# 网络协议

## 计算机网络是什么？

随着计算机技术发展，计算机的体积和价格都在下降，之前计算机多用于研究机构，现阶段逐步进入一般的公司用于办公。原来计算机之间传输数据需要通过软盘等第三方存储介质进行转存，人们需要将数据直接通过通信线路传输，来缩短传输时间，于是计算机网络开始诞生，并逐渐发展为现在巨大的Internet。

### 定义和分类

计算机网络的标准定义是：利用通信线路将地理上分散的、具有独立功能的计算机系统和通信设备按不同的形式连接起来，以功能完善的网络软件及协议实现资源共享和信息传递的系统。

计算机网络从覆盖范围上划分可以分为三类：局域网、城域网、广域网。局域网LAN（作用范围一般为几米到几十公里）、城域网MAN（界于WAN与LAN之间）、广域网WAN（作用范围一般为几十到几千公里）。当然计算机网络划分不止这一种分类方式，可以按拓扑结构分类（总线型、环型、星型、网状）、还可以按按信息的交换方式（电路交换、报文交换、报文分组交换）来分等等方式。

### 计算机网络发展简史

1、诞生阶段，20世纪60年代中期之前的第一代计算机网络是以单个计算机为中心的远程联机系统。端是一台计算机的外部设备包括显示器和键盘,无CPU和内存。随着远程终端的增多,在主机前增加了前端机。当时,人们把计算机网络定义为“以传输信息为目的而连接起来,实现远程信息处理或进一步达到资源共享的系统”,但这样的通信系统已具备了网络的雏形。

2、ARPANET，多个主机通过通信线路互联起来。60年代初。当时，美国国防部为了保证美国本土防卫力量和海外防御武装在受到前苏联第一次核打击以后仍然具有一定的生存和反击能力，认为有必要设计出一种分散的指挥系统；它由一个个分散的指挥点组成，当部分指挥点被摧毁后，其它点仍能正常工作，并且在这些点之间能够绕过那些已被摧毁的指挥点而继续保持联系。为了对这一构思进行验证，1969年，美国国防部国防高级研究计划署(DOD／DARPA)资助建立了一个名为ARPANET(即"阿帕网")的网络，将多个大学的计算机主机联接起来，位于各个结点的大型计算机采用分组交换技术，通过专门的通信交换机和专门的通信线路相互连接。E-mail、FTP和Telnet在ARPANET上已经诞生。

3、开放性的标准化体系结构，OSI诞生。ARPANET兴起后,计算机网络发展迅猛,各大计算机公司相继推出自己的网络体系结构及实现这些结构的软硬件产品。由于没有统一的标准,不同厂商的产品之间互联很困难,人们迫切需要一种开放性的标准化实用网络环境,这样应运而生了两种国际通用的最重要的体系结构, 为了实现网络设备间的互相通讯，ISO和IEEE（电气和电子工程师协会，是世界上最大的非营利性专业技术学会，其会员人数超过40万人，遍布160多个国家。IEEE致力于电气、电子、计算机工程和与科学有关的领域的开发和研究）相继提出了OSI参考模型及其TCP/IP模型。由于TCP/IP尽早地制定了可行性较强的协议，提出了应对技术快速革新的协议，并及时进行后期改良的方案，因此打败了OSI模型，成为了事实上的标准。

4、Internet互联网

20世纪90年代至今的第四代计算机网络, 1993年是因特网发展过程中非常重要的一年，在这一年中因特网完成了到目前为止所有最重要的技术创新，WWW（万维网）和浏览器的应用使因特网上有了一个令人耳目一新的平台：人们在因特网上所看到的内容不仅只是文字，而且有了图片、声音和动画，、甚至还有了电影。因特网演变成了一个文字、图像、声音、动画、影片等多种媒体交相辉映的新世界，更以前所未有的速度席卷了全世界。出现光纤及高速网络技术,多媒体网络,智能网络,整个网络就像一个对用户透明的大的计算机系统,发展为以Internet为代表的互联网。随着全球范围内以Internet为代表的信息基础设施的建立和发展，促进了信息产业和知识经济的诞生和迅猛发展，标志着人类已进入信息时代，计算机网络的普及与应用正向全人类展开其新的一页，也诞生了很多巨大的互联网公司。

## 计算机网络体系结构

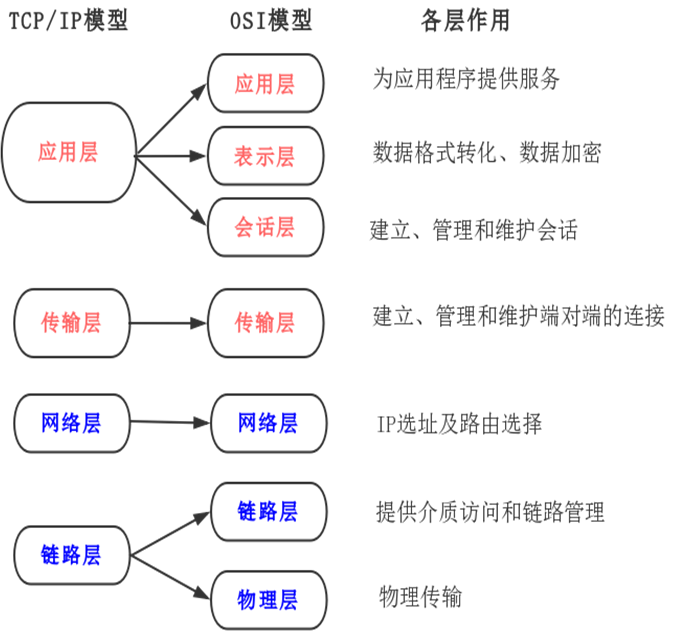
### OSI七层模型

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

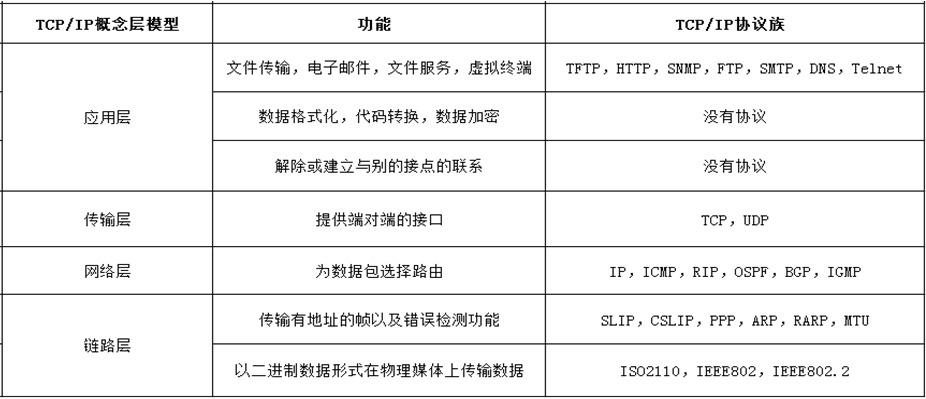
OSI模型比较复杂且学术化，所以我们实际使用的TCP/IP模型，共分4层，**链路层、网络层、传输层、应用层**。两个模型之间的对应关系如图所示：



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

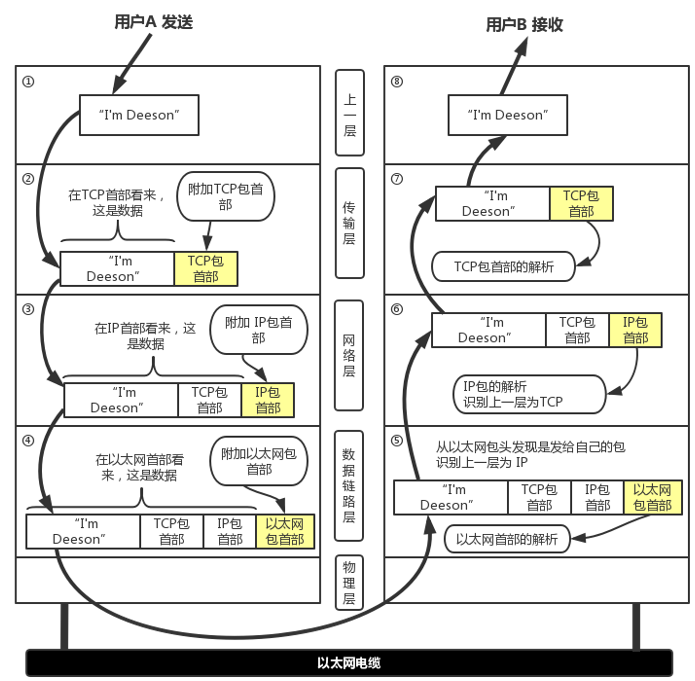
Transmission Control Protocol/Internet Protocol的简写，中译名为传输控制协议/因特网互联协议，是Internet最基本的协议、Internet国际互联网络的基础，由网络层的IP协议和传输层的TCP协议组成。协议采用了4层的层级结构。然而在很多情况下，它是利用 IP 进行通信时所必须用到的协议群的统称。也就是说，它其实是个协议家族，由很多个协议组成，并且是在不同的层， 是互联网的基础通信架构。



### TCP/IP网络传输中的数据

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。

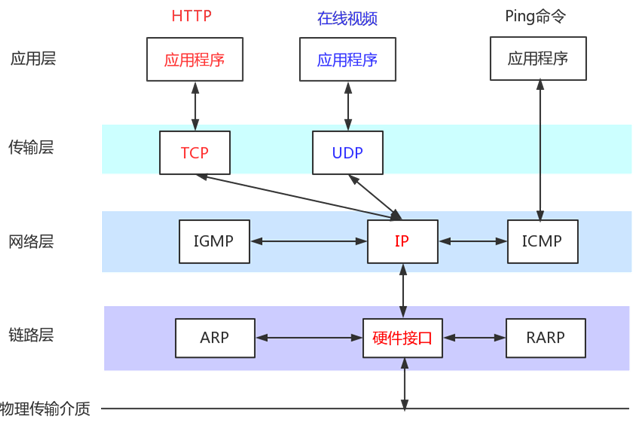
而TCP和UDP都是传输层的协议，传输层主要为两台主机上的应用程序提供端到端的通信。

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

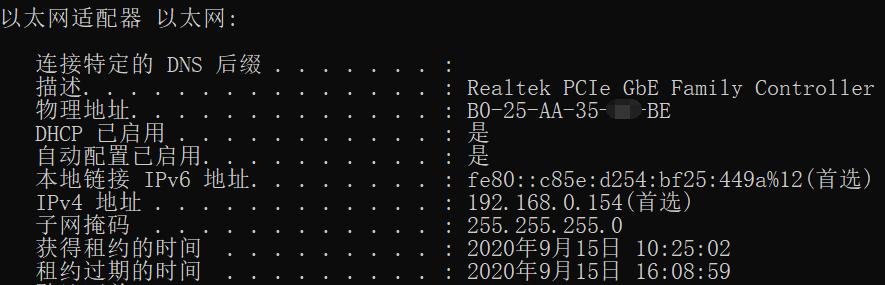
### 地址和端口号

#### MAC 地址

我们常听说 MAC 地址和 IP 地址。

MAC地址全称叫做媒体访问控制地址，也称为局域网地址（LAN Address），MAC位址，以太网地址（Ethernet Address）或物理地址（Physical Address），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。

MAC地址共48位（6个字节）。前24位由IEEE（电气和电子工程师协会）决定如何分配，后24位由实际生产该网络设备的厂商自行制定。例如：FF:FF:FF:FF:FF:FF或FF-FF-FF-FF-FF-FF





#### IP地址

IP地址（Internet Protocol Address）的全称叫作互联网协议地址，它的本义是为互联网上的每一个网络和每一台主机配置一个唯一的逻辑地址，用来与物理地址作区分。

所以IP 地址用来识别 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。

IP地址分为：IPv4和IPv6。我们这里着重讲的是IPv4地址，IP地址是由32位的二进制数组成，它们通常被分为4个“8位二进制数”，我们可以把它理解为4个字节，格式表示为：（A.B.C.D）。其中，A，B，C，D这四个英文字母表示为0-255的十进制的整数。例：192.168.1.1

*Tips：IP地址和MAC地址之间的区别*

*1、对于网络中的一些设备，路由器或者是PC及而言，IP地址的设计是出于拓扑设计出来的，只要在不重复IP地址的情况下，它是可以随意更改的；而MAC地址是根据生产厂商烧录好的，它一般不能改动的，一般来说，当一台PC机的网卡坏了之后，更换了网卡之后MAC地址就会变了。*

*2、在前面的介绍里面，它们最明显的区别就是长度不同，IP地址的长度为32位，而MAC地址为48位。*

*3、它们的寻址协议层不同。IP地址应用于OSI模型的网络层，而MAC地址应用在OSI模型的数据链路层。 数据链路层协议可以使数据从一个节点传递到相同链路的另一个节点上（通过MAC地址），而网络层协议使数据可以从一个网络传递到另一个网络上（ARP根据目的IP地址，找到中间节点的MAC地址，通过中间节点传送，从而最终到达目的网络）。*

*4、分配依据不同。IP地址的分配是基于我们自身定义的网络拓扑，MAC地址的分配是基于制造商。*

#### 端口号

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。

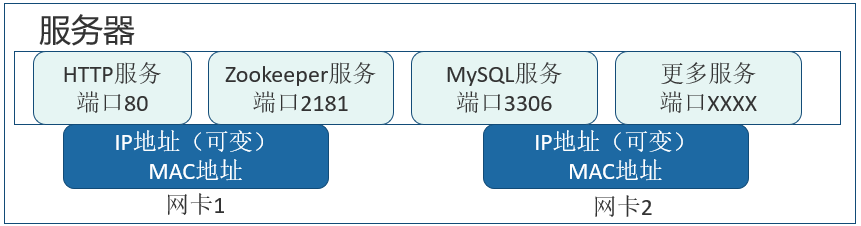
一台计算机上同时可以运行多个程序。传输层协议正是利用这些端口号识别本机中正在进行通信的应用程序，并准确地将数据传输。

##### 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。

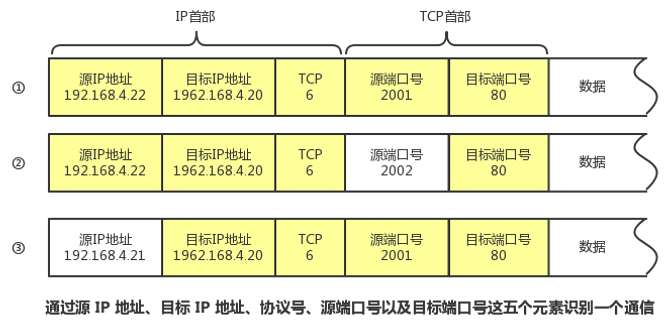
• 时序分配法：服务器有必要确定监听端口号，以让客户端程序访问服务器上的服务。但是使用服务的客户端没必要确定端口号。在这种方法下，客户端应用程序完全可以不用自己设置端口号，而全权交给操作系统进行分配。动态分配的端口号范围在 49152~65535 之间。

#### 综述



所以一般来说，不管计算机中有多少网卡，每个网卡都会有自己的MAC 地址，这个MAC 地址是不会变化的。而每个网卡在正常工作的情况下，都会有一个IP地址，这个IP地址完全是可以变化的。而这台计算机中承载的各种应用程序可以拥有自己的端口号，然后通过服务器的网卡，正确地进行网络通信。

所以通过源IP地址、目标IP地址、协议号（协议类型）、源端口号以及目标端口号这五个元素唯一性的识别一个网络上的通信。



## TCP概述

TCP（Transmission Control Protocol）是面向连接的通信协议，通过三次握手建立连接，然后才能开始数据的读写，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，数据有可能被拆分后发送，那么采用超时重传机制是和应答确认机制是组成TCP可靠传输的关键设计。

而超时重传机制中最最重要的就是重传超时（RTO，Retransmission TimeOut）的时间选择，很明显，在工程上和现实中网络环境是十分复杂多变的，有时候可能突然的抽风，有时候可能突然的又很顺畅。在数据发送的过程中，如果用一个固定的值一直作为超时计时器的时长是非常不经济也非常不准确的方法，这样的话，超时的时长就需要根据网络情况动态调整，就需要采样统计一个数据包从发送端发送出去到接收到这个包的回复这段时长来动态设置重传超时值，这个时长就是为RTT，学名往返时延，round-trip time，然后再根据这个RTT通过各种算法和公式平滑RTT值后，最终确定重传超时值。

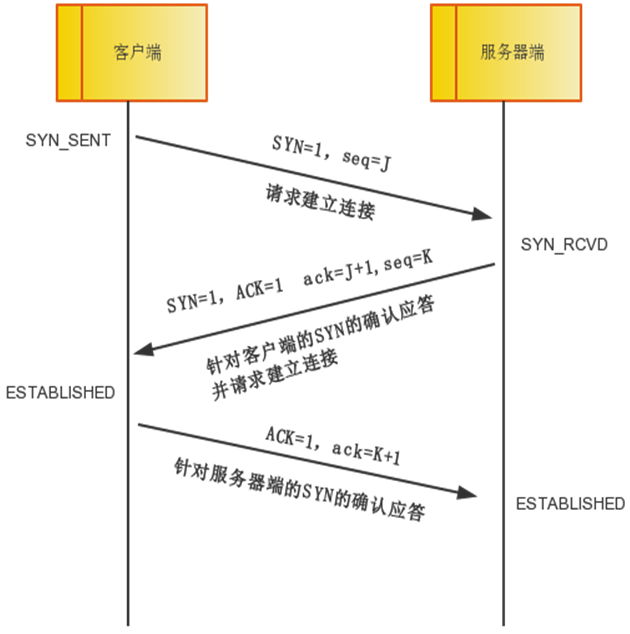
而IP层进行数据传输时，是不能保证数据包按照发送的顺序达到目的机器。当IP将把它们向‘上’传送到TCP层后，TCP将包排序并进行错误检查。TCP数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传。

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。

同时TCP还允许在一个TCP连接上，通信的双方可以同时传输数据，也就是所谓的全双工。

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。

#### 为什么TCP握手需要三次?

TCP是可靠的传输控制协议，而三次握手是保证数据可靠传输又能提高传输效率的最小次数。为什么？RFC793，也就是TCP的协议RFC中就谈到了原因，这是因为：

为了实现可靠数据传输， TCP协议的通信双方，都必须维护一个序列号， 以标识发送出去的数据包中，哪些是已经被对方收到的。

举例说明：发送方在发送数据包（假设大小为 10 byte）时， 同时送上一个序号( 假设为 500)，那么接收方收到这个数据包以后， 就可以回复一个确认号（510 = 500 + 10） 告诉发送方 “我已经收到了你的数据包， 你可以发送下一个数据包， 序号从 511 开始” 。

三次握手的过程即是通信双方**相互告知**序列号起始值，并**确认对方**已经收到了序列号起始值的必经步骤。

如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认。

至于为什么不是四次，很明显，三次握手后，通信的双方都已经知道了对方序列号起始值，也确认了对方知道自己序列号起始值，第四次握手已经毫无必要了。

#### TCP的三次握手的漏洞-SYN洪泛攻击

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，服务端向客户端应答请求，应答请求是需要客户端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。

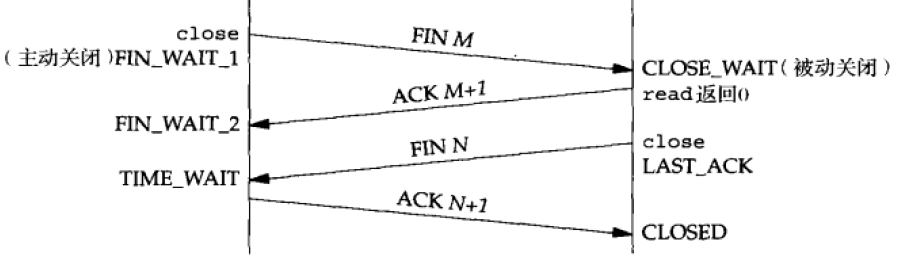
**使用防火墙**

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，这一原则是当甲方完成数据发送任务后，发送一个FIN给乙方来终止这一方向的连接，乙方收到一个FIN只是意味着不会再收到甲方数据了，但是乙方依然可以给甲方发送数据，直到这乙方也发送了FIN给甲方。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。



由此可见，TCP建立一个连接需3个分节，终止一个连接则需4个分节。

(1)某个应用进程首先调用close，我们称该端执行主动关闭（active close)。该端的TCP于是发送一个FIN分节，表示数据发送完毕，应用进程进入FIN-WAIT-1（终止等待1）状态。

(2）接收到这个FIN的对端执行被动关闭(passive close)，发出确认报文。这个FIN由TCP确认。因为FIN的接收意味着接收端应用进程在相应连接上再无额外数据可接收。接收端进入了CLOSE-WAIT（关闭等待）状态，这时候处于半关闭状态，即主动关闭端已经没有数据要发送了，但是被动关闭端若发送数据，主动关闭端依然要接受。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。主动关闭端收到确认报文后进入FIN-WAIT-2（终止等待2）状态。

(3)一段时间后，被动关闭的应用进程将调用close关闭它的套接字。这导致它的TCP也发送一个FIN。

(4)接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN发出一个确认ACK报文，并进入了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/最长分节生命期 max segement lifetime，MSL是任何IP数据报能够在因特网中存活的最长时间，任何TCP实现都必须为MSL选择一个值。RFC 1122[Braden 1989]的建议值是2分钟，不过源自Berkelcy的实现传统上改用30秒这个值。这意味着TIME\_WAIT状态的持续时间在1分钟到4分钟之间）的时间后，当主动关闭端撤销相应的TCB（传输控制块）后，才进入CLOSED状态。

(5) 被动关闭端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，被动关闭端结束TCP连接的时间要比主动关闭端早一些。

既然每个方向都需要一个FIN和一个ACK，因此通常需要4个分节。我们使用限定词“通常”是因为：某些情形下步骤1的FIN随数据一起发送;另外，步骤2和步骤3发送的分节都出自执行被动关闭那-一端，有可能被合并成一个分节。

#### 为什么TCP的挥手需要四次？

TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。

所以对全双工模式来说，为了彻底关闭，就需要通信两端的4次交互。

#### 为什么需要TIME-WAIT状态？

TIME\_WAIT状态存在的原因有两点

1、可靠的终止TCP连接。

2、保证让迟来的TCP报文有足够的时间被识别并丢弃。

根据前面的四次握手的描述，我们知道，客户端收到服务器的连接释放的FIN报文后，必须发出确认。如最后这个ACK确认报文丢失，那么服务器没有收到这个ACK确认报文，就要重发FIN连接释放报文，客户端要在某个状态等待这个FIN连接释放报文段然后回复确认报文段，这样才能可靠的终止TCP连接。

在Linux系统上，一个TCP端口不能被同时打开多次，当一个TCP连接处于TIME\_WAIT状态时，我们无法使用该链接的端口来建立一个新连接。反过来思考，如果不存在TIME\_WAIT状态，则应用程序能过立即建立一个和刚关闭的连接相似的连接（这里的相似，是指他们具有相同的IP地址和端口号）。这个新的、和原来相似的连接被称为原来连接的化身。新的化身可能受到属于原来连接携带应用程序数据的TCP报文段（迟到的报文段），这显然是不该发生的。这是TIME\_WAIT状态存在的第二个原因。

## 常用的网络工具Wireshark和tcpdump

### WireShark

#### 为什么要抓包

1、定位网络问题；

2、分析接口数据；

3、学习网络协议，使用抓包工具分析网络数据更直观。

大部分场合都可以通过程序调试来定位问题，但有些场景使用抓包来定位接口问题更准确、更方便，如以下场景：

1、你发送数据给后台，但后台没有收到，可以对接口进行抓包分析，看是后台处理有问题，还是没有将数据发出去，或是发送数据格式有误；

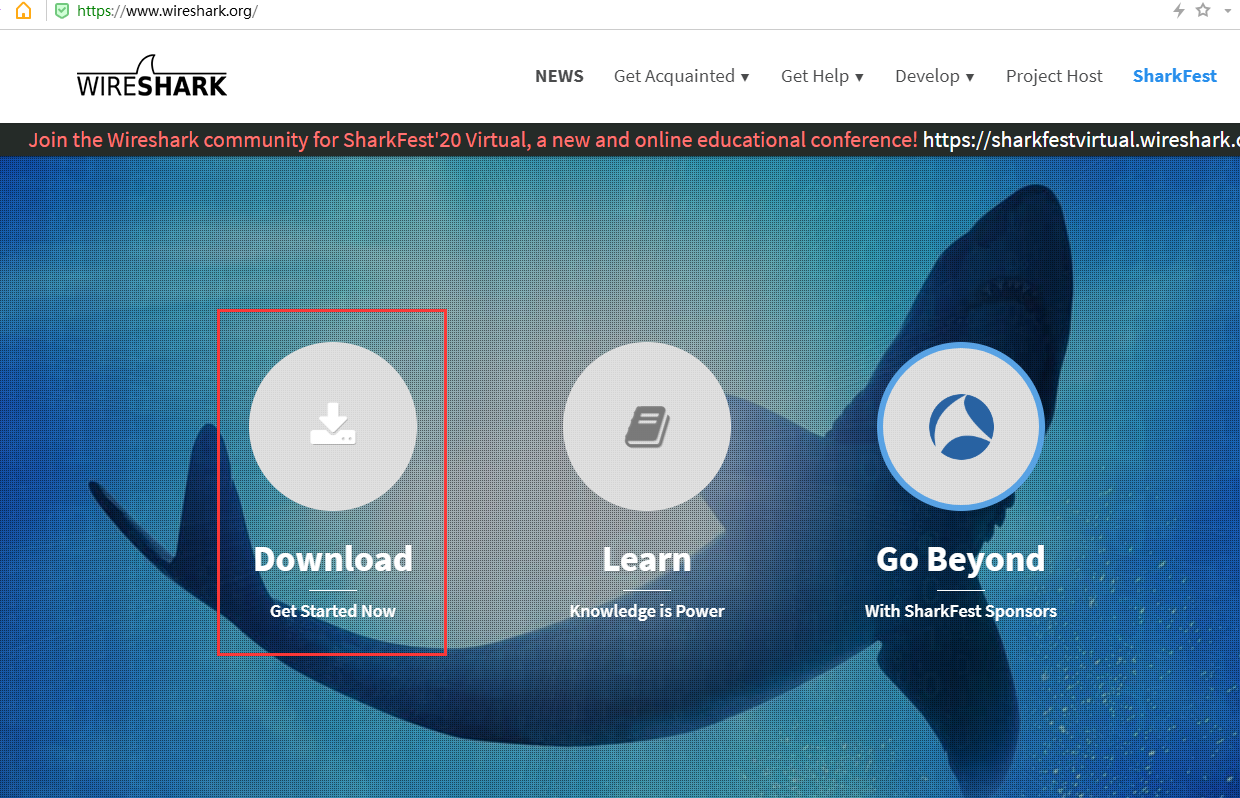
2、你和后台接口联调测通，但业务数据对不上，你认为是后台问题，后台认为是你发的问题，可以抓包确认问题所在；

3、线上出现bug需要定位，但你没在公司，没有代码可调试，可直接抓包分析；

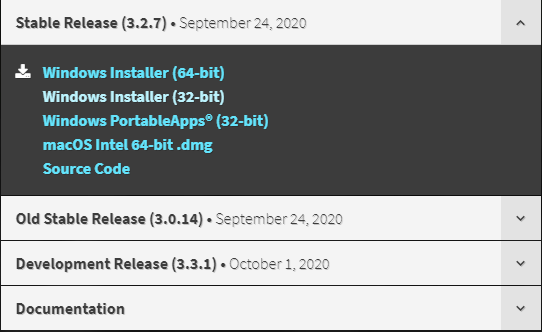
4、系统性能不佳，抓包看下接口响应时长，是不是后台出现性能问题。

常用的抓包工具有：F12（浏览器自带的抓包工具）、Fiddler、Charles，Wireshark。而Wireshark在支持的协议，用户友好度、价格（开源）、程序支持、支持的操作系统上都很好，所以Wireshark也是我们最常用的抓包工具和报文分析工具。

#### 下载与安装

在<https://www.wireshark.org/>的官方主页点击

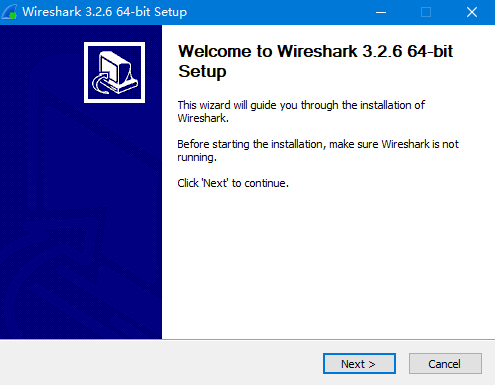
进入下载页面，选择适合自己的版本下载即可



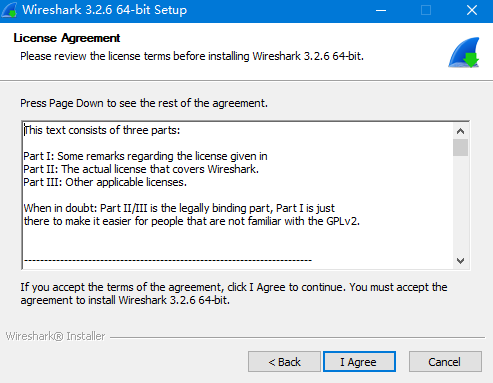
我们以当前版本3.2.7作为讲解版本，网盘中已经提供。

Wireshark的安装很简单，就是一般的windows程序的安装流程，如遇问题自行百度解决。

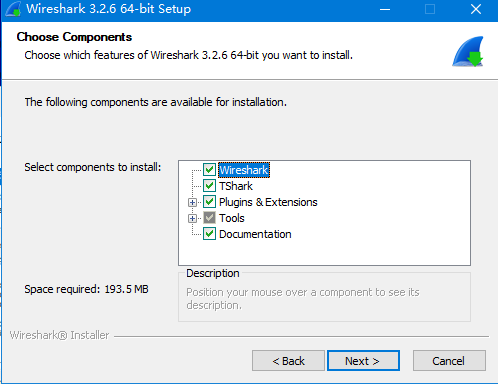
1、双击.exe文件开始进行安装，在介绍页面上单击Next。



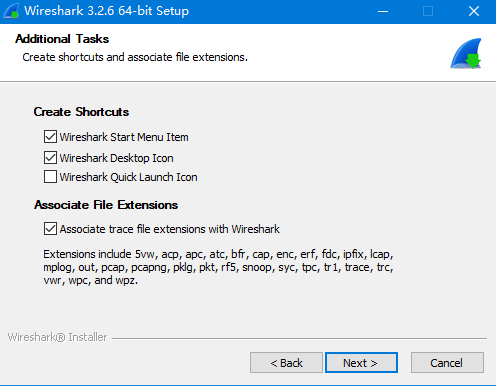
2、同意许可证条款，单击I Agree



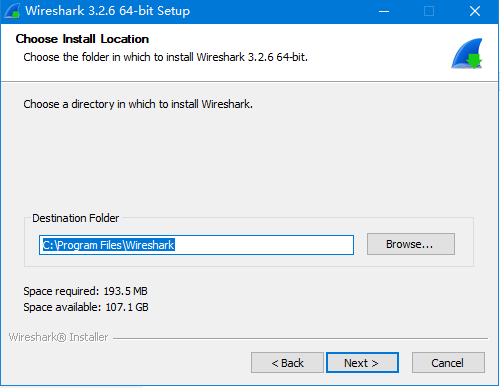
3、接受默认设置，单击Next



4、根据自己的需求在 Additional Tasks窗口中选择后并单击 Next。



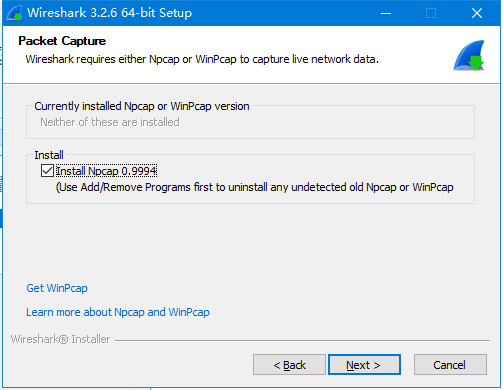
5、选择 Wireshark的安装位置，并单击Next。



6、当弹出是否需要安装 Npcap/WinPcap的对话框时，**务必确保**Install

选项已被勾选，然后单击Install。安装过程便会随即开始。（WinPcap 驱动是 Windows对于pcap数据包捕获的通用程序接口（APT）的实现，简单来说就是这个驱动能够通过操作系统捕捉原始数据包、应用过滤器，并能够让网卡切入或切出混杂模式。*[混杂模式就是接收所有经过网卡的数据包，包括不是发给本机的包，即不验证MAC地址。普通模式下网卡只接收发给本机的包（包括广播包）传递给上层程序，其它的包一律丢弃。一般来说，混杂模式不会影响网卡的正常工作，多在网络监听工具上使用。]*

USBPcap就没必要安装了。



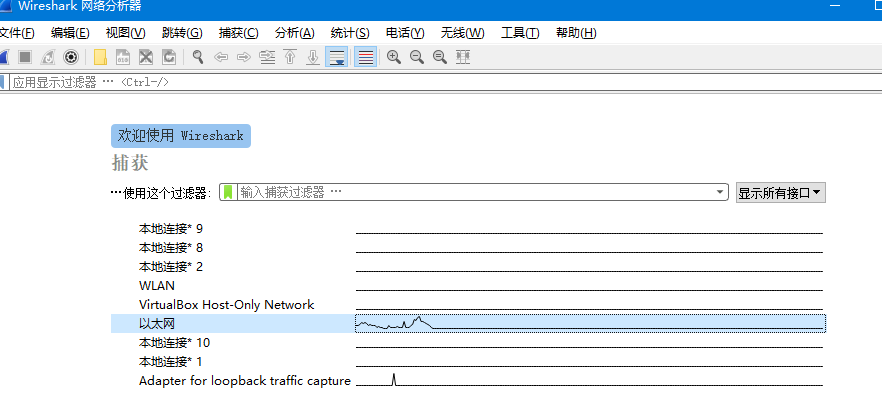
7、耐心等待，同时Wireshark的安装过程中，会开始安装 WinPcap。在介绍页面单击 Next之后，请阅读许可协议并单击I Agree，如有其它安装页面，选择默认配置Install即可。

8、后面一路Next和Finish，并重启电脑即可。

#### 数据包的捕获和基本用法

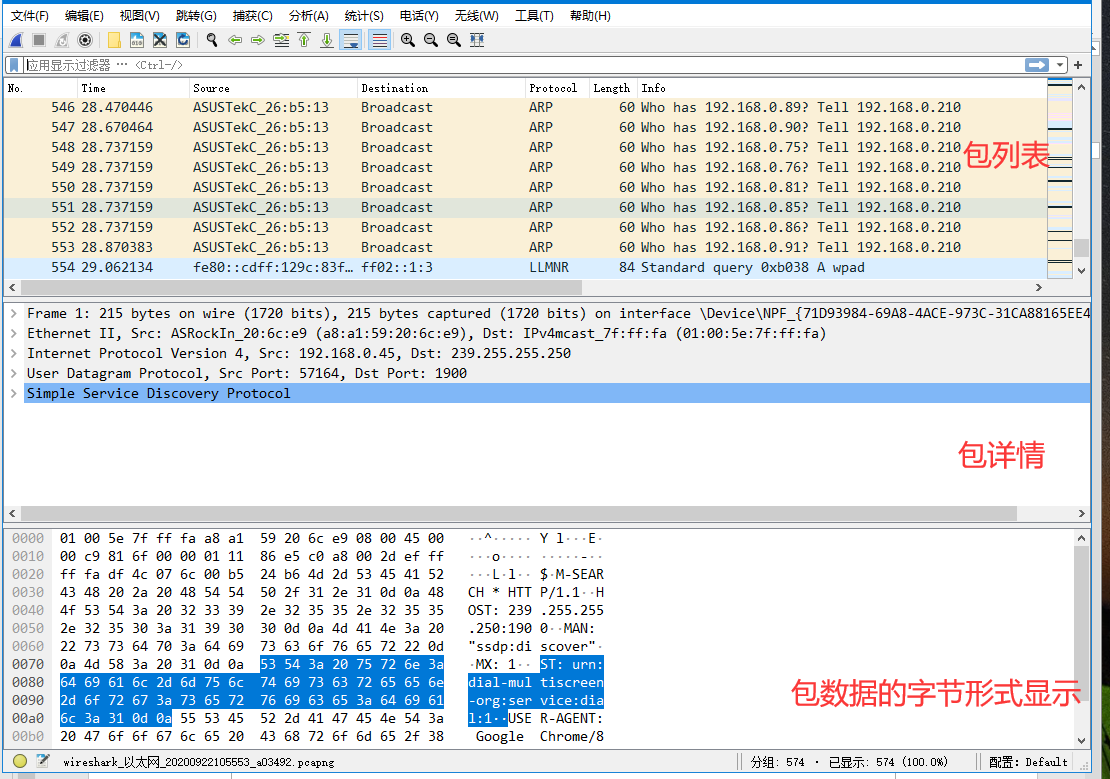
##### 基本用法

运行Wireshark，稍等，就会让我们选择捕获哪个设备（连接）的数据包



当前我们活跃的是“以太网”，双击这一行，Wireshark开始捕获数据，并在窗口中显示，可以看见随着时间前进，窗口中不断刷新。等上几秒钟，结束捕获，单机工具栏中的按钮，或者“捕获”下拉菜单中选择“停止”选项即可。

这个时候可以看见，Wireshark窗口中显示了大量的数据：

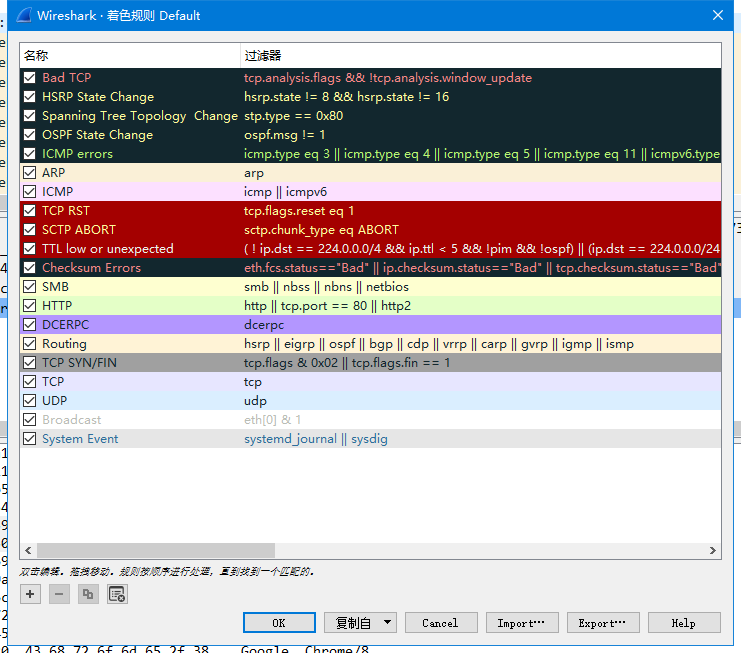


数据包列表：最上面的面板用表格显示了当前捕获文件中的所有数据包，其中包括了数据包序号、数据包被捕获的相对时间、数据包的源地址和目标地址、数据包的协议以及在数据包中找到的概况信息等列。

包详情：中间的面板分层次地显示了一个数据包中的内容，并且可以通过展开或是收缩来显示这个数据包中所捕获到的全部内容。

包数据的字节形式显示：最下面的面板显示了一个数据包未经处理的原始样子，也就是其在链路上传播时的样子。这些原始数据当然不容易理解。

在数据包列表中，可以很明显看见每行的五颜六色，这些颜色其实是Wireshark用来区分不同的协议的。可以通过 Coloring Rules (着色规则）窗口可以很容易地查看或修改每个协议所对应的颜色。如果想要打开这个窗口，可以在“视图”下拉菜单中选择的“着色规则”选项即可。



同时捕获的数据包可以导出为文件，也可以将数据包文件导入Wireshark进行分析。

##### 过滤器

过滤器可以让你找出你所希望进行分析的数据包。简单来说，一个过滤器就是定义了一定条件，用来包含或者排除数据包的表达式。如果你不希望看到一些数据包，你可以写一个过滤器来屏蔽它们。如果你希望只看到某些数据包，你可以写一个只显示这些数据包的过滤器。

Wireshark主要提供两种主要的过滤器。

·捕获过滤器:当进行数据包捕获时，只有那些满足给定的包含/排除表达式

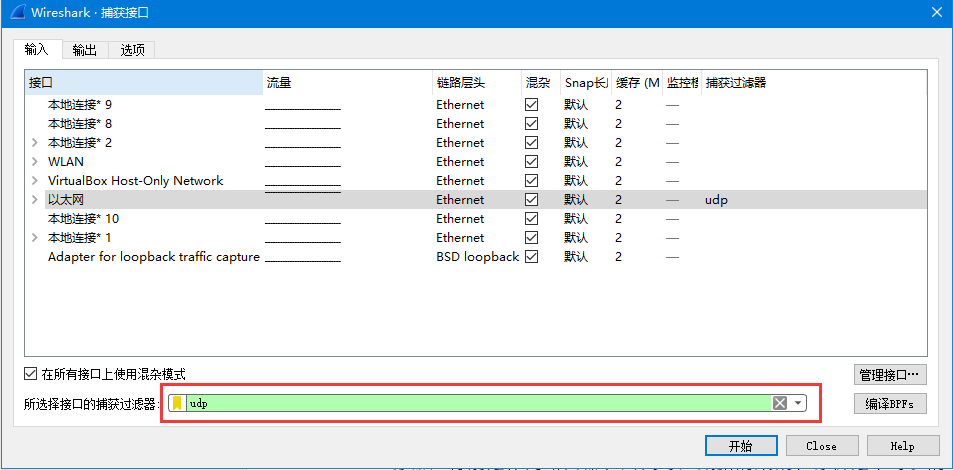
的数据包会被捕获。

显示过滤器:该过滤器根据指定的表达式用于在一个已捕获的数据包集合中，隐藏不想显示的数据包，或者只显示那些需要的数据包。

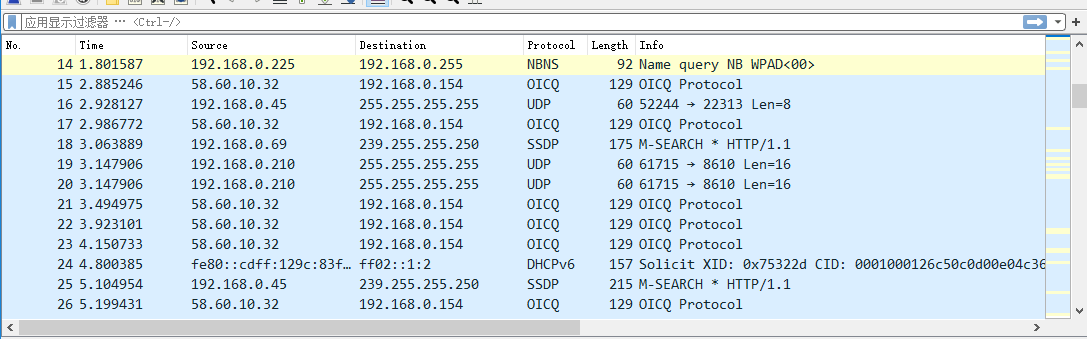
###### 捕获过滤器

捕获过滤器用于进行数据包捕获的实际场合，使用它的一个主要原因就是性能。如果你知道你并不需要分析某个类型的流量，你可以简单地使用捕获过滤器过滤掉它，从而节省那些会被用来捕获这些数据包的资源。

比如，我们现在要捕获我们机器上所有UDP类型的数据包，怎么做？在“捕获”下拉菜单中选择“选型”，弹出窗口中



选择或者输入UDP，然后单击“开始”按钮进行捕获。



捕获过滤器的BPF 语法

捕获过滤器应用于WinPcap/NPcap，并使用 Berkeley Packet Filter（BPF）语法。

这个语法被广泛用于多种数据包嗅探软件，主要因为大部分数据包嗅探软件都依赖于使用 BPF 的 libpcap/WinPcap库。掌握BPF 语法对你在数据包层级更深入地探索网络来说，非常关键。

使用 BPF 语法创建的过滤器被称为表达式，并且每个表达式包含一个或多个原语。每个原语包含一个或多个限定词，然后跟着一个ID名字或者数字

Type 指出名字或数字所代表的意，例如： host、port

Dir 指明传输方向是前往还是来自 例如：src、dst

Proto 限定所要匹配的协议 例如：ether、ip、tcp、udp、http、ftp

例如：

dst host 192.168.0.10 && tcp port 80

这是一个捕获过滤器样例，其中“dst host 192.168.0.10”就构成了一个原语，我们的样例里有两个原语，两个原语用逻辑运算符“&&”组合了起来。逻辑运算符共有三个：

连接运算符： 与 (&&)

选择运算符： 或 (||)

否定运算符： 非 (!)

对我们的样例来说，表示捕获的数据包应该满足条件为：目的主机是192.168.0.10，通信协议是tcp，源端口或者目的端口是80。

主机名和地址过滤器

我们所创建的大多数过滤器都会关注于一个或一些特定的网络设备。根据这个情况，可以根据设备的MAC地址、IPv4地址、IPv6地址或者 DNS主机名配置过滤规则。比如：

捕获所有和主机 IPv4地址相关的流量：

host 192.168.0.10

捕获所有和主机 Ipv6地址相关的流量：

host fe80::1945:cbf1:1393:8f17

使用基于一台设备的主机名host限定词进行过滤：

host remoteserver

考虑到一台主机的IP地址可能会变化，你可以通过加入ether协议限定词，对它的MAC地址进行过滤：

Ether host 24-41-8C-26-04-9F

当然还可在host之前加src 或者 dst进行传输方向的限定。

端口和协议过滤器

只对 8080端口进行流量捕获：

port 8080

想要捕获除8080端口外的所有流量

!port 8080

端口过滤器一样和传输方向限定符一起使用，比如：

dst port 80

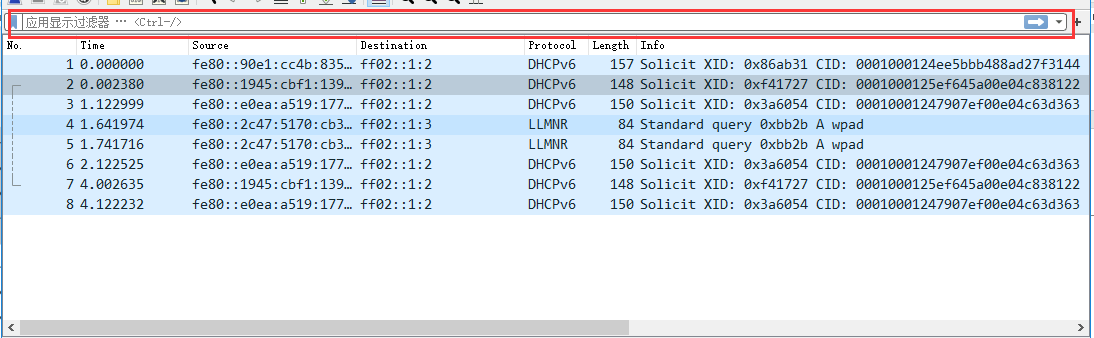
协议过滤器可以让我们基于特定协议进行数据包过滤。BPF 语法甚至提供给我们的一项强大功能，就是我们可以通过检查协议头中的每一字节来创建基于那些数据的特殊过滤器。比如：

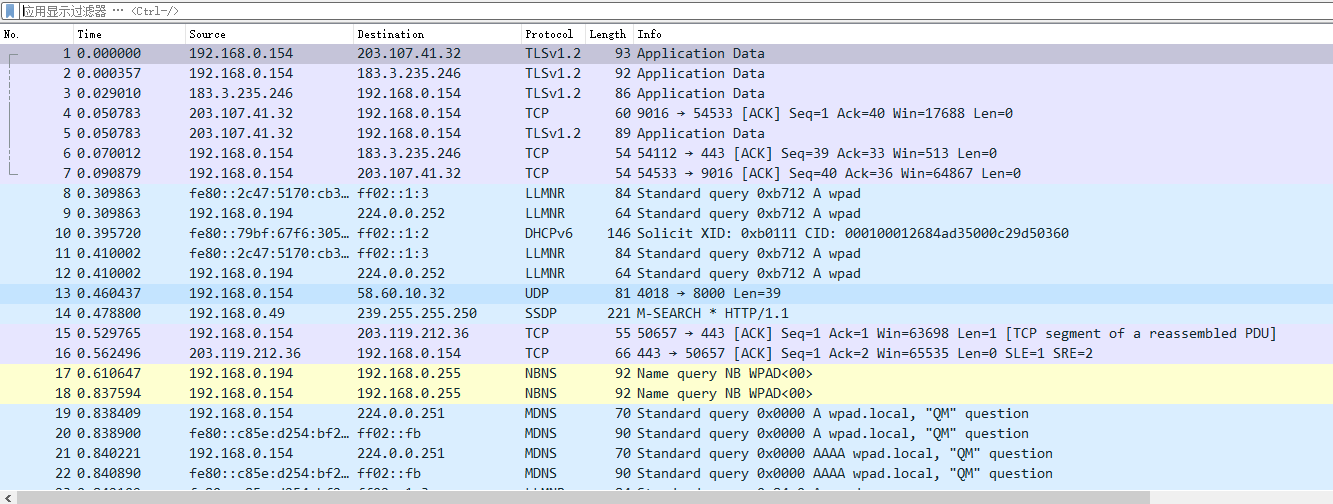
tcp[13]&4=4

表示捕获的数据是tcp协议的数据包，且tcp数据包中第13个字节的第4位被设置了。

###### 显示过滤器

显示过滤器应用于捕获文件，用来告诉Wireshark只显示那些符合过滤条件的数据包。你可以在包列表面板上方的Filter文本框中，输入一个显示过滤器。

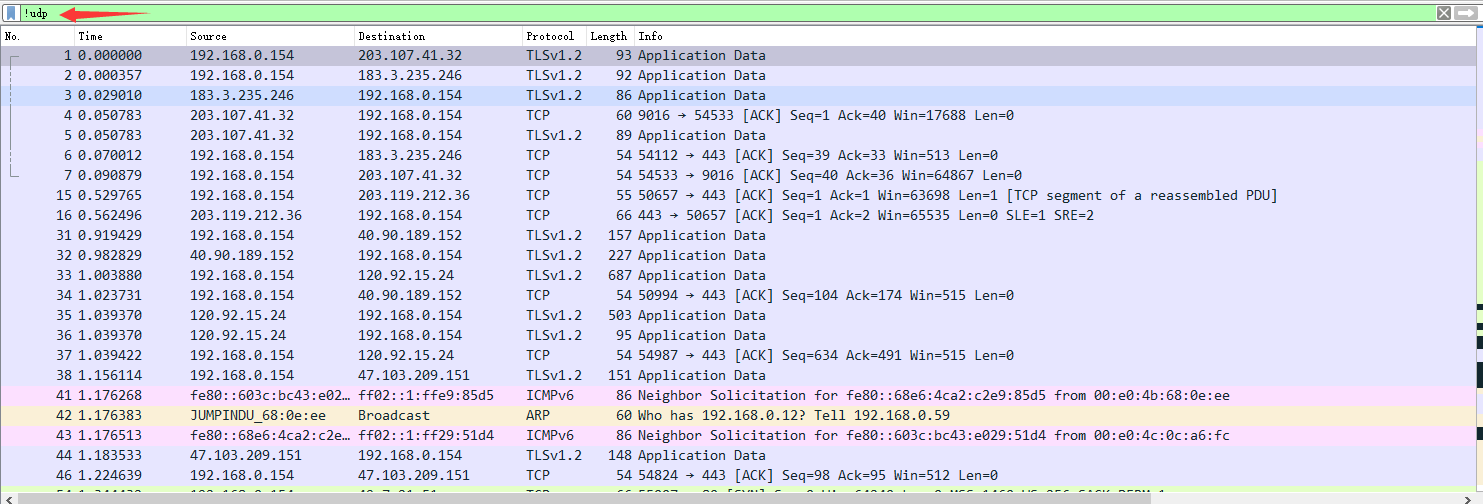


显示过滤器的语法不同于捕获过滤器，比如我们通过捕获功能，捕获了本机所有的数据包  


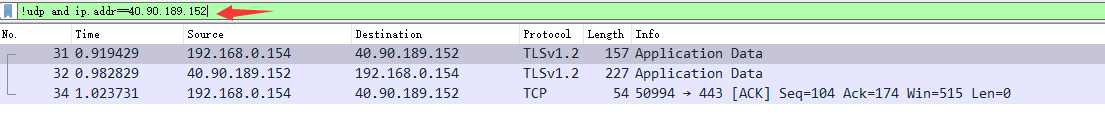
现在我们不需要看UDP类型的数据包，只需输入：

!udp

可以看见所有的udp类型的数据包不再显示了：



现在我们多加一个限定，只显示ip地址为40.90.189.152（新加坡 Microsoft数据中心）的数据包



可以看见，显示的数据包就变为：不是udp数据包，同时源地址或者目的地址是40.90.189.152。

在我们上面的过滤条件“**!udp and ip.addr==40.90.189.152**”中，牵涉到了过滤条件的**比较操作符**和**逻辑操作符**：

比较操作符可以让你进行值的比较。举例来说，当你在检查一个TCP/P 网络中的问题时，你可能经常需要检查和某一个IP地址相关的数据包。等于操作符可以让你创建一个只显示192.168.0.1这个IP地址相关数据包的过滤器。

ip.addr==192.168.0.1

**比较操作符**还有：

等于 ==

不等于 !=

大于 >

小于 <

大于或等于 >=

小于或等于 <=

**逻辑操作符**有：

and 两个条件需同时满足

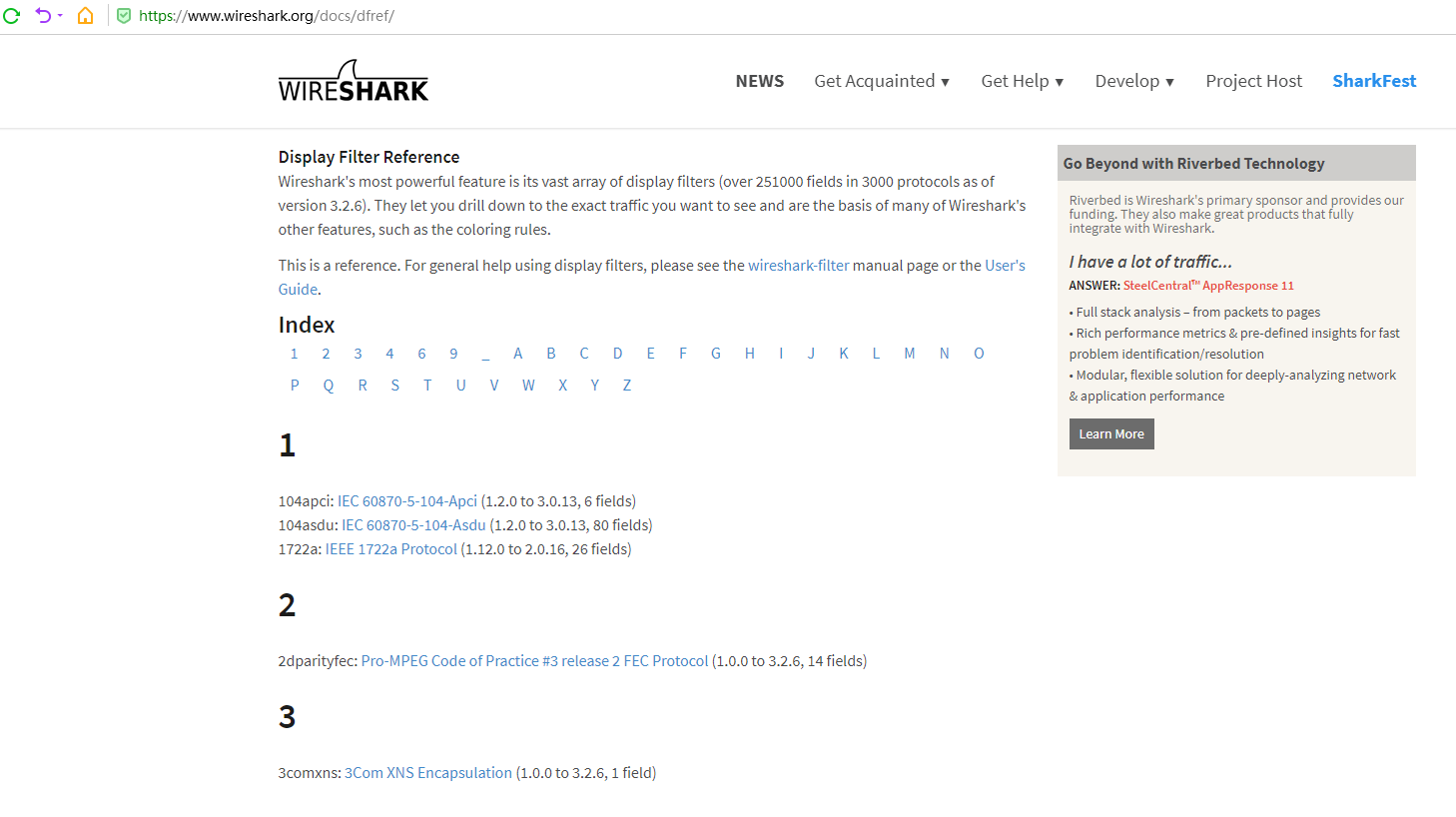
or 其中一个条件被满足

xor 有且仅有一个条件被满足

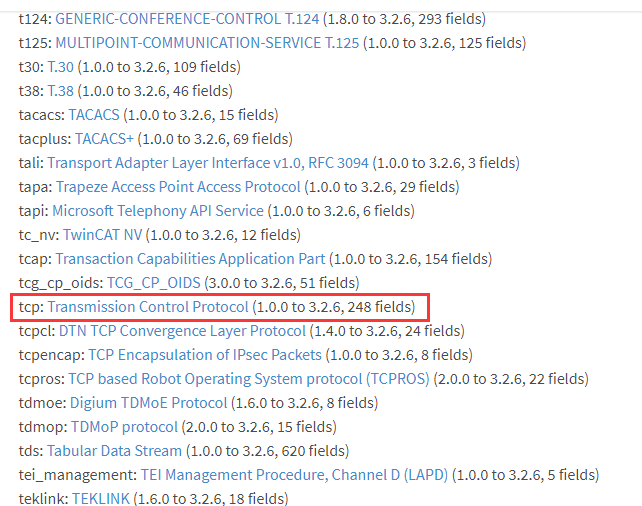
not 没有条件被满足

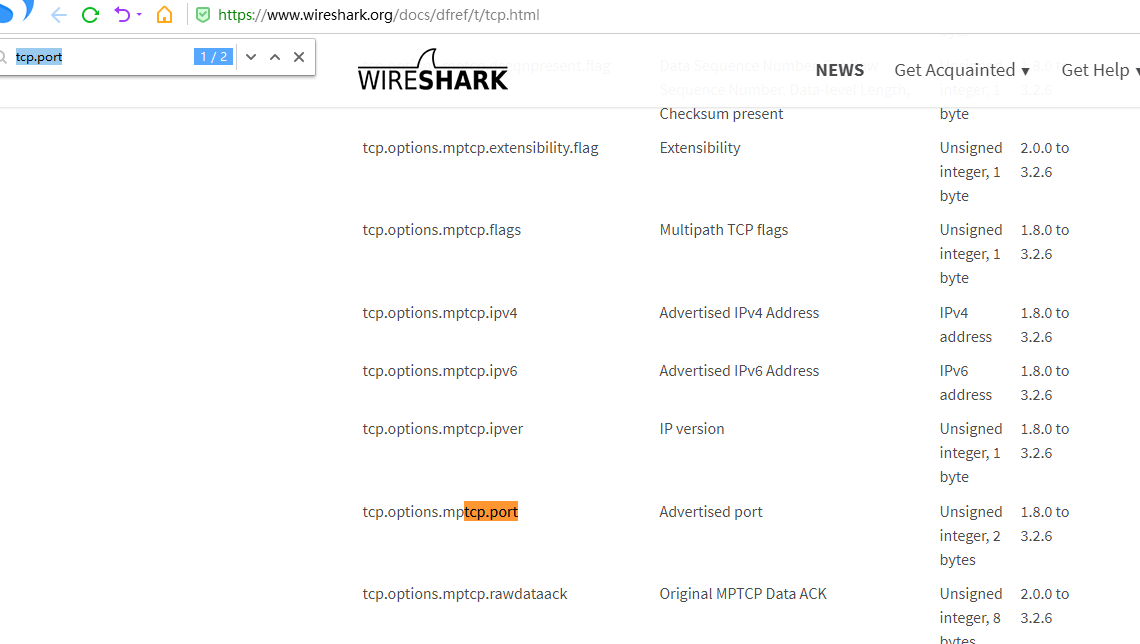
尽管在理论上编写过滤器表达式很简单，但针对不同问题创建过滤器时，依然需要许多特定的关键词与操作符。可以到Wireshark的官网查看：

<https://www.wireshark.org/docs/dfref>



比如tcp相关的，可以通过单击具体条目查看更多：





###### 预定义过滤器

过滤器可以事先设置好，不需要每次使用时都重新输入，捕获过滤器在菜单“捕获”-“捕获过滤器”中设置，显示过滤器在菜单“分析”-“Display Filters” 中设置。

#### 实战：用WireShark看看TCP的三次握手

##### 准备

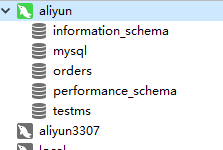
现在我们用平时比较常见的连接mysql服务器来看看，打开一个mysql的客户端，准备连接IP地址为47.112.44.148的云端mysql。

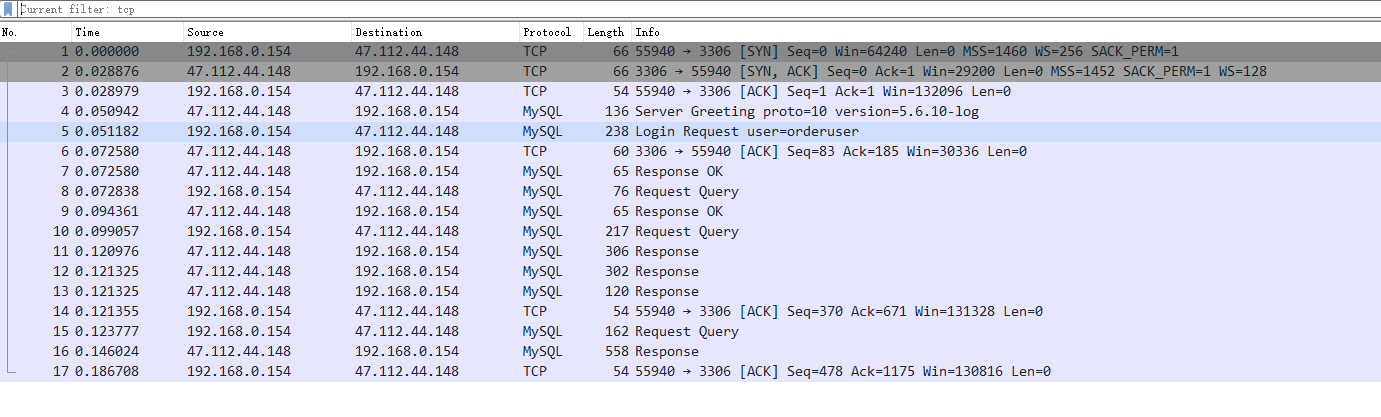
于是我们设定捕获过滤器，只捕获与IP地址47.112.44.148相关的数据包：

host 47.112.44.148

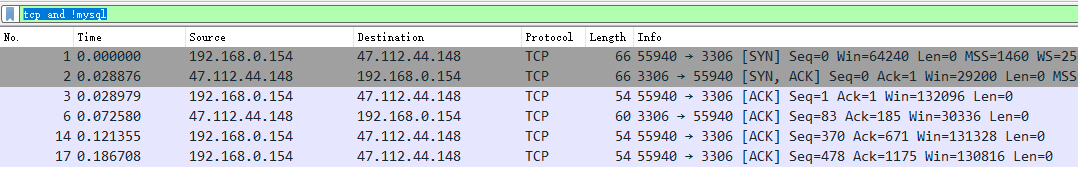


开始捕获，连接数据库：

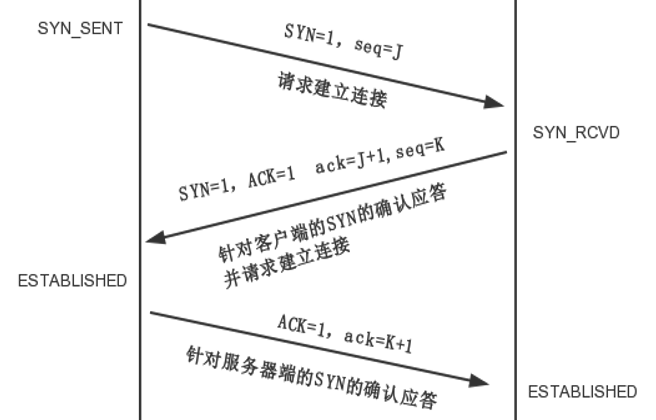




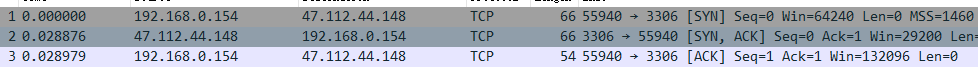
很明显，WireShark捕获到了本机所有和 47.112.44.148通信的报文，现在我们只需要观察TCP的三次握手和四次分手的报文，所以，我们在显示过滤器中写上tcp and !mysql



按照tcp三次握手的规则，三次握手包含一个SYN 包、 一个SYN/ACK 包和 一个ACK包。

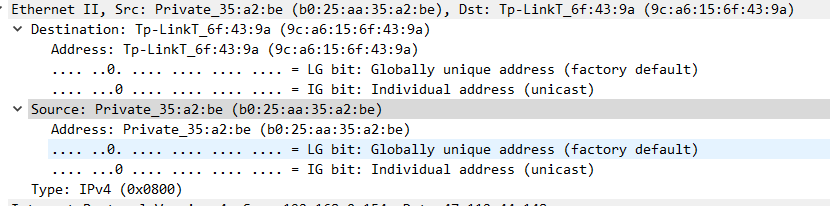


就是窗口里最上面的三条记录：

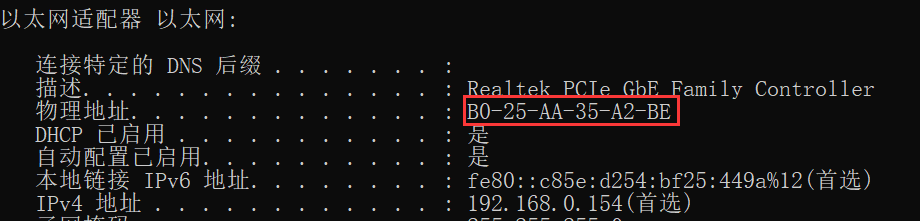


我们一条条看过来，

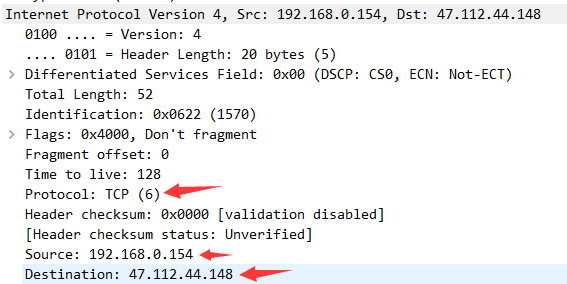
##### 第一次握手



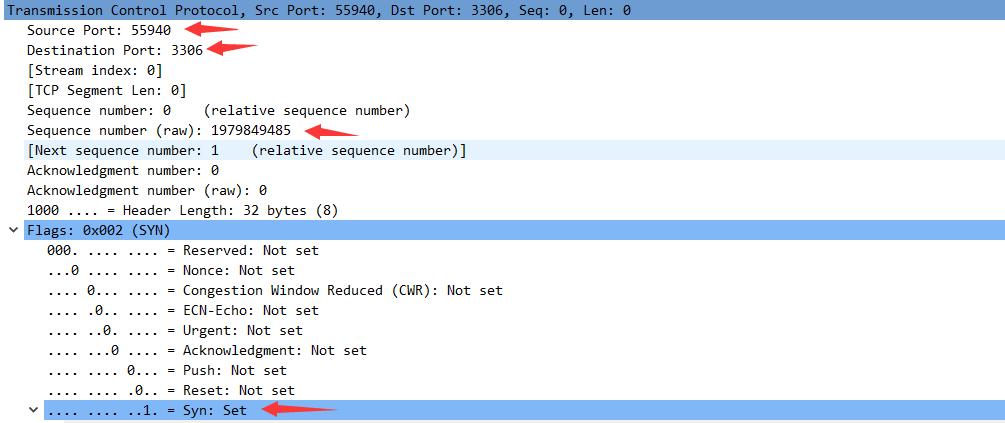
从这里我们可以看出，这是数据链路层相关的信息，source部分的地址和我们机器上的mac地址一模一样



接下来，就是IP层的相关信息，其中表明了它的上一层协议是TCP，同时本地和远程服务器的IP地址：

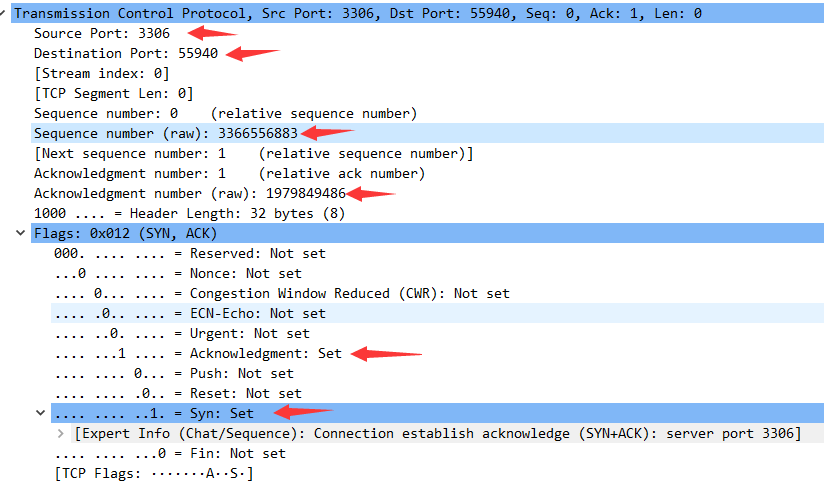


接下来，就是TCP层的相关信息，其中包括了本地端口和远程服务端口，既然是syn包，里面当然会带上seq值，本次通信是1979849485，tcp报文格式中的syn字段被设置为1，用来表明这是一个syn包。



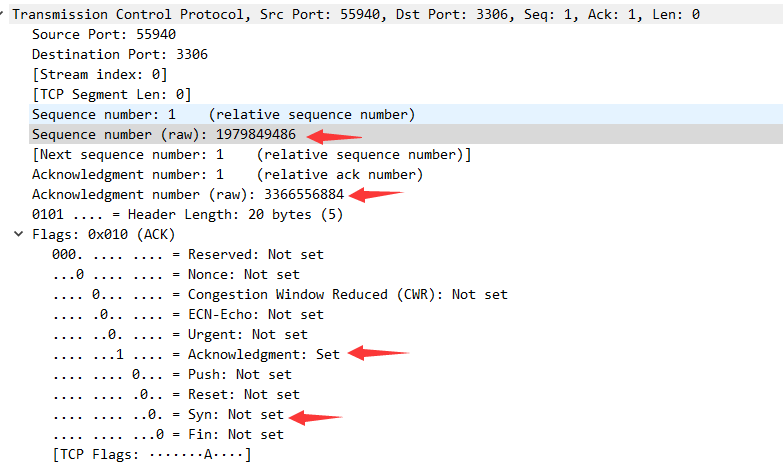
##### 第二次握手

数据链路层和IP层的报文我们不再查阅，直接看TCP层的报文，很明显，这是服务器给客户端的ACK报文，其中依然包括了远程服务端口和本地端口，同时服务器要把自己端的seq值告诉客户端，我们看到实际值是3366556883。同时服务器要把客户端传给自己的seq值做个应答确认，所以我们看见Acknowledgment number 字段值是 1979849486，刚好是第一次握手中，客户端传递给服务端的seq值1979849485加1。同时tcp报文格式中的syn字段被设置为1，Acknowledgment字段被设置为1，用来表明这是一个syn/ack包。



##### 第三次握手

这次握手的的报文分析，通过前两次的分析，相信同学们能够自行分析出来，这里就不在赘述。



### TCPDUMP

tcpdump可以抓所有层的数据，功能十分强大，tcpdump Linux作为网络服务器，特别是作为路由器和网关时，数据的采集和分析是不可少的。TcpDump是Linux中强大的网络数据采集分析工具之一。用简单的话来定义tcpdump，就是:dump the traffic on a network，根据使用者的定义对网络上的数据包进行截获的包分析工具。作为互联网上经典的的系统管理员或者运维人员必备工具，tcpdump以其强大的功能，灵活的截取第略，成为每个高级的系统管理员/运维人员分析网格，排查问题等所必备的工具之一。注意tcpdump必须以超级管理员的身份登录系统才能使用。

tcpdump的参数比较多：

tcp

各个参数的意义如下：

A：以ascii编码打印每个报文（不包括链路的头）。

a：将网络地址和广播地址转变成名字。

c：抓取指定数目的包。

C：用于判断用 -w 选项将报文写入的文件的大小是否超过这个值，如果超过了就新建文件（文件名后缀是1、2、3依次增加）；

d：将匹配信息包的代码以人们能够理解的汇编格式给出；

dd：将匹配信息包的代码以c语言程序段的格式给出；

ddd：将匹配信息包的代码以十进制的形式给出；

D：列出当前主机的所有网卡编号和名称，可以用于选项 -i；

e：在输出行打印出数据链路层的头部信息；

f：将外部的Internet地址以数字的形式打印出来；

F<表达文件>：从指定的文件中读取表达式,忽略其它的表达式；

i<网络界面>：监听主机的该网卡上的数据流，如果没有指定，就会使用最小网卡编号的网卡（在选项-D可知道，但是不包括环路接口），linux 2.2 内核及之后的版本支持 any 网卡，用于指代任意网卡；

l：如果没有使用 -w 选项，就可以将报文打印到 标准输出终端（此时这是默认）；

n：显示ip，而不是主机名；

nn：显示port，而不是服务名；

N：不列出域名；

O：不将数据包编码最佳化；

p：不让网络界面进入混杂模式；

q：快速输出，仅列出少数的传输协议信息；

r<数据包文件>：从指定的文件中读取包(这些包一般通过-w选项产生)；

s<数据包大小>：指定抓包显示一行的宽度，-s0表示可按包长显示完整的包，经常和-A一起用，默认截取长度为60个字节，但一般ethernet MTU都是1500字节。所以，要抓取大于60字节的包时，使用默认参数就会导致包数据丢失；

S：用绝对而非相对数值列出TCP关联数；

t：在输出的每一行不打印时间戳；

tt：在输出的每一行显示未经格式化的时间戳记；

T<数据包类型>：将监听到的包直接解释为指定的类型的报文，常见的类型有rpc （远程过程调用）和snmp（简单网络管理协议）；

v：输出一个稍微详细的信息，例如在ip包中可以包括ttl和服务类型的信息；

vv：输出详细的报文信息；

x/-xx/-X/-XX：以十六进制显示包内容，几个选项只有细微的差别，详见man手册；

w<数据包文件>：直接将包写入文件中，并不分析和打印出来；

expression：用于筛选的逻辑表达式。manip

在tcpdump中，tcpdump利用正则表达式作为过滤报文的条件，如果一个报文满足表达式的条件，则这个报文将会被捕获。如果没有给出任何条件，则网络上所有的信息包将会被截获。

在表达式中一般如下几种类型的关键字，一种是关于类型的关键字，主要包括host， port, 例如 host 210.27.48.2，指明 210.27.48.2是一台主机，port 23 指明端口号是23。如果没有指定类型，缺省的类型是host。

第二种是确定传输方向的关键字，主要包括src , dst这些关键字指明了传输的方向。举例说明，src 210.27.48.2 ,指明ip包中源地址是210.27.

48.2 , dst net 202.0.0.0 指明目的网络地址是202.0.0.0 。如果没有指明方向关键字，则缺省是src or dst关键字。

第三种是协议的关键字，主要包括fddi,ip ,arp,rarp,tcp,udp等类型。

如果没有指定任何协议，则tcpdump将会监听所有协议的信息包。

表达式还可以通过

! or not

&& or and

|| or or

进行连接

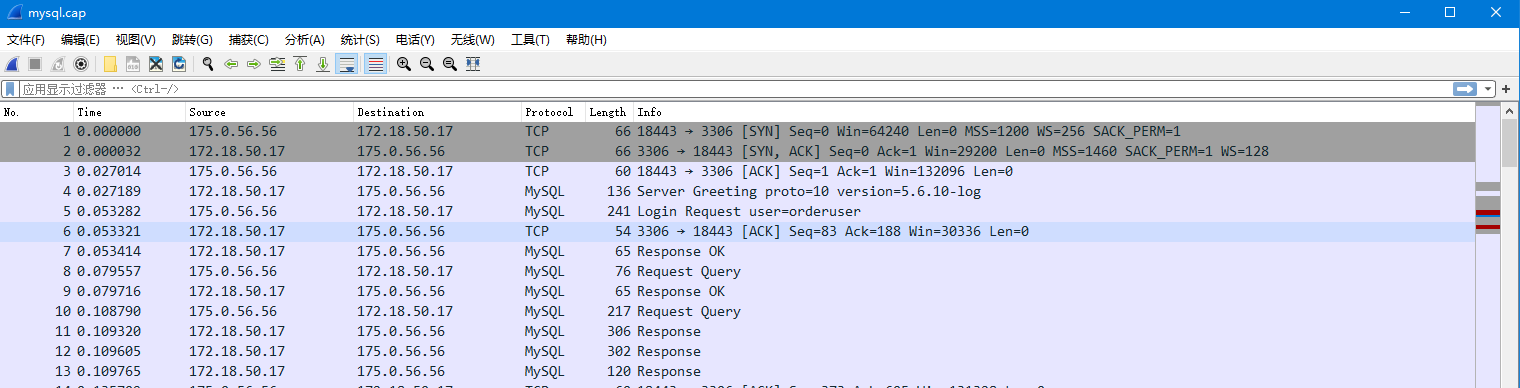
比如，

tcpdump -i eth0 tcp port 3306 -w ./mysql.cap

捕获本机网络设备eth0上tcp协议，端口3306的数据包，并写入当前目录下的mysql.cap文件。



不过tcpdump的输出不够直观，所以我们一般用tcpdump抓包以后下载到本地用Wireshark打开分析。



## 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。官方协议网站：https://tools.ietf.org/html/rfc2608

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

6.参数部分：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。

7.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

### 一次完整http请求的过程

1、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），首先会搜索浏览器自身的DNS缓存（缓存时间比较短，大概只有1分钟，且只能容纳1000条缓存）

b）如果浏览器自身的缓存里面没有找到，那么浏览器会搜索系统自身的DNS缓存

c）如果还没有找到，那么尝试从 hosts文件里面去找

d）在前面三个过程都没获取到的情况下，就去域名服务器去查找，

2、三次握手建立 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，HTTP连接是通过 TCP 来完成的。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；

3、客户端发起HTTP请求

4、服务器响应HTTP请求

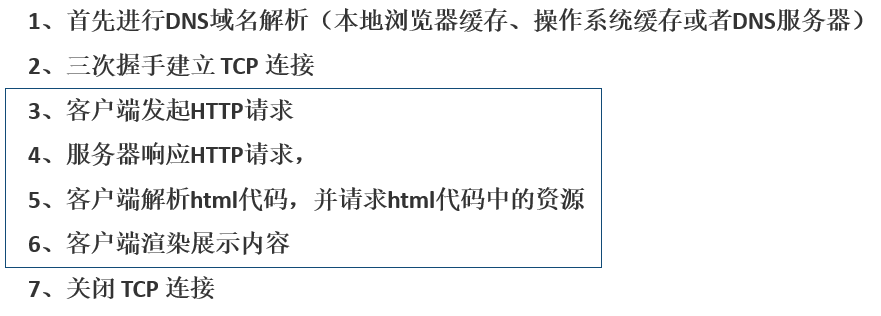
5、客户端解析html代码，并请求html代码中的资源

浏览器拿到html文件后，就开始解析其中的html代码，遇到js/css/image等静态资源时，就向服务器端去请求下载

6、客户端渲染展示内容

7、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，也就是说前面的3到6，可以反复进行。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。



### DNS劫持和HTTP劫持

DNS是Domain Name System的简写，即域名系统，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网。简单地说，如果我们想访问DNS.COM的官网，本来需要输入网站主机的IP地址，但是DNS可以将www.dns.com 解析成对应的IP地址，我们就不需要记住复杂的IP地址了。

DNS劫持就是通过劫持了DNS服务器，通过某些手段取得某域名的解析记录控制权，进而修改此域名的解析结果，导致对该域名的访问由原IP地址转入到修改后的指定IP，其结果就是对特定的网址不能访问或访问的是假网址，从而实现窃取资料或者破坏原有正常服务的目的。DNS劫持通过篡改DNS服务器上的数据返回给用户一个错误的查询结果来实现的。

一般来说DNS劫持有这三种情况：

1.错误域名解析到纠错导航页面，导航页面存在广告。判断方法:访问的域名是错误的，而且跳转的导航页面也是官方的，如电信的114，联通移网域名纠错导航页面。

2.错误域名解析到非正常页面，对错误的域名解析到导航页的基础上，有一定几率解析到一些恶意站点，这些恶意站点通过判断你访问的目标HOST、URI、 referrer等来确定是否跳转广告页面，这种情况就有可能导致跳转广告页面（域名输错）或者访问页面被加广告（页面加载时有些元素的域名错误而触发）这种劫持会对用户访问的目标HOST、URI、 referrer等会进行判定来确定是否解析恶意站点地址，不易被发现。

3.直接将特点站点解析到恶意或者广告页面，这种情况比较恶劣，而且出现这种情况未必就是运营商所为，家里路由器被黑，或者系统被入侵，甚至运营商的某些节点被第三方恶意控制都有可能。

DNS劫持常见于使用自动的DNS地址，所以，不管有没有被劫持，尽量不要使用运营商默认的DNS。

HTTP劫持：

在运营商的路由器节点上，设置协议检测，一旦发现是HTTP请求，而且是html类型请求，则拦截处理。后续做法往往分为2种，1种是类似DNS劫持返回302让用户浏览器跳转到另外的地址，还有1种是在服务器返回的HTML数据中插入js或dom节点（广告）。比如访问GitHub，出现了“我是渣渣辉，是兄弟就来砍我”之类的小弹窗。

有两种HTTP劫持防劫持思路

1.采用https协议加密请求。但是https多了ssl握手的过程，会耗费一定的时间和性能。

2.隐藏http请求的特征，例如使用对称加密算法加密整个url。市面上的各大手机应用商店都采用了这种做法。相较于https，url加密的兼容性更好，并且对CDN节点带来的解密额外负担可以忽略，但是url加密需要CDN节点的密切配合。

# 2、Java原生网络编程

## 一些常见术语

**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  
   传统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。

**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，下次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。

 总之，长连接和短连接的选择要视情况而定。

### 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容纳网络服务用的，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。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。

## 原生JDK网络编程BIO

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，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，同时数据的读取写入也必须阻塞在一个线程内等待其完成。

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

## 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）。

### NIO三大核心组件

NIO有三大核心组件：Selector选择器、Channel管道、buffer缓冲区。

#### 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。

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(Selectors)，然后使用一个单独的线程来操作这个选择器，进而“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。

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

后面会详细讲到其中的api等相关内容。



### 重要概念SelectionKey

#### 什么是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。SelectionKey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.

#### SelectionKey类型和就绪条件

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。

|  |  |
| --- | --- |
| **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。 |

#### 服务端和客户端分别感兴趣的类型

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。

### 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表示当前能写的位置。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，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。

##### 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。在比如：

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。

##### 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，再比如

get()属于相对读，从position位置读取一个byte，并将position+1，为下次读写作准备;

##### 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，准备下一次的写入

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。

##### 其他常用操作

**绝对读写**

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的值。

get(int index)属于绝对读，读取byteBuffer底层的bytes中下标为index的byte，不改变position。

更多Buffer实现的细节参考JavaDoc。

**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：

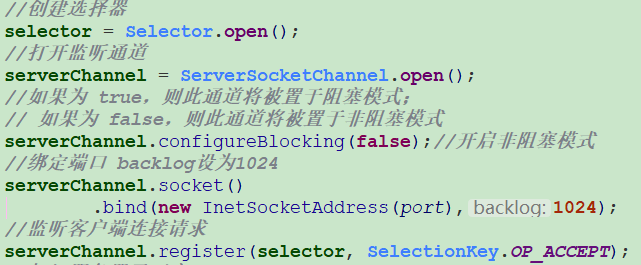
1. 第一个不相等的元素小于另一个Buffer中对应的元素 。
2. 所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

#### Buffer方法总结

|  |  |
| --- | --- |
| limit(), limit(10)等 | 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

### 原生JDK网络编程-NIO实战



1、

Selector对象是通过调用静态工厂方法open()来实例化的，如下：

Selector Selector=Selector.open()；

2、

要实现Selector管理Channel，需要将channel注册到相应的Selector上，如下：

channel.configureBlocking(false);

SelectionKey key= channel.register(selector,SelectionKey,OP\_READ);

通过调用通道的register()方法会将它注册到一个选择器上。与Selector一起使用时，Channel必须处于非阻塞模式下，否则将抛出IllegalBlockingModeException异常，这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式，而套接字通道都可以。另外通道一旦被注册，将不能再回到阻塞状态，此时若调用通道的configureBlocking(true)将抛出BlockingModeException异常。

register()方法的第二个参数是“interest集合”，表示选择器所关心的通道操作，它实际上是一个表示选择器在检查通道就绪状态时需要关心的操作的比特掩码。比如一个选择器对通道的read和write操作感兴趣，那么选择器在检查该通道时，只会检查通道的read和write操作是否已经处在就绪状态。

具体的操作类型和通道上能被支持的操作类型前面已经讲述过。

如果Selector对通道的多操作类型感兴趣，可以用“位或”操作符来实现：

int interestSet=SelectionKey.OP\_READ|SelectionKey.OP\_WRITE;

同时 一个 Channel 仅仅可以被注册到一个 Selector 一次, 如果将 Channel 注册到 Selector 多次, 那么其实就是相当于更新 SelectionKey 的 interest set。

通过SelectionKey可以判断Selector是否对Channel的某种事件感兴趣，比如

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = (interestSet & SelectionKey.OP\_ACCEPT) == SelectionKey.OP\_ACCEPT；

通过SelctionKey对象的readyOps()来获取相关通道已经就绪的操作。它是interest集合的子集，并且表示了interest集合中从上次调用select()以后已经就绪的那些操作。JAVA中定义几个方法用来检查这些操作是否就绪，比如selectionKey.isAcceptable();

同时，通过SelectionKey可以取出这个SelectionKey所关联的Selector和Channel。

如果我们要取消关联关系，怎么办？SelectionKey对象的cancel()方法来取消特定的注册关系。

在实际的应用中，我们还可以为SelectionKey绑定附加对象，在需要的时候取出。

SelectionKey key=channel.register(selector,SelectionKey.OP\_READ,theObject);

或selectionKey.attach(theObject);

取出这个附加对象，通过：

Object attachedObj = key.attachment();

3、

在实际运行中，我们通过Selector的select（）方法可以选择已经准备就绪的通道（这些通道包含你感兴趣的的事件）。

下面是Selector几个重载的select()方法：

select():阻塞到至少有一个通道在你注册的事件上就绪了。

select(long timeout)：和select()一样，但最长阻塞时间为timeout毫秒。

selectNow():非阻塞，立刻返回。

select()方法返回的int值表示有多少通道已经就绪,是自上次调用select()方法后有多少通道变成就绪状态。

一旦调用select()方法，并且返回值不为0时，则可以通过调用Selector的selectedKeys()方法来访问已选择键集合。

Set selectedKeys=selector.selectedKeys();

这个时候，循环遍历selectedKeys集中的每个键，并检测各个键所对应的通道的就绪事件，再通过SelectionKey关联的Selector和Channel进行实际的业务处理。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除，否则的话，下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

**具体与NIO编程相关的代码参见模块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，有事件来了，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，但是路人甲去的比较早，就告诉接待小姐，等10000技师上班了或者是空闲了，通知我。等路人甲接到接待小姐通知，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，路人甲再次接到接待小姐通知，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技师上班了”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。

前台的接待小姐不仅仅服务路人甲1人，他还可以同时服务路人乙、丙……..，每个人所感兴趣的事件是不一样的，前台的接待小姐会根据每个人感兴趣的事件通知对应的每个人。

### 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)。

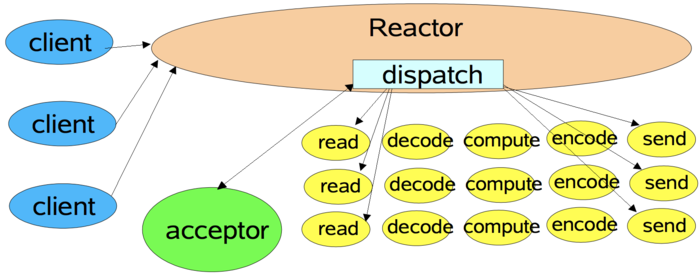
② 客户端向服务器端发起一个连接请求，Reactor监听到了该ACCEPT事件的发生并将该ACCEPT事件派发给相应的Acceptor处理器来进行处理。Acceptor处理器通过accept()方法得到与这个客户端对应的连接(SocketChannel)，然后将该连接所关注的READ事件以及对应的READ事件处理器注册到Reactor中，这样一来Reactor就会监听该连接的READ事件了。

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。

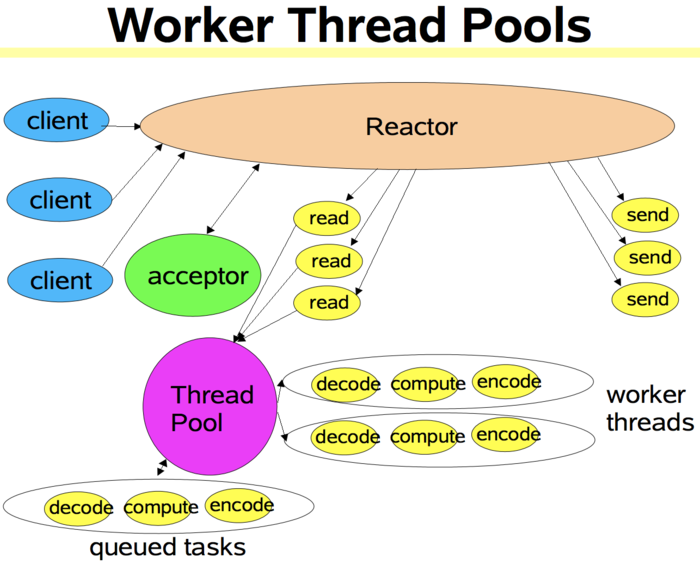
③ 通过适当调整线程池的大小，可以创建足够多的线程以便使处理器保持忙碌状态。同时还可以防止过多线程相互竞争资源而使应用程序耗尽内存或失败。

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。

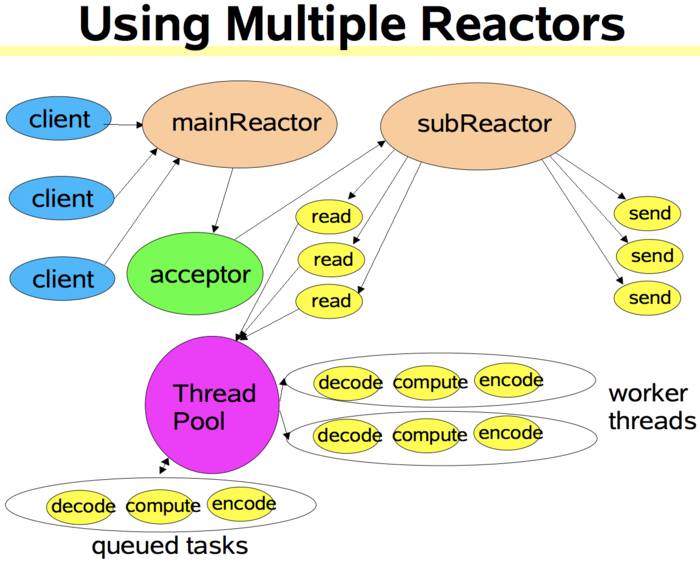
② 客户端向服务器端发起一个连接请求，mainReactor监听到了该ACCEPT事件并将该ACCEPT事件派发给Acceptor处理器来进行处理。Acceptor处理器通过accept()方法得到与这个客户端对应的连接(SocketChannel)，然后将这个SocketChannel传递给subReactor线程池。

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。

****

### 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。

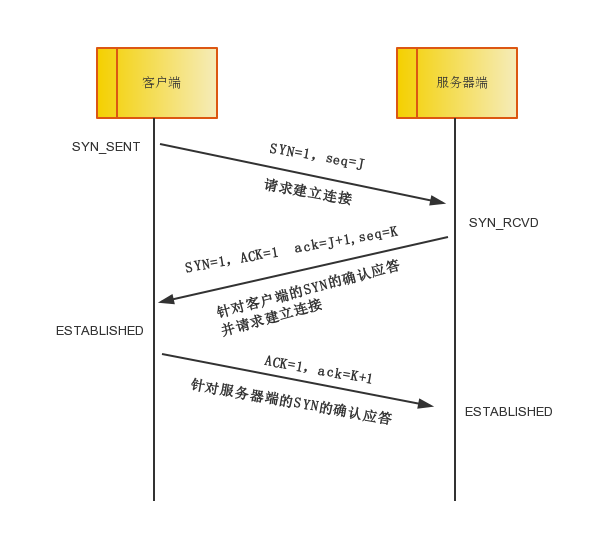
reactor模式与观察者模式有点像。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。当一个主体发生改变时，所有依属体都得到通知。

# 5、网络协议常见面试题汇总

## TCP

### 说一下TCP的三次握手过程

参见笔记



### 为什么TCP握手需要三次?

TCP是可靠的传输控制协议，而三次握手是保证数据可靠传输又能提高传输效率的最小次数。为什么？RFC793，也就是TCP的协议RFC中就谈到了原因，这是因为：

为了实现可靠数据传输， TCP协议的通信双方，都必须维护一个序列号， 以标识发送出去的数据包中，哪些是已经被对方收到的。

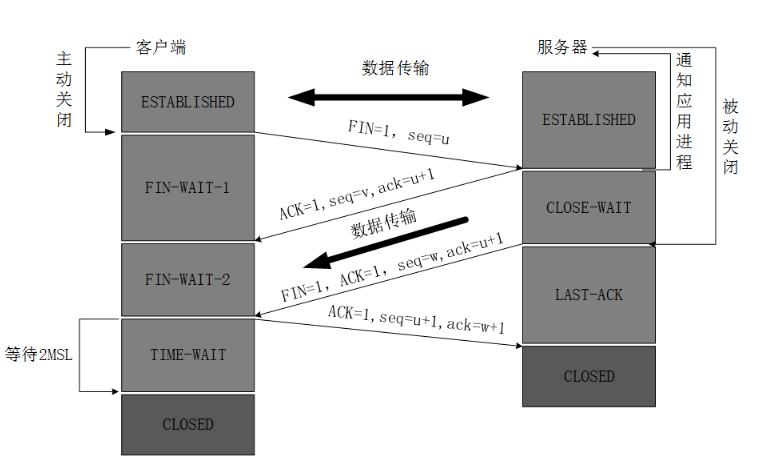
举例说明：发送方在发送数据包（假设大小为 10 byte）时， 同时送上一个序号( 假设为 500)，那么接收方收到这个数据包以后， 就可以回复一个确认号（510 = 500 + 10） 告诉发送方 “我已经收到了你的数据包， 你可以发送下一个数据包， 序号从 511 开始” 。

三次握手的过程即是通信双方**相互告知**序列号起始值，并**确认对方**已经收到了序列号起始值的必经步骤。

如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认。

至于为什么不是四次，很明显，三次握手后，通信的双方都已经知道了对方序列号起始值，也确认了对方知道自己序列号起始值，第四次握手已经毫无必要了。

### 解释一下TCP的四次挥手



### 为什么要有TIME\_WAIT状态？

TIME\_WAIT状态存在有两个原因。

<1>可靠终止TCP连接。如果最后一个ACK报文因为网络原因被丢弃，此时server因为没有收到ACK而超时重传FIN报文，处于TIME\_WAIT状态的client可以继续对FIN报文做回复，向server发送ACK报文。

<2>保证让迟来的TCP报文段有足够的时间被识别和丢弃。连接结束了，网络中的延迟报文也应该被丢弃掉，以免影响立刻建立的新连接。

### 为什么TCP的挥手需要四次？

TCP是全双工的连接，必须两端同时关闭连接，连接才算真正关闭。

如果一方已经准备关闭写，但是它还可以读另一方发送的数据。发送给FIN结束报文给对方对方收到后，回复ACK报文。当这方也已经写完了准备关闭，发送FIN报文，对方回复ACK。两端都关闭，TCP连接正常关闭。

### DDOS攻击

DDOS攻击利用合理的服务请求占用过多的服务资源，使正常用户的请求无法得到相应。

常见的DDOS攻击有计算机网络带宽攻击和连通性攻击。

带宽攻击指以极大的通信量冲击网络，使得所有可用网络资源都被消耗殆尽，最后导致合法的用户请求无法通过。

连通性攻击指用大量的连接请求冲击计算机，使得所有可用的操作系统资源都被消耗殆尽，最终计算机无法再处理合法用户的请求。

### SYN洪水攻击

SYN洪水攻击属于DDOS攻击的一种，它利用TCP协议缺陷，通过发送大量的半连接请求，耗费CPU和内存资源。

客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送SYN报文，服务器回复ACK确认报文，并等待客户的确认，由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN报文被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。

### 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，比如UDT。

常见的使用udp的例子：ICQ,QQ的聊天模块、DNS等等。

### 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。

### BIO、NIO和AIO的区别？

BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进行处理。线程开销大。

伪异步IO：将请求连接放入线程池，一对多，但线程还是很宝贵的资源。

NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO：一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO是面向流的，NIO是面向缓冲区的；BIO的各种流是阻塞的。而NIO是非阻塞的；BIO的Stream是单向的，而NIO的channel是双向的。

NIO的特点：事件驱动模型、单线程处理多任务、非阻塞I/O，I/O读写不再阻塞，而是返回0、基于block的传输比基于流的传输更高效、更高级的IO函数zero-copy、IO多路复用大大提高了Java网络应用的可伸缩性和实用性。基于Reactor线程模型。

在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生，事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。如在Reactor中实现读：注册读就绪事件和相应的事件处理器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

### 请概述NIO

Buffer：与Channel进行交互，数据是从Channel读入缓冲区，从缓冲区写入Channel中的

flip方法 ： 反转此缓冲区，将position给limit，然后将position置为0，其实就是切换读写模式

clear方法 ：清除此缓冲区，将position置为0，把capacity的值给limit。

rewind方法 ： 重绕此缓冲区，将position置为0

DirectByteBuffer可减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer，由JVM进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只能与Buffer 进行交互。通过源码可知，FileChannel的read方法和write方法都导致数据复制了两次！

Selector可使一个单独的线程管理多个Channel，open方法可创建Selector，register方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、accept。注册事件后会产生一个SelectionKey：它表示SelectableChannel 和Selector 之间的注册关系，wakeup方法：使尚未返回的第一个选择操作立即返回，唤醒的原因是：注册了新的channel或者事件；channel关闭，取消注册；优先级更高的事件触发（如定时器事件），希望及时处理。

NIO的服务端建立过程：Selector.open()：打开一个Selector；ServerSocketChannel.open()：创建服务端的Channel；bind()：绑定到某个端口上。并配置非阻塞模式；register()：注册Channel和关注的事件到Selector上；select()轮询拿到已经就绪的事件

### HTTP1.0和HTTP1.1的区别

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上和网络请求上，而HTTP1.1则在1999年才开始广泛应用于现在的各大浏览器网络请求中，同时HTTP1.1也是当前使用最为广泛的HTTP协议。 主要区别主要体现在：

1、缓存处理，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。

2、带宽优化及网络连接的使用，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3、错误通知的管理，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

4、Host头处理，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。

5、长连接，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

### HTTP2.0和HTTP1.X相比的新特性

新的二进制格式（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。

多路复用（MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。

header压缩，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。

服务端推送（server push），HTTP2.0也具有server push功能。

### HTTP2.0的多路复用和HTTP1.X中的长连接复用有什么区别？

HTTP/1.\* 一次请求-响应，建立一个连接，用完关闭；每一个请求都要建立一个连接；

HTTP/1.1 Pipeling解决方式为，若干个请求排队串行化单线程处理，后面的请求等待前面请求的返回才能获得执行机会，一旦有某请求超时等，后续请求只能被阻塞，毫无办法，也就是人们常说的线头阻塞；

HTTP/2多个请求可同时在一个连接上并行执行。某个请求任务耗时严重，不会影响到其它连接的正常执行；

### Http与Https的区别：

HTTPS 协议（HyperText Transfer Protocol over Secure Socket Layer）：一般理解为HTTP+SSL/TLS，通过 SSL证书来验证服务器的身份，并为浏览器和服务器之间的通信进行加密。

HTTP 的URL 以http:// 开头，而HTTPS 的URL 以https:// 开头

HTTP 是不安全的，而 HTTPS 是安全的

HTTP 标准端口是80 ，而 HTTPS 的标准端口是443

在OSI 网络模型中，HTTP工作于应用层，而HTTPS 的安全传输机制工作在传输层

HTTP 无法加密，而HTTPS 对传输的数据进行加密

HTTP无需证书，而HTTPS 需要CA机构颁发的SSL证书

### HTTPS方式与Web服务器通信时的步骤

　　（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

　　（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含服务器公钥<非对称加密>）传送一份给客户端。(HTTPS中，服务端将公钥发给数字证书认证机构进行安全认证并对公钥进行数字签名，完成后公钥和签名组合成数字证书。在和客户端通信时，服务端将数字证书发给客户端，客户端通过第三方安全认证机构（一般会在浏览器开发时，内置在浏览器中）对数字证书上的签名进行验证。)

　　（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

　　（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥<对称加密>，然后利用服务器公钥将会话密钥加密，并传送给网站。

　　（5）Web服务器利用自己的私钥解密出会话密钥。

　　（6）Web服务器利用会话密钥加密与客户端之间的通信。

### 什么是Http协议无状态协议?怎么解决?

无状态协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息

也就是说，当客户端一次HTTP请求完成以后，客户端再发送一次HTTP请求，HTTP并不知道当前客户端是一个”老用户“。

可以使用Cookie来解决无状态的问题，Cookie就相当于一个通行证，第一次访问的时候给客户端发送一个Cookie，当客户端再次来的时候，拿着Cookie(通行证)，那么服务器就知道这个是”老用户“

### 一次完整的HTTP请求所经历的步骤

1、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），首先会搜索浏览器自身的DNS缓存（缓存时间比较短，大概只有1分钟，且只能容纳1000条缓存）

b）如果浏览器自身的缓存里面没有找到，那么浏览器会搜索系统自身的DNS缓存

c）如果还没有找到，那么尝试从 hosts文件里面去找

d）在前面三个过程都没获取到的情况下，就去域名服务器去查找，

2、三次握手建立 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，HTTP连接是通过 TCP 来完成的。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；

3、客户端发起HTTP请求

4、服务器响应HTTP请求

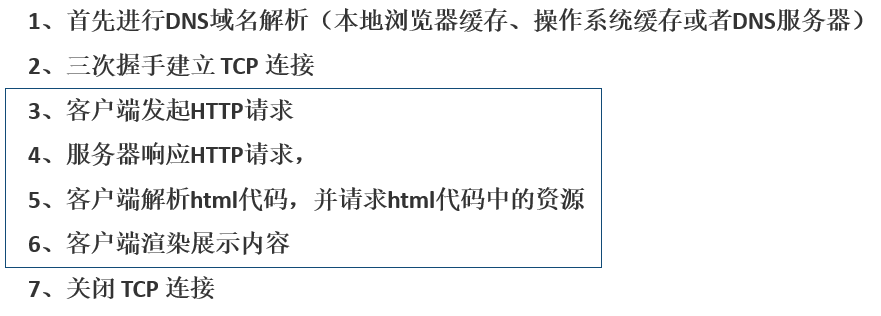
5、客户端解析html代码，并请求html代码中的资源

浏览器拿到html文件后，就开始解析其中的html代码，遇到js/css/image等静态资源时，就向服务器端去请求下载

6、客户端渲染展示内容

7、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，也就是说前面的3到6，可以反复进行。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。



### 常见的HTTP相应状态码

200：请求被正常处理

204：请求被受理但没有资源可以返回

206：客户端只是请求资源的一部分，服务器只对请求的部分资源执行GET方法，相应报文中通过Content-Range指定范围的资源。

301：永久性重定向

302：临时重定向

303：与302状态码有相似功能，只是它希望客户端在请求一个URI的时候，能通过GET方法重定向到另一个URI上

304：发送附带条件的请求时，条件不满足时返回，与重定向无关

307：临时重定向，与302类似，只是强制要求使用POST方法

400：请求报文语法有误，服务器无法识别

401：请求需要认证

403：请求的对应资源禁止被访问

404：服务器无法找到对应资源

500：服务器内部错误

503：服务器正忙

### 常用的HTTP方法有哪些？

GET： 用于请求访问已经被URI（统一资源标识符）识别的资源，可以通过URL传参给服务器

POST：用于传输信息给服务器，主要功能与GET方法类似，但一般推荐使用POST方式。

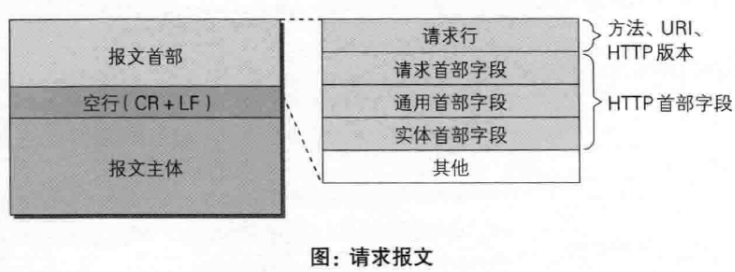
PUT： 传输文件，报文主体中包含文件内容，保存到对应URI位置。

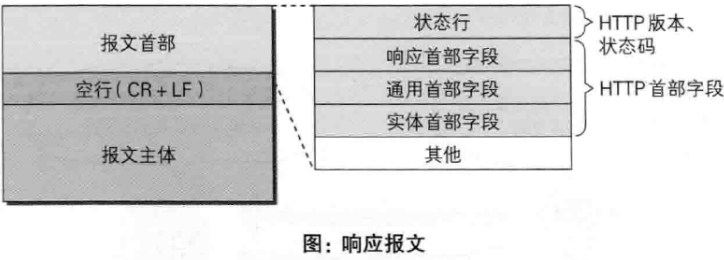
HEAD： 获得报文首部，与GET方法类似，只是不返回报文主体，一般用于验证URI是否有效。

DELETE：删除文件，与PUT方法相反，删除对应URI位置的文件。

OPTIONS：查询相应URI支持的HTTP方法

### HTTP请求报文与响应报文格式





### URI和URL的区别

URI，是uniform resource identifier，统一资源标识符，用来唯一的标识一个资源。

Web上可用的每种资源如HTML文档、图像、视频片段、程序等都是一个来URI来定位的

URI一般由三部组成：

①访问资源的命名机制

②存放资源的主机名

③资源自身的名称，由路径表示，着重强调于资源。

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。

URL是Internet上用来描述信息资源的字符串，主要用在各种WWW客户程序和服务器程序上，特别是著名的Mosaic。

采用URL可以用一种统一的格式来描述各种信息资源，包括文件、服务器的地址和目录等。URL一般由三部组成：

①协议(或称为服务方式)

②存有该资源的主机IP地址(有时也包括端口号)

③主机资源的具体地址。如目录和文件名等

URN，uniform resource name，统一资源命名，是通过名字来标识资源，比如mailto:java-net@java.sun.com。

URI是以一种抽象的，高层次概念定义统一资源标识，而URL和URN则是具体的资源标识的方式。URL和URN都是一种URI。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、news 和 isbn URI 都是 URN 的示例。

在Java的URI中，一个URI实例可以代表绝对的，也可以是相对的，只要它符合URI的语法规则。而URL类则不仅符合语义，还包含了定位该资源的信息，因此它不能是相对的。

在Java类库中，URI类不包含任何访问资源的方法，它唯一的作用就是解析。

相反的是，URL类可以打开一个到达资源的流。

### 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、安全漏洞较多。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

其它

protostuff 基于protobuf协议，但不需要配置proto文件，直接导包即可

Jboss marshaling 可以直接序列化java类， 无须实java.io.Serializable接口

Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级remoting onhttp工具

kryo 基于protobuf协议，只支持java语言,需要注册（Registration），然后序列化（Output），反序列化（Input）

### Netty是如何解决JDK中的Selector BUG的？

Selector BUG：JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有被根本解决。

这个问题的具体原因是：在部分Linux的2.6的kernel中，poll和epoll对于突然中断的连接socket会对返回的eventSet事件集合置为POLLHUP，也可能是POLLERR，eventSet事件集合发生了变化，这就可能导致Selector会被唤醒。这个时候selector的select方法，返回numKeys是0，所以下面本应该对key值进行遍历的事件处理根本执行不了，又回到最上面的while(true)循环，循环往复，不断的轮询，直到linux系统出现100%的CPU情况，最终导致程序崩溃。

修复的办法有两个：

一、将SelectKey去除掉，然后“刷新”一下Selector，刷新的方式也就是调用Selector.selectNow方法，这种修改仍然不是可靠的，一共有两点：

1.多个线程中的SelectionKey的key的cancel，很可能和下面的Selector.selectNow同时并发，如果是导致key的cancel后运行很可能没有效果

2.与其说第一点使得NIO空转出现的几率大大降低，经过Jetty服务器的测试报告发现，这种重复利用Selector并清空SelectionKey的改法很可能没有任何的效果。

二、创建一个新的Selector，这种处理方法要保险的多，基本上不会有任何的问题。

Netty采用就是这种解决办法：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。

### 如何让单机下Netty支持百万长连接？

单机下能不能让我们的网络应用支持百万连接？可以，但是有很多的工作要做。

#### 操作系统

首先就是要突破操作系统的限制。

在Linux平台上，无论编写客户端程序还是服务端程序，在进行高并发TCP连接处理时，最高的并发数量都要受到系统对用户单一进程同时可打开文件数量的限制（这是因为系统为每个TCP连接都要创建一个socket句柄，每个socket句柄同时也是一个文件句柄）。

可使用ulimit命令查看系统允许当前用户进程打开的文件数限制：

$ ulimit -n

1024

这表示当前用户的每个进程最多允许同时打开1024个文件，这1024个文件中还得除去每个进程必然打开的标准输入，标准输出，标准错误，服务器监听 socket,进程间通讯的unix域socket等文件，那么剩下的可用于客户端socket连接的文件数就只有大概1024-10=1014个左右。也就是说缺省情况下，基于Linux的通讯程序最多允许同时1014个TCP并发连接。

  对于想支持更高数量的TCP并发连接的通讯处理程序，就必须修改Linux对当前用户的进程同时打开的文件数量。

修改单个进程打开最大文件数限制的最简单的办法就是使用ulimit命令：

$ ulimit –n 1000000

如果系统回显类似于"Operation not permitted"之类的话，说明上述限制修改失败，实际上是因为在中指定的数值超过了Linux系统对该用户打开文件数的软限制或硬限制。因此，就需要修改Linux系统对用户的关于打开文件数的软限制和硬限制。

*软限制（soft limit）:是指Linux在当前系统能够承受的范围内进一步限制一个进程同时打开的文件数；*

*硬限制（hardlimit）:是根据系统硬件资源状况（主要是系统内存）计算出来的系统最多可同时打开的文件数量。*

第一步，修改/etc/security/limits.conf文件，在文件中添加如下行：

　　\* soft nofile 1000000

　　\* hard nofile 1000000

　　'\*'号表示修改所有用户的限制；

　　soft和hard为两种限制方式，其中soft表示警告的限制，hard表示真正限制，nofile表示打开的最大文件数。1000000则指定了想要修改的新的限制值，即最大打开文件数（请注意软限制值要小于或等于硬限制）。修改完后保存文件。

第二步，修改/etc/pam.d/login文件，在文件中添加如下行：

　　session required /lib/security/pam\_limits.so

这是告诉Linux在用户完成系统登录后，应该调用pam\_limits.so模块来设置系统对该用户可使用的各种资源数量的最大限制（包括用户可打开的最大文件数限制），而pam\_limits.so模块就会从/etc/security/limits.conf文件中读取配置来设置这些限制值。修改完后保存此文件。

第三步，查看Linux系统级的最大打开文件数限制，使用如下命令：

　　[speng@as4 ~]$ cat /proc/sys/fs/file-max

　　12158

　　这表明这台Linux系统最多允许同时打开（即包含所有用户打开文件数总和）12158个文件，是Linux系统级硬限制，所有用户级的打开文件数限制都不应超过这个数值。如果没有特殊需要，不应该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。

如何修改这个系统最大文件描述符的限制呢？修改sysctl.conf文件

vi /etc/sysctl.conf

# 在末尾添加

fs.file\_max = 1000000

# 立即生效

sysctl -p

#### JVM层面相关性能优化

当客户端的并发连接数达到数十万或者数百万时,系统一个较小的抖动就会导致很严重的后果,例如服务端的GC,导致应用暂停(STW)的GC持续几秒,就会导致海量的客户端设备掉线或者消息积压,一旦系统恢复,会有海量的设备接入或者海量的数据发送很可能瞬间就把服务端冲垮。

JVM层面的调优主要涉及GC参数优化,GC参数设置不当会导致频繁GC,甚至OOM异常,对服务端的稳定运行产生重大影响。

##### 1.确定GC优化目标

GC(垃圾收集)有三个主要指标。

(1)吞吐量:是评价GC能力的重要指标,在不考虑GC引起的停顿时间或内存消耗时,吞吐量是GC能支撑应用程序达到的最高性能指标。

(2)延迟:GC能力的最重要指标之一,是由于GC引起的停顿时间,优化目标是缩短延迟时间或完全消除停顿(STW),避免应用程序在运行过程中发生抖动。

(3)内存占用:GC正常时占用的内存量。

JVM GC调优的三个基本原则如下。

(1) Minor go回收原则:每次新生代GC回收尽可能多的内存,减少应用程序发生Full gc的频率。

2)GC内存最大化原则:垃圾收集器能够使用的内存越大,垃圾收集效率越高,应用程序运行也越流畅。但是过大的内存一次 Full go耗时可能较长,如果能够有效避免FullGC,就需要做精细化调优。

(3)3选2原则:吞吐量、延迟和内存占用不能兼得,无法同时做到吞吐量和暂停时间都最优,需要根据业务场景做选择。对于大多数应用,吞吐量优先,其次是延迟。当然对于时延敏感型的业务,需要调整次序。

##### 2.确定服务端内存占用

在优化GC之前,需要确定应用程序的内存占用大小,以便为应用程序设置合适的内存,提升GC效率。内存占用与活跃数据有关,活跃数据指的是应用程序稳定运行时长时间存活的Java对象。活跃数据的计算方式:通过GC日志采集GC数据,获取应用程序稳定时老年代占用的Java堆大小,以及永久代(元数据区)占用的Java堆大小,两者之和就是活跃数据的内存占用大小。

##### 3.GC优化过程

1、GC数据的采集和研读

2、设置合适的JVM堆大小

3、选择合适的垃圾回收器和回收策略

当然具体如何做，请参考JVM相关课程。而且GC调优会是一个需要多次调整的过程，期间不仅有参数的变化，更重要的是需要调整业务代码。

### 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 | 连接数基本上只受限于机器的内存大小 |

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。

### 什么是水平触发(LT)和边缘触发(ET)？

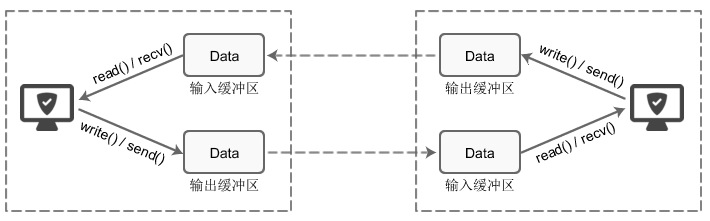
Level\_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll\_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

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。

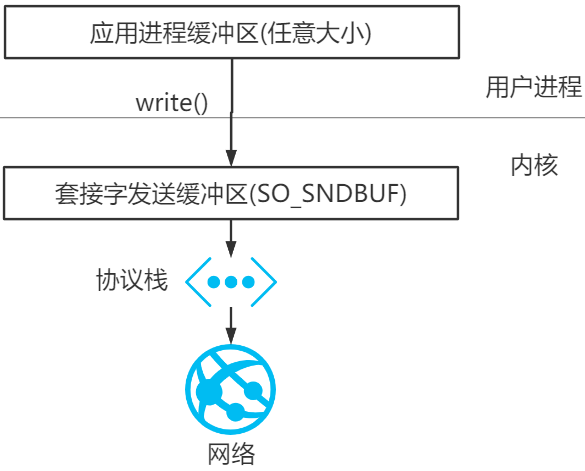
## 直接内存深入辨析

在所有的网络通信和应用程序中，每个TCP的Socket的内核中都有一个发送缓冲区(SO\_SNDBUF)和一个接收缓冲区(SO\_RECVBUF)，可以使用相关套接字选项来更改该缓冲区大小。



当某个应用进程调用write时，内核从该应用进程的缓冲区中复制所有数据到所写套接字的发送缓冲区。如果该套接字的发送缓冲区容不下该应用进程的所有数据(或是应用进程的缓冲区大于套接字的发送缓冲区，或是套接字的发送缓冲区中已有其他数据)，假设该套接字是阻塞的，则该应用进程将被投入睡眠。

内核将不从write系统调用返回，直到应用进程缓冲区中的所有数据都复制到套接字发送缓冲区。因此，从写一个TCP套接字的write调用成功返回仅仅表示我们可以重新使用原来的应用进程缓冲区，并不表明对端的TCP或应用进程已接收到数据。



Java程序自然也要遵守上述的规则。但在Java中存在着堆、垃圾回收等特性，所以在实际的IO中，在JVM内部的存在着这样一种机制：

在IO读写上，如果是使用堆内存，JDK会先创建一个DirectBuffer，再去执行真正的写操作。这是因为，当我们把一个地址通过JNI传递给底层的C库的时候，有一个基本的要求，就是这个地址上的内容不能失效。然而，在GC管理下的对象是会在Java堆中移动的。也就是说，有可能我把一个地址传给底层的write，但是这段内存却因为GC整理内存而失效了。所以必须要把待发送的数据放到一个GC管不着的地方。这就是调用native方法之前，数据—定要在堆外内存的原因。

可见，DirectBuffer并没有节省什么内存拷贝，只是因为HeapBuffer必须多做一次拷贝，使用DirectBuffer就会少一次内存拷贝。相比没有使用堆内存的Java程序，使用直接内存的Java程序当然更快一点。

从垃圾回收的角度而言，直接内存不受 GC(新生代的 Minor GC) 影响，只有当执行老年代的 Full GC 时候才会顺便回收直接内存，整理内存的压力也比数据放到HeapBuffer要小。

### 堆外内存的优点和缺点

堆外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。

## 零拷贝

### 什么是零拷贝?

零拷贝(英语: Zero-copy) 技术是指计算机执行操作时，CPU不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省CPU周期和内存带宽。

➢零拷贝技术可以减少数据拷贝和共享总线操作的次数，消除传输数据在存储器之间不必要的中间拷贝次数，从而有效地提高数据传输效率

➢零拷贝技术减少了用户进程地址空间和内核地址空间之间因为上下文切换而带来的开销

可以看出没有说不需要拷贝，只是说减少冗余[不必要]的拷贝。

下面这些组件、框架中均使用了零拷贝技术：Kafka、Netty、Rocketmq、Nginx、Apache。

### Linux的I/O机制与DMA

在早期计算机中，用户进程需要读取磁盘数据，需要CPU中断和CPU参与，因此效率比较低，发起IO请求，每次的IO中断，都带来CPU的上下文切换。因此出现了——DMA。

DMA(Direct Memory Access，直接内存存取) 是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于CPU 的大量中断负载。

DMA控制器，接管了数据读写请求，减少CPU的负担。这样一来，CPU能高效工作了。现代硬盘基本都支持DMA。

实际因此IO读取，涉及两个过程：

1、DMA等待数据准备好，把磁盘数据读取到操作系统内核缓冲区；

2、用户进程，将内核缓冲区的数据copy到用户空间。

这两个过程，都是阻塞的。

### 传统数据传送机制

比如：读取文件，再用socket发送出去，实际经过四次copy。

伪码实现如下：

buffer = File.read()

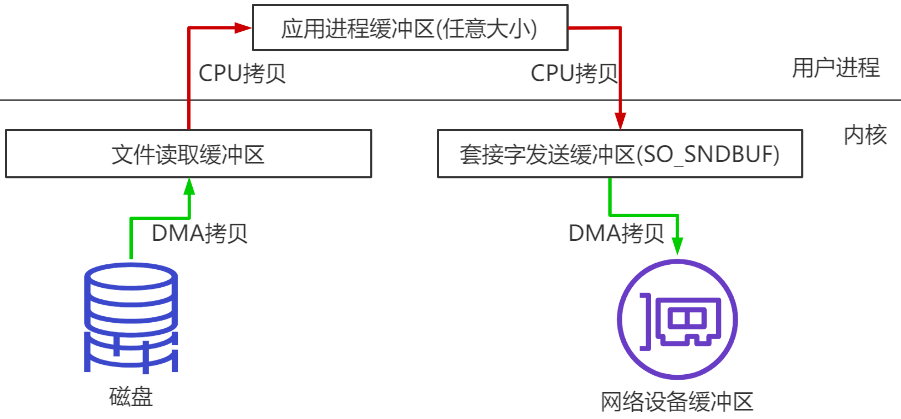
Socket.send(buffer)

1、第一次：将磁盘文件，读取到操作系统内核缓冲区；

2、第二次：将内核缓冲区的数据，copy到应用程序的buffer；

3、第三步：将application应用程序buffer中的数据，copy到socket网络发送缓冲区(属于操作系统内核的缓冲区)；

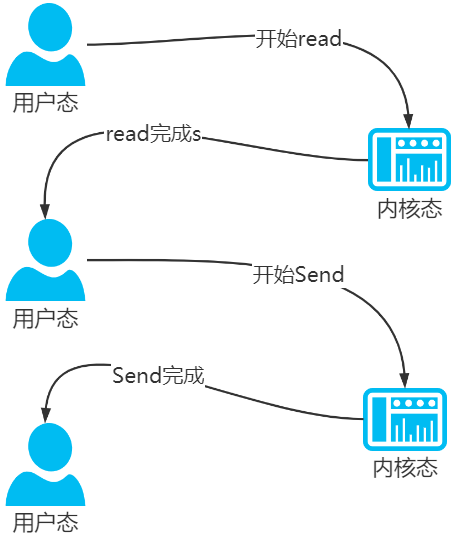
4、第四次：将socket buffer的数据，copy到网卡，由网卡进行网络传输。



分析上述的过程，虽然引入DMA来接管CPU的中断请求，但四次copy是存在“不必要的拷贝”的。实际上并不需要第二个和第三个数据副本。应用程序除了缓存数据并将其传输回套接字缓冲区之外什么都不做。相反，数据可以直接从读缓冲区传输到套接字缓冲区。

显然，第二次和第三次数据copy 其实在这种场景下没有什么帮助反而带来开销，这也正是零拷贝出现的背景和意义。

同时，read和send都属于系统调用，每次调用都牵涉到两次上下文切换：



总结下，传统的数据传送所消耗的成本：4次拷贝，4次上下文切换。

4次拷贝，其中两次是DMA copy，两次是CPU copy。

### Linux支持的(常见)零拷贝

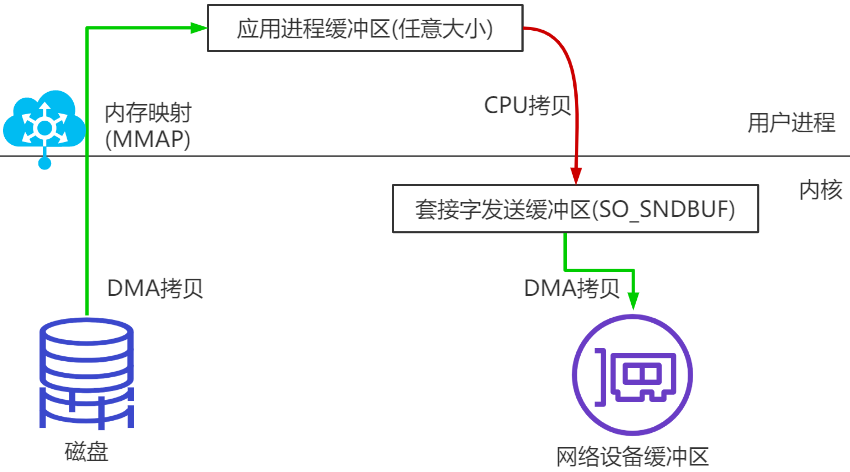
目的：减少IO流程中不必要的拷贝，当然零拷贝需要OS支持，也就是需要kernel暴露api。

#### mmap内存映射

硬盘上文件的位置和应用程序缓冲区(application buffers)进行映射（建立一种一一对应关系），由于mmap()将文件直接映射到用户空间，所以实际文件读取时根据这个映射关系，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝，不再有文件内容从硬盘拷贝到内核空间的一个缓冲区。

mmap内存映射将会经历：3次拷贝: 1次cpu copy，2次DMA copy；

以及4次上下文切换



#### sendfile

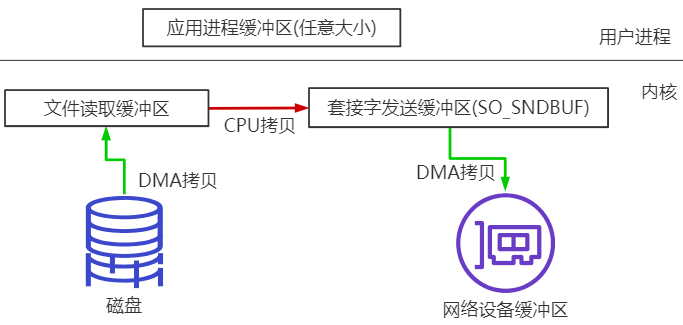
linux 2.1支持的sendfile

当调用sendfile()时，DMA将磁盘数据复制到kernel buffer，然后将内核中的kernel buffer直接拷贝到socket buffer。在硬件支持的情况下，甚至数据都并不需要被真正复制到socket关联的缓冲区内。取而代之的是，只有记录数据位置和长度的描述符被加入到socket缓冲区中，DMA模块将数据直接从内核缓冲区传递给协议引擎，从而消除了遗留的最后一次复制。

一旦数据全都拷贝到socket buffer，sendfile()系统调用将会return，代表数据转化的完成。socket buffer里的数据就能在网络传输了。

sendfile会经历：3次拷贝，1次CPU copy ，2次DMA copy；硬件支持的情况下，则是2次拷贝，0次CPU copy， 2次DMA copy。

以及2次上下文切换



#### splice

Linux 从2.6.17 支持splice

数据从磁盘读取到OS内核缓冲区后，在内核缓冲区直接可将其转成内核空间其他数据buffer，而不需要拷贝到用户空间。

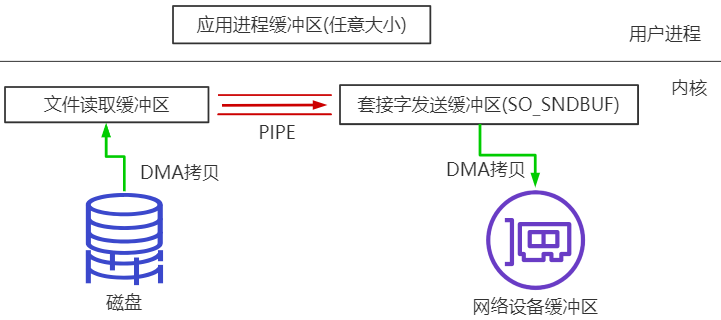
如下图所示，从磁盘读取到内核buffer后，在内核空间直接与socket buffer建立pipe管道。

和sendfile()不同的是，splice()不需要硬件支持。

注意splice和sendfile的不同，sendfile是将磁盘数据加载到kernel buffer后，需要一次CPU copy，拷贝到socket buffer。而splice是更进一步，连这个CPU copy也不需要了，直接将两个内核空间的buffer进行pipe。

splice会经历 2次拷贝: 0次cpu copy 2次DMA copy；

以及2次上下文切换



#### 总结Linux中零拷贝

最早的零拷贝定义，来源于

*Linux 2.4内核新增 sendfile 系统调用，提供了零拷贝。磁盘数据通过 DMA 拷贝到内核态 Buffer 后，直接通过 DMA 拷贝到 NIO Buffer(socket buffer)，无需 CPU 拷贝。这也是零拷贝这一说法的来源。这是真正操作系统 意义上的零拷贝(也就是狭义零拷贝)。*

但是我们知道，由OS内核提供的 操作系统意义上的零拷贝，发展到目前也并没有很多种，也就是这样的零拷贝并不是很多；

随着发展，零拷贝的概念得到了延伸，就是目前的减少不必要的数据拷贝都算作零拷贝的范畴。

### Java生态圈中的零拷贝

Linux提供的零拷贝技术 Java并不是全支持，支持2种(内存映射mmap、sendfile)；

#### NIO提供的内存映射 MappedByteBuffer

NIO中的FileChannel.map()方法其实就是采用了操作系统中的内存映射方式，底层就是调用Linux mmap()实现的。

将内核缓冲区的内存和用户缓冲区的内存做了一个地址映射。这种方式适合读取大文件，同时也能对文件内容进行更改，但是如果其后要通过SocketChannel发送，还是需要CPU进行数据的拷贝。

#### NIO提供的sendfile

Java NIO 中提供的 FileChannel 拥有 transferTo 和 transferFrom 两个方法，可直接把 FileChannel 中的数据拷贝到另外一个 Channel，或者直接把另外一个 Channel 中的数据拷贝到 FileChannel。该接口常被用于高效的网络 / 文件的数据传输和大文件拷贝。在操作系统支持的情况下，通过该方法传输数据并不需要将源数据从内核态拷贝到用户态，再从用户态拷贝到目标通道的内核态，同时也避免了两次用户态和内核态间的上下文切换，也即使用了“零拷贝”，所以其性能一般高于 Java IO 中提供的方法。