面向对象程序设计大作业

——源码阅读：Netty

宋嘉程 2018K8009937001

1.简单回顾

前文主要对transport模块进行了分析，着重关注了Bootstrap包与Channel包中的类及类间关系。下面我们将继续分析netty在这些部分涉及到的设计模式。

2.建造者模式

> **意图**：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。
>
> **主要解决**：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。
>
> **何时使用**：一些基本部件不会变，而其组合经常变化的时候。
>
> **如何解决**：将变与不变分离开。

建造者模式非常简单，通过链式调用来设置对象的属性，在对象属性繁多的场景下非常有用。建造者模式的优势就是可以像搭积木一样自由选择需要的属性，并不是强绑定的。对于使用者来说，必须清楚需要设置哪些属性，在不同场景下可能需要的属性也是不一样的。Netty 中 ServerBootStrap 和 Bootstrap 引导器是最经典的建造者模式实现，在构建过程中需要设置非常多的参数，例如配置线程池 EventLoopGroup、设置 Channel 类型、注册 ChannelHandler、设置 Channel 参数、端口绑定等。

3.工厂模式

> **意图**：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。
>
> **主要解决**：主要解决接口选择的问题。
>
> **何时使用**：我们明确地计划不同条件下创建不同实例时。
>
> **如何解决**：让其子类实现工厂接口，返回的也是一个抽象的产品。

Netty 在创建 Channel 的时候使用的就是工厂方法模式，因为服务端和客户端的 Channel 是不一样的。Netty 将反射和工厂方法模式结合在一起，只使用一个工厂类，然后根据传入的 Class 参数来构建出对应的 Channel，不需要再为每一种 Channel 类型创建一个工厂类。具体源码实现如下：

```
public class ReflectiveChannelFactory<T extends Channel> implements
ChannelFactory<T> {

    private final Constructor<? extends T> constructor;

    public ReflectiveChannelFactory(Class<? extends T> clazz) {
        ObjectUtil.checkNotNull(clazz, "clazz");
```

```
        try {
            this.constructor = clazz.getConstructor();
        } catch (NoSuchMethodException e) {
            throw new IllegalArgumentException("Class " +
StringUtil.simpleClassName(clazz) +
                    " does not have a public non-arg constructor", e);
        }
    }

    @Override
    public T newChannel() {
        try {
            return constructor.newInstance();
        } catch (Throwable t) {
            throw new ChannelException("Unable to create Channel from class " +
constructor.getDeclaringClass(), t);
        }
    }

    @Override
    public String toString() {
        return StringUtil.simpleClassName(ReflectiveChannelFactory.class) +
                '(' +
StringUtil.simpleClassName(constructor.getDeclaringClass()) + ".class)";
    }
}
```

虽然通过反射技术可以有效地减少工厂类的数据量，但是反射相比直接创建工厂类有性能损失，所以对于性能敏感的场景，应当谨慎使用反射。

4.责任链模式

> **意图：**避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。
>
> **主要解决：**职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。
>
> **何时使用：**在处理消息的时候以过滤很多道。
>
> **如何解决：**拦截的类都实现统一接口。

通过ChannelPipeline源码注释，可以清晰看到其责任链模式：

```
                                                  I/O Request
 *                                               via {@link Channel} or
 *                                           {@link ChannelHandlerContext}
 *                                                        |
 *  +---------------------------------------------------+---------------+
 *  |                           ChannelPipeline         |               |
 *  |                                                  \|/              |
 *  |    +---------------------+           +-----------+----------+    |
 *  |    | Inbound Handler  N  |           | Outbound Handler  1  |    |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |              /|\                                 |               |
 *  |               |                                 \|/              |
```

```
 *  |    +----------+----------+           +-----------+----------+    |
 *  |    | Inbound Handler N-1 |           | Outbound Handler  2  |    |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |              /|\                                 .               |
 *  |               .                                  .               |
 *  | ChannelHandlerContext.fireIN_EVT() ChannelHandlerContext.OUT_EVT()|
 *  |        [ method call]                       [method call]        |
 *  |               .                                  .               |
 *  |               .                                 \|/              |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |    | Inbound Handler  2  |           | Outbound Handler M-1 |    |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |              /|\                                 |               |
 *  |               |                                 \|/              |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |    | Inbound Handler  1  |           | Outbound Handler  M  |    |
 *  |    +----------+----------+           +-----------+----------+    |
 *  |              /|\                                 |               |
 *  +---------------+-----------------------------------+---------------+
 *                  |                                 \|/
 *  +---------------+-----------------------------------+---------------+
 *  |               |                                   |               |
 *  |       [ Socket.read() ]                    [ Socket.write() ]     |
 *  |                                                                   |
 *  |  Netty Internal I/O Threads (Transport Implementation)            |
 *  +-------------------------------------------------------------------+
```

ChannelHandlerContext：ChannelHandler的上下文，包含handler的执行环境信息，比如链表的前后节点的指针。

DefaultChannelPipeline是netty中ChannelPipeline接口的唯一实现类。在初始化HeadContext和TailContext时：

```
final AbstractChannelHandlerContext head;
final AbstractChannelHandlerContext tail;

tail = new TailContext(this);
head = new HeadContext(this);

head.next = tail;
tail.prev = head;
```

当有新的ChannelHandlerContext加入时，可以看到非常清晰的链表操作，代码如下：

```
private void addFirst0(AbstractChannelHandlerContext newCtx) {
    AbstractChannelHandlerContext nextCtx = head.next;
    newCtx.prev = head;
    newCtx.next = nextCtx;
    head.next = newCtx;
    nextCtx.prev = newCtx;
}
private void addLast0(AbstractChannelHandlerContext newCtx) {
    AbstractChannelHandlerContext prev = tail.prev;
    newCtx.prev = prev;
    newCtx.next = tail;
    prev.next = newCtx;
```

```
    tail.prev = newCtx;
}
```

5.小结

在netty中，使用大量的设计模式，常见的设计模式在 Netty 源码中都有所体现。在本节中，仅仅是以transport模块展示了其中的冰山一角。netty中还有许多优秀的设计等待着发掘与探索，希望能在以后的学习中继续钻研分析。