

# 网络模型/Netty

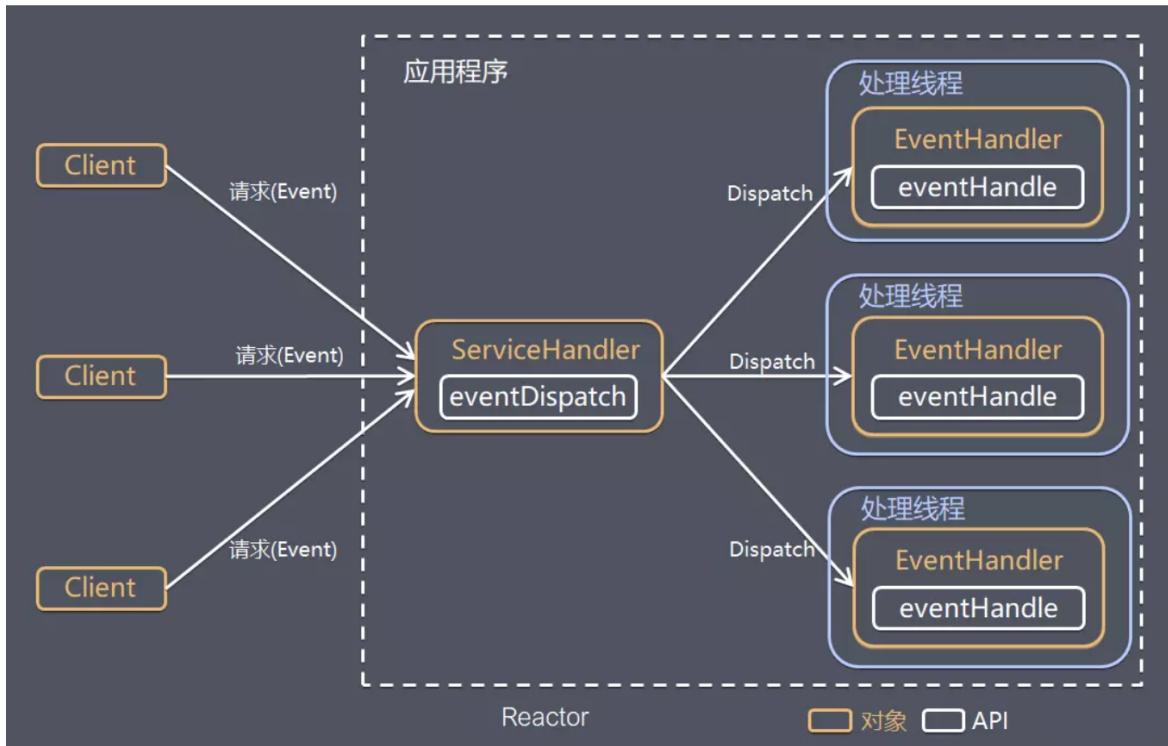
---

- 网络模型
  - 一次请求过程
    - 1) 客户端请求：通过网卡，读取客户端的请求数据，将数据读取到内核缓冲区
    - 2) 获取请求数据：服务器从内核缓冲区读取数据到进程缓冲区
    - 3) 处理业务：web服务进程处理请求构建响应
    - 4) 服务端返回数据：将已构建好的响应从用户缓冲区写入系统缓冲区
    - 5) 发送给客户端：内核通过网络I/O，将内核缓冲区的数据写入网卡，网卡通过底层的通讯协议将数据发给目标客户端
  - 并发模型关键点
    - 服务器如何管理连接并获取输入数据
    - 服务器如何处理请求
  - read/write，内核缓冲区/进程缓冲区
    - read系统调用，是把数据从内核缓冲区复制到进程缓冲区；而write系统调用，是把数据从进程缓冲区复制到内核缓冲区。这个两个系统调用，都不负责数据在内核缓冲区和磁盘之间的交换。底层的读写交换，是由操作系统kernel内核完成的。
    - 缓冲区的目的，是为了减少频繁的系统IO调用。大家都知道，系统调用需要保存之前的进程数据和状态等信息，而结束调用之后回来还需要恢复之前的信息，为了减少这种损耗时间、也损耗性能的系统调用，于是出现了缓冲区。
    - 有了缓冲区，操作系统使用read函数把数据从内核缓冲区复制到进程缓冲区，write把数据从进程缓冲区复制到内核缓冲区中。等待缓冲区达到一定数量的时候，再进行IO的调用，提升性能。至于什么时候读取和存储则由内核来决定，用户程序不需要关心。
    - 在linux系统中，系统内核也有个缓冲区叫做内核缓冲区。每个进程有自己独立的缓冲区，叫做进程缓冲区。所以，用户的IO读写程序，大多数情况下，并没有进行实际的IO操作，而是在读写自己的进程缓冲区。
  - 阻塞/非阻塞，同步/异步
    - 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回（内核IO操作彻底完成后，才返回到用户空间，执行用户的操作）
    - 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程（用户程序不需要等待内核IO操作完成后）
    - 同步处理是指被调用方得到最终结果之后才返回给调用方
    - 异步处理是指被调用方先返回应答，然后再计算调用结果，计算完最终结果后再通知并返回给调用方
  - IO操作的两个阶段（阶段一为阻塞，阶段二为同步）

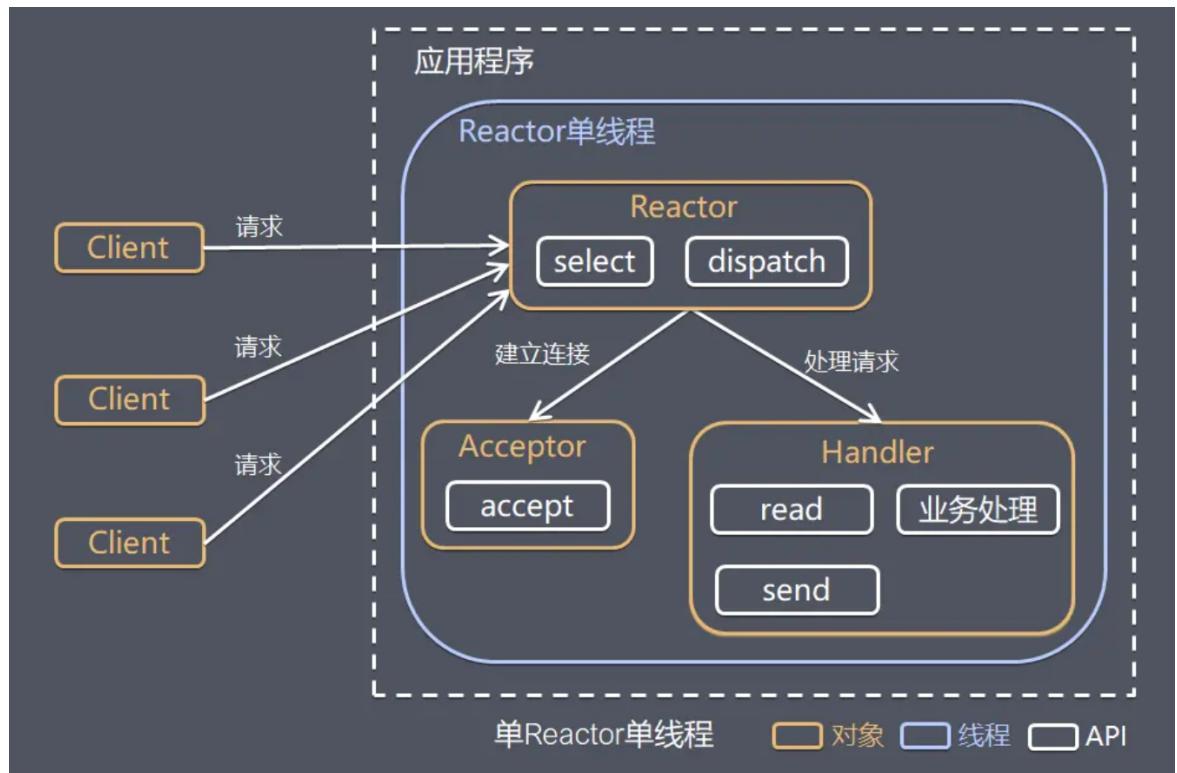
- (1) 等待数据准备
  - (2) 数据从内核空间拷贝到用户空间
- **五种IO模型**
    - IO模型就是指用什么样的通道进行数据的发送和接收，Java**共支持三种**网络编程IO模式：**BIO、NIO、AIO**
    - 阻塞IO
      - 从进程发起IO操作，一直等待上述两个阶段完成。两阶段一起阻塞。
    - 非阻塞IO
      - 返回一个错误而不是挂起，进程一直询问IO准备好了没有，准备好了再发起读取操作，这时才把数据从内核空间拷贝到用户空间。第一阶段不阻塞但要轮询，第二阶段阻塞。
    - **多路复用IO**
      - 多个连接使用同一个selector去询问IO准备好了没有，如果有准备好了的，就返回有数据准备好了，然后对应的连接再发起读取操作，把数据从内核空间拷贝到用户空间。两阶段分开阻塞。
    - **信号驱动IO**
      - 进程发起读取操作会立即返回，当数据准备好了会以通知的形式（发送一个sigio信号）告诉进程，进程再发起读取操作，把数据从内核空间拷贝到用户空间。第一阶段不阻塞，第二阶段阻塞。
    - 异步IO
      - 进程发起读取操作会立即返回，等到数据准备好且已经拷贝到用户空间了再通知进程拿数据。两个阶段都不阻塞。
  - **IO多路复用**
    - I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select, pselect, poll, epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。
    - 文件描述符
      - 用于表述指向文件的引用的抽象化概念。
    - select
      - select函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪（有数据可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。
      - 优点：跨平台支持
      - 缺点：1.单个进程打开的fd有限制，1024

- 缺点：2. 线性扫描，采用的是轮询的方法，效率较低
- **Poll**
  - 不同与select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。
  - 本质与select一样，将用户传入的数据拷贝到内核空间，然后查询每个fd对应的设备状态，没有发现则挂起之道就绪或者超时，被唤醒后再次轮询。没有最大连接数，原因是它是基于链表来存储的。
  - select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。
- **epoll**
  - 事件响应
  - epoll是用一个文件描述符管理多个描述符，将用户关系文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需要一次。
  - epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。
  - epoll 操作
    - epoll\_create
      - 创建一个epoll句柄，即一个总的文件描述符
    - epoll\_ctl
      - 创建需要监听的fd（文件描述符）
    - epoll\_wait
      - 等待io事件，通过callback来激活
  - 优点
    - 没有最大并发连接的限制
    - 只有活跃的的fd才可以调用callback函数
    - epoll使用mmap减少复制开销
- **Reactor模式**
  - I/O复用结合线程池，这就是Reactor模式基本设计思想
  - 基于I/O复用模型，多个连接共用一个阻塞对象，应用程序只需要在一个阻塞对象上等待，无需阻塞等待所有连接。当某条连接有新的数据可以处理时，操作系统通知应用程序，线程从阻塞状态返回，开始进行业务处理
  - 基于线程池复用线程资源，不必再为每个连接创建线程，将连接完成后的业务处理任务分配给线程进行处理，一个线程可以处理多个连接的业务
  - **Reactor模式也叫Dispatcher模式**，即I/O多了复用统一监听事件，收到事件后分发(Dispatch给某进程)，是编写高性能网络服务器的必备技术之一
  - 示例图

- 示例图



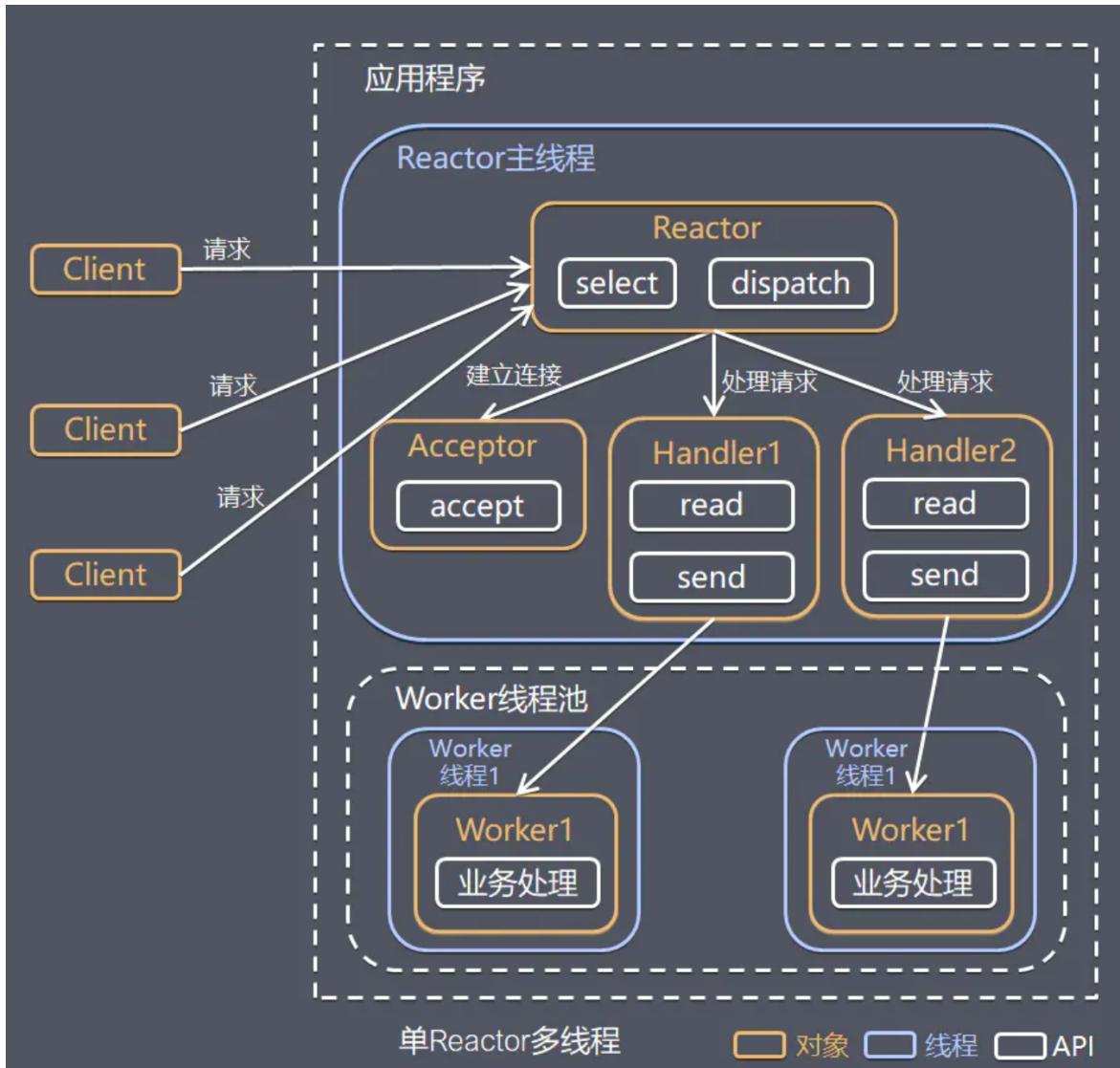
- Reactor模式中有2个关键组成：
  - Reactor: Reactor在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理器来对I/O事件做出反应。它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人
  - Handlers: 处理程序执行I/O事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor通过调度适当的处理器来响应I/O事件，处理器执行非阻塞操作
- Reactor的三种实现
  - 1) 单Reactor单线程
    - select是前面I/O复用模型介绍的标准网络编程API，可以实现应用程序通过一个阻塞对象监听多路连接请求
    - 1) Reactor对象通过select监控客户端请求事件，收到事件后通过dispatch进行分发
    - 2) 如果是建立连接请求事件，则由Acceptor通过accept处理连接请求，然后创建一个Handler对象处理连接完成后的后续业务处理
    - 3) 如果不是建立连接事件，则Reactor会分发调用连接对应的Handler来响应
    - 4) Handler会完成read->业务处理->send的完整业务流程
    - 优点：模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成
    - 缺点1.性能问题：只有一个线程，无法完全发挥多核CPU的性能Handler在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶颈 2.可靠性问题：线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障
    - 使用场景Redis
- 示例图



- 2) 单Reactor多线程

- 1.Reactor对象通过select监控客户端请求事件，收到事件后通过dispatch进行分发
- 2.如果是建立连接请求事件，则由Acceptor通过accept处理连接请求，然后创建一个Handler对象处理连接完成后的续各种事件
- 3.如果不是建立连接事件，则Reactor会分发调用连接对应的Handler来响应
- 4.Handler只负责响应事件，不做具体业务处理，通过read读取数据后，会分发给后面的Worker线程池进行业务处理
- 5.Worker线程池会分配独立的线程完成真正的业务处理，如何将响应结果发给Handler进行处理
- 6.Handler收到响应结果后通过send将响应结果返回给client

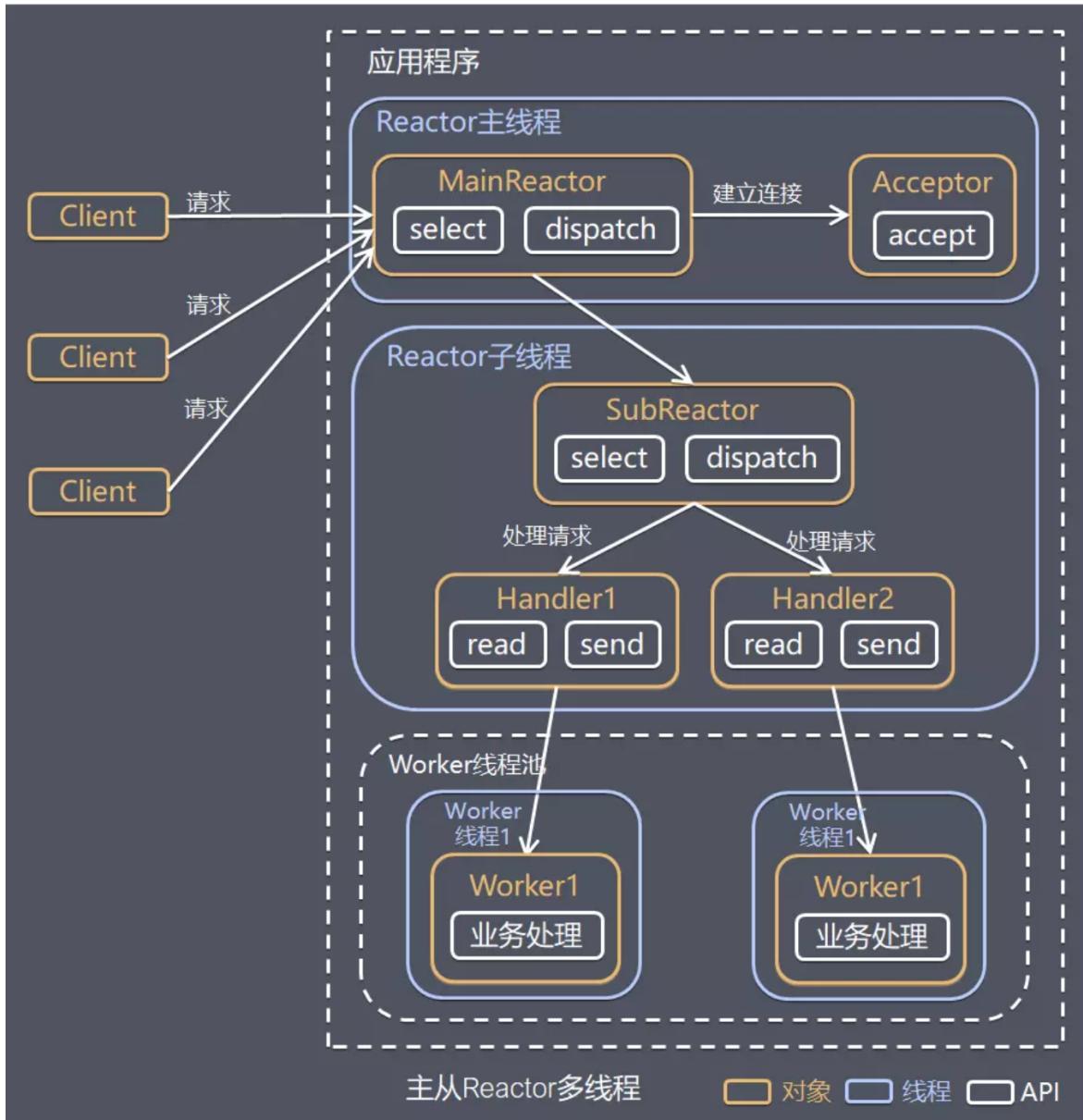
- 示例图



### • 3) 多Reactor多线程

- 针对单Reactor多线程模型中，Reactor在单线程中运行，高并发场景下容易成为性能瓶颈，可以让Reactor在多线程中运行
- 1.Reactor主线程MainReactor对象通过select监控建立连接事件，收到事件后通过Acceptor接收，处理建立连接事件
- 2.Acceptor处理建立连接事件后，MainReactor将连接分配Reactor子线程给SubReactor进行处理
- 3.SubReactor将连接加入连接队列进行监听，并创建一个Handler用于处理各种连接事件
- 4.当有新的事件发生时，SubReactor会调用连接对应的Handler进行响应
- 5.Handler通过read读取数据后，会分发给后面的Worker线程池进行业务处理
- 6.Worker线程池会分配独立的线程完成真正的业务处理，如何将响应结果发给Handler进行处理
- 7.Handler收到响应结果后通过send将响应结果返回给client
- 优点
- 父线程与子线程的数据交互简单职责明确，父线程只需要接收新连接，子线程完成后续的业务处理

- 父线程与子线程的数据交互简单，Reactor主线程只需要把新连接传给子线程，子线程无需返回数据
- 场景：Netty
- 示例图
- 



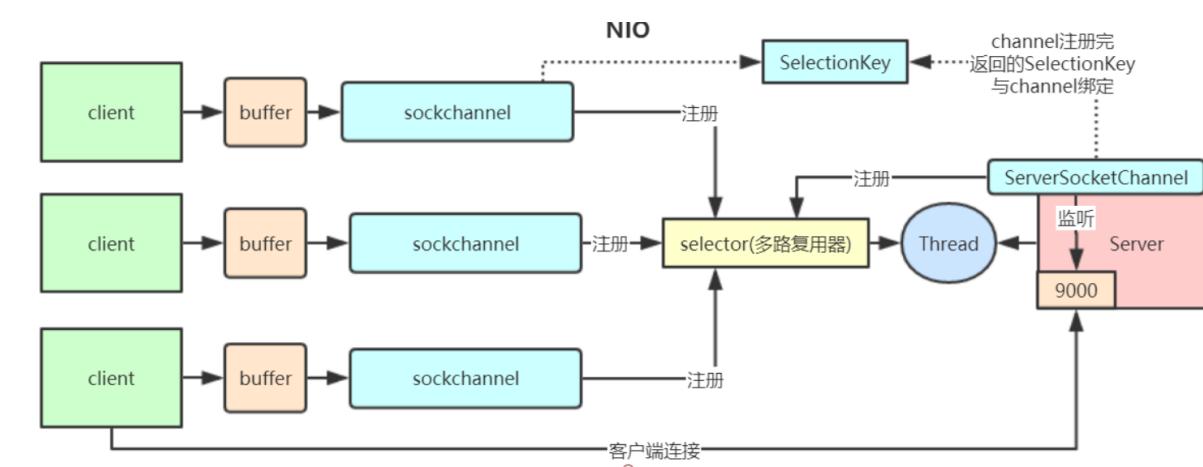
- Reactor优点
  - 响应快，不必为单个同步时间所阻塞，虽然Reactor本身依然是同步的
  - 编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销；
  - 可扩展性，可以方便的通过增加Reactor实例个数来充分利用CPU资源
  - 可复用性，Reactor模型本身与具体事件处理逻辑无关，具有很高的复用性
- BIO(Blocking IO)同步阻塞
  - 同步阻塞模型，一个客户端连接对应一个处理线程
  - 缺点：

- 1、IO里面read操作是阻塞操作，如果连接不做数据读写操作会导致线程阻塞，浪费资源
- 2、如果线程很多会导致服务器线程太多压力太大
- 应用场景：BIO方式适用于连接数目少且固定的架构，这种方式对服务器资源要求比较高，但程序简单易理解
- 相关代码：
  - `Socket socket=serverSocket.accept(); // 阻塞方法`
  - `int read = socket.getInputStream().read(bytes); // 接受客户端数据，阻塞方法，没有数据可读时就阻塞`
- NIO(Non Blocking IO)同步非阻塞
  - 同步非阻塞，服务器实现模式为一个线程可以处理多个请求（连接），客户端发送的连接请求都会注册到多路复用器selector上，多路复用器轮询到连接有IO请求就进行处理
  - Reactor 就是基于NIO中实现多路复用的一种模式，Java的NIO就是使用多路复用的IO模型
  - I/O多路复用底层一般用的Linux API (`select`, `poll`, `epoll`) 来实现，他们的区别如下表：
    - JDK1.4之前开始支持NIO；底层使用的是`select`、`poll`；jdk1.5之后使用的是`epoll`模型
    - 三者比较

	<b>select</b>	<b>poll</b>	<b>epoll(jdk 1.5及以上)</b>
<b>操作方式</b>	遍历	遍历	回调
<b>底层实现</b>	数组	链表	哈希表
<b>IO效率</b>	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当有IO事件就绪，系统注册的回调函数就会被调用，时间复杂度O(1)
<b>最大连接</b>	有上限	无上限	无上限

- 应用场景：NIO方式适用于连接数目多且比较短（短操作）的架构，比如聊天服务器，弹幕系统，服务器间通讯，编程比较复杂
- NIO有三大组件：Channel(通道), Buffer(缓冲区), Selector(选择器)
- NIO服务端代码详细分析
  - 1、创建一个 `ServerSocketChannel` 和 `Selector`，并将 `ServerSocketChannel` 注册到 `Selector` 上
  - 2、`selector` 通过 `select()` 方法监听 channel 事件，当客户端连接时，`selector` 监听到连接事件，获取到 `ServerSocketChannel` 注册时绑定的 `selectionKey`
  - 3、`selectionKey` 通过 `channel()` 方法可以获取绑定的 `ServerSocketChannel`
  - 4、`ServerSocketChannel` 通过 `accept()` 方法得到客户端的 `SocketChannel`

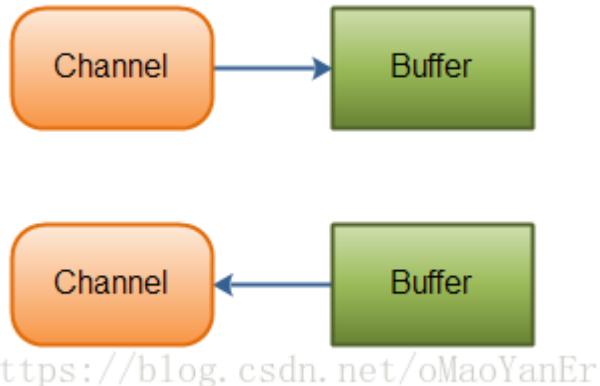
- 5、将 SocketChannel 注册到 Selector 上，关心 read 事件
- 6、注册后返回一个 SelectionKey，会和该 SocketChannel 关联
- 7、selector 继续通过 select() 方法监听事件，当客户端发送数据给服务端，selector 监听到read事件，获取到 SocketChannel 注册时绑定的 selectionKey
- 8、selectionKey 通过 channel() 方法可以获取绑定的 socketChannel
- 9、将 socketChannel 里的数据读取出来
- 10、用 socketChannel 将服务端数据写回客户端
- 总结：NIO模型的selector就像一个大总管，负责监听各种IO事件，然后转交给后端线程去处理
- **NIO流程总结**
  - NIO模型的selector就像一个大总管，负责监听各种IO事件，然后转交给后端线程去处理
  - NIO相对于BIO非阻塞的体现就在，BIO的后端线程需要阻塞等待客户端写数据(比如 read方法)，如果客户端不写数据线程就要阻塞，NIO把等待客户端操作的事情交给了大总管 selector，selector 负责轮询所有已注册的客户端，发现有事件发生了才转交给后端线程处理，后端线程不需要做任何阻塞等待，直接处理客户端事件的数据即可，处理完马上结束，或返回线程池供其他客户端事件继续使用。还有就是 channel 的读写是非阻塞的
  - 外层一个while循环，selector.select()方法阻塞住，当客户端有操作执行时，往下执行，复制出所有SelectionKey后，遍历selector监听的所有Key进行处理
- **详细流程图**



- **AIO(Asynchronous Blocking IO)异步非阻塞**
  - accept()不会阻塞，回调函数； read()不会同步
  - 异步非阻塞，由操作系统完成后回调通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用
  - JDK1.7开始支持
- **BIO、NIO、AIO对比**
  -

	BIO	NIO	AIO
IO 模型	同步阻塞	同步非阻塞（多路复用）	异步非阻塞
编程难度	简单	复杂	复杂
可靠性	差	好	好
吞吐量	低	高	高

- Channel和Buffers、NIO堆外内存
  - NIO的堆外内存DirectByteBuffer
    - 基于 channel 和 Buffer 的 I/O 方式，可用 Native 库直接分配堆外内存，然后利用一个存储在堆中的 DirectByteBuffer 对象作为这块内存引用来操作。避免了在 Java 堆和 Native 堆中来回复制数据。
  - HeapByteBuffer和DirectByteBuffer区别
    - heapByBuffer在堆上分配的，但是Java nio在读写到相应的Channel的时候，会将heap上的buffer拷贝到直接内存中（direct memory），增加额外的开销。
- Channel
  - JavaNIO Channels和流有一些相似，但是又有些不同：
    - 你可以同时读和写Channels，流Stream只支持单向的读或写（InputStream/OutputStream）
    - Channels可以异步的读和写，流Stream是同步的
    - Channels总是读取到buffer或者从buffer中写入
  - channel到channel的数据传输
    - 在Java NIO中我们可以直接将数据从一个Channel传输到另一个Channel中，比如 FileChannel中有transferTo()和transferFrom()方法。
  - Channel的一些实现类：
    - FileChannel : 可以读写文件中的数据
    - DatagramChannel: 可以通过UDP协议读写数据
    - SocketChannel: 可以通过TCP协议读写数据
    - ServerSocketChannel: 允许我们像一个web服务器那样监听TCP链接请求，为每一个链接请求创建一个SocketChannel
- Buffer
  - 在Java NIO中各类Buffer主要用于和NIO Channel进行交互，数据从Channel中读取到Buffer中，从Buffer写入到Channel中。



<https://blog.csdn.net/oMaoYanEr>

- 使用Buffer进行读写数据一般会通过下边四个步骤处理：
  - 将数据写到Buffer中
  - 调用buffer.flip()切换为读模式
  - 从Buffer中读取数据
  - 调用buffer.clear()或者buffer.compact()清空或压缩buffer
- 当我们将数据写入buffer时，buffer会记录我们写入了多少数据，当需要读取数据的时候，需要调用flip()方法将buffer从写模式切换到读模式，在读模式下，buffer允许用户读取已经写入buffer的所有数据。
- 一旦我们已经读取了buffer中的所有数据，我们需要清空buffer以便写一次写入数据。我们可以使用两种方法达到这个目的：
  - 调用clear()方法：清空整个buffer；
  - 调用compact()方法：仅清空已经读取的数据，未读取的数据移动到buffer的起始位置，新写入的数据会放到未读取数据的后边。
- Netty使用场景、NIO
  - NIO的类库和API繁杂，使用麻烦，需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等
  - 开发工作量和难度非常大：例如客户端会面临断连重连、网络闪断、心跳处理、半包读写、网络拥塞和异常流的处理等
  - Netty对JDK自带的NIO的API进行了良好的封装，解决了上述问题。且Netty拥有高性能、吞吐量高、延迟低、资源消耗少、最小化不必要的内存复制等优点
  - **使用场景**
    - 1、互联网行业：在分布式系统中各个节点间需要远程服务调用，需要高性能的rpc框架。Netty作为异步高性能的通信框架，往往作为基础通信组件被这些rpc框架使用。如Dubbo、rocketmq等
    - 2、游戏行业：Netty作为高性能的基础通信组件，它本身提供了TCP/UDP和HTTP协议栈
    - 3、大数据领域：经典的Hadoop的高性能通信和序列化Avro的rpc框架，默认采用Netty进行跨界点通信，它的Netty Service是基于Netty框架的二次封装实现
- Netty使用流程
  - Server端

- 创建 EventLoopGroup。
- 创建 ServerBootstrap。
- 指定所使用的 NIO 传输 Channel。
- 使用指定的端口设置套接字地址。
- 添加一个 ServerHandler 到 Channel 的 ChannelPipeline。
- 异步地绑定服务器；调用 sync() 方法阻塞等待直到绑定完成。
- 获取 Channel 的 CloseFuture，并且阻塞当前线程直到它完成。
- 关闭 EventLoopGroup，释放所有的资源。

```

public void start() throws Exception {
    final EchoServerHandler serverHandler = new EchoServerHandler();
    // (1) 创建EventLoopGroup
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        // (2) 创建ServerBootstrap
        ServerBootstrap b = new ServerBootstrap();
        b.group(group)
            // (3) 指定所使用的 NIO 传输 Channel
            .channel(NioServerSocketChannel.class)
            // (4) 使用指定的端口设置套接字地址
            .localAddress(new InetSocketAddress(port))
            // (5) 添加一个EchoServerHandler到于Channel的 ChannelPipeline
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(serverHandler);
                }
            });
        // (6) 异步地绑定服务器；调用 sync()方法阻塞等待直到绑定完成
        ChannelFuture f = b.bind().sync();
        System.out.println(EchoServer.class.getName() +
            " started and listening for connections on " + f.channel().localAddress());
        // (7) 获取 Channel 的CloseFuture，并且阻塞当前线程直到它完成
        f.channel().closeFuture().sync();
    } finally {
        // (8) 关闭 EventLoopGroup，释放所有的资源
        group.shutdownGracefully().sync();
    }
}

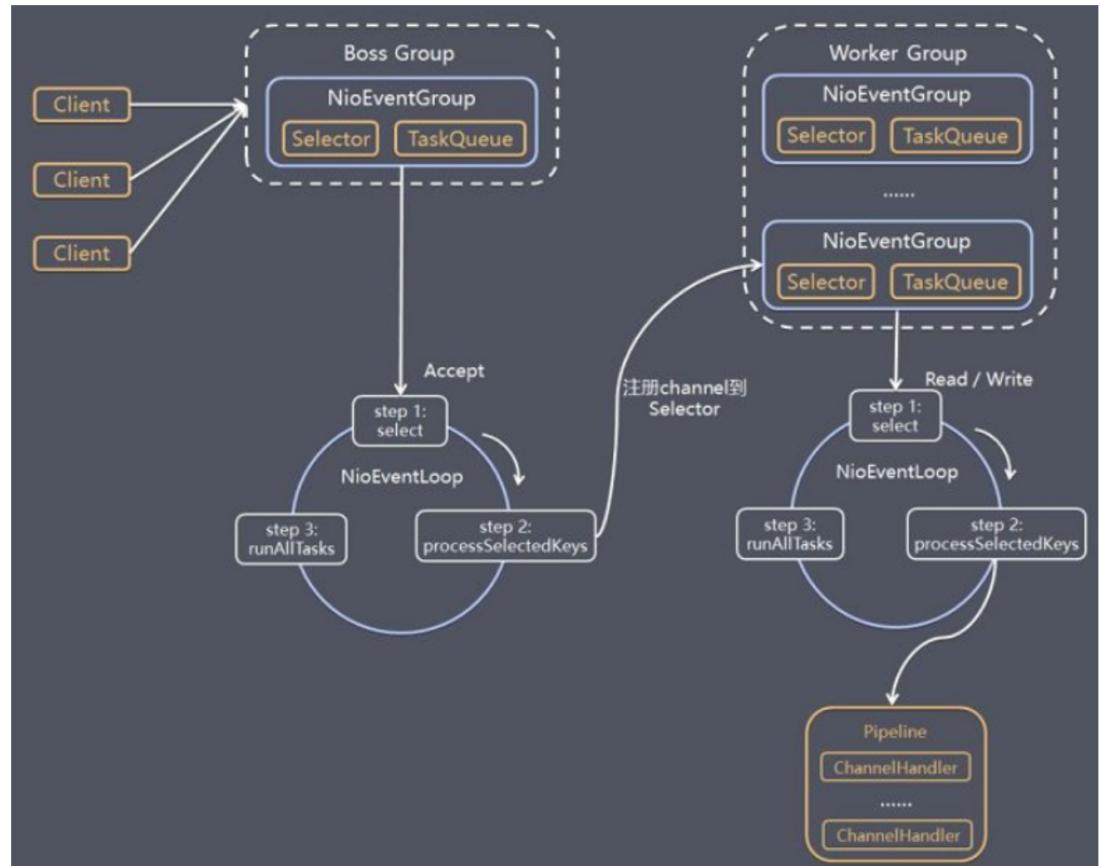
```

- 客户端

- 客户端和服务端的代码基本相似，在初始化时需要输入服务端的 IP 和 Port。
- 客户端启动程序的顺序：
- 创建 Bootstrap。
- 指定 EventLoopGroup 用来监听事件。
- 定义 Channel 的传输模式为 NIO（Non-BlockingInputOutput）。
- 设置服务器的 InetSocketAddress。
- 在创建 Channel 时，向 ChannelPipeline 中添加一个 EchoClientHandler 实例。
- 连接到远程节点，阻塞等待直到连接完成。
- 阻塞，直到 Channel 关闭。
- 关闭线程池并且释放所有的资源。

- 客户端在完成以上操作以后，会与服务端建立连接从而传输数据。同样在接受到 Channel 中触发的事件时，客户端会触发对应事件的操作。
- Netty高性能设计
  - 高性能主要来源于其I/O模型和线程处理模型
    - 由于读写操作都是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁 I/O阻塞导致的线程挂起，一个I/O线程可以并发处理N个客户端连接和读写操作，这从根本上解决了传统同步阻塞I/O—连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。**I/O复用模型**
    - 在I/O复用模型中，会用到select，这个函数也会使进程阻塞，但是和阻塞I/O所不同的，这两个函数可以同时阻塞多个I/O操作，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数
  - 基于buffer
    - 传统的I/O是面向字节流或字符流的，以流式的方式顺序地从一个Stream 中读取一个或多个字节，因此也就不能随意改变读取指针的位置。
    - 在NIO中，抛弃了传统的 I/O流，而是引入了Channel和Buffer的概念. 在NIO中，**只能从 Channel中读取数据到Buffer中或将数据 Buffer 中写入到 Channel**。基于buffer操作不像传统IO的顺序操作，NIO 中可以随意地读取任意位置的数据
    - 那么什么是 基于流 呢? 在一般的 Java IO 操作中，我们以流式的方式顺序地从一个 Stream 中读取一个或多个字节，因此我们也就**不能随意改变读取指针的位置**.
  - Netty线程模型
    - Netty主要基于**主从Reactors多线程模型**做了一定的修改，借用了MainReactor和 SubReactor的结构，但是实际实现上， SubReactor和Worker线程在同一个线程池中 WorkerGroup， bossGroup 和workerGroup是Bootstrap构造方法中传入的两个对象，这两个group均是线程池。bossGroup线程池则只是在bind某个端口后，获得其中一个线程作为**MainReactor**，专门处理端口的accept事件，每个端口对应一个boss线程， workerGroup线程池会被各个SubReactor和worker线程充分利用
    - Netty抽象出两组线程池，其中BossGroup负责接收客户端的连接，WorkerGroup专门负责网络的读写
    - BossGroup和WorkerGroup的类型都是NioEventLoopGroup
    - NioEventLoopGroup相当于是**一个事件循环线程组**，这个组中包含有多个事件循环线程，每一个事件循环线程都是NioEventLoop
    - 每一个NioEventLoop都有一个selector，用于监听注册在其上的socketChannel的网络通讯
    - BossGroup中的NioEventLoop线程内部循环的3个步骤
      - 1.处理accept事件，与client建立连接，生成NioSocketChannel
      - 2.将NioSocketChannel注册到某个worker NioEventLoop上的selector中
      - 3.处理任务队列中的任务，即runAllTasks
    - WorkerGroup中NioEventLoop线程执行循环的3个步骤

- 轮询注册到自己selector上的所有NioSocketChannel的read、write事件
- 处理I/O事件，即read/write事件，再对应的NioSocketChannel处理业务
- runAllTasks处理任务队列TaskQueue的任务，一些耗时的业务处理一般放在TaskQueue中慢慢处理，这样不影响数据在pipeline中的流动处理
- worker NioEventLoop处理NioSocketChannek业务时，会使用pipeline（管道），管道中维护了很多handler处理器用来处理channel中的数据
- 示例图
  - 示例



- Netty模块组件
  - Bootstrap、ServerBootstrap
    - Bootstrap的意思是引导，一个Netty应用通常有一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件，Netty中Bootstrap类是客户端程序的启动引导类，ServerBootstrap是服务端启动引导类
    - Bootstrap是Netty提供的一个便利的工厂类，我们可以通过它完成客户端和服务端的Netty初始化
  - Future、ChannelFuture
    - 在Netty中所有的IO操作都是异步的，不能立刻得知消息是否被正确处理，但是可以注册一个监听来查看，具体实现是通过Future和ChannelFuture，它们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件
  - Channel

- 1个Connection=1个Socket=1个Channel可以看作是等价的，只是在不同场景之下出现，是实体与实体之间的连接
- Channel 就是代表连接，实体之间的连接，程序之间的连接，文件之间的连接，设备之间的连接。同时它也是数据入站和出站的载体。
- 当客户端和服务端连接的时候会建立一个 Channel。这个 Channel 我们可以理解为 Socket 连接，它负责基本的 IO 操作，例如：bind ()，connect ()，read ()，write () 等等。
  - 1) 当前网络连接的通信状态（例如是否打开？是否已连接）
  - 2) 网络提供的配置参数（例如接受缓冲区的大小）
  - 3) 提供异步的网络I/O操作（如建立连接读写绑定端口），异步调用意味着任何 I/O 调用都将
- channel和Stream的区别
  - Channel可以同时支持读和写，而Stream只能支持单向的读或写（所以分成 InputStream 和 OutputStream）
  - Channel 支持异步读写， Stream 通常只支持同步
  - Channel 总是读向（read into） Buffer，或者写自（write from） Buffer
- ByteBuf
  - Netty 将 ByteBuf 作为数据容器，来存放数据。
  - 从结构上来说，ByteBuf 由一串字节数组构成。数组中每个字节用来存放信息。ByteBuf 提供了两个索引，一个用于读取数据，一个用于写入数据。这两个索引通过在字节数组中移动，来定位需要读或者写信息的位置。
    - 当从 ByteBuf 读取时，它的 readerIndex（读索引）将会根据读取的字节数递增。
    - 同样，当写 ByteBuf 时，它的 writerIndex 也会根据写入的字节数进行递增。
  - Buffer 用于与 Channel 交互时使用，通过上一章的学习我们知道，数据从 Channel 读取到 Buffer，或者从 Buffer 写入 Channel。
  - Buffer 本质上是一个内存块，可以向里面写入数据，或者从里面读取数据，在 Java 中它被包装成了 Buffer 对象，并提供了一系列的方法用于操作这个内存块。
  - channel -> read into -> Buffer
  - channel <- write from <- Buffer
- Selector
  - Netty 基于 Selector 对象实现 I/O 多路复用，通过 Selector，一个线程可以监听多个连接的 Channel 事件，当向一个 Selector 中注册 Channel 后，Selector 内部的机制就可以自动不断地查询(select) 这些注册的 Channel 是否有已就绪的 I/O 事件（例如可读，可写，网络连接完成等），这样程序就可以很简单地使用一个线程高效地管理多个 Channel。
  - selector 与 channel
    - Selector 通过不断轮询的方式同时监听多个 Channel 的事件，注意，这里是同时监听，一旦有 Channel 准备好了，它就会返回这些准备好了的 Channel，交给处理线程去处理。

- 所以，在NIO编程中，通过Selector我们就实现了一个线程同时处理多个连接请求的目标，也可以一定程度降低服务器资源的消耗。
- NioEventLoop
  - 消息的“出站”/“入站”就会产生事件（Event）。有了数据，数据的流动产生事件，那么就有一个机制去监控和协调事件。
  - 在Netty中每个Channel都会被分配到一个EventLoop。一个EventLoop可以服务于多个Channel。
  - 每个EventLoop会占用一个Thread，同时这个Thread会处理EventLoop上面发生的所有IO操作和事件（Netty 4.0）。
  - 在异步传输的情况下，一个EventLoop是可以处理多个Channel中产生的事件的，它主要的工作就是事件的发现以及通知。
  - NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用NioEventLoop的run方法，执行I/O任务和非I/O任务：
    - I/O任务即selectionKey中ready的事件，如accept、connect、read、write等，由processSelectedKeys方法触发。
    - 非I/O任务添加到taskQueue中的任务，如register0、bind0等任务，由runAllTasks方法触发。
  - 两种任务的执行时间比由变量ioRatio控制，默认为50，则表示允许非I/O任务执行的时间与I/O任务的执行时间相等。
- NioEventLoopGroup
  - NioEventLoopGroup，主要管理eventLoop的生命周期，可以理解为一个线程池，内部维护了一组线程，每个线程(NioEventLoop)负责处理多个Channel上的事件，而一个Channel只对应于一个线程。
- 客户端到服务端成为出站，服务端到客户端是入站。
- ChannelHandler，ChannelPipeline 和 ChannelHandlerContext
  - 如果说EventLoop是事件的通知者，那么ChannelHandler就是事件的处理者。
  - 针对出站和入站的事件，有不同的ChannelHandler，分别是：
    - ChannelInBoundHandler（入站事件处理器）
    - ChannelOutBoundHandler（出站事件处理器）
  - 假设每次请求都会触发事件，而由ChannelHandler来处理这些事件，这个事件的处理顺序是由ChannelPipeline来决定的。
  - ChannelPipeline为ChannelHandler链提供了容器。到Channel被创建的时候，会被Netty框架自动分配到ChannelPipeline上。
  - ChannelPipeline和ChannelHandler，前者管理后者的排列顺序。那么它们之间的关联就由ChannelHandlerContext来表示了。ChannelHandlerContext的主要功能是管理ChannelHandler和ChannelPipeline的交互。
- ChannelHandler

- ChannelHandler是一个接口，处理I/O事件或拦截I/O操作，并将其转发到其ChannelPipeline(业务处理链)中的下一个处理程序。充当了处理入站和出站数据的应用程序逻辑容器
- ChannelHandler本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类：
  - ChannelInboundHandler用于处理入站I/O事件
  - ChannelOutboundHandler用于处理出站I/O操作
- ChannelHandlerContext
  - 保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象
- ChannelPipeline
  - 保存ChannelHandler的List，用于处理或拦截Channel的入站事件和出站操作。ChannelPipeline实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及Channel中各个的ChannelHandler如何相互交互。
- 一个Channel包含了一个ChannelPipeline，而ChannelPipeline中又维护了一个由ChannelHandlerContext组成的双向链表，并且每个ChannelHandlerContext中又关联着一个ChannelHandler。入站事件和出站事件在一个双向链表中，入站事件会从链表head往后传递到最后一个入站的handler，出站事件会从链表tail往前传递到最前一个出站的handler，两种类型的handler互不干扰。
- 编码解码器
  - 入站消息会被解码，从字节转换成另一种形式
  - 出站的消息会被编码成字节
- 在Netty中客户端请求服务端，被称为“入站”操作。
- 粘包拆包
  - TCP粘包拆包是指发送方发送的若干包数据到接收方接收时粘成一包或某个数据包被拆开接收。如下图所示，client发了两个数据包D1和D2，但是server端可能会收到如下几种情况的数据。
  - 为什么会出现粘包拆包的情况
    - TCP是面向连接的，面向流的，提供高可靠性服务。收发两端（客户端和服务器端）都要有成对的socket，因此，发送端为了将多个发给接收端的包，更有效的发给对方，使用了优化方法（Nagle算法），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。这样做虽然提高了效率，但是接收端就难于分辨出完整的数据包了，因为面向流的通信是无消息保护边界的。
  - 解决方案
    - 1) 格式化数据
      - 每条数据都有固定的格式（开始符、结束符），这种方法简单易行，但是需要注意内部可能出现开始符或者结束符的情况

- 2) 发送长度
  - 发送每条数据的时候，将数据的长度也一起发送，比如何以选择每条数据的前4位作为数据的长度，应用层处理时可以根据长度来判断每条数据的开始和结束
  - 这种方法更稳妥
- 心跳检测机制
  - 心跳是指在tcp长连接中客户端和服务端之间定期发送的一种特殊的数据包，作用是为了通知对方自己还在线，以确保tcp连接的有效性
  - 实现心跳机制的关键是IdleStateHandler
    - 读超时、写超时、读写超时
  - 要实现心跳机制需要在服务端的channelInitialize中加入代码
    - pipeline.addList(new IdleStateHandler(3,0,0, TimeUnit.SECONDS));
- 零拷贝
  - Netty的接收和发送ByteBuffer采用**DIRECT BUFFERS**，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。
  - 如果使用传统的JVM堆内存（**HEAP BUFFERS**）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才能写入Socket中。JVM堆内存的数据是不能直接写入Socket中的。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。
  - 直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，某些情况下这部分内存也会被频繁地使用，而且也可能导致OutOfMemoryError异常出现。Java里用**DirectByteBuffer**可以分配一块直接内存(堆外内存)，元空间对应的内存也叫作直接内存，它们对应的都是机器的物理内存。
  - 使用直接内存的优缺点：
    - 优点1：不占用堆内存空间，减少了发生GC的可能
    - 优点2：java虚拟机实现上，本地IO会直接操作直接内存更快（直接内存=>系统调用=>硬盘/网卡），而非直接内存则需要二次拷贝（堆内存=>直接内存=>系统调用=>硬盘/网卡）
    - 缺点：初始分配较慢
    - 缺点2：没有JVM直接帮助管理内存，容易发生内存溢出。为了避免一直没有FULL GC，最终导致直接内存把物理内存被耗完。我们可以指定直接内存的最大值，通过-XX: MaxDirectMemorySize来指定，当达到阈值的时候，调用system.gc来进行一次FULL GC，间接把那些没有被使用的直接内存回收掉。