Netty

Netty

一、 网络编程基础原理

1 网络编程（Socket）概念

首先注意，Socket 不是 Java 中独有的概念，而是一个语言无关标准。任何可以实现网络编程的编程语言都有 Socket。

1.1 什么是 Socket

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个 *socket*。

建立网络通信连接至少要一个端口号。*socket* 本质是编程接口*(API)*，对 *TCP/IP* 的封装，*TCP/IP* 也要提供可供程序员做网络开发所用的接口，这就是 *Socket* 编程接口；*HTTP* 是轿车，提供了封装或者显示数据的具体形式；*Socket* 是发动机，提供了网络通信的能力。

Socket 的英文原义是“孔”或“插座”。作为 *BSD UNIX* 的进程通信机制，取后一种意思。通常也称作"套接字"，用于描述 *IP* 地址和端口，是一个通信链的句柄，可以用来实现不同虚拟机或不同计算机之间的通信。在 *Internet* 上的主机一般运行了多个服务软件，同时提供几种服务。每种服务都打开一个 *Socket*，并绑定到一个端口上，不同的端口对应于不同的服务。*Socket* 正如其英文原义那样，像一个多孔插座。一台主机犹如布满各种插座的房间，每个插座有一个编号，有的插座提供 *220* 伏交流电， 有的提供 *110* 伏交流电，有的则提供有线电视节目。 客户软件将插头插到不同编号的插座，就可以得到不同的服务。

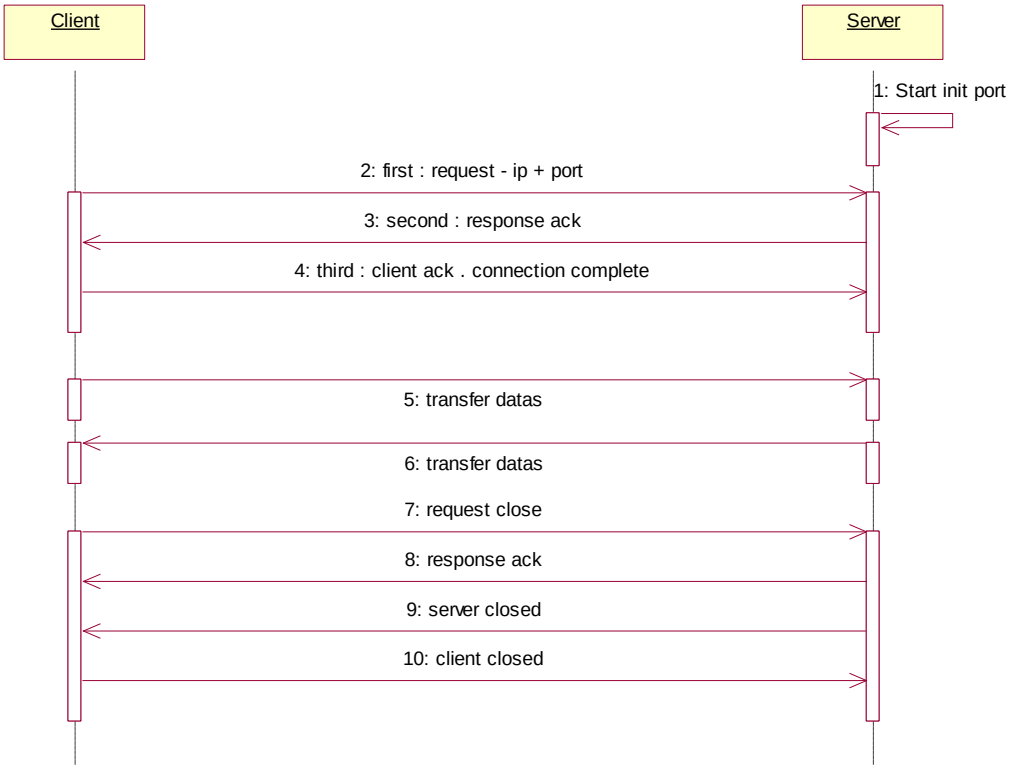
1.2 Socket 连接步骤

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。【如果包含数据交互+断开连接，那么一共是五个步骤】

（*1*）服务器监听：是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

（*2*）客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

（*3*）连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。



1.3 Java 中的 Socket

在 *java.net* 包是网络编程的基础类库。其中 *ServerSocket* 和 *Socket* 是网络编程的基础类型。*ServerSocket* 是服务端应用类型。*Socket* 是建立连接的类型。当连接建立成功后，服务器和客户端都会有一个 *Socket* 对象示例，可以通过这个 *Socket* 对象示例，完成会话的所有操作。

对于一个完整的网络连接来说，*Socket* 是平等的，没有服务器客户端分级情况。

2 什么是同步和异步

同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发 *IO* 操作并等待或者轮询的去查看 *IO* 操作是否就绪，而异步是指用户进程触发 *IO* 操作以后便开始做自己的事情，而当 *IO* 操作已经完成的时候会得到 *IO* 完成的通知。

以银行取款为例：

同步 ： 自己亲自出马持银行卡到银行取钱（使用同步 *IO* 时，*Java* 自己处理 *IO* 读写）；

异步 ： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步 *IO* 时，*Java* 将 *IO* 读写委托给 *OS* 处理，需要将数据缓冲区地址和大小传给 *OS(*银行卡和密码*)*，*OS* 需要支持异步 *IO* 操作 *API*）；

3 什么是阻塞和非阻塞

阻塞和非阻塞是针对于进程在访问数据的时候，根据 *IO* 操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作方法的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入方法会立即返回一个状态值。

以银行取款为例：

阻塞 ： *ATM* 排队取款，你只能等待（使用阻塞 *IO* 时，*Java* 调用会一直阻塞到读写完成才返回）；

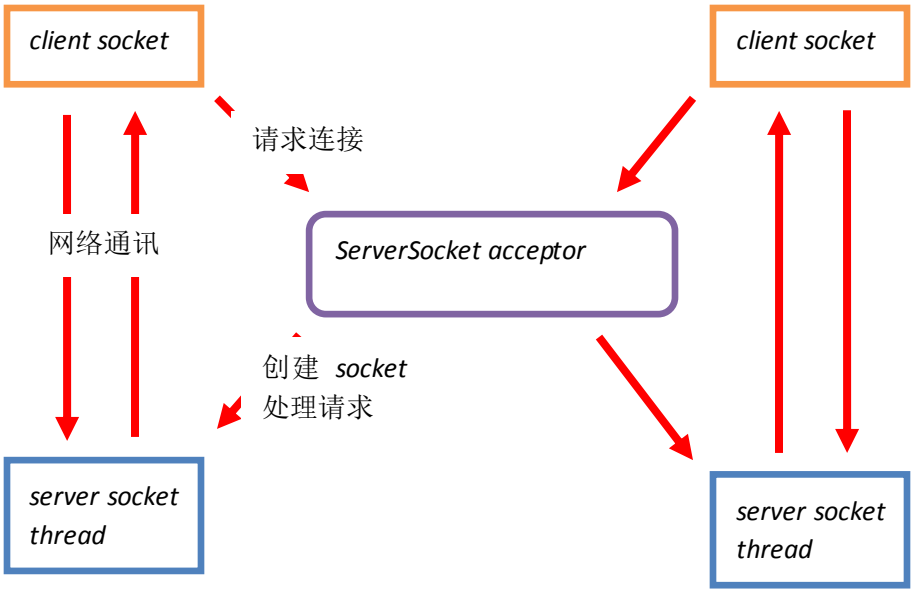
非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞 *IO* 时，如果不能读写 *Java* 调用会马上返回，当 *IO* 事件分发器通知可读写时再继续进行读写，不断循环直到读写完成）

4 BIO 编程

Blocking IO： 同步阻塞的编程方式。

BIO 编程方式通常是在 *JDK1.4* 版本之前常用的编程方式。编程实现过程为：首先在服务端启动一个 *ServerSocket* 来监听网络请求，客户端启动 *Socket* 发起网络请求，默认情况下 *ServerSocket* 回建立一个线程来处理此请求，如果服务端没有线程可用，客户端则会阻塞等待或遭到拒绝。

且建立好的连接，在通讯过程中，是同步的。在并发处理效率上比较低。大致结构如下：

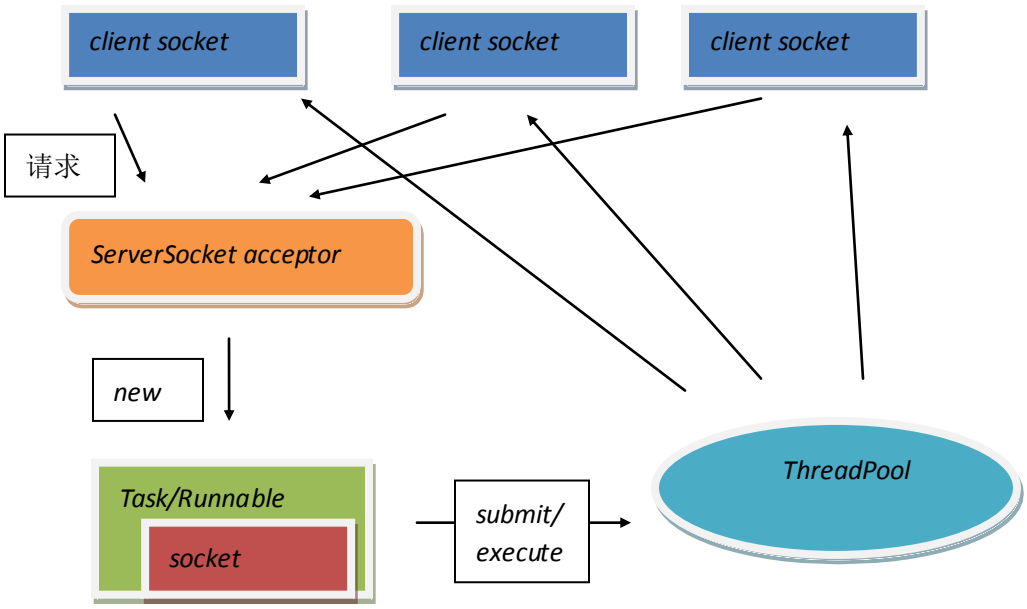


同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并

发局限于应用中，*JDK1.4* 以前的唯一选择，但程序直观简单易理解。

使用线程池机制改善后的 *BIO* 模型图如下*:*



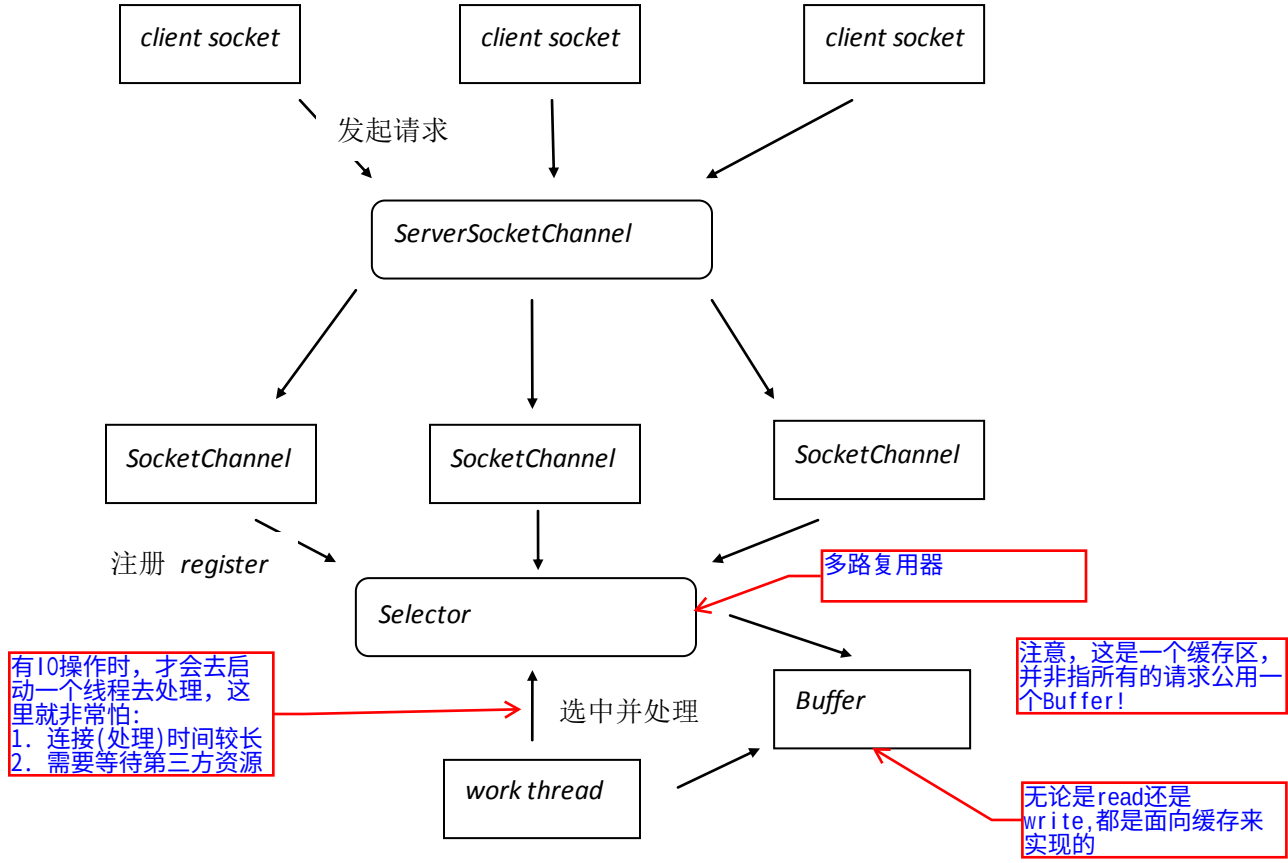
5 NIO 编程

Unblocking IO（*New IO*）： 同步非阻塞的编程方式。

NIO 本身是基于事件驱动思想来完成的，其主要想解决的是 *BIO* 的大并发问题，*NIO* 基于 *Reactor*，当 *socket* 有流可读或可写入 *socket* 时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

NIO 的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

在 *NIO* 的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源*(JDBC* 连接等*)*，其实这个线程就被阻塞了，当并发上来的话，还是会有 *BIO* 一样的问题。



同步非阻塞，服务器实现模式为一个请求一个通道，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。

NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程复杂，JDK1.4 开始支持。

Buffer:ByteBuffer,CharBuffer,ShortBuffer,IntBuffer,LongBuffer,FloatBuffer,DoubleBuffer 。

Channel:SocketChannel,ServerSocketChannel 。

Selector:Selector,AbstractSelector

SelectionKey:OP_READ,OP_WRITE,OP_CONNECT,OP_ACCEPT

6 AIO 编程

Asynchronous IO： 异步非阻塞的编程方式

与 NIO 不同，当进行读写操作时，只须直接调用 API 的 read 或 write 方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入 read 方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将 write 方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，read/write 方法都是异步的，完成后会主动调用回调函数。在 JDK1.7 中，这部分内容被称作 NIO.2，主要在 java.nio.channels 包下增加了下面四个异步通道：

AsynchronousSocketChannel

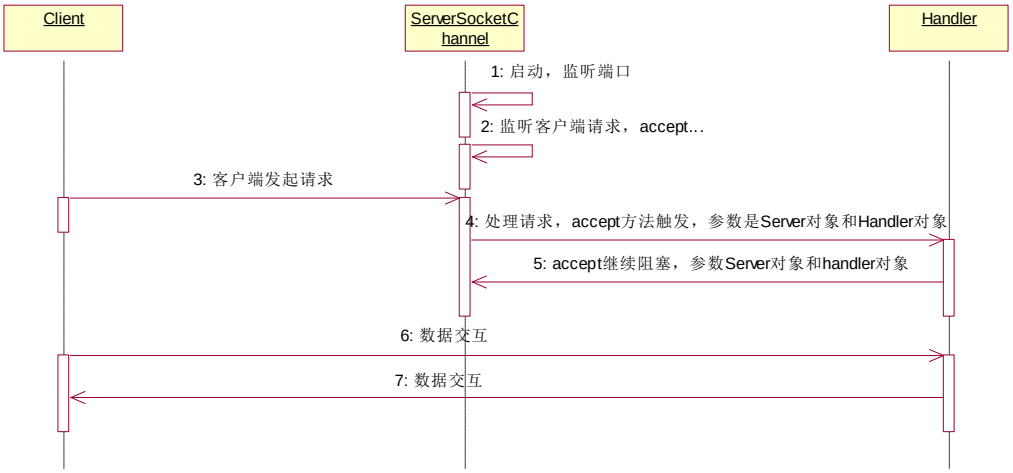
AsynchronousServerSocketChannel

AsynchronousFileChannel

AsynchronousDatagramChannel

异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的 *I/O* 请求都是由 *OS* 先完成了再通知服务器应用去启动线程进行处理。

AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 *OS* 参与并发操作，编程比较复杂，*JDK7* 开始支持。



二、 Netty

1 简介

Netty 是由 *JBOSS* 提供的一个 *java* 开源框架。*Netty* 提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

也就是说，*Netty* 是一个基于 *NIO* 的客户、服务器端编程框架，使用 *Netty* 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。*Netty* 相当简化和流线化了网络应用的编程开发过程，例如，*TCP* 和 *UDP* 的 *socket* 服务开发。

“快速”和“简单”并不用产生维护性或性能上的问题。*Netty* 是一个吸收了多种协议的实现经验，这些协议包括 *FTP,SMTP,HTTP*，各种二进制，文本协议，并经过相当精心设计的项目，最终，*Netty* 成功的找到了一种方式，在保证易于开发的同时还保证了其应用的性能，稳定性和伸缩性。

Netty 从 *4.x* 版本开始，需要使用 *JDK1.6* 及以上版本提供基础支撑。

在设计上：针对多种传输类型的统一接口 - 阻塞和非阻塞；简单但更强大的线程模型；真正的无连接的数据报套接字支持；链接逻辑支持复用；

在性能上：比核心 *Java API* 更好的吞吐量，较低的延时；资源消耗更少，这个得益于共享池和重用；减少内存拷贝

在健壮性上：消除由于慢，快，或重载连接产生的 *OutOfMemoryError*；消除经常发现在 *NIO* 在高速网络中的应用中的不公平的读/写比

在安全上：完整的 *SSL / TLS* 和 *StartTLS* 的支持

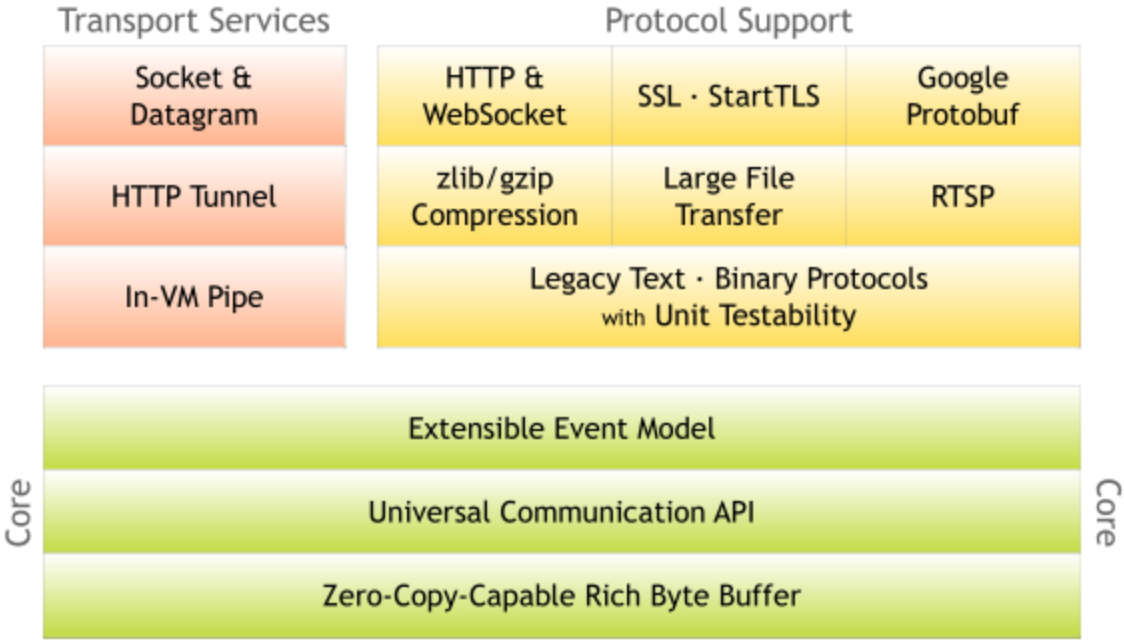
且已得到大量商业应用的真实验证,如：*Hadoop* 项目的 *Avro*（*RPC* 框架）、*Dubbo*、*Dubbox*

等 *RPC* 框架。

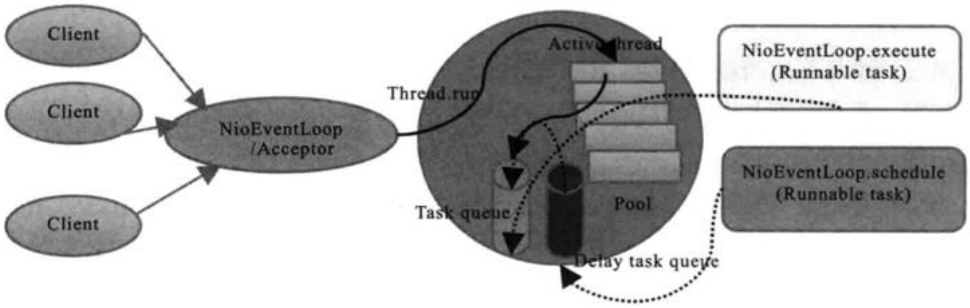
Netty 的官网是：*http://netty.io*

有三方提供的中文翻译 *Netty* 用户手册（官网提供源信息）：*http://ifeve.com/netty5-user-guide/*

2 Netty 架构



3 线程模型



Netty 中支持单线程模型，多线程模型，主从多线程模型。

3.1 单线程模型

在 *ServerBootstrap* 调用方法 *group* 的时候，传递的参数是同一个线程组，且在构造线程组的时候，构造参数为 *1*，这种开发方式，就是一个单线程模型。

个人机开发测试使用。不推荐。

3.2 多线程模型

在 *ServerBootstrap* 调用方法 *group* 的时候，传递的参数是两个不同的线程组。负责监听的 *acceptor* 线程组，线程数为 *1*，也就是构造参数为 *1*。负责处理客户端任务的线程组，线程数大于 *1*，也就是构造参数大于 *1*。这种开发方式，就是多线程模型。

长连接，且客户端数量较少，连接持续时间较长情况下使用。如：企业内部交流应用。

3.3 主从多线程模型

在 *ServerBootstrap* 调用方法 *group* 的时候，传递的参数是两个不同的线程组。负责监听的 *acceptor* 线程组，线程数大于 *1*，也就是构造参数大于 *1*。负责处理客户端任务的线程组，线程数大于 *1*，也就是构造参数大于 *1*。这种开发方式，就是主从多线程模型。

长连接，客户端数量相对较多，连接持续时间比较长的情况下使用。如：对外提供服务的相册服务器。

4 基础程序演示

详见代码

4.1 入门案例

4.2 拆包粘包问题解决

netty 使用 *tcp/ip* 协议传输数据。而 *tcp/ip* 协议是类似水流一样的数据传输方式。多次访问的时候有可能出现数据粘包的问题，解决这种问题的方式如下：

4.2.1 定长数据流

客户端和服务器，提前协调好，每个消息长度固定。（如：长度 *10*）。如果客户端或服务器写出的数据不足 *10*，则使用空白字符补足（如：使用空格）。

4.2.2 特殊结束符

客户端和服务器，协商定义一个特殊的分隔符号，分隔符号长度自定义。如：‘*#*’、‘*$_$*’、‘*AA@*’。在通讯的时候，只要没有发送分隔符号，则代表一条数据没有结束。

4.2.3 协议

相对最成熟的数据传递方式。有服务器的开发者提供一个固定格式的协议标准。客户端和服务器发送数据和接受数据的时候，都依据协议制定和解析消息。

4.3 序列化对象

JBoss Marshalling 序列化

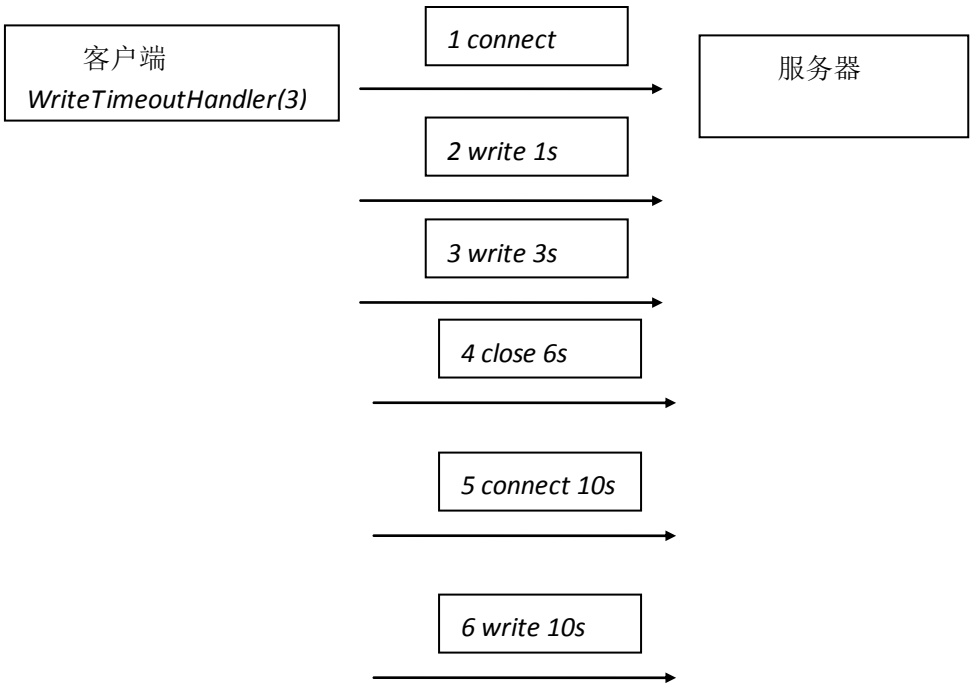
Java 是面向对象的开发语言。传递的数据如果是 *Java* 对象，应该是最方便且可靠。

4.4 定时断线重连

客户端断线重连机制。

客户端数量多，且需要传递的数据量级较大。可以周期性的发送数据的时候，使用。要求对数据的即时性不高的时候，才可使用。

优点： 可以使用数据缓存。不是每条数据进行一次数据交互。可以定时回收资源，对资源利用率高。相对来说，即时性可以通过其他方式保证。如： *120* 秒自动断线。数据变化 *1000* 次请求服务器一次。*300* 秒中自动发送不足 *1000* 次的变化数据。



4.5 心跳监测

使用定时发送消息的方式，实现硬件检测，达到心态检测的目的。

心跳监测是用于检测电脑硬件和软件信息的一种技术。如：*CPU* 使用率，磁盘使用率，内存使用率，进程情况，线程情况等。

4.5.1 sigar

需要下载一个 *zip* 压缩包。内部包含若干 *sigar* 需要的操作系统文件。*sigar* 插件是通过 *JVM* 访问操作系统，读取计算机硬件的一个插件库。读取计算机硬件过程中，必须由操作系

统提供硬件信息。硬件信息是通过操作系统提供的。*zip* 压缩包中是 *sigar* 编写的操作系统文件，如：*windows* 中的动态链接库文件。

解压需要的操作系统文件，将操作系统文件赋值到*${Java_home}/bin* 目录中。

4.6 HTTP 协议处理

使用 *Netty* 服务开发。实现 *HTTP* 协议处理逻辑。

5 流数据的传输处理

在基于流的传输里比如 *TCP/IP*，接收到的数据会先被存储到一个 *socket* 接收缓冲里。不幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使你发送了 *2* 个独立的数据包，操作系统也不会作为 *2* 个消息处理而仅仅是作为一连串的字节而言。因此这是不能保证你远程写入的数据就会准确地读取。所以一个接收方不管他是客户端还是服务端，都应该把接收到的数据整理成一个或者多个更有意思并且能够让程序的业务逻辑更好理解的数据。

在处理流数据粘包拆包时，可以使用下述处理方式：

使用定长数据处理，如：每个完整请求数据长度为 *8* 字节等。（*FixedLengthFrameDecoder*）

使用特殊分隔符的方式处理，如：每个完整请求数据末尾使用*'\0'*作为数据结束标记。（*DelimiterBasedFrameDecoder*）

使用自定义协议方式处理，如：*http* 协议格式等。

使用 *POJO* 来替代传递的流数据，如：每个完整的请求数据都是一个 *RequestMessage* 对象，在 *Java* 语言中，使用 *POJO* 更符合语种特性，推荐使用。