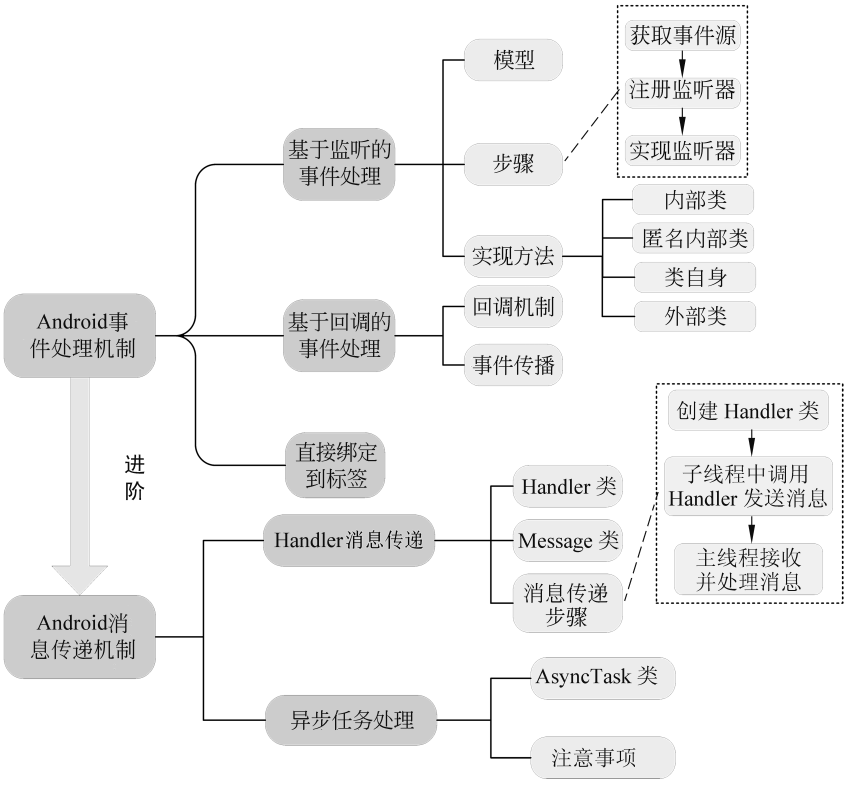
第3章

Android 事件处理

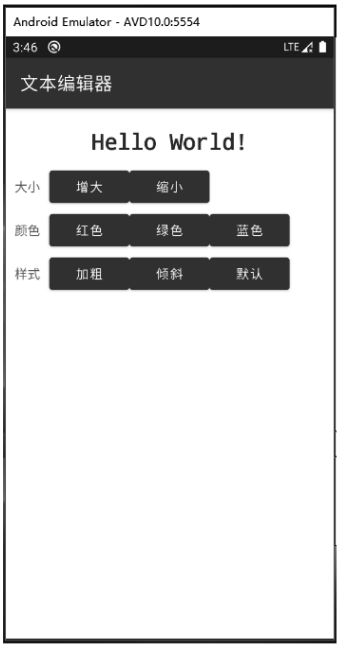
本章要点

- 基于监听的事件处理模型
- 实现事件监听器的 4 种方式
- 基于回调的事件处理模型
- 事件传播
- 事件直接绑定到标签
- Handler 消息传递机制
- AsyncTask 异步任务处理

本章知识结构图



本章示例



简单文本编辑器

文件下载提示进度条

前面学习了 Android 中提供的一些基本控件，将来在第 10 章还会介绍其他功能强大的界面控件，关于 Android 提供的其他控件，读者可以查找有关参考资料。但是，这些控件主要用来进行数据显示，如果用户想与之进行交互，实现具体的功能，则还需要相应事件处理的辅助。当用户在程序界面上执行各种操作，如单击一个按钮时，应用程序必须为用户动作提供响应动作，这种响应动作就需要通过事件处理来完成。

Android 中提供了三种事件处理方式：基于回调的事件处理、基于监听的事件处理和直接绑定到标签的事件处理。这三种事件处理方式各有优缺点和适合使用的场景。回调机制主要是重写一些系统中已定义好的方法，这些方法调用的时机是自动的，只是默认情况下什么都不做；监听机制则需要为控件绑定监听器，事件发生时将会执行相应的方法，对于开发人员来说更为灵活、自由、可控性较高。直接绑定到标签的机制则直接指定事件的处理方法，主要针对非常常见的单击事件，更为方便简单。Android 系统充分利用了三种事件处理的优点，允许开发者采用自己熟悉的事件处理方式来为用户操作提供响应。

在 Android 中，用户界面属于主线程，而子线程无法更新主线程的界面状态。那么，如何才能动态地显示用户界面呢？本章介绍通过 Handler 消息传递来动态更新界面。

如果在事件处理中需要做一些比较耗时的操作，直接放在主线程中将会阻塞程序的运行，给用户以不好的体验，甚至会造成程序没有响应或强制退出。本章将学习如何通过 AsyncTask 异步方式来处理耗时的操作。

学完本章之后，再结合前面所学知识，读者将可以开发出界面友好、人机交互良好的Android 应用。

3.1　Android 的事件处理机制

任何手机应用都离不开与用户的交互，只有通过用户的操作，才能知道用户的需求，从而实现具体的业务功能。因此，应用中经常需要处理的就是用户的操作，即为用户的操作提供响应，这种为用户操作提供响应的机制就是事件处理。

Android 提供了强大的事件处理机制，包括以下三种。

（1）基于监听的事件处理：主要做法是为 Android 界面控件绑定特定的事件监听器，在事件监听器的方法里编写事件处理代码，由系统监听用户的操作，一旦监听到用户事件，将自动调用相关方法来处理。

（2）基于回调的事件处理：主要做法是重写 Android 控件特定的回调方法，或者重写 Activity 的回调方法。Android 为绝大部分界面控件提供了事件响应的回调方法，只需重写它们即可，由系统根据具体情景自动调用。

（3）直接绑定到标签：主要做法是在界面布局文件中为指定标签设置事件属性，属性值是一个方法的方法名，然后再在 Activity 中定义该方法，编写具体的事件处理代码。

一般来说，直接绑定到标签只适合于少数指定的事件，非常方便；基于回调的事件处理代码比较简洁，可用于处理一些具有通用性的系统定义好的事件。但对于某些特定的事件，无法使用基于回调的事件处理，只能采用基于监听的事件处理。实际应用中，基于监听的事件处理方法应用最广泛。

3.1.1　基于监听的事件处理

Android 的基于监听的事件处理模型与 Java 的 AWT、Swing 的处理方式几乎完全一样，只是相应的事件监听器和事件处理方法名有所不同。在基于监听的事件处理模型中，主要涉及以下三类对象。

（1）**EventSource（事件源）**：产生事件的控件即事件发生的源头，如按钮、菜单等。

（2）**Event（事件）**：具体某一操作的详细描述，事件封装了该操作的相关信息，如果程序需要获得事件源上所发生事件的相关信息，一般通过 Event 对象来取得，例如按键事件按下的是哪个键、触摸事件发生的位置等。

（3）**EventListener（事件监听器）**：负责监听用户在事件源上的操作，并对用户的各种操作做出相应的响应。事件监听器中可包含多个事件处理器，一个事件处理器实际上就是一个事件处理方法。

那么在基于监听的事件处理中，这三类对象又是如何协作的呢？实际上，基于监听的事件处理是一种委托式事件处理。普通控件（事件源）将整个事件处理委托给特定的对象（事件监听器）；当该事件源发生指定的事件时，系统自动生成事件对象，并通知所委托的事件监听器，由事件监听器相应的事件处理器来处理这个事件。具体的事件处理模型如图 3-1 所示。当用户在 Android 控件上进行操作时，系统会自动生成事件对象，并将这个

事件对象以参数的形式传给注册到事件源上的事件监听器,事件监听器调用相应的事件处理器来处理。

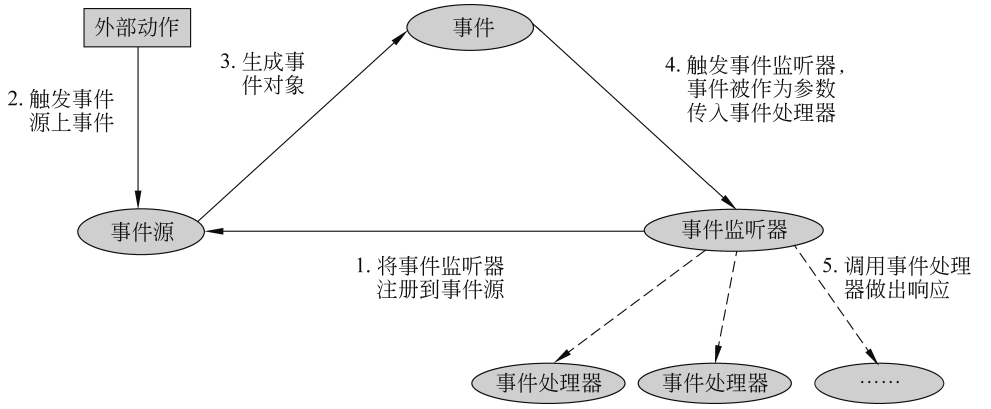


图 3-1　基于监听的事件处理模型

委托式事件处理非常好理解,就类似于生活中每个人能力都有限,**当碰到一些自己处理不了的事情时,就委托给某个机构或公司来处理**。委托人需要将遇到的事情和要求描述清楚,这样,其他人才能比较好地解决问题,然后该机构会选派具体的员工来处理这件事。其中,我们是**事件源**,**遇到的事情**就是**事件**,该**机构**就是**事件监听器**,**具体**解决事情的**员工就是事件处理器**。

基于监听的事件处理模型的主要编程步骤如下。

(1) 获取普通界面控件(事件源),也就是被监听的对象。

(2) 实现事件监听器类,该监听器类是一个特殊的 Java 类,必须实现一个 XxxListener 接口,并实现接口里的所有方法,每个方法用于处理一种事件。

(3) 调用事件源的 setXxxListener 方法将事件监听器对象注册给普通控件(事件源),即将事件源与事件监听器关联起来,这样,当事件发生时就可以自动调用相应的方法。

在上述步骤中,事件源比较容易获取,一般就是界面控件,根据 findViewById()方法即可得到;调用事件源的 setXxxListener 方法是由系统定义好的,只需要传入一个具体的事件监听器;所以,我们所要做的就是实现事件监听器。所谓事件监听器,其实就是实现了特定接口的 Java 类的实例。在程序中实现事件监听器,通常有如下几种形式。

(1) 内部类形式:将事件监听器类定义为当前类的内部类。

(2) 外部类形式:将事件监听器类定义成一个外部类。

(3) 类自身作为事件监听器类:让 Activity 本身实现监听器接口,并实现事件处理方法。

(4) 匿名内部类形式:使用匿名内部类创建事件监听器对象。

下面以一个简单的程序来示范基于监听的事件处理模型的实现过程。该程序实现简单文本编辑功能,可以控制文本颜色、大小、样式以及文本的内容,程序界面布局中定义了一些文本显示框和若干个按钮,并为所有的按钮注册了单击事件监听器,为测试文本编辑

框注册了长按事件监听器。为了演示各种实现事件监听器的方式，该程序中使用了 4 种实现事件监听器的方式。界面分析与运行效果如图 3-2 所示。

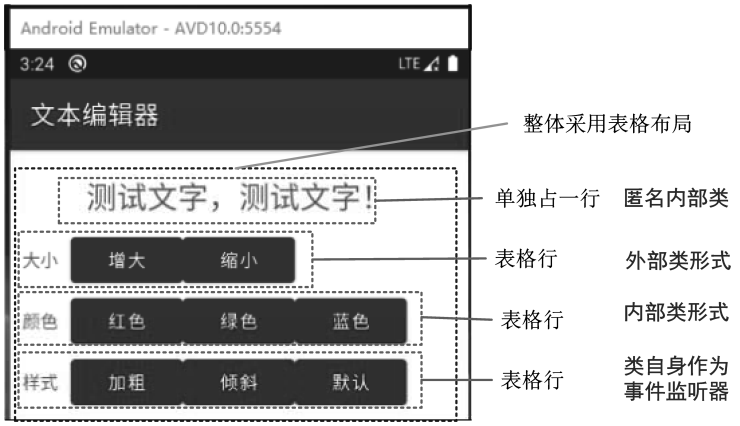


图 3-2　简单文本编辑器

界面整体采用表格布局，第一行仅包含一个 TextView 用于显示测试文本，第二行包含大小提示信息以及“增大”“缩小”两个按钮，第三行包含颜色提示信息以及“红色”“绿色”“蓝色”三个按钮，第四行包含样式提示信息以及“加粗”“倾斜”“默认”三个按钮。界面布局的详细代码如下。

程序清单：codes\ch03\TextEditorTest\app\src\main\res\layout\activity_main.xml

```
1  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
                                                          →表格布局
2      xmlns:tools="http://schemas.android.com/tools"
3      android:layout_width="match_parent"                →宽度为填充父容器
4      android:layout_height="match_parent"               →高度为填充父容器
5      android:padding="10dp">                            →内边距为 10dp
6      <TextView                                          →显示测试内容
7          android:id="@+id/testText"                     →添加 ID 属性
8          android:gravity="center"                       →内容居中显示
9          android:text="@string/test_text"               →测试文本
10         android:padding="10dp"                         →内边距为 10dp
11         android:textSize="24sp"/>                      →大小为 24sp
12     <TableRow                                          →表格行
13         android:layout_marginRight="10dp">             →右边距为 10dp
14         <TextView android:text="@string/size" />       →文本提示信息
15         <Button                                        →按钮
16             android:id="@+id/bigger"                   →为按钮添加 ID 属性
17             android:text="@string/bigger" />           →按钮上显示的文本
18         <Button                                        →按钮
19             android:id="@+id/smaller"                  →为按钮添加 ID 属性
```

```
20              android:text="@string/smaller" />          →按钮上显示的文本
21      </TableRow>                                        →表格行结束
22      <TableRow>……</TableRow>                            →省略颜色相关内容
23      <TableRow>……</TableRow>                            →省略样式相关内容
24  </TableLayout>                                         →表格布局结束
```

在该布局文件中，省略了一些类似的代码，保留了整体结构。界面设计完成后运行程序，得到上述界面效果，但此时单击按钮时没有任何反应，下面为这些按钮添加事件监听器。

首先为“红色”“绿色”“蓝色”三个按钮添加事件监听器，这里采用内部类的形式实现事件监听器，关键代码如下。

```
1   public class MainActivity extends AppCompatActivity{
2       private Button red, green, blue;                          →声明按钮成员变量
3       private TextView testText;                                →声明文本框变量
4       public void onCreate(Bundle savedInstanceState) {         →重写父类方法
5           super.onCreate(savedInstanceState);                   →调用父类方法
6           setContentView(R.layout.activity_main);               →设置界面布局文件
7           testText =(TextView) findViewById(R.id.testText);     →根据 ID 找到控件
8           red =(Button) findViewById(R.id.red);                 →根据 ID 找到控件
9           green =(Button) findViewById(R.id.green);             →根据 ID 找到控件
10          blue =(Button) findViewById(R.id.blue);               →根据 ID 找到控件
11          ColorListner myColorListner =new ColorListner();      →创建监听器对象
12          red.setOnClickListener(myColorListner);               →注册监听器
13          green.setOnClickListener(myColorListner);             →注册监听器
14          blue.setOnClickListener(myColorListner);              →注册监听器
15      }
16      private class ColorListner implements OnClickListener {  →定义内部类实现单
                                                                    击事件监听器接口
17          public void onClick(View v) {
18              switch (v.getId()) {                              →判断事件源
19              case R.id.red:                                    →如果是“红色”按钮
20                  testText.setTextColor(Color.RED); break;      →将字体设置为红色
21              case R.id.blue:                                   →如果是“蓝色”按钮
22                  testText.setTextColor(Color.BLUE); break;     →将字体设置为蓝色
23              case R.id.green:                                  →如果是“绿色”按钮
24                  testText.setTextColor(Color.GREEN); break;    →将字体设置为绿色
25              default: break;                                   →默认什么都不做
26              }                                                 →判断结束
27          }                                                     →单击事件方法结束
28      }                                                         →内部类结束
29  }                                                             →外部类结束
```

使用内部类作为事件监听器有以下两个优势。

(1) 可以在当前类中复用该监听器类，即多个事件源可以注册同一个监听器。

(2) 可以自由访问外部类的所有界面控件,内部类实质上是外部类的成员。

内部类形式比较适合于有多个事件源同时注册同一事件监听器的情形。

下面为“增大”和“缩小”按钮添加事件监听器,这里采用外部类的形式实现事件监听器,关键代码如下。

```
1   public class MainActivity extends  AppCompatActivity {
2       private Button bigger,smaller;                          →声明按钮成员变量
3       public void onCreate(Bundle savedInstanceState) {
4           ...
5           bigger =(Button) findViewById(R.id.bigger);         →根据 ID 找到控件
6           smaller =(Button) findViewById(R.id.smaller);       →根据 ID 找到控件
7           SizeListener mysizeListener=new SizeListener(testText);
                                                                →创建监听器对象
8           bigger.setOnClickListener(mysizeListener);          →注册监听器
9           smaller.setOnClickListener(mysizeListener);         →注册监听器
10      }                                                       →方法结束
11  }                                                           →类结束
```

SizeListener 是一个外部类,该类实现了 OnClickListener 接口,可以处理单击事件,但外部类无法获取到 Activity 里的界面控件,也就不能对控件进行设置和更新,那么如何在该类中获取到需要改变的控件呢? 在这里采用通过构造方法传入的方式。SizeListener 的代码如下。

程序清单: codes\ch03\TextEditorTest\app\src\main\java\iet\jxufe\cn\texteditortest \SizeListener.java

```
1   public class SizeListener implements View.OnClickListener {  →类的声明
2       private TextView tv;                                    →成员变量声明
3       public SizeListener(TextView tv) {                      →构造方法
4           this.tv =tv;                                        →初始化成员变量
5       }                                                       →构造方法结束
6       public void onClick(View v) {                           →单击事件处理方法
7           float f=tv.getTextSize();                           →获取当前的字体大小
8           switch (v.getId()) {                                →判断是增大还是缩小
9               case R.id.bigger:                               →如果是增大
10                  f=f+2;                                      →字体每次增大 2
11                  break;                                      →退出 switch
12              case R.id.smaller:                              →如果是缩小
13                  f=f-2;                                      →字体每次减小 2
14                  break;                                      →退出 switch
15              default:                                        →默认什么都不做
16                  break;                                      →退出 switch
17          }                                                   →判断结束
18          if(f<=8){                                           →判断字体是否小于 8
19              f=8;                                            →设置最小字体为 8
```

```
20          }
21          tv.setTextSize(TypedValue.COMPLEX_UNIT_PX,f);→设置字体大小
22      }                                                →单击事件方法结束
23  }                                                    →类结束
```

注意：调用 setTextSize()设置字体时，最好指定单位，如果不指定单位，则在不同的模拟器上显示效果会有所不同，甚至会出现单击缩小反而出现变大的效果。这是因为 getTextSize()方法获取的字体大小单位是 px，而默认的 setTextSize()方法设置的字体大小单位为 sp，对于不同密度的模拟器，sp 和 px 的转换关系不同。

使用外部类作为事件监听器类的形式较为少见，主要有如下两个原因。

(1) 事件监听器通常属于特定的 GUI(图形用户界面)，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 界面中的控件，编程不够简洁。

但如果某个事件监听器确实需要被多个 GUI 界面所共享，而且主要是完成某种业务逻辑的实现，则可以考虑使用外部类的形式来定义事件监听器类。

接着为“加粗”“倾斜”“默认”三个按钮添加事件处理器，这里采用 Activity 类本身实现用 OnClickListener 接口作为事件监听器，代码如下。

```
1   public class MainActivity extends AppCompatActivity implements View
    .OnClickListener
2       private Button bold, italic,normal;                →声明按钮成员变量
3       private boolean isItalic=false,isBold=false;       →是否加粗、倾斜标记
4       public void onCreate(Bundle savedInstanceState) {
5           …
6           testText.setTypeface(Typeface.DEFAULT);        →设置字体样式
7           bold=(Button)findViewById(R.id.bold);          →根据 ID 获取控件
8           italic =(Button) findViewById(R.id.italic);    →根据 ID 获取控件
9           normal =(Button)findViewById(R.id. normal);    →根据 ID 获取控件
10          italic.setOnClickListener(this);               →注册监听器
11          bold.setOnClickListener(this);                 →注册监听器
12          moren.setOnClickListener(this);                →注册监听器
13      }
14      public void onClick(View v) {
15          switch (v.getId()) {                           →判断哪个按钮被单击
16              case R.id.italic:                          →如果单击的是“倾斜”
17                  isItalic=!isItalic;                    →更换倾斜的状态
18                  break;                                 →退出 switch
19              case R.id.bold:                            →如果单击的是“加粗”
20                  isBold=!isBold;                        →更换加粗的状态
21                  break;                                 →退出 switch
22              case R.id.moren:                           →如果单击的是“默认”
23                  isItalic=false;                        →默认不倾斜
```

```
24                  isBold=false;                                   →默认不加粗
25                  break;                                          →退出 switch
26              default:
27                  break;
28          }                                                       →判断结束
29          if(isItalic){                                           →如果是倾斜
30              if(isBold){                                         →倾斜且加粗
31                  testText.setTypeface(Typeface.MONOSPACE,Typeface.BOLD_ITALIC);
32              }else{                                              →倾斜不加粗
33                  testText.setTypeface(Typeface.MONOSPACE,Typeface.ITALIC);
34              }
35          }else{                                                  →不倾斜
36              if(isBold){                                         →不倾斜但加粗
37                  testText.setTypeface(Typeface.MONOSPACE,Typeface.BOLD);
38              }else{                                              →不倾斜不加粗
39                  testText.setTypeface(Typeface.MONOSPACE,Typeface.NORMAL);
40              }
41          }
42      }
43  }
```

由于 Activity 自身可以充当事件监听器，因此为事件源注册监听器时，只需要将当前对象传入即可，而不用单独创建一个监听器对象。由于加粗和倾斜两种样式可以进行叠加，因此，需要定义两个 boolean 类型标记表示当前是否加粗和是否倾斜。如果当前是加粗状态再次单击加粗将会取消加粗，如果当前是倾斜状态再次单击倾斜将会取消倾斜。因此最终样式状态有四种：正常状态（既不加粗也不倾斜）、加粗不倾斜、倾斜不加粗、既加粗也倾斜。

Activity 类本身作为事件监听器，就如同生活中，我们自己刚好能够处理某一件事，不需要委托给他人处理。可以直接在 Activity 类中定义事件处理器方法，这种形式非常简洁，但也有两个缺点。

（1）可能造成程序结构混乱，Activity 的主要职责是完成界面初始化工作，但此时还需包含事件处理器方法，从而引起混乱。

（2）如果 Activity 界面类需要实现监听器接口，将会导致 Activity 类中代码增多，结构混乱，类的设计不符合高内聚、低耦合的原则，不是很规范。

思考：在上面的程序中，单击事件监听器的具体事件处理器，并没有接收到事件参数，即我们并没有发现事件的“踪迹”，这是为什么呢？这是因为 Android 对事件监听模型做了进一步简化：如果事件源触发的事件足够简单、事件里封装的信息比较有限，则无须封装事件对象。而对于键盘事件、触摸事件等，程序需要获取事件发生的详细信息，如由键盘中的哪个键触发事件，触摸所发生的位置等。对于这种包含更多信息的事件，Android 会将事件信息封装成 XxxEvent 对象，然后传递给事件监听器。

最后，为测试文本框添加长按事件监听器，采用匿名内部类的形式来实现该监听器，

具体代码如下。

```
1  public class MainActivity extends Activity{
2      …
3      public void onCreate(Bundle savedInstanceState) {
4          …
5          testText.setOnLongClickListener(new View.OnLongClickListener() {
6              @Override
7              public boolean onLongClick(View v) {
8                  AlertDialog.Builder builder=new AlertDialog.Builder
   (MainActivity.this);
9                  builder.setTitle("请输入新的内容");
10                 builder.setIcon(R.mipmap.ic_launcher);
11                 final EditText contentText=new EditText(MainActivity.this);
12                 builder.setView(contentText);
13                 builder.setPositiveButton("确定",new DialogInterface
   .OnClickListener(){
14                     @Override
15                     public void onClick(DialogInterface dialog, int which) {
16                         testText.setText(contentText.getText().toString());
17                     }
18                 });
19                 builder.setNegativeButton("取消",null);
20                 builder.create().show();
21                 return false;
22     }
23 }
```

当用户长按测试文本时,将会弹出一个对话框,提示用户输入新的内容,用户输入完成后,单击“确定”按钮,将会改变测试文本的内容,效果如图 3-3 所示。关于对话框的相关知识请查看第 10 章相关介绍,在此只是简单使用。

注意:contentText 应定义为 MainActivity 的成员变量或者 final 修饰的局部变量,否则无法在匿名内部类中访问该变量。

本例中,既有普通按钮的单击事件处理,也有对话框中按钮的单击事件处理,二者的事件监听器的接口名都为 OnClickListener,但又不是同一个类,它们位于不同的包中,完整的类名分别为 android.view.View.OnClickListener 和 android.content.DialogInterface.OnClickListener。对于这种在同一个类中需要使用多个具有相同的类名而又位于不同包中的类时,通常只能导入一个类,其他的类需要使用完整的包名+类名来进行访问。本例中使用 View.OnClickListener 和 DialogInterface.OnClickListener 进行区分,不能简单地缩写成 OnClickListener。

大部分时候,事件处理器都没有太大的复用价值(可复用代码通常都被抽象成了业务逻辑方法),因此大部分事件监听器只是临时使用一次,所以使用匿名内部类形式的事件监听器更合适。实际上,**这种形式也是目前使用最广泛的事件监听器形式**。

图 3-3　文本框长按事件处理效果

Android 中常见事件监听器接口及其处理方法如表 3-1 所示。

表 3-1　常见事件监听器接口及其处理方法

事件	接　口	处 理 方 法	描　述
单击事件	View.OnClickListener	public abstract void onClick (View v)	单击控件时触发
长按事件	View.OnLongClickListener	public abstract booleanon LongClick (View v)	长按控件时触发
键盘事件	View.OnKeyListener	public abstract boolean onKey (View v, int keyCode, KeyEvent event)	处理键盘事件
焦点事件	View.OnFocusChangeListener	public abstract void onFocusChange (View v, boolean hasFocus)	当焦点发生改变时触发
触摸事件	View.OnTouchListener	public abstract boolean onTouch (View v, MotionEvent event)	产生触摸事件
创建上下文菜单	View.OnCreateContextMenu Listener	public abstract void OnCreateContextMenu (ContextMenu menu, View v, ContextMenu. ContextMenuInfo menuInfo)	当上下文菜单创建时触发

事件监听器要与事件源关联起来，还需要相应注册方法的支持，事件源通常是界面的某个控件，而所有的界面控件都继承于 View 类，因此，View 类所拥有的事件注册方法，所有的控件都可以调用，表 3-2 列出了 View 类常见的事件注册方法。

表 3-2　View 类的常见事件注册方法

方　　法	类型	描　　述
public void setOnClickListener(View.OnClickListener l)	普通	注册单击事件
public void setOnLongClickListener(View.OnLongClickListener l)	普通	注册长按事件
public void setOnKeyListener(View.OnKeyListener l)	普通	注册键盘事件
public void setOnFocusChangeListener(View.OnFocusChangeListener l)	普通	注册焦点改变事件
public void setOnTouchListener(View.OnTouchListener l)	普通	注册触摸事件
public void setOnCreateContextMenuListener (View.OnCreateContextMenuListener l)	普通	注册上下文菜单事件

3.1.2　基于回调的事件处理

Android 平台中,每个 View 都有自己处理特定事件的回调方法,开发人员可以通过重写 View 中的这些回调方法来实现需要的响应事件。View 类包含的回调方法主要有如下几种。

(1) boolean onKeyDown (int keyCode, KeyEvent event):它是接口 KeyEvent.Callback 中的抽象方法,用于捕捉手机键盘被按下的事件。keyCode 为被按下的键值即键盘码,event 为按键事件的对象,包含了触发事件的详细信息,如事件的状态、类型、发生时间等。当用户按下按键时,系统会自动将事件封装成 KeyEvent 对象供应用程序使用。

(2) boolean onKeyUp (int keyCode, KeyEvent event):用于捕捉手机键盘按键抬起的事件。

(3) boolean onTouchEvent (MotionEvent event):该方法在 View 类中定义,用于处理手机屏幕的触摸事件,包括屏幕被按下、屏幕被抬起、在屏幕中拖动。

如果说事件监听机制是一种委托式的事件处理,那么回调机制则与之相反。在基于回调的事件处理模型中,事件源和事件监听器是统一的,或者说事件监听器完全消失了,当用户在 GUI 控件上激发某个事件时,控件自己特定的方法将负责处理该事件。回调机制所对应的方法都是系统已定义好的,调用时机也是系统设计的,只是默认情况下该方法内部什么都没做。开发人员需要做的就是重写该方法,做自己的业务逻辑处理。

下面以一个简单的程序来示范基于回调的事件处理机制。由于需要重写控件类的回调方法,因此通过自定义 View 来模拟,自定义 View 时,重写该 View 的事件处理方法即可。本例中定义一个自定义类 MyButton 从系统中的 Button 继承,然后重写了 Button 类的 onTouchEvent(MotionEvent event)方法来处理按钮上的触摸事件,当用户按下按钮时弹出一个 Toast 信息,运行效果如图 3-4 所示。

自定义按钮的关键代码如下。

图 3-4　文本框长按事件处理效果

程序清单：codes\ch03\CallBackEventTest\app\src\main\java\iet\jxufe\cn\callbackeventtest\MyButton.java

```
1  public class MyButton extends Button {                          →类的声明
2      private Context context;                                    →成员变量声明
3      public MyButton(Context context, AttributeSet attrs) {      →构造方法中必须要有
                                                                     AttributeSet 参数
4          super(context, attrs);                                  →调用父类构造方法
5          this.context=context;                                   →成员变量初始化
6      }                                                           →构造方法结束
7      @Override                                                   →注解表示重写方法
8      public boolean onTouchEvent(MotionEvent event) {            →重写触摸回调方法
9          if(event.getAction()==MotionEvent.ACTION_DOWN){         →如果是按下事件
10             Toast.makeText(context, "按钮被单击了", Toast.LENGTH_SHORT).
   show();                                                         →弹出消息
11         }                                                       →判断结束
12         return true;                                            →返回结果
13     }                                                           →方法结束
14 }                                                               →类结束
```

注意：自定义控件时必须提供构造方法，如果想要在布局文件中使用自定义控件，则构造方法中一定要传递 AttributeSet 类型参数。

Toast 类的 makeText()方法用于指定弹出信息，需传递三个参数：第一个参数表示上下文对象，通常为当前 Activity；第二个参数为字符串，表示弹出信息的内容；第三个参数表示弹出信息停留的时间，Toast 类中提供了两个常量 Toast.LENGTH_SHORT 和 Toast.LENGTH_LONG，分别表示时间长一点和短一点。默认情况下，弹出信息并不会

显示,需要调用 show()方法使其显示。

在布局文件中添加该控件,由于不是系统中自带的控件,需要使用完整的包名+类名,关键代码如下。

```
1   < RelativeLayout  xmlns: android =" http://schemas. android. com/apk/res/
    android"                                              →相对布局
2       xmlns:tools="http://schemas.android.com/tools"
3       android:layout_width="match_parent"               →宽度填充父容器
4       android:layout_height="match_parent">             →高度填充父容器
5       <iet.jxufe.cn.callbackeventtest.MyButton          →使用自定义控件,完整包名+
                                                            类名
6           android:layout_width="wrap_content"           →宽度内容包裹
7           android:layout_height="wrap_content"          →高度内容包裹
8           android:text="自定义按钮" />                    →按钮内容
9   </RelativeLayout>                                     →相对布局结束
```

几乎所有基于回调的事件处理方法都有一个 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。如果处理事件的回调方法返回 true,表明该处理方法已完全处理该事件,该事件不会传播出去;如果处理事件的回调方法返回 false,表明该处理方法并未完全处理该事件,该事件会传播出去。

对于基于回调的事件传播而言,某控件上所发生的事情不仅会激发该控件上的回调方法,也会触发该控件所在 Activity 的回调方法(**前提是事件能传播到** Activity)。

假设同一控件既采用监听模式,又采用回调模式,并且重写了该控件所在 Activity 对应的回调方法,而且程序没有阻止事件传播,即每个方法都返回为 false,那么 Android 系统处理该控件事件的顺序是怎样的呢?

下面以一个简单的例子来模拟这种情况。为上面自定义的按钮注册触摸事件监听器并重写它所在 Activity 上的触摸回调方法,在每个方法中打印出该方法被调用的信息,观察控制台里打印的信息。自定义控件代码如下。

程序清单:codes\ch03\ EventTransferTest\app\src\main\java\iet\jxufe\cn\eventtransfertest\MyButton.java

```
1   public class MyButton extends Button {                    →类的声明
2       public MyButton(Context context, AttributeSet attrs) {  →构造方法
3           super(context, attrs);                            →调用父类构造方法
4       }                                                     →构造方法结束
5       @Override                                             →注解表示重写方法
6       public boolean onTouchEvent(MotionEvent event) {      →重写触摸回调方法
7           if(event.getAction()==MotionEvent.ACTION_DOWN){   →如果是,按下事件
8               System.out.println("MyButton 中的事件处理触发了!");
                                                              →控制台打印信息
9           }                                                 →判断结束
10          return false;                  →返回结果为 false,表示事件可以向外传播
```

```
11      }                                                    →方法结束
12  }                                                        →类结束
```

新的布局文件代码如下。

程序清单：codes\ch03\EventTransferTest\app\src\main\res\layout\activity_main.xml

```
1   <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
    android"                                                 →相对布局
2       xmlns:tools="http://schemas.android.com/tools"
3       android:layout_width="match_parent"                  →宽度填充父容器
4       android:layout_height="match_parent">                →高度填充父容器
5       <iet.jxufe.cn.eventtransfertest.MyButton             →使用自定义控件,完整
                                                               包名+类名
6           android:layout_width="wrap_content"              →宽度内容包裹
7           android:layout_height="wrap_content"             →高度内容包裹
8           android:id="@+id/myBtn"                          →添加 ID 属性
9           android:text="自定义按钮" />                      →按钮内容
10  </RelativeLayout>                                        →相对布局结束
```

在 Java 代码中根据 ID 找到该控件，然后为其注册触摸监听器，同时重写 Activity 的触摸事件回调方法，关键代码如下。

程序清单：codes\ch03\EventTransferTest\app\src\main\java\iet\jxufe\cn\eventtransfertest\MainActivity.java

```
1   public class MainActivity extends AppCompatActivity {     →类的声明
2       private MyButton myButton;                           →成员变量声明
3       @Override                                            →注解表示重写方法
4       protected void onCreate(Bundle savedInstanceState) { →重写父类方法
5           super.onCreate(savedInstanceState);              →调用父类方法
6           setContentView(R.layout.activity_main);          →加载布局文件
7           myButton=(MyButton)findViewById(R.id.myBtn);     →根据 ID 找到控件
8           myButton.setOnTouchListener(new View.OnTouchListener() {
                                                             →注册触摸事件监听器
9               @Override                                    →注解表示重写方法
10              public boolean onTouch(View v, MotionEvent event) {  →触摸方法
11            if(event.getAction()==MotionEvent.ACTION_DOWN){ →判断是否为按下
12              System.out.println("监听器中的事件处理触发了!");→控制台打印信息
13            }                                              →判断结束
14                  return false;                            →返回结果
15          }                                                →方法结束
16          });                                              →监听器结束
17      }                                                    →方法结束
18      @Override
19      public boolean onTouchEvent(MotionEvent event) {     →回调方法
```

```
20          if(event.getAction()==MotionEvent.ACTION_DOWN){ →判断是否为按下
21              System.out.println("Activity 中的事件处理触发了!");
                                                          →控制台打印信息
22          }
23          return false;
24      }
25  }
```

单击按钮观察控制台打印信息,在 Android Studio 底部有一个 Logcat 选项,单击即可查看打印的信息,结果如图 3-5 所示。

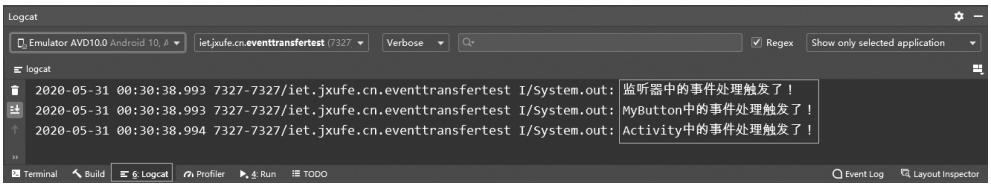


图 3-5　控制台打印信息

通过打印结果,可知**最先**触发的是该控件所绑定的**事件监听器**,**接着**才触发该**控件**提供的**事件回调方法**,**最后**才传播到该控件所在的 Activity,调用 Activity **相应的事件回调方法**。如果使某一个事件处理方法返回 true,那么该事件将不会继续向外传播。

试一试:改变方法的返回值(将 true 改为 false),观察控制台输出结果。

基于监听的事件处理模型分工更明确,事件源、事件监听由两个类分开实现,因此具有更好的可维护性;Android 的事件处理机制保证基于监听的事件监听器会被优先触发。除了 View 类有回调方法之外,Android 系统提供的一些组件类也都有回调方法,例如经常使用的 Activity 的 onCreate()方法,以及后面要介绍的菜单创建以及菜单项的事件处理,都采用了回调机制。

3.1.3　直接绑定到标签

Android 还有一种更简单直观的事件处理方式,即直接在界面布局文件中为指定标签添加属性绑定事件处理方法,主要用于处理单击事件。可以为界面控件标签添加 onClick 属性,该属性值是一个形如 xxx(View source)方法的方法名。例如在布局文件中为控件添加单击事件的处理方法,布局文件如下所示。

程序清单:codes\ch03\EventBindingTest\app\src\main\res\layout\activity_main.xml

```
1   < RelativeLayout  xmlns: android =" http://schemas. android. com/apk/res/
    android"
2       xmlns:tools="http://schemas.android.com/tools"
3       android:layout_width="match_parent"
4       android:layout_height="match_parent">
5       <Button
6           android:id="@+id/mBtn"
```

```
7          android:layout_width="wrap_content"
8          android:layout_height="wrap_content"
9          android:onClick="clickEventHandler"        →为按钮添加事件处理方法
10         android:text="直接绑定到标签" />
11 </RelativeLayout>
```

然后在该界面布局对应的 Activity 中定义一个 public void clickEventHandler (View view)方法,该方法将会负责处理该按钮上的单击事件。关键代码如下。

```
1  public void clickEventHandler(View view){                    →方法声明
2          Toast.makeText(this, "绑定到标签的事件处理执行了!", Toast.LENGTH_
   SHORT).show();                                               →弹出消息
3  }                                                            →方法结束
```

注意:*方法名必须与 onClick 属性值一致,否则会因为找不到相应的方法而导致程序强制退出。因此,为了避免拼写错误,开发时建议直接复制属性值。*

方法声明时不能使用 private 修饰,否则无法访问,一般建议使用 public,或省略不写。方法声明中包含一个 View 类型的参数,该参数表示当前被单击的控件,实际上可以为多个控件指定同一个单击事件处理方法,通过 View 参数就可以知道当前被单击的具体是哪一个控件。

如果此时为该按钮同时添加了事件监听器,那么执行结果如何呢? 为上述按钮添加 ID 属性,然后根据 ID 找到控件,为其注册单击事件监听器,完整代码如下。

程序清单:codes\ch03\EventBindingTest\app\src\main\java\iet\jxufe\cn\eventbindingtest\MainActivity.java

```
1  public class MainActivity extends AppCompatActivity {
2      private Button mBtn;
3      @Override
4      protected void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          setContentView(R.layout.activity_main);
7          mBtn=(Button)findViewById(R.id.mBtn);                →根据 ID 找到控件
8          mBtn.setOnClickListener(new View.OnClickListener() {→注册事件监听器
9              public void onClick(View v) {                    →事件处理方法
10                 Toast.makeText(MainActivity.this,"监听的事件处理执行
   了!",Toast.LENGTH_SHORT).show();                             →弹出消息
11             }                                                →处理方法结束
12         });                                                  →事件监听器结束
13     }                                                        →方法结束
14     public void clickEventHandler(View view){                →绑定事件方法
15         Toast.makeText(this, "绑定到标签的事件处理执行了!",
   Toast.LENGTH_SHORT).show();                                  →弹出消息
16     }
```

```
17  }                                                    →类结束
```

执行程序,结果是程序只执行监听事件处理,而不会执行我们自定义的事件处理方法。注意这和 3.1.2 节中基于回调的事件传播有所不同。单击事件方法返回值是 void 而不是 boolean 类型。

3.2 Handler 消息传递机制

除了用于响应用户操作的事件处理,实际应用中还有另一种事件:周期性变化的事件,例如希望每隔一段时间自动更新或者跳转页面等。涉及周期性变化就需要计时,也就涉及子线程操作。而在 Android 中,界面控件是非线程安全的,所谓非线程安全,是指当多个线程对其进行操作时,结果可能会不一致。为了避免出现这种情况,Android 中明确规定,所有对界面的操作只能放在主线程中,不能在子线程中更改界面控件。这样就陷入了一种矛盾:子线程想更改界面显示,但无法更改;主线程能更改界面显示但不知道更改时机。这时候就需要借助一定的中介使得二者进行交互。因此,Android 中的 Handler 消息传递机制应运而生,它为子线程与主线程之间协同工作搭建了桥梁。当子线程需要更改界面显示时,通过 Handler 发送一条消息,主线程接收到消息后,即可实时更改界面显示。Handler 类的常用方法如表 3-3 所示。

表 3-3 Handler 类的常用方法

方法签名	描　述
public void handleMessage (Message msg)	通过该方法获取并处理信息
public final boolean sendEmptyMessage (int what)	发送一个只含有标记的消息
public final boolean sendMessage (Message msg)	发送一个具体的消息
public final boolean hasMessages (int what)	监测消息队列中是否有指定标记的消息
public final boolean post (Runnable r)	将一个线程添加到消息队列中

从以上方法可以看出,Handler 类主要用于发送、接收和处理消息。执行过程为:在子线程中,当需要对界面进行操作时,通过 Handler 发送消息;消息一旦发送成功,将会回调 Handler 类的 handleMessage(Message msg)方法,由于该方法在主线程中,因此能够对界面执行更改操作。Handler 消息传递机制可以归纳为:谁发送谁处理,需要时发送消息,消息处理自动执行。

由于处理消息的 handleMessage(Message msg)方法是一种回调方法,当 Handler 接收到消息时,由系统自动调用,因此,通常创建 Handler 对象时,需要重写该方法,在该方法中写入相关的业务逻辑。由于一个 Handler 对象可以发送多个消息,因此接收时要判断消息的类别,然后针对不同的消息做不同的处理。

开发带有 Handler 类的程序步骤如下。

(1) 创建 Handler 类对象,并重写 handleMessage()方法。

(2) 在新启动的线程中,调用 Handler 对象的发送消息方法。

(3) 利用 Handler 对象的 handleMessage()方法接收消息,然后根据不同的消息执行不同的操作。

下面以一个简单的示例讲解如何通过 Handler 实现子线程与主线程的协同工作,程序运行效果为每隔 3s 自动更换界面背景,如图 3-6 所示。布局文件较为简单,仅包含一个相对布局,没有其他子控件,在此不给出界面布局代码。

图 3-6　程序运行两个瞬间的效果截图

该程序的核心业务逻辑代码如下。

程序清单：codes\ch03\HandlerTest\app\src\main\java\iet\jxufe\cn\handlertest\MainActivity.java

```
1  public class MainActivity extends AppCompatActivity {
2      private RelativeLayout root;
3      private int[] colors={Color.RED,Color.BLUE,Color.GREEN,
                  Color.YELLOW,Color.MAGENTA};        →定义一个数组用于保存颜色
4      private int currentIndex=0;                          →当前颜色的下标
5      private Handler mHandler;                            →声明 Handler 对象
6      protected void onCreate(Bundle savedInstanceState) { →重写父类方法
7          super.onCreate(savedInstanceState);              →调用父类方法
8          setContentView(R.layout.activity_main);          →加载布局文件
9          root=(RelativeLayout)findViewById(R.id.root);    →根据 ID 找到控件
10         mHandler=new Handler(){                          →创建 Handler 对象
11             public void handleMessage(Message msg) {     →重写父类方法
12                 if(msg.what==0x11){                      →判断消息标记
13                     currentIndex=(currentIndex+1)%colors.length;
                                                            →改变颜色下标
14                     root.setBackgroundColor(colors[currentIndex]);
```

```
                                                          →更改背景颜色
15                  }
16              }
17          };
18          start();                                      →调用方法
19      }
20      private void start(){                             →自定义方法
21          new Thread(){                                 →创建线程
22              public void run() {                       →线程执行体方法
23                  while(true){                          →死循环
24                      try {
25                          Thread.sleep(3000);           →休眠 3s
26                          mHandler.sendEmptyMessage(0x11); →发送空消息
27                      } catch (InterruptedException e) {  →捕获异常
28                          e.printStackTrace();          →打印异常信息
29                      }
30                  }
31              }
32          }.start();                                    →启动线程
33      }
34  }
```

该程序首先创建了一个 Handler 对象,然后自定义了一个方法 start()用于启动线程,线程一旦启动执行死循环,每次休眠 3s 后发送一条消息,主线程将会接收到消息,然后重写 Handler 类的 handleMessage()方法来处理消息。处理消息的业务逻辑是让背景颜色依次循环变化,在此定义一个数组用于保存所有的颜色,然后定义一个变量保存当前颜色的下标,每次变化时让下标往后移一位,即加 1。需注意的是,如果是最后一种颜色,再加 1 将会导致数组下标越界。在此指定,最后一种颜色的下一个颜色为第一个颜色,这样循环显示,所以每次下标的变化为:currentIndex=(currentIndex+1)%colors.length,这样下标永远不会越界。

注意:发送消息和处理消息的是同一个 Handler 对象,线程创建完成后一定要调用它的 start()方法启动线程。

3.3 异步任务处理

在开发 Android 应用时经常会涉及一些耗时操作,例如访问网络、下载资源等,如果放在主线程中将会阻塞主线程,给用户造成卡顿,停在页面中无法操作,用户体验非常不好。因此,通常将耗时的操作放在单独的线程中执行,但是在子线程中操作主线程(UI 线程)会出现错误。因此 Android 提供了一个类 Handler,通过发送消息实现子线程与主线程协同工作,这样解决了子线程更新 UI 的问题。

费时的任务操作总会启动一些匿名的子线程,给系统带来巨大的负担,随之带来一些性