15 Netty服务如何写出数据

更新时间：2020-07-29 09:40:29

> 天才就是长期劳动的结果。——牛顿

前言

你好，我是彤哥。

上一节，我们一起学习了 Netty 接收新数据过程的源码剖析，我们又发现了一个有趣的现象，Netty 的 ByteBuf 竟然也是对 Java 原生 ByteBuffer 的包装。

经过前面的学习，我想你一定迫不及待地想知道 Netty 中写出数据的过程了吧，也有可能，你自己已经根据我们前面的调试方法自己看了，也有可能，睿智的你已经猜测 Netty 中写出数据，肯定也是调用 Java 原生 SocketChannel 的写出数据。

那么，是不是如你所猜测呢？让我们进入今天的学习吧。

问题

我们知道，Java 原生 NIO 写出数据是从 ByeBuffer 中写出的，也就是下面这样的代码：

```
private static void send(SocketChannel socketChannel, String msg) {
    try {
        ByteBuffer writeBuffer = ByteBuffer.allocate(1024);
        writeBuffer.put(msg.getBytes());
        writeBuffer.flip();
        // 调用SocketChannel的写出方法
        socketChannel.write(writeBuffer);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

那么，今天的问题是：

1. Netty 底层是不是调用的 SocketChannel 的 write () 方法呢？
2. 写出数据在 ChannelPipeline 中的传递顺序是怎样的？
3. 写出为什么还有一个叫做 flush 的过程？之前写出的数据又在哪里呢？

好了，让我们带着这些问题进入今天的探索吧。

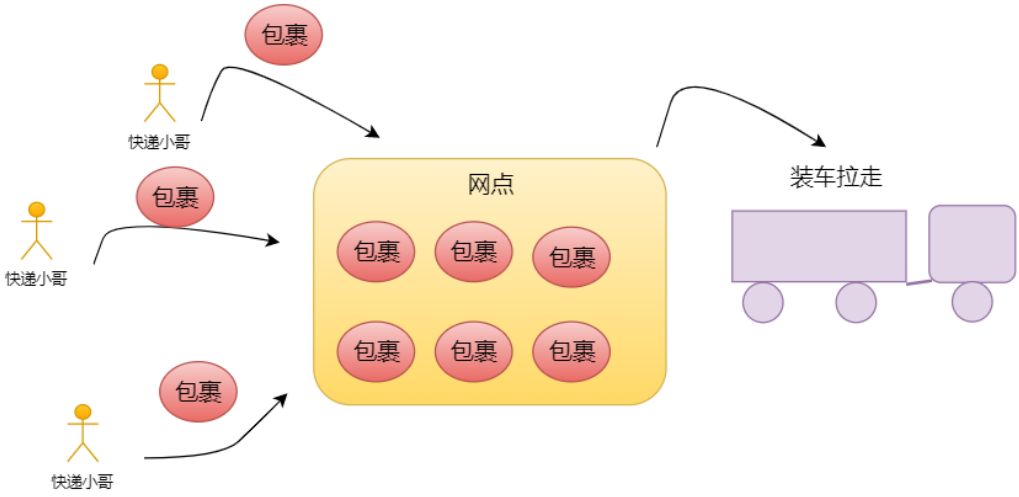
服务写出数据过程

今天的内容调试起来比较简单，我们继续使用 EchoServer 这个案例，为了便于讲解，我们先把 StringDecoder/StringEncoder 相关的内容注释掉，直接在 EchoServerHandler 的 channelRead () 方法中打个断点即可：


```
public class EchoServerHandler extends ChannelInboundHandlerAdapter {
    // 省略其它代码
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        // 读取数据后写回客户端
        // key1，断点在此处
        ctx.write(msg);
    }
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        // key2，这里也需要个断点
        ctx.flush();
    }
}
```

不过，一般来说，服务端发送数据不会在每次 write () 的时候就发送出去，而是先缓存起来，等到一定量之后或者显式地说明要发送的时候再真正地发送出去，这样能在一定程度上提高效率。

就像快递公司一般都会在各个地方设置网点一样，快递小哥并不是收取完快递就给你发出去了，而是，先集成存放在网点，等达到一定量之后，或者到晚上八九点钟之后，再装车发送出去。快递小哥往网点投放快递就类似于 write (msg) 的过程，而装车拉走就类似于 flush () 的过程。



所以，上面的代码，如果仅仅调用了 ctx.write(msg)，客户端可能并不能及时地收取到消息，我们还需要调用一个叫做 ctx.flush() 的方法，才能真正地把数据发送给客户端。为了与读取消息的过程一致，我们这里把 ctx.flush() 方法的调用放在了 channelReadComplete() 方法中，因此，这里也需要打个断点。

好了，让我们先跟踪第一个方法 —— ctx.write(msg) 吧。

清除其它断点，只保留这两个断点，启动服务，使用 XSHELL 模拟客户端发送一条消息过来，我们还是以 12345 为例，可以发现，断点成功停留在了 ctx.write(msg) 这一行，跟踪进去看看：

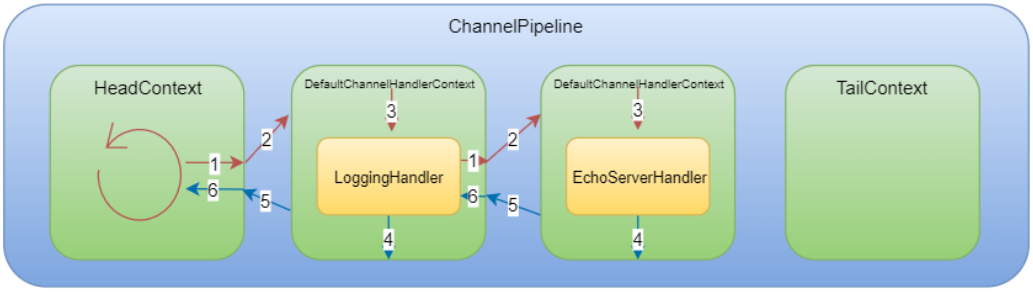

```
// 1. io.netty.channel.AbstractChannelHandlerContext#write
@Override
public ChannelFuture write(Object msg) {
    return write(msg, newPromise());
}
// 2. io.netty.channel.AbstractChannelHandlerContext#write
@Override
public ChannelFuture write(final Object msg, final ChannelPromise promise) {
    // 第二个参数为flush，这里传入的是false
    // 也就是默认不进行真正地发送
    write(msg, false, promise);

    return promise;
}
// 3. io.netty.channel.AbstractChannelHandlerContext#write
private void write(Object msg, boolean flush, ChannelPromise promise) {
    // 检查参数，可以跳过
    ObjectUtil.checkNotNull(msg, "msg");
    try {
        if (isNotValidPromise(promise, true)) {
            ReferenceCountUtil.release(msg);
            // cancelled
            return;
        }
    } catch (RuntimeException e) {
        ReferenceCountUtil.release(msg);
        throw e;
    }
    // key1，寻找下一个可用的outbound类型的ChannelHandlerContext
    // 在ChannelPipeline中也就是prev指针标记的那个
    // 这里找到的就是LoggingHandler对应的那个ChannelHandlerContext
    final AbstractChannelHandlerContext next = findContextOutbound(flush ?
                                                    (MASK_WRITE | MASK_FLUSH) : MASK_WRITE);
    // touch()可以先不管，可以把这里返回的对象看作与msg是一个对象
```

```
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            // key2，flush为false，所以走到了这里
            // 调用context的invokeWrite()方法
            next.invokeWrite(m, promise);
        }
    } else {
        final WriteTask task = WriteTask.newInstance(next, m, promise, flush);
        if (!safeExecute(executor, task, promise, m, !flush)) {
            task.cancel();
        }
    }
}
// 4. io.netty.channel.AbstractChannelHandlerContext#invokeWrite
void invokeWrite(Object msg, ChannelPromise promise) {
    if (invokeHandler()) {
        invokeWrite0(msg, promise);
    } else {
        write(msg, promise);
    }
}
// 5. io.netty.channel.AbstractChannelHandlerContext#invokeWrite0
private void invokeWrite0(Object msg, ChannelPromise promise) {
    try {
        // 调用Handler的write()方法
        ((ChannelOutboundHandler) handler()).write(this, msg, promise);
    } catch (Throwable t) {
        notifyOutboundHandlerException(t, promise);
    }
}
// 6. io.netty.handler.logging.LoggingHandler#write
@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
    if (logger.isEnabled(internalLevel)) {
        logger.log(internalLevel, format(ctx, "WRITE", msg));
    }
    // 又调回了第2步中的方法，继续寻找下一个可用的outbound类型的ChannelHandlerContext
    ctx.write(msg, promise);
}
```


所以，到这里，第 2 个问题我们倒是先解决了，数据在 ChannelPipeline 中的传递，也是通过 ChannelHandlerContext 进行的，每次寻找下一个可用的 outbound 类型的 ChannelHandlerContext，调用它里面的 ChannelHandler 的 write () 方法，然后，在 ChannelHandler 里面调用 `ctx.write(msg, promise)` 才能让这个链往下传递，继续寻找下一个可用的 outbound 类型的 ChannelHandlerContext。F

因此，我们可以把上一节的图补全为下面这样：



为了便于理解，我特意把箭头区分成两种颜色，并标上数字：

- 红色表示接收数据的过程，蓝色表示写出数据的过程；
- 1 表示调用 ctx.fireChannelRead (msg) 方法，触发下一个 ChannelHandlerContext 的调用；
- 2 表示调用 next.invokeChannelRead (m) 方法，调用到下一个 ChannelHandlerContext；
- 3 表示调用 ((ChannelInboundHandler) handler ()).channelRead (this, msg) 方法，调用 ChannelHandler 的 channelRead () 方法；
- 4 表示调用 ctx.write (msg) 或者 ctx.write (msg, promise) 方法，触发下一个 ChannelHandlerContext 的调用；
- 5 表示调用 next.invokeWrite (m, promise) 方法，调用到下一个 ChannelHandlerContext；
- 6 表示调用 ((ChannelOutboundHandler) handler ()).write (this, msg, promise) 方法，调用 ChannelHandler 的 write () 方法；

所以，对于接收数据，如果需要数据在 ChannelPipeline 中传递，就调用 `ctx.fireChannelRead(msg)` 方法；对于写出数据，如果需要数据在 ChannelPipeline 中传递，就调用 `ctx.write(msg)` 或者 `ctx.write(msg, promise)` 方法。

通过上面的图，我们知道， 最后一定会走到 ChannelPipeline 的 HeadContext 类的 write () 方法，所以，直接在这个方法中打一个断点，或者慢慢一步一步走到这里都可以：

```
// io.netty.channel.DefaultChannelPipeline.HeadContext#write
@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
    // 调用unsafe的write()方法
    unsafe.write(msg, promise);
}
// io.netty.channel.AbstractChannel.AbstractUnsafe#write
@Override
public final void write(Object msg, ChannelPromise promise) {
    assertEventLoop();
    // key，看起来像是一个缓存的东西
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        safeSetFailure(promise, newClosedChannelException(initialCloseCause));
        ReferenceCountUtil.release(msg);
        return;
    }

    int size;
    try {
        // 过滤消息
        msg = filterOutboundMessage(msg);
        // 计算消息的大小
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        ReferenceCountUtil.release(msg);
        return;
    }

    // key，添加到缓存中
    outboundBuffer.addMessage(msg, size, promise);
}
// io.netty.channel.ChannelOutboundBuffer#addMessage
public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    // 典型的单链表添加元素的过程
    if (tailEntry == null) {
        flushedEntry = null;
    } else {
        Entry tail = tailEntry;
        tail.next = entry;
    }
    tailEntry = entry;
    if (unflushedEntry == null) {
        unflushedEntry = entry;
    }

    incrementPendingOutboundBytes(entry.pendingSize, false);
}
```


所以，`ctx.write()` 方法最终只是把消息添加到了一个叫做 `ChannelOutboundBuffer` 的缓存中，并没有真正地发送出去。

那么，我们要怎么寻找在哪里真正发送数据出去的呢？

好了，我们在 `EchoServerHandler` 中打的第二个断点要登场了，`ctx.flush();` 的调用过程跟 `ctx.write(msg)` 是类似的，我们就不再赘述了，直接来到 HeadContext 的 `flush()` 方法。

```
// io.netty.channel.DefaultChannelPipeline.HeadContext#flush
@Override
public void flush(ChannelHandlerContext ctx) {
```

```
    unsafe.flush();
}
// io.netty.channel.AbstractChannel.AbstractUnsafe#flush
@Override
public final void flush() {
    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    // key，不知道大家发现没，这种带0结尾的一般也是干正事的方法
    flush0();
}
// io.netty.channel.nio.AbstractNioChannel.AbstractNioUnsafe#flush0
@Override
protected final void flush0() {
    if (!isFlushPending()) {
        super.flush0();
    }
}
// io.netty.channel.AbstractChannel.AbstractUnsafe#flush0
protected void flush0() {
    // 省略其它代码

    try {
        doWrite(outboundBuffer);
    } catch (Throwable t) {
        // // 省略其它代码
    } finally {
        inFlush0 = false;
    }
}
// io.netty.channel.socket.nio.NioSocketChannel#doWrite
@Override
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    SocketChannel ch = javaChannel();
    int writeSpinCount = config().getWriteSpinCount();
    do {
        if (in.isEmpty()) {
            clearOpWrite();
            return;
        }

        // Ensure the pending writes are made of ByteBufs only.
        int maxBytesPerGatheringWrite = ((NioSocketChannelConfig) config).getMaxBytesPerGatheringWrite();
        // key1，从ChannelOutboundBuffer中取出ByteBuffer
        // 前面分析write()的时候放里面放的实际是ByeBuf
        // 因为ByteBuf实际上是对ByteBuffer的包装
        // 所以这里取出来的时候直接就转换成ByteBuffer了
        ByteBuffer[] nioBuffers = in.nioBuffers(1024, maxBytesPerGatheringWrite);
        int nioBufferCnt = in.nioBufferCount();

        // Always us nioBuffers() to workaround data-corruption.
        // See https://github.com/netty/netty/issues/2761
        switch (nioBufferCnt) {
            case 0:
                writeSpinCount -= doWrite0(in);
                break;
            case 1: {
                // 我们只写了一条数据，所以实际是走到了这里
                ByteBuffer buffer = nioBuffers[0];
                int attemptedBytes = buffer.remaining();
                // key2，调用SocketChannel的write()方法写出数据
                final int localWrittenBytes = ch.write(buffer);
                if (localWrittenBytes <= 0) {
                    incompleteWrite(true);
                    return;
```

```
                return;
            }
            adjustMaxBytesPerGatheringWrite(attemptedBytes, localWrittenBytes, maxBytesPerGatheringWrite);
            in.removeBytes(localWrittenBytes);
            --writeSpinCount;
            break;
        }
        default: {
            long attemptedBytes = in.nioBufferSize();
            // key2，调用SocketChannel的write()方法写出数据
            final long localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);
            if (localWrittenBytes <= 0) {
                incompleteWrite(true);
                return;
            }
            adjustMaxBytesPerGatheringWrite((int) attemptedBytes, (int) localWrittenBytes,
                                            maxBytesPerGatheringWrite);
            in.removeBytes(localWrittenBytes);
            --writeSpinCount;
            break;
        }
    }
  } while (writeSpinCount > 0);

  incompleteWrite(writeSpinCount < 0);
}
```

这里有两个关键方法：

- 先从 ChannelOutboundBuffer 中取出 ByteBuffer；
- 再通过 Java 原生的 SocketChannel 写出数据。

至此，写出数据过程的源码剖析就讲完了，让我们再来总结一下：

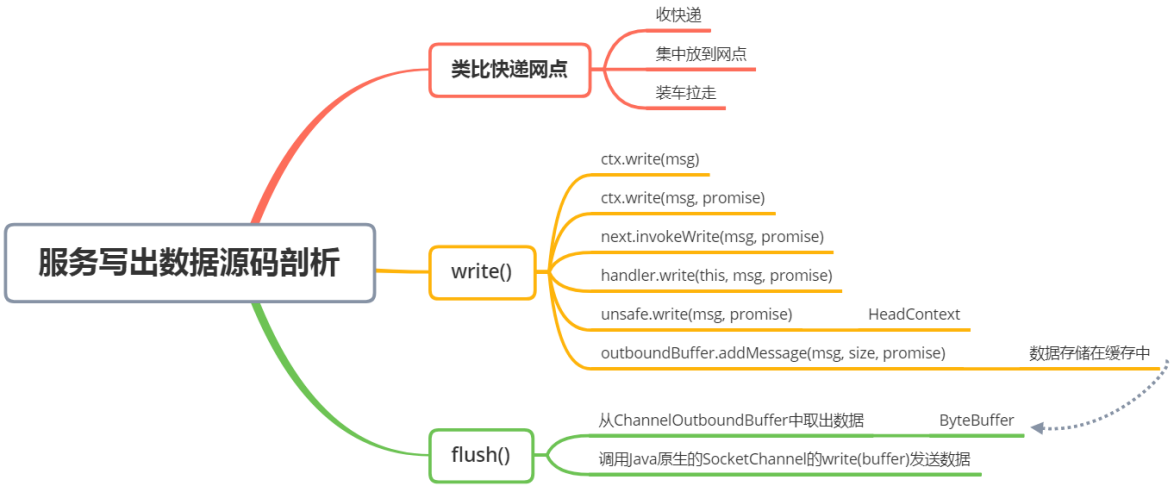
1. 调用 ctx.write () 方法时，只是把数据添加到 ChannelOutboundBuffer 缓存中；
2. 调用 ctx.flush () 方法时，才把数据从 ChannelOutboundBuffer 取出来；
3. 调用 Java 原生的 SocketChannel 把数据发送出去。

后记

本节，我们一起学习了 Netty 中服务写出数据过程的源码剖析，通过阅读源码，我们知道，Netty 中写出数据实际上分成了两步：ctx.write () 和 ctx.flush ()，write () 的时候只是把数据添加到缓存中，flush () 才真正把数据发送出去，之所以要分成两步，也是基于效率来考虑的，大家可以类比快递网点的生活案例进行对比，如果没有网点，则需要接收到快递就装车拉走，将要耗费巨大的人力物力财力。

到这里，数据流向角度的源码分析就快结束了，下一节，我们将看看 Netty 是如何做到优雅关闭的，敬请期待。

思维导图



}

14 Netty服务如何接收新的数据

16 Netty服务如何优雅关闭