链滴

Netty 面试必备知识点

作者：jingqueyimu

原文链接：https://ld246.com/article/1621860271615

来源网站：链滴

1、Netty 简介

- 是一个基于 NIO 的、异步的、事件驱动的网络通信框架。
- 简化了 TCP、UDP 等网络编程。
- 支持多种协议，如 FTP、SMTP、HTTP 等。

2、Netty 特点

- 高并发：基于 NIO，相比 BIO，并发性得到了很大的提高。
- 传输快：传输依赖于零拷贝。
- 封装好：封装了 NIO 操作的很多细节，提供易于使用的 API。

3、Netty 应用场景

- 实现特定协议的服务器，比如 HTTP 服务器。
- 作为 RPC 框架的网络通讯工具，比如 Dubbo。
- 实现即时通讯系统。
- 实现消息推送系统。

4、Netty 高性能表现在哪些方面

- 异步非阻塞通信：基于 NIO，支持阻塞和非阻塞两种模式。
- Reactor 线程模型：基于事件驱动、多路 IO 复用。
- 串行化处理读写：避免使用锁带来的性能开销。可同时启动多个串行化的线程并行运行。
- 高效的并发编程：对 volatile、CAS、原子类、线程安全容器、读写锁等的合理使用。
- 高性能序列化框架：支持 Google 的 Protobuf 等。
- 零拷贝：减少不必要的内存拷贝。
- 内存池：对申请的内存块进行划分，然后按需分配，并且可以重复使用。
- 灵活的 TCP 参数配置能力。

5、Netty 核心组件

- Bootstrap/ServerBootstrap：客户端/服务端启动引导类，用于串联各个组件。
- EventLoop：事件循环，用于处理连接过程中所发生的事件。主要负责监听网络事件，并调用事件理器处理 Channel 上发生的网络 IO 事件。
- EventLoopGroup：一组 EventLoop 的抽象。每个 EventLoop 内部包含一个线程。为了更好的利 CPU 资源，一般会有多个 EventLoop 同时工作。
- Channel：通道，网络 IO 操作的抽象类。用于执行网络 IO 操作，比如 bind()、connect()、read(、write() 等。
- ChannelFuture：用于保存 Channel 异步操作的结果。可通过 ChannelFuture 的 addListener() 册一个监听器，来监听操作结果。

● ChannelHandler：事件处理器，用于处理 Channel 上发生的事件。

● ChannelPipeline：将多个 ChannelHandler 组合在一起，形成一个链条。该链条会拦截并处理 Channel 上的事件。

6、Netty 服务端、客户端实现

服务端：

```
// bossGroup负责客户端连接
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
// workerGroup负责读写操作
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    // 创建服务端启动引导类
    ServerBootstrap bootstrap = new ServerBootstrap();
    // 设置事件循环组
    bootstrap.group(bossGroup, workerGroup)
            // 设置服务端通道(Channel),指定IO模型
            .channel(NioServerSocketChannel.class)
            // 初始化服务器连接队列大小,服务端处理客户端连接是顺序处理的,同一时间只能处理一个,
不及处理的将会放在队列中等待处理
            .option(ChannelOption.SO_BACKLOG, 1024)
            // 设置通道初始化器,用于初始化通道(Channel)
            .childHandler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel sc) throws Exception {
                    ChannelPipeline p = sc.pipeline();
                    // 设置ChannelHandler处理器链
                    p.addLast(new NettyServerHandler());
                }
            });
    // 启动服务端(并绑定端口),bind()是异步操作,sync()同步等待bind()执行完成
    ChannelFuture future = bootstrap.bind(9000).sync();
    // 监听通道关闭,closeFuture()是异步操作,sync()同步等待closeFuture()执行完成
    future.channel().closeFuture().sync();
} finally {
    // 优雅关闭事件循环组资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
```

客户端：

```
// 客户端事件循环组
EventLoopGroup eventGroup = new NioEventLoopGroup();
try {
    // 创建客户端启动引导类
    Bootstrap bootstrap = new Bootstrap();
    // 设置事件循环组
    bootstrap.group(eventGroup)
            // 设置客户端端通道(Channel),指定IO模型
            .channel(NioSocketChannel.class)
```

```
            // 设置通道初始化器,用于初始化通道(Channel)
            .handler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel sc) throws Exception {
                    ChannelPipeline p = sc.pipeline();
                    // 设置ChannelHandler处理器链
                    p.addLast(new NettyClientHandler());
                }
            });
    // 连接服务端,connect()是异步操作,sync()是等待connect()执行完成
    ChannelFuture future = bootstrap.connect("127.0.0.1", 9000).sync();
    // 监听通道关闭,closeFuture()是异步操作,sync()同步等待closeFuture()执行完成
    future.channel().closeFuture().sync();
} finally {
    // 优雅关闭事件循环组资源
    eventGroup.shutdownGracefully();
}
```

7、Reactor 模式

Reactor 模式是基于事件驱动的，它会监听事件的发生，当监听到事件发生后，根据多路复用策略，事件分发给相应的处理器处理。

核心组件：

- Handle（Event）：用于表示事件。
- Event Demultiplexer：事件分离器，用于同步等待事件的发生。
- Reactor：反应器，用于监听和分发事件。内部会调用 Event Demultiplexer 来同步等待事件的发，然后将事件交由 Event Handler 处理。
- Event Handler：事件处理器，用于处理事件。

8、网络编程中的 Reactor 模式

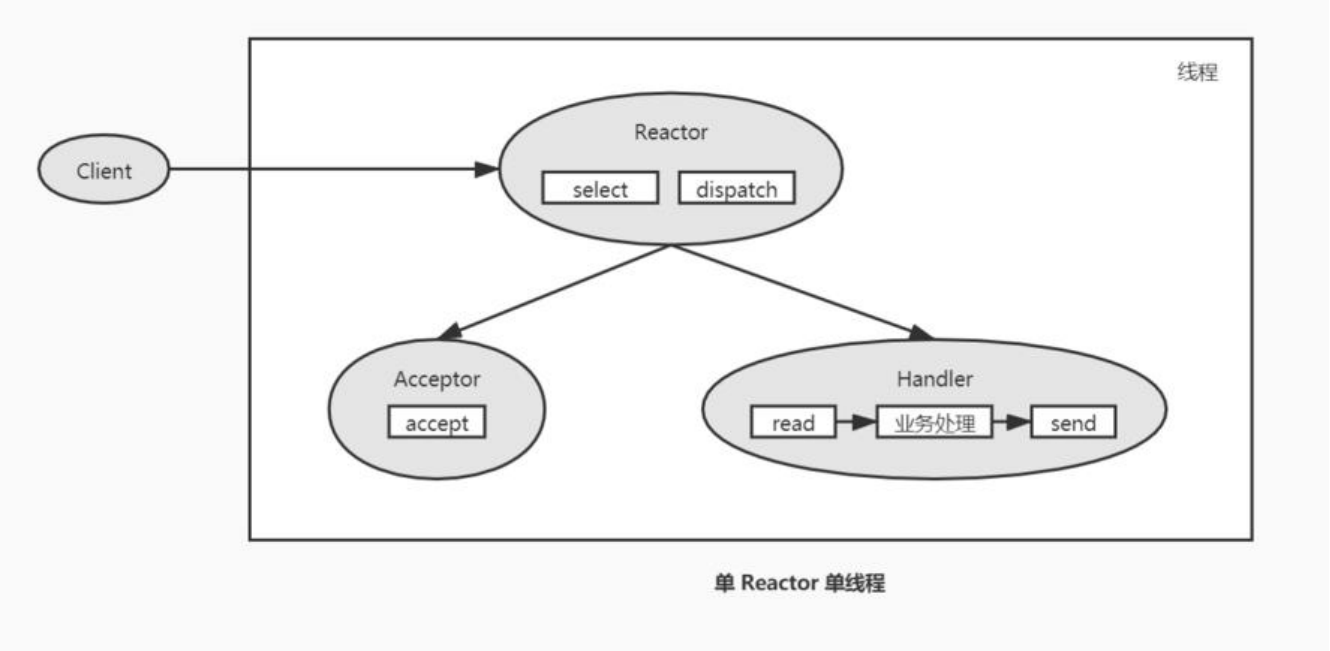
网络编程中，Reactor 模式的核心组成包括 Reactor 和处理资源池（进程池或线程池）。其中 Reacto 负责监听和分发事件，处理资源池负责处理事件。

主要包含三种角色：

- Reactor：负责监听事件，并将事件分发给绑定了该事件的 Handler。
- Handler：绑定了某类事件，负责处理事件。
- Acceptor：Handler 的一种，负责处理连接事件。

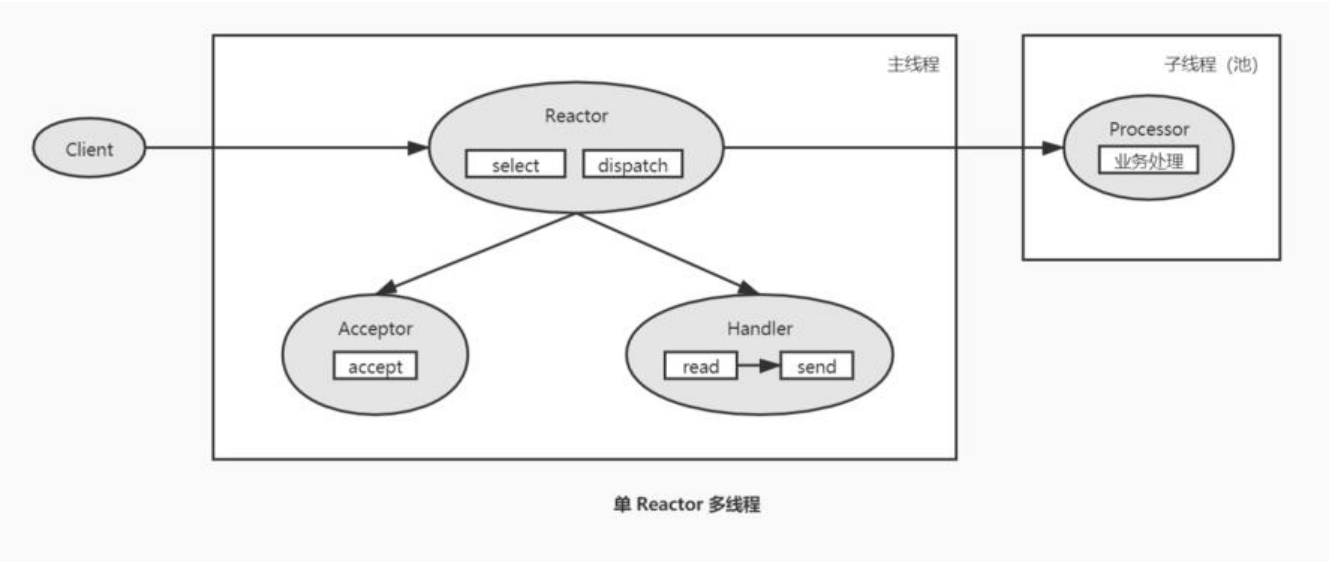
根据 Reactor 的数量和处理资源池（以线程池为例）的数量，可分为三种：

- 单 Reactor 单线程：Reactor 负责监听和分发事件，如果是连接事件，则由 Acceptor 处理，如果读写事件，则由 Handler 处理（同时进行业务处理）。实现简单，但是无法做到高性能，只适用于务处理非常快的场景。



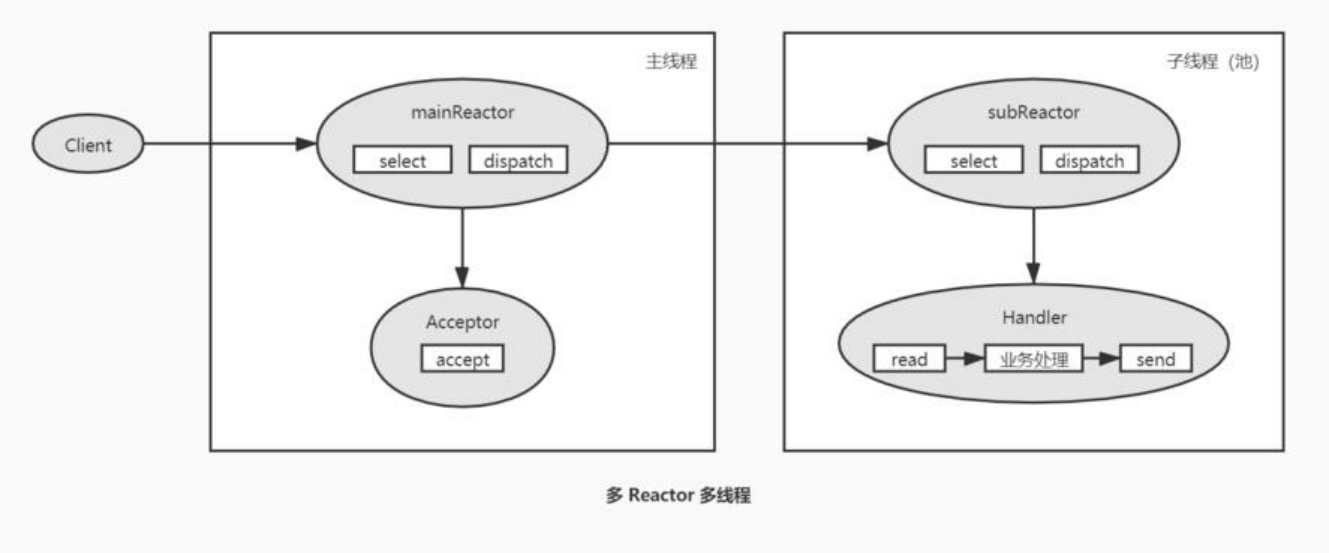
单 Reactor 单线程

● 单 Reactor 多线程：相对于单 Reactor 单线程来说，将 Handler 的执行放入线程池中（一般只将
务处理放入线程池中）。不适用于客户端并发连接量大的场景。



单 Reactor 多线程

● 多 Reactor 多线程（主从 Reactor 多线程）：将 Reactor 分成两部分：mainReactor、subReacto
。主线程的 mainReactor 负责监听连接事件，并交由 Acceptor 处理，Acceptor 将建立的连接注册
子线程的 subReactor。子线程的 subReactor 负责监听读写事件，并交由 Handler 处理（同时进行
务处理）。主流模型，性能最高。



多 Reactor 多线程

9、Netty 线程模型

Netty 线程模型就是 Reactor 模式的一个实现。主要靠 NioEventLoopGroup 线程池来实现具体的程模型。NioEventLoopGroup 默认线程数为 CPU 核心数 * 2。

Netty 实现服务端时，一般会创建两个线程组：bossGroup、workerGroup。其中 bossGroup 负责户端连接，workerGroup 负责读写操作以及业务处理。

● 单 Reactor 单线程模型：由一个线程同时负责客户端连接、读写操作以及业务处理。

代码实现：

```
// eventGroup(线程数为1)同时负责客户端连接,读写操作,业务处理
EventLoopGroup eventGroup = new NioEventLoopGroup(1);
// 创建服务端启动引导类
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(eventGroup, eventGroup)
...
```

● 单 Reactor 多线程模型：一个 Acceptor 线程负责客户端连接，一个 NIO 线程池负责读写操作、务处理。

代码实现：

```
// bossGroup(线程数为1,对应Acceptor线程)负责客户端连接
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
// workerGroup(对应NIO线程池)负责读写操作,业务处理
EventLoopGroup workerGroup = new NioEventLoopGroup();
// 创建服务端启动引导类
ServerBootstrap bootstrap = new ServerBootstrap();
// 设置线程组
bootstrap.group(bossGroup, workerGroup)
...
```

● 主从 Reactor 多线程模型：从 Acceptor 线程池中随机选择一个线程负责客户端连接，一个 NIO 程池负责读写操作、业务处理。

代码实现：

```
// bossGroup(对应Acceptor线程池)负责客户端连接
EventLoopGroup bossGroup = new NioEventLoopGroup();
// workerGroup(对应NIO线程池)负责读写操作
EventLoopGroup workerGroup = new NioEventLoopGroup();
// 创建服务端启动引导类
ServerBootstrap bootstrap = new ServerBootstrap();
// 设置线程组
bootstrap.group(bossGroup, workerGroup)
...
```

PS：实际上，Netty 中不存在主从 Reactor 多线程模型。因为服务端的 ServerSocketChannel 只会绑到 Acceptor 线程池中的一个线程上，因此，在调用 Java NIO 的 Selector.select() 处理客户端连接时实际上是在一个线程中。

10、TCP 粘包、拆包

● 粘包：基于 TCP 发送数据时，出现多次发送的数据“粘”在一起的情况。即接收端一次读取时，取到了发送端多次发送的数据。

● 拆包：基于 TCP 发送数据时，出现某次发送的数据被“拆”开的情况。即接收端一次读取时，只取到了发送端发送数据的一部分。

11、Netty 如何解决粘包、拆包

● 消息定长：每个数据包都固定长度，不足则以空格填充。Netty 提供 FixedLengthFrameDecoder 实现。

● 使用分隔符：在每个数据包的末尾加上特定的分隔符。Netty 提供 DelimiterBasedFrameDecoder 来实现。

● 将消息分为消息头和消息体，消息头保存消息的总长度。

● 自定义协议进行粘包、拆包处理。

12、TCP 短连接、长连接

● 短连接：TCP 客户端和服务端建立连接后，一旦该次读写完成就关闭连接。如果后续有新的读写，需要重新连接。短连接管理、实现简单，但是频繁连接会消耗网络资源、也耗费时间。

● 长连接：TCP 客户端和服务端建立连接后，即使该次读写完成也不会关闭连接。如果后续有新的读，则可继续使用该连接。长连接网络资源消耗低、节约时间，适合读写频繁的场景。

13、Netty 心跳机制

● 心跳机制原理：在 TCP 长连接过程中，客户端和服务端之间定期发送一种特殊的数据包，通知对自己还在线，以确保连接的有效性。

● Netty 实现心跳机制的核心类是 IdleStateHandler，它可以指定读超时、写超时、读/写超时时间一旦出现超时，则会触发 IdleStateEvent 事件。

14、Netty 零拷贝

零拷贝通常是指避免在操作系统的用户空间缓冲区（对应 JVM 的堆内存）与内核空间缓冲区（对应外直接内存）之间来回拷贝数据。

- Socket 读写零拷贝：ByteBuf 底层用于接收、发送数据的 ByteBuffer，使用堆外直接内存进行 Socet 读写，避免了堆内存和直接内存之间的拷贝。（使用堆内存进行 Socket 读写时，会先从 Socket 读数据到直接内存，再拷贝到堆内存缓冲区；或者先将堆内存缓冲区的数据拷贝到直接内存，再写入 Soket。）
- File 文件读写零拷贝：使用 FileRegion 的 transferTo 方法，直接把文件缓冲区的数据发送到目标hannel。
- ByteBuf 合并零拷贝：使用 CompositeByteBuf 类，将多个 ByteBuf 进行逻辑上的合并，避免多个ByteBuf 之间的拷贝。
- ByteBuf 拆分零拷贝：使用 ByteBuf 的 slice 方法，将 ByteBuf 进行逻辑上的拆分，避免内存的拷。

Reference

[1] https://snailclimb.gitee.io/javaguide-interview/#/./docs/e-4netty

[2] https://zhuanlan.zhihu.com/p/87630368

[3] https://zhuanlan.zhihu.com/p/93612337

[4] https://my.oschina.net/codingdiary/blog/4358541

[5] https://baijiahao.baidu.com/s?id=1643348149695182317

[6] https://zhuanlan.zhihu.com/p/88599349

[7] https://segmentfault.com/a/1190000007560884

[8] 《Netty 权威指南》

[9] 《从零开始学架构》