- NETTY

精尽 Netty 源码解析 —— ChannelHandler（三）之 SimpleChannelInboundHandler

1. 概述

在本文，我们来分享 SimpleChannelInboundHandler 处理器。考虑到 Simple**UserEvent**ChannelHandler 和 SimpleChannelInboundHandler 的实现基本一致，所以也会在本文中分享。

如果胖友对 SimpleChannelInboundHandler 的使用不了解，请先看下《一起学Netty（三）之 SimpleChannelInboundHandler》，嘿嘿。

2. SimpleChannelInboundHandler

文章目录

dHandler ，继承 ChannelInboundHandlerAdapter 类，抽象类，处理**指定类**
ChannelInboundHandler 后，实现对**指定类型**的消息的自定义处理。

```
boundHandler<I> extends ChannelInboundHandlerAdapter {

     */
    private final TypeParameterMatcher matcher;
    /**
     * 使用完消息，是否自动释放
     *
     * @see #channelRead(ChannelHandlerContext, Object)
     */
    private final boolean autoRelease;

    /**
     * see {@link #SimpleChannelInboundHandler(boolean)} with {@code true} as boolean parameter.
     */
    protected SimpleChannelInboundHandler() {
        this(true);
    }

    /**
     * Create a new instance which will try to detect the types to match out of the type parameter of
     *
```

```
     * @param autoRelease   {@code true} if handled messages should be released automatically by passi
     *                      {@link ReferenceCountUtil#release(Object)}.
     */
    protected SimpleChannelInboundHandler(boolean autoRelease) {
        // <1> 获得 matcher
        matcher = TypeParameterMatcher.find(this, SimpleChannelInboundHandler.class, "I");
        this.autoRelease = autoRelease;
    }

    /**
     * see {@link #SimpleChannelInboundHandler(Class, boolean)} with {@code true} as boolean value.
     */
    protected SimpleChannelInboundHandler(Class<? extends I> inboundMessageType) {
        this(inboundMessageType, true);
    }

    /**
     * Create a new instance
     *
     * @param inboundMessageType    The type of messages to match
     * @param autoRelease           {@code true} if handled messages should be released automatically
     *                              {@link ReferenceCountUtil#release(Object)}.
     */
    protected SimpleChannelInboundHandler(Class<? extends I> inboundMessageType, boolean autoRelease)
        // <2> 获得 matcher
        matcher = TypeParameterMatcher.get(inboundMessageType);
        e;
```

文章目录

应的 TypeParameterMatcher 类型匹配器。

- <2> 处，使用 `inboundMessageType` 参数对应的 TypeParameterMatcher 类型匹配器。
- 在大多数情况下，我们不太需要特别详细的了解 `io.netty.util.internal.TypeParameterMatcher` 的代码实现，感兴趣的胖友可以自己看看《netty 简单Inbound通道处理器（SimpleChannelInboundHandler）》的「TypeParameterMatcher」部分。

- `autoRelease` 属性，使用完消息，是否自动释放。

2.2 acceptInboundMessage

`#acceptInboundMessage(Object msg)` 方法，判断消息是否匹配。代码如下：

```
/**
 * Returns {@code true} if the given message should be handled. If {@code false} it will be passed to
 * {@link ChannelInboundHandler} in the {@link ChannelPipeline}.
 */
public boolean acceptInboundMessage(Object msg) {
    return matcher.match(msg);
}
```

一般情况下， `matcher` 的类型是 ReflectiveMatcher(它是 TypeParameterMatcher 的内部类)。代码如下：

```
private static final class ReflectiveMatcher extends TypeParameterMatcher {

    /**
     * 类型
     */
    private final Class<?> type;

    ReflectiveMatcher(Class<?> type) {
        this.type = type;
    }

    @Override
    public boolean match(Object msg) {
        return type.isInstance(msg); // <1>
    }

}
```

- 匹配逻辑，看 `<1>` 处，使用 `Class#isInstance(Object obj)` 方法。对于这个方法，如果我们定义的 `I` 泛型是个父类，那可以匹配所有的子类。例如 `I` 设置为 Object 类，那么所有消息，都可以被匹配列。

2.3 channelRead


```
ndlerContext ctx, Object msg) throws Exception {

msg)) {
checked")
10:             // 处理消息
11:             channelRead0(ctx, imsg);
12:         } else {
13:             // 不需要释放消息
14:             release = false;
15:             // 触发 Channel Read 到下一个节点
16:             ctx.fireChannelRead(msg);
17:         }
18:     } finally {
19:         // 判断，是否要释放消息
20:         if (autoRelease && release) {
21:             ReferenceCountUtil.release(msg);
22:         }
23:     }
24: }
```

- 第 4 行： `release` 属性，是否需要释放消息。
- 第 7 行：调用 `#acceptInboundMessage(Object msg)` 方法，判断是否为匹配的消息。

- ① **匹配**，调用 `#channelRead0(ChannelHandlerContext ctx, I msg)` **抽象**方法，处理消息。代码如下：

```
/**
 * <strong>Please keep in mind that this method will be renamed to
 * {@code messageReceived(ChannelHandlerContext, I)} in 5.0.</strong>
 *
 * Is called for each message of type {@link I}.
 *
 * @param ctx           the {@link ChannelHandlerContext} which this {@link SimpleChannelInbo
 *                      belongs to
 * @param msg           the message to handle
 * @throws Exception    is thrown if an error occurred
 */
protected abstract void channelRead0(ChannelHandlerContext ctx, I msg) throws Exception;
```

  - 子类实现 SimpleChannelInboundHandler 类后，实现该方法，就能很方便的处理消息。
- ② **不匹配**，标记不需要释放消息，并触发 Channel Read 到**下一个节点**。

- 第 18 至 23 行：通过 `release` 变量 + `autoRelease` 属性，判断是否需要释放消息。若需要，调用 `ReferenceCountUtil#release(Object msg)` 方法，释放消息。😈 还是蛮方便的。

3. SimpleUserEventChannelHandler

`io.netty.channel.SimpleUserEventChannelHandler` ，继承 ChannelInboundHandlerAdapter 类，抽象类，处理**指定事件**的消息。

文章目录

nannelInboundHandler 基本一致，差别在于将指定类型的消息，改成了制定类

```
ChannelHandler<I> extends ChannelInboundHandlerAdapter {

    private final TypeParameterMatcher matcher;
    /**
     * 使用完消息，是否自动释放
     *
     * @see #channelRead(ChannelHandlerContext, Object)
     */
    private final boolean autoRelease;

    /**
     * see {@link #SimpleUserEventChannelHandler(boolean)} with {@code true} as boolean parameter.
     */
    protected SimpleUserEventChannelHandler() {
        this(true);
    }

    /**
     * Create a new instance which will try to detect the types to match out of the type parameter of
     *
```

```
     * @param autoRelease   {@code true} if handled events should be released automatically by passing
     *                      {@link ReferenceCountUtil#release(Object)}.
     */
    protected SimpleUserEventChannelHandler(boolean autoRelease) {
        matcher = TypeParameterMatcher.find(this, SimpleUserEventChannelHandler.class, "I");
        this.autoRelease = autoRelease;
    }

    /**
     * see {@link #SimpleUserEventChannelHandler(Class, boolean)} with {@code true} as boolean value.
     */
    protected SimpleUserEventChannelHandler(Class<? extends I> eventType) {
        this(eventType, true);
    }

    /**
     * Create a new instance
     *
     * @param eventType      The type of events to match
     * @param autoRelease    {@code true} if handled events should be released automatically by passin
     *                       {@link ReferenceCountUtil#release(Object)}.
     */
    protected SimpleUserEventChannelHandler(Class<? extends I> eventType, boolean autoRelease) {
        matcher = TypeParameterMatcher.get(eventType);
        this.autoRelease = autoRelease;
    }
```

文章目录


```
ven user event should be handled. If {@code false} it will be pas
in the {@link ChannelPipeline}.

ect evt) throws Exception {
```

```
    public final void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
        // 是否要释放消息
        boolean release = true;
        try {
            // 判断是否为匹配的消息
            if (acceptEvent(evt)) {
                @SuppressWarnings("unchecked")
                I ievt = (I) evt;
                // 处理消息
                eventReceived(ctx, ievt);
            } else {
                // 不需要释放消息
                release = false;
                // 触发 Channel Read 到下一个节点
                ctx.fireUserEventTriggered(evt);
            }
        } finally {
            // 判断，是否要释放消息
```

```
            if (autoRelease && release) {
                ReferenceCountUtil.release(evt);
            }
        }
    }

    /**
     * Is called for each user event triggered of type {@link I}.
     *
     * @param ctx the {@link ChannelHandlerContext} which this {@link SimpleUserEventChannelHandler} b
     * @param evt the user event to handle
     *
     * @throws Exception is thrown if an error occurred
     */
    protected abstract void eventReceived(ChannelHandlerContext ctx, I evt) throws Exception;

}
```

666. 彩蛋

木有彩蛋，hoho 。