4

第 4 篇

Netty 实战篇

13

第 13 章

Netty 中的设计模式与性能调优

课程目标

1、了解设计模式在 Netty 中的应用。

2、了解实际应用中 Netty 的性能调优解决方案。

内容定位

1、已熟练掌握 Netty 操作 API 的人群。

2、需要基于 Netty 开发的人群。

13.1 设计模式在 Netty 中的应用

13.1.1 单例模式源码举例

单例模式要点回顾：

1、一个类在任何情况下只有一个对象，并提供一个全局访问点。

2、可延迟创建。

3、避免线程安全问题。

案例分析：

```
@Sharable
public final class MqttEncoder extends MessageToMessageEncoder<MqttMessage> {
    public static final MqttEncoder INSTANCE = new MqttEncoder();

    private MqttEncoder() {
    }

    protected void encode(ChannelHandlerContext ctx, MqttMessage msg, List<Object> out) throws Exception {
        out.add(doEncode(ctx.alloc(), msg));
    }
    ...
}
```

13.1.2 策略模式源码举例

策略模式要点回顾：

1、封装一系列可相互替换的算法家族。

2、动态选择某一个策略。

案例分析：

```
public final class DefaultEventExecutorChooserFactory implements EventExecutorChooserFactory {
    public static final DefaultEventExecutorChooserFactory INSTANCE = new DefaultEventExecutorChooserFactory();

    private DefaultEventExecutorChooserFactory() {
    }

    public EventExecutorChooser newChooser(EventExecutor[] executors) {
```

```
        return (EventExecutorChooser)(isPowerOfTwo(executors.length)?new
DefaultEventExecutorChooserFactory.PowerOfTowEventExecutorChooser(executors):new
DefaultEventExecutorChooserFactory.GenericEventExecutorChooser(executors));
    }

    private static boolean isPowerOfTwo(int val) {
        return (val & -val) == val;
    }
    ...
}
```

13.1.3 装饰者模式源码举例

装饰者模式要点回顾：

1、装饰者和被装饰者实现同一个接口。

2、装饰者通常继承被装饰者，同宗同源。

3、动态修改、重载被装饰者的方法。

WrappedByteBuf：

```
class WrappedByteBuf extends ByteBuf {
    protected final ByteBuf buf;

    protected WrappedByteBuf(ByteBuf buf) {
        if(buf == null) {
            throw new NullPointerException("buf");
        } else {
            this.buf = buf;
        }
}
...
}
```

UnreleasableByteBuf：

```
final class UnreleasableByteBuf extends WrappedByteBuf {
    private SwappedByteBuf swappedBuf;

    UnreleasableByteBuf(ByteBuf buf) {
        super(buf);
    }

    ...

    public boolean release() {
        return false;
    }

    public boolean release(int decrement) {
        return false;
```

```
    }
}
```

SimpleakAwareByteBuf:

```
final class SimpleLeakAwareByteBuf extends WrappedByteBuf {
    private final ResourceLeak leak;

    SimpleLeakAwareByteBuf(ByteBuf buf, ResourceLeak leak) {
        super(buf);
        this.leak = leak;
    }

     ...

    public boolean release() {
        boolean deallocated = super.release();
        if(deallocated) {
            this.leak.close();
        }

        return deallocated;
    }

    public boolean release(int decrement) {
        boolean deallocated = super.release(decrement);
        if(deallocated) {
            this.leak.close();
        }

        return deallocated;
    }
}
```

13.1.4 观察者模式源码举例

观察者模式要点回顾：

1、两个角色：观察者和被观察者。

2、观察者订阅消息，被观察者发布消息。

3、订阅则能收到消息，取消订阅则收不到。

channel.writeAndFlush()方法：

AbstractChannel:

```
public abstract class AbstractChannel extends DefaultAttributeMap implements Channel {

    ...

    public ChannelFuture writeAndFlush(Object msg) {
        return this.pipeline.writeAndFlush(msg);
    }
```

```
    public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {
        return this.pipeline.writeAndFlush(msg, promise);
    }
     ...
}
```

13.1.5 迭代器模式源码举例

迭代器模式要点回顾：

1. 实现迭代器接口

2. 实现对容器中的各个对象逐个访问的方法。

```
public class CompositeByteBuf extends AbstractReferenceCountedByteBuf implements Iterable<ByteBuf> {
protected byte _getByte(int index) {
        CompositeByteBuf.Component c = this.findComponent(index);
        return c.buf.getByte(index - c.offset);
}
...
}
```

13.1.6 责任链模式源码举例

责任链：是指多个对象都有机会处理同一个请求，从而避免请求的发送者和接收者之间的耦合关系。然后，将这些对象连成一条链，并且沿着这条链往下传递请求，直到有一个对象可以处理它为止。在每个对象处理过程中，每个对象只处理它自己关心的那一部分，不相关的可以继续往下传递，直到链中的某个对象不想处理，可以将请求终止或丢弃。

责任链模式要点回顾：

1、需要有一个顶层责任处理接口（ChannelHandler）。

2、需要有动态创建链、添加和删除责任处理器的接口（ChannelPipeline）。

3、需要有上下文机制（ChannelHandlerContext）。

4、需要有责任终止机制（不调用 ctx.fireXXX()方法,则终止传播）。

AbstractChannelHandlerContext:

```
abstract class AbstractChannelHandlerContext extends DefaultAttributeMap implements ChannelHandlerContext,
ResourceLeakHint {
    private AbstractChannelHandlerContext findContextInbound() {
        AbstractChannelHandlerContext ctx = this;
```

```
        do {
            ctx = ctx.next;
        } while(!ctx.inbound);

        return ctx;
}
}
```

13.1.7 工厂模式源码举例

工厂模式要点回顾：

1、将创建对象的逻辑封装起来。

ReflectiveChannelFactory：

```
public class ReflectiveChannelFactory<T extends Channel> implements ChannelFactory<T> {
    private final Class<? extends T> clazz;

    public ReflectiveChannelFactory(Class<? extends T> clazz) {
        if(clazz == null) {
            throw new NullPointerException("clazz");
        } else {
            this.clazz = clazz;
        }
    }

    public T newChannel() {
        try {
            return (Channel)this.clazz.newInstance();
        } catch (Throwable var2) {
            throw new ChannelException("Unable to create Channel from class " + this.clazz, var2);
        }
    }
}
```

13.2 Netty 高性能并发调优

13.2.1 Netty 应用程序性能调优

// 90.0% == 1ms 1000 100 > 1ms

// 95.0% == 10ms 1000 50 > 10ms

// 99.0% == 100ms 1000 10 > 100ms

// 99.9% == 1000ms 1000 1 > 1000ms

13.2.2 单机百万连接调优解决思路

如何模拟百万连接

单机 1024 及以下的端口只能给 ROOT 保留使用，客户端（1025~65535）

Server 8000

| 6W

Client 1025 ~ 65535

Server 8000 ~ 8020

| 20 * 6

Client 1025 ~ 65535

突破局部文件句柄限制

突破全局文件句柄限制

13.3 Netty 总结与面试答疑

1、Netty 定位：

A、作为开源码框架的底层框架（TCP 通信）

SpringBoot 内置的容器(Tomcat/Jerry)

Zookeper 数据交互

Dubbo 多协议 RPC 的支持

B、直接做服务器（消息推送服务，游戏后台）

2、Netty 如何确定要使用哪些编码器和解码器

很简单，看 API 文档

Netty 自带的编解码器可以解决 99%的业务需求

1%自己编解码

3、Netty 中大文件上传的那个 handler 是怎么做到防止内存撑爆的

ByteBuf 分片，

直接缓冲区，0 拷贝，提高内存的利用率

加内存

4、Tomcat NIO 方式的调优线程 ，本质上是对 netty 的调优吗

8.5 之后开始用 Netty?

5、责任链模式能否用在，一个操作出口参数为另一个操作的入口

执行顺序有关系，有先后

API 设计 callable(上一次调用的结果），msg（皮球）

6、Netty 里面 Pooled 缓冲区和 Unpooled 缓冲区内存分配（在录播中讲解）

7、Linux 底层 IO 模型，主从，多路复用的思想（录制一个基于硬件层面 IO 模型，显得更加专业）

8、Selector 客户端与服务端之间是什么关系?

客户端：CONNECT READ WRITE

服务端：ACCEPT READ WRITE

我的笔记有时候由于手误，如果发现错误，欢迎大家给我纠正，也许以后大家为 Tom 出 Netty 的书做贡献。