1、网络协议

计算机网络体系结构

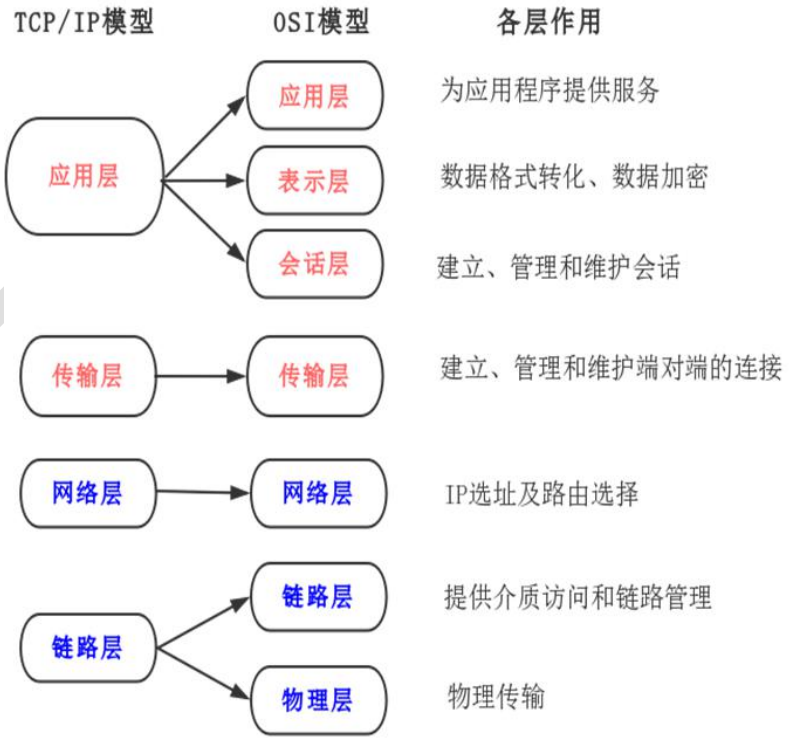
OSI 七层模型

开放系统互连参考模型（Open System Interconnect 简称 OSI）是国际标准化组织(ISO)和国际电报电话咨询委员会(CCITT)联合制定的开放系统互连参考模型，为开放式互连信息系统提供了一种功能结构的框架。其目的是为异种计算机互连提供一个共同的基础和标准框架，并为保持相关标准的一致性和兼容性提供共同的参考。这里所说的开放系统，实质上指的是遵循 OSI 参考模型和相关协议能够实现互连的具有各种应用目的的计算机系统。

OSI 采用了分层的结构化技术，共分七层，**物理层、数据链路层、网络层、传输层、会话层、表示层、应用层**。

TCP/IP 模型

OSI 模型比较复杂且学术化，所以我们实际使用的 TCP/IP 模型，共分 4 层，**链路层、网络层、传输层、应用层**。两个模型之间的对应关系如图所示：



无论什么模型，每一个抽象层建立在低一层提供的服务上，并且为高一层提供服务。

TCP/IP 协议族

Transmission Control Protocol/Internet Protocol 的简写，中译名为传输控制协议/因特网互联协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。协议采用了 4 层的层级结构。然而在很多情况下，它是利用 IP 进行通信时所必须用到的协议群的统称。也就是说，它其实是个协议家族，由很多个协议组成，并且是在不同的层， 是互联网的基础通信架构。

TCP/IP概念层模型	功能	TCP/IP协议族
应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
	数据格式化，代码转换，数据加密	没有协议
	解除或建立与别的接点的联系	没有协议
传输层	提供端对端的接口	TCP，UDP
网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
	以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802，IEEE802.2

TCP 和 UDP

在上述表格中，网际协议 IP 是 TCP/IP 中非常重要的协议。负责对数据加上 IP 地址（有发送它的主机的地址（源地址）和接收它的主机的地址（目的地址））和其他的数据以确定传输的目标。

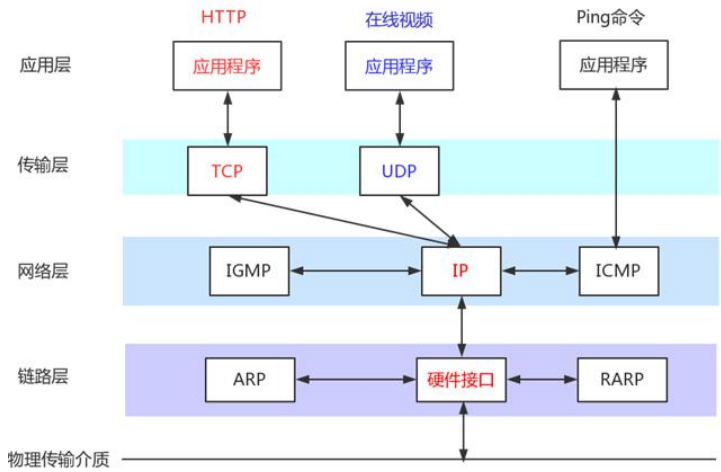
而 TCP 和 UDP 都是传输层的协议，传输层主要为两台主机上的应用程序提供端到端的通信。

但是 TCP 和 UDP 最不同的地方是，TCP 提供了一种可靠的数据传输服务，TCP 是面向连接的，也就是说，利用 TCP 通信的两台主机首先要经历一个建立连接的过程，等到连接建立后才开始传输数据，而且传输过程中采用“带重传的肯定确认”技术来实现传输的可靠性。TCP 还采用一种称为“滑动窗口”的方式进行流量控制，发送完成后还会关闭连接。所以 TCP 要比 UDP 可靠的多。

UDP（User Datagram Protocol 的简称， 中文名是用户数据报协议）是把数据直接发出去，而不管对方是不是在接收，也不管对方是否能接收的了，也不需要接收方确认，属于不可靠的传输，可能会出现丢包现象，实际应用中要求程序员编程验证。

注意：

我们一些常见的网络应用基本上都是基于 TCP 和 UDP 的，这两个协议又会使用网络层的 IP 协议。但是我们完全可以绕过传输层的 TCP 和 UDP，直接使用 IP，比如 Linux 中 LVS，甚至直接访问链路层，比如 tcpdump 程序就是直接和链路层进行通信的。



上图中，其他一些协议的名称解释，了解即可：

ICMP 控制报文协议

IGMP internet 组管理协议

ARP 地址解析协议

RARP 反向地址转化协议

地址和端口号

我们常听说 MAC 地址和 IP 地址。MAC 地址就是在媒体接入层上使用的地址，也叫物理地址、硬件地址或链路地址，由网络设备制造商生产时写在硬件内部。MAC 地址与网络无关，也即无论将带有这个地址的硬件（如网卡、集线器、路由器等）接入到网络的何处，都有相同的 MAC 地址，它由厂商写在网卡的 BIOS 里，从理论上讲，除非盗来硬件（网卡），否则是没有办法冒名顶替的。

IP 地址后者用来识别 TCP/IP 网络中互连的主机和路由器。IP 地址基于逻辑，比较灵活，不受硬件限制，也容易记忆。

在传输层也有这种类似于地址的概念，那就是端口号。端口号用来识别同一台计算机中进行通信的不同应用程序。因此，它也被称为程序地址。

一台计算机上同时可以运行多个程序。传输层协议正是利用这些端口号识别本机中正在进行通信的应用程序，并准确地将数据传输。

端口号的确定

- 标准既定的端口号：这种方法也叫静态方法。它是指每个应用程序都有其指定的端口号。但并不是说可以随意使用任何一个端口号。例如 HTTP、FTP、TELNET 等广为使用的应用协议中所使用的端口号就是固定的。这些端口号被称为知名端口号，分布在 0~1023 之间；除知名端口号之外，还有一些端口号被正式注册，它们分布在 1024~49151 之间，不过这些端口号可用于任何通信用途。

- 时序分配法：服务器有必要确定监听端口号，但是接受服务的客户端没必要确定端口号。在这种方法下，客户端应用程序完全可以不用自己设置端口号，而全权交给操作系统进行分配。动态分配的端口号范围在 49152~65535 之间。

端口号与协议

- 端口号由其使用的传输层协议决定。因此，不同的传输层协议可以使用相同的端口号。
- 此外，那些知名端口号与传输层协议并无关系。只要端口一致都将分配同一种应用程序进行处理。

TCP/IP

TCP 是面向连接的通信协议，通过三次握手建立连接，通讯完成时要拆除连接，由于 TCP 是面向连接的所以只能用于端到端的通讯。

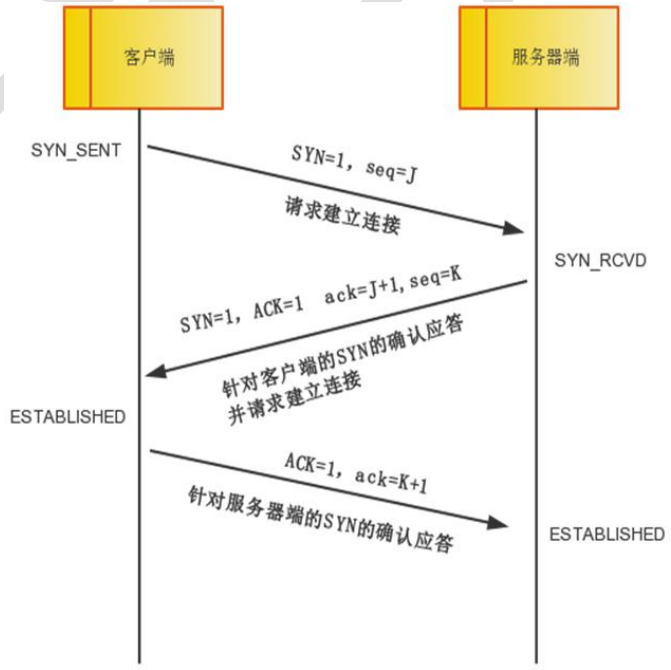
TCP 提供的是一种可靠的数据流服务，采用“带重传的肯定确认”技术来实现传输的可靠性。TCP 还采用一种称为“滑动窗口”的方式进行流量控制，所谓窗口实际表示接收能力，用以限制发送方的发送速度。

如果 IP 数据包中有已经封好的 TCP 数据包，那么 IP 将把它们向‘上’传送到 TCP 层。TCP 将包排序并进行错误检查，同时实现虚电路间的连接。TCP 数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传。

TCP 将它的信息送到更高层的应用程序，例如 Telnet 的服务程序和客户程序。应用程序轮流将信息送回 TCP 层，TCP 层便将它们向下传送到 IP 层，设备驱动程序和物理介质，最后到接收方。

面向连接的服务（例如 Telnet、FTP、rlogin、X Windows 和 SMTP）需要高度的可靠性，所以它们使用了 TCP。DNS 在某些情况下使用 TCP（发送和接收域名数据库），但使用 UDP 传送有关单个主机的信息。

TCP 三次握手



TCP 提供面向有连接的通信传输。面向有连接是指在数据通信开始之前先做好两端之间的准备工作。

所谓三次握手是指建立一个 TCP 连接时需要客户端和服务器端总共发送三个包以确认连接的建立。在 socket 编程中，这一过程由客户端执行 connect 来触发。

第一次握手：客户端将标志位 SYN 置为 1，随机产生一个值 seq=J，并将该数据包发送给服务器端，客户端进入 SYN_SENT 状态，等待服务器端确认。

第二次握手：服务器端收到数据包后由标志位 SYN=1 知道客户端请求建立连接，服务器端将标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个值 seq=K，并将该数据包发送给客户端以确认连接请求，服务器端进入 SYN_RCVD 状态。

第三次握手：客户端收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1，并将该数据包发送给服务器端，服务器端检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，客户端和服务器端进入 ESTABLISHED 状态，完成三次握手，随后客户端与服务器端之间可以开始传输数据了。

TCP 的三次握手的漏洞

但是在 TCP 三次握手中是有一个缺陷的，就是如果我们利用三次握手的缺陷进行攻击。这个攻击就是 SYN 洪泛攻击。三次握手中有一个第二次握手，服务端向客户端应道请求，应答请求是需要客户端 IP 的，服务端是需要知道客户端 IP 的，攻击者就伪造这个 IP，往服务器端狂发送第一次握手的内容，当然第一次握手中的客户端 IP 地址是伪造的，从而服务端忙于进行第二次握手但是第二次握手当然没有结果，所以导致服务器端被拖累，死机。

当然我们的生活中也有可能有这种例子，一个家境一般的 IT 男去表白他的女神被拒绝了，理由是他家里没矿，IT 男为了报复，采用了洪泛攻击，他请了很多人伪装成有钱人去表白那位追求矿的女神，让女生每次想交往时发现表白的人不见了同时还联系不上了。

面对这种攻击，有以下的解决方案，最好的方案是防火墙。

无效连接监视释放

这种方法不停监视所有的连接，包括三次握手的，还有握手一次的，反正是所有的，当达到一定(与)阈值时拆除这些连接，从而释放系统资源。这种方法对于所有的连接一视同仁，不管是正常的还是攻击的，所以这种方式不推荐。

延缓 TCB 分配方法

一般的做完第一次握手之后，服务器就需要为该请求分配一个 TCB（连接控制资源），通常这个资源需要 200 多个字节。延迟 TCB 的分配，当正常连接建立起来后再分配 TCB 则可以有效地减轻服务器资源的消耗。

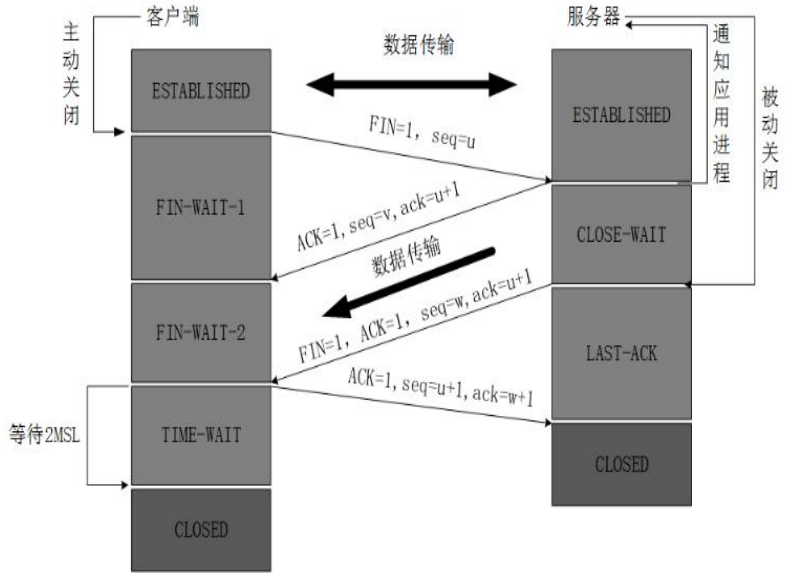
使用防火墙

防火墙在确认了连接的有效性后，才向内部的服务器（Listener）发起 SYN 请求，

TCP 四次挥手（分手）

四次挥手即终止 TCP 连接，就是指断开一个 TCP 连接时，需要客户端和服务端总共发送 4 个包以确认连接的断开。在 socket 编程中，这一过程由客户端或服务端任一方执行 close 来触发。

由于 TCP 连接是全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个 FIN 来终止这一方向的连接，收到一个 FIN 只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个 TCP 连接上仍然能够发送数据，直到这一方向也发送了 FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。



1. 客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。
2. 服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。
3. 客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。
4. 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此

时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

5. 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2**MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。

服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。

TCP/IP 中的数据包

每个分层中，都会对所发送的数据附加一个首部，在这个首部中包含了该层必要的信息，如发送的目标地址以及协议相关信息。通常，为协议提供的信息为包首部，所要发送的内容为数据。在下一层的角度看，从上一层收到的包全部都被认为是本层的数据。

网络中传输的数据包由两部分组成：一部分是协议所要用到的首部，另一部分是上一层传过来的数据。首部的结构由协议的具体规范详细定义。在数据包的首部，明确标明了协议应该如何读取数据。反过来说，看到首部，也就能够了解该协议必要的信息以及所要处理的数据。

① 应用程序处理

首先应用程序会进行编码处理，这些编码相当于 OSI 的表示层功能；
编码转化后，邮件不一定马上被发送出去，这种何时建立通信连接何时发送数据的管理功能，相当于 OSI 的会话层功能。

② TCP 模块的处理

TCP 根据应用的指示，负责建立连接、发送数据以及断开连接。TCP 提供将应用层发来的数据顺利发送至对端的可靠传输。为了实现这一功能，需要在应用层数据的前端附加一个 TCP 首部。

③ IP 模块的处理

IP 将 TCP 传过来的 TCP 首部和 TCP 数据合起来当做自己的数据，并在 TCP 首部的前端加上自己的 IP 首部。IP 包生成后，参考路由控制表决定接受此 IP 包的路由或主机。

④ 网络接口（以太网驱动）的处理

从 IP 传过来的 IP 包对于以太网来说就是数据。给这些数据附加上以太网首部并进行发送处理，生成的以太网数据包将通过物理层传输给接收端。

⑤ 网络接口（以太网驱动）的处理

主机收到以太网包后，首先从以太网包首部找到 MAC 地址判断是否为发送给自己的包，若不是则丢弃数据。
如果是发送给自己的包，则从以太网包首部中的类型确定数据类型，再传给相应的模块，如 IP、ARP 等。这里的例子则是 IP 。

⑥ IP 模块的处理

IP 模块接收到 数据后也做类似的处理。从包首部中判断此 IP 地址是否与自己

的 IP 地址匹配，如果匹配则根据首部的协议类型将数据发送给对应的模块，如 TCP、UDP。这里的例子则是 TCP。
另外吗，对于有路由器的情况，接收端地址往往不是自己的地址，此时，需要借助路由控制表，在调查应该送往的主机或路由器之后再进行转发数据。

⑦ TCP 模块的处理

在 TCP 模块中，首先会计算一下校验和，判断数据是否被破坏。然后检查是否在按照序号接收数据。最后检查端口号，确定具体的应用程序。数据被完整地接收以后，会传给由端口号识别的应用程序。

⑧ 应用程序的处理

接收端应用程序会直接接收发送端发送的数据。通过解析数据，展示相应的内容。

TCP 的通讯原理

Socket 套接字

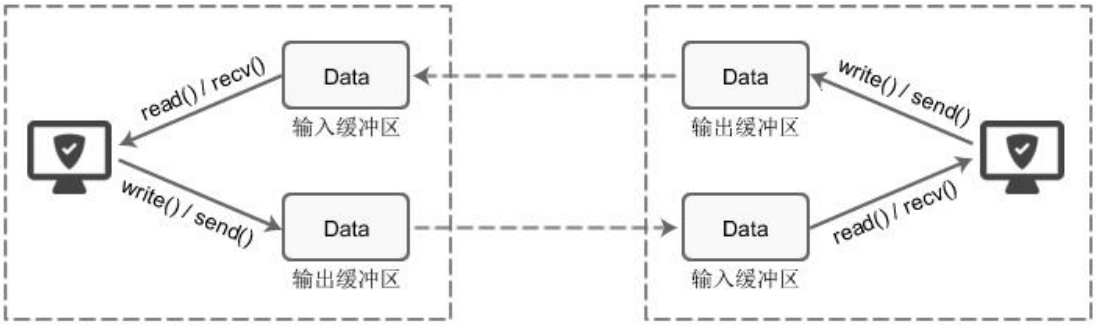
Socket 的原意是“插座”，在计算机通信领域，socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。TCP 用主机的 IP 地址加上主机上的端口号作为 TCP 连接的端点，这种端点就叫做套接字（socket）。

区分不同应用程序进程间的网络通信和连接，主要有 3 个参数：通信的目的 IP 地址、使用的传输层协议(TCP 或 UDP)和使用的端口号。通过将这 3 个参数结合起来，与一个“插座”Socket 绑定，应用层就可以和传输层通过套接字接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

套接字对是一个定义该连接的两个端点的四元组：本地 IP 地址、本地 TCP 端口号、外地 IP 地址、外地 T C P端口号。套接字对唯一标识一个网络上的每个 TCP 连接。

TCP 缓冲区

每个 TCP 的 Socket 的内核中都有一个发送缓冲区和一个接收缓冲区。现在我们假设用 write()方法发送数据，使用 read()方法接收数据。



write()并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由 TCP 协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是 TCP 协议负责的事情。

TCP 协议独立于 write()函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

read()也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。

总得来说，I/O 缓冲区在每个 TCP 套接字中单独存在；I/O 缓冲区在创建套接字时自动生成；

TCP 的可靠性

在 TCP 中，当发送端的数据到达接收主机时，接收端主机会返回一个已收到消息的通知。这个消息叫做确认应答（ACK）。当发送端将数据发出之后会等待对端的确认应答。如果有确认应答，说明数据已经成功到达对端。反之，则数据丢失的可能性很大。

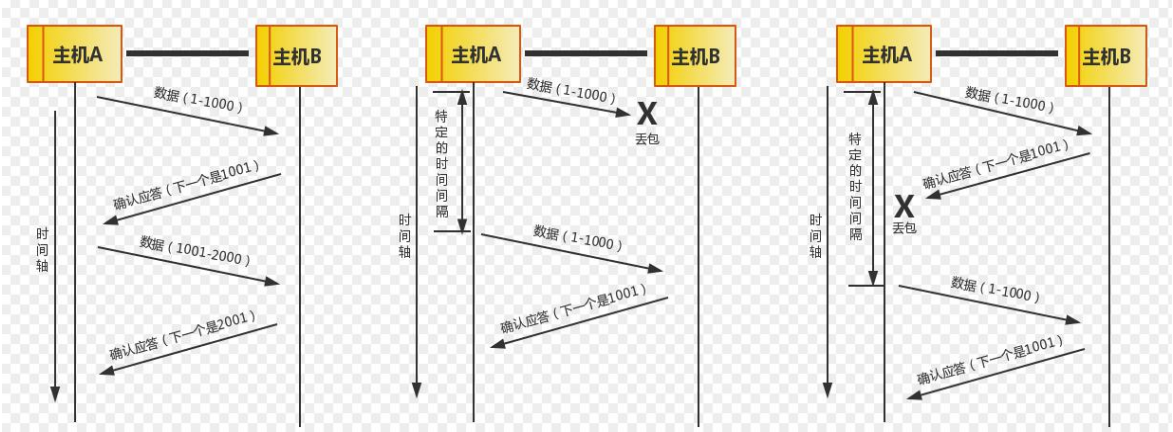
在一定时间内没有等待到确认应答，发送端就可以认为数据已经丢失，并进行重发。由此，即使产生了丢包，仍然能够保证数据能够到达对端，实现可靠传输。

未收到确认应答并不意味着数据一定丢失。也有可能是数据对方已经收到，只是返回的确认应答在途中丢失。这种情况也会导致发送端误以为数据没有到达目的地而重发数据。

此外，也有可能因为一些其他原因导致确认应答延迟到达，在源主机重发数据以后才到达的情况也屡见不鲜。此时，源主机只要按照机制重发数据即可。

对于目标主机来说，反复收到相同的数据是不可取的。为了对上层应用提供可靠的传输，目标主机必须放弃重复的数据包。为此我们引入了序列号。

序列号是按照顺序给发送数据的每一个字节（8 位字节）都标上号码的编号。接收端查询接收数据 TCP 首部中的序列号和数据的长度，将自己下一步应该接收的序列号作为确认应答返送回去。通过序列号和确认应答号，TCP 能够识别是否已经接收数据，又能够判断是否需要接收，从而实现可靠传输。



TCP 中的滑动窗口

发送方和接收方都会维护一个数据帧的序列，这个序列被称作窗口。发送方的窗口大小由接收方确认，目的是控制发送速度，以免接收方的缓存不够大导致溢出，同时控制流量也可以避免网络拥塞。

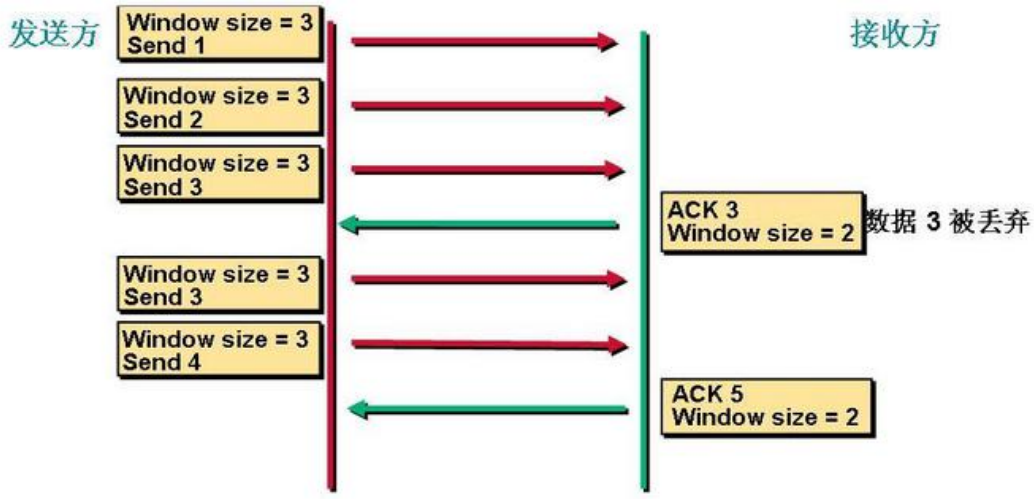
在 TCP 的可靠性的图中，我们可以看到，发送方每发送一个数据接收方就要给发送方一个 ACK 对这个数据进行确认。只有接收了这个确认数据以后发送方才能传输下个数据。

存在的问题：如果窗口过小，当传输比较大的数据的时候需要不停的对数据进行确认，这个时候就会造成很大的延迟。

如果窗口过大，我们假设发送方一次发送 100 个数据，但接收方只能处理 50 个数据，这样每次都只对这 50 个数据进行确认。发送方下一次还是发送 100 个数据，但接受方还是只能处理 50 个数据。这样就避免了不必要的数据来拥塞我们的链路。

因此，我们引入了滑动窗口。滑动窗口通俗来讲就是一种流量控制技术。

它本质上是描述接收方的 TCP 数据报缓冲区大小的数据，发送方根据这个数据来计算自己最多能发送多长的数据，如果发送方收到接收方的窗口大小为 0 的 TCP 数据报，那么发送方将停止发送数据，等到接收方发送窗口大小不为 0 的数据报的到来。



首先是第一次发送数据这个时候的窗口大小是根据链路带宽的大小来决定的。我们假设这个时候窗口的大小是 3。这个时候接受方收到数据以后会对数据进行确认告诉发送方我下次希望手到的是数据是多少。这里我们看到接收方发送的 ACK=3(这是发送方发送序列 2 的回答确认，下一次接收方期望接收到的是 3 序列信号)。这个时候发送方收到这个数据以后就知道我第一次发送的 3 个数据对方只收到了 2 个。就知道第 3 个数据对方没有收到。下次在发送的时候就从第 3 个数据开始发。

此时窗口大小变成了 2 。

于是发送方发送 2 个数据。看到接收方发送的 ACK 是 5 就表示他下一次希望收到的数据是 5，发送方就知道我刚才发送的 2 个数据对方收了这个时候开始发送第 5 个数据。

这就是滑动窗口的工作机制，当链路变好了或者变差了这个窗口还会发生变话，并不是第一次协商好了以后就永远不变了。

所以滑动窗口协议，是 TCP 使用的一种流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

只有在接收窗口向前滑动时（与此同时也发送了确认），发送窗口才有可能向前滑动。

收发两端的窗口按照以上规律不断地向前滑动，因此这种协议又称为滑动窗口协议。

HTTP

HTTP 协议是 Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。

HTTP 协议

我们使用 http 来访问 Web 上某个资源，比如 html/文本、word、avi 电影、其他资源。

Content-Type 指示响应的内容，这里是 text/html 表示 HTML 网页。请注意，浏览器就是依靠 Content-Type 来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠 URL 来判断响应的内容，所以，即使 URL 是 http://example.com/abc.jpg，它也不一定就是图片。

HTTP 使用统一资源标识符（Uniform Resource Identifiers, URI）来传输数据和建立连接。URL 是一种特殊类型的 URI，包含了用于查找某个资源的足够的信息。

URL，全称是 UniformResourceLocator，中文叫统一资源定位符,是互联网上用来标识某一处资源的地址。

URI 和 URL 的区别：

URI 是个纯粹的句法结构，用于指定标识 Web 资源的字符串的各个不同部分。URL 是 URI 的一个特例，它包含了定位 Web 资源的足够信息。其他 URI，比如

mailto：cay@horstman.com

则不属于定位符，因为根据该标识符无法定位任何资源。

URI 是统一资源**标识**符，而 URL 是统一资源定位符。因此，笼统地说，每个 URL 都是 URI，但不一定每个 URI 都是 URL。这是因为 URI 还包括一个子类，即统一资源名称 (URN)，它命名资源但不指定如何定位资源。上面的 mailto 就是一个 URN 的示例。

URL 是 uniform resource locator，统一资源**定位**器，它是一种具体的 URI，即 URL 可以用来标识一个资源，而且还指明了如何 locate 这个资源。

一个完整的 URL

包括以下几部分：

http://www.enjoyedu.com:8080/news/index.asp?boardID=5&ID=24618&page=1#name

1.协议部分：该 URL 的协议部分为“http：”，这代表网页使用的是 HTTP 协议。在 Internet 中可以使用多种协议，如 HTTP，FTP 等等本例中使用的是 HTTP 协议。在"HTTP"后面的“//”为分隔符

2.域名部分：该 URL 的域名部分为“www.enjoyedu.com”。一个 URL 中，也可以使用 IP 地址作为域名使用

3.端口部分：跟在域名后面的是端口，域名和端口之间使用“:”作为分隔符。端口不是一个 URL 必须的部分，如果省略端口部分，将采用默认端口

4.虚拟目录部分：从域名后的第一个“/”开始到最后一个“/”为止，是虚拟目录部分。虚拟目录也不是一个 URL 必须的部分。本例中的虚拟目录是“/news/”

5.文件名部分：从域名后的最后一个“/”开始到“？”为止，是文件名部分，如果没有“?”,则是从域名后的最后一个“/”开始到“#”为止，是文件部分，如果没有“？”和“#”，那么从域名后的最后一个“/”开始到结束，都是文件名部分。本例中的文件名是“index.asp”。文件名部分也不是一个 URL 必须的部分，如果省略该部分，则使用默认的文件名

6.锚部分：从“#”开始到最后，都是锚部分。本例中的锚部分是“name”。锚部分也不是一个 URL 必须的部分

7.参数部分：从“？”开始到“#”为止之间的部分为参数部分，又称搜索部分、查询部分。本例中的参数部分为“boardID=5&ID=24618&page=1”。参数可以允许有多个参数，参数与参数之间用“&”作为分隔符。

HTTP 请求的传输过程

首先作为发送端的客户端在应用层（HTTP 协议）发出一个想看某个 Web 页面的 HTTP 请求。

接着，为了传输方便，在传输层（TCP 协议）把从应用层处收到的数据（HTTP 请求报文）进行分割，并在各个报文上打上标记序号及端口号后转发给网络层。

在网络层（IP 协议），增加作为通信目的地的 MAC 地址后转发给链路层。这样一来，发往网络的通信请求就准备齐全了。

接收端的服务器在链路层接收到数据，按序往上层发送，一直到应用层。当传输到应用层，才能算真正接收到由客户端发送过来的 HTTP 请求。

一次完整 http 请求的过程

首先进行 DNS 域名解析（本地浏览器缓存、操作系统缓存或者 DNS 服务器）

建立 TCP 连接

在 HTTP 工作开始之前，客户端首先要通过网络与服务器建立连接，该连接是通过 TCP 来完成的，该协议与 IP 协议共同构建 Internet，即著名的 TCP/IP 协议族，因此 Internet 又被称作是 TCP/IP 网络。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后，才能进行高层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是 80；

- 客户端向服务器发送请求命令

一旦建立了 TCP 连接，客户端就会向服务器发送请求命令；

例如：GET/sample/hello.jsp HTTP/1.1

- 客户端发送请求头信息

客户端发送其请求命令之后，还要以头信息的形式向服务器发送一些别的信息，之后客户端发送了一空白行来通知服务器，它已经结束了该头信息的发送；

- 服务器应答

客户端向服务器发出请求后，服务器会客户端返回响应；

例如： HTTP/1.1 200 OK

响应的第一部分是协议的版本号和响应状态码

- 服务器返回响应头信息

正如客户端会随同请求发送关于自身的信息一样，服务器也会随同响应向用户发送关于它自己的数据及被请求的文档；

- 服务器向客户端发送数据

服务器向客户端发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 响应头信息所描述的格式发送用户所请求的实际数据；

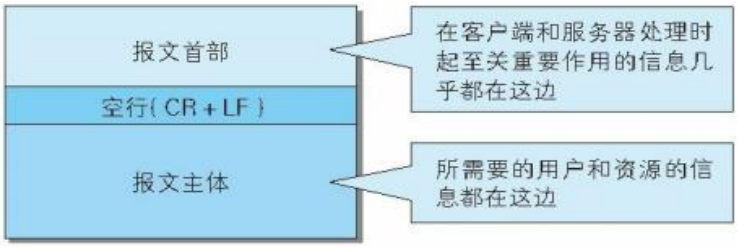
- 服务器关闭 TCP 连接

一般情况下，一旦服务器向客户端返回了请求数据，它就要关闭 TCP 连接，然后如果客户端或者服务器在其头信息加入了这行代码 Connection:keep-alive ，TCP 连接在发送后将仍然保持打开状态，于是，客户端可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

HTTP 协议报文结构

用于 HTTP 协议交互的信息被称为 HTTP 报文。请求端（客户端）的 HTTP 报文叫做请求报文；响应端（服务器端）的叫做响应报文。HTTP 报文本身是由多行（用 CR+LF 作换行符）数据构成的字符串文本。

HTTP 报文大致可分为报文首部和报文主体两部分。两者由最初出现的空行（CR+LF）来划分。通常，并不一定有报文主体。

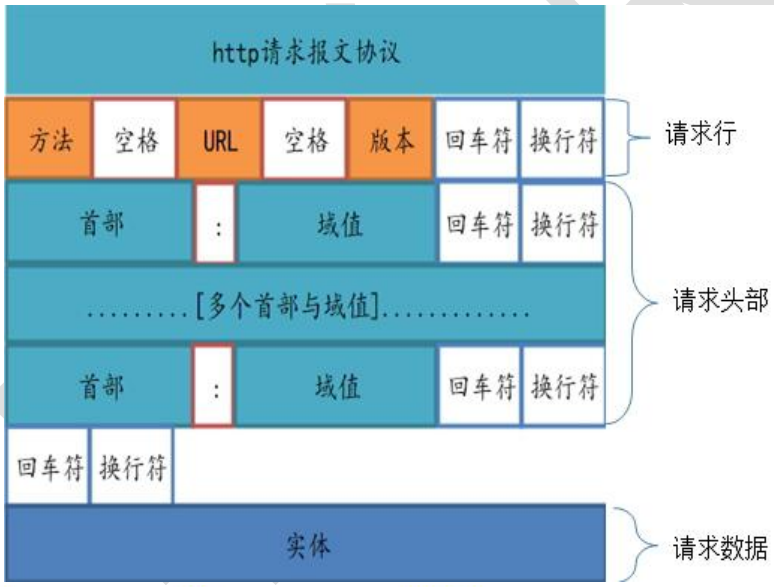


请求报文结构

请求报文的首部内容由以下数据组成：

请求行 —— 包含用于请求的方法、请求 URI 和 HTTP 版本。

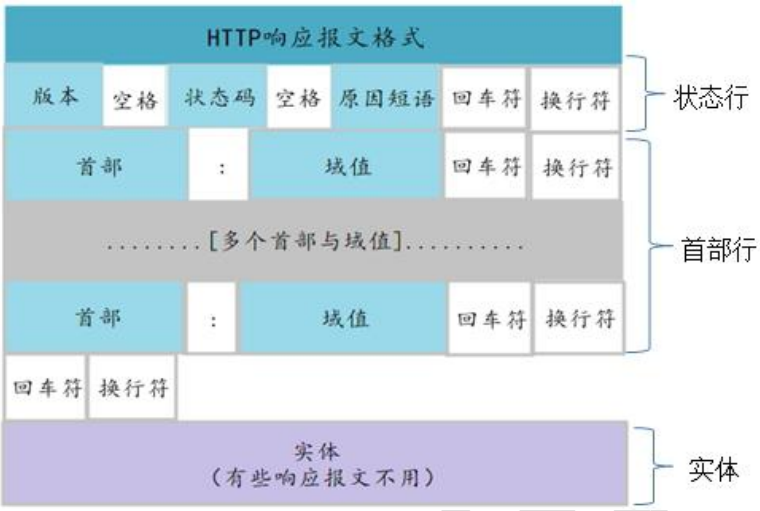
首部字段 —— 包含表示请求的各种条件和属性的各类首部。（通用首部、请求首部、实体首部以及 RFC 里未定义的首部如 Cookie 等）

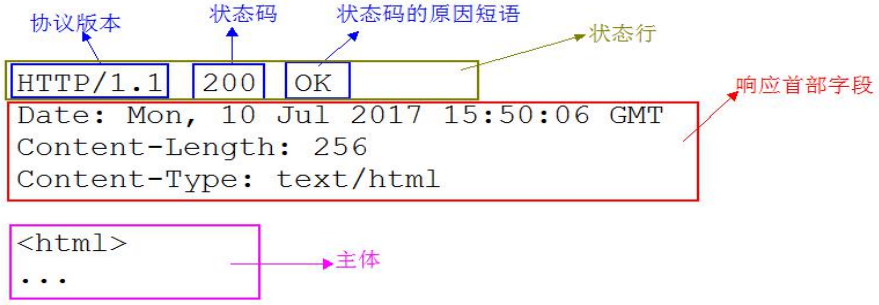


响应报文结构

状态行 —— 包含表明响应结果的状态码、原因短语和 HTTP 版本。

首部字段 —— 包含表示请求的各种条件和属性的各类首部。（通用首部、响应首部、实体首部以及 RFC 里未定义的首部如 Cookie 等）





	类别	原因
1xx	Informational(信息性状态码)	接收的请求正在处理
2xx	Success(成功状态码)	请求正常处理完毕
3xx	Redirection(重定向状态码)	需要进行附加操作以完成请求
4xx	Client Error(客户端错误状态码)	服务器无法处理请求
5xx	Server Error(服务器错误状态码)	服务器处理请求出错

UDP 协议

UDP 是面向无连接的通讯协议，UDP 数据包括目的端口号和源端口号信息，由于通讯不需要连接，所以可以实现广播发送。

UDP 通讯时不需要接收方确认，属于不可靠的传输，可能会出现丢包现象，实际应用中要求程序员编程验证。

UDP 与 TCP 位于同一层，但它不管数据包的顺序、错误或重发。因此，UDP 不被应用于那些面向连接的服务，UDP 主要用于那些面向查询---应答的服务，例如 NFS。相对于 FTP 或 Telnet，这些服务需要交换的信息量较小。使用 UDP 的服

务包括 NTP（网络时间协议）和 DNS（DNS 也使用 TCP），包总量较少的通信（DNS、SNMP 等）；2.视频、音频等多媒体通信（即时通信）；3.限定于 LAN 等特定网络中的应用通信；4.广播通信（广播、多播）。

常用的 QQ，就是一个以 UDP 为主，TCP 为辅的通讯协议。

TCP 和 UDP 的优缺点无法简单地、绝对地去做比较：TCP 用于在传输层有必要实现可靠传输的情况；而在一方面，UDP 主要用于那些对高速传输和实时性有较高要求的通信或广播通信。TCP 和 UDP 应该根据应用的目的按需使用。

2、Java 原生网络编程

一些常见术语

编程中的 Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。

主机 A 的应用程序要能和主机 B 的应用程序通信，必须通过 Socket 建立连接，而建立 Socket 连接必须需要底层 TCP/IP 协议来建立 TCP 连接。建立 TCP 连接需要底层 IP 协议来寻址网络中的主机。我们知道网络层使用的 IP 协议可以帮助我们根据 IP 地址来找到目标主机，但是一台主机上可能运行着多个应用程序，如何才能与指定的应用程序通信就要通过 TCP 或 UPD 的地址也就是端口号来指定。这样就可以通过一个 Socket 实例唯一代表一个主机上的一个应用程序的通信链路了。

短连接：

连接->传输数据->关闭连接

HTTP 是无状态的，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，但任务结束就中断连接。

也可以这样说：短连接是指 SOCKET 连接后发送后接收完数据后马上断开连接。

长连接：

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立 SOCKET 连接后不管是否使用都保持连接，但安全性较差。

什么时候用长连接，短连接？

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个 TCP 连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，处理时直接发送数据包就 OK 了，不用建立 TCP 连接。

例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成 socket 错误，而且频繁的 socket 创建也是对资源的浪费。

而像 WEB 网站的 http 服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像 WEB 网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

总之，长连接和短连接的选择要视情况而定。

Linux 网络 IO 模型

同步和异步，阻塞和非阻塞

同步和异步

关注的是结果消息的通信机制

同步:同步的意思就是调用方需要主动等待结果的返回

异步:异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。

阻塞和非阻塞

主要关注的是等待结果返回调用方的状态

阻塞:是指结果返回之前，当前线程被挂起，不做任何事

非阻塞:是指结果在返回之前，线程可以做一些其他事，不会被挂起。

两者的组合

1.同步阻塞:同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就在店里面一直等，期间不做任何事(包括看手机)，等着商家进货，直到有货为止，这个效率很低。

2.同步非阻塞:同步非阻塞在编程中可以抽象为一个轮询模式，你去了商店之后，发现衣服卖完了，这个时候不需要傻傻的等着，你可以去其他地方比如奶茶店，买杯水，但是你还是需要时不时的去商店问老板新衣服到了吗。

3.异步阻塞:异步阻塞这个编程里面用的较少，有点类似你写了个线程池,submit 然后马上 future.get()，这样线程其实还是挂起的。有点像你去商店买衣服，这个时候发现衣服没有了，这个时候你就给老板留给电话，说衣服到了就给我打电话，然后你就守着这个电话，一直等着他响什么事也不做。这样感觉的确有点傻，所以这个模式用得比较少。

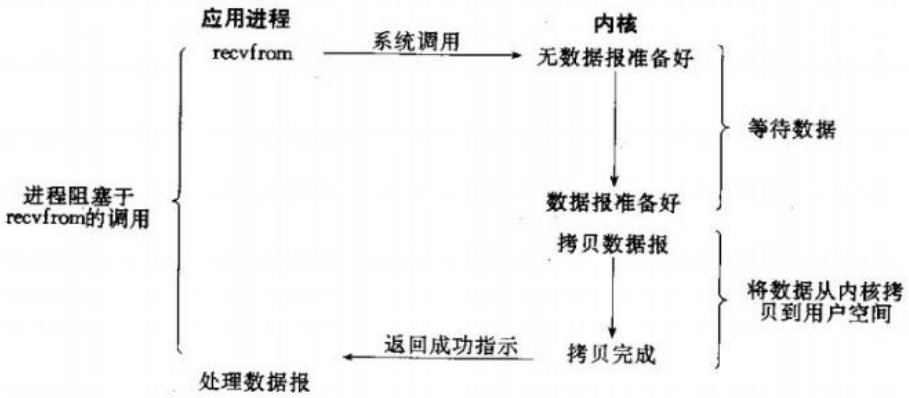
4.异步非阻塞:异步非阻塞。好比你去商店买衣服，衣服没了，你只需要给老板说这是我的电话，衣服到了就打。然后你就随心所欲的去玩，也不用操心衣服什么时候到，衣服一到，电话一响就可以去买衣服了。

五种 I/O 模型

1)阻塞I/O（blocking I/O）
2)非阻塞I/O （nonblocking I/O）
3) I/O复用(select 、poll和epoll)（I/O multiplexing）
4)信号驱动I/O （signal driven I/O (SIGIO)）
5)异步I/O （asynchronous I/O ）

同步
异步

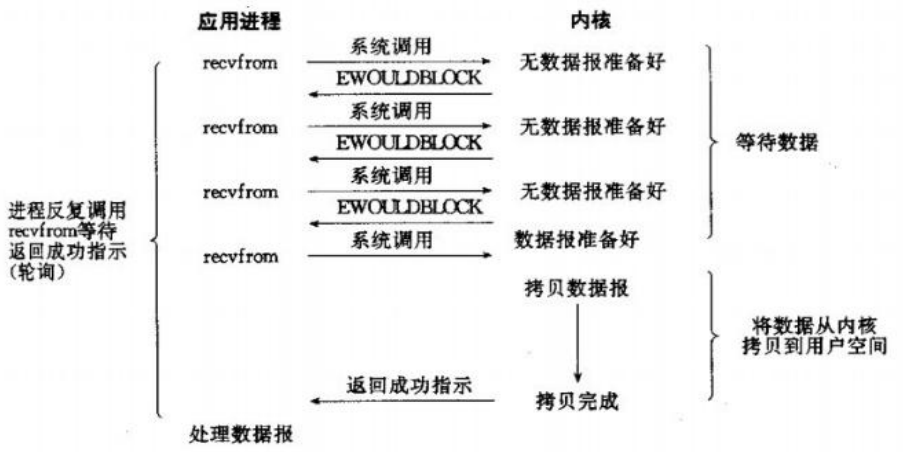
阻塞I/O 模型：



应用程序调用一个 IO 函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待....数据准备好了，从内核拷贝到用户空间,IO 函数返回成功指示。

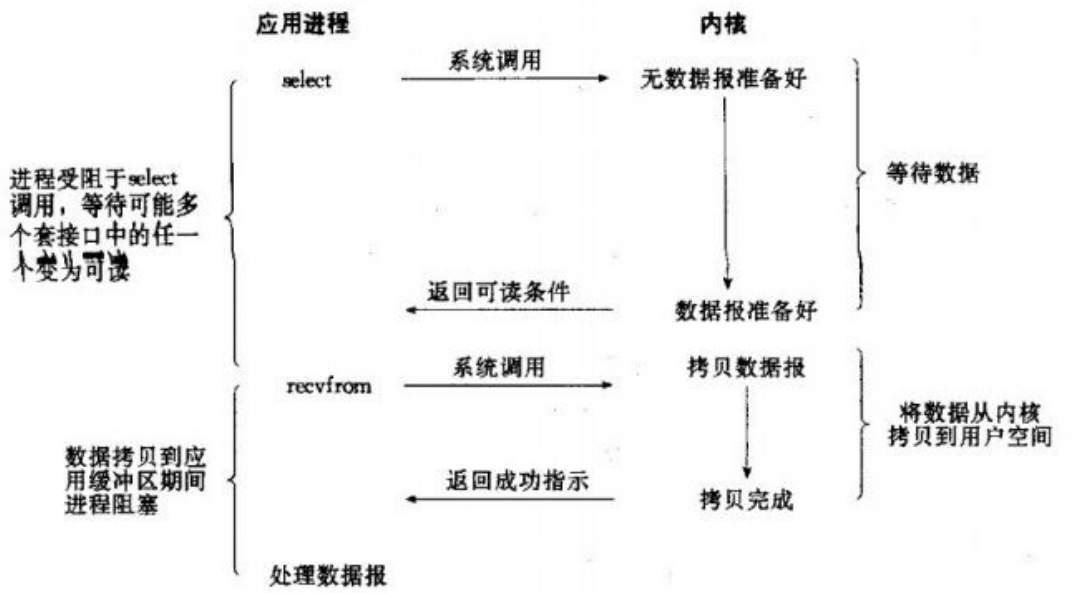
当调用 recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用 recv()函数时，未必用户空间就已经存在数据，那么此时 recv()函数就会处于等待状态。

非阻塞IO 模型



我们把一个 SOCKET 接口设置为非阻塞就是告诉内核，当所请求的 I/O 操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的 I/O 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用 CPU 的时间。上述模型绝不被推荐。

IO 复用模型：



简介：主要是 select 和 epoll 两个系统调用；对一个 IO 端口，两次调用，两次返回，比阻塞 IO 并没有什么优越性；关键是能实现同时对多个 IO 端口进行监听；

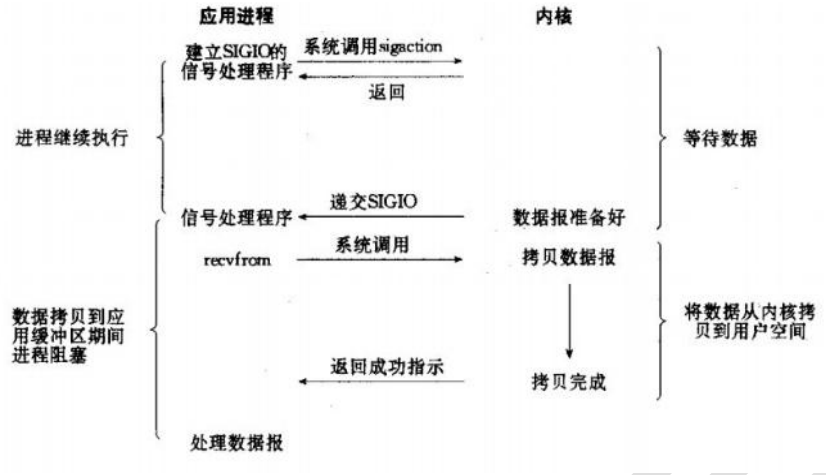
I/O 复用模型会用到 select、poll、epoll 函数，这几个函数也会使进程阻塞，但是和阻塞 I/O 所不同的的，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

当用户进程调用了 select，那么整个进程会被 block；而同时，kernel 会“监视”所有 select 负责的 socket；当任何一个 socket 中的数据准备好了，select 就会返回。这个时候，用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。

这个图和 blocking IO 的图其实并没有太大的不同，事实上还更差一些。因为这里需要使用两个系统调用(select 和 recvfrom)，而 blocking IO 只调用了一个系统调用(recvfrom)。但是，用 select 的优势在于它可以同时处理多个 connection。（多说一句：所以，如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking IO 的 web server 性能更好，可能延迟还更大。select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

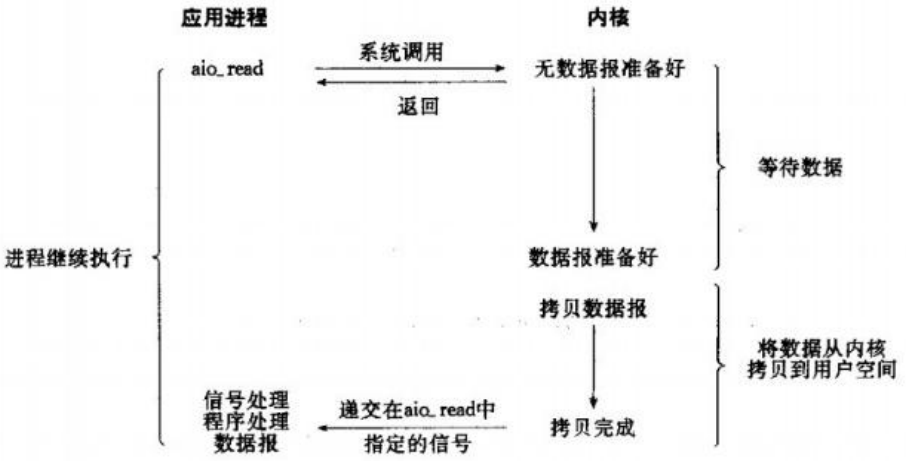
信号驱动IO

简介：两次调用，两次返回；



首先我们允许套接口进行信号驱动 I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。

异步IO 模型



当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作

5 个 I/O 模型的比较

不同 I/O 模型的区别，其实主要在等待数据和数据复制这两个时间段不同，图形中已经表示得很清楚了。

select、poll、epoll 的区别？ ：

select，poll，epoll 都是 操作系统实现 IO 多路复用的机制。 我们知道，I/O 多路复用就通过一种机制，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。那么这三种机制有什么区别呢。

1、支持一个进程所能打开的最大连接数

select	单个进程所能打开的最大连接数有 FD_SETSIZE 宏定义，其大小是 32 个整数的大小（在 32 位的机器上，大小就是 32*32，同理 64 位机器上 FD_SETSIZE 为 32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响。
poll	poll 本质上和 select 没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的
epoll	虽然连接数有上限，但是很大，1G 内存的机器上可以打开 10 万左右的连接，2G 内存的机器可以打开 20 万左右的连接

2、FD 剧增后带来的 IO 效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着 FD 的增加会造成遍历速度慢的“线性下降性能问题”。
poll	同上

epoll	因为 epoll 内核中实现是根据每个 fd 上的 callback 函数来实现的，只有活跃的 socket 才会主动调用 callback，所以在活跃 socket 较少的情况下，使用 epoll 没有前面两者的线性下降的性能问题，但是所有 socket 都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	同上
epoll	epoll 通过内核和用户空间共享一块内存来实现的。

总结：

综上，在选择 select，poll，epoll 时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看 epoll 的性能最好，但是在连接数少并且连接都十分活跃的情况下，select 和 poll 的性能可能比 epoll 好，毕竟 epoll 的通知机制需要很多函数回调。

2、select 低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

补充知识点：

Level_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

Edge_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用 epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！

select(),poll()模型都是水平触发模式，信号驱动 IO 是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。

网络编程里通用常识

既然是通信，那么是肯定是有两个对端的，（就和 James 老师去大保健一样的，一个人怎么大保健呢？必须要有 james 老师和技师两个人才能进行，james 老师总不能在大保健里自娱自乐，那还去大保健干嘛？那么在大保健里提供服务的场所叫会所或者某某中心，具体提供服务的那个人叫技师，享受服务的那个人叫 james 老师）。在通信编程里提供服务的叫服务端，连接服务端使用服务的叫客户端。在开发过程中，如果类的名字有 Server 或者 ServerSocket 的，表示这个类是给服务端用的，如果类的名字只有 Socket 的，那么表示这是负责具体的网

络读写的。那么对于服务端来说 ServerSocket 就只是个场所，具体和客户端沟通的还是一个一个的 socket，所以在通信编程里，ServerSocket 并不负责具体的网络读写，ServerSocket 就只是负责接收客户端连接后，新启一个 socket 来和客户端进行沟通。这一点对所有模式的通信编程都是适用的。

在通信编程里，我们关注的其实也就是三个事情：连接（客户端连接服务器，服务器等待和接收连接）、读网络数据、写网络数据，所有模式的通信编程都是围绕着这三件事情进行的。

原生 JDK 网络编程 BIO

服务端提供 IP 和监听端口，客户端通过连接操作想服务端监听的地址发起连接请求，通过三次握手连接，如果连接成功建立，双方就可以通过套接字进行通信。

传统的同步阻塞模型开发中，ServerSocket 负责绑定 IP 地址，启动监听端口；Socket 负责发起连接操作。连接成功后，双方通过输入和输出流进行同步阻塞式通信。

传统 BIO 通信模型：采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理没处理完成后，通过输出流返回应答给客户端，线程销毁。即典型的一请求一应答模型。

该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈 1:1 的正比关系，Java 中的线程也是比较宝贵的系统资源，线程数量快速膨胀后，系统的性能将急剧下降，随着访问量的继续增大，系统最终就**死-掉-了**。

为了改进这种一连接一线程的模型，我们可以使用线程池来管理这些线程，实现 1 个或多个线程处理 N 个客户端的模型（但是底层还是使用的同步阻塞 I/O），通常被称为“伪异步 I/O 模型“。

我们知道，如果使用 CachedThreadPool 线程池（不限制线程数量，如果不清楚请参考文首提供的文章），其实除了能自动帮我们管理线程（复用），看起来也就像是 1:1 的客户端：线程数模型，而使用 FixedThreadPool 我们就有效的控制了线程的最大数量，保证了系统有限的资源的控制，实现了 N:M 的伪异步 I/O 模型。

但是，正因为限制了线程数量，如果发生读取数据较慢时（比如数据量大、网络传输慢等），大量并发的情况下，其他接入的消息，只能一直等待，这就是最大的弊端。

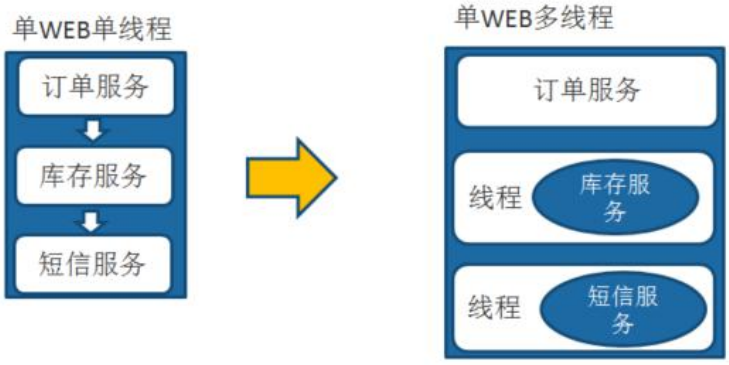
如何使用，参见模块 bio 下的代码

BIO 应用-RPC 框架

为什么要有 RPC？

我们最开始开发的时候，一个应用一台机器，将所有功能都写在一起，比如说比较常见的电商场景。

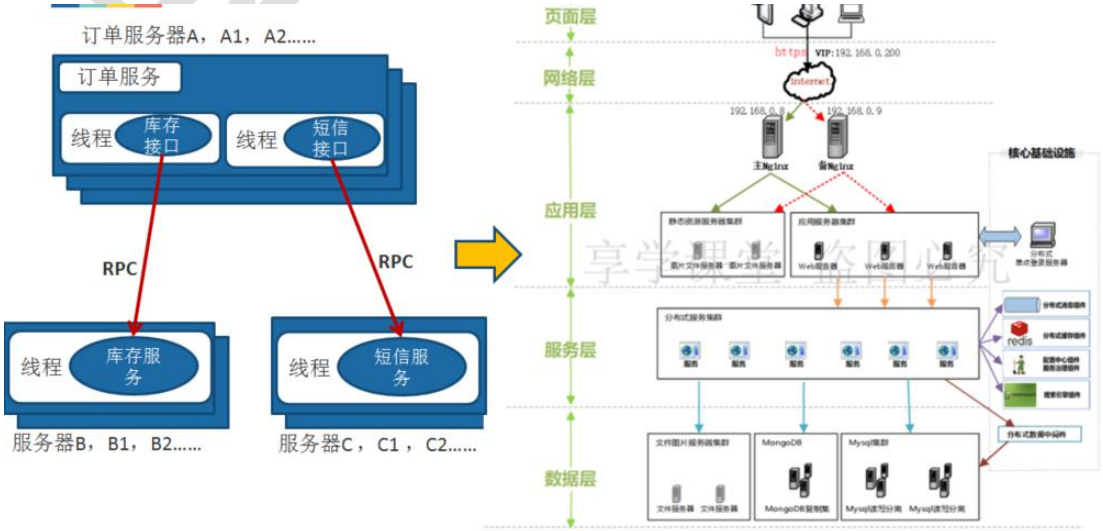
随着我们业务的发展，我们需要提示性能了，我们会怎么做？将不同的业务功能放到线程里来实现异步和提升性能。



但是业务越来越复杂，业务量越来越大，单个应用或者一台机器的资源是肯定背负不起的，这个时候，我们会怎么做？将核心业务抽取出来，作为独立的服务，放到其他服务器上或者形成集群。这个时候就会请出 RPC，系统变为分布式的架构。

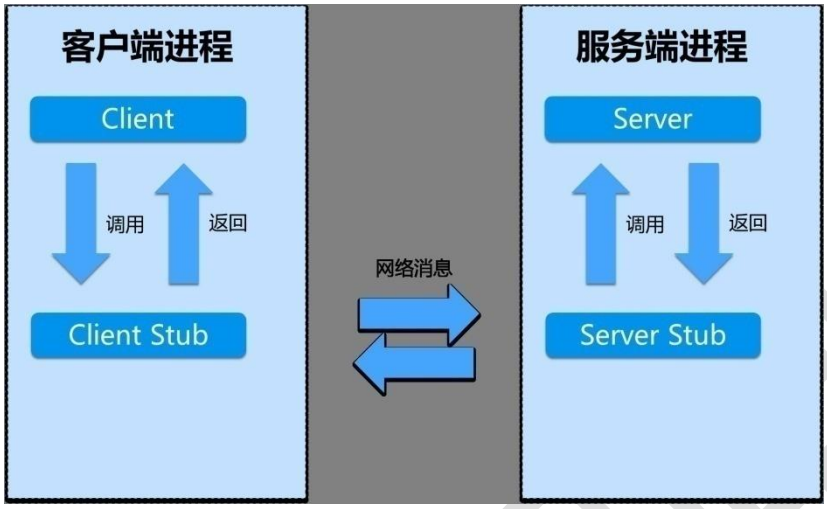
为什么说千万级流量分布式、微服务架构必备的 RPC 框架？和 LocalCall 的代码进行比较，因为引入 rpc 框架对我们现有的代码影响最小，同时又可以帮我们实现架构上的扩展。现在的开源 rpc 框架，有什么？dubbo，grpc 等等

当服务越来越多，各种 rpc 之间的调用会越来越复杂，这个时候我们会引入中间件，比如说 MQ、缓存，同时架构上整体往微服务去迁移，引入了各种比如容器技术 docker，DevOps 等等。最终会变为如图所示来应付千万级流量，但是不管怎样，rpc 总是会占有一席之地。



什么是RPC？

RPC（Remote Procedure Call ——远程过程调用），它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络的技术。



一次完整的 RPC 同步调用流程：

1）服务消费方（client）以本地调用方式调用客户端存根；

2）什么叫客户端存根？就是远程方法在本地的模拟对象，一样的也有方法名，也有方法参数，client stub 接收到调用后负责将方法名、方法的参数等包装，并将包装后的信息通过网络发送到服务端；

3）服务端收到消息后，交给代理存根在服务器的部分后进行解码为实际的方法名和参数

4） server stub 根据解码结果调用服务器上本地的实际服务；

5）本地服务执行并将结果返回给 server stub；

6）server stub 将返回结果打包成消息并发送至消费方；

7）client stub 接收到消息，并进行解码；

8）服务消费方得到最终结果。

RPC 框架的目标就是要中间步骤都封装起来，让我们进行远程方法调用的时候感觉到就像在本地调用一样。

RPC 和 HTTP

rpc 字面意思就是远程过程调用，只是对不同应用间相互调用的一种描述，一种思想。具体怎么调用？实现方式可以是最直接的 tcp 通信，也可以是 http 方式，在很多的消息中间件的技术书籍里，甚至还有使用消息中间件来实现 RPC 调用的，我们知道的 dubbo 是基于 tcp 通信的，gRPC 是 Google 公布的开源软件，基于最新的 HTTP2.0 协议，底层使用到了 Netty 框架的支持。所以总结来说，rpc 和 http 是完全两个不同层级的东西，他们之间并没有什么可比性。

实现 RPC 框架

实现 RPC 框架需要解决的那些问题

代理问题

代理本质上是要解决什么问题？要解决的是被调用的服务本质上是远程的服务，但是调用者不知道也不关心，调用者只要结果，具体的事情由代理的那个对象来负责这件事。既然是远程代理，当然是要用代理模式了。

代理(Proxy)是一种设计模式,即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能。那我们这里额外的功能操作是干什么，通过网络访问远程服务。

jdk 的代理有两种实现方式：静态代理和动态代理。

序列化问题

序列化问题在计算机里具体是什么？我们的方法调用，有方法名，方法参数，这些可能是字符串，可能是我们自己定义的 java 的类，但是在网络上传输或者保存在硬盘的时候，网络或者硬盘并不认得什么字符串或者 javabean，它只认得二进制的 01 串，怎么办？要进行序列化，网络传输后要进行实际调用，就要把二进制的 01 串变回我们实际的 java 的类，这个叫反序列化。java 里已经为我们提供了相关的机制 Serializable。

通信问题

我们在用序列化把东西变成了可以在网络上传输的二进制的 01 串，但具体如何通过网络传输？使用 JDK 为我们提供的 BIO。

登记的服务实例化

登记的服务有可能在我们的系统中就是一个名字，怎么变成实际执行的对象实例，当然是使用反射机制。

反射机制是什么？

反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

反射机制能做什么

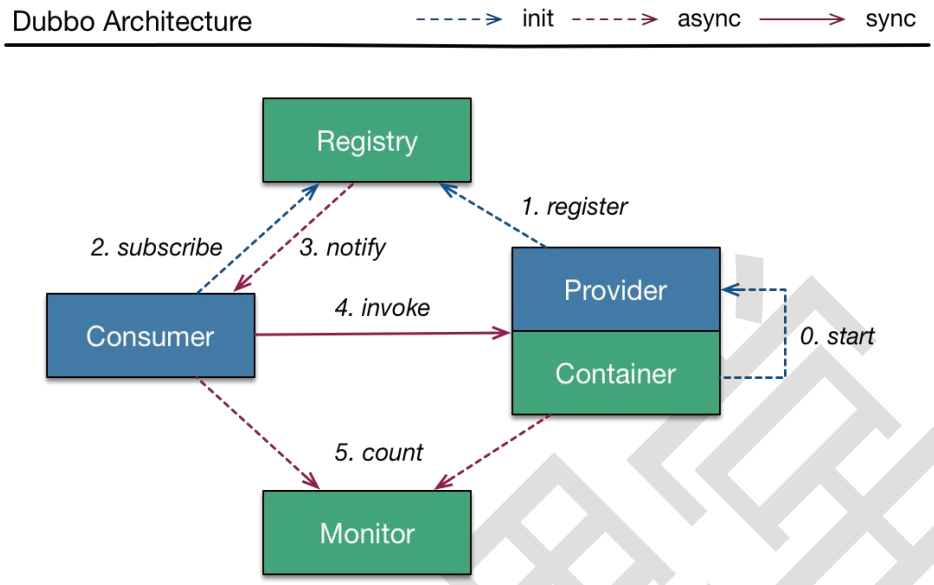
反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类；
- 在运行时构造任意一个类的对象；
- 在运行时判断任意一个类所具有的成员变量和方法；
- 在运行时调用任意一个对象的方法；
- 生成动态代理。

最后成型的代码参见模块 rpc-client 和 rpc-server

实现后的思考

Dubbo



在 Dubbo 里：

服务容器负责启动，加载，运行服务提供者。

服务提供者在启动时，向注册中心注册自己提供的服务。

服务消费者在启动时，向注册中心订阅自己所需的服务。

注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

我们的实现和 Dubbo 的比较可以看到

1、性能欠缺，表现在网络通信机制，序列化机制等等

2、负载均衡、容灾和集群功能很弱

3、服务的注册和发现机制也很差劲

Dubbo 和 SpringCloud 哪个更好

协议上比较：http 相对更规范，更标准，更通用，无论哪种语言都支持 http 协议。如果你是对外开放 API，例如开放平台，外部的编程语言多种多样，你无法拒绝对每种语言的支持，相应的，如果采用 http，无疑在你实现 SDK 之前，支持了所有语言，所以，现在开源中间件，基本最先支持的几个协议都包含 RESTful。

RPC 协议性能要高的多，例如 Protobuf、Thrift、Kyro 等，（如果算上序列化）吞吐量大概能达到 http 的二倍。响应时间也更为出色。千万不要小看这点性能

损耗，公认的，微服务做的比较好的，例如，netflix、阿里，曾经都传出过为了提升性能而合并服务。

服务全面上比较：当然是 springloud 更胜一筹，但也就意味着在使用 springloud 上其实更重量级一点，dubbo 目前版本专注于服务治理，使用上更轻量一点。

就国内的热度来说，如果我们看百度指数的查询结果，springloud 和 dubbo 几乎是半斤八两，dubbo 相比起来还略胜一筹

总的来说对外开放的服务推荐采用 RESTful，内部调用推荐采用 RPC 方式。当然不能一概而论，还要看具体的业务场景。

原生 JDK 网络编程- NIO

什么是 NIO？

NIO 库是在 JDK 1.4 中引入的。NIO 弥补了原来的 I/O 的不足，它在标准 Java 代码中提供了高速的、面向块的 I/O。NIO 翻译成 no-blocking io 或者 new io 都说得通。

和 BIO 的主要区别

面向流与面向缓冲

Java NIO 和 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。 Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO 的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

阻塞与非阻塞 IO

Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

选择器（Selectors）

Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

NIO 主要有三个核心部分组成：

buffer 缓冲区、Channel 管道、Selector 选择器

Selector

Selector 的英文含义是“选择器”，也可以称为为“轮询代理器”、“事件订阅器”、“channel 容器管理机”都行。

应用程序将向 Selector 对象注册需要它关注的 Channel，以及具体的某一个 Channel 会对哪些 IO 事件感兴趣。Selector 中也会维护一个“已经注册的 Channel”的容器。

Channels

通道，被建立的一个应用程序和操作系统交互事件、传递内容的渠道（注意是连接到操作系统）。那么既然是和操作系统进行内容的传递，那么说明应用程序可以通过通道读取数据，也可以通过通道向操作系统写数据，而且可以同时进行读写。

- 所有被 Selector（选择器）注册的通道，只能是继承了 SelectableChannel 类的子类。
- ServerSocketChannel：应用服务器程序的监听通道。只有通过这个通道，应用程序才能向操作系统注册支持“多路复用 IO”的端口监听。同时支持 UDP 协议和 TCP 协议。
- ScoketChannel：TCP Socket 套接字的监听通道，一个 Socket 套接字对应了一个客户端 IP：端口 到 服务器 IP：端口的通信连接。

通道中的数据总是要先读到一个 Buffer，或者总是要从一个 Buffer 中写入。

buffer 缓冲区

后面会讲到

操作类型 SelectionKey

SelectionKey 是一个抽象类,表示 selectableChannel 在 Selector 中注册的标识.每个 Channel 向 Selector 注册时,都将会创建一个 selectionKey。选择键将 Channel 与 Selector 建立了关系,并维护了 channel 事件。

可以通过 cancel 方法取消键,取消的键不会立即从 selector 中移除,而是添加到 cancelledKeys 中,在下一次 select 操作时移除它.所以在调用某个 key 时,需要使用 isValid 进行校验.

在向 Selector 对象注册感兴趣的事件时，JAVA NIO 共定义了四种：OP_READ、OP_WRITE、OP_CONNECT、OP_ACCEPT（定义在 SelectionKey 中），分别对应读、写、请求连接、接受连接等网络 Socket 操作。

ServerSocketChannel 和 SocketChannel 可以注册自己感兴趣的操作类型，当对应操作类型的就绪条件满足时 OS 会通知 channel，下表描述各种 Channel 允许注册的操作类型，Y 表示允许注册，N 表示不允许注册，其中服务器 SocketChannel 指由服务器 ServerSocketChannel.accept()返回的对象。

	OP_READ	OP_WRITE	OP_CONNECT	OP_ACCEPT
服务器 ServerSocketChannel				**Y**
服务器 SocketChannel	**Y**	**Y**		
客户端 SocketChannel	**Y**	**Y**	**Y**	

服务器启动 ServerSocketChannel，关注 OP_ACCEPT 事件，

客户端启动 SocketChannel，连接服务器，关注 OP_CONNECT 事件

服务器接受连接，启动一个服务器的 SocketChannel，这个 SocketChannel 可以关注 OP_READ、OP_WRITE 事件，一般连接建立后会直接关注 OP_READ 事件

客户端这边的客户端 SocketChannel 发现连接建立后，可以关注 OP_READ、OP_WRITE 事件，一般是需要客户端需要发送数据了才关注 OP_READ 事件

连接建立后客户端与服务器端开始相互发送消息（读写），根据实际情况来关注 OP_READ、OP_WRITE 事件。

我们可以看看每个操作类型的就绪条件。

操作类型	**就绪条件及说明**
OP_READ	当操作系统读缓冲区有数据可读时就绪。并非时刻都有数据可读，所以一般需要注册该操作，仅当有就绪时才发起读操作，有的放矢，避免浪费 CPU。
OP_WRITE	当操作系统写缓冲区有空闲空间时就绪。一般情况下写缓冲区都有空闲空间，小块数据直接写入即可，没必要注册该操作类型，否则该条件不断就绪浪费 CPU；但如果是写密集型的任务，比如文件下载等，缓冲区很可能满，注册该操作类型就很有必要，同时注意写完后取消注册。
OP_CONNECT	当 SocketChannel.connect()请求连接成功后就绪。该操作只给客户端使用。
OP_ACCEPT	当接收到一个客户端连接请求时就绪。该操作只给服务器使

	用。

原生 JDK 网络编程- Buffer

Buffer 用于和 NIO 通道进行交互。数据是从通道读入缓冲区，从缓冲区写入到通道中的。以写为例，应用程序都是将数据写入缓冲，再通过通道把缓冲的数据发送出去，读也是一样，数据总是先从通道读到缓冲，应用程序再读缓冲的数据。

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存（ 其实就是数组）。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。

重要属性

capacity

作为一个内存块，Buffer 有一个固定的大小值，也叫“capacity”.你只能往里写 capacity 个 byte、long，char 等类型。一旦 Buffer 满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

position

当你写数据到 Buffer 中时，position 表示当前的位置。初始的 position 值为 0. 当一个 byte、long 等数据写到 Buffer 后， position 会向前移动到下一个可插入数据的 Buffer 单元。position 最大可为 capacity – 1.

当读取数据时，也是从某个特定位置读。当将 Buffer 从写模式切换到读模式，position 会被重置为 0. 当从 Buffer 的 position 处读取数据时，position 向前移动到下一个可读的位置。

limit

在写模式下，Buffer 的 limit 表示你最多能往 Buffer 里写多少数据。 写模式下，limit 等于 Buffer 的 capacity。

当切换 Buffer 到读模式时， limit 表示你最多能读到多少数据。因此，当切换 Buffer 到读模式时，limit 会被设置成写模式下的 position 值。换句话说，你能读到之前写入的所有数据（limit 被设置成已写数据的数量，这个值在写模式下就是 position）

Buffer 的分配

要想获得一个 Buffer 对象首先要进行分配。 每一个 Buffer 类都有 **allocate** 方法(可以在堆上分配，也可以在直接内存上分配)。

分配 48 字节 capacity 的 ByteBuffer 的例子:ByteBuffer buf = ByteBuffer.allocate(48);

分配一个可存储 1024 个字符的 CharBuffer：CharBuffer buf = CharBuffer.allocate(1024);

wrap 方法：把一个 byte 数组或 byte 数组的一部分包装成 ByteBuffer：

ByteBuffer wrap(byte [] array)

ByteBuffer wrap(byte [] array, int offset, int length)

直接内存

HeapByteBuffer 与 DirectByteBuffer，在原理上，前者可以看出分配的 buffer 是在 heap 区域的，其实真正 flush 到远程的时候会先拷贝到直接内存，再做下一步操作；在 NIO 的框架下，很多框架会采用 DirectByteBuffer 来操作，这样分配的内存不再是在 java heap 上，而是在操作系统的 C heap 上，经过性能测试，可以得到非常快速的网络交互，在大量的网络交互下，一般速度会比 HeapByteBuffer 要快速好几倍。

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，但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现。

NIO 可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

堆外内存的优点和缺点

堆外内存，其实就是不受 JVM 控制的内存。相比于堆内内存有几个优势：

1 减少了垃圾回收的工作，因为垃圾回收会暂停其他的工作（可能使用多线程或者时间片的方式，根本感觉不到）

2 加快了复制的速度。因为堆内在 flush 到远程时，会先复制到直接内存（非堆内存），然后在发送；而堆外内存相当于省略掉了这个工作。

而福之祸所依，自然也有不好的一面：

1 堆外内存难以控制，如果内存泄漏，那么很难排查

2 堆外内存相对来说，不适合存储很复杂的对象。一般简单的对象或者扁平化的比较适合。

直接内存（堆外内存）与堆内存比较

直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显

直接内存 IO 读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显

Buffer 的读写

向 Buffer 中写数据

写数据到 Buffer 有两种方式：

- 读取 Channel 写到 Buffer。
- 通过 Buffer 的 put()方法写到 Buffer 里。

从 Channel 写到 Buffer 的例子

```
int bytesRead = inChannel.read(buf); //read into buffer.
```

通过 put 方法写 Buffer 的例子：

```
buf.put(127);
```

put 方法有很多版本，允许你以不同的方式把数据写入到 Buffer 中。例如，写到一个指定的位置，或者把一个字节数组写入到 Buffer。 更多 Buffer 实现的细节参考 JavaDoc。

flip()方法

flip 方法将 Buffer 从写模式切换到读模式。调用 flip()方法会将 position 设回 0，并将 limit 设置成之前 position 的值。

换句话说，position 现在用于标记读的位置，limit 表示之前写进了多少个 byte、char 等 —— 现在能读取多少个 byte、char 等。

从 Buffer 中读取数据

从 Buffer 中读取数据有两种方式：

1. 从 Buffer 读取数据写入到 Channel。
2. 使用 get()方法从 Buffer 中读取数据。

从 Buffer 读取数据到 Channel 的例子：

```
int bytesWritten = inChannel.write(buf);
```

使用 get()方法从 Buffer 中读取数据的例子

```
byte aByte = buf.get();
```

get 方法有很多版本，允许你以不同的方式从 Buffer 中读取数据。例如，从指定 position 读取，或者从 Buffer 中读取数据到字节数组。更多 Buffer 实现的细节参考 JavaDoc。

使用 Buffer 读写数据常见步骤：

1. 写入数据到 Buffer
2. 调用 flip()方法
3. 从 Buffer 中读取数据
4. 调用 clear()方法或者 compact()方法

当向 buffer 写入数据时，buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip()方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear()或 compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

其他常用操作

rewind()方法

Buffer.rewind()将 position 设回 0，所以你可以重读 Buffer 中的所有数据。limit 保持不变，仍然表示能从 Buffer 中读取多少个元素（byte、char 等）。

clear()与 compact()方法

一旦读完 Buffer 中的数据，需要让 Buffer 准备好再次被写入。可以通过 clear()或 compact()方法来完成。

如果调用的是 clear()方法，position 将被设回 0，limit 被设置成 capacity 的值。换句话说，Buffer 被清空了。Buffer 中的数据并未清除，只是这些标记告诉我们可以从哪里开始往 Buffer 里写数据。

如果 Buffer 中有一些未读的数据，调用 clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果 Buffer 中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用 compact()方法。

compact()方法将所有未读的数据拷贝到 Buffer 起始处。然后将 position 设到最后一个未读元素正后面。limit 属性依然像 clear()方法一样，设置成 capacity。现在 Buffer 准备好写数据了，但是不会覆盖未读的数据。

mark()与 reset()方法

通过调用 Buffer.mark()方法，可以标记 Buffer 中的一个特定 position。之后可以通过调用 Buffer.reset()方法恢复到这个 position。例如：

```
buffer.mark();//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset(); //set position back to mark.
```

equals()与 compareTo()方法

可以使用 equals()和 compareTo()方法两个 Buffer。

equals()

当满足下列条件时，表示两个 Buffer 相等：

1. 有相同的类型（byte、char、int 等）。
2. Buffer 中剩余的 byte、char 等的个数相等。
3. Buffer 中所有剩余的 byte、char 等都相同。

如你所见，equals 只是比较 Buffer 的一部分，不是每一个在它里面的元素都比较。实际上，它只比较 Buffer 中的剩余元素。

compareTo()方法

compareTo()方法比较两个 Buffer 的剩余元素(byte、char 等)， 如果满足下列条件，则认为一个 Buffer“小于”另一个 Buffer：

1. 第一个不相等的元素小于另一个 Buffer 中对应的元素 。
2. 所有元素都相等，但第一个 Buffer 比另一个先耗尽(第一个 Buffer 的元素个数比另一个少)。

Buffer 方法总结

limit(), limit(1	其中读取和设置这 4 个属性的方法的命名和 jQuery 中的 val(),val(10)类似，一个负

0)等	责 get，一个负责 set
reset()	把 position 设置成 mark 的值，相当于之前做过一个标记，现在要退回到之前标记的地方
clear()	position = 0;limit = capacity;mark = -1; 有点初始化的味道，但是并不影响底层 byte 数组的内容
flip()	limit = position;position = 0;mark = -1; 翻转，也就是让 flip 之后的 position 到 limit 这块区域变成之前的 0 到 position 这块，翻转就是将一个处于存数据状态的缓冲区变为一个处于准备取数据的状态
rewind()	把 position 设为 0，mark 设为-1，不改变 limit 的值
remaining()	return limit - position;返回 limit 和 position 之间相对位置差
hasRemaining()	return position < limit 返回是否还有未读内容
compact()	把从 position 到 limit 中的内容移到 0 到 limit-position 的区域内，position 和 limit 的取值也分别变成 limit-position、capacity。如果先将 positon 设置到 limit，再 compact，那么相当于 clear()
get()	相对读，从 position 位置读取一个 byte，并将 position+1，为下次读写作准备
get(int index)	绝对读，读取 byteBuffer 底层的 bytes 中下标为 index 的 byte，不改变 position
get(byte[] dst, int offset, int length)	从 position 位置开始相对读，读 length 个 byte，并写入 dst 下标从 offset 到 offset+length 的区域
put(byte b)	相对写，向 position 的位置写入一个 byte，并将 postion+1，为下次读写作准备
put(int index, byte b)	绝对写 ,向 byteBuffer 底层的 bytes 中下标为 index 的位置插入 byte b ,不改变 position
put(ByteBuffer src)	用相对写，把 src 中可读的部分（也就是 position 到 limit）写入此 byteBuffer
put(byte[] src, int offset, int length)	从 src 数组中的 offset 到 offset+length 区域读取数据并使用相对写写入此 byteBuffer

Buffer 相关的代码参见模块 nio 下包 cn.enjoyedu.nio.buffer

与 NIO 编程相关的代码参见 nio 下包 cn.enjoyedu.nio.nio

原生 JDK 网络编程- NIO 之 Reactor 模式

“反应”器名字中”反应“的由来：

“反应”即“倒置”，“控制逆转”,具体事件处理程序不调用反应器，而向反应器注册一个事件处理器，表示自己对某些事件感兴趣，有时间来了，具体事件处理程序通过事件处理器对某个指定的事件发生做出反应；这种控制逆转又称为“好莱坞法则”（不要调用我，让我来调用你）

例如，路人甲去做男士 SPA，前台的接待小姐接待了路人甲，路人甲现在只对 10000 技师感兴趣，就告诉接待小姐，等 10000 技师上班了或者是空闲了，通知我。等 james 接到通知了，做出了反应，把 10000 技师占住了，然后，路人甲想起上一次的那个 10000 号房间不错，设备舒适，灯光暧昧，又告诉前台的接待小姐，我对 10000 号房间很感兴趣，房间空出来了就告诉我，我现在先和 10000 这个小姐聊下人生，10000 号房间空出来了，james 接到通知了，路人甲再次做出了反应。路人甲就是具体事件处理程序，前台的接待小姐就是所谓的反应器，10000 技师和 10000 号房间空闲了就是事件，路人甲只对这两个事件感兴趣，其他，比如 10001 号技师或者 10002 号房间空闲了也是事件，但是路人甲不感兴趣。

单线程 Reactor 模式流程：

① 服务器端的 Reactor 是一个线程对象，该线程会启动事件循环，并使用 Selector(选择器)来实现 IO 的多路复用。注册一个 Acceptor 事件处理器到 Reactor 中，Acceptor 事件处理器所关注的事件是 ACCEPT 事件，这样 Reactor 会监听客户端向服务器端发起的连接请求事件(ACCEPT 事件)。

② 客户端向服务器端发起一个连接请求，Reactor 监听到了该 ACCEPT 事件的发生并将该 ACCEPT 事件派发给相应的 Acceptor 处理器来进行处理。Acceptor 处理器通过 accept()方法得到与这个客户端对应的连接(SocketChannel)，然后将该连接所关注的 READ 事件以及对应的 READ 事件处理器注册到 Reactor 中，这样一来 Reactor 就会监听该连接的 READ 事件了。

③ 当 Reactor 监听到有读或者写事件发生时，将相关的事件派发给对应的处理器进行处理。比如，读处理器会通过 SocketChannel 的 read()方法读取数据，此时 read()操作可以直接读取到数据，而不会堵塞与等待可读的数据到来。

④ 每当处理完所有就绪的感兴趣的 I/O 事件后，Reactor 线程会再次执行 select()阻塞等待新的事件就绪并将其分派给对应处理器进行处理。

注意，Reactor 的单线程模式的单线程主要是针对于 I/O 操作而言，也就是所有的 I/O 的 accept()、read()、write()以及 connect()操作都在一个线程上完成的。

但在目前的单线程 Reactor 模式中，不仅 I/O 操作在该 Reactor 线程上，连非 I/O 的业务操作也在该线程上进行处理了，这可能会大大延迟 I/O 请求的响应。所以我们应该将非 I/O 的业务逻辑操作从 Reactor 线程上卸载，以此来加速 Reactor 线程对 I/O 请求的响应。

单线程 Reactor，工作者线程池

与单线程 Reactor 模式不同的是，添加了一个工作者线程池，并将非 I/O 操作从 Reactor 线程中移出转交给工作者线程池来执行。这样能够提高 Reactor 线程的 I/O 响应，不至于因为一些耗时的业务逻辑而延迟对后面 I/O 请求的处理。

使用线程池的优势：

① 通过重用现有的线程而不是创建新线程，可以在处理多个请求时分摊在线程创建和销毁过程产生的巨大开销。

② 另一个额外的好处是，当请求到达时，工作线程通常已经存在，因此不会由于等待创建线程而延迟任务的执行，从而提高了响应性。

③ 通过适当调整线程池的大小，可以创建足够多的线程以便使处理器保持忙碌状态。同时还可以防止过多线程相互竞争资源而使应用程序耗尽内存或失败。

改进的版本中，所以的 I/O 操作依旧由一个 Reactor 来完成，包括 I/O 的 accept()、read()、write()以及 connect()操作。

对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发或大数据量的应用场景却不合适，主要原因如下：

① 一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的读取和发送；

② 当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈；

多 Reactor 线程模式

Reactor 线程池中的每一 Reactor 线程都会有自己的 Selector、线程和分发的事件循环逻辑。

mainReactor 可以只有一个，但 subReactor 一般会有多个。mainReactor 线程主要负责接收客户端的连接请求，然后将接收到的 SocketChannel 传递给 subReactor，由 subReactor 来完成和客户端的通信。

流程：

① 注册一个 Acceptor 事件处理器到 mainReactor 中，Acceptor 事件处理器所关注的事件是 ACCEPT 事件，这样 mainReactor 会监听客户端向服务器端发起的连接请求事件(ACCEPT 事件)。启动 mainReactor 的事件循环。

② 客户端向服务器端发起一个连接请求，mainReactor 监听到了该 ACCEPT 事件并将该 ACCEPT 事件派发给 Acceptor 处理器来进行处理。Acceptor 处理器通过 accept()方法得到与这个客户端对应的连接(SocketChannel)，然后将这个 SocketChannel 传递给 subReactor 线程池。

③ subReactor 线程池分配一个 subReactor 线程给这个 SocketChannel，即，将 SocketChannel 关注的 READ 事件以及对应的 READ 事件处理器注册到 subReactor 线程中。当然你也注册 WRITE 事件以及 WRITE 事件处理器到

subReactor 线程中以完成 I/O 写操作。Reactor 线程池中的每一 Reactor 线程都会有自己的 Selector、线程和分发的循环逻辑。

④ 当有 I/O 事件就绪时，相关的 subReactor 就将事件派发给响应的处理器处理。注意，这里 subReactor 线程只负责完成 I/O 的 read()操作，在读取到数据后将业务逻辑的处理放入到线程池中完成，若完成业务逻辑后需要返回数据给客户端，则相关的 I/O 的 write 操作还是会被提交回 subReactor 线程来完成。

注意，所以的 I/O 操作(包括，I/O 的 accept()、read()、write()以及 connect()操作)依旧还是在 Reactor 线程(mainReactor 线程 或 subReactor 线程)中完成的。Thread Pool(线程池)仅用来处理非 I/O 操作的逻辑。

多 Reactor 线程模式将“接受客户端的连接请求”和“与该客户端的通信”分在了两个 Reactor 线程来完成。mainReactor 完成接收客户端连接请求的操作，它不负责与客户端的通信，而是将建立好的连接转交给 subReactor 线程来完成与客户端的通信，这样一来就不会因为 read()数据量太大而导致后面的客户端连接请求得不到即时处理的情况。并且多 Reactor 线程模式在海量的客户端并发请求的情况下，还可以通过实现 subReactor 线程池来将海量的连接分发给多个 subReactor 线程，在多核的操作系统中这能大大提升应用的负载和吞吐量。

Netty 服务端使用了“多 Reactor 线程模式”

和观察者模式的区别

观察者模式：

也可以称为为 发布-订阅 模式，主要适用于多个对象依赖某一个对象的状态并，当某对象状态发生改变时，要通知其他依赖对象做出更新。是一种一对多的关系。当然，如果依赖的对象只有一个时，也是一种特殊的一对一关系。通常，观察者模式适用于消息事件处理，监听者监听到事件时通知事件处理者对事件进行处理（这一点上面有点像是回调，容易与反应器模式和前摄器模式的回调搞混淆）。

Reactor 模式：

reactor 模式，即反应器模式，是一种高效的异步 IO 模式，特征是回调，当 IO 完成时，回调对应的函数进行处理。这种模式并非是真正的异步，而是运用了异步的思想，当 IO 事件触发时，通知应用程序作出 IO 处理。模式本身并不调用系统的异步 IO 函数。

reactor 模式与观察者模式有点像。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。当一个主体发生改变时，所有依属体都得到通知。

3、Netty 应用

Netty 是什么？为什么要用 Netty？

为什么要用 Netty

1、虽然 JAVA NIO 框架提供了 多路复用 IO 的支持，但是并没有提供上层“信息格式”的良好封装。例如前两者并没有提供针对 Protocol Buffer、JSON 这些信息格式的封装，但是 Netty 框架提供了这些数据格式封装（基于责任链模式的编码和解码功能）；

2、NIO 的类库和 API 相当复杂，使用它来开发，需要非常熟练地掌握 Selector、ByteBuffer、ServerSocketChannel、SocketChannel 等，需要很多额外的编程技能来辅助使用 NIO,例如，因为 NIO 涉及了 Reactor 线程模型，所以必须必须对多线程和网络编程非常熟悉才能写出高质量的 NIO 程序

3、要编写一个可靠的、易维护的、高性能的 NIO 服务器应用。除了框架本身要兼容实现各类操作系统的实现外。更重要的是它应该还要处理很多上层特有服务，例如：客户端的权限、还有上面提到的信息格式封装、简单的数据读取，断连重连，半包读写，心跳等等，这些 Netty 框架都提供了响应的支持。

4、JAVA NIO 框架存在一个 poll/epoll bug：Selector doesn’t block on Selector.select(timeout)，不能 block 意味着 CPU 的使用率会变成 100%（这是底层 JNI 的问题，上层要处理这个异常实际上也好办）。当然这个 bug 只有在 Linux 内核上才能重现。

这个问题在 JDK 1.7 版本中还没有被完全解决，但是 Netty 已经将这个 bug 进行了处理。

这个 Bug 与操作系统机制有关系的，JDK 虽然仅仅是一个兼容各个操作系统平台的软件，但在 JDK5 和 JDK6 最初的版本中（严格意义上来将，JDK 部分版本都是），这个问题并没有解决，而将这个帽子抛给了操作系统方，这也就是这个 bug 最终一直到 2013 年才最终修复的原因(JDK7 和 JDK8 之间)。

为什么不用 Netty5

1. netty5 中使用了 ForkJoinPool，增加了代码的复杂度，但是对性能的改善却不明显

2. 多个分支的代码同步工作量很大

3. 作者觉得当下还不到发布一个新版本的时候

4. 在发布版本之前，还有更多问题需要调查一下，比如是否应该废弃 exceptionCaught， 是否暴露 EventExecutorChooser 等等。

为什么 Netty 使用 NIO 而不是 AIO？

Netty 不看重 Windows 上的使用，在 Linux 系统上，AIO 的底层实现仍使用 EPOLL，没有很好实现 AIO，因此在性能上没有明显的优势，而且被 JDK 封装了一层不容易深度优化。

AIO 还有个缺点是接收数据需要预先分配缓存，而不是 NIO 那种需要接收时才需要分配缓存，所以对连接数量非常大但流量小的情况，内存浪费很多。

据说 Linux 上 AIO 不够成熟，处理回调结果速度跟不上处理需求，有点像外卖员太少，顾客太多，供不应求，造成处理速度有瓶颈。

作者原话：

Not faster than NIO (epoll) on unix systems (which is true)

There is no daragram suppport

Unnecessary threading model (too much abstraction without usage)

第一个 Netty 程序

Channel

Channel 是 Java NIO 的一个基本构造。

它代表一个到实体（如一个硬件设备、一个文件、一个网络套接字或者一个能够执行一个或者多个不同的 I/O 操作的程序组件）的开放连接，如读操作和写操作

目前，可以把 Channel 看作是传入（入站）或者传出（出站）数据的载体。因此，它可以被打开或者被关闭，连接或者断开连接。

回调和 Future

一个回调其实就是一个方法，一个指向已经被提供给另外一个方法的方法的引用。这使得后者可以在适当的时候调用前者。回调在广泛的编程场景中都有应用，而且也是在操作完成后通知相关方最常见的方式之一。

Netty 在内部使用了回调来处理事件；当一个回调被触发时，相关的事件可以被一个 interface-ChannelHandler 的实现处理。

Future 提供了另一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操作的结果的占位符；它将在未来的某个时刻完成，并提供对其结果的访问。

JDK 预置了 interface java.util.concurrent.Future，但是其所提供的实现，只允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以 Netty 提供了它自己的实现——ChannelFuture，用于在执行异步操作的时候使用。

ChannelFuture 提供了几种额外的方法，这些方法使得我们能够注册一个或者多个 ChannelFutureListener 实例。监听器的回调方法 operationComplete()，将会在对应的操作完成时被调用。然后监听器可以判断该操作是成功地完成了还是出错了。如果是后者，我们可以检索产生的 Throwable。简而言之，由 ChannelFutureListener 提供的通知机制消除了手动检查对应的操作是否完成的必要。

每个 Netty 的出站 I/O 操作都将返回一个 ChannelFuture。

事件和 ChannelHandler

Netty 使用不同的事件来通知我们状态的改变或者是操作的状态。这使得我们能够基于已经发生的事件来触发适当的动作。

Netty 事件是按照它们与入站或出站数据流的相关性进行分类的。

可能由入站数据或者相关的状态更改而触发的事件包括：

连接已被激活或者连接失活； 数据读取； 用户事件； 错误事件。

出站事件是未来将会触发的某个动作的操作结果，这些动作包括：

打开或者关闭到远程节点的连接； 将数据写到或者冲刷到套接字。

每个事件都可以被分发给 ChannelHandler 类中的某个用户实现的方法。

可以认为每个 ChannelHandler 的实例都类似于一种为了响应特定事件而被执行的回调。

Netty 提供了大量预定义的可以开箱即用的 ChannelHandler 实现，包括用于各种协议（如 HTTP 和 SSL/TLS）的 ChannelHandler。

Hello，Netty！

参见模块 netty-basic 中包 cn.enjoyedu.nettybasic.echo

Netty 组件再了解

Channel、EventLoop(Group)和 ChannelFuture

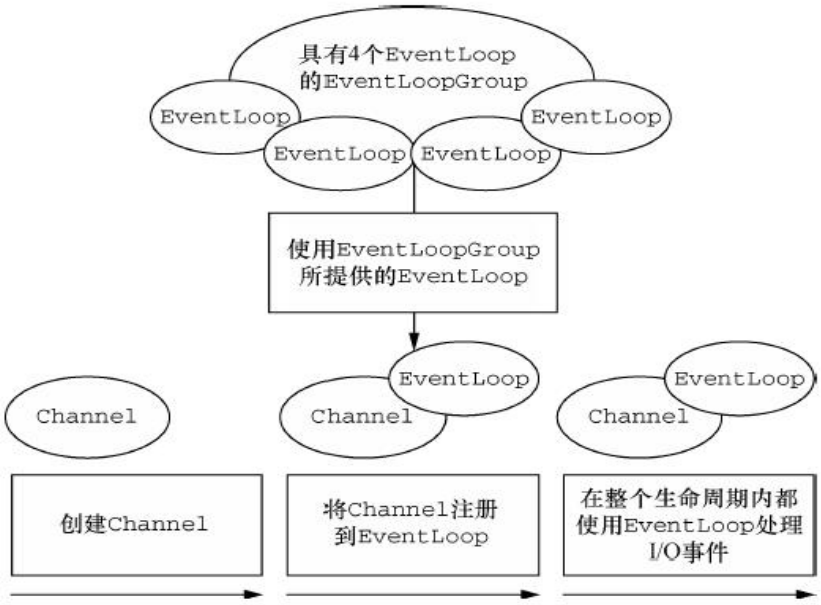
Netty 网络抽象的代表：

Channel—Socket；

EventLoop—控制流、多线程处理、并发；

ChannelFuture—异步通知。

Channel 和 EventLoop 关系如图：



Channel 接口

基本的 I/O 操作（bind()、connect()、read()和 write()）依赖于底层网络传输所提供的原语。在基于 Java 的网络编程中，其基本的构造是类 Socket。Netty 的 Channel 接口所提供的 API，被用于所有的 I/O 操作。大大地降低了直接使用 Socket 类的复杂性。此外，Channel 也是拥有许多预定义的、专门化实现的广泛类层次结构的根。

由于 Channel 是独一无二的，所以为了保证顺序将 Channel 声明为 java.lang.Comparable 的一个子接口。因此，如果两个不同的 Channel 实例都返回了相同的散列码，那么 AbstractChannel 中的 compareTo()方法的实现将会抛出一个 Error。

Channel 的生命周期状态

ChannelUnregistered ：Channel 已经被创建，但还未注册到 EventLoop

ChannelRegistered ：Channel 已经被注册到了 EventLoop

ChannelActive ：Channel 处于活动状态（已经连接到它的远程节点）。它现在可以接收和发送数据了

ChannelInactive ：Channel 没有连接到远程节点

当这些状态发生改变时，将会生成对应的事件。这些事件将会被转发给 ChannelPipeline 中的 ChannelHandler，其可以随后对它们做出响应。

最重要 Channel 的方法

eventLoop： 返回分配给 Channel 的 EventLoop

pipeline： 返回分配给 Channel 的 ChannelPipeline

isActive： 如果 Channel 是活动的，则返回 true。活动的意义可能依赖于底层的传输。例如，一个 Socket 传输一旦连接到了远程节点便是活动的，而一个 Datagram 传输一旦被打开便是活动的。

localAddress： 返回本地的 SokcetAddress

remoteAddress： 返回远程的 SocketAddress

write： 将数据写到远程节点。这个数据将被传递给 ChannelPipeline，并且排队直到它被冲刷

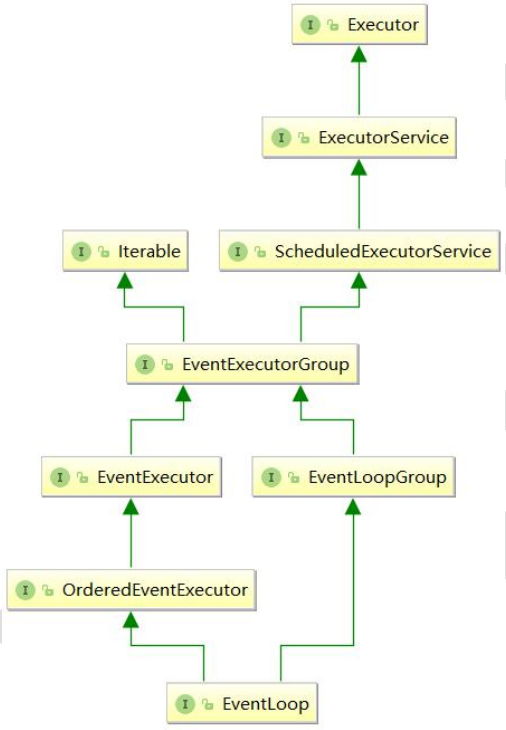
flush： 将之前已写的数据冲刷到底层传输，如一个 Socket

writeAndFlush： 一个简便的方法，等同于调用 write()并接着调用 flush()

EventLoop 和 EventLoopGroup

回想一下我们在 NIO 中是如何处理我们关心的事件的？在一个 while 循环中 select 出事件，然后依次处理每种事件。我们可以把它称为事件循环，这就是 EventLoop。interface io.netty.channel. EventLoop 定义了 Netty 的核心抽象，用于处理网络连接的生命周期中所发生的事件。

io.netty.util.concurrent 包构建在 JDK 的 java.util.concurrent 包上。而，io.netty.channel 包中的类，为了与 Channel 的事件进行交互，扩展了这些接口/类。一个 EventLoop 将由一个永远都不会改变的 Thread 驱动，同时任务（Runnable 或者 Callable）可以直接提交给 EventLoop 实现，以立即执行或者调度执行。



根据配置和可用核心的不同，可能会创建多个 EventLoop 实例用以优化资源的使用，并且单个 EventLoop 可能会被指派用于服务多个 Channel。

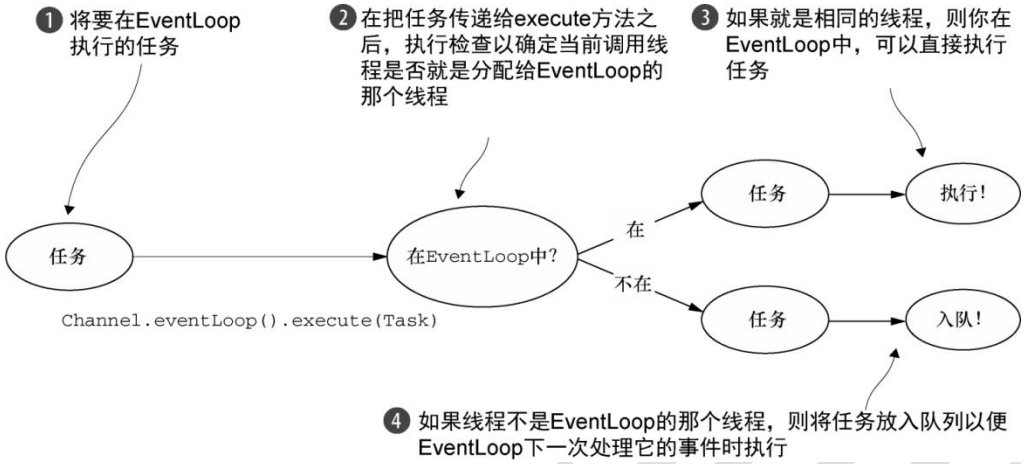
Netty 的 EventLoop 在继承了 ScheduledExecutorService 的同时，只定义了一个方法，parent()。在 Netty 4 中，所有的 I/O 操作和事件都由已经被分配给了 EventLoop 的那个 Thread 来处理。

任务调度

偶尔，你将需要调度一个任务以便稍后（延迟）执行或者周期性地执行。例如，你可能想要注册一个在客户端已经连接了 5 分钟之后触发的任务。一个常见的用例是，发送心跳消息到远程节点，以检查连接是否仍然还活着。如果没有响应，你便知道可以关闭该 Channel 了。

线程管理

在内部，当提交任务到如果（当前）调用线程正是支撑 EventLoop 的线程，那么所提交的代码块将会被（直接）执行。否则，EventLoop 将调度该任务以便稍后执行，并将它放入到内部队列中。当 EventLoop 下次处理它的事件时，它会执行队列中的那些任务/事件。

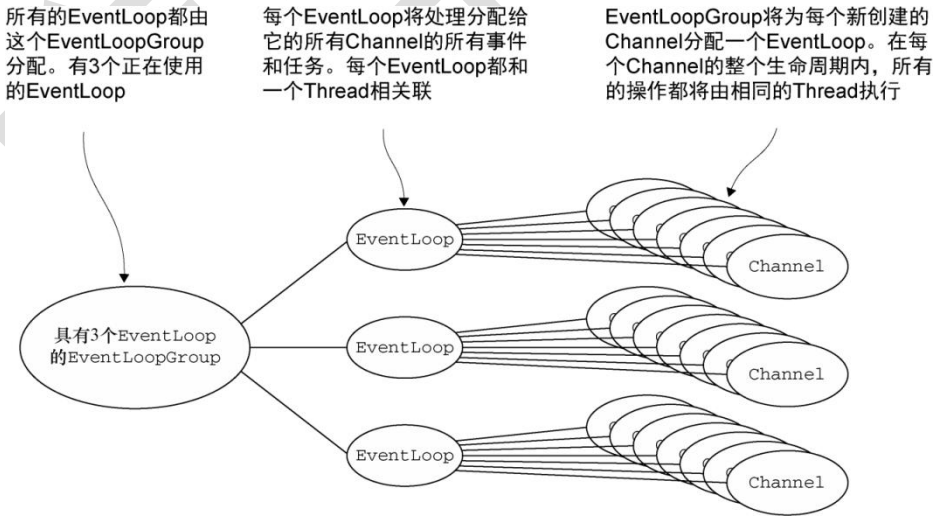


线程的分配

服务于 Channel 的 I/O 和事件的 EventLoop 则包含在 EventLoopGroup 中。

异步传输实现只使用了少量的 EventLoop（以及和它们相关联的 Thread），而且在当前的线程模型中，它们可能会被多个 Channel 所共享。这使得可以通过尽可能少量的 Thread 来支撑大量的 Channel，而不是每个 Channel 分配一个 Thread。EventLoopGroup 负责为每个新创建的 Channel 分配一个 EventLoop。在当前实现中，使用顺序循环（round-robin）的方式进行分配以获取一个均衡的分布，并且相同的 EventLoop 可能会被分配给多个 Channel。

一旦一个 Channel 被分配给一个 EventLoop，它将在它的整个生命周期中都使用这个 EventLoop（以及相关联的 Thread）。请牢记这一点，因为它可以使你从担忧你的 ChannelHandler 实现中的线程安全和同步问题中解脱出来。



需要注意，EventLoop 的分配方式对 ThreadLocal 的使用的影响。因为一个 EventLoop 通常会被用于支撑多个 Channel，所以对于所有相关联的 Channel 来说，ThreadLocal 都将是一样的。这使得它对于实现状态追踪等功能来说是个糟糕的选择。然而，在一些无状态的上

下文中，它仍然可以被用于在多个 Channel 之间共享一些重度的或者代价昂贵的对象，甚至是事件。

ChannelFuture 接口

Netty 中所有的 I/O 操作都是异步的。因为一个操作可能不会立即返回，所以我们需要一种用于在之后的某个时间点确定其结果的方法。为此，Netty 提供了 ChannelFuture 接口，其 addListener()方法注册了一个 ChannelFutureListener，以便在某个操作完成时（无论是否成功）得到通知。

可以将 ChannelFuture 看作是将来要执行的操作的结果的占位符。它究竟什么时候被执行则可能取决于若干的因素，因此不可能准确地预测，但是可以肯定的是它将会被执行。

ChannelHandler、ChannelPipeline 和 ChannelHandlerContext

ChannelHandler 接口

从应用程序开发人员的角度来看，Netty 的主要组件是 ChannelHandler，它充当了所有处理入站和出站数据的应用程序逻辑的容器。ChannelHandler 的方法是由网络事件触发的。事实上，ChannelHandler 可专门用于几乎任何类型的动作，例如将数据从一种格式转换为另外一种格式，例如各种编解码，或者处理转换过程中所抛出的异常。

举例来说，ChannelInboundHandler 是一个你将会经常实现的子接口。这种类型的 ChannelHandler 接收入站事件和数据，这些数据随后将会被你的应用程序的业务逻辑所处理。当你要给连接的客户端发送响应时，也可以从 ChannelInboundHandler 直接冲刷数据然后输出到对端。应用程序的业务逻辑通常实现在一个或者多个 ChannelInboundHandler 中。

这种类型的 ChannelHandler 接收入站事件和数据，这些数据随后将会被应用程序的业务逻辑所处理。

ChannelHandler 的生命周期

接口 ChannelHandler 定义的生命周期操作，在 ChannelHandler 被添加到 ChannelPipeline 中或者被从 ChannelPipeline 中移除时会调用这些操作。这些方法中的每一个都接受一个 ChannelHandlerContext 参数。

handlerAdded 当把 ChannelHandler 添加到 ChannelPipeline 中时被调用

handlerRemoved 当从 ChannelPipeline 中移除 ChannelHandler 时被调用

exceptionCaught 当处理过程中在 ChannelPipeline 中有错误产生时被调用

Netty 定义了下面两个重要的 ChannelHandler 子接口：

ChannelInboundHandler——处理入站数据以及各种状态变化；

ChannelOutboundHandler——处理出站数据并且允许拦截所有的操作。

ChannelInboundHandler 接口

下面列出了接口 ChannelInboundHandler 的生命周期方法。这些方法将会在数据被接收时或者与其对应的 Channel 状态发生改变时被调用。正如我们前面所提到的，这些方法和 Channel 的生命周期密切相关。

channelRegistered 当 Channel 已经注册到它的 EventLoop 并且能够处理 I/O 时被调用

channelUnregistered 当 Channel 从它的 EventLoop 注销并且无法处理任何 I/O 时被调用

channelActive 当 Channel 处于活动状态时被调用；Channel 已经连接/绑定并且已经就绪

channelInactive 当 Channel 离开活动状态并且不再连接它的远程节点时被调用

channelReadComplete 当 Channel 上的一个读操作完成时被调用

channelRead 当从 Channel 读取数据时被调用

ChannelWritabilityChanged

当 Channel 的可写状态发生改变时被调用。可以通过调用 Channel 的 isWritable()方法来检测 Channel 的可写性。与可写性相关的阈值可以通过 Channel.config().setWriteHighWaterMark()和 Channel.config().setWriteLowWaterMark()方法来设置

userEventTriggered 当 ChannelnboundHandler.fireUserEventTriggered()方法被调用时被调用。

ChannelOutboundHandler 接口

出站操作和数据将由 ChannelOutboundHandler 处理。它的方法将被 Channel、Channel-Pipeline 以及 ChannelHandlerContext 调用。

所有由 ChannelOutboundHandler 本身所定义的方法：

bind(ChannelHandlerContext,SocketAddress,ChannelPromise)

当请求将 Channel 绑定到本地地址时被调用

connect(ChannelHandlerContext,SocketAddress,SocketAddress,ChannelPromise)

当请求将 Channel 连接到远程节点时被调用

disconnect(ChannelHandlerContext,ChannelPromise)

当请求将 Channel 从远程节点断开时被调用

close(ChannelHandlerContext,ChannelPromise) 当请求关闭 Channel 时被调用

deregister(ChannelHandlerContext,ChannelPromise)

当请求将 Channel 从它的 EventLoop 注销时被调用

read(ChannelHandlerContext) 当请求从 Channel 读取更多的数据时被调用

flush(ChannelHandlerContext) 当请求通过 Channel 将入队数据冲刷到远程节点时被调用

write(ChannelHandlerContext,Object,ChannelPromise) 当请求通过 Channel 将数据写到远程节点时被调用

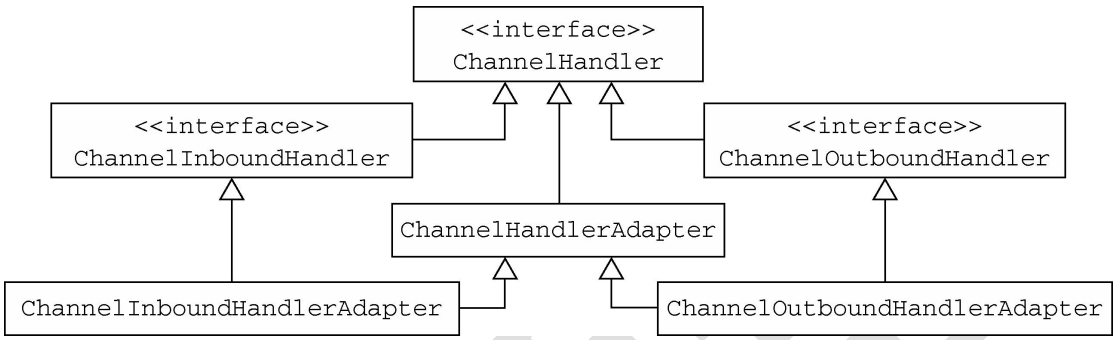
ChannelHandler 的适配器

有一些适配器类可以将编写自定义的 ChannelHandler 所需要的工作降到最低限度，因为它们提供了定义在对应接口中的所有方法的默认实现。因为你有时会忽略那些不感兴趣的

事件，所以 Netty 提供了抽象基类 ChannelInboundHandlerAdapter 和 ChannelOutboundHandlerAdapter。

你可以使用 ChannelInboundHandlerAdapter 和 ChannelOutboundHandlerAdapter 类作为自己的 ChannelHandler 的起始点。这两个适配器分别提供了 ChannelInboundHandler 和 ChannelOutboundHandler 的基本实现。通过扩展抽象类 ChannelHandlerAdapter，它们获得了它们共同的超接口 ChannelHandler 的方法。

ChannelHandlerAdapter 还提供了实用方法 isSharable()。如果其对应的实现被标注为 Sharable，那么这个方法将返回 true，表示它可以被添加到多个 ChannelPipeline。



ChannelPipeline 接口

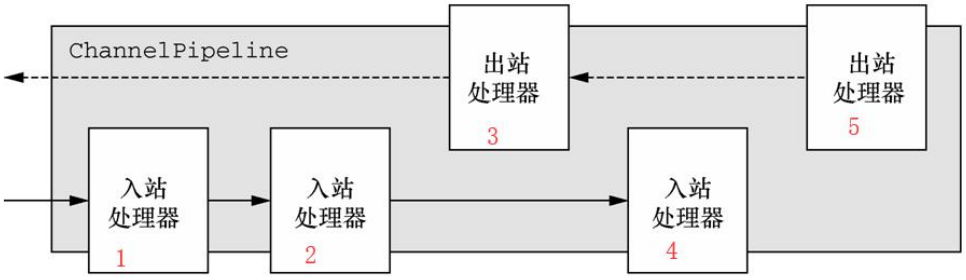
当 Channel 被创建时，它将会被自动地分配一个新的 ChannelPipeline。这项关联是永久性的；Channel 既不能附加另外一个 ChannelPipeline，也不能分离其当前的。在 Netty 组件的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

使得事件流经 ChannelPipeline 是 ChannelHandler 的工作，它们是在应用程序的初始化或者引导阶段被安装的。这些对象接收事件、执行它们所实现的处理逻辑，并将数据传递给链中的下一个 ChannelHandler。它们的执行顺序是由它们被添加的顺序所决定的。

入站和出站 ChannelHandler 可以被安装到同一个 ChannelPipeline 中。如果一个消息或者任何其他的入站事件被读取，那么它会从 ChannelPipeline 的头部开始流动，最终，数据将会到达 ChannelPipeline 的尾端，届时，所有处理就都结束了。

数据的出站运动（即正在被写的数据）在概念上也是一样的。在这种情况下，数据将从 ChannelOutboundHandler 链的尾端开始流动，直到它到达链的头部为止。在这之后，出站数据将会到达网络传输层，这里显示为 Socket。通常情况下，这将触发一个写操作。

如果将两个类别的 ChannelHandler 都混合添加到同一个 ChannelPipeline 中会发生什么。虽然 ChannelInboundHandle 和 ChannelOutboundHandle 都扩展自 ChannelHandler，但是 Netty 能区分 ChannelInboundHandler 实现和 ChannelOutboundHandler 实现，并确保数据只会在具有相同定向类型的两个 ChannelHandler 之间传递。



ChannelPipeline 上的方法

addFirst、addBefore、addAfter、addLast

将一个 ChannelHandler 添加到 ChannelPipeline 中

remove 将一个 ChannelHandler 从 ChannelPipeline 中移除

replace 将 ChannelPipeline 中的一个 ChannelHandler 替换为另一个 ChannelHandler

get 通过类型或者名称返回 ChannelHandler

context 返回和 ChannelHandler 绑定的 ChannelHandlerContext

names 返回 ChannelPipeline 中所有 ChannelHandler 的名称

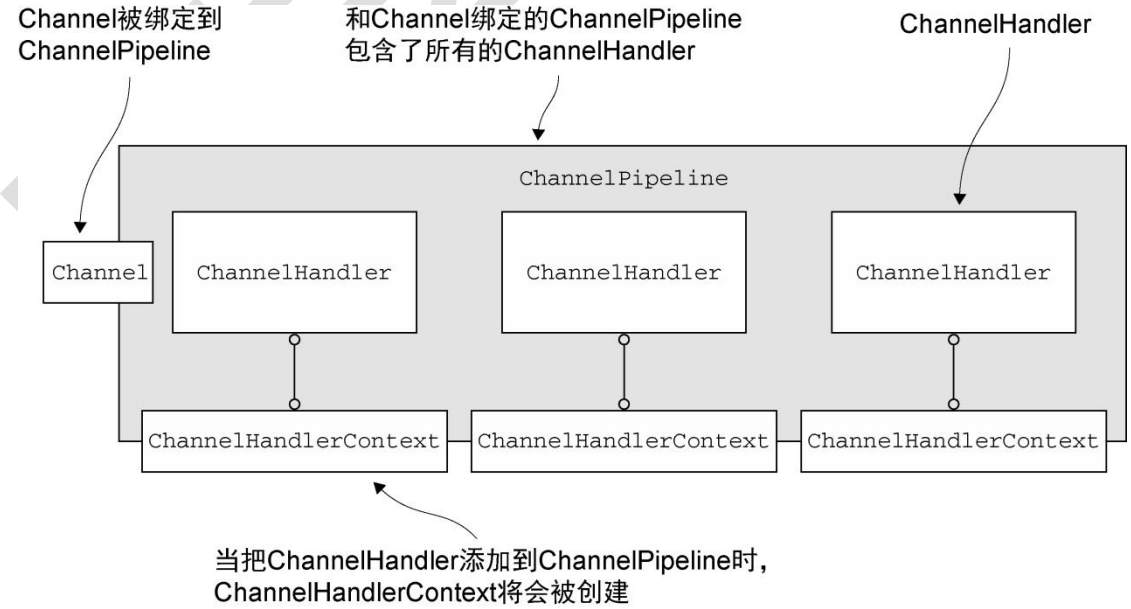
ChannelPipeline 的 API 公开了用于调用入站和出站操作的附加方法。

ChannelHandlerContext

通过使用作为参数传递到每个方法的 **ChannelHandlerContext**，事件可以被传递给当前 ChannelHandler 链中的下一个 ChannelHandler。虽然这个对象可以被用于获取底层的 Channel，但是它主要还是被用于写出站数据。

ChannelHandlerContext 代表了 ChannelHandler 和 ChannelPipeline 之间的关联，每当有 ChannelHandler 添加到 ChannelPipeline 中时，都会创建 ChannelHandler-Context。ChannelHandlerContext 的主要功能是管理它所关联的 ChannelHandler 和在同一个 ChannelPipeline 中的其他 ChannelHandler 之间的交互。

ChannelHandlerContext 有很多的方法，其中一些方法也存在于 Channel 和 Channel-Pipeline 本身上，**但是有一点重要的不同。**如果调用 Channel 或者 ChannelPipeline 上的这些方法，它们将沿着整个 ChannelPipeline 进行传播。而调用位于 ChannelHandlerContext 上的相同方法，则将从当前所关联的 ChannelHandler 开始，并且只会传播给位于该 ChannelPipeline 中的下一个（入站下一个，出站上一个）能够处理该事件的 ChannelHandler。



ChannelHandlerContext 的 API

alloc 返回和这个实例相关联的 Channel 所配置的 ByteBufAllocator

bind 绑定到给定的 SocketAddress，并返回 ChannelFuture

channel 返回绑定到这个实例的 Channel

close 关闭 Channel，并返回 ChannelFuture

connect 连接给定的 SocketAddress，并返回 ChannelFuture

deregister 从之前分配的 EventExecutor 注销，并返回 ChannelFuture

disconnect 从远程节点断开，并返回 ChannelFuture

executor 返回调度事件的 EventExecutor

fireChannelActive 触发对下一个 ChannelInboundHandler 上的 channelActive()方法（已连接）的调用

fireChannelInactive 触发对下一个 ChannelInboundHandler 上的 channelInactive()方法（已关闭）的调用

fireChannelRead 触发对下一个 ChannelInboundHandler 上的 channelRead()方法（已接收的消息）的调用

fireChannelReadComplete 触发对下一个 ChannelInboundHandler 上的 channelReadComplete()方法的调用

fireChannelRegistered 触发对下一个 ChannelInboundHandler 上的 fireChannelRegistered()方法的调用

fireChannelUnregistered 触发对下一个 ChannelInboundHandler 上的 fireChannelUnregistered()方法的调用

fireChannelWritabilityChanged 触发对下一个 ChannelInboundHandler 上的 fireChannelWritabilityChanged()方法的调用

fireExceptionCaught 触发对下一个 ChannelInboundHandler 上的 fireExceptionCaught(Throwable)方法的调用

fireUserEventTriggered 触发对下一个 ChannelInboundHandler 上的 fireUserEventTriggered(Object evt)方法的调用

handler 返回绑定到这个实例的 ChannelHandler

isRemoved 如果所关联的 ChannelHandler 已经被从 ChannelPipeline 中移除则返回 true

name 返回这个实例的唯一名称

pipeline 返回这个实例所关联的 ChannelPipeline

read 将数据从 Channel 读取到第一个入站缓冲区；如果读取成功则触发一个 channelRead 事件，并（在最后一个消息被读取完成后）通知 ChannelInboundHandler 的 channelReadComplete

(ChannelHandlerContext)方法

当使用 ChannelHandlerContext 的 API 的时候，有以下两点：

- ChannelHandlerContext 和 ChannelHandler 之间的关联（绑定）是永远不会改变的，所以缓存对它的引用是安全的；
- 如同我们在本节开头所解释的一样，相对于其他类的同名方法，ChannelHandlerContext 的方法将产生更短的事件流，应该尽可能地利用这个特性来获得最大的性能。

选择合适的内置通信传输模式

NIO io.netty.channel.socket.nio 使用 java.nio.channels 包作为基础——基于选择器的方式

Epoll io.netty.channel.epoll 由 JNI 驱动的 epoll()和非阻塞 IO。这个传输支持只有在 Linux 上可用的多种特性，如 SO_REUSEPORT，比 NIO 传输更快，而且是完全非阻塞的。将 NioEventLoopGroup 替换为 EpollEventLoopGroup ， 并且将 NioServerSocketChannel.class 替换为 EpollServerSocketChannel.class 即可。

OIO io.netty.channel.socket.oio 使用 java.net 包作为基础——使用阻塞流

Local io.netty.channel.local 可以在 VM 内部通过管道进行通信的本地传输

Embedded io.netty.channel.embedded Embedded 传输，允许使用 ChannelHandler 而又不需要一个真正的基于网络的传输。在测试 ChannelHandler 实现时非常有用

引导 Bootstrap

网络编程里，“服务器”和“客户端”实际上表示了不同的网络行为；换句话说，是监听传入的连接还是建立到一个或者多个进程的连接。

因此，有两种类型的引导：一种用于客户端（简单地称为 Bootstrap），而另一种（ServerBootstrap）用于服务器。无论你的应用程序使用哪种协议或者处理哪种类型的数据，唯一决定它使用哪种引导类的是它是作为一个客户端还是作为一个服务器。

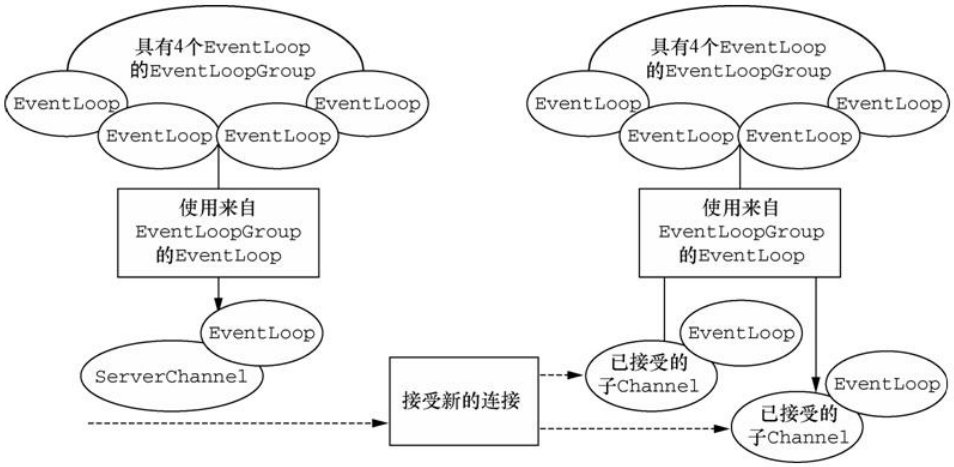
比较 **Bootstrap** 类

	Bootstrap	**ServerBootstrap**
网络编程中的作用	连接到远程主机和端口	绑定到一个本地端口
EventLoopGroup 的数目	1	2

ServerBootstrap 将绑定到一个端口，因为服务器必须要监听连接，而 Bootstrap 则是由想要连接到远程节点的客户端应用程序所使用的。

第二个区别可能更加明显。引导一个客户端只需要一个 EventLoopGroup，但是一个 ServerBootstrap 则需要两个（也可以是同一个实例）。

因为服务器需要两组不同的 Channel。第一组将只包含一个 ServerChannel，代表服务器自身的已绑定到某个本地端口的正在监听的套接字。而第二组将包含所有已创建的用来处理传入客户端连接（对于每个服务器已经接受的连接都有一个）的 Channel。



与 ServerChannel 相关联的 EventLoopGroup 将分配一个负责为传入连接请求创建 Channel 的 EventLoop。一旦连接被接受，第二个 EventLoopGroup 就会给它的 Channel 分配一个 EventLoop。

在引导过程中添加多个 ChannelHandler

Netty 提供了一个特殊的 ChannelInboundHandlerAdapter 子类：

public abstract class ChannelInitializer<C extends Channel> ext ends ChannelInboundHandlerAdapter

它定义了下面的方法：

protect ed abstract void initChannel(C ch) throws Exception;

这个方法提供了一种将多个 ChannelHandler 添加到一个 ChannelPipeline 中的简便方法。你只需要简单地向 Bootstrap 或 ServerBootstrap 的实例提供你的 ChannelInitializer 实现即可，并且一旦 Channel 被注册到了它的 EventLoop 之后，就会调用你的 initChannel()版本。在该方法返回之后，ChannelInitializer 的实例将会从 ChannelPipeline 中移除它自己。

ChannelOption

ChannelOption 的各种属性在套接字选项中都有对应。

1、ChannelOption.SO_BACKLOG

ChannelOption.SO_BACKLOG 对应的是 tcp/ip 协议 listen 函数中的 backlog 参数，函数 listen(int socketfd,int backlog)用来初始化服务端可连接队列，

服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接，多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，backlog 参数指定了队列的大小

2、ChannelOption.SO_REUSEADDR

ChanneOption.SO_REUSEADDR 对应于套接字选项中的 SO_REUSEADDR，这个参数表示允许重复使用本地地址和端口，

比如，某个服务器进程占用了 TCP 的 80 端口进行监听，此时再次监听该端口就会返回错误，使用该参数就可以解决问题，该参数允许共用该端口，这个在服务器程序中比较常使用，比如某个进程非正常退出，该程序占用的端口可能要被占用一段时间才能允许其他进程

使用，而且程序死掉以后，内核一需要一定的时间才能够释放此端口，不设置 SO_REUSEADDR 就无法正常使用该端口。

3、ChannelOption.SO_KEEPALIVE

Channeloption.SO_KEEPALIVE 参数对应于套接字选项中的 SO_KEEPALIVE，该参数用于设置 TCP 连接，当设置该选项以后，连接会测试链接的状态，这个选项用于可能长时间没有数据交流的连接。当设置该选项以后，如果在两小时内没有数据的通信时，TCP 会自动发送一个活动探测数据报文。

4、ChannelOption.SO_SNDBUF 和 ChannelOption.SO_RCVBUF

ChannelOption.SO_SNDBUF 参数对应于套接字选项中的 SO_SNDBUF，ChannelOption.SO_RCVBUF 参数对应于套接字选项中的 SO_RCVBUF 这两个参数用于操作接收缓冲区和发送缓冲区的大小，接收缓冲区用于保存网络协议站内收到的数据，直到应用程序读取成功，发送缓冲区用于保存发送数据，直到发送成功。

5、ChannelOption.SO_LINGER

ChannelOption.SO_LINGER 参数对应于套接字选项中的 SO_LINGER,Linux 内核默认的处理方式是当用户调用 close（）方法的时候，函数返回，在可能的情况下，尽量发送数据，不一定保证会发生剩余的数据，造成了数据的不确定性，使用 SO_LINGER 可以阻塞 close()的调用时间，直到数据完全发送

6、ChannelOption.TCP_NODELAY

ChannelOption.TCP_NODELAY 参数对应于套接字选项中的 TCP_NODELAY,该参数的使用与 Nagle 算法有关，Nagle 算法是将小的数据包组装为更大的帧然后进行发送，而不是输入一次发送一次,因此在数据包不足的时候会等待其他数据的到了，组装成大的数据包进行发送，虽然该方式有效提高网络的有效负载，但是却造成了延时，而该参数的作用就是禁止使用 Nagle 算法，使用于小数据即时传输，于 TCP_NODELAY 相对应的是 TCP_CORK，该选项是需要等到发送的数据量最大的时候，一次性发送数据，适用于文件传输。

ByteBuf

ByteBuf API 的优点：

它可以被用户自定义的缓冲区类型扩展；

通过内置的复合缓冲区类型实现了透明的零拷贝；

容量可以按需增长（类似于 JDK 的 StringBuilder）；

在读和写这两种模式之间切换不需要调用 ByteBuffer 的 flip()方法；

读和写使用了不同的索引；

支持方法的链式调用；

支持引用计数；

支持池化。

ByteBuf 维护了两个不同的索引，名称以 read 或者 write 开头的 ByteBuf 方法，将会推进其对应的索引，而名称以 set 或者 get 开头的操作则不会

如果打算读取字节直到 readerIndex 达到和 writerIndex 同样的值时会发生什么。在那时，你将会到达“可以读取的”数据的末尾。就如同试图读取超出数组末尾的数据一样，试图读取超出该点的数据将会触发一个 IndexOutOf-BoundsException。

可以指定 ByteBuf 的最大容量。试图移动写索引（即 writerIndex）超过这个值将会触发一个异常。（默认的限制是 Integer.MAX_VALUE。）

分配

堆缓冲区

最常用的 ByteBuf 模式是将数据存储在 JVM 的堆空间中。这种模式被称为支撑数组（backing array），它能在没有使用池化的情况下提供快速的分配和释放。可以由 hasArray() 来判断检查 ByteBuf 是否由数组支撑。如果不是，则这是一个直接缓冲区

直接缓冲区

直接缓冲区是另外一种 ByteBuf 模式。

直接缓冲区的主要缺点是，相对于基于堆的缓冲区，它们的分配和释放都较为昂贵。

ByteBufAllocator

Netty 通过 interface ByteBufAllocator 分配我们所描述过的任意类型的 ByteBuf 实例。

名称	描述
buffer()	返回一个基于堆或者直接内存存储的 ByteBuf
heapBuffer()	返回一个基于堆内存存储的 ByteBuf
directBuffer()	返回一个基于直接内存存储的 ByteBuf
compositeBuffer()	返回一个可以通过添加最大到指定数目的基于堆的或者直接内存存储的缓冲区来扩展的 CompositeByteBuf
ioBuffer()	返回一个用于套接字的 I/O 操作的 ByteBuf，当所运行的环境具有 sun.misc.Unsafe 支持时，返回基于直接内存存储的 ByteBuf，否则返回基于堆内存存储的 ByteBuf；当指定使用 PreferHeapByteBufAllocator 时，则只会返回基于堆内存存储的 ByteBuf。

可以通过 Channel（每个都可以有一个不同的 ByteBufAllocator 实例）或者绑定到 ChannelHandler 的 ChannelHandlerContext 获取一个到 ByteBufAllocator 的引用。

代码清单 5-14 获取一个到 ByteBufAllocator 的引用

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();
....
ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc();
...
```

从 Channel 获取一个到 ByteBufAllocator 的引用

从 ChannelHandlerContext 获取一个到 ByteBufAllocator 的引用

Netty 提供了两种 ByteBufAllocator 的实现：PooledByteBufAllocator 和 Unpooled-ByteBufAllocator。前者池化了 ByteBuf 的实例以提高性能并最大限度地减少内存碎片。后者的实现不池化 ByteBuf 实例，并且在每次它被调用时都会返回一个新的实例。

Netty4.1 默认使用了 PooledByteBufAllocator。

Unpooled 缓冲区

Netty 提供了一个简单的称为 Unpooled 的工具类，它提供了静态的辅助方法来创建未池化的 ByteBuf

实例。

buffer() 返回一个未池化的基于堆内存存储的 ByteBuf

directBuffer()返回一个未池化的基于直接内存存储的 ByteBuf

wrappedBuffer() 返回一个包装了给定数据的 ByteBuf

copiedBuffer() 返回一个复制了给定数据的 ByteBuf

Unpooled 类还可用于 ByteBuf 同样可用于那些并不需要 Netty 的其他组件的非网络项目。

随机访问索引/顺序访问索引/读写操作

如同在普通的 Java 字节数组中一样，ByteBuf 的索引是从零开始的：第一个字节的索引是 0，最后一个字节的索引总是 capacity() - 1。使用那些需要一个索引值参数(**随机访问**,也即是数组下标)的方法（的其中）之一来访问数据既不会改变 readerIndex 也不会改变 writerIndex。如果有需要，也可以通过调用 readerIndex(index)或者 writerIndex(index)来手动移动这两者。**顺序访问**通过索引访问

有两种类别的读/写操作：

get()和 set()操作，从给定的索引开始，并且保持索引不变；get+数据字长（bool.byte,int,short,long,bytes）

read()和 write()操作，从给定的索引开始，并且会根据已经访问过的字节数对索引进行调整。

更多的操作

isReadable() 如果至少有一个字节可供读取，则返回 true

isWritable() 如果至少有一个字节可被写入，则返回 true

readableBytes() 返回可被读取的字节数

writableBytes() 返回可被写入的字节数

capacity() 返回 ByteBuf 可容纳的字节数。在此之后，它会尝试再次扩展直到达到 maxCapacity()

maxCapacity() 返回 ByteBuf 可以容纳的最大字节数

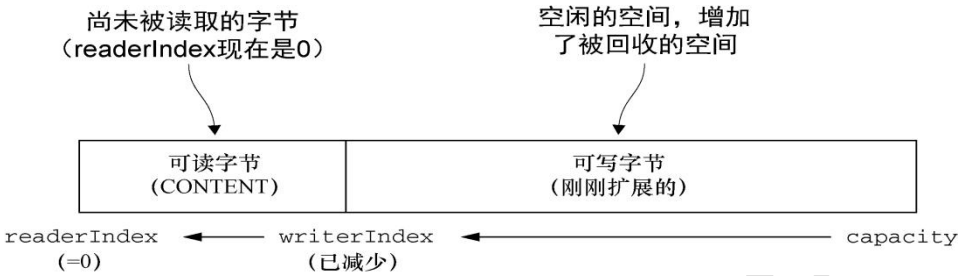
hasArray() 如果 ByteBuf 由一个字节数组支撑，则返回 true

array() 如果 ByteBuf 由一个字节数组支撑则返回该数组；否则，它将抛出一个 UnsupportedOperationException 异常

可丢弃字节

为可丢弃字节的分段包含了已经被读过的字节。通过调用 discardRead-Bytes()方法，可以丢弃它们并回收空间。这个分段的初始大小为 0，存储在 readerIndex 中，会随着 read 操作的执行而增加（get*操作不会移动 readerIndex）。

缓冲区上调用 discardReadBytes()方法后，可丢弃字节分段中的空间已经变为可写的了。频繁地调用 discardReadBytes()方法以确保可写分段的最大化，但是请注意，这将极有可能会导致内存复制，因为可读字节必须被移动到缓冲区的开始位置。建议只在有真正需要的时候才这样做，例如，当内存非常宝贵的时候。

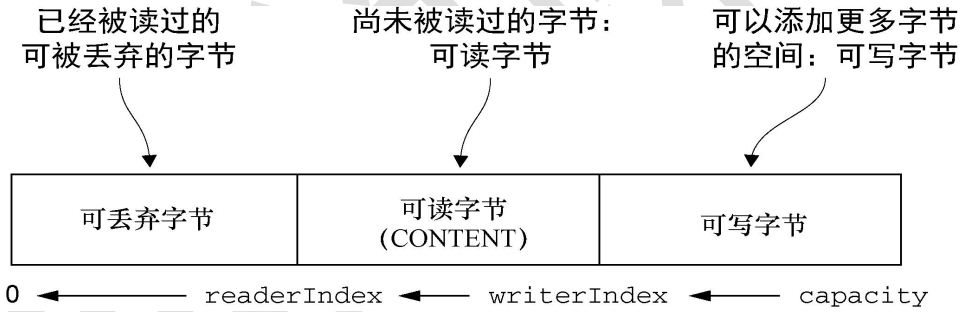


可读字节

ByteBuf 的可读字节分段存储了实际数据。新分配的、包装的或者复制的缓冲区的默认的 readerIndex 值为 0。

可写字节

可写字节分段是指一个拥有未定义内容的、写入就绪的内存区域。新分配的缓冲区的 writerIndex 的默认值为 0。任何名称以 write 开头的操作都将从当前的 writerIndex 处开始写数据，并将它增加已经写入的字节数。



索引管理

调用 markReaderIndex()、markWriterIndex()、resetWriterIndex()和 resetReaderIndex()来标记和重置 ByteBuf 的 readerIndex 和 writerIndex。

也可以通过调用 readerIndex(int)或者 writerIndex(int)来将索引移动到指定位置。试图将任何一个索引设置到一个无效的位置都将导致一个 IndexOutOfBoundsException。

可以通过调用 clear()方法来将 readerIndex 和 writerIndex 都设置为 0。注意，这并不会清除内存中的内容。

查找操作

在 ByteBuf 中有多种可以用来确定指定值的索引的方法。最简单的是使用 indexOf()方法。

较复杂的查找可以通过调用 forEach Byte()。

代码展示了一个查找回车符（\r）的例子。

```
ByteBuf buffer = ...;
int index = buffer.forEachByte(ByteBufProcessor.FIND_CR);
```

派生缓冲区

派生缓冲区为 ByteBuf 提供了以专门的方式来呈现其内容的视图。这类视图是通过以下方法被创建的：

duplicate()；

slice()；

slice(int, int)；

Unpooled.unmodifiableBuffer(…)；

order(ByteOrder)；

readSlice(int)。

每个这些方法都将返回一个新的 ByteBuf 实例，它具有自己的读索引、写索引和标记索引。其内部存储和 JDK 的 ByteBuffer 一样也是共享的。

ByteBuf 复制 如果需要一个现有缓冲区的真实副本，请使用 copy()或者 copy(int, int)方法。不同于派生缓冲区，由这个调用所返回的 ByteBuf 拥有独立的数据副本。

引用计数

引用计数是一种通过在某个对象所持有的资源不再被其他对象引用时释放该对象所持有的资源来优化内存使用和性能的技术。Netty 在第 4 版中为 ByteBuf 引入了引用计数技术，interface ReferenceCounted。

工具类

ByteBufUtil 提供了用于操作 ByteBuf 的静态的辅助方法。因为这个 API 是通用的，并且和池化无关，所以这些方法已然在分配类的外部实现。

这些静态方法中最有价值的可能就是 hexdump()方法，它以十六进制的表示形式打印 ByteBuf 的内容。这在各种情况下都很有用，例如，出于调试的目的记录 ByteBuf 的内容。十六进制的表示通常会提供一个比字节值的直接表示形式更加有用的日志条目，此外，十六进制的版本还可以很容易地转换回实际的字节表示。

另一个有用的方法是 boolean equals(ByteBuf, ByteBuf)，它被用来判断两个 ByteBuf 实例的相等性。

资源释放

当某个 ChannelInboundHandler 的实现重写 channelRead()方法时，它要负责显式地释放与池化的 ByteBuf 实例相关的内存。Netty 为此提供了一个实用方法 ReferenceCountUtil.release()

Netty 将使用 WARN 级别的日志消息记录未释放的资源，使得可以非常简单地在代码中发现违规的实例。但是以这种方式管理资源可能很繁琐。一个更加简单的方式是使用 SimpleChannelInboundHandler，SimpleChannelInboundHandler 会自动释放资源。

1、对于入站请求，Netty 的 EventLoo 在处理 Channel 的读操作时进行分配 ByteBuf，对于这类 ByteBuf，需要我们自行进行释放，有三种方式，或者使用 SimpleChannelInboundHandler，或者在重写 channelRead()方法使用 ReferenceCountUtil.release()或者使用 ctx.fireChannelRead 继续向后传递；

2、对于出站请求，不管 ByteBuf 是否由我们的业务创建的，当调用了 write 或者 writeAndFlush 方法后，Netty 会自动替我们释放，不需要我们业务代码自行释放。

解决粘包/半包问题

回顾我们的的 Hello,Netty

参见 cn.enjoyedu.nettybasic.splicing.demo 下的代码

什么是 TCP 粘包半包？

假设客户端分别发送了两个数据包 D1 和 D2 给服务端，由于服务端一次读取到的字节数是不确定的，故可能存在以下 4 种情况。

（1）服务端分两次读取到了两个独立的数据包，分别是 D1 和 D2，没有粘包和拆包；

（2）服务端一次接收到了两个数据包，D1 和 D2 粘合在一起，被称为 TCP 粘包；

（3）服务端分两次读取到了两个数据包，第一次读取到了完整的 D1 包和 D2 包的部分内容，第二次读取到了 D2 包的剩余内容，这被称为 TCP 拆包；

（4）服务端分两次读取到了两个数据包，第一次读取到了 D1 包的部分内容 D1_1，第二次读取到了 D1 包的剩余内容 D1_2 和 D2 包的整包。

如果此时服务端 TCP 接收滑窗非常小，而数据包 D1 和 D2 比较大，很有可能会发生第五种可能，即服务端分多次才能将 D1 和 D2 包接收完全，期间发生多次拆包。

TCP 粘包/半包发生的原因

由于 TCP 协议本身的机制（面向连接的可靠地协议-三次握手机制）客户端与服务器会维持一个连接（Channel），数据在连接不断开的情况下，可以持续不断地将多个数据包发往服务器，但是如果发送的网络数据包太小，那么他本身会启用 Nagle 算法（可配置是否启用）对较小的数据包进行合并（基于此，TCP 的网络延迟要 UDP 的高些）然后再发送（超

时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就无法区分哪些数据包是客户端自己分开发送的，这样产生了粘包；服务器在接收到数据库后，放到缓冲区中，如果消息没有被及时从缓存区取走，下次在取数据的时候可能就会出现一次取出多个数据包的情况，造成粘包现象

UDP：本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有 Nagle 算法之说了），他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP 头+IP 头等等发一次数据封装一次）也就没有粘包一说了。

分包产生的原因就简单的多：可能是 IP 分片传输导致的，也可能是传输过程中丢失部分包导致出现的半包，还有可能就是一个包可能被分成了两次传输，在取数据的时候，先取到了一部分（还可能与接收的缓冲区大小有关系），总之就是一个数据包被分成了多次接收。

更具体的原因有三个，分别如下。

1. 应用程序写入数据的字节大小大于套接字发送缓冲区的大小

2. 进行 MSS 大小的 TCP 分段。MSS 是最大报文段长度的缩写。MSS 是 TCP 报文段中的数据字段的最大长度。数据字段加上 TCP 首部才等于整个的 TCP 报文段。所以 MSS 并不是 TCP 报文段的最大长度，而是：MSS=TCP 报文段长度-TCP 首部长度

3. 以太网的 payload 大于 MTU 进行 IP 分片。MTU 指：一种通信协议的某一层上面所能通过的最大数据包大小。如果 IP 层有一个数据包要传，而且数据的长度比链路层的 MTU 大，那么 IP 层就会进行分片，把数据包分成托干片，让每一片都不超过 MTU。注意，IP 分片可以发生在原始发送端主机上，也可以发生在中间路由器上。

解决粘包半包问题

由于底层的 TCP 无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决，根据业界的主流协议的解决方案，可以归纳如下。

（1） 在包尾增加分割符，比如回车换行符进行分割，例如 FTP 协议；

参见 cn.enjoyedu.nettybasic.splicing.linebase 和 cn.enjoyedu.nettybasic.splicing.delimiter 下的代码

（2）消息定长，例如每个报文的大小为固定长度 200 字节，如果不够，空位补空格；

参见 cn.enjoyedu.nettybasic.splicing.fixed 下的代码

（3）将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路为消息头的第一个字段使用 int32 来表示消息的总长度，LengthFieldBasedFrameDecoder。

编解码器框架

什么是编解码器

每个网络应用程序都必须定义如何解析在两个节点之间来回传输的原始字节，以及如何将其和目标应用程序的数据格式做相互转换。这种转换逻辑由编解码器处理，编解码器由编

码器和解码器组成，它们每种都可以将字节流从一种格式转换为另一种格式。那么它们的区别是什么呢？

如果将消息看作是对于特定的应用程序具有具体含义的结构化的字节序列—它的数据。那么编码器是将消息转换为适合于传输的格式（最有可能的就是字节流）；而对应的解码器则是将网络字节流转换回应用程序的消息格式。因此，编码器操作出站数据，而解码器处理入站数据。我们前面所学的解决粘包半包的其实也是编解码器框架的一部分。

解码器

将字节解码为消息——ByteToMessageDecoder

将一种消息类型解码为另一种——MessageToMessageDecoder。

因为解码器是负责将入站数据从一种格式转换到另一种格式的，所以 Netty 的解码器实现了 ChannelInboundHandler。

什么时候会用到解码器呢？很简单：每当需要为 ChannelPipeline 中的下一个 Channel-InboundHandler 转换入站数据时会用到。此外，得益于 ChannelPipeline 的设计，可以将多个解码器链接在一起，以实现任意复杂的转换逻辑。

将字节解码为消息

抽象类 ByteToMessageDecoder

将字节解码为消息（或者另一个字节序列）是一项如此常见的任务，以至于 Netty 为它提供了一个抽象的基类：ByteToMessageDecoder。由于你不可能知道远程节点是否会一次性地发送一个完整的消息，所以这个类会对入站数据进行缓冲，直到它准备好处理。

它最重要方法

decode(ChannelHandlerContext ctx,ByteBuf in,List<Object> out)

这是你必须实现的唯一抽象方法。decode()方法被调用时将会传入一个包含了传入数据的 ByteBuf，以及一个用来添加解码消息的 List。对这个方法的调用将会重复进行，直到确定没有新的元素被添加到该 List，或者该 ByteBuf 中没有更多可读取的字节时为止。然后，如果该 List 不为空，那么它的内容将会被传递给 ChannelPipeline 中的下一个 ChannelInboundHandler。

将一种消息类型解码为另一种

在两个消息格式之间进行转换（例如，从 String->Integer）

decode(ChannelHandlerContext ctx,I msg,List<Object> out)

对于每个需要被解码为另一种格式的入站消息来说，该方法都将会被调用。解码消息随后会被传递给 ChannelPipeline 中的下一个 ChannelInboundHandler

MessageToMessageDecoder<T>，T 代表源数据的类型

TooLongFrameException

由于 Netty 是一个异步框架，所以需要在字节可以解码之前在内存中缓冲它们。因此，不能让解码器缓冲大量的数据以至于耗尽可用的内存。为了解除这个常见的顾虑，Netty 提供了 TooLongFrameException 类，其将由解码器在帧超出指定的大小限制时抛出。

为了避免这种情况，你可以设置一个最大字节数的阈值，如果超出该阈值，则会导致抛出一个 TooLongFrameException（随后会被 ChannelHandler.exceptionCaught()方法捕获）。然后，如何处理该异常则完全取决于该解码器的用户。某些协议（如 HTTP）可能允许你返回一个特殊的响应。而在其他的情况下，唯一的选择可能就是关闭对应的连接。

```
public class SafeByteToMessageDecoder extends ByteToMessageDecoder {
    private static final int MAX_FRAME_SIZE = 1024;
    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
            int readable = in.readableBytes();
            if (readable > MAX_FRAME_SIZE) {
                in.skipBytes(readable);
                throw new TooLongFrameException("Frame too big!");
            }
            // do something
            ...
    }
}
```

检查缓冲区中是否有超过 MAX_FRAME_SIZE 个字节

跳过所有的可读字节，抛出 TooLongFrame-Exception 并通知 ChannelHandler

编码器

解码器的功能正好相反。Netty 提供了一组类，用于帮助你编写具有以下功能的编码器：

将消息编码为字节；MessageToByteEncoder

将消息编码为消息：MessageToMessageEncoder<T>，T 代表源数据的类型

将消息编码为字节

encode(ChannelHandlerContext ctx,I msg,ByteBuf out)

encode()方法是你需要实现的唯一抽象方法。它被调用时将会传入要被该类编码为 ByteBuf 的（类型为 I 的）出站消息。该 ByteBuf 随后将会被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler

将消息编码为消息

encode(ChannelHandlerContext ctx,I msg,List<Object> out)

这是你需要实现的唯一方法。每个通过 write()方法写入的消息都将会被传递给 encode()方法，以编码为一个或者多个出站消息。随后，这些出站消息将会被转发给 ChannelPipeline 中的下一个 ChannelOutboundHandler

编解码器类

我们一直将解码器和编码器作为单独的实体讨论，但是你有时将会发现在同一个类中管理入站和出站数据和消息的转换是很有用的。Netty 的抽象编解码器类正好用于这个目的，因为它们每个都将捆绑一个解码器/编码器对，以处理我们一直在学习的这两种类型的操作。这些类同时实现了 ChannelInboundHandler 和 ChannelOutboundHandler 接口。

为什么我们并没有一直优先于单独的解码器和编码器使用这些复合类呢？因为通过尽可能地将这两种功能分开，最大化了代码的可重用性和可扩展性，这是 Netty 设计的一个基本原则。

相关的类：

抽象类 ByteToMessageCodec

抽象类 MessageToMessageCodec

Netty 内置的编解码器和 ChannelHandler

Netty 为许多通用协议提供了编解码器和处理器，几乎可以开箱即用，这减少了你在那些相当繁琐的事务上本来会花费的时间与精力。

通过 SSL/TLS 保护 Netty 应用程序

SSL 和 TLS 这样的安全协议，它们层叠在其他协议之上，用以实现数据安全。我们在访问安全网站时遇到过这些协议，但是它们也可用于其他不是基于 HTTP 的应用程序，如安全 SMTP（SMTPS）邮件服务器甚至是关系型数据库系统。

为了支持 SSL/TLS，Java 提供了 javax.net.ssl 包，它的 SSLContext 和 SSLEngine 类使得实现解密和加密相当简单直接。Netty 通过一个名为 SslHandler 的 ChannelHandler 实现利用了这个 API，其中 SslHandler 在内部使用 SSLEngine 来完成实际的工作。

Netty 还提供了使用 OpenSSL 工具包（www.openssl.org）的 SSLEngine 实现。这个 OpenSsl-Engine 类提供了比 JDK 提供的 SSLEngine 实现更好的性能。

如果 OpenSSL 库可用，可以将 Netty 应用程序（客户端和服务器）配置为默认使用 OpenSslEngine。如果不可用，Netty 将会回退到 JDK 实现。

在大多数情况下，SslHandler 将是 ChannelPipeline 中的第一个 ChannelHandler。

HTTP 系列

HTTP 是基于请求/响应模式的：客户端向服务器发送一个 HTTP 请求，然后服务器将会返回一个 HTTP 响应。Netty 提供了多种编码器和解码器以简化对这个协议的使用。

一个 HTTP 请求/响应可能由多个数据部分组成，并且它总是以一个 LastHttpContent 部分作为结束。FullHttpRequest 和 FullHttpResponse 消息是特殊的子类型，分别代表了完整的请求和响应。所有类型的 HTTP 消息（FullHttpRequest、LastHttpContent 等等）都实现了 HttpObject 接口。

HttpRequestEncoder 将 HttpRequest、HttpContent 和 LastHttpContent 消息编码为字节

HttpResponseEncoder 将 HttpResponse、HttpContent 和 LastHttpContent 消息编码为字节

HttpRequestDecoder 将字节解码为 HttpRequest、HttpContent 和 LastHttpContent 消息

HttpResponseDecoder 将字节解码为 HttpResponse、HttpContent 和 LastHttpContent 消息

```
@Override
protected void initChannel(Channel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    if (client) {
        pipeline.addLast("decoder", new HttpResponseDecoder());
        pipeline.addLast("encoder", new HttpRequestEncoder());
    } else {
        pipeline.addLast("decoder", new HttpRequestDecoder());
        pipeline.addLast("encoder", new HttpResponseEncoder());
    }
}
```

如果是客户端，则添加 HttpResponseDecoder 以处理来自服务器的响应

如果是服务器，则添加 HttpResponseEncoder 以向客户端发送响应

如果是服务器，则添加 HttpRequestDecoder 以接收来自客户端的请求

如果是客户端，则添加 HttpRequestEncoder 以向服务器发送请求

聚合 HTTP 消息

由于 HTTP 的请求和响应可能由许多部分组成，因此你需要聚合它们以形成完整的消息。为了消除这项繁琐的任务，Netty 提供了一个聚合器，它可以将多个消息部分合并为 FullHttpRequest 或者 FullHttpResponse 消息。通过这样的方式，你将总是看到完整的消息内容。

```
@Override
protected void initChannel(Channel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    if (isClient) {
        pipeline.addLast("codec", new HttpClientCodec());
    } else {
        pipeline.addLast("codec", new HttpServerCodec());
    }
    pipeline.addLast("aggregator",
        new HttpObjectAggregator(512 * 1024));
}
```

如果是客户端，则添加 HttpClientCodec

如果是服务器，则添加 HttpServerCodec

将最大的消息大小为 512 KB 的 HttpObjectAggregator 添加到 ChannelPipeline

HTTP 压缩

当使用 HTTP 时，建议开启压缩功能以尽可能多地减小传输数据的大小。虽然压缩会带来一些 CPU 时钟周期上的开销，但是通常来说它都是一个好主意，特别是对于文本数据来说。Netty 为压缩和解压缩提供了 ChannelHandler 实现，它们同时支持 gzip 和 deflate 编码。

```
@Override
protected void initChannel(Channel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    if (isClient) {
        pipeline.addLast("codec", new HttpClientCodec());
        pipeline.addLast("decompressor",
            new HttpContentDecompressor());
    } else {
        pipeline.addLast("codec", new HttpServerCodec());
        pipeline.addLast("compressor",
        new HttpContentCompressor());
    }
}
```

如果是客户端，则添加 HttpClientCodec

如果是客户端，则添加 HttpContentDecompressor 以处理来自服务器的压缩内容

如果是服务器，则添加 HttpServerCodec

如果是服务器，则添加 HttpContentCompressor 来压缩数据（如果客户端支持它）

使用 HTTPS

启用 HTTPS 只需要将 SslHandler 添加到 ChannelPipeline 的 ChannelHandler 组合中。

SSL 和 HTTP 的代码参见模块 netty-http

WebSocket

后面的课会细讲，这里略过。

空闲的连接和超时

检测空闲连接以及超时对于及时释放资源来说是至关重要的。由于这是一项常见的任务，Netty 特地为它提供了几个 ChannelHandler 实现。

IdleStateHandler 当连接空闲时间太长时，将会触发一个 IdleStateEvent 事件。然后，你可以通过在你的 ChannelInboundHandler 中重写 userEventTriggered()方法来处理该 IdleStateEvent 事件。

ReadTimeoutHandler 如果在指定的时间间隔内没有收到任何的入站数据，则抛出一个 Read-TimeoutException 并关闭对应的 Channel。可以通过重写你的 ChannelHandler 中的 exceptionCaught()方法来检测该 Read-TimeoutException。

WriteTimeoutHandler 如果在指定的时间间隔内没有任何出站数据写入，则抛出一个 Write-TimeoutException 并关闭对应的 Channel 。可以通过重写你的 ChannelHandler 的 exceptionCaught()方法检测该 WriteTimeout-Exception。

代码参见模块 netty-basic 下的包 cn.enjoyedu.nettybasic.idlestate，更具体的使用会在 netty 实战项目中看到。

序列化问题

Java 序列化的目的主要有两个：

1.网络传输

2.对象持久化

当选行远程跨迸程服务调用时，需要把被传输的 Java 对象编码为字节数组或者 ByteBuffer 对象。而当远程服务读取到 ByteBuffer 对象或者字节数组时，需要将其解码为发送时的 Java 对象。这被称为 Java 对象编解码技术。

Java 序列化仅仅是 Java 编解码技术的一种，由于它的种种缺陷，衍生出了多种编解码技术和框架

Java 序列化的缺点

Java 序列化从 JDK1.1 版本就已经提供，它不需要添加额外的类库，只需实现 java.io.Serializable 并生成序列 ID 即可，因此，它从诞生之初就得到了广泛的应用。

但是在远程服务调用（RPC）时，很少直接使用 Java 序列化进行消息的编解码和传输，这又是什么原因呢？下面通过分析.Tava 序列化的缺点来找出答案。

1 无法跨语言

对于跨进程的服务调用，服务提供者可能会使用 C 十＋或者其他语言开发，当我们需要和异构语言进程交互时 Java 序列化就难以胜任。由于 Java 序列化技术是 Java 语言内部的私

有协议，其他语言并不支持，对于用户来说它完全是黑盒。对于 Java 序列化后的字节数组，别的语言无法进行反序列化，这就严重阻碍了它的应用。

2 序列化后的码流太大

通过一个实例看下 Java 序列化后的字节数组大小。

3 序列化性能太低

无论是序列化后的码流大小，还是序列化的性能，JDK 默认的序列化机制表现得都很差。因此，我们边常不会选择 Java 序列化作为远程跨节点调用的编解码框架。

代码参见模块 netty-basic 下的包 cn.enjoyedu.nettybasic.serializable.protogenesis

序列化 – 内置和第三方的 MessagePack 实战

内置

Netty 内置了对 JBoss Marshalling 和 Protocol Buffers 的支持

Protocol Buffers 序列化机制代码参见模块 netty-basic 下的包 cn.enjoyedu.nettybasic.serializable.protobuf

集成第三方 MessagePack 实战（LengthFieldBasedFrame 详解）

LengthFieldBasedFrame 详解

maxFrameLength：表示的是包的最大长度，

lengthFieldOffset：指的是长度域的偏移量，表示跳过指定个数字节之后的才是长度域；

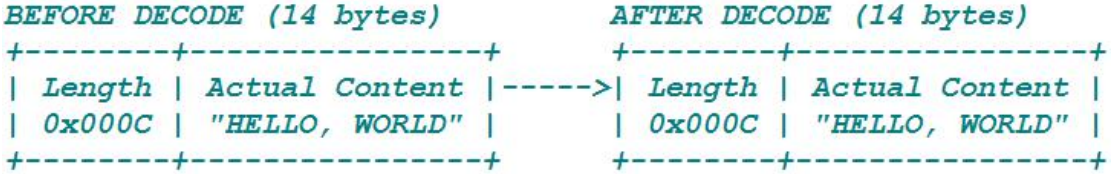
lengthFieldLength：记录该帧数据长度的字段，也就是长度域本身的长度；

lengthAdjustment：长度的一个修正值，可正可负；

initialBytesToStrip：从数据帧中跳过的字节数，表示得到一个完整的数据包之后，忽略多少字节，开始读取实际我要的数据

failFast：如果为 true，则表示读取到长度域，TA 的值的超过 maxFrameLength，就抛出一个 TooLongFrameException，而为 false 表示只有当真正读取完长度域的值表示的字节之后，才会抛出 TooLongFrameException，默认情况下设置为 true，建议不要修改，否则可能会造成内存溢出。

数据包大小: 14B = 长度域 2B + "HELLO, WORLD"（单词 HELLO+一个逗号+一个空格+单词 WORLD）



长度域的值为 12B(0x000c)。希望解码后保持一样，根据上面的公式,参数应该为：

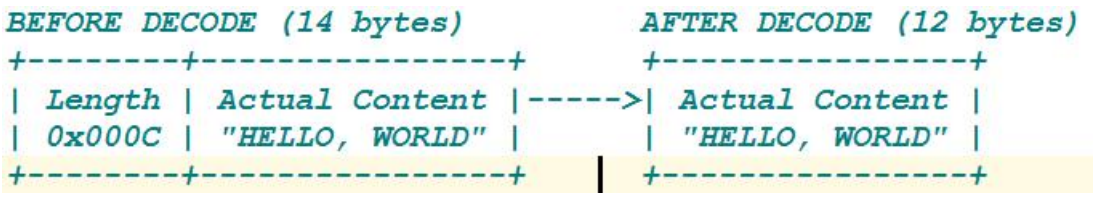
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment　无需调整

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

数据包大小: 14B = 长度域 2B + "HELLO, WORLD"



长度域的值为 12B(0x000c)。解码后，希望丢弃长度域 2B 字段，所以，只要 initialBytesToStrip = 2 即可。

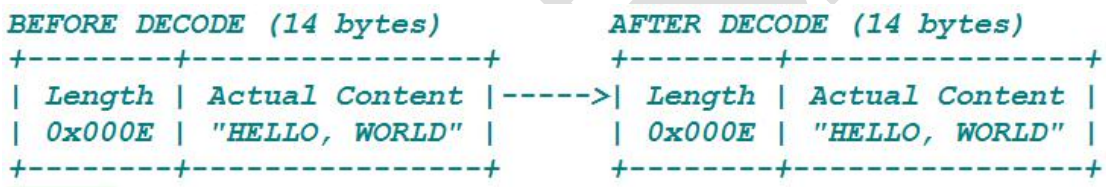
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment　无需调整

4. initialBytesToStrip = 2 解码过程中，丢弃 2 个字节的数据

数据包大小: 14B = 长度域 2B + "HELLO, WORLD"。长度域的值为 14(0x000E)



长度域的值为 14(0x000E)，包含了长度域本身的长度。希望解码后保持一样，根据上面的公式，参数应该为：

1. lengthFieldOffset = 0

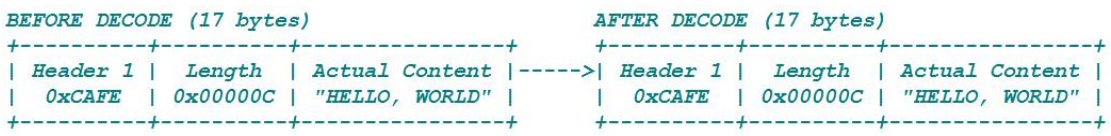
2. lengthFieldLength = 2

3. lengthAdjustment = -2　因为长度域为 14，而报文内容为 12，为了防止读取报文超出报文本体，和将长度字段一起读取进来，需要告诉 netty，实际读取的报文长度比长度域中的要少 2（12-14=-2）

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

在长度域前添加 2 个字节的 Header。长度域的值(0x00000C) = 12。总数据包长度:

17=Header(2B) + 长度域(3B) + "HELLO, WORLD"



长度域的值为 12B(0x000c)。编码解码后，长度保持一致，所以 initialBytesToStrip = 0。参数应该为:

1. lengthFieldOffset = 2

2. lengthFieldLength = 3

3. lengthAdjustment = 0 无需调整

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

Header 与长度域的位置换了。总数据包长度: 17=长度域(3B) + Header(2B) + "HELLO, WORLD"

```
BEFORE DECODE (17 bytes)                       AFTER DECODE (17 bytes)
+----------+----------+----------------+      +----------+----------+----------------+
|  Length  | Header 1 | Actual Content |----->|  Length  | Header 1 | Actual Content |
| 0x00000C |  0xCAFE  | "HELLO, WORLD" |      | 0x00000C |  0xCAFE  | "HELLO, WORLD" |
+----------+----------+----------------+      +----------+----------+----------------+
```

长度域的值为 12B(0x000c)。编码解码后，长度保持一致，所以 initialBytesToStrip = 0。参数应该为:

1. lengthFieldOffset = 0

2. lengthFieldLength = 3

3. lengthAdjustment = 2 因为长度域为 12，而报文内容为 12，但是我们需要把 Header 的值一起读取进来，需要告诉 netty，实际读取的报文内容长度比长度域中的要多 2（12+2=14）

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

带有两个 header。HDR1 丢弃，长度域丢弃，只剩下第二个 header 和有效包体，这种协议中，一般 HDR1 可以表示 magicNumber，表示应用只接受以该 magicNumber 开头的二进制数据，rpc 里面用的比较多。总数据包长度: 16=HDR1(1B)+长度域(2B) +HDR2(1B) + "HELLO, WORLD"

```
BEFORE DECODE (16 bytes)                              AFTER DECODE (13 bytes)
+------+--------+------+----------------+             +------+----------------+
| HDR1 | Length | HDR2 | Actual Content |----->| HDR2 | Actual Content |
| 0xCA | 0x000C | 0xFE | "HELLO, WORLD" |             | 0xFE | "HELLO, WORLD" |
+------+--------+------+----------------+             +------+----------------+
```

长度域的值为 12B(0x000c)

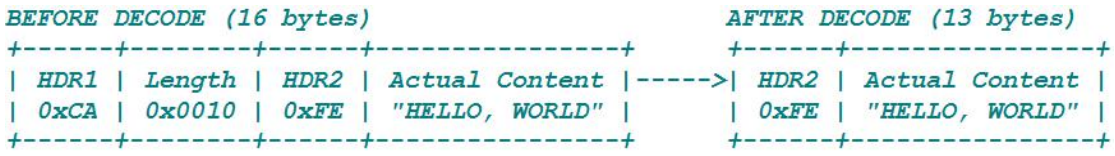
1. lengthFieldOffset = 1 (HDR1 的长度)

2. lengthFieldLength = 2

3. lengthAdjustment =1 因为长度域为 12，而报文内容为 12，但是我们需要把 HDR2 的值一起读取进来，需要告诉 netty，实际读取的报文内容长度比长度域中的要多 1（12+1=13）

4. initialBytesToStrip = 3 丢弃了 HDR1 和长度字段

带有两个 header，HDR1 丢弃，长度域丢弃，只剩下第二个 header 和有效包体。总数据包长度: 16=HDR1(1B)+长度域(2B) +HDR2(1B) + "HELLO, WORLD"



长度域的值为 16B(0x0010)，长度为 2，HDR1 的长度为 1，HDR2 的长度为 1，包体的长度为 12，1+1+2+12=16。

1. lengthFieldOffset = 1

2. lengthFieldLength = 2

3. lengthAdjustment = -3 因为长度域为 16，需要告诉 netty，实际读取的报文内容长度比长度域中的要 少 3（13-16= -3）

4. initialBytesToStrip = 3 丢弃了 HDR1 和长度字段

MessagePack 集成

代码参见模块 netty-basic 下的：cn.enjoyedu.ch02.serializable.msgpack 包下

如何进行单元测试

一种特殊的 Channel 实现——EmbeddedChannel，它是 Netty 专门为改进针对 ChannelHandler 的单元测试而提供的。

将入站数据或者出站数据写入到 EmbeddedChannel 中，然后检查是否有任何东西到达了 ChannelPipeline 的尾端。以这种方式，你便可以确定消息是否已经被编码或者被解码过了，以及是否触发了任何的 ChannelHandler 动作。

writeInbound(Object... msgs)

将入站消息写到 EmbeddedChannel 中。**如果**可以通过 readInbound()方法从 EmbeddedChannel 中读取数据，则返回 true

readInbound()

从 EmbeddedChannel 中读取一个入站消息。任何返回的东西都穿越了整个 ChannelPipeline。**如果**没有任何可供读取的，则返回 null

writeOutbound(Object... msgs)

将出站消息写到 EmbeddedChannel 中。**如果**现在可以通过 readOutbound()方法从 EmbeddedChannel 中读取到什么东西，则返回 true

readOutbound()

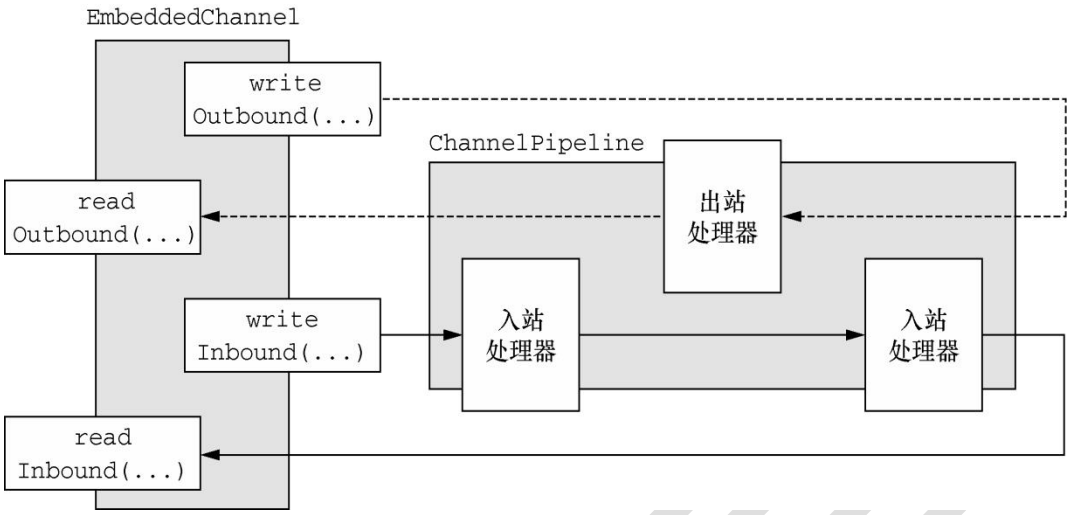
从 EmbeddedChannel 中读取一个出站消息。任何返回的东西都穿越了整个 ChannelPipeline。如果没有任何可供读取的，则返回 null

finish() 将 EmbeddedChannel 标记为完成，并且如果有可被读取的入站数据或者出站数据，则返回 true。这个方法还将会调用 EmbeddedChannel 上的 close()方法。

入站数据由 ChannelInboundHandler 处理，代表从远程节点读取的数据。出站数据由 ChannelOutboundHandler 处理，代表将要写到远程节点的数据。

使用 writeOutbound()方法将消息写到 Channel 中，并通过 ChannelPipeline 沿着出站的方向传递。随后，你可以使用 readOutbound()方法来读取已被处理过的消息，以确定结果是否和预期一样。 类似地，对于入站数据，你需要使用 writeInbound()和 readInbound()方法。

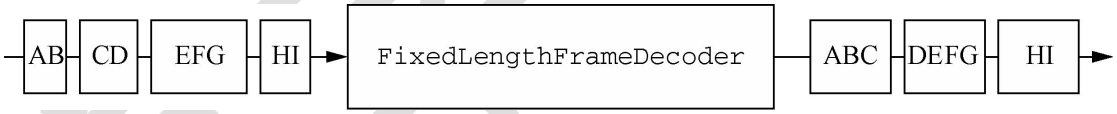
在每种情况下，消息都将会传递过 ChannelPipeline，并且被相关的 ChannelInboundHandler 或者 ChannelOutboundHandler 处理。



测试入站消息

我们有一个简单的 ByteToMessageDecoder 实现。给定足够的数据，这个实现将产生固定大小的帧。如果没有足够的数据可供读取，它将等待下一个数据块的到来，并将再次检查是否能够产生一个新的帧。

这个特定的解码器将产生固定为 3 字节大小的帧。因此，它可能会需要多个事件来提供足够的字节数以产生一个帧。



测试出站消息

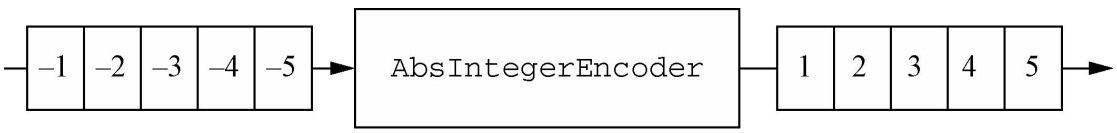
在测试的处理器—AbsIntegerEncoder，它是 Netty 的 MessageToMessageEncoder 的一个特殊化的实现，用于将负值整数转换为绝对值。

该示例将会按照下列方式工作：

持有 AbsIntegerEncoder 的 EmbeddedChannel 将会以 4 字节的负整数的形式写出站数据；

编码器将从传入的 ByteBuf 中读取每个负整数，并将会调用 Math.abs()方法来获取其绝对值；

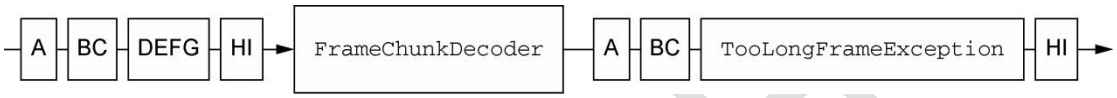
编码器将会把每个负整数的绝对值写到 ChannelPipeline 中。



测试异常处理

应用程序通常需要执行比转换数据更加复杂的任务。例如，你可能需要处理格式不正确的输入或者过量的数据。在下一个示例中，如果所读取的字节数超出了某个特定的限制，我们将会抛出一个 TooLongFrameException。

这是一种经常用来防范资源被耗尽的方法。设定最大的帧大小已经被设置为 3 字节。如果一个帧的大小超出了该限制，那么程序将会丢弃它的字节，并抛出一个 TooLongFrameException。位于 ChannelPipeline 中的其他 ChannelHandler 可以选择在 exceptionCaught()方法中处理该异常或者忽略它。



4、Netty 进阶和实战

实现 UDP 单播和广播

UDP 这样的无连接协议中，并没有持久化连接这样的概念，并且每个消息（一个 UDP 数据报）都是一个单独的传输单元。此外，UDP 也没有 TCP 的纠错机制。

通过类比，TCP 连接就像打电话，其中一系列的有序消息将会在两个方向上流动。相反，UDP 则类似于往邮箱中投入一叠明信片。你无法知道它们将以何种顺序到达它们的目的地，或者它们是否所有的都能够到达它们的目的地。

UDP 的这些方面可能会让你感觉到严重的局限性，但是它们也解释了为何它会比 TCP 快那么多：所有的握手以及消息管理机制的开销都已经被消除了。显然，UDP 很适合那些能够处理或者容忍消息丢失的应用程序，但可能不适合那些处理金融交易的应用程序。

本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有 Nagle 算法之说了），他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP 头+IP 头等等发一次数据封装一次）也就**没有粘包**一说了。

单播的传输模式，定义为发送消息给一个由唯一的地址所标识的单一的网络目的地。面向连接的协议和无连接协议都支持这种模式。

广播——传输到网络（或者子网）上的所有主机。

Netty 的 UDP 相关类

interface **AddressedEnvelope**<M, A extends SocketAddress>extends ReferenceCounted

定义一个消息，其包装了另一个消息并带有发送者和接收者地址。其中 M 是消息类型；A 是

地址类型

class **DefaultAddressedEnvelope**<M, A extends SocketAddress>implements AddressedEnvelope<M,A>提供了 interface AddressedEnvelope 的默认实现

class **DatagramPacket** extends DefaultAddressedEnvelope<ByteBuf, InetSocketAddress> implements ByteBufHolder

扩展了 DefaultAddressedEnvelope 以使用 ByteBuf 作为消息数据容器。DatagramPacket 是 final 类不能被继承，只能被使用。

通过 content()来获取消息内容

通过 sender();来获取发送者的消息

通过 recipient();来获取接收者的消息。

interface **DatagramChannel** extends Channel

扩展了 Netty 的 Channel 抽象以支持 UDP 的多播组管理

class **NioDatagramChannnel** extends AbstractNioMessageChannel implements DatagramChannel

定义了一个能够发送和接收 Addressed-Envelope 消息的 Channel 类型

Netty 的 DatagramPacket 是一个简单的消息容器，DatagramChannel 实现用它来和远程节点通信。类似于在我们先前的类比中的明信片，它包含了接收者（和可选的发送者）的地址以及消息的有效负载本身。

UDP 单播

参见模块 udp 下的包 cn.enjoyedu.udp.unicast

UDP 广播

参见模块 udp 下的包 cn.enjoyedu.udp.broadcast

服务器推送技术-短轮询和 Comet

服务器推送技术干嘛用？就是让用户在使用网络应用的时候，不需要一遍又一遍的去手动刷新就可以及时获得更新的信息。大家平时在上各种视频网站时，对视频节目进行欢乐的吐槽和评论，会看到各种弹幕，当然，他们是用 flash 技术实现的，对于我们没有用 flash 的应用，一样可以实现弹幕。又比如在股票网站，往往可以看到，各种股票信息的实时刷新，上面的这些都是基于服务器推送技术。

具体代码参见模块 comet

Ajax 短轮询

就是用一个定时器不停的去网站上请求数据。

Comet

“服务器推”是一种很早就存在的技术，以前在实现上主要是通过客户端的套接口，或是服务器端的远程调用。因为浏览器技术的发展比较缓慢，没有为“服务器推”的实现提供很好的支持，在纯浏览器的应用中很难有一个完善的方案去实现“服务器推”并用于商业程序。，因为 AJAX 技术的普及，gmail 等等在实现中使用了这些新技术；同时“服务器推”在现实应

用中确实存在很多需求。称这种基于 HTTP 长连接、无须在浏览器端安装插件的“服务器推”技术为“Comet”。

基于 AJAX 的长轮询

DeferredResult:

Spring mvc 的控制层接收用户的请求之后，如果要采用异步处理，那么就要返回 DeferedResult<>泛型对象。在调用完控制层之后，立即回返回 DeferedResult 对象，此时驱动控制层的容器主线程，可以处理更多的请求。

可以将 DeferedResult 对象作为真实响应数据的代理，而真实的数据是该对象的成员变量 result，它可以是 String 类型，或者 ModelAndView 类型等。

业务处理完毕之后，要执行 setResult 方法，将真实的响应数据赋值到 DeferedResult 对象中。此时，容器主线程会继续执行 getResult 方法，将真实数据响应到客户端。

SSE

严格地说，HTTP 协议无法做到服务器主动推送信息。但是，有一种变通方法，就是服务器向客户端声明，接下来要发送的是流信息（streaming）。

也就是说，发送的不是一次性的数据包，而是一个数据流，会连续不断地发送过来。这时，客户端不会关闭连接，会一直等着服务器发过来的新的数据流，视频播放就是这样的例子。本质上，这种通信就是以流信息的方式，完成一次用时很长的下载。

SSE 就是利用这种机制，使用流信息向浏览器推送信息。它基于 HTTP 协议，目前除了 IE/Edge，其他浏览器都支持。

SSE 与 WebSocket 作用相似，都是建立浏览器与服务器之间的通信渠道，然后服务器向浏览器推送信息。

总体来说，WebSocket 更强大和灵活。因为它是全双工通道，可以双向通信；SSE 是单向通道，只能服务器向浏览器发送，因为流信息本质上就是下载。如果浏览器向服务器发送信息，就变成了另一次 HTTP 请求。

SSE 也有自己的优点。

- SSE 使用 HTTP 协议，现有的服务器软件都支持。WebSocket 是一个独立协议。
- SSE 属于轻量级，使用简单；WebSocket 协议相对复杂。
- SSE 默认支持断线重连，WebSocket 需要自己实现。
- SSE 一般只用来传送文本，二进制数据需要编码后传送，WebSocket 默认支持传送二进制数据。
- SSE 支持自定义发送的消息类型。

HTTP 头信息

服务器向浏览器发送的 SSE 数据，必须是 UTF-8 编码的文本，具有如下的 **HTTP 头信息。**

Content-Type: text/event-stream

Cache-Control: no-cache

Connection: keep-alive

上面三行之中，第一行的 Content-Type 必须指定 MIME 类型为 event-steam。

信息格式

每一次发送的信息，由若干个 message 组成，每个 message 之间用\n\n 分隔。每个 message 内部由若干行组成，每一行都是如下格式。

[field]: value\n

上面的 field 可以取四个值。

- data
- event
- id
- retry

此外，还可以有冒号开头的行，表示注释。通常，服务器每隔一段时间就会向浏览器发送一个注释，保持连接不中断。例子 ：this is a test stream\n\n

data 字段

数据内容用 data 字段表示。

data: message\n\n

如果数据很长，可以分成多行，最后一行用\n\n 结尾，前面行都用\n 结尾。

data: begin message\n

data: continue message\n\n

下面是一个发送 JSON 数据的例子。

data: {\n

data: "foo": "bar",\n

data: "baz", 555\n

data: }\n\n

id 字段

数据标识符用 id 字段表示，相当于每一条数据的编号。

id: msg1\n

data: message\n\n

浏览器用 lastEventId 属性读取这个值。一旦连接断线，浏览器会发送一个 HTTP 头，里面包含一个特殊的 Last-Event-ID 头信息，将这个值发送回来，用来帮助服务器端重建连接。因此，这个头信息可以被视为一种同步机制。

event 字段

event 字段表示自定义的事件类型，默认是 message 事件。浏览器可以用 addEventListener()监听该事件。

event: foo\n

data: a foo event\n\n

data: an unnamed event\n\n

event: bar\n

data: a bar event\n\n

上面的代码创造了三条信息。第一条的名字是 foo，触发浏览器的 foo 事件；第二条未取名，表示默认类型，触发浏览器的 message 事件；第三条是 bar，触发浏览器的 bar 事件。

retry 字段

服务器可以用 retry 字段，指定浏览器重新发起连接的时间间隔。

retry: 10000\n

两种情况会导致浏览器重新发起连接：一种是时间间隔到期，二是由于网络错误等原因，导致连接出错。

技术比较

京东用的什么？Ajax 短轮询，这说明什么？这些技术并没有什么优劣之分，只有合不合适业务的问题。京东的痛点是什么？要用有限的资源来为千万级甚至上亿的用户提供服务，如果是用长连接，对于接入的服务器，比如说 Nginx，是很大的压力，光是为用户维持这个长连接都需要成百上千的 Nginx 的服务器，这是很划不来的。因为对于京东这类购物网站来说，用户的浏览查询量是远远大于用户下单量的，京东需要注重的是服务更多的用户，而且相对于用户浏览页面的图片等等的流量而言，这点带宽浪费占比是很小的。所以我们看京东的付款后的实现，是用的短轮询机制，而且时长放大到了 5 秒。

	短轮询	Serlvet异步(长轮询)	SSE	WebSocket
浏览器支持度	最高	很高	中(IE和Edge均不支持)	中(早期的浏览器不支持)
实时性	最低	较高	很高	很高
代码实现复杂度	最容易	较容易	容易	最复杂
连接性质	短连接	长连接	长连接	长连接
适用	需要服务极大量或极小量的用户，实时性要求不高	准实时性的应用，比较关注浏览器的兼容性	实时，基本都是文本交互的应用	实时，需要支持多样化的用户数据类型的应用或者是原生程序

SSE 和 WebSocket 相比的优势。最大的优势就是便利：不需要添加任何新组件，用任何你习惯的后端语言和框架就能继续使用。你不用为新建虚拟机、弄一个新的 IP 或新的端口号而劳神，就像在现有网站中新增一个页面那样简单。可以称为既存基础设施优势。

SSE 的第二个优势是服务端的简洁。相对而言，WebSocket 则很复杂，不借助辅助类库基本搞不定。WebSocket 能做的，SSE 也能做，反之亦然，但在完成某些任务方面，它们各有千秋。WebSocket 是一种更为复杂的服务端实现技术，但它是真正的双向传输技术，既能从服务端向客户端推送数据，也能从客户端向服务端推送数据。

WebSocket 通信

什么是 WebSocket？

WebSocket ——一种在 2011 年被互联网工程任务组（IETF）标准化的协议。

WebSocket 解决了一个长期存在的问题：既然底层的协议（HTTP）是一个请求/响应模式的交互序列，那么如何实时地发布信息呢？AJAX 提供了一定程度上的改善，但是数据流仍然是由客户端所发送的请求驱动的。还有其他的一些或多或少的取巧方式(Comet)

WebSocket 规范以及它的实现代表了对一种更加有效的解决方案的尝试。简单地说，WebSocket 提供了“在一个单个的 TCP 连接上提供双向的通信……结合 WebSocket API……它为网页和远程服务器之间的双向通信提供了一种替代 HTTP 轮询的方案。”

，但是最终它们仍然属于扩展性受限的变通之法。也就是说，WebSocket 在客户端和服务器之间提供了真正的双向数据交换。WebSocket 连接允许客户端和服务器之间进行全双工通信，以便任一方都可以通过建立的连接将数据推送到另一端。WebSocket 只需要建立一次连接，就可以一直保持连接状态。这相比于轮询方式的不停建立连接显然效率要大大提高。

Web 浏览器和服务器都必须实现 WebSockets 协议来建立和维护连接。

特点

- HTML5 中的协议，实现与客户端与服务器双向，基于消息的文本或二进制数据通信
- 适合于对数据的实时性要求比较强的场景，如通信、直播、共享桌面，特别适合于客户与服务频繁交互的情况下，如实时共享、多人协作等平台。
- 采用新的协议，后端需要单独实现
- 客户端并不是所有浏览器都支持

WebSocket 通信握手

Websocket 借用了 HTTP 的协议来完成一部分握手

客户端的请求：

Connection 必须设置 Upgrade，表示客户端希望连接升级。

Upgrade 字段必须设置 Websocket，表示希望升级到 Websocket 协议。

Sec-WebSocket-Key 是随机的字符串，服务器端会用这些数据来构造出一个 SHA-1 的信息摘要。把 “Sec-WebSocket-Key” 加上一个特殊字符串“258EAFA5-E914-47DA-95CA-C5AB0DC85B11”，然后计算 SHA-1 摘要，之后进行 BASE-64 编码，将结果做为 “Sec-WebSocket-Accept” 头的值，返回给客户端。如此操作，可以尽量避免普通 HTTP 请求被误认为 Websocket 协议。

Sec-WebSocket-Version 表示支持的 Websocket 版本。RFC6455 要求使用的版本是 13，之前草案的版本均应当弃用。

服务器端：

Upgrade: websocket

Connection: Upgrade

依然是固定的，告诉客户端即将升级的是 Websocket 协议，而不是 mozillasocket，lurnarsocket 或者 shitsocket。

然后， Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key 。

后面的， Sec-WebSocket-Protocol 则是表示最终使用的协议。

至此，HTTP 已经完成它所有工作了，接下来就是完全按照 Websocket 协议进行

WebSocket 通信-STOMP

WebSocket 是个规范，在实际的实现中有 HTML5 规范中的 WebSocket API、WebSocket 的子协议 STOMP。

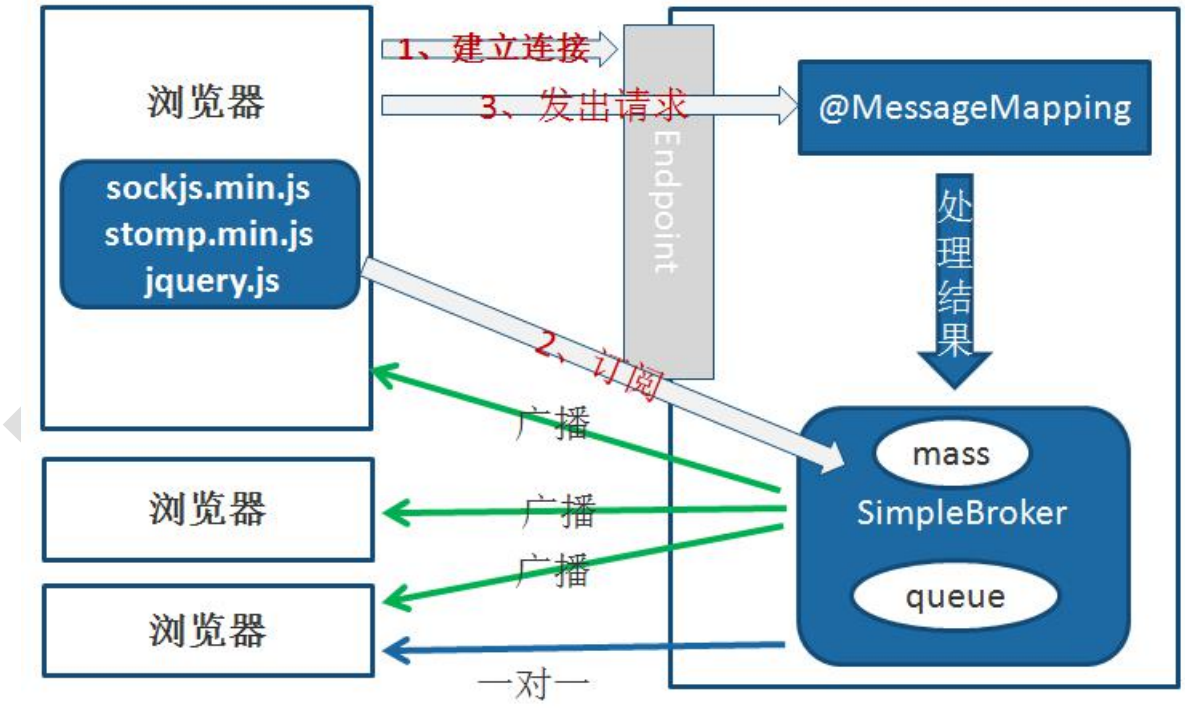
STOMP(Simple Text Oriented Messaging Protocol)

- 简单(流)文本定向消息协议
- STOMP 协议的前身是 TTMP 协议（一个简单的基于文本的协议），专为消息中间件设计。是属于消息队列的一种协议，和 AMQP, JMS 平级. 它的简单性恰巧可以用于定义 websocket 的消息体格式. STOMP 协议很多 MQ 都已支持，比如 RabbitMq, ActiveMq。
- 生产者（发送消息）、消息代理、消费者（订阅然后收到消息）
  STOMP 是基于帧的协议

WebSocket 通信实现

SpringBoot

基于 Stomp 的聊天室/IM 的实现



具体实现：参考 stomp 模块下的代码

和 WebSocket 的集成

具体实现：参考 ws 模块下的代码

Netty

由 IETF 发布的 WebSocket RFC，定义了 6 种帧，Netty 为它们每种都提供了一个 POJO 实现。同时 Netty 也为我们提供很多的 handler 专门用来处理数据压缩，ws 的通信握手等等。

表 12-1 WebSocketFrame 的类型

帧 类 型	描 述
BinaryWebSocketFrame	包含了二进制数据
TextWebSocketFrame	包含了文本数据
ContinuationWebSocketFrame	包含属于上一个 BinaryWebSocketFrame 或 TextWebSocket-Frame 的文本数据或者二进制数据
CloseWebSocketFrame	表示一个 CLOSE 请求，包含一个关闭的状态码和关闭的原因
PingWebSocketFrame	请求传输一个 PongWebSocketFrame
PongWebSocketFrame	作为一个对于 PingWebSocketFrame 的响应被发送

具体实现：参考 netty-ws 模块下的代码

高级通信服务实现-设计自己的协议栈

定义

通信协议从广义上区分，可以分为公有协议和私有协议。由于私有协议的灵活性，它往往会在某个公司或者组织内部使用，按需定制，也因为如此，升级起来会非常方便，灵活性好。绝大多数的私有协议传输层都基于 TCP/IP，所以利用 Netty 的 NIO TCP 协议栈可以非常方便地进行私有协议的定制和开发。

私有协议本质上是厂商内部发展和采用的标准，除非授权，其他厂商一般无权使用该协议。私有协议也称非标准协议，就是未经国际或国家标准化组织采纳或批准，由某个企业自己制订，协议实现细节不愿公开，只在企业自己生产的设备之间使用的协议。私有协议具有封闭性、垄断性、排他性等特点。

跨节点通信

在传统的 Java 应用中，通常使用以下 4 种方式进行跨节点通信。

（1）通过 RMI 进行远程服务调用；

（2）通过 Java 的 Socket+Java 序列化的方式进行跨节点调用；

（3）利用一些开源的 RPC 框架进行远程服务调用，例如 Facebook 的 Thrift，Apache 的 Avro 等；

（4）利用标准的公有协议进行跨节点服务调用，例如 HTTP+XML、RESTful+JSON 或者 WebService。

跨节点的远程服务调用，除了链路层的物理连接外，还需要对请求和响应消息进行编解码。在请求和应答消息本身以外，也需要携带一些其他控制和管理类指令，例如链路建立的握手请求和响应消息、链路检测的心跳消息等。当这些功能组合到一起之后，就会形成私有协议。

协议栈功能设计

协议栈功能描述

Netty 协议栈承载了业务内部各模块之间的消息交互和服务调用，它的主要功能如下。

（1）基于 Netty 的 NIO 通信框架，提供高性能的异步通信能力；

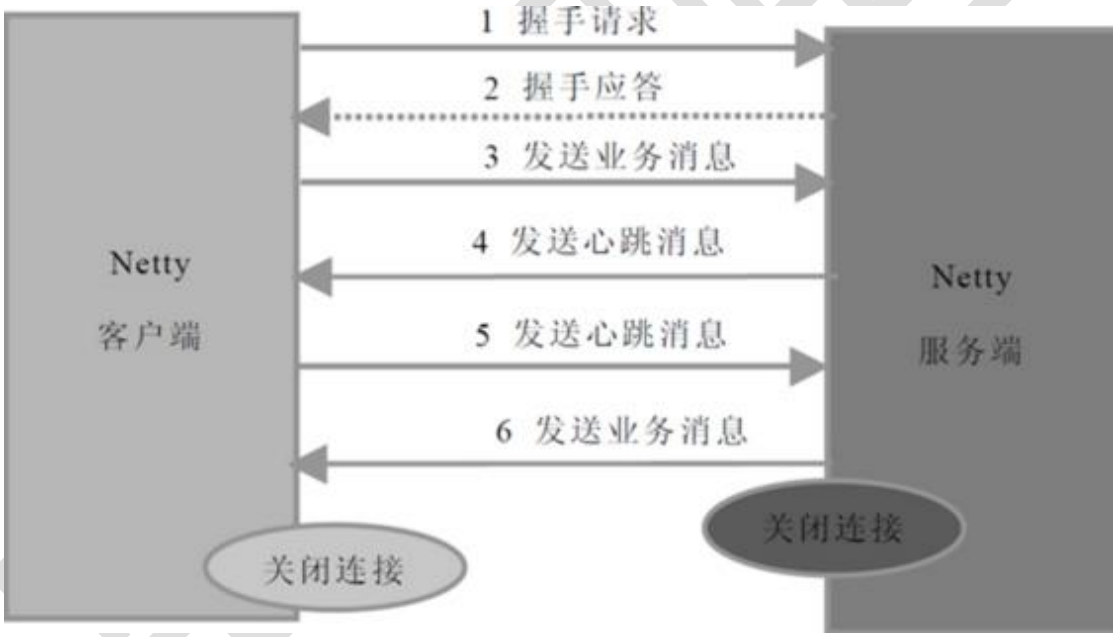
（2）提供消息的编解码框架，可以实现 POJO 的序列化和反序列化；

（3）提供基于 IP 地址的白名单接入认证机制；

（4）链路的有效性校验机制；

（5）链路的断连重连机制。

通信模型



（1）Netty 协议栈客户端发送握手请求消息，携带节点 ID 等有效身份认证信息；

（2）Netty 协议栈服务端对握手请求消息进行合法性校验，包括节点 ID 有效性校验、节点重复登录校验和 IP 地址合法性校验，校验通过后，返回登录成功的握手应答消息；

（3）链路建立成功之后，客户端发送业务消息；

（4）链路成功之后，服务端发送心跳消息；

（5）链路建立成功之后，客户端发送心跳消息；

（6）链路建立成功之后，服务端发送业务消息；

（7）服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

备注：需要指出的是，Netty 协议通信双方链路建立成功之后，双方可以进行全双工通信，无论客户端还是服务端，都可以主动发送请求消息给对方，通信方式可以是 TWO WAY 或者 ONE WAY。双方之间的心跳采用 Ping-Pong 机制，当链路处于空闲状态时，客户端主动发送 Ping 消息给服务端，服务端接收到 Ping 消息后发送应答消息 Pong 给客户端，如果客户端连续发送 N 条 Ping 消息都没有接收到服务端返回的 Pong 消息，说明链路已经挂死或者对方处于异常状态，客户端主动关闭连接，间隔周期 T 后发起重连操作，直到重连成功。

消息定义

Netty 协议栈消息定义包含两部分：

消息头；消息体。

Netty 消息定义表

名称	类型	长度	描述
Header	Header	变长	消息头定义
Body	Object	变长	对于请求消息，它只是方法的参数，对于响应消息，它是返回值

Netty 协议消息头定义（Header）

名称	类型	长度	描述
crcCode	Int	32	Netty 消息校验码
Length	Int	32	整个消息长度
sessionID	Long	64	会话 ID
Type	Byte	8	0:业务请求消息 1：业务响应消息 2：业务 one way 消息 3 握手请求消息 4 握手应答消息 5：心跳请求消息 6：心跳应答消息
Priority	Byte	8	消息优先级：0~255
Attachment	Map<String,Object>	变长	可选字段，由于推展消息头

链路的建立

Netty 协议栈支持服务端和客服端，对于使用 Netty 协议栈的应用程序而言，不需要刻意区分到底是客户端还是服务器端，在分布式组网环境中，一个节点可能既是客户端也是服务器端，这个依据具体的用户场景而定。

Netty 协议栈对客户端的说明如下：如果 A 节点需要调用 B 节点的服务，但是 A 和 B 之间还没有建立物理链路，则有调用方主动发起连接，此时，调用方为客户端，被调用方为服务端。

考虑到安全，链路建立需要通过基于 Ip 地址或者号段的黑白名单安全认证机制，作为样例，本协议使用基于 IP 地址的安全认知，如果有多个 Ip，通过逗号进行分割。在实际的商用项目中，安全认证机制会更加严格，例如通过密钥对用户名和密码进行安全认证。

客户端与服务端链路建立成功之后，由客户端发送握手请求消息，握手请求消息的定义如下

（1） 消息头的 type 字段值为 3；

（2） 可选附件数为 0；

（3） 消息头为空

（4） 握手消息的长度为 22 个字节

服务端接收到客户端的握手请求消息之后，如果 IP 校验通过，返回握手成功应答消息给客户端，应用层链路建立成功。握手应答消息定义如下：

（1）消息头的 type 字段值为 4

（2）可选附件个数为 0；

（3）消息体为 byte 类型的结果，0：认证成功；-1 认证失败；

链路建立成功之后，客户端和服务端就可以互相发送业务消息了。

链路的关闭

由于采用长连接通信，在正常的业务运行期间，双方通过心跳和业务消息维持链路，任何一方都不需要主动关闭连接。

但是，在以下情况下，客户端和服务端需要关闭连接：

（1）当对方宕机或者重启时，会主动关闭链路，另一方读取到操作系统的通知信号得知对方 REST 链路，需要关闭连接，释放自身的句柄等资源。由于采用 TCP 全双工通信，通信双方都需要关闭连接，释放资源；

（2）消息读写过程中，发生了 I/O 异常，需要主动关闭连接；

（3）心跳消息读写过程发生了 I/O 异常，需要主动关闭连接；

（4）心跳超时，需要主动关闭连接；

（5）发生编码异常等不可恢复错误时，需要主动关闭连接。

可靠性设计

Netty 协议栈可能会运行在非常恶劣的网络环境中，网络超时、闪断、对方进程僵死或者处理缓慢等情况都有可能发生。为了保证在这些极端异常场景下 Netty 协议栈仍能够正常工作或者自动恢复，需要对他的可靠性进行统一规划和设计。

心跳机制

在凌晨等业务低谷时段，如果发生网络闪断、连接被 Hang 住等问题时，由于没有业务消息，应用程序很难发现。到了白天业务高峰期时，会发生大量的网络通信失败，严重的会导致一段时间进程内无法处理业务消息。为了解决这个问题，在网络空闲时采用心跳机制来检测链路的互通性，一旦发现网络故障，立即关闭链路，主动重连。

当读或者写心跳消息发生 I/O 异常的时候，说明已经中断，此时需要立即关闭连接，如果是客户端，需要重新发起连接。如果是服务端，需要清空缓存的半包信息，等到客户端重连。

重连机制

如果链路中断，等到 INTEVAL 时间后，由客户端发起重连操作，如果重连失败，间隔周期 INTERVAL 后再次发起重连，直到重连成功。

为了保持服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待 INTERVAL 时间之后再发起重连，而不是失败后立即重连。

为了保证句柄资源能够及时释放，无论什么场景下重连失败，客户端必须保证自身的资源被及时释放，包括但不现居 SocketChannel、Socket 等。

重连失败后，需要打印异常堆栈信息，方便后续的问题定位。

重复登录保护

当客户端握手成功之后，在链路处于正常状态下，不允许客户端重复登录，以防止客户端在异常状态下反复重连导致句柄资源被耗尽。

服务端接收到客户端的握手请求消息之后，首先对 IP 地址进行合法性校验，如果校验成功，在缓存的地址表中查看客户端是否已经登录，如果登录，则拒绝重复登录，返回错误码-1，同时关闭 TCP 链路，并在服务端的日志中打印握手失败的原因。

客户端接收到握手失败的应答消息之后，关闭客户端的 TCP 连接，等待 INTERVAL 时间之后，再次发起 TCP 连接，知道认证成功。

为了防止由服务端和客户端对链路状态理解不一致导致的客户端无法握手成功问题，当服务端连续 N 次心跳超时之后需要主动关闭链路，清空改客户端的地址缓存信息，以保证后续改客户端可以重连成功，防止被重复登录保护机制拒绝掉。

测试

1、 正常情况

2、 客户端宕机，服务器应能清除客户端的缓存信息，允许客户端重新登录

3、 服务器宕机，客户端应能发起重连

4、在 LoginAuthRespHandler 中进行注释，可以模拟当服务器不处理客户端的请求时，客户端在超时后重新进行登录。