Android消息处理机制(Handler、Looper、MessageQueue与Message)

Android是消息驱动的，实现消息驱动有几个要素：

1. 消息的表示：Message
2. 消息队列：MessageQueue
3. 消息循环，用于循环取出消息进行处理：Looper
4. 消息处理，消息循环从消息队列中取出消息后要对消息进行处理：Handler

平时我们最常使用的就是Message与Handler了，如果使用过HandlerThread或者自己实现类似HandlerThread的东西可能还会接触到Looper，而MessageQueue是Looper内部使用的，对于标准的SDK，我们是无法实例化并使用的（构造函数是包可见性）。

我们平时接触到的Looper、Message、Handler都是用JAVA实现的，Android做为基于Linux的系统，底层用C、C++实现的，而且还有NDK的存在，消息驱动的模型怎么可能只存在于JAVA层，实际上，在Native层存在与Java层对应的类如Looper、MessageQueue等。

初始化消息队列

首先来看一下如果一个线程想实现消息循环应该怎么做，以HandlerThread为例：

```
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

主要是红色标明的两句，首先调用prepare初始化MessageQueue与Looper，然后调用loop进入消息循环。先看一下Looper.prepare。

```
public static void prepare() {
    prepare(true);
}
```

```
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));
}
```

重载函数，quitAllowed默认为true，从名字可以看出来就是消息循环是否可以退出，默认是可退出的，Main线程（UI线程）初始化消息循环时会调用prepareMainLooper，传进去的是false。使用了ThreadLocal，每个线程可以初始化一个Looper。

再来看一下Looper在初始化时都做了什么：

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

MessageQueue(boolean quitAllowed) {
    mQuitAllowed = quitAllowed;
    nativeInit();
}
```

在Looper初始化时，新建了一个MessageQueue的对象保存了在成员mQueue中。MessageQueue的构造函数是包可见性，所以我们是无法直接使用的，在MessageQueue初始化的时候调用了nativeInit，这是一个Native方法：

```
按 Ctrl+C 复制代码
```

按 Ctrl+C 复制代码

在nativeInit中，new了一个Native层的MessageQueue的对象，并将其地址保存在了Java层MessageQueue的成员mPtr中，Android中有好多这样的实现，一个类在Java层与Native层都有实现，通过JNI的GetFieldID与SetIntField把Native层的类的实例地址保存到Java层类的实例的mPtr成员中，比如Parcel。

再看NativeMessageQueue的实现：

```
NativeMessageQueue::NativeMessageQueue() : mInCallback(false), mExceptionObj(NULL) {
    mLooper = Looper::getForThread();
    if (mLooper == NULL) {
        mLooper = new Looper(false);
        Looper::setForThread(mLooper);
    }
}
```

在NativeMessageQueue的构造函数中获得了一个Native层的Looper对象，Native层的Looper也使用了线程本地存储，注意new Looper时传入了参数false。

```
Looper::Looper(bool allowNonCallbacks) :
        mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),
        mResponseIndex(0), mNextMessageUptime(LLONG_MAX) {
    int wakeFds[2];
    int result = pipe(wakeFds);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not create wake pipe.  errno=%d", errno);
```

```
    mWakeReadPipeFd = wakeFds[0];
    mWakeWritePipeFd = wakeFds[1];

    result = fcntl(mWakeReadPipeFd, F_SETFL, O_NONBLOCK);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not make wake read pipe non-blocking.
errno=%d",
            errno);

    result = fcntl(mWakeWritePipeFd, F_SETFL, O_NONBLOCK);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not make wake write pipe non-blocking.
errno=%d",
            errno);

    // Allocate the epoll instance and register the wake pipe.
    mEpollFd = epoll_create(EPOLL_SIZE_HINT);
    LOG_ALWAYS_FATAL_IF(mEpollFd < 0, "Could not create epoll instance.  errno=%d",
errno);

    struct epoll_event eventItem;
    memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data
field union
    eventItem.events = EPOLLIN;
    eventItem.data.fd = mWakeReadPipeFd;
    result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not add wake read pipe to epoll instance.
errno=%d",
            errno);
}
```

Native层的Looper使用了epoll。初始化了一个管道，用mWakeWritePipeFd与mWakeReadPipeFd分别保存了管道的写端与读端，并监听了读端的EPOLLIN事件。注意下初始化列表的值，mAllowNonCallbacks的值为false。

mAllowNonCallback是做什么的？使用epoll仅为了监听mWakeReadPipeFd的事件？其实Native Looper不仅可以监听这一个描述符，Looper还提供了addFd方法：

```
int addFd(int fd, int ident, int events, ALooper_callbackFunc callback, void* data);
int addFd(int fd, int ident, int events, const sp<LooperCallback>& callback, void*
data);
```

fd表示要监听的描述符。ident表示要监听的事件的标识，值必须>=0或者为ALOOPER_POLL_CALLBACK(-2)，event表示要监听的事件，callback是事件发生时的回调函数，mAllowNonCallbacks的作用就在于此，当mAllowNonCallbacks为true时允许callback为NULL，在pollOnce中ident作为结果返回，否则不允许callback为空，当

callback不为NULL时，ident的值会被忽略。还是直接看代码方便理解：

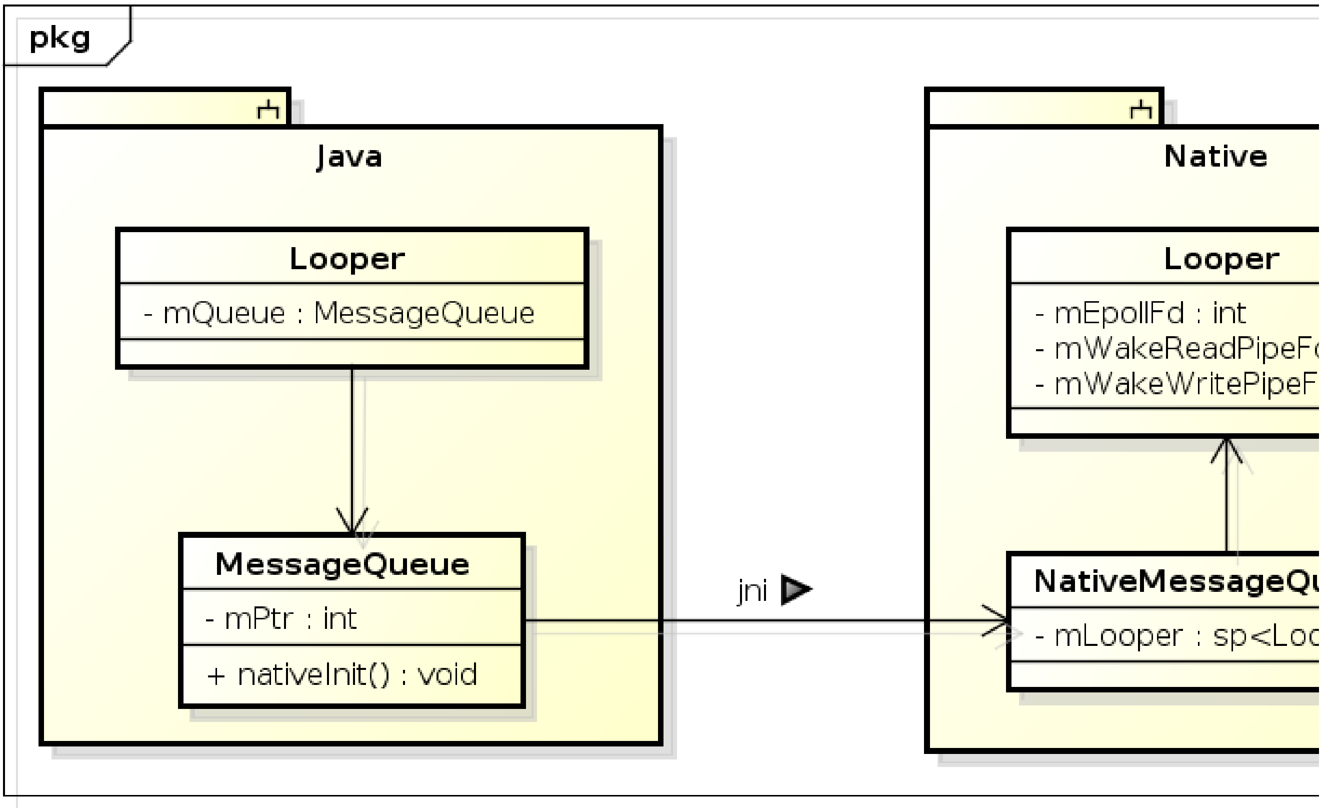
```
int Looper::addFd(int fd, int ident, int events, const sp<LooperCallback>& callback,
void* data) {
#if DEBUG_CALLBACKS
    ALOGD("%p ~ addFd - fd=%d, ident=%d, events=0x%x, callback=%p, data=%p", this,
fd, ident,
            events, callback.get(), data);
#endif
    if (!callback.get()) {
        if (! mAllowNonCallbacks) {
            ALOGE("Invalid attempt to set NULL callback but not allowed for this
looper.");
            return -1;
        }
        if (ident < 0) {
            ALOGE("Invalid attempt to set NULL callback with ident < 0.");
            return -1;
        }
    } else {
        ident = ALOOPER_POLL_CALLBACK;
    }

    int epollEvents = 0;
    if (events & ALOOPER_EVENT_INPUT) epollEvents |= EPOLLIN;
    if (events & ALOOPER_EVENT_OUTPUT) epollEvents |= EPOLLOUT;

    { // acquire lock
        AutoMutex _l(mLock);

        Request request;
        request.fd = fd;
        request.ident = ident;
        request.callback = callback;
        request.data = data;

        struct epoll_event eventItem;
        memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of
data field union
        eventItem.events = epollEvents;
        eventItem.data.fd = fd;

        ssize_t requestIndex = mRequests.indexOfKey(fd);
        if (requestIndex < 0) {
```

```
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, & eventItem);
            if (epollResult < 0) {
                ALOGE("Error adding epoll events for fd %d, errno=%d", fd, errno);
                return -1;
            }
            mRequests.add(fd, request);
        } else {
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_MOD, fd, & eventItem);
            if (epollResult < 0) {
                ALOGE("Error modifying epoll events for fd %d, errno=%d", fd, errno);
                return -1;
            }
            mRequests.replaceValueAt(requestIndex, request);
        }
    } // release lock
    return 1;
}
```

如果callback为空会检查mAllowNonCallbacks看是否允许callback为空，如果允许callback为空还会检测ident是否>=0。如果callback不为空会把ident的值赋值为ALOOPER_POLL_CALLBACK，不管传进来的是什么值。

接下来把传进来的参数值封装到一个Request结构体中，并以描述符为键保存到一个KeyedVector mRequests中，然后通过epoll_ctl添加或替换（如果这个描述符之前有调用addFD添加监听）对这个描述符事件的监听。

类图：



发送消息

通过Looper.prepare初始化好消息队列后就可以调用Looper.loop进入消息循环了，然后我们就可以向消息队列发送消息，消息循环就会取出消息进行处理，在看消息处理之前，先看一下消息是怎么被添加到消息队列的。

在Java层，Message类表示一个消息对象，要发送消息首先就要先获得一个消息对象，Message类的构造函数是public的，但是不建议直接new Message，Message内部保存了一个缓存的消息池，我们可以用obtain从缓存池获得一个消息，Message使用完后系统会调用recycle回收，如果自己new很多Message，每次使用完后系统放入缓存池，会占用很多内存的，如下所示：

```
public static Message obtain() {
    synchronized (sPoolSync) {
        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null;
            sPoolSize--;
            return m;
        }
    }
    return new Message();
}
```

```
    public void recycle() {
        clearForRecycle();

        synchronized (sPoolSync) {
            if (sPoolSize < MAX_POOL_SIZE) {
                next = sPool;
                sPool = this;
                sPoolSize++;
            }
        }
    }
```

Message内部通过next成员实现了一个链表，这样sPool就了为了一个Messages的缓存链表。

消息对象获取到了怎么发送呢，大家都知道是通过Handler的post、sendMessage等方法，其实这些方法最终都是调用的同一个方法sendMessageAtTime:

```
    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;
        }
        return enqueueMessage(queue, msg, uptimeMillis);
    }
```

sendMessageAtTime获取到消息队列然后调用enqueueMessage方法，消息队列mQueue是从与Handler关联的Looper获得的。

```
    private boolean enqueueMessage(MessageQueue queue, Message msg, long
uptimeMillis) {
        msg.target = this;
        if (mAsynchronous) {
            msg.setAsynchronous(true);
        }
        return queue.enqueueMessage(msg, uptimeMillis);
    }
```

enqueueMessage将message的target设置为当前的handler，然后调用MessageQueue的enqueueMessage，在调用queue.enqueueMessage之前判断了mAsynchronous，从名字看是异步消息的意思，要明白Asynchronous的作用，需要先了解一个概念Barrier。

Barrier与Asynchronous Message

Barrier是什么意思呢，从名字看是一个拦截器，在这个拦截器后面的消息都暂时无法执行，直到这个拦截器被移除了，MessageQueue有一个函数叫enqueueSyncBarier可以添加一个Barrier。

```
    int enqueueSyncBarrier(long when) {
        // Enqueue a new sync barrier token.
        // We don't need to wake the queue because the purpose of a barrier is to
stall it.
        synchronized (this) {
            final int token = mNextBarrierToken++;
            final Message msg = Message.obtain();
            msg.arg1 = token;

            Message prev = null;
            Message p = mMessages;
            if (when != 0) {
                while (p != null && p.when <= when) {
                    prev = p;
                    p = p.next;
                }
            }
            if (prev != null) { // invariant: p == prev.next
                msg.next = p;
                prev.next = msg;
            } else {
                msg.next = p;
                mMessages = msg;
            }
            return token;
        }
    }
```

在enqueueSyncBarrier中，obtain了一个Message，并设置msg.arg1=token，token仅是一个每次调用enqueueSyncBarrier时自增的int值，目的是每次调用enqueueSyncBarrier时返回唯一的一个token，这个Message同样需要设置执行时间，然后插入到消息队列，特殊的是这个Message没有设置target，即msg.target为null。

进入消息循环后会不停地从MessageQueue中取消息执行，调用的是MessageQueue的next函数，其中有这么一段：

```
Message msg = mMessages;
if (msg != null && msg.target == null) {
    // Stalled by a barrier.  Find the next asynchronous message in the queue.
    do {
        prevMsg = msg;
        msg = msg.next;
    } while (msg != null && !msg.isAsynchronous());
}
```

如果队列头部的消息的target为null就表示它是个Barrier，因为只有两种方法往mMessages中添加消息，一种是enqueueMessage，另一种是enqueueBarrier，而enqueueMessage中如果mst.target为null是直接抛异常的，后面会看到。

所谓的异步消息其实就是这样的，我们可以通过enqueueBarrier往消息队列中插入一个Barrier，那么队列中执行时间在这个Barrier以后的同步消息都会被这个Barrier拦截住无法执行，直到我们调用removeBarrier移除了这个Barrier，而异步消息则没有影响，消息默认就是同步消息，除非我们调用了Message的setAsynchronous，这个方法是隐藏的。只有在初始化Handler时通过参数指定往这个Handler发送的消息都是异步的，这样在Handler的enqueueMessage中就会调用Message的setAsynchronous设置消息是异步的，从上面Handler.enqueueMessage的代码中可以看到。

所谓异步消息，其实只有一个作用，就是在设置Barrier时仍可以不受Barrier的影响被正常处理，如果没有设置Barrier，异步消息就与同步消息没有区别，可以通过removeSyncBarrier移除Barrier：

```
void removeSyncBarrier(int token) {
    // Remove a sync barrier token from the queue.
    // If the queue is no longer stalled by a barrier then wake it.
    final boolean needWake;
    synchronized (this) {
        Message prev = null;
        Message p = mMessages;
```

```
        while (p != null && (p.target != null || p.arg1 != token)) {
            prev = p;
            p = p.next;
        }
        if (p == null) {
            throw new IllegalStateException("The specified message queue
synchronization "
                    + " barrier token has not been posted or has already been
removed.");
        }
        if (prev != null) {
            prev.next = p.next;
            needWake = false;
        } else {
            mMessages = p.next;
            needWake = mMessages == null || mMessages.target != null;
        }
        p.recycle();
    }
    if (needWake) {
        nativeWake(mPtr);
    }
}
```

参数token就是enqueueSyncBarrier的返回值，如果没有调用指定的token不存在是会抛异常的。

enqueueMessage

接下来看一下是怎么MessageQueue的enqueueMessage。

```
    final boolean enqueueMessage(Message msg, long when) {
        if (msg.isInUse()) {
            throw new AndroidRuntimeException(msg + " This message is already in
use.");
        }
        if (msg.target == null) {
            throw new AndroidRuntimeException("Message must have a target.");
        }

        boolean needWake;
        synchronized (this) {
```

```
            if (mQuiting) {
                RuntimeException e = new RuntimeException(
                        msg.target + " sending message to a Handler on a dead
thread");
                Log.w("MessageQueue", e.getMessage(), e);
                return false;
            }

            msg.when = when;
            Message p = mMessages;
            if (p == null || when == 0 || when < p.when) {
                // New head, wake up the event queue if blocked.
                msg.next = p;
                mMessages = msg;
                needWake = mBlocked;
            } else {
                // Inserted within the middle of the queue.  Usually we don't have to
wake
                // up the event queue unless there is a barrier at the head of the
queue
                // and the message is the earliest asynchronous message in the queue.
                needWake = mBlocked && p.target == null && msg.isAsynchronous();
                Message prev;
                for (;;) {
                    prev = p;
                    p = p.next;
                    if (p == null || when < p.when) {
                        break;
                    }
                    if (needWake && p.isAsynchronous()) {
                        needWake = false;
                    }
                }
                msg.next = p; // invariant: p == prev.next
                prev.next = msg;
            }
        }
        if (needWake) {
            nativeWake(mPtr);
        }
        return true;
    }
```

注意上面代码红色的部分，当msg.target为null时是直接抛异常的。

在enqueueMessage中首先判断，如果当前的消息队列为空，或者新添加的消息的执行时间when是0，或者新添加的消息的执行时间比消息队列头的消息的执行时间还早，就把消息添加到消息队列头（消息队列按时间排序），否则就要找到合适的位置将当前消息添加到消息队列。

Native发送消息

消息模型不只是Java层用的，Native层也可以用，前面也看到了消息队列初始化时也同时初始化了Native层的Looper与NativeMessageQueue，所以Native层应该也是可以发送消息的。与Java层不同的是，Native层是通过Looper发消息的，同样所有的发送方法最终是调用sendMessageAtTime：

```
void Looper::sendMessageAtTime(nsecs_t uptime, const sp<MessageHandler>& handler,
        const Message& message) {
#if DEBUG_CALLBACKS
    ALOGD("%p ~ sendMessageAtTime - uptime=%lld, handler=%p, what=%d",
            this, uptime, handler.get(), message.what);
#endif

    size_t i = 0;
    { // acquire lock
        AutoMutex _l(mLock);

        size_t messageCount = mMessageEnvelopes.size();
        while (i < messageCount && uptime >= mMessageEnvelopes.itemAt(i).uptime) {
            i += 1;
        }

        MessageEnvelope messageEnvelope(uptime, handler, message);
        mMessageEnvelopes.insertAt(messageEnvelope, i, 1);

        // Optimization: If the Looper is currently sending a message, then we can skip
        // the call to wake() because the next thing the Looper will do after processing
        // messages is to decide when the next wakeup time should be.  In fact, it does
        // not even matter whether this code is running on the Looper thread.
        if (mSendingMessage) {
            return;
        }
    } // release lock

    // Wake the poll loop only when we enqueue a new message at the head.
```

```
        if (i == 0) {
            wake();
        }
}
```

Native Message只有一个int型的what字段用来区分不同的消息，sendMessageAtTime指定了Message，Message要执行的时间when，与处理这个消息的Handler：MessageHandler，然后用MessageEnvelope封装了time, MessageHandler与Message，Native层发的消息都保存到了mMessageEnvelopes中，mMessageEnvelopes是一个Vector<MessageEnvelope>。Native层消息同样是按时间排序，与Java层的消息分别保存在两个队列里。

消息循环

消息队列初始化好了，也知道怎么发消息了，下面就是怎么处理消息了，看Handler.loop函数：

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

        for (;;) {
            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);
```

```
            }

            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.recycle();
        }
    }
```

loop每次从MessageQueue取出一个Message，调用msg.target.dispatchMessage(msg)，target就是发送message时跟message关联的handler，这样就调用到了熟悉的dispatchMessage，Message被处理后会被recycle。当queue.next返回null时会退出消息循环，接下来就看一下MessageQueue.next是怎么取出消息的，又会在什么时候返回null。

```
final Message next() {
        int pendingIdleHandlerCount = -1; // -1 only during first iteration
        int nextPollTimeoutMillis = 0;

        for (;;) {
            if (nextPollTimeoutMillis != 0) {
                Binder.flushPendingCommands();
            }
            nativePollOnce(mPtr, nextPollTimeoutMillis);

            synchronized (this) {
                if (mQuiting) {
                    return null;
                }
```

```
                // Try to retrieve the next message.  Return if found.
                final long now = SystemClock.uptimeMillis();
                Message prevMsg = null;
                Message msg = mMessages;
                if (msg != null && msg.target == null) {
                    // Stalled by a barrier.  Find the next asynchronous message in
the queue.
                    do {
                        prevMsg = msg;
                        msg = msg.next;
                    } while (msg != null && !msg.isAsynchronous());
                }
                if (msg != null) {
                    if (now < msg.when) {
                        // Next message is not ready.  Set a timeout to wake up when
it is ready.
                        nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
                    } else {
                        // Got a message.
                        mBlocked = false;
                        if (prevMsg != null) {
                            prevMsg.next = msg.next;
                        } else {
                            mMessages = msg.next;
                        }
                        msg.next = null;
                        if (false) Log.v("MessageQueue", "Returning message: " +
msg);
                        msg.markInUse();
                        return msg;
                    }
                } else {
                    // No more messages.
                    nextPollTimeoutMillis = -1;
                }

                // If first time idle, then get the number of idlers to run.
                // Idle handles only run if the queue is empty or if the first
message
                // in the queue (possibly a barrier) is due to be handled in the
future.
                if (pendingIdleHandlerCount < 0
                        && (mMessages == null || now < mMessages.when)) {
```

```
                    pendingIdleHandlerCount = mIdleHandlers.size();
                }
                if (pendingIdleHandlerCount <= 0) {
                    // No idle handlers to run.  Loop and wait some more.
                    mBlocked = true;
                    continue;
                }

                if (mPendingIdleHandlers == null) {
                    mPendingIdleHandlers = new
IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
                }
                mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
            }

            // Run the idle handlers.
            // We only ever reach this code block during the first iteration.
            for (int i = 0; i < pendingIdleHandlerCount; i++) {
                final IdleHandler idler = mPendingIdleHandlers[i];
                mPendingIdleHandlers[i] = null; // release the reference to the
handler

                boolean keep = false;
                try {
                    keep = idler.queueIdle();
                } catch (Throwable t) {
                    Log.wtf("MessageQueue", "IdleHandler threw exception", t);
                }

                if (!keep) {
                    synchronized (this) {
                        mIdleHandlers.remove(idler);
                    }
                }
            }

            // Reset the idle handler count to 0 so we do not run them again.
            pendingIdleHandlerCount = 0;

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

MessageQueue.next首先会调用nativePollOnce，然后如果mQuiting为true就返回null，Looper就会退出消息循环。

接下来取消息队列头部的消息，如果头部消息是Barrier（target==null）就往后遍历找到第一个异步消息，接下来检测获取到的消息（消息队列头部的消息或者第一个异步消息），如果为null表示没有消息要执行，设置nextPollTimeoutMillis = -1；否则检测这个消息要执行的时间，如果到执行时间了就将这个消息markInUse并从消息队列移除，然后从next返回到loop；否则设置nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE)，即距离最近要执行的消息还需要多久，无论是当前消息队列没有消息可以执行（设置了Barrier并且没有异步消息或消息队列为空）还是队列头部的消息未到执行时间，都会执行后面的代码，看有没有设置IdleHandler，如果有就运行IdleHandler，当IdleHandler被执行之后会设置nextPollTimeoutMillis = 0。

首先看一下nativePollOnce，native方法，调用JNI，最后调到了Native Looper::pollOnce，并从Java层传进去了nextPollTimeMillis，即Java层的消息队列中执行时间最近的消息还要多久到执行时间。

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
    int result = 0;
    for (;;) {
        while (mResponseIndex < mResponses.size()) {
            const Response& response = mResponses.itemAt(mResponseIndex++);
            int ident = response.request.ident;
            if (ident >= 0) {
                int fd = response.request.fd;
                int events = response.events;
                void* data = response.request.data;
#if DEBUG_POLL_AND_WAKE
                ALOGD("%p ~ pollOnce - returning signalled identifier %d: "
                        "fd=%d, events=0x%x, data=%p",
                        this, ident, fd, events, data);
#endif
                if (outFd != NULL) *outFd = fd;
                if (outEvents != NULL) *outEvents = events;
                if (outData != NULL) *outData = data;
                return ident;
            }
        }

        if (result != 0) {
#if DEBUG_POLL_AND_WAKE
            ALOGD("%p ~ pollOnce - returning result %d", this, result);
```

```
#endif
            if (outFd != NULL) *outFd = 0;
            if (outEvents != NULL) *outEvents = 0;
            if (outData != NULL) *outData = NULL;
            return result;
        }

        result = pollInner(timeoutMillis);
    }
}
```

先不看开始的一大串代码，先看一下pollInner：

```
int Looper::pollInner(int timeoutMillis) {
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ pollOnce - waiting: timeoutMillis=%d", this, timeoutMillis);
#endif

    // Adjust the timeout based on when the next message is due.
    if (timeoutMillis != 0 && mNextMessageUptime != LLONG_MAX) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        int messageTimeoutMillis = toMillisecondTimeoutDelay(now,
mNextMessageUptime);
        if (messageTimeoutMillis >= 0
                && (timeoutMillis < 0 || messageTimeoutMillis < timeoutMillis)) {
            timeoutMillis = messageTimeoutMillis;
        }
#if DEBUG_POLL_AND_WAKE
        ALOGD("%p ~ pollOnce - next message in %lldns, adjusted timeout:
timeoutMillis=%d",
                this, mNextMessageUptime - now, timeoutMillis);
#endif
    }

    // Poll.
    int result = ALOOPER_POLL_WAKE;
    mResponses.clear();
    mResponseIndex = 0;

    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS,
timeoutMillis);
```

```
    // Acquire lock.
    mLock.lock();

    // Check for poll error.
    if (eventCount < 0) {
        if (errno == EINTR) {
            goto Done;
        }
        ALOGW("Poll failed with an unexpected error, errno=%d", errno);
        result = ALOOPER_POLL_ERROR;
        goto Done;
    }

    // Check for poll timeout.
    if (eventCount == 0) {
#if DEBUG_POLL_AND_WAKE
        ALOGD("%p ~ pollOnce - timeout", this);
#endif
        result = ALOOPER_POLL_TIMEOUT;
        goto Done;
    }

    // Handle all events.
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ pollOnce - handling events from %d fds", this, eventCount);
#endif

    for (int i = 0; i < eventCount; i++) {
        int fd = eventItems[i].data.fd;
        uint32_t epollEvents = eventItems[i].events;
        if (fd == mWakeReadPipeFd) {
            if (epollEvents & EPOLLIN) {
                awoken();
            } else {
                ALOGW("Ignoring unexpected epoll events 0x%x on wake read pipe.",
epollEvents);
            }
        } else {
            ssize_t requestIndex = mRequests.indexOfKey(fd);
            if (requestIndex >= 0) {
                int events = 0;
                if (epollEvents & EPOLLIN) events |= ALOOPER_EVENT_INPUT;
                if (epollEvents & EPOLLOUT) events |= ALOOPER_EVENT_OUTPUT;
                if (epollEvents & EPOLLERR) events |= ALOOPER_EVENT_ERROR;
                if (epollEvents & EPOLLHUP) events |= ALOOPER_EVENT_HANGUP;
```

```
                pushResponse(events, mRequests.valueAt(requestIndex));
            } else {
                ALOGW("Ignoring unexpected epoll events 0x%x on fd %d that is "
                        "no longer registered.", epollEvents, fd);
            }
        }
    }
Done: ;

    // Invoke pending message callbacks.
    mNextMessageUptime = LLONG_MAX;
    while (mMessageEnvelopes.size() != 0) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);
        if (messageEnvelope.uptime <= now) {
            // Remove the envelope from the list.
            // We keep a strong reference to the handler until the call to
handleMessage
            // finishes.  Then we drop it so that the handler can be deleted *before*
            // we reacquire our lock.
            { // obtain handler
                sp<MessageHandler> handler = messageEnvelope.handler;
                Message message = messageEnvelope.message;
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();

#if DEBUG_POLL_AND_WAKE || DEBUG_CALLBACKS
                ALOGD("%p ~ pollOnce - sending message: handler=%p, what=%d",
                        this, handler.get(), message.what);
#endif
                handler->handleMessage(message);
            } // release handler

            mLock.lock();
            mSendingMessage = false;
            result = ALOOPER_POLL_CALLBACK;
        } else {
            // The last message left at the head of the queue determines the next
wakeup time.
            mNextMessageUptime = messageEnvelope.uptime;
            break;
        }
    }

    // Release lock.
    mLock.unlock();
```

```
    // Invoke all response callbacks.
    for (size_t i = 0; i < mResponses.size(); i++) {
        Response& response = mResponses.editItemAt(i);
        if (response.request.ident == ALOOPER_POLL_CALLBACK) {
            int fd = response.request.fd;
            int events = response.events;
            void* data = response.request.data;
#if DEBUG_POLL_AND_WAKE || DEBUG_CALLBACKS
            ALOGD("%p ~ pollOnce - invoking fd event callback %p: fd=%d, events=0x%x,
data=%p",
                    this, response.request.callback.get(), fd, events, data);
#endif
            int callbackResult = response.request.callback->handleEvent(fd, events,
data);
            if (callbackResult == 0) {
                removeFd(fd);
            }
            // Clear the callback reference in the response structure promptly
because we
            // will not clear the response vector itself until the next poll.
            response.request.callback.clear();
            result = ALOOPER_POLL_CALLBACK;
        }
    }
    return result;
}
```

Java层的消息都保存在了Java层MessageQueue的成员mMessages中，Native层的消息都保存在了Native Looper的mMessageEnvelopes中，这就可以说有两个消息队列，而且都是按时间排列的。timeOutMillis表示Java层下个要执行的消息还要多久执行，mNextMessageUpdate表示Native层下个要执行的消息还要多久执行，如果timeOutMillis为0，epoll_wait不设置TimeOut直接返回；如果为-1说明Java层无消息直接用Native的time out；否则pollInner取这两个中的最小值作为timeOut调用epoll_wait。当epoll_wait返回时就可能有以下几种情况：

1. 出错返回。
2. Time Out
3. 正常返回，描述符上有事件产生。

如果是前两种情况直接goto DONE。

否则就说明FD上有事件发生了，如果是mWakeReadPipeFd的EPOLLIN事件就调用awoken，如果不是mWakeReadPipeFd，那就是通过addFD添加的fd，在addFD中将要

监听的fd及其events，callback,data封装成了Request对象，并以fd为键保存到了KeyedVector mRequests中，所以在这里就以fd为键获得在addFD时关联的Request，并连同events通过pushResonse加入mResonse队列（Vector），Resonse仅是对events与Request的封装。如果是epoll_wait出错或timeout，就没有描述符上有事件，就不用执行这一段代码，所以直接goto DONE了。

```
void Looper::pushResponse(int events, const Request& request) {
    Response response;
    response.events = events;
    response.request = request;
    mResponses.push(response);
}
```

接下来进入DONE部分，从mMessageEnvelopes取出头部的Native消息，如果到达了执行时间就调用它内部保存的MessageeHandler的handleMessage处理并从Native 消息队列移除，设置result为ALOOPER_POLL_CALLBACK，否则计算mNextMessageUptime表示Native消息队列下一次消息要执行的时间。如果未到头部消息的执行时间有可能是Java层消息队列消息的执行时间小于Native层消息队列头部消息的执行时间，到达了Java层消息的执行时间epoll_wait TimeOut返回了，或都通过addFd添加的描述符上有事件发生导致epoll_wait返回，或者epoll_wait是出错返回。Native消息是没有Barrier与Asynchronous的。

最后，遍历mResponses（前面刚通过pushResponse存进去的），如果response.request.ident ==ALOOPER_POLL_CALLBACK，就调用注册的callback的handleEvent(fd, events, data)进行处理，然后从mResonses队列中移除，这次遍历完之后，mResponses中保留来来的就都是ident>=0并且callback为NULL的了。在NativeMessageQueue初始化Looper时传入了mAllowNonCallbacks为false，所以这次处理完后mResponses一定为空。

接下来返回到pollOnce。pollOnce是一个for循环，pollInner中处理了所有response.request.ident==ALOOPER_POLL_CALLBACK的Response，在第二次进入for循环后如果mResponses不为空就可以找到ident>0的Response，将其ident作为返回值返回由调用pollOnce的函数自己处理，在这里我们是在NativeMessageQueue中调用的Loope的pollOnce，没对返回值进行处理，而且mAllowNonCallbacks为false也就不可能进入这个循环。pollInner返回值不可能是0，或者说只可能是负数，所以pollOnce中的for循环只会执行两次，在第二次就返回了。

Native Looper可以单独使用，也有一个prepare函数，这时mAllowNonCallbakcs值可能为true，pollOnce中对mResponses的处理就有意义了。

wake与awoken

在Native Looper的构造函数中，通过pipe打开了一个管道，并用mWakeReadPipeFd与mWakeWritePipeFd分别保存了管道的读端与写端，然后用epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd,& eventItem)监听了读端的EPOLLIN事件，在pollInner中通过epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis)读取事件，那是在什么时候往mWakeWritePipeFd写，又是在什么时候读的mWakeReadPipeFd呢？

在Looper.cpp中我们可以发现如下两个函数：

```
void Looper::wake() {
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ wake", this);
#endif

    ssize_t nWrite;
    do {
        nWrite = write(mWakeWritePipeFd, "W", 1);
    } while (nWrite == -1 && errno == EINTR);

    if (nWrite != 1) {
        if (errno != EAGAIN) {
            ALOGW("Could not write wake signal, errno=%d", errno);
        }
    }
}

void Looper::awoken() {
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ awoken", this);
#endif

    char buffer[16];
    ssize_t nRead;
    do {
        nRead = read(mWakeReadPipeFd, buffer, sizeof(buffer));
    } while ((nRead == -1 && errno == EINTR) || nRead == sizeof(buffer));
}
```

wake函数向mWakeWritePipeFd写入了一个“W”字符，awoken从mWakeReadPipeFd读，往mWakeWritePipeFd写数据只是为了在pollInner中的epoll_wait

可以监听到事件返回。在pollInner也可以看到如果是mWakeReadPipeFd的EPOLLIN事件只是调用了awoken消耗掉了写入的字符就往后处理了。

那什么时候调用wake呢？这个只要找到调用的地方分析一下就行了，先看Looper.cpp，在sendMessageAtTime即发送Native Message的时候，根据发送的Message的执行时间查找mMessageEnvelopes计算应该插入的位置，如果是在头部插入，就调用wake唤醒epoll_wait，因为在进入pollInner时根据Java层消息队列头部消息的执行时间与Native层消息队列头部消息的执行时间计算出了一个timeout，如果这个新消息是在头部插入，说明执行时间至少在上述两个消息中的一个之前，所以应该唤醒epoll_wait，epoll_wait返回后，检查Native消息队列，看头部消息即刚插入的消息是否到执行时间了，到了就执行，否则就可能需要设置新的timeout。同样在Java层的MessageQueue中，有一个函数nativeWake也同样可以通过JNI调用wake，调用nativeWake的时机与在Native调用wake的时机类似，在消息队列头部插入消息，还有一种情况就是，消息队列头部是一个Barrier，而且插入的消息是第一个异步消息。

```
if (p == null || when == 0 || when < p.when) {
    // New head, wake up the event queue if blocked.
    msg.next = p;
    mMessages = msg;
    needWake = mBlocked;
} else {
    // Inserted within the middle of the queue.  Usually we don't have to wake
    // up the event queue unless there is a barrier at the head of the queue
    // and the message is the earliest asynchronous message in the queue.
    needWake = mBlocked && p.target == null && msg.isAsynchronous();//如果头部是Barrier并且新消息是异步消息则“有可能”需要唤醒
    Message prev;
    for (;;) {
        prev = p;
        p = p.next;
        if (p == null || when < p.when) {
            break;
        }
        if (needWake && p.isAsynchronous()) { // 消息队列中有异步消息并且执行时间在新消息之前，所以不需要唤醒。
            needWake = false;
        }
    }
    msg.next = p; // invariant: p == prev.next
    prev.next = msg;
}
```

在头部插入消息不一定调用nativeWake，因为之前可能正在执行IdleHandler，如果执行了IdleHandler，就在IdleHandler执行后把nextPollTimeoutMillis设置为0，下次进入for循环就用0调用nativePollOnce，不需要wake，只有在没有消息可以执行（消息队列为空或没到执行时间）并且没有设置IdleHandler时mBlocked才会为true。

如果Java层的消息队列被Barrier Block住了并且当前插入的是一个异步消息有可能需要唤醒Looper，因为异步消息可以在Barrier下执行，但是这个异步消息一定要是执行时间最早的异步消息。

退出Looper也需要wake，removeSyncBarrier时也可能需要。