

# Netty

## 概念

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

IO模型: BIO,NIO,AIO;

**BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。

**NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，于 Java 1.4 中引入，支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 `Socket` 和 `ServerSocket` 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。

**AIO (Asynchronous I/O):** Java 7 中引入了 NIO 的改进版 NIO 2，应用不广泛

## NIO的实现

select, poll, epoll, 为什么epoll更佳

- select, poll实现需要自己不断轮询所有fd集合（无差别轮询），直到设备就绪，期间可能要睡眠和唤醒多次交替。

epoll其实也需要调用 `epoll_wait` 不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在 `epoll_wait` 中进入睡眠的进程。

虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间，这就是回调机制带来的性能提升。

- select, poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在`epoll_wait`的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列），这也能节省不少的开销

## 那些开源项目用到了Netty?

Dubbo、RocketMQ、Elasticsearch、gRPC 等等都用到了 Netty

## Netty核心组件

- **Bytebuf字节容器**，网络通信最终通过字节流进行传输；Bytebuf内部是一个字节，对Java的Bytebuffer的抽象封装
  - Netty的Zero-Copy技术
    - OS的Zero-copy: 用户态A--核心态，用户态B--核心态；变成用户态A--直接到用户态B，支持用户态A映射到核心态，避免的重复反馈从核心态拷贝数据到用户态内存；

- Netty的Zero-Copy：异曲同工，前者偏向内存拷贝上，Netty的偏向**优化数据操作**上；主要组件:ByteBuf, CompositeByteBuf避免多个ByteBuf的拷贝
- Bootstrap和ServerBootstrap，前者是客户端启动引导类，后者是服务端启动引导类
  - 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。
  - 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。
  - 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 IO 处理。
- Channel网络操作抽象类
  - 比较常用的 Channel 接口实现类如下，可以和 BIO 编程模型中的 ServerSocket 以及 Socket 两个概念对应上
    - NioServerSocketChannel （服务端）
    - NioSocketChannel （客户端）
- EventLoop事件循环
  - 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（读写）的处理
  - 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 的 I/O 操作，两者配合进行 I/O 操作
  - EventLoopGroup 包含多个 EventLoop （每一个 EventLoop 通常内部包含一个线程）
  - 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
- 一个Channel，对应多个ChannelPipeline(ChannelHandler对象链表)，ChannelHandler消息处理器
- ChannelFuture操作执行结果，Netty中I/O操作为异步，则通过ChannelFuture注册监控器，当操作执行成功的时候，监控方法自动触发返回结果

## Netty线程模型

Netty网络框架，跟大部分网络框架一样，都是基于**Reactor线程模型**

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，特别适合处理海量的I/O事件。

Reactor线程模型分为**单线程模型、多线程模型以及主从多线程模型**

- 单线程模型（一般不用）
  - 所有的IO操作都由同一个NIO线程处理
- 多线程模型
  - 一个线程负责接受请求,一组NIO线程处理IO操作（并发百万，一个线程接受请求会有性能问题）
- 主从多线程模型
  - 一组NIO线程负责接受请求，一组NIO线程处理IO操作

## Netty 服务端和客户端的启动过程

服务端

---

```

// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
        // (非必备)打印日志
        .handler(new LoggingHandler(LogLevel.INFO))
        // 4.指定 IO 模型
        .channel(NioServerSocketChannel.class)
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) {
                ChannelPipeline p = ch.pipeline();
                //5.可以自定义客户端消息的业务处理逻辑
                p.addLast(new HelloServerHandler());
            }
        });
    // 6.绑定端口,调用 sync 方法阻塞知道绑定完成
    ChannelFuture f = b.bind(port).sync();
    // 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)
    f.channel().closeFuture().sync();
} finally {
    //8.优雅关闭相关线程组资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}

```

## 客户端

```

//1.创建一个 NioEventLoopGroup 对象实例
EventLoopGroup group = new NioEventLoopGroup();
try {
    //2.创建客户端启动引导/辅助类: Bootstrap
    Bootstrap b = new Bootstrap();
    //3.指定线程组
    b.group(group)
        //4.指定 IO 模型
        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws
Exception {
                ChannelPipeline p = ch.pipeline();
                // 5.这里可以自定义消息的业务处理逻辑
                p.addLast(new HelloClientHandler(message));
            }
        });
    // 6.尝试建立连接
    ChannelFuture f = b.connect(host, port).sync();
    // 7.等待连接关闭 (阻塞,直到Channel关闭)
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully();
}

```

```
}
```

## TCP 粘包/拆包

TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。

Netty 自带的解码器可解决，或者自定义序列化编解码器

- `LineBasedFrameDecoder`：发送端发送数据包的时候，每个数据包之间以换行符作为分隔，`LineBasedFrameDecoder` 的工作原理是它依次遍历 `ByteBuf` 中的可读字节，判断是否有换行符，然后进行相应的截取。
- `DelimiterBasedFrameDecoder`：可以自定义分隔符解码器，`LineBasedFrameDecoder` 实际上是一种特殊的 `DelimiterBasedFrameDecoder` 解码器。
- `FixedLengthFrameDecoder`：固定长度解码器，它能够按照指定的长度对消息进行相应的拆包

## Netty长连接、心跳机制

TCP的长连接和短连接

我们知道 TCP 在进行读写之前，server 与 client 之间必须提前建立一个连接。建立连接的过程，需要我们常说的三次握手，释放/关闭连接的话需要四次挥手。这个过程是比较消耗网络资源并且有时间延迟的。

所谓，短连接说的就是 server 端与 client 端建立连接之后，读写完成之后就关闭掉连接，如果下一次再要互相发送消息，就要重新连接。短连接的有点很明显，就是管理和实现都比较简单，缺点也很明显，每一次的读写都要建立连接必然会带来大量网络资源的消耗，并且连接的建立也需要耗费时间。

长连接说的就是 client 向 server 双方建立连接之后，即使 client 与 server 完成一次读写，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。长连接的可以省去较多的 TCP 建立和关闭的操作，降低对网络资源的依赖，节约时间。对于频繁请求资源的客户来说，非常适用长连接

### 心跳机制

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`。

## Netty 的零拷贝

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。