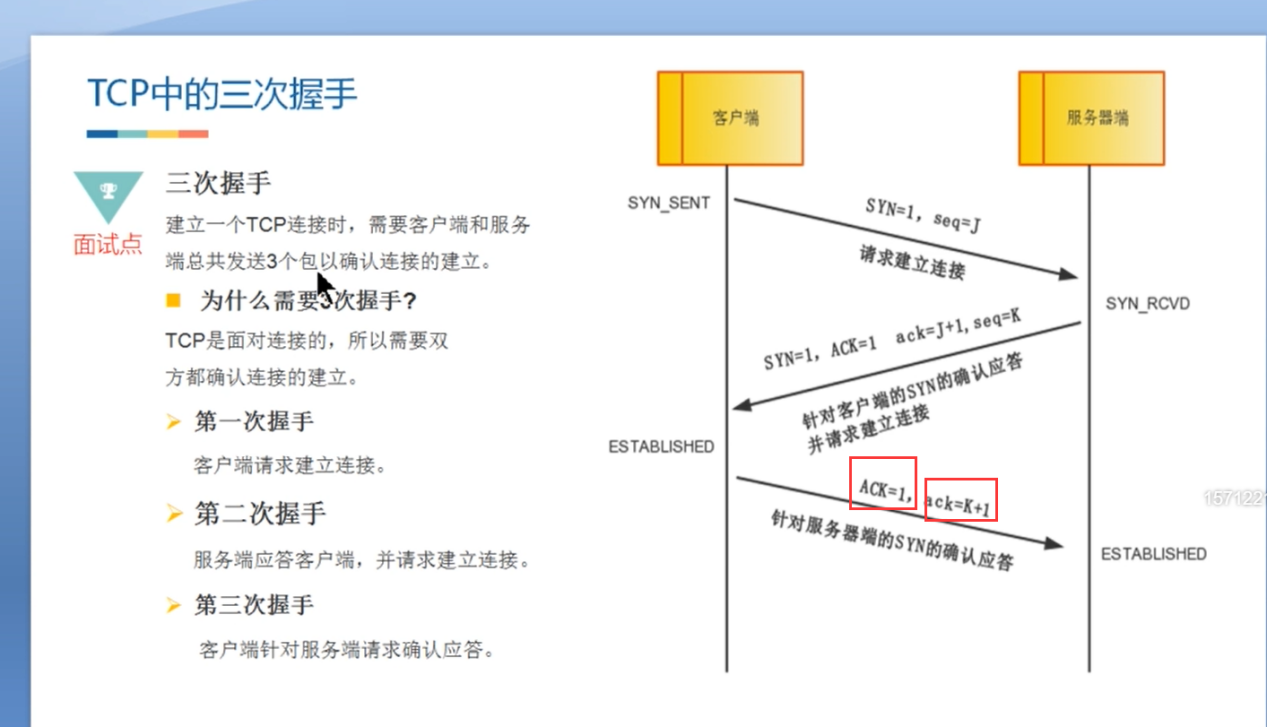
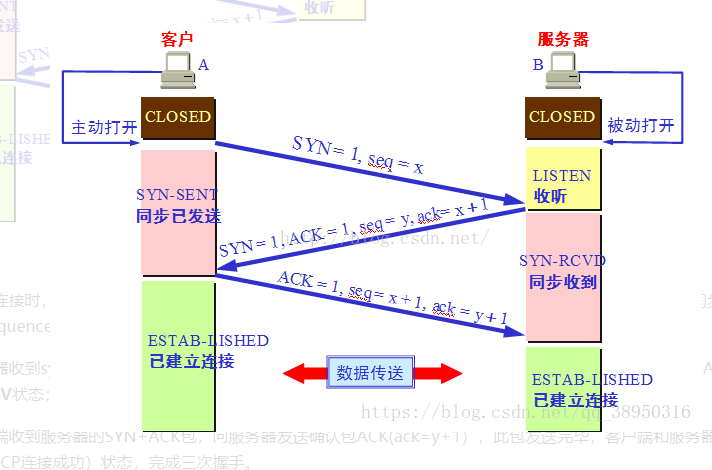
**网络通信**

# 三次握手



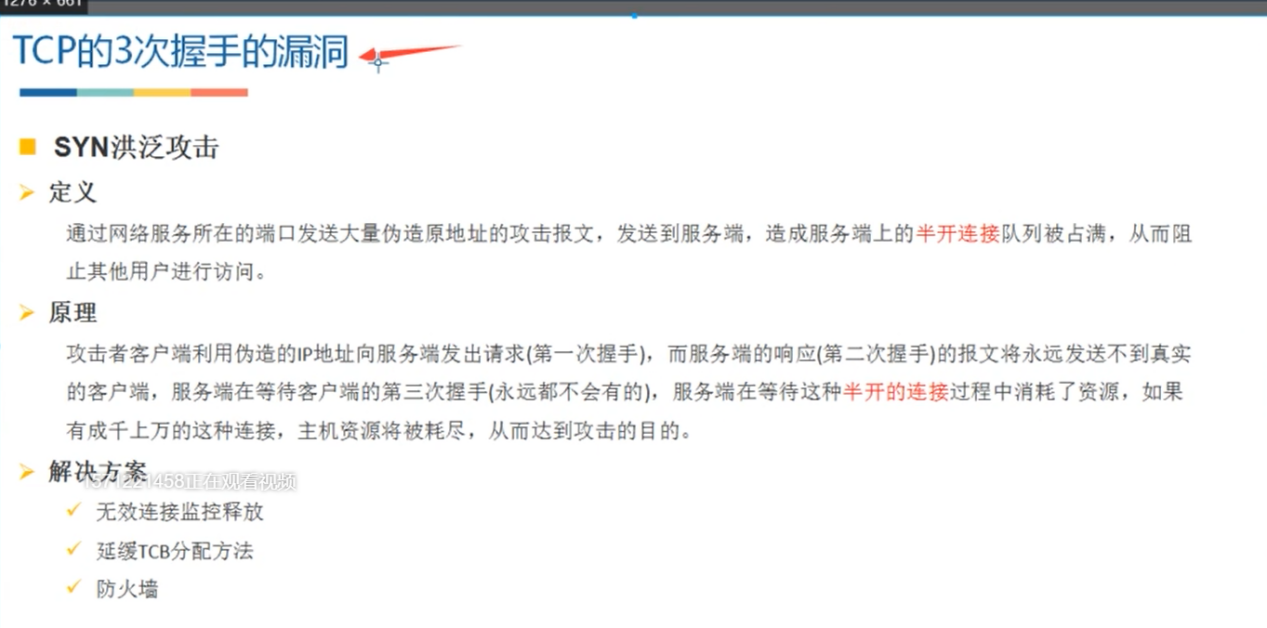


第一次握手：建立连接时，客户端发送syn包（syn=x）到服务器，并进入SYN\_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

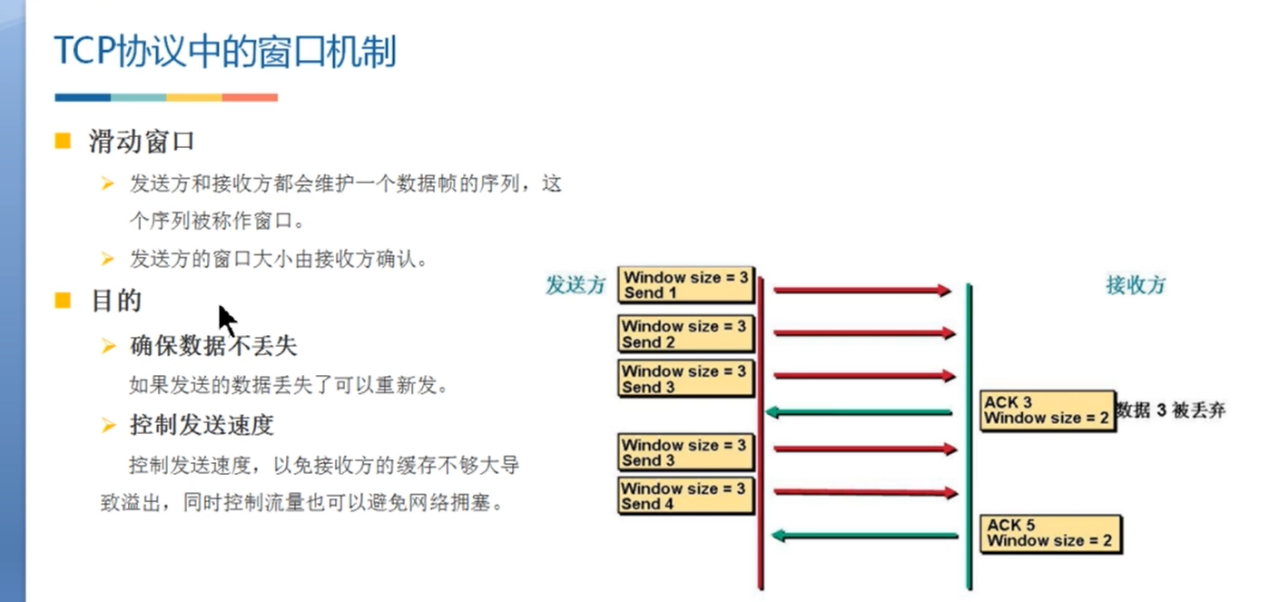
第二次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

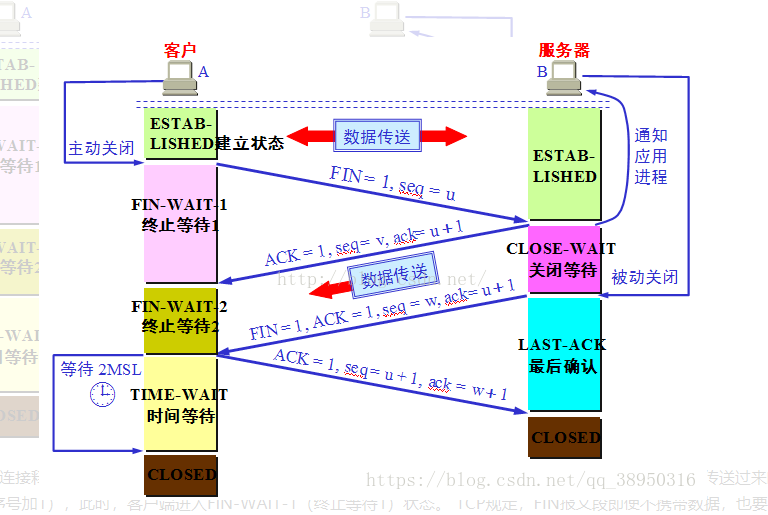
## 漏洞



## 滑动窗口



# 四次挥手



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。

6）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。

# 为什么连接的时候是三次握手，关闭的时候却是四次握手？

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

**为什么 TCP 握手需要三次?**

TCP 是可靠的传输控制协议，三次握手能保证数据可靠传输又能提高传输效

率。

如果 TCP 的握手是两次：

<1>如果 client 发给 server 的 SYN 报文因为网络原因，延迟发送。由于 client

没有收到 server 对 SYN 的确认报文，会重发 SYN 报文，服务器和回复 ACK，连接

建立。数据发送完毕，这条连接被正常关闭。这时，延迟的 SYN 报文发到了 server，

server 误以为这是 client 重新发送的同步报文，又回复了一个 ACK，和 client 建

立了连接。<2>如果 server 给 client 发送的 ACK 报文因为网络原因，报文被丢弃，此时

server 认为已经建立好连接，但是 client 没有收到确认报文，认为没有建立好连

接。client 会重发 SYN 报文，此时 server 已经处于就绪状态，认为已经建立好连

接。

如果 TCP 的握手是四次：

–1.client 给 server 发送 SYN 同步报文；

–2.server 收到 SYN 后，给 client 回复 ACK 确认报文；

–3.server 给 client 发送 SYN 同步报文；

–4.client 给 server 发送 ACK 确认报文。

第 2.3 步之间，server 和 client 没有任何的数据交互，分开发送相当于多发

了一次 TCP 报文段，SYN 和 ACK 标识只是 TCP 报头的一个标识位。很明显，这两

步可以合并，从而提高连接的速度和效率。

# 为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？

虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。Server如果没有收到ACK，将不断重复发送FIN片段。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME\_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

# 为什么不能用两次握手进行连接？

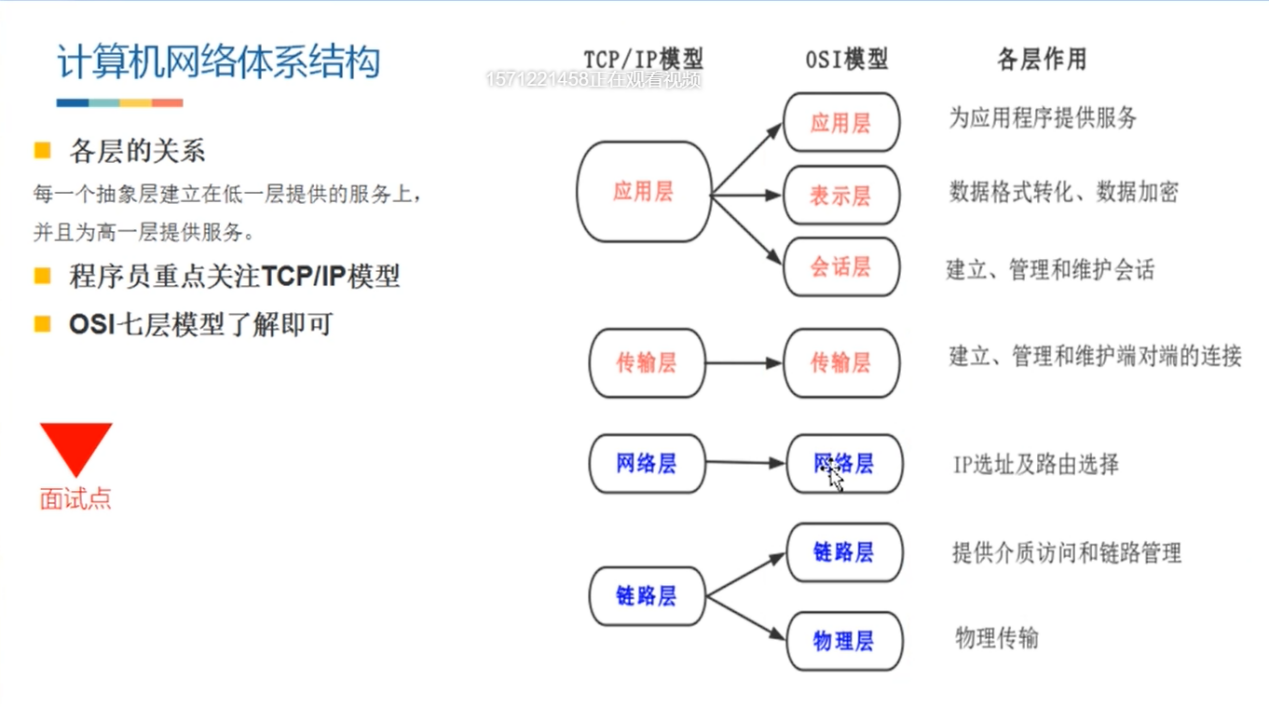
3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

# 如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

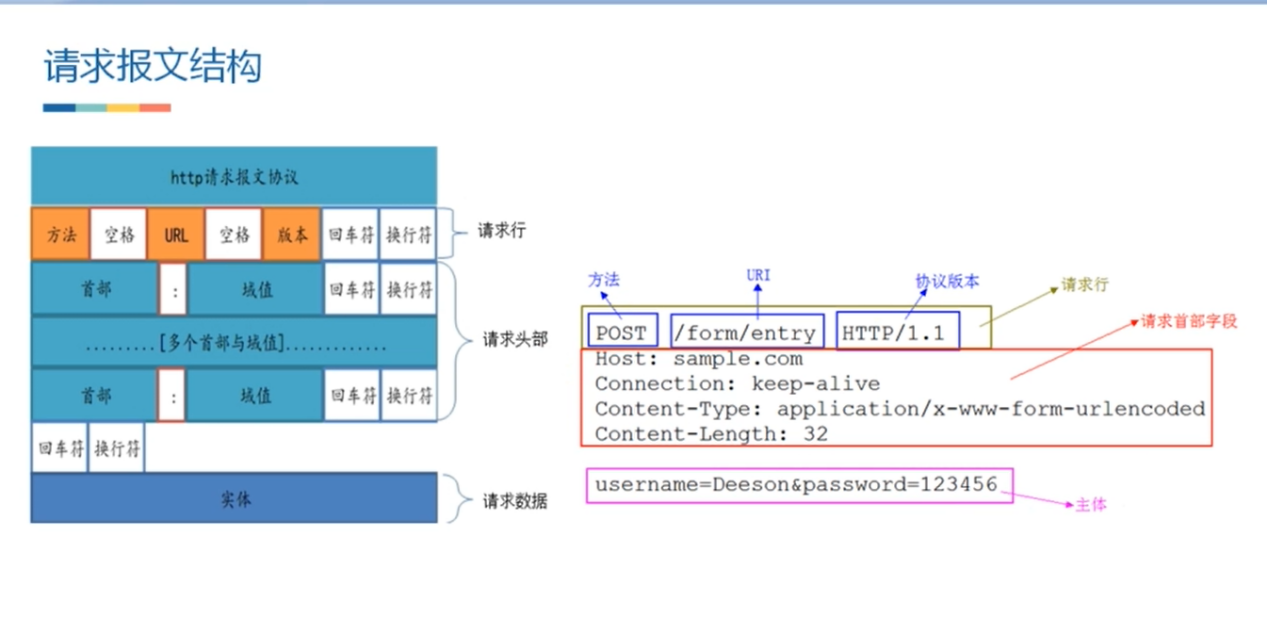
# 计算机网络体系的结构



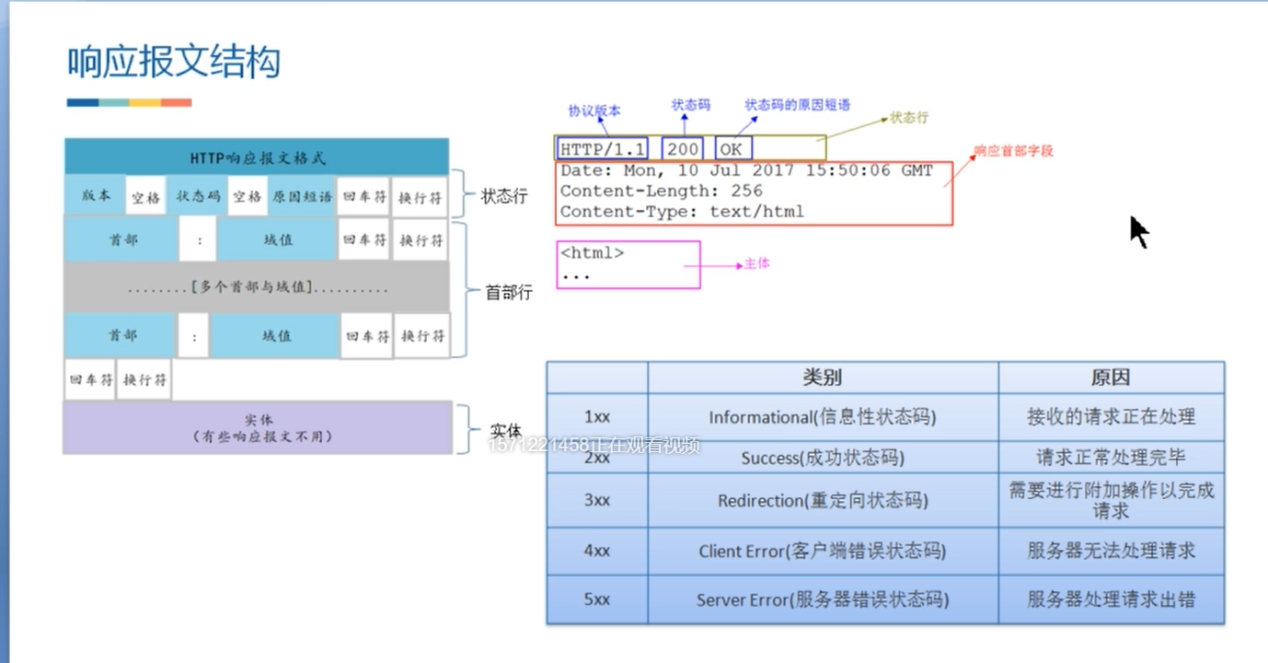
# 一次完整的http请求



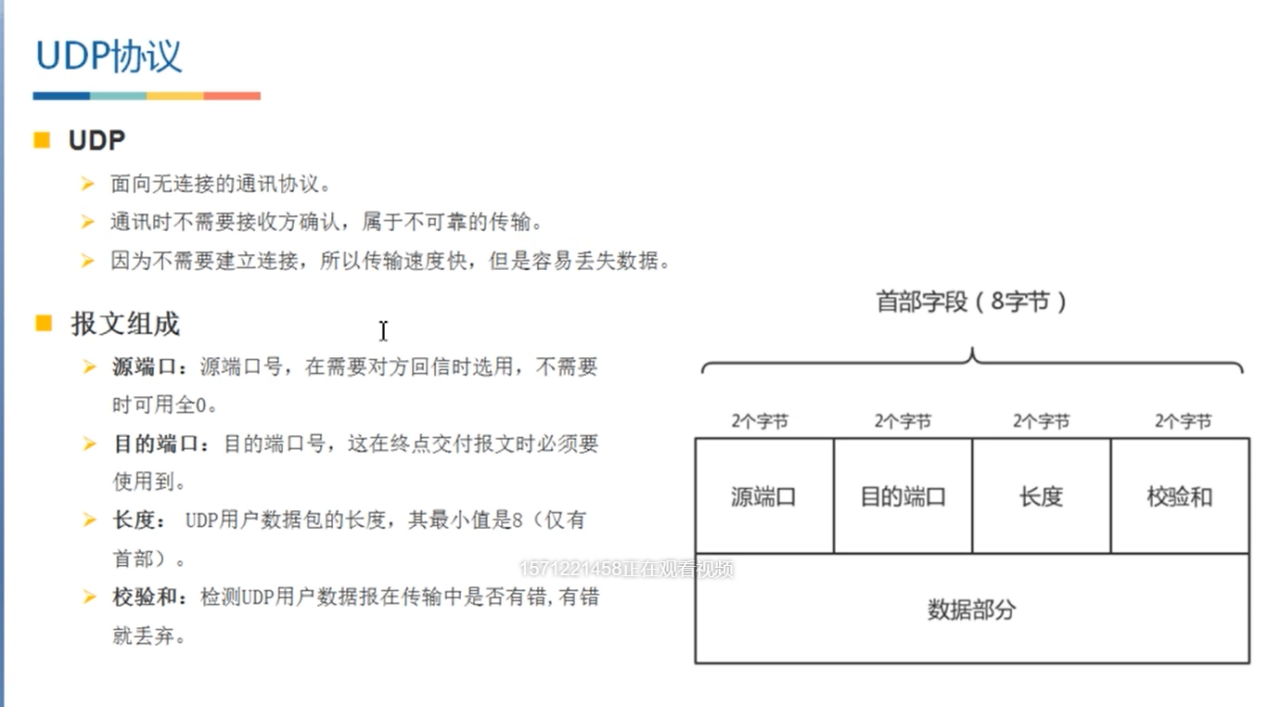
# 请求报文结构



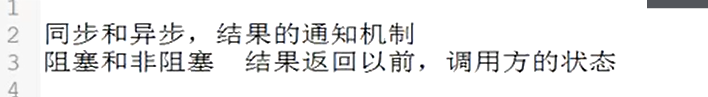
# 响应报文结构



# UDP

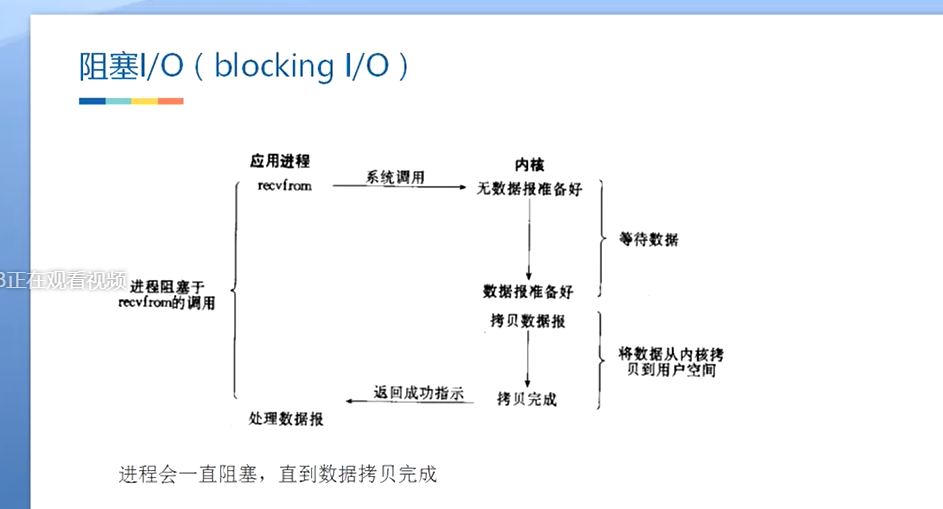


# 阻塞，非阻塞，同步和异步



# IO复用



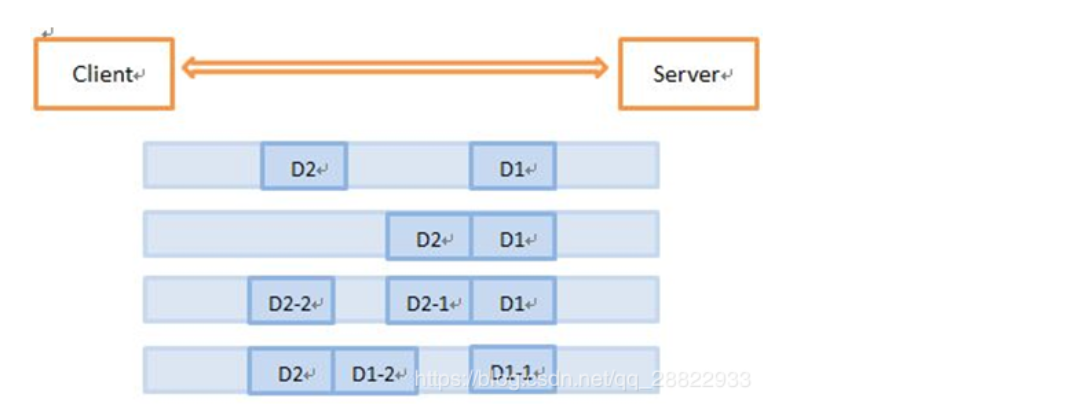


# select、poll、epoll之间的区别

<https://www.cnblogs.com/aspirant/p/9166944.html>

# 粘包，半包

## 定义



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。

## 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；linebase包和delimiter包下，分别使用LineBasedFrameDecoder和DelimiterBasedFrameDecoder，如果超过规定字节长度，会报错。

（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；fixed包下，使用FixedLengthFrameDecoder

（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；。

# **哪些应用比较适合用 udp 实现**

多播的信息一定要用 udp 实现，因为 tcp 只支持一对一通信。

如果一个应用场景中大多是简短的信息，适合用 udp 实现，因为 udp 是基

于报文段的，它直接对上层应用的数据封装成报文段，然后丢在网络中，如果信

息量太大，会在链路层中被分片，影响传输效率。

如果一个应用场景重性能甚于重完整性和安全性，那么适合于 udp，比如多

媒体应用，缺一两帧不影响用户体验，但是需要流媒体到达的速度快，因此比较

适合用 udp

如果要求快速响应，那么 udp 听起来比较合适

如果又要利用 udp 的快速响应优点，又想可靠传输，那么只能考上层应用自

己制定规则了。

常见的使用 udp 的例子：ICQ,QQ 的聊天模块。

# **如果要你来设计一个 QQ，在网络协议上你会考虑如何设计？**

登陆采用 TCP 协议和 HTTP 协议，你和好友之间发送消息，主要采用 UDP 协

议，内网传文件采用了 P2P 技术。总来的说：

1.登陆过程，客户端 client 采用 TCP 协议向服务器 server 发送信息，HTTP

协议下载信息。登陆之后，会有一个 TCP 连接来保持在线状态。

2.和好友发消息，客户端 client 采用 UDP 协议，但是需要通过服务器转发。

腾讯为了确保传输消息的可靠，采用上层协议来保证可靠传输。如果消息发送失

败，客户端会提示消息发送失败，并可重新发送。

3.如果是在内网里面的两个客户端传文件，QQ 采用的是 P2P 技术，不需要

服务器中转。

# Netty

## Netty 的特点？

一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输 的支持使用更高效的 socket 底层，对 epoll 空轮询引起的 cpu 占用飙升在内部进行 了处理，避免了直接使用 NIO 的陷阱，简化了 NIO 的处理方式。 采用多种 decoder/encoder 支持，对 TCP 粘包/分包进行自动化处理 可使用接受/处理线程池，提高连接效率，对重连、心跳检测的简单支持 可配置 IO 线程数、TCP 参数， TCP 接收和发送缓冲区使用直接内存代替堆 内存，通过内存池的方式循环利用 ByteBuf 通过引用计数器及时申请释放不再引用的对象，降低了 GC 频率 使用单线程串行化的方式，高效的 Reactor 线程模型 大量使用了 volitale、使用了 CAS 和原子类、线程安全类的使用、读写锁的 使用

## Netty 的线程模型

Netty 通过 Reactor 模型基于多路复用器接收并处理用户请求，内部实现了 两个线程池，boss 线程池和 work 线程池，其中 boss 线程池的线程负责处理请求 的 accept 事件，当接收到 accept 事件的请求时，把对应的 socket 封装到一个 NioSocketChannel 中，并交给 work 线程池，其中 work 线程池负责请求的 read 和 write 事件，由对应的 Handler 处理。 单线程模型：所有 I/O 操作都由一个线程完成，即多路复用、事件分发和处 理都是在一个 Reactor 线程上完成的。既要接收客户端的连接请求,向服务端发起 连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的 链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高 负载、大并发的应用场景不合适。

多线程模型：有一个 NIO 线程（Acceptor） 只负责监听服务端，接收客户 端的 TCP 连接请求；NIO 线程池负责网络 IO 的操作，即消息的读取、解码、编 码和发送；1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安 全认证时，一个 Acceptor 线程可能会存在性能不足问题。 主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理 I/O 的读写等操作，从而保证 mainReactor 只负责接 入认证、握手等操作；

## TCP 粘包/拆包的原因及解决方法？

TCP 是以流的方式来处理数据，一个完整的包可能会被 TCP 拆分成多个包进 行发送，也可能把小的封装成一个大的数据包发送。 TCP 粘包/分包的原因： 应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象， 而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到 网络上，这将会发生粘包现象； 进行 MSS 大小的 TCP 分段，当 TCP 报文长度-TCP 头部长度>MSS 的时候将发 生拆包以太网帧的 payload（净荷）大于 MTU（1500 字节）进行 ip 分片。 解决方法 消息定长：FixedLengthFrameDecoder 类 包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder 或自定义分 隔符类 ：DelimiterBasedFrameDecoder 将消息分为消息头和消息体：LengthFieldBasedFrameDecoder 类。分为有头 部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘 包

## 请概要介绍下序列化

序列化（编码）是将对象序列化为二进制形式（字节数组），主要用于网络 传输、数据持久化等；而反序列化（解码）则是将从网络、磁盘等读取的字节数 组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。 影响序列化性能的关键因素：序列化后的码流大小（网络带宽的占用）、序 列化的性能（CPU 资源占用）；是否支持跨语言（异构系统的对接和开发语言切 换）。Java 默认提供的序列化：无法跨语言、序列化后的码流太大、序列化的性能 差 XML，优点：人机可读性好，可指定元素或特性的名称。缺点：序列化数据 只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共

属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。适用场

景：当做配置文件存储数据，实时数据转换。

JSON，是一种轻量级的数据交换格式，优点：兼容性高、数据格式比较简单， 易于读写、序列化后数据较小，可扩展性好，兼容性好、与 XML 相比，其协议 比较简单，解析速度比较快。缺点：数据的描述性比 XML 差、不适合性能要求 为 ms 级别的情况、额外空间开销比较大。适用场景（可替代ＸＭＬ）：跨防火 墙访问、可调式性要求高、基于 Web browser 的 Ajax 请求、传输数据量相对小， 实时性要求相对低（例如秒级别）的服务。

Fastjson，采用一种“假定有序快速匹配”的算法。优点：接口简单易用、 目前 java 语言中最快的 json 库。缺点：过于注重快，而偏离了“标准”及功能 性、代码质量不高，文档不全。适用场景：协议交互、Web 输出、Android 客户 端

Thrift，不仅是序列化协议，还是一个 RPC 框架。优点：序列化后的体积小, 速 度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、 支持二进制压缩编码。缺点：使用者较少、跨防火墙访问时，不安全、不具有可 读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如 HTTP）、 无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议。适用场景： 分布式系统的 RPC 解决方案

Protobuf，将数据结构以.proto 文件进行描述，通过代码生成工具可以生成 对应数据结构的 POJO 对象和 Protobuf 相关的方法和属性。优点：序列化后码流 小，性能高、结构化数据存储格式（XML JSON 等）、通过标识字段的顺序，可 以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工 具生成代码、支持的语言相对较少，官方只支持 Java 、C++ 、python。适用场 景：对性能要求高的 RPC 调用、具有良好的跨防火墙的访问属性、适合应用层对 象的持久化。

## Netty 的零拷贝实现

Netty 的零拷贝主要包含三个方面：

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存

进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存

（HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接

内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了

一次缓冲区的内存拷贝。

Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以

像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷

贝的方式将几个小 Buffer 合并成一个大的 Buffer。

Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据

发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题

## Netty 是如何解决 JDK 中的 Selector BUG 的？

Selector BUG：若 Selector 的轮询结果为空，也没有 wakeup 或新消息处理， 则发生空轮询，CPU 使用率 100%， Netty 的解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空 的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则触发了 epoll 死循环 bug。重建 Selector，判断是否是其他线程发起的重建请求，若不是 则将原 SocketChannel 从旧的 Selector 上去除注册，重新注册到新的 Selector 上， 并将原来的 Selector 关闭。

## Netty 的优势有哪些？

使用简单：封装了 NIO 的很多细节，使用更简单。

功能强大：预置了多种编解码功能，支持多种主流协议。

定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。

性能高：通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优。

稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于

业务本身。

社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

## **Netty 高性能表现在哪些方面？**

**IO 线程模型**：同步非阻塞，用最少的资源做更多的事。

**内存零拷贝**：尽量减少不必要的内存拷贝，实现了更高效率的传输。

**内存池设计**：申请的内存可以重用，主要指直接内存。内部实现是用一颗二叉查找树管理内存分配情况。

**串形化处理读写**：避免使用锁带来的性能开销。即消息的处理尽可能在同一

个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。表面

上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO

线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化

的串行线程设计相比一个队列-多个工作线程模型性能更优。

**高性能序列化协议：**支持 protobuf 等高性能序列化协议。

**高效并发编程的体现：**volatile 的大量、正确使用；CAS 和原子类的广泛使用；

线程安全容器的使用；通过读写锁提升并发性能。

## Netty 中有哪些重要组件？

Channel：Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、

connect、read、write 等。

EventLoop：主要是配合 Channel 处理 I/O 操作，用来处理连接的生命周期

中所发生的事情。

ChannelFuture：Netty 框架中所有的 I/O 操作都为异步的，因此我们需要

ChannelFuture 的 addListener()注册一个 ChannelFutureListener 监听事件，当操

作执行成功或者失败时，监听就会自动触发返回结果。

ChannelHandler：充当了所有处理入站和出站数据的逻辑容器。

ChannelHandler 主要用来处理各种事件，这里的事件很广泛，比如可以是连接、

数据接收、异常、数据转换等。

ChannelPipeline：为 ChannelHandler 链提供了容器，当 channel 创建时，

就会被自动分配到它专属的 ChannelPipeline，这个关联是永久性的。

## Netty 发送消息有几种方式？

Netty 有两种发送消息的方式：

直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；

写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从

ChannelPipeline 中的下一个 ChannelHandler 中移动。

## Netty 的内存管理机制是什么？

首先会预申请一大块内存 Arena，Arena 由许多 Chunk 组成，而每个 Chunk

默认由 2048 个 page 组成。

Chunk 通过 AVL 树的形式组织 Page，每个叶子节点表示一个 Page，而中间

节点表示内存区域，节点自己记录它在整个 Arena 中的偏移地址。

当区域被分配出去后，中间节点上的标记位会被标记，这样就表示这个中间

节点以下的所有节点都已被分配了。

大于 8k 的内存分配在 poolChunkList 中，而 PoolSubpage 用于分配小于 8k 的

内存，它会把一个 page 分割成多段，进行内存分配。

## ByteBuf 的特点

支持自动扩容（4M），保证 put 方法不会抛出异常、通过内置的复合缓冲

类型，实现零拷贝（zero-copy）；

不需要调用 flip()来切换读/写模式，读取和写入索引分开；

引用计数基于 AtomicIntegerFieldUpdater 用于内存回收；

PooledByteBuf 采用二叉树来实现一个内存池，集中管理内存的分配和释放，

不用每次使用都新建一个缓冲区对象。UnpooledHeapByteBuf 每次都会新建一个

缓冲区对象