<http://blog.csdn.net/lmj623565791/article/details/38377229/>

**1、 概述**

Handler 、 Looper 、Message 这三者都与Android异步消息处理线程相关的概念。那么什么叫异步消息处理线程呢？  
**异步消息处理线程启动后会进入一个无限的循环体之中，每循环一次，从其内部的消息队列中取出一个消息，然后回调相应的消息处理函数，执行完成一个消息后则继续循环。若消息队列为空，线程则会阻塞等待。**

说了这一堆，那么和Handler 、 Looper 、Message有啥关系？

其实**Looper负责的就是创建一个MessageQueue，然后进入一个无限循环体不断从该MessageQueue中读取消息，而消息的创建者就是一个或多个Handler 。**

**2、 源码解析**

**1、Looper**

对于Looper主要是prepare()和loop()两个方法。  
首先看prepare()方法

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **static** **final** **void** prepare() {
2. **if** (sThreadLocal.get() != **null**) {
3. **throw** **new** RuntimeException("Only one Looper may be created per thread");
4. }
5. sThreadLocal.set(**new** Looper(**true**));
6. }

sThreadLocal是一个ThreadLocal对象，可以在一个线程中存储变量。可以看到，在第5行，将一个Looper的实例放入了ThreadLocal，并且2-4行判断了sThreadLocal是否为null，否则抛出异常。这也就说明了**Looper.prepare()方法不能被调用两次，同时也保证了一个线程中只有一个Looper实例**~相信有些哥们一定遇到这个错误。  
下面看Looper的构造方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **private Looper(boolean quitAllowed) {**
2. mQueue = **new** MessageQueue(quitAllowed);
3. mRun = **true**;
4. mThread = Thread.currentThread();
5. }

在构造方法中，创建了一个MessageQueue（消息队列）。  
然后我们看loop()方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **static** **void** loop() {
2. **final** Looper me = myLooper();
3. **if** (me == **null**) {
4. **throw** **new** RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
5. }
6. **final** MessageQueue queue = me.mQueue;
8. // Make sure the identity of this thread is that of the local process,
9. // and keep track of what that identity token actually is.
10. Binder.clearCallingIdentity();
11. **final** **long** ident = Binder.clearCallingIdentity();
13. **for** (;;) {
14. Message msg = queue.next(); // might block
15. **if** (msg == **null**) {
16. // No message indicates that the message queue is quitting.
17. **return**;
18. }
20. // This must be in a local variable, in case a UI event sets the logger
21. Printer logging = me.mLogging;
22. **if** (logging != **null**) {
23. logging.println(">>>>> Dispatching to " + msg.target + " " +
24. msg.callback + ": " + msg.what);
25. }
27. msg.target.dispatchMessage(msg);
29. **if** (logging != **null**) {
30. logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
31. }
33. // Make sure that during the course of dispatching the
34. // identity of the thread wasn't corrupted.
35. **final** **long** newIdent = Binder.clearCallingIdentity();
36. **if** (ident != newIdent) {
37. Log.wtf(TAG, "Thread identity changed from 0x"
38. + Long.toHexString(ident) + " to 0x"
39. + Long.toHexString(newIdent) + " while dispatching to "
40. + msg.target.getClass().getName() + " "
41. + msg.callback + " what=" + msg.what);
42. }
44. msg.recycle();
45. }
46. }

第2行：  
public static Looper myLooper() {  
return sThreadLocal.get();  
}  
方法直接返回了sThreadLocal存储的Looper实例，如果me为null则抛出异常，也就是说**loop方法必须在prepare方法之后运行。**  
第6行：拿到该looper实例中的mQueue（消息队列）  
13到45行：就进入了我们所说的无限循环。  
14行：取出一条消息，如果没有消息则阻塞。  
27行：使用调用 msg.target.dispatchMessage(msg);把消息交给msg的target的dispatchMessage方法去处理。Msg的target是什么呢？其实就是handler对象，下面会进行分析。  
44行：释放消息占据的资源。  
  
Looper主要作用：  
1、 **与当前线程绑定，保证一个线程只会有一个Looper实例，同时一个Looper实例也只有一个MessageQueue。**  
2、 loop()方法，不断从MessageQueue中去取消息，交给消息的target属性的dispatchMessage去处理。  
好了，我们的异步消息处理线程已经有了消息队列（MessageQueue），也有了在无限循环体中取出消息的哥们，现在缺的就是发送消息的对象了，于是乎：Handler登场了。

**2、Handler**

使用Handler之前，我们都是初始化一个实例，比如用于更新UI线程，我们会在声明的时候直接初始化，或者在onCreate中初始化Handler实例。所以我们首先看Handler的构造方法，看其如何与MessageQueue联系上的，它在子线程中发送的消息（一般发送消息都在非UI线程）怎么发送到MessageQueue中的。

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** Handler() {
2. **this**(**null**, **false**);
3. }
4. **public** Handler(Callback callback, **boolean** async) {
5. **if** (FIND\_POTENTIAL\_LEAKS) {
6. **final** Class<? **extends** Handler> klass = getClass();
7. **if** ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&
8. (klass.getModifiers() & Modifier.STATIC) == 0) {
9. Log.w(TAG, "The following Handler class should be static or leaks might occur: " +
10. klass.getCanonicalName());
11. }
12. }
14. mLooper = Looper.myLooper();
15. **if** (mLooper == **null**) {
16. **throw** **new** RuntimeException(
17. "Can't create handler inside thread that has not called Looper.prepare()");
18. }
19. mQueue = mLooper.mQueue;
20. mCallback = callback;
21. mAsynchronous = async;
22. }

14行：通过Looper.myLooper()获取了当前线程保存的Looper实例，然后在19行又获取了这个Looper实例中保存的MessageQueue（消息队列），这样就保证了handler的实例与我们Looper实例中MessageQueue关联上了。

然后看我们最常用的sendMessage方法

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **final** **boolean** sendMessage(Message msg)
2. {
3. **return** sendMessageDelayed(msg, 0);
4. }

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **final** **boolean** sendEmptyMessageDelayed(**int** what, **long** delayMillis) {
2. Message msg = Message.obtain();
3. msg.what = what;
4. **return** sendMessageDelayed(msg, delayMillis);
5. }

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **final** **boolean** sendMessageDelayed(Message msg, **long** delayMillis)
2. {
3. **if** (delayMillis < 0) {
4. delayMillis = 0;
5. }
6. **return** sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
7. }

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **boolean** sendMessageAtTime(Message msg, **long** uptimeMillis) {
2. MessageQueue queue = mQueue;
3. **if** (queue == **null**) {
4. RuntimeException e = **new** RuntimeException(
5. **this** + " sendMessageAtTime() called with no mQueue");
6. Log.w("Looper", e.getMessage(), e);
7. **return** **false**;
8. }
9. **return** enqueueMessage(queue, msg, uptimeMillis);
10. }

辗转反则最后调用了sendMessageAtTime，在此方法内部有直接获取MessageQueue然后调用了enqueueMessage方法，我们再来看看此方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **private** **boolean** enqueueMessage(MessageQueue queue, Message msg, **long** uptimeMillis) {
2. msg.target = **this**;
3. **if** (mAsynchronous) {
4. msg.setAsynchronous(**true**);
5. }
6. **return** queue.enqueueMessage(msg, uptimeMillis);
7. }

enqueueMessage中首先为meg.target赋值为this，【如果大家还记得Looper的loop方法会取出每个msg然后交给msg,target.dispatchMessage(msg)去处理消息】，也就是把当前的handler作为msg的target属性。最终会调用queue的enqueueMessage的方法，也就是说handler发出的消息，最终会保存到消息队列中去。

现在已经很清楚了Looper会调用prepare()和loop()方法，在当前执行的线程中保存一个Looper实例，这个实例会保存一个MessageQueue对象，然后当前线程进入一个无限循环中去，不断从MessageQueue中读取Handler发来的消息。然后再回调创建这个消息的handler中的dispathMessage方法，下面我们赶快去看一看这个方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **void** dispatchMessage(Message msg) {
2. **if** (msg.callback != **null**) {
3. handleCallback(msg);
4. } **else** {
5. **if** (mCallback != **null**) {
6. **if** (mCallback.handleMessage(msg)) {
7. **return**;
8. }
9. }
10. handleMessage(msg);
11. }
12. }

可以看到，第10行，调用了handleMessage方法，下面我们去看这个方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. /\*\*
2. \* Subclasses must implement this to receive messages.
3. \*/
4. **public** **void** handleMessage(Message msg) {
5. }

可以看到这是一个空方法，为什么呢，因为消息的最终回调是由我们控制的，我们在创建handler的时候都是复写handleMessage方法，然后根据msg.what进行消息处理。

例如：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **private** Handler mHandler = **new** Handler()
2. {
3. **public** **void** handleMessage(android.os.Message msg)
4. {
5. **switch** (msg.what)
6. {
7. **case** value:
9. **break**;
11. **default**:
12. **break**;
13. }
14. };
15. };

到此，这个流程已经解释完毕，让我们首先总结一下

1、首先Looper.prepare()在本线程中保存一个Looper实例，然后该实例中保存一个MessageQueue对象；因为Looper.prepare()在一个线程中只能调用一次，所以MessageQueue在一个线程中只会存在一个。

2、Looper.loop()会让当前线程进入一个无限循环，不断从MessageQueue的实例中读取消息，然后回调msg.target.dispatchMessage(msg)方法。

3、Handler的构造方法，会首先得到当前线程中保存的Looper实例，进而与Looper实例中的MessageQueue想关联。

4、Handler的sendMessage方法，会给msg的target赋值为handler自身，然后加入MessageQueue中。

5、在构造Handler实例时，我们会重写handleMessage方法，也就是msg.target.dispatchMessage(msg)最终调用的方法。

好了，总结完成，大家可能还会问，那么在Activity中，我们并没有显示的调用Looper.prepare()和Looper.loop()方法，为啥Handler可以成功创建呢，这是因为在Activity的启动代码中，已经在当前UI线程调用了Looper.prepare()和Looper.loop()方法。

**3、Handler post**

今天有人问我，你说Handler的post方法创建的线程和UI线程有什么关系？

其实这个问题也是出现这篇博客的原因之一；这里需要说明，有时候为了方便，我们会直接写如下代码：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. mHandler.post(**new** Runnable()
2. {
3. @Override
4. **public** **void** run()
5. {
6. Log.e("TAG", Thread.currentThread().getName());
7. mTxt.setText("yoxi");
8. }
9. });

然后run方法中可以写更新UI的代码，其实这个Runnable并没有创建什么线程，而是发送了一条消息，下面看源码：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **final** **boolean** post(Runnable r)
2. {
3. **return**  sendMessageDelayed(getPostMessage(r), 0);
4. }

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **private** **static** Message getPostMessage(Runnable r) {
2. Message m = Message.obtain();
3. m.callback = r;
4. **return** m;
5. }

可以看到，在getPostMessage中，得到了一个Message对象，然后将我们创建的Runable对象作为callback属性，赋值给了此message.

注：产生一个Message对象，可以new  ，也可以使用Message.obtain()方法；两者都可以，但是更建议使用obtain方法，因为Message内部维护了一个Message池用于Message的复用，避免使用new 重新分配内存。

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **final** **boolean** sendMessageDelayed(Message msg, **long** delayMillis)
2. {
3. **if** (delayMillis < 0) {
4. delayMillis = 0;
5. }
6. **return** sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
7. }

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **boolean** sendMessageAtTime(Message msg, **long** uptimeMillis) {
2. MessageQueue queue = mQueue;
3. **if** (queue == **null**) {
4. RuntimeException e = **new** RuntimeException(
5. **this** + " sendMessageAtTime() called with no mQueue");
6. Log.w("Looper", e.getMessage(), e);
7. **return** **false**;
8. }
9. **return** enqueueMessage(queue, msg, uptimeMillis);
10. }

最终和handler.sendMessage一样，调用了sendMessageAtTime，然后调用了enqueueMessage方法，给msg.target赋值为handler，最终加入MessagQueue.

可以看到，这里msg的callback和target都有值，那么会执行哪个呢？

其实上面已经贴过代码，就是dispatchMessage方法：

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

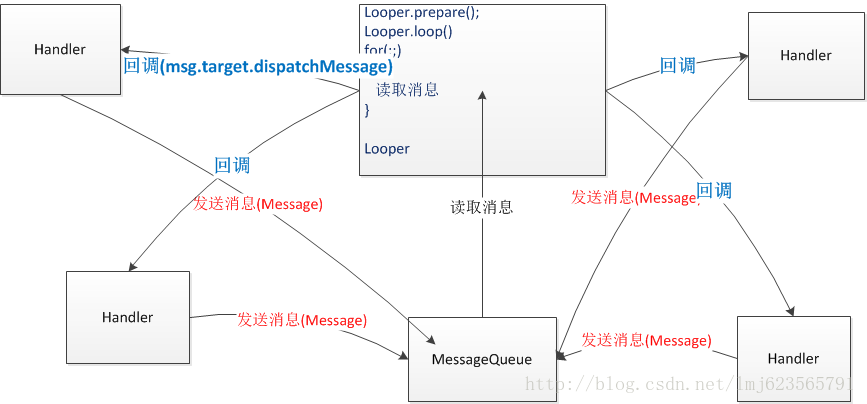
[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **public** **void** dispatchMessage(Message msg) {
2. **if** (msg.callback != **null**) {
3. handleCallback(msg);
4. } **else** {
5. **if** (mCallback != **null**) {
6. **if** (mCallback.handleMessage(msg)) {
7. **return**;
8. }
9. }
10. handleMessage(msg);
11. }
12. }

第2行，如果不为null，则执行callback回调，也就是我们的Runnable对象。

好了，关于Looper , Handler , Message 这三者关系上面已经叙述的非常清楚了。

最后来张图解：



希望图片可以更好的帮助大家的记忆~~

**4、后话**

其实Handler不仅可以更新UI，你完全可以在一个子线程中去创建一个Handler，然后使用这个handler实例在任何其他线程中发送消息，最终处理消息的代码都会在你创建Handler实例的线程中运行。

**[java]** [view plain](http://blog.csdn.net/lmj623565791/article/details/38377229/) [copy](http://blog.csdn.net/lmj623565791/article/details/38377229/)

[在CODE上查看代码片](https://code.csdn.net/snippets/445431)

1. **new** Thread()
2. {
3. **private** Handler handler;
4. **public** **void** run()
5. {
7. Looper.prepare();
9. handler = **new** Handler()
10. {
11. **public** **void** handleMessage(android.os.Message msg)
12. {
13. Log.e("TAG",Thread.currentThread().getName());
14. };
15. };<pre code\_snippet\_id="445431" snippet\_file\_name="blog\_20140808\_19\_1943618" name="code" **class**="java">                               Looper.loop();                                                                                                                              }             </pre>

Android不仅给我们提供了异步消息处理机制让我们更好的完成UI的更新，其实也为我们提供了异步消息处理机制代码的参考~~不仅能够知道原理，最好还可以将此设计用到其他的非Android项目中去~~

最新补充：

关于后记，有兄弟联系我说，到底可以在哪使用，见博客：[Android Handler 异步消息处理机制的妙用 创建强大的图片加载类](http://blog.csdn.net/lmj623565791/article/details/38476887)