BT

Netty系列之Netty可靠性分析

作者 李林锋 发布于 2014年6月18日

1. 背景

1.1. 宕机的代价

1.1.1. 电信行业

毕马威国际(KPMG International)在对46个国家的74家运营商进行调查后发现，全球通信行业每年的收益流失约为400亿美元，占总收入的1%-3%。导致收益流失的因素有多种，主要原因就是计费BUG。

1.1.2. 互联网行业

美国太平洋时间8月16日下午3点50分到3点55分（北京时间8月17日6点50分到6点55分），谷歌遭遇了宕机。根据事后统计，短短的5分钟，谷歌损失了54.5万美元。也就是服务每中断一分钟，损失就达10.8万美元。

2013年，从美国东部时间8月19日下午2点45分开始，有用户率先发现了亚马逊网站出现宕机，大约在20多分钟后又恢复正常。此次宕机让亚马逊每分钟损失近6.7万美元，在宕机期间，消费者无法通过Amazon.com、亚马逊移动端以及Amazon.ca等网站进行购物。

1.2. 软件可靠性

软件可靠性是指在给定时间内，特定环境下软件无错运行的概率。软件可靠性包含了以下三个要素：

1) 规定的时间：软件可靠性只是体现在其运行阶段，所以将运行时间作为规定的时间的度量。运行时间包括软件系统运行后工作与挂起(开启但空闲)的累计时间。由于软件运行的环境与程序路径选取的随机性，软件的失效为随机事件，所以运行时间属于随机变量;

2) 规定的环境条件:环境条件指软件的运行环境。它涉及软件系统运行时所需的各种支持要素，如支持硬件、操作系统、其它支持软件、输入数据格式和范围以及操作规程等。不同的环境条件下软件的可靠性是不同的。具体地说，规定的环境条件主要是描述软件系统运行时计算机的配置情况以及对输入数据的要求，并假定其它一切因素都是理想的。有了明确规定的环境条件，还可以有效判断软件失效的责任在用户方还是提供方;

3) 规定的功能:软件可靠性还与规定的任务和功能有关。由于要完成的任务不同，软件的运行剖面会有所区别，则调用的子模块就不同(即程序路径选择不同)，其可靠性也就可能不同。所以要准确度量软件系统的可靠性必须首先明确它的任务和功能。

1.3. Netty的可靠性

首先，我们要从Netty的主要用途来分析它的可靠性，Netty目前的主流用法有三种：

1) 构建RPC调用的基础通信组件，提供跨节点的远程服务调用能力;

2) NIO通信框架，用于跨节点的数据交换;

3) 其它应用协议栈的基础通信组件，例如HTTP协议以及其它基于Netty开发的应用层协议栈。

以阿里的分布式服务框架Dubbo为例，Netty是Dubbo RPC框架的核心。它的服务调用示例图如下：

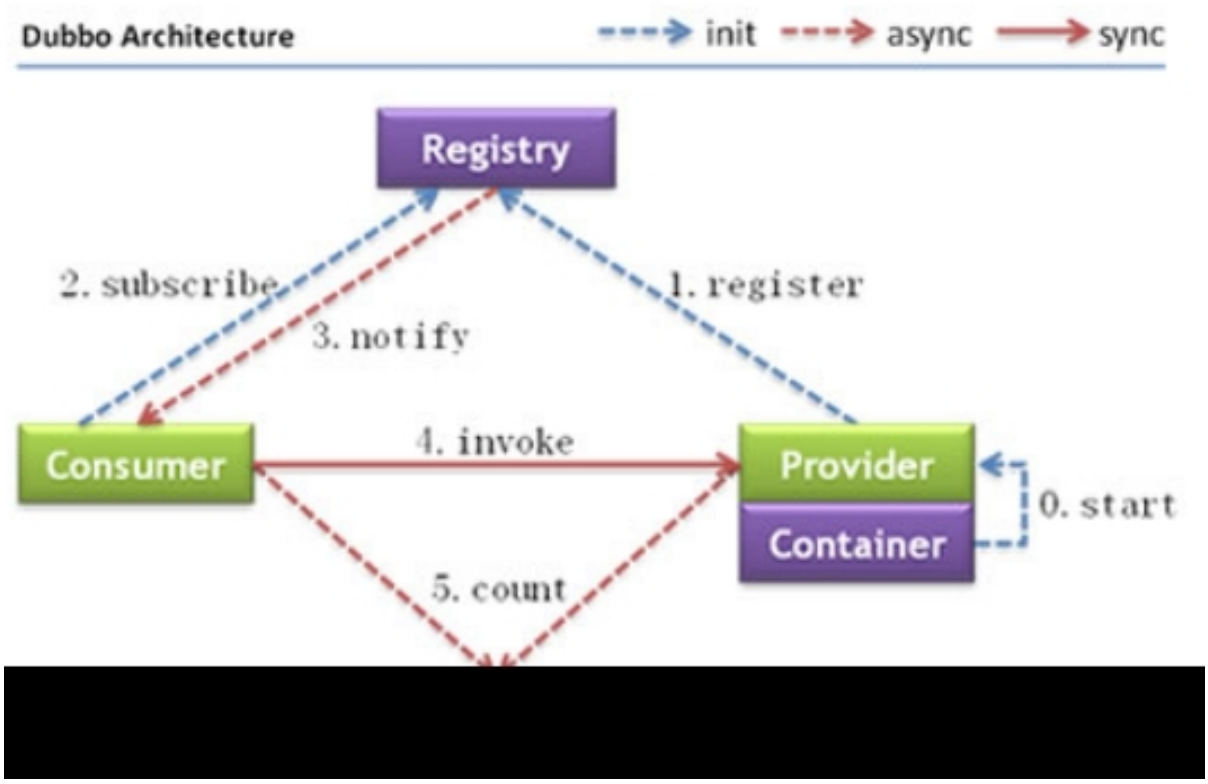

图1-1 Dubbo的节点角色说明图

其中，服务提供者和服务调用者之间可以通过Dubbo协议进行RPC调用，消息的收发默认通过Netty完成。

通过对Netty主流应用场景的分析，我们发现Netty面临的可靠性问题大致分为三类：

1）传统的网络I/O故障，例如网络闪断、防火墙Hang住连接、网络超时等；

2）NIO特有的故障，例如NIO类库特有的BUG、读写半包处理异常、Reactor线程跑飞等等；

3）编解码相关的异常。

在大多数的业务应用场景中，一旦因为某些故障导致Netty不能正常工作，业务往往会陷入瘫痪。所以，从业务诉求来看，对Netty框架的可靠性要求是非常的高。作为当前业界最流行的一款NIO框架，Netty在不同行业和领域都得到了广泛的应用，它的高可靠性已经得到了成百上千的生产系统检验。

Netty是如何支持系统高可靠性的？下面，我们就从几个不同维度出发一探究竟。

2. Netty高可靠性之道

2.1. 网络通信类故障

2.1.1. 客户端连接超时

在传统的同步阻塞编程模式下，客户端Socket发起网络连接，往往需要指定连接超时时间，这样做的目的主要有两个：

1）在同步阻塞I/O模型中，连接操作是同步阻塞的，如果不设置超时时间，客户端I/O线程可能会被长时间阻塞，这会导致系统可用I/O线程数的减少；

2）业务层需要：大多数系统都会对业务流程执行时间有限制，例如WEB交互类的响应时间要小于3S。客户端设置连接超时时间是为了实现业务层的超时。

JDK原生的Socket连接接口定义如下：

```
/**
 * Connects this socket to the server with a specified timeout value.
 * A timeout of zero is interpreted as an infinite timeout. The connection
 * will then block until established or an error occurs.
 *
 * @param   endpoint the <code>SocketAddress</code>
 * @param   timeout  the timeout value to be used in milliseconds.
 * @throws  IOException if an error occurs during the connection
 * @throws  SocketTimeoutException if timeout expires before connecting
 * @throws  java.nio.channels.IllegalBlockingModeException
 *          if this socket has an associated channel,
 *          and the channel is in non-blocking mode
 * @throws  IllegalArgumentException if endpoint is null or is a
 *          SocketAddress subclass not supported by this socket
 * @since 1.4
 * @spec JSR-51
 */
public void connect(SocketAddress endpoint, int timeout) throws IOException {
```

图2-1 JDK Socket连接超时接口

对于NIO的SocketChannel，在非阻塞模式下，它会直接返回连接结果，如果没有连接成功，也没有发生IO异常，则需要将SocketChannel注册到Selector上监听连接结果。所以，异步连接的超时无法在API层面直接设置，而是需要通过定时器来主动监测。

下面我们首先看下JDK NIO类库的SocketChannel连接接口定义：

```
     * @throws  AlreadyConnectedException
     *          If this channel is already connected
     *
     * @throws  ConnectionPendingException
     *          If a non-blocking connection operation is already in progress
     *          on this channel
     *
     * @throws  ClosedChannelException
     *          If this channel is closed
     *
     * @throws  AsynchronousCloseException
     *          If another thread closes this channel
     *          while the connect operation is in progress
     *
     * @throws  ClosedByInterruptException
     *          If another thread interrupts the current thread
     *          while the connect operation is in progress, thereby
     *          closing the channel and setting the current thread's
     *          interrupt status
     *
     * @throws  UnresolvedAddressException
     *          If the given remote address is not fully resolved
     *
     * @throws  UnsupportedAddressTypeException
     *          If the type of the given remote address is not supported
     *
     * @throws  SecurityException
     *          If a security manager has been installed
     *          and it does not permit access to the given remote endpoint
     *
     * @throws  IOException
     *          If some other I/O error occurs
     */
    public abstract boolean connect(SocketAddress remote) throws IOException;
```

图2-2 JDK NIO 类库SocketChannel连接接口

从上面的接口定义可以看出，NIO类库并没有现成的连接超时接口供用户直接使用，如果要在NIO编程中支持连接超时，往往需要NIO框架或者用户自己封装实现。

下面我们看下Netty是如何支持连接超时的，首先，在创建NIO客户端的时候，可以配置连接超时参数：

```
// Configure the client.
EventLoopGroup group = new NioEventLoopGroup();
try {
    Bootstrap b = new Bootstrap();
    b.group(group)
     .channel(NioSocketChannel.class)
     .option(ChannelOption.TCP_NODELAY, true)
     .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 3000)
     .handler(new ChannelInitializer<SocketChannel>() {
```

图2-3 Netty客户端创建支持设置连接超时参数

设置完连接超时之后，Netty在发起连接的时候，会根据超时时间创建ScheduledFuture挂载在Reactor线程上，用于定时监测是否发生连接超时，相关代码如下：

```
// Schedule connect timeout.
int connectTimeoutMillis = config().getConnectTimeoutMillis();
if (connectTimeoutMillis > 0) {
    connectTimeoutFuture = eventLoop().schedule(new Runnable() {
        @Override
        public void run() {
            ChannelPromise connectPromise = AbstractNioChannel.this.connectPromise;
            ConnectTimeoutException cause =
                    new ConnectTimeoutException("connection timed out: " + remoteAddress);
            if (connectPromise != null && connectPromise.tryFailure(cause)) {
                close(voidPromise());
            }
        }
    }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
}
```

图2-4 根据连接超时创建超时监测定时任务

创建连接超时定时任务之后，会由NioEventLoop负责执行。如果已经连接超时，但是服务端仍然没有返回TCP握手应答，则关闭连接，代码如上图所示。

如果在超时期限内处理完成连接操作，则取消连接超时定时任务，相关代码如下：

```
@Override
public void finishConnect() {
    // Note this method is invoked by the event loop only if the connect
    // neither cancelled nor timed out.

    assert eventLoop().inEventLoop();
    assert connectPromise != null;

    try {
        boolean wasActive = isActive();
        doFinishConnect();
        fulfillConnectPromise(connectPromise, wasActive);
    } catch (Throwable t) {
        if (t instanceof ConnectException) {
            Throwable newT = new ConnectException(t.getMessage() + ": "
            newT.setStackTrace(t.getStackTrace());
            t = newT;
        }

        // Use tryFailure() instead of setFailure() to avoid the race ag
        connectPromise.tryFailure(t);
        closeIfClosed();
    } finally {
        // Check for null as the connectTimeoutFuture is only created if
        // See https://github.com/netty/netty/issues/1770
        if (connectTimeoutFuture != null) {
            connectTimeoutFuture.cancel(false);
        }
```

图2-5 取消连接超时定时任务

Netty的客户端连接超时参数与其它常用的TCP参数一起配置，使用起来非常方便，上层用户不用关心底层的超时实现机制。这既满足了用户的个性化需求，又实现了故障的分层隔离。

2.1.2. 通信对端强制关闭连接

在客户端和服务端正常通信过程中，如果发生网络闪断、对方进程突然宕机或者其它非正常关闭链路事件时，TCP链路就会发生异常。由于TCP是全双工的，通信双方都需要关闭和释放Socket句柄才不会发生句柄的泄漏。

在实际的NIO编程过程中，我们经常会发现由于句柄没有被及时关闭导致的功能和可靠性问题。究其原因总结如下：

1）IO的读写等操作并非仅仅集中在Reactor线程内部，用户上层的一些定制行为可能会导致IO操作的外逸，例如业务自定义心跳机制。这些定制行为加大了统一异常处理的难度，IO操作越发散，故障发生的概率就越大；

2）一些异常分支没有考虑到，由于外部环境诱因导致程序进入这些分支，就会引起故障。

下面我们通过故障模拟，看Netty是如何处理对端链路强制关闭异常的。首先启动Netty服务端和客户端，TCP链路建立成功之后，双方维持该链路，查看链路状态，结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano|find "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       5032
  TCP    127.0.0.1:3410         127.0.0.1:8080         ESTABLISHED     3848
  TCP    127.0.0.1:8080         127.0.0.1:3410         ESTABLISHED     5032
```

图2-6 Netty服务端和客户端TCP链路状态正常

强制关闭客户端，模拟客户端宕机，服务端控制台打印如下异常：

```
java.io.IOException: 远程主机强迫关闭了一个现有的连接。
        at sun.nio.ch.SocketDispatcher.read0(Native Method)
        at sun.nio.ch.SocketDispatcher.read(SocketDispatcher.java:43)
        at sun.nio.ch.IOUtil.readIntoNativeBuffer(IOUtil.java:223)
        at sun.nio.ch.IOUtil.read(IOUtil.java:192)
        at sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:379)
        at io.netty.buffer.UnpooledUnsafeDirectByteBuf.setBytes(UnpooledUnsafeDirectByteBuf.java:446)
        at io.netty.buffer.AbstractByteBuf.writeBytes(AbstractByteBuf.java:871)
        at io.netty.channel.socket.nio.NioSocketChannel.doReadBytes(NioSocketChannel.java:208)
        at io.netty.channel.nio.AbstractNioByteChannel$NioByteUnsafe.read(AbstractNioByteChannel.java:119)
        at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:485)
        at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:452)
        at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:346)
        at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:794)
        at java.lang.Thread.run(Thread.java:744)
```

图2-7 模拟TCP链路故障

从堆栈信息可以判断，服务端已经监控到客户端强制关闭了连接，下面我们看下服务端是否已经释放了连接句柄，再次执行netstat命令，执行结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano|find "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       4112
```

图2-8 查看故障链路状态

从执行结果可以看出，服务端已经关闭了和客户端的TCP连接，句柄资源正常释放。由此可以得出结论，Netty底层已经自动对该故障进行了处理。

下面我们一起看下Netty是如何感知到链路关闭异常并进行正确处理的，查看AbstractByteBuf的writeBytes方法，它负责将指定Channel的缓冲区数据写入到ByteBuf中，详细代码如下：

```
@Override
public int writeBytes(ScatteringByteChannel in, int length) throws IOException {
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        writerIndex += writtenBytes;
    }
    return writtenBytes;
}
```

图2-9 AbstractByteBuf的writeBytes方法

在调用SocketChannel的read方法时发生了IOException，代码如下：

```
@Override
public int setBytes(int index, ScatteringByteChannel in, int length) throws IOException {
    ensureAccessible();
    ByteBuffer tmpBuf = internalNioBuffer();
    tmpBuf.clear().position(index).limit(index + length);
    try {
        return in.read(tmpBuf);
    } catch (ClosedChannelException e) {
        return -1;
    }
}
```

图2-10 读取缓冲区数据发生IO异常

为了保证IO异常被统一处理，该异常向上抛，由AbstractNioByteChannel进行统一异常处理，代码如下：

```
private void closeOnRead(ChannelPipeline pipeline) {
    SelectionKey key = selectionKey();
    setInputShutdown();
    if (isOpen()) {
        if (Boolean.TRUE.equals(config().getOption(ChannelOption.ALLOW_HALF_CLOSURE))) {
            key.interestOps(key.interestOps() & ~readInterestOp);
            pipeline.fireUserEventTriggered(ChannelInputShutdownEvent.INSTANCE);
        } else {
            close(voidPromise());
        }
```

图2-11 链路异常退出异常处理

为了能够对异常策略进行统一，也为了方便维护，防止处理不当导致的句柄泄漏等问题，句柄的关闭，统一调用AbstractChannel的close方法，代码如下：

```
@Override
public ChannelFuture close(ChannelPromise promise) {
    return pipeline.close(promise);
}
```

图2-12 统一的Socket句柄关闭接口

2.1.3. 正常的连接关闭

对于短连接协议，例如HTTP协议，通信双方数据交互完成之后，通常按照双方的约定由服务端关闭连接，客户端获得TCP连接关闭请求之后，关闭自身的Socket连接，双方正式断开连接。

在实际的NIO编程过程中，经常存在一种误区：认为只要是对方关闭连接，就会发生IO异常，捕获IO异常之后再关闭连接即可。实际上，连接的合法关闭不会发生IO异常，它是一种正常场景，如果遗漏了该场景的判断和处理就会导致连接句柄泄漏。

下面我们一起模拟故障，看Netty是如何处理的。测试场景设计如下：改造下Netty客户端，双发链路建立成功之后，等待120S，客户端正常关闭链路。看服务端是否能够感知并释放句柄资源。

首先启动Netty客户端和服务端,双方TCP链路连接正常：

```
C:\Documents and Settings\Administrator>netstat -ano|find "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       5032
  TCP    127.0.0.1:3410         127.0.0.1:8080         ESTABLISHED     3848
  TCP    127.0.0.1:8080         127.0.0.1:3410         ESTABLISHED     5032
```

图2-13 TCP连接状态正常

120S之后，客户端关闭连接，进程退出，为了能够看到整个处理过程，我们在服务端的Reactor线程处设置断点，先不做处理，此时链路状态如下：

```
C:\Documents and Settings\Administrator>netstat -ano|find "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       3080
  TCP    127.0.0.1:8080         127.0.0.1:3870         CLOSE_WAIT      3080
```

图2-14 TCP连接句柄等待释放

从上图可以看出，此时服务端并没有关闭Socket连接，链路处于CLOSE_WAIT状态，放开代码让服务端执行完，结果如下：

```
C:\Documents and Settings\Administrator>netstat -ano|find "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       3080
```

图2-15 TCP连接句柄正常释放

下面我们一起看下服务端是如何判断出客户端关闭连接的，当连接被对方合法关闭后，被关闭的SocketChannel会处于就绪状态，SocketChannel的read操作返回值为-1，说明连接已经被关闭，代码如下：

```
byteBuf = allocator.ioBuffer(byteBufCapacity);
int writable = byteBuf.writableBytes();
int localReadAmount = doReadBytes(byteBuf);
if (localReadAmount <= 0) {
    // not was read release the buffer
    byteBuf.release();
    close = localReadAmount < 0;
    break;
}
```

图2-16 需要对读取的字节数进行判断

如果SocketChannel被设置为非阻塞，则它的read操作可能返回三个值：

1）大于0，表示读取到了字节数；

2）等于0，没有读取到消息，可能TCP处于Keep-Alive状态，接收到的是TCP握手消息；

3）-1，连接已经被对方合法关闭。

通过调试，我们发现，NIO类库的返回值确实为-1：

```
@Override
public int writeBytes(ScatteringByteChannel in, int length)
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        writtenBytes= -1
    }
    retu
}

@Overrid-1
```

图2-17 链路正常关闭，返回值为-1

得知连接关闭之后，Netty将关闭操作位设置为true,关闭句柄，代码如下：

```
if (close) {
    closeOnRead(pipeline);
    close = false;
}
```

图2-18 连接正常关闭，释放资源

2.1.4. 故障定制

在大多数场景下，当底层网络发生故障的时候，应该由底层的NIO框架负责释放资源，处理异常等。上层的业务应用不需要关心底层的处理细节。但是，在一些特殊的场景下，用户可能需要感知这些异常，并针对这些异常进行定制处理，例如：

1）客户端的断连重连机制；

2）消息的缓存重发；

3）接口日志中详细记录故障细节；

4）运维相关功能，例如告警、触发邮件/短信等

Netty的处理策略是发生IO异常，底层的资源由它负责释放，同时将异常堆栈信息以事件的形式通知给上层用户，由用户对异常进行定制。这种处理机制既保证了异常处理的安全性，也向上层提供了灵活的定制能力。

具体接口定义以及默认实现如下：

```
/**
 * Calls {@link ChannelHandlerContext#fireExceptionCaught(Throwable)} to forward
 * to the next {@link ChannelHandler} in the {@link ChannelPipeline}.
 *
 * Sub-classes may override this method to change behavior.
 */
@Skip
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    ctx.fireExceptionCaught(cause);
}
```

图2-19 故障定制接口

用户可以覆盖该接口，进行个性化的异常定制。例如发起重连等。

2.2. 链路的有效性检测

当网络发生单通、连接被防火墙Hang住、长时间GC或者通信线程发生非预期异常时，会导致链路不可用且不易被及时发现。特别是异常发生在凌晨业务低谷期间，当早晨业务高峰期到来时，由于链路不可用会导致瞬间的大批量业务失败或者超时，这将对系统的可靠性产生重大的威胁。

从技术层面看，要解决链路的可靠性问题，必须周期性的对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。

心跳检测机制分为三个层面：

1）TCP层面的心跳检测，即TCP的Keep-Alive机制，它的作用域是整个TCP协议栈；

2）协议层的心跳检测，主要存在于长连接协议中。例如SMPP协议；

3）应用层的心跳检测，它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路可用，对方活着并且能够正常接收和发送消息。

做为高可靠的NIO框架，Netty也提供了心跳检测机制，下面我们一起熟悉下心跳的检测原理。

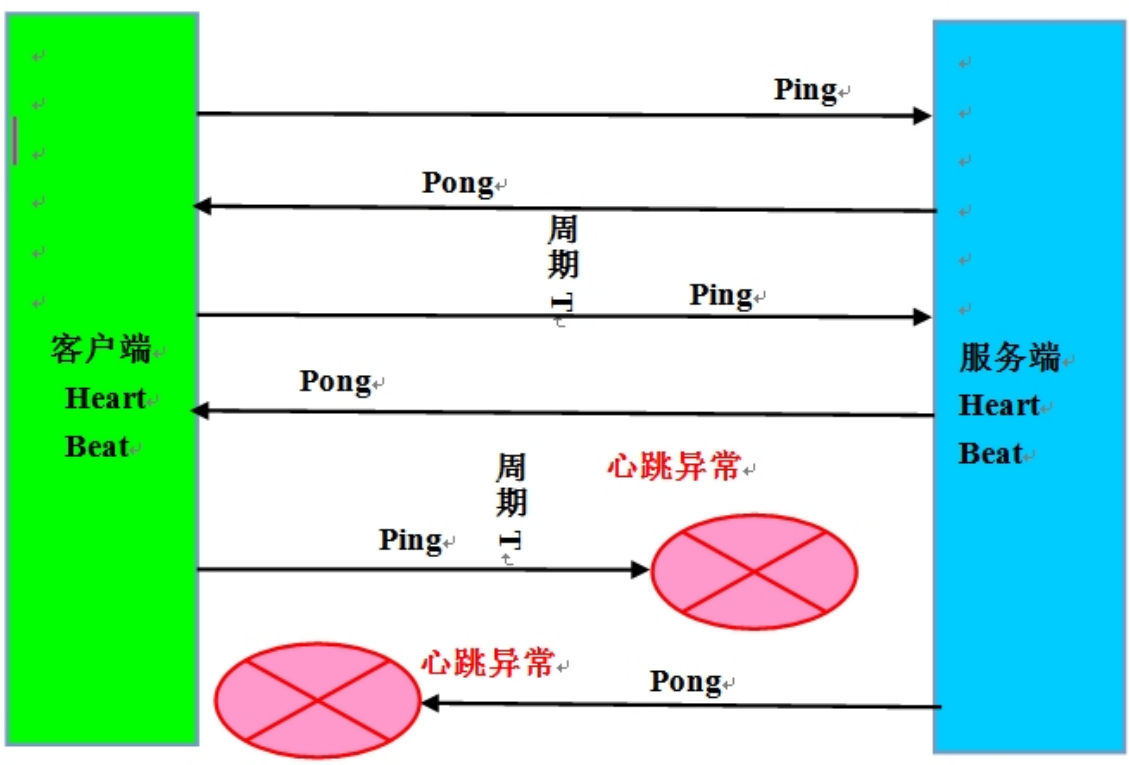

图2-20 心跳检测机制

不同的协议，心跳检测机制也存在差异，归纳起来主要分为两类：

1）Ping-Pong型心跳：由通信一方定时发送Ping消息，对方接收到Ping消息之后，立即返回Pong应答消息给对

方，属于请求-响应型心跳；

2）Ping-Ping型心跳：不区分心跳请求和应答，由通信双方按照约定定时向对方发送心跳Ping消息，它属于双向心跳。

心跳检测策略如下：

1）连续N次心跳检测都没有收到对方的Pong应答消息或者Ping请求消息，则认为链路已经发生逻辑失效，这被称作心跳超时；

2）读取和发送心跳消息的时候如何直接发生了IO异常，说明链路已经失效，这被称为心跳失败。

无论发生心跳超时还是心跳失败，都需要关闭链路，由客户端发起重连操作，保证链路能够恢复正常。

Netty的心跳检测实际上是利用了链路空闲检测机制实现的，相关代码如下：

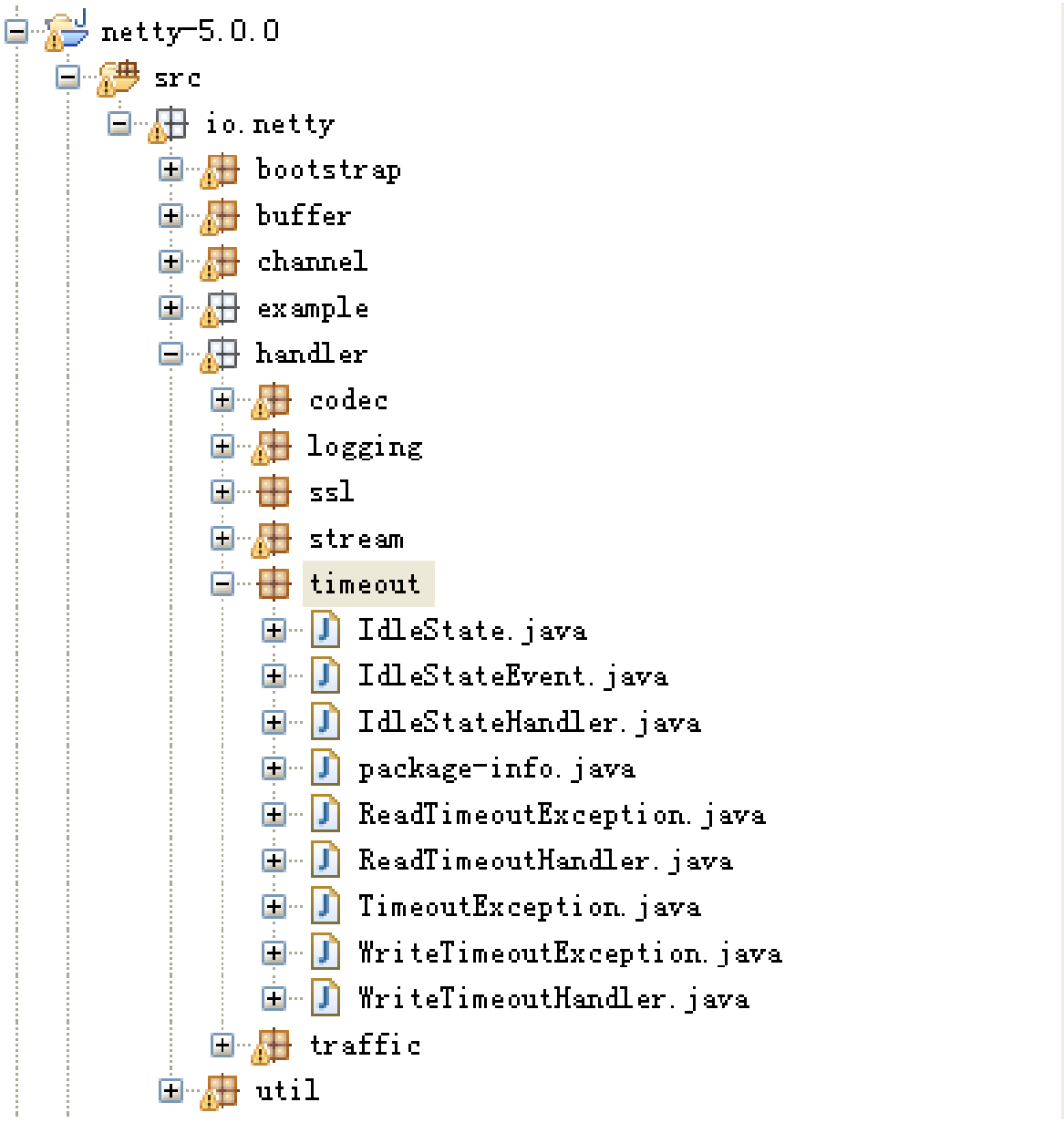

图2-21 心跳检测的代码包路径

Netty提供的空闲检测机制分为三种：

1）读空闲，链路持续时间t没有读取到任何消息；

2）写空闲，链路持续时间t没有发送任何消息；

3）读写空闲，链路持续时间t没有接收或者发送任何消息。

Netty的默认读写空闲机制是发生超时异常，关闭连接，但是，我们可以定制它的超时实现机制，以便支持不同的用户场景。

WriteTimeoutHandler的超时接口如下：

```
/**
 * Is called when a write timeout was detected
 */
protected void writeTimedOut(ChannelHandlerContext ctx) throws Exception {
    if (!closed) {
        ctx.fireExceptionCaught(WriteTimeoutException.INSTANCE);
        ctx.close();
        closed = true;
    }
}
```

图2-22 写超时

ReadTimeoutHandler的超时接口如下：

```
/**
 * Is called when a read timeout was detected.
 */
protected void readTimedOut(ChannelHandlerContext ctx) throws Exception {
    if (!closed) {
        ctx.fireExceptionCaught(ReadTimeoutException.INSTANCE);
        ctx.close();
        closed = true;
    }
}
```

图2-23 读超时

读写空闲的接口如下：

```
/**
 * Is called when an {@link IdleStateEvent} should be fired. This implementation calls
 * {@link ChannelHandlerContext#fireUserEventTriggered(Object)}.
 */
protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) throws Exception {
    ctx.fireUserEventTriggered(evt);
}
```

图2-24 读写空闲

利用Netty提供的链路空闲检测机制，可以非常灵活的实现协议层的心跳检测。在《Netty权威指南》中的私有协议栈设计和开发章节，我利用Netty提供的自定义Task接口实现了另一种心跳检测机制，感兴趣的朋友可以参阅该书。

2.3. Reactor线程的保护

Reactor线程是IO操作的核心，NIO框架的发动机，一旦出现故障，将会导致挂载在其上面的多路用复用器和多个链路无法正常工作。因此它的可靠性要求非常高。

笔者就曾经遇到过因为异常处理不当导致Reactor线程跑飞，大量业务请求处理失败的故障。下面我们一起看下Netty是如何有效提升Reactor线程的可靠性的。

2.3.1. 异常处理要当心

尽管Reactor线程主要处理IO操作，发生的异常通常是IO异常，但是，实际上在一些特殊场景下会发生非IO异常，如果仅仅捕获IO异常可能就会导致Reactor线程跑飞。为了防止发生这种意外，在循环体内一定要捕获Throwable，而不是IO异常或者Exception。

Netty的相关代码如下：

```
        processSelectedKeysOptimized(selectedKeys.flip());
    } else {
        processSelectedKeysPlain(selector.selectedKeys());
    }
    final long ioTime = System.nanoTime() - ioStartTime;

    final int ioRatio = this.ioRatio;
    runAllTasks(ioTime * (100 - ioRatio) / ioRatio);

    if (isShuttingDown()) {
        closeAll();
        if (confirmShutdown()) {
            break;
        }
    }
} catch (Throwable t) {
    logger.warn("Unexpected exception in the selector loop.", t);

    // Prevent possible consecutive immediate failures that lead to
    // excessive CPU consumption.
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // Ignore.
    }
```

图2-25 Reactor线程异常保护

捕获Throwable之后，即便发生了意外未知对异常，线程也不会跑飞，它休眠1S，防止死循环导致的异常绕接，然后继续恢复执行。这样处理的核心理念就是：

1）某个消息的异常不应该导致整条链路不可用；

2）某条链路不可用不应该导致其它链路不可用；

3）某个进程不可用不应该导致其它集群节点不可用。

2.3.2. 死循环保护

通常情况下，死循环是可检测、可预防但是无法完全避免的。Reactor线程通常处理的都是IO相关的操作，因此我们重点关注IO层面的死循环。

JDK NIO类库最著名的就是 epoll bug了，它会导致Selector空轮询，IO线程CPU 100%，严重影响系统的安全性和可靠性。

SUN在JKD1.6 update18版本声称解决了该BUG，但是根据业界的测试和大家的反馈，直到JDK1.7的早期版本，该BUG依然存在，并没有完全被修复。发生该BUG的主机资源占用图如下：

```
Processes: 168 total, 5 running, 6 stuck, 157 sleeping, 1215 threads
Load Avg: 6.79, 6.38, 6.13  CPU usage: 30.86% user, 13.37% sys, 55.76% idle  SharedLibs: 16
MemRegions: 92254 total, 4194M resident, 66M private, 761M shared. PhysMem: 1681M wired, 33
VM: 421G vsize, 1026M framework vsize, 13779046(0) pageins, 5483724(0) pageouts. Networks:
Disks: 10621319/215G read, 10691231/398G written.

PID    COMMAND      %CPU      TIME     #TH   #WQ  #PORT #MREGS RPRVT  RSHRD  RSIZE  VPRVT
58552  java         102.2     02:28.72 17/1  1    77    213    11M    9528K  23M    107M
0      kernel_task  19.2      11:14:21 100/5 0    2     1181-  123M-  0B     834M-  132M-
58569  top          17.3      00:11.65 1/1   0    34    32     2356K  216K   3104K  18M
```

图2-26 epoll bug CPU空轮询

SUN在解决该BUG的问题上不给力，只能从NIO框架层面进行问题规避，下面我们看下Netty是如何解决该问题的。

Netty的解决策略：

1）根据该BUG的特征，首先侦测该BUG是否发生；

2）将问题Selector上注册的Channel转移到新建的Selector上；

3）老的问题Selector关闭，使用新建的Selector替换。

下面具体看下代码，首先检测是否发生了该BUG：

```
try {
    int selectCnt = 0;
    long currentTimeNanos = System.nanoTime();
    long selectDeadLineNanos = currentTimeNanos + delayNanos(currentTimeNanos);
    for (;;) {
        long timeoutMillis = (selectDeadLineNanos - currentTimeNanos + 500000L) / 1000000L;
        if (timeoutMillis <= 0) {
            if (selectCnt == 0) {
                selector.selectNow();
                selectCnt = 1;
            }
            break;
        }

        int selectedKeys = selector.select(timeoutMillis);
        selectCnt ++;
```

图2-27 epoll bug 检测

一旦检测发生该BUG，则重建Selector，代码如下：

```
try {
    for (SelectionKey key: oldSelector.keys()) {
        Object a = key.attachment();
        try {
            if (key.channel().keyFor(newSelector) != null) {
                continue;
            }

            int interestOps = key.interestOps();
            key.cancel();
            key.channel().register(newSelector, interestOps, a);
            nChannels ++;
        } catch (Exception e) {
            logger.warn("Failed to re-register a Channel to the new Selector.", e);
            if (a instanceof AbstractNioChannel) {
                AbstractNioChannel ch = (AbstractNioChannel) a;
                ch.unsafe().close(ch.unsafe().voidPromise());
            } else {
                @SuppressWarnings("unchecked")
                NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
                invokeChannelUnregistered(task, key, e);
            }
        }
```

图2-28 重建Selector

重建完成之后，替换老的Selector，代码如下：

```
if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
        selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
    // The selector returned prematurely many times in a row.
    // Rebuild the selector to work around the problem.
    logger.warn(
            "Selector.select() returned prematurely {} times in a row; rebuilding selector.",
            selectCnt);

    rebuildSelector();
    selector = this.selector;

    // Select again to populate selectedKeys.
    selector.selectNow();
    selectCnt = 1;
    break;
}
```

图2-29 替换Selector

大量生产系统的运行表明，Netty的规避策略可以解决epoll bug 导致的IO线程CPU死循环问题。

2.4. 优雅退出

Java的优雅停机通常通过注册JDK的ShutdownHook来实现，当系统接收到退出指令后，首先标记系统处于退出状态，不再接收新的消息，然后将积压的消息处理完，最后调用资源回收接口将资源销毁，最后各线程退出执行。

通常优雅退出有个时间限制，例如30S，如果到达执行时间仍然没有完成退出前的操作，则由监控脚本直接kill -9 pid，强制退出。

Netty的优雅退出功能随着版本的优化和演进也在不断的增强，下面我们一起看下Netty5的优雅退出。

首先看下Reactor线程和线程组，它们提供了优雅退出接口。EventExecutorGroup的接口定义如下：

```
/**
 * Signals this executor that the caller wants the executor to be shut down.
 * {@link #isShuttingDown()} starts to return {@code true}, and the executor p
 * Unlike {@link #shutdown()}, graceful shutdown ensures that no tasks are sub
 * (usually a couple seconds) before it shuts itself down.  If a task is subm
 * it is guaranteed to be accepted and the quiet period will start over.
 *
 * @param quietPeriod the quiet period as described in the documentation
 * @param timeout     the maximum amount of time to wait until the executor is
 *                    regardless if a task was submitted during the quiet peri
 * @param unit        the unit of {@code quietPeriod} and {@code timeout}
 *
 * @return the {@link #terminationFuture()}
 */
Future<?> shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit);
```

图2-30 EventExecutorGroup优雅退出

NioEventLoop的资源释放接口实现：

```
    @Override
    protected void cleanup() {
        try {
            selector.close();
        } catch (IOException e) {
            logger.warn("Failed to close a selector.", e);
        }
    }
```

图2-31 NioEventLoop资源释放

ChannelPipeline的关闭接口：

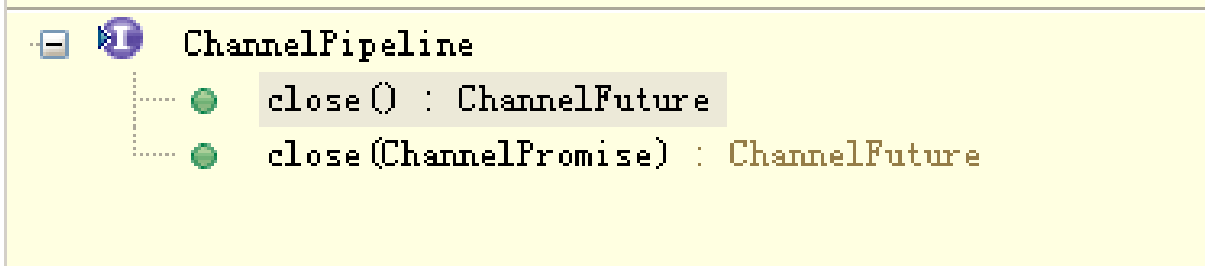

图2-32 ChannelPipeline关闭接口

目前Netty向用户提供的主要接口和类库都提供了资源销毁和优雅退出的接口，用户的自定义实现类可以继承这些接口，完成用户资源的释放和优雅退出。

2.5. 内存保护

2.5.1. 缓冲区的内存泄漏保护

为了提升内存的利用率，Netty提供了内存池和对象池。但是，基于缓存池实现以后需要对内存的申请和释放进行严格的管理，否则很容易导致内存泄漏。

如果不采用内存池技术实现，每次对象都是以方法的局部变量形式被创建，使用完成之后，只要不再继续引用它，JVM会自动释放。但是，一旦引入内存池机制，对象的生命周期将由内存池负责管理，这通常是个全局引用，如果不显式释放JVM是不会回收这部分内存的。

对于Netty的用户而言，使用者的技术水平差异很大，一些对JVM内存模型和内存泄漏机制不了解的用户，可能只记得申请内存，忘记主动释放内存，特别是JAVA程序员。

为了防止因为用户遗漏导致内存泄漏，Netty在Pipe line的尾Handler中自动对内存进行释放，相关代码如下：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    try {
        logger.debug(
                "Discarded inbound message {} that reached at the tail of the pipeline. " +
                        "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

图2-33 TailHandler的内存回收操作

对于内存池，实际就是将缓冲区重新放到内存池中循环使用，代码如下：

```
@Override
protected final void deallocate() {
    if (handle >= 0) {
        final long handle = this.handle;
        this.handle = -1;
        memory = null;
        chunk.arena.free(chunk, handle);
        recycle();
    }
}
```

图2-34 PooledByteBuf的内存回收操作

2.5.2. 缓冲区内存溢出保护

做过协议栈的读者都知道，当我们对消息进行解码的时候，需要创建缓冲区。缓冲区的创建方式通常有两种：

1）容量预分配，在实际读写过程中如果不够再扩展；

2）根据协议消息长度创建缓冲区。

在实际的商用环境中，如果遇到畸形码流攻击、协议消息编码异常、消息丢包等问题时，可能会解析到一个超长的长度字段。笔者曾经遇到过类似问题，报文长度字段值竟然是2G多，由于代码的一个分支没有对长度上限做有效保护，结果导致内存溢出。系统重启后几秒内再次内存溢出，幸好及时定位出问题根因，险些酿成严重的事故。

Netty提供了编解码框架，因此对于解码缓冲区的上限保护就显得非常重要。下面，我们看下Netty是如何对缓冲区进行上限保护的：

首先，在内存分配的时候指定缓冲区长度上限：

```
/**
 * Allocate a {@link ByteBuf} with the given initial capacity and the given
 * maximal capacity. If it is a direct or heap buffer depends on the actual
 * implementation.
 */
ByteBuf buffer(int initialCapacity, int maxCapacity);
```

图2-35 缓冲区分配器可以指定缓冲区最大长度

其次，在对缓冲区进行写入操作的时候，如果缓冲区容量不足需要扩展，首先对最大容量进行判断，如果扩展后的容量超过上限，则拒绝扩展：

```
if (minWritableBytes > maxCapacity - writerIndex) {
    throw new IndexOutOfBoundsException(String.format(
            "writerIndex(%d) + minWritableBytes(%d) exceeds maxCapacity(%d): %s",
            writerIndex, minWritableBytes, maxCapacity, this));
}
```

图2-35 缓冲区扩展上限保护

最后，在解码的时候，对消息长度进行判断，如果超过最大容量上限，则抛出解码异常，拒绝分配内存：

```
if (frameLength > maxFrameLength) {
    long discard = frameLength - in.readableBytes();
    tooLongFrameLength = frameLength;

    if (discard < 0) {
        // buffer contains more bytes then the frameLength so we can discard all no
        in.skipBytes((int) frameLength);
    } else {
        // Enter the discard mode and discard everything received so far.
        discardingTooLongFrame = true;
        bytesToDiscard = discard;
        in.skipBytes(in.readableBytes());
    }
    failIfNecessary(true);
    return null;
}
```

图2-36 超出容量上限的半包解码，失败

```
private void fail(long frameLength) {
    if (frameLength > 0) {
        throw new TooLongFrameException(
                        "Adjusted frame length exceeds " + maxFrameLength +
                        ": " + frameLength + " - discarded");
    } else {
        throw new TooLongFrameException(
                        "Adjusted frame length exceeds " + maxFrameLength +
                        " - discarding");
    }
}
```

图2-37 抛出TooLongFrameException异常

2.6. 流量整形

大多数的商用系统都有多个网元或者部件组成，例如参与短信互动，会涉及到手机、基站、短信中心、短信网关、SP/CP等网元。不同网元或者部件的处理性能不同。为了防止因为浪涌业务或者下游网元性能低导致下游网元被压垮，有时候需要系统提供流量整形功能。

下面我们一起看下流量整形(traffic shaping)的定义：流量整形（Traffic Shaping）是一种主动调整流量输出速率的措施。一个典型应用是基于下游网络结点的TP指标来控制本地流量的输出。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）。当令牌桶有足够的令牌时，再均匀的向外发送这些被缓存的报文。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。

流量整形的原理示意图如下：

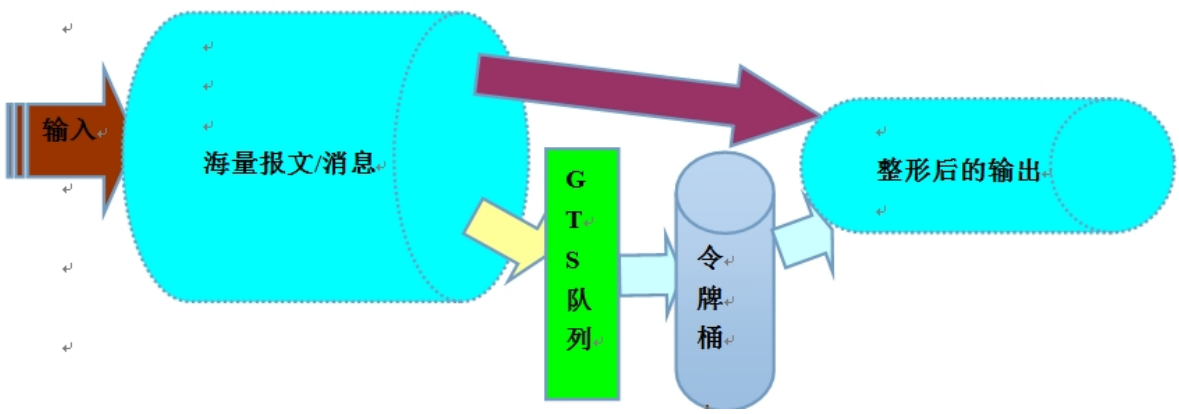

图2-38 流量整形原理图

作为高性能的NIO框架，Netty的流量整形有两个作用：

1）防止由于上下游网元性能不均衡导致下游网元被压垮，业务流程中断；

2）防止由于通信模块接收消息过快，后端业务线程处理不及时导致的“撑死”问题。

下面我们就具体学习下Netty的流量整形功能。

2.6.1. 全局流量整形

全局流量整形的作用范围是进程级的，无论你创建了多少个Channel，它的作用域针对所有的Channel。

用户可以通过参数设置：报文的接收速率、报文的发送速率、整形周期。相关的接口如下所示：

```
/**
 * Create a new instance
 *
 * @param executor
 *          the {@link ScheduledExecutorService} to use for the {@link TrafficCounter}
 * @param writeLimit
 *          0 or a limit in bytes/s
 * @param readLimit
 *          0 or a limit in bytes/s
 * @param checkInterval
 *          The delay between two computations of performances for
 *            channels or 0 if no stats are to be computed
 */
public GlobalTrafficShapingHandler(ScheduledExecutorService executor, long writeLimit,
        long readLimit, long checkInterval) {
    super(writeLimit, readLimit, checkInterval);
    createGlobalTrafficCounter(executor);
}
```

图2-39 全局流量整形参数设置

Netty流量整形的原理是：对每次读取到的ByteBuf可写字节数进行计算，获取当前的报文流量，然后与流量整形阈值对比。如果已经达到或者超过了阈值。则计算等待时间delay，将当前的ByteBuf放到定时任务Task中缓存，由定时任务线程池在延迟delay之后继续处理该ByteBuf。相关代码如下：

```
@Override
public void channelRead(final ChannelHandlerContext ctx, final Object msg) throws Exception {
    long size = calculateSize(msg);
    long curtime = System.currentTimeMillis();

    if (trafficCounter != null) {
        trafficCounter.bytesRecvFlowControl(size);
        if (readLimit == 0) {
            // no action
            ctx.fireChannelRead(msg);

            return;
        }
```

图2-40 动态计算当前流量

如果达到整形阈值，则对新接收的ByteBuf进行缓存，放入线程池的消息队列中，稍后处理，代码如下：

```
                TimeUnit.MILLISECONDS);
} else {
    // Create a Runnable to update the next handler in the chain. If one was
    // just be reused to limit object creation
    Runnable bufferUpdateTask = new Runnable() {
        @Override
        public void run() {
            ctx.fireChannelRead(msg);
        }
    };
    ctx.executor().schedule(bufferUpdateTask, wait, TimeUnit.MILLISECONDS);
    return;
}
```

图2-41 缓存当前的ByteBuf

定时任务的延时时间根据检测周期T和流量整形阈值计算得来，代码如下：

```
 */
private static long getTimeToWait(long limit, long bytes, long lastTime, long curtime) {
    long interval = curtime - lastTime;
    if (interval <= 0) {
        // Time is too short, so just lets continue
        return 0;
    }
    return (bytes * 1000 / limit - interval) / 10 * 10;
}
```

图2-42 计算缓存等待周期

需要指出的是，流量整形的阈值limit越大，流量整形的精度越高，流量整形功能是可靠性的一种保障，它无法做到100%的精确。这个跟后端的编解码以及缓冲区的处理策略相关，此处不再赘述。感兴趣的朋友可以思考下，Netty为什么不做到 100%的精确。

流量整形与流控的最大区别在于流控会拒绝消息，流量整形不拒绝和丢弃消息，无论接收量多大，它总能以近似恒定的速度下发消息，跟变压器的原理和功能类似。

2.6.2. 单条链路流量整形

除了全局流量整形，Netty也支持但链路的流量整形，相关的接口定义如下：

```
 */
public ChannelTrafficShapingHandler(long writeLimit,
        long readLimit, long checkInterval) {
    super(writeLimit, readLimit, checkInterval);
}
```

图2-43 单链路流量整形

单链路流量整形与全局流量整形的最大区别就是它以单个链路为作用域，可以对不同的链路设置不同的整形策略。

它的实现原理与全局流量整形类似，我们不再赘述。值得说明的是，Netty支持用户自定义流量整形策略，通过继承AbstractTrafficShapingHandler的doAccounting方法可以定制整形策略。相关接口定义如下：

```
/**
 * Called each time the accounting is computed from the Traffic
 * This method could be used for instance to implement almost r
 *
 * @param counter
 *            the TrafficCounter that computes its performance
 */
@SuppressWarnings("unused")
protected void doAccounting(TrafficCounter counter) {
    // NOOP by default
}
```

图2-44 定制流量整形策略

3. 总结

尽管Netty在架构可靠性上面已经做了很多精细化的设计，以及基于防御式编程对系统进行了大量可靠性保护。但是，系统的可靠性是个持续投入和改进的过程，不可能在一个版本中一蹴而就，可靠性工作任重而道远。

从业务的角度看，不同的行业、应用场景对可靠性的要求也是不同的，例如电信行业的可靠性要求是5个9，对于铁路等特殊行业，可靠性要求更高，达到6个9。对于企业的一些边缘IT系统，可靠性要求会低些。

可靠性是一种投资，对于企业而言，追求极端可靠性对研发成本是个沉重的包袱，但是相反，如果不重视系统的可靠性，一旦不幸遭遇网上事故，损失往往也是惊人的。

对于架构师和设计师，如何权衡架构的可靠性和其它特性的关系，是一个很大的挑战。通过研究和学习Netty的可靠性设计，也许能够给大家带来一些启示。

4. Netty学习推荐书籍

目前市面上介绍netty的文章很多，如果读者希望系统性的学习Netty，推荐两本书：

1) 《Netty in Action》

2) 《Netty权威指南》

5.作者简介

李林锋，2007年毕业于东北大学，2008年进入华为公司从事高性能通信软件的设计和开发工作，有6年NIO设计和开发经验，精通Netty、Mina等NIO框架。Netty中国社区创始人，《Netty权威指南》作者。

感谢郭蕾对本文的审校和策划。