


5.4 Netty 面试题总结

Netty 总算总结完了，Guide 也是长舒了一口气。有太多读者私信我让我总结 Netty 了，因为经常会在面试中碰到 Netty 相关的问题。

全文采用大家喜欢的与面试官对话的形式展开。如果大家觉得 Guide 总结的不错的话，不妨向好朋友们推荐一下 JavaGuide，这是最好礼物，哈哈！

5.4.1 Netty 是什么？

 **面试官**：介绍一下自己对 Netty 的认识吧！小伙子。

 **我**：好的！那我就简单用 3 点来概括一下 Netty 吧！

1. Netty 是一个 **基于 NIO** 的 client-server(客户端服务器)框架，使用它可以快速简单地开发网络应用程序。
2. 它极大地简化并优化了 TCP 和 UDP 套接字服务器等网络编程,并且性能以及安全性等很多方面甚至都要更好。
3. **支持多种协议** 如 FTP，SMTP，HTTP 以及各种二进制和基于文本的传统协议。


用官方的总结就是：**Netty 成功地找到了一种在不妥协可维护性和性能的情况下实现易于开发，性能，稳定性和灵活性的方法。**

除了上面介绍的之外，很多开源项目比如我们常用的 Dubbo、RocketMQ、Elasticsearch、gRPC 等等都用到了 Netty。

网络编程我愿意称中 Netty 为王。


5.4.2 为什么要用 Netty？


 **面试官**：为什么要用 Netty 呢？能不能说一下自己的看法。

 **我**：因为 Netty 具有下面这些优点，并且相比于直接使用 JDK 自带的 NIO 相关的 API 来说更加易用。

- 统一的 API，支持多种传输类型，阻塞和非阻塞的。
- 简单而强大的线程模型。
- 自带编解码器解决 TCP 粘包/拆包问题。
- 自带各种协议栈。
- 真正的无连接数据包套接字支持。
- 比直接使用 Java 核心 API 有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
- 安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。
- 社区活跃
- 成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty，比如我们经常接触的 Dubbo、RocketMQ 等等。
-

5.4.3 Netty 应用场景了解么？


 **面试官**：能不能通俗地说一下使用 Netty 可以做什么事情？

 **我**：凭借自己的了解，简单说一下吧！理论上来说，NIO 可以做的事情，使用 Netty 都可以做并且更好。Netty 主要用来做**网络通信**：

1. **作为 RPC 框架的网络通信工具**：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务节点之间的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！
2. **实现一个自己的 HTTP 服务器**：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为 Java 后端开发，我们一般使用 Tomcat 比较多。一个最基本的 HTTP 服务器可要以处理常见的 HTTP Method 的请求，比如 POST 请求、GET 请求等等。
3. **实现一个即时通讯系统**：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。
4. ****实现消息推送系统****：市面上有很多消息推送系统都是基于 Netty 来做的。
5.

5.4.4 Netty 核心组件有哪些？分别有什么作用？

 **面试官**：Netty 核心组件有哪些？分别有什么作用？

 **我**：表面上，嘴上开始说起 Netty 的核心组件有哪些，实则，内心已经开始 mmp 了，深度怀疑这面试官是存心搞我啊！

1.Channel

`Channel` 接口是 Netty 对网络操作抽象类，它除了包括基本的 I/O 操作，如 `bind()`、`connect()`、`read()`、`write()` 等。

比较常用的 `Channel` 接口实现类是 `NioServerSocketChannel`（服务端）和 `NioSocketChannel`（客户端），这两个 `Channel` 可以和 BIO 编程模型中的 `ServerSocket` 以及 `Socket` 两个概念对应上。Netty 的 `Channel` 接口所提供的 API，大大地降低了直接使用 `Socket` 类的复杂性。

2.EventLoop

这么说吧！`EventLoop`（事件循环）接口可以说是 Netty 中最核心的概念了！

《Netty 实战》这本书是这样介绍它的：

`EventLoop` 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。

是不是很难理解？说实话，我学习 Netty 的时候看到这句话是没太能理解的。

说白了，`EventLoop` 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

那 `Channel` 和 `EventLoop` 直接有啥联系呢？

`Channel` 为 Netty 网络操作(读写等操作)抽象类，`EventLoop` 负责处理注册到其上的 `Channel` 处理 I/O 操作，两者配合参与 I/O 操作。

3.ChannelFuture

Netty 是异步非阻塞的，所有的 I/O 操作都为异步的。

因此，我们不能立刻得到操作是否执行成功，但是，你可以通过 `ChannelFuture` 接口的 `addListener()` 方法注册一个 `ChannelFutureListener`，当操作执行成功或者失败时，监听就会自动触发返回结果。

并且，你还可以通过 `ChannelFuture` 的 `channel()` 方法获取关联的 `Channel`

```

1 public interface ChannelFuture extends Future<Void> {
2     Channel channel();
3
4     ChannelFuture addListener(GenericFutureListener<? extends Future<? super
Void>> var1);
5     .....
6
7     ChannelFuture sync() throws InterruptedException;
8 }

```

另外，我们还可以通过 `ChannelFuture` 接口的 `sync()` 方法让异步的操作变成同步的。

4.ChannelHandler 和 ChannelPipeline

下面这段代码使用过 Netty 的小伙伴应该不会陌生，我们指定了序列化编解码器以及自定义的 `ChannelHandler` 处理消息。

```

1         b.group(eventLoopGroup)
2             .handler(new ChannelInitializer<SocketChannel>() {
3                 @Override
4                 protected void initChannel(SocketChannel ch) {
5                     ch.pipeline().addLast(new
NettyKryoDecoder(kryoSerializer, RpcResponse.class));
6                     ch.pipeline().addLast(new
NettyKryoEncoder(kryoSerializer, RpcRequest.class));
7                     ch.pipeline().addLast(new KryoClientHandler());
8                 }
9             });


```

`ChannelHandler` 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

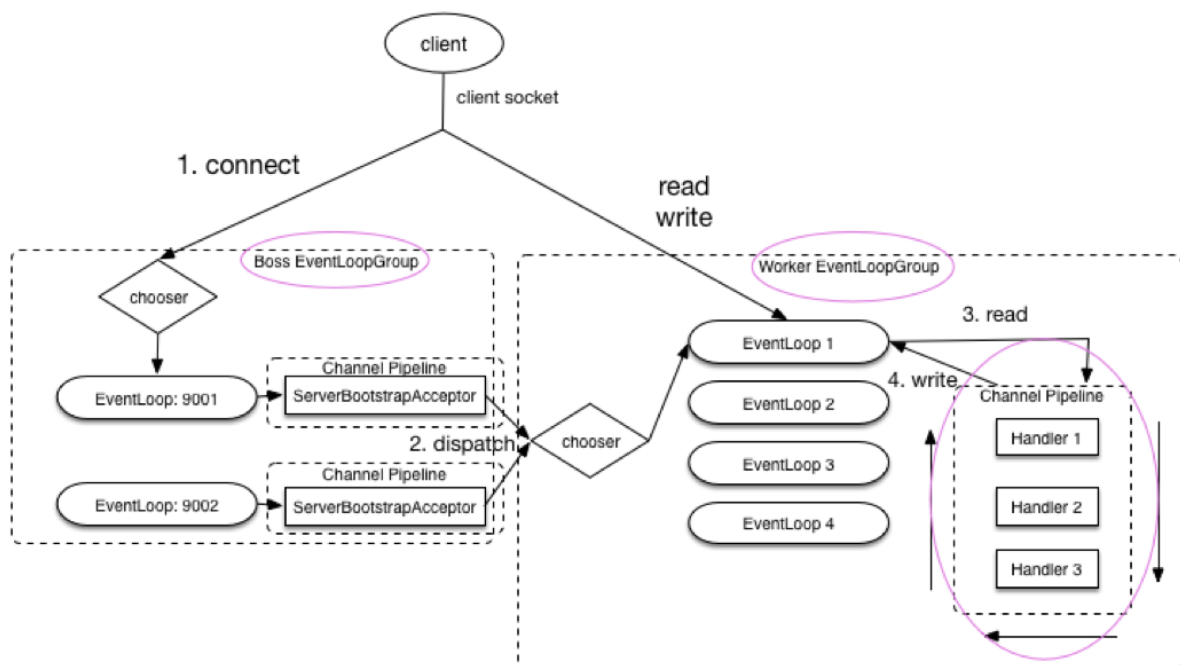
`ChannelPipeline` 为 `ChannelHandler` 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API。当 `Channel` 被创建时，它会被自动地分配到它专属的 `ChannelPipeline`。

我们可以在 `ChannelPipeline` 上通过 `addLast()` 方法添加一个或者多个 `ChannelHandler`，因为一个数据或者事件可能会被多个 Handler 处理。当一个 `ChannelHandler` 处理完之后就将数据交给下一个 `ChannelHandler`。

5.4.5 EventloopGroup 了解么?和 EventLoop 啥关系?

 **面试官**：刚刚你也介绍了 `EventLoop`。那你再说说 `EventloopGroup` 吧！和 `EventLoop` 啥关系？

 **我**：



EventLoopGroup 包含多个 EventLoop（每一个 EventLoop 通常内部包含一个线程），上面我们已经说了 EventLoop 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

并且 EventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理，即 Thread 和 EventLoop 属于 1:1 的关系，从而保证线程安全。

上图是一个服务端对 EventLoopGroup 使用的大致模块图，其中 Boss EventLoopGroup 用于接收连接，Worker EventLoopGroup 用于具体的处理（消息的读写以及其他逻辑处理）。

从上图可以看出：当客户端通过 connect 方法连接服务端时，bossGroup 处理客户端连接请求。当客户端处理完成后，会将这个连接提交给 workerGroup 来处理，然后 workerGroup 负责处理其 IO 相关操作。

5.4.6 Bootstrap 和 ServerBootstrap 了解么？

面试官：你再说说自己对 Bootstrap 和 ServerBootstrap 的了解吧！

我：

Bootstrap 是客户端的启动引导类/辅助类，具体使用方法如下：

```

1      EventLoopGroup group = new NioEventLoopGroup();
2      try {
3          //创建客户端启动引导/辅助类: Bootstrap
4          Bootstrap b = new Bootstrap();
5          //指定线程模型
6          b.group(group).
7          .....
8          // 尝试建立连接
9          channelFuture f = b.connect(host, port).sync();
10         f.channel().closeFuture().sync();
11     } finally {
12         // 优雅关闭相关线程组资源
13         group.shutdownGracefully();
14     }

```

ServerBootstrap 客户端的启动引导类/辅助类，具体使用方法如下：

```


1      // 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
2      EventLoopGroup bossGroup = new NioEventLoopGroup(1);
3      EventLoopGroup workerGroup = new NioEventLoopGroup();
4      try {
5          //2.创建服务端启动引导/辅助类: ServerBootstrap
6          ServerBootstrap b = new ServerBootstrap();
7          //3.给引导类配置两大线程组,确定了线程模型
8          b.group(bossGroup, workerGroup).
9              .....
10         // 6.绑定端口
11         ChannelFuture f = b.bind(port).sync();
12         // 等待连接关闭
13         f.channel().closeFuture().sync();
14     } finally {
15         //7.优雅关闭相关线程组资源
16         bossGroup.shutdownGracefully();
17         workerGroup.shutdownGracefully();
18     }
19 }


```

从上面的示例中, 我们可以看出:

1. `Bootstrap` 通常使用 `connect()` 方法连接到远程的主机和端口, 作为一个 Netty TCP 协议通信中的客户端。另外, `Bootstrap` 也可以通过 `bind()` 方法绑定本地的一个端口, 作为 UDP 协议通信中的一端。
2. `ServerBootstrap` 通常使用 `bind()` 方法绑定本地的端口上, 然后等待客户端的连接。
3. `Bootstrap` 只需要配置一个线程组— `EventLoopGroup`, 而 `ServerBootstrap` 需要配置两个线程组— `EventLoopGroup`, 一个用于接收连接, 一个用于具体的处理。

5.4.7 NioEventLoopGroup 默认的构造函数会起多少线程?

 **面试官**: 看过 Netty 的源码了么? `NioEventLoopGroup` 默认的构造函数会起多少线程呢?

 **我**: 嗯嗯! 看过部分。

回顾我们在上面写的服务器端的代码:

```

1      // 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
2      EventLoopGroup bossGroup = new NioEventLoopGroup(1);
3      EventLoopGroup workerGroup = new NioEventLoopGroup();

```

为了搞清楚 `NioEventLoopGroup` 默认的构造函数 到底创建了多少个线程, 我们来看一下它的源码。

```

1      /**
2       * 无参构造函数。
3       * nThreads:0
4       */
5      public NioEventLoopGroup() {
6          //调用下一个构造方法
7          this(0);
8      }
9
10     /**
11     * Executor: null
12     */
13     public NioEventLoopGroup(int nThreads) {

```

```

14         //继续调用下一个构造方法
15         this(nThreads, (Executor) null);
16     }
17
18     //中间省略部分构造函数
19
20     /**
21      * RejectedExecutionHandler () : RejectedExecutionHandlers.reject()
22      */
23     public NioEventLoopGroup(int nThreads, Executor executor, final
SelectorProvider selectorProvider, final SelectStrategyFactory
selectStrategyFactory) {
24         //开始调用父类的构造函数
25         super(nThreads, executor, selectorProvider, selectStrategyFactory,
RejectedExecutionHandlers.reject());
26     }
27

```

一直向下走下去的话，你会发现在 `MultithreadEventLoopGroup` 类中有相关的指定线程数的代码，如下：

```


1         // 从1，系统属性，CPU核心数*2 这三个值中取出一个最大的
2         //可以得出 DEFAULT_EVENT_LOOP_THREADS 的值为CPU核心数*2
3         private static final int DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
NettyRuntime.availableProcessors() * 2));
4
5         // 被调用的父类构造函数，NioEventLoopGroup 默认的构造函数会起多少线程的秘密所在
6         // 当指定的线程数nThreads为0时，使用默认的线程数DEFAULT_EVENT_LOOP_THREADS
7         protected MultithreadEventLoopGroup(int nThreads, ThreadFactory
threadFactory, Object... args) {
8             super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads,
threadFactory, args);
9         }

```

综上，我们发现 `NioEventLoopGroup` 默认的构造函数实际会起的线程数为 `CPU核心数*2`。

另外，如果你继续深入下去看构造函数的话，你会发现每个 `NioEventLoopGroup` 对象内部都会分配一组 `NioEventLoop`，其大小是 `nThreads`，这样就构成了一个线程池，一个 `NioEventLoop` 和一个线程相对应，这和我们上面说的 `EventLoopGroup` 和 `EventLoop` 关系这部分内容相对应。

5.4.8 Netty 线程模型了解么？

 **面试官**：说一下 Netty 线程模型吧！

 **我**：大部分网络框架都是基于 Reactor 模式设计开发的。

Reactor 模式基于事件驱动，采用多路复用将事件分发给相应的 Handler 处理，非常适合处理海量 IO 的场景。

在 Netty 主要靠 `NioEventLoopGroup` 线程池来实现具体的线程模型的。

我们实现服务端的时候，一般会初始化两个线程组：

1. `bossGroup` :接收连接。
2. `workerGroup` ：负责具体的处理，交由对应的 Handler 处理。

下面我们来详细看一下 Netty 中的线程模型吧！

1.单线程模型：

一个线程需要执行处理所有的 `accept`、`read`、`decode`、`process`、`encode`、`send` 事件。对于高负载、高并发，并且对性能要求比较高的场景不适用。

对应到 Netty 代码是下面这样的

使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 **CPU 核心数 *2**。

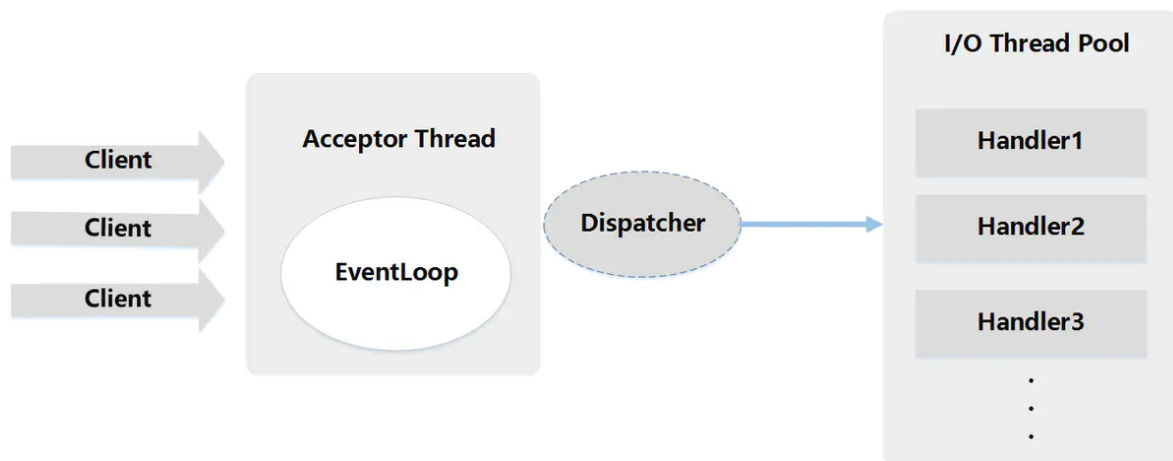
```
1 //1.eventGroup既用于处理客户端连接，又负责具体的处理。
2 EventLoopGroup eventGroup = new NioEventLoopGroup(1);
3 //2.创建服务端启动引导/辅助类: ServerBootstrap
4 ServerBootstrap b = new ServerBootstrap();
5     bootstrap.group(eventGroup, eventGroup)
6     //.....
```

2.多线程模型

一个 Acceptor 线程只负责监听客户端的连接，一个 NIO 线程池负责具体处理：`accept`、`read`、`decode`、`process`、`encode`、`send` 事件。满足绝大部分应用场景，并发连接量不大的时候没啥问题，但是遇到并发连接大的时候就可能会出现性能问题，成为性能瓶颈。

对应到 Netty 代码是下面这样的：

```
1 // 1.bossGroup 用于接收连接，workerGroup 用于具体的处理
2 EventLoopGroup bossGroup = new NioEventLoopGroup(1);
3 EventLoopGroup workerGroup = new NioEventLoopGroup();
4 try {
5     //2.创建服务端启动引导/辅助类: ServerBootstrap
6     ServerBootstrap b = new ServerBootstrap();
7     //3.给引导类配置两大线程组,确定了线程模型
8     b.group(bossGroup, workerGroup)
9     //.....
```



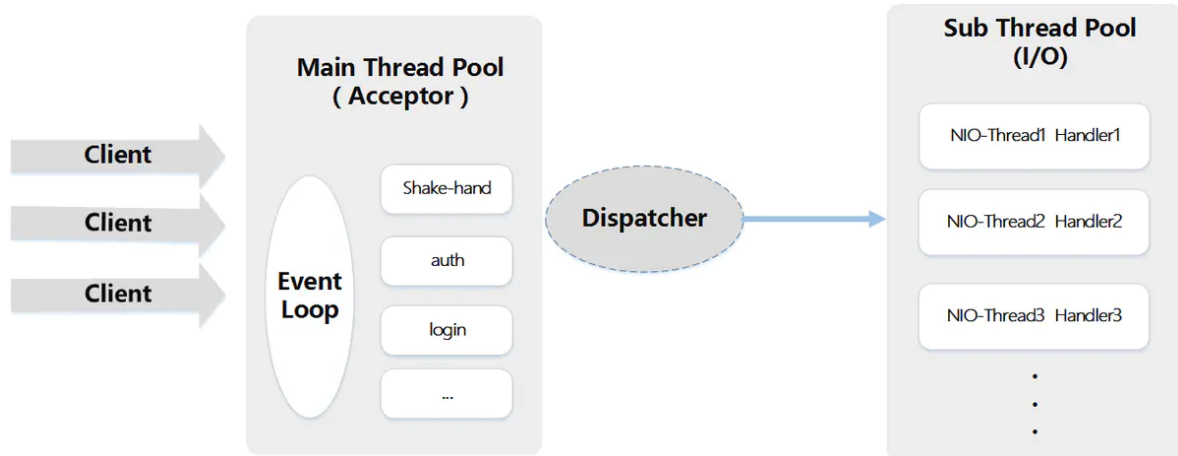
3.主从多线程模型

从一个主线程 NIO 线程池中选择一个线程作为 Acceptor 线程，绑定监听端口，接收客户端连接的连接，其他线程负责后续的接入认证等工作。连接建立完成后，Sub NIO 线程池负责具体处理 I/O 读写。如果多线程模型无法满足你的需求的时候，可以考虑使用主从多线程模型。

```

1 // 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
2 EventLoopGroup bossGroup = new NioEventLoopGroup();
3 EventLoopGroup workerGroup = new NioEventLoopGroup();
4 try {
5     //2.创建服务端启动引导/辅助类: ServerBootstrap
6     ServerBootstrap b = new ServerBootstrap();
7     //3.给引导类配置两大线程组,确定了线程模型
8     b.group(bossGroup, workerGroup)
9     //.....

```



5.4.9 Netty 服务端和客户端的启动过程了解么?

服务端

```

1 // 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
2 EventLoopGroup bossGroup = new NioEventLoopGroup(1);
3 EventLoopGroup workerGroup = new NioEventLoopGroup();
4 try {
5     //2.创建服务端启动引导/辅助类: ServerBootstrap
6     ServerBootstrap b = new ServerBootstrap();
7     //3.给引导类配置两大线程组,确定了线程模型
8     b.group(bossGroup, workerGroup)
9         // (非必备)打印日志
10        .handler(new LoggingHandler(LogLevel.INFO))
11        // 4.指定 IO 模型
12        .channel(NioServerSocketChannel.class)
13        .childHandler(new ChannelInitializer<SocketChannel>() {
14            @Override
15            public void initChannel(SocketChannel ch) {
16                ChannelPipeline p = ch.pipeline();
17                //5.可以自定义客户端消息的业务处理逻辑
18                p.addLast(new HelloServerHandler());
19            }
20        });
21    // 6.绑定端口,调用 sync 方法阻塞知道绑定完成
22    ChannelFuture f = b.bind(port).sync();
23    // 7.阻塞等待直到服务器Channel关闭(closeFuture()方法获取Channel 的
    CloseFuture对象,然后调用sync()方法)
24    f.channel().closeFuture().sync();
25    } finally {
26        //8.优雅关闭相关线程组资源
27        bossGroup.shutdownGracefully();
28        workerGroup.shutdownGracefully();

```


简单解析一下服务端的创建过程具体是怎样的：

1.首先你创建了两个 `NioEventLoopGroup` 对象实例：`bossGroup` 和 `workerGroup`。

- `bossGroup`：用于处理客户端的 TCP 连接请求。
- `workerGroup`：负责每一条连接的具体读写数据的处理逻辑，真正负责 I/O 读写操作，交由对应的 Handler 处理。

举个例子：我们把公司的老板当做 `bossGroup`，员工当做 `workerGroup`，`bossGroup` 在外面接完活之后，扔给 `workerGroup` 去处理。一般情况下我们会指定 `bossGroup` 的线程数为 1（并发连接量不大的时候），`workGroup` 的线程数量为 **CPU 核心数 * 2**。另外，根据源码来看，使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 **CPU 核心数 * 2**。

2.接下来 我们创建了一个服务端启动引导/辅助类：`ServerBootstrap`，这个类将引导我们进行服务端的启动工作。

3.通过 `.group()` 方法给引导类 `ServerBootstrap` 配置两大线程组，确定了线程模型。

通过下面的代码，我们实际配置的是多线程模型，这个在上面提到过。

```
1 EventLoopGroup bossGroup = new NioEventLoopGroup(1);
2 EventLoopGroup workerGroup = new NioEventLoopGroup();
```

4.通过 `channel()` 方法给引导类 `ServerBootstrap` 指定了 IO 模型为 `NIO`

- `NioServerSocketChannel`：指定服务端的 IO 模型为 NIO，与 BIO 编程模型中的 `ServerSocket` 对应
- `NioSocketChannel`：指定客户端的 IO 模型为 NIO，与 BIO 编程模型中的 `socket` 对应

5.通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer`，然后指定了服务端消息的业务处理逻辑 `HelloServerHandler` 对象

6.调用 `ServerBootstrap` 类的 `bind()` 方法绑定端口

客户端

```
1 //1.创建一个 NioEventLoopGroup 对象实例
2 EventLoopGroup group = new NioEventLoopGroup();
3 try {
4     //2.创建客户端启动引导/辅助类: Bootstrap
5     Bootstrap b = new Bootstrap();
6     //3.指定线程组
7     b.group(group)
8     //4.指定 IO 模型
9     .channel(NioSocketChannel.class)
10    .handler(new ChannelInitializer<SocketChannel>() {
11        @Override
12        public void initChannel(SocketChannel ch) throws
Exception {
13            ChannelPipeline p = ch.pipeline();
14            // 5.这里可以自定义消息的业务处理逻辑
15            p.addLast(new HelloClientHandler(message));
16        }
17    });
18    // 6.尝试建立连接
```

```

19         ChannelFuture f = b.connect(host, port).sync();
20         // 7.等待连接关闭（阻塞，直到Channel关闭）
21         f.channel().closeFuture().sync();
22     } finally {
23         group.shutdownGracefully();
24     }

```

继续分析一下客户端的创建流程：

1. 创建一个 `NioEventLoopGroup` 对象实例
2. 创建客户端启动的引导类是 `Bootstrap`
3. 通过 `.group()` 方法给引导类 `Bootstrap` 配置一个线程组
4. 通过 `channel()` 方法给引导类 `Bootstrap` 指定了 IO 模型为 `NIO`
5. 通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer`，然后指定了客户端消息的业务处理逻辑 `HelloClientHandler` 对象
6. 调用 `Bootstrap` 类的 `connect()` 方法进行连接，这个方法需要指定两个参数：

- `inetHost` : ip 地址
- `inetPort` : 端口号

```

1     public ChannelFuture connect(String inetHost, int inetPort) {
2         return this.connect(InetSocketAddress.createUnresolved(inetHost,
3             inetPort));
4     }
5     public ChannelFuture connect(SocketAddress remoteAddress) {
6         ObjectUtil.checkNotNull(remoteAddress, "remoteAddress");
7         this.validate();
8         return this.doResolveAndConnect(remoteAddress,
9             this.config.localAddress());
10    }

```

`connect` 方法返回的是一个 `Future` 类型的对象

```

1     public interface ChannelFuture extends Future<Void> {
2         .....
3     }

```

也就是说这个方是异步的，我们通过 `addListener` 方法可以监听到连接是否成功，进而打印出连接信息。具体做法很简单，只需要对代码进行以下改动：

```

1     ChannelFuture f = b.connect(host, port).addListener(future -> {
2         if (future.isSuccess()) {
3             System.out.println("连接成功!");
4         } else {
5             System.err.println("连接失败!");
6         }
7     }).sync();

```

5.4.10 什么是 TCP 粘包/拆包?有什么解决办法呢?

 面试官：什么是 TCP 粘包/拆包?

在 TCP 保持长连接的过程中, 可能会出现断网等网络异常出现, 异常发生的时候, client 与 server 之间如果没有交互的话, 它们是无法发现对方已经掉线的。为了解决这个问题, 我们就需要引入 **心跳机制**。

心跳机制的工作原理是: 在 client 与 server 之间在一定时间内没有数据交互时, 即处于 idle 状态时, 客户端或服务器就会发送一个特殊的数据包给对方, 当接收方收到这个数据报文后, 也立即发送一个特殊的数据报文, 回应发送方, 此即一个 PING-PONG 交互。所以, 当某一端收到心跳消息后, 就知道了对方仍然在线, 这就确保 TCP 连接的有效性。

TCP 实际上自带的就有长连接选项, 本身是也有心跳包机制, 也就是 TCP 的选项: `SO_KEEPALIVE`。但是, TCP 协议层面的长连接灵活性不够。所以, 一般情况下我们都是在应用层协议上实现自定义心跳机制的, 也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话, 核心类是 `IdleStateHandler`。

5.4.12 Netty 的零拷贝了解么?

 **面试官**: 讲讲 Netty 的零拷贝?

 **我**:

维基百科是这样介绍零拷贝的:

零复制 (英语: Zero-copy; 也译零拷贝) 技术是指计算机执行操作时, CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。

在 OS 层面上的 `zero-copy` 通常指避免在 `用户态(User-space)` 与 `内核态(kernel-space)` 之间来回拷贝数据。而在 Netty 层面, 零拷贝主要体现在对于数据操作的优化。

Netty 中的零拷贝体现在以下几个方面

1. 使用 Netty 提供的 `CompositeByteBuf` 类, 可以将多个 `ByteBuf` 合并为一个逻辑上的 `ByteBuf`, 避免了各个 `ByteBuf` 之间的拷贝。
2. `ByteBuf` 支持 slice 操作, 因此可以将 `ByteBuf` 分解为多个共享同一个存储区域的 `ByteBuf`, 避免了内存的拷贝。
3. 通过 `FileRegion` 包装的 `FileChannel.transferTo` 实现文件传输, 可以直接将文件缓冲区的数
据发送到目标 `Channel`, 避免了传统通过循环 write 方式导致的内存拷贝问题。

参考

- netty 学习系列二: NIO Reactor 模型 & Netty 线程模型: <https://www.jianshu.com/p/38b56531565d>
- 《Netty 实战》
- Netty 面试题整理(2):<https://metatronxl.github.io/2019/10/22/Netty-面试题整理-二/>
- Netty (3) —源码 NioEventLoopGroup:<https://www.cnblogs.com/qdhxhz/p/10075568.html>
- 对于 Netty ByteBuf 的零拷贝(Zero Copy) 的理解:
<https://www.cnblogs.com/xys1228/p/6088805.html>