事件

Android提供了两套强大的事件处理机制。

1. 基于监听的事件处理；
2. 基于回调的事件处理；

基于监听的事件处理就是在android的组件上绑定特定的监听器，而基于回调的事件处理就是重写UI组件或者Activity的回调方法。

基于回调的事件处理用于处理一些具有通用性的事件，基于监听的事件处理用于处理与具体业务相关的事件。

事件处理模型

事件处理模型中，主要涉及三类对象：

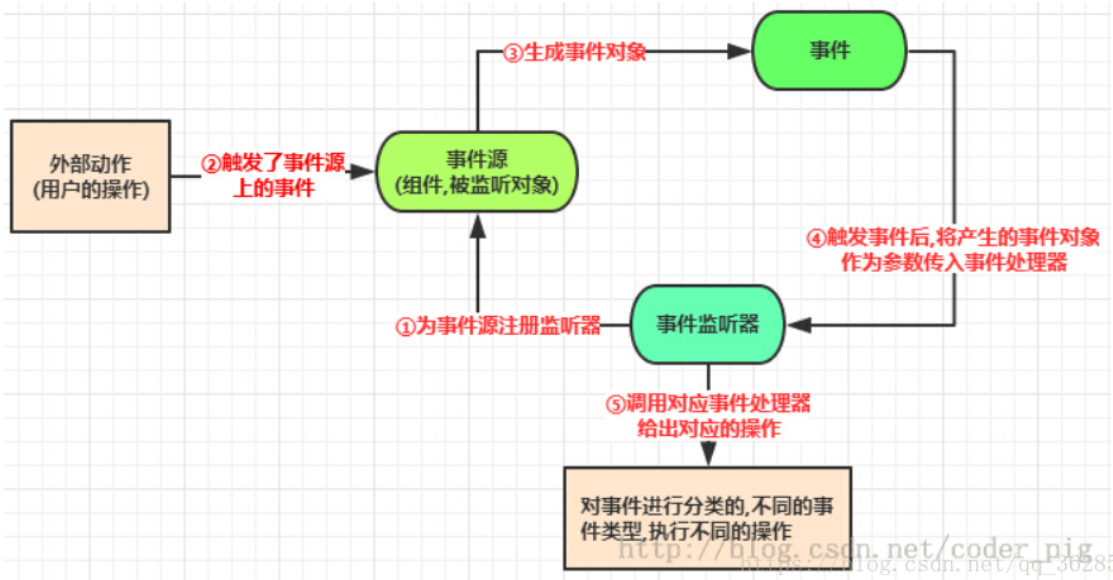
1.EventSource（事件源）。事件发生的场所，通常就是各个组件，比如按钮、窗口、菜单。

2.Event（事件）。事件封装了事件发生的相关信息。

3.EventListener（事件监听器）。监听事件源发生的事件，并对事件作出相应的响应。

基于回调的事件处理模型就是将EventSource和EventListener合二为一了，即事件源也是事件监听器（处理器）。

基于监听的事件处理



内部类作为事件监听器

优势：

1. 使用内部类可以在当前类中复用该监听器类；
2. 可以自由的访问外部类的所有界面组件；

外部类作为事件的监听器

缺点：

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. 外部类的形式不能自由的访问创建GUI界面的类中的组件，编程不够简洁；

Activity本身作为事件的监听器类

典型的是Activity implements OnClickListener；

Lamnda表达式作为事件监听器类

基于回调的事件处理

所谓基于回调的事件处理机制是指事件源和事件处理程序统一了，当事件发生时，直接调用事件源相关的方法完成具体事件处理。针对View对象，Android提供了很多默认的事件处理方法，例如onTouchEvent、onKeyDown等，当我们自定义View时，只需要重新这些方法，就可以按照自己的业务逻辑去完成具体的事件处理。

Handler消息传递机制

简介

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；

Handler类包含如下方法用于发送、处理消息。

- handleMessage（Message msg）:处理消息的方法。该方法用于被重写。
- hasMessages(int what)：检查消息队列中是否包含what属性为指定值得消息。
- hasMessages(int what,Object object):检查消息队列中是否包含what属性为指定值且object属性性为指定对象的消息。
- 多个重载的Message obtainMessage（）：获取消息；
- sendEmptyMessage（int what）:发送空消息；
- sendEmptyMessageDelayed(int what,long delayMillis):指定多少毫秒之后发送空消息；
- sendMessage(Message msg):立即发送消息；
- sendMessageDelayed(Message msg,long delayMillis):指定多少毫秒之后发送消息；

Handler、Loop、MessageQueue的工作原理

- **Message**：Handler接收和处理的消息对象；
- **Looper**：每个线程只能有一个Looper。它的loop方法负责读取MeesageQueue中的消息，读到信息之后就把消息交给发送该消息的Handler进行处理。
- **MessageQueue**：消息队列，他采用先进先出的方式来管理Message。程序创建Looper对象时，会在它的构造器中创建MessageQueue对象。Looper的构造器源代码如下：

```
private Looper(){
    mQueue = new MessageQueue();
    mRun = true;
    mThread = Thread.currentThread();
}
```

该构造器使用了private修饰，表明程序员无法通过构造器创建Looper对象。从上面的代码不难看出，程序初始化Looper时会创建一个与之关联的MessageQueue，这个MessageQueue就负责管理消息。

- Handler：它的作用有两个，即发送消息和处理消息，程序使用Handler发送消息，有Handler发送的消息必须被送到指定的MessageQueue。也就是说，如果希望Handler正常工作，必须在当前线程中有一个MessageQueue；否则消息就没有MessageQueue进行保存了。不过MessageQueue是由Looper负责管理的，也就是说，如果希望Hnadler正常工作，必须在当前的线程中有一个Looper对象。为了保证当前线程中有Looper对象，可以分如下两种情况处理。

  1. 在UI线程中，系统已经初始化了一个Looper对象，因此程序直接创建Handler即可，然后就可通过Handler来发送、处理消息了。
  2. 程序员自己启动的子线程，必须自己创建一个Looper对象，并启动它。创建Looper对象调用它的prepare()方法即可。

  prepare()方法保证每个线程最多只有一个Looper对象。prepare()方法的源代码如下：

```
public static final void prepare(){
    if(sThreadLocal.get() !=null){
        throw new RuntimeException("Only one Looper may be created per
thread")
    }
    sThreadLocal.set(new Looper());
}
```

接下来调用Looper的静态loop()方法来启动它。loop()方法使用一个死循环不断去取MessageQueue中的消息，并将取出的消息分给对应的Handler进行处理。下面是Looper类的loop()方法的源代码：

```
for(;;){
    Message msg = queue.next(); //获取消息队列中的下一个消息，如果没有消息，将会阻塞
    if(msg == null){
        //如果消息为null，表明消息队列正在退出
        return;
    }
    Printer logging = me.mLogging;
    if(logging != null){
        logging.println(">>>>> Dispathing to " + msg.target + "" +
msg.callback + ":" + msg.what);
    }
    msg.target.dispatchMessage(msg);
    if(logging != null){
        logging.println("<<<<< Finished to " + msg.target + "" +
msg.callback);
        //使用final修饰该标识符，保证在分发消息的过程中线程标识符不会被修改
        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();
            }
        }
    }
```

归纳起来，Looper、MessageQueue、Handler各自的作用如下：

- **Looper**：每个线程只有一个Looper，他负责管理MessageQueue，会不断的从MessageQueue中取出消息，并将消息分给对应的Handler处理。
- **MessageQueue**：由Looper负责管理。它采用先进先出的方式来管理Message。
- **Handler**：它能把消息发送给Looper管理的MessageQueue，并负责处理Looper分给它的消息。

在线程中使用Handler的步骤如下：

1. 调用Looper的prepare()方法为当前线程创建Looper对象，创建Looper对象时，它的构造器会创建与之配套的MessageQueue。

2. 有了Looper之后，创建Handler子类的实例，重写handlerMessage()方法，该方法负责处理来自其他线程的消息。
3. 调用Looper的loop方法启动Looper。

异步任务（AsyncTask）

为了解决在新线程（非UI线程）不能更新UI组件的问题，Android提供了如下几种解决方案。

- 使用Handler实现线程之间的通信；
- Activity.runOnUiThread(Runnable);
- View.post(Runable);
- View.postDelayed(Runnable,long);

以上方法导致编程略显繁琐；

`AsyncTask<Params,Progress,Result>`是一个抽象类，通常用于被继承，继承AsyncTask时需要指定如下三个泛型参数。

- Params：启动任务执行的输入参数的类型；
- Progress：后台任务完成的进度值的类型；
- Result：后台执行任务完成后返回结果的类型；

使用AsyncTask只要如下三步即可

1. 创建AsyncTask的子类，并为三个泛型参数指定类型。如果某个泛型不需要指定类型，则可将它指定为void
2. 根据需要，实现AsyncTask的如下方法。
    - doInBackground(Params...):重写该方法就是后台线程将要完成的任务。该方法可以调用publishProgress(Progress... values)方法更新任务执行进度。
    - onProgressUpdate(Progress... values):在doInBackground()方法中调用publishProgress（）方法执行进度后，将触发该方法。

    - onPreExecute():该方法将在执行后台耗时操作前被调用。通常该方法用于完成一些初始化的准备工作，比如界面上显示进度条等；
    - onPostExecute（Result result）：当doInBackground（）完成后，系统会自动调用onPostExecute（）方法，并将doInBackground（）方法的返回值传给该方法；
3. 调用AsyncTask子类的实例的execute（Params ...params）开始执行耗时任务；

使用AsyncTask时必须遵守如下规则

- 必须在UI线程中创建AsyncTask实例；
- 必须在UI线程中调用AsyncTask的execute方法；
- AsyncTask的onPreExecute、onPostExecute（Result result）、doInBackground(Params...)、onProgressUpdate(Progress... values)方法不应该有程序员代码调用，而是有Android系统负责调用。
- 每个AsyncTask只能被执行一次，多次调用将会引发异常；