UNP 总结

秋招快速复习之网络

刘洋洋

2018-7-26

CHINA
南三舍 425

目录

1 TCP 协议（基础）

1.1 TCP 首部格式

图1 TCP 首部格式

1.2 序号和确认号

1.21 序号

序号占用 4 字节，即 32 位。它的范围是 [0,232−1][0,232−1]，也就是说一共有 4 294 967 296 个序号。TCP 协议中的序号，指的是报文段序号。

- 字节序号

TCP 连接中，为传送的字节流（数据）中的**每一个字节按顺序编号**。也就是说，**在一次 TCP 连接建立的开始，到 TCP 连接的断开**，你要传输的所有数据的每一个字节都要编号。这个序号称为**字节序号**。

- 初始序号 ISN

当新**连接建立**的时候，第一个字节数据的序号称为 ISN(Initial Sequence Number)，即初始序号。ISN 一开始并不一定就是 1。在 RFC (规定网络协议的文档)中规定，ISN 的分配是根据时间来的。当**操作系统初始化的时候**，有一个全局变量假设为 g_number 被初始化为 1（或 0），然后每隔 4us 加 1. 当 g_number 达到最大值的时候又绕回到 0.当新连接建立时，就把 g_number 的值赋值给 ISN.

初始序号是非常非常重要的概念，它告诉对端，第一个报文段是谁！而三次握手的目的，就是为了确认初始序号，这个在后面会讲。

- 报文段序号

如果一个 TCP 报文段的序号为 301，它携带了 100 字节的数据，就表示这 100 个字节的数据的字节序号范围是 [301, 400]，该报文段携带的第一个字节序号是 301，最后一个字节序号是 400.

注意：**序号字段**只有在下面两种情况的任意一种才有意义：

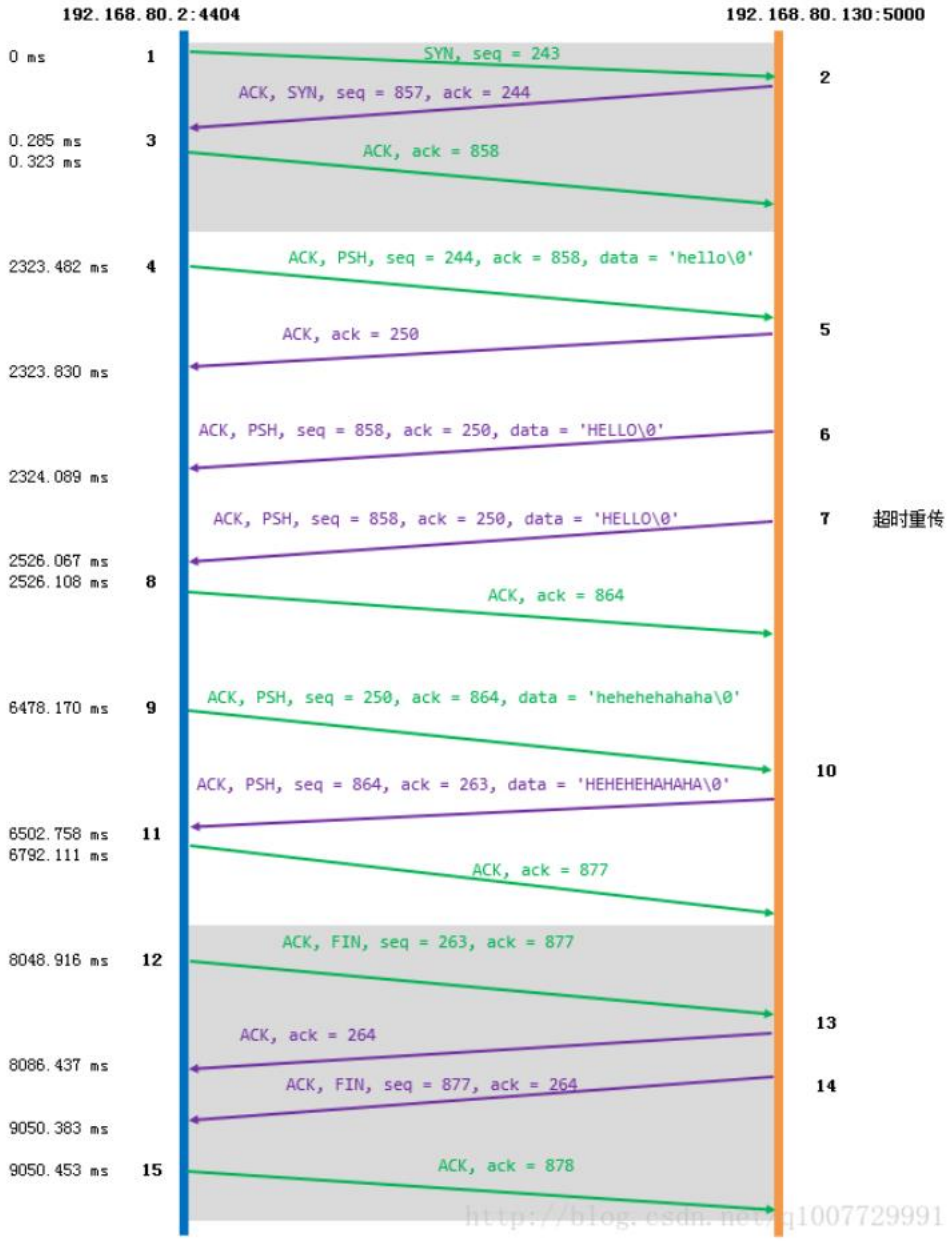
- 数据字段至少包含一个字节
- 这是一个 SYN 段，或者是 FIN 段，或者是 RST 段。

1.22 确认号

如果你还记得前面你和你对象发短信的例子的话，这里就不难了。每传送一个 TCP 段，都要等待对方回复一个确认。不过这种方式效率太低，在 TCP 协议中，一般采用**累积确认**的方式，即每传送多个连续 TCP 段，可以只对最后一个 TCP 段进行确认。

对方通过回复一个确认号，来表示确认已经接收到了哪个 TCP 段。比如发送方发送了一个**报文段序号**为 301 的 TCP 段，这个段携带了 100 字节数据，则接收方应当回复的确认号是 401，它表示接收方已经收到了**字节序号为** [0, 400] 的数据，现在期望你发送字节序号为 401 以及以后的数据。

只有当 ACK 标志位被置位的时候，确认号这个字段才有效。
一次完整的 TCP 连接到释放的过程


192.168.80.2:4404
192.168.80.130:5000
0 ms
1
SYN, seq = 243
2
ACK, SYN, seq = 857, ack = 244
0.285 ms
3
0.323 ms
ACK, ack = 858
2323.482 ms
4
ACK, PSH, seq = 244, ack = 858, data = 'hello\0'
ACK, ack = 250
5
2323.830 ms
ACK, PSH, seq = 858, ack = 250, data = 'HELLO\0'
6
2324.089 ms
ACK, PSH, seq = 858, ack = 250, data = 'HELLO\0'
7
超时重传
2526.067 ms
2526.108 ms
8
ACK, ack = 864
6478.170 ms
9
ACK, PSH, seq = 250, ack = 864, data = 'hehehehahaha\0'
10
ACK, PSH, seq = 864, ack = 263, data = 'HEHEHEHAHAHA\0'
6502.758 ms
11
6792.111 ms
ACK, ack = 877
ACK, FIN, seq = 263, ack = 877
8048.916 ms
12
13
ACK, ack = 264
8086.437 ms
ACK, FIN, seq = 877, ack = 264
14
9050.383 ms
9050.453 ms
15
ACK, ack = 878
http://blog.csdn.net/q1007729991

1.3 建立连接

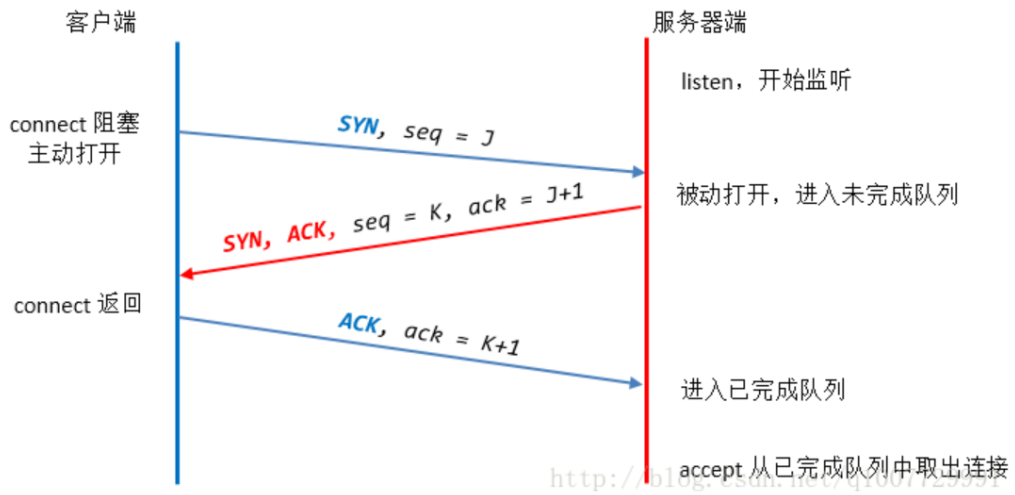
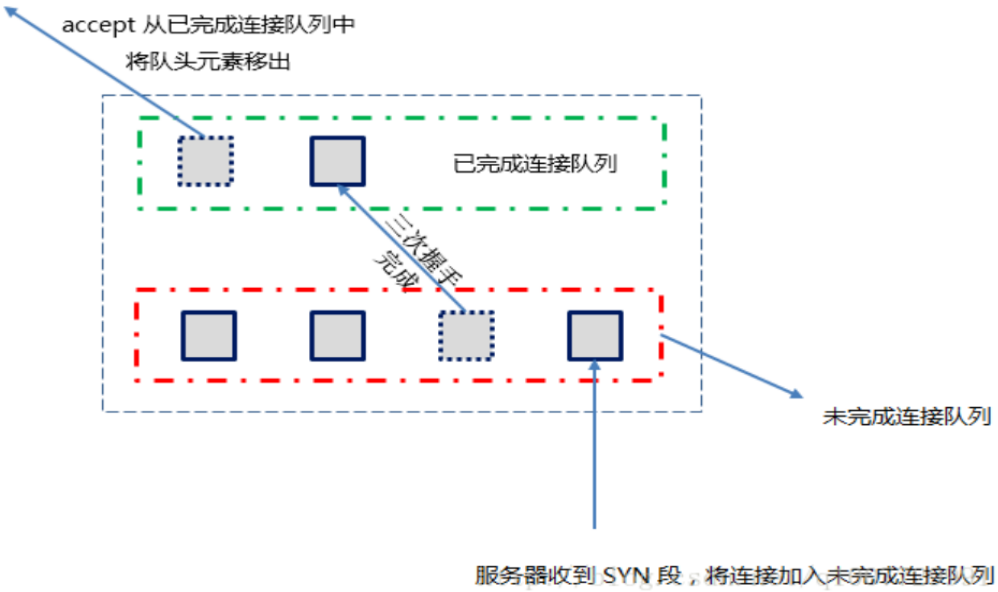


图1 三次握手

未完成队列和**已完成队列**，这两个队列合在一起称为**未决连接队列**

未完成连接队列（incomplete connection queue），服务器只要收到了 SYN 段，就将该连接加入到未完成连接队列。
已完成连接队列（complete connection queue），服务器收到客户端对 SYN 段的确认，就将未完成连接队列中的连接移入到已完成连接队列。



这个两个队列元素个数之和不超过 backlog，这是函数 listen 的参数。

1.31 为什么要三次握手

- 三次握手是有效的

建立连接的一个基础条件： ACK + 超时重传的机制

(1) A —— SYN ——> B
(2) B —— ACK ——> A
(3) B —— SYN ——> A
(4) A —— ACK ——> B
可以看出来步骤 2 和 3 可以合并成一个。因此，三次握手是有效的，正确的。

- 为什么要三次握手

从两方面解释：
(1) 防止某个在网络中滞留的 SYN 段突然又被传送到服务器端
(2) 三次握手的目的：三次握手的目的，即建立 TCP 连接的目的，是为了确认双方的初始序号 —— ISN.

1.4 断开连接

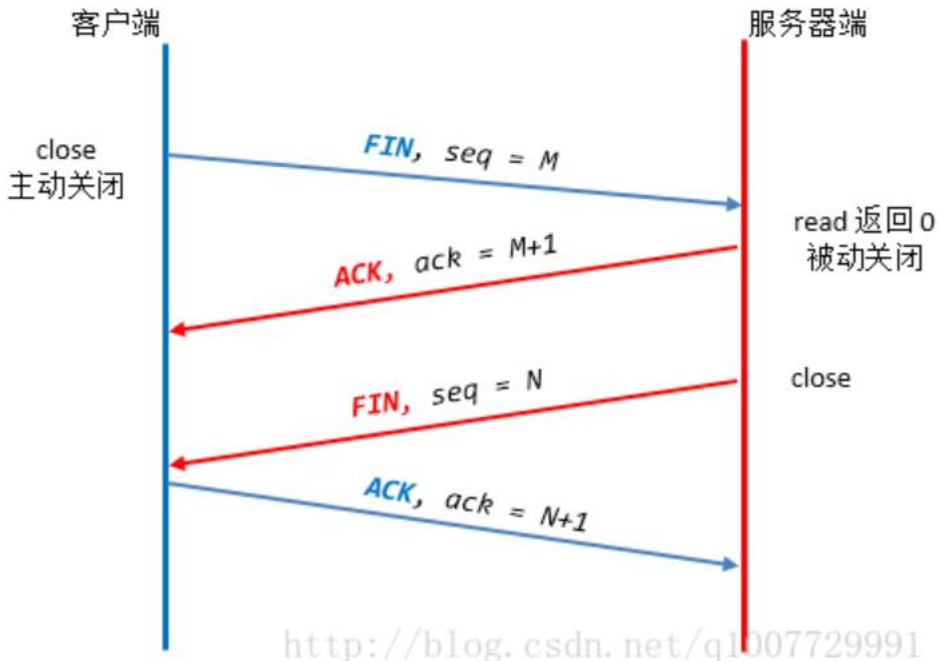


图1 四次挥手

(1) 首先，由客户端调用 close，将这一端称为主动关闭(active close)。然后该端发送一个 FIN 段到对端。

(2) 接收到 FIN 段的服务器执行被动关闭(passive close)。接下来，接收到此 FIN 段的服务器回复 ACK 进行确认（实际上是由内核自动完成回复的），同时内核会传递一个文件结束符 EOF（放在缓冲区末尾） 给应用进程。

(3) 一段时间后……服务器端也没有数据要发送给对端了，调用 close，这导致服务器端也发送一个 FIN 段到对端。

(4) 客户端接收到 FIN 后，回复 ACK 进行确认。

注意：

- 执行主动关闭的一方既可以是客户端，也可以是服务器，这两者之间是对等的。
- 通常断开连接需要 4 个 TCP 段。但是有时候不一定是这样。某些情况下，步骤 1 中的 FIN 段会随着数据一起发送到对端；另一种情况，**步骤 2 和步骤 3 有可能被合并成一个 TCP 段**

1.41 为什么是四次挥手

实际上，这是 TCP 连接的半关闭特性造成的。

因为 TCP 连接是全双工的（数据在两个方向上可以同时传递，图 4），因此每个方向就必须能够单独的关闭。就比如客户端执行了半关闭操作后，只是通知服务器它没有数据要发送了，并不代表它不能接收数据。因此，只要服务器还没有主动关闭，就能够向客户端继续发送数据。也就相当于在步骤 2 和 步骤 3 之间，服务器仍然能够向客户端发送数据。

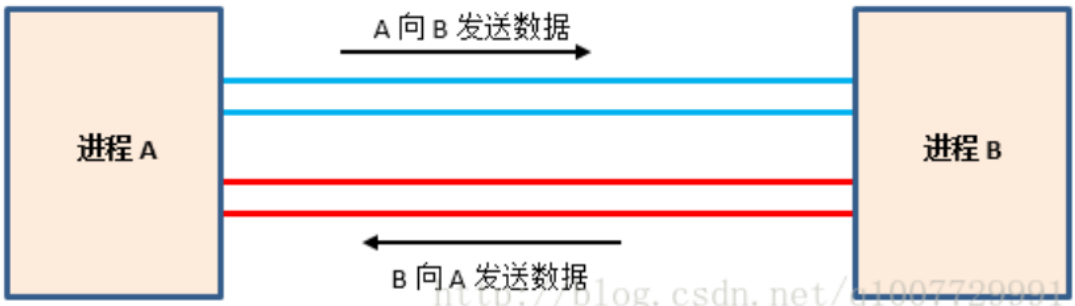


图4 TCP 连接是全双工的

四次挥手为被动关闭的一方提供了很大的伸缩空间，让被动关闭一方**有机会**继续向主动关闭一方发数据。**如果**TCP 协议标准要求步骤 2 和 步骤 3 合并成一个步骤，这种伸缩

空间就没有了，也就是说只要有一端关闭了，另一方就没有机会继续发送数据。

1.5 MSS（Maximum Segment Size）

MSS：最大报文段长度。在连接建立时，即在发送 SYN 段的时候，同时会将 MSS 发送给对方（MSS 只能出现在 SYN 报文段中!!!）。告诉对端它期望接收的 TCP 报文段数据部分的最大长度。
MSS 保存在 TCP 首部的可选项中

1.51 MSS 的作用

一般来说，TCP 报文段携带的数据当然是越多越好。

但是实际上并非如此。因为网络传输数据时，数据是最终是要交付到链路层协议上的，也就是说最后要封装成“帧”。**二型以太网（Ethernet Type 2）**中规定，**帧的大小不能超过 1518 个字节**（14 字节的帧头 + 4 字节帧校验和 + 最多 1500 字节数据）。所以 IP 数据报的大小如果超过了 1500 字节，要想交付给链路层就必须进行“分片”。
“分片”指的是一个 IP 数据报太大，需要拆分成一个一个的小段，变成多个 IP 数据报。

这种分片显然是不利的，有一定的开销。为了避免分片开销，我们希望 IP 数据报的大小不超过 1500 字节。除去 IP 数据报的首部 20 字节，也就是希望 TCP 报文段不超过 1480 字节。再减去 TCP 报文段首部 20 字节，也就是 TCP 携带的数据不超过 1460 字节。

图4 MSS 大小

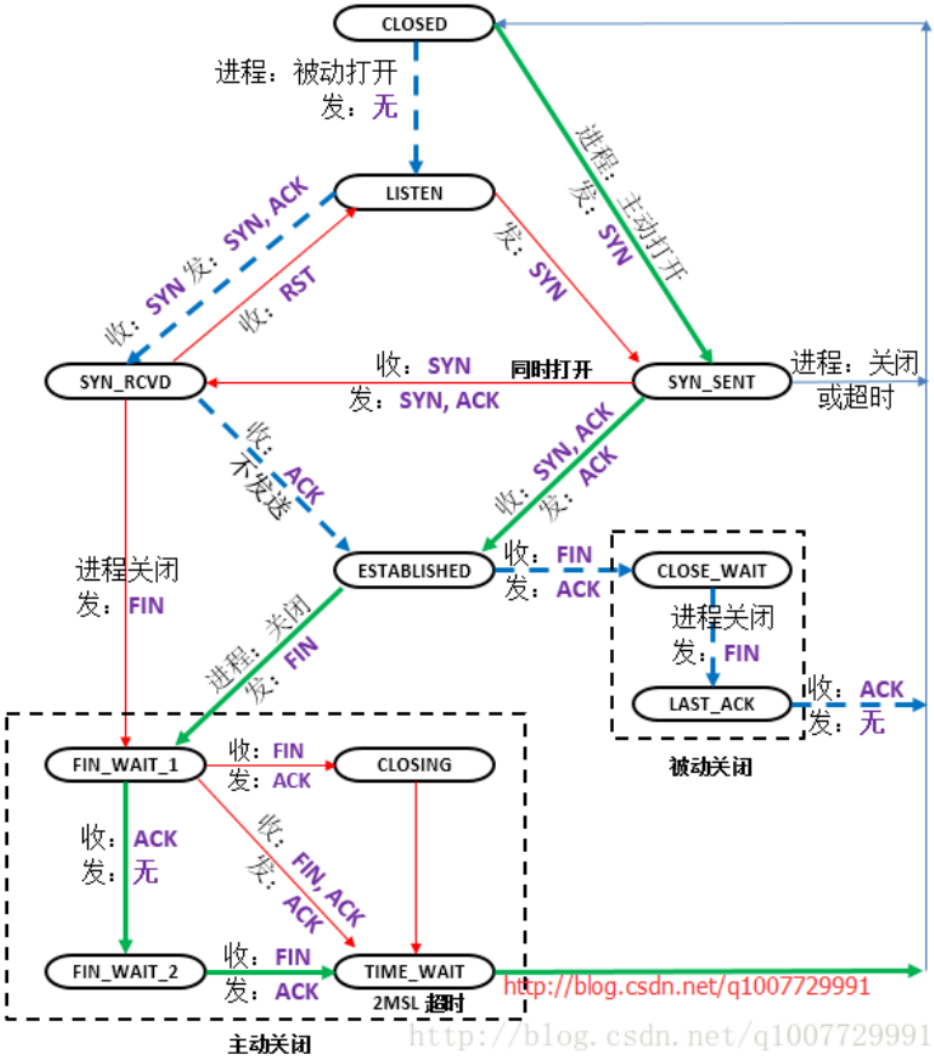
实际上，链路层对这种帧数据长度的限制称为**最大传输单元**(Maximum Transmission Unit, MTU)。

1.52 MTU

- MSS 是软件层的概念，它是由软件控制的
- MTU 是硬件（比如网卡出口）的属性，是指二层链路层帧携带的数据最大大小。

如果不指定 MSS，当数据部分超过了 MTU，就会进行分片传输。

1.6 状态机



- 绿色粗实线和蓝色粗虚线表示正常的状态变迁。
- 红色细实线表示异常变迁。
- 顶层的 CLOSED 状态是一个假想的起点和终点，并不是一个真正的状态。

这里再次将三次握手和四次挥手的时序图画出来，同时标记上变迁的状态

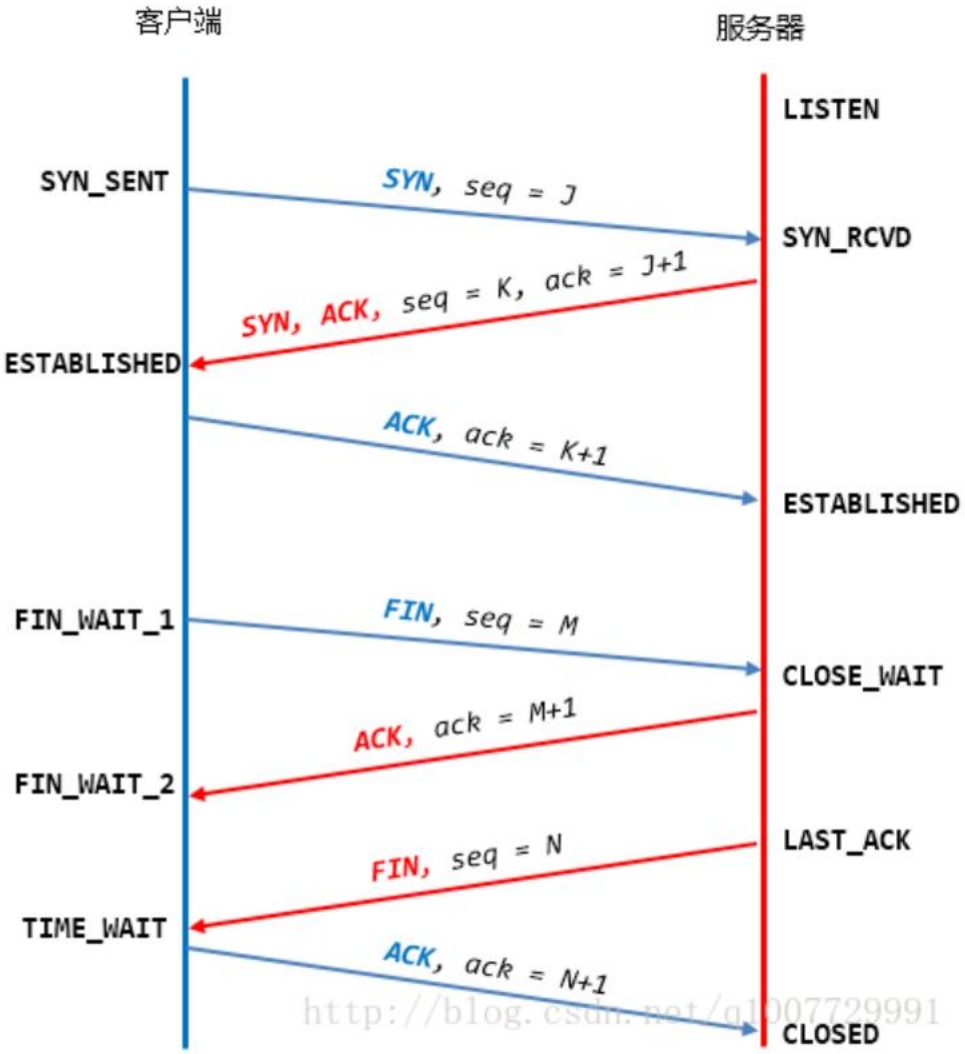


图2 TCP 正常连接建立和终止对应的状态

1.7 TIME_WAIT 状态

主动关闭一方最后的状态是 TIME_WAIT

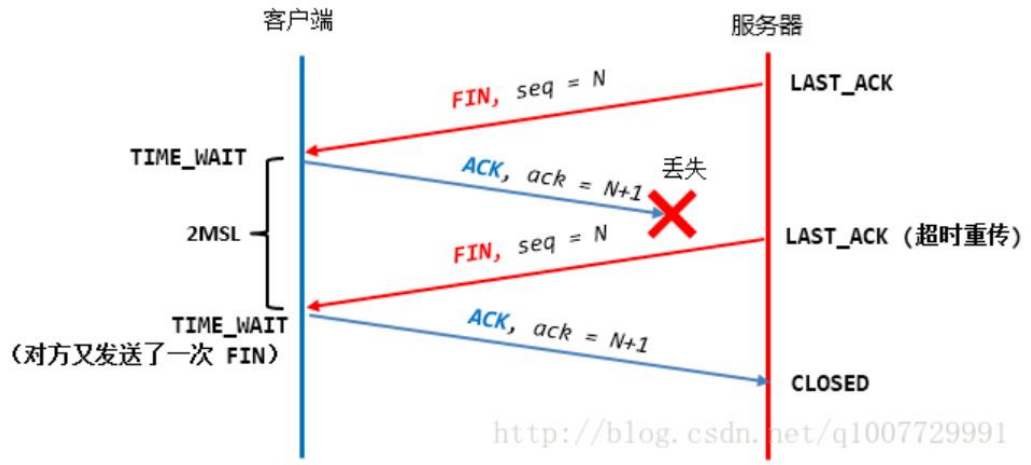
TIME_WAIT 状态，又称为 **2MSL 等待状态**。只有**主动关闭一方**才能进入 TIME_WAIT 状态。
MSL（Maximum Segment Lifetime）表示报文段最大生存时间，它表示任何报文段被丢弃前在网络内的最长时间，实际上这个时间和 TTL 有关（TTL 是 IP 协议中的一个概念，表示能够经历的路由器的跳数，这个跳数是有限制的，最大值为 255）。

尽管 MSL 的单位是时间而不是跳数，我们仍然假设：具有最大跳数（255）的报文在网络中存在的时间不可能超过 MSL 秒。

当 TCP 协议进入 TIME_WAIT 状态时，必须要在这个状态停留 2 倍 MSL 的时间。

- 等待 2MSL 的第一个原因

可以**防止连接终止的最后一个 ACK 丢失**。假设最后一个 ACK 在到达对端时恰好消失，此时对端已经等待了一个 **RTT**（报文段往返时间），于是进行重传最后一个 FIN，经过 0.5RTT 后到达对端。



- 等待 2MSL 的第二个原因

  假设在 ip1:port1 和 ip2:port2 建立了一个连接 A，发送完数据后关闭连接 A。如果没有 TIME_WAIT 状态，我们又立即在 ip1:port1 和 ip2:port2 建立了一个连接 B（虽然这种事情概率很小，但是仍然存在）。

  很不幸的是，连接 A 它有一个重复的 TCP 段被连接 B 收到了，然而连接 B 并不知道这个 TCP 段是连接 A 中的旧报文，这会造成错误！

  如果有了 TIME_WAIT 状态，等待 2MSL，就足以让连接 A 中重复的报文在网络中消逝；另一方面，TCP 协议规定，处于 TIME_WAIT 状态的端口，是无法建立新的连接的。这样就保证了每成功建立一个新连接时，旧连接中的重复 TCP 段都已消逝。

1.71 TIME_WAIT 状态的影响

如果主动关闭的是服务器端：如果 TCP 处于 TIME_WAIT 状态，会进行 2MSL 时间的等待，在这个时间内，定义此连接的本地端口（服务器端口）不能再次使用。

1.8 FIN_WAIT2 状态

当主动关闭一方进入 FIN_WAIT2 状态时，只要对端还没有发送 FIN 段过来（处于 CLOSE_WAIT 状态，等等再关闭，我还有数据要发送），就会一直停留在这个状态。

很不幸，主动关闭一方有可能永远处于 FIN_WAIT2 状态，只要对方不发送 FIN 段的话（比如对端在 CLOSE_WAIT 状态时突然断电、网线掉了）。

在有些系统实现中，为了防止这种无限 FIN_WAIT2，设置了一个定时器。如果这个连接空闲 10 分钟 75 秒，TCP 将进入 CLOSED 状态。实际上，这是违反协议的，但又未尝不可呢？

1.9 连接异常与 RST

当 TCP 连接出现严重的错误时，必须释放连接。通过将 TCP 首部中的 RST 标志位置 1，就可以通知对端发生错误，以终止连接。

正常情况下，关闭连接的方式是发送 FIN 段，即四次挥手。这种方式也称为**有序释放**(orderly release)。

也可以通过发送 RST 段给对端来释放连接，这种方式称为**异常释放**(abortive release)。

异常终止连接有两个特点：

- 丢弃任何尚未发送的数据，立即发送 RST 报文段
- RST 接收方会区分另一端是异常关闭还是正常关闭，从而做出不同响应

还有几点需要注意：

- 服务器端并没有给这个 RST 回复 ACK（任何收到 RST 段的一方根本不会为这个 RST 进行确认）。
- 主动发送 RST 段的一方，不会进入 TIME_WAIT 状态。

1.10 半打开与同时关闭

1、半打开

如果一方已经关闭或异常终止，而另一方却对此毫不知情，这种连接就称为半打开的。

假设在主机 A 和 B 之间建立了一条 TCP 连接，由于停电导致主机 B 异常关机。接下来，我们分三种情况讨论，A 此时继续给 B 发数据，会有什么现象。

- B 仍然关机
- B 开机，但是未启动服务
- B 开机并重新启动了服务

处于半连接的 A 向主机 B 发送数据：

- 如果主机 B 仍然断网或者已经连接上网络，但是服务未启动，A 向 B 发送数据，经过数次超时重传后放弃连接，并发送 RST 段给对方（不一定非得发送，这系统实现有关）。
- 如果主机 B 已经连接上网络且重新启动了服务，A 向 B 发送数据，B 收到后因为不认识这个连接，向 A 发送 RST 段。

2、同时关闭

同时关闭，指的是已建立连接的两端**同时发起主动关闭**。

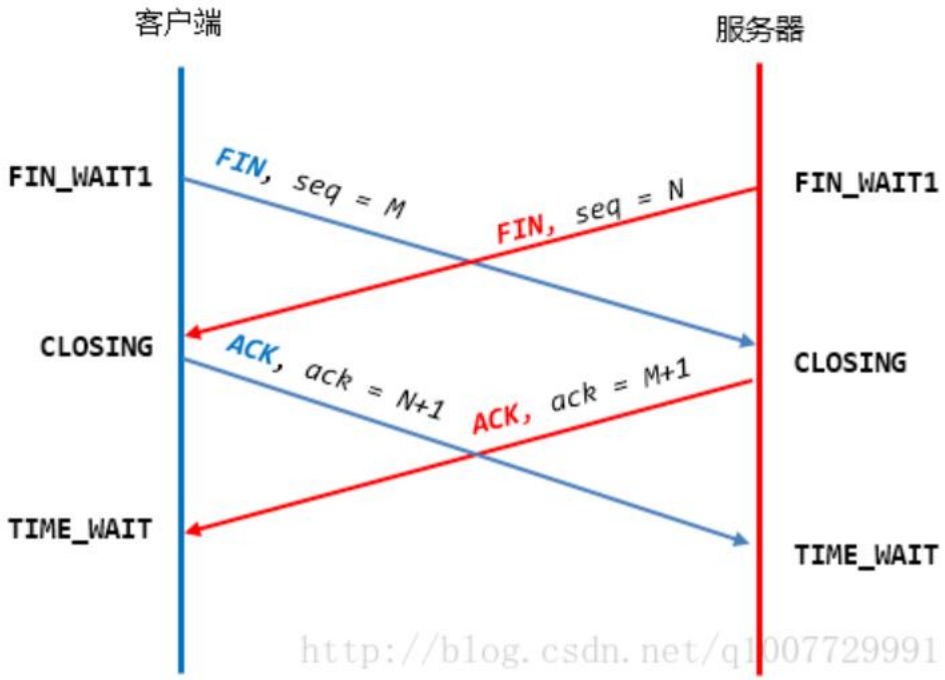


图1 同时关闭

应用层的进程，同时发出关闭命令，两端均从 ESTABLISHED 变为了 FIN_WAIT1 状态，同时发送 FIN 段给对方。

然而发送完 FIN 段后，并未收到对端的 ack 而是对方发来的一个 FIN 段，于是直接进入 CLOSING 状态，CLOSING 状态是一个新状态，之前我们没有遇到过，只在 TCP 状态机图里看到过。现在你要记住，CLOSING 状态是由于同时关闭导致的。

又过了一会儿，TCP 接收到 ack 后，进入 TIME_WAIT 状态。

2 TCP 协议（进阶）

2.1 延迟 ACK

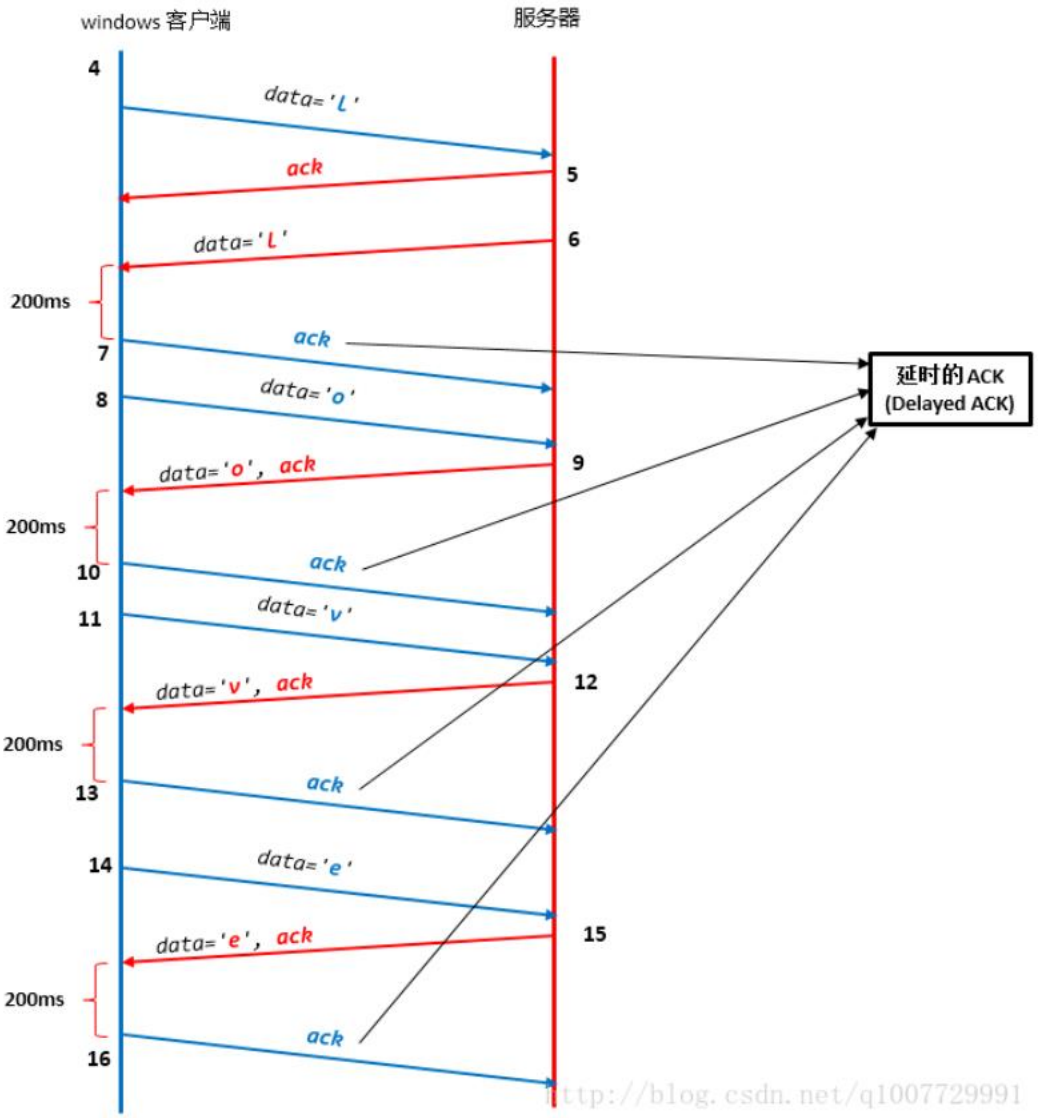
TCP 协议中，需要对接收到 TCP 段进行确认。有两种方式可以减少 TCP 报文段. 一种是累积确认，另一种是捎带确认。

- 累积确认

有时候，发送方发送速度非常快，接收方一下接收到了好几个 tcp 段，可以通过累积确认的方式，一次确认好几个 tcp 段，这样减少报文段的传输。

- 捎带确认

有时候，双方互相发送数据，当接收到对方的 tcp 段后，先不着急确认，而是等待一会儿，连同数据和 ack 一起发送过去，这种情况叫捎带确认。如果等了一会儿（到时间了），接收方还没有数据要发送，那就直接回复一个纯 ack 过去，这样的 ack 称为**延时的 ack（Delayed ACK）**.



上图是 windows 中的情况。对于 Linux：

当协议栈接受到 TCP 数据时，并不一定会立刻发送 ACK 响应，而是倾向于等待一个超时或者满足特殊条件时再发送。对于 Linux 实现，这些特殊条件如下：

1）收到的数据已经超过了 full frame size

2）或者处于快速回复模式

3）或者出现了乱序的包

4）或者接收窗口的数据足够多

如果接收方有数据回写，则 ACK 也会搭车一起发送。当以上条件都不满足时，接收方会延迟 40ms 再回应 ACK。

有时候，我们不希望 ack 被延时发送，可以设置 TCP_QUICKACK（>Linux 2.4.4）。另外，delay ack 会对 nagle 算法造成影响，导致数据传输速率变慢。关于 nagle 算法，后面会有分析。

2.2 Nagle 算法

Nagle 算法要求：

- 一个 TCP 连接上最多只能有一个未被确认的未完成的小分组，在它到达目的地前，不能发送其它分组。
- 在上一个小分组未到达目的地前，即还未收到它的 ack 前，TCP 会收集后来的小分组。当上一个小分组的 ack 收到后，TCP 就将收集的小分组合并成一个大分组发送出去。

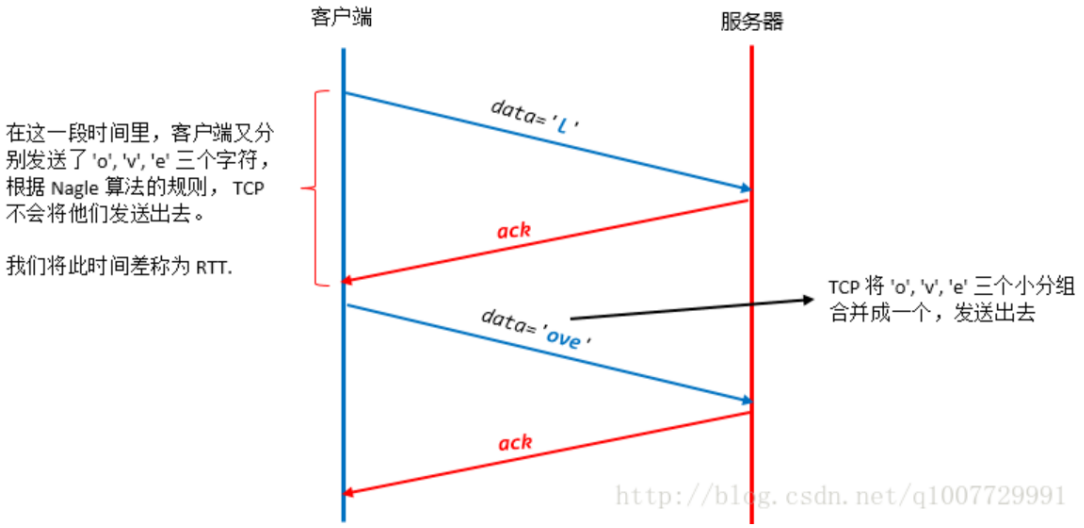


图1 Nagle 算法如何处理小分组

默认情况下 Nagle 算法就是开启的。

2.3 滑动窗口

滑动窗口，它的大小表示目前还能接收多少字节的数据。TCP 每次收到对方发来的报文，都会检查**窗口大小**字段
知道了对方的窗口大小后，就知道对方目前还能接收多少数据，接收的数据字节序号是 TCP 段中的 ACK 的值到 ACK + 窗口大小，即 [ACK,ACK+**窗口大小**)[ACK,ACK+窗口大小).

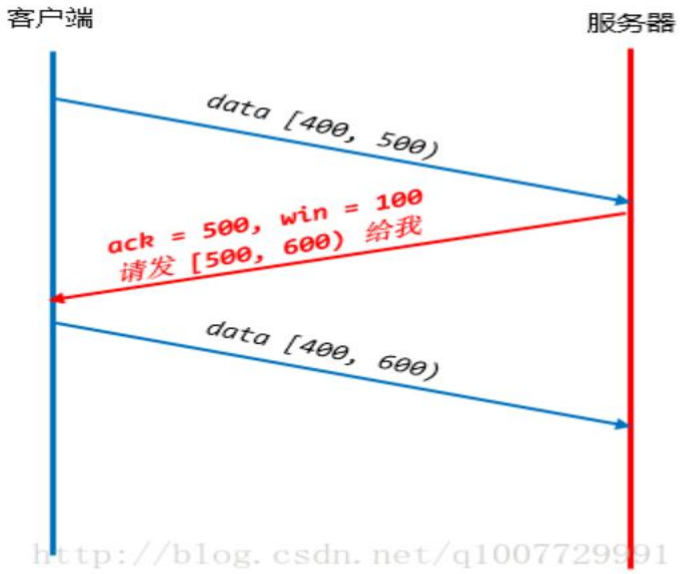


图2 滑动窗口

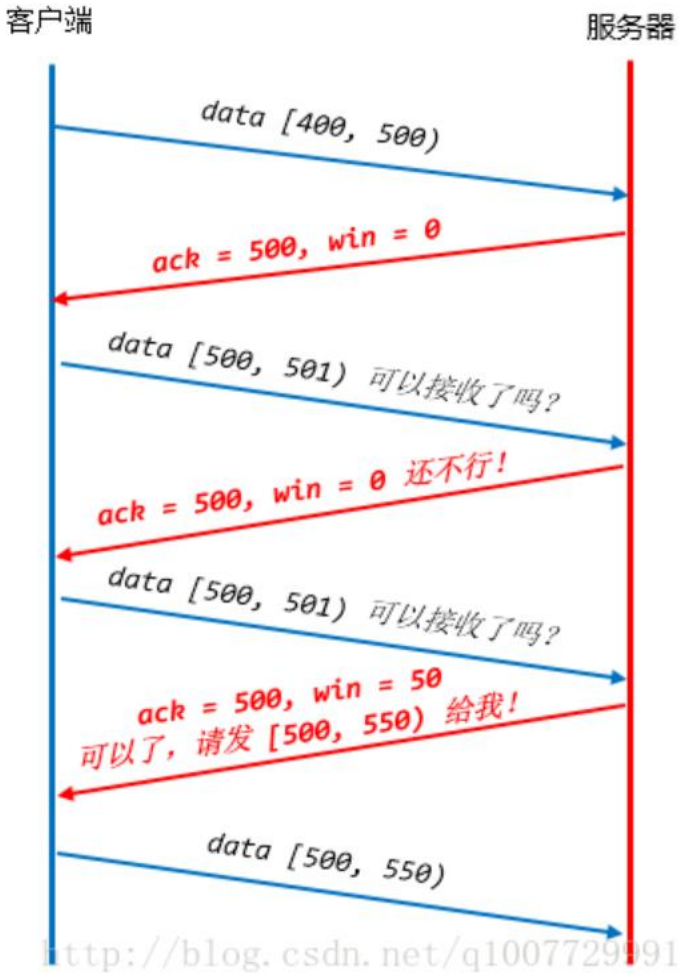


图3 对方回送 0 大小的窗口，接收端的反应

在 TCP 中，**滑动窗口是为了实现流量控制**。如果对方发送数据过快，接收方就来不及接收，接收方就需要通告对方，减慢数据的发送（图 3）。

需要特别注意的是，**在学习滑动窗口的时候，我们假设网络无限好，不拥塞**。只要你发送了数据，对方一定可以收到。
网络拥塞的含义，它是指你发送的数据滞留在网络中，迟迟未到达接收方。

2.4 PSH 标志

在谈 PSH 标志位前，先来说说 TCP 双方是如何发送数据的。
为了能讲清 PSH 的作用，**不妨假设我们可以自己指定 PSH 标志**。

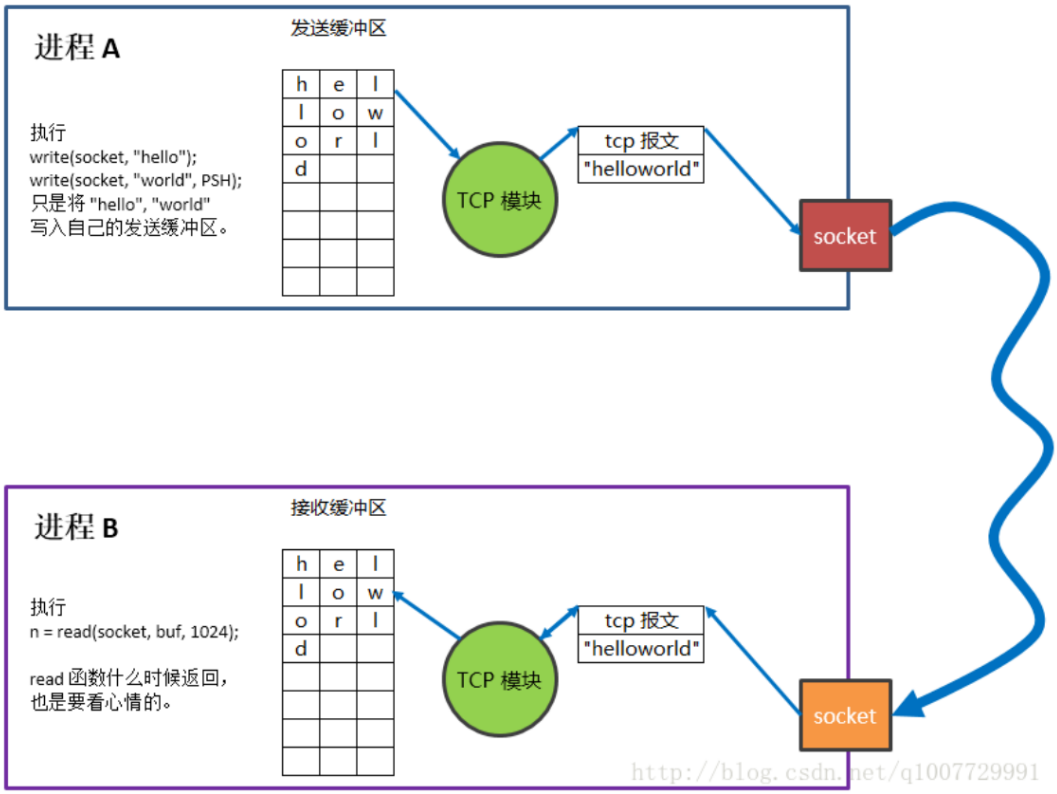


图1 TCP 协议中的发送缓冲区与接收缓冲区

PSH 的作用：

TCP 模块什么时候将数据发送出去（从发送缓冲区中取数据），以及 read 函数什么时候将数据从接收缓冲区读取都是未知的。

如果使用 PSH 标志，上面这件事就确认下来了：

- 发送端

对于发送方来说，**由 TCP 模块自行决定**，何时将接收缓冲区中的数据打包成 TCP 报文，并加上 PSH 标志（在图 1 中，为了演示，我们假设人为的干涉了 PSH 标志位）。一般来说，每一次 write，都会将这一次的数据打包成一个或多个 TCP 报文段（如果数据量大于 MSS 的话，就会被打包成多个 TCP 段），并将最后一个 TCP 报文段标记为 PSH。

当然上面说的只是一般的情况，如果发送缓冲区满了，TCP 同样会将发送缓冲区中的所有数据打包发送。

- 接收端

如果接收方接收到了某个 TCP 报文段包含了 PSH 标志，则立即将缓冲区中的所有数据推送给应用进程（read 函数返回）。

当然有时候接收缓冲区满了，也会推送。

2.5 紧急数据

16 位紧急指针，它正是配合 URG 标志位一起使用的，言外之意就是这个字段只有在 URG 被置位时才有意义。因为只有一个紧急指针，这也意味着它只能标识一个字节的数据。这个指针指向了紧急数据最后一个字节的下一个字节。

紧急标志的作用：
紧急标志可以用来通知对端：**我放了一个紧急数据在数据流中。**这个特性往往可以来达到通知的目的。

一旦 TCP 知道了你要发送紧急数据，那么在接下来的数据发送中，TCP 会将所有的 TCP 报文段中的 URG 标志置位，哪怕该报文段中不包含紧急数据，这个行为会持续到紧急数据被发送出去为止。

很多系统的实现，包括 Linux 将紧急数据称之为**带外数据**（out-of-band data, OOB），意为在连接之外传送的数据，实际上这是不对的。即使是紧急数据，仍然会随着普通数据流一起发送，并不会单独为紧急数据开辟一条新的连接通道单独发送。这从图 3 中我们也可以看到，紧急数据并没有被优先发送出去。

2.6 拥塞控制

在学习**拥塞控制**前，我们假设：**接收方总是有足够大的缓存空间，接收方的接收窗口大小总是很大——**这意味着接收方对数据来者不拒。

在基于这样的理想条件上，如果发送方发送的数据接收方没有收到，那么大抵上可以判断为网络出现了拥塞。

为什么会出现网络拥塞？

一般是因为交换机或者路由器处理不过来发送方发送的数据。因为交换机与路由器是基于“存储+转发”的方式来将发送方的数据进行转发的。如果发送方发送数据的速度过快，导致路由器或者交换机的换从空间不够，就会出现网络拥塞。

如何避免拥塞？

四种**拥塞控制**算法：

- 慢启动（slow-start）
- 拥塞避免（congestiono avoidance）
- 快重传（fast retransmit）
- 快恢复（fast recovery）

2.61 慢启动与拥塞避免

慢启动算法：

对于发送方来说，TCP 维护了一个变量 cwnd (congestion window)，这个变量称为拥塞窗口，它的大小就是 cwnd。它表示发送方一次想要发送多少字节的数据。

cwnd 是一个动态变化的值，它会根据网络的情况实时的调整自己，最后，cwnd 就会适应网络的情况，以保证发送 cwnd 字节的数据而网络不会拥塞。最终我们关注的问题就是 cwnd 是如何调整的问题。

慢启动算法是这样的：一开始发送数据的时候，如果把大量的数据注入到网络，就有可能引起网络拥塞，因为一开始并不知道网络的状况。因此，慢启动算法一开始先探测一下，即先发送大小为 cwnd = 1 的报文（一个报文）到网络中，当收到了对该报文的确认后，就把 cwnd 的值加 1，此时 cwnd 的值就变成 2 了。接下来，再发送大小为 cwnd = 2 的报文（两个报文）到网络中，发送方每收到一个确认（不能计算重传报文的 ack，另外我们把累积确认计算为多个确认），cwnd 的值就翻倍.

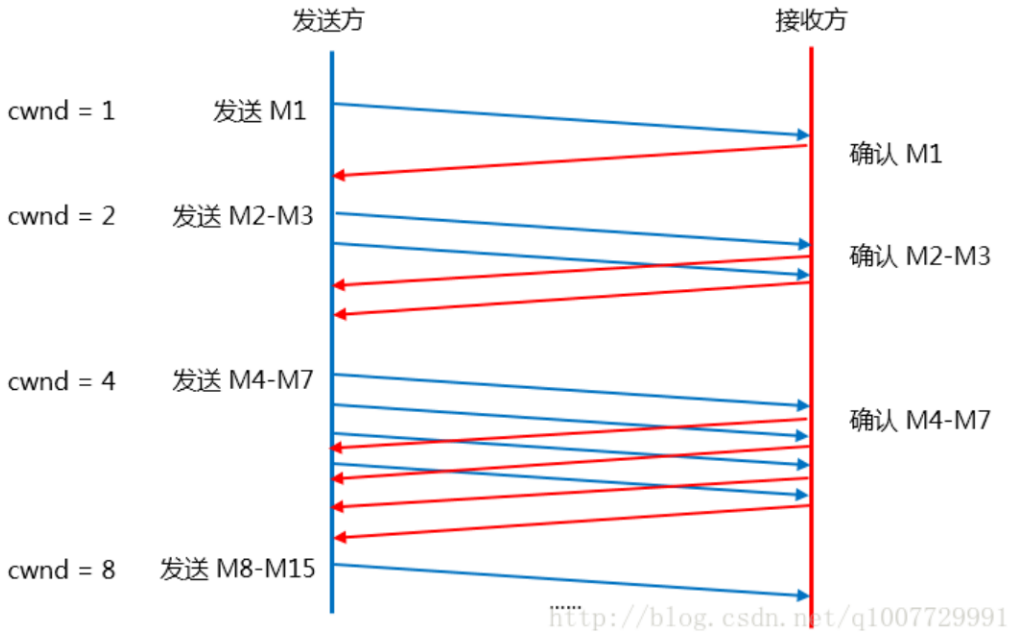


图1 慢启动算法

TCP 中还维护了另一个变量 ssthresh，它称之为**慢启动门限**，这是一个阈值，当 cwnd 超过这个值的时候，慢启动算法结束，进入拥塞避免算法！

ssthresh 变量在一开始也是有默认值的，比如 ssthresh = 16. 接下来，我们讨论 cwnd > ssthresh 后，TCP 的行为。

拥塞避免算法：

刚刚上面说了，当 cwnd > ssthresh 时，转而执行拥塞避免算法。这时候，TCP 发送 cwnd 个报文后，如果接收到了确认，**cwnd 的值只是加 1，而不是加倍**。这样，拥塞窗口 cwnd 就会按线性规律缓慢增长。

无论是在慢启动阶段，还是在拥塞避免阶段，只要发送方判断网络出现拥塞（依据就是没有按照收到确认），就要把 ssthresh 设置为出现拥塞时的 cwnd 值的一半（注意这只是一种策略，实际实现中不一定是这样的，RFC 中给出的公式是将已发出但是还未被确认的数据字节数来设置 ssthresh 的值）。

在更新了 ssthresh 后，同时将 cwnd 重新设置为 1，又开始执行慢启动算法。这样做的目的是要迅速减少主机发送到网络中的分组数，使得发生拥塞的中间设备有足够的时间把缓冲区中积压的分组处理完毕。

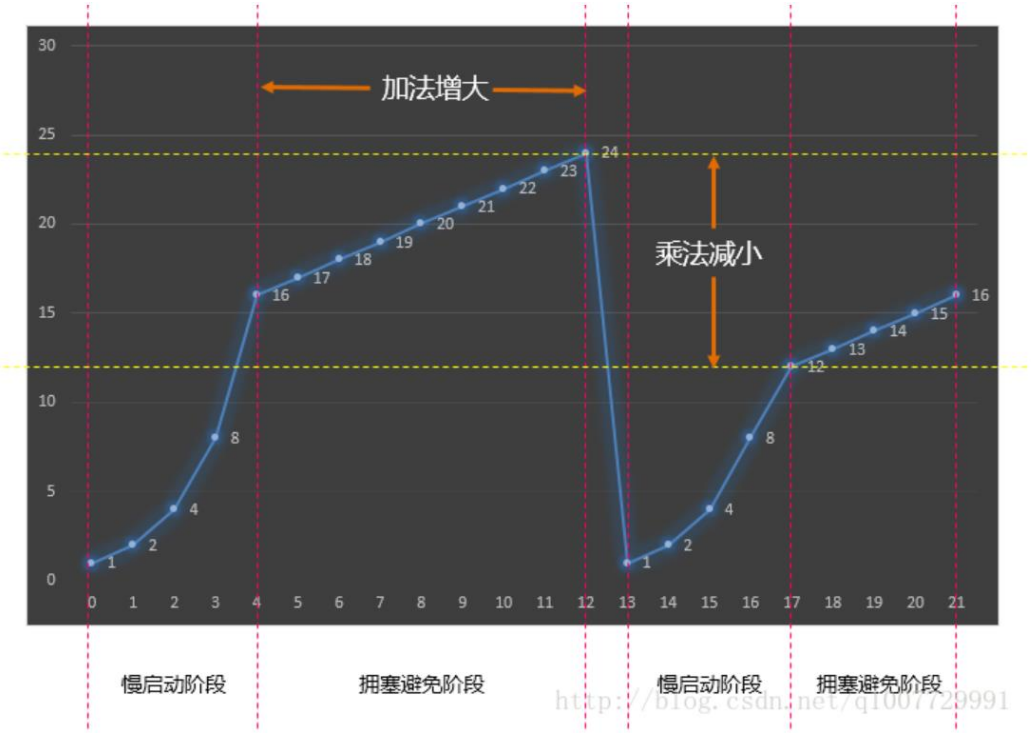


图2 慢启动和拥塞避免算法

加法增大是指执行拥塞避免算法后，使拥塞窗口 cwnd 缓慢增大，以防止网络过早出现拥塞。而乘法减小，是指不论在慢开始阶段还是在拥塞避免阶段，只要出现超时（即很可能出现了网络拥塞），就把 ssthresh 减半，即 ssthresh = cwnd/2，紧接着，cwnd = 1.

2.62 快重传与快恢复

在此之前，先来回忆一下，发送方如何判定网络产生拥塞？已知的一种情况是对方回复 ack 超时。其实还有一种情况，如果发送方连续收到接收方多个重复的 ack（接收方不会没事发送重复的 ack 的），则说明网络生产拥塞

那么简单总结一下，发送方判定网络拥塞就有两种情况了：

- ack 超时
- 发送方连续收到接收方发过来的重复的 ack

快重传：

首先对于接收方来说，如果接收方收到一个**失序**的报文段，就**立即**回送一个 ACK 给发送方，而不是等待发送**延时的 ACK**。所谓失序的报文是指，用户没有按照顺序收到 TCP 报文段，比如接收方收到了报文 M1, M2, M4，那么 M4 就称为**失序** 报文。

这样做的目的是可以让发送方尽可能早的知道报文段 M3 未到达接收方。快重传算法规定，如果发送方一连收到 3 个重复的确认，就应当立即传送对方未收到的报文 M3，而不必等待 M3 的重传计时器到期。

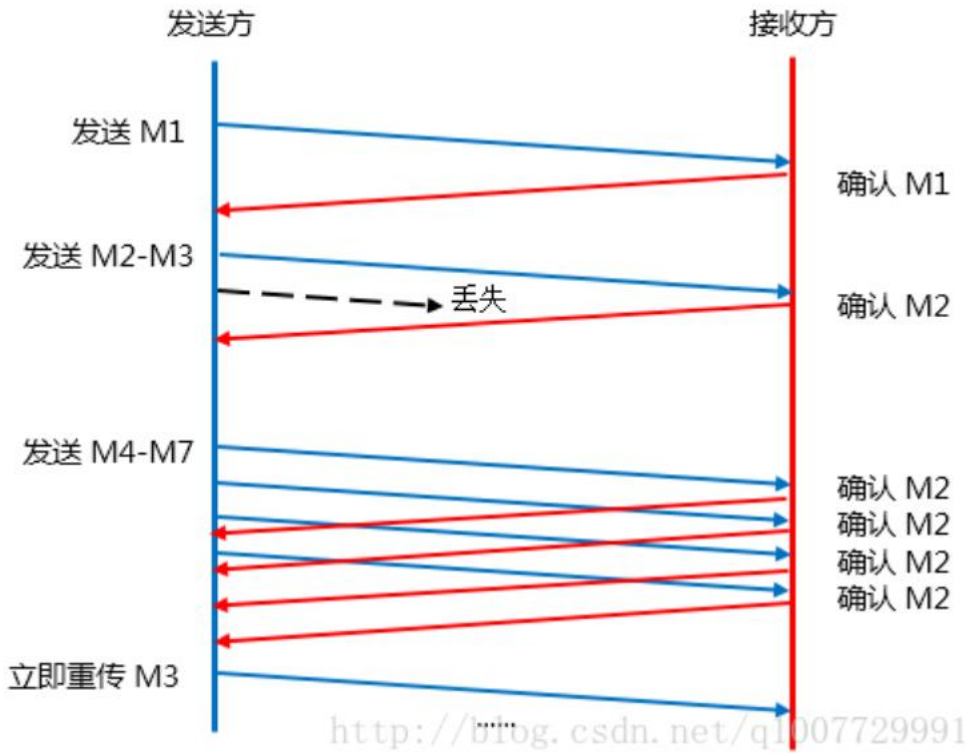


图1 快重传算法

快恢复：

在学习了上一节的慢启动和拥塞避免算法后，我们知道，一旦出现超时重传，TCP 就会把慢启动门限 ssthresh 的值设置为 cwnd 值的一半，同时 cwnd 设置成 1. 但是快恢复算法不这样做。

一旦出现超时重传，或者收到第三个重复的 ack 时（快重传），TCP 会把慢启动门限 ssthresh 的值设置为 cwnd 值的一半，同时 cwnd = ssthresh（在有些版本中，会让 cwnd = ssthresh + 3）。

之前的旧版本的算法是在 TCP 的 Tahoe 版本中，而改进的版本算法是在 TCP Reno 版本中。

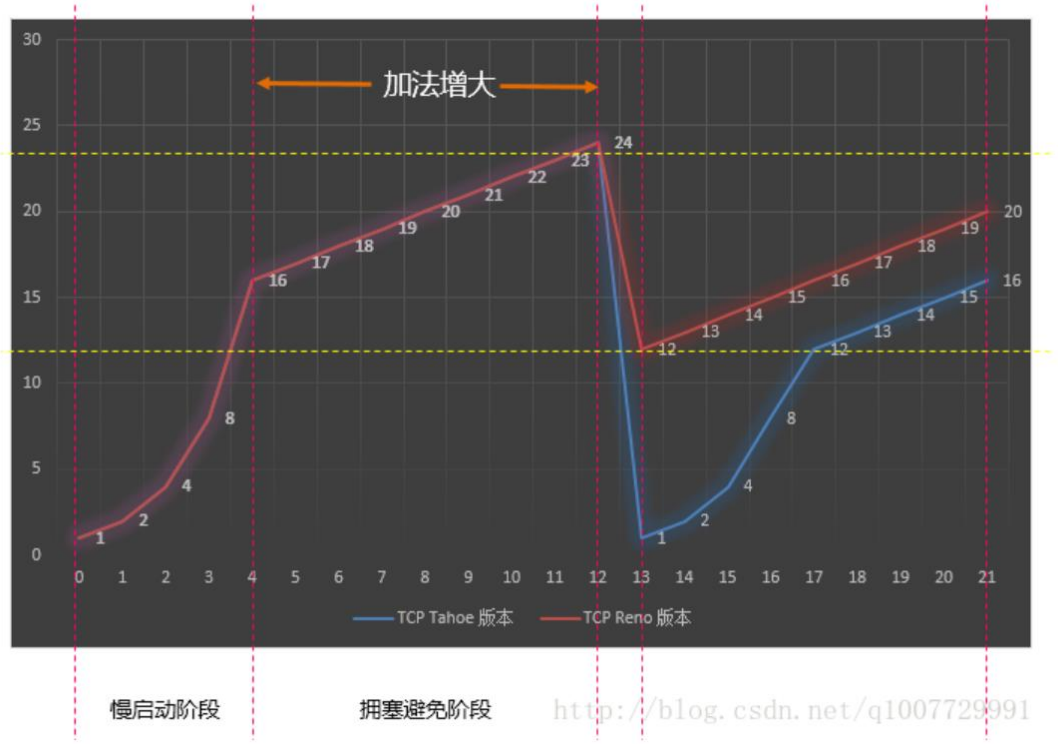


图2 TCP Reno 版本与 Tahoe 版本

图 2 中，连续收到三个重复确认后，TCP Reno（红色）版本转入了拥塞阶段，而 TCP Tahoe 版本（蓝色）转入了慢启动阶段。

2.621 流量控制与拥塞控制

在学习流量控制的时候，我们假设网络无限好，不拥塞。在学习拥塞控制的时候，我们又假设接收方缓冲区和接收窗口无限大，对数据来者不拒。现在，是时候综合考虑他们的时候了。

我们只要将接收方的窗口 rwnd 和拥塞窗口 cwnd 放在一起比较，取两者中的较小者，也就是：

$$\text{发送方的窗口上限值}=\min\{rwnd,cwnd\}$$

上式指出：

- rwnd < cwnd ：是接收方的接收能力限制了发送方窗口的最大值。
- cwnd < rwnd ：是网络的拥塞限制了发送方窗口的最大值。

2.7 超时重传

超时重传是 TCP 保证可靠传输的基础。当 TCP 在发送数据时，数据和 ack 都有可能会丢失，因此，TCP 通过在发送时设置一个超时重传定时器来解决这种问题。如果定时器溢出还没有收到确认，它就重传数据。

无论是 Windows 还是 Linux，关键之处就在于超时和重传的策略，需要考虑两方面：

- 超时时间设置
- 重传的频率（次数）

2.8 持续定时器

在我们学习滑动窗口协议的时，遇到过窗口大小为 0 的情况。接收方通告一个 0 窗口给发送方，可以用来阻止发送方继续发送数据。
如果在某个时候，接收方缓冲区有空间了，于是发送了一个非 0 窗口的通告给发送方，不幸的是这个通告丢失了，而发送方却还在死等接收方的非 0 窗口通告，接下来就成了死锁。

为了防止这种情况，发送方只要接收到了 0 窗口通告，就开启一个**持续定时器**（persist timer），周期性的向接收方发送 1 字节的 0 窗口探测报文。

2.9 糊涂窗口综合征

糊涂窗口综合症，原文叫 Silly Window Syndrome，简称 SWS.

SWS 是这样一种情况：接收方通告了一个 1 字节的窗口给发送方，然后发送方发送了 1 字节的数据给接收方。接着，接收方又通告了一个 1 字节的窗口，这样持续下去，使网络的效率很低。

如何避免 SWS？
有两种办法可以解决此问题：

- 针对接收方来说，先等一段时间，等到接收缓存有足够的空间了才发出确认。
- 针对发送方来说，发送方不要发送太小的报文，而是把数据积累成一个足够大的报文段（达到 MSS），或者是积累到接收方通告窗口大小一半的报文段。

如果发送方总是有很多小数据要发送怎么办呢？实际上，我们前面学习过 Nagle 算法，它正是用于解决这个问题的。Nagle 算法的目的就是把小数据组合成一个大报文段发送出去的。

如果发送方禁用了 Nagle 算法，就不应该发送小数据，否则很容易引起 SWS 问题。

2.10 保活定时器

TCP 中的 Keepalive 是 **TCP 本身实现的，和应用层无关**。它的目的在于**看看对方有没有发生异常**。

在 TCP 层面，如果双方没有任何数据交互，这个连接会永远存在，这意味着如果客户与服务器建立连接，可以在数小时，数天甚至数月之后连接依然保持。中间设备（比如路由器）可以崩溃，可以重启，网络可以断，只要两端的主机不重启，都没有关系。

很多时候，一个连接经过了很久都没有数据交互，服务器想知道对方是不是已经崩溃，或者又重新启动了，怎么办呢？TCP 的 Keep Alive，保活定时器就是做这事的。

它每隔一段时间会超时，超时后会检查连接是否空闲太久了，如果空闲的时间超过了设置时间，就会发送探测报文。然后通过对端是否响应、响应是否符合预期，来判断对端是否正常，如果不正常，就主动关闭连接。

大多数时候，我们写服务器程序的时候，会在应用层实现一个 Keep Alive，而不使用 TCP 提供的，比如常见的 HTTP 协议中就实现了这个功能，还有 SSH 协议也实现了自己的 keep alive 功能。

3 网络编程之简单的回射服务器

3.1 异常处理（accept 返回前连接终止）

一种比较特殊的情况，即客户端连接建立成功后（进入 **ESTABLISHED** 状态），立即关闭连接退出。而此时服务器中的 accept 函数还没调用或者还没有返回。

结果分析：accept 成功返回，但是 readline 出错

实际上，这是由操作系统实现来决定的，有些操作系统可能在 accept 时，悄无声息的把这个连接给 kill 掉，有些可能会让 accept 返回 ECONNABORTED。man 手册上给出的解释是：不同的 Linux 内核也**可能**会返回 ECONNABORTED。

不过，服务器在接收到 RST 后，在 readline 时做出了响应，它返回了一个 ECONNRESET 错误。

3.2 并发服务器（多进程）

如果最后客户端退出，那么对应的服务器端的子进程退出，但是此时父进程并没有退出，导致了僵尸进程，如何处理这僵尸进程：

处理僵尸进程：

方案 1：忽略 SIGCHILD 信号。

```
signal(SIGCHLD, SIG_IGN);
```

方案 2：捕捉 SIGCHILD 信号，并 wait。

信号打断低速系统调用：

这个是一个坑，之前我们也有讲过。这里我简单总结一下：

进程捕捉到信号后，会打断某些正在阻塞中的函数（低速系统调用，比如 read，accept，connect 等），这种函数如果被信号打断，会直接返回错误，同时设置 errno = EINTR.

实际上，这并不是错误，如果我们没有处理这种情况，让程序直接退出，会让一个运行的很好的服务器停止。所以为了让服务器或客户端更加健壮，我们需要额外的处理这种错误，比如：

```
// accept 返回错误，可能是被信号打断。如果被打断，不认为它是错误。

ret = accept(listenfd, ...);

if (ret < 0) {

  if (errno == EINTR) contine;

  else exit(1);

}
```

多进程并发服务器因为受系统进程数量的限制，并发量也就在几千左右。

如果只是连接，不处理 IO，那么单机实现连接的数量可以达到几万以上（根据服务器配置）

```
while(1) {
  sockfd = accept(listenfd);
```

```
}
```

注意：服务器不会有端口耗尽的情况，因为本机套接字地址始终都是固定的，变化的只是远端套接字地址，tcp 连接是一个四元组 {local ip:port <-> foreigh ip:port}。

3.3 连接断开异常

当一个进程向某个已收到 RST 段的套接字执行写操作时，内核向该进程发送一个 SIGPIPE 信号。

服务器崩溃，言外之意就是说服务器挂了，但是没有发送 FIN 报文给客户端。

对于服务器崩溃的情况，一般 read 会返回

- EHOSTUNREACH 错误（网络存在，但是主机不存在）
- ENETUNREACH（网络不存在，可能是中间路由器宕机了）
- ETIMEDOUT 主机存在，但是主机根本不响应。

虽然服务器崩溃后重启，但是客户端对此毫不知情，客户端此时却仍然持有着一个，若继续向对端发送数据，服务器主机可以收到此报文，但是并不认识这个连接（因为 flower 已经重启，之前的连接早已不存在），因此回复了一个 RST 段。

4 IO 多路复用

4.1 使用 select 改进客户端

回忆:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
```

```
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

（1）参数

- nfds 表示三个集合中最大描述符的值 + 1
- 后面三个 fd_set 集合分别表示监听哪种类型的事件，分别表示读事件，写事件和异常事件集合
- 最后一个参数是超时参数，可以为 NULL，表示永远等待

（2）返回值

- 小于 0，失败
- 等于 0，超时时间到
- 大于 0，发生了 IO 事件的个数

改进客户端：
旧版本的客户端：

```
void doClient(int sockfd) {
  // 客户端阻塞在标准输入上，一旦客户端收到了服务器的 FIN，也无能为力
  while(fgets(buf, stdin)) {
    write(sockfd, buf);
    read(sockfd, buf);
    puts(buf);
  }
}
```

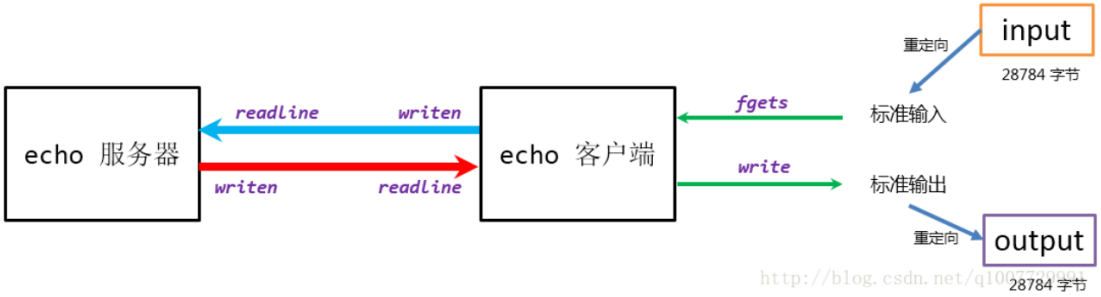
缺点是它无法感知服务器进程发来的 FIN 段，现在改进如下（伪代码）：

```
void doClient(int sockfd) {
  fds.add(STDIN_FILENO);
  fds.add(sockfd);
  maxfd = max(STDIN_FILENO, sockfd);

  while(1) {
    rfds = fds;
    // 只监听了标准输入和套接字 sockfd
    nready = select(maxfd + 1, &rfds, NULL, NULL, NULL);
    if (STDIN_FILENO 在 rfds 中为 1) {
      if (fgets(buf, stdin) != NULL) {
        write(sockfd, buf);
      }
      else {
        break;
```

```
    }
  }

  if (sockfd 在 rfds 中为 1) {
    if (read(sockfd, buf) == 0) {
      puts("peer closed");
      break;
    }
    puts(buf);
  }
 }
}
```

4.2 半关闭（shutdown）的引入（批量输入导致的异常）

当我们启动客户端的时候，将客户端的标准输入重定向到某一个文件中，另一方面，我们将客户端的标准输出重定向到 output 文件。



多次运行服务器与客户端后，会发现结果出现很多错误：
原因：要知道，cpu 的执行的速度是相当快的，可能在一瞬间，就把 input 文件中所有数据 fgets 完并发送给对端，甚至**在对方还没来得及收到数据的情况下，客户端已经 close 退出了。**
经过一段时间后。服务器收到了客户端发来的数据，然后处理完成（转换为大写），接着就立即发送给客户端，不幸的是服务器此时并不知道客户端进程已经关闭了套接字退出了。

如何改进：

别忘记了，TCP 断开连接的 4 次挥手是怎样的。即使其中一方发送了 FIN 段关闭了连接，另一方也可以继续传送数据，这就是所谓的半关闭。

然而 close 函数并不能实现半关闭，一旦 close，两个方向的数据传送通道全部被关闭，所以，我们只能使用 shutdown 函数。

4.21 shutdown 函数

终止网络连接一般使用 close，但是 close 一般有两个限制：

- close 把描述符引用计数减 1，只有该计数为 0 时才关闭套接字。而 shutdown 不管引用计数，只要调用，直接发送 FIN 报文。
- close 终止读和写两个方向的数据传送。

```
int shutdown(int sockfd, int howto);
```

该函数的行为依赖于 howto 的值：

- SHUT_RD: 关闭连接读这一半，而且接收缓冲区中现有数据全部丢弃。进程不能再对这样的套接字调用任何读函数。当然，对端对此毫不知情，仍然可以发送数据过来，关闭读半部的 TCP 收到对端发送的数据段进行确认后（ack，这并不是客户端发送的，确认报文时内核发送的，所以不会引起 TCP 报文段的传输），再丢弃掉，而不是放入接收缓冲区。**使用该选项，并不会引起任何 TCP 报文的传输。**
- SHUT_WR: 关闭连接写这一半。这就是**半关闭**。该行为首先将发送缓冲区中所有数据发送出去，最后引发 TCP 向对端发送 FIN 报文段。进程不能再对这样的套接字执行任何写操作。
- SHUT_RDWR: 连接的读写全部关闭，这相当于调用 `shutdown(sockfd, SHUT_RD); shutdown(sockfd, SHUT_WR)`.

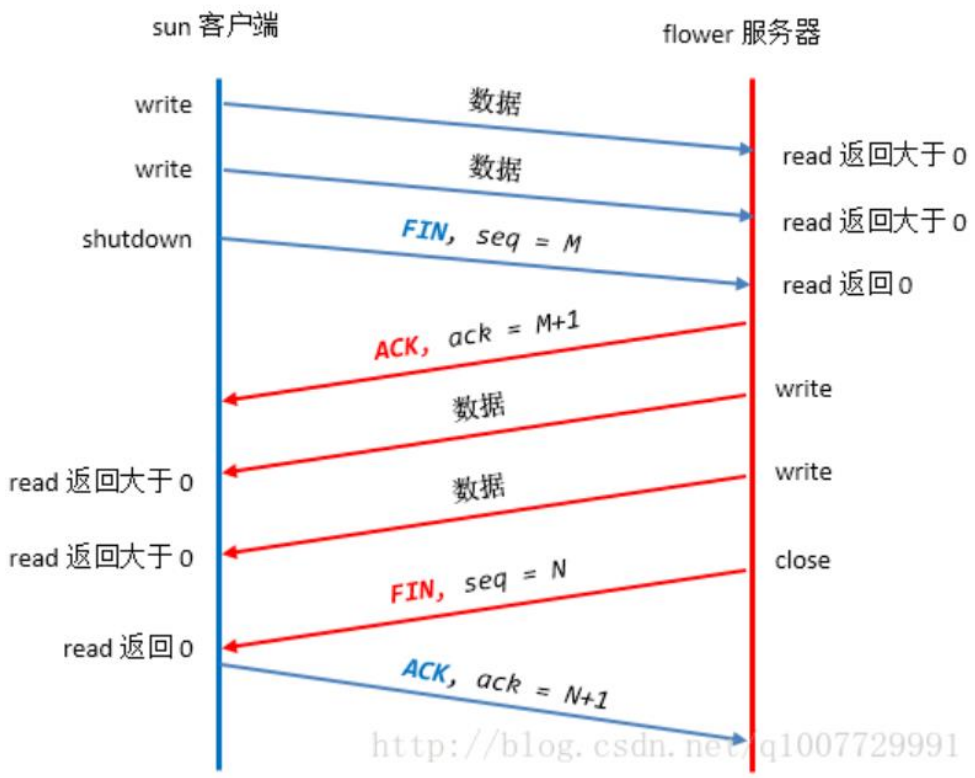


图2 使用 shutdown 执行半关闭

当客户端关闭写通道时，服务器仍然在向客户端发送数据，而客户端也要做好接收对端数据的准备，不能随意就退出程序。

4.3 IO 复用与带缓冲的 IO 函数

不要将 stdio 库提供的 C 语言函数与 IO 复用混合使用!!!

select 函数返回的条件是有 IO 事件发生。假设我们监听了 STDIN_FILENO（标准输入），某个时刻 select 监听到了标准输入上有数据，于是客户端调用 fgets 读取一行数据。

看起来一切是那么自然，不过别忘记了，stdio 的函数是带有缓冲区的。别以为 fgets 真的只读了一行数据，它只是将自己缓冲区中的一行数据返回给了你。实际上，fgets 早已把 STDIN_FILENO 上的所有数据读完了，保存到了自己的缓冲中了，

接下来，你的程序又回到了 select 调用上，很不幸，select 永远再也不会触发标准输入上的 IO 事件，因为 select 认为，你的数据早已被读取完（虽然这不是你干的），它并不知道 stdio 使用了缓冲区——它只是从 **read** 系统调用的角度指出是否有数据可读，而不是从 fgets 之类调用的角度考虑。

我们要小心任何带有缓冲的 IO 与 IO 复用的混合使用。

4.4 多进程并发服务器的改进

4.41 用 select 改进

我们知道服务器主要做两件事情：

- 接收新的连接请求，主要由**监听套接字**来完成
- 和已连接的客户端交互数据，主要由**已连接套接字**来完成（accept 函数返回的那个值）

因此，服务器要使用 select 同时管理**监听套接**和**已连接套接字**。

我们需要事先准备一个 fd_set 类型的读集合 rfds，该集合保存了所有的**监听套接**和**已连接套接字**。另一方面，当 IO 事件产生时，我们需要挨个遍历 rfds 中的每个描述符，但是 rfds 并不像 C++ 中的 set 集合那么方便，它本身不提供任何一种方法来帮助我们遍历所有元素。因此，我们需要单独再使用一个数组 fds[1024] 来保存所有套接字。

4.42 DOS 攻击

拒绝服务型攻击（Denial-of-Service Attacks），简称 dos 攻击。它针对服务器漏洞，使用特殊手段消耗服务器资源，从而达到服务器无法正常为其它客户端提供正常服务的目的。

Attack 客户端(攻击型客户端)代码:

```
void doClient(int sockfd) {
  int ret;
  char c = 'A';
  ret = iwrite(sockfd, &c, 1);
  if (ret < 0) {
    ERR_EXIT("iwrite");
  }
  while(1) {
    sleep(100);
  }
}
```

它只向我们的服务器发送了一个字符 `'A'`，仅仅如此，就能让服务器陷于阻塞而不能自拔。

因为服务器是使用 select 改写的，它和所有客户端的交互都在一个线程中，当它与 attack 客户端交互时，使用 readline 函数从套接字中读取数据，然而，readline 函数在读取到 `'`

\n'字符前或者 EOF 字符前是不会返回的，因此服务器就阻塞在了 read 函数上，什么时候 attack 关闭了，或者发送了一个 '\n'字符，readline 才会返回。阻塞在 readline 函数的服务器，此时没有任何额外的线程来接收新的客户连接，也拒绝为其它任何已连接的客户端提供服务。

解决方案？：

- 使用非阻塞式 IO
- 采用多进程或多线程为客户端服务（就是我们之前写的多进程并发服务器）
- 对 IO 操作设置超时

4.43 用 poll 改进

回忆 poll 函数：

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
  int   fd;         /* file descriptor */
  short events;     /* requested events */
  short revents;    /* returned events */
};
```

Poll 与 select：

这两个函数所做的事件是一样的，但是它们也有区别：

- select 使用 fd_set 来存放描述符，poll 使用结构体数组。
- select 能够一次监听的描述符数量是受 fd_set 集合的限制的，通常这个集合最多只能放 1024 个描述符。而 poll 一次能够监听的描述符个数是根据数组大小来决定的，这要看 nfds_t 被定义成什么类型了，如果是 unsigned long，4 字节宽度的情况下，poll 能监听 $2^{32}-1$ 个描述符。

4.44 用 epoll 改进

参考代码： APUE 中关于 epoll 的说明

```
#include <sys/epoll.h>

int epoll_create(int size);
```

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int
timeout);

void server_routine() {
  // ...
  int epfd;
  struct epoll_event evts[5]; // 接收 epoll_wait 的返回值
  struct epoll_event ev;
  listen(listenfd);

  epfd = epoll_create(1); // 参数只要大于 0 就行了

  // 添加 listenfd 到 epoll
  ev.data.fd = listenfd;
  ev.events = EPOLLIN;
  epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev);

  while(1) {
    // 监听事件
    nready = epoll_wait(epfd, evts, 5,  0);
    for (i = 0; i < nready, ++i) {
      // 查看是否有新的连接
      if (evts[i].data.fd == listenfd && (evts[i].events & EPOLLIN)) {
        sockfd = accept(listenfd);
        ev.data.fd = sockfd;
        ev.events = EPOLLIN;
        // epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &ev);
        // 此行原来有误，应该是添加 sockfd 而非 listenfd.感谢网友 Neymar 指
出，修改如下:
        epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &evt);
      }
      // 与客户端交互数据
      else if (evts[i].events & EPOLLIN) {
        ret = doServer(evts[i].data.fd);
        if (ret <= 0) {
          // 客户端关闭，则将其从 epoll 中删除
          epoll_ctl(epfd, EPOLL_CTL_DEL, evts[i].data.fd, NULL);
          close(evts[i].data.fd);
        }
      }
    }
  }
}
```

5 套接字选项

5.1 套接字选项概述

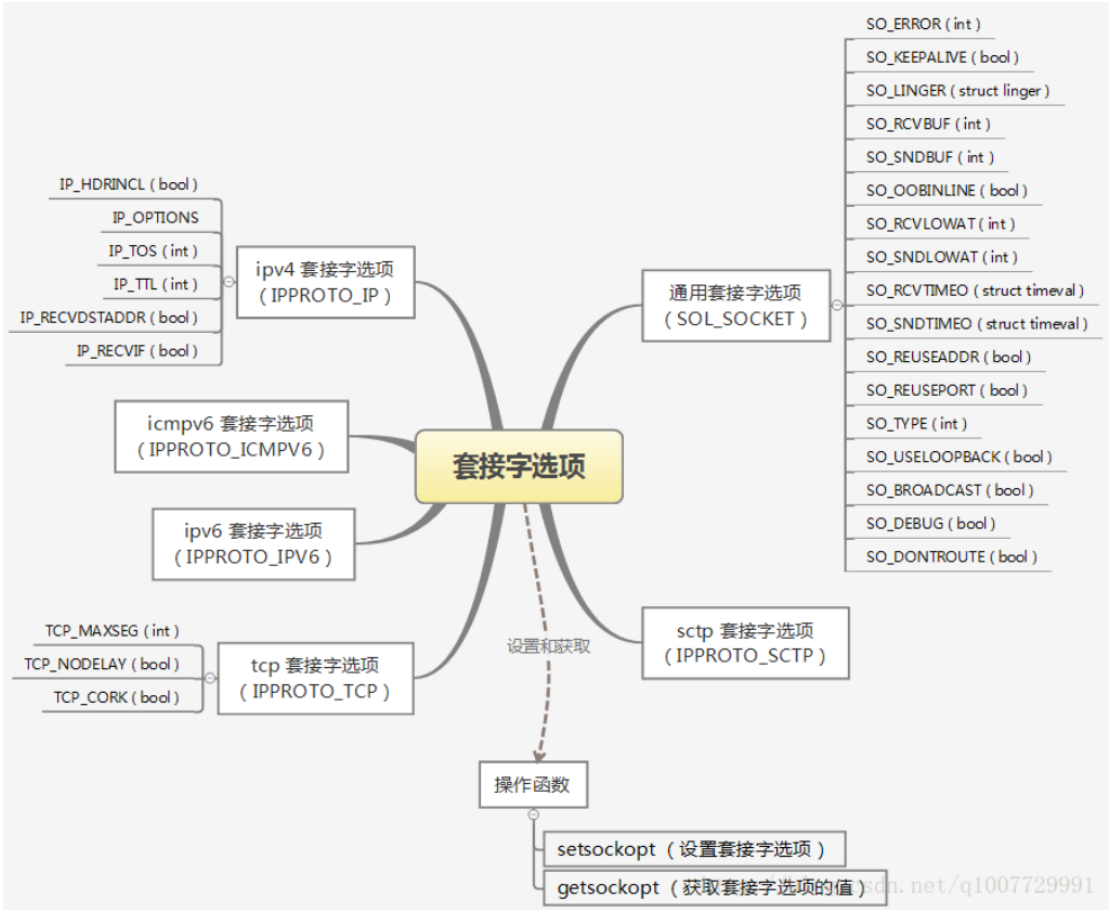
相信你学过前面的 TCP 协议也知道，里头的算法非常多，有些算法你可以开启，有些你可以关闭。比如你可以设置发送和接收缓冲区大小，也可以设置是否打开 Nagle 算法等等。这些是如何做到的呢？

有很多方法可以用来获取和设置影响套接字的选项：

- getsockopt 和 setsockopt 函数
- fcntl 函数
- ioctl 函数

大多数情况，我们都使用 getsockopt 和 setsockopt 函数。

getsockopt 和 setsockopt 函数是根据套接字的类别对其进行设置的，比如有些套接字选项和协议无关，这种就称为通用套接字选项。有些套接字选项和 TCP 协议相关，那么这种套接字选项就称为 TCP 套接字选项，比如用来打开和关闭 Nagle 算法的选项 TCP_NODELAY。



5.2 setsockopt 和 getsockopt

```
int getsockopt(int sockfd, int level, int optname,
                    void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
                    const void *optval, socklen_t optlen);
```

参数 level 表示套接字选项的级别（类别），比如你如果要设置通用套接字选项，那么 level 就应该设置为 SOL_SOCKET，如果你要设置 TCP 套接字选项，level 就是 IPPROTO_TCP

参数 optname 就是具体的选项名称了，比如你要设置 SO_REUSEADDR，你可以这样：

```
int reuse = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
```

5.3 套接字选项：SO_REUSEADDR

SO_REUSEADDR 是非常常用的选项之一，它有四个功能

注意，一定要在 bind 监听套接字前设置此选项，否则会因为时序问题导致没有效果。

- 1) 允许绑定处于 TIME_WAIT 状态的套接字地址
- 2) 允许在同一个端口上启动同一个服务器的多个实例，只要每个实例绑定的 ip 地址不同即可。

如果捆绑相同的 ip 地址和端口号，即使指定 SO_REUSEADDR 也没有用。我们将这种绑定相同 ip 地址和端口号称为 completely duplicate binding，即完全重复的捆绑，这在 TCP 协议中是不允许的。

- 3) 允许单一进程捆绑同一端口到多个套接字上，只要每次指定的 ip 地址不同即可。
- 4) 针对 UDP 套接字，允许**完全重复的捆绑**（completely duplicate binding），即 ip 地址和端口号都重复。

对于这种情况，假设同一主机运行同一个应用程序的多个副本，当一个 UDP 数据报需要由这些重复捆绑套接字中的一个接收时，规则为：如果该数据报的目的地址是一个广播或多播地址，那就给每个匹配的套接字递送一个该数据报的副本；但是如果该数据报的目的地址是单播地址，那么它只递送给单个套接字。在单播数据报的情况下，如果有多个套接字匹配该数据报，那么该选择由哪个套接字接收它取决于系统实现。

UNP 推荐，所有的 TCP 服务器都应该指定这个套接字选项。所以我们编写服务器时，应当在一开始就把这个选项打开。

5.4 套接字选项：TCP_NODELAY

开启本选项，将禁止 Nagle 算法。有关 Nagle 算法的细节，请参考本文 2.2 Nagle 算法以及博客《TCP 协议（Nagle）》。

5.41 回忆 Nagle

Nagle 算法的目的是为了减少网络中小分组的数目。“小”分组(tiny gram) 指是是小于 MSS 的分组。它利用收到 ack 前这一小段时间“攒”数据，收到 ack 后，再将“攒”起来的小分组合并成一个大分组一次性发出去。
在局域网上，通常我们察觉不到 Nagle 算法对客户进程的影响。但是在广域网上，小分组所需要的确认时间可能长达 1 秒，这样客户端就会有非常明显的延时。如果受延时的 ack 影响，这种延时会更加明显（参考本文：2.1 延迟 ACK 或博客 TCP 协议（迟到的 ACK —— Windows ）和 TCP 协议（迟到的 ACK—— Linux））。

5.42 Nagle 算法与 write-write-read

```
// client

while(1) {
  write(sockfd, buf, 4);
  write(sockfd, buf, 396);
  read(sockfd, buf, 1024);
}
```

第 2 次 write 的数据会一直驻留在发送缓冲区中，直到第一个 4 字节的小分组被确认。而服务器并没有数据要发送给客户端，因此会发送延时的 ack，比如已经过了 40ms，客户端才会收到这 4 字节的小分组的确认。接下来 TCP 发送 396 字节大小的分组，等待服务器处理完后发送数据给客户端，客户端从 read 返回。

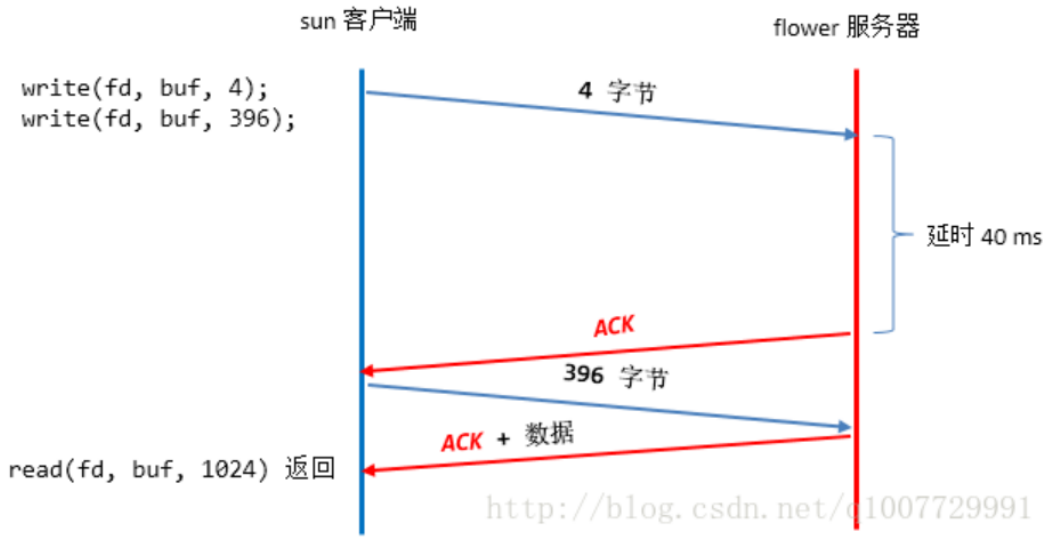


图1 write-write-read

flower 服务器之所以有 40ms 延时发送 ack，这是因为受到了**延时的 ack** 算法影响。该知识点在介绍 tcp 的时候已经详细的讲解过了。

在这种情况下，Nagle 算法并没有什么好处，反而会降低数据传输效率。有三种方法可以修正这种问题：

- 使用 writev 函数，聚集写。
- 在应用层将前 4 个字节和后 396 字节复制到单个缓冲区，然后调用一次 write.
- 使用 TCP_NODELAY 关闭 Nagle 算法。unp 中提到说这是最不可取的方法，而且有损于网络，通常不应该考虑。（很多博客和文章都推荐关闭 Nagle？Nginx 默认也是关闭了 Nagle 算法的。陈硕老师也推荐关闭 Nagle 算法……所以，大家以后该怎么做？）

5.43 关闭 Nagle 算法

TCP_NODELAY 选项可以用来关闭 Nagle 算法，具体代码如下：

```
int onoff = 1;

ret = setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &onoff, sizeof(onof
f));

if (ret < 0) perror("setsockopt");
```

5.4 套接字选项：TCP_CORK

开启 Nagle 算法尽量减少小分组出现在网络上，关闭 Nagle 算法使在没有收到 ack 前就一个一个将数据发送出去，而 开启 TCP_CORK 是完全避免小分组出现在网络上。
cork 这个单词，意为“塞子”，我们把 TCP 发送缓冲区想象成一个水池，而 cork 的作用就是把这个水池的出口堵住，直到下面的事情发生才会拔掉塞子，发送数据：

- 水池满了
- 达到一定的大小，比如超过 MSS 了
- 一段时间都没有水进来（比如 200ms）
- 遇到 FIN 了
- 会不会还有其它情况？

这两个选项完全是相反的作用，所以使用的时候，二者不能同时开启。

6 基本的 UDP 套接字编程

6.1 基于 UDP 协议的回射服务器

6.11 UDP 协议概述

UDP 协议比 TCP 协议要简单的多，它是一种无连接的协议。无连接，意味着各个报文没有顺序性，你可能先发了一个数据包 A，后发一个数据包 B，结果对方却先收到 B，后收到 A.

UDP 协议首部也非常简单，如图 1.

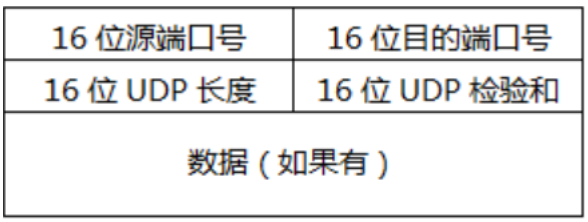


图1 UDP 首部

UDP 是不可靠的，它不像 TCP 能够保证数据一定到达对端接收缓冲区。
实际上，每个 UDP 套接字都有一个接收缓冲区，所有到达该套接字的数据报按顺序进入缓冲区。当进程调用 recvfrom 时，缓冲区中的下一个数据报以 FIFO（先进先出）顺序返回给进程。

有一些可用的解决方案，比如给 recvfrom 设置超时。如果超过一定的时间，recvfrom 还没有返回，基本上就可以认为数据报丢失，重新回到 sendto 继续执行。

还有一种方案是使用多线程，我们将发送和接收放到两个不同的线程中去执行。

还有其它更多的方案，比如给 udp 增加可靠性，模拟 tcp 协议，给 udp 程序添加两个特性：超时和和重传、序列号。

6.12 回射服务器伪代码

- 编写 UDP 程序时，并没有像 TCP 那样建立连接的过程，只要知道对方绑定的套接字地址，就可以直接将数据发送过去。
- 在创建套接字时，第二个参数需要指定 SOCK_DGRAM，第三个参数是 0.
- 无论是服务器还是客户端，都需要绑定套接字地址。对于服务器来说，需要的显式的绑定套接字地址。对于客户端，如果没有绑定，`sendto 还会默认为其分配一个 ip:port 进行绑定`

1、 recvfrom 和 sendto

- 函数原型

```
// recvfrom 接收任何发给自己的数据报
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
struct sockaddr *src_addr, socklen_t *addrlen);

// sendto 发送数据报给指定的套接字地址。如果当前 sockfd 还未绑定套接字地址，sendto 还会默认为其分配一个 ip:port 进行绑定
```

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
const struct sockaddr *dest_addr, socklen_t addrlen);
```

- 要注意的地方

  1) sendto 可以发送长度为 0 的数据报，这意味着 UDP 只会发送 8 字节的 UDP 首部，而没有数据。

  2) recvfrom 也可以返回 0，它不像 TCP，read 返回 0 表示对端发送了 FIN。UDP 是无连接的，因此也没有关闭 UDP 连接之类的事情。

6.2 UDP 客户端也可以作为服务器

- 找到客户端绑定的套接字地址

要想让客户端成为服务器很简单，只要客户端绑定了套接字地址即可。然而，客户端绑定套接字地址是隐式发生的，即在 sendto 的时候发生的，系统为随机会客户端分配一个端口号。只要我们知道了在 sendto 时的端口号，就可以给客户端发送数据了。

- 让客户端阻塞到 recvfrom 上

这种我们在上一篇文章中已经学会了，只要给一个不存在的服务器发数据，客户端就能阻塞到 recvfrom 上了。然后对客户端抓包，找打端口号。

这绝对是客户端的漏洞，那么如何验证客户端收到的数据是服务器发来的而不是其他进程发来的？答：通过验证只要将 recvfrom 接收到的 ip 地址进行比对即可。

6.3 面向连接的 UDP

前面我们已经知道，UDP 编程中的调用 sendto 和 recvfrom 产生的错误是不会返回的。主要原因在于套接字是无连接的。

要想让它们返回错误，只有让套接字变成有连接的。

我们可以在 UDP 套接字上调用 connect，但它于 TCP 连接却不同，在 UDP 套接字上调用 connect 没有三次握手的过程。本质上，connect 函数是一个重载的函数，针对不同的套接字，它会做不同的事情。

- 无连接的 UDP 套接字(unconnected UDP socket)，新创建的 UDP 套接字默认如此；
- 有连接的 UDP 套接字(connected UDP socket)，对 UDP 套接字调用 connect.

6.31 有连接 VS 无连接

1.sockname 与 peername

在网络编程中，对于**有连接**的协议来说，有两个基本的概念，即 sockname 与 peername sockname 表示本地(local)套接字地址，而 peername 表示外地(foreign)套接字地址。有两个函数 getsockname 与 getpeername 可以用来获取有连接协议的本地套接字地址与外地套接字地址。
连接，是一种抽象的概念，它相当于 sockname + peername. 因此，我们说：
有连接 = `{sockname, peername} = {(ip1:port1), (ip2:port2)}`

2.再看 bind 与 connect

很遗憾，存在 getsockname 与 getpeername 这两个函数，却不见 setsockname 与 setpeername 这样的函数

bind 函数本质上就是 setsockname，而 connect 函数本质上就是 setpeername 函数。

unp 在介绍有连接 UDP 的时候就提到过：

bind 函数更好的名字是 setsockname，而 connect 函数更好的名字是 setpeername

即有连接就是 sockname + peername，而无连接，总是缺少其中的 peername。

6.32 让 UDP 套接字成为有连接

作为 UDP 客户端来说，默认情况下它是无连接的，要想让其变成有连接，只要给它指定 peername 即可，使用 connect 函数。

注意有连接与无连接发生的三点变化：

- 1) 不能再使用 sendto 函数指定目的 IP 和 port，这个参数需要指定成 NULL，或者干脆使用 write 函数。
- 2) 不需要再使用 recvfrom 来获取数据报的发送者了，应该改用 read 或 recv 等函数。
- 3) 有连接的 UDP 套接字引发错误，会返回给它所在的进程。无连接的 UDP 套接字不会。

对于第三点，内核只是检查是否存在**立即**可知的错误（例如一个显然不可达的目的地）。

6.4 UDP 有连接与无连接的性能对比

无连接：
在无连接 UDP 套接字上调用 sendto 时，Berkele 的内核会暂时连接该套接字， 发送数据报，然后断开该连接。比如下面这断代码：

```
// 连接两次调用 sendto
sendto(sockfd, buf, 100, 0, &servaddr, sizeof(servaddr));
sendto(sockfd, buf, 200, 0, &servaddr, sizeof(servaddr));
```

它会执行下面 6 个步骤：

- 连接套接字
- 发送第一个数据报
- 断开套接字连接
- 连接套接字
- 发送第二个数据报
- 断开套接字连接

第一次连接需要为目的 IP 地址搜索路由表，并进行缓存。第二次连接，会直接到缓存中取。

有连接：

```
connect(sockfd, &servaddr, sizeof(servaddr));
write(sockfd, buf, 100);
write(sockfd, buf, 200);
```

它会执行下面的步骤：

- 连接套接字
- 发送第一个数据报
- 发送第二个数据报

这种情况下，内核只复制一次含有目的 ip 和 port 的套接字地址，而使用 sendto 时，需要复制两次。

临时连接未连接的 UDP 套接字大约会耗费每个 UDP 传输三分之一的开销。

6.5 UDP 协议的不可靠性

发送数据包的数量越多，丢包率越高。且改变接收缓冲区，不能从根本上解决问题！

最后：tcp 与 udp 的端口号是彼此独立的，这在服务器中混合使用（多线程或者 select）TCP 与 UDP 时有体现。

7 高级 IO 函数与技术

这一次在网络编程中，我们要学习的高级 I/O 当然只是针对网络编程来说的，这次的内容包含函数和编程技巧的讲解：

- 为 I/O 操作设置超时
- recv 和 send 函数
- recvmsg 和 sendmsg 函数

这些知识会帮你把网络编程相关的知识提升一个新的高度。

7.1 套接字超时

我们知道，使用 read 或 recvfrom 函数从 socket 上读取数据时，可能会导致阻塞。特别是 UDP 服务器，稍有不甚，数据报就丢失就会直接导致程序假死（阻塞在 recvfrom）。

还有 connect 函数，如果去连接一个网络上不存在的主机，需要等待很长时间，可能长达 2 分钟。如果我们能控制它在指定的时间内连接不上，就直接报错，行不行呢？
很可惜，上面的问题 linux 并未直接提供带超时的函数帮我们解决，所以我们需要一些奇技淫巧来完成上面的功能。

7.12 如何设置超时

常用的有 3 种方法来设置超时：

- 1) 使用 alarm 函数，一旦到期就产生 SIGALRM 信号，打断正在执行的系统调用。
- 2) 使用 select 函数。
- 3) 使用套接字选项 SO_RCVTIMEO 和 SO_SNDTIMEO.

关于这几个方案，有几点要说明：

- 三个技术都可用于输入和输出(read 和 write)
- 对于阻塞的 connect 只能使用方案一，对于非阻塞的 connect，只能使用 select.
- 方案三对 connect 不适用。
- 方案一和二适用于任何描述符，而方案三仅适用于套接字描述符。

7.13 使用 alarm 设置超时

它的伪代码如下，这里以 connect 为例：

```
void sighandler(int sig) {
  // 信号处理函数，什么也不用做
  return;
}

int connect_timeo(/*...*/, int nsec) {
   // 注册信号
   registSignal(SIGALRM, sighandler, &oldhandler);
   // 开启一个 nsec 秒的 alarm
   alarm(nsec);
   ret = connect(/*...*/);
   // 如果被信号打断，则表示超时，此时 connect 函数返回-1，且 errno 全局变量被
//设为 EINTR,
   if (ret < 0 && errno == EINTR) {
     errno = ETIMEDOUT;
   }
   // 删除 alarm
   alarm(0);
   registSignal(SIGALRM, oldhandler);
   return ret;
}
```

7.14 使用 select 设置超时

使用 select 设置超时的原理是：
它使用了 select 函数的最后一个超时参数，对套接字描述符进行了“预读”。

- 用于判断是否可读的函数

```
// 该函数利用 select 函数判断在 nsec 时间内描述符 fd 是否可读
int readable_timeo(int fd, int nsec) {
  int ret;
  fd_set rfds;
  struct timeval tv;

  FD_ZERO(&rfds);
  FD_SET(fd, &rfds);

  tv.tv_sec = nsec;
```

```
  tv.tv_usec = 0;

  ret = select(fd + 1, &rfds, NULL, NULL, &tv);

  return ret;
}
```

- 带超时的 recvfrom 函数

```
int recvfrom_timeo(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen, int nsec) {
  int ret;

  ret = readable_timeo(sockfd, nsec);

  // select 函数返回 0，表示超时
  if (ret == 0) {
    errno = ETIMEDOUT;
    ret = -1;
  }
  else {
    ret = recvfrom(sockfd, buf, len, flags, src_addr, addrlen);
  }

  return ret;
}
```

7.15 设置套接字选项 SO_RCVTIMEO 与 SO_SNDTIMEO 来设置超时

设置套接字超时的第三个技术是使用套接字选项 SO_RCVTIMEO 与 SO_SNDTIMEO，它的优势在于一次设置，所有应用于该套接字的操作都自动带有超时时间。它就好像是一个全局开关。

比如对于 SO_RCVTIMEO 来说，如果设置了它，所有对该套接字的读操作在规定的时间里没完成，就直接返回并设置 `errno = EWOULDBLOCK`，对于 SO_SNDTIMEO 选项来说也是一样。

- 设置 SO_RCVTIMEO 超时

```
sockfd = socket(/*...*/);

// 设置超时时间为 5 秒

struct timeval tv;

tv.tv_sec = 5;
```

```
tv.tv_usec = 0;

setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

经过一次性设置后，以后调用 read 系列的函数，最多只能阻塞 5 秒。对于 SO_SNDTIMEO 选项来说设置方法也是一样的。

- recvfrom 判断超时

如果 recvfrom 返回 EWOULDBLOCK，说明超时。

```
nr = recvfrom(sockfd, /*...*/);

if (nr < 0) {

   if (errno == EWOULDBLOCK) {

      // 超时

}

}
```

7.2 recv 和 send 函数

recv 和 send 函数专门用于套接字描述符的，相比于 read 和 write，前三个参数一模一样。recv 和 send 只多了一个参数 —— flags.

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
 ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

参数 flags：

选项	说明
MSG_DONTROUTE	告诉内核目的主机在直连的本地网络上（同一网段），不要使用网关将数据包路由出去（**仅支持 send 和 sendto 函数**）
MSG_DONTWAIT	临时将套接字描述符设置成非阻塞（**支持 recv 和 send**）
MSG_OOB	对于 send，表示即将发送带外数据（out-of-band data）；对于 recv 来说，表示即将读入带外数据（**支持 recv 和 send**）
MSG_PEEK	查看可读取的数据，返回后内核不会将已读取的数据从接收缓冲区中删除。（**仅支持 recv 和 recvfrom 函数**）
MSG_WAITALL	指定该标志，就相当于 readn 函数。（**仅支持 recv 和 recvfrom 函数**）

7.3 散布读 与 聚集写

看下面一段代码：

```
char buf1[10];

char buf2[20];

char buf3[15];

write(fd, buf1, 10);

write(fd, buf2, 20);

write(fd, buf3, 15);
```

上面这样的代码实际上很常见，不知道你是否还记得我们在讲解 TCP_NODELAY（关闭 Nagle 算法） 套接字选项的时候，遇到过一种 write-write-read 的情形，和这个很类似。

接下来我想说的是，有没有一种办法，只执行一次 write 就可以把所有 buf 中的数据写入？最容易想到的就是将三个 buf 缓冲中的数据合并成一个 buf，一次写入。linux 为我们提供了另一个更加方便的函数——writev，它意为 write vector，即一次写入很多个 buf，比如可以这样：

```
writev(fd, buf1, 10, buf2, 20, buf3, 15);
```

当然啦，上面这种方法只是为了方便描述，实际上 writev 函数的参数是一个结构体数组。

writev 函数和 readv 函数原型：

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

struct iovec {
  void  *iov_base;    /* 缓冲区地址 */
  size_t iov_len;     /* 要读写多少字节 */
};
```

7.4 recvmsg 和 sendmsg 函数

这两个函数用起来过于复杂，暂时不研究。参考文档：链接

7.5 套接字与标准 IO

避免在套接字上使用标准 I/O 函数库。
因为标准 IO 函数都带有自己的缓冲

8 UNIX 域协议

8.1 引入（如何在不同的进程间传递描述符）

正式进入 Unix Domain Protocols —— Unix 域协议。
先来看看 unix 域协议的一个应用：如何在不同的进程间传送描述符
早先在 Linux 环境编程中学习 fork 函数的时候，我们就知道子进程可以继承父进程所有打开的描述符，很明显，这是一种传递描述符的方式。

问题：

> **更进一步，能否将子进程打开的描述符传递给父进程呢？再者，能否在非亲缘的进程间传递描述符？**

这个问题的确有一定的使用价值：它可以借助第三方进程打开我们本来没有权限打开的文件，并获取该文件的描述符。

8.2 UNIX 域套接字地址结构

不同于 ipv4 的 AF_INET，unix 域只用于本机进程间通信，它所使用的完全是另一套协议。在使用 unix 域的时候，socket 函数的第一个参数必须指定为 **AF_LOCAL** 或者 **AF_UNIX**，表示创建一个 unix 域套接字。

不同于 ipv4 的 sockaddr_in{} 结构，unix 域的套接字地址结构为 sockaddr_un{}

```
struct sockaddr_un {
  unsigned short int sun_family;
  char sun_path[108];
}
```

sockaddr_un 成员里没有 ip 地址与端口号的概念，它的成员 sun_path 是某个具体的文件路径，该**路径必须以`'\0'`作为结尾**。这种方式称为普通命名。

另一种方式是抽象命名，此时需要让 sun_path[0] = 0. 这个我们后面再说，这里我们先学会普通命名。

构造这样的套接字地址非常简单：

```
struct sockaddr_un addr;
addr.sun_family = AF_UNIX; // 或者让它等于 AF_LOCAL，它们的值是一样的
strncpy(addr.sun_path, "/tmp/dog", sizeof(addr.sun_path) - 1);
注意：unix 域套接字关联的路径应该是一个绝对路径名，而不是相对路径
sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr*)&addr1, sizeof(addr1));
```

8.3 UNIX 域字节流回射服务器

很久以前我们早已学过各种各样的进程间通信的方法，比如无名管道，有名管道共享内存等等。而使用 unix 域协议进行通信，也是一种进程通信的方法。所以以后面试官问你进程间通信手段，不要忘记了这种。
知道了 unix 域的套接字地址以及绑定后，其它的和 ipv4 的操作没什么两样

8.4 抽象 UNIX 域套接字地址

前面我们已经学习过了 sockaddr_un 结构，它有一个成员 sun_path，通常它保存的是一个以`'\0'`为结尾的绝对路径。一旦绑定了一个绝对路径，就会生成一个文件。这种 unix 域套接字地址我们称为**普通 unix 域套接字地址**。

还有一种 unix 域套接字地址，它的特征是 sun_path[0] 是 null，即`sun_path[0] == '\0'`. 这种 unix 域套接字地址称为**抽象 unix 域套接字地址**。

使用这种套接字地址，它并不会在文件系统中产生真正的套接字文件，而是由内核帮我们维护一个抽象套接字文件，它总是以`'@'`开头，比如`@dog`。所以，它的好处自然很明显，不用产生实际文件了。

8.5 socketpair 函数

如果涉及到父子进程间的通信，我们就没有必要弄的那么麻烦，不需要再创建套接字地址，不需要绑定。函数 socketpair 会创建两个连接好的套接字。

```
int socketpair(int family, int protocol, int  sockfd[2]);
```

family 参数只能是 AF_LOCAL 或 AF_UNIX，protocol 只能是 0. type 参数既可以是 SOCK_STREAM，也可以是 SOCK_DGRAM.

将 type 指定为 SOCK_STREAM 得到的结果称为流管道(stream pipe)，它和 pipe 创建的普通 unix 管道非常像，只不过，stream pipe 是全双工的。所以使用起来，它比 pipe 函数更加方便。

下面是使用 socketpair 函数用于父子进程通信的伪代码：

```
int main() {
  int nr, pid, sockfd[2];
  char buf[4096];
  socketpair(AF_LOCAL, 0, sockfd);
  pid = fork();
  if (pid == 0) {
    // 子进程关闭 0 号描述符，因为它只使用 1 号
    close(sockfd[0]);
    nr = read(sockfd[1], buf, 4096);
    write(STDOUT_FILENO, buf, nr);
    exit(0);
  }

  // 父进程关闭 1 号描述符，它只使用 0 号。
  close(sockfd[1]);
  // 父进程向流管道写入数据
  write(sockfd[0], "hello", 5);
  waitpid(pid, NULL, 0);
  return 0;
}
```

8.6 进程间传递描述符

recvmsg 和 sendmsg 函数的第二个参数类型。

```
struct msghdr {
  void         *msg_name;
  socklen_t     msg_namelen;
  struct iovec *msg_iov;
  size_t        msg_iovlen;
  void         *msg_control; // 辅助数据
  size_t        msg_controllen; // 辅助数据大小
  int           msg_flags;
};
```

要想在进程间传递描述符，使用普通方法直接传递描述符是行不通的，所以辅助数据在这里派上了用场。
辅助数据的类型是 cmsghdr{}，如下：

```
struct cmsghdr {
  socklen_t     cmsg_len;
  int           cmsg_level;
  int           cmsg_type;
/* followed by
  unsigned char cmsg_data[]; */
};
```

如果要想在进程间传递描述符，可通过适当的设置 cmsg_level 和 cmsg_type 的值，并把描述符的值保存在 cmsg_data 中，就可以通过 sendmsg 将描述符发送给其它进程了。

msghdr{} 的成员 msg_control 是一个指针，它指向一个 cmsghdr{} 类型的"数组"。准确的说，它指向第一个辅助数据对象。

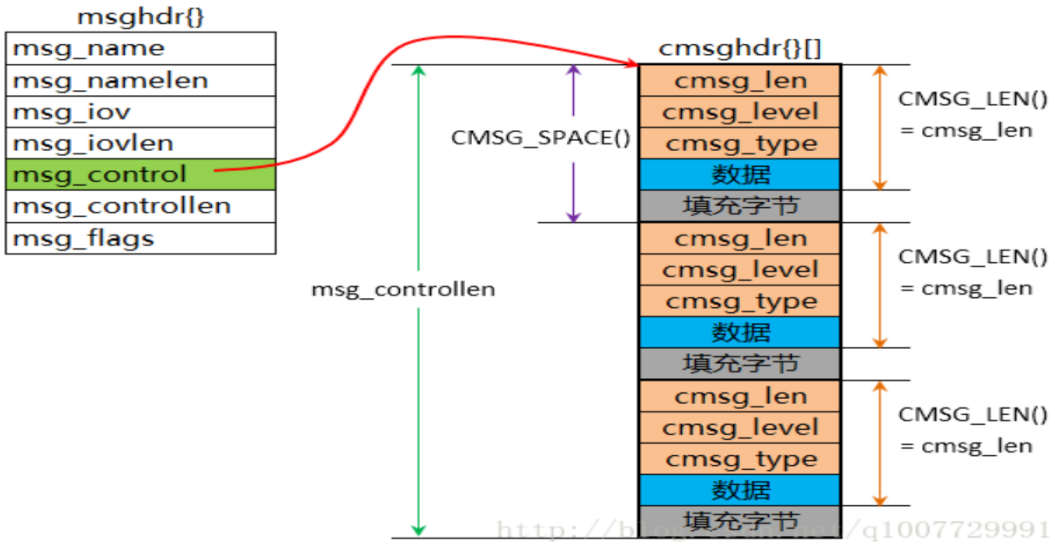


图1 msghdr{} 与 cmsghdr{}

详细说明见文档：辅助数据 感觉不会考，暂时不细究了。

我们已经知道，可以通过辅助数据传递描述符了，那么接下来怎么做？
大概是一下结构，注意理解 UNIX 域套接字实现进程间通信并传送描述符。

```
int main() {
  /****************** 封装成 myOpen ******************/
  // 调用 socketpair
  pid = fork();
  if (pid == 0) {
    // 执行 openfile，它打开 filename 文件，将把描述符 sendmsg 给父进程
    execl("./openfile", "./openfile", sockfd[1], filename, mode);
    exit(1);
  }

  // 从子进程读取数据
  recvmsg(sockfd[0], &msg, 0);
  // 从 msg.msg_control 拿到描述符 fd ...
  /****************** 封装成 myOpen ******************/

  // 从文件读取数据并打印到屏幕
  read(fd, buf);
  write(STDOUT_FILENO, buf);
}
```

9 非阻塞 I/O

9.1 select + 阻塞 IO（bug）

Select+阻塞 IO 的客户端会产生 BUG，原因如下：
我们的客户端伪代码大概这样：

```
while(1) {
  rfds = {stdin, sockfd};
  select(rfds);
  if (stdin in rfds) {
    read(stdin);
    // 风险代码，可能产生阻塞
    writen(sockfd);
  }

  if (sockfd in rfds) {
    read(sockfd);
    writen(stdout);
  }
}
```

writen(sockfd)永远阻塞？
writen 函数表示写 n 字节的字符，只要这 n 个字节没有写完，writen 就会一直尝试发送，直到全部写出去。writen 如果要阻塞，唯一的可能就是发送缓冲区满了。但是服务器不是一直都在接收数据吗？总有一个时候发送缓冲区会有空闲出来吧，这样看起来，writen 最多只会临时阻塞。
其实不然!!
只要不断的增大 writen 发送的数据量，最终就会导致客户端永久阻塞。这不是服务器的问题，服务器一次接收 4096（或其他） 字节，然后再将其发送回去，这没什么问题。最终服务器阻塞在了 writen 上，只是因为客户端那边——没有及时的收数据。这导致服务器发送缓冲区也被填满。

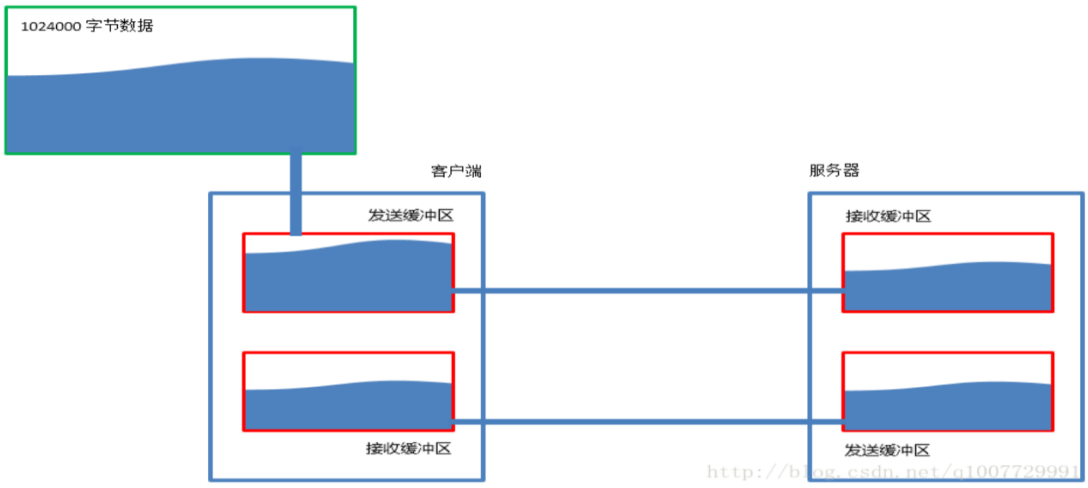


图3 客户端一次 write 1024000 字节，当前尚未阻塞

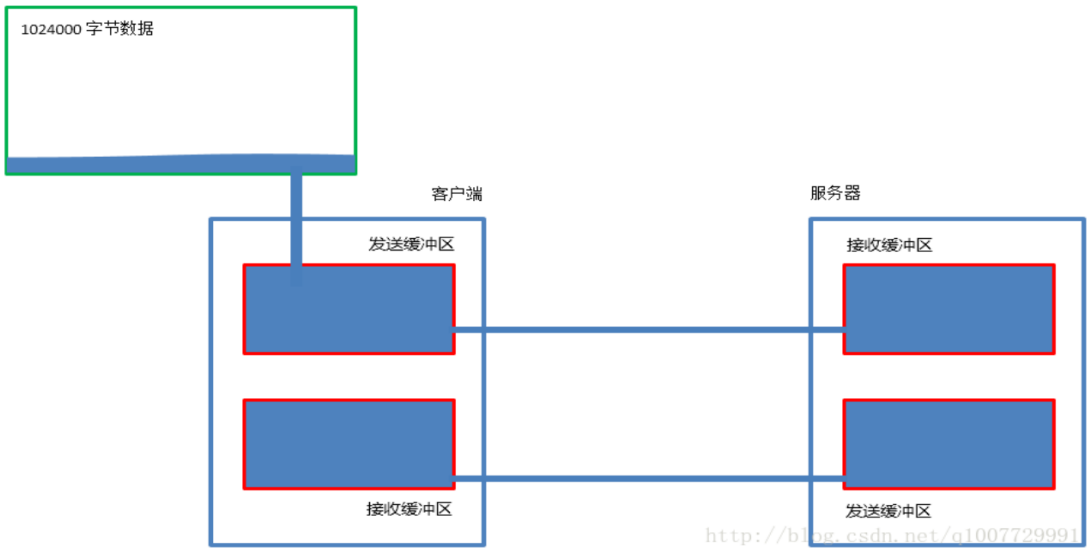


图4 客户端一次 write 1024000 字节，双方的接收和发送缓冲区都被填满

图 4 描述的情况是，双方的接收和发送缓冲区都被填满。不幸的是，客户端的 1024000 个字节还没发完，阻塞在了 writen 上。然而，客户端这个时候并没有机会去 read(sockfd)。因为客户端的代码是下面这样。

```
if (stdin in rfds) {
  read(stdin);
  // 风险代码，可能产生阻塞
  writen(sockfd);
}
```

客户端没有机会 read，最后客户端的接收缓冲区也被填满，因此客户端会向服务器发送 **0 窗口通告**。服务器收到 0 窗口通告后，就不再向客户端发数据，于是服务器的发送缓冲区最终被填满，服务器也阻塞到了 writen 上。

9.12 解决方案（非阻塞 IO）

如何解决上述的问题：

- 在客户端的应用层增加发送缓冲区和接收缓冲区。不使用 writen 而改为 write 函数。另一方面，即使是使用 write 函数，谁能保证它就没有阻塞风险呢？最佳的策略是使用非阻塞 IO。
- 使用多进程或多线程客户端，将 read 和 write 放到两个不同的进程或线程中。

9.2 非阻塞 I/O

操作阻塞与非阻塞函数的分类：

1) 输入操作，包括 read, readv, recv, recvfrom, recvmsg.

对于 TCP 和 UDP 而言，如果接收缓冲区为空，调用这些函数就导致进程阻塞。如果套接字是非阻塞的，则返回 -1, 置 errno 为 EWOUDBLOCK.

2) 输出操作，包括 write, writev, send, sendto, sendmsg.

对于 TCP 而言，如果发送缓冲区满了，调用这些函数会导致进程阻塞。UDP 没有发送缓冲区，因此不太可能阻塞（unp 说可能会因为其它原因而阻塞）。如果套接字是非阻塞的，同样返回 -1，errno 置为 EWOULDBLOCK.

3) 接受连接，accept 函数。

如果没有新的连接，调用 accept 会导致进程阻塞。对于非阻塞套接字，返回 -1，errno 置为 EWOULDBLOCK.

4) 发起连接，connect 函数。

对于 TCP 来说，connect 函数要等到到自己的 SYN 的 ACK 收到才会返回。虽然 UDP 也可以使用 connect，但是它的含义等同于 setpeername，所以不会阻塞。

对于非阻塞套接字，如果连接不能立即建立，connect 在发送 SYN 后直接返回了，此时 errno 设置为 EINPROGRESS.

9.3 使用非阻塞 IO 与多线程改写客户端

使用非阻塞 IO 时注意要设置 IO 为非阻塞的，如：

```
setNonblock(sockfd);
setNonblock(stdin);
setNonblock(stdout);
```

更推荐使用多线程：

非阻塞 I/O+缓冲区 处理起来确实很麻烦，有时候代码的复杂程度可能会让你得不偿失。使用多线程 + 阻塞 I/O 其实是更为推荐的方法，而且程序的效率也不会比非阻塞 I/O 差多少

既然 writen 函数可能会阻塞，那就把读写 sockfd 这两个操作放到不同线程。

9.4 非阻塞 connect

非阻塞 i/o 上调用 connect 比非阻塞 i/o 上调用 read/write 要麻烦一点，一方面 connect 函数不能像 read/write 那样反复调用，它只能调用一次；另一方面，connect 函数返回错误，并不代表连接建立不成功（下面第二种情况）。

对于 TCP 协议，在非阻塞 i/o 上调用 connect，意味着 connect 会发送 SYN 段给服务器：

- 如果在 connect 返回时，收到了服务器的 ACK，则 connect 返回 0，意味着连接建立成功，这通常只会发生在本机连接上。
- 如果在 connect 返回时，未收到服务器的 ACK，则 connect 返回 -1，同时 errno 置为 EINPROGRESS，这个错误表示“正在进行……”.
- 如果 connect 返回错误，errno 不是 EINPROGRESS 可以立即判断连接建立失败。

9.41 如何判断连接成功或失败

对于 TCP 连接：

- 连接建立成功：**套接字描述符可写**。
- 连接建立失败：**套接字描述符可读可写**。

注 1：上面说的可读可写的含义是 select 函数返回的读集合和写集合是否存在这个描述符。若描述符在读集合里，说明可读；如果在写集合里，说明可写。
注 2：一般来说，新创建的描述符，在执行 connect 前既不可读也不可写，连接成功后，则变得可写（这是一定的），但是不一定可读（没有数据就不可读，读会返回-1）。如果新创建的描述符变得可读了，大概率意味着出错，因为错误是通过 read 系统调用间接返回的。

根据上面两条规则，我们可以使用 select 来处理这两种情况。事先建立读写集合，然后使用 select 监听。
但是，反过来根据套接字描述符**可读可写**来判断连接成功是不可行的。换句话说，下面这样做是**不对的**：

- 如果可写不可读：连接建立成功（可以这样判断）
- 如果可写可读：连接建立失败（不可以这样判断）

  原因很简单，可写可读，并不一定就是失败，也许是对端发来数据了呢？所以，得使用另外一种办法来判断——使用套接字选项 SO_ERROR. 这也是 SO_ERROR 极少能派上用场的地方之一。

  只要套接字描述符变得可读或可写，直接使用 `getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len)` 取得套接字的状态。

9.42 并发连接数太多可能出现的问题

客户端一味的追求并发连接数，可能会导致性能下降，特别是在网络状况不好的时候。前面我们介绍过慢启动和拥塞避免算法的细节。

如果同时发起多个连接，其中某个连接遇到 TCP 报文丢失，很可能已经网络已经拥塞，于是执行**乘法减小算法（Multiplicative Decrease）**，但很可惜的是，其它连接并不会得到通知，这种情况下，其它连接仍然还在无脑的一次发送多个 TCP 报文，接下来导致网络更加拥塞。

9.5 非阻塞 accept

容易产生错误的代码：

```
while(1) {
   rfds = {listenfd, ...}
   select(maxfd + 1, &rfds, ...);
   if (listenfd in rfds) {
      // 可能会阻塞
      accept(listenfd, ...);
   }
}
```

在**有些系统的实现**中，如果在 select 返回后，accept 调用前，连接异常终止，内核会把该连接从已连接队列中删除，并释放相应的已连接套接字。如此一来，就好像连接从未发生过一样，这时候调用 accept 就可能会阻塞。

另外有些系统会让 accept 直接返回一个 ECONNABORTED 错误或 EPROTO 错误。为了防止 accept 被阻塞，应该使用非阻塞 i/o. 这样就可以捕捉到 EWOULDBLOCK 错误，让程序继续下去。

```
// 设置 listenfd 为非阻塞套接字。
setNonblock(listefd, 1);
while(1) {
   rfds = {listenfd, ...}
   select(maxfd + 1, &rfds, ...);
   if (listenfd in rfds) {
      ret = accept(listenfd, ...);
      if (ret < 0) {
        // 如果是 EWOULDBLOCK 则继续循环
        if (errno == EWOULDBLOCK) continue;
        ERR_EXIT("accept");
      }
   }
}
```

10 ICMP 协议

本章开始讲解 ICMP 协议，通过学习 ICMP 协议，我们可以顺便掌握 Raw Sockets，中译名为**原始套接字**。在学习 ICMP 前，还需要简单的了解一下 IP 协议。

IP 协议是 TCP/IP 协议族中最核心的协议，TCP、UDP、ICMP 等众多协议需要依赖它工作。

10.1 IP 协议（基础）

IP 首部：

图1 IP 首部

4 位版本号

目前协议版本号是 4，即 IPv4.

4 位首部长度

它的单位是 4 字节。比如如果首部长度是 0x05，则首部长度就是 20 字节。

服务类型（TOS）

不同的应用程序，可能有不能的需求，有的需要大吞吐量，有的要低时延，有的要可靠性高等等，该字段可以设置不同的值，来满足不同应用程序的需求。如今，大多数 TCP/IP 协议都不使用该字段了，将其设置为 0.

总长度字段

单位是 1 字节。利用总长度字段和首部长度，可以计算出 IP 数据报中内容的起始位置和长度（总长度减首部长度）。因为该字段为 16 bit，这意味着 IP 数据报总长度可以达到 65535 字节。

标识字段

该字段能够唯一标识主机发送的每一个完整的数据报。所谓一个完整的数据报，是指两种情况：

- 如果 IP 数据报不分片，它本身就是完整的
- 如果 IP 数据报被分片，则所有**标识号相同的数据报**可以组装成一个完整的数据报。

3 位标识

这三个标志位分别是：

- RF (Reservd Fragment，保留位，总是为 0)
- DF (Do Not Fragment，不分片，没有分片)
- MF (More Fragment，还有更多分片)

TTL 字段

该字段表示该数据报最多可以经过多少路由器。当该字段为 0 时，数据报就被丢弃。

首部检验和

只计算 IP 首部的检验和，该算法名为二进制反码和（ones-complement sum）。为了计算一份数据报的 IP 检验和，首先把检验和字段设置成 0，然后对首部每 16 bit 进行二进制反码求和。

8 位协议

表示 IP 数据报的数据部分承载的是什么协议。比如该值为 6 时，则 IP 数据报承载 TCP 协议，如果是 17，则承载 UDP 协议，如果是 1，则承载 ICMP 协议。

回忆 socket 函数的第三个参数 —— protocol，在创建 IP 数据报时，内核会将该值填入到 8 位协议字段或者该套接字只接收 protocol 协议的 IP 数据报。

你可以在 netinet/in.h 文件中找到所有协议代号。

源 IP 地址和目的 IP 地址

每一个 IP 数据报都包含这两个字段，它标识了发送者和接收者的 IP 地址。

可选项

该选项可以不存在。如果有的话，通常有下面几种：

- 安全和处理限制
- 记录路径
- 时间戳

- 宽松的源站路由
- 严格的源站路由

10.2 接收 IP 数据报

本文我们需要用到一种新的类型的套接字 —— Raw Sockets，原始套接字。
它出现的理由很简单，我们可以自己构造一个完整的 IP 数据报，通过原始套接字发送出去。也可以从原始套接字中读取一个完整的 IP 数据报。

```
// 创建一个 IPv4 原始套接字
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

注意：只有 root 权限的用户才能创建原始套接字

第三个参数 protocol 在上一节讲解 IP 协议的时候讲过，它是 IP 数据报中的 8 位协议字段。

```
// 使用 recvfrom 接收 ip 数据报，它会将网卡上所有符合要求的 ip 数据报读取到 buf 中。recvfrom 的最后两个参数可以是 NULL.
    nr = recvfrom(sockfd, buf, BUF_SIZE, 0, (struct sockaddr*)&from, &len);
```

10.3 ICMP 协议（基础）

ICMP 协议（Internet Control Messages Protocol，网际控制报文协议）是网络层最重要的协议之一。前面的实验我们也看到了，它由 IP 协议承载，封装在 IP 协议的数据部分。

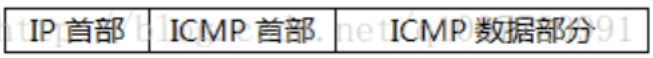


图1 ICMP 被封装在 IP 数据报的数据部分

1、ICMP 报文的作用

网络上经常会出现一些错误，比如目标主机不存在，目标端口不存在什么的，那么你要怎么知道呢？ICMP 的功能之一就是将错误反馈给你，这种 ICMP 报文通常被称为差错报文，表示网络出问题啦！

当然，ICMP 还有其它的功能，比如你发送一个时间戳请求 ICMP 报文给一台主机，那台主机知道这是请求时间的，就会把自己的时间放在 ICMP 报文中，然后再发回去给你，像这种 ICMP 报文称为 ICMP 查询报文和 ICMP 应答报文。

还有一种就是用于控制网络行为的，比如路由重定向啊等等。

当然啦，上面只是粗略的分类，后面我们写程序来练习。

2、ICMP 首部

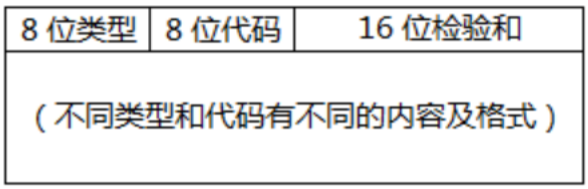


图2 ICMP 首部格式

任何 ICMP 报文的首 4 字节的格式都是一样的，如图 2 所示。我们程序在收到了 ICMP 报文的时候，可以根据 ICMP 的类型，代码号来判断这个 ICMP 报文是哪个类别。

比如类型号是 3，表示这是差错报文，具体到是哪种差错报文，还要看 8 位代码号。如果类型号是 13 表示这是一个时间戳请求报文等等。

我列举的几个，是我们将要学习的几种（用 type 表示类型，code 表示代码号）：

- ICMP 回显与应答（type = 8 或 0，code = 0）
- ICMP 地址掩码请求与应答（type = 17 或 18，code = 0，已废弃， 如今已使用 DHCP 协议取代，参考 RFC 6918 的 2.4 和 2.5 小节.）
- ICMP 时间戳请求与应答（type = 13 或 14，code = 0）
- ICMP 端口不可达差错报文（type = 3，code = 3）

10.4 接收 ICMP 报文

如果你练习过前面的接收 IP 数据报的程序，相信写出这个不会很难。

因为这一次我们只想要 ICMP 协议，因此可以只接收 IP 数据报中协议字段为 1 的 IP 数据报。具体就是创建下面这样的原始套接字：

```
// IPPROTO_ICMP 的宏定义值就是 1
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

然后从 IP 数据报中拿到 IP 数据部分，即拿到 ICMP 报文即可。代码略，原理懂即可。

10.4 ICMP 协议（回显请求与应答）

当 ICMP 首部 type = 8, code = 0，该 ICMP 是回显请求报文。当 type = 0, code = 0 时，是回显应答报文。

1、回显请求与应答报文首部格式

图1 ICMP 回显请求与应答报文首部

当 ICMP 报文是回显请求与应答报文时，我们可以看到首部的第 4、5 两个字节是标识符字段，第 6、7 两个字节是序号字段。（程序员都是从 0 开始计数）

```
// icmp 回显报文头部
struct icmp_echo {
  uint8_t icmp_type;
  uint8_t icmp_code;
  uint16_t icmp_cksum;
  uint16_t icmp_id;  //标识符
  uint16_t icmp_seq; //序号
  char icmp_data[0];
};
```

2、回显请求与应答过程

应用进程可以发送一个 type = 8, code = 0 的 ICMP 回显**请求**报文给目标主机，目标主机收到该 ICMP 报文后，将该报文的 type 字段值改为 0（回显应答报文），原封不动的发送回去。

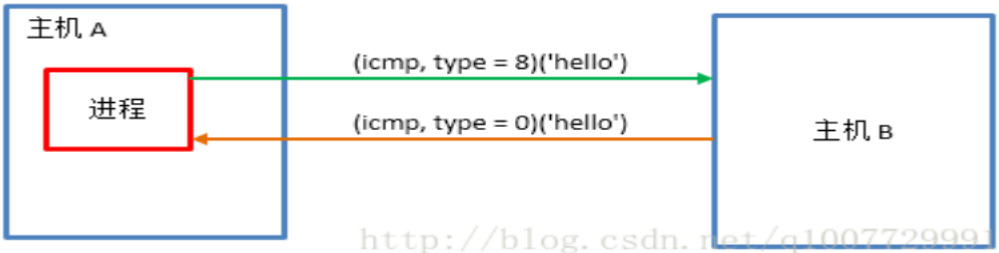


图2 ICMP 回显请求与应答

3、程序设计

之前已经练习过接收 ICMP 报文了，但是我们还没有写过如何发送 ICMP 报文，其实发送也很简单。不过要注意，要不要我们自己构造 IP 数据报？

回答：**默认情况下，我们不需要自己构造**，直接构造好 IP 数据报的内容发出去就行了。如果你想自己构造，可以使用 setsockotp 函数指定选项 IP_HDRINCL，像下面这样：

```
int onoff = 1;

setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &onoff, sizeof(onoff));
```

在这里，我们并不想自己构造 IP 数据报，因此也不用开启 IP_HDRINCL 选项。所以我们只需要构造一个 ICMP 回显请求报文就行了。

```
char sendbuf[4096];
struct icmp_echo *icmp_echo = (struct icmp_echo*)sendbuf;
// 开始填充 icmp_echo
icmp_echo->icmp_type = 8; // 类型为 8，表示回显请求
icmp_echo->icmp_code = 0;
icmp_echo->icmp_cksum = 0; // 注意，这个字段一定要正确填写
icmp_echo->icmp_id = getpid() & 0xffff; // 这个字段随便你填，我就填一个进程的 id 号。
icmp_echo->icmp_seq = 0; // 序号，这个字段也是随便你填什么的。
strcpy(icmp_echo->data, "hello");

// 计算整个 icmp 报文的长度(首部长度 + 数据部分长度)
icmplen = sizeof(struct icmp_echo) + strlen("hello");

// 所有字段填写完成后，就开始计算校验和
icmp_echo->icmp_cksum = cksum((unsigned short*)sendbuf, icmplen);
```

如果校验和填不对，这个 IP 数据报会直接被你的主机一声不吭的丢弃。校验和的算法叫 one's complement sum，中译名为二进制反码求和，如果你感兴趣可以自行搜索它的算法。这里直接给出代码：

```
unsigned short cksum(unsigned short *addr, int len){
  unsigned int sum = 0;
  while(len > 1){
    sum += *addr++;
    len -= 2;
  }
```

```
    // 处理剩下的一个字节
    if(len == 1){
        sum += *(unsigned char*)addr;
    }

    // 将 32 位的高 16 位与低 16 位相加
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);

    return (unsigned short) ~sum;
}
```

伪代码：

```
void handler(int sig) {
    // 每隔一秒发送一个回显请求报文给目标主机
    sender();
    alarm(1);
}


int main() {
    registSignal(SIGALRM, handler);
    alarm(1);

    recver();
}

void sender() {
    // 构造 ICMP 回显请求报文发送给目标主机
    // ...
}

void recver() {
    // 接收 IP 数据报，并解析出 ICMP 报文
    // 过滤出 ICMP 回显应答报文(type = 0, code = 0)
    // 打印 ICMP 报文中的数据部分
}
```

当然你也可以使用线程的方式发送 ICMP 报文……无所谓啦。

完全可以将本程序稍作修改，它就是一个 ping 命令。

10.5 PING 命令实现

纯手工实现 PING 命令。

学完了 ICMP 回显请求与应答报文，就可以用它来实现我们平时使用的 PING 命令了。

ping 命令本质上就是给目标主机发送一个 ICMP 回显请求报文，然后等待对方答复 ICMP 回显应答报文，如果对方不应答，可能就是我们的主机和对方的网路不通。

ping 命令还可以计算出往返时间，其实这很简单，我们只要在发送 ICMP 报文的时候，记录一下当前时间，再接收报文的时候再记录一下时间，两个时间相减就是 RTT 时间。

ping 命令中的 ttl 字段可以从 ip 数据报首部中拿到。

```
#include "common.h"

#define BUF_SIZE 4096
#define ICMP_SIZE 64

char *hostname; // 主机名或 ip
char recvbuf[BUF_SIZE]; // 接收缓冲区
char sendbuf[ICMP_SIZE]; // 发送缓冲区

void run(); // ping 主程序
void handler(int sig); // 信号处理函数
int nsend, nrecv; // 用于统计
int64_t start; // 用于记录发送 ICMP 报文时的时间

int main(int argc, char* argv[]) {
  if (argc < 2) {
    ERR_QUIT("Usage: %s <hostname or ip>\n", argv[0]);
  }
  hostname = argv[1]; // 记录需要 ping 的主机
  registSignal(SIGALRM, handler); // 注册信号处理函数
  registSignal(SIGINT, handler);
  run();
}

void run() {
  struct ip *ip; // ip 首部
  struct icmp_echo *icmp_echo, *icmp_echo_reply; // 请求与应答报文结构体
  struct sockaddr_in to; // 目标主机地址
  int i, len, sockfd, ret, nr;
```

```
  int64_t timerecv, timesend;
  double rtt;

  nsend = 0; // 已经发送包的个数
  nrecv = 0; // 已经接收包的个数

  // 随机初始化
  for (i = 0; i < ICMP_SIZE; ++i)
    sendbuf[i] = "abcdefghijklmnopqrstuvwxyz"[i%26];

  // 构造套接字地址，端口并没有什么用，随便填
  ret = resolve(hostname, 0, &to);
  if (ret < 0) ERR_EXIT("resolve");


  // 创建原始套接字，只接收承载 ICMP 协议的数据报
  sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
  if (sockfd < 0) ERR_EXIT("socket");

  // 打印信息
  WARNING("PING %s (%s) %d bytes of data.\n", hostname,
inet_ntoa(to.sin_addr), ICMP_SIZE);

  icmp_echo = (struct icmp_echo*)sendbuf;
  ip = (struct ip*)recvbuf;

  // 用于时间统计
  start = now();

  // 填充 icmp 回显请求报文，这三个字段以后都不用再改，放在循环外面
  icmp_echo->icmp_type = 8;
  icmp_echo->icmp_code = 0;
  icmp_echo->icmp_id = getpid() & 0xffff;

  while(1) {
    icmp_echo->icmp_cksum = 0;
    icmp_echo->icmp_seq = nsend + 1; // 序号，起始序号为 1
    *((int64_t*)icmp_echo->icmp_data) = now(); // 当前时间，微秒
    icmp_echo->icmp_cksum = cksum((unsigned short*)sendbuf, ICMP_SIZE);
// 指定大小为 64

    // 发送数据
    ret = sendto(sockfd, sendbuf, ICMP_SIZE, 0, (struct sockaddr*)&to,
sizeof(to));
```

```
    if (ret < 0) {
      if (errno == EINTR) continue;
      ERR_EXIT("sendto");
    }
    ++nsend; // 序号加 1

again:
    // 接收 ip 数据报，超时时间设置为 5 秒。
    alarm(5);
    nr = recvfrom(sockfd, recvbuf, BUF_SIZE, 0, NULL, NULL);
    if (nr < 0) {
      if (errno == EINTR) {
        ERR_PRINT("TIMEDOUT.\n");
        continue; // 超时，丢包
      }
      ERR_EXIT("recvfrom");
    }

    // 从 ip 数据报里拿到 icmp 报文
    icmp_echo_reply = (struct icmp_echo*)((char*)ip + (ip->ip_hl << 2));
    if (icmp_echo_reply->icmp_type != 0 || icmp_echo_reply->icmp_code !=
0
        || icmp_echo_reply->icmp_id != (getpid() & 0xffff)) {
      // 收到的不是我们想要的包，重来继续收，不能使用 continue 是因为我们不能
判定包就丢失了
      goto again;
    }
    // 执行到这里说明收到了自己的包，将接收个数递增
    ++nrecv;
    // 计算往返时间
    timerecv = now();
    timesend = *((int64_t*)icmp_echo_reply->icmp_data);
    rtt = (timerecv - timesend) / 1000.0;
    LOG("%d bytes from %s (%s): icmp_seq=%d ttl=%d time=%.1f ms\n",
        ICMP_SIZE, hostname, inet_ntoa(to.sin_addr),
icmp_echo_reply->icmp_seq, ip->ip_ttl, rtt);
    sleep(1);
  }
}

void handler(int sig) {
  int64_t end;
  if (sig == SIGINT) {
    end = now();
```

```
    WARNING("\n--- %s ping statistics ---\n", hostname);
    WARNING("%d packets transmitted, %d received, %d%% packet loss,
time %dms\n",
        nsend, nrecv, (nsend - nrecv) * 100 / nsend, (int)(end - start)
/ 1000);
    exit(0);
  }
}
```

注意的就是为了防止对端没有回应应答报文而导致 recvfrom 阻塞，我们需要给 recvfrom 设置超时时间。使用 alarm 信号为读写 I/O 设置超时时间是惯用的技俩

代码中使用了 goto 语句，当然你也可以使用 while 循环，但在这里使用 goto 反而能让代码可读性更好。一味的抨击 goto 是不正确的，任何事物都有两面性。

10.6 ICMP 协议（时间戳请求与应答）

ICMP 时间戳请求允许系统向另一个系统查询当前的时间。

1、ICMP 首部格式

类型（17或18）	代码（0）	检验和
标识符		序号
发起时间戳		
接收时间戳		
发送时间戳		

图1 ICMP 时间戳请求与应答报文

它的 type 字段为 17（请求）或 18（应答），code 字段为 0.

- 发起时间戳：发送者的发送时间，由发送者填写
- 接收时间戳：接收者的接收时间，由接收者填写
- 发送时间戳：接收者的发送时间，由接收者填写

通常，接收时间戳等于发送时间戳。可以理解为接收者接收到 ICMP 时间戳请求报文后立即发送应答报文回去。

ICMP 时间戳请求与应答报文提供了毫秒级的分辨率，并且返回的时间是从午夜 00:00 开始到当前时间的毫秒数。这是它的一个缺陷。

2、ICMP 时间戳结构体

```
// icmp 时间戳头部
struct icmp_time {
  uint8_t icmp_type;
  uint8_t icmp_code;
  uint16_t icmp_cksum;
  uint16_t icmp_id;
  uint16_t icmp_seq;
  uint32_t icmp_origtime;
  uint32_t icmp_recvtime;
  uint32_t icmp_sendtime;
};
```

程序的编写方法其实和前面的 ping 命令基本上差不多，但是在这里需要注意的几个地方：

- 发送的时候，时间戳需要填写午夜到当前的毫秒数，计算公式如下：

```
// 绝对时间
cur = now() / 1000;
// 午夜到当前的毫秒数
origtime = cur % (24*60*60*1000);
```

- 接收到的时间，都是网络字节序，打印它或者使用它的时候，注意要转换成本机字节序。我们自己在填写发起时间戳的时候，可以直接填写本机字节序，也可以填写网络字节序，因为接收者不会修改发起时间戳的任何内容。

大量的 ICMP 请求会造成服务器的负担，为了防止被人利用，应该把服务器上的这些功能关掉（将时间戳请求与应答报文关闭）。

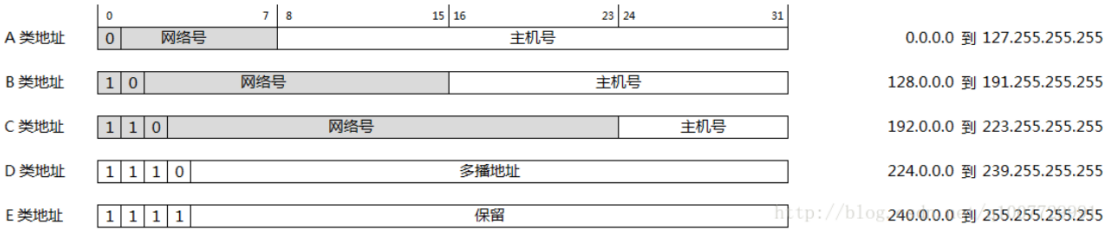
11 广播

11.1 IPV4 地址

很久以前我们非常简单的讨论过《IPv4 地址》，那时候是出于网络编程基础的需要，介绍了和 IP 地址相关的函数。这一次，我们来详细的讨论下 IP 地址的结构。

1、IPV4 地址分类

32 位的 IP 地址由 {netid, hostid} 两部分构成，根据这个规则，依据 netid 将 IP 地址分成 5 大类：



2、常用 IP 地址

A 类地址：

A 类地址的网络号占用 1 字节，第 1 位固定成 0，范围是 0 到 127 （包含 127），一共 128 个。但是，可指派的网络号只有 126 个，其中 网络号 0 和 127 我们不使用，它们用作特殊含义：

- 网络号为 0，表示本网络。
- 网络号为 127，保留作本地软件**环回测试(loop test)**本主机进程之间的通信之用。如 127.0.0.1

A 类地址的主机号占用 3 字节，它的范围是 $[0, 2^{24}-1][0, 2^{24}-1]$。主机号 0 和 $2^{24}-1$ 不能用作主机 IP 地址，它们的含义如下：

- 主机号 0 表示该 IP 地址是“本主机”所连接到的单个网络地址（例如，一主机 IP 地址是 5.6.7.8，则该主机所在的网络地址就是 5.0.0.0）。
- 主机号为全 1，表示“所有的”，即表示该网络上所有主机。（这是一个广播地址。）

B 类地址：

B 类地址的网络号占 2 字节，前 2 位固定(10)，范围是 128.0 到 191.255。然而，[COME06]规定，128.0 这个网络号是不能指派的！因此，B 类网络号最小也只能是 128.1，也就是说

最小的网络地址 128.1.0.0. 所以 B 类能用的网络数只有 $2^{14}-1$ 个。

B 类网络上最大的主机数是 $2^{16}-2$ 个，主机号全 0 表示网络地址，主机号全 1 表示广播地址。

C 类地址：

C 类地址的网络号占用 3 字节，前 3 位固定(110)，它的范围是 192.0.0 到 223.255.255，然而，[COME06] 规定，192.0.0 是不可使用的。因此 C 类地址能用的网络总数是 $2^{21}-1$ 个。C 类网络上最大的主机数是 $2^8-2=126$ 个，主机号全 0 表示网络地址，主机号全 1 表示广播地址。

下表总结了 IP 地址指派的范围：

网络类别	最大可指派的网络数	第一个可指派的网络号	最后一个可指派的网络号	每个网络中最大主机数
A	2^7-2	1	126	$2^{24}-2$
B	$2^{14}-1$	128.1	191.255	$2^{16}-2$
C	$2^{21}-1$	192.0.1	223.255.255	2^8-2

3、划分子网

鉴于基本的编址方法有以下缺陷：

- IP 地址空间利用率低
- 给每个物理网络分配一个网络号使路由表变得太大，使网络性能变坏
- 两级 IP 地址不够灵活，比如想添加一个新的网络，就得去因特网管理机构申请。

为了解决上面的问题，1985 年起在 IP 地址中又增加了一个“subnetid”，即子网号字段，它的结构如下：

IP 地址 = {netid, subnetid, hostid}

划分子网的方法是从网络的主机号中**借用**若干位作为子网号 subnetid，当然这也造成了主机号也就相应的减少了同样的位数。

1）子网掩码

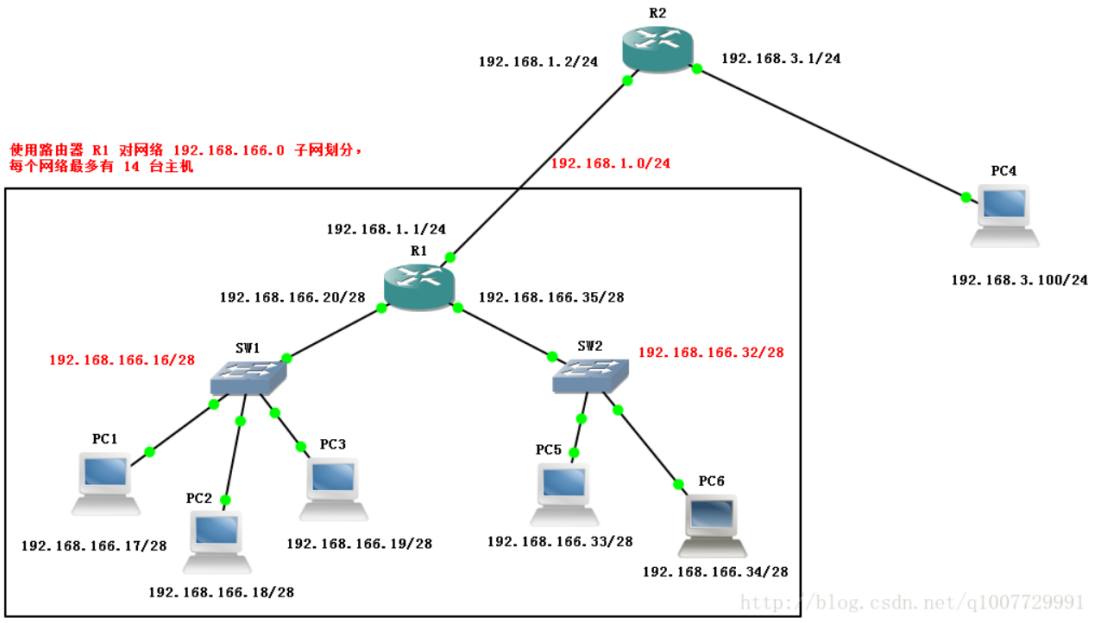


图2 将 C 类网络 192.168.166.0 划分成了 2 个子网

具体划分方法详细说明参考原文：原文，这里我们只记录子网掩码等的作用概述

192.168.166.0 划分出了两个子网，分别是 192.168.166.16 以及 192.168.166.32.

假设 PC4 发送了一份 IP 数据报给目标主机 192.168.166.34 主机，PC4 并不知道目的主机所连接的网络是否进行了子网划分（要知道，IP 协议首部并不包含子网掩码的信息）。

当路由器 R1 接收到了发给 C 类网络地址 192.168.166.34 的数据报后，应该如何才能将其转发到子网 192.168.166.32 上呢？此时就需要子网掩码来帮我们搞定它。

具体做法就是从 hostid 拿出若干位，用来表示 subnetid，在图 2 这个例子中，我们从 hostid 中拿出了 4 位做子网号。

例如 192.168.166.34 拆分成 2 进制（只拆分最后一段）：

```
192.168.166.00100010 --> {192.168.166}{0010}{0010}
```

因此子网号是 0010，即 10 进制的 2，而主机号是 0010，10 进制的 2.

为了能让路由器方便的从目的 IP 地址中提取出要找的子网的网络地址，就需要使用**子网掩码**。子网掩码是由一串 1 跟随一串 0 组成，子网掩码中 1 对应于 IP 地址中原来的 netid 加上 subnetid 部分，而 0 对应于 hostid 部分。

在图 2 这个例子中，192.168.166.0 这个网络中的主机的子网掩码是 255.255.255.240. 图 3 演示了如何根据子网掩码计算 netid, subnetid 和 hostid.

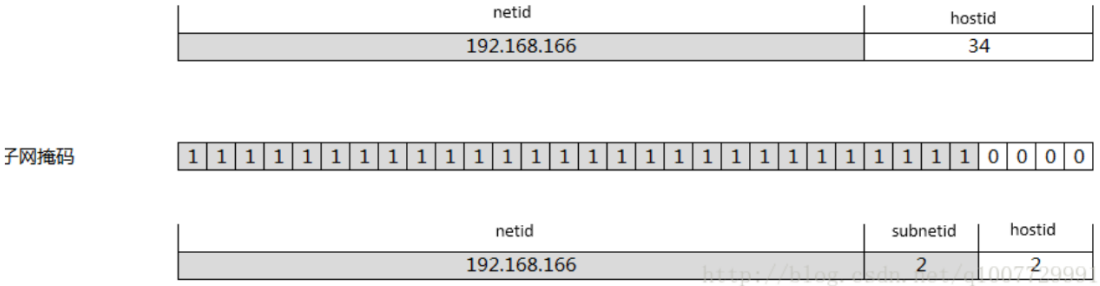


图3 根据子网掩码计算 netid, subnetid，以及 hostid

给路由器的每个接口设置了子网掩码后，路由器就可用“与”的计算方式，找到 netid 和 subnetid 部分啦。（192.168.166.34 & 255.255.255.240 = 192.168.166.32）

总结：子网掩码的作用就是与目标地址相与从而获得目标地址所在的子网划分

11.2 广播

详细内容参考：广播
我们要学习的广播是三层广播，这一层的广播地址，是指 IP 地址中主机号为全 1 的 IP 地址。

广播有什么用？简单粗暴的理解就是，如果目的地址是广播地址，则把该数据报发送给所有该网络上的主机。

现在基本已经淘汰了 A、B、C 类网络的说法，统一采用子网掩码来划分 IP 地址。
现在，我们只剩下受限的广播和指向子网的广播。

- 受限的广播（也有叫本地广播）：它的 IP 地址是 255.255.255.255. **在任何情况下，路由器都不会转发目的地址为 255.255.255.255 的数据报。**这样的数据报只会出现在发送者所在的网络中。
- 指向子网的广播：广播地址是主机号（hostid）全 1，同时指定了某个子网号（subnetid）的 IP 地址。比如上一篇博文中的例子，发往 192.168.166.16/28 这个子网的广播地址为 192.168.166.31/28.

（192.168.166.16==192.168.166.00010000->192.168.166.00011111==192.168.166.31）
比如：192.168.166.255/24，我们不再说它是指向网络的广播，应该说是**指向子网的广播**（因为已经没有 A、B、C 类地址的概念，因此也并无划分子网的概念，所有的网络都称为子网）。
注意：不再使用子网是指不在 32 位的 IP 地址中指明若干为做 subnetid。而是只有 subnetid 和 hostid。比如地址 192.168.166.4/24，我们就说它的 subnetid 是 192.168.166，hostid 是 4.

11.3 指向子网的广播

参考原文：指向子网的广播

11.4 受限广播地址

这种也称之为**本地广播地址**，它的目标地址为 255.255.255.255. 意思是只在本网络进行广播，绝对不会被路由器转发。

路由器只要检测到是一个广播地址，在转发的时候就会把目标地址修改成 255.255.255.255，我不能保证所有路由器是这样，目前我使用的路由器型号是思科的 c3600.
如果路由器不知道这是不是广播地址，那它就认为这是一个单播地址，直接按照路由表，该怎么转发就怎么转发了。

11.5 使用广播的 UDP 客户端

使用 UDP 而不使用 TCP，是因为 TCP 不支持广播！

这个程序非常简单，只要在客户端中修改两处：

- 因为客户端可以发送广播，则应该循环 recvfrom
- Linux 是默认情况下禁止发送广播的，要想发送广播，需要指定套接字选项 SO_BROADCAST.

```
int onoff = 1;
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &onoff, sizeof(onoff));
```

12 多播

详细参考：多播基础 多播地址
多播（或叫组播，**Multicast**）的出现，正是为了解决广播的不足。我们已经学会用 udp 发广播数据包，然而，对于不想接收广播数据包的主机来说，这就是一种骚扰。比如某主机接收到以太网帧（目的 MAC 地址为 ff:ff:ff:ff:ff:ff），一路沿着协议栈往上解包，到达 UDP 层，根据端口进行分发，但是并没有任何进程在这个端口监听，从而不得不丢弃这个数据包，这无疑是一种对资源的浪费。

多播，只有想接收多播数据包的主机，才会接收数据，不想接收的，就直接忽视。

多播地址，类似“QQ 群号”，它是 D 类 IP 地址（`224.0.0.0~239.255.255.255`）。

D 类 IP 地址的首 4 位总是 2 进制 1110 开头，因此，可用的范围就只剩下后 28 位。这 28 位称为 group id——**多播组 ID**，完整的 32 位称为 group address——**组地址**。

如何让你的进程加入多播组。
参考：如何将进程加入多播组

1、udp 客户端部分无需做任何更改，在指定目标 ip 地址的时候，直接使用多播地址。意思是说，向这个“QQ 群”发信息。

2、udp 服务器相对于之前的区别就是加入一个多播组，所以这里关键是如何加入多播组的问题。
一旦有多个服务器加入了同一个多播组，客户端向此多播组发送信息，所有的服务器都能收到信息。