那么你是否想要增加一些复杂性呢？比如，定义一个矩形？有很多方法可以用来定义矩形，不过在OpenGL ES中最典型的办法是使用两个三角形拼接在一起：



再一次地，你需要以逆时针的形式为三角形顶点定义坐标来表现这个图形，并将值放入一个[ByteBuffer](#)中。为了避免由两个三角形共享的顶点被重复定义，可以使用一个绘制列表来告诉OpenGL ES图形处理流程应该如何画这些顶点。下面是代码样例：

```
public class Square {

    private FloatBuffer vertexBuffer;
    private ShortBuffer drawListBuffer;

    // number of coordinates per vertex in this array
    static final int COORDS_PER_VERTEX = 3;
    static float squareCoords[] = {
        -0.5f,  0.5f, 0.0f,    // top left
        -0.5f, -0.5f, 0.0f,   // bottom left
        0.5f,  -0.5f, 0.0f,   // bottom right
        0.5f,  0.5f, 0.0f    // top right
    };

    private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; // order to draw vertices

    public Square() {
        // initialize vertex byte buffer for shape coordinates
        ByteBuffer bb = ByteBuffer.allocateDirect(
            // (# of coordinate values * 4 bytes per float)
            squareCoords.length * 4);
        bb.order(ByteOrder.nativeOrder());
        vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(squareCoords);
        vertexBuffer.position(0);

        // initialize byte buffer for the draw list
        ByteBuffer dlb = ByteBuffer.allocateDirect(
            // (# of coordinate values * 2 bytes per short)
            drawOrder.length * 2);
        dlb.order(ByteOrder.nativeOrder());
        drawListBuffer = dlb.asShortBuffer();
        drawListBuffer.put(drawOrder);
        drawListBuffer.position(0);
    }
}
```

该样例给了你一个如何使用OpenGL创建复杂图形的启发，通常来说，你需要使用三角形的集合来绘制对象。在下一节课中，你将学习如何在屏幕上画这些形状。

编写:[jdneo](#)

校对:

绘制Shapes

在你定义了需要OpenGL绘制的形状之后，你可能希望绘制它们。使用OpenGL ES 2.0绘制图形可能会比你想象当中花费更多的代码，因为API中提供了大量对于图形处理流程的控制。

这节课将解释如何使用OpenGL ES 2.0接口画出在上一节课中定义的图形。

初始化形状

在你开始绘画之前，你需要初始化并加载你期望绘制的图形。除非你所使用的形状结构（原始坐标）在执行过程中发生了变化，不然的话你应该在渲染器的[onSurfaceCreated\(\)](#)方法中初始化它们，这样做是处于内存和执行效率的考量。

```
public void onSurfaceCreated(GL10 unused, EGLConfig config) {
    ...

    // initialize a triangle
    mTriangle = new Triangle();
    // initialize a square
    mSquare = new Square();
}
```

画一个形状

使用OpenGL ES 2.0画一个定义的形状需要较多代码，因为你需要提供很多图形处理流程的细节。具体而言，你必须定义如下几项：

- 顶点着色器（Vertex Shader）：OpenGL ES代码用来渲染形状的顶点。
- 片段着色器（Fragment Shader）：OpenGL ES代码用来渲染形状的表面，使用颜色或纹理。
- 程式（Program）：一个OpenGL ES对象，包含了你希望用来绘制一个活更多图形所要用到的着色器。

你需要至少一个顶点着色器来绘制一个形状，以及一个片段着色器为该形状上色。这些着色器必须被编译然后添加至一个OpenGL ES程式当中，它用来绘制形状。下面的代码展示了你可以用来画一个图形的基本着色器：

```
private final String vertexShaderCode =
    "attribute vec4 vPosition;" +
    "void main() {" +
    "    gl_Position = vPosition;" +
    "}";

private final String fragmentShaderCode =
    "precision mediump float;" +
    "uniform vec4 vColor;" +
    "void main() {" +
    "    gl_FragColor = vColor;" +
    "}";
```

着色器包含了OpenGL Shading Language（GLSL）代码，它必须先被编译然后才能在OpenGL环境中使用。要编译这个代码，在你的渲染器类中创建一个辅助方法：

```
public static int loadShader(int type, String shaderCode){

    // create a vertex shader type (GL_VERTEX_SHADER)
    // or a fragment shader type (GL_FRAGMENT_SHADER)
    int shader = GLES20.glCreateShader(type);

    // add the source code to the shader and compile it
    GLES20.glShaderSource(shader, shaderCode);
    GLES20.glCompileShader(shader);

    return shader;
}
```

为了绘制你的图形，你必须编译着色器代码，将它们添加至一个OpenGL ES程式对象中，然后执行连接。在你的绘制对象的构造函数里做这些事情，这样上述步骤就只用执行一次。

Note：编译OpenGL ES着色器及连接操作对于CPU周期和处理时间而言，消耗是巨大的，所以你应该避免重复执行这些事情。如果你在执行期间不知道你的着色器内容，那么你应该在构建你的应用时，确保它们只创建了一次，并且缓存以备后续使用。

```
public class Triangle() {
    ...

    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexShaderCode);
    int fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentShaderCode);

    mProgram = GLES20.glCreateProgram(); // create empty OpenGL ES Program
```

```

    GLES20.glAttachShader(mProgram, vertexShader); // add the vertex shader to program
    GLES20.glAttachShader(mProgram, fragmentShader); // add the fragment shader to program
    GLES20.glLinkProgram(mProgram); // creates OpenGL ES program executable
}

```

至此，你已经完全准备好添加实际的调用来绘制你的图形了。使用OpenGL ES绘制图形需要你定义一些变量来告诉渲染流程你需要画什么以及如何去画。既然绘制属性会根据形状的不同而发生变化，把绘制逻辑包含在形状类里面将是一个不错的主意。

为图形创建一个“draw()”方法。这个代码为形状的顶点着色器和形状着色器设置了位置和颜色值，进而执行绘图功能：

```

public void draw() {
    // Add program to OpenGL ES environment
    GLES20.glUseProgram(mProgram);

    // get handle to vertex shader's vPosition member
    mPositionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");

    // Enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(mPositionHandle);

    // Prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
                                GLES20.GL_FLOAT, false,
                                vertexStride, vertexBuffer);

    // get handle to fragment shader's vColor member
    mColorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");

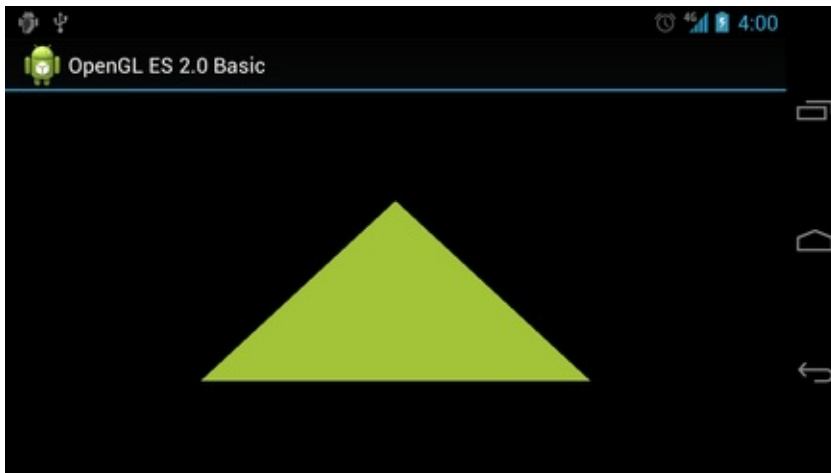
    // Set color for drawing the triangle
    GLES20.glUniform4fv(mColorHandle, 1, color, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // Disable vertex array
    GLES20.glDisableVertexAttribArray(mPositionHandle);
}

```

一旦你完成了上述所有代码，画这个对象就仅需要在你渲染器的[onDrawFrame\(\)](#)方法中调用draw()方法就可以了。当你运行这个应用时，它看上去会像是这样：



在这个代码样例中，还存在一些问题。首先，它无法给你的朋友带来什么深刻的印象。其次，这个三角形看上去有一些扁，另外当你改变屏幕方向时，它的形状也会随之改变。形状发生形变的原因是因为对象的顶点没有根据显示[GLSurfaceView](#)的屏幕区域的比例进行修正。你可以在下一节课中使用投影（projection）或者相机视图（camera view）来解决这个问题。

最后，这个三角形是静止的，这看上去有些无聊。在[Adding Motion](#)课程当中（后续课程），你会让这个形状发生旋转，并使用一些OpenGL ES图形处理流程的更加新奇的用法。

编写:[jdneo](#)

校对:

运用投影与相机视图

在OpenGL ES环境中，投影和相机视图允许你显示绘图对象时，可以以一个更加酷似于你用肉眼看到的真实物体。这个物理视图的仿真是使用绘制对象坐标的数学变换实现的：

- 投影（**Projection**）：这个变换会基于显示它们的[GLSurfaceView](#)的长和宽，来调整绘图对象的坐标。如果没有该计算，那么用OpenGL ES绘制的对象会由于视图窗口比例的不匹配而发生形变。一个投影变换一般仅需要在渲染器的[onSurfaceChanged\(\)](#)方法中，OpenGL视图的比例建立时或发生变化时才被计算。关于更多OpenGL ES投影和坐标映射的知识，可以阅读[Mapping Coordinates for Drawn Objects](#)。
- 相机视图（**camera view**）：这个变化会基于一个虚拟相机位置改变绘图对象的坐标。注意到OpenGL ES并没有定义一个实际的相机对象，但是取而代之的，它提供了一些辅助方法，通过变化绘图对象的显示来模拟相机。一个相机视图变换可能仅在你建立你的[GLSurfaceView](#)时计算一次，也可能根据用户的行为或者你的应用的功能进行动态调整。

这节课将解释如何创建一个投影和一个相机视图，并应用它们到[GLSurfaceView](#)中的绘制图像上。

定义一个投影

投影变换的数据会在你的[GLSurfaceView.Renderer](#)类中的[onSurfaceChanged\(\)](#)方法中被计算。下面的代码首先接收[GLSurfaceView](#)的高和宽，然后用它来填充一个投影变换矩阵（[Matrix](#)），使用[Matrix.frustumM\(\)](#)方法：

```
@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0, 0, width, height);

    float ratio = (float) width / height;

    // this projection matrix is applied to object coordinates
    // in the onDrawFrame() method
    Matrix.frustumM(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
```

该代码填充了一个投影矩阵：mProjectionMatrix，在下一节中，你可以在[onDrawFrame\(\)](#)将它和一个相机视图变换结合起来。

Note：在你的绘图对象上只应用一个投影变换会导致一个看上去很空的显示效果。一般而言，你必须同时为每一个要在屏幕上显示的任何东西实现一个相机视图。

定义一个相机视图

通过添加一个相机视图变换作为绘图过程的一部分，以此来完成你的绘图对象变换的所有步骤。在下面的代码中，使用[Matrix.setLookAtM\(\)](#)方法来计算相机视图变换，然后与之前计算的投影矩阵结合起来。结合后的变换矩阵传递给绘制图像：

```
@Override
public void onDrawFrame(GL10 unused) {
    ...
    // Set the camera position (View matrix)
    Matrix.setLookAtM(mViewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0, mViewMatrix, 0);

    // Draw shape
    mTriangle.draw(mMVPMatrix);
}
```

应用投影和相机变换

为了使用在之前章节中结合了的相机视图变换和投影变换，修改你的图形对象的draw()方法，接收结合的变换并将其应用到图形上：

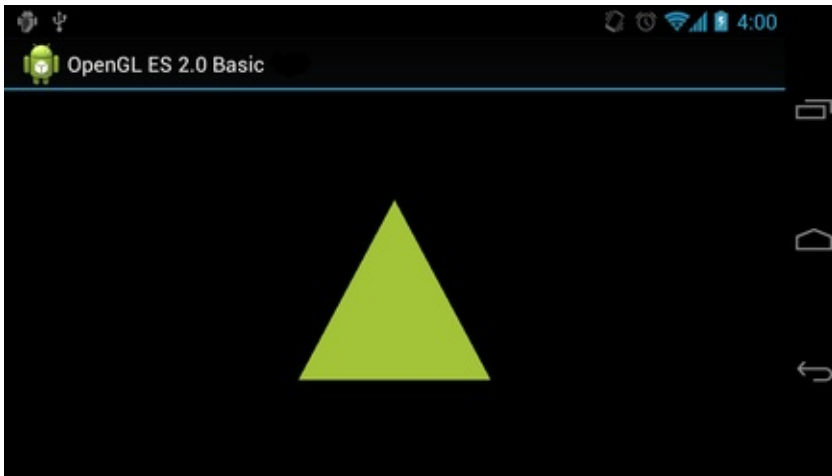
```
public void draw(float[] mvpMatrix) { // pass in the calculated transformation matrix
    ...

    // get handle to shape's transformation matrix
    mMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

    // Pass the projection and view transformation to the shader
    GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);

    // Draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);
    ...
}
```

一旦你正确地计算了投影变换和相机视图变换，并应用了它们，你的图形就会以正确的比例画出，看上去会像是这样：



现在你有了一个能以正确的比例显示你的图形的应用了，下面就该为图形添加一些动画效果了！

编写:[jdneo](#)

校对:

添加移动

在屏幕上绘制图形是OpenGL的一个基本特性，但你也可以通过其它的Android图形框架类做这些事情，包括[Canvas](#)和[Drawable](#)对象。OpenGL ES提供额外的功能，能够在三维空间对绘制图形进行移动和变换操作，或者还可以通过其它独有的方法创建出引人入胜的用户体验。

在这节课中，一会更深入的学习OpenGL ES的知识：对一个形状添加旋转动画。

旋转一个形状

使用OpenGL ES 2.0 旋转一个绘制图形是比较简单的。首先创建一个变换矩阵（一个旋转矩阵）并且将它和你的投影变换矩阵和相机视图变换矩阵结合在一起：

```
private float[] mRotationMatrix = new float[16];
public void onDrawFrame(GL10 gl) {
    ...
    float[] scratch = new float[16];

    // Create a rotation transformation for the triangle
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, angle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera view
    // Note that the mMVPMatrix factor *must* be first* in order
    // for the matrix multiplication product to be correct.
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);

    // Draw triangle
    mTriangle.draw(scratch);
}
```

如果完成了这些变更以后，你的三角形还是没有旋转的话，确认一下你是否将[GLSurfaceView.RENDERMODE_WHEN_DIRTY](#)的配置注释掉了，有关该方面的知识会在下一节课展开。

启用连续渲染

如果你严格按照这节课的样例代码走到了现在这一步，那么请确定您将设置渲染模式为“RENDERMODE_WHEN_DIRTY”的这行注释了，不然的话OpenGL只会对这个形状执行一个增量的旋转，然后就等待[GLSurfaceView](#)容器的[requestRender\(\)](#)方法被调用。

```
public MyGLSurfaceView(Context context) {  
    ...  
    // Render the view only when there is a change in the drawing data.  
    // To allow the triangle to rotate automatically, this line is commented out:  
    //setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);  
}
```

除非你的某个对象，它的变化和用户的交互无关，不然的话一般还是建议将这个配置打开。在下一节课将会将这个注释放开，并且再次调用。

编写:[jdneo](#)

校对:

响应触摸事件

让对象根据预设的程序运动，如让一个三角形旋转可以有效地让人引起注意，但是如果你希望可以`OpenGL ES`与用户交互呢？让你的`OpenGL ES`应用可以与触摸交互的关键点在于，拓展你的[GLSurfaceView](#)的实现，覆写[onTouchEvent\(\)](#)方法来监听触摸事件。

这节课将会向你展示如何监听触摸事件，让用户旋转一个`OpenGL ES`对象。

配置触摸监听器

为了让你的OpenGL ES应用响应触摸事件，你必须实现在[GLSurfaceView](#)类中的[onTouchEvent\(\)](#)方法。下述实现的样例展示了如何监听[MotionEvent.ACTION_MOVE](#)事件，并将它们转换为形状旋转的角度：

```
@Override
public boolean onTouchEvent(MotionEvent e) {
    // MotionEvent reports input details from the touch screen
    // and other input controls. In this case, you are only
    // interested in events where the touch position changed.

    float x = e.getX();
    float y = e.getY();

    switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:

            float dx = x - mPreviousX;
            float dy = y - mPreviousY;

            // reverse direction of rotation above the mid-line
            if (y > getHeight() / 2) {
                dx = dx * -1;
            }

            // reverse direction of rotation to left of the mid-line
            if (x < getWidth() / 2) {
                dy = dy * -1;
            }

            mRenderer.setAngle(
                mRenderer.getAngle() +
                ((dx + dy) * TOUCH_SCALE_FACTOR); // = 180.0f / 320
            requestRender();

        }

    mPreviousX = x;
    mPreviousY = y;
    return true;
}
```

注意在计算旋转角度后，该方法会调用[requestRender\(\)](#)来告诉渲染器现在可以进行渲染了。该方法对于这个例子来说是最有效的，因为图形并不需要重新绘制，除非有一个旋转角度的变化。然而，它对于执行效率并没有任何影响，除非你需要渲染器仅在数据变化时才会重新绘制（使用[setRenderMode\(\)](#)方法），所以请确保这一行没有被注释掉：

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing data
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}
```

公开旋转角度

上述样例代码需要你公开旋转的角度，方法是在你的渲染器中添加一个共有成员。由于渲染器代码运行在一个独立的线程中（非主UI线程），你必须将你的这个公共变量声明为`volatile`。请看下面的代码：

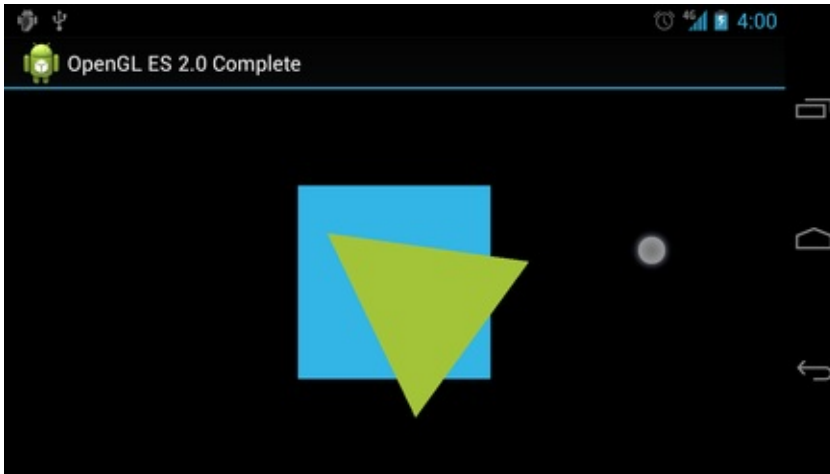
```
public class MyGLRenderer implements GLSurfaceView.Renderer {  
    ...  
    public volatile float mAngle;
```

应用旋转

为了应用触摸输入所导致的旋转，注释掉创建一个旋转角度的代码，然后添加mAngle，该变量包含了输入所导致的角度：

```
public void onDrawFrame(GL10 gl) {  
    ...  
    float[] scratch = new float[16];  
  
    // Create a rotation for the triangle  
    // long time = SystemClock.uptimeMillis() % 4000L;  
    // float angle = 0.090f * ((int) time);  
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, -1.0f);  
  
    // Combine the rotation matrix with the projection and camera view  
    // Note that the mMVPMatrix factor *must be first* in order  
    // for the matrix multiplication product to be correct.  
    Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);  
  
    // Draw triangle  
    mTriangle.draw(scratch);  
}
```

当你完成了上述的步骤，运行这个程序并用你的手指在屏幕上拖动，来旋转三角形：



编写:[lltowq](#)

校对:(未验证)

动画

课程

淡入淡出两个View

学习怎样实现重叠视图交叉淡入。这节课将展示怎样交叉淡入一个一个进度指示器对于一个视图包含文本信息???

使用ViewPager实现滑屏

学习怎样实现水平相邻屏幕滑动转换的动画

显示翻转动画

学习怎样实现两个视图的翻转动画

视图缩放

学习怎样通过触摸缩放动画实现放大视图

动画布局的改变

学习怎样使用内置动画布局当你增加、移除或更新子视图

编写:[lltowq](#)

校对:(未验证)

淡入淡出两个View

淡入淡出动画（也做溶解），渐变淡出某个UI组件同时淡出另一个。这个动画是有用，当你app想交换文本或视图的情况。渐变非常微妙和简短但是提供流动转换从一个屏幕到下一个。当你不使用它们，不管怎么样转换经常感到生硬或匆忙。

下面有一个例子关于进度指示器渐变到一些文本内容。

编写:[wangyan3](#)

校对:

使用**ViewPager**实现屏幕滑动

滑屏是在两个屏幕的装换，是普通的UI设置导向和幻灯放映。这节课将展示怎样实现滑屏通过一个[ViewPager](#)提供的[[support library](#)].[ViewPager](#)能自动实现活泼的滑屏。下面一个滑屏看起来下个从一个屏幕到内容到一下的转换。

编写:[lltowq](#)

校对:(未验证)

卡片翻转的动画

这节课展示怎样实现定制动画片段和卡片翻转动画。卡片翻转动画在文本视图间通过显示动画模拟卡片翻转。

下面是卡片翻转动画样子

编写:[lltowq](#)

校对:(未验证)

缩放动画

这节课案例是怎样实现触摸缩放动画，这对相册app实现相片视图大小控制很有用。

下面触摸缩放动画看起来扩大图片填充屏幕缩略图。

编写:[lltowq](#)

校对:(未验证)

控件切换动画

动画的布局是加载动画的前提，系统运行你每次改变的布局配置。你需要在布局文件里设置属性告诉Android系统动画布局的改变，系统默认动画是执行你的。

小点：如果你想提供定制动画布局，创建[LayoutTransition](#)对象和提供它的[setLayoutTransition](#)布局方法。

下面是默认布局动画想起来像是添加项目列表

编写:

校对:

连接

编写:[acenodie](#)

校对:

无线连接设备

除了能够在云端通信，Android的无线接口（wireless APIs）也允许同一局域网中的设备进行通信，甚至没有连接到网络上，而是物理上隔得很近，也可以相互通信。此外，网络服务发现（Network Service Discovery，简称NSD）可以进一步通过允许应用程序运行能相互通信的服务去寻找附近运行相同服务的设备。把这个功能整合到你的应用中可以提供范围广泛的特点，如在同一个房间，用户玩游戏，可以利用NSD实现从一个网络摄像头获取图像，或远程登录到在同一网络中的其他机器。

本节课介绍了一些使你的应用程序能够寻找和连接其他设备的主要的接口（APIs）。具体地说，它介绍了用于发现可用服务的NSD API和能实现点对点无线连接的无线点对点（the Wi-Fi Peer-to-Peer，简称Wi-Fi P2P）API。本节课也将告诉我们怎样将NSD和Wi-Fi P2P结合起来去检测其他设备所提供的服务，当检测到时，连接到相应的设备上。

课程

[使得网络服务可发现](#)

学习如何广播由你自己的应用程序提供的服务，如何发现在本地网络上提供的服务并用NSD获取你将要连接的服务的详细信息。

[使用WiFi建立P2P连接](#)

学习如何获取附近附近的对等设备，如何创建一个设备接入点，如何连接到其他具有Wi-Fi P2P连接功能的设备。

[使用WiFi P2P服务](#)

学习如何使用WiFi P2P服务去发现附近的不在同一个网络的服务。

编写:

校对:

使得网络服务可发现

编写:

校对:

使用**WiFi**建立**P2P**连接

编写:

校对:

使用**WiFi P2P**服务

编写:[kesenhoo](#)

校对:

网络连接操作

这一章会介绍一些基本的网络操作，监视网络链接（包括网络改变），让用户来控制app对网络的选择使用。还会介绍如何解析与使用XML数据。

[NetworkUsage.zip](#)

通过学习这章节的课程，你已经会学习一些基础知识，如何在最小化网络阻塞的情况下，创建一个高效的app，用来下载数据与解析数据。

你还可以参考下面文章进阶学习：

- [Optimizing Battery Life](#)
- [Transferring Data Without Draining the Battery](#)
- [Web Apps Overview](#)

编写:[kesenhoo](#)

校对:

连接到网络Connecting to the Network

这一课会演示如何实现一个简单的连接到网络的程序。它提供了一些你应该follow的最好示例，用来创建最简单的网络连接程序。请注意，想要执行网络操作首先需要在程序的manifest文件中添加下面的permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```


Choose an HTTP Client(选择一个HTTP Client)

大多数连接网络的Android app会使用HTTP来发送与接受数据。Android提供了两种HTTP clients: [HttpURLConnection](#) 与Apache [HttpClient](#)。他们二者均支持HTTPS，都以流方式进行上传与下载，都有可配置timeout, IPv6 与连接池（connection pooling）。推荐从Gingerbread版本开始使用 **HttpURLConnection**。关于这部分的更多详情，请参考 [Android's HTTP Clients](#)。

Check the Network Connection(检测网络连接)

在你的app尝试进行网络连接之前，需要检测当前是否有可用的网络。请注意，设备可能会不在网络覆盖范围内，或者用户可能关闭Wi-Fi与移动网络连接。

```
public void myClickHandler(View view) {  
    ...  
    ConnectivityManager connMgr = (ConnectivityManager)  
        getSystemService(Context.CONNECTIVITY_SERVICE);  
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();  
    if (networkInfo != null && networkInfo.isConnected()) {  
        // fetch data  
    } else {  
        // display error  
    }  
    ...  
}
```

Perform Network Operations on a Separate Thread(在另外一个Thread执行网络操作)

网络操作会遇到不可预期的延迟。显然为了避免一个不好的用户体验，总是在UI Thread之外去执行网络操作。AsyncTask 类提供了一种简单的方式来处理这个问题。关于更多的详情，请参考：[Multithreading For Performance](#)。

在下面的代码示例中，myClickHandler() 方法会触发一个新的DownloadWebpageTask().execute(stringUrl)。它继承自AsyncTask，实现了下面两个方法：

- [doInBackground\(\)](#) 执行 downloadUrl()方法。Web URL作为参数，方法downloadUrl() 获取并处理网页返回的数据，执行完毕后，传递结果到onPostExecute()。参数类型为String。
- [onPostExecute\(\)](#) 获取到返回数据并显示到UI上。

```
public class HttpExampleActivity extends Activity {
    private static final String DEBUG_TAG = "HttpExample";
    private EditText urlText;
    private TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        urlText = (EditText) findViewById(R.id.myUrl);
        textView = (TextView) findViewById(R.id.myText);
    }

    // When user clicks button, calls AsyncTask.
    // Before attempting to fetch the URL, makes sure that there is a network connection.
    public void myClickHandler(View view) {
        // Gets the URL from the UI's text field.
        String urlString = urlText.getText().toString();
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
        if (networkInfo != null && networkInfo.isConnected()) {
            new DownloadWebpageText().execute(stringUrl);
        } else {
            textView.setText("No network connection available.");
        }
    }

    // Uses AsyncTask to create a task away from the main UI thread. This task takes a
    // URL string and uses it to create an HttpURLConnection. Once the connection
    // has been established, the AsyncTask downloads the contents of the webpage as
    // an InputStream. Finally, the InputStream is converted into a string, which is
    // displayed in the UI by the AsyncTask's onPostExecute method.
    private class DownloadWebpageText extends AsyncTask {
        @Override
        protected String doInBackground(String... urls) {

            // params comes from the execute() call: params[0] is the url.
            try {
                return downloadUrl(urls[0]);
            } catch (IOException e) {
                return "Unable to retrieve web page. URL may be invalid.";
            }
        }

        // onPostExecute displays the results of the AsyncTask.
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result);
        }
    }
    ...
}
```

关于上面那段代码的示例详解，请参考下面：

- When users click the button that invokes myClickHandler(), the app passes the specified URL to the AsyncTask subclass DownloadWebpageTask.
- The AsyncTask method doInBackground() calls the downloadUrl() method.
- The downloadUrl() method takes a URL string as a parameter and uses it to create a URL object.
- The URL object is used to establish an HttpURLConnection.
- Once the connection has been established, the HttpURLConnection object fetches the web page content as an InputStream.
- The InputStream is passed to the readIt() method, which converts the stream to a string.
- Finally, the AsyncTask's onPostExecute() method displays the string in the main activity's UI.

Connect and Download Data(连接并下载数据)

在执行网络交互的线程里面，你可以使用 `HttpURLConnection` 来执行一个 GET 类型的操作并下载数据。在你调用 `connect()` 之后，你可以通过调用 `getInputStream()` 来得到一个包含数据的 [InputStream](#) 对象。

在下面的代码示例中，`doInBackground()` 方法会调用 `downloadUrl()`。这个 `downloadUrl()` 方法使用给予的 URL，通过 `HttpURLConnection` 连接到网络。一旦建立连接，app 使用 `getInputStream()` 来获取数据。

```
// Given a URL, establishes an HttpURLConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

        // Makes sure that the InputStream is closed after the app is
        // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

请注意，`getResponseCode()` 会返回连接状态码(status code)。这是一种获知额外网络连接信息的有效方式。status code 是 200 则意味着连接成功。

Convert the InputStream to a String(把InputStream的数据转换为String)

InputStream 是一种可读的byte数据源。如果你获得了一个 InputStream, 通常会进行decode或者转换为制定的数据类型。例如, 如果你是在下载一张image数据, 你可能需要像下面一下进行decode :

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

在上面演示的示例中, InputStream 包含的是web页面的文本内容。下面会演示如何把 InputStream 转换为string, 以便显示在UI上。

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

编写:[kesenhoo](#)

校对:

管理使用的网络Managing Network Usage

这一课会介绍如何细化管理使用的网络资源。如果你的程序需要执行很多网络操作，你应该提供用户设置选项来允许用户控制程序的数据偏好。例如，同步数据的频率，是否只在连接到WiFi才进行下载与上传操作，是否在漫游时使用套餐数据流量等等。这样用户才能在快到达流量上限时关闭你的程序获取数据功能。

关于如何编写一个最小化下载与网络操作对电量影响的程序，请参考：

- [Optimizing Battery Life:](#)
- [Transferring Data Without Draining the Battery:](#)

Check a Device's Network Connection(检查设备的网络连接信息)

设备可以有多种网络连接。关于所有可能的网络连接类型，请看[ConnectivityManager](#)。

通常Wi-Fi是比较快的。移动数据通常都是需要按流量计费，会比较贵。通常我们会选择让app在连接到WiFi时去获取大量的数据。

那么，我们就需要在执行网络操作之前检查当前连接的网络信息。这样可以防止你的程序不经意连接使用了非意向的网络频道。为了实现这个目的，我们需要使用到下面两个类：

- [ConnectivityManager](#): Answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.
- [NetworkInfo](#): Describes the status of a network interface of a given type (currently either Mobile or Wi-Fi).

下面示例了检查WiFi与Mobile是否连接上(请注意available与isConnected的区别)：

```
private static final String DEBUG_TAG = "NetworkStatusExample";
...
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

一个更简单的检查网络是否可用的示例如下：

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

你可以使用[NetworkInfo.DetailedState](#) 来获取更加详细的网络信息。

Manage Network Usage(管理网络使用)

你可以实现一个偏好设置的activity，来允许用户设置程序的网络资源的使用。例如：

- 你可以允许用户在仅仅连接到WiFi时上传视频。
- 你可以根据诸如网络可用等条件来选择是否做同步的操作。

网络操作需要添加下面的权限：

- [android.permission.INTERNET](#)—Allows applications to open network sockets.
- [android.permission.ACCESS_NETWORK_STATE](#)—Allows applications to access information about networks.

你可以为你的设置Activity声明intent filter for the ACTION_MANAGE_NETWORK_USAGE action (introduced in Android 4.0),这样你的这个activity就可以提供数据控制的选项了。在章节概览提供的Sample中，这个action is handled by the class SettingsActivity, 它提供了偏好设置UI来让用户决定何时进行下载。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.networkusage"
    ...>

    <uses-sdk android:minSdkVersion="4"
        android:targetSdkVersion="14" />

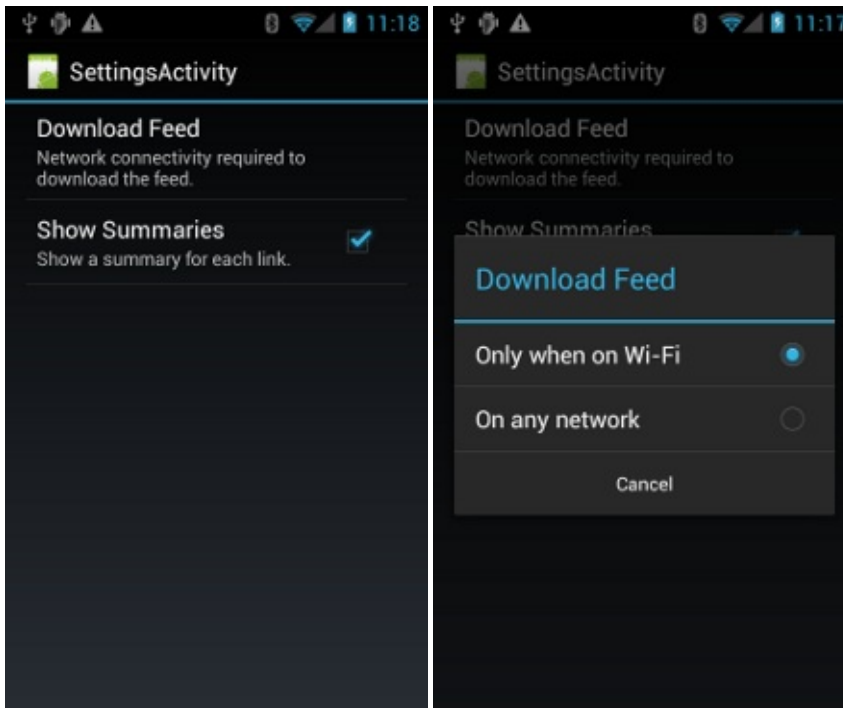
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        ...>
        ...
        <activity android:label="SettingsActivity" android:name=".SettingsActivity">
            <intent-filter>
                <action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Implement a Preferences Activity(实现一个偏好设置activity)

正如上面看到的那样，SettingsActivity is a subclass of PreferenceActivity.

所实现的功能见下图：



下面是一个 SettingsActivity. 请注意它实现了 OnSharedPreferencesChangeListener. 当用户改变了他的偏好，就会触发 onSharedPreferencesChanged(), 这个方法会设置 refreshDisplay 为 true(这里的变量存在于自己定义的 activity, 见下一部分的代码示例). 这会使得当用户返回到 main activity 的时候进行 refresh. (请注意，代码中的注释，不得不说，Googler 写的 Code 看起来就是舒服)

```
public class SettingsActivity extends PreferenceActivity implements OnSharedPreferencesChangeListener {

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

        // Loads the XML preferences file
        addPreferencesFromResource(R.xml.preferences);
    }

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

        // Registers a listener whenever a key changes
        getPreferenceScreen().getSharedPreferences().registerOnSharedPreferencesChangeListener(this);
    }

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

        // Unregisters the listener set in onResume().
        // It's best practice to unregister listeners when your app isn't using them to cut down on
        // unnecessary system overhead. You do this in onPause().
        getPreferenceScreen().getSharedPreferences().unregisterOnSharedPreferencesChangeListener(this);
    }

    // When the user changes the preferences selection,
    // onSharedPreferencesChanged() restarts the main activity as a new
    // task. Sets the the refreshDisplay flag to "true" to indicate that
    // the main activity should update its display.
    // The main activity queries the PreferenceManager to get the latest settings.

    @Override
    public void onSharedPreferencesChanged(SharedPreferences sharedPreferences, String key) {
        // Sets refreshDisplay to true so that when the user returns to the main
```

```
        // activity, the display refreshes to reflect the new settings.  
        NetworkActivity.refreshDisplay = true;  
    }  
}
```

Respond to Preference Changes(对偏好改变进行响应)

当用户在设置界面改变了偏好，它通常都会对app的行为产生影响。在下面的代码示例中，app会在onStart(). 方法里面检查偏好设置。如果设置的类型与当前设备的网络连接类型相一致，那么程序就会下载数据并刷新显示。(for example, if the setting is "Wi-Fi" and the device has a Wi-Fi connection)。 (这是一个很好的代码示例， 如何选择合适的网络类型进行下载操作)

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagnames=android&s

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;

    // The user's current network preference setting.
    public static String sPref = null;

    // The BroadcastReceiver that tracks network connectivity changes.
    private NetworkReceiver receiver = new NetworkReceiver();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Registers BroadcastReceiver to track network connection changes.
        IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
        receiver = new NetworkReceiver();
        this.registerReceiver(receiver, filter);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // Unregisters BroadcastReceiver when app is destroyed.
        if (receiver != null) {
            this.unregisterReceiver(receiver);
        }
    }

    // Refreshes the display if the network connection and the
    // pref settings allow it.

    @Override
    public void onStart () {
        super.onStart();

        // Gets the user's network preference settings
        SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this);

        // Retrieves a string value for the preferences. The second parameter
        // is the default value to use if a preference value is not found.
        sPref = sharedPrefs.getString("listPref", "Wi-Fi");

        updateConnectedFlags();

        if(refreshDisplay){
            loadPage();
        }
    }

    // Checks the network connection and sets the wifiConnected and mobileConnected
    // variables accordingly.
    public void updateConnectedFlags() {
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo activeInfo = connMgr.getActiveNetworkInfo();
        if (activeInfo != null && activeInfo.isConnected()) {
            wifiConnected = activeInfo.getType() == ConnectivityManager.TYPE_WIFI;
            mobileConnected = activeInfo.getType() == ConnectivityManager.TYPE_MOBILE;
        } else {
```

```
        wifiConnected = false;
        mobileConnected = false;
    }
}

// Uses AsyncTask subclass to download the XML feed from stackoverflow.com.
public void loadPage() {
    if (((sPref.equals(ANY)) && (wifiConnected || mobileConnected))
        || ((sPref.equals(WIFI)) && (wifiConnected))) {
        // AsyncTask subclass
        new DownloadXmlTask().execute(URL);
    } else {
        showErrorPage();
    }
}

...
}
```

Detect Connection Changes(监测网络连接的改变)

最后一部分是关于 BroadcastReceiver 的子类：NetworkReceiver. 当设备网络连接改变时，NetworkReceiver会监听到 CONNECTIVITY_ACTION, 这时需要判断当前网络连接类型并相应的设置好 wifiConnected 与 mobileConnected .

我们需要控制好BroadcastReceiver的使用，不必要的声明注册会浪费系统资源。通常是在 onCreate() 去registers 这个 BroadcastReceiver， 在 onPause()或者 onDestroy() 时unregisters它。这样做会比直接在manifest里面直接注册 更轻量. 当你在 manifest里面注册了一个， 你的程序可以在任何时候被唤醒, 即使你已经好几个星期没有使用这个程序了。而通过前面的办法进行注册，可以确保用户离开你的程序之后，不会因为那个Broadcast而被唤起。如果你确保知道何时需要使用到它，你可以在合适的地方使用 [setComponentEnabledSetting\(\)](#) 来开启或者关闭它。

```
public class NetworkReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        ConnectivityManager conn = (ConnectivityManager)
            context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = conn.getActiveNetworkInfo();

        // Checks the user prefs and the network connection. Based on the result, decides whether
        // to refresh the display or keep the current display.
        // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi connection.
        if (WIFI.equals(sPref) && networkInfo != null && networkInfo.getType() == ConnectivityManager.TYPE_WIFI) {
            // If device has its Wi-Fi connection, sets refreshDisplay
            // to true. This causes the display to be refreshed when the user
            // returns to the app.
            refreshDisplay = true;
            Toast.makeText(context, R.string.wifi_connected, Toast.LENGTH_SHORT).show();

            // If the setting is ANY network and there is a network connection
            // (which by process of elimination would be mobile), sets refreshDisplay to true.
        } else if (ANY.equals(sPref) && networkInfo != null) {
            refreshDisplay = true;

            // Otherwise, the app can't download content--either because there is no network
            // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and there
            // is no Wi-Fi connection.
            // Sets refreshDisplay to false.
        } else {
            refreshDisplay = false;
            Toast.makeText(context, R.string.lost_connection, Toast.LENGTH_SHORT).show();
        }
    }
}
```

编写:[kesenhoo](#)

校对:

解析XML数据Parsing XML Data

Extensible Markup Language (XML) .很多网站或博客上都提供XML feed来记录更新的信息，以便用户进行订阅读取。

那么上传[?]与解析XML数据就成了app的一个常见的功能。 这一课会介绍如何解析XML文档并使用他们的数据。

([?]这里很奇怪，为什么是Upload，看文章最后一段代码示例的注释， 应该是Download才对)

Choose a Parser(选择一个解析器)

我们推荐[XmlPullParser](#), 它是在Android上一个高效且可维护的解析XML方法。 Android 上有这个接口的两种实现方式：

- [KXmlParser](#) via [XmlPullParserFactory.newPullParser\(\)](#).
- ExpatPullParser, via [Xml.newPullParser\(\)](#).

两个选择都是比较好的。下面的示例中是使用ExpatPullParser, via Xml.newPullParser().

Analyze the Feed(分析Feed)

解析一个feed的第一步是决定需要获取哪些字段。这样解析器才去抽取出那些需要的字段而忽视剩下的。下面一段章节概览Sample app中截取的一段代码示例。

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:creativecommons="http://backend.userland.com/creativecommons" type="text">newest questions tagged android - Stack Overflow</title>
...
  <entry>
    ...
  </entry>
  <entry>
    <id>http://stackoverflow.com/q/9439999</id>
    <re:rank scheme="http://stackoverflow.com">0</re:rank>
    <title type="text">Where is my data file?</title>
    <category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sort=newest/tag">android</category>
    <category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sort=newest/tag">android</category>
    <author>
      <name>cliff2310</name>
      <uri>http://stackoverflow.com/users/1128925</uri>
    </author>
    <link rel="alternate" href="http://stackoverflow.com/questions/9439999/where-is-my-data-file?<
    <published>2012-02-25T00:30:54Z</published>
    <updated>2012-02-25T00:30:54Z</updated>
    <summary type="html">
      <p>I have an Application that requires a data file...</p>
    </summary>
  </entry>
  <entry>
    ...
  </entry>
...
</feed>
```

在sample app中抽取了entry 标签与它的子标签 title, link,summary.

Instantiate the Parser(实例化解析器)

下一步就是实例化一个parser并开始解析的操作。请看下面的示例：

```
public class StackOverflowXmlParser {
    // We don't use namespaces
    private static final String ns = null;

    public List parse(InputStream in) throws XmlPullParserException, IOException {
        try {
            XmlPullParser parser = Xml.newPullParser();
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            parser.setInput(in, null);
            parser.nextTag();
            return readFeed(parser);
        } finally {
            in.close();
        }
    }
    ...
}
```

Read the Feed(读取Feed)

The readFeed() 实际上并没有处理feed的内容。它只是在寻找一个 "entry" 的标签作为递归（recursively）处理整个feed的起点。如果一个标签它不是"entry", readFeed()方法会跳过它. 当整个feed都被递归处理后, readFeed() 会返回一个包含了entry标签（包括里面的数据成员）的 List。

```
private List readFeed(XmlPullParser parser) throws XmlPullParserException, IOException {
    List entries = new ArrayList();

    parser.require(XmlPullParser.START_TAG, ns, "feed");
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        // Starts by looking for the entry tag
        if (name.equals("entry")) {
            entries.add(readEntry(parser));
        } else {
            skip(parser);
        }
    }
    return entries;
}
```

Parse XML(解析XML)

解析步骤如下：

- 正如在上面“分析Feed”所说的，判断出你想要的tag。这个example抽取了 entry 标签与它的内部标签 title, link,summary.
- 创建下面的方法：
 - 为每一个你想要获取的标签创建一个 "read" 方法。例如 readEntry(), readTitle() 等等. 解析器从input stream中读取 tag. 当读取到 entry, title, link 或者 summary 标签时，它会为那些标签调用相应的方法，否则，跳过这个标签。
 - 为每一个不同的标签的提取数据方法进行优化，例如：
 - 对于 title and summary tags, 解析器调用 readText(). 通过调用parser.getText().来获取返回数据。
 - 对于 link tag,解析器先判断这个link是否是我们想要的类型，然后再读取数据。可以使用 parser.getAttributeValue() 来获取返回数据。
 - 对于 entry tag, 解析器调用 readEntry(). 这个方法解析entry的内部标签并返回一个带有title, link, and summary数据成员的Entry对象。
 - 一个帮助方法：skip(). 关于这部分的讨论，请看下面一部分内容：Skip Tags You Don't Care About

下面的代码演示了如何解析 entries, titles, links, 与 summaries.

```
public static class Entry {
    public final String title;
    public final String link;
    public final String summary;

    private Entry(String title, String summary, String link) {
        this.title = title;
        this.summary = summary;
        this.link = link;
    }
}

// Parses the contents of an entry. If it encounters a title, summary, or link tag, hands th
// to their respective "read" methods for processing. Otherwise, skips the tag.
private Entry readEntry(XmlPullParser parser) throws XmlPullParserException, IOException {
    parser.require(XmlPullParser.START_TAG, ns, "entry");
    String title = null;
    String summary = null;
    String link = null;
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        if (name.equals("title")) {
            title = readTitle(parser);
        } else if (name.equals("summary")) {
            summary = readSummary(parser);
        } else if (name.equals("link")) {
            link = readLink(parser);
        } else {
            skip(parser);
        }
    }
    return new Entry(title, summary, link);
}

// Processes title tags in the feed.
private String readTitle(XmlPullParser parser) throws IOException, XmlPullParserException {
    parser.require(XmlPullParser.START_TAG, ns, "title");
    String title = readText(parser);
    parser.require(XmlPullParser.END_TAG, ns, "title");
    return title;
}

// Processes link tags in the feed.
private String readLink(XmlPullParser parser) throws IOException, XmlPullParserException {
    String link = "";
    parser.require(XmlPullParser.START_TAG, ns, "link");
    String tag = parser.getName();
    String relType = parser.getAttributeValue(null, "rel");
    if (tag.equals("link")) {
        if (relType.equals("alternate")){
            link = parser.getAttributeValue(null, "href");
            parser.nextTag();
        }
    }
}
```

```
        parser.require(XmlPullParser.END_TAG, ns, "link");
        return link;
    }

    // Processes summary tags in the feed.
    private String readSummary(XmlPullParser parser) throws IOException, XmlPullParserException {
        parser.require(XmlPullParser.START_TAG, ns, "summary");
        String summary = readText(parser);
        parser.require(XmlPullParser.END_TAG, ns, "summary");
        return summary;
    }

    // For the tags title and summary, extracts their text values.
    private String readText(XmlPullParser parser) throws IOException, XmlPullParserException {
        String result = "";
        if (parser.next() == XmlPullParser.TEXT) {
            result = parser.getText();
            parser.nextTag();
        }
        return result;
    }
    ...
}
```

Skip Tags You Don't Care About(跳过你不在意标签)

下面演示解析器的 skip() 方法:

```
private void skip(XmlPullParser parser) throws XmlPullParserException, IOException {
    if (parser.getEventType() != XmlPullParser.START_TAG) {
        throw new IllegalStateException();
    }
    int depth = 1;
    while (depth != 0) {
        switch (parser.next()) {
            case XmlPullParser.END_TAG:
                depth--;
                break;
            case XmlPullParser.START_TAG:
                depth++;
                break;
        }
    }
}
```

上面这个方法是如何工作的呢？

- It throws an exception if the current event isn't a START_TAG.
- It consumes the START_TAG, and all events up to and including the matching END_TAG.
- To make sure that it stops at the correct END_TAG and not at the first tag it encounters after the original START_TAG, it keeps track of the nesting depth.

因此如果目前的标签有子标签, depth 的值就不会为 0, 直到解析器已经处理了所有位于 START_TAG 与 END_TAG 之间的事件。例如, 看解析器如何跳过 标签, 它有 2 个子标签, 与 :

- The first time through the while loop, the next tag the parser encounters after is the START_TAG for . The value for depth is incremented to 2.
- The second time through the while loop, the next tag the parser encounters is the END_TAG . The value for depth is decremented to 1.
- The third time through the while loop, the next tag the parser encounters is the START_TAG . The value for depth is incremented to 2.
- The fourth time through the while loop, the next tag the parser encounters is the END_TAG . The value for depth is decremented to 1.
- The fifth time and final time through the while loop, the next tag the parser encounters is the END_TAG . The value for depth is decremented to 0, indicating that the element has been successfully skipped.

Consume XML Data(使用XML数据)

示例程序是在 AsyncTask 中获取与解析XML数据的。当获取到数据后，程序会在main activity(NetworkActivity)里面进行更新操作。

在下面示例代码中，loadPage() 方法做了下面的事情：

- 初始化一个带有URL地址的String变量，用来订阅XML feed。
- 如果用户设置与网络连接都允许，会触发 new DownloadXmlTask().execute(url). 这会初始化一个新的 DownloadXmlTask(AsyncTask subclass) 对象并且开始执行它的 execute() 方法。

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagnames=android&S

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;
    public static String sPref = null;

    ...

    // Uses AsyncTask to download the XML feed from stackoverflow.com.
    public void loadPage() {

        if((sPref.equals(ANY)) && (wifiConnected || mobileConnected)) {
            new DownloadXmlTask().execute(URL);
        }
        else if ((sPref.equals(WIFI)) && (wifiConnected)) {
            new DownloadXmlTask().execute(URL);
        } else {
            // show error
        }
    }
}
```

下面是DownloadXmlTask是怎么工作的：

```
// Implementation of AsyncTask used to download XML feed from stackoverflow.com.
private class DownloadXmlTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... urls) {
        try {
            return loadXmlFromNetwork(urls[0]);
        } catch (IOException e) {
            return getResources().getString(R.string.connection_error);
        } catch (XmlPullParserException e) {
            return getResources().getString(R.string.xml_error);
        }
    }

    @Override
    protected void onPostExecute(String result) {
        setContentView(R.layout.main);
        // Displays the HTML string in the UI via a WebView
        WebView myWebView = (WebView) findViewById(R.id.webview);
        myWebView.loadData(result, "text/html", null);
    }
}
```

下面是loadXmlFromNetwork是怎么工作的：

```
// Uploads XML from stackoverflow.com, parses it, and combines it with
// HTML markup. Returns HTML string. [这里可以看出应该是Download]
private String loadXmlFromNetwork(String urlString) throws XmlPullParserException, IOException {
    InputStream stream = null;
    // Instantiate the parser
    StackOverflowXmlParser stackOverflowXmlParser = new StackOverflowXmlParser();
    List<Entry> entries = null;
    String title = null;
```



```

String url = null;
String summary = null;
Calendar rightNow = Calendar.getInstance();
DateFormat formatter = new SimpleDateFormat("MMM dd h:mmaa");

// Checks whether the user set the preference to include summary text
SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this);
boolean pref = sharedPrefs.getBoolean("summaryPref", false);

StringBuilder htmlString = new StringBuilder();
htmlString.append("<h3>" + getResources().getString(R.string.page_title) + "</h3>");
htmlString.append("<em>" + getResources().getString(R.string.updated) + " " +
    formatter.format(rightNow.getTime()) + "</em>");

try {
    stream = downloadUrl(urlString);
    entries = stackOverflowXmlParser.parse(stream);
    // Makes sure that the InputStream is closed after the app is
    // finished using it.
} finally {
    if (stream != null) {
        stream.close();
    }
}

// StackOverflowXmlParser returns a List (called "entries") of Entry objects.
// Each Entry object represents a single post in the XML feed.
// This section processes the entries list to combine each entry with HTML markup.
// Each entry is displayed in the UI as a link that optionally includes
// a text summary.
for (Entry entry : entries) {
    htmlString.append("<p><a href='");
    htmlString.append(entry.link);
    htmlString.append(">" + entry.title + "</a></p>");
    // If the user set the preference to include summary text,
    // adds it to the display.
    if (pref) {
        htmlString.append(entry.summary);
    }
}
return htmlString.toString();
}

// Given a string representation of a URL, sets up a connection and gets
// an input stream.
【关于Timeout具体应该设置多少，可以借鉴这里的数据，当然前提是一般情况下】
// Given a string representation of a URL, sets up a connection and gets
// an input stream.
private InputStream downloadUrl(String urlString) throws IOException {
    URL url = new URL(urlString);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000 /* milliseconds */);
    conn.setConnectTimeout(15000 /* milliseconds */);
    conn.setRequestMethod("GET");
    conn.setDoInput(true);
    // Starts the query
    conn.connect();
    return conn.getInputStream();
}

```

编写:[kesenhoo](#)

校对:

高效下载

在这一章，我们将学习为了最小化某些操作对电量的影响是如何处理下载，网络连接，尤其是无线电连接的。

下面几节课会演示了如何使用缓存caching，轮询polling，预取prefetching等技术来计划与执行下载操作。 我们还会学习无线电波的power-use属性配置是如何影响我们对于在何时，用什么，以何种方式来传输数据的选择。 当然这些选择是为了最小化对电池寿命的影响。

Dependencies and prerequisites

Android 2.0 (API Level 5) or higher

You should also read [Optimizing Battery Life](#)

Lessons

- **Optimizing Downloads for Efficient Network Access**[使用有效的网络连接方式来最优化下载]

This lesson introduces the wireless radio state machine, explains how your app's connectivity model interacts with it, and how you can minimize your data connection and use prefetching and bundling to minimize the battery drain associated with your data transfers.

- **Minimizing the Effect of Regular Updates**[优化常规更新操作的效果]

This lesson will examine how your refresh frequency can be varied to best mitigate the effect of background updates on the underlying wireless radio state machine.

- **Redundant Downloads are Redundant**[重复的下载是冗余的]

The most fundamental way to reduce your downloads is to download only what you need. This lesson introduces some best practices to eliminate redundant downloads.

- **Modifying your Download Patterns Based on the Connectivity Type**[根据网络连接类型来更改下载模式]

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications.

编写:[kesenhoo](#)

校对:

Optimizing Downloads for Efficient Network Access(用有效的网络访问来最优化下载)

也许使用无线电波(wireless radio)进行传输数据会是我们app最耗电的操作之一。所以为了最小化网络连接的电量消耗，懂得连接模式(connectivity model)会如何影响底层的音频硬件设备是至关重要的。这节课介绍了无线电波状态机(wireless radio state machine)，并解释了app的connectivity model是如何与状态机进行交互的。然后会提出建议的方法来最小化我们的数据连接，使用预取(prefetching)与捆绑(bundle)的方式进行数据的传输，这些操作都是为了最小化电量的消耗。

The Radio State Machine(无线电状态机)

一个完全活动的无线电台会消耗很大部分的电量，因此需要学习如何在不同状态下进行过渡，这样能够避免电量的浪费。典型的3G无线网络有三种能量状态：

- **Full power:** 当无线连接被激活的时候，允许设备以最大的传输速率进行操作。
- **Low power:** 相对Full power来说，算是一种中间状态，差不多50%的传输速率。
- **Standby:** 最低的状态，没有数据连接需要传输。

在最低并且空闲的状态下，电量消耗相对来说是少的。这里需要介绍一延时(latency)的机制，从low status返回到full status大概需要花费1.5秒，从idle status返回到full status需要花费2秒。为了最小化延迟，状态机使用了一种后滞过渡到更低能量状态的机制。下图是一个典型的3G无线电波状态机的图示(AT&T电信的一种制式)。



- 在每一台设备上的无线状态机都会根据无线电波的制式(2G,3G,LTE等)而改变，并且由设备本身自己所使用的网络进行定义与配置。
- 这一课描述了一种典型的3G无线电波状态机，[data provided by AT&T](#)。这些原理是具有通用性的，在其他的无线电波上同样适用。
- 这种方法在浏览通常的网页操作上是特别有效的，因为它可以阻止一些不必要的浪费。而且相对较短的后期处理时间也保证了当一个session结束的时候，无线电波可以转移到相对较低的能量状态。
- 不幸的是，这个方法会导致在现代的智能机系统例如Android上的apps效率低下。因为Android上的apps不仅仅可以在前台运行，也可以在后台运行。(无线电波的状态改变会影响到本来的设计，有些想在前台运行的可能会因为切换到低能量状态而影响程序效率。坊间说手机在电量低的状态下无线电波的强度会增大好几倍来保证信号，可能与这个有关。)

How Apps Impact the Radio State Machine[看apps如何影响无线状态机(使用bundle与unbundle传输数据的差异)]

每一次新创建一个网络连接，无线电波就切换到full power状态。在上面典型的3G无线电波状态机情况下，无线电波会在传输数据时保持在full power的状态，结束之后会有一个附加的5秒时间切换到low power,再之后会经过12秒进入到low energy的状态。因此对于典型的3G设备，每一次数据传输的会话都会引起无线电波都会持续消耗大概20秒的能量。

实际上，这意味着一个app传递1秒钟的unbundled data会使得无线电波持续活动18秒(18=1秒的传输数据+5秒过渡时间回到low power+12秒过渡时间回到standby)。因此每一分钟，它会消耗18秒high power的电量，42秒的low power的电量。

通过比较，如果每分钟app会传输bundle的数据持续3秒的话，其中会使得无线电波持续在high power状态仅仅8秒钟，在low power状态仅仅12秒钟。上面第二种传输bundle data的例子，可以看到减少了大量的电量消耗。图示如下：



Prefetch Data(预取数据)

预取(Prefetching)数据是一种减少独立数据传输会话数量的有效方法。预取技术允许你在单次操作的时候，通过一次连接，在最大能力下，根据给出的时间下载到所有的数据。

通过前面的传输数据的技术，你减少了大量的无线电波激活时间。这样的话，不仅仅是保存了电量，也提高了潜在风险，降低了带宽，减少了下载时间。

预取技术提供了一种提高用户体验的方法，通过减少可能因为下载时间过长而导致预览后者后续操作等待漫长。

然而，使用预取技术过于频繁，不仅仅会导致电量消耗快速增长，还有可能预取到一些并不需要的数据。同样，确保app不会因为等待预取全部完成而卡到程序的开始播放也是非常重要的。从实践的角度，那意味着需要逐步处理数据，并且按照有优先级的顺序开始进行数据传递，这样能确保不卡到程序的开始播放的同时数据也能够得到持续的下载。

那么应该如何控制预取的操作呢？这需要根据正在下载的数据大小与可能被用到的数据量来决定。一个基于上面状态机情况的比较大概的建议是：对于数据来说，大概有50%的机会可能用在当前用户的会话中，那么我们可以预取大约6秒(大约1-2Mb)，这大概使得潜在可能要用的数据量与可能已经下载好的数据量相一致。

通常来说，预取1-5Mb会比较好，这种情况下，我们仅仅只需要每隔2-5分钟开始另一段下载。根据这个原理，大数据的下载，比如视频文件，应该每隔2-5秒开始另一段下载，这样能有效的预取到下面几分钟内的数据进行预览。

值得注意的是，下载需要是bundled的形式，而且上面那些大概的数据与时间可能会根据网络连接的类型与速度有所变化，这些都将在下面两部分内容讲到。

让我们来看一些例子：

A music player 你可以选择预取整个专辑，然而这样用户在第一首歌曲之后停止监听，那么就浪费了大量的带宽于电量。一个比较好的方法是维护一首歌曲的缓冲区。对于流媒体音乐，不应该去维护一段连续的数据流，因为这样会使得无线电波一直保持激活状态，应该考虑把HTTP的数据流集中一次传输到音频流，就像上面描述的预取技术一样(下载好2Mb，然后开始一次取出，再去下载下面的2Mb)。

A news reader 许多news apps尝试通过只下载新闻标题来减少带宽，完整的文章仅在用户想要读取的时候再去读取，而且文章也会因为太长而刚开始只显示部分信息，等用户下滑时再去读取完整信息。使用这个方法，无线电波仅仅会在用户点击更多信息的时候才会被激活。但是，在切换文章分类预阅读文章的时候仍然会造成大量潜在的消耗。

一个比较好的方法是在启动的时候预取一个合理数量的数据，比如在启动的时候预取一些文章的标题与缩略图信息。之后开始获取剩余的标题预缩略图信息。

另一个方法是预取所有的标题，缩略图信息，文章文字，甚至是所有文章的图片-根据既设的后台程序进行逐一获取。这样做的风险是花费了大量的带宽与电量去下载一些不会阅读到的内容，因此这需要比较小心思考是否合适。其中的一个解决方案是，当在连接至Wi-Fi时有计划的下载所有的内容，并且如果有可能最好是设备正在充电的时候。关于这个的细节的实现，我们将在后面的课程中涉及到。【这让我想起了网易新闻的离线下载，在连接到Wi-Fi的时候，可以选择下载所有的内容到本地，之后直接打开阅读】

Batch Transfers and Connections(批量传输与连接)

使用典型3G无线网络制式的时候，每一次初始化一个连接(与需要传输的数据量无关)，你都有可能导致无线电波持续花费大约20秒的电量。

一个app，若是每20秒进行一次ping server的操作，假设这个app是正在运行且对用户可见，那么这会导致无线电波不确定什么时候被开启，最终可能使得电量花费在没有实际传输数据的情况下。

因此，对数据进行bundle操作并且创建一个序列来存放这些bundle好的数据就显的非常重要。操作正确的话，可以使得大量的数据集中进行发送，这样使得无线电波的激活时间尽可能的少，同时减少大部分电量的花费。这样做的潜在好处是尽可能在每次传输数据的会话中尽可能多的传输数据而且减少了会话的次数。

Reduce Connections(减少连接次数)

重用已经存在的网络连接比起重新建立一个新的连接通常来说是更有效率的。重用网络连接同样可以使得在拥挤不堪的网络环境中进行更加智能的互动。当可以捆绑所有请求在一个GET里面的时候不要同时创建多个网络连接或者把多个GET请求进行串联。

例如，可以一起请求所有文章的情况下，不要根据多个栏目进行多次请求。无线电波会在等待接受返回信息或者timeout信息之前保持激活状态，所以如果不需要的连接请立即关闭而不是等待他们timeout。

之前说道，如果关闭一个连接过于及时，会导致后面再次请求时重新建立一个在Server与Client之间的连接，而我们说过要尽量避免建立重复的连接，那么有个有效的折中办法是不要立即关闭，而是在timeout之前关闭(即稍微晚点却又不至于到timeout)。

Note:使用URLConnection，而不是Apache的HttpClient,前者有做response cache.

Use the DDMS Network Traffic Tool to Identify Areas of Concern[使用DDMS网络通信工具来检测网络使用情况]

The Android [DDMS \(Dalvik Debug Monitor Server\)](#) 包含了一个查看网络使用详情的栏目来允许跟踪app的网络请求。使用这个工具，可以监测app是在何时，如何传输数据的，从而可以进行代码的优化。

下图显示了传输少量的网络模型，可以看到每次差不多相隔15秒，这意味着可以通过预取技术或者批量上传来大幅提高效率。



通过监测数据传输的频率与每次传输的数据量，可以查看出哪些位置应该进行优化，通常的，图中显示的短小的类似钉子形状的位置，可以进行与附近位置的请求进行做merge的动作。

为了更好的检测出问题所在，**Traffic Status API**允许你使用**TrafficStats.setThreadStatsTag()**的方法标记数据传输发生在某个Thread里面，然后可以手动的使用tagSocket()进行标记到或者使用untagSocket()来取消标记，例如：

```
TrafficStats.setThreadStatsTag(0xF00D);
TrafficStats.tagSocket(outputSocket);
// Transfer data using socket
TrafficStats.untagSocket(outputSocket);
```

Apache的HttpClient与URLConnection库可以自动tag sockets使用当前getThreadStatsTag()的值。那些库在通过keep-alive pools循环的时候也会tag与untag sockets。

```
TrafficStats.setThreadStatsTag(0xF00D);
try {
    // Make network request using HttpClient.execute()
} finally {
    TrafficStats.clearThreadStatsTag();
}
```

Socket tagging 是在Android 4.0上才被支持的, 但是实际情况是仅仅会在运行Android 4.0.3 or higher的设备上才会显示.

编写:[kesenhoo](#)

校对:

Minimizing the Effect of Regular Updates(最小化定期更新操作的副作用)

最佳的定时更新频率是不确定的，通常由设备状态，网络连接状态，用户行为与用户定义明确的偏好而决定。

[Optimizing Battery Life](#)这一章有讨论如何根据设备状态来修改更新频率。里面介绍了当断开网络连接的时候去关闭后台服务，在电量比较低的时候减少更新的频率。

这一课会介绍更新频率是多少才会使得更新操作对无线电状态机的影响最小。(C2DM与指数退避算法的使用)

Use Google Cloud Messaging as an Alternative to Polling[使用C2DM作为轮询方式之一]

关于Android Cloud to Device Messaging (C2DM)详情,请参考:<http://code.google.com/intl/zh-CN/android/c2dm/>

每次app去向server询问检查是否有更新操作的时候会激活无线电, 这样造成了不必要的能量消耗(在3G情况下, 会差不多消耗20秒的能量)。

C2DM是一个用来从server到特定app传输数据的轻量级的机制。使用C2DM,server会在某个app有需要获取新数据的时候通知app有这个消息。

比起轮询方式(app为了即时拿到最新的数据需要定时向server请求数据), C2DM这种有事件驱动的模式会在仅仅有数据更新的时候通知app去创建网络连接来获取数据(很显然这样减少了app的大量操作, 当然也减少了很多电量)。

C2DM需要通过使用固定TCP/IP来实现操作。当在你的设备上可以实现固定IP的时候, 最好使用C2DM。(这个地方应该不是传统意义上的固定IP, 可以理解为某个会话情况下)。很明显, 使用C2DM既减少了网络连接次数, 也优化了带宽, 还减少了对电量的消耗。

Ps:大陆的Google框架通常被移除掉, 这导致C2DM实际上根本没有办法在大陆的App上使用

Optimize Polling with Inexact Repeating Alarms and Exponential Backoffs(通过不定时的重复提醒与指数退避来优化轮询操作)

如果需要使用轮询机制，在不影响用户体验的前提下，当然设置默认更新频率是越低越好(减少电量的浪费)。

一个简单的方法是给用户提供更更新频率的选择，允许用户自己来处理如何平衡数据及时性与电量的消耗。

当设置安排好更新操作后，可以使用不确定重复提醒的方式来允许系统把当前这个操作进行定向移动(比如推迟一会)。

```
int alarmType = AlarmManager.ELAPSED_REALTIME;
long interval = AlarmManager.INTERVAL_HOUR;
long start = System.currentTimeMillis() + interval;

alarmManager.setInexactRepeating(alarmType, start, interval, pi);
```

若是多个提醒都安排在某个点同时被触发，那么这样就可以使得多个操作在同一个无线电状态下操作完。

如果可以，请设置提醒的类型为ELAPSED_REALTIME or RTC而不是_WAKEUP。这样能够更进一步的减少电量的消耗。

我们还可以通过根据app被使用的频率来有选择性的减少更新的频率。

另一个方法在app在上一次更新操作之后还未被使用的情况下，使用指数退避算法exponential back-off algorithm来减少更新频率。当然我们也可以使用一些类似指数退避的方法。

```
SharedPreferences sp =
    context.getSharedPreferences(PREFS, Context.MODE_WORLD_READABLE);

boolean appUsed = sp.getBoolean(PREFS_APPUSED, false);
long updateInterval = sp.getLong(PREFS_INTERVAL, DEFAULT_REFRESH_INTERVAL);

if (!appUsed)
    if ((updateInterval * 2) > MAX_REFRESH_INTERVAL)
        updateInterval = MAX_REFRESH_INTERVAL;

Editor spEdit = sp.edit();
spEdit.putBoolean(PREFS_APPUSED, false);
spEdit.putLong(PREFS_INTERVAL, updateInterval);
spEdit.apply();

rescheduleUpdates(updateInterval);
executeUpdateOrPrefetch();
```

初始化一个网络连接的花费不会因为是否成功下载了数据而改变。我们可以使用指数退避算法来减少重复尝试(retry)的次数，这样能够避免浪费电量。例如：

```
private void retryIn(long interval) {
    boolean success = attemptTransfer();

    if (!success) {
        retryIn(interval * 2 < MAX_RETRY_INTERVAL ?
            interval * 2 : MAX_RETRY_INTERVAL);
    }
}
```

笔者结语:这一课讲到C2DM与指数退避算法等，其实这些细节很值得我们注意，如果能在实际项目中加以应用，很明显程序的质量上升了一个档次！

编写:[kesenhoo](#)

校对:

Redundant Downloads are Redundant(重复的下载是冗余的)

减少下载的最基本方法是仅仅下载那些你需要的。从数据的角度看，我们可以通过传递类似上次更新时间这样的参数来制定查询数据的条件。同样，在下载图片的时候，server那边最好能够减少图片的大小，而不是让我们下载完整大小的图片。

1)Cache Files Locally(缓存文件到本地)

避免下载重复的数据是很重要的。可以使用缓存机制来处理这个问题。缓存static的资源，例如完整的图片。这些缓存的资源需要分开存放。为了保证app不会因为缓存而导致显示的是旧数据，请从缓存中获取最新的数据，当数据过期的时候，会提示进行刷新。

```
long currentTime = System.currentTimeMillis();

URLConnection conn = (URLConnection) url.openConnection();

long expires = conn.getHeaderFieldDate("Expires", currentTime);
long lastModified = conn.getHeaderFieldDate("Last-Modified", currentTime);

setDataExpirationDate(expires);

if (lastModified < lastUpdateTime) {
    // Skip update
} else {
    // Parse update
}
```

使用这种方法，可以有效保证缓存里面一直是最新的数据。

可以使用下面的方法来获取External缓存的目录：(目录会是sdcard下面的Android/data/data/com.xxx.xxx/cache)

```
Context.getExternalCacheDir();
```

下面是获取内部缓存的方法，请注意，存放在内存中的数据有可能因内部空间不够而被清除。(类似:/system/data/data/com.xxx.xxx./cache)

```
Context.getCache();
```

上面两个Cache的文件都会在app卸载的时候被清除。

Ps:请注意这点:发现很多应用总是随便在sdcard下面创建一个目录用来存放缓存，可是这些缓存又不会随着程序的卸载而被删除，这其实是很令人讨厌的，程序都被卸载了，为何还要留那么多垃圾文件，而且这些文件有可能会泄漏一些隐私信息。除非你的程序是音乐下载，拍照程序等等，这些确定程序生成的文件是会被用户需要留下的，不然都应该使用上面的那种方式来获取Cache目录

2)Use the HttpURLConnection Response Cache(使用HttpURLConnection Response缓存)

在Android 4.0里面为HttpURLConnection增加了一个response cache(这是一个很好的减少http请求次数的机制，Android官方推荐使用HttpURLConnection而不是Apache的DefaultHttpClient，就是因为前者不仅仅有针对android做http请求的优化，还在4.0上增加了Reponse Cache，这进一步提高了效率)

我们可以使用反射机制开启HTTP response cache，看下面的例子：

```
private void enableHttpResponseCache() {
    try {
        long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
        File httpCacheDir = new File(getCacheDir(), "http");
        Class.forName("android.net.http.HttpResponseCache")
            .getMethod("install", File.class, long.class)
            .invoke(null, httpCacheDir, httpCacheSize);
    } catch (Exception httpResponseCacheNotAvailable) {
        Log.d(TAG, "HTTP response cache is unavailable.");
    }
}
```

上面的sample code会在Android 4.0以上的设备上开启response cache，同时不会影响到之前的程序。在cache被开启之后，所有cache中的HTTP请求都可以直接在本地存储中进行响应，并不需要开启一个新的网络连接。被cache起来的response可以被server所确保没有过期，这样就减少了带宽。没有被cached的response会因为方便下次请求而被存储在response cache中。

Ps:Cache机制在很多实际项目上都有使用到，实际操作会复杂许多，有机会希望能够分享一个Cache的例子。

编写:[kesenhoo](#)

校对:

根据网络类型改变下载模式**Modifying your Download Patterns Based on the Connectivity Type**

并不是所有的网络类型(Wi-Fi,3G,2G,etc)对电量的消耗是平等的。不仅仅Wi-Fi电波比无线电波消耗的电量要少很多，而且不同的无线电波(3G,2G,LTE.....)也存在使用不同电量的区别。

1)Use Wi-Fi[使用Wi-Fi]

在大多数情况下，Wi-Fi电波会在使用相对较低的电量的情况下提供一个相对较大的带宽。因此，我们需要努力争取尽量使用Wi-Fi来传递数据。我们可以使用Broadcast Receiver来监听当网络连接切换为Wi-Fi，这个时候我们可以进行大量的数据传递操作，例如下载，执行定时的更新操作，甚至是在这个时候加大更新的频率。这些内容都可以在前面的课程中找到。

2)Use Greater Bandwidth to Download More Data Less Often[使用更大的带宽来下载更多的数据，而不是经常去下载]

当通过无线电进行连接的时候，更大的带宽通常伴随着更多的电量消耗。这意味着LTE(一种4G网络制式)会比3G制式消耗更多，当然比起2G更甚。

从Lesson 1我们知道了无线电状态机是怎么回事，通常来说相对更宽的带宽网络制式会有更长的状态切换时间(也就是从full power过渡到standby有更长一段时间的延迟)。同时，更高的带宽意味着可以更贪婪的进行prefetch，下载更多的数据。也许这个说法不是很直观，因为过渡的时间比较长，而过渡时间的长短我们无法控制，也就是过渡时间的电量消耗差不多是固定了，既然如此，我们在每次传输会话中为了减少更新的频率而把无线电激活的时间拉长，这样显的更有效率。也就是尽量一次性把事情做完，而不是断断续续的请求。

例如：如果LTE无线电的带宽与电量消耗都是3G无线电的2倍，我们应该在每次会话的时候都下载4倍于3G的数据量，或者是差不多10Mb(前面文章有说明3G一般每次下载2Mb)。当然，下载到这么多数据的时候，我们需要好好考虑prefetch本地存储的效率并且需要经常刷新预取的cache。我们可以使用connectivity manager来判断当前激活的无线电波，并且根据不同结果来修改prefetch操作。

```
ConnectivityManager cm =
    (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

TelephonyManager tm =
    (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
    case (ConnectivityManager.TYPE_WIFI):
        PrefetchCacheSize = MAX_PREFETCH_CACHE; break;
    case (ConnectivityManager.TYPE_MOBILE): {
        switch (tm.getNetworkType()) {
            case (TelephonyManager.NETWORK_TYPE_LTE |
                 TelephonyManager.NETWORK_TYPE_HSPAP):
                PrefetchCacheSize *= 4;
                break;
            case (TelephonyManager.NETWORK_TYPE_EDGE |
                 TelephonyManager.NETWORK_TYPE_GPRS):
                PrefetchCacheSize /= 2;
                break;
            default: break;
        }
        break;
    }
    default: break;
}
```

Ps：想要最大化效率与最小化电量的消耗，需要考虑的东西太多了，通常来说，会根据app的功能需求来选择有所侧重，那么前提就是需要了解到底哪些对效率的影响比较大,这有利于我们做出最优选择。

编写:

校对:

使用**Sync Adapter**传输数据

编写:

校对:

创建Stub授权器

编写:

校对:

创建Stub Content Provider

编写:

校对:

创建Sync Adpater

编写:

校对:

执行**Sync Adpater**

编写:[kesenhoo](#)(2014-06-24)

校对:

使用Volley传输网络数据(Transmitting Network Data Using Volley)

Volley 是一个HTTP库，它能够帮助Android apps更方便的执行网络操作，最重要的是，它更快速高效。可以通过开源的[AOSP](#) 仓库获取到Volley。

DEPENDENCIES AND PREREQUISITES

Android 1.6 (API Level 4) or higher

YOU SHOULD ALSO SEE

使用Volley来编写一个app，请参考[2013 Google I/O schedule app](#). 另外需要特别关注下面2个部分：

- [ImageLoader](#)
- [BitmapCache](#)

[VIDEO - Volley:Easy,Fast Networking for Android](#)

Volley 有如下的优点：

- 自动执行网络请求。
- 高并发网络连接。
- 通过标准的HTTP的[cache coherence](#)(高速缓存一致性)使得磁盘与内存缓存不可见(Transparent)。
- 支持指定请求的优先级。
- 支持取消已经发出的请求。你可以取消单个请求，或者指定取消请求队列中的一个区域。
- 框架容易被定制，例如，定制重试或者回退功能。
- 强大的指令(Strong ordering)可以使得异步加载网络数据并显示到UI的操作更加简单。
- 包含了Debugging与tracing工具。

Volley擅长执行用来显示UI的RPC操作，例如获取搜索结果的数据。它轻松的整合了任何协议，并输出操作结果的数据，可以是raw strings，也可以是images，或者是JSON。通过提供内置你可能使用到的功能，Volley可以使得你免去重复编写样板代码，使你可以把关注点放在你的app的功能逻辑上。

Volley不适合用来下载大的数据文件。因为Volley会在解析的过程中保留持有所有的响应数据在内存中。对于下载大量的数据操作，请考虑使用[DownloadManager](#)。

Volley框架的核心代码是托管在AOSP仓库的frameworks/volley中，相关的工具放在toolbox下。把Volley添加到你的项目中的最简便的方法是Clone仓库然后把它设置为一个library project：

- 通过下面的命令来Clone仓库：

```
git clone https://android.googlesource.com/platform/frameworks/volley
```

- 以一个Android library project的方式导入下载的源代码到你的项目中。(如果你是使用Eclipse，请参考[Managing Projects from Eclipse with ADT](#))，或者编译成一个.jar文件。

Lessons

- [发送一个简单的网络请求\(Sending a Simple Request\)](#)

学习如何通过Volley默认的行为发送一个简单的请求，以及如何取消一个请求。

- [建立一个请求队列\(Setting Up a RequestQueue\)](#)

学习如何建立一个请求队列，以及如何实现一个单例模式来创建一个请求队列。

- [生成一个标准的请求\(Making a Standard Request\)](#)

学习如何使用Volley的out-of-the-box的请求类型(raw strings, images, and JSON)来发送一个请求。

- [实现自定义的请求\(Implementing a Custom Request\)](#)

学习如何实现一个自定义的请求

编写:[kesenhoo](#)(2014-06-24)

校对:

发送简单的网络请求(Sending a Simple Request)

使用Volley的方式是，你通过创建一个RequestQueue并传递Request对象给它。RequestQueue管理工作线程用来执行网络操作，从Cache中读取与写入数据，以及解析Http的响应内容。Requests执行raw responses的解析，Volley会把响应的数据分发给主线程。

这节课会介绍如何使用Volley.newRequestQueue这个建立请求队列的方法来发送一个请求，在下一节课[建立一个请求队列Setting Up a RequestQueue](#)中会介绍你自己如何建立一个请求队列。

这节课也会介绍如何添加一个请求到RequestQueue以及如何取消一个请求。

Add the INTERNET Permission

为了使用Volley，你必须添加`android.permission.INTERNET`权限到你的manifest文件中。没有这个权限，你的app将无法访问网络。

Use newRequestQueue

Volley提供了一个简便的方法：`Volley.newRequestQueue`用来为你建立一个`RequestQueue`，使用默认值，并启动这个队列。例如：

```
final TextView mTextView = (TextView) findViewById(R.id.text);
...

// Instantiate the RequestQueue.
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";

// Request a string response from the provided URL.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
    new Response.Listener() {
        @Override
        public void onResponse(String response) {
            // Display the first 500 characters of the response string.
            mTextView.setText("Response is: "+ response.substring(0,500));
        }
    }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            mTextView.setText("That didn't work!");
        }
    });
// Add the request to the RequestQueue.
queue.add(stringRequest);
```

Volley总是把解析过后的数据返回到主线程中。在主线程中更加合适使用接收到到的数据用来操作UI控件，这样你可以在响应的handler中轻松的修改UI，但是对于库提供的一些其他方法是有些特殊的，例如与取消有关的。

关于如何创建你自己的请求队列，不要使用`Volley.newRequestQueue`方法，请查看[建立一个请求队列Setting Up a RequestQueue](#)。

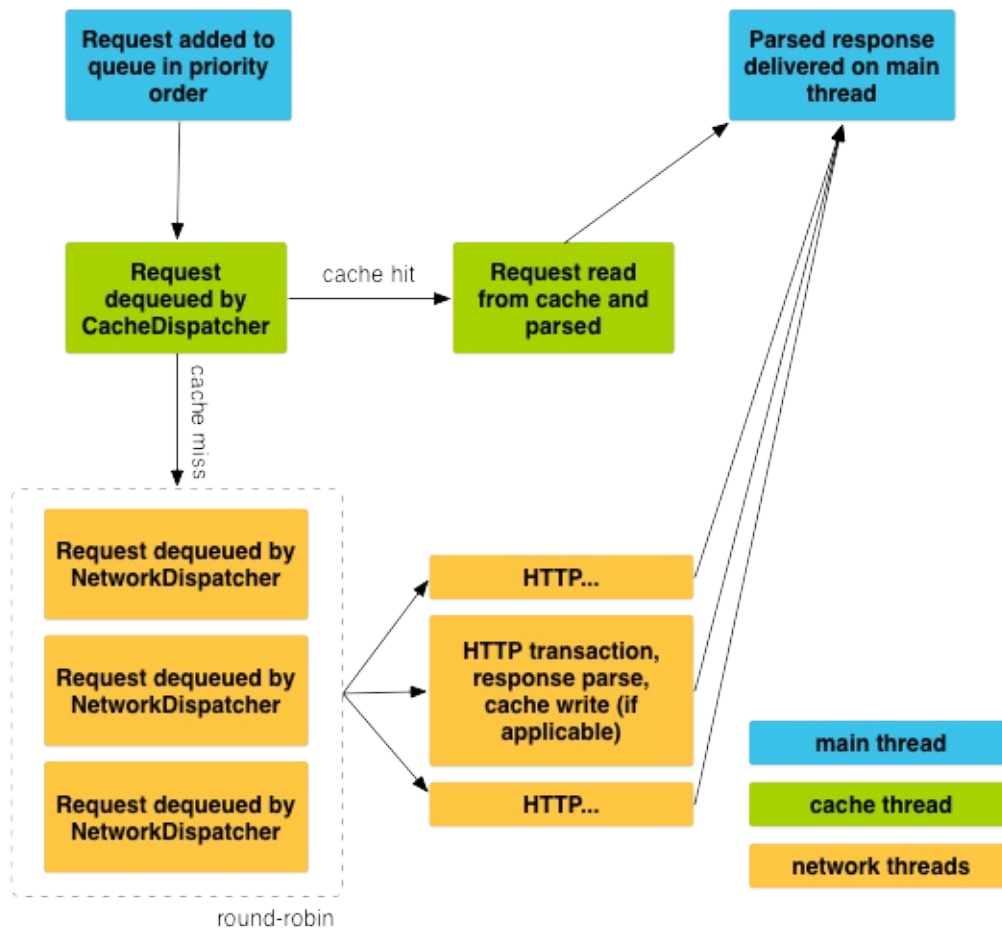
Send a Request

为了发送一个请求，你只需要构造一个请求并通过`add()`方法添加到`RequestQueue`中。一旦你添加了这个请求，它会通过队列，得到处理，然后得到原始的响应数据并返回。

当你执行`add()`方法时，Volley触发执行一个缓存处理线程以及网络一系列的网络处理线程。当你添加一个请求到队列中，它将被缓存线程所捕获并触发：如果这个请求可以被缓存处理，那么会在缓存线程中执行响应数据的解析并返回到主线程。如果请求不能被缓存所处理，它会被放到网络队列中。网络线程池中的第一个可用的网络线程会从队列中获取到这个请求并执行HTTP操作，解析响应数据，把数据写到缓存中之后再把解析之后的数据返回到主线程。

请注意那些比较耗时的操作，例如I/O与解析parsing/decoding都是执行在工作线程。你可以在任何线程中添加一个请求，但是响应结果都是返回到主线程的。

下图1，演示了一个请求的生命周期：



Cancel a Request

为了取消一个请求，对你的请求对象执行`cancel()`方法。一旦取消，Volley会确保你的响应Handler不会被执行。这意味着在实际操作中你可以在activity的`onStop()`方法中取消所有pending在队列中的请求。你不需要通过检测`getActivity() == null`来丢弃你的响应handler，其他类似`onSaveInstanceState()`等保护性的方法里面也都不需要检测。

为了利用这种优势，你应该跟踪所有已经发送的请求，以便在需要的时候，可以取消他们。有一个简便的方法：你可以为每一个请求对象都绑定一个tag对象。你可以使用这个tag来提供取消的范围。例如，你可以为你的所有请求都绑定到执行的Activity上，然后你可以在`onStop()`方法执行`requestQueue.cancelAll(this)`。同样的，你可以为ViewPager中的所有请求缩略图Request对象分别打上对应Tab的tag。并在滑动时取消这些请求，用来确保新生成的tab不会被前面tab的请求任务所卡到。

下面一个使用String来打Tag的例子：

1. 定义你的tag并添加到你的请求任务中。

```
public static final String TAG = "MyTag";
StringRequest stringRequest; // Assume this exists.
RequestQueue mRequestQueue; // Assume this exists.

// Set the tag on the request.
stringRequest.setTag(TAG);

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
```

1. 在activity的`onStop()`方法里面，取消所有的包含这个tag的请求任务。

```
@Override
protected void onStop () {
    super.onStop();
    if (mRequestQueue != null) {
        mRequestQueue.cancelAll(TAG);
    }
}
```

当取消请求时请注意：如果你依赖你的响应handler来标记状态或者触发另外一个进程，你需要为此给出有力的解释。再说一次，response handler是不会被执行的。

编写:[kesenhoo](#)(2014-06-24)

校对:

建立请求队列(Setting Up a RequestQueue)

前一节课演示了如何使用`Volley.newRequestQueue`这一简便的方法来建立一个`RequestQueue`，这是利用了Volley默认的优势。这节课会介绍如何显式的建立一个`RequestQueue`，以便满足你自定义的需求。

这节课同样会介绍一种推荐的实现方式：创建一个单例的`RequestQueue`，这使得`RequestQueue`能够持续保持在你的app的生命周期中。

Set Up a Network and Cache

一个RequestQueue需要两部分来支持它的工作：一部分是网络操作用来执行请求的数据传输，另外一个是用来处理缓存操作的Cache。在Volley的工具箱中包含了标准的实现方式：DiskBasedCache提供了每个文件与对应响应数据一一映射的缓存实现。BasicNetwork提供了一个网络传输的实现，连接方式可以是[AndroidHttpClient](#) 或者是 [HttpURLConnection](#)。

BasicNetwork是Volley默认的网络操作实现方式。一个BasicNetwork必须使用HTTP Client进行初始化。这个Client通常是AndroidHttpClient 或者 HttpURLConnection：

- 对于app target API level低于API 9(Gingerbread)的使用AndroidHttpClient。在Gingerbread之前，HttpURLConnection是不可靠的。对于这个的细节，请参考[Android's HTTP Clients](#)。
- 对于API Level 9以及以上的，会使用HttpURLConnection。

为了创建一个能够执行在所有Android版本上的应用，你可以通过检查系统版本选择合适的HTTP Client。例如：

```
HttpStack stack;
...
// If the device is running a version >= Gingerbread...
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
    // ...use HttpURLConnection for stack.
} else {
    // ...use AndroidHttpClient for stack.
}
Network network = new BasicNetwork(stack);
```

下面的代码片段会掩饰如何一步步建立一个RequestQueue:

```
RequestQueue mRequestQueue;

// Instantiate the cache
Cache cache = new DiskBasedCache(getCacheDir(), 1024 * 1024); // 1MB cap

// Set up the network to use HttpURLConnection as the HTTP client.
Network network = new BasicNetwork(new HurlStack());

// Instantiate the RequestQueue with the cache and network.
mRequestQueue = new RequestQueue(cache, network);

// Start the queue
mRequestQueue.start();

String url = "http://www.myurl.com";

// Formulate the request and handle the response.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response) {
            // Do something with the response
        }
    },
    new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            // Handle error
        }
    });

// Add the request to the RequestQueue.
mRequestQueue.add(stringRequest);
...
```

如果你仅仅是想做一个单次的请求并且不想要线程池一直保留，你可以通过使用在前面一课：[发送一个简单的请求\(Sending a Simple Request\)](#)文章中提到Volley.newRequestQueue()方法在任何需要的时刻创建RequestQueue，然后在你的响应回调里面执行stop()方法来停止操作。但是更通常的做法是创建一个RequestQueue并设置为一个单例。下面将演示这种做法。

Use a Singleton Pattern

如果你的程序需要持续的使用网络，更加高效的方式应该是建立一个RequestQueue的单例，这样它能够持续保持在整个app的生命周期中。你可以通过多种方式来实现这个单例。推荐的方式是实现一个单例类，里面封装了RequestQueue对象与其他Volley的方法。另外一个方法是继承Application类，并在Application.onCreate()方法里面建立RequestQueue。但是这个方法是不推荐的。因为一个static的单例能够以一种更加模块化的方式提供同样的功能。

一个关键的概念是RequestQueue必须和Application context所关联的。而不是Activity的context。这可以确保RequestQueue可以在你的app生命周期中一直存活，而不会因为activity的重新创建而重新创建RequestQueue。（例如，当用户旋转设备时）。

下面是一个单例类，提供了RequestQueue与ImageLoader的功能：

```
private static MySingleton mInstance;
private RequestQueue mRequestQueue;
private ImageLoader mImageLoader;
private static Context mContext;

private MySingleton(Context context) {
    mContext = context;
    mRequestQueue = getRequestQueue();

    mImageLoader = new ImageLoader(mRequestQueue,
        new ImageLoader.ImageCache() {
            private final LruCache<String, Bitmap>
                cache = new LruCache<String, Bitmap>(20);

            @Override
            public Bitmap getBitmap(String url) {
                return cache.get(url);
            }

            @Override
            public void putBitmap(String url, Bitmap bitmap) {
                cache.put(url, bitmap);
            }
        });
}

public static synchronized MySingleton getInstance(Context context) {
    if (mInstance == null) {
        mInstance = new MySingleton(context);
    }
    return mInstance;
}

public RequestQueue getRequestQueue() {
    if (mRequestQueue == null) {
        // getApplicationContext() is key, it keeps you from leaking the
        // Activity or BroadcastReceiver if someone passes one in.
        mRequestQueue = Volley.newRequestQueue(mContext.getApplicationContext());
    }
    return mRequestQueue;
}

public <T> void addToRequestQueue(Request<T> req) {
    getRequestQueue().add(req);
}

public ImageLoader getImageLoader() {
    return mImageLoader;
}
}
```

下面演示了利用单例类来执行RequestQueue的操作：

```
// Get a RequestQueue
RequestQueue queue = MySingleton.getInstance(this.getApplicationContext()).
    getRequestQueue();
...

// Add a request (in this example, called stringRequest) to your RequestQueue.
MySingleton.getInstance(this).addToRequestQueue(stringRequest);
```

编写:[kesenhoo](#)(2014-06-24)

校对:

创建标准的网络请求(Making a Standard Request)

这一课会介绍如何使用Volley支持的常用请求类型：

- **StringRequest**。指定一个URL并在相应回调中接受一个原始的raw string数据。请参考前一课的示例。
- **ImageRequest**。指定一个URL并在相应回调中接受一个image。
- **JsonObjectRequest**与**JsonArrayRequest** (均为JsonRequest的子类)。指定一个URL并在相应回调中获取到一个JSON对象或者JSON数组。

如果你需要的是上面演示的请求类型，那么你应该不需要自己实现一个自定义的请求。这节课会演示如何使用那些标准的请求类型。关于如何实现自定义的请求，请看下一课：[实现自定义的请求](#)。

1)Request an Image

Volley为请求图片提供了如下的类。这些类依次有着依赖关系，用来支持在不同的层级进行图片处理：

- **ImageRequest** - 一个封装好的，用来处理URL请求图片并且返回一张decode好的bitmap的类。它同样提供了一些简便的接口方法，例如指定一个大小进行重新裁剪。它的主要好处是Volley回确保类似decode，resize等耗时的操作执行在工作线程中。
- **ImageLoader** - 一个用来处理加载与缓存从网络上获取到的图片的帮助类。ImageLoader是管理协调大量的ImageRequest的类。例如，在ListView中需要显示大量缩略图的时候。ImageLoader为通常的Volley cache提供了更加前瞻的内存缓存，这个缓存对于防止图片抖动非常有用。。这还使得能够在避免阻挡或者延迟主线程的前提下在缓存中能够被Hit到。ImageLoader还能够实现响应联合Coalescing，每一个响应回调里面都可以设置bitmap到view上面。联合Coalescing使得能够同时提交多个响应，这提升了性能。
- **NetworkImageView** - 在ImageLoader的基础上建立，替换ImageView进行使用。对于需要对ImageView设置网络图片的情况下使用很有效。NetworkImageView同样可以在view被detached的时候取消pending的请求。

1.1)Use ImageRequest

下面是一个使用ImageRequest的示例。它会获取指定URL的image并显示到app上。里面演示的RequestQueue是通过上一课提到的单例类实现的。

```
ImageView mImageView;
String url = "http://i.imgur.com/7spzG.png";
mImageView = (ImageView) findViewById(R.id.myImage);
...

// Retrieves an image specified by the URL, displays it in the UI.
ImageRequest request = new ImageRequest(url,
    new Response.Listener() {
        @Override
        public void onResponse(Bitmap bitmap) {
            mImageView.setImageBitmap(bitmap);
        }
    }, 0, 0, null,
    new Response.ErrorListener() {
        public void onErrorResponse(VolleyError error) {
            mImageView.setImageResource(R.drawable.image_load_error);
        }
    });
// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(request);
```

1.2)Use ImageLoader and NetworkImageView

你可以使用ImageLoader与NetworkImageView用来处理类似ListView等大量显示图片的情况。在你的layout XML文件中，你可以使用NetworkImageView来替代通常的ImageView，例如：

```
<com.android.volley.toolbox.NetworkImageView
    android:id="@+id/networkImageView"
    android:layout_width="150dp"
    android:layout_height="170dp"
    android:layout_centerHorizontal="true" />
```

你可以使用ImageLoader来显示一张图片，例如：

```
ImageLoader mImageLoader;
ImageView mImageView;
// The URL for the image that is being loaded.
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...
mImageView = (ImageView) findViewById(R.id.regularImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();
mImageLoader.get(IMAGE_URL, ImageLoader.getImageListener(mImageView,
    R.drawable.def_image, R.drawable.err_image));
```

然而，如果你要做得是为ImageView进行图片设置，你可以使用NetworkImageView来实现，例如：

```
ImageLoader mImageLoader;
```

```

NetworkImageView mNetworkImageView;
private static final String IMAGE_URL =
    "http://developer.android.com/images/training/system-ui.png";
...

// Get the NetworkImageView that will display the image.
mNetworkImageView = (NetworkImageView) findViewById(R.id.networkImageView);

// Get the ImageLoader through your singleton class.
mImageLoader = MySingleton.getInstance(this).getImageLoader();

// Set the URL of the image that should be loaded into this view, and
// specify the ImageLoader that will be used to make the request.
mNetworkImageView.setImageUrl(IMAGE_URL, mImageLoader);

```

上面的代码是通过前一节课的单例模式来实现访问到RequestQueue与ImageLoader的。之所以这样做得原因是：对于ImageLoader(一个用来处理加载与缓存图片的帮助类)来说，单例模式可以避免旋转所带来的抖动。使用单例模式可以使得bitmap的缓存与activity的生命周期无关。如果你在activity中创建ImageLoader，这个ImageLoader有可能会在手机进行旋转的时候被重新创建。这可能会导致抖动。

1.3)Example LRU cache

Volley工具箱中提供了通过DiskBasedCache实现的一种标准缓存。这个类能够缓存文件到磁盘的制定目录。但是为了使用ImageLoader，你应该提供一个自定义的内存LRC缓存，这个缓存需要实现ImageLoader.ImageCache的接口。你可能想把你的缓存设置成一个单例。关于更多的有关内容，请参考[建立请求队列Setting Up a RequestQueue](#)。

下面是一个内存LRU Cache的实例。它继承自LruCache并实现了ImageLoader.ImageCache的接口：

```

import android.graphics.Bitmap;
import android.support.v4.util.LruCache;
import android.util.DisplayMetrics;
import com.android.volley.toolbox.ImageLoader.ImageCache;

public class LruBitmapCache extends LruCache<String, Bitmap>
    implements ImageCache {

    public LruBitmapCache(int maxSize) {
        super(maxSize);
    }

    public LruBitmapCache(Context ctx) {
        this(getCacheSize(ctx));
    }

    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getRowBytes() * value.getHeight();
    }

    @Override
    public Bitmap getBitmap(String url) {
        return get(url);
    }

    @Override
    public void putBitmap(String url, Bitmap bitmap) {
        put(url, bitmap);
    }

    // Returns a cache size equal to approximately three screens worth of images.
    public static int getCacheSize(Context ctx) {
        final DisplayMetrics displayMetrics = ctx.getResources().
            getDisplayMetrics();
        final int screenWidth = displayMetrics.widthPixels;
        final int screenHeight = displayMetrics.heightPixels;
        // 4 bytes per pixel
        final int screenBytes = screenWidth * screenHeight * 4;

        return screenBytes * 3;
    }
}

```

下面是如何初始化ImageLoader并使用cache的实例:

```
RequestQueue mRequestQueue; // assume this exists.  
ImageLoader mImageLoader = new ImageLoader(mRequestQueue, new LruBitmapCache(LruBitmapCache.))
```

2)Request JSON

Volley提供了以下的类用来执行JSON请求：

- `JSONArrayRequest` - 一个为了获取JSONArray返回数据的请求。
- `JsonObjectRequest` - 一个为了获取JSONObject返回数据的请求。允许把一个JSONObject作为请求参数。

这两个类都是继承自`JsonRequest`的。你可以使用类似的方法来处理这两种类型的请求。如下演示了如果获取一个JSON feed并显示到UI上：

```
TextView mTxtDisplay;
ImageView mImageView;
mTxtDisplay = (TextView) findViewById(R.id.txtDisplay);
String url = "http://my-json-feed";

JsonObjectRequest jsonObjRequest = new JsonObjectRequest
    (Request.Method.GET, url, null, new Response.Listener() {

        @Override
        public void onResponse(JSONObject response) {
            mTxtDisplay.setText("Response: " + response.toString());
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) {
            // TODO Auto-generated method stub

        }
    });

// Access the RequestQueue through your singleton class.
MySingleton.getInstance(this).addToRequestQueue(jsonObjRequest);
```

关于基于[Gson](#)实现一个自定义的JSON请求对象，请参考下一节课：[实现一个自定义的请求Implementing a Custom Request](#).

编写:[kesenhoo](#)(2014-06-25)

校对:

实现自定义的网络请求Implementing a Custom Request

这节课会介绍如何实现你自定义的请求类型，这些自定义的类型不属于Volley内置支持包里面。

编写一个自定义的请求Write a Custom Request

大多数的请求类型都已经包含在Volley的工具箱里面。如果你的请求返回数值是一个string, image或者JSON, 那么你是不需要自己去实现请求类的。

对于那些你需要自定义的请求类型, 下面是你需要做得步骤:

- 继承Request<T>类, <T>表示了请求返回的数据类型。因此如果你需要解析的响应类型是一个String, 可以通过继承Request<String>来创建你自定义的请求。请参考Volley工具类中的StringRequest与 ImageRequest来学习如何继承Request。
- 实现抽象方法parseNetworkResponse()与deliverResponse(), 下面会详细介绍。

parseNetworkResponse

为了能够提交一种指定类型的数据(例如, string, image, JSON等), 需要对解析后的结果进行封装。下面会演示如何实现parseNetworkResponse()。

```
@Override
protected Response<T> parseNetworkResponse(
    NetworkResponse response) {
    try {
        String json = new String(response.data,
            HttpHeaderParser.parseCharset(response.headers));
        return Response.success(gson.fromJson(json, clazz),
            HttpHeaderParser.parseCacheHeaders(response));
    }
    // handle errors
    ...
}
```

请注意:

- parseNetworkResponse()的参数是类型是NetworkResponse, 这种参数包含了的响应数据内容有一个 byte[], HTTP status code以及response headers.
- 你实现的方法必须返回一个Response, 它包含了你响应对象与缓存metadata或者是一个错误。

如果你的协议没有标准的cache机制, 你可以自己建立一个Cache.Entry, 但是大多数请求都可以用下面的方式来处理:

```
return Response.success(myDecodedObject,
    HttpHeaderParser.parseCacheHeaders(response));
```

Volley在工作线程中执行parseNetworkResponse()方法。这确保了耗时的解析操作, 例如decode一张JPEG图片成bitmap, 不会阻塞UI线程。

deliverResponse

Volley会把parseNetworkResponse()方法返回的数据带到主线程的回调中。如下所示:

```
protected void deliverResponse(T response) {
    listener.onResponse(response);
}
```

Example: GsonRequest

[Gson](#)是一个使用映射支持JSON与Java对象之间相互转换的库文件。你可以定义和JSON keys想对应名称的Java对象。把对象传递给传递Gson, 然后Gson会帮你为对象填充字段值。下面是一个完整的示例: 演示了使用Gson解析Volley数据:

```
public class GsonRequest<T> extends Request<T> {
    private final Gson gson = new Gson();
    private final Class<T> clazz;
    private final Map<String, String> headers;
    private final Listener<T> listener;

    /**
     * Make a GET request and return a parsed object from JSON.
     *
     * @param url URL of the request to make
     * @param clazz Relevant class object, for Gson's reflection
     * @param headers Map of request headers
     */
    public GsonRequest(String url, Class<T> clazz, Map<String, String> headers,
        Listener<T> listener, ErrorListener errorListener) {
        super(Method.GET, url, errorListener);
    }
}
```



```

        this.clazz = clazz;
        this.headers = headers;
        this.listener = listener;
    }

    @Override
    public Map<String, String> getHeaders() throws AuthFailureError {
        return headers != null ? headers : super.getHeaders();
    }

    @Override
    protected void deliverResponse(T response) {
        listener.onResponse(response);
    }

    @Override
    protected Response<T> parseNetworkResponse(NetworkResponse response) {
        try {
            String json = new String(
                response.data,
                HttpHeaderParser.parseCharset(response.headers));
            return Response.success(
                gson.fromJson(json, clazz),
                HttpHeaderParser.parseCacheHeaders(response));
        } catch (UnsupportedEncodingException e) {
            return Response.error(new ParseError(e));
        } catch (JsonSyntaxException e) {
            return Response.error(new ParseError(e));
        }
    }
}

```

如果你愿意使用的话，Volley提供了现成的JsonArrayRequest与JsonArrayObject类。参考上一课[创建标准的网络请求](#)

编写:[kesenhoo](#)

校对:

云服务

学习如何同步与备份数据到云端以及如何在不同的设备上恢复备份的数据。

- [云同步：Syncing to the Cloud](#)
- [解决云同步的保存冲突：Resolving Cloud Save Conflicts](#)

编写:[kesenhoo](#), [jdneo](#)

校对:

云同步

通过为网络连接提供强大的APIs，Android Framework帮助你建立丰富的，具有云功能的App，这些App可以同步数据到远程服务器端，这使得所有你的设备都保持数据同步，并且重要的数据都能够备份在云端。

这章节会介绍几种不同的策略来实现具有云功能的App。这样用当用户安装你的app到新的一台设备上的时候能够恢复之前的使用记录。

Lessons

- [使用备份API](#)

学习如何集成Backup API到你的应用中。这样使得例如Preference，笔记与最高分记录等数据都能够无缝在用户的多台设备上进行同步更新。

- [使用Google Cloud Messaging](#)

学习如何高效的发送多路广播，如何正确的响应接收到的Google Cloud Messaging (GCM) 消息，以及如何使用GCM消息来与服务器进行同步。

编写:[kesenhoo](#)

校对:

Using the Backup API[使用Backup API]

当一个用户购买了新的设备或者是把当前的设备做了的恢复出厂设置的操作，用户希望在进行初始化设置的时候，Google Play能够把之前安装过的应用恢复到设备上。默认情况是，那些操作不会发生，用户之前的设置与数据都会丢失。

对于一些数据量相对较少的情况下(通常少于1MB)，例如用户偏好设置，笔记，游戏分数或者是其他的一些状态数据，可以使用Backup API来提供一个轻量级的解决方案。这一课会介绍如何使用Backup API。

1)Register for the Android Backup Service[为Android备份服务进行注册]

这一课会使用Android Backup Service, 它需要进行注册. 点击这个链接进行注册:[register here](#). 注册成功后, 服务器会提供一段类似下面的代码用来添加到程序的Manifest文件中:

```
<meta-data android:name="com.google.android.backup.api_key"
  android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
```

请注意, 每一个备份key都只能在特定的包名下工作, 如果你有不同的程序需要使用这个方法进行备份, 那么需要为他们分别进行注册。

2)Configure Your Manifest[确认你的Manifest]

使用Android的备份服务需要添加2个内容到你的程序Manifest中，首先，声明作为你的备份代理的类名，然后添加一段类似上面的代码作为Application标签的根标签。假设你的备份代理是TheBackupAgent, 下面演示里如何在Manifest中添加上面这些信息:

```
<application android:label="MyApp"
             android:backupAgent="TheBackupAgent">
    ...
    <meta-data android:name="com.google.android.backup.api_key"
             android:value="ABcDe1FGHij2KlMn3oPQRs4TUvw5xYZ" />
    ...
</application>
```

3)Write Your Backup Agent[编写你的备份代理]

最简单的创建你的备份代理的方法是继承[BackupAgentHelper](#). 创建这个帮助类实际上是非常简单。仅仅是创建一个你上面Manifest文件中声明的类去继承BackupAgentHelper.然后重写onCreate(). 在onCreate() 创建一个[BackupHelper](#). 目前Android framework包含了两种那样的帮助类: [FileBackupHelper](#) 与 [SharedPreferencesBackupHelper](#). 在你创建一个帮助类并且指向需要备份的数据的时候, 仅仅需要使用 addHelper() 方法来添加到BackupAgentHelper, 在后面再增加一个key用来retrieve数据. 大多数情况下, 完整的实现差不多只需要10行代码.

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.FileBackupHelper;

public class TheBackupAgent extends BackupAgentHelper {
    // The name of the SharedPreferences file
    static final String HIGH_SCORES_FILENAME = "scores";

    // A key to uniquely identify the set of backup data
    static final String FILES_BACKUP_KEY = "myfiles";

    // Allocate a helper and add it to the backup agent
    @Override
    void onCreate() {
        FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME);
        addHelper(FILES_BACKUP_KEY, helper);
    }
}
```

为了使得程序更加灵活, FileBackupHelper的constructor可以带有一些文件名, 你可以简单的通过增加一个额外的参数实现备份最高分文件与游戏程序文件, 像下面一样:

```
@Override
void onCreate() {
    FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME, PROGRESS_FILENAME);
    addHelper(FILES_BACKUP_KEY, helper);
}
```

备份用户偏好同样比较简单. 像创建FileBackupHelper一样来创建一个SharedPreferencesBackupHelper. 在这种情况下, 不是添加文件名到constructor,而是添加被你的程序所用的shared preference groups的名称.请看示例:

```
import android.app.backup.BackupAgentHelper;
import android.app.backup.SharedPreferencesBackupHelper;

public class TheBackupAgent extends BackupAgentHelper {
    // The names of the SharedPreferences groups that the application maintains. These
    // are the same strings that are passed to getSharedPreferences(String, int).
    static final String PREFS_DISPLAY = "displayprefs";
    static final String PREFS_SCORES = "highscores";

    // An arbitrary string used within the BackupAgentHelper implementation to
    // identify the SharedPreferencesBackupHelper's data.
    static final String MY_PREFS_BACKUP_KEY = "myprefs";

    // Simply allocate a helper and install it
    void onCreate() {
        SharedPreferencesBackupHelper helper =
            new SharedPreferencesBackupHelper(this, PREFS_DISPLAY, PREFS_SCORES);
        addHelper(MY_PREFS_BACKUP_KEY, helper);
    }
}
```

你可以根据你的喜好增加许多备份帮助类,但是请记住你仅仅需要为每一类添加一个既可。一个FileBackupHelper 处理了所有的你想要备份的文件, 一个SharedPreferencesBackupHelper 则处理了所有的你想要备份的shared preference groups.

4)Request a Backup[请求一个备份]

为了请求一个备份，仅仅需要创建一个BackupManager的实例，然后调用它的dataChanged() 方法既可。

```
import android.app.backup.BackupManager;
...

public void requestBackup() {
    BackupManager bm = new BackupManager(this);
    bm.dataChanged();
}
```

执行这个调用通知了backup manager 即将有数据会被备份到云端。在之后的某个时间点，backup manager会执行备份代理的onBackup() 方法。无论任何时候，只要你的数据有发生改变的都可以去调用它，不用担心会导致过度的网络活动。如果你在上一个备份还没有发生之前再次请求了备份，那么这个备份操作仅仅会出现一次。

5)Restore from a Backup[从备份中恢复]

通常是，你不应该手动去请求一个恢复，而是应该在你的程序安装到设备上的时候自动进行恢复。然而，如果那确实有必要手动去触发恢复，只需要调 `requestRestore()` 方法。

编写:[jdneo](#)

校对:

使用Google Cloud Messaging

谷歌云消息（GCM）是一个用来给Android设备发送消息的免费服务。GCM消息可以极大地提升用户体验。你的应用可以一直保持更新的状态而不会使你的设备在唤醒无线电或者在没有更新时对服务器发起询问而消耗电量。同时，GCM可以让你最多单一的消息可以发送给1000个人，使得你可以在恰当地时候很轻松地联系大量的用户，同时大量地减轻你的服务器负担。

这节课将包含将GCM集成到你的应用中的一些最佳实践方法，假定你已经对该服务的基本实现有了一个了解。如果不是这样的话，你可以先阅读一下：[GCM demo app tutorial](#)。

高效地发送多播消息

一个GCM所支持的最有用的特性是单条消息最多可以发送给1,000个接受者。这个功能可以更加简单地将重要消息发送给你的所有用户群体。例如，比方说你有一条消息需要发送给1,000,000个人，而你的服务器每秒能发送500条消息。如果你每次只给一个接受者发送消息，那么将会耗时 $1,000,000/500=2,000$ 秒，大约半小时。然而，如果一条消息可以一次性地发送给1,000个人的话，那么耗时将会是 $1,000,000/1,000/5,00=2$ 秒。这不仅仅体现在功能的实用性上，对于具有高时效性的消息而言，比如灾难预警或者体育比分播报，如果延迟了30分钟，消息的价值就大打折扣了。

想要利用这一功能非常简单。如果你使用Java的[GCM helper library](#)，只需要像“send”或者“sendNoRetry”方法提供一个注册ID的List就行了（不要只给单个的注册ID）。

```
// This method name is completely fabricated, but you get the idea.
List regIds = whoShouldISendThisTo(message);

// If you want the SDK to automatically retry a certain number of times, use the
// standard send method.
MulticastResult result = sender.send(message, regIds, 5);

// Otherwise, use sendNoRetry.
MulticastResult result = sender.sendNoRetry(message, regIds);
```

对于除了Java之外的语言，要实现GCM的支持，可以构建一个带有下列头部信息的HTTP POST请求：

- Authorization: key=YOUR_API_KEY
- Content-type: application/json

之后将你想要的参数编码成一个JSON对象，列出所有在“registration_ids”这个key下的注册ID。下面的代码片段是一个例子。除了“registration_ids”之外的所有参数都是可选的，在“data”内的项目代表了用户定义的载荷数据，而非GCM定义的参数。这个HTTP POST消息将会发送到：<https://android.googleapis.com/gcm/send>：

```
{ "collapse_key": "score_update",
  "time_to_live": 108,
  "delay_while_idle": true,
  "data": {
    "score": "4 x 8",
    "time": "15:16.2342"
  },
  "registration_ids":["4", "8", "15", "16", "23", "42"]
}
```

关于更多GCM多播的消息格式，可以阅读：[Sending Messages](#)。

对可替换的消息执行折叠

GCM经常被用作作为一个触发器，告诉移动应用向服务器发起链接并刷新数据。在GCM中，可以（也推荐）在新消息要替代旧消息时，使用可折叠的消息（collapsible messages）。我们用体育比赛作为例子，如果你向所有用户发送了一条消息包含了比赛的比分，然后再15分钟后，又发送了一条消息更新新的比分，那么第一条消息就没有意义了。对于那些还没有收到第一条消息的用户，就没有必要接收两条消息，并且如果接收了两条消息，那么设备不得不响应两次（比如对用户发出通知或警告），但实际上两条消息中只有一条是重要的。

当你定义了一个折叠键，如果多个消息在GCM服务器中，对于相同的用户形成了一个队列，那么只有最后的那一条消息会被发出。对于之前所说的体育比分的例子，这样做能让设备免于处理不必要的任务，也不会让设备对用户造成太多打扰。对于其他的一些场景比如与服务器同步数据（检查邮件接收），这样做的话可以减少设备需要执行的同步次数。例如，如果有10封邮件在服务器中等待被接收，那么实际上只需要发送一个GCM，让设备一次性把10封邮件都同步了。

为了使用这一特性，只需要在你发出的消息中添加一个消息折叠key。如果你在使用GCM帮助库，那么就使用Message类的collapseKey(String key)方法。

```
Message message = new Message.Builder(regId)
    .collapseKey("game4_scores") // The key for game 4.
    .ttl(600) // Time in seconds to keep message queued if device offline.
    .delayWhileIdle(true) // Wait for device to become active before sending.
    .addPayload("key1", "value1")
    .addPayload("key2", "value2")
    .build();
```

如果你没有使用帮助库，那么就直接在你要构建的POST头部中添加一个变量。collapse_key作为变量名，你要更新的字段以字符串的形式作为值。

在GCM消息中嵌入数据

通常，GCM消息作为一个激活器，或者用来告诉设备，有一些待更新的数据需要去服务器或者别的地方去获取。然而，一个GCM消息的大小最大可以有4kb，有时候可以在GCM消息中放置一些简单的数据，这样的话设备就不需要再去和服务器发起连接了。在下列情形都满足的情况下，我们可以将数据放置在GCM消息中：

- 数据的总大小在4kb以内。
- 每一条消息都很重要，应该保留。
- 这些消息不适用于消息折叠的使用情形。

例如，短消息或者回合制网游中玩家的移动数据等都是将数据直接嵌入在GCM消息中的例子。而电子邮件就是反面例子了。因为电子邮件的数据量一般都大于4kb，且用户不需要对每个邮件都收到一个GCM提醒的消息。

同时在发送多播消息时，也可以考虑这一方法，这样的话就不会导致大量用户在接收到GCM的更新提醒后，同时向你的服务器发起连接。

这一策略不适用于发送大量的数据（你可能会想要中这个方法将数据分割后发送），有这么一些原因：

- 为了防止恶意软件发送垃圾消息，GCM有发送频率的限制。
- 无法保证消息按照既定的顺序到达。
- 无法保证消息可以在你发送后立即到达。假设设备每一秒都接收一条消息，最大为1K，即8kbps，或者说是1990年代的家庭拨号上网的速度。那么如此大量的消息，一定会让你的应用在Google Play上的评分非常尴尬。

如果恰当地使用，直接将数据嵌入到GCM消息中，可以加速你的应用的“感知速度”，因为它不必再去服务器获取数据了。

智能地响应GCM消息

你的应用不应该仅仅对收到的GCM消息进行响应就够了，还应该响应地更智能一些。至于要如何响应需要结合具体情况而定。

不要太过激进

当提醒用户更新数据时，很容易不小心从“有用的消息”变成“干扰消息”。如果你的应用使用状态栏通知，那么应该[更新现有的通知](#)，而不是创建第二个。如果你通过铃声或者震动的方式提醒用户，一定要设置一个计时器。不要让应用的提醒时间超过1分钟，不然的话用户很可能为不堪其扰而卸载你的应用，关机，或者把手机扔到河里：)

用聪明的办法同步数据，别用笨办法

当使用GCM告知设备有数据需要从服务器下载时，记住你有4kb的数据大小和消息一起发出，这可以帮助你的应用做出更智能地响应。例如，如果你有一个源阅读应用，而你的用户订阅了100个源，那么这就可以帮助你的应用更智能地决定应该去服务器下载什么数据。下面的例子说明了在GCM载荷中可以发送那些数据，以及设备可以做出什么样的反应：

- refresh - 你的应用被告知向每一个源请求数据。此时你的应用可以向100个不同的服务器发起获取源的请求，或者如果你在你的服务器上有一个聚合服务，那么可以只发送一个请求，将100个源的数据进行打包并获取，这样一次性完成更新。
- refresh, freshID - 一种更好的解决方案，你的应用可以有针对性的完成更新。
- refresh, freshID, timestamp - 一种更好的解决方案，如果正好用户在GCM消息收到之前手动做了更新，那么应用可以利用时间戳和当前的更新时间进行对比，并决定是否要执行下一步的行动。

编写:[jdneo](#)

校对:

解决云同步的保存冲突

这篇文章介绍了当应用使用[Cloud Save service](#)存储数据到云端时，如何设计一个鲁棒性较高的冲突解决策略。云存储服务允许你为每一个在Google服务上的应用用户，存储他们的应用数据。你的应用可以通过使用云存储API，从Android设备，iOS设备或者web应用恢复或更新这些数据。

在云存储过程中的保存和加载是很直接的：它只是一个数据和byte数组之间的相互转换，并将这些数组存储在云端。然而，当你的用户有多个设备，并且两个以上的设备尝试将它们的数据存储在云端时，这一保存可能会引起冲突，因此你必须决定应该如何处理。你在云端存储的数据结构在很大程度上决定了你的冲突解决方案的鲁棒性，所以小心地设计你的数据，使得你的冲突检测解决方案的逻辑可以正确地处理每一种情况。

本篇文章从描述一些有缺陷的方法入手，并解释他们为何具有缺陷。之后呈现一个解决方案来避免冲突。用于讨论的例子关注于游戏，但解决问题的宗旨是可以适用于任何将数据存储于云端的应用的。

冲突时获得通知

[OnStateLoadedListener](#)方法负责从Google服务器下载应用的状态数据。回调函数[OnStateLoadedListener.onStateConflict](#)为你的应用在本地状态和云端存储的状态发生冲突时，提供了一个解决机制：

```
@Override
public void onStateConflict(int stateKey, String resolvedVersion,
    byte[] localData, byte[] serverData) {
    // resolve conflict, then call mAppStateClient.resolveConflict()
    ...
}
```

此时你的应用必须决定要保留哪一个数据，或者它自己提交一个新的数据来表示合并后的数据状态，解决冲突的逻辑由你来实现。

我们必须意识到云存储服务是在后台执行同步的。所以你应该确保你的应用能够在创建这一数据的context之外接收回调。特别地，如果Google Play服务应用在后台检测到了一个冲突，该回调函数可以在你下一次加载数据时被调用，而不是下一次用户启动该应用时。

因此，你的云存储代码和冲突解决代码的设计必须是和当前context无关的：即给两个冲突的数据，你必须仅通过数据集中获取的数据区解决冲突，而不依赖于任何其它外部环境。

处理简单情况

下面列举一些冲突解决的简单例子。对于很多应用而言，用这些策略或者其变体就足够解决大多数问题了：

新的比旧的更有效：在一些情况下，新的数据总是替代老数据。例如，如果数据代表了用户选择角色的衣服颜色，那么最近的新的选择就应该覆盖老的选择。在这种情况下，你可能会选择在云存储数据中存储时间戳。当处理这些冲突时，选择时间戳最新的数据（记住要选择一个可靠的时钟，并注意对不同时区的处理）。

一个数据好于其他数据：在一些情况下，我们是可以有方法在若干数据集中选取一个最好的。例如，如果数据代表了玩家在赛车比赛中的最佳时间，那么显然，在冲突发生时，你应该保留成绩最好的那个数据。

进行合并：有可能通过计算两个数据集的合并版本来解决冲突。例如，如果你的数据代表了用户解锁关卡的进度，那么解决的数据就是冲突集的并集。通过这个方法，用户不会丢失任何他的游戏进度。这里的[例子](#)使用了这一操作的一个变形。

为更复杂的情况设计一个策略

一个更复杂的情况是当你的游戏允许玩家收集可以互换的东西时（比如金币或者经验点数），我们来假想一个游戏，叫做“金币跑酷”，一个无限跑步的角色，其目标是不断地收集金币是自己变的富有。每个收集到的金币都会加入到玩家的储蓄罐中。

下面的章节将展示三种在多个设备间解决冲突的方案：有两个听上去很不错，可惜最终还是不能适用于所有的场景，最后一个解决方案可以解决多个设备间的冲突。

第一个尝试：只保存总数

首先，这个问题看上去像是说：云存储的数据只要存储金币的数量就行了。但是如果就只有这些数据是可用的，那么解决冲突的方案将会严重受到限制。此时最佳的方案就是在冲突发生时存储数值最大的数据。

想一下表一中所展现的场景。假设玩家一开始有20枚硬币，然后再设备A上收集了10个，在设备B上收集了15个。然后设备B将数据存储到了云端。当设备A尝试去存储的时候，冲突发生了。“只存储总数”的冲突解决方案会存储35作为这一数据的值（两数之间最大的）。

表1. 值保存最大的数（不佳的策略）

事件	设备A的数据	设备B的数据	云端的数据	实际的总数
开始阶段	20	20	20	20
玩家在A设备上收集了10个硬币	30	20	20	30
玩家在B设备上收集了15个硬币	30	35	20	45
设备B将数据存储至云端	30	35	35	45
设备A尝试将数据存储至云端，发生冲突	30	35	35	45
设备A通过选择两数中最大的数来解决冲突	35	35	35	45

这一策略会失败：玩家的金币数从20变成35，但实际上玩家总共收集了25个硬币（A设备10个，B设备15个）。所以有10个硬币丢失了。只在云端存储硬币的总数是不足以实现一个鲁棒的冲突解决算法的。

第二个尝试：存储总数和变化值

另一个方法是在存储数据中包括一些额外的数据：自上次提交后硬币增加的数量（delta）。在这一方法中，存储的数据可以用一个二元组来表示（T, d），其中T是硬币的总数，而d是硬币增加的数量。

在这个结构中，你的冲突检测算法在鲁棒性上有更大的提升空间。但是这个方法还是无法给出一个可靠的玩家最终的状态。

下面是包含delta的冲突解决算法过程：

- 本地数据：（T, d）
- 云端数据：（T', d'）
- 解决后的数据：（T'+d, d）

例如，当你在本地状态（T, d）和云端状态（T', d'）之间发生了冲突时，你可以将它们合并成（T'+d, d）。意味着你从本地拿出delta数据，并将它和云端的数据结合起来，乍一看，这种方法可以很好的计量多个设备所收集的金币。

看上去很可靠的方法，但这个方法在移动环境中难以适用：

- 用户可能在设备不在线时存储数据。这些改变会以队列形式等待手机联网后提交。
- 这个方法的同步机制是用最新的变化覆盖掉任何之前的变化。换句话说，第二次写入的变化会提交到云端（当设备联网了以后），而第一次写入的变化就被忽略了。

为了进一步说明，我们考虑一下表2所列的场景。在表2的一系列操作后，云端的状态将是（130, +5），之后最终冲突解决后的状态时（140, +10）。这是不正确的，因为从总体上而言，用户一共在A上收集了110枚硬币而在B上收集了120枚硬币。总数应该为250。

表2. “总数+增量”策略的失败案例

事件	设备A的数据	设备B的数据	云端的数据	实际的总数
开始阶段	(20, x)	(20, x)	(20, x)	20
玩家在A设备上收集了100个硬币	(120, +100)	(20, x)	(20, x)	120
玩家在A设备上又收集了10个硬币	(130, +10)	(20, x)	(20, x)	130
玩家在B设备上收集了115个硬币	(130, +10)	(125, +115)	(20, x)	245
玩家在B设备上又收集了5个硬币	(130, +10)	(130, +5)	(20, x)	250
设备B将数据存储至云端	(130, +10)	(130, +5)	(130, +5)	250
设备A尝试将数据存储至云端，发生冲突	(130, +10)	(130, +5)	(130, +5)	250
设备A通过将本地的增量和云端的总数相加来解决冲突	(140, +10)	(130, +5)	(140, +10)	250

注：x代表与该场景无关的数据

你可能会尝试在每次保存后不重置增量数据来解决此问题，这样的话在每个设备上的第二次存储所收集到的硬币将不会产生问题。这样的话设备A在第二次本地存储完成后，数据将是（130, +110）而不是（130, +10）。然而，这样做的话就会发生如表3所述的情况：

表3. 算法改进后的失败案例

事件	设备A的数据	设备B的数据	云端的数据	实际的总数
开始阶段	(20, x)	(20, x)	(20, x)	20
玩家在A设备上收集了100个硬币	(120, +100)	(20, x)	(20, x)	120
设备A将状态存储到云端	(120, +100)	(20, x)	(120, +100)	120
玩家在A设备上又收集了10个硬币	(130, +110)	(20, x)	(120, +100)	130
玩家在B设备上收集了1个硬币	(130, +110)	(21, +1)	(120, +100)	131
设备B尝试向云端存储数据，发生冲突	(130, +110)	(21, +1)	(120, +100)	131
设备B通过将本地的增量和云端的总数相加来解决冲突	(130, +110)	(121, +1)	(121, +1)	131
设备A尝试将数据存储至云端，发生冲突	(130, +110)	(121, +1)	(121, +1)	131
设备A通过将本地的增量和云端的总数相加来解决冲突	(231, +110)	(121, +1)	(231, +110)	131

注：x代表与该场景无关的数据

现在你碰到了另一个问题：你给予了玩家过多的硬币。这个玩家拿到了211枚硬币，但实际上他只收集了111枚。

解决办法：

分析之前的几次尝试，我们发现这些策略都没有这样一个能力：知晓哪些硬币已经计数了，哪些硬币没有被计数，尤其是当多个设备连续提交的时候，算法会出现混乱。

该问题的解决办法将你云端的存储结构改为字段，使用字符串+整形的键值对。每一个键值对都会代表一个包含硬币的“委托”，而总数就应该是将所有值加起来。这一设计的宗旨是每个设备有它自己的委托，并且只有设备自己可以把硬币放到其委托中。

字典的结构是：(A:a, B:b, C:c, ...)，其中a代表了委托A所拥有的硬币，b是委托B所拥有的硬币，以此类推。

这样的话，新的冲突解决策略算法将如下所示：

- 本地数据：(A:a, B:b, C:c, ...)
- 云端数据：(A:a', B:b', C:c', ...)
- 解决后的数据：(A:max(a,a'), B:max(b,b'), C:max(c,c'), ...)

例如，如果本地数据是(A:20, B:4, C:7)并且云端数据是(B:10, C:2, D:14)，这样的话解决冲突后的数据将会是(A:20, B:10, C:7, D:14)。注意，你应用的冲突解决逻辑会根据具体的场景可能有所差异。比如，有一些应用你可能希望挑选最小的值。

为了测试新的算法，将它应用于任何一个之前提到过的场景。你将会发现它都能取得正确地结果。

表4阐述了这一点，它基于表3的场景。注意下面所列的：

在初始状态，玩家有20枚硬币。此数值在所有设备和云端都是正确的，我们用（X:20）这一元组代表它，其中X我们不用太多关心，我们不去追求这个初始化的数据是哪儿来的。

当玩家在设备A上收集了100枚硬币，这一变化会作为一个元组保存到云端。它的值是100是因为这就是玩家在设备A上收集的硬币数量。在这一过程中，没有要执行数据的计算（设备A仅仅是将玩家所收集的数据汇报给了云端）。

每一个新的硬币提交会打包成一个于设备关联的元组并保存到云端。例如，假设玩家又在设备A上收集了100枚硬币，那么元组的值被更新为110。

最终的结果就是，应用知道了玩家在每个设备上收集硬币的总数。这样它就能轻易地计算总数了。

表4. 键值对策略的成功应用案例

事件	设备A的数据	设备B的数据	云端的数据	实际的总数
开始阶段	(X:20, x)	(X:20, x)	(X:20, x)	20
玩家在A设备上收集了100个硬币	(X:20, A:100)	(X:20)	(X:20)	120
设备A将状态存储到云端	(X:20, A:100)	(X:20)	(X:20, A:100)	120
玩家在A设备上又收集了10个硬币	(X:20, A:110)	(X:20)	(X:20, A:100)	130
玩家在B设备上收集了1个硬币	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
设备B尝试向云端存储数据，发生冲突	(X:20, A:110)	(X:20, B:1)	(X:20, A:100)	131
设备B解决冲突	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
设备A尝试将数据存储至云端，发生冲突	(X:20, A:110)	(X:20, A:100, B:1)	(X:20, A:100, B:1)	131
设备A解决冲突	(X:20, A:110, B:1)	(X:20, A:100, B:1)	(X:20, A:110, B:1), total 131	131

清除你的数据

在云端存储数据的大小是有限制的，所以在后续的论述中，我们将会关注与如何避免创建过大的词典。一开始，看上去每个设备只会会有一个词典字段，即使是非常激进的用户也不太会拥有上千条字段。然而，获取设备ID的方法很难，并且我们认为这是一种不好的实践方式，所以你应该使用一个安装ID，这更容易获取也更可靠。这样的话就意味着，每一次用户在每台设备安装一次就会产生一个ID。假设每个键值对占据32字节，由于一个个人云存储缓存最多可以有128K的大小，那么你最多可以存储4096个字段。

在现实场景中，你的数据可能更加复杂。在这种情况下，存储的数据字段数也会进一步受到限制。具体而言则需要取决于实现，比如可能需要添加时间戳来指明每个字段是何时修改的。当你检测到有一个字段在过去几个礼拜或者几个月的时间内都没有被修改，那么就可以安全地将它转移到另一个字段中并删除老的字段。

编写:[spencer198711](#)

校对:

用户信息

编写:[spencer198711](#)

校对:

获取联系人列表

这一课展示了如何根据要搜索的字符串去匹配联系人的数据，从而得到联系人列表，你可以使用以下方法去实现：

匹配联系人名字

根据搜索字符串来匹配部分或者全部联系人的名字来获得联系人列表，联系人数据库允许多个人拥有相同的名字，所以这种方法能够取得相匹配的列表。

匹配特定的数据类型，比如电话号码

根据搜索字符串来匹配联系人的特定类型的数据，比如电子邮件，来取得符合要求的联系人列表。比如说，这种方法可以让你取得联系人列表，他们的电子邮件于搜索字符相匹配。

匹配任意类型的数据

根据搜索字符串来匹配联系人详情的所有类型的数据，包括名字、电话号码、地址、电子邮件地址等等。比如说，这种方法可以让你根据任意类型的数据，去获取与联系人详情数据相匹配的列表。

提示：这一课的所有例子都使用了CursorLoader去从ContactsProvider中获取数据。CursorLoader在一个工作线程中去运行查询操作，使得能够与UI线程分开，这保证了数据查询不会降低UI响应的时间，以免引起糟糕的用户体验。更多信息，请参照在后台加载数据。

请求读取联系人的权限

为了能够在联系人数据库中做任意类型的搜索，你的应用必须拥有READ_CONTACTS的权限，为了拥有这个权限，你需要向项目的清单文件中添加以下结点作为的子结点

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

根据名字取得联系人并列出结果列表

这种方法试图通过根据一个搜索字符串，去匹配联系人数据库ContactsContract.Contacts表中的联系人名字，从而取得一个或者多个联系人。通常希望在ListView中展示结果，去让用户在所有匹配的联系人中做选择。

定义列表和列表项的布局

为了能够将搜索结果展示在列表中，你需要一个包含ListView以及其他布局控件的主布局文件，和定义列表中每一项的布局文件。例如，你可以使用以下的XML代码去创建主布局文件res/layout/contacts_list_view.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

这个XML代码使用了Android内建的ListView控件,他的id是android:id/list。

使用以下XML代码定义列表项布局文件contacts_list_item.xml：

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:clickable="true"/>
```

这个XML代码使用了Android内建的TextView控件,他的id是android:text1。

提示：本课并不会描述如何从用户那里获取搜索字符串的界面，因为你可能会间接地获取这个字符串。比如说，你可能会给用户一个选项，让他从收到的短信中的部分内容作为名字去搜索匹配的联系人。

刚刚写的这两个布局文件定义了展示在ListView的用户界面，下一步是编写使用这些界面显示联系人列表的代码。

定义显示联系人列表的Fragment

为了显示联系人列表，需要定义一个由Activity加载的Fragment。使用Fragment是一个比较灵活的方法，因为你可以使用一个Fragment去显示列表，当用户选择列表的中的某一个联系人的时候，用第二个Fragment显示此联系人的详情。使用这种方式，你可以结合本课程中展示的方法和另外一课“获取联系人详情”。

想要学习如何在Activity中使用一个或者多个Fragment，请阅读培训课程“使用Fragment构建灵活的用户界面”。

为了帮你编写对联系人数据库的查询，android框架提供了一个叫做ContactsContract的契约类，这个类定义了一些对查询数据库很有用的常量和方法。当你使用这个类的时候，你不用自己定义内容URI、表名、列名等常量。使用这个类，你需要引入以下类声明：

```
import android.provider.ContactsContract;
```

由于代码中使用了CursorLoader去从provider中获取数据，你必须实现加载器接口LoaderManager.LoaderCallbacks。同时，为了检测用户从结果列表中选择了哪一个联系人，必须实现适配器接口AdapterView.OnItemClickListener。例如：

```
...
import android.support.v4.app.Fragment;
import android.support.v4.app.LoaderManager.LoaderCallbacks;
import android.widget.AdapterView;
...
public class ContactsFragment extends Fragment implements
    LoaderManager.LoaderCallbacks<Cursor>,
    AdapterView.OnItemClickListener {
```

定义全局变量

定义在其他代部分码中使用的全局变量：

```
...
/*
 * Defines an array that contains column names to move from
 * the Cursor to the ListView.
 */
```



```

@SuppressLint("InlinedApi")
private final static String[] FROM_COLUMNS = {
    Build.VERSION.SDK_INT
        >= Build.VERSION_CODES.HONEYCOMB ?
        Contacts.DISPLAY_NAME_PRIMARY :
        Contacts.DISPLAY_NAME
};
/*
 * Defines an array that contains resource ids for the layout views
 * that get the Cursor column contents. The id is pre-defined in
 * the Android framework, so it is prefaced with "android.R.id"
 */
private final static int[] TO_IDS = {
    android.R.id.text1
};
// Define global mutable variables
// Define a ListView object
ListView mContactsList;
// Define variables for the contact the user selects
// The contact's _ID value
long mContactId;
// The contact's LOOKUP_KEY
String mContactKey;
// A content URI for the selected contact
Uri mContactUri;
// An adapter that binds the result Cursor to the ListView
private SimpleCursorAdapter mCursorAdapter;
...

```

提示：由于Contacts.DISPLAY_NAME_PRIMARY需要在android 3.0（API版本11）之后才能使用，如果你的应用的minSdkVersion是10或者更小，会在eclipse中产生警告信息。为了关闭这个警告，你可以在FROM_COLUMNS定义之前加上@SuppressLint("InlinedApi")注解。

初始化Fragment

为了初始化Fragment，android系统需要你为这个Fragment添加空的、公有的构造方法，同时在回调方法onCreateView()中绑定界面。例如：

```

// Empty public constructor, required by the system
public ContactsFragment() {}
// A UI Fragment must inflate its View
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // Inflate the fragment layout
    return inflater.inflate(R.layout.contact_list_fragment,
        container, false);
}

```

为ListView绑定CursorAdapter数据

将绑定到搜索结果的SimpleCursorAdapter设置到ListView。为了获得显示联系人列表的ListView控件，需要使用Fragment的父Activity调用Activity.findViewById()。当你调用setAdapter()的时候，需要使用父Activity的上下文（Context）。

```

public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    ...
    // Gets the ListView from the View list of the parent activity
    mContactsList =
        (ListView) getActivity().findViewById(R.layout.contact_list_view);
    // Gets a CursorAdapter
    mCursorAdapter = new SimpleCursorAdapter(
        getActivity(),
        R.layout.contact_list_item,
        null,
        FROM_COLUMNS, TO_IDS,
        0);
    // Sets the adapter for the ListView
    mContactsList.setAdapter(mCursorAdapter);
}

```

为选择的联系人设置监听器

当你显示搜索列表结果的时候，你通常会让用户选择某一个联系人去做进一步的处理。例如，当用户选择某一个联系人的时候，你可以在地图上显示这个人的地址。为了能够提供这个功能。你需要定义当前的Fragment为一个点击监听器，这需要这个类实现AdapterView.OnItemClickListener接口，就像“定义显示联系人列表的Fragment”那一节展示的那样。

继续设置这个监听器，需要在onActivityCreated()方法中调用setOnItemClickListener()以使得这个监听器绑定到ListView。例如：

```
public void onActivityCreated(Bundle savedInstanceState) {  
    ...  
    // Set the item click listener to be the current fragment.  
    mContactsList.setOnItemClickListener(this);  
    ...  
}
```

由于指定了当前的Fragment作为ListView的点击监听器，现在你需要实现处理点击事件的onItemClick()方法。这个会在随后讨论。

定义查询映射

定义一个常量，这个常量是你想要的查询返回值所包含的列。Listview中得每一行显示了一个联系人的“显示名字”，它包含了联系人名字的主要部分。在android 3.0之后，这个列的名字是Contacts.DISPLAY_NAME_PRIMARY,在android 3.0之前，这个列的名字是Contacts.DISPLAY_NAME。

Contacts._ID列在SimpleCursorAdapter绑定过程中会用到。Contacts._ID和LOOKUP_KEY一同用来构建用户选择的联系人的内容URI。

```
...  
@SuppressWarnings("InlinedApi")  
private static final String[] PROJECTION =  
{  
    Contacts._ID,  
    Contacts.LOOKUP_KEY,  
    Build.VERSION.SDK_INT  
        >= Build.VERSION_CODES.HONEYCOMB ?  
        Contacts.DISPLAY_NAME_PRIMARY :  
        Contacts.DISPLAY_NAME  
};
```

定义Cursor的列索引常量

为了从Cursor中获得单独某一列的数据，你需要知道这一列在Cursor中的索引值。你需要定义Cursor列的索引值，这些索引值同你定义的查询映射的列的顺序是一样的。例如：

```
// The column index for the _ID column  
private static final int CONTACT_ID_INDEX = 0;  
// The column index for the LOOKUP_KEY column  
private static final int LOOKUP_KEY_INDEX = 1;
```

指定查询标准

为了指定你想要查询的数据，你需要创建一个包含字符串表达式和变量组成的条件，去告诉provider你需要的数据列和想要的值。

对于字符串表达式，你需要定义一个所有列要满足的条件的常量。尽管这个表达式可以包含变量值，但是一个比较好的建议是用"?"占位符来替代这个值，在搜索的时候，占位符里的值会被数组里的值所取代。使用"?"占位符确保了搜索条件是由绑定产生而不是有SQL编译产生。这条实践消除了恶意SQL注入的可能。例如：

```
// Defines the text expression  
@SuppressWarnings("InlinedApi")  
private static final String SELECTION =  
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?  
    Contacts.DISPLAY_NAME_PRIMARY + " LIKE ?" :  
    Contacts.DISPLAY_NAME + " LIKE ?";  
// Defines a variable for the search string  
private String mSearchString;  
// Defines the array to hold values that replace the ?  
private String[] mSelectionArgs = { mSearchString };
```

定义onItemClick()方法

在之前的内容中，你为ListView设置了列表项点击监听器，现在需要定义AdapterView.OnItemClickListener.onClick()方法以实现监听器行为：

```
@Override
public void onItemClick(
    AdapterView<?> parent, View item, int position, long rowID) {
    // Get the Cursor
    Cursor cursor = parent.getAdapter().getCursor();
    // Move to the selected contact
    cursor.moveToPosition(position);
    // Get the _ID value
    mContactId = getLong(CONTACT_ID_INDEX);
    // Get the selected LOOKUP KEY
    mContactKey = getString(CONTACT_KEY_INDEX);
    // Create the contact's content Uri
    mContactUri = Contacts.getLookupUri(mContactId, mContactKey);
    /*
     * You can use mContactUri as the content URI for retrieving
     * the details for a contact.
     */
}
```

初始化loader

由于使用了CursorLoader获取数据，你必须初始化后台线程和其他的控制异步获取数据的变量。需要在onActivityCreated()方法中做初始化的工作，这个方法是在Fragment的界面显示之前调用的，相关代码展示如下：

```
public class ContactsFragment extends Fragment implements
    LoaderManager.LoaderCallbacks<Cursor> {
    ...
    // Called just before the Fragment displays its UI
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        // Always call the super method first
        super.onActivityCreated(savedInstanceState);
        ...
        // Initializes the loader
        getLoaderManager().initLoader(0, null, this);
    }
}
```

实现onCreateLoader()方法

你需要实现onCreateLoader()方法，这个方法是在你调用initLoader后被loader框架直接调用的。

在onCreateLoader()方法中，设置搜索字符串模式。为了让一个字符串符合一个模式，可以插入"%"字符代表0个或多个字符或者插入"_"代表单独一个字符。例如，模式%Jefferson%将会匹配“Thomas Jefferson”和“Jefferson Davis”。

这个方法返回一个CursorLoader对象。对于内容URI，则使用了Contacts.CONTENT_URI，这个URI关联到整个表，例子如下所示：

```
...
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    /*
     * Makes search string into pattern and
     * stores it in the selection array
     */
    mSelectionArgs[0] = "%" + mSearchString + "%";
    // Starts the query
    return new CursorLoader(
        getActivity(),
        Contacts.CONTENT_URI,
        PROJECTION,
        SELECTION,
        mSelectionArgs,
        null
    );
}
```

实现onLoadFinished()方法和onLoaderReset()方法

实现onLoadFinished()方法。当联系人provider返回查询结果的时候，Android loader框架会调用onLoadFinished()方法。在这个方法中，将查询结果Cursor传给SimpleCursorAdapter，这将会使用这个搜索结果自动更新ListView。

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {  
    // Put the result Cursor in the adapter for the ListView  
    mCursorAdapter.swapCursor(cursor);  
}
```

当loader框架检测到结果集Cursor包含过时的数据时，它会调用onLoaderReset()。你需要删除SimpleCursorAdapter对已经存在Cursor的引用。如果不这么做的话，loader框架将不会回收Cursor对象，这将会导致内存泄漏。例如：

```
@Override  
public void onLoaderReset(Loader<Cursor> loader) {  
    // Delete the reference to the existing Cursor  
    mCursorAdapter.swapCursor(null);  
}
```

你已经实现了根据搜索字符串匹配联系人名字，并将获得的结果展示在ListView中的关键部分。用户可以点击选择一个联系人名字，这将会触发一个监听器，在监听器的回调函数中，你可以使用此联系人的数据做进一步的处理。例如，你可以进一步获取此联系人的详情，想要知道如何获取联系人详情，请继续学习下一课——获取联系人详情。

想要了解更多搜索用户界面的知识，请参考API指导——创建搜索界面。

这一课的以下内容展示了在联系人数据库中查找联系人的其他方法。

根据特定类型的数据匹配联系人

这种方法可以让你指定你想要匹配的数据类型，根据名字去检索是这种类型的查询的一个具体的例子。但也可以用任何与联系人详情数据相关的数据类型去做查询。例如，您可以检索具有特定邮政编码联系人，在这种情况下，搜索字符串将会去匹配存储在一个邮政编码列中的数据。

为了实现这种类型的检索，首先实现以下的代码，正如之前的内容所展示的：

- 请求读取联系人的权限
- 定义列表和列表项的布局
- 定义显示联系人列表的Fragment
- 定义全局变量
- 初始化Fragment
- 为ListView绑定CursorAdapter数据
- 设置选择联系人的监听器

- 定义Cursor的列索引常量

尽管你现在从不同的表中取数据，检索列的映射顺序是一样的，所以你可以为这个Cursor使用同样的索引常量。

- 定义onItemClickListener方法
- 初始化loader
- 实现onLoadFinished()方法和onLoaderReset()方法

为了将搜索字符串匹配特定类型的详请数据并显示结果，以下的步骤展示了你需要做的额外的代码。

选择要查询的数据类型和数据库表

为了从特定类型的详请数据中查询，你必须知道的数据类型的自定义MIME类型的值。每一个数据类型拥有唯一的MIME类型值，这个值在ContactsContract.CommonDataKinds的子类中被定义为常量CONTENT_ITEM_TYPE，并且与实际的数据类型相关。子类的名字会表明它们的实际数据类型，例如，email数据的子类是ContactsContract.CommonDataKinds.Email，并且email的自定义MIME类型是Email.CONTENT_ITEM_TYPE。

在你的搜索中需要使用ContactsContract.Data类，同时所有需要的常量，包括数据映射、选择字句、排序规则都是由这个类定义或继承自此类。

定义查询映射

为了定义一个查询映射，请选择一个或者多个由ContactsContract.Data或其子类定义的列名称。Contacts Provider在返回行结果集之前，隐式的连接了ContactsContract.Data表和其他表。例如：

```
@SuppressWarnings("InlinedApi")
private static final String[] PROJECTION =
{
    /*
     * The detail data row ID. To make a ListView work,
     * this column is required.
     */
    Data._ID,
    // The primary display name
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
        Data.DISPLAY_NAME_PRIMARY :
        Data.DISPLAY_NAME,
    // The contact's _ID, to construct a content URI
    Data.CONTACT_ID
    // The contact's LOOKUP_KEY, to construct a content URI
    Data.LOOKUP_KEY (a permanent link to the contact
};
```

定义查询标准

为了能在特定类型的联系人数据中查询字符串，请按照以下方法构建查询选择子句：

- 列名称包含你要搜索的字符串。这个名字根据数据类型所变化，所以你需要找到与你搜索的数据类型有关的ContactsContract.CommonDataKinds的子类，并从这个子类中选择列名称。例如，想要搜索email地址，需要使用Email.ADDRESS列。
- 搜索字符串本身，请在查询选择子句里使用"?"表示。
- 列名字包含自定义的MIME类型值。这个列名字总是Data.MIMETYPE。
- 自定义MIME类型值的数据类型。如之前描述，这需要使用ContactsContract.CommonDataKinds子类中的CONTENT_ITEM_TYPE常量。例如，email数据的MIME类型值是Email.CONTENT_ITEM_TYPE。需要在这个常量值的开头和结尾加上单引号，否则的话，provider会把这个值翻译成一个变量而不是一个字符串。你不需要为这个值提供占位符，因为你在使用一个常量而不是用户提供的值。例如：

```
/*
```

```

    * Constructs search criteria from the search string
    * and email MIME type
    */
    private static final String SELECTION =
        /*
         * Searches for an email address
         * that matches the search string
         */
        Email.ADDRESS + " LIKE ? " + "AND " +
        /*
         * Searches for a MIME type that matches
         * the value of the constant
         * Email.CONTENT_ITEM_TYPE. Note the
         * single quotes surrounding Email.CONTENT_ITEM_TYPE.
         */
        Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'";

```

下一步，定义包含选择字符串的变量：

```

String mSearchString;
String[] mSelectionArgs = { "" };

```

实现onCreateLoader()方法

现在，你已经指定了你想要的数据和如何找到这些数据。然后需要在onCreateLoader方法中定义一个查询，使用你的数据映射、查询选择表达式和一个数组作为选择表达式的参数，并从这个方法中返回一个新的CursorLoader对象。而内容URI需要使用Data.CONTENT_URI，例如：

```

@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    // OPTIONAL: Makes search string into pattern
    mSearchString = "%" + mSearchString + "%";
    // Puts the search string into the selection criteria
    mSelectionArgs[0] = mSearchString;
    // Starts the query
    return new CursorLoader(
        getActivity(),
        Data.CONTENT_URI,
        PROJECTION,
        SELECTION,
        mSelectionArgs,
        null
    );
}

```

这段代码片段是基于一种特定类型的联系人详情数据的简单反向查找。如果你的应用关注于某一种特定类型的数据，比如说email地址，并且允许用户获得与此数据相关的联系人名字，这种形式的查询是最好的方法。

根据任意类型的数据匹配联系人

根据任意类型的数据获取联系人，如果它们的数据能匹配要搜索的字符串。这些数据包括名字、email地址、邮件地址和电话号码等等。这种搜索结果会比较广泛。例如，如果搜索字符串是"Doe"，搜索任意类型的数据将会返回名字为"Jone Doe"的联系人，也会返回一个住在"Doe Street"的联系人。

为了完成这种类型的查询，就像之前展示的那样，首先需要实现以下代码：

- 请求读取联系人的权限
- 定义列表和列表项的布局
- 定义显示联系人列表的Fragment
- 定义全局变量
- 初始化Fragment
- 为ListView绑定CursorAdapter数据
- 设置选择联系人的监听器

- 定义Cursor的列索引常量

对于这种形式的查询，你需要使用与在“使用特定类型的数据匹配联系人”那一节中相同的表，也可以使用相同的列索引。

- 定义onItemClickListener()方法
- 初始化loader
- 实现onLoadFinished()方法和onLoaderReset()方法

以下的步骤展示了为了能够根据任意类型的数据去匹配查询字符串并显示结果列表，你需要做的额外代码。

去除查询标准

不需要为mSelectionArgs定义查询标准常量SELECTION。这些内容在根据任意类型的数据匹配联系人数据不会用到。

实现onCreateLoader()方法

实现onCreateLoader()方法，返回一个新的CursorLoader对象。你不需要把搜索字符串转化成一个搜索模式，因为Contacts Provider会自动做这件事。使用Contacts.CONTENT_FILTER_URI作为基础查询URI，并使用Uri.withAppendedPath()方法将搜索字符串添加到基础URI中。使用这个URI会自动触发对任意数据类型的搜索，就像以下例子所示：

```
@Override
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    /*
     * Appends the search string to the base URI. Always
     * encode search strings to ensure they're in proper
     * format.
     */
    Uri contentUri = Uri.withAppendedPath(
        Contacts.CONTENT_FILTER_URI,
        Uri.encode(mSearchString));
    // Starts the query
    return new CursorLoader(
        getActivity(),
        contentUri,
        PROJECTION,
        null,
        null,
        null
    );
}
```

这段代码片段，是想要在Contacts Provider中建立广泛搜索类型的应用的基础部分。这种方法对那些想要实现与通讯录应用联系人列表中相似的搜索功能的应用，会很有帮助。

编写:[spencer198711](#)

校对:

获取联系人详情

编写:[spencer198711](#)

校对:

修改联系人信息

编写:[spencer198711](#)

校对:

显示联系人头像

编写:[penkzhou](#)

校对:

位置信息

移动应用一个独特的特征就是对地址的感知。移动用户把他们的设备带到各个地方，这时将位置的感知能力添加到你的应用里可以让用户有更多的地理位置相关的体验。最新的位置服务API集成在Google Play服务里面，内置有自动位置记录，地理围栏，用户活动识别。这个API让Android平台的位置API优势更加突出了。

这个课程教你如何在你的应用里使用位置服务，获取周期性的位置更新，查询地址，创建并监视地理围栏以及探测用户的活动。这个课程包括示例应用和代码片段，你可以使用它们让你的应用拥有位置感知能力。

注意: 因为这个课程基于Google Play services client library，在使用这些示例应用和代码段之前确保你安装了最新版本的Google Play services client library。要想学习如何安装最新版的client library，请参考[安装Google Play services 向导](#)。

分集课程

- [获取当前的位置](#)

学习如何获取用户当前的位置。

- [接收位置更新](#)

学习如何请求和接收周期性的位置更新。

- [显示一个地点位置](#)

学习如何讲一个位置的经纬度转化成一个地址（反向 geocoding）。

- [创建和监视Geofences](#)

学习如何将一个或多个地理区域定义成一个兴趣位置集合，称为地理围栏。学习如何探测用户靠近或者进入地理围栏事件。

- [识别用户当前的活动](#)

学习如何识别用户当前的活动，比如步行，骑行，或者驾车行驶。学习如何使用这些信息去更改你应用的位置策略。

- [使用Mock Locations测试你的应用](#)

学习如何使用虚拟的位置数据来测试一个位置应用。在mock模式里面，，位置服务将会发送一些虚拟的位置数据。

编写:[penkzhou](#)

校对:

获取当前位置

位置服务会自动持有用户当前的位置信息，你的应用在需要位置的时候获取一下即可。位置的精确度基于你所请求的位置权限以及当前设备已经激活的位置传感器。

位置服务通过一个[LocationClient](#)（位置服务类LocationClient的一个实例）将当前的位置发送给你的应用。关于位置信息的所有请求都是通过这个类发送。

注意: 在开始这个课程之前，确定你的开发环境和测试设备处于正常可用状态。要了解更多，请阅读[Google Play services](#) 引导。

确定应用的权限

使用位置服务的应用必须用户位置权限。Android拥有两种位置权限：[ACCESS_COARSE_LOCATION](#)和[ACCESS_FINE_LOCATION](#)。选择不同的权限决定你的应用最后获取的位置信息的精度。如果你只请求了一个精度比较低的位置权限，位置服务会对返回的位置信息处理成一个相当于城市级别精确度的位置。

请求[ACCESS_FINE_LOCATION](#)权限时也包含了[ACCESS_COARSE_LOCATION](#)权限。

举个例子，如果你要添加[ACCESS_COARSE_LOCATION](#)权限，你需要将下面的权限添加到<manifest>标签中：

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

检测Google Play Services

位置服务是Google Play services 中的一部分。由于很难预料用户设备的状态，所以你在尝试连接位置服务之前应该要检测你的设备是否安装了Google Play services安装包。为了检测这个安装包是否被安装，你可以调用[GooglePlayServicesUtil.isGooglePlayServicesAvailable\(\)](#)，这个方法将会返回一个结果代码。你可以通过查阅[ConnectionResult](#)的参考文档中结果代码列表来理解对应的结果代码。如果你碰到了错误，你可以调用[GooglePlayServicesUtil.getErrorDialog\(\)](#)获取本地化的对话框来提示用户采取适当地行为，接着你需要将这个对话框置于一个[DialogFragment](#)中显示。这个对话框可以让用户去纠正这个问题，这个时候Google Services可以将结果返回给你的activity。为了处理这个结果，重写[onActivityResult\(\)](#)即可。

因为你的代码里通常会不止一次地检测Google Play services是否安装, 为了方便，可以定义一个方法来封装这种检测行为。下面的代码片段包含了所有检测Google Play services是否安装需要用到的代码：

```
public class MainActivity extends FragmentActivity {
    ...
    //全局变量
    /*
     * 定义一个发送给Google Play services的请求代码
     * 这个代码将会在Activity.onActivityResult的方法中返回
     */
    private final static int
        CONNECTION_FAILURE_RESOLUTION_REQUEST = 9000;
    ...
    // 定义一个显示错误对话框的DialogFragment
    public static class ErrorDialogFragment extends DialogFragment {
        // 表示错误对话框的全局属性
        private Dialog mDialog;
        // 默认的构造函数，将 dialog 属性设为空
        public ErrorDialogFragment() {
            super();
            mDialog = null;
        }
        // 设置要显示的dialog
        public void setDialog(Dialog dialog) {
            mDialog = dialog;
        }
        // 返回一个 Dialog 给 DialogFragment.
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
            return mDialog;
        }
    }
    ...
    /*
     * 处理来自Google Play services 发给FragmentActivity的结果
     *
     */
    @Override
    protected void onActivityResult(
        int requestCode, int resultCode, Intent data) {
        // 根据请求代码来决定做什么
        switch (requestCode) {
            ...
            case CONNECTION_FAILURE_RESOLUTION_REQUEST :
                /*
                 * 如果结果代码是 Activity.RESULT_OK, 尝试重新连接
                 */
                switch (resultCode) {
                    case Activity.RESULT_OK :
                        /*
                         * 尝试重新请求
                         */
                        ...
                        break;
                }
            ...
        }
    }
    ...
    private boolean servicesConnected() {
        // 检测Google Play services 是否可用
        int resultCode =
            GooglePlayServicesUtil.
                isGooglePlayServicesAvailable(this);
    }
}
```

```

// 如果 Google Play services 可用
if (ConnectionResult.SUCCESS == resultCode) {
    // 在 debug 模式下, 记录程序日志
    Log.d("Location Updates",
        "Google Play services is available.");
    // Continue
    return true;
} // 因为某些原因Google Play services 不可用
else {
    // 获取error code
    int errorCode = connectionResult.getErrorCode();
    // 从Google Play services 获取 error dialog
    Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(
        errorCode,
        this,
        CONNECTION_FAILURE_RESOLUTION_REQUEST);

    // 如果 Google Play services可以提供一個error dialog
    if (errorDialog != null) {
        // 為這個error dialog 創建一個新的DialogFragment
        AlertDialogFragment errorFragment =
            new AlertDialogFragment();
        // 在DialogFragment中設置dialog
        errorFragment.setDialog(errorDialog);
        // 在DialogFragment中顯示error dialog
        errorFragment.show(getSupportFragmentManager(),
            "Location Updates");
    }
}
...
}

```

下面的代码片段使用了这个方法检查Google Play services是否可用。

定义位置服务回调函数

为了获取当前的位置，你需要创建一个location client，将它连接到Location Services，然后调用它的 [getLastLocation\(\)](#) 方法。最后返回的值是基于你应用请求的权限以及当时启用的位置传感器的最佳位置信息。

在你创建location client之前，你必须实现一些被 Location Services用来同你的应用通信的接口

[ConnectionCallbacks](#)

- 设置了当一个location client连接成功或者断开连接时 Location Services必须调用了方法。

[OnConnectionFailedListener](#)

- 设置了当一个错误出现而需要去连接location client时Location Services需要调用的方法。这个方法用到了之前定义好的 `showErrorDialog` 方法来显示一个error dialog，并尝试用Google Play services来修复这个问题。

下面的代码片段展示了如何实现这些接口并定义对应的方法：

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener {

    ...
    /*
     * 当连接到client的请求成功结束时被Location Services 调用。这时你可以请求当前位置或者开始周期性的更新
     */
    @Override
    public void onConnected(Bundle dataBundle) {
        // 显示连接状态
        Toast.makeText(this, "Connected", Toast.LENGTH_SHORT).show();
    }

    ...
    /*
     * 当连接因为错误被location client丢弃时，Location Services调用此方法
     */
    @Override
    public void onDisconnected() {
        // 显示连接状态
        Toast.makeText(this, "Disconnected. Please re-connect.",
            Toast.LENGTH_SHORT).show();
    }

    ...
    /*
     * 尝试连接Location Services失败后被 Location Services调用的方法
     */
    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {
        /*
         * Google Play services 可以解决它探测到的一些错误。
         * 如果这个错误有一个解决方案，这个方法会试着发送一个Intent去启动一个Google Play services activity
         */
        if (connectionResult.hasResolution()) {
            try {
                // 启动一个尝试解决问题的Activity
                connectionResult.startResolutionForResult(
                    this,
                    CONNECTION_FAILURE_RESOLUTION_REQUEST);
            } catch (IntentSender.SendIntentException e) {
                // 记录错误
                e.printStackTrace();
            }
        } else {
            /*
             * 如果没有可用的解决方案，将错误通过一个 dialog 显示给用户
             */
            showErrorDialog(connectionResult.getErrorCode());
        }
    }

    ...
}
```

连接 Location Client

既然这个回调函数已经写好了，现在我们可以创建location client并将它连接到Location Services。

首先你要在onCreate()方法里面创建location client，然后在onStart()方法里面连接它 then connect it in，这样当你的activity对用户可见时 Location Services 就保存着当前的位置信息了。你需要在onStop()方法里面断开连接，这样当你的activity不可见时，Location Services 就不会保存你的位置信息。下面连接和断开连接的方式对节省电池很有帮助。例如：

注意: 当前的位置信息只有在location client 连接到 Location Service时才会被保存。假设没有其他应用连接到 Location Services，如果你断开 client 的连接，那么这时你调用 [getLastLocation\(\)](#)所获取到的位置信息可能已经过时。

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        /*
         * Create a new location client, using the enclosing class to
         * handle callbacks.
         */
        mLocationClient = new LocationClient(this, this, this);
        ...
    }
    ...
    /*
     * 当Activity可见时调用
     */
    @Override
    protected void onStart() {
        super.onStart();
        // 连接 client.
        mLocationClient.connect();
    }
    ...
    /*
     * Called when the Activity is no longer visible.
     */
    @Override
    protected void onStop() {
        // 断开 client 与Location Services的连接, 使client失效。
        mLocationClient.disconnect();
        super.onStop();
    }
    ...
}
```

获取当前的位置信息

为了获取当前的位置信息，调用[getLastLocation\(\)](#)方法。例如：

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener {
    ...
    // 保存当前位置信息的全局变量
    Location mCurrentLocation;
    ...
    mCurrentLocation = mLocationClient.getLastLocation();
    ...
}
```

下一课，[获取位置更新](#)，教你如果周期性地从Location Services获取位置信息更新。

编写:[penkzhou](#)

校对:

获取位置更新

如果你的应用需要导航或者记录路径，你可能会周期性地获取用户的位置信息。此时你可以使用Location Services 里面的 [LocationClient.getLastLocation\(\)](#)来进行周期性的位置信息更新。使用这个方法之后，Location Services 会基于当前可用的位置信息提供源（比如WiFi和GPS）返回最准确的位置信息更新。

你可以使用一个location client从 Location Services 那里请求周期性的位置更新。根据不同请求的形式，Location Services 要么调用一个回调函数并传入一个 [Location](#) 对象，或者发送一个包含位置信息的 [Intent](#)。位置更新的精度和频率与你的应用所申请的权限相关联。

确定应用的权限

使用位置服务的应用必须用户位置权限。Android拥有两种位置权限：[ACCESS_COARSE_LOCATION](#)和[ACCESS_FINE_LOCATION](#)。选择不同的权限决定你的应用最后获取的位置信息的精度。如果你只请求了一个精度比较低的位置权限，位置服务会对返回的位置信息处理成一个相当于城市级别精确度的位置。

请求[ACCESS_FINE_LOCATION](#)权限时也包含了[ACCESS_COARSE_LOCATION](#)权限。

举个例子，如果你要添加[ACCESS_COARSE_LOCATION](#)权限，你需要将下面的权限添加到<manifest>标签中：

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

检测Google Play Services

位置服务是Google Play services 中的一部分。由于很难预料用户设备的状态，所以你在尝试连接位置服务之前应该要检测你的设备是否安装了Google Play services安装包。为了检测这个安装包是否被安装，你可以调用[GooglePlayServicesUtil.isGooglePlayServicesAvailable\(\)](#)，这个方法将会返回一个结果代码。你可以通过查阅[ConnectionResult](#)的参考文档中结果代码列表来理解对应的结果代码。如果你碰到了错误，你可以调用[GooglePlayServicesUtil.getErrorDialog\(\)](#)获取本地化的对话框来提示用户采取适当地行为，接着你需要将这个对话框置于一个[DialogFragment](#)中显示。这个对话框可以让用户去纠正这个问题，这个时候Google Services可以将结果返回给你的activity。为了处理这个结果，重写[onActivityResult\(\)](#)即可。

注意: 为了让你的应用能够兼容 Android 1.6 之后的版本，用来显示DialogFragment的必须是FragmentActivity而不是之前的Activity。使用FragmentActivity同样可以调用 `getSupportFragmentManager()` 方法来显示 DialogFragment。

因为你的代码里通常会不止一次地检测Google Play services是否安装, 为了方便，可以定义一个方法来封装这种检测行为。下面的代码片段包含了所有检测Google Play services是否安装需要用到的代码：

```
public class MainActivity extends FragmentActivity {
    ...
    //全局变量
    /*
     * 定义一个发送给Google Play services的请求代码
     * 这个代码将会在Activity.onActivityResult的方法中返回
     */
    private final static int
        CONNECTION_FAILURE_RESOLUTION_REQUEST = 9000;
    ...
    // 定义一个显示错误对话框的DialogFragment
    public static class ErrorDialogFragment extends DialogFragment {
        // 表示错误对话框的全局属性
        private Dialog mDialog;
        // 默认的构造函数，将 dialog 属性设为空
        public ErrorDialogFragment() {
            super();
            mDialog = null;
        }
        // 设置要显示的dialog
        public void setDialog(Dialog dialog) {
            mDialog = dialog;
        }
        // 返回一个 Dialog 给 DialogFragment.
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
            return mDialog;
        }
    }
    ...
    /*
     * 处理来自Google Play services 发给FragmentActivity的结果
     */
    @Override
    protected void onActivityResult(
        int requestCode, int resultCode, Intent data) {
        // 根据请求代码来决定做什么
        switch (requestCode) {
            ...
            case CONNECTION_FAILURE_RESOLUTION_REQUEST :
                /*
                 * 如果结果代码是 Activity.RESULT_OK，尝试重新连接
                 */
                switch (resultCode) {
                    case Activity.RESULT_OK :
                        /*
                         * 尝试重新请求
                         */
                        ...
                        break;
                }
            ...
        }
    }
}
```

```

...
private boolean servicesConnected() {
    // 检测Google Play services 是否可用
    int resultCode =
        GooglePlayServicesUtil.
            isGooglePlayServicesAvailable(this);
    // 如果 Google Play services 可用
    if (ConnectionResult.SUCCESS == resultCode) {
        // 在 debug 模式下, 记录程序日志
        Log.d("Location Updates",
            "Google Play services is available.");
        // Continue
        return true;
    }
    // 因为某些原因Google Play services 不可用
    } else {
        // 获取error code
        int errorCode = connectionResult.getErrorCode();
        // 从Google Play services 获取 error dialog
        Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(
            errorCode,
            this,
            CONNECTION_FAILURE_RESOLUTION_REQUEST);

        // 如果 Google Play services可以提供一个error dialog
        if (errorDialog != null) {
            // 为这个error dialog 创建一个新的DialogFragment
            AlertDialogFragment errorFragment =
                new AlertDialogFragment();
            // 在DialogFragment中设置dialog
            errorFragment.setDialog(errorDialog);
            // 在DialogFragment中显示error dialog
            errorFragment.show(getSupportFragmentManager(),
                "Location Updates");
        }
    }
}
...
}

```

下面的代码片段使用了这个方法检查Google Play services是否可用。

定义位置服务回调函数

在你创建location client之前,你必须实现一些被 Location Services用来同你的应用通信的接口

[ConnectionCallbacks](#)

- 设置了当一个location client连接成功或者断开连接时 Location Services必须调用了方法。

[OnConnectionFailedListener](#)

- 设置了当一个错误出现而需要去连接location client时Location Services需要调用的方法。这个方法用到了之前定义好的 `showErrorDialog` 方法来显示一个error dialog, 并尝试用Google Play services来修复这个问题。

下面的代码片段展示了如何实现这些接口并定义对应的方法：

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener {
    ...
    /**
     * 当连接到client的请求成功结束时被Location Services 调用。这时你可以请求当前位置或者开始周期性的更新
     */
    @Override
    public void onConnected(Bundle dataBundle) {
        // 显示连接状态
        Toast.makeText(this, "Connected", Toast.LENGTH_SHORT).show();
    }
    ...
    /**
     * 当连接因为错误被location client丢弃时, Location Services调用此方法
     */
    @Override
    public void onDisconnected() {
        // 显示连接状态
        Toast.makeText(this, "Disconnected. Please re-connect.",
            Toast.LENGTH_SHORT).show();
    }
    ...
    /**
     * 尝试连接Location Services失败后被 Location Services调用的方法
     */
    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {
        /**
         * Google Play services 可以解决它探测到的一些错误。
         * 如果这个错误有一个解决方案, 这个方法会试着发送一个Intent去启动一个Google Play services activity
         */
        if (connectionResult.hasResolution()) {
            try {
                // 启动一个尝试解决问题的Activity
                connectionResult.startResolutionForResult(
                    this,
                    CONNECTION_FAILURE_RESOLUTION_REQUEST);
            } catch (IntentSender.SendIntentException e) {
                // 记录错误
                e.printStackTrace();
            }
        } else {
            /**
             * 如果没有可用的解决方案, 将错误通过一个 dialog 显示给用户
             */
            showErrorDialog(connectionResult.getErrorCode());
        }
    }
    ...
}
```

现在你已经写好了回调函数, 你可以设置位置更新的请求了。第一步就是确定可以控制位置更新的参数。

确定位置更新参数

Location Services可以让你通过设置[LocationRequest](#)里面的值来控制位置更新的频率和精度，然后把[LocationRequest](#)这个对象作为更新请求的一部分发送出去，接着就可以开始更新位置信息了。

首页，设置下面的周期参数：

更新频率

- 更新频率通过 [LocationRequest.setInterval\(\)](#)方法来设置。这个方法设置的毫秒数表示你的应用在这个时间内尽可能的接受位置更新信息。如果当时没有其他应用从Location Services获取位置更新，那么你的应用就会已设置的频率接收位置更新。

最快更新频率

- 最快更新频率通过[LocationRequest.setFastestInterval\(\)](#)方法设置。这个方法设置你的应用能够接收位置更新最快的频率。你必须设置这个频率因为其他应用也会影响位置更新的频率。Location Services 会以选择所有请求位置更新的应用中频率最快的发送位置更新。。如果这个频率比你的应用能处理的频率还要快，那么你的应用可能会出现UI闪烁或者数据溢出。为了防止这样的情况出现，调用[LocationRequest.setFastestInterval\(\)](#)方法来设置位置更新频率的上限，同时还可以节约电量。当你通过 [LocationRequest.setInterval\(\)](#)方法设置理想的更新频率，通过 [LocationRequest.setFastestInterval\(\)](#)设置更新频率的上限，然后你的应用就会在系统中获得最快的位置更新频率。如果其他应用设置的更新频率更快，那么你的应用也跟着受益。如果其他应用的更新频率没有你的频率快，那么你的应用将会以你通过[LocationRequest.setInterval\(\)](#)方法设置的频率更新位置信息。

接着，设置精度参数。在一个前台应用（foreground app）中，你需要不断地获取高精度的位置更新，因此需要使用[LocationRequest.PRIORITY_HIGH_ACCURACY](#)。

下面的代码片段展示了如何在[onCreate\(\)](#)方法里面设置更新频率和精度：

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener,
    LocationListener {
    ...
    // Global constants
    ...
    // Milliseconds per second
    private static final int MILLISECONDS_PER_SECOND = 1000;
    // Update frequency in seconds
    public static final int UPDATE_INTERVAL_IN_SECONDS = 5;
    // Update frequency in milliseconds
    private static final long UPDATE_INTERVAL =
        MILLISECONDS_PER_SECOND * UPDATE_INTERVAL_IN_SECONDS;
    // The fastest update frequency, in seconds
    private static final int FASTEST_INTERVAL_IN_SECONDS = 1;
    // A fast frequency ceiling in milliseconds
    private static final long FASTEST_INTERVAL =
        MILLISECONDS_PER_SECOND * FASTEST_INTERVAL_IN_SECONDS;
    ...
    // 定义一个包含定位精度和定位频率的对象
    LocationRequest mLocationRequest;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Create the LocationRequest object
        mLocationRequest = LocationRequest.create();
        // 使用高精度
        mLocationRequest.setPriority(
            LocationRequest.PRIORITY_HIGH_ACCURACY);
        // 设置更新频率为 5 seconds
        mLocationRequest.setInterval(UPDATE_INTERVAL);
        // 设置最快更新频率为 1 second
        mLocationRequest.setFastestInterval(FASTEST_INTERVAL);
        ...
    }
    ...
}
```

注意：如果你的应用在获取位置更新后需要访问网络或者进行长时的操作，你可以将最快更新频率调整至更慢的值。这样可以让你的应用不会接受无法使用的位置更新。一旦这样的长时操作完成，将最快更新频率设回原值。

开始进行位置更新

To send the request for location updates, create a location client in `onCreate()`, then connect it and make the request by calling `requestLocationUpdates()`. Since your client must be connected for your app to receive updates, you should connect the client in `onStart()`. This ensures that you always have a valid, connected client while your app is visible. Since you need a connection before you can request updates, make the update request in `ConnectionCallbacks.onConnected()`

Remember that the user may want to turn off location updates for various reasons. You should provide a way for the user to do this, and you should ensure that you don't start updates in `onStart()` if updates were previously turned off. To track the user's preference, store it in your app's `SharedPreferences` in `onPause()` and retrieve it in `onResume()`.

The following snippet shows how to set up the client in `onCreate()`, and how to connect it and request updates in `onStart()`:

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener,
    LocationListener {
    ...
    // Global variables
    ...
    LocationClient mLocationClient;
    boolean mUpdatesRequested;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Open the shared preferences
        mPrefs = getSharedPreferences("SharedPreferences",
            Context.MODE_PRIVATE);
        // Get a SharedPreferences editor
        mEditor = mPrefs.edit();
        /*
         * Create a new location client, using the enclosing class to
         * handle callbacks.
         */
        mLocationClient = new LocationClient(this, this, this);
        // Start with updates turned off
        mUpdatesRequested = false;
        ...
    }
    ...
    @Override
    protected void onPause() {
        // Save the current setting for updates
        mEditor.putBoolean("KEY_UPDATES_ON", mUpdatesRequested);
        mEditor.commit();
        super.onPause();
    }
    ...
    @Override
    protected void onStart() {
        ...
        mLocationClient.connect();
    }
    ...
    @Override
    protected void onResume() {
        /*
         * Get any previous setting for location updates
         * Gets "false" if an error occurs
         */
        if (mPrefs.contains("KEY_UPDATES_ON")) {
            mUpdatesRequested =
                mPrefs.getBoolean("KEY_UPDATES_ON", false);

            // Otherwise, turn off location updates
        } else {
            mEditor.putBoolean("KEY_UPDATES_ON", false);
            mEditor.commit();
        }
    }
    ...
    /*
     * Called by Location Services when the request to connect the
     * client finishes successfully. At this point, you can
```



```
        * request the current location or start periodic updates
        */
    @Override
    public void onConnected(Bundle dataBundle) {
        // Display the connection status
        Toast.makeText(this, "Connected", Toast.LENGTH_SHORT).show();
        // If already requested, start periodic updates
        if (mUpdatesRequested) {
            mLocationClient.requestLocationUpdates(mLocationRequest, this);
        }
    }
    ...
}
```

For more information about saving preferences, read [Saving Key-Value Sets](#).

Stop Location Updates

To stop location updates, save the state of the update flag in `onPause()`, and stop updates in `onStop()` by calling `removeLocationUpdates(LocationListener)`. For example:

```
public class MainActivity extends FragmentActivity implements
    GooglePlayServicesClient.ConnectionCallbacks,
    GooglePlayServicesClient.OnConnectionFailedListener,
    LocationListener {
    ...
    /*
     * Called when the Activity is no longer visible at all.
     * Stop updates and disconnect.
     */
    @Override
    protected void onStop() {
        // If the client is connected
        if (mLocationClient.isConnected()) {
            /*
             * Remove location updates for a listener.
             * The current Activity is the listener, so
             * the argument is "this".
             */
            removeLocationUpdates(this);
        }
        /*
         * After disconnect() is called, the client is
         * considered "dead".
         */
        mLocationClient.disconnect();
        super.onStop();
    }
    ...
}
```

You now have the basic structure of an app that requests and receives periodic location updates. You can combine the features described in this lesson with the geofencing, activity recognition, or reverse geocoding features described in other lessons in this class.

The next lesson, [Displaying a Location Address](#), shows you how to use the current location to display the current street address.

编写:[penkzhou](#)

校对:

显示位置地址

[获取当前的位置](#)和[接收位置更新](#)课程描述了如何以一个[Location](#)对象的形式获取用户当前的位置信息，这个位置信息包括了经纬度。尽管经纬度对计算地理距离和在地图上显示位置很有帮助，但是更多情况下位置信息的地址更有用。

Android平台API提供一个根据地理经纬度返回一个大概的街道地址信息这一课教你如何使用这个地址检索功能。

注意：地址检索需要一个后台服务，然后这个后台服务是不包含在核心的Android框架里面的。如果这个后台服务不可用，[Geocoder.getFromLocation\(\)](#)方法将会返回一个空列表。在Android API 9以上的API里面有一个辅助方法[isPresent\(\)](#)可以检查这个后台服务是否可用。

下面的代码假设你已经获取到了位置信息并将位置信息以[Location](#)对象的形式保存到全局变量mLocation里面。

定义地址检索任务

为了通过给定的经纬度获取地址信息，你需要调用[Geocoder.getFromLocation\(\)](#)方法并返回一个地址列表。由于这个方法是同步的，所以在获取地址信息的时候可能耗时较长，因此你需要通过[AsyncTask](#)的[doInBackground\(\)](#)方法来执行这个方法。

当你的应用在获取地址信息的时候，可以使用一个进度条之类的控件来表示你的应用正在后台工作中。你可以将这个控件的初始状态设为[android:visibility="gone"](#)，这样可以让这个控件不可见。当你开始进行地址检索的时候，你需要将这个控件的可见属性设为["visible"](#)。

下面的代码教你如何在你的布局文件里面添加一个进度条：

```
<ProgressBar
    android:id="@+id/address_progress"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:indeterminate="true"
    android:visibility="gone" />
```

要创建一个后台任务，首先要定义一个[AsyncTask](#)的子类来调用[getFromLocation\(\)](#)方法，然后返回地址。定义一个[TextView](#)对象[mAddress](#)来显示返回的地址信息，进度条则用来显示请求的进度过程。例如：

```
public class MainActivity extends FragmentActivity {
    ...
    private TextView mAddress;
    private ProgressBar mActivityIndicator;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        mAddress = (TextView) findViewById(R.id.address);
        mActivityIndicator =
            (ProgressBar) findViewById(R.id.address_progress);
    }
    ...
    /**
     * A subclass of AsyncTask that calls getFromLocation() in the
     * background. The class definition has these generic types:
     * Location - A Location object containing
     * the current location.
     * Void      - indicates that progress units are not used
     * String    - An address passed to onPostExecute()
     */
    private class GetAddressTask extends
        AsyncTask<Location, Void, String> {
        Context mContext;
        public GetAddressTask(Context context) {
            super();
            mContext = context;
        }
        ...
        /**
         * Get a Geocoder instance, get the latitude and longitude
         * look up the address, and return it
         *
         * @params params One or more Location objects
         * @return A string containing the address of the current
         * location, or an empty string if no address can be found,
         * or an error message
         */
        @Override
        protected String doInBackground(Location... params) {
            Geocoder geocoder =
                new Geocoder(mContext, Locale.getDefault());
            // Get the current location from the input parameter list
            Location loc = params[0];
            // Create a list to contain the result address
            List<Address> addresses = null;
            try {
                /*
                 * Return 1 address.
                 */
                addresses = geocoder.getFromLocation(loc.getLatitude(),
```

```

        loc.getLongitude(), 1);
    } catch (IOException e1) {
        Log.e("LocationSampleActivity",
            "IO Exception in getFromLocation()");
        e1.printStackTrace();
        return ("IO Exception trying to get address");
    } catch (IllegalArgumentException e2) {
        // Error message to post in the log
        String errorString = "Illegal arguments " +
            Double.toString(loc.getLatitude()) +
            ", " +
            Double.toString(loc.getLongitude()) +
            " passed to address service";
        Log.e("LocationSampleActivity", errorString);
        e2.printStackTrace();
        return errorString;
    }
    // If the reverse geocode returned an address
    if (addresses != null && addresses.size() > 0) {
        // Get the first address
        Address address = addresses.get(0);
        /*
         * Format the first line of address (if available),
         * city, and country name.
         */
        String addressText = String.format(
            "%s, %s, %s",
            // If there's a street address, add it
            address.getMaxAddressLineIndex() > 0 ?
                address.getAddressLine(0) : "",
            // Locality is usually a city
            address.getLocality(),
            // The country of the address
            address.getCountryName());
        // Return the text
        return addressText;
    } else {
        return "No address found";
    }
}
...
}
...
}

```

下一部分教你如何在用户界面上显示地址信息。

定义显示结果的方法

[doInBackground\(\)](#)方法返回一个包含地址检索结果的字符串。这个值会被传入[onPostExecute\(\)](#)方法，通过这个方法你可以对结果进行更深的处理。因为[onPostExecute\(\)](#)运行在UI主线程上面，它可以更新用户界面；例如，它可以隐藏进度条然后显示返回的地址结果给用户：

```
private class GetAddressTask extends
    AsyncTask<Location, Void, String> {
    ...
    /**
     * A method that's called once doInBackground() completes. Turn
     * off the indeterminate activity indicator and set
     * the text of the UI element that shows the address. If the
     * lookup failed, display the error message.
     */
    @Override
    protected void onPostExecute(String address) {
        // Set activity indicator visibility to "gone"
        mActivityIndicator.setVisibility(View.GONE);
        // Display the results of the lookup.
        mAddress.setText(address);
    }
    ...
}
```

最后一步就是运行地址检索任务。

运行地址检索任务

为了获取地址信息，调用[execute\(\)](#)方法即可。例如，下面的代码片段展示了当用户点击"Get Address"按钮时应用就开始检索地址信息了：

```
public class MainActivity extends FragmentActivity {
    ...
    /**
     * The "Get Address" button in the UI is defined with
     * android:onClick="getAddress". The method is invoked whenever the
     * user clicks the button.
     *
     * @param v The view object associated with this method,
     * in this case a Button.
     */
    public void getAddress(View v) {
        // Ensure that a Geocoder services is available
        if (Build.VERSION.SDK_INT >=
            Build.VERSION_CODES.GINGERBREAD
                &&
            Geocoder.isPresent()) {
            // Show the activity indicator
            mActivityIndicator.setVisibility(View.VISIBLE);
            /*
             * Reverse geocoding is long-running and synchronous.
             * Run it on a background thread.
             * Pass the current location to the background task.
             * When the task finishes,
             * onPostExecute() displays the address.
             */
            (new GetAddressTask(this)).execute(mLocation);
        }
        ...
    }
    ...
}
```

下一课，[创建和监视Geofences](#)将会教你如何定义地理围栏以及如何使用地理围栏来探测用户对一个兴趣位置的接近程度。

编写:[penkzhou](#)

校对:

创建并监视异常区域

地理围栏将用户当前位置感知和附件地点特征感知相结合，定义了用户对位置的接近程度。为了让一个位置有感知，你必须确定这个位置的经纬度。为了度量用户对位置的接近程度，你需要添加一个半径。综合经纬度和半径即可确定一个地理围栏。当然你可以一次性定义多个地理围栏。

Location Services将一个地理围栏看成是一片区域而不是一个点和一个接近程度。这样可以让它去探测用户是否进入或者正在某个地理围栏中。对于每个地理围栏，你可以让Location Services给你发送进入或者退出地理围栏事件。你还可以通过设置一个毫秒级别的有效时间来限制地理围栏的生命周期。当地理围栏失效后，Location Services会自动移除这个地理围栏。

请求地理围栏监视

请求地理围栏监视的第一步就是设置必要的权限。在使用地理围栏时，你必须设置[ACCESS_FINE_LOCATION](#)权限。添加如下代码即可：

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

检查Google Play Services是否可用

位置服务是Google Play services 中的一部分。由于很难预料用户设备的状态，所以你在尝试连接位置服务之前应该要检测你的设备是否安装了Google Play services安装包。为了检测这个安装包是否被安装，你可以调用[GooglePlayServicesUtil.isGooglePlayServicesAvailable\(\)](#)，这个方法将会返回一个结果代码。你可以通过查询[ConnectionResult](#)的参考文档中结果代码列表来理解对应的结果代码。如果你碰到了错误，你可以调用[GooglePlayServicesUtil.getErrorDialog\(\)](#)获取本地化的对话框来提示用户采取适当地行为，接着你需要将这个对话框置于一个[DialogFragment](#)中显示。这个对话框可以让用户去纠正这个问题，这个时候Google Services可以将结果返回给你的activity。为了处理这个结果，重写[onActivityResult\(\)](#)即可。

注意: 为了让你的应用能够兼容 Android 1.6 之后的版本，用来显示DialogFragment的必须是FragmentActivity而不是之前的Activity。使用FragmentActivity同样可以调用 `getSupportFragmentManager()` 方法来显示 DialogFragment。

因为你的代码里通常会不止一次地检测Google Play services是否安装, 为了方便，可以定义一个方法来封装这种检测行为。下面的代码片段包含了所有检测Google Play services是否安装需要用到的代码：

```
public class MainActivity extends FragmentActivity {
    ...
    //全局变量
    /*
     * 定义一个发送给Google Play services的请求代码
     * 这个代码将会在Activity.onActivityResult的方法中返回
     */
    private final static int
        CONNECTION_FAILURE_RESOLUTION_REQUEST = 9000;
    ...
    // 定义一个显示错误对话框的DialogFragment
    public static class ErrorDialogFragment extends DialogFragment {
        // 表示错误对话框的全局属性
        private Dialog mDialog;
        // 默认的构造函数，将 dialog 属性设为空
        public ErrorDialogFragment() {
            super();
            mDialog = null;
        }
        // 设置要显示的dialog
        public void setDialog(Dialog dialog) {
            mDialog = dialog;
        }
        // 返回一个 Dialog 给 DialogFragment.
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
            return mDialog;
        }
    }
    ...
    /*
     * 处理来自Google Play services 发给FragmentActivity的结果
     *
     */
    @Override
    protected void onActivityResult(
        int requestCode, int resultCode, Intent data) {
        // 根据请求代码来决定做什么
        switch (requestCode) {
            ...
            case CONNECTION_FAILURE_RESOLUTION_REQUEST :
                /*
                 * 如果结果代码是 Activity.RESULT_OK, 尝试重新连接
                 */
                switch (resultCode) {
                    case Activity.RESULT_OK :
                        /*
```

```

        * 尝试重新请求
        */
        ...
        break;
    }
    ...
}
...
private boolean servicesConnected() {
    // 检测Google Play services 是否可用
    int resultCode =
        GooglePlayServicesUtil.
            isGooglePlayServicesAvailable(this);
    // 如果 Google Play services 可用
    if (ConnectionResult.SUCCESS == resultCode) {
        // 在 debug 模式下, 记录程序日志
        Log.d("Location Updates",
            "Google Play services is available.");
        // Continue
        return true;
    }
    // 因为某些原因Google Play services 不可用
    } else {
        // 获取error code
        int errorCode = connectionResult.getErrorCode();
        // 从Google Play services 获取 error dialog
        Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(
            errorCode,
            this,
            CONNECTION_FAILURE_RESOLUTION_REQUEST);

        // 如果 Google Play services 可以提供一个error dialog
        if (errorDialog != null) {
            // 为这个error dialog 创建一个新的DialogFragment
            AlertDialogFragment errorFragment =
                new AlertDialogFragment();
            // 在DialogFragment中设置dialog
            errorFragment.setDialog(errorDialog);
            // 在DialogFragment中显示error dialog
            errorFragment.show(getSupportFragmentManager(),
                "Geofence Detection");
        }
    }
}
...
}

```

下面的代码片段使用了这个方法检查Google Play services是否可用。

要使用地理围栏，你得先定义你要监控的地理围栏。通常你可以在本地保存地理围栏数据或者从互联网上获取地理围栏数据，然后你需要发送一个由[Geofence.Builder](#)创建的[Geofence](#)对象给Location Services。每一个[Geofence](#)对象都包括了以下数据：

精度, 纬度和半径

- 为地理围栏定义一个圆形区域。使用经纬度标记一个兴趣地点，然后使用半径定义地理围栏有效区域。半径越大，用户越有可能触发地理围栏的警报。例如，为一个家庭灯具地理围栏应用设置一个大的半径，这样当用户回家就可能触发地理围栏，然后灯就亮了。

有效时间

- 地理围栏保持激活状态的时间。一旦达到了有效时间，Location Services会删除这个地理围栏。大部分时候，你都应该为你的应用设置一个有效时间，但对于家居或者工作空间等类型的应用，可能设置需要永久的地理围栏。nces for the user's home or place of work.

触发事件类型

- Location Services 能探测到用户进入地理围栏的有效范围内或者走出有效范围的事件。

地理围栏 ID

- 一个与地理围栏一同保存的字符串。你必须保证这个字符串唯一，这样你就可以使用它从Location Services的记录里来移除特定地理围栏。

定义地理围栏存储

一个地理围栏应用需要将地理围栏数据做持久化的读写。但是你不能使用[Geofence](#)进行这样的操作；你可以使用数据库等方式来保存地理围栏的相关数据。

作为一个保存地理围栏数据的实例，下面的代码片段定义了两个使用[SharedPreferences](#) 的类来进行地理围栏的数据持久化。[SimpleGeofence](#)类，类似于一条数据库记录，为一个[Geofence](#) 对象存储数据。[SimpleGeofenceStore](#)类，类似于一个数据库，对[SimpleGeofence](#) 的读写应用到 [SharedPreferences](#) 实例。

```
public class MainActivity extends FragmentActivity {
    ...
    /**
     * A single Geofence object, defined by its center and radius.
     */
    public class SimpleGeofence {
        // Instance variables
        private final String mId;
        private final double mLatitude;
        private final double mLongitude;
        private final float mRadius;
        private long mExpirationDuration;
        private int mTransitionType;

        /**
         * @param geofenceId The Geofence's request ID
         * @param latitude Latitude of the Geofence's center.
         * @param longitude Longitude of the Geofence's center.
         * @param radius Radius of the geofence circle.
         * @param expiration Geofence expiration duration
         * @param transition Type of Geofence transition.
         */
        public SimpleGeofence(
            String geofenceId,
            double latitude,
            double longitude,
            float radius,
            long expiration,
            int transition) {
            // Set the instance fields from the constructor
            this.mId = geofenceId;
            this.mLatitude = latitude;
            this.mLongitude = longitude;
            this.mRadius = radius;
            this.mExpirationDuration = expiration;
            this.mTransitionType = transition;
        }
        // Instance field getters
        public String getId() {
            return mId;
        }
        public double getLatitude() {
            return mLatitude;
        }
        public double getLongitude() {
            return mLongitude;
        }
        public float getRadius() {
            return mRadius;
        }
        public long getExpirationDuration() {
            return mExpirationDuration;
        }
        public int getTransitionType() {
            return mTransitionType;
        }
    }
    /**
     * Creates a Location Services Geofence object from a
     * SimpleGeofence.
     *
     * @return A Geofence object
     */
    public Geofence toGeofence() {
        // Build a new Geofence object
        return new Geofence.Builder()
            .setRequestId(getId())
            .setTransitionTypes(mTransitionType)
            .setCircularRegion(
                getLatitude(), getLongitude(), getRadius())
            .setExpirationDuration(mExpirationDuration)
            .build();
    }
}
```

```

    }
}

...
/**
 * Storage for geofence values, implemented in SharedPreferences.
 */
public class SimpleGeofenceStore {
    // Keys for flattened geofences stored in SharedPreferences
    public static final String KEY_LATITUDE =
        "com.example.android.geofence.KEY_LATITUDE";
    public static final String KEY_LONGITUDE =
        "com.example.android.geofence.KEY_LONGITUDE";
    public static final String KEY_RADIUS =
        "com.example.android.geofence.KEY_RADIUS";
    public static final String KEY_EXPIRATION_DURATION =
        "com.example.android.geofence.KEY_EXPIRATION_DURATION";
    public static final String KEY_TRANSITION_TYPE =
        "com.example.android.geofence.KEY_TRANSITION_TYPE";
    // The prefix for flattened geofence keys
    public static final String KEY_PREFIX =
        "com.example.android.geofence.KEY";

    /*
     * Invalid values, used to test geofence storage when
     * retrieving geofences
     */
    public static final long INVALID_LONG_VALUE = -999l;
    public static final float INVALID_FLOAT_VALUE = -999.0f;
    public static final int INVALID_INT_VALUE = -999;
    // The SharedPreferences object in which geofences are stored
    private final SharedPreferences mPrefs;
    // The name of the SharedPreferences
    private static final String SHARED_PREFERENCES =
        "SharedPreferences";
    // Create the SharedPreferences storage with private access only
    public SimpleGeofenceStore(Context context) {
        mPrefs =
            context.getSharedPreferences(
                SHARED_PREFERENCES,
                Context.MODE_PRIVATE);
    }
    /**
     * Returns a stored geofence by its id, or returns null
     * if it's not found.
     *
     * @param id The ID of a stored geofence
     * @return A geofence defined by its center and radius. See
     */
    public SimpleGeofence getGeofence(String id) {
        /*
         * Get the latitude for the geofence identified by id, or
         * INVALID_FLOAT_VALUE if it doesn't exist
         */
        double lat = mPrefs.getFloat(
            getGeofenceFieldKey(id, KEY_LATITUDE),
            INVALID_FLOAT_VALUE);

        /*
         * Get the longitude for the geofence identified by id, or
         * INVALID_FLOAT_VALUE if it doesn't exist
         */
        double lng = mPrefs.getFloat(
            getGeofenceFieldKey(id, KEY_LONGITUDE),
            INVALID_FLOAT_VALUE);

        /*
         * Get the radius for the geofence identified by id, or
         * INVALID_FLOAT_VALUE if it doesn't exist
         */
        float radius = mPrefs.getFloat(
            getGeofenceFieldKey(id, KEY_RADIUS),
            INVALID_FLOAT_VALUE);

        /*
         * Get the expiration duration for the geofence identified
         * by id, or INVALID_LONG_VALUE if it doesn't exist
         */
        long expirationDuration = mPrefs.getLong(
            getGeofenceFieldKey(id, KEY_EXPIRATION_DURATION),
            INVALID_LONG_VALUE);
    }
}

```

```

    /**
     * Get the transition type for the geofence identified by
     * id, or INVALID_INT_VALUE if it doesn't exist
     */
    int transitionType = mPrefs.getInt(
        getGeofenceFieldKey(id, KEY_TRANSITION_TYPE),
        INVALID_INT_VALUE);
    // If none of the values is incorrect, return the object
    if (
        lat != GeofenceUtils.INVALID_FLOAT_VALUE &&
        lng != GeofenceUtils.INVALID_FLOAT_VALUE &&
        radius != GeofenceUtils.INVALID_FLOAT_VALUE &&
        expirationDuration !=
            GeofenceUtils.INVALID_LONG_VALUE &&
        transitionType != GeofenceUtils.INVALID_INT_VALUE) {

        // Return a true Geofence object
        return new SimpleGeofence(
            id, lat, lng, radius, expirationDuration,
            transitionType);
    } else {
        return null;
    }
}
/**
 * Save a geofence.
 * @param geofence The SimpleGeofence containing the
 * values you want to save in SharedPreferences
 */
public void setGeofence(String id, SimpleGeofence geofence) {
    /**
     * Get a SharedPreferences editor instance. Among other
     * things, SharedPreferences ensures that updates are atomic
     * and non-concurrent
     */
    Editor editor = mPrefs.edit();
    // Write the Geofence values to SharedPreferences
    editor.putFloat(
        getGeofenceFieldKey(id, KEY_LATITUDE),
        (float) geofence.getLatitude());
    editor.putFloat(
        getGeofenceFieldKey(id, KEY_LONGITUDE),
        (float) geofence.getLongitude());
    editor.putFloat(
        getGeofenceFieldKey(id, KEY_RADIUS),
        geofence.getRadius());
    editor.putLong(
        getGeofenceFieldKey(id, KEY_EXPIRATION_DURATION),
        geofence.getExpirationDuration());
    editor.putInt(
        getGeofenceFieldKey(id, KEY_TRANSITION_TYPE),
        geofence.getTransitionType());
    // Commit the changes
    editor.commit();
}
public void clearGeofence(String id) {
    /**
     * Remove a flattened geofence object from storage by
     * removing all of its keys
     */
    Editor editor = mPrefs.edit();
    editor.remove(getGeofenceFieldKey(id, KEY_LATITUDE));
    editor.remove(getGeofenceFieldKey(id, KEY_LONGITUDE));
    editor.remove(getGeofenceFieldKey(id, KEY_RADIUS));
    editor.remove(getGeofenceFieldKey(id,
        KEY_EXPIRATION_DURATION));
    editor.remove(getGeofenceFieldKey(id, KEY_TRANSITION_TYPE));
    editor.commit();
}
/**
 * Given a Geofence object's ID and the name of a field
 * (for example, KEY_LATITUDE), return the key name of the
 * object's values in SharedPreferences.
 *
 * @param id The ID of a Geofence object

```

```

        * @param fieldName The field represented by the key
        * @return The full key name of a value in SharedPreferences
        */
        private String getGeofenceFieldKey(String id,
            String fieldName) {
            return KEY_PREFIX + "_" + id + "_" + fieldName;
        }
    }
    ...
}

```

创建地理围栏对象

下面的代码片段使用SimpleGeofence和SimpleGeofenceStore类从用户界面上获取地理围栏数据，然后将这些数据保存到SimpleGeofence对象里面，接着把这些SimpleGeofence对象保存到一个SimpleGeofenceStore里面，然后就可以创建 [Geofence](#) 对象了：

```

public class MainActivity extends FragmentActivity {
    ...
    /*
     * Use to set an expiration time for a geofence. After this amount
     * of time Location Services will stop tracking the geofence.
     */
    private static final long SECONDS_PER_HOUR = 60;
    private static final long MILLISECONDS_PER_SECOND = 1000;
    private static final long GEOFENCE_EXPIRATION_IN_HOURS = 12;
    private static final long GEOFENCE_EXPIRATION_TIME =
        GEOFENCE_EXPIRATION_IN_HOURS *
        SECONDS_PER_HOUR *
        MILLISECONDS_PER_SECOND;

    ...
    /*
     * Handles to UI views containing geofence data
     */
    // Handle to geofence 1 latitude in the UI
    private EditText mLatitude1;
    // Handle to geofence 1 longitude in the UI
    private EditText mLongitude1;
    // Handle to geofence 1 radius in the UI
    private EditText mRadius1;
    // Handle to geofence 2 latitude in the UI
    private EditText mLatitude2;
    // Handle to geofence 2 longitude in the UI
    private EditText mLongitude2;
    // Handle to geofence 2 radius in the UI
    private EditText mRadius2;
    /*
     * Internal geofence objects for geofence 1 and 2
     */
    private SimpleGeofence mUIGeofence1;
    private SimpleGeofence mUIGeofence2;
    ...
    // Internal List of Geofence objects
    List<Geofence> mGeofenceList;
    // Persistent storage for geofences
    private SimpleGeofenceStore mGeofenceStorage;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        // Instantiate a new geofence storage area
        mGeofenceStorage = new SimpleGeofenceStore(this);

        // Instantiate the current List of geofences
        mCurrentGeofences = new ArrayList<Geofence>();
    }
    ...
    /**
     * Get the geofence parameters for each geofence from the UI
     * and add them to a List.
     */
    public void createGeofences() {
        /*
         * Create an internal object to store the data. Set its

```



```

    * ID to "1". This is a "flattened" object that contains
    * a set of strings
    */
    mUIGeofence1 = new SimpleGeofence(
        "1",
        Double.valueOf(mLatitude1.getText().toString()),
        Double.valueOf(mLongitude1.getText().toString()),
        Float.valueOf(mRadius1.getText().toString()),
        GEOFENCE_EXPIRATION_TIME,
        // This geofence records only entry transitions
        Geofence.GEOFENCE_TRANSITION_ENTER);
    // Store this flat version
    mGeofenceStorage.setGeofence("1", mUIGeofence1);
    // Create another internal object. Set its ID to "2"
    mUIGeofence2 = new SimpleGeofence(
        "2",
        Double.valueOf(mLatitude2.getText().toString()),
        Double.valueOf(mLongitude2.getText().toString()),
        Float.valueOf(mRadius2.getText().toString()),
        GEOFENCE_EXPIRATION_TIME,
        // This geofence records both entry and exit transitions
        Geofence.GEOFENCE_TRANSITION_ENTER |
        Geofence.GEOFENCE_TRANSITION_EXIT);
    // Store this flat version
    mGeofenceStorage.setGeofence(2, mUIGeofence2);
    mGeofenceList.add(mUIGeofence1.toGeofence());
    mGeofenceList.add(mUIGeofence2.toGeofence());
}
...
}

```

除了这些你要监视的[Geofence](#)列表之外，你还需要为Location Services添加[Intent](#)，这个Intent在你的应用探测到地理围栏触发事件时会将这个事件发送给你的应用。

为地理围栏触发事件定义Intent

从Location Services发送来的Intent能够触发各种应用内的动作，但是不能用它来打开一个Activity或者Fragment，因为应用内的组件只能在响应用户动作时才能可见。大多数情况下，处理这一类的Intent最好使用[IntentService](#)。一个[IntentService](#)可以推送一个通知，可以进行长时的后台作业，可以将intent发送给其他的services，还可以广播intent。下面的代码展示了如何定义一个[PendingIntent](#)来启动一个IntentService:

```

public class MainActivity extends FragmentActivity {
    ...
    /**
     * Create a PendingIntent that triggers an IntentService in your
     * app when a geofence transition occurs.
     */
    private PendingIntent getTransitionPendingIntent() {
        // Create an explicit Intent
        Intent intent = new Intent(this,
            ReceiveTransitionsIntentService.class);

        /*
         * Return the PendingIntent
         */
        return PendingIntent.getService(
            this,
            0,
            intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
    }
    ...
}

```

现在你已经拥有了所有发送监视地理围栏请求给Location Services的代码了。

发送监视请求

发送监视请求需要两个异步操作。第一个操作就是为这个请求获取一个location client，第二个操作就是使用这个client来生成请求。这两个操作里面，Location Services都会在操作结束的时候调用一个回调函数。处理这些操作最好的方法就是将这些方法调用连接起来。下面的代码展示了如何建立一个Activity，接着定义回调方法，然后以合适的顺序调用他们。首先，让Activity实现必要的回调接口。需要添加以下接口：[ConnectionCallbacks](#)

- 设置当location client已连接或者断开连接时Location Services调用的方法。

[OnConnectionFailedListener](#)

- 设置当Location Services连接location client出错时Location Services调用的方法。

[OnAddGeofencesResultListener](#)

- 设置当Location Services已经添加地理围栏的时候Location Services调用的方法。

例如：

```
public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
}
```

开始请求进程

接下啦，在连接Location Services的时候定义一个启动请求进程的方法。记得将这个请求设置为全局变量，这样就可以让你使用回调方法[ConnectionCallbacks.onConnected\(\)](#)来添加地理围栏，或者移除地理围栏。

为了防止当你的应用在第一个请求还没结束就开始第二个请求的时候不出现竞争状况，你可以定义一个boolean标志位来记录当前请求的状态：

```
public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    // Holds the location client
    private LocationClient mLocationClient;
    // Stores the PendingIntent used to request geofence monitoring
    private PendingIntent mGeofenceRequestIntent;
    // Defines the allowable request types.
    public enum REQUEST_TYPE = {ADD}
    private REQUEST_TYPE mRequestType;
    // Flag that indicates if a request is underway.
    private boolean mInProgress;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Start with the request flag set to false
        mInProgress = false;
        ...
    }
    ...
    /**
     * Start a request for geofence monitoring by calling
     * LocationClient.connect().
     */
    public void addGeofences() {
        // Start a request to add geofences
        mRequestType = ADD;
        /*
         * Test for Google Play services after setting the request type.
         * If Google Play services isn't present, the proper request
         * can be restarted.
         */
        if (!servicesConnected()) {
            return;
        }
        /*
         * Create a new location client object. Since the current
         * activity class implements ConnectionCallbacks and
         * OnConnectionFailedListener, pass the current activity object
         * as the listener for both parameters
         */
        mLocationClient = new LocationClient(this, this, this)
        // If a request is not already underway
        if (!mInProgress) {
            // Indicate that a request is underway
            mInProgress = true;
            // Request a connection from the client to Location Services
            mLocationClient.connect();
        } else {
```

```

        /*
        * A request is already underway. You can handle
        * this situation by disconnecting the client,
        * re-setting the flag, and then re-trying the
        * request.
        */
    }
}
...
}

```

发送添加地理围栏的请求

在你回调方法[ConnectionCallbacks.onConnected\(\)](#)的实现里面，调用[LocationClient.addGeofences\(\)](#)。注意如果连接失败，[onConnected\(\)](#)方法不会被调用，这个请求也会停止。

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /*
    * Provide the implementation of ConnectionCallbacks.onConnected()
    * Once the connection is available, send a request to add the
    * Geofences
    */
    @Override
    private void onConnected(Bundle dataBundle) {
        ...
        switch (mRequestType) {
            case ADD :
                // Get the PendingIntent for the request
                mTransitionPendingIntent =
                    getTransitionPendingIntent();
                // Send a request to add the current geofences
                mLocationClient.addGeofences(
                    mCurrentGeofences, pendingIntent, this);
                ...
            }
        }
        ...
    }
}

```

注意[addGeofences\(\)](#)方法会直接返回，但是请求的状态却不是直接返回的，只有等到Location Services调用[onAddGeofencesResult\(\)](#)方法。一旦这个方法被调用，你就能知道这个请求是否成功。

通过Location Services检测请求返回的结果

当 Location Services 你对回调函数[onAddGeofencesResult\(\)](#)的实现的时候，说明请求已经结束，你可以检测最终的结果状态码。如果请求成功，那么你轻轻的地理围栏是激活的，如果没有成功，那么你请求的地理围栏没有激活。如果没成功，你需要重试或者报告错误。例如：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /*
    * Provide the implementation of
    * OnAddGeofencesResultListener.onAddGeofencesResult.
    * Handle the result of adding the geofences
    */
    @Override
    public void onAddGeofencesResult(
        int statusCode, String[] geofenceRequestIds) {
        // If adding the geofences was successful
        if (LocationStatusCodes.SUCCESS == statusCode) {
            /*
            * Handle successful addition of geofences here.
            * You can send out a broadcast intent or update the UI.
            * geofences into the Intent's extended data.
            */
        } else {

```

```

        // If adding the geofences failed
        /*
         * Report errors here.
         * You can log the error using Log.e() or update
         * the UI.
         */
    }
    // Turn off the in progress flag and disconnect the client
    mInProgress = false;
    mLocationClient.disconnect();
}
...
}

```

处理断开连接

某些情况下，Location Services可能会在你调用disconnect()方法之前断开连接。为了处理这种情况，你需要实现onDisconnected()方法。在这个方法里面，设置请求状态标志位来表示这个请求已经不处于进程中，然后删除这个client：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /*
     * Implement ConnectionCallbacks.onDisconnected()
     * Called by Location Services once the location client is
     * disconnected.
     */
    @Override
    public void onDisconnected() {
        // Turn off the request flag
        mInProgress = false;
        // Destroy the current location client
        mLocationClient = null;
    }
    ...
}

```

处理连接错误

在处理正常的回调函数之外，你还得提供一个回调函数来处理连接出现错误的情况。这个回调函数重用了前面在检查Google Play service的时候用到的DialogFragment类。它还可以重用之前在onActivityResult()方法里用来接收当用户和错误对话框交互时产生的结果用到的代码。下面的代码展示了如何实现回调函数：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    // Implementation of OnConnectionFailedListener.onConnectionFailed
    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {
        // Turn off the request flag
        mInProgress = false;
        /*
         * If the error has a resolution, start a Google Play services
         * activity to resolve it.
         */
        if (connectionResult.hasResolution()) {
            try {
                connectionResult.startResolutionForResult(
                    this,
                    CONNECTION_FAILURE_RESOLUTION_REQUEST);
            } catch (SendIntentException e) {
                // Log the error
                e.printStackTrace();
            }
        }
        // If no resolution is available, display an error dialog
    } else {
        // Get the error code
        int errorCode = connectionResult.getErrorCode();
        // Get the error dialog from Google Play services
        Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(

```

```
        errorCode,
        this,
        CONNECTION_FAILURE_RESOLUTION_REQUEST);
// If Google Play services can provide an error dialog
if (errorDialog != null) {
    // Create a new DialogFragment for the error dialog
    AlertDialogFragment errorFragment =
        new AlertDialogFragment();
    // Set the dialog in the DialogFragment
    errorFragment.setDialog(errorDialog);
    // Show the error dialog in the DialogFragment
    errorFragment.show(
        getSupportFragmentManager(),
        "Geofence Detection");
}
}
...
}
```

处理地理围栏触发事件

当Location Services探测到用户进入或者退出一个地理围栏，它会发送一个Intent，这个Intent就是

定义一个IntentService

下面的代码展示了如何定义一个当一个地理围栏触发事件出现的时候发送通知。当用户点击这个通知，这个应用的主界面出现：

```
public class ReceiveTransitionsIntentService extends IntentService {
    ...
    /**
     * Sets an identifier for the service
     */
    public ReceiveTransitionsIntentService() {
        super("ReceiveTransitionsIntentService");
    }
    /**
     * Handles incoming intents
     * @param intent The Intent sent by Location Services. This
     * Intent is provided
     * to Location Services (inside a PendingIntent) when you call
     * addGeofences()
     */
    @Override
    protected void onHandleIntent(Intent intent) {
        // First check for errors
        if (LocationClient.hasError(intent)) {
            // Get the error code with a static method
            int errorCode = LocationClient.getErrorCode(intent);
            // Log the error
            Log.e("ReceiveTransitionsIntentService",
                "Location Services error: " +
                    Integer.toString(errorCode));

            /*
             * You can also send the error code to an Activity or
             * Fragment with a broadcast Intent
             */

            /*
             * If there's no error, get the transition type and the IDs
             * of the geofence or geofences that triggered the transition
             */
        } else {
            // Get the type of transition (entry or exit)
            int transitionType =
                LocationClient.getGeofenceTransition(intent);
            // Test that a valid transition was reported
            if (
                (transitionType == Geofence.GEOFENCE_TRANSITION_ENTER)
                ||
                (transitionType == Geofence.GEOFENCE_TRANSITION_EXIT)
            ) {
                List <Geofence> triggerList =
                    getTriggeringGeofences(intent);

                String[] triggerIds = new String[triggerList.size()];

                for (int i = 0; i < triggerList.size(); i++) {
                    // Store the Id of each geofence
                    triggerIds[i] = triggerList.get(i).getRequestId();
                }
                /*
                 * At this point, you can store the IDs for further use
                 * display them, or display the details associated with
                 * them.
                 */

            }
            // An invalid transition was reported
        } else {
            Log.e("ReceiveTransitionsIntentService",
                "Geofence transition error: " +
                    Integer.toString(transitionType));
        }
    }
    ...
}
```

```
}
```

在manifest里面设置IntentService

为了在系统里面申明这个IntentService，在manifest里面添加一个<service> 元素即可。例如：

```
<service
    android:name="com.example.android.location.ReceiveTransitionsIntentService"
    android:label="@string/app_name"
    android:exported="false">
</service>
```

注意你不必为这个service设置intent filters，因为它只接收特定的intent。这些地理围栏触发事件的intent是如何被创建的，请参看[发送监视请求](#)这一课。

停止地理围栏监视

要停止地理围栏监视，你要移除这些地理围栏。你可以移除特定的某个地理围栏集合或者移除与某个[PendingIntent](#)相关所有的地理围栏。这个过程与添加地理围栏类似。第一个操作就是获取一个移除请求的location client，然后使用这个client来生成请求。

Location Services在它完成移除地理围栏这个过程的时候调用的回调函数定义在[LocationClient.OnRemoveGeofencesResultListener](#)这个接口里面。在你的类里面申明这个接口，然后为它的两个方法添加定义：

[onRemoveGeofencesByPendingIntentResult\(\)](#)

- 这个回调函数是Location Services完成移除所有地理围栏的请求时调用的，它是由 [removeGeofences\(PendingIntent, LocationClient.OnRemoveGeofencesResultListener\)](#)方法生成的。

[onRemoveGeofencesByRequestIdsResult\(List, LocationClient.OnRemoveGeofencesResultListener\)](#)

- 这个回调函数是Location Services完成移除特定地理围栏ID集合的地理围栏的请求时调用的，它由 [removeGeofences\(List, LocationClient.OnRemoveGeofencesResultListener\)](#)方法生成的。

这些实现的代码将在下一个代码区域出现。

移除所有的地理围栏

因为移除地理围栏使用了添加地理围栏时的一些方法，你只需要添加另外几种请求类型即可：

```
public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    // Enum type for controlling the type of removal requested
    public enum REQUEST_TYPE = {ADD, REMOVE_INTENT}
    ...
}
```

在连接上Location Services的时候启动移除的请求。如果连接失败，那么[onConnected\(\)](#)方法就不会被调用，请求也就停止了。下面的代码就展示了如何启动这个请求：

```
public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /**
     * Start a request to remove geofences by calling
     * LocationClient.connect()
     */
    public void removeGeofences(PendingIntent requestIntent) {
        // Record the type of removal request
        mRequestType = REMOVE_INTENT;
        /*
         * Test for Google Play services after setting the request type.
         * If Google Play services isn't present, the request can be
         * restarted.
         */
        if (!servicesConnected()) {
            return;
        }
        // Store the PendingIntent
        mGeofenceRequestIntent = requestIntent;
        /*
         * Create a new location client object. Since the current
         * activity class implements ConnectionCallbacks and
         * OnConnectionFailedListener, pass the current activity object
         * as the listener for both parameters
         */
        mLocationClient = new LocationClient(this, this, this);
        // If a request is not already underway
        if (!mInProgress) {
            // Indicate that a request is underway
            mInProgress = true;
            // Request a connection from the client to Location Services
            mLocationClient.connect();
        } else {
        }
```



```

        /*
        * A request is already underway. You can handle
        * this situation by disconnecting the client,
        * re-setting the flag, and then re-trying the
        * request.
        */
    }
}
...
}

```

当Location Services调用这个函数的时候就表明连接已经打开，可以进行移除所有地理围栏的请求了。在这个请求完成之后就可以断开连接了。例如：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /**
     * Once the connection is available, send a request to remove the
     * Geofences. The method signature used depends on which type of
     * remove request was originally received.
     */
    private void onConnected(Bundle dataBundle) {
        /*
        * Choose what to do based on the request type set in
        * removeGeofences
        */
        switch (mRequestType) {
            ...
            case REMOVE_INTENT :
                mLocationClient.removeGeofences(
                    mGeofenceRequestIntent, this);
                break;
            ...
        }
    }
    ...
}

```

对[removeGeofences\(PendingIntent, LocationClient.OnRemoveGeofencesResultListener\)](#)会直接返回，而移除地理围栏的请求的结果要等到Location Services 调用onRemoveGeofencesByPendingIntentResult()方法才会返回。下面的代码展示了如何定义这个方法：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /**
     * When the request to remove geofences by PendingIntent returns,
     * handle the result.
     *
     * @param statusCode the code returned by Location Services
     * @param requestIntent The Intent used to request the removal.
     */
    @Override
    public void onRemoveGeofencesByPendingIntentResult(int statusCode,
        PendingIntent requestIntent) {
        // If removing the geofences was successful
        if (statusCode == LocationStatusCodes.SUCCESS) {
            /*
            * Handle successful removal of geofences here.
            * You can send out a broadcast intent or update the UI.
            * geofences into the Intent's extended data.
            */
        } else {
            // If adding the geocodes failed
            /*
            * Report errors here.
            * You can log the error using Log.e() or update
            * the UI.
            */
        }
    }
}

```

```

        /*
         * Disconnect the location client regardless of the
         * request status, and indicate that a request is no
         * longer in progress
         */
        mInProgress = false;
        mLocationClient.disconnect();
    }
    ...
}

```

移除特定的地理围栏

移除个别地理围栏和地理围栏集合与移除所有地理围栏的过程类似。为了确定你要移除的地理围栏，需要将他们的地理围栏ID存储到一个字符串列表里面。将这个列表传给removeGeofences方法。这个方法接下来就可以启动这个移除过程了。

开始的时候要添加一个请求类型来申明这是一个删除里一个列表里面的地理围栏请求。同时还要全局变量来保存地理围栏的ID列表：

```

...
// Enum type for controlling the type of removal requested
public enum REQUEST_TYPE = {ADD, REMOVE_INTENT, REMOVE_LIST}
// Store the list of geofence Ids to remove
String<List> mGeofencesToRemove;

```

接下来，定义一个你要移除的地理围栏列表。例如，下面的代码就移除了地理围栏ID为1的地理围栏：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    List<String> listOfGeofences =
        Collections.singletonList("1");
    removeGeofences(listOfGeofences);
    ...
}

```

下面的代码定义了removeGeofences()方法：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /**
     * Start a request to remove monitoring by
     * calling LocationClient.connect()
     */
    public void removeGeofences(List<String> geofenceIds) {
        // If Google Play services is unavailable, exit
        // Record the type of removal request
        mRequestType = REMOVE_LIST;
        /*
         * Test for Google Play services after setting the request type.
         * If Google Play services isn't present, the request can be
         * restarted.
         */
        if (!servicesConnected()) {
            return;
        }
        // Store the list of geofences to remove
        mGeofencesToRemove = geofenceIds;
        /*
         * Create a new location client object. Since the current
         * activity class implements ConnectionCallbacks and
         * OnConnectionFailedListener, pass the current activity object
         * as the listener for both parameters
         */
        mLocationClient = new LocationClient(this, this, this);
        // If a request is not already underway
        if (!mInProgress) {

```

```

        // Indicate that a request is underway
        mInProgress = true;
        // Request a connection from the client to Location Services
        mLocationClient.connect();
    } else {
        /*
         * A request is already underway. You can handle
         * this situation by disconnecting the client,
         * re-setting the flag, and then re-trying the
         * request.
         */
    }
}
...
}

```

当Location Services 调用这个回调函数说明这个连接成功，可以进行移除一个列表的地理围栏请求了。完成请求后就可以断开连接。例如：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    private void onConnected(Bundle dataBundle) {
        ...
        switch (mRequestType) {
            ...
            // If removeGeofencesById was called
            case REMOVE_LIST :
                mLocationClient.removeGeofences(
                    mGeofencesToRemove, this);
                break;
            ...
        }
        ...
    }
    ...
}

```

定义 [onRemoveGeofencesByRequestIdsResult\(\)](#) 方法的实现。Location Services 调用这个方法的时候说明移除一个列表的地理围栏的请求已经完成了。在这个方法里面，检测得到的状态码并作出对应的动作：

```

public class MainActivity extends FragmentActivity implements
    ConnectionCallbacks,
    OnConnectionFailedListener,
    OnAddGeofencesResultListener {
    ...
    /**
     * When the request to remove geofences by IDs returns, handle the
     * result.
     *
     * @param statusCode The code returned by Location Services
     * @param geofenceRequestIds The IDs removed
     */
    @Override
    public void onRemoveGeofencesByRequestIdsResult(
        int statusCode, String[] geofenceRequestIds) {
        // If removing the geocodes was successful
        if (LocationStatusCodes.SUCCESS == statusCode) {
            /*
             * Handle successful removal of geofences here.
             * You can send out a broadcast intent or update the UI.
             * geofences into the Intent's extended data.
             */
        } else {
            // If removing the geofences failed
            /*
             * Report errors here.
             * You can log the error using Log.e() or update
             * the UI.
             */
        }
        // Indicate that a request is no longer in progress
    }
}

```

```
mInProgress = false;
// Disconnect the location client
mLocationClient.disconnect();
}
...
}
```

你可以将地理围栏同其他位置感知的特性结合起来，比如周期性的位置更新或者用户的活动识别。

下一课，[识别用户的活动状态](#)将会告诉你如何请求和接受活动更新。Location Services 会以一个正常的频率向你发送当前用户的身体活动信息。基于这些信息，你可以改变你的应用的一些行为。；例如，当你探测到用户由驾驶改为步行时，你可以把应用的信息更新频率设置为更长的时间。

编写:

校对:

识别用户的当下活动

编写:

校对:

使用模拟位置进行测试

编写:

校对:

编写:

校对:

设计高效的导航

编写:

校对:

规划屏幕界面与他们之间的关系

编写:

校对:

为多种大小的屏幕进行规划

编写:

校对:

提供向下与侧滑的导航

编写:

校对:

提供向上与暂时的导航

编写:

校对:

综合上面所有的导航

编写:

校对:

实现高效的导航

编写:

校对:

使用**Tabs**创建**Swipe**视图

编写:

校对:

创建抽屉导航

编写:

校对:

提供向上的导航

编写:

校对:

提供向后的导航

编写:

校对:

实现向下的导航

编写:[fastcome1985](#)

校对:

通知提示用户

- Notification是一种在你APP常规UI外展示、用来指示某个事件发生的用户交互元素。用户可以在使用其它apps时查看notification，并在方便的时候做出回应。
- [Notification设计指导](#)向你展示如何设计实用的notifications以及何时使用它们。这节课将会教你实现大多数常用的notification设计。
- 完整的Demo示例：[NotifyUser.zip](#)

Lessons

- [建立一个Notification](#) 学习如何创建一个notification [Builder](#)，设置需要的特征，以及发布notification。
- [当Activity启动时保留导航](#) 学习如何为一个从notification启动的[Activity](#)执行适当的导航。
- [更新notifications](#) 学习如何更新与移除notifications
- [使用BigView风格](#) 学习用扩展的notification来创建一个BigView，并且维持老版本的兼容性。
- [在notification中展示进度](#) 学习在notification中显示某个操作的进度，既可以用于那些你可以估算已经完成多少（确定进度，determinate）的操作，也可以用于那些你无法知道完成了多少（不确定进度，indefinite）的操作

编写:[fastcome1985](#)

校对:

建立一个Notification

- 这节课向你说明如何创建与发布一个Notification。
- 这节课的例子是基于[NotificationCompat.Builder](#)类的，[NotificationCompat.Builder](#)在[Support Library](#)中。为了给许多各种不同的平台提供最好的notification支持，你应该使用[NotificationCompat](#)以及它的子类，特别是[NotificationCompat.Builder](#)。

创建Notification Builder

- 创建Notification时，可以用[NotificationCompat.Builder](#)对象指定Notification的UI内容与行为。一个[Builder](#)至少包含以下内容：
 - 一个小的icon，用[setSmallIcon\(\)](#)方法设置
 - 一个标题，用[setContentTitle\(\)](#)方法设置。
 - 详细的文本，用[setContentText\(\)](#)方法设置

比如：

```
NotificationCompat.Builder mBuilder =  
    new NotificationCompat.Builder(this)  
        .setSmallIcon(R.drawable.notification_icon)  
        .setContentTitle("My notification")  
        .setContentText("Hello World!");
```


定义Notification的Action（行为）

- 尽管在Notification中Actions是可选的，但是你应该至少添加一种Action。一种Action可以让用户从Notification直接进入你应用内的[Activity](#)，在这个activity中他们可以查看引起Notification的事件或者做下一步的处理。在Notification中，action本身是由[PendingIntent](#)定义的，PendingIntent包含了一个启动你应用内[Activity](#)的[Intent](#)。
- 如何构建一个[PendingIntent](#)取决于你要启动的[activity](#)的类型。当从Notification中启动一个[activity](#)时，你必须保存用户的导航体验。在下面的代码片段中，点击Notification启动一个新的activity，这个activity有效地扩展了Notification的行为。在这种情形下，就没必要人为地去创建一个返回栈（更多关于这方面的信息，请查看 [Preserving Navigation when Starting an Activity](#)）

```
Intent resultIntent = new Intent(this, ResultActivity.class);
...
// Because clicking the notification opens a new ("special") activity, there's
// no need to create an artificial back stack.
PendingIntent resultPendingIntent =
    PendingIntent.getActivity(
        this,
        0,
        resultIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
    );
```

设置Notification的点击行为

- 可以通过调用[NotificationCompat.Builder](#)中合适的方法，将上一步创建的[PendingIntent](#)与一个手势产生关联。比方说，当点击Notification抽屉里的Notification文本时，启动一个activity，可以通过调用[setContentIntent\(\)](#)方法把[PendingIntent](#)添加进去。

比如：

```
PendingIntent resultPendingIntent;  
...  
mBuilder.setContentIntent(resultPendingIntent);
```

发布Notification

- 为了发布notification：
 - 获取一个[NotificationManager](#)实例
 - 使用[notify\(\)](#)方法发布Notification。当你调用[notify\(\)](#)方法时，指定一个notification ID。你可以在以后使用这个ID来更新你的notification。这在[Managing Notifications](#)中有更详细的描述。
 - 调用[build\(\)](#)方法，会返回一个包含你的特征的[Notification](#)对象。

举个例子：

```
NotificationCompat.Builder mBuilder;
...
// Sets an ID for the notification
int mNotificationId = 001;
// Gets an instance of the NotificationManager service
NotificationManager mNotifyMgr =
    (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
// Builds the notification and issues it.
mNotifyMgr.notify(mNotificationId, mBuilder.build());
```

编写:[fastcome1985](#)

校对:

启动Activity时保留导航

- 部分设计一个notification的目的是为了保持用户的导航体验。为了详细讨论这个课题，请看 [Notifications](#) API引导，分为下列两种主要情况：
 - 常规的activity
你启动的是你application工作流中的一部分[Activity](#)。
 - 特定的activity
用户只能从notification中启动，才能看到这个[Activity](#)，在某种意义上，这个[Activity](#)是notification的扩展，额外展示了一些notification本身难以展示的信息。

设置一个常规的Activity PendingIntent

- 设置一个直接启动的入口Activity的PendingIntent，遵循以下步骤：

1 在manifest中定义你application的[Activity](#)层次，最终的manifest文件应该像这个：

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

2 在基于启动[Activity](#)的[Intent](#)中创建一个返回栈，比如：

```
int id = 1;
...
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
// Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
    stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
...
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
builder.setContentIntent(resultPendingIntent);
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(id, builder.build());
```

设置一个特定的Activity PendingIntent

- 一个特定的Activity不需要一个返回栈，所以你不需要在manifest中定义Activity的层次，以及你不需要调用[addParentStack\(\)](#)方法去构建一个返回栈。作为代替，你需要用manifest设置Activity任务选项，以及调用[getActivity\(\)](#)创建PendingIntent

1 manifest中，在Activity的 标签中增加下列属性：

[android:name="activityclass"](#)

activity的完整的类名。

[android:taskAffinity=""](#)

结合你在代码里设置的[FLAG_ACTIVITY_NEW_TASK](#)标识，确保这个Activity不会进入application的默认任务。任何与application的默认任务有密切关系的任务都不会受到影响。

[android:excludeFromRecents="true"](#)

将新任务从最近列表中排除，目的是为了防止用户不小心返回到它。

2 建立以及发布notification：

a.创建一个启动Activity的Intent.

b.通过调用[setFlags\(\)](#)方法并设置标识[FLAG_ACTIVITY_NEW_TASK](#)与[FLAG_ACTIVITY_CLEAR_TASK](#)，来设置Activity在一个新的，空的任务中启动。

c.在Intent中设置其他你需要的选项。d.通过调用[getActivity\(\)](#)方法从Intent中创建一个PendingIntent，你可以把这个PendingIntent 当做 [setContentIntent\(\)](#)的参数来使用。

- 下面的代码片段演示了这个过程：

```
// Instantiate a Builder object.
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
// Creates an Intent for the Activity
Intent notifyIntent =
    new Intent(new ComponentName(this, ResultActivity.class));
// Sets the Activity to start in a new, empty task
notifyIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
    Intent.FLAG_ACTIVITY_CLEAR_TASK);
// Creates the PendingIntent
PendingIntent notifyIntent =
    PendingIntent.getActivity(
        this,
        0,
        notifyIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
    );

// Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
// Notifications are issued by sending them to the
// NotificationManager system service.
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Builds an anonymous Notification object from the builder, and
// passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());
```

编写:[fastcome1985](#)

校对:

更新Notification

- 当你需要对同一事件发布多次Notification时，你应该避免每次都生成一个全新的Notification。相反，你应该考虑去更新先前的Notification，或者改变它的值，或者增加一些值，或者两者同时进行。
- 下面的章节描述了如何更新Notifications，以及如何移除它们。

改变一个Notification

- 想要设置一个可以被更新的Notification，需要在发布它的时候调用[NotificationManager.notify\(ID, notification\)](#)方法为它指定一个notification ID。更新一个已经发布的Notification，需要更新或者创建一个[NotificationCompat.Builder](#)对象，并从这个对象创建一个[Notification](#)对象，然后用与先前一样的ID去发布这个[Notification](#)。
- 下面的代码片段演示了更新一个notification来反映事件发生的次数，它把notification堆积起来，显示一个总数。

```
mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Sets an ID for the notification, so it can be updated
int notifyID = 1;
mNotifyBuilder = new NotificationCompat.Builder(this)
    .setContentTitle("New Message")
    .setContentText("You've received new messages.")
    .setSmallIcon(R.drawable.ic_notify_status)
numMessages = 0;
// Start of a loop that processes data and then notifies the user
...
    mNotifyBuilder.setContentText(currentText)
        .setNumber(++numMessages);
    // Because the ID remains unchanged, the existing notification is
    // updated.
    mNotificationManager.notify(
        notifyID,
        mNotifyBuilder.build());
...
```

移除Notification

- Notifications 将持续可见，除非下面任何一种情况发生。

- * 用户清除Notification单独地或者使用“清除所有”（如果Notification能被清除）。
- * 你在创建notification时调用了 `setAutoCancel()` ([developer.android.com/reference/android/support/v4/app/Notification.Builder#setAutoCancel\(boolean\)](http://developer.android.com/reference/android/support/v4/app/Notification.Builder#setAutoCancel(boolean)))
- * 你为一个指定的 notification ID调用了`cancel()` ([developer.android.com/reference/android/app/NotificationManager.html#cancel\(int\)](http://developer.android.com/reference/android/app/NotificationManager.html#cancel(int)))
- * 你调用了`cancelAll()` ([developer.android.com/reference/android/app/NotificationManager.html#cancelAll\(\)](http://developer.android.com/reference/android/app/NotificationManager.html#cancelAll()))

编写:[fastcome1985](#)

校对:

使用BigView样式

- Notification抽屉中的Notification主要有两种视觉展示形式，normal view（平常的视图，下同）与 big view（大视图，下同）。Notification的 big view 样式只有当Notification被扩展时才能出现。当Notification在Notification抽屉的最上方或者用户点击Notification时才会展现大视图。
- Big views在Android4.1被引进的，它不支持老版本设备。这节课叫你如何让把big view notifications合并进你的APP，同时提供normal view的全部功能。更多信息请见[Notifications API guide](#)。
- 这是一个 normal view的例子

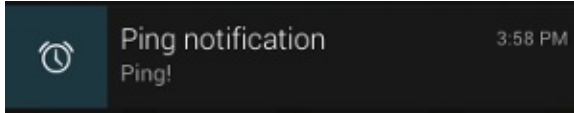


图1 Normal view notification.

- 这是一个 big view的例子

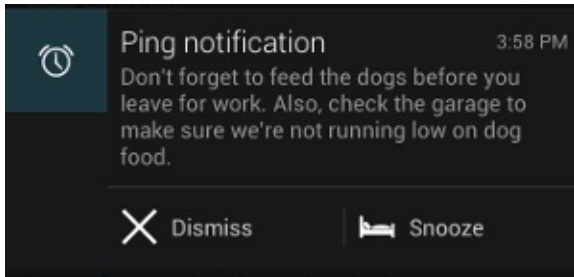


图2 Big view notification.

- 在这节课的例子应用中，normal view 与 big view给用户相同的功能：
 - 继续小睡或者消除Notification
 - 一个查看用户设置的类似计时器的提醒文字的方法，
- normal view 通过当用户点击Notification来启动一个新的activity的方式提供这些特性，记住当你设计你的notifications时，首先在normal view 中提供这些功能，因为很多用户会与notification交互。

设置Notification用来登陆一个新的Activity

- 这个例子应用用[IntentService](#)的子类（PingService）来构造以及发布notification。
- 在这个代码片段中，[IntentService](#)中的方法[onHandleIntent\(\)](#) 指定了当用户点击notification时启动一个新的activity。方法[setContentIntent\(\)](#)定义了pending intent在用户点击notification时被激发，因此登陆这个activity。

```
Intent resultIntent = new Intent(this, ResultActivity.class);
resultIntent.putExtra(CommonConstants.EXTRA_MESSAGE, msg);
resultIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
    Intent.FLAG_ACTIVITY_CLEAR_TASK);

// Because clicking the notification launches a new ("special") activity,
// there's no need to create an artificial back stack.
PendingIntent resultPendingIntent =
    PendingIntent.getActivity(
        this,
        0,
        resultIntent,
        PendingIntent.FLAG_UPDATE_CURRENT
    );

// This sets the pending intent that should be fired when the user clicks the
// notification. Clicking the notification launches a new activity.
builder.setContentIntent(resultPendingIntent);
```

构造big view

- 这个代码片段展示了如何在big view中设置buttons

```
// Sets up the Snooze and Dismiss action buttons that will appear in the
// big view of the notification.
Intent dismissIntent = new Intent(this, PingService.class);
dismissIntent.setAction(CommonConstants.ACTION_DISMISS);
PendingIntent piDismiss = PendingIntent.getService(this, 0, dismissIntent, 0);

Intent snoozeIntent = new Intent(this, PingService.class);
snoozeIntent.setAction(CommonConstants.ACTION_SNOOZE);
PendingIntent piSnooze = PendingIntent.getService(this, 0, snoozeIntent, 0);
```

- 这个代码片段展示了如何构造一个Builder对象，它设置了big view 的样式为"big text",同时设置了它的内容为提醒文字。它使用[addAction\(\)](#)方法来添加将要在big view中出现的Snooze与Dismiss按钮（以及它们相关联的pending intents）。

```
// Constructs the Builder object.
NotificationCompat.Builder builder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_stat_notification)
        .setContentTitle(getString(R.string.notification))
        .setContentText(getString(R.string.ping))
        .setDefaults(Notification.DEFAULT_ALL) // requires VIBRATE permission
    /*
     * Sets the big view "big text" style and supplies the
     * text (the user's reminder message) that will be displayed
     * in the detail area of the expanded notification.
     * These calls are ignored by the support library for
     * pre-4.1 devices.
     */
    .setStyle(new NotificationCompat.BigTextStyle()
        .bigText(msg))
    .addAction(R.drawable.ic_stat_dismiss,
        getString(R.string.dismiss), piDismiss)
    .addAction(R.drawable.ic_stat_snooze,
        getString(R.string.snooze), piSnooze);
```

编写:[fastcome1985](#)

校对:

显示Notification进度

- Notifications可以包含一个展示用户正在进行的操作状态的动画进度指示器。如果你可以在任何时候估算这个操作得花多少时间以及当前已经完成多少，你可以用“determinate（确定的，下同）”形式的指示器（一个进度条）。如果你不能估算这个操作的长度，使用“indeterminate（不确定，下同）”形式的指示器（一个活动的指示器）。
- 进度指示器用[ProgressBar](#)平台实现类来显示。
- 使用进度指示器，可以调用 [setProgress\(\)](#)方法。determinate 与 indeterminate形式将在下面的章节中介绍。

展示固定长度的进度指示器

- 为了展示一个确定长度的进度条，调用 `setProgress(max, progress, false)` 方法将进度条添加进notification，然后发布这个notification，第三个参数是个boolean类型，决定进度条是 indeterminate (true) 还是 determinate (false)。在你操作进行时，增加progress，更新notification。在操作结束时，progress应该等于max。一个常用的调用 `setProgress()` 的方法是设置max为100，然后增加progress就像操作的“完成百分比”。
- 当操作完成的时候，你可以选择或者让进度条继续展示，或者移除它。无论哪种情况下，记得更新notification的文字来显示操作完成。移除进度条，调用 `setProgress(0, 0, false)` 方法。比如：

```
int id = 1;
...
mNotifManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mBuilder = new NotificationCompat.Builder(this);
mBuilder.setContentTitle("Picture Download")
    .setContentText("Download in progress")
    .setSmallIcon(R.drawable.ic_notification);
// Start a lengthy operation in a background thread
new Thread(
    new Runnable() {
        @Override
        public void run() {
            int incr;
            // Do the "lengthy" operation 20 times
            for (incr = 0; incr <= 100; incr+=5) {
                // Sets the progress indicator to a max value, the
                // current completion percentage, and "determinate"
                // state
                mBuilder.setProgress(100, incr, false);
                // Displays the progress bar for the first time.
                mNotifManager.notify(id, mBuilder.build());
                // Sleeps the thread, simulating an operation
                // that takes time
                try {
                    // Sleep for 5 seconds
                    Thread.sleep(5*1000);
                } catch (InterruptedException e) {
                    Log.d(TAG, "sleep failure");
                }
            }
            // When the loop is finished, updates the notification
            mBuilder.setContentText("Download complete")
            // Removes the progress bar
            .setProgress(0,0,false);
            mNotifManager.notify(id, mBuilder.build());
        }
    }
).start();
```

- 结果notifications显示在图1中，左边是操作正在进行中的notification的快照，右边是操作已经完成的notification的快照。

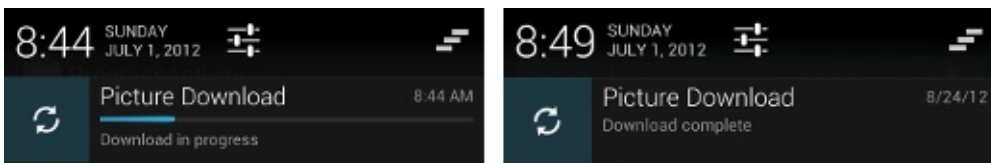


图1 操作正在进行中与完成时的进度条

展示持续的活动的指示器

- 为了展示一个持续的(indeterminate)活动的指示器,用`setProgress(0, 0, true)`方法把指示器添加进notification, 然后发布这个notification。前两个参数忽略, 第三个参数决定indicator 还是 indeterminate。结果是指示器与进度条有同样的样式, 除了它的动画正在进行。
- 在操作开始的时候发布notification, 动画将会一直进行直到你更新notification。当操作完成时, 调用 `setProgress(0, 0, false)` 方法, 然后更新notification来移除这个动画指示器。一定要这么做, 否责即使你操作完成了, 动画还是会在那运行。同时也要记得更新notification的文字来显示操作完成。
- 为了观察持续的活动的指示器是如何工作的, 看前面的代码。定位到下面的几行：

```
// Sets the progress indicator to a max value, the current completion
// percentage, and "determinate" state
mBuilder.setProgress(100, incr, false);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

- 将你找到的代码用下面的几行代码代替, 注意 `setProgress()`方法的第三个参数设置成了true,表示进度条是 indeterminate 类型的。

```
// Sets an activity indicator for an operation of indeterminate length
mBuilder.setProgress(0, 0, true);
// Issues the notification
mNotifyManager.notify(id, mBuilder.build());
```

- 结果显示在图2中:

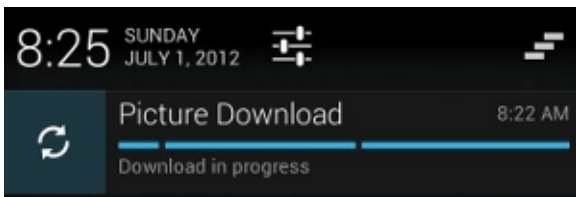


图2 正在进行的活动的指示器

编写:[Lin-H](#)

校对:

增加搜索功能

Android的内置搜索功能，能够在app中方便地为所有用户提供一个统一的搜索体验。根据设备所运行的Android版本，有两种方式可以在你的app中实现搜索。本节课程涵盖如何像Android 3.0中介绍的那样用[SearchView](#)添加搜索，使用系统提供的默认搜索框来向下兼容旧版本Android。

Lessons

建立搜索界面

[学习如何向你的app中添加搜索界面，如何设置activity去处理搜索请求](#)

保存并搜索数据

[学习在SQLite虚拟数据库表中用简单的方法储存和搜索数据](#)

保持向后兼容

[通过使用搜索功能来学习如何向下兼容旧版本设备](#)

编写:[Lin-H](#)

校对:

建立搜索界面

从Android 3.0开始，在action bar中使用[SearchView](#)作为item，是在你的app中提供搜索的一种更好方法。像其他所有在action bar中的item一样，你可以定义[SearchView](#)在有足够空间的时候总是显示，或设置为一个折叠操作(collapsible action),一开始[SearchView](#)作为一个图标显示，当用户点击图标时再显示搜索框占据整个action bar。

Note:在本课程的后面，你会学习对那些不支持[SearchView](#)的设备，如何使你的app向下兼容至Android 2.1版本。

添加SearchView到中action bar中添加

为了在action bar中添加[SearchView](#)，在你的工程目录res/menu/中创建一个名为options_menu.xml的文件，再把下列代码添加到文件中。这段代码定义了如何创建search item，比如使用的图标和item的标题。`collapseActionView`属性允许你的[SearchView](#)占据整个action bar，在不使用的时候折叠成普通的action bar item。由于在手持设备中action bar的空间有限，建议使用`collapsibleActionView`属性来提供更好的用户体验。

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/search"
        android:title="@string/search_title"
        android:icon="@drawable/ic_search"
        android:showAsAction="collapseActionView|ifRoom"
        android:actionViewClass="android.widget.SearchView" />
</menu>
```

Note:如果你的menu item已经有一个XML文件，你可以只把<item>元素添加入文件。

要在action bar中显示[SearchView](#)，把XML菜单资源(res/menu/options_menu.xml)填充到你的activity中的[onCreateOptionsMenu\(\)](#)方法:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    return true;
}
```

如果你立即运行你的app，[SearchView](#)就会显示在你app的action bar中，但还无法使用。你现在需要定义[SearchView](#)如何运行。

创建一个检索配置

[检索配置\(searchable configuration\)](#)在 `res/xml/searchable.xml` 文件中定义了 [SearchView](#) 如何运行。检索配置中至少要包含一个 `android:label` 属性，与 Android manifest 中的 `<application>` 或 `<activity>` `android:label` 属性值相同。但我们还是建议添加 `android:hint` 属性来告诉用户应该在搜索框中输入什么内容：

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

<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_name"
    android:hint="@string/search_hint" />
```

在你的应用的 manifest 文件中，声明一个指向 `res/xml/searchable.xml` 文件的元素，来告诉你的应用在哪里能找到检索配置。在你想要显示 [SearchView](#) 的 `<activity>` 中声明 `<meta-data>` 元素：

```
<activity ... >
    ...
    <meta-data android:name="android.app.searchable"
        android:resource="@xml/searchable" />

</activity>
```

在你之前创建的 `onCreateOptionsMenu()` 方法中，调用 `setSearchableInfo(SearchableInfo)` 把 [SearchView](#) 和检索配置关联在一起：

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    // 关联检索配置和SearchView
    SearchManager searchManager =
        (SearchManager) getSystemService(Context.SEARCH_SERVICE);
    SearchView searchView =
        (SearchView) menu.findItem(R.id.search).getActionView();
    searchView.setSearchableInfo(
        searchManager.getSearchableInfo(getComponentName()));

    return true;
}
```

调用 `getSearchableInfo()` 返回一个 [SearchableInfo](#) 由检索配置 XML 文件创建的对象。检索配置与 [SearchView](#) 正确关联后，当用户提交一个搜索请求时，[SearchView](#) 会以 `ACTION_SEARCH` intent 启动一个 activity。所以你现在需要一个能过滤这个 intent 和处理搜索请求的 activity。

创建一个检索activity

当用户提交一个搜索请求时，[SearchView](#)会尝试以[ACTION_SEARCH](#)启动一个activity。检索activity会过滤[ACTION_SEARCH](#) intent并在某种数据集中根据请求进行搜索。要创建一个检索activity，在你选择的activity中声明对[ACTION_SEARCH](#) intent过滤：

```
<activity android:name=".SearchResultsActivity" ... >
    ...
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    ...
</activity>
```

在你的检索activity中，通过在[onCreate\(\)](#)方法中检查[ACTION_SEARCH](#) intent来处理它。

Note:如果你的检索activity在single top mode下启动(`android:launchMode="singleTop"`)，也要在[onNewIntent\(\)](#)方法中处理[ACTION_SEARCH](#) intent。在single top mode下你的activity只有一个会被创建，而随后启动的activity将不会在栈中创建新的activity。这种启动模式很有用，因为用户可以在当前activity中进行搜索，而不用在每次搜索时都创建一个activity实例。

```
public class SearchResultsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        handleIntent(getIntent());
    }

    @Override
    protected void onNewIntent(Intent intent) {
        ...
        handleIntent(intent);
    }

    private void handleIntent(Intent intent) {

        if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
            String query = intent.getStringExtra(SearchManager.QUERY);
            //通过某种方法，根据请求检索你的数据
        }
    }
    ...
}
```

如果你现在运行你的app，[SearchView](#)就能接收用户的搜索请求，以[ACTION_SEARCH](#) intent启动你的检索activity。现在就由你来解决如何依据请求来储存和搜索数据。

编写:[Lin-H](#)

校对:

保存并搜索数据

有很多方法可以储存你的数据，比如储存在线上的数据库，本地的SQLite数据库，甚至是文本文件。你自己来选择最适合你应用的存储方式。本节课程会向你展示如何创建一个健壮的可以提供全文搜索的SQLite虚拟表。并从一个每行有一组单词-解释对的文件中将数据填入。

创建虚拟表

虚拟表与SQLite表的运行方式类似，但虚拟表是通过回调来向内存中的对象进行读取和写入，而不是通过数据库文件。要创建一个虚拟表，首先为该表创建一个类：

```
public class DatabaseTable {
    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }
}
```

在DatabaseTable类中创建一个继承SQLiteOpenHelper的内部类。你必须重写类SQLiteOpenHelper中定义的abstract方法，才能在必要的时候创建和更新你的数据库表。例如，下面一段代码声明了一个数据库表，用来储存字典app所需的单词。

```
public class DatabaseTable {

    private static final String TAG = "DictionaryDatabase";

    //字典的表中将要包含的列项
    public static final String COL_WORD = "WORD";
    public static final String COL_DEFINITION = "DEFINITION";

    private static final String DATABASE_NAME = "DICTIONARY";
    private static final String FTS_VIRTUAL_TABLE = "FTS";
    private static final int DATABASE_VERSION = 1;

    private final DatabaseOpenHelper mDatabaseOpenHelper;

    public DatabaseTable(Context context) {
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);
    }

    private static class DatabaseOpenHelper extends SQLiteOpenHelper {

        private final Context mHelperContext;
        private SQLiteDatabase mDatabase;

        private static final String FTS_TABLE_CREATE =
            "CREATE VIRTUAL TABLE " + FTS_VIRTUAL_TABLE +
            " USING fts3 (" +
            COL_WORD + ", " +
            COL_DEFINITION + ")";

        DatabaseOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
            mHelperContext = context;
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            mDatabase = db;
            mDatabase.execSQL(FTS_TABLE_CREATE);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                + newVersion + ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS " + FTS_VIRTUAL_TABLE);
            onCreate(db);
        }
    }
}
```


填入虚拟表

现在，表需要数据来储存。下面的代码会向你展示如何读取一个内容为单词和解释的文本文件(位于res/raw/definitions.txt)，如何解析文件与如何将文件中的数据按行插入虚拟表中。为防止UI锁死这些操作会在另一条线程中执行。将下面的一段代码添加到你的DatabaseOpenHelper内部类中。

Tip:你也可以设置一个回调来通知你的UI activity线程的完成结果。

```
private void loadDictionary() {
    new Thread(new Runnable() {
        public void run() {
            try {
                loadWords();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }).start();
}

private void loadWords() throws IOException {
    final Resources resources = mHelperContext.getResources();
    InputStream inputStream = resources.openRawResource(R.raw.definitions);
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));

    try {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] strings = TextUtils.split(line, "-");
            if (strings.length < 2) continue;
            long id = addWord(strings[0].trim(), strings[1].trim());
            if (id < 0) {
                Log.e(TAG, "unable to add word: " + strings[0].trim());
            }
        }
    } finally {
        reader.close();
    }
}

public long addWord(String word, String definition) {
    ContentValues initialValues = new ContentValues();
    initialValues.put(COL_WORD, word);
    initialValues.put(COL_DEFINITION, definition);

    return mDatabase.insert(FTS_VIRTUAL_TABLE, null, initialValues);
}
```

任何恰当的地方，都可以调用loadDictionary()方法向表中填入数据。一个比较好的地方是DatabaseOpenHelper类的onCreate()方法中，紧随创建表之后：

```
@Override
public void onCreate(SQLiteDatabase db) {
    mDatabase = db;
    mDatabase.execSQL(FTS_TABLE_CREATE);
    loadDictionary();
}
```

搜索请求

当你的虚拟表创建好并填入数据后，根据[SearchView](#)提供的请求搜索数据。将下面的方法添加到DatabaseTable类中，用来创建搜索请求的SQL语句：

```
public Cursor getWordMatches(String query, String[] columns) {
    String selection = COL_WORD + " MATCH ?";
    String[] selectionArgs = new String[] {query+"*"};

    return query(selection, selectionArgs, columns);
}

private Cursor query(String selection, String[] selectionArgs, String[] columns) {
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(FTS_VIRTUAL_TABLE);

    Cursor cursor = builder.query(mDatabaseOpenHelper.getReadableDatabase(),
        columns, selection, selectionArgs, null, null, null);

    if (cursor == null) {
        return null;
    } else if (!cursor.moveToFirst()) {
        cursor.close();
        return null;
    }
    return cursor;
}
```

调用getWordMatches()来搜索请求。任何符合的结果返回到[Cursor](#)中，可以直接遍历或是建立一个[ListView](#)。这个例子是在检索activity的handleIntent()方法中调用getWordMatches()。请记住，因为之前创建的intent filter，检索activity会在[ACTION_SEARCH](#) intent中额外接收请求作为变量存储：

```
DatabaseTable db = new DatabaseTable(this);

...

private void handleIntent(Intent intent) {

    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        Cursor c = db.getWordMatches(query, null);
        //执行Cursor并显示结果
    }
}
```

编写:[Lin-H](#)

校对:

保持向下兼容

[SearchView](#)和action bar只在Android 3.0以及以上版本可用。为了支持旧版本平台，你可以回到搜索对话框。搜索框是系统提供的UI，在调用时会覆盖在你的应用的最顶端。

设置最小和目标API级别

要设置搜索对话框，首先在你的manifest中声明你要支持旧版本设备，并且目标平台为Android 3.0或更新版本。当你这么做之后，你的应用会自动地在Android 3.0或以上使用action bar，在旧版本的设备使用传统的目录系统：

```
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="15" />

<application>
  ...
```

为旧版本设备提供搜索对话框

要在旧版本设备中调用搜索对话框，可以在任何时候，当用户从选项目录中选择搜索项时，就会调用[onSearchRequested\(\)](#)。因为Android 3.0或以上会在action bar中显示[SearchView](#)(就像在第一节课中演示的那样)，所以当用户选择目录的搜索项时，只有Android 3.0以下版本的会调用[onOptionsItemSelected\(\)](#)。

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.search:
            onSearchRequested();
            return true;
        default:
            return false;
    }
}
```

在运行时检查Android的构建版本

在运行时，检查设备的版本可以保证在旧版本设备中，不使用不支持的[SearchView](#)。在我们这个例子中，这一操作在[onCreateOptionsMenu\(\)](#)方法中：

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        SearchManager searchManager =
            (SearchManager) getSystemService(Context.SEARCH_SERVICE);
        SearchView searchView =
            (SearchView) menu.findItem(R.id.search).getActionView();
        searchView.setSearchableInfo(
            searchManager.getSearchableInfo(getComponentName()));
        searchView.setIconifiedByDefault(false);
    }
    return true;
}
```

编写:

校对:

使得你的**App**内容可被**Google**搜索

编写:

校对:

为**App**内容开启深度链接

编写:

校对:

为索引指定**App**内容

编写:

校对:

编写:

校对:

为多屏幕设计

编写:River

校对:

兼容不同的屏幕大小

这节课教你如何通过以下几种方式支持多屏幕：

- 1：确保你的布局能自适应屏幕
- 2：根据你的屏幕配置提供合适的UI布局
- 3：确保你当前的布局适合当前的屏幕。
- 4：提供合适的位图（bitmap）

使用“wrap_content”和“match_parent”

为了确保你的布局能灵活的适应不同的屏幕尺寸，针对一些view组件，你应该使用wrap_content和match_parent来设置他们的宽和高。如果你使用了wrap_content，view的宽和高会被设置为该view所包含的内容的大小值。如果是match_parent（在API 8之前是fill_parent）则被设置为该组件的父控件的大小。

通过使用wrap_content和match_parent尺寸值代替硬编码的尺寸，你的视图将分别只使用控件所需要的空间或者被拓展以填充所有有效的空间。比如：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:id="@+id/linearLayout1"
        android:gravity="center"
        android:layout_height="50dp">
        <ImageView
            android:id="@+id/imageView1"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:src="@drawable/logo"
            android:paddingRight="30dp"
            android:layout_gravity="left"
            android:layout_weight="0" />
        <View
            android:layout_height="wrap_content"
            android:id="@+id/view1"
            android:layout_width="wrap_content"
            android:layout_weight="1" />
        <Button
            android:id="@+id/categorybutton"
            android:background="@drawable/button_bg"
            android:layout_height="match_parent"
            android:layout_weight="0"
            android:layout_width="120dp"
            style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment
        android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />
</LinearLayout>
```

注意上面的例子使用wrap_content和match_parent来指定组件尺寸而不是使用固定的尺寸。这样就能使你的布局正确的适配不同的屏幕尺寸和屏幕配置（这里的配置主要是指屏幕的横竖屏切换）。

例如，下图演示的就是该布局在竖屏和横屏模式下的效果，注意组件的尺寸是自动适应宽和高的。



图1：News Reader示例app（左边竖屏，右边横屏）。

使用绝对布局（RelativeLayout）

你可以使用LinearLayout以及wrap_content和match_parent组合来构建复杂的布局，但是LinearLayout却不允许你精准的控制它子view的关系，子view在LinearLayout中只能简单一个接一个的排成行。如果你需要你的子view不只是简简单单的排成行的排列，更好的方法是使用RelativeLayout，它允许你指定你布局中控件与控件之间的关系，比如，你可以指定一个子view在左边，另一个则在屏幕的右边。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```

<TextView
    android:id="@+id/label"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Type here:"/>
<EditText
    android:id="@+id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/label"/>
<Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
    android:layout_alignParentRight="true"
    android:layout_marginLeft="10dp"
    android:text="OK" />
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
</RelativeLayout>

```

图2：QVGA（小尺寸屏幕）屏幕截图

图3：WSVGA（大尺寸屏幕）屏幕截图

注意：尽管组件的尺寸发生了变化，但是它的子view之间的关系还是通过RelativeLayout.LayoutParams已经指定好了。

使用据尺寸限定词

（译者注：这里的限定词只要是指在编写布局文件时，将布局文件放在加上类似large，sw600dp等这样限定词的文件夹中，以此来告诉系统根据屏幕选择对应的布局文件，比如下面例子的layout-large文件夹）

从上一节的学习里程中，我们知道如何编写灵活的布局或者相对布局，它们都能通过拉伸或者填充控件来适应不同的屏幕，但是它们却无法为每个不同屏幕尺寸提供最好的用户体验。因此，你的应用不应该只是实现灵活的布局，同时也应该为不同的屏幕配置提供几种不同的布局方式。你可以通过配置限定（configuration qualifiers）来做这件事情，它能在运行时根据你当前设备的配置（比如不同的屏幕尺寸设计了不同的布局）来选择合适的资源。

比如，很多应用都为大屏幕实现了“两个方框”模式（应用可能在一个方框中实现一个list，另外一个则实现list的内容），平板和电视都是大到能能在一个屏幕上适应两个方框，但是手机屏幕却只能单个显示。所以，如果你想实现这些布局，你就需要以下文件：

res/layout/main.xml.单个方框（默认）布局：

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />

</LinearLayout>

```

res/layout-large/main.xml,两个方框布局：

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"

```

```
        android:layout_width="fill_parent" />
    </LinearLayout>
```

注意第二个布局文件的目录名字“large qualifier”，在大尺寸的设备屏幕时（比如7寸平板或者其他大屏幕的设备）就会选择该布局文件，而其他比较小的设备则会选择没有限定词的另一个布局（也就是第一个布局文件）。

使用最小宽度限定词

在Android 3.2之前，开发者还有一个困难，那就是Android设备的“large”屏幕尺寸，其中包括Dell Streak（设备名称），老版Galaxy Tab和一般的7寸平板，有很多的应用都想针对这些不同的设备（比如5和7寸的设备）定义不同的布局，但是这些设备都被定义为了large尺寸屏幕。也是因为这个，所以Android在3.2的时候开始使用最小宽度限定词。

最小宽度限定词允许你根据设备的最小宽度（dp单位）来指定不同布局。比如，传统的7寸平板最小宽度为600dp，如果你希望你的UI能够在这样的屏幕上显示两个方框（一个方框的显示在小屏幕上），你可以使用上节中提到的同样的两个布局文件，不同的是，使用sw600来指定两个方框的布局使用在最小宽度为600dp的设备上。

res/layout/main.xml, 单个方框（默认）布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />

</LinearLayout>
```

res/layout-sw600dp/main.xml, 两个方框布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp" />
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />
</LinearLayout>
```

这意味着当你的设备的最小宽度等于600dp或者更大时，系统选择layout-sw600dp/main.xml（两个方框）的布局，而小一点的屏幕则会选择layout/main.xml（单个方框）的布局。然而，在3.2之前的设备上，这样做并不是很好的选择。因为3.2之前还没有将sw600dp作为一个限定词出现，所以，你还是需要使用large限定词来做。因此，你还是应该要有一个布局文件名为res/layout-large/main.xml，和res/layout-sw600dp/main.xml一样。在下一节中，你将学到如何避免像这样出现重复的布局文件。

使用布局别名

最小宽度限定词只能在android 3.2或者更高的版本上使用。因此，你还是需要使用抽象尺寸（small, normal, large, xlarge）来兼容以前的版本。比如，你想要将你的UI设计为在手机上只显示一个方框的布局，而在7寸平板或电视，或者其他大屏幕设备上显示多个方框的布局，你可能得提供这些文件：

res/layout/main.xml：单个方框布局

res/layout-large：多个方框布局

res/layout-sw600dp：多个方框布局

最后两个文件都是一样的，因为其中一个将会适配Android 3.2的设备，而另外一个则会适配其他Android低版本的平板或者电视。为了避免这些重复的文件（维护让人感觉头痛就是因为这个），你可以使用别名文件。比如，你可以定义如下布局：res/layout/main.xml，单个方框布局 res/layout/main_twopanes.xml，两个方框布局 然后添加这两个文件：res/values-layout.xml：

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

res/values-sw600dp/layout.xml：

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

最后两个文件拥有相同的内容，但它们并没有真正意义上的定义布局。它们只是将main_twopanes设置成为了别名main，它们分别处在large和sw600dp选择器中，所以它们能适配Android任何版本的平板和电视（在3.2之前平板和电视可以直接匹配large，而3.2或者以上的则匹配sw600dp）。

使用方向限定词

有一些布局不管是在横向还是纵向的屏幕配置中都能显示的非常好，但是更多的时候，适当的调整一下会更好。在News Reader应用例子中，以下是布局在不同屏幕尺寸和方向的行为：

小屏幕，纵向：一个方框加logo 小屏幕，横向：一个方框加logo 7寸平板，纵向：一个方框加action bar 7寸平板，横向：两个宽方框加action bar 10寸平板，纵向：两个窄方框加action bar 10寸平板，横向：两个宽方框加action bar 电视，横向：两个宽方框加action bar

这些每个布局都会再res/layout目录下定义一个xml文件，如此，应用就能根据屏幕配置的变化根据别名匹配到对应的布局来适应屏幕。

res/layout/onpane.xml：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />

</LinearLayout>
```

res/layout/onepane_with_bar.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:id="@+id/linearLayout1"
        android:gravity="center"
        android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:src="@drawable/logo"
            android:paddingRight="30dp"
            android:layout_gravity="left"
            android:layout_weight="0" />
        <View android:layout_height="wrap_content"
            android:id="@+id/view1"
            android:layout_width="wrap_content"
            android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
            android:background="@drawable/button_bg"
            android:layout_height="match_parent"
            android:layout_weight="0"
            android:layout_width="120dp"
            style="@style/CategoryButtonStyle"/>
    </LinearLayout>

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />

</LinearLayout>
```

res/layout/twopanes.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
```

```

        android:orientation="horizontal">
        <fragment android:id="@+id/headlines"
            android:layout_height="fill_parent"
            android:name="com.example.android.newsreader.HeadlinesFragment"
            android:layout_width="400dp"
            android:layout_marginRight="10dp" />
        <fragment android:id="@+id/article"
            android:layout_height="fill_parent"
            android:name="com.example.android.newsreader.ArticleFragment"
            android:layout_width="fill_parent" />
    </LinearLayout>

```

res/layout/twopanes_narrow.xml:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="200dp"
        android:layout_marginRight="10dp" />
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />
</LinearLayout>

```

现在所有可能的布局我们都已经定义了，唯一剩下的问题是使用方向限定词来匹配对应的布局给屏幕。这时候，你就可以使用布局别名的功能了：

res/values/layouts.xml：

```

<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
    <bool name="has_two_panes">false</bool>
</resources>

```

res/values-sw600dp-land/layouts.xml:

```

<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>

```

res/values-sw600dp-port/layouts.xml:

```

<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>

```

res/values-large-land/layouts.xml:

```

<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>

```

res/values-large-port/layouts.xml:

```

<resources>
    <item name="main_layout" type="layout">@layout/twopanes_narrow</item>
    <bool name="has_two_panes">true</bool>
</resources>


```


使用点9图片

支持不同的屏幕尺寸同时也意味着你的图片资源也必须能兼容不同的屏幕尺寸。比如，一个button的背景图片就必须适应该

button的各种形状。

如果你在使用组件时可以改变图像的大小，你很快就会发现这是一个不明确的选择，因为运行的时候，图片会被拉伸或者压缩（这样容易造成图像失真）。避免这种情况的解决方案就是使用点9图片，这是一种能够指定哪些区域能够或者不能够拉伸的特殊png文件。

因此，在设计图像需要与组件一起变大变小时，一定要使用点9。若要将位图转换为点9，你可以用一个普通的图像开始（下图，是在4倍变焦情况下的图像显示）。

你可以通过sdk中的draw9patch程序（位于tools/directory目录下）来画点9图片。通过沿左侧和顶部边框绘制像素来标记应该被拉伸的区域。也可以通过沿右侧和底部边界绘制像素来标记。就像下图所示一样：

请注意，上图沿边界的黑色像素。在顶部边框和左边框的那些表明图像的可拉伸区域，右边和底部边框则表示内容应该放置的地方。

此外，注意.9.png这个格式，你也必须用这个格式，因为框架会检测这是一个点9图片而不是一个普通图片。

当你将这个应用到组件的背景的时候（通过设置`android:background="@drawable/button"`），android框架会自动正确的拉伸图像以适应按钮的大小，下图就是各种尺寸中的显示效果：

编写:

校对:

兼容不同的屏幕密度

编写:

校对:

实现可适应的UI

编写:

校对:

为**TV**进行设计

编写:

校对:

为TV优化Layout

编写:

校对:

为**TV**优化导航

编写:

校对:

处理不支持**TV**的功能

编写:[kesenhoo](#)

校对:

创建自定义View

Android的framework有大量的Views用来与用户进行交互并显示不同种类的数据。但是有时候你的程序有个特殊的需求，而Android内置的views组件并不能实现。这一章节会演示如何创建你自己的views，并使得它们是robust与reusable的。

Dependencies and Prerequisites

Android 2.1 (API level 7) or higher

YOU should also read

- [Custom Components](#)
- [Input Events](#)
- [Property Animation](#)
- [Hardware Acceleration](#)
- [Accessibility](#) developer guide

Try it out

Download the sample [CustomView.zip](#)

Lesson

(1) 创建一个**View**类

Create a class that acts like a built-in view, with custom attributes and support from the ADT layout editor.

(2) 自定义**Drawing**

Make your view visually distinctive using the Android graphics system.

(3) 使得**View**是可交互的

Users expect a view to react smoothly and naturally to input gestures. This lesson discusses how to use gesture detection, physics, and animation to give your user interface a professional feel.

(4) 优化**View**

No matter how beautiful your UI is, users won't love it if it doesn't run at a consistently high frame rate. Learn how to avoid common performance problems, and how to use hardware acceleration to make your custom drawings run faster.

编写:[kesenhoo](#)

校对:

创建自定义的View类

设计良好的类总是相似的。它使用一个好用的接口来封装一个特定的功能，它有效的使用CPU与内存，等等。为了成为一个设计良好的类，自定义的view应该：

- 遵守Android标准规则。
- 提供自定义的风格属性值并能够被Android XML Layout所识别。
- 发出可访问的事件。
- 能够兼容Android的不同平台。

Android的framework提供了许多基类与XML标签用来帮助你创建一个符合上面要求的View。这节课会介绍如何使用Android framework来创建一个view的核心功能。

Subclass a View

Android framework里面定义的view类都继承自View。你自定义的view也可以直接继承View，或者你可以通过继承既有的一个子类(例如Button)来节约一点时间。

为了允许Android Developer Tools能够识别你的view，你必须至少提供一个constructor，它包含一个Context与一个AttributeSet对象作为参数。这个constructor允许layout editor创建并编辑你的view的实例。

```
class PieChart extends View {  
    public PieChart(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
}
```

Define Custom Attributes

为了添加一个内置的View到你的UI上，你需要通过XML属性来指定它的样式与行为。为了实现自定义的view的行为，你应该：

- 为你的view在资源标签下定义自设的属性
- 在你的XML layout中指定属性值
- 在运行时获取属性值
- 把获取到的属性值应用在你的view上

为了定义自设的属性，添加 资源到你的项目中。放置于res/values/attrs.xml文件中。下面是一个attrs.xml文件的示例：

```
<resources>
  <declare-styleable name="PieChart">
    <attr name="showText" format="boolean" />
    <attr name="labelPosition" format="enum">
      <enum name="left" value="0"/>
      <enum name="right" value="1"/>
    </attr>
  </declare-styleable>
</resources>
```

上面的代码声明了2个自设的属性，**showText**与**labelPosition**，它们都归属于PieChart的项目下的styleable实例。styleable实例的名字，通常与自定义的view名字一致。尽管这并没有严格规定要遵守这个convention，但是许多流行的代码编辑器都依靠这个命名规则来提供statement completion。

一旦你定义了自设的属性，你可以在layout XML文件中使用它们。唯一不同的是你自设的属性是归属于不同的命名空间。不是属于<http://schemas.android.com/apk/res/android>的命名空间，它们归属于[http://schemas.android.com/apk/res/\[your package name\]](http://schemas.android.com/apk/res/[your package name])。例如，下面演示了如何为PieChart使用上面定义的属性：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
  <com.example.customviews.charting.PieChart
    custom:showText="true"
    custom:labelPosition="left" />
</LinearLayout>
```

为了避免输入长串的namespace名字，示例上面使用了:custom作为别名，你也可以选择其他的名称作为你的namespace。

请注意，如果你的view是一个inner class，你必须指定这个view的outer class。同样的，如果PieChart有一个inner class叫做PieView。为了使用这个类中自设的属性，你应该使用com.example.customviews.charting.PieChart\$PieView。

Apply Custom Attributes

当view从XML layout被创建的时候，在xml标签下的属性值都是从resource下读取出来并传递到view的constructor作为一个AttributeSet参数。尽管可以从AttributeSet中直接读取数值，可是这样做有些弊端（没有看懂下面的两个原因）：

- 拥有属性的资源并没有经过分解
- Styles并没有运用上

取而代之的是，通过obtainStyledAttributes()来获取属性值。这个方法会传递一个[TypedArray](#)对象，它是间接referenced并且styled的。

请看下面的示例：

```
public PieChart(Context context, AttributeSet attrs) {  
    super(context, attrs);  
    TypedArray a = context.getTheme().obtainStyledAttributes(  
        attrs,  
        R.styleable.PieChart,  
        0, 0);  
  
    try {  
        mShowText = a.getBoolean(R.styleable.PieChart_showText, false);  
        mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);  
    } finally {  
        a.recycle();  
    }  
}
```

请注意TypedArray对象是一个shared资源，必须被在使用后进行回收。

Add Properties and Events

Attributes是一个强大的控制view的行为与外观的方法，但是他们仅仅能够在view被初始化的时候被读取到。为了提供一个动态的行为，需要暴露出一些合适的getter与setter方法。下面的代码演示了如何使用这个技巧：

```
public boolean isShowText() {  
    return mShowText;  
}  
  
public void setShowText(boolean showText) {  
    mShowText = showText;  
    invalidate();  
    requestLayout();  
}
```

请注意，在setShowText方法里面有调用[invalidate\(\)](#) and [requestLayout\(\)](#)。当view的某些内容发生变化时，需要调用invalidate来通知系统对这个view进行redraw，当某些元素变化会引起组件大小变化时，需要调用requestLayout方法。

自定义的view也需要能够支持响应事件的监听器。例如，PieChart暴露了一个自设的事件OnCurrentItemChanged来通知监听器，用户已经切换了焦点到一个新的组件上。

我们很容易忘记了暴露属性与事件，特别是当你是这个view的唯一用户时。请花费一些时间来仔细定义你的view的交互。一个好的规则是总是暴露任何属性与事件。

Design For Accessibility

Your custom view should support the widest range of users. This includes users with disabilities that prevent them from seeing or using a touchscreen. To support users with disabilities, you should:

- Label your input fields using the `android:contentDescription` attribute
- Send accessibility events by calling `sendAccessibilityEvent()` when appropriate.
- Support alternate controllers, such as D-pad and trackball

For more information on creating accessible views, see [Making Applications Accessible](#) in the Android Developers Guide.

编写:[kesenhoo](#)

校对:

实现自定义**View**的绘制

自定义view的最重要的一个部分是自定义它的外观。根据你的程序的需求，自定义绘制动作可能简单也可能很复杂。这节课会演示一些最常见的操作。

Override onDraw()

重绘一个自定义的view的最重要的步骤是重写onDraw()方法。onDraw()的参数是一个Canvas对象。Canvas类定义了绘制文本，线条，图像与许多其他图形的方法。你可以在onDraw方法里面使用那些方法来创建你的UI。

在你调用任何绘制方法之前，你需要创建一个Paint对象。

Create Drawing Objects

android.graphics framework把绘制定义为下面两类：

- 绘制什么，由Canvas控制
- 如何绘制，由Paint控制

例如Canvas提供绘制一条直线的方法，Paint提供直线颜色。所以在绘制之前，你需要创建一个或者多个Paint对象。

```
private void init() {
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mTextPaint.setColor(mTextColor);
    if (mTextHeight == 0) {
        mTextHeight = mTextPaint.getTextSize();
    } else {
        mTextPaint.setTextSize(mTextHeight);
    }

    mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mPiePaint.setStyle(Paint.Style.FILL);
    mPiePaint.setTextSize(mTextHeight);

    mShadowPaint = new Paint(0);
    mShadowPaint.setColor(0xff101010);
    mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));

    ...
}
```

刚开始就创建对象是一个重要的优化技巧。Views会被频繁的重新绘制，初始化许多绘制对象需要花费昂贵的代价。在onDraw方法里面创建绘制对象会严重影响到性能并使得你的UI显得卡顿。

Handle Layout Events

为了正确的绘制你的view，你需要知道view的大小。复杂的自定义view通常需要根据在屏幕上的大小与形状执行多次layout计算。你不应该去估算这个view在屏幕上的显示大小。即使只有一个程序会使用你的view，仍然是需要处理屏幕大小不同，密度不同，方向不同所带来的影响。

尽管view有许多方法是用来计算大小的，但是大多数是不需要重写的。如果你的view不需要特别的控制它的大小，唯一需要重写的方法是[onSizeChanged\(\)](#)。

onSizeChanged()，当你的view第一次被赋予一个大小时，或者你的view大小被更改时会被执行。在onSizeChanged方法里面计算位置，间距等其他与你的view大小值。

```
// Account for padding
float xpad = (float)(getPaddingLeft() + getPaddingRight());
float ypad = (float)(getPaddingTop() + getPaddingBottom());

// Account for the label
if (mShowText) xpad += mTextWidth;

float ww = (float)w - xpad;
float hh = (float)h - ypad;

// Figure out how big we can make the pie.
float diameter = Math.min(ww, hh);
```

如果你想更加精确的控制你的view的大小，需要重写[onMeasure\(\)](#)方法。这个方法的参数是View.MeasureSpec，它会告诉你的view的夫控件的大小。那些值被包装成int类型，你可以使用静态方法来获取其中的信息。

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // Try for a width based on our minimum
    int minw = getPaddingLeft() + getPaddingRight() + getSuggestedMinimumWidth();
    int w = resolveSizeAndState(minw, widthMeasureSpec, 1);

    // Whatever the width ends up being, ask for a height that would let the pie
    // get as big as it can
    int minh = MeasureSpec.getSize(w) - (int)mTextWidth + getPaddingBottom() + getPaddingTop();
    int h = resolveSizeAndState(MeasureSpec.getSize(w) - (int)mTextWidth, heightMeasureSpec, 1);

    setMeasuredDimension(w, h);
}
```

上面的代码有三个重要的事情需要注意：

- 计算的过程有把view的padding考虑进去。这个在后面会提到，这部分是view所控制的。
- 帮助方法resolveSizeAndState()是用来创建最终的宽高值的。这个方法会通过比较view的需求大小与spec值返回一个合适的View.MeasureSpec值，并传递到onMeasure方法中。
- onMeasure()没有返回值。它通过调用setMeasuredDimension()来获取结果。调用这个方法是强制执行的，如果你遗漏了这个方法，会出现运行时异常。

Draw!

每个view的onDraw都是不同的，但是有下面一些常见的操作：

- 绘制文字使用drawText()。指定字体通过调用setTypeface(), 通过setColor()来设置文字颜色。
- 绘制基本图形使用drawRect(), drawOval(), drawArc(). 通过setStyle()来指定形状是否需要filled, outlined.
- 绘制一些复杂的图形，使用Path类. 通过给Path对象添加直线与曲线, 然后使用drawPath()来绘制图形. 和基本图形一样，paths也可以通过setStyle来设置是outlined, filled, both.
- 通过创建LinearGradient对象来定义渐变。调用setShader()来使用LinearGradient。
- 通过使用drawBitmap来绘制图片。

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // Draw the shadow
    canvas.drawOval(
        mShadowBounds,
        mShadowPaint
    );

    // Draw the label text
    canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);

    // Draw the pie slices
    for (int i = 0; i < mData.size(); ++i) {
        Item it = mData.get(i);
        mPiePaint.setShader(it.mShader);
        canvas.drawArc(mBounds,
            360 - it.mEndAngle,
            it.mEndAngle - it.mStartAngle,
            true, mPiePaint);
    }

    // Draw the pointer
    canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
    canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

编写:[kesenhoo](#)

校对:

使得View可交互

绘制UI仅仅是创建自定义View的一部分。你还需要使得你的View能够以模拟现实世界的方式来进行反馈。Objects应该总是与现实情景能够保持一致。例如，图片不应该突然消失又从另外一个地方出现，因为在现实世界里面不会发生那样的事情。正确的应该是，图片从一个地方移动到另外一个地方。

用户应该可以感受到UI上的微小变化，并对这些变化反馈到现实世界中。例如，当用户做fling(迅速滑动)的动作，应该再滑动开始与结束的时候给用户一定的反馈。

这节课会演示如何使用Android framework的功能来为自定义的View添加那些现实世界中的行为。

Handle Input Gestures

像许多其他UI框架一样，Android提供一个输入事件模型。用户的动作会转换成触发一些回调函数的事件，你可以重写这些回调方法来定制你的程序应该如何响应用户的输入事件。在Android中最常用的输入事件是touch，它会触发[onTouchEvent\(android.view.MotionEvent\)](#)的回调。重写这个方法来处理touch事件：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    return super.onTouchEvent(event);
}
```

Touch事件本身并不是特别有用。如今的touch UI定义了touch事件之间的相互作用，叫做gestures。例如tapping,pulling,flinging与zooming。为了把那些touch的源事件转换成gestures, Android提供了[GestureDetector](#)。

下面演示了如何构造一个GestureDetector。

```
class mListener extends GestureDetector.SimpleOnGestureListener {
    @Override
    public boolean onDown(MotionEvent e) {
        return true;
    }
}
mDetector = new GestureDetector(PieChart.this.getContext(), new mListener());
```

不管你是否使用GestureDetector.SimpleOnGestureListener, 你必须总是实现onDown()方法，并返回true。这一步是必须的，因为所有的gestures都是从onDown()开始的。如果你再onDown()里面返回false，系统会认为你想要忽略后续的gesture,那么GestureDetector.OnGestureListener的其他回调方法就不会被执行到了。一旦你实现了GestureDetector.OnGestureListener并且创建了GestureDetector的实例, 你可以使用你的GestureDetector来中止你在onTouchEvent里面收到的touch事件。

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    boolean result = mDetector.onTouchEvent(event);
    if (!result) {
        if (event.getAction() == MotionEvent.ACTION_UP) {
            stopScrolling();
            result = true;
        }
    }
    return result;
}
```

当你传递一个touch事件到onTouchEvent()时，若这个事件没有被认为是gesture中的一部分，它会返回false。你可以执行自定义的gesture-decection代码。

Create Physically Plausible(貌似可信的) Motion

Gestures是控制触摸设备的一种强有力的方式，但是除非你能够产生出一个合理的触摸反馈，否则将是违反用户直觉的。一个很好的例子是fling手势，用户迅速的在屏幕上移动手指然后抬手离开屏幕。这个手势应该使得UI迅速的按照fling的方向进行滑动，然后慢慢停下来，就像是用户旋转一个飞轮一样。

幸运的是，Android有提供帮助类来模拟这些物理行为。

```
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
    mScroller.fling(currentX, currentY, velocityX / SCALE, velocityY / SCALE, minX, minY, maxX,
        maxY);
    postInvalidate();
}
```

Note: 尽管速率是通过GestureDetector来计算的，许多开发者感觉使用这个值使得fling动画太快。通常把x与y设置为4到8倍的关系。

```
if (!mScroller.isFinished()) {
    mScroller.computeScrollOffset();
    setPieRotation(mScroller.getCurY());
}
```

[Scroller](#) 类会为你计算滚动位置，但是他不会自动把哪些位置运用到你的view上面。你有责任确保View获取并运用到新的坐标。你有两种方法来实现这件事情：

- 在调用fling()之后执行postInvalidate(), 这是为了确保能强制进行重画。这个技术需要每次在onDraw里面计算过scroll offsets(滚动偏移量)之后调用postInvalidate()。
- 使用[ValueAnimator](#)

第二个方法使用起来会稍微复杂一点，但是它更有效率并且避免了不必要的重画的view进行重绘。缺点是ValueAnimator是从API Level 11才有的。因此他不能运用到3.0的系统之前的版本上。

Note: ValueAnimator虽然是API 11才有的，但是你还是可以在最低版本低于3.0的系统上使用它，做法是在运行时判断当前的API Level，如果低于11则跳过。

```
mScroller = new Scroller(getContext(), null, true);
mScrollAnimator = ValueAnimator.ofFloat(0, 1);
mScrollAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator valueAnimator) {
        if (!mScroller.isFinished()) {
            mScroller.computeScrollOffset();
            setPieRotation(mScroller.getCurY());
        } else {
            mScrollAnimator.cancel();
            onScrollFinished();
        }
    }
});
```


Make Your Transitions Smooth

用户期待一个UI之间的切换是能够平滑过渡的。UI元素需要做到渐入淡出来取代突然出现与消失。Android从3.0开始有提供[property animation framework](#)用来使得平滑过渡变得更加容易。

如果你不想改变View的属性，只是做一些动画的话，你可以使用ObjectAnimator。

```
mAutoCenterAnimator = ObjectAnimator.ofInt(PieChart.this, "PieRotation", 0);
mAutoCenterAnimator.setIntValues(targetAngle);
mAutoCenterAnimator.setDuration(AUTOCENTER_ANIM_DURATION);
mAutoCenterAnimator.start();
```

如果你想改变的是view的某些基础属性，你可以使用[ViewPropertyAnimator](#)，它能够同时执行多个属性的动画。

```
animate().rotation(targetAngle).setDuration(ANIM_DURATION).start();
```

编写:[kesenhoo](#)

校对:

优化自定义 **View**

前面的课程学习到了如何创建设计良好的View，并且能够使之在手势与状态切换时得到正确的反馈。下面要介绍的是如何使得view能够执行更快。为了避免UI显得卡顿，你必须确保动画能够保持在60fps以上。

Do Less, Less Frequently

为了加速你的view，对于频繁调用的方法，需要尽量减少不必要的代码。先从onDraw开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致GC，从而导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。

你还需要尽可能的减少onDraw被调用的次数，大多数时候导致onDraw都是因为调用了invalidate().因此请尽量减少调用invalide()的次数。如果可能的话，尽量调用含有4个参数的invalidate()方法而不是没有参数的invalidate()。没有参数的invalidate会强制重绘整个view。

另外一个非常耗时的操作是请求layout。任何时候执行requestLayout()，会使得Android UI系统去遍历整个View的层级来计算出每一个view的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持View的层级是扁平化的，这样对提高效率很有帮助。

如果你有一个复杂的UI，你应该考虑写一个自定义的ViewGroup来执行他的layout操作。与内置的view不同，自定义的view可以使得程序仅仅测量这一部分，这避免了遍历整个view的层级结构来计算大小。

Use Hardware Acceleration

从Android 3.0开始，Android的2D图像系统可以通过GPU来加速。GPU硬件加速可以提高许多程序的性能。但是这并不是说它适合所有的程序。

参考[Hardware Acceleration](#) 来学习如何在程序中启用加速。

一旦你开启了硬件加速，性能的提示并不一定可以明显察觉到。移动GPU在某些例如scaling,rotating与translating的操作中表现良好。但是对其他一些任务则表现不佳。

在下面的例子中，绘制pie是相对来说比较费时的。解决方案是把pie放到一个子view中，并设置View使用LAYER_TYPE_HARDWARE来进行加速。

```
private class PieView extends View {

    public PieView(Context context) {
        super(context);
        if (!isInEditMode()) {
            setLayerType(View.LAYER_TYPE_HARDWARE, null);
        }
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        for (Item it : mData) {
            mPiePaint.setShader(it.mShader);
            canvas.drawArc(mBounds,
                360 - it.mEndAngle,
                it.mEndAngle - it.mStartAngle,
                true, mPiePaint);
        }
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        mBounds = new RectF(0, 0, w, h);
    }

    RectF mBounds;
}
```

通过这样的修改以后，PieChart.PieView.onDraw()只会在第一次现实的时候被调用。之后，pie chart会被缓存为一张图片，并通过GPU来进行重画不同的角度。

缓存图片到hardware layer会消耗video memory，而video memory又是有限的。基于这样的考虑，仅仅在用户触发scrolling的时候使用LAYER_TYPE_HARDWARE，在其他时候，使用LAYER_TYPE_NONE。

编写:

校对:

创建向后兼容的UI

编写:

校对:

抽象新的APIs

编写:

校对:

代理至新的**APIs**

编写:

校对:

使用旧的**APIs**实现新**API**的效果

编写:

校对:

使用版本敏感的组件

编写:[K0ST](#)

校对:

实现辅助功能

当我们需要尽可能扩大我们用户的基数的时候，就要开始注意我们软件的可达性了(*Accessibility* 易接近，可亲性)。界面中的小提示对大多数用户而言是可行的，比如说当按钮被按下时视觉上的变化，但是对于那些视力上有些缺陷的用户而言效果就不是那么理想了。

本章将给您演示如何最大化利用Android框架中的Accessibility特性。包括如何利用焦点导航(*focus navigation*)与内容描述(*content description*)对你的应用的可达性进行优化。也包括了了创建Accessibility Service，使用户与应用（不仅仅是你自己的应用）之间的交互更加容易。

课程

[开发Accessibility应用](#)

学习如何让你的程序更易用，具有可达性。允许使用键盘或者十字键(*directional pad*)来进行导航，利用Accessibility Service特性设置标签或执行事件来打造更舒适的用户体验。

[编写Accessibility Services](#)

编写一个Accessibility Service来监听可达性事件，利用这些不同类型的事件和内容描述来帮助用户与应用的交互。本例将会实现利用一个TTS引擎来向用户发出语音提示的功能。

编写:[K0ST](#)

校对:

开发辅助程序

本课程将教您：

1. 添加内容描述(*Content Descriptions*)
2. 设计焦点导航 (*Focus Navigation*)
3. 执行可达性事件(*Accessibility Events*)
4. 测试你的程序

Android平台本身有一些专注可达性的特性，这些特性可以帮助你专门为那些视觉上或生理上有缺陷的人在应用上做特别的优化。然而，正确的优化方式或最简单利用这个特性的方法往往不是那么显而易见的。本课程将给您演示如何利用和实现这些策略和平台的特性，构建一个更好的具有可达性的Android应用。

添加内容描述

一个好的交互界面上的元素通常不需要特别使用一个标签来表明这个元素的作用。例如对于一个任务型应用来说，一个项目旁边的勾选框表达的意思就非常明确，或者对于一个文件管理应用，垃圾桶的图标表达的意思也非常清除。然而对于具有视觉障碍的用户来说，其他类型的UI交互提示是有必要的。

幸运的是，我们可以很轻松的给一个UI元素加上标签，这样类似于[TalkBack](#)这样的基于语音的Accessibility Service就可以将标签的内容朗读出来。如果你的标签在整个应用的声明周期中不太可能会发生变化(比如‘停止’或者‘购买’)，你就可以在XML布局文件中对`android:contentDescription`属性进行修改。例子如下：

```
<Button
    android:id="@+id/pause_button"
    android:src="@drawable/pause"
    android:contentDescription="@string/pause"/>
```

然而，在很多情况下描述的内容是基于上下文环境的，比如说一个开关按钮的状态，或者在list中一片可选的数据项。在运行时编辑内容描述可以使用`setContentDescription()`方法，例子如下：

```
String contentDescription = "Select " + strValues[position];
label.setContentDescription(contentDescription);
```

将这个添加进您的代码是提高您应用可达性的最简单的方法。尝试着将那些有用的地方都加入内容描述，但是要避免像web开发者那样将所有的元素都标注，那样会产生大量的无用信息。比如说，不要将应用图标的内容描述设置为‘应用图标’。这仅仅会对用户的浏览产生干扰。

来试试吧！下载TalkBack(谷歌开发的一款可达性应用)，在**Settings > Accessibility > TalkBack**将它开启。然后使用你的应用听听看TalkBack发出的语音提示。

设计焦点导航

你的应用除了支持触摸操作外，更应该支持其他的导航方式。很多Android设备不仅仅提供了触摸屏，还提供了其他的导航硬件比如说十字键、方向键、轨迹球等等。除此之外，最新的Android发行版本也支持蓝牙或USB的外接设备，比如键盘等等。

为了实现这种方式的导航，一切用户可以用来可导航的元素(*navigational elements*)都需要设置为focusable（聚焦），这个设置也可以在运行时通过`View.setFocusable()`方法来进行设定，或者也可以在XML布局文件中使用`android:focusable`来设置。

每个UI控件有四个属性，`android:nextFocusUp`,`android:nextFocusDown`,`android:nextFocusLeft`,`android:nextFocusRight`,用户在导航时可以利用这些属性来指定下一个焦点的位置。系统会自动根据布局的方向来确定导航的顺序，如果在您的应用中系统提供的方案并不合适，您可以用这些属性来进行修改。

比如说，下面就是一个关于按钮和标签的例子，他们都是可聚焦的(*focusable*)，按向下键会将焦点从按钮移到文字上，按向上会重新将焦点移到按钮上。

```
<Button android:id="@+id/doSomething"
        android:focusable="true"
        android:nextFocusDown="@id/label"
        ... />
<TextView android:id="@+id/label"
        android:focusable="true"
        android:text="@string/labelText"
        android:nextFocusUp="@id/doSomething"
        ... />
```

证实您的应用运行正确的直观方法，最简单的方式就是在Android虚拟机里运行您的应用，然后使用虚拟器的方向键来在各个元素之间导航，使用OK按钮来代替触摸元素的事件。

填充可达性事件

如果你在Android框架中使用了View组件，当你选中了一个View或者是焦点变化的时候，可达性事件(*AccessibilityEvent*)都会产生。这些事件会被传递到Accessibility Service中进行处理，实现一些辅助功能，如语音提示等。

如果你写了一个自定义的View，请确保它在合适的时候产生事件。使用*sendAccessibilityEvent(int)*函数可以产生可达性事件，其中的参数表示事件的类型。完整的可达性事件类型可查阅[AccessibilityEvent](#)参考文档。

比如说，你拓展了一个图片的View，你希望在它聚焦的时候使用键盘打字可以在其中插入题注，这时候发送一个*TYPE_VIEW_TEXT_CHANGED*事件就非常合理，尽管它不是本身就构建在这个图片View中的。产生事件的代码如下：

```
public void onTextChanged(String before, String after) {
    ...
    if (AccessibilityManager.getInstance(mContext).isEnabled()) {
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);
    }
    ...
}
```

测试你的程序

请确保您在添加可达性功能后测试它的有效性。为了测试内容描述可可达性事件，请安装并启用一个Accessibility Service。其中的一个选择就是使用TalkBack，它是一个免费的开源的屏幕读取软件，可在Google Play上进行下载。Service启动后，请测试您应用中所有的功能，同时听听TalkBack的语音反馈。

同时，尝试着用是一个方向控制器来控制你的应用，而非使用触摸的方式。你可以使用一个物理设备，比如十字键、轨迹球等。如果没有条件，可以使用android虚拟机，它提供了虚拟的按键控制。

在测试导航与反馈的同时，和没有任何视觉提示的情况下，你应该对你的应用大概是一个什么样子有所认识。出现问题就修复优化它们，你最终就会开发出一个更易用可达的Android程序。

编写:[K0ST](#)

校对:

开发辅助服务

本课程将教您：

1. 创建Accessibility Service
2. 配置你的Accessibility Service
3. 响应AccessibilityEvents
4. 从View层级中提取更多信息

Accessibility Service是Android系统框架提供给安装在设备上应用的一个可选的导航反馈特性。Accessibility Service 可以替代应用与用户交流反馈，比如将文本转化为语音提示，或是用户的手指悬停在屏幕上一个较重要的区域时的触摸反馈等。本课程将教您如何创建一个Accessibility Service，同时处理来自应用的信息，并将这些信息反馈给用户。

创建Accessibility Service

Accessibility Service可以绑定在一个正常的应用中，或者是单独的一个Android项目都可以。创建一个Accessibility Service的步骤与创建普通Service的步骤相似，在你的项目中创建一个继承于[AccessibilityService](#)的类：

```
package com.example.android.apis.accessibility;

import android.accessibilityservice.AccessibilityService;

public class MyAccessibilityService extends AccessibilityService {
    ...
    @Override
    public void onAccessibilityEvent(AccessibilityEvent event) {
    }

    @Override
    public void onInterrupt() {
    }

    ...
}
```

与其他Service类似，你必须在manifest文件当中声明这个Service。记得标明它监听处理了android.accessibilityservice事件，以便Service在其他应用产生[AccessibilityEvent](#)的时候被调用。

```
<application ...>
...
<service android:name=".MyAccessibilityService">
    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>
    ...
</service>
...
</application>
```

如果你为这个Service创建了一个新项目，且仅仅是一个Service而不准备做成一个应用，那么你就可以移除启动的Activity(一般为MainActivity.java)，同样也记得在manifest中将这个Activity声明移除。

配置你的Accessibility Service

设置Accessibility Service的配置变量会告诉系统如何让Service运行与何时运行。你希望响应哪种类型的事件？Service是否对所有的应用有效还是对部分指定包名的应用有效？使用哪些不同类型的反馈？

你有两种设置这些变量属性的方法，一种向下兼容的办法是通过代码来进行设定，使用setServiceInfo([android.accessibilityservice.AccessibilityServiceInfo](#))。你需要重写(override)onServiceConnected()方法，并在这里进行Service的配置。

```
@Override
public void onServiceConnected() {
    // Set the type of events that this service wants to listen to. Others
    // won't be passed to this service.
    info.eventTypes = AccessibilityEvent.TYPE_VIEW_CLICKED |
        AccessibilityEvent.TYPE_VIEW_FOCUSED;

    // If you only want this service to work with specific applications, set their
    // package names here. Otherwise, when the service is activated, it will listen
    // to events from all applications.
    info.packageNames = new String[]
        {"com.example.android.myFirstApp", "com.example.android.mySecondApp"};

    // Set the type of feedback your service will provide.
    info.feedbackType = AccessibilityServiceInfo.FEEDBACK_SPOKEN;

    // Default services are invoked only if no package-specific ones are present
    // for the type of AccessibilityEvent generated. This service *is*
    // application-specific, so the flag isn't necessary. If this was a
    // general-purpose service, it would be worth considering setting the
    // DEFAULT flag.

    // info.flags = AccessibilityServiceInfo.DEFAULT;

    info.notificationTimeout = 100;

    this.setServiceInfo(info);
}
```

在Android 4.0之后，就用另一种方式来设置了：通过设置XML文件来进行配置。一些特性的选项比如canRetrieveWindowContent仅仅可以在XML可以配置。对于上面所示的相应的配置，利用XML配置如下：

```
<accessibility-service
    android:accessibilityEventTypes="typeViewClicked|typeViewFocused"
    android:packageNames="com.example.android.myFirstApp, com.example.android.mySecondApp"
    android:accessibilityFeedbackType="feedbackSpoken"
    android:notificationTimeout="100"
    android:settingsActivity="com.example.android.apis.accessibility.TestBackActivity"
    android:canRetrieveWindowContent="true"
/>
```

如果你确定是通过XML进行配置，那么请确保在manifest文件中通过< meta-data >标签指定这个配置文件。假设此配置文件存放的地址为：res/xml/serviceconfig.xml，那么标签应该如下：

```
<service android:name=".MyAccessibilityService">
    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>
    <meta-data android:name="android.accessibilityservice"
        android:resource="@xml/serviceconfig" />
</service>
```

响应Accessibility Event

现在你的Service已经配置好并可以监听Accessibility Event了，来写一些响应这些事件的代码吧！首先就是要重写`onAccessibilityEvent(AccessibilityEvent)`方法，在这个方法中，使用`getEventType()`来确定事件的类型，使用`getContentDescription()`来提产生这个事件的View相关的文本标签。

```
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
    final int eventType = event.getEventType();
    String eventText = null;
    switch(eventType) {
        case AccessibilityEvent.TYPE_VIEW_CLICKED:
            eventText = "Focused: ";
            break;
        case AccessibilityEvent.TYPE_VIEW_FOCUSED:
            eventText = "Focused: ";
            break;
    }

    eventText = eventText + event.getContentDescription();

    // Do something nifty with this text, like speak the composed string
    // back to the user.
    speakToUser(eventText);
    ...
}
```

从View层级中提取更多信息

这一步并不是必要步骤，但是却非常有用。Android 4.0版本中增加了一个新特性，就是能够用AccessibilityService来遍历View层级，并从产生Accessibility事件的组件与它的父子组件中提取必要的信息。为了实现这个目的，你需要在XML文件中进行如下的配置：

1. 立即获取到产生这个事件的Parent
2. 在这个Parent中寻找文本标签或勾选框
3. 如果找到，创建一个字符串来反馈给用户，提示内容和是否已勾选。
4. 如果当遍历View的时候某处返回了null值，那么就直接结束这个方法。

```
// Alternative onAccessibilityEvent, that uses AccessibilityNodeInfo

@Override
public void onAccessibilityEvent(AccessibilityEvent event) {

    AccessibilityNodeInfo source = event.getSource();
    if (source == null) {
        return;
    }

    // Grab the parent of the view that fired the event.
    AccessibilityNodeInfo rowNode = getListItemNodeInfo(source);
    if (rowNode == null) {
        return;
    }

    // Using this parent, get references to both child nodes, the label and the checkbox.
    AccessibilityNodeInfo labelNode = rowNode.getChild(0);
    if (labelNode == null) {
        rowNode.recycle();
        return;
    }

    AccessibilityNodeInfo completeNode = rowNode.getChild(1);
    if (completeNode == null) {
        rowNode.recycle();
        return;
    }

    // Determine what the task is and whether or not it's complete, based on
    // the text inside the label, and the state of the check-box.
    if (rowNode.getChildCount() < 2 || !rowNode.getChild(1).isCheckable()) {
        rowNode.recycle();
        return;
    }

    CharSequence taskLabel = labelNode.getText();
    final boolean isComplete = completeNode.isChecked();
    String completeStr = null;

    if (isComplete) {
        completeStr = getString(R.string.checked);
    } else {
        completeStr = getString(R.string.not_checked);
    }
    String reportStr = taskLabel + completeStr;
    speakToUser(reportStr);
}
```

现在你已经实现了一个完整可运行的Accessibility Service。尝试着调整它与用户的交互方式吧！比如添加语音引擎，或者添加震动来提供触觉上的反馈都是不错的选择！

编写:

校对:

编写:

校对:

淡化系统Bar

编写:

校对:

隐藏系统Bar

编写:

校对:

隐藏导航Bar

编写:

校对:

全屏沉浸式应用

编写:

校对:

响应**UI**可见性的变化

编写:

校对:

用户输入

编写:Andrwyw

校对:

使用触摸手势

这一章节描述如何编写允许用户通过触摸手势进行交互的app。Android提供了多种API帮助你创建和检测手势。

尽管你的app不应该依赖于触摸手势来完成基本操作（因为某些情况下手势是不用的），但为你的app添加基于触摸的交互，将会大大地提高app的可用性以及吸引力。

为了给用户提供一致、直觉性的使用体验，你的app应该遵守Android触摸手势的惯常做法。[手势设计指南](#)展示了Android app中的常用手势。同样，设计指南也提供了[触摸反馈](#)的相关内容。

Lessons

- [检测常用的手势](#)

学习如何使用[GestureDetector](#)检测基本的触摸手势,如滑动,惯性滑动以及双击。

- [跟踪手势移动](#)

学习如何跟踪手势移动。

- [Scroll手势动画](#)

学习如何使用scrollers（[Scrollers](#)以及[OverScroll](#)）来产生滚动动画以响应触摸事件。

- [处理多触摸手势](#)

学习如何检测多点(手指)触摸手势。

- [拖拽与缩放](#)

学习如何实现基于触摸的拖拽与缩放。

- [管理ViewGroup中的触摸事件](#)

学习如何管理[ViewGroup](#)中的触摸事件，以确保事件能被正确地分发到目标views。

编写:Andrwyw

校对:

检测常用的手势

“触摸手势”出现在用户用一根或多根手指触碰屏幕时，并且你的应用会把这样的触摸方式解释成一种特定的手势。手势检测有以下相应的两个阶段：

1. 采集触摸事件的相关数据。
2. 分析这些数据，看它们是否是任何一种你的app所支持的手势。

支持库中的类

本节课程的示例程序使用的是[GestureDetectorCompat](#)类和[MotionEventCompat](#)类。这些类都在[Support Library](#)中。你可以通过使用支持库中的类，来为运行着Android 1.6及以上系统的设备提供兼容性功能。需要注意的一点是，[MotionEventCompat](#)类不是[MotionEvent](#)类的替代品。它只是提供了一些静态工具类函数，你可以把[MotionEvent](#)对象作为参数来使用这些函数，从而得到与事件相关的动作。

采集数据

当用户用一根或多根手指触碰屏幕时，接受触摸事件的那个View的 [onTouchEvent\(\)](#) 回调函数就会被触发。对于触摸事件的每个阶段（放置，按压，添加另一个手指等等），[onTouchEvent\(\)](#) 都会被调用数次，并且最终被识别为一种手势。

手势开始于用户刚触摸屏幕时，其后系统会持续地追踪用户手指的位置，当用户手指都离开屏幕时手势结束。在整个交互期间，MotionEvent都会被分发给onTouchEvent()函数，来提供所有交互的详细信息。你的app可以使用MotionEvent提供的数据来判断是否发生了某种特定的手势。

为Activity或View捕获触摸事件

为了捕获Activity或View中的触摸事件，你可以重写onTouchEvent()回调函数。

接下来的代码段使用了getActionMasked()函数，该函数可以从参数event中抽取出用户完成的操作。它会提供一些关于原始的触摸数据，你可以使用这些数据来判断是否发生了某个的特定手势。

```
public class MainActivity extends Activity {
    ...
    // This example shows an Activity, but you would use the same approach if
    // you were subclassing a View.
    @Override
    public boolean onTouchEvent(MotionEvent event){

        int action = MotionEventCompat.getActionMasked(event);

        switch(action) {
            case (MotionEvent.ACTION_DOWN) :
                Log.d(DEBUG_TAG, "Action was DOWN");
                return true;
            case (MotionEvent.ACTION_MOVE) :
                Log.d(DEBUG_TAG, "Action was MOVE");
                return true;
            case (MotionEvent.ACTION_UP) :
                Log.d(DEBUG_TAG, "Action was UP");
                return true;
            case (MotionEvent.ACTION_CANCEL) :
                Log.d(DEBUG_TAG, "Action was CANCEL");
                return true;
            case (MotionEvent.ACTION_OUTSIDE) :
                Log.d(DEBUG_TAG, "Movement occurred outside bounds " +
                    "of current screen element");
                return true;
            default :
                return super.onTouchEvent(event);
        }
    }
}
```

你可以自行处理这些events来判断是否出现了某个手势。当你需要检测自定义手势时，你可以使用这种方式。然而，如果你的app仅仅需要使用一些常见的手势，如双击，长按，惯性滑动等，你可以利用GestureDetector类来完成。GestureDetector可以让你更简单地检测常见手势，并且无需自行处理单个的触摸事件。相关内容将会在下面的Detect Gestures中讨论。

捕获单个view对象的触摸事件

除了使用 onTouchEvent()来捕获触摸事件，你也可以使用setOnTouchListener()函数给View对象关联一个 View.OnTouchListener 对象来捕获触摸事件。这样做可以让你不继承一个已有的view就能监听它的触摸事件。比如：

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {
        // ... Respond to touch events
        return true;
    }
});
```

创建listener对象时，谨防对ACTION_DOWN事件返回false。如果你这样做了，会导致listener对象监听不到后续的ACTION_MOVE、ACTION_UP等系列事件。因为ACTION_DOWN事件是所有触摸事件的开端。

如果你正在写一个自定义View,你也可以像上面描述的那样重写onTouchEvent()函数。

检测手势

Android提供了GestureDetector类来检测一般手势。它支持的手势包括onDown(), onLongPress(), onFling()等。你可以把GestureDetector和上面描述的onTouchEvent()函数结合在一起使用。

检测所有支持的手势

当你实例化一个GestureDetectorCompat对象时，需要一个实现了GestureDetector.OnGestureListener接口的类的对象作为参数。当某个特定的触摸事件发生时，GestureDetector.OnGestureListener会通知用户。为了让你的GestureDetector对象能接收到触摸事件，你需要重写View或Activity的onTouchEvent()函数，并且把所有捕获到的事件传递给detector对象。

接下来的代码段中，on型的函数返回值是true意味着你已经处理完这个触摸事件了。如果返回false，则会把事件沿view栈传递，直到触摸事件被成功地处理了。

运行下面的代码段，来了解你与触摸屏交互时一个动作是如何被触发的，以及每个触摸事件的MotionEvent对象中的内容。你也会了解到一个简单的交互会产生多少的数据。

```
public class MainActivity extends Activity implements
    GestureDetector.OnGestureListener,
    GestureDetector.OnDoubleTapListener{

    private static final String DEBUG_TAG = "Gestures";
    private GestureDetectorCompat mDetector;

    // Called when the activity is first created.
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Instantiate the gesture detector with the
        // application context and an implementation of
        // GestureDetector.OnGestureListener
        mDetector = new GestureDetectorCompat(this,this);
        // Set the gesture detector as the double tap
        // listener.
        mDetector.setOnDoubleTapListener(this);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        // Be sure to call the superclass implementation
        return super.onTouchEvent(event);
    }

    @Override
    public boolean onDown(MotionEvent event) {
        Log.d(DEBUG_TAG,"onDown: " + event.toString());
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
        float velocityX, float velocityY) {
        Log.d(DEBUG_TAG, "onFling: " + event1.toString()+event2.toString());
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onLongPress: " + event.toString());
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX,
        float distanceY) {
        Log.d(DEBUG_TAG, "onScroll: " + e1.toString()+e2.toString());
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        Log.d(DEBUG_TAG, "onShowPress: " + event.toString());
    }
}
```

```

@Override
public boolean onSingleTapUp(MotionEvent event) {
    Log.d(DEBUG_TAG, "onSingleTapUp: " + event.toString());
    return true;
}

@Override
public boolean onDoubleTap(MotionEvent event) {
    Log.d(DEBUG_TAG, "onDoubleTap: " + event.toString());
    return true;
}

@Override
public boolean onDoubleTapEvent(MotionEvent event) {
    Log.d(DEBUG_TAG, "onDoubleTapEvent: " + event.toString());
    return true;
}

@Override
public boolean onSingleTapConfirmed(MotionEvent event) {
    Log.d(DEBUG_TAG, "onSingleTapConfirmed: " + event.toString());
    return true;
}
}

```

检测部分支持的手势

如果你仅仅只想处理几种手势，你可以选择继承GestureDetector.SimpleOnGestureListener类，而不是实现GestureDetector.OnGestureListener接口。

GestureDetector.SimpleOnGestureListener类实现了所有的on型函数，并且都返回false。因此你可以仅仅重写你所需要的函数。比如，下面的代码段中创建了一个继承自GestureDetector.SimpleOnGestureListener的类，并且只重写了onFling()和onDown()函数。

无论你是否使用GestureDetector.OnGestureListener类，最好都实现onDown()函数并且返回true。这是因为所有的手势都是由onDown()消息开始的。如果你让onDown()函数返回false，就像GestureDetector.SimpleOnGestureListener类默认的那样，系统会假定你想忽略手势的剩余部分，GestureDetector.OnGestureListener中的其他函数也就永远不会被调用。这可能让你的app出现意想不到的问题。仅仅当你真的想忽略整个手势时，你才应该让onDown()函数返回false。

```

public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG, "onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent event2,
                               float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " + event1.toString()+event2.toString());
            return true;
        }
    }
}

```

编写:

校对:

跟踪手势移动

编写:

校对:

Scroll手势动画

编写:

校对:

处理多触摸手势

编写:

校对:

拖拽与缩放

编写:

校对:

管理ViewGroup中的触摸事件

编写:

校对:

处理按钮点击

编写:

校对:

指定输入法类型

编写:

校对:

处理输入法可见性

编写:

校对:

编写:

校对:

处理输入命令

编写:

校对:

兼容游戏控制器

编写:

校对:

处理控制器输入动作

编写:

校对:

支持不同的**Android**系统版本

编写:

校对:

支持多个控制器

编写:

校对:

后台任务

编写:[kesenhoo](#)

校对:

在IntentService中执行后台任务

除非你特别指定，否则大部分在前台UI界面上的操作都执行在一个叫做UI Thread的特殊线程中。这可能会导致某些问题，因为耗时操作可能会干扰界面的响应性能。为了避免这样的问题，Android Framework提供了几个类，用来帮助你把那些耗时操作移动到后台线程中执行。那些类中最常用的就是[IntentService](#)。

这一章节会讲到如何实现一个IntentService，向它发送任务并反馈它的结果给其他模块。

Lessons

- [Creating a Background Service:创建IntentService](#)

学习如何创建一个IntentService。

- [Sending Work Requests to the Background Service:发送任务请求到IntentService](#)

学习如何发送工作任务到IntentService。

- [Reporting Work Status:报告后台任务的执行状态](#)

学习如何使用Intent与LocalBroadcastManager在Activit与IntentService之间进行交互。

编写:[kesenhoo](#)

校对:

Creating a Background Service: 创建IntentService

IntentService为执行一个操作在单个后台线程，提供了一种直接的实现方式。它可以处理一个长时间操作的任务并确保不影响到UI的响应性。而且IntentService的执行并不受UI的生命周期的影响。

IntentService有下面几个局限性：

- 不可以直接和UI做交互。为了把他执行的结果体现在UI上，需要发送给Activity。
- 工作任务队列是顺序执行的，如果一个任务正在IntentService中执行，此时你再发送一个任务请求，这个任务会一直等待直到前面一个任务执行完毕。
- 正在执行的任务无法打断。

然而，在大多数情况下，IntentService都是简单后台任务操作的理想选择。

这节课会演示如何创建继承的IntentService。同样也会演示如何创建必须实现的回调[onHandleIntent\(\)](#)。最后，还会解释如何在manifest文件中定义这个IntentService。

1)创建IntentService

为了给你的app创建一个IntentService，定义一个类，extends IntentService，在里面override onHandleIntent()方法，如下所示：

```
public class RSSPullService extends IntentService {
    @Override
    protected void onHandleIntent(Intent workIntent) {
        // Gets data from the incoming Intent
        String dataString = workIntent.getDataString();
        ...
        // Do work here, based on the contents of dataString
        ...
    }
}
```

注意一个普通Service组件的其他回调，例如onStartCommand()会被IntentService自动触发。在IntentService中，要避免override那些回调。

2)在Manifest文件中定义IntentService

IntentService需要在manifest文件的标签下进行定义，如下所示：

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name">
    ...
    <!--
        Because android:exported is set to "false",
        the service is only available to this app.
    -->
    <service
        android:name=".RSSPullService"
        android:exported="false"/>
    ...
</application>
```

android:name属性指明了IntentService的名字。

注意标签并没有包含任何intent filter。因为发送任务给IntentService的Activity需要使用显式Intent，所以不需要filter。这也意味着只有在同一个app或者其他使用同一个UserID的组件才能够访问到这个Service。

至此，已经学习了IntentService的基础知识，下节会学习如何发送任务到IntentService。

编写:[kesenhoo](#)

校对:

Sending Work Requests to the Background Service: 发送任务请求到IntentService

前一篇文章演示了如何创建一个IntentService类。这次会演示如何通过发送一个Intent来触发IntentService执行任务。这个Intent可以传递一些数据给IntentService。可以在Activity或者Fragment的任何时间点发送这个Intent。

为了创建一个工作请求并发送到IntentService。需要先创建一个explicit Intent，添加数据到intent，然后通过执行[startService\(\)](#) 把它发送到IntentService。

下面的是代码示例：

- 创建一个新的显式的Intent用来启动IntentService。

```
/*
 * Creates a new Intent to start the RSSPullService
 * IntentService. Passes a URI in the
 * Intent's "data" field.
 */
mServiceIntent = new Intent(getActivity(), RSSPullService.class);
mServiceIntent.setData(Uri.parse(dataUrl));
```

- 执行startService()

```
// Starts the IntentService
getActivity().startService(mServiceIntent);
```

注意：可以在Activity或者Fragment的任何位置发送任务请求。

一旦执行了startService()，IntentService在自己本身的[onHandleIntent\(\)](#)方法里面开始执行这个任务。

下一步是如何把工作任务的执行结果返回给发送任务的Activity或者Fragment。下节课会演示如何使用[BroadcastReceiver](#)来完成这个任务。

编写:[kesenhoo](#)

校对:

Reporting Work Status: 报告后台任务的执行状态

这章节会演示如何回传IntentService中执行的任务状态与结果给发送方。例如，回传任务的状态给Activity并进行更新UI。推荐的方式是使用[LocalBroadcastManager](#)，这个组件可以限制broadcast只在自己的App中进行传递。

Report Status From an IntentService

为了在IntentService中向其他组件发送任务状态，首先创建一个Intent并在data字段中包含需要传递的信息。作为一个可选项，还可以给这个Intent添加一个action与data URI。

下一步，通过执行[LocalBroadcastManager.sendBroadcast\(\)](#)来发送Intent。Intent被发送到任何有注册接受它的组件中。为了获取到LocalBroadcastManager的实例，可以执行getInstance()。代码示例如下：

```
public final class Constants {
    ...
    // Defines a custom Intent action
    public static final String BROADCAST_ACTION =
        "com.example.android.threadsample.BROADCAST";
    ...
    // Defines the key for the status "extra" in an Intent
    public static final String EXTENDED_DATA_STATUS =
        "com.example.android.threadsample.STATUS";
    ...
}
public class RSSPullService extends IntentService {
    ...
    /*
     * Creates a new Intent containing a Uri object
     * BROADCAST_ACTION is a custom Intent action
     */
    Intent localIntent =
        new Intent(Constants.BROADCAST_ACTION)
        // Puts the status into the Intent
        .putExtra(Constants.EXTENDED_DATA_STATUS, status);
    // Broadcasts the Intent to receivers in this app.
    LocalBroadcastManager.getInstance(this).sendBroadcast(localIntent);
    ...
}
```

下一步是在发送任务的组件中接收发送出来的broadcast数据。

Receive Status Broadcasts from an IntentService

为了接受广播的数据对象，需要使用BroadcastReceiver的子类并实现[BroadcastReceiver.onReceive\(\)](#)的方法，这里可以接收LocalBroadcastManager发出的广播数据。

```
// Broadcast receiver for receiving status updates from the IntentService
private class ResponseReceiver extends BroadcastReceiver
{
    // Prevents instantiation
    private DownloadStateReceiver() {
    }
    // Called when the BroadcastReceiver gets an Intent it's registered to receive
    @
    public void onReceive(Context context, Intent intent) {
    ...
        /*
         * Handle Intents here.
         */
    ...
    }
}
```

一旦定义了BroadcastReceiver，也应该定义actions，categories与data用来做广播过滤。为了实现这些，需要使用[IntentFilter](#)，如下所示：

```
// Class that displays photos
public class DisplayActivity extends FragmentActivity {
    ...
    public void onCreate(Bundle stateBundle) {
        ...
        super.onCreate(stateBundle);
        ...
        // The filter's action is BROADCAST_ACTION
        IntentFilter mStatusIntentFilter = new IntentFilter(
            Constants.BROADCAST_ACTION);

        // Adds a data filter for the HTTP scheme
        mStatusIntentFilter.addDataScheme("http");
    }
}
```

为了给系统注册这个BroadcastReceiver，需要通过LocalBroadcastManager执行registerReceiver()的方法。如下所示：

```
// Instantiates a new DownloadStateReceiver
DownloadStateReceiver mDownloadStateReceiver =
    new DownloadStateReceiver();
// Registers the DownloadStateReceiver and its intent filters
LocalBroadcastManager.getInstance(this).registerReceiver(
    mDownloadStateReceiver,
    mStatusIntentFilter);
...
```

一个BroadcastReceiver可以处理多种类型的广播数据。每个广播数据都有自己的ACTION。这个功能使得不用定义多个不同的BroadcastReceiver来分别处理不同的ACTION数据。为BroadcastReceiver定义另外一个IntentFilter，只需要创建一个新的IntentFilter并重复执行registerReceiver()即可。例如：

```
/*
    * Instantiates a new action filter.
    * No data filter is needed.
    */
statusIntentFilter = new IntentFilter(Constants.ACTION_ZOOM_IMAGE);
...
// Registers the receiver with the new filter
LocalBroadcastManager.getInstance(getActivity()).registerReceiver(
    mDownloadStateReceiver,
    mIntentFilter);
```

发送一个广播并不会start或者resume一个Activity。BroadcastReceiver可以接收广播数据，即使是你的app是在后台运行中。但是这不会强迫使得你的app变成foreground的。如果想在app不可见的时候通知用户一个后台的事件，建议使用[Notification](#)。永远不要为了响应一个广播而去启动Activity。

笔者评论:使用LocalBroadcastManager结合IntentService其实是一种很典型高效的办法，同时也更符合OO的思想，通过广播注册与反注册的方式，对两个组件进行解耦。如果使用Handler传递到后台线程作为回调，容易带来的内存泄漏。原因是：匿名内

部类对外面的Activity持有引用，如果在Activity被销毁的时候，没有对Handler进行显式的解绑，会导致Activity无法正常销毁，这样自然就有了内存泄漏。当然，如果用文章中的方案，通常也要记得在Activity的onPause的时候进行unRegisterReceiver，除非你有充足的理由为解释这里为何要继续保留。

编写:[kesenhoo](#)

校对:

使用CursorLoader在后台加载数据

从[ContentProvider](#)查询你需要显示的数据是比较耗时的。如果你在Activity中直接执行查询的操作，那么有可能导致Activity出现ANR的错误。即使没有发生ANR，用户也会看到一个令人烦恼的UI延迟。为了避免那些问题，你应该在另外一个线程中执行查询的操作，等待查询操作完成，然后再显示查询结果。

你可以通过使用[CursorLoader](#)来实现，它会在后台异步查询数据并在查询结束时和Activity重新进行连接。CursorLoader不仅仅能够实现在后台查询数据，还能够在查询数据发生变化时自动执行重新查询的操作。

这节课会介绍如何使用CursorLoader来执行一个后台查询数据的操作。在这节课中的演示代码使用的是[v4 Support Library](#)中的类。

下载演示代码

[ThreadSample](#)

Lessons

- [使用CursorLoader执行查询任务：Running a Query with a CursorLoader](#)

学习如何使用CursorLoader在后台执行查询操作。

- [处理查询的结果：Handling the Results](#)

学习如何处理从CursorLoader查询到的数据，以及在loader框架重置CursorLoader时如何解除当前Cursor的引用。

编写:[kesenhoo](#)

校对:

使用**CursorLoader**执行查询任务

CursorLoader依靠ContentProvider在后台执行一个异步的查询操作，并且返回数据给调用它的Activity或者FragmentActivity。这使得Activity 或者 FragmentActivity 能够在查询任务正在执行的时候可以与用户继续其他的交互。

定义使用CursorLoader的Activity

为了在Activity或者FragmentActivity中使用CursorLoader，需要实现[LoaderCallbacks](#)接口。CursorLoader会触发这些回调方法；这节课与下节课会详细介绍每一个回调方法。

例如，下面演示了FragmentActivity如何使用CursorLoader。

```
public class PhotoThumbnailFragment extends FragmentActivity implements
    LoaderManager.LoaderCallbacks<Cursor> {
    ...
}
```

初始化查询

为了初始化查询，需要执行[LoaderManager.initLoader\(\)](#)。这个方法可以初始化后台任务。你可以在用户输入查询条件之后触发初始化的操作，如果你不需要用户输入数据作为查询条件，你可以触发这个方法在[onCreate\(\)](#)或者[onCreateView\(\)](#)。例如：

```
// Identifies a particular Loader being used in this component
private static final int URL_LOADER = 0;
...
/* When the system is ready for the Fragment to appear, this displays
 * the Fragment's View
 */
public View onCreateView(
    LayoutInflater inflater,
    ViewGroup viewGroup,
    Bundle bundle) {
    ...
    /*
     * Initializes the CursorLoader. The URL_LOADER value is eventually passed
     * to onCreateLoader().
     */
    getLoaderManager().initLoader(URL_LOADER, null, this);
    ...
}
```

Note: `getLoaderManager()` 仅仅是在Fragment类中可以直接访问。为了在FragmentActivity中获取到LoaderManager，需要执行[getSupportLoaderManager\(\)](#)。

开始查询

一旦后台任务被初始化好，它会执行你实现的回调方法[onCreateLoader\(\)](#)。为了启动查询任务，会在这个方法里面返回CursorLoader。你可以初始化一个空的CursorLoader然后使用它的方法来定义你的查询条件，或者你可以在初始化CursorLoader对象的时候就同时定义好查询条件：

```
/*
 * Callback that's invoked when the system has initialized the Loader and
 * is ready to start the query. This usually happens when initLoader() is
 * called. The loaderID argument contains the ID value passed to the
 * initLoader() call.
 */
@Override
public Loader<Cursor> onCreateLoader(int loaderID, Bundle bundle)
{
    /*
     * Takes action based on the ID of the Loader that's being created
     */
    switch (loaderID) {
        case URL_LOADER:
            // Returns a new CursorLoader
            return new CursorLoader(
                getActivity(),    // Parent activity context
                mDataUrl,         // Table to query
                mProjection,      // Projection to return
                null,             // No selection clause
                null,             // No selection arguments
                null               // Default sort order
            );
        default:
            // An invalid id was passed in
            return null;
    }
}
```

一旦后台查询任务获取到了这个Loader对象，就开始在后台执行查询的任务。当查询完成之后，会执行[onLoadFinished\(\)](#)这个回调函数，关于这些内容会在下一节讲到。

编写:[kesenhoo](#)

校对:

处理查询的结果

正如前面一节课讲到的，你应该在 [onCreateLoader\(\)](#) 的回调里面使用CursorLoader执行加载数据的操作。接下去Loader会提供查询数据的结果给Activity或者FragmentActivity实现的[LoaderCallbacks.onLoadFinished\(\)](#)回调方法。这个回调方法的参数之一是[Cursor](#)，它包含了查询的数据。你可以使用Cursor对象来更新需要显示的数据或者进行下一步的处理。

除了[onCreateLoader\(\)](#)与[onLoadFinished\(\)](#)，你也需要实现[onLoaderReset\(\)](#)。这个方法在CursorLoader检测到[Cursor](#)上的数据发生变化时会被触发。当数据发生变化时，系统会触发重新查询的操作。

Handle Query Results

为了显示CursorLoader返回的Cursor数据，需要使用实现AdapterView的类，并为这个类绑定一个实现了CursorAdapter的Adapter。系统会自动把Cursor中的数据显式到View上。

你可以在显示数据之前建立View与Adapter的关联。然后在[onLoadFinished\(\)](#)的时候把Cursor与Adapter进行绑定。一旦你把Cursor与Adapter进行绑定之后，系统会自动更新View。当Cursor上的内容发生改变的时候，也会触发这些操作。

例如:

```
public String[] mFromColumns = {
    DataProviderContract.IMAGE_PICTURENAME_COLUMN
};
public int[] mToFields = {
    R.id.PictureName
};
// Gets a handle to a List View
ListView mListView = (ListView) findViewById(R.id.dataList);
/*
 * Defines a SimpleCursorAdapter for the ListView
 */
SimpleCursorAdapter mAdapter =
    new SimpleCursorAdapter(
        this,           // Current context
        R.layout.list_item, // Layout for a single row
        null,           // No Cursor yet
        mFromColumns,   // Cursor columns to use
        mToFields,      // Layout fields to use
        0               // No flags
    );
// Sets the adapter for the view
mListView.setAdapter(mAdapter);
...
/*
 * Defines the callback that CursorLoader calls
 * when it's finished its query
 */
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    ...
    /*
     * Moves the query results into the adapter, causing the
     * ListView fronting this adapter to re-display
     */
    mAdapter.changeCursor(cursor);
}
```

Delete Old Cursor References

当Cursor失效的时候，CursorLoader会被重置。这通常发生在Cursor相关的数据改变的时候。在重新执行查询操作之前，系统会执行你的[onLoaderReset\(\)](#)回调方法。在这个回调方法中，你应该删除当前Cursor上的所有数据，避免发生内存泄露。一旦onLoaderReset()执行结束，CursorLoader就会重新执行查询操作。

例如:

```
/*
 * Invoked when the CursorLoader is being reset. For example, this is
 * called if the data in the provider changes and the Cursor becomes stale.
 */
@Override
public void onLoaderReset(Loader<Cursor> loader) {

    /*
     * Clears out the adapter's reference to the Cursor.
     * This prevents memory leaks.
     */
    mAdapter.changeCursor(null);
}
```

编写:[lttowq](#)(未验证)

校对:

管理设备的唤醒状态

当你的一个Android设备闲置时，屏幕将会变暗，然后关闭屏幕，最后关闭CPU。这是减少你的设备电量的快速消耗，然而，有些时候，你的应用程序可能需要不同的行为时间：

- 1.例如游戏或电影应用需要保持屏幕亮着
- 2.其他的应用也许不需要屏幕开着，但或许会请求CPU保持运行直到一个关键操作结束。

这节课描述如何在必要的时候保持设备唤醒但消耗它的电量。

Lesson

[Keeping the Device Awake](#)

- 学习当需要时如何保持屏幕和CPU唤醒，同时减少对电池的生命周期的影响。

[Scheduling Repeating Alarms](#)

- 学习使用重复闹钟对于发生在生命周期外之应用的作业调度，即使应用没有运行或者设备处于睡眠状态。

编写:[littowq](#)(未验证)

校对:

保持设备唤醒

为了避免电源消耗，一个Android设备闲置时会迅速进入睡眠状态。然而这里有些时间当一个应用需要唤醒屏幕或者CPU并且保持唤醒完成一些任务。

你采取的方法依赖于你的应用的需要。但是，一般规则是你应该使用最轻量级的方法对的应用，减小你的应用对系统资源的影响。接下来的部分描述怎样处理当设备默认睡眠的行为与你请求不相容的情况。

保持屏幕亮着

确定你的应用需要保持屏幕变亮，比如游戏与电影的应用。最好的方式是使用[FLAG_KEEP_SCREEN_ON](#)在你的Activity（仅在Activity不在Service或其他组件里）例如：

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    }
}
```

这个方法的优点像醒锁([在 Keep the CPU On讨论](#))，它不需要特殊的权限，平台正确管理你的用户在应用之间移动，不需要你的应用担心释放未使用资源。

另外一种实现在你的应用布局xml文件里，通过使用[android:keepScreenOn](#)属性:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:keepScreenOn="true">
    ...
</RelativeLayout>
```

使用android:keepScreenOn="true"与使用[FLAG_KEEP_SCREEN_ON](#)等效。你能无论使用哪个方法对你的应用都不错。编程方式设置该标志在你Activity的优点，它让你的编程后清除标志，从而使屏幕关闭该选项。

注意：你不需要清除[FLAG_KEEP_SCREEN_ON](#)便签除非你不在想屏幕呆在你正在运行的应用里面（例如：如果你想要屏幕延时的一个确定的周期静止）。窗口管理照顾确保正确事情发生当你的应用进入后台或者返回前台。但是如果你明确清除从而允许屏幕再次关闭，使用[clearFlags\(\)](#); getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON).

保持CPU运行

如果你需要保持CPU运行为了完成一些工作在设备睡眠，你可以使用[PowerManager](#)系统服务特性回调唤醒锁。唤醒锁允许你应用控制本地设备电源状态。

创建和支持唤醒锁能有个引人注目的影响对本地设备电源周期。因此你使用唤醒锁仅仅当你确实需要，并保持他们尽可能短的时间尽可能。例如，你不应该需要使用唤醒锁在一个Activity中。以上描述，如果你想保持屏幕亮着在你的Activity，使用[FLAG_KEEP_SCREEN_ON](#)。

一种合理情况对于使用唤醒锁可能在后台服务需要抢占一个唤醒锁保持CPU运行当屏幕关闭时。再一次，可是，这个实践应该最小因为它影响电池周期。

为了使用唤醒锁，首先你增加[WAKE_LOCK](#)权限在应用主要清单文件：

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

如果你的应用包括一个广播接收器使用这个服务做一些工作，你能管理你唤醒锁通过一个[WakefulBroadcastReceiver](#)，作为描述[Using a WakefulBroadcastReceiver](#)，这是优先的方法。如果你的应用不允许这个模式，这里告诉你之间设置唤醒锁：

```
PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
Wakelock wakelock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK),
    "MyWakelockTag");
wakelock.acquire();
```

为了释放唤醒锁，使用[wakelock.release\(\)](#)。这释放你的要求的CPU，它是重要的 对于释放一个唤醒锁当你的应用使用完毕，避免消耗电量。

使用WakefulBroadcastReceiver

使用一个广播接收器结合一个服务让你管理循环周期在后台任务。[WakefulBroadcastReceiver](#)是一个特殊广播接收器类型小心创建和管理一个[PARTIAL_WAKE_LOCK](#)对于你的应用程序。[WakefulBroadcastReceiver](#)忽略任务对于一个[Service](#)（典型的一个[IntentService](#)），当确保设备不转换到睡眠状态。如果你不支持一个唤醒锁当转换工作对于一个服务，你实际上允许设备返回睡眠状态在工作完成之前。网络结果是应用可能没有完成正在做的工作直到一些任意点在未来，不是你想要的。

首先你增加[WakefulBroadcastReceiver](#)在你的主manifest文件里面，作为其他广播接收器。

```
<receiver android:name=".MyWakefulReceiver"></receiver>
```

代码开始在MyIntentService的方法[startWakefulService\(\)](#)。这个方法是完成[startService\(\)](#)。除了[WakefulBroadcastReceiver](#)支持唤醒锁当服务开启。通信停止与[startWakefulService\(\)](#)支持一个额外验证唤醒锁。

```
public class MyWakefulReceiver extends WakefulBroadcastReceiver{
    @Override
    public void onReceive(Context context, Intent intent){
        Intent service = new Intent(context, MyIntentService.class)
        startWakefulService(context, service);
    }
}
```

当服务完成，回调[MyWakefulReceiver.completeWakefulIntent\(\)](#)释放唤醒锁。[completeWakefulIntent\(\)](#)方法有它的相同参数停止从[WakefulBroadcastReceiver](#);

```
public class MyWakefulReceiver extends IntentService{
    public static final int NOTIFICATION_ID = 1;
    private NotificationManager mNotificationManager;
    NotificationCompat.Builder builder;
    public MyIntentService(){
        Bundle extra = intent.getExtras();
        MyWakefulReceiver.completeWakefulIntent(intent);
    }
}
```

[下一课：调度重复闹钟](#)

编写:[littowq](#)(未验证)

校对:

调度重复的闹钟

闹钟([基本类AlarmManager](#))给你一种方式执行基本时间操作你app生命周期外。例如你可以使用闹钟初始化一个长时间操作，例如开启一个服务一天为了下载天气预报。

闹钟的特性

- 它们让设置次数或间距点燃你意图
- 你可以使用它结合广播接收器开启服务和执行其他操作
- 它们执行在你的应用之外，所以你可以使用它触发事件或动作即使你的app没有运行或设备处于睡眠状态。
- 它们可以帮助你app最小化资源请求。你可以调度无需依赖的或者连续运行在后台的服务。

注意

对于定时操作保证结果在你app生命周期之内，替代可虑使用[Handler](#)类结合[定时器](#)与[线程](#)。这个方法给你Android更好的控制系统资源。

理解交替使用

一个重复闹钟是相对简单的机制和有限的灵活性。它或许不是最好的选择对于你的app，特别是如果你的app需要触发网络操作。一个坏的闹钟设计能造成电池漏电和让一个有意义的服务负载。

一个普通的情景对于触发一个你的app同步数据和一个服务器的生命周期外的操作。这个案例你可能冒险使用一个重复的闹钟。但是你自己服务器是本地你的app数据，使用[Google Cloud Messaging\(GCM\)](#)在结合你的[sync adapter](#)是更好解决方案比[AlarmManager](#)。一个同步适配器给你所有相同的调度选项作为[AlarmManager](#)，但是它提供你更灵活性。比如：一个同步可能基本在“新数据”消息从服务器或设备（细节见[Running a Sync Adapter](#)）。用户活动或静止，一天的时间或更久。看下面两个链接对于什么时候怎样使用GCM和同步适配器细节的讨论。1.[The App Clinic:Cricket](#)(需出墙)

2.[DevBytes:Efficient Data Transfers](#)（需出墙）

最好的练习

每个选择让你设计你的重复闹钟可以用序列在你的app使用或滥用系统资源。例如，想象一个流行的app和一个服务器同步。如果你同步操作在计时器的操作上，每一个app的实例同步在11：00P.M，服务器负载造成高延时或者甚至“服务器拒绝”。下面是使用闹钟的建议：

- 增加随意（颤动）对任何网络请求触发作为一个重复闹钟的结果；
 - 做本地任务当一个闹钟触发。“本地任务”意味任何事情不需要敲击服务器或请求一个数据从服务器
 - 在相同的时间，调度闹钟包含网络请求点燃相同定时周期
- 保持你的闹钟频率最小；
- 不是必要的情况不要唤醒设备(这个行为被闹钟决定，细节在[Choose an alarm type](#))；
- 不要使用你的闹钟触发时间比它精确；使用[setInexactRepeating](#)替代[setRepeating](#)。当你使用[setInexactRepeating](#)，Android同步重复闹钟从多个app和在相同的时间点燃它。这减少系统必须唤醒设备的数目，以此减少电源能耗。Android4.4（API Level19），所有的重复闹钟是不精确的。注意当[setInexactRepeating](#)是一个改进[setRepeating](#)，它能覆盖一个服务如果每个app的实例撞击服务器在相同的时间。因此对于网络请求，增加相同随意的闹钟、作为以上描述。
- 避免在你的闹钟基础上计时如果可能 重复的闹钟在预测触发器的没有扫描好的基础上。使用[ELAPSED REALTIME](#)如果你能。不同的闹钟类型描述的细节在下面选项。

设置重复闹钟

下面的描述，重制闹钟作为一个好的选择有规律调度时间或数据备份。一个重复闹钟好如下特性：

- 一个闹钟类型的讨论见[Choose an alarm type](#).
- 一个触发时间。如果触发时间你制定在过去，闹钟触发立即执行。
- 闹钟间距。例如，某一天、每小时、每5秒等等。
- 一个行将发生的意图是当闹钟被触发。当你设置一秒闹钟使用相同悬而未决的意图，它能替换原始闹钟。

选择一个闹钟的类型

其中第一个考虑是使用什么类型重置闹钟。这里有两个通用的计时器闹钟：“elapsed real time”和“real time clock”。elapsed real time使用“计时自从系统引导”作为引用，和real time clock使用UTC计时。这意味着elapsed real time适合设置一个闹钟在一段时间基础上(例如：一个闹钟点燃每30秒)且它不受地区和时区的影响。real time clock类型更好适配闹钟依赖当前时区。两个类型有“唤醒”版本，都能唤醒设备的CPU如果屏幕关闭。这确保闹钟将启动在调度时间、这是有用的如果你的app有一个时间依赖。例如，如果它有一个限制窗口将启动当你的设备在下个唤醒。如果你简单的需要那种启动特殊意图（例如：每半小时），使用其中elapsed real time类型。一般，这是更好的选择。如果你需要闹钟启动在一天中特殊的时间，然后选择某个计real time clock类型。注意,但是这个方法有些缺点——app或许不会翻译好对于其他地区，如果用户改变设备时间设置，它可能造成意外行为在你app。使用一个真实时间计时的闹钟类型也不会扫描好，综上，我们建议你使用elapsed real time。

这里列出类型：

- [ELAPSED_REALTIME](#)-点燃悬而未决意图在计时基础上从设备被引导，但是不需要唤醒设备。The elapsed 时间包括一些次数在设备处于睡眠期间。
- [ELAPSED_REALTIME_WAKEUP](#)唤醒设备并且启动悬而未决意图后指定过去的时间长度自从设备启动。
- [RTC](#)点燃悬而未决的意图在指定时间但是没有唤醒设备。
- [RTC_WAKEUP](#)唤醒设备在悬而未决意图在指定时间。

ELAPSED_REALTIME_WAKEUP例子

这里有一些相同案例使用[ELAPSED_REALTIME_WAKEUP](#)

唤醒设备启动闹钟在30分钟内和每30分钟后：

```
// Hopefully your alarm will have a lower frequency than this!
alarmMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
    AlarmManager.INTERVAL_HALF_HOUR,
    AlarmManager.INTERVAL_HALF_HOUR, alarmIntent);
```

唤醒设备在启动一次（无重复）闹钟在每分钟内：

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
    SystemClock.elapsedRealtime() +
    60 * 1000, alarmIntent);
```

RTC案例

这里有相同案例使用[RTC_WAKEUP](#)

唤醒设备启动闹钟在约定时间2:00PM，一天内重复一次:

```
// Set the alarm to start at approximately 2:00 p.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 14);
// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
    AlarmManager.INTERVAL_DAY, alarmIntent);
```

唤醒设备启动闹钟在8:30am,每20分钟后在启动:

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
```



```
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
// Set the alarm to start at 8:30 a.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 8);
calendar.set(Calendar.MINUTE, 30);
// setRepeating() lets you specify a precise custom interval--in this case,
// 20 minutes.
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
    1000 * 60 * 20, alarmIntent);
```

闹钟启动时间的设定

上面的描述，选择闹钟类型对于创建闹钟是第一步。第二不是设定闹钟启动的时间。对于大多数的app，[setInexactRepeating](#)是正确的选择。当你使用这个方法，Android同步多个不精确的重复闹钟和启动它们在相同的时间。这减少电源的能耗。

对真实的app有严格的要求-例如，闹钟需要精确启动在8:30am和每隔一小时之后-使用[setRepeating](#)，但是你应该避免使用精确地闹钟如果可能。

伴随[setInexactRepeating](#)，你不能指定客户意图一种方式你能[setRepeating](#)。你使用间距常量，例如[INTERVAL_FIFTEEN_MINUTES](#)、[INTERVAL_DAY](#)等等。见[AlarmManager](#)里面的完整的列表。

取消闹钟

依赖你app，你可能想报考一些取消闹钟的能力。取消闹钟回调[cancel](#)在你闹钟管理器，通过[PendingIntent](#)你不在启动，例如：

```
// If the alarm has been set, cancel it.
if (alarmMgr != null) {
    alarmMgr.cancel(alarmIntent);
}
```

启动闹钟当你设备启动时

默认的设置是所有的闹钟被取消当一个设备关闭时。为了阻止发生，你可以你的app自动重启一个重复闹钟如果用户重启设备。这确保在[AlarmManager](#)将继续不需要用户手动启动闹钟。

这里的步骤：

1. 设置[RECEIVE_BOOT_COMPLETED](#)权限在你app主菜单（manifest）允许你的app接受[ACTION_BOOT_COMPLETED](#)在广播后系统完成启动（如果你app已经运行通过用户至少一次才有效）

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

2. 实现一个[BroadcastReceiver](#)接收广播；

```
public class SampleBootReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("android.intent.action.BOOT_COMPLETED")) {
            // Set the alarm here.
        }
    }
}
```

3. 增加接收器在你app的manifest文件里面与一个意图过滤，过滤器在[ACTION_BOOT_COMPLETED](#):

```
<receiver android:name=".SampleBootReceiver"
    android:enabled="false">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/></action>
    </intent-filter>
</receiver>
```

注意在manifest文件里，引导接收器设置android:enabled="false"。这意味着接收器将不会回调除非app程序显式地启用它。这是阻止不必要引导接收器的回调。你能启动一个接收器（例如：如果用户设置如下一个闹钟）如下：

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class);
PackageManager pm = context.getPackageManager();
pm.setComponentEnabledSetting(receiver,
```

```
PackageManager.COMPONENT_ENABLED_STATE_ENABLED,  
PackageManager.DONT_KILL_APP);
```

一旦你启动接收器，它将保持启动，即使用户重启设备。总之，编程式启动接收器重写manifest里的设置，几多次重启。接收器将保持启动直到你的app不在使用它。你可以禁用一个接收器（例如：如果用户取消一个闹钟）如下：

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class);  
PackageManager pm = context.getPackageManager();  
pm.setComponentEnabledSetting(receiver,  
    PackageManager.COMPONENT_ENABLED_STATE_DISABLED,  
    PackageManager.DONT_KILL_APP);
```

[下一课：最佳性能实践](#)



编写:

校对:

性能优化

编写:[kesenhoo](#)

校对:

管理应用的内存

Random Access Memory(RAM)在任何软件开发环境中都是一个很宝贵的资源。这一点在物理内存通常很有限的移动操作系统上，显得尤为突出。尽管Android的Dalvik虚拟机扮演了常规的垃圾回收的角色，但这并不意味着你可以忽视app的内存分配与释放的时机与地点。

为了GC能够从你的app中及时回收内存，你需要避免Memory Leaks(这通常由引用的不能释放而导致)并且在适当的时机(下面会讲到的lifecycle callbacks)来释放引用。对于大多数apps来说，Dalvik的GC会自动把离开活动线程的对象进行回收。

这篇文章会解释Android如何管理app的进程与内存分配，并且你可以在开发Android应用的时候主动的减少内存的使用。关于Java的资源管理机制，请参加其它书籍或者线上材料。如果你正在寻找如何分析你的内存使用情况的文章，请参考这里[Investigating Your RAM Usage](#)。

第1部分:Android是如何管理内存的

Android并没有提供内存的交换区(Swap space),但是它使用

1)共享内存

Android通过下面几个方式在不同的Process中来共享RAM:

- 每一个app的process都是从同一个被叫做Zygote的进程中fork出来的。Zygote进程在系统启动并且载入通用的framework的代码与资源之后开始启动。为了启动一个新的程序进程,系统会fork Zygote进程生成一个新的process,然后新的process中加载并运行app的代码。这使得大多数的RAM pages被用来分配给framework的代码与资源,并在应用的所有进程中进行共享。
- 大多数static的数据被mmapped到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享,而且使得它能够在需要的时候被paged out。例如下面几种static的数据:
 - Dalvik code (by placing it in a pre-linked .odex file for direct mmappping)
 - App resources (by designing the resource table to be a structure that can be mmapped and by aligning the zip entries of the APK)
 - Traditional project elements like native code in .so files.
- 在许多地方,Android通过显式的分配共享内存区域(例如ashmem或者gralloc)来实现一些动态RAM区域的能够在不同进程间的共享。例如,window surfaces在app与screen compositor之间使用共享的内存,cursor buffers在content provider与client之间使用共享的内存。

关于如何查看app所使用的共享内存,请查看[Investigating Your RAM Usage](#)

2)分配与回收内存

这里有下面几点关于Android如何分配与回收内存的事实:

- 每一个进程的Dalvik heap都有一个限制的虚拟内存范围。这就是逻辑上讲的heap size,它可以随着需要进行增长,但是会有一个系统为它所定义的上限。
- 逻辑上讲的heap size和实际物理上使用的内存数量是不等的,Android会计算一个叫做Proportional Set Size(PSS)的值,它记录了那些和其他进程进行共享的内存大小。(假设共享内存大小是10M,一共有20个Process在共享使用,根据权重,可能认为其中有0.3M才能真正算是你的进程所使用的)
- Dalvik heap与逻辑上的heap size不吻合,这意味着Android并不会去做heap中的碎片整理用来关闭空闲区域。Android仅仅会在heap的尾端出现不使用的空间时才会做收缩逻辑heap size大小的动作。但是这并不意味着被heap所使用的物理内存大小不能被收缩。在垃圾回收之后,Dalvik会遍历heap并找出不使用的pages,然后使用madvise把那些pages返回给kernel。因此,成对的allocations与deallocations大块的数据可以使得物理内存能够被正常的回收。然而,回收碎片化的内存则会使效率低下很多,因为那些碎片化的分配页面也许会被其他地方所共享到。

3)限制应用的内存

为了维持多任务的功能环境,Android为每一个app都设置了一个硬性的heap size限制。准确的heap size限制随着不同设备的不同RAM大小而各有差异。如果你的app已经到了heap的限制大小并且再尝试分配内存的话,会引起OutOfMemoryError的错误。

在一些情况下,你也许想要查询当前设备的heap size限制大小是多少,然后决定cache的大小。可以通过getMemoryClass()来查询。这个方法会返回一个整数,表明你的app heap size限制是多少megabates。

4)切换应用

当用户在不同应用之间进行切换的时候,不是使用交换空间的办法。Android会把那些不包含foreground组件的进程放到LRU cache中。例如,当用户刚开始启动了一个应用,这个时候为它创建了一个进程,但是当用户离开这个应用,这个进程并没有离开。系统会把这个进程放到cache中,如果用户后来回到这个应用,这个进程能够被resued,从而实现app的快速切换。

如果你的应用有一个被缓存的进程,它被保留在内存中,并且当前不再需要它了,这会对系统的整个性能有影响。因此当系统开始进入低内存状态时,它会由系统根据LRU的规则与其他因素选择杀掉某些进程,为了保持你的进程能够尽可能长久的被cached,请参考下面的章节学习何时释放你的引用。

更对关于不在foreground的进程是Android是如何决定kill掉哪一类进程的问题,请参考[Processes and Threads](#).

第2部分:你的应用该如何管理内存

你应该在开发过程的每一个阶段都考虑到RAM的有限性, 甚至包括在开发开始之前的设计阶段。有许多种设计与实现方式, 他们有着不同的效率, 尽管是对同一种技术的不断组合与演变。

为了使得你的应用效率更高, 你应该在设计与实现代码时, 遵循下面的技术要点。

1)珍惜Services资源

如果你的app需要在后台使用service, 除非它被触发执行一个任务, 否则其他时候都应该是非运行状态。同样需要注意当这个service已经完成任务后停止service失败引起的泄漏。

当你启动一个service, 系统会倾向为了这个Service而一直保留它的Process。这使得process的运行代价很高, 因为系统没有办法把Service所占用的RAM让给其他组件或者被Paged out。这减少了系统能够存放LRU缓存当中的process数量, 它会影响app之间的切换效率。它甚至会导致系统内存使用不稳定, 从而无法继续Hold住所有目前正在运行的Service。

限制你的service的最好办法是使用[IntentService](#), 它会在处理完扔给它的intent任务之后尽快结束自己。更多信息, 请阅读[Running in a Background Service](#)。

当一个service已经不需要的时候还继续保留它, 这对Android应用的内存管理来说是最糟糕的错误之一。因此千万不要贪婪的使得一个Service持续保留。不仅仅是因为它会使得你的app因RAM的限制而性能糟糕, 而且用户会发现那些行为奇怪的app并且卸载它。

2)当你的UI隐藏时释放内存

当用户切换到其它app并且你的app UI不再可见时, 你应该释放你的UI上占用的任何资源。在这个时候释放UI资源可以显著的增加系统cached process的能力, 它会对用户的质量体验有着直接的影响。

为了能够接收到用户离开你的UI时的通知, 你需要实现Activity类里面的[onTrimMemory\(\)](#)回调方法。你应该使用这个方法来监听到[TRIM_MEMORY_UI_HIDDEN](#)级别, 它意味着你的UI已经隐藏, 你应该释放那些仅仅被你的UI使用的资源。

请注意: 你的app仅仅会在所有UI组件的被隐藏的时候接收到onTrimMemory()的回调并带有参数TRIM_MEMORY_UI_HIDDEN。这与onStop()的回调是不同的, onStop会在activity的实例隐藏时会执行, 例如当用户从你的app的某个activity跳转到另外一个activity时onStop会被执行。因此你应该实现onStop回调, 并且在此回调里面释放activity的资源, 例如网络连接, unregister广播接收者。除非接收到[onTrimMemory\(TRIM_MEMORY_UI_HIDDEN\)](#)的回调, 否则你不应该释放你的UI资源。这确保了用户从其他activity切回来时, 你的UI资源仍然可用, 并且可以迅速恢复activity。

3)当内存紧张时释放部分内存

在你的app生命周期的任何阶段, onTrimMemory回调方法同样可以告诉你整个设备的内存资源已经开始紧张。你应该根据onTrimMemory方法中的内存级别来进一步决定释放哪些资源。

- [TRIM_MEMORY_RUNNING_MODERATE](#):你的app正在运行并且不会被列为可杀死的。但是设备正运行于低内存状态下, 系统开始激活杀死LRU Cache中的Process的机制。
- [TRIM_MEMORY_RUNNING_LOW](#):你的app正在运行且没有被列为可杀死的。但是设备正运行于更低内存的状态下, 你应该释放不用的资源用来提升系统性能, 这会直接影响了你的app的性能。
- [TRIM_MEMORY_RUNNING_CRITICAL](#):你的app仍在运行, 但是系统已经把LRU Cache中的大多数进程都已经杀死, 因此你应该立即释放所有非必须的资源。如果系统不能回收足够的RAM数量, 系统将会清除所有的LRU缓存中的进程, 并且开始杀死那些之前被认为不应该杀死的进程, 例如那个进程包含了一个运行中的Service。

同样, 当你的app进程正在被cached时, 你可能会接受到从onTrimMemory()中返回的下面的值之一:

- [TRIM_MEMORY_BACKGROUND](#): 系统正运行于低内存状态并且你的进程正处于LRU缓存名单中最不容易杀掉的位置。尽管你的app进程并不是处于被杀掉的高危险状态, 系统可能已经开始杀掉LRU缓存中的其他进程了。你应该释放那些容易恢复的资源, 以便于你的进程可以保留下来, 这样当用户回退到你的app的时候才能够迅速恢复。
- [TRIM_MEMORY_MODERATE](#): 系统正运行于低内存状态并且你的进程已经接近LRU名单的中部位置。如果系统开始变得更加内存紧张, 你的进程是有可能被杀死的。
- [TRIM_MEMORY_COMPLETE](#): 系统正运行与低内存的状态并且你的进程正处于LRU名单中最容易被杀掉的位置。你应该释放任何不影响你的app恢复状态的资源。

因为onTrimMemory()的回调是在API 14才被加进来的, 对于老的版本, 你可以使用[onLowMemory](#)回调来进行兼容。onLowMemory相当与TRIM_MEMORY_COMPLETE。

Note: 当系统开始清除LRU缓存中的进程时, 尽管它首先按照LRU的顺序来操作, 但是它同样会考虑进程的内存使用量。因此消耗越少的进程则越容易被留下来。

4)检查你应该使用多少的内存

正如前面提到的, 每一个Android设备都会有不同的RAM总大小与可用空间, 因此不同设备为app提供了不同大小的heap限制。你可以通过调用[getMemoryClass\(\)](#)来获取你的app的可用heap大小。如果你的app尝试申请更多的内存, 会出现OutOfMemory的错误。

在一些特殊的情景下, 你可以通过在manifest的application标签下添加largeHeap=true的属性来声明一个更大的heap空间。如果你这样做, 你可以通过[getLargeMemoryClass\(\)](#)来获取到一个更大的heap size。

然而, 能够获取更大heap的设计本意是为了一小部分会消耗大量RAM的应用(例如一个大图片的编辑应用)。不要轻易的因为你需要使用大量的内存而去请求一个大的heap size。只有当你清楚的知道哪里会使用大量的内存并且为什么这些内存必须被保

留时才去使用large heap. 因此请尽量少使用large heap。使用额外的内存会影响系统整体的用户体验，并且会使得GC的每次运行时间更长。在任务切换时，系统的性能会变得大打折扣。

另外, large heap并不一定能够获取到更大的heap。在某些有严格限制的机器上, large heap的大小和通常的heap size是一样的。因此即使你申请了large heap, 你还是应该通过执行getMemoryClass()来检查实际获取到的heap大小。

5)避免bitmaps的浪费

当你加载一个bitmap时, 仅仅需要保留适配当前屏幕设备分辨率的数据即可, 如果原图高于你的设备分辨率, 需要做缩小的动作。请记住, 增加bitmap的尺寸会对内存呈现出2次方的增加, 因为X与Y都在增加。

Note:在Android 2.3.x (API level 10)及其以下, bitmap对象是的pixel data是存放在native内存中的, 它不便于调试。然而, 从Android 3.0(API level 11)开始, bitmap pixel data是分配在你的app的Dalvik heap中, 这提升了GC的工作并且更加容易Debug。因此如果你的app使用bitmap并在旧的机器上引发了一些内存问题, 切换到3.0以上的机器上进行Debug。

6)使用优化的数据容器

利用Android Framework里面优化过的容器类, 例如[SparseArray](#), SparseBooleanArray, 与 LongSparseArray. 通常的HashMap的实现方式更加消耗内存, 因为它需要一个额外的实例对象来记录Mapping操作。另外, SparseArray更加高效在于他们避免了对key与value的autobox自动装箱, 并且避免了装箱后的解箱。

7)请注意内存开销

对你所使用的语言与库的成本与开销有所了解, 从开始到结束, 在设计你的app时谨记这些信息。通常, 表面上看起来无关痛痒(innocuous)的事情也许实际上会导致大量的开销。例如:

- Enums的内存消耗通常是static constants的2倍。你应该尽量避免在Android上使用enums。
- 在Java中的每一个类(包括匿名内部类)都会使用大概500 bytes。
- 每一个类的实例花销是12-16 bytes。
- 往HashMap添加一个entry需要额外占用的32 bytes的entry对象。

8)请注意代码“抽象”

通常, 开发者使用抽象简单的作为"好的编程实践",因为抽象能够提升代码的灵活性与可维护性。然而, 抽象会导致一个显著的开销:通常他们需要同等量的代码用于可执行。那些代码会被map到内存中。因此如果你的抽象没有显著的提升效率, 应该尽量避免他们。

9)为序列化的数据使用nano protobufs

[Protocol buffers](#)是由Google为序列化结构数据而设计的, 一种语言无关, 平台无关, 具有良好扩展性的协议。类似XML, 却比XML更加轻量, 快速, 简单。如果你需要为你的数据实现协议化, 你应该在客户端的代码中总是使用nano protobufs。通常的协议化操作会生成大量繁琐的代码, 这容易给你的app带来许多问题:增加RAM的使用量, 显著增加APK的大小, 更慢的执行速度, 更容易达到DEX的字符限制。

关于更多细节, 请参考[protobuf readme](#)的"Nano version"章节。

10)Avoid dependency injection frameworks

使用类似[Guice](#)或者[RoboGuice](#)等framework injection包是很有效的, 因为他们能够简化你的代码。

RoboGuice 2 smoothes out some of the wrinkles in your Android development experience and makes things simple and fun. Do you always forget to check for null when you getIntent().getExtras()? RoboGuice 2 will help you. Think casting findViewById() to a TextView shouldn't be necessary? RoboGuice 2 is on it. RoboGuice 2 takes the guesswork out of development. Inject your View, Resource, System Service, or any other object, and let RoboGuice 2 take care of the details.

然而, 那些框架会通过扫描你的代码执行许多初始化的操作, 这会导致你的代码需要大量的RAM来map代码。但是mapped pages会长时间的被保留在RAM中。

11)谨慎使用external libraries

很多External library的代码都不是为移动网络环境而编写的, 在移动客户端则显示的效率不高。至少, 当你决定使用一个external library的时候, 你应该针对移动网络做繁琐的porting与maintenance的工作。

即使是针对Android而设计的library, 也可能是很危险的, 因为每一个library所做的事情都是不一样的。例如, 其中一个lib使用的是nano protobufs, 而另外一个使用的是micro protobufs。那么这样, 在你的app里面就有2种protobuf的实现方式。这样的冲突同样可能发生在输出日志, 加载图片, 缓存等等模块里面。

同样不要陷入为了1个或者2个功能而导入整个library的陷阱。如果没有一个合适的库与你的需求相吻合, 你应该考虑自己去实现, 而不是导入一个大而全的解决方案。

12)优化整体性能

官方有列出许多优化整个app性能的文章: [Best Practices for Performance](#). 这篇文章就是其中之一。有些文章是讲解如何优化app的CPU使用效率, 有些是如何优化app的内存使用效率。

你还应该阅读[optimizing your UI](#)来为layout进行优化。同样还应该关注lint工具所提出的建议，进行优化。

13)使用ProGuard来剔除不需要的代码

[ProGuard](#)能够通过移除不需要的代码，重命名类，域与方法等方对代码进行压缩,优化与混淆。使用ProGuard可以是的你的代码更加紧凑，这样能够使用更少mapped代码所需要的RAM。

14)对最终的APK使用zipalign

在编写完所有代码，并通过编译系统生成APK之后，你需要使用[zipalign](#)对APK进行重新校准。如果你不做这个步骤，会导致你的APK需要更多的RAM，因为一些类似图片资源的东西不能被mapped。

Notes::Google Play不接受没有经过zipalign的APK。

15)分析你的RAM使用情况

一旦你获取到一个相对稳定的版本后，需要分析你的app整个生命周期内使用的内存情况，并进行优化，更多细节请参考[Investigating Your RAM Usage](#).

16)使用多进程

如果合适的话，有一个更高级的技术可以帮助你的app管理内存使用：通过把你的app组件切分成多个组件，运行在不同的进程中。这个技术必须谨慎使用，大多数app都不应该运行在多个进程中。因为如果使用不当，它会显著增加内存的使用，而不是减少。当你的app需要在后台运行与前台一样的大量的任务的时候，可以考虑使用这个技术。

一个典型的例子是创建一个可以长时间后台播放的Music Player。如果整个app运行在一个进程中，当后台播放的时候，前台的那些UI资源也没有办法得到释放。类似这样的app可以切分成2个进程：一个用来操作UI，另外一个用来后台的Service。

你可以通过在manifest文件中声明'android:process'属性来实现某个组件运行在另外一个进程的操作。

```
<service android:name=".PlaybackService"
        android:process=":background" />
```

更多关于使用这个技术的细节，请参考原文，链接如下。 <http://developer.android.com/training/articles/memory.html>

编写:[kesenhoo](#)

校对:

Performance Tips性能优化Tips

这篇文章主要是介绍了一些小细节的优化技巧，当这些小技巧综合使用起来的时候，对于整个App的性能提升还是有作用的，只是不能较大幅度的提升性能而已。选择合适的算法与数据结构才应该是你首要考虑的因素，在这篇文章中不会涉及这方面。你应该使用这篇文章中的小技巧作为平时写代码的习惯，这样能够提升代码的效率。

通常来说，高效的代码需要满足下面两个规则：

- 不要做冗余的动作
- 如果能避免，尽量不要分配内存

代码的执行效果会受到设备CPU,设备内存,系统版本等诸多因素的影响。为了确保代码能够在不同设备上都运行良好，需要最大化代码的效率。

避免创建不必要的对象

虽然GC可以回收不用的对象，可是为这些对象分配内存，并回收它们同样是需要耗费资源的。因此请尽量避免创建不必要的对象，有下面一些例子来说明这个问题：

- 如果你需要返回一个String对象，并且你知道它最终会需要连接到一个StringBuffer，请修改你的实现方式，避免直接进行连接操作，应该采用创建一个临时对象来做这个操作。
- 当从输入的数据集中抽取Strings的时候，尝试返回原数据的substring对象，而不是创建一个重复的对象。

一个稍微激进点的做法是把所有多维的数据分解成1维的数组：

- 一组int数据要比一组Integer对象要好很多。可以得知，两组1维数组要比一个2维数组更加的有效率。同样的，这个道理可以推广至其他原始数据类型。
- 如果你需要实现一个数组用来存放(Foo,Bar)的对象，尝试分解为Foo[]与Bar[]要比(Foo,Bar)好很多。(当然，为了某些好的API的设计，可以适当做一些妥协。但是在自己的代码内部，你应该多多使用分解后的容易。

通常来说，需要避免创建更多的对象。更少的对象意味着更少的GC动作，GC会对用户体验有比较直接的影响。

选择**Static**而不是**Virtual**

如果你不需要访问一个对象的值域,请保证这个方法是**static**类型的,这样方法调用将快15%-20%。这是一个好的习惯,因为你可以通过方法声明中得知调用无法改变这个对象的状态。

常量声明为Static Final

先看下面这种声明的方式

```
static int intVal = 42;  
static String strVal = "Hello, world!";
```

编译器会使用方法来初始化上面的值，之后访问的时候会需要先到它那里查找，然后才返回数据。我们可以使用static final来提升性能：

```
static final int intVal = 42;  
static final String strVal = "Hello, world!";
```

这时再也不需要上面的那个方法来多余查找动作了。所以，请尽可能的为常量声明为**static final**类型的。

避免内部的Getters/Setters

像C++等native language,通常使用getters(`i = getCount()`)而不是直接访问变量(`i = mCount`).这是编写C++的一种优秀习惯,而且通常也被其他面向对象的语言所采用,例如C#与Java,因为编译器通常会做inline访问,而且你需要限制或者调试变量,你可以在任何时候在getter/setter里面添加代码。然而,在Android上,这是一个糟糕的写法。Virtual method的调用比起直接访问变量要耗费更多。那么合理的做法是:在面向对象的设计当中应该使用getter/setter,但是在类的内部你应该直接访问变量.没有JIT(Just In Time Compiler)时,直接访问变量的速度是调用getter的3倍。有JIT时,直接访问变量的速度是通过getter访问的7倍。请注意,如果你使用[ProGuard](#),你可以获得同样的效果,因为ProGuard可以为你inline accessors.

使用增强的For循环

请比较下面三种循环的方法：

```
static class Foo {
    int mSplat;
}

Foo[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one() {
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;

    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two() {
    int sum = 0;
    for (Foo a : mArray) {
        sum += a.mSplat;
    }
}
```

- zero()是最慢的，因为JIT没有办法对它进行优化。
- one()稍微快些。
- two() 在没有做JIT时是最快的，可是如果经过JIT之后，与方法one()是差不多一样快的。它使用了增强的循环方法for-each。

所以请尽量使用for-each的方法，但是对于ArrayList，请使用方法one()。

使用包级访问而不是内部类的私有访问

参考下面一段代码

```
public class Foo {  
    private class Inner {  
        void stuff() {  
            Foo.this.doStuff(Foo.this.mValue);  
        }  
    }  
  
    private int mValue;  
  
    public void run() {  
        Inner in = new Inner();  
        mValue = 27;  
        in.stuff();  
    }  
  
    private void doStuff(int value) {  
        System.out.println("Value is " + value);  
    }  
}
```

Foo\$Inner里面有访问外部类的一个变量。这样的做法会给系统造成额外的麻烦，请尽量避免。

避免使用**float**类型

Android系统中float类型的数据存取速度是int类型的一半，尽量优先采用int类型。

使用库函数

尽量使用`System.arraycopy()`等一些封装好的库函数，它的效率是手动编写`copy`实现的9倍多。

Tip: Also see Josh Bloch's *Effective Java*, item 47.

谨慎使用**native**函数

当你需要把已经存在的native code迁移到Android，请谨慎使用JNI。如果你要使用JNI,请学习[JNI Tips](#)

关于性能的误区

在没有做JIT之前，使用一种确切的数据类型确实要比抽象的数据类型速度要更有效率。(例如，使用HashMap要比Map效率更高。)有误差效率要高一倍，实际上只是6%左右。而且，在JIT之后，他们直接并没有太多差异。

关于测量

上面文档中出现的数据是Android的实际运行效果。我们可以用[Traceview](#) 来测量，但是测量的数据是没有经过JIT优化的，所以实际的效果应该是要比测量的数据稍微好些。

关于如何测量与调试，还可以参考下面两篇文章：

- [Profiling with Traceview and dmtracedump](#)
- [Analysing Display and Performance with Systrace](#)

编写:

校对:

提升**Layout**的性能

编写:

校对:

优化**layout**的层级

编写:

校对:

使用include标签重用layouts

编写:

校对:

按需加载视图

编写:

校对:

使得**ListView**滑动顺畅

编写:[kesenhoo](#)

校对:

优化电池寿命

显然，手持设备的电量需要引起很大的重视。通过这一系列的课程，可以学会如何根据设备电池状态来改变App的某些行为与功能。

通过在断开连接时关闭后台服务，在电量减少时减少更新数据的频率等等操作可以在不影响用户体验的前提下，确保App对电池寿命的影响减到最小。

编写:[kesenhoo](#)

校对:

Monitoring the Battery Level and Charging State[监测电池的电量与充电状态]

当你想通过改变后台更新操作的频率来减少对电池寿命的影响，那么先手需要检查当前电量与充电状态。

电池的电量与是否在充电状态会影响到一个程序去执行更新的操作。当设备在进行AC充电时，程序做任何操作都不太会受到电量的影响，所以在大多数时候，我们可以在设备充电时做很多想做的事情（刷新数据，下载文件等），相反的，如果设备没有在充电状态，那么我们就需要尽量减少设备的更新操作等来延长电池的续航能力。

同样的，我们可以通过检查电池目前的电量来减少甚至停止一些更新操作。

1)Determine the Current Charging State[判断当前充电状态]

[BatteryManager](#)会广播一个带有电池与充电详情的[Sticky Intent](#) 因为广播的是一个sticky intent, 那么不需要注册BroadcastReceiver。仅仅只需要简单的call一个参null参数的registerReceiver()方法。

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
```

我们可以从intent里面提取出当前的充电状态与是否通过USB或者AC充电器来充电。

```
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
    status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

我们可以从intent里面提取出当前的充电状态与是否通过USB或者AC充电器来充电。通常的做法是在使用AC充电时最大化后台更新操作, 在使用USB充电时降低更新操作, 不在充电状态时, 最小化更新操作。

2)Monitor Changes in Charging State[监测充电状态的改变]

充电状态随时可能改变，显然，需要通过检查充电状态的改变来通知App改变某些行为。

BatteryManager会在设备连接或者断开充电器的时候广播一个action。接收到这个广播是很重要的，即使我们的app没有在运行。特别是在是否接收这个广播会对app决定后台更新频率产生影响的前提下。因此很有必要在manifest文件里面注册一个监听来接收ACTION_POWER_CONNECTED 与 ACTION_POWER_DISCONNECTED的intent。

```
<receiver android:name=".PowerConnectionReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
    </intent-filter>
</receiver>
```

```
public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
        boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
            status == BatteryManager.BATTERY_STATUS_FULL;

        int chargePlug = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        boolean usbCharge = chargePlug == BATTERY_PLUGGED_USB;
        boolean acCharge = chargePlug == BATTERY_PLUGGED_AC;
    }
}
```


3)Determine the Current Battery Level[判断当前电池电量]

在一些情况下，获取到当前电池电量也是很有帮助的。我们可以在获知电量少于某个级别的时候减少某些后台操作。我们可以从获取到电池状态的intent中提取出电池电量与容量等信息。

```
int level = battery.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);  
int scale = battery.getIntExtra(BatteryManager.EXTRA_SCALE, -1);  
float batteryPct = level / (float)scale;
```

4)Monitor Significant Changes in Battery Level[检测电量的有效改变]

虽然我们可以轻易的不间断的检测电池状态，但是这并不是必须的。通常来说，我们只需要检测电量的某些有效改变，特别是设备在进入或者离开低电量状态的时候。下面的例子，电量监听器只会在设备电量进入低电量或者离开低电量时候才会触发，仅仅需要监听ACTION_BATTERY_LOW与ACTION_BATTERY_OKAY.

```
<receiver android:name=".BatteryLevelReceiver">
<intent-filter>
  <action android:name="android.intent.action.ACTION_BATTERY_LOW"/>
  <action android:name="android.intent.action.ACTION_BATTERY_OKAY"/>
</intent-filter>
</receiver>
```

通常我们都需要在进入低电量的情况下，关闭所有后台程序来维持设备的续航，因为这个时候做任何更新等操作都是无谓的，很可能在你还没有来的及操作刚才更新的内容的时候就自动关机了。 In many cases, the act of charging a device is coincident with putting it into a dock. The next lesson shows you how to determine the current dock state and monitor for changes in device docking.

编写:[kesenhoo](#)

校对:

Determining and Monitoring the Docking State and Type[判断并监测设备的停驻状态与类型]

在上一课中有这样一句话：In many cases, the act of charging a device is coincident with putting it into a dock.

在很多情况下，为设备充电也是一种设备停驻方式

Android设备能够有好几种停驻状态。包括车载模式，家庭模式与数字对战模拟模式[这个有点奇怪]。停驻状态通常与充电状态是非常密切关联的。

停驻模式会如何影响更新频率这完全取决于app的设置。我们可以选择在桌面模式下频繁的更新数据也可以选择车载模式下关闭更新操作。相反的，你也可以选择在车载模式下最大化更新交通数据频率。

停驻状态也是以sticky intent方式来广播的，这样可以通过查询intent里面的数据来判断是否目前处于停驻状态，处于哪种停驻状态。

1) Determine the Current Docking State[判断当前停驻状态]

因为停驻状态的广播内容也是sticky intent(ACTION_DOCK_EVENT)，所以不需要注册BroadcastReceiver。

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_DOCK_EVENT);
Intent dockStatus = context.registerReceiver(null, ifilter);

int dockState = battery.getIntExtra(EXTRA_DOCK_STATE, -1);
boolean isDocked = dockState != Intent.EXTRA_DOCK_STATE_UNDOCKED;
```


3)Monitor for Changes in the Dock State or Type[监测停驻状态或者类型的改变]

只需要像下面一样注册一个监听器：

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

Receiver获取到信息后可以像上面那样检查需要的数据。

编写:[kesenhoo](#)

校对:

Determining and Monitoring the Connectivity Status[判断并监测网络连接状态]

通常我们会有一些计划的任务，比如重复闹钟，后台定时启动的任务等。但是如果我们的网络没有连接上，那么就没有必要启动那些需要连接网络的任务。我们可以使用ConnectivityManager来检查是否连接上网络，是何种网络。[通过网络的连接状况改变，相应的改变app的行为，减少无谓的操作，从而延长设备的续航能力]

1)Determine if You Have an Internet Connection[判断当前是否有网络连接]

显然如果没有网络连接，那么就没有必要做那些需要联网的事情。下面是一个检查是否有网络连接的例子：

```
ConnectivityManager cm =  
    (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);  
  
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();  
boolean isConnected = activeNetwork.isConnectedOrConnecting();
```

2)Determine the Type of your Internet Connection[判断连接网络的类型]

设备通常可以有移动网络，WiMax,Wi-Fi与以太网连接等类型。通过查询当前活动的网络类型，可以根据网络的带宽做适合的事情。

```
boolean isWiFi = activeNetwork.getType() == ConnectivityManager.TYPE_WIFI;
```

使用移动网络会比Wi-Fi花费代价更大，所以多数情况下，在移动网络情况下减少一些数据的获取操作，同样，一些像下载文件等操作需要等有Wi-Fi的情况下才开始。 如果已经关闭了更新操作，那么需要监听网络切换，当有比较好的网络时重新启动之前取消的操作。

3)Monitor for Changes in Connectivity[监测网络连接的切换]

当网络连接被改变的时候，ConnectivityManager会broadcast CONNECTIVITY_ACTION ("android.net.conn.CONNECTIVITY_CHANGE") 的动作消息。我们需要在manifest文件里面注册一个带有像下面action一样的Receiver:

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

通常网络的改变会比较频繁，我们没有必要不间断的注册监听网络的改变。通常我们会在有Wi-Fi的时候进行下载动作，若是网络切换到移动网络则通常会暂停当前下载，监听到恢复到Wi-Fi的情况则开始恢复下载。

编写:[kesenhoo](#)

校对:

Manipulating Broadcast Receivers On Demand[按需操控广播接收者]

简单的方法是为我们监测的状态创建一个BroadcastReceiver，并在manifest中为每一个状态进行注册监听。然后，每一个Receiver根据当前设备的状态来简单重新安排下一步执行的任务。[这句话感觉理解有点问题]

上面那个方法的副作用是，设备会在每次收到广播都被唤醒，这有点超出期望，因为有些广播是不希望唤醒设备的。

更好的方法是根据程序运行情况开启或者关闭广播接收者。这样的话，那些在manifest中注册的receivers仅仅会在需要的时候才被激活。

1)Toggle and Cascade State Change Receivers to Improve Efficiency[切换是否开启这些状态Receivers来提高效率]

我们可以使用PackageManager来切换任何一个在manifest里面定义好的组件的开启状态。可以使用下面的方法来开启或者关闭任何一个broadcast receiver:

```
ComponentName receiver = new ComponentName(context, myReceiver.class);

PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
    PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
    PackageManager.DONT_KILL_APP)
```

使用这种技术，如果我们判断到网络链接已经断开，那么可以在这个时候关闭除了connectivity-change的之外的所有Receivers。

相反的，一旦重新建立网络连接，我们可以停止监听网络链接的改变。而仅仅在执行需要联网的操作之前判断当前网络是否可以用即可。

你可以使用上面同样的技术来暂缓一个需要带宽的下载操作。可以开启receiver来监听是否连接上Wi-Fi来重新开启下载的操作。

编写:

校对:

多线程操作

编写:

校对:

指定一段代码执行在一个线程

编写:

校对:

为多线程创建线程池

编写:

校对:

执行代码运行在线程池中

编写:

校对:

与UI线程进行交互

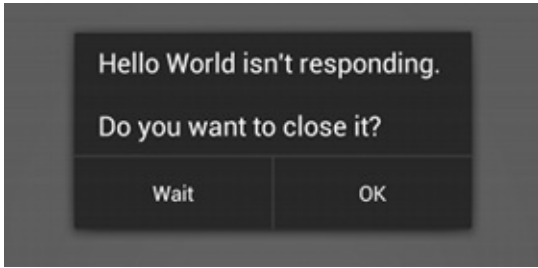
编写:[kesenhoo](#)

校对:

避免出现程序无响应ANR(Keeping Your App Responsive)

可能你写的代码在性能测试上表现良好，但是你的应用仍然有时候会反应迟缓(sluggish)，停顿(hang)或者长时间卡死(freeze)，或者应用处理输入的数据花费时间过长。对于你的应用来说最糟糕的事情是出现"程序无响应(Application Not Responding)"(ANR)的警示框。

在Android中，系统通过显示ANR警示框来保护程序的长时间无响应。对话框如下：



此时，你的应用已经经历过一段时间的无法响应了，因此系统提供用户可以退出应用的选择。为你的程序提供良好的响应性是至关重要的，这样才能够避免系统为用户显示ANR的警示框。

这节课描述了Android系统是如何判断一个应用不可响应的。这节课还会提供程序编写的指导原则，确保你的程序保持响应性。

是什么导致了ANR?(What Triggers ANR?)

通常来说，系统会在程序无法响应用户的输入事件时显示ANR。例如，如果一个程序在UI线程执行I/O操作(通常是网络请求或者是文件读写)，这样系统就无法处理用户的输入事件。或者是应用在UI线程花费了太多的时间用来建立一个复杂的在内存中的数据结构，又或者是在一个游戏程序的UI线程中执行了一个复杂耗时的计算移动的操作。确保那些计算操作高效是很重要的，不过即使是最高效的代码也是需要花时间执行的。

对于你的应用中任何可能执行时间长的操作，你都不应该执行在**UI**线程。你可以创建一个工作线程，把那些操作都执行在工作线程中。这确保了UI线程(这个线程会负责处理UI事件)能够顺利执行，也预防了系统因代码僵死而崩溃。因为UI线程是和类级别相关联的，你可以把相应性作为一个类级别(class-level)的问题(相比来说，代码性能则属于方法级别(method-level)的问题)

在Android中，程序的响应性是由Activity Manager与Window Manager系统服务来负责监控的。当系统监测到下面的条件之一时会显示ANR的对话框:

- 对输入事件(例如硬件点击或者屏幕触摸事件)，5秒内都无响应。
- BroadcastReceiver不能在10秒内结束接收到任务。

如何避免ANRs(How to Avoid ANRs)

Android程序通常是执行在默认的UI线程(也可以成为main线程)中的。这意味着在UI线程中执行的任何长时间的操作都可能触发ANR，因为程序没有给自己处理输入事件或者broadcast事件的机会。

因此，任何执行在UI线程的方法都应该尽可能的简短快速。特别是，在activity的生命周期的关键方法onCreate()与onResume()方法中应该尽可能的做比较少的事情。类似网络或者DB操作等可能长时间执行的操作，或者是类似调整bitmap大小等需要长时间计算的操作，都应该执行在工作线程中。(在DB操作中，可以通过异步的网络请求)。

为了执行一个长时间的耗时操作而创建一个工作线程最方便高效的方式是使用AsyncTask。只需要继承AsyncTask并实现doInBackground()方法来执行任务即可。为了把任务执行的进度呈现给用户，你可以执行publishProgress()方法，这个方法会触发onProgressUpdate()的回调方法。在onProgressUpdate()的回调方法中(它执行在UI线程)，你可以执行通知用户进度的操作，例如：

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

为了能够执行这个工作线程，只需要创建一个实例并执行execute()：

```
new DownloadFilesTask().execute(url1, url2, url3);
```

相比起AsyncTask来说，创建自己的线程或者HandlerThread稍微复杂一点。如果你想这样做，你应该通过Process.setThreadPriority()并传递THREAD_PRIORITY_BACKGROUND来设置线程的优先级为"background"。如果你不通过这种方式来给线程设置一个低的优先级，那么这个线程仍然会使得你的应用显得卡顿，因为这个线程默认与UI线程有着同样的优先级。

如果你实现了Thread或者HandlerThread，请确保你的UI线程不会因为等待工作线程的某个任务而去执行Thread.wait()或者Thread.sleep()。UI线程不应该去等待工作线程完成某个任务，你的UI现场应该提供一个Handler给其他工作线程，这样工作线程能够通过这个Handler在任务结束的时候通知UI线程。使用这样的方式来设计你的应用程序可以使得你的程序UI线程保持响应性，以此来避免ANR。

BroadcastReceiver有特定执行时间的限制说明了broadcast receivers应该做的是：简短快速的任務，避免执行费时的操作，例如保存数据或者注册一个Notification。正如在UI线程中执行的方法一样，程序应该避免在broadcast receiver中执行费时的长任务。但不是采用通过工作线程来执行复杂的任务的方式，你的程序应该启动一个IntentService来响应intent broadcast的长时间任务。

Tip: 你可以使用StrictMode来帮助寻找因为不小心加入到UI线程的潜在的长时间执行的操作，例如网络或者DB相关的任务。

增加响应性(Reinforce Responsiveness)

通常来说，100ms - 200ms是用户能够察觉到卡顿的上限。这样的话，下面有一些避免ANR的技巧：

- 如果你的程序需要响应正在后台加载的任务，在你的UI中可以显示ProgressBar来显示进度。
- 对游戏程序，在工作线程执行计算的任务。
- 如果你的程序在启动阶段有一个耗时的初始化操作，可以考虑显示一个闪屏，要么尽快的显示主界面，然后马上显示一个加载的对话框，异步加载数据。无论哪种情况，你都应该显示一个进度信息，以免用户感觉程序有卡顿的情况。
- 使用性能测试工具，例如Systrace与Traceview来判断程序中影响响应性的瓶颈。

编写:

校对:

JNI Tips

编写:

校对:

SMP for Android

编写:[lltowq](#)(未验证)

校对:

安全与隐私

这些课程与文章提供关于保持你的app数据安全。

安全要点

怎样执行多个任务和保持你app数据或用户数据安全。

HTTPS和SSL的安全

怎样确保当你执行网络传输时app的安全。

企业级开发

怎样实现驱动管理政策对于企业级的app

编写:[lltowq](#)

校对:

安全要点

Android安全特性建立在操作系统和显著降低应用程序安全问题的频次和影响。系统被设计成你可以默认使用系统和文件权限建立app，避免关于安全方面困难的决定。

数据存储

使用权限

使用网络

输入验证

处理用户数据

使用**WebView**

使用密码

使用进程间通信

动态加载代码

在虚拟机器安全性

在本地代码的安全

编写:

校对:

使用HTTPS与SSL

SSL现在叫传输层安全([TLS](#))是普通区对于加密通信对于客户端和服务端。它的app可能使用错误的SSL例如恶意的实体可能打断app数据在网络层上。帮助你确保不会发生在你的app上。这篇文章重点关于普通的陷阱，当你使用安全网络协议和地址，一些大的问题集中在[Public-Key Infrastructure\(PKI\)](#)

概念

一个**HTTP**的例子

服务器普通问题的验证

证书

- 无法识别证书机构
- 自我签名服务器证书
- 缺少中间证书颁发机构

主机名的问题

验证

关于直接使用**SSL Socket**的警告

黑名单

阻塞

客户端验证

编写:

校对:

开发企业版App

在这节课你将会学到APIs和技术，你使用它开发企业级App。

课程

加强与设备管理策略安全

在这节课程，你将学习怎样创建一个具有安全意识能管理访问它的内容app，通过加强设备管理策略。

编写:

校对:

测试程序

编写:

校对:

测试你的**Activity**

编写:

校对:

建立测试环境

编写:

校对:

创建与执行测试用例

编写:

校对:

测试UI组件

编写:

校对:

创建单元测试

编写:

校对:

创建功能测试

编写:

校对:

分发与盈利

编写:

校对:

售卖**App**内置产品

编写:

校对:

准备你的App

编写:

校对:

建立售卖的产品

编写:

校对:

购买产品

编写:

校对:

测试你的**App**

编写:

校对:

维护多个**APK**

编写:

校对:

为不同**API Level**创建多个**APK**

编写:

校对:

为不同屏幕大小创建多个**APK**

编写:

校对:

为不同的**GL Texture**创建多个**APK**

编写:

校对:

为**2**种以上的维度创建多个**APK**

编写:

校对:

编写:

校对:

在不影响用户体验的前提下添加广告

Table of Contents

序言	2
开始	4
建立你的第一个App	6
创建一个Android项目	8
执行你的程序	12
建立一个简单的用户界面	16
启动另外的Activity	18
添加ActionBar	20
建立ActionBar	23
添加Action按钮	27
ActionBar的风格化	33
ActionBar的覆盖层叠	42
兼容不同的设备	46
适配不同的语言	49
适配不同的屏幕	53
适配不同的系统版本	57
管理Activity的生命周期	62
启动与销毁Activity	65
暂停与恢复Activity	71
停止与重启Activity	75
重新创建Activity	79
使用Fragment建立动态的UI	83
创建一个Fragment	86
建立灵活动态的UI	90
Fragments之间的交互	94
数据保存	99
保存到Preference	101
保存到文件	106
保存到数据库	114
与其他应用的交互	122
Intent的发送	125
接收Activity返回的结果	131
Intent过滤	135
分享	140
分享简单的数据	142
给其他App发送简单的数据	144
接收从其他App返回的数据	148
添加一个简便的分享动作	152
分享文件	156
建立文件分享	158
分享文件	162
请求分享一个文件	169
获取文件信息	173
使用NFC分享文件	177
发送文件给其他设备	179
接收其他设备的文件	185
多媒体	185
管理音频播放	193
控制你得应用的音量与播放	195
管理音频焦点	200
兼容音频输出设备	205
拍照	209
简单的拍照	211
简单的录像	221
控制相机硬件	226
打印	235

打印照片	237
打印HTML文档	240
打印自定义文档	245
图像	252
高效显示Bitmap	254
高效加载大图	257
非UI线程处理Bitmap	261
缓存Bitmap	266
管理Bitmap的内存占用	273
在UI上显示Bitmap	278
使用OpenGL ES显示图像	286
建立OpenGL ES的环境	289
定义Shapes	295
绘制Shapes	299
运用投影与相机视图	303
添加移动	308
响应触摸事件	312
动画	317
淡入淡出两个View	319
使用ViewPager实现屏幕滑动	321
卡片翻转的动画	323
缩放动画	325
控件切换动画	327
连接	329
无线连接设备	331
使得网络服务可发现	334
使用WiFi建立P2P连接	336
使用WiFi P2P服务	338
网络连接操作	340
连接到网络	342
管理使用的网络	349
解析XML数据	358
高效下载	369
为网络访问更加高效而优化下载	372
最小化更新操作的影响	380
避免下载多余的数据	384
根据网络类型改变下载模式	388
使用Sync Adapter传输数据	392
创建Stub授权器	394
创建Stub Content Provider	396
创建Sync Adapter	398
执行Sync Adapter	400
使用Volley执行网络数据传输	402
发送简单的网络请求	405
建立请求队列	411
创建标准的网络请求	415
实现自定义的网络请求	421
云服务	425
云同步	427
使用备份API	430
使用Google Cloud Messaging	437
解决云同步的保存冲突	443
用户信息	450
获取联系人列表	452
获取联系人详情	463
修改联系人信息	465
显示联系人头像	467
位置信息	469

获取当前位置	472
获取位置更新	480
显示位置地址	490
创建并监视异常区域	496
识别用户的当下活动	516
使用模拟位置进行测试	518
交互	520
设计高效的导航	522
规划屏幕界面与他们之间的关系	524
为多种大小的屏幕进行规划	526
提供向下与侧滑的导航	528
提供向上与暂时的导航	530
综合上面所有的导航	532
实现高效的导航	534
使用Tabs创建Swipe视图	536
创建抽屉导航	538
提供向上的导航	540
提供向后的导航	542
实现向下的导航	544
通知提示用户	546
建立Notification	549
当启动Activity时保留导航	555
更新Notification	559
使用BigView风格	563
显示Notification进度	567
增加搜索功能	571
建立搜索界面	576
保存并搜索数据	581
保持向下兼容	586
使得你的App内容可被Google搜索	591
为App内容开启深度链接	593
为索引指定App内容	595
UI	597
为多屏幕设计	599
兼容不同的屏幕大小	601
兼容不同的屏幕密度	608
实现可适应的UI	610
为TV进行设计	612
为TV优化Layout	614
为TV优化导航	616
处理不支持TV的功能	618
创建自定义View	620
创建自定义的View类	623
实现自定义View的绘制	630
使得View可交互	636
优化自定义View	641
创建向后兼容的UI	645
抽象新的APIs	647
代理至新的APIs	649
使用旧的APIs实现新API的效果	651
使用版本敏感的组件	653
实现辅助功能	655
开发辅助程序	658
开发辅助服务	664
管理系统UI	670
淡化系统Bar	672
隐藏系统Bar	674
隐藏导航Bar	676
全屏沉浸式应用	678

响应UI可见性的变化	680
用户输入	682
使用触摸手势	684
检测常用的手势	687
跟踪手势移动	687
Scroll手势动画	694
处理多触摸手势	696
拖拽与缩放	698
管理ViewGroup中的触摸事件	700
处理按键点击	702
指定输入法类型	704
处理输入法可见性	706
兼容硬件导航	708
处理输入命令	710
兼容游戏控制器	712
处理控制器输入动作	714
支持不同的Android系统版本	716
支持多个控制器	718
后台任务	720
在IntentService中执行后台任务	722
创建IntentService	725
发送工作任务到IntentService	729
报告后台任务执行状态	731
使用CursorLoader在后台加载数据	736
使用CursorLoader执行查询任务	740
处理查询的结果	745
管理设备的唤醒状态	749
保持设备的唤醒	752
制定重复定时的任务	756
性能优化	763
管理应用的内存	765
性能优化Tips	771
提升Layout的性能	784
优化layout的层级	786
使用include标签重用layouts	788
按需加载视图	790
使得ListView滑动顺畅	792
优化电池寿命	794
监测电量与充电状态	796
判断与监测Docking状态	802
判断与监测网络连接状态	807
根据需要操作Broadcast接受者	812
多线程操作	815
指定一段代码执行在一个线程	817
为多线程创建线程池	819
执行代码运行在线程池中	821
与UI线程进行交互	823
避免出现程序无响应ANR	825
JNI Tips	830
SMP for Android	832
安全与隐私	834
Security Tips	839
使用HTTPS与SSL	842
企业版App	846
测试程序	849
测试你的Activity	851
建立测试环境	853
创建与执行测试用例	855

测试UI组件	857
创建单元测试	859
创建功能测试	861
分发与盈利	863
售卖App内置产品	865
准备你的App	867
建立售卖的产品	869
购买产品	871
测试你的App	873
维护多个APK	875
为不同API Level创建多个APK	877
为不同屏幕大小创建多个APK	879
为不同的GL Texture创建多个APK	881
为2种以上的维度创建多个APK	883
App盈利	885
在不影响用户体验的前提下添加广告	887