1. 什么是Netty

官网地址：https://netty.io/

Netty 是一个 NIO 客户端服务器框架，它是基于NIO基础上的封装，本质上是提供高性能网络IO通信的功能,可以快速轻松的开发网络应用程序，例如协议服务器和客户端。它极大地简化了网络编程，例如 TCP 和 UDP 套接字服务器。

2. 应用场景

(1)作为 RPC 框架的网络通信工具,例如Dubbo

(2)实现一个自己的 HTTP 服务器,例如Tomcat

(3)实现一个即时通讯系统,例如微信

(4)消息推送系统，例如Rocketmq

3. 网络编程的发展史

1. BIO

2. NIO

3. 多路复用

4. AIO

5. Reactor模型

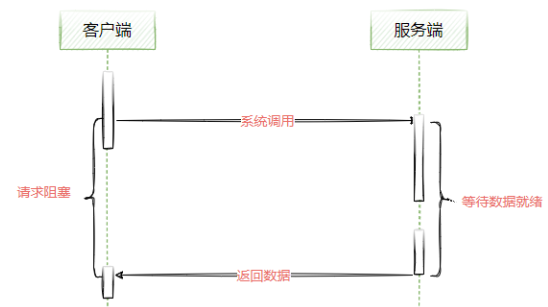
单线程单Reactor模型

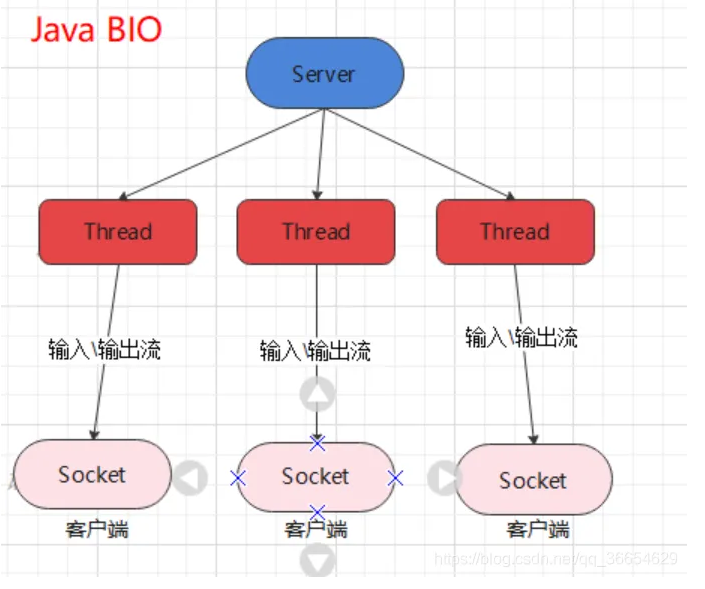
多线程单Reactor模型

主从多线程 Reactor模型

（1）同步阻塞IO(BIO)

如果客户端向服务端发送一个请求，客户端在收到服务端返回数据之前，一直处于阻塞状态，即客户端需要等待服务端的连接以及数据的返回，完成本次会话，这个过程就叫做同步阻塞IO(BIO)。



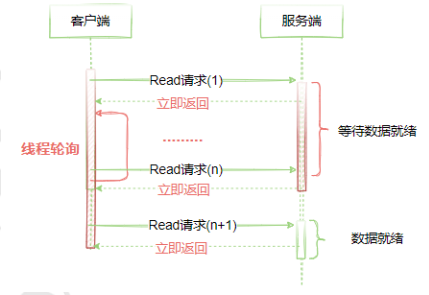


（2）非阻塞IO(NIO)

非阻塞IO，就是客户端向服务端发起请求时，如果服务端的数据未就绪的情况下， 客户端请求不会被阻塞，而是直接返回。有可能服务端的数据还未准备好的时候，客户端收到的返回是一个空的,所以客户端会通过轮询的方式获取结果。

NIO相比BIO来说，少了阻塞的过程在性能和连接数上都会有明显提高，但是轮询过程中会有很多空轮询，

而这个轮询会存在大量的系统调用（发起内核指令从网卡缓冲区中加载数据，用户空间到内核空间的切换），随着连接数量的增加，会导致性能问题。



（3） 多路复用机制

I/O多路复用的本质是通过一种机制（系统内核缓冲I/O数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作。

（linux一切皆文件，对于一个socket的读写也会有相应的文件描述符，成为socketfd）

常见的IO多路复用方式有【select、poll、epoll】。

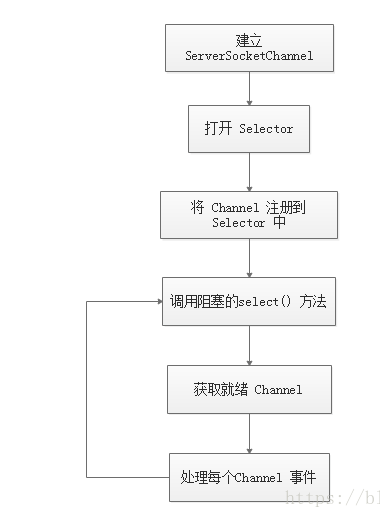
其中select和poll区别不大，主要就是减少了最大连接数的控制，select中最大连接数默认是1024。

那么接下来重点讲一下select、和epoll这两个模型：

- select：进程可以通过把一个或者多个fd传递给select系统调用，进程会阻塞在select操作上，这样select可以帮我们检测多个fd是否处于就绪状态。

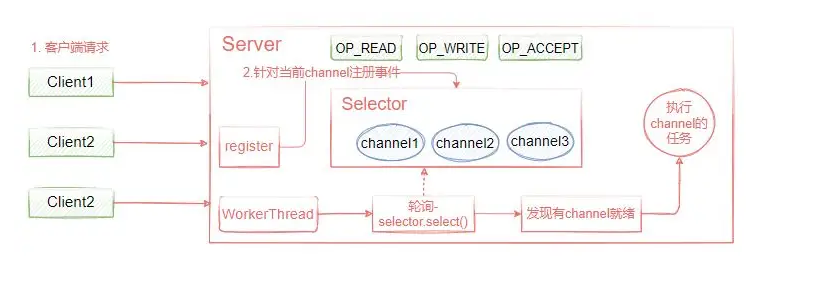
这个模式有两个缺点由于他能够同时监听多个文件描述符，假如说有1000个，这个时候如果其中一个fd 处于就绪状态了，那么当前进程需要线性轮询所有的fd，也就是监听的fd越多，性能开销越大。

同时，select在单个进程中能打开的fd是有限制的，默认是1024，对于那些需要支持单机上万的TCP连接来说确实有点少



- epoll：linux还提供了epoll的系统调用，epoll是基于事件驱动方式来代替顺序扫描，因此性能相对来说更高，

主要原理是，当被监听的fd中，有fd就绪时，会告知当前进程具体哪一个fd就绪，那么当前进程只需要去从指定的fd上读取数据即可，另外，epoll所能支持的fd上线是操作系统的最大文件句柄，这个数字要远远大于1024



（4） 异步IO

异步IO和多路复用机制，最大的区别在于：当数据就绪后，客户端不需要发送内核指令从内核空间读取数据，而是系统会异步把这个数据直接拷贝到用户空间，应用程序只需要直接使用该数据即可。

（5）Reactor模型

Reactor本质上就是基于NIO多路复用机制提出的一个高性能IO设计模式，它的核心思想是把响应IO事件和业务处理进行分离，通过一个或者多个线程来处理IO事件，然后将就绪得到事件分发到业务处理handlers线程去异步非阻塞处理。

Reactor模型有三个重要的组件：

\*\*Reactor\*\* ：将I/O事件发派给对应的Handler

\*\*Acceptor\*\* ：处理客户端连接请求

\*\*Handlers\*\* ：执行非阻塞读/写

单线程reactor模型

整体的I/O操作是由同一个线程完成的。Reactor线程，负责多路分离套接字，有新连接到来触发connect 事件之后，交由Acceptor进行处理，有IO读写事件之后交给hanlder 处理。

多线程单Reactor模型

单线程Reactor这种实现方式有存在着缺点，从实例代码中可以看出，handler的执行是串行的，如果其中一个handler处理线程阻塞将导致其他的业务处理阻塞。由于handler和reactor在同一个线程中的执行，这也将导致新的无法接收新的请求。为了解决这种问题，有人提出使用多线程的方式来处理业务，也就是在业务处理的地方加入线程池异步处理，将reactor和handler在不同的线程来执行，

多线程多reactro模型

在多线程单Reactor模型中，我们发现所有的I/O操作是由一个Reactor来完成，而Reactor运行在单个线程中，它需要处理包括Accept() / read() / write / connect 操作，对于小容量的场景，影响不大。但是对于高负载、大并发或大数据量的应用场景时，容易成为瓶颈。

所以，我们还可以更进一步优化，引入多Reactor多线程模式，如图所示，Main Reactor负责接收客户端的连接请求，然后把接收到的请求传递给SubReactor（其中subReactor可以有多个），具体的业务IO处理由SubReactor完成。

- Acceptor，请求接收者，在实践时其职责类似服务器，并不真正负责连接请求的建立，而只将其

请求委托 Main Reactor 线程池来实现，起到一个转发的作用。

- Main Reactor，主 Reactor 线程组，主要负责连接事件，并将IO读写请求转发到 SubReactor

线程池。

- Sub Reactor，Main Reactor 通常监听客户端连接后会将通道的读写转发到 Sub Reactor 线程池

中一个线程(负载均衡)，负责数据的读写。在 NIO 中 通常注册通道的读(OP\_READ)、写事件

(OP\_WRITE)。

Netty的基本使用

目前我们使用的Netty版本是4.x版本，之前有一段时间netty发布了一个5.x版本，但是被官方舍弃了，原因是：使用ForkJoinPool增加了复杂性，并且没有显示出明显的性能优势。同时保持所有的分支同步是相当多的工作，没有必要。

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-all</artifactId>

<vesion>4.1.66</version>

</dependency>

Netty相比于NIO原生API，它有以下特点：

1. 提供了高效的I/O模型、线程模型和事件处理机制

2. 提供了非常简单易用的API，相比NIO来说，针对基础的Channel、Selector、Sockets、Buffers等api提供了更高层次的封装，屏蔽了NIO的复杂性

3. 对数据协议和序列化提供了很好的支持

4. 稳定性，Netty修复了JDK NIO较多的问题，比如select空转导致的cpu消耗100%、TCP断线重连、keep-alive检测等问题。

5. 可扩展性在同类型的框架中都是做的非常好的，比如一个是可定制化的线程模型，用户可以在启动参数中选择Reactor模型、 可扩展的事件驱动模型，将业务和框架的关注点分离。

6. 性能层面的优化，作为网络通信框架，需要处理大量的网络请求，必然就面临网络对象需要创建和销毁的问题，这种对JVM的GC来说不是很友好，为了降低JVM垃圾回收的压力，引入了两种优化机制：

- 对象池复用

- 零拷贝技术

netty中的核心组件

ServerBootstrap 与 Bootstrap

Bootstrap， 客户端启动api，用来链接远程netty server，只绑定一个EventLoopGroup

ServerBootStrap，服务端监听api，用来监听指定端口，会绑定两个EventLoopGroup，

bootstrap组件可以非常方便快捷的启动Netty应用程序

Channel

Channel是 Java NIO的一个基本构造。可以看作是传入或传出数据的载体。因此，它可以被打开或关闭，连接或者断开连接。

ChannelHandler 与 ChannelPipeline

ChannelHandler 是对 Channel 中数据的处理器，这些处理器可以是系统本身定义好的编解码器，也可以是用户自定义的。这些处理器会被统一添加到一个 ChannelPipeline 的对象中，然后按照添加的顺序对 Channel 中的数据进行依次处理。

EventLoop 与 EventLoopGroup

EventLoop 定义了Netty的核心抽象，用来处理连接的生命周期中所发生的事件，在内部，将会为每个Channel分配一个EventLoop。Netty 为每个 Channel 分配了一个 EventLoop，用于处理用户连接请求、对用户请求的处理等所有事件。EventLoop 本身只是一个线程驱动，在其生命周期内只会绑定一个线程，让该线程处理一个 Channel 的所有 IO 事件。

EventLoop与线程的关系是 1:1

EventLoopGroup 是一个 EventLoop 池，包含很多的 EventLoop。

ChannelFuture

Netty 中所有的 I/O 操作都是异步的，即操作不会立即得到返回结果，所以 Netty 中定义了一个 ChannelFuture 对象作为这个异步操作的“代言人”，表示异步操作本身。如果想获取到该异步操作的返回值，可以通过该异步操作对象的addListener() 方法为该异步操作添加监 NIO 网络编程框架 Netty 听器，为其注册回调：当结果出来后马上调用执行。

ByteBuf

Netty 中ByteBuff等同于JAVA NIO中的ByteBuffer，但是功能上做了很多增强。

ByteBuff创建方式

1. 第一种，创建基于堆内存的ByteBuf

ByteBuf buffer=ByteBufAllocator.DEFAULT.heapBuffer(10);

2. 第二种，创建基于直接内存（堆外内存）的ByteBuf（默认情况下用的是这种）

ByteBufAllocator.DEFAULT.directBuffer(10);

Java中的内存分为两个部分，一部分是不需要jvm管理的直接内存，也被称为堆外内存。堆

外内存就是把内存对象分配在JVM堆以外的内存区域，这部分内存不是虚拟机管理，而是由

操作系统来管理，这样可以减少垃圾回收对应用程序的影响

3. 两者的优缺点

- 优点

直接内存（堆外内存）的好处是读写性能会高一些，如果数据存放在堆中，此时需要把Java堆空间的数据发送到远程服务器，首先需要把堆内部的数据拷贝到直接内存（堆外内存），然后再发送。如果是把数据直接存储到堆外内存中，发送的时候就少了一个复制步骤。

- 缺点

直接内存需要自己维护，防止内存溢出，而堆内存有JVM帮我们管理。

在ByteBuf中，有两个指针

- readerIndex： 读指针，每读取一个字节，readerIndex自增加1。ByteBuf里面总共witeIndex-readerIndex

个字节可读，当readerIndex和writeIndex相等的时候，ByteBuf不可读

- writeIndex： 写指针，每写入一个字节，writeIndex自增加1，直到增加到capacity后，可以触发

扩容后继续写入。

- ByteBuf中还有一个maxCapacity最大容量，默认的值是Integer.MAX\_VALUE ，当ByteBuf写入数

据时，如果容量不足时，会触发扩容，直到capacity扩容到maxCapacity。

1. Write相关方法

对于write方法来说，ByteBuf提供了针对各种不同数据类型的写入，比如

- writeChar，写入char类型

- writeInt，写入int类型

- writeFloat，写入float类型

- writeBytes， 写入nio的ByteBuffer

- writeCharSequence， 写入字符串

2. Reader相关方法

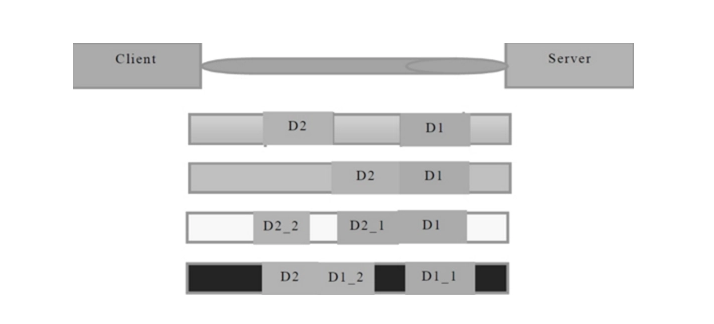
reader方法也同样针对不同数据类型提供了不同的操作方法

readByte ，读取单个字节

readInt ， 读取一个int类型

readFloat ，读取一个float类型

什么是 TCP 粘包/拆包



编解码器

1. FixedLengthFrameDecoder解码器

固定长度解码器FixedLengthFrameDecoder的原理很简单，就是通过构造方法设置一个固定消息大小frameLength，无论接收方一次收到多大的数据，都会严格按照frameLength进行解码。

如果累计读取的长度大小为frameLength的消息，那么解码器会认为已经获取到了一个完整的消息，如

果消息长度小于frameLength，那么该解码器会一直等待后续数据包的达到，直到获得指定长度后返

回。

2. DelimiterBasedFrameDecoder解码器

特殊分隔符解码器： DelimiterBasedFrameDecoder，它有以下几个属性

- delimiters，delimiters指定特殊分隔符，参数类型是ByteBuf，ByteBuf可以传递一个数组，意味着我们可以同时指定多个分隔符，但最终会选择长度最短的分隔符进行拆分。

比如接收方收到的消息体为

`hello\nworld\r\n`

此时指定多个分隔符\n 和\r\n ，那么最终会选择最短的分隔符解码，得到如下数据

`hello | world |`

- maxLength，表示报文的最大长度限制，如果超过maxLength还没检测到指定分隔符，将会抛出

TooLongFrameException。

- failFast，表示容错机制，它与maxLength配合使用。如果failFast=true，当超过maxLength后会

立刻抛出TooLongFrameException，不再进行解码。如果failFast=false，那么会等到解码出一个

完整的消息后才会抛出TooLongFrameException

- stripDelimiter，它的作用是判断解码后的消息是否去除分隔符，如果stripDelimiter=false，而

制定的特定分隔符是\n ，那么数据解码的方式如下。

`hello\nworld\r\n`

当stripDelimiter=false时，解码后得到

`hello\n | world\r\n`

3. LengthFieldBasedFrameDecoder解码器

LengthFieldBasedFrameDecoder是长度域解码器，它是解决拆包粘包最常用的解码器，基本上能覆盖

大部分基于长度拆包的场景。其中开源的消息中间件RocketMQ就是使用该解码器进行解码的。

首先来说明一下该解码器的核心参数

- lengthFieldOffset，长度字段的偏移量，也就是存放长度数据的起始位置

- lengthFieldLength，长度字段锁占用的字节数

- lengthAdjustment，在一些较为复杂的协议设计中，长度域不仅仅包含消息的长度，还包含其他

数据比如版本号、数据类型、数据状态等，这个时候我们可以使用lengthAdjustment进行修正，

它的值=包体的长度值-长度域的值

- initialBytesToStrip，解码后需要跳过的初始字节数，也就是消息内容字段的起始位置

- lengthFieldEndOffset，长度字段结束的偏移量， 该属性的值=lengthFieldOffset+lengthFieldLength

序列化

\*目的:把一个对象实现跨JVM、跨网络传输

Java序列化是指把Java对象转换为字节序列的过程，

Java反序列化是指把字节序列恢复为Java对象的过程

序列化选择考虑

1. 性能

- 空间-> 序列化之后的数据报文大小

- 时间-> 消耗的时间

2. 语言特性

- 是否支持多种开发语言

- 是否支持跨平台

3. 成熟度

4. 扩展性、兼容性

序列化技术

json （fastjson/gson/jackson）

xml

java

protobuf

kyro

avro

jute

messagepack

marshalling

thrift

hessian

hessian(dubbo)

hessian(sofa)

...

Protobuf序列化

Protobuf是Google的一种数据交换格式，它独立于语言、独立于平台。Google提供了多种语言来实

现，比如Java、C、Go、Python，每一种实现都包含了相应语言的编译器和库文件，Protobuf是一个纯

粹的表示层协议，可以和各种传输层协议一起使用。

Protobuf使用比较广泛，主要是空间开销小和性能比较好，非常适合用于公司内部对性能要求高的RPC

调用。 另外由于解析性能比较高，序列化以后数据量相对较少，所以也可以应用在对象的持久化场景中

但是要使用Protobuf会相对来说麻烦些，因为他有自己的语法，有自己的编译器，如果需要用到的话必

须要去投入成本在这个技术的学习中

protobuf有个缺点就是要传输的每一个类的结构都要生成对应的proto文件，如果某个类发生修

改，还得重新生成该类对应的proto文件

使用protobuf开发的一般步骤是

1. 配置开发环境，安装protocol compiler代码编译器

2. 编写.proto文件，定义序列化对象的数据结构

3. 基于编写的.proto文件，使用protocol compiler编译器生成对应的序列化/反序列化工具类

4. 基于自动生成的代码，编写自己的序列化应用