**EPOLL事件有两种模型：**

Edge Triggered (ET) 边缘触发只有数据到来,才触发,不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发只要缓冲区有数据都会触发。

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表．

优点：当进行socket通信的时候，保证了数据的完整输出，进行IO操作的时候，如果还有数据，就会一直的通知你。

缺点：由于只要还有数据，内核就会不停的从内核空间转到用户空间，所有占用了大量内核资源，试想一下当有大量数据到来的时候，每次读取一个字节，这样就会不停的进行切换。内核资源的浪费严重。效率来讲也是很低的。

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once).

优点：每次内核只会通知一次，大大减少了内核资源的浪费，提高效率。

缺点：不能保证数据的完整。不能及时的取出所有的数据。

应用场景： 处理大数据。使用non-block模式的socket。

**TCP三次握手过程的面试题：**

1. 写一下socket网络编程服务端和客户端常用的函数。

server: socket, bind, listen, accept

client: socket, connect

2. 如果服务端在listen之后没有accept，那客户端的connect会返回吗？为什么？

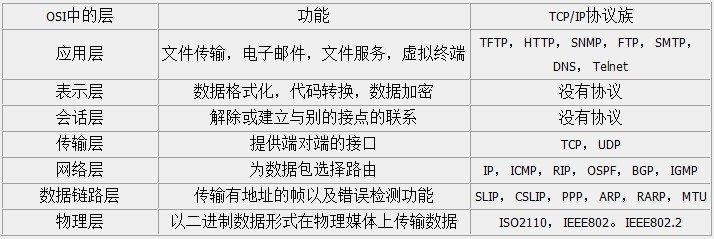
会。三次握手发生在connect与listen之间，而不是与accept之间。accept是从队列中取出建立好的链接的fd。如果队列满了，有个参数可以控制是直接reset client还是丢弃fd。

3. 此时调用send发数据会怎么样？

可能是成功的。因为send是把数据写入socket的缓冲区。

**TCP三次握手：**

TCP工作在网络OSI的七层模型中的第四层——Transport层，IP在第三层——Network层，ARP在第二层——Data Link层；在第二层上的数据，我们把它叫Frame，在第三层上的数据叫Packet，第四层的数据叫Segment。 同时，我们需要简单的知道，数据从应用层发下来，会在每一层都会加上头部信息，进行封装，然后再发送到数据接收端。这个基本的流程你需要知道，就是每个数据都会经过数据的封装和解封装的过程。 在OSI七层模型中，每一层的作用和对应的协议如下：

****

上面就是TCP协议头部的格式，由于它太重要了，是理解其它内容的基础，下面就将每个字段的信息都详细的说明一下。

(1) Source Port和Destination Port:分别占用16位，表示源端口号和目的端口号；用于区别主机中的不同进程，而IP地址是用来区分不同的主机的，源端口号和目的端口号配合上IP首部中的源IP地址和目的IP地址就能唯一的确定一个TCP连接；

(2) Sequence Number:用来标识从TCP发端向TCP收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节在数据流中的序号；主要用来解决网络报乱序的问题；

(3) Acknowledgment Number:32位确认序列号包含发送确认的一端所期望收到的下一个序号，因此，确认序号应当是上次已成功收到数据字节序号加1。不过，只有当标志位中的ACK标志（下面介绍）为1时该确认序列号的字段才有效。主要用来解决不丢包的问题；

(4) Offset:给出首部中32 bit字的数目，需要这个值是因为任选字段的长度是可变的。这个字段占4bit（最多能表示15个32bit的的字，即4\*15=60个字节的首部长度），因此TCP最多有60字节的首部。然而，没有任选字段，正常的长度是20字节；

(5) TCP Flags:TCP首部中有6个标志比特，它们中的多个可同时被设置为1，主要是用于操控TCP的状态机的，依次为URG，ACK，PSH，RST，SYN，FIN。每个标志位的意思如下：

1) URG：此标志表示TCP包的紧急指针域（后面马上就要说到）有效，用来保证TCP连接不被中断，并且督促中间层设备要尽快处理这些数据；

2) ACK：此标志表示应答域有效，就是说前面所说的TCP应答号将会包含在TCP数据包中；有两个取值：0和1，为1的时候表示应答域有效，反之为0；**TCP协议规定，只有ACK=1时有效，也规定连接建立后所有发送的报文的ACK必须为1。**

3) PSH：这个标志位表示Push操作。所谓Push操作就是指在数据包到达接收端以后，立即传送给应用程序，而不是在缓冲区中排队；

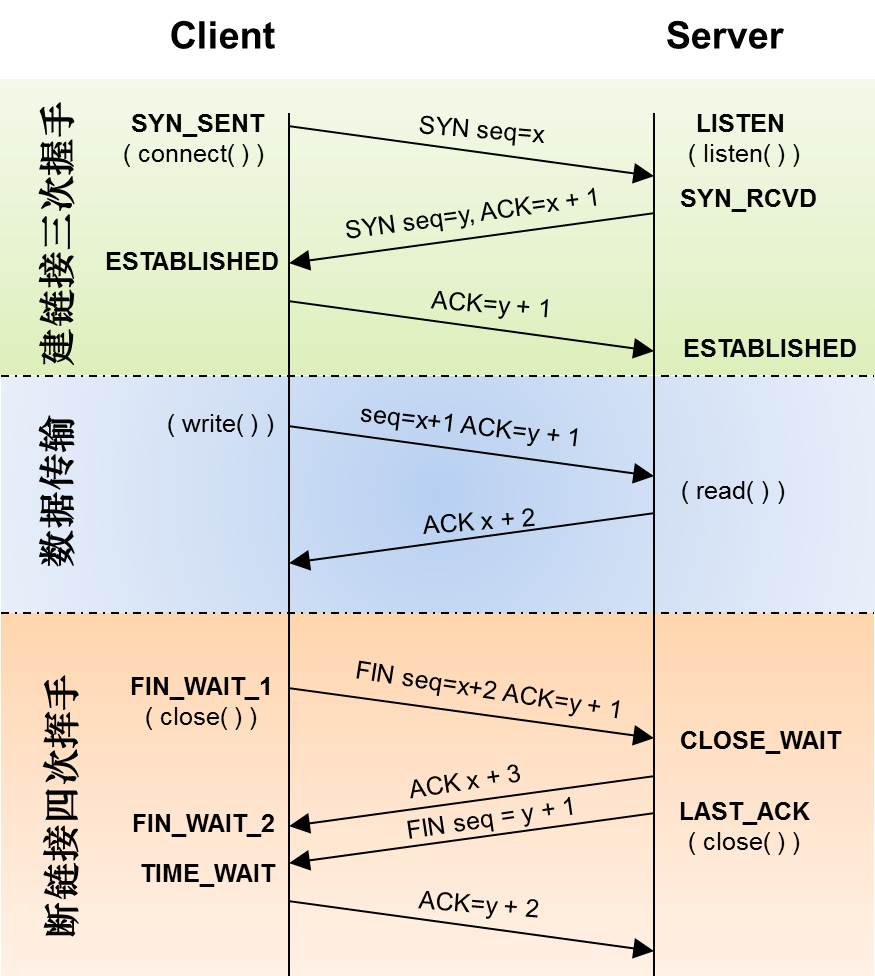
4) RST：这个标志表示连接复位请求。用来复位那些产生错误的连接，也被用来拒绝错误和非法的数据包；

5) SYN：表示同步序号，用来建立连接。SYN标志位和ACK标志位搭配使用，当连接请求的时候，SYN=1，ACK=0；连接被响应的时候，SYN=1，ACK=1；这个标志的数据包经常被用来进行端口扫描。扫描者发送一个只有SYN的数据包，如果对方主机响应了一个数据包回来 ，就表明这台主机存在这个端口；但是由于这种扫描方式只是进行TCP三次握手的第一次握手，因此这种扫描的成功表示被扫描的机器不很安全，一台安全的主机将会强制要求一个连接严格的进行TCP的三次握手；**在连接建立时用来同步序号。当SYN=1而ACK=0时，表明这是一个连接请求报文。对方若同意建立连接，则应在响应报文中使SYN=1和ACK=1. 因此, SYN置1就表示这是一个连接请求或连接接受报文。**

6) FIN： 表示发送端已经达到数据末尾，也就是说双方的数据传送完成，没有数据可以传送了，发送FIN标志位的TCP数据包后，连接将被断开。这个标志的数据包也经常被用于进行端口扫描。**当 FIN = 1 时，表明此报文段的发送方的数据已经发送完毕，并要求释放连接。**

Window:窗口大小，也就是有名的滑动窗口，用来进行流量控制。

**三次握手和四次挥手的过程：**



**三次握手过程：**

(1) 第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN\_SEND状态，等待服务器的确认；

(2) 第二次握手：服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN\_RECV状态；

(3) 第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。  
完成了三次握手，客户端和服务器端就可以开始传送数据。

**为什么三次握手?**

在谢希仁著《计算机网络》第四版中讲“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”。在另一部经典的《计算机网络》一书中讲“三次握手”的目的是为了解决“网络中存在延迟的重复分组”的问题。例子如下：

"已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。

**四次挥手过程：**

(1) 第一次分手：主机1（可以使客户端，也可以是服务器端），设置Sequence Number和Acknowledgment Number，向主机2发送一个FIN报文段；此时，主机1进入FIN\_WAIT\_1状态；这表示主机1没有数据要发送给主机2了；

(2) 第二次分手：主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1；主机1进入FIN\_WAIT\_2状态；主机2告诉主机1，我“同意”你的关闭请求；

(3) 第三次分手：主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST\_ACK状态；

(4) 第四次分手：主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME\_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

**为什么四次挥手?**

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。如果要正确的理解四次分手的原理，就需要了解四次分手过程中的状态变化。

(1) 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。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，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，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，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。（主动方）

(2) 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，但另外还告诉对方，我暂时还有点数据需要传送给你(ACK信息)，稍后再关闭连接。（主动方）

(3) CLOSE\_WAIT：这种状态的含义其实是表示在等待关闭。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，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，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。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。（被动方）

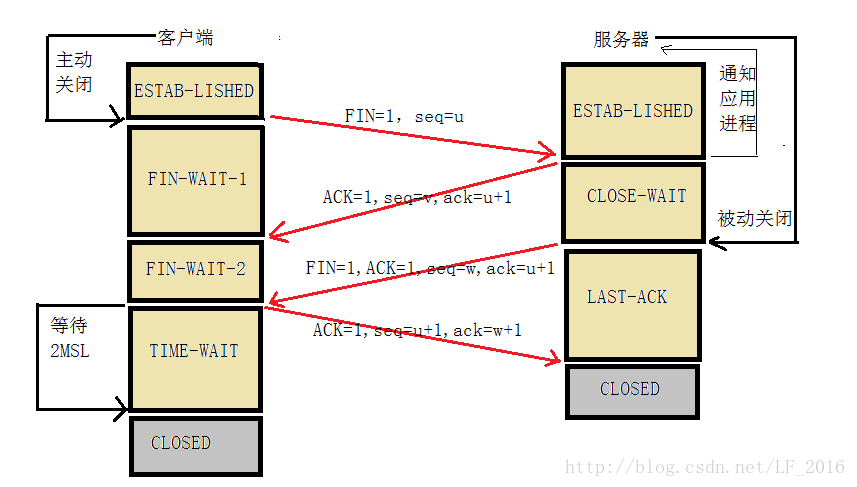
(4) LAST\_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。（被动方）

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。如果FINWAIT1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。（主动方）

(5) CLOSED: 表示连接中断。

**TCP四次挥手：**

当客户端进入TIME-WAIT状态的时候(也就是第四次挥手的时候)，必须经过时间计数器设置的时间2MSL(Maximum Segment Lifetime，最长报文段寿命)后，才能进入关闭状态，为什么？



这最主要是因为两个理由：

(1) 为了保证客户端发送的最后一个ACK报文段能够到达服务器。因为这个ACK有可能丢失，从而导致处在LAST-ACK状态的服务器收不到对FIN-ACK的确认报文。服务器会超时重传这个FIN-ACK，接着客户端再重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待2MSL，而是在发送完ACK之后直接释放关闭，一但这个ACK丢失的话，服务器就无法正常的进入关闭连接状态。

(2) 可以防止已失效的报文段。客户端在发送最后一个ACK之后，再经过经过2MSL，就可以使本链接持续时间内所产生的所有报文段都从网络中消失。从保证在关闭连接后不会有还在网络中滞留的报文段去骚扰服务器。假设在12.106.32.254的1500端口和206.168.1.112.219的21端口之间有一个TCP连接。我们关闭这个链接，过一段时间后在 相同的IP地址和端口建立另一个连接。后一个链接成为前一个的化身。因为它们的IP地址和端口号都相同。TCP必须防止来自某一个连接的老的重复分组在连接已经终止后再现，从而被误解成属于同一链接的某一个某一个新的化身。为做到这一点，TCP将不给处于TIME\_WAIT状态的链接发起新的化身。既然TIME\_WAIT状态的持续时间是MSL的2倍，这就足以让某个方向上的分组最多存活msl秒即被丢弃，另一个方向上的应答最多存活msl秒也被丢弃。 通过实施这个规则，我们就能保证每成功建立一个TCP连接时。来自该链接先前化身的重复分组都已经在网络中消逝了。

注意：在服务器发送了FIN-ACK之后，会立即启动超时重传计时器。客户端在发送最后一个ACK之后会立即启动时间等待计时器。

**Linux下高并发socket最大连接数所受的各种限制**

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：  
[root@RHEL6 ~]# 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左右个TCP并发连接。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的软限制(soft limit)和硬限制(hardlimit)：

软限制是指Linux在当前系统能够承受的范围内进一步限制用户同时打开的文件数；

硬限制则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。通常软限制小于或等于硬限制。

修改上述限制的最简单的办法就是使用ulimit命令：  
[root@RHEL6 ~]# ulimit -n 2048 #修改单一进程允许打开的最大文件数2048

执行上述命令中，如果系统回显类似于“Operation notpermitted”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。  
第一步，修改/etc/security/limits.conf文件，在文件中添加如下行：  
root soft nofile 10240  
root hard nofile 10240

可用’\*'号表示修改所有用户的限制；soft或hard指定要修改软限制还是硬限制；10240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：  
[root@RHEL6 ~]# cat /proc/sys/fs/file-max  
149240

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)149240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。通常这个系统级硬限制是Linux系统在启动时根据系统硬件资源状况计算出来的最佳的最大同时打开文件数限制，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。修改此硬限制的方法是修改/etc/rc.local脚本，在脚本中添加如下行：  
echo 289240 > /proc/sys/fs/file-max  
重启系统，Linux在启动完成后强行将系统级打开文件数硬限制设置为289240。

如果重启后用 ulimit-n命令查看用户可打开文件数限制仍然低于上述步骤中设置的最大值，这可能是因为在用户登录脚本/etc/profile中使用ulimit -n命令已经将用户可同时打开的文件数做了限制。由于通过ulimit-n修改系统对用户可同时打开文件的最大数限制时，新修改的值只能小于或等于上次 ulimit-n设置的值，因此想用此命令增大这个限制值是不可能的。所以，如果有上述问题存在，就只能去打开/etc/profile脚本文件，在文件中查找是否使用了ulimit-n限制了用户可同时打开的最大文件数量，如果找到，则删除这行命令，或者将其设置的值改为合适的值，然后保存文件，用户退出并重新登录系统即可。

**修改网络内核对TCP连接的有关限制**

在Linux上编写支持高并发TCP连接的客户端通讯处理程序时，有时会发现尽管已经解除了系统对用户同时打开文件数的限制，但仍会出现并发TCP连接数增加到一定数量时，再也无法成功建立新的TCP连接的现象。出现这种现在的原因有多种。  
**第一种原因可能是因为Linux网络内核对本地端口号范围有限制。**此时，进一步分析为什么无法建立TCP连接，会发现问题出在connect()调用返回失败，查看系统错误提示消息是“Can’t assign requestedaddress”。同时，如果在此时用tcpdump工具监视网络，会发现根本没有TCP连接时客户端发SYN包的网络流量。这些情况说明问题在于本地Linux系统内核中有限制。其实，问题的根本原因在于Linux内核的TCP/IP协议实现模块对系统中所有的客户端TCP连接对应的本地端口号的范围进行了限制(例如，内核限制本地端口号的范围为1024~32768之间)。当系统中某一时刻同时存在太多的TCP客户端连接时，由于每个TCP客户端连接都要占用一个唯一的本地端口号(此端口号在系统的本地端口号范围限制中)，如果现有的TCP客户端连接已将所有的本地端口号占满，则此时就无法为新的TCP客户端连接分配一个本地端口号了，因此系统会在这种情况下在connect()调用中返回失败，并将错误提示消息设为“Can’t assignrequested address”。有关这些控制逻辑可以查看Linux内核源代码，以linux2.6内核为例，可以查看tcp\_ipv4.c文件中如下函数：  
static int tcp\_v4\_hash\_connect(struct sock \*sk)  
请注意上述函数中对变量sysctl\_local\_port\_range的访问控制。变量sysctl\_local\_port\_range的初始化则是在tcp.c文件中的如下函数中设置：  
void \_\_init tcp\_init(void)  
内核编译时默认设置的本地端口号范围可能太小，因此需要修改此本地端口范围限制。  
第一步，修改/etc/sysctl.conf文件，在文件中添加如下行：  
net.ipv4.ip\_local\_port\_range = 1024 65000  
这表明将系统对本地端口范围限制设置为1024~65000之间。请注意，本地端口范围的最小值必须大于或等于1024；而端口范围的最大值则应小于或等于65535。修改完后保存此文件。  
第二步，执行sysctl命令：  
[root@RHEL6 ~]# sysctl -p  
如果系统没有错误提示，就表明新的本地端口范围设置成功。如果按上述端口范围进行设置，则理论上单独一个进程最多可以同时建立60000多个TCP客户端连接。  
**第二种无法建立TCP连接的原因可能是因为Linux网络内核的IP\_TABLE防火墙对最大跟踪的TCP连接数有限制。**此时程序会表现为在 connect()调用中阻塞，如同死机，如果用tcpdump工具监视网络，也会发现根本没有TCP连接时客户端发SYN包的网络流量。由于 IP\_TABLE防火墙在内核中会对每个TCP连接的状态进行跟踪，跟踪信息将会放在位于内核内存中的conntrackdatabase中，这个数据库的大小有限，当系统中存在过多的TCP连接时，数据库容量不足，IP\_TABLE无法为新的TCP连接建立跟踪信息，于是表现为在connect()调用中阻塞。此时就必须修改内核对最大跟踪的TCP连接数的限制，方法同修改内核对本地端口号范围的限制是类似的：  
第一步，修改/etc/sysctl.conf文件，在文件中添加如下行：  
net.ipv4.ip\_conntrack\_max = 10240  
这表明将系统对最大跟踪的TCP连接数限制设置为10240。请注意，此限制值要尽量小，以节省对内核内存的占用。  
第二步，执行sysctl命令：  
[root@RHEL6 ~]# sysctl -p  
如果系统没有错误提示，就表明系统对新的最大跟踪的TCP连接数限制修改成功。如果按上述参数进行设置，则理论上单独一个进程最多可以同时建立10000多个TCP客户端连接。

[**详解Linux服务器最大tcp连接数**](https://www.cnblogs.com/fjping0606/p/4729389.html)

网络编程  
在tcp应用中，server事先在某个固定端口监听，client主动发起连接，经过三次握手后建立tcp连接。那么对单机，其最大并发tcp连接数是多少？

如何标识一个TCP连接  
在确定最大连接数之前，先来看看系统如何标识一个tcp连接。系统用一个4四元组来唯一标识一个TCP连接：{local ip, local port,remote ip,remote port}。

client最大tcp连接数  
client每次发起tcp连接请求时，除非绑定端口，通常会让系统选取一个空闲的本地端口（local port），该端口是独占的，不能和其他tcp连接共享。tcp端口的数据类型是unsigned short，因此本地端口个数最大只有65536，端口0有特殊含义，不能使用，这样可用端口最多只有65535，所以在全部作为client端的情况下，最大tcp连接数为65535，这些连接可以连到不同的server ip。

server最大tcp连接数  
server通常固定在某个本地端口上监听，等待client的连接请求。不考虑地址重用（unix的SO\_REUSEADDR选项）的情况下，即使server端有多个ip，本地监听端口也是独占的，因此server端tcp连接4元组中只有remote ip（也就是client ip）和remote port（客户端port）是可变的，因此最大tcp连接为客户端ip数×客户端port数，对IPV4，不考虑ip地址分类等因素，最大tcp连接数约为2的32次方（ip数）×2的16次方（port数），也就是server端单机最大tcp连接数约为2的48次方。

**那么单台服务器上的并发TCP连接数可以有多少？**

上面给出的是理论上的单机最大连接数，在实际环境中，受到机器资源、操作系统等的限制，特别是sever端，其最大并发tcp连接数远不能达到理论上限。在unix/linux下限制连接数的主要因素是内存和允许的文件描述符个数（每个tcp连接都要占用一定内存，每个socket就是一个文件描述符），另外1024以下的端口通常为保留端口。在默认2.6内核配置下，经过试验，每个socket占用内存在15~20k之间。  
影响一个socket占用内存的参数包括：  
rmem\_max  
wmem\_max  
tcp\_rmem  
tcp\_wmem  
tcp\_mem  
对server端，通过增加内存、修改最大文件描述符个数等参数，单机最大并发TCP连接数超过10万 是没问题的，国外 Urban Airship 公司在产品环境中已做到 50 万并发 。在实际应用中，对大规模网络应用，还需要考虑C10K 问题。

下面再详解一下文件句柄限制和端口限制两大常识

常识一：文件句柄限制

在linux下编写网络服务器程序的朋友肯定都知道每一个tcp连接都要占一个文件描述符，一旦这个文件描述符使用完了，新的连接到来返回给我们的错误是“Socket/File:Can't open so many files”。

这时你需要明白操作系统对可以打开的最大文件数的限制。

进程限制

执行 ulimit -n 输出 1024，说明对于一个进程而言最多只能打开1024个文件，所以你要采用此默认配置最多也就可以并发上千个TCP连接。

临时修改：ulimit -n 1000000，但是这种临时修改只对当前登录用户目前的使用环境有效，系统重启或用户退出后就会失效。

重启后失效的修改（不过我在CentOS 6.5下测试，重启后未发现失效）：编辑 /etc/security/limits.conf 文件， 修改后内容为

\* soft nofile 1000000

\* hard nofile 1000000

永久修改：编辑/etc/rc.local，在其后添加如下内容

ulimit -SHn 1000000

全局限制

执行 cat /proc/sys/fs/file-nr 输出 9344 0 592026，分别为：1.已经分配的文件句柄数，2.已经分配但没有使用的文件句柄数，3.最大文件句柄数。但在kernel 2.6版本中第二项的值总为0，这并不是一个错误，它实际上意味着已经分配的文件描述符无一浪费的都已经被使用了 。

我们可以把这个数值改大些，用 root 权限修改 /etc/sysctl.conf 文件:

fs.file-max = 1000000

net.ipv4.ip\_conntrack\_max = 1000000

net.ipv4.netfilter.ip\_conntrack\_max = 1000000

常识二：端口号范围限制？

操作系统上端口号1024以下是系统保留的，从1024-65535是用户使用的。由于每个TCP连接都要占一个端口号，所以我们最多可以有60000多个并发连接。我想有这种错误思路朋友不在少数吧？（我过去就一直这么认为）

**优化linux服务器最大TCP连接数**

------------优化服务器单服可承载的最大tcp连接数

在linux中，开启tcp连接实际上就是打开文件句柄，所以，可以打开多少文件句柄，就可以创建多少tcp连接。

----------全局配置，所有进程共享的上限  
查看系统配置最大句柄数   cat /proc/sys/fs/file-nr  
修改： /etc/sysctl.conf  
fs.file-max = 1000000  
net.ipv4.ip\_conntrack\_max = 1000000  
net.ipv4.netfilter.ip\_conntrack\_max = 1000000

---------单进程配置，单个进程独立的上限配置  
查看最大进程数  ulimit -u  
查看各种系统限制   ulimit -a  
查看进程最大句柄数 ulimit -n  
修改：/etc/security/limits.conf  
\*(所有用户) soft(软限制) nofile(单进程可打开最大句柄) 1000000  
\* hard(硬限制) nofile 1000000  
\* soft nopro(最大进程数) 1000000  
\* hard nopro 1000000

在tcp连接的四次挥手过程中，有一个固定的连接断开等待时间配置，如果这个时间过长，就会造成大量等待断开链接状态的资源堆积，所以，调试出一个合理的等待时间，可以有效增加无效链接所占资源的回收利用。

----------linux优化tcp连接存活时间，关闭连接时等待时间  
修改/etc/sysctl.conf，添加如下几行：  
#改系統默认的TIMEOUT时间  
net.ipv4.tcp\_fin\_timeout=2

#启重用，允许将TIME\_WAIT sockets重新用于新的TCP连接 默认为0表示关闭  
net.ipv4.tcp\_tw\_reuse=1

#开启TCP连接中TIME\_WAIT sockets的快速回收 默认为0 表示关闭  
net.ipv4.tcp\_tw\_recycle=1

以上就是linux服务器的一些配置上的优化。