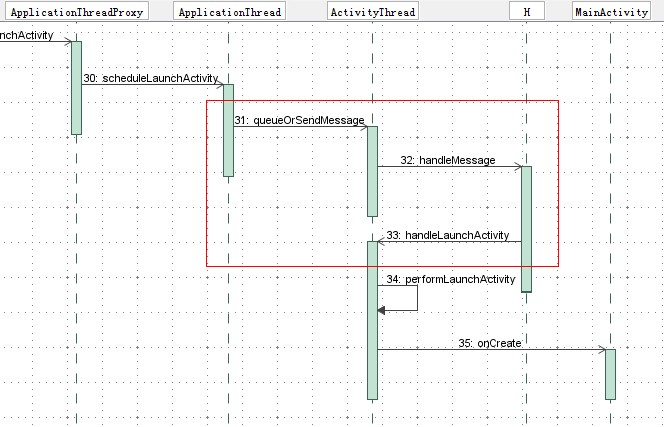
[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)

Android应用程序是通过消息来驱动的，系统为每一个应用程序维护一个消息队例，应用程序的主线程不断地从这个消息队例中获取消息（Looper），然后对这些消息进行处理（Handler），这样就实现了通过消息来驱动应用程序的执行，本文将详细分析Android应用程序的消息处理机制。

        前面我们学习Android应用程序中的Activity启动（[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)和[Android应用程序内部启动Activity过程（startActivity）的源代码分析](http://blog.csdn.net/luoshengyang/article/details/6703247)）、Service启动（[Android系统在新进程中启动自定义服务过程（startService）的原理分析](http://blog.csdn.net/luoshengyang/article/details/6677029)和[Android应用程序绑定服务（bindService）的过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6745181)）以及广播发送（[Android应用程序发送广播（sendBroadcast）的过程分析](http://blog.csdn.net/luoshengyang/article/details/6744448)）时，它们都有一个共同的特点，当ActivityManagerService需要与应用程序进行并互时，如加载Activity和Service、处理广播待，会通过[Binder进程间通信机制](http://blog.csdn.net/luoshengyang/article/details/6618363)来知会应用程序，应用程序接收到这个请求时，它不是马上就处理这个请求，而是将这个请求封装成一个消息，然后把这个消息放在应用程序的消息队列中去，然后再通过消息循环来处理这个消息。这样做的好处就是消息的发送方只要把消息发送到应用程序的消息队列中去就行了，它可以马上返回去处理别的事情，而不需要等待消息的接收方去处理完这个消息才返回，这样就可以提高系统的并发性。实质上，这就是一种异步处理机制。

        这样说可能还是比较笼统，我们以[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文中所介绍的应用程序启动过程的一个片断来具体看看是如何这种消息处理机制的。在这篇文章中，要启动的应用程序称为Activity，它的默认Activity是MainActivity，它是由[Launcher](http://blog.csdn.net/luoshengyang/article/details/6767736)来负责启动的，而Launcher又是通过ActivityManagerService来启动的，当ActivityManagerService为这个即将要启的应用程序准备好新的进程后，便通过一个[Binder进程间通信过程](http://blog.csdn.net/luoshengyang/article/details/6618363)来通知这个新的进程来加载MainActivity，如下图所示：



        它对应Android应用程序启动过程中的Step 30到Step 35，有兴趣的读者可以回过头去参考[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文。这里的Step 30中的scheduleLaunchActivity是ActivityManagerService通过[Binder进程间通信机制](http://blog.csdn.net/luoshengyang/article/details/6618363)发送过来的请求，它请求应用程序中的ActivityThread执行Step 34中的performLaunchActivity操作，即启动MainActivity的操作。这里我们就可以看到，Step 30的这个请求并没有等待Step 34这个操作完成就返回了，它只是把这个请求封装成一个消息，然后通过Step 31中的queueOrSendMessage操作把这个消息放到应用程序的消息队列中，然后就返回了。应用程序发现消息队列中有消息时，就会通过Step 32中的handleMessage操作来处理这个消息，即调用Step 33中的handleLaunchActivity来执行实际的加载MainAcitivy类的操作。

        了解Android应用程序的消息处理过程之后，我们就开始分样它的实现原理了。与Windows应用程序的消息处理过程一样，Android应用程序的消息处理机制也是由消息循环、消息发送和消息处理这三个部分组成的，接下来，我们就详细描述这三个过程。

        1. 消息循环

        在消息处理机制中，消息都是存放在一个消息队列中去，而应用程序的主线程就是围绕这个消息队列进入一个无限循环的，直到应用程序退出。如果队列中有消息，应用程序的主线程就会把它取出来，并分发给相应的Handler进行处理；如果队列中没有消息，应用程序的主线程就会进入空闲等待状态，等待下一个消息的到来。在Android应用程序中，这个消息循环过程是由Looper类来实现的，它定义在frameworks/base/core/java/android/os/Looper.java文件中，在分析这个类之前，我们先看一下Android应用程序主线程是如何进入到这个消息循环中去的。

        在[Android应用程序进程启动过程的源代码分析](http://blog.csdn.net/luoshengyang/article/details/6747696)一文中，我们分析了Android应用程序进程的启动过程，Android应用程序进程在启动的时候，会在进程中加载ActivityThread类，并且执行这个类的main函数，应用程序的消息循环过程就是在这个main函数里面实现的，我们来看看这个函数的实现，它定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
2. ......
4. **public** **static** **final** **void** main(String[] args) {
5. ......
7. Looper.prepareMainLooper();
9. ......
11. ActivityThread thread = **new** ActivityThread();
12. thread.attach(**false**);
14. ......
16. Looper.loop();
18. ......
20. thread.detach();
22. ......
23. }
24. }

        这个函数做了两件事情，一是在主线程中创建了一个ActivityThread实例，二是通过Looper类使主线程进入消息循环中，这里我们只关注后者。

        首先看Looper.prepareMainLooper函数的实现，这是一个静态成员函数，定义在frameworks/base/core/java/android/os/Looper.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Looper {
2. ......
4. **private** **static** **final** ThreadLocal sThreadLocal = **new** ThreadLocal();
6. **final** MessageQueue mQueue;
8. ......
10. /\*\* Initialize the current thread as a looper.
11. \* This gives you a chance to create handlers that then reference
12. \* this looper, before actually starting the loop. Be sure to call
13. \* {@link #loop()} after calling this method, and end it by calling
14. \* {@link #quit()}.
15. \*/
16. **public** **static** **final** **void** prepare() {
17. **if** (sThreadLocal.get() != **null**) {
18. **throw** **new** RuntimeException("Only one Looper may be created per thread");
19. }
20. sThreadLocal.set(**new** Looper());
21. }
23. /\*\* Initialize the current thread as a looper, marking it as an application's main
24. \*  looper. The main looper for your application is created by the Android environment,
25. \*  so you should never need to call this function yourself.
26. \* {@link #prepare()}
27. \*/
29. **public** **static** **final** **void** prepareMainLooper() {
30. prepare();
31. setMainLooper(myLooper());
32. **if** (Process.supportsProcesses()) {
33. myLooper().mQueue.mQuitAllowed = **false**;
34. }
35. }
37. **private** **synchronized** **static** **void** setMainLooper(Looper looper) {
38. mMainLooper = looper;
39. }
41. /\*\*
42. \* Return the Looper object associated with the current thread.  Returns
43. \* null if the calling thread is not associated with a Looper.
44. \*/
45. **public** **static** **final** Looper myLooper() {
46. **return** (Looper)sThreadLocal.get();
47. }
49. **private** Looper() {
50. mQueue = **new** MessageQueue();
51. mRun = **true**;
52. mThread = Thread.currentThread();
53. }
55. ......
56. }

        函数prepareMainLooper做的事情其实就是在线程中创建一个Looper对象，这个Looper对象是存放在sThreadLocal成员变量里面的，成员变量sThreadLocal的类型为ThreadLocal，表示这是一个线程局部变量，即保证每一个调用了prepareMainLooper函数的线程里面都有一个独立的Looper对象。在线程是创建Looper对象的工作是由prepare函数来完成的，而在创建Looper对象的时候，会同时创建一个消息队列MessageQueue，保存在Looper的成员变量mQueue中，后续消息就是存放在这个队列中去。消息队列在Android应用程序消息处理机制中最重要的组件，因此，我们看看它的创建过程，即它的构造函数的实现，实现frameworks/base/core/java/android/os/MessageQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** MessageQueue {
2. ......
4. **private** **int** mPtr; // used by native code
6. **private** **native** **void** nativeInit();
8. MessageQueue() {
9. nativeInit();
10. }
12. ......
13. }

        它的初始化工作都交给JNI方法nativeInit来实现了，这个JNI方法定义在frameworks/base/core/jni/android\_os\_MessageQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **static** **void** android\_os\_MessageQueue\_nativeInit(JNIEnv\* env, jobject obj) {
2. NativeMessageQueue\* nativeMessageQueue = **new** NativeMessageQueue();
3. **if** (! nativeMessageQueue) {
4. jniThrowRuntimeException(env, "Unable to allocate native queue");
5. **return**;
6. }
8. android\_os\_MessageQueue\_setNativeMessageQueue(env, obj, nativeMessageQueue);
9. }

        在JNI中，也相应地创建了一个消息队列NativeMessageQueue，NativeMessageQueue类也是定义在frameworks/base/core/jni/android\_os\_MessageQueue.cpp文件中，它的创建过程如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. NativeMessageQueue::NativeMessageQueue() {
2. mLooper = Looper::getForThread();
3. **if** (mLooper == NULL) {
4. mLooper = **new** Looper(**false**);
5. Looper::setForThread(mLooper);
6. }
7. }

        它主要就是在内部创建了一个Looper对象，注意，这个Looper对象是实现在JNI层的，它与上面Java层中的Looper是不一样的，不过它们是对应的，下面我们进一步分析消息循环的过程的时候，读者就会清楚地了解到它们之间的关系。

        这个Looper的创建过程也很重要，不过我们暂时放一放，先分析完android\_os\_MessageQueue\_nativeInit函数的执行，它创建了本地消息队列NativeMessageQueue对象之后，接着调用android\_os\_MessageQueue\_setNativeMessageQueue函数来把这个消息队列对象保存在前面我们在Java层中创建的MessageQueue对象的mPtr成员变量里面：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **static** **void** android\_os\_MessageQueue\_setNativeMessageQueue(JNIEnv\* env, jobject messageQueueObj,
2. NativeMessageQueue\* nativeMessageQueue) {
3. env->SetIntField(messageQueueObj, gMessageQueueClassInfo.mPtr,
4. **reinterpret\_cast**<jint>(nativeMessageQueue));
5. }

        这里传进来的参数messageQueueObj即为我们前面在Java层创建的消息队列对象，而gMessageQueueClassInfo.mPtr即表示在Java类MessageQueue中，其成员变量mPtr的偏移量，通过这个偏移量，就可以把这个本地消息队列对象natvieMessageQueue保存在Java层创建的消息队列对象的mPtr成员变量中，这是为了后续我们调用Java层的消息队列对象的其它成员函数进入到JNI层时，能够方便地找回它在JNI层所对应的消息队列对象。

        我们再回到NativeMessageQueue的构造函数中，看看JNI层的Looper对象的创建过程，即看看它的构造函数是如何实现的，这个Looper类实现在frameworks/base/libs/utils/Looper.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. Looper::Looper(**bool** allowNonCallbacks) :
2. mAllowNonCallbacks(allowNonCallbacks),
3. mResponseIndex(0) {
4. **int** wakeFds[2];
5. **int** result = pipe(wakeFds);
6. ......
8. mWakeReadPipeFd = wakeFds[0];
9. mWakeWritePipeFd = wakeFds[1];
11. ......
13. #ifdef LOOPER\_USES\_EPOLL
14. // Allocate the epoll instance and register the wake pipe.
15. mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT);
16. ......
18. **struct** epoll\_event eventItem;
19. memset(& eventItem, 0, **sizeof**(epoll\_event)); // zero out unused members of data field union
20. eventItem.events = EPOLLIN;
21. eventItem.data.fd = mWakeReadPipeFd;
22. result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mWakeReadPipeFd, & eventItem);
23. ......
24. #else
25. ......
26. #endif
28. ......
29. }

        这个构造函数做的事情非常重要，它跟我们后面要介绍的应用程序主线程在消息队列中没有消息时要进入等待状态以及当消息队列有消息时要把应用程序主线程唤醒的这两个知识点息息相关。它主要就是通过pipe系统调用来创建了一个管道了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **int** wakeFds[2];
2. **int** result = pipe(wakeFds);
3. ......
5. mWakeReadPipeFd = wakeFds[0];
6. mWakeWritePipeFd = wakeFds[1];

        管道是Linux系统中的一种进程间通信机制，具体可以参考前面一篇文章[Android学习启动篇](http://blog.csdn.net/luoshengyang/article/details/6557518)推荐的一本书《Linux内核源代码情景分析》中的第6章--传统的Uinx进程间通信。简单来说，管道就是一个文件，在管道的两端，分别是两个打开文件描述符，这两个打开文件描述符都是对应同一个文件，其中一个是用来读的，另一个是用来写的，一般的使用方式就是，一个线程通过读文件描述符中来读管道的内容，当管道没有内容时，这个线程就会进入等待状态，而另外一个线程通过写文件描述符来向管道中写入内容，写入内容的时候，如果另一端正有线程正在等待管道中的内容，那么这个线程就会被唤醒。这个等待和唤醒的操作是如何进行的呢，这就要借助Linux系统中的epoll机制了。 Linux系统中的epoll机制为处理大批量句柄而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著减少程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。但是这里我们其实只需要监控的IO接口只有mWakeReadPipeFd一个，即前面我们所创建的管道的读端，为什么还需要用到epoll呢？有点用牛刀来杀鸡的味道。其实不然，这个Looper类是非常强大的，它除了监控内部所创建的管道接口之外，还提供了addFd接口供外界面调用，外界可以通过这个接口把自己想要监控的IO事件一并加入到这个Looper对象中去，当所有这些被监控的IO接口上面有事件发生时，就会唤醒相应的线程来处理，不过这里我们只关心刚才所创建的管道的IO事件的发生。

        要使用Linux系统的epoll机制，首先要通过epoll\_create来创建一个epoll专用的文件描述符：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT);

       传入的参数EPOLL\_SIZE\_HINT是在这个mEpollFd上能监控的最大文件描述符数。

       接着还要通过epoll\_ctl函数来告诉epoll要监控相应的文件描述符的什么事件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **struct** epoll\_event eventItem;
2. memset(& eventItem, 0, **sizeof**(epoll\_event)); // zero out unused members of data field union
3. eventItem.events = EPOLLIN;
4. eventItem.data.fd = mWakeReadPipeFd;
5. result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mWakeReadPipeFd, & eventItem);

       这里就是告诉mEpollFd，它要监控mWakeReadPipeFd文件描述符的EPOLLIN事件，即当管道中有内容可读时，就唤醒当前正在等待管道中的内容的线程。  
       C++层的这个Looper对象创建好了之后，就返回到JNI层的NativeMessageQueue的构造函数，最后就返回到Java层的消息队列MessageQueue的创建过程，这样，Java层的Looper对象就准备好了。有点复杂，我们先小结一下这一步都做了些什么事情：

       A. 在Java层，创建了一个Looper对象，这个Looper对象是用来进入消息循环的，它的内部有一个消息队列MessageQueue对象mQueue；

       B. 在JNI层，创建了一个NativeMessageQueue对象，这个NativeMessageQueue对象保存在Java层的消息队列对象mQueue的成员变量mPtr中；

       C. 在C++层，创建了一个Looper对象，保存在JNI层的NativeMessageQueue对象的成员变量mLooper中，这个对象的作用是，当Java层的消息队列中没有消息时，就使Android应用程序主线程进入等待状态，而当Java层的消息队列中来了新的消息后，就唤醒Android应用程序的主线程来处理这个消息。

       回到ActivityThread类的main函数中，在上面这些工作都准备好之后，就调用Looper类的loop函数进入到消息循环中去了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Looper {
2. ......
4. **public** **static** final **void** loop() {
5. Looper me = myLooper();
6. MessageQueue queue = me.mQueue;
8. ......
10. **while** (**true**) {
11. Message msg = queue.next(); // might block
12. ......
14. **if** (msg != null) {
15. **if** (msg.target == null) {
16. // No target is a magic identifier for the quit message.
17. **return**;
18. }
20. ......
22. msg.target.dispatchMessage(msg);
24. ......
26. msg.recycle();
27. }
28. }
29. }
31. ......
32. }

        这里就是进入到消息循环中去了，它不断地从消息队列mQueue中去获取下一个要处理的消息msg，如果消息的target成员变量为null，就表示要退出消息循环了，否则的话就要调用这个target对象的dispatchMessage成员函数来处理这个消息，这个target对象的类型为Handler，下面我们分析消息的发送时会看到这个消息对象msg是如设置的。

        这个函数最关键的地方便是从消息队列中获取下一个要处理的消息了，即MessageQueue.next函数，它实现frameworks/base/core/java/android/os/MessageQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** MessageQueue {
2. ......
4. **final** Message next() {
5. **int** pendingIdleHandlerCount = -1; // -1 only during first iteration
6. **int** nextPollTimeoutMillis = 0;
8. **for** (;;) {
9. **if** (nextPollTimeoutMillis != 0) {
10. Binder.flushPendingCommands();
11. }
12. nativePollOnce(mPtr, nextPollTimeoutMillis);
14. **synchronized** (**this**) {
15. // Try to retrieve the next message.  Return if found.
16. **final** **long** now = SystemClock.uptimeMillis();
17. **final** Message msg = mMessages;
18. **if** (msg != **null**) {
19. **final** **long** when = msg.when;
20. **if** (now >= when) {
21. mBlocked = **false**;
22. mMessages = msg.next;
23. msg.next = **null**;
24. **if** (Config.LOGV) Log.v("MessageQueue", "Returning message: " + msg);
25. **return** msg;
26. } **else** {
27. nextPollTimeoutMillis = (**int**) Math.min(when - now, Integer.MAX\_VALUE);
28. }
29. } **else** {
30. nextPollTimeoutMillis = -1;
31. }
33. // If first time, then get the number of idlers to run.
34. **if** (pendingIdleHandlerCount < 0) {
35. pendingIdleHandlerCount = mIdleHandlers.size();
36. }
37. **if** (pendingIdleHandlerCount == 0) {
38. // No idle handlers to run.  Loop and wait some more.
39. mBlocked = **true**;
40. **continue**;
41. }
43. **if** (mPendingIdleHandlers == **null**) {
44. mPendingIdleHandlers = **new** IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
45. }
46. mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
47. }
49. // Run the idle handlers.
50. // We only ever reach this code block during the first iteration.
51. **for** (**int** i = 0; i < pendingIdleHandlerCount; i++) {
52. **final** IdleHandler idler = mPendingIdleHandlers[i];
53. mPendingIdleHandlers[i] = **null**; // release the reference to the handler
55. **boolean** keep = **false**;
56. **try** {
57. keep = idler.queueIdle();
58. } **catch** (Throwable t) {
59. Log.wtf("MessageQueue", "IdleHandler threw exception", t);
60. }
62. **if** (!keep) {
63. **synchronized** (**this**) {
64. mIdleHandlers.remove(idler);
65. }
66. }
67. }
69. // Reset the idle handler count to 0 so we do not run them again.
70. pendingIdleHandlerCount = 0;
72. // While calling an idle handler, a new message could have been delivered
73. // so go back and look again for a pending message without waiting.
74. nextPollTimeoutMillis = 0;
75. }
76. }
78. ......
79. }

        调用这个函数的时候，有可能会让线程进入等待状态。什么情况下，线程会进入等待状态呢？两种情况，一是当消息队列中没有消息时，它会使线程进入等待状态；二是消息队列中有消息，但是消息指定了执行的时间，而现在还没有到这个时间，线程也会进入等待状态。消息队列中的消息是按时间先后来排序的，后面我们在分析消息的发送时会看到。

        执行下面语句是看看当前消息队列中有没有消息：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. nativePollOnce(mPtr, nextPollTimeoutMillis);

        这是一个JNI方法，我们等一下再分析，这里传入的参数mPtr就是指向前面我们在JNI层创建的NativeMessageQueue对象了，而参数nextPollTimeoutMillis则表示如果当前消息队列中没有消息，它要等待的时候，for循环开始时，传入的值为0，表示不等待。

        当前nativePollOnce返回后，就去看看消息队列中有没有消息：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **final** Message msg = mMessages;
2. **if** (msg != **null**) {
3. **final** **long** when = msg.when;
4. **if** (now >= when) {
5. mBlocked = **false**;
6. mMessages = msg.next;
7. msg.next = **null**;
8. **if** (Config.LOGV) Log.v("MessageQueue", "Returning message: " + msg);
9. **return** msg;
10. } **else** {
11. nextPollTimeoutMillis = (**int**) Math.min(when - now, Integer.MAX\_VALUE);
12. }
13. } **else** {
14. nextPollTimeoutMillis = -1;
15. }

        如果消息队列中有消息，并且当前时候大于等于消息中的执行时间，那么就直接返回这个消息给Looper.loop消息处理，否则的话就要等待到消息的执行时间：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. nextPollTimeoutMillis = (**int**) Math.min(when - now, Integer.MAX\_VALUE);

        如果消息队列中没有消息，那就要进入无穷等待状态直到有新消息了：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. nextPollTimeoutMillis = -1;

        -1表示下次调用nativePollOnce时，如果消息中没有消息，就进入无限等待状态中去。

        这里计算出来的等待时间都是在下次调用nativePollOnce时使用的。

        这里说的等待，是空闲等待，而不是忙等待，因此，在进入空闲等待状态前，如果应用程序注册了IdleHandler接口来处理一些事情，那么就会先执行这里IdleHandler，然后再进入等待状态。IdlerHandler是定义在MessageQueue的一个内部类：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** MessageQueue {
2. ......
4. /\*\*
5. \* Callback interface for discovering when a thread is going to block
6. \* waiting for more messages.
7. \*/
8. **public** **static** **interface** IdleHandler {
9. /\*\*
10. \* Called when the message queue has run out of messages and will now
11. \* wait for more.  Return true to keep your idle handler active, false
12. \* to have it removed.  This may be called if there are still messages
13. \* pending in the queue, but they are all scheduled to be dispatched
14. \* after the current time.
15. \*/
16. **boolean** queueIdle();
17. }
19. ......
20. }

        它只有一个成员函数queueIdle，执行这个函数时，如果返回值为false，那么就会从应用程序中移除这个IdleHandler，否则的话就会在应用程序中继续维护着这个IdleHandler，下次空闲时仍会再执会这个IdleHandler。MessageQueue提供了addIdleHandler和removeIdleHandler两注册和删除IdleHandler。

        回到MessageQueue函数中，它接下来就是在进入等待状态前，看看有没有IdleHandler是需要执行的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. // If first time, then get the number of idlers to run.
2. **if** (pendingIdleHandlerCount < 0) {
3. pendingIdleHandlerCount = mIdleHandlers.size();
4. }
5. **if** (pendingIdleHandlerCount == 0) {
6. // No idle handlers to run.  Loop and wait some more.
7. mBlocked = **true**;
8. **continue**;
9. }
11. **if** (mPendingIdleHandlers == **null**) {
12. mPendingIdleHandlers = **new** IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
13. }
14. mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);

        如果没有，即pendingIdleHandlerCount等于0，那下面的逻辑就不执行了，通过continue语句直接进入下一次循环，否则就要把注册在mIdleHandlers中的IdleHandler取出来，放在mPendingIdleHandlers数组中去。

        接下来就是执行这些注册了的IdleHanlder了：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. // Run the idle handlers.
2. // We only ever reach this code block during the first iteration.
3. **for** (**int** i = 0; i < pendingIdleHandlerCount; i++) {
4. **final** IdleHandler idler = mPendingIdleHandlers[i];
5. mPendingIdleHandlers[i] = **null**; // release the reference to the handler
7. **boolean** keep = **false**;
8. **try** {
9. keep = idler.queueIdle();
10. } **catch** (Throwable t) {
11. Log.wtf("MessageQueue", "IdleHandler threw exception", t);
12. }
14. **if** (!keep) {
15. **synchronized** (**this**) {
16. mIdleHandlers.remove(idler);
17. }
18. }
19. }

         执行完这些IdleHandler之后，线程下次调用nativePollOnce函数时，就不设置超时时间了，因为，很有可能在执行IdleHandler的时候，已经有新的消息加入到消息队列中去了，因此，要重置nextPollTimeoutMillis的值：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. // While calling an idle handler, a new message could have been delivered
2. // so go back and look again for a pending message without waiting.
3. nextPollTimeoutMillis = 0;

        分析完MessageQueue的这个next函数之后，我们就要深入分析一下JNI方法nativePollOnce了，看看它是如何进入等待状态的，这个函数定义在frameworks/base/core/jni/android\_os\_MessageQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **static** **void** android\_os\_MessageQueue\_nativePollOnce(JNIEnv\* env, jobject obj,
2. jint ptr, jint timeoutMillis) {
3. NativeMessageQueue\* nativeMessageQueue = **reinterpret\_cast**<NativeMessageQueue\*>(ptr);
4. nativeMessageQueue->pollOnce(timeoutMillis);
5. }

        这个函数首先是通过传进入的参数ptr取回前面在Java层创建MessageQueue对象时在JNI层创建的NatvieMessageQueue对象，然后调用它的pollOnce函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **void** NativeMessageQueue::pollOnce(**int** timeoutMillis) {
2. mLooper->pollOnce(timeoutMillis);
3. }

        这里将操作转发给mLooper对象的pollOnce函数处理，这里的mLooper对象是在C++层的对象，它也是在前面在JNI层创建的NatvieMessageQueue对象时创建的，它的pollOnce函数定义在frameworks/base/libs/utils/Looper.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **int** Looper::pollOnce(**int** timeoutMillis, **int**\* outFd, **int**\* outEvents, **void**\*\* outData) {
2. **int** result = 0;
3. **for** (;;) {
4. ......
6. **if** (result != 0) {
7. ......
9. **return** result;
10. }
12. result = pollInner(timeoutMillis);
13. }
14. }

        为了方便讨论，我们把这个函数的无关部分都去掉，它主要就是调用pollInner函数来进一步操作，如果pollInner返回值不等于0，这个函数就可以返回了。

        函数pollInner的定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **int** Looper::pollInner(**int** timeoutMillis) {
2. ......
4. **int** result = ALOOPER\_POLL\_WAKE;
6. ......
8. #ifdef LOOPER\_USES\_EPOLL
9. **struct** epoll\_event eventItems[EPOLL\_MAX\_EVENTS];
10. **int** eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);
11. **bool** acquiredLock = **false**;
12. #else
13. ......
14. #endif
16. **if** (eventCount < 0) {
17. **if** (errno == EINTR) {
18. **goto** Done;
19. }
21. LOGW("Poll failed with an unexpected error, errno=%d", errno);
22. result = ALOOPER\_POLL\_ERROR;
23. **goto** Done;
24. }
26. **if** (eventCount == 0) {
27. ......
28. result = ALOOPER\_POLL\_TIMEOUT;
29. **goto** Done;
30. }
32. ......
34. #ifdef LOOPER\_USES\_EPOLL
35. **for** (**int** i = 0; i < eventCount; i++) {
36. **int** fd = eventItems[i].data.fd;
37. uint32\_t epollEvents = eventItems[i].events;
38. **if** (fd == mWakeReadPipeFd) {
39. **if** (epollEvents & EPOLLIN) {
40. awoken();
41. } **else** {
42. LOGW("Ignoring unexpected epoll events 0x%x on wake read pipe.", epollEvents);
43. }
44. } **else** {
45. ......
46. }
47. }
48. **if** (acquiredLock) {
49. mLock.unlock();
50. }
51. Done: ;
52. #else
53. ......
54. #endif
56. ......
58. **return** result;
59. }

        这里，首先是调用epoll\_wait函数来看看epoll专用文件描述符mEpollFd所监控的文件描述符是否有IO事件发生，它设置监控的超时时间为timeoutMillis：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **int** eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

        回忆一下前面的Looper的构造函数，我们在里面设置了要监控mWakeReadPipeFd文件描述符的EPOLLIN事件。

        当mEpollFd所监控的文件描述符发生了要监控的IO事件后或者监控时间超时后，线程就从epoll\_wait返回了，否则线程就会在epoll\_wait函数中进入睡眠状态了。返回后如果eventCount等于0，就说明是超时了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **if** (eventCount == 0) {
2. ......
3. result = ALOOPER\_POLL\_TIMEOUT;
4. **goto** Done;
5. }

       如果eventCount不等于0，就说明发生要监控的事件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **for** (**int** i = 0; i < eventCount; i++) {
2. **int** fd = eventItems[i].data.fd;
3. uint32\_t epollEvents = eventItems[i].events;
4. **if** (fd == mWakeReadPipeFd) {
5. **if** (epollEvents & EPOLLIN) {
6. awoken();
7. } **else** {
8. LOGW("Ignoring unexpected epoll events 0x%x on wake read pipe.", epollEvents);
9. }
10. } **else** {
11. ......
12. }
13. }

        这里我们只关注mWakeReadPipeFd文件描述符上的事件，如果在mWakeReadPipeFd文件描述符上发生了EPOLLIN就说明应用程序中的消息队列里面有新的消息需要处理了，接下来它就会先调用awoken函数清空管道中把内容，以便下次再调用pollInner函数时，知道自从上次处理完消息队列中的消息后，有没有新的消息加进来。

        函数awoken的实现很简单，它只是把管道中的内容都读取出来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **void** Looper::awoken() {
2. ......
4. **char** buffer[16];
5. ssize\_t nRead;
6. **do** {
7. nRead = read(mWakeReadPipeFd, buffer, **sizeof**(buffer));
8. } **while** ((nRead == -1 && errno == EINTR) || nRead == **sizeof**(buffer));
9. }

        因为当其它的线程向应用程序的消息队列加入新的消息时，会向这个管道写入新的内容来通知应用程序主线程有新的消息需要处理了，下面我们分析消息的发送的时候将会看到。

        这样，消息的循环过程就分析完了，这部分逻辑还是比较复杂的，它利用Linux系统中的管道（pipe）进程间通信机制来实现消息的等待和处理，不过，了解了这部分内容之后，下面我们分析消息的发送和处理就简单多了。

        2. 消息的发送  
        应用程序的主线程准备就好消息队列并且进入到消息循环后，其它地方就可以往这个消息队列中发送消息了。我们继续以文章开始介绍的[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文中的应用程序启动过为例，说明应用程序是如何把消息加入到应用程序的消息队列中去的。

        在[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)这篇文章的Step 30中，ActivityManagerService通过调用ApplicationThread类的scheduleLaunchActivity函数通知应用程序，它可以加载应用程序的默认Activity了，这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // we use token to identify this activity without having to send the
10. // activity itself back to the activity manager. (matters more with ipc)
11. **public** **final** **void** scheduleLaunchActivity(Intent intent, IBinder token, **int** ident,
12. ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
13. List<Intent> pendingNewIntents, **boolean** notResumed, **boolean** isForward) {
14. ActivityClientRecord r = **new** ActivityClientRecord();
16. r.token = token;
17. r.ident = ident;
18. r.intent = intent;
19. r.activityInfo = info;
20. r.state = state;
22. r.pendingResults = pendingResults;
23. r.pendingIntents = pendingNewIntents;
25. r.startsNotResumed = notResumed;
26. r.isForward = isForward;
28. queueOrSendMessage(H.LAUNCH\_ACTIVITY, r);
29. }
31. ......
33. }
35. ......
36. }

        这里把相关的参数都封装成一个ActivityClientRecord对象r，然后调用queueOrSendMessage函数来往应用程序的消息队列中加入一个新的消息（H.LAUNCH\_ACTIVITY），这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
7. ......
9. // if the thread hasn't started yet, we don't have the handler, so just
10. // save the messages until we're ready.
11. **private** **final** **void** queueOrSendMessage(**int** what, Object obj) {
12. queueOrSendMessage(what, obj, 0, 0);
13. }
15. ......
17. **private** **final** **void** queueOrSendMessage(**int** what, Object obj, **int** arg1, **int** arg2) {
18. **synchronized** (**this**) {
19. ......
20. Message msg = Message.obtain();
21. msg.what = what;
22. msg.obj = obj;
23. msg.arg1 = arg1;
24. msg.arg2 = arg2;
25. mH.sendMessage(msg);
26. }
27. }
29. ......
31. }
33. ......
34. }

        在queueOrSendMessage函数中，又进一步把上面传进来的参数封装成一个Message对象msg，然后通过mH.sendMessage函数把这个消息对象msg加入到应用程序的消息队列中去。这里的mH是ActivityThread类的成员变量，它的类型为H，继承于Handler类，它定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. ......
13. }
15. ......
17. }
19. ......
20. }

        这个H类就是通过其成员函数handleMessage函数来处理消息的了，后面我们分析消息的处理过程时会看到。  
        ActivityThread类的这个mH成员变量是什么时候创建的呢？我们前面在分析应用程序的消息循环时，说到当应用程序进程启动之后，就会加载ActivityThread类的main函数里面，在这个main函数里面，在通过Looper类进入消息循环之前，会在当前进程中创建一个ActivityThread实例：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
2. ......
4. **public** **static** **final** **void** main(String[] args) {
5. ......
7. ActivityThread thread = **new** ActivityThread();
8. thread.attach(**false**);
10. ......
11. }
12. }

        在创建这个实例的时候，就会同时创建其成员变量mH了：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
2. ......
4. **final** H mH = **new** H();
6. ......
7. }

        前面说过，H类继承于Handler类，因此，当创建这个H对象时，会调用Handler类的构造函数，这个函数定义在frameworks/base/core/java/android/os/Handler.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Handler {
2. ......
4. **public** Handler() {
5. ......
7. mLooper = Looper.myLooper();
8. ......
10. mQueue = mLooper.mQueue;
11. ......
12. }

15. **final** MessageQueue mQueue;
16. **final** Looper mLooper;
17. ......
18. }

        在Hanlder类的构造函数中，主要就是初始成员变量mLooper和mQueue了。这里的myLooper是Looper类的静态成员函数，通过它来获得一个Looper对象，这个Looper对象就是前面我们在分析消息循环时，在ActivityThread类的main函数中通过Looper.prepareMainLooper函数创建的。Looper.myLooper函数实现在frameworks/base/core/java/android/os/Looper.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Looper {
2. ......
4. **public** **static** **final** Looper myLooper() {
5. **return** (Looper)sThreadLocal.get();
6. }
8. ......
9. }

        有了这个Looper对象后，就可以通过Looper.mQueue来访问应用程序的消息队列了。

        有了这个Handler对象mH后，就可以通过它来往应用程序的消息队列中加入新的消息了。回到前面的queueOrSendMessage函数中，当它准备好了一个Message对象msg后，就开始调用mH.sendMessage函数来发送消息了，这个函数定义在frameworks/base/core/java/android/os/Handler.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Handler {
2. ......
4. **public** **final** **boolean** sendMessage(Message msg)
5. {
6. **return** sendMessageDelayed(msg, 0);
7. }
9. **public** **final** **boolean** sendMessageDelayed(Message msg, **long** delayMillis)
10. {
11. **if** (delayMillis < 0) {
12. delayMillis = 0;
13. }
14. **return** sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
15. }
17. **public** **boolean** sendMessageAtTime(Message msg, **long** uptimeMillis)
18. {
19. **boolean** sent = **false**;
20. MessageQueue queue = mQueue;
21. **if** (queue != **null**) {
22. msg.target = **this**;
23. sent = queue.enqueueMessage(msg, uptimeMillis);
24. }
25. **else** {
26. ......
27. }
28. **return** sent;
29. }
31. ......
32. }

        在发送消息时，是可以指定消息的处理时间的，但是通过sendMessage函数发送的消息的处理时间默认就为当前时间，即表示要马上处理，因此，从sendMessage函数中调用sendMessageDelayed函数，传入的时间参数为0，表示这个消息不要延时处理，而在sendMessageDelayed函数中，则会先获得当前时间，然后加上消息要延时处理的时间，即得到这个处理这个消息的绝对时间，然后调用sendMessageAtTime函数来把消息加入到应用程序的消息队列中去。

        在sendMessageAtTime函数，首先得到应用程序的消息队列mQueue，这是在Handler对象构造时初始化好的，前面已经分析过了，接着设置这个消息的目标对象target，即这个消息最终是由谁来处理的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. msg.target = **this**;

        这里将它赋值为this，即表示这个消息最终由这个Handler对象来处理，即由ActivityThread对象的mH成员变量来处理。

        函数最后调用queue.enqueueMessage来把这个消息加入到应用程序的消息队列中去，这个函数实现在frameworks/base/core/java/android/os/MessageQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** MessageQueue {
2. ......
4. **final** **boolean** enqueueMessage(Message msg, **long** when) {
5. ......
7. **final** **boolean** needWake;
8. **synchronized** (**this**) {
9. ......
11. msg.when = when;
12. //Log.d("MessageQueue", "Enqueing: " + msg);
13. Message p = mMessages;
14. **if** (p == **null** || when == 0 || when < p.when) {
15. msg.next = p;
16. mMessages = msg;
17. needWake = mBlocked; // new head, might need to wake up
18. } **else** {
19. Message prev = **null**;
20. **while** (p != **null** && p.when <= when) {
21. prev = p;
22. p = p.next;
23. }
24. msg.next = prev.next;
25. prev.next = msg;
26. needWake = **false**; // still waiting on head, no need to wake up
27. }
29. }
30. **if** (needWake) {
31. nativeWake(mPtr);
32. }
33. **return** **true**;
34. }
36. ......
37. }

        把消息加入到消息队列时，分两种情况，一种当前消息队列为空时，这时候应用程序的主线程一般就是处于空闲等待状态了，这时候就要唤醒它，另一种情况是应用程序的消息队列不为空，这时候就不需要唤醒应用程序的主线程了，因为这时候它一定是在忙着处于消息队列中的消息，因此不会处于空闲等待的状态。

        第一种情况比较简单，只要把消息放在消息队列头就可以了：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. msg.next = p;
2. mMessages = msg;
3. needWake = mBlocked; // new head, might need to wake up

        第二种情况相对就比较复杂一些了，前面我们说过，当往消息队列中发送消息时，是可以指定消息的处理时间的，而消息队列中的消息，就是按照这个时间从小到大来排序的，因此，当把新的消息加入到消息队列时，就要根据它的处理时间来找到合适的位置，然后再放进消息队列中去：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. Message prev = **null**;
2. **while** (p != **null** && p.when <= when) {
3. prev = p;
4. p = p.next;
5. }
6. msg.next = prev.next;
7. prev.next = msg;
8. needWake = **false**; // still waiting on head, no need to wake up

        把消息加入到消息队列去后，如果应用程序的主线程正处于空闲等待状态，就需要调用natvieWake函数来唤醒它了，这是一个JNI方法，定义在frameworks/base/core/jni/android\_os\_MessageQueue.cpp文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **static** **void** android\_os\_MessageQueue\_nativeWake(JNIEnv\* env, jobject obj, jint ptr) {
2. NativeMessageQueue\* nativeMessageQueue = reinterpret\_cast<NativeMessageQueue\*>(ptr);
3. **return** nativeMessageQueue->wake();
4. }

        这个JNI层的NativeMessageQueue对象我们在前面分析消息循环的时候创建好的，保存在Java层的MessageQueue对象的mPtr成员变量中，这里把它取回来之后，就调用它的wake函数来唤醒应用程序的主线程，这个函数也是定义在frameworks/base/core/jni/android\_os\_MessageQueue.cpp文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **void** NativeMessageQueue::wake() {
2. mLooper->wake();
3. }

        这里它又通过成员变量mLooper的wake函数来执行操作，这里的mLooper成员变量是一个C++层实现的Looper对象，它定义在frameworks/base/libs/utils/Looper.cpp文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **void** Looper::wake() {
2. ......
4. ssize\_t nWrite;
5. **do** {
6. nWrite = write(mWakeWritePipeFd, "W", 1);
7. } **while** (nWrite == -1 && errno == EINTR);
9. .......
10. }

        这个wake函数很简单，只是通过打开文件描述符mWakeWritePipeFd往管道的写入一个"W"字符串。其实，往管道写入什么内容并不重要，往管道写入内容的目的是为了唤醒应用程序的主线程。前面我们在分析应用程序的消息循环时说到，当应用程序的消息队列中没有消息处理时，应用程序的主线程就会进入空闲等待状态，而这个空闲等待状态就是通过调用这个Looper类的pollInner函数来进入的，具体就是在pollInner函数中调用epoll\_wait函数来等待管道中有内容可读的。

        这时候既然管道中有内容可读了，应用程序的主线程就会从这里的Looper类的pollInner函数返回到JNI层的nativePollOnce函数，最后返回到Java层中的MessageQueue.next函数中去，这里它就会发现消息队列中有新的消息需要处理了，于就会处理这个消息。

        3. 消息的处理

        前面在分析消息循环时，说到应用程序的主线程是在Looper类的loop成员函数中进行消息循环过程的，这个函数定义在frameworks/base/core/java/android/os/Looper.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Looper {
2. ......
4. **public** **static** **final** **void** loop() {
5. Looper me = myLooper();
6. MessageQueue queue = me.mQueue;
8. ......
10. **while** (**true**) {
11. Message msg = queue.next(); // might block
12. ......
14. **if** (msg != **null**) {
15. **if** (msg.target == **null**) {
16. // No target is a magic identifier for the quit message.
17. **return**;
18. }
20. ......
22. msg.target.dispatchMessage(msg);
24. ......
26. msg.recycle();
27. }
28. }
29. }
31. ......
32. }

        它从消息队列中获得消息对象msg后，就会调用它的target成员变量的dispatchMessage函数来处理这个消息。在前面分析消息的发送时说过，这个消息对象msg的成员变量target是在发送消息的时候设置好的，一般就通过哪个Handler来发送消息，就通过哪个Handler来处理消息。

        我们继续以前面分析消息的发送时所举的例子来分析消息的处理过程。前面说到，在[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)这篇文章的Step 30中，ActivityManagerService通过调用ApplicationThread类的scheduleLaunchActivity函数通知应用程序，它可以加载应用程序的默认Activity了，而ApplicationThread类的scheduleLaunchActivity函数最终把这个请求封装成一个消息，然后通过ActivityThread类的成员变量mH来把这个消息加入到应用程序的消息队列中去。现在要对这个消息进行处理了，于是就会调用H类的dispatchMessage函数进行处理。

        H类没有实现自己的dispatchMessage函数，但是它继承了父类Handler的dispatchMessage函数，这个函数定义在frameworks/base/core/java/android/os/ Handler.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **class** Handler {
2. ......
4. **public** **void** dispatchMessage(Message msg) {
5. **if** (msg.callback != **null**) {
6. handleCallback(msg);
7. } **else** {
8. **if** (mCallback != **null**) {
9. **if** (mCallback.handleMessage(msg)) {
10. **return**;
11. }
12. }
13. handleMessage(msg);
14. }
15. }
17. ......
18. }

        这里的消息对象msg的callback成员变量和Handler类的mCallBack成员变量一般都为null，于是，就会调用Handler类的handleMessage函数来处理这个消息，由于H类在继承Handler类时，重写了handleMessage函数，因此，这里调用的实际上是H类的handleMessage函数，这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6817933)

1. **public** **final** **class** ActivityThread {
3. ......
5. **private** **final** **class** H **extends** Handler {
7. ......
9. **public** **void** handleMessage(Message msg) {
10. ......
11. **switch** (msg.what) {
12. **case** LAUNCH\_ACTIVITY: {
13. ActivityClientRecord r = (ActivityClientRecord)msg.obj;
15. r.packageInfo = getPackageInfoNoCheck(
16. r.activityInfo.applicationInfo);
17. handleLaunchActivity(r, **null**);
18. } **break**;
19. ......
20. }
22. ......
24. }
26. ......
27. }

         因为前面在分析消息的发送时所举的例子中，发送的消息的类型为H.LAUNCH\_ACTIVITY，因此，这里就会调用ActivityThread类的handleLaunchActivity函数来真正地处理这个消息了，后面的具体过程就可以参考[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)这篇文章了。

         至此，我们就从消息循环、消息发送和消息处理三个部分分析完Android应用程序的消息处理机制了，为了更深理解，这里我们对其中的一些要点作一个总结：

         A. Android应用程序的消息处理机制由消息循环、消息发送和消息处理三个部分组成的。

         B. Android应用程序的主线程在进入消息循环过程前，会在内部创建一个Linux管道（Pipe），这个管道的作用是使得Android应用程序主线程在消息队列为空时可以进入空闲等待状态，并且使得当应用程序的消息队列有消息需要处理时唤醒应用程序的主线程。

         C. Android应用程序的主线程进入空闲等待状态的方式实际上就是在管道的读端等待管道中有新的内容可读，具体来说就是是通过Linux系统的Epoll机制中的epoll\_wait函数进行的。

         D. 当往Android应用程序的消息队列中加入新的消息时，会同时往管道中的写端写入内容，通过这种方式就可以唤醒正在等待消息到来的应用程序主线程。

         E. 当应用程序主线程在进入空闲等待前，会认为当前线程处理空闲状态，于是就会调用那些已经注册了的IdleHandler接口，使得应用程序有机会在空闲的时候处理一些事情。