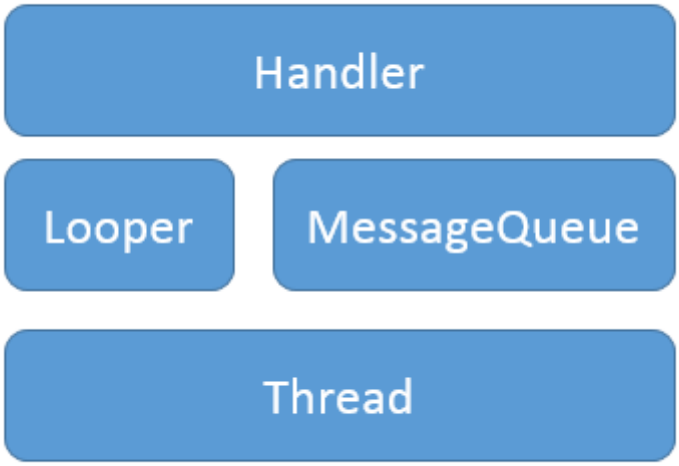
概括来说，Handler是Android中引入的一种让开发者参与处理线程中消息循环的机制。我们在使用Handler的时候与Message打交道最多，Message是Hanlder机制向开发人员暴露出来的相关类，可以通过Message类完成大部分操作Handler的功能。但作为程序员，我不能只知道怎么用Handler，还要知道其内部如何实现的。Handler的内部实现主要涉及到如下几个类: Thread、MessageQueue和Looper。这几类之间的关系可以用如下的图来简单说明：



Thread是最基础的，Looper和MessageQueue都构建在Thread之上，Handler又构建在Looper和MessageQueue之上，我们通过Handler间接地与下面这几个相对底层一点的类打交道。

MessageQueue

最基础最底层的是Thread，每个线程内部都维护了一个消息队列——MessageQueue。消息队列MessageQueue，顾名思义，就是存放消息的队列（好像是废话...）。那队列中存储的消息是什么呢？假设我们在UI界面上单击了某个按钮，而此时程序又恰好收到了某个广播事件，那我们如何处理这两件事呢？ 因为一个线程在某一时刻只能处理一件事情，不能同时处理多件事情，所以我们不能同时处理按钮的单击事件和广播事件，我们只能挨个对其进行处理，只要挨个处理就要有处理的先后顺序。 为此Android把UI界面上单击按钮的事件封装成了一个Message，将其放入到MessageQueue里面去，即将单击按钮事件的Message入栈到消息队列中，然后再将广播事件的封装成以Message，也将其入栈到消息队列中。也就是说一个Message对象表示的是线程需要处理的一件事情，消息队列就是一堆需要处理的Message的池。线程Thread会依次取出消息队列中的消息，依次对其进行处理。MessageQueue中有两个比较重要的方法，一个是enqueueMessage方法，一个是next方法。enqueueMessage方法用于将一个Message放入到消息队列MessageQueue中，next方法是从消息队列MessageQueue中阻塞式地取出一个Message。在Android中，消息队列负责管理着顶级程序对象（Activity、BroadcastReceiver等）以及由其创建的所有窗口。需要注意的是，消息队列不是Android平台特有的，其他的平台框架也会用到消息队列，比如微软的MFC框架等。

Looper

消息队列MessageQueue只是存储Message的地方，真正让消息队列循环起来的是Looper，这就好比消息队列MessageQueue是个水车，那么Looper就是让水车转动起来的河水，如果没有河水，那么水车就是个静止的摆设，没有任何用处，Looper让MessageQueue动了起来，有了活力。

Looper是用来使线程中的消息循环起来的。默认情况下当我们创建一个新的线程的时候，这个线程里面是没有消息队列MessageQueue的。为了能够让线程能够绑定一个消息队列，我们需要借助于Looper：首先我们要调用Looper的prepare方法，然后调用Looper的Loop方法。典型的代码如下所示：

```
class LooperThread extends Thread {
      public Handler mHandler;

      public void run() {
          Looper.prepare();

          mHandler = new Handler() {
              public void handleMessage(Message msg) {
                  // process incoming messages here
              }
          };

          Looper.loop();
      }
  }
```

需要注意的是Looper.prepare()和Looper.loop()都是在新线程的run方法内调用的，这两个方法都是静态方法。我们通过查看Looper的源码可以发现，Looper的构造函数是private的，也就是在该类的外部不能用new Looper()的形式得到一个Looper对象。根据我们上面的描述，我们知道线程Thread和Looper是一对一绑定的，也就是一个线程中最多只有一个Looper对象，这也就能解释Looper的构造函数为什么是private的了，我们只能通过工厂方法Looper.myLooper()这个静态方法获取当前线程所绑定的Looper。

Looper通过如下代码保存了对当前线程的引用：

```
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
```

所以在Looper对象中通过sThreadLocal就可以找到其绑定的线程。ThreadLocal中有个set方法和get方法，可以通过set方法向ThreadLocal中存入一个对象，然后可以通过get方法取出存入的对象。ThreadLocal在new的时候使用了泛型，从上面的代码中我们可以看到此处的泛型类型是Looper，也就是我们通过ThreadLocal的set和get方法只能写入和读取Looper对象类型，如果我们调用其ThreadLocal的set方法传入一个Looper，将该Looper绑定给了该线程，相应的get就能获得该线程所绑定的Looper对象。

我们再来看一下Looper.prepare()，该方法是让Looper做好准备，只有Looper准备好了之后才能调用Looper.loop()方法，Looper.prepare()的代码如下：

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

上面的代码首先通过sThreadLocal.get()拿到线程sThreadLocal所绑定的Looper对象，由于初始情况下sThreadLocal并没有绑定Looper，所以第一次调用prepare方法时，sThreadLocal.get()返回null，不会抛出异常。重点是下面的代码**sThreadLocal.set(new Looper(quitAllowed))**，首先通过私有的构造函数创建了一个Looper对象的实例，然后通过sThreadLocal的set方法将该Looper绑定到sThreadLocal中。
这样就完成了线程sThreadLocal与Looper的双向绑定：
a. 在Looper内通过sThreadLocal可以获取Looper所绑定的线程；
b.线程sThreadLocal通过sThreadLocal.get()方法可以获取该线程所绑定的Looper对象。

上面的代码执行了Looper的构造函数，我们看一下其代码：

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

我们可以看到在其构造函数中实例化一个消息队列MessageQueue，并将其赋值给其成员字段mQueue，这样Looper也就与MessageQueue通过成员字段mQueue进行了关联。

在执行完了Looper.prepare()之后，我们就可以在外部通过调用Looper.myLooper()获取当前线程绑定的Looper对象。
myLooper的代码如下所示：

```
public static Looper myLooper() {
        return sThreadLocal.get();
}
```

需要注意的是，在一个线程中，只能调用一次Looper.prepare()，因为在第一次调用了Looper.prepare()之后，当前线程就已经绑定了Looper，在该线程内第二次调用Looper.prepare()方法的时候，sThreadLocal.get()会返回第一次调用prepare的时候绑定的Looper，不是null，这样就会走的下面的代码throw new RuntimeException(“Only one Looper may be created per thread”)，从而抛出异常，告诉开发者一个线程只能绑定一个Looper对象。

在调用了Looper.prepare()方法之后，当前线程和Looper就进行了双向的绑定，这时候我们就可以调用Looper.loop()方法让消息队列循环起来了。
需要注意的是Looper.loop()应该在该Looper所绑定的线程中执行。

Looper.loop()的代码如下:

```
public static void loop() {
        final Looper me = myLooper();
        if (me == null) {
            throw new RuntimeException("No Looper; Looper.prepare() wasn't
called on this thread.");
        }
        //注意下面这行
        final 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();

        //注意下面这行
        for (;;) {
            //注意下面这行
            Message msg = queue.next(); // might block
            if (msg == null) {
                // No message indicates that the message queue is quitting.
                return;
            }

            // 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);
            }

            //注意下面这行
            msg.target.dispatchMessage(msg);

            if (logging != null) {
                logging.println("<<<<< Finished to " + msg.target + " " +
msg.callback);
            }

            // 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.recycleUnchecked();
        }
}
```

上面有几行代码是关键代码:

1. **final MessageQueue queue = me.mQueue;**：变量me是通过静态方法myLooper()获得的当前线程所绑定的Looper，me.mQueue是当前线程所关联的消息队列。
2. **for (;;)**：我们发现for循环没有设置循环终止的条件，所以这个for循环是个死循环。
3. **Message msg = queue.next(); // might block**：我们通过消息队列MessageQueue的next方法从消息队列中取出一条消息，如果此时消息队列中有Message，那么next方法会立即返回该Message，如果此时消息队列中没有Message，那么next方法就会**阻塞式**地等待获取Message。
4. **msg.target.dispatchMessage(msg);**：msg的target属性是Handler，该代码的意思是让Message所关联的Handler通过dispatchMessage方法让Handler处理该Message，关于Handler的dispatchMessage方法将会在下面详细介绍。

Handler

Handler是暴露给开发者最顶层的一个类，其构建在Thread、Looper与MessageQueue之上。

Handler具有多个构造函数，签名分别如下所示:

1. publicHandler()
2. publicHandler(Callbackcallback)
3. publicHandler(Looperlooper)
4. publicHandler(Looperlooper, Callbackcallback)

第1个和第2个构造函数都没有传递Looper，这两个构造函数都将通过调用Looper.myLooper()获取当前线程绑定的Looper对象，然后将该Looper对象保存到名为mLooper的成员字段中。

第3个和第4个构造函数传递了Looper对象，这两个构造函数会将该Looper保存到名为mLooper的成员字段中。

第2个和第4个构造函数还传递了Callback对象，Callback是Handler中的内部接口，需要实现其内部的handleMessage方法，Callback代码如下:

```
public interface Callback {
        public boolean handleMessage(Message msg);
}
```

Handler.Callback是用来处理Message的一种手段，如果没有传递该参数，那么就应该重写Handler的handleMessage方法，也就是说为了使得Handler能够处理Message，我们有两种办法:

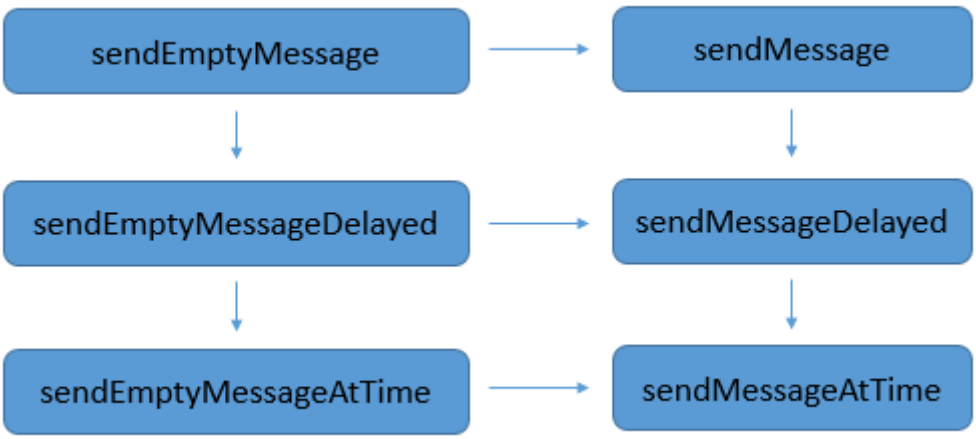
1. 向Hanlder的构造函数传入一个Handler.Callback对象，并实现Handler.Callback的handleMessage方法
2. 无需向Hanlder的构造函数传入Handler.Callback对象，但是需要重写Handler本身的handleMessage方法 也就是说无论哪种方式，我们都得通过某种方式实现handleMessage方法，这点与Java中对Thread的设计有异曲同工之处。 在Java中，如果我们想使用多线程，有两种办法:
3. 向Thread的构造函数传入一个Runnable对象，并实现Runnable的run方法
4. 无需向Thread的构造函数传入Runnable对象，但是要重写Thread本身的run方法

所以只要用过多线程Thread，应该就对Hanlder这种需要实现handleMessage的两种方式了然于心了。

我们知道通过sendMessageXXX系列方法可以向消息队列中添加消息，我们通过源码可以看出这些方法的调用顺序， sendMessage调用了sendMessageDelayed，sendMessageDelayed又调用了sendMessageAtTime。

Handler中还有一系列的sendEmptyMessageXXX方法，而这些sendEmptyMessageXXX方法在其内部又分别调用了其对应的sendMessageXXX方法。

通过以下调用关系图我们可以看的更清楚些:



由此可见所有的sendMessageXXX方法和sendEmptyMessageXXX最终都调用了sendMessageAtTime方法。

我们再来看看postXXX方法，会发现postXXX方法在其内部又调用了对应的sendMessageXXX方法，我们可以查看下sendMessage的源码:

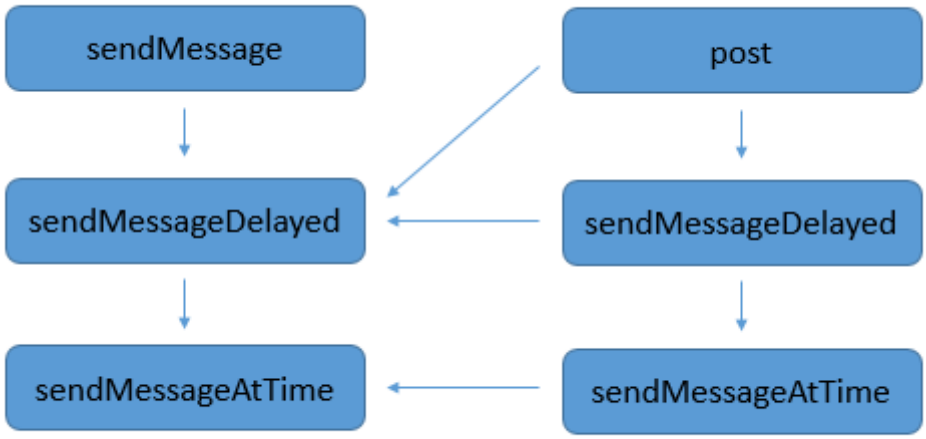
```
public final boolean post(Runnable r)
{
       return  sendMessageDelayed(getPostMessage(r), 0);
}
```

可以看到内部调用了**getPostMessage**方法，该方法传入一个Runnable对象，得到一个Message对象，getPostMessage的源码如下：

```
private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
 }
```

通过上面的代码我们可以看到在getPostMessage方法中，我们创建了一个Message对象，并将传入的Runnable对象赋值给Message的callback成员字段，然后返回该Message，然后在post方法中该携带有Runnable信息的Message传入到sendMessageDelayed方法中。由此我们可以看到所有的postXXX方法内部都需要借助sendMessageXXX方法来实现，所以postXXX与sendMessageXXX并不是对立关系，而是postXXX依赖sendMessageXXX，所以postXXX方法可以通过sendMessageXXX方法向消息队列中传入消息，只不过通过postXXX方法向消息队列中传入的消息都携带有Runnable对象（Message.callback）。

我们可以通过如下关系图看清楚postXXX系列方法与sendMessageXXX方法之间的调用关系：



通过分别分析sendEmptyMessageXXX、postXXX方法与sendMessageXXX方法之间的关系，我们可以看到在Handler中所有可以直接或间接向消息队列发送Message的方法最终都调用了sendMessageAtTime方法，该方法的源码如下：

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
        MessageQueue queue = mQueue;
        if (queue == null) {
            RuntimeException e = new RuntimeException(
                    this + " sendMessageAtTime() called with no mQueue");
            Log.w("Looper", e.getMessage(), e);
            return false;
        }
        //注意下面这行代码
        return enqueueMessage(queue, msg, uptimeMillis);
}
```

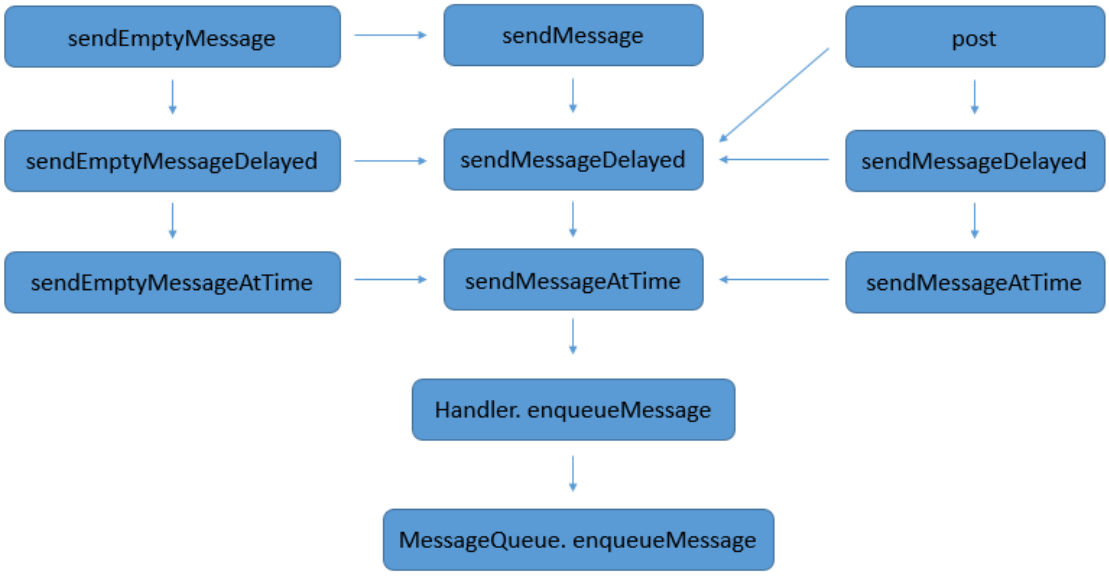
该方法内部调用了**enqueueMessage**方法，该方法的源码如下：

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long
uptimeMillis) {
        //注意下面这行代码
        msg.target = this;
        if (mAsynchronous) {
            msg.setAsynchronous(true);
        }
        //注意下面这行代码
        return queue.enqueueMessage(msg, uptimeMillis);
}
```

在该方法中有两件事需要注意：

1. **msg.target = this** 该代码将Message的target绑定为当前的Handler
2. **queue.enqueueMessage** 变量queue表示的是Handler所绑定的消息队列MessageQueue，通过调用queue.enqueueMessage(msg, uptimeMillis)我们将Message放入到消息队列中。

所以我们通过下图可以看到完整的方法调用顺序：



我们在分析Looper.loop()的源码时发现，Looper一直在不断的从消息队列中通过MessageQueue的next方法获取Message，然后通过代码**msg.target.dispatchMessage(msg)**让该msg所绑定的Handler（Message.target）执行dispatchMessage方法以实现对Message的处理。

Handler的dispatchMessage的源码如下：

```
public void dispatchMessage(Message msg) {
        //注意下面这行代码
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
             //注意下面这行代码
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
             //注意下面这行代码
            handleMessage(msg);
        }
```

```
}
```

我们来分析下这段代码：

1.首先会判断msg.callback存不存在，msg.callback是Runnable类型，如果msg.callback存在，那么说明该Message是通过执行Handler的postXXX系列方法将Message放入到消息队列中的，这种情况下会执行handleCallback(msg), handleCallback源码如下：

```
private static void handleCallback(Message message) {
        message.callback.run();
}
```

这样我们我们就清楚地看到我们执行了msg.callback的run方法，也就是执行了postXXX所传递的Runnable对象的run方法。

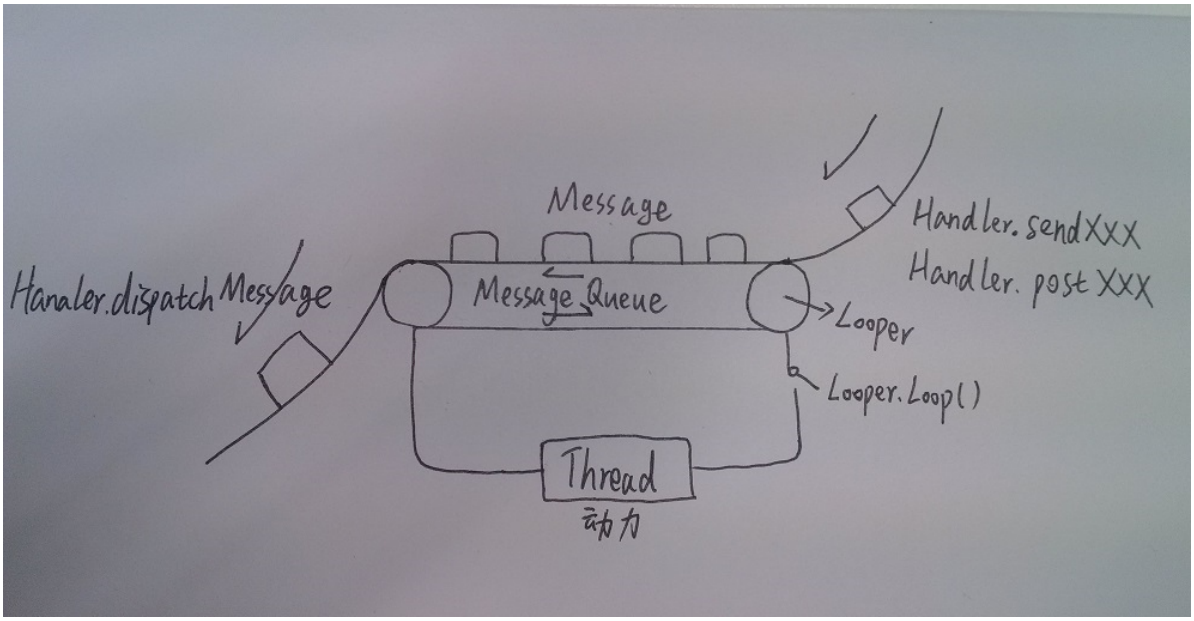
2.如果我们不是通过postXXX系列方法将Message放入到消息队列中的，那么msg.callback就是null，代码继续往下执行，接着我们会判断Handler的成员字段mCallback存不存在。mCallback是Hanlder.Callback类型的，我们在上面提到过，在Handler的构造函数中我们可以传递Hanlder.Callback类型的对象，该对象需要实现handleMessage方法，如果我们在构造函数中传递了该Callback对象，那么我们就会让Callback的handleMessage方法来处理Message。

3.如果我们在构造函数中没有传入Callback类型的对象，那么mCallback就为null,那么我们会调用Handler自身的hanldeMessage方法，该方法默认是个空方法，我们需要自己是重写实现该方法。

综上，我们可以看到Handler提供了三种途径处理Message，而且处理有前后优先级之分：首先尝试让postXXX中传递的Runnable执行，其次尝试让Handler构造函数中传入的Callback的handleMessage方法处理，最后才是让Handler自身的handleMessage方法处理Message。

一图胜千言

我们在本文讨论了Thread、MessageQueue、Looper以及Hanlder的之间的关系，我们可以通过如下一张传送带的图来更形象的理解他们之间的关系。



在现实生活的生产生活中，存在着各种各样的传送带，传送带上面洒满了各种货物，传送带在发动机滚轮的带动下一直在向前滚动，不断有新的货物放置在传送带的一端，货物在传送带的带动下送到另一端进行收集处理。

我们可以把传送带上的货物看做是一个个的Message，而承载这些货物的传送带就是装载Message的消息队列MessageQueue。传送带是靠发送机滚轮带动起来转动的，我们可以把发送机滚轮看做是Looper，而发动机的转动是需要电源的，我们可以把电源看做是线程Thread，所有的消息循环的一切操作都是基于某个线程的。一切准备就绪，我们只需要按下电源开关发动机就会转动起来，这个开关就是Looper的loop方法，当我们按下开关的时候，我们就相当于执行了Looper的loop方法，此时Looper就会驱动着消息队列循环起来。

那Hanlder在传送带模型中相当于什么呢？我们可以将Handler看做是放入货物以及取走货物的管道：货物从一端顺着管道划入传送带，货物又从另一端顺着管道划出传送带。我们在传送带的一端放入货物的操作就相当于我们调用了Handler的sendMessageXXX、sendEmptyMessageXXX或postXXX方法，这就把Message对象放入到了消息队列MessageQueue中了。当货物从传送带的另一端顺着管道划出时，我们就相当于调用了Hanlder的dispatchMessage方法，在该方法中我们完成对Message的处理。