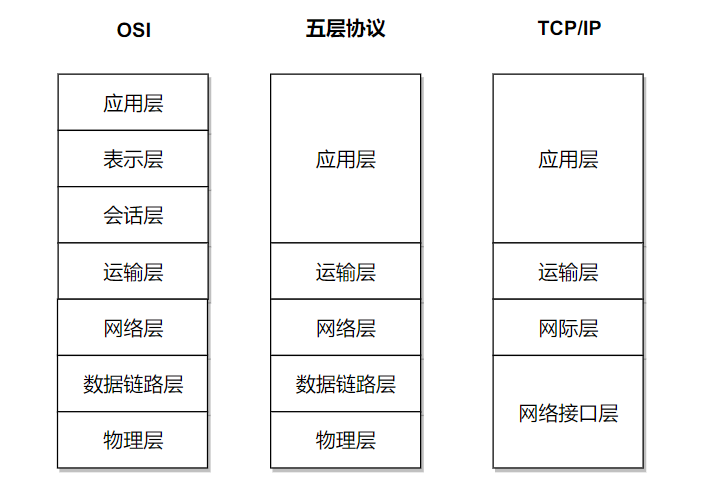
**计算机网络体系结构**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/7fa66f48-bc42-4b98-93c2-56b20feaa990.png)

**OSI**

其中表示层和会话层用途如下：

* **表示层** ：数据压缩、加密以及数据描述，这使得应用程序不必关心在各台主机中数据内部格式不同的问题。
* **会话层** ：建立及管理会话。

五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

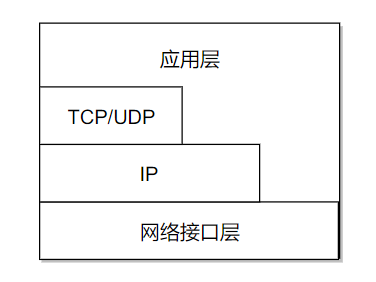
**五层协议**

* **应用层** ：为特定应用程序提供数据传输服务，例如 HTTP、DNS 等。数据单位为报文。
* **传输层** ：为进程提供通用数据传输服务。由于应用层协议很多，定义通用的传输层协议就可以支持不断增多的应用层协议。运输层包括两种协议：传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为报文段；用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。TCP 主要提供完整性服务，UDP 主要提供及时性服务。
* **网络层** ：为主机提供数据传输服务。而传输层协议是为主机中的进程提供数据传输服务。网络层把传输层传递下来的报文段或者用户数据报封装成分组。
* **数据链路层** ：网络层针对的还是主机之间的数据传输服务，而主机之间可以有很多链路，链路层协议就是为同一链路的主机提供数据传输服务。数据链路层把网络层传下来的分组封装成帧。
* **物理层** ：考虑的是怎样在传输媒体上传输数据比特流，而不是指具体的传输媒体。物理层的作用是尽可能屏蔽传输媒体和通信手段的差异，使数据链路层感觉不到这些差异。

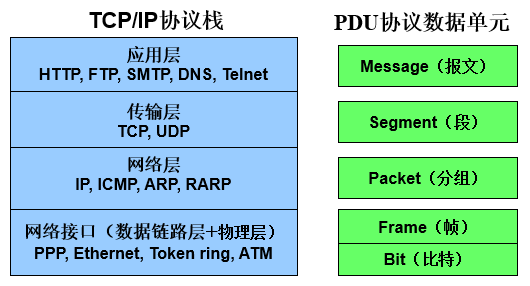
**TCP/IP**

它只有四层，相当于五层协议中数据链路层和物理层合并为网络接口层。

TCP/IP 体系结构不严格遵循 OSI 分层概念，应用层可能会直接使用 IP 层或者网络接口层。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/7048a730-a2d0-43d1-99ce-f774e4ef1b27.png)

TCP/IP 是 Internet上的标准通信协议集，该协议集由数十个具有层次结构的协议组成，其中 TCP 和 IP 是该协议集中的两个最重要的核心协议。TCP/IP协议族按层次可分为以下四层：应用层、传输层、网络层和网络接口层，各层对应的 PDU 数据单元的名称如下图所示。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/TCPIP%E4%B8%8EPSU.png)

**小结**

OSI 七层体系结构具有概念清楚、理论完整的特点，是一个理论上的国际标准，但却不是事实上的国际标准；而具有简单易用特点的 TCP/IP 四层体系结构则是事实上的标准。 需要指出的是，五层体系结构虽然综合了 OSI 和 TCP/IP 的优点，但其只是为了学术学习研究而提出的，没有具体的实际意义。

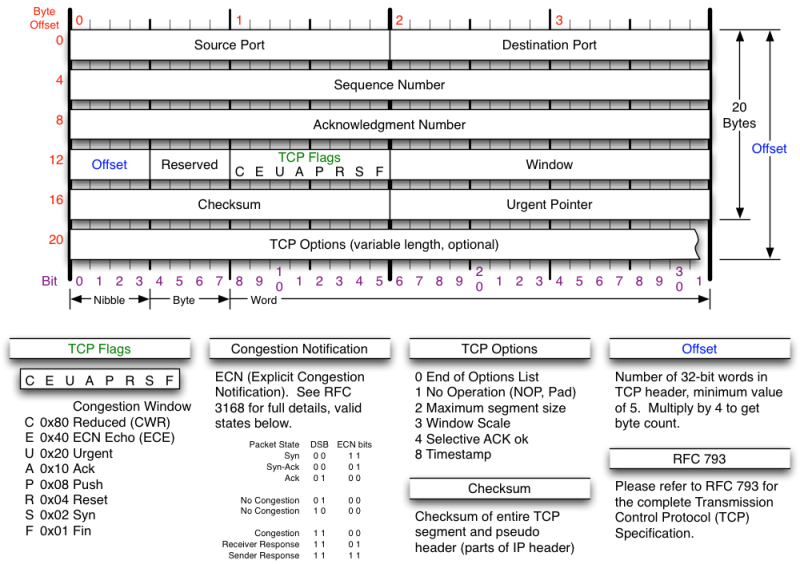
**说说 TCP 的三次握手**

这一到很常见的面试题。

**传输控制协议 TCP 简介**

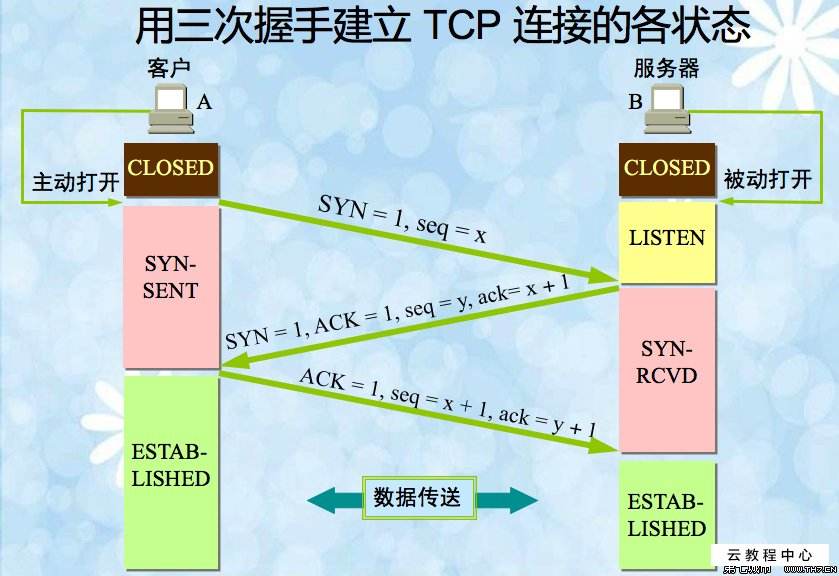
* 面向连接的、可靠的基于字节流的传输层通信协议
* 将应用层的数据流分割成报文段并发送给目标节点的 TCP 层
* 数据包都是由序号，对方收到则发送 ACK 确认，未收到则重传
* 使用校验和来检验数据在传输过程中是否有误

**TCP 报文头**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/MJB-TCP-Header-800x564.png)

* **源端口、目的端口** ：标记进程。
* **序号** ：用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。
* **确认号** ：期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
* **数据偏移** ：指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
* **连接标志（TCP Flags）**：表示控制功能，下面是常见的连接标志。
  + **确认 ACK** ：当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
  + **同步 SYN** ：在连接建立时用来同步序号。当 SYN=a1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
  + **终止 FIN** ：用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
* **窗口** ：窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。

**三次握手**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/timg.jpg)

在 TCP/IP 协议中， TCP 协议提供可靠的连接服务，采用三次握手建立一个连接。

假设 A 为客户端，B 为服务器端。

* 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
* A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
* B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
* A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
* B 收到 A 的确认后，连接建立。

**对于建链接的3次握手**

主要是要初始化Sequence Number 的初始值。通信的双方要互相通知对方自己的初始化的 Sequence Number（缩写为ISN：Inital Sequence Number）——所以叫SYN，全称Synchronize Sequence Numbers。也就上图中的 x 和 y。这个号要作为以后的数据通信的序号，以保证应用层接收到的数据不会因为网络上的传输的问题而乱序（TCP会用这个序号来拼接数据）。

**关于建连接时SYN超时**

试想一下，如果server端接到了 client 发的 SYN 后回了 SYN-ACK 后 client 掉线了，server 端没有收到 client 回来的ACK，那么，这个连接处于一个中间状态，即没成功，也没失败。于是，server 端如果在一定时间内没有收到的TCP会重发 SYN-ACK。在Linux下，默认重试次数为5次，重试的间隔时间从 1s 开始每次都翻售，5 次的重试时间间隔为1s, 2s, 4s, 8s, 16s，总共31s，第5次发出后还要等32s都知道第5次也超时了，所以，总共需要 1s + 2s + 4s+ 8s+ 16s + 32s = 2^6 -1 = 63s，TCP才会把断开这个连接。

**关于SYN Flood攻击**

一些恶意的人就为此制造了SYN Flood攻击，给服务器发了一个SYN后，就下线了，于是服务器需要默认等 63s 才会断开连接，这样，攻击者就可以把服务器的 syn 连接的队列耗尽，让正常的连接请求不能处理。于是，Linux下给了一个叫 **tcp\_syncookies** 的参数来应对这个事——当SYN队列满了后，TCP会通过源地址端口、目标地址端口和时间戳打造出一个特别的Sequence Number发回去（又叫cookie），如果是攻击者则不会有响应，如果是正常连接，则会把这个 SYN Cookie发回来，然后服务端可以通过cookie建连接（即使你不在SYN队列中）。

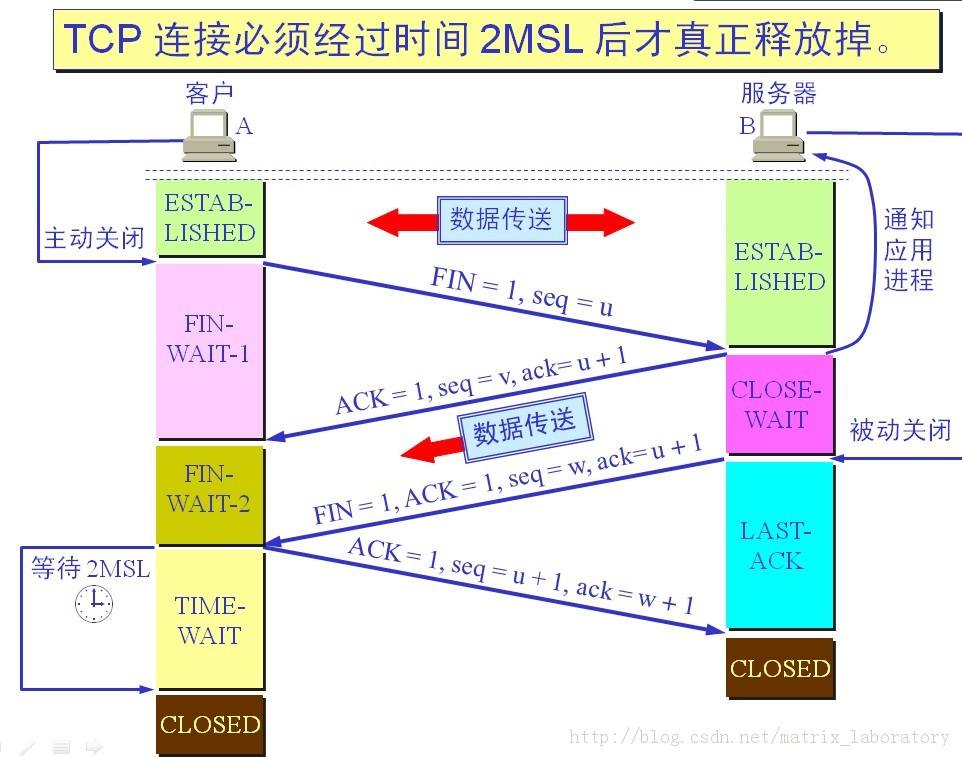
请注意，**请先千万别用 tcp\_syncookies 来处理正常的大负载的连接的情况**。因为，synccookies 是妥协版的TCP协议，并不严谨。对于正常的请求，你应该调整三个TCP参数可供你选择。

* 第一个是：tcp\_synack\_retries 可以用他来减少重试次数；
* 第二个是：tcp\_max\_syn\_backlog，可以增大SYN连接数；
* 第三个是：tcp\_abort\_on\_overflow 处理不过来干脆就直接拒绝连接了 ；

**保活机制**

* 向对方发送保活探测报文，如果未收到响应则继续发送
* 尝试次数达到保活探测数仍然未收到响应则中断连接

**谈谈四次挥手**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/tisfsmg.jpg)

以下描述不讨论序号和确认号，因为序号和确认号的规则比较简单。并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

* A 发送连接释放报文，FIN=1。
* B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
* 当 B 不再需要连接时，发送连接释放报文，FIN=1。
* A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。
* B 收到 A 的确认后释放连接。

**四次挥手的原因**

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TCP连接是全双工的，服务端可以发送数据到客户端，客户端也可以发送数据到服务端，发送方和接收方都需要两次挥手才能关闭 。

**TIME\_WAIT**

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

* 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
* 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

**TCP 和 UDP 的区别**

前面我们说了 TCP 现在我们来认识一下 UDP。

**UPD 的特点**

* 面向非连接的
* 不维护连接状态，支持同时向多个客户端传输相同的消息
* 数据包报头只有 8 个字节，额外开销较小
* 吞吐量只受限于数据生成率、传输速率以及机器性能
* 尽最大努力交付，不保证可靠交付，不需要维持复杂的链接状态表
* 面向报文，不对应用程序提交的报文信息进行拆分或则合并

**对比**

* TCP 是面向连接的；UDP 是无连接的。
* TCP 是可靠的；UDP 是不可靠的。
* TCP 只支持点对点通信；UDP 支持一对一、一对多、多对一、多对多的通信模式。
* TCP 是面向字节流的；UDP 是面向报文的。
* TCP 有拥塞控制机制；UDP 没有拥塞控制，适合媒体通信。
* TCP 首部开销(20 个字节)，比 UDP 的首部开销(8 个字节)要大。

**TCP 的滑动窗口**

首先明确：

TCP滑动窗口分为接受窗口，发送窗口。

滑动窗口协议是**传输层进行流控**的一种措施，**接收方通过通告发送方自己的窗口大小**，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。

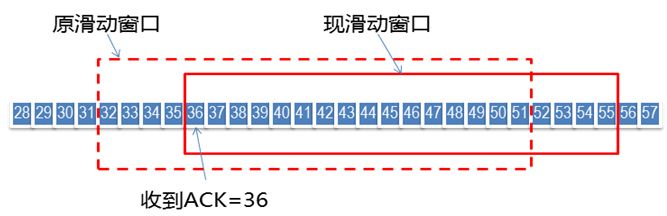
**重要概念**

对ACK的再认识，ack通常被理解为收到数据后给出的一个确认ACK，ACK包含两个非常重要的信息：

* 一是期望接收到的下一字节的序号n，该n代表接收方已经接收到了前n-1字节数据，此时如果接收方收到第n+1字节数据而不是第n字节数据，接收方是不会发送序号为n+2的ACK的。举个例子，假如接收端收到1-1024字节，它会发送一个确认号为1025的ACK,但是接下来收到的是2049-3072，它是不会发送确认号为3072的ACK,而依旧发送1025的ACK。
* 二是当前的窗口大小m，如此发送方在接收到ACK包含的这两个数据后就可以计算出还可以发送多少字节的数据给对方，假定当前发送方已发送到第x字节，则可以发送的字节数就是y=m-(x-n).这就是滑动窗口控制流量的基本原理

**重点：发送方根据收到ACK当中的期望收到的下一个字节的序号n以及窗口m，还有当前已经发送的字节序号x，算出还可以发送的字节数。**

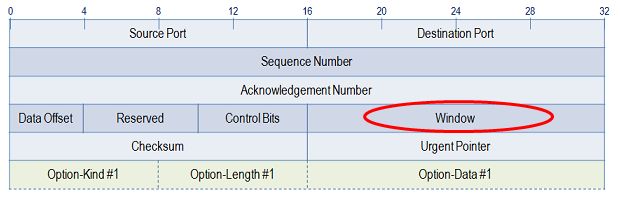
发送端窗口的第一个字节序号一定是ACK中期望收到的下一个字节序号，比如下图：

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/9c21786770459afa47bfa2e4606cc454_hd.jpg)

上图52 53 54 55 字节都是可以新发送的字节序。

接受端窗口的第一个字节序之前一定是已经完全接收的，后面窗口里面的数据都是希望接受的，窗口后面的数据都是不希望接受的。

TCP的滑动窗口分为接收窗口和发送窗口 不分析这两种窗口就讨论是不妥当的。 TCP的滑动窗口主要有两个作用，一是提供TCP的可靠性，二是提供TCP的流控特性。同时滑动窗口机制还体现了TCP面向字节流的设计思路。TCP 段中窗口的相关字段。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/d6b970fb6d44aafeeec4a4c9d61a9225_hd.jpg)

TCP的Window是一个16bit位字段，它代表的是窗口的字节容量，也就是TCP的标准窗口最大为2^16-1=65535个字节。

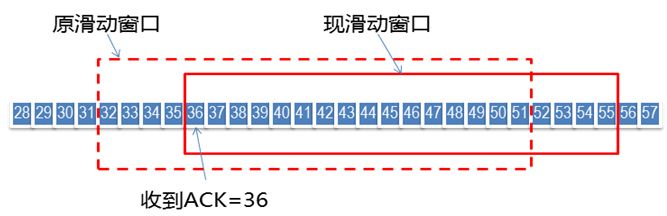
另外在TCP的选项字段中还包含了一个TCP窗口扩大因子，option-kind为3，option-length为3个字节，option-data取值范围0-14。窗口扩大因子用来扩大TCP窗口，可把原来16bit的窗口，扩大为31bit。

**滑动窗口基本原理**

对于TCP会话的发送方，任何时候在其发送缓存内的数据都可以分为4类，“已经发送并得到对端ACK的”，“已经发送但还未收到对端ACK的”，“未发送但对端允许发送的”，“未发送且对端不允许发送”。“已经发送但还未收到对端ACK的”和“未发送但对端允许发送的”这两部分数据称之为发送窗口。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/a1d5c050ad957880094a5f003b1ccd24_hd.jpg)

当收到接收方新的ACK对于发送窗口中后续字节的确认是，窗口滑动，滑动原理如下图。

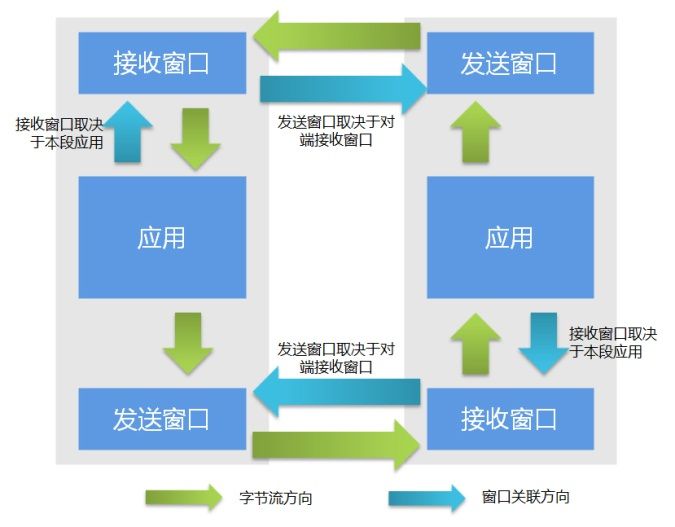
[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/9c21786770459afa47bfa2e4606cc454_hd.jpg)

当收到ACK=36时窗口滑动。

2）对于TCP的接收方，在某一时刻在它的接收缓存内存在3种。“已接收”，“未接收准备接收”，“未接收并未准备接收”（由于ACK直接由TCP协议栈回复，默认无应用延迟，不存在“已接收未回复ACK”）。其中“未接收准备接收”称之为接收窗口。

**发送窗口与接收窗口关系**

TCP是双工的协议，会话的双方都可以同时接收、发送数据。TCP会话的双方都各自维护一个“发送窗口”和一个“接收窗口”。其中各自的“接收窗口”大小取决于应用、系统、硬件的限制（TCP传输速率不能大于应用的数据处理速率）。各自的“发送窗口”则要求取决于对端通告的“接收窗口”，要求相同。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/c798dd393fcf7c03b1db78f5bcf0304b_hd.jpg)

**滑动窗口实现面向流的可靠性**

* 最基本的传输可靠性来源于“确认重传”机制。
* TCP的滑动窗口的可靠性也是建立在“确认重传”基础上的。
* 发送窗口只有收到对端对于本段发送窗口内字节的ACK确认，才会移动发送窗口的左边界。
* 接收窗口只有在前面所有的段都确认的情况下才会移动左边界。当在前面还有字节未接收但收到后面字节的情况下，窗口不会移动，并不对后续字节确认。以此确保对端会对这些数据重传。

**滑动窗口的流控特性**

TCP的滑动窗口是动态的，我们可以想象成小学常见的一个数学题，一个水池，体积V，每小时进水量V1，出水量V2。当水池满了就不允许再注入了，如果有个液压系统控制水池大小，那么就可以控制水的注入速率和量。这样的水池就类似TCP的窗口。应用根据自身的处理能力变化，通过本端TCP接收窗口大小控制来对对对端的发送窗口流量限制。

应用程序在需要（如内存不足）时，通过API通知TCP协议栈缩小TCP的接收窗口。然后TCP协议栈在下个段发送时包含新的窗口大小通知给对端，对端按通知的窗口来改变发送窗口，以此达到减缓发送速率的目的。

**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。

主要特点如下：

* 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST 等等。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
* 数据格式灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由Content-Type 加以标记。
* 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
* 无状态：HTTP 协议是无状态协议。无状态，是指协议对于事务处理没有记忆能力。无状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。
* 支持 B/S 及 C/S 模式。

**GET 和 POST 区别**

从三个层面来解答：

* Http 报文层面：GET 将请求信息放在 URL中，POST 方法报文中
* 数据库层面：GET 符合幂等性和安全性，POST 不符合
* 其他层面：GET 可以被缓存、被存储（书签），而 POST 不行

**Cookie 和 Session 的区别**

**Cookie 简介：**

* 是由服务器发给客户端的特殊信息，以文本的形式存放在客户端
* 客户端再次请求的时候，会把 Cookie 回发给服务端
* 服务器接收到后，会解析 Cookie 生成与客户端相对的内容

**Cookiet 的设置以及发送过程：**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/tisdfsdfrgremg.jpg)

**Session 简介：**

* 服务端的机制，在服务端保存的信息
* 解析客户端请求并操作 Session id ，按需保存状态信息

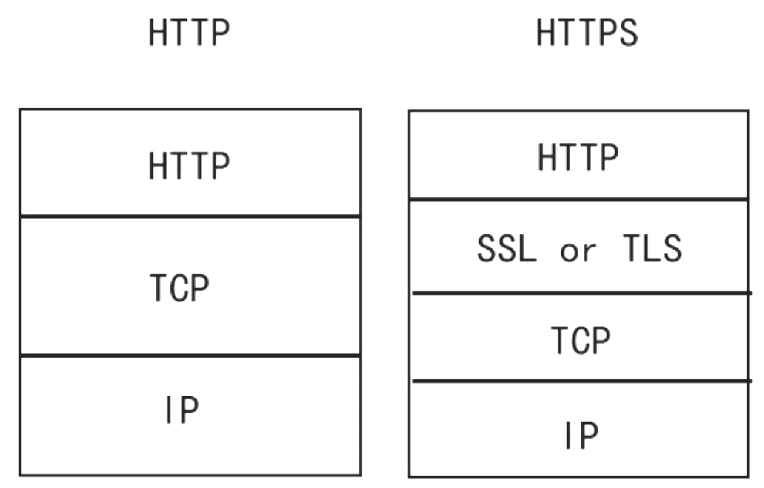
**Session 的实现方式：**

* 使用 Cookie 来实现
* 使用 URL 回写来实现，每次在 URL 添加 Session id 信息

**区别**：

* Cookie 数据存放在客户端的浏览器上，Session 数据存放在服务器上
* Session 相对于 Cookie 更安全
* 若考虑减轻服务器负担，应当使用 Cookie

**HTTP 和 HTTPs 的区别**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/1550579240583.png)

**SSL (Security Sockets Layer) 安全套接层**

* 为网络通信提供安全及数据完整性的一种安全协议
* 是操作系统对外的 API，SSL 3.0 更名为 TLS
* 采用身份验证和数据加密来保证网络的通信的安全和数据的完整性

**区别**

* HTTPS 需要到 CA 申请证书，HTTP 不需要
* HTTPS 密文传输，HTTP 明文传输
* 连接方式不同，HTTPS 默认使用 443 端口，HTTP 使用 80 端口
* HTTPS = HTTP + 加密 + 认证 + 完整性保护，较 HTTP 安全

**其他内容**

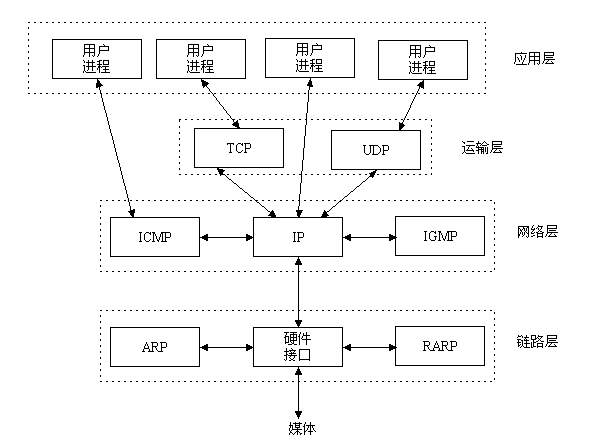
* 一 、基础概念
  + [URI](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#uri)
  + [请求和响应报文](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%AF%B7%E6%B1%82%E5%92%8C%E5%93%8D%E5%BA%94%E6%8A%A5%E6%96%87)
* 二、HTTP 方法
  + [GET](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#get)
  + [HEAD](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#head)
  + [POST](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#post)
  + [PUT](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#put)
  + [PATCH](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#patch)
  + [DELETE](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#delete)
  + [OPTIONS](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#options)
  + [CONNECT](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#connect)
  + [TRACE](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#trace)
* 三、HTTP 状态码
  + [1XX 信息](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#1xx-%E4%BF%A1%E6%81%AF)
  + [2XX 成功](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#2xx-%E6%88%90%E5%8A%9F)
  + [3XX 重定向](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#3xx-%E9%87%8D%E5%AE%9A%E5%90%91)
  + [4XX 客户端错误](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#4xx-%E5%AE%A2%E6%88%B7%E7%AB%AF%E9%94%99%E8%AF%AF)
  + [5XX 服务器错误](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#5xx-%E6%9C%8D%E5%8A%A1%E5%99%A8%E9%94%99%E8%AF%AF)
* 四、HTTP 首部
  + [通用首部字段](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E9%80%9A%E7%94%A8%E9%A6%96%E9%83%A8%E5%AD%97%E6%AE%B5)
  + [请求首部字段](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%AF%B7%E6%B1%82%E9%A6%96%E9%83%A8%E5%AD%97%E6%AE%B5)
  + [响应首部字段](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%93%8D%E5%BA%94%E9%A6%96%E9%83%A8%E5%AD%97%E6%AE%B5)
  + [实体首部字段](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%AE%9E%E4%BD%93%E9%A6%96%E9%83%A8%E5%AD%97%E6%AE%B5)
* 五、具体应用
  + [连接管理](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%BF%9E%E6%8E%A5%E7%AE%A1%E7%90%86)
  + [Cookie](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#cookie)
  + [缓存](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E7%BC%93%E5%AD%98)
  + [内容协商](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%86%85%E5%AE%B9%E5%8D%8F%E5%95%86)
  + [内容编码](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%86%85%E5%AE%B9%E7%BC%96%E7%A0%81)
  + [范围请求](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%8C%83%E5%9B%B4%E8%AF%B7%E6%B1%82)
  + [分块传输编码](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%88%86%E5%9D%97%E4%BC%A0%E8%BE%93%E7%BC%96%E7%A0%81)
  + [多部分对象集合](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%A4%9A%E9%83%A8%E5%88%86%E5%AF%B9%E8%B1%A1%E9%9B%86%E5%90%88)
  + [虚拟主机](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%99%9A%E6%8B%9F%E4%B8%BB%E6%9C%BA)
  + [通信数据转发](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E9%80%9A%E4%BF%A1%E6%95%B0%E6%8D%AE%E8%BD%AC%E5%8F%91)
* 六、HTTPs
  + [加密](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%8A%A0%E5%AF%86)
  + [认证](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E8%AE%A4%E8%AF%81)
  + [完整性保护](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%AE%8C%E6%95%B4%E6%80%A7%E4%BF%9D%E6%8A%A4)
  + [HTTPs 的缺点](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#https-%E7%9A%84%E7%BC%BA%E7%82%B9)
* 七、HTTP/2.0
  + [HTTP/1.x 缺陷](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#http1x-%E7%BC%BA%E9%99%B7)
  + [二进制分帧层](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E4%BA%8C%E8%BF%9B%E5%88%B6%E5%88%86%E5%B8%A7%E5%B1%82)
  + [服务端推送](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E6%9C%8D%E5%8A%A1%E7%AB%AF%E6%8E%A8%E9%80%81)
  + [首部压缩](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E9%A6%96%E9%83%A8%E5%8E%8B%E7%BC%A9)
* [八、HTTP/1.1 新特性](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%85%ABhttp11-%E6%96%B0%E7%89%B9%E6%80%A7)
* 九、GET 和 POST 比较
  + [作用](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E4%BD%9C%E7%94%A8)
  + [参数](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%8F%82%E6%95%B0)
  + [安全](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%AE%89%E5%85%A8)
  + [幂等性](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%B9%82%E7%AD%89%E6%80%A7)
  + [可缓存](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#%E5%8F%AF%E7%BC%93%E5%AD%98)
  + [XMLHttpRequest](https://github.com/CyC2018/CS-Notes/blob/master/docs/notes/HTTP.md#xmlhttprequest)

**浏览器输入地址回车后发生的事情**

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

**Socket 通信**

TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WANs）设计的。UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议。它是属于 TCP/IP 协议族中的一种。 这里有一张图，表明了这些协议的关系。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/socket1.jpg)  
TCP/IP协议族包括运输层、网络层、链路层。现在你知道TCP/IP与UDP的关系了吧。

**Socket在哪里呢？**

上图我们没有看到 Socket 的影子，那么它到底在哪里呢？还是用图来说话，一目了然。

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/socket2.jpg)

**Socket 是什么呢？**

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。

**Socket 通信原理**

[](http://blog.cuzz.site/2019/02/19/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/socket3.jpg)

先从服务器端说起。服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

**TCP 实现**

服务端

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 | /\*\*  \* @Author: cuzz  \* @Date: 2019/2/19 22:36  \* @Description:  \*/ public class TCPServer {  public static void main(String[] args) throws IOException {  // 创建socket，并将socket绑定到65000端口  ServerSocket serverSocket = new ServerSocket(65000);  // 死循环，使socket一直等待并处理客户端发过来的请求  while (true) {  // 监听6500端口，直到客户端返回连接信息后才返回  Socket socket = serverSocket.accept();  // 获取客户端请求信息后，执行相关逻辑  new LengthCalculator(socket).start();  }  } }  class LengthCalculator extends Thread {   private Socket socket;   public LengthCalculator(Socket socket) {  this.socket = socket;  }   @Override  public void run() {  try {  // 获取socket的输出流  OutputStream os = socket.getOutputStream();  // 获取socket的输入流  InputStream is = socket.getInputStream();  byte[] bytes = new byte[1024];  int len = 0;  StringBuilder sb = new StringBuilder();  while ((len = is.read(bytes)) != -1) {  os.write(bytes, 0 , len);  System.out.println(new String(bytes, 0 , len));  }  // 不要忘记关闭输入输出流  os.close();  is.close();  socket.close();  } catch (IOException e) {  e.printStackTrace();  }  } } |

客服端

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | public class TCPClinet {  public static void main(String[] args) throws IOException {  // 创建socket，并指定连接的是ip和端口号  Socket socket = new Socket("127.0.0.1", 65000);  // 获取输出流  OutputStream os = socket.getOutputStream();  // 获取输入流  InputStream is = socket.getInputStream();  os.write("hello world".getBytes());  int len = 0;  byte[] bytes = new byte[1024];  len = is.read(bytes);  String content = new String(bytes, 0, len);  System.out.println(content);  is.close();  os.close();  socket.close();  } } |

**UDP 实现**

服务端

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | public class UDPServer {  public static void main(String[] args) throws Exception {  // 服务端接受客户端发送的数据报  DatagramSocket socket = new DatagramSocket(65001); //监听的端口号  byte[] buff = new byte[100]; //存储从客户端接受到的内容  DatagramPacket packet = new DatagramPacket(buff, buff.length);  //接受客户端发送过来的内容，并将内容封装进DatagramPacket对象中  socket.receive(packet);   byte[] data = packet.getData(); //从DatagramPacket对象中获取到真正存储的数据  //将数据从二进制转换成字符串形式  String content = new String(data, 0, packet.getLength());  System.out.println(content);  //将要发送给客户端的数据转换成二进制  byte[] sendedContent = String.valueOf(content.length()).getBytes();  // 服务端给客户端发送数据报  //从DatagramPacket对象中获取到数据的来源地址与端口号  DatagramPacket packetToClient = new DatagramPacket(sendedContent,  sendedContent.length, packet.getAddress(), packet.getPort());  socket.send(packetToClient); //发送数据给客户端  } } |

客服端

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | public class UDPClient {  public static void main(String[] args) throws Exception {  // 客户端发数据报给服务端  DatagramSocket socket = new DatagramSocket();  // 要发送给服务端的数据  byte[] buf = "Hello World".getBytes();  // 将IP地址封装成InetAddress对象  InetAddress address = InetAddress.getByName("127.0.0.1");  // 将要发送给服务端的数据封装成DatagramPacket对象 需要填写上ip地址与端口号  DatagramPacket packet = new DatagramPacket(buf, buf.length, address,  65001);  // 发送数据给服务端  socket.send(packet);   // 客户端接受服务端发送过来的数据报  byte[] data = new byte[100];  // 创建DatagramPacket对象用来存储服务端发送过来的数据  DatagramPacket receivedPacket = new DatagramPacket(data, data.length);  // 将接受到的数据存储到DatagramPacket对象中  socket.receive(receivedPacket);  // 将服务器端发送过来的数据取出来并打印到控制台  String content = new String(receivedPacket.getData(), 0,  receivedPacket.getLength());  System.out.println(content);  } } |