

# 实战：使用 channelHandler 的热插拔实现客户端身份校验

在前面的小节中，细心的读者可能会注意到，客户端连上服务端之后，即使没有进行登录校验，服务端在收到消息之后仍然会进行消息的处理，这个逻辑其实是有问题的。本小节，我们来学习一下如何使用 pipeline 以及 handler 强大的热插拔机制实现客户端身份校验。

## 1. 身份检验

首先，我们在客户端登录成功之后，标记当前的 channel 的状态为已登录：

LoginRequestHandler.java

```
protected void channelRead0(ChannelHandlerContext ctx, LoginRequestPacket loginRequestPacket) {  
    if (valid(loginRequestPacket)) {  
        // ...  
        // 基于我们前面小节的代码，添加如下一行代码  
        LoginUtil.markAsLogin(ctx.channel());  
    }  
    // ...  
}
```

LoginUtil.java

```
public static void markAsLogin(Channel channel) {  
    channel.attr(Attributes.LOGIN).set(true);  
}
```

在登录成功之后，我们通过给 channel 打上属性标记的方式，标记这个 channel 已成功登录，那么，接下来，我们是不是需要在后续的每一种指令的处理前，都要来判断一下用户是否登录？

LoginUtil.java

```
public static boolean hasLogin(Channel channel) {  
    Attribute<Boolean> loginAttr =  
channel.attr(Attributes.LOGIN);  
  
    return loginAttr.get() != null;  
}
```

判断一个用户是否登录很简单，只需要调用一下 LoginUtil.hasLogin(channel) 即可，但是，Netty 的 pipeline 机制帮我们省去了重复添加同一段逻辑的烦恼，我们只需要在后续所有的指令处理 handler 之前插入一个用户认证 handle:

NettyServer.java

```
.childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel
ch) {
        ch.pipeline().addLast(new
PacketDecoder());
        ch.pipeline().addLast(new
LoginRequestHandler());
        // 新增加用户认证handler
        ch.pipeline().addLast(new AuthHandler());
        ch.pipeline().addLast(new
MessageRequestHandler());
        ch.pipeline().addLast(new
PacketEncoder());
    }
});
```

从上面代码可以看出，我们在 MessageRequestHandler 之前插入了一个 AuthHandler，因此 MessageRequestHandler 以及后续所有指令相关的 handler（后面小节会逐个添加）的处理都会经过 AuthHandler 的一层过滤，只要在 AuthHandler 里面处理掉身份认证相关的逻辑，后续所有的 handler 都不用操心身份认证这个逻辑，接下来我们来看一下 AuthHandler 的具体实现：

AuthHandler.java

```
public class AuthHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext
ctx, Object msg) throws Exception {
        if (!LoginUtil.hasLogin(ctx.channel())) {
            ctx.channel().close();
        } else {
            super.channelRead(ctx, msg);
        }
    }
}
```

1. AuthHandler 继承自 ChannelInboundHandlerAdapter，覆盖了 channelRead() 方法，表明他可以处理所有类型的数据
2. 在 channelRead() 方法里面，在决定是否把读到的数据传递到后续指令处理器之前，首先会判断是否登录成功，如果未登录，直接强制关闭连接（实际生产环境可能逻辑要复杂些，这里我们的重心在于学习 Netty，这里就粗暴些），否则，就把读到的数据向下传递，传递给后续指令处理器。

AuthHandler 的处理逻辑其实就是这么简单。但是，有的读者可能要问了，如果客户端已经登录成功了，那么在每次处理客户端数据之前，我们都要经历这么一段逻辑，比如，平均每次用户登录之后发送 100 次消息，其实剩余的 99 次身份校验逻辑都是没有必要的，因为只要连接未断开，客户端只要成功登录过，后续就不需要再进行客户端的身份校验。

这里我们为了演示，身份认证逻辑比较简单，实际生产环境中，身份认证的逻辑可能会更加复杂，我们需要寻找一种途径来避免资源与性能的浪费，使用 pipeline 的热插拔机制完全可以做到这一点。

## 2. 移除校验逻辑

对于 Netty 的设计来说，handler 其实可以看做是一段功能相对聚合的逻辑，然后通过 pipeline 把这些一个个小的逻辑聚合起来，串起一个功能完整的逻辑链。既然可以把逻辑串起来，也可以做到动态删除一个或多个逻辑。

在客户端校验通过之后，我们不再需要 AuthHandler 这段逻辑，而这一切只需要一行代码即可实现：

```
AuthHandler.java
```

```

public class AuthHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext
ctx, Object msg) throws Exception {
        if (!LoginUtil.hasLogin(ctx.channel())) {
            ctx.channel().close();
        } else {
            // 一行代码实现逻辑的删除
            ctx.pipeline().remove(this);
            super.channelRead(ctx, msg);
        }
    }

    @Override
    public void
handlerRemoved(ChannelHandlerContext ctx) {
        if (LoginUtil.hasLogin(ctx.channel())) {
            System.out.println("当前连接登录验证完
毕，无需再次验证，AuthHandler 被移除");
        } else {
            System.out.println("无登录验证，强制关闭
连接!");
        }
    }
}

```

上面的代码中，判断如果已经经过权限认证，那么就直接调用 pipeline 的 remove() 方法删除自身，这里的 this 指的其实就是 AuthHandler 这个对象，删除之后，这条客户端连接的逻辑链中就不再有这段逻辑了。

另外，我们还覆盖了 `handlerRemoved()` 方法，主要用于后续的演示部分的内容，接下来，我们就来进行实际演示。

### 3. 身份校验演示

在演示之前，对于客户端侧的代码，我们先把客户端向服务端发送消息的逻辑中，每次都判断是否登录的逻辑去掉，这样我们就可以在客户端未登录的情况下向服务端发送消息

NettyClient.java

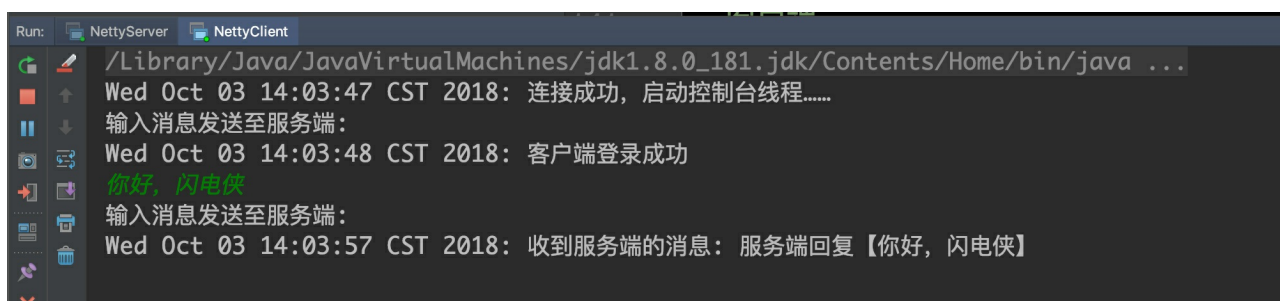
```
private static void
startConsoleThread(Channel channel) {
    new Thread(() -> {
        while (!Thread.interrupted()) {
            // 这里注释掉
            if
(LoginUtil.hasLogin(channel)) {
                System.out.println("输入消息发
送至服务端：");
                Scanner sc = new
Scanner(System.in);
                String line = sc.nextLine();

                channel.writeAndFlush(new
MessageRequestPacket(line));
            }
        }
    }).start();
}
```

## 3.1 有身份认证的演示

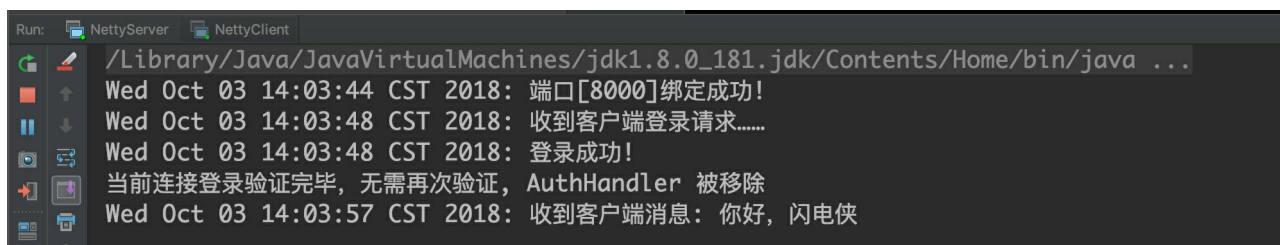
我们先启动服务端，再启动客户端，在客户端的控制台，我们输入消息发送至服务端，这个时候服务端与客户端控制台的输出分别为

客户端



```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:03:47 CST 2018: 连接成功, 启动控制台线程.....
输入消息发送至服务端:
Wed Oct 03 14:03:48 CST 2018: 客户端登录成功
你好, 闪电侠
输入消息发送至服务端:
Wed Oct 03 14:03:57 CST 2018: 收到服务端的消息: 服务端回复【你好, 闪电侠】
```

服务端



```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:03:44 CST 2018: 端口[8000]绑定成功!
Wed Oct 03 14:03:48 CST 2018: 收到客户端登录请求.....
Wed Oct 03 14:03:48 CST 2018: 登录成功!
当前连接登录验证完毕, 无需再次验证, AuthHandler 被移除
Wed Oct 03 14:03:57 CST 2018: 收到客户端消息: 你好, 闪电侠
```

观察服务端侧的控制台，我们可以看到，在客户端第一次发来消息的时候，AuthHandler 判断当前用户已通过身份认证，直接移除掉自身，移除掉之后，回调 handlerRemoved，这块内容也是属于上小节我们学习的 ChannelHandler 生命周期的一部分

## 3.2 无身份认证的演示

接下来，我们再来演示一下，客户端在未登录的情况下发送消息到服务端，我们到 LoginResponseHandler 中，删除发送登录指令的逻辑：



## LoginResponseHandler.java

```
public class LoginResponseHandler extends
SimpleChannelInboundHandler<LoginResponsePacket>
{

    @Override
    public void
channelActive(ChannelHandlerContext ctx) {
        // 创建登录对象
        LoginRequestPacket loginRequestPacket =
new LoginRequestPacket();

        loginRequestPacket.setUserId(UUID.randomUUID().to
String());
        loginRequestPacket.setUsername("flash");
        loginRequestPacket.setPassword("pwd");

        // 删除登录的逻辑
//
ctx.channel().writeAndFlush(loginRequestPacket);
    }

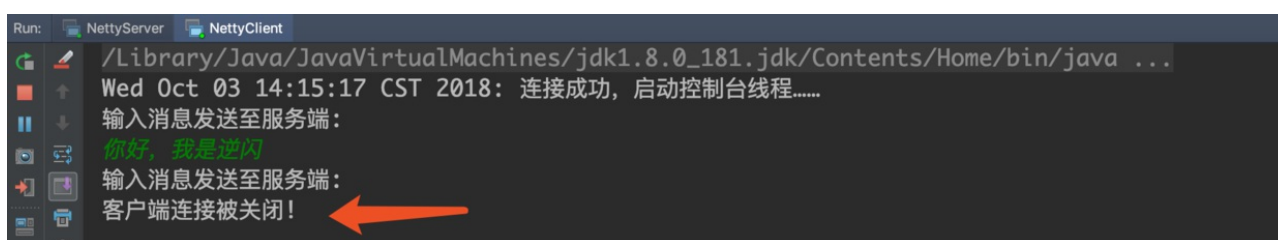
    @Override
    public void
channelInactive(ChannelHandlerContext ctx) {
        System.out.println("客户端连接被关闭!");
    }
}
```

我们把客户端向服务端写登录指令的逻辑进行删除，然后覆盖一下 channelInactive() 方法，用于验证客户端连接是否会被关闭。

接下来，我们先运行服务端，再运行客户端，并且在客户端的控制台输入文本之后发送给服务端

这个时候服务端与客户端控制台的输出分别为：

客户端

A screenshot of a Java IDE console window showing the output of the NettyClient. The output includes the Java path, a successful connection message at 14:15:17, a message '你好, 我是逆闪' sent to the server, and a final message '客户端连接被关闭!' (Client connection closed!) with a red arrow pointing to it.

```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:15:17 CST 2018: 连接成功, 启动控制台线程.....
输入消息发送至服务端:
你好, 我是逆闪
输入消息发送至服务端:
客户端连接被关闭!
```

服务端

A screenshot of a Java IDE console window showing the output of the NettyServer. The output includes the Java path, a successful binding message at 14:15:15, and a message '无登录验证, 强制关闭连接!' (No login verification, forced connection closure) with a red arrow pointing to it.

```
Run: NettyServer NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Wed Oct 03 14:15:15 CST 2018: 端口[8000]绑定成功!
无登录验证, 强制关闭连接!
```

由此看到，客户端如果第一个指令为非登录指令，AuthHandler 直接将客户端连接关闭，并且，从上小节，我们学到的有关 ChannelHandler 的生命周期相关的内容中也可以看到，服务端侧的 handlerRemoved() 方法和客户端侧代码的 channelInactive() 会被回调到。

关于 ChannelHandler 的热插拔机制相关的内容我们就暂且讲到这，最后，我们来对本小节内容做下总结。

## 总结

1. 如果有很多业务逻辑的 handler 都要进行某些相同的操作，我们完全可以抽取出一个 handler 来单独处理

2. 如果某一个独立的逻辑在执行几次之后（这里是一次）不需要再执行了，那么我们可以通过 ChannelHandler 的热插拔机制来实现动态删除逻辑，应用程序性能处理更为高效

## 思考

对于最后一部分的演示，对于客户端在登录情况下发送消息以及客户端在未登录情况下发送消息，AuthHandler 的其他回调方法分别是如何执行的，为什么？欢迎留言一起讨论。