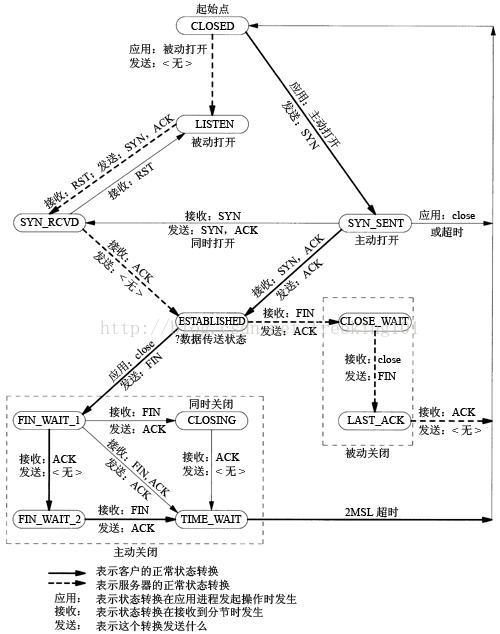
**TCP十一种状态**



**全部11种状态**

**1. 客户端独有的：（1）SYN\_SENT （2）FIN\_WAIT1 （3）FIN\_WAIT2 （4）CLOSING （5）TIME\_WAIT 。**

**2. 服务器独有的：（1）LISTEN （2）SYN\_RCVD （3）CLOSE\_WAIT （4）LAST\_ACK 。**

**3. 共有的：（1）CLOSED （2）ESTABLISHED 。**

**各个状态的意义如下：**

1. LISTEN - 侦听来自远方TCP端口的连接请求；
2. SYN-SENT - 在发送连接请求后等待匹配的连接请求；
3. SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；
4. ESTABLISHED - 代表一个打开的连接，数据可以传送给用户；
5. FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；
6. FIN-WAIT-2 - 从远程TCP等待连接中断请求；
7. CLOSE-WAIT - 等待从本地用户发来的连接中断请求；
8. CLOSING - 等待远程TCP对连接中断的确认；
9. LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；
10. TIME-WAIT - 等待足够的时间以确保远程TCP接收到连接中断请求的确认；
11. CLOSED - 没有任何连接状态；

**各个状态详解：**

**LISTENING ：侦听来自远方的TCP端口的连接请求 .**

1. 首先服务端需要打开一个 socket 进行监听，状态为LISTEN。
2. 有提供某种服务才会处于LISTENING状态， TCP状态变化就是某个端口的状态变化，提供一个服务就打开一个端口，
3. 例如：提供www服务默认开的是80端口，提供ftp服务默认的端口为21，当提供的服务没有被连接时就处于LISTENING状态。
4. FTP服务启动后首先处于侦听(LISTENING)状态。处于侦听LISTENING状态时，该端口是开放的，等待连接，但还没有被连接。就像你房子的门已经敞开的，但还没有人进来。
5. 看LISTENING状态最主要的是看本机开了哪些端口，这些端口都是哪个程序开的，关闭不必要的端口是保证安全的一个非常重要的方面，
6. 服务端口都对应一个服务(应用程序)，停止该服务就关闭了该端口，例如要关闭21端口只要停止IIS服务中的FTP服务即可。关于这方面的知识请参阅其它文章。
7. 如果你不幸中了服务端口的木马，木马也开个端口处于LISTENING状态。

**SYN-SENT： 客户端SYN\_SENT状态：**

1. 在客户端发送连接请求后，等待匹配的连接请求:
2. 客户端通过应用程序调用connect进行active open.于是客户端tcp发送一个SYN以请求建立一个连接.之后状态置为SYN\_SENT.
3. */\*The socket is actively attempting to establish a connection. 在发送连接请求后等待匹配的连接请求 \*/*
5. 当请求连接时客户端首先要发送同步信号给要访问的机器，此时状态为SYN\_SENT，如果连接成功了就变为ESTABLISHED，正常情况下SYN\_SENT状态非常短暂。
6. 例如要访问网站http:*//www.baidu.com,如果是正常连接的话，*
7. 用TCPView观察 IEXPLORE .EXE(IE)建立的连接会发现很快从SYN\_SENT变为ESTABLISHED，表示连接成功。SYN\_SENT状态快的也许看不到。
9. 如果发现有很多SYN\_SENT出现，那一般有这么几种情况，
10. 一是你要访问的网站不存在或线路不好，
11. 二是用扫描软件扫描一个网段的机器，也会出出现很多SYN\_SENT，
12. 另外就是可能中了病毒了，例如中了"冲击波"，病毒发作时会扫描其它机器，这样会有很多SYN\_SENT出现。

**SYN-RECEIVED： 服务器端状态SYN\_RCVD**

1. 再收到和发送一个连接请求后等待对方对连接请求的确认。当服务器收到客户端发送的同步信号时，将标志位ACK和SYN置1发送给客户端，此时服务器端处于SYN\_RCVD状态，
2. 如果连接成功了就变为ESTABLISHED，正常情况下SYN\_RCVD状态非常短暂。如果发现有很多SYN\_RCVD状态，那你的机器有可能被SYN Flood的DoS(拒绝服务攻击)攻击了。

**SYN Flood的攻击原理是：**

1. 在进行三次握手时，攻击软件向被攻击的服务器发送SYN连接请求(握手的第一步)，但是这个地址是伪造的，
2. 如攻击软件随机伪造了51.133.163.104、65.158.99.152等等地址。 服务器 在收到连接请求时将标志位 ACK和 SYN 置1发送给客户端(握手的第二步)，
3. 但是这些客户端的IP地址都是伪造的，服务器根本找不到客户机，也就是说握手的第三步不可能完成。
4. 这种情况下服务器端一般会重试(再次发送SYN+ACK给客户端)并等待一段时间后丢弃这个未完成的连接，这段时间的长度我们称为SYN Timeout，
5. 一般来说这个时间是分钟的数量级(大约为30秒-2分钟);
6. 一个用户出现异常导致服务器的一个线程等待1分钟并不是什么很大的问题，
7. 但如果有一个恶意的攻击者大量模拟这种情况，服务器端将为了维护一个非常大的半连接列表而消耗非常多的资源----数以万计的半连接，
8. 即使是简单的保存并遍历也会消耗非常多的 CPU 时间和内存，何况还要不断对这个列表中的IP进行SYN+ACK的重试。
9. 此时从正常客户的角度看来，服务器失去响应，这种情况我们称做： 服务器端受到了SYN Flood攻击(SYN洪水攻击 )

**ESTABLISHED：代表一个打开的连接。**

1. ESTABLISHED状态是表示两台机器正在传输数据，观察这个状态最主要的就是看哪个程序正在处于ESTABLISHED状态。
2. 服务器出现很多 ESTABLISHED状态： netstat -nat |grep 9502或者使用lsof -i:9502可以检测到。
3. 当客户端未主动close的时候就断开连接：即客户端发送的FIN丢失或未发送。
4. 这时候若客户端断开的时候发送了FIN包，则服务端将会处于CLOSE\_WAIT状态;
5. 这时候若客户端断开的时候未发送FIN包，则服务端处还是显示ESTABLISHED状态;
6. 结果客户端重新连接服务器。
7. 而新连接上来的客户端(也就是刚才断掉的重新连上来了)在服务端肯定是ESTABLISHED; 如果客户端重复的上演这种情况，
8. 那么服务端将会出现大量的假的ESTABLISHED连接和CLOSE\_WAIT连接。
9. 最终结果就是新的其他客户端无法连接上来，但是利用netstat还是能看到一条连接已经建立，并显示ESTABLISHED，但始终无法进入程序代码。

**FIN-WAIT-1： 等待远程TCP连接中断请求，或先前的连接中断请求的确认**

1. 主动关闭(active close)端应用程序调用close，于是其TCP发出FIN请求主动关闭连接，之后进入FIN\_WAIT1状态.
2. */\* The socket is closed, and the connection is shutting down. 等待远程TCP的连接中断请求，或先前的连接中断请求的确认 \*/*

**FIN-WAIT-2：从远程TCP等待连接中断请求**

主动关闭端接到ACK后，就进入了FIN-WAIT-2 ./\* Connection is closed, and the socket is waiting for a shutdown from the remote end. 从远程TCP等待连接中断请求 \*/

**这就是著名的半关闭的状态了**，这是在关闭连接时，客户端和服务器两次握手之后的状态。在这个状态下，应用程序还有接受数据的能力，但是已经无法发送数据，但是也有一种可能是，客户端一直处于FIN\_WAIT\_2状态，而服务器则一直处于WAIT\_CLOSE状态，而直到应用层来决定关闭这个状态。

**CLOSE-WAIT：等待从本地用户发来的连接中断请求**

socket端口如何解除bind：<http://blog.csdn.net/awu999328/article/details/24473033>

linux 下端口close\_wait 过多：<http://blog.csdn.net/natureice/article/details/6439226>

被动关闭(passive close)端TCP接到FIN后，就发出ACK以回应FIN请求(它的接收也作为文件结束符传递给上层应用程序),并进入CLOSE\_WAIT. /\* The remote end has shut down, waiting for the socket to close. 等待从本地用户发来的连接中断请求 \*/

对方主动关闭连接或者网络异常导致连接中断，这时我方的状态会变成CLOSE\_WAIT 此时我方要调用close()来使得连接正确关闭

**CLOSING：等待远程TCP对连接中断的确认**

比较少见./\* Both sockets are shut down but we still don't have all our data sent. 等待远程TCP对连接中断的确认 \*/

**LAST-ACK：等待原来的发向远程TCP的连接中断请求的确认**

被动关闭端一段时间后，接收到文件结束符的应用程序将调用CLOSE关闭连接。这导致它的TCP也发送一个 FIN,等待对方的ACK.就进入了LAST-ACK . /\* The remote end has shut down, and the socket is closed. Waiting for acknowledgement. 等待原来发向远程TCP的连接中断请求的确认 \*/

**TIME-WAIT：等待足够的时间以确保远程TCP接收到连接中断请求的确认**

在主动关闭端接收到FIN后，TCP就发送ACK包，并进入TIME-WAIT状态。/\* The socket is waiting after close to handle packets still in the network.等待足够的时间以确保远程TCP接收到连接中断请求的确认 \*/  
  
TIME\_WAIT等待状态，这个状态又叫做2MSL状态，说的是在TIME\_WAIT2发送了最后一个ACK数据报以后，要进入TIME\_WAIT状态，这个状态是防止最后一次握手的数据报没有传送到对方那里而准备的(注意这不是四次握手，这是第四次握手的保险状态)。这个状态在很大程度上保证了双方都可以正常结束，但是，问题也来了。  
  
由于插口的2MSL状态(插口是IP和端口对的意思，socket)，使得应用程序在2MSL时间内是无法再次使用同一个插口的，对于客户程序还好一些，但是对于服务程序，例如httpd，它总是要使用同一个端口来进行服务，而在2MSL时间内，启动httpd就会出现错误(插口被使用)。为了避免这个错误，服务器给出了一个平静时间的概念，这是说在2MSL时间内，虽然可以重新启动服务器，但是这个服务器还是要平静的等待2MSL时间的过去才能进行下一次连接。

我方主动调用close()断开连接，收到对方确认后状态变为TIME\_WAIT。TCP协议规定TIME\_WAIT状态会一直持续2MSL(即两倍的分段最大生存期)，以此来确保旧的连接状态不会对新连接产生影响。处于TIME\_WAIT状态的连接占用的资源不会被内核释放，所以作为服务器，在可能的情况下，尽量不要主动断开连接，以减少TIME\_WAIT状态造成的资源浪费。

**CLOSED：没有任何连接状态**

被动关闭端在接受到ACK包后，就进入了closed的状态。连接结束./\* The socket is not being used. 没有任何连接状态 \*/

**TCP状态迁移**

大家对netstat -a命令很熟悉，但是，你注意到STATE一栏没，基本上显示着established,time\_wait,close\_wait等，这些到底是 什么意思呢？  
大家很明白TCP初始化连接三次握手吧：发SYN包，然后返回SYN/ACK包，再发ACK包，连接正式建立。但是这里有点出入，当请求者收到SYS /ACK包后，就开始建立连接了，而被请求者第三次握手结束后才建立连接。但是大家明白关闭连接的工作原理吗？关闭连接要四次握手：发FIN包，ACK 包，FIN包，ACK包，四次握手！！为什么呢，因为TCP连接是全双工，我关了你的连接，并不等于你关了我的连接。

客户端正常情况下TCP状态迁移：

|  |
| --- |
| CLOSED->SYN\_SENT->ESTABLISHED->FIN\_WAIT\_1->FIN\_WAIT\_2->TIME\_WAIT->CLOSED |

服务器正常情况下TCP状态迁移：

|  |
| --- |
| CLOSED->LISTEN->SYN收到 ->ESTABLISHED->CLOSE\_WAIT->LAST\_ACK->CLOSED |

当客户端开始连接时，服务器还处于LISTENING，  
客户端发一个SYN包后，他就处于SYN\_SENT状态,服务器就处于SYS收到状态,  
然后互相确认进入连接状态ESTABLISHED.  
当客户端请求关闭连接时,客户端发送一个FIN包后,客户端就进入FIN\_WAIT\_1状态,等待对方的确认包,  
服务器发送一个ACK包给客户,客户端收到ACK包后结束FIN\_WAIT\_1状态,进入FIN\_WAIT\_2状态,等待服务器发过来的关闭请求,  
服务器发一个FIN包后,进入CLOSE\_WAIT状态,  
当客户端收到服务器的FIN包,FIN\_WAIT\_2状态就结束,然后给服务器端的FIN包给以一个确认包,客户端这时进入TIME\_WAIT,  
当服务器收到确认包后,CLOSE\_WAIT状态结束了,  
这时候服务器端真正的关闭了连接.但是客户端还在TIME\_WAIT状态下,  
什么时候结束呢.我在这里再讲到一个新名词:2MSL等待状态,其实TIME\_WAIT就是2MSL等待状态,  
为什么要设置这个状态,原因是有足够的时间让ACK包到达服务器端,如果服务器端没收到ACK包，超时了，然后重新发一个FIN包，直到服务器收到ACK 包.  
TIME\_WAIT状态等待时间是在TCP重新启动后不连接任何请求的两倍.  
大家有没有发现一个问题:如果对方在第三次握手的时候出问题,如发FIN包的时候,不知道什么原因丢了这个包,然而这边一直处在FIN\_WAIT\_2状 态,而且TCP/IP并没有设置这个状态的过期时间,那他一直会保留这个状态下去,越来越多的FIN\_WAIT\_2状态会导致系统崩溃.  
上面我碰到的这个问题主要因为TCP的结束流程未走完，造成连接未释放。现设客户端主动断开连接，流程如下:

|  |
| --- |
| Client 消息 Server  close() ------ FIN -------> FIN\_WAIT1 CLOSE\_WAIT <----- ACK ------- FIN\_WAIT2  close() <------ FIN ------  TIME\_WAIT LAST\_ACK   ------ ACK ------->  CLOSED CLOSED |

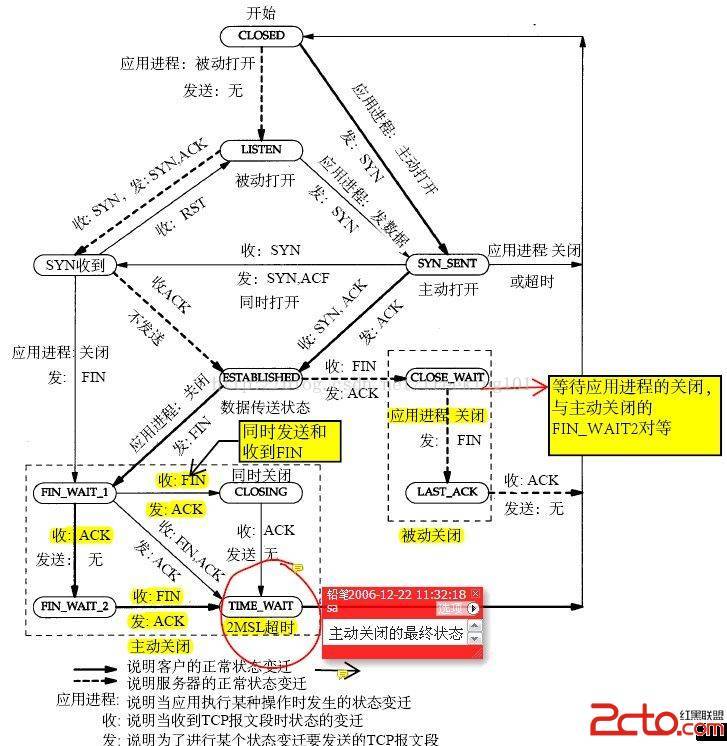
由于Server的Socket在客户端已经关闭时而没有调用关闭，造成服务器端的连接处在“挂起”状态，而客户端则处在等待应答的状态上。  
此问题的典型特征是：一端处于FIN\_WAIT2 ，而另一端处于CLOSE\_WAIT。不过，根本问题还是程序写的不好，有待提高

-------------------------------------------------------------------------  
CLOSE\_WAIT，TCP的癌症，TCP的朋友。  
CLOSE\_WAIT状态的生成原因  
首先我们知道，如果我们的服务器程序APACHE处于CLOSE\_WAIT状态的话，说明套接字是被动关闭的！  
因为如果是CLIENT端主动断掉当前连接的话，那么双方关闭这个TCP连接共需要四个packet：  
Client ---> FIN ---> Server  
Client <--- ACK <--- Server  
这时候Client端处于FIN\_WAIT\_2状态；而Server 程序处于CLOSE\_WAIT状态。  
Client <--- FIN <--- Server  
这时Server 发送FIN给Client，Server 就置为LAST\_ACK状态。  
Client ---> ACK ---> Server  
Client回应了ACK，那么Server 的套接字才会真正置为CLOSED状态。  
Server 程序处于CLOSE\_WAIT状态，而不是LAST\_ACK状态，说明还没有发FIN给Client，那么可能是在关闭连接之前还有许多数据要发送或者其 他事要做，导致没有发这个FIN packet。  
通常来说，一个CLOSE\_WAIT会维持至少2个小时的时间。如果有个流氓特地写了个程序，给你造成一堆的 CLOSE\_WAIT，消耗你的资源，那么通常是等不到释放那一刻，系统就已经解决崩溃了。  
只能通过修改一下TCP/IP的参数，来缩短这个时间：修改tcp\_keepalive\_\*系列参数有助于解决这个 问题。  
解决这个问题的方法是修改系统的参数，系统默认超时时间的是7200秒，也就是2小时， 这个太大了，可以修改如下几个参数：

|  |
| --- |
| sysctl -w net.ipv4.tcp\_keepalive\_time=30 sysctl -w net.ipv4.tcp\_keepalive\_probes=2 sysctl -w net.ipv4.tcp\_keepalive\_intvl=2 |

然后，执行sysctl命令使修改生效。  
连接进程是通过一系列状态表示的，这些状态有：

LISTEN，SYN-SENT，SYN-RECEIVED，ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE- WAIT，CLOSING，LAST-ACK，TIME-WAIT和CLOSED



**1、建立连接协议（三次握手）**

* （1） 客户 端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的报文1。
* （2） 服务器端回应客户端的，这是三次握手中的第2个报文，这个报文同时带ACK标志和SYN标 志。因此它表示对刚才客户端SYN报文的回应；同时又标志SYN给客户端，询问客户端是否准备好进行数据通 讯。
* （3）  客户必须再次回应服务段一个ACK报文，这是报文段3。

**2、连接终止协议（四次握手）**

　 　由于TCP连 接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终 止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接 在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

* （1） TCP客 户端发送一个FIN，用来关闭客户到服务器的数据传送（报文段4）。
* （2） 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（报文段5）。和SYN一 样，一个FIN将占用一个序号。
* （3） 服务器关闭客户端的连接，发送一个FIN给客户端（报文段6）。
* （4） 客户段发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

1. CLOSED: 这个没什么好说的了，表示初始状态。
2. LISTEN: 这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处 于监听状态，可以接受连接了。
3. SYN\_RCVD: 这个状态表示接受到了SYN报 文，
4. 在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂，
5. 用netstat很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次TCP握手 过程中最后一个ACK报文不予发送。
6. 因此这种状态时，当收到客户端的ACK报文 后，它会进入到ESTABLISHED状态。
7. SYN\_SENT: 这个状态与SYN\_RCVD遥想呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，
8. 因此也随即它会进入到了SYN\_SENT状态，并等待服务端的发送三次握手中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。
9. ESTABLISHED：这个容易理解了，表示连接已经建立了。
10. FIN\_WAIT\_1: 这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报 文。
11. 而这两种状态的区别是：FIN\_WAIT\_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，
12. 此时该SOCKET即进入到FIN\_WAIT\_1状态。而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态，当然在实际的正常情况 下，
13. 无论对方何种情况下，都应该马上回应ACK报文，所以FIN\_WAIT\_1状态一般是比较难见到的，
14. 而FIN\_WAIT\_2状态还有时常常可以用netstat看到。
15. FIN\_WAIT\_2：上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示半连接，也即有一方要求close连接，
16. 但另外还告诉对方，我暂时还有点 数据需要传送给你，稍后再关闭连接。
17. TIME\_WAIT: 表示收到了对方的FIN报 文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。
18. 如果FIN\_WAIT\_1状态下收到了对方同时带FIN标志和ACK标志的报文，则直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。
19. CLOSING: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。
20. 正常情况下，当你发 送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报 文，再收到对方的FIN报文。
21. 但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报 文，反而却也收到了对方的FIN报文。
22. 什么情况下会出现此种情况呢？
23. 其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一 个SOCKET的话，
24. 那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。
25. CLOSE\_WAIT: 这种状态的含义其实是表示在等待关闭。
26. 怎么理解呢？当对方close一 个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文 给对方，
27. 此时则进入到CLOSE\_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，
28. 如果没有的话， 那么你也就可以close这个SOCKET，发送FIN报文给对方，也即关闭连接。
29. 所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。
30. LAST\_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报 文后，最后等待对方的ACK报文。
31. 当收到ACK报文后，也即可以进入到CLOSED可用状态了。

最后有2个问题 的回答，我自己分析后的结论（不一定保证100%正确）

1、 为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起 应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文 通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文 和FIN报文多数情况下都是分开发送的。

2、 为什么TIME\_WAIT状态还需要等2MSL后才能返回到CLOSED状 态？

这是因为：虽然双方 都同意关闭连接了，而且握手的4个报文也都协调和发送完毕，按理可以直接回到CLOSED状 态（就好比从SYN\_SEND状态到ESTABLISH状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的ACK报 文会一定被对方收到，因此对方处于LAST\_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报 文，所以这个TIME\_WAIT状态的作用就是用来重发可能丢失的ACK报 文，并保证于此。

     断开连接的时候， 当发起主动关闭的左边这方发送一个FIN过去后，

右边被动关闭的这方要回应一个ACK，这个ACK是TCP回应的，而不是应用程序发送的，

此时，被动关闭的一方就处于CLOSE\_WAIT状态了。

如果此时被动关闭的这一方不再继续调用closesocket,那么他就不会发送接下来的FIN，导致自己老是处于CLOSE\_WAIT。

只有被动关闭的这一方调用了 closesocket,才会发送一个FIN给主动关闭的这一方，同时也使得自己的状态变迁为LAST\_ACK。

比如被动关闭的是客户端

当对方调用closesocket的时候，你的程序正在

1. int nRet = recv(s,....);
2. if (nRet == SOCKET\_ERROR)
3. {
4. *// closesocket(s);*
5. return FALSE;
6. }

很多人就是忘记了那句closesocket，这种代码太常见了。   
我的理解，  
当主动关闭的一方发送FIN到被动关闭这边后，被动关闭这边的TCP马上回应一个ACK过去，同时向上面应用程序提交一个ERROR，  
导致上面的SOCKET的send或者recv返回SOCKET\_ERROR.  
正常情况下，如果上面在返回SOCKET\_ERROR后调用了closesocket, 那么被动关闭的者一方的TCP就会发送一个FIN过去，自己的状态就变迁到LAST\_ACK.  
服务器上出现大量的close\_wait的例子和解决方法（例子从网上找的，基本差不多）

1. $ /usr/sbin/lsof -i | grep 6800
2. $ /usr/sbin/lsof -i | grep 6800
3. Oracle 22725 oracle9i 3u IPv4 18621468 TCP RHEL3:6800 (LISTEN)
4. oracle 22725 oracle9i 4u IPv4 18621469 TCP RHEL3:6800->RHEL3:2174 (CLOSE\_WAIT)
5. oracle 22725 oracle9i 8u IPv4 18621568 TCP RHEL3:6800->RHEL3:2175 (CLOSE\_WAIT)
6. oracle 22725 oracle9i 9u IPv4 18621578 TCP RHEL3:6800->RHEL3:2176 (CLOSE\_WAIT)
7. oracle 22726 oracle9i 3u IPv4 18621468 TCP RHEL3:6800 (LISTEN)
8. oracle 22726 oracle9i 4u IPv4 18621469 TCP RHEL3:6800->RHEL3:2174 (CLOSE\_WAIT)
9. oracle 22726 oracle9i 8u IPv4 18621568 TCP RHEL3:6800->RHEL3:2175 (CLOSE\_WAIT)
10. oracle 22726 oracle9i 9u IPv4 18621578 TCP RHEL3:6800->RHEL3:2176 (CLOSE\_WAIT)
11. $ kill -9 22725
12. *# 22725, 22726就是使用该6800端口的进程号(PID)。*
13. $ /usr/sbin/lsof -i | grep 6800

进程被kill时，会释放占用的所有链接句柄。   
该问题的出现原因网上到处都是，也就是Socket的Client端出现异常没有Close就退出了。

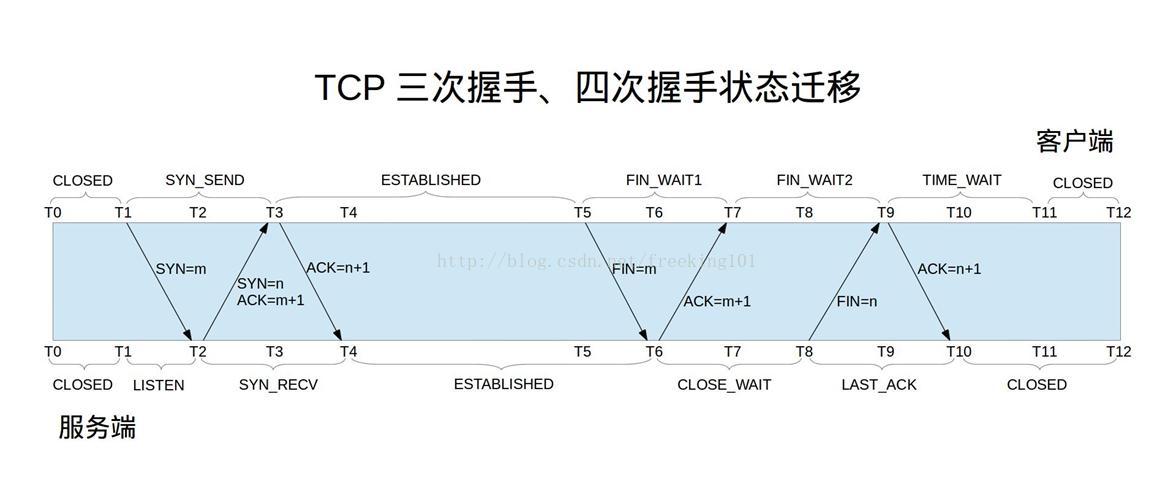
**TCP连接过程是状态的转换，促使发生状态转换的是用户调用**

1. TCP三次握手和四次握手的状态迁移
2. 在任意时刻发生丢包或者重复包时，TCP/IP的处理策略
3. Linux系统调用对TCP/IP可以进行哪些设置，主要针对哪些方面的优化

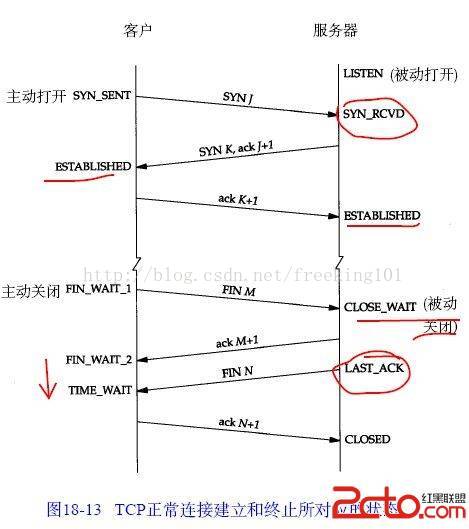
**TCP基本知识点**

1. TCP由RFC793、RFC1122、RFC1323、RFC2001、RFC2018以及RFC2581定义
2. TCP提供可靠性保证
3. TCP发送数据后，要求对方返回确认，如果没有收到确认，TCP会进行重传，数次重传失败后，TCP才会放弃
4. TCP含有动态估算RTT（round-trip time）的算法，可以根据网络拥塞情况动态调整RTT，重新传等待时间就是使用RTT来确定的
5. TCP通过给所发送数据的每一个字节关联一个序列号进行排序，从而处理分包非顺序到达和重复包的情况
6. TCP提供流量控制。TCP总能告诉对方自己还能接收多少字节的数据（advertised window——通告窗口），防止接收缓冲区溢出。窗口随着数据的到来和从缓冲区中取走数据而动态变化。
7. TCP是全双工的。所以TCP必须跟踪每个方向数据流的状态信息（如序列号和通告窗口的大小）

**TCP三次握手和四次握手的状态迁移**



上面的状态迁移图，基本上把TCP三次握手和四次握手的大致流程描述的非常清楚了，下面我们用文字将上面的过程描述一遍，并对异常情况进行分析：



**三次握手概述：**

1. 服务器主动进入LISTEN状态，监听端口
2. 客户发送第一次握手请求，发送完毕后进入SYN\_SEND状态，等待服务器响应
3. 服务器收到第一次握手请求，向客户确认第一次请求，连带发送第二次握手请求，发送完毕后进入SYN\_RECV状态，等待客户响应
4. 客户收到确认和第二次握手请求，对第二次握手请求进行确认（第三次握手），发送确认完毕后，进入ESTABLISHED状态
5. 服务器收到对第二次握手请求的确认之后（第三次握手），进入ESTABLISHED状态
6. 至此，三次握手完成，客户-服务器完成连接的建立，开始数据通信

**三次握手和编程的关联：**

1. 服务器通过socket()、bind()和listen()来完成CLOSED状态到LISTEN状态的转化，称为被动打开。被动打开完成之后，accept()阻塞，等待客户请求
2. 客户通过connect()进行主动打开。这引起客户TCP发送一个SYN分节，用于通知服务器客户将在连接中发送数据的初始序列号（一般SYN分节不包含任何数据，只有TCP和IP的头部信息）
3. 服务器以单个分节，同时对客户的SYN序列号进行确认，并发送自己的SYN序列号（此时accept()还在阻塞中）
4. 客户对服务器的SYN数据进行确认。客户在收到服务器SYN并进行确认之后，connect()返回
5. 服务器收到客户的确认，accept()返回

**三次握手时的异常：**

1. 第一次握手丢包：默认情况下，connect()是阻塞式的，如果请求无法发送到服务器，那么connect会进行一段很长时间的等待和重试（重传次数和时间间隔我们暂且不去深究），此时我们可以使用通过设置SO\_SNDTIMEO来为connect设置超时以减少connect的等待时间
2. 第二次握手丢包：对于客户来说，依然是connect超时，所以处理方式和第一次握手丢包是一样的。对于服务器来说，由于收不到第三次握手请求，所以会进行等待重传，直到多次重传失败后，关闭半连接。  
   1.  这里需要提一下的是，服务器会维护一个半连接队列，用于等待客户的第三次握手请求。当收到第三次握手请求或者多次重传失败后，服务器会将该半连接从队列中删除。（这里暂且不去深究半连接队列的等待重新策略和配置）  
   2.  我们经常听说的DDos攻击，就可以这个环节实现，syn flood就是一种常见的DDos攻击方式。简单来说，syn flood就是只发送第一次握手请求后，就关闭连接，将服务器的半连接队列占满，从而让正常用户无法得到服务。
3. 第三次握手丢包：由于客户在发送第三次握手包后，不再等待确认，就直接进入了ESTABLISHED状态，所以一旦第三次握手失败，客户和服务器的状态就不同步了。当然，此时服务器会进行多次重发，一旦客户再次收到SYN+ACK（第二次握手请求），会再次确认。不过，如果第三次握手一直失败，则会出现，客户已经建立连接，而服务器关闭连接的情况。随后，一旦客户向服务器发送数据，则会收到一条RST回应，告诉用户连接已经重置，需要重新进行三次握手。  
           RST和SIGPIPE：有过网络编程经验的人都知道在写网络通信的时候，需要屏蔽SIGPIPE信号，否则的话，一旦收到PIPE信号会导致程序异常退出。其实这个SIGPIPE就是由于write()的时候，我们自己的状态是ESTABLISHED而对方的状态不是ESTABLISHED，那么对方就会给我们一个RST回应，收到这个回应之后，系统就会自动生成一个PIPE信号。

**四次挥手概述：**

1. 客户发送FIN请求（第一次握手），通知关闭连接，然后进入FIN\_WAIT1状态
2. 服务器收到FIN请求后，发送ACK（第二次握手），对客户的FIN进行确认，然后进入CLOSE\_WAIT状态
3. 服务器进行一些收尾工作，然后主动相客户发送FIN请求（第三次握手），通知关闭连接，然后进入LAST\_ACK状态
4. 客户收到FIN，对FIN进行确认（第四次握手），并进入TIME\_WAIT状态
5. 服务器收到客户的确认，关闭连接
6. 客户等待一段时间后，关闭连接

**四次握手和编程的关联：**

1. 客户调用close()执行主动关闭，发送FIN到服务器，FIN表示不会再发送数据了
2. 服务器收到FIN进行被动关闭，由TCP对FIN进行确认。FIN作为文件结束符，传递给recv()。因为收到FIN以后就意味着不会再有数据了
3. 一段时间后，服务器调用close()关闭自己的socket，并发送FIN给客户，宣告自己不会再发送数据了
4. 客户收到FIN后，不再确认，等待一段时间后，自行关闭自己的socket

说明：

1. TCP是全双工的连接，所以关闭的过程必须是两个方向都关闭才行，这也就是为什么需要两次不同方向的FIN
2. FIN并不像SYN一样，一定是一个独立的包，有时FIN会随着数据一起发送，而对方也有可能将ACK和FIN放在一个包中进行发送，这成为捎带。捎带的机制在数据传输中也会出现。
3. 四次握手的过程不像三次握手一样，一定是由客户发起。虽然一般来说，是由客户发起，但是某些协议（例如HTTP）则是服务器执行主动关闭

**两个WAIT：**

1. CLOSE\_WAIT：CLOSE\_WAIT的状态位于向对方确认FIN之后，向对方发送FIN之前，这段时间由于对方已经发送了FIN，也就表示不会再收到数据，但是这并不表示自己没有数据要发，毕竟只有在发送了FIN之后，才表示发送完毕。所以，CLOSE\_WAIT这段时间主要的工作就是给对方发送必要的数据，对自己的数据进行收尾，所有工作结束之后，调用close()，发送FIN，等待LAST\_ACK
2. TIME\_WAIT：存在TIME\_WAIT状态有如下两个理由：
   1. 实现终止TCP全双工连接的可靠性：假如LAST-ACK丢失，对方重发，但是自己已经关闭连接，那么会返回一个RST包，对放会将其解释为错误，从而无法正常关闭。也就是说，TIME\_WAIT的作用之一就是解决LAST-ACK可能丢包的情况，因为在有些网络不好的情况下，不得不重发LAST-ACK
   2. 允许老的网络分组在网络中消逝：2MSL的时间足够让所有的FIN数据在网络中消失，如果不等待，并立即开始一个新的连接，有可能出现老FIN关闭了新连接的情况，因为在IP和端口一直的情况下，很难区分一个数据包是属于哪一次连接的

**四次握手的异常：**

1. 第一次握手丢包：FIN\_WAIT1丢失会导致客户重传，如果多次重传失败，则客户超时关闭连接，而服务器依然保持ESTABLISHED状态。如果服务器主动发送数据，则会收到一个RST包，重置连接。设置KeepAlive道理相同，核心是要求服务器主动发数据。如果服务器永远不会主动发数据，那么就会一直保持这样一个“假连接”
2. 第二次握手丢包：由于服务器第二次握手不会重发，所以即使丢包也不管，直接向对方发送FIN，此时客户执行”同时关闭“的流程（这个流程后面再说），等待TIME\_WAIT时间后关闭。在客户进入TIME\_WAIT之后，自己由于FIN没有相应，会重发，如果被客户TIME\_WAIT收到并发送LAST-ACK，则流程正常结束，如果反复重发没有响应，那么超时关闭
3. 第三次握手丢包：服务器会持续等待在LAST\_ACK状态，而客户会持续等待在FIN\_WAIT2状态，最后双方超时关闭
4. 第四次握手丢包：客户端进入TIME\_WAIT状态，等待2MSL，服务器由于收不到LAST-ACK则进行重发，如果多次重发失败，则超时关闭（这个流程和第二次握手丢包的后半段状态是一样的）

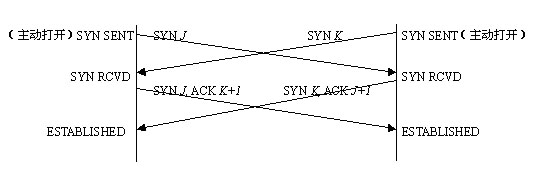
**TCP的同时打开和同时关闭**

除了上面的顺序打开，和顺序关闭方式，TCP还有同时打开和同时关闭的流程：

**同时打开流程**：（引自：http://hi.baidu.com/psorqkxcsfbbghd/item/70f3bd91943b9248f14215cd）

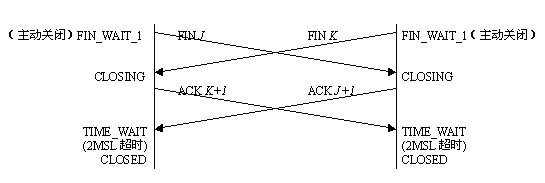
两个应用程序同时执行主动打开的情况是可能的，虽然发生的可能性较低。每一端都发送一个SYN,并传递给对方，且每一端都使用对端所知的端口作为本地端口。

例如：  
主机a中一应用程序使用7777作为本地端口，并连接到主机b 8888端口做主动打开。  
主机b中一应用程序使用8888作为本地端口，并连接到主机a 7777端口做主动打开。  
tcp协议在遇到这种情况时，只会打开一条连接。  
这个连接的建立过程需要4次数据交换，而一个典型的连接建立只需要3次交换(即3次握手)  
但多数伯克利版的tcp/ip实现并不支持同时打开



**同时关闭流程**：（引自：http://hi.baidu.com/psorqkxcsfbbghd/item/70f3bd91943b9248f14215cd）

如果应用程序同时发送FIN，则在发送后会首先进入FIN\_WAIT\_1状态。在收到对端的FIN后，回复一个ACK，会进入CLOSING状态。在收到对端的ACK后，进入TIME\_WAIT状态。这种情况称为同时关闭。  
  
同时关闭也需要有4次报文交换，与典型的关闭相同。



       如果上面的顺序流程已经非常清楚的话，那么这两个同时打开、同时关闭的状态图就不难理解了……

       大家可以通过这两张图来对应上面socket关闭流程中，“第二次握手失败”的解释，其实也就不难理解，为什么客户会进入同时关闭状态了。因为客户在发送了FIN之后，没有等到ACK，而是等到了服务器的FIN，自然符合同步关闭的流程。

**TCP通信中服务器处理客户端意外断开**

如果TCP连接被对方正常关闭，也就是说，对方是正确地调用了closesocket(s)或者shutdown(s)的话，那么上面的Recv或Send调用就能马上返回，并且报错。这是由于close socket(s)或者shutdown(s)有个正常的关闭过程，会告诉对方“TCP连接已经关闭，你不需要再发送或者接受消息了”。  
  
但是，如果意外断开，客户端(3g的移动设备)并没有正常关闭socket。双方并未按照协议上的四次挥手去断开连接。  
  
那么这时候正在执行Recv或Send操作的一方就会因为没有任何连接中断的通知而一直等待下去，也就是会被长时间卡住。  
  
像这种如果一方已经关闭或异常终止连接，而另一方却不知道，我们将这样的TCP连接称为半打开 的。

**解决意外中断办法都是利用保活机制。**而保活机制分又可以让底层实现也可自己实现。

**1、 自己编写心跳包程序**

简单的说也就是在自己的程序中加入一条线程，定时向对端发送数据包，查看是否有ACK，如果有则连接正常，没有的话则连接断开

**2、 启动TCP编程里的keepAlive机制**

**一）双方拟定心跳(自实现)**

一般由客户端发送心跳包，服务端并不回应心跳，只是定时轮询判断一下与上次的时间间隔是否超时(超时时间自己设定)。服务器并不主动发送是不想增添服务器的通信量，减少压力。

但这会出现三种情况：

**情况1.**

客户端由于某种网络延迟等原因很久后才发送心跳(它并没有断)，这时服务器若利用自身设定的超时判断其已经断开，而后去关闭socket。若客户端有重连机制，则客户端会重新连接。若不确定这种方式是否关闭了原本正常的客户端，则在ShutDown的时候一定要选择send,表示关闭发送通道，服务器还可以接收一下，万一客户端正在发送比较重要的数据呢，是不?

**情况2.**

客户端很久没传心跳，确实是自身断掉了。在其重启之前，服务端已经判断出其超时，并主动close，则四次挥手成功交互。

**情况3.**

客户端很久没传心跳，确实是自身断掉了。在其重启之前，服务端的轮询还未判断出其超时，在未主动close的时候该客户端已经重新连接。  
  
这时候若客户端断开的时候发送了FIN包，则服务端将会处于CLOSE\_WAIT状态;  
  
这时候若客户端断开的时候未发送FIN包，则服务端处还是显示ESTABLISHED状态;  
  
而新连接上来的客户端(也就是刚才断掉的重新连上来了)在服务端肯定是ESTABLISHED;这时候就有个问题，若利用轮询还未检测出上条旧连接已经超时(这很正常，timer总有个间隔吧)，而在这时，客户端又重复的上演情况3，那么服务端将会出现大量的假的ESTABLISHED连接和CLOSE\_WAIT连接。  
  
最终结果就是新的其他客户端无法连接上来，但是利用netstat还是能看到一条连接已经建立，并显示ESTABLISHED，但始终无法进入程序代码。个人最初感觉导致这种情况是因为假的ESTABLISHED连接和 CLOSE\_WAIT连接会占用较大的系统资源，程序无法再次创建连接(因为每次我发现这个问题的时候我只连了10个左右客户端却已经有40多条无效连接)。而最近几天测试却发现有一次程序内只连接了2，3个设备，但是有8条左右的虚连接，此时已经连接不了新客户端了。这时候我就觉得我想错了，不可能这几条连接就占用了大量连接把，如果说几十条还有可能。但是能肯定的是，这个问题的产生绝对是设备在不停的重启，而服务器这边又是简单的轮询，并不能及时处理，暂时还未能解决。

**二）利用KeepAlive**

其实keepalive的原理就是TCP内嵌的一个心跳包,

以服务器端为例，如果当前 server 端检测到超过一定时间(默认是 7,200,000 milliseconds ，也就是 2 个小时)没有数据传输，那么会向 client 端发送一个 keep-alive packet (该 keep-alive packet 就是 ACK和 当前 TCP 序列号减一的组合)，此时 client 端应该为以下三种情况之一：

**1. client 端仍然存在，网络连接状况良好。**此时 client 端会返回一个 ACK 。server 端接收到 ACK 后重置计时器(复位存活定时器)，在 2 小时后再发送探测。如果 2 小时内连接上有数据传输，那么在该时间基础上向后推延 2 个小时。

**2. 客户端异常关闭，或是网络断开**。在这两种情况下， client 端都不会响应。服务器没有收到对其发出探测的响应，并且在一定时间(系统默认为 1000 ms )后重复发送 keep-alive packet ，并且重复发送一定次数( 2000 XP 2003 系统默认为 5 次 , Vista 后的系统默认为 10 次)。

**3. 客户端曾经崩溃，但已经重启。**这种情况下，服务器将会收到对其存活探测的响应，但该响应是一个复位，从而引起服务器对连接的终止。对于应用程序来说，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。

全局设置可更改 /etc/sysctl.conf ,加上:  
net.ipv4.tcp\_keepalive\_intvl = 20  
net.ipv4.tcp\_keepalive\_probes = 3  
net.ipv4.tcp\_keepalive\_time = 60  
在程序中设置如下:

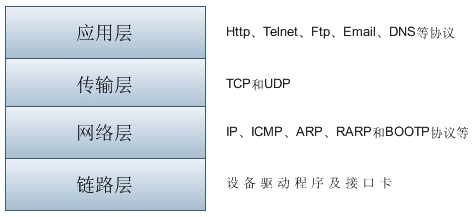
1. #include <sys/socket.h>
2. #include <netinet/in.h>
3. #include <arpa/inet.h>
4. #include <sys/types.h>
5. #include <netinet/tcp.h>
7. int keepAlive = 1; *// 开启keepalive属性*
8. int keepIdle = 60; *// 如该连接在60秒内没有任何数据往来,则进行探测*
9. int keepInterval = 5; *// 探测时发包的时间间隔为5 秒*
10. int keepCount = 3; *// 探测尝试的次数.如果第1次探测包就收到响应了,则后2次的不再发.*
12. setsockopt(rs, SOL\_SOCKET, SO\_KEEPALIVE, (void \*)&keepAlive, sizeof(keepAlive));
13. setsockopt(rs, SOL\_TCP, TCP\_KEEPIDLE, (void\*)&keepIdle, sizeof(keepIdle));
14. setsockopt(rs, SOL\_TCP, TCP\_KEEPINTVL, (void \*)&keepInterval, sizeof(keepInterval));
15. setsockopt(rs, SOL\_TCP, TCP\_KEEPCNT, (void \*)&keepCount, sizeof(keepCount));

在程序中表现为,当tcp检测到对端socket不再可用时(不能发出探测包,或探测包没有收到ACK的响应包),select会返回socket可读,并且在recv时返回-1,同时置上errno为ETIMEDOUT.

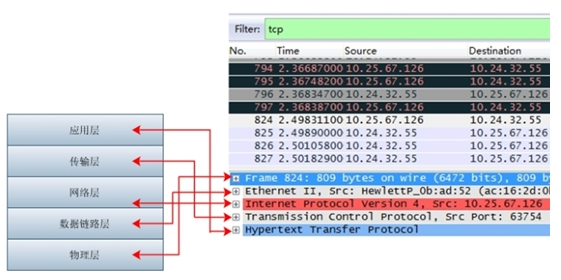
**wireshark 抓包图解 TCP 三次握手四次断开**

**一. TCP/IP协议族**

      TCP/IP是一个协议族，通常分不同层次进行开发，每个层次负责不同的通信功能。包含以下四个层次：



**1. 链路层，**也称作数据链路层或者网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。  
**2. 网络层，**也称作互联网层，处理分组在网络中的活动，例如分组的选路。网络层协议包括IP协议（网际协议）、ICMP协议（Internet互联网控制报文协议），以及IGMP协议（Internet组管理协议）。  
**3. 传输层。**主要为两台主机上的应用程序提供端到端的通信。在TCP/IP协议族中，有两个互不相同的传输协议：TCP（传输控制协议）和UDP（用户数据报协 议）。TCP为两台主机提供高可靠性的数据通信。他所作的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最 后确认分组的超时时钟等。由于运输层提供了高可靠性的端到端通信，因此应用层可以忽略所有这些细节。而另一方面，UDP则为应用层提供一种非常简单的服 务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。任何必须的可靠性必须由应用层来提供。  
**4. 应用层。**负责处理特定的应用程序细节。包括Telnet（远程登录）、FTP（文件传输协议）、SMTP（简单邮件传送协议）以及SNMP（简单网络管理协议）等。  
wireshark抓到的包与对应的协议层如下图所示：



* 1. Frame:   物理层的数据帧概况
* 2. Ethernet II: 数据链路层以太网帧头部信息
* 3. Internet Protocol Version 4: 互联网层IP包头部信息
* 4. Transmission Control Protocol:  传输层的数据段头部信息，此处是TCP
* 5. Hypertext Transfer Protocol:  应用层的信息，此处是HTTP协议

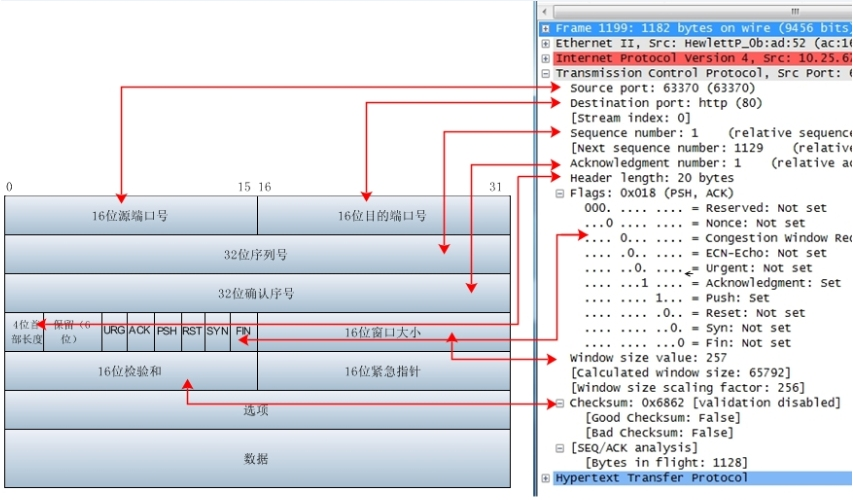
**二. TCP协议**

      TCP是一种面向连接（连接导向）的、可靠的基于字节流的传输层通信协议。TCP将用户数据打包成报文段，它发送后启动一个定时器，另一端收到的数据进行确认、对失序的数据重新排序、丢弃重复数据。  
      TCP的特点有：  
1. TCP是面向连接的运输层协议  
2. 每一条TCP连接只能有两个端点，每一条TCP连接只能是点对点的  
3. TCP提供可靠交付的服务  
4. TCP提供全双工通信。数据在两个方向上独立的进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。  
5. 面向字节流。面向字节流的含义：虽然应用程序和TCP交互是一次一个数据块，但TCP把应用程序交下来的数据仅仅是一连串的无结构的字节流  
      TCP报文首部，如下图所示：



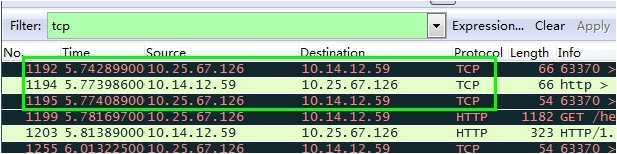
* 1. 源端口号：数据发起者的端口号，16bit
* 2. 目的端口号：数据接收者的端口号，16bit
* 3. 序号：32bit的序列号，由发送方使用
* 4. 确认序号：32bit的确认号，是接收数据方期望收到发送方的下一个报文段的序号，因此确认序号应当是上次已成功收到数据字节序号加1。
* 5. 首部长度：首部中32bit字的数目，可表示15\*32bit=60字节的首部。一般首部长度为20字节。
* 6. 保留：6bit, 均为0
* 7. 紧急URG：当URG=1时，表示报文段中有紧急数据，应尽快传送。
* 8. 确认比特ACK：ACK = 1时代表这是一个确认的TCP包，取值0则不是确认包。
* 9. 推送比特PSH：当发送端PSH=1时，接收端尽快的交付给应用进程。
* 10. 复位比特（RST）：当RST=1时，表明TCP连接中出现严重差错，必须释放连接，再重新建立连接。
* 11. 同步比特SYN：在建立连接是用来同步序号。SYN=1， ACK=0表示一个连接请求报文段。SYN=1，ACK=1表示同意建立连接。
* 12. 终止比特FIN：FIN=1时，表明此报文段的发送端的数据已经发送完毕，并要求释放传输连接。
* 13. 窗口：用来控制对方发送的数据量，通知发放已确定的发送窗口上限。
* 14. 检验和：该字段检验的范围包括首部和数据这两部分。由发端计算和存储，并由收端进行验证。
* 15. 紧急指针：紧急指针在URG=1时才有效，它指出本报文段中的紧急数据的字节数。
* 16. 选项：长度可变，最长可达40字节

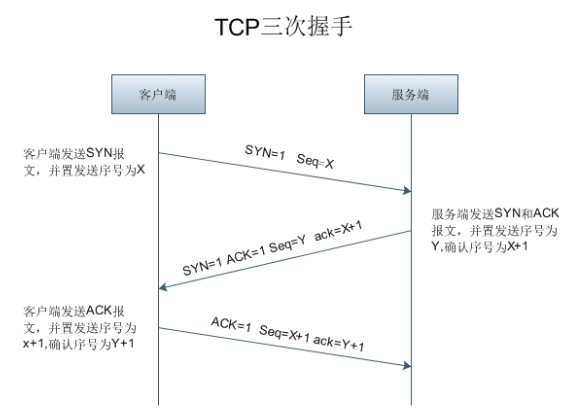
wireshark捕获到的TCP包中的每个字段如下图所示：



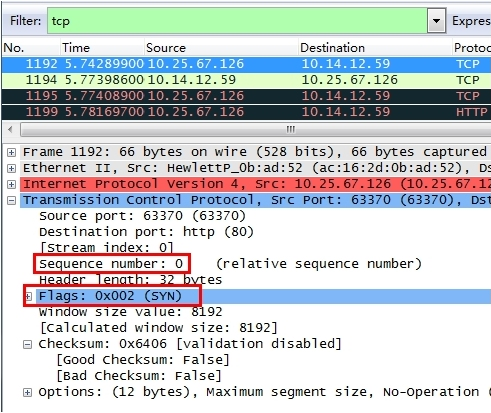
**三. TCP三次握手**

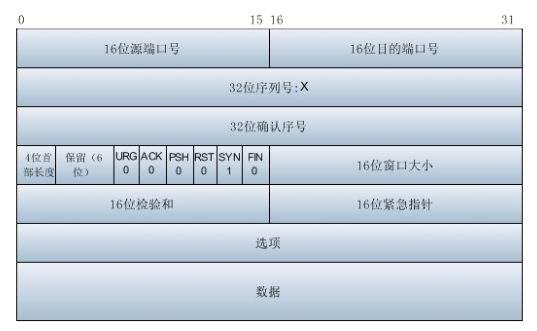
       TCP建立连接时，会有三次握手过程，如下图所示，wireshark截获到了三次握手的三个数据包。第四个包才是http的，说明http的确是使用TCP建立连接的。



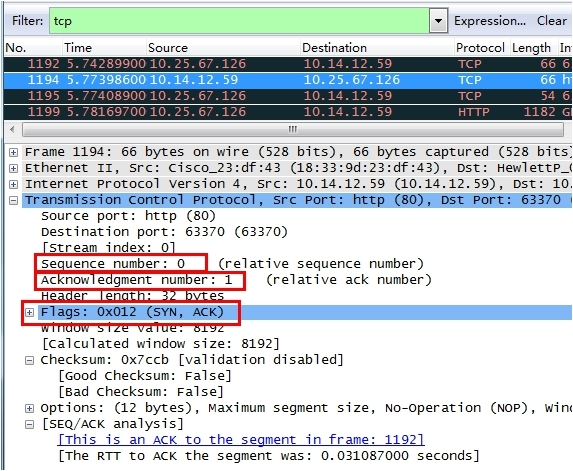


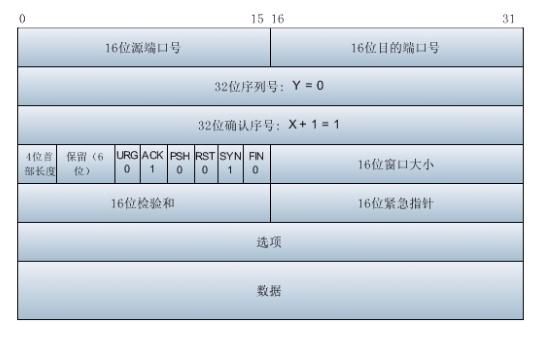
下面来逐步分析三次握手过程：  
第一次握手：客户端向服务器发送连接请求包，标志位SYN（同步序号）置为1，序号为X=0



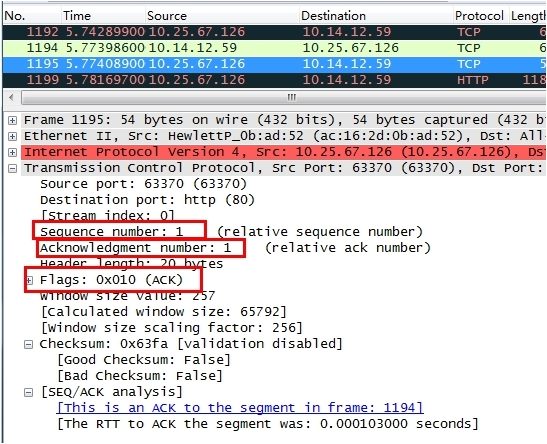


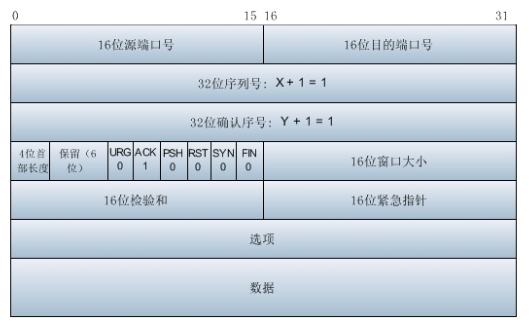
第 二次握手：服务器收到客户端发过来报文，由SYN=1知道客户端要求建立联机。向客户端发送一个SYN和ACK都置为1的TCP报文，设置初始序号 Y=0，将确认序号(Acknowledgement Number)设置为客户的序列号加1，即X+1 = 0+1=1, 如下图：





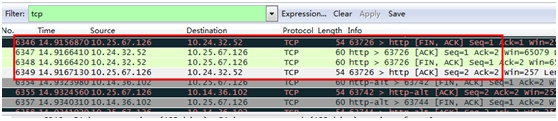
第 三次握手：客户端收到服务器发来的包后检查确认序号(Acknowledgement Number)是否正确，即第一次发送的序号加1（X+1=1）。以及标志位ACK是否为1。若正确，服务器再次发送确认包，ACK标志位为1，SYN标 志位为0。确认序号(Acknowledgement Number)=Y+1=0+1=1，发送序号为X+1=1。客户端收到后确认序号值与ACK=1则连接建立成功，可以传送数据了。

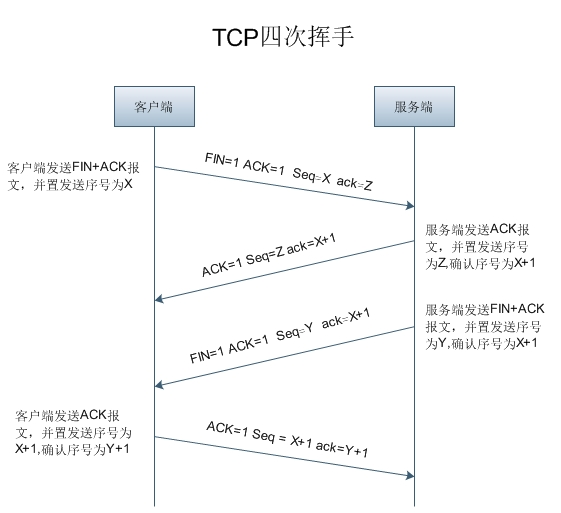




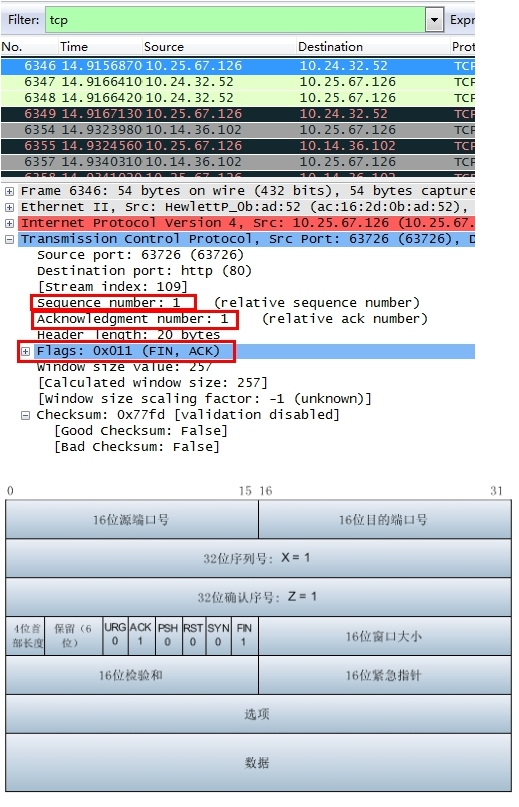
**四. TCP四次挥手**

       TCP断开连接时，会有四次挥手过程，如下图所示，wireshark截获到了四次挥手的四个数据包。

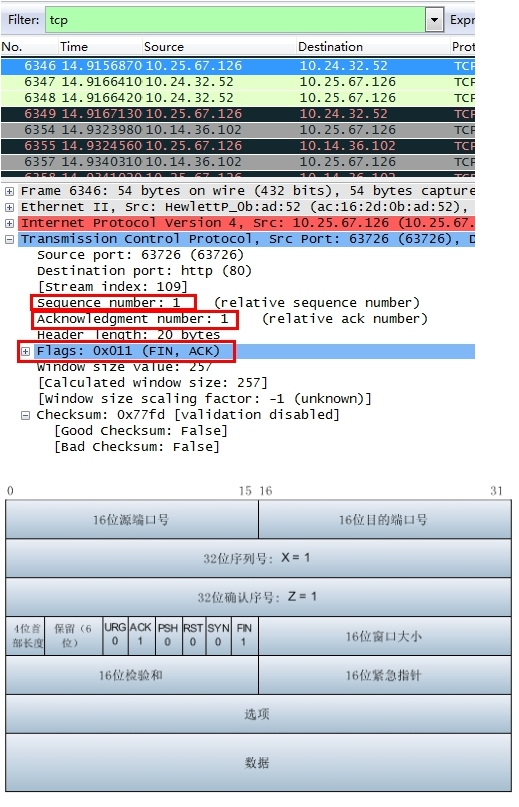




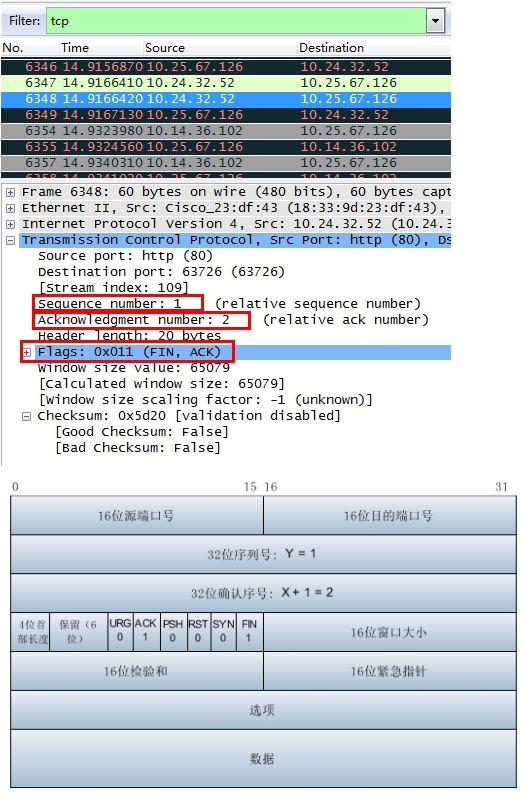
下面来逐步分析四次挥手过程：  
第一次挥手：客户端给服务器发送TCP包，用来关闭客户端到服务器的数据传送。将标志位FIN和ACK置为1，序号为X=1，确认序号为Z=1。



服务器收到FIN后，发回一个ACK(标志位ACK=1),确认序号为收到的序号加1，即X=X+1=2。序号为收到的确认序号=Z。



服务器关闭与客户端的连接，发送一个FIN。标志位FIN和ACK置为1，序号为Y=1，确认序号为X=2。



客户端收到服务器发送的FIN之后，发回ACK确认(标志位ACK=1),确认序号为收到的序号加1，即Y+1=2。序号为收到的确认序号X=2。

