**1:编码与解码**

字节（byte）是计算机网络通信过程中实际传输的二进制数，每个字节由8位二进制数构成，范围从00000000至11111111，转化为十进制数为0到255；而编程语言中使用的为字符，除非使用者主动请求对字符和外部使用的实际字节进行转化，否则对使用者可见的只有字符；

解码（decoding）：是在应用程序使用字节时发生的，当应用程序从文件或者网络接受到字节时，需要对通信信道间传输的原始字节进行解码‘

编码（encoding）：是程序把字符串对外输出时所实施的过程，此时，程序使用某一种数字计算机使用的编码方法将字符串转化为字节；

**2：网际协议**

网络互连（networking）指的是通过物理链路将多台计算机链接，使之可以互相通信；而网际互联（Internetworking）指的是将相邻的物理网络连接，使之形成更大的网络系统，比如互联网。

**网络设备间**进行共享的基本单元是数据包（packet）；数据包在物理层通常只有两个属性：包含的字节串数据以及目标传输地址；

**网卡**是工作在**链路层**的网络组件，是局域网中连接计算机和传输介质的接口，不仅能实现与局域网传输介质之间的物理连接和电信号匹配，还涉及帧的发送与接收、帧的封装与拆封、介质访问控制、数据的编码与解码以及数据缓存的功能等；

网际协议（IP）是为全世界通过互联网链接的计算机赋予统一地址系统的一种机制，它使得数据包能够从互联网的一端发送到另一端；

**3：IP**

IP的最初版本是为连接到万维网的每台计算机分配了一个4字节的地址，格式如下：127.120.110.100（传统的4字节IP地址）；由于纯数字不方便记忆，人们使用互联网时常常用主机名（hostname）来代替IP地址，主机名通过DNS（Domain Name System）域名系统解析为IP地址；

传统的IP地址可以从左往右读：前1~2个字节表示某个机构，接下来的字节通常表示目标机器所在的特定子网，最后一个字节将地址细化至该特定的机器或服务；以127开头的IP地址是特殊的预留地址段，这一地址段是由机器上运行的本地应用程序使用，127.0.0.1表示该程序的机器本身，通常可以通过主机名localhost来访问；IP地址是以网络号和主机号来表示网络上的主机的，只有在一个网络号下的计算机之间才能“直接”互通，不同网络号的计算机要通过网关（Gateway）才能互通。

**4：路由**

一旦应用程序向操作系统请求向某一特定IP地址发送数据，操作系统就需要决定如何使用该机器连接的某一物理网络来传输数据，这一决定（根据目的IP地址选择将IP数据包发往何处）就叫做**路由（routing）**。

**5：数据包分组**

以太网只支持1500B的数据包；网络数据包中包含一个表示不分组（DF，dont Fragment）的标记；如果没有设置DF标记，那么表示允许分组，当数据包的大小超过网络能够允许的上限时，网关可以将其分成多个小数据包，并进行标记，以表示接收方在接受之后需要将这些小数据包重组为原始数据包；如果设置了DF标记，表示不允许分组，此时如果网络无法容纳数据包，将会丢弃该数据包，并发回 一条错误信息。一个互联网子网能够接受的最大数据包叫做最大传输单元（MTU， Maximum Transmission Unit）。

**6：UDP**

IP协议只负责将每个数据包传输至正确的机器中，但无法确定传输至哪个应用程序中。故需要为两台主机间传送的大量数据包打上标签，使数据包与该机器上正在进行的其他网络会话使用的数据包分开，这一过程叫做多路复用（multiplexing）; 对两台主机间独立传输的数据包流发生的任何错误，都需要进行修复，而丢失的数据包也需要进行重传，直到发送成功，另外，如果数据包到达时顺序错乱，则要将这些数据包进行重组，最后，要丢弃重复的数据包，以保证数据流中的信息没有冗余，提供这些保证的特性叫做可靠传输（reliable transport）。

UDP用户数据报协议实现了多路复用和分解；TCP传输控制协议则实现了上面两个全部功能。多路复用就是允许多个会话共享同一介质或机制的一种解决方案，该方案为每个UDP数据报分配了一堆无符号16位端口号，范围从0到65536。

附注：OSI模型

名称 层次 功能

物理层 1 实现计算机系统与网络间的物理连接

数据链路层 2 进行数据打包与解包，形成信息帧

网络层 3 提供数据通过的路由

传输层 4 提供传输顺序信息与响应

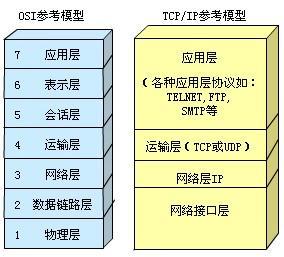
会话层 5 建立和中止连接

表示层 6 数据转换、确认数据格式

应用层 7 提供用户程序接口

**7：TCP**

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。在简化的计算机网络OSI模型中，它完成第四层传输层所指定的功能，用户数据报协议（UDP）是同一层内的另一个重要的传输协议。TCP层是位于IP层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是IP层不提供这样的流机制，而是提供不可靠的包交换。应用层向TCP层发送用于网间传输的、用8位字节表示的数据流，然后TCP把数据流分成适当长度的报文段（通常受该计算机连接的网络的数据链路层的MTU的限制）。之后TCP把结果包传给IP层，由它来通过网络将包传送给接收端机器的TCP层。TCP为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端机器对已成功收到的包发回一个相应的确认；如果发送端实体在允许的往返时延内未收到确认，那么对应的数据包就被假设为已丢失将会被进行重传。TCP用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。



**TCP/IP模型四层的功能如下**：

（引用自：<https://www.jianshu.com/p/ef892323e68f>）

应用层: 向用户提供一组常用的应用程序，比如电子邮件、远程登录等。

传输层: 提供应用程序间的通信。其功能包括：格式化信息流；提供可靠传输。为实现后者，传输层协议规定接收端必须发回确认，并且假如分组丢失，必须重新发送。

网络层 ：负责相邻计算机之间的通信。其功能包括三方面：处理来自传输层的分组发送请求，收到请求后，将分组装入IP数据报，填充报头，选择去往信宿机的路径，然后将数据报发往适当的网络接口；处理输入数据报：首先检查其合法性，然后进行寻径--假如该数据报已到达信宿机，则去掉报头，将剩下部分交给适当的传输协议；假如该数据报尚未到达信宿，则转发该数据报；处理路径、流控、拥塞等问题。

网络接口层：这是TCP/IP软件的最低层，负责接收IP数据报并通过网络

送之，或者从网络上接收物理帧，抽出IP数据报，交给IP层。

TCP/IP 意味着 TCP 和 IP 在一起协同工作。

TCP 负责应用软件（比如你的浏览器）和网络软件之间的通信。

IP 负责计算机之间的通信。

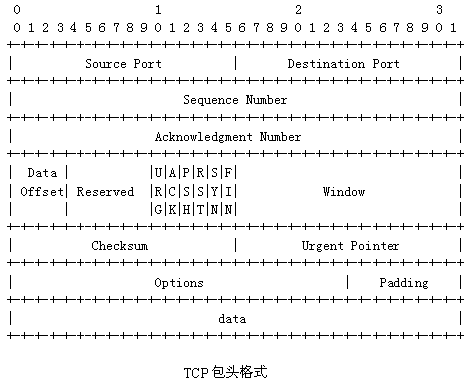
TCP 负责将数据分割并装入 IP 包，然后在它们到达的时候重新组合它们。

IP 负责将包发送至目标IP地址

注释：SYN（synchronous）是TCP/IP建立连接时使用的握手信号。在客户机和服务器之间建立正常的TCP网络连接时，客户机首先发出一个SYN消息，服务器使用SYN+ACK应答表示接收到了这个消息，最后客户机再以ACK消息响应。这样在客户机和服务器之间才能建立起可靠的TCP连接，数据才可以在客户机和服务器之间传递。

ACK (Acknowledgement）即是确认字符，在数据通信中，接收站发给发送站的一种传输类控制字符。表示发来的数据已确认接收无误。

**TCP 报头格式：**



TCP协议头最少20个字节，包括以下的区域：

TCP源端口(Source Port)：16位的源端口，其中包含初始化通信的端口。

源端口和源IP地址的作用是标示报文的返回地址；

TCP目的端口(Destination port)：16位的目的端口域定义传输至哪里。这个端口指明报文接收计算机上的应用程序地址端口；

TCP序列号（序列码,Sequence Number）：32位的序列号标识了TCP报文中第一个byte在对应方向的传输中对应的字节序号。当SYN出现，序列码实际上是初始序列码（ISN，initial sequence number)，而第一个数据字节是ISN+1，单位是byte。比如发送端发送的一个TCP包净荷(不包含TCP头)为12byte，SN为5，则发送端接着发送的下一个数据包的时候，SN应该设置为5+12=17。通过系列号，TCP接收端可以识别出重复接收到的TCP包，从而丢弃重复包，同时对于乱序数据包也可以依靠系列号进行重排序，进而对高层提供有序的数据流。另外SYN标志和FIN标志在逻辑上也占用一个byte，当SYN标志位有效的时候，该字段也称为ISN；

TCP应答号(Acknowledgment Number)：32位的ACK Number标识了报文发送端期望接收的字节序列。如果设置了ACK控制位，这个值表示一个准备接收的包的序列码，注意是准备接收的包，比如当前接收端接收到一个净荷为12byte的数据包，SN为5，则接收端可能会回复一个确认收到的数据包，如果这个数据包之前的数据也都已经收到了，这个数据包中的ACK Number则设置为12+5=17，表示17byte之前的数据都已经收到了。在举一个例子，如果在这个数据包之前有个SN为3，净荷为2byte的数据包丢失，则在接受端接收到这个SN为5的乱序数据包的时候，协议要求接收端必须要回复一个ACK确认包，这个确认包中的Ack Number只能设置为3。

数据偏移量(data offset)：4位包括TCP头大小，指示何处数据开始；

保留(Reserved)：6位值域，这些位必须是0。为了将来定义新的用途所保留。

标志(Code Bits)：6位标志域。按照顺序排列是：URG、ACK、PSH、RST、SYN、FIN。

1.URG：紧急标志(The urgent pointer) ：紧急标志为"1"表明该位有效；

2.ACK：确认标志：表明确认编号栏有效。大多数情况下该标志位是置位的。TCP报头内的确认编号栏内包含的确认编号（w+1）为下一个预期的序列编号，同时提示远端系统已经成功接收所有数据；

3.PSH：推标志：该标志置位时，接收端不将该数据进行队列处理，而是尽可能快地将数据转由应用处理。在处理Telnet或rlogin等交互模式的连接时，该标志总是置位的；

4.RST：复位标志：用于复位相应的TCP连接。通常在发生异常或者错误的时候会触发复位TCP连接；

5.SYN：同步标志：该标志仅在三次握手建立TCP连接时有效。它提示TCP连接的服务端检查序列编号，该序列编号为TCP连接初始端(一般是客户端)的初始序列编号。在这里，可以把TCP序列编号看作是一个范围从0到4，294，967，295的32位计数器。通过TCP连接交换的数据中每一个字节都经过序列编号。在TCP报头中的序列编号栏包括了TCP分段中第一个字节的序列编号。类似的后续文章介绍中当这个SYN标志位有效的时候我们称呼这个包为SYN包；

6.FIN：结束标志：带有该标志置位的数据包用来结束一个TCP回话，但对应端口仍处于开放状态，准备接收后续数据，当FIN标志有效的时候我们称呼这个包为FIN包；

窗口大小(Window Size)：16位，该值指示了从Ack Number开始还愿意接收多少byte的数据量，也即用来表示当前接收端的接收窗还有多少剩余空间。用于TCP的流量控制；

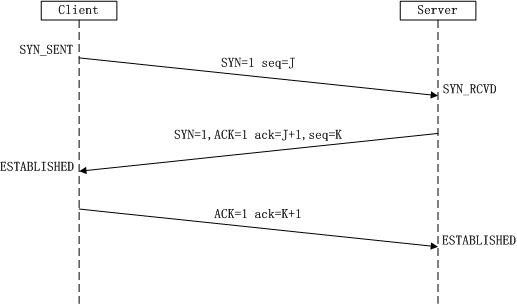
校验位(Checksum)：16位TCP头。发送端基于数据内容计算一个数值，接收端要与发送端数值结果完全一样，才能证明数据的有效性。接收端checksum校验失败的时候会直接丢掉这个数据包。CheckSum是根据伪头+TCP头+TCP数据三部分进行计算的。另外对于大的数据包，checksum并不能可靠的反应比特错误，应用层应该再添加自己的校验方式；

优先指针（紧急,Urgent Pointer）：16位，指向后面是优先数据的字节，在URG标志设置了时才有效。如果URG标志没有被设置，紧急域作为填充。加快处理标示为紧急的数据段；

选项(Option)：长度不定，但长度必须以是32bits的整数倍。常见的选项包括MSS、SACK、Timestamp等等，后续的内容会分别介绍相关选项。

**TCP三次握手（建立连接）：**

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3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，整个流程如下图所示：



（1）第一次握手：Client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Server，Client进入SYN\_SENT状态，等待Server确认。

（2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server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Client以确认连接请求，Server进入SYN\_RCVD状态。

（3）第三次握手：Client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Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

简单来说，就是

1、建立连接时，客户端发送SYN包（SYN=i）到服务器，并进入到SYN-SEND状态，等待服务器确认

2、服务器收到SYN包，必须确认客户的SYN（ack = i+1）,同时自己也发送一个SYN包（SYN = k）,即SYN+ACK包，此时服务器进入SYN-RECV状态

3、客户端收到服务器的SYN+ACK包，向服务器发送确认报ACK（ack=k+1）,此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手，客户端与服务器开始传送数据

**TCP四次挥手（连接终止）：**

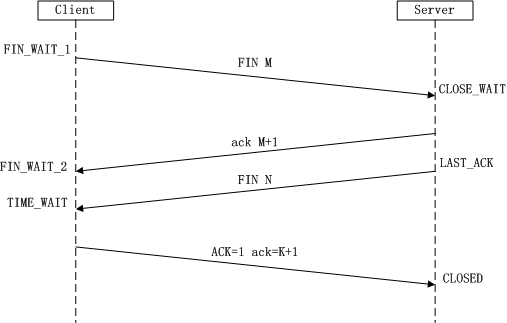
由于TCP连接是全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，下图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。



**为什么建立连接是三次握手，而关闭连接却是四次挥手呢？**

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

**TCP协议的可靠性实现：**

　TCP提供一种面向连接的、可靠的字节流服务。面向连接意味着两个使用TCP的应用（通常是一个客户和一个服务器）在彼此交换数据包之前必须先建立一个TCP连接。在一个TCP连接中，仅有两方进行彼此通信。

TCP通过下列方式来提供可靠性：

1．应用数据被分割成**TCP认为**最适合发送的数据块。这和UDP完全不同，应用程序产生的数据长度将保持不变。由TCP传递给IP的信息单位称为报文段或段；

2．当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。当TCP收到发自TCP连接另一端的数据，它将发送一个确认。TCP有延迟确认的功能，若此功能没有打开，则是立即确认。功能打开，则由定时器触发确认时间点；

3．TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）；

4．既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文段的到达也可能会失序。如果必要，TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层；

5．既然IP数据报会发生重复，TCP的接收端必须丢弃重复的数据；

6．TCP还能提供流量控制。TCP连接的每一方都有固定大小的缓冲空间。TCP的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

**注释**：TCP对字节流的内容不作任何解释。TCP不知道传输的数据字节流是二进制数据，还是ASCII字符、EBCDIC字符或者其他类型数据。对字节流的解释由TCP连接双方的应用层解释。

一、什么是长连接

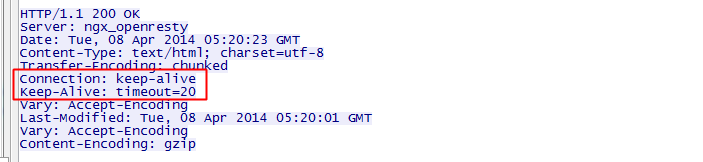
     HTTP1.1规定了默认保持长连接（HTTP persistent connection ，也有翻译为持久连接），数据传输完成了保持TCP连接不断开（不发RST包、不四次挥手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

　HTTP首部的Connection: Keep-alive是HTTP1.0浏览器和服务器的实验性扩展，当前的HTTP1.1 RFC2616文档没有对它做说明，因为它所需要的功能已经默认开启，无须带着它，但是实践中可以发现，浏览器的报文请求都会带上它。如果HTTP1.1版本的HTTP请求报文不希望使用长连接，则要在HTTP请求报文首部加上Connection: close。《HTTP权威指南》提到，有部分古老的HTTP1.0 代理不理解Keep-alive，而导致长连接失效：客户端-->代理-->服务端，客户端带有Keep-alive，而代理不认识，于是将报文原封不动转给了服务端，服务端响应了Keep-alive，也被代理转发给了客户端，于是保持了“客户端-->代理”连接和“代理-->服务端”连接不关闭，但是，当客户端第发送第二次请求时，代理会认为当前连接不会有请求了，于是忽略了它，长连接失效。书上也介绍了解决方案：当发现HTTP版本为1.0时，就忽略Keep-alive，客户端就知道当前不该使用长连接。其实，在实际使用中不需要考虑这么多，很多时候代理是我们自己控制的，如Nginx代理，代理服务器有长连接处理逻辑，服务端无需做patch处理，常见的是客户端跟Nginx代理服务器使用HTTP1.1协议&长连接，而Nginx代理服务器跟后端服务器使用HTTP1.0协议&短连接。

    在实际使用中，HTTP头部有了Keep-Alive这个值并不代表一定会使用长连接，客户端和服务器端都可以无视这个值，也就是不按标准来。

二、长连接的过期时间

    客户端的长连接不可能无限期的拿着，会有一个超时时间，服务器有时候会告诉客户端超时时间，譬如：



     上图中的Keep-Alive: timeout=20，表示这个TCP通道可以保持20秒。另外还可能有max=XXX，表示这个长连接最多接收XXX次请求就断开。对于客户端来说，如果服务器没有告诉客户端超时时间也没关系，服务端可能主动发起四次挥手断开TCP连接，客户端能够知道该TCP连接已经无效；另外TCP还有心跳包来检测当前连接是否还活着，方法很多，避免浪费资源。

三、长连接的数据传输完成识别

使用长连接之后，客户端、服务端怎么知道本次传输结束呢？两部分：1是判断传输数据是否达到了Content-Length指示的大小；2动态生成的文件没有Content-Length，它是分块传输（chunked），这时候就要根据chunked编码来判断，chunked编码的数据在最后有一个空chunked块，表明本次传输数据结束。更细节的介绍可以看[这篇文章](http://www.cnblogs.com/skynet/archive/2010/12/11/1903347.html" \t "_blank)。

四、并发连接数的数量限制

在web开发中需要关注浏览器并发连接的数量，[RFC文档](http://tools.ietf.org/html/rfc2616" \l "page-47" \t "_blank)说，客户端与服务器最多就连上两通道，但服务器、个人客户端要不要这么做就随人意了，有些服务器就限制同时只能有1个TCP连接，导致客户端的多线程下载（客户端跟服务器连上多条TCP通道同时拉取数据）发挥不了威力，有些服务器则没有限制。浏览器客户端就比较规矩，[知乎这里有分析](http://www.zhihu.com/question/20474326" \t "_blank)，限制了同域名下能启动若干个并发的TCP连接去下载资源。并发数量的限制也跟长连接有关联，打开一个网页，很多个资源的下载可能就只被放到了少数的几条TCP连接里，这就是TCP通道复用（长连接）。如果并发连接数少，意味着网页上所有资源下载完需要更长的时间（用户感觉页面打开卡了）；并发数多了，服务器可能会产生更高的资源消耗峰值。浏览器只对同域名下的并发连接做了限制，也就意味着，web开发者可以把资源放到不同域名下，同时也把这些资源放到不同的机器上，这样就完美解决了。

五、容易混淆的概念——**TCP的keep alive和HTTP的Keep-alive**

    TCP的keep alive是检查当前TCP连接是否活着；HTTP的Keep-alive是要让一个TCP连接活久点。它们是不同层次的概念。

    TCP keep alive的表现：

    当一个连接“一段时间”没有数据通讯时，一方会发出一个心跳包（Keep Alive包），如果对方有回包则表明当前连接有效，继续监控。

这个“一段时间”可以设置。

六、HTTP 流水线技术

    使用了HTTP长连接（HTTP persistent connection ）之后的好处，包括可以使用HTTP 流水线技术（HTTP pipelining，也有翻译为管道化连接），它是指，**在一个TCP连接内，多个HTTP请求可以并行，下一个HTTP请求在上一个HTTP请求的应答完成之前就发起。**从wiki上了解到这个技术目前并没有广泛使用，使用这个技术必须要求客户端和服务器端都能支持，目前有部分浏览器完全支持，而服务端的支持仅需要：按HTTP请求顺序正确返回Response（也就是请求&响应采用FIFO模式），wiki里也特地指出，只要服务器能够正确处理使用HTTP pipelinning的客户端请求，那么服务器就算是支持了HTTP pipelining。

    由于要求服务端返回响应数据的顺序必须跟客户端请求时的顺序一致，这样也就是要求FIFO，这容易导致Head-of-line blocking：第一个请求的响应发送影响到了后边的请求，因为这个原因导致HTTP流水线技术对性能的提升并不明显（wiki提到，这个问题会在HTTP2.0中解决）。另外，使用这个技术的还必须是幂等的HTTP方法，因为客户端无法得知当前已经处理到什么地步，重试后可能发生不可预测的结果。POST方法不是幂等的：同样的报文，第一次POST跟第二次POST在服务端的表现可能会不一样。

    在HTTP长连接的wiki中提到了HTTP1.1的流水线技术对RFC规定一个用户最多两个连接的指导意义：流水线技术实现好了，那么多连接并不能提升性能。我也觉得如此，并发已经在单个连接中实现了，多连接就没啥必要，除非瓶颈在于单个连接上的资源限制迫使不得不多开连接抢资源。

HTTP确定信息长度：

在HTTP协议中，有Content-Length的详细解读。Content-Length用于描述HTTP消息实体的传输长度the transfer-length of the message-body。在HTTP协议中，消息实体长度和消息实体的传输长度是有区别，比如说gzip压缩下，消息实体长度是压缩前的长度，消息实体的传输长度是gzip压缩后的长度。

在具体的HTTP交互中，客户端是如何获取消息长度的呢，主要基于以下几个规则：

响应为1xx，204，304相应或者head请求，则直接忽视掉消息实体内容。

如果有Transfer-Encoding，则优先采用Transfer-Encoding里面的方法来找到对应的长度。比如说Chunked模式。

“如果head中有Content-Length，那么这个Content-Length既表示实体长度，又表示传输长度。如果实体长度和传输长度不相等(比如说设置了Transfer-Encoding)，那么则不能设置Content-Length。如果设置了Transfer-Encoding，那么Content-Length将被忽视”。这句话翻译的优点饶，其实关键就一点：有了Transfer-Encoding，则不能有Content-Length。

Range传输。不关注，没详细看了：)

通过服务器关闭连接能确定消息的传输长度。(请求端不能通过关闭连接来指明请求消息体的结束，因为这样可以让服务器没有机会继续给予响应)。这种情况主要对应为短连接，即非keep-alive模式。

HTTP1.1必须支持chunk模式。因为当不确定消息长度的时候，可以通过chunk机制来处理这种情况。

在包含消息内容的header中，如果有content-length字段，那么该字段对应的值必须完全和消息主题里面的长度匹配。

“The entity-length of a message is the length of the message-body before any transfer-codings have been applied”

也就是有chunk就不能有content-length 。

其实后面几条几乎可以忽视，简单总结后如下：

1、Content-Length如果存在并且有效的话，则必须和消息内容的传输长度完全一致。(经过测试，如果过短则会截断，过长则会导致超时。)

2、如果存在Transfer-Encoding(重点是chunked)，则在header中不能有Content-Length，有也会被忽视。

3、如果采用短连接，则直接可以通过服务器关闭连接来确定消息的传输长度。(这个很容易懂)

结合HTTP协议其他的特点，比如说Http1.1之前的不支持keep alive。那么可以得出以下结论：

1、在Http 1.0及之前版本中，content-length字段可有可无。

2、在http1.1及之后版本。如果是keep alive，则content-length和chunk必然是二选一。若是非keep alive，则和http1.0一样。content-length可有可无。

Keep Alive：

**Keep-Alive模式：**   
我们知道Http协议采用“请求-应答”模式，当使用普通模式，即非Keep-Alive模式时，每个请求/应答，客户端和服务器都要新建一个连接，完成之后立即断开连接；当使用Keep-Alive模式时，Keep-Alive功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive功能避免了建立或者重新建立连接。   
http1.0中默认是关闭的，需要在http头加入”Connection: Keep-Alive”，才能启用Keep-Alive；   
http 1.1中默认启用Keep-Alive，如果加入”Connection: close “才关闭。目前大部分浏览器都是用http1.1协议，也就是说默认都会发起Keep-Alive的连接请求了，所以是否能完成一个完整的Keep- Alive连接就看服务器设置情况。

**开启Keep-Alive的优缺点：**   
优点：Keep-Alive模式更加高效，因为避免了连接建立和释放的开销。   
缺点：长时间的Tcp连接容易导致系统资源无效占用，浪费系统资源。

**当保持长连接时，如何判断一次请求已经完成？**   
Content-Length   
Content-Length表示实体内容的长度。浏览器通过这个字段来判断当前请求的数据是否已经全部接收。   
所以，当浏览器请求的是一个静态资源时，即服务器能明确知道返回内容的长度时，可以设置Content-Length来控制请求的结束。但当服务器并不知道请求结果的长度时，如一个动态的页面或者数据，Content-Length就无法解决上面的问题，这个时候就需要用到Transfer-Encoding字段。

Transfer-Encoding   
Transfer-Encoding是指传输编码，在上面的问题中，当服务端无法知道实体内容的长度时，就可以通过指定Transfer-Encoding: chunked来告知浏览器当前的编码是将数据分成一块一块传递的。当然, 还可以指定Transfer-Encoding: gzip, chunked表明实体内容不仅是gzip压缩的，还是分块传递的。最后，当浏览器接收到一个长度为0的chunked时， 知道当前请求内容已全部接收。

**Keep-Alive timeout：**   
Http守护进程，一般都提供了keep-alive timeout时间设置参数。比如nginx的keepalive\_timeout，和Apache的KeepAliveTimeout。这个keepalive\_timout时间值意味着：一个http产生的tcp连接在传送完最后一个响应后，还需要hold住keepalive\_timeout秒后，才开始关闭这个连接。   
当http守护进程发送完一个响应后，理应马上主动关闭相应的tcp连接，设置 keepalive\_timeout后，httpd守护进程会想说：”再等等吧，看看浏览器还有没有请求过来”，这一等，便是keepalive\_timeout时间。如果守护进程在这个等待的时间里，一直没有收到浏览器发过来http请求，则关闭这个http连接。

**Tcp的Keepalive：**   
连接建立之后，如果客户端一直不发送数据，或者隔很长时间才发送一次数据，当连接很久没有数据报文传输时如何去确定对方还在线，到底是掉线了还是确实没有数据传输，连接还需不需要保持，这种情况在TCP协议设计中是需要考虑到的。   
TCP协议通过一种巧妙的方式去解决这个问题，当超过一段时间之后，TCP自动发送一个数据为空的报文（侦测包）给对方，如果对方回应了这个报文，说明对方还在线，连接可以继续保持，如果对方没有报文返回，并且重试了多次之后则认为链接丢失，没有必要保持连接。

tcp keep-alive是TCP的一种检测TCP连接状况的保鲜机制。tcp keep-alive保鲜定时器，支持三个系统内核配置参数：   
net.ipv4.tcp\_keepalive\_intvl = 15   
net.ipv4.tcp\_keepalive\_probes = 5   
net.ipv4.tcp\_keepalive\_time = 1800   
keepalive是TCP保鲜定时器，当网络两端建立了TCP连接之后，闲置（双方没有任何数据流发送往来）了tcp\_keepalive\_time后，服务器就会尝试向客户端发送侦测包，来判断TCP连接状况(有可能客户端崩溃、强制关闭了应用、主机不可达等等)。如果没有收到对方的回答(ack包)，则会在 tcp\_keepalive\_intvl后再次尝试发送侦测包，直到收到对方的ack,如果一直没有收到对方的ack,一共会尝试 tcp\_keepalive\_probes次，每次的间隔时间在这里分别是15s, 30s, 45s, 60s, 75s。如果尝试tcp\_keepalive\_probes,依然没有收到对方的ack包，则会丢弃该TCP连接。TCP连接默认闲置时间是2小时，一般设置为30分钟足够了。

心跳包的发送，通常有两种技术：  
方法1：应用层自己实现的心跳包   
由***应用程序自己发送心跳包来检测连接***是否正常，大致的方法是：服务器在一个 Timer事件中定时 向客户端发送一个短小精悍的数据包，然后启动一个低级别的线程，在该线程中不断检测客户端的回应， 如果在一定时间内没有收到客户端的回应，即认为客户端已经掉线；同样，如果客户端在一定时间内没有收到服务器的心跳包，则认为连接不可用。  
  
方法2：TCP的KeepAlive保活机制  
因为要考虑到一个服务器通常会连接多个客户端，因此由用户在应用层自己实现心跳包，代码较多且稍显复杂，而利用TCP／IP协议层为内置的KeepAlive功能来实现心跳功能则简单得多。 不论是服务端还是客户端，一方开启KeepAlive功能后，就会自动在规定时间内向对方发送心跳包， 而另一方在收到心跳包后就会自动回复，以告诉对方我仍然在线。 因为开启KeepAlive功能需要消耗额外的宽带和流量，所以TCP协议层默认并不开启KeepAlive功 能，尽管这微不足道，但在按流量计费的环境下增加了费用，另一方面，KeepAlive设置不合理时可能会 因为短暂的网络波动而断开健康的TCP连接。并且，默认的KeepAlive超时需要7,200，000 MilliSeconds， 即2小时，探测次数为5次。对于很多服务端应用程序来说，2小时的空闲时间太长。因此，我们需要手工开启KeepAlive功能并设置合理的KeepAlive参数。

HTTPs：

**简介**

HTTPS是在HTTP的基础上和ssl/tls证书结合起来的一种协议,保证了传输过程中的安全性,减少了被恶意劫持的可能.很好的解决了http的三个缺点（被监听、被篡改、被伪装）

**对称加密和非对称加密**

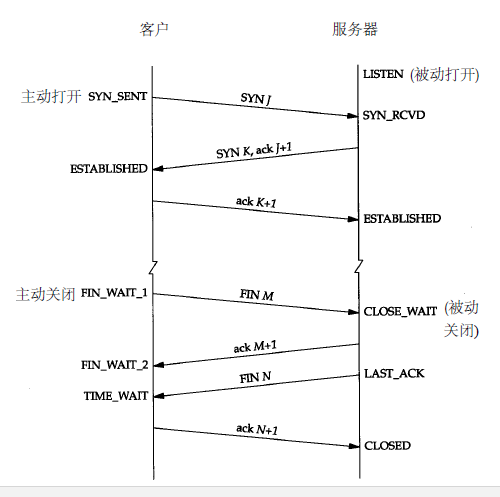
* 对称加密
  + 即加密的密钥和解密的密钥相同,
* 非对称加密
  + 非对称加密将密钥分为公钥和私钥,公钥可以公开,私钥需要保密,客户端公钥加密的数据,服务端可以通过私钥来解密

**建立连接**

* HTTP和HTTPS都需要在建立连接的基础上来进行数据传输,是基本操作
* 当客户在浏览器中输入网址的并且按下回车,浏览器会在浏览器DNS缓存,本地DNS缓存,和Hosts中寻找对应的记录,如果没有获取到则会请求DNS服务来获取对应的ip
* 当获取到ip后,tcp连接会进行三次握手建立连接

**tcp的三次挥手和四次挥手**

**过程简图**



**三次挥手(建立连接)**

* 第一次：建立连接时，客户端发送SYN包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；
* 第二次：服务器收到SYN包，向客户端返回ACK（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RCVD状态；
* 第三次：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。
* 完成三次握手，客户端与服务器开始传送数据，也就是ESTABLISHED状态。
* 三次握手保证了不会建立无效的连接，从而浪费资源。

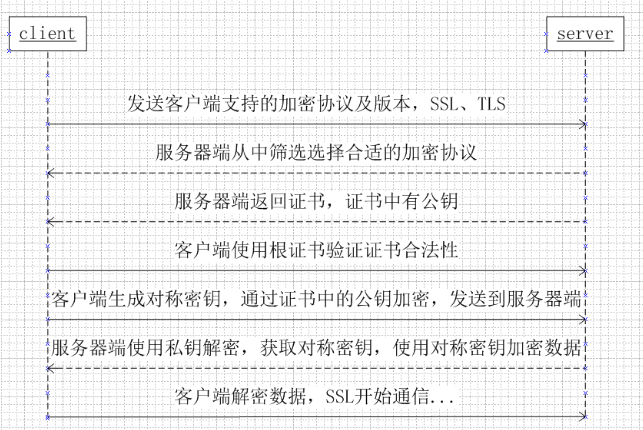
**四次挥手(断开连接)**

* 第一次： TCP客户端发送一个FIN，用来关闭客户到服务器的数据传送。
* 第二次：服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。
* 第三次：服务器关闭客户端的连接，发送一个FIN给客户端。
* 第四次：客户端发回ACK报文确认，并将确认序号设置为收到序号加1。

**HTTP请求过程**

* 建立连接完毕以后客户端会发送响应给服务端
* 服务端接受请求并且做出响应发送给客户端
* 客户端收到响应并且解析响应响应给客户

**HTTPS**



* 在使用HTTPS是需要保证服务端配置正确了对应的安全证书
* 客户端发送请求到服务端
* 服务端返回公钥和证书到客户端
* 客户端接收后会验证证书的安全性,如果通过则会随机生成一个随机数,用公钥对其加密,发送到服务端
* 服务端接受到这个加密后的随机数后会用私钥对其解密得到真正的随机数,随后用这个随机数当做私钥对需要发送的数据进行对称加密
* 客户端在接收到加密后的数据使用私钥(即生成的随机值)对数据进行解密并且解析数据呈现结果给客户
* SSL加密建立