- NETTY

精尽 Netty 源码解析 —— Channel（四）之 write 操作

1. 概述

本文分享 Netty NioSocketChannel **写入**对端数据的过程。和**写入**相关的，在 Netty Channel 有三种 API 方法：

```
ChannelFuture write(Object msg)
ChannelFuture write(Object msg, ChannelPromise promise);

ChannelOutboundInvoker flush();

ChannelFuture writeAndFlush(Object msg);
ChannelFuture writeAndFlush(Object msg, ChannelPromise promise);
```

原生的 Java NIO SocketChannel 只有一种 write 方法，将数据写到对端。而 Netty Channel 竟然有三种方法，我们来一个个看看：

- write 方法：将数据写到**内存队列**中。
  - 也就是说，此时数据**并没有**写入到对端。
- flush 方法：刷新**内存队列**，将其中的数据写入到对端。
  - 也就是说，此时数据才**真正**写到对端。
- writeAndFlush 方法：write + flush 的组合，将数据写到内存队列后，立即刷新**内存队列**，又将其中的数据写入到对端。
  - 也就是说，此时数据**已经**写到对端。

严格来说，上述的描述不是完全准确。因为 Netty Channel 的 `#write(Object msg, ...)` 和 `#writeAndFlush(Object msg, ...)` 方法，是**异步写入**的过程，需要通过监听返回的 ChannelFuture 来确实是真正写入。例如：

```
// 方式一：异步监听
channel.write(msg).addListener(new ChannelFutureListener() {

    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        // ... 相关逻辑，例如是否成功？
```

文章目录

注意。😈 如果感兴趣，可以看看 Dubbo 和 Motan 等等 RPC 框架是怎么使用这个

write(Object msg, ...) 方法返回的 Promise 对象，只有在数据真正被
才会被回调通知。如果胖友不理解，请自己测试一下。

对端数据的代码太多，所以笔者拆成 write 和 flush 相关的两篇文章。所以，本文当然是
相关性很高，所以本文也会包括 flush 部分的内容。

AbstractChannel 对 `#write(Object msg, ...)` 方法的实现，代码如下：

```
@Override
public ChannelFuture write(Object msg) {
    return pipeline.write(msg);
}

@Override
public ChannelFuture write(Object msg, ChannelPromise promise) {
    return pipeline.write(msg, promise);
}
```

- 在方法内部，会调用对应的 `ChannelPipeline#write(Object msg, ...)` 方法，将 write 事件在 pipeline 上传播。详细解析，见 「3. DefaultChannelPipeline」 。
  - 最终会传播 write 事件到 `head` 节点，将数据写入到内存队列中。详细解析，见 「5. HeadContext」 。

3. DefaultChannelPipeline

`DefaultChannelPipeline#write(Object msg, ...)` 方法，代码如下：

```
@Override
public final ChannelFuture write(Object msg) {
    return tail.write(msg);
}

@Override
public final ChannelFuture write(Object msg, ChannelPromise promise) {
    return tail.write(msg, promise);
}
```

- 在方法内部，会调用 `TailContext#write(Object msg, ...)` 方法，将 write 事件在 pipeline 中，从尾节点向头节
text」 。

Object msg, ...) 方法的实现，是从 AbstractChannelHandlerContext 抽象类继

```
(Object msg) {
romise());

(final Object msg, final ChannelPromise promise) {
出异常

terException("msg");

12:
13:     try {
14:         // 判断是否为合法的 Promise 对象
15:         if (isNotValidPromise(promise, true)) {
16:             // 释放消息( 数据 )相关的资源
17:             ReferenceCountUtil.release(msg);
18:             // cancelled
19:             return promise;
20:         }
21:     } catch (RuntimeException e) {
22:         // 发生异常，释放消息( 数据 )相关的资源
23:         ReferenceCountUtil.release(msg);
24:         throw e;
25:     }
26:
27:     // 写入消息( 数据 )到内存队列
28:     write(msg, false, promise);
29:     return promise;
30: }
```

- 在【第 2 行】的代码，我们可以看到， `#write(Object msg)` 方法，会调用 `#write(Object msg, ChannelPromise promise)` 方法。
  - 缺少的 `promise` 方法参数，通过调用 `#newPromise()` 方法，进行创建 Promise 对象，代码如下：

```
@Override
public ChannelPromise newPromise() {
    return new DefaultChannelPromise(channel(), executor());
}
```

    - 返回 DefaultChannelPromise 对象。
  - 在【第 29 行】的代码，返回的结果就是传入的 `promise` 对象。
- 第 8 至 11 行：若消息(消息)为空，抛出异常。
- 第 15 行：调用 `#isNotValidPromise(promise, true)` 方法，判断是否为**不合法**的 Promise 对象。该方法，在 elPipeline（四）之 Outbound 事件的传播》 中已经详细解析。
  tUtil#release(msg)` 方法，释放释放消息(数据)相关的资源。
  一般情况下，出现这种情况是 `promise` 已经被取消，所以不再有必要写入数
  取消。
  用 `ReferenceCountUtil#release(msg)` 方法，释放释放消息(数据)相关的

  sg, boolean flush, ChannelPromise promise)` 方法，写入消息(数据)到内

```
t msg, boolean flush, ChannelPromise promise) {
d 节点
erContext next = findContextOutbound();

eline.touch(msg, next);
or = next.executor();
程中
Loop()) {
flush 事件到下一个节点

riteAndFlush(m, promise);
        // 执行 write 事件到下一个节点
        } else {
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;
        // 创建 writeAndFlush 任务
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        // 创建 write 任务
        }  else {
            task = WriteTask.newInstance(next, m, promise);
        }
        // 提交到 EventLoop 的线程中，执行该任务
        safeExecute(executor, task, promise, m);
    }
}
```

- 方法参数 `flush` 为 `true` 时，该方法执行的是 write + flush 的组合操作，即将数据写到内存队列后，立即刷新**内存队列**，又将其中的数据写入到对端。
- 第 3 行：调用 `#findContextOutbound()` 方法，获得**下一个** Outbound 节点。
- 第 5 行：调用 `DefaultChannelPipeline#touch(Object msg, AbstractChannelHandlerContext next)` 方法，记录 Record 记录。代码如下：

```
java
msg, AbstractChannelHandlerContext next) {
ceCountUtil.touch(msg, next) : msg;

ferenceCounted#touch(Object)} if the specified message implements
}.  If the specified message doesn't implement {@link ReferenceCoun
g.

@SuppressWarnings("unchecked")
public static <T> T touch(T msg, Object hint) {
    if (msg instanceof ReferenceCounted) {
        return (T) ((ReferenceCounted) msg).touch(hint);
    }
    return msg;
}
```

- 详细解析，见《精尽 Netty 源码解析 —— Buffer 之 ByteBuf（三）内存泄露检测》 。

- 第 7 行：**在** EventLoop 的线程中。
  - 第 10 至 11 行：如果 `flush = true` 时，调用 `AbstractChannelHandlerContext#invokeWriteAndFlush()` 方法，执行 writeAndFlush 事件到下一个节点。
  - 第 12 至 15 行：如果 `flush = false` 时，调用 `AbstractChannelHandlerContext#invokeWrite()` 方法，执行 write 事件到下一个节点。
  - 后续的逻辑，和《精尽 Netty 源码解析 —— ChannelPipeline（四）之 Outbound 事件的传播》分享的 **bind** 事件在 pipeline 中的传播是**基本一致**的。
  - 随着 write 或 writeAndFlush **事件**不断的向下一个节点传播，最终会到达 HeadContext 节点。详细解析，见「5. HeadContext」 。
- 第 16 行：**不在** EventLoop 的线程中。
  - 第 19 至 20 行：如果 `flush = true` 时，创建 WriteAndFlushTask 任务。
  - 第 21 至 24 行：如果 `flush = false` 时，创建 WriteTask 任务。
  - 第 26 行：调用 `#safeExecute(executor, task, promise, m)` 方法，提交到 EventLoop 的线程中，执行该任务。从而实现，**在** EventLoop 的线程中，执行 writeAndFlush 或 write 事件到下一个节点。详细解析，见「7. AbstractWriteTask」 中。
- 第 29 行：返回 `promise` 对象。

5. HeadContext

在 pipeline 中，write 事件最终会到达 HeadContext 节点。而 HeadContext 的 `#write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)` 方法，会处理该事件，代码如下：


```
erContext ctx, Object msg, ChannelPromise promise) throws Exception {
```

afe#write(Object msg, ChannelPromise promise) 方法，将数据写到**内存队**
nsafe」 。

g, ChannelPromise promise) 方法，将数据写到**内存队列**中。代码如下：

```
dBuffer outboundBuffer = new ChannelOutboundBuffer(AbstractChannel.this

bject msg, ChannelPromise promise) {
    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    // 内存队列为空
    if (outboundBuffer == null) {
        // 内存队列为空，一般是 Channel 已经关闭，所以通知 Promise 异常结果
        // If the outboundBuffer is null we know the channel was closed and so
        // need to fail the future right away. If it is not null the handling of the rest
        // will be done in flush0()
        // See https://github.com/netty/netty/issues/2362
        safeSetFailure(promise, WRITE_CLOSED_CHANNEL_EXCEPTION);
        // 释放消息( 对象 )相关的资源
        // release message now to prevent resource-leak
        ReferenceCountUtil.release(msg);
        return;
    }

    int size;
    try {
        // 过滤写入的消息( 数据 )
        msg = filterOutboundMessage(msg);
        // 计算消息的长度
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        // 通知 Promise 异常结果
        safeSetFailure(promise, t);
        // 释放消息( 对象 )相关的资源
        ReferenceCountUtil.release(msg);
        return;
    }

存队列
sage(msg, size, promise);
```


], 用于缓存写入的数据(消息)。

annel **已经关闭**。

se, WRITE_CLOSED_CHANNEL_EXCEPTION) 方法，通知 Promise 异常结果。

tUtil#release(msg) 方法，释放释放消息(数据)相关的资源。

Channel#filterOutboundMessage(msg) 方法，过滤写入的消息(数据)。代码如

```
va

rOutboundMessage(Object msg) {

uf) {
        ByteBuf buf = (ByteBuf) msg;
        // 已经是内存 ByteBuf
        if (buf.isDirect()) {
            return msg;
        }

        // 非内存 ByteBuf ，需要进行创建封装
        return newDirectBuffer(buf);
    }

    // <2> FileRegion 的情况
    if (msg instanceof FileRegion) {
        return msg;
    }

    // <3> 不支持其他类型
    throw new UnsupportedOperationException("unsupported message type: " + StringUtil.simpleClassN
}
```

- <1> 处，消息(数据)是 ByteBuf 类型，如果是非 Direct ByteBuf 对象，需要调用 #newDirectBuffer(ByteBuf) 方法，复制封装成 Direct ByteBuf 对象。原因是：在使用 Socket 传递数据时性能很好，由于数据直接在内存中，不存在从 JVM 拷贝数据到直接缓冲区的过程，性能好。(来自 《[netty核心类]–缓冲区ByteBuf》)
- <2> 处，消息(数据)是 FileRegion 类型，直接返回。
- <3> 处，不支持其他数据类型。

- 第 24 至 28 行：计算消息的长度。
- 第 29 行：若发生异常时：
  - 第 31 行：调用 #safeSetFailure(promise, Throwable t) 方法，通知 Promise 异常结果。
  - 第 33 行：调用 ReferenceCountUtil#release(msg) 方法，释放释放消息(数据)相关的资源。
  - 第 34 行： return ，结束执行。
- 第 38 行：调用 ChannelOutboundBuffer#addMessage(msg, size, promise) 方法，写入消息(数据)到内存队列。关于 ChannelOutboundBuffer ，我们在 《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》 中，详细解析。

队列中的过程。

还是需要看完 《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》 一文。

sk

口，写入任务**抽象类**。它有两个子类实现：

任务实现类。

text 的内部静态类。那么让我们先来 AbstractWriteTask 的代码。

```
riteTask 对象的自身占用内存大小

TIMATE_TASK_SIZE_ON_SUBMIT = SystemPropertyUtil.getBoolean("io.netty.tr
```

```
/**
 * 每个 AbstractWriteTask 对象自身占用内存的大小。
 */
// Assuming a 64-bit JVM, 16 bytes object header, 3 reference fields and one int field, plus alignment
private static final int WRITE_TASK_OVERHEAD = SystemPropertyUtil.getInt("io.netty.transport.writeTask

private final Recycler.Handle<AbstractWriteTask> handle;
/**
 * pipeline 中的节点
 */
private AbstractChannelHandlerContext ctx;
/**
 * 消息( 数据 )
 */
private Object msg;
/**
 * Promise 对象
 */
private ChannelPromise promise;
/**
 * 对象大小
 */
private int size;

@SuppressWarnings("unchecked")
private AbstractWriteTask(Recycler.Handle<? extends AbstractWriteTask> handle) {
    this.handle = (Recycler.Handle<AbstractWriteTask>) handle;
}
```

- 每个字段，看代码注释。
- `ESTIMATE_TASK_SIZE_ON_SUBMIT` **静态**字段，提交任务时，是否计算 AbstractWriteTask 对象的自身占用内存大小。
- `WRITE_TASK_OVERHEAD` **静态**字段，每个 AbstractWriteTask 对象自身占用内存的大小。为什么占用的 48 字节呢?

，对象头，16 字节。
个**对象引用**字段，3 * 8 = 24 字节。
字段，4 字节。
倍，因此 4 字节。
M 虚拟机，并且不考虑压缩)。
JVM中 对象的内存布局 以及 实例分析》 。
而 Recycler 是 Netty 用来实现对象池的工具类。在网络通信中，写入是非常频繁的
ractWriteTask 对象，减少对象的频繁创建，降低 GC 压力，提升性能。

AbstractChannelHandlerContext ctx, Object msg, ChannelPromise
eTask 对象。代码如下：

```
ractWriteTask task, AbstractChannelHandlerContext ctx, Object msg, Chan

象大小 <1>
UBMIT) {
    task.size = ctx.pipeline.estimatorHandle().size(msg) + WRITE_TASK_OVERHEAD;
    // 增加 ChannelOutboundBuffer 的 totalPendingSize 属性  <2>
    ctx.pipeline.incrementPendingOutboundBytes(task.size);
} else {
    task.size = 0;
}
}
```

- 在下文中，我们会看到 AbstractWriteTask 对象是从 Recycler 中获取，所以获取完成后，需要通过该方法，初始化该对象的属性。
- `<1>` 处，计算 AbstractWriteTask 对象大小。并且在 `<2>` 处，调用 `ChannelPipeline#incrementPendingOutboundBytes(long size)` 方法，增加 ChannelOutboundBuffer 的 `totalPendingSize` 属性。代码如下：

```
// DefaultChannelPipeline.java
@UnstableApi
protected void incrementPendingOutboundBytes(long size) {
    ChannelOutboundBuffer buffer = channel.unsafe().outboundBuffer();
    if (buffer != null) {
        buffer.incrementPendingOutboundBytes(size);
    }
}
```

- 内部，会调用 `ChannelOutboundBuffer#incrementPendingOutboundBytes(long size)` 方法，增加 ChannelOutboundBuffer 的 `totalPendingSize` 属性。详细解析，见《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》的「10.1 incrementPendingOutboundBytes」小节。

7.3 run

`#run()` **实现**方法，

```
1: @Override
```


oundBuffer 的 totalPendingSize 属性 <1>
s it may be set to null if the channel is closed already
IZE_ON_SUBMIT) {
crementPendingOutboundBytes(size);

下一个节点
mise);

```
he GC can collect them directly

);
```

- 在 `<1>` 处，调用 `ChannelPipeline#decrementPendingOutboundBytes(long size)` 方法，减少 ChannelOutboundBuffer 的 `totalPendingSize` 属性。代码如下：

```
@UnstableApi
protected void decrementPendingOutboundBytes(long size) {
    ChannelOutboundBuffer buffer = channel.unsafe().outboundBuffer();
    if (buffer != null) {
        buffer.decrementPendingOutboundBytes(size);
    }
}
```

  - 内部，会调用 `ChannelOutboundBuffer#decrementPendingOutboundBytes(long size)` 方法，减少 ChannelOutboundBuffer 的 `totalPendingSize` 属性。详细解析，见《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》的「10.2 decrementPendingOutboundBytes」小节。

- 第 10 行：调用 `#write(ctx, msg, promise)` 方法，执行 write 事件到下一个节点。代码如下：

```
protected void write(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
    ctx.invokeWrite(msg, promise);
}
```

- 第 11 至 19 行：置空 AbstractWriteTask 对象，并调用 `Recycler.Handle#recycle(this)` 方法，回收该对象。

7.4 WriteTask

WriteTask ，实现 SingleThreadEventLoop.NonWakeupRunnable 接口，继承 AbstractWriteTask 抽象类，write 任务实现类。

为什么会实现 SingleThreadEventLoop.NonWakeupRunnable 接口呢？write 操作，仅仅将数据写到**内存队列**中，无需唤醒 EventLoop ，从而提升性能。关于 SingleThreadEventLoop.NonWakeupRunnable 接口，在《精尽 Netty 源码解析 —— EventLoop（三）之 EventLoop 初始化》有详细解析。

7.4.1 newInstance

`#newInstance(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise)` 方法，创建


```
riteTask> RECYCLER = new Recycler<WriteTask>() {

ct(Handle<WriteTask> handle) {
ndle); // 创建 WriteTask 对象
```

```
stance(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise pro
得 WriteTask 对象
get();
性
se);
```

7.4.2 构造方法

```
private WriteTask(Recycler.Handle<WriteTask> handle) {
    super(handle);
}
```

7.4.3 write

WriteTask 无需实现 `#write(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise)` 方法，直接**重用**父类该方法即可。

7.5 WriteAndFlushTask

WriteAndFlushTask ，继承 WriteAndFlushTask 抽象类，write + flush 任务实现类。

7.5.1 newInstance

`#newInstance(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise)` 方法，创建 WriteAndFlushTask 对象。代码如下：

```
private static final Recycler<WriteAndFlushTask> RECYCLER = new Recycler<WriteAndFlushTask>() {

    @Override
    protected WriteAndFlushTask newObject(Handle<WriteAndFlushTask> handle) {
        return new WriteAndFlushTask(handle); // 创建 WriteAndFlushTask 对象
    }

};

private static WriteAndFlushTask newInstance(AbstractChannelHandlerContext ctx, Object msg,  ChannelPr
    // 从 Recycler 的对象池中获得 WriteTask 对象
    WriteAndFlushTask task = RECYCLER.get();
    // 初始化 WriteTask 对象的属性
se);
```

```
ler.Handle<WriteAndFlushTask> handle) {
```

ontext ctx, Object msg, ChannelPromise promise) 方法，在父类的该方法的节点。代码如下：

```
public void write(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
    // 执行 write 事件到下一个节点
    super.write(ctx, msg, promise);
    // 执行 flush 事件到下一个节点
    ctx.invokeFlush();
}
```

666. 彩蛋

最后，我们来看一个真的彩蛋，嘿嘿嘿。

在一些 ChannelHandler 里，我们想要写入数据到对端，可以有两种写法，代码如下：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ctx.write(msg); // <1>
    ctx.channel().write(msg); // <2>
}
```

这两者有什么异同呢？

- `<2>` 种，实际就是本文所描述的，将 write 事件，从 pipeline 的 `tail` 节点到 `head` 节点的过程。
- `<1>` 种，和 `<2>` 种**不同**，将 write 事件，从当前的 `ctx` 节点的**下一个**节点到 `head` 节点的过程。
- 为什么呢？胖友自己调试理解下。😁😁😁

推荐阅读文章：

- 占小狼 《深入浅出Netty write》
- 闪电侠 《netty 源码分析之 writeAndFlush 全解析》

访问量 次