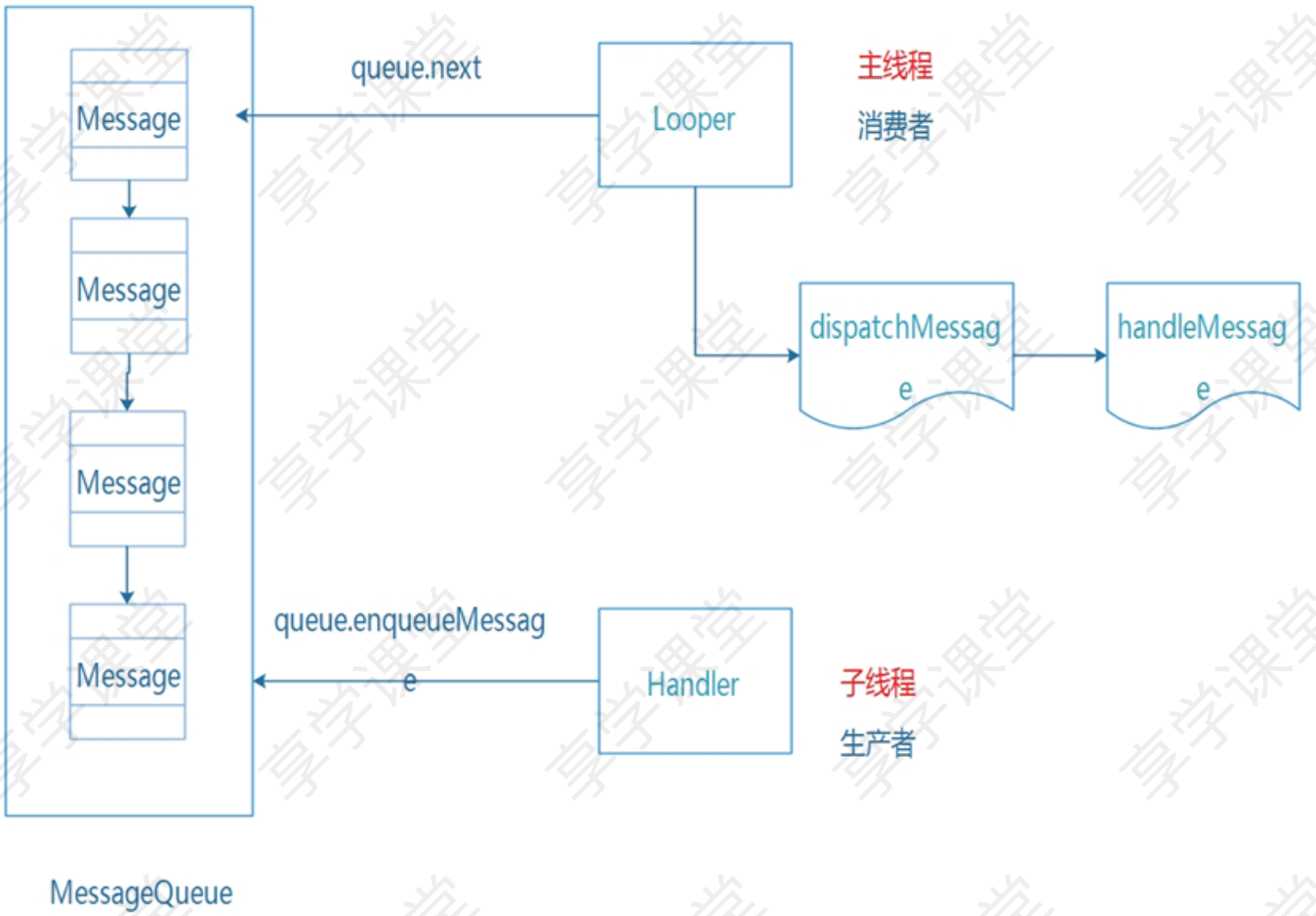
在android开发中，经常会在子线程中进行一些操作，当操作完毕后会通过handler发送一些数据给主线程，通知主线程做相应的操作。 **探索其背后的原理**：子线程 handler 主线程 其实构成了线程模型中的经典问题 生产者-消费者模型。 生产者-消费者模型：生产者和消费者在同一时间段内共用同一个存储空间，生产者往存储空间中添加数据，消费者从存储空间中取走数据。



好处： - 保证数据生产消费的顺序（通过MessageQueue,先进先出） - 不管是生产者(子线程)还是消费者（主线程）都只依赖缓冲区（handler），生产者消费者之间不会相互持有，使他们之间没有任何耦合

源码分析：

- Handler
  - Handler机制的相关类
  - 创建Looper
    - 创建MessageQueue以及Looper与当前线程的绑定
  - Looper.loop()
  - 创建Handler
  - 创建Message
  - Message和Handler的绑定
  - Handler发送消息
  - Handler处理消息

Handler机制的相关类

Hanlder：发送和接收消息 Looper：用于轮询消息队列，一个线程只能有一个Looper Message： 消息实体 MessageQueue： 消息队列用于存储消息和管理消息

创建Looper

> 创建Looper的方法是调用Looper.prepare() 方法

在ActivityThread中的main方法中为我们prepare了

```
public static void main(String[] args) {
        Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ActivityThreadMain");
        //其他代码省略...
        Looper.prepareMainLooper(); //初始化Looper以及MessageQueue

        ActivityThread thread = new ActivityThread();
        thread.attach(false);

        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }

        if (false) {
            Looper.myLooper().setMessageLogging(new
                    LogPrinter(Log.DEBUG, "ActivityThread"));
        }

        // End of event ActivityThreadMain.
        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        Looper.loop(); //开始轮循操作

        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
```

Looper.prepareMainLooper();

```
 public static void prepareMainLooper() {
        prepare(false);//消息队列不可以quit
        synchronized (Looper.class) {
            if (sMainLooper != null) {
                throw new IllegalStateException("The main Looper has already been
prepared.");
            }
            sMainLooper = myLooper();
        }
    }
```

prepare有两个重载的方法，主要看 prepare(boolean quitAllowed) quitAllowed的作用是在创建MessageQueue时标识消息队列是否可以销毁， **主线程不可被销毁** 下面有介绍

```
  public static void prepare() {
        prepare(true);//消息队列可以quit
    }
    //quitAllowed 主要
    private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {//不为空表示当前线程已经创建了Looper
            throw new RuntimeException("Only one Looper may be created per thread");
            //每个线程只能创建一个Looper
        }
        sThreadLocal.set(new Looper(quitAllowed));//创建Looper并设置给sThreadLocal，这样get的
时候就不会为null了
    }
```

创建MessageQueue以及Looper与当前线程的绑定

```
    private Looper(boolean quitAllowed) {
        mQueue = new MessageQueue(quitAllowed);//创建了MessageQueue
        mThread = Thread.currentThread(); //当前线程的绑定
    }
```

MessageQueue的构造方法

```
 MessageQueue(boolean quitAllowed) {
 //mQuitAllowed决定队列是否可以销毁 主线程的队列不可以被销毁需要传入false，在MessageQueue的quit()方法
就不贴源码了
        mQuitAllowed = quitAllowed;
        mPtr = nativeInit();
    }
```

Looper.loop()

同时是在main方法中 Looper.prepareMainLooper() 后Looper.loop(); 开始轮询

```
public static void loop() {
        final Looper me = myLooper();//里面调用了sThreadLocal.get()获得刚才创建的Looper对象
        if (me == null) {
            throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this
thread.");
        }//如果Looper为空则会抛出异常
        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 (;;) {
            //这是一个死循环，从消息队列不断的取消息
            Message msg = queue.next(); // might block
            if (msg == null) {
```

```
                //由于刚创建MessageQueue就开始轮询，队列里是没有消息的,等到Handler sendMessage enqueueMessage后
                //队列里才有消息
                // 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);//msg.target就是绑定的Handler，详见后面Message的部分，Handler开始
            //后面代码省略.....

            msg.recycleUnchecked();
        }
    }
```

创建Handler

最常见的创建handler

```
        Handler handler=new Handler(){
            @Override
            public void handleMessage(Message msg) {
                super.handleMessage(msg);
            }
        };
```

在内部调用 this(null, false);

```
public Handler(Callback callback, boolean async) {
        //前面省略
        mLooper = Looper.myLooper();//获取Looper，**注意不是创建Looper**!
        if (mLooper == null) {
            throw new RuntimeException(
                "Can't create handler inside thread that has not called Looper.prepare()");
        }
        mQueue = mLooper.mQueue;//创建消息队列MessageQueue
        mCallback = callback; //初始化了回调接口
        mAsynchronous = async;
    }
```

Looper.myLooper();

```
    //这是Handler中定义的ThreadLocal  ThreadLocal主要解多线程并发的问题
  // sThreadLocal.get() will return null unless you've called prepare().
   static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

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

sThreadLocal.get() will return null unless you've called prepare(). 这句话告诉我们get可能返回null 除非先调用prepare()方法创建Looper。在前面已经介绍了

创建Message

可以直接new Message 但是有更好的方式 Message.obtain。因为可以检查是否有可以复用的Message,用过复用避免过多的创建、销毁Message对象达到优化内存和性能的目地

```
public static Message obtain(Handler h) {
        Message m = obtain();//调用重载的obtain方法
        m.target = h;//并绑定的创建Message对象的handler

        return m;
    }

public static Message obtain() {
        synchronized (sPoolSync) {//sPoolSync是一个Object对象，用来同步保证线程安全
            if (sPool != null) {//sPool是就是handler dispatchMessage 后 通过recycleUnchecked 回收用以复用的Message
                Message m = sPool;
                sPool = m.next;
                m.next = null;
                m.flags = 0; // clear in-use flag
                sPoolSize--;
                return m;
            }
        }
        return new Message();
    }
```

Message和Handler的绑定

创建Message的时候可以通过 Message.obtain(Handler h) 这个构造方法绑定。当然可以在 在Handler 中的enqueueMessage（）也绑定了，所有发送Message的方法都会调用此方法入队，所以在创建Message的时候是可以不绑定的

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

Handler发送消息

Handler发送消息的重载方法很多，但是主要只有2种。 sendMessage(Message) sendMessage方法通过一系列重载方法的调用，sendMessage调用sendMessageDelayed，继续调用sendMessageAtTime，继续调用enqueueMessage，继续调用messageQueue的enqueueMessage方法，将消息保存在了消息队列中，而最终由Looper取出，交给Handler的dispatchMessage进行处理

我们可以看到在dispatchMessage方法中，message中callback是一个Runnable对象，如果callback不为空，则直接调用callback的run方法，否则判断mCallback是否为空，mCallback在Handler构造方法中初始化，在主线程通直接通过无参的构造方法new出来的为null,所以会直接执行后面的handleMessage()方法。

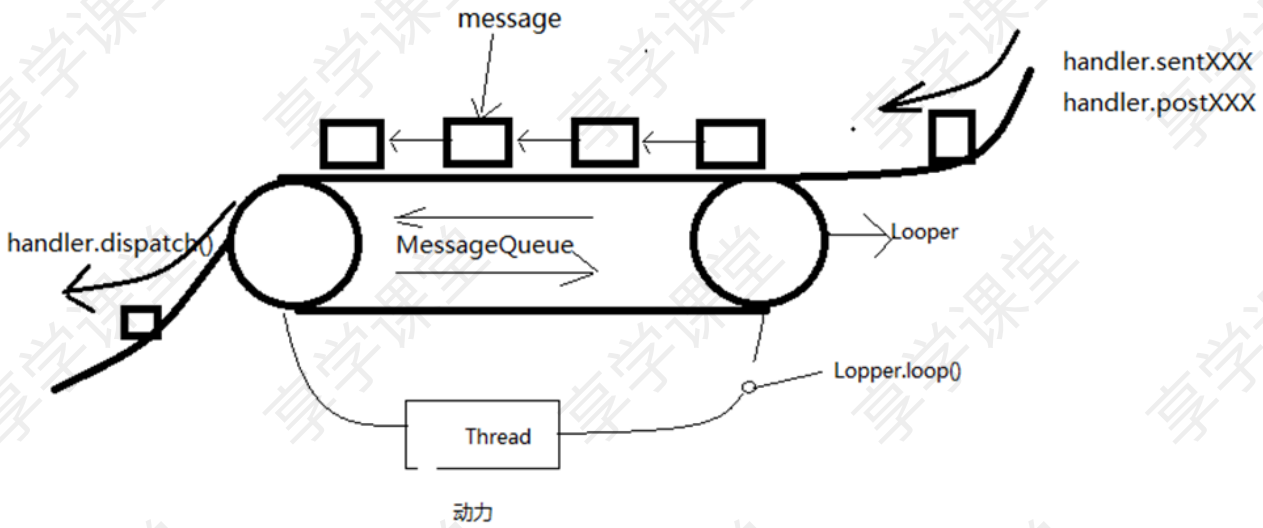
```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {//callback在message的构造方法中初始化或者使用
handler.post(Runnable)时候才不为空
        handleCallback(msg);
    } else {
        if (mCallback != null) {//mCallback是一个Callback对象，通过无参的构造方法创建出来的handler，
该属性为null，此段不执行
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);//最终执行handleMessage方法
    }
}

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

Handler处理消息

在handleMessage(Message)方法中，我们可以拿到message对象，根据不同的需求进行处理，整个Handler机制的流程就结束了。

小结

handler.sendMessage 发送消息到消息队列MessageQueue，然后looper调用自己的loop()函数带动MessageQueue从而轮询messageQueue里面的每个Message，当Message达到了可以执行的时间的时候开始执行，执行后就会调用message绑定的Handler来处理消息。大致的过程如下图所示



handler机制就是一个传送带的运转机制。

1）MessageQueue就像履带。

2）Thread就像背后的动力，就是我们通信都是基于线程而来的。

3）传送带的滚动需要一个开关给电机通电，那么就相当于我们的loop函数，而这个loop里面的for循环就会带着不断的滚动，去轮询messageQueue

4）Message就是 我们的货物了。

难点问题

1. 线程同步问题

Handler是用于线程间通信的，但是它产生的根本并不只是用于UI处理，而更多的是handler是整个app通信的框架，大家可以在ActivityThread里面感受到，整个App都是用它来进行线程间的协调。Handler既然这么重要，那么它的线程安全就至关重要了，那么它是如何保证自己的线程安全呢?

Handler机制里面最主要的类MessageQueue，这个类就是所有消息的存储仓库，在这个仓库中，我们如何的管理好消息，这个就是一个关键点了。消息管理就2点：1）消息入库（enqueueMessage），2）消息出库（next），所以这两个接口是确保线程安全的主要档口。

enqueueMessage 源码如下：

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

    // 锁开始的地方
    synchronized (this) {
        if (mQuitting) {
            IllegalStateException e = new IllegalStateException(
```

```
                    msg.target + " sending message to a Handler on a dead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }

        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        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;
        }

        // We can assume mPtr != 0 because mQuitting is false.
        if (needWake) {
            nativeWake(mPtr);
        }
    }
    //锁结束的地方
```

synchronized锁是一个内置锁，也就是由系统控制锁的lock unlock时机的。在多线程的课程中我们有详细分析过，有问题的同学可以去研究一下。

```
synchronized (this)
```

这个锁，说明的是对所有调用同一个MessageQueue对象的线程来说，他们都是互斥的，然而，在我们的Handler里面，一个线程是对应着一个唯一的Looper对象，而Looper中又只有一个唯一的MessageQueue（这个在上文中也有介绍）。所以，我们主线程就只有一个MessageQueue对象，也就是说，所有的子线程向主线程发送消息的时候，主线程一次都只会处理一个消息，其他的都需要等待，那么这个时候消息队列就不会出现混乱。

另外，在看next函数

```
Message next() {

    ....

    for (;;) {
        ....

        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // Try to retrieve the next message.  Return if found.
                    ...
                     return msg;
                }
            } else {
                // No more messages.
                nextPollTimeoutMillis = -1;
            }

             ...
        }//synchronized 结束之处

        // 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(TAG, "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;
    }
}
```

next函数很多同学会有疑问：我从线程里面取消息，而且每次都是队列的头部取，那么它加锁是不是没有意义呢？答案是否定的，我们必须要在next里面加锁，因为，这样由于synchronized（this）作用范围是所有 this正在访问的代码块都会有保护作用，也就是它可以保证 next函数和 enqueueMessage函数能够实现互斥。这样才能真正的保证多线程访问的时候messagequeue的有序进行。

小结： 这个地方是面试官经常问的点，而且他们会基于这个点来拓展问你多线程，所以，这个地方请大家重视。

2. 消息机制之同步屏障

同步屏障，view绘制中用 <https://juejin.im/post/6844903910113705998

同步屏障的概念，在Android开发中非常容易被人忽略，因为这个概念在我们普通的开发中太少见了，很容易被忽略。

大家经过上面的学习应该知道，线程的消息都是放到同一个MessageQueue里面，取消息的时候是互斥取消息，而且***只能从头部取消息***，**而添加消息是按照消息的执行的先后顺序进行的排序**，那么问题来了，同一个时间范围内的消息，如果它是需要立刻执行的，那我们怎么办，按照常规的办法，我们需要等到队列轮询到我自己的时候才能执行哦，那岂不是黄花菜都凉了。所以，我们需要给紧急需要执行的消息一个绿色通道，这个绿色通道就是同步屏障的概念。

同步屏障是什么?

屏障的意思即为阻碍，顾名思义，**同步屏障就是阻碍同步消息，只让异步消息通过**。如何开启同步屏障呢？如下而已：

```
MessageQueue#postSyncBarrier()
```

我们看看它的源码

-

```
    /**
     *
    @hide
    **/
       public int postSyncBarrier() {
            return postSyncBarrier(SystemClock.uptimeMillis());
       }
private int postSyncBarrier(long when) {
    // Enqueue a new sync barrier token
    synchronized (this) {
        final int token = mNextBarrierToken++;
        //从消息池中获取Message
        final Message msg = Message.obtain();
        msg.markInUse();

        //就是这里！！！初始化Message对象的时候，并没有给target赋值，因此 target==null
        msg.when = when;
        msg.arg1 = token;

        Message prev = null;
        Message p = mMessages;
```

```
        if (when != 0) {
            while (p != null && p.when <= when) {
          //如果开启同步屏障的时间（假设记为T）T不为0，且当前的同步消息里有时间小于T，则prev也不为null
                prev = p;
                p = p.next;
            }
        }
        //根据prev是不是为null，将 msg 按照时间顺序插入到 消息队列（链表）的合适位置
        if (prev != null) { // invariant: p == prev.next
            msg.next = p;
            prev.next = msg;
        } else {
            msg.next = p;
            mMessages = msg;
        }
        return token;
    }
}
```

可以看到，Message 对象初始化的时候并没有给 `target` 赋值，因此，`target == null` 的 来源就找到了。上面消息的插入也做了相应的注释。这样，一条 `target == null` 的消息就进入了消息队列。

那么，开启同步屏障后，所谓的异步消息又是如何被处理的呢？

如果对消息机制有所了解的话，应该知道消息的最终处理是在消息轮询器 `Looper#loop()` 中，而 `loop()` 循环中会调用 `MessageQueue#next()` 从消息队列中进行取消息。

//MessageQueue.java

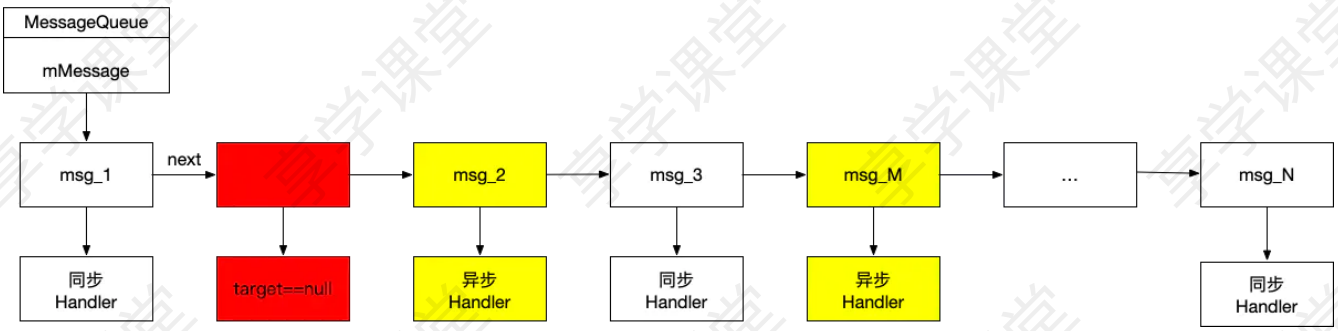
Message next()

```
    .....//省略一些代码
    int pendingIdleHandlerCount = -1; // -1 only during first iteration
    // 1.如果nextPollTimeoutMillis=-1，一直阻塞不会超时。
    // 2.如果nextPollTimeoutMillis=0，不会阻塞，立即返回。
    // 3.如果nextPollTimeoutMillis>0，最长阻塞nextPollTimeoutMillis毫秒(超时)
    //   如果期间有程序唤醒会立即返回。
    int nextPollTimeoutMillis = 0;
    //next()也是一个无限循环
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }
        nativePollOnce(ptr, nextPollTimeoutMillis);
        synchronized (this) {
            //获取系统开机到现在的时间
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages; //当前链表的头结点

            //关键！！！
            //如果target==null，那么它就是屏障，需要循环遍历，一直往后找到第一个异步的消息
            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) {
                //如果有消息需要处理，先判断时间有没有到，如果没到的话设置一下阻塞时间，
                //场景如常用的postDelay
                if (now < msg.when) {
                   //计算出离执行时间还有多久赋值给nextPollTimeoutMillis，
                   //表示nativePollOnce方法要等待nextPollTimeoutMillis时长后返回
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now,
Integer.MAX_VALUE);
                } else {
                    // 获取到消息
                    mBlocked = false;
                   //链表操作，获取msg并且删除该节点
                    if (prevMsg != null)
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    msg.markInUse();
                    //返回拿到的消息
                    return msg;
                }
            } else {
                //没有消息，nextPollTimeoutMillis复位
                nextPollTimeoutMillis = -1;
            }
            .....//省略

}
```

从上面可以看出，当消息队列开启同步屏障的时候（即标识为`msg.target == null`），消息机制在处理消息的时候，优先处理异步消息。这样，同步屏障就起到了一种过滤和优先级的作用。

下面用示意图简单说明：



如上图所示，在消息队列中有同步消息和异步消息（黄色部分）以及一道墙----同步屏障（红色部分）。有了同步屏障的存在，msg_2 和 msg_M 这两个异步消息可以被优先处理，而后面的 msg_3 等同步消息则不会被处理。那么这些同步消息什么时候可以被处理呢？那就需要先移除这个同步屏障，即调用 `removeSyncBarrier()`。

同步消息的应用场景

似乎在日常的应用开发中，很少会用到同步屏障。那么，同步屏障在系统源码中有哪些使用场景呢？Android 系统中的 UI 更新相关的消息即为异步消息，需要优先处理。

比如，在 View 更新时，draw、requestLayout、invalidate 等很多地方都调用了 `ViewRootImpl#scheduleTraversals()`，如下：

//ViewRootImpl.java

```
void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        //开启同步屏障
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        //发送异步消息
        mChoreographer.postCallback(
                Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
        if (!mUnbufferedInputDispatch) {
            scheduleConsumeBatchedInput();
        }
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}
```

`postCallback()` 最终走到了 `ChoreographerpostCallbackDelayedInternal()`：

```
  private void postCallbackDelayedInternal(int callbackType,
            Object action, Object token, long delayMillis) {
        if (DEBUG_FRAMES) {
            Log.d(TAG, "PostCallback: type=" + callbackType- ", action=" + action + ",
token=" + token  =" + delayMillis);
        }
        synchronized (mLock) {
        final long now = SystemClock.uptimeMillis();
        final long dueTime = now + delayMillis;
        mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);

        if (dueTime <= now) {
            scheduleFrameLocked(now);
        } else {
            Message msg = mHandler.obtainMessage(MSG_DO_SCHEDULE_CALLBACK, action);
            msg.arg1 = callbackType;
            msg.setAsynchronous(true);   //异步消息
            mHandler.sendMessageAtTime(msg, dueTime);
        }
    }
}
```

这里就开启了同步屏障，并发送异步消息，由于 UI 更新相关的消息是优先级最高的，这样系统就会优先处理这些异步消息。

最后，当要移除同步屏障的时候需要调用 `ViewRootImpl#unscheduleTraversals()` 。

```
    void unscheduleTraversals() {
        if (mTraversalScheduled) {
            mTraversalScheduled = false;
            //移除同步屏障
            mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
            mChoreographer.removeCallbacks(
                    Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
        }
    }
```

小结

同步屏障的设置可以方便地处理那些优先级较高的异步消息。当我们调用 `Handler.getLooper().getQueue().postSyncBarrier()` 并设置消息的 `setAsynchronous(true)` 时，target 即为 null ，也就开启了同步屏障。当在消息轮询器 Looper 在 `loop()` 中循环处理消息时，如若开启了同步屏障，会优先处理其中的异步消息，而阻碍同步消息。

Handler常问面试题

请大家研究一下，课程中会重点讲解。

1. 一个线程有几个 Handler?

2.一个线程有几个 Looper？如何保证?

3.Handler内存泄漏原因？ 为什么其他的内部类没有说过有这个问题?

4.为何主线程可以new Handler？如果想要在子线程中new Handler 要做些什么准备?

5.子线程中维护的Looper，消息队列无消息的时候的处理方案是什么？有什么用?

6.既然可以存在多个 Handler 往 MessageQueue 中添加数据（发消息时各个 Handler 可能处于不同线程），那它内部是如何确保线程安全的？取消息呢?

7.我们使用 Message 时应该如何创建它?

8.Looper死循环为什么不会导致应用卡死