## 一、TCP连接建立，三次握手详细过程

（1）第一次握手：

建立连接时，客户端A发送SYN包（SYN=j）到服务器B，并进入SYN\_SEND状态，等待服务器B确认。

（2）第二次握手：

服务器B收到SYN包，必须确认客户A的SYN（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B进入SYN\_RECV状态。

（3）第三次握手：

客户端A收到服务器B的SYN＋ACK包，向服务器B发送确认包ACK（ACK=k+1），此包发送完毕，客户端A和服务器B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。

确认号：其数值等于发送方的发送序号 +1(即接收方期望接收的下一个序列号)。

## 二、TCP连接断开，四次挥手详细过程

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。这个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。

TCP的连接的拆除需要发送四个包，因此称为四次挥手(four-way handshake)。客户端或服务器均可主动发起挥手动作，在socket编程中，任何一方执行close()操作即可产生挥手操作。

（1）第一次挥手

主动断开连接一方A，发送FIN报文给B，用来关闭A到B的数据传送，并进入FIN-WAIT-1状态，等待对方确认。

（2）第二次挥手

B接收到FIN之后，给它回一个ACK，并进入CLOSE\_TIMEWAIT状态。A接收到之后，进入FIN-WAIT-2状态。

（3）第三次挥手

B发送一个FIN报文给A，用来关闭B到A的数据传送，并进入到LAST\_ACK状态。

（4）第四次挥手

A接收到FIN之后，给B回一个ACK，并进入TIME\_WAIT状态。B收到之后，关闭连接，进入CLOSED状态。TIME\_WAIT状态将持续2个MSL(Max Segment Lifetime最大报文段生存时间)，在Windows下默认为4分钟，没收到B重发的FIN，就关闭socket。

深入理解TCP连接的释放：

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。

TCP协议的连接是全双工连接，一个TCP连接存在双向的读写通道。

简单说来是 “先关读，后关写”，一共需要四个阶段。以客户机发起关闭连接为例：

1.服务器读通道关闭

2.客户机写通道关闭

3.客户机读通道关闭

4.服务器写通道关闭

关闭行为是在发起方数据发送完毕之后，给对方发出一个FIN（finish）数据段。直到接收到对方发送的FIN，且对方收到了接收确认ACK之后，双方的数据通信完全结束，过程中每次接收都需要返回确认数据段ACK。

详细过程：

第一阶段 客户机发送完数据之后，向服务器发送一个FIN数据段，序列号为i；

1.服务器收到FIN(i)后，返回确认段ACK，序列号为i+1，关闭服务器读通道；

2.客户机收到ACK(i+1)后，关闭客户机写通道；

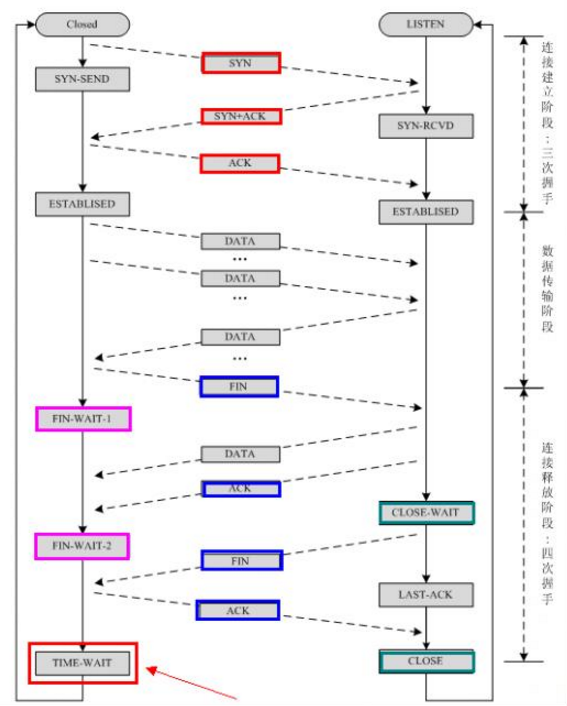
（此时，客户机仍能通过读通道读取服务器的数据，服务器仍能通过写通道写数据）

第二阶段 服务器发送完数据之后，向客户机发送一个FIN数据段，序列号为j；

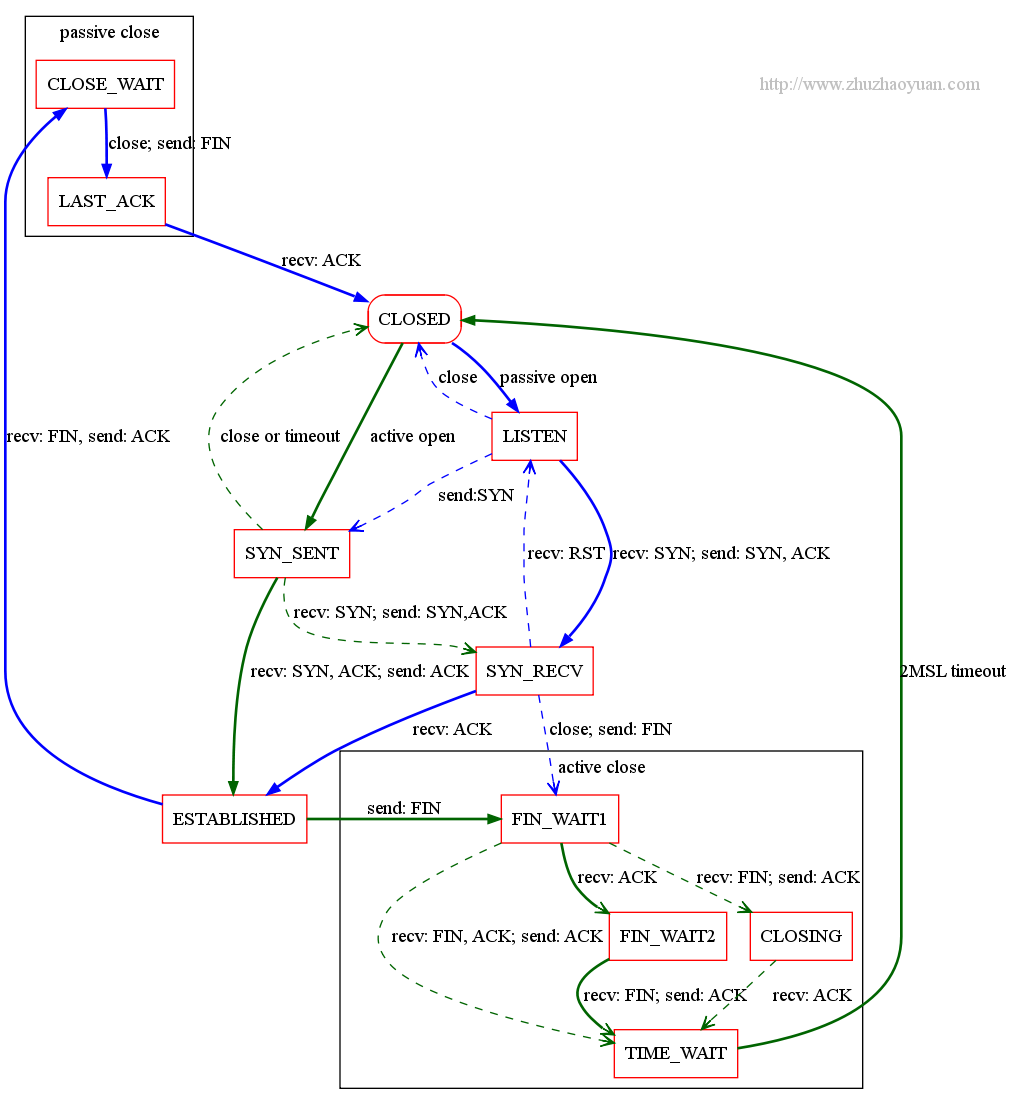
3.客户机收到FIN(j)后，返回确认段ACK，序列号为j+1，关闭客户机读通道；

4.服务器收到ACK(j+1)后，关闭服务器写通道。

这是标准的TCP关闭两个阶段，服务器和客户机都可以发起关闭，完全对称。

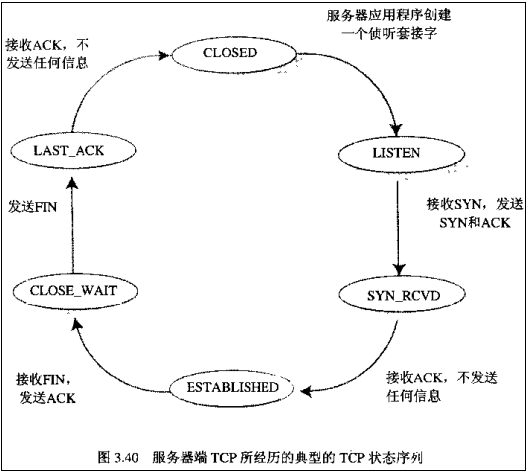


## 三、TCP状态图

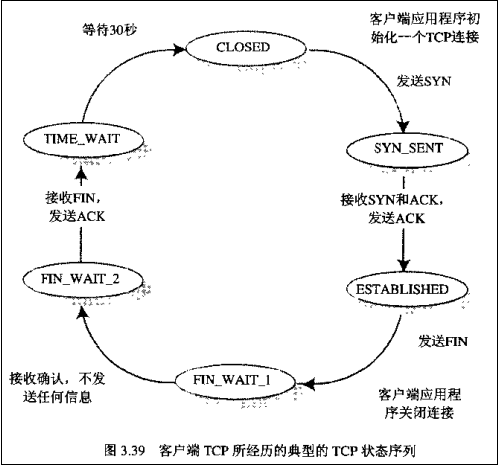


关闭socket分为主动关闭（Active closure）和被动关闭（Passive closure）两种情况。前者是指有本地主机主动发起的关闭；而后者则是指本地主机检测到远程主机发起关闭之后，作出回应，从而关闭整个连接。

1、TCP服务端状态图



2、TCP客户端状态图



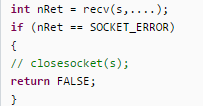
3、CLOSE\_WAIT状态

在被动关闭连接情况下，在已经接收到FIN，但是还没有发送相应的ACK的时刻，连接处于CLOSE\_WAIT状态。

通常来讲，CLOSE\_WAIT状态的持续时间应该很短，正如SYN\_RCVD状态。但是在一些特殊情况下，就会出现连接长时间处于CLOSE\_WAIT状态的情况。

**现象：SOCKET出现大量处于CLOSE\_WAIT状态。**

**原因：主要原因是某种情况下对方关闭了socket链接，但是本方没有关闭相应连接（可能正在从此链路读取或者写数据，收到TCP返回的SOCKET\_ERROR未做关闭操作）。**

****

**解决措施：一般情况下是应用层代码编写问题。代码需要判断socket，一旦读到0，断开连接，read返回负，检查一下errno，如果不是AGAIN，就断开连接。**

4、TIME\_WAIT状态

**现象：SOCKET出现大量处于TIME\_WAIT状态，严重情况下无法创建SOCKET。**

**原因：发起主动关闭的一方 socket将进入TIME\_WAIT状态，TIME\_WAIT状态将持续2个MSL，在Windows下默认为4分钟，不同的操作系统，默认时间不相同。TIME\_WAIT状态下的socket不能被回收使用。具体现象是对于一个处理大量短连接的服务器，如果是由服务器主动关闭客户端的连接，将导致服务器端存在大量的处于TIME\_WAIT状态的socket，严重影响服务器的处理能力，甚至耗尽可用的socket，停止服务。**

**解决SOCKET耗尽措施：修改系统关于TIME\_WAIT相关的参数。《1》修改TIME\_WAIT等待的时间。《2》允许将TIME-WAIT sockets重新用于新的TCP连接。《3》开启TCP连接中TIME-WAIT sockets的快速回收。**

为什么要这样做而不是直接进入CLOSED状态？

原因有二：

一、保证TCP协议的全双工连接能够可靠关闭

二、保证这次连接的重复数据段从网络中消失

先说第一点，如果Client直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致Server没有收到Client最后回复的ACK。那么Server就会在超时之后继续发送FIN，此时由于Client已经CLOSED了，就找不到与重发的FIN对应的连接，最后Server就会收到RST而不是ACK，Server就会以为是连接错误把问题报告给高层。这样的情况虽然不会造成数据丢失，但是却导致TCP协议不符合可靠连接的要求。所以，Client不是直接进入CLOSED，而是要保持TIME\_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

再说第二点，如果Client直接CLOSED，然后又再向Server发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中，这些延迟数据在建立新连接之后才到达Server，由于新连接和老连接的端口号是一样的，又因为TCP协议判断不同连接的依据是socket pair，于是，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失。

## 四、IO多路复用

#### 1、基本概念

　　IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

　（1）当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。

　（2）当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

　（3）如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

　（4）如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。

　（5）如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

　 与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

参考：<http://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html>

#### 2、select的实现原理

select 用于查询设备的状态，以便用户程序获知是否能对设备进行非阻塞的访问，需要设备驱动程序中的poll 函数支持。驱动程序中 poll 函数中最主要用到的一个 API 是 poll\_wait，其原型如下：

void poll\_wait(struct file \*filp, wait\_queue\_heat\_t \*queue, poll\_table \* wait);

poll\_wait 函数所做的工作是把当前进程添加到 wait 参数指定的等待列表（poll\_table）中。

需要说明的是，poll\_wait 函数并不阻塞，程序中 poll\_wait(filp, &outq, wait)这句话的意思并不是说一直等待 outq 信号量可获得，真正的阻塞动作是上层的 select/poll 函数中完成的。select/poll 会在一个循环中对每个需要监听的设备调用它们自己的 poll 支持函数以使得当前进程被加入各个设备的等待列表。若当前没有任何被监听的设备就绪，则内核进行调度（调用 schedule）让出 cpu 进入阻塞状态，schedule 返回时将再次循环检测是否有操作可以进行，如此反复；否则，若有任意一个设备就绪，select/poll 都立即返回。

过程：

（1）从用户空间拷贝fd\_set到内核空间；

（2）遍历所有fd，调用全部指定设备的poll函数，将当前进程加入到各个设备的等待列表中；

（3）当设备就绪时，将唤醒等待队列中的所有节点，当前进程获取到完成信号，select/poll将返回。

（4）如果设备未就绪，则让出CPU进入阻塞状态，timeout之后将再次进行上述操作。

参考：<http://www.cnblogs.com/zhuyp1015/archive/2012/05/31/2529079.html>

select的缺点：

（1）最大并发数有限制。支持的文件描述符数量太少，默认是32\* 32 = 1024，可修改；

（2）效率低。每次都会线性扫描整个fd\_set，集合越大速度越慢；fd\_set内核/用户空间内存拷贝问题。

参考：<http://m.blog.csdn.net/article/details?id=50811553>

#### 3、epoll的实现原理

（1）调用epoll\_create

**在虚拟的epoll文件系统里创建一个file结点。**在内核里，一切皆文件。所以，epoll向内核注册了一个文件系统，用于存储上述的被监控socket，当然这个file不是普通文件，它只服务于epoll。

**在内核cache里创建一个红黑树用于存储需要监听的socket**，以支持快速的查找、插入、删除。epoll在被内核初始化时（操作系统启动），同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的socket。

**建立一个list链表，用于存储准备就绪的事件**。

（2）调用epoll\_ctl

**把socket放到epoll文件系统里file对象对应的红黑树上**；

**给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里**。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

（3）调用epoll\_wait

**观察list链表里有没有数据。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。**而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已。

总结：

一颗红黑树，一张准备就绪句柄链表，少量的内核cache，解决了大并发下的socket处理问题。执行epoll\_create时，创建了红黑树和就绪链表； 执行epoll\_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据; 执行epoll\_wait时立刻返回准备就绪链表里的数据即可。

两种触发模式的区别：

LT（Level Triggered水平触发）模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时重复返回这个句柄，而ET（Edge Triggered 边沿触发）模式仅在第一次返回。

两种模式的实现：

当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait检查这些socket，如果是LT模式，并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表。所以，LT模式的句柄，只要它上面还有事件，epoll\_wait每次都会返回。

参考：

<http://blog.csdn.net/hdutigerkin/article/details/7517390>

<http://m.blog.csdn.net/article/details?id=50811553>

#### 4、IOCP实现原理

IOCP是一种典型的异步IO设计范式，简单的说就是当发起一个IO操作后，不等待操作结束就立刻返回，IO操作的结果在另外一个队列上得到通知并回调。

Windows利用与一个端口相关联的并发值参数来控制一个应用程序中活动线程的数量。如果与一个端口相关的活动线程数达到并发值，那么，在这个端口上等待的线程将不允许再运行了。相反，它将等待某个活动线程处理完当前操作并检查是否有别的包正在该端口上等待。如果有的话，该线程只是简单的抓获该包然后处理。在这个过程中，没有上下文切换，CPU得到最大限度的利用。

<http://blog.csdn.net/zzhongcy/article/details/38657399>

<http://www.cnblogs.com/FCoding/archive/2012/07/16/2594442.html>