# handler 消息机制

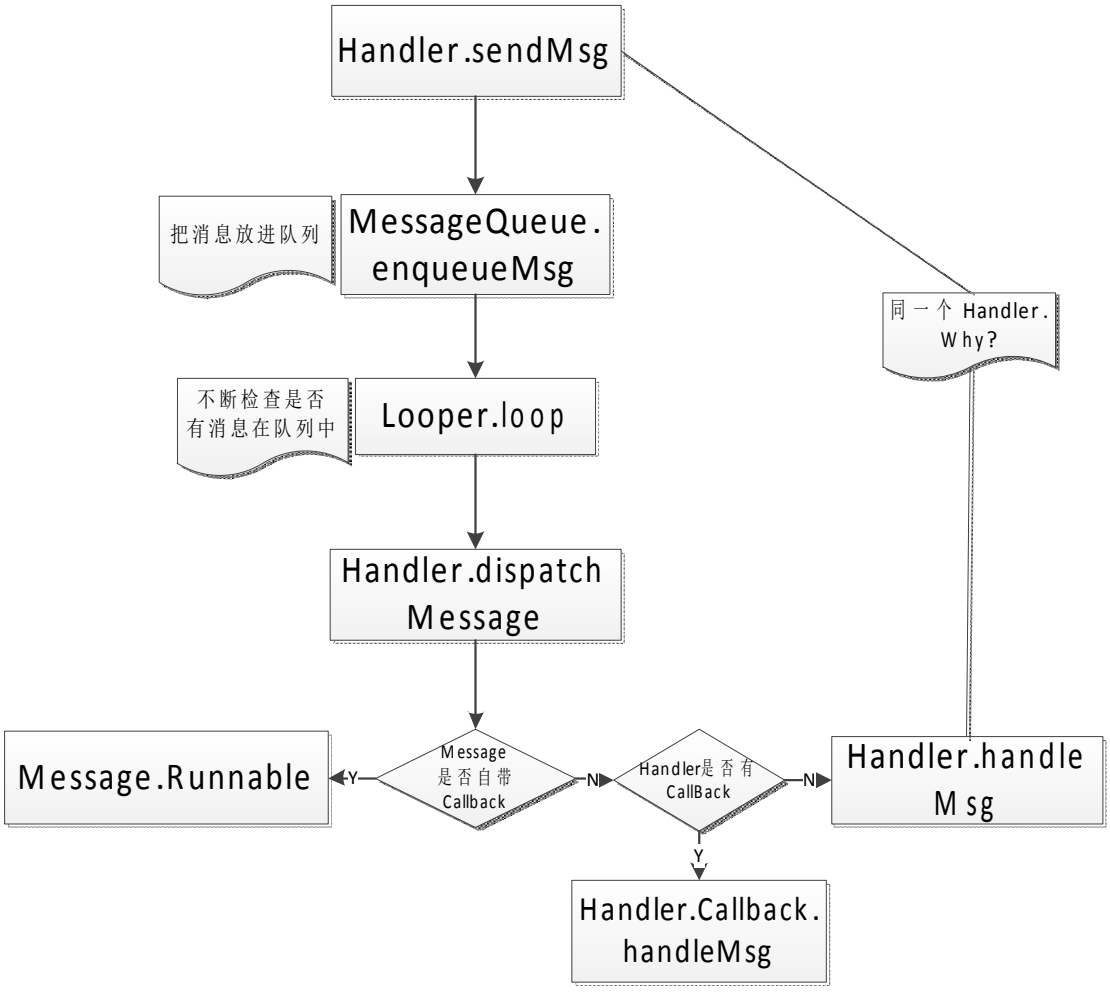
Jammy 2013 年 7 月 2 日

## 一、Handler 和 Message 简介

多用在大量操作线程和 UI 线程之间，为的是不让大量的操作，把 UI 卡死，出现 ANR。

Handler 发送的 Message 被放到一个队列中：Message Queue。Android 里并没有 Global 的 Message Queue 数据结构。例如，不同 APK 里的对象不能透过 Massage Queue 来交换讯息(Message)。

## 二、消息发送和处理流程

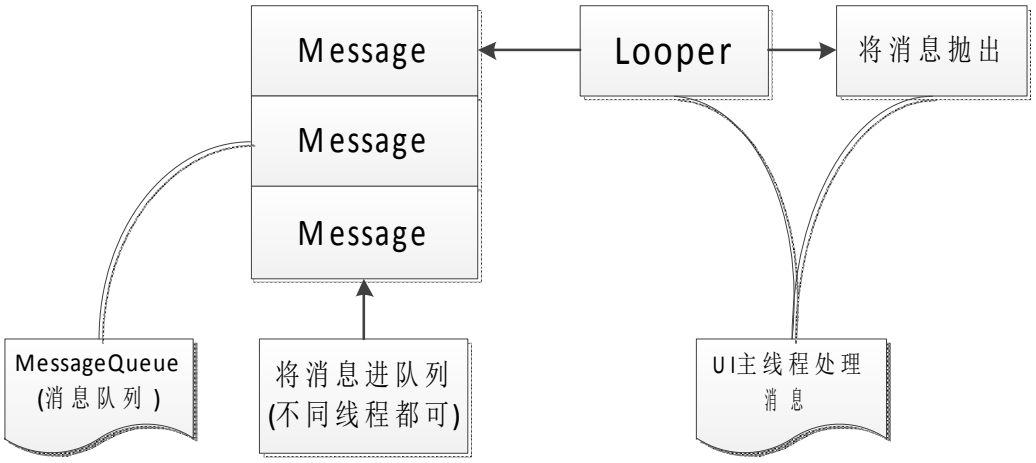



消息发送和处理流程图

### 1、为什么 Handler 的处理消息，就是 UI 线程在处理?

Looper 和 MessageQueue 在 ActivityThread 启动的时候，就已经被初始化了。Looper 在初始化之后，会**在主线程运行**，不断判断消息队列中是否有消息，如果有，那么把消息抛出去给对应的 Handler 处理。

将消息塞入消息队列的可以是各种线程，但是去拿这些消息，并将这些消息分发出去，都是 UI 主线程在处理，所以最后 Handler 处理消息都是在 UI 线程。




## 2、如何确定是同一个 Handler

Message 在被发送出去时，最后都会调用 Handler 的函数 sendMessageAtTime 来发送，在该函数中，Message 的属性 target 会被赋值。

r2 / android.os.Handler

```
451     public boolean sendMessageAtTime(Message msg, long uptimeMillis)
452     {
453         boolean sent = false;
454         MessageQueue queue = mQueue;
455         if (queue != null) {
456             msg.target = this;
457             sent = queue.enqueueMessage(msg, uptimeMillis);
458         }
459         else {
460             RuntimeException e = new RuntimeException(
461                     this + " sendMessageAtTime() called with no mQueue");
462             Log.w("Looper", e.getMessage(), e);
463         }
464         return sent;
465     }
```

target被赋值为发送msg的Handler

而 Looper 在获取到消息，在抛出消息的时候，会找对应的 Handler 来处理这个消息。

```
2 / android.os.Looper

106     public static final void loop() {
107         Looper me = myLooper();
108         MessageQueue queue = me.mQueue;
109         while (true) {
110             Message msg = queue.next(); // might block
111             //if (!me.mRun) {
112             //    break;
113             //}
114             if (msg != null) {
115                 if (msg.target == null) {
116                     // No target is a magic identifier for the quit message.
117                     return;
118                 }
119                 if (me.mLogging!= null) me.mLogging.println(
120                         ">>>>> Dispatching to " + msg.target + " "
121 msg.target就是Handler   + msg.callback + ": " + msg.what
122                         );
123                 msg.target.dispatchMessage(msg);
124                 if (me.mLogging!= null) me.mLogging.println(
125                         "<<<<< Finished to    " + msg.target + " "
126                         + msg.callback);
127                 msg.recycle();
128             }
129         }
130     }
```

从这两段代码中，可以得出，Handler 发送的消息，通常是由自己来处理。

### 3、三种种消息处理方式介绍

在上一点中提到通常是由 Handler 自身来处理消息的，实际上每个消息都有三种处理方式。

```
r2 / android.os.Handler

90      public void dispatchMessage(Message msg) {
91          if (msg.callback != null) {
92              handleCallback(msg);  <-----
93          } else {
94              if (mCallback != null) {
95                  if (mCallback.handleMessage(msg)) {  <-----
96                      return;
97                  }                          三种方式去处理Message
98              }
99              handleMessage(msg);  <-----    通常是使用最后这种
100         }
101     }
```

①、第一种是如果 Message 自身已经有 callback(是一个 Runnable 的属性)，那么会调用自己的 callback。dispatchMessage 是由 Looper 中调用的，所以是运行在 UI 线程中了，无其他线程处理 UI 事务之忧。

如果想要用这种方式来处理 Message，可以在实例化 Message 的时候，传入 Runnable

参数。如下：

r2 / android.os.Message

```
134     public static Message obtain(Handler h, Runnable callback) {
135         Message m = obtain();
136         m.target = h;
137         m.callback = callback;
138
139         return m;
140     }
```

②、第二种是 Handler 中的 mCallback 属性被实例化了，那么也会优先调用这个。mCallback 是一个定义在 Handler 中的一个接口。

r2 / android.os.Handler

Find Usages Diff

Callback interface you can use when instantiating a Handler to avoid having to implement your own subclass of Handler.

```
76
77      public interface Callback {
78          public boolean handleMessage(Message msg);
79      }
```

同样，使用方法，在初始化 Handler 的时候，传入 Callback 参数，即可。

③、第三种是常用的实例化 Handler 中，重写 handlerMessage 函数即可。这个在平时中多用到，此处略过。

## 三、Message 的缓存机制

在 Looper 将 msg 抛出去之后，有个 msg.recycle()的操作，这个就是将该 msg 回收，会执行 Message 的 recycle 函数。

r2 / android.os.Message

Find Usages Diff Raw

Return a Message instance to the global pool. You MUST NOT touch the Message after calling this function -- it has effectively been freed.

```
221
222     public void recycle() {
223         synchronized (mPoolSync) {
224             if (mPoolSize < MAX_POOL_SIZE) {
225                 clearForRecycle();
226
227                 next = mPool;
228                 mPool = this;
229             }
230         }
231     }
```

看代码没发现mPoolSize会增加，值一直为0

所谓的恢复出厂设置。将属性置null或0

把这Message加入到mPool链表中

每次在获取 Message 的时候，如果调用 Message 的 obtain 函数，就极大可能返回缓存中的 Message。


这里可以看出，以后在需要用 Message 的时候，尽量调用 Message 的 obtain 来获取，或者调用 Handler 的 obtainMessage（后者也是调用 Message 的 obtain），可以减少内存分配。