Java. Углубленное изучение

Урок 3

Фреймворк Netty

Гибкий и высокопроизводительный фреймворк для разработки сетевых приложений.

Фреймворк Netty

Netty — это NIO клиент-серверный фреймворк, позволяющий разрабатывать гибкие, высокопроизводительные и масштабируемые сетевые приложения.

Написание DISCARD-сервера

DISCARD — самый простой сетевой протокол, который принимает сообщения без ответа на них. Для его реализации достаточно игнорировать все получаемые данные. Написание сервера начинаем, реализуя обработчик событий из библиотеки Netty:

```
package io.netty.example.discard;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class DiscardServerHandler extends ChannelInboundHandlerAdapter { //(1)
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {     //(2)
        // Discard the received data silently.
        ((ByteBuf) msg).release();                                       //(3)
    }

    @Override
    // (4)
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // Close the connection when an exception is raised.
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. **DiscardServerHandler** наследуется от ChannelInboundHandlerAdapter, который реализует ChannelInboundHandler. ChannelInboundHandler предоставляет методы, обрабатывающие события получения и отправки сообщений.

2. Необходимо переопределить метод **channelRead()**, который вызывается при получении данных от клиента. В данном случае входящее сообщение попадает в объект класса **ByteBuf**.

3. Для реализации DISCARD-протокола обработчик должен игнорировать все входящие сообщения. ByteBuf — это буфер данных, который очищается вызовом метода **release()**. Необходимо учесть, что освобождение ресурсов — задача обработчика. Метод **channelRead()**, как правило, реализуется следующим образом:

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    try {
        // Обрабатываем входящее сообщение msg
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

4. Метод **exceptionCaught()** выполняется, когда появляются исключения, брошенные **Netty** из-за ошибок, возникших при чтении/записи данных или обработке событий **Handler**-ом. В большинстве случаев перехваченное исключение должно быть залогировано, а канал, в котором это произошло, закрыт. В других задачах подход может отличаться: например, перед закрытием соединения можно отправить клиенту код ошибки.

После написания обработчика событий необходимо реализовать сам DISCARD-сервер:

```
package io.netty.example.discard;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class DiscardServer {
    private int port;

    public DiscardServer(int port) {
        this.port = port;
    }

    public void run() throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(); // (1)
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap(); // (2)
            b.group(bossGroup, workerGroup)
             .channel(NioServerSocketChannel.class) // (3)
             .childHandler(new ChannelInitializer<SocketChannel>() { // (4)
                 @Override
                 public void initChannel(SocketChannel ch) throws Exception {
                     ch.pipeline().addLast(new DiscardServerHandler());
                 }
             })
             .option(ChannelOption.SO_BACKLOG, 128)          // (5)
```

```
             .childOption(ChannelOption.SO_KEEPALIVE, true); // (6)

            // Bind and start to accept incoming connections.
            ChannelFuture f = b.bind(port).sync(); // (7)

            // Wait until the server socket is closed.
            // In this example, this does not happen, but you can do that to
gracefully
            // shut down your server.
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        int port;
        if (args.length > 0) {
            port = Integer.parseInt(args[0]);
        } else {
            port = 8080;
        }
        new DiscardServer(port).run();
    }
}
```

1. NioEventLoopGroup — это пул потоков, обрабатывающий входящие и исходящие операции. Netty предлагает реализации EventLoopGroup для разных видов сетевых протоколов. Для текущей реализации сервера будет использоваться два пула NioEventLoopGroup. Первый, как правило, называется '**boss**' и занимается приемом входящих подключений. Второй — '**worker**' — обрабатывает потоки данных. Какое количество потоков создается и как они привязываются к каналам, зависит от реализации EventLoopGroup и настраивается через конструктор.

2. ServerBootstrap позволяет настроить сервер перед запуском.

3. Указываем использование класса NioServerSocketChannel для создания канала после того, как принято входящее соединение.

4. Указываем обработчик, который будем использовать для открытого канала (Channel). ChannelInitializer помогает пользователю сконфигурировать новый канал.

5. Можно настроить параметры канала. Для TCP/IP-сервера можно настроить такие опции, как **tcpNoDelay** и **keepAlive**.

6. **option()** применяются к NioServerSocketChannel, который принимает входящие подключения, а **childOption()** — для обрабатываемых каналов.

Простой DISCARD-сервер готов.

Обработка входящих сообщений

Для тестирования полученного сервера можно использовать **telnet**, подключившись на **localhost 8080**. Но учитывая принцип работы DISCARD-сервера, не будет понятно, действует он или нет, так как полностью игнорирует входящие данные. Необходимо модифицировать сервер так, чтобы он отображал получаемые сообщения. Для этого переопределим метод **channelRead()** в **DiscardServerHandler**.

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ByteBuf in = (ByteBuf) msg;
    try {
        while (in.isReadable()) {         // (1)
            System.out.print((char) in.readByte());
            System.out.flush();
        }
    } finally {
        ReferenceCountUtil.release(msg); // (2)
    }
}
```

1. Цикл чтения данных из буфера.
2. Можно использовать **in.release().**

Теперь при запуске **telnet** сервер будет видеть клиентские сообщения.

Написание эхо-сервера

Прошлые версии сервера только получали сообщения, но обычно сервер должен отвечать на входящие запросы. Для этого реализуем ECHO-протокол, в котором входящие данные отправляются обратно клиенту. Для этого модифицируем метод **channelRead()**:

```
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ctx.write(msg); // (1)
        ctx.flush();    // (2)
    }
```

1. ChannelHandlerContext-объект предоставляет возможные операции, которые можно выполнять при получении событий ввода/вывода. В данном случае используется **write(Object)** для отправки сообщений клиенту. Для входящего сообщения не освобождается буфер через **release()**, поскольку **Netty** делает **release()** при отправке сообщений в канал.
2. **ctx.write(Object)** — буферизированная запись в канал, поэтому для отправки используется **ctx.flush()**. Либо можно использовать метод **ctx.writeAndFlush(msg)**.

Написание TIME-сервера

Реализуем сервер для работы с TIME-протоколом. В отличие от прошлых примеров, сообщения будут содержать 32-битное целое число и канал закроется сразу после отправки сообщений.
Поскольку в данном варианте нет необходимости читать входящие сообщения, а отправлять исходящее надо сразу при установке соединения, метод **channelRead()** не нужен. Вместо него переопределим **channelActive()**:

```
package io.netty.example.time;

public class TimeServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(final ChannelHandlerContext ctx) { // (1)
        final ByteBuf time = ctx.alloc().buffer(4);              // (2)
        time.writeInt((int) (System.currentTimeMillis() / 1000L + 2208988800L));

        final ChannelFuture f = ctx.writeAndFlush(time);         // (3)
        f.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) {
                assert f == future;
                ctx.close();
            }
        });                                                      // (4)
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. Метод **channelActive()** вызывается при открытии соединения и готовности передавать данные. В данном случае при открытии соединения отправим 32-битное целое число, обозначающее время.

2. Для отправки сообщения необходимо выделить буфер, содержащий его. Поскольку мы передаем 32-битное число, емкость **ByteBuf** должна составлять 4 байта. Для этого получим ByteBufAllocator через **ChannelHandlerContext.alloc()** и выделим новый буфер.

3. Теперь запишем само сообщение.

Метод **java.nio.ByteBuffer.flip()** перед отправкой в NIO отсутствует? У **ByteBuf** нет такого метода, вместо него используется два указателя: для операций чтения и для записи. При записи в буфер индекс записывающего указателя увеличивается, в то время как читающий остается неизменным. Читающий и записывающий индексы указывают, где сообщение начинается и заканчивается.

В отличие от **Netty**, в **java.nio** нельзя узнать, где начало и конец сообщения, без вызова метода **flip()**. Могут возникнуть проблемы, если пропустить вызов **buffer.flip()**: либо будет отправлено не то, что ожидалось, либо ничего. В Netty такого произойти не может, поскольку для операций используются разные указатели. Это упрощает жизнь разработчику.

Важный момент: методы **ChannelHandlerContext.write()** (и **writeAndFlush()**) возвращают ChannelFuture, который представляет еще не выполненную **I/O**-операцию. Это означает, что любая запрошенная операция может быть еще не выполнена, поскольку все операции в Netty — асинхронные. Например, в приведенном ниже коде соединение может быть закрыто до отправки сообщения:

```
Channel ch = ...;
ch.writeAndFlush(message);
ch.close();
```

Метод **close()** нужно вызвать только после того, как будет выполнен ChannelFuture, который возвращается из метода **write()** и сообщает, что все операции записи завершены. **Close()** тоже не моментально закрывает соединение и возвращает ChannelFuture.

Чтобы узнать, что операция завершена, надо добавить ChannelFutureListener к возвращенному **ChannelFuture**. Создается анонимный ChannelFutureListener, который закрывает **Channel** по завершении операции отправки сообщения. Или можно использовать готовое решение:

```
f.addListener(ChannelFutureListener.CLOSE);
```

Написание TIME-клиента

В отличие от DISCARD- и ECHO-серверов, для работы с TIME-протоколом необходим клиент, поскольку пользователю неудобно вручную преобразовывать 32-битное целое число в конкретную дату. Наибольшее отличие Netty-клиента и сервера — это используемые **Bootstrap-** и **Channel**-реализации.

```
package io.netty.example.time;

public class TimeClient {
    public static void main(String[] args) throws Exception {
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {
            Bootstrap b = new Bootstrap();                    // (1)
            b.group(workerGroup);                             // (2)
            b.channel(NioSocketChannel.class);                // (3)
            b.option(ChannelOption.SO_KEEPALIVE, true);
            b.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new TimeClientHandler());
                }
            });

            // Start the client.
            ChannelFuture f = b.connect(host, port).sync();   // (4)

            // Wait until the connection is closed.
            f.channel().closeFuture().sync();
```

```
        } finally {
            workerGroup.shutdownGracefully();
        }
    }
}
```

1. Bootstrap похож на ServerBootstrap, но предназначен для клиентских каналов.
2. При указании одного объекта EventLoopGroup он будет использоваться и в качестве **boss group**, и как **worker group**. С другой стороны, **boss worker** не используется на клиенте.
3. Вместо NioServerSocketChannel используется клиентский NioSocketChannel.
4. Вместо метода **bind()** применяется **connect()**.

Реализация ChannelHandler должна получать 32-битное целое число от сервера, переводить его в понятный формат, печатать полученное время в консоль и закрывать соединение:

```
package io.netty.example.time;

import java.util.Date;

public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg; // (1)
        try {
            long currentTimeMillis = (m.readUnsignedInt() - 2208988800L) *
1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        } finally {
            m.release();
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

Реализация кажется довольно простой и не сильно отличающейся от серверной стороны, но может возникнуть **IndexOutOfBoundsException**, о котором поговорим далее.

Работа со Stream-based протоколом

В **stream-based** протоколе (вроде TCP/IP) полученные данные хранятся во входящем буфере сокета. Но буфер **stream-based** протокола хранит не отдельные пакеты, а просто полученные байты. Это означает, что даже если два сообщения пришли в разных пакетах, операционная система считает их

просто пачкой байт. Так что нет гарантии, что мы прочитаем именно то, что нам отправили, поскольку могла прийти только часть сообщения, или в этой пачке байт их было несколько.

Первое решение

Такая же проблема возникает при реализации TIME-клиента. 32-битное целое число может приходить частями. Самое простое решение — создать внутренний кумулятивный буфер для ожидания и записи всех четырех байт. Для этого необходимо модифицировать **TimeClientHandler**:

```
package io.netty.example.time;

import java.util.Date;

public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    private ByteBuf buf;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        buf = ctx.alloc().buffer(4);     // (1)
    }

    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) {
        buf.release();                   // (1)
        buf = null;
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg;
        buf.writeBytes(m);               // (2)
        m.release();

        if (buf.readableBytes() >= 4) {  // (3)
            long currentTimeMillis = (buf.readUnsignedInt() - 2208988800L) *
1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. ChannelHandler имеет два метода жизненного цикла: **handlerAdded()** и **handlerRemoved()**. В них можно производить (де)инициализацию работы, если это не занимает продолжительное время.

2. Все входящие данные должны быть накоплены в **buf**.

3. Далее обработчик должен проверить, что в **buf** достаточно данных (4 байта в нашем случае), и выполнить преобразование в **int32**.

Второе решение

Предложенный вариант решает проблему TIME-клиента, но такая модификация обработчика выглядит неаккуратной. При более сложном протоколе, если сообщение имеет поля переменной длины или используется усложненная логика, написание обработчика будет трудоемкой задачей. И его скоро станет тяжело сопровождать. Можно добавить больше ChannelHandler в ChannelPipeline, разбить один монолитный ChannelHandler на множество более мелких блоков. Например, **TimeClientHandler** можно представить как два обработчика:

- **TimeDecoder**, который собирает сообщение из байтов;
- Простая версия **TimeClientHandler**.

Но **Netty** предоставляет готовые решения этих проблем:

```
package io.netty.example.time;

public class TimeDecoder extends ByteToMessageDecoder { // (1)
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) {                                                  // (2)
        if (in.readableBytes() < 4) {
            return;                                     // (3)
        }

        out.add(in.readBytes(4));                       // (4)
    }
}
```

1. ByteToMessageDecoder — реализация ChannelInboundHandler, которая решает проблему фрагментации сообщений.
2. ByteToMessageDecoder вызывает метод **decode()** с внутренним кумулятивным буфером при получении новых данных.
3. **decode()** может ничего не добавлять в **out**, если в кумулятивном буфере не хватает данных для получения готового сообщения. ByteToMessageDecoder будет и дальше вызывать **decode()** при получении новых данных.
4. Если метод **decode()** добавляет объект в **out**, это означает, что декодеру удалось декодировать/собрать готовое сообщение. Затем ByteToMessageDecoder выкидывает из буфера обработанную часть.

Чтобы добавить новый обработчик в ChannelPipeline, необходимо модифицировать ChannelInitializer-реализацию в **TimeClient**:

```
b.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new TimeDecoder(), new TimeClientHandler());
    }
});
```

Кроме того, **Netty** предоставляет готовые декодеры, которые могут решать множество стандартных задач.

Работа с POJO вместо ByteBuf

Все приведенные выше примеры работали с ByteBuf как основной структурой данных для передачи сообщений. Попробуем модифицировать TIME-протокол на стороне сервера и клиента, чтобы использовать POJO вместо ByteBuf. Это сделает код более чистым и поддерживаемым.

Создадим новый тип данных — **UnixTime**:

```
package io.netty.example.time;

import java.util.Date;

public class UnixTime {

    private final long value;

    public UnixTime() {
        this(System.currentTimeMillis() / 1000L + 2208988800L);
    }

    public UnixTime(long value) {
        this.value = value;
    }

    public long value() {
        return value;
    }

    @Override
    public String toString() {
        return new Date((value() - 2208988800L) * 1000L).toString();
    }
}
```

Теперь можно использовать **TimeDecoder** для получения **UnixTime** вместо ByteBuf:

```
@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    if (in.readableBytes() < 4) {
        return;
    }

    out.add(new UnixTime(in.readUnsignedInt()));
}
```

После модификации декодера **TimeClientHandler** больше не получает на вход ByteBuf:

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    UnixTime m = (UnixTime) msg;
    System.out.println(m);
    ctx.close();
}
```

Теперь код выглядит намного проще. То же самое нужно сделать и на стороне сервера:

```
@Override
public void channelActive(ChannelHandlerContext ctx) {
    ChannelFuture f = ctx.writeAndFlush(new UnixTime());
    f.addListener(ChannelFutureListener.CLOSE);
}
```

Не хватает только энкодера, который является реализацией ChannelOutboundHandler и преобразует **UnixTime** обратно в ByteBuf. Энкодер намного проще декодера, поскольку нет проблем с фрагментацией сообщений.

```
package io.netty.example.time;

public class TimeEncoder extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise
promise) {
        UnixTime m = (UnixTime) msg;
        ByteBuf encoded = ctx.alloc().buffer(4);
        encoded.writeInt((int)m.value());
        ctx.write(encoded, promise); // (1)
    }
}
```

1. В этой строке передаем ChannelPromise как есть, и Netty помечает его как удачный или проваленный после записи байтовых данных в канал. Ещё важный момент: мы не вызывали метод **ctx.flush()**. Существует отдельный метод обработчика **void flush(ChannelHandlerContext ctx)**, который может быть переопределен.

Можно упростить решение, используя MessageToByteEncoder:

```
public class TimeEncoder extends MessageToByteEncoder<UnixTime> {
    @Override
    protected void encode(ChannelHandlerContext ctx, UnixTime msg, ByteBuf out)
{
        out.writeInt((int)msg.value());
    }
}
```

Последняя задача — добавить **TimeEncoder** в ChannelPipeline на стороне сервера перед **TimeServerHandler**.

Завершение работы приложения

Работа Netty-приложения должна заканчиваться через завершение всех EventLoopGroup с помощью метода **shutdownGracefully()**. Он возвращает Future, сообщающий, что все EventLoopGroup завершены и все каналы, обрабатываемые этими пулами, закрыты.

Домашнее задание

1. Подготовить текстовый файл с описанием проделанной за неделю работы, вопросами по решению отдельных задач (если они возникли), отдельными блоками кода, которые вызвали у вас затруднения (если такие есть).

Дополнительные материалы

1. http://netty.io/index.html

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. http://netty.io/wiki/user-guide-for-4.x.html