

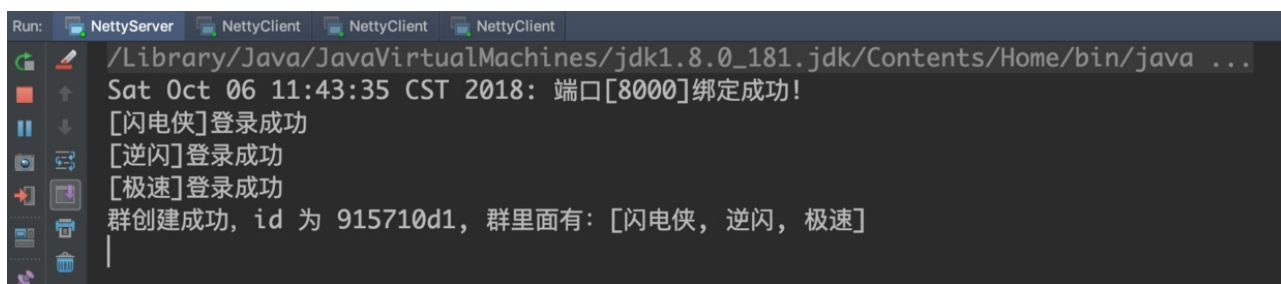
群聊消息的收发及 Netty 性能优化

通过前面小节的学习，相信大家看到本小节标题就已经知道该如何实现本小节的功能了吧，为了给大家学到更多的知识，在实现了群聊消息收发之后，本小节将给大家带来更多的惊喜。

开始实现之前，我们还是先来看一下群聊的最终效果。

1. 群聊消息最终效果

服务端



```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:35 CST 2018: 端口[8000]绑定成功!
[闪电侠]登录成功
[逆闪]登录成功
[极速]登录成功
群创建成功, id 为 915710d1, 群里面有: [闪电侠, 逆闪, 极速]
```

闪电侠，逆闪，极速先后登录，然后闪电侠拉逆闪，极速和自己加入群聊，下面，我们来看一下各位客户端的控制台界面

客户端 - 闪电侠

```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:38 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 闪电侠
[闪电侠]登录成功, userId 为: 60fae16e 创建群聊
createGroup
【拉入群聊】输入 userId 列表, userId 之间英文逗号隔开: 60fae16e,0b271dc1,b81f593f
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
sendToGroup 发送群消息
发送消息给某个群组: 915710d1 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩瞎扯, 整个宇宙我跑得最快!
```

闪电侠第一个输入 "sendToGroup" 发送群消息。

客户端 - 逆闪

```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:42 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 逆闪
[逆闪]登录成功, userId 为: 0b271dc1
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
sendToGroup
发送消息给某个群组: 915710d1 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩瞎扯, 整个宇宙我跑得最快!
```

逆闪第二个输入 "sendToGroup" 发送群消息, 在前面已经收到了闪电侠的消息。

客户端 - 极速

```
Run: NettyServer NettyClient NettyClient NettyClient
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Sat Oct 06 11:43:43 CST 2018: 连接成功, 启动控制台线程.....
输入用户名登录: 极速
[极速]登录成功, userId 为: b81f593f
群创建成功, id 为[915710d1], 群里面有: [闪电侠, 逆闪, 极速]
收到群[915710d1]中[60fae16e:闪电侠]发来的消息: 大家好, 我是闪电侠, 世界上跑得最快的男人!
收到群[915710d1]中[0b271dc1:逆闪]发来的消息: 大家好, 我是逆闪, 世界上跑得比闪电侠快一丢丢的男人!
sendToGroup
发送消息给某个群组: 915710d1 大家好, 我是极速, 别听他俩瞎扯, 整个宇宙我跑得最快!
收到群[915710d1]中[b81f593f:极速]发来的消息: 大家好, 我是极速, 别听他俩瞎扯, 整个宇宙我跑得最快!
```

逆闪最后一个输入 "sendToGroup" 发送消息，在前面已经收到了闪电侠和逆闪的消息。

1. 在闪电侠的控制台，输入 "sendToGroup" 指令之后，再输入 groupId + 空格 + 消息内容，发送消息给群里各位用户，随后，群组里的所有用户的控制台都显示了群消息。
2. 随后，陆续在逆闪和极速的控制台做做相同的操作，群组里的所有用户的控制台陆续展示了群消息。

这个实现过程和我们前面的套路一样，下面我们仅关注核心实现部分。

2. 群聊消息的收发的实现

核心实现部分其实就是服务端处理群消息的 handler – GroupMessageRequestHandler

GroupMessageRequestHandler.java

```

public class GroupMessageRequestHandler extends
SimpleChannelInboundHandler<GroupMessageRequestPa
cket> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
GroupMessageRequestPacket requestPacket) {
        // 1.拿到 groupId 构造群聊消息的响应
        String groupId =
requestPacket.getToGroupId();
        GroupMessageResponsePacket responsePacket
= new GroupMessageResponsePacket();
        responsePacket.setFromGroupId(groupId);

responsePacket.setMessage(requestPacket.getMessag
e());

responsePacket.setFromUser(SessionUtil.getSession
(ctx.channel()));

        // 2. 拿到群聊对应的 channelGroup, 写到每个客
        户端
        ChannelGroup channelGroup =
SessionUtil.getChannelGroup(groupId);

channelGroup.writeAndFlush(responsePacket);
    }
}

```

1. 首先, 通过 groupId 构造群聊响应 GroupMessageResponsePacket, 然后再把发送群聊的用户信息填入, 这里的用户信息我们就直接复用与 channel 绑定

的 session 了。

2. 然后，我们拿到对应群组的 ChannelGroup，通过 writeAndFlush() 写到客户端。

完整代码大家可以参考 [github](#)

(<https://github.com/lightningMan/flash-netty>) 对应本小节分支，下面进入我们本小节的几个重要知识点，可以拿小本本开始记了。

3. 共享 handler

在使用 Netty 完成了一个 IM 系统的核心功能之后，我们再来仔细看一下服务端

```
NettyServer.java
```

```

serverBootstrap
    .childHandler(new
ChannelInitializer<NioSocketChannel>() {
        protected void
initChannel(NioSocketChannel ch) {
            ch.pipeline().addLast(new
Splitter());
            ch.pipeline().addLast(new
PacketDecoder());
            ch.pipeline().addLast(new
LoginRequestHandler());
            ch.pipeline().addLast(new
AuthHandler());
            ch.pipeline().addLast(new
MessageRequestHandler());
            ch.pipeline().addLast(new
CreateGroupRequestHandler());
            ch.pipeline().addLast(new
JoinGroupRequestHandler());
            ch.pipeline().addLast(new
QuitGroupRequestHandler());
            ch.pipeline().addLast(new
ListGroupMembersRequestHandler());
            ch.pipeline().addLast(new
GroupMessageRequestHandler());
            ch.pipeline().addLast(new
LogoutRequestHandler());
            ch.pipeline().addLast(new
PacketEncoder());
        }
    });

```

1. 我们看到，服务端的 pipeline 链里面已经有 12 个 handler，

其中，与指令相关的 handler 有 9 个。

2. Netty 在这里的逻辑是：每次有新连接到来的时候，都会调用 ChannelInitializer 的 initChannel() 方法，然后这里 9 个指令相关的 handler 都会被 new 一次。
3. 我们可以看到，其实这里的每一个指令 handler，他们内部都是没有成员变量的，也就是说是无状态的，我们完全可以使用单例模式，即调用 pipeline().addLast() 方法的时候，都直接使用单例，不需要每次都 new，提高效率，也避免了创建很多小的对象。

比如，我们拿 LoginRequestHandler 举例，来看一下如何改造

LoginRequestHandler.java

```
// 1. 加上注解标识，表明该 handler 是可以多个 channel
共享的
@ChannelHandler.Sharable
public class LoginRequestHandler extends
SimpleChannelInboundHandler<LoginRequestPacket> {

    // 2. 构造单例
    public static final LoginRequestHandler
INSTANCE = new LoginRequestHandler();

    protected LoginRequestHandler() {
    }
}
```

1. 首先，非常重要的一点，如果一个 handler 要被多个 channel

进行共享，必须要加上 `@ChannelHandler.Sharable` 显示地告诉 Netty，这个 handler 是支持多个 channel 共享的，否则会报错，读者可以自行尝试一下。

2. 然后，我们仿照 Netty 源码里面单例模式的写法，构造一个单例模式的类。

接着，我们在服务端的代理里面就可以这么写

NettyServer.java

```
serverBootstrap
    .childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void
initChannel(NioSocketChannel ch) {
        // ...单例模式，多个 channel 共享同一
        个 handler

ch.pipeline().addLast(LoginRequestHandler.INSTANCE);
        // ...
    }
});
```

这样的话，每来一次新的连接，添加 handler 的时候就不需要每次都 new 了，剩下的 8 个指令，读者可以自行尝试改造一下。

4. 压缩 handler – 合并编解码器

当我们改造完了之后，我们再来看一下服务端代码

NettyServer.java

```
serverBootstrap
    .childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void
initChannel(NioSocketChannel ch) {
        ch.pipeline().addLast(new
Splitter());
        ch.pipeline().addLast(new
PacketDecoder());

ch.pipeline().addLast(LoginRequestHandler.INSTANCE);

ch.pipeline().addLast(AuthHandler.INSTANCE);

ch.pipeline().addLast(MessageRequestHandler.INSTANCE);

ch.pipeline().addLast(CreateGroupRequestHandler.INSTANCE);

ch.pipeline().addLast(JoinGroupRequestHandler.INSTANCE);

ch.pipeline().addLast(QuitGroupRequestHandler.INSTANCE);

ch.pipeline().addLast(ListGroupMembersRequestHandler.INSTANCE);
```

```
ch.pipeline().addLast(GroupMessageRequestHandler.
INSTANCE);

ch.pipeline().addLast(LoginRequestHandler.INSTANCE);

ch.pipeline().addLast(new
PacketEncoder());
    }
});
```

pipeline 中第一个 handler - Splitter，我们是无法动它的，因为他内部实现是与每个 channel 有关，每个 Splitter 需要维持每个 channel 当前读到的数据，也就是说他是有状态的。而 PacketDecoder 与 PacketEncoder 我们是可以继续改造的，Netty 内部提供了一个类，叫做 MessageToMessageCodec，使用它可以让我们编解码操作放到一个类里面去实现，首先我们定义一个 PacketCodecHandler

```
PacketCodecHandler.java
```

```
@ChannelHandler.Sharable
public class PacketCodecHandler extends
    MessageToMessageCodec<ByteBuf, Packet> {
    public static final PacketCodecHandler
    INSTANCE = new PacketCodecHandler();

    private PacketCodecHandler() {

    }

    @Override
    protected void decode(ChannelHandlerContext
    ctx, ByteBuf byteBuf, List<Object> out) {

    out.add(PacketCodec.INSTANCE.decode(byteBuf));
    }

    @Override
    protected void encode(ChannelHandlerContext
    ctx, Packet packet, List<Object> out) {
        ByteBuf byteBuf =
    ctx.channel().alloc().ioBuffer();
        PacketCodec.INSTANCE.encode(byteBuf,
    packet);
        out.add(byteBuf);
    }
}
```

1. 首先，这里 PacketCodecHandler，他是一个无状态的 handler，因此，同样可以使用单例模式来实现。
2. 我们看到，我们需要实现 decode() 和 encode() 方法，decode 是将二进制数据 ByteBuf 转换为 java 对象 Packet，

而 encode 操作是一个相反的过程，在 encode() 方法里面，我们调用了 channel 的 内存分配器手工分配了 ByteBuf。

接着，PacketDecoder 和 PacketEncoder都可以删掉，我们的 server 端代码就成了如下的样子

```
serverBootstrap
    .childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void
initChannel(NioSocketChannel ch) {
        ch.pipeline().addLast(new
Splitter());

ch.pipeline().addLast(PacketCodecHandler.INSTANCE
);

ch.pipeline().addLast(LoginRequestHandler.INSTANCE
);

ch.pipeline().addLast(AuthHandler.INSTANCE);

ch.pipeline().addLast(MessageRequestHandler.INS
TANCE);

ch.pipeline().addLast(CreateGroupRequestHandler.I
NSTANCE);

ch.pipeline().addLast(JoinGroupRequestHandler.INS
TANCE);

ch.pipeline().addLast(QuitGroupRequestHandler.INS
TANCE);
```

```
ch.pipeline().addLast(ListGroupMembersRequestHandler.INSTANCE);

ch.pipeline().addLast(GroupMessageRequestHandler.INSTANCE);

ch.pipeline().addLast(LoginRequestHandler.INSTANCE);

        }
    });
```

可以看到，除了拆包器，所有的 handler 都写成了单例，当然，如果你的 handler 里有与 channel 相关成员变量，那就不要写成单例的，不过，其实所有的状态都可以绑定在 channel 的属性上，依然是可以改造成单例模式。

这里，我提一个问题，为什么 PacketCodecHandler 这个 handler 可以直接移到前面去，原来的 PacketEncoder 不是在最后吗？读者可以结合前面 handler 与 pipeline 相关的小节思考一下。

如果我们再仔细观察我们的服务端代码，发现，我们的 pipeline 链中，绝大部分都是与指令相关的 handler，我们把这些 handler 编排在一起，是为了逻辑简洁，但是随着指令相关的 handler 越来越多，handler 链越来越长，在事件传播过程中性能损耗会被逐渐放大，因为解码器解出来的每个 Packet 对象都要在每个 handler 上经过一遍，我们接下来来看一下如何缩短这个事件传播的路径。

5. 缩短事件传播路径

5.1 压缩 handler – 合并平行 handler

对我们这个应用程序来说，每次 decode 出来一个指令对象之后，其实只会在一个指令 handler 上进行处理，因此，我们其实可以把这么多的指令 handler 压缩为一个 handler，我们来看一下如何实现

我们定义一个 IMHandler，实现如下：

IMHandler.java

```
@ChannelHandler.Sharable
public class IMHandler extends
SimpleChannelInboundHandler<Packet> {
    public static final IMHandler INSTANCE = new
IMHandler();

    private Map<Byte,
SimpleChannelInboundHandler<? extends Packet>>
handlerMap;

    private IMHandler() {
        handlerMap = new HashMap<>();

        handlerMap.put(MESSAGE_REQUEST,
MessageRequestHandler.INSTANCE);
        handlerMap.put(CREATE_GROUP_REQUEST,
CreateGroupRequestHandler.INSTANCE);
        handlerMap.put(JOIN_GROUP_REQUEST,
JoinGroupRequestHandler.INSTANCE);
        handlerMap.put(QUIT_GROUP_REQUEST,
QuitGroupRequestHandler.INSTANCE);

        handlerMap.put(LIST_GROUP_MEMBERS_REQUEST,
```

```

ListGroupMembersRequestHandler.INSTANCE);
        handlerMap.put(GROUP_MESSAGE_REQUEST,
GroupMessageRequestHandler.INSTANCE);
        handlerMap.put(LOGOUT_REQUEST,
LogoutRequestHandler.INSTANCE);
    }

    @Override
    protected void
channelRead0(ChannelHandlerContext ctx, Packet
packet) throws Exception {

handlerMap.get(packet.getCommand()).channelRead(c
tx, packet);
    }
}

```

1. 首先，IMHandler 是无状态的，依然是可以写成一个单例模式的类。
2. 我们定义一个 map，存放指令到各个指令处理器的映射。
3. 每次回调到 IMHandler 的 channelRead0() 方法的时候，我们通过指令找到具体的 handler，然后调用指令 handler 的 channelRead，他内部会做指令类型转换，最终调用到每个指令 handler 的 channelRead0() 方法。

接下来，我们来看一下，如此压缩之后，我们的服务端代码

```
NettyServer.java
```

```
serverBootstrap
    .childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void
initChannel(NioSocketChannel ch) {
        ch.pipeline().addLast(new
Splitter());

ch.pipeline().addLast(PacketCodecHandler.INSTANCE
);

ch.pipeline().addLast(LoginRequestHandler.INSTANCE
);

ch.pipeline().addLast(AuthHandler.INSTANCE);

ch.pipeline().addLast(IMHandler.INSTANCE);
    }
});
```

可以看到，现在，我们服务端的代码已经变得很清爽了，所有的平行指令处理 handler，我们都压缩到了一个 IMHandler，并且 IMHandler 和指令 handler 均为单例模式，在单机十几万甚至几十万的连接情况下，性能能得到一定程度的提升，创建的对象也大大减少了。

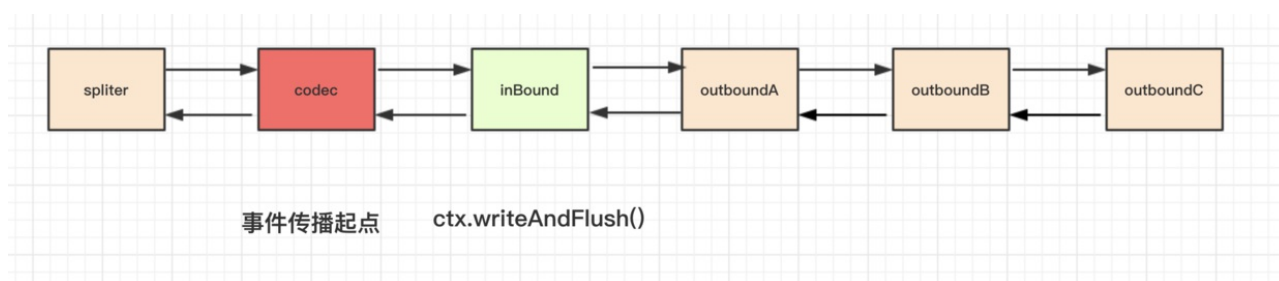
当然，如果你对性能要求没这么高，大可不必搞得这么复杂，还是按照我们前面小节的方式来实现即可，比如，我们的客户端多数情况下是单连接的，其实并不需要搞得如此复杂，还是保持原样即可。

5.2 更改事件传播源

另外，关于缩短事件传播路径，除了压缩 handler，还有一个就是，如果你的 outBound 类型的 handler 较多，在写数据的时候能用 `ctx.writeAndFlush()` 就用这个方法。

`ctx.writeAndFlush()` 事件传播路径

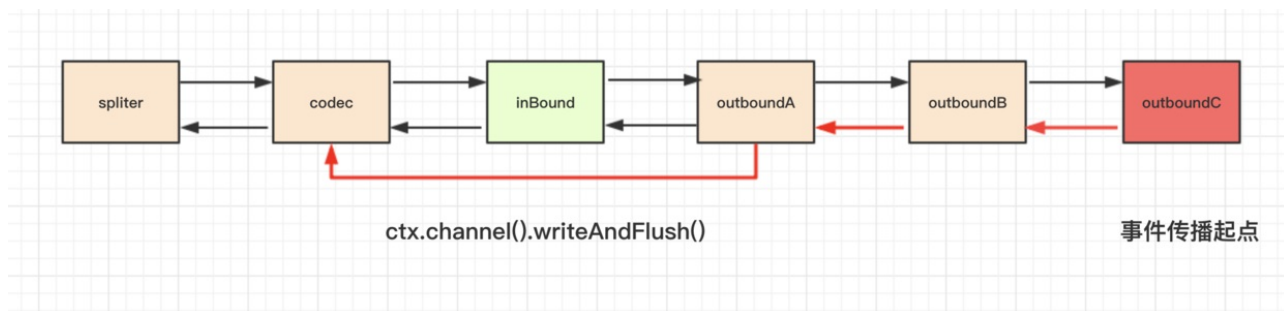
`ctx.writeAndFlush()` 是从 pipeline 链中的当前节点开始往前找到第一个 outBound 类型的 handler 把对象往前进行传播，如果这个对象确认不需要经过其他 outBound 类型的 handler 处理，就使用这个方法。



如上图，在某个 inBound 类型的 handler 处理完逻辑之后，调用 `ctx.writeAndFlush()` 可以直接一口气把对象送到 codec 中编码，然后写出去。

`ctx.channel().writeAndFlush()` 事件传播路径

`ctx.channel().writeAndFlush()` 是从 pipeline 链中的最后一个 outBound 类型的 handler 开始，把对象往前进行传播，如果你确认当前创建的对象需要经过后面的 outBound 类型的 handler，那么就调用此方法。



如上图，在某个 `inBound` 类型的 handler 处理完逻辑之后，调用 `ctx.channel().writeAndFlush()`，对象会从最后一个 `outBound` 类型的 handler 开始，逐个往前进行传播，路径是要比 `ctx.writeAndFlush()` 要长的。

由此可见，在我们的应用程序中，当我们没有改造编解码之前，我们必须调用 `ctx.channel().writeAndFlush()`，而经过改造之后，我们的编码器（既属于 `inBound`，又属于 `outBound` 类型的 handler）已处于 pipeline 的最前面，因此，可以大胆使用 `ctx.writeAndFlush()`。

6. 减少阻塞主线程的操作

这部分内容可能会引起部分读者不适，如果不能理解，记住结论即可。

通常我们的应用程序会涉及到数据库或者网络，比如以下这个例子

```
protected void channelRead0(ChannelHandlerContext
ctx, T packet) {
    // 1. balabala 一些逻辑
    // 2. 数据库或者网络等一些耗时的操作
    // 3. writeAndFlush()
    // 4. balabala 其他的逻辑
}
```

我们看到，在 `channelRead0()` 这个方法里面，第二个过程中，我们有一些耗时的操作，这个时候，我们万万不能将这个操作直接就在这个方法中处理了，为什么？

默认情况下，Netty 在启动的时候会开启 2 倍的 cpu 核数个 NIO 线程，而通常情况下我们单机会几万或者十几万的连接，因此，一条 NIO 线程会管理着几千或几万个连接，在传播事件的过程中，单条 NIO 线程的处理逻辑可以抽象成以下一个步骤，我们就拿 `channelRead0()` 举例

单个 NIO 线程执行的抽象逻辑

```
List<Channel> channelList = 已有数据可读的 channel
for (Channel channel in channelList) {
    for (ChannelHandler handler in
channel.pipeline()) {
        handler.channelRead0();
    }
}
```

从上面的抽象逻辑中可以看到，其中只要有一个 channel 的一个 handler 中的 `channelRead0()` 方法阻塞了 NIO 线程，最终都会拖慢绑定在该 NIO 线程上的其他所有的 channel，当然，这里抽象的逻辑已经做了简化，想了解细节可以参考我关于 Netty 中 NIO 线程（即 reactor 线程）文章的分析，

「[netty 源码分析之揭开 reactor 线程的面纱（一）](https://www.jianshu.com/p/0d0eece6d467)
(<https://www.jianshu.com/p/0d0eece6d467>)」，
「[netty 源码分析之揭开 reactor 线程的面纱（二）](https://www.jianshu.com/p/467a9b41833e)
(<https://www.jianshu.com/p/467a9b41833e>)」，
「[netty 源码分析之揭开 reactor 线程的面纱（二）](https://www.jianshu.com/p/58fad8e42379)
(<https://www.jianshu.com/p/58fad8e42379>)」

而我们需要怎么做？对于耗时的操作，我们需要把这些耗时的操作丢到我们的业务线程池中去处理，下面是解决方案的伪代码

```
ThreadPool threadPool = xxx;

protected void channelRead0(ChannelHandlerContext
ctx, T packet) {
    threadPool.submit(new Runnable() {
        // 1. balabala 一些逻辑
        // 2. 数据库或者网络等一些耗时的操作
        // 3. writeAndFlush()
        // 4. balabala 其他的逻辑
    })
}
```

这样，就可以避免一些耗时的操作影响 Netty 的 NIO 线程，从而影响其他的 channel。

7. 如何准确统计处理时长

我们接着前面的逻辑来讨论，通常，应用程序都有统计某个操作响应时间的需求，比如，基于我们上面的栗子，我们会这么做

```
protected void channelRead0(ChannelHandlerContext
ctx, T packet) {
    threadPool.submit(new Runnable() {
        long begin = System.currentTimeMillis();
        // 1. balabala 一些逻辑
        // 2. 数据库或者网络等一些耗时的操作
        // 3. writeAndFlush()
        // 4. balabala 其他的逻辑
        long time = System.currentTimeMillis() -
begin;
    })
}
```

这种做法其实是不推荐的，为什么？因为 `writeAndFlush()` 这个方法如果在非 NIO 线程（这里，我们其实是在业务线程中调用了该方法）中执行，它是一个异步的操作，调用之后，其实是会立即返回的，剩下的所有的操作，都是 Netty 内部有一个任务队列异步执行的，想了解底层细节的可以阅读一下我的这篇文章

[「netty 源码分析之 writeAndFlush 全解析」](https://www.jianshu.com/p/feaeaab2ce56)
(<https://www.jianshu.com/p/feaeaab2ce56>)

因此，这里的 `writeAndFlush()` 执行完毕之后，并不能代表相关的逻辑，比如事件传播、编码等逻辑执行完毕，只是表示 Netty 接收了这个任务，那么如何才能判断 `writeAndFlush()` 执行完毕呢？我们可以这么做

```
protected void channelRead0(ChannelHandlerContext
ctx, T packet) {
    threadPool.submit(new Runnable() {
        long begin = System.currentTimeMillis();
        // 1. balabala 一些逻辑
        // 2. 数据库或者网络等一些耗时的操作

        // 3. writeAndFlush
        xxx.writeAndFlush().addListener(future ->
{
            if (future.isDone()) {
                // 4. balabala 其他的逻辑
                long time =
System.currentTimeMillis() - begin;
            }
        });
    })
}
```

writeAndFlush() 方法会返回一个 ChannelFuture 对象，我们给这个对象添加一个监听器，然后在回调方法里面，我们可以监听这个方法执行的结果，进而再执行其他逻辑，最后统计耗时，这样统计出来的耗时才是最准确的。

最后，需要提出的一点就是，Netty 里面很多方法都是异步的操作，在业务线程中如果要统计这部分操作的时间，都需要使用监听器回调的方式来统计耗时，如果在 NIO 线程中调用，就不需要这么干。

8. 总结

这小节的知识点较多，每一个知识点都是我在线上千万级长连接应用摸索总结出来的实践经验，了解这些知识点会对你的线上应用有较大帮助，最后，我们来总结一下

1. 我们先在开头实现了群聊消息的最后一个部分：群聊消息的收发，这部分内容对大家来说已经非常平淡无奇了，因此没有贴完整的实现，重点在于实现完这最后一步接下来所做的改造和优化。
2. 所有指令都实现完之后，我们发现我们的 handler 已经非常臃肿庞大了，接下来，我们通过单例模式改造、编解码器合并、平行指令 handler 合并、慎重选择两种类型的 `writeAndFlush()` 的方式来压缩优化。
3. 在 handler 的处理中，如果有耗时的操作，我们需要把这些操作都丢到我们自定义的业务线程池中处理，因为 NIO 线程是会有很多 channel 共享的，我们不能阻塞他。
4. 对于统计耗时的场景，如果在自定义业务线程中调用类似 `writeAndFlush()` 的异步操作，需要通过添加监听器的方式来统计。

9. 思考

本文的思考题在文中已经穿插给出，欢迎留言讨论。