[1. 绑定到Service 3](#_Toc449620569)

[2. 管理绑定服务的生命周期 4](#_Toc449620570)

[3. 通知 5](#_Toc449620571)

[1. 创建一个通知 5](#_Toc449620572)

[2. 一个通知必需的内容 5](#_Toc449620573)

[3. 通知的actions 6](#_Toc449620574)

[4. 通知的优先级 6](#_Toc449620575)

[5. 创建一个简单的通知 6](#_Toc449620576)

[6. 为通知应用一个可展开的布局 7](#_Toc449620577)

[7. 处理版本兼容性（Notification） 8](#_Toc449620578)

[8. 管理通知： 8](#_Toc449620579)

[9. 更新通知 8](#_Toc449620580)

[10. 移除通知 9](#_Toc449620581)

[11. 当打开一个Activity时保留导航 9](#_Toc449620582)

[12. 建立一个一般的PendingIntent 10](#_Toc449620583)

[13. 建立一个特殊Activity的PendingIntent 11](#_Toc449620584)

[14. 在通知中显示进度条 13](#_Toc449620585)

[15. 显示一个固定持续时间的进度指示器 13](#_Toc449620586)

[16. 显示一个持续的活动的指示器 14](#_Toc449620587)

[17. 通知的元数据 14](#_Toc449620588)

[18. 灵活的通知 15](#_Toc449620589)

[19. 锁屏通知 15](#_Toc449620590)

[20. 设置可见 15](#_Toc449620591)

[21. 在锁屏时控制媒体播放 15](#_Toc449620592)

[22. 自定义通知的布局 16](#_Toc449620593)

[23. 为自定义通知文本使用样式资源 16](#_Toc449620594)

[4. 在Android5.0中通知的新特色（Notifications New in Android5.0） 17](#_Toc449620595)

[1. 在Android5.0中的新通知 17](#_Toc449620596)

[2. 剖析一个通知 17](#_Toc449620597)

[1) 基本布局 17](#_Toc449620598)

[2) 可展开的布局 17](#_Toc449620599)

[3) 通知的actions 19](#_Toc449620600)

[3. Heads-up Notification 19](#_Toc449620601)

[4. 让通知变得个性化的方法 19](#_Toc449620602)

[5. 为通知设置一个分类 19](#_Toc449620603)

[6. Summarize your notifications 20](#_Toc449620604)

[7. 是通知成为可选项Make notifications optional 21](#_Toc449620605)

[8. 使通知LED灯合适的跳动 21](#_Toc449620606)

[9. 正在进行的通知（Ongoing notifications） 21](#_Toc449620607)

[5. Material Design 21](#_Toc449620608)

[1. 开始 21](#_Toc449620609)

[2. 支持向后兼容（Maintain backward compatibility） 22](#_Toc449620610)

[3. 使用material design更新你的app 22](#_Toc449620611)

[4. 应用Material 主题 22](#_Toc449620612)

[5. 设计你的布局 22](#_Toc449620613)

[6. 在你的视图中指定Elevation 22](#_Toc449620614)

[7. 创建列表和卡片 23](#_Toc449620615)

[8. 自定义动画 23](#_Toc449620616)

[6. 使用Material 主题 24](#_Toc449620617)

[1. 自定义颜色调色板 25](#_Toc449620618)

[2. 自定义状态栏 25](#_Toc449620619)

[7. 创建列表和卡片（Creating Lists and Cards） 25](#_Toc449620620)

[1. 创建列表 25](#_Toc449620621)

[2. 动画 26](#_Toc449620622)

[3. 例子 26](#_Toc449620623)

[4. 创建卡片 27](#_Toc449620624)

[5. 添加CardView和RecyclerView的引用 28](#_Toc449620625)

[8. 定义阴影和Clipping Views 28](#_Toc449620626)

[1. 为你的视图指定Elevation 29](#_Toc449620627)

[2. 自定义视图的阴影和轮廓 29](#_Toc449620628)

[3. Clip Views 30](#_Toc449620629)

[9. 高效显示位图（Displaying Bitmaps Efficiently） 30](#_Toc449620630)

[1. 高效加载大的位图（Loading Large Bitmaps Efficiently） 30](#_Toc449620631)

[1) 读取位图的尺寸和类型 30](#_Toc449620632)

[2) 加载一个按比例缩小的版本到内存 31](#_Toc449620633)

[2. 不在UI线程中处理位图（Processing Bitmaps Off the UI Thread） 33](#_Toc449620634)

[1) 使用AsyncTask 33](#_Toc449620635)

[2) 处理并发 34](#_Toc449620636)

[3. 缓存位图 37](#_Toc449620637)

[1) 使用内存缓存（Use a Memory Cache） 37](#_Toc449620638)

[2) 使用存盘缓存（Use a Disk Cache） 39](#_Toc449620639)

[3) 处理配置变化（Handle Configuration Changes） 42](#_Toc449620640)

[4. 管理存放位图的内存（Managing Bitmap Memory） 43](#_Toc449620641)

[1) 在Android2.3.3及以下如何管理内存（Manage Memory on Android 2.3.3 and Lower） 44](#_Toc449620642)

[2) 在Android3.0及更高版本中如何管理内存(Manage Memory on Android 3.0 and Higher) 45](#_Toc449620643)

[3) 为之后使用保存位图（Save a bitmap for later use） 45](#_Toc449620644)

[4) 使用一个存在的位图（Use an existing bitmap） 47](#_Toc449620645)

[10. 用户界面最佳实践（Best Practices for User Interface） 50](#_Toc449620646)

[1. 适应多种屏幕的设计（Designing for Multiple Screens） 50](#_Toc449620647)

[1) 支持不同屏幕大小 50](#_Toc449620648)

[2. 创建自定义视图（Creating Custom Views） 50](#_Toc449620649)

[1) 创建视图类 50](#_Toc449620650)

[2) 自定义绘制（Custom Drawing） 53](#_Toc449620651)

1. 绑定到Service

应用程序组件（客户端）可以通过调用 **bindService()** 绑定到一个service。之后Android操作系统调用service的 **onBind()** 方法，它会返回一个 **IBinder** 来与service互相影响。

这个绑定时 **异步** 的，**bindService()** 方法会立即返回并且不会把 **IBinder** 对象返回给客户端。为了接收 **IBinder** 对象，客户端必须创建一个 **ServiceConnection** 实例并且把它传给 **bindService()** 方法。这个**ServiceConnection** 实例包含一个回调方法，使得系统可以通过调用它把 **IBinder** 对象传递给客户端。

**注意：**四大组件中只有：activities，services和content providers可以绑定到一个service。你不能使用broadcast receiver绑定到一个service。（**Note:** Only activities, services, and content providers can bind to a service—you **cannot** bind to a service from a broadcast receiver.）

所以，从你的客户端绑定到service，你必须做以下事情：

1. 实现 **ServiceConnection** 接口。  
   你的实现必须重写以下两个方法：  
   **onServiceConnected():**  
   系统会调用这个方法将service 的 onBind() 方法返回的**IBinder** 对象传递给客户端。  
   **onServiceDisconnected():**  
   当客户端和服务端建立的连接意外的断开时系统会调用这个方法，比如说当服务崩溃或者被杀死的时候。当然，当客户端主动调用 **unbindService()** 方法解绑时，系统不会调用这个方法。
2. 调用 **bindService()** ,并传递 **ServiceConnection** 实现。
3. 当系统调用你的 **onServiceConnected()** 回调方法时，你可以开始向service 发送请求，使用定义在Binder 的继承类中的接口。
4. 通过调用 **unbindService()** 方法断开连接。  
   当你的客户端被销毁的时候，它会与service解绑，但是你应该总是在你完成了与service的互相操作或者当你的activity暂停的时候与service解绑，这样service可以在不再使用的时候关闭掉。（合适的绑定和解绑的时间我们在下面讨论。）

这里有一些关于绑定到一个service的非常重要的**注意事项：**

* 你应该总是捕获 **DeadObjectException** 这个异常，当一个连接断开时会抛出这个异常。这是远程方法会抛出的唯一一个异常。
* Objects are reference counted across processes.（个人理解：在进程间，对象通过引用计数的方式避免一个对象被回收）
* 你应该总是在客户端生命周期的启动和销毁阶段成对的使用 binding 和 unbinding。例如：  
  1.如果你仅仅需要在你的activity对用户可见时与service 交互，你应该在 **onStart()** 的时候绑定，并且在 **onStop()** 的时候解绑。  
  2.如果你想让你的activity在停止到后台的时候也能去接收响应，这个时候你可以在 **onCreate()** 方法中绑定并且在 **onDestroy()** 中解绑。注意这在暗示你的activity在整个运行时间里（即使运行在后台）都需要使用service ，所以如果那个service是在另一个进程（process）中,那么你会增加这个进程变得被操作系统杀死的可能性。

**注意：**

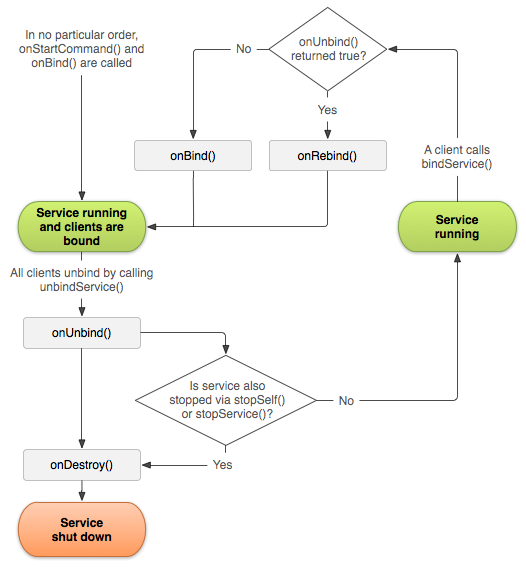
你应该不要在你的activity的**onResume()** 和 **onPause()** 中绑定和解绑一个service，因为这两个生命周期方法在每一个生命周期转变中都会调用。

1. 管理绑定服务的生命周期

当所有的客户端都从一个service解绑时，Android操作系统会销毁它（除非这个服务是通过 **onStartCommand()** 方法启动的）。这样一来，你不需要管理你的service的生命周期如果它仅仅是一个bound service： Android操作系统通过判断是否有客户端绑定到这个service来为你管理它。

然而，如果你选择实现 onStartCommand() 回调方法，这是你必须显式停止这个service，因为这个service被认为是启动的。在这种情况下，这个服务一直运行直到服务通过调用 **stopSelf()** 来停止自己，或者另一个组件调用了 **stopService()** ，不管是否有客户端绑定它。

此外，如果你的服务启动了并且接受绑定，当系统调用你的 onUnbind() 方法时，如果你想在下次客户端绑定到服务时接收到一个调用 **onRebind()** 的通知你可以返回 true ，**onRebind()** 方法返回值是空，但是客户端在 **onServiceConnected()** 回调中任然等够接收到 IBinder 实例。下图是service的生命周期：



1. 通知
2. 创建一个通知

你在 **NotificationCompat.Builder** 对象中为一个通知指定UI信息和actions。你调用 **NotificationCompat.Builder.build()** 来创建它自己，这个方法会返回一个包含了你指定的信息的**Notification**对象。为了显示一个通知，你通过调用**NotificationManager.notify()** 将这个**Notification**对象传递给系统。

1. 一个通知必需的内容

一个 Notification 对象必须包含以下内容：

1. 一个小图标，通过 setSmallIcon() 设定。
2. 一个标题，通过 setContentTitle() 设定。
3. 详情文本，通过 setContentText() 设定。

其他所有通知对象的设定和内容都是可选的。可以参看**NotificationCompat.Builder** API接口文档。

1. 通知的actions

尽管通知的actions是可选项，但是你应该为你的通知至少添加一个action。一个action 允许用户直接从一个通知到一个你应用程序内的Activity，这个Activity可以使用户看到更多的信息或者做进一步的工作。

一个通知可以提供多个actions。你应该总是定义一个当用户点击通知的时候能够触发的action。通常这个action会打开一个你的应用程序内的Activity。你也可以为通知添加一个按钮来执行附带的actions，例如关闭一个闹钟或者直接回复一个文本消息。这个特性在Android4.1中已经存在。如果你使用额外的按钮，你必须使它们存在你的应用程序的一个Activity中。

在一个通知内部，action自身被一个 **PendingIntent** 定义，这个**PendingIntent** 包含一个用于打开你的应用程序内的Activity。为 **PendingIntent** 关联一个手势去调用合适的 **NotificationCompat.Builder** 方法。例如，当用户点击通知文本的时候你想打开一个Activity，你可以通过调用 **setContentIntent()** 添加**PendingIntent 。**

1. 通知的优先级

如果你希望，你可以设置一个 通知的 优先级。这个优先级作为一个如何显示通知的暗示。要设置一个通知的优先级，通过调用**NotificationCompat.Builder.setPriority()** 并且传递一个 **NotificationCompat** 优先级的常量值。这里有5种优先级水平，范围从 **PRIORITY\_MIN(-2)** 到 **PRIORITY\_MAX(2)**之间。如果不设置，优先级默认为 **PRIORITY\_DEFAULT(0)**;

1. 创建一个简单的通知

下面的片段阐明了如何创建一个简单的通知，当用户点击通知的时候会打开一个Activity。注意：下面的代码创建了一个 **TaskSatckBuilder** 对象，并且使用它为action创建一个PendingIntent。

NotificationCompat.Builder mBuilder =  
        new NotificationCompat.Builder(this)  
        .setSmallIcon(R.drawable.notification\_icon)  
        .setContentTitle("My notification")  
        .setContentText("Hello World!");  
// Creates an explicit intent for an Activity in your app  
Intent resultIntent = new Intent(this, ResultActivity.class);  
  
// The stack builder object will contain an artificial back stack for the  
// started Activity.  
// This ensures that navigating backward from the Activity leads out of  
// your application to the Home screen.  
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);  
// Adds the back stack for the Intent (but not the Intent itself)  
stackBuilder.addParentStack(ResultActivity.class);  
// Adds the Intent that starts the Activity to the top of the stack  
stackBuilder.addNextIntent(resultIntent);  
PendingIntent resultPendingIntent =  
        stackBuilder.getPendingIntent(  
            0,  
            PendingIntent.FLAG\_UPDATE\_CURRENT  
        );  
mBuilder.setContentIntent(resultPendingIntent);  
NotificationManager mNotificationManager =  
    (NotificationManager) getSystemService(Context.NOTIFICATION\_SERVICE);  
// mId allows you to update the notification later on.  
mNotificationManager.notify(mId, mBuilder.build());

1. 为通知应用一个可展开的布局

为了让你一个通知以一个可扩展的视图显示出来，首先用正常的视图选项创建一个 **NotificationCompat.Builder** 对象，然后调用**Builder.setStyle()** 并传递一个可扩展的布局作为它的参数。

请记住，可展开布局的通知在Android4.1之前是不存在的。在后面的内容我们会介绍如何处理对于Android4.1及之前的版本。

例如，下面的代码片段展示了如何使用一个可展开的布局修改之前的那个通知：

NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)  
    .setSmallIcon(R.drawable.notification\_icon)  
    .setContentTitle("Event tracker")  
    .setContentText("Events received")  
NotificationCompat.InboxStyle inboxStyle =  
        new NotificationCompat.InboxStyle();  
String[] events = new String[6];  
// Sets a title for the Inbox in expanded layout  
inboxStyle.setBigContentTitle("Event tracker details:");  
...  
// Moves events into the expanded layout  
for (int i=0; i < events.length; i++) {  
  
    inboxStyle.addLine(events[i]);  
}  
// Moves the expanded layout object into the notification object.  
mBuilder.setStyle(inBoxStyle);  
...  
// Issue the notification here.

1. 处理版本兼容性（Notification）

不是所有的通知特色功能只存在于个别版本，尽管设置他们的方法存在于支持包的类 **NotificationCompat.Builder** .例如：按钮，它依赖于仅仅出现在Android4.1以及更高版本的可展开的通知，因为可展开的通知本身就只存在与Android4.1以及更高的版本。

为了保证更好的兼容性，尽量使用类 **NotificationCompat** 及其子类，特别是 **NotificationCompat.Builder** 来创建一个通知。此外，当你实现一个通知时按照下面的步骤来做：

1. 为了给所有用户提供通知具有的所有功能，而不管他们使用的是哪一个版本。为了达到这样的效果，要确保所有的功能存在于你的应用程序的一个Activity中。你可能需要添加一个 Activity 来做这件事。  
   例如，如果你想使用 addAction() 来提供控制后台播放音乐的开始和停止，首先在你应用程序的Activity中实现这个控制。
2. 确保所有的用户可以在用户点击通知时打开的那个Activity中使用这些功能，为了实现这个，创建一个 PendingIntent 指向一个Activity。通过调用 setContentIntent() 来为通知增加一个 PendingIntent。
3. 现在为你的通知添加一个可展开的特色功能。请谨记：你在通知上添加的功能要同时存在于用户点击通知打开的那个Activity中。
4. 管理通知：

如果你需要多次显示一个通知，而且这个通知要处理同样的事件，你应该避免创建一个完整的新的通知。取而代之的是，你应该考虑更新之前的那个通知，不管是更改它的一些值还是向它上面添加东西，或者这两种操作都有。

例如：Gmail 通过增加未读消息的数量并且将每一个邮件的摘要添加到通知来通知用户新邮件到达。这被称作堆叠通知。关于这种通知的详细介绍会在后面的部分介绍。

1. 更新通知

创建一个通知以便来更新它，通过调用 **NotificationManager.notifiy()** 来显示通知并为它指定一个 ID。更新那个通知一旦你已经查看过它，并且使用之前使用过的那个同样的**id**来显示它。如果之前的那个通知任然可见，系统会更新通知的内容。如果之前的通知已经消失了，一个新的通知会被创建来取代之前的。

下面的代码片段展示了更新一个通知来反映发生了多少次事件。

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());  
...

1. 移除通知

一个通知会一直存在直到发生下面这些事：  
1.用户单独或者通过“清除所有”来清除通知（除非那个通知是不能被青春的）

2.用户点击了通知，并且在你创建一个通知时设置了 setAutoCancel() 属性。

3.你为一个指定的通知id调用**cancel()**。这个方法也删除了正在进行的通知。

4.你调用了**cancelAll()**,这个方法会删除之前显示的所有通知。

1. 当打开一个Activity时保留导航

当你通过通知打开一个Activity时，你必须保留用户期望的导航体验。当点击back键时，你应该将用户带到Home 界面，并且你应该显示那个Activity作为一个隔离的任务（task）。为了保留导航体验，你应该在一个新的任务（task）中显示Activity。如何在一个新的任务（task）中启动一个 PendingIntent取决于你要启动的Activity的性质。这里有两个一般的情况：

* 一般Activity  
  你打开的Activity是你的应用程序正常工作流程的一部分。在这种情况下，在一个新的任务（task）中启动 **PendingIntent**，并且给**PendingIntent** 提供一个回退栈用来复制应用程序的常规的返回操作。  
  Gmail 发出的通知就展示出了这种情况。当你为了单条邮件消息点击一个通知时，你会看到消息内容。触摸返回键会不经过Gmail而直接将你带到Home 界面，就像如果你从主界面进入Gmail而不是从一个通知进入。  
  不管你点击通知的时候你处在哪个应用程序这都会发生。例如：如果你在Gmail中写一封邮件，并且你为了看单个邮件点击一个通知时，你会径直到达哪个邮件显示界面。触摸返回会先到达收件箱紧接着是Home 界面，而不是到达你正在编辑的地方。
* 特殊的Activity  
  如果使用通知进入的话用户仅仅看到进入的这个Activity。有一种场景，就是一个Activity提供那些一个通知很难展示的信息。在这种情况下，在一个新的任务中启动一个PendingIntent。没有必要创建一个回退栈，因为启动的那个Activity并不是应用程序Activity的一部分。点击返回仍然会将用户带到Home 界面。

1. 建立一个一般的PendingIntent

建立一个直接打开Activity的PendingIntent，要按照下面的步骤：

1. 在manifest文件中定义你的应用程序的Activity层级  
   a.为Android4.0.3及更早版本添加支持。为此，通过添加**<meta-data>**元素作为**<activity>**的子元素指定你要打开的Activity的父Activity。  
   对于这个元素，设置 android:name=”android.support.PARENT\_ACTIVITY”.设置 android:value=”<parent\_activity\_name>” , <parent\_activity\_name>是<activity>子元素android:name 的值。  
   b.也要为Android4.1及之后的版本添加支持。为此，为你要打开的Activity的<activity>元素添加 android:parentActivityName 属性。  
   最终的xml文件是这样的：

<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>

1. 基于要启动的Activity的**intent**创建一个回退栈  
   a.创建一个Intent来启动Activity。  
   b.通过调用**TaskStackBuilder.create()** 来创建一个栈。  
   c.通过调用 **addParentStack()** 将回退栈添加到之前创建的那个栈中。对于任意一个你在manifest中定义的Activity，这个回退栈包含一个 **Intent** 对象来开始这个Activity。这种方法仍然添加了一个在新任务中打开这个栈的标志标志。

注意：尽管**addParentStack()** 的参数是一个要开启Activity的引用，但是这个方法并没有添加要打开的Activity的Intent。反而，你要注意下面的这一步。  
d.通过调用 addNextIntent() 添加从通知进入的那个Activity的Intent，将这个Intent 作为参数传递给addNextIntent()。  
e.如果需要的话，通过调用 TaskStackBuilder.editIntentAt() 将参数Intent添加栈中。有些时候当用户利用返回键导航到这个Activity的时候要确保目标Activit显示有意义的数据。

f.通过调用 getPendingIntent() 来从这个回退栈中得到PendingIntent。稍后你可以利用这个PendingIntent作为方法 setContentIntent() 的参数。

例如：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());

1. 建立一个特殊Activity的PendingIntent

下面这一节描述如何建立一个特殊Activity的PendingIntent

一个特殊Activity的PendingIntent不需要使用回退栈，所以你不需要在manifest文件中定义Activity的层级结构，并且也不需要调用 addParentStack() 来构建一个回退栈。取而代之的是，使用manifest文件建立Activity的任务选项，并且通过调用 getActivity()创建PendingIntent。

1.在你的manifest文件中，为Activity的<activity>元素添加以下属性：  
**android:name** = “activityclass”   
 Activity的全类名  
**android:taskAffinity**=””  
 结合你设置在代码中的标志 **FLAG\_ACTIVITY\_NEW\_TASK** ，这会确保这个Activity不会进入到应用程序默认的任务。一些已经存在的与应用程序默认任务相同的任务不会受影响。

**android:excludeFromRecents**=”true”

从最近你的任务中排除新的任务，这样一来用户就不会无意中导航到它。

下面的代码片段显示了这个元素：

<activity  
    android:name=".ResultActivity"  
...  
    android:launchMode="singleTask"  
    android:taskAffinity=""  
    android:excludeFromRecents="true">  
</activity>

2.构建并显示通知  
 a.创建一个启动**Activity**的**Intent**。

b.通过使用标志**FLAG\_ACTIVITY\_NEW\_TASK**和**FLAG\_ACTIVITY\_CLEAR\_TASK** 调用 **setFlags()** 方法设置**Activity**在一个新的，空的任务中启动。

c.为Intent设置一些其他的选项。

d.通过调用 **getActivity()** 从一个**Intent**创建一个**PendingIntent**。你可以使用这个**PendingIntent** 作为方法 **setContentIntent()** 的参数。

下面的代码片段展示了这个过程：

// Instantiate a Builder object.  
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);  
// Creates an Intent for the Activity  
Intent notifyIntent =  
        new Intent(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 notifyPendingIntent =  
        PendingIntent.getActivity(  
        this,  
        0,  
        notifyIntent,  
        PendingIntent.FLAG\_UPDATE\_CURRENT  
);  
  
// Puts the PendingIntent into the notification builder  
builder.setContentIntent(notifyPendingIntent);  
// 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());

1. 在通知中显示进度条

通知可以包含一个动态的进度指示器用来告诉用户一个正在进行的操作的状态。如果你可以估计一个操作要花多长时间以及在任何时刻完成了多少，使用这个确定关系构成指示器。如果你不能估计一个操作要多长时间，使用不确定关系构成一个活动的指示器（an activity indicator）。

进度指示器以类 **ProgressBar** 的实现为展示平台。

在平台上使用一个进度指示器是从Android4.0开始的，通过调用 **setProgress()** 方法。对于之前的版本，你必须创建一个你自己自定义的包含 **ProgressBar** 视图的通知布局。

下面的章节描述了如何通过使用**setProgress()**  在通知中显示一个进度条

1. 显示一个固定持续时间的进度指示器

为了显示一个确定的进度条，通多调用 **setProgress(max,progress,false)** 方法将进度条添加到你的通知里,之后显示这个进度条。在你的操作进行的时候，增加 progress 的值，并且更新那个通知。在操作完成时，progress 应该与 max 相等。一个常用的方法是将 max 设置为100，之后增加progress 作为这个操作的 完成百分比。

当操作执行完成时你既可以让进度条显示也可以移除它。不管是哪一种情况，记得更新通知的文本告诉用户这个操作已经完成。至于如何移除一个进度条，通过调用 **setProgress(0,0,false)**。例如：

...  
mNotifyManager =  
        (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.  
                    mNotifyManager.notify(0, 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);  
            mNotifyManager.notify(ID, mBuilder.build());  
        }  
    }  
// Starts the thread by calling the run() method in its Runnable  
).start();

1. 显示一个持续的活动的指示器

显示一个不确定的活动指示器，调用**setProgress(0,0,true)**（头两个参数会被忽略）将他添加到你的通知里，并且显示这个通知。结果是一个拥有和进度条同样样式的指示器，只不过这个指示器有一个动画在执行。

在操作的开始显示通知。那个动画会一直运行直到你改变你的通知。当操作完成时，调用 setProgress(0,0,false)并且更新那个通知来移除活动的指示器。记得总是这样做，否则那个动画会一直执行即使操作已经完成。也记住要改变通知的文本来暗示用户操作已经完成了。

来看活动的指示器是如何工作的，跳到前面的代码片段并替换成下面的代码：

 // Sets an activity indicator for an operation of indeterminate length  
mBuilder.setProgress(0, 0, true);  
// Issues the notification  
mNotifyManager.notify(0, mBuilder.build());

1. 通知的元数据

通知可以根据你调用 **NotificationCompat.Builder**的以下方法为它指定的元数据来被排序：

* **setCategory()** 告诉系统当设备是优先策略模式时如何处理你的通知（比如，如果你的通知代表一个来电，消息或者闹钟）
* **setPriority()** 当通知的优先属性设置为 PRIORITY\_MAX 或者 PRIORITY\_HIGH时如果通知拥有声音或者震动会使通知以一个浮动窗口显示出来。
* **addPerson()** 允许你给通知添加一个人员列表。你的APP可以使用这个给系统发出应该将通知根据指定的人分组或者排序的信号。

1. 灵活的通知

在Android5.0中，当设备处在活动状态时（指设备没有锁住并且屏幕在打开状态）通知可以以一个小浮动窗口的形式显示。

1. 用户的Activity处在全屏模式
2. 这个通知有高的优先级并且使用了声音或者震动。
3. 锁屏通知

随着Android5.0的发布，通知可以显示在一个锁屏界面。你的应用程序可以利用这个功能提供媒体播放控制以及其它的行为。用户可以通过设置是否在锁屏时显示通知，你也可以指定你的应用程序发出的通知是否可以锁屏显示。

1. 设置可见

你的应用程序可以控制通知在锁屏显示的时候的详情的可见级别，你可以调用 setVisibility() 并且指定下面值中的一个：

* VISIBILITY\_PUBLIC 显示通知的所有内容。
* VISIBILITY\_SECRET 在锁屏时不显示通知的任何部分。
* VISIBILITY\_PRIVATE 显示基本的信息，比如通知的图标和标题，但是隐藏通知的所有内容。

1. 在锁屏时控制媒体播放

在Android5.0上锁屏界面不在基于 **RemoteControlClient** 来显示媒体控制，那种方式已经弃用了。取而代之的是使用 **Notification.MediaStyle** 模板结合 **addAction()** 方法，方法会将**actions**转为可点击的图标。

注意：那个模板和**addAction()** 方法在支持包中不存在，所以这个特性只能运行在Android5.0以及更高版本。

为了在Android5.0的锁屏界面显示媒体控制，就行上面描述的那样设置可见性为 **VISIBILITY\_PUBLIC** 。然后添加**actions**并且设置**Notification.MediaStyle** 模板，如下面的代码片段：

Notification notification = new Notification.Builder(context)  
    // Show controls on lock screen even when user hides sensitive content.  
    .setVisibility(Notification.VISIBILITY\_PUBLIC)  
    .setSmallIcon(R.drawable.ic\_stat\_player)  
    // Add media control buttons that invoke intents in your media service  
    .addAction(R.drawable.ic\_prev, "Previous", prevPendingIntent) // #0  
    .addAction(R.drawable.ic\_pause, "Pause", pausePendingIntent)  // #1  
    .addAction(R.drawable.ic\_next, "Next", nextPendingIntent)     // #2  
    // Apply the media style template  
    .setStyle(new Notification.MediaStyle()  
    .setShowActionsInCompactView(1 /\* #1: pause button \*/)  
    .setMediaSession(mMediaSession.getSessionToken())  
    .setContentTitle("Wonderful music")  
    .setContentText("My Awesome Band")  
    .setLargeIcon(albumArtBitmap)  
    .build();

1. 自定义通知的布局

通知框架允许你定义一个自定义的通知布局，通过一个 RemoteViews 来定义通知的样子。自定义布局的通知与普通的通知很相似，但是它们基于定义在一个xml布局文件的 RemoteViews 。

一个自定义通知的布局的看见高度取决于这个通知的视图。正常视图的布局被限制在 64dp，并且看展开的视图布局被限制在256dp。

为了定义一个自定义的通知布局，从实例化一个 RemoteViews对象开始。然后，调用 setContent() 而不是调用 setContentTitle()。要在自定义通知里设置内容详情，使用 RemoteViews里的方法来设置子视图的值。

1. 在一个单独文件里创建一个xml布局。你可以使用你希望的任何文件名，但是你必须使用 .xml 扩展名。
2. 在你的app里，使用 RemoteViews 的方法来定义你的通知的文本和图标。把这个 RemoteViews 对象通过setContent()放到你的 Notification.Builder 中。避免在你的 RemoteViews 上设置一个Drawable背景，因为你的文字颜色可能变得看不见。

这个 RemoteViews 类也包含了你可以容易的添加一个计时器或者进度条到你的通知的布局的方法。要得到更多的关于创建自定义布局的知识请查看 RemoteViews。

1. 为自定义通知文本使用样式资源

始终为自定义通知的文本使用一个样式资源。通知的背景颜色对于不同的版本和设备是大不相同的，使用样式资源可以帮助你解决它。从Android2.3开始，系统为标准通知布局的文本定义了一个样式。在Android2.3及更高版本中如果你在应用程序中使用同样的样式，你就可以确保你的文本相对于显示的背景色是可见的。

1. 在Android5.0中通知的新特色（Notifications New in Android5.0）

通知系统会保持通知用户有重要的、及时性的事件，比如一个朋友的新的聊天信息或者一个日历提醒。把通知看做是一个消息的通道来告诉用户重要的事情发生了，这会同步到他们所有的设备。

1. 在Android5.0中的新通知

在Android5.0中，通知在这些方面得到了重要的更新：结构上、视觉上以及功能上：

* 通知在视觉上与新的 material design 主题有了一致的改变
* 现在通知在设备锁屏界面可以显示，但同时敏感的内容仍然可以被影藏在后面。
* 当设备处于使用时收到高优先级的通知时使用一个新的格式，这种格式被称为 heads-up Notifications。
* 云同步通知：当你在你的一个Android设备上关闭一个通知的时候，在其他的设备上这个通知也会消失。

注意：这个Android版本上的通知的设计很大程度上违背了之前的Android版本。要获得更多的关于之前版本的通知的设计请看：在Android4.4及低版本的通知。

1. 剖析一个通知

这一节我们重温一个通知的基础部分，以及它们在不同种类的设备上怎样显示。

1. 基本布局

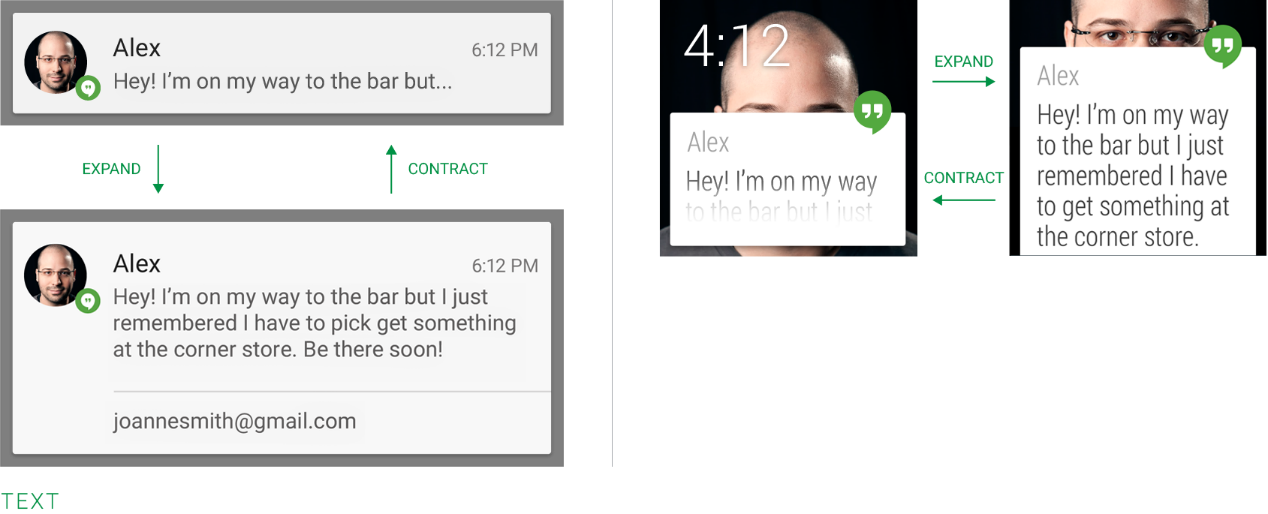
最基本的，所有的通知由一个基本的布局组成，包含：

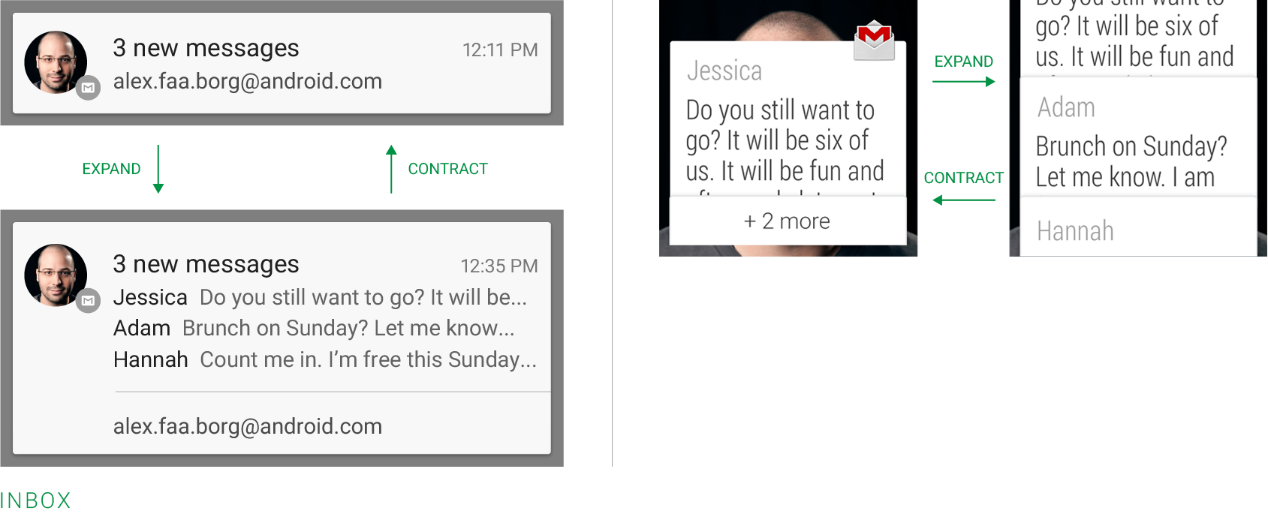
* 通知的图标。图标标志着发出通知的源app。如果一个app产生不止一种类型的通知的话它也可以潜在的暗示通知的类型。
* 一个通知标题和额外的文本。
* 一个时间戳。

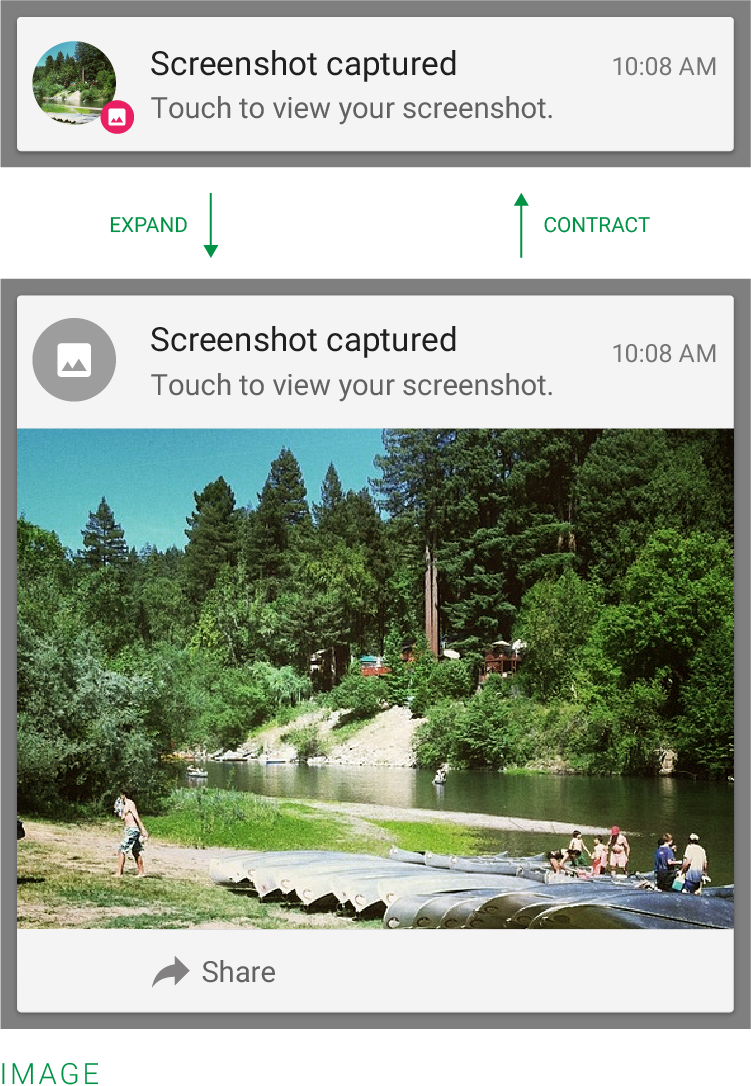
以前版本使用 Notification.Builder 创建的通知在Android5.0中仍然起作用，之前版本的通知系统仅仅给你提供了较少样式变化。

1. 可展开的布局

你可以选择你的应用程序的通知要提供多少细节。它可以显示消息的前几行或者显示一个大的预览图。额外的信息提供给用户更多的信息，并且在某些场景下可能允许用户去读取整条信息。用户可以双指缩放或者单指滑动来在收缩和扩展布局之间切换。对于单一事件的通知，Android为你提供了三种扩展的布局模板（文本，收件箱以及图片）用在你的应用程序中。下面的图片为你展示了单一事件的通知在手持设备以及穿戴设备上如何显示：







1. 通知的actions

对于那些显示在通知底部的控件，Android支持可选择性的为它们提供actions。借助actions，用户可以针对特殊的通知在通知内部处理很多常用的任务，而不需要打开源app应用。这加速了交互并且结合滑动消失的功能，帮助用户将注意力集中在对他们要紧的通知上。

对你的通知里有多少个actions要保持谨慎。你包含的actions越多，你创造的认识难度越多。通过只包含非常重要的并且有意义的actions来限制你的actions的最小数量。

对于通知的actions有以下比较好的候选者：

* 必要的、时常发生的以及你显示的内容类型是特有的。
* 允许用户快速的完成任务。

应该避免的actions是：

* 模糊不清的
* 跟通知的默认的action一样（比如 阅读 或者 打开）

你可以指定3个actions为最大限度，每一个action由一个图标和名称组成。将actions添加到基本布局使得这个通知是可展开的，即使这个通知没有展开的布局。在actions仅仅是为可展开的通知显示而其他的会隐藏的时候，确保任何一个用户可以从通知唤醒的actions是有效的。

1. Heads-up Notification

当一个高优先级的通知到达时，it is presented to users for a short period of time with an expanded layout exposing possible actions.

在这个期间之后，通知会退回到通知栏。如果一个通知的优先级时 High，Max,或者全屏，他会变成一个Heads-up 通知。

1. 让通知变得个性化的方法

对于从另一个人那里发过来的通知，通过方法 setLargeIcon() 来包含那个人的图片。也可以关联那个人的信息到通知的元数据中。

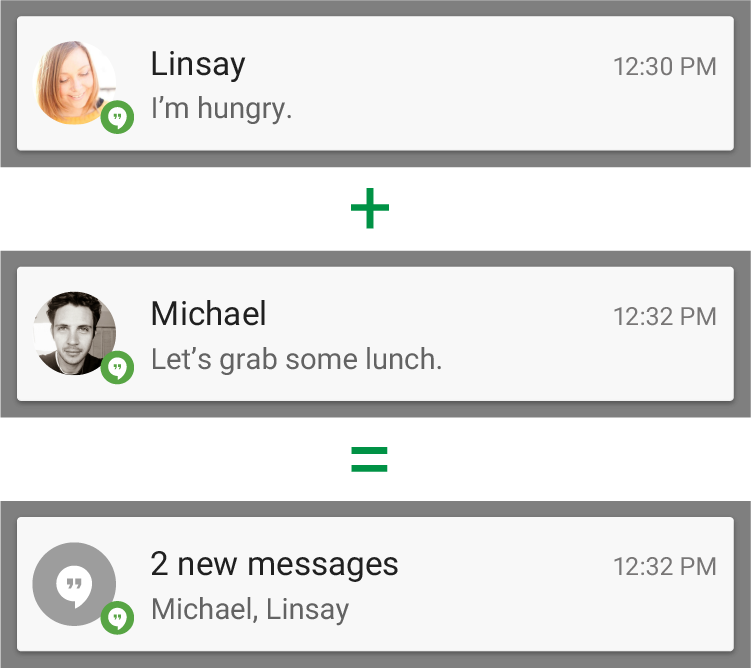
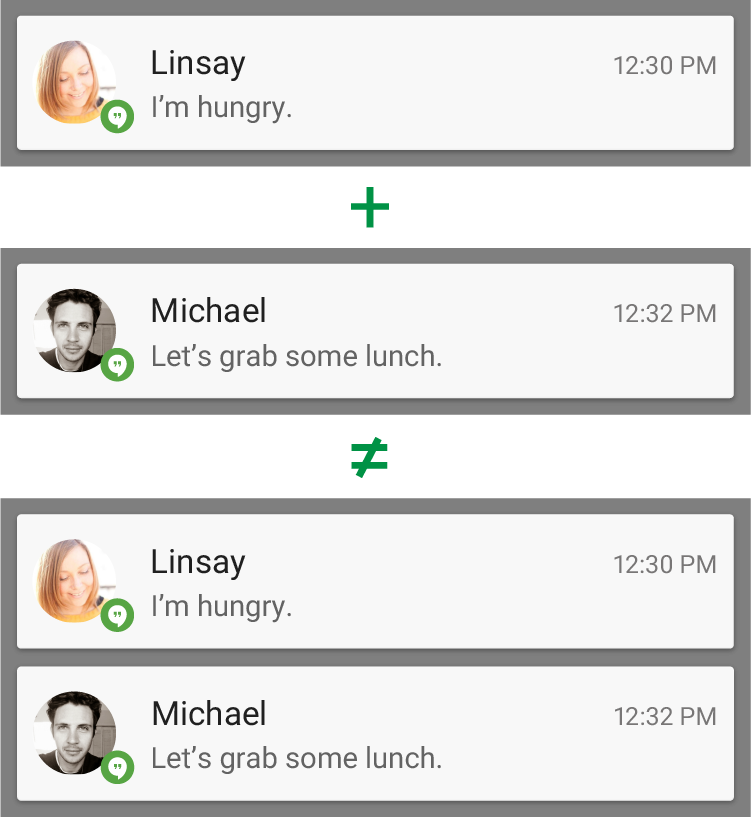
1. 为通知设置一个分类

如果你的通知属于预定义分类中的一个，相应的指派一个分类。系统UI会使用这个信息来划分级以及过滤策略。

|  |  |
| --- | --- |
| [CATEGORY\_CALL](file:///D:\reference\android\app\Notification.html#CATEGORY_CALL) | Incoming call (voice or video) or similar synchronous communication request |
| [CATEGORY\_MESSAGE](file:///D:\reference\android\app\Notification.html#CATEGORY_MESSAGE) | Incoming direct message (SMS, instant message, etc.) |
| [CATEGORY\_EMAIL](file:///D:\reference\android\app\Notification.html#CATEGORY_EMAIL) | Asynchronous bulk message (email) |
| [CATEGORY\_EVENT](file:///D:\reference\android\app\Notification.html#CATEGORY_EVENT) | Calendar event |
| [CATEGORY\_PROMO](file:///D:\reference\android\app\Notification.html#CATEGORY_PROMO) | Promotion or advertisement |
| [CATEGORY\_ALARM](file:///D:\reference\android\app\Notification.html#CATEGORY_ALARM) | Alarm or timer |
| [CATEGORY\_PROGRESS](file:///D:\reference\android\app\Notification.html#CATEGORY_PROGRESS) | Progress of a long-running background operation |
| [CATEGORY\_SOCIAL](file:///D:\reference\android\app\Notification.html#CATEGORY_SOCIAL) | Social network or sharing update |
| [CATEGORY\_ERROR](file:///D:\reference\android\app\Notification.html#CATEGORY_ERROR) | Error in background operation or authentication status |
| [CATEGORY\_TRANSPORT](file:///D:\reference\android\app\Notification.html#CATEGORY_TRANSPORT) | Media transport control for playback |
| [CATEGORY\_SYSTEM](file:///D:\reference\android\app\Notification.html#CATEGORY_SYSTEM) | System or device status update. Reserved for system use. |
| [CATEGORY\_SERVICE](file:///D:\reference\android\app\Notification.html#CATEGORY_SERVICE) | Indication of running background service |
| [CATEGORY\_RECOMMENDATION](file:///D:\reference\android\app\Notification.html#CATEGORY_RECOMMENDATION) | A specific, timely recommendation for a single thing. For example, a news app might want to recommend a news story it believes the user will want to read next. |
| [CATEGORY\_STATUS](file:///D:\reference\android\app\Notification.html#CATEGORY_STATUS) | Ongoing information about device or contextual status |

1. Summarize your notifications

如果一个已经确定类型的通知已经发出，当你的app将要发送一个同样类型的新通知时，那就把他们结合起来成为单个的通知。不要创建一个新的对象。



1. 是通知成为可选项Make notifications optional

通知应该总是在用户的控制之下。允许用户让你的app不能发送通知或者改变他们的警戒属性，比如说改变声音以及是否使用振动器，通过在你的应用程序的设置中添加一个通知设置选项。

1. 使通知LED灯合适的跳动

有些Android设备包含一个通知LED灯，用来在屏幕关闭的时候仍然能够通知用户。拥有 MAX,HIGH 或者 DEFAULT 优先级的通知应该使LED灯闪烁，当然那些低优先级的通知可以不用。

1. 正在进行的通知（Ongoing notifications）

正在进行的通知会一直通知用户一个正在执行的后台进程。例如：音乐播放器使用通知系统展示当前播放的进度，并且一直显示直到用户停止后台播放。正在进行的通知也可以显示一个长任务的反馈比如下载一个文件，或者编码一个视频。用户无法从通知抽屉移除一个正在进行的通知。

1. Material Design
2. 开始

用 material design创建一个app：

1. 预览 material design 说明书。
2. 给你的app应用material主题。
3. 使用material design的指导方针来创建你的布局。
4. 指定你的视图的 elevation （正面图）来投射阴影。
5. 为list和card使用系统部件。
6. 在你的app中自定义动画。
7. 支持向后兼容（Maintain backward compatibility）

你可以为你的app添加一些material design特色，并且保持兼容Android5.0之前的版本。

1. 使用material design更新你的app

更新一个已经存在的app为material design，要根据material design指导方针更新你的布局。要确保合并的深度，触摸反馈以及动画。

1. 应用Material 主题

应用Material 主题到你的app，要指定应用程序的样式继承自 android.Theme.Material:

<!-- res/values/styles.xml -->  
<resources>  
  <!-- your theme inherits from the material theme -->  
  <style name="AppTheme" parent="android:Theme.Material">  
    <!-- theme customizations -->  
  </style>  
</resources>

Material 主题提供了更新系统部件来使你设置他们的颜色和默认的触摸反馈动画以及Activity的过渡动画。

1. 设计你的布局

除了应用以及自定义material 主题外，你的布局应该符合material design直到方针。当你设计你的布局时，特别注意以下几点：

* Baseline grids
* Keylines
* Keylines
* Keylines
* Keylines

1. 在你的视图中指定Elevation

视图可以计算阴影，一个视图的elevation决定了它的阴影的大小以及绘制顺序。设置一个视图的elevation，在你的布局中使用 android:elevation 属性：

<TextView  
    android:id="@+id/my\_textview"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    android:text="@string/next"  
    android:background="@color/white"  
    android:elevation="5dp" />

视图的 translationZ 属性让你创建动画来反映在一个视图上的elevation的临时变化。

1. 创建列表和卡片

RecyclerView 是ListView的一个可插的版本，它支持不同的布局类型，并且在性能上有所改善。CardView 让你在卡片内部显示一条信息。下面的代码展示了如何在你的布局里一个包含CardView。

<android.support.v7.widget.CardView  
    android:id="@+id/card\_view"  
    android:layout\_width="200dp"  
    android:layout\_height="200dp"  
    card\_view:cardCornerRadius="3dp">  
    ...  
</android.support.v7.widget.CardView>

1. 自定义动画

Android5.0包含新的API来创建自定义的动画。例如：

public class MyActivity extends Activity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // enable transitions  
        getWindow().requestFeature(Window.FEATURE\_CONTENT\_TRANSITIONS);  
        setContentView(R.layout.activity\_my);  
    }  
  
    public void onSomeButtonClicked(View view) {  
        getWindow().setExitTransition(new Explode());  
        Intent intent = new Intent(this, MyOtherActivity.class);  
        startActivity(intent,  
                      ActivityOptions  
                          .makeSceneTransitionAnimation(this).toBundle());  
    }  
}

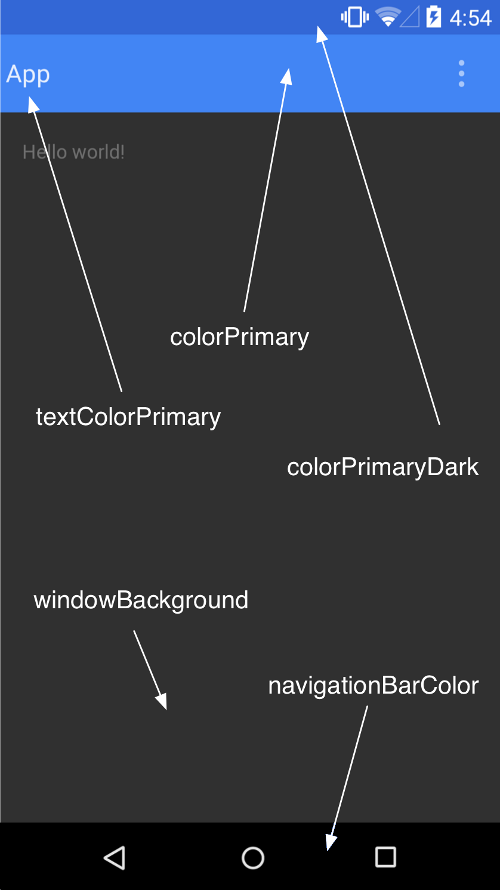
当你打开另一个Activity时，那个exit过渡就会激活。

1. 使用Material 主题

新的material 主题提供了：

1. 系统部件让你设置他们的颜色调色板。
2. 为系统部件提供触摸反馈动画。
3. Activity过渡动画。

你可以自定义material 主题的样子。你可以使用主题的属性给action bar 和 status bar(状态栏)着色。



系统部件有一个新的设计和触摸反馈动画。你可以自定义颜色调色板，触摸反馈动画以及activity过渡动画。

Material 主题定义如下：

* @android:style/Theme.Material(dark version)
* @android:style/Theme.Material.Light(light version)
* @android:style/Theme.Material.Light.DarkActionBar

注意：Material 主题仅仅存在于Android5.0及以上版本。V7 Support Libraries 为一些部件提供material design样式的主题以及支持自定义颜色调色板。

1. 自定义颜色调色板

自定义主题的基本颜色来适应你的商标，当你继承自material 主题时使用主题属性定义你的自定义颜色。

<resources>  
  <!-- inherit from the material theme -->  
  <style name="AppTheme" parent="android:Theme.Material">  
    <!-- Main theme colors -->  
    <!--   your app branding color for the app bar -->  
    <item name="android:colorPrimary">@color/primary</item>  
    <!--   darker variant for the status bar and contextual app bars -->  
    <item name="android:colorPrimaryDark">@color/primary\_dark</item>  
    <!--   theme UI controls like checkboxes and text fields -->  
    <item name="android:colorAccent">@color/accent</item>  
  </style>  
</resources>

1. 自定义状态栏

**Material** 主题让你可以轻松的自定义状态栏，所以你可以指定一个颜色来适应你的主题色，并提供足够的对比度来显示白色的状态图标。要为状态栏设置一个自定义的颜色，当你继承material主题时使用 **android:statusBarColor** 属性。默认情况下**android:statusBarColor** 继承自**android:colorPrimaryDark**.

你也可以在状态栏后面自己绘制。例如，如果你想在一个照片上面明亮的显示状态栏，

并使用一层稀薄的深色梯度变化的颜色来确保白色的状态图标清晰可见。如果这样做的话，设置**android:statusBarColor** 属性为 @android:color/transparent 并且按照要求调整窗口标志。你也可以为动画或者淡入淡出使用 Window.setStatusBarColor() 方法。

1. 创建列表和卡片（Creating Lists and Cards）

在你的app里使用material design 样式创建复杂的列表和卡片，你可以使用RecyclerView 和 CardView 控件。

1. 创建列表

RecyclerView控件是ListView的一个先进并且灵活的版本。这个控件是一个显示大量数据集合的容器，它通过维持有限个视图来提供高效的滚动。当你的数据集合的元素在运行时会基于网络事件和用户的行为而改变时可以使用RecyclerView控件。

RecyclerView类通过提供以下特性来简化以及处理大数据集合：

* 布局管理器来定位（Layout managers for positioning items）。
* 为公共条目操作提供默认的动画，比如移除或者添加条目。

你也可以为RecyclerView控件灵活的自定义布局管理器以及动画。



**Figure 1**. The RecyclerView widget.

使用 RecyclerView控件，你必须指定一个适配器和一个布局管理器。创建一个适配器，要继承 RecyclerView.Adapter 类。实现的细节取决于你的数据集的详情以及视图的类别。

一个布局管理器为RecyclerView里的视图定位，并且决定什么时候去重用那些对用户不再可见的条目视图。重用视图，布局管理器可能让适配器使用数据集合的另一个不同的元素的内容替换视图的内容。使用这种方式回收视图来提高性能，通过避免创建不必要的视图以及执行性能低下的 findViewById()查找操作。

RecyclerView 提供这些内置的布局管理器：

* LinearLayoutManager 以垂直或者水平滚动的方式显示条目。
* GridLayoutManager 以网格的形式显示条目。
* StaggeredGridLayoutManager 使用一个错开的网格显示条目。

要创建一个自定义的布局管理器，要继承 RecyclerView.LayoutManager 类。

1. 动画

默认的 RecyclerView 视图有添加或者删除条目的动画。要自定义这些动画，继承 RecyclerView.ItemAnimator 类并且使用 RecyclerView.setItemAnimator() 方法。

1. 例子

下面的代码展示了如何向一个布局文件添加 RecyclerView：

<android.support.v7.widget.RecyclerView  
    android:id="@+id/my\_recycler\_view"  
    android:scrollbars="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"/>

一旦你向你的布局文件添加了 RecyclerView控件，获取这个对象的句柄，把它与一个布局管理器连接，并且为要显示的数据绑定一个适配器：

public class MyActivity extends Activity {  
    private RecyclerView mRecyclerView;  
    private RecyclerView.Adapter mAdapter;  
    private RecyclerView.LayoutManager mLayoutManager;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.my\_activity);  
        mRecyclerView = (RecyclerView) findViewById(R.id.my\_recycler\_view);  
  
        // use this setting to improve performance if you know that changes  
        // in content do not change the layout size of the RecyclerView  
        mRecyclerView.setHasFixedSize(true);  
  
        // use a linear layout manager  
        mLayoutManager = new LinearLayoutManager(this);  
        mRecyclerView.setLayoutManager(mLayoutManager);  
  
        // specify an adapter (see also next example)  
        mAdapter = new MyAdapter(myDataset);  
        mRecyclerView.setAdapter(mAdapter);  
    }  
    ...  
}

1. 创建卡片

CardView继承 FrameLayout 类，让你在卡片内显示信息。CardView控件可以有阴影以及圆角。

创建一个有阴影的卡片，使用card\_view:cardElevation 属性。CardView使用真实的高度和动态的阴影在Android5.0及以上，and falls back to a programmatic shadow implementation on earlier versions. For more information, see Maintaining Compatibility.

使用这些属性来自定义CardView控件的外表显示：

* 在你的布局中设置边角半径，使用card\_view:cardConnerRadius 属性.
* 在你的代码中设置边角半径使用 CardView.setRadius 方法。
* 设置卡片的背景色，使用card\_view:cardBackgroundColor属性。

下面的代码展示了如何在你的布局中包含一个 CardView：

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    xmlns:card\_view="http://schemas.android.com/apk/res-auto"  
    ... >  
    <!-- A CardView that contains a TextView -->  
    <android.support.v7.widget.CardView  
        xmlns:card\_view="http://schemas.android.com/apk/res-auto"  
        android:id="@+id/card\_view"  
        android:layout\_gravity="center"  
        android:layout\_width="200dp"  
        android:layout\_height="200dp"  
        card\_view:cardCornerRadius="4dp">  
  
        <TextView  
            android:id="@+id/info\_text"  
            android:layout\_width="match\_parent"  
            android:layout\_height="match\_parent" />  
    </android.support.v7.widget.CardView>  
</LinearLayout>

1. 添加CardView和RecyclerView的引用

RecyclerView和CardView控件是v7 Support Libraries 的一部分。要在你的项目中使用这些控件，在你的app 的module中加入以下Gradle引用。

dependencies {  
    ...  
    compile 'com.android.support:cardview-v7:21.0.+'  
    compile 'com.android.support:recyclerview-v7:21.0.+'  
}

1. 定义阴影和Clipping Views

Material design 为UI 元素引入了elevation。Elevation 帮助用户理解每一个元素的重要性，让他们集中注意力于手头的任务。

一个视图的elevation由Z属性来表现，它决定了阴影的视觉效果。一个有高Z值的视图计算出大的，温和的阴影。一个有高Z值的视图会使一个拥有低Z值的视图封闭。当然，Z值并不影响一个视图的大小。

阴影是一个 elevated view的父视图绘制，and thus subject to standard view clipping, clipped by the parent by default.

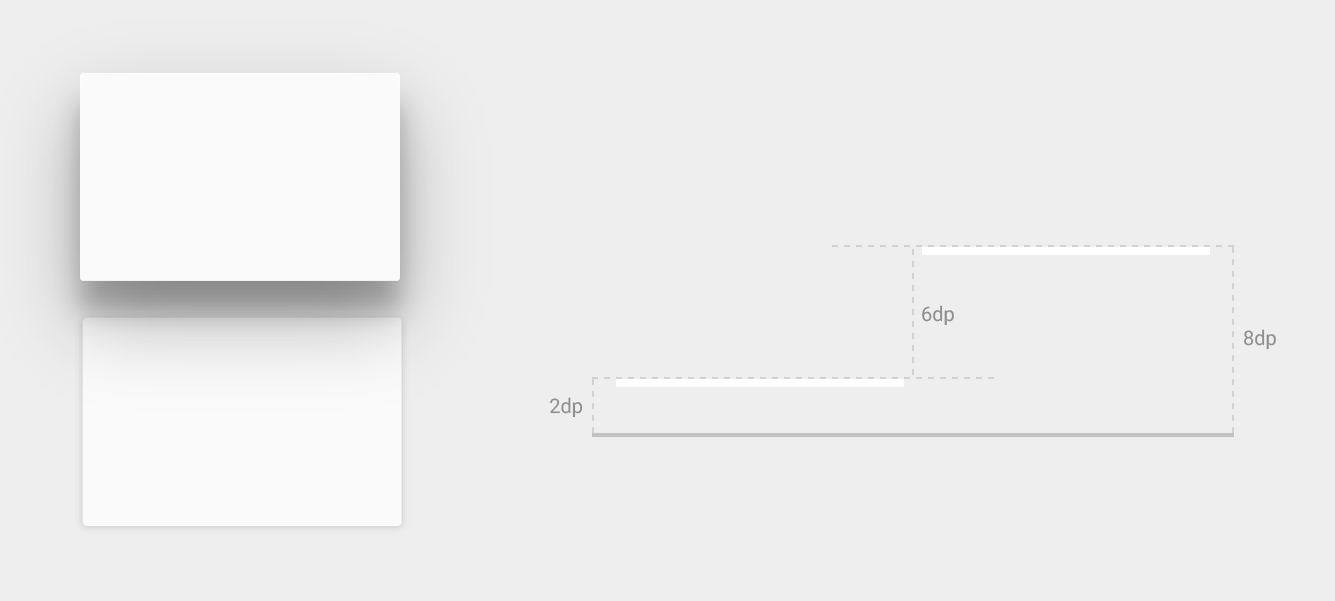
当执行一些操作时控件临时的上升到视图面板，Elevation对于创建这种动画是很有用的。

1. 为你的视图指定Elevation

为一个视图设置Z值，由两部分：

* Elevation：静态的部分
* Translation：动画时使用的动态部分

Z = elevation + translationZ



在布局中设置一个视图的elevation，使用android:elevation 属性。也可以在activity中使用 View.setElevation()方法。

设置一个视图的translation，使用View.setTranslationZ() 方法。

新的 ViewPropertyAnimator.z() 和 ViewPropertyAnimator.translationZ()方法允许你很简单的改变一个视图的elevation值。

你也可以使用 StateListAnimator 来指定这些这些动画。在那些状态改变而触发的动画的场景下，比如用户按下按钮，这很有用。

1. 自定义视图的阴影和轮廓

视图的可绘制背景的边界取决于它的阴影的默认的形状。轮廓代表一个图形对象的最外层形状并且定义了触摸反馈的产生涟漪的区域。

<TextView  
    android:id="@+id/myview"  
    ...  
    android:elevation="2dp"  
    android:background="@drawable/myrect" />

<!-- res/drawable/myrect.xml -->  
<shape xmlns:android="http://schemas.android.com/apk/res/android"  
       android:shape="rectangle">  
    <solid android:color="#42000000" />  
    <corners android:radius="5dp" />  
</shape>

从可绘制背景定义了是视图的轮廓时，视图开始计算一个拥有圆角的阴影，提供了通过重写视图默认的阴影形状来自定义的轮廓。

在你的代码里定义一个自定义的轮廓：

1. 继承 ViewOutlineProvider类。
2. 重写 getOutline() 方法。
3. 通过 View.setOutlineProvider()方法为你的视图指定新的轮廓。
4. Clip Views

Clip Views允许你很简单的改变一个视图的形状。你可以裁剪视图使得与其他设计的元素保持一致性，或者改变视图的形状来响应用户的输入。你可以使用 View.setClipToOutline()方法或者是 android:clipToOutline 属性裁剪你的视图。只有矩形，圆和圆角矩形轮廓支持裁剪，这些由 Outline.canClip() 方法来决定。

To clip a view to the shape of a drawable, set the drawable as the background of the view (as shown above) and call the View.setClipToOutline() method.

Clipping views is an expensive operation, so don't animate the shape you use to clip a view. To achieve this effect, use the Reveal Effect animation.

1. 高效显示位图（Displaying Bitmaps Efficiently）
2. 高效加载大的位图（Loading Large Bitmaps Efficiently）

图片有各种大小和形状。在某些情况下，它们比典型的应用程序界面还要大。例如，系统的 Gallery 应用程序显示那些用你的Android设备的照相机拍的照片，这些照片比你设备屏幕密度高得多。

分配给应用程序的内存是有限的，理想情况下你仅仅想加载低分辨的版本到内存。低分辨的版本应该匹配显示它的UI组件的大小。一个高分辨率的图片没有提供明显的益处，但是，但是仍然占用宝贵的内存资源，并且会因为敏捷的缩放引发额外的性能问题。

1. 读取位图的尺寸和类型

BitmapFactory 类提供几个编码方法（比如decodeByteArray(),decodeFile(),decodeResource()等等）根据各种资源来创建 Bitmap。基于你的图片数据源选择最合适的编码方法。这些方法尝试为构造好的位图分配内存，所以很容易造成 OutOfMemory 异常。每一种类型的编码方法有额外的签名来让你使用BitmapFactory.Options类来明确编码选项。设置 inJustDecodeBounds 属性为 true在编码时避免内存分配，此时bitmap对象返回的为 null，但是设置了outWidth，outHeight以及outMimeType。这个技术允许你读取图片数据的尺寸和类型先于构建位图和分配内存。

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 异常，在编码它之前检查位图的尺寸，除非你确认给你提供的图片资源大小合适并且适应可用的内存。

1. 加载一个按比例缩小的版本到内存

现在图片的尺寸大小我们知道了，可以用来决定是将整个图片加载进内存还是加载一个缩放版本到内存。

* 估算加载整个图片需要使用多少内存
* Amount of memory you are willing to commit to loading this image given any other memory requirements of your application.
* 要加载图片的目标ImageView的尺寸或者UI组件的尺寸
* 当前设备的屏幕大小和密度

例如，如果最终在ImageView中显示一个128X96缩略图就没有必要加载一个1024x768的图片到内存。

告诉编码器缩小这个图片，加载一个更小的版本到内存，在你的BitmapFactory.Options对象中设置 inSampleSize 为true。例如，一个分辨率为2048x1536的图片以 inSampleSize 为4编码生成一个大约为512x383的位图。加载这个到内存只用0.75M而不是12M。这里有一个方法根据目标宽度和高度来计算一个2的幂的缩放大小。

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) {  
  
        final int halfHeight = height / 2;  
        final int halfWidth = width / 2;  
  
        // Calculate the largest inSampleSize value that is a power of 2 and keeps both  
        // height and width larger than the requested height and width.  
        while ((halfHeight / inSampleSize) > reqHeight  
                && (halfWidth / inSampleSize) > reqWidth) {  
            inSampleSize \*= 2;  
        }  
    }  
  
    return inSampleSize;  
}

**Note:** A power of two value is calculated because the decoder uses a final value by rounding down to the nearest power of two, as per the [inSampleSize](file:///D:\Develop_Software\Android\sdk\docs\reference\android\graphics\BitmapFactory.Options.html#inSampleSize) documentation.

使用这个方法，首先将inJustDecodeBounds设置为true去编码，传递这个选项，然后使用新的inSampleSize值以及inJustDecodeBounds设置为false去再次编码：

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);  
}

这个方法使得加载一张位图到一个显示100x100缩略图的ImageView变得简单。

mImageView.setImageBitmap(  
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));

1. 不在UI线程中处理位图（Processing Bitmaps Off the UI Thread）

在高效加载大图中介绍的BitmapFactory.decode\* 方法，如果是从磁盘或者网络地址读取源数据的话就不应该在主线程中执行这些方法，读取这些数据的所耗费的时间取决于很多因素（比如读取磁盘或者网络的速度，图片的大小，CPU的主频等等）。如果这些任务中的某一个阻塞了UI线程，系统会标志你的应用为无响应并且用户有关闭它的选项。

这节课带你使用AsyncTask在后台线程处理位图并且告诉你如何处理并发问题。

1. 使用AsyncTask

AsyncTask类提供了一个简单的方式在后台线程去执行一些工作并将结果返回给主线程。要使用这个类，可以创建一个子类并且重写它提供的方法。这里有一个使用AsyncTask加载一个大图片到ImageView的例子：

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    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);  
            }  
        }  
    }  
}

ImageView 的弱引用（WeakReference）确保异步任务（AsyncTask）不会阻止ImageView以及它所引用的任何东西被垃圾收集。无法担保当任务结束时ImageView仍然存在，所以你必须在 onPostExecute() 中检查引用。那个ImageView可能已经不存在了，比如说如果用户导航到别的activity，或者配置在任务结束之前已经改变。

开始异步加载位图，简单的创建一个任务并执行它。

public void loadBitmap(int resId, ImageView imageView) {  
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);  
    task.execute(resId);  
}

1. 处理并发

像ListView和GridView这样的公共视图组件提出了当像上面的一节那样同时使用AsyncTask时会出现的问题。为了高效利用内存，当用户滑动时这些组件会回收子视图。如果每一个子视图触发一个 AsyncTask，无法担保当异步任务结束时，与异步任务关联的视图已经被回收给另一个子视图使用。此外，更无法保证异步任务开始的顺序与结束的顺序相同。

博客http://android-developers.blogspot.com/2010/07/multithreading-for-performance.html进一步讨论了如何处理并发，并且提供了一个解决方法：ImageView为最近的AsyncTask任务存储一个引用，稍后当任务结束时可以检查这个引用。Using a similar method, the AsyncTask from the previous section can be extended to follow a similar pattern.

创建一个专用的 Drawable 子类来为任务存储引用。这种情况下，用一个 BitmapDrawable作为占位图片可以先显示在 ImageView中直到任务结束。

static class AsyncDrawable extends BitmapDrawable {  
    private final WeakReference<BitmapWorkerTask> bitmapWorkerTaskReference;  
  
    public AsyncDrawable(Resources res, Bitmap bitmap,  
            BitmapWorkerTask bitmapWorkerTask) {  
        super(res, bitmap);  
        bitmapWorkerTaskReference =  
            new WeakReference<BitmapWorkerTask>(bitmapWorkerTask);  
    }  
  
    public BitmapWorkerTask getBitmapWorkerTask() {  
        return bitmapWorkerTaskReference.get();  
    }  
}

在执行 BitmapWorkerTask之前，你创建一个 AsyncDrawable 并且绑定它到一个目标 Imagev：

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);  
    }  
}

calcelPotentialWork方法用来检查是否有另一个正在运行的任务已经被关联到了这个ImageView。如果是的话，它会通过调用cancel()尝试取消之前的任务。在很少的情况下，新任务的数据会与已经存在的任务相同and nothing further needs to happen.

public static boolean cancelPotentialWork(int data, ImageView imageView) {  
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);  
  
    if (bitmapWorkerTask != null) {  
        final int bitmapData = bitmapWorkerTask.data;  
        // If bitmapData is not yet set or it differs from the new data  
        if (bitmapData == 0 || 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;  
}

辅助方法getBitmapWorkerTask()，用来检索与特定ImageView关联度任务：

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;  
}

最后一步是更新BitmapWorkerTask中的 onPostExecute()方法，这样它可以检查任务是否已经取消并且当前任务是否与一个已经和ImageView关联的任务匹配。

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
  
    @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);  
            }  
        }  
    }  
}

这个实现现在适合用来使用在 ListView和GridView这样的复用他们的子视图的组件中。在你正常的设置一个图片到你的ImageView的地方简单的调用loadBitmap。比如在一个GridView的适配器的getView（）方法中。

1. 缓存位图

加载一张图片到你的用户界面中是很简单的，然而如果你需要加载同时加载大量图片集合，事情会变得非常复杂。在多数情况下（比如说使用ListView，GridView或者ViewPager），在屏幕上的图片总量加上即将滑动到屏幕上的图片本质上是无限的。

像这些组件通过在它们滑出屏幕的时候复用子视图来保持内存的低使用率。假设你不在持有任何长期存活的引用，那么垃圾收集器也会释放你加载过的位图。这所有都很好，但是为了保持一个流畅的并且快速加载的UI，你想避免当它们每次重新回到界面时频繁的处理这些图片。一个内存和磁盘缓存可以提供帮助，允许组件快速的重新加载处理过的图片。

接下来带你使用一个内存和磁盘图片缓存来提高你的UI的响应性和流畅度，当你的UI需要加载大量图片的时候。

1. 使用内存缓存（Use a Memory Cache）

一个内存缓存以占用宝贵的应用程序内存为代价提供了高速访问位图的途径。LruCache类非常适合做缓存图片的任务，在一个强引用的LinkedHashMap里持有最近被引用的对象，并且在缓存大小超过设计的大小之前删除最近最少使用的成员。

**注意：**在过去，一个受欢迎的内存缓存的实现是一个 **SoftReference（软引用）** 或者 **WeakReference（弱引用）**的位图缓存，然而它不在被推荐。从Android2.3 开始垃圾收集器对收集 软引用/弱引用 有了增强，而这会使它们变得无效。除此之外，在Android3.0之前，位图的备份数据存储在一个不会按照预期的规则释放的内存区域。可能造成应用程序暂时的超过分配给他的内存限制出现崩溃。

为了给LruCache选择一个合适的大小，一些因素应该考虑在内：

* How memory intensive is the rest of your activity and/or application?
* 一次有多少张图片同时出现在屏幕上。有多少图片需要存在来为再次出现在屏幕上做准备。
* 设备的屏幕大小以及密度。在内存中缓存同样数量的图片一个超高密度（xhdpi）的屏幕相比高密度（hdpi）的设备将会需要更大的内存。
* 位图的尺寸和配置是什么，每一张图片占用的内存又是多大。
* 图片存取的频率是多少？有些图片会不会比其他的访问的频率高？如果是这样，或许你可能想确保每一项总在内存，或者甚至对于不同组的位图有多个LruCache对象来管理。
* 你能权衡质量和数量吗？有时存储大量低质量的位图是很有用的，在另一个后台任务中加载高质量的版本。

没有一个指定的大小或者公式可以适应多有的应用程序，这就由你去分析你的使用并且想出一个合适的解决办法。缓存太小会没有益处，缓存太大会再次导致 java.lang.OutOfMemory 异常，并且留给你的应用程序很少的内存去运行。

这里有一个LruCache的例子：

private LruCache<String, Bitmap> mMemoryCache;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    // Get max available VM memory, exceeding this amount will throw an  
    // OutOfMemory exception. Stored in kilobytes as LruCache takes an  
    // int in its constructor.  
    final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);  
  
    // Use 1/8th of the available memory for this memory cache.  
    final int cacheSize = maxMemory / 8;  
  
    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {  
        @Override  
        protected int sizeOf(String key, Bitmap bitmap) {  
            // The cache size will be measured in kilobytes rather than  
            // number of items.  
            return bitmap.getByteCount() / 1024;  
        }  
    };  
    ...  
}  
  
public void addBitmapToMemoryCache(String key, Bitmap bitmap) {  
    if (getBitmapFromMemCache(key) == null) {  
        mMemoryCache.put(key, bitmap);  
    }  
}  
  
public Bitmap getBitmapFromMemCache(String key) {  
    return mMemoryCache.get(key);  
}

当加载一个位图到ImageView时，LruCache 会首先检查。如果存在条目，会立即使用它来更新ImageView，否则会引起后台线程处理这张图片：

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);  
    }  
}

BitmapWorkerTask 也需要更新来加载条目到内存缓存：

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
    // 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;  
    }  
    ...  
}

1. 使用存盘缓存（Use a Disk Cache）

内存缓存对于高速加载最近的视图的位图是非常有用的，然而你不能信赖图片一定存在缓存中。像GridView这样有大数据集合的组件很容易填满内存缓存。你的应用程序可能被另一个任务打断，比如说电话来了，当它在后台它可能被杀死并且内存缓存被销毁。一旦用户恢复，你的应用程序必须再次处理每一张图片。

磁盘缓存可以使用在这种情况下来存留处理过的位图并且当图片不在内存缓存的时候帮助减少加载时间。当然，从磁盘获取图片比从内存获取图片慢得多，在磁盘读取时间不可预知的时候应该在后台线程操作。

**注意：**一个 ContentProvider 可能是一个更适合存储缓存照片的地方，如果它们被频繁的访问，比如在一个图片画廊程序中。

这个类的代码使用了 DiskLruCache的实现。这里更新示例代码添加磁盘缓存，除了已经存在的内存缓存。

private DiskLruCache mDiskLruCache;  
private final Object mDiskCacheLock = new Object();  
private boolean mDiskCacheStarting = true;  
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  
    ...  
    // Initialize disk cache on background thread  
    File cacheDir = getDiskCacheDir(this, DISK\_CACHE\_SUBDIR);  
    new InitDiskCacheTask().execute(cacheDir);  
    ...  
}  
  
class InitDiskCacheTask extends AsyncTask<File, Void, Void> {  
    @Override  
    protected Void doInBackground(File... params) {  
        synchronized (mDiskCacheLock) {  
            File cacheDir = params[0];  
            mDiskLruCache = DiskLruCache.open(cacheDir, DISK\_CACHE\_SIZE);  
            mDiskCacheStarting = false; // Finished initialization  
            mDiskCacheLock.notifyAll(); // Wake any waiting threads  
        }  
        return null;  
    }  
}  
  
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {  
    ...  
    // 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(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  
    synchronized (mDiskCacheLock) {  
        if (mDiskLruCache != null && mDiskLruCache.get(key) == null) {  
            mDiskLruCache.put(key, bitmap);  
        }  
    }  
}  
  
public Bitmap getBitmapFromDiskCache(String key) {  
    synchronized (mDiskCacheLock) {  
        // Wait while disk cache is started from background thread  
        while (mDiskCacheStarting) {  
            try {  
                mDiskCacheLock.wait();  
            } catch (InterruptedException e) {}  
        }  
        if (mDiskLruCache != null) {  
            return mDiskLruCache.get(key);  
        }  
    }  
    return null;  
}  
  
// 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 getDiskCacheDir(Context context, String uniqueName) {  
    // Check if media is mounted or storage is built-in, if so, try and use external cache dir  
    // otherwise use internal cache dir  
    final String cachePath =  
            Environment.MEDIA\_MOUNTED.equals(Environment.getExternalStorageState()) ||  
                    !isExternalStorageRemovable() ? getExternalCacheDir(context).getPath() :  
                            context.getCacheDir().getPath();  
  
    return new File(cachePath + File.separator + uniqueName);  
}

内存缓存在主线程中检查而磁盘缓存在后台线程中检查。磁盘操作从来不应该在主线程中发生。当图片处理完毕，最终的位图应该同时添加到内存缓存和磁盘缓存。

1. 处理配置变化（Handle Configuration Changes）

运行时配置改变，比如说屏幕方向变化，会造成Android系统销毁正在运行的activity并且采用新的配置来重启它。当配置改变的时候你想避免必须再次处理你所有的图片来使用户有流畅且快速的体验。

幸运的是，你在之前的一节中有一个很好的内存缓存来缓存图片。缓存数据可以使用一个通过调用 setRetainInstance(true)保存的Fragment传给新的activity实例。在新的activity实例被创建之后，这个重新获取的Fragment会被重新绑定，并且你又可以访问已经存在的缓存对象了，允许图片被快速的获取到并且显示在ImageView中。

这里有一个使用Fragment在配置改变时重新获取LruCache对象的例子：

private LruCache<String, Bitmap> mMemoryCache;  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    RetainFragment retainFragment =  
            RetainFragment.findOrCreateRetainFragment(getFragmentManager());  
    mMemoryCache = retainFragment.mRetainedCache;  
    if (mMemoryCache == null) {  
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {  
            ... // Initialize cache here as usual  
        }  
        retainFragment.mRetainedCache = mMemoryCache;  
    }  
    ...  
}  
  
class RetainFragment extends Fragment {  
    private static final String TAG = "RetainFragment";  
    public LruCache<String, Bitmap> mRetainedCache;  
  
    public RetainFragment() {}  
  
    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {  
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);  
        if (fragment == null) {  
            fragment = new RetainFragment();  
            fm.beginTransaction().add(fragment, TAG).commit();  
        }  
        return fragment;  
    }  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        **setRetainInstance(true);**  
    }  
}

1. 管理存放位图的内存（Managing Bitmap Memory）

除了在 缓存位图 一节中描述的步骤外，那里有一些细节可以促进垃圾回收以及位图重用。推荐的策略取决于你的目标Android版本是哪一个。BitmapFun 这个列子包含了这一节要展示的如何设计你的app，让它在各个不同的Android版本中高效运行。

这里有Android管理位图内存的演变：

* 在Android2.2（API level 8）及以下，当发生垃圾收集时，你的应用程序的所有线程都会停止。这个延时会降低程序的性能。Android2.3 添加了并发垃圾收集，这意味着如果一个位图不在被引用将会很快被回收再利用。
* 在Android2.3.3 及以下，一个位图的备份像素数据存储在native memory中。它与位图本事是独立的，存在Dalvik heap（虚拟机堆内存中）。Native memory中的数据不会以预期的方式释放，可能造成一个应用程序超过分配给他的有限的内存并崩溃。对于Android3.0，这些像素数据与它关联的位图一起存在Dalvik head（虚拟机堆内存）中。

下面的章节描述了如何对不同的Android版本优化其位图内存管理。

1. 在Android2.3.3及以下如何管理内存（Manage Memory on Android 2.3.3 and Lower）

在Android2.3.3及以下版本，被推荐使用 recycle()。如果在你的app中显示了大量位图数据，你很有可能引起 OutOfMemoryError 错误。recycle()方法允许你的app尽可能的回收再利用内存。

**注意：**你应该使用recycle()仅仅在你确保位图不在使用的时候。如果你调用 recycle() 后尝试绘制这个位图，会出现错误："Canvas: trying to use a recycled bitmap".

下面的代码片段是调用 recycle() 的一个例子。它使用引用计数（在变量mDisplayRefCount和mCacheRefCount）来跟踪位图目前是否正在显示或者在缓存。当发送以下情况的时候代码会回收位图：

* mDisplayRefCount和mCacheRefCount 的引用计数都是0。
* 位图不为空，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();  
  }

1. 在Android3.0及更高版本中如何管理内存(Manage Memory on Android 3.0 and Higher)

Android3.0介绍了BitmapFactory.Options.inBitmap字段。如果这个属性被设置，带有Options对象decode方法在加载内容时会尝试重用一个已经存在的位图。这表示位图的内存重用了，这会提高性能，并同时删除分配的内存和隔离的内存。然而，至于如何使用 inBitmap 肯定会有限制条件。特别的，在Android4.4之前，只有大小相等的位图才支持。

1. 为之后使用保存位图（Save a bitmap for later use）

下面的代码片段展示了如果存储一个在之后可能使用位图。当app运行在Android3.0或者更高版本并且从LruCache中删除位图：

Set<SoftReference<Bitmap>> mReusableBitmaps;  
private LruCache<String, BitmapDrawable> mMemoryCache;  
  
// If you're running on Honeycomb or newer, create a  
// synchronized HashSet of references to reusable bitmaps.  
if (Utils.hasHoneycomb()) {  
    mReusableBitmaps =  
            Collections.synchronizedSet(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()));  
            }  
        }  
    }  
....  
}

1. 使用一个存在的位图（Use an existing bitmap）

在运行的app中，decoder方法会检查那里是否存在它们可用的位图。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);  
}

在下面的代码片段展示了在上面的代码片段中调用的 addInBitmapOptions() 方法。它查找一个已经存在的位图设置它的inBitmap。注意这个方法仅仅设置inBitmap的值如果他找到了合适的匹配的位图。

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()) {  
        synchronized (mReusableBitmaps) {  
            final Iterator<SoftReference<Bitmap>> iterator  
                    = mReusableBitmaps.iterator();  
            Bitmap item;  
  
            while (iterator.hasNext()) {  
                item = iterator.next().get();  
  
                if (null != item && item.isMutable()) {  
                    // Check to see it 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;  
}

Finally, this method determines whether a candidate bitmap satisfies the size criteria to be used for inBitmap:（最终，这个方法决定候选位图是否满足大小条件）

static boolean canUseForInBitmap(  
        Bitmap candidate, BitmapFactory.Options targetOptions) {  
  
    if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.KITKAT) {  
        // From Android 4.4 (KitKat) onward we can re-use if the byte size of  
        // the new bitmap is smaller than the reusable bitmap candidate  
        // allocation byte count.  
        int width = targetOptions.outWidth / targetOptions.inSampleSize;  
        int height = targetOptions.outHeight / targetOptions.inSampleSize;  
        int byteCount = width \* height \* getBytesPerPixel(candidate.getConfig());  
        return byteCount <= candidate.getAllocationByteCount();  
    }  
  
    // On earlier versions, the dimensions must match exactly and the inSampleSize must be 1  
    return candidate.getWidth() == targetOptions.outWidth  
            && candidate.getHeight() == targetOptions.outHeight  
            && targetOptions.inSampleSize == 1;  
}  
  
/\*\*  
 \* A helper function to return the byte usage per pixel of a bitmap based on its configuration.  
 \*/  
static int getBytesPerPixel(Config config) {  
    if (config == Config.ARGB\_8888) {  
        return 4;  
    } else if (config == Config.RGB\_565) {  
        return 2;  
    } else if (config == Config.ARGB\_4444) {  
        return 2;  
    } else if (config == Config.ALPHA\_8) {  
        return 1;  
    }  
    return 1;  
}

1. 用户界面最佳实践（Best Practices for User Interface）
2. 适应多种屏幕的设计（Designing for Multiple Screens）

Android支持多种屏幕的上百种设备，小到手机大到电视。因此，设计你的APP去兼容各种屏幕大小是非常重要的，这样就能让尽可能多的用户来使用它。

但是，兼容不同的设备类型是不够的。每一种屏幕大小都为用户界面提出了不同的可能性以及挑战。为了让用户真正的满意并且留下深刻的印象，你的应用程序必须支持多种屏幕：这必将使每一种屏幕配置都能得到最佳的用户体验。

这节课为你展示了如何实现一个为多种屏幕配置优化的用户界面。

例子：<http://developer.android.com/shareables/training/NewsReader.zip>

1. 支持不同屏幕大小

通过以下方式来支持不同大小的屏幕

* 确保你的布局可以适当的调整大小来适应屏幕
* 根据屏幕配置提供合适的UI布局
* 确保恰当的尺寸应用到恰当的屏幕上
* 提供恰当缩放了的位图

1. 创建自定义视图（Creating Custom Views）

Android框架有大量的与用户交互并且显示不同数据的View 类。但是有时你的app需要特殊的而内置视图又没有覆盖的视图。

1. 创建视图类

一个设计的很好的自定义view就像一个设计的很好的类。它用一个简单的用户界面封装了特定的功能集合，它高效的使用CPU和内存。除了成为一个设计的很好的类，一个自定义视图也应该：

* 遵循Android标准
* 使用Android XML 布局提供自定义的样式属性
* 发送可得到的事件
* 兼容Android不同平台

Android框架提供了基础的类集合以及XML标记来帮助你创建一个符合所有要求的视图。接下来，我们讨论如何使用Android框架创建一个视图类的核心功能。

**创建View的子类**

定义在Android框架中的所有视图类都继承自View。你的视图类也可以直接继承View类，或者你也可以继承一个已经存在的视图子类，比如Button。

为了让Android Developer Tools 与你的视图相互作用，你至少要提供一个带了Context和AttributeSet对象最为参数的构造器。这个构造器允许布局编辑器创建和编辑你的视图的一个实例：

class PieChart extends View {  
    public PieChart(Context context, AttributeSet attrs) {  
        super(context, attrs);

    }  
}

**定义自定义属性**

添加一个内置的View到你的用户界面，你在一个XML元素中指定它并且适用元素属性控制它的样式和行为。一个写的比较好的自定义view可以通过XML被添加和定义样式。在你的自定义view中允许这样做，你必须：

* 在<declare-styleable>资源节点中为你的视图定义自定义属性。、
* 在你的XML布局文件中指定属性的值。
* 运行时检索属性值。
* 为你的视图应用检索出来的值。

这一节讨论如何定义自定义属性并为它们赋值。下一节在运行时处理检索和应用这些值。

定义自定义属性，在你的资源中添加<declare-styleable>。通常把这些资源放在 res/values/attrs.xml 文件中。这里有一个attrs.xml的例子：

<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>

这段代码定义了两个自定义属性，showText和labelPosition，它们属于一个叫PieChart的样式实体。样式实体的名字按照约定应该和所对应的自定义视图的类名相同。尽管不用严格遵守这个约定，一些受欢迎的代码编辑器取决于这个命名约定来提供声明补全。

一旦你定义了自定义属性，你可在一个XML布局文件中就像使用内置属性一样使用它们。仅仅的差别在于你的自定义属性属于一个不同的命名空间。并不是属于http://schemas.android.com/apk/res/android 命名空间，而是属于 http://schemas.android.com/apk/res/[your package name]命名空间。例如，这里是如何使用为PieChart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>

注意：在Android5.0中变为：

xmlns:custom="http://schemas.android.com/apk/res-auto "

为了避免必须重复写很长的命名空间 URI，简单的用一个 xmlns 指令即可。这个指令指定了命名空间 http://schemas.android.com/apk/res/com.example.customviews 的别名为 custom。你可以为这个空间选取任意的别名。

注意添加自定义视图到布局文件时xml的节点名称。它是自定义视图类的全类名。如果你的类是一个内部类，你必须使用视图类的外部类来限制它。

**应用自定义属性**

当从一个XML布局文件创建一个视图时，在XML标签内的所有属性从resource bundle  中读取并且将它作为一个AttributeSet传递给视图的构造器。尽管直接从AttributeSet中读取值是可能的，但是这有以下弊端：

* Resource references within attribute values are not resolved
* Styles are not applied

取而代之，将AttributeSet传递给obtainStyledAttributes().这个方法返回一个已经被应用和样式化的值的 TypedArray 数组。Android资源编译器为了做了很多工作才使得调用obtainStyledAttributes()方法变得简单。在资源文件夹中的每一个<declare-styleable>,已经生成的R.java定义一个属性id的数组以及一个常量集合来定义每一个属性在数组中的索引。你从TypedArray中使用之前定义的常量来读取属性。这里是读取属性的代码：

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();  
   }  
}

注意：TypedArray对象是一个公共的资源，在使用完之后必须被回收。

**添加属性和事件**

属性是控制一个视图的行为和样式的一种功能强大的方式，但是它们只有在视图初始化的时候才能读取。要提供动态的行为，为每一个自定义属性成对的暴露属性的 get和set方法。例如下面的代码暴露了PieChart的showText属性方法：

public boolean isShowText() {  
   return mShowText;  
}  
  
public void setShowText(boolean showText) {  
   mShowText = showText;  
   invalidate();  
   requestLayout();  
}

注意：在setShowText中调用了 invalidate()和requestLayout()方法。这些调用对确保视图的行为是可靠的是非常重要的。你必须在你改变了视图的可能改变它的样子的属性后使它无效，这样系统知道它需要重绘。同样的，如果一个属性的改变可能影响视图的大小和形状时你需要请求一个新的布局。忘记这些方法的调用会造成一些很难找出的bug。

自定义视图应该也支持事件监听来处理重要事件。例如：PieChart暴露一个叫做OnCurrentItemChanged的自定义事件来通知监听器用户旋转饼图并聚焦到另一部分。

很容易忘记暴露属性和事件，特别是自定义视图的使用者仅仅是你自己的时候。

1. 时自定义绘制（Custom Drawing）

自定义视图最重要的部分就是它的显示。根据你的应用程序需要自定义绘制可能简单也可能很复杂。

**重写 onDraw()方法**

绘制自定义视图最重要的步骤就是重写 onDraw() 方法。onDraw()方法 的参数是一个可以用来绘制自身的Canvas对象。Canvas类定义了绘制文本，线，位图以及其他图像的方法。你可以在onDraw()中使用这些方法创建出你自定义的UI。

在你可以调用绘制方法之前，创建一个 Paint 对象很有必要。下一节我们讨论 Paint的更多细节。

**创建绘制对象**

Android.graphics框架将绘制分为两个部分：

* 绘制什么，由Canvas来处理
* 怎么绘制，由Paint来处理

例如，Canvas 提供一个方法来绘制一条线，Paint提供方法来定义线的颜色。Canvas有一个绘制矩形的方法，Paint定义了是使用一种颜色填充它还是让它空白。

所以，在你绘制任何东西之前，你需要创建一个或更多个Paint对象。PieChart例子在 init 方法中做这些事，在构造器中会调用init方法：

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));  
  
   ...

提前创建对象是很重要的优化。视图被频繁的重新绘制，并且一些绘制对象需要很复杂 的初始化。在你的onDraw()方法中创建绘制对象会降低性能并且使你的UI延迟出现。

**处理布局事件**

为了正确的绘制你的自定义视图，你需要知道它有多大。复杂的自定义视图经常需要执行多次布局计算来决定他们在屏幕上的大小和形状。你不应该在屏幕上假设你的视图的大小。即使只有一个app使用你的视图，这个app也需要处理不同的屏幕尺寸，多种屏幕密度以及横屏和竖屏模式。

尽管视图有多个方法处理测量，它们中的大多数都不需要重写。如果你的视图不需要特殊的控制它的大小，你仅仅需要重写一个方法：onSizeChanged()。

当你的视图**第一次指定大小的时候会调用onSizeChanged()方法**，并且如果你的视图**因为任何原因改变大小时会再次调用他**。在**onSizeChanged()中计算位置，尺寸以及其他与你的视图绑定的值**，而**不是在每次绘制的时候计算它们**。

当你的视图被指定大小的时候，布局管理器承担包括视图的所有padding的大小。当你计算你的视图的值时你必须处理padding值。这里告诉你在PieChart.onSizeChanged()中如何做：

       // 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);

如果你需要在你的视图布局参数基础上更好的控制它，实现 onMeasure() 方法。这个方法的参数 View.MeasureSpec 告诉你你的视图否父视图想让你的视图有多大，这个大小时硬性的最大值还是仅仅是一个建议。最佳的，这些值使用包装的整型值存储，你使用View.MeasureSpec的静态方法解析存在每一个整型里面的信息。

这里有一个实现了 onMeasure() 的例子。在这个实现中，PieChart尝试使它的区域足够大来使饼图与标签同样大：

@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, 0);  
  
   setMeasuredDimension(w, h);  
}

在这段代码中有三件很重要的事情需要注意：

* 计算时需要将视图的padding包含在内。就像之前提醒的那样，这是视图的职责。
* 辅助方法 resolveSizeAndState() 用来创建最终的宽度和高度值。辅助方法返回一个合适的View.MeasureSpec值通过比较视图要求的大小与传递给onMeasure()的参数。
* onMeasure()方法没有返回值。而是通过调用 setMeasureDimension() 方法来让结果生效。调用这个方法是强制的。如果你省略调用它，视图类会抛出一个运行时异常。

**绘制**

一旦你的对象创建并且测量代码定义完成，你可以实现onDraw()方法。每一个视图实现onDraw()是不同的，但是这里有一些公共的操作是大多数视图可以共用的：

* 绘制文本使用drawText()。调用 setTypeface()来指定字型，调用setColro()设置字的颜色。
* 使用drawRect()，drawOval()和drawArc()绘制简单形状。通过调用setStyle()来改变形状是否填充，是否有轮廓或者二者结合。
* 绘制复杂形状使用 Path 类。向一个Path对象加入直线和曲线来定义一个形状，它使用drawPath()绘制形状。就像基本形状，使用Path绘制的图形也可以使用setStyle()方法。
* 通过创建LinearGradient对象定义梯度填充物。在填充形状上调用setShader()来使用你的LinearGradient。
* 绘制位图使用drawBitmap()。

实例代码：

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);  
}

1. 多线程
2. 指定运行在线程中的代码

这几课告诉你如何实现一个Runnable类，用来在一个单独线程的Runnable.run()方法中执行代码。你也可以传递一个Runnable给另一个对象来关联一个线程并运行它。一个或多个Runnable对象执行一个特殊的操作我们称为任务。

Thread和Runnable是基本的类，只有有限的能力。然而，它们是Android中像HandlerThread，AsyncTask和IntentService这样功能强大的类的主要成分。它们也是ThreadPoolExecutor类的主要成分。这个类自动的管理线程和任务队列，并且可以并行的运行多线程。

1. 定义一个类来实现Runnable

实现一个实现Runnable的类很直接：

public class PhotoDecodeRunnable implements Runnable {  
    ...  
    @Override  
    public void run() {  
        /\*  
         \* Code you want to run on the thread goes here  
         \*/  
        ...  
    }  
    ...  
}

1. 实现run()方法

在类中，Runnable.run()方法包含要执行的代码。通常，任何东西都可以在Runnable中。记住，因为Runnable不会运行在UI线程中，所以无法直接改变UI对象，比如View对象。要与UI线程交互，你必须使用在后面会讲解的一种技术。

在run()方法的开始，通过调用Process.setThreadPriority()方法以及THREAD\_PRIORITY\_BACKGROUND 参数来设置线程使用后台优先级。这种方式减少Runnable对象和UI线程之间的资源竞争。

你也应该在Runnable自身中存储一个Runnable对象的Thread引用，通过调用Thread.currentThread()。

下面的代码片段告诉你如何建立run()方法：

class PhotoDecodeRunnable implements Runnable {  
...  
    /\*  
     \* Defines the code to run for this task.  
     \*/  
    @Override  
    public void run() {  
        // Moves the current Thread into the background  
        android.os.Process.setThreadPriority(android.os.Process.THREAD\_PRIORITY\_BACKGROUND);  
        ...  
        /\*  
         \* Stores the current Thread in the PhotoTask instance,  
         \* so that the instance  
         \* can interrupt the Thread.  
         \*/  
        mPhotoTask.setImageDecodeThread(Thread.currentThread());  
        ...  
    }  
...  
}

1. 为多线程创建一个管理器
2. 定义线程池类