计算机网络

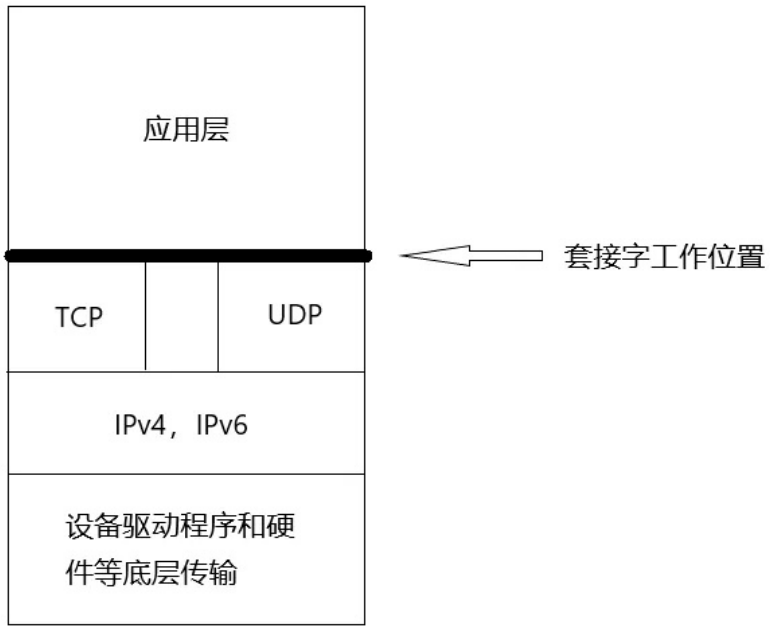
一 TCP协议

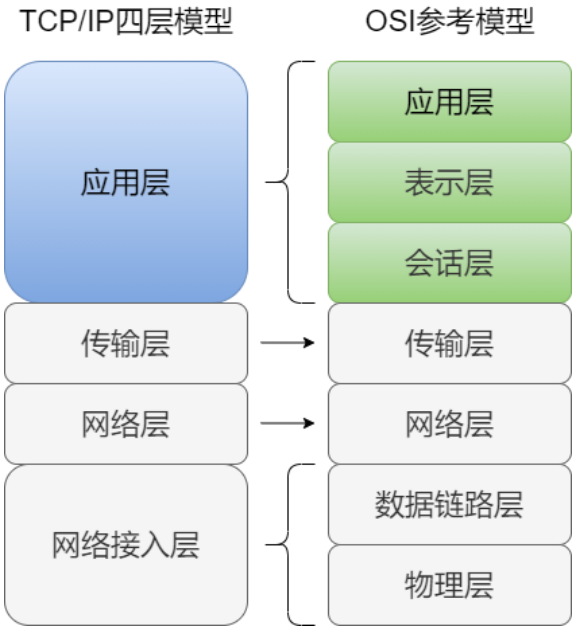
1 TCP连接简介

特性

- `TCP` 提供一种**面向连接的、可靠的**字节流服务
- 在一个 `TCP` 连接中，仅有两方进行彼此通信。广播和多播不能用于 `TCP`
- `TCP` 使用校验和，确认和重传机制来保证可靠传输
- `TCP` 给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复
- `TCP` 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制

传输模型





- **OSI参考模型**

OSI 参考层编号	等效的 OSI 层	TCP/IP 层	TCP/IP 协议示例
5、6、7	应用、会话、表示	应用	NFS、NIS、DNS、LDAP、telnet、ftp、rlogin、rsh、rcp、RIP、RDISC、SNMP 等
4	传输	传输	TCP、UDP、SCTP
3	网络	Internet	IPv4、IPv6、ARP、ICMP
2	数据链路	数据链路	PPP、IEEE 802.2
1	物理	物理网络	以太网 (IEEE 802.3)、令牌环、RS-232、FDDI 等等

- **主机A与主机B的数据传输路线(AB均使用网际网协议族)**

  A的应用层 --> A的传输层（TCP/UDP） --> A的网络层（IPv4，IPv6） --> A的底层硬件（此时已经转化为物理信号了） --> B的底层硬件 --> B的网络层 --> B的传输层 --> B的应用层

- **使用socket套接字编程时，简化掉了底层细节时的数据传输路线**

  A的应用层 --> A的传输层 --> B的传输层 --> B的应用层

  步骤：

  1.数据通过socket套接字**构造符合TCP协议的数据包**

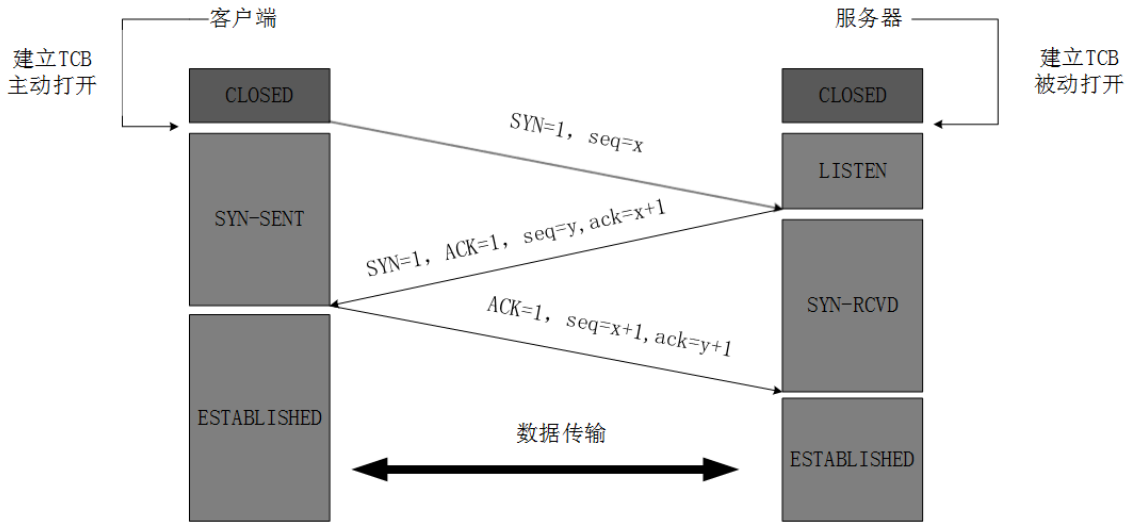
  2.在屏蔽底层协议的情况下，可以理解为TCP层直接将该数据包发往目标机器的TCP层

  3.**目标机器解包**得到数据

- **TCP连接收发数据主要做了三件事**

  1. **建立连接**
  2. **收发数据**
  3. **断开连接**

2 建立连接：TCP三次握手



- **三次握手协议过程**

> 在没进行连接的情况下，客户端的`TCP`状态处于`CLOSED`状态，服务端的`TCP`处于`CLOSED`（未开启监听）或者`LISTEN`（开启监听）状态。

1. **客户端**（通过执行`connect`函数）向服务器端发送一个`SYN`包，请求一个主动打开。该包携带客户端为这个连接请求而设定的随机数**X**作为消息序列号，此时客户端状态从`CLOSED`切换为`SYN_SENT`。

   > 这个`SYN`包可以看作是一个小数据包，不过其中没有任何实际数据，仅有诸如`TCP`首部和`TCP`选项等协议包必须数据。可以看作是客户端给服务端发送的一如果服务器端接到了客户端发的`SYN`后回了`SYN`-`ACK`后客户端掉线了，服务器端没有收到客户端回来的`ACK`，那个信号

2. **服务器端**收到一个合法的`SYN`(同步)包后，把该包放入`SYN`队列中；并返回一个针对该`SYN`包的响应包（`ACK`(确认)包）和一个新的`SYN`包。`ACK`的确认码应为**X+1**，`SYN/ACK`包本身携带一个随机产生的序号**y**，此时服务端状态通过调用socket、bind和listen函数从`LISTEN`切换为`SYN_RCVD`。
3. **客户端**收到`SYN/ACK`包后，发送一个新的`ACK`包，该包的序号被设定为**X+1**，而`ACK`的确认码则为**y+1**。然后客户端的`connect`函数成功返回，此时客户端状态从`SYN_SENT`切换至`ESTABLISHED`。当**服务器端**收到这个`ACK`包的时候，把请求帧从`SYN`队列中移出，放至`ACCEPT`队列中；这时`accept`函数如果处于阻塞状态，可以被唤醒，从`ACCEPT`队列中取出`ACK`包，重新创建一个新的用于双向通信的套接字`sockfd`，并返回，此时服务端状态从`SYN_RCVD`切换至`ESTABLISHED`。

思考

- 假如服务器端接到了客户端发的`SYN`后回了`SYN`-`ACK`后客户端掉线了

如果服务器端接到了客户端发的`SYN`后回了`SYN`-`ACK`后客户端掉线了，服务器端没有收到客户端回来的`ACK`，那么，这个连接处于一个中间状态，既没成功，也没失败。于是，服务器端如果在一定时间内没有收到的`TCP`会重发`SYN`-`ACK`。在Linux下，默认重试次数为5次，重试的间隔时间从1s开始每次都翻倍，5次的重试时间间隔为1s, 2s, 4s, 8s, 16s，总共31s，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`才会断开这个连接。使用三个`TCP`参数来调整行为：tcp_syn`ACK`_retries 减少重试次数；tcp_max_syn_b`ACK`log，增大`SYN`连接数；tcp_abort_on_overflow决定超出能力时的行为。

- 为什么TCP客户端最后还要发送一次确认呢?

  目的是防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

- **SYN攻击**

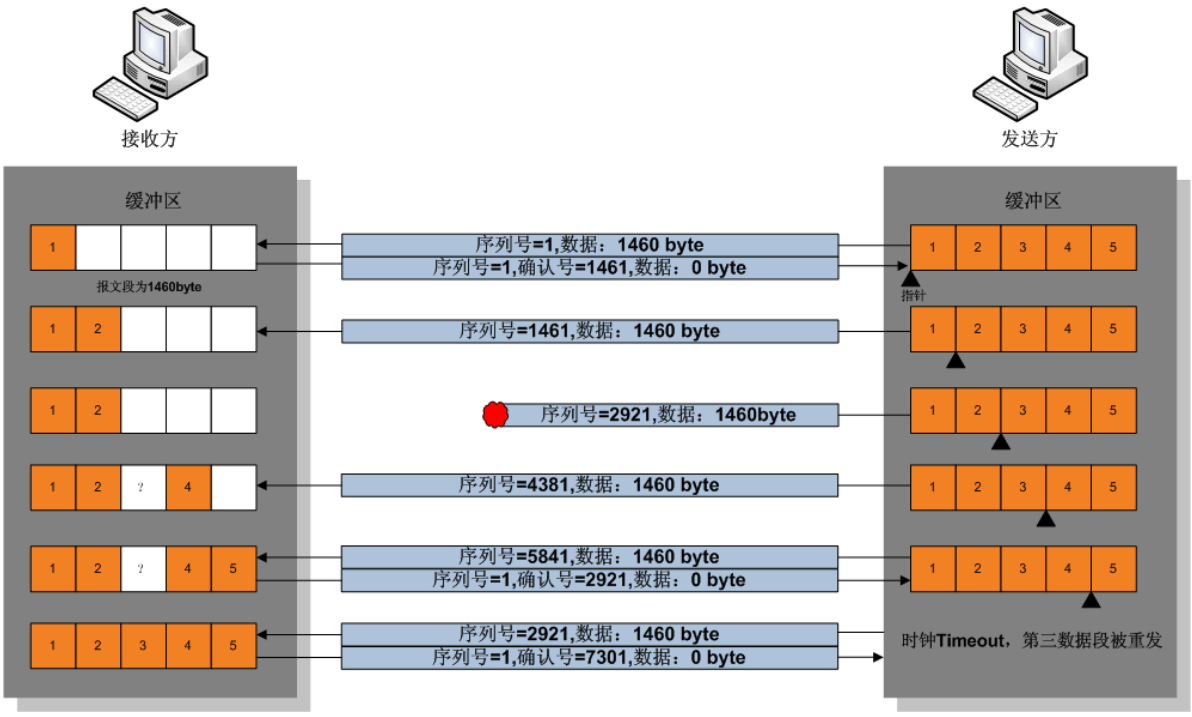
在三次握手过程中，服务器发送 SYN-ACK 之后，收到客户端的 ACK 之前的 TCP 连接称为半连接(half-open connect)。此时服务器处于 SYN_RCVD 状态。当收到 ACK 后，服务器才能转入 ESTABLISHED 状态.

SYN 攻击指的是，攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送SYN包，服务器回复确认包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。

SYN 攻击是一种典型的 DoS/DDoS 攻击。检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

3 收发数据

数据传输案例



1. 发送方首先发送第一个包含序列号为1（可变化）和1460字节数据的TCP报文段给接收方。接收方以一个没有数据的TCP报文段来回复（只含报头），用确认号1461来表示已完全收到并请求下一个报文段。

2. 发送方然后发送第二个包含序列号为1461，长度为1460字节的数据的`TCP`报文段给接收方。正常情况下，接收方以一个没有数据的`TCP`报文段来回复，用确认号2921（1461+1460）来表示已完全收到并请求下一个报文段。发送接收这样继续下去。
3. 然而当这些数据包都是相连的情况下，接收方没有必要每一次都回应。比如，**他收到第1到5条`TCP`报文段，只需回应第五条就行（累计确认）**了。在例子中第3条`TCP`报文段被丢失了，所以尽管他收到了第4和5条，然而他只能回应第2条。
4. 发送方在发送了第三条以后，没能收到回应，因此当时钟（timer）过时（expire）时，他重发第三条。（每次发送者发送一条`TCP`报文段后，都会再次启动一次时钟：RTT）。
5. 这次第三条被成功接收，接收方可以直接确认第5条，因为4，5两条已收到。
6.

累计确认

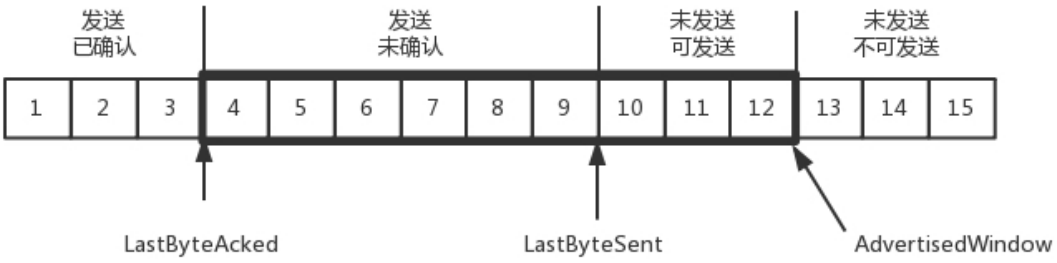
首先为了保证顺序性，每个包都有一个 ID。在建立连接的时候会商定起始 ID 是什么，然后按照 ID 一个个发送，为了保证不丢包，需要对发送的包都要进行应答，当然，这个应答不是一个一个来的，而是会应答某个之前的 ID，表示都收到了，这种模式成为**累计应答**或**累计确认**。

数据包有一下几种状态：

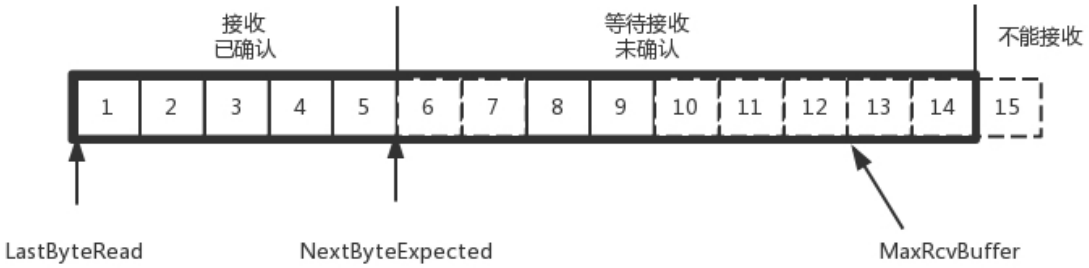
- 发送并且确认的
- 发送尚未确认的
- 没有发送等待发送的
- 没有发送并且暂时不会发送的

顺序问题和丢包问题

发送端数据结构：



接收端数据结构：



结合上面的图看，

- **在发送端**，1、2、3 已发送并确认；4、5、6、7、8、9 都是发送了还没确认；10、11、12 是还没发出的；13、14、15 是接收方没有空间，不准备发的。
- **在接收端**，1、2、3、4、5 是已经完成 ACK 但是还没读取的；6、7 是等待接收的；8、9 是已经接收还没有 ACK 的。

发送端和接收端当前的状态如下：

- 1、2、3 没有问题，双方达成了一致
- 4、5 接收方说 ACK 了，但是发送方还没收到
- 6、7、8、9 肯定都发了，但是 8、9 已经到了，6、7 没到，出现了乱序，缓存着但是没办法 ACK。

根据这个例子可以知道顺序问题和丢包问题都有可能存在，所以我们先理解**确认与重传机制**。

超时重传

发送方使用一个保守估计的时间作为收到数据包的确认的超时上限。**如果超过这个上限仍未收到确认包，发送方将重传这个数据包**。每当发送方收到确认包后，会重置这个重传定时器。一般定时器的值设定为$\mathbf{smoothed\ RTT + \max(G, 4 \times RTT\ variation)}$，其中$G$是时钟粒度。进一步，如果重传定时器被触发，仍然没有收到确认包，定时器的值将被设为前次值的二倍（直到特定阈值）。这是由于存在一类通过欺骗发送者使其重传多次，进而压垮接收者的攻击，而使用前述的定时器策略可以避免此类中间人攻击方式的拒绝服务攻击。

流量控制

> 在数据传输中，**数据流量控制**是一个能管理两个节点（数据发送方和数据接收方）之间数据传输速度的机制，由于数据发送方的发送速度和数据接收方的处理速度并不一致，数据流量控制为数据接收方提供了一种控制发送方传输速度的机制，使数据接收方节点不会被数据发送方节点发来的数据所淹没

流量控制用来避免主机分组发送得过快而使接收方来不及完全收下，一般由接收方通告给发送方进行调控，也就是**接收方在发送 ACK 的时候会带上缓冲区的窗口大小**。

`TCP` 使用滑动窗口协议实现流量控制。接收方在“接收窗口”域指出还可接收的字节数量。发送方在没有新的确认包的情况下至多发送“接收窗口”允许的字节数量。**接收方可修改“接收窗口”的值**。

当接收方宣布接收窗口的值为0，发送方停止进一步发送数据，开始了“保持定时器”（persist timer），以避免因随后的修改接收窗口的数据包丢失使连接的双侧进入死锁，发送方无法发出数据直至收到接收方修改窗口的指示。当“保持定时器”到期时，`TCP`发送方尝试恢复发送一个小的ZWP包（Zero Window Probe），期待接收方回复一个带着新的接收窗口大小的确认包。一般ZWP包会设置成3次，如果3次过后还是0的话，有的`TCP`实现就会发RST把链接断了。

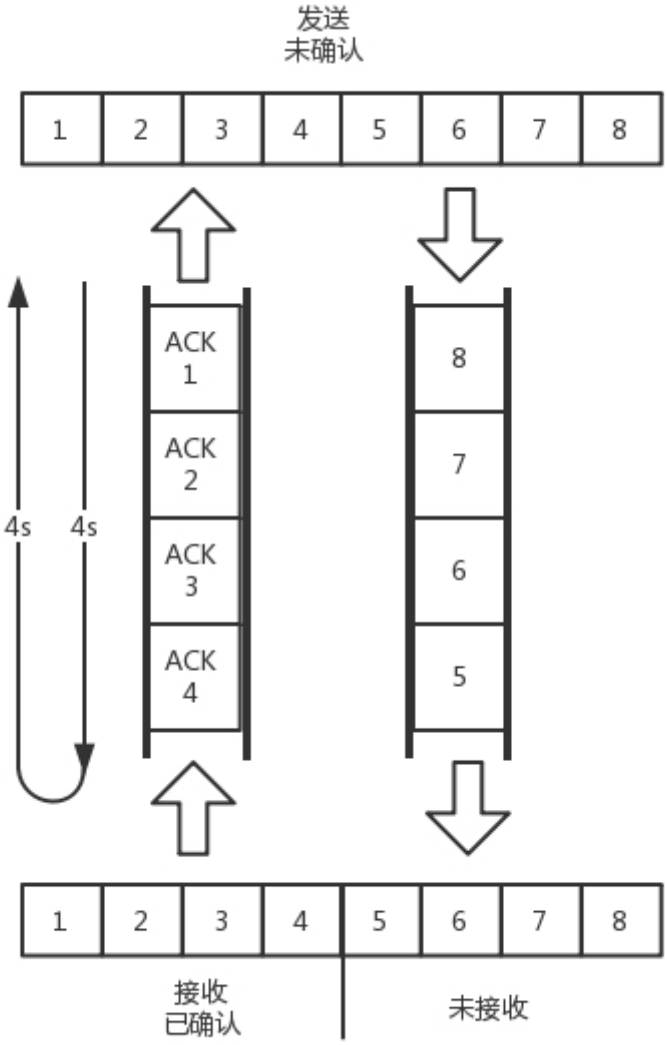
拥塞控制

> 拥塞控制是发送方根据网络的承载情况控制分组的发送量，以获取高性能又能避免拥塞崩溃（congestion collapse，网络性能下降几个数量级）。这在网络流之间产生近似最大最小公平分配。
>
> 发送方与接收方根据确认包或者包丢失的情况，以及定时器，估计网络拥塞情况，从而修改数据流的行为，这称为拥塞控制或网络拥塞避免。

拥塞控制也是通过窗口的大小来控制的，但是检测网络满不满是很难的事情，所以 `TCP` 发送包经常被比喻成往谁管理灌水，所以拥塞控制就是在不堵塞，不丢包的情况下尽可能的发挥带宽。

水管有粗细，**网络有带宽，即每秒钟能发送多少数据**；水管有长度，**端到端有时延**。理想状态下，水管里面的水 = 水管粗细 * 水管长度。对于网络上，**通道的容量 = 带宽 * 往返时延**。

如果我们设置发送窗口(**即设置每次发送包的个数**)，使得发送但未确认的包为通道的容量，就能撑满整个管道。



如图所示，假设往返时间为 8 秒，去 4 秒，回 4 秒，每秒发送一个包，已经过去了 8 秒，则 8 个包都发出去了，其中前四个已经到达接收端，但是 ACK 还没返回，不能算发送成功，5-8 后四个包还在路上，还没被接收，这个时候，管道正好撑满，在发送端，已发送未确认的 8 个包，正好等于带宽，也即每秒发送一个包，也即每秒发送一个包，乘以来回时间 8 秒。

如果在这个基础上调大窗口，使得单位时间可以发送更多的包，那么会出现接收端处理不过来，多出来的包会被丢弃，这个时候，我们可以增加一个缓存，但是缓存里面的包 4 秒内肯定达不到接收端口，它的缺点会增加时延，如果时延达到一定程度就会超时重传

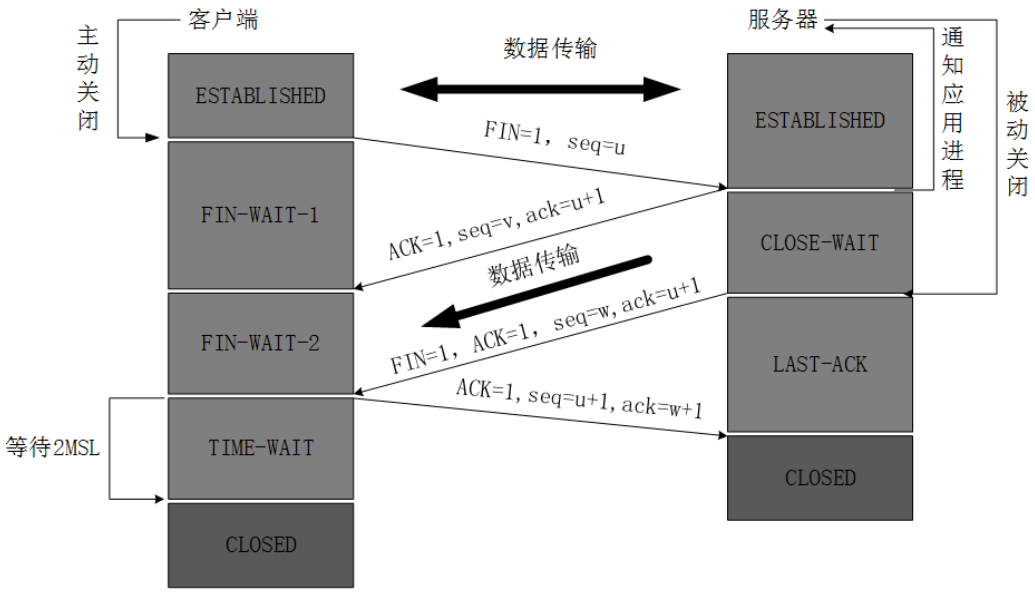
`TCP` 拥塞控制主要来避免两种现象，包丢失和超时重传，一旦出现了这些现象说明发送的太快了，要慢一点。

具体的方法就是发送端慢启动，比如倒水，刚开始倒的很慢，渐渐变快。然后设置一个阈值，当超过这个值的时候就要慢下来

慢下来还是在增长，这时候就可能水满则溢，出现拥塞，需要降低倒水的速度，等水慢慢渗下去。

拥塞的一种表现是丢包，需要超时重传，这个时候，采用快速重传算法，将当前速度变为一半。所以速度还是在比较高的值，也没有一夜回到解放前。

4 断开连接：TCP四次挥手



四次挥手协议过程

> 断开连接使用了四次挥手的过程（four-way handshake），在这个过程中连接的每一侧都独立地被终止。当一个端点要停止它这一侧的连接，就向对侧发送`FIN`，对侧回复`ACK`表示确认。因此，拆掉一侧的连接过程需要一对`FIN`和`ACK`，分别由两侧端点发出

1. **客户端**进程发出连接释放报文，并且停止发送数据。释放数据报文首部，`FIN`=1(结束)，其序列号为`seq`=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入`FIN`-WAIT-1（终止等待1）状态。`TCP`规定，`FIN`报文段即使不携带数据，也要消耗一个序号。
2. **服务器**收到连接释放报文，发出确认报文，`ACK`=1，`ACK`=u+1，并且带上自己的序列号`seq`=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。`TCP`服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于**半关闭状态**，即**客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受**。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。
3. **客户端**收到服务器的确认请求后，此时，客户端就进入`FIN-WAIT`-2（终止等待2）状态，**等待服务器发送连接释放报文**（在这之前还需要接受服务器发送的最后的数据）。服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，`FIN`=1，`ACK`=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为`seq`=w，此时，服务器就进入了`LAST-ACK`（最后确认）状态，等待客户端的确认。
4. **客户端**收到服务器的连接释放报文后，必须发出确认，`ACK`=1，`ACK`=w+1，而自己的序列号是`seq`=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时`TCP`连接还没有释放，**必须经过2*MSL（最长报文段寿命，Linux设置成了30s）的时间**后，当客户端撤销相应的TCB后，才进入CLOSED状态。服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销`T CB`后，就结束了这次的`TCP`连接。可以看到，**服务器结束`TCP`连接的时间要比客户端早一些**。

思考

何为TCB?

主机收到一个`TCP`包时，用两端的IP地址与端口号来标识这个`TCP`包属于哪个session。使用一张表来存储所有的session，表中的每条称作Transmission Control Block（**TCB**），TCB结构的定义包括连接使用的源端口、目的端口、目的ip、序号、应答序号、对方窗口大小、己方窗口大小、tcp状态、tcp输入/输出队列、应用层输出队列、tcp的重传有关变量等。

为什么客户端最后还要等待2*MSL(即TIME_WAIT状态)?

MSL（Maximum Segment Lifetime 最长分段生命周期），`TCP`允许不同的实现可以设置不同的MSL值。

第一，保证客户端发送的最后一个`ACK`报文能够到达服务器，因为这个`ACK`报文可能丢失，站在服务器的角度看来，我已经发送了`FIN`+`ACK`报文请求断开了，客户端还没有给我回应，应该是我发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个2MSL时间段内收到这个重传的报文，接着给出回应报文，并且会重启2MSL计时器。

第二，防止类似与“三次握手”中提到了的“已经失效的连接请求报文段以前一个连接的化身”出现在本连接中。客户端发送完最后一个确认报文后，在这个2MSL时间中，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。

为什么建立连接是三次握手，关闭连接确是四次挥手呢?

建立连接的时候， 服务器在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，服务器收到对方的`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，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送`FIN`报文给对方来表示同意现在关闭连接，因此，己方ACK和`FIN`一般都会分开发送，从而导致多了一次。

如果已经建立了连接，但是客户端突然出现故障了怎么办?

`TCP`还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

5 状态编码

下表为`TCP`状态码列表，以**S**指代服务器，**C**指代客户端，**S&C**表示两者，**S/C**表示两者之一

- **LISTEN S**

  服务器等待从任意远程`TCP`端口的连接请求。侦听状态。

- **`SYN`-SENT C**

  客户在发送连接请求后等待匹配的连接请求。通过connect()函数向服务器发出一个同步（SYNC）信号后进入此状态。

- **`SYN`-RECEIVED S**

  服务器已经收到并发送同步（SYNC）信号之后等待确认（ACK）请求。

- **ESTABLISHED S&C**

  服务器与客户的连接已经打开，收到的数据可以发送给用户。数据传输步骤的正常情况。此时连接两端是平等的。这称作全连接。

- **`FIN`-WAIT-1 S&C**

（服务器或客户）主动关闭端调用close（）函数发出`FIN`请求包，表示本方的数据发送全部结束，等待`TCP`连接另一端的ACK确认包或`FIN`&ACK请求包。

- **`FIN`-WAIT-2 S&C**

  主动关闭端在`FIN`-WAIT-1状态下收到ACK确认包，进入等待远程`TCP`的连接终止请求的半关闭状态。这时可以接收数据，但不再发送数据。

- **CLOSE-WAIT S&C**

  被动关闭端接到`FIN`后，就发出ACK以回应`FIN`请求，并进入等待本地用户的连接终止请求的半关闭状态。这时可以发送数据，但不再接收数据。

- **CLOSING S&C**

  在发出`FIN`后，又收到对方发来的`FIN`后，进入等待对方对己方的连接终止（`FIN`）的确认（ACK）的状态。少见。

- **LAST-ACK S&C**

  被动关闭端全部数据发送完成之后，向主动关闭端发送`FIN`，进入等待确认包的状态。

- **TIME-WAIT S/C**

  主动关闭端接收到`FIN`后，就发送ACK包，等待足够时间以确保被动关闭端收到了终止请求的确认包。（按照RFC 793，一个连接可以在TIME-WAIT保证最大四分钟，即最大分段寿命（maximum segment lifetime）的2倍）

- **CLOSED S&C**

  完全没有连接

6 标志位

- `URG`:它为了标志紧急指针是否有效。
- `ACK`： 标识确认号是否有效。
- `PSH`:提示接收端应用程序立即将接收缓冲区的数据拿走。
- `RST`： 它是为了处理异常连接的， 告诉连接不一致的一方，我们的连接还没有建立好, 要求对方重新建立连接。我们把携带`RST`标识的称为复位报文段。
- `SYN`: 请求建立连接; 我们把携带`SYN`标识的称为同步报文段。
- `FIN`:通知对方, 本端要关闭连接了, 我们称携带FIN标识的为结束报文段。

二 UDP协议

1 UDP简介

User Datagram Protocol，缩写：**UDP**；又称**用户资料包协议**，是一个**无连接**的简单的**面向数据报**的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是**并不能保证它们能到达目的地**。由于UDP在传输数据报前**不用在客户和服务器之间建立一个连接**，且**没有超时重发**等机制，故而传输速度很快。

2 特性

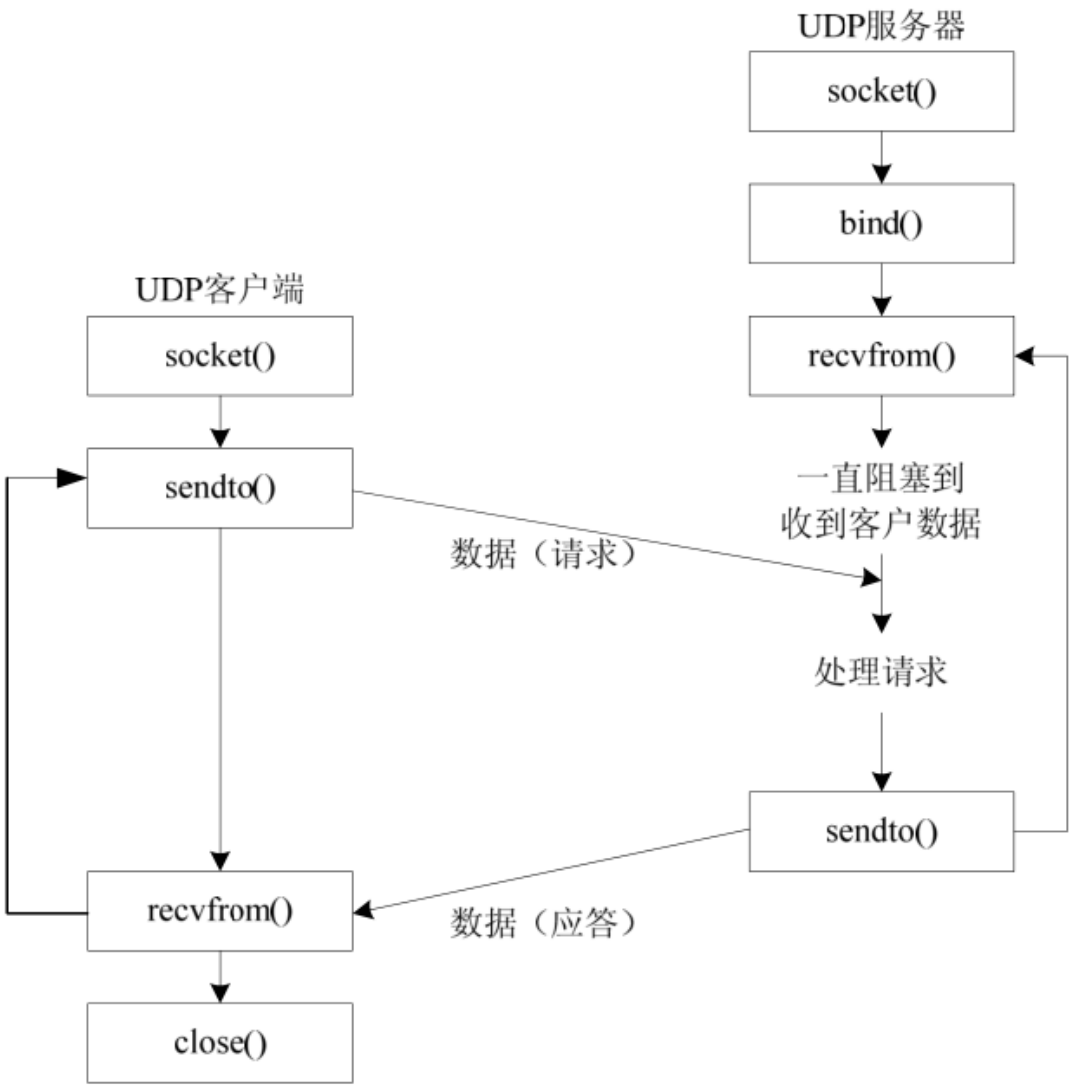
- UDP 缺乏可靠性。UDP 本身不提供确认，序列号，超时重传等机制。UDP 数据报可能在网络中被复制，被重新排序。即 UDP 不保证数据报会到达其最终目的地，也不保证各个数据报的先后顺序，也不保证每个数据报只到达一次

- UDP 数据报是有长度的。每个 UDP 数据报都有长度，如果一个数据报正确地到达目的地，那么该数据报的长度将随数据一起传递给接收方。而 `TCP` 是一个字节流协议，没有任何（协议上的）记录边界。
- UDP 是无连接的。UDP 客户和服务器之前不必存在长期的关系。UDP 发送数据报之前也不需要经过握手创建连接的过程。
- UDP 支持多播和广播。

3 UDP网络程序发送数据流程

客户端

1. 创建客户端套接字
2. 发送/接收数据
3. 关闭套接字



案例：https://www.geeksforgeeks.org/udp-server-client-implementation-c/

三 Socket编程

1 基础知识

何为套接字?

- 标识每个端点的两个值(**`IP`地址和端口号**)通常成为一个**套接字**
- 一个TCP连接的套接字对是一个定义该连接两端的四元组：本地（外地）IP地址、本地（外地）TCP端口号
- **网络中的进程之间如何通信?**
    - **如何在网络中标识唯一进程?**

`TCP`/IP协议族已经帮我们解决了这个问题，网络层的**“ip地址”**可以唯一标识网络中的主机，而传输层的**“协议+端口”**可以唯一标识主机中的应用程序（进程）。这样利用三元组（ip地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互。

使用`TCP`/IP协议的应用程序通常采用应用编程接口：UNIX BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，网络中进程通信是无处不在，这就是我为什么说“一切皆socket”。

Socket是什么?

Socket 起源于 Unix，Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开(open) –> 读写(write/read) –> 关闭(close)”模式来进行操作。因此 Socket 也被处理为一种特殊的文件。

Socket的干了什么?

Socket 是对 `TCP`/IP 协议族的一种封装，是应用层与`TCP`/IP协议族通信的中间软件抽象层。从设计模式的角度看来，Socket其实就是一个门面模式，它把复杂的`TCP`/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

2 Socket API

socket()函数

```
 #include <sys/socket.h>
int socket(int family, int type, int protocol);
```

socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

family参数

该参数指明要创建的sockfd的协议族，一般比较常用的有两个：

- `AF_INET`：IPv4协议族
- `AF_INET6`：IPv6协议族

type参数

该参数用于指明套接字类型，具体有：

- `SOCK_STREAM`：**字节流套接字**，适用于`TCP`或SCTP协议

- `SOCK_DGRAM`：**数据报套接字**，适用于UDP协议
- `SOCK_SEQPACKET`：有序分组套接字，适用于SCTP协议
- `SOCK_RAW`：原始套接字，适用于绕过传输层直接与网络层协议（IPv4/IPv6）通信

protocol参数

该参数用于指定协议类型。

如果是`TCP`协议的话就填写`IPPROTO_TCP`，UDP和SCTP协议类似。

也可以直接填写0，这样的话则会默认使用`family`参数和`type`参数组合制定的默认协议

（参照上面type参数的适用协议）

- bind()函数
- listen()、connect()函数
- accept()函数
- read()、write()函数等
- close()函数

返回值

`socket`函数在成功时会返回套接字描述符，失败则返回-1。

失败的时候可以通过输出`errno`来详细查看具体错误类型。通常一个内核函数运行出错的时候，它会定义全局变量`errno`并赋值。

当我们引入`errno.h`头文件时便可以使用这个变量。并利用这个变量查看具体出错原因。

- 借助`strerror()`函数，使用`strerror(errno)`得到一个具体描述其错误的字符串。

> **注意**
>
> 1. 并不是上面的type和protocol可以随意组合的，如SOCK_STREAM不可以跟IPPROTO_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。
> 2. 当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则当调用connect()、listen()时系统会自动随机分配一个端口

bind()函数

> bind()函数把一个地址族中的特定地址赋给socket。例如对应`AF_INET`、`AF_INET6`就是把一个`ipv4`或`ipv6`地址和端口号组合赋给socket.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);//返回：若成功则为0，若出错则为-1
```

其中**第一个参数**为监听套接字，它是由socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。

第二个参数指向要绑定给`sockfd`的协议地址。这个地址结构根据地址创建`socket`时的地址协议族的不同而不同(有ipv4、ipv6等差别)

```
//例如IPV4套接字结构体定义
```

```c
#include <netinet/in.h>
struct in_addr
{
    in_addr_t       s_addr;         // 32位IPv4地址
};
struct sockaddr_in
{
    uint8_t         sin_len;        // 结构长度，非必需
    sa_family_t     sin_family;     // 地址族，一般为AF_****格式，常用的是AF_INET
    in_port_t       sin_port;       // 16位`TCP`或UDP端口号
    struct in_addr  sin_addr;       // 32位IPv4地址
    char            sin_zero[8];    // 保留数据段，一般置零
};
```

上述地址协议族需要赋值有

- `sin_family`
- `sin_addr`
- `sin_port`

第三个参数为该地址结构的长度

返回值

若成功则返回0，否则返回-1并置相应的`errno`。

比较常见的错误是错误码`EADDRINUSE`（"Address already in use"，地址已使用）。

实例

```c
#define DEFAULT_PORT 16555
struct sockaddr_in servaddr;    // 定义一个IPv4套接字地址结构体
bzero(&servaddr, sizeof(servaddr));    // 将该结构体的所有数据置零
servaddr.sin_family = AF_INET;    // 指定其协议族为IPv4协议族
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);    // 指定IP地址为通配地址，此时就交由内核选择IP地址绑定,那服务器如果有多个网络接口，服务器进程就可以在任意网络接口上接受客户连接
servaddr.sin_port = htons(DEFAULT_PORT);    // 指定端口号为16555
// 调用bind，注意第二个参数使用了类型转换，第三个参数直接取其sizeof即可
if (-1 == bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)))
{
    printf("Bind error(%d): %s\n", errno, strerror(errno));
    return -1;
}
```

> 注意
>
> 1. 为什么要有`(struct sockaddr*)&servaddr`这一个步骤?
>
>   从内核的角度来看，内核必须取调用者的指针，把他的类型强制转换成`struct sockaddr*`类型，然后检查其中的`sa_family`字段的值才能确定这个结构体的真实类型。
> 2. 通常**服务器在启动的时候都会绑定一个众所周知的地址**（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，**由系统自动分配一个端口号和自身的ip地址组合**。这就是为什么通常服务器端在listen之前会调用bind()，而**客户端就不会调用，而是在connect()时由系统随机生成一个**

> 3. 实际案例中指定端口号的函数`htonl()`,和指定ip函数`htons()`，涉及到**字节排序函数**这个概念

字节排序函数

- **网络字节序与主机字节序**

主机字节序就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

网络字节序：4个字节的32 bit值以下面的次序传输：首先是`0～7bit`，其次`8～15bit`，然后`16～23bit`，最后是`24~31bit`。这种传输次序称作大端字节序。**由于`TCP`/`IP`首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序**。字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：这其实是很常规的**统一标准中间件**的做法,我们需要做的就是调用适当的函数去交换给定的某个定值，而不去管这个值是大端或是小端。

在Linux中，位于`<netinet/in.h>`中有用于主机字节序和网络字节序之间相互转换的函数：

```
#include <netinet/in.h>
传递规则：
  1.机子A先将变量由自身的字节序转换为网络字节序
      uint16_t htons(uint16_t host16bitvalue);     //返回网络字节序的值
  2.发送转换后的数据
  3.机子B接到转换后的数据之后，再将其由网络字节序转换为自己的字节序
        uint16_t ntohs(uint16_t net16bitvalue);      //返回主机字节序的值

// h代表host、n代表network、s代表short、l代表long
```

字符操纵函数

操作多字节字段有两组，他们既不对数据做解释，也不假设数据是以空字符串结束的C字符串。

- 第一组是源自Berkeley

```
#include <string.h>
//将目标字节串中指定数目的字节置位0
void bzero(void *dest,size_t nbytes);
//此外还有bcopy(),bcmp(), 不过这两个用得不多
```

- 第二组是ANSI 给出的C函数

```
#include<string.h>
//把目标字节串指定数目的字节置位值c
void *memset(void *dest, int c, size_t len);
//将指定数目的字节从源字节(*src)转移到目标字节(*dest)
void *memcpy(void *dest, const void *src, size_t nbytes);
//比较任意两个字节串，相同则返回0，否则返回非0值，其正负有二者的第一个字符比较决定
int memcmp(const void *ptl, const void *ptr2, size_t nbytes)
```

listen()函数

> listen()函数把一个未连接的套接字转换成一个被动套接字，指示内核应该接受指向该套接字的连接请求，调用该函数后，套接字从closed状态转换为Listen状态
>
> 通俗理解就是服务器调用socket函数生成一个未绑定的主动套接字，listen()函数将此此套接字与由客户端调用connect()函数传过来的客户端套接字相绑定，然后存入队列中，等待TCP的三次握手(此处可以参考上文关于TCP三次握手的图示)。

该函数的原型如下：

```
#include <sys/socket.h>
int listen(int sockfd, int backlog); //返回：若成功则为0，若出错则为-1
```

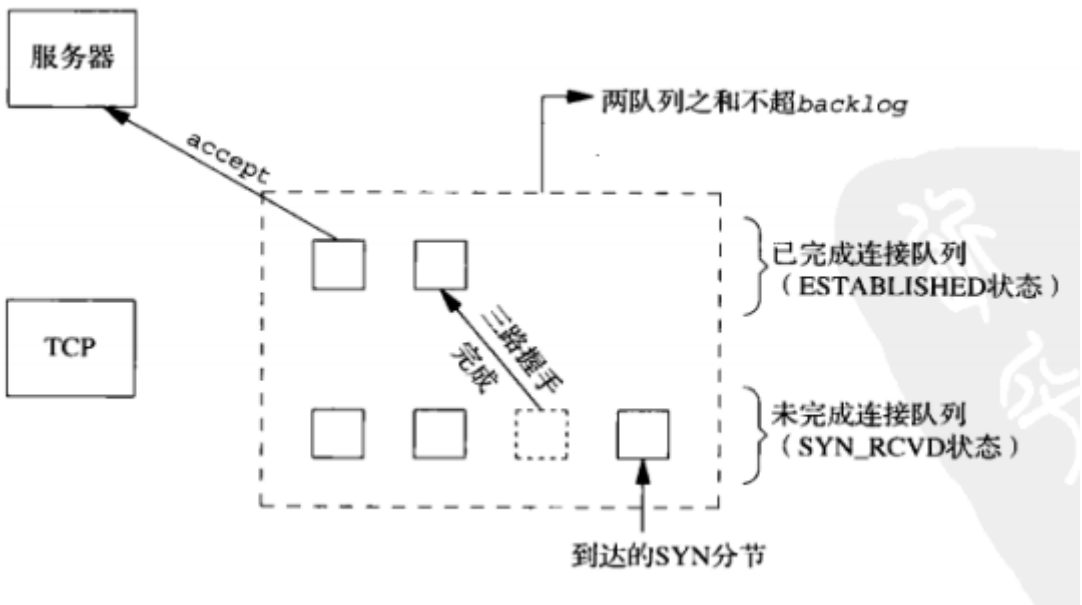
本函数通常在调用socket和bind这两个函数之后，并在accept函数之间调用。

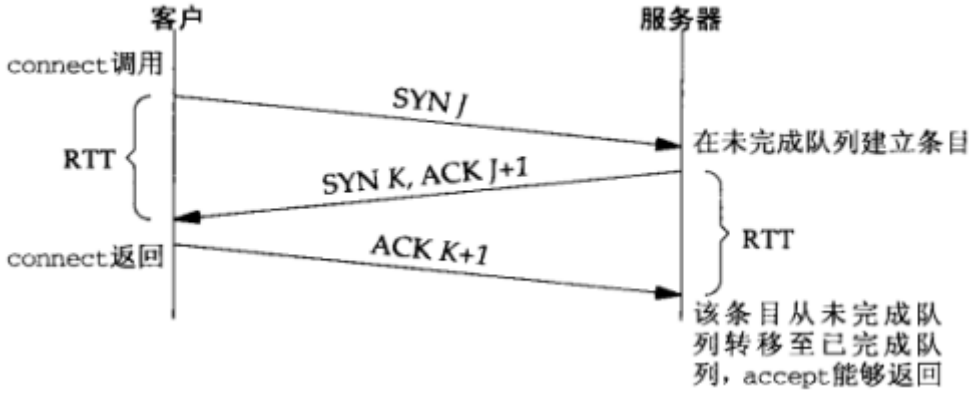
函数参数

`sockfd`为服务器套接字

`backlog`为TCP为监听套接字维护的两个队列的个数之和

- 未完成连接队列
  - 每个有客户端发过来的SYN都会保存到这个队列中，然后等待服务器完成TCP三次握手，此时这些套接字处于`SYN_RCVD`状态
- 已完成连接队列
  - 每个已完成的TCP三次握手的客户都会存入该项中，这些套接字处于`ESTABLISHED`状态





上图描述的是，当来自客户端的SYN到达时，TCP在未完成连接队列中创建一个新的成员，然后响应TCP三次握手中的第二次握手(服务器的SYN相应，其中附带有对客户端SYN的ACK)，这一项一直保留在未完成连接队列中，直到TCP三次握手中的第三次握手(客户端对服务器SYN的ACK)到达或者该项超时。如果第三次握手正常完成，该项就从未完成队列移动到已完成连接队列，当调用进程`accpet()`时已完成队列中的对首返回给该进程，或该队列为空，那么进程将被投入睡眠直到已完成队列中有对象。

connect()函数

> connect()函数用于客户端跟绑定了指定的ip和port并且处于`LISTEN`状态的服务端进行连接。
>
> 在调用connect函数的时候，调用方（也就是客户端）便会主动发起`TCP`三次握手。

该函数的原型如下：

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

其中**第一个参数**为客户端套接字，**第二个参数**为用于指定服务端的ip和port的套接字地址结构体，**第三个参数**为该结构体的长度。

返回值

- 若成功则返回0，否则返回-1并置相应的`errno`
- connect函数出现错误的几种情况
  - 若客户端在发送SYN包之后长时间没有收到响应，则返回`ETIMEOUT`错误
  - 若客户端在发送SYN包之后收到的是RST包的话，则会立刻返回`ECONNREFUSED`错误
  - 若客户端在发送SYN包的时候在中间的某一台路由器上发生ICMP错误，则会发生`EHOSTUNREACH`或`ENETUNREACH`错误
- 由于connect函数在发送SYN包之后就会将自身的套接字从`CLOSED`状态置为`SYN_SENT`状态，故当connect报错之后需要主动将套接字状态置回`CLOSED`。此时需要通过**调用close函数主动关闭套接字实现**。

```
if (-1 == connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)))
{
    printf("Connect error(%d): %s\n", errno, strerror(errno));
    close(sockfd);        // 新增代码，当connect出错时需要关闭套接字
    return -1;
}
```

操作上比较类似于服务端使用bind函数（虽然做的事情完全不一样），唯一的区别在于指定ip这块。**服务端调用bind函数的时候既可以指定地址或端口，也可以都不指定，但客户端调用connect函数的时候则需要指定服务端的`ip`，但客户端的端口号常由内核为其指定一个临时端口**

> 值得注意的是：
>
> 客户端的端口是临时的，而服务端的端口是未指定，那该怎么让客户端与服务端在同一个端口号进行通讯呢？
>
> 答案是通过远程过程调用服务器(`RPC`),客户在connect这些服务器之前，必须与端口映射器联系以获得他们的临时端口，这种情况也适合使用UPD的RPC服务器。

在客户端的代码中，令套接字地址结构体指定ip的代码如下：

```
inet_pton(AF_INET, SERVER_IP, &servaddr.sin_addr);
```

这个就涉及到**ip地址的表达格式与数值格式相互转换**的函数。

IP地址格式转换函数

- IP地址一共有两种格式
  - 表达格式：也就是我们能看得懂的格式，例如`"192.168.19.12"`这样的字符串
  - 数值格式：可以存入套接字地址结构体的格式，数据类型为整型

显然，当我们需要**将一个IP赋进套接字地址结构体中，就需要将其转换为数值格式**。

在`<arpa/inet.h>`中提供了两个函数用于IP地址格式的相互转换：

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr); //返回：若字符串有效则为1，否则为0
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len); //返回：指向一个点分十进制数串的指针

//函数名中：p表示presentation(表达式)、n表示numeric(数值)
//地址的表达式的格式为ASSII字符串，数值格式则是存放到套接字地址结构中的二进制值
```

其中：

- `inet_pton()`函数用于将`IP`**地址从表达格式转换为数值格式**
  - 第一个参数指定协议族（`AF_INET`或`AF_INET6`）
  - 第二个参数指定要转换的表达格式的IP地址
  - 第三个参数指定用于存储转换结果的指针
  - 对于返回结果而言：
    - 若转换成功则返回1
    - 若表达格式的IP地址格式有误则返回0
    - 若出错则返回-1
- `inet_ntop()`函数用于将`IP`**地址从数值格式转换为表达格式**
  - 第一个参数指定协议族
  - 第二个参数指定要转换的数值格式的IP地址
  - 第三个参数指定用于存储转换结果的指针

- 第四个参数指定第三个参数指向的空间的大小，用于防止缓存区溢出
  - 第四个参数可以使用预设的变量

```
#include <netinet/in.h>
#define INET_ADDRSTRLEN  16   // IPv4地址的表达格式的长度
#define INET6_ADDRSTRLEN 46     // IPv6地址的表达格式的长度
```

- 对于返回结果而言
  - 若转换成功则返回指向返回结果的指针
  - 若出错则返回NULL

accept()函数

> accept()函数由服务器调用，用于从已完成连接队列对首返回下一个已完成连接。如果已完成连接队列为空，那么进程投入休眠(假设套接字默认为阻塞方式)
>
> 更准确的说，accept函数由`TCP`服务器调用，用于从**Accept队列**中pop出一个已完成的连接。若Accept队列为空，则accept函数所在的进程阻塞。

`TCP`服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址。`TCP`客户端依次调用socket()、connect()之后就向`TCP`服务器发送了一个连接请求。`TCP`服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好。之后就可以开始网络I/O操作，即类同于普通文件的读写I/O操作。

该函数的原型如下：

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen); //返回：若成功则为非负描述符，若出错则为-1
```

其中**第一个参数**为服务端自身的套接字，**第二个参数**用于接收客户端的套接字地址结构体，**第三个参数**在调用前为`cliaddr`所指的套接字地址结构的长度，返回时该数值为由内核存放在该套接字地址结构内的确切字节数。**第二三个参数都可以置位空指针，表明我们对客户的身份不感兴趣**。

返回值

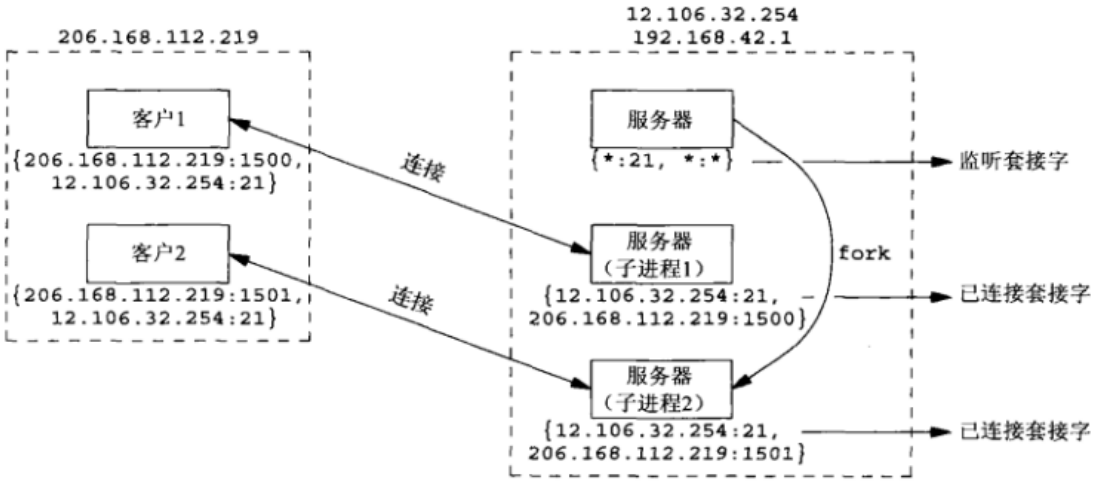
当accept函数成功从已完成连接队列中拿到一个已完成连接时，其返回值是由内核自动生成的一个**全新的客户端套接字描述符**，代表与返回客户的`TCP`连接，用于后续的数据传输。

若发生错误则返回-1并置相应的`errno`。

accept()返回的socket描述符和listen描述符是一样的吗?

一个socket是由一个五元组来唯一标示的，即(协议，server_ip, server_port, client_ip, client_port)。只要该五元组中任何一个值不同，则其代表的socket就不同。这里忽略协议的区别，在同一协议的基础上，服务器端的listen socket的端口可以看成(server_ip, server_port, ***, ***),其中***是通配符，它跟任何一个client_ip, client_port值都不同，可以简单看成是（0,0）对，当然实现不是这样的。这样在服务器端accept之后，返回的连接socket的四元组就是(server_ip, server_port, client_ip, client_port)，这里的client_ip，client_port因连接的客户端的不同而不同。所以accept返回的socket和listen socket是不同的，不同之处就在于四元组中的客户端ip和port，而服务器端的server_ip和server_port还是相同的，也就是accpet()函数返回的新的socket描述符的端口和listen端口是一样的。可以使用getsockname()函数来查看它们之间的不同。

> 注意
>
> accept的**第一个参数为服务器的socket描述符**，是服务器开始调用socket()函数生成的，称为**监听socket描述字**；而**accept函数返回的是已连接套接字**。一个服务器通常通常仅仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，**当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭**。



recv()函数 & send()函数

> 通过前面的步骤，服务器跟客户端已经建立好连接，可以调用网络I/O进行读写造作(即网络中不同的进程之间的通讯)，recv函数用于通过套接字接收数据，send函数用于通过套接字发送数据.

网络I/O操作有下面几组:

```
read()/write()
readv()/writev()

recv()/send()
recvmsg()/sendmsg()
recvfrom()/sendto()
```

recv()/send()这两个函数的原型如下:

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

其中:

- 第一个参数为要读写的套接字
- 第二个参数指定要接收数据的空间的指针（recv）或要发送的数据（send）
- 第三个参数指定最大读取的字节数（recv）或发送的数据的大小（send）
- 第四个参数用于设置一些参数，默认为0，`flags` 一般设置为0，此时send为阻塞式发送，即发送不成功会一直阻塞，直到被某个信号终端终止，或者直到发送成功为止。
    - 指定`MSG_NOSIGNAL`，表示当连接被关闭时不会产生`SIGPIPE`信号

- 指定`MSG_DONTWAIT`表示非阻塞发送指定`MSG_DONTWAIT`表示非阻塞发送,如果数据不能立刻传输，该调用不会阻塞，而是调用失败，伴随的错误代码为`EAGAIN`
- 指定`MSG_OOB`表示带外数据

事实上，去掉第四个参数的情况下，recv跟read函数类似，send跟write函数类似。这两个函数的本质也是一种通过描述符进行的IO，只是在这里的描述符为套接字描述符。

返回值

在recv函数中：

- 若成功，则返回所读取到的字节数(recv只是单纯的拷贝数据，不作处理)
- 否则返回-1，置`errno`

在send函数中：

- 若成功，则返回成功写入的字节数
- 事实上，当返回值与`nbytes`不等时，也可以认为其出错。
- 否则返回-1，置`errno`

close()函数

> 在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，close()函数用于断开连接。或者更具体的讲，该函数用于关闭套接字，并终止`TCP`连接。

该函数的原型如下：

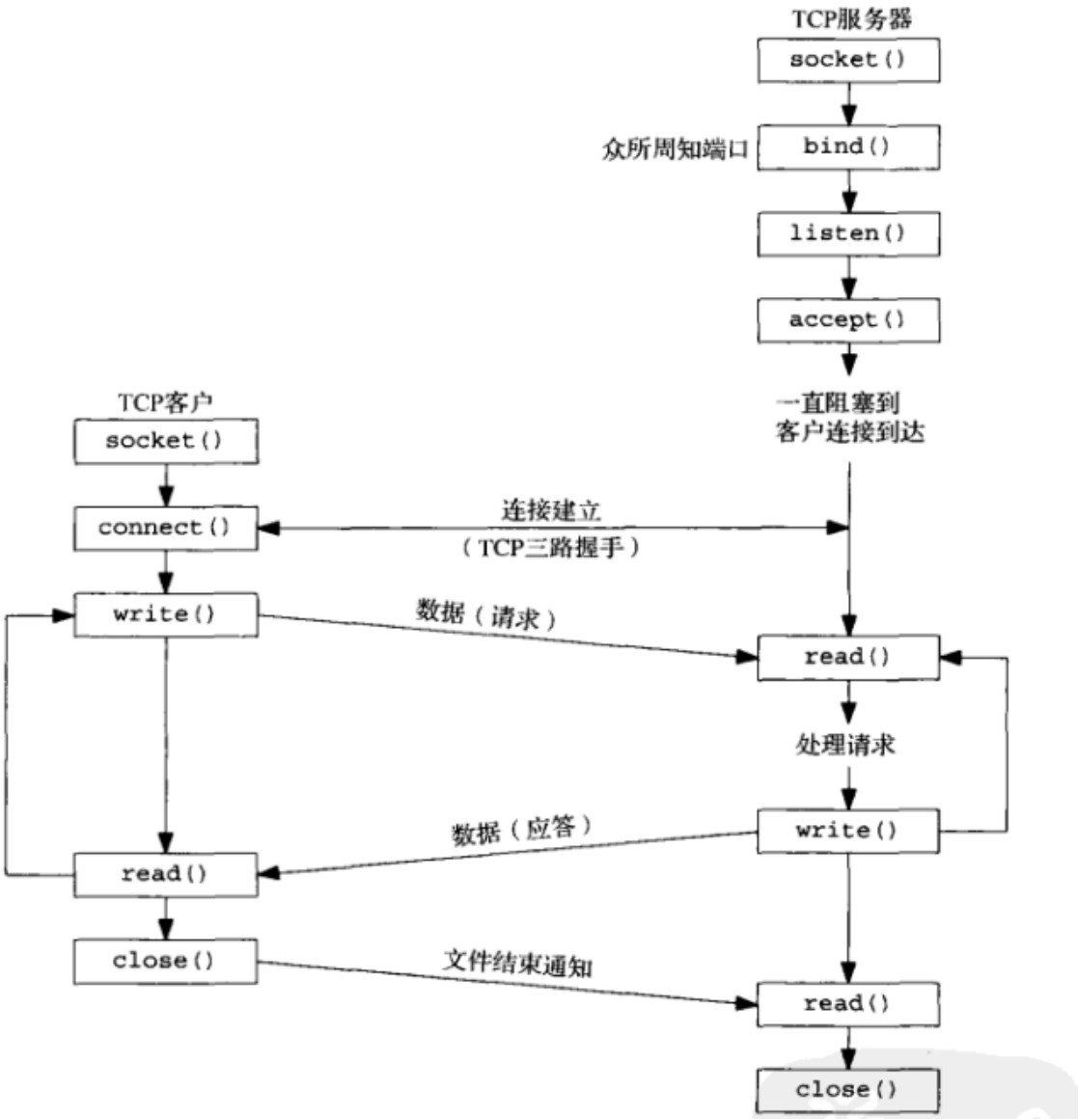
```
#include <unistd.h>
int close(int sockfd);
```

返回值

同样的，若close成功则返回0，否则返回-1并置`errno`。

常见的错误为**关闭一个无效的套接字**。

TCP客户端/服务端的套接字使用流程



四 Socket缓冲区

1 什么是socket缓冲区?

网络编程的时候要跟某个IP建立联系，需要调用系统提供的socket API.**socket** 在操作系统层面，可以理解为一个**文件**。我们可以对这个文件进行一些**方法操作**。

- 用`listen`方法，可以让程序作为服务器**监听**其他客户端的连接。
- 用`connect`，可以作为客户端**连接**服务器。
- 用`send`或`write`可以**发送**数据，`recv`或`read`可以**接收**数据。

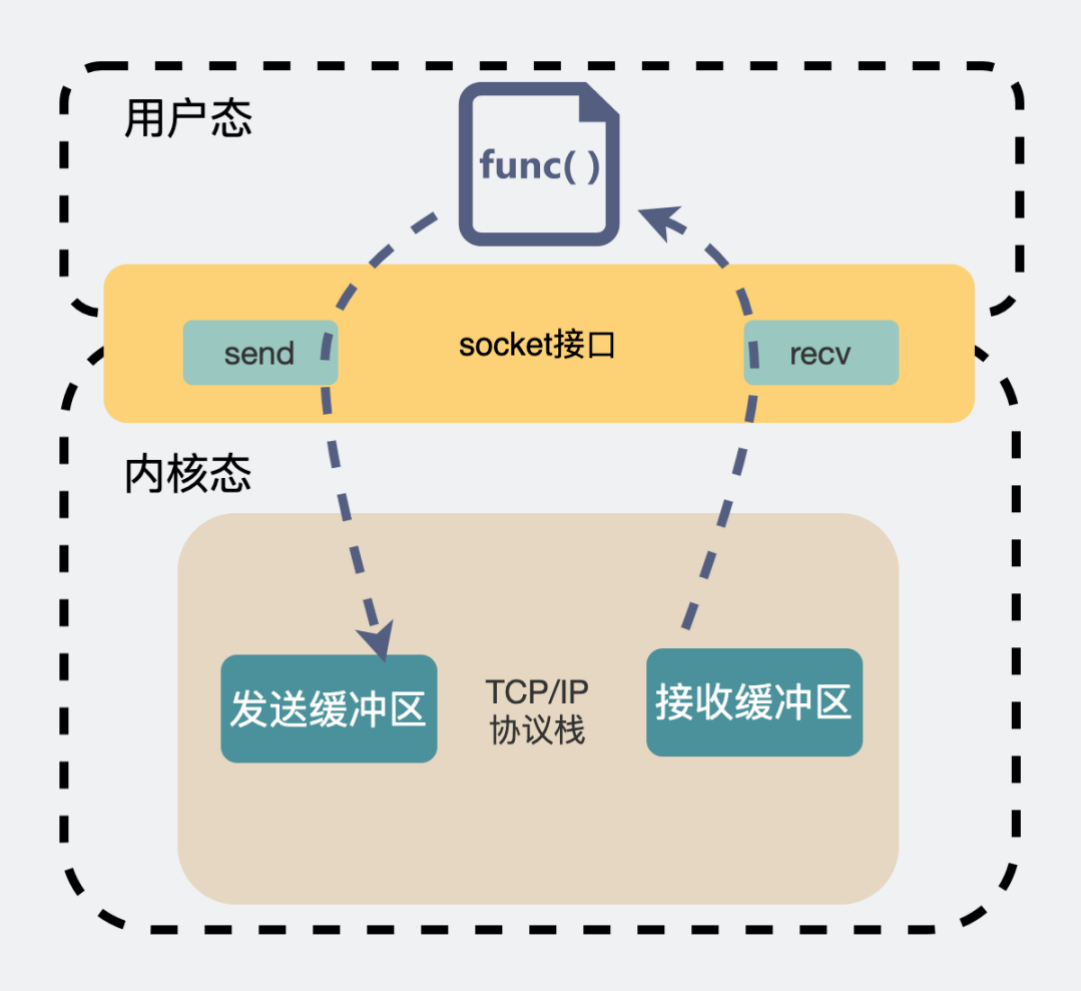
在建立好连接之后，这个 **socket** 文件就像是远端机器的 **"代理人"** 一样。比如，如果我们想给远端服务发东西，那就只需要对这个文件执行写操作就行了。

那写到了这个文件之后，剩下的发送工作自然就是由操作系统**内核**来完成了。既然是写给操作系统，那操作系统就需要**提供一个地方给用户写**。同理，接收消息也是一样。

这个地方就是 **socket 缓冲区**。

- 用户**发送**消息的时候写给 send buffer（发送缓冲区）
- 用户**接收**消息的时候写给 recv buffer（接收缓冲区）

也就是说一个**socket，会带有两个缓冲区**，一个用于发送，一个用于接收。因为这是个先进先出的结构，有时候也叫它们**发送、接收队列**。

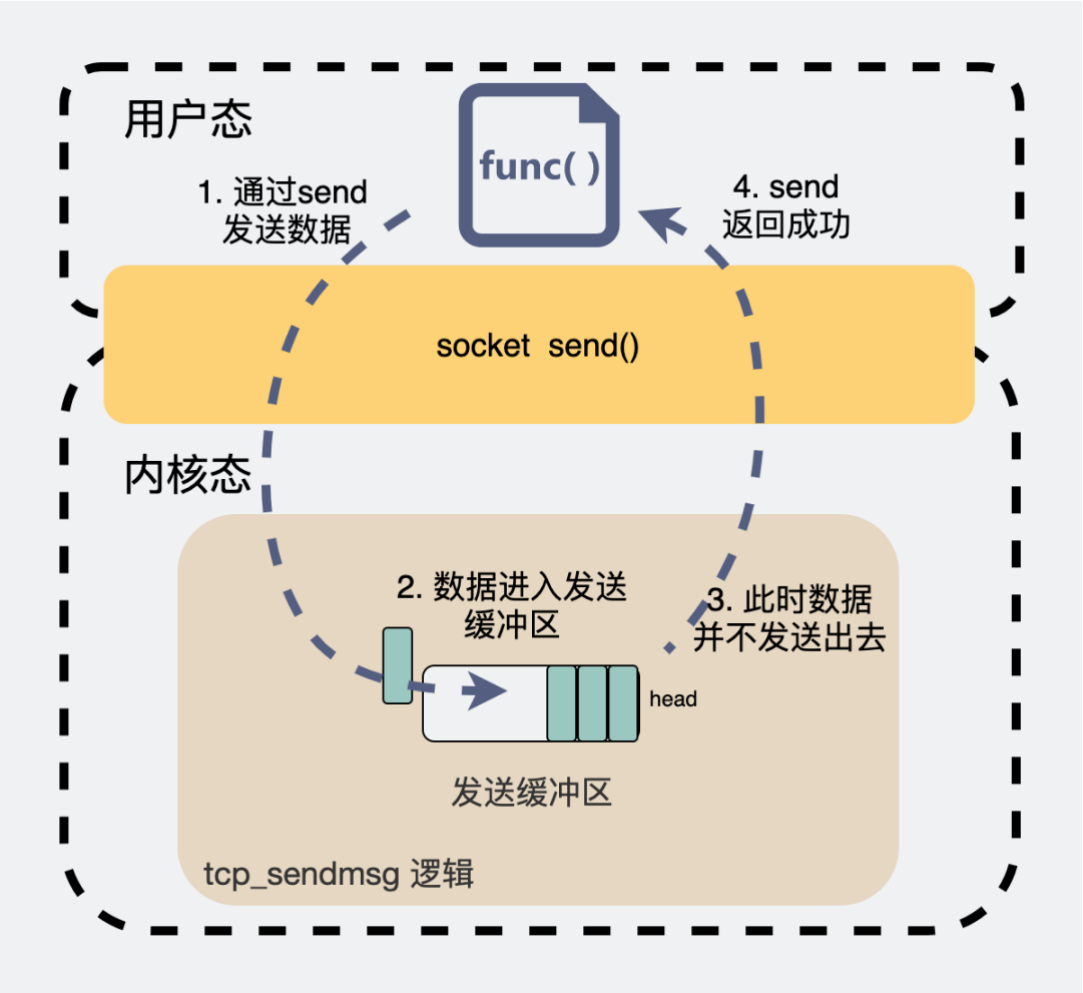


> Tips:
>
> 可以通过`netstat -nt`命令来查看socket缓冲区
>
> ```
> # netstat -nt
> Active Internet connections (w/o servers)
> Proto Recv-Q Send-Q Local Address           Foreign Address         State
> tcp        0     60 172.22.66.69:22         122.14.220.252:59889
> ESTABLISHED
> ```
>
> 这上面表明了，这里有一个协议（Proto）类型为 TCP 的连接，同时还有本地（Local Address）和远端（Foreign Address）的IP信息，状态（State）是已连接。
>
> 还有Send-Q 是发送缓冲区，下面的数字60是指，当前还有60 Byte在发送缓冲区中未发送。而Recv-Q 代表接收缓冲区，此时是空的，数据都被应用进程接收干净了

2 TCP部分

执行send()后会立即发送数据吗?

答案是不确定的，在建立连接之后，执行 send ，数据只是拷贝到了socket 缓冲区。至 于什么时候会发数据，发多少数据，全听操作系统安排。



在用户进程中，程序通过操作 socket 会从用户态进入内核态，而 send方法会将数据一路传到传输层。在识别到是 TCP协议后，会调用 tcp_sendmsg 方法。

在 tcp_sendmsg 中， 核心工作就是将待发送的数据组织按照先后顺序放入到发送缓冲区中， 然后根据实际情况（比如拥塞窗口等）判断是否要发数据。如果不发送数据，那么此时直接返回。

如果发送缓冲区满了，执行send()会怎样?

```
//socket创建的时候可以设置 阻塞式 or 非阻塞式
int s = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
```

- 如果是`sock_stream`阻塞式

当发送的缓冲区满了，如果此时还继续向缓冲区send数据，此时程序会在那里等着，直到释放出新的缓存空间。然后继续把数据考进去再返回。

- 如果是`sock_noblock`非阻塞式

程序就会**立刻返回**一个`EAGAIN`错误信息，意思是`Try again`，现在缓冲区满了，告诉你别等了，待会再试一次。

如果接收缓冲区为空，执行 recv()会怎么样?

```
//flag 默认为0，此时send为阻塞式发送，设为MSG_DONTWAIT 为非阻塞式发送
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
```

- 如果此时 socket 是阻塞的，那么程序会在那**干等**，直到接收缓冲区有数据，就会把数据从接收缓冲区拷贝到用户缓冲区，然后**返回**。
- 如果此时 socket 是非阻塞的，程序就会**立刻返回**一个 `EAGAIN` 错误信息。

如果socket缓冲区还有数据，执行close()会怎样?

> 正常情况下收发两个缓冲区都应该为空，如果发送、接收缓冲区**长时间非空**，说明有**数据堆积**，这往往是由于一些网络问题或用户应用层问题，导致数据没有正常处理。那么正常情况下，如果 `socket` 缓冲区**为空**，执行 `close`。就会触发四次挥手。

- **接收缓冲区有数据**
  - 如果接收缓冲区还有数据未读，会先把接收缓冲区的数据清空，然后给对端发一个RST。
  - 如果接收缓冲区是空的，那么就调用 `tcp_send_fin()` 开始进行四次挥手过程的第一次挥手

```
void tcp_close(struct sock *sk, long timeout)
{
  // 如果接收缓冲区有数据，那么清空数据
    while ((skb = __skb_dequeue(&sk->sk_receive_queue)) != NULL) {
        u32 len = TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq -
              tcp_hdr(skb)->fin;
        data_was_unread += len;
        __kfree_skb(skb);
    }

   if (data_was_unread) {
    // 如果接收缓冲区的数据被清空了，发 RST
        tcp_send_active_reset(sk, sk->sk_allocation);
     } else if (tcp_close_state(sk)) {
    // 正常四次挥手，  发 FIN
        tcp_send_fin(sk);
    }
    // 等待关闭
    sk_stream_wait_close(sk, timeout);
}
```

- **发送缓冲区有数据**
  - 内核会把发送缓冲区最后一个数据块拿出来。然后置为 FIN

```
void tcp_send_fin(struct sock *sk)
{
  // 获得发送缓冲区的最后一块数据
    struct sk_buff *skb, *tskb = tcp_write_queue_tail(sk);
    struct tcp_sock *tp = tcp_sk(sk);

  // 如果发送缓冲区还有数据
```

```
    if (tskb && (tcp_send_head(sk) || sk_under_memory_pressure(sk)))
{
        TCP_SKB_CB(tskb)->tcp_flags |= TCPHDR_FIN; // 把最后一块数据值
为 FIN
        TCP_SKB_CB(tskb)->end_seq++;
        tp->write_seq++;
    }  else {
    // 发送缓冲区没有数据，就造一个FIN包
  }
  // 发送数据
    __tcp_push_pending_frames(sk, tcp_current_mss(sk),
TCP_NAGLE_OFF);
}
```

- **注意**

  `socket` 缓冲区是个**先进先出**的队列，这种情况是指内核会等待TCP层安静把发送缓冲区数据都发完，最后再执行四次挥手的第一次挥手（FIN包）。

有一点需要注意的是，只有在**接收缓冲区为空的前提下**，我们才有可能走到 `tcp_send_fin()` 。而只有在进入了这个方法之后，我们才有可能考虑发送缓冲区是否为空的场景。

3 UDP部分

```
ssize_t send(int sock, const void *buf, size_t len, int flags); //flag置为
MSG_MORE，意为缓冲一定数量再发送,但一般不用这个设置
```

UDP socket 也是 socket，一个socket 就是会有收和发两个缓冲区，跟用什么协议关系不大。UDP是来一个数据包发一个数据包,缓冲区并没有“缓冲”这个作用。

五 Unix errno值

只要有一个Unix函数中有错误发生，全局变量就被置为一个指明该错误类型的正值，函数本身则通常返回-1.可以用err_sys查看error变量并输出相应的出错消息。例如当error的值为ETIMEDOUT时，将输出“Connection timed out”(连接超时)。

errno的值只在函数发生错误时设置，如果函数不返回错误，errno的值就没有定义。errno的所有正数错误都是常值，具有以"E"开头的全大写字母名字，并通常在<sys/errno.h>头文件中定义。值0不表示任何错误。