- NETTY

精尽 Netty 源码解析 —— ChannelHandler（六）之 AbstractTrafficShapingHandler

笔者先把 Netty 主要的内容写完，所以关于 AbstractTrafficShapingHandler 的分享，先放在后续的计划里。

当然，良心如我，还是为对这块感兴趣的胖友，先准备好了一篇不错的文章：

- tomas家的小拨浪鼓 《Netty 那些事儿 ——— Netty实现“流量整形”原理分析及实战》

为避免可能 《Netty 那些事儿 ——— Netty实现“流量整形”原理分析及实战》 被作者删除，笔者这里先复制一份作为备份。

666. 备份

> 本文是Netty文集中“Netty 那些事儿”系列的文章。主要结合在开发实战中，我们遇到的一些“奇奇怪怪”的问题，以及如何正确且更好的使用Netty框架，并会对Netty中涉及的重要设计理念进行介绍。

Netty实现“流量整形”原理分析

流量整形

流量整形（Traffic Shaping）是一种主动调整流量输出速率的措施。流量整形与流量监管的主要区别在于，流量整形对流量监管中需要丢弃的报文进行缓存——通常是将它们放入缓冲区或队列内，也称流量整形（Traffic Shaping，简称TS）。当报文的发送速度过快时，首先在缓冲区进行缓存；再通过流量计量算法的控制下“均匀”地发送这些被缓冲的报文。流量整形与流量监管的另一区别是，整形可能会增加延迟，而监管几乎不引入额外的延迟。

Netty提供了GlobalTrafficShapingHandler、ChannelTrafficShapingHandler、GlobalChannelTrafficShapingHandler三个类来实现流量整形，他们都是AbstractTrafficShapingHandler抽象类的实现类，下面我们就对其进行介绍，让我们来了解Netty是如何实现流量整形的。

文章目录

afficShapingHandler）或者每个session的带宽（见

ounter会在每个检测间期（checkInterval）调用这个处理器

限制或者改变检测间期（checkInterval），可以使用如下方

nterval）；

启动监控，直接改变检测间期（checkInterval），或去访问它

它会根据给定的检测间期周期性的计算统计入站和出站的流量，并会回调AbstractTrafficShapingHandler的doAccounting方法。
如果检测间期（checkInterval）是0，将不会进行计数并且统计只会在每次读或写操作时进行计算。

- configure

```
public void configure(long newWriteLimit, long newReadLimit,
        long newCheckInterval) {
    configure(newWriteLimit, newReadLimit);
    configure(newCheckInterval);
}
```

配置新的写限制、读限制、检测间期。该方法会尽最大努力进行此更改，这意味着已经被延迟进行的流量将不会使用新的配置，它仅用于新的流量中。

- ReopenReadTimerTask

```
static final class ReopenReadTimerTask implements Runnable {
    final ChannelHandlerContext ctx;
    ReopenReadTimerTask(ChannelHandlerContext ctx) {
        this.ctx = ctx;
    }

    @Override
    public void run() {
        ChannelConfig config = ctx.channel().config();
        if (!config.isAutoRead() && isHandlerActive(ctx)) {
            // If AutoRead is False and Active is True, user make a direct setAutoRead(false)
            // Then Just reset the status
            if (logger.isDebugEnabled()) {
                logger.debug("Not unsuspend: " + config.isAutoRead() + ':' +
                        isHandlerActive(ctx));
            }
            ctx.attr(READ_SUSPENDED).set(false);
        } else {
            // Anything else allows the handler to reset the AutoRead
            if (logger.isDebugEnabled()) {
                if (config.isAutoRead() && !isHandlerActive(ctx)) {
                    fig.isAutoRead() + ':' +


                    " + config.isAutoRead() + ':'
```

```
" + config.isAutoRead() + ':'
        }
    }
}
```

重启读操作的定时任务。该定时任务总会实现：
a) 如果Channel的autoRead为false，并且AbstractTrafficShapingHandler的READ_SUSPENDED属性设置为null或false（说明读暂停未启用或开启），则直接将READ_SUSPENDED属性设置为false。
b) 否则，如果Channel的autoRead为true，或者READ_SUSPENDED属性的值为true（说明读暂停开启了），则将READ_SUSPENDED属性设置为false，并将Channel的autoRead标识为true（该操作底层会将该Channel的OP_READ事件重新注册为感兴趣的事件，这样Selector就会监听该Channel的读就绪事件了），最后触发一次Channel的read操作。
也就说，若“读操作”为“开启”状态（READ_SUSPENDED为null或false）的情况下，Channel的autoRead是保持Channel原有的配置，此时并不会做什么操作。但当“读操作”从“暂停”状态（READ_SUSPENDED为true）转为“开启”状态（READ_SUSPENDED为false）时，则会将Channel的autoRead标志为true，并将“读操作”设置为“开启”状态（READ_SUSPENDED为false）。

- channelRead

```
public void channelRead(final ChannelHandlerContext ctx, final Object msg) throws Exception {
    long size = calculateSize(msg);
    long now = TrafficCounter.milliSecondFromNano();
    if (size > 0) {
        // compute the number of ms to wait before reopening the channel
        long wait = trafficCounter.readTimeToWait(size, readLimit, maxTime, now);
        wait = checkWaitReadTime(ctx, wait, now);
        if (wait >= MINIMAL_WAIT) { // At least 10ms seems a minimal
            // time in order to try to limit the traffic
            // Only AutoRead AND HandlerActive True means Context Active
            ChannelConfig config = ctx.channel().config();
            if (logger.isDebugEnabled()) {
                logger.debug("Read suspend: " + wait + ':' + config.isAutoRead() + ':'
                        + isHandlerActive(ctx));
            }
            if (config.isAutoRead() && isHandlerActive(ctx)) {
                config.setAutoRead(false);
                ctx.attr(READ_SUSPENDED).set(true);
                // Create a Runnable to reactive the read if needed. If one was create before it will
                // reused to limit object creation
                Attribute<Runnable> attr = ctx.attr(REOPEN_TASK);
```



```
Task(ctx);
wait, TimeUnit.MILLISECONDS);
us => " + config.isAutoRead() + ':'
" will reopened at: " + wait);
```

```
}
```

① 『long size = calculateSize(msg);』计算本次读取到的消息的字节数。
② 如果读取到的字节数大于0，则根据数据的大小、设定的readLimit、最大延迟时间等计算（『long wait = trafficCounter.readTimeToWait(size, readLimit, maxTime, now);』）得到下一次开启读操作需要的延迟时间（距当前时间而言）wait(毫秒)。
③ 如果a）wait >= MINIMAL_WAIT(10毫秒)。并且b）当前Channel为自动读取（即，autoRead为true）以及c）当前的READ_SUSPENDED标识为null或false（即，读操作未被暂停），那么将Channel的autoRead设置为false（该操作底层会将该Channel的OP_READ事件从感兴趣的事件中移除，这样Selector就不会监听该Channel的读就绪事件了），并且将READ_SUSPENDED标识为true（说明，接下来的读操作会被暂停），并将“重新开启读操作“封装为一个任务，让入Channel所注册NioEventLoop的定时任务队列中（延迟wait时间后执行）。
也就说，只有当计算出的下一次读操作的时间大于了MINIMAL_WAIT(10毫秒)，并且当前Channel是自动读取的，且“读操作”处于“开启”状态时，才会去暂停读操作，而暂停读操作主要需要完成三件事：[1]将Channel的autoRead标识设置为false，这使得OP_READ会从感兴趣的事件中移除，这样Selector就会不会监听这个Channel的读就绪事件了；[2]将“读操作”状态设置为“暂停”（READ_SUSPENDED为true）；[3]将重启开启“读操作”的操作封装为一个task，在延迟wait时间后执行。
当你将得Channel的autoRead都会被设置为false时，Netty底层就不会再去执行读操作了，也就是说，这时如果有数据过来，会先放入到内核的接收缓冲区，只有我们执行读操作的时候数据才会从内核缓冲区读取到用户缓冲区中。而对于TCP协议来说，你不要担心一次内核缓冲区会溢出。因为如果应用进程一直没有读取，接收缓冲区满了之后，发生的动作是：通知对端TCP协议中的窗口关闭。这个便是滑动窗口的实现。保证TCP套接口接收缓冲区不会溢出，从而保证了TCP是可靠传输。因为对方不允许发出超过所通告窗口大小的数据。 这就是TCP的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方TCP将丢弃它。
④ 将当前的消息发送给ChannelPipeline中的下一个ChannelInboundHandler。

- write

```
public void write(final ChannelHandlerContext ctx, final Object msg, final ChannelPromise promise)
        throws Exception {
    long size = calculateSize(msg);
    long now = TrafficCounter.milliSecondFromNano();
    if (size > 0) {
        // compute the number of ms to wait before continue with the channel
        long wait = trafficCounter.writeTimeToWait(size, writeLimit, maxTime, now);
        if (wait >= MINIMAL_WAIT) {
            if (logger.isDebugEnabled()) {
                logger.debug("Write suspend: " + wait + ':' + ctx.channel().config().isAutoRead() + ':
                        + isHandlerActive(ctx));
            }
            submitWrite(ctx, msg, size, wait, now, promise);
```


eLimit、最大延迟时间等计算（『long wait =
）得到本次写操作需要的延迟时间（距当前时间而言）

te(ctx, msg, size, wait, now, promise);』wait即为延迟时间，（10毫秒），则调用『submitWrite(ctx, msg, size, 0, now,

GlobalTrafficShapingHandler

这实现了AbstractTrafficShapingHandler的全局流量整形，也就是说它限制了全局的带宽，无论开启了几个channel。
注意『 OutboundBuffer.setUserDefinedWritability(index, boolean)』中索引使用’2’。

一般用途如下：
创建一个唯一的GlobalTrafficShapingHandler

```
GlobalTrafficShapingHandler myHandler = new GlobalTrafficShapingHandler(executor);
pipeline.addLast(myHandler);
```

executor可以是底层的IO工作池

注意，这个处理器是覆盖所有管道的，这意味着只有一个处理器对象会被创建并且作为所有channel间共享的计数器，它必须于所有的channel共享。
所有你可以见到，该类的定义上面有个 `@Sharable` 注解。

在你的处理器中，你需要考虑使用『channel.isWritable()』和『channelWritabilityChanged(ctx)』来处理可写性，或通过在ctx.write()返回的future上注册listener来实现。

你还需要考虑读或写操作对象的大小需要和你要求的带宽相对应：比如，你将一个10M大小的对象用于10KB/s的带宽将会导致爆发效果，若你将100KB大小的对象用于在1M/s带宽那么将会被流量整形处理器平滑处理。

一旦不在需要这个处理器时请确保调用『release()』以释放所有内部的资源。这不会关闭EventExecutor，因为它可能是共享的，所以这需要你自己做。

GlobalTrafficShapingHandler中持有一个Channel的哈希表，用于存储当前应用所有的Channel：

```
private final ConcurrentMap<Integer, PerChannel> channelQueues = PlatformDependent.newConcurrentHashMa
```

key为Channel的hashCode；value是一个PerChannel对象。
PerChannel对象中维护有该Channel的待发送数据的消息队列（ArrayDeque messagesQueue）。

- submitWrite

```
void submitWrite(final ChannelHandlerContext ctx, final Object msg,
        final long size, final long writedelay, final long now,
        final ChannelPromise promise) {
    Channel channel = ctx.channel();
    Integer key = channel.hashCode();
```


is raised for this handler

eue.isEmpty()) {

- 注意事项
- 扩展

关于 “OP_WRITE” 与 “Channel#isWritable()”

```
(size);
            return;
        }
        if (delay > maxTime && now + delay - perChannel.lastWriteTimestamp > maxTime) {
            delay = maxTime;
        }
        newToSend = new ToSend(delay + now, msg, size, promise);
        perChannel.messagesQueue.addLast(newToSend);
        perChannel.queueSize += size;
        queuesSize.addAndGet(size);
        checkWriteSuspend(ctx, delay, perChannel.queueSize);
        if (queuesSize.get() > maxGlobalWriteSize) {
            globalSizeExceeded = true;
        }
    }
    if (globalSizeExceeded) {
        setUserDefinedWritability(ctx, false);
    }
    final long futureNow = newToSend.relativeTimeAction;
    final PerChannel forSchedule = perChannel;
    ctx.executor().schedule(new Runnable() {
        @Override
        public void run() {
            sendAllValid(ctx, forSchedule, futureNow);
        }
    }, delay, TimeUnit.MILLISECONDS);
}
```

写操作提交上来的数据。

① 如果写延迟为0，且当前该Channel的messagesQueue为空（说明，在此消息前没有待发送的消息了），那么直接发送该消息包。并返回，否则到下一步。

② 『newToSend = new ToSend(delay + now, msg, size, promise);
perChannel.messagesQueue.addLast(newToSend);』

将待发送的数据封装成ToSend对象放入PerChannel的消息队列中（messagesQueue）。注意，这里的messagesQueue是一个ArrayDeque队列，我们总是从队列尾部插入。然后从队列的头获取消息来依次发送，这就保证了消息的有序性。但是，如果一个大数据包前于一个小数据包发送的话，小数据包也会因为大数据包的延迟发送而被延迟到大数据包发送后才会发送。

ToSend 对象中持有带发送的数据对象、发送的相对延迟时间（即，根据数据包大小以及设置的写流量限制值（writeLimit）等计算出来的延迟操作的时间）、消息数据的大小、异步写操作的promise。

③ 『checkWriteSuspend(ctx, delay, perChannel.queueSize);』

检查单个Channel待发送的数据包是否超过了maxWriteSize（默认4M），或者延迟时间是否超过了maxWriteDelay（默认
);』该方法会将ChannelOutboundBuffer中的unwritable属性
true。以及会在unwritable从0到非0间变化时触发

的数据大小）大于了maxGlobalWriteSize（默认400M），
ritability(ctx, false)』将ChannelOutboundBuffer中的

me默认15s）delay，将『sendAllValid(ctx, forSchedule,
务队列中。
Queue，依次取出perChannel.messagesQueue中的消息
送给到ChannelPipeline中的下一个ChannelOutboundHandler
perChannel.queueSize（当前Channel待发送的总数据大

文章目录

立的值（即，被发送出去的这个数据包的大小）。循环遍历前
该Channel的消息队列中的消息全部都发送出去的话（即，
teSuspended(ctx);』来释放写暂停。而该方法底层会将
ChannelOutboundBuffer中的unwritable属性值相应的标志位重置。

ChannelTrafficShapingHandler

ChannelTrafficShapingHandler是针对单个Channel的流量整形，和GlobalTrafficShapingHandler的思想是一样的。只是实现中没有对全局概念的检测，仅检测了当前这个Channel的数据。
这里就不再赘述了。

GlobalChannelTrafficShapingHandler

相比于GlobalTrafficShapingHandler增加了一个误差概念，以平衡各个Channel间的读/写操作。也就是说，使得各个Channel间的读/写操作尽量均衡。比如，尽量避免不同Channel的大数据包都延迟近乎一样的是时间再操作，以及如果小数据包在一个大数据包后才发送，则减少该小数据包的延迟发送时间等。。

“流量整形”实战

这里仅展示服务端和客户端中使用“流量整形”功能涉及的关键代码，完整demo可见github
服务端
使用GlobalTrafficShapingHandler来实现服务端的“流量整形”，每当有客户端连接至服务端时服务端就会开始往这个客户端发送26M的数据包。我们将GlobalTrafficShapingHandler的writeLimit设置为10M/S。并使用了ChunkedWriteHandler来实现大数据包拆分成小数据包发送的功能。

MyServerInitializer实现：在ChannelPipeline中注册了GlobalTrafficShapingHandler

```
public class MyServerInitializer extends ChannelInitializer<SocketChannel> {

    Charset utf8 = Charset.forName("utf-8");
    final int M = 1024 * 1024;

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {

        GlobalTrafficShapingHandler globalTrafficShapingHandler = new GlobalTrafficShapingHandler(ch.e
//        globalTrafficShapingHandler.setMaxGlobalWriteSize(50 * M);
//        globalTrafficShapingHandler.setMaxWriteSize(5 * M);

        ch.pipeline()
                .addLast("LengthFieldBasedFrameDecoder", new LengthFieldBasedFrameDecoder(Integer.MAX_
                .addLast("LengthFieldPrepender", new LengthFieldPrepender(4, 0))
                .addLast("GlobalTrafficShapingHandler", globalTrafficShapingHandler)
                .addLast("chunkedWriteHandler", new ChunkedWriteHandler())
                .addLast("myServerChunkHandler", new MyServerChunkHandler())
                Decoder(utf8))
                Encoder(utf8))
                erverHandlerForPlain());
```


。并且通过『Channel#isWritable』方法以及
需要停止数据的写出，啥时可以开始继续写出数据。同时写了

- 注意事项
- 扩展

关于 “OP_WRITE” 与 “Channel#isWritable()”

```
erCommonHandler {

    protected void sentData(ChannelHandlerContext ctx) {
        sentFlag = true;
        ctx.writeAndFlush(tempStr, getChannelProgressivePromise(ctx, future -> {
            if(ctx.channel().isWritable() && !sentFlag) {
                sentData(ctx);
            }
        }));
    }

    @Override
    public void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception {
        if(ctx.channel().isWritable() && !sentFlag) {
//            System.out.println(" ###### 重新开始写数据 ######");
            sentData(ctx);
        } else {
//            System.out.println(" ===== 写暂停 =====");
        }
    }
}

public abstract class MyServerCommonHandler extends SimpleChannelInboundHandler<String> {

    protected final int M = 1024 * 1024;
    protected String tempStr;
    protected AtomicLong consumeMsgLength;
    protected Runnable counterTask;
    private long priorProgress;
    protected boolean sentFlag;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        consumeMsgLength = new AtomicLong();
        counterTask = () -> {
          while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {

            }
```

文章目录

- 注意事项
- 扩展

关于 “OP_WRITE” 与 “Channel#isWritable()”

```
ctx) throws Exception {
        sentData(ctx);
        new Thread(counterTask).start();
    }

    protected ChannelProgressivePromise getChannelProgressivePromise(ChannelHandlerContext ctx, Consum
        ChannelProgressivePromise channelProgressivePromise = ctx.newProgressivePromise();
        channelProgressivePromise.addListener(new ChannelProgressiveFutureListener(){
            @Override
            public void operationProgressed(ChannelProgressiveFuture future, long progress, long total
                consumeMsgLength.addAndGet(progress - priorProgress);
                priorProgress = progress;
            }

            @Override
            public void operationComplete(ChannelProgressiveFuture future) throws Exception {
                sentFlag = false;
                if(future.isSuccess()){
                    System.out.println("成功发送完成！");
                    priorProgress -= 26 * M;
                    Optional.ofNullable(completedAction).ifPresent(action -> action.accept(future));
                } else {
                    System.out.println("发送失败！！！！！");
                    future.cause().printStackTrace();
                }
            }
        });
        return channelProgressivePromise;
    }

    protected abstract void sentData(ChannelHandlerContext ctx);

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {
        System.out.println("===== receive client msg : " + msg);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        cause.printStackTrace();
        ctx.channel().close();
```

文章目录

流量整形”，并将readLimit设置为1M/S。

```
ializer<SocketChannel> {
```

注意事项
- 扩展

关于 “OP_WRITE” 与 “Channel#isWritable()”

```
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelTrafficShapingHandler channelTrafficShapingHandler = new ChannelTrafficShapingHandler(1
        ch.pipeline()
                .addLast("channelTrafficShapingHandler",channelTrafficShapingHandler)
                .addLast("lengthFieldBasedFrameDecoder", new LengthFieldBasedFrameDecoder(Integer.MAX_
                .addLast("lengthFieldPrepender", new LengthFieldPrepender(4, 0))
                .addLast("stringDecoder", new StringDecoder(utf8))
                .addLast("stringEncoder", new StringEncoder(utf8))
                .addLast("myClientHandler", new MyClientHandler());
    }
}
```

注意事项

① 注意，trafficShaping是通过程序来达到控制流量的作用，并不是网络层真实的传输流量大小的控制。TrafficShapingHandler仅仅是根据消息大小（待发送出去的数据包大小）和设定的流量限制来得出延迟发送该包的时间，即同一时刻不会发送过大的数据导致带宽负荷不了。但是并没有对大数据包进行拆分的作用，这会使在发送这个大数据包时同样可能会导致带宽爆掉的情况。所以你需要注意一次发送数据包的大小，不要大于你设置限定的写带宽大小(writeLimit)。你可以通过在业务handler中自己控制的方式，或者考虑使用ChunkedWriteHandler，如果它能满足你的要求的话。同时注意，不要将writeLimit和readLimit设置的过小，这是没有意义的，只会导致读/写操作的不断停顿。。
② 注意，不要在非NioEventLoop线程中不停歇的发送非ByteBuf、ByteBufHolder或者FileRegion对象的大数据包，如：

```
new Thread(() -> {
        while (true) {
            if(ctx.channel().isWritable()) {
                ctx.writeAndFlush(tempStr, getChannelProgressivePromise(ctx, null));
            }
        }
    }).start();
```

因为写操作是一个I/O操作，当你在非NioEventLoop线程上执行了Channel的I/O操作的话，该操作会封装为一个task 被提交至NioEventLoop的任务队列中，以使得I/O操作最终是NioEventLoop线程上得到执行。
而提交这个任务的流程，仅会对ByteBuf、ByteBufHolder或者FileRegion对象进行真实数据大小的估计（其他情况默认估计大小为8 bytes），并将估计后的数据大小值对该ChannelOutboundBuffer的totalPendingSize属性值进行累加。而totalPendingSize同WriteBufferWaterMark一起来控制着Channel的unwritable。所以，如果你在一个非NioEventLoop线程中不断地发送一个非ByteBuf、ByteBufHolder或者FileRegion对象的大数据包时，最终就会导致提交大量的任务到NioEventLoop线程的任务队列中，而当NioEventLoop线程在真实执行这些task时可能发生OOM。

文章目录

able()” 虽然都是的对数据的可写性进行检测，但是它们是分

rite操作（这里是真实写操作了，将数据通过TCP协议进事件。这样当发送缓冲区空闲时就OP_WRITE事件就会触统层面的可写性的一种检测。
据大小超过了我们设定的相关最大写数据大小，如果超过行write操作了（这里写操作一般为通过

行过不少介绍了，这里不再赘述。下面我们来看看

```
public boolean isWritable() {
    return unwritable == 0;
}
```

ChannelOutboundBuffer 的 unwritable属性为0时，Channel的isWritable()方法将返回true；否之，返回false；unwritable可以看做是一个二进制的开关属性值。它的二进制的不同位表示的不同状态的开关。如：

img

ChannelOutboundBuffer有四个方法会对unwritable属性值进行修改：clearUserDefinedWritability、setUnwritable、setUserDefinedWritability、setWritable。并且，当unwritable从0到非0间改变时还会触发ChannelWritabilityChanged事件，以通知ChannelPipeline中的各个ChannelHandler当前Channel可写性发生了改变。

其中setUnwritable、setWritable这对方法是由于待写数据大小高于或低于了WriteBufferWaterMark的水位线而导致的unwritable属性值的改变。
我们所执行的『ChannelHandlerContext#write』和『Channel#write』操作会先将待发送的数据包放到Channel的输出缓冲区（ChannelOutboundBuffer）中，然后在执行flush操作的时候，会从ChannelOutboundBuffer中依次出去数据包进行真实的网络数据传输。而WriteBufferWaterMark控制的就是ChannelOutboundBuffer中待发送的数据总大小（即，totalPendingSize：包含一个个ByteBuf中待发送的数据大小，以及数据包对象占用的大小）。如果totalPendingSize的大小超过了WriteBufferWaterMark高水位（默认为64KB），则会unwritable属性的’WriteBufferWaterMark状态位’置位1；随着数据不断写出（每写完一个ByteBuf后，就会将totalPendingSize减少相应的值），当totalPendingSize的大小小于WriteBufferWaterMark低水位（默认为32KB）时，则会将unwritable属性的’WriteBufferWaterMark状态位’置位0。

而本文的主题“流量整形”则是使用了clearUserDefinedWritability、setUserDefinedWritability这对方法来控制unwritable相应的状态位。
当数据write到GlobalTrafficShapingHandler的时候，估计的数据大小大于0，且通过trafficCounter计算出的延迟时间大于最小延迟时间（MINIMAL_WAIT，默认为10ms）时，满足如下任意条件会使得unwritable的’GlobalTrafficShaping状态位’置为1：

- 当perChannel.queueSize（单个Channel中待写出的总数据大小）设定的最大写数据大小时（默认为4M）
- 当queuesSize（所有Channel的待写出的总数据大小）超过设定的最大写数据大小时（默认为400M）
- 对于Channel发送的单个数据包如果太大，以至于计算出的延迟发送时间大于了最大延迟发送时间（maxWriteDelay，默认为4s）时

随着写延迟时间的到达GlobalTrafficShaping中积压的数据不断被写出，当某个Channel中所有待写出的数据都写出后（注意，这里指将数据写到ChannelPipeline中的下一个ChannelOutboundBuffer中）会将unwritable的’GlobalTrafficShaping状态位’置为0。

次

文章目录