Android基础

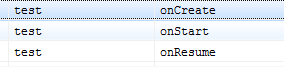
**1 Android应用程序的组成**

1. 活动：这是面向用户的组件，展示一个用户界面用于交互
2. 服务：这是工作在后台的进程，没有可见的用户界面。例如一个服务可以负责轮询邮件服务器上的新电子邮件
3. 内容提供程序（Content Provider）：该组件允许将应用程序的一部分数据用于其他应用程序
4. 意图（intent）：这是由系统或应用程序本身所创建的消息，它可以传递到任何对其感兴趣的一方
5. 广播接收者（boardcast receiver）：用于相应特定的意图，并可以执行一个动作

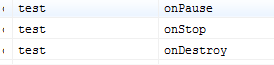
**2 Android应用程序的生命周期**



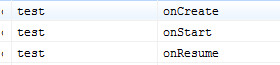
1 程序初始化运行（一个测试用例）：



2 Back键



3 Back键后重新进入



4 Home键



5 Home键后重新进入



6 在onCreate函数中调用finish函数

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

**this**.finish();

Log.*v*("test", "onCreate");

}



**从上面我们可以看到，即使Activity调用了finish方法，也不会立即调用onDestroy方法，而是执行完finish后面的代码后才会调用onDestroy方法**

理论分析：

一个Android程序的进程是何时被系统结束的呢？通俗地说，一个即将被系统关闭的程序是系统在内存不足，根据“重要性”选择的，重要性从高到低的原则是：

1. 前台进程：这样的进程拥有一个在屏幕上显示并和用户交互的Activity或者它的一个IntentReciver正在运行，这样的程序重要性最高，只有在系统内存非常低，万不得已的情况下才会结束；
2. 可见进程：在屏幕上显示，但是不在前台的进程，比如一个前台进程以对话框的形式显示在该进程前面
3. 服务进程：这样的进程在后台持续运行，比如后台音乐播放、后台数据上传下载等
4. 后台进程：这样的程序拥有一个用户不可见的Activity，这样的程序在系统内存不足时，按照LRU的顺序结束
5. 空进程：这样的进程不包含任何活动的程序部件，系统随时可以关闭这些进程

一个活动有以下三种状态：

1. 激活状态或运行状态：这时它运行在屏幕的前台（处于当前任务活动栈的最上面），这个活动有用户的操作焦点
2. 暂停状态：这时活动失去焦点但是它对用户仍可见，也就是说，另一个活动在它的上面且那个活动是透明的或者没有覆盖整个屏幕，因此用过它可以看见暂停状态的活动，一个暂停的活动完全是活着的（它维护者所有的状态和成员信息，且仍然依附在窗口管理器），但是当内存极低时可以被系统杀掉
3. 停止状态：这时活动完全被其他活动掩盖，它仍然保留所有状态和成员信息，但是对用户它不可见，因此它的窗口隐藏时且当其他地方需要内存时它往往被系统杀掉

**3 Android的Manifest文件简单分析**

**<manifest>标记是AndroidManifest文件的根元素**，下面是一个基本的示例：

<manifest xmlns:android=*"http://schemas.android.com/apk/res/android"*

package=*"com.doodle.test"*

android:versionCode=*"1"*

android:versionName=*"1.0"* >

xmlns：XML Namespaces的缩写，指定了一个名为android的命名空间

@在Android中表示引用

**<application>标签**

<application

android:icon=*"@drawable/ic\_launcher"*

android:label=*"@string/activity\_name"*

android:debuggable=*"true"*>

<activity

android:name=*".TestAndroidLifeCircleActivity"*

android:label=*"@string/activity\_name"* >

<intent-filter>

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

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

</intent-filter>

</activity>

</application>

Android:icon标签是表示应用程序的图标

同时我们发现再<application>标签与<Activity>标签中均有android:label标签

**android:label标签特别说明（已经验证）：**

activity标签中的android:label标签的优先级高于application，也就是说两者都设计这个标签的话，activity的值覆盖application

1. application设置了此标签，其他activity没有设置的情况下，应用程序在桌面上的名字和所有activity的title都是这个设置的标签
2. application设置了此标签，主activity中也设置了此标签，则应用程序名和主activity的title都是主activity中设置的标签，其他非主activity的title如果没有自己设置此标签，还是使用application中设置的标签，如果有其他非activity也设置了此标签，则其它title就是自己设置的这个标签

**<Activity>标签**

<activity

android:name=*".TestAndroidLifeCircleActivity"*

android:label=*"@string/activity\_name"*

android:screenOrientation=*"portrait"*

android:configChanges=*"keyboard|keyboardHidden|orientation"*>

<intent-filter>

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

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

</intent-filter>

</activity>

**android:name：**用于指定活动类的名

**android:label：**指定Activity的标题

**android:screenOrientation：**指定活动使用的方向，portrait（纵向）与landscape（横向）

一个小测试例子：

在没有使用screenOrientation属性的时候，进行屏幕旋转：



也就是说，如果我们不设置该属性，那么活动将会使用设备的当前方向，它取决于加速计的数据，这就意味着当设备的方向发生变化时，活动将会被销毁并重新启动

**android:configChanges：**调整设备方向或滑出键盘可认为是一种配置更改，在这样一种更改下，android系统将会销毁并重启应用程序以适应这种变化，这在游戏中就很糟糕了，这时候configChanges就派上用场了，它允许我们制定某种想处理的配置更改，而不是销毁并重启应用程序

**<Intent-filter>标签**

Android系统上的应用程序没有单一主入口点的概念，相反，我们可以在意图和服务中制定多个入口点，这些入口点可通过系统或第三方应用程序所发出的特定意图来启动。我们需要通过某种方式与android系统进行通信，并通过应用程序的活动和服务来响应特定的意图，这里正是<Intent-filter>所起的作用

在上面的示例中，我们指定了两种意图过滤器：<action>与<category>，<action>元素告诉Android此活动是应用程序的主入口，<category>元素指定要把活动添加到应用程序启动器中。这两个元素结合使用，android就知道应该在应用程序启动器中按下应用程序图标时启用这个特定的活动

对于<action>和<category>这两个属性，只需要指定name属性，以便标致活动将相应的意图，android.intent.action.MAIN是一个特殊的意图，它可被android用来启动一个应用程序的主活动，而android.intent.category.LAUNCHER这个意图用于发告诉android系统，应用程序是否有一个特定的活动应该在应用程序启动器中有一个入口

**<uses-permission>标签**

Android有一个非常安全的模型，让每一个应用程序都运行在它自己的进程和VM中，有自己的用户和用户组，而不会影响其他应用程序，但是会限制使用系统资源，如SD卡、联网设备等，因此我们在使用这些资源时，就必须请求这些资源权限，这就是<uses-permission>元素所要做的事

一个权限的形式通常如下所示，其中string用户指定我们要请求的权限名称：

<uses-permission android:name=”string”>

常见的string有：

android:permission.RECORD\_AUDIO

android:permission.INTERNET

android:permission.WRITE\_EXTERNAL\_STORAGE等等

**<uses-feature>标签**

Android market应用程序会通过硬件配置文件过滤所有可用的应用程序，通过<uses-feature>元素，一个应用程序可指定其所需的硬件功能---如多点触摸或OpenGL ES2.0等，任何没有这些指定功能的设备将会被过滤，并使用户在一开始就无法看到这些应用程序，例如：

<uses-feature android:glEsVersion=*"0x00010001"* />

**4 Activity的四种跳转方式**

http://blog.csdn.net/luoshengyang/article/details/6714543

解开Android应用程序组件Activity的“singleTask”之谜

**5 Intent的四个主要属性**

http://wisekingokok.iteye.com/blog/1155396

**6 Android-sharedUserId数据权限**

http://wallage.blog.163.com/blog/static/17389624201011010539408/

**7 ListView的详解**

列表的显示需要三个元素：

1. ListView用来展示列表的View
2. 适配器：用来把数据映射到ListView上的中介
3. 数据：具体的将被映射的字符串、图片或者基本组件

根据列表的适配器类型，列表分为三种：ArrayAdapter、SimpleAdapter和SimpleCursorAdapter，其中以ArrayAdapter最为简单，只能展示一行字，SimpleAdapter有最好的扩充性，可以定义出各种效果，SimpleCursorAdapter可以认为是SimpleAdapter对数据库的简单结合，可以方便的把数据库的内容以列表的形式展示出来。

1）最简单的ListView开始（ArrayAdapter适配器）

listView = **new** ListView(**this**);

listView.setAdapter(**new** ArrayAdapter<String>(**this**, android.R.layout.*simple\_expandable\_list\_item\_1*, GetData()));

setContentView(listView);

**public** List<String> GetData(){

List<String> list = **new** ArrayList<String>();

list.add("测试用例1");

list.add("测试用例2");

list.add("测试用例3");

list.add("测试用例4");

**return** list;

}



2）SimpleCursorAdapter适配器

这个适配器简单地说就是方便把从游标得到的数据进行列表显示，并可以把指定的列映射到对应的TextView中。例如把电话簿中的联系人显示到类列表中：

listView = **new** ListView(**this**);

Cursor cursor = getContentResolver().query(android.provider.ContactsContract.Data.*CONTENT\_URI*, **null**, **null**, **null**, **null**);

startManagingCursor(cursor);

SimpleCursorAdapter simpleCursorAdapter = **new** SimpleCursorAdapter(**this**, android.R.layout.*simple\_expandable\_list\_item\_1*,cursor, **new** String[]{android.provider.ContactsContract.Data.*DISPLAY\_NAME*}, **new** **int**[]{android.R.id.*text1*});

listView.setAdapter(simpleCursorAdapter);

setContentView(listView);



分析：

Cursor获取一个纸箱系统通讯录数据库的Cursor对象获得数据来源

startManagingCursor(cursor);我们激昂获得的Cursor对象交由Activity管理，这样Activity的生命周期和Activity便能够自动同步，省去自己手动管理Cursor

SimpleCursorAdapter构造函数的前面三个参数和ArrayAdapter是一样的，最后两个参数：一个包含数据库的列的String型数组，一个包含布局文件中对应组件id的int型数组。其作用是自动的将String型数组所表示的每一列数据映射到布局文件对应id的组件上，上面的代码将NAME数据一次映射到布局文件的is为text1的组件上

3）SimpleAdapter适配器

SimpleAdapter的扩展性最好，可以定义各种各样的布局，可以放上ImageView，还可以放上Button、CheckBox等

ListView listView = **new** ListView(**this**);

SimpleAdapter simpleAdapter = **new** SimpleAdapter(**this**, GetSimpleAdapterData(), R.layout.*vlist*, **new** String[]{"title", "information", "image"}, **new** **int**[]{R.id.*title*, R.id.*infomation*,

R.id.*image*});

listView.setAdapter(simpleAdapter);

setContentView(listView);

获取数据源：

**public** List<Map<String, Object>> GetSimpleAdapterData(){

List<Map<String, Object>> listMap = **new** ArrayList<Map<String, Object>>();

Map<String, Object> map = **new** HashMap<String, Object>();

map.put("title", "g1");

map.put("information", "google1");

map.put("image", R.drawable.*first*);

listMap.add(map);

map = **new** HashMap<String, Object>();

map.put("title", "g2");

map.put("information", "google2");

map.put("image", R.drawable.*second*);

listMap.add(map);

map = **new** HashMap<String, Object>();

map.put("title", "g3");

map.put("information", "google3");

map.put("image", R.drawable.*third*);

listMap.add(map);

**return** listMap;

}

说明：

SimpleAdapter的构造函数如下：

SimpleAdapter(Context context, List<? Extends Map<String, ?>> data, int recource, String[] from. Int[] to);

其中resource参数是就是指定的listView的列表项的布局，data是数据源，因此，from参数Map对象的key，而to表示组件的id

Vlist.xml文件如下：

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

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:orientation=*"horizontal"* >

<ImageView android:id=*"@+id/image"*

android:layout\_width=*"50dp"*

android:layout\_height=*"50dp"*

android:contentDescription=*"@string/app\_name"*

android:layout\_gravity=*"center\_horizontal"*

android:layout\_margin=*"5dp"*/>

<LinearLayout android:orientation=*"vertical"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*>

<TextView android:id=*"@+id/title"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:textColor=*"#ffffffff"*

android:textSize=*"22dp"*></TextView>

<TextView android:id=*"@+id/infomation"*

android:layout\_width=*"wrap\_content"*

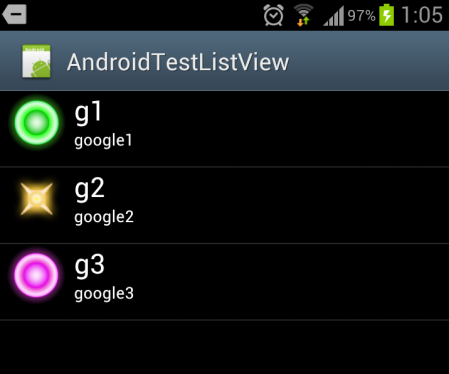
android:layout\_height=*"wrap\_content"*

android:textColor=*"#ffffffff"*

android:textSize=*"13dp"*></TextView>

</LinearLayout>

</LinearLayout>



**android:layout\_gravity 和 android:gravity 的区别**

从名字上可以看到，android:gravity是对元素本身说的，元素本身的文本显示在什么地方靠着换个属性设置，默认是在左侧的。android:layout\_gravity是相对与它的父元素说的，说明元素显示在父元素的什么位置。

比如说button： android:layout\_gravity 表示按钮在界面上的位置。 android:gravity表示button上的字在button上的位置。

**8 Andorid中的android:gravity与android:layout\_gravity属性**

Android:gravity属性是对view中的内容的限定，比如button上面的text

Androd:layout\_gravity属性是用来设置该view相对于父view的位置，比如一个button在linearlayout里，你想把它放在linearlayout里靠左、靠右等

例子：

<Button android:id=*"@+id/button1"*

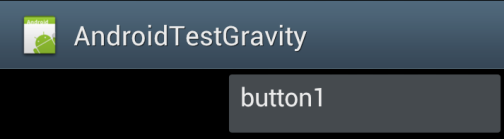
android:layout\_width=*"200dp"*

android:layout\_height=*"wrap\_content"*

android:layout\_gravity=*"right"//基于父窗口右对齐*

android:gravity=*"left"//button里面的文字左对齐*

android:text=*"@string/button1"*></Button>



**注意一点：**如果将LinearLayout的方向改为水平时，由于几个按钮不能同时在一行出现，因此后面的按钮会显示不出来，也就是说LinearLayout不会自动换行

**9 几种布局**

1）表格布局

表格布局由TableLayout代表，每次向TableLayout添加一个TableRow，该TableRow就是一个表格行，TableRow也可以是容器，因此它可以不断地添加其他组件，每添加一个子组件就增加一行，在表格布局管理中，可以为单元格设置如下三种方式：

（1）Shrinkable：该列所有单元格可以被收缩

（2）Stretchable：该列所有单元格可以被拉伸

（3）Collapsed：该列所有单元格被隐藏

一个简单的示例：

<TableLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"*

android:stretchColumns=*"1,2"*>

<Button android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"button1"*/>

<TableRow android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*>

<Button android:id=*"@+id/button1"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"普通按钮"*/>

<Button android:id=*"@+id/button2"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"允许被拉伸的按钮"*/>

<Button android:id=*"@+id/button3"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"允许被拉伸的按钮"*/>

</TableRow>

<TableRow android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*>

<Button android:id=*"@+id/button4"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"普通按钮"*/>

<Button android:id=*"@+id/button5"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"测试允许被拉伸的按钮"*/>

</TableRow>

<Button android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"butto6"*/>

</TableLayout>



从上面的例子可以看出：

如果没有添加TableRow标签则所添加的控件占一行

TableLayout的列数由TableRow的最大列数决定

TableLayout的每列宽度由该列的最大宽度决定

如果某个TableRow中的控件个数小于TableLayout的最大列数，则某些列为空

2）帧布局

帧布局组件为每个加入其中的组件创建一个空白的区域（称为一帧），所有每个子组件占据一帧，这些帧都会根据gravity属性执行自动对齐（会产生叠加效果）。

3）相对布局

相对布局由RelativeLayout代表，相对布局容器内的子组件的位置总是相对兄弟组件、父容器来决定的，如果A组件的位置是由B组件的位置来决定的，Android要求先定义B组件，再定义A组件。

其两个重要属性：

android:gravity：设置该布局容器各子组件的对齐方式

android:ignoreGravity：设置哪个组件不受gravity组件的影响

<RelativeLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<TextView android:id=*"@+id/view0"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:layout\_centerInParent=*"true"*/>

<!-- layout\_above:位于上方 layout\_alignLeft:左边对齐 -->

<TextView android:id=*"@+id/view1"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:layout\_above=*"@id/view0"*

android:layout\_alignLeft=*"@id/view0"*/>

<TextView android:id=*"@+id/view2"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:layout\_below=*"@id/view0"*

android:layout\_alignLeft=*"@id/view0"*/>

<TextView android:id=*"@+id/view3"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:layout\_toLeftOf=*"@id/view0"*

android:layout\_alignTop=*"@id/view0"*/>

<TextView android:id=*"@+id/view4"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:layout\_toRightOf=*"@id/view0"*

android:layout\_alignTop=*"@id/view0"*/>

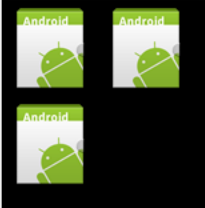
</RelativeLayout>

显示效果：



将view0的对齐属性改为：android:layout\_alignParentLeft=*"true"*

显示效果如下：



4）绝对布局

绝对布局就是android不提供任何布局控制，而是由开发人员自己通过XY坐标来控制组件的位置。使用绝对布局时，每个子组件都可以指定如下两个XML属性：

layout\_x：指定该组件的X坐标

layout\_y：指定该组件的y坐标

下面实现一个登陆界面：

<AbsoluteLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<TextView android:id=*"@+id/name"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:layout\_x=*"20dip"*

android:layout\_y=*"20dip"*

android:textSize=*"15dip"*

android:text=*"用户名:"*/>

<EditText android:id=*"@+id/nameEdit"*

android:hint=*""*

android:layout\_x=*"80dip"*

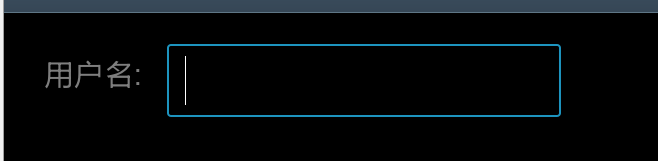
android:layout\_y=*"15dip"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:width=*"200dip"*/>

</AbsoluteLayout>



**10 常用控件**

1）TextView与EditText

TextView直接继承于View，它还是EditText、Button两个UI控件的父类，几种常用控件的继承关系：



两个属性的介绍：

**android:autoLink属性值：**

none：不设置任何超链接

web：将文本的URL地址转化为超链接

email：将文本的email地址转化为超链接

phone：将文本的超链接转化为超链接

map：将文本的街道地址转化为超链接

all：相当于指定web、email、phone与map属性

**android:ellipsize属性值：**

none：不进行任何处理

start：在文本开头部分进行省略

middle：在文本中间部分进行省略

end：在文本末尾部分进行省略

marquee：在文本结尾处以淡出的方式省略

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<!-- 设置文字大小 -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:text=*"我爱java"*

android:textSize=*"20pt"*/>

<!-- 设置省略号 -->

<TextView android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:singleLine=*"true"*

android:text=*"我爱java我爱java我爱java我爱java我爱java我爱java我爱java我爱java"*

android:ellipsize=*"middle"*/>

<!-- 设置email属性 -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:singleLine=*"true"*

android:autoLink=*"email"*

android:text=*"测试kongyeeku@163.com"*/>

<!-- 设置文本颜色、大小、并使用阴影 -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:text=*"测试文字"*

android:shadowColor=*"#0000ff"*

android:shadowDx=*"15.0"*

android:shadowDy=*"20.0"*

android:shadowRadius=*"45.0"*

android:textColor=*"#ff0000"*

android:textSize=*"25pt"*/>

<!-- 设置密码 -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:singleLine=*"true"*

android:text=*"hello"*

android:password=*"true"*/>

</LinearLayout>



**带背景、图片的文本：**

默认情况下，TextView是不带边框的，如果想为TextView带边框，我们可以考虑为TextView设置一个背景，这个背景只是一个边框，这样就实现了带边框的TextView

<!-- 带背景的TextView -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/ic\_launcher"*

android:text=*"带背景的TextView"*/>

<!-- 带图片的TextView -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:drawablePadding=*"50dp"*

android:text=*"带图片的TextView"*



**带边框的文本：**

<!-- 带边框的文本 -->

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:background=*"@drawable/bg\_border"*

android:text=*"带边框的文本"*/>



其中bg\_border是定义在drawable文件中的xml文件，内容如下：

<shape xmlns:android=*"http://schemas.android.com/apk/res/android"* >

<solid android:color=*"#000000"*></solid>

<stroke android:width=*"2dip"* android:color=*"#ff0000"*></stroke>

</shape>

2）Button与ImageButton

Button继承了TextView，ImageButton继承了ImageView，Button上面显示文字，ImageButton上面显示图像，在ImageButton上设置文字属性没用

<Button android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:text=*"普通按钮"*

android:textSize=*"10pt"*/>

<ImageButton android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:contentDescription=*"null"*

android:background=*"@drawable/ic\_launcher"*

/>



<ImageButton android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:background=*"#00000000"*

**android:src=*"@drawable/button\_selector"***/>

其中button\_selector为xml文件：

<selector xmlns:android=*"http://schemas.android.com/apk/res/android"* >

<!-- 指定按钮按下时的图片 -->

<item android:state\_pressed=*"true"*

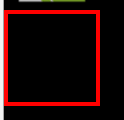
android:drawable=*"@drawable/bg\_border"*></item>

<!-- 指定按钮放松时的图片 -->

<item android:state\_pressed=*"false"*

android:drawable=*"@drawable/ic\_launcher"*></item>

</selector>

普通状态： 按下状态：

在ImageButton中有两个属性，background与src，其中background是设置背景颜色，而src是设置图片内容，即前景

3）RadioButton与CheckBox

RadioButton与CheckBox都继承了Button，两者与普通按钮不同的是它们多了一个可选中的功能，因此，RadioButton与CheckBox都可额外指定一个android:checked属性

<TableLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:orientation=*"vertical"* >

<TableRow>

<TextView android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"性别:"*

android:textSize=*"11pt"*></TextView>

<RadioGroup android:orientation=*"horizontal"*

android:layout\_gravity=*"center\_horizontal"*>

<RadioButton android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:checked=*"true"*

android:text=*"男"*/>

<RadioButton android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"女"*/>

</RadioGroup>

</TableRow>

<TableRow>

<TextView android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"喜欢的颜色:"*

android:textSize=*"12pt"*/>

<LinearLayout android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:orientation=*"vertical"*

android:layout\_gravity=*"center\_horizontal"*>

<CheckBox android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"红色"*

android:checked=*"true"*/>

<CheckBox android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"蓝色"*/>

<CheckBox android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"白色"*

android:checked=*"true"*/>

</LinearLayout>

</TableRow>

</TableLayout>



4）状态开关按钮（ToggleButton）

ToggleButton主要用于切换程序中的某种状态，它支持的主要属性如下：

Android:checked:设置该按钮是否被选中

Android:textOff:设置当按钮没有被选中时显示的文本

Android:textoOn:设置当按钮被选中时显示的文本

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<ToggleButton android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:id=*"@+id/toggle"*

**android:checked=*"true"***

**android:textOn=*"竖屏排列"***

**android:textOff=*"横屏排列"***/>

<LinearLayout android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:id=*"@+id/test"*

android:orientation=*"vertical"*

>

<Button android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"button1"*/>

<Button android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"button2"*/>

</LinearLayout>

</LinearLayout>

在Activity类中进行控制：

ToggleButton toggleButton =(ToggleButton)findViewById(R.id.*toggle*);

**final** LinearLayout linearLayout = (LinearLayout)findViewById(R.id.*test*);

toggleButton.setOnCheckedChangeListener(**new**OnCheckedChangeListener() {

@Override

**public** **void** onCheckedChanged(CompoundButton buttonView, **boolean** isChecked) {

**if**(isChecked){

linearLayout.setOrientation(LinearLayout.*VERTICAL*);

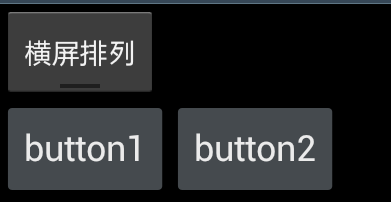
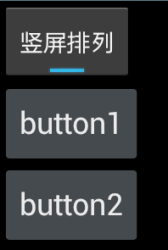
}**else**{

linearLayout.setOrientation(LinearLayout.*HORIZONTAL*);

}

}

});



5）模拟时钟（AnologClock）与数字时钟（DigitalClock）

AnologClock时钟继承了View组件，它重写了View的onDraw方法，它会在View上显示模拟时钟，不会显示当前秒数；DigitalClock时钟继承了TextView，也就说它本身就是文本框，只是它里面显示的内容是当前时间，会显示当前秒数

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<AnalogClock android:layout\_width=*"wrap\_content"*

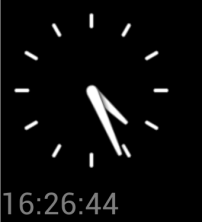
android:layout\_height=*"wrap\_content"*/>

<DigitalClock android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:textSize=*"12pt"*/>

</LinearLayout>



6）图像视图（ImageView）

ImageView继承于View组件，任何Drawable对象都可用ImageView显示，几个比较重要的属性：

android:adjustViewbounds:设置ImageView是否调整自己的边界来保持所显示图片的长宽比

android:src:设置ImageView所显示的Drawable对象的ID

android:scaleType:设置所显示的图片如何缩放或移动以适应ImageView的大小

一个例子：

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<LinearLayout android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:orientation=*"horizontal"*

android:gravity=*"center\_horizontal"*>

<Button android:id=*"@+id/plus"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"增大透明度"*/>

<Button android:id=*"@+id/minus"*

android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:text=*"减少透明度"*/>

<Button android:id=*"@+id/next"*

android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:text=*"下一张"*/>

</LinearLayout>

<!-- 定义显示图片整体的ImageView -->

<ImageView android:id=*"@+id/image1"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"400px"*

android:background=*"#0000ff"*

android:src=*"@drawable/first"*

/>

<!-- 定义显示图片局部细节的ImageView -->

<ImageView android:id=*"@+id/image2"*

android:layout\_width=*"120dp"*

android:layout\_height=*"120dp"*

android:background=*"#0000ff"*

android:layout\_marginTop=*"10dp"*/>

</LinearLayout>

Java代码控制：

**private** **int**[] imageSrc = **new** **int**[]{R.drawable.*first*, R.drawable.*second*, R.drawable.*third*};

**private** **int** currentImageIndex = 0;//当前默认为第一张图片

**private** **int** alpha = 255;

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*imageview*);

**final** ImageView image1 = (ImageView)findViewById(R.id.*image1*);

**final** ImageView image2 = (ImageView)findViewById(R.id.*image2*);

**final** Button plus = (Button)findViewById(R.id.*plus*);

**final** Button minus = (Button)findViewById(R.id.*minus*);

**final** Button next = (Button)findViewById(R.id.*next*);

next.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** onClick(View v) {

BitmapDrawable bitmapDrawable = (BitmapDrawable)image1.getDrawable();

**if(!bitmapDrawable.getBitmap().isRecycled()){**

**bitmapDrawable.getBitmap().recycle();**

**}**

**if**(currentImageIndex == imageSrc.length - 1){

currentImageIndex = 0;

}**else**{

currentImageIndex += 1;

}

image1.setImageResource(imageSrc[currentImageIndex]);

}

});

//定义改变图片透明度的方法

OnClickListener listener = **new** OnClickListener() {

@Override

**public** **void** onClick(View v) {

**if**(v == plus)

alpha += 20;

**if**(v == minus)

alpha -= 20;

**if**(alpha >= 255)

alpha = 255;

**if**(alpha <= 0)

alpha = 0;

image1.setAlpha(alpha);

}

};

plus.setOnClickListener(listener);

minus.setOnClickListener(listener);

//设置触屏监听器

image1.setOnTouchListener(**new** OnTouchListener(){

@Override

**public** **boolean** onTouch(View v, MotionEvent event) {

BitmapDrawable bitmapDrawable = (BitmapDrawable)image1.getDrawable();

Bitmap bitmap = bitmapDrawable.getBitmap();

**int** x = (**int**)event.getX();

**int** y = (**int**)event.getY();

**if**(x + 120 > bitmap.getWidth()){

x = bitmap.getWidth() - 120;

}

**if**(x - 120 < 0){

x = 120;

}

**if**(y + 120 > bitmap.getHeight()){

y = bitmap.getHeight() - 120;

}

**if**(y - 120 < 0){

y = 120;

}

image2.setImageBitmap(Bitmap.*createBitmap*(bitmap, x, y, 120, 120));

image2.setAlpha(alpha);

**return** **true**;//此时表示所有的触屏事件都由这里处理

}

}

);

}



在上面的代码中：对于Bitmap调用createBitmap可以截取位图的指定部分—该方法返回截取区域的新位图，上面加粗的部主动调用了recycle方法进行回收，这样可以保证不会抛出一个OOM（Out Of Memory）错误

7）自动完成文本框（AutoCompleteTextView）

自动完成文本框由EditText派生，常用属性为：

android:completionHint:设置下拉菜单中的提示标题

android:completionThreshold:设置用户至少输入几个字符才会显示提示

android:dropDownHeight:设置下拉框的高度

android:dropDownHorizontalOffset:设置下拉框与文本框之间的水平偏移，下拉框默认与文本框左对齐

android:dropDownVerticalOffset:设置下拉框与文本框之间的垂直偏移，下拉框默认紧跟文本框

android:dropDownWidth:设置下拉框的宽度

android:popupBackground:设置下拉菜单的背景

一个例子：

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"match\_parent"*

android:layout\_height=*"match\_parent"*

android:orientation=*"vertical"* >

<AutoCompleteTextView android:id=*"@+id/auto"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:completionHint=*"请选择喜欢的运动"*

android:completionThreshold=*"1"*/>

</LinearLayout>

Java代码：

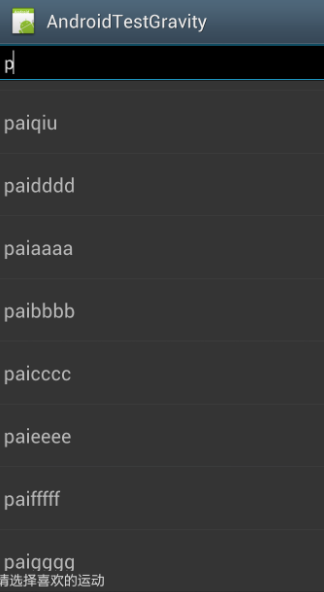
setContentView(R.layout.*autocompletetextview*);

AutoCompleteTextView autoCompleteTextView = (AutoCompleteTextView)findViewById(R.id.*auto*);

ArrayAdapter<String> arrayAdapt = **new** ArrayAdapter<String>(**this**, android.R.layout.*simple\_dropdown\_item\_1line*, yundong);

autoCompleteTextView.setAdapter(arrayAdapt);

**private** String[] yundong = **new** String[]{"lanqiu", "zuqiu", "pingpangqiu", "paiqiu", "ganlanqiu","wangqiu", "taiqiu", "paidddd", "paiaaaa", "paibbbb", "paicccc", "paieeee", "paifffff","paigggg", "paihhhh"};



8）Spinner的功能和用法

这里的Spinner实际上就是一个列表选择框，不过android的列表选择框并不是需要显式下拉列表的，而是相当于弹出一个菜单供用户选择

Spinner是ViewGroup的间接子类，因此它也可作为容易使用，其XML属性：

android:prompt：设置该列表选择框的提示

android:entries：使用数组资源设置该下拉列表框的列表项目

如果我们在使用spinner时已经可以确定下拉列表框项的列表项，则可以直接使用android:entries属性

<Spinner android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:entries=*"@array/bookname"*/>

其中bookname为一个xml中的数组（XML文件名为books）：

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

<resources>

<string-array name=*"bookname"*>

<item>疯狂Java讲义</item>

<item>疯狂Ajax讲义</item>

<item>疯狂XML讲义</item>

<item>疯狂Javascript讲义</item>

</string-array>

</resources>



9）不带存储项目的Spinner

Spinner spinner = (Spinner)findViewById(R.id.*spinner*);

//创建适配器

BaseAdapter baseAdapter = **new** BaseAdapter(){

@Override

**public** **int** getCount() {

**return** 50;//指定移动包含50个选项

}

@Override

**public** Object getItem(**int** arg0) {

**return** **null**;

}

@Override

**public** **long** getItemId(**int** arg0) {

**return** 0;

}

//重写该方法，该方法返回的View将作为列表框的选项

@Override

**public** View getView(**int** position, View convertView, ViewGroup arg2) {

**TextView textView = new TextView(TestandroidcomActivity.this);**

textView.setText(position + "");

textView.setTextSize(20);

textView.setTextColor(Color.*RED*);

**return textView;**

}

};

spinner.setAdapter(baseAdapter);

上面的黑体部分决定了Spinner的列表框的每个表项显示什么：该方法返回什么View对象，每个列表项就显示什么View，例如我们改为LinearLayout：

//重写该方法，该方法返回的View将作为列表框的选项

@Override

**public** View getView(**int** position, View convertView, ViewGroup arg2){

LinearLayout linearLayout = **new** LinearLayout(TestandroidcomActivity.**this**);

linearLayout.setOrientation(LinearLayout.*HORIZONTAL*);

ImageView imageView = **new** ImageView(TestandroidcomActivity.**this**);

imageView.setImageResource(R.drawable.*ic\_launcher*);

TextView textView = **new** TextView(TestandroidcomActivity.**this**);

textView.setText(position + "");

textView.setTextSize(20);

textView.setTextColor(Color.*RED*);

linearLayout.addView(imageView);

linearLayout.addView(textView);

**return** linearLayout;

}



10）日期时间选择器（DatePicker与TimePicker）

DatePicker与TimePicker都从FrameLayout派生而来，其中DatePicker供用户选择日期，TimePicker则供用户选择时间

<TextView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:text=*"请选择时间："*/>

<DatePicker android:id=*"@+id/datePicker"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:layout\_gravity=*"center\_horizontal"*/>

<TimePicker android:id=*"@+id/timePicker"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:layout\_gravity=*"center\_horizontal"*/>

<EditText android:id=*"@+id/show"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:editable=*"false"*

android:cursorVisible=*"false"*/>

OnCreate函数中的代码为：

DatePicker datePicker = (DatePicker)findViewById(R.id.*datePicker*);

TimePicker timePicker = (TimePicker)findViewById(R.id.*timePicker*);

//获取当前的年月日小时分钟

Calendar calendar = Calendar.*getInstance*();

year = calendar.get(Calendar.*YEAR*);

month = calendar.get(Calendar.*MONTH*);

day = calendar.get(Calendar.*DAY\_OF\_MONTH*);

hour = calendar.get(Calendar.*HOUR*);

minute = calendar.get(Calendar.*MINUTE*);

datePicker.init(year, month, day, **new** OnDateChangedListener(){

@Override

**public** **void** onDateChanged(DatePicker arg0, **int** year, **int** month, **int** day) {

TestandroidcomActivity.**this**.year = year;

TestandroidcomActivity.**this**.month = month;

TestandroidcomActivity.**this**.day = day;

ShowDate();

}

});

timePicker.setOnTimeChangedListener(**new** OnTimeChangedListener(){

@Override

**public** **void** onTimeChanged(TimePicker arg0, **int** hour, **int** minute) {

TestandroidcomActivity.**this**.hour = hour;

TestandroidcomActivity.**this**.minute = minute;

ShowDate();

}

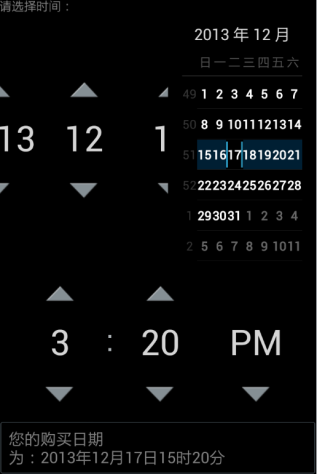
});

**public** **void** ShowDate(){

EditText exitText = (EditText)findViewById(R.id.*show*);

exitText.setText("您的购买日期为：" + year + "年" + (month + 1) + "月" + day + "日" + hour + "时" + minute + "分");

}



11）进度条（ProgressBar）

Android支持几种风格的进度条，通过style可以为ProgressBar指定风格，有一个比较常用的属性android:progressDrawable属性，该属性可以指定为一个LayerDrawable对象（该对象可以通过在XML文件中用<layer-list>元素进行配置）的引用，下面的例子是两种方式的使用（普通的和自定义的）

<ProgressBar android:id=*"@+id/bar"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:max=*"100"*

style=*"@android:style/Widget.ProgressBar.Horizontal"*/>

<ProgressBar android:id=*"@+id/bar1"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:max=*"100"*

android:progressDrawable=*"@drawable/my\_bar"*

style=*"@android:style/Widget.ProgressBar.Horizontal"*/>

在Java代码中：

**final** ProgressBar progress = (ProgressBar)findViewById(R.id.*bar*);

**final** ProgressBar progress1 = (ProgressBar)findViewById(R.id.*bar1*);

//创建handle

**final** Handler handler = **new** Handler(){

@Override

**public** **void** handleMessage(Message msg){

**if**(msg.what == 0x0111){

progress.setProgress(hasComplete);

progress1.setProgress(hasComplete);

}

}

};

**new** Thread(){

@Override

**public** **void** run(){

**while**(hasComplete < 100){

hasComplete++;

**try**{

Thread.*sleep*(100);

Message message = **new** Message();

message.what = 0x0111;

handler.sendMessage(message);

}**catch**(Exception e){

}

}

}

}.start();



显示在标题上的进度条

这种进度条不需要ProgressBar组件，它直接由Activity的方法启用，一般采用如下两个步骤：

（1）调用Activity的requestWindowFeature()方法，该方法根据传入的参数可启用特定的窗口特征，例如传入Window.FEATURE\_INDETERMINATE\_PROGRESS在窗口标题上显示不带进度的进度条，传入Window.FEATURE\_PROGRESS则显示带进度的进度条

（2）调用Activity的setProgressBarVisibility(boolean)或者setProgressBarIndeterminateVosono(boolean)方法即可控制进度条的显示和隐藏

**注意点：requestWindowFeature方法必须在setContentView函数之前调用**

在OnCreate函数中：

**this**.requestWindowFeature(Window.*FEATURE\_PROGRESS*);//带进度的进度条**this**.requestWindowFeature(Window.*FEATURE\_INDETERMINATE\_PROGRESS*);//显示不带进度的进度条

setContentView(R.layout.*main*);

Button show = (Button)findViewById(R.id.*show*);

Button hide = (Button)findViewById(R.id.*hide*);

show.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** onClick(View view) {

setProgressBarIndeterminateVisibility(**true**);

setProgressBarVisibility(**true**);

TestandroidcomActivity.**this**.setProgress(4500);//放在标题栏上的水平进度条不能设置最大值，因为系统已经将其设置为10000

}

});

hide.setOnClickListener(**new** OnClickListener() {

@Override

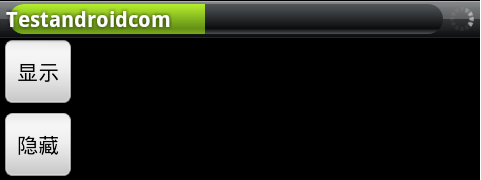
**public** **void** onClick(View v) {

setProgressBarIndeterminateVisibility(**false**);

setProgressBarVisibility(**false**);

}

});



12）拖动条（SeekBar）

SeekBar中有二个属性

android:thumb可以指定一个drawable对象，该对象将作为自定义滑块

android:progressDrawable可以指定图片背景

一个例子：

<SeekBar android:id=*"@+id/seekBar"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:max=*"255"*

android:progress=*"255"*

android:thumb=*"@drawable/ic\_launcher"*

android:progressDrawable=*"@drawable/gate"*/>

<ImageView android:id=*"@+id/imageView"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:src=*"@drawable/gate"*/>

**Java代码：**

**final** ImageView imageView = (ImageView)findViewById(R.id.*imageView*);

**final** SeekBar seekBar = (SeekBar)findViewById(R.id.*seekBar*);

seekBar.setOnSeekBarChangeListener(**new** OnSeekBarChangeListener(){

@Override

**public** **void** onProgressChanged(SeekBar seekBar, **int** progress, **boolean** fromUser) {

imageView.setAlpha(progress);

}

@Override

**public** **void** onStartTrackingTouch(SeekBar arg0) {

}

@Override

**public** **void** onStopTrackingTouch(SeekBar arg0) {

}

});



13）星级评分条（RatingBar）

星级评分常用的xml属性：

android:isIndicator:设置评分条是否允许用户改变（true为不允许改变）

android:numStars:设置该评分条共有多少个星级

android:rating:设置该评分条默认的星级

android:stepSize:设个每次最少需要改变多少个星级

XML文件：

<RatingBar android:id=*"@+id/ratingBar"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:max=*"255"*

android:rating=*"255"*

android:isIndicator=*"false"*

android:numStars=*"7"*

android:stepSize=*"0.3"*/>

<ImageView android:id=*"@+id/imageView"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:src=*"@drawable/gate"*/>

Java代码：

**final** ImageView imageView = (ImageView)findViewById(R.id.*imageView*);

**final** RatingBar ratingBar = (RatingBar)findViewById(R.id.*ratingBar*);

ratingBar.setOnRatingBarChangeListener(**new** OnRatingBarChangeListener(){

@Override

**public** **void** onRatingChanged(RatingBar ratingBar, **float** rating, **boolean** fromUser) {

imageView.setAlpha((**int**)(rating \* 255 / 7));//注意这里的rating是0-7.0（星星的个数）

}

});



14）选项卡（TabHost）

详细解释：<http://www.blogjava.net/freeman1984/archive/2009/11/18/302803.html>

TabHost仅仅是一个容器，它提供了如下两个方法来创建选项卡、添加选项卡：newTabSpec(String tag)：创建选项卡

addTab(TabHost.TabSpec tabSpec)：添加选项卡

使用TabHost的 一般步骤为：

（1）在界面布局中定义TabHost组件，并为该组件定义该选项卡的内容

（2）Avtivity应该继承TabActivity

（3）调用TabActivity的getTabHost方法获取TabHost对象

（4）通过TabHost对象的方法来创建选项卡、添加选项卡

**关于android中的LayoutInflater解释：**

LayoutInflater的作用类似于findViewById，不同点是LayoutInflater是用来找layout下的xml布局文件，**并且实例化**，而findViewById是找具体xml西安的具体widget控件（如Button、TextView等），查看LayoutInflater的from源码：

**public** **static** LayoutInflater from(Context context) {

LayoutInflater LayoutInflater =(LayoutInflater) context.getSystemService(Context.LAYOUT\_INFLATER\_SERVICE);

**if** (LayoutInflater == **null**) {

**throw** **new** AssertionError("LayoutInflater not found.");

}

**return** LayoutInflater;

}

我们会发现它所使用的LayoutInflater仍然是通过获得系统服务得到的，我们在外面的代码中也可以通过获取系统服务得到

**关于setContentView函数**

使用setContentView可以在Activity中动态切换显示的View,这样，不需要多个Activity就可以显示不同的界面，因此不再需要在Activity间传送数据，变量可以直接引用。但是，在android SDK给我们建的默认的Hello World程序中，调用的是setContentView(int layoutResID)方法，如果使用该方法切换view,在切换后再切换回，无法显示切换前修改后的样子，也就是说，相当于重新显示一个view,并非是把原来的view隐藏后再显示。其实setContentView是个多态方法，我们可以先用LayoutInflater把布局xml文件引入成View对象，再通过setContentView(View view)方法来切换视图。因为所有对View的修改都保存在View对象里，所以，当切换回原来的view时，就可以直接显示原来修改后的样子

**关于Activity与View中的findViewById函数**

一般来说Activity中的findViewById是通过调用了setContentView之后调用的，因为此时setContentView已经将xml中的控件实例化了，而如果仅仅采用LayoutInflater对xml进行实例化，LayoutInflater一般是处理动态生成View 的，如果调用Activity的findViewById则会返回空指针，必须调用返回View中的findViewById函数才可以，

**关于setContentView与inflate的区别：**

Inflate这个方法总共有四种形式，目的都是把xml表述的layout转化为View对象，其中有一个比较常用，View inflate(int resource, ViewGroup root)，其中：

int resource：也就是resource/layout文件在R文件中对应的ID，这个必须指定

ViewGroup root：可以为null，如果为null则表示值创建一个resource对应的View，不是null时，会将创建的view自动加为root的child

setContent一旦调用，layout就会立刻显示UI，而inflate只会吧layout形成一个以View类实现成的对象，有需要时再用setContentView(view)显示出来

一个示例：

Log.*v*("rrrr","before setContent");

View view = LayoutInflater.*from*(**this**).inflate(R.layout.*main*, **null**);

**if**(view != **null**){

Log.*v*("rrrr", "view not null");

}**else**{

Log.*v*("rrrr", "view null");

}

LinearLayout rootLinearLayout = (LinearLayout)findViewById(R.id.*root*);

**if**(rootLinearLayout != **null**){

Log.*v*("rrrr", "root not null");

}**else**{

Log.*v*("rrrr", "root null");

}

setContentView(view);

Log.*v*("rrrr","after setContent");

rootLinearLayout = (LinearLayout)findViewById(R.id.*root*);

**if**(rootLinearLayout != **null**){

Log.*v*("rrrr", "root not null");

}**else**{

Log.*v*("rrrr", "root null");

}

LinearLayout linearLayout = (LinearLayout)findViewById(R.id.*tab0*);

**if**(linearLayout != **null**){

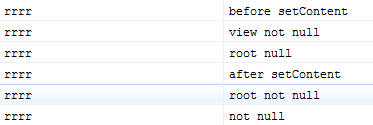
Log.*v*("rrrr", "not null");

}**else**{

Log.*v*("rrrr", "null");

}

上面的代码中如果没有添加setContentView(view)运行结果为：



从上面的代码中我们可以猜测setContentView实际上进行了xml文件中控件的实例化，但是如果我们在调用的是Layoutinflater返回View中的findViewById则可以得到实际的控件

**一段比较好的解释：**

当Activity调用setContentView时，android才会去绘制layout上的各个元素，并为其分配内存，只有分配了内存以后，才能继续执行findViewById，才能得到引用，不然得到空引用，空引用意味着，后面使用相应变量时就会发生访问的对象不存在的问题，而且当Actitivy重新setContentView之后，那些之前绘制的控件，内存都被灭掉了，所以，若是通过setContentView来达到画面切换目的的，要注意重新绘制以后重新取得引用

首先可以看看TabHost的源码，有以下几个变量：

**private** TabWidget mTabWidget;

**private** FrameLayout mTabContent;

**private** List<TabSpec> mTabSpecs = **new** ArrayList<TabSpec>(2);

也就是说TabHost必须有这三个东西，所以我们的.xml就会有规定，

在TabHost中有下面的一个函数：

Call setup() before adding tabs if loading TabHost using findViewById()

**public** **void** setup() {

mTabWidget = (TabWidget) findViewById(com.android.internal.R.id.tabs);

**if** (mTabWidget == **null**) {

**throw** **new** RuntimeException(

"Your TabHost must have a TabWidget whose id attribute is 'android.R.id.tabs'");

}

mTabContent = (FrameLayout) findViewById(com.android.internal.R.id.tabcontent);

**if** (mTabContent == **null**) {

**throw** **new** RuntimeException(

"Your TabHost must have a FrameLayout whose id attribute is " + "'android.R.id.tabcontent'");

}

}

也就是说我们的.xml文件需要TabWidget和FrameLayout标签

对于TabHost的使用有如下两种方式：

（1）使用TabActivity

在TabActivity中有一个变量TabHost mTabHost，通过调用

**public** TabHost getTabHost() {

ensureTabHost();

**return** mTabHost;

}

**private** **void** ensureTabHost() {

**if** (mTabHost == **null**) {

**this**.setContentView(com.android.internal.R.layout.tab\_content);

}

}

（2）自定义的TabHost，不用继承TabHostActivity

</LinearLayout>

<TabHost

android:id=*"@+id/tabhost"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:layout\_weight=*"1.0"* >

<LinearLayout

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:orientation=*"vertical"* >

<TabWidget

android:id=*"****@android:id/tabs****"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"* >

</TabWidget>

<FrameLayout

android:id=*"****@android:id/tabcontent****"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"* >

<LinearLayout

android:id=*"@+id/tab0"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"* >

<TextView

android:id=*"@+id/textview0"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"textview0"* />

</LinearLayout>

<LinearLayout

android:id=*"@+id/tab1"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"* >

<TextView

android:id=*"@+id/textvie1"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"textview1"* />

</LinearLayout>

<LinearLayout

android:id=*"@+id/tab2"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"* >

<TextView

android:id=*"@+id/textview2"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"textview2"* />

</LinearLayout>

</FrameLayout>

</LinearLayout>

</TabHost>

</LinearLayout>

**Java代码：**

setContentView(R.layout.*main*);

TabHost tabHost = (TabHost)findViewById(R.id.*tabhost*);

tabHost.setup();//非常重要

TabSpec tabSpec0 = tabHost.newTabSpec("tab0");

tabSpec0.setIndicator("选项卡一");

tabSpec0.setContent(R.id.*tab0*);

tabHost.addTab(tabSpec0);

TabSpec tabSpec1 = tabHost.newTabSpec("tab1");

tabSpec1.setIndicator("选项卡二");

tabSpec1.setContent(R.id.*tab1*);

tabHost.addTab(tabSpec1);

TabSpec tabSpec2 = tabHost.newTabSpec("tab2");

tabSpec2.setIndicator("选项卡三三三三撒三三撒上男萨斯那你算啊");

tabSpec2.setContent(R.id.*tab2*);

tabHost.addTab(tabSpec2);

tabHost.setCurrentTab(2);

1）注意上面的TabWidget标签中的id为系统***@android:id/tabs，***FrameLayout标签中的id为系统***@android:id/tabcontent***

2）在上面的Java代码中一定要注意到**tabHost.setup**的调用，因为setup函数中会初始化我们TabHost类中的TabWidget与FrameLayout成员变量

3）还有一点需要注意的是，在上面的TabWidget与FrameLayout标签的外层包上一个LinearLayout标签，这样两者才会垂直排列

4）如果我们想把标签放入底部，有两种方法：

方法一：可以将包含TabWidget与FrameLayout两个标签的LinearLayout标签改为RelativeLayout标签，同时在TabWidget标签中添加android:layout\_alignParentBotton=”true”标签，注意如果是采用的LinearLayout标签，则android:layout\_alignParentBotton=”true”是无效的

方法二：仍然使用LinearLayout标签，但是将TabWidget与FrameLayout两个标签的顺序颠倒，同时在FrameLayout标签中添加属性android:layout\_weight=”1.0”

15）滚动视图（ScrollView）

滚动视图ScrollView由FrameLayout派生而来，它就是一个用于为普通控件添加滚动条的组件，ScrollView里最多只能包含一个组件，而ScrollView的作用就是为该组件添加垂直滚动条，如果要添加水平滚动条可以使用HorizontalScrollView控件

<ScrollView

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"* >

<HorizontalScrollView

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"* >

**<LinearLayout**

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

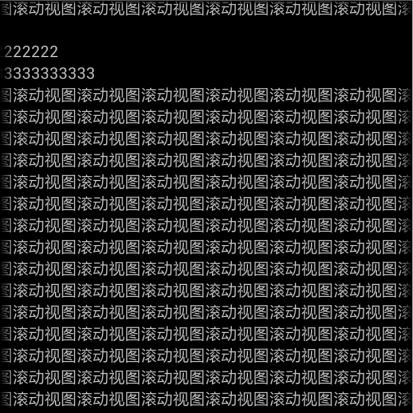
android:orientation=*"vertical"*>

**一些定义的TextView组件**

**</LinearLayout>**

</HorizontalScrollView>

</ScrollView>



使用这个控件需要注意的是：

1. 如果想添加水平和垂直滚动条则可以再ScorellView中嵌套HorizontalScrollView或者在HorizontalScrollView中嵌套ScorellView
2. 无论是在ScorellView还是HorizontalScrollView中都只能直接嵌套一个直接子控件，就像上面的例子一样，只是在HorizontalScrollView嵌套了一个LinearLayout子控件，如果有多个直接子控件则会报错，例如如果我们将上面例子中HorizontalScrollView下面的LinearLayout控件去掉，则会报下面的错：



16）列表视图（ListView与ListActivity）

创建ListView有两种方式：

1. 直接使用ListView
2. 让Activity继承ListActivity

AutoCompleteTextView、Spinner、ListView采用Adapter提供数据的设计就是MVC的典型应用

ListView的常用属性：

Android:choiceMode:设置ListView的选择行为

Android:divider:设置List列表项的分割线（既可用颜色，也可用Drawable分隔）

Android:dividerHeight:设置分隔条的高度

Android:entries:指定一个数组资源，android将根据该数组资源来生成ListView

Android:footDividersEnabled:如果设置为false，则不在footer View之前绘制分割线

Android:headerDividersEnabled: 如果设置为false，则不在header View之前绘制分割线

例子一：

<!-- 采用数组资源给出列表项 -->

<ListView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:entries=*"@array/bookname"*

android:divider=*"@drawable/my\_bar"*

android:dividerHeight=*"10dp"*></ListView>

其中my\_bar是一张图片，book为一个xml文件

<resources>

<string-array name=*"bookname"*>

<item>疯狂Java讲义</item>

<item>疯狂Ajax讲义</item>

<item>疯狂XML讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

<item>疯狂Javascript讲义</item>

</string-array>

</resources>



例子二：

<ListView android:id=*"@+id/list"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:divider=*"@drawable/my\_bar"*

android:dividerHeight=*"10dp"*></ListView>

Java代码：

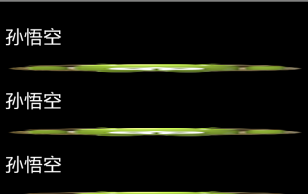
setContentView(R.layout.*main*);

ListView listView = (ListView)findViewById(R.id.*list*);

String[] array = **new** String[]{"孙悟空", "猪八戒", "孙悟空", "孙悟空", "孙悟空", "孙悟空", "孙悟空", "孙悟空", "孙悟空", "孙悟空"};

ArrayAdapter<String> arrayAdapter = **new** ArrayAdapter<String>(**this**, android.R.layout.*simple\_list\_item\_1*, array);

listView.setAdapter(arrayAdapter);



创建ArrayAdapter时必须指定一个textViewResourceId，该参数决定了每个列表项的外观形式，Android属性提供了如下属性值：

Simple\_list\_item\_1：每个列表项都是一个普通的TextView

Simple\_list\_item\_2：每个列表项都是一个普通的TextView（字体略大）

Simple\_list\_item\_checked：每个列表项都是一个已勾选的列表项

Simple\_list\_item\_multiple\_choice：每个列表项都是带多选框的文本

Simple\_list\_item\_single\_choice：每个列表项都是带单选框的文本

**使用SimpleAdapter作为适配器**

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:id=*"@+id/root"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:orientation=*"vertical"* >

<ListView android:id=*"@+id/list"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:divider=*"@drawable/my\_bar"*

android:dividerHeight=*"10dp"*></ListView>

<ImageView android:id=*"@+id/header"*

android:layout\_width=*"50dp"*

android:layout\_height=*"50dp"*

android:paddingLeft=*"10dp"*/>

<TextView android:id=*"@+id/name"*

android:layout\_height=*"wrap\_content"*

android:layout\_width=*"wrap\_content"*

android:textSize=*"16dp"*

android:gravity=*"center\_vertical"*

android:paddingLeft=*"10dp"*/>

</LinearLayout>

Java代码：

**private** String[] name = **new** String[]{"虎头", "弄玉", "李清照", "李白"};

**private** **int**[] image = **new** **int**[]{R.drawable.*abc*, R.drawable.*gate*, R.drawable.*ic\_launcher*, R.drawable.*my\_bar*};

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

ListView listView = (ListView)findViewById(R.id.*list*);

List<Map<String, Object>> listItems = **new** ArrayList<Map<String, Object>>();

**for**(**int** i = 0; i < name.length; i++){

Map<String, Object> listItem = **new** HashMap<String, Object>();

listItem.put("header", image[i]);

listItem.put("name", name[i]);

listItems.add(listItem);

}

//创建一个SimpleAdapter

SimpleAdapter simpleAdapter = **new** SimpleAdapter(**this**, listItems, R.layout.*main*, **new** String[]{"header", "name"}, **new** **int**[]{R.id.*header*, R.id.*name*});

listView.setAdapter(simpleAdapter);



使用SimpleAdapter的最大难点在于创建SimpleAdapter对象，它需要五个参数，其中后面四个参数非常关键：

第二个参数：该参数应该是一个List<? Extends Map<String, ?>>类型的集合对象，该几何中每个Map<String, ?>对象生成一个列表项

第三个参数：指定一个界面布局的ID，表示第五个参数中的界面组件来自该界面布局

第四个参数：应该是一个String[]，类型，决定提取Map<String, ?>对象中哪些key对应value来生生列表项

第五个参数：应该是一个int[]类型，指定使用哪些View组件来组合成一个列表项

17）可展开的列表组件（ExpandableListView）

ExpandableListView是ListView的子类，它在ListView基础上进行了扩展，把应用中的列表项分为几组，每组里又可包含多个列表项，ExpandableListView与ListView使用非常类似，只是ExpandableListView所显示的列表项应该由ExpandableAdapter提供，其中ExpandableListView所额外支持的常用属性：

android:childDivider:指定各组内各子列表项之间的分隔线

android:childIndicator:显示在子列表项旁边的Drawable对象

android:groupIndicator:显示在组列表项旁边的Drawable对象

在OnCreate函数中调用：

setContentView(R.layout.*main*);

ExpandableListView expandableListView = (ExpandableListView)findViewById(R.id.*list*);

ExpandableListAdapter expandableListAdaper = **new** ExpandableListAdapter(){

**int**[] logos = **new** **int**[]{R.drawable.*abc*, R.drawable.*gate*, R.drawable.*ic\_launcher*};

**private** String[] armTypes = **new** String[]{"神族兵种", "虫族兵种", "人族兵种"};

**private** String[][] arms = **new** String[][]{

{"狂战士", "龙骑士", "黑暗圣坛", "电兵"},

{"小狗", "刺蛇", "飞龙", "自爆飞机"},

{"机枪兵", "护士MM", "幽灵"}

};

@Override

**public** **boolean** areAllItemsEnabled() {

**return** **false**;

}

@Override

**public** Object getChild(**int** groupPosition, **int** childPosition) {

**return** arms[groupPosition][childPosition];

}

@Override

**public** **long** getChildId(**int** groupPosition, **int** childPosition) {

**return** childPosition;

}

@Override

**public** View getChildView(**int** groupPosition, **int** childPosition, **boolean** isLastChild, View convertView, ViewGroup parent) {

TextView textView = GetTextView();

textView.setText(getChild(groupPosition, childPosition).toString());

**return** textView;

}

**private** TextView GetTextView(){

AbsListView.LayoutParams lp = **new** AbsListView.LayoutParams(ViewGroup.LayoutParams.~~FILL\_PARENT~~, 64);

TextView textView = **new** TextView(TestandroidcomActivity.**this**);

textView.setLayoutParams(lp);

textView.setGravity(Gravity.*CENTER\_VERTICAL* | Gravity.*LEFT*);

textView.setPadding(36, 0, 0, 0);

textView.setTextSize(20);

**return** textView;

}

@Override

**public** **int** getChildrenCount(**int** groupPosition) {

**return** arms[groupPosition].length;

}

@Override

**public** **long** getCombinedChildId(**long** arg0, **long** arg1) {

**return** 0;

}

@Override

**public** **long** getCombinedGroupId(**long** arg0) {

**return** 0;

}

@Override

**public** Object getGroup(**int** groupPosition) {

**return** armTypes[groupPosition];

}

@Override

**public** **int** getGroupCount() {

**return** armTypes.length;

}

@Override

**public** **long** getGroupId(**int** groupPosition) {

**return** groupPosition;

}

@Override

**public** View getGroupView(**int** groupPosition, **boolean** isExpanded, View convertView, ViewGroup parent) {

LinearLayout linearLayout = **new** LinearLayout(TestandroidcomActivity.**this**);

linearLayout.setOrientation(LinearLayout.*HORIZONTAL*);

ImageView logo = **new** ImageView(TestandroidcomActivity.**this**);

logo.setImageResource(logos[groupPosition]);

logo.setLayoutParams(**new** LinearLayout.LayoutParams(150, 150));

linearLayout.addView(logo);

TextView textView = GetTextView();

textView.setLayoutParams(**new** LinearLayout.LayoutParams(300, 150));

textView.setText(getGroup(groupPosition).toString());

linearLayout.addView(textView);

**return** linearLayout;

}

@Override

**public** **boolean** hasStableIds() {

**return** **true**;

}

@Override

**public** **boolean** isChildSelectable(**int** arg0, **int** arg1) {

**return** **true**;

}

@Override

**public** **boolean** isEmpty() {

**return** **false**;

}

@Override

**public** **void** onGroupCollapsed(**int** arg0) {

}

@Override

**public** **void** onGroupExpanded(**int** arg0) {

}

@Override

**public** **void** registerDataSetObserver(DataSetObserver arg0) {

}

@Override

**public** **void** unregisterDataSetObserver(DataSetObserver arg0) {

}

};

expandableListView.setAdapter(expandableListAdaper);



在上面的例子中，关键代码就是提供开发者自己的ExpandableListAdapter，当开发者实现自己的ExpandableListAdapter时，该组件包括如下两个关键方法：

getChildView：该方法返回的View对象将作为子列表

getGroupView：该方法返回的View对象将作为组列表项

18）网格视图（GridView）与图像切换器（ImageSwitcher）

GridView用于在界面上按行、列分布的方式来显示多个组件，GridView与ListView有共同的父类：AbsListView，区别主要在于ListView只是在一个方向上分布，而GridView则会在两个方向上分布

ImageSwitcher由FrameLayout派生而出，ImageSwitcher组件和ImageView基本相同，但是它多一个功能：它所显示的图片切换时可以设置动画效果

XML文件：

<GridView android:id=*"@+id/gridView"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:verticalSpacing=*"2pt"*

android:horizontalSpacing=*"0pt"*

android:numColumns=*"4"*

android:gravity=*"center"*></GridView>

<ImageSwitcher android:id=*"@+id/switcher"*

android:layout\_height=*"320dp"*

android:layout\_width=*"320dp"*

android:layout\_gravity=*"center\_horizontal"*></ImageSwitcher>

Java文件：

setContentView(R.layout.*main*);

**final** ImageSwitcher imageSwitcher = (ImageSwitcher)findViewById(R.id.*switcher*);

//设置图片更换的动画效果

imageSwitcher.setInAnimation(AnimationUtils.*loadAnimation*(**this**, android.R.anim.*fade\_in*)); imageSwitcher.setOutAnimation(AnimationUtils.*loadAnimation*(**this**, android.R.anim.*fade\_out*));

imageSwitcher.setFactory(**new** ViewFactory(){

@Override

**public** View makeView() {

ImageView imageView = **new** ImageView(TestandroidcomActivity.**this**);

imageView.setBackgroundColor(0xff0000);

imageView.setScaleType(ImageView.ScaleType.*FIT\_CENTER*);

imageView.setLayoutParams(**new** ImageSwitcher.LayoutParams(LayoutParams.*WRAP\_CONTENT*,

LayoutParams.*WRAP\_CONTENT*));

**return** imageView;

}

});

//创建一个list集合，list集合的元素是Map

List<Map<String, Object>> listItems = **new** ArrayList<Map<String, Object>>();

**for**(**int** i = 0; i < imageIds.length; i++){

Map<String, Object> listItem = **new** HashMap<String, Object>();

listItem.put("image", imageIds[i]);

listItems.add(listItem);

}

//创建一个SimpleAdapter

SimpleAdapter simpleAdapter = **new** SimpleAdapter(**this**, listItems, R.layout.*cell*, **new** String[]{"image"}, **new** **int**[]{R.id.*imageView*});

GridView graiView = (GridView)findViewById(R.id.*gridView*);

graiView.setAdapter(simpleAdapter);

graiView.setOnItemSelectedListener(**new** OnItemSelectedListener(){

@Override

**public** **void** onItemSelected(AdapterView<?> parent, View view,

**int** position, **long** id) {

imageSwitcher.setImageResource(imageIds[position % imageIds.length]);

}

@Override

**public** **void** onNothingSelected(AdapterView<?> arg0) {

}

});

graiView.setOnItemClickListener(**new** OnItemClickListener(){

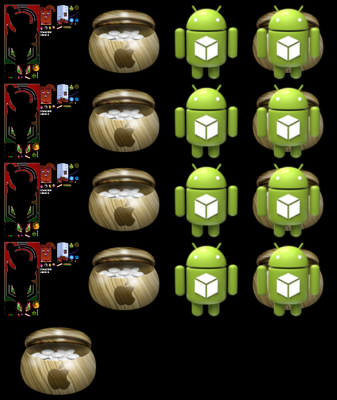
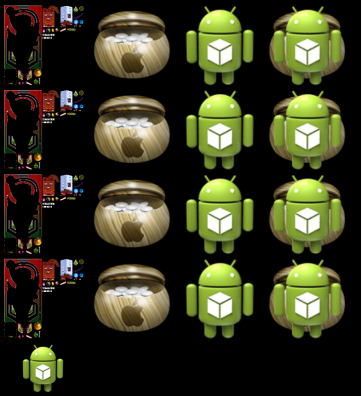
@Override

**public** **void** onItemClick(AdapterView<?> parent, View view, **int** position, **long** id) {

imageSwitcher.setImageResource(imageIds[position % imageIds.length]);

}

});



19）画廊视图（Gallery）

Gallery与Spinner两个组件有共同的父类：AbsSpinner，表明Gallery和Spinner都是一个列表框，他们之间的区别在于Spinner显示的是一个垂直的列表选择框，而Gallery表示的是一个水平的列表选择框。两者还有一个区别：Spinner的作用是供用户选择的，而Gallery则允许用户通过拖动来查看上一个、下一个列表项

**11 Android下事件处理**

Android提供了两种方式的事件处理：基于回调的事件处理和基于监听器的事件处理

**（1）基于监听的事件处理方式**

每个组件均可以针对特定的事件指定一个事件监听器，每个事件监听器也可监听一个活多个事件源。因为同一个事件源上可能发生多种事件，委派式事件处理方式可以把事件源上所有可能发生的事件分别授权给不同的事件监听器来处理，同时也可以让一类事件都是用同一个事件监听器

飞机类PlaneView

**public** **class** PlaneView **extends** View{

**public** **float** currentX, currentY;

**private** Bitmap plane;

**public** PlaneView(Context context){

**super**(context);

currentX = 0.0f;

currentY = 0.0f;

plane = BitmapFactory.*decodeResource*(context.getResources(), R.drawable.*gate*);

}

@Override

**protected** **void** onDraw(Canvas canvas) {

**super**.onDraw(canvas);

//创建画笔

Paint paint = **new** Paint();

//绘制飞机

canvas.drawBitmap(plane, currentX, currentY, paint);

}

}

主Activity

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

//去掉窗口标题

requestWindowFeature(Window.*FEATURE\_NO\_TITLE*);

//去掉状态栏 getWindow().setFlags(WindowManager.LayoutParams.*FLAG\_FULLSCREEN*, WindowManager.LayoutParams.*FLAG\_FULLSCREEN*);

PlaneView planeView = **new** PlaneView(**this**);

setContentView(planeView);

//获取窗口管理器

WindowManager windowManager = getWindowManager();

Display display = windowManager.getDefaultDisplay();

//获取屏幕宽高

**int** screenWidth = display.~~getWidth~~();

**int** screenHeight = display.~~getHeight~~();

//设置飞机的初始位置

planeView.currentX = screenWidth / 2;

planeView.currentY = screenHeight / 2;

planeView.setBackgroundResource(R.drawable.*ic\_launcher*);

}



在基于事件监听器的处理模型中，事件监听器必须实现事件监听器接口，Android为不同的界面组件提供了不同的监听器接口，这些接口通常以内部类的形式存在，以View为例，它包含了如下几个内部接口：

View.OnClickListener:单击事件的事件监听器

View:OnCreateContextMenuListener:创建上下文菜单事件的事件监听器

View.onFocusChangeListener:焦点改变事件的事件监听器

View.OnKeyListener:按键事件的事件监听器

View.OnLongClickListener:长单击事件的事件监听器

View.OnTouchListener:触屏事件的事件监听器

通过上面的介绍不难看书，所谓的事件监听器，其实就是实现了特定接口的Java类的实例，在程序中实现监听器，通常有以下几种形式：

1. 内部类形式：可以直接访问外部类的所有界面组件，这是内部类形式的优势
2. 外部类形式
3. Activity本身作为事件监听器类
4. 匿名内部类形式

外部类的形式比较少用，主要原因如下：

1. 事件监听器通常属于特定的GUI界面，定义成外部类不利于提高程序的内聚性
2. 外部类形式的事件监听器不能自由访问创建GUI界面的类中的组件，编程不简洁

**注意：还有一种更加直接简单的方式：直接绑定到标签**

对于很多Android界面组件标签而言，它们都支持如onClick、onLongClick等属性，这个属性的属性值就是一个形如\*\*\*(View source)的方法的方法名，下面就是一个例子：

XML文件：

<EditText android:id=*"@+id/editText"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:hint=*"点击按钮"*/>

<Button android:id=*"@+id/button"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"click"*

android:onClick=*"OnClick"*/>

Java代码（在主Activity中添加函数OnClick）：

**public** **void** OnClick(View view){

EditText editText = (EditText)findViewById(R.id.*editText*);

editText.setText("按钮被点击了");

}

**（2）基于回调的事件处理方式**

为了使用回调机制类处理GUI组件上所发生的事件，我们需要为该组件提供对应的事件处理方法—而Java又是一种静态语言，我们无法为某个对象动态地添加方法，因此只能继承GUI组件，并重写该类的时间处理方法来实现，在下面的例子中重写了View控件：

**public** **class** MyView **extends** View{

**public** MyView(Context context, AttributeSet set){

**super**(context, set);

**this**.setBackgroundResource(R.drawable.*gate*);

}

}

在XML中我们可以定义该组件：

<com.test.MyView android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*/>

运行结果如下：



我们会发现此时的View已经是我们重写后的View，重写控件需要注意的是构造函数：

**public** MyView(Context context, AttributeSet set){

**super**(context, set);

**this**.setBackgroundResource(R.drawable.*gate*);

}

注意它需要传一个属性标签，我们在使用xml生成布局时，系统会自动调用View的**public** MyView(Context context, AttributeSet set)这个构造函数，set这个参数是由系统帮我们自动实现的，实际上在上面的例子中我们调用了**super**(context, set);而在View中解析set时，也是用枚举方式进行的，附部分源码：

**final** **int** N = a.getIndexCount();

**for** (**int** i = 0; i < N; i++) {

**int** attr = a.getIndex(i);

**switch** (attr) {

**case** com.android.internal.R.styleable.View\_background:

background = a.getDrawable(attr);

**break**;

**case** com.android.internal.R.styleable.View\_padding:

padding = a.getDimensionPixelSize(attr, -1);

**break**;

…………………

}

通过上面的介绍不难发现：对于基于监听的事件处理模型来说，事件源和事件监听器是分离的，当事件源上发生特定事件后，该事件交给事件监听器负责处理；对于基于回调事件处理模型来说，事件源和事件监听器是统一的，当事件源发生特定事件后，该事件还是由事件源本身进行处理

几乎所有基于回调的事件吃力方法都有一个boolean类型的返回值，该返回值用于标识该处理方法是否能完全处理该事件，对于基于回调的事件，某组件上所发生的事情不仅激发该组件上的回调方法，也会触发该组件所在Avtivity的回调方法---只要事件能传播到该Activity

基于回调的事件处理更适合应付那种事件处理逻辑比较固定的View，比如之前的跟随用户手指的View，可以将触屏事件封装在重写的View中

**12响应的系统设置事件**

主要用到Configuration类，Configuration类专门用来描述手机设备上的配置信息，这些配置信息既包括用户特定的配置项，也包括系统的动态设置配置

XML文件：

<EditText android:id=*"@+id/org"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:hint=*"显示屏幕方向"*/>

<EditText android:id=*"@+id/touch"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:hint=*"显示触摸屏状态"*/>

<EditText android:id=*"@+id/mnc"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:hint=*"显示移动网络代号"*/>

<Button android:id=*"@+id/information"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:text=*"获取设备信息"*/>

Java代码：

**final** EditText ori = (EditText)findViewById(R.id.*org*);

**final** EditText touch = (EditText)findViewById(R.id.*touch*);

**final** EditText mnc = (EditText)findViewById(R.id.*mnc*);

**final** Button information = (Button)findViewById(R.id.*information*);

information.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View view) {

Configuration configuration = getResources().getConfiguration();

ori.setText(configuration.orientation == Configuration.*ORIENTATION\_LANDSCAPE* ? "横向屏幕" : "竖向屏幕");

mnc.setText(configuration.mnc + "");

**switch**(configuration.touchscreen){

**case** Configuration.*TOUCHSCREEN\_FINGER*:

touch.setText("接受手指的触摸屏");

**break**;

**case** Configuration.*TOUCHSCREEN\_NOTOUCH*:

touch.setText("无触摸屏");

**break**;

**case** Configuration.*TOUCHSCREEN\_STYLUS*:

touch.setText("触摸笔式的触摸屏");

**break**;

**case** Configuration.*TOUCHSCREEN\_UNDEFINED*:

touch.setText("未知的触摸屏");

**break**;

}

}

});



如果程序需要监听系统设置的更改，则可以考虑重写Activity的onConfigurationChanged方法，该方法是一个基于回调的事件处理方法，当系统设置发生更改时，该方法将会自动触发，下面是一个例子：

OnCreate函数中：

setContentView(R.layout.*main*);

**final** Button change = (Button)findViewById(R.id.*change*);

change.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View arg0) {

Configuration configuration = getResources().getConfiguration();

**if**(configuration.orientation == Configuration.*ORIENTATION\_LANDSCAPE*){//当前是横屏

TestandroidcomActivity.**this**.setRequestedOrientation(ActivityInfo.*SCREEN\_ORIENTATION\_PORTRAIT*);

}

**if**(configuration.orientation == Configuration.*ORIENTATION\_PORTRAIT*){//当前是竖屏

TestandroidcomActivity.**this**.setRequestedOrientation(ActivityInfo.*SCREEN\_ORIENTATION\_LANDSCAPE*);

}

}

});

@Override

**public** **void** onConfigurationChanged(Configuration newConfig) {

**super**.onConfigurationChanged(newConfig);

String screen = newConfig.orientation == Configuration.*ORIENTATION\_LANDSCAPE* ? "横向屏幕" : "竖向屏幕";

Toast.*makeText*(**this**, "系统的屏幕方向发生变化\n" + "修改后的屏幕方向为：" + screen, Toast.*LENGTH\_LONG*).show();

}

注意在XML中要添加两个重要的选项：

<uses-permission android:name=*"android.permission.CHANGE\_CONFIGURATION"*/>

同时在Activity的属性中添加

android:configChanges=*"orientation"*

添加了这个属性，当我们横竖屏切换的时候会直接调用onCreate方法中的onConfigurationChanged方法，而不会重新执行onCreate方法，如果不配置这个属性的话就会重新调用onCreate方法了

**13 深入理解Activity**

Android应用多个Activity组成Activity栈，当前活动的Activity位于栈顶

采用LauncherActivity开发启动Activity的列表：

LauncherActivity继承了ListActivity，它所开发的列表界面与普通列表界面不同，它开发的列表界面中每个列表项对应于一个Intent，因此，当用户单击不同的列表项时，应用程序会自动启动对应的Activity，继承LauncherActivity时通常应高重写Intent intentForPosition(int position)方法，该方法根据不同列表项返回不同的Intent

**14 Service与BroadcastReceiver**

Service是Android四大组件中与Activity最接近的，它们都代表可执行的程序，Setvice与Activity的区别在于：Service一直在后台运行，它没有用户界面，所以绝不会到前台来，一旦Service启动，它就和Activity一样，完全具有自己的生命周期。Android系统本身提供了大量的Service组件，开发者可通过这些系统Service来操作Android系统本身

创建一个Service步骤：

（1）定义一个继承Service的子类

（2）在AndroidManifest.xml文件中配置该Service

与Activity相似，Service中也定义了系列生命周期方法，如下：

**Abstract IBinder onBind(Intent intent)：**该方法是Service子类必须实现的方法，该方法返回一个IBinder对象，应用程序可通过该对象与Service组件通信

**Void onCreate：**当该Service第一次被创建后将立即回调该方法

**Void onDestroy：**当该Service被关闭之前将会回调该方法

**Void onStartCommand（Intent intent，int flags，int startId）：**每次客户端调用startService（Intent）方法启动该Service时都会调用该方法

**Boolean onUnbind（Intent intent）：**放该Service上绑定的所有客户端都断开连接时将会调用该方法

当一个Service创建完成后，接下来就可以在程序中运行该Service了，android系统中运行Service有如下两种方式：

1. 通过Context的startService方法：通过该方法启用Service，访问者与Service之间没有关联，即使访问者退出了，Service仍然运行
2. 通过Context的bindService方法：使用该方法启用Service，访问者与Service绑定在了一起，访问者一旦退出，Service也就终止了

创建一个Service：

**public** **class** FirstService **extends** Service{

@Override

**public** IBinder onBind(Intent arg0) {

**return** **null**;

}

@Override

**public** **void** onCreate() {

Log.*v*("test", "Service onCreate");

**super**.onCreate();

}

@Override

**public** **void** onDestroy() {

Log.*v*("test", "Service onDestroy");

**super**.onDestroy();

}

@Override

**public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {

Log.*v*("test", "Service onStartCommand");

**return** **super**.onStartCommand(intent, flags, startId);

}

}

在AndroidManifest.xml文件中添加service：

<service android:name=*".FirstService"*>

<intent-filter >

<action android:name=*"com.test.service.FIRST\_SERVER"*/>

</intent-filter>

</service>

在Java代码中调用：

Button startServer = (Button)findViewById(R.id.*start*);

Button stopServer = (Button)findViewById(R.id.*stop*);

//创建启动Service的Intent

**final** Intent intent = **new** Intent();

intent.setAction("com.test.service.FIRST\_SERVER");

startServer.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View v) {

**startService(intent);**

}

});

stopServer.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View v) {

**stopService(intent);**

}

});

按下启动服务按钮：



再按下启动服务按钮：



按下停止服务按钮：



再按下启动服务按钮：



从上面可以看出，每当Service被创建时会调用onCreate方法，每次Service被启动时都会回调onStart方法---多次启动一个已有的Service组件将不会再回调onCreate方法，但每次启动时都会调用onStartCommand方法

程序通过startService与stopService启动关闭Service时，Service与访问者之间基本上不存在太多的关联，因此Service和访问者之间也无法进行通信与数据交换，

如果Service和访问者之间需要进行方法调用或数据交换，则应该使用bindService和unbindServive方法启动和关闭服务

Contextde的bindService完整方法为：

**bindService（Intent service，ServiceConnection conn，int flags）**

service：通过Intent指定要启动的Service

conn：用于监听访问者与Service之间的连接情况，当访问者与Service之间连接成功时将回调该ServiceConnection的onServiceConnected（ComponentNama name，Ibinder service）方法；当访问者与Service之间断开连接时将回调该ServiceConnection的onServiceDisconnected（ComponentName name）方法

flags：指定绑定时是否自动创建Service（如果Service还没有创建）。该参数可指定为0（不自动创建）或BIND\_AUTO\_CREATE（自动创建）

下面是一个例子：

**public** **class** FirstService **extends** Service{

**private** **int** count;

**private** **boolean** quit;

**private** MyBinder binder = **new** MyBinder();

**class** MyBinder **extends** Binder{

**public** **int** GetCount(){

**return** count;

}

}

@Override

**public** IBinder onBind(Intent intent) {

Log.*v*("test", "server onBind");

**return** binder;

}

@Override

**public** **boolean** onUnbind(Intent intent){

Log.*v*("test", "server onUnBind");

**return** **true**;

}

@Override

**public** **void** onCreate() {

Log.*v*("test", "Service onCreate");

**new** Thread(){

@Override

**public** **void** run(){

**while**(!quit){

**try**{

Thread.*sleep*(1000);

}**catch**(Exception e){

}

count++;

}

}

}.start();

**super**.onCreate();

}

@Override

**public** **void** onDestroy() {

Log.*v*("test", "Service onDestroy");

quit = **true**;

**super**.onDestroy();

}

@Override

**public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {

Log.*v*("test", "Service onStartCommand");

**return** **super**.onStartCommand(intent, flags, startId);

}

}

主Activity代码：

setContentView(R.layout.*main*);

Button bindServer = (Button)findViewById(R.id.*bind*);

Button unbindServer = (Button)findViewById(R.id.*unbind*);

Button information = (Button)findViewById(R.id.*information*);

//创建ServiceConnection对象

conn = **new** ServiceConnection(){

@Override

**public** **void** onServiceConnected(ComponentName conn, IBinder binder)

{

Log.*v*("test", "service connection");

TestServiceActivity.**this**.binder = (FirstService.MyBinder)binder;//获取Service的引用

}

@Override

**public** **void** onServiceDisconnected(ComponentName arg0) {

Log.*v*("test", "service unconnection");

}

};

//创建启动Service的Intent

**final** Intent intent = **new** Intent();

intent.setAction("com.test.service.FIRST\_SERVER");

bindServer.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View v) {

bindService(intent, conn, Service.*BIND\_AUTO\_CREATE*);

}

});

unbindServer.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View v) {

unbindService(conn);

}

});

information.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View arg0) {

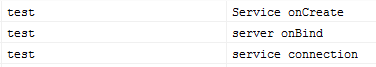
Toast.*makeText*(TestServiceActivity.**this**, "Service的Count值为：" + binder.GetCount(), 4000).show();

}

});



点击绑定服务按钮：



点击解除服务：



注意：不管我们点击绑定服务多少次，onBind都只调用一次（已证明），这一点与startService不同，每次点击启动Service都会调用onStartCommand函数

注意，在上面的例子中我们会发现没有调用ServiceConnection中的onServiceDisConnected函数，在连接正常关闭的情况下是不会被调用的，该方法只在Service被破坏了或者被杀死的时候调用，即被异常销毁时调用。例如，系统资源不足、要关闭一些Services，刚好连接绑定的Service是被关闭者之一，这个时候onServiceDisconnected就会被调用。

**上面的运行流程为：**

调用bindService函数，会触发Service的onCreate与onBind函数，这时Service会通过onBind返回一个Binder对象，通常这个对象都是在Service中的成员变量，用于与客户端进行通信，并且这个Binder对象通常都由自己重写，定义一些函数来访问Service的数据，在连接成功的情况下回触发bindService参数中ServiceConnection对象的onServiceConnected函数，通常可以利用这个函数得到Service中Binder引用（该引用可以访问Service中的数据），在我们要获取数据时，直接访问该引用的成员函数即可

通过startService启动 通过bindService启动

**跨进程调用Service（AIDL服务）**

在andorid系统中，各应用程序都运行在自己的进程中，进程之间一般无法直接进行数据交换（AIDL：Android interface definition language）

在上面的例子中，都是一种泵定本地Service方法，本地Service的onBind方法会直接把Ibinder对象本身传给客户端的ServiceConnection的onServiceConnected方法的第二个参数，但远程Service的onBind方法只是将IBinder对象的代理传给客户端的ServiceConnection的onServiceConnected方法的第二个参数，当客户端获取了远程Service的Ibinder对象的代理之后，接下来就可以通过该IBinder对象去回调远程Service的属性或方法了，android需要AIDL来定义远程接口，所以我们需要定义AIDL接口，注意AIDL文件的定义中不能添加修饰符，如public、private等

创建一个AIDL文件，然后采用android sdk中的aidl.exe文件将adil文件转化为java文件，该java文件存放于gen文件夹下，定义好上面的AIDL后，就可以定义一个Service实现类了，该Service的onBind方法所返回的Ibinder对象应该是ADT自动生成的ICat.Stub的子类的实例了，至于其他的部分，则与本地开发完全一样

**一个小插曲（Java中的代理）**

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以再客户端和目标对象之间起到中介的作用

**14 Android中图形与图像处理**

**public** **class** BitmapTestActivity **extends** Activity {

**private** AssetManager assetManager;

**private** String[] images = **null**;

**private** **int** currentImageIndex = 0;

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

**final** ImageView imageView = (ImageView)findViewById(R.id.*imageView*);

**try**{

assetManager = getAssets();

images = assetManager.list("");

}**catch**(Exception e){

e.printStackTrace();

}

**final** Button button = (Button)findViewById(R.id.*button*);

button.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** onClick(View v) {

currentImageIndex++;

Log.*v*("camera", "" + images.length);

**if**(currentImageIndex >= images.length){

currentImageIndex = 0;

}

//寻找下一张图片

**while**(!images[currentImageIndex].endsWith(".jpg") &&

!images[currentImageIndex].endsWith(".png") &&

!images[currentImageIndex].endsWith(".gif")){

currentImageIndex++;

**if**(currentImageIndex >= images.length){

currentImageIndex = 0;

}

}

InputStream assetFile = **null**;

**try**{

assetFile = assetManager.open(images[currentImageIndex]);

}**catch**(Exception e){

e.printStackTrace();

}

//对之前绑定的Bitmap进行强制回收

BitmapDrawable bitmapDrawable = (BitmapDrawable)imageView.getDrawable();

**if(bitmapDrawable != null && !bitmapDrawable.getBitmap().isRecycled()){**

**bitmapDrawable.getBitmap().recycle();**

**}**

**imageView.setImageBitmap(BitmapFactory.*decodeStream*(assetFile));**

}

});

}

}

**关于里面的Bitmap，在上面的代码中发现，实际上在ImageView中封装了一个Drawable，而ImageDrawable中封装了一个Bitmap，看源码：**

**public** **class** BitmapDrawable **extends** Drawable {

**private** **static** **final** **int** DEFAULT\_PAINT\_FLAGS =

Paint.FILTER\_BITMAP\_FLAG | Paint.DITHER\_FLAG;

**private** BitmapState mBitmapState;

**private** Bitmap mBitmap;

}

**public** **class** ImageView **extends** View {

// settable by the client

**private** Uri mUri;

**private** **int** mResource = 0;

**private** Matrix mMatrix;

**private** Drawable mDrawable = **null**;

**}**

**Android中的绘图**

Android中的绘图应该继承View组件，并重写它的onDraw(Canvas canvas)方法

Canvas除了可以应用DrawLine等之外，还提供了变换函数：rotate、scale、translate，Canvas提供的上面的方法还设计一个API：Paint，Paint代表了Canvas上的画笔，因此Paint主要用于设置绘制风格、包括画笔颜色、画笔粗细、填充风格等

重写的View的onDraw函数：

**public** **void** onDraw(Canvas canvas){

**super**.onDraw(canvas);

canvas.drawColor(Color.*BLUE*);//设置画板背景颜色

Paint paint = **new** Paint();

paint.setAntiAlias(**true**);//去锯齿

paint.setColor(Color.*RED*);

**paint.setStyle(Paint.Style.*STROKE*);//线条绘制**

paint.setStrokeWidth(3);//线条颜色

canvas.drawCircle(40, 40, 30, paint);//绘制圆形

canvas.drawRect(10, 80, 70, 140, paint);//绘制矩形

RectF rect1 = **new** RectF(10, 200, 70, 230);

canvas.drawRoundRect(rect1, 15, 15, paint);//绘制圆角矩形

RectF rect2 = **new** RectF(10, 240, 70, 270);

canvas.drawOval(rect2, paint);//绘制椭圆

//顶一个Path对象，封闭成一个三角形

Path path1 = **new** Path();

path1.moveTo(10, 340);

path1.lineTo(70, 340);

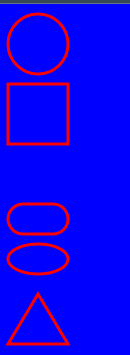
path1.lineTo(40, 290);

path1.close();

//格局Path进行绘制三角形

canvas.drawPath(path1, paint);

}

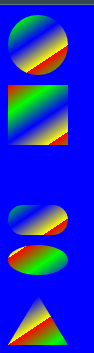


我们仍然可以设置阴影，将上面的**paint.setStyle(Paint.Style.*STROKE*);//线条绘制**，改为

Shader mShader = **new** LinearGradient(0, 0, 40, 60, **new** **int**[]{Color.*RED*, Color.*GREEN*, Color.*BLUE*, Color.*YELLOW*}, **null**, Shader.TileMode.*REPEAT*);

paint.setShader(mShader);

运行结果为：



在上面的例子中用到了Android的Path，实际上Android还未路径绘制提供了PathEffect来定义绘制效果，PathEffect包含了下面的子类（每个子类代表一个效果）：

ComposePathEffect、CornerPathEffect、DashPathEffect、DiscretePathEffect、PathDashPathEffect、SumPathEffect

同时android还提供了drawTextOnPath函数用于绘制文字路径

**实现游戏动画**

本质上就是不断地调用View的onDraw函数，通知View重绘可调用invalidate（在UI线程中）或postInvalidate（在非UI线程中）

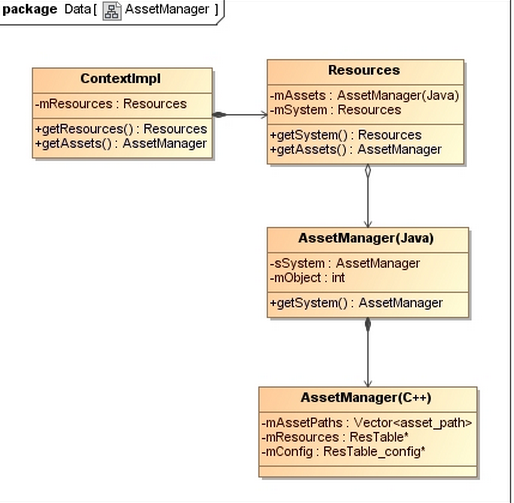
**采用双缓冲实现画板图**

所谓双缓冲技术其实很简单：当陈旭需要在指定View上进行绘制时，程序并不直接绘制到该View组件上，而是先绘制到一个内存中的Bitmap图片上（这就是缓冲）上，等到内存中的Bitmap绘制好之后，再一次性地将Bitmap绘制到View组件上

**Android中的AssetManager**

http://blog.csdn.net/luoshengyang/article/details/8791064

在android系统中，一个进程是可以同时加载多个应用程序的，也就是可以同时加载多个APK文件，每个APK文件在进程中都对应有一个全局的Resources对象以及一个全局的AssetManager对象，其中，这个全局的Resources对象保存在一个对应的ContextImpl对象的成员变量mResources中，而这个全局的AssetManager对象保存在这个全局的Resources对象的成员变量mAssets中



**14 Android中的手势判断**

Android的手势检测提供了一个GestureDetector类，创建GestureDetector需要传入一个OnGestureListener实例，实际上就是一个监听器，负责对用户的手势行为提供响应，该监听器存在下面方法：

1. Boolean onDown(MotionEvent e)
2. Boolean onFliing(MotionEvent e1, MotionEvent e2, float velocity, float velocityY)
3. Abstract void onLongPress(MotionEvent e)
4. Boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)
5. Void onShowPress(MotionEvent e)
6. Boolean onSingleTapUp(MotionEvent e)

一个通过手势变换图像大小的例子：

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

gestureDetector = **new** ~~GestureDetector~~(**this**);

imageView = (ImageView)findViewById(R.id.*imageView*);

matrix = **new** Matrix();

bitmap = BitmapFactory.*decodeResource*(**this**.getResources(), R.drawable.*ic\_launcher*);

bitmapOrigin = BitmapFactory.*decodeResource*(**this**.getResources(), R.drawable.*ic\_launcher*);

//获取屏幕大小

screenWidth = getWindowManager().getDefaultDisplay().~~getWidth~~();

screenHeight = getWindowManager().getDefaultDisplay().~~getHeight~~();

imageView.setImageBitmap(bitmap);

imageView.setX(screenWidth / 2 - bitmap.getWidth() / 2);

imageView.setY(screenHeight / 2 - bitmap.getHeight() / 2);

}

@Override

**public** **boolean** onTouchEvent(MotionEvent event) {

**return** gestureDetector.onTouchEvent(event);

}

@Override

**public** **boolean** onFling(MotionEvent event1, MotionEvent event2, **float** velocityX, **float** velocityY) {

velocityX = velocityX > 4000 ? 4000 : velocityX;

velocityX = velocityX < -4000 ? -4000 : velocityX;

currentScale += (currentScale \* velocityX / 4000.0f);

currentScale = currentScale > 0.01f ? currentScale : 0.01f;

matrix.reset();

matrix.setScale(currentScale, currentScale);

BitmapDrawable temp = (BitmapDrawable)imageView.getDrawable();

//根据原始位图和Matrix创建图片

Bitmap bitmap2 = Bitmap.*createBitmap*(bitmapOrigin, 0, 0, bitmapOrigin.getWidth(), bitmapOrigin.getHeight(), matrix, **true**);

imageView.setImageBitmap(bitmap2);

imageView.setX(screenWidth / 2 - bitmap2.getWidth() / 2);

imageView.setY(screenHeight / 2 - bitmap2.getHeight() / 2);

**if**(!temp.getBitmap().isRecycled()){

temp.getBitmap().recycle();

}

**return** **true**;

}



在上面的例子中，用到了createBitmap(Bitmap source, int x, int y, int width, int height, Matrix m, Boolean filter)

该函数表示从源位图source的指定坐标点（给定x、y）开始，从中“挖取”宽width、高height的一块出来，创建新的Bitmap对象，并按Matrix指定的规则进行变换

**14 Android中的OpenGL ES**

Android为openGL ES提供了GLSurfaceView组件，这个组件用户显示3D图形，GLSurfaceView本身并不提供绘制3D图形的功能，而是由GLSurfaceView.Renderer来完成SurfaceView中3D图形的绘制，其使用需要三个步骤：

1. 创建GLSurfaceView组件，使用Activity来显示GLSurfaceView组件
2. 为GLSurfaceView组件创建GLSurfaceView.Renderer实例，实现GLSurfaceView.Renderer类时需要实现该接口里的三个方法

**Abstract void onDrawFrame(GL10 gl)**：Renderer调用该方法绘制GLSurfaceView的当前帧

**Abstract void onSurfaceChanged（GL10 gl，int width，int height）**：当GLSurfaceView大小改变时回调该方法

**Abstract void onSurfaceCreated（GL10 gl，EGLConfig congif）：**当GLSurfaceVuew被创建时回调该方法

1. 调用GLSurfaceView组件的setRenderer（）方法指定Renderer对象，该Renderer对象将会完成GLSurfaceView里的3D图形绘制

小插曲：

openGl中的防止抖动，对于可用颜色较少的系统，可以以牺牲分辨率为代价，通过颜色值的抖动来增加可用颜色的数量，所以对分辨率足够高的机器采用抖动基本上没有太大影响，在openGL中通过glEnable(GL\_DITHER)启动

[**Android数据的四种存储方式SharedPreferences、SQLite、Content Provider和File**](http://www.cnblogs.com/wisekingokok/archive/2011/09/13/2174484.html)

SharedPreferences底层采用XML文件，XML处理时Dalvik会通过自带底层的本地XML Parse进行解析

**14 Android四大组件**

<http://www.cnblogs.com/bravestarrhu/archive/2012/05/02/2479461.html>

**android---ContentProvider**

为了在不同应用之间交换数据，android提供了ContentProvider，ContentProvider是不同应用之间进行数据交换的标准API，当一个应用程序需要把自己的数据暴露给其他程序使用时，该应用程序就可以通过提供ContentPrivider来实现；其它应用程序就可通过ContentResolver来操作ContentPrivider暴露的数据

一旦某个应用程序通过ContentProvider暴露了自己的数据操作接口，那么不管该用用程序是否启动，其它应用程序都可通过该接口来操作该应用程序的内部数据，包括增加数据、删除数据、修改数据、查询数据等。ContentPrivider以某种Uri的形式对外提供数据，允许其它应用访问或修改数据；其它应用程序使用ContentResolver根据Uri访问操作指定数据

ContentProvider的使用步骤：

1. 定义自己的ContentProvider类，该类需要继承Android提供的ContentProvider
2. 向Android系统注册这个“网站”，也就是在AndroidManifest.Xml文件中注册这个ContentProvider

通过ContentResolver操作数据有如下方法：

1. getContentResolver
2. insert
3. delete
4. update
5. query

一般来说，ContentProvider是单例模式，当多个应用程序通过ContentResolver操作ContentProvider提供的数据时，ContentResolver调用的数据操作将会委托给同一个ContentProvider处理

使用ContentResolver操作数据步骤：

（1）调用Activity的getContentResolver获取ContentResolver对象

（2）根据需要调用ContentResolver的insert、update、delete和query方法操作数据

**开发自己的ContentProvider**

1. 开发一个ContentProvider的子类，该子类需要实现增、删、改、查等功能
2. 在AndroidManifest.xml文件中注册该ContentProvider

开发ContentProvider时所实现的增、删、改、查方法都需要一个Uri参数，该参数决定了对哪个Uri执行数据操作，为了确定该ContentProvider实际能匹配的Uri，以及确定每个方法中Uri参数所操作的数据，android提供了UriMatcher工具类

注意ContentProvider有一个onCreate方法，该方法在ContentProvider创建后会被调用，当其他应用程序第一次访问ContentProvider时，该ContentProvider会被创建出来，并立即回调该onCreate函数

同时我们在开发的时候需要使用两个工具类：

（1）UriMatcher

Void addURI(String authority,String path,int code)：该方法用于向UriMatcher对象注册Uri。其中authority和path组合成一个Uri，而code则代表该Uri对应的标志码

Int match(Uri uri)：根据前面注册的Uri来判断指定Uri对应的标识码，如果找不到匹配的标志码，该方法将会返回-1

（2）ContentUris

withAppendedId(uri,id)：用于为路径加上ID部分

parseUd(uri)：用于从Uri中解析出所包含的ID值

同时我们还可以监听Provider时候改变，采用ContentObserver类，使用方式为

getContent().getContentResolver().notigyChange(uri, null)

这行代码可以用于同时所有注册在该Uri上的监听者：该ContentProvider所共享的数据发生了改变，在使用监听器的时候首先需要注册：

getContentResolver().registerContentObserver(Uri.parse(“content  
://sms”), true, new SmsObserver(new Hander));

**android---BroadcastReceiver**

broadcastReceiver本质上是一种全局监听器，用于监听系统全局的广播消息，因此它可以非常方便地实现系统中不同组件之间的通信，例如我们希望客户端程序与startService方法启动的Service之间通信，就可以采用BroadcastReceiver

当应用程序发出一个Broadcast Intent，所有匹配该Intent的BroadcastReceiver都有可能被启动

程序启动BroadcastReceiver只需要两步：

1. 创建需要启动的BroadcastReceiver的Intent
2. 调用Context的sendBroadcast或sendOrderedBroadcast方法来启动指定的BroadcastReceiver

与Activity、Sevice有完成的生命周期不同，BroadcastReceiver本质上只是一个系统级的监听器，专门负责监听各程序所发出的Broadcast

BroadcastReceiver与前面介绍的OnListener不同，OnListener只是程序级别的监听器，而BroadcastReceiver是系统级别的，它拥有自己的进程，只要存在与之匹配的Intent被广播出来，BroadcastReceiver总会被激发

每次系统Broadcast事件发生后，系统就会创建对应的BroadcastReceiver实例，并自动触它的onReceiver方法，onReceiver方法执行后，BroadcastReceiver的实例就会被销毁，如果BroadcastReceiver的onReceive方法不能在10s内完成，android会认为该程序无响应，所以不要在BroadcastReceiver的onReceive方法里执行一些耗时操作，否则会弹出ANR对话框

**发送广播**

发送方：

Intent intent = **new** Intent();

intent.setAction("com.test.intent.boradcastReceiver");

intent.putExtra("message", "简单消息");

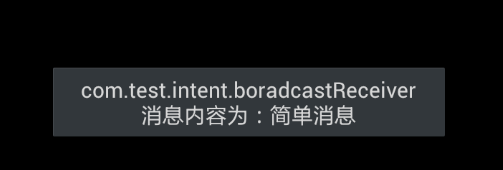
**sendBroadcast(intent);**

MyReceiver:

**public** **void** onReceive(Context context, Intent intent) {

Toast.*makeText*(context, intent.getAction() + "\n消息内容为：" + intent.getStringExtra("message"), 50000).show();

}



**有序广播：**

在Android中Broadcast可以分为两种：

**Normal Boradcast（普通广播）：**这种广播是完全异步的，可以再同一时刻（逻辑上）被所有接受者收到，消息传递的效率比较高。缺点是接受者不能将处理结果传递给下一个接受者，并且无法终止Broadcast Intent的传播

**Ordered Broadcast（有序广播）：**这种广播的接受者将按预先声明的优先级依次接受Broadcast，如果A高于B，那么消息将先传递给A，然后传给B，，优先界别在<intent-filter>元素中的android:priority属性中设置，取值范围在-1000-1000，

OrderedBroadcast接受者可以终止Broadcast Intent的传播，一旦终止后面的接受者就无法接收到Broadcast，另外OrderedBroadcast的接受者可以讲数据传递给下一个接受者，并且可以修改了之后再进行传递

对于Ordered Broadcast而言，优先接收到Broadcast的接受者可以通过setResultExtras(Bundle)方法将处理结果存入Broadcast中，然后传给下一个接受者，下一个接受者通过代码：Bundle bubdle = getResultExtras(true)可以获取上一个接受者存入的数据。

系统收发短信，发出的Broadcast属于Ordered Broadcast。如果想阻止用户收到短信，额可以通过设置优先级，让自定义的BroadcastReceiver先获取Broadcast，然后终止Broadcast

一个例子：

<receiver android:name=*".MyReceiver"*>

<intent-filter android:priority=*"20"*>

<action android:name=*"com.test.intent.boradcastReceiver"*/>

</intent-filter>

</receiver>

<receiver android:name=*".MySecondReciver"*>

<intent-filter android:priority=*"0"*>

<action android:name=*"com.test.intent.boradcastReceiver"*></action>

</intent-filter>

</receiver>

**MainActivity：**

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

Button button = (Button)findViewById(R.id.*button*);

button.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View view) {

Intent intent = **new** Intent();

intent.setAction("com.test.intent.boradcastReceiver");

intent.putExtra("message", "简单消息");

sendOrderedBroadcast(intent, **null**);

}

});

}

**MyReceiver：**

**public** **class** MyReceiver **extends** BroadcastReceiver{

@Override

**public** **void** onReceive(Context context, Intent intent) {

Toast.*makeText*(context, intent.getAction() + "\n消息内容为：" + intent.getStringExtra("message"), 50000).show();

//创建一个新的Bundle对象，并存入数据

Bundle bundle = **new** Bundle();

bundle.putString("first", "第一个BroadcastReceiver存入的消息");

setResultExtras(bundle);

**abortBroadcast();**//取消Broadcast的继续传播

}

}

**MySecondReceiver：**

**public** **class** MySecondReciver **extends** BroadcastReceiver{

@Override

**public** **void** onReceive(Context context, Intent intent) {

Bundle bundle = getResultExtras(**true**);

String first = bundle.getString("first");

Toast.*makeText*(context, "第一个Broadcast存取的消息为：" + first, 5000).show();

}

}

在运行上面的程序中，如果添加了abortBroadcast函数，则



添加之后结果为：





、

**使用Intent和IntentFilter进行通信**

使用Intent的好处：

1. 使用Intnet提供了一致的编程模型，启动Activity、Service、BroadcastReceiver都可以采用Intent
2. 在某些时候，应用程序指向启动具有某种特征的组件，并不想和某个具体的组件耦合

使用Intent启动系统组件

Android的应用程序包含三种重要组件：Activity、Service、BroadcastReceiver，应用程序采用一致的方式启动它们—依靠Intent启动

Intent对象大致包含Component、Action、Category、Data、Type、Extra和Flag这7个属性，其中Component用于明确指定需要启动的目标组件，而Extra则用于“携带”需要交换的数据

1 Component属性

Intnet的Component属性需要接受一个ComponentName对象，创建一个ComponentName对象需要指定包名和类名，这样应用程序可以根据给定的组件类去启动特定的组件

一个简单的例子：

**MainAvtivity：**

button.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View view) {

Intent intent = **new** Intent(BroadcastReceiverActivity.**this**, SecondActivity.**class**);

startActivity(intent);

}

});

**SecondActivity：**

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*secondmain*);

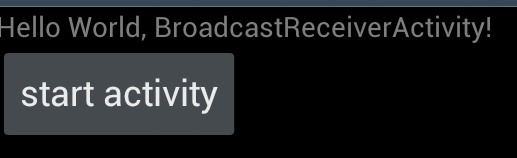
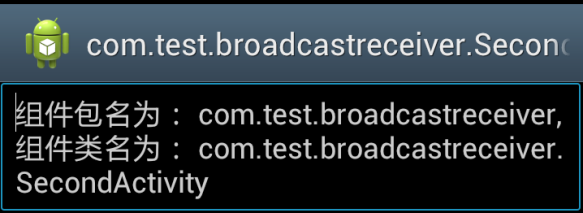
EditText editText = (EditText)findViewById(R.id.*editText*);

ComponentName name = getIntent().getComponent();

//显示该ComponentName的包名、类名

editText.setText("组件包名为： " + name.getPackageName() + ", 组件类名为： " + name.getClassName());

}

🡪 

注意，我们需要在AndroidManifest文件中注册SecondActivity

采用这种方式启动Activity，我们都不需要设置Intent-Filter属性

2 Action与Category属性

Action与Category属性都一个普通的字符串，其中Action代表该Intent所要完成的一个抽象“动作”，而Category则用于为Action增加额外的附加类别信息，通常这两个属性会结合使用

采用设置Action属性的方式启动组件，我们从代码中并不能看到是启动的哪个组件（或许是Activity，或者是BroadcastReceiver）

**public** **final** **static** String *STR* = "com.test.intent.INTENT\_ACTION";

button.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View view) {

Intent intent = **new** Intent();

intent.setAction(BroadcastReceiverActivity.*STR*);

startActivity(intent);

}

});

从上面的代码可以看出根据Intent来启动Activity—但该Intent并没有以“硬编码”的方式指定要启动哪一个Activity，具体启动哪一个呢？这取决于Activity配置中的intent-filter元素的配置

intent-filter元素中通常可以包含下面的子元素：

0-N个action子元素

0-N个category子元素

0-1个data子元素

**需要指出的是**，一个Intent对象最多只能包括一个Action属性，程序可以调用Intent的setAction(String str)方法设置Action属性，但一个Intent对象可以包含多个Category属性，程序可以通过addCategory(String str)方法为Intent添加Category属性，当程序创建Intent时，该Intent默认启动Category属性值为Intent.CATEGORY\_DEFAULT常量，因此虽然上面的程序并未指定目标的Intent的Category属性，但该Intent已有一个值为android.intent.category.DEFAULT的Category属性值，因此被启动的Activity对应的配置元素intent-filter元素中至少还要包含一个Category属性值

<activity android:name=*".SecondActivity"*>

<intent-filter >

<action android:name=*"com.test.intent.INTENT\_ACTION"*/>

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

</intent-filter>

</activity>

////////////////////////////小插曲////////////////////////////////

Activity启动其他Activity有如下两种方式：

1. startActivity(Intent intent)
2. startActivityForResult(Intent intent, int requestCode)

Activity关闭有如下两种方式：

1. finish：结束当前Activity
2. finishActivity(int requestCode)：结束已startActivityForResult(Intent intent, int requeatCode)方法启动的Activity

使用Bundle在Activity之间交换数据

Intent提供了多个重载的方法来“携带”额外的数据：

putExtras(Bubdle data)：向Intent中放入需要“携带”的数据

上面方法中的Bundle是一个简单的数据携带包，该Bubdle对象包含了多个方法来存入数据：

putXXX(String key, XXX data)：向Bubdle放入Int、Long等数据

putSerializable(String key, Serializable data)：向Bubdle中放入一个可序列化的对象

/////////////////////////////////////////////////////////////////

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

Button button = (Button)findViewById(R.id.*button*);

button.setOnClickListener(**new** OnClickListener(){

@Override

**public** **void** onClick(View view) {

String name = ((EditText)findViewById(R.id.*name*)).getText().toString();

**int** age = Integer.*parseInt*(((EditText)findViewById(R.id.*age*)).getText().toString());

**Bundle bundle = new Bundle();**

**Person person = new Person(name, age);**

**bundle.putSerializable("person", person);**

Intent intent = **new** Intent();

**intent.putExtras(bundle);**

intent.setAction(BroadcastReceiverActivity.*STR*);

startActivity(intent);

}

});

}

}

第二个Activity内容：

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*secondmain*);

EditText editText = (EditText)findViewById(R.id.*editText*);

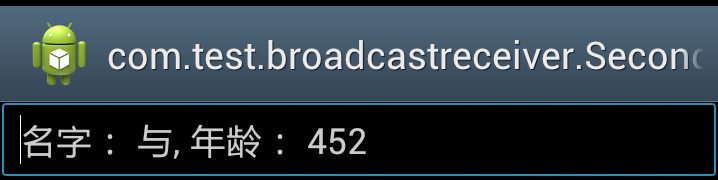
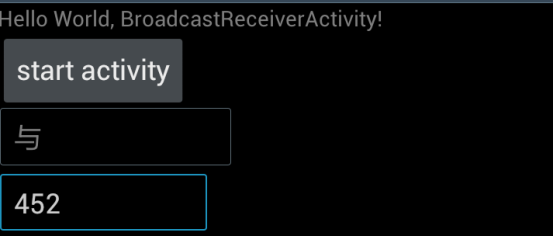
Intent intent = getIntent();

Bundle bundle = intent.getExtras();

Person person = (Person)bundle.getSerializable("person");

editText.setText("名字： " + person.GetName() + ", 年龄： " + person.GetAge());

}



**启动其他Activity并返回结果**

采用startActivityForResult(Intent intent, int requestCode)方法启动其他Activity。这种方法用于启动指定的Activity，并且希望获取指定Activity返回的结果，为了获取被启动的Activity所返回的结果，当前Activity需要重写onActivityResult(int requestCode, int resultCode, Intent intent)，其中requestCode代表请求码，而resultCode代表Activity返回的结果码，这个结果是由开发者根据业务自行设定的。一个Activity可能会调用多个Activity，当关闭这些信的Activity后，系统都会调用当前Activity的onActivityResult(int requestCode, intresultCode, Intent data)方法。为了知道该方法是由哪个请求的结果所触发的，可利用requestCode请求码；为了知道返回的数据来自于哪个新的Activity，可利用resultCode结果码

**Android中的SQLite数据库**

从本质上来说，SQLite数据库只是一个文件，SQLite的操作方式知识一种更为便捷的文件操作，当应用程序创建或打开一个SQLite数据库时，其实只是打开一个文件准备读写，Android提供了SQLiteDatabase代表一个数据库（底层就是一个数据库文件），一旦应用程序获得了代表指定数据库的SQLiteDatabase对象，接下来就可以通过SQLiteDatabase对象来管理、操作数据库了

SQLilteDatabase提供了如下静态方法打开一个文件对应的数据库：

openDatabase

openOrCreateDatabase

下面的代码创建了一个数据库并在数据库中创建一张表

SQLiteDatabase db = SQLiteDatabase.*openOrCreateDatabase*("/sdcard/temp.db3", **null**);

String sql = "create table user\_inf(user\_id integer primary key, " + " user\_name varchar(255), " + " user\_pass varchar(255))";

db.execSQL(sql);

**注意这里写入sd卡需要添加写入权限：**

<uses-permission android:name=*"android.permission.WRITE\_EXTERNAL\_STORAGE"*/>

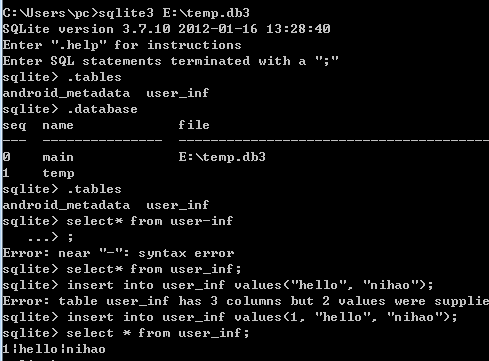
使用SQLiteDatabase进行数据库操作的步骤如下：

1. 获取SQLiteDatabase对象，它代表了与数据库的连接
2. 调用SQLiteDatabase的方法来执行SQL语句
3. 操作SQL语句的执行结果，比如用SimpleCursorAdapter封装Cursor
4. 关闭SQLiteDatabase，回收资源

我们可以采用sqlite3.exe操作数据库，首先将.db3文件从手机中导入电脑：



然后就可以操纵数据库了



SimpleCursorAdapter adapter = **new** ~~SimpleCursorAdapter~~(**this**, R.layout.*line*, cursor, **new** String[]{"title", "content"}, **new** **int**[]{

R.id.*my\_title*, R.id.*my\_content*});

listView.setAdapter(adapter);

将数据库数据绑定到ListView，在上面的代码中将Cursor封装成SimpleCursorAdapter，其构造器与SimpleAdapter大致相同，区别是SimpleAdapter负责封装集合元素为Map的List，而SimpleCursorAdapter负责封装Cursor—如果我们把Cursor里的结果集合当成List集合，Cursor里的每一行当成Map处理，两者就统一起来了

注意：使用SimpleCursorAdapter封装Cursor时要求底层数据表的主键列的列名为\_id，因为SimipleCursorAdapter只能识别列名为\_id的主键

同时我们也可以采用SQLite带有的insert。Update等方法进行数据库的操作，里面有一个重要的类型就是ContentValues，这个类似于Map，可以采用put方法

**14 Android专题**

<http://codingnow.cn/tag/android>

<http://blog.csdn.net/column/details/androiddept.html>

**15 Android下的多线程（Looper的理解）**

在andorid中线程分为有消息循环的线程和没有消息循环的线程，有消息循环的线程一般都会有一个Looper，这个是andorid的新概念。我们的主线程（UI线程）就是一个消息循环线程，针对这种消息循环的机制，我们引入一个新的机制Handler，我们有消息循环，就要往消息循环里面发送相应的消息，自定义消息一般都会有自己对应的处理、消息的发送和清除、消息的处理都封装在Handle里面，主要Handler只是针对那些有Looper的线程，不管是UI线程还是子线程，只要你有Looper，我们就可以往你的消息队列里面添加东西并做相应的处理。注意：android中关于UI的不能放在子线程处理，因为子线程是不能操作UI的，只能进行数据、系统等其他非UI的操作。

我们在创建Handler时候，里面包含了一个Looper，我们的子线程要变成消息循环线程，需要调用Looper.prepare()函数：

**public** **static** **void** prepare() {

**if** (sThreadLocal.get() != **null**) {

**throw** **new** RuntimeException("Only one Looper may be created per thread");

}

sThreadLocal.set(**new** Looper());

}

从上面的代码中我们可以看出，实际上是给该线程设置了一个Looper，同时保证在一个线程中只能存在一个Looper对象，再看看Looper的构造函数：

**private** **static** Looper mMainLooper = **null**; // guarded by Looper.class

**private** Looper() {

mQueue = **new** MessageQueue();

mRun = **true**;

mThread = Thread.currentThread();

}

实际上是创建一个消息队列，同时将该Looper与当前线程关联，同时我们发现构造函数是私有的，也就是说我们不能new一个Looper出来，我们要让自己的线程产生一个消息队列就必须要调用Looper.prepare()函数了。

**我们再看看Handler的构造函数：**

**public** Handler() {

**if** (FIND\_POTENTIAL\_LEAKS) {

**final** Class<? **extends** Handler> klass = getClass();

**if** ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&

(klass.getModifiers() & Modifier.STATIC) == 0) {

Log.w(TAG, "The following Handler class should be static or leaks might occur: " +

klass.getCanonicalName());

}

}

mLooper = Looper.myLooper();

**if** (mLooper == **null**) {//表示没有Looper，则没有消息队列

**throw** **new** RuntimeException(

"Can't create handler inside thread that has not called Looper.prepare()");

}

mQueue = mLooper.mQueue;

mCallback = **null**;

}

**16 Handler消息传递机制**

http://www.cnblogs.com/codingmyworld/archive/2011/09/14/2174255.html

Android中的消息传递机制是另一种“事件处理”，这种机制主要是为了解决android中的多线程问题—android平台不允许Activity新启动的线程访问该Activity里的界面组件，这就会导致新启动的线程无法动态改变界面组件的属性值。

Handler类的主要作用有两个：

1. 在新启动的线程中发送消息
2. 在主线程中获取、处理消息

Android的消息处理有三个核心类：Looper、Handler和Message，其实还有一个Message Queue（消息队列），但是MQ被封装在Looper里面了。

**Handler消息的发送：**

通常我们在调用Handler的sendEmptyMessage时候调用了下面的函数：

**public** **boolean** sendMessageAtTime(Message msg, **long** uptimeMillis)

{

**boolean** sent = **false**;

MessageQueue queue = mQueue;

**if** (queue != **null**) {

**msg.target = this;//在这里定义了消息是由哪个Handler发出的**

sent = queue.enqueueMessage(msg, uptimeMillis);

}

**else** {

RuntimeException e = **new** RuntimeException(

**this** + " sendMessageAtTime() called with no mQueue");

Log.w("Looper", e.getMessage(), e);

}

**return** sent;

}

**Handler消息的处理：**

将普通线程变换为Looper线程需要调用Looper.prepare()与Looper.loop()

**public** **static** **void** loop() {

Looper me = myLooper();

**if** (me == **null**) {

**throw** **new** RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");

}

MessageQueue queue = me.mQueue;

// Make sure the identity of this thread is that of the local process,

// and keep track of what that identity token actually is.

Binder.clearCallingIdentity();

**final** **long** ident = Binder.clearCallingIdentity();

**while** (**true**) {

Message msg = queue.next(); // might block

**if** (msg != **null**) {

**if** (msg.target == **null**) {

// No target is a magic identifier for the quit message.

**return**;

}

**long** wallStart = 0;

**long** threadStart = 0;

// This must be in a local variable, in case a UI event sets the logger

Printer logging = me.mLogging;

**if** (logging != **null**) {

logging.println(">>>>> Dispatching to " + msg.target + " " +

msg.callback + ": " + msg.what);

wallStart = SystemClock.currentTimeMicro();

threadStart = SystemClock.currentThreadTimeMicro();

}

**msg.target.dispatchMessage(msg);//target实际上就是Handler，用于处理消息**

**if** (logging != **null**) {

**long** wallTime = SystemClock.currentTimeMicro() - wallStart;

**long** threadTime = SystemClock.currentThreadTimeMicro() - threadStart;

logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);

**if** (logging **instanceof** Profiler) {

((Profiler) logging).profile(msg, wallStart, wallTime,

threadStart, threadTime);

}

}

// Make sure that during the course of dispatching the

// identity of the thread wasn't corrupted.

**final** **long** newIdent = Binder.clearCallingIdentity();

**if** (ident != newIdent) {

Log.wtf(TAG, "Thread identity changed from 0x"

+ Long.toHexString(ident) + " to 0x"

+ Long.toHexString(newIdent) + " while dispatching to "

+ msg.target.getClass().getName() + " "

+ msg.callback + " what=" + msg.what);

}

msg.recycle();

}

}

}

**Handler的dispatchMessage函数：（调用handleMessage函数，即我们需要重写的）**

**public** **void** dispatchMessage(Message msg) {

**if** (msg.callback != **null**) {

handleCallback(msg);

} **else** {

**if** (mCallback != **null**) {

**if** (mCallback.handleMessage(msg)) {

**return**;

}

}

handleMessage(msg);

}

}

**Message类：**

在整个消息处理机制中，message又叫task，封装了任务携带的信息和处理该任务的handler，使用需要注意下面几点：

1. 尽管Message有public方法，但是应该通过Message.oain来从消息池中获取，节省资源
2. 如果你的message比较简单，优先使用Message.arg1和Message.arg2，这逼用Bundle更节省内存
3. 用Message.what标识信息，以便用不同方式处理message

**17 android的一个示例**

该Activity继承于ListActivity