“三次握手，四次挥手”你真的懂吗?

记得刚毕业找工作面试的时候，经常会被问到：你知道“3次握手，4次挥手”吗？这时候我会“胸有成竹”地“背诵”前期准备好的“答案”，第一次怎么怎么，第二次.....答完就没有下文了，面试官貌似也没有深入下去的意思，深入下去我也不懂，皆大欢喜！

作为程序员，要有“刨根问底”的精神。知其然，更要知其所以然。这篇文章希望能抽丝剥茧，还原背后的原理。

目录

什么是“3次握手，4次挥手”

TCP是一种面向连接的单播协议，在发送数据前，通信双方必须在彼此间建立一条连接。所谓的“连接”，其实是客户端和服务器的内存里保存的一份关于对方的信息，如ip地址、端口号等。

TCP可以看成是一种字节流，它会处理IP层或以下的层的丢包、重复以及错误问题。在连接的建立过程中，双方需要交换一些连接的参数。这些参数可以放在TCP头部。

TCP提供了一种可靠、面向连接、字节流、传输层的服务，采用三次握手建立一个连接。采用4次挥手来关闭一个连接。

TCP服务模型

在了解了建立连接、关闭连接的“三次握手和四次挥手”后，我们再来看下TCP相关的东西。

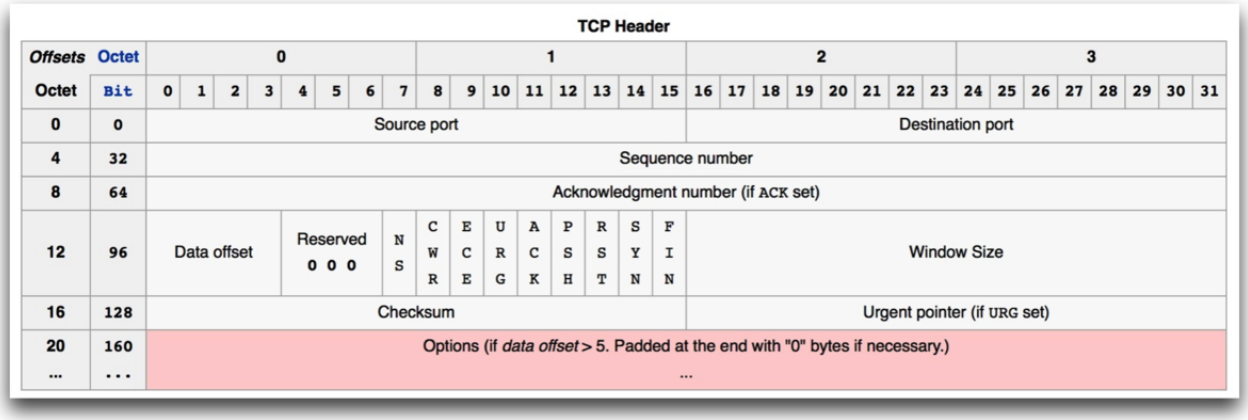
一个TCP连接由一个4元组构成，分别是两个IP地址和两个端口号。一个TCP连接通常分为三个阶段：启动、数据传输、退出（关闭）。

当TCP接收到另一端的数据时，它会发送一个确认，但这个确认不会立即发送，一般会延迟一会儿。ACK是累积的，一个确认字节号N的ACK表示所有直到N的字节（不包括N）已经成功被接收了。这样的好处是如果一个ACK丢失，很可能后续的ACK就足以确认前面的报文段了。

一个完整的TCP连接是双向和对称的，数据可以在两个方向上平等地流动。给上层应用程序提供一种双工服务。一旦建立了一个连接，这个连接的一个方向上的每个TCP报文段都包含了相反方向上的报文段的一个ACK。

序列号的作用是使得一个TCP接收端可丢弃重复的报文段，记录以杂乱次序到达的报文段。因为TCP使用IP来传输报文段，而IP不提供重复消除或者保证次序正确的功能。另一方面，TCP是一个字节流协议，绝不会以杂乱的次序给上层程序发送数据。因此TCP接收端会被迫先保持大序列号的数据不交给应用程序，直到缺失的小序列号的报文段被填满。

TCP头部



TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	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
0	0	Source port																Destination port														
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																	Urgent pointer (if URG set)													
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

源端口和目的端口在TCP层确定双方进程，序列号表示的是报文段数据中的第一个字节号，ACK表示确认号，该确认号的发送方期待接收的下一个序列号，即最后被成功接收的数据字节序列号加1，这个字段只有在ACK位被启用的时候才有效。

当新建一个连接时，从客户端发送到服务端的第一个报文段的SYN位被启用，这称为SYN报文段，这时序列号字段包含了在本次连接的这个方向上要使用的第一个序列号，即初始序列号ISN，之后发送的数据是ISN加1，因此SYN位字段会消耗一个序列号，这意味着使用重传进行可靠传输。而不消耗序列号的ACK则不是。

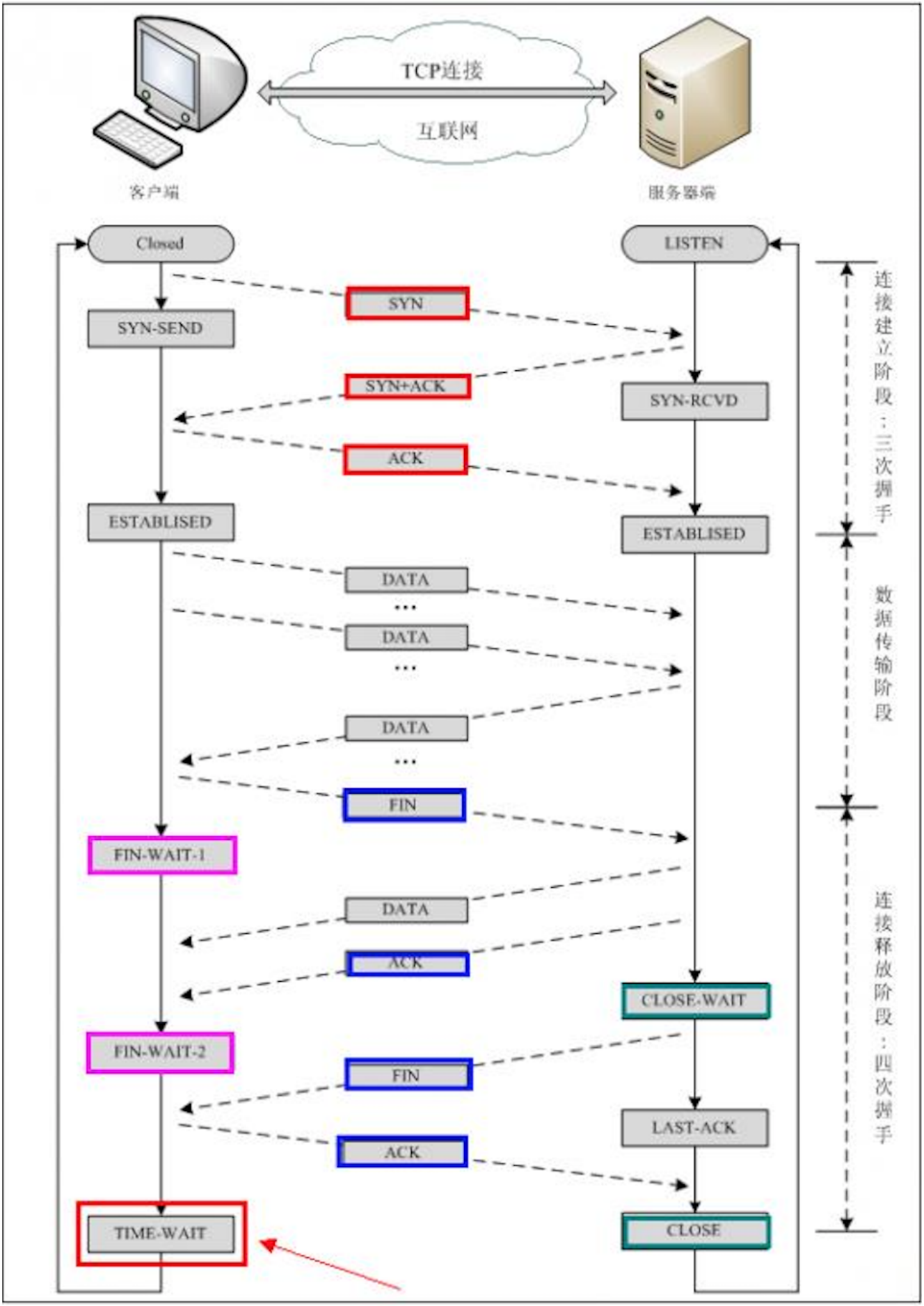
头部长度（图中的数据偏移）以32位字为单位，也就是以4bytes为单位，它只有4位，最大为15，因此头部最大长度为60字节，而其最小为5，也就是头部最小为20字节（可变选项为空）。

ACK —— 确认，使得确认号有效。
RST —— 重置连接（经常看到的reset by peer）就是此字段搞的鬼。
SYN —— 用于初如化一个连接的序列号。
FIN —— 该报文段的发送方已经结束向对方发送数据。

当一个连接被建立或被终止时，交换的报文段只包含TCP头部，而没有数据。

状态转换

三次握手和四次挥手的状态转换如下图。



为什么要“三次握手，四次挥手”

三次握手

换个易于理解的视角来看为什么要3次握手。

客户端和服务端通信前要进行连接，“3次握手”的作用就是双方都能明确自己和对方的收、发能力是正常的。

第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

第二次握手：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。
从客户端的视角来看，我接到了服务端发送过来的响应数据包，说明服务端接收到了我在第一次握手时发送的网络包，并且成功发送了响应数据包，这就说明，服务端的接收、发送能力正常。而另一方面，我收到了服务端的响应数据包，说明我第一次发送的网络包成功到达服务端，这样，我自己的发送和接收能力也是正常的。

第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。
第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

经历了上面的三次握手过程，客户端和服务端都确认了自

己的接收、发送能力是正常的。之后就可以正常通信了。

每次都是接收到数据包的一方可以得到一些结论，发送的一方其实没有任何头绪。我虽然有发包的动作，但是我怎么知道我有没有发出去，而对方有没有接收到呢?

而从上面的过程可以看到，最少是需要三次握手过程的。两次达不到让双方都得出自己、对方的接收、发送能力都正常的结论。其实每次收到网络包的一方至少是可以得到：对方的发送、我方的接收是正常的。而每一步都是有关联的，下一次的“响应”是由于第一次的“请求”触发，因此每次握手其实是可以得到额外的结论的。比如第三次握手时，服务端收到数据包，表明看服务端只能得到客户端的发送能力、服务端的接收能力是正常的，但是结合第二次，说明服务端在第二次发送的响应包，客户端接收到了，并且作出了响应，从而得到额外的结论：客户端的接收、服务端的发送是正常的。

用表格总结一下：

视角	客收	客发	服收	服发
客视角	二	一 + 二	一 + 二	二
服视角	二 + 三	一	一	二 + 三

四次挥手

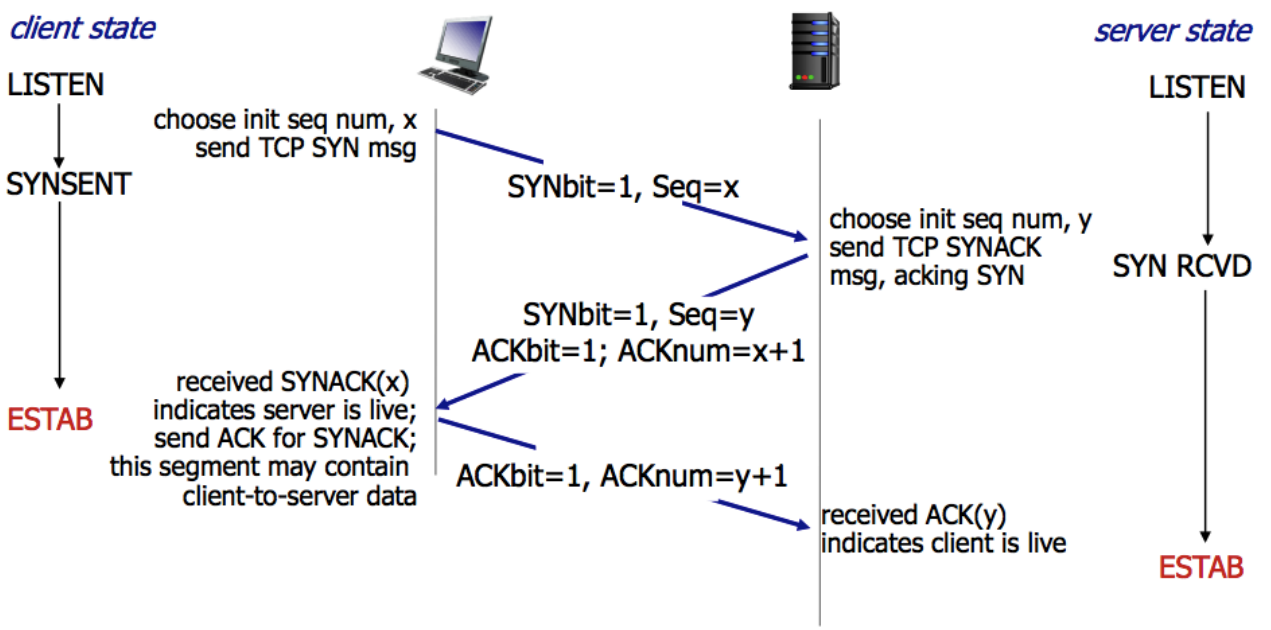
TCP连接是双向传输的对等的模式，就是说双方都可以同时向对方发送或接收数据。当有一方要关闭连接时，会发送指令告知对方，我要关闭连接了。这时对方会回一个ACK，此时一个方向的连接关闭。但是另一个方向仍然可

以继续传输数据，等到发送完了所有的数据后，会发送一个FIN段来关闭此方向上的连接。接收方发送ACK确认关闭连接。注意，接收到FIN报文的一方只能回复一个ACK，它是无法马上返回对方一个FIN报文段的，因为结束数据传输的“指令”是上层应用层给出的，我只是一个“搬运工”，我无法了解“上层的意志”。

“三次握手，四次挥手”怎么完成?

其实3次握手的目的并不只是让通信双方都了解到一个连接正在建立，还在于利用数据包的选项来传输特殊的信息，交换初始序列号ISN。

3次握手是指发送了3个报文段，4次挥手是指发送了4个报文段。注意，SYN和FIN段都是会利用重传进行可靠传输的。



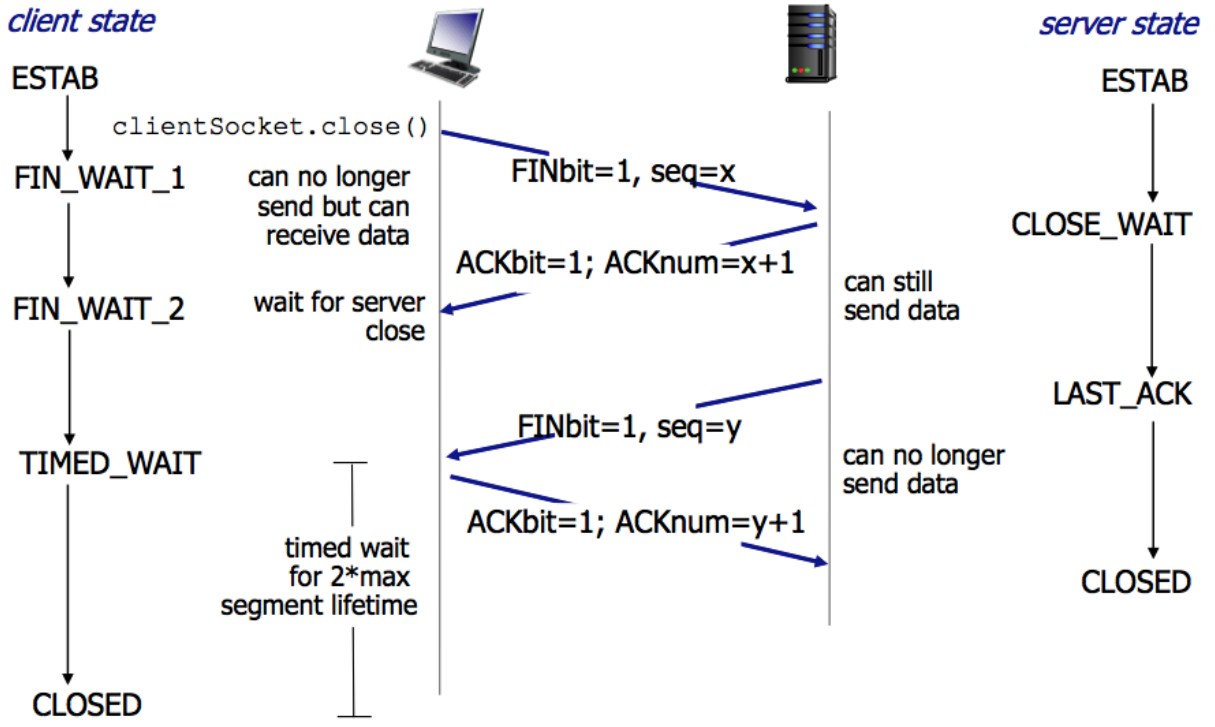
三次握手

1. 客户端发送一个SYN段，并指明客户端的初始序列

号，即ISN(c).

2. 服务端发送自己的SYN段作为应答，同样指明自己的ISN(s)。为了确认客户端的SYN，将ISN(c)+1作为ACK数值。这样，每发送一个SYN，序列号就会加1. 如果有丢失的情况，则会重传。
3. 为了确认服务器端的SYN，客户端将ISN(s)+1作为返回的ACK数值。

四次挥手



1. 客户端发送一个FIN段，并包含一个希望接收者看到的自己当前的序列号K. 同时还包含一个ACK表示确认对方最近一次发过来的数据。
2. 服务端将K值加1作为ACK序号值，表明收到了上一个包。这时上层的应用程序会被告知另一端发起了关闭操作，通常这将引起应用程序发起自己的关闭操作。
3. 服务端发起自己的FIN段，ACK=K+1, Seq=L
4. 客户端确认。ACK=L+1

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，己方是否现在关闭发送数据通道，需要上层应用来决定，因此，己方ACK和FIN一般都会分开发送。

“三次握手，四次挥手”进阶

ISN

三次握手的一个重要功能是客户端和服务端交换ISN(Initial Sequence Number), 以便让对方知道接下来接收数据的时候如何按序列号组装数据。

如果ISN是固定的，攻击者很容易猜出后续的确认号。

```
ISN = M + F(localhost, localport, remotehost, remoteport)
```

M是一个计时器，每隔4微秒加1。
F是一个Hash算法，根据源IP、目的IP、源端口、目的端口生成一个随机数值。要保证hash算法不能被外部轻易推算得出。

序列号回绕

因为ISN是随机的，所以序列号容易就会超过2^31-1. 而

tcp对于丢包和乱序等问题的判断都是依赖于序列号大小比较的。此时就出现了所谓的tcp序列号回绕（sequence wraparound）问题。怎么解决?

```
/*
* The next routines deal with comparing 32 bit unsigned
ints
* and worry about wraparound (automatic with unsigned
arithmetic).
*/
static inline int before(__u32 seq1, __u32 seq2)
{
    return (__s32)(seq1-seq2) < 0;
}

#define after(seq2, seq1) before(seq1, seq2)
```

上述代码是内核中的解决回绕问题代码。__s32是有符号整型的意思，而__u32则是无符号整型。序列号发生回绕后，序列号变小，相减之后，把结果变成有符号数了，因此结果成了负数。

```
假设seq1=255,  seq2=1（发生了回绕）。
seq1 = 1111 1111 seq2 = 0000 0001
我们希望比较结果是
 seq1 - seq2=
 1111 1111
-0000 0001
-----------
 1111 1110
```

由于我们将结果转化成了有符号数，由于最高位是1，因此结果是一个负数，负数的绝对值为

```
0000 0001 + 1 = 0000 0010 = 2
```

因此`seq1 - seq2 < 0`

syn flood攻击

最基本的DoS攻击就是利用合理的服务请求来占用过多的服务资源，从而使合法用户无法得到服务的响应。syn flood属于Dos攻击的一种。

如果恶意的向某个服务器端口发送大量的SYN包，则可以使服务器打开大量的半开连接，分配TCB（Transmission Control Block）, 从而消耗大量的服务器资源，同时也使得正常的连接请求无法被相应。当开放了一个TCP端口后，该端口就处于Listening状态，不停地监视发到该端口的Syn报文，一 旦接收到Client发来的Syn报文，就需要为该请求分配一个TCB，通常一个TCB至少需要280个字节，在某些操作系统中TCB甚至需要1300个字节，并返回一个SYN ACK命令，立即转为SYN-RECEIVED即半开连接状态。系统会为此耗尽资源。

常见的防攻击方法有：

无效连接的监视释放

监视系统的半开连接和不活动连接，当达到一定阈值时拆除这些连接，从而释放系统资源。这种方法对于所有的连接一视同仁，而且由于SYN Flood造成的半开连接数量很

大，正常连接请求也被淹没在其中被这种方式误释放掉，因此这种方法属于入门级的SYN Flood方法。

延缓TCB分配方法

消耗服务器资源主要是因为当SYN数据报文一到达，系统立即分配TCB，从而占用了资源。而SYN Flood由于很难建立起正常连接，因此，当正常连接建立起来后再分配TCB则可以有效地减轻服务器资源的消耗。常见的方法是使用Syn Cache和Syn Cookie技术。

Syn Cache技术

系统在收到一个SYN报文时，在一个专用HASH表中保存这种半连接信息，直到收到正确的回应ACK报文再分配TCB。这个开销远小于TCB的开销。当然还需要保存序列号。

Syn Cookie技术

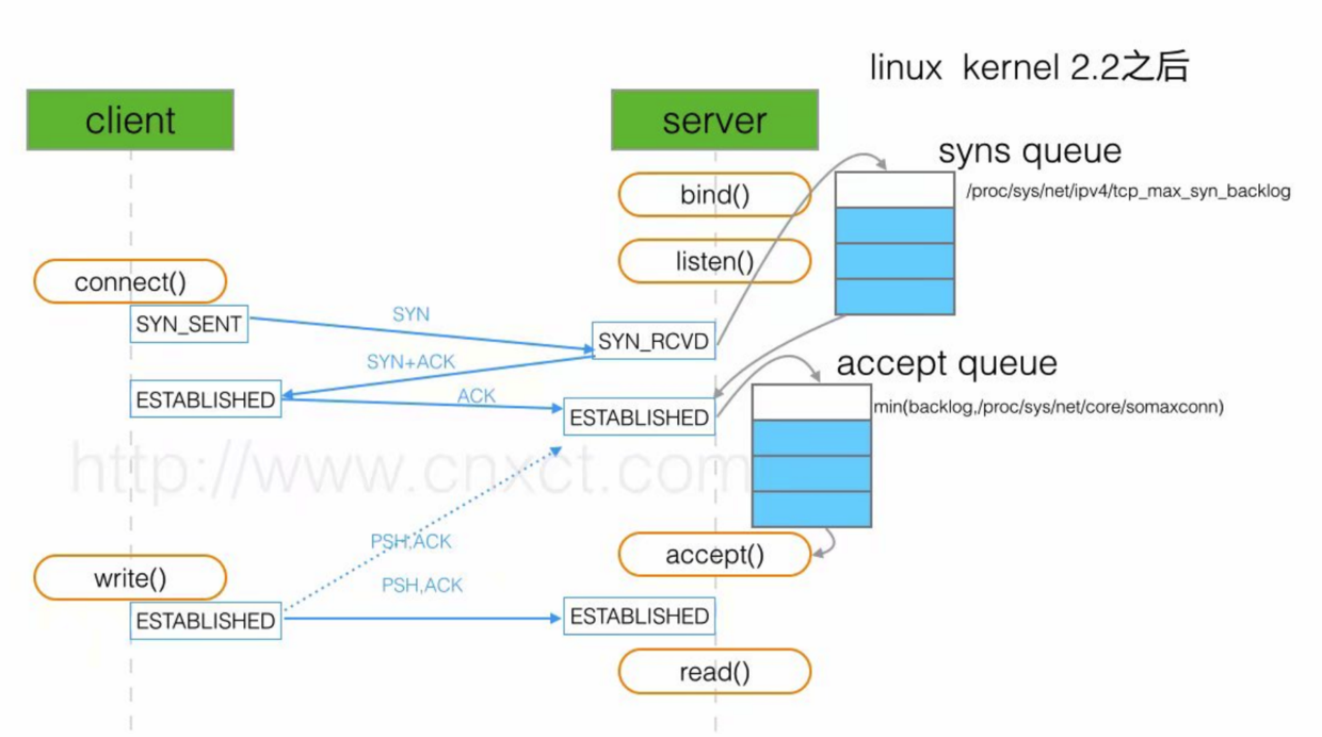
Syn Cookie技术则完全不使用任何存储资源，这种方法比较巧妙，它使用一种特殊的算法生成Sequence Number，这种算法考虑到了对方的IP、端口、己方IP、端口的固定信息，以及对方无法知道而己方比较固定的一些信息，如MSS(Maximum Segment Size，最大报文段大小，指的是TCP报文的最大数据报长度，其中不包括TCP首部长度。)、时间等，在收到对方 的ACK报文后，重新计算一遍，看其是否与对方回应报文中的（Sequence Number-1）相同，从而决定是否分配TCB资源。

使用SYN Proxy防火墙

一种方式是防止墙dqywb连接的有效性后，防火墙才会向内部服务器发起SYN请求。防火墙代服务器发出的SYN ACK包使用的序列号为c, 而真正的服务器回应的序列号为c', 这样，在每个数据报文经过防火墙的时候进行序列号的修改。另一种方式是防火墙确定了连接的安全后，会发出一个safe reset命令，client会进行重新连接，这时出现的syn报文会直接放行。这样不需要修改序列号了。但是，client需要发起两次握手过程，因此建立连接的时间将会延长。

连接队列

在外部请求到达时，被服务程序最终感知到前，连接可能处于SYN_RCVD状态或是ESTABLISHED状态，但还未被应用程序接受。



对应地，服务器端也会维护两种队列，处于SYN_RCVD状态的半连接队列，而处于ESTABLISHED状态但仍未被应用程序accept的为全连接队列。如果这两个队列满了之后，

就会出现各种丢包的情形。

查看是否有连接溢出

```
netstat -s | grep LISTEN
```

半连接队列满了

在三次握手协议中，服务器维护一个半连接队列，该队列为每个客户端的SYN包开设一个条目(服务端在接收到SYN包的时候，就已经创建了request_sock结构，存储在半连接队列中)，该条目表明服务器已收到SYN包，并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于Syn_RECV状态，当服务器收到客户的确认包时，删除该条目，服务器进入ESTABLISHED状态。

> 目前，Linux下默认会进行5次重发SYN-ACK包，重试的间隔时间从1s开始，下次的重试间隔时间是前一次的双倍，5次的重试时间间隔为1s, 2s, 4s, 8s, 16s, 总共31s, 称为指数退避，第5次发出后还要等32s才知道第5次也超时了，所以，总共需要 1s + 2s + 4s+ 8s+ 16s + 32s = 63s, TCP才会把断开这个连接。由于，SYN超时需要63秒，那么就给攻击者一个攻击服务器的机会，攻击者在短时间内发送大量的SYN包给Server(俗称 SYN flood攻击)，用于耗尽Server的SYN队列。对于应对SYN 过多的问题，linux提供了几个TCP参数：tcp_syncookies、tcp_synack_retries、tcp_max_syn_backlog、tcp_abort_on_overflow 来调整应对。

参数	作用
tcp_syncookies	SYNcookie将连接信息编码在ISN(initialsequencenumber)中返回给客户端，这时server不需要将半连接保存在队列中，而是利用客户端随后发来的ACK带回的ISN还原连接信息，以完成连接的建立，避免了半连接队列被攻击SYN包填满。
tcp_syncookies	内核放弃建立连接之前发送SYN包的数量。
tcp_synack_retries	内核放弃连接之前发送SYN+ACK包的数量
tcp_max_syn_backlog	默认为1000. 这表示半连接队列的长度，如果超过则放弃当前连接。
tcp_abort_on_overflow	如果设置了此项，则直接reset. 否则，不做任何操作，这样当服务器半连接队列有空了之后，会重新接受连接。Linux坚持在能力许可范围内不忽略进入的连接。客户端在这期间会重复发送sys包，当重试次数到达上限之后，会得到connection time out响应。

全连接队列满了

当第三次握手时，当server接收到ACK包之后，会进入一个新的叫 accept 的队列。

当accept队列满了之后，即使client继续向server发送ACK的包，也会不被响应，此时ListenOverflows+1，同时server通过tcp_abort_on_overflow来决定如何返回，0表示直接丢弃该ACK，1表示发送RST通知client；相应的，client则会分别返回read timeout 或者 connection reset

by peer。另外，tcp_abort_on_overflow是0的话，server过一段时间再次发送syn+ack给client（也就是重新走握手的第二步），如果client超时等待比较短，就很容易异常了。而客户端收到多个 SYN ACK 包，则会认为之前的 ACK 丢包了。于是促使客户端再次发送 ACK ，在 accept 队列有空闲的时候最终完成连接。若 accept队列始终满员，则最终客户端收到 RST 包（此时服务端发送syn+ack的次数超出了tcp_synack_retries）。

服务端仅仅只是创建一个定时器，以固定间隔重传syn和ack到服务端

参数	作用
tcp_abort_on_overflow	如果设置了此项，则直接reset. 否则，不做任何操作，这样当服务器半连接队列有空了之后，会重新接受连接。Linux坚持在能力许可范围内不忽略进入的连接。客户端在这期间会重复发送sys包，当重试次数到达上限之后，会得到connection time out响应。
min(backlog, somaxconn)	全连接队列的长度。

命令

netstat -s命令

```
[root@server ~]#  netstat -s | egrep "listen|LISTEN"
667399 times the listen queue of a socket overflowed
667399 SYNs to LISTEN sockets ignored
```

上面看到的 667399 times ，表示全连接队列溢出的次数，隔几秒钟执行下，如果这个数字一直在增加的话肯定全连接队列偶尔满了。

```
[root@server ~]#  netstat -s | grep TCPBacklogDrop
```

查看 Accept queue 是否有溢出

ss命令

```
[root@server ~]#  ss -lnt
State Recv-Q Send-Q Local Address:Port Peer Address:Port
LISTEN     0      128 *:6379 *:*
LISTEN     0      128 *:22 *:*
```

如果State是listen状态，Send-Q 表示第三列的listen端口上的全连接队列最大为50，第一列Recv-Q为全连接队列当前使用了多少。
非 LISTEN 状态中 Recv-Q 表示 receive queue 中的 bytes 数量；Send-Q 表示 send queue 中的 bytes 数值。

小结

当外部连接请求到来时，TCP模块会首先查看max_syn_backlog，如果处于SYN_RCVD状态的连接数目超过这一阈值，进入的连接会被拒绝。根据tcp_abort_on_overflow字段来决定是直接丢弃，还是直接reset.

从服务端来说，三次握手中，第一步server接受到client的

syn后，把相关信息放到半连接队列中，同时回复syn+ack给client. 第三步当收到客户端的ack, 将连接加入到全连接队列。

一般，全连接队列比较小，会先满，此时半连接队列还没满。如果这时收到syn报文，则会进入半连接队列，没有问题。但是如果收到了三次握手中的第3步(ACK)，则会根据tcp_abort_on_overflow字段来决定是直接丢弃，还是直接reset.此时，客户端发送了ACK, 那么客户端认为三次握手完成，它认为服务端已经准备好了接收数据的准备。但此时服务端可能因为全连接队列满了而无法将连接放入，会重新发送第2步的syn+ack, 如果这时有数据到来，服务器TCP模块会将数据存入队列中。一段时间后，client端没收到回复，超时，连接异常，client会主动关闭连接。

“三次握手，四次挥手”redis实例分析

1. 我在dev机器上部署redis服务，端口号为6379,
2. 通过tcpdump工具获取数据包，使用如下命令

```
tcpdump -w /tmp/a.cap port 6379 -s0
```

-w把数据写入文件，-s0设置每个数据包的大小默认为68字节，如果用-S 0则会抓到完整数据包

3. 在dev2机器上用redis-cli访问dev:6379, 发送一个ping, 得到回复pong
4. 停止抓包，用tcpdump读取捕获到的数据包

```
tcpdump -r /tmp/a.cap -n -nn -A -x| vim -
```

（-x 以16进制形式展示，便于后面分析）

共收到了7个包。

抓到的是IP数据包，IP数据包分为IP头部和IP数据部分，IP数据部分是TCP头部加TCP数据部分。

IP的数据格式为：

它由固定长度20B+可变长度构成。

```
10:55:45.662077 IP dev2.39070 > dev.6379: Flags [S], seq
4133153791, win 29200, options [mss 1460,sackOK,TS val
2959270704 ecr 0,nop,wscale 7], length 0
        0x0000:  4500 003c 08cf 4000 3606 14a5 0ab3 b561
        0x0010:  0a60 5cd4 989e 18eb f65a ebff 0000 0000
        0x0020:  a002 7210 872f 0000 0204 05b4 0402 080a
        0x0030:  b062 e330 0000 0000 0103 0307
```

对着IP头部格式，来拆解数据包的具体含义。

字节值	字节含义
0x4	IP版本为ipv4
0x5	首部长度为5 * 4字节=20B
0x00	服务类型，现在基本都置为0
0x003c	总长度为3*16+12=60字节，上面所有的长度就是60字节
0x08cf	标识。同一个数据报的唯一标识。当IP数据报被拆分时，会复制到每一个数据中。
0x4000	`3bit 标志 + 13bit 片偏移`。3bit 标志对应 R、DF、MF。目前只有后两位有效，DF位：为1表示不分片，为0表示分片。MF：为1表示“更多的片”，为0表示这是最后一片。13bit 片位移：本分片在原先数据报文中相对首位的偏移位。（需要再乘以8 ）
0x36	生存时间TTL。IP报文所允许通过的路由器的最大数量。每经过一个路由器，TTL减1，当为 0 时，路由器将该数据报丢弃。TTL 字段是由发送端初始设置一个 8 bit字段.推荐的初始值由分配数字 RFC 指定。发送 ICMP 回显应答时经常把 TTL 设为最大值 255。TTL可以防止数据报陷入路由循环。 此处为54.
0x06	协议类型。指出IP报文携带的数据使用的是哪种协议，以便目的主机的IP层能知道要将数据报上交到哪个进程。TCP 的协议号为6，UDP 的协议号为17。ICMP 的协议号为1，IGMP 的协议号为2。该 IP 报文携带的数据使用 TCP 协议，得到了验证。
0x14a5	16bitIP首部校验和。
0x0ab3b561	32bit源ip地址。
0x0a605cd4	32bit目的ip地址。

剩余的数据部分即为TCP协议相关的。TCP也是20B固定长度+可变长度部分。

字节值	字节含义
0x989e	16bit源端口。1*16*16+8*16*16+14*16+11=39070
0x18eb	16bit目的端口6379
0xf65a ebff	32bit序列号。4133153791
0x0000 0000	32bit确认号。
0xa	4bit首部长度，以4byte为单位。共10*4=40字节。因此TCP报文的可选长度为40-20=20
0b000000	6bit保留位。目前置为0.
0b000010	6bitTCP标志位。从左到右依次是紧急 URG、确认 ACK、推送 PSH、复位 RST、同步 SYN 、终止 FIN。
0x7210	滑动窗口大小，滑动窗口即tcp接收缓冲区的大小，用于tcp拥塞控制。29200
0x872f	16bit校验和。
0x0000	紧急指针。仅在 URG = 1时才有意义，它指出本报文段中的紧急数据的字节数。当 URG = 1 时，发送方 TCP 就把紧急数据插入到本报文段数据的最前面，而在紧急数据后面的数据仍是普通数据。

可变长度部分，协议如下：

字节值	字节含义
0x0204 05b4	最大报文长度为，05b4=1460. 即可接收的最大包长度，通常为MTU减40字节，IP头和TCP头各20字节
0x0402	表示支持SACK
0x080a b062 e330 0000 0000	时间戳。Ts val=b062 e330=2959270704, ecr=0

0x01	无操作
0x03 0307	窗口扩大因子为7. 移位7, 乘以128

这样第一个包分析完了。dev2向dev发送SYN请求。也就是三次握手中的第一次了。

```
SYN seq(c)=4133153791
```

第二个包，dev响应连接，ack=4133153792. 表明dev下次准备接收这个序号的包，用于tcp字节注的顺序控制。dev（也就是server端）的初始序号为seq=4264776963, syn=1.

```
SYN ack=seq(c)+1 seq(s)=4264776963
```

第三个包，client包确认，这里使用了相对值应答。seq=4133153792, 等于第二个包的ack.
ack=4264776964.

```
ack=seq(s)+1, seq=seq(c)+1
```

至此，三次握手完成。接下来就是发送ping和pong的数据了。

接着第四个包。

```
10:55:48.090073 IP dev2.39070 > dev.6379: Flags [P.], seq
1:15, ack 1, win 229, options [nop,nop,TS val 2959273132
ecr 3132256230], length 14
        0x0000:  4500 0042 08d1 4000 3606 149d 0ab3 b561
        0x0010:  0a60 5cd4 989e 18eb f65a ec00 fe33 5504
        0x0020:  8018 00e5 4b5f 0000 0101 080a b062 ecac
        0x0030:  bab2 6fe6 2a31 0d0a 2434 0d0a 7069 6e67
        0x0040:  0d0a
```

tcp首部长度为32B, 可选长度为12B. IP报文的总长度为66B, 首部长度为20B, 因此TCP数据部分长度为14B.
seq=0xf65a ec00=4133153792
ACK, PSH. 数据部分为2a31 0d0a 2434 0d0a 7069 6e67 0d0a

```
0x2a31          -> *1
0x0d0a          -> \r\n
0x2434          -> $4
0x0d0a          -> \r\n
0x7069 0x6e67   -> ping
0x0d0a          -> \r\n
```

dev2向dev发送了ping数据，第四个包完毕。

第五个包，dev2向dev发送ack响应。
序列号为0xfe33 5504=4264776964, ack确认号为0xf65a ec0e=4133153806=(4133153792+14).

第六个包，dev向dev2响应pong消息。序列号fe33 5504，确认号f65a ec0e, TCP头部可选长度为12B, IP数据报总长度为59B, 首部长度为20B, 因此TCP数据长度为7B. 数据部分2b50 4f4e 470d 0a, 翻译过来就是`+PONG\r\n`.

至此，Redis客户端和Server端的三次握手过程分析完毕。

总结

“三次握手，四次挥手”看似简单，但是深究进去，还是可以延伸出很多知识点的。比如半连接队列、全连接队列等

等。以前关于TCP建立连接、关闭连接的过程很容易就会忘记，可能是因为只是死记硬背了几个过程，没有深入研究背后的原理。

所以，“三次握手，四次挥手”你真的懂了吗？欢迎一起交流~~

参考资料

【redis】https://segmentfault.com/a/1190000015044878

【tcp option】https://blog.csdn.net/wdscq1234/article/details/52423272

【滑动窗口】https://www.zhihu.com/question/32255109

【全连接队列】http://jm.taobao.org/2017/05/25/525-1/

【client fooling】https://github.com/torvalds/linux/commit/5ea8ea2cb7f1d0db15762c9b0bb9e7330425a071

【backlog RECV_Q】http://blog.51cto.com/59090939/1947443

【定时器】https://www.cnblogs.com/menghuanbiao/p/5212131.html

【队列图示】https://www.itcodemonkey.com/article/5834.html

【tcp flood攻击】https://www.cnblogs.com/hubavyn/p/4477883.html

【MSS MTU】https://blog.csdn.net/LoseInVain/article/details/53694265