- NETTY

精尽 Netty 源码解析 —— Channel（三）之 read 操作

1. 概述

据的过程、和 Netty NIO 客户端接收(**read**)来自服务端数据的结果。实

文章目录

etChannel 。
应的 Netty NioSocketChannel 。

对端的数据的过程。

简单来说：

1. NioSocketChannel 所在的 EventLoop 线程轮询是否有新的数据写入。
2. 当轮询到有新的数据写入，NioSocketChannel 读取数据，并提交到 pipeline 中进行处理。

比较简单，和《精尽 Netty 源码解析 —— Channel（二）之 accept 操作》有几分相似。或者我们可以说：

- NioServerSocketChannel 读取新的连接。
- NioSocketChannel 读取新的数据。

2. NioByteUnsafe#read

NioByteUnsafe ，实现 AbstractNioUnsafe 抽象类，AbstractNioByteChannel 的 Unsafe 实现类。代码如下：

```
protected class NioByteUnsafe extends AbstractNioUnsafe {

    public final void read() { /** 省略内部实现 **/ }

    private void handleReadException(ChannelPipeline pipeline, ByteBuf byteBuf, Throwable cause, boole

    private void closeOnRead(ChannelPipeline pipeline) { /** 省略内部实现 **/ }

}
```

- 一共有 3 个方法。但是实现上，入口为 `#read()` 方法，而另外 2 个方法被它所调用。所以，我们赶紧开始 `#read()` 方法的理解吧。

2.1 read

在 NioEventLoop 的 `#processSelectedKey(SelectionKey k, AbstractNioChannel ch)` 方法中，我们会看到这样一段代码：

```
// SelectionKey.OP_READ 或 SelectionKey.OP_ACCEPT 就绪
// readyOps == 0 是对 JDK Bug 的处理，防止空的死循环
// Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead
// to a spin loop
if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
    unsafe.read();
}
```

- 当 `(readyOps & SelectionKey.OP_READ) != 0` 时，这就是 NioSocketChannel 所在的 EventLoop 的线程**轮询到**有新的数据写入。
- 然后，调用 `NioByteUnsafe#read()` 方法，读取新的写入数据。

文章目录

。代码如下：

```
g();
 true ，移除对 SelectionKey.OP_READ 事件的感兴趣。
{
        return;
    }
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    // 获得 RecvByteBufAllocator.Handle 对象
    final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();
    // 重置 RecvByteBufAllocator.Handle 对象
    allocHandle.reset(config);

    ByteBuf byteBuf = null;
    boolean close = false; // 是否关闭连接
    try {
        do {
            // 申请 ByteBuf 对象
            byteBuf = allocHandle.allocate(allocator);
            // 读取数据
            // 设置最后读取字节数
            allocHandle.lastBytesRead(doReadBytes(byteBuf));
            // <1> 未读取到数据
            if (allocHandle.lastBytesRead() <= 0) {
                // 释放 ByteBuf 对象
                // nothing was read. release the buffer.
                byteBuf.release();
                // 置空 ByteBuf 对象
                byteBuf = null;
                // 如果最后读取的字节为小于 0 ，说明对端已经关闭
                close = allocHandle.lastBytesRead() < 0;
                // TODO
                if (close) {
```

```
37:                     // There is nothing left to read as we received an EOF.
38:                     readPending = false;
39:                 }
40:                 // 结束循环
41:                 break;
42:             }
43:
44:             // <2> 读取到数据
45:
46:             // 读取消息数量 + localRead
47:             allocHandle.incMessagesRead(1);
48:             // TODO 芋艿 readPending
49:             readPending = false;
50:             // 触发 Channel read 事件到 pipeline 中。 TODO
51:             pipeline.fireChannelRead(byteBuf);
52:             // 置空 ByteBuf 对象
```



```
63:             closeOnRead(pipeline);
64:         }
65:     } catch (Throwable t) {
66:         handleReadException(pipeline, byteBuf, t, close, allocHandle);
67:     } finally {
68:         // TODO 芋艿 readPending
69:         // Check if there is a readPending which was not processed yet.
70:         // This could be for two reasons:
71:         // * The user called Channel.read() or ChannelHandlerContext.read() in channelRead(...) me
72:         // * The user called Channel.read() or ChannelHandlerContext.read() in channelReadComplete
73:         //
74:         // See https://github.com/netty/netty/issues/2254
75:         if (!readPending && !config.isAutoRead()) {
76:             removeReadOp();
77:         }
78:     }
79: }
```

- 第 5 至 9 行：若 inputClosedSeenErrorOnRead = true ，移除对 SelectionKey.OP_READ 事件的感兴趣。详细解析，见《精尽 Netty 源码解析 —— Channel（七）之 close 操作》 的 「5. 服务端处理客户端主动关闭连接」 小节。
- 第 12 至 15 行：获得 RecvByteBufAllocator.Handle 对象，并重置它。这里的逻辑，和 `NioMessageUnsafe#read()` 方法的【第 14 至 17 行】的代码是一致的。相关的解析，见《精尽 Netty 源码解析 —— Channel（二）之 accept 操作》 。
- 第 20 至 64 行：**while 循环** 读取新的写入数据。
  - 第 22 行：调用 `RecvByteBufAllocator.Handle#allocate(ByteBufAllocator allocator)` 方法，申请 ByteBuf 对象。关于它的内容，我们放在 ByteBuf 相关的文章，详细解析。

- 第 25 行：调用 `AbstractNioByteChannel#doReadBytes(ByteBuf buf)` 方法，读取数据。详细解析，胖友先跳到 「3. AbstractNioMessageChannel#doReadMessages」 中，看完记得回到此处。
- 第 25 行：调用 `RecvByteBufAllocator.Handle#lastBytesRead(int bytes)` 方法，设置**最后**读取字节数。代码如下：

```
// AdaptiveRecvByteBufAllocator.HandleImpl.java
@Override
public void lastBytesRead(int bytes) {
    // If we read as much as we asked for we should check if we need to ramp up the size of o
    // This helps adjust more quickly when large amounts of data is pending and can avoid goi
    // the selector to check for more data. Going back to the selector can add significant la
    // data transfers.
    if (bytes == attemptedBytesRead()) {
        record(bytes);
    }
    super.lastBytesRead(bytes);
}
```


  - 代码比较多，我们只看重点，当然也不细讲。
  - 在 `<1>` 处，设置最后一次读取字节数。
- 读取有，有两种结果，**是**/**否**读取到数据。
- `<1>` **未**读取到数据，即 `allocHandle.lastBytesRead() <= 0` 。
- 第 30 行：调用 `ByteBuf#release()` 方法，释放 ByteBuf 对象。
  - 第 32 行：置空 ByteBuf 对象。
- 第 34 行：如果最后读取的字节为小于 0 ，说明对端已经关闭。
- 第 35 至 39 行：TODO 芋艿 细节
- 第 41 行： `break` 结束循环。
- `<2>` **有**读取到数据，即 `allocHandle.lastBytesRead() > 0` 。
- 第 47 行：调用 `AdaptiveRecvByteBufAllocator.HandleImpl#incMessagesRead(int amt)` 方法，读取消息(客户端)数量 + `localRead = 1` 。
- 第 49 行：TODO 芋艿 readPending
- 第 51 行：调用 `ChannelPipeline#fireChannelRead(Object msg)` 方法，触发 Channel read 事件到 pipeline 中。
  - **注意**，一般情况下，我们会在自己的 Netty 应用程序中，自定义 ChannelHandler 处理读取到的数据。😈 当然，此时读取的数据，大多数情况下是需要在解码(Decode)。关于这一块，在后续关于 Codec (编解码)的文章中，详细解析。
  - 如果没有自定义 ChannelHandler 进行处理，最终会被 pipeline 中的尾节点 TailContext 所处理。代码如下：

```
// TailContext.java
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    onUnhandledInboundMessage(msg);
```

```
    }

    // DefaultChannelPipeline.java
    protected void onUnhandledInboundMessage(Object msg) {
        try {
            logger.debug("Discarded inbound message {} that reached at the tail of the pipeli
        } finally {
            ReferenceCountUtil.release(msg);
        }
    }
```

- 最终也会**释放** ByteBuf 对象。这就是为什么【第 53 行】的代码，只去置空 ByteBuf 对象，而不用再去释放的原因。

- 第 53 行：置空 ByteBuf 对象。
- 第 54 行：调用 `AdaptiveRecvByteBufAllocator.HandleImpl#incMessagesRead(int` 循环是否继续，读取新的数据。代码如下：


```
    ator.MaxMessageHandle.java
    r defaultMaybeMoreSupplier = new UncheckedBooleanSupplie


    lastBytesRead; // 最后读取的字节数，是否等于，最大可写入的字

    @Override
    public boolean continueReading() {
        return continueReading(defaultMaybeMoreSupplier);
    }

    @Override
    public boolean continueReading(UncheckedBooleanSupplier maybeMoreDataSupplier) {
        return config.isAutoRead() &&
                (!respectMaybeMoreData || maybeMoreDataSupplier.get()) && // <1>
                totalMessages < maxMessagePerRead &&
                totalBytesRead > 0;
    }
```

- 一般情况下，最后读取的字节数，**不等于**最大可写入的字节数，即 `<1>` 处的代码 `UncheckedBooleanSupplier#get()` 返回 `false` ，则不再进行数据读取。因为 😈 也没有数据可以读取啦。

- 第 57 行：调用 `RecvByteBufAllocator.Handle#readComplete()` 方法，读取完成。暂无重要的逻辑，不详细解析。
- 第 59 行：调用 `ChannelPipeline#fireChannelReadComplete()` 方法，触发 Channel readComplete 事件到 pipeline 中。
  - *如果有需要，胖友可以自定义处理器，处理该事件。一般情况下，不需要。*
  - 如果没有自定义 ChannelHandler 进行处理，最终会被 pipeline 中的尾节点 TailContext 所处理。代码如下：

```
    // TailContext.java
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
```

```
        onUnhandledInboundChannelReadComplete();
    }

    // DefaultChannelPipeline.java
    protected void onUnhandledInboundChannelReadComplete() {
    }
```

- 具体的调用是**空方法**。

- 第 61 至 64 行：关闭客户端的连接。详细解析，见《精尽 Netty 源码解析 —— Channel（七）之 close 操作》的「5. 服务端处理客户端主动关闭连接」小节。
- 第 65 至 66 行：当发生异常时，调用 `#handleReadException(hannelPipeline pipeline, ByteBuf byteBuf, Throwable cause, boolean close, RecvByteBufAllocator.Handle allocHandle)` 方法，处理异常。详细解析，见「2.2 handleReadException」中。
- 第 67 至 78 行：TODO 芋艿 细节

2.2 handleReadException

`ine, ByteBuf byteBuf, Throwable cause, boolean close,` 方法，处理异常。代码如下：

```
elPipeline pipeline, ByteBuf byteBuf, Throwable cause, boole
 6:             // 触发 Channel read 事件到 pipeline 中。
 7:             pipeline.fireChannelRead(byteBuf);
 8:         } else {
 9:             // 释放 ByteBuf 对象
10:             byteBuf.release();
11:         }
12:     }
13:     // 读取完成
14:     allocHandle.readComplete();
15:     // 触发 Channel readComplete 事件到 pipeline 中。
16:     pipeline.fireChannelReadComplete();
17:     // 触发 exceptionCaught 事件到 pipeline 中。
18:     pipeline.fireExceptionCaught(cause);
19:     // // TODO 芋艿 细节
20:     if (close || cause instanceof IOException) {
21:         closeOnRead(pipeline);
22:     }
23: }
```

- 第 2 行： `byteBuf` 非空，说明在发生异常之前，至少申请 ByteBuf 对象是**成功**的。
  - 第 3 行：调用 `ByteBuf#isReadable()` 方法，判断 ByteBuf 对象是否可读，即剩余可读的字节数据。
    - 该方法的英文注释如下：

```
/**
 * Returns {@code true}
 * if and only if {@code (this.writerIndex - this.readerIndex)} is greater
```

```
     * than {@code 0}.
     */
    public abstract boolean isReadable();
```

  - 即 `this.writerIndex - this.readerIndex > 0` 。
- 第 5 行：TODO 芋艿 细节
- 第 7 行：调用 `ChannelPipeline#fireChannelRead(Object msg)` 方法，触发 Channel read 事件到 pipeline 中。
- 第 8 至 11 行：ByteBuf 对象不可读，所以调用 `ByteBuf#release()` 方法，释放 ByteBuf 对象。

- 第 14 行：调用 `RecvByteBufAllocator.Handle#readComplete()` 方法，读取完成。暂无重要的逻辑，不详细解析。
- 第 16 行：调用 `ChannelPipeline#fireChannelReadComplete()` 方法，触发 Channel readComplete 事件到 pipeline 中。
- 第 18 行：调用 `ChannelPipeline#fireExceptionCaught(Throwable)` 方法，触发 exceptionCaught 事件到 pipeline 中。

程序中，自定义 ChannelHandler 处理异常。

终会被 pipeline 中的尾节点 TailContext 所处理。代码如下：

```
ndlerContext ctx, Throwable cause) throws Exception {
);
```

```
// DefaultChannelPipeline.java
protected void onUnhandledInboundException(Throwable cause) {
    try {
        logger.warn("An exceptionCaught() event was fired, and it reached at the tail of the
                    "It usually means the last handler in the pipeline did not handle the
                cause);
    } finally {
        ReferenceCountUtil.release(cause);
    }
}
```

  - 打印**告警**日志。
  - 调用 `ReferenceCountUtil#release(Object msg)` 方法，释放和异常相关的资源。
- 第 19 至 22 行：TODO 芋艿，细节

2.3 closeOnRead

TODO 芋艿，细节

3. AbstractNioByteChannel#doReadBytes

`doReadBytes(ByteBuf buf)` **抽象**方法，读取写入的数据到方法参数 `buf` 中。它是一个**抽象**方法，定义在 AbstractNioByteChannel 抽象类中。代码如下：

```
/**
 * Read bytes into the given {@link ByteBuf} and return the amount.
```

```
     */
    protected abstract int doReadBytes(ByteBuf buf) throws Exception;
```

- 返回值为读取到的字节数。
- **当返回值小于 0 时，表示对端已经关闭。**

NioSocketChannel 对该方法的实现代码如下：

```
1: @Override
2: protected int doReadBytes(ByteBuf byteBuf) throws Exception {
3:     // 获得 RecvByteBufAllocator.Handle 对象
4:     final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();
5:     // 设置最大可读取字节数量。因为 ByteBuf 目前最大写入的大小为 byteBuf.writableBytes()
6:     allocHandle.attemptedBytesRead(byteBuf.writableBytes());
7:     // 读取数据到 ByteBuf 中
                                                    nel(), allocHandle.attemptedBytesRead());
```


Buf 对象**目前**最大可写入的大小为 `ByteBuf#writableBytes()`

ngByteChannel in, int length) 方法，读取数据到 ByteBuf 对象
oledUnsafeDirectByteBuf 举例子。代码如下：

```
// AbstractByteBuf.java
@Override
public int writeBytes(ScatteringByteChannel in, int length) throws IOException {
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length); // <1>
    if (writtenBytes > 0) { // <3>
        writerIndex += writtenBytes;
    }
    return writtenBytes;
}

// PooledUnsafeDirectByteBuf.java
@Override
public int setBytes(int index, ScatteringByteChannel in, int length) throws IOException {
    checkIndex(index, length);
    ByteBuffer tmpBuf = internalNioBuffer();
    index = idx(index);
    tmpBuf.clear().position(index).limit(index + length);
    try {
        return in.read(tmpBuf); // <2>
    } catch (ClosedChannelException ignored) {
        return -1;
    }
}
```

- 代码比较多，我们只看重点，当然也不细讲。还是那句话，关于 ByteBuf 的内容，我们在 ByteBuf 相关的文章详细解析。
- 在 `<1>` 处，会调用 `#setBytes(int index, ScatteringByteChannel in, int length)` 方法。

- 在 `<2>` 处，会调用 Java NIO 的 `ScatteringByteChannel#read(ByteBuffer)` 方法，读取**数据**到临时的 Java NIO ByteBuffer 中。
  - 在对端未断开时，返回的是读取数据的**字节数**。
  - 在对端已断开时，返回 `-1` ，表示断开。这也是为什么 `<3>` 处做了 `writtenBytes > 0` 的判断的原因。

666. 彩蛋

推荐阅读文章：

- 闪电侠 《深入浅出 Netty read》
- Hypercube 《自顶向下深入分析Netty（六）– Channel源码实现》

文章目录