android手势识别

Last edited by **caoquanli** 1 month ago

一、概述

Android对于手势的检测，可以分为两种形式，一种形式是重写Activity、View中的onTouchEvent方法，根据自定义的手势的特征，组合MotionEvent中的动作常量，对于实现复杂的手势有一定难度，但灵活性较强，用户可以根据自己的需要设计自己的手势；如果应用使用常用的手势，比如双击，长按，滚动等，我们可以使用GestureDetector类，使用GestureDetector类可以轻松地检测常见的手势，而不必自己处理单个触摸事件，对于GestureDetector类的使用下文将作具体介绍，无论采用哪一种方法都需要先对MotionEvent类进行了解。

二、MotionEvent类

运动事件描述了动作的动作常量和一些列的坐标值。动作常量表明了当触点按下或者弹起等引起的状态变化，MotionEvent中常用的动作常量如表所示。坐标值描述了位置信息以及以他的运动属性。例如，当用户第一次触摸屏幕的时候，系统给窗体发出一个触摸事件，动作常量为ACTION_DOWN，并提供了一些列的坐标值，比如触摸的X、Y坐标，接触区域的压力、尺寸、方向等信息。

动作常量	常量值	说明
ACTION_DOWN	0	当屏幕检测到第一个触点按下之后就会触发到这个事件
ACTION_UP	1	单点触摸抬起时被触发
ACTION_MOVE	2	手指移动时触发
ACTION_CANCEL	3	触摸动作取消，不是由用户直接触发
ACTION_OUTSIDE	4	触摸动作超出边界
ACTION_POINTER_DOWN	5	多点触摸，非origin按下
ACTION_POINTER_UP	6	多点触摸，非origin抬起

MotionEvent类提供了许多可以查看触点的位置或者其他信息的方式，比如getX(int)、getY(int)、getAxisValue(int)、getPointerId(int)、getToolType(int)。这其中的大部分方法都将触点的索引值作为参数而不是触点的id。在事件中，每个触点的索引号的取值范围是从0到getPointerCount()-1，下面具体说明MotionEvent中的常用方法。

- **getX()/getY()：** 获得事件发生时,触摸的中间区域的X/Y坐标，由这两个函数获得的X/Y值是相对坐标，相对于消费这个事件的视图的左上角的坐标。
- **getRawX()/getRawY()：** 由这两个函数获得的X/Y值是绝对坐标，是相对于屏幕的。
- **getDownTime()：** 按下开始时间
- **getEventTime()** 事件结束时间
- **getActionMasked()：** 多点触控获取经过掩码后的动作类型
- **getPointerCount()：** 获取触控点的数量，比如2则可能是两个手指同时按压屏幕
- **getPointerId(nID)** 对于每个触控的点的细节，我们可以通过一个循环执行getPointerId方法获取索引
- **getDeviceId()** 获取设备编号，该编号会在开机、重连等中断后发生改变。

三、触摸事件处理机制

Android针对UI事件，提供了两种处理机制：一种是基于监听方式的处理机制，另一种是基于回调方式的处理机制。对于基于监听的处理方式，主要是为Android 界面组件绑定特定的事件监听器；对于基于回调的处理方式，主要的做法是通过为Android 组件重写特定的回调方法实现。下面我们详细介绍二者具体模型、流程、使用方法。

3.1 基于监听的事件处理机制

基于监听的事件处理方式实际是一种委托式（代理）模式，某个组件（事件源）将整个事件的处理委托给特定的对象（绑定在该组件上的事件监听器），由这个特定的对象来进行该事件的响应。 事件监听处理模型涉及到三类对象：

- 事件源-事件产生的地方，一般就是界面组件。
- 事件-事件封装了界面组件上的一次用户操作，如果程序需要获取界面组件上所发生事件的相关信息，就可以通过Event对象来获取。
- 事件监听器-包含事件处理方法，负责监听事件源所发生的事情，并对各种事件做出响应。

事件监听机制中由事件源，事件，事件监听器三类对象组成 处理流程如下:

- Step 1:为某个事件源(组件)设置一个监听器,用于监听用户操作
- Step 2:用户的操作,触发了事件源的监听器
- Step 3:生成了对应的事件对象
- Step 4:将这个事件源对象作为参数传给事件监听器
- step 5:事件监听器对事件对象进行判断,执行对应的事件处理器(对应事件的处理方法)

3.2 常见的事件监听器实现方式

在事件处理模型三个重要组成部分中，事件由系统负责生成、任意界面组件都可作为事件源，而事件监听器是整个事件处理核心。因此我们主要的工作就是实现事件监听器。所谓实现事件监听器其实就是实现特定接口的Java类实例，常用的方法包括：

1、匿名内部类: 目前是使用最广泛的事件监听器形式，大多数时候，事件处理器都没有复用价值，因此大部分事件监听器只是临时用一次，所以使用匿名内部类形式的事件监听器更适合。示例代码如下：

```
btnshow.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        //具体处理逻辑
    }
});
```

2、内部类实现: 内部类实现事件监听器的优点是可以在当前外部类中复用该监听器类、监听器类是外部类的内部类，可以自由访问外部类的所有界面组件。示例代码如下：

```
class BtnClickListener implements View.OnClickListener {
    @Override
    public void onClick(View v) {
        //具体处理逻辑
    }
}
```

3、外部类实现： 使用外部类定义事件监听器的形式通常用的比较少，使用外部类实现的缺点是事件监听器属于特定GUI，定义成外部类不利于提高程序内聚性、外部类形式的事件监听器不能自由访问创建GUI界面的类中组件，编程不够简洁。但是如果某个事件监听器确实需要被多个GUI界面所共享，而且主要是完成了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。

```
public class MyClick implements OnClickListener {
    @Override
    public void onClick(View v) {
          //具体处理逻辑
    }
}
```

4、Activity实现: 直接在Activity中实现监听器接口，这种方式虽然简单，但是却使得Activity类的职能混乱，Activity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，违反面向对象单职能原则。

```
public class MainActivity extends Activity implements OnClickListener{
    private Button btnshow;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnshow = (Button) findViewById(R.id.btnshow);
        btnshow.setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
         //具体处理逻辑
```

```
    }
}
```

3.3 基于回调的事件处理

与监听的委托式事件处理不一样，基于回调事件处理模型，事件源和事件监听器是统一的。用户在组件上激发某个事件，组件自己特定的方法将会负责处理该事件。通常实现的方法是通过继承组件类，重写相关的事件处理方法。 为了实现回调机制的事件处理，Android为所有的GUI组件提供了一些事件处理的方法，以View为例，该类包含如下方法：

- **boolean onKeyDown(int keyCode,KeyEvent keyEvent):** 当用户在该组件上按下某个按键时触发该方法。
- **boolean onKeyLongPress(int keyCode,KeyEvent Event):** 当用户在该组件上长按某个按键时触发该方法。
- **boolean onKeyShortcut(int keyCode,KeyEvent event):** 当一个键盘快捷键事件触发时触发该方法。
- **boolean onKeyUp(int keyCode,KeyEvent event):** 当用户在该组件上松开某个按键时触发该方法。
- **boolean onTouchEvent(MotionEvent event):** 当用户在该组件上触发触摸屏事件时触发该方法。
- **boolean onTrackballEvent(MotionEvent event):** 当用户在该组件上触发轨迹球事件时触发该方法。

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之后，该事件交给事件监听器负责处理；对于基于回调的事件处理模型来说，事件源和事件监听器是统一的，当事件源发生特定事件之后，该事件还是由事件源本身负责处理。

四、GestureDetector详解

Gesture手势检测用于辅助检测用户单击行为、滑动、长按、双击等行为，是对以上触摸事件的封装和补充，下面做具体介绍： GestureDetector对外提供了三个接口和一个内部类：

- 接口：OnGestureListener、OnDoubleTapListener、OnContextClickListener
- 内部类：SimpleOnGestureListener

该内部类实现了三个接口，包含了接口里所有必须实现的函数而且都已经重写，但方法体是空的，且该类是静态类，我们可以在外边继承该类，根据需要重写里面的方法；除此之外，我们还可以新建一个类继承此以上三个接口，但必须重写里面的全部方法，相对来说用起来不是很方便。 下面我们对常用的两种接口OnGestureListener、OnDoubleTapListener和内部类：SimpleOnGestureListener做进一步理解。

4.1 OnGestureListener使用方法

第一步：实现OnGestureListener或OnDoubleTapListener接口中的方法，这里有两种方法匿名内部类和实现了接口的类实例

```
private class gesturelistener implements GestureDetector.OnGestureListener{

    public boolean onDown(MotionEvent e) {
        // TODO Auto-generated method stub
        return false;
    }

    public void onShowPress(MotionEvent e) {
        // TODO Auto-generated method stub

    }

    public boolean onSingleTapUp(MotionEvent e) {
        // TODO Auto-generated method stub
        return false;
    }

    public boolean onScroll(MotionEvent e1, MotionEvent e2,
            float distanceX, float distanceY) {
        // TODO Auto-generated method stub
        return false;
    }

    public void onLongPress(MotionEvent e) {
```

```
        // TODO Auto-generated method stub

    }

    public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX,
            float velocityY) {
        // TODO Auto-generated method stub
        return false;
    }

}
```

第二步：创建GestureDetector类的实例，有以下几种形式：

```
public GestureDetector(OnGestureListener listener, Handler handler) {
    this(null, listener, handler);
}
```

```
public GestureDetector(OnGestureListener listener) {
    this(null, listener, null);
}
```

```
public GestureDetector(Context context, OnGestureListener listener) {
    this(context, listener, null);
}
```

```
public GestureDetector(Context context, OnGestureListener listener, Handler handler)
```

```
public GestureDetector(Context context, OnGestureListener listener, Handler handler,boolean
    this(context, listener, handler);
}
```

第三步：当一个特定触摸事件发生后，GestureDetector.OnGestureListener接口就会通知到用户，为了使GestureDetector对象接受事件，我们必须重写View或Activity中的onTouchEvent方法，将所有观察到的事件传递给监听器实例

```
@Override
public boolean onTouch(View v, MotionEvent event) {
    return mGestureDetector.onTouchEvent(event);
}
```

第四步 调用控件的View.setOnTouchListener()将接口的具体实现的引用传递进去或者如果是监听双击的话调用GestureDetector.setOnDoubleTapListener()

```
super.setOnTouchListener(this);
```

```
mGestureDetector.setOnDoubleTapListener(new MyGestureListener());
```

OnDoubleTapListener监听器和OnGestureListener监听器用法一样，在此不再赘述。

4.2 SimpleOnGestureListener

当我们写一个类并实现OnGestureListener，里面的方法不管用不用到，都必须对其进行重写，如果只用到了其中的几种手势，我们可以创建一个类使其继承GestureDetector.SimpleOnGestureListener类。

```
public class MainActivity extends Activity {

    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new MyGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event){
```

```
        this.mDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }

    class MyGestureListener extends GestureDetector.SimpleOnGestureListener {
        private static final String DEBUG_TAG = "Gestures";

        @Override
        public boolean onDown(MotionEvent event) {
            Log.d(DEBUG_TAG,"onDown: " + event.toString());
            return true;
        }

        @Override
        public boolean onFling(MotionEvent event1, MotionEvent event2,
                float velocityX, float velocityY) {
            Log.d(DEBUG_TAG, "onFling: " + event1.toString() + event2.toString());
            return true;
        }
    }
}
```